COMPILADORES

Principios, Técnicas y Herramientas

dragon book

CAPITULO 1

Introducción a la compilación

Los principios y técnicas de escritura de compiladores son tan amplios que las ideas encontradas en este libro se usarán muchas veces en la carrera de un científico de la computación. La escritura de compiladores comprende los lenguajes de programación, la arquitectura de computadores, la teoría de lenguajes, los algoritmos y la ingeniería de *software*. Por fortuna, con algunas técnicas básicas de escritura de compiladores se pueden construir traductores para una gran variedad de lenguajes y máquinas. En este capítulo, se introduce el tema de la compilación describiendo los componentes de un compilador, el entorno en el que trabajan los compiladores y algunas herramientas de software que facilitan la construcción de compiladores.

1.1 COMPILADORES

A grandes rasgos, un compilador es un programa que lee un programa escrito en un lenguaje, el lenguaje *fuente,* y lo traduce a un programa equivalente en otro lenguaje, el lenguaje *objeto* (véase Fig. 1.1). Como parte importante de este proceso de traducción, el compilador informa a su usuario de la presencia de errores en el programa fuente.

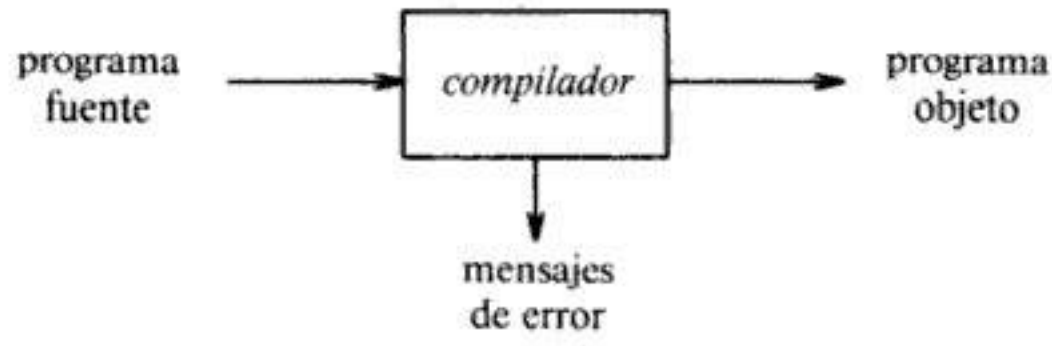


Fig. 1.1. Un compilador.

A primera vista, la diversidad de compiladores puede parecer abrumadora. Hay miles de lenguajes fuente, desde los lenguajes de programación tradicionales, como FORTRAN o Pascal, hasta los lenguajes especializados que han surgido virtualmente en todas las áreas de aplicación de la informática. Los lenguajes objeto son igualmente variados; un lenguaje objeto puede ser otro lenguaje de programación o

el lenguaje de máquina de cualquier computador entre un microprocesador y un supercomputador. Los compiladores a menudo se clasifican como de una pasada, de múltiples pasadas, de carga y ejecución, de depuración o de optimación, dependiendo de cómo hayan sido construidos o de qué función se supone que realizan. A pesar de esta aparente complejidad, las tareas básicas que debe realizar cualquier compilador son esencialmente las mismas. Al comprender tales tareas, se pueden construir compiladores para una gran diversidad de lenguajes fuente y máquinas objeto utilizando las mismas técnicas básicas.

Nuestro conocimiento sobre cómo organizar y escribir compiladores ha aumentado mucho desde que comenzaron a aparecer los primeros compiladores a principios de los años cincuenta. Es difícil dar una fecha exacta de la aparición del primer compilador, porque en un principio gran parte del trabajo de experimentación y aplicación se realizó de manera independiente por varios grupos. Gran parte de los primeros trabajos de compilación estaba relacionada con la traducción de fórmulas aritméticas a código de máquina.

En la década de 1950, se consideró a los compiladores como programas notablemente difíciles de escribir. El primer compilador de FORTRAN, por ejemplo, necesitó para su implantación 18 años de trabajo en grupo (Backus y otros [1975]). Desde entonces, se han descubierto técnicas sistemáticas para manejar muchas de las importantes tareas que surgen en la compilación. También se han desarrollado buenos lenguajes de implantación, entornos de programación y herramientas de software. Con estos avances, puede hacerse un compilador real incluso como proyecto de estudio en un curso de un semestre sobre diseño de compiladores.

Modelo de análisis y síntesis de la compilación

En la compilación hay dos partes: análisis y síntesis. La parte del análisis divide al programa fuente en sus elementos componentes y crea una representación intermedia del programa fuente. La parte de la síntesis construye el programa objeto deseado a partir de la representación intermedia. De las dos partes, la síntesis es la que requiere las técnicas más especializadas. En la sección 1.2 se examinará el análisis de manera informal y en la sección 1.3 se esbozará la forma de sintetizar el código objeto en un compilador estándar.

Durante el análisis, se determinan las operaciones que implica el programa fuente y se registran en una estructura jerárquica llamada árbol. A menudo, se usa una clase especial de árbol llamado árbol sintáctico, donde cada nodo representa una operación y los hijos de un nodo son los argumentos de la operación. Por ejemplo, en la figura 1.2 se muestra un árbol sintáctico para una proposición de asignación.

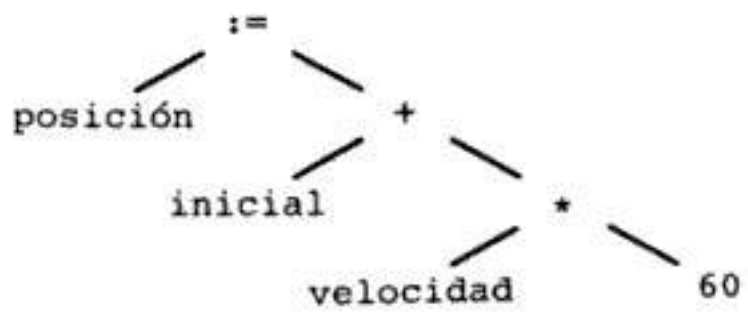


Fig. 1.2. Arbol sintáctico para `posición := inicial + velocidad * 60`.

Muchas herramientas de software que manipulan programas fuente realizan primero algún tipo de análisis. Algunos ejemplos de tales herramientas son:

1. *Editores de estructuras.* Un editor de estructuras toma como entrada una secuencia de órdenes para construir un programa fuente. El editor de estructuras no sólo realiza las funciones de creación y modificación de textos de un editor de textos ordinario, sino que también analiza el texto del programa, imponiendo al programa fuente una estructura jerárquica apropiada. De esa manera, el editor de estructuras puede realizar tareas adicionales útiles para la preparación de programas. Por ejemplo, puede comprobar si la entrada está formada correctamente, puede proporcionar palabras clave de manera automática (por ejemplo, cuando el usuario escribe `while`, el editor proporciona el correspondiente `do` y le recuerda al usuario que entre las dos palabras debe ir un condicional) y puede saltar desde un `begin` o un paréntesis izquierdo hasta su correspondiente `end` o paréntesis derecho. Además, la salida de tal editor suele ser similar a la salida de la fase de análisis de un compilador.
2. *Impresoras estéticas.* Una impresora estética analiza un programa y lo imprime de forma que la estructura del programa resulte claramente visible. Por ejemplo, los comentarios pueden aparecer con un tipo de letra especial, y las proposiciones pueden aparecer con una indentación proporcional a la profundidad de su anidamiento en la organización jerárquica de las proposiciones.
3. *Verificadores estáticos.* Un verificador estático lee un programa, lo analiza e intenta descubrir errores potenciales sin ejecutar el programa. La parte del análisis a menudo es similar a la que se encuentra en los compiladores de optimación del tipo estudiado en el capítulo 10. Así, un verificador estático puede detectar si hay partes de un programa que nunca se podrán ejecutar o si cierta variable se usa antes de ser definida. Además, puede detectar errores de lógica, como intentar utilizar una variable real como apuntador, empleando las técnicas de verificación de tipos que se analizan en el capítulo 6.
4. *Intérpretes.* En lugar de producir un programa objeto como resultado de una traducción, un intérprete realiza las operaciones que implica el programa fuente. Para una proposición de asignación, por ejemplo, un intérprete podría construir un árbol como el de la figura 1.2, y después efectuar las operaciones de los nodos conforme "recorre" el árbol. En la raíz descubriría que tiene que realizar una asignación, y llamaría a una rutina para evaluar la expresión de la derecha y después almacenaría el valor resultante en la localidad de memoria asociada con el identificador `posición`. En el hijo derecho de la raíz, la rutina descubriría que tiene que calcular la suma de dos expresiones. Se llamaría a sí misma de manera recursiva para calcular el valor de la expresión `velocidad*60`. Después sumaría ese valor al valor de la variable `inicial`.

Muchas veces los intérpretes se usan para ejecutar lenguajes de órdenes, pues cada operador que se ejecuta en un lenguaje de órdenes suele ser una invocación de una rutina compleja, como un editor o un compilador. Del mismo modo, algunos lenguajes de "muy alto nivel", como APL, normalmente son interpretados, porque hay muchas cosas sobre los datos, como el tamaño y la forma de las matrices, que no se pueden deducir en el momento de la compilación.

Tradicionalmente, se concibe un compilador como un programa que traduce un programa fuente, como FORTRAN, al lenguaje ensamblador o de máquina de algún computador. Sin embargo, hay lugares, al parecer, no relacionados donde la tecnología de los compiladores se usa con regularidad. La parte de análisis de cada uno de los siguientes ejemplos es parecida a la de un compilador convencional.

1. *Formadores de textos.* Un formador de textos toma como entrada una cadena de caracteres, la mayor parte de la cual es texto para componer, pero alguna incluye órdenes para indicar párrafos, figuras o estructuras matemáticas, como subíndices o superíndices. En la siguiente sección se menciona algo del análisis que realizan los formadores de textos.

2. *Compiladores de circuitos de silicio.* Un compilador de circuitos de silicio tiene un lenguaje fuente similar o idéntico a un lenguaje de programación convencional. Sin embargo, las variables del lenguaje no representan localidades de memoria, sino señales lógicas (0 ó 1) o grupos de señales en un circuito de conmutación. La salida es el diseño de un circuito en un lenguaje apropiado. Véanse Johnson [1983], Ullman [1984], o Trickey [1985] sobre un análisis de los compiladores de circuitos de silicio.

3. *Intérpretes de consultas.* Un intérprete de consultas traduce un predicado que contiene operadores relacionales y booleanos a órdenes para buscar en una base de datos registros que satisfagan ese predicado. (Véase Ullman [1982] o Date [1986].)

El contexto de un compilador

Además de un compilador, se pueden necesitar otros programas para crear un programa objeto ejecutable. Un programa fuente se puede dividir en módulos almacenados en archivos distintos. La tarea de reunir el programa fuente a menudo se confía a un programa distinto, llamado preprocesador. El preprocesador también puede expandir abreviaturas, llamadas macros, a proposiciones del lenguaje fuente.

La figura 1.3 muestra una "compilación" típica. El programa objeto creado por el compilador puede requerir procesamiento adicional antes de poderlo ejecutar. El compilador de la figura 1.3 crea código en lenguaje ensamblador el cual es traducido por un ensamblador a código de máquina y después se enlaza a algunas rutinas de biblioteca para producir el código que realmente se ejecute en la máquina.

En las dos secciones siguientes se estudiarán los componentes de un compilador; los programas restantes de la figura 1.3 se analizan en la sección 1.4.

1.2 ANALISIS DEL PROGRAMA FUENTE

En esta sección se introduce el análisis y se ilustra su uso en algunos lenguajes de formación de textos. Este tema se trata con más detalle en los capítulos 2 al 4 y en el 6. En la compilación, el análisis consta de tres fases:

1. *Análisis lineal*, en el que la cadena de caracteres que constituye el programa fuente se lee de izquierda a derecha y se agrupa en *componentes léxicos*, que son secuencias de caracteres que tienen un significado colectivo.

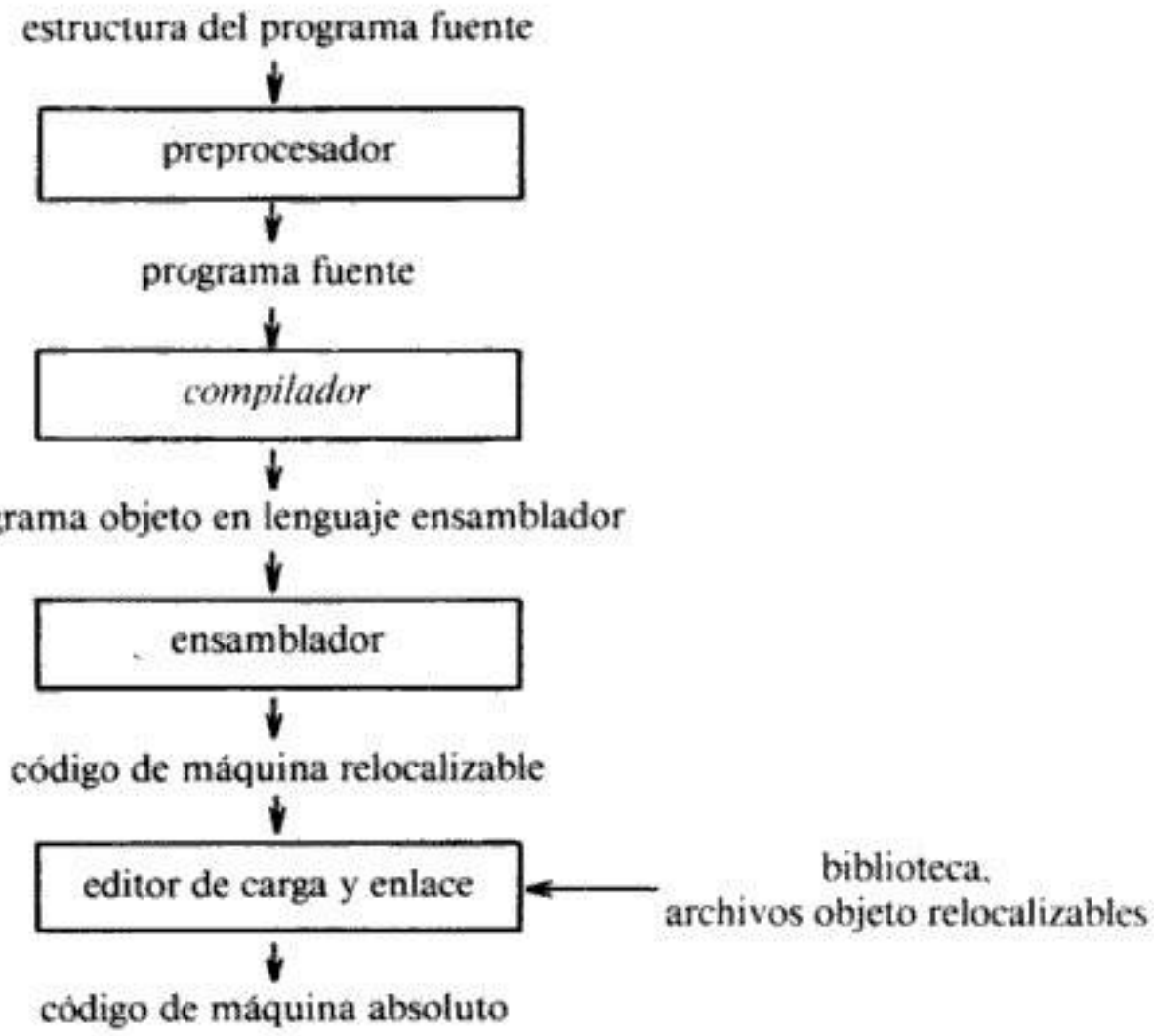


Fig. 1.3. Sistema para procesamiento de un lenguaje.

2. *Análisis jerárquico*, en el que los caracteres o los componentes léxicos se agrupan jerárquicamente en colecciones anidadas con un significado colectivo.
3. *Análisis semántico*, en el que se realizan ciertas revisiones para asegurar que los componentes de un programa se ajustan de un modo significativo.

Análisis léxico

En un compilador, el análisis lineal se llama *análisis léxico* o *exploración*. Por ejemplo, en el análisis léxico los caracteres de la proposición de asignación

```
posición := inicial + velocidad * 60
```

se agruparían en los componentes léxicos siguientes:

1. El identificador `posición`.
2. El símbolo de asignación `:=`.
3. El identificador `inicial`.
4. El signo de suma.
5. El identificador `velocidad`.
6. El signo de multiplicación.
7. El número `60`.

Los espacios en blanco que separan los caracteres de estos componentes léxicos normalmente se eliminan durante el análisis léxico.

Análisis sintáctico

El análisis jerárquico se denomina *análisis sintáctico.* Este implica agrupar los componentes léxicos del programa fuente en frases gramaticales que el compilador utiliza para sintetizar la salida. Por lo general, las frases gramaticales del programa fuente se representan mediante un árbol de análisis sintáctico como el que se ilustra en la figura 1.4.

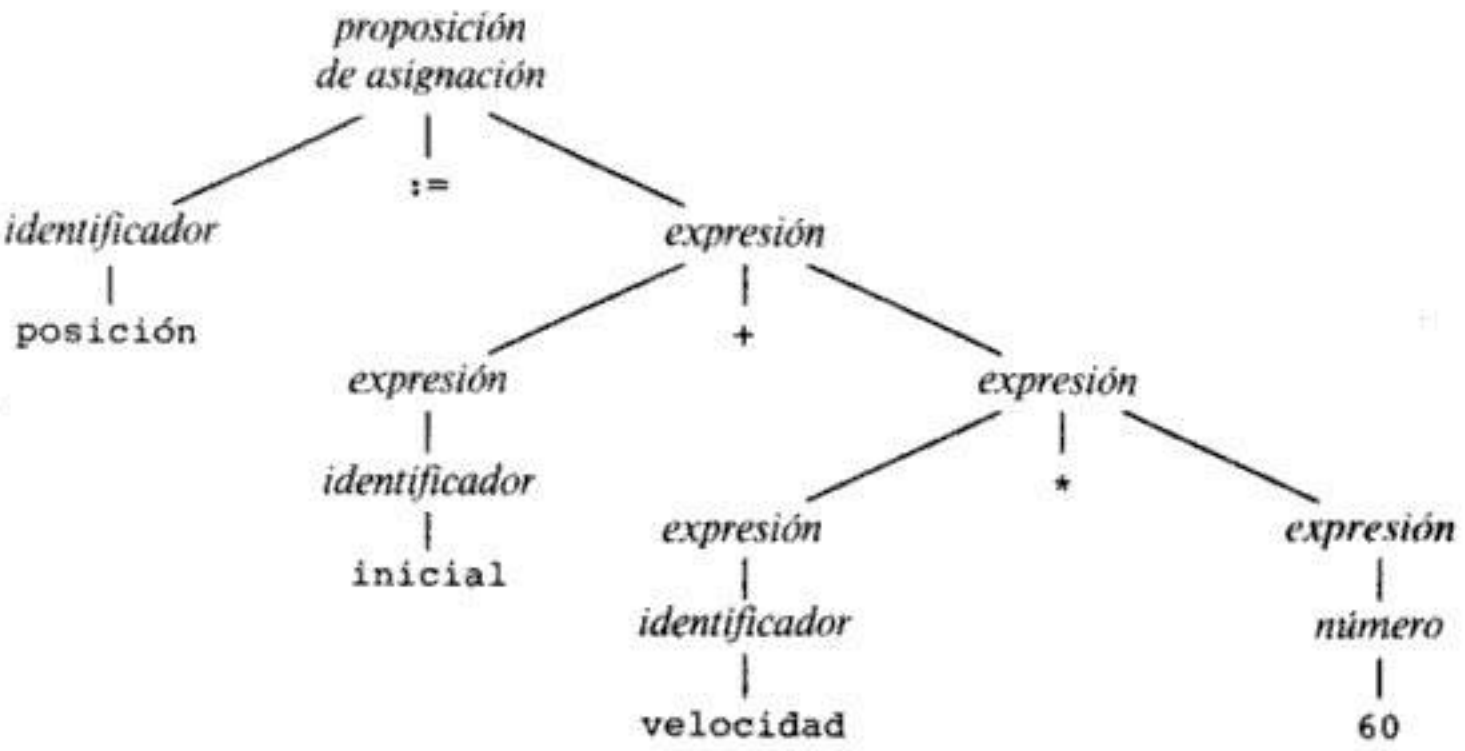


Fig. 1.4. Arbol de análisis sintáctico para `posición := inicial + velocidad * 60`.

En la expresión `inicial + velocidad * 60`, la frase `velocidad * 60` es una unidad lógica, porque las convenciones usuales de las expresiones aritméticas indican que la multiplicación se hace antes que la suma. Puesto que la expresión `inicial + velocidad` va seguida de un `*`, no se agrupa en una sola frase independiente en la figura 1.4.

La estructura jerárquica de un programa normalmente se expresa utilizando reglas recursivas. Por ejemplo, se pueden dar las siguientes reglas como parte de la definición de expresiones:

1. Cualquier *identificador* es una expresión.
2. Cualquier *número* es una expresión.
3. Si *expresión*$_1$ y *expresión*$_2$ son expresiones, entonces también lo son

   expresión$_1$ + *expresión*$_2$
   expresión$_1$ * *expresión*$_2$
   (*expresión*$_1$)

Las reglas (1) y (2) son reglas básicas (no recursivas), en tanto que la regla (3) define expresiones en función de operadores aplicados a otras expresiones. Así, por la regla (1), `inicial` y `velocidad` son expresiones. Por la regla (2), `60` es una expresión, mientras que por la regla (3), primero podemos inferir que `velocidad * 60` es una expresión, y finalmente, que `inicial + velocidad * 60` también es una expresión.

De manera similar, muchos lenguajes definen recursivamente las proposiciones mediante reglas como:

1. Si *identificador*$_1$ es un identificador y *expresión*$_2$ es una expresión, entonces

   identificador$_1$ **:=** *expresión*$_2$

   es una proposición.

2. Si *expresión*$_1$ es una expresión y *proposición*$_2$ es una proposición, entonces

   while (*expresión*$_1$) **do** *proposición*$_2$
   if (*expresión*$_1$) **then** *proposición*$_2$

   son proposiciones.

La división entre análisis léxico y análisis sintáctico es algo arbitraria. Generalmente se elige una división que simplifique la tarea completa del análisis. Un factor para determinar la división es si una construcción del lenguaje fuente es inherentemente recursiva o no. Las construcciones léxicas no requieren recursión, mientras que las construcciones sintácticas suelen requerirla. Las gramáticas independientes del contexto son una formalización de reglas recursivas que se pueden usar para guiar el análisis sintáctico. Estas gramáticas se dan a conocer en el capítulo 2 y se estudian ampliamente en el capítulo 4.

Por ejemplo, no se requiere recursión para reconocer los identificadores, que suelen ser cadenas de letras y dígitos que comienzan con una letra. Normalmente, se reconocen los identificadores por el simple examen del flujo de entrada, esperando hasta encontrar un carácter que no sea ni letra ni dígito, y agrupando después todas las letras y dígitos encontrados hasta ese punto en un componente léxico identificador. Los caracteres así agrupados se registran en una tabla, llamada tabla de símbolos, y se retiran de la entrada, para que pueda empezar el procesamiento del siguiente elemento léxico.

Por otra parte, esta clase de análisis léxico lineal no es suficientemente poderoso para analizar expresiones o proposiciones. Por ejemplo, no podemos emparejar de manera apropiada los paréntesis de las expresiones, o las palabras `begin` y `end` en proposiciones sin imponer alguna clase de estructura jerárquica o de anidamiento a la entrada.

El árbol de análisis sintáctico de la figura 1.4 describe la estructura sintáctica de la entrada. Una representación interna más común de esta estructura sintáctica es

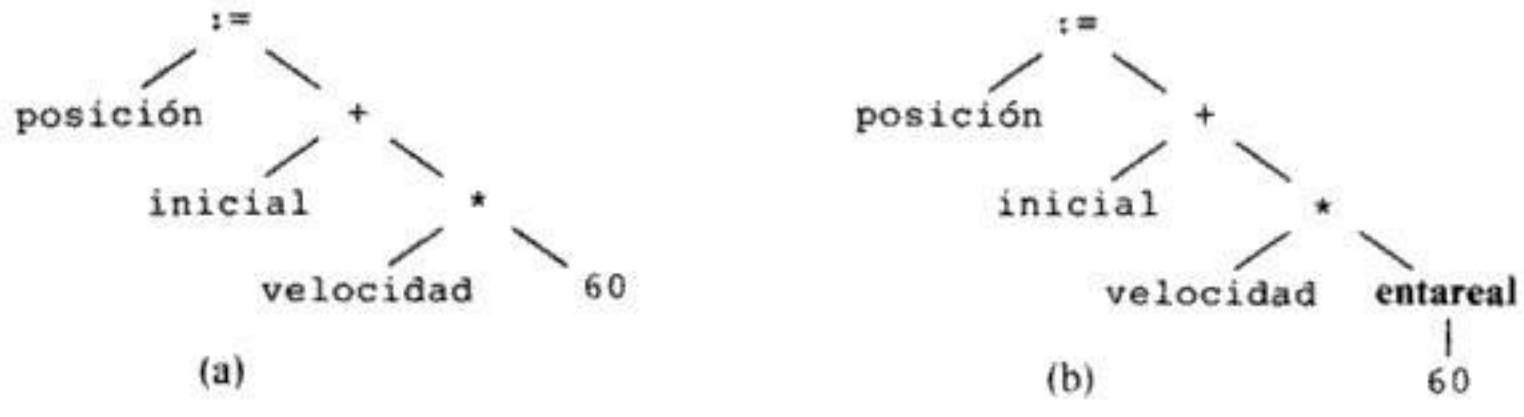


Fig. 1.5. El análisis semántico inserta una conversión de entero a real.

la que da el árbol sintáctico de la figura 1.5(a). Un árbol sintáctico es una representación compacta del árbol de análisis sintáctico en el que los operadores aparecen como los nodos interiores y los operandos de un operador son los hijos del nodo para ese operador. La construcción de árboles como el de la figura 1.5(a) se estudia en la sección 5.2. En el capítulo 2, y con más detalle en el capítulo 5, se estudiará el tema de la *traducción dirigida por la sintaxis*, en la que el compilador utiliza la estructura jerárquica de la entrada para ayudar a generar la salida.

Análisis semántico

La fase de análisis semántico revisa el programa fuente para tratar de encontrar errores semánticos y reúne la información sobre los tipos para la fase posterior de generación de código. En ella se utiliza la estructura jerárquica determinada por la fase de análisis sintáctico para identificar los operadores y operandos de expresiones y proposiciones.

Un componente importante del análisis semántico es la verificación de tipos. Aquí, el compilador verifica si cada operador tiene operandos permitidos por la especificación del lenguaje fuente. Por ejemplo, las definiciones de muchos lenguajes de programación requieren que el compilador indique un error cada vez que se use un número real como índice de una matriz. Sin embargo, la especificación del lenguaje puede permitir ciertas coerciones a los operandos, por ejemplo, cuando un operador aritmético binario se aplica a un número entero y a un número real. En este caso, el compilador puede necesitar convertir el número entero a real. La comprobación de tipos y el análisis semántico se estudian en el capítulo 6.

Ejemplo 1.1. Dentro de una máquina el patrón de bits que representa un entero es en general distinto del patrón de bits para un real, aun cuando el número entero y el real tengan el mismo valor. Por ejemplo, supóngase que todos los identificadores de la figura 1.5 se han declarado reales y que tan sólo `60` se supone entero. La verificación de tipos de la figura 1.5(a) revela que `*` se aplica a un real, `velocidad`, y a un entero, `60`. El tratamiento general es convertir el entero a real. Esto se ha logrado en la figura 1.5(b) creando un nodo extra para el operador **entareal** que de manera explícita convierte un entero a real. Por otra parte, como el operando de **entareal** es una constante, el compilador podría reemplazar la constante entera por una constante real equivalente. □

Análisis en formadores de textos

Es útil considerar que la entrada de un formador de textos especifica una jerarquía de *cajas*: regiones rectangulares que se van a llenar con algún patrón de bits, representando píxeles claros y oscuros para ser impresos por el dispositivo de salida.

Por ejemplo, el sistema TEX (Knuth [1984a]) considera su entrada de esta manera. Cada carácter que no sea parte de una orden representa una caja que contiene el patrón de bits de ese carácter con el tipo y tamaño apropiados. Los caracteres consecutivos no separados por "espacios en blanco" (espacios o caracteres de nueva línea) se agrupan en palabras, que consisten en una secuencia de cajas dispuestas horizontalmente, mostradas en el dibujo de la figura 1.6. El agrupamiento de caracteres

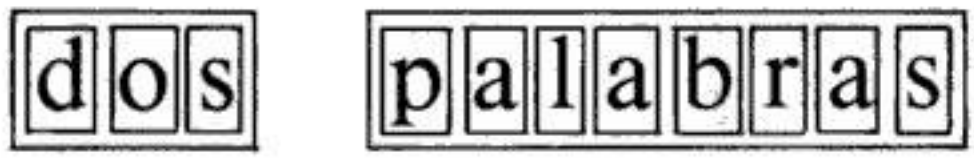


Fig. 1.6. Agrupamiento de caracteres y palabras en cajas.

en palabras (u órdenes) es el aspecto lineal o léxico del análisis en un formador de textos.

Las cajas en TEX se pueden construir a partir de cajas más pequeñas en combinaciones arbitrarias horizontales y verticales. Por ejemplo,

```
\hbox{ <lista de cajas> }
```

agrupa la lista de cajas yuxtaponiéndolas horizontalmente, mientras que el operador `\vbox` agrupa de manera similar una lista de cajas por yuxtaposición vertical. Así, si se indica en TEX

```
\hbox{\vbox{! 1} \vbox{@ 2}}
```

se obtiene la disposición de cajas que se muestra en la figura 1.7. Determinar la disposición jerárquica de las cajas implicadas en la entrada es parte del análisis sintáctico en TEX.

Fig. 1.7. Jerarquía de cajas en TEX.

Como otro ejemplo, el preprocesador EQN para matemáticas (Kernighan y Cherry [1975]), o el procesador matemático de TEX, construye expresiones matemáticas a partir de operadores como `sub` y `sup` para subíndices y superíndices. Si EQN encuentra un texto de entrada de la forma

CAJA `sub` *caja*

reduce el tamaño de *caja* y la une a *CAJA* cerca de la esquina inferior derecha, como se ilustra en la figura 1.8. De manera similar, el operador `sup` une *caja* a la esquina superior derecha.

Fig. 1.8. Construcción de la estructura de subíndice en texto matemático.

Estos operadores se pueden aplicar recursivamente, así que, por ejemplo, el texto en EQN

```
a sub {i sup 2}
```

da como resultado a_{i^2}. Agrupar los operadores `sub` y `sup` en componentes léxicos es parte del análisis léxico del texto en EQN. Sin embargo, se necesita la estructura sintáctica del texto para determinar el tamaño y la posición de las cajas.

1.3 LAS FASES DE UN COMPILADOR

Conceptualmente, un compilador opera en *fases*, cada una de las cuales transforma al programa fuente de una representación en otra. En la figura 1.9 se muestra una descomposición típica de un compilador. En la práctica, se pueden agrupar algunas fases, como se menciona en la sección 1.5, y las representaciones intermedias entre las fases agrupadas no necesitan ser construidas explícitamente.

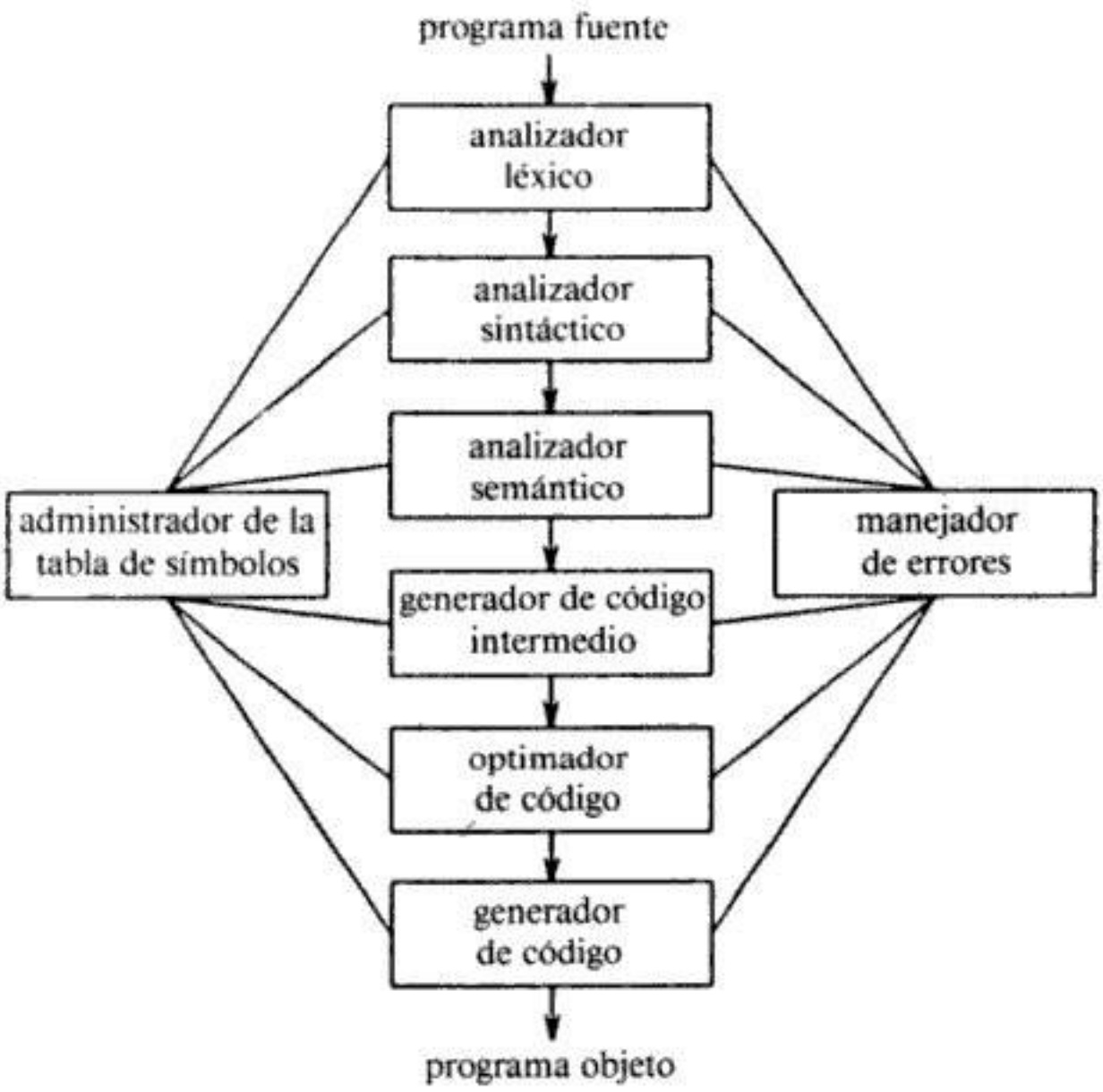


Fig. 1.9. Fases de un compilador.

Las tres primeras fases, que forman la mayor parte de la porción de análisis de un compilador, se introdujeron en la sección anterior. Otras dos actividades, la administración de la tabla de símbolos y el manejo de errores, se muestran en interacción con las seis fases de análisis léxico, análisis sintáctico, análisis semántico,

generación de código intermedio, optimación de código y generación de código. De modo informal, también se llamarán "fases" al administrador de la tabla de símbolos y al manejador de errores.

Administración de la tabla de símbolos

Una función esencial de un compilador es registrar los identificadores utilizados en el programa fuente y reunir información sobre los distintos atributos de cada identificador. Estos atributos pueden proporcionar información sobre la memoria asignada a un identificador, su tipo, su ámbito (la parte del programa donde tiene validez) y, en el caso de nombres de procedimientos, cosas como el número y tipos de sus argumentos, el método de pasar cada argumento (por ejemplo, por referencia) y el tipo que devuelve, si lo hay.

Una *tabla de símbolos* es una estructura de datos que contiene un registro por cada identificador, con los campos para los atributos del identificador. La estructura de datos permite encontrar rápidamente el registro de cada identificador y almacenar o consultar rápidamente datos de ese registro. Las tablas de símbolos se estudian en los capítulos 2 y 7.

Cuando el analizador léxico detecta un identificador en el programa fuente, el identificador se introduce en la tabla de símbolos. Sin embargo, normalmente los atributos de un identificador no se pueden determinar durante el análisis léxico. Por ejemplo, en una declaración en Pascal como

```
var posición, inicial, velocidad : real ;
```

el tipo real no se conoce cuando el analizador léxico encuentra `posición`, `inicial`, y `velocidad`.

Las fases restantes introducen información sobre los identificadores en la tabla de símbolos y después la utilizan de varias formas. Por ejemplo, cuando se está haciendo el análisis semántico y la generación de código intermedio, se necesita saber cuáles son los tipos de los identificadores, para poder comprobar si el programa fuente los usa de una forma válida y así poder generar las operaciones apropiadas con ellos. El generador de código, por lo general, introduce y utiliza información detallada sobre la memoria asignada a los identificadores.

Detección e información de errores

Cada fase puede encontrar errores. Sin embargo, después de detectar un error, cada fase debe tratar de alguna forma ese error, para poder continuar la compilación, permitiendo la detección de más errores en el programa fuente. Un compilador que se detiene cuando encuentra el primer error, no resulta tan útil como debiera.

Las fases de análisis sintáctico y semántico por lo general manejan una gran porción de los errores detectables por el compilador. La fase léxica puede detectar errores donde los caracteres restantes de la entrada no forman ningún componente léxico del lenguaje. Los errores donde la cadena de componentes léxicos violan las reglas de estructura (sintaxis) del lenguaje son determinados por la fase de análisis sintáctico. Durante el análisis semántico el compilador intenta detectar construcciones que tengan la estructura sintáctica correcta, pero que no tengan significado para la operación implicada, por ejemplo, si se intenta sumar dos identificadores, uno de

los cuales es el nombre de una matriz, y el otro, el nombre de un procedimiento. En cada parte del libro dedicada a cada fase se estudia el manejo de errores de esa fase.

Las fases de análisis

Conforme avanza la traducción, la representación interna del programa fuente que tiene el compilador se modifica. Para ilustrar esas representaciones, considérese la traducción de la proposición

```
posición := inicial + velocidad * 60          (1.1)
```

La figura 1.10 muestra la representación de esta proposición después de cada fase.

La fase de análisis léxico lee los caracteres en el programa fuente y los agrupa en una cadena de componentes léxicos en los que cada componente representa una secuencia lógicamente coherente de caracteres, como un identificador, una palabra clave (`if`, `while`, etcétera), un carácter de puntuación, o un operador de varios caracteres, como `:=`. La secuencia de caracteres que forma un componente léxico se denomina *lexema* del componente.

A ciertos componentes léxicos se les agregará un "valor léxico". Así, cuando se encuentra un identificador como `velocidad`, el analizador léxico no sólo genera un componente léxico, por ejemplo, **id**, sino que también introduce el lexema `velocidad` en la tabla de símbolos, si aún no estaba allí. El valor léxico asociado con esta aparición de **id** señala la entrada de la tabla de símbolos correspondiente a `velocidad`.

En esta sección, se usarán $\mathbf{id}_1$, $\mathbf{id}_2$ e $\mathbf{id}_3$ para `posición`, `inicial` y `velocidad`, respectivamente, para enfatizar que la representación interna de un identificador es diferente de la secuencia de caracteres que forman el identificador. Por tanto, la representación de (1.1) después del análisis léxico queda sugerida por:

$$\mathbf{id}_1 \ := \ \mathbf{id}_2 \ + \ \mathbf{id}_3 \ * \ 60 \qquad (1.2)$$

Se deberían construir componentes léxicos para el operador de varios caracteres `:=` y el número `60`, para reflejar su representación interna, pero esto se deja para el capítulo 2. El análisis léxico se trata en detalle en el capítulo 3.

En la sección 1.2 ya se introdujeron las fases segunda y tercera: los análisis sintáctico y semántico. El análisis sintáctico impone una estructura jerárquica a la cadena de componentes léxicos, que se representará por medio de árboles sintácticos, como se muestra en la figura 1.11(a). Una estructura de datos típica para el árbol se muestra en la figura 1.11(b), en la que un nodo interior es un registro con un campo para el operador y dos campos que contienen apuntadores a los registros de los hijos izquierdo y derecho. Una hoja es un registro con dos o más campos, uno para identificar al componente léxico de la hoja, y los otros para registrar información sobre el componente léxico. Se puede tener información adicional sobre las construcciones del lenguaje añadiendo más campos a los registros de los nodos. En los capítulos 4 y 6 se estudian el análisis sintáctico y el análisis semántico, respectivamente.

Generación de código intermedio

Después de los análisis sintáctico y semántico, algunos compiladores generan una representación intermedia explícita del programa fuente. Se puede considerar esta

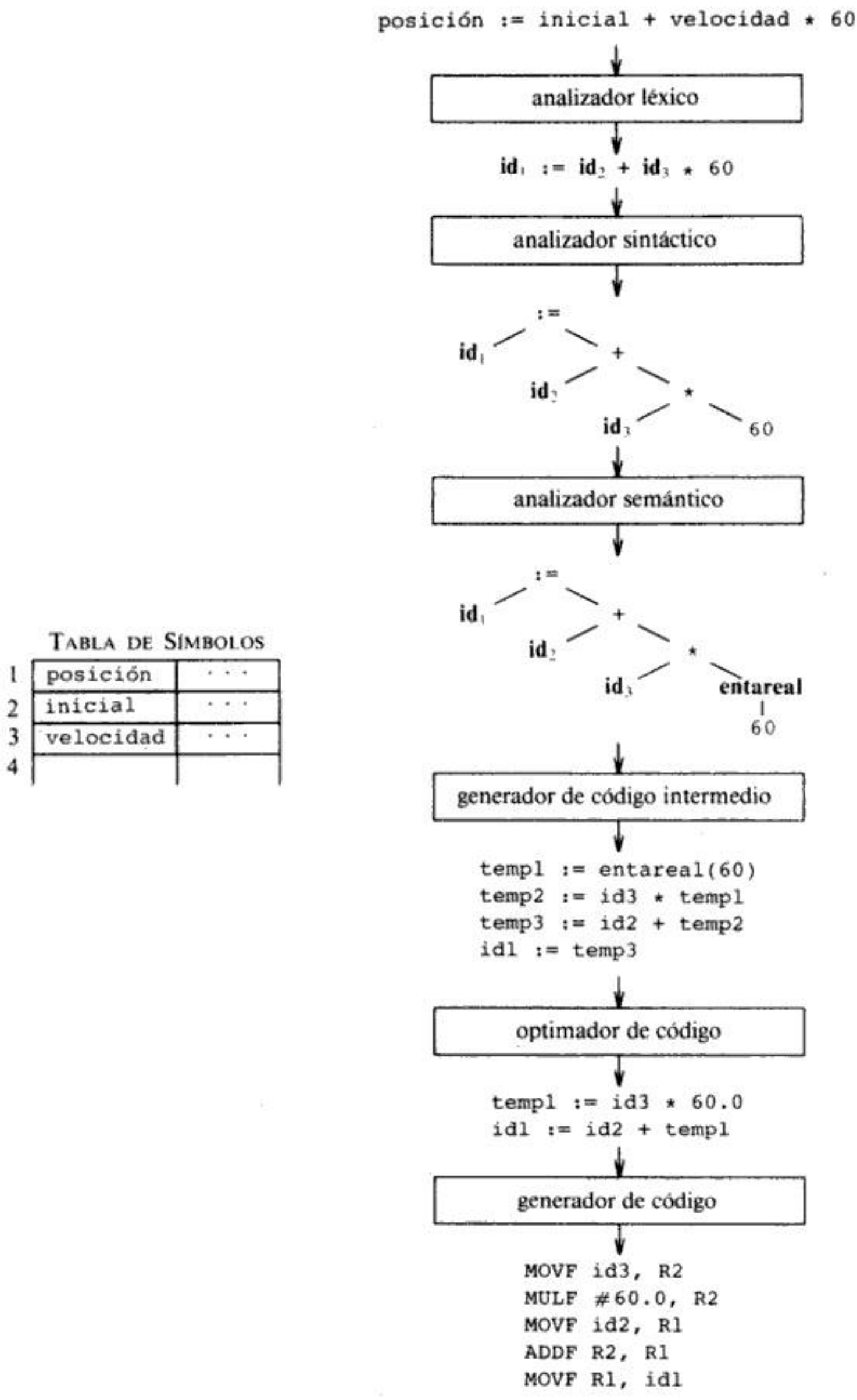


Fig. 1.10. Traducción de una proposición.

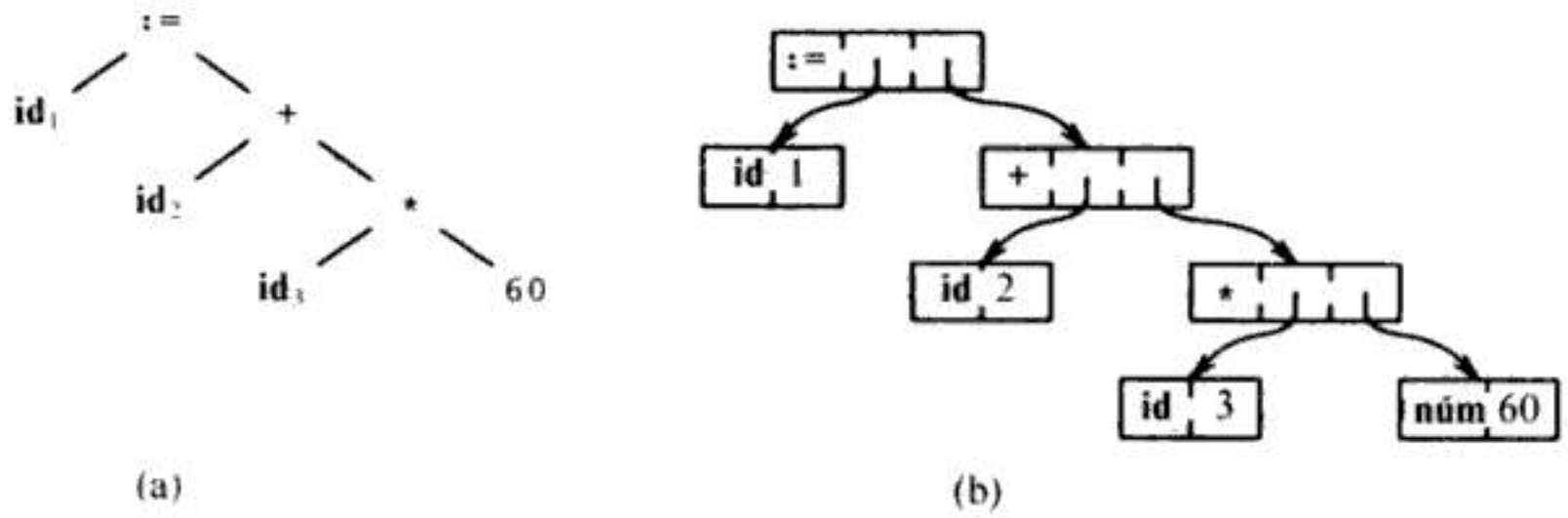


Fig. 1.11. La estructura de datos en (b) corresponde al árbol en (a).

representación intermedia como un programa para una máquina abstracta. Esta representación intermedia debe tener dos propiedades importantes; debe ser fácil de producir y fácil de traducir al programa objeto.

La representación intermedia puede tener diversas formas. En el capítulo 8 se trata una forma intermedia llamada "código de tres direcciones", que es como el lenguaje ensamblador para una máquina en la que cada posición de memoria puede actuar como un registro. El código de tres direcciones consiste en una secuencia de instrucciones, cada una de las cuales tiene como máximo tres operandos. El programa fuente de (1.1) puede aparecer en código de tres direcciones como

```
temp1 := entareal(60)
temp2 := id3 * temp1                    (1.3)
temp3 := id2 + temp2
id1 := temp3
```

Esta representación intermedia tiene varias propiedades. Primera, cada instrucción de tres direcciones tiene a lo sumo un operador, además de la asignación. Por tanto, cuando se generan esas instrucciones, el compilador tiene que decidir el orden en que deben efectuarse las operaciones; la multiplicación precede a la adición en el programa fuente de (1.1). Segunda, el compilador debe generar un nombre temporal para guardar los valores calculados por cada instrucción. Tercera, algunas instrucciones de "tres direcciones" tienen menos de tres operandos, por ejemplo, la primera y la última instrucciones de (1.3).

En el capítulo 8 se tratan las principales representaciones intermedias empleadas en los compiladores. En general, estas representaciones deben hacer algo más que calcular expresiones; también deben manejar construcciones de flujo de control y llamadas a procedimientos. Los capítulos 5 y 8 presentan algoritmos para generar código intermedio para construcciones típicas de lenguajes de programación.

Optimación de código

La fase de optimación de código trata de mejorar el código intermedio, de modo que resulte un código de máquina más rápido de ejecutar. Algunas optimaciones son triviales. Por ejemplo, un algoritmo natural genera el código intermedio (1.3) utilizando una instrucción para cada operador de la representación de árbol después

del análisis semántico, aunque hay una forma mejor de realizar los mismos cálculos usando las dos instrucciones

```
temp1 := id3 * 60.0
id1 := id2 + temp1                    (1.4)
```

Este sencillo algoritmo no tiene nada de malo, puesto que el problema se puede solucionar en la fase de optimación de código. Esto es, el compilador puede deducir que la conversión de `60` de entero a real se puede hacer de una vez por todas en el momento de la compilación, de modo que la operación `entareal` se puede eliminar. Además, `temp3` se usa sólo una vez, para transmitir su valor a `id1`. Entonces resulta seguro sustituir `id1` por `temp3`, a partir de lo cual la última proposición de (1.3) no se necesita y se obtiene el código de (1.4).

Hay mucha variación en la cantidad de optimación de código que ejecutan los distintos compiladores. En los que hacen mucha optimación, llamados "compiladores optimadores", una parte significativa del tiempo del compilador se ocupa en esta fase. Sin embargo, hay optimaciones sencillas que mejoran sensiblemente el tiempo de ejecución del programa objeto sin retardar demasiado la compilación. En el capítulo 9 se estudian muchos de estos aspectos, mientras que en el capítulo 10 se da la tecnología utilizada por los compiladores optimadores más potentes.

Generación de código

La fase final de un compilador es la generación de código objeto, que por lo general consiste en código de máquina relocalizable o código ensamblador. Las posiciones de memoria se seleccionan para cada una de las variables usadas por el programa. Después, cada una de las instrucciones intermedias se traduce a una secuencia de instrucciones de máquina que ejecuta la misma tarea. Un aspecto decisivo es la asignación de variables a registros.

Por ejemplo, utilizando los registros 1 y 2, la traducción del código de (1.4) podría convertirse en

```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1                          (1.5)
ADDF R2, R1
MOVF R1, id1
```

El primero y segundo operandos de cada instrucción especifican una fuente y un destino, respectivamente. La `F` de cada instrucción indica que las instrucciones trabajan con números de punto flotante. Este código traslada el contenido de la dirección[1] `id3` al registro 2, después lo multiplica por la constante real `60.0`. El signo `#` significa que `60.0` se trata como una constante. La tercera instrucción pasa `id2` al registro 1. La cuarta instrucción le suma el valor previamente calculado en el re-

---

[1] Se ha evitado el importante aspecto de la asignación de memoria para los identificadores del programa fuente. Como se verá en el capítulo 7, la organización de memoria en el tiempo de ejecución depende del lenguaje que se esté compilando. Las decisiones de asignación de memoria se hacen durante la generación del código intermedio o durante la generación de código.

gistro 2. Por último, el valor del registro 1 se pasa a la dirección de `id1`, de modo que el código aplica la asignación de la figura 1.10. En el capítulo 9 se trata la generación de código.

1.4 PROGRAMAS DE SISTEMAS RELACIONADOS CON UN COMPILADOR

Como se vio en la figura 1.3, la entrada para un compilador puede producirse por uno o varios preprocesadores, y puede necesitarse otro procesamiento de la salida que produce el compilador antes de obtener un código de máquina ejecutable. En esta sección se analiza el contexto en el que suele funcionar un compilador.

Preprocesadores

Los preprocesadores producen la entrada para un compilador, y pueden realizar las funciones siguientes:

1. *Procesamiento de macros.* Un preprocesador puede permitir a un usuario definir macros, que son abreviaturas de construcciones más grandes.
2. *Inclusión de archivos.* Un preprocesador puede insertar archivos de encabezamiento en el texto del programa. Por ejemplo, el preprocesador de C hace que el contenido del archivo `<global.h>` reemplace a la proposición `#include <global.h>` cuando procesa un archivo que contenga a esa proposición.
3. *Preprocesadores "racionales".* Estos preprocesadores enriquecen los lenguajes antiguos con recursos más modernos de flujo de control y de estructuras de datos. Por ejemplo, un preprocesador de este tipo podría proporcionar al usuario macros incorporadas para construcciones, como proposiciones `while` o `if`, en un lenguaje de programación que no las tenga.
4. *Extensiones a lenguajes.* Estos procesadores tratan de crear posibilidades al lenguaje que equivalen a macros incorporadas. Por ejemplo, el lenguaje Equel (Stonebraker y otros [1976]) es un lenguaje de consulta de base de datos integrado en C. El preprocesador considera las proposiciones que empiezan con `##` como proposiciones de acceso a la base de datos, sin relación con C, y se traducen a llamadas de procedimiento a rutinas que realizan el acceso a la base de datos.

Los procesadores de macros tratan dos clases de proposiciones: definición de macros y uso de macros. Las definiciones normalmente se indican con algún carácter exclusivo o palabra clave, como `define` o `macro`. Constan de un nombre para la macro que se está definiendo y de un *cuerpo*, que constituye su definición. A menudo, los procesadores de macros admiten *parámetros formales* en su definición, esto es, símbolos que se reemplazarán por valores (en este contexto, un "valor" es una cadena de caracteres). El uso de una macro consiste en dar nombre a la macro y proporcionar *parámetros reales*, es decir, valores para sus parámetros formales. El procesador de macros sustituye los parámetros reales por los parámetros formales

del cuerpo de la macro; después, el cuerpo transformado reemplaza el uso de la propia macro.

Ejemplo 1.2. El sistema de composición tipográfica TEX mencionado en la sección 1.2 contiene un recurso de macros general. Las definiciones de macros son de la forma

```
\define <nombre de la macro> <plantilla> {<cuerpo>}
```

El nombre de una macro es cualquier cadena de letras precedida por una diagonal invertida. La plantilla es cualquier cadena de caracteres en donde las cadenas de la forma `#1, #2, . . . , #9` se consideran parámetros formales. Estos símbolos también pueden aparecer en el cuerpo las veces que se quiera. Por ejemplo, la siguiente macro define una cita del *Journal of the ACM*.

```
\define\JACM #1;#2;#3.
    {{\sl J. ACM} {\bf #1}:#2, págs. #3.}
```

El nombre de la macro es JACM, y la plantilla es "`#1;#2;#3.`"; los símbolos de punto y coma separan los parámetros y después del último parámetro se pone un punto. Un uso de esta macro debe tomar la forma de la plantilla, excepto que se pueden sustituir cadenas arbitrarias por los parámetros formales[2]. Así, se puede escribir

```
\JACM 17;4;715-728.
```

y se espera que aparezca

J. ACM **17**:4, págs. 715-728.

La parte del cuerpo {`\sl J. ACM`} pide "*J. ACM*" en *cursiva* (*sl* es por *slanted*, "inclinado" en inglés). La expresión {`bf #1`} indica que el primer parámetro real se escribirá en **negritas** (*bf* es por *boldface*, "negrita" en inglés); este parámetro es el número de volumen.

TEX admite cualquier puntuación o cadena de texto para separar el volumen, el número del ejemplar y los números de página de la definición de la macro \JACM. Incluso se podría haber prescindido totalmente de la puntuación, en cuyo caso TEX tomaría cada parámetro real como un solo carácter o una cadena encerrada entre { } □

Ensambladores

Algunos compiladores producen código ensamblador, como en el caso (1.5), que se pasa a un ensamblador para su procesamiento. Otros compiladores realizan el trabajo del ensamblador, produciendo código de máquina relocalizable que se puede

---

[2] Bueno, cadenas casi arbitrarias, puesto que sólo se hace un simple análisis léxico de izquierda a derecha del uso de la macro, y tan pronto como se encuentre un símbolo que concuerde con el texto que sigue a un símbolo `#i` de la plantilla, se considera que la cadena precedente concuerda con `#i`. Por tanto, si se intentara sustituir `ab;cd` por `#1`, resultaría que sólo `ab` concuerda con `#1` y que `cd` concuerda con `#2`.

pasar directamente al editor de carga y enlace. Se supone que el lector tiene cierta familiaridad sobre cómo es un lenguaje ensamblador y qué hace el ensamblador; aquí se revisará la relación entre el código ensamblador y el código de máquina.

El *código ensamblador* es una versión mnemotécnica del código de máquina, donde se usan nombres en lugar de códigos binarios para operaciones, y también se usan nombres para las direcciones de memoria. Una secuencia típica de instrucciones en ensamblador puede ser

```
MOV a, R1
ADD #2, R1          (1.6)
MOV R1, b
```

Este código pasa el contenido de la dirección a al registro 1; después le suma la constante 2, tratando al contenido del registro 1 como un número de punto fijo, y por último almacena el resultado en la posición de memoria que representa b. De ese modo, calcula `b:=a + 2`.

Es común que los lenguajes ensambladores tengan recursos para manejar macros que son similares a las consideradas antes para los preprocesadores de macros.

Ensamblado de dos pasadas

La forma más simple de un ensamblador hace dos pasadas sobre la entrada, en donde una *pasada* consiste en leer una vez un archivo de entrada. En la primera pasada, se encuentran todos los identificadores que denoten posiciones de memoria y se almacenan en una tabla de símbolos (distinta de la del compilador). Cuando se encuentran por primera vez los identificadores, se les asignan posiciones de memoria, de modo que después de leer (1.6), por ejemplo, la tabla de símbolos contendría las entradas que aparecen en la figura 1.12. En esa figura, se supone que se reserva una palabra, que consta de cuatro *bytes*, para cada identificador, y que las direcciones se asignan empezando a partir del *byte* 0.

IDENTIFICADOR	DIRECCIÓN
a	0
b	4

Fig. 1.12. Tabla de símbolos de un ensamblador con los identificadores de (1.6).

En la segunda pasada, el ensamblador examina el archivo de entrada de nuevo. Esta vez traduce cada código de operación a la secuencia de *bits* que representa esa operación en lenguaje de máquina, y traduce cada identificador que representa una posición de memoria a la dirección dada por ese identificador en la tabla de símbolos.

El resultado de la segunda pasada normalmente es código de máquina *relocalizable*, lo cual significa que puede cargarse empezando en cualquier posición L de la memoria; es decir, si se suma L a todas las direcciones del código, entonces todas las referencias serán correctas. Por tanto, la salida del ensamblador debe distinguir aquellas partes de instrucciones que se refieren a direcciones que se pueden relocalizar.

Ejemplo 1.3. El siguiente es un código de máquina hipotético al que se pueden traducir las instrucciones en ensamblador (1.6).

```
0001 01 00 00000000 *
0011 01 10 00000010                                    (1.7)
0010 01 00 00000100 *
```

Se concibe una pequeña palabra de instrucción, en la que los cuatro primeros *bits* son el código de la instrucción, donde `0001`, `0010` y `0011` representan las instrucciones LOAD, STORE y ADD, respectivamente. LOAD y STORE significan trasladar de memoria a un registro y viceversa. Los dos *bits* siguientes designan un registro y `01` se refiere al registro 1 de cada una de las tres instrucciones anteriores. Los dos *bits* siguientes representan un marcador, donde `00` es el modo de direccionamiento ordinario, y los últimos ocho *bits* se refieren a una dirección de memoria. El marcador `10` es el modo "inmediato", donde los últimos ocho *bits* se toman literalmente como el operando. Este modo aparece en la segunda instrucción de (1.7).

En (1.7) también se ve un * asociado con la primera y la tercera instrucciones. Este * representa el *bit de relocalización* que se asocia con cada operando en código de máquina relocalizable. Supóngase que el espacio de direcciones que contiene los datos se va a cargar empezando en la posición L. La presencia del * significa que se debe sumar L a la dirección de la instrucción. Por tanto, si L = `00001111`, esto es, 15, entonces `a` y `b` estarían en las posiciones 15 y 19, respectivamente, y las instrucciones de (1.7) aparecerían como

```
0001 01 00 00001111
0011 01 10 00000010                                    (1.8)
0010 01 00 00010011
```

en código de máquina *absoluto* o no relocalizable. Nótese que no hay ningún * asociado con la segunda instrucción de (1.7), de modo que L no se sumó a su dirección en (1.8), lo cual es correcto, porque los *bits* representan la constante 2 y no la posición 2. □

Cargadores y editores de enlace

Por lo general, un programa llamado *cargador* realiza las dos funciones de carga y edición de enlaces. El proceso de carga consiste en tomar el código de máquina relocalizable, modificar las direcciones relocalizables, como se indica en el ejemplo 1.3, y ubicar las instrucciones y los datos modificados en las posiciones apropiadas de la memoria.

El editor de enlace permite formar un sólo programa a partir de varios archivos de código de máquina relocalizable. Estos archivos pueden haber sido el resultado de varias compilaciones distintas, y uno o varios de ellos pueden ser archivos de biblioteca de rutinas proporcionadas por el sistema y disponibles para cualquier programa que las necesite.

Si los archivos se van a usar juntos de manera útil, puede haber algunas referencias *externas*, en las que el código de un archivo hace referencia a una posición de otro archivo. Esta referencia puede ser a una posición de datos definida en un ar-

chivo y utilizada en otro, o puede ser el punto de entrada de un procedimiento que aparece en el código de un archivo y se llama desde otro. El archivo con el código de máquina relocalizable debe conservar la información de la tabla de símbolos para cada posición de datos o etiqueta de instrucción a la que se hace referencia externamente. Si no se sabe por anticipado a qué se va a hacer referencia, es preciso incluir completa la tabla de símbolos del emsamblador como parte del código de máquina relocalizable.

Por ejemplo, el código de (1.7) iría precedido de

```
a  0
b  4
```

Si un archivo cargado con (1.7) hiciera referencia a b, entonces esa referencia se reemplazaría por 4 más el desplazamiento con el que se localizaron las posiciones del archivo (1.7).

1.5 EL AGRUPAMIENTO DE LAS FASES

El estudio de las fases de la sección 1.3 trata la organización lógica de un compilador. En una implantación, a menudo se agrupan las actividades en dos o más fases.

Etapa inicial y etapa final

Con frecuencia, las fases se agrupan en una *etapa inicial* y una *etapa final*. La etapa inicial comprende aquellas fases, o partes de fases, que dependen principalmente del lenguaje fuente y que son en gran parte independientes de la máquina objeto. Ahí normalmente se incluyen los análisis léxico y sintáctico, la creación de la tabla de símbolos, el análisis semántico y la generación de código intermedio. La etapa inicial también puede hacer cierta optimación de código. La etapa inicial incluye, además, el manejo de errores correspondiente a cada una de esas fases.

La etapa final incluye aquellas partes del compilador que dependen de la máquina objeto y, en general, esas partes no dependen del lenguaje fuente, sino sólo del lenguaje intermedio. En la etapa final, se encuentran aspectos de la fase de optimación de código, además de la generación de código, junto con el manejo de errores necesario y las operaciones con la tabla de símbolos.

Se ha convertido en rutina el tomar la etapa inicial de un compilador y rehacer su etapa final asociada para producir un compilador para el mismo lenguaje fuente en una máquina distinta. Si la etapa final se diseña con cuidado, incluso puede no ser necesario rediseñarla demasiado; este tema se estudia en el capítulo 9. También resulta tentador compilar varios lenguajes distintos en el mismo lenguaje intermedio y usar una etapa final común para las distintas etapas iniciales, obteniéndose así varios compiladores para una máquina. Sin embargo, dadas las sutiles diferencias en los puntos de vista de los distintos lenguajes, sólo se ha obtenido un éxito limitado en ese aspecto.

Pasadas

Normalmente se aplican varias fases de la compilación en una sola *pasada*, que consiste en la lectura de un archivo de entrada y en la escritura de un archivo de salida.

En la práctica, hay muchas formas de agrupar en pasadas las fases de un compilador, así que es preferible organizar el análisis de la compilación por las fases, en lugar de por las pasadas. En el capítulo 12 se analizan algunos compiladores representativos y se menciona la forma en que estructuran las fases en pasadas.

Como ya se señaló, es común agrupar varias fases en una pasada, y entrelazar la actividad de estas fases durante la pasada. Por ejemplo, el análisis léxico, el análisis sintáctico, el análisis semántico y la generación de código intermedio pueden agruparse en una pasada. En ese caso, la cadena de componentes léxicos después del análisis léxico puede traducirse directamente a código intermedio. Con más detalle, el analizador sintáctico puede considerarse como el "encargado" del control. Este intenta descubrir la estructura gramatical de los componentes léxicos observados; obtiene los componentes léxicos cuando los necesita, llamando al analizador léxico para que le proporcione el siguiente componente léxico. A medida que se descubre la estructura gramatical, el analizador sintáctico llama al generador de código intermedio para que haga el análisis semántico y genere una parte del código. En el capítulo 2 se presenta un compilador organizado de esta forma.

Reducción del número de pasadas

Es deseable tener relativamente pocas pasadas, dado que la lectura y escritura de archivos intermedios lleva tiempo. Además, si se agrupan varias fases dentro de una pasada, puede ser necesario tener que mantener el programa completo en memoria, porque una fase puede necesitar información en un orden distinto al que produce una fase previa. La forma interna del programa puede ser considerablemente mayor que el programa fuente o el programa objeto, de modo que este espacio no es un tema trivial.

Para algunas fases, el agrupamiento en una pasada presenta pocos problemas. Por ejemplo, como se mencionó antes, la interfaz entre los analizadores léxico y sintáctico a menudo puede limitarse a un solo componente léxico. Por otra parte, muchas veces resulta muy difícil generar código hasta que se haya generado por completo la representación intermedia. Por ejemplo, lenguajes como PL/I y ALGOL 68 permiten usar las variables antes de declararlas. No se puede generar el código objeto para una construcción si no se conocen los tipos de las variables implicadas en esa construcción. De manera similar, la mayoría de los lenguajes admiten construcciones `goto` que saltan hacia adelante en el código. No se puede determinar la dirección objeto de dichos saltos hasta haber visto el código fuente implicado y haber generado código objeto para él.

En algunos casos, es posible dejar un segmento en blanco para la información que falta, y llenar la ranura cuando la información esté disponible. En particular, la generación de codigo intermedio y de código objeto a menudo se pueden fusionar en una sola pasada utilizando una técnica llamada "relleno de retroceso" *(backpatching)*. Aunque no se pueden explicar todos los detalles hasta que en el capítulo 8 se estudie la generación de código intermedio, se puede ilustrar el relleno de retroceso partiendo de un ensamblador. Recuérdese que en la sección anterior se analizó un ensamblador de dos pasadas, en el que la primera pasada descubría todos los identificadores que representaban posiciones de memoria y deducía sus direcciones al

descubrirlas. Después, en una segunda pasada sustituía las direcciones por identificadores.

Se puede combinar la acción de las pasadas como sigue. Al encontrar una proposición en ensamblador que sea una referencia hacia adelante, por ejemplo,

```
GOTO destino
```

se genera la estructura de una instrucción, con el código de operación de máquina para GOTO y se dejan espacios en blanco para la dirección. Todas las instrucciones con espacios en blanco para la dirección de `destino` se guardan en una lista asociada con la entrada de `destino` en la tabla de símbolos. Los espacios se llenan cuando por fin se encuentra una instrucción como

```
destino: MOV algo, R1
```

y se determina el valor de `destino`; es la dirección de la instrucción en curso. Entonces se hace el relleno de retroceso, recorriendo la lista de `destino` de todas las instrucciones que necesitan su dirección, sustituyendo la dirección de `destino` en los espacios en blanco que aparecen en los campos de dirección de esas instrucciones. Este enfoque es fácil de implantar si las instrucciones se pueden guardar en memoria hasta que se hayan determinado todas las direcciones de destino.

Este enfoque es razonable para un ensamblador que pueda guardar toda una salida en memoria. Como las representaciones intermedia y final del código para un ensamblador son aproximadamente iguales, y con seguridad casi de la misma longitud, el relleno de retroceso en toda la longitud del programa ensamblador no es inviable. Sin embargo, en un compilador, con un código intermedio que consuma mucho espacio, habrá que tener cuidado con la distancia en que se hace el relleno de retroceso.

1.6 HERRAMIENTAS PARA LA CONSTRUCCION DE COMPILADORES

El escritor del compilador, como cualquier programador, puede usar con provecho herramientas de software tales como depuradores, administradores de versiones, analizadores, etcétera. En el capítulo 11, se verá cómo se usan algunas de estas herramientas para implantar un compilador. Además de estas herramientas de desarrollo de software, se han creado herramientas más especializadas para ayudar a implantar varias fases de un compilador. En esta sección se mencionan brevemente; en los capítulos apropiados se tratan en detalle.

Poco después de escribirse el primer compilador, aparecieron sistemas para ayudar en el proceso de escritura de compiladores. A menudo se hace referencia a estos sistemas como *compiladores de compiladores, generadores de compiladores* o *sistemas generadores de traductores.* En gran parte, se orientan en torno a un modelo particular de lenguaje, y son más adecuados para generar compiladores de lenguajes similares al del modelo.

Por ejemplo, es tentador suponer que los analizadores léxicos para todos los lenguajes son en esencia iguales, excepto por las palabras clave y signos particulares que se reconocen. Muchos compiladores de compiladores de hecho producen rutinas fijas de análisis léxico para usar en el compilador generado. Estas rutinas sólo difieren

en la lista de palabras clave que reconocen, y esta lista es todo lo que debe proporcionar el usuario. El planteamiento es válido, pero puede no ser funcional si se requiere que reconozca componentes léxicos no estándar, como identificadores que pueden incluir ciertos caracteres distintos de letras y dígitos.

Se han creado algunas herramientas generales para el diseño automático de componentes específicos de compilador. Estas herramientas utilizan lenguajes especializados para especificar e implantar el componente, y pueden utilizar algoritmos bastante complejos. Las herramientas más efectivas son las que ocultan los detalles del algoritmo de generación y producen componentes que se pueden integrar con facilidad al resto del compilador. La siguiente es una lista de algunas herramientas útiles para la construcción de compiladores:

1. *Generadores de analizadores sintácticos.* Estos generadores producen analizadores sintácticos, normalmente a partir de una entrada fundamentada en una gramática independiente del contexto. En los primeros compiladores, el análisis sintáctico consumía no sólo gran parte del tiempo de ejecución del compilador, sino gran parte del esfuerzo intelectual de escribirlo. Esta fase se considera ahora una de las más fáciles de aplicar. Muchos de los "pequeños lenguajes" utilizados en la composición de este libro, como PIC (Kernighan [1982]) y EQN, se aplicaron en unos días por medio del generador de analizadores sintácticos descrito en la sección 4.7. Muchos de los generadores de analizadores sintácticos utilizan poderosos algoritmos de análisis sintáctico, y son demasiado complejos para realizarlos manualmente.

2. *Generadores de analizadores léxicos.* Estas herramientas generan automáticamente analizadores léxicos, por lo general a partir de una especificación basada en expresiones regulares, que se estudian en el capítulo 3. La organización básica del analizador léxico resultante es en realidad un autómata finito. En las secciones 3.5 y 3.8 se estudia un generador de analizadores léxicos típico y su implantación.

3. *Dispositivos de traducción dirigida por la sintaxis.* Estos producen grupos de rutinas que recorren el árbol de análisis sintáctico, como el de la figura 1.4, generando código intermedio. La idea básica es que se asocian una o más "traducciones" con cada nodo del árbol de análisis sintáctico, y cada traducción se define partiendo de traducciones en sus nodos vecinos en el árbol. Dichas herramientas se estudian en el capítulo 5.

4. *Generadores automáticos de código.* Tales herramientas toman un conjunto de reglas que definen la traducción de cada operación del lenguaje intermedio al lenguaje de máquina para la máquina objeto. Las reglas deben incluir suficiente detalle para poder manejar los distintos métodos de acceso posibles a los datos; por ejemplo, las variables pueden estar en registros, en una posición fija (estática) de memoria o pueden tener asignada una posición en una pila. La técnica fundamental es la de "concordancia de plantillas". Las proposiciones de código intermedio se reemplazan por "plantillas" que representan secuencias de instrucciones de máquina, de modo que las suposiciones sobre el almacenamiento de las variables concuerden de plantilla a plantilla. Como suele haber muchas

opciones en relación con la ubicación de las variables (por ejemplo, en uno o varios registros o en memoria), hay muchas formas posibles de "cubrir" el código intermedio con un conjunto dado de plantillas, y es necesario seleccionar una buena cobertura sin una explosión combinatoria en el tiempo de ejecución del compilador. Las herramientas de esta naturaleza se estudian en el capítulo 9.

5. *Dispositivos para análisis de flujo de datos.* Mucha de la información necesaria para hacer una buena optimación de código implica hacer un "análisis de flujo de datos", que consiste en la recolección de información sobre la forma en que se transmiten los valores de una parte de un programa a cada una de las otras partes. Las distintas tareas de esta naturaleza se pueden efectuar esencialmente con la misma rutina, en la que el usuario proporciona los detalles relativos a la relación que hay entre las proposiciones en código intermedio y la información que se está recolectando. En la sección 10.11 se describe una herramienta de esta naturaleza.

NOTAS BIBLIOGRAFICAS

Knuth [1962], al escribir sobre la historia de la escritura de compiladores en 1962, observaba que "en este campo ha habido una cantidad insólita de descubrimiento paralelo de la misma técnica por gente que trabajaba de manera independiente". A continuación hacía la observación de que, de hecho, varios individuos descubrieron "varios aspectos de una técnica, que se ha refinado con los años para producir un algoritmo ideal, que ninguno de los autores originales había llegado a imaginar". Dar crédito a las técnicas es una tarea arriesgada; la intención de las notas bibliográficas de este libro es sencillamente servir de ayuda para el posterior estudio de las publicaciones.

Los datos históricos sobre el desarrollo de los lenguajes de programación y compiladores hasta la llegada de FORTRAN se pueden encontrar en Knuth y Trabb Pardo [1977]. El libro de Wexelblat [1981] contiene memorias históricas sobre varios lenguajes de programación hechas por quienes participaron en su desarrollo.

Algunos de los primeros artículos fundamentales sobre la compilación están reunidos en las obras de Rosen [1967] y de Pollack [1972]. El número de enero de 1961 de *Communications of the ACM* da una imagen del estado en que se encontraba la escritura de compiladores en ese momento. En los trabajos de Randell y Russell [1964] se da un informe detallado de uno de los primeros compiladores de ALGOL 60.

Desde principios de los años sesenta, con el estudio de la sintaxis, las investigaciones prácticas han influido de manera profunda en el desarrollo de la tecnología de los compiladores, han tenido al menos tanta influencia como en cualquier otro área de la ciencia de la computación. La fascinación por la sintaxis ha declinado bastante, pero la compilación en conjunto continúa siendo objeto de una investigación muy dinámica. Los frutos de esta investigación resultarán evidentes cuando se examine la compilación con más detalle en los siguientes capítulos.

CAPITULO 2

Un compilador sencillo de una pasada

Este capítulo es una introducción al material de los capítulos 3 al 8 de este libro. Se presentan varias técnicas de compilación básicas ilustradas con el desarrollo de un programa en C operativo que traduce expresiones infijas a la forma postfija. Aquí, se hace énfasis en la etapa inicial de un compilador, esto es, en el análisis léxico, el análisis sintáctico y la generación de código intermedio. En los capítulos 9 y 10 se tratan los temas de generación y optimación de código.

2.1 PERSPECTIVA

Se puede definir un lenguaje de programación describiendo el aspecto de sus programas (la *sintaxis* del lenguaje) y el significado de sus programas (la *semántica* del lenguaje). Para especificar la sintaxis de un lenguaje, se presenta una notación muy usada llamada gramáticas independientes del contexto o BNF (abreviatura en inglés de Forma de Backus-Naur). Con las notaciones disponibles hoy, es mucho más difícil describir la semántica de un lenguaje que su sintaxis. Por consiguiente, para especificar la semántica de un lenguaje se usarán descripciones informales y ejemplos ilustrativos.

Además de servir para especificar la sintaxis de un lenguaje, se puede usar de apoyo una gramática independiente del contexto para guiar la traducción de programas. Una técnica de compilación orientada a la gramática, conocida como *traducción dirigida por la sintaxis*, es muy útil para organizar la etapa inicial de un compilador y se usará mucho en todo este capítulo.

Durante el estudio de la traducción dirigida por la sintaxis, se construirá un compilador que traduce expresiones infijas a la forma postfija, una notación en la que los operadores aparecen después de sus operandos. Por ejemplo, la forma postfija de la expresión `9-5+2` es `95-2+`. La notación postfija puede ser convertida directamente en código por un computador que haga todos sus cálculos utilizando una estructura de datos de pila (*stack*). Se empieza a construir un programa sencillo para traducir expresiones consistentes en dígitos separados por los signos más y menos en la forma postfija. Cuando las ideas básicas resulten evidentes, se extenderá el pro-

grama para poder manejar construcciones de lenguajes de programación más generales. Cada traductor se forma por la extensión sistemática del traductor anterior.

En este compilador, el *analizador léxico* convierte la cadena de caracteres de entrada en una cadena de componentes léxicos que se convierte en la entrada para la siguiente fase, como se muestra en la figura 2.1. El "traductor dirigido por la sintaxis" de la figura es una combinación de un analizador sintáctico y un generador de código intermedio. Una razón para empezar con expresiones formadas por dígitos y operadores consiste en hacer que el analizador léxico sea en un principio muy fácil; cada carácter de entrada forma un componente léxico único. Más adelante, se amplía el lenguaje para incluir construcciones léxicas, como números, identificadores y palabras clave. Para este lenguaje ampliado se construirá un analizador léxico que reúna los caracteres consecutivos de la entrada en componentes léxicos apropiados. La construcción de analizadores léxicos se estudiará en detalle en el capítulo 3.

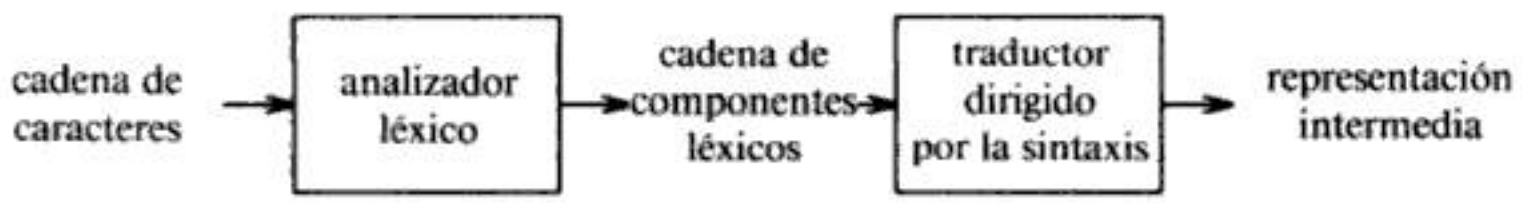


Fig. 2.1. Estructura de la etapa inicial del compilador.

2.2 DEFINICION DE LA SINTAXIS

En esta sección se introduce una notación, llamada gramática independiente del contexto (para abreviar, gramática), para especificar la sintaxis de un lenguaje. Esta notación se usará en todo el libro como parte de la especificación de la etapa inicial de un compilador.

Una gramática describe de forma natural la estructura jerárquica de muchas construcciones de los lenguajes de programación. Por ejemplo, una proposición **if-else** en C tiene la forma

if (expresión) proposición **else** proposición

Esto es, la proposición es la concatenación de la palabra clave **if**, un paréntesis que abre, una expresión, un paréntesis que cierra, una proposición, la palabra clave **else** y otra proposición. (En C no existe la palabra clave **then**.) Empleando la variable *expr* para denotar una expresión, y la variable *prop*, para una proposición, esta regla de estructuración se expresa

$$prop \rightarrow \textbf{if} \text{ (} expr \text{) } prop \textbf{ else } prop \qquad (2.1)$$

donde es posible leer la flecha como "puede tener la forma". Dicha regla se denomina *producción*. En una producción, los elementos léxicos, como la palabra clave **if** y los paréntesis, se llaman *componentes léxicos*. Las variables *expr* y *prop* representan secuencias de componentes léxicos y se llaman *no terminales*.

Una *gramática independiente del contexto* tiene cuatro componentes:

1. Un conjunto de componentes léxicos, denominados símbolos *terminales*.

2. Un conjunto de no terminales.
3. Un conjunto de producciones, en el que cada producción consta de un no terminal, llamado *lado izquierdo* de la producción, una flecha y una secuencia de componentes léxicos y no terminales, o ambos, llamado *lado derecho* de la producción.
4. La denominación de uno de los no terminales como símbolo *inicial*.

Se sigue la regla convencional de especificar las gramáticas dando una lista de sus producciones, donde las producciones del símbolo inicial se listan primero. Se supone que los dígitos, los signos como <= y las cadenas en **negritas**, como **while** son terminales. Un nombre en *cursiva* es un no terminal, y se supondrá que cualquier nombre o símbolo que no esté en *cursiva* es un componente léxico[1]. Por comodidad de notación, las producciones con el mismo no terminal del lado izquierdo pueden tener sus lados derechos agrupados, con los lados derechos alternativos separados por el símbolo |, que se leerá "o".

Ejemplo 2.1. En varios ejemplos de este capítulo se utilizan expresiones formadas por dígitos y signos *más* y *menos*, sea el caso, `9-5+2`, `3-1`, y `7`. Como un signo *más* o *menos* debe aparecer entre dos dígitos, se dice de dichas expresiones que son "listas de dígitos separados por signos *más* o *menos*". La siguiente gramática describe la sintaxis de esas expresiones. Las producciones son:

$$\textit{lista} \rightarrow \textit{lista} + \textit{dígito} \tag{2.2}$$
$$\textit{lista} \rightarrow \textit{lista} - \textit{dígito} \tag{2.3}$$
$$\textit{lista} \rightarrow \textit{dígito} \tag{2.4}$$
$$\textit{dígito} \rightarrow \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \tag{2.5}$$

Los lados derechos de las tres producciones con no terminal *lista* del lado izquierdo pueden agruparse de forma equivalente:

$$\textit{lista} \rightarrow \textit{lista} + \textit{dígito} \mid \textit{lista} - \textit{dígito} \mid \textit{dígito}$$

De acuerdo con las convenciones, los componentes léxicos de la gramática son los símbolos

```
+ - 0 1 2 3 4 5 6 7 8 9
```

Los no terminales son los nombres en *cursivas lista* y *dígito*, siendo *lista* el no terminal inicial, porque sus producciones se dieron primero. □

Se dice que una producción es *para* un no terminal si el no terminal aparece en el lado izquierdo de la producción. Una cadena de componentes léxicos es una secuencia de cero o más componentes léxicos. La cadena que contiene cero componentes léxicos, que se escribe ϵ, recibe el nombre de cadena *vacía*.

---

[1] Las letras *cursivas* individuales se usarán para propósitos adicionales cuando se estudien en detalle las gramáticas en el capítulo 4. Por ejemplo, se usarán X, Y y Z cuando se trata de un símbolo que es un componente léxico o un no terminal. Sin embargo, un nombre en *cursivas* con dos o más caracteres seguirá representando un no terminal.

De una gramática se derivan cadenas empezando con el símbolo inicial y reemplazando repetidamente un no terminal por el lado derecho de una producción para ese no terminal. Las cadenas de componentes léxicos derivadas del símbolo inicial forman el *lenguaje* que define la gramática.

Ejemplo 2.2. El lenguaje definido por la gramática del ejemplo 2.1 está formado por listas de dígitos separados por los signos más y menos.

Las diez producciones para el no terminal *dígito* hacen posible la representación de cualquiera de los componentes léxicos `0`, `1`, . . . , `9`. A partir de la producción (2.4), un dígito por sí solo es una lista. Las producciones (2.2) y (2.3) expresan el hecho de que al tomar cualquier lista y poner a continuación un signo más o menos, y después otro dígito, se tiene otra lista nueva.

Todo lo que se precisa para definir el lenguaje que interesa son las producciones (2.2) a (2.5). Por ejemplo, se puede deducir que `9-5+2` es una *lista* como sigue:

a) `9` es una *lista* de la producción (2.4), dado que `9` es un *dígito*.

b) `9-5` es una *lista* de la producción (2.3), dado que `9` es una *lista* y `5` es un *dígito*.

c) `9-5+2` es una *lista* de la producción (2.2), dado que `9-5` es una *lista* y `2` es un *digito*.

Este razonamiento se ilustra con el árbol de la figura 2.2. Cada nodo en el árbol está etiquetado con un símbolo de la gramática. Un nodo interior y sus hijos corresponden a una producción; el nodo interior corresponde al lado izquierdo de la producción, los hijos, al lado derecho. Estos árboles se conocen con el nombre de árboles de análisis sintáctico y se estudian más adelante. □

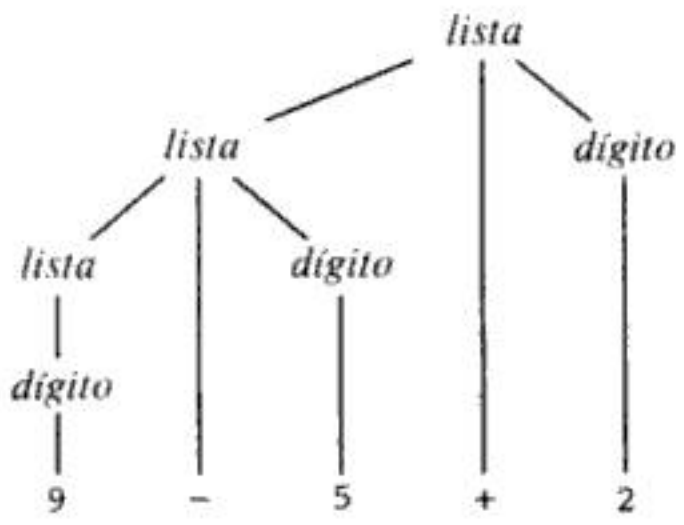


Fig. 2.2. Arbol de análisis sintáctico para `9-5+2` según la gramática del ejemplo 2.1.

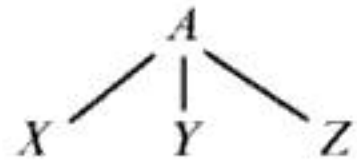
Ejemplo 2.3. Una clase algo distinta de listas es la secuencia de proposiciones separadas por los símbolos de punto y coma que se encuentran en los bloques **begin-end** de Pascal. Una característica de estas listas es que una lista vacía de proposiciones puede encontrarse entre los componentes léxicos **begin** y **end**. Se puede empezar a desarrollar una gramática para los bloques **begin-end** incluyendo las producciones:

$$
\begin{aligned}
\textit{bloque} &\rightarrow \textbf{begin}\ \textit{props\_opc}\ \textbf{end} \\
\textit{props\_opc} &\rightarrow \textit{lista\_props} \mid \epsilon \\
\textit{lista\_props} &\rightarrow \textit{lista\_props}\ ;\ \textit{prop} \mid \textit{prop}
\end{aligned}
$$

Obsérvese que el segundo lado derecho posible para *props_opc* ("lista de proposiciones opcional") es ϵ, que representa la cadena de símbolos vacía. Esto es, *props_opc* se puede reemplazar por la cadena vacía, de modo que un *bloque* puede estar formado por la cadena de dos componentes léxicos **begin end**. Fíjese que las producciones para *lista_props* son análogas a las de *lista* del ejemplo 2.1, con un punto y coma en lugar de un operador aritmético, y *prop*, en lugar de *dígito*. No se han mostrado las producciones para *prop*. Un poco más adelante se estudiarán las producciones apropiadas para varias clases de proposiciones: proposiciones **if**, proposiciones de asignación y otras.

Arboles de análisis sintáctico

Un árbol de análisis sintáctico indica gráficamente cómo del símbolo inicial de una gramática deriva una cadena del lenguaje. Si el no terminal A tiene una producción $A \rightarrow XYZ$, entonces un árbol de análisis sintáctico puede tener un nodo interior etiquetado con A y tres hijos etiquetados con X, Y y Z, de izquierda a derecha:



Formalmente, dada una gramática independiente del contexto, un *árbol de análisis sintáctico* es un árbol con las propiedades siguientes:

1. La raíz está etiquetada con el símbolo inicial.
2. Cada hoja está etiquetada con un componente léxico o con ϵ.
3. Cada nodo interior está etiquetado con un no terminal.
4. Si A es el no terminal que etiqueta a algún nodo interior y $X_1, X_2, \ldots, X_n$ son las etiquetas de los hijos de ese nodo, de izquierda a derecha, entonces $A \rightarrow X_1X_2 \ldots X_n$ es una producción. Aquí, $X_1, X_2, \ldots, X_n$ representa un símbolo que es un terminal o un no terminal. Como caso especial, si $A \rightarrow \epsilon$, entonces un nodo etiquetado con A tiene sólo un hijo etiquetado con ϵ.

Ejemplo 2.4. En la figura 2.2, la raíz está etiquetada con *lista*, que es el símbolo inicial de la gramática del ejemplo 2.1. Los hijos de la raíz están etiquetados, de izquierda a derecha, *lista*, +, y *dígito*. Obsérvese que

$$\textit{lista} \rightarrow \textit{lista}\ +\ \textit{dígito}$$

es una producción en la gramática del ejemplo 2.1. El mismo patrón con – se repite en el hijo izquierdo de la raíz, y cada uno de los tres nodos etiquetados con *dígito* tiene un hijo que está etiquetado con un dígito. □

Las hojas de un árbol de análisis sintáctico, leídas de izquierda a derecha, forman la *producción* del árbol, que es la cadena *generada* o *derivada* del no terminal

de la raíz del árbol de análisis sintáctico. En la figura 2.2, la cadena generada es 9−5+2, y todas las hojas se muestran en el nivel inferior. A partir de aquí, las hojas no se alinearán de esa forma. Cualquier árbol imparte un orden natural, de izquierda a derecha, a sus hojas, basándose en la idea de que si *a* y *b* son dos hijos con el mismo padre, y *a* está a la izquierda de *b*, entonces todos los descendientes de *a* están a la izquierda de los descendientes de *b*.

Otra definición del lenguaje generado por una gramática es el conjunto de cadenas que pueden ser generadas por un árbol de análisis sintáctico. El proceso de búsqueda de un árbol de análisis sintáctico para una cadena dada de componentes léxicos se denomina *análisis sintáctico* de esa cadena.

Ambigüedad

Se ha de tener cuidado al considerar *la* estructura de una cadena según una gramática. Aunque es evidente que cada árbol de análisis sintáctico deriva exactamente la cadena que se lee en sus hojas, una gramática puede tener más de un árbol de análisis sintáctico que genere una cadena dada de componentes léxicos. Esta clase de gramática se dice que es *ambigua*. Para demostrar que una gramática es ambigua, lo único que se requiere es encontrar una cadena de componentes léxicos que tenga más de un árbol de análisis sintáctico. Como una cadena que cuenta con más de un árbol de análisis sintáctico suele tener más de un significado, para aplicaciones de compilación es necesario diseñar gramáticas no ambiguas o utilizar gramáticas ambiguas con reglas adicionales para resolver las ambigüedades.

Ejemplo 2.5. Supóngase que no se hizo la distinción entre dígitos y listas según el ejemplo 2.1. Se podía haber escrito la gramática

$$\textit{cadena} \rightarrow \textit{cadena} + \textit{cadena} \mid \textit{cadena} - \textit{cadena} \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

Combinando la noción de *dígito* y *lista* en el no terminal *cadena* parece tener sentido superficial, porque un solo *dígito* es un caso especial de una *lista*.

Sin embargo, en la figura 2.3 se muestra la expresión 9−5+2 tiene ahora más de un árbol de análisis sintáctico. Los dos árboles de 9−5+2 corresponden a dos formas de agrupamiento entre paréntesis de la expresión: (9−5)+2 y 9−(5+2). Esta segunda forma de agrupamiento entre paréntesis da a la expresión el valor 2, en lugar del valor acostumbrado 6. La gramática del ejemplo 2.1 no permitía esta interpretación. □

Asociatividad de operadores

Por convención, 9+5+2 es equivalente a (9+5)+2, y 9−5−2 es equivalente a (9−5)−2. Cuando un operando con 5 tiene operadores a su izquierda y derecha, se necesitan convenciones para decidir qué operador considera ese operando. Se dice que el operador + *asocia a la izquierda*, porque un operando que tenga un signo más a ambos lados es tomado por el operador que esté a su izquierda. En la mayoría de los lenguajes de programación, los cuatro operadores aritméticos, adición, sustracción, multiplicación y división son asociativos por la izquierda.

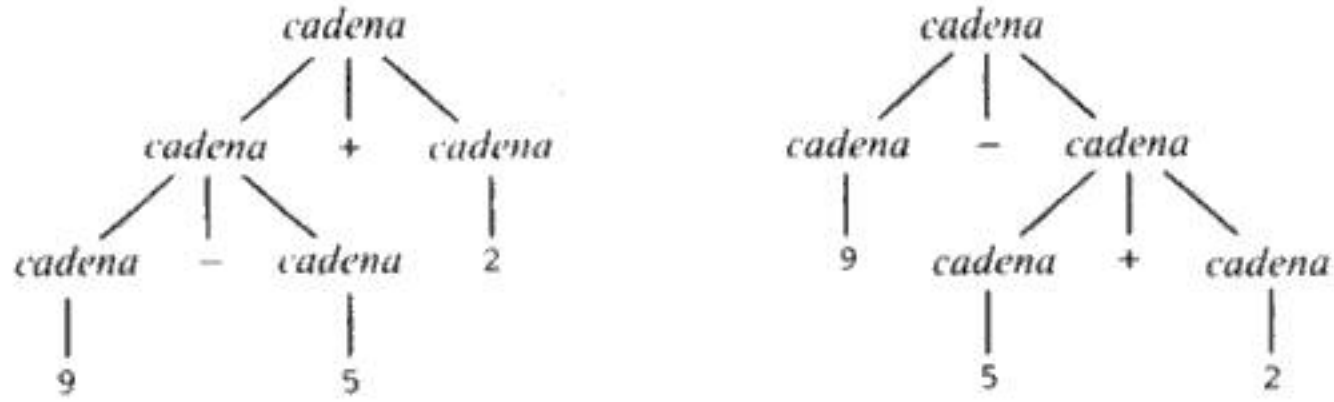


Fig. 2.3. Dos árboles de análisis sintáctico para 9−5+2.

Algunos operadores comunes, como la exponenciación, son asociativos por la derecha. Otro ejemplo análogo, el operador de asignación = en C es asociativo por la derecha: en C, la expresión a=b=c se trata igual que la expresión a=(b=c).

Las cadenas como a=b=c, con un operador asociativo por la derecha, son generadas por la siguiente gramática:

derecha → *letra* = *derecha* | *letra*
letra → a | b | . . . | z

El contraste entre un árbol de análisis sintáctico para un operador asociativo por la izquierda como −, y un árbol de análisis sintáctico para un operador asociativo por la derecha como =, se muestra en la figura 2.4. Adviértase que el árbol de análisis sintáctico para 9−5−2 desciende hacia la izquierda, mientras que el árbol de análisis sintáctico para a=b=c desciende hacia la derecha.

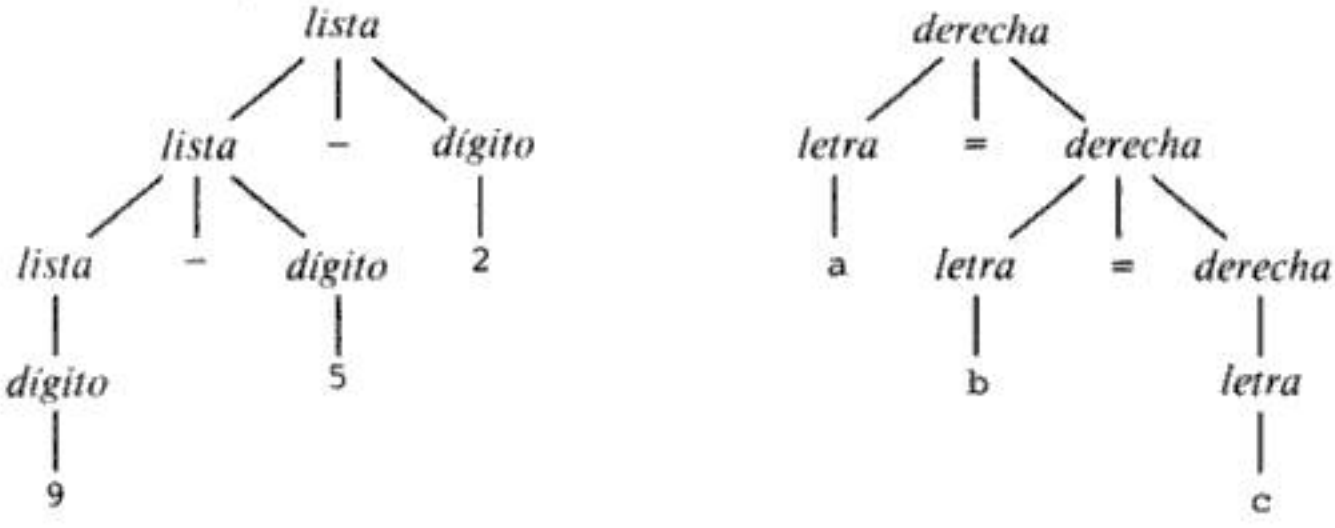


Fig. 2.4. Arboles de análisis sintáctico para operadores asociativos por la izquierda y por la derecha.

Precedencia de operadores

Considérese la expresión 9+5*2. Hay dos interpretaciones posibles de esta expresión: (9+5)*2 o 9+(5*2). La asociatividad de + y * no resuelve esta ambigüedad. Por esta razón, se necesita conocer la precedencia relativa de los operadores cuando esté presente más de una clase de operadores.

Se dice que * tiene *mayor precedencia* que + si * considera sus operandos antes de que lo haga +. En aritmética elemental, la multiplicación y división tienen

mayor precedencia que la adición y sustracción. Por tanto, `5` es considerado por `*` en `9+5*2` y en `9*5+2`; es decir, las expresiones son equivalentes a `9+(5*2)` y `(9*5)+2`, respectivamente.

Sintaxis de expresiones. Utilizando una tabla que muestre la asociatividad y precedencia de operadores se puede construir una gramática para expresiones aritméticas. Se empieza con los cuatro operadores aritméticos básicos y una tabla de precedencias, mostrando los operadores en orden de precedencia creciente, con los operadores de la misma precedencia en la misma línea:

asociativos por la izquierda:	+ −
asociativos por la izquierda:	* /

Se crean dos no terminales *expr* y *término* para los dos niveles de precedencia, y un no terminal adicional *factor* para generar unidades básicas en las expresiones. Las unidades básicas de las expresiones son de momento dígitos y expresiones entre paréntesis.

factor → **dígito** | (*expr*)

Ahora, considérese los operadores binarios, * y /, que tienen mayor precedencia. Como estos operadores asocian por la izquierda, las producciones son similares a las de las listas que asocian por la izquierda.

término → *término* * *factor*
| *término* / *factor*
| *factor*

De manera similar, *expr* genera listas de términos separados por los operadores aditivos.

expr → *expr* + *término*
| *expr* − *término*
| *término*

Por tanto, la gramática resultante es

expr → *expr* + *término* | *expr* − *término* | *término*
término → *término* * *factor* | *término* / *factor* | *factor*
factor → **dígito** | (*expr*)

Esta gramática considera una expresión como una lista de términos separados por los signos + o −, y un término, como una lista de factores separados por los signos * o /. Adviértase que cualquier expresión entre paréntesis es un factor, de manera que con los paréntesis se pueden desarrollar expresiones que tengan anidamiento de profundidad arbitraria (y también árboles de profundidad arbitraria).

Sintaxis de proposiciones. Las palabras clave permiten reconocer proposiciones en la mayoría de los lenguajes. Todas las proposiciones de Pascal comienzan con una palabra clave, excepto las asignacions y las llamadas a procedimientos. Algunas proposiciones de Pascal se definen por medio de la siguiente gramática (ambigua) en la que el componente léxico **id** representa un identificador.

prop → **id** := *expr*
| **if** *expr* **then** *prop*
| **if** *expr* **then** *prop* **else** *prop*
| **while** *expr* **do** *prop*
| **begin** *props_opc* **end**

El no terminal *props_opc* genera una lista de proposiciones, posiblemente vacía, separada por los símbolos de punto y coma, utilizando las producciones del ejemplo 2.3.

2.3 TRADUCCION DIRIGIDA POR LA SINTAXIS

Para traducir una construcción de un lenguaje de programación, un compilador puede necesitar tener en cuenta muchas características, además del código generado para la construcción. Por ejemplo, puede ocurrir que el compilador necesite conocer el tipo de la construcción, la posición de la primera instrucción del código objeto o el número de instrucciones generadas. Por tanto, los *atributos* asociados con las construcciones se mencionan de manera abstracta. Un atributo puede representar cualquier cantidad, por ejemplo, un tipo, una cadena, una posición de memoria o cualquier otra cosa.

En esta sección, se presenta un formalismo llamado definición dirigida por la sintaxis para especificar las traducciones para las construcciones de lenguajes de programación. Una definición dirigida por la sintaxis especifica la traducción de una construcción en función de atributos asociados con sus componentes sintácticos. En capítulos posteriores, las traducciones dirigidas por la sintaxis se usan para especificar muchas de las traducciones que ocurren en la etapa inicial de un compilador.

Para especificar traducciones, se introduce también una notación más orientada a procedimientos, denominada esquema de traducción. En este capítulo, se emplean esquemas de traducción para traducir expresiones infijas a la forma postfija. En el capítulo 5 se hace un análisis más detallado de las definiciones dirigidas por la sintaxis y su implantación.

Notación postfija

La *notación postfija* de una expresión E se puede definir de manera inductiva como sigue:

1. Si E es una variable o una constante, entonces la notación postfija de E es también E.
2. Si E es una expresión de la forma E_1 *op* E_2, donde *op* es cualquier operador binario, entonces la notación postfija de E es E_1' E_2' *op*, donde E_1' y E_2' son las notaciones postfijas de E_1 y E_2, respectivamente.
3. Si E es una expresión de la forma (E_1), entonces la notación postfija de E_1 es también la notación postfija de E.

La notación postfija no necesita paréntesis, porque la posición y la *ariedad* (número de argumentos) de los operadores permiten sólo una descodificación de una

expresión postfija. Por ejemplo, la notación postfija de `(9-5)+2` es `95-2+` y la notación postfija de `9-(5+2)` es `952+-`.

Definiciones dirigidas por la sintaxis

Una *definición dirigida por la sintaxis* utiliza una gramática independiente del contexto para especificar la estructura sintáctica de la entrada. A cada símbolo de la gramática le asocia un conjunto de atributos y a cada producción, un conjunto de *reglas semánticas* para calcular los valores de los atributos asociados con los símbolos que aparecen en esa producción. La gramática y el conjunto de reglas semánticas constituyen la definición dirigida por la sintaxis.

Una traducción es una transformación de una entrada en una salida. La salida para cada entrada x se especifica de la forma siguiente. Primero, se construye un árbol de análisis sintáctico para x. Supóngase que un nodo n del árbol de análisis sintáctico está etiquetado con el símbolo X de la gramática. Se escribe $X.a$ para indicar el valor del atributo a de X en ese nodo. El valor de $X.a$ en n se calcula por la regla semántica para el atributo a asociado con la producción de X utilizada en el nodo n. Al árbol de análisis sintáctico que muestre los valores de los atributos en cada nodo se dice que es un árbol de análisis sintáctico *con anotaciones*.

Atributos sintetizados

Se dice que un atributo está *sintetizado* si su valor en un nodo del árbol de análisis sintáctico se determina a partir de los valores de atributos de los hijos del nodo. Los atributos sintetizados tienen la atractiva propiedad de que se pueden calcular durante un solo recorrido ascendente del árbol de análisis sintáctico. En este capítulo sólo se usan atributos sintetizados; los atributos "heredados" se tratan en el capítulo 5.

Ejemplo 2.6. En la figura 2.5 se muestra una definición dirigida por la sintaxis para traducir expresiones, formadas por dígitos separados por los signos más o menos, a notación postfija. A cada no terminal está asociado un atributo t con un valor de la cadena que representa la notación postfija de la expresión generada por ese no terminal en un árbol de análisis sintáctico.

PRODUCCIÓN	REGLA SEMÁNTICA
$expr \rightarrow expr_1 + término$	$expr.t := expr_1.t \parallel término.t \parallel$ '+'
$expr \rightarrow expr_1 - término$	$expr.t := expr_1.t \parallel término.t \parallel$ '−'
$expr \rightarrow término$	$expr.t := término.t$
$término \rightarrow$ 0	$término.t :=$ '0'
$término \rightarrow$ 1	$término.t :=$ '1'
. . .	. . .
$término \rightarrow$ 9	$término.t :=$ '9'

Fig. 2.5. Definición dirigida por la sintaxis para traducción de infija a postfija.

La forma postfija de un dígito es el propio dígito; por ejemplo, la regla semántica asociada con la producción *término* → 9 define que *término.t* es 9 cuando esta producción se use en un nodo de un árbol de análisis sintáctico. Cuando se aplica la producción *expr* → *término*, el valor de *término.t* se transforma en el valor de *expr.t*.

La producción *expr* → $expr_1$ + *término* deriva una expresión con un operador más (el subíndice en $expr_1$ distingue el caso de *expr* en el lado derecho de aquel que está en el lado izquierdo). El operando izquierdo del operador más está dado por $expr_1$, y el operando derecho, por *término*. La regla semántica

$$expr.t := expr_1.t \parallel término.t \parallel \text{'+'}$$

asociada con esta producción define el valor del atributo *expr.t* mediante la concatenación de las formas postfijas $expr_1.t$ y *término.t* de los operandos izquierdo y derecho, respectivamente, y después agregando el signo más. El operador $\parallel$ en las reglas semánticas representa la concatenación de cadenas.

La figura 2.6 comprende el árbol de análisis sintáctico con anotaciones correspondiente al árbol de la figura 2.2. El valor del atributo *t* en cada nodo se calculó por la regla semántica asociada con la producción empleada en ese nodo. El valor del atributo en la raíz es la notación postfija de la cadena generada por el árbol de análisis sintáctico. □

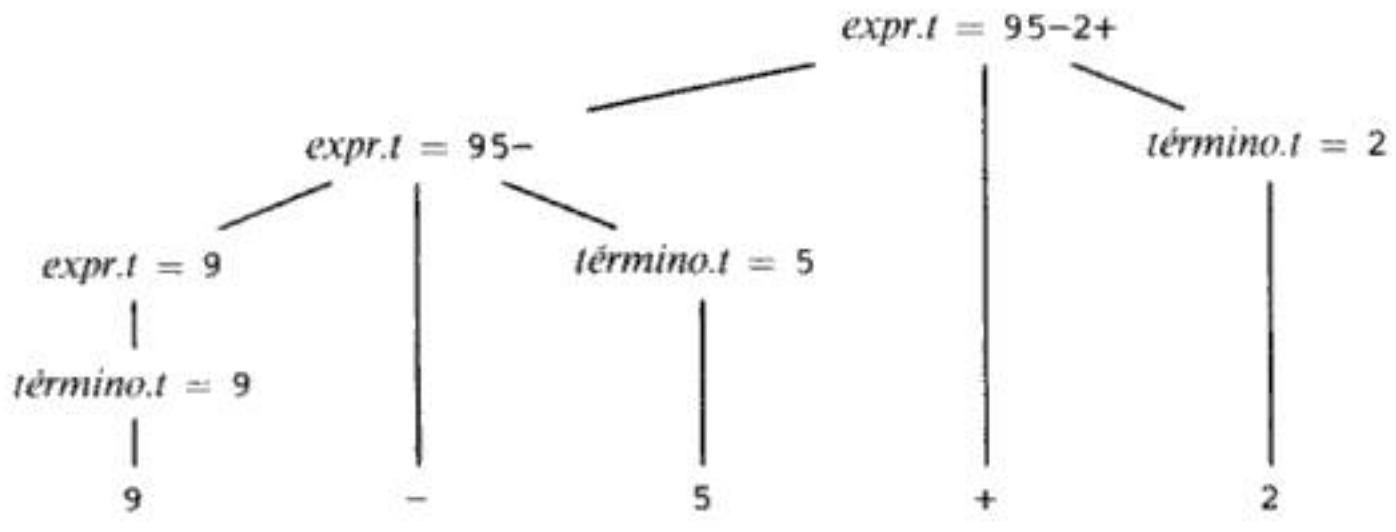


Fig. 2.6. Valores de atributos en los nodos de un árbol de análisis sintáctico.

Ejemplo 2.7. Supóngase que un robot se puede instruir para moverse un paso al este, norte, oeste o sur desde su posición inicial. Una secuencia de estas instrucciones se genera con la gramática siguiente:

sec → *sec instr* | **comienza**
instr → **este** | **norte** | **oeste** | **sur**

En la figura 2.7 se muestran los cambios en la posición del robot si se le proporciona la entrada

comienza oeste sur este este este norte norte

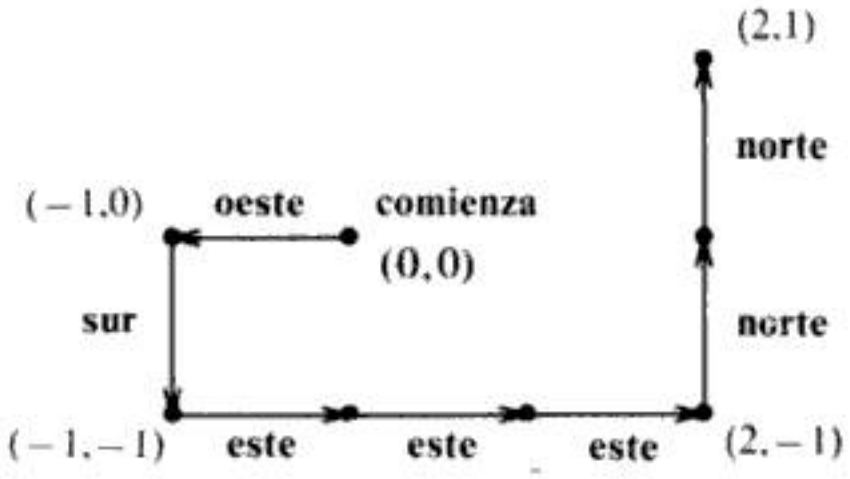


Fig. 2.7. Seguimiento de una posición del robot.

En la figura, una posición se marca con un par (x,y), donde x e y representan el número de pasos al este y al norte, respectivamente, desde la posición inicial. (Si x es negativo, entonces el robot se encuentra al oeste de la posición inicial; de manera similar, si y es negativo, entonces el robot se encuentra al sur de la posición inicial.)

Para traducir una secuencia de instrucciones a una posición del robot, se construirá una definición dirigida por la sintaxis. Se usarán dos atributos, *sec.x* y *sec.y*, para seguir la posición que resulte de una secuencia de instrucciones generada por el no terminal *sec*. Al principio, *sec* genera **comienza**, asignando el valor inicial 0 a *sec.x* y *sec.y*, según se indica en el nodo interior del árbol de análisis sintáctico de **comienza oeste sur**, situado en el extremo izquierdo de la figura 2.8.

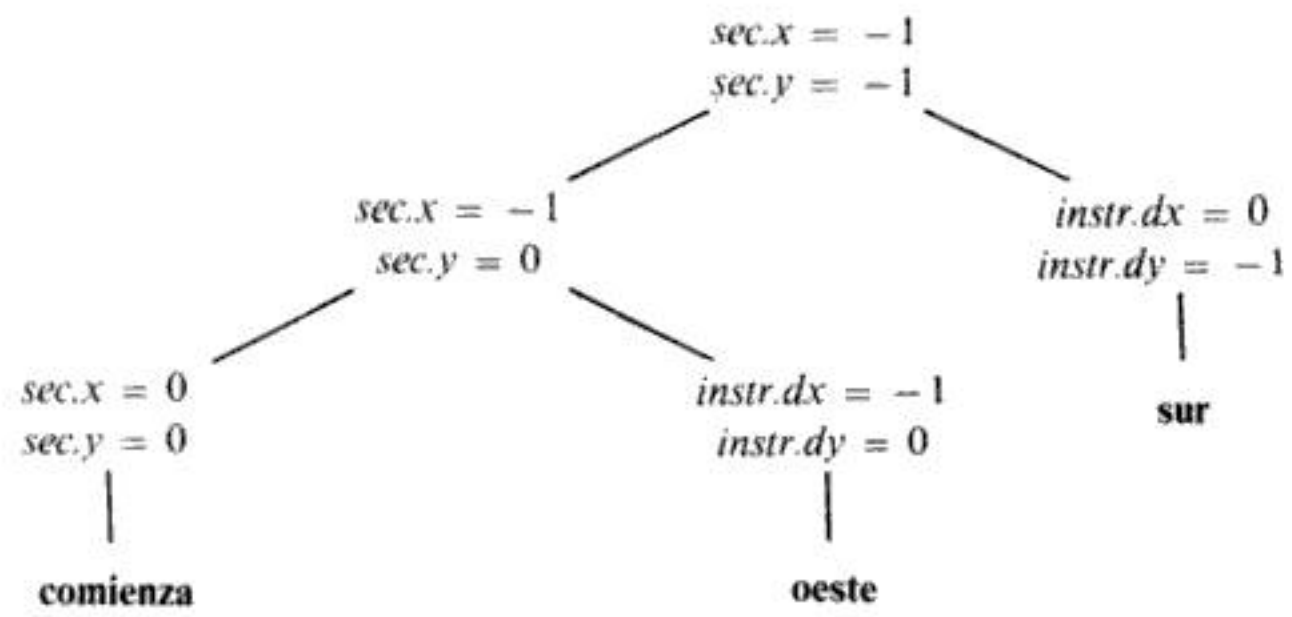


Fig. 2.8. Arbol de análisis sintáctico con anotaciones para **comienza oeste sur.**

El cambio en la posición a causa de una instrucción individual derivada de *instr* se da por los atributos *instr.dx* e *instr.dy*. Por ejemplo, si *instr* deriva **oeste**, entonces $instr.dx = -1$ e $instr.dy = 0$. Supóngase que una secuencia *sec* se forma con una secuencia sec_1 seguida de una nueva instrucción *instr*. Entonces, la nueva posición del robot está dada por las reglas

$$sec.x := sec_1.x + instr.dx$$
$$sec.y := sec_1.y + instr.dy$$

En la figura 2.9 se muestra una definición dirigida por la sintaxis para traducir una secuencia de instrucciones a una posición del robot. □

PRODUCCIÓN	REGLA SEMÁNTICA
sec → **comienza**	*sec.x* := 0 *sec.y* := 0
sec → sec_1 *instr*	*sec.x* := $sec_1.x$ + *instr.dx* *sec.y* := $sec_1.y$ + *instr.dy*
instr → **este**	*instr.dx* := 1 *instr.dy* := 0
instr → **norte**	*instr.dx* := 0 *instr.dy* := 1
instr → **oeste**	*instr.dx* := −1 *instr.dy* := 0
instr → **sur**	*instr.dx* := 0 *instr.dy* := −1

Fig. 2.9. Definición dirigida por la sintaxis de la posición del robot.

Recorridos en profundidad

Una definición dirigida por la sintaxis no impone ningún orden específico a la evaluación de atributos en un árbol de análisis sintáctico; cualquier orden de evaluación que calcule un atributo *a*, después de haber calculado todos los demás atributos de los que *a* depende es aceptable. En general, es posible que haya que evaluar algunos atributos cuando se llega por primera vez a un nodo durante un recorrido del árbol de análisis sintáctico, otros, después de haber visitado todos sus hijos o en algún punto entre las visitas a los hijos del nodo. En el capítulo 5 se analizan con más detalle los órdenes de evaluación apropiados.

Todas las traducciones de este capítulo se pueden hacer evaluando las reglas semánticas de los atributos en un árbol de análisis sintáctico en un orden predeterminado. Un *recorrido* de un árbol comienza en la raíz y visita cada nodo del árbol en un orden indeterminado. En este capítulo, las reglas semánticas se evaluarán mediante el recorrido en profundidad que se define en la figura 2.10. Este recorrido empieza en la raíz y visita recursivamente a los hijos de cada nodo en orden de izquierda a derecha, como se muestra en la figura 2.11. Las reglas semánticas en un nodo dado se evalúan cuando todos los descendientes de ese nodo hayan sido visi-

```
procedure visita (n: nodo);
begin
    for cada hijo m de n, de izquierda a derecha do
        visita (m);
    evalúa reglas semánticas en el nodo n
end
```

Fig. 2.10. Un recorrido en profundidad de un árbol.

tados. Se llama "en profundidad" porque siempre que pueda, visita a un hijo no visitado de un nodo, de modo que intenta visitar los nodos más alejados de la raíz lo antes posible.

Esquemas de traducción

En el resto de este capítulo, se usa una especificación orientada a procedimientos para definir una traducción. Un *esquema de traducción* es una gramática independiente del contexto en la que se encuentran intercalados, en los lados derechos de las producciones, fragmentos de programa llamados *acciones semánticas*. Un esquema de traducción es como una definición dirigida por la sintaxis, con la excepción de que el orden de evaluación de las reglas semánticas se muestra explícitamente. La posición en la que se ejecuta alguna acción se da entre llaves y se escribe en el lado derecho de una producción, por ejemplo,

$$resto \rightarrow + \ término \ \{ \ print('+') \ \} \ resto_1$$

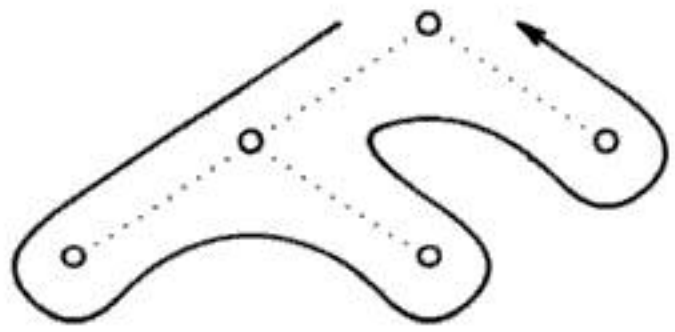


Fig. 2.11. Ejemplo de un recorrido en profundidad de un árbol.

Un esquema de traducción genera una salida para cada frase x generada por la gramática subyacente mediante la ejecución de las acciones en el orden en que aparecen durante un recorrido en profundidad de un árbol de análisis sintáctico para x. Sea el caso de un árbol de análisis sintáctico con un nodo etiquetado con *resto* que represente a esta producción. La acción $\{ print('+') \}$ se efectuará después de recorrer el subárbol de *término*, pero antes de visitar al hijo $resto_1$.

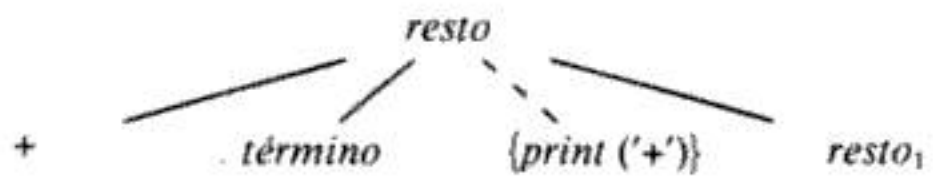


Fig. 2.12. Construcción de una hoja adicional correspondiente a una acción semántica.

Cuando se dibuja un árbol de análisis sintáctico de un esquema de traducción, se indica una acción construyendo un hijo adicional, conectado al nodo para su producción por una línea de puntos. Por ejemplo, la parte del árbol de análisis sintáctico para la producción y la acción anteriores se representa en la figura 2.12. El nodo para una acción semántica no tiene hijos, de modo que la acción se realiza cuando se ve por primera vez ese nodo.

Emisión de una traducción

En este capítulo, las acciones semánticas en los esquemas de traducción escribirán la salida de una traducción en un archivo, una cadena o un carácter a la vez. Por ejemplo, se traduce 9−5+2 a 95−2+ imprimiendo cada carácter de 9−5+2 justo una vez, sin usar ningún almacenamiento para la traducción de subexpresiones. Cuando la salida se crea incrementalmente de este modo, es importante el orden en que se imprimen los caracteres.

Adviértase que las definiciones dirigidas por las sintaxis mencionadas hasta ahora tienen la siguiente propiedad importante: la cadena que representa la traducción del no terminal del lado izquierdo de cada producción es la concatenación de las traducciones de los no terminales de la derecha, en igual orden que en la producción, con algunas cadenas adicionales (tal vez ninguna) intercaladas. Con esta propiedad, una definición dirigida por la sintaxis se denomina *simple*. Por ejemplo, considérense la primera producción y la regla semántica de la definición dirigida por la sintaxis de la figura 2.5:

PRODUCCIÓN	REGLA SEMÁNTICA	
$expr \rightarrow expr_1 + término$	$expr.t := expr_1.t \parallel término.t \parallel '+'$	(2.6)

Aquí, la traducción *expr.t* es la concatenación de las traducciones de $expr_1$ y *término*, seguida del símbolo +. Adviértase que $expr_1$ aparece antes que *término* en el lado derecho de la producción.

Entre *término.t* y $resto_1.t$ aparece una cadena adicional en

PRODUCCIÓN	REGLA SEMÁNTICA	
$resto \rightarrow + término\ resto_1$	$resto.t := término.t \parallel '+' \parallel resto_1.t$	(2.7)

pero, de nuevo, el no terminal *término* aparece antes que $resto_1$ en el lado derecho.

Las definiciones simples dirigidas por la sintaxis se pueden implantar con esquemas de traducción en los que las acciones impriman las cadenas adicionales en el orden en que aparecen en la definición. Las acciones de las siguientes producciones imprimen las cadenas adicionales de (2.6) y (2.7), respectivamente:

$$expr \rightarrow expr_1 + término \{ print('+') \}$$
$$resto \rightarrow + término \{ print('+') \} resto_1$$

Ejemplo 2.8. La figura 2.5 contiene una definición simple para traducir expresiones a la forma postfija. En la figura 2.13 se da un esquema de traducción derivado de esta definición y en la figura 2.14 se muestra un árbol de análisis sintáctico con ac-

$$\begin{aligned}
expr &\rightarrow expr + término && \{ print('+') \} \\
expr &\rightarrow expr - término && \{ print('-') \} \\
expr &\rightarrow término \\
término &\rightarrow \texttt{0} && \{ print('0') \} \\
término &\rightarrow \texttt{1} && \{ print('1') \} \\
&\ldots \\
término &\rightarrow \texttt{9} && \{ print('9') \}
\end{aligned}$$

Fig. 2.13. Acciones que traducen expresiones a la notación postfija.

ciones para 9−5+2. Obsérvese que aunque las figuras 2.6 y 2.14 representan la misma transformación de entrada a salida, la traducción se construye de manera distinta en los dos casos; la figura 2.6 vincula la salida a la raíz del árbol de análisis sintáctico, mientras que la figura 2.14 imprime la salida de forma incremental.

La raíz de la figura 2.14 representa la primera producción de la figura 2.13. En un recorrido en profundidad, primero se realizan todas las acciones del subárbol para el operando izquierdo *expr* cuando se recorre el subárbol situado más a la izquierda de la raíz, después se visita la hoja +, en la que no hay ninguna acción, a continuación realizan las acciones del subárbol para el operando derecho *término* y, por último, se realiza la acción semántica { *print* ('+') } en el nodo adicional.

Como las producciones para *término* tienen sólo un dígito en el lado derecho, ese dígito se imprime por medio de las acciones para las producciones. No se necesita ninguna salida para la producción *expr* → *término*, y sólo se requiere imprimir el operador en las dos primeras producciones. Cuando se ejecutan durante un recorrido en profundidad del árbol de análisis sintáctico, las acciones de la figura 2.14 imprimen 95−2+. □

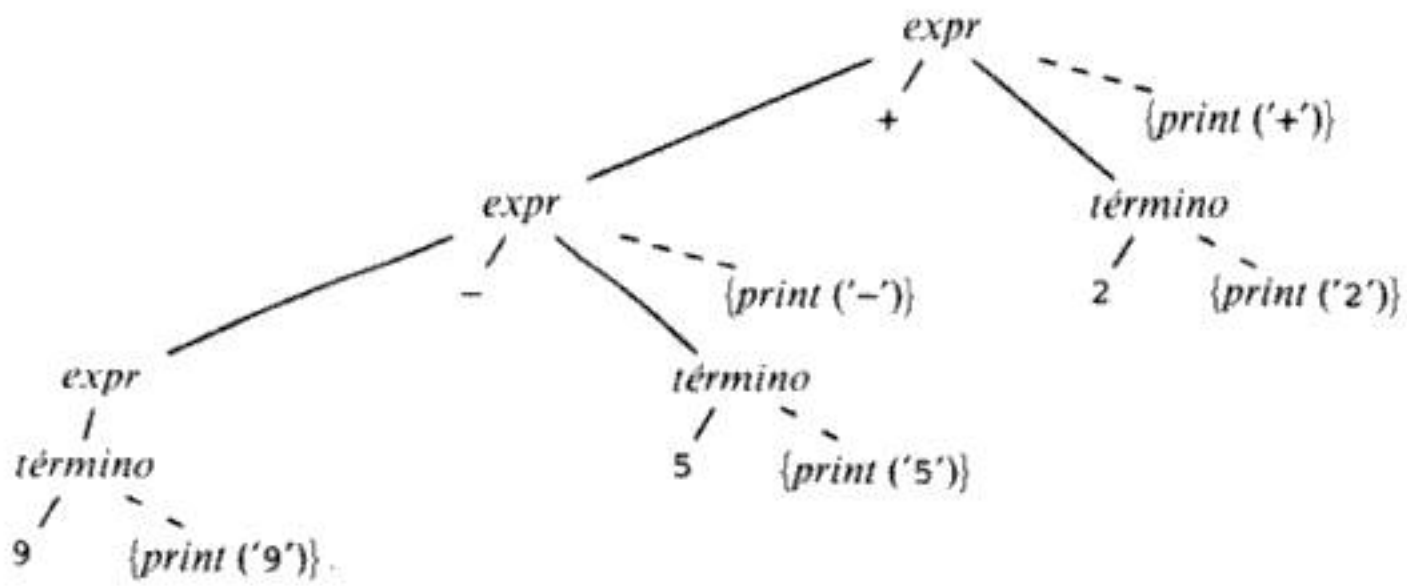


Fig. 2.14. Acciones que traducen 9−5+2 a 95−2+.

Como regla general, la mayoría de los métodos de análisis sintáctico procesan su entrada de izquierda a derecha de forma "voraz"; esto es, construyen el máximo posible de un árbol de análisis sintáctico antes de leer el siguiente componente léxico de la entrada. En un esquema de traducción simple (obtenido de una definición simple dirigida por la sintaxis) las acciones se efectúan también de izquierda a derecha. Por tanto, para implantar un esquema de traducción simple se pueden ejecutar las acciones semánticas durante el análisis sintáctico; no es necesario construir el árbol de análisis sintáctico.

2.4 ANALISIS SINTACTICO

El análisis sintáctico es el proceso de determinar si una cadena de componentes léxicos puede ser generada por una gramática. En el estudio de este problema, es útil pensar en construir un árbol de análisis sintáctico, aunque, de hecho, un compilador no lo construya. Sin embargo, un analizador sintáctico deberá poder construir el árbol, pues de otro modo, no se puede garantizar que la traducción sea correcta.

En esta sección se introduce un método de análisis sintáctico que puede aplicarse en la construcción de traductores dirigidos por la sintaxis. En la siguiente sección se presenta un programa completo en C que implanta el esquema de traducción de la figura 2.13. Una posibilidad viable es utilizar una herramienta de software para generar un traductor directamente a partir de un esquema de traducción. Para la descripción de esa herramienta, véase la sección 4.9; con ella se puede implantar sin modificación el esquema de traducción de la figura 2.13.

Para cualquier gramática, se puede construir un analizador sintáctico. Sin embargo, las gramáticas que se usan en la práctica tienen una forma especial. Para cualquier gramática independiente del contexto hay un analizador sintáctico que toma como máximo un tiempo de $O(n^3)$ para hacer el análisis de una cadena de n componentes léxicos. Pero un tiempo de orden cúbico es demasiado caro. Dado un lenguaje de programación, en general se puede construir una gramática que se pueda analizar sintácticamente con rapidez. Los algoritmos lineales son suficientes para hacer el análisis sintáctico de casi todos los lenguajes que surgen en la práctica. Los analizadores sintácticos de lenguajes de programación suelen hacer un examen simple de izquierda a derecha de la entrada, viendo un componente léxico a la vez.

La mayoría de los métodos de análisis sintáctico están comprendidos en dos clases, llamadas métodos *descendente* y *ascendente*. Estos términos hacen referencia al orden en que se construyen los nodos del árbol de análisis sintáctico. En el primero, la construcción se inicia en la raíz y avanza hacia las hojas, mientras que en el segundo, la construcción se inicia en las hojas y avanza hacia la raíz. La popularidad de los analizadores sintácticos descendentes se debe al hecho de poder construir manualmente analizadores sintácticos eficientes con mayor facilidad, utilizando métodos descendentes. Sin embargo, el análisis sintáctico ascendente puede manejar una clase mayor de gramáticas y esquemas de traducción, de modo que las herramientas de software para generar analizadores sintácticos directamente a partir de las gramáticas tienden a utilizar métodos ascendentes.

Análisis sintáctico descendente

Se presenta el análisis sintáctico descendente considerando una gramática adecuada para esta clase de método. Más adelante, en esta sección, se considera la construcción de analizadores sintácticos descendentes en general. La siguiente gramática genera un subconjunto de los tipos de Pascal. Se utiliza el componente léxico **puntopunto** para enfatizar que la secuencia de caracteres se trata como una unidad.

$$
\begin{array}{rcl}
\textit{tipo} & \rightarrow & \textit{simple} \\
 & | & \uparrow \textbf{id} \\
 & | & \textbf{array} \; [\; \textit{simple} \;] \; \textbf{of} \; \textit{tipo} \\
\textit{simple} & \rightarrow & \textbf{integer} \\
 & | & \textbf{char} \\
 & | & \textbf{núm puntopunto núm}
\end{array}
\tag{2.8}
$$

La construcción descendente de un árbol de análisis sintáctico se hace empezando por la raíz, etiquetada con el no terminal inicial, y realizando de forma repetida los dos pasos siguientes (véase un ejemplo en la Fig. 2.5).

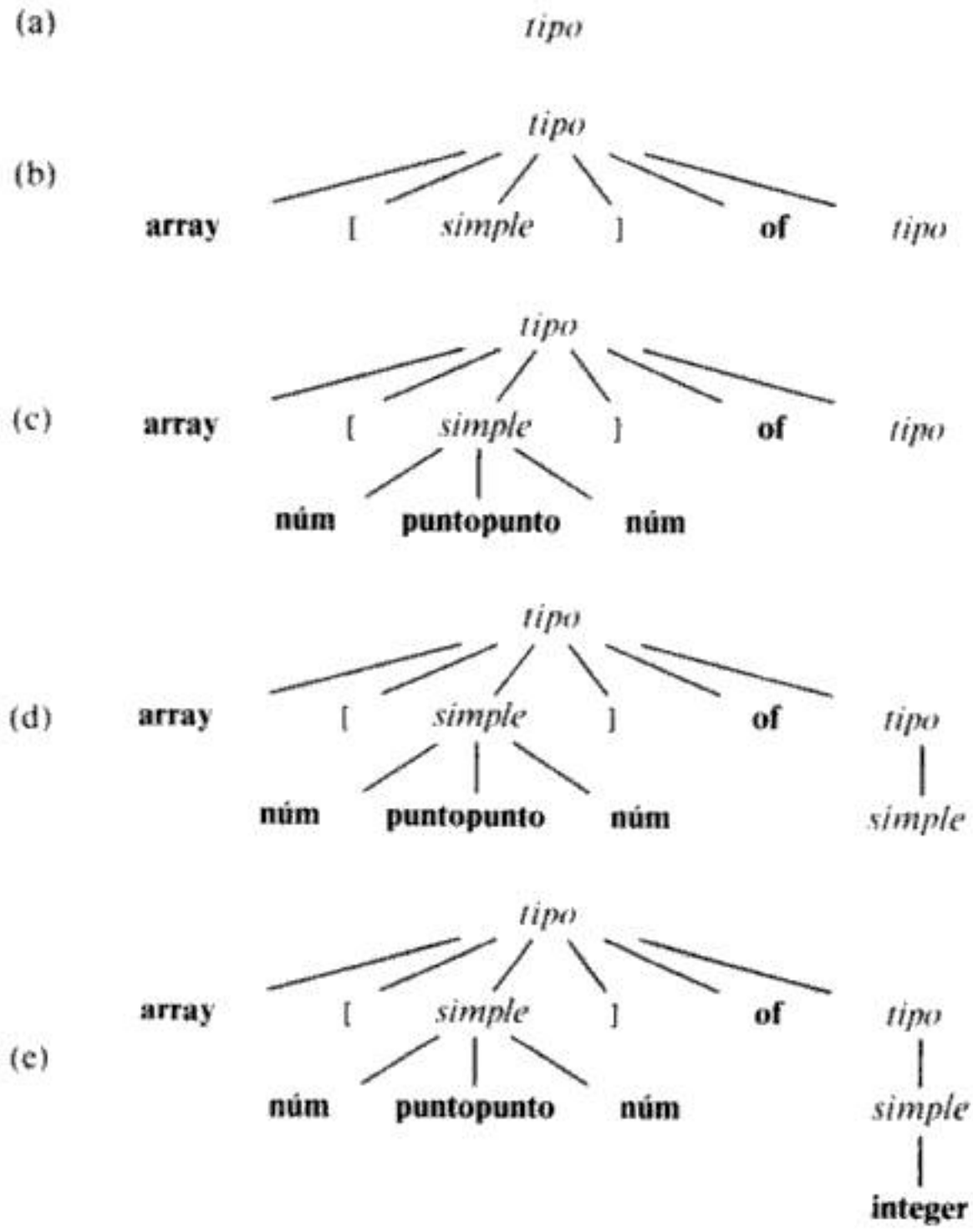


Fig. 2.15. Pasos en la construcción descendente de un árbol de análisis sintáctico.

1. En el nodo *n*, etiquetado con el no terminal *A*, selecciónese una de las producciones para *A* y constrúyase los hijos de *n* para los símbolos del lado derecho de la producción.
2. Encuéntrese el siguiente nodo en el que ha de construirse un subárbol.

Para algunas gramáticas, los pasos anteriores se aplican durante un examen sencillo, de izquierda a derecha, de la cadena de entrada. Muchas veces, el componente léxico en curso analizado en la entrada se denomina símbolo de *preanálisis*. Inicialmente, el símbolo de preanálisis es el primero, es decir, el componente léxico situado más a la izquierda de la cadena de entrada. La figura 2.16 ilustra el análisis sintáctico de la cadena

array [núm puntopunto núm] of integer

Inicialmente, el componente léxico **array** es el símbolo de preanálisis y la parte conocida del árbol de análisis sintáctico consiste en la raíz, etiquetada con el no terminal inicial *tipo* en la figura 2.16(a). El objetivo es construir el resto del árbol de

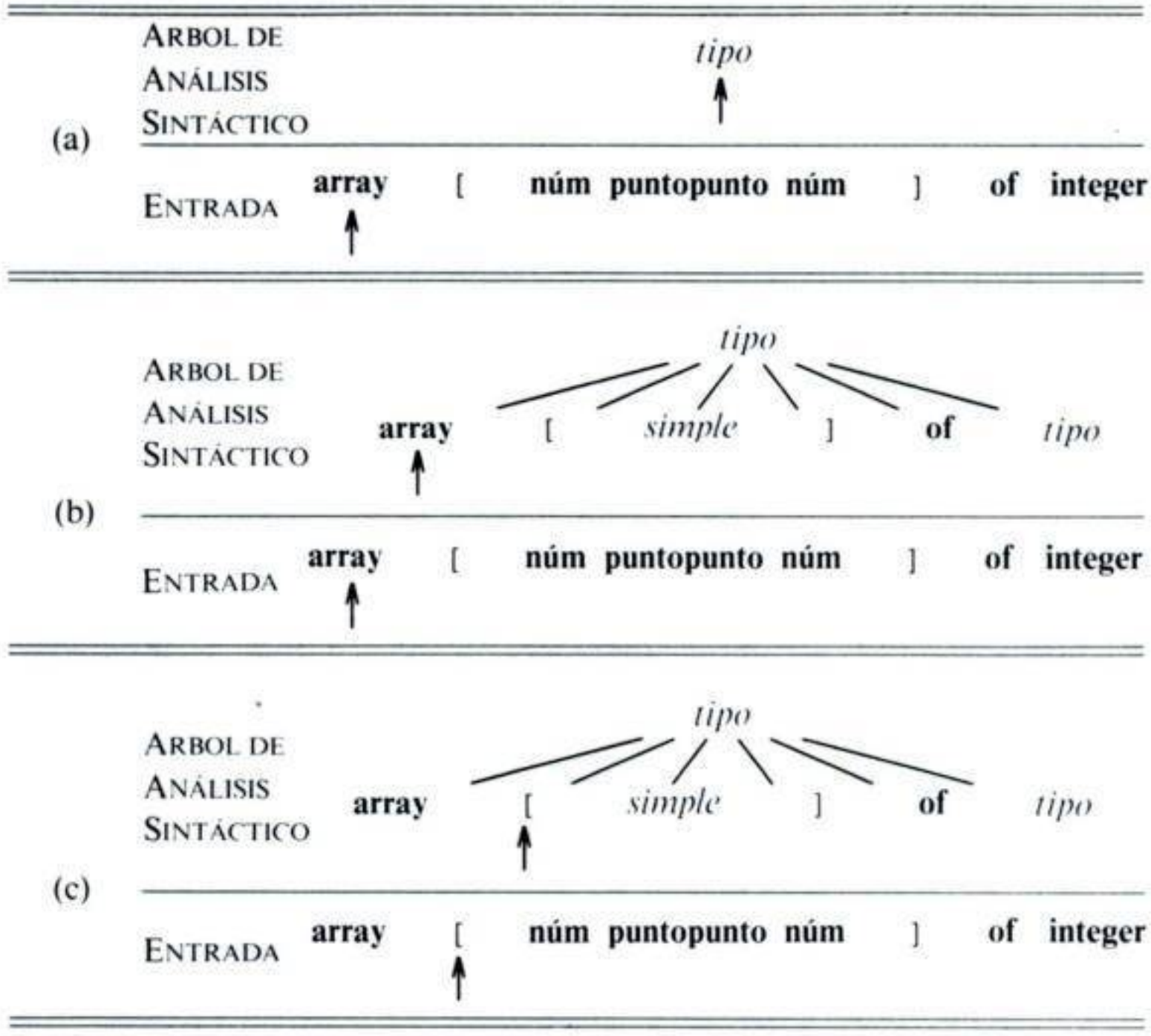


Fig. 2.16. Análisis sintáctico descendente durante el examen de la entrada de izquierda a derecha.

análisis sintáctico de modo que la cadena generada por él concuerde con la cadena de entrada.

Para que ocurra una concordancia, el no terminal *tipo* de la figura 2.16(a) debe derivar una cadena que empiece con el símbolo de preanálisis **array**. En la gramática (2.8) hay exactamente una producción para *tipo* que deriva tal cadena, por lo que se elige, y se construyen los hijos de la raíz etiquetados con los símbolos del lado derecho de la producción.

Cada una de las tres imágenes de la figura 2.16 tiene flechas que indican el símbolo de preanálisis de la entrada y el nodo que se está considerando. Cuando se han construido los hijos de un nodo, después se considera el hijo que está más a la izquierda. En la figura 2.16(b), se han construido los hijos de la raíz, y se está considerando el hijo situado más a la izquierda, etiquetado con **array**.

Cuando el nodo que se está considerando en el árbol de análisis sintáctico es el de un terminal y el terminal concuerda con el símbolo de preanálisis, se avanza en el árbol de análisis sintáctico y en la entrada. El siguiente componente léxico de la entrada se convierte en el nuevo símbolo de preanálisis y se considera el siguiente hijo del árbol de análisis sintáctico. En la figura 2.16(c), la flecha del árbol de análisis sintáctico avanza hasta el siguiente hijo de la raíz y la flecha de la entrada avanza

hasta el siguiente componente léxico [. Después del siguiente avance, la flecha del árbol de análisis sintáctico apuntará al hijo etiquetado con el no terminal *simple*. Cuando se considera un nodo etiquetado con un no terminal, se repite el proceso de seleccionar una producción para el no terminal.

En general, la selección de una producción para un no terminal puede implicar un proceso de prueba y error; esto es, se puede probar con una producción y retroceder para hacer el intento con otra producción si la primera resulta inadecuada. Una producción es inadecuada cuando, después de usar las producciones, no se puede completar el árbol para que concuerde con la cadena de entrada. Sin embargo, hay un caso de especial importancia, llamado analizador sintáctico predictivo, en el que no hay retroceso.

Análisis sintáctico predictivo

El *análisis sintáctico descendente recursivo* es un método descendente en el que se ejecuta un conjunto de procedimientos recursivos para procesar la entrada. A cada no terminal de una gramática se asocia un procedimiento. Aquí, se considera una forma especial de análisis sintáctico descendente recursivo, llamado análisis sintáctico predictivo, en el que el símbolo de preanálisis determina sin ambigüedad el procedimiento seleccionado para cada no terminal. La secuencia de procedimientos llamados en el procesamiento de la entrada define implícitamente un árbol de análisis sintáctico para la entrada.

El analizador sintáctico predictivo de la figura 2.17 consta de procedimientos para los no terminales *tipo* y *simple* de la gramática (2.8) y un procedimiento adicional, *parea*. Se usa *parea* para simplificar el código de *tipo* y *simple*; si su argumento *t* concuerda con el símbolo de preanálisis, avanza hacia el siguiente componente léxico de entrada. De ese modo, *parea* modifica a la variable *preanálisis*, que es el componente léxico en curso que acaba de entregar el análisis léxico.

El análisis sintáctico comienza con una llamada al procedimiento del no terminal inicial *tipo* de esta gramática. Con la misma entrada que en la figura 2.16, *preanálisis* es inicialmente el primer componente léxico **array**. El procedimiento *tipo* ejecuta el código.

parea (**array**); *parea* ('['); *simple*; *parea* (']') *parea* (**of**); *tipo* (2.9)

que corresponde al lado derecho de la producción

tipo → **array** [*simple*] **of** *tipo*

Obsérvese que cada terminal del lado derecho se parea con el símbolo de preanálisis y que cada no terminal de la derecha proporciona una llamada a su procedimiento.

Con la entrada de la figura 2.16, después de haber concordancia entre los componentes léxicos **array** y [, el símbolo de preanálisis es **núm**. En este punto se llama al procedimiento *simple* y se ejecuta el código de su cuerpo.

parea (**núm**); *parea* (**puntopunto**); *parea* (**núm**)

El símbolo de preanálisis guía la selección de la producción que se desea utilizar. Si el lado derecho de una producción empieza con un componente léxico, entonces la producción se usa cuando el símbolo de preanálisis coincide con el componente

```
procedure parea (t: complex);
begin
    if preanálisis = t then
        preanálisis := sigcomplex
    else error
end;

procedure tipo;
begin
    if preanálisis is in { integer, char, núm } then
        simple
    else if preanálisis = '↑' then begin
        parea ('↑'); parea (id)
    end
    else if preanálisis = array then begin
        parea (array); parea ('['); simple; parea (']'); parea (of); tipo
    end
    else error
end;

procedure simple;
begin
    if preanálisis = integer then
        parea (integer)
    else if preanálisis = char then
        parea (char)
    else if preanálisis = núm then begin
        parea (núm); parea (puntopunto); parea (núm)
    end
    else error
end;
```

Fig. 2.17. Seudocódigo de un analizador sintáctico predictivo.

léxico. Ahora, considérese un lado derecho que empiece con un no terminal, como en el caso

$$tipo \rightarrow simple \tag{2.10}$$

Esta producción se emplea cuando el símbolo de preanálisis se puede generar a partir de *simple*. Por ejemplo, durante la ejecución del fragmento de código (2.9), supóngase que el símbolo de preanálisis es **integer** cuando el control llega a la llamada a procedimiento *tipo*. No hay ninguna producción para *tipo* que comience con el componente léxico **integer**. Sin embargo, sí hay una producción para *simple* que comienza con **integer**, de manera que se usa la producción (2.10) teniendo en cuenta que *tipo* llama al procedimiento *simple* en el símbolo de preanálisis **integer**.

El análisis sintáctico predictivo depende de la información sobre los primeros símbolos que pueden ser generados por el lado derecho de una producción. Para

precisar, sea α el lado derecho de una producción para el no terminal A. Se define PRIMERO(α) como el conjunto de componentes léxicos que opere como los primeros símbolos de una o más cadenas generadas a partir de α. Si α es ϵ o puede generar ϵ, entonces ϵ también está en PRIMERO(α)[2]. Por ejemplo,

$$\text{PRIMERO}(\textit{simple}) = \{\ \textbf{integer}, \textbf{char}, \textbf{núm}\ \}$$
$$\text{PRIMERO}(\uparrow \textbf{id}) = \{\ \uparrow\ \}$$
$$\text{PRIMERO}(\textbf{array}\ [\ \textit{simple}\ ]\ \textbf{of}\ \textit{tipo}) = \{\ \textbf{array}\ \}$$

En la práctica, muchos lados derechos de una producción comienzan con componentes léxicos, lo que simplifica la construcción de conjuntos PRIMERO. En la sección 4.4 se da un algoritmo para calcular los conjuntos PRIMERO.

Se deben considerar los conjuntos PRIMERO si hay dos producciones $A \rightarrow \alpha$ y $A \rightarrow \beta$. El análisis sintáctico descendente recursivo sin retroceso requiere que PRIMERO(α) y PRIMERO(β) sean disjuntos. El símbolo de preanálisis se puede usar entonces para decidir el tipo de producción a utilizar; si el símbolo de preanálisis está en PRIMERO(α), entonces se usa α. De otro modo, si el símbolo de preanálisis está en PRIMERO(β), entonces se usa β.

Cuándo se usan las producciones ϵ

Las producciones con ϵ del lado derecho necesitan un tratamiento especial. El analizador sintáctico descendente recursivo usará una producción ϵ por defecto cuando no se pueda aplicar otra producción. Por ejemplo, considérese:

$$\textit{prop} \rightarrow \textbf{begin}\ \textit{props\_opc}\ \textbf{end}$$
$$\textit{props\_opc} \rightarrow \textit{lista\_props} \mid \epsilon$$

Durante el análisis sintáctico de *props_opc*, si el símbolo de preanálisis no está en PRIMERO(*lista_props*), entonces se usa la producción ϵ. Esta elección es justo la correcta si el símbolo de preanalisis es **end**. Cualquier símbolo de preanálisis diferente de **end** dará como resultado un error, que se detectará durante el análisis sintáctico de *prop*.

Diseño de un analizador sintáctico predictivo

Un *analizador sintáctico predictivo* es un programa que consiste en un procedimiento para cada no terminal. Cada procedimiento hace dos cosas:

1. Decide la producción que utilizará analizando el símbolo de preanálisis. Si el símbolo de preanálisis está en PRIMERO(α), se usa la producción con lado derecho α. Si hay un conflicto entre dos lados derechos de cualquier símbolo de preanálisis, entonces en esa gramática no se puede emplear este método de aná-

---

[2] Las producciones con ϵ en el lado derecho complican la determinación de los primeros símbolos generados por un no terminal. Por ejemplo, si el no terminal B deriva la cadena vacía y hay una producción $A \rightarrow BC$, entonces el primer símbolo generado por C puede ser también el primer símbolo generado por A. Si C también puede generar ϵ, entonces tanto PRIMERO (A) como PRIMERO (BC) contienen a ϵ.

lisis sintáctico. Si el símbolo de preanálisis no está en el conjunto PRIMERO de ningún otro lado derecho, se usa una producción con ϵ en el lado derecho.

2. El procedimiento usa una producción imitando al lado derecho. Un no terminal da como resultado una llamada al procedimiento del no terminal, y un componente léxico que coincida con el símbolo de preanálisis da como resultado que se lea el siguiente componente léxico. Si el componente léxico de la producción no coincide en algún punto con el símbolo de preanálisis, se declara un error. La figura 2.17 es el resultado de aplicar estas reglas a la gramática (2.8).

Al igual que se crea un esquema de traducción extendiendo una gramática, se puede crear un traductor dirigido por la sintaxis extendiendo un analizador sintáctico predictivo. En la sección 5.5 se da un algoritmo para este propósito. La siguiente construcción limitada es suficiente por el momento, pues los esquemas de traducción que se implantan en este capítulo no asocian atributos a los no terminales:

1. Constrúyase un analizador sintáctico predictivo, ignorando las acciones en las producciones.
2. Cópiense las acciones del esquema de traducción en el analizador sintáctico. Si una acción aparece después del símbolo gramatical X en la producción p, entonces se copia después del código que implanta X. De otro modo, si aparece al principio de la producción, entonces se copia justo antes del código que implanta la producción.

En la siguiente sección se construirá un traductor de este tipo.

Recursividad por la izquierda

Es posible que un analizador sintáctico descendente recursivo entre en un lazo (bucle) indefinido. Hay un problema con producciones recursivas por la izquierda del tipo

$$expr \rightarrow expr + término$$

en la que el símbolo situado más a la izquierda del lado derecho es el mismo que el no terminal del lado izquierdo de la producción. Supóngase que el procedimiento para *expr* decide aplicar esta producción. El lado derecho comienza con *expr*, de modo que el procedimiento para *expr* se llama recursivamente, y el analizador sintáctico entra en un lazo indefinido. Obsérvese que el símbolo de preanálisis cambia sólo cuando coincide un terminal del lado derecho. Como la producción comienza con el no terminal *expr*, no se realiza ningún cambio en la entrada entre llamadas recursivas, lo cual causa el lazo infinito.

Reescribiendo la producción transgresora, se puede eliminar una producción recursiva por la izquierda. Considérese un no terminal A con dos producciones

$$A \rightarrow A\alpha \mid \beta$$

donde α y β son secuencias de terminales y no terminales que no comienzan con A.

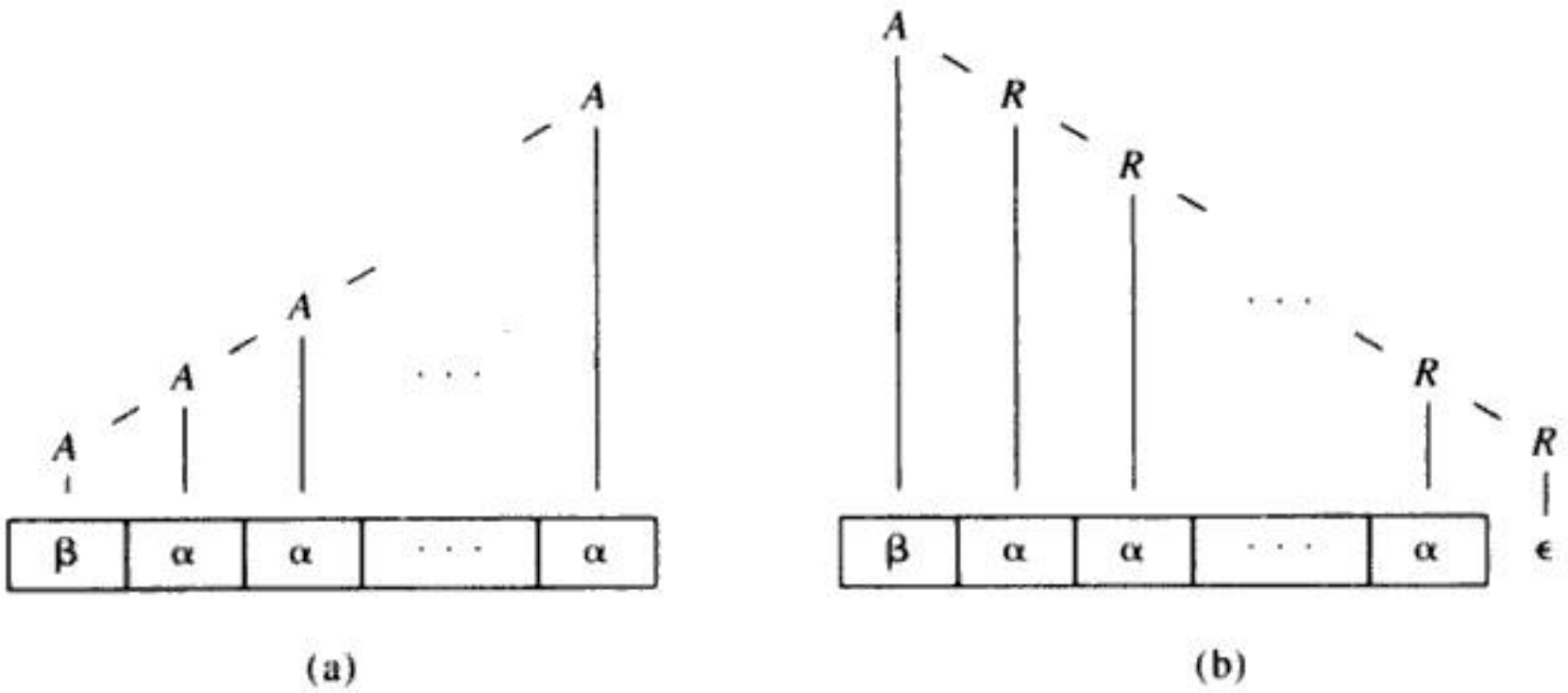


Fig. 2.18. Formas recursivas izquierda y derecha de generar una cadena.

Por ejemplo, en

$$expr \rightarrow expr + término \mid término$$

$A = expr$, $\alpha = + término$, y $\beta = término$.

El no terminal A es *recursivo por la izquierda*, porque la producción $A \rightarrow A\alpha$ tiene la propia A como el símbolo situado más a la izquierda del lado derecho. La aplicación repetida de esta producción forma una secuencia de símbolos α a la derecha de A, como en el caso de la figura 2.18(a). Cuando finalmente se reemplaza A por β, se tiene una β seguida de una secuencia de cero o más α.

El mismo efecto se puede lograr, como en la figura 2.18(b), reescribiendo las producciones para A de la forma siguiente.

$$\begin{aligned} A &\rightarrow \beta\ R \\ R &\rightarrow \alpha\ R \mid \epsilon \end{aligned} \tag{2.11}$$

Aquí, R es un terminal nuevo. La producción $R \rightarrow \alpha\ R$ es *recursiva por la derecha*, porque esta producción para R tiene al propio R como último símbolo del lado derecho. Las producciones recursivas por la derecha forman árboles de crecimiento descendente hacia la derecha, como en la figura 2.18(b). Los árboles de crecimiento descendente hacia la derecha dificultan la traducción de expresiones que contengan operadores asociativos a la izquierda, como el operador *menos*. Sin embargo, en la siguiente sección se explicará también la manera de obtener la traducción apropiada de expresiones a notación postfija mediante un diseño cuidadoso del esquema de traducción, basado en una gramática recursiva por la derecha.

En el capítulo 4 se consideran formas más generales de recursividad por la izquierda y se muestra cómo se puede eliminar de una gramática toda recursividad por la izquierda.

2.5 TRADUCTOR DE EXPRESIONES SIMPLES

Utilizando las técnicas de las tres secciones anteriores, se construirá ahora un traductor dirigido a la sintaxis, en forma de programa operativo en C, el cual traduce expresiones aritméticas a la forma postfija. Con objeto de conservar el programa inicial manejablemente pequeño, se empieza con expresiones compuestas de dígitos separados por los signos más y menos. El lenguaje se amplía en las dos secciones siguientes para incluir números, identificadores y otros operadores. Teniendo en cuenta que las expresiones aparecen como una construcción en tantos lenguajes, vale la pena estudiar en detalle su traducción.

expr → *expr* + *término*	{ *print* ('+') }
expr → *expr* − *término*	{ *print* ('−') }
expr → *término*	
término → 0	{ *print* ('0') }
término → 1	{ *print* ('1') }
...	
término → 9	{ *print* ('9') }

Fig. 2.19. Especificación inicial del traductor infijo a postfijo.

A menudo, un esquema de traducción dirigida a la sintaxis puede servir como especificación de un traductor. El esquema de la figura 2.19 (repetido de la Fig. 2.13) se usa como definición de la traducción que ha de ejecutarse. Muchas veces, se da el caso de tener que modificar la gramática subyadente de un esquema dado antes de poderla analizar por un analizador sintáctico predictivo. En particular, la gramática subyacente al esquema de la figura 2.19 es recursiva por la izquierda, y según se expuso en la última sección, un analizador sintáctico predictivo no puede manejar una gramática recursiva por la izquierda. Eliminando la recursión por la izquierda, se puede obtener una gramática adecuada para ser usada en un traductor descendente recursivo predictivo.

Sintaxis abstracta y sintaxis concreta

Un punto de partida útil para considerar la traducción de una cadena de entrada es un *árbol de sintaxis abstracta*, donde cada nodo representa un operador, y los hijos de ese nodo, los operandos. Por contraste, un árbol de análisis sintáctico se denomina *árbol de sintaxis concreta*, y la gramática subyacente, *sintaxis concreta* del len-

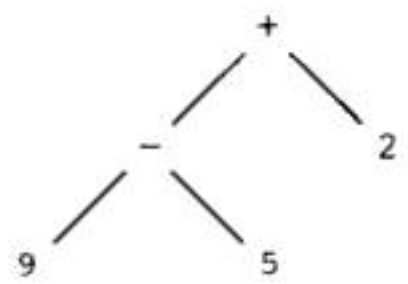


Fig. 2.20. Arbol de sintaxis para 9−5+2.

guaje. Los árboles de sintaxis abstracta, o simplemente *árboles sintácticos* difieren de los árboles de análisis sintáctico en que las distinciones superficiales de forma, sin importancia en la traducción, no aparecen en los árboles sintácticos.

Por ejemplo, el árbol sintáctico para 9−5+2 se muestra en la figura 2.20. Dado que + y − tienen el mismo nivel de precedencia, y los operadores con igual nivel de precedencia se evalúan de izquierda a derecha, el árbol presenta 9−5 agrupado como una subexpresión. Haciendo una comparación de la figura 2.20 con el correspondiente árbol de análisis sintáctico de la figura 2.2, se observa que el árbol sintáctico asocia un operador con un nodo interior, y no hace que el operador sea uno de los hijos.

Es deseable que un esquema de traducción se base en una gramática cuyos árboles de análisis sintáctico se parezcan al máximo a árboles sintácticos. El agrupamiento de subexpresiones hecho por la gramática de la figura 2.19 es similar a su agrupamiento en árboles sintácticos. Desafortunadamente, la gramática de la figura 2.19 es recursiva por la izquierda, y por tanto, no es adecuada para el análisis sintáctico predictivo. En esto, parece existir una contradicción; por una parte, se necesita una gramática que facilite el análisis sintáctico, y por otra, se precisa una gramática radicalmente distinta que facilite la traducción. La solución obvia es eliminar la recursividad por la izquierda. Sin embargo, esto ha de hacerse con cuidado, como se muestra en el siguiente ejemplo.

Ejemplo 2.9. La gramática siguiente no es apropiada para traducir expresiones a la forma postfija, aunque genere exactamente el mismo lenguaje que la gramática de la figura 2.19 y se pueda usar para análisis sintáctico descendente o recursivo.

$$\begin{aligned} \textit{expr} &\rightarrow \textit{término resto} \\ \textit{resto} &\rightarrow + \ \textit{expr} \mid - \ \textit{expr} \mid \epsilon \\ \textit{término} &\rightarrow 0 \mid 1 \mid \ldots \mid 9 \end{aligned}$$

El problema de esta gramática es que los operandos de los operadores generados por $\textit{resto} \rightarrow + \ \textit{expr}$ y $\textit{resto} \rightarrow - \ \textit{expr}$ no resultan evidentes a partir de las producciones. Ninguna de las siguientes elecciones para formar la traducción de *resto.t* a partir de la de *expr.t* es aceptable:

$$\textit{resto} \rightarrow - \ \textit{expr} \{ \textit{resto.t} := \text{'-'} \parallel \textit{expr.t} \} \tag{2.12}$$

$$\textit{resto} \rightarrow - \ \textit{expr} \{ \textit{resto.t} := \textit{expr.t} \parallel \text{'-'} \} \tag{2.13}$$

(Sólo se muestra la producción y la acción semántica para el operador menos.) La traducción de 9−5 es 95−. Sin embargo, utilizando la acción de (2.12), entonces el signo menos aparece antes que *expr.t* y 9−5 queda incorrectamente en la traducción como 9−5.

Por otra parte, usando (2.13) y la regla análoga para el operador más, los operadores se trasladan de manera consistente al extremo derecho y 9−5+2 se traduce incorrectamente a 952+− (la traducción correcta es 95−2+). □

Adaptación del esquema de traducción

La técnica para la eliminación de la recursividad por la izquierda esbozada en la figura 2.18 se puede aplicar también a producciones que contengan acciones semán-

ticas. En la sección 5.5 se amplía la transformación para tener en cuenta atributos sintetizados. La técnica transforma las producciones $A \rightarrow A\,\alpha \mid A\,\beta \mid \gamma$ en

$$A \rightarrow \gamma\, R$$
$$R \rightarrow \alpha\, R \mid \beta\, R \mid \epsilon$$

Cuando las acciones semánticas se intercalan en las producciones, se trasladan junto con la transformación. Aquí, si se hace A = *expr*, α = + *término* { *print* ('+') }, β = − *término* { *print*('−') } y γ = *término*, la transformación anterior produce el esquema de traducción (2.14). Las producciones para *expr* de la figura 2.19 se transformaron en las producciones para *expr* y para el nuevo no terminal *resto* de (2.14). Las producciones para *término* se repiten en la figura 2.19. Fíjese que la gramática subyacente es distinta de la del ejemplo 2.9, y la diferencia hace posible la traducción que se desea.

```
expr    → término resto
resto   → + término { print ('+') } resto | − término { print('−') } resto | ϵ
término → 0 { print ('0') }
término → 1 { print ('1') }                                              (2.14)
   . . .
término → 9 { print('9') }
```

La figura 2.21 muestra cómo se traduce `9−5+2` utilizando la gramática anterior.

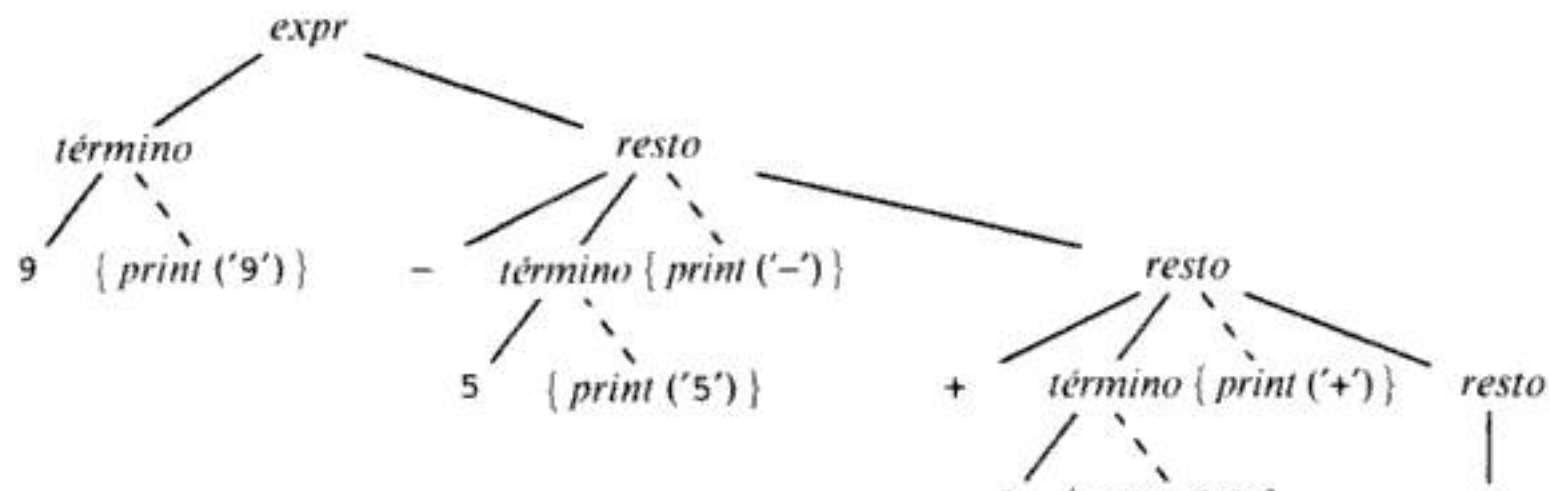


Fig. 2.21. Traducción de `9−5+2` a `95−2+`.

Procedimientos para los no terminales *expr*, *término* **y** *resto*

Ahora se aplica un traductor en C utilizando el esquema de traducción dirigida por la sintaxis (2.14). La esencia del traductor es el código en C de la figura 2.22 para las funciones `expr`, `término` y `resto`. Estas funciones aplican los correspondientes no terminales de (2.14).

La función `parea`, que se presenta más adelante, es la contraparte en C del código de la figura 2.17 para parear un componente léxico con el símbolo de preanálisis y avanzar por la entrada. Puesto que cada componente léxico es un solo carácter en este lenguaje, `parea` puede hacerse comparando y leyendo caracteres.

```
expr()
{
        término(); resto();
}

resto()
{
        if (preanálisis == '+') {
            parea('+'); término(); putchar('+'); resto();
        }
        else if (preanálisis) == '-') {
            parea('-'); término(); putchar('-'); resto();
        }
        else ;
}

término()
{
        if (isdigit(preanálisis)) {
             putchar(preanálisis); parea(preanálisis);
        }
        else error();
}
```

Fig. 2.22. Funciones para los no terminales *expr*, *resto* y *término*.

Para los lectores que no estén familiarizados con el lenguaje de programación C, se mencionan las principales diferencias entre C y otros lenguajes derivados de ALGOL, como Pascal, conforme se encuentren usos a las características de C. Un programa en C está constituido por una secuencia de definiciones de funciones, donde la ejecución comienza en una función distinguida llamada `main`. Las definiciones de funciones no se pueden anidar. Los paréntesis que encierran a los parámetros de una función son necesarios, aunque no haya parámetros; por tanto, se escribe `expr()`, `término()` y `resto()`. Las funciones se comunican por el paso de parámetros "por valor" o por el acceso a datos globales a todas las funciones. Por ejemplo, las funciones `término()` y `resto()` revisan el símbolo de preanálisis utilizando el identificador global `preanálisis`.

Los lenguajes C y Pascal usan los siguientes símbolos para asignaciones y pruebas de igualdad:

OPERACIÓN	C	PASCAL
asignación	=	:=
prueba de igualdad	==	=
prueba de desigualdad	!=	<>

Las funciones para los no terminales imitan a los lados derechos de las producciones. Por ejemplo, la producción *expr* → *término resto* se aplica para las llamadas a `término()` y `resto()` en la función `expr()`.

Otro ejemplo, la función `resto()` utiliza la primera producción para *resto* en (2.14) si el símbolo de preanálisis es un signo más; utiliza la segunda producción si el símbolo de preanálisis es menos, y utiliza la producción *resto* → ϵ por omisión. La primera producción para *resto* queda implantada por la primera proposición `if` de la figura 2.22. Si el símbolo de preanálisis es +, el signo más se parea mediante la llamada `parea('+')`. Después de la llamada a `término()`, la rutina de biblioteca estándar de C `putchar('+')` implanta la acción semántica mediante la impresión de un carácter "más". Como la tercera producción para *resto* tiene ϵ como lado derecho, el último `else` de `resto()` no hace nada.

Las diez producciones para *término* generan los diez dígitos. En la figura 2.22, la rutina `isdigit` prueba si el símbolo de preanálisis es un dígito. Si el resultado es positivo, se imprime y se parea el dígito; de otro modo, aparece un error. (Obsérvese que `parea` modifica al símbolo de preanálisis, de modo que la impresión debe producirse antes de emparejar el dígito.) Antes de mostrar un programa completo, se hará una transformación para mejorar la velocidad del código de la figura 2.22.

Optimación del traductor

Ciertas llamadas recursivas se pueden reemplazar por iteraciones. Cuando la última proposición ejecutada en el cuerpo de un procedimiento es una llamada recursiva al mismo procedimiento, se dice que la llamada es *recursiva por el final*. Por ejemplo, las llamadas de `resto()` al final de la cuarta y séptima líneas de la función `resto()` son recursivas por el final, porque el control fluye hacia el final del cuerpo de la función después de cada una de esas llamadas.

Se puede imprimir mayor velocidad a un programa reemplazando la recursión por el final con una construcción de iteración. Para un procedimiento sin parámetros, una llamada recursiva por el final simplemente se puede reemplazar por un salto al inicio del procedimiento. El código de `resto` se puede reescribir como:

```
resto()
{
L:      if (preanálisis == '+') {
            parea('+'); término(); putchar('+'); goto L;
        }
        else if (preanálisis == '-') {
            parea('-'); término(); putchar('-'); goto L;
        }
        else ;
}
```

Mientras el símbolo de preanálisis sea un signo más o menos, el procedimiento `resto` parea el signo, llama a `término` para que paree un dígito, y repite el proceso. Adviértase que, como `parea` elimina el signo cada vez que es llamado, este lazo ocurre sólo en secuencias alternantes de signos y dígitos. Si se hace este cambio

en la figura 2.22, la única llamada que queda de `resto` es desde `expr` (véase la línea 3). Por tanto, las dos funciones se pueden integrar en una sola, como en la figura 2.23. En C, una proposición *prop* se puede ejecutar repetidamente escribiendo

```
while(1) prop
```

porque la condición `1` siempre es verdadera. Se puede salir de un ciclo ejecutando una proposición de interrupción de ciclo, la proposición `break`. La forma estilizada del código de la figura 2.23 permite añadir oportunamente otros operadores.

```
expr()
{
        término();
        while(1)
                if (preanálisis == '+') {
                        parea('+'); término(); putchar('+');
                }
                else if (preanálisis == '-') {
                        parea('-'); término(); putchar('-');
                }
                else break;
}
```

Fig. 2.23. Reemplazo de las funciones `expr` y `resto` de la figura 2.22.

El programa completo

El programa completo en C para el traductor se muestra en la figura 2.24. La primera línea, que comienza con `#include`, carga `<ctype.h>`, que es un archivo de rutinas estándar que contiene el código del predicado `isdigit`.

Los componentes léxicos, que consisten en caracteres simples, los proporciona la rutina de biblioteca estándar `getchar`, que lee el siguiente carácter del archivo de entrada. Sin embargo, `preanálisis` está declarado como entero en la línea 2 de la figura 2.24 para prever los componentes léxicos adicionales, que no son caracteres simples y que se introducirán en secciones posteriores. Ya que `preanálisis` está declarado fuera de cualquiera de las funciones, es global a cualesquiera funciones que se definan después de la línea 2 de la figura 2.24.

La función `parea` revisa componentes léxicos; lee el siguiente componente léxico de la entrada si el símbolo de preanálisis concuerda, y llama a la rutina de error en otro caso.

La función `error` utiliza la función de biblioteca estándar `printf` para imprimir el mensaje `"error de sintaxis"`, y después termina la ejecución mediante la llamada `exit(1)` a otra función de biblioteca estándar.

2.6 ANALISIS LEXICO

Ahora se puede añadir al traductor de la sección anterior un analizador léxico que lea y convierta la entrada en una cadena de componentes léxicos para el analizador sintáctico. Recuérdese de la definición de una gramática (Sec. 2.2) que las frases de

```
#include <ctype.h> /* carga el archivo que contiene al
                      predicado isdigit */
int preanálisis;

main()
{
     preanálisis = getchar();
     expr();
     putchar('\n'); /* agrega un carácter de línea nueva al final */
}

expr()
{
     término()
     while(1)
          if (preanálisis == '+') {
               parea('+'); término(); putchar('+');
          }
          else if (preanálisis == '-') {
               parea('-'); término(); putchar('-');
          }
          else break;
}
término()
{
     if (isdigit(preanálisis)) {
          putchar(preanálisis);
          parea(preanálisis);
     }
     else error();
}
parea(t)
     int t;
{
     if (preanálisis == t)
          preanálisis = getchar();
     else error();
}

error()
{
     printf("error de sintaxis\n"); /* imprime mensaje de error */
     exit(1);                       /* y después se detiene */
}
```

Fig. 2.24. Programa en C para traducir una expresión de la forma infija a la forma postfija.

un lenguaje constan de una cadena de componentes léxicos. Una secuencia de caracteres de entrada que comprenda un solo componente léxico se llama lexema. Un analizador léxico puede aislar un analizador sintáctico de la representación en lexemas de los componentes léxicos. Se empieza haciendo una lista de algunas de las funciones para que las realice un analizador léxico.

Eliminación de espacios en blanco y comentarios

El traductor de expresiones de la sección anterior reconoce todos los caracteres de la entrada, de modo que los caracteres extraños, como los espacios en blanco, harán que falle. Muchos lenguajes permiten que aparezcan "espacios en blanco" (caracteres en blanco, caracteres TAB y de nueva línea) entre los componentes léxicos. Los comentarios también pueden no ser considerados por el analizador sintáctico y el traductor, por tanto también se pueden tratar como espacios en blanco.

Si el analizador léxico elimina los espacios en blanco, el analizador sintáctico nunca tendrá que considerarlos. La alternativa de modificar la gramática para incorporar los espacios en blanco dentro de la situación no es tan fácil de implantar.

Constantes

En cualquier momento que aparece un dígito solo en una expresión, parece razonable poner una constante entera arbitraria en su lugar. Como una constante entera es una secuencia de dígitos, pueden admitirse constantes enteras añadiendo producciones a la gramática de las expresiones o creando un componente léxico para tales constantes. La tarea de agrupar dígitos para formar enteros se le asigna, por lo general, a un analizador léxico, porque los números se pueden tratar como unidades simples durante la traducción.

Sea **núm** el componente léxico que representa un entero. Cuando una secuencia de dígitos aparece en la cadena de entrada, el analizador léxico pasará **núm** al analizador sintáctico. El valor del entero se pasará como atributo del componente léxico **núm**. Lógicamente, el analizador léxico pasa al componente léxico y el atributo al analizador sintáctico. Al escribir un componente léxico y su atributo como una tupla encerrada entre < >, la entrada

```
31 + 28 + 59
```

se transforma en la secuencia de tuplas

<**núm**, 31> <+, > <**núm**, 28> <+, > <**núm**, 59>

El componente léxico + no tiene atributos. Los segundos componentes de las tuplas, los atributos, no desempeñan papel alguno durante el análisis sintáctico, pero son necesarios en la traducción.

Reconocimiento de identificadores y palabras clave

Los lenguajes utilizan identificadores como nombres de variables, matrices, funciones y similares. A menudo, una gramática para un lenguaje trata a un identificador como un componente léxico. Un analizador basado en esta gramática espera ver el

mismo componente léxico, por ejemplo, **id**, cada vez que un identificador aparezca en la entrada. Por ejemplo, la entrada

```
cuenta = cuenta + incremento;
```
(2.15)

sería convertida por el analizador léxico en la cadena de componentes léxicos

id = id + id ; (2.16)

Esta cadena de componentes léxicos se utiliza en el análisis sintáctico.

Cuando se considera el análisis léxico de la línea de entrada (2.15), es útil distinguir entre el componente léxico **id** y los lexemas `cuenta` e `incremento` asociados con los casos de este componente léxico. El traductor necesita saber que el lexema `cuenta` forma los dos primeros casos de **id** en (2.16) y que el lexema `incremento` forma el tercer caso de **id**.

Cuando en la entrada aparece un lexema que forma un identificador, se necesita algún mecanismo para determinar si el lexema apareció antes. Para tal mecanismo se usa una tabla de símbolos, como ya se mencionó en el capítulo 1. El lexema se almacena en la tabla de símbolos y un apuntador a esa entrada de la tabla de símbolos se convierte en un atributo del componente léxico **id**.

Muchos lenguajes utilizan cadenas de caracteres fijas, como `begin`, `end`, `if`, y otras más, como signos de puntuación o para identificar ciertas construcciones. Estas cadenas de caracteres, llamadas *palabras clave*, suelen satisfacer las reglas para formar identificadores, por lo que se necesita un mecanismo para decidir cuándo un lexema forma una palabra clave y cuándo forma un identificador. El problema resulta más fácil de resolver si las palabras clave son *reservadas*, es decir, si no se pueden usar como identificadores. Entonces, una cadena de caracteres forma un identificador sólo si no es una palabra clave.

El problema del aislamiento de los componentes léxicos también surge si aparecen los mismos caracteres en los lexemas de más de un componente léxico, como en <, <= y <> en Pascal. En el capítulo 3 se estudian técnicas eficientes de reconocimiento de tales componentes léxicos.

Interfaz con el analizador léxico

Cuando entre el analizador sintáctico y la cadena de entrada se inserta un analizador léxico, éste interactúa con los dos como se indica en la figura 2.25. Lee los caracteres de la entrada, los agrupa en lexemas y pasa los componentes léxicos formados por los lexemas, junto con los valores de sus atributos, a las etapas posteriores del compilador. En algunas situaciones, el analizador léxico tiene que leer algunos caracteres por adelantado antes de poder decidir qué componente léxico va a devolver al analizador sintáctico. Por ejemplo, un analizador léxico en Pascal tiene que leer más adelante después de ver el carácter >. Si el siguiente carácter es =, entonces la secuencia de caracteres >= es el lexema que forma el componente léxico para el operador "mayor o igual que". De otro modo, > es el lexema que forma el operador "mayor que", y el analizador léxico ha leído un carácter de más. El carácter adicional tiene que devolverse a la entrada, porque puede ser el inicio del siguiente lexema de la entrada.

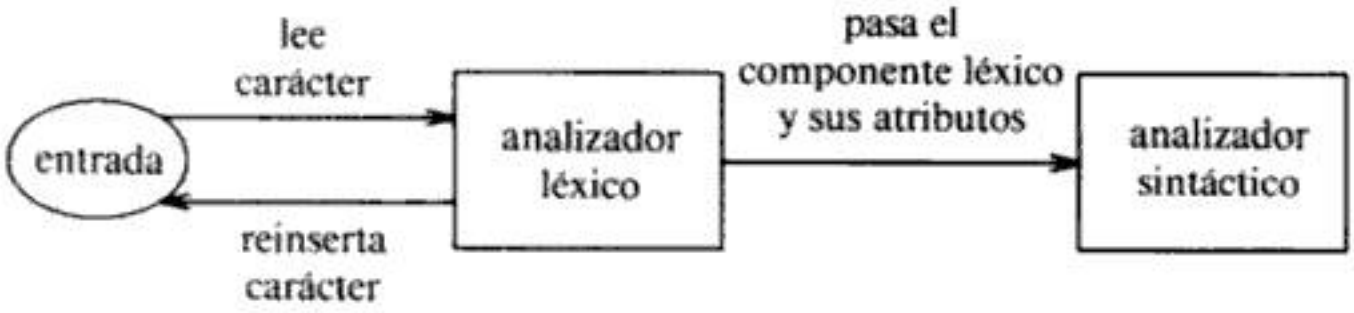


Fig. 2.25. Inserción de un analizador léxico entre la entrada y el analizador sintáctico.

El analizador léxico y el analizador sintáctico forman un par *productor-consumidor*. El analizador léxico produce componentes léxicos y el analizador sintáctico los consume. Los componentes léxicos producidos se pueden conservar en un *buffer* hasta ser consumidos. La interacción entre los dos sólo está restringida por el tamaño del *buffer*, puesto que el analizador léxico no puede avanzar cuando el *buffer* está lleno y el analizador sintáctico no puede proseguir cuando el *buffer* está vacío. Por lo general, el *buffer* contiene sólo un componente léxico. En este caso, la interacción se puede aplicar simplemente haciendo que el analizador léxico sea un procedimiento llamado por el analizador sintáctico, que devuelva componentes léxicos cuando se le pidan.

La aplicación de la lectura y devolución de caracteres suele hacerse estableciendo un *buffer* de entrada. Un bloque de caracteres se lee al *buffer* de una vez; un apuntador señala la porción de la entrada ya analizada. La operación de devolución de carácter se implanta moviendo hacia atrás el apuntador. También puede ser necesario tener que guardar los caracteres de la entrada para el informe de errores, pues hay que dar alguna indicación de dónde se produjo el error en el texto de entrada. Aunque sólo sea por razones de eficiencia, está justificado el manejo del *buffer* para los caracteres de entrada. La extracción de un bloque de caracteres suele ser más eficiente que la extracción de un carácter a la vez. En la sección 3.2 se estudian las técnicas para el manejo de los *buffer* de entrada.

Un analizador léxico

Ahora se construirá un analizador léxico rudimentario para el traductor de expresiones de la sección 2.5. El propósito del analizador léxico es permitir que aparezcan espacios en blanco y números dentro de las expresiones. En la siguiente sección, se amplía el analizador léxico para admitir también identificadores.

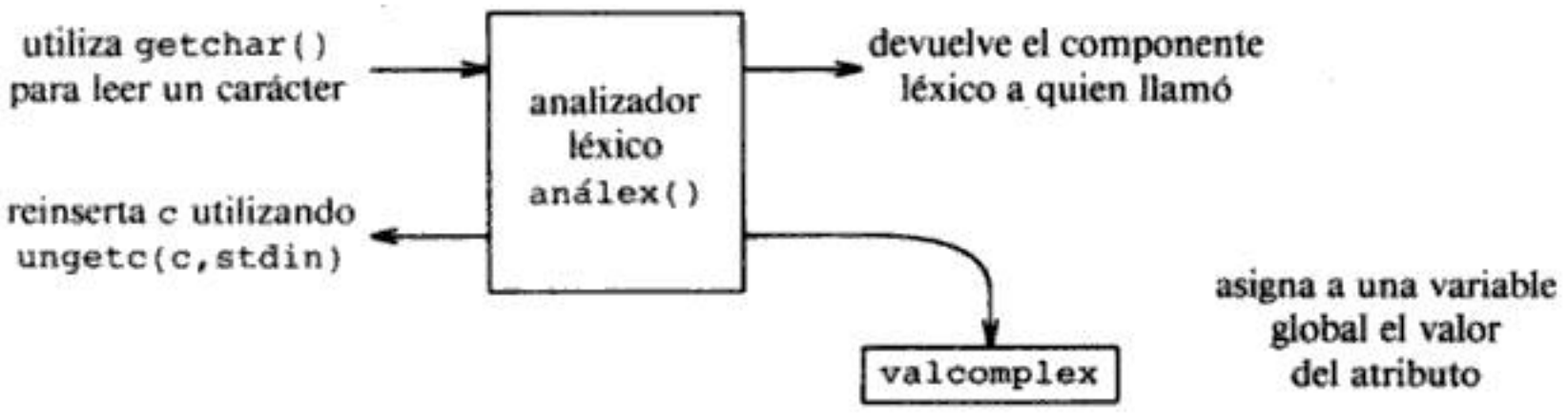


Fig. 2.26. Implantación de las interacciones de la figura 2.25.

La figura 2.26 sugiere cómo el analizador léxico realiza las interacciones de la figura 2.25, escrito como la función `análex` en C. Las rutinas `getchar` y `ungetc` del archivo incluido estándar `<stdio.h>` se encargan del manejo del *buffer* de entrada; `análex` lee y devuelve los caracteres de la entrada llamando a las rutinas `getchar` y `ungetc`, respectivamente. Si `c` se declara como carácter, el par de proposiciones

```
c = getchar(); ungetc(c, stdin);
```

no altera la cadena de entrada. La llamada a `getchar` asigna el siguiente carácter de la entrada a `c`; la llamada a `ungetc` devuelve el valor de `c` a la entrada estándar `stdin`.

Si el lenguaje de implantación no permite que las funciones devuelvan estructuras de datos, entonces los componentes léxicos y sus atributos se tienen que pasar por separado. La función `análex` devuelve un entero que es el código de un componente léxico. El componente léxico para un carácter puede ser cualquier entero convencional que sea el código de ese carácter. Un componente léxico, como **núm**, se puede codificar entonces por un entero mayor que cualquier entero que codifique un carácter, por ejemplo, `256`. Para permitir que la codificación se pueda modificar con facilidad, se usa una constante simbólica `NUM` para hacer referencia a la codificación con un número entero de **núm**. En Pascal, la asociación entre `NUM` y la codificación se pueden hacer mediante una declaración **const**; en C, se puede hacer que `NUM` represente al entero `256` mediante una proposición `define`:

```
#define NUM 256
```

La función `análex` devuelve `NUM` cuando se observe una secuencia de dígitos en la entrada. A la variable global `valcomplex` se le asigna la secuencia de dígitos que forma el valor del componente léxico. Así, si un `7` va seguido inmediatamente de un `6` en la entrada, a `valcomplex` se le asigna el valor entero `76`.

Al admitir números dentro de las expresiones se requiere una modificación de la gramática de la figura 2.19. Se reemplazan los dígitos individuales por el no terminal *factor* y se introducen las producciones y acciones semánticas siguientes:

factor → (*expr*)
| **núm** { *print* (**núm**.*valor*) }

El código en C para *factor* de la figura 2.27 es una implantación directa de las producciones anteriores. Cuando `preanálisis` es igual a `NUM`, el valor del atributo **núm**.*valor* está dado por la variable global `valcomplex`. La acción de imprimir este valor la realiza la función de biblioteca estándar `printf`. El primer argumento de `printf` es una cadena entre comillas que especifica el formato que se ha de usar en la impresión de los restantes argumentos. En el lugar donde `%d` aparece en la cadena, se imprime la representación decimal del siguiente argumento. Por tanto, la proposición `printf` de la figura 2.27 imprime un espacio en blanco seguido de la representación decimal de `valcomplex` y seguido de otro espacio en blanco.

En la figura 2.28 se muestra la implantación de la función `análex`. Cuando se ejecuta el cuerpo de la proposición `while` en las líneas 8 a 28, se lee un carácter en `t` en la línea 9. Si el carácter es un espacio en blanco o un TAB (escrito `'\t'`), enton-

```
factor()
{
     if (preanálisis == '(') {
          parea('('); expr(); parea(')');
     }
     else if (preanálisis == NUM) {
          printf(" %d ", valcomplex); parea(NUM);
     }
     else error();
}
```

Fig. 2.27. Código en C para *factor* cuando los operandos pueden ser números.

```
(1)  #include <stdio.h>
(2)  #include <ctype.h>
(3)  int numlínea = 1;
(4)  int valcomplex = NINGUNO;

(5)  int análex()
(6)  {
(7)       int t;
(8)       while(1) {
(9)            t = getchar();
(10)           if (t == ' ' || t == '\t')
(11)                ;     /* elimina espacios en blanco y símbolos tab */
(12)           else if (t == '\n')
(13)                numlínea = numlínea + 1;
(14)           else if (isdigit(t)) {
(15)                valcomplex = t - '0';
(16)                t = getchar ();
(17)                while (isdigit(t)) {
(18)                     valcomplex = valcomplex*10 + t-'0';
(19)                     t = getchar();
(20)                }
(21)                ungetc(t, stdin);
(22)                return NUM;
(23)           }
(24)           else {
(25)                valcomplex = NINGUNO;
(26)                return t;
(27)           }
(28)      }
(29) }
```

Fig. 2.28. Código en C para el analizador léxico que elimina espacios en blanco y reconoce números.

ces no se devuelve ningún componente léxico al analizador sintáctico; simplemente se inicia de nuevo el lazo `while`. Si el carácter es de línea nueva (escrito `'\n'`), entonces se incrementa la variable global `númlin`, con lo cual se controlan los números de línea de la entrada, pero, de nuevo, no se devuelve ningún componente léxico. Proporcionando un número de línea con un mensaje de error se ayuda a localizar los errores.

El código para la lectura de una secuencia de dígitos está en las líneas 14 a 23. El predicado `isdigit(t)` del archivo incluido `<ctype.h>` se usa en las líneas 14 y 17 para determinar si un carácter de entrada `t` es un dígito. Si lo es, entonces su valor entero está dado por la expresión `t-'0'` en los códigos ASCII y EBCDIC. Con otros conjuntos de caracteres, tal vez se necesite hacer la conversión de modo distinto. En la sección 2.9 se incorpora este analizador léxico al traductor de expresiones.

2.7 INCORPORACION DE UNA TABLA DE SIMBOLOS

Para almacenar información en varias construcciones del lenguaje fuente, se usa por lo general una estructura de datos llamada tabla de símbolos. La información se reúne en las fases de análisis del compilador y la emplea la fase de síntesis para generar el código objeto. Por ejemplo, durante el análisis léxico, la cadena de caracteres, o lexema, que forma un identificador se guarda en una entrada de la tabla de símbolos. Las fases posteriores del compilador pueden añadir a esta entrada información, como el tipo del identificador, su uso (por ejemplo, procedimiento, variable o etiqueta) y su posición en la memoria. La fase de generación de código usaría después esta información para generar el código apropiado para almacenar y acceder a esta variable. En la sección 7.6, se estudia en detalle la implantación y uso de tablas de símbolos, y se ilustra cómo el analizador léxico de la sección anterior puede interactuar con una tabla de símbolos.

La interfaz de la tabla de símbolos

Las rutinas de la tabla de símbolos se refieren principalmente a la protección y recuperación de lexemas. Cuando se guarda un lexema, también se guarda el componente léxico asociado con él. Sobre la tabla de símbolos se realizarán las siguientes operaciones:

`inserta(s,t)`: Devuelve el índice de la nueva entrada para la cadena `s` y el componente léxico `t`.

`busca(s)`: Devuelve el índice de la entrada para la cadena `s`, o `0` si no encontró a `s`.

El analizador léxico utiliza la operación `busca` para determinar si hay o no una entrada para un lexema en la tabla de símbolos. Si no existe entrada, entonces utiliza la operación `inserta` para crear una. Se estudiará una implantación en la que tanto el analizador léxico como el analizador sintáctico conocen el formato de las entradas de la tabla de símbolos.

Manejo de palabras clave reservadas

Las rutinas anteriores de la tabla de símbolos pueden manejar cualquier conjunto de palabras clave reservadas. Por ejemplo, considérense los componentes léxicos **div** y **mod** con lexemas `div` y `mod`, respectivamente. Se puede inicializar la tabla de símbolos mediante las llamadas

```
inserta("div", div);
inserta("mod", mod);
```

Cualquier llamada posterior `busca("div")` devuelve el componente léxico **div**, de modo que `div` no puede usarse como identificador.

Cualquier conjunto de palabras clave reservadas se puede manejar de esta forma inicializando apropiadamente la tabla de símbolos.

Una implantación de una tabla de símbolos

En la figura 2.29 se esboza una estructura de datos para una implantación particular de una tabla de símbolos. No se desea asignar una cantidad fija de espacio para guardar los lexemas que forman los identificadores; una cantidad fija de espacio puede no ser suficientemente grande para guardar un identificador muy largo y puede ser innecesariamente grande para un identificador corto como `i`. En la figura 2.29, una matriz separada `lexemas` guarda la cadena de caracteres que forma un identificador. La cadena está terminada mediante un carácter de fin-de-cadena, representado por `FDC`, y que no pueda aparecer en los identificadores. Cada entrada en la matriz `tablasimb` de la tabla de símbolos es un registro que contiene dos campos, `aplex`, que apunta al principio de un lexema, y `complex`. Los campos adicionales pueden contener valores de atributos, aunque aquí no se hará eso.

En la figura 2.29 la entrada 0 se deja vacía, porque `busca` devuelve `0` para indicar que no hay entrada para una cadena. La primera y segunda entradas son para las palabras clave `div` y `mod`. La tercera y la cuarta son para los identificadores `cuenta` e `i`.

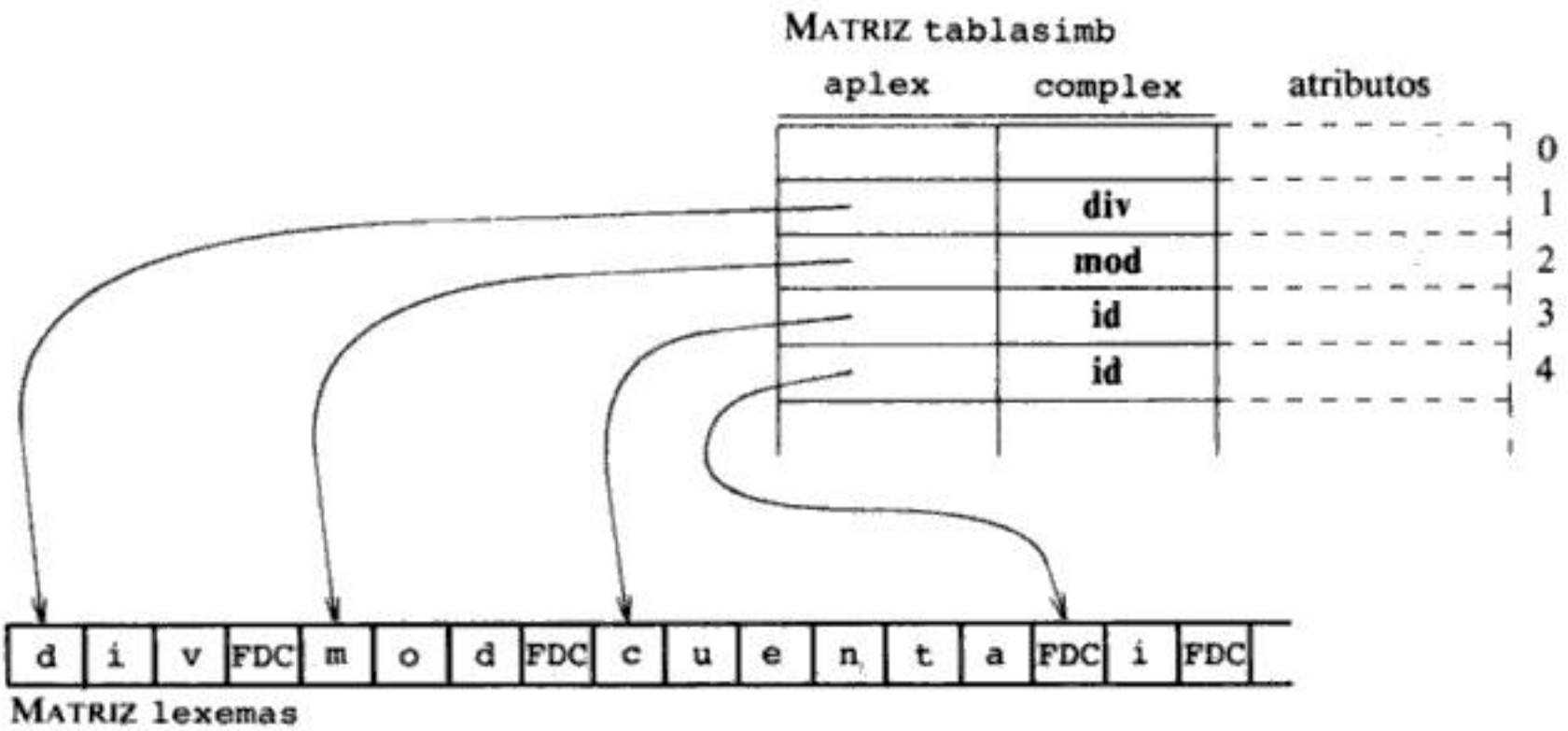


Fig. 2.29. Tabla de símbolos y matriz para el almacenamiento de las cadenas.

En la figura 2.30 se muestra el seudocódigo de un analizador léxico que maneje identificadores; en la sección 2.9 se introduce una aplicación en C. El analizador léxico maneja los espacios en blanco y las constantes enteras de la misma forma que en la figura 2.28 de la sección anterior.

Cuando este analizador léxico lee una letra, comienza a guardar letras y dígitos en un *buffer*, `buflex`. La cadena recogida en `buflex` se busca después en la tabla de símbolos, utilizando la operación `busca`. Como la tabla de símbolos está inicializada con entradas para las palabras clave `div` y `mod`, como se muestra en la figura 2.29, la operación `busca` encontrará estas entradas si `buflex` contiene `div` o `mod`. Si no hay una entrada para la cadena que tenga `buflex`, esto es, `busca` devuelve `0`, entonces `buflex` contiene un lexema para un identificador nuevo. Mediante `inserta`, se crea una entrada para el identificador nuevo. Después de hacer la inserción, `p` es el índice de la entrada de la tabla de símbolos para la cadena `buflex`. Este índice se le comunica al analizador sintáctico asignando `p` a `valcomplex`, y en el campo `complex` de la entrada se devuelve el componente léxico.

Cuando no se presentan esos casos, la acción consiste en devolver el entero que codifica el carácter como componente léxico. Como aquí los componentes léxicos de un solo carácter no tienen atributos, a `valcomplex` se le asigna `NINGUNO`.

```
function análex: integer;
var     buflex: array [0..100] of char;
        c:      char;
begin
    loop begin
        lee un carácter en c;
        if c es un espacio en blanco o un símbolo tab then
            no hacer nada;
        else if c es un carácter de línea nueva then
            numlínea := numlínea + 1
        else if c es un dígito then begin
            asignar a valcomplex el valor de éste y los dígitos siguientes;
            return NUM
        end
        else if c es una letra then begin
            poner c y las letras y dígitos sucesivos en buflex;
            p := busca(buflex);
            if p = 0 then
                p := inserta(buflex, ID);
            valcomplex := p;
            return el campo complex de la entrada p de la tabla
        end
        else begin  /* el componente léxico tiene un solo carácter */
            asignar NINGUNO a valcomplex;  /* no hay atributo */
            return el número entero del código del carácter c
        end
    end
end;
```

Fig. 2.30. Seudocódigo de un analizador léxico.

2.8 MAQUINAS DE PILA ABSTRACTAS

La etapa inicial de un compilador construye una representación intermedia del programa fuente a partir de la cual la etapa final genera el programa objeto. Una forma común de representación intermedia es el código para una máquina de pila abstracta. Como se mencionó en el capítulo 1, la división de un compilador en una etapa inicial y una etapa final facilita su modificación para que funcione en una nueva máquina.

En esta sección se presenta una máquina de pila abstracta y se muestra cómo se puede generar su código. La máquina tiene memorias independientes para las instrucciones y los datos, y todas las operaciones aritméticas se realizan con los valores en una pila. Las instrucciones son bastante limitadas y están comprendidas en tres clases: aritmética entera, manipulación de la pila y flujo de control. En la figura 2.31 se representa la máquina. El apuntador *cp* indica la instrucción que se va a ejecutar. Los significados de las instrucciones se estudiarán un poco más adelante.

Instrucciones aritméticas

La máquina abstracta ha de implantar cada operador en el lenguaje intermedio. Una operación básica, como la adición o la sustracción, está soportada directamente por la máquina abstracta. Sin embargo, una operación más compleja, puede tener que ser implantada como una secuencia de instrucciones de la máquina abstracta. La descripción de la máquina se simplifica suponiendo que hay una instrucción para cada operador aritmético.

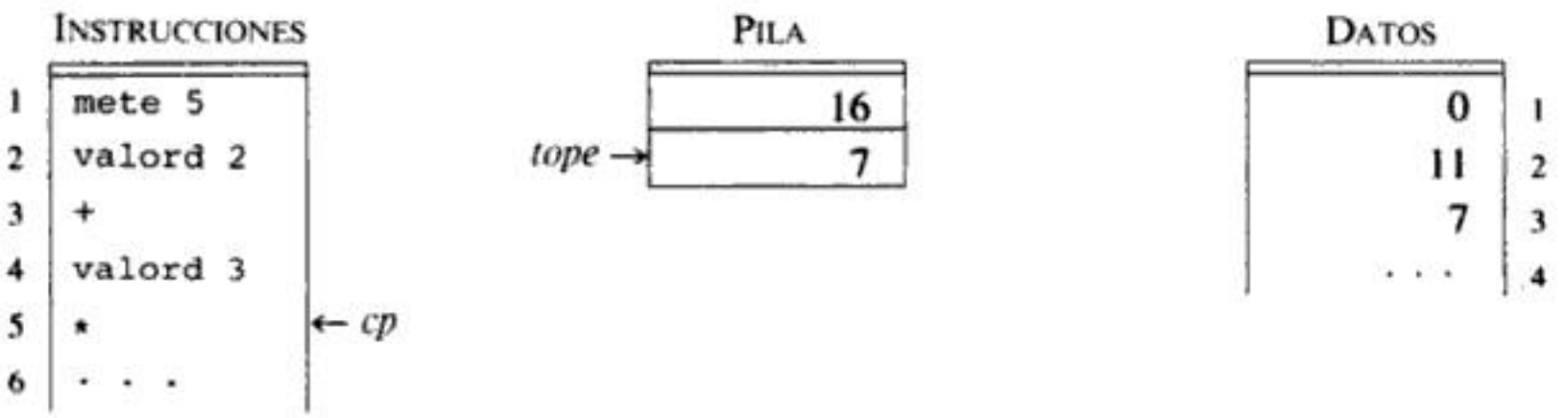


Fig. 2.31. Estado de la máquina de pila después de ejecutarse las primeras cuatro instrucciones.

El código de la máquina abstracta de una expresión aritmética simula la evaluación de una representación postfija de esa expresión utilizando una pila. La evaluación se realiza procesando la representación postfija de izquierda a derecha, insertando los operandos en la pila a medida que los encuentra. Cuando se encuentra un operador k-ario, su argumento situado más a la izquierda está k-1 posiciones por debajo de la cima de la pila y su argumento más a la derecha está en la cima. La evaluación aplica el operador a los k valores de la cima de la pila, extrae los operandos e inserta el resultado en la pila. Por ejemplo, en la evaluación de la expresión postfija `1 3 + 5 *`, se realizan las acciones siguientes:

1. Insertar 1 en la pila.
2. Insertar 3 en la pila.
3. Sumar los dos elementos de la cima, extraerlos e insertar en la pila el resultado 4.
4. Insertar 5 en la pila.
5. Multiplicar los dos elementos de la cima, extraerlos e insertar en la pila el resultado 20.

Al final, el valor de la cima de la pila (aquí es 20) corresponde a la expresión completa.

En el lenguaje intermedio, todos los valores serán enteros, donde `0` corresponde a `false` y los enteros distintos de cero corresponden a `true`. Los operadores booleanos `and` y `or` requieren la evaluación de sus dos argumentos.

Valores de lado izquierdo y de lado derecho

Hay diferencia entre el significado de los identificadores del lado izquierdo y los del lado derecho de una asignación. En cada una de las asignaciones

```
i := 5;
i := i + 1;
```

el lado derecho especifica un valor entero, mientras que el lado izquierdo especifica dónde se va a almacenar el valor. De manera similar, si `p` y `q` son apuntadores a caracteres, y

```
p↑ := q↑;
```

el lado derecho `q↑` especifica un carácter, mientras que `p↑` especifica dónde se va a almacenar el carácter. Los términos *valor de lado izquierdo* y *valor de lado derecho* hacen referencia a los valores apropiados para los lados izquierdo y derecho, respectivamente, de una asignación. Esto es, los valores de lado derecho suelen considerarse como "valores", mientras que los valores de lado izquierdo son posiciones.

Manipulación de la pila

Además de las instrucciones obvias para insertar una constante entera en la pila y extraer un valor de la cima de la pila, hay instrucciones para acceder a la memoria de datos:

`inserta` *v*	inserta *v* en la pila
`valord` *l*	inserta el contenido de la posición de datos *l*
`valori` *l*	inserta la dirección de la posición de datos *l*
`saca`	elimina el valor de la cima de la pila
`:=`	el valor de lado derecho de la cima se pone en el valor de lado izquierdo que está debajo y se extraen los dos
`copia`	inserta una copia del valor de la cima en la pila

Traducción de expresiones

El código que evalúa una expresión en una máquina de pila está estrechamente relacionado con la notación postfija para esa expresión. Por definición, la forma post-

fija de la expresión $E + F$ es la concatenación de la forma postfija de E, la forma postfija de F y $+$. Del mismo modo, el código de la máquina de pila que sirve para evaluar $E + F$ es la concatenación del código para evaluar E, el código para evaluar F y la instrucción para sumar sus valores. La traducción de expresiones al código de la máquina de pila puede hacerse, por tanto, adaptando los traductores de las secciones 2.6 y 2.7.

Aquí se genera código de pila para las expresiones en las que las localidades de datos se direccionan simbólicamente. (La asignación de localidades de datos para los identificadores se estudia en el Cap. 7.) La expresión `a+b` se traduce a:

```
valord a
valord b
+
```

Expresado en palabras: insertar los contenidos de las localidades de datos para `a` y `b` en la pila; después, extraer los dos valores de la cima de la pila, sumarlos e insertar el resultado en la pila.

La traducción de asignaciones a código de máquina de pila se hace como sigue: el valor de lado izquierdo asignado al identificador se inserta en la pila, se evalúa la expresión y su valor de lado derecho se asigna al identificador. Por ejemplo, la asignación

```
día := (1461*a) div 4 + (153*m + 2) div 5 + d          (2.17)
```

se traduce al código de la figura 2.32.

```
valori día          mete 2
mete 1461           +
valord a            mete 5
*                   div
mete 4              +
div                 valord d
mete 153            +
valord m            :=
*
```

Fig. 2.32. Traducción de `día := (1461*a) div 4 + (153*m+2) div 5 + d.`

Estos comentarios se pueden expresar formalmente como sigue. Cada no terminal tiene un atributo t que da su traducción. El atributo *lexema* de **id** proporciona la representación en forma de cadena del identificador.

prop → **id** := *expr*
{ *prop.t* := `'valori'` ‖ **id**.*lexema* ‖ *expr.t* ‖ `':='` }

Flujo de control

La máquina de pila ejecuta instrucciones en una secuencia numérica, a no ser que se le indique algo diferente con una proposición de salto condicional o incondicional. Existen varias opciones para la especificación de los destinos de los saltos:

1. El operando de la instrucción da la localidad destino.
2. El operando de la instrucción especifica la distancia relativa, positiva o negativa, del salto.
3. El destino se especifica simbólicamente; esto es, la máquina maneja etiquetas.

Con las dos primeras opciones existe la posibilidad adicional de quitar el operando de la cima de la pila.

Aquí se eligió la tercera opción para la máquina abstracta porque es más fácil generar dichos saltos. Además, no es necesario modificar las direcciones simbólicas si, después de generar el código de la máquina abstracta, se hacen ciertas mejoras al código que den como resultado la inserción o borrado de instrucciones.

Las instrucciones del flujo de control de la máquina de pila son:

`etiqueta` *l*	destino de los saltos a *l*; no tiene otro efecto
`vea` *l*	la siguiente instrucción se toma de la proporción etiquetada con `etiqueta` *l*
`sifalsovea` *l*	saca el valor del tope de la pila; salta si es de cero
`siciertovea` *l*	saca el valor del tope de la pila; salta si es distinto de cero
`alto`	detiene la ejecución

Traducción de proposiciones

El esquema de la figura 2.33 describe el código de la máquina abstracta para las proposiciones condicionales **while**. El siguiente análisis se centra en la creación de etiquetas.

Considérese la disposición del código para proposiciones **if** de la figura 2.33. Sólo puede haber una instrucción `etiqueta etiq` en la traducción de un programa fuente; de otro modo, no se sabrá con certeza hacia dónde fluye el control desde una proposición `vea etiq`. Por tanto, cuando se traduzca una proposición **if**, se necesita algún mecanismo que reemplace consistentemente `etiq` en la estructura del código utilizando una etiqueta única.

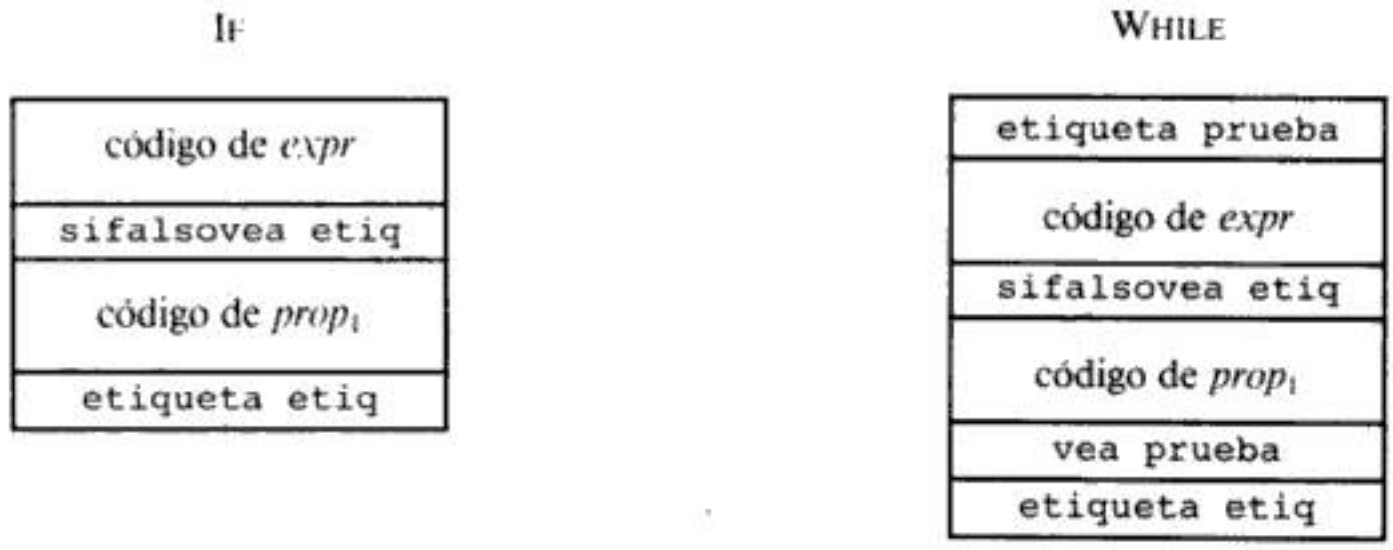


Fig. 2.33. Disposición del código para las proposiciones condicional y **while**.

Supóngase que *etiqnueva* es un procedimiento que devuelve una etiqueta nueva cada vez que se le llama. En la siguiente acción semántica, la etiqueta devuelta por una llamada a *etiqnueva* se registra utilizando una variable local *etiq*:

$$
\begin{aligned}
prop \rightarrow \textbf{if}\ expr\ \textbf{then}\ prop_1 \quad \{ \quad & etiq := etiqnueva\ ; \\
& prop.t := expr.t \ \| \\
& \texttt{'sifalsovea'}\ etiq \ \| \\
& prop_1.t \ \| \\
& \texttt{'etiqueta'}\ etiq \}
\end{aligned}
\tag{2.18}
$$

Emisión de una traducción

Los traductores de expresiones de la sección 2.5 usaban proposiciones *print* para generar incrementalmente la traducción de una expresión. Se pueden usar proposiciones *print* semejantes para emitir la traducción de proposiciones. En lugar de proposiciones *print*, se usa un procedimiento *emite* para ocultar los detalles de la impresión. Por ejemplo, *emite* se ocupa de si cada instrucción de la máquina abstracta necesita estar en una línea aparte. Por medio del procedimiento *emite*, en lugar de (2.18) se puede escribir lo siguiente:

```
prop → if
       expr      { etiq := etiqnueva ; emite ('sifalsovea', etiq); }
       then
       prop1     { emite ('etiqueta', etiq); }
```

Cuando las acciones semánticas aparecen en una producción, los elementos del lado derecho de la producción se consideran en un orden de izquierda a derecha. Para la producción anterior, el orden de las acciones es el siguiente: se realizan las acciones durante el análisis sintáctico de *expr*, a *etiq* se le asigna la etiqueta devuelta por *etiqnueva* y se emite la instrucción `sifalsovea`, se realizan las acciones durante el análisis sintáctico de $prop_1$ y, finalmente, se emite la instrucción `etiqueta`. Supo-

```
procedure prop;
var prueba, etiq: integer;    /* para etiquetas */
begin
    if preanálisis = id then begin
        emite ('valori', valcomplex); parea (id); parea (':='); expr
    end
    else if preanálisis = 'if' then begin
        parea ('if');
        expr;
        etiq := etiqnueva;
        emite ('sifalsovea', etiq);
        parea ('then');
        prop;
        emite ('etiqueta', etiq)
    end
        /* aquí va el código del resto de las proposiciones */
    else error
end;
```

Fig. 2.34. Seudocódigo de la traducción de proposiciones.

niendo que las acciones durante el análisis sintáctico de *expr* y $prop_1$ emiten el código para esos no terminales, la producción anterior implanta el esquema del código de la figura 2.33.

El seudocódigo para traducir las proposiciones de asignación y condicional se muestra en la figura 2.34. Como la variable *etiq* es local al procedimiento *prop*, su valor no se ve afectado por las llamadas a los procedimientos *expr* y *prop*. Supóngase que las etiquetas de la traducción tienen la forma `L1`, `L2`, ... El seudocódigo maneja tales etiquetas utilizando el entero que sigue a `L`. Por tanto, *etiq* se declara como entero, *etiqnueva* devuelve un entero que se convierte en el valor de *etiq*, y se debe escribir *emite* para que imprima una etiqueta dado un entero.

La disposición del código para las proposiciones **while** de la figura 2.33 se puede convertir en código de una forma similar. La traducción de una secuencia de proposiciones es simplemente la concatenación de las proposiciones de la secuencia, y se deja como ejercicio para el lector.

La traducción de la mayoría de las construcciones de una entrada y una salida es similar a la de las proposiciones **while**. Esto se explica considerando el flujo de control en las expresiones.

Ejemplo 2.10. El analizador léxico de la sección 2.7 contiene un condicional de la forma:

if t = **blanco or** t = **tab then** . . .

Si t es un espacio en blanco, es evidente que no es necesario probar si t es un TAB, porque la primera igualdad implica que la condición es verdadera. La expresión

$expr_1$ **or** $expr_2$

se puede implantar, por tanto, de la forma

if $expr_1$ **then true else** $expr_2$

El lector puede verificar que el código siguiente implanta el operador **or**:

```
código para expr1
copia                          /* copia el valor de expr1 */
siciertovea etiq
saca                           /* saca el valor de expr1 */
código para expr2
etiqueta etiq
```

Recuérdese que las instrucciones `siciertovea` y `sifalsovea` sacan el valor del tope de la pila para simplificar la generación de código para las proposiciones condicional y **while**. Copiando el valor de $expr_1$ se asegura que el valor de la cima de la pila es verdadero si la instrucción `siciertovea` origina un salto. □

2.9 REUNION DE LAS TECNICAS

En este capítulo se presentaron varias técnicas dirigidas por la sintaxis para la construcción de la etapa inicial de un compilador. Para hacer un resumen de esas técnicas, en esta sección se construye un programa en C que funciona como traductor

infijo a postfijo para un lenguaje que consta de secuencias de expresiones terminadas por símbolos de punto y coma. Las expresiones están constituidas por números, identificadores y los operadores +, −, *, /, `div` y `mod`. La salida que produce el traductor es una representación postfija de cada expresión. El traductor es una ampliación de los programas desarrollados en las secciones 2.5 a 2.7. Al final de esta sección se da un listado completo del programa en C.

Descripción del traductor

El traductor se diseña utilizando el esquema de traducción dirigida por la sintaxis de la figura 2.35. El componente léxico **id** representa una secuencia no vacía de letras y dígitos que comienzan con una letra, **núm**, una secuencia de dígitos, y **fda**, un carácter de fin de archivo. Los componentes léxicos se separan por secuencias de blancos, TAB y nueva línea ("espacios en blanco"). El atributo *lexema* del componente léxico **id** da la cadena de caracteres que forma el componente léxico; el atributo *valor* del componente léxico **núm** da el entero representado por el **núm**.

El código para el traductor se organiza en siete módulos, cada uno almacenado en un archivo aparte. La ejecución comienza en el módulo `principal.c`, que consta de una llamada a `inic()` para la inicialización, seguida de una llamada a `analizsint()` para la traducción. Los seis módulos restantes se muestran en la figura 2.36. También hay un archivo de encabezamiento global `global.h` que contiene las definiciones comunes a más de un módulo; la primera proposición de cada módulo

```
#include "global.h"
```

hace que este archivo de encabezamiento se incluya como parte del módulo. Antes de mostrar el código del traductor, se describirá brevemente cada módulo y su construcción.

inicial	→	*lista* **eof**	
lista	→	*expr* ; *lista*	
	|	ϵ	
expr	→	*expr* + *término*	{ *print* ('+') }
	|	*expr* − *término*	{ *print* ('−') }
	|	*término*	
término	→	*término* * *factor*	{ *print* ('*') }
	|	*término* / *factor*	{ *print* ('/') }
	|	*término* **div** *factor*	{ *print* ('DIV') }
	|	*término* **mod** *factor*	{ *print* ('MOD') }
	|	*factor*	
factor	→	(*expr*)	
	|	**id**	{ *print* (**id**.*lexema*) }
	|	**núm**	{ *print* (**núm**.*valor*) }

Fig. 2.35. Especificación del traductor de forma infija a postfija.

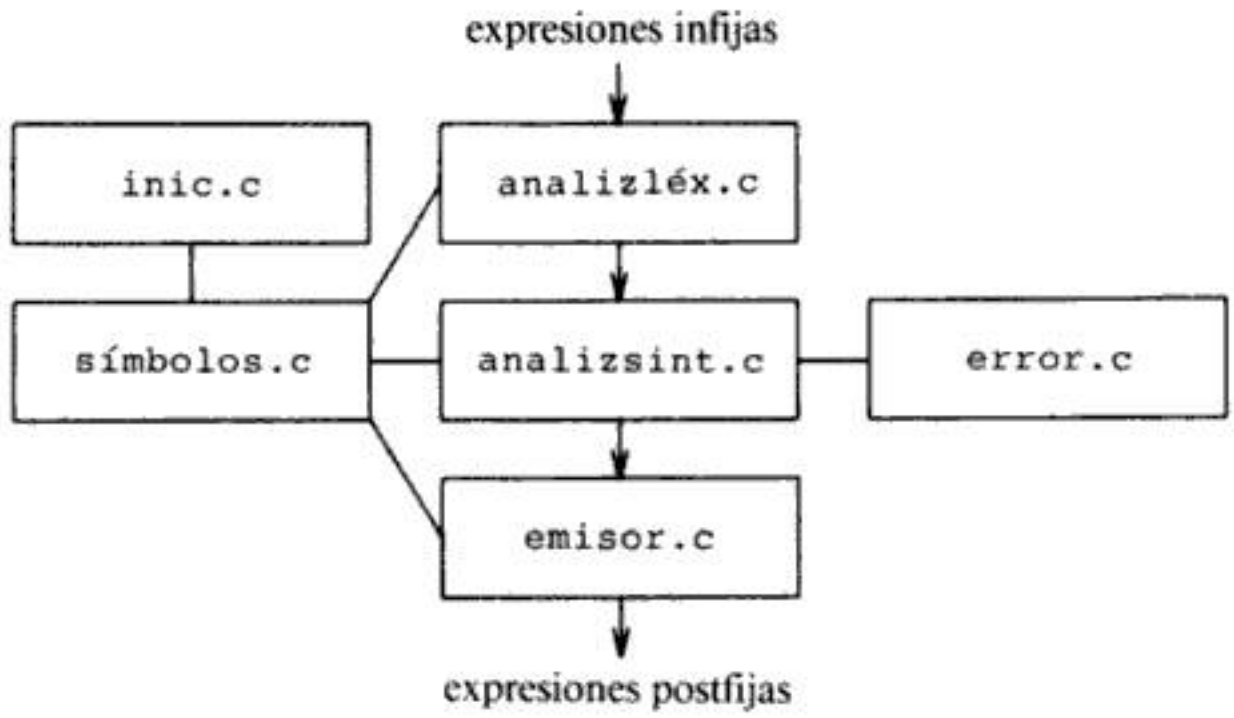


Fig. 2.36. Módulos del traductor de expresiones infijas a postfijas.

Módulo de análisis léxico `analizléx.c`

El analizador léxico es una rutina denominada `análex()`, que llama el analizador sintáctico para que encuentre componentes léxicos. Aplicada a partir del seudocódigo de la figura 2.30, la rutina lee la entrada, un carácter a la vez, y devuelve al analizador sintáctico el componente léxico encontrado. El valor del atributo asociado con el componente léxico se asigna a una variable global `valcomplex`.

Los componentes léxicos que espera el analizador sintáctico son los siguientes:

```
+ - * / DIV MOD ( ) ID NUM FIN
```

Aquí, `ID` representa un identificador, `NUM`, un número, y `FIN`, el carácter de fin de archivo. El analizador léxico elimina los espacios en blanco sin devolver nada al analizador sintáctico. La tabla de la figura 2.37 muestra el componente léxico y el valor de atributo producido por el analizador léxico en cada lexema del lenguaje fuente.

LEXEMA	COMPONENTE LÉXICO	VALOR DEL ATRIBUTO
espacio en blanco		
secuencia de dígitos	`NUM`	valor numérico de la secuencia
`div`	`DIV`	
`mod`	`MOD`	
otras secuencias de una letra seguida de letras y dígitos	`ID`	índice de `tablasimb`
carácter fin-de-archivo	`FIN`	
cualquier carácter	ese carácter	`NINGUNO`

Fig. 2.37. Descripción de componentes léxicos.

El analizador léxico utiliza la rutina de la tabla de símbolos `busca` para determinar si el lexema de un identificador ya se encontró, y la rutina `inserta`, para almacenar un lexema nuevo en la tabla de símbolos. También incrementa una variable global `numlínea` cuando encuentra un carácter de nueva línea.

Módulo de análisis sintáctico `analizsint.c`

El analizador sintáctico se construye utilizando las técnicas de la sección 2.5. Primero se elimina la recursividad por la izquierda del esquema de traducción de la figura 2.35, para poder analizar la gramática subyacente con un analizador sintáctico descendente recursivo. El esquema transformado se muestra en la figura 2.38.

Después se construyen las funciones de los no terminales *expr*, *término* y *factor* igual que en la figura 2.24. La función `análsint()` aplica el símbolo inicial de la gramática y llama a `análex` cuando necesita un nuevo componente léxico. El analizador sintáctico utiliza la función `emite` para generar la salida, y la función `error`, para informar de un error sintáctico.

inicio → *lista* **eof**
lista → *expr* ; *lista*
| ϵ
expr → *término masterminos*
masterminos → + *término* { *print* ('+') } *masterminos*
| − *término* { *print* ('−') } *masterminos*
| ϵ
término → *factor masfactores*
masfactores → * *factor* { *print* ('*') } *masfactores*
| / *factor* { *print* ('/') } *masfactores*
| **div** *factor* { *print* ('DIV') } *masfactores*
| **mod** *factor* { *print* ('MOD') } *masfactores*
| ϵ
factor → (*expr*)
| **id** { *print* (**id**.*lexema*) }
| **núm** { *print* (**núm**.*valor*) }

Fig. 2.38. Esquema de traducción dirigida por la sintaxis posterior a la eliminación de la recursividad por la izquierda.

Módulo emisor `emisor.c`

El módulo emisor consta de una sola función `emite(t.tval)`, que genera la salida para el componente léxico `t` con valor de atributo `tval`.

Módulos de la tabla de símbolos `símbolos.c` **e** `inic.c`

El módulo `simbolos.c` de la tabla de símbolos implanta la estructura de datos que se muestra en la figura 2.29 de la sección 2.7. Los elementos de entrada en la matriz `tablasimb` son pares que consisten en un apuntador a la matriz `lexemas` y un entero indicativo del lexema almacenado allí. La operación `inserta(s,t)` de-

vuelve el índice de `tablasimb` para el lexema `s` que forma el componente léxico `t`. La función `busca(s)` devuelve el índice de la entrada de `tablasimb` para el lexema `s`, o devuelve `0` si `s` no está allí.

El módulo `inic.c` se usa para precargar `tablasimb` con palabras clave. Las representaciones de los lexemas y los componentes léxicos de todas las palabras clave se almacenan en la matriz `palsclave`, que es del mismo tipo que la matriz `tablasimb`. La función `inic()` recorre secuencialmente la matriz `palsclave` usando la función `inserta` para meter las palabras clave en la tabla de símbolos. Esta disposición permite modificar de manera apropiada la representación de los componentes léxicos de las palabras clave.

Módulo de errores `error.c`

El módulo de errores administra la emisión de informes de error, que es muy primitivo. Al encontrar un error de sintaxis, el compilador imprime un mensaje informando de un error en la línea en curso y después se detiene. Una técnica mejor de recuperación de errores podría saltar al siguiente símbolo de punto y coma y continuar el análisis sintáctico; se invita al lector a que haga esta modificación al traductor. En el capítulo 4 se presentan técnicas de recuperación de errores más complicadas.

Creación del compilador

El código de los módulos aparece en siete archivos: `analizléx.c`, `analizsint.c`, `emisor.c`, `símbolos.c`, `inic.c`, `error.c`, y `principal.c`. El archivo `principal.c` contiene la rutina principal del programa en C que llama a `inic()`, después llama a `análsint()` y, si se termina con éxito, llama a `exit(0)`.

En el sistema operativo UNIX, el compilador se puede crear ejecutando el mandato

```
cc analizléx.c analizsint.c emisor.c símbolos.c
inic.c error.c principal.c
```

o compilando por separado los archivos, con

```
cc -c nombrearch.c
```

y enlazando los archivos resultantes *nombrearch*`.o`:

```
cc analizléx.o analizsint.o emisor.o símbolos.o
inic.o error.o principal.o
```

El mandato `cc` crea un archivo `a.out` que contiene al traductor. Después se puede utilizar el traductor tecleando `a.out` seguido de la expresión que se desea traducir; por ejemplo,

```
2+3*5;
12 div 5 mod 2;
```

o cualquier otra expresión que se desee. Hágase la prueba.

El listado

El siguiente es un listado del programa en C que aplica el traductor. Se muestra el archivo de encabezamiento global `global.h`, seguido de siete archivos fuente. Por claridad, el programa está escrito en un estilo elemental de C.

```
/****   global.h   ********************************************/

#include <stdio.h>      /* carga las rutinas de entrada y salida */
#include <ctype.h>      /* carga las rutinas de prueba de
                           caracteres */

#define TAMBUFF 128     /* tamaño del buffer */
#define NINGUNO  -1
#define FDC      '\0'

#define NUM      256
#define DIV      257
#define MOD      258
#define ID       259
#define FIN      260

int valcomplex;      /* valor del atributo del componente léxico */

int numlínea;

struct entrada {     /* forma del elemento de entrada de la tabla
                        de símbolos */
     char *aplex;
     int  complex;
};

struct entrada tablasimb[];   /* tabla de símbolos */

/****   analizléx.c   ******************************************/

#include "global.h"

char buflex[TAMBUFF];
int  numlínea = 1;
int  valcomplex = NINGUNO;

int  análex()      /* analizador léxico */
{
     int  t;
     while(1) {
          t = getchar();
          if (t == ' ' || t == '\t')
               ;  /* elimina espacios en blanco */
          else if (t == '\n')
               numlínea = numlínea + 1;
```

```
else if (isdigit(t)) {   /* t es un dígito */
     ungetc(t, stdin);
     scanf("%d", &valcomplex);
     return NUM;
}
else if (isalpha(t)) {   /* t es una letra */
     int p, b = 0;
     while (isalnum(t)) {   /* t es alfanumérico */
          buflex[b] = t;
          t = getchar();
          b = b + 1;
          if (b >= TAMBUFF)
               error("error del compilador");
          }
          buflex[b] = FDC;
          if (t != EOF)
               ungetc(t, stdin);
          p = busca(buflex);
          if (p == 0)
               p = inserta(buflex, ID);
          valcomplex = p;
          return tablasimb[p].complex;
     }
     else if (t == EOF)
          return FIN;
     else {
          valcomplex = NINGUNO;
          return t;
     }
   }
}
```

```
/****   analizsint.c   ****************************************/

#include "global.h"

int preanálisis;

análsint()     /* analiza sintácticamente y traduce la lista de
                  la expresión */
{
     preanálisis = análex();
     while (preanálisis != FIN ) {
          expr(); parea(';');
     }
}
```

```
expr()
{
     int  t;
     término();
     while(1)
          switch (preanálisis) {
          case '+': case '-':
               t = preanálisis;
               parea (preanálisis); término(); emite(t, NINGUNO);
               continúe;
          default:
               return;
          }
}

término()
{
     int t;
     factor();
     while(1)
          switch (preanálisis) {
          case '*': case '/': case DIV: case MOD:
               t = preanálisis;
               parea (preanálisis); factor(); emite(t, NINGUNO);
               continúe;
          default:
               return;
          }
}

factor()
{
     switch(preanálisis) {
          case '(':
               parea('('); expr(); parea(')'); break;
          case NUM:
               emite(NUM, valcomplex); parea(NUM); break;
          case ID:
               emite(ID, valcomplex); parea(ID); break;
          default:
               error("error de sintaxis");
     }
}

parea(t)
     int  t;
{
```

```
     if (preanálisis == t)
          preanálisis = análex();
     else error ("error de sintaxis");
}

/****   emisor.c   ********************************************/

#include "global.h"

emite(t, tval)          /* genera la salida */
     int  t, tval;
{
     switch(t) {
          case '+': case '-': case '*': case '/':
               printf("%c\n", t); break;
          case DIV:
               printf("DIV\n"); break;
          case MOD:
               printf("MOD\n"); break;
          case NUM:
               printf("%d\n", tval); break;
          case ID:
               printf("%s\n", tablasimb[tval].aplex); break;
          default:
               printf("complex %d, valcomplex %d\n", t, tval);
     }
}

/****   símbolos.c   ******************************************/

#include "global.h"
#define MAXLEX  999       /* tamaño de la matriz de lexemas */
#define MAXSIMB 100       /* tamaño de tablasimb */
char lexemas[MAXLEX];
int  ultcar = -1;  /* última posición usada en los lexemas */
struct entrada tablasimb[MAXSIMB];
int  ultent = 0;  /* última posición usada en tablasimb */

int  busca(s)     /* devuelve la posición del elemento de entrada
                     de s */
     char s[];
{
     int  p;
     for (p = ultent; p > 0; p = p - 1)
          if (strcmp(tablasimb[p].aplex, s) == 0)
               return p;
     return 0;
}
```

```
int inserta(s,clex)  /* devuelve la posición del elemento de
                        entrada de s */
char s[];
int  clex;
{
int  lon;
lon = strlen(s); /* strlen evalúa la longitud de s */
if (ultent + 1 >= MAXSIMB)
     error("tabla de símbolos llena");
if (ultcar + lon + 1 >= MAXLEX)
     error("matriz de lexemas llena");
ultent = ultent + 1;
tablasimb[ultent].complex = clex;
     tablasimb[ultent].aplex = &lexemas[ultcar + 1];
     ultcar = ultcar + lon + 1;
     strcpy(tablasimb[ultent].aplex, s);
     return ultent;
}

/****   inic.c   ************************************************/

#include "global.h"

struct entrada palsclave[] = {
     "div", DIV,
     "mod", MOD,
     0,     0
};

inic()     /* carga las palabras clave en la tabla de símbolos */
{
     struct entrada *p;
     for (p = palsclave; p->complex; p++)
          inserta(p->aplex, p->complex);
}

/****   error.c   ************************************************/

#include "global.h"

error(m)        /* genera todos los mensajes de error */
     char *m;
{
     fprintf(stderr, "línea %d: %s\n", numlínea, m);
     exit(1);     /* terminación sin éxito */
}
```

```
/****    principal.c    ***********************************************/

#include "global.h"

main()
{
      inic();
      análsint();
      exit(0);      /* terminación con éxito */
}

/*****************************************************************/
```

EJERCICIOS

2.1 Considérese la gramática independiente del contexto

$$S \rightarrow S\ S\ + \mid S\ S\ * \mid \mathtt{a}$$

a) Demuéstrese cómo se puede generar la cadena aa+a* con esta gramática.
b) Constrúyase un árbol de análisis sintáctico para esta cadena.
c) ¿Qué lenguaje genera esta gramática? Explíquese la respuesta.

2.2 ¿Qué lenguajes generan las siguientes gramáticas? En cada caso, explíquese la respuesta.

a) $S \rightarrow \mathtt{0}\ S\ \mathtt{1} \mid \mathtt{0}\ \mathtt{1}$
b) $S \rightarrow +\ S\ S \mid -\ S\ S \mid \mathtt{a}$
c) $S \rightarrow S\ (\ S\ )\ S \mid \epsilon$
d) $S \rightarrow \mathtt{a}\ S\ \mathtt{b}\ S \mid \mathtt{b}\ S\ \mathtt{a}\ S \mid \epsilon$
e) $S \rightarrow \mathtt{a} \mid S + S \mid S\ S \mid S * \mid (\ S\ )$

2.3 ¿Cuáles de las gramáticas del ejercicio 2.2 son ambiguas?

2.4 Constrúyase una gramática independiente del contexto no ambigua para cada uno de los lenguajes siguientes. En cada caso, demuéstrese que la gramática es correcta.

a) Expresiones aritméticas de notación postfija.
b) Listas asociativas por la izquierda de identificadores separados por comas.
c) Listas asociativas por la derecha de identificadores separados por comas.
d) Expresiones aritméticas de enteros e identificadores con los cuatro operadores binarios +, −, *, /.
e) Añádanse los operadores más y menos unarios a los operadores aritméticos de (d).

***2.5** a) Demuéstrese que todas las cadenas binarias generadas por la siguiente gramática tienen valores divisibles por 3. *Sugerencia*: Utilícese inducción sobre el número de nodos en un árbol de análisis sintáctico.

$$\mathit{núm} \rightarrow \mathtt{11} \mid \mathtt{1001} \mid \mathit{núm}\ \mathtt{0} \mid \mathit{núm}\ \mathit{núm}$$

b) ¿Genera la gramática todas las cadenas binarias con valores divisibles por 3?

2.6 Constrúyase una gramática independiente del contexto para los números romanos.

2.7 Constrúyase un esquema de traducción dirigida por la sintaxis que traduzca expresiones aritméticas de notación infija a notación prefija, en la que un operador aparece antes que sus operandos; por ejemplo, $-xy$ es la notación prefija de $x-y$. Constrúyanse árboles de análisis sintáctico con anotaciones para las entradas `9-5+2` y `9-5*2`.

2.8 Constrúyase un esquema de traducción dirigida por la sintaxis que traduzca expresiones aritméticas de notación postfija a notación infija. Constrúyanse árboles de análisis sintáctico con anotaciones para las entradas `95-2*` y `952*-`.

2.9 Constrúyase un esquema de traducción dirigida por la sintaxis que traduzca enteros a números romanos.

2.10 Constrúyase un esquema de traducción dirigida por la sintaxis que traduzca números romanos a enteros.

2.11 Constrúyanse los analizadores sintácticos descendentes recursivos para las gramáticas del ejercicio 2.2 a), b) y c).

2.12 Constrúyase un traductor dirigido por la sintaxis que compruebe si todos los paréntesis de una cadena de entrada están equilibrados de manera apropiada.

2.13 Las siguientes reglas definen la traducción de una palabra en inglés al seudolatín:

a) Si la palabra comienza con una cadena no vacía de consonantes, desplazar la cadena inicial de consonantes al final de la palabra y añadir el sufijo `AY`; por ejemplo, `pig` se convierte en `igpay`.
b) Si la palabra comienza con una vocal, añadir el sufijo `YAY`; por ejemplo, `owl` se convierte en `owlyay`.
c) `U` después de una `Q` es una consonante.
d) `Y` al inicio de una palabra es una vocal si no va seguida de una vocal.
e) Las palabras de una sola letra no se modifican.

Constrúyase un esquema de traducción dirigida por la sintaxis para el seudolatín.

2.14 En el lenguaje de programación C, la proposición **for** tiene la forma:

`for` ($expr_1$; $expr_2$; $expr_3$) *prop*

La primera expresión se ejecuta antes del lazo; generalmente se utiliza para inicializar el índice del lazo. La segunda expresión es una prueba realizada antes de cada iteración del lazo; se sale del lazo si la expresión se convierte en 0. El ciclo en sí mismo consiste en una proposición { *prop* $expr_3$; }. La tercera expresión se ejecuta al final de cada iteración; por lo general, se usa para incrementar el índice del lazo. El significado de la proposición **for** es similar a

$expr_1$; `while` ($expr_2$) { *prop* $expr_3$; }

Constrúyase un esquema de traducción dirigida por la sintaxis para traducir proposiciones **for** de C al código de la máquina de pila.

***2.15** Considérese la siguiente proposición **for**:

for $i := 1$ **step** $10 - j$ **until** $10 * j$ **do** $j := j + 1$

Se pueden dar tres definiciones semánticas para esta proposición. La primera es que el límite $10 * j$ y el incremento $10 - j$ deben ser evaluados una vez antes del lazo como en PL/I. Por ejemplo, si $j = 5$ antes del lazo, habría que recorrer el lazo diez veces y se saldría. Existe un segundo significado, completamente distinto, cuando es necesario evaluar el límite y el incremento cada vez que se recorre el lazo. Por ejemplo, si $j = 5$ antes del lazo, el ciclo no terminaría nunca. Un tercer significado estaría determinado por los lenguajes de tipo ALGOL. Cuando el incremento es negativo, la prueba realizada para la conclusión del lazo es $i < 10 * j$, y no $i > 10 * j$. Para cada una de estas tres definiciones semánticas, constrúyase un esquema de traducción dirigida por la sintaxis para traducir estos tres lazos **for** al código de la máquina de pila.

2.16 Considérese el siguiente fragmento de gramática para las proposiciones **if-then** e **if-then-else**:

prop → **if** *expr* **then** *prop*
| **if** *expr* **then** *prop* **else** *prop*
| **otras**

donde **otras** representa a las demás proposiciones del lenguaje.

a) Demuéstrese que esta gramática es ambigua.
b) Constrúyase una gramática no ambigua equivalente que asocie cada **else** con el **then** sin emparejar inmediatamente anterior.
c) Constrúyase un esquema de traducción dirigida por la sintaxis basado en esta gramática para traducir proposiciones condicionales al código de la máquina de pila.

***2.17** Constrúyase un esquema de traducción dirigida por la sintaxis que traduzca expresiones aritméticas de notación infija a expresiones aritméticas de notación infija sin paréntesis redundantes. Muéstrese el árbol de análisis sintáctico con anotaciones para la entrada `(((1 + 2) * (3 * 4)) + 5)`.

EJERCICIOS DE PROGRAMACION

P2.1 Aplíquese un traductor de enteros a números romanos basado en el esquema de traducción dirigida por la sintaxis desarrollado en el ejercicio 2.9.

P2.2 Modifíquese el traductor de la sección 2.9 con objeto de producir como salida un código para la máquina de pila abstracta de la sección 2.8.

P2.3 Modifíquese el módulo de recuperación de errores del traductor de la sección 2.9 para saltar a la siguiente expresión de entrada al encontrar un error.

P2.4 Amplíese el traductor de la sección 2.9 para manejar todas las expresiones de Pascal.

P2.5 Amplíese el compilador de la sección 2.9 para traducir a código de la máquina de pila las proposiciones generadas por la gramática siguiente:

$$
\begin{aligned}
\mathit{prop} &\rightarrow \textbf{id} := \mathit{expr} \\
&\quad | \ \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{prop} \\
&\quad | \ \textbf{while}\ \mathit{expr}\ \textbf{do}\ \mathit{prop} \\
&\quad | \ \textbf{begin}\ \mathit{props\_opc}\ \textbf{end} \\
\mathit{props\_opc} &\rightarrow \mathit{lista\_props} \mid \epsilon \\
\mathit{lista\_props} &\rightarrow \mathit{lista\_props}\ ;\ \mathit{prop} \mid \mathit{prop}
\end{aligned}
$$

***P2.6** Constrúyase un conjunto de expresiones de prueba para el compilador de la sección 2.9, de modo que cada producción se use como mínimo una vez para obtener alguna expresión de prueba. Constrúyase un programa de prueba que pueda ser utilizado como herramienta general de prueba para los compiladores. Utilícese el programa de prueba para ejecutar su compilador con esas expresiones de prueba.

P2.7 Constrúyase un conjunto de proposiciones de prueba para el compilador del ejercicio P2.5, de modo que cada producción se use como mínimo una vez para generar algunas proposiciones de prueba. Utilícese el programa de prueba del ejercicio P2.6 para ejecutar el compilador con estas expresiones de prueba.

NOTAS BIBLIOGRAFICAS

Este capítulo preliminar hace referencia a diversos temas tratados más detalladamente en el resto del libro. En los capítulos que contienen información adicional se hace referencia a la bibliografía recomendada.

Chomsky [1956] introdujo las gramáticas independientes del contexto como parte de un estudio sobre los lenguajes naturales. Su utilización para especificar la sintaxis de los lenguajes de programación surgió independientemente. Mientras trabajaba en un borrador de ALGOL 60, John Backus "adaptó de inmediato [las producciones de Emil Post] a ese uso" (Wexelblat [1981, pág. 162]). La notación resultante fue una variante de las gramáticas independientes del contexto. El lingüista Panini diseñó una notación sintáctica equivalente para especificar las reglas de la gramática del sánscrito de entre el 400 a. de C. y el 200 a. de C. (Ingerman [1967]).

En una carta de Knuth [1964], está contenida la propuesta de que BNF, que comenzó como abreviatura de *Backus Normal Form* (forma normal de Backus), se leyera *Backus-Naur Form* (forma de Backus-Naur), para reconocer las contribuciones de Naur como editor del informe de ALGOL 60 (Naur [1963]).

Las definiciones dirigidas por la sintaxis son una forma de definición inductiva en la cual la inducción se encuentra en la estructura sintáctica. Como tales, han sido muy utilizadas informalmente en matemáticas. Su aplicación a los lenguajes de programación se introdujo con el uso de una gramática para estructurar el informe de ALGOL 60. Poco tiempo después, Irons [1961] construyó un compilador dirigido por la sintaxis.

El análisis sintáctico descendente recursivo se utiliza aproximadamente desde 1960. Bauer [1976] atribuye el método a Lucas [1961]. Hoare [1962b, pág. 128] describe un compilador de ALGOL organizado como "un conjunto de procedimientos, cada uno de los cuales puede procesar una de las unidades sintácticas del informe de ALGOL 60". Foster [1968] analiza la eliminación de la recursividad por la izquierda de las producciones con acciones semánticas que no afecten a los valores de los atributos.

McCarthy [1963] abogaba por que la traducción de un lenguaje se basara en una sintaxis abstracta. En el mismo artículo, McCarthy [1963, pág. 24] dejaba "que el lector se convenciera por sí mismo "de que una formulación recursiva por el final de la función factorial es equivalente a un programa iterativo.

Las ventajas de dividir un compilador en una etapa inicial y otra final se analizaron en un informe de comité de Strong y colaboradores [1958]. El informe acuñó el término UNCOL (siglas, en inglés, de lenguaje orientado a un computador universal, *universal computer oriented language*) para un lenguaje intermedio universal. El concepto ha quedado como un ideal.

Una buena forma de aprender técnicas de implantación es leer el código de los compiladores existentes. Lamentablemente, no se suele publicar el código. Randell y Russell [1964] hacen una amplia descripción de uno de los primeros compiladores de ALGOL. El código de compiladores también es tratado por McKeeman, Horning y Wortman [1970]. Barron [1981], es un conjunto de artículos sobre la implantación de Pascal, que incluye notas sobre implantación distribuidas con el compilador de Pascal P (Nori y colaboradores [1981]), detalles de la generación de código (Ammann [1977]) y el código para una implantación de Pascal S, un subconjunto de Pascal diseñado por Wirth [1981] para el uso de estudiantes. Knuth [1985] da una descripción excepcionalmente clara y detallada del traductor de TEX.

Kernighan y Pike [1984] describen en detalle cómo construir un programa de calculadora de escritorio a partir de un esquema de traducción dirigida por la sintaxis, utilizando las herramientas para la construcción de compiladores disponibles en el sistema operativo UNIX. La ecuación (2.17) está tomada de Tantzen [1963].

CAPITULO 3

Análisis léxico

Este capítulo trata sobre las técnicas para especificar e implantar analizadores léxicos. Una forma sencilla de crear un analizador léxico consiste en la construcción de un diagrama que ilustre la estructura de los componentes léxicos del lenguaje fuente, y después hacer la traducción "a mano" del diagrama a un programa para encontrar los componentes léxicos. De esta forma, se pueden producir analizadores léxicos eficientes.

Las técnicas utilizadas para construir analizadores léxicos también se pueden aplicar a otras áreas, como, por ejemplo, a lenguajes de consulta y sistemas de recuperación de información. En cada aplicación, el problema de fondo es la especificación y diseño de programas que ejecuten las acciones activadas por patrones dentro de las cadenas. Como la programación dirigida por patrones es de mucha utilidad, se introduce un lenguaje de patrón-acción, llamado LEX, para especificar los analizadores léxicos. En este lenguaje, los patrones se especifican por medio de expresiones regulares, y un compilador de LEX puede generar un reconocedor de las expresiones regulares mediante un autómata finito eficiente.

Otros lenguajes utilizan expresiones regulares para describir patrones. Por ejemplo, el lenguaje analizador de patrones AWK utiliza expresiones regulares para seleccionar las líneas de la entrada que se han de procesar, y el *shell* del sistema UNIX permite al usuario referirse a un conjunto de nombres de archivo mediante la escritura de una expresión regular. La instrucción `rm *.o` de UNIX, por ejemplo, borra todos los archivos cuyos nombres terminen en "`.o`"[1].

Una herramienta de *software* que automatiza la construcción de analizadores léxicos permite que personas con diferentes conocimientos utilicen la concordancia de patrones en sus propias áreas de aplicación. Por ejemplo, Jarvis [1976] utilizó un generador de analizadores léxicos para crear un programa que reconoce imperfecciones en tarjetas de circuitos impresos. Los circuitos se examinan digitalmente y se transforman en "cadenas" de segmentos de recta a distintos ángulos. El "analizador léxico" busca patrones correspondientes a imperfecciones en la cadena de segmentos de recta. La gran ventaja de un generador de analizadores léxicos es que puede

[1] La expresión `*.o` es una variante de la notación usual para las expresiones regulares. En los ejercicios 3.10 y 3.14 se mencionan algunas variantes de uso común de las notaciones de las expresiones regulares.

utilizar los algoritmos más conocidos de concordancia de patrones, con lo cual crea analizadores léxicos eficientes para los no especialistas en dichas técnicas.

3.1 FUNCION DEL ANALIZADOR LEXICO

El analizador léxico es la primera fase de un compilador. Su principal función consiste en leer los caracteres de entrada y elaborar como salida una secuencia de componentes léxicos que utiliza el analizador sintáctico para hacer el análisis. Esta interacción, esquematizada en la figura 3.1, suele aplicarse convirtiendo al analizador léxico en una subrutina o corrutina del analizador sintáctico. Recibida la orden "obtén el siguiente componente léxico" del analizador sintáctico, el analizador léxico lee los caracteres de entrada hasta que pueda identificar el siguiente componente léxico.

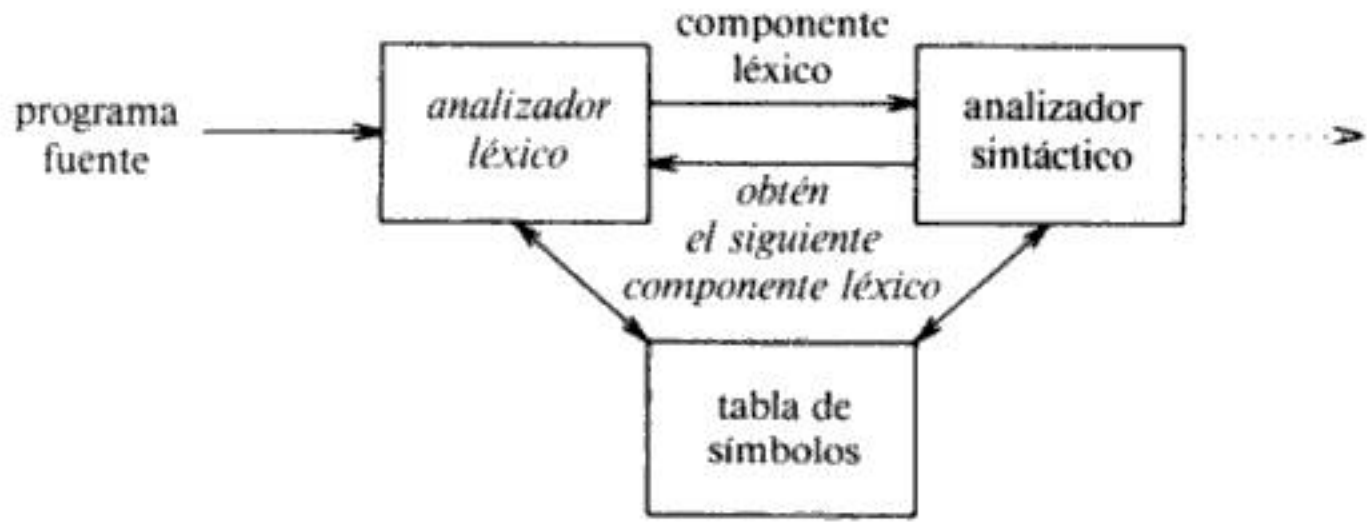


Fig. 3.1. Interacción de un analizador léxico con el analizador sintáctico.

Como el analizador léxico es la parte del compilador que lee el texto fuente, también puede realizar ciertas funciones secundarias en la interfaz del usuario, como eliminar del programa fuente comentarios y espacios en blanco en forma de caracteres de espacio en blanco, caracteres TAB y de línea nueva. Otra función es relacionar los mensajes de error del compilador con el programa fuente. Por ejemplo, el analizador léxico puede tener localizado el número de caracteres de nueva línea detectados, de modo que se pueda asociar un número de línea con un mensaje de error. En algunos compiladores, el analizador léxico se encarga de hacer una copia del programa fuente en el que están marcados los mensajes de error. Si el lenguaje fuente es la base de algunas funciones de preprocesamiento de macros, entonces esas funciones del preprocesador también se pueden aplicar al hacer el análisis léxico.

En algunas ocasiones, los analizadores léxicos se dividen en una cascada de dos fases; la primera, llamada "examen", y la segunda, "análisis léxico". El examinador se encarga de realizar tareas sencillas, mientras que el analizador léxico es el que realiza las operaciones más complejas. Por ejemplo, un compilador de FORTRAN puede utilizar un examinador para eliminar espacios en blanco de la entrada.

Aspectos del análisis léxico

Hay varias razones para dividir la fase de análisis de la compilación en análisis léxico y análisis sintáctico.

1. Un diseño sencillo es quizá la consideración más importante. Separar el análisis léxico del análisis sintáctico a menudo permite simplificar una u otra de dichas fases. Por ejemplo, un analizador sintáctico que incluya las convenciones de los comentarios y espacios en blanco es bastante más complejo que uno que pueda comprobar si los comentarios y espacios en blanco ya han sido eliminados por el analizador léxico. Si se está diseñando un lenguaje nuevo, la separación de las convenciones léxicas de las sintácticas puede dar origen a un diseño del lenguaje más claro.
2. Se mejora la eficiencia del compilador. Un analizador léxico independiente permite construir un procesador especializado y potencialmente más eficiente para esta función. Gran parte de tiempo se consume en leer el programa fuente y dividirlo en componentes léxicos. Con técnicas especializadas de manejo de *buffers* para la lectura de caracteres de entrada y procesamiento de componentes léxicos se puede mejorar significativamente el rendimiento de un compilador.
3. Se mejora la transportabilidad del compilador. Las peculiaridades del alfabeto de entrada y otras anomalías propias de los dispositivos pueden limitarse al analizador léxico. La representación de símbolos especiales o no estándar, como ↑ en Pascal, pueden ser aisladas en el analizador léxico.

Se han diseñado herramientas especializadas que ayudan a automatizar la construcción de analizadores léxicos y analizadores sintácticos cuando están separados. En este libro se tratarán algunos ejemplos de estas herramientas.

Componentes léxicos, patrones y lexemas

Cuando se menciona el análisis sintáctico, los términos "componente léxico" (*token*), "patrón" y "lexema" se emplean con significados específicos. En la figura 3.2 aparecen ejemplos de dichos usos. En general, hay un conjunto de cadenas en la entrada para el cual se produce como salida el mismo componente léxico. Este conjunto de cadenas se describe mediante una regla llamada *patrón* asociado al componente léxico. Se dice que el patrón *concuerda* con cada cadena del conjunto. Un lexema es una secuencia de caracteres en el programa fuente con la que concuerda el patrón para un componente léxico. Por ejemplo, en la proposición de Pascal

```
const pi = 3.1416;
```

la subcadena `pi` es un lexema para el componente léxico "identificador".

COMPONENTE LÉXICO	LEXEMAS DE EJEMPLO	DESCRIPCIÓN INFORMAL DEL PATRÓN
const	`const`	`const`
if	`if`	`if`
relación	<, <=, =, <>, >, >=	< o <= o = o <> o >= o >
id	`pi, cuenta, D2`	letra seguida de letras y dígitos
núm	`3.1416, 0, 6.02E23`	cualquier constante numérica
literal	`"vaciado de memoria"`	cualquier carácter entre " y ", excepto "

Fig. 3.2. Ejemplos de componentes léxicos.

Los componentes léxicos se tratan como símbolos terminales de la gramática del lenguaje fuente, con nombres en **negritas** para representarlos. Los lexemas para el componente léxico que concuerdan con el patrón representan cadenas de caracteres en el programa fuente que se pueden tratar juntos como una unidad léxica.

En la mayoría de los lenguajes de programación, se consideran componentes léxicos las siguientes construcciones: palabras clave, operadores, identificadores, constantes, cadenas literales y signos de puntuación, como paréntesis, coma y punto y coma. En el ejemplo anterior, cuando la secuencia de caracteres `pi` aparece en el programa fuente, se devuelve al analizador sintáctico un componente léxico que representa un identificador. La devolución de un componente léxico a menudo se realiza mediante el paso de un número entero correspondiente al componente léxico. Este entero es al que hace referencia el **id** en **negritas** de la figura 3.2.

Un patrón es una regla que describe el conjunto de lexemas que pueden representar a un determinado componente léxico en los programas fuente. El patrón para el componente léxico **const** de la figura 3.2 es simplemente la cadena sencilla **const** que deletrea la palabra clave. El patrón para el componente léxico **relación** es el conjunto de los seis operadores relacionales de Pascal. Para describir con precisión los patrones para componentes léxicos más complejos, como **id** (para identificador) y **núm** (para número), se utilizará la notación de expresiones regulares desarrollada en la sección 3.3.

Ciertas convenciones del lenguaje se reflejan en la dificultad del análisis léxico. Algunos lenguajes, como FORTRAN, exigen que ciertas construcciones aparezcan en posiciones fijas en la línea de entrada. Por tanto, la alineación de un lexema puede ser importante para determinar si un programa fuente es o no correcto. En el diseño de lenguajes modernos se tiende a la entrada independiente del formato, permitiendo la colocación de las construcciones en cualquier parte de la línea de entrada, de modo que este aspecto del análisis léxico está perdiendo importancia.

El tratamiento de los espacios en blanco varía mucho de un lenguaje a otro. En algunos lenguajes, como FORTRAN o ALGOL 68, los espacios en blanco no son significativos, excepto en las cadenas literales, y se pueden añadir a voluntad para mejorar la legibilidad de un programa. Las convenciones relativas a los espacios en blanco pueden complicar mucho la tarea de identificar los componentes léxicos.

Un ejemplo bastante conocido que ilustra la dificultad potencial de reconocer los componentes léxicos es la proposición `DO` de FORTRAN. En la proposición

```
DO 5 I = 1.25
```

no se puede saber hasta ver el punto decimal si `DO` es una palabra clave, o es más bien parte del identificador `DO5I`. Por otro lado, en la proposición

```
DO 5 I = 1,25
```

hay siete componentes léxicos, que corresponden a la palabra clave `DO`, la etiqueta de proposición `5`, el identificador `I`, el operador `=`, la constante `1`, la coma y la constante `25`. Aquí, no se puede estar seguro hasta encontrar la coma si `DO` es una palabra clave. Para solventar esta incertidumbre, FORTRAN 77 permite incluir una coma opcional entre la etiqueta y el índice de la proposición `DO`. Se recomienda el uso de esta coma porque ayuda a hacer más clara y legible la proposición `DO`.

En muchos lenguajes, ciertas cadenas son *reservadas*; es decir, su significado está predefinido y el usuario no lo puede modificar. Si las palabras clave no son reservadas, entonces el analizador léxico debe distinguir entre una palabra clave y un identificador definido por el usuario. En PL/I, las palabras clave no son reservadas, por lo que las reglas para distinguir las palabras clave de los identificadores son bastante complicadas, como ilustra la siguiente proposición de PL/I:

```
IF THEN THEN THEN = ELSE; ELSE ELSE = THEN;
```

Atributos de los componentes léxicos

Cuando concuerda con un lexema más de un patrón, el analizador léxico debe proporcionar información adicional sobre el lexema concreto que concordó con las siguientes fases del compilador. Por ejemplo, el patrón **núm** concuerda con las cadenas `0` y `1`, pero es indispensable que el generador de código conozca qué cadena fue realmente la que se emparejó.

El analizador léxico recoge información sobre los componentes léxicos en sus atributos asociados. Los componentes léxicos influyen en las decisiones del análisis sintáctico, y los atributos, en la traducción de los componentes léxicos. En la práctica, los componentes léxicos suelen tener un solo atributo —un apuntador a la entrada de la tabla de símbolos donde se guarda la información sobre el componente léxico; el apuntador se convierte en el atributo del componente léxico. A efectos de diagnóstico, puede considerarse tanto el lexema para un identificador como el número de línea en el que éste se encontró por primera vez. Estos dos elementos de información se pueden almacenar en la entrada de la tabla de símbolos para el identificador.

Ejemplo 3.1. Los componentes léxicos y los valores de atributos asociados para la proposición de FORTRAN.

```
E = M * C ** 2
```

se escriben a continuación como una secuencia de parejas:

<**id**, apuntador a la entrada de la tabla de símbolos para `E`>
<**op_asign**, >
<**id**, apuntador a la entrada de la tabla de símbolos para `M`>
<**op_mult**, >
<**id**, apuntador a la entrada de la tabla de símbolos para `C`>
<**op_exp**, >
<**núm**, valor entero `2`>

Obsérvese que en ciertas parejas no se necesita un valor de atributo: el primer componente es suficiente para identificar el lexema. En este pequeño ejemplo, se ha dado al componente léxico **núm** un atributo de valor entero. El compilador puede almacenar la cadena de caracteres que forma un número en una tabla de símbolos y dejar que el atributo del componente léxico **núm** sea un apuntador a la entrada de la tabla. □

Errores léxicos

Son pocos los errores que se pueden detectar simplemente en el nivel léxico porque un analizador léxico tiene una visión muy restringida de un programa fuente. Si aparece la cadena `fi` por primera vez en un programa en C en el contexto

```
fi ( a == f(x) ) ...
```

un analizador léxico no puede distinguir si `fi` es un error de escritura de la palabra clave `if` o si es un identificador de función no declarado. Como `fi` es un identificador válido, el analizador léxico debe devolver el componente léxico de un identificador y dejar que alguna otra fase del compilador se ocupe de los errores.

Pero, supóngase que surge una situación en la que el analizador léxico no puede continuar porque ninguno de los patrones concuerda con un prefijo de la entrada restante. Tal vez la estrategia de recuperación más sencilla sea la recuperación en "modo de pánico". Se borran caracteres sucesivos de la entrada restante hasta que el analizador léxico pueda encontrar un componente léxico bien formado. Esta técnica de recuperación puede confundir en ocasiones al analizador sintáctico, pero en un ambiente de computación interactivo puede resultar bastante adecuada.

Otras posibles acciones de recuperación de errores son:

1. borrar un carácter extraño
2. insertar un carácter que falta
3. reemplazar un carácter incorrecto por otro correcto
4. intercambiar dos caracteres adyacentes.

Se puede probar este tipo de transformaciones de error para intentar reparar la entrada. La más sencilla de tales estrategias consiste en observar si un prefijo de la entrada restante se puede transformar en un lexema válido mediante una sola transformación de error. Esta estrategia da por supuesto que la mayoría de los errores léxicos se deben a una sola transformación de error, suposición que normalmente, pero no siempre, se cumple en la práctica.

Una forma de encontrar los errores en un programa consiste en calcular el número mínimo de transformaciones necesarias para transformar el programa erróneo en otro que esté sintácticamente bien construido. Se dice que el programa erróneo tiene k errores cuando la secuencia más corta de transformaciones de error que lo transformará en algún programa válido tiene la longitud k. La corrección de errores de distancia mínima es un criterio teórico apropiado, pero no se suele usar en la práctica porque su aplicación es demasiado costosa. Sin embargo, algunos compiladores experimentales han empleado el criterio de la distancia mínima para hacer correcciones locales.

3.2 MANEJO DE LOS *BUFFERS* DE ENTRADA

Esta sección trata algunos aspectos de la eficiencia relacionados con el manejo de los *buffers* de entrada. Primero se menciona un esquema de dos *buffers* de entrada que resulta útil cuando es necesario un preanálisis en la entrada para identificar los componentes léxicos. Después se introducen algunas técnicas útiles para aumentar

la velocidad del analizador léxico, como el uso de "centinelas" que sirven para marcar el final del *buffer*.

Hay tres métodos generales de implantación de un analizador léxico.

1. Utilizar un generador de analizadores léxicos, como el compilador LEX de la sección 3.5, para producir el analizador léxico a partir de una especificación basada en expresiones regulares. En este caso, el generador proporciona rutinas para leer la entrada y manejarla con *buffers*.
2. Escribir el analizador léxico en un lenguaje convencional de programación de sistemas, utilizando las posibilidades de entrada y salida de este lenguaje para leer la entrada.
3. Escribir el analizador léxico en lenguaje ensamblador y manejar explícitamente la lectura de la entrada.

Las tres opciones se relacionan en orden de dificultad creciente para el encargado de la implantación. Lamentablemente, los enfoques más difíciles de implantar, muchas veces dan como resultado analizadores léxicos más rápidos. Como el analizador léxico es la única fase del compilador que lee el programa fuente carácter a carácter, es posible que se consuma mucho tiempo en la fase de análisis léxico, aunque las fases posteriores sean conceptualmente más complejas. Así, la velocidad del análisis léxico supone un problema en el diseño de compiladores. Aunque la mayor parte del capítulo está dedicado al primer método, el diseño y uso de un generador automático, también se consideran técnicas útiles en el diseño manual. En la sección 3.4 se estudian los diagramas de transiciones, un concepto útil para la organización de un analizador léxico diseñado a mano.

Parejas de *buffers*

En muchos lenguajes fuente hay veces en que el analizador léxico necesita preanalizar varios caracteres, además del lexema para un patrón, antes de poder anunciar una concordancia. Los analizadores léxicos del capítulo 2 utilizaron la función `ungetc` para reinsertar caracteres preanalizados en la cadena de entrada. Como se puede consumir mucho tiempo moviendo caracteres, se han desarrollado técnicas especializadas en el manejo de *buffers* para reducir el número de operaciones necesarias para procesar un carácter de entrada. Se pueden emplear muchos esquemas de manejo de *buffers*, pero, como las técnicas dependen en cierto modo de los parámetros del sistema, aquí tan sólo se señalarán los principios que hay detrás de una clase de esquemas.

Se utiliza un *buffer* dividido en dos mitades de N caracteres cada una, tal como se indica en la figura 3.3. Por lo general, N es el número de caracteres en un bloque de disco, por ejemplo, 1024 ó 4096.

Se leen N caracteres de entrada en cada mitad del *buffer* con una orden de lectura de sistema, en vez de invocar una instrucción de lectura para cada carácter de entrada. Si quedan menos de N caracteres en la entrada, entonces se lee un carácter especial **eof** en el *buffer* después de los caracteres de entrada, como en la figura 3.3. Es decir, **eof** marca el final del archivo fuente y es distinto a cualquier carácter de la entrada.

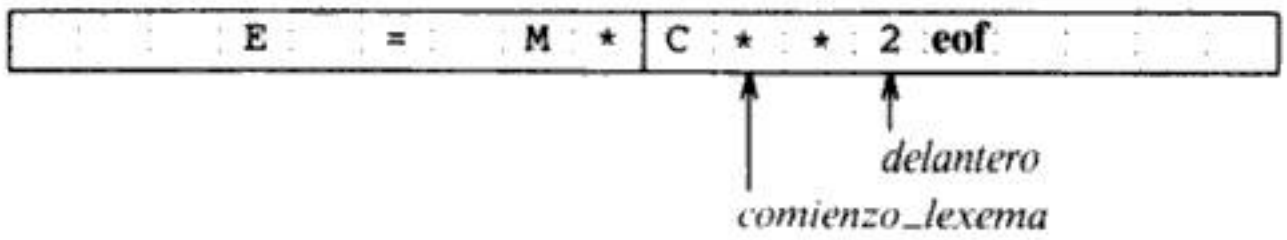


Fig. 3.3. Un *buffer* de entrada en dos mitades.

Se mantienen dos apuntadores al *buffer* de entrada. La cadena de caracteres entre los dos apuntadores es el lexema en curso. Al principio, los dos apuntadores apuntan al primer carácter del próximo lexema que hay que encontrar. Uno de ellos, llamado apuntador delantero, examina hacia adelante hasta encontrar una concordancia con un patrón. Una vez determinado el siguiente lexema, el apuntador delantero se coloca en el carácter de su extremo derecho. Después de haber procesado el lexema, ambos apuntadores se colocan en el carácter situado inmediatamente después del lexema. Con este esquema, se pueden considerar los comentarios y los espacios en blanco como patrones que no producen componentes léxicos.

Cuando el apuntador delantero está a punto de sobrepasar por la marca intermedia del *buffer*, se llena la mitad derecha con N nuevos caracteres de entrada. Cuando el apuntador delantero está a punto de sobrepasar el extremo derecho del *buffer*, se llena la mitad izquierda con N nuevos caracteres de entrada y el apuntador delantero se regresa al principio del *buffer*.

Este esquema de manejo de *buffers* casi siempre funciona muy bien, pero limita la cantidad de caracteres de preanálisis, y esto puede imposibilitar el reconocimiento de los componentes léxicos cuando la distancia recorrida por el apuntador delantero sea mayor que la longitud del *buffer*. Por ejemplo, si se encuentra

```
DECLARE ( ARG1, ARG2, . . . , ARGn )
```

en un programa de PL/I, no se puede determinar si `DECLARE` es una palabra clave o un nombre de arreglo hasta ver el carácter que sigue al paréntesis derecho. En cualquier caso, el lexema termina en la segunda `E`, pero la cantidad de preanálisis necesaria es proporcional al número de argumentos, que, en principio, es ilimitado.

```
if delantero está al final de la primera mitad then begin
    recargar la segunda mitad;
    delantero := delantero + 1
end
else if delantero está al final de la segunda mitad then begin
    recargar la primera mitad;
    pasar delantero al principio de la primera mitad
end
else delantero := delantero + 1;
```

Fig. 3.4. Código para avanzar el apuntador delantero.

Centinelas

Si se utiliza el esquema de la figura 3.3 tal como se ha expuesto, cada vez que se mueva el apuntador delantero se debe comprobar si se ha salido de una mitad del *buffer*; si así ocurriera se deberá recargar la otra mitad. Es decir, el código para hacer avanzar el apuntador delantero realiza pruebas como las que se muestran en la figura 3.4.

Excepto en los extremos de las mitades del *buffer*, el código de la figura 3.4 necesita dos pruebas para cada avance del apuntador delantero. Se pueden reducir estas dos pruebas a una si se amplía cada mitad del *buffer* para admitir un carácter *centinela* al final. El centinela es un carácter especial que no puede ser parte del programa fuente. Una elección natural es **eof** (fin de archivo, en inglés); en la figura 3.5 se muestra la misma disposición de *buffer* que la figura 3.3, añadidos los centinelas.

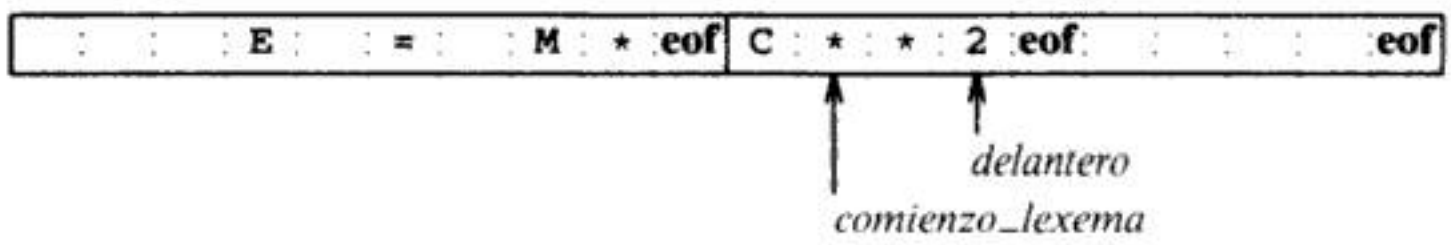


Fig. 3.5. Centinelas al final de cada mitad del *buffer*.

Con la disposición de la figura 3.5, se puede utilizar el código de la figura 3.6 para hacer avanzar el apuntador delantero (y determinar el final del archivo fuente). En la mayoría de las ocasiones el código realiza sólo una prueba para ver si *delantero* apunta hacia un carácter **eof**. Sólo se realizan más pruebas cuando se alcanza el final de una mitad del *buffer* o el final del archivo. Como se encuentran N caracteres de entrada entre los símbolos **eof**, el promedio de pruebas por cada carácter de entrada es próximo a 1.

```
delantero := delantero + 1;
if delantero↑ = eof then begin
    if delantero está al final de la primera mitad then begin
        recargar la segunda mitad;
        delantero := delantero + 1
    end
    else if delantero está al final de la segunda mitad then begin
        recargar la primera mitad;
        pasar delantero al principio de la primera mitad
    end
    else /* eof dentro de un buffer significa el final de la entrada */
        terminar el análisis léxico
end
```

Fig. 3.6. Código de preanálisis con centinelas.

También hay que decidir cómo procesar el carácter examinado por el apuntador delantero; ¿marca el fin de un componente léxico?, ¿representa un avance para la búsqueda de determinada palabra clave?, ¿o qué hace? Una forma de estructurar estas pruebas es utilizar una proposición **case**, si existe en el lenguaje de implantación. La prueba

if *delantero* ↑ = **eof**

se puede implantar entonces como uno de los distintos casos.

3.3 ESPECIFICACION DE LOS COMPONENTES LEXICOS

Las expresiones regulares son una notación importante para especificar patrones. Cada patrón concuerda con una serie de cadenas, de modo que las expresiones regulares servirán como nombres para conjuntos de cadenas. La sección 3.5 amplía esta notación a un lenguaje dirigido por patrones para el análisis léxico.

Cadenas y lenguajes

El término *alfabeto* o *clase de carácter* denota cualquier conjunto finito de símbolos. Ejemplos típicos de símbolos son las letras y los caracteres. El conjunto {`0, 1`} es el *alfabeto binario*. Los códigos ASCII y EBCDIC son dos ejemplos de alfabetos de computador.

Una *cadena* sobre algún alfabeto es una secuencia finita de símbolos tomados de ese alfabeto. En teoría del lenguaje, los términos *frase* y *palabra* a menudo se utilizan como sinónimos del término "cadena". La longitud de una cadena s, que suele escribirse $|s|$, es el número de apariciones de símbolos en s. Por ejemplo, `camino` es una cadena de longitud seis. La cadena *vacía*, representada por ϵ, es una cadena especial de longitud cero. En la figura 3.7 se recogen algunos términos comunes asociados con las partes de una cadena.

El término *lenguaje* se refiere a cualquier conjunto de cadenas de un alfabeto fijo. Esta definición es muy amplia, y abarca lenguajes abstractos como $\varnothing$, el conjunto *vacío*, o $\{\epsilon\}$ y el conjunto que sólo contiene la cadena vacía, así como al conjunto de todos los programas de Pascal sintácticamente bien formados y el conjunto de todas las oraciones en inglés gramaticalmente correctas, aunque los dos últimos conjuntos son mucho más difíciles de especificar. Obsérvese también que esta definición no atribuye ningún significado a las cadenas de un lenguaje. Los métodos para asignar significados a las cadenas se estudian en el capítulo 5.

Si x e y son cadenas, entonces la *concatenación* de x e y, que se escribe xy, es la cadena que resulta de agregar y a x. Por ejemplo, si $x =$ `caza` e $y =$ `fortunas`, entonces $xy =$ `cazafortunas`. La cadena vacía es el elemento identidad que se concatena. Es decir, $s\epsilon = \epsilon s = s$.

Cuando se considera la concatenación como un "producto", cabe definir la "exponenciación" de cadenas de la siguiente manera. Se define s^0 como ϵ, y para $i > 0$ se define s^i como $s^{i-1}s$. Dado que ϵs es s, $s^1 = s$. Entonces, $s^2 = ss$, $s^3 = sss$, etcétera.

TÉRMINO	DEFINICIÓN
prefijo de *s*	Una cadena que se obtiene eliminando cero o más símbolos desde la derecha de la cadena *s*; por ejemplo, `ban` es un prefijo de `bandera`.
sufijo de *s*	Una cadena que se forma suprimiendo cero o más símbolos desde la izquierda de una cadena *s*; por ejemplo, `era` es un sufijo de `bandera`.
subcadena de *s*	Una cadena que se obtiene suprimiendo un prefijo y un sufijo de *s*; por ejemplo, `ande` es una subcadena de `bandera`. Todo prefijo y sufijo de *s* es una cadena de *s*, pero no toda subcadena de *s* es un prefijo o un sufijo de *s*. Para toda cadena *s*, tanto *s* como ϵ son prefijos, sufijos y subcadenas de *s*.
prefijo, sufijo o subcadena *propios* de *s*	Cualquier cadena no vacía *x* que sea, respectivamente, un prefijo, sufijo o subcadena de *s* tal que $s \neq x$.
subsecuencia de *s*	Cualquier cadena formada mediante la eliminación de cero o más símbolos no necesariamente contiguos a *s*; por ejemplo, `bada` es una subsecuencia de `bandera`.

Fig. 3.7. Términos de partes de una cadena.

Operaciones aplicadas a lenguajes

Hay varias operaciones importantes que se pueden aplicar a los lenguajes. Para el análisis léxico, interesan principalmente la unión, la concatenación y la cerradura, definidas en la figura 3.8. También se puede extender el operador de "exponenciación" a los lenguajes definiendo L^0 como $\{\epsilon\}$, y L^i como $L^{i-1}L$. Por tanto, L^i es L concatenado consigo mismo $i-1$ veces.

Ejemplo 3.2 Sea L el conjunto {`A`, `B`, . . . , `Z`, `a`, `b`, . . . , `z`} y D el conjunto {`0`, `1`, . . . , `9`}. Se pueden considerar L y D de dos maneras: L, como el alfabeto que contiene el conjunto de letras mayúsculas y minúsculas, y D, como el alfabeto que contiene el conjunto de los diez dígitos decimales. Se puede dar el caso, puesto que un símbolo puede ser considerado como una cadena de longitud uno, de que L y D sean los dos lenguajes finitos. Los siguientes son algunos ejemplos de nuevos lenguajes creados a partir de L y D mediante la aplicación de los operadores definidos en la figura 3.8.

1. $L \cup D$ es el conjunto de letras y dígitos.
2. LD es el conjunto de cadenas que consta de una letra seguida de un dígito.
3. L^4 es el conjunto de todas las cadenas de cuatro letras.
4. L^* es el conjunto de todas las cadenas de letras, incluyendo ϵ, la cadena vacía.

OPERACIÓN	DEFINICIÓN
unión de L y M, que se escribe $L \cup M$	$L \cup M = \{ s \mid s \text{ está en } L \text{ o } s \text{ está en } M \}$
concatenación de L y M, que se escribe LM	$LM = \{ st \mid s \text{ está en } L \text{ y } t \text{ está en } M \}$
cerradura de Kleene de L, que se escribe L^*	$L^* = \bigcup_{i=0}^{\infty} L^i$ L^* denota "cero o más concatenaciones de" L.
cerradura positiva de L, que se escribe L^+	$L^+ = \bigcup_{i=1}^{\infty} L^i$ L^+ denota "una o más concatenaciones de" L.

Fig. 3.8. Definiciones de operaciones sobre lenguajes.

5. $L(L \cup D)^*$ es el conjunto de todas las cadenas de letras y dígitos que comienzan con una letra.
6. D^+ es el conjunto de todas las cadenas de uno o más dígitos. □

Expresiones regulares

En Pascal, un identificador es una letra seguida de cero o más letras o dígitos; es decir, un identificador es un miembro del conjunto definido en el apartado 5 del ejemplo 3.2. En esta sección, se presenta una notación, llamada expresiones regulares, que permite definir de manera precisa conjuntos como éste. Con esta notación, se pueden definir los identificadores de Pascal como

letra (letra | dígito) *

La barra vertical aquí significa "o", los paréntesis se usan para agrupar subexpresiones, el asterisco significa "cero o más casos de" la expresión entre paréntesis, y la yuxtaposición de **letra** con el resto de la expresión significa concatenación.

Una expresión regular se construye a partir de expresiones regulares más simples utilizando un conjunto de reglas definitorias. Cada expresión regular r representa un lenguaje $L(r)$. Las reglas de definición especifican cómo se forma $L(r)$ combinando de varias maneras los lenguajes representados por las subexpresiones de r.

Las siguientes son las reglas que definen las *expresiones regulares del alfabeto* $\sum$. Asociada a cada regla hay una especificación del lenguaje representado por la expresión regular que se está definiendo.

1. ϵ es una expresión regular designada por $\{\epsilon\}$; es decir, el conjunto que contiene la cadena vacía.
2. Si a es un símbolo de $\sum$, entonces a es una expresión regular designada por $\{a\}$; por ejemplo, el conjunto que contiene la cadena a. Aunque se usa la misma notación para las tres, técnicamente, la expresión regular a es distinta de la cadena

a o del símbolo *a*. El contexto aclarará si se habla de *a* como expresión regular, cadena o símbolo.

3. Suponiendo que *r* y *s* sean expresiones regulares representadas por los lenguajes $L(r)$ y $L(s)$, entonces,

   a) $(r) \mid (s)$ es una expresión regular representada por $L(r) \cup L(s)$.
   b) $(r)(s)$ es una expresión regular representada por $L(r)L(s)$.
   c) $(r)^*$ es una expresión regular representada por $(L(r))^*$.
   d) (r) es una expresión regular representada por $L(r)$.[2]

Se dice que un lenguaje designado por una expresión regular es un *conjunto regular*.

La especificación de una expresión regular es un ejemplo de definición recursiva. Las reglas 1 y 2 son la base de la definición; se usa el término *símbolo básico* para referirse a ϵ o a un símbolo de $\sum$ que aparezcan en una expresión regular. La regla 3 proporciona el paso inductivo.

Se pueden evitar los paréntesis innecesarios en las expresiones regulares si se adoptan las convenciones:

1. el operador unario * tiene la mayor precedencia y es asociativo por la izquierda,
2. la concatenación tiene la segunda mayor precedencia y es asociativa por la izquierda,
3. $\mid$ tiene la menor precedencia y es asociativo por la izquierda.

Según estas convenciones, $(a) \mid ((b)^*(c))$ es equivalente a $a|b^*c$. Estas dos expresiones designan el conjunto de cadenas que tienen una sola *a*, o cero o más *b* seguidas de una *c*.

Ejemplo 3.3 Sea $\sum = \{a, b\}$.

1. La expresión regular $a \mid b$ designa el conjunto $\{a, b\}$.
2. La expresión regular $(a \mid b)(a \mid b)$ se indica con $\{aa, ab, ba, bb\}$, el conjunto de todas las cadenas de *a* y *b* de longitud dos. Otra expresión regular para este mismo conjunto es $aa \mid ab \mid ba \mid bb$.
3. La expresión regular a^* designa el conjunto de todas las cadenas de cero o más *a*, por ejemplo, $\{\epsilon, a, aa, aaa, \ldots\}$.
4. La expresión regular $(a \mid b)^*$ designa el conjunto de todas las cadenas que contienen cero o más casos de una *a* o *b*, es decir, el conjunto de todas las cadenas de *a* y *b*. Otra expresión regular para este conjunto es $(a^* b^*)^*$.
5. La expresión regular $a \mid a^*b$ designa el conjunto que contiene la cadena *a* y todas las que se componen de cero o más *a* seguidas de una *b*. □

Si dos expresiones regulares *r* y *s* representan al mismo lenguaje, se dice que *r* y *s* son *equivalentes* y se escribe $r = s$. Por ejemplo, $(a \mid b) = (b \mid a)$.

---

[2] Esta regla establece que, si se desea, se pueden poner pares de paréntesis adicionales en torno a las expresiones regulares.

AXIOMA	DESCRIPCIÓN
$r \mid s = s \mid r$	$\mid$ es conmutativo
$r \mid (s \mid t) = (r \mid s) \mid t$	$\mid$ es asociativo
$(rs)t = r(st)$	la concatenación es asociativa
$r(s \mid t) = rs \mid rt$ $(s \mid t)r = sr \mid tr$	la concatenación distribuye sobre $\mid$
$\epsilon r = r$ $r \epsilon = r$	ϵ es el elemento identidad para la concatenación
$r^* = (r \mid \epsilon)^*$	la relación entre * y ϵ
$r^{**} = r^*$	* es idempotente

Fig. 3.9. Propiedades algebraicas de las expresiones regulares.

Son varias las leyes algebraicas que obedecen las expresiones regulares y pueden ser utilizadas para transformar las expresiones regulares a formas equivalentes. En la figura 3.9 se muestran algunas leyes algebraicas que se cumplen para las expresiones regulares r, s y t.

Definiciones regulares

Por conveniencia de notación, puede ser deseable dar nombres a las expresiones regulares y definir expresiones regulares utilizando dichos nombres como si fueran símbolos. Si $\sum$ es un alfabeto de símbolos básicos, entonces una *definición regular* es una secuencia de definiciones de la forma

$$d_1 \rightarrow r_1$$
$$d_2 \rightarrow r_2$$
$$\cdots$$
$$d_n \rightarrow r_n$$

donde cada d_i es un nombre distinto, y cada r_i es una expresión regular sobre los símbolos de $\sum \cup \{d_1, d_2, \ldots, d_{i-1}\}$, por ejemplo, los símbolos básicos y los nombres previamente definidos. Al limitar cada r_i a los símbolos de $\sum$ y a los nombres previamente definidos, se puede construir una expresión regular en $\sum$ para cualquier r_i, reemplazando una y otra vez los nombres de las expresiones regulares por las expresiones que designan. Si r_i utilizara d_j para alguna $j \geqslant i$, entonces r_i se podría definir recursivamente y este proceso de sustitución no tendría fin.

Para distinguir los nombres de los símbolos, se imprimen en **negritas** los nombres de las definiciones regulares.

Ejemplo 3.4. Como ya se estableció antes, el conjunto de identificadores de Pascal es el conjunto de cadenas de letras y dígitos que empiezan con una letra. A continuación se da una definición regular para este conjunto.

letra → `A` | `B` | `. . .` | `Z` | `a` | `b` | `. . .` | `z`
dígito → `0` | `1` | `. . .` | `9`
id → **letra** (**letra** | **dígito**)* □

Ejemplo 3.5. Los números sin signo en Pascal son cadenas, como `5280`, `39.37`, `6.336E4`, o `1.894E-4`. La siguiente definición regular proporciona una especificación precisa para esta clase de cadenas:

dígito → `0` | `1` | `. . .` | `9`
dígitos → **dígito** **dígito***
fracción_optativa → `.` **dígitos** | ϵ
exponente_optativo → (`E` (`+` | `-` | ϵ) **dígitos**) | ϵ
núm → **dígitos** **fracción_optativa** **exponente_optativo**

Esta definición establece que una **fracción_optativa** es un punto decimal seguido de uno o más dígitos, o está ausente (la cadena vacía). Un **exponente_optativo**, si no está ausente, es una `E` seguida de un signo + o − opcional, seguido de uno o más dígitos. Obsérvese que, como mínimo, debe ir un dígito después del punto, de modo que **núm** no concuerde con `1`, pero sí con `1.0`. □

Abreviaturas en la notación

Ciertas construcciones aparecen con tanta frecuencia en una expresión regular, que es conveniente introducir algunas abreviaturas.

1. *Uno o más casos.* El operador unitario postfijo $^+$ significa "uno o más casos de". Si r es una expresión regular que designa al lenguaje $L(r)$, entonces $(r)^+$ es una expresión regular que designa al lenguaje $(L(r))^+$. Así, la expresión regular a^+ representa al conjunto de todas las cadenas de una o más a. El operador $^+$ tiene la misma precedencia y asociatividad que el operador *. Las dos identidades algebraicas $r^* = r^+ | \epsilon$ y $r^+ = rr^*$ relacionan los operadores de la cerradura de Kleene y los de la cerradura positiva.

2. *Cero o un caso.* El operador unitario postfijo ? significa "cero o un caso de". La notación $r?$ es una abreviatura de $r | \epsilon$. Si r es una expresión regular, entonces $(r)?$ es una expresión regular que designa el lenguaje $L(r) \cup \{\epsilon\}$. Por ejemplo, usando los operadores $^+$ y ?, se puede reescribir la definición regular para **núm** del ejemplo 3.5 en la forma

dígito → `0` | `1` | `. . .` | `9`
dígitos → **dígito**$^+$
fracción_optativa → (`.` **dígitos**)?
exponente_optativo → (`E` (`+` | `-`)? **dígitos**)?
núm → **dígitos** **fracción_optativa** **exponente_optativo**

3. *Clases de caracteres.* La notación `[abc]`, donde `a`, `b` y `c` son símbolos del alfabeto, designa la expresión regular `a` | `b` | `c`. Una clase abreviada de carácter como `[a-z]` designa la expresión regular `a` | `b` | `. . .` | `z`. Utilizando cla-

ses de caracteres, se puede definir los identificadores como cadenas generadas por la expresión regular

```
[A-Za-z][A-Za-z0-9]*
```

Conjuntos no regulares

Algunos lenguajes no se pueden describir con ninguna expresión regular. Para ilustrar los límites del poder descriptivo de las expresiones regulares, se dan a continuación ejemplos de construcciones de lenguajes de programación que no se pueden describir con expresiones regulares. En las referencias están las pruebas de tales afirmaciones.

No se pueden utilizar las expresiones regulares para describir construcciones equilibradas o anidadas. Por ejemplo, el conjunto de todas las cadenas de paréntesis equilibrados no se puede describir con una expresión regular. Por otra parte, este conjunto se puede especificar mediante una gramática independiente del contexto.

Las cadenas de repetición no se pueden describir con expresiones regulares. El conjunto

$$\{\, wcw \mid w \text{ es una cadena de símbolos } a \text{ y } b \,\}$$

no se puede representar con ninguna expresión regular, ni se puede describir con una gramática independiente del contexto.

Las expresiones regulares se pueden utilizar para designar sólo un número fijo de repeticiones o un número no especificado de repeticiones de una determinada construcción. No se pueden comparar dos números arbitrarios para comprobar si son iguales. Por tanto, las cadenas Hollerith de la forma $n\text{H}a_1 a_2 \ldots a_n$, pertenecientes a las primeras versiones de FORTRAN, no se pueden describir con una expresión regular, pues el número de caracteres que sigue a H debe concordar con el número decimal n que precede a H.

3.4 RECONOCIMIENTO DE COMPONENTES LEXICOS

En la sección anterior, se trató el problema de cómo especificar los componentes léxicos. En esta sección, se estudia el reconocimiento de los componentes léxicos y se utiliza como ejemplo el lenguaje generado por la siguiente gramática.

Ejemplo 3.6. Considérese el siguiente fragmento gramatical:

$$\begin{aligned}
\textit{prop} &\rightarrow \textbf{if } \textit{expr} \textbf{ then } \textit{prop} \\
&\mid \textbf{if } \textit{expr} \textbf{ then } \textit{prop} \textbf{ else } \textit{prop} \\
&\mid \boldsymbol{\epsilon} \\
\textit{expr} &\rightarrow \textit{término} \textbf{ oprel } \textit{término} \\
&\mid \textit{término} \\
\textit{término} &\rightarrow \textbf{id} \\
&\mid \textbf{núm}
\end{aligned}$$

donde los terminales **if**, **then**, **else**, **oprel**, **id** y **núm** generan conjuntos de cadenas dados por las siguientes definiciones regulares:

if → `if`
then → `then`
else → `else`
oprel → `<` | `<=` | `=` | `<>` | `>` | `>=`
id → **letra** (**letra** | **dígito**)*
núm → **dígito**$^+$ (`.` **dígito**$^+$)? (`E` (`+` | `-`)? **dígito**$^+$)?

donde **letra** y **dígito** se han definido anteriormente.

Para este fragmento de lenguaje, el analizador léxico reconocerá las palabras clave `if`, `then`, `else`, al igual que los lexemas representados por **oprel, id** y **núm**. Para simplificar las cosas, se supone que las palabras clave son reservadas; es decir, no se pueden usar como identificadores. Como en el ejemplo 3.5, **núm** representa los números enteros y reales sin signo de Pascal.

Además, se supone que los lexemas están separados por espacio en blanco, formados por secuencias no nulas de espacios en blanco, caracteres TAB y caracteres de nueva línea. El analizador léxico eliminará los espacios en blanco. Esto lo hará comparando una cadena con la definición de la expresión regular **eb** siguiente.

delim → **blanco** | **tab** | **líneanueva**
eb → **delim**$^+$

Si se encuentra una concordancia para **eb**, el analizador léxico no devuelve un componente léxico al analizador sintáctico, sino que se dispone a encontrar un componente léxico a continuación del espacio en blanco y lo devuelve al analizador sintáctico.

El objetivo es construir un analizador léxico que aísle el lexema para el siguiente componente léxico del *buffer* de entrada y que produzca como salida un par for-

EXPRESIÓN REGULAR	COMPONENTE LÉXICO	VALOR DEL ATRIBUTO
eb	–	–
`if`	**if**	–
`then`	**then**	–
`else`	**else**	–
id	**id**	apuntador a la entrada en la tabla
núm	**núm**	apuntador a la entrada en la tabla
`<`	**oprel**	`MEN`
`<=`	**oprel**	`MEI`
`=`	**oprel**	`IGU`
`<>`	**oprel**	`DIF`
`>`	**oprel**	`MAY`
`>=`	**oprel**	`MAI`

Fig. 3.10. Patrones de expresiones regulares para componentes léxicos.

mado por el componente léxico apropiado y el valor de atributo, utilizando la tabla de traducción de la figura 3.10. Los valores de atributo para los operadores relacionales están dados por las constantes simbólicas `MEN`, `MEI`, `IGU`, `MAY`, `MAI`. □

Diagramas de transiciones

Como paso intermedio en la construcción de un analizador léxico, primero se produce un diagrama de flujo estilizado, llamado *diagrama de transiciones*. Los diagramas de transiciones representan las acciones que tienen lugar cuando el analizador léxico es llamado por el analizador sintáctico para obtener el siguiente componente léxico, como sugiere la figura 3.1. Supóngase que el *buffer* de entrada es como el de la figura 3.3 y que el apuntador del principio del lexema apunta al carácter que sigue al último lexema encontrado. Se utiliza un diagrama de transición para localizar la información sobre los caracteres que se detectan a medida que el apuntador delantero examina la entrada. Esto se hace cambiando de posición en el diagrama según se leen los caracteres.

Las posiciones en un diagrama de transición se representan con un círculo y se llaman *estados*. Los estados se conectan mediante flechas, llamadas *aristas*. Las aristas que salen del estado s tienen etiquetas que indican los caracteres de entrada que pueden aparecer después de haber llegado el diagrama de transición al estado s. La etiqueta **otro** se refiere a todo carácter que no haya sido indicado por ninguna de las otras aristas que salen de s.

Se supone que los diagramas de transiciones de esta sección son *deterministas;* es decir, ningún símbolo puede concordar con las etiquetas de dos aristas que salgan de un estado. Al empezar la sección 3.5, se hará menos rigurosa esta condición, facilitando las cosas al diseñador del analizador léxico y, con las herramientas adecuadas, también al encargado de su implantación.

Un estado se etiqueta como el estado de *inicio*; es en el estado inicial del diagrama de transición donde reside el control cuando se empieza a reconocer un componente léxico. Ciertos estados pueden tener acciones que se ejecutan cuando el flujo del control alcanza dicho estado. Al entrar en un estado se lee el siguiente carácter de entrada. Si hay una arista del estado en curso de ejecución cuya etiqueta concuerde con ese carácter de entrada, entonces se va al estado apuntado por la arista. De otro modo, se indica un fallo.

En la figura 3.11 se muestra un diagrama de transiciones para los patrones >= y >. El diagrama de transiciones funciona de la siguiente forma. Su estado de inicio es el estado 0. En el estado 0 se lee el siguiente carácter de entrada. La arista etiquetada con > del estado 0 se debe seguir hasta el estado 6 si este carácter de entrada es >. De otro modo, significa que no se habrá reconocido ni > ni >=.

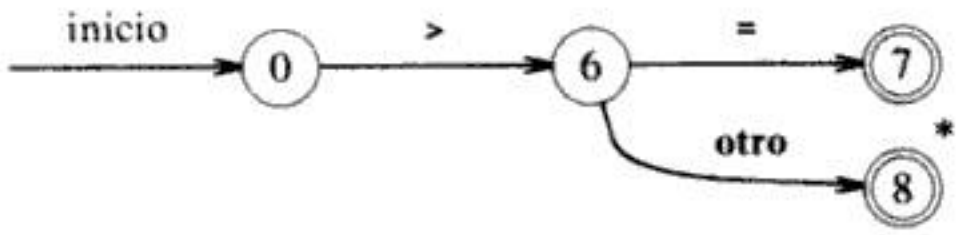


Fig. 3.11. Diagrama de transiciones para >=.

Al llegar al estado 6 se lee el siguiente carácter de entrada. La arista etiquetada con = que sale del estado 6 deberá seguirse hasta el estado 7 si este carácter de entrada es un =. De otro modo, la arista etiquetada con **otro** indica que se deberá ir al estado 8. El círculo doble del estado 7 indica que éste es un estado de aceptación, un estado en el cual se ha encontrado el componente léxico >=.

Obsérvese que el carácter > y otro carácter adicional se leen a medida que se sigue la secuencia de aristas desde el estado inicial al estado de aceptación 8. Como el carácter adicional no es parte del operador relacional >, se debe retroceder un carácter el apuntador delantero. Se usa un * para indicar los estados en que se debe llevar a cabo este retroceso en la entrada.

En general, puede haber varios diagramas de transiciones, cada uno de los cuales especifique un grupo de componentes léxicos. Si surge un fallo mientras se está siguiendo un diagrama de transiciones, se debe retroceder el apuntador delantero hasta donde estaba en el estado inicial de dicho diagrama, y activar el siguiente diagrama de transiciones. Dado que los apuntadores de inicio de lexema y los delanteros marcaban la misma posición en el estado inicial del diagrama, se retrocede el apuntador delantero hasta la posición que marca el apuntador al inicio de lexema. Si el fallo surge en todos los diagramas de transiciones, es que se ha detectado un error léxico y se invoca una rutina de recuperación de errores.

Ejemplo 3.7. En la figura 3.12 se muestra un diagrama de transiciones para el componente léxico **oprel**. Obsérvese que la figura 3.11 es una parte de este diagrama de transición más complejo. □

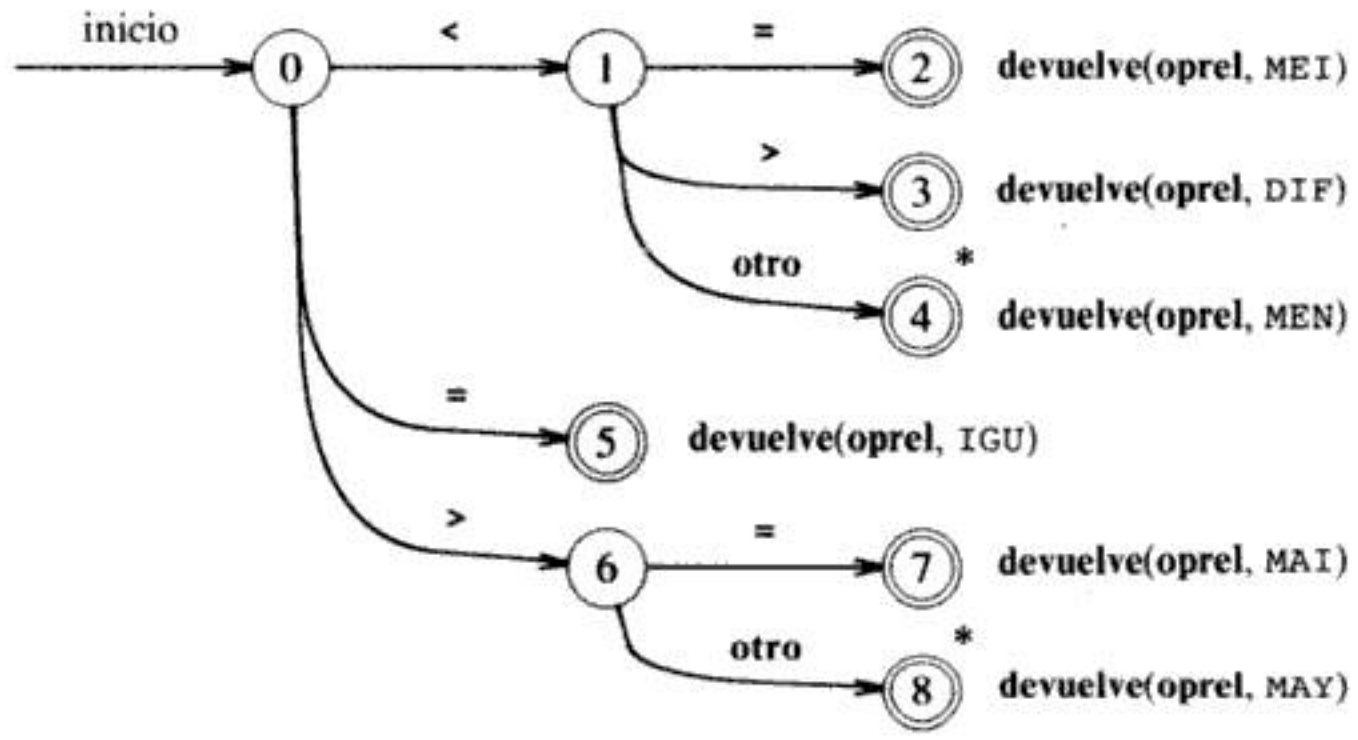


Fig. 3.12. Diagrama de transiciones para operadores relacionales.

Ejemplo 3.8. Como las palabras clave son secuencias de letras, son la excepción a la regla de que una secuencia de letras y dígitos que comience con una letra es un identificador. En vez de codificar las excepciones en un diagrama de transiciones, es mejor considerar las palabras clave como identificadores especiales, como en la sección 2.7. Cuando se llega al estado de aceptación de la figura 3.13, se ejecuta algún código para determinar si el lexema que condujo al estado de aceptación es una palabra clave o un identificador.

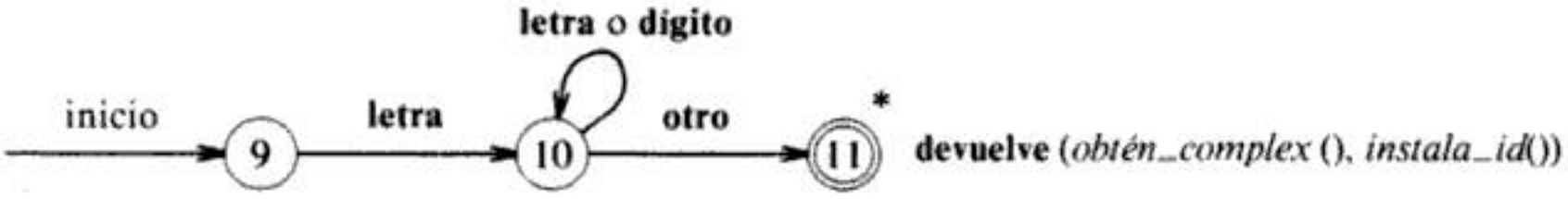


Fig. 3.13. Diagrama de transiciones para identificadores y palabras clave.

Una técnica sencilla para separar las palabras clave de los identificadores es inicializar adecuadamente la tabla de símbolos en donde se guarda la información sobre identificadores. Para los componentes léxicos de la figura 3.10, es necesario introducir las cadenas `if`, `then` y `else` en la tabla de símbolos antes de que aparezca ningún carácter en la entrada. También se pone una nota en la tabla de símbolos del componente léxico que debe ser devuelto cuando se reconoce una de estas cadenas. La proposición *devuelve* después del estado de aceptación de la figura 3.13 utiliza *obtén_complex*() e *instala_id*() para obtener el componente léxico y el valor de atributo, respectivamente, que deben ser devueltos. El procedimiento *instala_id*() tiene acceso al *buffer*, en donde está localizado el lexema del identificador. Se examina la tabla de símbolos y si se encuentra el lexema marcado como una palabra clave, *instala_id*() devuelve 0. Si se encuentra el lexema y es una variable de programa, *instala_id*() devuelve un apuntador a la entrada de la tabla de símbolos. Si el lexema no se encuentra en la tabla de símbolos, se instala como una variable y se devuelve un apuntador a la entrada recién creada.

El procedimiento *obtén_complex* busca el lexema de forma parecida en la tabla de símbolos. Si el lexema es una palabra clave, se devuelve el correspondiente componente léxico; si no, se devuelve el componente léxico *id*.

Obsérvese que el diagrama de transiciones no se modifica si se reconocen palabras clave adicionales; simplemente se inicializa la tabla de símbolos con las cadenas y componentes léxicos de las palabras clave adicionales. □

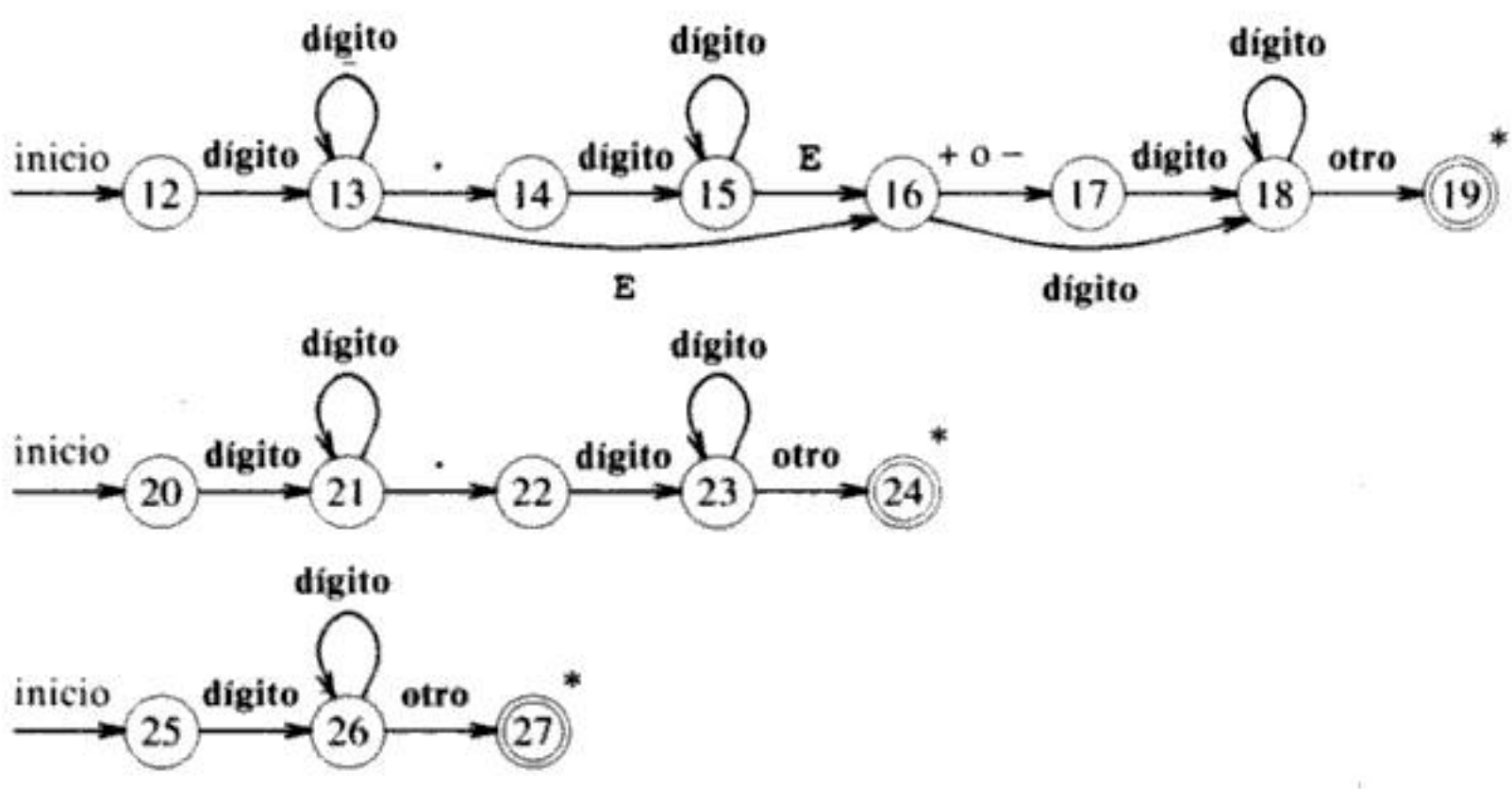


Fig. 3.14. Diagramas de transiciones para números sin signo en Pascal.

La técnica de colocar las palabras clave en la tabla de símbolos es casi indispensable cuando el analizador léxico se codifica manualmente. De no hacerlo así, el número de estados en un analizador léxico para un lenguaje de programación típico es de varios cientos, mientras que utilizando el truco, quizá baste con menos de cien estados.

Ejemplo 3.9. Cuando se construye un reconocedor de números sin signo dados por la definición regular

$$\textbf{núm} \rightarrow \textbf{dígito}^{+}\ (\textbf{.dígito}^{+})?\ (\mathrm{E}(+\ |\ -)?\ \textbf{dígito}^{+})?$$

surgen varias cuestiones. Obsérvese que la definición tiene la forma **dígitos fracción**? **exponente**?, donde **fracción** y **exponente** son opcionales.

El lexema de un determinado componente léxico debe ser el más grande posible. Por ejemplo, el analizador léxico no debe detenerse después de aparecer `12` o incluso `12.3` cuando la entrada es `12.3E4`. Empezando en los estados `25`, `20` y `12` de la figura 3.14, los estados de aceptación se alcanzarán después de que hayan aparecido `12`, `12.3` y `12.3E4`, respectivamente, suponiendo que `12.3E4` vaya seguido de un carácter distinto de dígito en la entrada. Los diagramas de transiciones con estados de inicio `25`, `20` y `12` son para **dígitos**, **dígitos fracción** y **dígitos fracción**? **exponente**, respectivamente, de modo que se deben probar los estados de inicio en el orden inverso `12`, `20`, `25`.

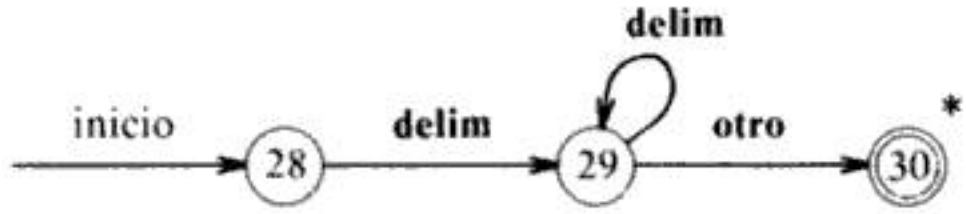
La acción, cuando se llega a cualquiera de los estados de aceptación `19`, `24` ó `27`, es llamar al procedimiento *instala_núm*, que introduce el lexema en una tabla de números y devuelve un apuntador a la entrada creada. El analizador léxico devuelve el componente léxico **núm** con este apuntador como valor léxico. □

Se puede utilizar la información sobre el lenguaje que no está en las definiciones regulares de los componentes léxicos para señalar los errores en la entrada. Por ejemplo, con la entrada `1.<x`, se fracasa en los estados `14` y `22` de la figura 3.14 con < como siguiente carácter de entrada. En vez de devolver el número `1`, se podría informar de un error y continuar como si la entrada fuera `1.0<x`. Este conocimiento se puede usar también para simplificar los diagramas de transiciones, puesto que el manejo de errores se puede utilizar para recuperarse de algunas situaciones que de otro modo conducirían a un fallo.

Hay varias formas de evitar la concordancia redundante de los diagramas de transiciones de la figura 3.14. Una de ellas consiste en reescribir los diagramas de transiciones combinándolos en uno, tarea generalmente laboriosa. Otra forma es modificar la respuesta al fallo durante el proceso de seguimiento de un diagrama. Un método estudiado más adelante en este capítulo permite atravesar varios estados de aceptación; se retrocede al último estado de aceptación que se atravesó cuando se produjo el fallo.

Ejemplo 3.10. Uniendo los diagramas de transiciones de las figuras 3.12, 3.13 y 3.14, se puede obtener una secuencia de diagramas de transiciones para todos los componentes léxicos del ejemplo 3.6. Los estados de inicio con números menores deben ser alcanzados antes que los estados de inicio con números mayores.

La única cuestión que resta es la relacionada con el espacio en blanco. El tratamiento de **eb**, que representa el espacio en blanco, es distinto del de los patrones analizados anteriormente, porque no se devuelve nada al analizador sintáctico cuando se encuentra el espacio en blanco en la entrada. Un diagrama de transiciones que reconoce a **eb** por sí mismo es



No se devuelve nada cuando se alcanza el estado de aceptación; simplemente se regresa al estado inicial del primer diagrama de transiciones para buscar otro patrón.

Siempre que sea posible, es mejor buscar componentes léxicos que aparezcan con mayor frecuencia antes que los menos frecuentes, porque a un diagrama de transiciones sólo se llega después de haber fallado en todos los diagramas anteriores. Como se espera que el espacio en blanco aparezca con frecuencia, es mejor poner el diagrama de transiciones para el espacio en blanco cerca del principio que probar el espacio en blanco al final. □

Implantación de un diagrama de transiciones

Una secuencia de diagramas de transiciones se puede convertir en un programa que busque los componentes léxicos especificados por los diagramas. Se adopta un enfoque sistemático que sirve para todos los diagramas de transiciones y que construye programas cuyo tamaño es proporcional al número de estados y de aristas de los diagramas.

A cada estado le corresponde un segmento de código. Si hay aristas que salen de un estado, entonces su código lee un carácter y selecciona una arista para seguir, si es posible. Se utiliza la función `sigtecar()` para leer el siguiente carácter del *buffer* de entrada, para avanzar el apuntador delantero en cada llamada y para devolver el carácter leído[3]. Si hay una arista etiquetada con el carácter leído, o etiquetada con una clase de caracteres que contenga al carácter leído, entonces el control se transfiere al código del estado apuntado por esa arista. Si no hay tal arista y el estado en curso de ejecución no es el que indica que se ha encontrado un componente léxico, entonces se llama a la rutina `fallo()` para hacer retroceder el apuntador delantero a la posición del apuntador al comienzo e iniciar la búsqueda del componente léxico especificado por el siguiente diagrama de transiciones. Si no hay que probar más diagramas de transiciones, `fallo()` llama a una rutina de recuperación de errores.

Para devolver los componentes léxicos se utiliza la variable global `valor_léxico`, a la que se asignan los apuntadores devueltos por las funciones `instala_id()` e `instala_núm()` cuando se encuentra un identificador o un número, respectivamente. Se devuelve la clase del componente léxico por el procedimiento principal del analizador léxico, llamado `sigte_complex()`.

---

[3] Una implantación más eficiente utilizaría una macro en línea en lugar de la función `sigtecar()`.

Se utiliza una proposición **case** para encontrar el estado inicial del siguiente diagrama de transiciones. En la aplicación en C de la figura 3.15, dos variables, `estado` e `inicio`, recorren el estado actual y el estado inicial del diagrama de transiciones en curso. Los números de estado del código son para los diagramas de transiciones de las figuras 3.12 a 3.14.

Las aristas de los diagramas de transiciones se encuentran seleccionando repetidamente el fragmento de código para un estado y ejecutando ese fragmento de código para determinar el siguiente estado, como se indica en la figura 3.16. Se muestran el código para el estado 0, tal como se modificó en el ejemplo 3.10 para considerar los espacios en blanco, y el código para dos de los diagramas de transiciones de las figuras 3.13 y 3.14. Obsérvese que la construcción en C

`while(1)` *prop*

repite *prop* "indefinidamente", es decir, hasta que aparezca una instrucción `return`.

```
int estado = 0, inicio = 0;
int valor_léxico;
     /* para "devolver" el segundo componente del componente
        léxico */

int fallo()
{
     delantero = inicio_lexema;
     switch (inicio) {
          case 0:  inicio = 9; break;
          case 9:  inicio = 12; break;
          case 12: inicio = 20; break;
          case 20: inicio = 25; break;
          case 25: recupera(); break;
          default: /* error del compilador */
     }
}
```

Fig. 3.15. Código en C para encontrar el siguiente estado de inicio.

Como en C no se permite que se devuelvan un componente léxico y un valor de atributo, `instala_id()` e `instala_núm()` asignan en consecuencia alguna variable global al valor de atributo correspondiente a la entrada en la tabla del **id** o **núm** en cuestión.

Si el lenguaje de implantación no tiene una proposición **case**, se puede crear una matriz para cada estado, indexada por caracteres. Si *estado*1 es una matriz de este tipo, entonces *estado*1[*c*] es un apuntador a una parte de código que debe ejecutarse siempre que el carácter de preanálisis sea *c*. Por lo general, este código finalizaría con una instrucción **goto** al código para el siguiente estado. La matriz del estado *s* se conoce como tabla de transferencia indirecta de *s*.

```
complex sigte_complex()
{     while(1) {
            switch (estado) {
            case 0:     c = sigtecar();
                 /* c es el carácter de preanálisis */
                 if (c==blanco || c==tab || c==línea_nueva) {
                      estado = 0;
                      inicio_lexema++;
                           /* se avanza el inicio del lexema */
                 }
                 else if (c == '<') estado = 1;
                 else if (c == '=') estado = 5;
                 else if (c == '>') estado = 6;
                 else estado = fallo();
                 break;
                 .../* aquí van los casos del 1 al 8 */
            case 9:     c = sigtecar();
                 if (isletter(c)) estado = 10;
                 else estado = fallo();
                 break;
            case 10:    c = sigtecar();
                 if (isletter(c)) estado = 10;
                 else if (isdigit(c)) estado = 10;
                 else estado = 11;
                 break;
            case 11:    regresa(1); instala_id;
                 return ( obtén_complex() );
                 .../* aquí van los casos del 12 al 24 */
            case 25:    c = sigtecar();
                 if (isdigit(c)) estado = 26;
                 else estado = fallo();
                 break;
            case 26:    c = sigtecar();
                 if (isdigit(c)) estado = 26;
                 else estado = 27;
                 break;
            case 27:    regresa(1); instala_núm();
                 return( NUM );
            }
      }
}
```

Fig. 3.16. Código en C para el analizador léxico.

3.5 UN LENGUAJE PARA LA ESPECIFICACION DE ANALIZADORES LEXICOS

Se han desarrollado algunas herramientas para construir analizadores léxicos a partir de notaciones de propósito especial basadas en expresiones regulares. Ya se ha estudiado el uso de expresiones regulares en la especificación de patrones de componentes léxicos. Antes de considerar los algoritmos para compilar expresiones regulares en programas de concordancia de patrones, se da un ejemplo de una herramienta que pueda ser utilizada por dicho algoritmo.

En esta sección se describe una herramienta concreta, llamada LEX, muy utilizada en la especificación de analizadores léxicos para varios lenguajes. Esa herramienta se denomina *compilador LEX*, y la especificación de su entrada, *lenguaje LEX*. El estudio de una herramienta existente permitirá mostrar cómo, utilizando expresiones regulares, se puede combinar la especificación de patrones con acciones, por ejemplo, haciendo entradas en una tabla de símbolos, cuya ejecución se pueda pedir a un analizador léxico. Se pueden utilizar las especificaciones tipo LEX aunque no se disponga de un compilador LEX; las especificaciones se pueden transcribir manualmente a un programa operativo empleando las técnicas de diagramas de transiciones de la sección anterior.

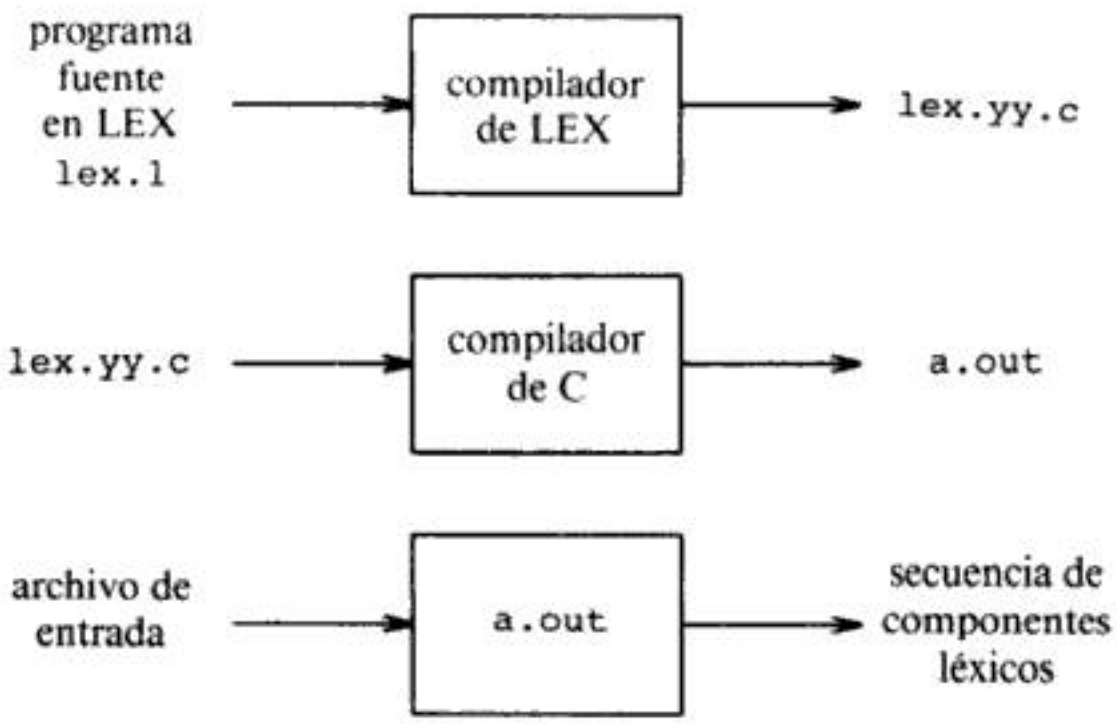


Fig. 3.17. Creación de un analizador léxico con LEX.

Por lo general, se utiliza el LEX de la forma representada en la figura 3.17. Primero, se prepara una especificación del analizador léxico creando un programa `lex.l` en lenguaje LEX. Después `lex.l` se pasa por el compilador LEX para producir el programa en C `lex.yy.c`. El programa `lex.yy.c` consta de una representación tabular de un diagrama de transiciones construido a partir de las expresiones regulares de `lex.l`, junto con una rutina estándar que utiliza la tabla para reconocer lexemas. Las acciones asociadas a las expresiones regulares de `lex.l` son partes de código en C y se transfieren directamente a `lex.yy.c`. Por último,

`lex.yy.c` se ejecuta en el compilador de C para producir un programa objeto `a.out`, que es el analizador léxico que transforma un archivo de entrada en una secuencia de componentes léxicos.

Especificaciones en LEX

Un programa en LEX consta de tres partes:

```
declaraciones
%%
reglas de traducción
%%
procedimientos auxiliares
```

La sección de declaraciones incluye declaraciones de variables, constantes manifiestas y definiciones regulares. (Una constante manifiesta es un identificador que se declara para representar una constante.) Las definiciones regulares son proposiciones similares a las estudiadas en la sección 3.2, y se utilizan como componentes de las expresiones regulares que aparecen en las reglas de traducción.

Las reglas de traducción de un programa en LEX son proposiciones de la forma

$$\begin{array}{ll} p_1 & \{\ acción_1\ \} \\ p_2 & \{\ acción_2\ \} \\ \ldots & \ldots \\ p_n & \{\ acción_n\ \} \end{array}$$

donde p_i es una expresión regular y cada *acción* es un fragmento de programa que describe cuál ha de ser la acción del analizador léxico cuando el patrón p_i concuerda con un lexema. En LEX, las acciones se escriben en C, en general, sin embargo, pueden estar en cualquier lenguaje de implantación.

La tercera sección contiene todos los procedimientos auxiliares que puedan necesitar las acciones. A veces, estos procedimientos se pueden compilar por separado y cargar con el analizador léxico.

Un analizador léxico creado por LEX se comporta en sincronía con un analizador sintáctico como sigue. Cuando es activado por el analizador sintáctico, el analizador léxico comienza a leer su entrada restante, un carácter a la vez, hasta que encuentre el mayor prefijo de la entrada que concuerde con una de las expresiones regulares p_i. Entonces, ejecuta la $acción_i$. Generalmente, $acción_i$ devolverá el control al analizador sintáctico. Sin embargo, si no lo hace, el analizador léxico se dispone a encontrar más lexemas, hasta que una acción hace que el control regrese al analizador sintáctico. La búsqueda repetida de lexemas hasta encontrar una instrucción **return** explícita permite al analizador léxico procesar espacios en blanco y comentarios de manera apropiada.

El analizador léxico devuelve una única cantidad, el componente léxico, al analizador sintáctico. Para pasar un valor de atributo con la información del lexema, se puede asignar una variable global llamada `yylval`.

Ejemplo 3.11. La figura 3.18 es un programa en LEX que reconoce los componentes léxicos de la figura 3.10 y devuelve el componente léxico encontrado. Algunas observaciones sobre el código servirán para introducir muchas características importantes de LEX.

```
%{
     /* definición de las constantes manifiestas
        MEN, MEI, IGU, DIF, MAY, MAI,
        IF, THEN, ELSE, ID, NUMERO, OPREL */
%}

/* definiciones regulares */
delim       [ \t\n]
eb          {delim}+
letra       [A-Za-z]
dígito      [0-9]
id          {letra}({letra} | {dígito})*
número      {dígito}+(\.{dígito}+)?(E[+\-]?{dígito}+)?

%%

{eb}        {/* no hay acción ni se devuelve nada */}
if          {return(IF);}
then        {return(THEN);}
else        {return(ELSE);}
{id}        {yylval = instala_id(); return(ID);}
{número}    {yylval = instala_núm; return(NUMERO);}
"<"         {yylval = MEN; return(OPREL);}
"<="        {yylval = MEI; return(OPREL);}
"="         {yylval = IGU; return(OPREL);}
"<>"        {yylval = DIF; return(OPREL);}
">"         {yylval = MAY; return(OPREL);}
">="        {yylval = MAI; return(OPREL);}

%%

instala_id() {
     /* procedimiento para instalar el lexema, cuyo primer
        carácter está apuntado por yytexto y cuya longitud es
        yylong, dentro de la tabla de símbolos y devuelve un
        apuntador a él */
}

instala_núm() {
     /* procedimiento similar para instalar un lexema que es
        un número */
}
```

Fig. 3.18. Programa en LEX para los componentes léxicos de la figura 3.10.

En la sección de declaraciones, aparece (un lugar para) la declaración de ciertas constantes manifiestas utilizadas por las reglas de traducción[4]. Estas declaraciones están encerradas entre llaves especiales `%{` y `%}`. Todo lo que aparezca entre estas llaves se copia directamente en el analizador léxico `lex.yy.c`, y no se considera como parte de las definiciones regulares ni de las reglas de traducción. Los procedimientos auxiliares de la tercera sección tienen asignado exactamente el mismo tratamiento. En la figura 3.18 hay dos procedimientos, `instala_id` e `instala_núm`, que son utilizados por las reglas de traducción, y que se copiarán literalmente en `lex.yy.c`.

En la sección de definiciones también se incluyen algunas definiciones regulares. Cada una de estas definiciones consiste en un nombre y una expresión regular representada por ese nombre. Por ejemplo, el primer nombre definido es `delim`, que representa a la clase de caracteres `[ \t\n]`, es decir, cualquiera de los tres símbolos, blanco, TAB (representado por `\t`), o nueva línea. La segunda definición es la de espacio en blanco, representada por el nombre `eb`. El espacio en blanco es toda secuencia de uno o más caracteres delimitadores. Obsérvese que en LEX la palabra `delim` debe estar entre llaves para distinguirla del patrón que consta de las cinco letras `delim`.

En la definición de `letra`, se ve el uso de una clase de caracteres. La abreviatura `[A-Za-z]` representa cualquiera de las letras mayúsculas de la `A` a la `Z` o de las letras minúsculas de la `a` a la `z`. La quinta definición, de `id`, usa paréntesis, que son metasímbolos de LEX, con su significado natural de agrupadores. Igualmente, la barra vertical es un metasímbolo de LEX que significa unión.

En la última definición regular, de `número`, se observa algún otro detalle. `?` se utiliza como metasímbolo, con su significado habitual de "cero o un caso de". También se observa la diagonal invertida utilizada como salida, para permitir que un carácter que sea metasímbolo de LEX tenga su significado natural. En particular, el punto decimal en la definición de `número` se expresa con un `\.`, porque un punto por sí mismo representa la clase de caracteres de todos los caracteres salvo la nueva línea, tanto en LEX como en muchos programas de sistemas en UNIX que trabajan con expresiones regulares. En la clase de caracteres `[+\-]`, se coloca una diagonal invertida delante del signo menos, porque solo, el signo menos se podría confundir con su uso para representar un intervalo, como en `[A-Z]`[5].

Hay otra forma de hacer que los caracteres tengan su significado natural, aunque sean metasímbolos de LEX: encerrarlos entre comillas. En el apartado de las reglas de traducción se muestra un ejemplo de esta convención donde los seis operadores relacionales están entre comillas[6].

---

[4] Es normal usar el programa `lex.yy.c` como subrutina de un analizador sintáctico generado por Yacc, un generador de analizadores sintácticos que se estudiará en el capítulo 4. En este caso, el analizador sintáctico proporcionaría la declaración de las constantes manifiestas al compilarlo con el programa `lex.yy.c`.

[5] En realidad, LEX maneja la clase de caracteres `[+-]` correctamente sin la diagonal invertida, porque el signo menos que aparece al final no puede representar un intervalo.

[6] Se hizo así porque < y > son metasímbolos de LEX; encierran entre comillas a los nombres de los "estados", permitiendo a LEX cambiar de estado al encontrar determinados componentes léxicos, como comentarios o cadenas entre comillas, que han de tener un tratamiento diferente al del texto habitual. No es necesario entrecomillar el signo *igual*, pero tampoco está prohibido.

Se van a considerar ahora las reglas de traducción de la sección que sigue a los primeros `%%`. La primera regla establece que si se encuentra un `eb`, es decir, cualquier secuencia máxima de espacios en blanco, caracteres TAB o de nueva línea, no se hace nada. Sobre todo, no se devuelve el control al analizador sintáctico. Recuérdese que la estructura del analizador léxico es tal que sigue intentando reconocer componentes léxicos hasta que la acción asociada a un encuentro exija un **return**.

La segunda regla establece que si aparecen las letras `if`, se devuelve el componente léxico `IF`, que es una constante manifiesta que representa un número entero que el analizador sintáctico interpreta como el componente léxico `if`. Las dos reglas siguientes tratan las palabras clave `then` y `else` de forma similar.

En la regla para **id**, aparecen dos proposiciones en la acción asociada. Primero, a la variable `yyval` se le asigna el valor devuelto por el procedimiento `instala\id`; la definición de este procedimiento está en la tercera sección. `yylval` es una variable cuya definición aparece en la salida de LEX `lex.yy.c`, que también está a disposición del analizador sintáctico. El propósito de `yylval` es retener el valor léxico devuelto, puesto que la segunda proposición de la acción, `return(ID)`, sólo puede devolver un código para la clase de componentes léxicos.

No se muestran los detalles del código para `instala_id()`. Sin embargo, se puede suponer que busca en la tabla de símbolos el lexema que concordó con el patrón **id**. LEX pone el lexema a disposición de las rutinas que aparecen en la tercera sección mediante las dos variables `yytexto` e `yylong`. La variable `yytexto` corresponde a la variable que se ha estado llamando *inicio_lexema*, es decir, un apuntador al primer carácter del lexema; `yylong` es un entero que indica la longitud del lexema. Por ejemplo, si `instala_id` no encuentra al identificador en la tabla de símbolos, le puede crear una nueva entrada. Los `yylong` caracteres de la entrada, comenzando en `yytexto`, se pueden copiar en una matriz de caracteres y delimitarse con un marcador de fin de cadena (FDC), como en la sección 2.7. La nueva entrada en la tabla de símbolos apuntaría al principio de esta copia.

La siguiente regla considera de igual manera los números. En las seis últimas reglas, `yylval` se utiliza para devolver un código para el operador relacional encontrado, mientras que el verdadero valor que se devuelve en cada caso es el código del componente léxico **oprel**.

Supóngase que al analizador léxico que resulta del programa de la figura 3.1 se le da una entrada formada por dos caracteres TAB, las letras `if` y un espacio en blanco. Los dos caracteres TAB son el prefijo inicial más largo de la entrada emparejado por un patrón, o sea, el patrón `eb`. La acción que corresponde a `eb` es no hacer nada, de modo que el analizador léxico pasa el apuntador de inicio del lexema, `yytexto`, a la `i` y empieza a buscar otro componente léxico.

El siguiente lexema a emparejar es `if`. Adviértase que los patrones, `if` e {`id`}, concuerdan con este lexema, y ningún patrón concuerda con una cadena mayor. Como el patrón de la palabra clave `if` precede al patrón de los identificadores de la lista de la figura 3.18, el conflicto se resuelve a favor de la palabra clave. En general, esta estrategia de resolución de ambigüedades facilita el reservar palabras clave relacionándolas por delante del patrón de los identificadores.

Como otro ejemplo, supóngase que <= son los dos primeros caracteres que se leen. Aunque el patrón < concuerda con el primer carácter, no es el patrón más largo

que concuerda con un prefijo de la entrada. De este modo, la estrategia de LEX de seleccionar el mayor prefijo que concuerde con un patrón facilita la resolución del conflicto entre < y <= según se esperaba (eligiendo <= como el siguiente componente léxico). □

El operador de preanálisis

Como se ha indicado en la sección 3.1, los analizadores léxicos para algunas construcciones de los lenguajes de programación necesitan hacer un preanálisis del fin de un lexema antes de poder determinar con certeza un componente léxico. Recuérdese el ejemplo de FORTRAN para el par de proposiciones

```
DO 5 I = 1.25
DO 5 I = 1,25
```

En FORTRAN, los espacios en blanco no son significativos fuera de los comentarios y de las cadenas Hollerith, de modo que supóngase que todos los espacios en blanco eliminables se suprimen antes de comenzar el análisis léxico. En tal caso, las proposiciones anteriores aparecerían al analizador léxico como

```
DO5I=1.25
DO5I=1,25
```

En la primera proposición, no se puede saber que la cadena DO es parte del identificador DO5I, hasta haber visto el punto decimal. En la segunda proposición, DO por sí sola es una palabra clave.

En LEX, un patrón se puede escribir de la forma r_1/r_2, donde r_1 y r_2 son expresiones regulares, y quiere decir que se empareje una cadena en r_1, pero sólo si va seguida de una cadena en r_2. La expresión regular r_2 después del operador de preanálisis indica el contexto adecuado para una concordancia; sólo se utiliza para restringir una concordancia, no para ser parte de ella. Por ejemplo, una especificación en LEX que reconoce la palabra clave DO en el contexto anterior es

```
DO/({letra} ¦ {dígito})* = ({letra} ¦ {dígito})*,
```

Con esta especificación, el analizador léxico buscará en su *buffer* de entrada una secuencia de letras y dígitos seguida de un signo igual, seguido de letras y dígitos, seguidos de una coma, para tener la seguridad de que no tenía una proposición de asignación. Sólo entonces los caracteres D y O, que preceden al operador de preanálisis / serán parte del lexema emparejado. Después de una concordancia con éxito, `yytexto` apunta a D y a `yylong` = 2. Obsérvese que este sencillo patrón de preanálisis permite el reconocimiento de DO cuando vaya seguido de basura, como en Z4=6Q, pero nunca reconocerá a DO como parte de un identificador.

Ejemplo 3.12. Se puede utilizar el operador de preanálisis para tratar otro problema difícil de análisis léxico en FORTRAN: distinguir las palabras clave de los identificadores. Por ejemplo, la entrada

```
IF(I, J) = 3
```

es una proposición de asignación perfectamente válida en FORTRAN, no una pro-

posición **if** lógica. Una forma de especificar la palabra clave `IF` utilizando LEX es definir sus posibles contextos adecuados utilizando el operador de preanálisis. La forma simple de la proposición **if** lógica es

`IF (` *condición* `)` *proposición*

FORTRAN 77 introdujo otra forma de la proposición **if** lógica:

```
IF ( condición ) THEN
      bloque_then
ELSE
      bloque_else
END IF
```

Se observa que toda proposición no etiquetada de FORTRAN comienza con una letra y que todo paréntesis derecho utilizado para subindizar o agrupar operandos debe ir seguido de un símbolo de operador, como =, + o coma, otro paréntesis derecho o el fin de la proposición. Este paréntesis derecho no puede ir seguido de una letra. En esta situación, para confirmar que `IF` es una palabra clave en vez de un nombre de matriz, se examina por adelantado, buscando un paréntesis derecho seguido de una letra antes de que aparezca un carácter de nueva línea (se supone que las tarjetas de continuación "cancelan" el carácter de nueva línea anterior). Este patrón para la palabra clave `IF` se puede escribir como

```
IF / \( .* \) {letra}
```

El punto representa "cualquier carácter salvo nueva línea" y las diagonales invertidas delante de los paréntesis exigen que LEX las considere literalmente, y no como metasímbolos para agrupar expresiones regulares (véase Ejercicio 3.10). □

Otra forma de abordar el problema planteado por las proposiciones **if** de FORTRAN es, después de haber visto `IF(`, determinar si `IF` se declaró como matriz. Sólo en tal caso se busca todo el patrón anterior. Estas pruebas hacen más difícil la aplicación automática de un analizador léxico a partir de una especificación en LEX, y a largo plazo pueden incluso hacer perder tiempo, dadas las frecuentes verificaciones que debe hacer el programa que simula un diagrama de transiciones para determinar si se deben efectuar tales pruebas. Obsérvese que dividir FORTRAN en componentes léxicos es una tarea tan irregular que muchas veces es más fácil escribir un analizador léxico especial para FORTRAN en un lenguaje de programación convencional, que usar un generador automático de analizadores léxicos.

3.6 AUTOMATAS FINITOS

Un *reconocedor* de un lenguaje es un programa que toma como entrada una cadena x y responde "sí" si x es una frase del programa, y "no", si no lo es. Se compila una expresión regular en un reconocedor construyendo un diagrama de transiciones generalizado llamado autómata finito. Un autómata finito puede ser determinista o no determinista, donde "no determinista" significa que en un estado se puede dar el caso de tener más de una transición para el mismo símbolo de entrada.

Tanto los autómatas finitos deterministas como los no deterministas pueden reconocer con precisión a los conjuntos regulares. Por tanto, ambos pueden reconocer con precisión lo que denotan las expresiones regulares. Sin embargo, hay un conflicto entre espacio y tiempo; mientras que un autómata finito determinista puede dar reconocedores más rápidos que uno no determinista, un autómata finito determinista puede ser mucho mayor que un autómata no determinista equivalente. En la siguiente sección, se introducen métodos para convertir expresiones regulares en ambas clases de autómatas finitos. La conversión en un autómata no determinista es más directa, por lo que primero se estudia este caso.

Los ejemplos de esta sección y de la siguiente se refieren principalmente al lenguaje representado por la expresión regular $(a \mid b)^*abb$, que está formada por el conjunto de todas las cadenas de caracteres a y b que terminen en abb. En la práctica existen lenguajes similares. Por ejemplo, una expresión regular para los nombres de todos los archivos que terminen en `.o` es de la forma `(.| o | c)* .o`, donde c representa cualquier carácter salvo un punto o una o. Otro ejemplo: después de la secuencia de apertura `/*`, los comentarios en C consisten en cualquier secuencia de caracteres que termine en `*/`, con el requisito adicional de que ningún prefijo propio termine en `*/`.

Autómatas finitos no deterministas

Un *autómata finito no determinista* (abreviado, AFN) es un modelo matemático formado por:

1. un conjunto de *estados* S
2. un conjunto de símbolos de entrada $\sum$ (el *alfabeto de símbolos de entrada*)
3. una función de transición *mueve* que transforma pares estado-símbolo en conjuntos de estados
4. un estado s_0 que se considera el *estado de inicio* (o *inicial*)
5. un conjunto de estados F considerados como *estados de aceptación* (o *finales*)

Un AFN se puede representar diagramáticamente mediante un grafo dirigido etiquetado, llamado *grafo de transiciones*, en el que los nodos son los estados y las aristas etiquetadas representan la función de transición. Este grafo se parece a un diagrama de transiciones, pero el mismo carácter puede etiquetar dos o más transiciones fuera de un estado, y las aristas pueden etiquetarse con el símbolo especial ϵ y con símbolos de entrada.

En la figura 3.19 se muestra el grafo de transiciones de un AFN que reconoce al

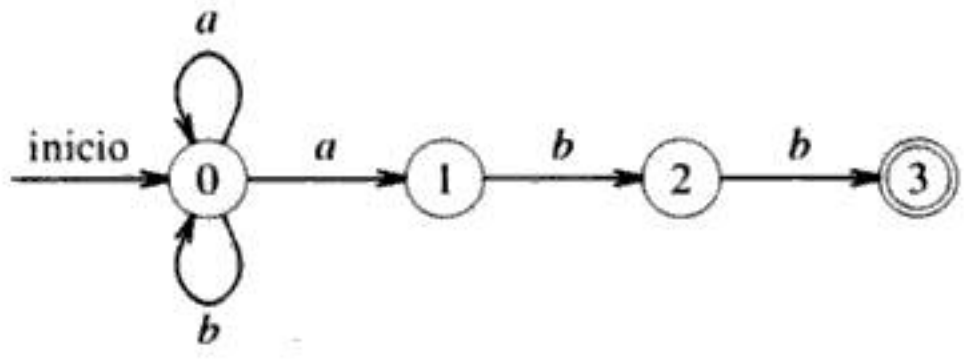


Fig. 3.19. Un autómata finito no determinista.

lenguaje $(a \mid b)^* abb$. El conjunto de estados del AFN es {0, 1, 2, 3} y el alfabeto de símbolos de entrada es {*a*, *b*}. El estado 0 de la figura 3.19 se considera el estado de inicio, y el estado de aceptación 3 está indicado mediante un círculo doble.

Cuando se describe un AFN, se utiliza la representación de grafo de transiciones. En un computador, puede aplicarse la función de transición de un AFN de varias formas, como se verá más adelante. La implantación más sencilla es una *tabla de transiciones* en donde hay una fila por cada estado y una columna por cada símbolo de entrada y ϵ, si es necesario. La entrada para la fila *i* y el símbolo *a* en la tabla es el conjunto de estados (o más probablemente en la práctica, un apuntador al conjunto de estados) que puede ser alcanzado por una transición del estado *i* con la entrada *a*. En la figura 3.20 se muestra la tabla de transiciones para el AFN de la figura 3.19.

Estado	Símbolo de Entrada	
	a	*b*
0	{0, 1}	{0}
1	–	{2}
2	–	{3}

Fig. 3.20. Tabla de transiciones para el autómata finito de la figura 3.19.

La representación en forma de tabla de transiciones tiene la ventaja de que proporciona rápido acceso a las transiciones de un determinado estado en un carácter dado; su inconveniente es que puede ocupar gran cantidad de espacio cuando el alfabeto de entrada es grande y la mayoría de las transiciones son hacia el conjunto vacío. Las representaciones de listas de adyacencias de la función de transición proporcionan implantaciones más compactas, pero el acceso a una transición dada es más lento. Debe quedar claro que se puede transformar fácilmente cualquiera de estas implantaciones de un autómata finito en otra.

Un AFN *acepta* una cadena de entrada *x* si, y sólo si, hay algún camino en el grafo de transiciones desde el estado de inicio a algún estado de aceptación, de forma que las etiquetas de las aristas a lo largo de dicho camino deletreen *x*. El AFN de la figura 3.19 acepta las cadenas de entrada *abb*, *aabb*, *babb*, *aaabb*, ... Por ejemplo, *aabb* es aceptada por el camino desde 0, siguiendo la arista etiquetada *a* de nuevo al estado 0 y después a los estados 1, 2 y 3 por las aristas etiquetadas *a*, *b* y *b*, respectivamente.

Se puede representar un camino mediante una secuencia de transiciones de estados llamada *movimientos*. El siguiente diagrama muestra los movimientos realizados para aceptar la cadena de entrada *aabb*:

$$0 \xrightarrow{a} 0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{a} 3$$

En general, puede haber más de una secuencia de movimientos que conduzca a un estado de aceptación. Obsérvese que pueden realizarse otras secuencias de movi-

mientos en la cadena de entrada *aabb*, pero ocurre que ninguna de dichas secuencias termina en un estado de aceptación. Por ejemplo, otra secuencia de movimientos en la entrada *aabb* sigue reinsertando el estado de no aceptación 0:

$$0 \xrightarrow{a} 0 \xrightarrow{a} 0 \xrightarrow{b} 0 \xrightarrow{a} 0$$

El *lenguaje definido por* un AFN es el conjunto de cadenas de entrada que acepta. No es difícil demostrar que el AFN de la figura 3.19 acepta $(a \mid b)^* abb$.

Ejemplo 3.13. En la figura 3.21, se ve cómo un AFN reconoce $aa^* \mid bb^*$. La cadena *aaa* es aceptada recorriendo los estados 0, 1, 2, 2 y 2. Las etiquetas de estas aristas son ϵ, *a*, *a* y *a*, cuya concatenación es *aaa*. Obsérvese que los símbolos ϵ "desaparecen" en una concatenación. □

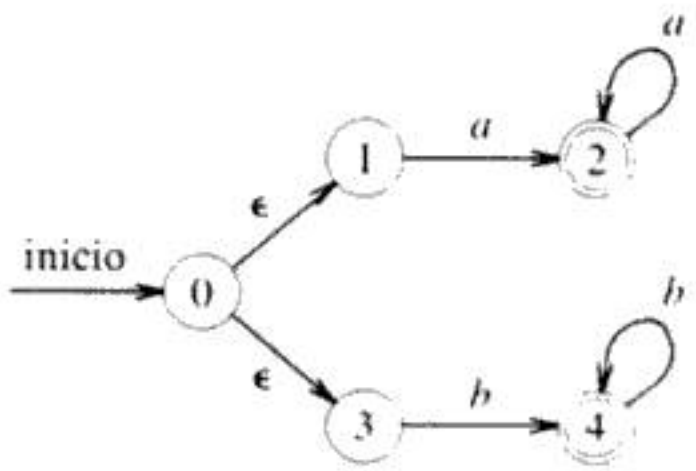


Fig. 3.21. Un AFN que acepta $aa^* \mid bb^*$.

Autómatas finitos deterministas

Un *autómata finito determinista* (abreviado AFD) es un caso especial de un autómata finito no determinista en el cual

1. ningún estado tiene una transición ϵ, es decir, una transición con la entrada ϵ, y
2. para cada estado *s* y cada símbolo de entrada *a*, hay a lo sumo una arista etiquetada *a* que sale de *s*.

Un autómata finito determinista tiene a lo sumo una transición desde cada estado con cualquier entrada. Si se está usando una tabla de transiciones para representar la función de transición de un AFD, entonces cada entrada en la tabla de transiciones es un solo estado. Como consecuencia, es muy fácil determinar si un autómata finito determinista acepta o no una cadena de entrada, puesto que hay a lo sumo un camino desde el estado de inicio etiquetado con esa cadena. El siguiente algoritmo muestra cómo simular el comportamiento de un AFD con una cadena de entrada.

Algoritmo 3.1. Simulación de un AFD.

Entrada. Una cadena de entrada *x* que termina con un carácter de fin de archivo **eof**. Un AFD *D* con un estado de inicio s_0 y un conjunto *F* de estados de aceptación.

Salida. La respuesta "sí", si *D* acepta *x*; "no", en caso contrario.

Método. Aplíquese el algoritmo de la figura 3.22 a la cadena de entrada x. La función *mueve* (s, c) da el estado al cual hay una transición desde el estado s en un carácter de entrada c. La función *sigtecar* devuelve el siguiente carácter de la cadena de entrada x. □

```
s := s0;
c := sigtecar;
while c ≠ eof do
    s := mueve(s, c);
    c := sigtecar
end;
if s está en F then
    return "sí"
else return "no";
```

Fig. 3.22. Simulación de un AFD.

Ejemplo 3.14. En la figura 3.23 se ve al grafo de transiciones de un autómata finito determinista aceptar el mismo lenguaje $(a \mid b)^*abb$ aceptado por el AFN de la figura 3.19. Con este AFD y la cadena de entrada *ababb*, el algoritmo 3.1 sigue la secuencia de estados 0, 1, 2, 1, 2, 3 y devuelve "sí". □

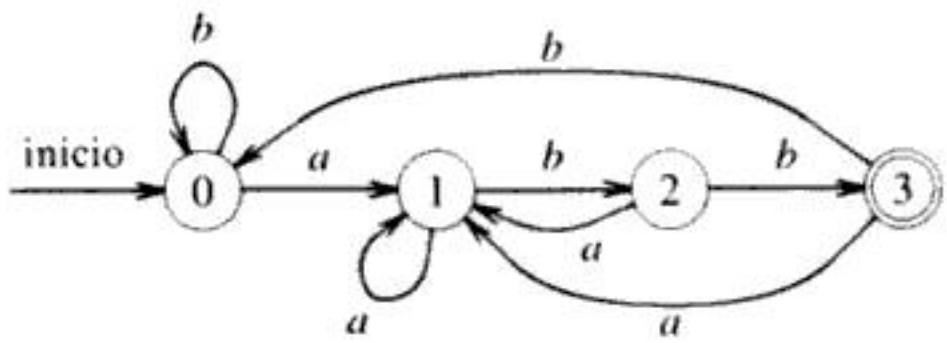


Fig. 3.23. AFD que acepta $(a \mid b)^*abb$.

Conversión de un AFN en un AFD

Obsérvese que el AFN de la figura 3.19 tiene dos transiciones desde el estado 0 con la entrada a; es decir, puede ir al estado 0 o al 1. Igualmente, el AFN de la figura 3.21 tiene dos transiciones en ϵ desde el estado 0. Aunque no se haya ilustrado con un ejemplo, una situación en la que se podría elegir una transición con ϵ o con un símbolo de entrada real también produce ambigüedad. Dichas situaciones, donde la función de transición tiene varios valores, hacen difícil simular un AFN con un programa de computador. La definición de aceptación simplemente establece que debe haber algún camino etiquetado por la cadena de entrada en cuestión que conduzca desde el estado de inicio a un estado de aceptación. Pero si hay muchos caminos que deletreen la misma cadena de entrada, quizás haya que considerarlos todos antes de encontrar uno que conduzca a la aceptación o descubrir que ningún camino conduce a un estado de aceptación.

Ahora se introduce un algoritmo para construir a partir de un AFN un AFD que reconozca el mismo lenguaje. Este algoritmo, a menudo llamado *construcción de subconjuntos*, es útil para simular un AFN por medio de un programa de computador. En el siguiente capítulo un algoritmo estrechamente relacionado desempeña un papel fundamental en la construcción de analizadores sintácticos LR.

En la tabla de transiciones de un AFN, cada entrada es un conjunto de estados; en la tabla de transiciones de un AFD, cada entrada es tan sólo un estado. La idea general tras la construcción AFN a AFD es que cada estado de AFD corresponde a un conjunto de estados del AFN. El AFD utiliza un estado para localizar todos los posibles estados en los que puede estar el AFN después de leer cada símbolo de la entrada. Es decir, después de leer la entrada $a_1a_2 \ldots a_n$, el AFD se encuentra en un estado que representa al subconjunto T de los estados del AFN alcanzables desde el estado de inicio del AFN a lo largo de algún camino etiquetado con $a_1a_2 \ldots a_n$. El número de estados de AFD puede ser exponencial en el número de estados del AFN, pero en la práctica este peor caso ocurre raramente.

Algoritmo 3.2. (*Construcción de subconjuntos.*) Construcción de un AFD a partir de un AFN.

Entrada. Un AFN N.

Salida. Un AFD D que acepta el mismo lenguaje.

Método. El algoritmo construye una tabla de transiciones *tranD* para D. Cada estado del AFD es un conjunto de estados del AFN y se construye *tranD* de modo que D simulará "en paralelo" todos los posibles movimientos que N puede realizar con una determinada cadena de entrada.

Se utilizan las operaciones de la figura 3.24 para localizar los conjuntos de los estados del AFN (s representa un estado del AFN, y T, un conjunto de estados del AFN).

OPERACIÓN	DESCRIPCIÓN
cerradura-ϵ (s)	Conjunto de estados del AFN alcanzables desde el estado s del AFN con transiciones ϵ solamente.
cerradura-ϵ (T)	Conjunto de estados del AFN alcanzables desde algún estado s en T con transiciones ϵ solamente.
mueve (T, a)	Conjunto de estados del AFN hacia los cuales hay una transición con el símbolo de entrada a desde algún estado s en T del AFN.

Fig. 3.24. Operaciones sobre los estados de un AFN.

Antes de detectar el primer símbolo de entrada, N se puede encontrar en cualquiera de los estados del conjunto *cerradura*-ϵ (s_0), donde s_0 es el estado de inicio de N. Supóngase que exactamente los estados del conjunto T son alcanzables desde s_0

con una secuencia dada de símbolos de entrada, y sea a el siguiente símbolo de entrada. Al ver a, N puede trasladarse a culquiera de los estados del conjunto *mueve* (T, a). Cuando se permiten transiciones-ϵ, N puede encontrarse en cualquiera de los estados de *cerradura*-ϵ (T, a)), después de ver la a.

```
al inicio, cerradura-ε (s0) es el único estado dentro de estadosD y no está marcado;
while haya un estado no marcado T en estadosD do begin
    marcar T;
    for cada símbolo de entrada a do begin
        U := cerradura-ε (mueve (T, a));
        if U no está en estadosD then
            añadir U como estado no marcado a estadosD;
        tranD [T, a] := U
    end
end
```

Fig. 3.25. La construcción de subconjuntos.

Se construyen *estadosD*, el conjunto de estados de D, y *tranD*, la tabla de transiciones de D, de la siguiente forma. Cada estado de D corresponde a un conjunto de estados de AFN en los que podría estar N después de leer alguna secuencia de símbolos de entrada, incluidas todas las posibles transiciones-ϵ anteriores o posteriores a la lectura de símbolos. El estado de inicio de D es *cerradura*-ϵ (s_0). Se añaden los estados y las transiciones a D utilizando el algoritmo de la figura 3.25. Un estado de D es un estado de aceptación si es un conjunto de estados de AFN que contenga al menos un estado de aceptación de N.

```
meter todos los estados de T en pila;
inicializar cerradura-ε (T) a T;
while pila no esté vacía do begin
    sacar t, el elemento del tope, de pila;
    for cada estado u con una arista desde t a u etiquetada con ε do
    if u no está en cerradura-ε (T) do begin
        añadir u a cerradura-ε (T);
        meter u en pila
    end
end
```

Fig. 3.26. Cálculo de *cerradura*-ϵ .

El cálculo de *cerradura*-ϵ (T) es un proceso típico de búsqueda en un grafo de nodos alcanzables desde un conjunto dado de nodos. En este caso, los estados de T son el conjunto dado de nodos, y el grafo está compuesto solamente por las aristas del AFN etiquetadas por ϵ. Un algoritmo sencillo para calcular *cerradura*-ϵ (T) utiliza una estructura de datos tipo pila para guardar estados en cuyas aristas no se hayan buscado transiciones etiquetadas con ϵ. Este procedimiento se muestra en la figura 3.26. □

Ejemplo 3.15. La figura 3.27 muestra otro AFN N aceptando el lenguaje $(a \mid b)^*abb$. (El de la siguiente sección es el que se construirá mecánicamente a partir de la expresión regular.) Se aplica el algoritmo 3.2 a N. El estado de inicio del AFD equivalente es *cerradura*-ϵ (0), que es $A = \{0, 1, 2, 4, 7\}$, puesto que estos son exactamente los estados alcanzables desde el estado 0 por un camino en el que todas las aristas están etiquetadas por ϵ. Obsérvese que un camino puede no tener aristas, de modo que 0 es alcanzado desde sí mismo por dicho camino.

Aquí, el alfabeto de símbolos de entrada es $\{a, b\}$. El algoritmo de la figura 3.25 indica que hay que marcar A y después calcular

cerradura-ϵ (*mueve* (A , a)).

Primero se calcula *mueve*(A, a), el conjunto de estados de N que tiene transiciones en a desde miembros de A. Entre los estados 0, 1, 2, 4 y 7 sólo 2 y 7 tienen dichas transiciones, a 3 y a 8, de modo que

cerradura-ϵ (*mueve* ($\{0, 1, 2, 4, 7\}$, a)) = *cerradura*-ϵ ($\{3,8\}$) = $\{1, 2, 3, 4, 6, 7, 8\}$

Este conjunto se denominará B. Así, *tranD*[A, a] = B.

Entre los estados de A, sólo 4 tienen una transición en b a 5, de modo que el AFD tiene una transición en b desde A a

C = *cerradura*-ϵ ($\{5\}$) = $\{1, 2, 4, 5, 6, 7\}$.

Por tanto, *tranD*[A, b] = C.

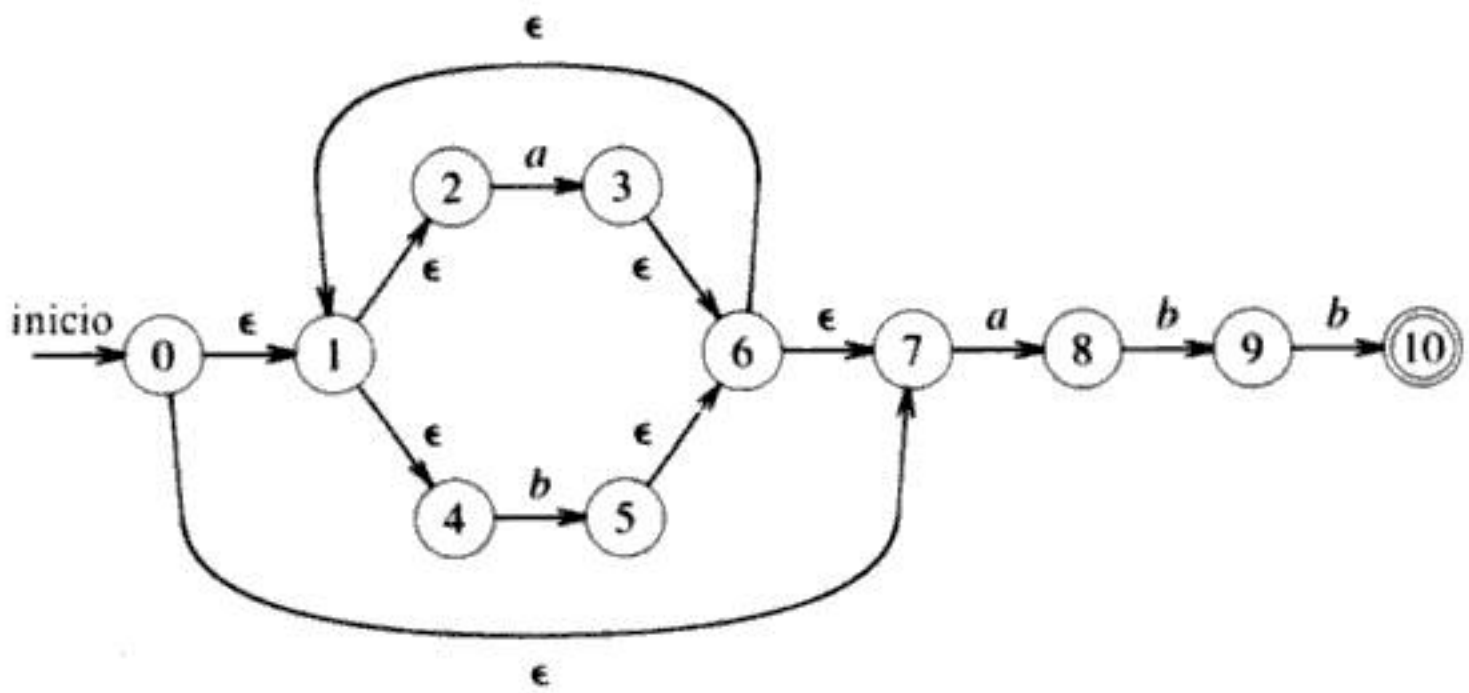


Fig. 3.27. AFN N que acepta $(a \mid b)^*abb$.

Si se continúa este proceso con los conjuntos B y C, ahora sin marcar, finalmente se llegará al punto en que todos los conjuntos que son estados del AFD estén marcados. Esto es cierto porque "sólo" hay 2^{11} subconjuntos distintos de un conjunto de 11 estados, y un conjunto, una vez marcado, queda marcado para siempre. Los cinco conjuntos de estados diferentes realmente construidos son:

$A = \{0, 1, 2, 4, 7\}$
$B = \{1, 2, 3, 4, 6, 7, 8\}$
$C = \{1, 2, 4, 5, 6, 7\}$
$D = \{1, 2, 4, 5, 6, 7, 9\}$
$E = \{1, 2, 4, 5, 6, 7, 10\}$

El estado A es el estado de inicio, y el estado E es el único estado de aceptación. La tabla de transiciones completa *tranD* se muestra en la figura 3.28.

Estado	Símbolo de Entrada	
	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

Fig. 3.28. Tabla de transiciones *tranD* para el AFD.

Además, en la figura 3.29 se muestra un grafo de transiciones para el AFD resultante. Debe tenerse en cuenta que el AFD de la figura 3.23 también acepta $(a \mid b)^* abb$ y tiene un estado menos. En la sección 3.9 se estudia la cuestión de minimización del número de estados de un AFD. □

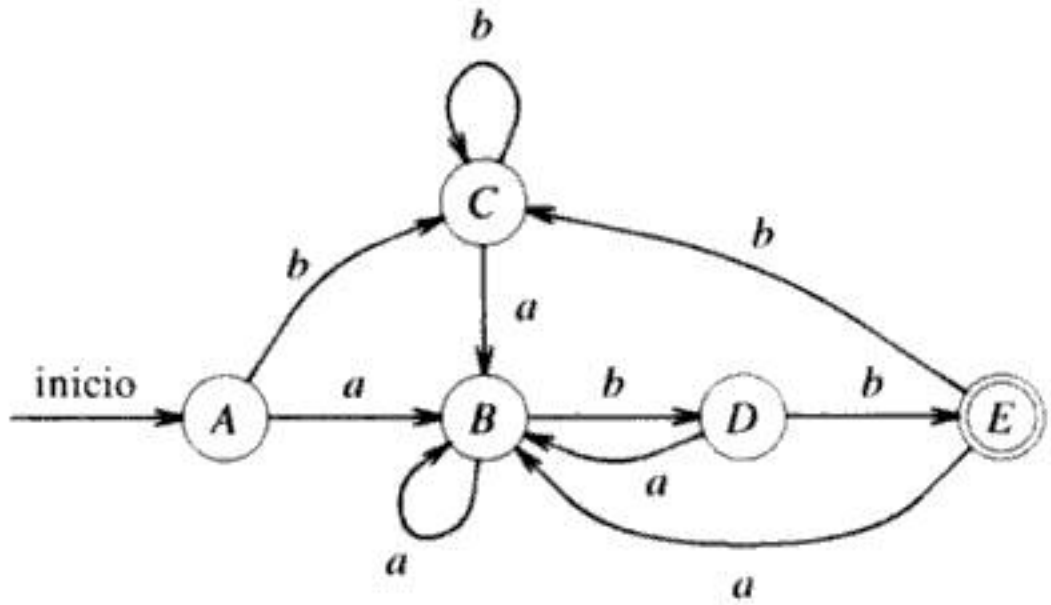


Fig. 3.29. Resultado de aplicar la construcción de subconjuntos a la figura 3.27.

3.7 PASO DE UNA EXPRESION REGULAR A UN AFN

Existen muchas estrategias para construir un reconocedor a partir de una expresión regular, cada una con sus puntos fuertes y sus puntos débiles. Una estrategia utilizada en varios programas de edición de textos consiste en construir un AFN a partir de una expresión regular y después simular el comportamiento del AFN con una cadena de entrada utilizando los algoritmos 3.3 y 3.4 de esta sección. Si la velocidad de ejecución es fundamental, se puede convertir el AFN en un AFD utilizando la construcción de subconjuntos de la sección anterior. En la sección 3.9 se presenta una aplicación opcional de un AFD a partir de una expresión regular en la que no

se construye de manera explícita un AFN intermedio. Esta sección concluye con un análisis del aspecto tiempo-espacio en la implantación de reconocedores basados en AFN y AFD.

Construcción de un AFN a partir de una expresión regular

A continuación se proporciona un algoritmo para construir un AFN a partir de una expresión regular. Hay muchas variantes de este algoritmo, pero aquí se introduce una versión sencilla fácil de aplicar. El algoritmo está dirigido por la sintaxis en el sentido de que utiliza la estructura sintáctica de la expresión regular para guiar el proceso de construcción. Los casos del algoritmo siguen a los casos de la definición de una expresión regular. Primero, se muestra cómo construir autómatas para reconocer ϵ y cualquier símbolo del alfabeto. Después, se muestra cómo construir autómatas para expresiones que contengan una alternación, concatenación o el operador de la cerradura de Kleene. Por ejemplo, para la expresión $r \mid s$, se construye un AFN de manera inductiva a partir de los AFN para r y s.

A medida que avanza la construcción, cada paso introduce a lo sumo dos nuevos estados, de modo que el AFN resultante construido para una expresión regular tendrá a lo sumo el doble de estados que símbolos y operadores hay en la expresión regular.

Algoritmo 3.3. (*Construcción de Thompson.*) Construcción de un AFN a partir de una expresión regular.

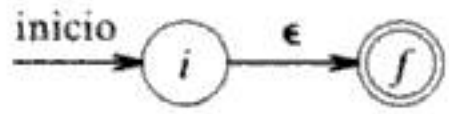
Entrada. Una expresión regular r en un alfabeto $\sum$.

Salida. Un AFN N que acepte $L(r)$.

Método. Primero se hace el análisis sintáctico de r en sus subexpresiones constituyentes. Después, por las reglas 1 y 2 que se verán más adelante, se construyen los AFN para cada uno de los símbolos básicos de r (aquellos que sean ϵ o un símbolo del alfabeto). Los símbolos básicos corresponden a las partes 1 y 2 de la definición de una expresión regular. Es importante comprender que si un símbolo a aparece varias veces en r, se construye un AFN independiente en cada caso.

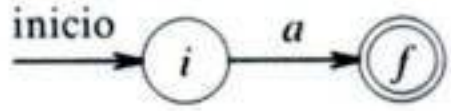
Después, guiándose por la estructura sintáctica de la expresión regular r, se combinan inductivamente esos AFN utilizando la regla 3 que se verá más adelante hasta obtener el AFN para la expresión completa. Todos los AFN intermedios producidos durante el curso de la construcción corresponden a una subexpresión de r y tienen varias propiedades importantes: tienen exactamente un estado final, ninguna arista entra en el estado de inicio, y ninguna arista sale del estado final.

1. Para ϵ, construir el AFN



Aquí, i es un nuevo estado de inicio y f es un nuevo estado de aceptación. Ciertamente este AFN reconoce $\{\epsilon\}$.

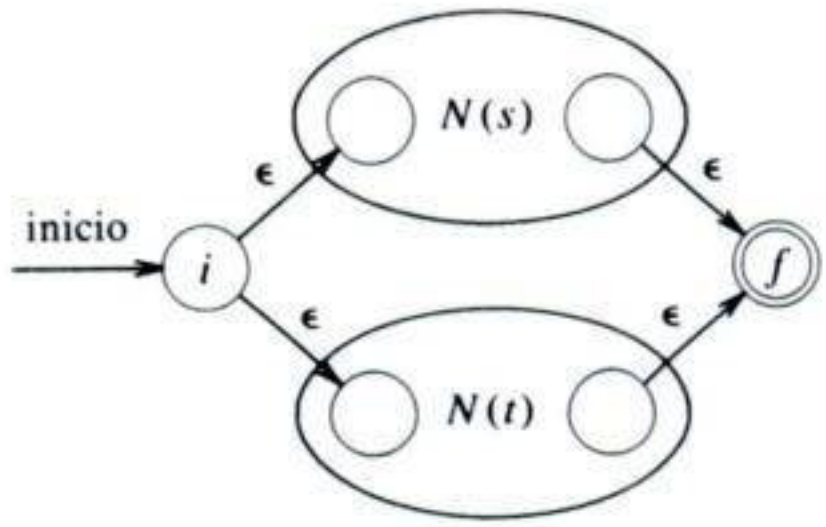
2. Para a de $\sum$, construir el AFN



De nuevo, i es un nuevo estado de inicio, y f, un nuevo estado de aceptación. Esta máquina reconoce $\{a\}$.

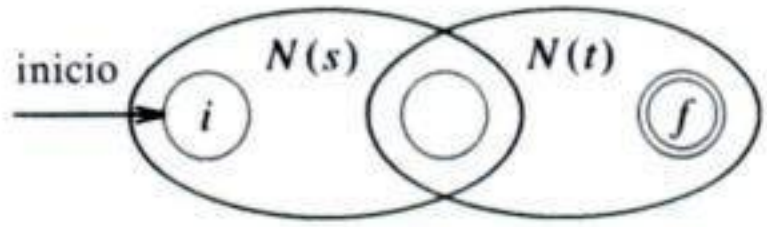
3. Supóngase que $N(s)$ y $N(t)$ son AFN para las expresiones regulares s y t.

   a) Para la expresión regular $s \mid t$, constrúyase el siguiente AFN compuesto $N(s \mid t)$:



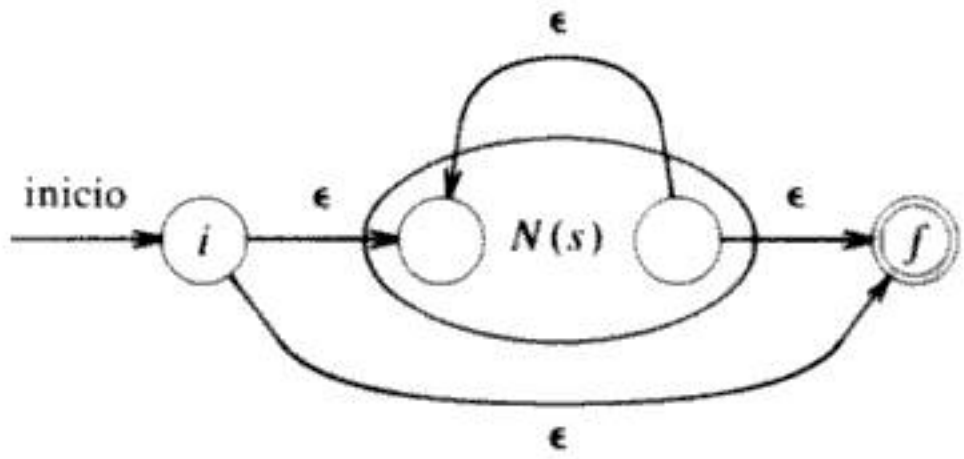
Aquí, i es un nuevo estado de inicio, y f, un nuevo estado de aceptación. Hay una transición con ϵ desde i a los estados de inicio de $N(s)$ y $N(t)$. Hay una transición con ϵ desde los estados de aceptación de $N(s)$ y $N(t)$ al nuevo estado de aceptación f. Los estados de inicio y aceptación de $N(s)$ y $N(t)$ no son estados de inicio o aceptación de $N(s \mid t)$. Obsérvese que todos los caminos desde i a f deben pasar exclusivamente por $N(s)$ o por $N(t)$. Así, se ve que el AFN compuesto reconoce $L(s) \cup L(t)$.

   b) Para la expresión regular st, constrúyase el AFN compuesto $N(st)$:



El estado de inicio de $N(s)$ se convierte en el estado de inicio del AFN compuesto y el estado de aceptación de $N(t)$ se convierte en el estado de aceptación del AFN compuesto. El estado de aceptación de $N(s)$ se fusiona con el estado de inicio de $N(t)$; es decir, todas las transiciones desde el estado de inicio de $N(t)$ se convierten en transiciones desde el estado de aceptación de $N(s)$. El nuevo estado fusionado pierde su condición de estado de inicio o de aceptación en el AFN compuesto. Un camino desde i a f debe pasar primero a través de $N(s)$, y después, por $N(t)$, de modo que la etiqueta de ese camino será una cadena de $L(s)L(t)$. Puesto que ninguna arista entra en el estado de inicio de $N(t)$ ni sale del estado de aceptación de $N(s)$, no puede haber ningún camino desde i a f que vuelva de $N(t)$ a $N(s)$. Por tanto, el AFN compuesto reconoce $L(s)L(t)$.

c) Para la expresión regular s^*, constrúyase el AFN compuesto $N(s^*)$:



Aquí, i es un nuevo estado de inicio y f un nuevo estado de aceptación. En el AFN compuesto, se puede ir desde i a f directamente, a lo largo de una arista etiquetada por ϵ, que representa el hecho de que ϵ está en $L(s))^*$, o se puede ir desde i a f pasando por $N(s)$ una o más veces. Ciertamente, el AFN compuesto reconoce $(L(s))^*$.

d) Para la expresión regular entre paréntesis (s), utilícese $N(s)$ como AFN.

Cada vez que se construye un nuevo estado, se le da un nombre distinto. De esta forma, no puede haber dos estados de un AFN componentes con el mismo nombre. Aunque aparezcan los mismos símbolos varias veces en r, se crea para cada ejemplo de ese símbolo un AFN aparte con sus propios estados. □

Se puede comprobar que cada paso de la construcción del algoritmo 3.3 produce un AFN que reconoce el lenguaje correcto. Además, la construcción produce un AFN $N(r)$ con las siguientes propiedades.

1. $N(r)$ tiene a lo sumo el doble de estados que de símbolos y operadores en r. Esto se debe al hecho de que en cada paso de la construcción se crean a lo sumo dos nuevos estados.

2. $N(r)$ tiene exactamente un estado de inicio y otro de aceptación. El estado de aceptación no tiene transiciones salientes. Esta propiedad se cumple igualmente para todos los autómatas constituyentes.

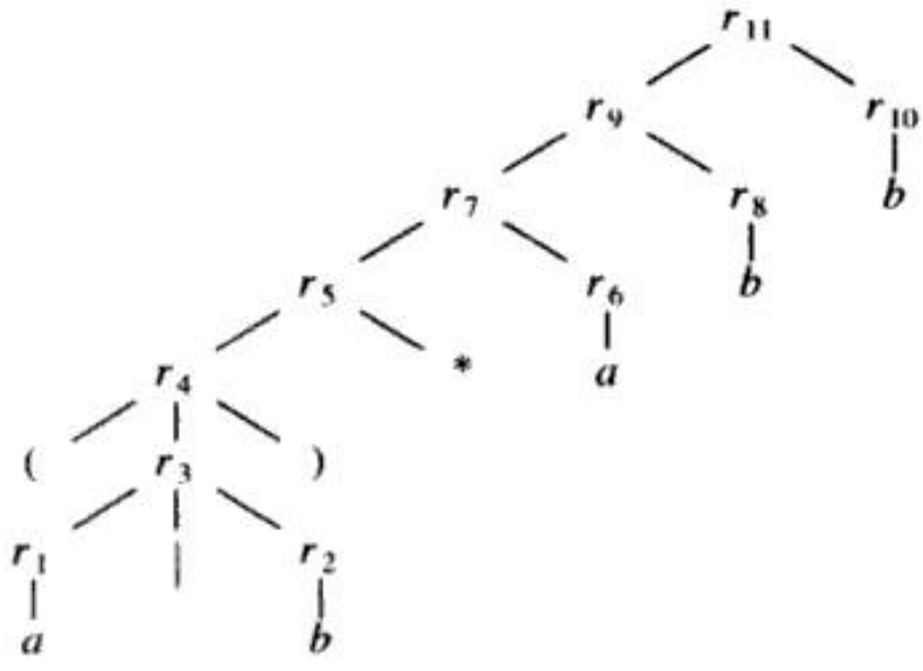
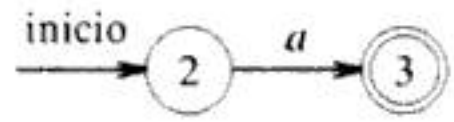


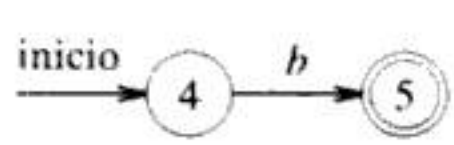
Fig. 3.30. Descomposición de $(a \mid b)^*abb$.

3. Cada estado de $N(r)$ tiene una transición saliente con un símbolo en $\sum$o a lo sumo dos transiciones ϵ salientes.

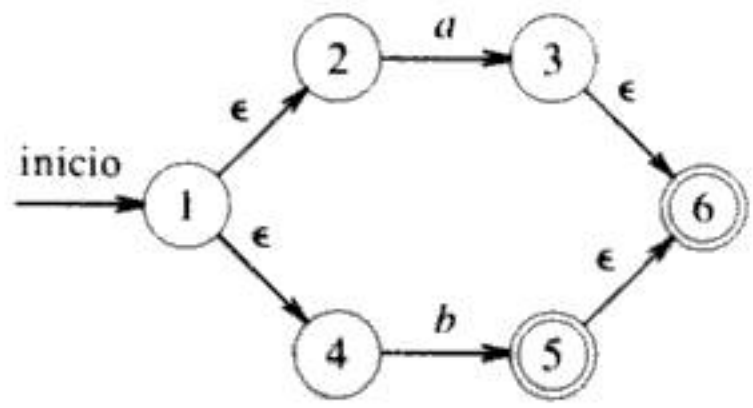
Ejemplo 3.16. Utilícese el algoritmo 3.3 para construir $N(r)$ para la expresión regular $r = (a \mid b)^*abb$. En la figura 3.30 se muestra un árbol de análisis sintáctico para r análogo a los árboles de análisis sintáctico construidos para las expresiones aritméticas de la sección 2.2. Para el constituyente r_1, la primera a, se construye el AFN



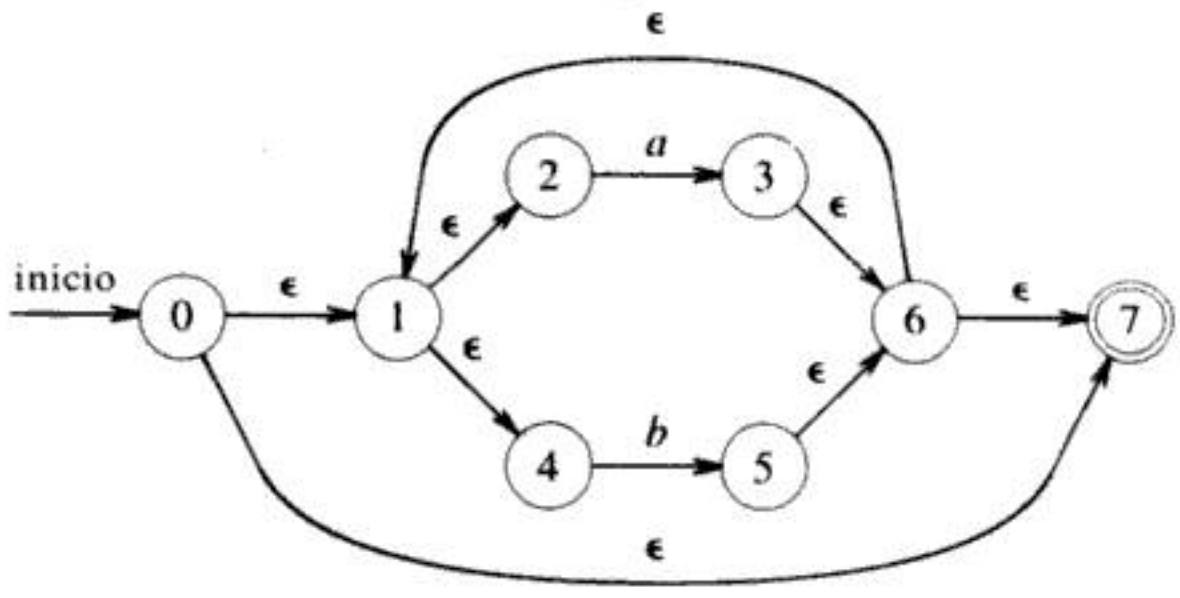
Para r_2, se construye



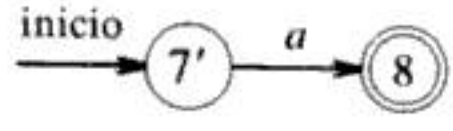
Ahora se pueden combinar $N(r_1)$ y $N(r_2)$ utilizando la regla de la unión para obtener el AFN para $r_3 = r_1 \mid r_2$



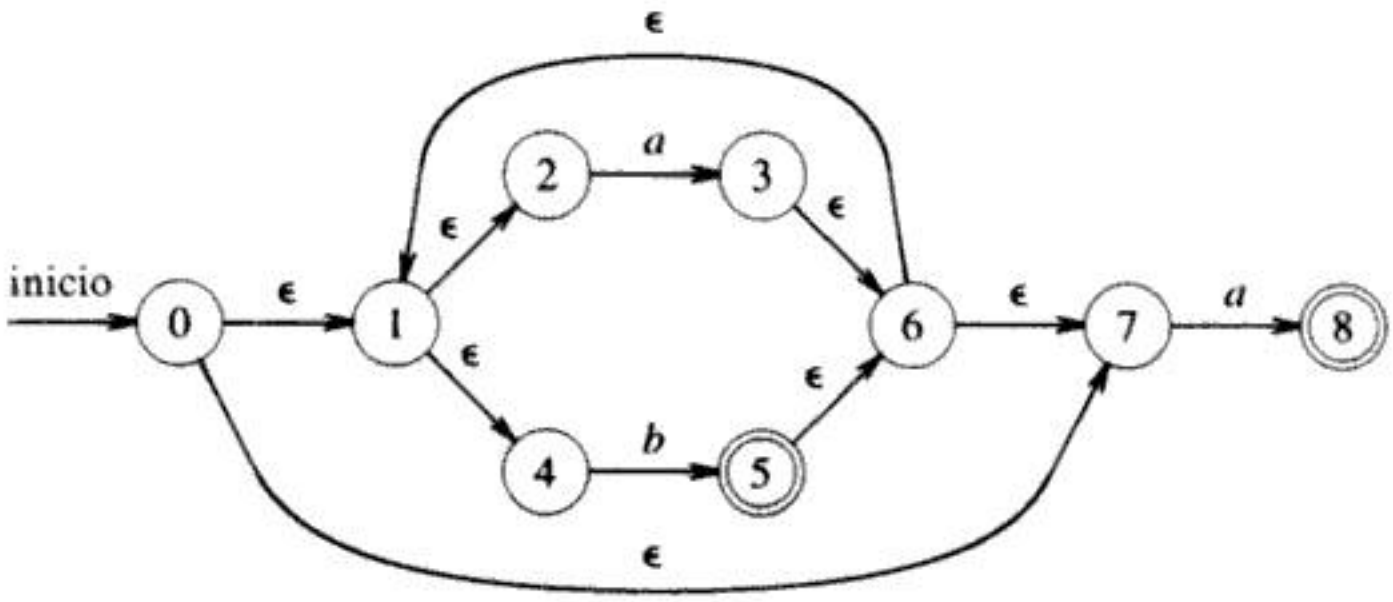
El AFN para (r_3) es el mismo que para r_3. Entonces, el AFN para $(r_3)^*$ es



El AFN para $r_6 = a$ es



Para obtener el autómata para $r_5\ r_6$, se fusionan los estados 7 y 7', llamando al estado 7 resultante, para obtener



Si se continúa así, se obtiene el AFN para $r_{11} = (a \mid b)^* abb$ mostrado por primera vez en la figura 3.27. □

Simulación de un AFN por medio de dos pilas

A continuación se introduce un algoritmo que, desde un AFN N construido por el algoritmo 3.3 y una cadena de entrada x, determina si N acepta o no a x. El algoritmo funciona leyendo la entrada un carácter a la vez y calculando todo el conjunto de estados en los que podría estar N después de haber leído todos los prefijos de la entrada. El algoritmo aprovecha las propiedades especiales del AFN producido por el algoritmo 3.3 para calcular eficazmente cada conjunto de estados no deterministas. Puede implantarse para que se ejecute en un tiempo proporcional a $|N| \times |x|$, donde $|N|$ es el número de estados de N y $|x|$ es la longitud de x.

Algoritmo 3.4. Simulación de un AFN.

Entrada. Un AFN N construido por el algoritmo 3.3 y una cadena de entrada x. Se supone que x termina con un carácter de fin de archivo **eof**. N tiene estado inicial s_0 y un conjunto de estados de aceptación F.

Salida. La respuesta "sí", si N acepta x; "no", en caso contrario.

Método. Aplíquese el algoritmo esbozado en la figura 3.31 a la cadena de entrada x. El algoritmo realiza de hecho la construcción de subconjuntos en el momento de la ejecución. Calcula una transición desde el conjunto en curso de estados S al siguiente conjunto de estados en dos etapas. Primero, determina *mueve* (S, a), todos los estados que se pueden alcanzar desde un estado en S mediante una transición con a, el carácter de entrada en curso. Después, calcula la *cerradura*-ϵ de *mueve* (s, a), es decir, todos los estados que se pueden alcanzar desde *mueve* (s, a) mediante cero o más transiciones ϵ. El algoritmo utiliza la función *sigtecar* para leer los caracteres de x, uno a la vez. Cuando han aparecido todos los caracteres de x, el algoritmo devuelve "sí", si un estado de aceptación está en el conjunto S de estados en curso; de lo contrario, devuelve "no". □

```
S := cerradura-ϵ ({s0});
a := sigtecar;
while a ≠ eof do begin
    S := cerradura-ϵ (mueve (S, a));
    a := sigtecar
end
if S ∩ F ≠ ∅ then
    return "sí";
else return "no";
```

Fig. 3.31. Simulación del AFN del algoritmo 3.3.

El algoritmo 3.4 puede ser implantado de manera eficiente utilizando dos pilas y un vector de bits indexado por estados del AFN. Se utiliza una pila para tener localizado el conjunto en curso de estados no deterministas, y la otra, para calcular el siguiente conjunto de estados no deterministas. Se puede usar el algoritmo de la figura 3.26 para calcular la *cerradura*-ϵ, y el vector de bits, para determinar en un tiempo constante si un estado no determinista ya está en una pila, para no ser añadido dos veces. Una vez calculado el siguiente estado en la segunda pila, se pueden intercambiar las funciones de las dos pilas. Puesto que cada estado no determinista tiene a lo sumo dos transiciones de salida, cada estado puede dar lugar a lo sumo a dos nuevos estados en una transición. Sea $|N|$ el número de estados de N. Como puede haber a lo sumo $|N|$ estados en una pila, se puede hacer el cálculo del siguiente conjunto de estados a partir del conjunto actual de estados en un tiempo proporcional a $|N|$. Así, el tiempo total necesario para simular el comportamiento de N con la entrada x es proporcional a $|N| \times |x|$.

Ejemplo 3.17. Sea N el AFN de la figura 3.27 y sea x la cadena compuesta únicamente por el carácter a. El estado de inicio es *cerradura*-ϵ $(\{0\}) = \{0, 1, 2, 4, 7\}$. Con el símbolo de entrada a hay una transición desde 2 a 3 y desde 7 a 8. Por tanto, T es $\{3, 8\}$. Tomando la *cerradura*-ϵ de T se obtiene el siguiente estado $\{1, 2, 3, 4, 6, 7, 8\}$. Como ninguno de estos estados no deterministas es de aceptación, el algoritmo devuelve "no".

Obsérvese que el algoritmo 3.4 hace la construcción de subconjuntos en el momento de la ejecución. Por ejemplo, compárense las transiciones anteriores con los estados del AFD de la figura 3.29 construido a partir del AFN de la figura 3.27. Los conjuntos de estado de inicio y estado siguiente en la entrada a corresponden a los estados A y B del AFD. □

El aspecto tiempo-espacio

Dada una expresión regular r y una cadena de entrada x, existen ahora dos métodos para determinar si x está en $L(r)$. Un enfoque consiste en utilizar el algoritmo 3.3 para construir un AFN N a partir de r. Se puede realizar dicha construcción en un tiempo de $O(|r|)$, donde $|r|$ es la longitud de r. N tiene a lo sumo el doble de estados que $|r|$, y a lo sumo dos transiciones desde cada estado, de modo que se puede

almacenar una tabla de transiciones para N en un espacio de $O(|r|)$. Entonces se puede utilizar el algoritmo 3.4 para determinar si N acepta a x en un tiempo $O(|r| \times |x|)$. Por tanto, utilizando este enfoque, se puede determinar si x está en $L(r)$ en un tiempo total proporcional a la longitud de r multiplicada por la longitud de x. Este enfoque ha sido utilizado en varios editores de texto para buscar patrones de expresiones regulares donde la cadena objetivo x no suele ser muy larga.

Un segundo enfoque es construir un AFD a partir de la expresión regular r aplicando la construcción de Thompson a r y después la construcción de subconjuntos, algoritmo 3.2, al AFN resultante. (En la Sec. 3.9 se da una implantación que evita construir el AFN intermedio de manera explícita.) Implantando la función de transición con una tabla de transiciones, se puede utilizar el algoritmo 3.1 para simular el AFD con la entrada x en un tiempo proporcional a la longitud de x, independientemente del número de estados del AFD. Este enfoque ha sido utilizado a menudo en programas de concordancia de patrones que buscan en archivos de texto patrones de expresiones regulares. Una vez construido el autómata finito, la búsqueda avanza muy rápido, de modo que este enfoque es el más adecuado cuando la cadena objetivo x es muy larga.

Hay, sin embargo, ciertas expresiones regulares cuyo menor AFD tiene un número de estados con un tamaño de expresión regular exponencial. Por ejemplo, la expresión regular $(a \mid b)^* a(a \mid b)(a \mid b) \ldots (a \mid b)$, donde hay $n-1$ expresiones $(a \mid b)$ al final, no tiene ningún AFD con menos de 2^n estados. Esta expresión regular representa todas las cadenas de caracteres a y b donde el n-ésimo carácter desde el extremo derecho es una a. No es difícil demostrar que cualquier AFD para esta expresión debe registrar los n últimos caracteres que vea en la entrada; de lo contrario, puede dar una respuesta errónea. Es evidente que se requieren al menos 2^n estados para localizar todas las posibles secuencias de n caracteres a y b. Afortunadamente, expresiones como ésta no se presentan con frecuencia en aplicaciones de análisis léxico, pero sí en otras aplicaciones.

Un tercer enfoque consiste en utilizar un AFD, pero hay que evitar construir toda la tabla de transiciones mediante la técnica llamada "evaluación diferida de transiciones". Aquí, se calculan las transiciones en el momento de la ejecución, pero no se determina una transición desde un estado determinado con un carácter dado hasta que es realmente necesario. Las transiciones calculadas se almacenan en un *cache*. Se consulta el *cache* cada vez que se va a hacer una transición. Si la transición no está ahí, se calcula y se almacena en el *cache*. Si el *cache* está lleno, se puede borrar alguna transición ya calculada para dejar sitio a la nueva transición.

En la figura 3.32 se resumen los requisitos de espacio y tiempo del peor caso para determinar si una cadena de entrada x está en el lenguaje representado por una ex-

AUTÓMATA	ESPACIO	TIEMPO
AFN	$O(\|r\|)$	$O(\|r\| \times \|x\|)$
AFD	$O(2^{\|r\|})$	$O(\|x\|)$

Fig. 3.32. Espacio y tiempo ocupados en reconocer expresiones regulares.

presión regular r utilizando reconocedores construidos a partir de autómatas finitos deterministas y no deterministas. La técnica de "evaluación diferida" combina los requisitos de espacio del método del AFN con el requerimiento de tiempo del enfoque del AFD. Su requerimiento de espacio es el tamaño de la expresión regular más el tamaño del *cache*; su tiempo de ejecución observado es casi tan rápido como el del reconocedor determinista. En algunas aplicaciones, la técnica de "evaluación diferida" es considerablemente más rápida que el enfoque del AFD, porque no se pierde tiempo en calcular transiciones de estados que nunca se utilizan.

3.8 DISEÑO DE UN GENERADOR DE ANALIZADORES LEXICOS

En esta sección se considera el diseño de una herramienta de *software* que construye automáticamente un analizador léxico a partir de un programa escrito en lenguaje LEX. Aunque se analizan varios métodos, y ninguno es idéntico al usado por el mandato LEX del sistema UNIX, dichos programas para construir analizadores léxicos se denominarán compiladores de LEX.

Supóngase que se tiene una especificación de un analizador léxico de la forma

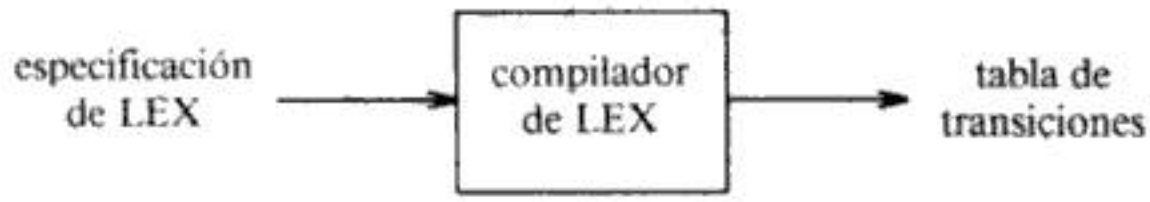
$$\begin{array}{ll} p_1 & \{ \textit{acción}_1 \} \\ p_2 & \{ \textit{acción}_2 \} \\ \cdots & \cdots \\ p_n & \{ \textit{acción}_n \} \end{array}$$

donde, como en la sección 3.5, cada patrón p_i es una expresión regular y cada acción $\textit{acción}_i$ es un fragmento de programa que debe ejecutarse siempre que se encuentre en la entrada un lexema que concuerde con p_i.

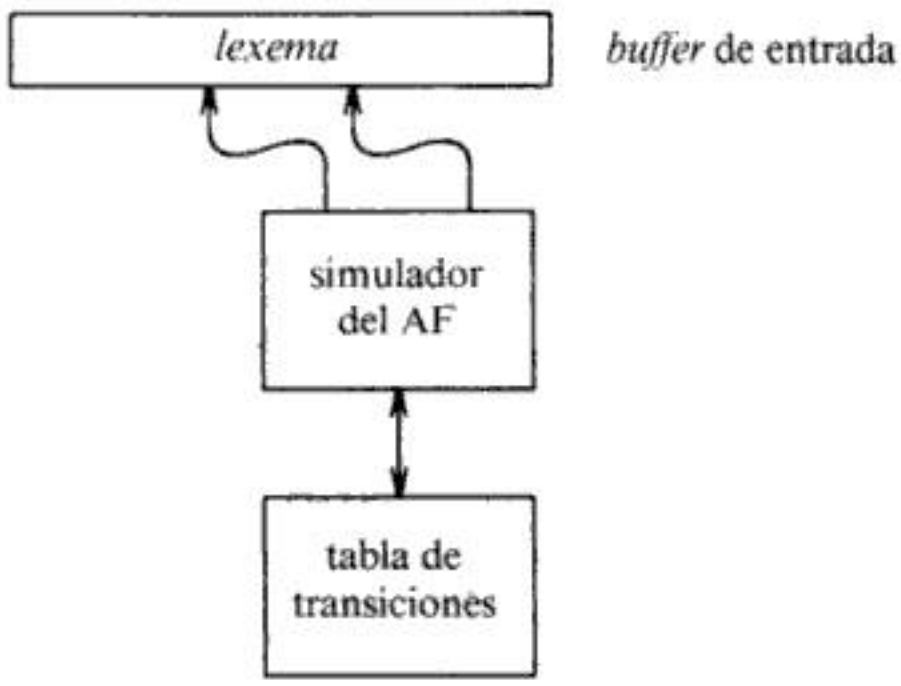
El problema es construir un reconocedor que busque lexemas en el *buffer* de la entrada. Si concuerda más de un patrón, el reconocedor elegirá el lexema más largo que haya concordado. Si hay dos o más patrones que concuerden con el lexema más largo, se elige el primer patrón que haya concordado de la lista.

Un autómata finito es un modelo natural sobre el que se construye un analizador léxico, y el construido por este compilador de LEX tiene la forma que se muestra en la figura 3.33(b). Hay un *buffer* de entrada con dos apuntadores hacia él, uno al inicio del lexema y el otro un apuntador delantero, como se vio en la sección 3.2. El compilador de LEX construye una tabla de transiciones para un autómata finito a partir de patrones de expresiones regulares en la especificación de LEX. El analizador léxico en sí consta de un simulador de autómata finito que utiliza esta tabla de transiciones para buscar los patrones de las expresiones regulares en el *buffer* de entrada.

El resto de esta sección demuestra que la implantación de un compilador de LEX se puede basar en autómatas deterministas o no deterministas. Al final de la última sección se vio que la tabla de transiciones de un AFN para un patrón de una expresión regular puede ser considerablemente menor que la de un AFD, pero el AFD tiene la gran ventaja de que puede reconocer patrones más rápidamente que el AFN.



(a) Compilador de LEX



(b) Analizador léxico esquemático

Fig. 3.33. Modelo de compilador de LEX.

Concordancia de patrones basada en los AFN

Un método es construir la tabla de transiciones de un autómata finito no determinista N para el patrón compuesto $p_1 \mid p_2 \mid \cdots \mid p_n$. Esto se puede hacer creando primero un AFN $N(p_i)$ para cada patrón p_i utilizando el algoritmo 3.3, añadiendo después un nuevo estado de inicio s_0, y por último enlazando s_0 al estado de inicio de cada $N(p_i)$ con una transición ϵ, tal como se muestra en la figura 3.34.

Para simular este AFN, se puede utilizar una modificación del algoritmo 3.4. La modificación garantiza que el AFN combinado reconoce el prefijo más largo de la

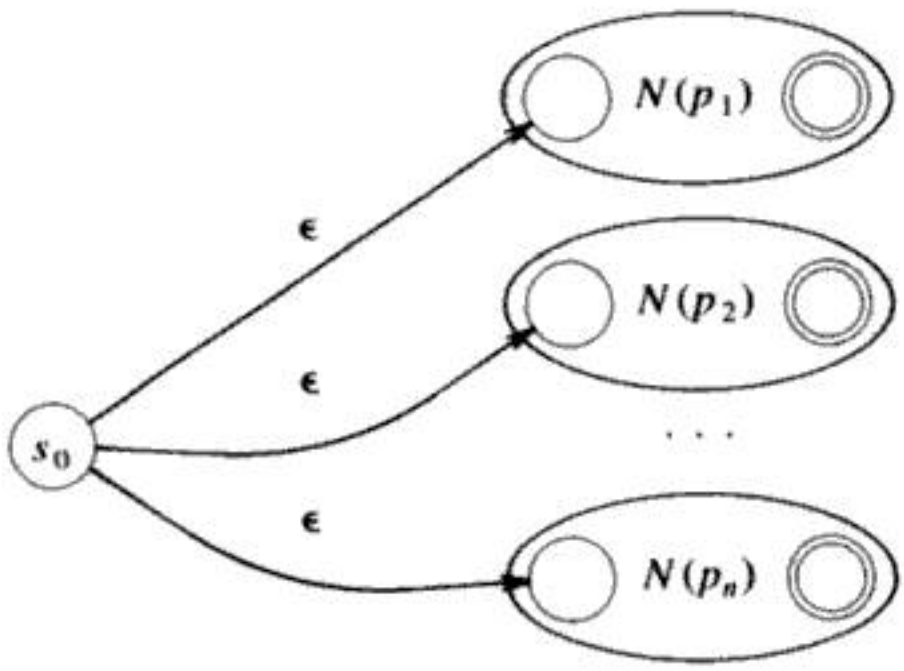


Fig. 3.34. AFN construido a partir de una especificación de LEX.

entrada que haya concordado con un patrón. En el AFN combinado, hay un estado de aceptación para cada patrón p_i. Cuando se simula el AFN utilizando el algoritmo 3.4, se construye la secuencia de conjuntos de estados donde puede estar el AFN combinado después de leer cada carácter de entrada. Incluso si se encuentra un conjunto de estados que contenga un estado de aceptación, para encontrar la concordancia más larga se debe seguir simulando el AFN hasta alcanzar *terminación*, es decir, un conjunto de estados desde el que no hay transiciones con el símbolo de entrada en curso.

Se presupone que la especificación de LEX está diseñada de modo que un programa fuente válido no puede llenar por completo el *buffer* de entrada, a no ser que el AFN haya alcanzado la terminación. Por ejemplo, cada compilador impone algunas restricciones a la longitud de un identificador, y se sabrá que no se ha respetado este límite cuando se desborde el *buffer* de la entrada o incluso antes.

Para encontrar la concordancia adecuada, se hacen dos modificaciones al algoritmo 3.4. Primero, siempre que se añada un estado de aceptación al conjunto de estados en curso, se registran la posición en curso de entrada y el patrón p_i correspondiente a este estado de aceptación. Si el conjunto de estados en curso ya contiene un estado de aceptación, entonces sólo se registra el patrón que aparezca primero en la especificación de LEX. Segundo, se continúa haciendo transiciones hasta que se alcanza la terminación. En la terminación, se retrocede el apuntador delantero a la posición en que ocurrió la última concordancia. El patrón que hizo dicha concordancia identifica al componente léxico encontrado, y el lexema emparejado es la cadena entre los apuntadores de inicio del lexema y los delanteros.

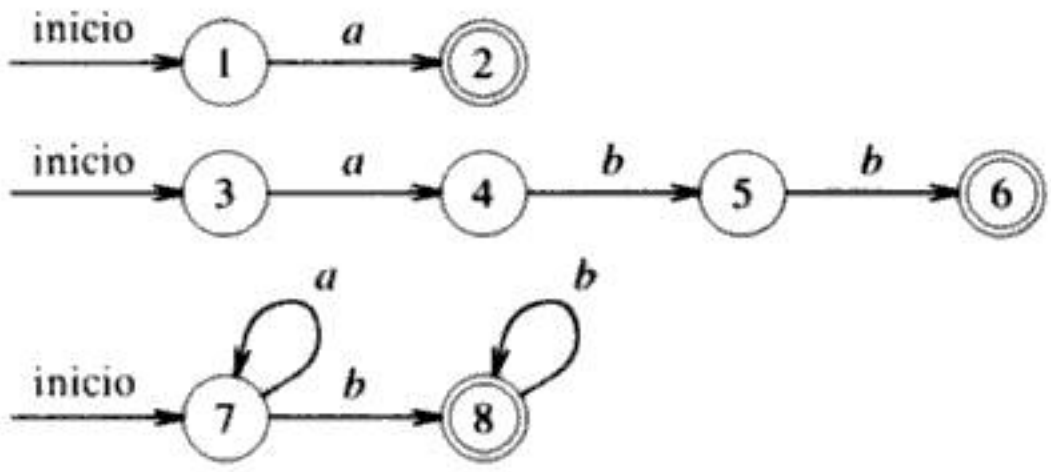
Normalmente, la especificación de LEX es tal que algún patrón, posiblemente un patrón de error, siempre concuerde. Sin embargo, si esto no es así se tiene una condición de error no prevista y el analizador léxico deberá transferir el control a alguna rutina de recuperación de error por omisión.

Ejemplo 3.18. Un sencillo ejemplo ilustra las ideas anteriores. Supóngase que se tiene el siguiente programa en LEX, que consta de tres expresiones regulares y ninguna definición regular.

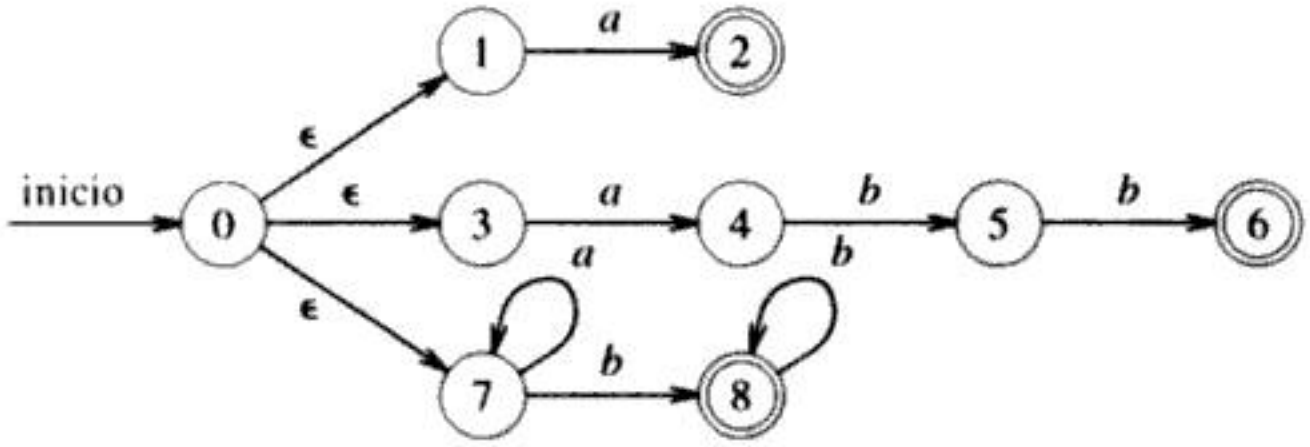
```
a      { }  /* aquí se omiten las acciones */
abb    { }
a*b+   { }
```

Los tres componentes léxicos anteriores se reconocen por medio del autómata de la figura 3.35(a). Se ha simplificado un poco el tercer autómata con relación al que produciría el algoritmo 3.3. Como se ha indicado anteriormente, se pueden convertir los AFN de la figura 3.35(a) en un AFN combinado N de la figura 3.35(b).

Ahora se considerará el comportamiento de N con la cadena de entrada *aaba* utilizando la modificación del algoritmo 3.4. En la figura 3.36 se muestran los conjuntos de estados y patrones que concuerdan conforme se procesa cada carácter de la entrada *aaba*. Esta figura muestra que el conjunto inicial de estados es {0, 1, 3, 7}. Cada uno de los estados 1, 3 y 7 tiene una transición en *a*, a los estados 2, 4 y 7, respectivamente. Puesto que el estado 2 es el estado de aceptación del primer patrón, se registra el hecho de que el primer patrón concuerda después de leer la primera *a*.



(a) AFN para *a*, *abb* y a^*b^+.



(b) AFN combinado.

Fig. 3.35. AFN que reconoce tres patrones distintos.

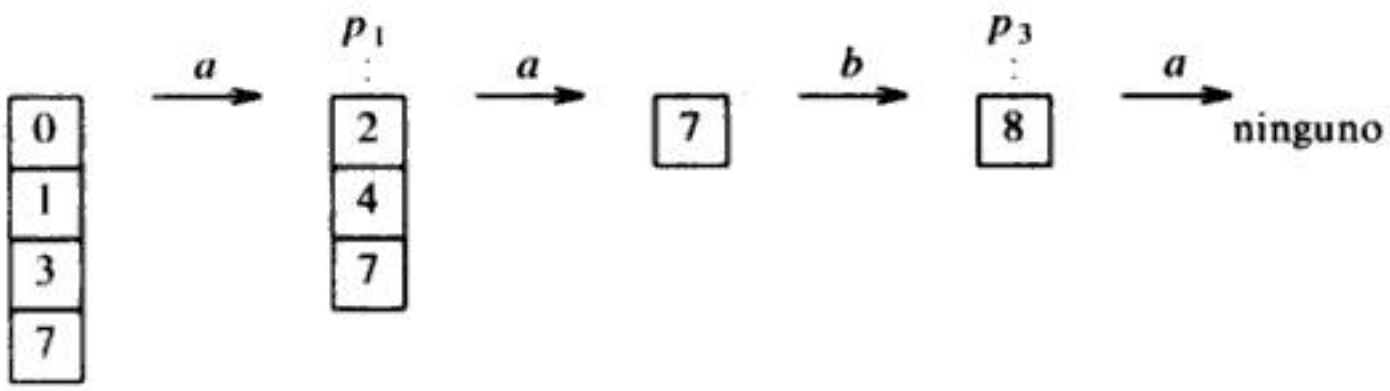


Fig. 3.36. Secuencia de conjuntos de estados visitados al procesar la entrada *aaba*.

Sin embargo, hay una transición del estado 7 al estado 7 en el segundo carácter de la entrada, de modo que se debe seguir haciendo transiciones. Hay una transición desde el estado 7 al estado 8 en el carácter de entrada *b*. El estado 8 es el estado de aceptación para el tercer patrón. Una vez alcanzado el estado 8, no hay transiciones posibles con el siguiente carácter de entrada *a*, así que se ha alcanzado la terminación. Puesto que se produjo la última concordancia después de leer el tercer carácter de entrada, se informa de que el tercer patrón ha concordado con el lexema *aab*. □

El papel de *acción*$_i$ asociado con el patrón p_i en la especificación de LEX es como sigue. Cuando se reconoce un caso de p, el analizador léxico ejecuta el programa asociado *acción*$_i$. Obsérvese que *acción*$_i$ no se ejecuta porque el AFN entra en un estado que incluye el estado de aceptación de p_i; *acción*$_i$ se ejecuta solamente si p_i resulta ser el patrón que produce la concordancia más larga.

AFD para analizadores léxicos

Otro enfoque para la construcción de un analizador léxico a partir de una especificación de LEX es utilizar un AFD para realizar la concordancia de patrones. La única diferencia es asegurarse de encontrar las adecuadas concordancias con los patrones. La situación es completamente análoga a la simulación modificada del AFN que se acaba de describir. Cuando se convierte un AFN en un AFD utilizando el algoritmo 3.2 para la construcción de subconjuntos, puede haber varios estados de aceptación en un subconjunto dado de estados no deterministas. En tal caso, tiene prioridad el estado de aceptación correspondiente al patrón listado en primer lugar en la especificación de LEX. Como en la simulación del AFN, la única modificación que hay que realizar es continuar haciendo transiciones de estados hasta alcanzar un estado sin estado siguiente (por ejemplo, el estado $\varnothing$) para el símbolo de entrada en curso. Para encontrar el lexema emparejado, se regresa a la última posición de entrada donde el AFD entró en un estado de aceptación.

Ejemplo 3.19. Si se convierte el AFN de la figura 3.35 en un AFD, se obtiene la tabla de transiciones de la figura 3.37, donde se han nombrado los estados del AFD mediante listas de los estados del AFN. La última columna de la figura 3.37 indica uno de los patrones reconocidos al entrar en ese estado del AFD. Por ejemplo, entre los estados 2, 4 y 7 del AFN, sólo el 2 es de aceptación, y es el estado de aceptación del autómata para la expresión regular a de la figura 3.35(a). Por tanto, el estado 247 del AFD reconoce el patrón a.

Obsérvese que la cadena abb concuerda con dos patrones, abb y a^*b^+, reconocidos en los estados 6 y 8 del AFN. El estado 68 del AFD, en la última línea de la tabla de transiciones, incluye por tanto dos estados de aceptación del AFN. Se ob-

Estado	Símbolo de Entrada		Patrón Anunciado
	a	b	
0137	247	8	ninguno
247	7	58	a
8	–	8	a^*b^+
7	7	8	ninguno
58	–	68	a^*b^+
68	–	8	abb

Fig. 3.37. Tabla de transiciones para un AFD.

serva que *abb* aparece antes que a^*b^+ en las reglas de traducción de la especificación de LEX, de modo que se anuncia que *abb* ha sido hallado en el estado 68 del AFD.

Con la cadena de entrada *aaba*, el AFD entra en el estado sugerido por la simulación del AFN que se muestra en la figura 3.36. Considérese un segundo ejemplo, la cadena de entrada *aba*. El AFD de la figura 3.37 empieza en el estado 0137. Con la entrada *a*, va al estado 247. Después, con la entrada *b* avanza hasta el estado 58, y con la entrada *a* no tiene un estado siguiente. Por tanto, se ha alcanzado la terminación, avanzando por los estados del AFD 0137, después 247 y luego 58. El último de éstos incluye el estado de aceptación 8 del AFN de la figura 3.35(a). Así, en el estado 58 el AFD anuncia que ha sido reconocido el patrón a^*b^+, y selecciona a *ab*, el prefijo de la entrada que condujo al estado 58, como lexema. □

Implantación del operador de preanálisis

Recuérdese de la sección 3.4 que es necesario el operador de preanálisis / en algunas situaciones, puesto que el patrón que representa un determinado componente léxico puede necesitar describir algún contexto posterior al lexema real. Cuando se convierte un patrón con / en un AFN, se puede considerar al / como si fuera ϵ, de modo que en realidad no se busca / en la entrada. Sin embargo, si una cadena representada por esta expresión regular es reconocida en el *buffer* de entrada, el final del lexema no es la posición del estado de aceptación del AFN. Más bien está en la última aparición del estado de este AFN en que tuvo una transición en el / (imaginario).

Ejemplo 3.20. En la figura 3.38, se muestra el AFN que reconoce el patrón para `IF` dado en el ejemplo 3.12. El estado 6 indica la presencia de la palabra clave `IF`; sin embargo, se encuentra el componente léxico `IF` buscando hacia atrás hasta la última aparición del estado 2. □

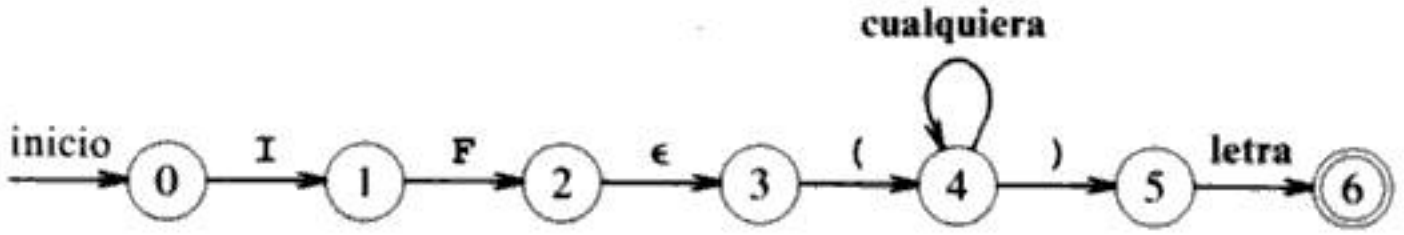


Fig. 3.38. AFN que reconoce la palabra clave `IF` de FORTRAN.

3.9 OPTIMACION DE BUSCADORES POR CONCORDANCIA DE PATRONES BASADOS EN AFD

En esta sección se introducen tres algoritmos que se han utilizado para implantar y optimar buscadores por concordancia de patrones construidos a partir de expresiones regulares. El primer algoritmo se puede incluir en un compilador de LEX porque construye un AFD directamente a partir de una expresión regular, sin construir un AFN intermedio durante el proceso.

El segundo algoritmo minimiza el número de estados de cualquier AFD, así que puede utilizarse para reducir el tamaño de un buscador por concordancia de patrones basado en AFD. El algoritmo es eficiente; su tiempo de ejecución es $O(n \log n)$, donde n es el número de estados del AFD. El tercer algoritmo se puede utilizar para producir representaciones rápidas y más compactas para la tabla de transiciones de un AFD que una sencilla tabla de dos dimensiones.

Estados significativos de un AFN

Se llama *significativo* a un estado de un AFD si tiene una transición de salida que no sea con ϵ. La construcción de subconjuntos de la figura 3.25 sólo utiliza los estados significativos en un subconjunto T cuando determina *cerradura-*ϵ (*mueve* (T, a)), el conjunto de estados es alcanzable desde T con entrada a. El conjunto *mueve* (s, a) es no vacío sólo cuando el estado s es significativo. Durante la construción, se pueden identificar dos subconjuntos si tienen los mismos estados significativos, y los dos o ninguno incluyen estados de aceptación del AFN.

Cuando se aplica la construcción de subconjuntos a un AFN obtenido a partir de una expresión regular mediante el algoritmo 3.3, se pueden aprovechar las propiedades exclusivas del AFN para combinar las dos construcciones. La construcción combinada relaciona los estados significativos del AFN con los símbolos de la expresión regular. La construcción de Thompson realiza un estado significativo justo cuando un símbolo del alfabeto aparece en una expresión regular. Por ejemplo, se construirán estados significativos para cada a y b en $(a \mid b)^* abb$.

Además, el AFN resultante tiene exactamente un estado de aceptación, pero el estado de aceptación no es significativo porque no tiene transiciones que salgan de él. Concatenando un marcador único del extremo derecho # a una expresión regular r, se da al estado de aceptación de r una transición en #, convirtiéndolo en estado significativo del AFN para $r\#$. En otras palabras, utilizando la expresión regular aumentada $(r)\#$ no hace falta ocuparse de los estados de aceptación conforme avanza la construcción de subconjuntos; cuando la construcción está completa, cualquier estado del AFD con una transición en # debe ser un estado de aceptación.

Una expresión regular aumentada se representa mediante un árbol sintáctico con símbolos básicos en las hojas y operadores en los nodos interiores. Un nodo interior se denomina *nodo-cat*, *nodo-o* o *nodo-ast* si está etiquetado con un operador de concatenación, de |, o de *, respectivamente. En la figura 3.39(a) se muestra un árbol sintáctico para una expresión regular aumentada donde los *nodos-cat* están marcados con puntos. Se puede construir el árbol sintáctico para una expresión regular igual que un árbol sintáctico para una expresión aritmética (véase Cap. 2).

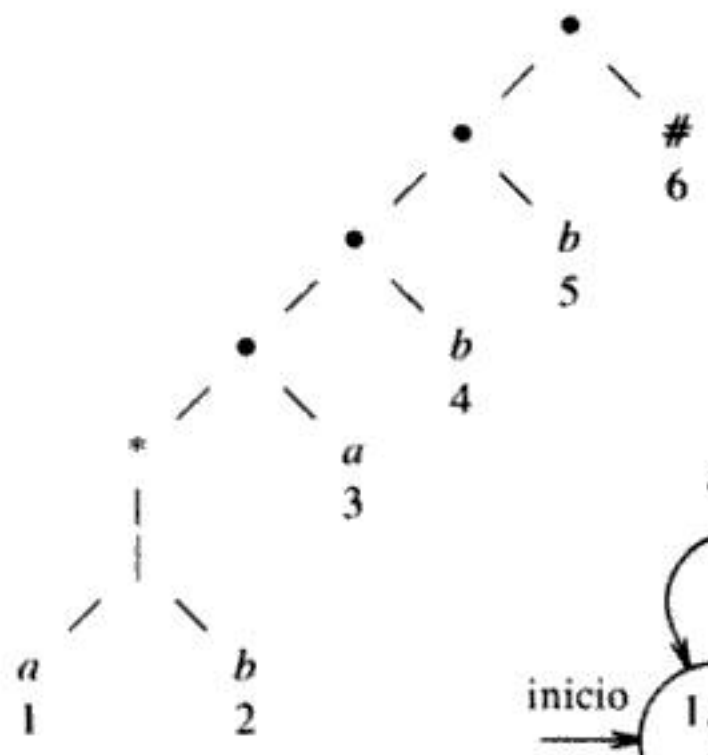
Se etiquetan las hojas del árbol sintáctico para una expresión regular con símbolos del alfabeto o con ϵ. A cada hoja no etiquetada con ϵ se le asocia un entero único y este entero se denomina *posición* de la hoja y también posición de su símbolo. Un símbolo repetido tiene por tanto varias posiciones. En el árbol sintáctico de la figura 3.39(a) las posiciones se muestran por debajo de los símbolos. Los estados numerados del AFN de la figura 3.39(c) corresponden a las posiciones de las hojas en el árbol sintáctico de la figura 3.39(a). No es una coincidencia que estos

estados sean los estados significativos del AFN. Los estados no significativos se nombran con letras mayúsculas en la figura 3.39(c).

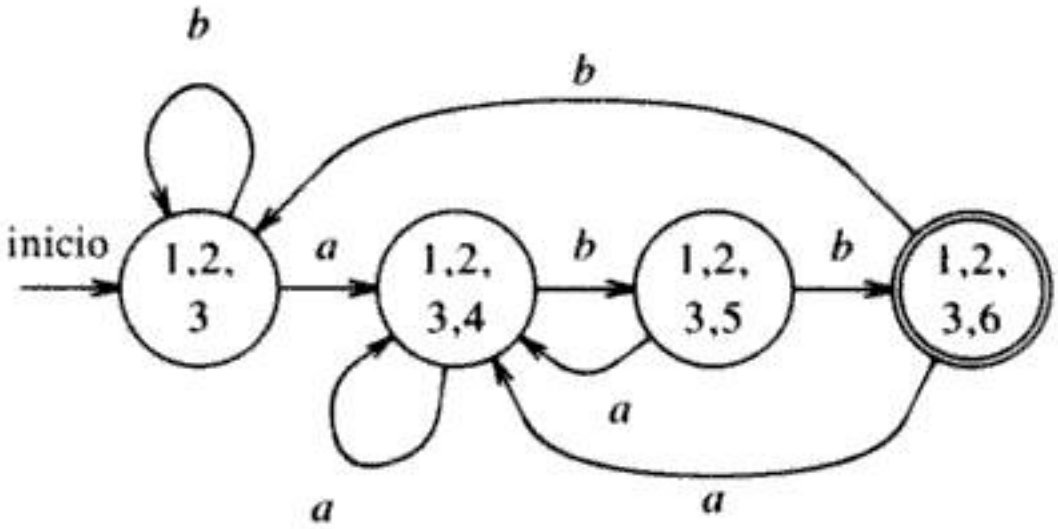
Se puede obtener el AFD de la figura 3.39(b) a partir del AFN de la figura 3.39(c) si se aplica la construcción de subconjuntos y se identifican los subconjuntos que contengan los mismos estados significativos. La identificación da como resultado la construcción de un estado menos, tal como demuestra una comparación con la figura 3.29.

Construcción de un AFD a partir de una expresión regular

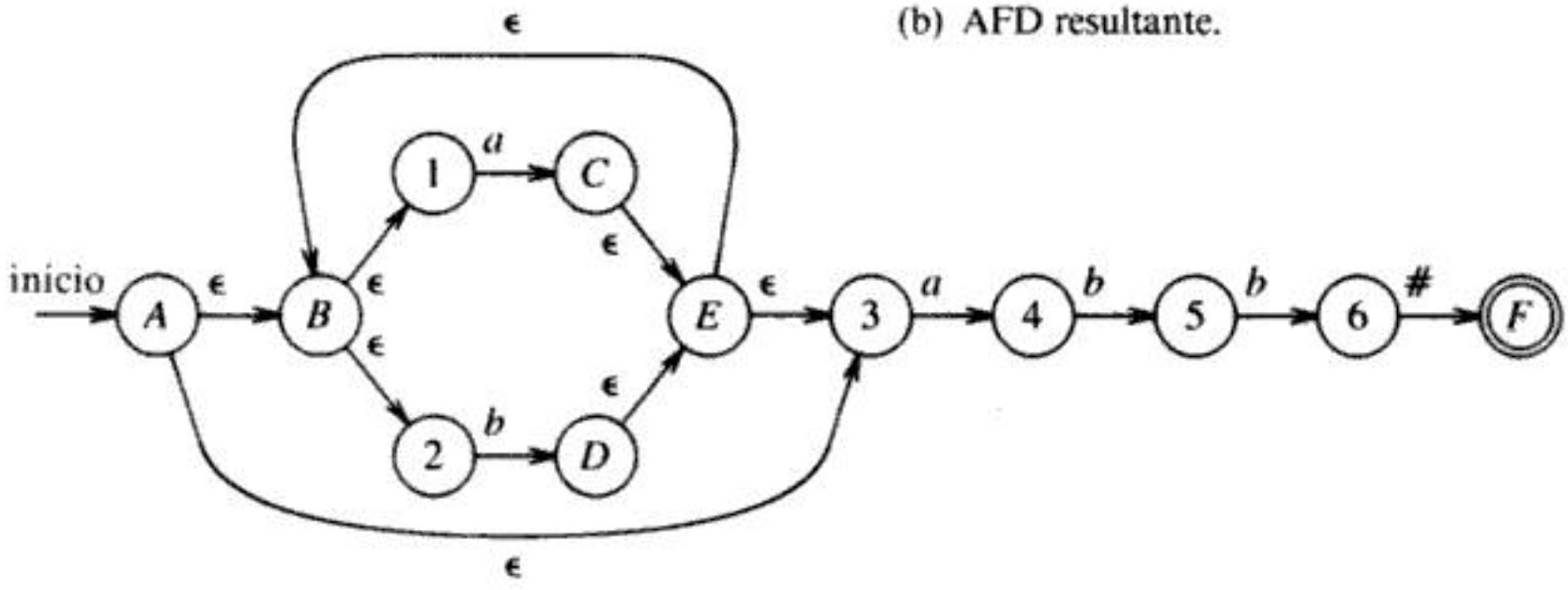
En esta sección se muestra cómo construir directamente un AFD a partir de una expresión regular aumentada $(r)\#$. Primero se construye un árbol sintáctico T para $(r)\#$ y después se calculan cuatro funciones: *anulable*, *primerapos*, *últimapos* y *si-*



(a) Arbol de sintaxis para $(a \mid b)^*abb\#$.



(b) AFD resultante.



(c) AFN subyacente.

Fig. 3.39. AFD y AFN construidos a partir de $(a \mid b)^*\ abb\#$.

guientepos, haciendo recorridos sobre T. Por último se construye el AFD a partir de *siguientepos*. Las funciones *anulable*, *primerapos* y *últimapos* se definen sobre los nodos del árbol sintáctico y se usan para calcular *siguientepos*, que está definida en el conjunto de posiciones.

Recordando la equivalencia entre los estados significativos del AFN y las posiciones de las hojas en el árbol sintáctico de la expresión regular, se puede abreviar la construcción del AFN construyendo el AFD cuyos estados corresponden a conjuntos de posiciones en el árbol. Las transiciones ϵ del AFN representan algunas estructuras de las posiciones bastante complicadas; en particular, codifican la información referente a la posibilidad de que una posición pueda seguir a otra. Es decir, cada símbolo de una cadena de entrada para un AFN puede ser emparejado por determinadas posiciones. Un símbolo de entrada c sólo puede emparejarse con posiciones en las que haya una c, pero no todas las posiciones con una c concuerdan necesariamente con un determinado caso de c en la cadena de entrada.

La noción de una posición concordando con un símbolo de entrada será definida en cuanto a la función *siguientepos* en posiciones del árbol sintáctico. Si i es una posición, *siguientepos* (i) es el conjunto de posiciones j tales que hay alguna cadena de entrada . . . cd . . . tal que i corresponde a esta aparición de c y j a esta aparición de d.

Ejercicio 3.21. En la figura 3.39(a), *siguientepos*(1) = {1, 2, 3}. El razonamiento es que si se ve una s correspondiente a la posición 1, se ha observado solamente una aparición de $a \mid b$ en la cerradura $(a \mid b)^*$. A continuación se podría ver la primera posición de otro caso de $a \mid b$, lo que explica por qué 1 y 2 están en *siguientepos* (1). Luego se podría observar la primera posición de lo que sigue a $(a \mid b)^*$, es decir, la posición 3. □

Para calcular la función *siguientepos*, es necesario conocer qué posiciones pueden concordar con el primer o último símbolo de una cadena generada por una determinada subexpresión de una expresión regular. (Esta información se utilizó informalmente en el Ejemplo 3.21.) Si r^* es tal subexpresión, entonces toda posición que pueda estar primero en r sigue a toda posición que pueda estar al final en r. De forma similar, si rs es una subexpresión, entonces toda primera posición de s sigue a toda última posición de r.

En cada nodo n del árbol sintáctico de una expresión regular, se define la función *primerapos* (n) que proporciona el conjunto de posiciones que pueden concordar con el primer símbolo de una cadena generada por la subexpresión con raíz en n. Asimismo, se define la función *últimapos* (n) que proporciona el conjunto de posiciones que pueden concordar con el último símbolo en esa cadena. Por ejemplo, si n es la raíz del árbol completo de la figura 3.39(a), entonces *primerapos* (n) = = {1, 2, 3} y *últimapos* (n) = {6}. En breve se dará un algoritmo para calcular estas funciones.

Para calcular *primerapos* y *últimapos*, es necesario conocer qué nodos son las raíces de las subexpresiones que generan lenguajes que incluyen la cadena vacía. A dichos nodos se les denomina *anulables*, y se define *anulable*(n) como verdadero si el nodo n es anulable, y falso en caso contrario.

Ahora se pueden dar las reglas para calcular las funciones *anulable*, *primerapos*,

últimapos y *siguientepos*. Para las tres primeras funciones, hay una regla base acerca de expresiones de un símbolo básico, y también tres reglas inductivas que permiten determinar el valor de las funciones que constituyen el árbol sintáctico desde abajo; en todos estos casos, las reglas inductivas corresponden a los tres operadores, unión, concatenación y cerradura. En la figura 3.40 se dan las reglas para *anulable* y *primerapos*. Las reglas para *últimapos* (*n*) son las mismas que para *primerapos* (*n*), pero con c_1 y c_2 invertidas, y no se mostrarán.

La primera regla para *anulable* establece que si *n* es una hoja etiquetada con ϵ, entonces *anulable* (*n*) es verdadera. La segunda regla establece que si *n* es una hoja etiquetada con un símbolo del alfabeto, entonces *anulable* (*n*) es falsa. En este caso, cada hoja corresponde a un solo símbolo de entrada, y por tanto no puede generar a ϵ. La última regla para *anulable* establece que si *n* es un nodo-ast con hijo c_1, entonces *anulable* (*n*) es verdadera, puesto que la cerradura de una expresión genera un lenguaje que incluye a ϵ.

Otro ejemplo, la cuarta regla para *primerapos* establece que si *n* es un nodo-cat con hijo izquierdo c_1 e hijo derecho c_2, y si *anulable* (c_1) es verdadera, entonces

$$primerapos\,(n) = primerapos\,(c_1) \cup primerapos\,(c_2)$$

de lo contrario, *primerapos* (*n*) = *primerapos* (c_1). Lo que esta regla establece es que si en una expresión *rs*, *r* genera ϵ, entonces las primeras posiciones de *s* "aparecen a lo largo de" *r* y también son las primeras posiciones de *rs*; de otro modo, sólo las primeras posiciones de *r* son primeras posiciones de *rs*. El razonamiento es similar para el resto de las reglas para *anulable* y *primerapos*.

La función *siguientepos* (*i*) indica qué posiciones pueden seguir a la posición *i* en el árbol sintáctico. Dos reglas definen todas las formas en que una posición puede seguir a otra.

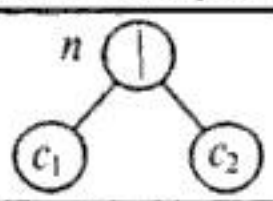
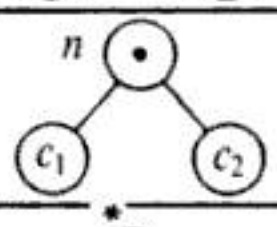
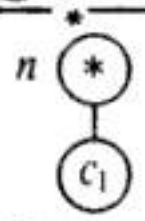
NODO *n*	*anulable* (*n*)	*primerapos* (*n*)
n es una hoja etiquetada con ϵ	**true**	$\varnothing$
n es una hoja etiquetada con la posición *i*	**false**	$\{i\}$
	*anulable* (c_1) **or** *anulable* (c_2)	*primerapos* (c_1) ∪ *primerapos* (c_2)
	*anulable* (c_1) **and** *anulable* (c_2)	**if** *anulable* (c_1) **then** *primerapos* (c_1) ∪ *primerapos* (c_2) **else** *primerapos* (c_1)
	**true**	*primerapos* (c_1)

Fig. 3.40. Reglas para calcular *anulable* y *primerapos*.

1. Si n es un nodo-cat con hijo izquierdo c_1 e hijo derecho c_2, e i es una posición dentro de *últimapos* (c_1), entonces todas las posiciones de *primerapos* (c_2) están en *siguientepos* (i).
2. Si n es un nodo-ast, e i es una posición dentro de *últimapos* (n), entonces todas las posiciones de *primerapos* (n) están en *siguientepos* (i).

Si se han calculado *primerapos* y *últimapos* para cada nodo, *siguientepos* de cada posición se puede calcular haciendo un recorrido en profundidad del árbol sintáctico.

Ejemplo 3.22. En la figura 3.41 se muestran los valores de *primerapos* y *últimapos* en todos los nodos del árbol de la figura 3.39(a); *primerapos* (n) aparece a la izquierda del nodo n y *últimapos* (n) a la derecha. Por ejemplo, *primerapos* de la hoja situada más a la izquierda etiquetada con a es {1}, puesto que dicha hoja está etiquetada con la posición 1. Asimismo, *primerapos* de la segunda hoja es {2}, puesto que esta hoja está etiquetada con la posición 2. Por la tercera regla de la figura 3.40, *primerapos* de sus padres es {1, 2}.

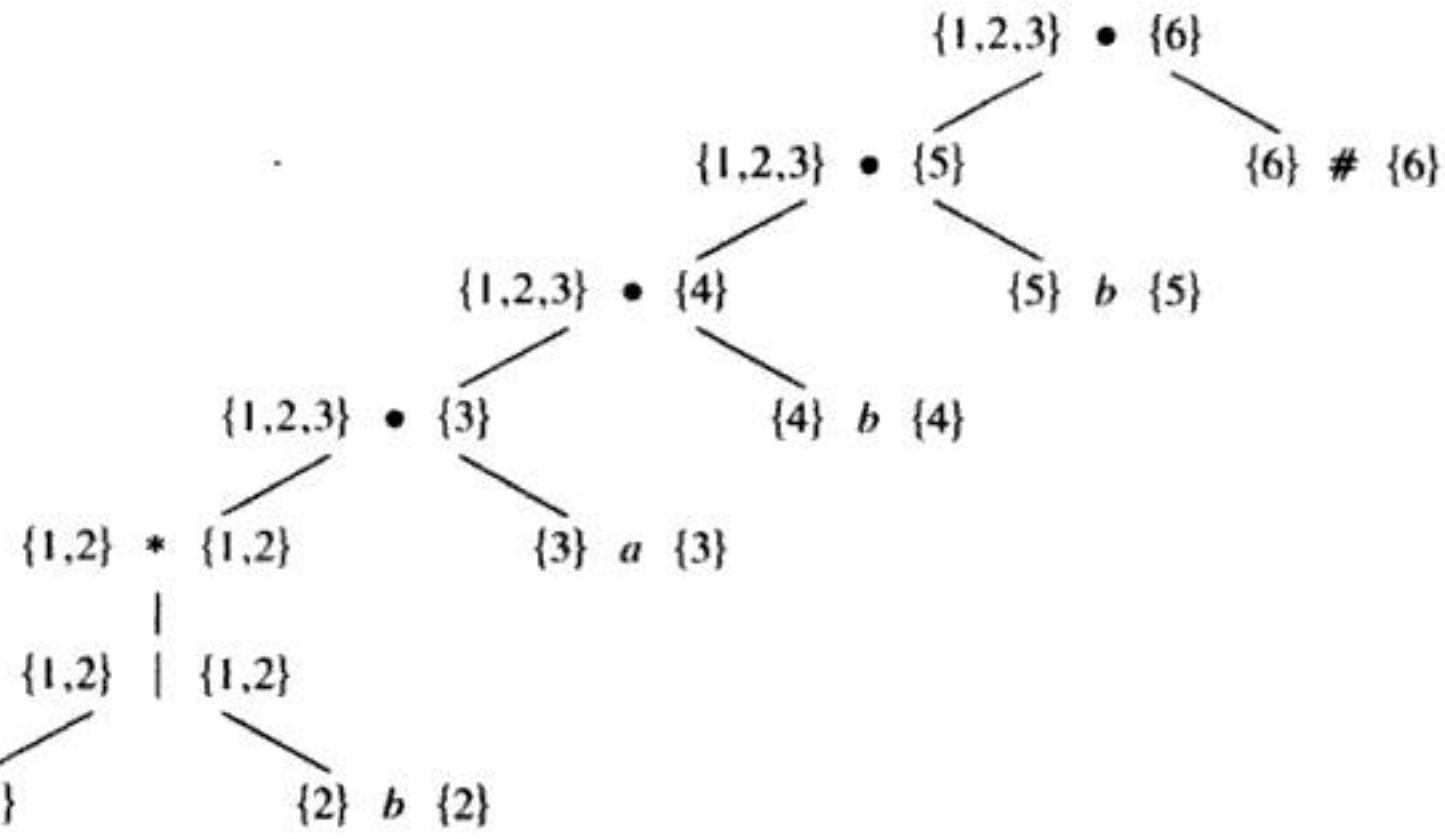


Fig. 3.41. *primerapos* y *últimapos* para los nodos del árbol de sintaxis para $(a \mid b)^*abb\#$.

El nodo etiquetado con * es el único nodo anulable. Por tanto, por la condición **if** de la cuarta regla, *primerapos* para el padre de este nodo (el que representa la expresión $(a \mid b)^*a$) es la unión de {1, 2} y {3}, que son las *primerapos* de sus hijos izquierdo y derecho. Por otra parte, la condición **else** se aplica a *últimapos* de este nodo, puesto que la hoja de la posición 3 no es anulable. Por tanto, *últimapos* del padre del nodo-ast contiene sólo 3.

Ahora se calcula *siguientepos* de abajo a arriba para cada nodo del árbol sintáctico de la figura 3.41. En el nodo-ast, se añaden 1 y 2 a *siguientepos* (1) y a *siguientepos* (2) utilizando la regla 2. En el padre del nodo-ast, se añade 3 a *siguientepos* (1) y a *siguientepos* (2) por la regla 1. En el siguiente nodo-ast, se añade 4 a *siguiente-*

pos (3) por la regla 1. En los dos nodos-est siguientes se añade 5 a *siguientepos* (4) y 6 a *siguientepos* (5) utilizando la misma regla. Esto completa la construcción de *siguientepos*. En la figura 3.42 se resume *siguientepos*.

NODO	*siguientepos*
1	{1, 2, 3}
2	{1, 2, 3}
3	{4}
4	{5}
5	{6}
6	–

Fig. 3.42. La función *siguientepos*.

Se puede ilustrar la función *siguientepos* creando un grafo dirigido con un nodo para cada posición y una arista dirigida desde el nodo *i* al modo *j* si *j* está en *siguientepos* (*i*). En la figura 3.43 se muestra este grafo dirigido para *siguientepos* de la figura 3.42.

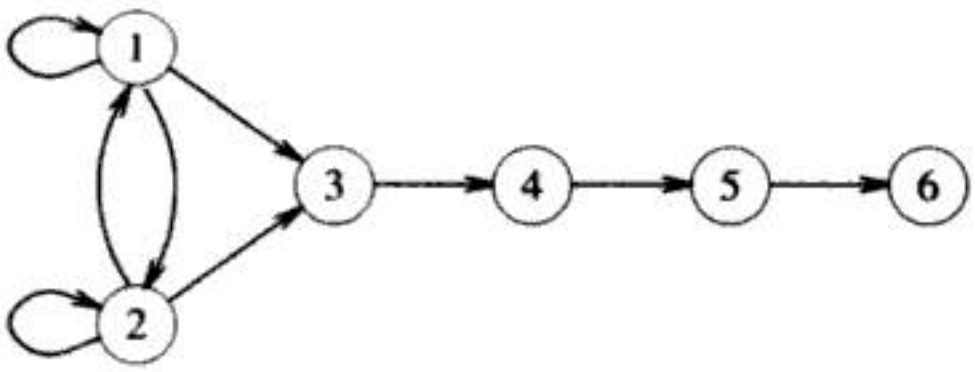


Fig. 3.43. Grafo dirigido para la función *siguientepos*.

Es interesante observar que este diagrama se convertiría en un AFN sin transiciones ϵ para la expresión regular en cuestión si:

1. se convierten todas las posiciones de *primerapos* de la raíz en estados de inicio,
2. se etiqueta cada arista dirigida (i, j) con el símbolo de la posición j, y
3. se convierte la posición asociada con # en el único estado de aceptación.

Por tanto, no es sorprendente que se pueda convertir el grafo de *siguientepos* en un AFD utilizando la construcción de subconjuntos. Toda la construcción puede efectuarse sobre las posiciones, usando el siguiente algoritmo. □

Algoritmo 3.5. Construcción de un AFD a partir de una expresión regular *r*.

Entrada. Una expresión regular *r*.

Salida. Un AFD *D* que reconoce a $L(r)$.

Método.

1. Constrúyase un árbol sintáctico para la expresión regular aumentada $(r)\#$, donde $\#$ es un marcador de final único que se añade a (r).
2. Constrúyanse las funciones *anulable*, *primerapos*, *últimapos* y *siguientepos* haciendo recorridos en profundidad de T.
3. Constrúyanse *estadosD*, el conjunto de estados de D, y *tranD*, la tabla de transiciones para D por el procedimiento de la figura 3 44. Los estados dentro de *estadosD* son conjuntos de posiciones; al principio, cada estado está "no marcado", y un estado se convierte en "marcado" justo antes de considerar sus transiciones de salida. El estado de inicio de D es *primerapos* (*raíz*), y los estados de aceptación son todos los que contienen la posición asociada con el marcador de final $\#$. □

Ejemplo 3.23. Constrúyase un AFD para la expresión regular $(a \mid b)^*abb$. En la figura 3.39(a) se muestra el árbol sintáctico para $((a \mid b)^*abb)\#$. *anulable* es verdadero sólo para el nodo etiquetado con *. En la figura 3.41 se muestran las funciones *primerapos* y *últimapos*, y *siguientepos* se muestra en la figura 3.42.

En la figura 3.41, *primerapos* de la raíz es $\{1, 2, 3\}$. Sea A este conjunto y considérese el símbolo de entrada a. Las posiciones 1 y 3 son para a, así que sea $B = siguientepos\,(1) \cup siguientepos\,(3) = \{1, 2, 3, 4\}$. Puesto que este conjunto no ha aparecido hasta ahora se hace $tranD\,[A, a] := B$.

```
al principio, el único estado no marcado en estadosD es
    primerapos (raíz), donde raíz es la raíz del árbol
    de sintaxis para (r)#;
while hay un estado sin marcar T en estadosD do begin
    marcar T;
    for cada símbolo de entrada a do begin
        sea U el conjunto de posiciones que están en
            siguientepos (p) para alguna posición p en T, tal
            que el símbolo en la posición p es a;
        if U no está vacío y no está en estadosD then
            añadir U como estado no marcado a estadosD;
        tranD [T, a] := U
    end
end
```

Fig. 3.44. Construcción de un AFD.

Cuando se considera la entrada b, se observa que de las posiciones de A, sólo 2 está asociada a b, así que se debe considerar el conjunto $siguientepos\,(2) = \{1, 2, 3\}$. Puesto que este conjunto ya había aparecido, no se añade a *estadosD*, sino que se añade la transición $tranD\,[A, b] := A$.

Ahora se continúa con $B = \{1, 2, 3, 4\}$. Los estados y transiciones que finalmente se obtienen son los mismos que se mostraron en la figura 3.39(b). □

Minimización del número de estados de un AFD

Una conclusión teórica importante es que todo conjunto regular es reconocido por un AFD con el mínimo de estados que es único hasta nombres de estados. En esta sección, se muestra cómo construir este AFD del mínimo de estados reduciendo al mínimo posible el número de estados en un AFD determinado sin afectar al lenguaje que se está reconociendo. Supóngase que se tiene un AFD M con su conjunto de estados S y su alfabeto de símbolos de entrada $\sum$. Se supone que cada estado tiene una transición con cada símbolo de la entrada. Si no fuera así, se puede introducir un nuevo "estado inactivo" d, con transiciones de d a d con todas las entradas, y añadir una transición desde el estado s al d en la entrada a si no hubo transición desde s en a.

Se dice que la cadena w *distingue* al estado s del estado t si, empezando con el AFD M en el estado s y alimentándolo con la entrada w, se termina en un estado de aceptación, pero comenzando en el estado t y alimentándolo con la entrada w, se termina en un estado de no aceptación o viceversa. Por ejemplo, ϵ distingue cualquier estado de aceptación de cualquier estado de no aceptación, y en el AFD de la figura 3.29, los estados A y B han sido diferenciados por la entrada bb, puesto que A va al estado de no aceptación C en la entrada bb, mientras que B va al estado de aceptación E en la misma entrada.

El algoritmo para minimizar el número de estados de un AFD funciona encontrando todos los grupos de estados que pueden ser diferenciados por una cadena de entrada. Cada grupo de estados que no puede diferenciarse se fusiona entonces en un único estado. El algoritmo opera manteniendo y refinando una partición del conjunto de estados. Cada grupo de estados dentro de la partición está formado por estados que aún no han sido distinguidos unos de otros, y todos los pares de estados escogidos de entre grupos diferentes han sido considerados distinguible por una entrada.

Al principio, la partición consta de dos grupos: los estados de aceptación y los estados de no aceptación. El paso fundamental consiste en tomar un grupo de estados, por ejemplo $A = \{s_1, s_2, \ldots, s_k\}$ y un símbolo de entrada a y comprobar qué transiciones tienen los estados $s_1, s_2, \ldots, s_k$ con la entrada a. Si dichas transiciones son hacia estados de dos o más grupos distintos de la partición en curso, entonces hay que dividir A para que las transiciones desde los subconjuntos de A queden todas confinadas en un único grupo de la partición en curso. Supóngase, por ejemplo, que s_1 y s_2 van a los estados t_1 y t_2 con la entrada a, y que t_1 y t_2 están en diferentes grupos de la partición. Entonces se debe dividir A al menos en dos subconjuntos, para que un subconjunto contenga a s_1, y el otro, a s_2. Obsérvese que t_1 y t_2 son diferenciados por alguna cadena w, y s_1 y s_2, por la cadena aw.

Este proceso de dividir grupos dentro de la partición en curso se repite hasta que no sea necesario dividir ningún otro grupo. Aunque se ha justificado por qué pueden realmente diferenciarse los estados que han sido divididos en diferentes grupos, no se ha indicado por qué los estados que no han sido divididos en grupos diferentes no pueden en ningún caso ser diferenciados por ninguna cadena de entrada. Sin embargo, tal es el caso y se deja la prueba de ese hecho al lector interesado en la teoría (véase, por ejemplo, Hopcroft y Ullman [1979]). También se deja al lector intere-

sado la prueba de que el AFD construido tomando un estado de cada grupo de la partición final y eliminando después los estados inactivos y los estados no alcanzables desde el estado de inicio que tiene tan pocos estados como cualquier AFD que acepte el mismo lenguaje.

Algoritmo 3.6. Minimización del número de estados de un AFD.

Entrada. Un AFD M con un conjunto de estados S, un conjunto de entradas $\sum$, transiciones definidas para todos los estados y las entradas, un estado de inicio s_0 y un conjunto de estados de aceptación F.

Salida. Un AFD M' que acepta el mismo lenguaje que M y tiene el menor número de estados posible.

Método.

1. Constrúyase una partición inicial $\prod$ del conjunto de estados con dos grupos: los estados de aceptación F y los estados de no aceptación $S-F$.
2. Aplíquese el procedimiento de la figura 3.45 a $\prod$ para construir una nueva partición $\prod_{\text{nueva}}$.
3. Si $\prod_{\text{nueva}} = \prod$, hacer $\prod_{\text{final}} = \prod$ y continuar con el paso (4). Si no, repetir el paso (2) con $\prod := \prod_{\text{nueva}}$.
4. Escójase un estado en cada grupo de la partición $\prod_{\text{final}}$ como *representante* de este grupo. Los representantes serán los estados de AFD reducidos M'. Sea s un estado representante, y supóngase que con la entrada a hay una transición de M desde s a t. Sea r el representante del grupo de t (r puede ser t). Entonces M' tiene una transición desde s a r con la entrada a. Sea el estado de inicio de M' el representante del grupo que contiene al estado de inicio s_0 de M, y sean los estados de aceptación de M' los representantes que están en F. Obsérvese que cada grupo de $\prod_{\text{final}}$ consta únicamente de estados en F o no tiene ningún estado en F.
5. Si M' tiene un estado inactivo, es decir, un estado d que no es de aceptación y que tiene transiciones hacia él mismo con todos los símbolos de entrada, elimínese d de M'. Elimínense igualmente todos los estados que no sean alcanzables desde el estado inicial. Todas las transiciones a d desde otros estados se convierten en indefinidas. □

```
for cada grupo G de ∏ do begin
    partición de G en subgrupos tales que dos estados s y t
        de G están en el mismo subgrupo si, y sólo si, para todos
        los símbolos de entrada a, los estados s y t tienen
        transiciones en a hacia estados del mismo grupo de ∏;
    /* en el peor caso, un estado estará sólo en un subgrupo */
    sustituir G en ∏nueva por el conjunto de todos los subgrupos formados
end
```

Fig. 3.45. Construcción de $\prod_{\text{nueva}}$.

Ejemplo 3.24. Considérese de nuevo el AFD representado en la figura 3.29. La partición inicial $\prod$ consta de dos grupos: (E), el estado de aceptación, y $(ABCD)$, los estados de no aceptación. Para construir $\prod_{\text{nueva}}$, el algoritmo de la figura 3.45 primero considera (E). Puesto que este grupo consta de un solo estado, ya no se puede dividir más, así que (E) se coloca en $\prod_{\text{nueva}}$. Entonces, el algoritmo considera el grupo $(ABCD)$. Con la entrada a, cada uno de estos estados tiene una transición a B, así que todos podrían permanecer en un mismo grupo en lo que a la entrada a se refiere. Sin embargo, con la entrada b, A, B y C van a miembros del grupo $(ABCD)$ de $\prod$, mientras que D va a E, un miembro de otro grupo. Por tanto, dentro de $\prod_{\text{nueva}}$ el grupo $(ABCD)$ se debe dividir en dos nuevos grupos, (ABC) y (D); $\prod_{\text{nueva}}$ es entonces $(ABC)(D)(E)$.

En el siguiente recorrido por el algoritmo de la figura 3.45, de nuevo no hay división en la entrada a, pero (ABC) debe dividirse en dos nuevos grupos, $(AC)(B)$, puesto que en la entrada b, A y C tienen ambas una transición a C, mientras que B tiene una transición a D, un miembro de un grupo de la partición distinto del de C. Así, el siguiente valor de $\prod$ es $(AC)(B)(D)(E)$.

En el siguiente recorrido por el algoritmo de la figura 3.45, no se puede dividir ninguno de los grupos de un solo estado. La única posibilidad es intentar dividir (AC). Sin embargo, A y C van al mismo estado B en la entrada a y al mismo estado C en la entrada b. Por consiguiente, después de este recorrido, $\prod_{\text{nueva}} = \prod$. $\prod_{\text{final}}$ es entonces $(AC)(B)(D)(E)$.

Si se escoge A como representante del grupo (AC), y B, D y E, como representantes de los grupos de un solo estado, se obtiene el autómata reducido cuya tabla de transiciones se muestra en la figura 3.46. El estado A es el estado de inicio y el estado E es el único estado de aceptación.

Estado	Símbolo de Entrada	
	a	b
A	B	A
B	B	D
D	B	E
E	B	A

Fig. 3.46. Tabla de transiciones del AFD reducido.

Por ejemplo, en el autómata reducido, el estado E tiene una transición al estado A con la entrada b, puesto que A es el representante del grupo de C y hay una transición de E a C con la entrada b en el autómata original. Una modificación similar tuvo lugar en la entrada para A y la entrada b. Todas las demás transiciones están copiadas de la figura 3.29. No hay ningún estado inactivo en la figura 3.46, y todos los estados son alcanzables desde el estado de inicio A. □

Minimización de estados en analizadores léxicos

Para aplicar el procedimiento de minimización de estados a los AFD construidos en la sección 3.7, se debe comenzar el algoritmo 3.5 con una partición inicial que coloque en grupos diferentes a todos los estados que indiquen distintos componentes léxicos.

Ejemplo 3.25. En el caso del AFD de la figura 3.37, la partición inicial agruparía 0137 con 7, puesto que ninguno de ellos indicó un componente léxico; también se agruparían 8 y 58, porque ambos indicaron el componente léxico $a^*\ b^+$. Otros estados estarían solos en un grupo. Inmediatamente se descubre que 0137 y 7 pertenecen a grupos distintos, puesto que van a diferentes grupos en la entrada *a*. Asimismo, 8 y 58 no están juntos dadas sus transiciones en la entrada *b*. Por tanto, el AFD de la figura 3.37 es el autómata con el número mínimo de estados que hace este trabajo. □

Métodos para compresión de tablas

Como ya se ha indicado, existen muchas formas de implantar la función de transición de un autómata finito. El proceso del análisis léxico ocupa una parte considerable del tiempo del compilador, puesto que es el único proceso que debe observar en la entrada un carácter a la vez. Por tanto, el analizador léxico debe minimizar el número de operaciones que realiza por cada carácter de entrada. Si se utiliza un AFD para ayudar a implantar el analizador léxico, es aconsejable una representación eficiente de la función de transición. Una matriz bidimensional, indexada por estados y caracteres, proporciona el acceso más rápido, pero puede ocupar demasiado espacio (por ejemplo, varios cientos de estados por 128 caracteres). Un esquema más compacto, pero más lento, es utilizar una lista enlazada para almacenar las transiciones de salida de cada estado, con una transición "por omisión" al final de la lista. Obviamente la transición que ocurre con más frecuencia es la elegida para dicha omisión.

Existe una implantación más sutil que combina el acceso rápido de la representación por medio de matrices con la compacidad de las estructuras de listas. Se utiliza una estructura de datos que consta de cuatro matrices indexadas por números de estados, como se mostró en la figura 3.47[7]. Se utiliza la matriz *base* para determinar la posición base de las entradas para cada estado almacenado en las matrices *siguiente* y *revisa*. Se utiliza la matriz *omisión* para determinar una posición base alternativa en caso de que la posición base en curso no sea válida.

Para calcular *sigte edo* (*s*, *a*), la transición para el estado *s* con el símbolo de entrada *a*, primero se consulta el par de matrices *siguiente* y *revisa*. Sus entradas para el estado *s* se encuentran en la posición $l = base\,[s]+a$, donde el carácter *a* es considerado como un entero. Se considera *siguiente* [*l*] como el siguiente estado de *s* en

---

[7] En la práctica hay otra matriz indexada por *s*, que da el patrón con el que concuerda, si lo hay, cuando se entra en el estado *s*. Esta información se deriva de los estados del AFN que constituyen al estado *s* del AFD.

la entrada a si *revisa* [l] = s. Si *revisa* [l] ≠ s, se determina q = *omisión* [s] y se repite el procedimiento recursivamente, utilizando q en lugar de s. El procedimiento es el siguiente:

```
procedure sigte edo (s, a);
        if revisa [base [s]+a] = s then
                return siguiente [base [s]+a ]
        else
                return sigte edo (omisión [s], a)
```

La finalidad de la estructura de la figura 3.47 es acortar las matrices *siguiente* y *revisa* pequeñas, aprovechando las similitudes entre estados. Por ejemplo, el estado q, el valor por omisión para el estado s, puede ser el estado que establezca que se está "trabajando con un identificador", como el estado 10 de la figura 3.13. Quizá se entre en s después de mostrar `th` un prefijo de la palabra clave `then` además de como un prefijo de un identificador. En el carácter de entrada `e` se debe ir a un estado especial que recuerde que se ha observado la cadena `the`, pero si no, el estado s se comporta como el estado q. Por tanto, se asigna *revisa* [*base* [s+]+`e`] a s y *siguiente* [*base*[s]+`e`] al estado de `the`.

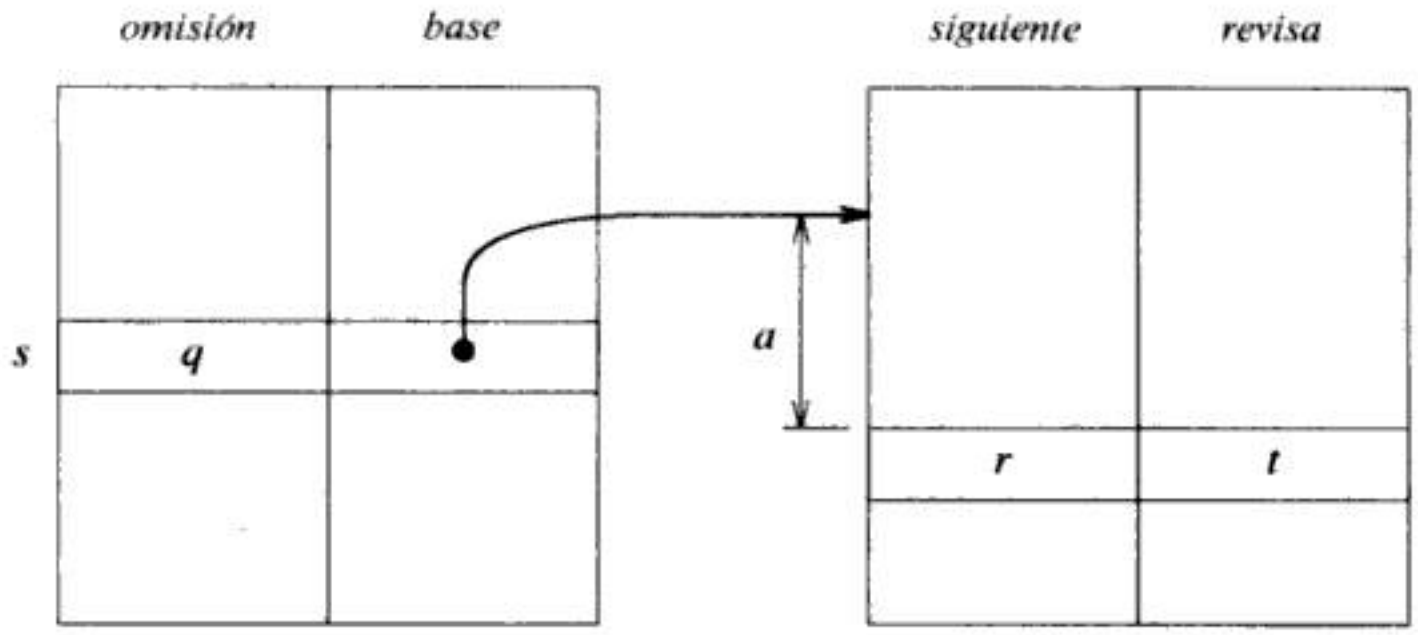


Fig. 3.47. Estructura de datos para representar tablas de transiciones.

Aunque quizá no se puedan elegir los valores de *base* para que no queden sin utilizar entradas de *siguiente* y *revisa*, la experiencia muestra que la sencilla estrategia de asignar la *base* al número menor de forma que las entradas especiales puedan llenarse sin interferir con las entradas ya existentes, es bastante buena y utiliza poco más espacio que el mínimo posible.

Se puede reducir *revisa* a una matriz indexada por estados si el AFD tiene la propiedad de que las aristas entrantes en cada estado t tengan la misma etiqueta a. Para implantar este esquema, se hace *revisa* [t] = a y se reemplaza la prueba de la línea 2 del procedimiento *sigte edo* por

```
if r [base [s]+a]] = a then
```

EJERCICIOS

3.1 ¿Cuál es el alfabeto de entrada de cada uno de los siguientes lenguajes?
a) Pascal
b) C
c) FORTRAN 77
d) Ada
e) LISP

3.2 ¿Cuáles son las convenciones concernientes al uso de espacios en blanco en cada uno de los lenguajes del ejercicio 3.1?

3.3 Identifíquense los lexemas que forman los componentes léxicos en los siguientes programas. Dense valores razonables de atributo para los componentes léxicos.
a) Pascal

```
function max (i, j, : integer ) : integer;
{ devuelve el máximo de los enteros i y j }
begin
    if i > j then max := i
    else max := j
end;
```

b) C

```
int max ( i, j ) int i, j;
/* devuelve el máximo de los enteros i y j */
{
    return i>j?i:j;
}
```

c) FORTRAN 77

```
      FUNCTION MAX ( I, J )
C     DEVUELVE EL MAXIMO DE LOS ENTEROS I Y J
      IF (I .GT. J) THEN
           MAX = I
      ELSE
           MAX = J
      END IF
      RETURN
```

3.4 Escríbase un programa para la función `sigtecar()` de la sección 3.4 utilizando el esquema de manejo de *buffers* con centinelas descrito en la sección 3.2.

3.5 En una cadena de longitud n, ¿cuántos de los siguientes hay?
a) prefijos
b) sufijos
c) subcadenas

d) prefijos propios
e) subsecuencias

*3.6 Descríbanse los lenguajes representados por las siguientes expresiones regulares:
a) 0(0 | 1)*0
b) ((ϵ | 0)1*)*
c) (0 | 1)*0(0 | 1)(0 | 1)
d) 0* 10* 10* 10*
e) (00 | 11)* ((01 | 10)(00 | 11)* (01 | 10)(00 | 11)*)*

*3.7 Escríbanse definiciones regulares para los siguientes lenguajes.
a) Todas las cadenas de letras que contienen las cinco vocales en orden.
b) Todas las cadenas de letras en las que las letras están en orden lexicográfico ascendente.
c) Comentarios que consisten en una cadena encerrada entre /* y */ sin ningún */ intermedio, a menos que aparezca entre las comillas " y ".
*d) Todas las cadenas de dígitos sin ningún dígito repetido.
e) Todas las cadenas de dígitos con a lo sumo un dígito repetido.
f) Todas las cadenas de 0 y 1 con un número par de dígitos 0 y un número impar de dígitos 1.
g) El conjunto de movimientos del ajedrez, como $p-$k4 o $kbp \times qn$.
h) Todas las cadenas de 0 y 1 que no contienen la subcadena 011.
i) Todas las cadenas de 0 y 1 que no contienen la subsecuencia 011.

3.8 Especifíquese la forma lexicográfica de las constantes numéricas en los lenguajes del ejercicio 3.1.

3.9 Especifíquese la forma lexicográfica de los identificadores y palabras clave de los lenguajes del ejercicio 3.1.

3.10 En la figura 3.48 se relacionan en orden decreciente de precedencia las construcciones de expresiones regulares que permite LEX. En esta tabla, *c* representa todo carácter simple, *r* representa una expresión regular, y *s* una cadena.
a) Se debe eliminar el significado especial de los símbolos de operadores

```
\" . ^ $ [ ] * + ? { } | /
```

si el símbolo de operador se utiliza como carácter de emparejamiento. Esto se puede hacer encerrando entre comillas el carácter, utilizando uno de dos estilos de entrecomillado. La expresión "*s*" se empareja con la cadena *s* literalmente, siempre que no aparece ninguna " en *s*. Por ejemplo, "**" concuerda con la cadena **. También se podría haber emparejado esta cadena con la expresión **. Obsérvese que un * sin comillas es un ejemplo del operador de la cerradura de Kleene. Escríbase una expresión regular en LEX que concuerde con la cadena "\.
b) En LEX, una clase de caracteres *complementada* es una clase de caracteres en la que el primer símbolo es ^. Una clase de caracteres comple-

EXPRESIÓN	EMPAREJA CON	EJEMPLO
c	cualquier carácter *c* que no sea operador	`a`
c	el carácter *c* literalmente	`\*`
"*s*"	la cadena *s* literalmente	`"**"`
.	cualquier carácter excepto de nueva línea	`a.*b`
^	el comienzo de línea	`^abc`
$	el fin de línea	`abc$`
[*s*]	cualquier carácter en *s*	`[abc]`
[^*s*]	cualquier carácter que no esté en *s*	`[^abc]`
*r**	cero o más *r*	`a*`
r+	una o más *r*	`a+`
r?	cero o una *r*	`a?`
r{*m*,*n*}	*m* a *n* casos de *r*	`a{1,5}`
r_1 r_2	r_1 y entonces r_2	`ab`
r_1 | r_2	r_1 o r_2	`a | b`
(*r*)	*r*	`(a | b)`
r_1/r_2	r_1 cuando va seguida de r_2	`abc/123`

Fig. 3.48. Expresiones regulares en LEX.

mentada concuerda con cualquier carácter que no esté en la clase. Por tanto, `[^a]` concuerda con cualquier carácter que no sea una `a`, `[^A-Za-z]` concuerda con cualquier carácter que no sea una letra mayúscula o minúscula, etcétera. Demuéstrese que para toda definición regular con clases de caracteres complementadas existe una expresión regular equivalente sin clases de caracteres complementadas.

c) La expresión regular *r*{*m*,*n*} empareja de *m* a *n* ocurrencias del patrón *r*. Por ejemplo, `a {1,5}` concuerda con una cadena de una a cinco `a`. Demuéstrese que para toda expresión regular que contenga operadores de repetición existe una expresión regular equivalente sin dichos operadores.

d) El operador ^ concuerda con el extremo izquierdo de una línea. Este es el mismo operador que introduce una clase de caracteres complementada, pero el contexto en donde aparezca ^ siempre determinará un significado único para este operador. El operador `$` concuerda con el extremo derecho de una línea. Por ejemplo, `^[^aeiou]*$` concuerda con cualquier línea que no contenga una vocal en minúsculas. ¿Existe para toda expresión regular que contenga los operadores ^ y `$` una expresión regular equivalente sin dichos operadores?

3.11 Escríbase un programa en LEX que copie un archivo, sustituyendo cada secuencia no nula de espacios en blanco por un solo espacio en blanco.

3.12 Escríbase un programa en LEX que copie un programa en FORTRAN, sustituyendo todos los ejemplos de `DOUBLE PRECISION` por `REAL`.

3.13 Utilícese una especificación propia para palabras clave e identificadores para el FORTRAN 77 del ejercicio 3.9 para identificar los componentes léxicos de las siguientes proposiciones:

```
IF(I) = CMPLEX
IF(I) ASSIGN5CMPLEX
IF(I) 10,20,30
IF(I) GOTO15
IF(I) THEN
```

Puede escribirse una especificación propia para palabras clave e identificadores en LEX.

3.14 En el sistema UNIX, el mandato `sh` del *shell* utiliza los operadores de la figura 3.49 en expresiones de nombres de archivo para describir conjuntos de nombres de archivo. Por ejemplo, la expresión de nombres de archivo `*.o` concuerda con todos los nombres de archivo que terminen con `.o`; `sort.?` concuerda con todos los archivos de la forma `sort.`*c*, donde *c* es cualquier carácter. Las clases de caracteres se pueden abreviar como en `[a-z]`. Demuéstrese cómo se pueden expresar por medio de expresiones regulares las expresiones de nombres de archivos del *shell*.

3.15 Modifíquese el algoritmo 3.1 para encontrar el mayor prefijo de la entrada que sea aceptado por el AFD.

3.16 Constrúyanse autómatas finitos no deterministas para las siguientes expresiones regulares utilizando el algoritmo 3.3. Muéstrese la secuencia de movimiento realizada por cada uno de ellos al procesar la cadena de entrada *ababbab*.

a) $(a \mid b)^*$
b) $(a^* \mid b^*)^*$
c) $((\epsilon \mid a)b^*)^*$
d) $(b \mid b)^* abb\ (a \mid b)^*$

EXPRESIÓN	EMPAREJA CON	EJEMPLO
'*s*'	la cadena *s* literalmente	'\'
c	el carácter *c* literalmente	\'
*	cualquier cadena	`*.o`
?	cualquier carácter	`sort1.?`
[*s*]	cualquier carácter en *s*	`sort.[cso]`

Fig. 3.49. Expresiones de nombre de archivo en el programa `sh`.

3.17 Conviértanse los AFN del ejercicio 3.16 en AFD utilizando el algoritmo 3.2. Muéstrese la secuencia de movimientos realizada por cada uno de ellos al procesar la cadena de entrada *ababbab*.

3.18 Constrúyanse AFD para las expresiones regulares del ejercicio 3.16 usando el algoritmo 3.5. Compárese el tamaño de los AFD con el de los construidos en el ejercicio 3.17.

3.19 Constrúyase un autómata finito determinista a partir del diagrama de transiciones para los componentes léxicos de la figura 3.10.

3.20 Amplíese la tabla de la figura 3.40 para incluir los operadores de expresiones regulares ? y $^+$.

3.21 Minimícese el número de estados en los AFD del ejercicio 3.18 utilizando el algoritmo 3.6.

3.22 Se puede demostrar que dos expresiones regulares son equivalentes comprobando que su AFD de número mínimo de estados son los mismos, excepto para los nombres de estados. Utilizando esta técnica, demuéstrese que las siguientes expresiones regulares son todas equivalentes.

a) $(a \mid b)^*$
b) $(a^* \mid b^*)^*$
c) $((\epsilon \mid a)b^*)^*$

3.23 Constrúyanse AFD con mínimo de estados para las siguientes expresiones regulares.

a) $(a \mid b)^*a\ (a \mid b)$
b) $(a \mid b)^*a\ (a \mid b)(a \mid b)$
c) $(a \mid b)^*a\ (a \mid b)(a \mid b)(a \mid b)$

**d) Demuéstrese que cualquier autómata finito determinista para las expresiones regulares $(a \mid b)^*a\ (a \mid b)(a \mid b) \ldots (a \mid b)$, donde hay $n-1$ $(a \mid b)$ al final, debe tener al menos 2^n estados.

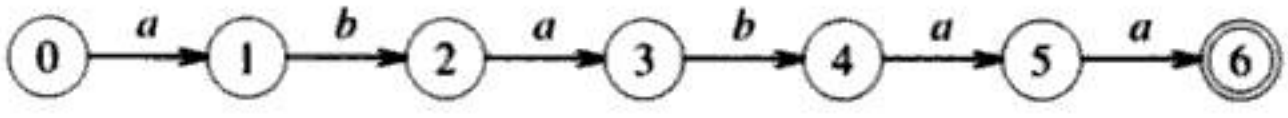
3.24 Constrúyase la representación de la figura 3.47 para la tabla de transiciones del ejercicio 3.19. Selecciónense estados por omisión y pruébense los dos métodos siguientes de construcción de la matriz *siguiente* y compárense las cantidades de espacio utilizado:

a) Comenzando con los estados más densos (aquellos con el mayor número de entradas que difieran de sus estados de omisión), primero colóquense las entradas para los estados en la matriz *siguiente*.
b) Colóquense las entradas para los estados en la matriz *siguiente* en orden aleatorio.

3.25 Una variante del esquema de compresión de tablas de la sección 3.9 sería evitar un procedimiento *sigte_edo* recursivo, utilizando una posición por omisión fija para cada estado. Constrúyase la representación de la figura 3.47 para la tabla de transiciones del ejercicio 3.19 utilizando esta técnica no recursiva. Compárense los requisitos de espacio con los del ejercicio 3.24.

3.26 Sea $b_1b_2 \ldots b_m$ una cadena de patrones, llamada *palabra clave*. Un *trie* para una palabra clave es un diagrama de transiciones con $m+1$ estados donde cada estado corresponde a un prefijo de la palabra clave. Para $1 \leqslant s \leqslant m$,

hay una transición del estado $s-1$ al estado s en el símbolo b_s. Los estados inicial y final corresponden a la cadena vacía y a la palabra completa, respectivamente. El *trie* para la palabra clave *ababaa* es:



Ahora se define una *función de fallo* f en cada estado del diagrama de transiciones, excepto el estado de inicio. Supóngase que los estados s y t representan a los prefijos u y v de la palabra clave. Entonces se define $f(s) = t$ si, y sólo si, v es el sufijo propio mayor de u que también es el prefijo de la palabra clave. La función de fallo f para el *trie* anterior es

s	1	2	3	4	5	6
$f(s)$	0	0	1	2	3	1

Por ejemplo, los estados 3 y 1 representan a los prefijos *aba* y *a* de la palabra clave *ababaa*. $f(3) = 1$ porque *a* es el sufijo propio mayor de *aba* que es prefijo de la palabra clave.

a) Constrúyase la función de fallo para la palabra clave *abababaab*.

*b) Sean 0, 1, . . . , m los estados del *trie*, donde 0 es el estado de inicio. Demuéstrese que el algoritmo de la figura 3.50 calcula correctamente la función de fallo.

*c) Demuéstrese que durante toda la ejecución del algoritmo de la figura 3.50, se ejecuta la proposición de asignación $t := f(t)$ en el lazo interno a lo sumo m veces.

*d) Demuéstrese que el algoritmo funciona en un tiempo $O(m)$.

```
/* calcula la función de fallo f para b1 . . . bm */
t := 0; f(1) := 0;
for s := 1 to m − 1 do begin
    while t > 0 and b_{s+1} ≠ b_{t+1} do t := f(t);
    if b_{s+1} = b_{t+1} then begin t := t+1; f(s+1) := t end;
    else f(s+1) := 0
end
```

Fig. 3.50. Algoritmo para calcular la función de fallo del ejercicio 3.26.

3.27 El algoritmo KMP de la figura 3.51 utiliza la función de fallo f construida como en el ejercicio 3.26 para determinar si la palabra clave $b_1 \ldots b_m$ es o no una subcadena de una cadena objetivo $a_1 \ldots a_n$. Se numeran los estados en el *trie* para $b_1 \ldots b_m$ de 0 a m como en el ejercicio 3.26(b).

a) Aplíquese el algoritmo KMP para determinar si *ababaa* es o no una subcadena de *abababaab*.

```
/* revisa si a_1 ... a_n contiene a b_1 ... b_m como subcadena */
s := 0;
for i := 1 to n do begin
    while s > 0 and a_i ≠ b_{s+1} do s := f(s);
    if a_i = b_{s+1} then s := s+1
    if s = m then return "sí"
end;
return "no"
```

Fig. 3.51. Algoritmo KMP.

*b) Demuéstrese que el algoritmo KMP devuelve "sí" si, y sólo si, $b_1 \ldots b_m$ es una subcadena de $a_1 \ldots a_n$.

*c) Demuéstrese que el algoritmo KMP se ejecuta en un tiempo $O(m+n)$.

*d) Dada una palabra clave y, demuéstrese que la función de fallo puede utilizarse para construir, en un tiempo $O(|y|)$, un AFD con $|y| + 1$ estados para la expresión regular .*y.*, donde . representa cualquier carácter de entrada.

**3.28 Defínase el *periodo* de una cadena s como un entero p tal que s se puede expresar como $(uv)^k u$, para una $k \geqslant 0$, donde $|uv| = p$ y v no es la cadena vacía. Por ejemplo, 2 y 4 son periodos de la cadena *abababa*.

a) Demuéstrese que p es un periodo de una cadena s si, y sólo si, $st = us$ para algunas cadenas t y u de longitud p.

b) Demuéstrese que si p y q son periodos de una cadena s y si $p+q \leqslant |s| +$ $+$ mcd(p,q), entonces mcd (p,q) es un periodo de s, donde mcd(p,q) es el máximo común divisor de p y q.

c) Sea $mp(s_i)$ el menor periodo de un prefijo de longitud i de una cadena s. Demuéstrese que la función de fallo f tiene la propiedad de que $f(j) = j - mp(s_{j-1})$.

*3.29 Sea el *prefijo repetitivo más corto* de una cadena s el prefijo más corto u de s tal que $s = u^k$, para una $k \geqslant 1$. Por ejemplo, *ab* es el prefijo repetitivo más corto de *abababab* y *aba* es el prefijo repetitivo más corto de *aba*. Constrúyase un algoritmo que encuentre el prefijo repetitivo más corto de una cadena s en un tiempo $O(|s|)$. *Sugerencia.* Utilícese la función de fallo del ejercicio 3.26.

3.30 Una *cadena de Fibonacci* se define como sigue:

$$s_1 = b$$
$$s_2 = a$$
$$s_k = s_{k-1}\, s_{k-2}, \text{ para } k > 2.$$

Por ejemplo, $s_3 = ab$, $s_4 = aba$ y $s_5 = abaab$.

a) ¿Cuál es la longitud de s_n?

**b) ¿Cuál es el periodo más pequeño de s_n?

c) Constrúyase la función de fallo para s_6.

*d) Por inducción, demuéstrese que la función de fallo para s_n se puede expresar por $f(j) = j - |s_{k-1}|$, donde k es tal que $|s_k| \leqslant j+1 < |s_{k+1}|$ para $1 \leqslant j \leqslant |s_n|$.

e) Aplíquese el algoritmo KMP para determinar si s_6 es o no una subcadena de la cadena objeto s_7.

f) Constrúyase un AFD para la expresión regular .*s_6.*.

**g) En el algoritmo KMP, ¿cuál es el número máximo de aplicaciones consecutivas de la función de fallo ejecutada para determinar si s_k es una subcadena de la cadena objeto s_{k+1}?

3.31 Se pueden ampliar los conceptos de *trie* y función de fallo del ejercicio 3.26 de una sola palabra clave a un conjunto de palabras clave como sigue. Cada estado del *trie* corresponde a un prefijo de una o más palabras clave. El estado de inicio corresponde a la cadena vacía, y un estado que corresponda a una palabra clave completa es un estado final. Estados adicionales pueden convertirse en finales durante el cálculo de la de función de fallo. En la figura 3.52 se muestra el diagrama de transiciones para el conjunto de palabras clave {`he, she, his, hers`}.

Para el *trie* se define una *función de transición q* que transforma pares estado-símbolo en estados tal que $g(s, b_{j+1}) = s'$ si el estado s corresponde a un prefijo $b_1 \ldots b_j$ de una palabra clave, y s' corresponde a un prefijo $b_1 \ldots b_j b_{j+1}$. Si s_0 es el estado de inicio, se define $g(s_0, a) = s_0$ para todos los símbolos de entrada a que no sean el símbolo inicial de ninguna palabra clave. Entonces se establece $g(s, a) = fallo$ para cualquier transición no definida. Obsérvese que no hay transiciones *fallo* para el estado de inicio.

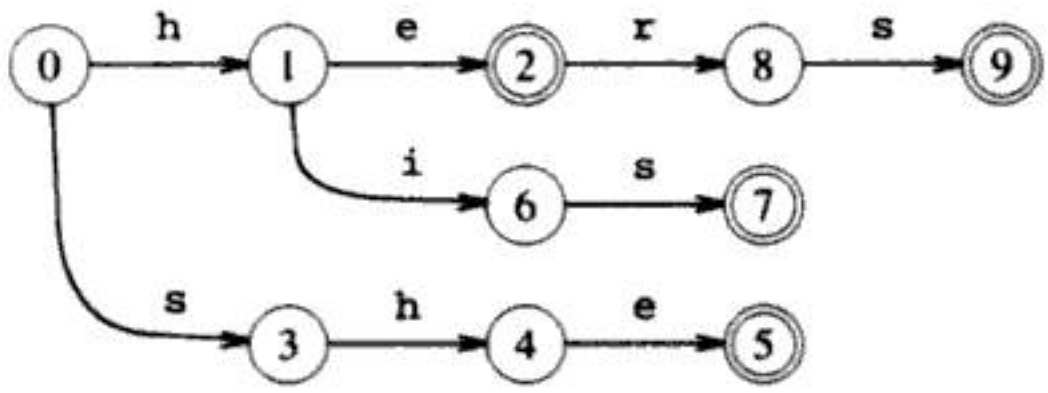


Fig. 3.52. *Trie* para las palabras clave {`he, she, his, hers`}.

Supóngase que los estados s y t representan a los prefijos u y v de algunas palabras clave. Entonces, se define $f(s) = t$ si, y sólo si, v es el mayor sufijo propio de u que es además prefijo de una palabra clave. La función de fallo f para el diagrama de transiciones anterior es:

s	1	2	3	4	5	6	7	8	9
$f(s)$	0	0	0	1	2	0	3	0	3

Por ejemplo, los estados 4 y 1 representan a los prefijos `sh` y `h`. $f(4) = 1$ porque `h` es el mayor sufijo propio de `sh` prefijo de alguna palabra clave. Se puede calcular la función de fallo f para estados de profundidad creciente utilizando el algoritmo de la figura 3.53. La profundidad de un estado es su distancia desde el estado de inicio.

```
for cada estado s de profundidad 1 do
    f(s) := s0;
for cada profundidad d ≥ 1 do
    for cada estado sd de profundidad d y carácter a
        tal que g(sd, a) = s' do begin
            s := f(sd);
            while g(s, a) = fallo do s := f(s);
            f(s') := g(s, a);
end
```

Fig. 3.53. Algoritmo para calcular la función de fallo para el *trie* de palabras clave.

Obsérvese que como $g(s_0, c) \neq fallo$ para cualquier carácter c, se garantiza la terminación del lazo **while** de la figura 3.53. Después de asignar $g(t, a)$ a $f(s')$, si $g(t, a)$ es un estado final, también se convierte s' en un estado final, si no lo es ya.

a) Constrúyase la función de fallo para el conjunto de palabras clave {*aaa*, *abaaa*, *ababaaa*}.

*b) Demuéstrese que el algoritmo de la figura 3.53 calcula correctamente la función de fallo.

*c) Demuéstrese que se puede calcular la función de fallo en un tiempo proporcional a la suma de las longitudes de las palabras clave.

3.32 Sea g la función de transición, y f, la función de fallo del ejercicio 3.31 para un conjunto de palabras clave $K = \{y_1, y_2, \ldots, y_k\}$. El algoritmo AC de la figura 3.54 utiliza g y f para determinar si una cadena objeto $a_1 \ldots a_n$ contiene o no una subcadena que sea una palabra clave. El estado s_0 es el estado de inicio del diagrama de transiciones para K, y F es el conjunto de estados finales.

```
/* revisa si a1 ... an contiene una palabra clave como subcadena */
s := s0;
for i := 1 to n do begin
    while g(s, ai) = fallo do s := f(s);
    s := g(s, ai);
    if s está en F then return "sí"
end;
return "no"
```

Fig. 3.54. Algoritmo AC.

a) Aplíquese el algoritmo AC a la cadena de entrada `ushers` utilizando las funciones de transición y de fallo del ejercicio 3.31.

*b) Demuéstrese que el algoritmo AC devuelve "sí" si, y sólo si, alguna palabra clave y es una subcadena de $a_1 \ldots a_n$.

*c) Demuéstrese que el algoritmo AC hace a lo sumo $2n$ transiciones al procesar una cadena de entrada de longitud n.

d) Demuéstrese que a partir del diagrama de transiciones y la función de fallo para un conjunto de palabras clave $\{y_1, y_2, \ldots, y_k\}$, se puede construir un AFD con a lo sumo $\sum_{i=1}^{k} |y_i| + 1$ estados en un tiempo lineal para la expresion regular. $(y_1 \mid y_2 \mid \ldots \mid y_k)$.*.

e) Modifíquese el algoritmo AC para imprimir cada palabra clave que se encuentre en la cadena objeto.

3.33 Utilícese el algoritmo del ejercicio 3.32 para construir un analizador léxico para las palabras clave de Pascal.

3.34 Defínase *scm*(x, y), una *subsecuencia común más larga* de dos cadenas x e y, como una cadena que es subsecuencia tanto de x como de y y es tan larga como cualquiera de estas subsecuencias. Por ejemplo, `rana` es una subsecuencia común más larga de `razona` y `rebana`. Se define $d(x, y)$, la *distancia* entre x e y, como el número mínimo de inserciones y supresiones necesario para transformar x en y. Por ejemplo, d(`razona`, `rebana`) = 4.

a) Demuéstrese que para dos cadenas cualesquiera x e y, la distancia entre x e y y la longitud de su subsecuencia común más larga están relacionadas por $d(x, y) = |x| + |y| - |scm(x, y)|$.

*b) Escríbase un algoritmo que considere dos cadenas x e y como entrada y produzca una subsecuencia común más larga de x e y como salida.

3.35 Defínase $e(x, y)$, la *distancia de edición* entre dos cadenas x e y, como el número mínimo de inserciones, supresiones y sustituciones de caracteres necesario para transformar x en y. Sea $x = a_1 \ldots a_m$ e $y = b_1 \ldots b_n$. Se puede calcular $e(x, y)$ mediante un algoritmo de programación dinámica utilizando una matriz de distancias $d[0..m, 0..n]$, donde $d[i, j]$ es la distancia de edición entre $a_1 \ldots a_i$ y $b_1 \ldots b_j$. Se puede utilizar el algoritmo de la figura 3.55 para calcular la matriz d. La función *reemp* es el coste de reemplazar un carácter: *reemp* $(a_i, b_j) = 0$ si $a_i = b_j$, 1 en otro caso.

```
for i := 0 to m do d[i, 0] := i;
for j := 1 to n do d[0, i] := j;
for i := 1 to m do
    for j := 1 to n do
        D[i, j] := min (d[i-1, j-1] + reemp(a_i, b_j),
                        d[i-1, j] + 1,
                        d[i, j-1] + 1)
```

Fig. 3.55. Algoritmo para calcular la distancia de edición entre dos cadenas.

a) ¿Cuál es la relación entre la distancia del ejercicio 3.34 y la distancia de edición?
b) Utilícese el algoritmo de la figura 3.55 para calcular la distancia de edición entre *ababb* y *babaaa*.
c) Constrúyase un algoritmo que imprima la secuencia mínima de transformaciones de edición necesaria para transformar x en y.

3.36 Dése un algoritmo que tome como entrada una cadena x y una expresión regular r, y produzca como salida una cadena y en $L(r)$ tal que $d(x, y)$ sea lo más pequeña posible, donde d es la función de distancia del ejercicio 3.34.

EJERCICIOS DE PROGRAMACION

P3.1 Escríbase un analizador léxico en Pascal o C para los componentes léxicos mostrados en la figura 3.10.

P3.2 Escríbase una especificación para los componentes léxicos de Pascal, y a partir de esta especificación constrúyanse los diagramas de transiciones. Utilícense los diagramas de transiciones para implantar un analizador léxico para Pascal en un lenguaje como C o Pascal.

P3.3 Complétese el programa en LEX de la figura 3.18. Compárense el tamaño y la velocidad del analizador léxico resultante producido por LEX con el programa escrito en el ejercicio P3.1.

P3.4 Escríbase una especificación en LEX para los componentes léxicos de Pascal y con el compilador de LEX constrúyase un analizador léxico para Pascal.

P3.5 Escríbase un programa que tome como entrada una expresión regular y el nombre de un archivo, y produzca como salida todas las líneas del archivo que contengan una subcadena representada por la expresión regular.

P3.6 Añádase un esquema de recuperación de errores al programa en LEX de la figura 3.18 para permitirle seguir buscando componentes léxicos en presencia de errores.

P3.7 Prográmese un analizador léxico a partir del AFD construido en el ejercicio 3.18 y compárese este analizador léxico con el construido en los ejercicios P3.1 y P3.3.

P3.8 Constrúyase una herramienta que produzca un analizador léxico a partir de una descripción de una expresión regular de un conjunto de componentes léxicos.

NOTAS BIBLIOGRAFICAS

Las limitaciones impuestas a los aspectos léxicos de un lenguaje suelen estar determinadas por el ambiente en que se creó el lenguaje. Cuando se diseñó FORTRAN en 1954, las tarjetas perforadas eran un medio común de entrada. En FORTRAN

se ignoraron los espacios en blanco debido en parte a que los perforistas, que preparaban las tarjetas a partir de notas escritas a mano tendían a equivocarse al contar los espacios en blanco (Backus [1981]). La separación en ALGOL 58 de la representación en *hardware* a partir del lenguaje de referencia fue un acuerdo alcanzado debido a que un miembro del comité de diseño insistió, "No, nunca usaré un punto para el signo decimal". (Wegstein [1981]).

Knuth [1973a] presenta otras técnicas para manejar la entrada con *buffers*. Feldman [1979b] analiza las dificultades prácticas del reconocimiento de componentes léxicos en FORTRAN 77.

Las expresiones regulares fueron estudiadas por primera vez por Kleene [1956], que estaba interesado en describir los acontecimientos que se podían representar con el modelo de autómata finito de actividad nerviosa de McCulloch y Pitts [1943]. La minimización de los autómatas finitos fue estudiada por primera vez por Huffman [1954] y Moore [1956]. La equivalencia entre autómatas deterministas y no deterministas en cuanto a su capacidad para reconocer lenguajes fue mostrada por Rabin y Scott [1959]. McNaughton y Yamada [1960] describen un algoritmo para construir un AFD directamente a partir de una expresión regular. En Hopcroft y Ullman [1979] se puede encontrar más información sobre la teoría de las expresiones regulares.

Pronto se comprendió que las herramientas para construir analizadores léxicos a partir de especificaciones en forma de expresiones regulares serían útiles en la implantación de compiladores. En Johnson y otros [1968] se analiza uno de estos primeros sistemas. LEX, el lenguaje estudiado en este capítulo, se debe a Lesk [1975], y se ha utilizado para construir analizadores léxicos para muchos compiladores que usan el sistema UNIX. El compacto esquema de la sección 3.9 para las tablas de transición es obra de S. C. Johnson, que fue el primero en usarlo para la implantación del generador de analizadores sintácticos Yacc (Johnson [1975]). En Dencker, Dürre y Heuft [1984], se estudian y evalúan otros esquemas de compresión de tablas.

El problema de la implantación compacta de tablas de transiciones ha sido estudiado teóricamente en un planteamiento general por Tarjan y Yao [1979], y por Fredman, Komlós y Szemerédi [1984]. Cormack, Horspool y Kaiserswerth [1985] introducen un algoritmo de dispersión perfecta basado en este trabajo.

Se han utilizado expresiones regulares y autómatas finitos para muchas aplicaciones, además de para la compilación. Muchos editores de texto usan expresiones regulares para búsquedas en contexto. Thompson [1968], por ejemplo, describe la construcción de un AFN a partir de una expresión regular (Algoritmo 3.3) en el contexto del editor de textos QED. El sistema UNIX tiene tres programas de búsqueda de propósito general basados en expresiones regulares: `grep`, `egrep` y `fgrep`. `grep` no permite unión o paréntesis para agrupar en sus expresiones regulares, pero sí una forma limitada de referencia hacia atrás como en SNOBOL. `grep` emplea los algoritmos 3.3 y 3.4 para buscar sus patrones de expresiones regulares. Las expresiones regulares de `egrep` son similares a las de LEX, salvo para iteración y preanálisis. `egrep` utiliza un AFD con una construcción diferida de estados para buscar sus patrones de expresiones regulares, como se indicó en la sección 3.7. `fgrep` busca patrones formados por conjuntos de palabras clave utilizando el algoritmo de Aho y Co-

rasick [1975], que se analiza en los ejercicios 3.31 y 3.32. Aho [1980] analiza el rendimiento relativo de dichos programas.

Las expresiones regulares han sido muy utilizadas en sistemas de recuperación de textos, en lenguajes de consulta de bases de datos y en lenguajes para procesamiento de archivos, como AWK (Aho, Kernighan y Weinberger [1979]). Jarvis [1976] utilizó expresiones regulares para describir imperfecciones en circuitos impresos. Cherry [1982] utilizó el algoritmo de concordancia de palabras clave del ejercicio 3.32 para buscar mal lenguaje en manuscritos.

El algoritmo de concordancia de patrones de cadenas de los ejercicios 3.26 y 3.27 es obra de Knuth, Morris y Pratt [1977]. Este artículo también contiene un buen estudio de periodos dentro de cadenas. Otro algoritmo eficiente para la concordancia de cadenas fue inventado por Boyer y Moore [1977], quienes demostraron que normalmente se puede determinar la concordancia de una subcadena sin tener que examinar todos los caracteres de la cadena objeto. La dispersión también ha resultado ser una técnica efectiva para la concordancia de patrones de cadenas (Harrison [1971]).

El concepto de subsecuencia común más larga analizado en el ejercicio 3.34 ha sido utilizado en el diseño del programa `diff` para comparación de archivos del sistema UNIX (Hunt y McIlroy [1976]). Se describe en Hunt y Szymanski [1977] un algoritmo práctico eficaz para calcular subsecuencias comunes más largas. El algoritmo para calcular las distancias de edición mínimas del ejercicio 3.35 es obra de Wagner y Fischer [1974]. Wagner [1974] incluye una solución al ejercicio 3.36. El trabajo de Sankoff y Kruskal [1983] contiene un estudio fascinante de la gran variedad de aplicaciones de los algoritmos de reconocimiento de distancia mínima, desde el estudio de patrones en secuencias genéticas hasta problemas en el procesamiento del lenguaje hablado.

CAPITULO 4

Análisis sintáctico

Todo lenguaje de programación tiene reglas que prescriben la estructura sintáctica de programas bien formados. En Pascal, por ejemplo, un programa se compone de bloques, un bloque de proposiciones, una proposición de expresiones, una expresión de componentes léxicos, y así sucesivamente. Se puede describir la sintaxis de las construcciones de los lenguajes de programación por medio de gramáticas independientes del contexto o notación BNF (forma de Backus-Naur), que se introdujo en la sección 2.2. Las gramáticas ofrecen ventajas significativas a los diseñadores de lenguajes y a los escritores de compiladores.

- Una gramática da una especificación sintáctica precisa y fácil de entender de un lenguaje de programación.

- A partir de algunas clases de gramáticas se puede construir automáticamente un analizador sintáctico eficiente que determine si un programa fuente está sintácticamente bien formado. Otra ventaja es que el proceso de construcción del analizador sintáctico puede revelar ambigüedades sintácticas y otras construcciones difíciles de analizar que de otro modo podrían pasar sin detectar en la fase inicial de diseño de un lenguaje y de su compilador.

- Una gramática diseñada adecuadamente imparte una estructura a un lenguaje de programación útil para la traducción de programas fuente a código objeto correcto y para la detección de errores. Existen herramientas para convertir descripciones de traducciones basadas en gramáticas en programas operativos.

- Los lenguajes evolucionan con el tiempo, adquiriendo nuevas construcciones y realizando tareas adicionales. Estas nuevas construcciones se pueden añadir con más facilidad a un lenguaje cuando existe una aplicación basada en una descripción gramatical del lenguaje.

La mayor parte de este capítulo está dedicada a los métodos de análisis sintáctico de uso típico en compiladores. Primero se introducen los conceptos básicos, después las técnicas adecuadas para la aplicación manual y, por último, los algoritmos que han sido utilizados en herramientas automatizadas. Como los programas pueden contener errores sintácticos, se amplían los métodos de análisis sintáctico para que se recuperen de los errores de ocurrencia más frecuente.

4.1 EL PAPEL DEL ANALIZADOR SINTACTICO

En este modelo de compilador, el analizador sintáctico obtiene una cadena de componentes léxicos del analizador léxico, como se muestra en la figura 4.1, y comprueba si la cadena pueda ser generada por la gramática del lenguaje fuente. Se supone que el analizador sintáctico informará de cualquier error de sintaxis de manera inteligible. También debería recuperarse de los errores que ocurren frecuentemente para poder continuar procesando el resto de su entrada.

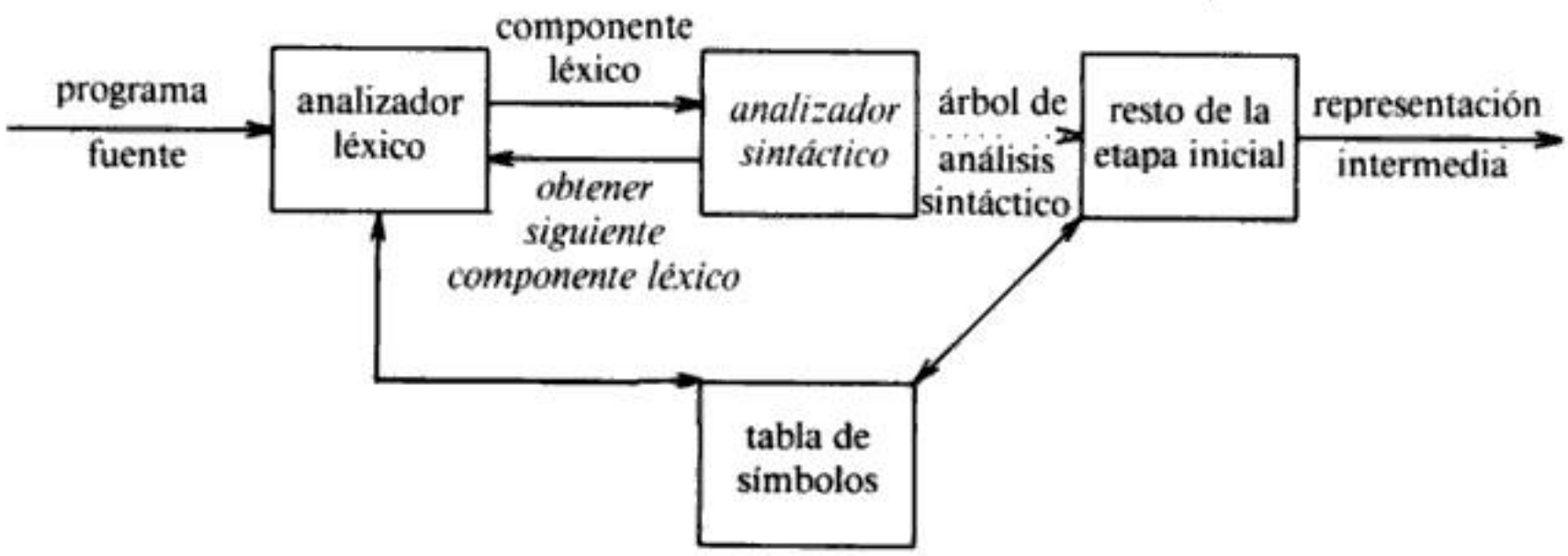


Fig. 4.1. Posición del analizador sintáctico en el modelo del compilador.

Existen tres tipos generales de analizadores sintácticos para gramáticas. Los métodos universales de análisis sintáctico, como el algoritmo de Cocke-Younger-Kasami y el de Earley, pueden analizar cualquier gramática (véanse las notas bibliográficas). Estos métodos, sin embargo, son demasiado ineficientes para usarlos en la producción de compiladores. Los métodos empleados generalmente en los compiladores se clasifican como descendentes o ascendentes. Como sus nombres indican, los analizadores sintácticos descendentes construyen árboles de análisis sintáctico desde arriba (la raíz) hasta abajo (las hojas), mientras que los analizadores sintácticos ascendentes comienzan en las hojas y suben hacia la raíz. En ambos casos, se examina la entrada al analizador sintáctico de izquierda a derecha, un símbolo a la vez.

Los métodos descendentes y ascendentes más eficientes trabajan sólo con subclases de gramáticas, pero varias de estas subclases, como las gramáticas LL y LR, son lo suficientemente expresivas para describir la mayoría de las construcciones sintácticas de los lenguajes de programación. Los analizadores sintácticos implantados a mano a menudo trabajan con gramáticas LL1; por ejemplo, el método de la sección 2.4 construye analizadores sintácticos para gramáticas LL1. Los analizadores sintácticos para la clase más grande de gramáticas LR se construyen normalmente con herramientas automatizadas.

En este capítulo se asume que la salida del analizador sintáctico es una representación del árbol de análisis sintáctico para la cadena de componentes léxicos pro-

ducida por el analizador léxico. En la práctica, hay varias tareas que se pueden realizar durante el análisis sintáctico, como recoger información sobre distintos componentes léxicos en la tabla de símbolos, realizar la verificación de tipo y otras clases de análisis semántico, y generar código intermedio como en el capítulo 2. Se han agrupado todas estas actividades en la casilla de "resto de la etapa inicial" de la figura 4.1 y se analizará con detalle en los tres capítulos siguientes.

En el resto de esta sección, se considera la naturaleza de los errores sintácticos y las estrategias generales para su recuperación. Dos de estas estrategias, llamadas recuperación en modo de pánico y a nivel de frase, se estudian con más detalle junto con los métodos de análisis sintáctico individuales. La implantación de cada estrategia depende del criterio del que escribe el compilador, pero aquí se darán algunas sugerencias respecto al método.

Manejo de errores sintácticos

Si un compilador tuviera que procesar sólo programas correctos, su diseño e implantación se simplificarían mucho. Pero los programadores a menudo escriben programas incorrectos, y un buen compilador debería ayudar al programador a identificar y localizar errores. Es sorprendente que aunque los errores sean tan frecuentes, pocos lenguajes han sido diseñados teniendo en cuenta el manejo de errores. Esta civilización sería completamente distinta si los lenguajes hablados exigieran tanta exactitud sintáctica como los lenguajes de programación. La mayoría de las especificaciones de los lenguajes de programación no describen cómo debe responder un compilador a los errores; la respuesta se deja al diseñador del compilador. Considerar desde el principio el manejo de errores puede simplificar la estructura de un compilador y mejorar su respuesta a los errores.

Se sabe que los programas pueden contener errores de muy diverso tipo. Por ejemplo, los errores pueden ser:

- léxicos, como escribir mal un identificador, palabra clave u operador
- sintácticos, como una expresión aritmética con paréntesis no equilibrados
- semánticos, como un operador aplicado a un operando incompatible
- lógicos, como una llamada infinitamente recursiva

A menudo, gran parte de la detección y recuperación de errores en un compilador se centra en la fase de análisis sintáctico. Una razón es que muchos errores son de naturaleza sintáctica o se manifiestan cuando la cadena de componentes léxicos que proviene del analizador léxico desobedece las reglas gramaticales que definen al lenguaje de programación. Otra razón es la precisión de los métodos modernos de análisis sintácticos, que pueden detectar la presencia de errores dentro de los programas de una forma muy eficiente. La detección exacta de la presencia de errores semánticos y lógicos en el momento de la compilación es mucho más difícil. En esta sección se presentan algunas técnicas básicas para recuperarse de errores sintácticos; en este capítulo se estudia su implantación junto con los métodos de análisis sintáctico.

El manejador de errores en un analizador sintáctico tiene objetivos fáciles de establecer:

- Debe informar de la presencia de errores con claridad y exactitud.
- Se debe recuperar de cada error con la suficiente rapidez como para detectar errores posteriores.
- No debe retrasar de manera significativa el procesamiento de programas correctos.

La realización efectiva de estos objetivos plantea desafíos importantes.

Afortunadamente, los errores más comunes son simples y a menudo basta con un mecanismo sencillo de manejo de errores. Sin embargo, en algunos casos un error pudo haber ocurrido mucho antes de la posición en que se detectó su presencia, y puede ser muy difícil deducir la naturaleza precisa del error. En los casos difíciles, el manejador de errores quizá tenga que adivinar qué tenía en mente el programador cuando escribió el programa.

Varios métodos de análisis sintáctico, como los métodos LL y LR, detectan un error lo antes posible. Es decir, tienen la *propiedad del prefijo viable*, lo cual quiere decir que detectan la presencia de un error nada más ver un prefijo de la entrada que no es prefijo de ninguna cadena del lenguaje.

Ejemplo 4.1. Para comprobar la clase de errores que ocurren en la práctica, se examinan los errores que Ripley y Druseikis [1978] encontraron en una muestra de programas en Pascal realizados por estudiantes.

Descubrieron que los errores no ocurren tan frecuentemente; el 60 por 100 de los programas compilados era correcto sintáctica y semánticamente. Aun cuando había errores, éstos estaban bastante dispersos; el 80 por 100 de las proposiciones con errores sólo tenía un error, y el 13 por 100, dos. Por último, la mayoría eran errores triviales; el 90 por 100 eran errores de un componente léxico.

Muchos de los errores se podrían clasificar simplemente: 60 por 100 eran errores de puntuación, el 20 por 100 eran errores de operador y operando, el 15 por 100, errores de palabras clave, y el restante 5 por 100, de otras clases. La mayoría de los errores de puntuación giraba alrededor del uso incorrecto del punto y coma.

Como ejemplo concreto, considérese el siguiente programa en Pascal.

```
(1)   program impmax(input, output);
(2)   var
(3)       x, y: integer;

(4)   function max(i:integer; j:integer) : integer;
(5)   {devuelve el máximo de los enteros i y j}
(6)   begin
(7)       if i > j then max := i
(8)       else max := j
(9)   end;

(10)  begin
(11)      readln (x,y);
(12)      writeln (max(x,y))
(13)  end.
```

Un error de puntuación frecuente es usar una coma en lugar del punto y coma en la lista de argumentos de una declaración de función (por ejemplo, usar una coma en lugar del primer punto y coma en la línea (4)); otro es no poner un punto y coma obligatorio al final de una línea (por ejemplo, el punto y coma del final de la línea (4)); otro es poner un punto y coma indebido al final de una línea antes de un `else` (por ejemplo, poner un punto y coma al final de la línea (7)).

Tal vez los errores de punto y coma son tan comunes porque el uso del punto y coma varía mucho de un lenguaje a otro. En Pascal, un punto y coma es un separador de proposiciones; en PL/1 y en C, termina una proposición. Algunos estudios han sugerido que este último uso es menos propenso a errores (Gannon y Horning [1975]).

Un ejemplo típico de un error de operador es no poner los dos puntos en `:=`. Los errores de escritura de las palabras clave son bastante raros, pero un ejemplo ilustrativo es olvidar la `i` en `writeln`.

Muchos compiladores de Pascal no tienen dificultades para manejar errores comunes de inserción, borrado y mutación. De hecho, varios compiladores de Pascal compilarán correctamente el programa anterior con un error común de puntuación o de operador; emitirán sólo un diagnóstico de advertencia, señalando la construcción errónea.

Sin embargo, hay otra clase habitual de error mucho más difícil de reparar correctamente: no poner un `begin` o un `end` (por ejemplo, la omisión de la línea (9)). La mayoría de los compiladores no intentaría reparar esta clase de error. □

¿Cómo debe informar un manejador de errores de la presencia de un error? Al menos debe informar del lugar en el programa fuente donde se detecta el error, porque es muy probable que el error real se haya producido en alguno de los componentes léxicos anteriores. Una estrategia común empleada por muchos compiladores es imprimir la línea errónea con un apuntador a la posición donde se detecta el error. Si hay una posibilidad razonable de saber cuál es realmente el error, también se incluye un mensaje de diagnóstico informativo y comprensible; por ejemplo, "falta punto y coma en esta posición".

Una vez detectado el error, ¿cómo se debe recuperar el analizador sintáctico? Como se verá, existen varias estrategias generales, pero ningún método es claramente superior. En la mayoría de los casos, no es adecuado que el analizador sintáctico abandone después de detectar el primer error, porque el posterior procesamiento de la entrada podría revelar más errores. Normalmente, hay alguna forma de recuperación del error donde el analizador sintáctico intenta volver él mismo a un estado en el que el procesamiento de la entrada pueda continuar con una esperanza razonable de que se hará el análisis de la entrada correcta o de que será manejada correctamente por el compilador.

Una recuperación inadecuada puede introducir una avalancha abrumadora de errores "espurios", no cometidos por el programador, sino introducidos por los cambios hechos al estado del analizador sintáctico durante la recuperación del error. De forma similar, la recuperación del error sintáctico puede introducir errores semánticos espurios que más tarde detectarán las fases de análisis semántico o de generación de código. Por ejemplo, al recuperarse de un error, el analizador sintáctico

puede haber omitido una declaración de alguna variable, por ejemplo `zap`. Cuando luego se encuentra `zap` en expresiones, nada es sintácticamente incorrecto, pero como no hay una entrada en la tabla de símbolos para `zap`, se genera el mensaje "`zap` indefinida".

Una estrategia conservadora para un compilador es inhibir los mensajes de error que provengan de errores descubiertos demasiado cerca unos de otros en la cadena de entrada. Después de descubrir un error sintáctico, el compilador podría exigir que varios componentes léxicos se analizarán sintácticamente con éxito antes de permitir otro mensaje de error. En algunos casos, puede haber demasiados errores como para que el compilador continúe un procesamiento razonable. (Por ejemplo, ¿cómo debe responder un compilador de Pascal a un programa en FORTRAN como entrada?) Parece que una estrategia de recuperación de errores tiene que ser un compromiso cuidadosamente considerado, teniendo en cuenta las clases de errores que se pueden presentar y que sean de procesamiento razonable.

Como ya se ha mencionado, algunos compiladores intentan reparar el error, proceso en el cual el compilador intenta adivinar lo que pretendía escribir el programador. El compilador de PL/C (Conway y Wilcox [1973]) es un ejemplo de este tipo de compilador. Excepto tal vez en un entorno de programas cortos escritos por estudiantes inexpertos, no es probable que la recuperación completa de los errores sea rentable. De hecho, con la importancia creciente de la informática interactiva y los buenos entornos de programación, la tendencia parece ser hacia mecanismos sencillos de recuperación de errores.

Estrategias de recuperación de errores

Hay muchas estrategias generales distintas que puede emplear un analizador sintáctico para recuperarse de un error sintáctico. Aunque ninguna de ellas ha demostrado ser de aceptación universal, algunos métodos tienen una amplia aplicabilidad. Aquí se introducen las siguientes estrategias:

- en modo de pánico
- a nivel de frase
- de producciones de error
- de corrección global

Recuperación en modo de pánico. Este es el método más sencillo de implantar y pueden utilizarlo la mayoría de los métodos de análisis sintáctico. Al descubrir un error, el analizador sintáctico desecha símbolos de entrada, de uno en uno, hasta que encuentra uno perteneciente a un conjunto designado de componentes léxicos de sincronización. Estos componentes léxicos de sincronización son generalmente delimitadores, como el punto y coma o la palabra clave **end**, cuyo papel en el programa fuente está claro. Es evidente que quien diseña el compilador debe seleccionar los componentes léxicos de sincronización adecuados para el lenguaje fuente. Aunque la corrección en modo de pánico a menudo omite una cantidad considerable de entrada sin comprobar la existencia de errores adicionales, tiene la ventaja de la sencillez y, a diferencia de otros métodos considerados más adelante, está garantizado contra lazos infinitos. En situaciones en donde son raros los errores múltiples en la misma proposición, este método puede resultar bastante adecuado.

Recuperación a nivel de frase. Al descubrir un error, el analizador sintáctico puede realizar una corrección local de la entrada restante; es decir, puede sustituir un prefijo de la entrada restante por alguna cadena que permita continuar al analizador sintáctico. Una corrección local típica sería sustituir una coma por un punto y coma, suprimir un punto y coma sobrante, o insertar un punto y coma que falta. La elección de la corrección local corresponde al diseñador del compilador. Por supuesto, se debe tener cuidado de elegir sustituciones que no conduzcan a lazos infinitos, como sería el caso, por ejemplo, si siempre se insertara algo en la entrada por delante del símbolo de entrada en curso.

Este tipo de sustitución puede corregir cualquier cadena de entrada y ha sido empleado en varios compiladores que corrigen los errores. El método se usó por primera vez en el análisis sintáctico descendente. Su principal desventaja es su dificultad para afrontar situaciones en que el error real se produjo antes del punto de detección.

Producciones de error. Si se tiene una buena idea de los errores comunes que pueden encontrarse, se puede aumentar la gramática del lenguaje con producciones que generen las construcciones erróneas. Entonces se usa esta gramática aumentada con las producciones de error para construir el analizador sintáctico. Si el analizador sintáctico usa una producción de error, se pueden generar diagnósticos de error apropiados para indicar la construcción errónea reconocida en la entrada.

Corrección global. Idealmente, sería deseable que un compilador hiciera el mínimo de cambios posibles al procesar una cadena de entrada incorrecta. Existen algoritmos para elegir una secuencia mínima de cambios para obtener una corrección global de menor costo. Dada una cadena de entrada incorrecta x y la gramática G, estos algoritmos encontrarán un árbol de análisis sintáctico para una cadena relacionada y, tal que el número de inserciones, supresiones y modificaciones de componentes léxicos necesario para transformar x en y sea el mínimo posible. Por desgracia, la implantación de estos métodos es en general demasiado costosa en términos de tiempo y espacio, así que estas técnicas en la actualidad sólo son de interés teórico.

Se debe señalar que un programa correcto más parecido al original puede no ser lo que el programador tenía en mente. Sin embargo, la noción de corrección de costo mínimo proporciona una escala para evaluar las técnicas de recuperación de errores, y se ha usado para encontrar cadenas de sustitución óptimas para la recuperación a nivel de frase.

4.2 GRAMATICAS INDEPENDIENTES DEL CONTEXTO

Muchas construcciones de los lenguajes de programación tienen una estructura inherentemente recursiva que se puede definir mediante gramáticas independientes del contexto. Por ejemplo, se puede tener una proposición condicional definida por una regla como

> Si S_1 y S_2 son proposiciones y E es una expresión, entonces
>
> "**if** E **then** S_1 **else** S_2" es una proposición. (4.1)

No se puede especificar esta forma de proposición condicional usando la notación de las expresiones regulares; en el capítulo 3 se vio que las expresiones regulares pueden especificar la estructura lexicográfica de los componentes léxicos. Por otro lado, utilizando la variable sintáctica *prop* para denotar la clase de las proposiciones y *expr* para la clase de las expresiones, ya se puede expresar (4.1) usando la producción gramatical

$$prop \rightarrow \textbf{if } expr \textbf{ then } prop \textbf{ else } prop \tag{4.2}$$

En esta sección se revisa la definición de una gramática independiente del contexto y se introduce terminología para el análisis sintáctico. Según la sección 2.2, una gramática independiente del contexto (gramática, por brevedad) consta de terminales, no terminales, un símbolo inicial y producciones.

1. Los terminales son los símbolos básicos con que se forman las cadenas. "Componente léxico" es un sinónimo de "terminal" cuando se trata de gramáticas para lenguajes de programación. En (4.2), cada una de las palabras clave **if**, **then** y **else** es un terminal.

2. Los no terminales son variables sintácticas que denotan conjuntos de cadenas. En (4.2), *prop* y *expr* son no terminales. Los no terminales definen conjuntos de cadenas que ayudan a definir el lenguaje generado por la gramática. También imponen una estructura jerárquica sobre el lenguaje que es útil tanto para el análisis sintáctico como para la traducción.

3. En una gramática, un no terminal es considerado como el símbolo inicial, y el conjunto de cadenas que representa es el lenguaje definido por la gramática.

4. Las producciones de una gramática especifican cómo se pueden combinar los terminales y los no terminales para formar cadenas. Cada producción consta de un no terminal, seguido por una flecha (a veces se usa el símbolo ::=, en lugar de la flecha), seguida por una cadena de no terminales y terminales.

Ejemplo 4.2. La gramática con las siguientes producciones define expresiones aritméticas simples.

$$\begin{aligned}
expr &\rightarrow expr\ op\ expr \\
expr &\rightarrow (\ expr\ ) \\
expr &\rightarrow -\ expr \\
expr &\rightarrow \textbf{id} \\
op &\rightarrow + \\
op &\rightarrow - \\
op &\rightarrow * \\
op &\rightarrow / \\
op &\rightarrow \uparrow
\end{aligned}$$

En esta gramática, los símbolos terminales son

$$\textbf{id}\ +\ -\ *\ /\ \uparrow\ (\ )$$

Los símbolos no terminales son *expr* y *op*, y *expr* es el símbolo inicial. □

Convenciones de notación

Para evitar tener que establecer siempre que "estos son los terminales", "estos son los no terminales", etcétera, a partir de ahora se emplearán las siguientes convenciones de notación con respecto a las gramáticas.

1. Estos símbolos son terminales:
   a) Las primeras letras minúsculas del alfabeto, como *a*, *b*, *c*.
   b) Los símbolos de operador, como +, −, etcétera.
   c) Los símbolos de puntuación, como paréntesis, coma, etcétera.
   d) Los dígitos 0, 1, . . . , 9.
   e) Cadenas en **negritas**, como **id** o **if**.
2. Estos símbolos son no terminales:
   a) Las primeras letras mayúsculas del alfabeto, como *A*, *B*, *C*.
   b) La letra *S*, que cuando aparece suele ser el símbolo inicial.
   c) Los nombres en cursivas minúsculas, como *expr* o *prop*.
3. Las últimas letras mayúsculas del alfabeto, como *X*, *Y*, *Z*, representan *símbolos gramaticales*, es decir, terminales o no terminales.
4. Las últimas letras minúsculas del alfabeto, principalmente *u*, *v*, . . . , *z*, representan cadenas de terminales.
5. Las letras griegas minúsculas, α, β, γ, por ejemplo, representan cadenas de símbolos gramaticales. Por tanto, una producción genérica podría escribirse $A \rightarrow \alpha$, indicando que hay un solo no terminal A a la izquierda de la flecha (el *lado izquierdo* de la producción) y una cadena de símbolos gramaticales α a la derecha de la flecha (el *lado derecho* de la producción).
6. Si $A \rightarrow \alpha_1, A \rightarrow \alpha_2, \ldots, A \rightarrow \alpha_k$ son todas producciones con A a la izquierda (se les llama *producciones de A*), se puede escribir $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_k$. $\alpha_1, \alpha_2, \ldots, \alpha_k$ se denominan *alternativas* de A.
7. A menos que se diga otra cosa, el lado izquierdo de la primera producción es el símbolo inicial.

Ejemplo 4.3. Usando estas abreviaturas, se podría escribir en forma concisa la gramática del ejemplo 4.2 como

$$E \rightarrow E\ A\ E \mid (\ E\ ) \mid -\ E \mid \mathbf{id}$$
$$A \rightarrow + \mid - \mid * \mid / \mid \uparrow$$

Las convenciones de notación indican que E y A son no terminales, con E como símbolo inicial. El resto de los símbolos son terminales. □

Derivaciones

Hay varias formas de considerar el proceso mediante el cual una gramática define un lenguaje. En la sección 2.2 se consideró este proceso como el de construcción de árboles de análisis sintáctico, pero existe también una visión derivativa relacionada que suele resultar útil. De hecho, esta visión derivativa da una descripción precisa

de la construcción descendente de un árbol de análisis sintáctico. La idea central es que se considera una producción como una regla de reescritura, donde el no terminal de la izquierda es sustituido por la cadena del lado derecho de la producción.

Por ejemplo, considérese la siguiente gramática para expresiones aritméticas, donde el no terminal E representa una expresión.

$$E \rightarrow E + E \mid E * E \mid (E) \mid - E \mid \mathbf{id} \tag{4.3}$$

La producción $E \rightarrow - E$ significa que una expresión precedida por un signo menos es también una expresión. Esta producción se puede usar para generar expresiones más complejas a partir de expresiones más simples permitiendo sustituir cualquier presencia de E por $- E$. En el caso más simple, se puede sustituir una sola E por $- E$. Se puede describir esta acción escribiendo

$$E \Rightarrow - E$$

que se lee "E deriva $-E$". La producción $E \rightarrow (E)$ establece que también se podría sustituir una presencia de una E en cualquier cadena de símbolos gramaticales por (E); por ejemplo, $E*E \Rightarrow (E)*E$ o $E*E \Rightarrow E*(E)$.

Se puede tomar una sola E y aplicar repetidamente producciones en cualquier orden para obtener una secuencia de sustituciones. Por ejemplo,

$$E \Rightarrow - E \Rightarrow - (E) \Rightarrow -(\mathbf{id})$$

A dicha secuencia de sustituciones se le llama *derivación* de $-(\mathbf{id})$ a partir de E. Esta derivación proporciona una prueba de que un caso determinado de una expresión es la cadena $-(\mathbf{id})$.

De forma más abstracta, se dice que $\alpha A \beta \Rightarrow \alpha\gamma\beta$ si $A \rightarrow \gamma$ es una producción y α y β son cadenas arbitrarias de símbolos gramaticales. Si $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n$, se dice que α_1 *deriva* a α_n. El símbolo $\Rightarrow$ significa "deriva en un paso". A menudo se desea decir "deriva en cero o más pasos". Para este propósito se puede usar el símbolo $\overset{*}{\Rightarrow}$. Así:

1. $\alpha \overset{*}{\Rightarrow} \alpha$ para cualquier cadena α, y
2. Si $\alpha \overset{*}{\Rightarrow} \beta$ y $\beta \Rightarrow \gamma$, entonces $\alpha \overset{*}{\Rightarrow} \gamma$.

Del mismo modo se puede usar $\overset{+}{\Rightarrow}$ para expresar "deriva en uno o más pasos".

Dada una gramática G con símbolo inicial S, se puede utilizar la relación $\overset{+}{\Rightarrow}$ para definir $L(G)$, el *lenguaje generado por G*. Las cadenas de $L(G)$ pueden contener sólo símbolos terminales de G. Se dice que una cadena de terminales w está en $L(G)$ si, y sólo si, $S \overset{+}{\Rightarrow} w$. A la cadena w se le llama *frase* de G. De un lenguaje que pueda ser generado por una gramática se dice que es un *lenguaje independiente del contexto*. Si dos gramáticas generan el mismo lenguaje, se dice que son *equivalentes*.

Si $S \overset{*}{\Rightarrow} \alpha$, donde α puede contener no terminales, entonces se dice que α es una *forma de frase* de G. Una frase es una forma de frase sin no terminales.

Ejemplo 4.4. La cadena $-(\mathbf{id}+\mathbf{id})$ es una frase de la gramática (4.3), porque existe la derivación

$$E \Rightarrow - E \Rightarrow - (E) \Rightarrow - (E+E) \Rightarrow - (\mathbf{id}+E) \Rightarrow - (\mathbf{id}+\mathbf{id}) \tag{4.4}$$

Las cadenas E, $-E$, $-(E)$, . . . , $-(\mathbf{id}+\mathbf{id})$ que aparecen en esta derivación son todas formas de frases de esta gramática. Se escribe $E \overset{*}{\Rightarrow} -(\mathbf{id}+\mathbf{id})$ para indicar que $-(\mathbf{id}+\mathbf{id})$ se puede derivar de E.

Se puede demostrar por inducción sobre la longitud de una derivación que toda frase del lenguaje de la gramática (4.3) es una expresión aritmética que comprende a los operadores binarios $+$ y $*$, al operador unario $-$, paréntesis y al operando **id**. De manera similar, se puede demostrar por inducción sobre la longitud de una expresión aritmética que todas estas expresiones pueden ser generadas por esta gramática. Así, la gramática (4.3) genera precisamente el conjunto de todas las expresiones aritméticas que comprenden a los binarios $+$ y $*$, al $-$ unario, a los paréntesis y al operando **id**. □

En cada paso de una derivación hay que hacer dos elecciones. Es necesario escoger qué no terminal se debe sustituir y, una vez hecha esta elección, qué alternativa usar para este no terminal. Por ejemplo, la derivación (4.4) del ejemplo 4.4 podría continuar desde $-(E+E)$ como sigue

$$-(E+E) \Rightarrow -(E+\mathbf{id}) \Rightarrow -(\mathbf{id}+\mathbf{id}) \tag{4.5}$$

Se sustituye cada no terminal de (4.5) por el mismo lado derecho que en el ejemplo 4.4, pero el orden de sustitución es diferente.

Para comprender cómo trabajan algunos analizadores sintácticos, hay que considerar derivaciones donde tan sólo el no terminal de más a la izquierda de cualquier forma de frase se sustituya a cada paso. Dichas derivaciones se denominan *por la izquierda*. Si $\alpha \Rightarrow \beta$ mediante un paso en el que se sustituye el no terminal más a la izquierda de α, se escribe $\alpha \underset{mi}{\Rightarrow} \beta$. Puesto que la derivación (4.4) es por la izquierda, se puede escribir así:

$$E \underset{mi}{\Rightarrow} -E \underset{mi}{\Rightarrow} -(E) \underset{mi}{\Rightarrow} -(E+E) \underset{mi}{\Rightarrow} -(\mathbf{id}+E) \underset{mi}{\Rightarrow} -(\mathbf{id}+\mathbf{id})$$

Usando las convenciones de notación, todo paso por la izquierda se puede escribir $wA\gamma \underset{mi}{\Rightarrow} w\delta\gamma$, donde w consta sólo de terminales, $A \rightarrow \delta$ es la producción aplicada y γ es una cadena de símbolos gramaticales. Para subrayar el hecho de que α deriva a β por medio de una derivación por la izquierda, se escribe $\alpha \overset{*}{\underset{mi}{\Rightarrow}} \beta$. Si $S \overset{*}{\underset{mi}{\Rightarrow}} \alpha$, entonces se dice que α es una *forma de frase izquierda* de la gramática en cuestión.

Análogas definiciones se aplican a las derivaciones *derechas*, donde el no terminal más a la derecha se sustituye en cada paso. Las derivaciones derechas a menudo se denominan derivaciones *canónicas*.

Arboles de análisis sintáctico y derivaciones

Un árbol de análisis sintáctico se puede considerar como una representación gráfica de una derivación que no muestra la elección relativa al orden de sustitución. Como se vio en la sección 2.2, cada nodo interior de un árbol de análisis sintáctico se etiqueta con algún no terminal A, y que los hijos de ese nodo se etiquetan, de izquierda a derecha, con los símbolos del lado derecho de la producción por la cual se sustituyó esta A en la derivación. Las hojas del árbol de análisis sintáctico se etiquetan

con terminales o no terminales y, leídas de izquierda a derecha, constituyen una forma de frase, llamada el producto o frontera del árbol. Por ejemplo, en la figura 4.2 se muestra el árbol de análisis sintáctico para $-(\mathbf{id}+\mathbf{id})$ indicado por la derivación (4.4).

Para ver la relación entre derivaciones y árboles de análisis sintáctico, considérese cualquier derivación $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n$, donde α_1 es un solo no terminal A. Para cada forma de frase α_i de la derivación, se construye un árbol de análisis sintáctico cuyo producto es α_i. El proceso es una inducción sobre i. Para la base, el árbol para $\alpha_1 = A$ es un solo nodo etiquetado con A. Para hacer la inducción, supóngase que ya se ha construido un árbol de análisis sintáctico cuyo producto es $\alpha_{i-1} = X_1 X_2 \ldots X_k$. (Recordando las convenciones, cada X_i es o un terminal o un no terminal.) Supóngase que α_i se deriva de α_{i-1} sustituyendo X_j, que es un no terminal, por $\beta = Y_1 Y_2 \ldots Y_r$. Es decir, en el i-ésimo paso de la derivación, la producción $X_j \to \beta$ se aplica a α_{i-1} para derivar $\alpha_i = X_1 X_2 \ldots X_{j-1} \beta X_{j+1} \ldots X_k$.

Para modelar este paso de la derivación, hay que encontrar la j-ésima hoja por la izquierda en el árbol de análisis sintáctico en curso. Esta hoja está etiquetada con X_j. A esta hoja se le asignan r hijos, etiquetados con $Y_1, Y_2, \ldots, Y_r$ desde la izquierda. Como caso especial, si $r = 0$, es decir, $\beta = \epsilon$, entonces a la j-ésima hoja se le asigna un hijo etiquetado con ϵ.

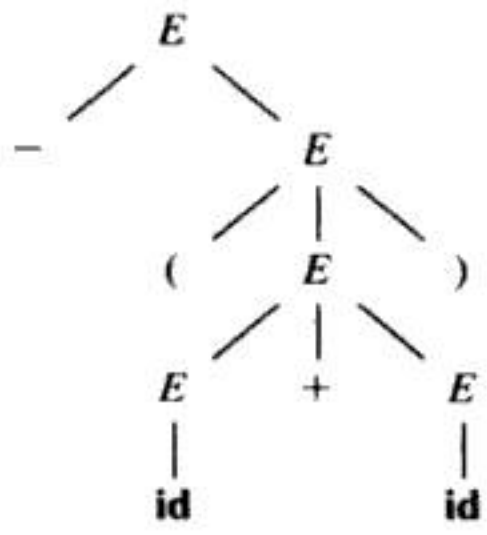


Fig. 4.2. Arbol de análisis sintáctico para $-(\mathbf{id} + \mathbf{id})$.

Ejemplo 4.5. Considérese la derivación (4.4). La secuencia de árboles de análisis sintáctico construidos a partir de esta derivación se muestra en la figura 4.3. En el primer paso de la derivación, $E \Rightarrow -E$. Para modelar este paso, se añaden dos hijos, etiquetados con $-$ y con E, a la raíz E del árbol inicial para crear el segundo árbol.

En el segundo paso de la derivación, $-E \Rightarrow -(E)$. En consecuencia, se añaden tres hijos, etiquetados con (, E y), a la hoja etiquetada con E del segundo árbol para obtener el tercer árbol con producto $-(E)$. Si se continúa así se obtiene el árbol de análisis sintáctico completo como sexto árbol. □

Como ya se ha mencionado, un árbol de análisis sintáctico ignora las variaciones en el orden en que se sustituyen los símbolos en las formas de frases. Por ejemplo, si se continuara la derivación (4.4) como en la línea (4.5), daría como resultado el

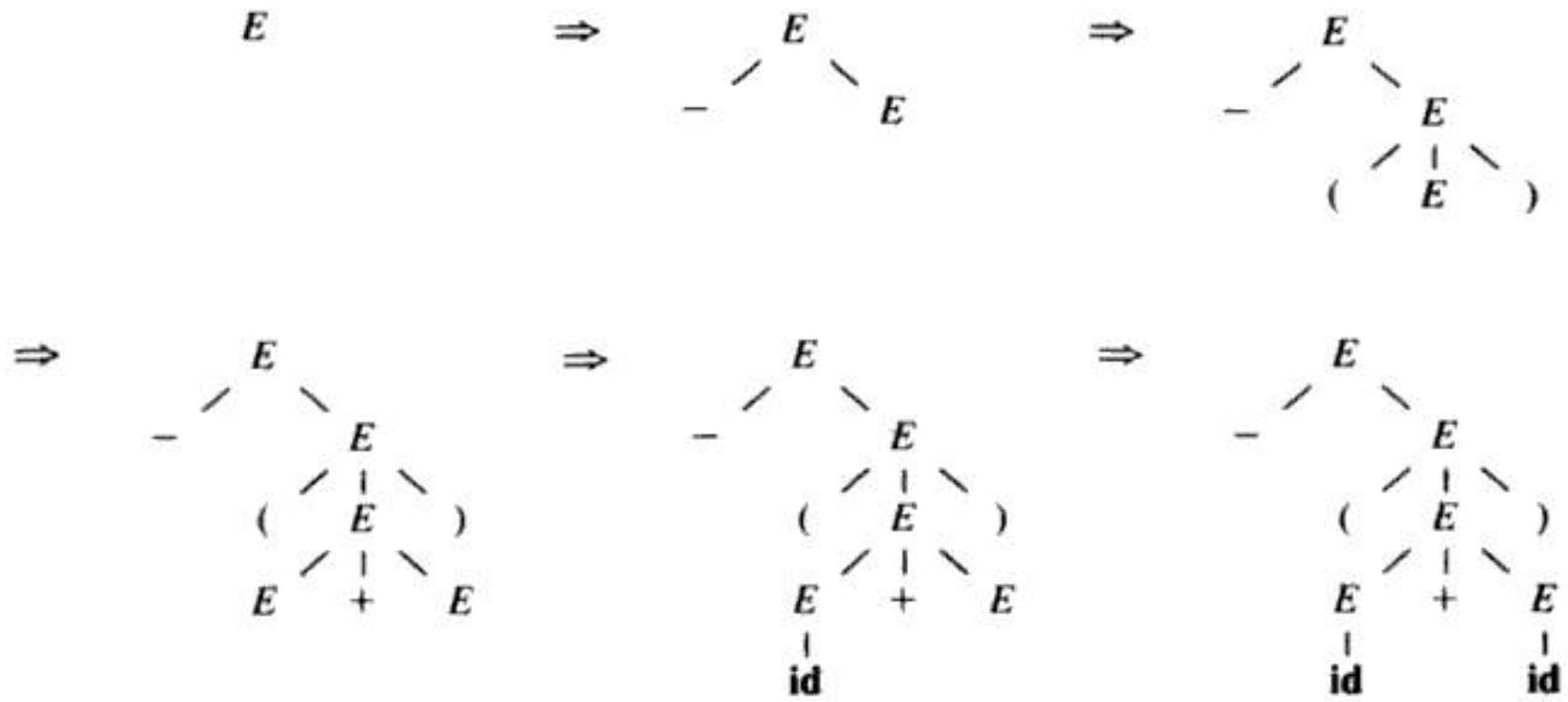


Fig. 4.3. Construcción del árbol de análisis sintáctico a partir de la derivación (4.4).

mismo árbol de análisis sintáctico final de la figura 4.3. También se pueden eliminar estas variaciones en el orden en que se aplican las producciones si se consideran únicamente derivaciones por la izquierda (o por la derecha). No es difícil ver que todo árbol de análisis sintáctico tiene asociado una única derivación izquierda y una única derivación derecha. En lo que sigue, a menudo se hará el análisis sintáctico produciendo una derivación por la izquierda o por la derecha, sabiendo que en lugar de esta derivación se podría producir el propio árbol de análisis sintáctico. Sin embargo, no se debe suponer que toda frase tiene obligatoriamente un solo árbol de análisis sintáctico o una sola derivación por la izquierda o por la derecha.

Ejemplo 4.6. Se considera de nuevo la gramática (4.3) de expresiones aritméticas. La frase **id** + **id*****id** tiene las dos claras derivaciones por la izquierda:

$$
\begin{aligned}
E &\Rightarrow E+E \\
&\Rightarrow \mathbf{id}+E \\
&\Rightarrow \mathbf{id}+E*E \\
&\Rightarrow \mathbf{id}+\mathbf{id}*E \\
&\Rightarrow \mathbf{id}+\mathbf{id}*\mathbf{id}
\end{aligned}
\qquad\qquad
\begin{aligned}
E &\Rightarrow E*E \\
&\Rightarrow E+E*E \\
&\Rightarrow \mathbf{id}+E*E \\
&\Rightarrow \mathbf{id}+\mathbf{id}*E \\
&\Rightarrow \mathbf{id}+\mathbf{id}*\mathbf{id}
\end{aligned}
$$

con los dos árboles de análisis sintáctico correspondientes que se muestran en la figura 4.4. □

Obsérvese que el árbol de análisis sintáctico de la figura 4.4(a) refleja la precedencia comúnmente aceptada de + y *, mientras que el árbol de la figura 4.4(b) no. Es decir, es habitual considerar que el operador * tiene mayor precedencia que +, lo cual corresponde al hecho de que una expresión como $a+b*c$ normalmente se evaluaría como $a+(b*c)$, en lugar de como $(a+b)*c$.

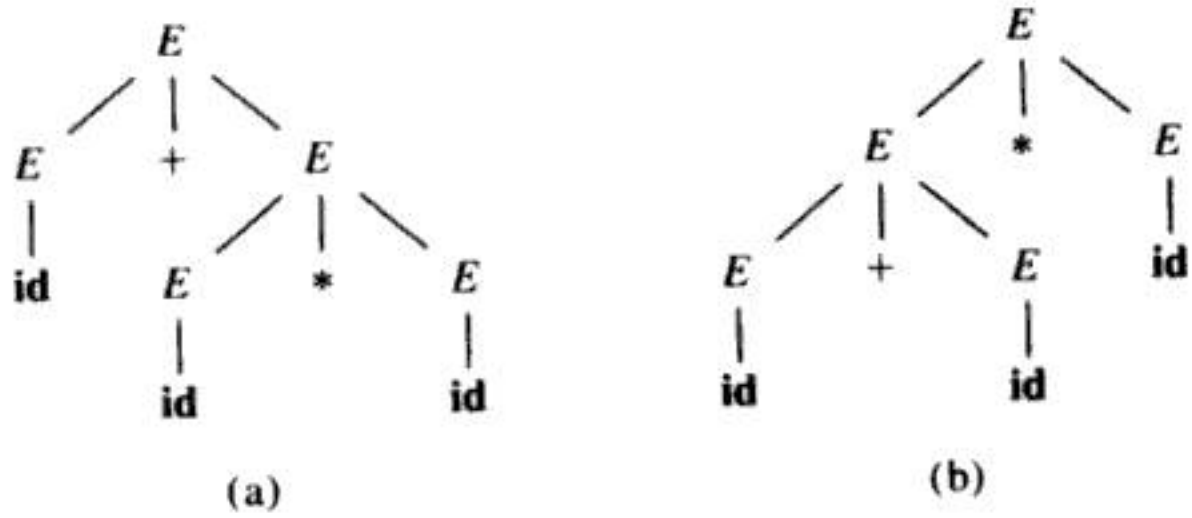


Fig. 4.4. Dos árboles de análisis sintáctico para **id + id*id**.

Ambigüedad

Se dice que una gramática que produce más de un árbol de análisis sintáctico para alguna frase es *ambigua*. O, dicho de otro modo, una gramática ambigua es la que produce más de una derivación por la izquierda o por la derecha para la misma frase. Para algunos tipos de analizadores sintácticos es preferible que la gramática no sea ambigua, pues si lo fuera, no se podría determinar de manera exclusiva qué árbol de análisis sintáctico seleccionar para una frase. Para algunas aplicaciones se considerarán también los métodos mediante los cuales se puedan utilizar ciertas gramáticas ambiguas, junto con *reglas para eliminar ambigüedades* que desechan árboles de análisis sintáctico indeseables, dejando sólo un árbol para cada frase.

4.3 ESCRITURA DE UNA GRAMATICA

Las gramáticas son capaces de describir la mayoría, pero no todas, de las sintaxis de los lenguajes de programación. Un analizador léxico efectúa una cantidad limitada de análisis sintáctico conforme produce la secuencia de componentes léxicos a partir de los caracteres de entrada. Ciertas limitaciones de la entrada, como el requisito de que los identificadores se declaren antes de ser utilizados, no pueden describirse mediante una gramática independiente del contexto. Por tanto, las secuencias de componentes léxicos aceptadas por un analizador sintáctico forman un superconjunto de un lenguaje de programación; las fases posteriores deben analizar la salida del analizador sintáctico para garantizar la obediencia a reglas que el analizador sintáctico no comprueba. (Véase Cap. 6.)

Esta sección se inicia considerando la división del trabajo entre un analizador léxico y un analizador sintáctico. Puesto que cada método de análisis sintáctico puede manejar sólo gramáticas de una cierta forma, quizá se deba reescribir la gramática inicial para hacerla analizable por el método elegido. A menudo se pueden construir gramáticas adecuadas para expresiones utilizando la información acerca de la asociatividad y la precedencia, como en la sección 2.2. Aquí, se consideran transformaciones útiles para reescribir gramáticas y hacerlas apropiadas para el análisis sintáctico descendente. Esta sección concluye considerando algunas construcciones de los lenguajes de programación que no pueden ser descritas por ninguna gramática.

Expresiones regulares, o gramáticas independientes del contexto

Toda construcción que se pueda describir mediante una expresión regular también se puede describir por medio de una gramática. Por ejemplo, la expresión regular $(a|b)*abb$ y la gramática

$$A_0 \rightarrow aA_0 \mid bA_0 \mid aA_1$$
$$A_1 \rightarrow bA_2$$
$$A_2 \rightarrow bA_3$$
$$A_3 \rightarrow \epsilon$$

describen el mismo lenguaje, el conjunto de cadenas de caracteres a y b que terminan en abb.

Se puede convertir de manera mecánica un autómata finito no determinista (AFN) en una gramática que genere el mismo lenguaje reconocido por el AFN. La gramática anterior se construyó a partir del AFN de la figura 3.23 utilizando la siguiente construcción: para cada estado i del AFN, créese un símbolo no terminal A_i. Si el estado i tiene una transición al estado j con el símbolo de entrada a, introdúzcase la producción $A_i \rightarrow aA_j$. Si el estado i va al estado j con la entrada ϵ, introdúzcase la producción $A_i \rightarrow A_j$. Si i es un estado de aceptación, introdúzcase $A_i \rightarrow \epsilon$. Si i es el estado de inicio, hacer de A_i el símbolo inicial de la gramática.

Puesto que todo conjunto regular es un lenguaje independiente del contexto, es razonable preguntar: ¿Por qué utilizar expresiones regulares para definir la sintaxis lexicográfica de un lenguaje? Existen varias razones.

1. Las reglas lexicográficas de un lenguaje a menudo son bastante sencillas, y para describirlas no se necesita una notación tan poderosa como las gramáticas.
2. Las expresiones regulares por lo general proporcionan una notación más concisa y fácil de entender para los componentes léxicos que una gramática.
3. Se pueden construir automáticamente analizadores léxicos más eficientes a partir de expresiones regulares que de gramáticas arbitrarias.
4. Separar la estructura sintáctica de un lenguaje en partes léxicas y no léxicas proporciona una forma conveniente de modularizar la etapa inicial de un compilador en dos componentes de tamaño razonable.

No existen normas fijas en cuanto a qué poner en las reglas lexicográficas, en vez de las reglas sintácticas. Las expresiones regulares son muy útiles para describir la estructura de las construcciones léxicas, como identificadores, constantes, palabras clave, etcétera. Las gramáticas, por otra parte, son muy útiles para describir estructuras anidadas, como paréntesis equilibrados, concordancia de las palabras clave **begin** y **end**, los correspondientes **if-then-else**, etcétera. Como ya se ha señalado, estas estructuras anidadas no se pueden describir con expresiones regulares.

Comprobación del lenguaje generado por una gramática

Aunque los diseñadores de compiladores rara vez lo hacen para una gramática completa de un lenguaje de programación, es importante ser capaz de razonar que un conjunto dado de producciones genera un lenguaje determinado. Las construccio-

nes problemáticas se pueden estudiar escribiendo una gramática abstracta concisa y estudiando el lenguaje que genera. Más adelante se construirá una de estas gramáticas para los condicionales.

Una prueba de que una gramática G genera un lenguaje L tiene dos partes: se debe demostrar que toda cadena generada por G está en L, y lo opuesto, que toda cadena de L puede de hecho ser generada por G.

Ejemplo 4.7. Considérese la gramática (4.6)

$$S \rightarrow (S)S \mid \epsilon \tag{4.6}$$

Al principio puede no resultar evidente, pero esta simple gramática genera todas las cadenas de paréntesis equilibrados y sólo estas cadenas. Para comprobarlo, primero se demostrará que toda frase derivable de S está equilibrada, y después, que toda cadena equilibrada es derivable de S. Para demostrar que toda frase derivable de S está equilibrada, se usa una prueba inductiva sobre el número de pasos de una derivación. Para el paso base, se observa que la única cadena de terminales derivable de S en un paso es la cadena vacía, que sin duda está equilibrada.

Ahora, supóngase que todas las derivaciones de menos de n pasos producen frases equilibradas y considérese una derivación por la izquierda de exactamente n pasos. Dicha derivación debe tener la forma

$$S \Rightarrow (S)S \overset{*}{\Rightarrow} (x)S \overset{*}{\Rightarrow} (x)y$$

Las derivaciones de x e y a partir de S ocupan menos de n pasos, de modo que, por la hipótesis de inducción, x e y están equilibradas. Por tanto, la cadena $(x)y$ debe estar equilibrada.

Se ha demostrado así que cualquier cadena derivable de S está equilibrada. A continuación se debe demostrar que toda cadena equilibrada es derivable de S. Para hacerlo, se utiliza inducción sobre la longitud de una cadena. Para el paso base, la cadena vacía es derivable de S.

Ahora, supóngase que toda cadena equilibrada de longitud menor que $2n$ es derivable de S y considérese una cadena equilibrada w de longitud $2n$, $n \geqslant 1$. Sin duda, w comienza con un paréntesis izquierdo. Sea (x) el prefijo más corto de w que tenga un número igual de paréntesis izquierdos y derechos. Entonces w se puede escribir $(x)y$, donde tanto x como y están equilibrados. Puesto que x e y son de longitud menor que $2n$, son derivables de S por la hipótesis de inducción. Por tanto, se puede encontrar una derivación de la forma

$$S \Rightarrow (S)S \overset{*}{\Rightarrow} (x)S \overset{*}{\Rightarrow} (x)y$$

lo cual demuestra que $w = (x)y$ también es derivable de S. □

Supresión de la ambigüedad

A veces, una gramática ambigua se puede reescribir para eliminar la ambigüedad. Como ejemplo, se eliminará la ambigüedad de la siguiente gramática con "**else** ambiguo":

$$
\begin{array}{lll}
\textit{prop} \rightarrow & \textbf{if } \textit{expr} \textbf{ then } \textit{prop} & \\
 & \textbf{if } \textit{expr} \textbf{ then } \textit{prop} \textbf{ else } \textit{prop} & (4.7) \\
 & \textbf{otra} &
\end{array}
$$

Aquí, **otra** representa cualquier otra proposición. De acuerdo con esta gramática, la proposición condicional compuesta

if E_1 **then** S_1 **else if** E_2 **then** S_2 **else** S_3

tiene el árbol de análisis sintáctico que se muestra en la figura 4.5. La gramática (4.7) es ambigua, puesto que la cadena

if E_1 **then if** E_1 **then** S_1 **else** S_2 (4.8)

tiene los dos árboles de análisis sintáctico que se muestran en la figura 4.6.

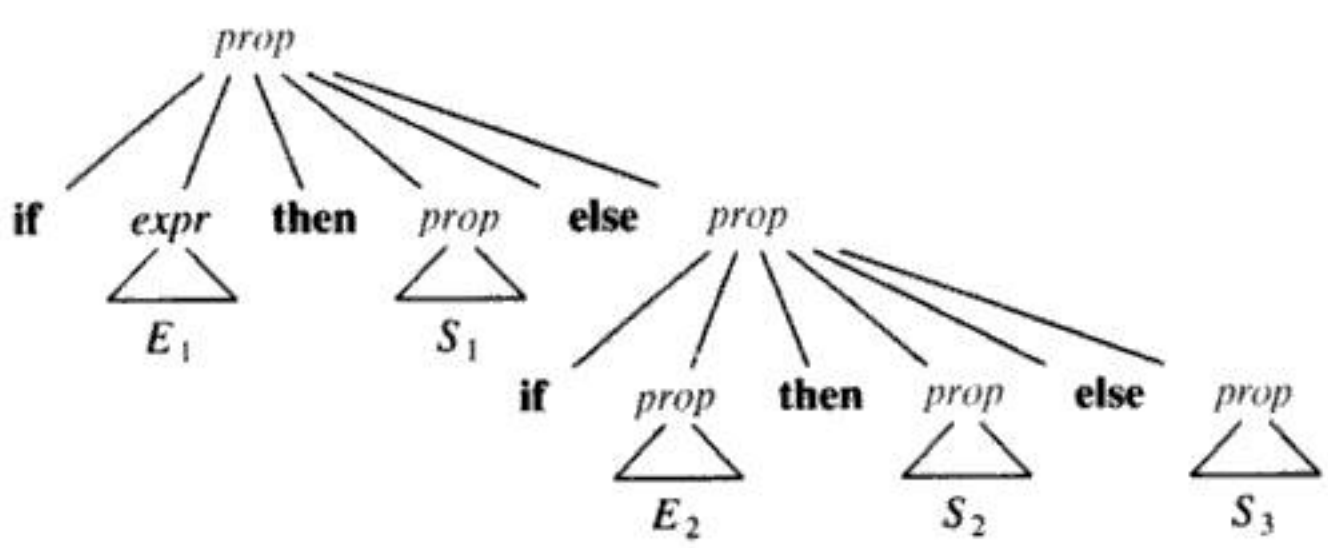


Fig. 4.5. Arbol de análisis sintáctico para la proposición condicional.

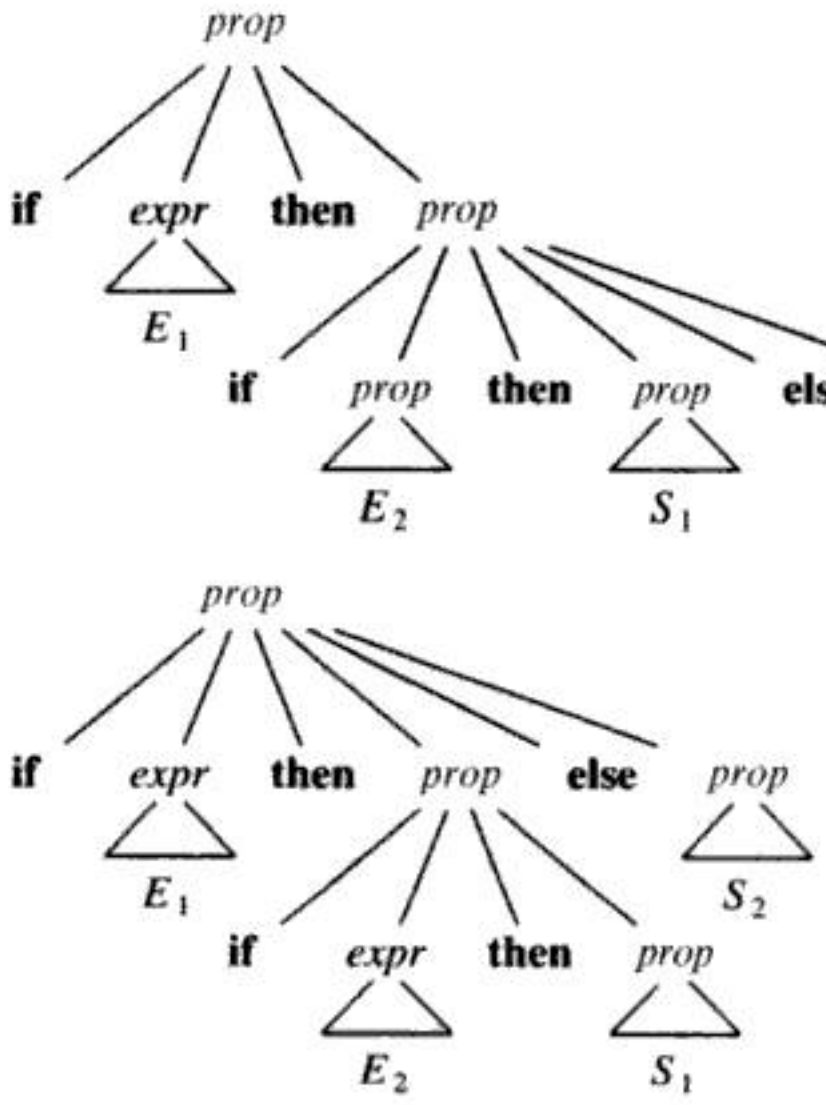


Fig. 4.6. Dos árboles de análisis sintáctico para una frase ambigua.

En todos los lenguajes de programación con proposiciones condicionales de esta forma, se prefiere el primer árbol de análisis sintáctico. La regla general es, "emparejar cada **else** con el **then** sin emparejar anterior más cercano". Esta regla para eliminar ambigüedades se puede incorporar directamente a la gramática. Por ejemplo, se puede reescribir la gramática (4.7) como la siguiente gramática no ambigua. La idea es que una proposición que aparezca entre un **then** y un **else** debe estar "emparejada"; es decir, no debe terminar con un **then** sin emparejar seguido de cualquier proposición, porque entonces el **else** estaría obligado a concordar con este **then** no emparejado. Una proposición emparejada es o una proposición **if-then-else** que no contenga proposiciones sin emparejar o cualquier otra clase de proposición no condicional. Así, se puede utilizar la gramática

$$
\begin{aligned}
\textit{prop} &\rightarrow \textit{prop\_emparejada} \\
&\mid \textit{pro\_no\_emparejada} \\
\textit{prop\_emparejada} &\rightarrow \textbf{if } \textit{expr} \textbf{ then } \textit{prop\_emparejada} \textbf{ else } \textit{prop\_emparejada} \\
&\mid \textbf{otra} \\
\textit{prop\_no\_emparejada} &\rightarrow \textbf{if } \textit{expr} \textbf{ then } \textit{prop} \\
&\mid \textbf{if } \textit{expr} \textbf{ then } \textit{prop\_emparejada} \textbf{ else } \textit{prop\_no\_emparejada}
\end{aligned}
\tag{4.9}
$$

Esta gramática genera el mismo conjunto de cadenas que (4.7), pero permite sólo un análisis sintáctico para la cadena (4.8), es decir, el que asocia cada **else** con el **then** sin emparejar anterior más cercano.

Eliminación de la recursión por la izquierda

Una gramática es *recursiva por la izquierda* si tiene un no terminal A tal que existe una derivación $A \overset{+}{\Rightarrow} A\alpha$ para alguna cadena α. Los métodos de análisis sintáctico descendente no pueden manejar gramáticas recursivas por la izquierda, así que se necesita una transformación que elimine la recursión por la izquierda. En la sección 2.4 se analizó la recursión simple por la izquierda, donde había una producción de la forma $A \rightarrow A\alpha$. Aquí se estudia el caso general. En la sección 2.4 se demostró que el par de producciones recursivas por la izquierda $A \rightarrow A\alpha \mid \beta$ podían sustituirse por las producciones no recursivas por la izquierda

$$
\begin{aligned}
A &\rightarrow \beta A' \\
A' &\rightarrow \alpha A' \mid \epsilon
\end{aligned}
$$

sin modificar el conjunto de cadenas derivables de A. Esta regla ya es suficiente en muchas gramáticas.

Ejemplo 4.8. Considérese la siguiente gramática para expresiones aritméticas.

$$
\begin{aligned}
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (E) \mid \textbf{id}
\end{aligned}
\tag{4.10}
$$

Eliminando la *recursión directa por la izquierda* (producciones de la forma $A \rightarrow A\alpha$) a las producciones de E y después a las de T, se obtiene

$$
\begin{aligned}
E &\to TE' \\
E' &\to +TE' \mid \epsilon \\
T &\to FT' \\
T' &\to *FT' \mid \epsilon \\
F &\to (E) \mid \mathbf{id}
\end{aligned}
\tag{4.11}
$$

□

Independientemente de cuántas producciones de A existan, se puede eliminar de ellas la recursión directa por la izquierda mediante la siguiente técnica. Primero se agrupan las producciones de A en la forma

$$A \to A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

donde ninguna β_i comienza con una A. Después se sustituyen las producciones de A por

$$
\begin{aligned}
A &\to \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A' \\
A' &\to \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \epsilon
\end{aligned}
$$

El no terminal A genera las mismas cadenas que antes, pero ya no es recursivo por la izquierda. Este procedimiento elimina toda la recursión inmediata por la izquierda de las producciones A y A' (suponiendo que ningún α_i es ϵ), pero no elimina la recursión por la izquierda que incluya derivaciones de dos o más pasos. Por ejemplo, considérese la gramática

$$
\begin{aligned}
S &\to Aa \mid b \\
A &\to Ac \mid Sd \mid \epsilon
\end{aligned}
\tag{4.12}
$$

El no terminal S es recursivo por la izquierda, porque $S \Rightarrow Aa \Rightarrow Sda$, pero no es recursivo inmediato por la izquierda.

El algoritmo 4.1, eliminará sistemáticamente la recursión por la izquierda de una gramática. Siempre funciona si la gramática no tiene ciclos (derivaciones de la forma $A \overset{+}{\Rightarrow} A$) o producciones ϵ (producciones de la forma $A \to \epsilon$). Los ciclos, al igual que las producciones ϵ, se pueden eliminar sistemáticamente de una gramática (véanse Ejercicios 4.20 y 4.22).

Algoritmo 4.1. Eliminación de la recursión por la izquierda.

Entrada. La gramática G sin ciclos ni producciones ϵ.

Salida. Una gramática equivalente sin recursión por la izquierda.

1. Ordénense los no terminales en un orden $A_1, A_2, \ldots, A_n$.
2. **for** $i := 1$ **to** n **do**
   begin
   for $j := 1$ **to** $i-1$ **do**
   sustituir cada producción de la forma $A_i \to A_j\gamma$
   por las producciones $A_i \to \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$,
   donde $A_j \to \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ son todas
   las producciones actuales de A_j;
   eliminar la recursividad inmediata por la izquierda entre las producciones de A_i
   end

Fig. 4.7. Algoritmo para eliminar la recursividad por la izquierda de una gramática.

Método. Aplíquese el algoritmo de la figura 4.7 a G. Obsérvese que la gramática sin recursividad por la izquierda resultante puede tener producciones ϵ. □

La razón por la que el procedimiento de la figura 4.7 funciona es que después de la $(i-1)$-ésima iteración del lazo **for** externo en el paso 2, cualquier producción de la forma $A_k \rightarrow A_l\alpha$, donde $k<i$, debe tener $l>k$. Como resultado, en la siguiente iteración, el lazo interno (sobre j) aumenta progresivamente el límite inferior en m en cualquier producción $A \rightarrow A_m\alpha$, hasta que se tenga $m \geq i$. Entonces, eliminar la recursión directa por la izquierda de las producciones de A_i hace que m sea mayor que i.

Ejemplo 4.9. Aplicación de este procedimiento a la gramática (4.12). Desde el punto de vista técnico, el algoritmo 4.1 no siempre funciona, debido a la producción ϵ, pero en este caso la producción $A \rightarrow \epsilon$ resulta ser inofensiva.

Se ordenan los no terminales S, A. No hay recursión directa por la izquierda entre las producciones de S, de modo que no ocurre nada durante el paso 2 para el caso $i = 1$. Para $i = 2$, se sustituyen las producciones de S en $A \rightarrow Sd$ para obtener las siguientes producciones de A.

$$A \rightarrow Ac \mid Aad \mid bd \mid \epsilon$$

Eliminando la recursión directa por la izquierda entre las producciones de A, se obtiene la siguiente gramática.

$$S \rightarrow Aa \mid b$$
$$A \rightarrow bdA' \mid A'$$
$$A' \rightarrow cA' \mid adA' \mid \epsilon$$

□

Factorización por la izquierda

La factorización por la izquierda es una transformación gramatical útil para producir una gramática adecuada para el análisis sintáctico predictivo. La idea básica es que cuando no está claro cuál de dos producciones alternativas utilizar para ampliar un no terminal A, se pueden reescribir las producciones de A para retrasar la decisión hasta haber visto lo suficiente de la entrada como para elegir la opción correcta.

Por ejemplo, si se tienen las dos producciones

$$\begin{aligned} \textit{prop} \rightarrow\ & \textbf{if}\ \textit{expr}\ \textbf{then}\ \textit{prop}\ \textbf{else}\ \textit{prop} \\ \mid\ & \textbf{if}\ \textit{expr}\ \textbf{then}\ \textit{prop} \end{aligned}$$

al ver el componente léxico de entrada **if**, no se puede saber de inmediato qué producción elegir para expandir *prop*. En general, si $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ son dos producciones de A y la entrada comienza con una cadena no vacía derivada de α, no se sabe si expandir A a $\alpha\beta_1$ o a $\alpha\beta_2$. Sin embargo, se puede retrasar la decisión expandiendo A a $\alpha A'$. Entonces, después de ver la entrada derivada de α, se puede expandir A' a β_1 o a β_2. Es decir, factorizadas por la izquierda, las producciones originales se convierten en

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta_1 \mid \beta_2$$

Algoritmo 4.2. Factorización por la izquierda de una gramática.

Entrada. La gramática G.

Salida. Una gramática equivalente factorizada por la izquierda.

Método. Para cada no terminal A, encuéntrese el prefijo α más largo común a dos o más de sus alternativas. Si $\alpha \neq \epsilon$, es decir, existe un prefijo común no trivial, sustitúyanse todas las producciones de A, $A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma$, donde γ representa todas las alternativas que no comienzan con α, por

$$\begin{aligned} A &\to \alpha A' \mid \gamma \\ A' &\to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n \end{aligned}$$

Aquí, A' es un nuevo no terminal. Aplicar repetidamente esta transformación hasta que no haya dos alternativas para un no terminal con un prefijo común. □

Ejemplo 4.10. La siguiente gramática resume el problema del **else** ambiguo:

$$\begin{aligned} P &\to iEtP \mid iEtPeP \mid a \\ E &\to b \end{aligned} \tag{4.13}$$

Aquí, i, t y e representan **if**, **then** y **else**; E y P representan "expresión" y "proposición". Factorizada por la izquierda, esta gramática se convierte en:

$$\begin{aligned} P &\to iEtPP' \mid a \\ P' &\to eP \mid \epsilon \\ E &\to b \end{aligned} \tag{4.14}$$

Así, se puede expandir P a $iEtPP'$ con la entrada i, y esperar hasta que $iEtP$ haya aparecido para decidir si expandir P' a eP o a ϵ. Por supuesto, las gramáticas (4.13) y (4.14) son ambiguas, y con la entrada e, no está claro qué alternativa de P' se debería elegir. El ejemplo 4.19 analiza una manera de solucionar este dilema. □

Construcciones de lenguajes no independientes del contexto

No debe sorprender que algunos lenguajes no puedan ser generados por ninguna gramática. De hecho, unas cuantas construcciones sintácticas de muchos lenguajes de programación no se pueden especificar utilizando tan sólo gramáticas. En esta sección se introducen algunas de estas construcciones usando lenguajes abstractos sencillos para ilustrar las dificultades.

Ejemplo 4.11. Considérese el lenguaje abstracto $L_1 = \{wcw \mid w \text{ está en } (a \mid b)^*\}$. L_1 consta de todas las palabras compuestas por una cadena repetida de caracteres a y b separados por una c, como $aabcaab$. Se puede demostrar que este lenguaje no es independiente del contexto. Este lenguaje resume el problema de asegurar que los identificadores se declaren antes de su uso en un programa. Es decir, la primera w de wcw representa la declaración de un identificador w. La segunda w representa su uso. Mientras que demostrarlo está más allá del propósito de este libro, la falta de independencia del contexto de L_1 implica directamente la no independencia del contexto de lenguajes de programación como ALGOL y Pascal, que exigen decla-

ración de los identificadores antes de su uso, y que admiten identificadores de cualquier longitud.

Por esta razón, una gramática para la sintaxis de ALGOL o Pascal no especifica los caracteres en un identificador, sino que todos los identificadores se representan en la gramática mediante un componente léxico como **id**. En un compilador para un lenguaje de este tipo, la fase de análisis semántico comprueba si los identificadores han sido declarados antes de su uso. □

Ejemplo 4.12. El lenguaje $L_2 = \{a^n b^m c^n d^m \mid n \geqslant 1 \text{ y } m \geqslant 1\}$ no es independiente del contexto. Es decir, L_2 consta de cadenas en el lenguaje generado por la expresión regular $a^*b^*c^*d^*$ tales que los números de a y c son iguales, lo mismo que los números de b y d. (Recuérdese que a^n significa a escrita n veces.) L_2 resume el problema de comprobar si el número de parámetros formales en la declaración de un procedimiento coincide con el número de parámetros actuales en un uso de este procedimiento. Es decir, a^n y b^m podrían representar las listas de parámetros formales en dos procedimientos con n y m argumentos, respectivamente. Entonces, c^n y d^m representan las listas de parámetros reales en llamadas a dichos procedimientos.

De nuevo, se observa que la sintaxis típica de las definiciones y usos de procedimientos no se ocupa de contar el número de parámetros. Por ejemplo, la proposición `CALL` en un lenguaje del tipo FORTRAN se podría describir

$$\begin{aligned} \textit{prop} &\rightarrow \textbf{call id } (\textit{lista\_expr}) \\ \textit{lista\_expr} &\rightarrow \textit{lista\_expr} \text{ , } \textit{expr} \\ &\mid \textit{expr} \end{aligned}$$

con las producciones adecuadas para *expr*. Generalmente se comprueba si el número de parámetros actuales en la llamada es correcto durante la fase de análisis semántico. □

Ejemplo 4.13. El lenguaje $L_3 = \{a^n b^n c^n \mid n > 0\}$, es decir, cadenas en $L(a^*b^*c^*)$ con el mismo número de caracteres a, b y c, no es independiente del contexto. Un ejemplo de un programa que incluye L_3 es el siguiente. Los textos de tipografía utilizan *cursivas* donde los textos corrientes utilizan el subrayado. Al convertir un archivo de texto destinado a imprimirse en una impresora de líneas en texto adecuado para un dispositivo de fotocomposición, hay que sustituir las palabras subrayadas por *cursivas*. Una palabra subrayada es una cadena de letras seguida de un mismo número de retrocesos (caracteres de **back space** de ASCII) y de un número igual de caracteres de subrayado. Si se considera a como una letra, b como el carácter de retroceso y c como el carácter de subrayado, el lenguaje L_3 representa palabras subrayadas. La conclusión es que no se puede utilizar una gramática para describir palabras subrayadas de esta forma. Por otra parte, si se representa una palabra subrayada mediante una secuencia de triples letra-retroceso-subrayado, entonces se pueden representar las palabras subrayadas con la expresión regular $(abc)^*$. □

Es interesante observar que lenguajes muy similares a L_1, L_2 y L_3 son independientes del contexto. Por ejemplo, $L'_1 = \{wcw^R \mid w \text{ está en } (a \mid b)^*\}$, donde w^R representa una w invertida, es independiente del contexto. Se genera por la gramática

$$S \rightarrow aSa \mid bSb \mid c$$

El lenguaje $L'_2 = \{a^n b^m c^m d^n \mid n \geqslant 1 \text{ y } m \geqslant 1\}$ es independiente del contexto, con la gramática

$$S \rightarrow aSd \mid aAd$$
$$A \rightarrow bAc \mid bc$$

Asimismo, $L''_2 = \{a^n b^n c^m d^m \mid n \geqslant 1 \text{ y } m \geqslant 1\}$ es independiente del contexto, con la gramática

$$S \rightarrow AB$$
$$A \rightarrow aAb \mid ab$$
$$B \rightarrow cBd \mid cd$$

Por último, $L'_3 = \{a^n b^n \mid n \geqslant 1\}$ es independiente del contexto, con la gramática

$$S \rightarrow aSb \mid ab$$

Es conveniente observar que L'_3 es el típico ejemplo de lenguaje no definible por ninguna expresión regular. Como prueba, supóngase que L'_3 fuera el lenguaje definido por una expresión regular. Asimismo, supóngase que se pudiera construir un AFD D que aceptara L'_3. D debe tener un número finito de estados, por ejemplo k. Considérese la secuencia de estados $s_0, s_1, s_2, \ldots, s_k$ introducidos por D al haber leído ϵ, a, aa, $\ldots$, a^k. Es decir, s_i es el estado visitado por D al haber leído i caracteres a.

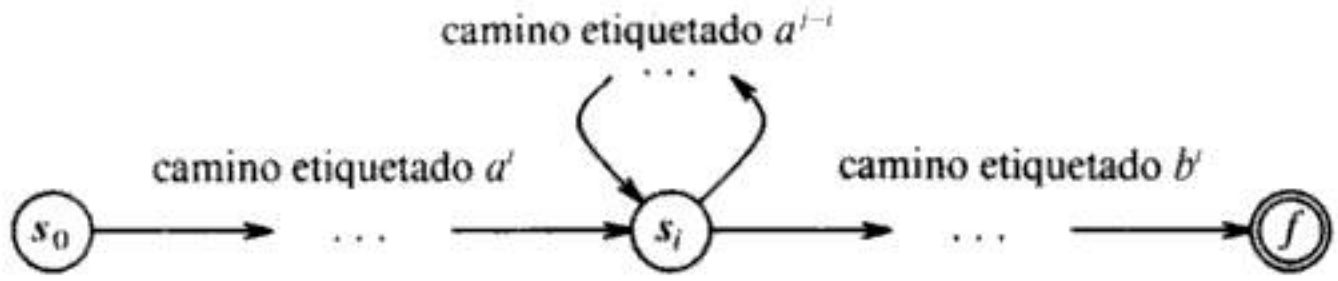


Fig. 4.8. AFD D que acepta $a^i b^j$ y $a^j b^i$.

Puesto que D sólo tiene k estados distintos, al menos dos estados de la secuencia $s_0, s_1, s_2, \ldots, s_k$ deben ser el mismo, por ejemplo s_i y s_j. Desde el estado s_i, una secuencia de i caracteres b lleva a D a un estado de aceptación f, ya que $a^i b^i$ está en L'_3. Pero entonces también hay un camino desde el estado inicial s_0 a s_i y de ahí a f etiquetado con $a^j b^i$, como se muestra en la figura 4.8. Por tanto, D también acepta $a^j b^i$, que no está en L'_3, contradiciendo el supuesto de que L'_3 es el lenguaje aceptado por D.

Coloquialmente, se dice que "un autómata finito no puede contar", lo cual significa que un autómata finito no puede aceptar un lenguaje como L'_3, que le exigiría llevar la cuenta del número de caracteres a antes de ver los caracteres b. Asimismo, se dice que "una gramática puede llevar la cuenta de dos elementos, pero no de tres", puesto que con una gramática se puede definir L'_3 pero no L_3.

4.4 ANALISIS SINTACTICO DESCENDENTE

En esta sección se introducen las ideas básicas del análisis sintáctico descendente y se enseña a construir una forma eficiente sin retroceso de un analizador sintáctico descendente llamada analizador sintáctico predictivo. Se define la clase de gramáticas LL(1), a partir de las cuales se pueden construir de manera automática analizadores sintácticos predictivos. Además de formalizar el estudio de los analizadores sintácticos predictivos de la sección 2.4, se consideran analizadores sintácticos predictivos no recursivos. Esta sección concluye con un análisis sobre la recuperación de errores. Los analizadores sintácticos ascendentes se estudian en las secciones 4.5 a 4.7.

Análisis sintáctico por descenso recursivo

Se puede considerar el análisis sintáctico descendente como un intento de encontrar una derivación por la izquierda para una cadena de entrada. También se puede considerar como un intento de construir un árbol de análisis sintáctico para la entrada comenzando desde la raíz y creando los nodos del árbol en orden previo. En la sección 2.4 se estudió el caso especial del análisis sintáctico por descenso recursivo, llamado análisis sintáctico predictivo, donde no se necesita retroceso. Ahora se considera una forma general de análisis sintáctico descendente, denominado por descenso recursivo, que puede incluir retrocesos, es decir, varios exámenes de la entrada. Sin embargo, no hay muchos analizadores sintácticos con retroceso. En parte, porque casi nunca se necesita el retroceso para analizar sintácticamente las construcciones de los lenguajes de programación. En casos como el análisis sintáctico del lenguaje natural, el retroceso tampoco es muy eficiente, y se prefieren los métodos tabulares, como el algoritmo de programación dinámica del ejercicio 4.63 o el método de Earley [1970]. Véase en Aho y Ullman [1972b] una descripción de métodos generales de análisis sintáctico.

En el siguiente ejemplo, el retroceso es necesario, y se sugiere una forma de no perder la entrada cuando tiene lugar el retroceso.

Ejemplo 4.14. Considérese la gramática

$$\begin{aligned} S &\rightarrow cAd \\ A &\rightarrow ab \mid a \end{aligned} \tag{4.15}$$

y la cadena de entrada $w = cad$. Para construir un árbol de análisis sintáctico descendente para esta cadena, primero se crea un árbol formado por un solo nodo etiquetado con S. Un apuntador a la entrada apunta a c, el primer símbolo de w. Después se utiliza la primera producción de S para expandir el árbol y obtener el árbol de la figura 4.9(a).

Se empareja la hoja situada más a la izquierda, etiquetada con c, con el primer símbolo de w, y a continuación se aproxima el apuntador de entrada a a, el segundo símbolo de w, y se considera la siguiente hoja etiquetada con A. Entonces se puede expandir A utilizando la primera alternativa de A para obtener el árbol de la figura 4.9(b). Como ya se tiene una concordancia para el segundo símbolo de la entrada se lleva el apuntador de entrada a d, el tercer símbolo de la entrada, y se compara d

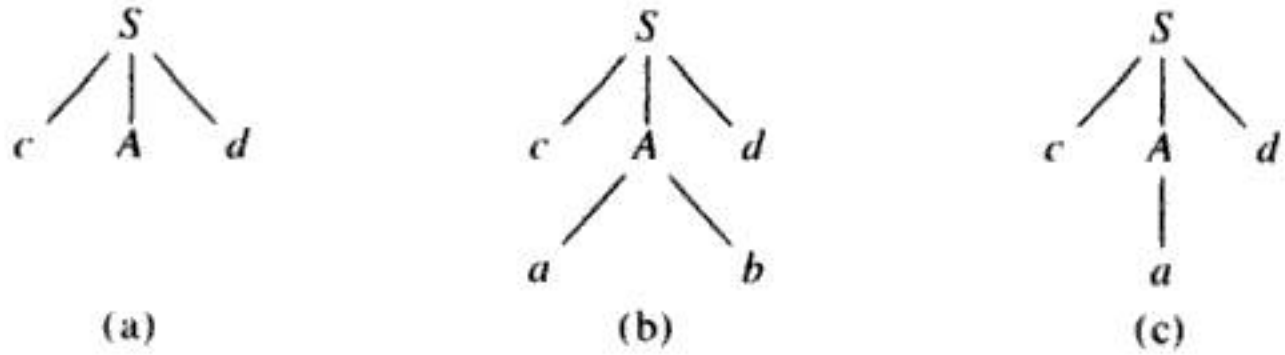


Fig. 4.9. Pasos en el análisis sintáctico descendente.

con la hoja siguiente, etiquetada con b. Como b no concuerda con d, se indica fallo y se regresa a A para saber si existe otra alternativa de A que no se haya intentado, pero que pueda dar lugar a un emparejamiento.

Al regresar a A, se debe restablecer el apuntador de entrada a la posición 2, aquella que tenía al ir a A por primera vez, lo cual significa que el procedimiento para A (análogo al procedimiento para no terminales de la Fig. 2.17) debe almacenar el apuntador a la entrada en una variable local. Se intenta a continuación la segunda alternativa de A para obtener el árbol de la figura 4.9(c). Se empareja la hoja a con el segundo símbolo de w, y la hoja d, con el tercer símbolo. Como ya se ha producido un árbol de análisis sintáctico para w, se para y se anuncia el éxito de la realización completa del análisis sintáctico. □

Una gramática recursiva por la izquierda puede hacer que un analizador sintáctico por descenso recursivo, incluso uno con retroceso, entre en un lazo infinito. Es decir, cuando se intenta expandir A, puede que de nuevo se esté intentando expandir A sin haber consumido ningún símbolo de entrada.

Analizadores sintácticos predictivos

En muchos casos, escribiendo con cuidado una gramática, eliminando su recursión por la izquierda y factorizando por la izquierda la gramática resultante, se puede obtener una gramática analizable con un analizador sintáctico por descenso recursivo que no necesite retroceso, es decir, un analizador sintáctico predictivo, como se estudió en la sección 2.4. Para construir un analizador sintáctico predictivo, se debe conocer, dado el símbolo actual a de entrada y el no terminal A a expandir, cuál de las alternativas de producción $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$ es la única alternativa que da lugar a una cadena que comience con a. Es decir, la alternativa apropiada debe ser detectable con sólo ver el primer símbolo al que da lugar. Así se detectan generalmente las construcciones de flujo de control de la mayoría de los lenguajes de programación, con sus palabras clave diferenciadoras. Por ejemplo, si se tienen las producciones

prop → **if** *expr* **then** *prop* **else** *prop*
| **while** *expr* **do** *prop*
| **begin** *lista_props* **end**

las palabras clave **if**, **while** y **begin** indican qué alternativa es la única con posibilidad de éxito para encontrar una proposición.

Diagramas de transiciones para analizadores sintácticos predictivos

En la sección 2.4 se estudió la implantación de analizadores sintácticos predictivos mediante procedimientos recursivos, como los de la figura 2.17. Igual que en la sección 3.4 se vio que un diagrama de transiciones es un plan o diagrama de flujo útil para un analizador léxico, se puede crear un diagrama de transiciones como plan para un analizador sintáctico predictivo.

En seguida se evidencian varias diferencias entre los diagramas de transiciones para un analizador léxico y para un analizador sintáctico predictivo. En el caso de un analizador sintáctico, hay un diagrama para cada no terminal. Las etiquetas de las aristas son componentes léxicos y no terminales. Una transición con un componente léxico (terminal) supone que se debe tomar dicha transición si ese componente léxico es el siguiente símbolo de entrada. Una transición con un no terminal A es una llamada al procedimiento para A.

Para construir el diagrama de transiciones de un analizador sintáctico predictivo a partir de una gramática, primero se debe eliminar la recursión por la izquierda de la gramática, y después factorizar dicha gramática por la izquierda. Luego, para cada no terminal A se hace lo siguiente:

1. Créese un estado inicial y un estado final (de retorno).
2. Para cada producción $A \rightarrow X_1 X_2 \ldots X_n$, créese un camino desde el estado inicial al estado final, con aristas etiquetadas con $X_1, X_2, \ldots, X_n$.

El analizador sintáctico predictivo que se desprende de los diagramas de transiciones se comporta como sigue. Comienza en el estado de inicio del símbolo inicial. Si después de algunos movimientos se encuentra en el estado s con una arista etiquetada con el terminal a al estado t, y si el siguiente símbolo de entrada es a, entonces el analizador sintáctico cambia el cursor de la entrada una posición a la derecha y se va al estado t. Si, por otra parte, la arista está etiquetada con un no terminal A, el analizador sintáctico va al estado de inicio de A, sin mover el cursor de la entrada. Si llega a alcanzar el estado final de A, inmediatamente va al estado t, habiendo en efecto "leído" A de la entrada cuando se trasladó del estado s al t. Por último, si hay una arista de s a t etiquetada con ϵ, el analizador sintáctico va inmediatamente del estado s al t sin avanzar la entrada.

Un programa para hacer análisis sintáctico predictivo basado en un diagrama de transiciones intenta emparejar símbolos terminales con la entrada, y realiza una llamada potencialmente recursiva a un procedimiento siempre que deba seguir una arista etiquetada con un no terminal. Se puede obtener una implantación no recursiva con una pila para guardar los estados s cuando hay una transición con un no terminal saliendo de s, y eliminando la pila al alcanzar el estado final de un no terminal. Muy pronto se analizará más detalladamente la implantación de diagramas de transición.

El enfoque anterior funciona si el diagrama de transiciones dado no presenta indeterminismo, en el sentido de que haya más de una transición desde un estado con la misma entrada. Si existe ambigüedad, se puede solucionar de una forma específica, como se verá en el siguiente ejemplo. Si no se puede eliminar el indeterminismo, no se puede construir un analizador sintáctico predictivo, pero sí un anali-

zador sintáctico por descenso recursivo utilizando el retroceso para intentar sistemáticamente todas las posibilidades, si ésa fuera la mejor estrategia de análisis posible.

Ejemplo 4.15. La figura 4.10 contiene un conjunto de diagramas de transiciones para la gramática (4.11). Las únicas ambigüedades se refieren a si se debe o no tomar una arista ϵ. Si se interpreta que las aristas que salen del estado inicial de E' como indicativas de tomar la transición con $+$ siempre que ésta sea la entrada siguiente o tomar la transición con ϵ en otro caso, y realizar el mismo supuesto para T', entonces se elimina la ambigüedad y se puede escribir un programa de análisis sintáctico predictivo para la gramática (4.11). □

Se pueden simplificar los diagramas de transiciones sustituyendo unos diagramas por otros; estas sustituciones son similares a las transformaciones en las gramáticas de la sección 2.5. Por ejemplo, en la figura 4.11(a), la llamada de E' se ha reemplazado a sí misma por un salto hasta el principio del diagrama de E'.

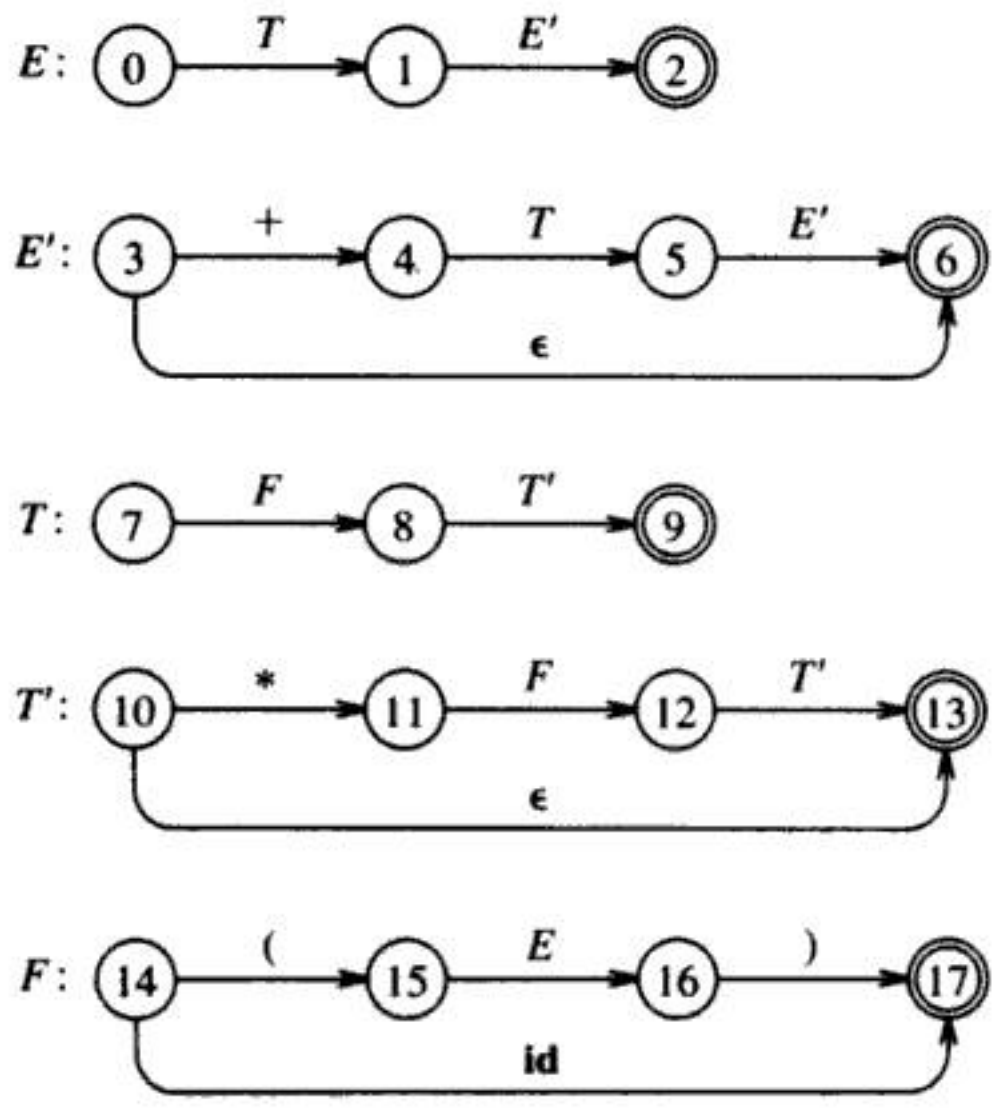


Fig. 4.10. Diagramas de transiciones para la gramática (4.11).

En la figura 4.11(b) se muestra un diagrama de transiciones equivalente para E'. Después se podría sustituir el diagrama de la figura 4.11 para la transición para E' en el diagrama de E de la figura 4.10, obteniéndose el diagrama de la figura 4.11(c). Por último, se observa que el primero y tercer nodos de la figura 4.11(c) son equivalentes y se fusionan. El resultado, figura 4.11(d), se repite en el primer diagrama

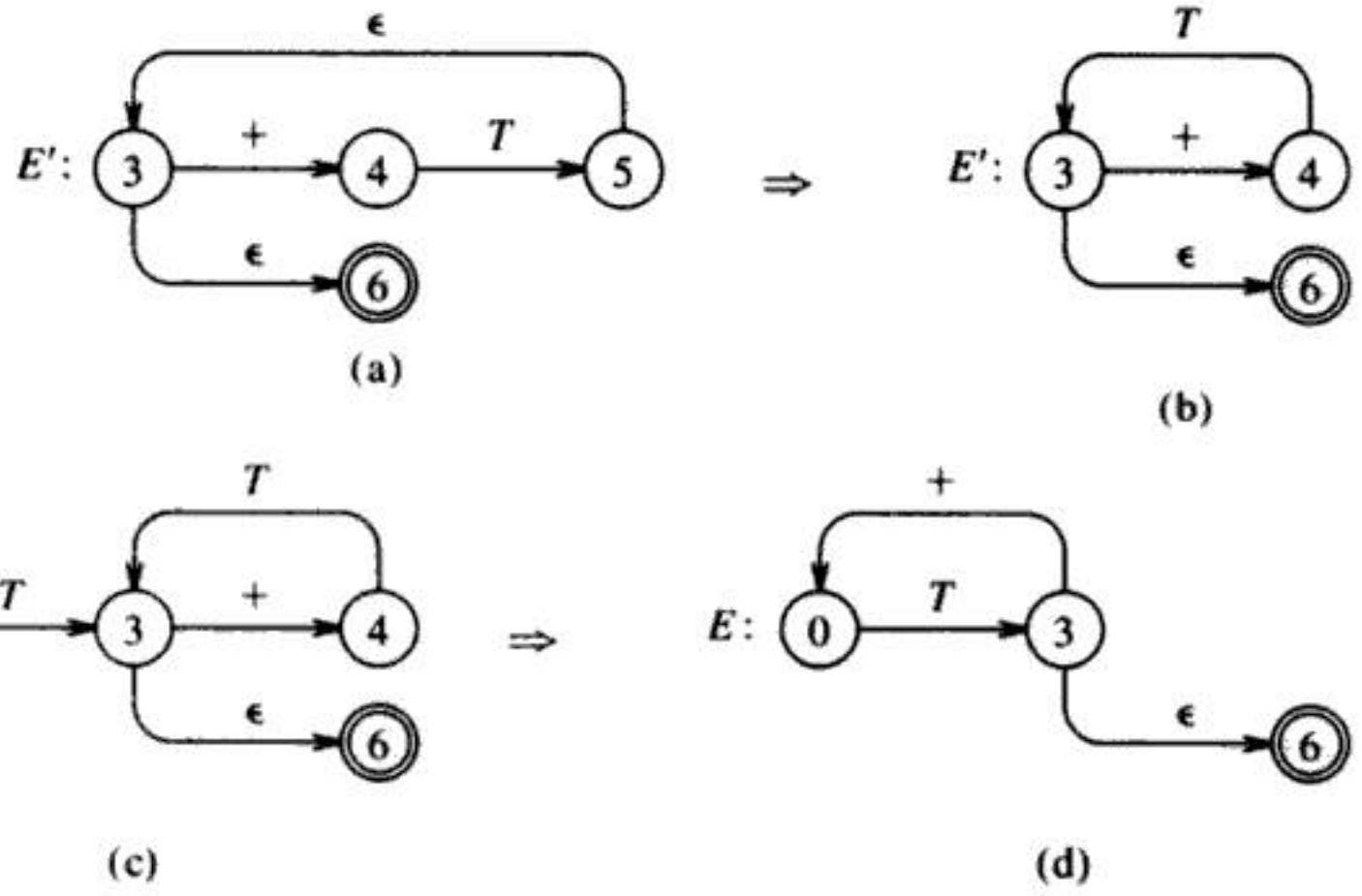


Fig. 4.11. Diagramas de transiciones simplificados.

de la figura 4.12. Las mismas técnicas sirven para los diagramas de T y T'. En la figura 4.12 se muestra el conjunto completo de los diagramas obtenidos. Una implantación en C de este analizador sintáctico predictivo funciona un 20 ó 25 por 100 más rápidamente que una implantación en el mismo lenguaje de la figura 4.10.

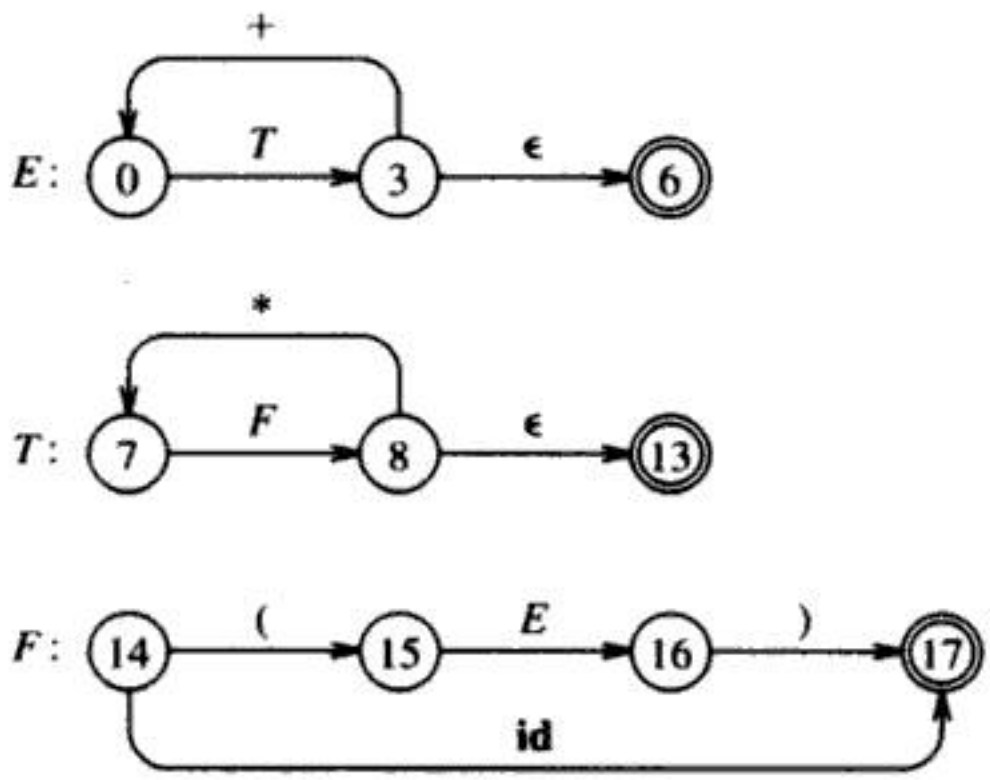


Fig. 4.12. Diagramas de transiciones simplificados para las expresiones aritméticas.

Análisis sintáctico predictivo no recursivo

Se puede construir un analizador sintáctico predictivo no recursivo explícitamente manteniendo una pila, en lugar de hacerlo implícitamente mediante llamadas recursivas. El problema clave durante el análisis sintáctico predictivo es determinar la

producción que debe aplicarse a un no terminal. El analizador sintáctico no recursivo de la figura 4.13 busca la producción que debe aplicarse en una tabla de análisis sintáctico. A continuación se verá cómo se puede construir directamente la tabla a partir de ciertas gramáticas.

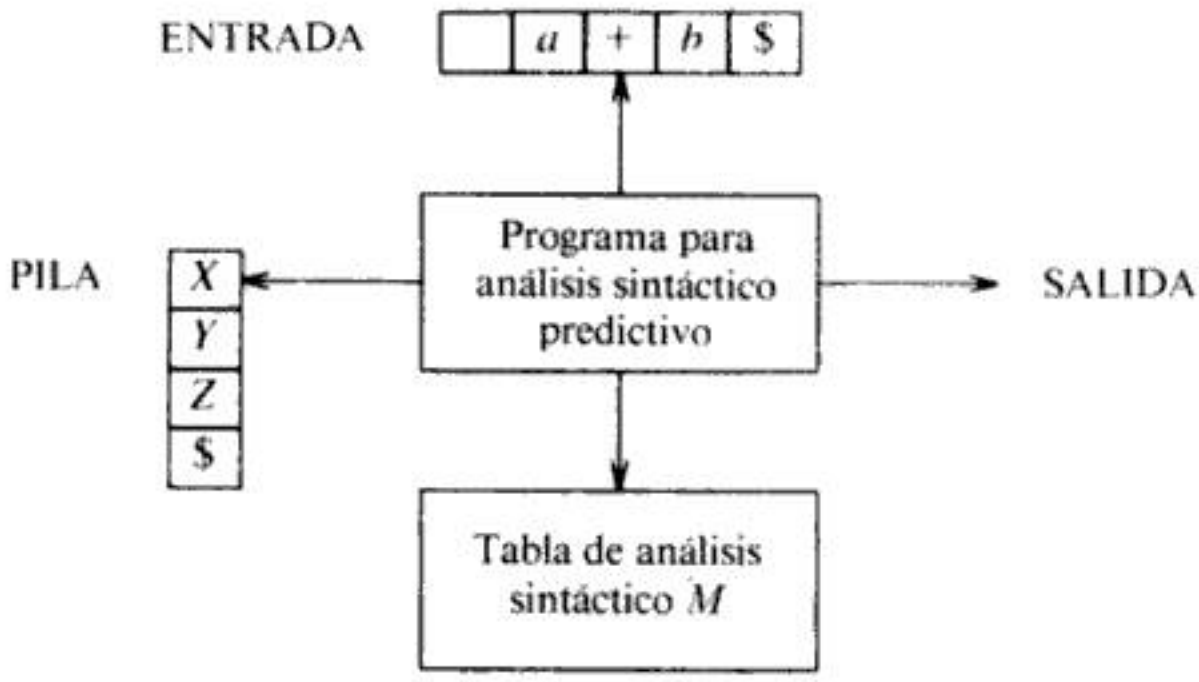


Fig. 4.13. Modelo de un analizador sintáctico predictivo no recursivo.

Un analizador sintáctico predictivo guiado por tablas tiene un *buffer* de entrada, una pila, una tabla de análisis sintáctico y una cadena de salida. El *buffer* de entrada contiene la cadena que se va a analizar, seguida de $, un símbolo utilizado como delimitador derecho para indicar el fin de la cadena de entrada. La pila contiene una secuencia de símbolos gramaticales con $ en la parte de abajo, que indica la base de la pila. Al principio, la pila contiene el símbolo inicial de la gramática encima de $. La tabla de análisis sintáctico es una matriz bidimensional $M[A, a]$, donde A es un no terminal, y a es un terminal o el símbolo $.

Se controla el analizador sintáctico mediante un programa que se comporta como sigue. El programa tiene en cuenta X, el símbolo de la cima de la pila, y a, el símbolo en curso de la entrada. Estos dos símbolos determinan la acción del analizador. Existen tres posibilidades:

1. Si $X = a = \$$, el analizador sintáctico se detiene y anuncia el éxito de la realización del análisis.
2. Si $X = a \neq \$$, el analizador sintáctico saca a X de la pila y mueve el apuntador de entrada al siguiente símbolo de entrada.
3. Si X es un no terminal, el programa consulta la entrada $M[X, a]$ de la tabla M de análisis sintáctico. Esta entrada será o una producción de X de la gramática o una entrada de error. Si, por ejemplo, $M[X, a] = \{X \rightarrow UVW\}$, el analizador sintáctico sustituye la X de la cima de la pila por WVU (con U en la cima). Como salida, se sabe que el analizador sintáctico sólo imprime la producción utilizada; ahí se podría ejecutar cualquier otro código. Si $M[X, a] =$ **error**, el analizador sintáctico llama a una rutina de recuperación de error.

Se puede describir el comportamiento del analizador sintáctico en función de sus *configuraciones*, que dan el contenido de la pila y la entrada restante.

Algoritmo 4.3. Análisis sintáctico predictivo no recursivo.

Entrada. Una cadena w y una tabla de análisis sintáctico M para la gramática G.

Salida. Si w está en $L(G)$, una derivación por la izquierda de w; de lo contrario, una indicación de error.

Método. Al principio, el analizador sintáctico está en una configuración en la que tiene a $S en la pila con S, el símbolo inicial de G en el tope, y w en el *buffer* de entrada. En la figura 4.14 se muestra el programa que utiliza la tabla de análisis sintáctico predictivo M para producir un análisis de la entrada. □

```
apuntar ae al primer símbolo de w$;
repeat
    sea X el símbolo de la cima de la pila y a el símbolo apuntado por ae;
    if X es un terminal o $ then
        if X = a then
            extraer X de la pila y avanzar ae
        else error()
    else        /* X es un no terminal */
        if M[X, a] = X → Y1 Y2 ... Yk then begin
            extraer X de la pila;
            meter Yk, Yk-1, ..., Y1 en la pila, con Y1 en la cima;
            emitir la producción X → Y1 Y2 ... Yk
        end
        else error()
until X = $     /* la pila está vacía */
```

Fig. 4.14. Programa para análisis sintáctico predictivo.

Ejemplo 4.16. Considérese la gramática (4.11) del ejemplo 4.8. En la figura 4.15 se muestra una tabla de análisis sintáctico predictivo para esta gramática. Los espacios en blanco son entradas de error; los otros espacios indican una producción con la cual expandir el no terminal de la cima en la pila. Obsérvese que aún no se ha indicado cómo seleccionar dichas entradas, pero se indicará en breve.

Con la entrada **id** + **id** * **id** el analizador sintáctico predictivo realiza la secuencia de movimientos de la figura 4.16. El apuntador de entrada apunta al símbolo de la extrema izquierda de la cadena en la columna ENTRADA. Si se observan con atención las acciones de este analizador sintáctico, se nota que está buscando una derivación por la izquierda para la entrada, es decir, las producciones emitidas son las de una derivación por la izquierda. Los símbolos de entrada que ya se han examinado, seguidos de los símbolos gramaticales de la pila (de la cima al fondo), son las formas de frase izquierdas de la derivación. □

NO TERMINAL	SÍMBOLO DE ENTRADA					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow$ **id**			$F \rightarrow (E)$		

Fig. 4.15. Tabla de análisis sintáctico M para la gramática (4.11).

PILA	ENTRADA	SALIDA
$E	**id** + **id** * **id**$	
$E'T	**id** + **id** * **id**$	$E \rightarrow TE'$
$E'T' F	**id** + **id** * **id**$	$T \rightarrow FT'$
$E'T'$**id**	**id** + **id** * **id**$	$F \rightarrow$ **id**
$E'T'	+ **id** * **id**$	
$E'	+ **id** * **id**$	$T' \rightarrow \epsilon$
$E'T +	+ **id** * **id**$	$E' \rightarrow +TE'$
$E'T	**id** * **id**$	
$E'T'F	**id** * **id**$	$T \rightarrow FT'$
$E'T'$**id**	**id** * **id**$	$F \rightarrow$ **id**
$E'T'	* **id**$	
$E'T'F *	* **id**$	$T' \rightarrow *FT'$
$E'T'F	**id**$	
$E'T'$**id**	**id**$	$F \rightarrow$ **id**
$E'T'	$	
$E'	$	$T' \rightarrow \epsilon$
$	$	$E' \rightarrow \epsilon$

Fig. 4.16. Movimientos realizados por el analizador sintáctico predictivo con la entrada **id** + **id*id**.

PRIMERO y SIGUIENTE

Se facilita la construcción de un analizador sintáctico predictivo con dos funciones asociadas a una gramática G. Estas funciones, PRIMERO y SIGUIENTE, permiten rellenar, siempre que sea posible, las entradas de una tabla de análisis sintáctico predictivo para G. También se pueden utilizar los conjuntos de componentes léxicos devueltos por la función SIGUIENTE como componentes léxicos de sincronización durante la recuperación de errores en modo de pánico.

Si α es una cadena de símbolos gramaticales, se considera *PRIMERO*(α) como el conjunto de terminales que inician las cadenas derivadas de α. Si $\alpha \stackrel{*}{\Rightarrow} \epsilon$, entonces ϵ también está en PRIMERO(α).

Se define *SIGUIENTE*(A), para el no terminal A, como el conjunto de terminales a que pueden aparecer inmediatamente a la derecha de A en alguna forma de frase, es decir, el conjunto de terminales a tal que haya una derivación de la forma $S \stackrel{*}{\Rightarrow} \alpha A a \beta$ para algún α y β. Obsérvese que en algún momento de la derivación pudieron haber existido símbolos entre A y a, pero si así fue, derivaron a ϵ y desaparecieron. Si A puede ser el símbolo situado más a la derecha en una forma de frase, entonces \$ está en SIGUIENTE(A).

Para calcular PRIMERO(X) para todos los símbolos gramaticales X, aplíquense las reglas siguientes hasta que no se puedan añadir más terminales o ϵ a ningún conjunto PRIMERO.

1. Si X es terminal, entonces PRIMERO(X) es $\{X\}$.
2. Si $X \rightarrow \epsilon$ es una producción, entonces añádase ϵ a PRIMERO(X).
3. Si X es no terminal y $X \rightarrow Y_1 Y_2 \ldots Y_k$ es una producción, entonces póngase a en PRIMERO(X) si, para alguna i, a está en PRIMERO(Y_i) y ϵ está en todos los PRIMERO(Y_1), . . . , PRIMERO(Y_{i-1}); es decir, $Y_1 \ldots Y_{i-1} \stackrel{*}{\Rightarrow} \epsilon$. Si ϵ está en PRIMERO(Y_j) para toda $j = 1, 2, \ldots, k$, entonces añádase ϵ a PRIMERO(X). Por ejemplo, todo lo que está en PRIMERO(Y_1) sin duda está en PRIMERO(X). Si Y_1 no deriva a ϵ, entonces no se añade nada más a PRIMERO(X), pero si $Y_1 \stackrel{*}{\Rightarrow} \epsilon$, entonces se le añade PRIMERO(Y_2), y así sucesivamente.

Ahora se puede calcular PRIMERO para cualquier cadena $X_1 X_2 \ldots X_n$ de la siguiente forma: añádanse a PRIMERO($X_1 X_2 \ldots X_n$) todos los símbolos distintos de ϵ de PRIMERO(X_1). Si ϵ está en PRIMERO(X_1), añádanse también los símbolos distintos de ϵ de PRIMERO(X_2); si ϵ está tanto en PRIMERO(X_1) como en PRIMERO(X_2), añádanse también los símbolos distintos de ϵ de PRIMERO(X_3), y así sucesivamente. Por último, añádase ϵ a PRIMERO($X_1 X_2 \ldots X_n$) si, para toda i, PRIMERO(X_i) contiene ϵ.

Para calcular SIGUIENTE(A) para todos los no terminales A, aplíquense las reglas siguientes hasta que no se pueda añadir nada más a ningún conjunto SIGUIENTE.

1. Póngase \$ en SIGUIENTE(S), donde S es el símbolo inicial y \$ es el delimitador derecho de la entrada.
2. Si hay una producción $A \rightarrow \alpha B \beta$, entonces todo lo que esté en PRIMERO(β) excepto ϵ se pone en SIGUIENTE(B).
3. Si hay una producción $A \rightarrow \alpha B$ o una producción $A \rightarrow \alpha B \beta$, donde PRIMERO($\beta$) contenga ϵ (es decir, $\beta \stackrel{*}{\Rightarrow} \epsilon$), entonces todo lo que esté en SIGUIENTE(A) se pone en SIGUIENTE(B).

Ejemplo 4.17. Considérese de nuevo la gramática (4.11), que se repite a continuación:

$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \epsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT' \mid \epsilon$$
$$F \rightarrow (\ E\ ) \mid \mathbf{id}$$

Entonces:

$$\text{PRIMERO}(E) = \text{PRIMERO}(T) = \text{PRIMERO}(F) = \{(, \mathbf{id}\}$$
$$\text{PRIMERO}(E') = \{+, \epsilon\}$$
$$\text{PRIMERO}(T') = \{*, \epsilon\}$$
$$\text{SIGUIENTE}(E) = \text{SIGUIENTE}(E') = \{), \$\}$$
$$\text{SIGUIENTE}(T) = \text{SIGUIENTE}(T') = \{+,), \$\}$$
$$\text{SIGUIENTE}(F) = \{+, *,), \$\}$$

Por ejemplo, se añaden **id** y el paréntesis izquierdo a PRIMERO(F) por la regla 3 de la definición de PRIMERO, con $i = 1$ en cada caso, puesto que PRIMERO(**id**) = {**id**} y PRIMERO('(') = { (} por la regla 1. Entonces por la regla 3, con $i = 1$, la producción $T \rightarrow FT'$ supone que **id** y el paréntesis izquierdo están asimismo en PRIMERO(T). Otro ejemplo más, ϵ está en PRIMERO(E') por la regla 2.

Para calcular los conjuntos SIGUIENTE, se pone \$ en SIGUIENTE(E) por la regla 1 de SIGUIENTE. Por la regla 2 aplicada a la producción $F \rightarrow (E)$, el paréntesis derecho también está en SIGUIENTE(E). Por la regla 3 aplicada a la producción $E \rightarrow TE'$, \$ y el paréntesis derecho están en SIGUIENTE(E'). Como $E' \stackrel{*}{\Rightarrow} \epsilon$, también están en SIGUIENTE(T). Como último ejemplo de la aplicación de las reglas de SIGUIENTE, la producción $E \rightarrow TE'$ supone, por la regla 2, que todo lo que esté en PRIMERO(E'), salvo ϵ, debe ponerse en SIGUIENTE(T). Ya se ha visto que \$ está en SIGUIENTE(T). □

Construcción de tablas de análisis sintáctico

Se puede utilizar el siguiente algoritmo para construir una tabla de análisis sintáctico predictivo para una gramática G. La idea en que se basa el algoritmo es la siguiente. Supóngase que $A \rightarrow \alpha$ es una producción con a en PRIMERO(α). Entonces, el analizador sintáctico expandirá A por α cuando el símbolo actual de la entrada sea a. La única complicación surge cuando $\alpha = \epsilon$ o $\alpha \stackrel{*}{\Rightarrow}$. En este caso, se debe expandir de nuevo A en α si el símbolo actual de la entrada está en SIGUIENTE(A), o si ya se ha alcanzado en \$ de la entrada y \$ está en SIGUIENTE(A).

Algoritmo 4.4. Construcción de una tabla de análisis sintáctico predictivo.

Entrada. Una gramática G.

Salida. La tabla de análisis sintáctico M.

Método.

1. Para cada producción $A \rightarrow \alpha$ de la gramática, dense los pasos 2 y 3.
2. Para cada terminal a de PRIMERO(α), añádase $A \rightarrow \alpha$ a $M[A, a]$.
3. Si ϵ está en PRIMERO(α), añádase $A \rightarrow \alpha$ a $M[A, b]$ para cada terminal b de SIGUIENTE(A). Si ϵ está en PRIMERO(α) y \$ está en SIGUIENTE(A), añádase $A \rightarrow \alpha$ a $M[A, \$]$.
4. Hágase que cada entrada no definida de M sea **error**.

Ejemplo 4.18. Aplicación del algoritmo 4.4 a la gramática (4.11). Puesto que PRIMERO(TE') = PRIMERO(T) = {(, **id**}, la producción $E \rightarrow TE'$ hace que $M[E, (]$ y $M[E,$ **id**$]$ adquieran la entrada $E \rightarrow TE'$.

La producción $E' \rightarrow +TE'$ hace que $M[E', +]$ adquiera $E' \rightarrow +TE'$. La producción $E' \rightarrow \epsilon$ hace que $M[E',)]$ y $M[E', \$]$ adquieran $E' \rightarrow \epsilon$, puesto que SIGUIENTE(E') = {), \$}.

En la figura 4.15 se mostró la tabla de análisis sintáctico producida por el algoritmo 4.4 para la gramática (4.11). □

Gramáticas LL(1)

Se puede aplicar el algoritmo 4.4 a cualquier gramática G para producir una tabla de análisis sintáctico M. Sin embargo, para algunas gramáticas, M puede tener algunas entradas con definiciones múltiples. Por ejemplo, si G es recursiva por la izquierda o ambigua, entonces M tendrá al menos una entrada con definición múltiple.

Ejemplo 4.19. Considérese de nuevo la gramática (4.13) del ejemplo 4.10; conviene repetirla a continuación.

$$P \rightarrow iEtPP' \mid a$$
$$P' \rightarrow eP \mid \epsilon$$
$$E \rightarrow b$$

En la figura 4.17 se muestra la tabla de análisis sintáctico para esta gramática.

No Terminal	Símbolo de Entrada					
	a	b	e	i	t	\$
P	$P \rightarrow a$			$P \rightarrow iEtPP'$		
P'			$P' \rightarrow \epsilon$ $P' \rightarrow eP$			$P' \rightarrow \epsilon$
E		$E \rightarrow b$				

Fig. 4.17. Tabla de análisis sintáctico M para la gramática (4.13).

La entrada para $M[P', e]$ contiene a $P' \rightarrow eP$ y a $P' \rightarrow \epsilon$, puesto que SIGUIENTE(P') = $\{e, \$\}$. La gramática es ambigua y la ambigüedad se manifiesta en la elección de la producción que se va a utilizar cuando se encuentra un e (**else**). Se puede resolver la ambigüedad escogiendo $P' \rightarrow eP$. Esta elección corresponde a asociar los **else** con los **then** previos más cercanos. Obsérvese que la elección de $S' \rightarrow \epsilon$ impediría que e se insertara en la pila o se eliminara de la entrada, lo cual, por supuesto, es incorrecto. □

Una gramática cuya tabla de análisis sintáctico no tiene entradas con definiciones múltiples se define como *LL*(1). La primera "L" de LL(1) representa (por *left*, en inglés, *izquierda*) el examen de la entrada de izquierda a derecha, la segunda "L" representa una derivación por la izquierda, y el "1" es por utilizar un símbolo de entrada de examen por anticipado a cada paso para tomar las decisiones de la acción en el análisis sintáctico. Se puede demostrar que el algoritmo 4.4 produce para toda gramática G en forma LL(1) una tabla de análisis sintáctico que analiza todas, y exclusivamente, las frases de G.

Las gramáticas LL(1) tienen varias propiedades distintivas. Ninguna gramática ambigua o recursiva por la izquierda puede ser LL(1). También se puede demostrar que una gramática G es LL(1) si, y sólo si, cuando $A \rightarrow \alpha \mid \beta$ sean dos producciones distintas de G se cumplen las siguientes condiciones:

1. Para ningún terminal a tanto α como β derivan a la vez cadenas que comiencen con a.
2. A lo sumo una de α y β puede derivar la cadena vacía.
3. Si $\beta \stackrel{*}{\Rightarrow} \epsilon$, α no deriva ninguna cadena que comience con un terminal en SIGUIENTE(A).

Está claro que la gramática (4.11) para las expresiones aritméticas es LL(1). No lo es la gramática (4.13), que modela las proposiciones **if-then-else**.

Queda la cuestión de lo que se debe hacer cuando la tabla de análisis sintáctico tiene entradas con definiciones múltiples. Un recurso es transformar la gramática eliminando toda recursión por la izquierda y factorizando por la izquierda siempre que sea posible, con la esperanza de producir una gramática para la cual la tabla de análisis sintáctico no tenga entradas con definiciones múltiples. Desgraciadamente, hay algunas gramáticas a las que ninguna transformación convertirá en LL(1), por ejemplo, la gramática (4.13), cuyo lenguaje no tiene ninguna gramática LL(1). Como ya se ha visto, también es posible analizar sintácticamente la gramática (4.13) con un analizador predictivo haciendo arbitrariamente que $M[P', e] = \{P' \rightarrow eP\}$. En general, no hay reglas universales por las que las entradas con entradas múltiples se puedan convertir en entradas de un solo valor sin que afecte al lenguaje reconocido por el analizador.

La mayor dificultad al usar el análisis sintáctico predictivo consiste en escribir una gramática para el lenguaje fuente tal que el analizador se pueda construir a partir de dicha gramática. Aunque son fáciles de realizar, la eliminación de la recursividad por la izquierda y la factorización por la izquierda, hacen que la gramática resultante sea difícil de leer y de utilizar para la traducción. Para disminuir esta dificultad, una organización habitual de un analizador sintáctico en un compilador es

utilizar un analizador sintáctico predictivo para las construcciones de control y la precedencia de operadores (que se estudian en la Sec. 4.6) para las expresiones. Sin embargo, si hay disponible un generador de analizadores sintácticos LR, como el que se estudia en la sección 4.9, se pueden aprovechar automáticamente todas las ventajas del análisis sintáctico predictivo y de la precedencia de operadores.

Recuperación de errores en el análisis sintáctico predictivo

La pila de un analizador sintáctico no recursivo hace explícitos los terminales y no terminales que el analizador espera emparejar con el resto de la entrada. Por tanto, se hará referencia a los símbolos de la pila de un analizador sintáctico en la siguiente exposición. Durante el análisis sintáctico predictivo se detecta un error cuando el terminal de la cima de la pila no concuerda con el siguiente símbolo de entrada o cuando el no terminal A está en la cima de la pila, a es el siguiente símbolo de entrada, y la entrada $M[A, a]$ de la tabla de análisis sintáctico está vacía.

La recuperación en modo de pánico se basa en la idea de saltarse símbolos de la entrada hasta que aparezca un componente léxico que pertenezca a un conjunto seleccionado de componentes léxicos de sincronización. Su efectividad depende de la elección del conjunto de sincronización. Los conjuntos deben elegirse de forma que el analizador sintáctico se recupere con rapidez de los errores con más probabilidades de ocurrir en la práctica. Algunas técnicas heurísticas son las siguientes:

1. Como punto de partida, se pueden colocar todos los símbolos de SIGUIENTE(A) dentro del conjunto de sincronización para el no terminal A. Si se saltan componentes léxicos hasta encontrar un elemento de SIGUIENTE(A) y se saca a A de la pila, es probable que el análisis sintáctico pueda continuar.

2. No es suficiente usar SIGUIENTE(A) como conjunto de sincronización para A. Por ejemplo, si los símbolos de punto y coma terminan las proposiciones, como en C, entonces las palabras clave que comienzan proposiciones pueden no aparecer en el conjunto SIGUIENTE del no terminal que genera las expresiones. Por tanto, un punto y coma que falte después de una asignación puede dar como resultado que se salte la palabra clave que inicia la siguiente proposición. A menudo hay una estructura jerárquica en las construcciones de un lenguaje; por ejemplo, las expresiones aparecen dentro de proposiciones, las cuales aparecen dentro de bloques, y así sucesivamente. Se pueden añadir al conjunto de sincronización de una construcción de menor jerarquía los símbolos que inician las construcciones de mayor jerarquía. Por ejemplo, se pueden agregar palabras clave que comienzan proposiciones a los conjuntos de sincronización para los no terminales que generan expresiones.

3. Si se añaden símbolos de PRIMERO(A) al conjunto de sincronización para el no terminal A, entonces se puede continuar el análisis sintáctico según A si aparece en la entrada un símbolo de PRIMERO(A).

4. Si un no terminal puede generar la cadena vacía, se puede usar la producción que derive a ϵ como alternativa por omisión. Esto puede posponer alguna detección de errores pero no la omisión de un error. Este método reduce el número de no terminales que hay que considerar durante la recuperación del error.

5. Si no se puede emparejar un terminal de la cima de la pila, una idea sencilla es sacar el terminal, emitir un mensaje que indique que se insertó el terminal y continuar el análisis. En realidad, este método considera al conjunto de sincronización de un componente léxico como si estuviera compuesto por todos los otros componentes léxicos.

Ejemplo 4.20. Utilizar los símbolos de SIGUIENTE y PRIMERO como componentes léxicos de sincronización funciona bastante bien cuando las expresiones se analizan sintácticamente según la gramática (4.11). La tabla de análisis sintáctico para esta gramática de la figura 4.15 se repite en la figura 4.18, donde "sinc" indica los componentes léxicos de sincronización obtenidos del conjunto SIGUIENTE del no terminal en cuestión. Los conjuntos SIGUIENTE para los no terminales se obtienen del ejemplo 4.17.

La tabla de la figura 4.18 debe utilizarse de la forma siguiente. Si el analizador sintáctico busca la entrada $M[A, a]$ y ve que está en blanco, debe saltarse el símbolo de entrada a. Si la entrada es sinc, se saca el no terminal de la cima de la pila para continuar el análisis. Si un componente léxico de la cima de la pila no concuerda con el símbolo de entrada, entonces se saca el componente léxico de la pila, como ya se ha mencionado.

Con la entrada errónea)**id*** + **id**, el analizador sintáctico y el mecanismo de recuperación de errores de la figura 4.18 se comportan como en la figura 4.19. □

No Terminal	Símbolo de Entrada					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$	sinc	sinc
E'		$E' \to +\, TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$	sinc		$T \to FT'$	sinc	sinc
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to$ **id**	sinc	sinc	$F \to (E)$	sinc	sinc

Fig. 4.18. Componentes léxicos de sincronización añadidos a la tabla de análisis sintáctico de la figura 4.15.

El análisis anterior de la recuperación en modo de pánico no trata el aspecto importante de los mensajes de error. En general, el diseñador del compilador tiene que proporcionar los mensajes informativos de los errores.

Recuperación a nivel de frase. La recuperación a nivel de frase se aplica llenando las entradas en blanco en la tabla de análisis sintáctico predictivo con apuntadores a rutinas de error. Estas rutinas pueden cambiar, insertar o eliminar símbolos de entrada y enviar los mensajes de error apropiados. También pueden sacar elementos de la pila. Se cuestiona si se debe permitir la alteración de los símbolos de la pila o la introducción de símbolos nuevos en ella, puesto que los pasos llevados a cabo por el analizador sintáctico podrían no corresponder a la derivación de nin-

PILA	ENTRADA	COMENTARIO
$\$E$	) **id** * + **id** $	error, saltar)
$\$E$	**id** * + **id** $	**id** está en PRIMERO(E)
$\$E'T$	**id** * + **id** $	
$\$E'T'F$	**id** * + **id** $	
$\$E'T'$**id**	**id** * + **id** $	
$\$E'T'$	* + **id** $	
$\$E'T'F$*	* + **id** $	
$\$E'T'F$	+ **id** $	error, $M[F, +] =$ sinc
$\$E'T'$	+ **id** $	F ha sido extraída de la pila
$\$E'$	+ **id** $	
$\$E'T$ +	+ **id** $	
$\$E'T$	**id** $	
$\$E'T'F$	**id** $	
$\$E'T'$**id**	**id** $	
$\$E'T'$	$	
$\$E'$	$	
$	$	

Fig. 4.19. Movimientos para el análisis y la recuperación de errores realizados por el analizador sintáctico predictivo.

guna palabra del lenguaje. En cualquier caso, se debe estar seguro de que no puede haber un lazo infinito. Comprobar que cualquier acción de recuperación supone que se consuma un símbolo de entrada (o que la pila se acorte si ya se ha alcanzado el final de la entrada) es una buena forma de protegerse contra dichos lazos.

4.5 ANALISIS SINTACTICO ASCENDENTE

En esta sección se introduce un estilo general de análisis sintáctico ascendente, conocido como análisis sintáctico por desplazamiento y reducción. En la sección 4.6 se introduce una forma fácil de aplicar el análisis por desplazamiento y reducción, llamada análisis sintáctico por precedencia de operadores. En la sección 4.7 se estudia un método mucho más general de análisis sintáctico por desplazamiento y reducción, llamado análisis sintáctico LR. El análisis sintáctico LR se utiliza en varios generadores automáticos de analizadores sintácticos.

El análisis sintáctico por desplazamiento y reducción intenta construir un árbol de análisis sintáctico para una cadena de entrada que comienza por las hojas (el fondo) y avanza hacia la raíz (la cima). Se puede considerar este proceso como de "reducir" una cadena w al símbolo inicial de la gramática. En cada paso de *reducción* se sustituye una subcadena determinada que concuerde con el lado derecho de una producción por el símbolo del lado izquierdo de dicha producción y si en cada paso se elige correctamente la subcadena, se traza una derivación por la derecha en sentido inverso.

Ejemplo 4.21. Considérese la gramática

$$S \rightarrow aABe$$
$$A \rightarrow Abc \mid b$$
$$B \rightarrow d$$

La frase *abbcde* se puede reducir a S por los siguientes pasos:

abbcde
aAbcde
aAde
aABe
S

Se examina *abbcde* buscando una subcadena que concuerde con el lado derecho de alguna producción. Las subcadenas b y d sirven. Elíjase la b situada más a la izquierda y sustitúyase por A, el lado izquierdo de la producción $A \rightarrow b$; así se obtiene la cadena *aAbcde*. A continuación, las subcadenas Abc, b y d concuerdan con el lado derecho de alguna producción. Aunque b es la subcadena situada más a la izquierda que concuerda con el lado derecho de una producción, se elige sustituir la subcadena Abc por A, que es el lado derecho de la producción $A \rightarrow Abc$. Se obtiene ahora *aAde*. Sustituyendo después d por B, que es el lado izquierdo de la producción $B \rightarrow d$, se obtiene *aABe*. Ahora se puede sustituir toda esta cadena por S. Por tanto, mediante una secuencia de cuatro reducciones se puede reducir *abbcde* a S. De hecho, estas reducciones trazan la siguiente derivación por la derecha en orden inverso:

$$S \underset{rd}{\Rightarrow} aABe \underset{rd}{\Rightarrow} aAde \underset{rd}{\Rightarrow} aAbcde \underset{rd}{\Rightarrow} abbcde$$ □

Mangos

Informalmente, un "mango" de una cadena es una subcadena que concuerda con el lado derecho de una producción y cuya reducción al no terminal del lado izquierdo de la producción representa un paso a lo largo de la inversa de una derivación por la derecha. En muchos casos, la subcadena situada más a la izquierda β que concuerda con el lado derecho de alguna producción $A \rightarrow \beta$ no es un mango, porque una reducción por la producción $A \rightarrow \beta$ produce una cadena no reducible al símbolo inicial. En el ejemplo 4.21, si se sustituyera b por A en la segunda cadena *aAbcde* se obtendría la cadena *aAAcde* que no se puede reducir posteriormente a S. Por esta razón, se debe dar una definición más precisa de un mango.

Formalmente, un *mango* de una forma de frase derecha γ es una producción $A \rightarrow \beta$ y una posición de γ donde la cadena β podría encontrarse y sustituirse por A para producir la forma de frase derecha previa en una derivación por la derecha de γ. Es decir, si $S \overset{*}{\underset{rd}{\Rightarrow}} \alpha Aw \overset{*}{\underset{rd}{\Rightarrow}} \alpha\beta w$, entonces $A \rightarrow \beta$ si la posición que sigue de α es un mango de $\alpha\beta w$. La cadena w a la derecha del mango contiene sólo símbolos terminales. Obsérvese que dice "un mango" en lugar de "el mango", porque la gramática podría ser ambigua, con más de una derivación por la derecha de $\alpha\beta w$. Si una gramática no es ambigua, entonces toda forma de frase derecha de la gramática tiene exactamente un mango.

En el ejemplo anterior, *abbcde* es una forma de frase derecha cuyo mango es $A \rightarrow b$ en la posición 2. Del mismo modo, *aAbcde* es una forma de frase derecha cuyo mango es $A \rightarrow Abc$ en la posición 2. Algunas veces se dice "la subcadena β es un mango de $\alpha\beta w$" si están claras la posición de β y la producción $A \rightarrow \beta$ que se tienen en mente.

En la figura 4.20 se representa el mango $A \rightarrow \beta$ en el árbol de análisis sintáctico de una forma de frase derecha $\alpha\beta w$. El mango representa al subárbol completo situado más a la izquierda que consta de un nodo y todos sus hijos. En la figura 4.20, A es el nodo interior situado más abajo y más a la izquierda con todos sus hijos en el árbol. Se puede considerar como "poda del mango", es decir, eliminación de los hijos de A del árbol de análisis sintáctico.

Ejemplo 4.22. Considérese la siguiente gramática:

$$\begin{array}{ll} (1) & E \rightarrow E + E \\ (2) & E \rightarrow E * E \\ (3) & E \rightarrow (E) \\ (4) & E \rightarrow \mathbf{id} \end{array} \tag{4.16}$$

y la derivación por la izquierda

$$\begin{aligned} E &\underset{izq}{\Rightarrow} \underline{E + E} \\ &\underset{izq}{\Rightarrow} E * \underline{E * E} \\ &\underset{izq}{\Rightarrow} E + E * \underline{\mathbf{id}_3} \\ &\underset{izq}{\Rightarrow} E + \underline{\mathbf{id}_2} * \mathbf{id}_3 \\ &\underset{izq}{\Rightarrow} \underline{\mathbf{id}_1} + \mathbf{id}_2 * \mathbf{id}_3 \end{aligned}$$

Fig. 4.20. El mango $A \rightarrow \beta$ en el árbol de análisis sintáctico para $\alpha\beta w$.

Para facilitar la notación se han puesto subíndices a los símbolos **id** y se ha subrayado un mango de cada forma de frase derecha. Por ejemplo, $\mathbf{id}_1$ es un mango de la forma de frase derecha $\mathbf{id}_1 + \mathbf{id}_2 * \mathbf{id}_3$, porque **id** es el lado derecho de la producción

$E \rightarrow \mathbf{id}$, y sustituir $\mathbf{id}_1$ por E produce la forma de frase derecha previa $E + \mathbf{id}_2 * \mathbf{id}_3$. Obsérvese que la cadena que aparece a la derecha de un mango contiene sólo símbolos terminales.

Puesto que la gramática (4.16) es ambigua, hay otra derivación por la derecha de la misma cadena:

$$\begin{aligned} E &\underset{rd}{\Rightarrow} \underline{E * E} \\ &\underset{rd}{\Rightarrow} E * \underline{\mathbf{id}_3} \\ &\underset{rd}{\Rightarrow} \underline{E + E} * \mathbf{id}_3 \\ &\underset{rd}{\Rightarrow} E + \underline{\mathbf{id}_2} * \mathbf{id}_3 \\ &\underset{rd}{\Rightarrow} \underline{\mathbf{id}_1} + \mathbf{id}_2 * \mathbf{id}_3 \end{aligned}$$

Considérese la forma de frase derecha $E + E * \mathbf{id}_3$. En esta derivación, $E + E$ es un mango de $E + E * \mathbf{id}_3$, mientras que $\mathbf{id}_3$ por sí mismo es un mango de esta misma forma de frase derecha según la derivación anterior.

Las dos derivaciones por la derecha de este ejemplo son análogas a las dos derivaciones por la izquierda del ejemplo 4.6. La primera derivación le da a * una mayor precedencia que a +, mientras que la segunda le da a + la mayor precedencia. □

Poda

Se puede obtener una derivación por la derecha en orden inverso mediante la "poda de mangos". Es decir, se comienza con una cadena de terminales w que se desee analizar sintácticamente. Si w es una frase de la gramática en cuestión, entonces $w = \gamma_n$, donde γ_n es la n-ésima forma de frase derecha de una, aún desconocida, derivación por la derecha.

$$S = \gamma_0 \underset{rd}{\Rightarrow} \gamma_1 \underset{rd}{\Rightarrow} \gamma_2 \underset{rd}{\Rightarrow} \cdots \underset{rd}{\Rightarrow} \gamma_{n-1} \underset{rd}{\Rightarrow} \gamma_n = w.$$

Para reconstruir esta derivación en orden inverso, se coloca el mango β_n en γ_n y se reemplaza β_n por el lado izquierdo de alguna producción $A_n \rightarrow \beta_n$ para obtener la $(n-1)$-ésima forma de frase derecha γ_{n-1}. Obsérvese que aún no se sabe cómo encontrar los mangos, pero pronto se verán los métodos para hacerlo.

Después se repite este proceso. Es decir, se sitúa el mango β_{n-1} en γ_{n-1} y se reduce este mango para obtener la forma de frase derecha γ_{n-2}. Si al continuar este proceso se produce una forma de frase derecha que conste sólo del símbolo inicial S, entonces se para y se anuncia la realización con éxito del análisis sintáctico. La inversa de la secuencia de producciones utilizada en estas reducciones es una derivación por la derecha de la cadena de entrada.

Ejemplo 4.23. Considérese la gramática (4.16) del ejemplo 4.22 y la cadena de entrada $\mathbf{id}_1 + \mathbf{id}_2 * \mathbf{id}_3$. La secuencia de reducciones que se muestra en la figura 4.21 reduce $\mathbf{id}_1 + \mathbf{id}_2 * \mathbf{id}_3$ al símbolo inicial E. Obsérvese que la secuencia de formas de frase derecha de este ejemplo es precisamente la inversa de la secuencia de la primera derivación por la derecha del ejemplo 4.22. □

FORMA DE FRASE DERECHA	MANGO	PRODUCCIÓN DE REDUCCIÓN
$\mathbf{id}_1 + \mathbf{id}_2 * \mathbf{id}_3$	$\mathbf{id}_1$	$E \rightarrow \mathbf{id}$
$E + \mathbf{id}_2 * \mathbf{id}_3$	$\mathbf{id}_2$	$E \rightarrow \mathbf{id}$
$E + E * \mathbf{id}_3$	$\mathbf{id}_3$	$E \rightarrow \mathbf{id}$
$E + E * E$	$E * E$	$E \rightarrow E * E$
$E + E$	$E + E$	$E \rightarrow E + E$
E		

Fig. 4.21. Reducciones realizadas por el analizador sintáctico por desplazamiento y reducción.

Implantación por medio de una pila del análisis sintáctico por desplazamiento y reducción

Hay dos problemas a resolver si se va a hacer el análisis sintáctico mediante poda. El primero consiste en situar la subcadena a reducir en una forma de frase derecha, y el segundo, en determinar qué producción elegir en caso de que haya más de una producción con dicha subcadena en el lado derecho. Antes de considerar estas cuestiones, considérese primero el tipo de estructuras de datos que se debe utilizar en un analizador sintáctico por desplazamiento y reducción.

Un modo adecuado de implantar un analizador sintáctico por desplazamiento y reducción es mediante la utilización de una pila para manejar los símbolos gramaticales, y un *buffer* de entrada para manejar la cadena w que se ha de analizar. Se utiliza \$ para marcar el fondo de la pila y el extremo derecho de la entrada. Al principio, la pila está vacía, y la cadena w está en la entrada, como sigue:

PILA	ENTRADA
\$	w \$

El analizador sintáctico funciona desplazando cero o más símbolos de la entrada a la pila hasta que un mango β esté en su cima. Entonces, el analizador reduce β al lado izquierdo de la producción adecuada. El analizador repite este lazo hasta que detecta un error o hasta que la pila contiene el símbolo inicial y la entrada está vacía:

PILA	ENTRADA
\$$S$	\$

Después de esta configuración, el analizador se para y anuncia la terminación con éxito del análisis sintáctico.

Ejemplo 4.24. Hágase el recorrido paso a paso de las acciones que puede realizar un analizador sintáctico por desplazamiento y reducción para analizar la cadena de entrada $\mathbf{id}_1 + \mathbf{id}_2 * \mathbf{id}_3$ según la gramática (4.16), utilizando la primera derivación del ejemplo 4.22. La secuencia se muestra en la figura 4.22. Obsérvese que, cómo la gramática (4.16) tiene dos derivaciones por la derecha para esta entrada, existe otra secuencia de pasos que puede dar un analizador por desplazamiento y reducción. □

	PILA	ENTRADA	ACCIÓN
(1)	\$	$\mathbf{id}_1 + \mathbf{id}_2 * \mathbf{id}_3\$$	desplazar
(2)	$\$\mathbf{id}_1$	$+ \mathbf{id}_2 * \mathbf{id}_3\$$	reducir por $E \rightarrow \mathbf{id}$
(3)	$\$E$	$+ \mathbf{id}_2 * \mathbf{id}_3\$$	desplazar
(4)	$\$E +$	$\mathbf{id}_2 * \mathbf{id}_3\$$	desplazar
(5)	$\$E + \mathbf{id}_2$	$* \mathbf{id}_3\$$	reducir por $E \rightarrow \mathbf{id}$
(6)	$\$E + E$	$* \mathbf{id}_3\$$	desplazar
(7)	$\$E + E *$	$\mathbf{id}_3\$$	desplazar
(8)	$\$E + E * \mathbf{id}_3$	\$	reducir por $E \rightarrow \mathbf{id}$
(9)	$\$E + E * E$	\$	reducir por $E \rightarrow E * E$
(10)	$\$E + E$	\$	reducir por $E \rightarrow E + E$
(11)	$\$E$	\$	aceptar

Fig. 4.22. Configuraciones del analizador sintáctico por desplazamiento y reducción con la entrada $\mathbf{id}_1 + \mathbf{id}_2 * \mathbf{id}_3$.

Aunque las principales operaciones del analizador son el desplazamiento y la reducción, existen en realidad cuatro acciones posibles que un analizador por desplazamiento y reducción puede realizar: 1) desplazar, 2) reducir, 3) aceptar y 4) error.

1. En una acción de *desplazar*, el siguiente símbolo de entrada se desplaza a la cima de la pila.
2. En una acción de *reducir*, el analizador sabe que el extremo derecho del mango está en la cima de la pila. Entonces debe localizar el extremo izquierdo del mango dentro de la pila y decidir el no terminal con qué debe sustituir el mango.
3. En una acción de *aceptar*, el analizador anuncia la terminación con éxito del análisis sintáctico.
4. En una acción de *error*, el analizador descubre que se ha producido un error sintáctico y llama a una rutina de recuperación de errores.

Hay un hecho importante que justifica el uso de una pila en el análisis sintáctico por desplazamiento y reducción: el mango siempre aparecerá en la cima de la pila, nunca dentro. Esto resulta obvio cuando se consideran las formas posibles de dos pasos sucesivos en cualquier derivación por la derecha. Estos dos pasos pueden ser de la forma

(1) $S \overset{*}{\underset{rd}{\Rightarrow}} \alpha Az \overset{*}{\underset{rd}{\Rightarrow}} \alpha\beta Byz \overset{*}{\underset{rd}{\Rightarrow}} \alpha\beta\gamma yz$

(2) $S \overset{*}{\underset{rd}{\Rightarrow}} \alpha BxAz \overset{*}{\underset{rd}{\Rightarrow}} \alpha Bxyz \overset{*}{\underset{rd}{\Rightarrow}} \alpha\gamma xyz$

En el caso (1), A se sustituye por βBy, y después, el no terminal situado más a la derecha B en ese lado derecho se sustituye por γ. En el caso (2), A se sustituye otra vez el primero, pero esta vez el lado derecho es una cadena y que consta sólo de terminales. El siguiente no terminal situado más a la derecha B estará en algún lugar a la izquierda de y.

Considérese el caso (1) en orden inverso, donde un analizador sintáctico por desplazamiento y reducción acaba de alcanzar la configuración

PILA	ENTRADA
$\$\alpha\beta\gamma$	$yz\$$

El analizador reduce ahora el mango γ a B para alcanzar la configuración

PILA	ENTRADA
$\$\alpha\beta B$	$yz\$$

Como B es el no terminal más a la derecha en $\alpha\beta Byz$, el extremo derecho del mango de $\alpha\beta Byz$ no puede aparecer dentro de la pila. Por tanto, el analizador puede desplazar la cadena y sobre la pila para alcanzar la configuración

PILA	ENTRADA
$\$\alpha\beta By$	$z\$$

en la que βBy es el mango, que queda reducido a A.

En el caso (2), en la configuración

PILA	ENTRADA
$\$\alpha\gamma$	$xyz\$$

el mango γ está en la cima de la pila. Después de reducir el mango γ a B, el analizador puede desplazar la cadena xy para colocar el siguiente mango y en la cima de la pila:

PILA	ENTRADA
$\$\alpha Bxy$	$z\$$

El analizador reduce ahora y a A.

En ambos casos, después de realizar una reducción, el analizador tuvo que desplazar cero o más símbolos para colocar el mango siguiente en la cima de la pila. Nunca tuvo que buscar dentro de la pila para encontrar el mango. Este aspecto de podado es el que convierte una pila en una estructura de datos particularmente adecuada para aplicar un analizador sintáctico por desplazamiento y reducción. Aún queda por explicar cómo elegir las acciones para que el analizador por desplazamiento y reducción trabaje adecuadamente. Los analizadores sintácticos LR y por precedencia de operadores son dos de estas técnicas y se estudiarán en breve.

Prefijos viables

Los prefijos de las formas de frase derecha que pueden aparecer en la pila de un analizador sintáctico por desplazamiento y reducción se denominan *prefijos viables*. Una definición equivalente de un prefijo viable es la de que es un prefijo de una forma de frase derecha que no continúa más allá del extremo derecho del mango situado más a la derecha de esta forma de frase. Con esta definición, siempre es posible añadir símbolos terminales al final de un prefijo viable para obtener una forma de frase derecha. Por tanto, aparentemente no hay error siempre que la porción examinada de la entrada hasta un punto dado pueda reducirse a un prefijo viable.

Conflictos durante el análisis sintáctico por desplazamiento y reducción

Existen gramáticas independientes del contexto para las cuales no se pueden utilizar analizadores sintácticos por desplazamiento y reducción. Todo analizador por desplazamiento y reducción para estas gramáticas puede alcanzar una configuración en la que el analizador sintáctico, conociendo el contenido total de la pila y el siguiente símbolo de entrada, no puede decidir si desplazar o reducir (un *conflicto de desplazamiento/reducción*), o no puede decidir qué tipo de reducción efectuar (un *conflicto de reducción/reducción*). A continuación se verán algunos ejemplos de construcciones sintácticas que dan lugar a dichas gramáticas. Técnicamente, estas gramáticas no están dentro de la clase LR(k) de gramáticas definida en la sección 4.7; se les denomina gramáticas no LR. La k de LR(k) se refiere al número de símbolos de preanálisis sobre la entrada. Por lo general, las gramáticas utilizadas en compilación se incluyen en la clase LR(1), con un símbolo de anticipación.

Ejemplo 4.25. Una gramática ambigua no puede ser nunca LR. Por ejemplo, considérese la gramática (4.7) de **else** ambiguo de la sección 4.3:

prop → **if** *expr* **then** *prop*
| **if** *expr* **then** *prop* **else** *prop*
| **otro**

Si se tiene un analizador sintáctico por desplazamiento y reducción en la configuración

PILA	ENTRADA
. . . **if** *expr* **then** *prop*	**else** . . . $

no se puede saber si **if** *expr* **then** *prop* es el mango, independientemente de lo que aparezca debajo de él en la pila. Aquí hay un conflicto de desplazamiento/reducción. Dependiendo de lo que siga al **else** de la entrada, puede ser correcto reducir **if** *expr* **then** *prop* a *prop*, o puede ser correcto desplazar **else** para luego buscar otra *prop* para completar la alternativa **if** *expr* **then** *prop* **else** *prop*. Por tanto, no se puede saber en este caso si desplazar o reducir, así que la gramática no es LR(1). En general, ninguna gramática ambigua, y ciertamente ésta lo es, puede ser LR(k) para ninguna k.

Sin embargo, se debe mencionar que el análisis sintáctico por desplazamiento y reducción se puede adaptar fácilmente para analizar algunas gramáticas ambiguas, como la gramática del **if-then-else** anterior. Cuando se construye uno de estos analizadores para una gramática que contenga las dos producciones anteriores, habrá un conflicto de desplazamiento/reducción: con **else**, o bien desplazar o reducir en *prop* → **if** *expr* **then** *prop*. Si el conflicto se resuelve en favor del desplazamiento, el analizador se comportará de manera natural. En la sección 4.8 se estudian los analizadores para dichas gramáticas ambiguas. □

Otra causa común de la falta de características LR se produce cuando se sabe que se tiene un mango, pero el contenido de la pila y el siguiente símbolo de entrada no son suficientes para determinar qué producción debe ser utilizada en una reducción. El siguiente ejemplo ilustra esta situación.

Ejemplo 4.26. Supóngase que se tiene un analizador léxico que devuelve el componente léxico **id** para todos los identificadores, independientemente de su uso. Supóngase también que el lenguaje invoca procedimientos dando sus nombres, con parámetros encerrados entre paréntesis, y que con la misma sintaxis se hace referencia a las matrices. Dado que la traducción de índices en las referencias a matrices y la de los parámetros en las llamadas a procedimientos es distinta, se utilizan distintas producciones para generar listas de parámetros actuales e índices. Por tanto, la gramática debe tener (entre otras) producciones como:

(1) *prop* → **id**(*lista_params*)
(2) *prop* → *expr* := *expr*
(3) *lista_params* → *lista_params*, *parámetro*
(4) *lista_params* → *parámetro*
(5) *parámetro* → **id**
(6) *expr* → **id**(*lista_expr*)
(7) *expr* → **id**
(8) *lista_expr* → *lista_expr*, *expr*
(9) *lista_expr* → *expr*

Una proposición que comience con `A(I,J)` parecería, para el analizador sintáctico, como la cadena de componentes léxicos **id(id,id)**. Después de desplazar los tres primeros componentes léxicos dentro de la pila, un analizador sintáctico por desplazamiento y reducción tendría la configuración

PILA	ENTRADA
. . . **id(id**	, **id**) . . .

Es evidente que se debe reducir el **id** de la cima de la pila, pero ¿por qué producción? La elección correcta es la producción (5) si `A` es un procedimiento y si `A` es una matriz, la elección correcta es la producción (7). La pila no dice cuál; se debe utilizar la información de la tabla de símbolos obtenida de la declaración de `A`.

Una solución es cambiar el componente léxico **id** en la producción (1) a **idproc** y usar un analizador léxico más complejo que devuelva el componente léxico **idproc** cuando reconozca un identificador que sea el nombre de un procedimiento. Esto exigiría que el analizador léxico consultara la tabla de símbolos antes de devolver un componente léxico.

Si se hiciera esta modificación, entonces al procesar `A(I,J)` el analizador sintáctico estaría en la configuración

PILA	ENTRADA
. . . **idproc (id**	, **id**) . . .

o en la configuración anterior. En el primer caso, se elige la reducción por la producción (5); en el segundo, por la producción (7). Obsérvese cómo el símbolo tercero de la cima de la pila determina qué reducción se debe hacer, aunque no esté implicado en la reducción. El análisis sintáctico por desplazamiento y reducción puede utilizar información que esté muy por debajo de la cima de la pila para guiar el análisis. □

4.6 ANALISIS SINTACTICO POR PRECEDENCIA DE OPERADORES

En la sección 4.7 se estudiará la mayor clase de gramáticas para las que se pueden construir con éxito analizadores sintácticos por desplazamiento y reducción (las gramáticas LR). Sin embargo, para una pequeña, pero importante, clase de gramáticas, se pueden construir con facilidad a mano eficientes analizadores sintácticos por desplazamiento y reducción. Estas gramáticas tienen la propiedad (entre otros requisitos fundamentales) de que ningún lado derecho de la producción es ϵ ni tiene dos no terminales adyacentes. Una gramática con esta última propiedad se denomina *gramática de operadores*.

Ejemplo 4.27. La siguiente gramática para expresiones

$$E \to EAE \mid (E) \mid -E \mid \mathbf{id}$$
$$A \to + \mid - \mid * \mid / \mid \uparrow$$

no es una gramática de operadores, porque el lado derecho EAE tiene dos (de hecho tres) no terminales consecutivos. Sin embargo, si se sustituye cada una de sus alternativas por A, se obtiene la siguiente gramática de operadores:

$$E \to E+E \mid E-E \mid E*E \mid E/E \mid E \uparrow E \mid (E) \mid -E \mid \mathbf{id} \tag{4.17}$$

A continuación se describe una técnica de análisis sintáctico fácil de implantar llamada análisis sintáctico por precedencia de operadores. Históricamente, la técnica se describió primero como una manipulación de componentes léxicos sin hacer referencia a ninguna gramática subyacente. De hecho, cuando se termina de construir un analizador sintáctico por precedencia de operadores a partir de una gramática, se puede efectivamente prescindir de la gramática, utilizando los no terminales de la pila tan sólo como indicadores de los atributos asociados a los no terminales.

Como técnica general de análisis sintáctico, el análisis por precedencia de operadores tiene varios inconvenientes. Por ejemplo, es difícil manejar componentes léxicos como el signo menos, que tiene dos precedencias distintas (dependiendo de si es unario o binario). Peor aún, como la relación entre una gramática para el lenguaje que está siendo analizado y el mismo analizador sintáctico por precedencia de operadores es muy frágil, no siempre se puede tener la seguridad de que el analizador acepta exactamente el lenguaje deseado. Por último, sólo una pequeña clase de gramática puede analizarse usando las técnicas de precedencia de operadores.

Sin embargo, dada su sencillez, se han construido con éxito muchos compiladores que utilizan las técnicas de análisis sintáctico por precedencia de operadores para expresiones. Con frecuencia, estos analizadores utilizan el descenso recursivo, descrito en la sección 4.4, para proposiciones y construcciones de alto nivel. Incluso se han construido analizadores sintácticos por precedencia de operadores para lenguajes completos.

En el análisis sintáctico por precedencia de operadores, se definen tres *relaciones de precedencia* disjuntas, $<\cdot$, $\doteq$, y $\cdot>$, entre algunos pares de terminales. Estas relaciones de precedencia guían la selección de mangos y tienen los siguientes significados:

RELACIÓN	SIGNIFICADO
$a <\cdot b$	a "cede la precedencia a" b
$a \doteq b$	a "tiene la misma precedencia que" b
$a \cdot> b$	a "tiene más precedencia que" b

Se debe prevenir al lector de que aunque estas relaciones pueden parecer similares a las relaciones aritméticas "menor que", "igual a" y "mayor que", las relaciones de precedencia tienen propiedades muy diferentes. Por ejemplo, se podría tener $a <\cdot b$ y $a \cdot> b$ para el mismo lenguaje, o podría no cumplirse ninguna de $a <\cdot b$, $a \doteq b$ y $a \cdot> b$ para algunos terminales a y b.

Hay dos maneras habituales de determinar qué relaciones de precedencia deben cumplirse entre un par de terminales. El primer método que se estudia es intuitivo y se basa en las nociones tradicionales de asociatividad y precedencia de operadores. Por ejemplo, si * tiene mayor precedencia que +, se hace $+ <\cdot *$ y $* \cdot> +$. Este método se estudiará para resolver las ambigüedades de la gramática (4.17) y permitirá escribir para ella un analizador sintáctico por precedencia de operadores (aunque el signo menos unitario cause problemas).

El segundo método para seleccionar las relaciones de precedencia de operadores consiste en construir primero una gramática no ambigua para el lenguaje, una gramática que refleje la asociatividad y precedencia correctas en sus árboles de análisis sintáctico. Esta tarea no es difícil para las expresiones; la sintaxis de las expresiones de la sección 2.2 proporciona el paradigma. Para la otra fuente habitual de ambigüedad, el **else** ambiguo, la gramática (4.9) es un modelo útil. Una vez obtenida una gramática no ambigua, existe un método mecánico para, a partir de ella, construir las relaciones de precedencia de operadores. Estas relaciones podrían no ser disjuntas, y podrían analizar un lenguaje distinto del generado por la gramática, pero con las clases estándar de expresiones aritméticas, se encuentran pocos problemas en la práctica. Estas construcciones no se estudiarán aquí; véase Aho y Ullman [1972b].

Uso de las relaciones de precedencia de operadores

La intención de las relaciones de precedencia es delimitar el mango de una forma de frase derecha, con $<\cdot$ marcando el extremo izquierdo, $\doteq$ apareciendo en el interior del mango y $\cdot>$ marcando el extremo derecho. Para mayor precisión, supóngase que se tiene una forma de frase derecha de una gramática de operadores. El hecho de que no aparezcan no terminales adyacentes en los lados derechos de las producciones supone que tampoco ninguna forma de frase derecha tendrá dos no terminales adyacentes. Por tanto, se puede escribir la forma de frase derecha como $\beta_0 a_1 \beta_1 \ldots a_n \beta_n$, en donde cada β_i es o ϵ (la cadena vacía) o un solo no terminal, y cada a_i es un solo terminal.

Supóngase que entre a_i y a_{i+1} se cumple exactamente una de las relaciones $<\cdot$, $\doteq$, o $\cdot>$. Además, úsese \$ para marcar cada extremo de la cadena y defínanse $\$ <\cdot b$ y $b \cdot> \$$ para todos los terminales b. Ahora supóngase que se eliminan los no terminales de la cadena y se coloca la relación correcta $<\cdot$, $\doteq$ o $\cdot>$, entre cada par de terminales y entre los terminales de los extremos y los símbolos \$ que marcan los

finales de la cadena. Por ejemplo, supóngase que inicialmente se tiene la forma de frase derecha **id** + **id** * **id** y que las relaciones de precedencia son las de la figura 4.23. Estas relaciones son algunas de las que se podrían escoger para analizar según la gramática (4.17).

	id	+	*	$
id		$\cdot>$	$\cdot>$	$\cdot>$
+	$<\cdot$	$\cdot>$	$<\cdot$	$\cdot>$
*	$<\cdot$	$\cdot>$	$\cdot>$	$\cdot>$
$	$<\cdot$	$<\cdot$	$<\cdot$	

Fig. 4.23. Relaciones de precedencia de operadores.

Entonces, la cadena con las relaciones de precedencia insertadas es:

$$\$ <\cdot \textbf{id} \cdot> + <\cdot \textbf{id} \cdot> * <\cdot \textbf{id} \cdot> \$ \tag{4.18}$$

Por ejemplo, se inserta $<\cdot$ entre el $ de la izquierda e **id** puesto que $<\cdot$ es la entrada en la fila $ y la columna **id**. Se puede encontrar el mango mediante el siguiente proceso:

1. Examínese la cadena desde el extremo izquierdo hasta encontrar el primer $\cdot>$. En (4.18), esto ocurre entre el primer **id** y +.
2. Después, examínese hacia atrás (a la izquierda) saltando sobre los $\doteq$ hasta encontrar un $<\cdot$. En (4.18) se examina hacia atrás hasta el símbolo $.
3. El mango contiene todo lo que esté a la izquierda del primer $\cdot>$ y a la derecha del $<\cdot$ encontrado en el paso 2, incluidos los no terminales intermedios o que rodeen a $<\cdot$ y $\cdot>$. (Es necesaria la inclusión de los no terminales que rodean para que no aparezcan dos no terminales adyacentes en una forma de frase derecha.) En (4.18), el mango es el primer **id**.

Si se trabaja con la gramática (4.17), entonces se reduce **id** a E. Llegados a este punto, se tiene la forma de frase derecha $E+$**id***id. Después de reducir los restantes **id** a E con los mismos pasos, se obtiene la forma de frase derecha $E+E*E$. Considérese ahora la cadena $+*$ obtenida eliminando los no terminales. Si se insertan las relaciones de precedencia, se obtiene

$$\$ <\cdot + <\cdot * \cdot> \$$$

que indica que el extremo izquierdo del mango se encuentra entre + y * y que el extremo derecho se encuentra entre * y $. Estas relaciones de precedencia indican que, en la forma de frase derecha $E+E*E$, el mango es $E*E$. Obsérvese cómo las E que rodean * se convierten en parte del mango.

Como los no terminales no influyen en el análisis sintáctico, no hay que preocuparse por diferenciarlos entre sí. Se puede guardar un solo marcador "no terminal" en la pila de un analizador sintáctico por desplazamiento y reducción para indicar las posiciones para los valores de los atributos.

Por la exposición anterior podría parecer que se debe examinar toda la forma de frase derecha en cada paso para encontrar el mango. Este no es el caso si se usa una pila para almacenar los símbolos de entrada que ya han aparecido y si se utilizan las relaciones de precedencia para guiar las acciones de un analizador sintáctico. Si se cumplen las relaciones de precedencia $<\cdot$ o $\doteq$ entre el símbolo terminal más a la cima de la pila y el siguiente símbolo de entrada, el analizador sintáctico hace un desplazamiento; todavía no ha encontrado el extremo derecho del mango. Si se cumple la relación $\cdot>$, es necesaria una reducción. En este punto, el analizador ya ha encontrado el extremo derecho del mango y se pueden utilizar las relaciones de precedencia para encontrar el extremo izquierdo del mango en la pila.

Si no se cumple ninguna relación de precedencia entre un par de terminales (indicado con una entrada en blanco en la Fig. 4.23), entonces es que se ha detectado un error sintáctico y debe invocarse una rutina de recuperación del error, como se estudiará más adelante en esta sección. Las ideas anteriores pueden formalizarse mediante el siguiente algoritmo.

Algoritmo 4.5. Algoritmo de análisis sintáctico por precedencia de operadores.

Entrada. Una cadena de entrada w y una tabla de relaciones de precedencia.

Salida. Si w está bien formada, resulta una *estructura* de árbol de análisis sintáctico, con un no terminal de posición E que etiqueta todos los nodos interiores; de lo contrario, resulta una indicación de error.

Método. Inicialmente, la pila contiene \$, y el *buffer* de entrada, la cadena w\$. Para hacer el análisis sintáctico, se ejecuta el programa de la figura 4.24. □

```
(1)  apuntar ae al primer símbolo de w$;
(2)  repeat forever
(3)      if $ está en la cima de la pila y ae apunta a $ then
(4)          return
         else begin
(5)          sea a el símbolo terminal más a la cima de la pila y
                 sea b el símbolo apuntado por ae;
(6)          if a <· b o a ≐ b then begin
(7)              meter b en la pila;
(8)              avanzar ae al siguiente símbolo de entrada;
             end;
(9)          else if a ·> b then          /* reduce */
(10)             repeat
(11)                 extraer el elemento de la cima de la pila
(12)             until el terminal de la cima de la pila esté relacionado por <·
                     con el terminal más recientemente extraído de la pila.
(13)         else error ()
         end
```

Fig. 4.24. Algoritmo de análisis sintáctico por precedencia de operadores.

Obtención de relaciones de precedencia de operadores a partir de la asociatividad y la precedencia

Siempre se pueden crear relaciones de precedencia de operadores de la forma que se considere adecuada y esperar que el algoritmo de análisis sintáctico por precedencia de operadores funcione correctamente al guiarse por ellas. Para un lenguaje de expresiones aritméticas, como el generado por la gramática (4.17), se pueden usar las siguientes técnicas heurísticas para producir un conjunto adecuado de relaciones de precedencia. Obsérvese que la gramática (4.17) es ambigua, y las formas de frase derechas podrían tener muchos mangos. Se establecen las siguientes reglas para seleccionar mangos "apropiados" para reflejar un determinado conjunto de reglas de asociatividad y precedencia para operadores binarios.

1. Si el operador θ_1 tiene mayor precedencia que el operador θ_2, hágase $\theta_1 \cdot> \theta_2$ y $\theta_2 <\cdot \theta_1$. Por ejemplo, si * tiene mayor precedencia que + hágase * $\cdot>$ + y + $<\cdot$ * . Estas relaciones garantizan que, en una expresión de la forma $E+E*E+E$, el central $E*E$ será el mango que se reducirá primero.

2. Si θ_1 y θ_2 son operadores de igual precedencia (de hecho, pueden ser el mismo operador), entonces hágase $\theta_1 \cdot> \theta_2$ y $\theta_2 \cdot> \theta_1$ si los operadores son asociativos por la izquierda, o hágase $\theta_1 <\cdot \theta_2$ y $\theta_2 <\cdot \theta_1$ si son asociativos por la derecha. Por ejemplo, si + y − son asociativos por la izquierda, entonces hágase + $\cdot>$ +, + $\cdot>$ −, − $\cdot>$ − y − $\cdot>$ +. Si $\uparrow$ es asociativo por la derecha, entonces hágase $\uparrow <\cdot \uparrow$. Estas relaciones garantizan que para $E-E+E$ se seleccionará $E-E$ como mango y que para $E \uparrow E \uparrow E$ se seleccionará la última $E \uparrow E$.

3. Hágase $\theta <\cdot$ **id**, **id** $\cdot>$ θ, $\theta <\cdot$ (, ($<\cdot$ θ,) $\cdot>$ θ, $\theta \cdot>$), $\theta \cdot>$ \$, y \$ $<\cdot$ θ para todos los operadores θ. Hágase también

($\doteq$)	\$ $<\cdot$ (	\$ $<\cdot$ **id**
($<\cdot$ (	**id** $\cdot>$ \$	) $\cdot>$ \$
($<\cdot$ **id**	**id** $\cdot>$)	) $\cdot>$)

Estas reglas garantizan que tanto **id** como (E) se reducirán a E. Asimismo, \$ sirve como marcador de los extremos izquierdo y derecho, lo cual hace que los mangos se encuentren entre los dos símbolos \$ siempre que sea posible.

Ejemplo 4.28. La figura 4.25 contiene las relaciones de precedencia de operadores para la gramática (4.17), asumiendo que

1. $\uparrow$ tiene la mayor precedencia y es asociativo por la derecha,
2. * y / tienen la siguiente mayor precedencia y son asociativos por la izquierda, y
3. + y − tienen la menor precedencia y son asociativos por la izquierda.

(Los espacios en blanco señalan entradas de error.) El lector debe practicar con la tabla para comprobar si funciona correctamente, ignorando por el momento los problemas con el menos unario. Pruébese la tabla con la entrada **id** * (**id** $\uparrow$ **id**) − **id/id**, por ejemplo. □

	+	−	*	/	↑	id	(	)	$
+	·>	·>	<·	<·	<·	<·	<·	·>	·>
−	·>	·>	<·	<·	<·	<·	<·	·>	·>
*	·>	·>	·>	·>	<·	<·	<·	·>	·>
/	·>	·>	·>	·>	<·	<·	<·	·>	·>
↑	·>	·>	·>	·>	<·	<·	<·	·>	·>
id	·>	·>	·>	·>	·>			·>	·>
(	<·	<·	<·	<·	<·	<·	<·	≐	
)	·>	·>	·>	·>	·>			·>	·>
$	<·	<·	<·	<·	<·	<·	<·		

Fig. 4.25. Relaciones de precedencia de operadores.

Manejo de operadores unarios

Si se tiene un operador unario como ¬ (negación lógica), que no sea asimismo un operador binario, se puede incorporar al esquema anterior para crear las relaciones de precedencia de operadores. Suponiendo que ¬ sea un operador unario prefijo, se hace $\theta <\cdot \neg$ para cualquier operador θ, ya sea unario o binario. Se hace $\neg \cdot> \theta$, si ¬ tiene mayor precedencia que θ y si no, $\neg <\cdot \theta$. Por ejemplo, si ¬ tiene mayor precedencia que &, y & es asociativo por la izquierda, mediante dichas reglas se podría agrupar $E\&\neg E\&E$ como $(E\&(\neg E))\&E$. La regla para los operadores unarios postfijos es similar.

La situación cambia cuando se tiene un operador como el signo menos − que es tanto prefijo unario como infijo binario. Aun dando la misma precedencia a los signos menos unario y binario, la tabla de la figura 4.25 no conseguirá analizar correctamente cadenas como **id*** − **id**. El mejor método en este caso es usar el analizador léxico para diferenciar el menos unario del binario, haciendo que devuelva un componente léxico distinto cuando vea un menos unario. Lamentablemente, el analizador léxico no puede utilizar el examen por anticipado para diferenciar los dos; debe recordar el componente léxico anterior. En FORTRAN, por ejemplo, un signo menos es unario si el componente léxico anterior era un operador, un paréntesis izquierdo, una coma o un símbolo de asignación.

Funciones de precedencia

Los compiladores que usan analizadores sintácticos por precedencia de operadores no necesitan almacenar la tabla de relaciones de precedencia. En la mayoría de los casos, la tabla se puede codificar mediante dos *funciones de precedencia* f y g que transforman símbolos terminales en enteros. Se intenta seleccionar f y g de modo que, para los símbolos a y b,

1. $f(a) < g(b)$ siempre que $a <\cdot b$,
2. $f(a) = g(b)$ siempre que $a \doteq b$, y
3. $f(a) > g(b)$ siempre que $a \cdot> b$.

por tanto, se pueden determinar las relaciones de precedencia entre a y b mediante una comparación numérica entre $f(a)$ y $g(b)$. Sin embargo, obsérvese que las entradas de error en la matriz de precedencia están oscurecidas, ya que se cumple 1, 2 ó 3, independientemente de cómo sean $f(a)$ y $g(b)$. Por lo general, la pérdida de capacidad de detección de errores no se considera lo suficientemente seria como para impedir el uso de las funciones de precedencia cuando sea posible; se pueden seguir localizando errores cuando se llama a una reducción y no se pueda encontrar ningún mango.

No todas las tablas de relaciones de precedencia tienen funciones de precedencia que las codifiquen, pero en casos prácticos suelen existir dichas funciones.

Ejemplo 4.29. La tabla de precedencia de la figura 4.25 tiene el siguiente par de funciones de precedencia

	+	−	*	/	↑	(	)	id	$
f	2	2	4	4	4	0	6	6	0
g	1	1	3	3	5	5	0	5	0

Por ejemplo, * $<\cdot$ **id**, y f(*) $< g$(**id**). Obsérvese que f(**id**) $> g$(**id**) sugiere que **id** $>$ **id**; pero en realidad no se cumple ninguna relación de precedencia entre **id** e **id**. En la figura 4.25 se sustituyen de manera similar otras entradas de errores por una u otra relación de precedencia. □

Un método sencillo para encontrar las funciones de precedencia de una tabla, si existen dichas funciones, es el siguiente.

Algoritmo 4.6. Construcción de las funciones de precedencia.

Entrada. Una matriz de precedencia de operadores.

Salida. Las funciones de precedencia que representen a la matriz de entrada, o una indicación de que no existen.

Método.

1. Créense los símbolos f_a y g_a para cada a que sea un terminal o $.
2. Divídanse los símbolos creados en tantos grupos como sea posible, de manera que si $a \doteq b$, entonces f_a y g_b están en el mismo grupo. Obsérvese que quizás haya que poner símbolos en el mismo grupo, aunque no estén relacionados por $\doteq$. Por ejemplo, si $a \doteq b$ y $c \doteq b$, entonces f_a y f_c deben estar en el mismo grupo, puesto que ambos están en el mismo grupo que g_b. Si además $c \doteq b$, entonces f_a y g_d están en el mismo grupo aunque $a \doteq d$ pueda no cumplirse.
3. Créese un grafo dirigido cuyos nodos sean los grupos encontrados en 2. Para todo a y b, si $a <\cdot b$, colóquese una arista desde el grupo de g_b al grupo de f_a. Si $a \cdot> b$, colóquese una arista desde el grupo de f_a al de g_b. Obsérvese que una arista o camino desde f_a a g_b significa que $f(a)$ debe sobrepasar a $g(b)$; un camino desde g_b a f_a significa que $g(b)$ debe sobrepasar a $f(a)$.

4. Si el grafo construido en 3 tiene un ciclo, entonces no existen funciones de precedencia. Si no hay ciclos, sea $f(a)$ la longitud del camino más largo que comienza en el grupo de f_a; sea $g(a)$ la longitud del camino más largo desde el grupo de g_a. □

Ejemplo 4.30. Considérese la matriz de la figura 4.23. No hay relaciones $\doteq$, de modo que cada símbolo está solo en un grupo. En la figura 4.26 se muestra el grafo construido utilizando el algoritmo 4.6.

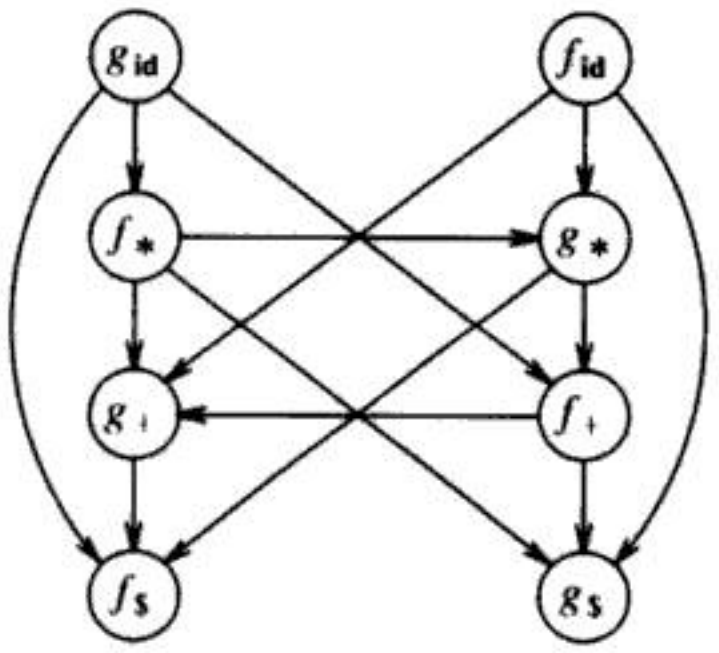


Fig. 4.26. Grafo que representa las funciones de precedencia.

No hay ciclos, así que existen las funciones de precedencia. Como $f_\$$ y $g_\$$ no tienen aristas de salida, $f(\$) = g(\$) = 0$. El camino más largo desde g_+ tiene longitud 1, de modo que $g(+) = 1$. Hay un camino desde g_{id} a f_* a g_* a f_+ a $f_\$$, de modo que $g(\mathbf{id}) = 5$. Las funciones de precedencia así obtenidas son

	+	*	id	$
f	2	4	4	0
g	1	3	5	0

□

Recuperación de errores en el análisis sintáctico por precedencia de operadores

Existen dos puntos en los procesos de análisis sintáctico en los que un analizador por precedencia de operadores puede descubrir errores sintácticos:

1. Si no se cumple ninguna relación de precedencia entre el terminal de la cima de la pila y la entrada en curso[1].
2. Si se ha encontrado un mango, pero no existe ninguna producción con este mango como lado derecho.

Recuérdese que el algoritmo de análisis sintáctico por precedencia de operadores (Algoritmo 4.5) aparece como si redujese mangos compuestos sólo por terminales.

[1] En compiladores que usan funciones de precedencia para representar las tablas de precedencia, esta fuente de detección de error puede no estar disponible.

Sin embargo, aunque se consideran los no terminales anónimamente, siguen teniendo su lugar en la pila de análisis sintáctico. Así que cuando en 2 se habla de un mango que concuerda con el lado derecho de una producción, se entiende que los terminales son los mismos al igual que las posiciones ocupadas por los no terminales.

Se debe observar que, además de los puntos 1 y 2 anteriores, no existen otros en los que se puedan detectar errores. Cuando se examina la pila para encontrar el extremo izquierdo del mango en los pasos (10 al 12) de la figura 4.24, el algoritmo de análisis sintáctico por precedencia de operadores, se tiene la seguridad de encontrar una relación $<\cdot$, puesto que \$ marca el fondo de la pila y está relacionada por $<\cdot$ con cualquier símbolo que pudiera aparecer inmediatamente por encima de él en la pila. Obsérvese asimismo que nunca se admiten símbolos adyacentes en la pila de la figura 4.24, a menos que estén relacionados por $<\cdot$ o $\doteq$. Por tanto, los pasos (10 al 12) deben tener éxito al hacer una reducción.

El hecho de encontrar una secuencia de símbolos $a <\cdot b_1 \doteq b_2 \doteq \ldots \doteq b_k$ en la pila no significa, sin embargo, que $b_1 b_2 \ldots b_k$ sea la cadena de símbolos terminales del lado derecho de una producción. No se exigió esta condición en la figura 4.24, pero es evidente que se puede, y de hecho se debe, hacer si se quieren asociar reglas semánticas a reducciones. Por tanto, se tiene la oportunidad de detectar errores en la figura 4.24, modificada en los pasos (10 al 12) para determinar qué producción es el mango en una reducción.

Manejo de errores durante las reducciones

Se puede dividir la rutina de detección y recuperación de errores en varias partes. Una parte maneja los errores del tipo 2. Por ejemplo, esta rutina puede extraer símbolos de la pila, como en los pasos 10 al 12 de la figura 4.24. Sin embargo, como no hay ninguna producción por la cual reducir, no se toman acciones semánticas, sino que se imprime un mensaje de diagnóstico. Para determinar lo que debe establecer el diagnóstico, la rutina que maneja el caso 2 debe decidir a qué producción se "parece" el lado derecho que se está extrayendo. Por ejemplo, supóngase que se saca *abc*, y no hay lado derecho de producción que conste de *a*, *b* y *c* junto con cero o más no terminales. Entonces se debe considerar si la eliminación de uno de *a*, *b* o *c* produce un lado derecho legal (omitiendo los no terminales). Por ejemplo, si hubiera un lado derecho *aEcE*, se podría emitir el diagnóstico

`hay una` *b* `ilegal en la línea` (línea que contiene *b*)

También se puede considerar el cambio o inserción de un terminal. Así, si *abEdc* fuera un lado derecho, se podría emitir el diagnóstico

`falta una` *d* `en la línea` (línea que contiene *c*)

Asimismo, puede ocurrir que haya un lado derecho con la secuencia apropiada de terminales, pero con el patrón de no terminales erróneo. Por ejemplo, si se saca *abc* de la pila sin no terminales internos o que lo rodeen, y *abc* no es un lado derecho pero sí lo es *aEbc*, se puede emitir el diagnóstico

`falta un` *E* `en la línea` (línea que contiene *b*)

Aquí, E representa una categoría sintáctica apropiada señalada por el no terminal E. Por ejemplo, si a, b o c es un operador, se puede decir "expresión"; si a es una palabra clave como **id**; se puede decir "condicional".

En general, la dificultad de determinar los diagnósticos adecuados cuando no se encuentra ningún lado derecho depende de si hay un número finito o infinito de posibles cadenas que se puedan extraer en las líneas (10 a 12) de la figura 4.24. Todas estas cadenas $b_1b_2 \ldots b_k$ deben tener relaciones $\doteq$ que se cumplan entre símbolos adyacentes, de modo que $b_1 \doteq b_2 \doteq \ldots \doteq b_k$. Si una tabla de precedencia de operadores indica que sólo hay un número finito de secuencias de terminales relacionados por $\doteq$, entonces se pueden manejar estas secuencias caso por caso. Para cada una de estas cadenas x se puede determinar con antelación un lado derecho legal y de distancia mínima y emitir un diagnóstico que indique que se encontró x cuando se esperaba y.

Es fácil determinar todas las cadenas que podrían extraerse de la pila en los pasos (10 al 12) de la figura 4.24. Estas son evidentes en el grafo dirigido cuyos nodos representan los terminales, con una arista desde a a b si, y sólo si, $a \doteq b$. En este caso, las posibles cadenas son las etiquetas de los nodos a lo largo de los caminos de este grafo. Es posible que haya caminos con un solo nodo. Sin embargo, para que un camino $b_1b_2 \ldots b_k$ sea extraíble en una entrada, debe haber un símbolo a (posiblemente \$) tal que $a <\cdot b_1$. Llámese a este b_1 *inicial*. Asimismo, debe haber un símbolo c (posiblemente \$) tal que $b_k \cdot> c$. Llámese a b_k *final*. Sólo entonces se puede llamar a una reducción y $b_1b_2 \ldots b_k$ sería la secuencia de símbolos extraídos. Si el grafo tiene un camino desde un nodo inicial a uno final que contenga un ciclo, entonces hay infinidad de cadenas que pueden extraerse; de lo contrario, hay sólo un número finito.

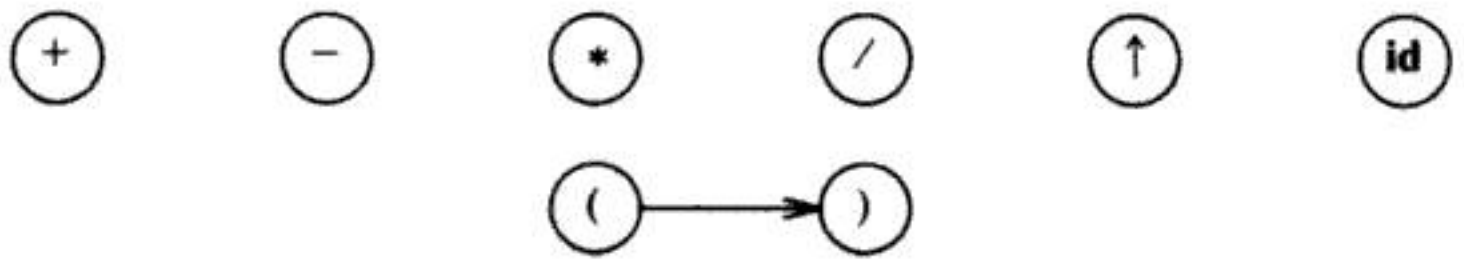


Fig. 4.27. Grafo para la matriz de precedencia de la figura 4.25.

Ejemplo 4.31. Reconsidérese la gramática (4.17):

$$E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid E \uparrow E \mid (E) \mid -E \mid \mathbf{id}$$

En la figura 4.25, se mostró la matriz de precedencia para esta gramática y su grafo aparece en la figura 4.27. Sólo hay una arista, porque el único par relacionado por $\doteq$ es el de los paréntesis izquierdo y derecho. Todos son iniciales, excepto el paréntesis derecho, y todos son finales, excepto el paréntesis izquierdo. Por tanto, los únicos caminos desde un nodo inicial a uno final son los caminos $+$, $-$, *, /, **id**, y $\uparrow$ de longitud uno, y el camino de (a) de longitud dos. Sólo hay un número finito y cada uno corresponde a los terminales del lado derecho de una producción en la gramática. Por tanto, el revisor de errores en las reducciones sólo necesita comprobar que el conjunto apropiado de marcadores no terminales aparezca entre las cadenas de terminales sujetas a reducción. Específicamente, el revisor hace lo siguiente:

1. Si se reduce $+$, $-$, *, / o $\uparrow$, se asegura de que aparezcan no terminales a ambos lados. Si no, envía el diagnóstico

   `falta operando`

2. Si se reduce **id**, comprueba que no haya no terminales a la derecha o a la izquierda. Si los hay, puede avisar

   `falta operador`

3. Si se reduce ( ), comprueba que haya un no terminal entre los paréntesis. Si no lo hay, puede indicar

   `no hay expresión entre los paréntesis`

También debe asegurarse de que no haya no terminales a ningún lado de los paréntesis. Si hay alguno, envía el mismo diagnóstico que en 2 □

Si puede extraerse una infinidad de cadenas, los mensajes de error no se pueden tabular caso por caso. Se puede utilizar una rutina general para determinar si el lado derecho de una producción está cerca (por ejemplo, a una distancia de 1 ó 2, donde la distancia se mide en función de componentes léxicos, en lugar de caracteres, insertos, eliminados o modificados de la cadena extraída, y en ese caso emitir un diagnóstico específico basándose en que se pretendía dicha producción. Si ninguna producción está cerca de la cadena extraída, se puede emitir un diagnóstico general a efectos de que "hay algo erróneo en la línea en curso".

Tratamiento de errores de desplazamiento/reducción

Ahora se estudia otra forma que tiene el analizador sintáctico por precedencia de operadores de detectar errores. Cuando se consulta la matriz de precedencias para decidir si desplazar o reducir [líneas (6) y (9) de la Fig. 4.24], se puede dar el caso de que no se cumpla ninguna relación entre el símbolo del tope de la pila y el primer símbolo de entrada. Por ejemplo, supóngase que a y b son los dos símbolos del tope de la pila (b está en el tope), c y d son los dos símbolos siguientes de entrada, y no hay relación de precedencia entre b y c. Para recuperar hay que modificar la pila, la entrada o ambas. Se pueden cambiar símbolos, insertar símbolos en la entrada o en la pila, o eliminar símbolos de la entrada o de la pila. Si se insertan o modifican, hay que tener cuidado de no caer en un lazo infinito, donde, por ejemplo, se insertan indefinidamente símbolos al principio de la entrada sin poder reducir o desplazar ninguno de los símbolos insertados.

Un método que asegura la inexistencia de lazos infinitos es garantizar que después de la recuperación se puede desplazar el símbolo en curso de entrada (si el símbolo en curso es \$, se garantiza que ningún símbolo se coloque en la entrada y se acorta eventualmente la pila). Por ejemplo, dado ab en la pila y cd en la entrada, si $a \lessdot c$[2], se puede extraer b de la pila. Otra opción es eliminar c de la entrada si $b \lessdot d$. Una tercera opción es encontrar un símbolo e tal que $b \lessdot e \lessdot c$ e insertar e delante de c en la entrada. En general, se debe insertar una cadena de símbolos tal que

$$b \lessdot e_1 \lessdot e_2 \lessdot \ldots \lessdot e_n \lessdot c$$

si no se pudiera encontrar un símbolo simple que insertar. La acción exacta elegida debe reflejar la intuición del diseñador del compilador en cuanto al error que puede aparecer en cada caso.

Para cada entrada en blanco en la matriz de precedencia hay que especificar una rutina de recuperación de errores; la misma rutina puede utilizarse en varios lugares. Cuando el analizador sintáctico consulta la entrada para a y b en el paso (6) de la figura 4.24 y no se cumple ninguna relación de precedencia entre a y b, encuentra un apuntador a la rutina de recuperación de errores para dicho error.

Ejemplo 4.32. Considérese de nuevo la matriz de precedencia de la figura 4.25. En la figura 4.28 se muestran las filas y las columnas de dicha matriz que tienen una o más entradas en blanco, y se han llenado estas entradas con los nombres de las rutinas para el manejo de errores.

	id	(	)	$
id	e3	e3	$\cdot>$	$\cdot>$
(	$<\cdot$	$<\cdot$	$\doteq$	e4
)	e3	e3	$\cdot>$	$\cdot>$
$	$<\cdot$	$<\cdot$	e2	e1

Fig. 4.28. Matriz de precedencia de operadores con entradas de error.

La base de estas rutinas para el tratamiento de errores es la siguiente:

```
e1: /* se llama cuando falta una expresión completa */
    insertar id en la entrada
    emitir el diagnóstico: "falta operando"

e2: /* se llama cuando las expresiones comienzan con un paréntesis derecho */
    eliminar ) de la entrada
    emitir el diagnóstico: "paréntesis derecho no equilibrado"

e3: /* se llama cuando id o ) va seguido de id o ( */
    insertar + en la entrada
    emitir el diagnóstico: "falta operador"

e4: /* se llama cuando las expresiones terminan con un paréntesis izquierdo */
    extraer ( de la pila
    emitir el diagnóstico: "falta paréntesis derecho"
```

Considérese cómo este mecanismo de manejo de errores trataría la entrada errónea **id** +). Las primeras acciones realizadas por el analizador sintáctico son despla-

[2] Se utiliza $\leqslant\cdot$ para representar $<\cdot$ o $\doteq$.

zar **id**, reducirlo a E (de nuevo se utiliza E para un no terminal anónimo en la pila), y después desplazar el signo $+$. Ahora se tiene la configuración

PILA	ENTRADA
$\$E+$	)$

Puesto que $+ \cdot >$), se solicita una reducción, y el mango es $+$. Se necesita el revisor de errores en las reducciones para que busque símbolos E a izquierda y derecha. Si comprueba que falta uno, emite el diagnóstico

```
falta operando
```

y realiza la reducción de todas maneras.

Ahora la configuración es

$\$E$	)$

No hay relación de precedencia entre $ y), y la entrada en la figura 4.28 para este par de símbolos es e2. La rutina e2 hace que se imprima el diagnóstico

```
paréntesis derecho no equilibrado
```

y elimina el paréntesis derecho de la entrada. Ahora queda la configuración final para el analizador sintáctico.

$\$E$	$	□

4.7 ANALIZADORES SINTACTICOS LR

En esta sección se analiza una técnica eficiente de análisis sintáctico ascendente que se puede utilizar para analizar una clase más amplia de gramáticas independientes del contexto. La técnica se denomina análisis sintáctico LR(k); la "L" es por el examen de la entrada de izquierda a derecha (en inglés, *left-to-right*), la "R" por construir una derivación por la derecha (en inglés, *rightmost derivation*) en orden inverso, y la k por el número de símbolos de entrada de examen por anticipado utilizados para tomar las decisiones del análisis sintáctico. Cuando se omite, se asume que k, es 1. El análisis sintáctico LR es atractivo por varias razones.

- Se pueden construir analizadores sintácticos LR para reconocer prácticamente todas las construcciones de los lenguajes de programación para los que se pueden escribir gramáticas independientes del contexto.
- El método de análisis sintáctico LR es el método de análisis por desplazamiento y reducción sin retroceso más general que se conoce, y sin embargo se puede aplicar tan eficientemente como los otros métodos de desplazamiento y reducción.
- La clase de gramáticas que pueden analizarse con los métodos LR es un supraconjunto de la clase de gramáticas que se pueden analizar con analizadores sintácticos predictivos.
- Un analizador sintáctico LR puede detectar un error sintáctico tan pronto como sea posible hacerlo en un examen de izquierda a derecha de la entrada.

El principal inconveniente del método es que supone demasiado trabajo construir un analizador sintáctico LR a mano para una gramática de un lenguaje de programación típico. Se necesita una herramienta especializada —un generador de analizadores sintácticos LR—. Por fortuna, existen disponibles estos generadores, y en la sección 4.9 se estudiará el diseño y uso de uno, el programa YACC. Con este generador se puede escribir una gramática independiente del contexto y el generador produce automáticamente un analizador sintáctico para dicha gramática. Si la gramática contiene ambigüedades u otras construcciones difíciles de analizar en un examen de izquierda a derecha de la entrada, el generador puede localizar dichas construcciones e informar al diseñador del compilador de su presencia.

Después de estudiar la operación de un analizador sintáctico LR, se introducen tres técnicas para construir una tabla de análisis sintáctico LR para una gramática. El primer método, llamado LR sencillo (SLR, en inglés) es el más fácil de implantar, pero el menos poderoso de los tres. Puede que no consiga producir una tabla de análisis sintáctico para algunas gramáticas que otros métodos sí consiguen. El segundo método, llamado LR canónico, es el más poderoso y costoso. El tercer método, llamado LR con examen por anticipado (LALR, en inglés), está entre los otros dos en cuanto a poder y costo. El método LALR funciona con las gramáticas de la mayoría de los lenguajes de programación y, con un poco de esfuerzo, se puede implantar en forma eficiente. En esta sección se consideran más adelante algunas técnicas para comprimir el tamaño de una tabla de análisis sintáctico LR.

El algoritmo de análisis sintáctico LR

En la figura 4.29 se muestra la forma esquemática de un analizador sintáctico LR. Consta de una entrada, una salida, una pila, un programa conductor y una tabla de análisis sintáctico con dos partes (*acción* e *ir_a*). El programa conductor es el mismo para todos los analizadores sintácticos LR; sólo cambian las tablas de un analizador a otro. El programa analizador lee caracteres de un *buffer* de entrada de uno en uno. El programa utiliza una pila para almacenar una cadena de la forma $s_0X_1s_1X_2s_2 \ldots X_ms_m$, donde s_m está en la cima. Cada X_i es un símbolo gramatical y cada s_i es un símbolo llamado *estado*. Cada símbolo de estado resume la información contenida

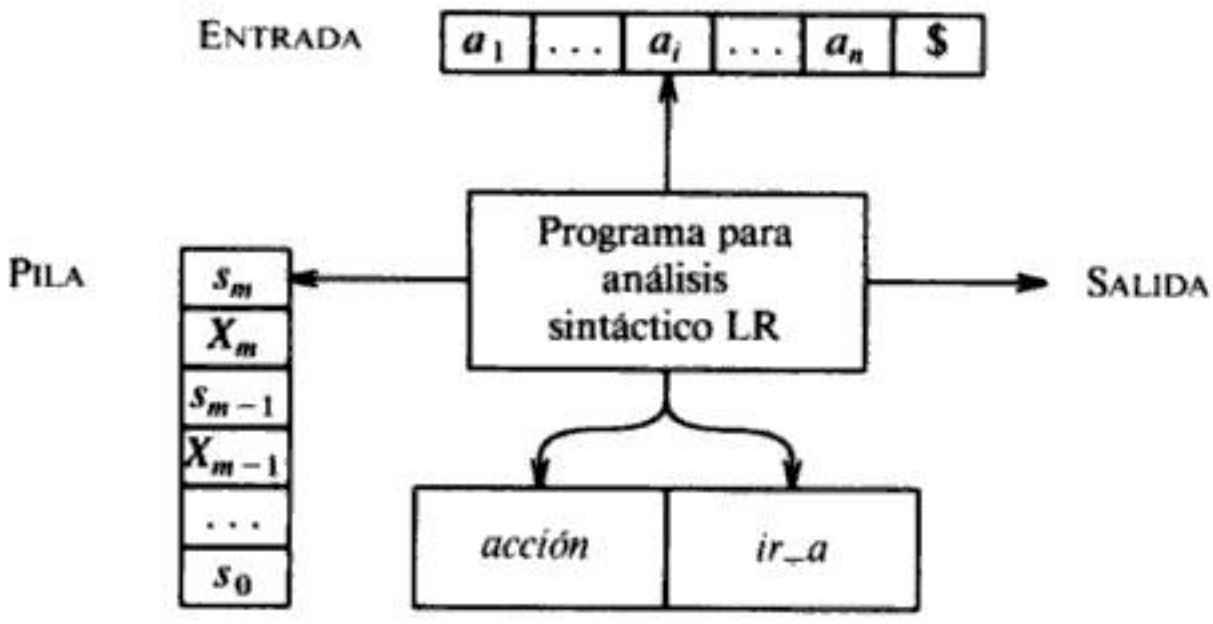


Fig. 4.29. Modelo de un analizador sintáctico LR.

debajo de él en la pila, y se usan la combinación del símbolo de estado en la cima de la pila y el símbolo en curso de la entrada para indexar la tabla de análisis sintáctico y determinar la decisión de desplazamiento a reducción del analizador. En una implantación no es necesario que los símbolos de la gramática aparezcan en la pila; sin embargo, siempre se incluirán en las siguientes exposiciones, para facilitar la explicación del comportamiento de un analizador sintáctico LR.

La tabla de análisis sintáctico consta de dos partes, la función *acción*, que indica una acción del analizador, y la función *ir_a*, que indica las transiciones entre estados. El programa que maneja el analizador sintáctico LR se comporta como sigue: determina s_m, el estado de la cima de la pila, y a_i, el símbolo en curso de la entrada. Después consulta la entrada *acción*$[s_m, a_i]$ de la tabla de acciones del analizador para el estado s_m, y la entrada a_i, que puede tener uno de estos cuatro valores:

1. desplazar s, donde s es un estado,
2. reducir por una producción gramatical $A \rightarrow \beta$,
3. aceptar y
4. error.

La función *ir_a* toma un estado y un símbolo gramatical como argumentos y produce un estado. Se verá que la función *ir_a* de una tabla de análisis sintáctico construida a partir de una gramática G utilizando el método SLR, LR canónico o LALR es la función de transiciones de un autómata finito determinista que reconoce los prefijos viables de G. Recuérdese que los prefijos viables de G son aquellos prefijos de formas de frase derecha que pueden aparecer en la pila de un analizador sintáctico por desplazamiento y reducción, porque no sobrepasan el mango situado más a la derecha. El estado inicial de esta AFD es el estado puesto inicialmente en la cima de la pila del analizador LR.

Una *configuración* de un analizador sintáctico LR es un par cuyo primer componente es el contenido de la pila, y el segundo, la entrada todavía sin procesar

$$(s_0\, X_1\, s_1\, X_2\, s_2 \ldots X_m\, s_m,\ a_i\, a_{i+1} \ldots a_n\$)$$

Esta configuración representa la forma de frase derecha

$$X_1\, X_2 \ldots X_m\, a_i\, a_{i+1} \ldots a_n$$

fundamentalmente de la manera en que lo haría un analizador sintáctico por desplazamiento y reducción; sólo es nueva la presencia de los estados en la pila.

El siguiente movimiento del analizador se determina leyendo a_i, el símbolo de la entrada en curso, y s_m, el estado del tope de la pila, y consultando después la entrada *acción*$[s_m, a_i]$ de la tabla de acciones del analizador. Las configuraciones obtenidas después de cada uno de los cuatro tipos de movimiento son las siguientes:

1. Si *acción*$[s_m, a_i]$ = desplazar s, el analizador ejecuta un movimiento de desplazamiento, entrando en la configuración

   $$(s_0\, X_1\, s_1\, X_2\, s_2 \ldots X_m\, s_m\, a_i\, s,\ a_{i+1} \ldots a_n\$)$$

   Aquí, el analizador ha desplazado a la pila al símbolo de entrada en curso a y al siguiente estado s, que está dado en *acción*$[s_m, a_i]$; a_{i+1} se convierte en el símbolo de entrada en curso.

2. Si *acción* $[s_m, a_i]$ = reducir $A \rightarrow \beta$, entonces el analizador ejecuta un movimiento de reducción, entrando en la configuración

$$(s_0 X_1 s_1 X_2 s_2 \ldots X_{m-r} s_{m-r} A s, a_i a_{i+1} \ldots a_n\$)$$

donde $s = ir\_a[s_{m-r}, A]$, y r es la longitud de β, el lado derecho de la producción. Aquí el analizador extrajo primero $2r$ símbolos de la pila (r símbolos de estados y r símbolos de la gramática), exponiendo el estado s_{m-r}. Luego introdujo A, el lado izquierdo de la producción, y s, la entrada de $ir\_a[s_{m-r}, A]$, en la pila. En un movimiento de reducción no se modifica el símbolo de entrada en curso. Para los analizadores LR que se construirán, $X_{m-r+1} \ldots X_m$, la secuencia de símbolos gramaticales extraídos de la pila, siempre concordarán con β, el lado derecho de la producción con que se efectúa la reducción.

   Después de un movimiento de reducción, se genera la salida de un analizador LR ejecutando la acción semántica asociada a la producción con que se efectúa la reducción. Por el momento, se asumirá que la salida consiste únicamente en imprimir la producción con que se efectúa la reducción.

3. Si *acción*$[s_m, a_i]$ = aceptar, el análisis sintáctico ha terminado.

4. Si *acción*$[s_m, a_i]$ = error, el analizador ha descubierto un error y llama a una rutina de recuperación de errores.

El algoritmo de análisis sintáctico LR se resume más adelante. Todos los analizadores sintácticos LR se comportan de esta forma; la única diferencia entre uno y otro es la informacion de los campos de acción y de transición de la tabla de análisis sintáctico.

Algoritmo 4.7. Algoritmo de análisis sintáctico LR.

Entrada. Una cadena de entrada w y una tabla de análisis sintáctico LR con las funciones *acción* e ir_a para la gramática G.

Salida. Si w está en $L(G)$, un análisis sintáctico ascendente de w; de lo contrario, se indica error.

Método. Inicialmente, s_0 está en la pila del analizador sintáctico, donde s_0 es el estado inicial, y $w\$$ está en el *buffer* de entrada. El analizador ejecuta entonces el programa de la figura 4.30 hasta encontrar una acción de aceptación o de error. □

Ejemplo 4.33. En la figura 4.31 se muestran las funciones *acción* e ir_a del análisis sintáctico de una tabla de análisis sintáctico LR para la siguiente gramática para expresiones aritméticas con los operadores binarios + y *:

(1) $E \rightarrow E + T$
(2) $E \rightarrow T$
(3) $T \rightarrow T * F$
(4) $T \rightarrow F$
(5) $F \rightarrow (E)$
(6) $F \rightarrow$ **id**

Los códigos de las acciones son:

1. d*i* significa desplazar y meter en la pila el estado *i*,
2. r*j* significa reducir por la producción con número *j*,
3. acep significa aceptar,
4. el espacio en blanco significa error.

```
apuntar ae al primer símbolo de w$;
repeat forever begin
    sea s el estado en la cima de la pila y
        a el símbolo apuntado por ae;
    if acción [s, a] = desplazar s' then begin
        meter a y después s' en la cima de la pila;
        avanzar ae al siguiente símbolo de entrada
    end
    else if acción [s, a] = reducir A → β then begin
        sacar 2* | β | símbolos de la pila;
        sea s' el estado que ahora está en la cima de la pila;
        meter A y después ir_a [s', A] en la cima de la pila;
        emitir la producción A → β
    end
    else if acción [s, a] = aceptar then
        return
    else error ()
end
```

Fig. 4.30. Programa para análisis sintáctico LR.

ESTADO	*acción*						*ir_a*		
	id	+	*	(	)	$	*E*	*T*	*F*
0	d5			d4			1	2	3
1		d6				acep			
2		r2	d7		r2	r2			
3		r4	r4		r4	r4			
4	d5			d4			8	2	3
5		r6	r6		r6	r6			
6	d5			d4				9	3
7	d5			d4					10
8		d6			d11				
9		r1	d7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Fig. 4.31. Tabla de análisis sintáctico para la gramática de expresiones.

Obsérvese que el valor de *ir_a*[*s*, *a*] para el terminal *a* se encuentra en el campo acción conectado con la acción de desplazar en la entrada *a* para el estado *s*. El campo de *ir_a* da *ir_a*[*s*, *A*] para los no terminales *A*. Asimismo, téngase en cuenta que aún no se ha explicado cómo se seleccionaron las entradas de la figura 4.31; este aspecto se considerará más adelante.

	PILA	ENTRADA	ACCIÓN
(1)	0	**id** * **id** + **id** $	desplazar
(2)	0 **id** 5	* **id** + **id** $	reducir por $F \rightarrow$ **id**
(3)	0 F 3	* **id** + **id** $	reducir por $T \rightarrow F$
(4)	0 T 2	* **id** + **id** $	desplazar
(5)	0 T 2 * 7	**id** + **id** $	desplazar
(6)	0 T 2 * 7 **id** 5	+ **id** $	reducir por $F \rightarrow$ **id**
(7)	0 T 2 * 7 F 10	+ **id** $	reducir por $T \rightarrow T*F$
(8)	0 T 2	+ **id** $	reducir por $E \rightarrow T$
(9)	0 E 1	+ **id** $	desplazar
(10)	0 E 1 + 6	**id** $	desplazar
(11)	0 E 1 + 6 **id** 5	$	reducir por $F \rightarrow$ **id**
(12)	0 E 1 + 6 F 3	$	reducir por $T \rightarrow F$
(13)	0 E 1 + 6 T 9	$	$E \rightarrow E + T$
(14)	0 E 1	$	aceptar

Fig. 4.32. Movimientos del analizador sintáctico LR con la entrada **id** * **id** + **id**.

Con la entrada **id** * **id** + **id**, en la figura 4.32 se muestra la secuencia de contenidos de la pila y de la entrada. Por ejemplo, en la línea (1) el analizador LR está en el estado 0, siendo **id** el primer símbolo de entrada. La acción en la fila 0 y columna **id** del campo acción de la figura 4.31 es d5, que significa desplazar y tapar la cima de la pila con el estado 5. Esto es lo que ha ocurrido en la línea (2): el primer componente léxico **id** y el símbolo del estado 5 han sido introducidos en la pila y se ha eliminado **id** de la entrada.

Entonces, * se convierte en el símbolo de entrada en curso y la acción del estado 5 con la entrada * es reducir por $F \rightarrow$ **id**. Se extraen dos símbolos de la pila (un símbolo de estado y un símbolo gramatical). El estado 0 queda expuesto en la cima de la pila. Como el ir a del estado 0 en F es 3, se introducen F y 3 en la pila. Ya se tiene la configuración de la línea (3). Los movimientos restantes se determinan de manera similar. □

Gramáticas LR

¿Cómo se construye una tabla de análisis sintáctico LR para una determinada gramática? Una gramática para la que se puede construir una tabla de análisis sintáctico se denomina *gramática LR*. Hay gramáticas independientes del contexto que

no son LR, pero en general se pueden evitar en las construcciones típicas de los lenguajes de programación. Intuitivamente, para que una gramática sea LR basta con que un analizador sintáctico por desplazamiento y reducción que opere de izquierda a derecha pueda reconocer los mangos cuando aparezcan en la cima de la pila.

Un analizador LR no tiene que examinar la pila completa para saber cuándo aparecen los mangos en la cima. Por el contrario, el símbolo del estado en la cima de la pila contiene toda la información necesaria. Es un hecho curioso que, si se puede reconocer un mango conociendo sólo los símbolos gramaticales de la pila, entonces existe un autómata finito que puede, leyendo los símbolos gramaticales de la pila de arriba a abajo, determinar el mango, si existe, que está en el tope de la pila. La función *ir_a* de una tabla de análisis sintáctico LR es esencialmente dicho autómata finito. Sin embargo, el autómata no necesita leer la pila para cada movimiento. El símbolo estado almacenado en la cima de la pila es el estado en que estaría el autómata finito reconocedor de los mangos si hubiera leído los símbolos gramaticales de la pila desde abajo hasta la cima. Por tanto, el analizador sintáctico LR puede determinar a partir del estado de la cima de la pila todo lo que se necesita saber sobre lo que hay en ella.

Otra fuente de información que puede utilizar un analizador LR como ayuda para tomar las decisiones de desplazamiento y reducción son los k símbolos siguientes de entrada. Los casos en que $k = 0$ o $k = 1$ tienen interés práctico, y aquí sólo se considerarán los analizadores sintácticos LR con $k \leq 1$. Por ejemplo, la tabla de acciones de la figura 4.31 utiliza un símbolo de examen por anticipado. Una gramática que se puede analizar mediante un analizador sintáctico LR que examina hasta k símbolos de entrada en cada movimiento se denomina *gramática LR*(k).

Existe una diferencia significativa entre las gramáticas LL y las LR. Para que una gramática sea LR(k), hay que ser capaz de reconocer la presencia del lado derecho de una producción, habiendo visto todo lo que deriva de dicho lado derecho con k símbolos de examen por anticipado. Este requisito es mucho menos riguroso que el de las gramáticas LL(k), donde hay que ser capaz de reconocer el uso de una producción viendo sólo los primeros k símbolos de los que se deriva su lado derecho. Por consiguiente, las gramáticas LR pueden describir más lenguajes que las gramáticas LL.

Construcción de tablas de análisis sintáctico SLR

A continuación se muestra cómo construir una tabla de análisis sintáctico SLR a partir de una gramática. Se darán tres métodos, que tienen distintos grados de poder y facilidad de aplicación. El primero, llamado "LR sencillo" (o *SLR*, en inglés), es el más débil de los tres en cuanto al número de gramáticas para las que funciona con éxito, pero es el más fácil de implantar. La tabla de análisis sintáctico construida con este método se denominará tabla SLR y un analizador LR que utilice una tabla de análisis SLR se denominará analizador sintáctico SLR. Una gramática para la que se pueda construir un analizador sintáctico SLR se denomina gramática SLR. Los otros dos métodos amplían el método SLR con información de examen por anticipado, así que el método SLR es un buen punto de partida para estudiar el análisis sintáctico LR.

Un *elemento del análisis sintáctico LR(0)* (*elemento*, para abreviar) de una gramática G es una producción de G con un punto en alguna posición del lado derecho. Por tanto, la producción $A \rightarrow XYZ$ produce los cuatro elementos

$$A \rightarrow \cdot XYZ$$
$$A \rightarrow X \cdot YZ$$
$$A \rightarrow XY \cdot Z$$
$$A \rightarrow XYZ \cdot$$

La producción $A \rightarrow \epsilon$ genera sólo un elemento, $A \rightarrow \cdot$. Un elemento se puede representar mediante dos enteros, el primero de los cuales da el número de la producción, y el segundo, la posición del punto. Intuitivamente, un elemento indica hasta dónde se ha visto una producción en un momento dado del proceso del análisis sintáctico. Por ejemplo, el primer elemento de arriba indica que se espera ver a continuación en la entrada una cadena derivable de XYZ. El segundo elemento indica que se acaba de ver en la entrada una cadena derivable de X y que a continuación se espera ver una cadena derivable de YZ.

La idea central del método SLR es construir primero a partir de la gramática un autómata finito determinista para reconocer los prefijos viables. Los elementos se agrupan en conjuntos, que dan lugar a los estados del analizador sintáctico SLR. Los elementos se pueden considerar como los estados de un AFN que reconoce los prefijos viables, y el "agrupamiento" es en realidad la construcción de subconjuntos estudiada en la sección 3.6.

Una serie de conjuntos de elementos LR(0), que se denomina colección *canónica* LR(0), proporciona la base para construir analizadores sintácticos SLR. Para construir la colección canónica LR(0) para una gramática, se define una gramática aumentada y dos funciones, *cerradura* e *ir_a*.

Si G es una gramática con símbolo inicial S, entonces G', la *gramática aumentada* para G, es G con un nuevo símbolo inicial S' y la producción $S' \rightarrow S$. El propósito de esta nueva producción inicial es indicar al analizador cuándo debe detener el análisis sintáctico y anunciar la aceptación de la cadena. Es decir, la aceptación se produce cuando, y sólo cuando, el analizador está a punto de reducir por $S' \rightarrow S$.

La operación cerradura

Si I es un conjunto de elementos para una gramática G, entonces *cerradura* (I) es el conjunto de elementos construido a partir de I por las dos reglas:

1. Inicialmente, todo elemento de I se añade a *cerradura* (I).
2. Si $A \rightarrow \alpha \cdot B\beta$ está en *cerradura* (I) y $B \rightarrow \gamma$ es una producción, entonces añádase el elemento $B \rightarrow \cdot\gamma$ a *cerradura* (I), si todavía no está ahí. Se aplica esta regla hasta que no se puedan añadir más elementos a *cerradura* (I).

Intuitivamente, si $A \rightarrow \alpha \cdot B\beta$ está en *cerradura* (I) indica que, en algún momento del proceso de análisis sintáctico, se cree posible ver a continuación una cadena derivable de B como entrada. Si $B \rightarrow \gamma$ es una producción, también se espera ver una subcadena derivable de γ en este punto. Por esta razón se incluye $B \rightarrow \cdot\gamma$ en *cerradura* (I).

Ejemplo 4.34. Considérese la gramática de expresiones aumentada:

$$\begin{aligned} E' &\to E \\ E &\to E + T \mid T \\ T &\to T * F \mid F \\ F &\to (E) \mid \mathbf{id} \end{aligned} \tag{4.19}$$

Si I es el conjunto de un elemento $\{[E' \to \cdot E]\}$, entonces *cerradura* (I) contiene los elementos

$$\begin{aligned} E' &\to \cdot E \\ E &\to \cdot E + T \\ E &\to \cdot T \\ T &\to \cdot T*F \\ T &\to \cdot F \\ F &\to \cdot(E) \\ F &\to \cdot \mathbf{id} \end{aligned}$$

Aquí, $E' \to \cdot E$ se coloca en *cerradura* (I) por la regla 1. Como hay una E inmediatamente a la derecha de un punto, por la regla 2 se añaden las producciones de E con puntos en el extremo izquierdo, es decir, $E \to \cdot E + T$ y $E \to \cdot T$. Ahora hay una T inmediatamente a la derecha de un punto, así que se añade $T \to \cdot T*F$ y $T \to \cdot F$. A continuación, la F a la derecha de un punto obliga a añadir $F \to \cdot(E)$ y $F \to \cdot \mathbf{id}$. Por la regla 2 no se colocan más elementos dentro de *cerradura* (I). □

Se puede calcular la función *cerradura* como se muestra en la figura 4.33. Una forma apropiada de implantar la función *cerradura* es mantener una matriz booleana *añadida*, indexada por los no terminales de G, de forma que a *añadida*$[B]$ se le asigna **true** siempre y cuando se añadan los elementos $B \to \cdot\gamma$ para cada producción de B $B \to \gamma$.

```
function cerradura ( I );
begin
    J := I;
    repeat
        for cada elemento A → α·Bβ en J y cada producción
            B → γ de G tal que B → ·γ no esté en J do
                añadir B → ·γ a J
    until no se puedan añadir más elementos a J;
    return J
end
```

Fig. 4.33. Cálculo de *cerradura*.

Obsérvese que si se añade una producción de B a la cerradura de I con el punto en el extremo izquierdo, entonces todas las producciones de B se añadirán de manera similar a la cerradura. De hecho, en algunas circunstancias no es realmente necesario listar los elementos $B \to \cdot\gamma$ añadidos a I por *cerradura*. Bastará con dar una

lista de los no terminales B cuyas producciones se añadieron así. De hecho, se pueden dividir todos los conjuntos de los elementos que interesan en dos clases de elementos.

1. *Elementos nucleares*, que incluyen el elemento inicial, $S' \rightarrow \cdot S$, y todos los elementos cuyos puntos no estén en el extremo izquierdo.
2. *Elementos no nucleares*, cuyos puntos están en el extremo izquierdo.

Además, cada conjunto de elementos que interesa se forma tomando la cerradura de un conjunto de elementos del núcleo; los elementos añadidos en la cerradura nunca pueden ser elementos, por supuesto. Entonces se pueden representar los conjuntos de elementos que realmente interesen con muy poca memoria si se desechan todos los elementos no nucleares, sabiendo que se podrían regenerar mediante el proceso de cerradura.

La operación ir_a

La segunda función útil es $ir\_a(I, X)$, dónde I es un conjunto de elementos y X es un símbolo de la gramática. Se define $ir\_a(I, X)$ como la cerradura del conjunto de todos los elementos $[A \rightarrow \alpha X \cdot \beta]$ tales que $[A \rightarrow \alpha \cdot X\beta]$ esté en I. Intuitivamente, si I es el conjunto de elementos válidos para algún prefijo viable, entonces $ir\_a(I, X)$ es el conjunto de elementos válidos para el prefijo viable γX.

Ejemplo 4.35. Si I es el conjunto de dos elementos $\{[E' \rightarrow E \cdot], [E \rightarrow E \cdot + T]\}$, entonces $ir\_a(I, +)$ consta de

$$E \rightarrow E + \cdot T$$
$$T \rightarrow \cdot T * F$$
$$T \rightarrow \cdot F$$
$$F \rightarrow \cdot (E)$$
$$F \rightarrow \cdot \mathbf{id}$$

Se calculó $ir\_a(I, +)$ examinando I para buscar elementos con $+$ inmediatamente a la derecha del punto. $E' \rightarrow E \cdot$ no es uno de estos elementos pero $E \rightarrow E \cdot + T$, sí. Se desplazó el punto más allá de $+$ para obtener $\{E \rightarrow E + \cdot T\}$ y después se tomó la cerradura de este conjunto. □

La construcción de conjuntos de elementos

Ahora ya se puede dar el algoritmo para construir C, la colección canónica de conjuntos de elementos LR(0) para una gramática aumentada G'; el algoritmo se muestra en la figura 4.34.

Ejemplo 4.36. En la figura 4.35 se muestra la colección canónica de conjuntos de elementos LR(0) para la gramática (4.19) del ejemplo 4.34. La función ir_a para este conjunto de elementos se muestra en la figura 4.36 como el diagrama de transiciones de un autómata finito determinista D. □

```
procedure elementos (G');
begin
    C := {cerradura ({[S' → ·S]})};
    repeat
        for cada conjunto de elementos I en C y cada símbolo
            gramatical X tal que ir_a (I, X) no esté vacío y no
                esté en C do
                    añadir ir_a (I, X) a C
    until no se puedan añadir más conjuntos de elementos a C
end
```

Fig. 4.34. Construcción de conjuntos de elementos.

Si cada estado de D de la figura 4.36 es un estado final e I_0 es el estado inicial, entonces D reconoce exactamente los prefijos viables de la gramática (4.19). Esto no es ninguna casualidad. Para toda gramática G, la función *ir_a* de la colección canónica de los conjuntos del elementos define un autómata finito determinista que reconoce los prefijos viables de G. De hecho, se puede visualizar un autómata finito

I_0:
$E' \to \cdot E$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$

I_1:
$E' \to E \cdot$
$E \to E \cdot + T$

I_2:
$E \to T \cdot$
$T \to T \cdot * F$

I_3:
$T \to F \cdot$

I_4:
$F \to (\cdot E)$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$

I_5:
$F \to \mathbf{id} \cdot$

I_6:
$E \to E + \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$

I_7:
$T \to T * \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$

I_8:
$F \to (E \cdot)$
$E \to E \cdot + T$

I_9:
$E \to E + T \cdot$
$T \to T \cdot * F$

I_{10}:
$T \to T * F \cdot$

I_{11}:
$F \to (E) \cdot$

Fig. 4.35. Colección del análisis sintáctico LR(0) canónico para la gramática (4.19).

no determinista N cuyos estados son los propios elementos. Hay una transición de $A \rightarrow \alpha \cdot X\beta$ a $A \rightarrow \alpha X \cdot \beta$ etiquetada con X, y hay una transición de $A \rightarrow \alpha \cdot B\beta$ a $B \rightarrow \cdot\gamma$ etiquetada con ϵ. Entonces, *cerradura*(I) para el conjunto de elementos (estados de N) I es exactamente la *cerradura-*ϵ de un conjunto de estados de AFN definida en la sección 3.6. Por tanto, *ir_a*(I, X) da la transición desde I con el símbolo X en el AFD construido a partir de N por la construcción de subconjuntos. Así considerado, el procedimiento *elementos*(G') de la figura 4.34 es simplemente la propia construcción de subconjuntos aplicada al AFN N construido a partir de G', como ya se ha descrito.

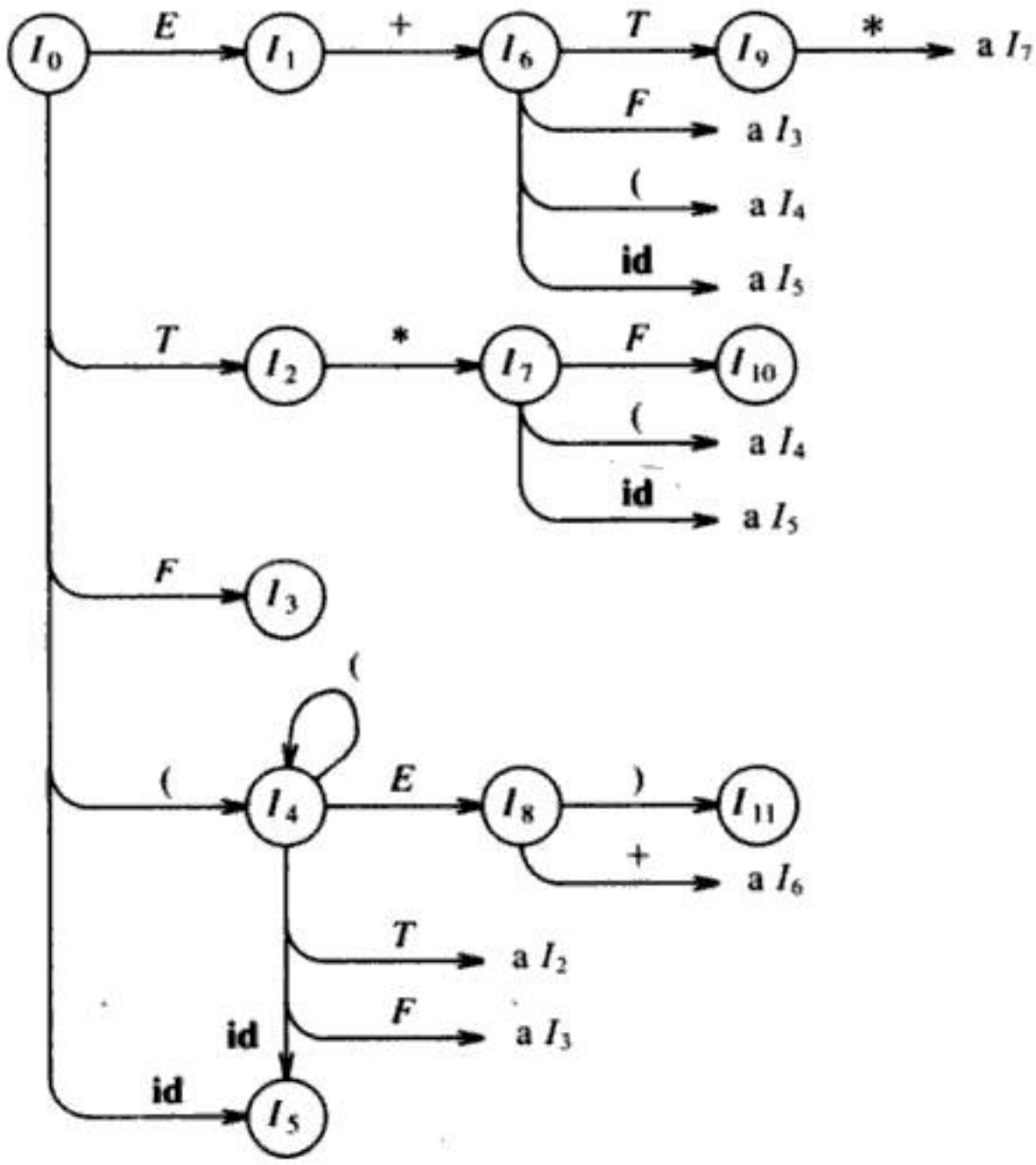


Fig. 4.36. Diagrama de transiciones del AFD D para los prefijos viables.

Elementos válidos. Se dice que el elemento $A \rightarrow \beta_1 \cdot \beta_2$ es *válido* para un prefijo viable $\alpha\beta_1$ si existe una derivación $S' \overset{*}{\underset{rd}{\Rightarrow}} \alpha A w \overset{*}{\underset{rd}{\Rightarrow}} \alpha\beta_1\beta_2 w$. En general, un elemento será válido para muchos prefijos viables. El hecho de que $A \rightarrow \beta_1 \cdot \beta_2$ sea válido para $\alpha\beta_1$ informa sobre si desplazar o reducir cuando se encuentre $\alpha\beta_1$ en la pila de análisis sintáctico. En concreto, si $\beta_2 \neq \epsilon$, entonces indica que aún no se ha desplazado el mango hacia la pila, así que el movimiento debe ser desplazar. Si $\beta_2 = \epsilon$, entonces parece que $A \rightarrow \beta_1$ es el mango, y se debe reducir mediante esta producción. Por

supuesto, dos elementos válidos pueden indicar dos cosas distintas para el mismo prefijo válido. Se pueden resolver algunos de estos conflictos observando el siguiente símbolo de entrada y otros se pueden resolver con los métodos de la siguiente sección, pero no hay que suponer que todos los conflictos de acciones de análisis sintáctico se pueden resolver utilizando el método LR para construir una tabla de análisis sintáctico para una gramática arbitraria.

El conjunto de elementos válidos para cada prefijo viable que pueda aparecer en la pila de un analizador LR es fácil de calcular. De hecho, un teorema básico de la teoría del análisis sintáctico LR es que el conjunto de elementos válidos para un prefijo viable γ es exactamente el conjunto de elementos alcanzados desde el estado inicial a lo largo de un camino etiquetado con γ en el AFD construido a partir de la colección canónica de conjuntos de elementos con transiciones dadas por *ir_a*. En resumen, el conjunto de elementos válidos abarca toda la información útil que se puede extraer de la pila. Aunque aquí no se demostrará, se dará un ejemplo de este teorema.

Ejemplo 4.37. Considérese de nuevo la gramática (4.19), cuyos conjuntos de elementos y función *ir_a* se exhiben en las figuras 4.35 y 4.36. Evidentemente, la cadena $E + T*$ es un prefijo viable de (4.19). El autómata de la figura 4.36 se encontrará en el estado I_7 después de haber leído $E + T*$. El estado I_7 contiene los elementos

$$T \rightarrow T* \cdot F$$
$$F \rightarrow \cdot(E)$$
$$F \rightarrow \cdot \mathbf{id}$$

que son precisamente los elementos válidos para $E + T*$. Para comprobarlo, considérense las tres siguientes derivaciones por la derecha

$$\begin{aligned} E' &\Rightarrow \mathrm{E} \\ &\Rightarrow E + T \\ &\Rightarrow E + T*F \\ &\Rightarrow E + T*(E) \end{aligned} \qquad \begin{aligned} E' &\Rightarrow E \\ &\Rightarrow E + T \\ &\Rightarrow E + T*F \\ &\Rightarrow E + T*\mathbf{id} \end{aligned}$$

La primera derivación muestra la validez de $T \rightarrow T* \cdot F$; la segunda, la validez de $F \rightarrow \cdot(E)$, y la tercera, la validez de $F \rightarrow \cdot \mathbf{id}$ para el prefijo viable $E + T*$. Se puede demostrar que no hay otros elementos válidos para $E + T*$; se deja la prueba al lector interesado. □

Tablas de análisis sintáctico SLR

A continuación se muestra cómo construir las funciones de acción e *ir_a* del análisis sintáctico SLR a partir del autómata finito determinista que reconoce prefijos viables. Este algoritmo no producirá únicamente tablas de acciones definidas para todas las gramáticas, pero funcionará correctamente con muchas gramáticas para lenguajes de programación. Dada una gramática, G, se aumenta G para producir G',

y a partir de G' se construye C, la colección canónica de conjuntos de elementos para G'. Se construye *acción*, la función de acciones del analizador sintáctico, e *ir_a*, la función de transiciones de estados, a partir de C utilizando el siguiente algoritmo. El algoritmo exige que se conozca SIGUIENTE(A) para cada no terminal A de una gramática (véase Sec. 4.4).

Algoritmo 4.8. Construcción de una tabla de análisis sintáctico SLR.

Entrada. Una gramática aumentada G'.

Salida. Las funciones *acción* e *ir_a* de la tabla de análisis sintáctico SLR para G'.

Método.

1. Constrúyase $C = \{I_0, I_1, \ldots, I_n\}$, la colección de conjuntos de elementos LR(0) para G'.

2. El estado i se construye a partir de I_i. Las acciones de análisis sintáctico para el estado i se determinan como sigue:

   a) Si $[A \to \alpha \cdot a\beta]$ está en I_i e $ir\_a(I_i, a) = I_j$, entonces asígnese "desplazar j" a *acción*$[i, a]$. Aquí, a debe ser un terminal.

   b) Si $[A \to \alpha \cdot]$ está en I_i, entonces asígnese "reducir $A \to \alpha$" a *acción*$[i, a]$ para toda a en SIGUIENTE(A); aquí, A puede no ser S'.

   c) Si $[S' \to S \cdot]$ está en I_i, entonces asígnese "aceptar" a *acción*$[i, \$]$.

Si las reglas anteriores generan acciones contradictorias, se dice que la gramática no es SLR(1). El algoritmo no consigue en este caso producir un analizador sintáctico.

3. Las transiciones *ir_a* para el estado i se construyen para todos los no terminales A utilizando la regla: si $ir\_a(I_i, A) = I_j$, entonces $ir\_a[i, A] = j$.

4. Todas las entradas no definidas por las reglas 2 y 3 son consideradas "error".

5. El estado inicial del analizador es el construido a partir del conjunto de elementos que contiene $[S' \to \cdot S]$. □

La tabla de análisis sintáctico formada por las funciones de *acción* e *ir_a* del análisis sintáctico determinada por el algoritmo 4.8 se denomina *tabla SLR(1) para G*. Un analizador sintáctico LR que use la tabla SLR(1) para G se denomina analizador sintáctico SLR(1) para G, y una gramática que tenga una tabla de análisis sintáctico SLR(1) se denomina *SLR*(1). Normalmente, se omite el "(1)" después de "SLR", ya que no se consideran los analizadores sintácticos con más de un símbolo de examen por anticipado.

Ejemplo 4.38. Construcción de la tabla SLR para la gramática (4.19). En la figura 4.35 se mostró la serie canónica de conjuntos de elementos del análisis sintáctico LR(0) para (4.19). Primero considérese el conjunto de elementos I_0:

$$E' \to \cdot E$$
$$E \to \cdot E + T$$
$$E \to \cdot T$$

$T \rightarrow \cdot T*F$
$T \rightarrow \cdot F$
$F \rightarrow \cdot(E)$
$F \rightarrow \cdot \mathbf{id}$

El elemento $F \rightarrow \cdot(E)$ da lugar a la entrada *acción*[0, (] = desplazar 4, y el elemento $F \rightarrow \cdot \mathbf{id}$ a la entrada *acción*[0, **id**] = desplazar 5. Los otros elementos en I_0 no dan lugar a acciones. Ahora considérese I_1:

$E' \rightarrow E\cdot$
$E \rightarrow E\cdot + T$

El primer elemento produce *acción*[1, \$] = aceptar, el segundo produce *acción*[1, +] = desplazar 6. Ahora considérese I_2:

$E \rightarrow T\cdot$
$T \rightarrow T\cdot *F$

Como SIGUIENTE(E) = {\$, +,)}, el primer elemento hace que *acción*[2, \$] = *acción*[2, +] = *acción*[2,)] = reducir $E \rightarrow T$. El segundo elemento hace *acción*[2, *] = desplazar 7. Si se continúa así, se obtienen las tablas de *acción* e *ir_a* del análisis sintáctico de la figura 4.31. En dicha figura, los números de producciones en las acciones de reducción son los mismos que el orden en que aparecieron en la gramática original (4.18). Es decir, $E \rightarrow E + T$ es el número 1, $E \rightarrow T$ es el 2, etcétera. □

Ejemplo 4.39. Toda gramática SLR(1) es no ambigua, pero hay muchas gramáticas no ambiguas que no son SLR(1). Considérese la gramática con producciones

$$\begin{aligned} &S \rightarrow L = R \\ &S \rightarrow R \\ &L \rightarrow * R \\ &L \rightarrow \mathbf{id} \\ &R \rightarrow L \end{aligned} \tag{4.20}$$

Se puede considerar que L y R representan un *valor de lado izquierdo* y un *valor de lado derecho*, respectivamente, y que * es un operador que indica "contenido de"[3]. En la figura 4.37 se muestra la colección canónica de conjuntos de elementos LR(0) para la gramática (4.20).

Considérese el conjunto de elementos I_2. El primer elemento de este conjunto hace que *acción*[2, =] sea "desplazar 6". Como SIGUIENTE(R) contiene =, (para saber por qué, considérese $S \Rightarrow L = R \Rightarrow * R = R$), el segundo elemento hace que *acción*[2, =] sea "reducir $R \rightarrow L$". Por tanto, la entrada *acción*[2, =] tiene múltiples definiciones. Como en *acción*[2, =], existen las dos entradas, desplazar y reducir, el estado 2 tiene un conflicto de desplazamiento/reducción con el símbolo de entrada =.

La gramática (4.20) no es ambigua. Este conflicto de desplazamiento/reducción surge porque el método de construcción de analizadores sintácticos SLR no es lo

[3] Como se vio en la sección 2.8, un valor de lado izquierdo designa una posición, y un valor de lado derecho es un valor que se puede almacenar en una posición.

I_0: $S' \to \cdot S$
$S \to \cdot L = R$
$S \to \cdot R$
$L \to \cdot *R$
$L \to \cdot \mathbf{id}$
$R \to \cdot L$

I_1: $S' \to S\cdot$

I_2: $S \to L\cdot = R$
$R \to L\cdot$

I_3: $S \to R\cdot$

I_4: $L \to *\cdot R$
$R \to \cdot L$
$L \to \cdot *R$
$L \to \cdot \mathbf{id}$

I_5: $L \to \mathbf{id}\cdot$

I_6: $S \to L = \cdot R$
$R \to \cdot L$
$L \to \cdot *R$
$L \to \cdot \mathbf{id}$

I_7: $L \to *R\cdot$

I_8: $R \to L\cdot$

I_9: $S \to L = R\cdot$

Fig. 4.37. Colección del análisis sintáctico LR(0) canónico para la gramática (4.20).

suficientemente poderoso como para recordar el contexto a la izquierda indispensable para decidir qué acción debe realizar el analizador con la entrada = habiendo visto una cadena reducible a L. Los métodos canónico y LALR, que se estudian a continuación, serán los adecuados para una serie mayor de gramáticas, incluida la gramática (4.20). Sin embargo, hay que resaltar que existen gramáticas no ambiguas para las que todo método de construcción de analizadores sintácticos LR producirá una tabla de acciones de análisis sintáctico con conflictos de acciones. Afortunadamente, dichas gramáticas se suelen poder evitar en las aplicaciones de lenguajes de programación. □

Construcción de tablas de análisis sintáctico LR canónico

A continuación se introduce la técnica más general para construir una tabla de análisis sintáctico LR a partir de una gramática. Recuérdese que en el método SLR, el estado i pide una reducción en $A \to \alpha$ si el conjunto de elementos I_i contiene el elemento $[A \to \alpha\cdot]$ y a está en SIGUIENTE(A). Sin embargo, a veces, cuando el estado i aparece en la cima de la pila, el prefijo viable $\beta\alpha$ de la pila es tal que βA no puede ir seguida de una a en una forma de frase derecha. Por tanto, la reducción por $A \to \alpha$ sería no válida con la entrada a.

Ejemplo 4.40. Considérese de nuevo el ejemplo 4.39, donde en el estado 2 estaba el elemento $R \to L\cdot$, que podía corresponder a la regla $A \to \alpha$ anterior, y a podría ser el signo =, que está en SIGUIENTE(R). Por tanto, el analizador sintáctico SLR pide una reducción por $R \to L$ en el estado 2 con = como símbolo de entrada siguiente (también se pide la acción de desplazar a causa de los elementos $S \to L\cdot = R$ en el estado 2). Sin embargo, no hay ninguna forma de frase derecha de la gramática que

comience con $R = \ldots$. Por tanto, el estado 2, que corresponde al prefijo viable L solamente, en realidad no debería pedir una reducción de esa L a R. □

Es posible llevar más información en el estado que permita prohibir algunas de estas reducciones no válidas por $A \rightarrow \alpha$. Dividiendo estados siempre que sea necesario, se puede lograr que cada estado de un analizador sintáctico LR indique exactamente qué símbolos de entrada pueden seguir a un mango α para el cual hay una posible reducción a A.

Se incorpora la información adicional al estado redefiniendo elementos para incluir un símbolo terminal como segundo componente. La forma general de un elemento se convierte en $[A \rightarrow \alpha \cdot \beta, a]$, donde $A \rightarrow \alpha\beta$ es una producción y a es un terminal o el marcador del extremo derecho \$. Dicho objeto se denomina *elemento del análisis sintáctico LR(1)*. El 1 se refiere a la longitud del segundo componente, llamado *símbolo de anticipación* del elemento[4]. El símbolo de anticipación no tiene efecto en un elemento de la forma $[A \rightarrow \alpha \cdot \beta, a]$, donde β no es ϵ, pero un elemento de la forma $[A \rightarrow \alpha \cdot, a]$ pide una reducción por $A \rightarrow \alpha$ sólo si el siguiente símbolo de entrada es a. Por tanto, se debe reducir por $A \rightarrow \alpha$ sólo con aquellos símbolos de entrada a para los que $[A \rightarrow \alpha \cdot, a]$ es un elemento LR(1) en el estado de la cima de la pila. El conjunto de dichas a siempre será un subconjunto de SIGUIENTE(A), pero podría ser un subconjunto propio, como en el ejemplo 4.40.

Formalmente, se dice que un elemento LR(1) $[A \rightarrow \alpha \cdot \beta, a]$ es *válido* para un prefijo viable γ si existe una derivación $S \underset{rmd}{\overset{*}{\Rightarrow}} \delta A w \underset{rmd}{\Rightarrow} \delta\alpha\beta w$, donde

1. $\gamma = \delta\alpha$, y
2. a es el primer símbolo de w, o w es ϵ y a es \$.

Ejemplo 4.41. Considérese la gramática

$$S \rightarrow BB$$
$$B \rightarrow aB \mid b$$

Hay una derivación por la derecha $S \underset{rmd}{\overset{*}{\Rightarrow}} aaBab \underset{rmd}{\Rightarrow} aaaBab$. El elemento $[B \rightarrow a \cdot B, a]$ es válido para un prefijo viable $\gamma = aaa$ haciendo $\delta = aa$, $A = B$, $w = ab$, $\alpha = a$ y $\beta = B$ en la definición anterior.

También hay una derivación por la derecha $S \underset{rmd}{\overset{*}{\Rightarrow}} BaB \underset{rmd}{\Rightarrow} BaaB$. De aquí se deduce que el elemento $[B \rightarrow a \cdot B, \$]$ es válido para el prefijo viable Baa. □

El método para construir la colección de conjuntos de elementos LR(1) válidos es fundamentalmente el mismo que la forma en que se construyó la colección canónica de conjuntos de elementos del análisis sintáctico LR(0). Sólo hay que modificar los dos procedimientos *cerradura* e *ir_a*.

Para apreciar la nueva definición de la operación *cerradura*, considérese un elemento de la forma $[A \rightarrow \alpha \cdot B\beta, a]$ en el conjunto de elementos válidos para un prefijo viable γ. Entonces hay una derivación por la derecha $S \underset{rmd}{\overset{*}{\Rightarrow}} \delta Aax \underset{rmd}{\Rightarrow} \delta aB\beta ax$, donde $\gamma = \delta\alpha$. Supóngase que de βax se deriva la cadena de terminales by. Entonces, para cada producción de la forma $B \rightarrow \eta$ para algún η, se tiene entonces la de-

[4] Son posibles, desde luego, los símbolos de anticipación con cadenas de longitud mayor que uno, pero no se tratarán aquí.

rivación $S \overset{*}{\underset{rd}{\Rightarrow}} \gamma Bby \underset{rd}{\Rightarrow} \gamma\eta by$. Por tanto, $[B \to \cdot\eta, b]$ es válido para γ. Obsérvese que b puede ser el primer terminal derivado de β, o es posible que β derive ϵ en la derivación $\beta ax \overset{*}{\Rightarrow} by$, y por tanto b puede ser a. Para resumir ambas posibilidades, se establece que b puede ser cualquier terminal en PRIMERO(βax), donde PRIMERO es la función de la sección 4.4. Obsérvese que x no puede contener el primer terminal de by, de modo que PRIMERO(βax) = PRIMERO(βa). A continuación se da la construcción de los conjuntos de elementos LR(1).

Algoritmo 4.9. Construcción de los conjuntos de elementos LR(1).

Entrada. Una gramática aumentada G'.

Salida. Los conjuntos de elementos LR(1) que son el conjunto de elementos válido para uno o más prefijos viables de G'.

Método. Los procedimientos *cerradura* e *ir_a* y la rutina principal *elementos* para construir los conjuntos de elementos se muestran en la figura 4.38. □

```
function cerradura (I);
begin
    repeat
        for cada elemento [A → α·Bβ, a] en I,
            cada producción B → γ en G'
            y cada terminal b en PRIMERO(βa)
            tal que [B → ·γ, b] no esté en I do
                añadir [B → ·γ, b] a I;
    until no se puedan añadir más elementos a I;
    return I
end;

function ir_a (I, X);
begin
    sea J el conjunto de elementos [A → αX·β, a] tal que
        [A → α·Xβ, a] esté en I;
    return cerradura (J)
end;

procedure elementos (G');
begin
    C := {cerradura ({[S' → ·S, $]})};
    repeat
        for cada conjunto de elementos I en C y cada símbolo
            gramatical X tal que ir_a (I, X) no esté vacío y no esté en C do
                añadir ir_a (I, X) a C
    until no se puedan añadir más conjuntos de elementos a C
end
```

Fig. 4.38. Construcción de conjuntos de elementos LR(1) para la gramática G'.

Ejemplo 4.42. Considérese la siguiente gramática aumentada.

$$\begin{aligned} S' &\to S \\ S &\to CC \\ C &\to cC \mid d \end{aligned} \tag{4.21}$$

Se comienza por calcular la cerradura de $\{[S' \to \cdot S, \$]\}$. Para la cerradura, se empareja el elemento $[S' \to \cdot S, \$]$ con el elemento $[A \to \alpha \cdot B\beta, a]$ en el procedimiento *cerradura*. Es decir, $A = S'$, $\alpha = \epsilon$, $B = S$, $\beta = \epsilon$, y $a = \$$. La función *cerradura* indica que hay que añadir $[B \to \cdot\gamma, b]$ para cada producción $B \to \gamma$ y terminal b en PRIMERO(βa). En función de esta gramática, $B \to \gamma$ debe ser $S \to CC$, y como β es ϵ y a es \$, b sólo puede ser \$. Por tanto, se añade $[S \to \cdot CC, \$]$.

Se continúa el cálculo de la cerradura añadiendo todos los elementos $[C \to \cdot\gamma, b]$ para b en PRIMERO($C\,\$$). Es decir, emparejando $[S \to \cdot CC, \$]$ con $[A \to \alpha \cdot B\beta, a]$ se tiene $A = S$, $\alpha = \epsilon$, $B = C$, $\beta = C$, y $a = \$$. Como de C no se deriva la cadena vacía, PRIMERO($C\$$) = PRIMERO(C). Puesto que PRIMERO(C) contiene los terminales c y d, se añaden los elementos $[C \to \cdot cC, c]$, $[C \to \cdot cC, d]$, $[C \to \cdot d, c]$ y $[C \to \cdot d, d]$. Ninguno de los nuevos elementos tiene un no terminal inmediatamente a la derecha del punto, de manera que se ha completado el primer conjunto de elementos LR(1). El conjunto inicial de elementos es:

$$\begin{aligned} I_0: \quad & S' \to \cdot S, \$ \\ & S \to \cdot CC, \$ \\ & C \to \cdot cC, c/d \\ & C \to \cdot d, c/d \end{aligned}$$

Para facilitar la notación se han omitido los corchetes, y la notación $[C \to \cdot cC, c/d]$ se utiliza como abreviatura de los dos elementos $[C \to \cdot cC, c]$ y $[C \to \cdot cC, d]$.

Ahora se calcula $ir\_a(I_0, X)$ para los distintos valores de X. Para $X = S$ se debe cerrar el elemento $[S' \to S\cdot, \$]$. No se puede hacer ninguna otra cerradura porque el punto está en el lado derecho. Entonces, se tiene el siguiente conjunto de elementos:

$$I_1: \quad S' \to S\cdot, \$$$

Para $X = C$ se cierra $[S \to C\cdot C, \$]$. Se añaden las producciones de C con segundo componente \$ y ya no se puede añadir ninguna más, obteniéndose:

$$\begin{aligned} I_2: \quad & S \to C\cdot C, \$ \\ & C \to \cdot cC, \$ \\ & C \to \cdot d, \$ \end{aligned}$$

A continuación, sea $X = c$. Se debe cerrar $\{[C \to c\cdot C, c/d]\}$. Se añaden las producciones de C con segundo componente c/d, resultando:

$$\begin{aligned} I_3: \quad & C \to c\cdot C, c/d \\ & C \to \cdot cC, c/d \\ & C \to \cdot d, c/d \end{aligned}$$

Por último, se hace $X = d$ y se termina con el conjunto de elementos:

$$I_4: \quad C \to d\cdot, c/d$$

Se termina considerando *ir_a* en I_0. No se obtienen nuevos conjuntos de I_1 pero I_2 tiene elementos de *ir_a* con C, c y d. En C se obtiene:

I_5: $S \rightarrow CC\cdot, \$$

no se necesita ninguna cerradura. En c se toma la cerradura de $\{[C \rightarrow c\cdot C, \$]$ para obtener:

I_6: $C \rightarrow c\cdot C, \$$
$C \rightarrow \cdot cC, \$$
$C \rightarrow \cdot d, \$$

Obsérvese que I_6 difiere de I_3 sólo en los segundos componentes. Se comprobará que es habitual que varios conjuntos de elementos LR(1) de una gramática tengan los mismos primeros componentes y difieran en sus segundos componentes. Cuando se construye la colección de conjuntos de elementos LR(0) para la misma gramática, cada conjunto de elementos LR(0) coincidirá con el conjunto de primeros componentes de uno o más conjuntos de elementos LR(1). Se tratará este fenómeno más a fondo al estudiar el análisis sintáctico LALR.

Continuando con la función *ir_a* para I_2, *ir_a*(I_2, d) resulta:

I_7: $C \rightarrow d\cdot, \$$

Volviendo a I_3, los elementos de *ir_a* de I_3 con c y d son I_3 e I_4, respectivamente, e *ir_a*(I_3, C) es:

I_8: $C \rightarrow cC\cdot, c/d$

I_4 e I_5 no tienen elementos *ir_a*. Los elementos *ir_a* de I_6 con c y d son I_6 e I_7, respectivamente, e *ir_a*(I_6, C) es:

I_9: $C \rightarrow cC\cdot, \$$

Los restantes conjuntos de elementos no producen elementos de *ir_a*, por lo que se acaba. En la figura 4.39 se muestran los diez conjuntos de elementos con sus elementos *ir_a*. □

A continuación se dan las reglas por las que se construyen las funciones de acción e ir a del análisis sintáctico LR(1) a partir de los conjuntos de elementos LR(1). Las funciones de acción e ir a se representan mediante una tabla, como se vio anteriormente. La única diferencia radica en los valores de las entradas.

Algoritmo 4.10. Construcción de la tabla de análisis sintáctico LR canónico.

Entrada. Una gramática aumentada G'.

Salida. Las funciones *acción* e *ir_a* de la tabla de análisis sintáctico LR canónico para G'.

Método.

1. Constrúyase $C = \{I_0, I_1 \ldots, I_n\}$, la colección de conjuntos de elementos LR(1) para G'.

2. El estado i del analizador sintáctico se construye a partir de I_i. Las acciones de análisis sintáctico para el estado i se determinan de la siguiente manera:

   a) Si $[A \to \alpha \cdot a\beta, b]$ está en I_i e $ir\_a(I_i, a) = I_j$, entonces atribúyase *acción*$[i, a]$ a "desplazar j". En este caso es necesario que a sea un terminal.

   b) Si $[A \to \alpha \cdot, a]$ está en I_i, $A \neq S'$, entonces atribúyase *acción*$[i, a]$ a "reducir $A \to \alpha$".

   c) Si $[S' \to S \cdot, \$]$ está en I_i, entonces atribúyase *acción*$[i, \$]$ a "aceptar".

Si se produce un conflicto por las reglas anteriores, se dice que la gramática no es LR(1) y que el algoritmo falla.

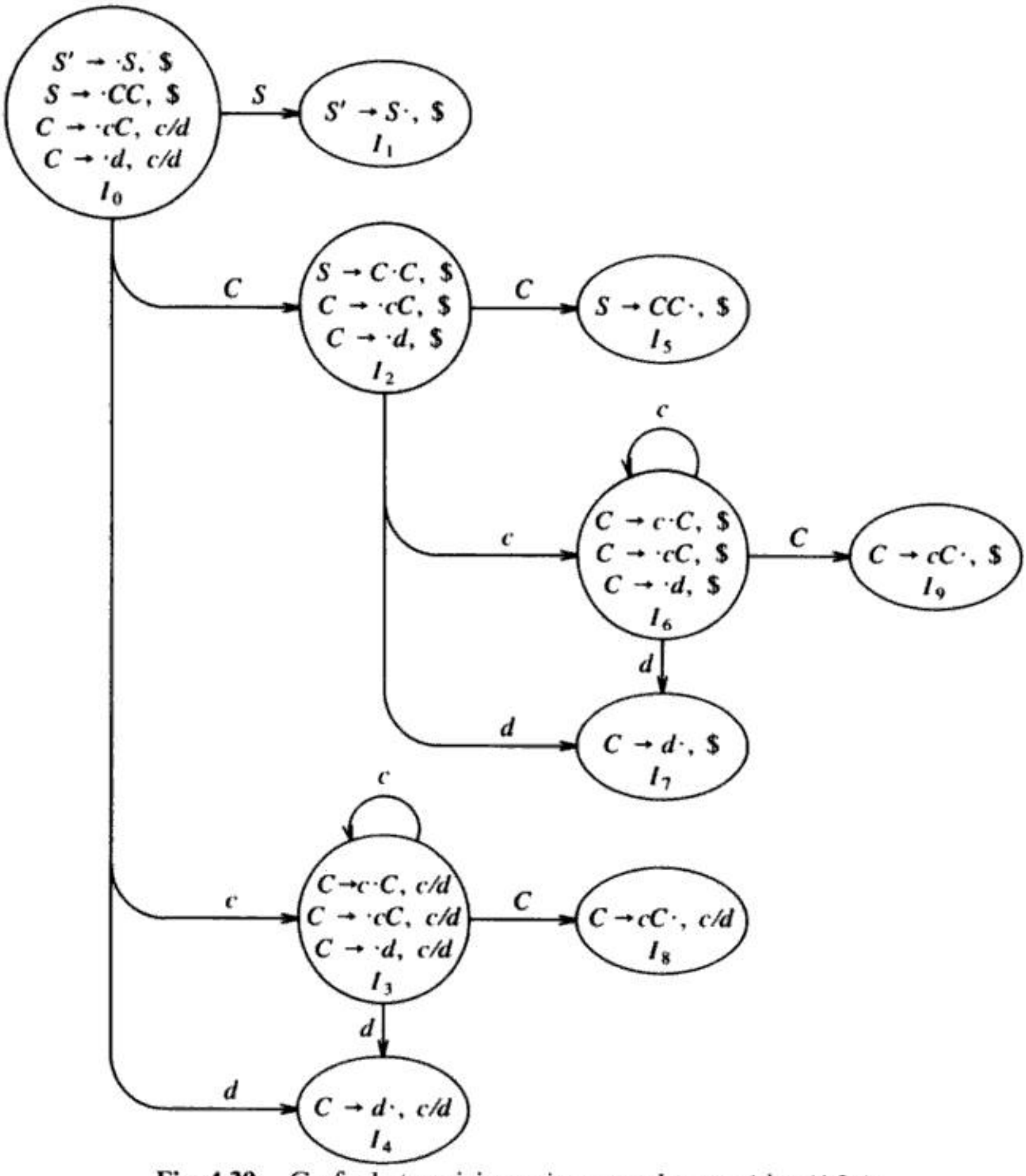


Fig. 4.39. Grafo de transiciones *ir_a* para la gramática (4.21).

3. Las transiciones ir a para el estado i se determinan como sigue: si $ir\_a(I_i, A) = I_j$, entonces $ir\_a[i, A] = j$.
4. Todas las entradas no definidas por las reglas 2 y 3 se consideran "error".
5. El estado inicial del analizador sintáctico es el construido a partir del conjunto que contiene el elemento $[S' \rightarrow \cdot S, \$]$. □

La tabla formada a partir de las funciones acción e ir a de análisis sintáctico producida por el algoritmo 4.10 se denomina tabla de análisis sintáctico LR(1) *canónico*. Un analizador LR que utilice esta tabla se denomina analizador sintáctico LR(1) canónico. Si la función de acción del análisis sintáctico no tiene entradas de múltiples definiciones, entonces la gramática dada se denomina *gramática LR(1)*. Como antes, si se sobreentiende, se omite el "(1)".

Ejemplo 4.43. En la figura 4.40 se muestra la tabla de análisis sintáctico canónico para la gramática (4.21). Las producciones 1, 2 y 3 son $S \rightarrow CC$, $C \rightarrow cC$ y $C \rightarrow d$. □

Toda gramática SLR(1) es una gramática LR(1), pero para una gramática SLR(1) el analizador sintáctico LR canónico puede tener más estados que el analizador SLR para la misma gramática. La gramática de los ejemplos anteriores es SLR y tiene un analizador sintáctico SLR con siete estados, comparados con los diez de la figura 4.40.

ESTADO	*acción*			*ir_a*	
	c	*d*	$	*S*	*C*
0	d3	d4		1	2
1			acep		
2	d6	d7			5
3	d3	d4			8
4	r3	r3			
5			r1		
6	d6	d7			9
7			r3		
8	r2	r2			
9			r2		

Fig. 4.40. Tabla de análisis sintáctico canónico para la gramática (4.21).

Construcción de las tablas de análisis sintáctico LALR

A continuación se analiza el último método de construcción de analizadores sintácticos, la técnica LALR (del inglés, *lookahead*-LR, *análisis sintáctico LR con símbolo de anticipación*). A menudo se utiliza este método en la práctica porque las tablas con él obtenidas son bastante más pequeñas que las tablas del análisis LR canónico, y las construcciones sintácticas más frecuentes de los lenguajes de programación

pueden expresarse convenientemente con una gramática LALR. Algo parecido ocurre con las gramáticas SLR, pero existen unas cuantas construcciones que no se pueden manejar convenientemente con las técnicas SLR (véase Ejemplo 4.39).

A efectos de comparar el tamaño de un analizador sintáctico, las tablas SLR y LALR para una gramática siempre tienen el mismo número de estados, normalmente varios cientos de estados para un lenguaje como Pascal. La tabla del análisis LR canónico tendría generalmente varios miles de estados para un lenguaje del mismo tamaño. Por tanto, es mucho más fácil y económico construir tablas SLR y LALR que tablas del análisis LR canónico.

Como introducción, considérese de nuevo la gramática (4.21), cuyos conjuntos de elementos LR(1) se mostraron en la figura 4.39. Tómese un par de estados que parezcan similares, como I_4 e I_7. Cada uno de estos estados tiene únicamente elementos con el primer componente $C \rightarrow d\cdot$. En I_4, los símbolos de anticipación son c o d; en I_7, \$ es el único símbolo de anticipación.

Para ver la diferencia entre las funciones de I_4 e I_7 en el analizador sintáctico, obsérvese que la gramática (4.21) genera el conjunto regular c^*dc^*d. Al leer una entrada $cc \ldots cdcc \ldots cd$, el analizador introduce el primer grupo de símbolos c y su d siguiente en la pila, entrando en el estado 4 después de leer la d. El analizador llama entonces una reducción por $C \rightarrow d$, suponiendo que el siguiente símbolo de entrada sea c o d. El requisito de que vaya seguida de c o d tiene sentido, puesto que éstos son los símbolos que podrían comenzar cadenas en c^*d. Si \$ sigue a la primera d, se produce una entrada como ccd, que no está en el lenguaje, y el estado 4 declara convenientemente un error si \$ es el siguiente símbolo de entrada.

El analizador entra en el estado 7 después de leer la segunda d. Después, el analizador debe ver \$ en la entrada, o de lo contrario es que empezó con una cadena distinta del c^*dc^*d. Por tanto, tiene sentido que el estado 7 reduzca por $C \rightarrow d$ con la entrada \$ y declare error con las entradas c o d.

A continuación sustitúyanse I_4 e I_7 por I_{47}, la unión de I_4 e I_7, que consta del conjunto de tres elementos representados por $[C \rightarrow d\cdot, c/d/\$]$. Las transiciones ir a con d a I_4 o I_7 desde I_0, I_2, I_3 e I_6 entran ahora en I_{47}. La acción del estado 47 es reducir con cualquier entrada. El analizador sintáctico revisado se comporta fundamentalmente como el original, aunque es posible que reduzca d a C en casos en que el original declararía error, por ejemplo, con entradas como ccd o $cdcdc$. El error se descubrirá de todas formas; de hecho, se descubrirá antes de que se desplacen más símbolos de entrada.

Por lo general, se pueden buscar conjuntos de elementos LR(1) que tengan el mismo *corazón*, es decir, el conjunto de primeros elementos, y se pueden fusionar en un conjunto de elementos estos conjuntos con corazones comunes. Por ejemplo, en la figura 4.39, I_4 e I_7 forman uno de dichos pares, con corazón $\{C \rightarrow d\cdot\}$. De manera similar, I_3 e I_6 forman otro par, con corazón $\{C \rightarrow c\cdot C, C \rightarrow \cdot cC, C \rightarrow \cdot d\}$. Existe otro par más, I_8 e I_9, con corazón $\{C \rightarrow cC\cdot\}$. Obsérvese que, en general, un corazón es un conjunto de elementos LR(0) para la gramática en cuestión, y que una gramática LR(1) puede producir más de dos conjuntos de elementos con el mismo corazón.

Como el corazón de $ir\_a(I, X)$ depende sólo del corazón de I, las transiciones ir a de los estados fusionados también se pueden fusionar. Por tanto, revisar la función

ir a conforme se fusionan conjuntos de elementos no supone un problema. Se modifican las funciones de acción para reflejar las acciones distintas de error de todos los conjuntos de elementos de la fusión.

Supóngase que se tiene una gramática LR(1), es decir, una cuyos conjuntos de elementos LR(1) no producen conflictos en las acciones del análisis. Si se sustituyen todos los estados que tengan el mismo corazón por su unión, es posible que la unión obtenida tenga un conflicto, pero no es probable por la siguiente razón: supóngase que en la unión hay un conflicto con el símbolo de anticipación a porque hay un elemento $[A \rightarrow \alpha\cdot, a]$ que pide una reducción por $A \rightarrow \alpha$, y hay otro elemento $[B \rightarrow \beta\cdot a\gamma, b]$ que pide un desplazamiento. Entonces, algún conjunto de elementos a partir del cual se formó la unión tiene el elemento $[A \rightarrow \alpha\cdot, a]$ y, como los corazones de todos estos estados son el mismo, debe tener un elemento $[B \rightarrow \beta\cdot a\gamma, c]$ para alguna c. Pero entonces este estado tiene el mismo conflicto de desplazamiento/reducción con a, y la gramática no era LR(1) como se dio por supuesto. Por tanto, la fusión de estados con corazones comunes nunca puede producir un conflicto de desplazamiento/reducción que no estuviera ya presente en uno de los estados originales, porque las acciones de desplazar sólo dependen del corazón, no del símbolo de anticipación.

Sin embargo, es posible que una fusión produzca un conflicto de reducción/reducción, como se muestra en el siguiente ejemplo.

Ejemplo 4.44. Considérese la gramática

$$
\begin{aligned}
&S' \rightarrow S\\
&S \rightarrow aAd \mid bBd \mid aBe \mid bAe\\
&A \rightarrow c\\
&B \rightarrow c
\end{aligned}
$$

que genera las cuatro cadenas acd, ace, bcd y bce. El lector puede comprobar que la gramática es LR(1) construyendo los conjuntos de elementos. Al hacerlo, se ve que el conjunto de elementos $\{[A \rightarrow c\cdot, d], [B \rightarrow c\cdot, e]\}$ es válido para el prefijo viable ac y que a $\{[A \rightarrow c\cdot, e], [B \rightarrow c\cdot, e]\}$ es válido para bc. Ninguno de estos conjuntos genera un conflicto, y sus corazones son los mismos. Sin embargo, su unión, que es

$$
\begin{aligned}
&A \rightarrow c\cdot, d/e\\
&B \rightarrow c\cdot, d/e
\end{aligned}
$$

genera un conflicto de reducción/reducción, puesto que con las entradas d y e se piden reducciones por $A \rightarrow c$ y por $B \rightarrow c$. □

Ahora se está en condiciones de dar el primero de dos algoritmos para la construcción de tablas LALR. La idea general es construir los conjuntos de elementos LR(1) y, si no surgen conflictos, fusionar los conjuntos con corazones comunes. Después, se construye la tabla de análisis sintáctico a partir de la serie de conjuntos de elementos fusionados. El método que se va a describir sirve en principio como definición de gramáticas LALR(1). Construir la serie completa de conjuntos de elementos LR(1) exige demasiado espacio y tiempo como para que resulte útil en la práctica.

Algoritmo 4.11. Una construcción fácil, pero que ocupa mucho espacio, de una tabla LALR.

Entrada. Una gramática aumentada G'.

Salida. Las funciones *acción* e *ir_a* de la tabla de análisis sintáctico LALR para G'.

Método.

1. Constrúyase $C = \{I_0, I_1, \ldots, I_n\}$, la colección de conjuntos de elementos LR(1).
2. Para cada corazón presente entre el conjunto de elementos LR(1), encuéntrense todos los conjuntos que tengan dicho corazón, y sustitúyanse estos conjuntos por su unión.
3. Sea $C' = \{J_0, J_1, \ldots, J_m\}$ los conjuntos de elementos LR(1) obtenidos. Se construyen las acciones de análisis sintáctico para el estado i a partir de J_i del mismo modo que en el algoritmo 4.10. Si hay un conflicto en las acciones del análisis sintáctico, el algoritmo no consigue producir un analizador sintáctico, y se dice que la gramática no es LALR(1).
4. La tabla *ir_a* se construye de la siguiente manera. Si J es la unión de uno o más conjuntos de elementos LR(1), es decir, $J = I_1 \cup I_2 \cup I_k$, entonces los corazones de $ir\_a(I_1, X)$, $ir\_a(I_2, X)$, $\ldots$, $ir\_a(I_k, X)$ son el mismo, puesto que $I_1, I_2, \ldots, I_k$ tienen todos el mismo corazón. Sea K la unión de todos los conjuntos de elementos que tienen el mismo corazón que $ir\_a(I_1, X)$. Entonces $ir\_a(J, X) = K$. □

La tabla producida por el algoritmo 4.11 se denomina *tabla de análisis sintáctico LALR* para G. Si no hay conflictos en las acciones del análisis sintáctico, entonces se dice que la gramática dada es una *gramática LALR*(1). La colección de conjuntos de elementos construida en el paso 3 se denomina *colección LALR*(1).

Ejemplo 4.45. Considérese de nuevo la gramática 4.21 cuyo grafo de *ir_a* se mostró en la figura 4.39. Como ya se ha mencionado, hay tres pares de conjuntos de elementos que pueden fusionarse. I_3 e I_6 se sustituyen por su unión:

I_{36}: $C \to c \cdot C, c/d/\$$
$C \to \cdot cC, c/d/\$$
$C \to \cdot d, c/d/\$$

I_4 e I_7 se sustituyen por su unión:

I_{47}: $C \to d \cdot, c/d/\$$

e I_8 e I_9 se sustituyen por su unión:

I_{89}: $C \to cC \cdot, c/d/\$$

En la figura 4.41 se muestran las funciones de acción e ir a del análisis LALR para los conjuntos de elementos condensados.

Para comprobar cómo se calculan las transiciones ir a, considérese $ir\_a(I_{36}, C)$. En el conjunto original de elementos LR(1), $ir\_a(I_3, C) = I_8$, y ahora I_8 es parte de

ESTADO	*acción*			*ir_a*	
	c	*d*	$	*S*	*C*
0	d36	d47		1	2
1			acep		
2	d36	d47			5
36	d36	d47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Fig. 4.41. Tabla de análisis sintáctico LALR para la gramática (4.21).

I_{89}, así que $ir\_a(I_{36}, C)$ es igual a I_{89}. Se podría haber llegado a la misma conclusión si se hubiera considerado I_6, la otra parte de I_{36}. Es decir, $ir\_a(I_6, C) = I_9$, y ahora I_9 es parte de I_{89}. Como otro ejemplo, considérese $ir\_a(I_2, c)$, una entrada que se ejecuta después de la acción de desplazamiento de I_2 con la entrada c. En los conjuntos de elementos LR(1) originales, $ir\_a(I_2, c) = I_6$. Como I_6 es ahora parte de I_{36}, $ir\_a(I_2, c)$ se convierte en I_{36}. Así, la entrada en la figura 4.41 para el estado 2 y la entrada c se iguala a d36, lo cual supone desplazar e introducir el estado 36 en la pila. □

Cuando se encuentran con una cadena del lenguaje c^*dc^*d, tanto el analizador sintáctico LR de la figura 4.40 como el analizador sintáctico LALR de la figura 4.41 realizan exactamente la misma secuencia de desplazamientos y reducciones, aunque pueden variar los nombres de los estados de la pila; es decir, si el analizador sintáctico LR pone I_3 o I_6 en la pila, el analizador sintáctico LALR pondrá I_{36} en la pila. Esta relación se cumple en general para una gramática LALR. Los analizadores sintácticos LR y LALR se imitarán uno a otro con entradas correctas.

Sin embargo, cuando se encuentran con una entrada errónea, es posible que el analizador sintáctico LALR continúe haciendo algunas reducciones después de que el analizador sintáctico LR haya declarado un error, aunque el analizador LALR nunca desplazará ningún otro símbolo después de que el analizador LR haya declarado un error. Por ejemplo, con la entrada *ccd* seguida de $, el analizador LR de la figura 4.34 pondrá

0 *c* 3 *c* 3 *d* 4.

en la pila, y en el estado 4 descubrirá un error, porque $ es el siguiente símbolo de entrada y el estado 4 tiene acción de error con $. Por el contrario, el analizador LALR de la figura 4.41 hará los movimientos correspondientes, poniendo

0 *c* 36 *c* 36 *d* 47

en la pila. Pero el estado 47 con entrada $ tiene la acción de reducir $C \rightarrow d$. Por tanto, el analizador LALR cambiará su pila a

0 *c* 36 *c* 36 *C* 89

La acción del estado 89 con entrada \$ es ahora reducir $C \to cC$, y la pila es

0 c 36 C 89

con lo cual se pide una reducción similar, obteniéndose la pila

0 C 2

Por último, el estado 2 tiene acción de error con entrada \$, y se descubre ahora. □

Construcción eficiente de tablas de análisis sintáctico LALR

Se pueden hacer varias modificaciones al algoritmo 4.11 para evitar construir la colección completa de conjuntos de elementos LR(1) en el proceso de crear una tabla de análisis sintáctico LALR(1). La primera observación es que se puede representar un conjunto con elementos I mediante su núcleo, es decir, mediante aquellos elementos que son el elemento inicial $[S' \to \cdot S, \$]$ o que tengan el punto en algún lugar que no sea al principio del lado derecho.

Segundo, se pueden calcular las acciones de análisis sintáctico generadas por I a partir únicamente del núcleo. Cualquier elemento que pida una reducción por $A \to \alpha$ estará en el núcleo, a menos que $\alpha = \epsilon$. Se pide la reducción por $A \to \epsilon$ si, y sólo si, existe un elemento del núcleo $[B \to \gamma C\delta, b]$ tal que $C \overset{*}{\underset{md}{\Rightarrow}} A\eta$ para alguna η, y a está en PRIMERO($\eta\delta b$). Se puede precalcular el conjunto de no terminales A tales que $C \overset{*}{\underset{md}{\Rightarrow}} A\eta$ para cada no terminal C.

Se pueden determinar las acciones de desplazamiento generadas por I a partir del núcleo de I de la siguiente manera. Se hace un desplazamiento con la entrada a si hay un elemento del núcleo $[B \to \gamma \cdot C\delta, b]$, donde $C \overset{*}{\underset{md}{\Rightarrow}} ax$ en una derivación en que el último paso no utiliza una producción ϵ. También se puede precalcular el conjunto de dichas a para cada C.

Así es cómo se pueden calcular las transiciones ir a para I a partir del núcleo. Si $[B \to \gamma \cdot X\delta, b]$ está en el núcleo de I, entonces $[B \to \gamma X \cdot \delta, b]$ está en el núcleo de $ir\_a(I, X)$. El elemento $[A \to X \cdot \beta, a]$ está también en el núcleo de $ir\_a(I, X)$ si hay un elemento $[B \to \gamma \cdot C\delta, b]$ en el núcleo de I, y $C \overset{*}{\underset{md}{\Rightarrow}} A\eta$ para una η. Si para cada par de no terminales C y A se precalcula si $C \overset{*}{\underset{md}{\Rightarrow}} A\eta$ para una η, entonces calcular los conjuntos de elementos a partir de los núcleos sólo es un poco menos eficiente que hacerlo con conjuntos cerrados de elementos.

Para calcular los conjuntos de elementos LALR(1) para una gramática aumentada G', se comienza con el núcleo $S' \to \cdot S$ del conjunto inicial de elementos I_0. Después se calculan los núcleos de las transiciones ir a a partir de I_0, como ya se ha mencionado. Se siguen calculando las transiciones ir a para cada nuevo núcleo generado hasta conseguir los núcleos de la colección completa de conjuntos de elementos LR(0).

Ejemplo 4.46. Considérese de nuevo la gramática aumentada

$$
\begin{aligned}
S' &\to S \\
S &\to L = R \mid R \\
L &\to *R \mid \mathbf{id} \\
R &\to L
\end{aligned}
$$

Los núcleos de los conjuntos de elementos LR(0) para esta gramática se muestran en la figura 4.42. □

I_0: $S' \rightarrow \cdot S$

I_1: $S' \rightarrow S\cdot$

I_2: $S \rightarrow L\cdot = R$
$R \rightarrow L\cdot$

I_3: $S \rightarrow R\cdot$

I_4: $L \rightarrow *\cdot R$

I_5: $L \rightarrow \mathbf{id}\cdot$

I_6: $S \rightarrow L = \cdot R$

I_7: $L \rightarrow *R\cdot$

I_8: $R \rightarrow L\cdot$

I_9: $S \rightarrow L = R\cdot$

Fig. 4.42. Núcleos de los conjuntos de elementos LR(0) para la gramática (4.20).

Ahora se amplían los núcleos, asociando a cada elemento LR(0) los símbolos de anticipación apropiados (los segundos componentes). Para ver cómo se propagan los símbolos de anticipación desde un conjunto de elementos I hasta $ir\_a(I, X)$, considérese un elemento LR(0) $B \rightarrow \gamma\cdot C\delta$ en el núcleo de I. Supóngase que $C \overset{*}{\underset{rd}{\Rightarrow}} A\eta$ para una η (tal vez $C = A$ y $\eta = \epsilon$), y $A \rightarrow X\beta$ es una producción. Entonces el elemento LR(0) $A \rightarrow X\cdot\beta$ está en $ir\_a(I, X)$.

Ahora supóngase que se están calculando elementos que no son LR(0), sino LR(1), y que $[B \rightarrow \gamma\cdot C\delta, b]$ está en el conjunto I. Entonces, ¿para qué valores de a estará $[A \rightarrow X\cdot\beta, a]$ en $ir\_a(I, X)$? Evidentemente, si una a está en PRIMERO($\eta\delta$), entonces la derivación $C \overset{*}{\underset{rd}{\Rightarrow}} A\eta$ indica que $[A \rightarrow X\cdot\beta, a]$ debe estar en $ir\_a(I, X)$. En este caso, el valor de b es irrelevante y se dice que a, como símbolo de anticipación para $A \rightarrow X\cdot\beta$, se genera *espontáneamente*. Por definición, \$ se genera espontáneamente como símbolo de anticipación para el elemento $S' \rightarrow \cdot S$ en el conjunto de elementos inicial.

Pero existe otra fuente de símbolos de anticipación para el elemento $A \rightarrow X\cdot\beta$. Si $\eta\delta \overset{*}{\Rightarrow} \epsilon$, entonces $[A \rightarrow X\cdot\beta, b]$ estará también en $ir\_a(I, X)$. En este caso, se dice que los símbolos de anticipación se *propagan* desde $B \rightarrow \gamma\cdot C\delta$ hasta $A \rightarrow X\cdot\beta$. En el siguiente algoritmo está contenido un método sencillo para determinar cuándo un elemento LR(1) en I genera un símbolo de anticipación en $ir\_a(I, X)$ espontáneamente, y cuándo se propagan los símbolos de anticipación.

Algoritmo 4.12. Determinación de los símbolos de anticipación.

Entrada. El núcleo K de un conjunto de elementos LR(0) y un símbolo gramatical X.

Salida. Los símbolos de anticipación generados espontáneamente por elementos en I para elementos nucleares en $ir\_a(I, X)$ y los elementos en I a partir de los cuales los símbolos de anticipación se propagan a elementos nucleares en $ir\_a(I, X)$.

Método. En la figura 4.43 se ilustra el algoritmo. Utiliza un símbolo de anticipación ficticio # para detectar situaciones en que se propagan los símbolos de anticipación. □

```
for cada elemento B → γ·δ en K do begin
    J' := cerradura ({[B → γ·δ, # ]});
    if [A → α·Xβ, a] está en J' donde a no es # then
        el símbolo de preanálisis a se genera espontáneamente para el elemento
            A → αX·β en ir_a (I, X);
    if [A → α·Xβ, # ] está en J' then
        los símbolos de preanálisis se propagan desde
            B → γ·δ en I hasta A → αX·β en ir_a (I, X)
end;
```

Fig. 4.43. Descubrimiento de símbolos de anticipación propagados y espontáneos.

Considérese ahora cómo se buscan los símbolos de anticipación asociados a los elementos de los núcleos de los conjuntos de elementos LR(0). Primero, se sabe que \$ es un símbolo de anticipación para $S' \rightarrow \cdot S$ en el conjunto inicial de elementos LR(0). El algoritmo 4.12 proporciona todos los símbolos de anticipación generados espontáneamente. Después de listar todos estos símbolos de anticipación, se les permite propagarse hasta que la propagación ya no sea posible. Hay muchos enfoques distintos y todos, en algún sentido, tienen registrados los símbolos de anticipación "nuevos" que se han propagado a un elemento, pero que aún no se han propagado del todo. El siguiente algoritmo describe una técnica para propagar los símbolos de preanálisis a todos los elementos.

Algoritmo 4.13. Cálculo eficiente de los núcleos de la colección de conjuntos de elementos LALR(1).

Entrada. Una gramática aumentada G'.

Salida. Los núcleos de la colección de conjuntos de elementos LALR(1) para G'.

Método.

1. Usando el método esbozado anteriormente, constrúyanse los núcleos de los conjuntos de elementos LR(0) para G.
2. Aplíquese el algoritmo 4.12 al núcleo de cada conjunto de elementos LR(0) y símbolo gramatical X para determinar qué símbolos de anticipación se generan espontáneamente para elementos del núcleo en $ir\_a(I, X)$, y a partir de qué elementos en I se propagan los símbolos de anticipación a elementos del núcleo en $ir\_a(I, X)$.
3. Inicialícese una tabla que dé los símbolos de anticipación asociados para cada elemento del núcleo de cada conjunto de elementos. Inicialmente, cada elemento tiene asociados sólo aquellos símbolos de anticipación generados espontáneamente que se determinaron en 2.
4. Háganse pasadas repetidas sobre los elementos del núcleo de todos los conjuntos. Cuando se visite un elemento i, se buscan los elementos del núcleo a los

cuales i propaga sus símbolos de anticipación, usando la información tabulada en 2. El conjunto en curso de símbolos de anticipación para i se añade a los ya asociados a cada uno de los elementos a los que i propaga sus símbolos de anticipación. Se continúa pasando sobre los elementos del núcleo hasta que no se propaguen más símbolos nuevos de anticipación. □

Ejemplo 4.47. Construcción de los núcleos de los elementos LALR(1) para la gramática del ejemplo anterior. En la figura 4.42 se mostraron los núcleos de los elementos LR(0). Cuando se aplica el algoritmo 4.12 al núcleo del conjunto de elementos I_0, se calcula *cerradura*($\{[S' \rightarrow \cdot S,\ \#]\}$), que es

$S' \rightarrow \cdot S,\ \#$
$S \rightarrow \cdot L = R,\ \#$
$S \rightarrow \cdot R,\ \#$
$L \rightarrow \cdot *R,\ \#/=$
$L \rightarrow \cdot \mathbf{id},\ \#/=$
$R \rightarrow \cdot L,\ \#$

Dos elementos en esta cerradura hacen que los símbolos de anticipación se generen espontáneamente. El elemento $[L \rightarrow \cdot *R,\ =]$ hace que el símbolo de anticipación $=$ se genere espontáneamente para el elemento del núcleo $L \rightarrow *\cdot R$ en I_4 y el elemento $[L \rightarrow \cdot \mathbf{id},\ =]$ hace que $=$ se genere espontáneamente para el elemento del núcleo $L \rightarrow \cdot \mathbf{id}$ en I_5.

DE	A
I_0: $S' \rightarrow \cdot S$	I_1: $S' \rightarrow S\cdot$ I_2: $S \rightarrow L\cdot = R$ I_2: $R \rightarrow L\cdot$ I_3: $S \rightarrow R\cdot$ I_4: $L \rightarrow *\cdot R$ I_5: $L \rightarrow \mathbf{id}\cdot$
I_2: $S \rightarrow L\cdot = R$	I_6: $S \rightarrow L = \cdot R$
I_4: $L \rightarrow *\cdot R$	I_4: $L \rightarrow *\cdot R$ I_5: $L \rightarrow \mathbf{id}\cdot$ I_7: $L \rightarrow *R\cdot$ I_8: $R \rightarrow L\cdot$
I_6: $S \rightarrow L = \cdot R$	I_4: $L \rightarrow *\cdot R$ I_5: $L \rightarrow *\cdot R$ I_8: $R \rightarrow L\cdot$ I_9: $S \rightarrow L = R\cdot$

Fig. 4.44. Propagación de símbolos de anticipación.

CON-JUNTO	ELEMENTO	SÍMBOLOS DE ANTICIPACIÓN			
		INIC	PASO 1	PASO 2	PASO 3
I_0:	$S' \rightarrow \cdot S$	\$	\$	\$	\$
I_1:	$S' \rightarrow S\cdot$		\$	\$	\$
I_2:	$S \rightarrow L\cdot = R$		\$	\$	\$
I_2:	$R \rightarrow L\cdot$		\$	\$	\$
I_3:	$S \rightarrow R\cdot$		\$	\$	\$
I_4:	$L \rightarrow *\cdot R$	=	=/\$	=/\$	=/\$
I_5:	$L \rightarrow \mathbf{id}\cdot$	=	=/\$	=/\$	=/\$
I_6:	$S \rightarrow L=\cdot R$			\$	\$
I_7:	$L \rightarrow *R\cdot$		=	=/\$	=/\$
I_8:	$R \rightarrow L\cdot$		=	=/\$	=/\$
I_9:	$S \rightarrow L=R\cdot$				\$

Fig. 4.45. Cálculo de símbolos de anticipación.

En la figura 4.44 se resume el patrón de propagación de los símbolos de anticipación entre los elementos del núcleo determinados en el paso 2 del algoritmo 4.13. Por ejemplo, las transiciones ir a de I_0 con los símbolos S, L, R, * e **id** son, respectivamente, I_1, I_2, I_3, I_4 e I_5. Para I_0 sólo se ha calculado la cerradura del único elemento del núcleo $[S' \rightarrow \cdot S, \#]$. Por tanto, $S' \rightarrow \cdot S$ propaga su símbolo de anticipación a cada elemento del núcleo desde I_1 hasta I_5.

En la figura 4.45 se muestran los pasos 3 y 4 del algoritmo 4.13. La columna etiquetada con INIC muestra los símbolos de anticipación generados espontáneamente para cada elemento del núcleo. En el primer paso, el símbolo de anticipación \$ se propaga desde $S' \rightarrow S$ en I_0 a los seis elementos listados en la figura 4.44. El símbolo de anticipación $=$ se propaga desde $L \rightarrow *\cdot R$ en I_4 a los elementos $L \rightarrow *R\cdot$ en I_7 y $R \rightarrow L\cdot$ en I_8. También se propaga a sí mismo y a $L \rightarrow \mathbf{id}\cdot$ en I_5, pero estos símbolos de anticipación ya están presentes. En el segundo y tercer pasos, el único símbolo de anticipación nuevo propagado es \$, descubierto para los sucesores de I_2 e I_4 en el paso 2 y para el sucesor de I_6 en el paso 3. En el paso 4 no se propaga ningún símbolo de anticipación nuevo, de modo que se muestra el conjunto final de símbolos de anticipación en la columna situada más a la derecha en la figura 4.45.

Obsérvese que el conflicto de desplazamiento/reducción que se presenta en el ejercicio 4.39 al usar el método SLR ha desaparecido con la técnica LALR. La razón es que sólo el símbolo de anticipación \$ está asociado con $R \rightarrow L\cdot$ en I_2, así que no hay conflicto con la acción del análisis sintáctico de desplazar con $=$ generado por el elemento $S \rightarrow L\cdot = R$ en I_2. □

Compactación de las tablas de análisis sintáctico LR

Una gramática de un lenguaje de programación típico con 50 a 100 terminales y 100 producciones puede tener una tabla de análisis sintáctico LALR con varios cientos de estados. La función de acción puede tener fácilmente 20 000 entradas, y cada una necesita al menos ocho bits para codificarse. Evidentemente, una codificación más eficiente que una matriz bidimensional puede ser importante. Posteriormente se mencionarán unas cuantas técnicas empleadas para comprimir los campos de acción e ir a de una tabla de análisis sintáctico LR.

Una técnica útil para condensar los campos de acción es reconocer que generalmente muchas filas de la tabla de acción son idénticas. Por ejemplo, en la figura 4.40, los estados 0 y 3 tienen entradas de acción idénticas, al igual que 2 y 6. Por tanto, se puede ahorrar bastante espacio y no hay que emplear mucho más tiempo, si se crea un apuntador para cada estado de una matriz unidimensional. Los apuntadores para estados con las mismas acciones apuntan a la misma posición. Para acceder a la información de esta matriz, se asigna a cada terminal un número de cero al número de terminales menos uno y se utiliza este entero como desplazamiento desde el valor del apuntador para cada estado. En un estado determinado, la acción de análisis sintáctico para el i-ésimo terminal estará i posiciones después del valor del apuntador para dicho estado.

Se puede ahorrar todavía más espacio a costa de un analizador sintáctico un poco más lento (generalmente considerado un arreglo razonable, puesto que un analizador sintáctico del tipo LR consume sólo una pequeña fracción del tiempo total de compilación) creando una lista para las acciones de cada estado. La lista consta de pares (símbolo terminal-acción). La acción más frecuente para un estado puede situarse al extremo de la lista, y en lugar de un terminal se puede emplear la notación "cualquiera", indicando que si todavía no se ha encontrado el símbolo en curso de entrada en la lista, se debe realizar esa acción, independientemente de cuál sea el símbolo de entrada. Además, las entradas de error pueden sustituirse sin peligro por acciones de reducción, para mayor uniformidad en la fila. Los errores se detectarán después, antes de un movimiento de desplazamiento.

Ejemplo 4.48. Considérese la tabla de análisis sintáctico de la figura 4.31. Primero, obsérvese que las acciones para los estados 0, 4, 6 y 7 coinciden. Todas se pueden representar mediante la lista:

SÍMBOLO	ACCIÓN
id	d5
(	d4
cualquiera	error

El estado 1 tiene una lista similar:

+	d6
$	acep
cualquiera	error

En el estado 2, se pueden sustituir las entradas de error por r2, de modo que la reducción por la producción 2 ocurrirá con cualquier entrada, excepto *. Por tanto, la lista para el estado 2 es:

*	d7
cualquiera	r2

El estado 3 sólo tiene acciones de error y r4. Se pueden sustituir las primeras por las últimas, de modo que la lista para el estado 3 consta sólo del par (cualquiera, r4). Los estados 5, 10 y 11 se pueden tratar de manera similar. La lista para el estado 8 es:

+	d6
)	d11
cualquiera	error

y para el estado 9:

*	d7
cualquiera	r1

□

También se puede codificar la tabla *ir_a* mediante una lista, pero aquí resulta más eficiente hacer una lista de pares para cada no terminal A. Cada par de la lista de A tiene la forma (*estado_actual*, *siguiente_estado*), que indica

$$ir\_a[estado\_actual, A] = siguiente\_estado$$

Esta técnica es útil porque tiende a haber menos estados en cada columna de la tabla de transiciones *ir_a*. La razón es que la transición con el no terminal A sólo puede ser un estado derivable de un conjunto de elementos en el que algunos elementos tienen A inmediatamente a la izquierda de un punto. Ningún conjunto tiene elementos con X e Y inmediatamente a la izquierda de un punto si $X \neq Y$. Por tanto, cada estado aparece a lo sumo en una columna de *ir_a*.

Para reducir más espacio, se observa que nunca se consultan las entradas de error en la tabla ir a. Por tanto, se puede sustituir cada entrada de error por la entrada distinta de error más habitual en su columna. Esta entrada se convierte en el valor por omisión; se representa en la lista para cada columna mediante un par con "cualquiera", en lugar del estado actual.

Ejemplo 4.49. Considérese de nuevo la figura 4.31. La columna para F tiene la entrada 10 para el estado 7 y todas las otras entradas son 3 o error. Se puede sustituir error por 3 y crear para la columna F la lista:

estado_actual	*siguiente_estado*
7	10
cualquiera	3

De manera similar, una lista adecuada para la columna T es:

6	9
cualquiera	2

Para la columna E se puede escoger 1 u 8 como valor por omisión; en ambos casos se necesitan dos entradas. Por ejemplo, se puede crear para la columna E la lista:

4	8
cualquiera	1

□

Si el lector suma el número de entradas en las listas creadas en este ejemplo y en el anterior, y después añade los apuntadores de los estados a las listas de acciones y de los no terminales a las listas de siguiente estado, no quedará impresionado por los ahorros de espacio en la implantación con matrices de la figura 4.31. Sin embargo, no hay que dejarse engañar por este pequeño ejemplo. Para gramáticas prácticas, el espacio necesario para la representación con listas es generalmente inferior al diez por ciento del necesario para la representación con matrices.

También hay que señalar que los métodos de compresión de tablas para los autómatas finitos estudiados en la sección 3.9 se pueden utilizar asimismo para representar tablas de análisis sintáctico LR. La aplicación de estos métodos se estudia en los ejercicios.

4.8 USO DE GRAMATICAS AMBIGUAS

Es un teorema que toda gramática ambigua no es LR, por lo que no está en ninguna de las clases de gramáticas estudiadas en la sección anterior. Sin embargo, ciertos tipos de gramáticas ambiguas son útiles en la especificación e implantación de lenguajes, como se verá en esta sección. Para construcciones de lenguajes como las expresiones, una gramática ambigua proporciona una especificación más natural y corta que cualquier gramática no ambigua equivalente. Otro uso de las gramáticas ambiguas está en el aislamiento de construcciones sintácticas habituales para optimación en casos especiales. Con una gramática ambigua se pueden especificar las construcciones de casos especiales añadiendo cuidadosamente nuevas producciones a la gramática.

Se debe insistir en que, aunque las gramáticas utilizadas son ambiguas, en todos los casos se especifican reglas para eliminar ambigüedades que permiten sólo un árbol de análisis sintáctico para cada frase. De esta manera, la especificación total del lenguaje sigue siendo no ambigua. Hay que señalar también que las construcciones ambiguas se deben usar raramente y de una manera estrictamente controlada, pues de lo contrario no se puede conocer con seguridad el lenguaje que reconoce el analizador.

Uso de precedencia y asociatividad para resolver conflictos en las acciones del análisis sintáctico

Considérense las expresiones en los lenguajes de programación. La siguiente gramática para las expresiones aritméticas con operadores + y *

$$E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id} \tag{4.22}$$

es ambigua porque no especifica la asociatividad o precedencia de los operadores + y *. La gramática no ambigua

$$
\begin{aligned}
&E \rightarrow E + T \mid T \\
&T \rightarrow T * F \mid F \\
&F \rightarrow (E) \mid \mathbf{id}
\end{aligned}
\tag{4.23}
$$

genera el mismo lenguaje, pero da a + una menor precedencia que a *, y convierte a ambos operadores en asociativos por la izquierda. Hay dos razones por las que se prefiere usar la gramática (4.22) en lugar de la (4.23). Primero, como se verá más adelante, se pueden cambiar fácilmente las asociatividades y niveles de precedencia de los operadores + y * sin interferir en las producciones de (4.22) o en el número de estados del analizador sintáctico resultante. Segundo, el analizador sintáctico para (4.23) consumirá una fracción importante de su tiempo reduciendo por las producciones $E \rightarrow T$ y $T \rightarrow F$, cuya única función es asegurar la asociatividad y precedencia. El analizador para (4.22) no perderá tiempo reduciendo por estas producciones *simples*, como se denominan.

I_0:
$E' \rightarrow \cdot E$
$E \rightarrow \cdot E+E$
$E \rightarrow \cdot E*E$
$E \rightarrow \cdot(E)$
$E \rightarrow \cdot\mathbf{id}$

I_1:
$E' \rightarrow E\cdot$
$E \rightarrow E\cdot +E$
$E \rightarrow E\cdot *E$

I_2:
$E \rightarrow (\cdot E)$
$E \rightarrow \cdot E+E$
$E \rightarrow \cdot E*E$
$E \rightarrow \cdot(E)$
$E \rightarrow \cdot\mathbf{id}$

I_3:
$E \rightarrow \mathbf{id}\cdot$

I_4:
$E \rightarrow E+\cdot E$
$E \rightarrow \cdot E+E$
$E \rightarrow \cdot E*E$
$E \rightarrow \cdot(E)$
$E \rightarrow \cdot\mathbf{id}$

I_5:
$E \rightarrow E*\cdot E$
$E \rightarrow \cdot E+E$
$E \rightarrow \cdot E*E$
$F \rightarrow \cdot(E)$
$E \rightarrow \cdot\mathbf{id}$

I_6:
$E \rightarrow (E\cdot)$
$E \rightarrow E\cdot +E$
$E \rightarrow E\cdot *E$

I_7:
$E \rightarrow E+E\cdot$
$E \rightarrow E\cdot +E$
$E \rightarrow E\cdot *E$

I_8:
$E \rightarrow E*E\cdot$
$E \rightarrow E\cdot +E$
$E \rightarrow E\cdot *E$

I_9:
$E \rightarrow (E)\cdot$

Fig. 4.46. Conjuntos de elementos LR(0) para la gramática aumentada (4.22).

En la figura 4.46 se muestran los conjuntos de elementos LR(0) para (4.22) aumentada con $E' \rightarrow E$. Como la gramática (4.22) es ambigua, se generarán conflictos en las acciones del análisis sintáctico cuando se intente producir una tabla de análisis sintáctico LR a partir de los conjuntos de elementos. Los estados correspon-

dientes a los conjuntos de elementos I_7 e I_8 generan estos conflictos. Supóngase que se utiliza el método SLR para construir la tabla de acciones del análisis sintáctico. El conflicto generado por I_7 entre la reducción por $E \rightarrow E + E$ y el desplazamiento con + y * no se puede resolver porque + y * están en SIGUIENTE(E). Por tanto, ambas acciones se pedirían con las entradas + y *. I_8 genera un conflicto similar entre la reducción por $E \rightarrow E * E$ y el desplazamiento con las entradas + y *. De hecho, todos los métodos de construcción de las tablas de análisis sintáctico LR generarán estos conflictos.

Sin embargo, estos problemas se pueden resolver usando la información sobre precedencia y asociatividad para + y *. Considérese la entrada **id** + **id** * **id**, que hace que un analizador sintáctico basado en la figura 4.46 entre en el estado 7 después de procesar **id** + **id**; en particular, el analizador alcanza una configuración

PILA	ENTRADA
$0\ E\ 1 + 4\ E\ 7$	* **id** $

Dando por supuesto que * tiene precedencia sobre +, se sabe que el analizador debe desplazar * a la pila, preparándose para reducir el * y sus **id** que lo rodean a una expresión. Esto es lo que haría el analizador sintáctico SLR de la figura 4.31 para el mismo lenguaje, y lo que haría un analizador sintáctico por precedencia de operadores. Por otra parte, si + tiene precedencia sobre *, se sabe que el analizador debe reducir $E + E$ a E. Por tanto, la precedencia relativa de + seguido de * determina de manera única cómo se debe resolver el conflicto de acción del análisis sintáctico entre reducir $E \rightarrow E + E$ y desplazar con * en el estado 7.

Si la entrada hubiera sido **id** + **id** + **id**, el analizador también alcanzaría una configuración en la que tuviera la pila $0E1 + 4E7$ después de procesar la entrada **id** + **id**. Con la entrada + también hay un conflicto de desplazamiento/reducción en el estado 7. Sin embargo, ahora la asociatividad del operador + determina cómo se debe resolver este conflicto. Si + es asociativo por la izquierda, la acción correcta es reducir por $E \rightarrow E + E$. Es decir, se deben agrupar primero los **id** que rodean al primer +. De nuevo esta elección coincide con lo que harían los analizadores sintácticos SLR o de precedencia de operadores para la gramática del ejemplo 4.34.

En resumen, sabiendo que + es asociativo por la izquierda, la acción del estado 7 con la entrada + debería ser reducir por $E \rightarrow E + E$, y sabiendo que * tiene precedencia sobre +, la acción del estado 7 con la entrada * debería ser desplazar. De manera similar, sabiendo que * es asociativo por la izquierda y tiene precedencia sobre +, se puede argumentar que el estado 8, que aparece en el tope de la pila sólo cuando $E * E$ son los tres símbolos gramaticales del tope, debe tener la acción de reducir $E \rightarrow E * E$, tanto con la entrada + como con *. En el caso de la entrada +, la razón es que * tiene precedencia sobre +, mientras que en el caso de la entrada *, la razón es que * es asociativo por la izquierda.

Si se continúa de esta manera, se obtiene la tabla de análisis sintáctico LR que se muestra en la figura 4.47. Las producciones 1 a 4 son $E \rightarrow E + E$, $E \rightarrow E * E$, $E \rightarrow (E)$ y $E \rightarrow$ **id**, respectivamente. Es interesante comprobar que se produciría una tabla de acciones de análisis sintáctico similar eliminando las reducciones por las producciones simples $E \rightarrow T$ y $T \rightarrow F$ de la tabla SLR para la gramática (4.23) que

se muestra en la figura 4.31. Las gramáticas ambiguas, como la (4.22), se pueden considerar de una manera similar en el contexto del análisis sintáctico LALR y LR canónico.

El caso del *else* ambiguo

Considérese de nuevo la siguiente gramática para las proposiciones condicionales

prop → **if** *expr* **then** *prop* **else** *prop*
| **if** *expr* **then** *prop*
| **otra**

ESTADO	*acción*						*ir_a*
	id	+	*	(	)	$	*E*
0	d3			d2			1
1		d4	d5			acep	
2	d3			d2			6
3		r4	r4		r4	r4	
4	d3			d2			8
5	d3			d2			8
6		d4	d5		d9		
7		r1	d5		r1	r1	
8		r2	r2		r2	r2	
9		r3	r3		r3	r3	

Fig. 4.47. Tabla de análisis sintáctico para la gramática (4.22).

Como se indicó en la sección 4.3, esta gramática es ambigua porque no resuelve la ambigüedad del **else**. Para simplificar el estudio, considérese una abstracción de la gramática anterior, donde *i* representa **if** *expr* **then**, *e* representa **else**, y *a* representa "las demás producciones". Entonces, con la producción aumentada $S' \rightarrow S$, la gramática se puede escribir:

$$S' \rightarrow S$$
$$S \rightarrow iSeS \mid iS \mid a \qquad (4.24)$$

En la figura 4.48 se muestran los conjuntos de elementos LR(0) para la gramática (4.24). La ambigüedad en (4.24) da lugar a un conflicto de desplazamiento/reducción en I_4. Entonces, $S \rightarrow iS \cdot eS$ pide un desplazamiento de *e* y, como SIGUIENTE(S) = {*e*, $}, el elemento $S \rightarrow iS \cdot$ pide una reducción por $S \rightarrow iS$ con la entrada *e*.

Traduciendo de nuevo a la terminología de **if** . . . **then** . . . **else**, dados

if *expr* **then** *prop*

en la pila y **else** como primer símbolo de entrada, ¿se debe desplazar **else** dentro de la pila (por ejemplo, desplazar *e*) o reducir **if** *expr* **then** *prop* a *prop* (es decir, reducir

I_0: $S' \rightarrow \cdot S$
$S \rightarrow \cdot iSeS$
$S \rightarrow \cdot iS$
$S \rightarrow \cdot a$

I_1: $S' \rightarrow S\cdot$

I_2: $S \rightarrow i\cdot SeS$
$S \rightarrow i\cdot S$
$S \rightarrow \cdot iSeS$
$S \rightarrow \cdot iS$
$S \rightarrow \cdot a$

I_3: $S \rightarrow a\cdot$

I_4: $S \rightarrow iS\cdot eS$
$S \rightarrow iS\cdot$

I_5: $S \rightarrow iSe\cdot S$
$S \rightarrow \cdot iSeS$
$S \rightarrow \cdot iS$
$S \rightarrow \cdot a$

I_6: $S \rightarrow iSeS\cdot$

Fig. 4.48. Estados LR(0) para la gramática aumentada (4.24).

por $S \rightarrow iS$)? La respuesta es que se debe desplazar el **else**, porque está "asociado" al **then** anterior. En la terminología de la gramática (4.24), la *e* de la entrada, que representa a **else**, sólo puede formar parte del lado derecho que comienza con la *iS* del tope de la pila. Si lo que sigue a *e* en la entrada no se puede analizar sintácticamente como una *S*, completando el lado derecho de *iSeS*, entonces se demuestra que no hay otro análisis sintáctico posible.

ESTADO	*acción*				*ir_a*
	i	*e*	*a*	$	*S*
0	d2		d3		1
1				acep	
2	d2		d3		4
3		r3		r3	
4		d5		r2	
5	d2		d3		6
6		r1		r1	

Fig. 4.49. Tabla de análisis sintáctico LR para la gramática abstracta del **else** ambiguo.

Se llega a la conclusión de que el conflicto de desplazamiento/reducción en I_4 debe resolverse en favor del desplazamiento con la entrada *e*. En la figura 4.49 se muestra la tabla de análisis sintáctico SLR construida a partir de los conjuntos de elementos de la figura 4.48, utilizando esta resolución del conflicto de acción del análisis sintáctico en I_4 con la entrada *e*. Las producciones 1 a 3 son $S \rightarrow iSeS$, $S \rightarrow iS$ y $S \rightarrow a$, respectivamente.

	PILA	ENTRADA
(1)	0	*iiaea*$
(2)	0*i* 2	*iaea*$
(3)	0*i* 2*i* 2	*aea*$
(4)	0*i* 2*i* 2*a* 3	*ea*$
(5)	0*i* 2*i* 2*S* 4	*ea*$
(6)	0*i* 2*i* 2*S* 4*e* 5	*a*$
(7)	0*i* 2*i* 2*S* 4*e* 5*a* 3	$
(8)	0*i* 2*i* 2*S* 4*e* 5*S* 6	$
(9)	0*i* 2*S* 4	$
(10)	0*S* 1	$

Fig. 4.50. Acciones de análisis sintáctico realizadas con la entrada *iiaea*.

Por ejemplo, con la entrada *iiaea*, el analizador sintáctico realiza los movimientos ilustrados en la figura 4.50, correspondientes a la resolución correcta del "**else** ambiguo". En la línea (5), el estado 4 selecciona la acción de desplazar con la entrada *e*, mientras que en la línea (9), el estado 4 llama a una reducción por $S \rightarrow iS$ con la entrada $.

A modo de comparación, si no se puede utilizar una gramática ambigua para especificar las proposiciones condicionales, entonces habría que utilizar una gramática más voluminosa, del tipo de la (4.9).

Ambigüedades de producciones de casos especiales

Un ejemplo final que muestra la utilidad de las gramáticas ambiguas es el de la introducción de una producción adicional para especificar un caso especial de una construcción sintáctica generada de una forma más general por el resto de la gramática. Cuando se añade la producción adicional, se genera un conflicto de acción del análisis sintáctico. A menudo se puede resolver el conflicto satisfactoriamente mediante una regla para eliminar ambigüedades que indica reducir por la producción del caso especial. La acción semántica asociada a la producción adicional permite entonces que el caso especial sea manejado por un mecanismo más específico.

Un uso interesante de producciones de casos especiales lo hicieron Kernighan y Cherry [1975] en su preprocesador de tipografía de ecuaciones EQN, que se utilizó para la composición de este libro. En EQN, la sintaxis de una expresión matemática se describe mediante una gramática que utiliza un operador de subíndice **sub** y un operador de superíndice **sup**, como se muestra en el fragmento de gramática (4.25). El preprocesador utiliza las llaves para agrupar expresiones compuestas, y *c* se usa como un componente léxico que representa cualquier cadena de texto.

$$
\begin{array}{ll}
(1) & E \rightarrow E \ \textbf{sub} \ E \ \textbf{sup} \ E \\
(2) & E \rightarrow E \ \textbf{sub} \ E \\
(3) & E \rightarrow E \ \textbf{sup} \ E \\
(4) & E \rightarrow \{ \ E \ \} \\
(5) & E \rightarrow c
\end{array}
\qquad (4.25)
$$

La gramática (4.25) es ambigua por varias razones. La gramática no especifica la asociatividad y precedencia de los operadores **sub** y **sup**. Aunque se resuelvan las ambigüedades debidas a la asociatividad y precedencia de **sub** y **sup**, por ejemplo, haciendo que estos dos operadores tengan igual precedencia y asociatividad por la derecha, la gramática seguirá siendo ambigua, debido a que la producción (1) aísla un caso especial de expresiones generadas por las producciones (2) y (3), que son expresiones de la forma E **sub** E **sup** E. La razón para tratar de manera especial las expresiones de este tipo es que muchos tipógrafos preferirían tipografiar una expre-

I_0:
$E' \rightarrow \cdot E$
$E \rightarrow \cdot E$ **sub** E **sup** E
$E \rightarrow \cdot E$ **sub** E
$E \rightarrow \cdot E$ **sup** E
$E \rightarrow \cdot \{ E \}$
$E \rightarrow \cdot c$

I_1:
$E' \rightarrow E\cdot$
$E \rightarrow E\cdot$**sub** E **sup** E
$E \rightarrow E\cdot$**sub** E
$E \rightarrow E\cdot$**sup** E

I_2:
$E \rightarrow \{\cdot E \}$
$E \rightarrow \cdot E$ **sub** E **sup** E
$E \rightarrow \cdot E$ **sub** E
$E \rightarrow \cdot E$ **sup** E
$E \rightarrow \cdot \{ E \}$
$E \rightarrow \cdot c$

I_3:
$E \rightarrow c\cdot$

I_4:
$E \rightarrow E$ **sub**$\cdot E$ **sup** E
$E \rightarrow E$ **sub**$\cdot E$
$E \rightarrow \cdot E$ **sub** E **sup** E
$E \rightarrow \cdot E$ **sub** E
$E \rightarrow \cdot E$ **sup** E
$E \rightarrow \cdot \{ E \}$
$E \rightarrow \cdot c$

I_5:
$E \rightarrow E$ **sup**$\cdot E$
$E \rightarrow \cdot E$ **sub** E **sup** E
$E \rightarrow \cdot E$ **sub** E
$E \rightarrow \cdot E$ **sup** E
$E \rightarrow \cdot \{ E \}$
$E \rightarrow \cdot c$

I_6:
$E \rightarrow E\cdot$**sub** E **sup** E
$E \rightarrow E\cdot$**sub** E
$E \rightarrow E\cdot$**sup** E
$E \rightarrow \{ E\cdot \}$

I_7:
$E \rightarrow E\cdot$**sub** E **sup** E
$E \rightarrow E$ **sub** $E\cdot$**sup** E
$E \rightarrow E\cdot$**sub** E
$E \rightarrow E$ **sub** $E\cdot$
$E \rightarrow E\cdot$**sup** E

I_8:
$E \rightarrow E\cdot$**sub** E **sup** E
$E \rightarrow E\cdot$**sub** E
$E \rightarrow E\cdot$**sup** E
$E \rightarrow E$ **sup** $E\cdot$

I_9:
$E \rightarrow \{ E \}\cdot$

I_{10}:
$E \rightarrow E$ **sub** E **sup**$\cdot E$
$E \rightarrow E$ **sup**$\cdot E$
$E \rightarrow \cdot E$ **sub** E **sup** E
$E \rightarrow \cdot E$ **sub** E
$E \rightarrow \cdot E$ **sup** E
$E \rightarrow \cdot \{ E \}$
$E \rightarrow \cdot c$

I_{11}:
$E \rightarrow E\cdot$**sub** E **sup** E
$E \rightarrow E$ **sub** E **sup** $E\cdot$
$E \rightarrow E\cdot$**sub** E
$E \rightarrow E\cdot$**sup** E
$E \rightarrow E$ **sup** $E\cdot$

Fig. 4.51. Conjunto de elementos LR(0) para la gramática (4.25).

sión del tipo *a* **sub** *i* **sup** 2 como a_i^2, en lugar de como a_i^2. Simplemente añadiendo una producción de caso especial, Kernighan y Cherry consiguieron que EQN produjera esta salida de caso especial.

Para ver cómo se puede tratar esta clase de ambigüedad en el contexto LR, se construirá un analizador sintáctico SLR para la gramática (4.25). En la figura 4.51 se muestran los conjuntos de elementos LR(0) para esta gramática. En esta colección, tres conjuntos de elementos producen conflictos de acción de análisis sintáctico. I_7, I_8 e I_{11} generan conflictos de desplazamiento/reducción con los componentes léxicos **sub** y **sup** porque la asociatividad y precedencia de estos operadores no ha sido especificada. Estos conflictos de acciones del análisis sintáctico se resuelven haciendo que **sub** y **sup** sean de igual precedencia y asociativos por la derecha. Por tanto, en ambos casos se prefiere el desplazamiento.

I_{11} también genera un conflicto de reducción/reducción con las entradas } y $ entre las dos producciones

$E \rightarrow E$ **sub** E **sup** E
$E \rightarrow E$ **sup** E

El estado I_{11} estará en el tope de la pila cuando se haya visto una entrada reducida a E **sub** E **sup** E en la pila. Si se resuelve el conflicto de reducción/reducción en favor de la producción (1), una ecuación de la forma E **sub** E **sup** E se considerará como un caso especial. Con estas reglas para eliminar ambigüedades, se obtiene la tabla de análisis sintácticos SLR de la figura 4.52.

ESTADO	*acción*						*ir_a*
	sub	**sup**	{	}	*c*	$	*E*
0			d2		d3		1
1	d4	d5				acep	
2			d2		d3		6
3	r5	r5		r5		r5	
4			d2		d3		7
5			d2		d3		8
6	d4	d5		d9			
7	d4	d10		r2		r2	
8	d4	d5		r3		r3	
9	r4	r4		r4		r4	
10			d2		d3		11
11	d4	d5		r1		r1	

Fig. 4.52. Tabla de análisis sintáctico para la gramática (4.25).

Escribir gramáticas no ambiguas que factoricen construcciones sintácticas de casos especiales es muy difícil. Para apreciar el alcance de la dificultad, se invita al lector a construir una gramática no ambigua equivalente para (4.25) que aísle expresiones de la forma E **sub** E **sup** E.

Recuperación de errores en el análisis sintáctico LR

Un analizador sintáctico LR detectará un error cuando consulte la tabla de acciones de análisis sintáctico y encuentre una entrada de error. Los errores nunca se detectan consultando la tabla de transiciones ir a. A diferencia de un analizador sintáctico por precedencia de operadores, un analizador LR anunciará un error tan pronto como no haya una continuación válida para la parte de entrada examinada hasta entonces. Un analizador sintáctico LR canónico nunca hará ni una sola reducción antes de anunciar un error. Los analizadores SLR y LALR pueden hacer varias reducciones antes de anunciar un error, pero nunca desplazarán un símbolo de entrada erróneo a la pila.

En el análisis sintáctico LR, se puede implantar una recuperación de errores en modo de pánico como sigue. Se examina la pila hasta encontrar un estado s con un valor de ir a para un determinado no terminal A. Entonces se desechan cero o más símbolos de entrada hasta encontrar un símbolo a que pueda seguir legalmente a A. Entonces, el analizador sintáctico mete en la pila el estado $ir\_a[s, A]$ y prosigue normalmente el análisis sintáctico. Puede haber más de una opción para el no terminal A. Generalmente, estas serían no terminales que representan las partes más importantes de un programa, como una expresión, una proposición o un bloque. Por ejemplo, si A es el no terminal *prop*, a puede ser el símbolo de punto y coma o **end**.

Este método de recuperación intenta aislar la frase que tiene el error sintáctico. El analizador determina que una cadena derivable de A tiene un error. Parte de esta cadena ya ha sido procesada, y el resultado de este procesamiento es una secuencia de estados en el tope de la pila. El resto de la cadena aún está en la entrada, y el analizador intenta saltar el resto de la cadena buscando un símbolo en la entrada que pueda seguir legítimamente a A. Eliminando estados de la pila, saltando la entrada e introduciendo $ir\_a[s, A]$ en la pila, el analizador supone que ha encontrado un caso de A y continúa el análisis normal.

La recuperación a nivel de frase se implanta examinando cada entrada de error en la tabla de análisis sintáctico LR y decidiendo, basándose en el uso del lenguaje, los errores de los programadores que más probablemente darían lugar a dicho error. Entonces se puede construir un procedimiento de recuperación apropiado; presumiblemente, el tope de la pila o los primeros símbolos de entrada o ambos se modificarían de la forma adecuada a cada entrada de error.

Comparado con los analizadores sintácticos por precedencia de operadores, el diseño de rutinas específicas para el manejo de errores para un analizador LR es relativamente sencillo. Por ejemplo, no hay que preocuparse por las reducciones defectuosas; cualquier reducción pedida por un analizador LR es correcta. Por tanto, se puede llenar cada entrada en blanco en el campo de acción con un apuntador a una rutina de error que realizará una acción apropiada elegida por el diseñador del compilador. Las acciones pueden incluir la inserción o eliminación de símbolos de la pila, de la entrada o de ambas, o la alteración y transposición de símbolos de entrada, al igual que para el analizador sintáctico por precedencia de operadores. Lo mismo que para ese analizador, se deben elegir las opciones de modo que el analizador LR no pueda caer en un lazo infinito. A este respecto, es suficiente con una estrategia que garantice que al menos un símbolo de entrada será eliminado o quizá

desplazado, o que en algún momento la pila se reducirá si se ha alcanzado el final de la entrada. Se debe evitar extraer un estado de la pila que abarque un no terminal, porque esta modificación elimina de la pila una construcción que ya se había analizado con éxito.

Ejemplo 4.50. Considérese de nuevo la gramática de expresiones

$$E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id}$$

En la figura 4.53 se muestra la tabla de análisis sintáctico LR a partir de la figura 4.47 para esta gramática, modificada para la detección y recuperación de errores. Se ha cambiado cada estado que pide una reducción determinada con algunos símbolos de entrada sustituyendo las entradas de error en dicho estado por la reducción. Esta modificación pospone la detección de errores hasta que se hayan hecho una o más reducciones, pero, de todos modos, el error será detectado antes de que tenga lugar cualquier movimiento de desplazamiento. Las restantes entradas en blanco de la figura 4.47 se han sustituido por llamadas a rutinas de errores.

ESTADO	*acción*						*ir_a*
	id	+	*	(	)	$	*E*
0	d3	e1	e1	d2	e2	e1	1
1	e3	d4	d5	e3	e2	acep	
2	d3	e1	e1	d2	e2	e1	6
3	r4	r4	r4	r4	r4	r4	
4	d3	e1	e1	d2	e2	e1	7
5	d3	e1	e1	d2	e2	e1	8
6	e3	d4	d5	e3	d9	e4	
7	r1	r1	d5	r1	r1	r1	
8	r2	r2	r2	r2	r2	r2	
9	r3	r3	r3	r3	r3	r3	

Fig. 4.53. Tabla de análisis sintáctico LR con rutinas de error.

Las rutinas de error son como sigue. Debe observarse la similitud entre estas acciones y los errores que representan y las acciones de error del ejemplo 4.32 (de precedencia de operadores). Sin embargo, el caso e1 del analizador LR lo maneja el procesador de reducciones del analizador sintáctico por precedencia de operadores.

e1: /* Esta rutina se llama desde los estados 0, 2, 4 y 5, los cuales esperan el comienzo de un operando, ya sea un **id** o un paréntesis izquierdo. En vez de eso, se encontró un operador, + o *, o el fin de la entrada. */
introdúzcase un **id** imaginario en la pila y cúbrase con el estado 3 (la transición ir a de los estados 0, 2, 4 y 5 con **id**)[5]
emítase el diagnóstico "falta operando"

[5] Obsérvese que en la práctica los símbolos de la gramática no se colocan en la pila. Es útil imaginarlos ahí para recordar los símbolos que "representan" los estados.

e2: /* Esta rutina se llama desde los estados 0, 1, 2, 4 y 5 al encontrar un paréntesis. */
elimínese el paréntesis derecho de la entrada
emítase el diagnóstico "paréntesis derecho no equilibrado"

e3: /* Esta rutina se llama desde los estados 1 ó 6 cuando esperan un operador y se encuentra un **id** o un paréntesis derecho. */
introdúzcase + en la pila y cúbrase con el estado 4.
emítase el diagnóstico "falta operador"

e4: /* Esta rutina se llama desde el estado 6 cuando se encuentra el fin de la entrada. El estado 6 espera un operador a un paréntesis derecho. */
introdúzcase un paréntesis derecho en la pila y cúbrase con el estado 9.
emítase el diagnóstico "falta paréntesis derecho"

PILA	ENTRADA	MENSAJE DE ERROR Y ACCIÓN
0	**id** +)$	
0**id**3	+)$	
0*E*1	+)$	
0*E*1 + 4	)$	
0*E*1 + 4	$	`"paréntesis derecho no equilibrado"` e2 elimina el paréntesis derecho
0*E*1 + 4**id**3	$	`"falta operando"` el mete **id** 3 en la pila
0*E*1 + 4*E*7	$	
0*E*1	$	

Fig. 4.54. Movimientos de análisis y de recuperación de errores realizados por el analizador sintáctico LR.

Con la entrada errónea **id** +) del ejemplo 4.32, en la figura 4.54 se muestra la secuencia de configuraciones en que ha entrado el analizador sintáctico. □

4.9 GENERADORES DE ANALIZADORES SINTACTICOS

Esta sección muestra cómo utilizar un generador de analizadores sintácticos para facilitar la construcción de la etapa inicial de un compilador. El generador de analizadores sintácticos LALR YACC se usará como base de esta exposición, puesto que implanta muchos de los conceptos estudiados en las dos secciones anteriores y es fácil de encontar. YACC significa "otro compilador de compiladores más" (del inglés *Yet Another Compiler-Compiler*), lo que refleja la popularidad de los generadores de analizadores sintácticos al principio de los años setenta, cuando S. C. John-

son creó la primera versión de YACC. Este generador se encuentra disponible como una orden del sistema UNIX, y se ha utilizado para facilitar la implantación de cientos de compiladores.

El generador de analizadores sintácticos YACC

Se puede construir un traductor utilizando YACC de la forma que se ilustra en la figura 4.55. Primero, se prepara un archivo, por ejemplo `traduce.y`, que contiene una especificación en YACC del traductor. La orden del sistema UNIX

```
yacc traduce.y
```

transforma al archivo `traduce.y` en un programa escrito en C llamado `y.tab.c` usando el método LALR esbozado en el algoritmo 4.13. El programa `y.tab.c` es una representación de un analizador sintáctico escrito en C, junto con otras rutinas en C que el usuario pudo haber preparado. La tabla de análisis sintáctico LALR se comprime como se describió en la sección 4.7. Al compilar `y.tab.c` junto con la biblioteca `ly` que contiene el programa de análisis sintáctico LR utilizando la orden

```
cc y.tab.c -ly
```

se obtiene el programa objeto deseado `a.out` que realiza la traducción especificada por el programa original en YACC[6]. Si se necesitan otros procedimientos, se pueden compilar o cargar con `y.tab.c`, igual que en cualquier programa en C.

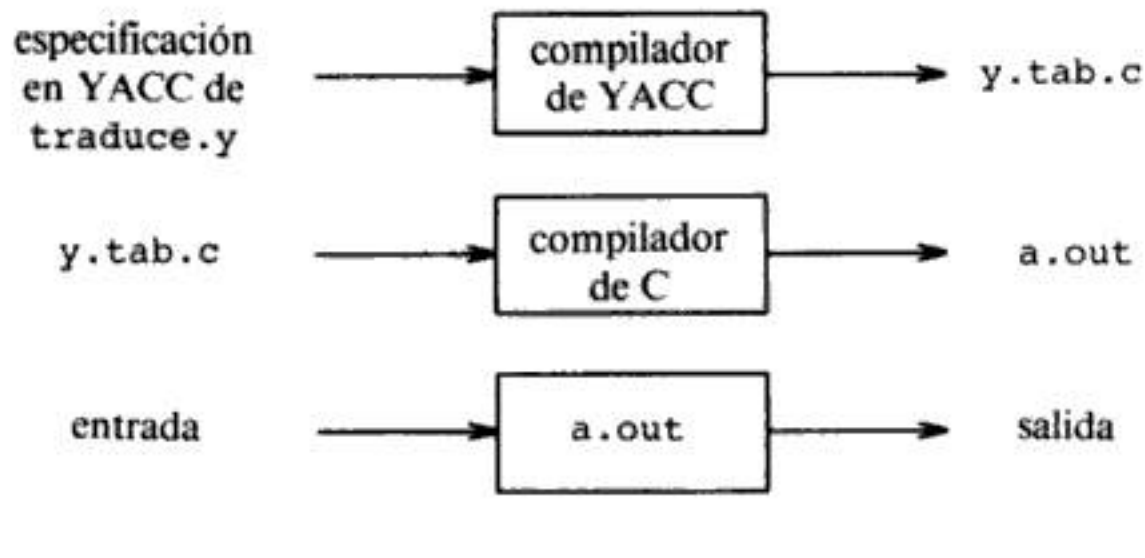


Fig. 4.55. Creación de un traductor de entrada/salida con YACC.

Un programa fuente en YACC tiene tres partes:

```
declaraciones
%%
reglas de traducción
%%
rutinas en C de apoyo
```

[6] El nombre `ly` depende del sistema.

Ejemplo 4.51. Para ilustrar la preparación de un programa fuente en YACC, se construirá una calculadora sencilla de escritorio que lea una expresión aritmética, la evalúe y después imprima su valor numérico. Se construirá la calculadora de escritorio comenzando con la siguiente gramática para expresiones aritméticas:

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid \textbf{dígito}$$

El componente léxico **dígito** es un solo dígito entre 0 y 9. En la figura 4.56 se muestra un programa en YACC para la calculadora de escritorio derivado de esta gramática. □

La parte de declaraciones. Hay dos secciones opcionales en la parte de declaraciones de un programa en YACC. En la primera sección, se ponen declaraciones ordinarias en C, delimitadas por `%{` y `%}`. Aquí se sitúan las declaraciones de todas las temporales usadas por las reglas de traducción o los procedimientos de la segunda y tercera secciones. En la figura 4.56, esta sección sólo contiene la proposición **include**

```
#include <ctype.h>
```

que hace que el preprocesador de C incluya el archivo de encabezamiento estándar `<ctype.h>` que contiene el predicado `isdigit`.

También en la parte de declaraciones hay declaraciones de los componentes léxicos de la gramática. En la figura 4.56, la proposición

```
%token DIGITO
```

declara que `DIGITO` es un componente léxico (*token*). Los componentes léxicos que se declaran en esta sección se pueden utilizar después en la segunda y tercera partes de la especificación en YACC.

La parte de las reglas de traducción. En la parte de la especificación en YACC después del primer par `%%` se ponen las reglas de traducción. Cada regla consta de una producción de la gramática y la acción semántica asociada. Un conjunto de producciones que se han escrito

$$\text{<lado izquierdo>} \rightarrow \text{<alt 1>} \mid \text{<alt 2>} \mid \ldots \mid \text{<alt } n\text{>}$$

en YACC se escribiría

```
<lado izquierdo>   :  <alt 1>   { acción semántica 1 }
                   |  <alt 2>   { acción semántica 2 }
                   .  .  .
                   |  <alt n>   { acción semántica n }
                   ;
```

En una producción en YACC, un carácter simple entrecomillado 'c' se considera como el símbolo terminal c, y las cadenas sin comillas de letras y dígitos no declarados como componentes léxicos se consideran no terminales. Los lados derechos alternativos se pueden separar con una barra vertical, y un símbolo de punto y coma

```
%{
#include <ctype.h>
%}

%token DIGITO

%%
línea    :    expr '\n'              { printf("%d\n", $1); }
         ;
expr     :    expr '+' término       { $$ = $1 + $3; }
         |    término
         ;
término  :    término '*' factor     { $$ = $1 * $3; }
         |    factor
         ;
factor   :    '(' expr ')'           { $$ = $2; }
         |    DIGITO
         ;

%%
yylex() {
    int c;
    c = getchar();
    if (isdigit(c)) {
       yylval = c-'0';
       return DIGIT;
    }
    return c;
}
```

Fig. 4.56. Especificación en YACC de una calculadora de escritorio sencilla.

sigue a cada lado izquierdo con sus alternativas y sus acciones semánticas. El primer lado izquierdo se considera como el símbolo inicial.

Una acción semántica en YACC es una secuencia de proposiciones en C. En una acción semántica, el símbolo $$ se refiere al valor del atributo asociado con el no terminal del lado izquierdo, mientras que `$i` se refiere al valor asociado con el i-ésimo símbolo gramatical (terminal o no terminal) del lado derecho. La acción semántica se realiza siempre que se reduzca por la producción asociada, por lo que normalmente la acción semántica calcula un valor para $$ en función de los `$i`. En la especificación en YACC las dos producciones de E se han escrito

$$E \rightarrow E + T \mid T$$

y sus acciones semánticas asociadas,

```
expr     :    expr '+' término       { $$ = $1 + $3; }
         |    término
         ;
```

Obsérvese que el no terminal `término` de la primera producción es el tercer símbolo gramatical del lado derecho, mientras que '+' es el segundo. La acción semántica asociada a la primera producción añade el valor de `expr` y de `término`, y asigna el resultado como el valor del no terminal `expr` del lado izquierdo. Se ha omitido totalmente la acción semántica para la segunda producción, puesto que copiar el valor es la acción por omisión para producciones con un solo símbolo gramatical del lado derecho. En general, `{ $$ = $1; }` es la acción semántica por omisión.

Obsérvese que se ha añadido una nueva producción inicial

```
línea    :   expr '\n'    { printf("%d\n", $1); }
```

a la especificación en YACC. Esta producción establece que una entrada para la calculadora de bolsillo debe ser una expresión seguida de un carácter de nueva línea. La acción semántica asociada con esta producción imprime el valor decimal de la expresión seguida de un carácter de nueva línea.

La parte de las rutinas de apoyo en C. La tercera parte de una especificación en YACC consta de rutinas de apoyo escritas en C. Se debe proporcionar un análisis léxico de nombre `yylex()`. En caso necesario se pueden agregar otros procedimientos, como rutinas de recuperación de errores.

El analizador léxico `yylex()` produce pares formados por un componente léxico y su valor de atributo asociado. Si se devuelve un componente léxico como `DIGITO`, el componente léxico se debe declarar en la primera sección de la especificación en YACC. El valor del atributo asociado a un componente léxico se comunica al analizador sintáctico mediante una variable YACC `yylval` definida por.

El analizador léxico de la figura 4.56 es muy tosco. Lee caracteres de entrada de uno en uno, utilizando la función `getchar()` de C. Si el carácter es un dígito, el valor del dígito se almacena en la variable `yylval`, y se devuelve el componente léxico `DIGITO`. De lo contrario, se devuelve el propio carácter como componente léxico.

Uso de YACC con gramáticas ambiguas

A continuación se modifica la especificación en YACC de forma que la calculadora de escritorio obtenida sea más útil. Primero se permitirá a la calculadora de escritorio evaluar una secuencia de expresiones, una por línea. También se admitirán líneas en blanco entre las expresiones. Esto se hace cambiando la primera regla por

```
líneas    :    líneas expr '\n' { printf("%g\n", $2); }
          |    líneas '\n'
          |
          ;
```

En YACC, una alternativa vacía, como lo es la tercera línea, indica ϵ.

Segundo, se ampliará la clase de expresiones para incluir números en lugar de dígitos simples y para incluir los operadores aritméticos $+$, $-$ (tanto binarios como unarios), * y /. La manera más fácil de especificar esta clase de expresiones es utilizar la gramática ambigua

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid (E) \mid - E \mid \textbf{número}$$

```
%{
#include <ctype.c>
#include <stdio.h>
#define YYSTYPE double /* se usa el tipo double para la pila
                          de YACC */
%}

%token NUMERO
%left '+' '-'
%left '*' '/'
%right MENOSU

%%
líneas    :    líneas expr '\n'     { printf("%g\n", $2); }
           |    líneas '\n'
           |    /* ϵ */
           ;
expr      :    expr '+' expr        { $$ = $1 + $3; }
           |    expr '-' expr        { $$ = $1 - $3; }
           |    expr '*' expr        { $$ = $1 * $3; }
           |    expr '/' expr        { $$ = $1 / $3; }
           |    '(' expr ')'         { $$ = $2; }
           |    '-' expr %prec MENOSU { $$ = - $2; }
           |    NUMERO
           ;
%%
yylex() {
    int c;
    while ( ( c = getchar() ) == ' ' );
    if ( (c = = '.') || (isdigit(c)) ) {
       ungetc(c, stdin);
       scanf("%lf", &yylval);
       return NUMERO;
       }
    return c;
}
```

Fig. 4.57. Especificación en YACC para una calculadora de escritorio más avanzada.

En la figura 4.57 se muestra la especificación en YACC obtenida.

Como la gramática de la especificación en YACC de la figura 4.57 es ambigua, el algoritmo LALR generará conflictos en las acciones del analizador sintáctico. YACC informará del número de conflictos en las acciones del análisis sintáctico que se generen. Se puede obtener una descripción de los conjuntos de elementos y de los conflictos en las acciones de análisis sintáctico invocando a YACC con la opción `-v`. Esta opción genera un archivo adicional `y.output` que contiene los núcleos de

los conjuntos de elementos encontrados por el analizador sintáctico, una descripción de los conflictos en las acciones del análisis generados por el algoritmo LALR y una representación legible de la tabla de análisis sintáctico LR que muestra cómo se resolvieron los conflictos de las acciones del análisis sintáctico. Siempre que YACC informa de que se han encontrado conflictos en las acciones del análisis sintáctico, es conveniente crear y consultar el archivo `y.output` para saber por qué se generaron los conflictos en las acciones del análisis sintáctico y si se resolvieron correctamente.

A menos que se le ordene lo contrario, YACC resolverá todos los conflictos en las acciones del análisis sintáctico utilizando las dos reglas siguientes:

1. Un conflicto de reducción/reducción se resuelve eligiendo la producción en conflicto que se haya listado primero en la especificación en YACC. Así que para lograr la resolución correcta en la gramática de tipografía (4.25), basta con listar la producción (1) antes que la producción (3).
2. Un conflicto de desplazamiento/reducción se resuelve en favor del desplazamiento. Esta regla resuelve correctamente el conflicto de desplazamiento/reducción que se deriva del **else** ambiguo.

Como estas reglas que se siguen por omisión, no siempre reflejan lo que quiere el escritor del compilador, YACC proporciona un mecanismo general para resolver los conflictos de desplazamiento/reducción. En la parte de declaraciones, se pueden asignar precedencias y asociatividades a los terminales. La declaración

```
%left '+' '-'
```

hace que + y – tengan la misma precedencia y que sean asociativos por la izquierda. Se puede declarar que un operador es asociativo por la derecha diciendo

```
%right '^'
```

y se puede obligar a un operador a ser un operador binario no asociativo (por ejemplo, dos casos del operador no se pueden combinar en absoluto) diciendo

```
%nonassoc '<'
```

A los componentes léxicos se les dan precedencias en el orden en que aparecen en la parte de declaraciones, los más bajos los primeros. Los componentes léxicos de la misma declaración tienen la misma precedencia. Así, la declaración

```
%right MENOSU
```

de la figura 4.57 da al componente léxico `MENOSU` un nivel de precedencia mayor que el de los cinco terminales anteriores.

YACC resuelve los conflictos de desplazamiento/reducción asociando una precedencia y asociatividad a cada producción implicada en un conflicto, así como a cada terminal implicado en un conflicto. Si debe elegir entre desplazar el símbolo de entrada a y reducir por la producción $A \rightarrow \alpha$, YACC reduce si la precedencia de la producción es mayor que la de a o si las precedencias son las mismas y la asociatividad de la producción es `left`. De lo contrario, se elige la acción de desplazar.

Generalmente, la precedencia de una producción se considera igual a la de su terminal situado más a la derecha. En la mayoría de los casos, esta es la decisión sensata. Por ejemplo, dadas las producciones

$$E \rightarrow E + E \mid E * E$$

es preferible reducir por $E \rightarrow E + E$ con símbolo de anticipación $+$, porque $+$ en el lado derecho tiene la misma precedencia que el símbolo de anticipación, pero es asociativo por la izquierda. Con el símbolo de anticipación *, es preferible desplazar, porque el símbolo de anticipación tiene mayor precedencia que el $+$ en la producción.

En las situaciones en que el terminal situado más a la derecha no proporcione la precedencia adecuada a una producción, se puede forzar una precedencia añadiendo a la producción la etiqueta

```
%prec <terminal>
```

Entonces, la precedencia y asociatividad de la producción serán las mismas que las del terminal, que se define seguramente en la sección de declaraciones. YACC no informa de los conflictos de desplazamiento/reducción que se resuelven utilizando este mecanismo de precedencia y asociatividad.

Este "terminal" puede ser un marcador, como MENOSU de la figura 4.57; este terminal no es devuelto por el analizador léxico, sino que se declara tan sólo para definir una precedencia para una producción. En la figura 4.57, la declaración

```
right MENOSU
```

asigna al componente léxico una precedencia superior a * y /. En la parte de las reglas de traducción, la etiqueta

```
%prec MENOSU
```

al final de la producción

```
expr    :   '-' expr
```

hace que el operador menos unario de esta producción tenga mayor precedencia que cualquier otro operador.

Creación de analizadores léxicos para YACC con LEX

LEX se diseñó para producir analizadores léxicos para utilizar con YACC. La biblioteca `ll` de LEX proporcionará un programa manejador llamado `yylex()`, que es el nombre que YACC exige para su analizador léxico. Si se usa LEX para producir el analizador léxico, se sustituye la rutina `yylex()` en la tercera parte de la especificación en YACC por la proporción

```
#include "lex.yy.c"
```

y cada acción de LEX devuelve un terminal conocido por YACC. Utilizando la proposición `#include "lex.yy.c"`, el programa `yylex` tiene acceso a los nombres de los componentes léxicos de YACC, puesto que el archivo de salida de LEX se compila como parte del archivo de salida de YACC `y.tab.c`.

Con el sistema UNIX, si la especificación en LEX está en el archivo `primero.l` y la especificación en YACC está en `segundo.y`, se puede decir

```
lex primero.l
yacc segundo.y
cc y.tab.c -ly -ll
```

para obtener el traductor deseado.

Se puede utilizar la especificación en LEX de la figura 4.58 en lugar del analizador léxico de la figura 4.57. El último patrón es `\n¦.` puesto que `.` en LEX concuerda con cualquier carácter, excepto con el de nueva línea.

Recuperación de errores en YACC

En YACC se puede realizar la recuperación de errores usando una forma de producciones de error. Primero, el usuario decide qué no terminales "principales" tendrán recuperación de errores asociados a ellos. Las elecciones típicas son algún subconjunto de los no terminales que generan expresiones, proposiciones, bloques y procedimientos. Entonces, el usuario añade a la gramática producciones de error de la forma $A \rightarrow$ **error** α, donde A es un no terminal principal y α es una cadena de símbolos gramaticales, quizá la cadena vacía; **error** es una palabra reservada de YACC. YACC generará un analizador sintáctico a partir de dicha especificación, considerando las producciones de error como producciones normales.

```
número      [0-9]+\.?¦[0-9]*\.[0-9]+
%%
[ ]         { /* salta los espacios en blanco */ }
{número}    { sscanf(yytext, "%lf", &yylval);
              return NUMERO; }
\n¦.        { return yytext[0]; }
```

Fig. 4.58. Especificación en LEX para `yylex()` de la figura 4.57.

Sin embargo, cuando el analizador sintáctico generado por YACC encuentra un error, trata a los estados cuyos conjuntos de elementos contengan producciones de error de manera especial. Al encontrar un error, YACC extrae símbolos de su pila hasta que encuentre el estado más alto cuyo conjunto de elementos subyacente incluya un elemento de la forma $A \rightarrow \cdot$ **error** α. Entonces, el analizador "desplaza" un componente léxico ficticio **error** a la pila, como si hubiera visto el componente léxico **error** en su entrada.

Cuando α es ϵ, se produce inmediatamente una reducción a A y se invoca la acción semántica asociada a la producción $A \rightarrow$ **error** (que puede ser una rutina de recuperación de errores especificada por el usuario). Después, el analizador sintáctico elimina símbolos de entrada hasta que encuentra un símbolo de entrada con el que pueda proseguir el análisis sintáctico normal.

Si α no es vacía, YACC salta la entrada buscando una subcadena que se pueda reducir a α. Si α consta solamente de terminales, entonces busca esta cadena de terminales en la entrada, y los "reduce" desplazándolos a la pila. Llegado a este punto, el analizador tendrá **error** α en la cima de su pila. Entonces, el analizador reducirá **error** α a *A*, y proseguirá el análisis sintáctico normal.

Por ejemplo, una producción de error de la forma

prop → **error** ;

especificaría al analizador que debe saltar hasta después del siguiente símbolo de punto y coma al ver un error, y suponer que se ha encontrado una proposición. La rutina semántica para esta producción de error no necesitaría manipular la entrada, sino que podría generar un mensaje de diagnóstico y levantar una bandera para inhibir la generación de código objeto, por ejemplo.

```
%{
#include <ctype.c>
#include <stdio.h>
#define YYSTYPE double /* se usa el tipo double para la pila
                          de YACC */
%}

%token NUMERO
%left '+' '-'
%left '*' '/'
%right MENOSU

%%
líneas    :    líneas expr '\n'    { printf("%g\n", $2); }
           |   líneas '\n'
           |   /* vacía */
           |   error '\n' { yyerror("reintroduzca la última
                            línea:"); yyerrok; }
           ;
expr      :    expr '+' expr       { $$ = $1 + $3; }
           |   expr '-' expr       { $$ = $1 - $3; }
           |   expr '*' expr       { $$ = $1 * $3; }
           |   expr '/' expr       { $$ = $1 / $3; }
           |   '(' expr ')'        { $$ = $2; }
           |   '-' expr %prec MENOSU { $$ = - $2; }
           |   NUMERO
           ;
%%
#include "lex.yy.c"
```

Fig. 4.59. Calculadora de escritorio con recuperación de errores.

Ejemplo 4.52. En la figura 4.59 se muestra la calculadora de escritorio en YACC de la figura 4.57 con la producción de error

```
líneas  :  error '\n'
```

Esta producción de error hace que la calculadora de escritorio suspenda el análisis sintáctico normal cuando se encuentra un error de sintaxis en una línea de entrada. Al encontrar el error, el analizador sintáctico de la calculadora de escritorio comienza a extraer símbolos de su pila hasta que encuentra un estado que tenga una acción de desplazar con el componente léxico **error**. Dicho estado es el 0 (en este ejemplo, es el único de dichos estados), puesto que sus elementos incluye

líneas → ·**error** '\n'

Asimismo, el estado 0 se encuentra siempre en el fondo de la pila. El analizador sintáctico desplaza al componente léxico **error** a la pila, y después se dispone a saltar la entrada hasta que encuentre un carácter de nueva línea. Llegado a este punto, el analizador desplaza el carácter de nueva línea a la pila, reduce **error** '\n' a *líneas*, y emite el mensaje diagnóstico "reintroducir la última línea:". La rutina especial de YACC `yyerrok` devuelve el analizador sintáctico a su modo habitual de operación. □

EJERCICIOS

4.1 Considérese la gramática

$$S \rightarrow (L) \mid a$$
$$L \rightarrow L , S \mid S$$

a) ¿Cuáles son los terminales, no terminales y el símbolo inicial?
b) Encuéntrense árboles de análisis sintáctico para las siguientes frases:
   i) (a, a)
   ii) $(a, (a, a))$
   iii) $(a, ((a, a), (a, a)))$
c) Constrúyase una derivación por la izquierda para cada una de las frases de b).
d) Constrúyase una derivación por la derecha para cada una de las frases de b).
*e) ¿Qué lenguaje genera esta gramática?

4.2 Considérese la gramática

$$S \rightarrow aSbS \mid bSaS \mid \epsilon$$

a) Demuéstrese que esta gramática es ambigua construyendo dos derivaciones por la izquierda distintas para la frase *abab*.
b) Constrúyanse las derivaciones por la derecha correspondiente a *abab*.
c) Constrúyanse los árboles de análisis sintáctico correspondientes a *abab*.
*d) ¿Qué lenguaje genera esta gramática?

4.3 Considérese la gramática

bexpr → *bexpr* **or** *bterm* | *bterm*
bterm → *bterm* **and** *bfactor* | *bfactor*
bfactor → **not** *bfactor* | (*bexpr*) | **true** | **false**

a) Constrúyase un árbol de análisis sintáctico para la frase **not (true or false)**.
b) Demuéstrese que esta gramática genera todas las expresiones booleanas.
*c) ¿Es ambigua esta gramática? ¿Por qué?

4.4 Considérese la gramática $R \rightarrow R' \mid ' R \mid RR \mid R* \mid (R) \mid a \mid b$

Obsérvese que la primera barra vertical es el símbolo "o", no un separador entre alternativas.

a) Demuéstrese que esta gramática genera todas las expresiones regulares sobre los símbolos a y b.
b) Demuéstrese que esta gramática es ambigua.
*c) Constrúyase una gramática no ambigua equivalente que dé a los operadores *, concatenación y | las precedencias y asociatividades definidas en la sección 3.3
d) Constrúyase un árbol de análisis sintáctico en ambas gramáticas para la frase $a \mid b*c$.

4.5 Se propone la siguiente gramática para las proposiciones **if-then-else** para remediar la ambigüedad del **else**

prop → **if** *expr* **then** *prop*
| *prop_emparejada*
prop_emparejada → **if** *expr* **then** *prop_emparejada* **else** *prop*
| **otras**

Demuéstrese que esta gramática sigue siendo ambigua.

***4.6** Inténtese diseñar una gramática para cada uno de los siguientes lenguajes. ¿Qué lenguajes son regulares?

a) El conjunto de todas las cadenas de símbolos 0 y 1 tales que todo 0 va seguido inmediatamente de al menos un 1.
b) Las cadenas en símbolos 0 y 1 con un número igual de 0 y 1.
c) Las cadenas de símbolos 0 y 1 con un número distinto de símbolos 0 y 1.
d) Las cadenas de símbolos 0 y 1 en las que 011 no aparece como una subcadena.
e) Las cadenas de símbolos 0 y 1 de la forma xy, donde $x \neq y$.
f) Las cadenas de símbolos 0 y 1 de la forma xx.

4.7 Constrúyase una gramática para las expresiones de cada uno de los siguientes lenguajes:

a) Pascal
b) C
c) FORTRAN 77
d) Ada
e) LISP

4.8 Constrúyanse gramáticas no ambiguas para las proposiciones de cada uno de los lenguajes del ejercicio 4.7.

4.9 Se pueden usar operadores del tipo de las expresiones regulares en los lados derechos de las producciones gramaticales. Se pueden utilizar los corchetes para indicar una parte opcional de una producción. Por ejemplo, se puede escribir

prop → **if** *expr* **then** *prop* [**else** *prop*]

para indicar una proposición **else** opcional. En general, $A \rightarrow \alpha\ [\ \beta\ ]\ \gamma$ es equivalente a las dos producciones $A \rightarrow \alpha\beta\gamma$ y $A \rightarrow \alpha\gamma$.

Las llaves se pueden usar para indicar una frase que se puede repetir cero o más veces. Por ejemplo,

prop → **begin** *prop* { ; *prop* } **end**

indica una lista de proposiciones *prop* separadas por símbolos de punto y coma encerrada entre **begin** y **end**. En general, $A \rightarrow \alpha\ \{\ \beta\ \}\ \gamma$ es equivalente a

$A \rightarrow \alpha B\gamma$ y $B \rightarrow \beta B \mid \epsilon$.

En un sentido, [β] representa a la expresión regular $\beta \mid \epsilon$ y { β } representa a β*. Se pueden generalizar estas notaciones para admitir cualquier expresión regular de los símbolos gramaticales de los lados derechos de las producciones.

a) Modifíquese la producción de *prop* anterior de modo que aparezca en el lado derecho una lista de *prop* terminada con un símbolo de punto y coma.
b) Dése un conjunto de producciones independientes del contexto que generen el mismo conjunto de cadenas que $A \rightarrow B*a(C \mid D)$.
c) Muéstrese cómo sustituir cualquier producción $A \rightarrow r$, donde r es una expresión regular, por una serie finita de producciones independientes del contexto.

4.10 La siguiente gramática genera declaraciones para un identificador simple:

prop → **declare id** *lista opciones*
lista opciones → *lista opciones opción* | ϵ
opción → *modo* | *escala* | *precisión* | *base*
modo → **real** | **complex**
escala → **fixed** | **floating**
precisión → **single** | **double**
base → **binary** | **decimal**

a) Demuéstrese cómo se puede generalizar esta gramática para admitir n opciones A_i, $1 \leq i \leq n$, cada una de las cuales puede ser a_i o b_i.
b) La gramática anterior admite declaraciones redundantes o contradictorias, como

```
declare zap real fixed real floating
```

Se podría insistir en que la sintaxis del lenguaje prohíbe tales declaraciones. De ese modo queda un número finito de secuencias de componentes léxicos sintácticamente correctos. Obviamente, estas declaraciones legales forman un lenguaje independiente del contexto, en realidad un conjunto regular. Escríbase una gramática para declaraciones con n opciones, donde cada opción aparezca a lo sumo una vez.

**c) Demuéstrese que una gramática para el apartado b) debe tener al menos 2^n símbolos.

d) ¿Qué indica c) sobre la posibilidad de forzar la no redundancia y no contradicción entre opciones en las declaraciones a través de la definición sintáctica de un lenguaje?

4.11 a) Elimínese la recursividad por la izquierda de la gramática del ejercicio 4.1.

b) Constrúyase un analizador sintáctico predictivo para la gramática de a). Muéstrese el comportamiento del analizador con las frases del ejercicio 4.1 b).

4.12 Constrúyase un analizador sintáctico por descenso recursivo con retroceso para la gramática del ejercicio 4.2. ¿Se puede construir un analizador sintáctico predictivo para esta gramática?

4.13 La gramática

$$S \rightarrow aSa \mid aa$$

genera todas las cadenas de longitud par de símbolos a excepto la cadena vacía.

a) Constrúyase un analizador sintáctico por descenso recursivo con retroceso para esta gramática que intente la alternativa aSa antes que aa. Demuéstrese que el procedimiento para S funciona con 2, 4 u 8 a, pero falla con 6 a.

*b) ¿Qué lenguaje reconoce este analizador sintáctico?

4.14 Constrúyase un analizador sintáctico predictivo para la gramática del ejercicio 4.3.

4.15 Constrúyase un analizador sintáctico predictivo a partir de la gramática no ambigua para expresiones regulares del ejercicio 4.4.

***4.16** Demuéstrese que ninguna gramática recursiva por la izquierda puede ser LL(1).

***4.17** Demuéstrese que ninguna gramática LL(1) puede ser ambigua.

4.18 Demuéstrese que una gramática sin producciones ϵ donde cada alternativa comience con un terminal diferente siempre es LL(1).

4.19 Un símbolo gramatical X es *inútil* si no hay ninguna derivación de la forma $S \stackrel{*}{\Rightarrow} wXy \stackrel{*}{\Rightarrow} xyz$. Es decir, X nunca puede aparecer en la derivación de alguna frase.

*a) Escríbase un algoritmo para eliminar de una gramática todas las producciones que contengan símbolos inútiles.

b) Aplíquese este algoritmo a la gramática

$$S \rightarrow 0 \mid A$$
$$A \rightarrow AB$$
$$B \rightarrow 1$$

4.20 Se dice que una gramática está libre de producciones ϵ si no tiene ninguna producción ϵ o si hay exactamente una producción ϵ $S \rightarrow \epsilon$ y después el símbolo inicial S no aparece en el lado derecho de ninguna producción.
a) Escríbase un algoritmo para convertir una gramática dada en una gramática equivalente libre de producciones ϵ. *Sugerencia*: Primero determínense todos los no terminales que pueden generar la cadena vacía.
b) Aplíquese este algoritmo a la gramática del ejercicio 4.2.

4.21 Una producción *simple* es la que tiene un solo no terminal por lado derecho.
a) Escríbase un algoritmo para convertir una gramática en una gramática equivalente sin producciones simples.
b) Aplíquese el algoritmo a la gramática de expresiones (4.10).

4.22 Una gramática *sin ciclos* no tiene derivaciones de la forma $A \overset{+}{\Rightarrow} A$ para ningún no terminal A.
a) Escríbase un algoritmo para convertir una gramática en una gramática equivalente sin ciclos.
b) Aplíquese el algoritmo a la gramática

$$S \rightarrow SS \mid (S) \mid \epsilon .$$

4.23 a) Utilizando la gramática del ejercicio 4.1, constrúyase una derivación por la derecha para $(a, (a, a))$ y muéstrese el mango de cada forma de frase derecha.
b) Muéstrense los pasos de un analizador sintáctico por desplazamiento y reducción correspondientes a la derivación por la derecha de a).
c) Muéstrense los pasos de la construcción ascendente de un árbol de análisis sintáctico durante el análisis por desplazamiento y reducción de b).

4.24 En la figura 4.60 se muestran las relaciones de precedencia de operadores para la gramática del ejercicio 4.1. Utilizando estas relaciones de precedencia, analícense las frases del ejercicio 4.1 b).

	a	(	)	,	$
a			$\cdot>$	$\cdot>$	$\cdot>$
(	$<\cdot$	$<\cdot$	$\doteq$	$<\cdot$	
)			$\cdot>$	$\cdot>$	$\cdot>$
,	$<\cdot$	$<\cdot$	$\cdot>$	$\cdot>$	
$	$<\cdot$	$<\cdot$			

Fig. 4.60. Relaciones de precedencia de operadores para la gramática del ejercicio 4.1.

4.25 Encuéntrense funciones de precedencia de operadores para la tabla de la figura 4.60.

4.26 Existe una forma mecánica de producir relaciones de precedencia de operadores a partir de una gramática de operadores, incluidas las que tengan muchos no terminales distintos. Se define *inicial* (A) para el no terminal A como el conjunto de terminales a tales que a es el terminal situado más a la izquierda en alguna cadena derivada de A, y *final* (A) como el conjunto de terminales que pueden estar situados más a la derecha en una cadena derivada de A. Entonces, para los terminales a y b, se dice que $a \doteq b$ si hay un lado derecho de la forma $\alpha a \beta b \gamma$, donde β es la cadena vacía o un no terminal simple, y α y γ son arbitrarias. Se dice que $a <\cdot b$ si hay un lado derecho de la forma $\alpha a A \beta$, y b está en *inicial*(A); se dice que $a \cdot > b$ si hay un lado derecho de la forma $\alpha A b \beta$, y a está en *final*(A). En ambos casos, α y β son cadenas arbitrarias. También $\$ <\cdot b$, siempre que b esté en *inicial*(S), donde S es el símbolo inicial, y $a \cdot > \$$ siempre que a esté en *final*(S).

a) Para la gramática del ejercicio 4.1, calcúlense *inicial* y *final* para S y T.

b) Compruébese si las relaciones de precedencia de la figura 4.60 son las derivadas de esta gramática.

4.27 Genérense las relaciones de precedencia de operadores para las siguientes gramáticas:

a) La gramática del ejercicio 4.2.

b) La gramática del ejercicio 4.3.

c) La gramática de expresiones (4.10).

4.28 Constrúyase un analizador sintáctico por precedencia de operadores para expresiones regulares.

4.29 Se dice que una gramática es una *gramática de precedencia de operadores* (invertible de manera única) si es una gramática de operadores que no tenga dos lados derechos con el mismo patrón de terminales, y el método del ejercicio 4.26 produce a lo sumo una relación de precedencia entre cualquier par de terminales. ¿Qué gramáticas del ejercicio 4.27 son gramáticas de precedencia de operadores?

4.30 Se dice que una gramática está en la *forma normal de Greibach* (GNF) si está libre de producciones ϵ y cada producción (excepto $S \rightarrow \epsilon$, si es que existe) es de la forma $A \rightarrow a\alpha$, donde a es un terminal, y α, una cadena de no terminales, posiblemente vacía.

**a) Escríbase un algoritmo para convertir una gramática en una gramática equivalente en la forma normal de Greibach.

b) Aplíquese el algoritmo a la gramática de expresiones (4.10).

***4.31** Demuéstrese que toda gramática se puede convertir en una gramática de operadores equivalente. *Sugerencia*: Primero transfórmese la gramática en la forma normal de Greibach.

***4.32** Demuéstrese que toda gramática se puede convertir en una gramática de operadores en la que cada producción tiene una de las formas

$A \rightarrow aBcC \quad A \rightarrow aBb \quad A \rightarrow aB \quad A \rightarrow a$

Si ϵ está en el lenguaje, entonces $S \rightarrow \epsilon$ también es una producción.

4.33 Considérese la gramática ambigua

$S \rightarrow AS \mid b$
$A \rightarrow SA \mid a$

a) Constrúyase la colección de conjuntos de elementos LR(0) para esta gramática.
b) Constrúyase un AFN en el que cada estado sea un elemento LR(0) de a). Demuéstrese que el grafo de transiciones ir a de la colección canónica de elementos LR(0) para esta gramática es el mismo que el del AFD construido a partir de AFN utilizando la construcción de subconjuntos.
c) Constrúyase la tabla de análisis sintáctico utilizando el algoritmo 4.8 SLR.
d) Muéstrense todos los movimientos admitidos por la tabla de c) con la entrada *abab*.
e) Constrúyase la tabla de análisis sintáctico canónico.
f) Constrúyase la tabla de análisis sintáctico utilizando el algoritmo 4.11 LALR.
g) Constrúyase la tabla de análisis sintáctico utilizando el algoritmo 4.13 LALR.

4.34 Constrúyase una tabla de análisis sintáctico SLR para la gramática del ejercicio 4.3.

4.35 Considérese la siguiente gramática:

$E \rightarrow E + T \mid T$
$T \rightarrow T\,F \mid F$
$F \rightarrow F* \mid a \mid b$

a) Constrúyase la tabla de análisis sintáctico SLR para esta gramática.
b) Constrúyase la tabla de análisis sintáctico LALR.

4.36 Comprímanse las tablas de análisis sintáctico construidas en los ejercicios 4.33, 4.34 y 4.35 según el método de la sección 4.7.

4.37 a) Demuéstrese que la siguiente gramática

$S \rightarrow AaAb \mid BbBa$
$A \rightarrow \epsilon$
$B \rightarrow \epsilon$

es LL(1) pero no SLR(1).
**b) Demuéstrese que toda gramática LL(1) es una gramática LR(1).

***4.38** Demuéstrese que ninguna gramática LR(1) puede ser ambigua.

4.39 Demuéstrese que la siguiente gramática

$S \rightarrow Aa \mid bAc \mid dc \mid bda$
$A \rightarrow d$

es LALR(1), pero no SLR(1).

4.40 Demuéstrese que la siguiente gramática

$$S \rightarrow Aa \mid bAc \mid Bc \mid bBa$$
$$A \rightarrow d$$
$$B \rightarrow d$$

es LR(1), pero no LARL(1).

***4.41** Considérese la familia de gramáticas G_n definida por:

$$S \rightarrow A_i b_i \qquad 1 \leq i \leq n$$
$$A_i \rightarrow a_j A_i \mid a_j \qquad 1 \leq i, j \leq n \text{ y } j \neq i$$

a) Demuéstrese que G_n tiene $2n^2 - n$ producciones y $2^n + n^2 + n$ conjuntos de elementos LR(0). ¿Qué indica este resultado acerca de lo grande que puede llegar a ser un analizador sintáctico LR comparado con el tamaño de la gramática?
b) ¿Es G_n SLR(1)?
c) ¿Es G_n LALR(1)?

4.42 Escríbase un algoritmo para calcular para cada no terminal A de una gramática el conjunto de no terminales B tales que $A \stackrel{*}{\Rightarrow} B\alpha$ para alguna cadena de símbolos gramaticales α.

4.43 Escríbase un algoritmo que calcule para cada no terminal A de una gramática el conjunto de terminales a tales que $A \stackrel{*}{\Rightarrow} aw$ para alguna cadena de terminales w, donde el último paso de la derivación no utilice una producción ϵ.

4.44 Constrúyase una tabla de análisis sintáctico SLR para la gramática del ejercicio 4.4. Resuélvanse los conflictos en las acciones del análisis sintáctico de forma que las expresiones regulares se puedan analizar normalmente.

4.45 Constrúyase un analizador sintáctico SLR para la gramática del **else** ambiguo (4.7), considerando *expr* como un terminal. Resuélvase el conflicto de acciones de análisis sintáctico de la forma habitual.

4.46 a) Constrúyase una tabla de análisis sintáctico SLR para la gramática:

$$E \rightarrow E \textbf{ sub } R \mid E \textbf{ sup } E \mid \{ E \} \mid c$$
$$R \rightarrow E \textbf{ sup } E \mid E$$

Resuélvanse los conflictos en las acciones del análisis sintáctico de modo que las expresiones se analicen de la misma forma que si lo hiciera el analizador sintáctico LR de la figura 4.52.

b) ¿Se pueden convertir todos los conflictos de reducción/reducción generados en el proceso de construcción de la tabla de análisis sintáctico LR en conflictos de desplazamiento/reducción transformando la gramática?

***4.47** Constrúyase una gramática LR equivalente para la gramática de tipografía (4.25) que factorice las expresiones de la forma E **sub** E **sup** E como un caso especial.

***4.48** Considérese la siguiente gramática ambigua para n operadores infijos binarios:

$$E \rightarrow E\,\theta_1\,E \mid E\,\theta_2\,E \mid \cdots \mid E\,\theta_n\,E \mid (E) \mid \mathbf{id}$$

Supóngase que todos los operadores son asociativos por la izquierda y que θ_i tiene precedencia sobre θ_j si $i > j$.

a) Constrúyanse los conjuntos de elementos SLR para esta gramática. ¿Cuántos conjuntos de elementos hay, como función de n?

b) Constrúyase la tabla de análisis sintáctico SLR para esta gramática y comprímase utilizando la representación con listas de la sección 4.7. ¿Cuál es la longitud total de todas las listas utilizadas en la representación en función de n?

c) ¿Cuántos pasos necesita para analizar **id** θ_i **id** θ_j **id**?

***4.49** Repítase el ejercicio 4.48 para la gramática no ambigua

$$\begin{aligned} E_1 &\rightarrow E_1\,\theta_1\,E_2 \mid E_2 \\ E_2 &\rightarrow E_2\,\theta_2\,E_3 \mid E_3 \\ &\cdots \\ E_n &\rightarrow E_n\,\theta_n\,E_{n+1} \mid E_{n+1} \\ E_{n+1} &\rightarrow (E_1) \mid \mathbf{id} \end{aligned}$$

¿Qué indican las respuestas a los ejercicios 4.48 y 4.49 sobre la eficiencia relativa de los analizadores sintácticos para gramáticas ambiguas y no ambiguas equivalentes? ¿Qué indican acerca de la eficiencia relativa de construir el analizador sintáctico?

4.50 Escríbase un programa en YACC que tome expresiones aritméticas como entrada y produzca la correspondiente expresión postfija como salida.

4.51 Escríbase un programa "calculadora de escritorio" en YACC que evalúe expresiones booleanas.

4.52 Escríbase un programa en YACC que tome como entrada una expresión regular y produzca un arbol de análisis sintáctico como salida.

4.53 Indíquense los movimientos que ejecutarían los analizadores sintácticos predictivo, de precedencia de operadores y LR de los ejemplos 4.20, 4.32 y 4.50 con las siguientes entradas erróneas:

a) (**id** + (* **id**)

b) * + **id**) + (**id** *

***4.54** Constrúyase un analizador sintáctico por precedencia de operadores corrector de errores y uno LR para la siguiente gramática:

$$\begin{aligned} \mathit{prop} \rightarrow\ & \mathbf{if}\ e\ \mathbf{then}\ \mathit{prop} \\ \mid\ & \mathbf{if}\ e\ \mathbf{then}\ \mathit{prop}\ \mathbf{else}\ \mathit{prop} \\ \mid\ & \mathbf{while}\ e\ \mathbf{do}\ \mathit{prop} \\ \mid\ & \mathbf{begin}\ \mathit{lista}\ \mathbf{end} \\ \mid\ & s \\ \mathit{lista} \rightarrow\ & \mathit{lista}\ ;\ \mathit{prop} \\ \mid\ & \mathit{prop} \end{aligned}$$

*4.55 La gramática del ejercicio 4.54 se puede convertir en LL sustituyendo las producciones de *lista* por

lista → *prop lista'*
lista' → ; *prop* | ϵ

Constrúyase un analizador sintáctico predictivo corrector de errores para la gramática revisada.

4.56 Muéstrese el comportamiento de los analizadores sintácticos de los ejercicios 4.54 y 4.55 con las entradas erróneas
a) **if** *e* **then** *s* ; **if** *e* **then** *s* **end**
b) **while** *e* **do begin** *s* ; **if** *e* **then** *s* ; **end**

4.57 Escríbanse analizadores sintácticos predictivos, por precedencia de operadores y LR con recuperación de errores en modo de pánico para las gramáticas de los ejercicios 4.54 y 4.55, utilizando el símbolo de punto y coma y **end** como componentes léxicos de sincronización. Muéstrese el comportamiento de los analizadores con las entradas erróneas del ejercicio 4.56.

4.58 En la sección 4.6 se propuso un método orientado a grafos para determinar el conjunto de cadenas que se podrían extraer de la pila en un movimiento de reducción de un analizador sintáctico por precedencia de operadores.
*a) Dése un algoritmo para encontrar una expresión regular que represente a todas esas cadenas.
b) Dése un algoritmo para determinar si el conjunto de tales cadenas es finito o infinito, listándolas si es finito.
c) Aplíquense los algoritmos de a) y b) a la gramática del ejercicio 4.54.

**4.59 Se insistió para los analizadores sintácticos correctores de errores de las figuras 4.18, 4.28 y 4.53 en que cualquier corrección de error diera finalmente como resultado la eliminación de por lo menos un símbolo más de entrada o que la pila se acortara si se había alcanzado el final de la entrada. Sin embargo, no todas las correcciones elegidas hacían que se consumiera inmediatamente un símbolo de entrada, ¿Se puede demostrar la imposibilidad de lazos infinitos para los analizadores sintácticos de las figuras 4.18, 4.28 y 4.53? *Sugerencia*: Es útil observar que para el analizador sintáctico por precedencia de operadores los terminales consecutivos dentro de la pila están relacionados por $\leq \cdot$, aunque haya habido errores. Para el analizador sintáctico LR, la pila seguirá conteniendo un prefijo viable, aun en presencia de errores.

**4.60 Dése un algoritmo para detectar entradas inalcanzables en las tablas de análisis predictivo, por precedencia de operadores y LR.

4.61 El analizador sintáctico LR de la figura 4.53 maneja los cuatro casos en que el estado del tope es 4 ó 5 (que ocurren cuando + y * están en el tope de la pila, respectivamente) y la siguiente entrada es + o * exactamente de la misma forma: llamando a la rutina e1, la cual inserta un **id** entre ellos. Se podría imaginar fácilmente un analizador sintáctico LR para expresiones que

engloben el conjunto completo de operadores aritméticos y que se comporten de la misma manera: insértese **id** entre los operadores adyacentes. En algunos lenguajes (como PL/1 o C, pero no FORTRAN o Pascal) sería conveniente considerar, de manera especial, el caso en que / está en el tope de la pila y * es el siguiente símbolo de entrada. ¿Por qué? ¿Qué curso de acción razonable debería seguir el corrector de errores?

4.62 Se dice que una gramática está en la *forma normal de Chomsky* (CNF) si está libre de producciones ϵ y cada producción distinta de ϵ es de la forma $A \rightarrow BC$ o de la $A \rightarrow a$.

*a) Dése un algoritmo para convertir una gramática en una gramática equivalente de la forma normal de Chomsky.

b) Aplíquese el algoritmo a la gramática de expresiones (4.10).

4.63 Dada una gramática G en la forma normal de Chomsky y una cadena de entrada $w = a_1 a_2 \ldots a_n$, escríbase un algoritmo para determinar si w está en $L(G)$. *Sugerencia*: Utilizando programación dinámica complétese una tabla T de $n \times n$ en la que $T[i, j] = \{A \mid A \stackrel{*}{\Rightarrow} a_i a_{i+1} \ldots a_j\}$. La cadena de entrada w está en $L(G)$ si, y sólo si, S está en $T[1, n]$.

***4.64** a) Dada una gramática G en la forma normal de Chomsky, muéstrese cómo añadir producciones para la inserción, eliminación y mutación simples de errores a la gramática de manera que la gramática aumentada genere todas las posibles cadenas de componentes léxicos.

b) Modifíquese el algoritmo de análisis sintáctico del ejercicio 4.63 de manera que, dada cualquier cadena w, encuentre un análisis sintáctico para que w utilice un número mínimo de producciones de error.

4.65 Escríbase un analizador sintáctico en YACC para las expresiones aritméticas que utilice el mecanismo de recuperación de errores del ejemplo 4.50.

NOTAS BIBLIOGRAFICAS

El muy influyente informe de ALGOL 60 (Naur [1963]) utilizó la forma de Backus-Naur (BNF) para definir la sintaxis de un importante lenguaje de programación. Se descubrió la equivalencia entre BNF y las gramáticas independientes del contexto, y la teoría de los lenguajes formales fue objeto de una gran atención en la década de 1960. Hopcroft y Ullman [1979] cubren los aspectos básicos en este campo.

Los métodos de análisis sintáctico se hicieron mucho más sistemáticos tras el desarrollo de las gramáticas independientes del contexto. Se inventaron diversas técnicas generales para analizar cualquier gramática independiente del contexto. Una de las primeras es la técnica de programación dinámica propuesta en el ejercicio 4.63, descubierta independientemente por J. Cocke, Younger [1967] y Kasami [1965]. Como tesis doctoral, Earley [1970] también desarrolló un algoritmo de análisis sintáctico universal para todas las gramáticas independientes del contexto. Aho y Ullman [1972b y 1973a] estudian detalladamente éstos y otros métodos de análisis sintáctico.

Se han empleado muchos métodos distintos de análisis sintáctico en los compiladores. Sheridan [1959] describe el método de análisis sintáctico utilizado en el compilador original de FORTRAN que introdujo paréntesis adicionales a uno y otro lado de los operandos para poder analizar sintácticamente las expresiones. La idea de la precedencia de operadores y el uso de funciones de precedencia es obra de Floyd [1963]. En la década de 1960, se propuso una gran cantidad de estrategias de análisis sintáctico ascendente. Estas incluyen la precedencia simple (Wirth y Weber [1966]), el acotamiento de contexto (Floyd [1964], Graham [1964]), la precedencia de estrategia mixta (McKeeman, Horning y Wortman [1970]) y de precedencia débil (Ichbiah y Morse [1970]).

Los análisis sintácticos por descenso recursivo y predictivo son muy utilizados en la práctica. Dada su flexibilidad, se utilizó el análisis sintáctico por descenso recursivo en muchos de los primeros sistemas generadores de compiladores como META (Schorre [1964]) y TMG (McClure [1965]). En Birman y Ullman [1973], se puede encontrar una solución al ejercicio 4.13, junto con una parte de la teoría de este método de análisis sintáctico. Pratt [1973] propone un método de análisis sintáctico descendente por precedencia de operadores.

Las gramáticas LL fueron estudiadas por Lewis y Stearns [1968] y sus propiedades se desarrollaron en Rosenkrantz y Stearns [1970]. Los analizadores sintácticos predictivos fueron estudiados a fondo por Knuth [1971a]. Lewis, Rosenkrantz y Stearns [1976] describen el uso de los analizadores sintácticos predictivos en los compiladores. Los algoritmos para convertir gramáticas a la forma LL(1) se introducen en Foster [1968], Wood [1969], Stearns [1971] y Soisalon-Soininen y Ukkonen [1979].

Las gramáticas y los analizadores sintácticos LR fueron introducidos por primera vez por Knuth [1965], quien describió la construcción de las tablas de análisis sintáctico LR canónico. El método LR no resultó práctico hasta que Korenjak [1969] mostró que con él se podrían producir analizadores sintácticos de tamaño razonable para gramáticas de lenguajes de programación. Cuando DeRemer [1969, 1971] inventó los métodos SLR y LALR, que son más simples que el de Korenjak, la técnica LR se convirtió en el método elegido para los generadores automáticos de analizadores sintácticos. Hoy en día, los generadores de analizadores LR son habituales en los entornos de construcción de compiladores.

Gran parte de la investigación se dedicó a la construcción de analizadores sintácticos LR. El uso de gramáticas ambiguas en el análisis sintáctico LR se debe a Aho, Johnson y Ullman [1975] y a Earley [1975a]. La eliminación de reducciones por producciones simples ha sido estudiada en Anderson, Eve y Horning [1973], Aho y Ullman [1973b], Demers [1975], Backhouse [1976], Joliat [1976], Pager [1977b], Soisalon-Soininen [1980] y Tokuda [1981].

Las técnicas para calcular conjuntos de símbolos de anticipación LALR(1) han sido propuestas por LaLonde [1971], Anderson, Eve y Horning [1973], Pager [1977a], Kristensen y Madsen [1981], DeRemer y Pennello [1982] y Park, Choe y Chang [1985], quienes también proporcionan algunas comparaciones experimentales.

Aho y Johnson [1974] realizan un estudio general del análisis sintáctico LR y analizan algunos de los algoritmos en que se basa el generador de analizadores sintácticos YACC, incluido el uso de producciones de error para la recuperación de

errores. Aho y Ullman [1972b y 1973a] dan un tratamiento bastante completo del análisis sintáctico LR y de sus fundamentos teóricos.

Se han propuesto muchas técnicas de recuperación de errores para los analizadores sintácticos. Las técnicas para la recuperación de errores son estudiadas por Ciesinger [1979] y por Sippu [1981]. Irons [1963] propuso un enfoque para la recuperación de errores sintácticos basado en la gramática. Las producciones de errores fueron empleadas por Wirth [1968] para manejar errores en un compilador de PL360. Leinius [1970] propuso la estrategia de recuperación en las frases. Aho y Peterson [1972] muestran cómo lograr una recuperación de errores global de costo mínimo utilizando producciones de error junto con algoritmos de análisis sintáctico general para gramáticas independientes del contexto. Mauney y Fischer [1982] aplican dichas ideas a la reparación local de costo mínimo para analizadores sintácticos LL y LR utilizando la técnica de análisis sintáctico de Graham, Harrison y Ruzzo [1980]. Graham y Rhodes [1975] estudian la recuperación de errores en el contexto del análisis sintáctico por precedencia.

Horning [1976] estudia las cualidades que deben tener los buenos mensajes de error. Sippu y Soisalon-Soininen [1983] comparan el uso de la técnica de recuperación de errores en el Helsinki Language Processor (Räihä y otros [1983]) con la técnica de recuperación de "movimiento hacia adelante" de Pennello y DeRemer [1978], con la técnica de recuperación de errores de Graham, Haley y Joy [1979] y con la técnica de recuperación de "contexto global" de Pai y Kieburtz [1980].

La corrección de errores durante el análisis sintáctico es estudiada por Conway y Maxwell [1963], Moulton y Muller [1967]. Conway y Wilcox [1973], Levy [1975], Tai [1978] y Röhrich [1980]. En Aho y Peterson [1972] se da una solución al ejercicio 4.63.

CAPITULO 5

Traducción dirigida por la sintaxis

Este capítulo desarrolla el tema de la sección 2.3, la traducción de lenguajes guiada por gramáticas independientes del contexto. Se asocia información a una construcción del lenguaje de programación proporcionando atributos a los símbolos de la gramática que representan la construcción. Los valores de los atributos se calculan mediante "reglas semánticas" asociadas a las producciones gramaticales.

Hay dos notaciones para asociar reglas semánticas con producciones, las definiciones dirigidas por la sintaxis y los esquemas de traducción. Las definiciones dirigidas por la sintaxis son especificaciones de alto nivel para traducciones. Ocultan muchos detalles de la implantación y no es necesario que el usuario especifique explícitamente el orden en el que tiene lugar la traducción. Los esquemas de traducción indican el orden en que se deben evaluar las reglas semánticas, así que algunos detalles de la implantación quedan visibles. En el capítulo 6 se utilizan ambas notaciones para especificar la comprobación semántica, en particular la determinación de tipos, y en el capítulo 8 para generar código intermedio.

Conceptualmente, tanto con las definiciones dirigidas por la sintaxis como con los esquemas de traducción, se analiza sintácticamente la cadena de componentes léxicos de entrada, se construye el árbol de análisis sintáctico y después se recorre el árbol para evaluar las reglas semánticas en sus nodos (véase Fig. 5.1). La evaluación de las reglas semánticas puede generar código, guardar información en una tabla de símbolos, emitir mensajes de error o realizar otras actividades. La traducción de la cadena de componentes léxicos es el resultado obtenido al evaluar las reglas semánticas.

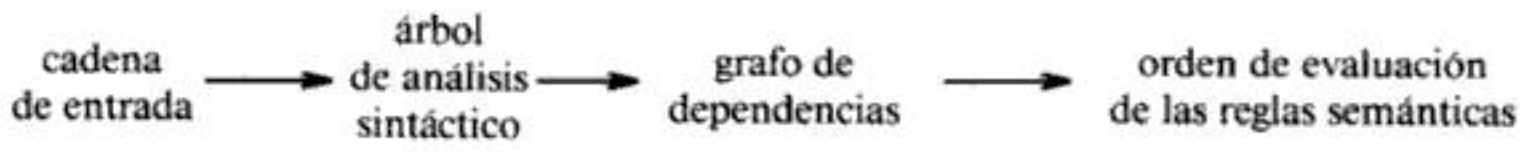


Fig. 5.1. Aspecto conceptual de una traducción dirigida por la sintaxis.

Una implantación no tiene que seguir al pie de la letra el esquema de la figura 5.1. Hay casos especiales de definiciones dirigidas por la sintaxis que se pueden implantar en una sola pasada evaluando las reglas semánticas durante el análisis sin-

táctico, sin construir explícitamente un árbol de análisis sintáctico o un grafo que muestre las dependencias entre los atributos. Como la implantación en una sola pasada es importante para la eficiencia en cuanto al tiempo de compilación, gran parte de este capítulo está dedicada al estudio de dichos casos especiales. Una subclase importante, llamada las definiciones "con atributos por la izquierda", abarca prácticamente todas las traducciones que se puedan realizar sin la construcción explícita de un árbol de análisis sintáctico.

5.1 DEFINICIONES DIRIGIDAS POR LA SINTAXIS

Una definición dirigida por la sintaxis es una generalización de una gramática independiente del contexto en la que cada símbolo gramatical tiene un conjunto de atributos asociado, dividido en dos subconjuntos llamados atributos sintetizados y los heredados de dicho símbolo gramatical. Si se considera un nodo de un símbolo gramatical de un árbol de análisis sintáctico como un registro con campos para guardar información, entonces un atributo corresponde al nombre de un campo.

Un atributo puede representar cualquier cosa: una cadena, un número, un tipo, una posición de memoria, etc. El valor de un atributo en un nodo de un árbol de análisis sintáctico se define mediante una regla semántica asociada a la producción utilizada en dicho nodo. El valor de un atributo sintetizado en un nodo se calcula a partir de los valores de los atributos de los hijos de dicho nodo en el árbol de análisis sintáctico; el valor de un atributo heredado se calcula a partir de los valores de los atributos en los hermanos y el padre de dicho nodo.

Las reglas semánticas establecen las dependencias entre los atributos que serán representadas mediante un grafo. Del grafo de dependencias se obtiene un orden de evaluación de las reglas semánticas. La evaluación de las reglas semánticas define los valores de los atributos en los nodos del árbol de análisis sintáctico para la cadena de entrada. Una regla semántica también puede tener efectos colaterales, por ejemplo, imprimir un valor o actualizar una variable global. Por supuesto, una aplicación no necesita construir explícitamente un árbol de análisis sintáctico o un grafo de dependencias; sólo tiene que producir el mismo resultado para cada cadena de entrada.

Un árbol de análisis sintáctico que muestre los valores de los atributos en cada nodo se denomina un árbol de análisis sintáctico con anotaciones. El proceso de calcular los valores de los atributos en los nodos se denomina *anotar* o *decorar* el árbol de análisis sintáctico.

Forma de una definición dirigida por la sintaxis

En una definición dirigida por la sintaxis, cada producción gramatical $A \rightarrow \alpha$ tiene asociado un conjunto de reglas semánticas de la forma $b := f(c_1, c_2, \ldots, c_k)$, donde f es una función, y, o bien

1. b es un atributo sintetizado de A y $c_1, c_2, \ldots, c_k$ son atributos que pertenecen a los símbolos gramaticales de la producción, o bien

2. b es un atributo heredado de uno de los símbolos gramaticales del lado derecho de la producción, y $c_1, c_2, \ldots, c_k$ son atributos que pertenecen a los símbolos gramaticales de la producción.

En cualquier caso, se dice que el atributo b depende de los atributos $c_1, c_2, \ldots, c_k$. Una *gramática con atributos* es una definición dirigida por la sintaxis en la que las funciones en las reglas semánticas no pueden tener efectos colaterales.

Las funciones de las reglas semánticas a menudo se escribirán como expresiones. Ocasionalmente, el único propósito de una regla semántica en una definición dirigida por la sintaxis es crear un efecto colateral. Dichas reglas semánticas se escriben como llamadas a procedimientos o fragmentos de programa. Se pueden considerar como reglas que definen los valores de atributos sintetizados ficticios del no terminal del lado izquierdo de la producción asociada; no se muestran el atributo ficticio y el signo := de la regla semántica.

Ejemplo 5.1. La definición dirigida por la sintaxis de la figura 5.2 es para un programa para una calculadora de escritorio. Esta definición asocia un atributo sintetizado con un valor entero llamado *val* a cada uno de los no terminales E, T y F. Para cada producción de E, T y F, la regla semántica calcula el valor del atributo *val* para el no terminal del lado izquierdo a partir de los valores de *val* de los no terminales del lado derecho.

PRODUCCIÓN	REGLAS SEMÁNTICAS
$L \rightarrow E$ **n**	*print* (*E.val*)
$E \rightarrow E_1 + T$	$E.val := E_1.val + T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T_1 * F$	$T.val := T_1.val \times F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow (E)$	$F.val := E.val$
$F \rightarrow$ **dígito**	$F.val :=$ **dígito**.*valex*

Fig. 5.2. Definición dirigida por la sintaxis de una calculadora de escritorio sencilla.

El componente léxico **dígito** tiene un atributo sintetizado *valex* cuyo valor viene proporcionado por el analizador léxico. La regla asociada a la producción $L \rightarrow E$ **n** para el no terminal inicial L es sólo un procedimiento que imprime como resultado el valor de la expresión aritmética generada por E; se puede considerar que esta regla define un falso atributo para el no terminal L. En la figura 4.56 se introdujo una especificación en YACC para esta calculadora de escritorio para ilustrar la traducción durante el análisis sintáctico LR. □

En una definición dirigida por la sintaxis, se asume que los terminales sólo tienen atributos sintetizados, ya que la definición no proporciona ninguna regla se-

mántica para los terminales. El analizador léxico es el que proporciona generalmente los valores para los atributos de los terminales como se estudió en la sección 3.1. Además, se asume que el símbolo inicial no tiene ningún atributo heredado, a menos que se indique lo contrario.

Atributos sintetizados

Los atributos sintetizados son muy utilizados en la práctica. Una definición dirigida por la sintaxis que usa atributos sintetizados exclusivamente se denomina *definición con atributos sintetizados*. Siempre se puede anotar un árbol de análisis sintáctico para una definición con atributos sintetizados mediante la evaluación de las reglas semánticas para los atributos en cada nodo de forma ascendente, de las hojas a la raíz. La sección 5.3 describe cómo se puede adaptar un generador de analizadores sintácticos LR para aplicar mecánicamente una definición con atributos sintetizados basada en una gramática LR.

Ejemplo 5.2. La definición con atributos sintetizados del ejemplo 5.1 especifica una calculadora de escritorio que lee una línea de entrada que contiene una expresión aritmética que incluye dígitos, paréntesis, los operadores + y $\star$, seguida de un carácter de nueva línea **n**, e imprime el valor de la expresión. Por ejemplo, dada la expresión 3$\star$5+4 seguida de una nueva línea, el programa imprime el valor 19. La figura 5.3 contiene un árbol de análisis sintáctico con anotaciones para la entrada 3$\star$5+4**n**. El resultado, que se imprime en la raíz del árbol, es el valor de *E.val* en el primer hijo de la raíz.

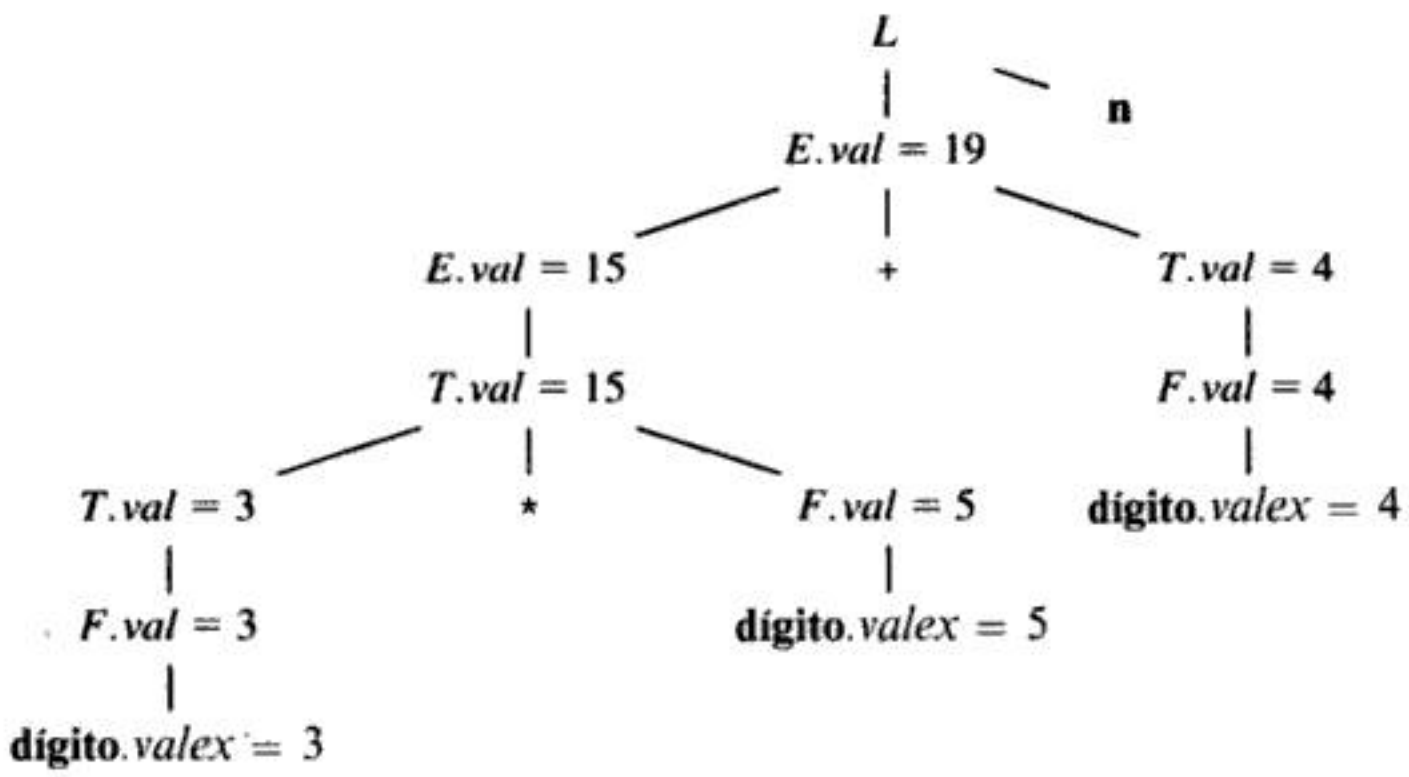


Fig. 5.3. Arbol de análisis sintáctico con anotaciones para 3$\star$5+4**n**.

Para ver cómo se calculan los valores de los atributos, considérese el nodo interior situado en el extremo más bajo de la izquierda, que corresponde al uso de la producción $F \rightarrow$ **dígito**. La regla semántica correspondiente, *F.val* := **dígito**.*valex*,

establece que el atributo *F.val* en el nodo tiene el valor 3 porque el valor de **dígito**.*valex* en el hijo de este nodo es 3. De forma similar, en el padre de este nodo *F*, el atributo *T.val* tiene el valor 3.

A continuación considérese el nodo para la producción $T \rightarrow T * F$. El valor del atributo *T.val* en este nodo está definido por:

PRODUCCIÓN	REGLA SEMÁNTICA
$T \rightarrow T_1 * \mathrm{F}$	$T.val := T_1.val \times F.val$

Cuando se aplica la regla semántica en este nodo, $T_1.val$ tiene el valor 3 del hijo izquierdo y *F.val* el valor 5 del hijo derecho. Por tanto, *T.val* adquiere el valor 15 en este nodo.

La regla asociada con la producción para el no terminal inicial $L \rightarrow E$ **n** imprime el valor de la expresión generada por *E*. □

Atributos heredados

Un atributo heredado es uno cuyo valor en un nodo de un árbol de análisis sintáctico está definido a partir de los atributos en el padre y/o de los hermanos de dicho nodo. Los atributos heredados sirven para expresar la dependencia de una construcción de un lenguaje de programación en el contexto en el que aparece. Por ejemplo, se puede utilizar un atributo heredado para comprobar si un identificador aparece en el lado izquierdo o en el derecho de una asignación para decidir si se necesita la dirección o el valor del identificador. Aunque siempre es posible reescribir una definición dirigida por la sintaxis para que sólo se utilicen atributos sintetizados, a veces es más natural utilizar definiciones dirigidas por la sintaxis con atributos heredados.

En el siguiente ejemplo, un atributo heredado distribuye la información sobre los tipos a los distintos identificadores de una declaración.

Ejemplo 5.3. Una declaración generada por el no terminal *D* en la definición dirigida por la sintaxis en la figura 5.4 consta de la palabra clave **int** o **real**, seguida de una lista de identificadores. El no terminal *T* tiene un atributo sintetizado *tipo*, cuyo

PRODUCCIÓN	REGLAS SEMÁNTICAS
$D \rightarrow T\ L$	$L.her := T.tipo$
$T \rightarrow$ **int**	$T.tipo := integer$
$T \rightarrow$ **real**	$T.tipo := real$
$L \rightarrow L_1$, **id**	$L_1.her := L.her$ *añadetipo* (**id**.*entrada*, *L.her*)
$L \rightarrow$ **id**	*añadetipo* (**id**.*entrada*, *L.her*)

Fig. 5.4. Definición dirigida por la sintaxis con el atributo heredado *L.her*.

valor viene determinado por la palabra clave de la declaración. La regla semántica *L.her* := *T.tipo*, asociada con la producción $D \rightarrow T\ L$, asigna al atributo heredado *L.her* el tipo de la declaración. Entonces las reglas pasan este tipo por el árbol de análisis sintáctico utilizando el atributo heredado *L.her*. Las reglas asociadas con las producciones de *L* llaman al procedimiento *añadetipo* para añadir el tipo de cada identificador a su entrada en la tabla de símbolos (apuntada por el atributo *entrada*).

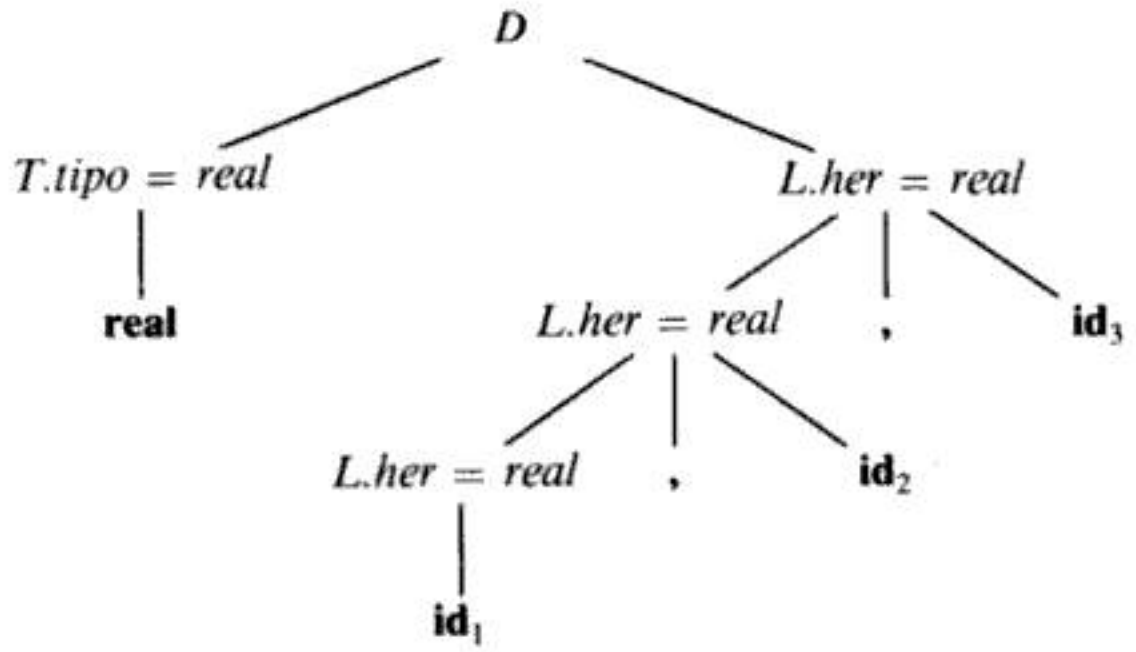


Fig. 5.5. Arbol de análisis sintáctico con el atributo heredado *her* en cada nodo etiquetado con *L*.

En la figura 5.5 se muestra un árbol de análisis sintáctico con anotaciones para la frase **real** $\mathbf{id}_1$, $\mathbf{id}_2$, $\mathbf{id}_3$. El valor de *L.her* en los tres nodos de *L* da el tipo de los identificadores $\mathbf{id}_1$, $\mathbf{id}_2$ e $\mathbf{id}_3$. Estos valores se determinan calculando el valor del atributo *T.tipo* en el hijo izquierdo de la raíz y evaluando después *L.her* de forma descendente en los tres nodos de *L* en el subárbol derecho de la raíz. En cada nodo de *L* también se llama al procedimiento *añadetipo* para insertar en la tabla de símbolos el hecho de que el identificador en el hijo derecho de este nodo tiene tipo real. □

Grafos de dependencias

Si un atributo *b* en un nodo de un árbol de análisis sintáctico depende de un atributo *c*, entonces se debe evaluar la regla semántica para *b* en ese nodo después de la regla semántica que define a *c*. Las interdependencias entre los atributos heredados y sintetizados en los nodos de un árbol de análisis sintáctico se pueden representar mediante un grafo dirigido llamado *grafo de dependencias*.

Antes de construir un grafo de dependencias para un árbol de análisis sintáctico, se escribe cada regla semántica en la forma $b := f(c_1, c_2, \ldots, c_k)$, introduciendo un falso atributo sintetizado *b* para cada regla semántica que conste de una llamada de procedimiento. El grafo tiene un nodo por cada atributo y una arista al nodo de *b* desde el nodo de *c* si el atributo *b* depende del atributo *c*. Más detalladamente, el grafo de dependencias para un determinado árbol de análisis sintáctico se construye de la siguiente manera:

```
for cada nodo n en el árbol de análisis sintáctico do
    for cada atributo a del símbolo gramatical en el nodo n do
        construir un nodo en el grafo de dependencias para a;
for cada nodo n en el árbol de análisis sintáctico do
    for cada regla semántica b := f(c1, c2, . . . , ck)
            asociada con la producción utilizada en n do
        for i := 1 to k do
            construir una arista desde el nodo para ci hasta el nodo para b;
```

Por ejemplo, supóngase que $A.a := f(X.x, Y.y)$ es una regla semántica para la producción $A \rightarrow XY$. Esta regla define un atributo sintetizado $A.a$ que depende de los atributos $X.x$ y $Y.y$. Si se utiliza esta producción en el árbol de análisis sintáctico, entonces habrá tres nodos, $A.a$, $X.x$ y $Y.y$, en el grafo de dependencias con una arista hacia $A.a$ desde $X.x$ puesto que $A.a$ depende de $X.x$, y una arista hacia $A.a$ desde $Y.y$ puesto que $A.a$ también depende de $Y.y$.

Si la producción $A \rightarrow XY$ tiene asociada la regla semántica $X.i := g(A.a, Y.y)$, entonces habrá una arista hacia $X.i$ desde $A.a$ y también una arista hacia $X.i$ desde $Y.y$, puesto que $X.i$ depende tanto de $A.a$ como de $Y.y$.

Ejemplo 5.4. Siempre que se utilice la siguiente producción en un árbol de análisis sintáctico, se añaden al grafo de dependencias las aristas que se muestran en la figura 5.6.

PRODUCCIÓN	REGLA SEMÁNTICA
$E \rightarrow E_1 + E_2$	$E.val := E_1.val + E_2.val$

Los tres nodos del grafo de dependencias marcados con • representan los atributos sintetizados $E.val$, $E_1.val$ y $E_2.val$ en los nodos correspondientes del árbol de análisis

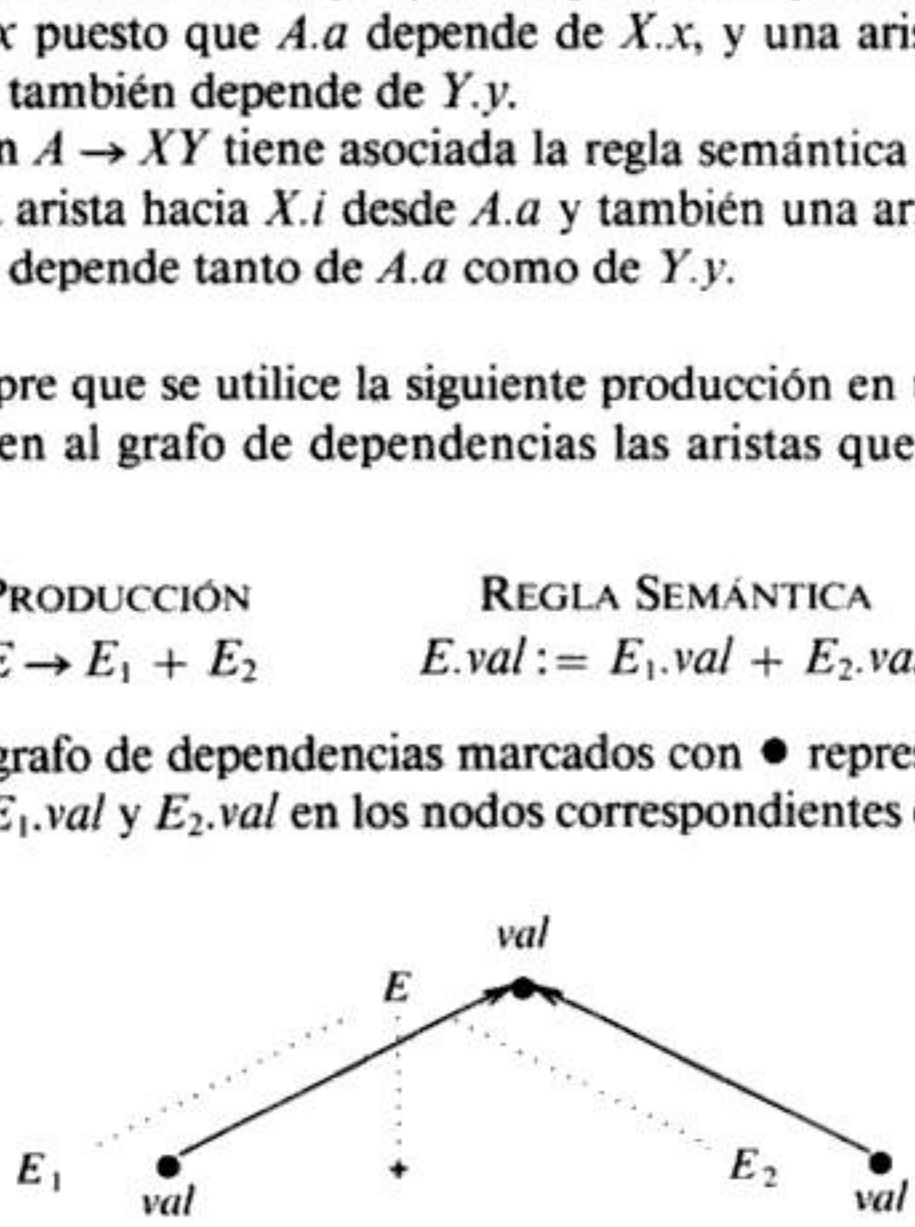


Fig. 5.6. $E.val$ se sintetiza a partir de $E_1.val$ y E_2val.

sintáctico. La arista hacia $E.val$ desde $E_1.val$ muestra que $E.val$ depende de $E_1.val$ y la arista hacia $E.val$ desde $E_2.val$ muestra que $E.val$ también depende de $E_2.val$. Las líneas con puntos representan al árbol de análisis sintáctico y no son parte del grafo de dependencias. □

Ejemplo 5.5. En la figura 5.7 se muestra el grafo de dependencias para el árbol de análisis sintáctico de la figura 5.5. Los nodos en los grafos de dependencias están marcados con números; estos números serán utilizados posteriormente. Hay una

arista hacia el nodo 5 para *L.her* desde el nodo 4 para *T.tipo* porque el atributo heredado *L.her* depende del atributo *T.tipo* según la regla semántica $L.her := T.tipo$ para la producción $D \rightarrow T\ L$. Las dos aristas que apuntan hacia abajo en los nodos 7 y 9 surgen porque depende de *L.her* según la regla semántica $L_1.her := L.her$ para la producción $L \rightarrow L_1$, **id**. Cada una de las reglas semánticas *añadetipo*(**id**.*entrada*, *L.her*) asociada con las producciones de L conduce a la creación de un falso atributo. Los nodos 6, 8 y 10 se constituyen para dichos falsos atributos. □

Orden de evaluación

Un *ordenamiento topológico* de un grafo dirigido acíclico es todo ordenamiento m_1, $m_2, \ldots, m_k$ de los nodos del grafo tal que las aristas vayan desde los nodos que aparecen primero en el ordenamiento a los que aparecen más tarde; es decir, si $m_i \rightarrow m_j$ es una arista desde m_i a m_j, entonces m_i aparece antes que m_j en el ordenamiento.

Todo ordenamiento topológico de un grafo de dependencias da un orden válido en el que se pueden evaluar las reglas semánticas asociadas con los nodos de un árbol de análisis sintáctico. Es decir, en el ordenamiento topológico, los atributos dependientes $c_1, c_2, \ldots, c_k$ en una regla semántica $b := f(c_1, c_2, \ldots, c_k)$ están disponibles en un nodo antes de que se evalúe f.

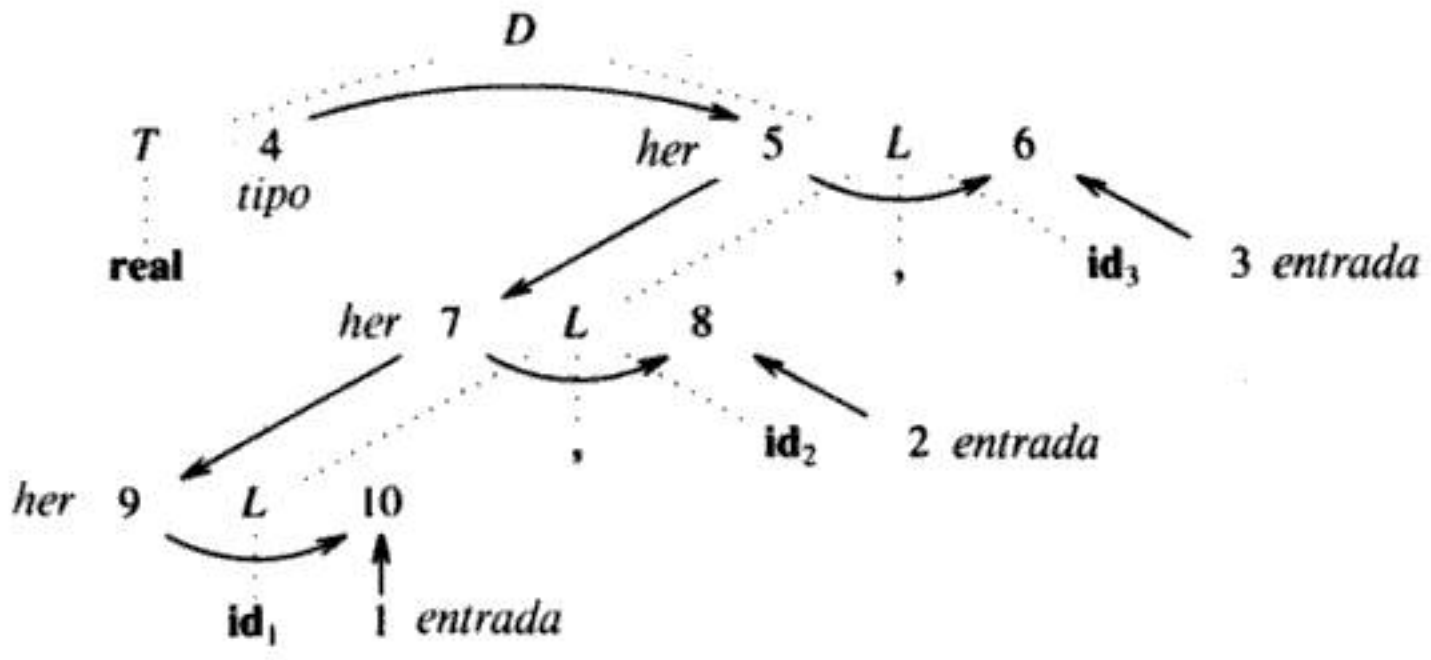


Fig. 5.7. Grafo de dependencias para el árbol de análisis sintáctico de la figura 5.5.

La traducción especificada por una definición dirigida por la sintaxis se puede precisar como sigue. Se utiliza la gramática subyacente para construir un árbol de análisis sintáctico para la entrada. El grafo de dependencias se construye como se indica más arriba. A partir de un ordenamiento topológico del grafo de dependencias, se obtiene un orden de evaluación para las reglas semánticas. La evluación de las reglas semánticas en este orden produce la traducción de la cadena de entrada.

Ejemplo 5.6. Cada una de las aristas en el grafo de dependencias de la figura 5.7 va desde un nodo con un número menor hacia un nodo con un número mayor. Por tanto, un ordenamiento topológico del grafo de dependencias se obtiene escribiendo los nodos en el orden de sus números. A partir de este orden topológico, se obtiene

el siguiente programa. Se escribe a_n para el atributo asociado con el nodo numerado con n en el grafo de dependencias.

$a_4 := real$;
$a_5 := a_4$;
añadetipo(**id**$_3$.*entrada*, a_5);
$a_7 := a_5$;
añadetipo(**id**$_2$.*entrada*,a_7);
$a_9 := a_7$;
añadetipo(**id**$_1$.*entrada*, a_9);

La evaluación de estas reglas semánticas almacena el tipo *real* en la entrada de la tabla de símbolos para cada identificador. □

Se han propuesto varios métodos para evaluar las reglas semánticas:

1. *Métodos con árbol de análisis sintáctico.* En el momento de la compilación, estos métodos obtienen un orden de evaluación a partir de un ordenamiento topológico del grado de dependencias construido según el árbol de análisis sintáctico para cada entrada. Estos métodos no conseguirán encontrar un orden de evaluación sólo si el grafo de dependencias para el árbol de análisis sintáctico determinado que se considera tiene un ciclo.

2. *Métodos basados en reglas.* En el momento de la construcción del compilador, las reglas semánticas asociadas con las producciones se analizan a mano o con una herramienta especializada. Para cada producción el orden en que se evalúan los atributos asociados con dicha producción queda predeterminado en el momento de la construcción del compilador.

3. *Métodos "sin recuerdo".* Se escoge un orden de evaluación sin considerar las reglas semánticas. Por ejemplo, si la traducción tiene lugar durante el análisis sintáctico, entonces el orden de evaluación se ve forzado por el método de análisis, independientemente de las reglas semánticas. Un orden de evaluación sin recuerdo limita la clase de definiciones dirigidas por la sintaxis que pueden implantarse.

Los métodos sin recuerdo y los basados en reglas no necesitan construir de manera explícita el grafo de dependencias en el momento de la compilación, así que pueden ser más eficaces en el uso que hacen del tiempo y del espacio del compilador.

Se dice que una definición dirigida por la sintaxis es *circular* si el grafo de dependencias para un árbol de análisis sintáctico generado por su gramática tiene un ciclo. La sección 5.10 estudia cómo comprobar si una definición dirigida por la sintaxis es circular.

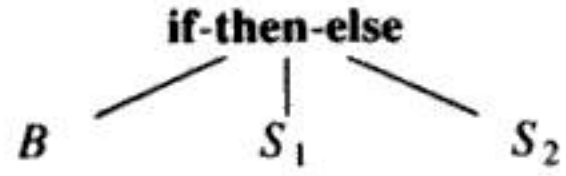
5.2 CONSTRUCCION DE ARBOLES SINTACTICOS

En esta sección se muestra cómo se pueden utilizar las definiciones dirigidas por la sintaxis para especificar la construcción de árboles sintácticos y otras representaciones gráficas de construcciones de lenguajes.

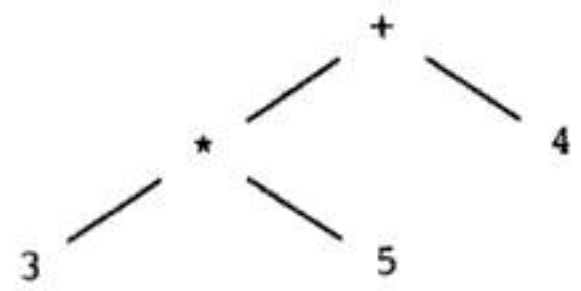
El uso de árboles sintácticos como representación intermedia permite que la traducción se separe del análisis sintáctico. Las rutinas de traducción invocadas durante el análisis sintáctico deben activarse con dos clases de limitaciones. La primera, una gramática que resulte adecuada para el análisis sintáctico puede no reflejar la estructura jerárquica natural de las construcciones del lenguaje. Por ejemplo, una gramática para FORTRAN puede considerar que una subrutina consta simplemente de una lista de proposiciones. Sin embargo, el análisis de la subrutina puede simplificarse utilizando una representación del árbol que refleje el anidamiento de los lazos DO. La segunda, el método de análisis sintáctico restringe el orden en que se consideran los nodos de un árbol de análisis sintáctico. Este orden puede no coincidir con el orden en que se va disponiendo de la información sobre una construcción. Por esta razón, los compiladores para C generalmente construyen árboles sintácticos para las declaraciones.

Arboles sintácticos

Un árbol sintáctico (abstracto) es una forma condensada de un árbol de análisis sintáctico, útil para representar construcciones de lenguajes. La producción $S \rightarrow$ **if** B **then** S_1 **else** S_2 puede aparecer en un árbol sintáctico como



En un árbol sintáctico, los operadores y las palabras clave no aparecen como hojas, sino que más bien están asociadas con el nodo interior que sería el padre de dichas hojas en el árbol de análisis sintáctico. Otra simplificación hallada en los árboles sintácticos es que las cadenas de las producciones simples pueden estar rotas; el árbol de análisis sintáctico de la figura 5.3 se convierte en el árbol sintáctico



La traducción dirigida por la sintaxis se puede basar en árboles sintácticos así como en árboles de análisis sintáctico. El enfoque es el mismo en cada caso; se asocian atributos a los nodos como en un árbol de análisis sintáctico.

Construcción de árboles sintácticos para expresiones

La construcción de un árbol sintáctico para una expresión es similar a la traducción de la expresión a una forma postfija. Se construyen subárboles para las subexpresiones creando un nodo para cada operador y cada operando. Los hijos de un nodo de

un operador son las raíces de los nodos que representan las subexpresiones que constituyen los operandos de dicho operador.

Se puede implantar cada nodo en un árbol sintáctico como un registro con varios campos. En el nodo para un operador, un campo identifica el operador y el resto de los campos contiene apuntadores a los nodos de los operandos. El operador a menudo se denomina la *etiqueta* del nodo. Cuando se usan para traducir los nodos de un árbol sintáctico pueden tener campos adicionales para guardar los valores (o apuntadores a los valores) de los atributos asociados al nodo. En esta sección, se utilizan las siguientes funciones para crear los nodos de los árboles sintácticos para expresiones con operadores binarios. Cada función devuelve un apuntador a un nodo recién creado.

1. *haznodo*(*op*, *izquierda*, *derecha*) crea un nodo para un operador con etiqueta *op* y dos campos que contienen apuntadores a *izquierda* y *derecha*.
2. *hazhoja*(**id**, *entrada*) crea un nodo para un identificador con etiqueta **id** y un campo que contiene *entrada*, que es un apuntador a la entrada de la tabla de símbolos para el identificador.
3. *hazhoja*(**núm**, *val*) crea un nodo para un número con etiqueta **núm** y un campo que contiene *val*, el valor del número.

Ejemplo 5.7. La siguiente secuencia de llamadas a funciones crea el árbol sintáctico para la expresión a−4+c de la figura 5.8. En esta secuencia, $p_1, p_2, \ldots, p_5$ son apuntadores a nodos y *entradaa* y *entradac* son apuntadores a las entradas de la tabla de símbolos para los identificadores a y c, respectivamente.

(1) $p_1 :=$ *hazhoja*(**id**, *entradaa*);
(2) $p_2 :=$ *hazhoja*(**núm**, 4);
(3) $p_3 :=$ *haznodo*('−', p_1, p_2);
(4) $p_4 :=$ *hazhoja*(**id**, *entradac*);
(5) $p_5 :=$ *haznodo*('+', p_3, p_4);

El árbol se construye de abajo a arriba. Las llamadas de funciones *hazhoja*(**id**, *entradaa*) y *hazhoja*(**núm**, 4) construyen las hojas para a y 4; se guardan los apuntadores a estos nodos utilizando p_1 y p_2. La llamada a *haznodo*('−', p_1, p_2) construye después el nodo interior con las hojas para a y 4 como hijos. Después de dos pasos más p_5 queda apuntando a la raíz. □

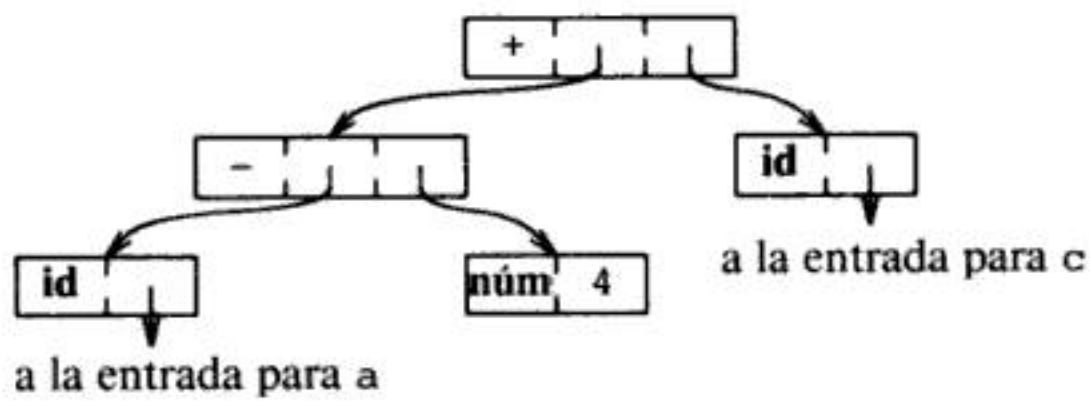


Fig. 5.8. Arbol sintáctico para a−4+c.

Una definición dirigida por la sintaxis para construir árboles sintácticos

La figura 5.9 contiene una definición con atributos sintetizados para construir un árbol sintáctico para una expresión que contiene los operadores + y −. Utiliza las producciones subyacentes de la gramática para organizar las llamadas a las funciones *haznodo* y *hazhoja* para construir el árbol. El atributo sintetizado *apn* para E y T conserva los apuntadores devueltos por las llamadas a las funciones.

PRODUCCIÓN	REGLAS SEMÁNTICAS
$E \rightarrow E_1 + T$	$E.apn := haznodo('+', E_1.apn, T.apn)$
$E \rightarrow E_1 - T$	$E.apn := haznodo('-', E_1.apn, T.apn)$
$E \rightarrow T$	$E.apn := T.apn$
$T \rightarrow (E)$	$T.apn := E.apn$
$T \rightarrow$ **id**	$T.apn :=$ *hazhoja* (**id**, **id**.*entrada*)
$T \rightarrow$ **núm**	$T.apn :=$ *hazhoja* (**núm**, **núm**.*val*)

Fig. 5.9. Definición dirigida por la sintaxis para construir un árbol sintáctico de una expresión.

Ejemplo 5.8. En la figura 5.10 se muestra un árbol de análisis sintáctico con anotaciones que representa la construcción de un árbol sintáctico para la expresión a−4+c. El árbol de análisis sintáctico aparece con puntos. Los nodos del árbol de análisis sintáctico etiquetados con los no terminales E y T usan el atributo sintetizado *apn* para guardar un apuntador al nodo del árbol sintáctico para la expresión representada por el no terminal.

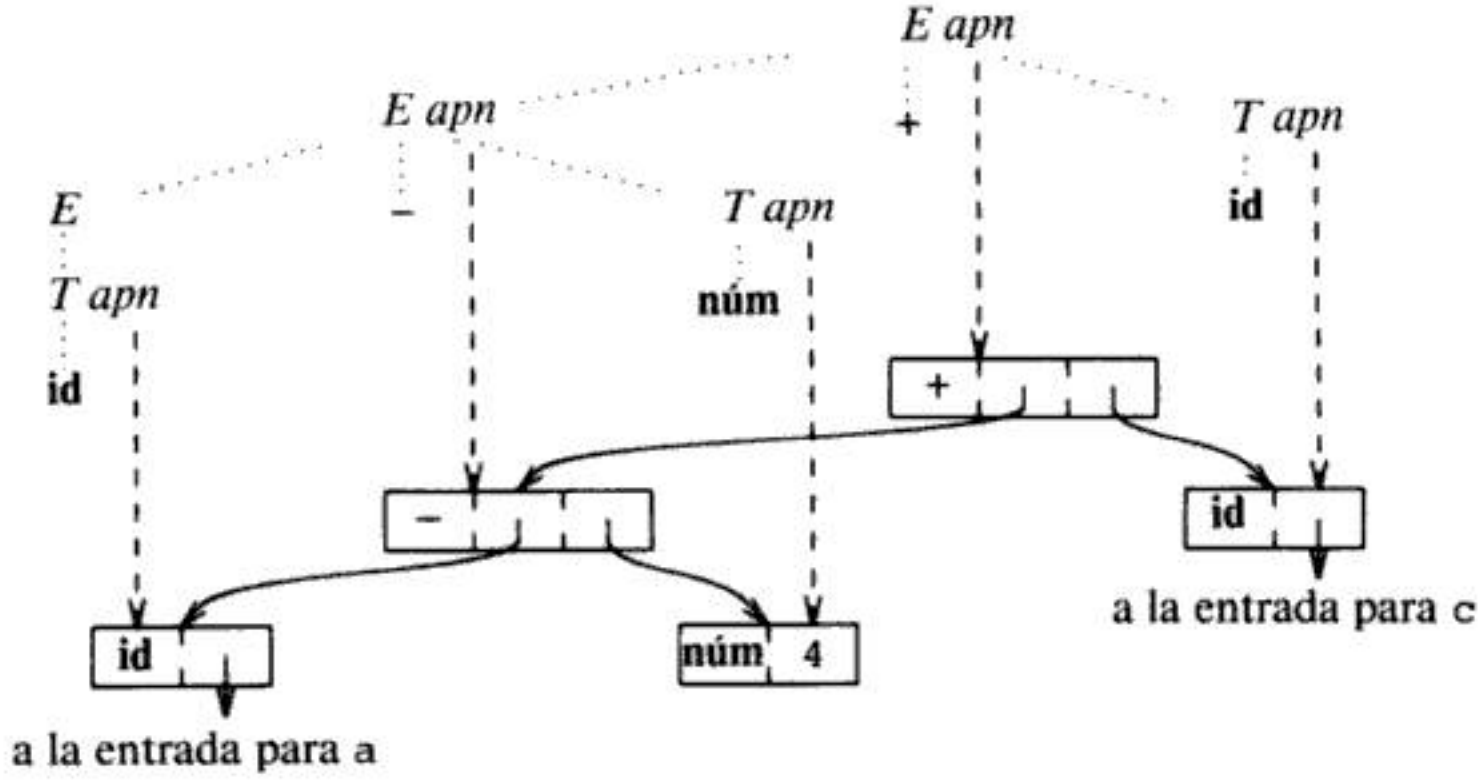


Fig. 5.10. Construcción de un árbol sintáctico para a−4+c.

Las reglas semánticas asociadas con las producciones $T \rightarrow$ **id** y $T \rightarrow$ **núm** definen el atributo *T.apn* como un apuntador a una hoja nueva para un identificador y un número, respectivamente. Los atributos **id**.*entrada* y **núm**.*val* son los valores léxicos que se suponen haber sido devueltos por el analizador léxico con los componentes léxicos **id** y **núm**.

En la figura 5.10, cuando una expresión E es un solo término, correspondiente a un uso de la producción $E \rightarrow T$, el atributo *E.apn* obtiene el valor de *T.apn*. Cuando se invoca la regla semántica $E.apn := haznodo('-', E_1.apn, T.apn)$ asociada con la producción $E \rightarrow E_1 - T$, reglas anteriores han asignado a $E_1.apn$ y *T.apn* la función de apuntadores a las hojas para a y 4, respectivamente.

Al interpretar la figura 5.10, es importante observar que el árbol inferior, formado a partir de registros, es un árbol sintáctico "real" que constituye la salida, mientras que el árbol superior con puntos es el árbol de análisis sintáctico que puede existir sólo en un sentido figurativo. En la siguiente sección se muestra cómo puede implantarse una definición con atributos sintetizados utilizando simplemente la pila de un analizador sintáctico ascendente para guardar los valores de los atributos. De hecho, con esta implantación, las funciones para la construcción de nodos se invocan en el mismo orden que en el ejemplo 5.7. □

Grafos dirigidos acíclicos para expresiones

Un grafo dirigido acíclico (que a partir de ahora se llamará *GDA*) para una expresión identifica sus subexpresiones comunes. Como un árbol sintáctico, un GDA tiene un nodo para toda subexpresión de la expresión; un nodo interior representa un operador y sus hijos representan sus operandos. La diferencia es que un nodo en un GDA que representa a una subexpresión común tiene más de un "padre"; en un árbol sintáctico, la subexpresión común se representaría como un subárbol duplicado.

La figura 5.11 contiene un GDA para la expresión

```
a + a * ( b - c ) + ( b - c ) * d
```

La hoja para a tiene dos padres porque a es común a las dos subexpresiones a y a * (b - c). Asimismo, ambas ocurrencias de la subexpresión común b - c se representan con el mismo nodo, que también tiene dos padres.

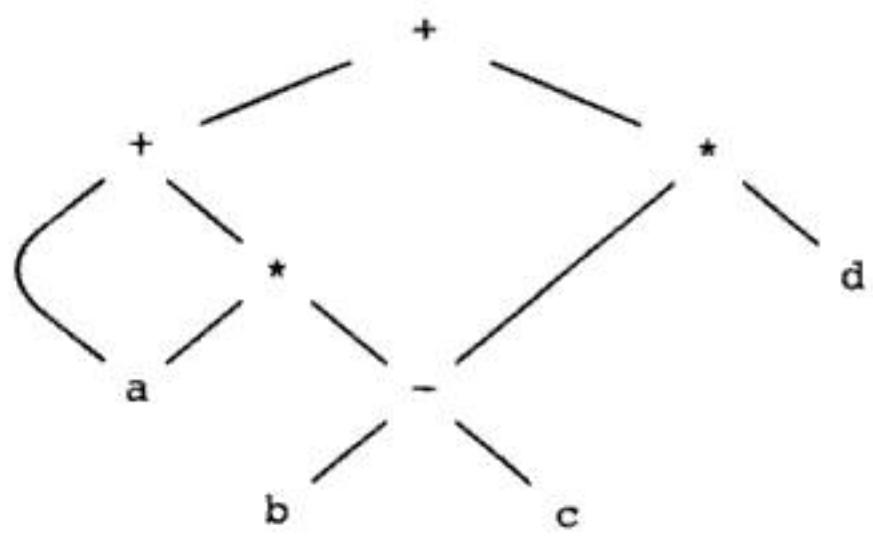


Fig. 5.11. GDA para la expresión a+a*(b-c)+(b-c)*d.

La definición dirigida por la sintaxis de la figura 5.9 construirá un GDA en lugar de un árbol sintáctico si se modifican las operaciones para construir nodos. Se obtiene un GDA si la función que construye un nodo comprueba primero si ya existe un nodo idéntico. Por ejemplo, antes de construir un nuevo nodo con etiqueta *op* y campos con apuntadores a *izquierda* y *derecha*, *haznodo*(*op*, *izquierda*, *derecha*), puede comprobar si dicho nodo ya ha sido construido. Si así es, *haznodo*(*op*, *izquierda*, *derecha*) puede devolver un apuntador al nodo previamente construido. La función para construir hojas *hazhoja* se puede comportar de manera similar.

Ejemplo 5.9. La secuencia de instrucciones de la figura 5.12 construye el GDA de la figura 5.11, a condición de que *haznodo* y *hazhoja* creen nuevos nodos sólo cuando sea necesario, devolviendo siempre que sea posible los apuntadores a los nodos existentes con la etiqueta e hijos adecuados. En la figura 5.12, *a*, *b*, *c*, y *d* apuntan a las entradas de la tabla de símbolos para los identificadores `a`, `b`, `c` y `d`.

(1) $p_1 :=$ *hazhoja*(**id**, *a*);
(2) $p_2 :=$ *hazhoja*(**id**, *a*);
(3) $p_3 :=$ *hazhoja*(**id**, *b*);
(4) $p_4 :=$ *hazhoja*(**id**, *c*);
(5) $p_5 :=$ *haznodo*('−', p_3, p_4);
(6) $p_6 :=$ *haznodo*('*', p_2, p_5);
(7) $p_7 :=$ *haznodo*('+', p_1, p_6);
(8) $p_8 :=$ *hazhoja*(**id**, *b*);
(9) $p_9 :=$ *hazhoja*(**id**, *c*);
(10) $p_{10} :=$ *haznodo*('−', p_8, p_9);
(11) $p_{11} :=$ *hazhoja*(**id**, *d*);
(12) $p_{12} :=$ *haznodo*('*', p_{10}, p_{11});
(13) $p_{13} :=$ *haznodo*('+', p_7, p_{12});

Fig. 5.12. Instrucciones para construir el grafo dirigido acíclico de la figura 5.11.

Cuando la llamada *hazhoja*(**id**, *a*) se repite en la línea 2, se devuelve el nodo construido por la llamada previa *hazhoja*(**id**, *a*), de modo que $p_1 = p_2$. Asimismo, los nodos devueltos en las líneas 8 y 9 son los mismos que los devueltos en las líneas 3 y 4, respectivamente. Por tanto, el nodo devuelto en la línea 10 debe ser el mismo construido por la llamada de *haznodo* en la línea 5. □

En muchas aplicaciones, los nodos se implantan como registros almacenados en una matriz, como en la figura 5.13. En la figura, cada registro tiene un campo de etiqueta que determina la naturaleza del nodo. Se puede hacer referencia a un nodo por su índice o posición en la matriz. El índice entero de un nodo a menudo se de-

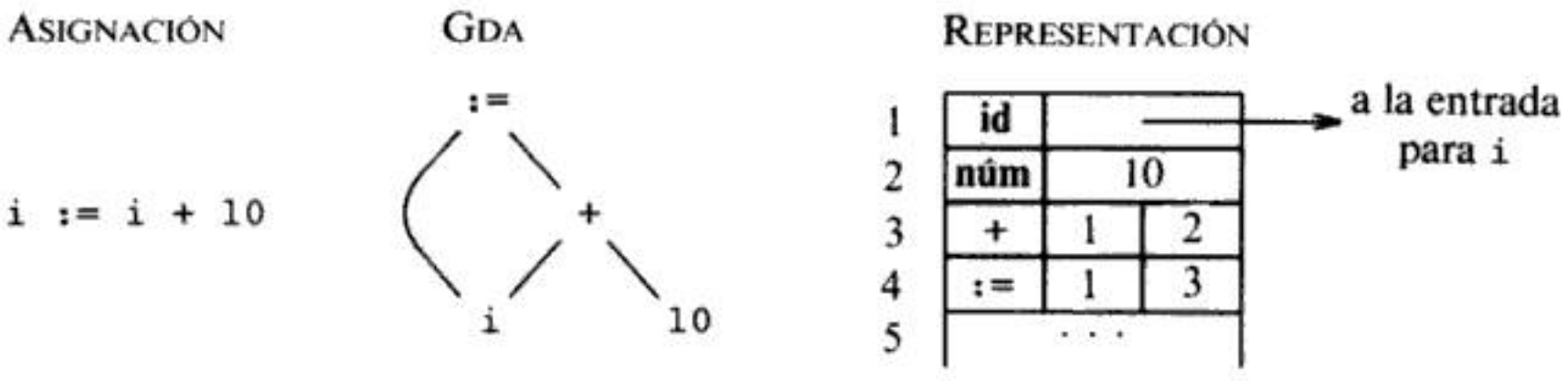


Fig. 5.13. Nodos de un GDA para `i := i + 10` asignados a partir de una matriz.

nomina un *número de valor* para razones históricas. Por ejemplo, utilizando números de valor, se puede decir que el nodo 3 tiene etiqueta +, su hijo izquierdo es el nodo 1 y su hijo derecho es el nodo 2. Se puede utilizar el siguiente algoritmo para crear nodos para una representación con GDA de una expresión.

Algoritmo 5.1. Método del número de valor para construir un nodo en un GDA.

Supóngase que los nodos están almacenados en una matriz como en la figura 5.13, y que se hace referencia a cada nodo por su número de valor. Sea la *signatura* de un nodo de un operador un triple $<op, i, d>$ que consta de su etiqueta *op*, el hijo izquierdo *i* y el hijo derecho *d*.

Entrada. La etiqueta *op*, el nodo *i* y el nodo *d*.

Salida. Un nodo con signatura $<op, i, d>$.

Método. Búsquese la matriz para un nodo *m* con etiqueta *op*, hijo izquierdo *i* e hijo derecho *d*. Si existe tal nodo, devuélvase *m*; de lo contrario, créese un nuevo nodo *n* con etiqueta *op*, hijo izquierdo *i* e hijo derecho *d* y devuélvase *n*.

Una manera evidente de determinar si el nodo *m* ya está en la matriz es guardar todos los nodos creados con anterioridad en una lista y comprobar cada nodo de la lista para ver si tiene la signatura deseada. La búsqueda de *m* puede ser más eficiente si se utilizan *k* listas, llamadas cubetas, y una función de dispersión *h* para determinar qué cubeta se debe buscar[1].

La función de dispersión *h* calcula el número de una cubeta a partir del valor de *op*, *i* y *d*. Siempre devolverá el mismo número de cubeta, dados los mismos argumentos. Si *m* no está en la cubeta *h*(*op*, *i*, *d*), entonces se crea un nuevo nodo *n* y se añade a la cubeta, de modo que las próximas búsquedas lo encontrarán allí. Varias signaturas pueden coincidir por la función de dispersión en el mismo número de cubeta, pero en la práctica se supone que cada cubeta contiene un número pequeño de nodos.

Como se muestra en la figura 5.14, cada cubeta se puede implantar como una lista enlazada. Cada celda en una lista enlazada representa un nodo. Los encabezamientos de las cubetas, que constan de apuntadores a la primera celda de una lista, se almacenan en una matriz. El número de cubeta devuelto por *h*(*op*, *i*, *d*) es un índice dentro de dicha matriz de encabezamientos de cubetas.

Se puede adaptar este algoritmo para la aplicación a nodos que no estén ubicados secuencialmente a partir de una matriz. En muchos compiladores, los nodos se ubican conforme se van necesitando para evitar preasignar una matriz que quizá contenga demasiados nodos la mayor parte del tiempo y no los suficientes algunas veces. En este caso, no se puede suponer que los nodos estén en almacenamiento secuencial, así que deben utilizarse apuntadores para referirse a ellos. Si se puede

---

[1] Basta cualquier estructura de datos que implante diccionarios en el sentido de Aho, Hopcroft y Ullman [1983]. La propiedad importante de la estructura es que, dada una clave, por ejemplo, una etiqueta *op* y dos nodos *i* y *d*, se puede obtener rápidamente un nodo *m* con signatura $<op, i, d>$, o determinar que no existe ninguno.

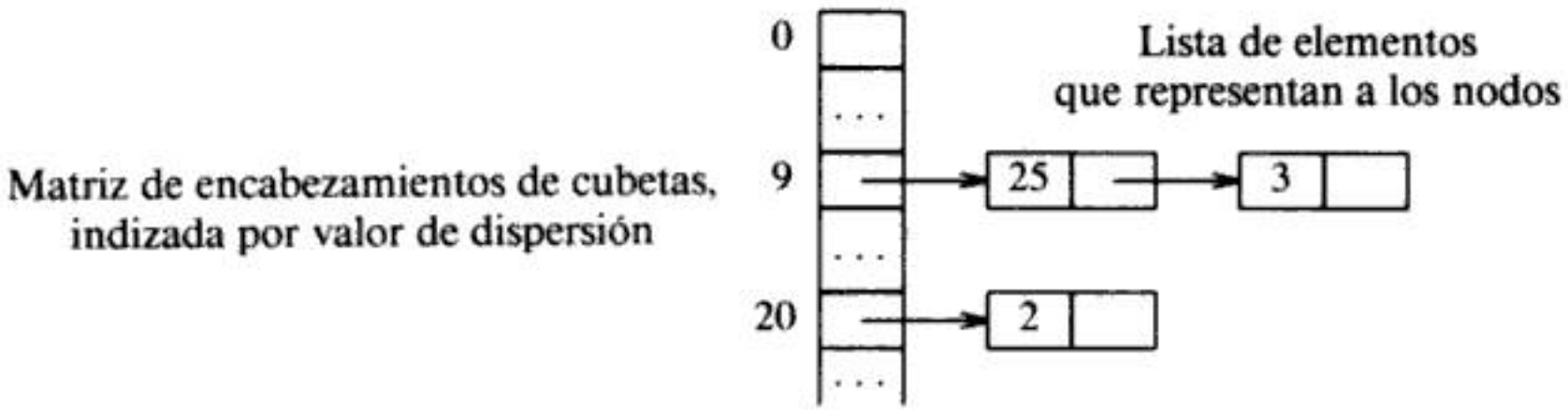


Fig. 5.14. Estructura de datos para la búsqueda de cubetas.

conseguir que la función de dispersión calcule el número de cubeta a partir de una etiqueta y de apuntadores a los hijos, entonces se pueden utilizar apuntadores a los nodos en lugar de números de valor. De lo contrario, se pueden numerar los nodos de cualquier manera y utilizar este número como el número de valor del nodo. □

También se pueden utilizar los GDA para representar conjuntos de expresiones, ya que un GDA puede tener más de una raíz. En los capítulos 9 y 10, los cálculos realizados por una secuecia de proposiciones de asignación se representarán como un GDA.

5.3 EVALUACION ASCENDENTE DE DEFINICIONES CON ATRIBUTOS SINTETIZADOS

Ahora que se ha aprendido a utilizar las definiciones dirigidas por sintaxis para especificar traducciones, se puede comenzar a estudiar la implantación de traductores para ellas. Puede ser difícil construir un traductor para una definición dirigida por la sintaxis arbitraria. Sin embargo, existen amplias clases de útiles definiciones dirigidas por la sintaxis para las cuales es fácil construir traductores. En esta sección se examina una de dichas clases: las definiciones con atributos sintetizados, es decir, las definiciones dirigidas por la sintaxis que sólo contienen atributos sintetizados. Las siguientes secciones consideran la implantación de definiciones que tengan asimismo atributos heredados.

Los atributos sintetizados se pueden evaluar con un analizador sintáctico ascendente conforme la entrada es analizada. El analizador sintáctico puede conservar en su pila los valores de los atributos sintetizados asociados con los símbolos gramaticales. Siempre que se haga una reducción se calculan los valores de los nuevos atributos sintetizados a partir de los atributos que aparecen en la pila para los símbolos gramaticales del lado derecho de la producción con la que se reduce. Esta sección muestra cómo se puede ampliar la pila del analizador sintáctico para guardar los valores de estos atributos sintetizados. En la sección 5.6 se verá que esta implantación también admite algunos atributos heredados.

En la definición dirigida por la sintaxis de la figura 5.9 sólo aparecen atributos sintetizados para construir el árbol sintáctico de una expresión. Por tanto, el enfoque de esta sección se puede aplicar en la construcción de árboles sintácticos du-

rante el análisis sintáctico ascendente. Como se verá en la sección 5.5, la traducción de expresiones durante el análisis sintáctico descendente utiliza a menudo atributos heredados. Por eso se aplazará la traducción durante el análisis sintáctico descendente hasta que en la próxima sección se hayan examinado las dependencias de "izquierda a derecha".

Atributos sintetizados en la pila del analizador sintáctico

A menudo se puede implantar un traductor para una definición con atributos sintetizados con la ayuda de un generador de analizadores sintácticos LR como el estudiado en la sección 4.9. A partir de una definición con atributos sintetizados, el generador de analizadores sintácticos puede construir un traductor que evalúe los atributos conforme analiza la entrada.

Un analizador sintáctico ascendente utiliza una pila para guardar información acerca de los subárboles que ya han sido analizados. Se pueden utilizar campos adicionales en la pila del analizador para guardar los valores de los atributos sintetizados. En la figura 5.15 se muestra un ejemplo de la pila de un analizador sintáctico con espacio para un valor de atributo. Supóngase que, como en la figura, la pila se implanta mediante un par de matrices *estado* y *val*. Cada entrada de *estado* es un apuntador (o índice) a una tabla de análisis sintáctico LR(1). (Obsérvese que el símbolo gramatical está implícito en el estado y no necesita almacenarse en la pila.) Sin embargo, es conveniente referirse al estado mediante el único símbolo gramatical que cubre cuando se sitúa en la pila de análisis sintáctico como se describió en la sección 4.7. Si el i-ésimo símbolo de *estado* es A, entonces $val[i]$ contendrá el valor del atributo asociado con el nodo del árbol de análisis sintáctico correspondiente a esta A.

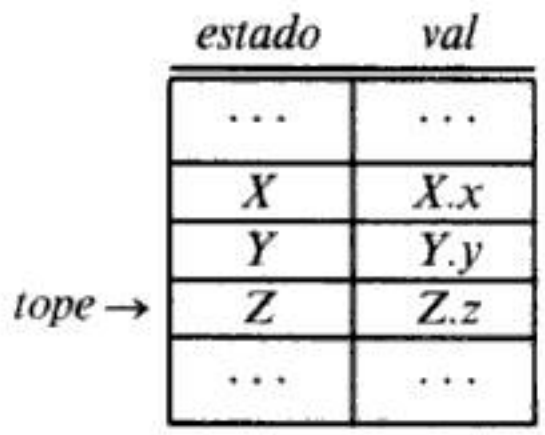


Fig. 5.15. Pila del analizador sintáctico con un campo para los atributos sintetizados.

El tope en curso de la pila se indica con el apuntador *tope*. Se supone que los atributos sintetizados se evalúan justo antes de cada reducción. Supóngase que la regla semántica $A.a := f(X.x, Y.y, Z.z)$ se asocia con la producción $A \rightarrow XYZ$. Antes de que XYZ se reduzca a A, el valor del atributo $Z.z$ está en $val[tope]$, el de $Y.y$ está en $val[tope - 1]$, y el de $X.x$ en $val[tope - 2]$. Si un símbolo no tiene ningún atributo, entonces la entrada correspondiente en el arreglo *val* está sin definir. Después de la reducción, *tope* se reduce en 2, el estado que cubre A se sitúa en $estado[tope]$ (es decir, donde estaba X), y el valor del atributo sintetizado $A.a$ se sitúa en $val\,[tope]$.

Ejemplo 5.10. Considérese de nuevo la definición dirigida por la sintaxis de la calculadora de escritorio de la figura 5.2. Los atributos sintetizados en el árbol de análisis sintáctico con anotaciones de la figura 5.3 se pueden evaluar mediante un analizador sintáctico LR durante un análisis ascendente de la línea de entrada `3*5+4n`. Como antes, se supone que el analizador léxico proporciona el valor del atributo **dígito**.*valex*, que es el valor numérico de cada componente léxico que representa a un dígito. Cuando el analizador sintáctico desplaza un dígito dentro de la pila, el componente léxico **dígito** se coloca en *estado*[*tope*] y su valor de atributo se coloca en *val*[*tope*].

Se pueden utilizar las técnicas de la sección 4.7 para construir un analizador sintáctico LR para la gramática en cuestión. Para evaluar los atributos, se modifica el analizador sintáctico para que ejecute los fragmentos de código de la figura 5.16 justo antes de hacer la reducción adecuada. Obsérvese que se puede asociar la evaluación de atributos con las reducciones, porque cada reducción determina la producción que debe aplicarse. Los fragmentos de código se han obtenido de las reglas semánticas de la figura 5.2 sustituyendo cada atributo por una posición en la matriz *val*.

PRODUCCIÓN	FRAGMENTO DE CÓDIGO
$L \rightarrow E$ **n**	*print*(*val*[*tope*])
$E \rightarrow E_1 + T$	$val[ntope] := val[tope - 2] + val[tope]$
$E \rightarrow T$	
$T \rightarrow T_1 * F$	$val[ntope] := val[tope - 2] \times val[tope]$
$T \rightarrow F$	
$F \rightarrow (E)$	$val[ntope] := val[tope - 1]$
$F \rightarrow$ **dígito**	

Fig. 5.16. Implantación de una calculadora de escritorio con un analizador sintáctico LR.

Los fragmentos de código no muestran cómo se manejan las variables *tope* y *ntope*. Cuando se reduce una producción con r símbolos en el lado derecho, se asigna el valor de *ntope* a $tope - r + 1$. Después de ejecutar cada fragmento de código se asigna *tope* a *ntope*.

En la figura 5.17 se muestra la secuencia de movimientos hechos por el analizador sintáctico con la entrada `3*5+4n`. Los contenidos de los campos de *estado* y *val* de la pila del analizador sintáctico se muestran después de cada movimiento. De nuevo se toma la libertad de sustituir estados de la pila por sus correspondientes símbolos gramaticales. También se ha tomado la libertad de mostrar, en lugar del componente léxico **dígito**, el dígito real de entrada.

Considérese la secuencia de sucesos al ver el símbolo de entrada 3. En el primer movimiento, el analizador sintáctico desplaza el estado correspondiente al componente léxico **dígito** (cuyo valor de atributo es 3) dentro de la pila. (El estado se representa con 3 y el valor 3 está en el campo *val*.). En el segundo movimiento, el

analizador sintáctico reduce por la producción $F \rightarrow$ **dígito** e implanta la regla semántica $F.val := \textbf{dígito}.valex$. En el tercer movimiento, el analizador sintáctico reduce por $T \rightarrow F$. No se asocia ningún fragmento de código a esta producción y la matriz *val* no cambia. Obsérvese que después de cada reducción, el tope de la pila *val* tiene el valor del atributo asociado al lado izquierdo de la producción que reduce. □

ENTRADA	*estado*	*val*	PRODUCCIÓN UTILIZADA
3*5+4 **n**	_	_	
*5+4 **n**	3	3	
*5+4 **n**	F	3	$F \rightarrow$ **dígito**
*5+4 **n**	T	3	$T \rightarrow F$
5+4 **n**	T *	3 _	
+4 **n**	T * 5	3 _ 5	
+4 **n**	T * F	3 _ 5	$F \rightarrow$ **dígito**
+4 **n**	T	15	$T \rightarrow T * F$
+4 **n**	E	15	$E \rightarrow T$
4 **n**	E +	15 _	
n	E + 4	15 _ 4	
n	E + F	15 _ 4	$F \rightarrow$ **dígito**
n	E + T	15 _ 4	$T \rightarrow F$
n	E	19	$E \rightarrow E + T$
	E **n**	19 _	
	L	19	$L \rightarrow E$ **n**

Fig. 5.17. Movimientos realizados por el traductor con la entrada 3*5+4**n**.

En la implantación esbozada anteriormente, los fragmentos de código se ejecutan justo antes de que tenga lugar una reducción. Las reducciones proporcionan un "gancho" en el que se pueden colgar las acciones que constan de fragmentos de código arbitrarios. Es decir, se puede permitir al usuario que asocie una acción con una producción ejecutada cuando tiene lugar una reducción según dicha producción. Los esquemas de traducción que se considerarán en la siguiente sección proporcionan una notación para intercalar acciones con el análisis sintáctico. En la sección 5.6 se verá cómo se puede implantar una clase mayor de definiciones dirigidas por la sintaxis durante el análisis sintáctico ascendente.

5.4 DEFINICIONES CON ATRIBUTOS POR LA IZQUIERDA

Cuando la traducción tiene lugar durante el análisis sintáctico, el orden de evaluación de los atributos va unido al orden en el que, por el método de análisis sintáctico, se "crean" los nodos de un árbol de análisis sintáctico. Un orden natural que

caracteriza muchos métodos de traducción descendente y ascendente es el que se obtiene aplicando el procedimiento *visitaprof* de la figura 5.18 a la raíz de un árbol de análisis sintáctico. Este orden de evaluación se denomina *orden de evaluación en profundidad*. Aunque de hecho no se construye el árbol de análisis sintáctico, es útil estudiar la traducción durante el análisis sintáctico considerando la evaluación en profundidad de los atributos en los nodos de un árbol de análisis sintáctico.

```
procedure visitaprof(n : nodo);
begin
    for cada hijo m de n, de izquierda a derecha do begin
        evaluar los atributos heredados de m;
        visitaprof(m)
    end;
    evaluar los atributos sintetizados de n
end
```

Fig. 5.18. Orden de evaluación en profundidad para los atributos de un árbol de análisis sintáctico.

A continuación se introduce una clase de definiciones dirigidas por la sintaxis, llamadas definiciones con atributos por la izquierda, cuyos atributos siempre se pueden evaluar en un orden en profundidad. (Se les llama "por la izquierda" porque la información de los atributos parece fluir de izquierda a derecha.) En las siguientes tres secciones de este capítulo se considera la implantación de clases cada vez mayores de definiciones con atributos por la izquierda. Las definiciones con atributos por la izquierda incluyen todas las definiciones dirigidas por la sintaxis basadas en gramáticas LL(1); en la sección 5.5 se proporciona un método para implantar dichas definiciones en una sola pasada utilizando los métodos de análisis sintáctico predictivo. En la sección 5.6 se implanta una clase mayor de definiciones con atributos por la izquierda durante el análisis sintáctico ascendente, ampliando los métodos de traducción de la sección 5.3. En la sección 5.7 se esboza un método general para implantar todas las definiciones con atributos por la izquierda.

Definiciones con atributos por la izquierda

Una definición dirigida por la sintaxis es una definición con *atributos por la izquierda* si cada atributo heredado de X_j, $1 \leq j \leq n$, del lado derecho de $A \rightarrow X_1 X_2 \ldots X_n$, depende sólo de:

1. los atributos de los símbolos $X_1, X_2, \ldots, X_{j-1}$ a la izquierda de X_j en la producción y
2. los atributos heredados de A.

Obsérvese que toda definición con atributos sintetizados es una definición con atributos por la izquierda, porque las limitaciones 1 y 2 se refieren sólo a atributos heredados.

Ejemplo 5.11. La definición dirigida por la sintaxis de la figura 5.19 no es una definición con atributos por la izquierda porque el atributo heredado $Q.h$ del símbolo gramatical Q depende del atributo $R.s$ del símbolo gramatical a su derecha. En las secciones 5.8 y 5.9 se pueden encontrar otros ejemplos de definiciones que no son definiciones con atributos por la izquierda. □

Esquemas de traducción

Un esquema de traducción es una gramática independiente del contexto en la que se asocian atributos con los símbolos gramaticales y se insertan acciones semánticas encerradas entre llaves { } dentro de los lados derechos de las producciones, como en la sección 2.3. En este capítulo se utilizarán esquemas de traducción como una notación útil para especificar la traducción durante el análisis sintáctico.

PRODUCCIÓN	REGLAS SEMÁNTICAS
$A \rightarrow L\ M$	$L.i := l(A.i)$ $M.i := m(L.s)$ $A.s := f(M.s)$
$A \rightarrow Q\ R$	$R.i := r(A.i)$ $Q.i := q(R.s)$ $A.s := f(Q.s)$

Fig. 5.19. Una definición dirigida por la sintaxis que no es una definición con atributos por la izquierda.

Los esquemas de traducción considerados en este capítulo pueden tener tanto atributos sintetizados como heredados. En los esquemas de traducción simples considerados en el capítulo 2, los atributos son de tipo cadena, uno por cada símbolo, y para cada producción $A \rightarrow X_1 \ldots X_n$, la regla semántica formó la cadena de A mediante la concatenación de cadenas de $X_1, \ldots, X_n$, en orden, con algunas cadenas opcionales adicionales en medio. Se comprobó que se podría realizar la traducción imprimiendo simplemente las cadenas literales en el orden en que aparecían en las reglas semánticas.

Ejemplo 5.12. Este es un esquema de traducción simple que transforma expresiones infijas con suma y sustracción en las expresiones postfijas correspondientes. Es una pequeña reelaboración del esquema de traducción (2.14) del capítulo 2.

$$
\begin{aligned}
&E \rightarrow T\ R \\
&R \rightarrow \textbf{opsuma}\ T\ \{\ print(\textbf{opsuma}.lexema)\ \}\ R_1 \mid \epsilon \\
&T \rightarrow \textbf{núm}\ \{\ print(\textbf{núm}.val)\ \}
\end{aligned}
\tag{5.1}
$$

En la figura 5.20 se muestra el árbol de análisis sintáctico para la entrada 9−5+2 con cada acción semántica asociada como el hijo adecuado del nodo que corres-

ponde al lado izquierdo de su producción. En efecto, se consideran las acciones como si fueran símbolos terminales, un punto de vista que es mnemotécnicamente conveniente para establecer cuándo se deben ejecutar las acciones. Se ha tomado la libertad de mostrar los números reales y el operador aditivo en lugar de los componentes léxicos **núm** y **opsuma**. Cuando se realizan en orden de profundidad, las acciones de la figura 5.20 imprimen el resultado `95-2+`. □

Cuando se diseña un esquema de traducción, se deben respetar algunas limitaciones para asegurarse de que el valor de un atributo esté disponible cuando una acción se refiera a él. Estas limitaciones, motivadas por las definiciones con atributos por la izquierda, garantizan que las acciones no hagan referencia a un atributo que aún no haya sido calculado.

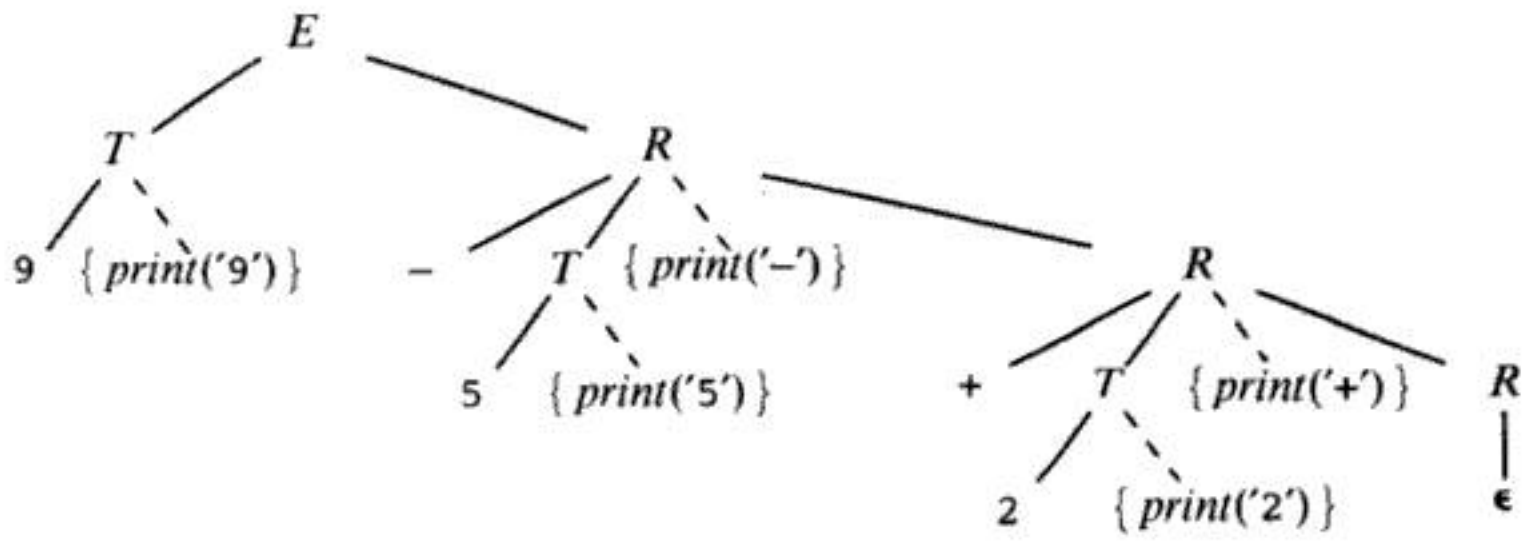


Fig. 5.20. Arbol de análisis sintáctico para 9−5+2 que muestra las acciones.

El ejemplo más sencillo ocurre cuando sólo se necesitan atributos sintetizados. En este caso, se puede construir el esquema de traducción creando una acción que conste de una asignación para cada regla semántica y colocando esta acción al final del lado derecho de la producción asociada. Por ejemplo, la producción y la regla semántica

PRODUCCIÓN	REGLA SEMÁNTICA
$T \rightarrow T_1 * F$	$T.val := T_1.val \times F.val$

dan como resultado la siguiente producción y acción semántica:

$$T \rightarrow T_1 * F \{ T.val := T_1.val \times F.val \}$$

Si se tienen atributos tanto heredados como sintetizados, se debe ser más prudente:

1. Un atributo heredado para un símbolo en el lado derecho de una producción se debe calcular en una acción antes que dicho símbolo.
2. Una acción no debe referirse a un atributo sintetizado de un símbolo que esté a la derecha de la acción.

3. Un atributo sintetizado para el no terminal de la izquierda sólo se puede calcular después de que se hayan calculado todos los atributos a los que hace referencia. La acción que calcula dichos atributos se puede colocar generalmente al final del lado derecho de la producción.

En las próximas dos secciones, se muestra cómo implantar un esquema de traducción que cumpla estos tres requisitos mediante generalizaciones de los analizadores sintácticos descentes y ascendentes.

El siguiente esquema de traducción no cumple el primero de los tres requisitos.

$S \rightarrow A_1\ A_2$ $\quad\quad$ $\{ A_1.her := 1; A_2.her := 2 \}$
$a \rightarrow a$ $\quad\quad$ $\{ print(A.her) \}$

Resulta que el atributo heredado $A.her$ en la segunda producción no está definido cuando se intenta imprimir su valor durante un recorrido en profundidad del árbol de análisis sintáctico para la cadena de entrada *aa*. Es decir, un recorrido en profundidad comienza en S y visita los subárboles para A_1 y A_2 antes de que se asignen los valores de $A_1.her$ y $A_2.her$. Si se intercala la acción que define los valores de $A_1.her$ y $A_2.her$ antes que los A en el lado derecho de $S \rightarrow A_1A_2$, en lugar de hacerlo después, entonces se definirá $A.her$ cada vez que ocurra $print(A.her)$.

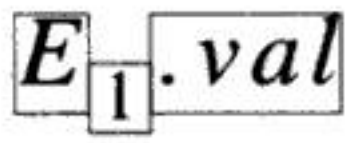


Fig. 5.21. Colocación de cajas dirigida por la sintaxis.

Siempre es posible comenzar con una definición dirigida por la sintaxis con atributos por la izquierda y construir un esquema de traducción que cumpla los tres requisitos anteriores. El siguiente ejemplo ilustra dicha construcción. Se basa en el lenguaje EQN para formatos matemáticos, que fue descrito brevemente en la sección 1.2. Dada la entrada

```
E sub 1 .val
```

EQN sitúa E, 1 y *.val* en las posiciones y tamaños relativos que se muestran en la figura 5.21. Obsérvese que el subíndice 1 se imprime en un tamaño y tipo menor, y se traslada hacia abajo en relación con E y *.val*.

Ejemplo 5.13. A partir de la definición con atributos por la izquierda de la figura 5.22, se construirá el esquema de traducción de la figura 5.23. En las figuras, el no terminal C (por caja) representa una fórmula. La producción $C \rightarrow C\ C$ representa la yuxtaposición de dos cajas, y $C \rightarrow C$ **sub** C representa la colocación de la segunda caja con subíndice en un tamaño menor que el de la primera caja en la posición relativa apropiada para un subíndice.

El atributo heredado *tp* (por tamaño del punto) afecta la altura de una fórmula. La regla para la producción $C \rightarrow$ **texto** hace que la altura normalizada del texto se

PRODUCCIÓN	REGLAS SEMÁNTICAS
$S \rightarrow C$	$C.tp := 10$ $S.al := C.al$
$C \rightarrow C_1\ C_2$	$C_1.tp := C.tp$ $C_2.tp := C.tp$ $C.al := máx(C_1.al, C_2.al)$
$C \rightarrow C_1$ **sub** C_2	$C_1.tp := C.tp$ $C_2.tp := contrae(C.tp)$ $C.al := desp(C_1.al, C_2.al)$
$C \rightarrow$ **texto**	$C.al :=$ **texto**$.a \times C.tp$

Fig. 5.22. Definición dirigida por la sintaxis para el tamaño y la altura de cajas.

multiplique por el tamaño del punto para obtener la altura real del texto. El atributo *a* de **texto** se obtiene mediante la búsqueda en una tabla, dado el carácter representado por el componente léxico **texto**. Cuando se aplica la producción $C \rightarrow C_1 C_2$, C_1 y C_2 heredan el tamaño del punto de C mediante reglas de copiado. La altura de B, representada por el atributo sintetizado *al*, es la máxima de las alturas de C_1 y C_2.

Cuando se utiliza la producción $C \rightarrow C_1$ **sub** C_2, la función *contrae* reduce el tamaño del punto de C_2 en un 30 %. La función *desp* permite el desplazamiento hacia abajo de la caja C_2 mientras calcula la altura de C. No se muestran las reglas que generan los mandatos de tipografía reales como salida.

La definición de la figura 5.22 es una definición con atributos por la izquierda. El único atributo heredado es *tp* del no terminal C. Cada regla semántica define a *tp* sólo en términos del atributo heredado del no terminal de la izquierda de la producción. Por tanto, es una definición con atributos por la izquierda.

$S \rightarrow$		$\{ C.tp := 10 \}$
	C	$\{ S.al := C.al \}$
$C \rightarrow$		$\{ C_1.tp := C.tp \}$
	C_1	$\{ C_2.tp := C.tp \}$
	C_2	$\{ C.al := máx(C_1.al, C_2.al) \}$
$C \rightarrow$		$\{ C_1.tp := C.tp \}$
	C_1	
	sub	$\{ C_2.tp := contrae(C.tp) \}$
	C_1	$\{ C.al := desp(C_1.al, C_2.al) \}$
$C \rightarrow$ **texto**		$\{ C.al :=$ **texto**$.a \times C.tp \}$

Fig. 5.23. Esquema de traducción construido a partir de la figura 5.22.

El esquema de traducción de la figura 5.23 se obtiene insertando asignaciones correspondientes a las reglas semánticas de la figura 5.22 dentro de las producciones, siguiendo los tres requisitos mencionados anteriormente. Para facilitar la lectura, cada símbolo gramatical en una producción se escribe en una línea distinta y las acciones aparecen a la derecha. Así,

$$S \rightarrow \{\ C.tp := 10\ \}\ C\ \{\ S.al :- C.al\ \}$$

se escribe como

$$\begin{array}{lll} S \rightarrow & & \{\ C.tp := 10\ \} \\ & C & \{\ S.al := C.al\ \} \end{array}$$

Obsérvese que las acciones que asignan los atributos heredados $C_1.tp$ y $C_2.tp$ aparecen justo antes que C_1 y C_2 en los lados derechos de las producciones. □

5.5 TRADUCCION DESCENDENTE

En esta sección, se implantarán las definiciones con atributos por la izquierda durante el análisis sintáctico predictivo. Se trabaja con esquemas de traducción en lugar de hacerlo con definiciones dirigidas por la sintaxis así que se puede ser explícito en cuanto al orden en que tienen lugar las acciones y las evaluaciones de los atributos. También se amplía el algoritmo para la eliminación de la recursión por la izquierda a esquemas de traducción con atributos sintetizados.

Eliminación de la recursión por la izquierda de un esquema de traducción

Como la mayoría de los operadores aritméticos son asociativos por la izquierda, es natural utilizar gramáticas recursivas por la izquierda para las expresiones. Ahora se amplía el algoritmo para eliminar la recursión por la izquierda de las secciones 2.4 y 4.3 para admitir atributos cuando se transforma la gramática subyacente de un esquema de traducción. La transformación se aplica a esquemas de traducción con atributos sintetizados. Ello permite que muchas de las definiciones dirigidas por la sintaxis de las secciones 5.1 y 5.2 se implanten mediante un análisis sintáctico predictivo. El siguiente ejemplo motiva la transformación.

Ejemplo 5.14. El esquema de traducción de la figura 5.24 se transforma en el esquema de traducción de la figura 5.25. El nuevo esquema produce al árbol de análisis sintáctico de la figura 5.26 para la expresión 9−5+2. Las flechas en la figura sugieren una forma de determinar el valor de la expresión.

En la figura 5.26, los números individuales son generados por T, y $T.val$ toma su valor del valor léxico del número, dado por el atributo **núm**.*val*. El 9 en la subexpresión 9−5 es generado por la T situada más a la izquierda pero el operador menos y el 5 son generados por la R en el hijo derecho de la raíz. El atributo heredado $R.h$ obtiene el valor 9 de $T.val$. La resta 9−5 y el paso del resultado 4 hasta el nodo medio de R se realizan intercalando la siguiente acción entre T y R_1 en $R \rightarrow - T R_1$:

$$\{\ R_1.h := R_1.h - T.val\ \}$$

$$
\begin{array}{ll}
E \rightarrow E_1 + T & \{ E.val := E_1\text{val} + T.val \} \\
E \rightarrow E_1 - T & \{ E.val := E_1\text{val} - T.val \} \\
E \rightarrow T & \{ E.val := T.val \} \\
T \rightarrow (E) & \{ T.val := E.val \} \\
T \rightarrow \textbf{núm} & \{ T.val := \textbf{núm}.val \}
\end{array}
$$

Fig. 5.24. Esquema de traducción con una gramática con recursividad por la izquierda.

Una acción similar suma 2 al valor de 9−5, produciendo el resultado $R.h = 6$ en el nodo inferior de R. Se necesita el resultado en la raíz como el valor de $E.val$; el atributo sintetizado s para R, que no aparece en la figura 5.26, se utiliza para copiar el resultado hasta la raíz. □

Para el análisis sintáctico descendente, se supone que una acción se ejecuta en el mismo momento en que se expandiría un símbolo en la misma posición. Por tanto, en la segunda producción de la figura 5.25, la primera acción (asignación a $R_1.h$) se realiza después de que T haya sido completamente expandida a símbolos terminales y la segunda acción se realiza después de que R_1 haya sido completamente expan-

$$
\begin{array}{ll}
E \rightarrow T & \{ R.h := T.val \} \\
\quad R & \{ E.val := R.s \} \\
R \rightarrow + & \\
\quad T & \{ R_1.h := R.h + T.val \} \\
\quad R_1 & \{ R.s := R_1.s \} \\
R \rightarrow - & \\
\quad T & \{ R_1.h := R.h - T.val \} \\
\quad R_1 & \{ R.s := R_1.s \} \\
R \rightarrow \epsilon & \{ R.s := R.h \} \\
T \rightarrow (& \\
\quad E & \\
\quad) & \{ T.val := E.val \} \\
T \rightarrow \textbf{núm} & \{ T.val := \textbf{núm}.val \}
\end{array}
$$

Fig. 5.25. Esquema de traducción transformado con una gramática con recursividad por la derecha.

dida. Como ya se mencionó en el estudio de las definiciones con atributos por la izquierda de la sección 5.4, un atributo heredado de un símbolo debe ser calculado por una acción que aparezca antes que el símbolo, y un atributo sintetizado del no terminal de la izquierda se debe calcular después de que hayan sido calculados todos los atributos de los que depende.

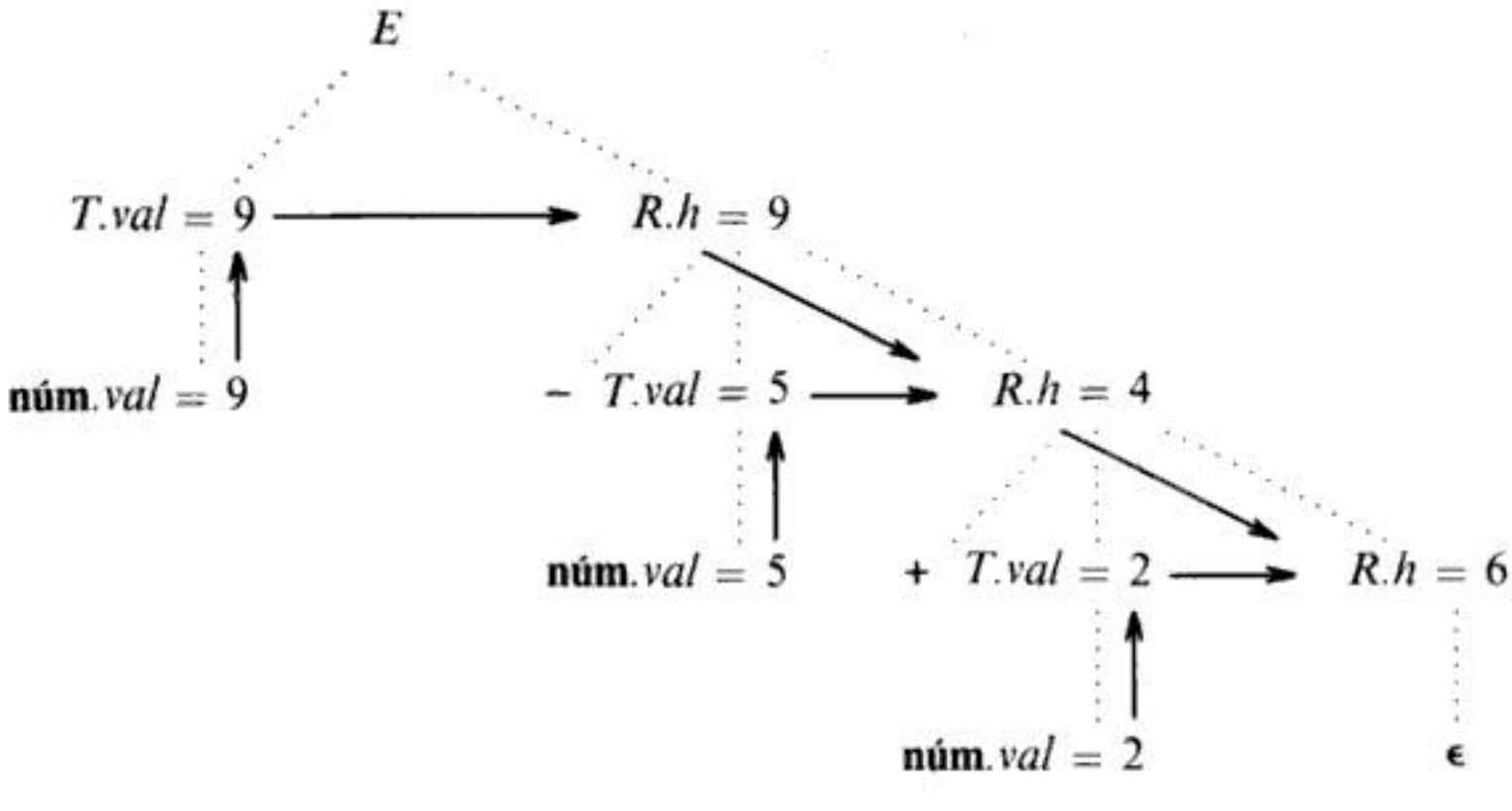


Fig. 5.26. Evaluación de la expresión 9−5+2.

Para adaptar otros esquemas de traducción recursivos por la izquierda al análisis sintáctico predictivo, se expresará el uso de los atributos $R.h$ y $R.s$ de la figura 5.25 de manera más abstracta. Supóngase que se tiene el siguiente esquema de traducción:

$$\begin{array}{ll} A \rightarrow A_1\ Y & \{ A.a := g(A_1.a,\ Y.y) \} \\ A \rightarrow X & \{ A.a := f(X.x) \} \end{array} \tag{5.2}$$

Cada símbolo gramatical tiene un atributo sintetizado escrito con la letra minúscula correspondiente, y f y g son funciones arbitrarias. Se puede realizar la generalización a producciones de A adicionales y a producciones con cadenas en lugar de símbolos X e Y tal como se hace en el ejemplo 5.15, más adelante.

El algoritmo para eliminar la recursión por la izquierda de la sección 2.4 construye la siguiente gramática a partir de (5.2):

$$\begin{array}{l} A \rightarrow X\ R \\ R \rightarrow Y\ R \mid \epsilon \end{array} \tag{5.3}$$

Teniendo en cuenta las acciones semánticas, el esquema transformado se convierte en

$$\begin{array}{lll} A \rightarrow X & \{ & R.h := f(X.x) \} \\ \quad R & \{ & A.a := R.s \} \\ R \rightarrow Y & \{ & R_1.h := g(R.h,\ Y.y) \} \\ \quad R_1 & \{ & R.s := R_1.s \} \\ R \rightarrow \epsilon & \{ & R.s := R.h \} \end{array} \tag{5.4}$$

El esquema transformado utiliza atributos h y s para R, igual que en la figura 5.25. Para saber por qué los resultados de (5.2) y (5.4) son el mismo, considérense los dos árboles de análisis sintáctico con anotaciones de la figura 5.27. El valor de $A.a$ se calcula según (5.2) de la figura 5.27(a). La figura 5.27(b) contiene el cálculo

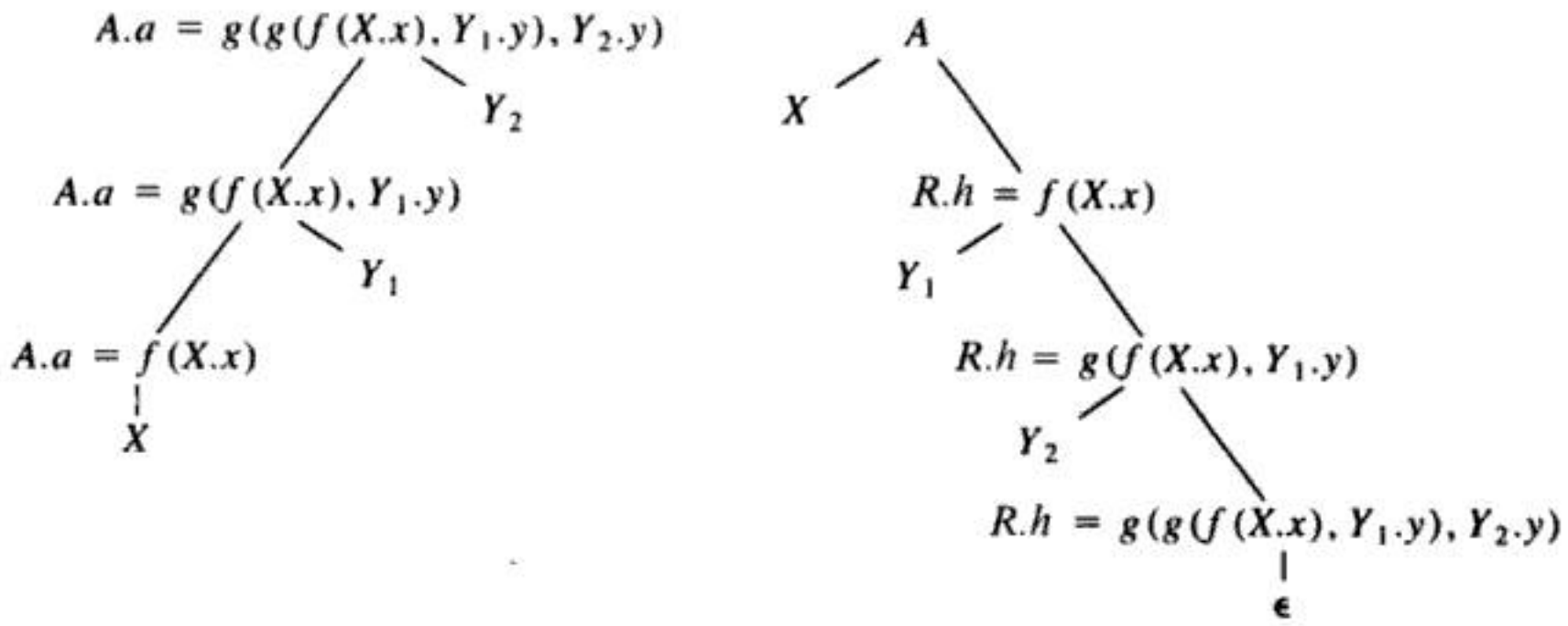


Fig. 5.27. Dos formas de calcular un valor de atributo.

de $R.h$ a lo largo del árbol según (5.4). El valor de $R.h$ en el fondo se traslada arriba sin modificar, como $R.s$, y se convierte en el valor correcto de $A.a$ en la raíz ($R.s$ no se muestra en la Fig. 5.27(b)).

Ejemplo 5.15. Si se convierte la definición dirigida por la sintaxis de la figura 5.9 para construir árboles sintácticos en un esquema de traducción, entonces las producciones y las acciones semánticas para E resultan:

$$
\begin{array}{ll}
E \rightarrow E_1 + T & \{\, E.apn := haznodo('+', E_1.apn, T.apn) \,\} \\
E \rightarrow E_1 - T & \{\, E.apn := haznodo('-', E_1.apn, T.apn) \,\} \\
E \rightarrow T & \{\, E.apn := T.apn \,\}
\end{array}
$$

Cuando se elimina la recursión por la izquierda de este esquema de traducción, el no terminal E corresponde a A en (5.2) y las cadenas $+T$ y $-T$ de las primeras dos producciones corresponden a Y; el no terminal T de la tercera producción corresponde a X. En la figura 5.28 se muestra el esquema de traducción transformado. Las producciones y las acciones semánticas para T son similares a las de la definición original de la figura 5.9.

En la figura 5.29 se muestra cómo las acciones de la figura 5.28 construyen un árbol sintáctico para a−4+c. Los atributos sintetizados se muestran a la derecha del nodo para un símbolo gramatical, y los atributos heredados a la izquierda. En el árbol sintáctico, una hoja se construye mediante acciones asociadas con las producciones $T \rightarrow$ **id** y $T \rightarrow$ **núm**, como en el ejemplo 5.8. En la T situada más a la izquierda, el atributo $T.apn$ apunta a la hoja para a. En el lado derecho de $E \rightarrow T\,R$ se hereda un apuntador al nodo para a como atributo $R.h$.

Cuando se aplica la producción $R \rightarrow -T\,R_1$ al hijo derecho de la raíz, $R.h$ apunta al nodo para a, y $T.apn$ al nodo para 4. El nodo para a−4 se construye aplicando *haznodo* al operador menos y a dichos apuntadores.

$E \rightarrow T$	$\{ R.h := T.apn \}$
$\quad R$	$\{ E.apn := R.s \}$
$R \rightarrow +$	
$\quad T$	$\{ R_1.h := haznodo('+', R.h, T.apn) \}$
$\quad R_1$	$\{ R.s := R_1.s \}$
$R \rightarrow -$	
$\quad T$	$\{ R_1.h := haznodo('-', R.h, T.apn) \}$
$\quad R_1$	$\{ R.s := R_1.s \}$
$R \rightarrow \epsilon$	$\{ R.s := R.h \}$
$T \rightarrow ($	
$\quad E$	
$\quad)$	$\{ T.apn := E.apn \}$
$T \rightarrow$ **id**	$\{ T.apn := hazhoja(\textbf{id}, \textbf{id}.entrada) \}$
$T \rightarrow$ **núm**	$\{ T.apn := hazhoja(\textbf{núm}, \textbf{núm}.val) \}$

Fig. 5.28. Esquema de traducción transformado para construir árboles sintácticos.

Por último, cuando se aplica la producción $R \rightarrow \epsilon$, $R.h$ apunta a la raíz de todo el árbol sintáctico. Todo el árbol se devuelve a través de los atributos s de los nodos para R (que no se muestran en la Fig. 5.29) hasta que se convierte en el valor de $E.apn$. □

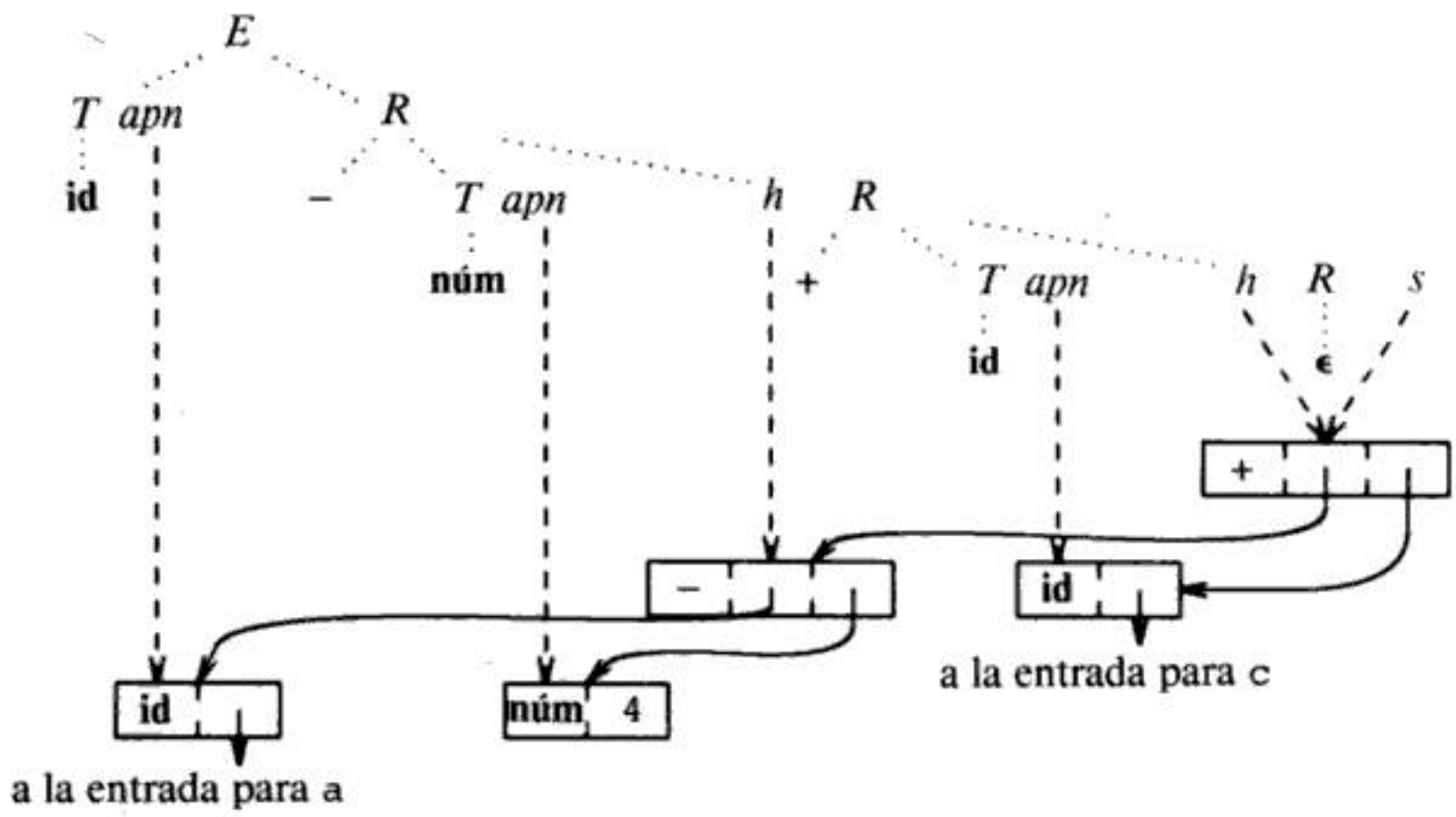


Fig. 5.29. Uso de atributos heredados para construir árboles sintácticos.

Diseño de un traductor predictivo

El siguiente algoritmo generaliza la construcción de analizadores sintácticos predictivos para implantar un esquema de traducción basado en una gramática adecuada para el análisis sintáctico descendente.

Algoritmo 5.2. Construcción de un traductor predictivo dirigido por la sintaxis.

Entrada. Un esquema de traducción dirigido por la sintaxis con una gramática subyacente adecuada para el análisis sintáctico predictivo.

Salida. Código del traductor dirigido por la sintaxis.

Método. La técnica es una modificación de la construcción de analizadores sintácticos predictivos de la sección 2.4.

1. Para cada no terminal A, constrúyase una función que tenga un parámetro formal por cada atributo heredado de A y que devuelva los valores de los atributos sintetizados de A (posiblemente como un registro, como un apuntador a un registro con un campo para cada atributo, o utilizando el mecanismo de llamada por referencia para el paso de parámetros, estudiado en la Sec. 7.5). Para simplificar, se supone que cada no terminal sólo tiene un atributo sintetizado. La función para A tiene una variable local para cada atributo de cada símbolo gramatical que aparezca en una producción para A.
2. Al igual que en la sección 2.4, el código para el no terminal A decide qué producción utilizar basándose en el símbolo en curso de entrada.
3. El código asociado con cada producción hace lo siguiente. Se consideran los componentes léxicos, no terminales y acciones del lado derecho de la producción de izquierda a derecha.
   i) Para el componente léxico X con atributo sintetizado x, guárdese el valor de x en la variable declarada para $X.x$. Después genérese una llamada para concordar el componente léxico X y avánce se la entrada.
   ii) Para el no terminal B, genérese una asignación $c := B(b_1, b_2, \ldots, b_k)$ con la llamada a una función en el lado derecho, donde $b_1, b_2, \ldots, b_k$ son las variables para los atributos heredados de B y c es la variable para el atributo sintetizado de B.
   iii) Para el caso de una acción, cópiese el código dentro del analizador sintáctico, sustituyendo cada referencia a un atributo por la variable correspondiente a dicho atributo. □

Se amplía el algoritmo 5.2 en la sección 5.7 para implantar cualquier definición con atributos por la izquierda, con la condición de que ya se haya construido un árbol de análisis sintáctico. En la sección 5.8 se consideran algunas maneras de mejorar los traductores construidos por el algoritmo 5.2. Por ejemplo, puede ser posible eliminar la copia de proposiciones de la forma $x := y$ o utilizar una sola variable para guardar los valores de varios atributos. Algunas de estas mejoras también se pueden realizar automáticamente con los métodos del capítulo 10.

Ejemplo 5.16. La gramática de la figura 5.28 es LL(1) y por tanto resulta adecuada para el análisis sintáctico descendente. A partir de los atributos de los no terminales de la gramática, se obtienen los tipos siguientes para los argumentos y resultados de las funciones para F, R y T. Como E y T no tienen atributos heredados, tampoco tienen argumentos.

```
function E : ↑ nodo_árbol_sintáctico;
function R(h : ↑ nodo_árbol_sintáctico): ↑ nodo_árbol_sintáctico;
function T : ↑ nodo_árbol_sintáctico;
```

Se combinan dos de las producciones de R de la figura 5.28 para hacer el traductor más pequeño. Las nuevas producciones utilizan el componente léxico **opsuma** para representar + y −:

$$
\begin{array}{lll}
R \rightarrow \textbf{opsuma} & & \\
\quad T & \{ R_1.h := \mathit{haznodo}(\textbf{opsuma}.\mathit{lexema}, R.h, T.apn) \} & \\
\quad R_1 & \{ R.s := R_1.s \} & (5.5) \\
R \rightarrow \epsilon & \{ R.s := R.h \} &
\end{array}
$$

El código para R se basa en el procedimiento de análisis de la figura 5.30. Si el símbolo de anticipación es **opsuma**, entonces la producción $R \rightarrow \textbf{opsuma}\ T\ R$ se aplica por el procedimiento *concuerda* para leer el componente léxico de entrada que sigue a **opsuma**, y llama luego a los procedimientos para T y R. De lo contrario, el procedimiento no hace nada, para imitar la producción $R \rightarrow \epsilon$.

```
procedure R;
begin
    if símbolo-anticipación = opsuma then begin
        concuerda(opsuma); T; R
    end
    else begin /* no hacer nada */
    end
end;
```

Fig. 5.30. Procedimiento de análisis sintáctico para las producciones $R \rightarrow \textbf{opsuma}\ T\ R \mid \epsilon$.

El procedimiento para R de la figura 5.31 contiene código para evaluar atributos. Se guarda el valor léxico *valex* del componente léxico **opsuma** en *lexemaopsuma*, se concuerda **opsuma**, se llama a T y se guarda su resultado utilizando *apn*. La variable *h1* corresponde al atributo heredado $R_1.h$, y *s1* corresponde al atributo sintetizado $R_1.s$. La proposición **return** devuelve el valor de s justo antes de que el control deje la función. Las funciones para E y T se construyen igual. □

```
function R(h: ↑ nodo árbol sintáctico): ↑ nodo árbol sintáctico;
    var apn, h1, s1, s: ↑ nodo árbol sintáctico;
        lexemaopsuma: char;
begin
    if símbolo-anticipación = opsuma then begin
        /* producción R → opsuma T R */
        lexemaopsuma := valex;
        concuerda(opsuma);
        apn := T;
        h1 := haznodo(lexemaopsuma, h, apn);
        s1 := R(h1);
        s := s1
    end
    else s := h; /* producción R → ϵ */
    return s
end;
```

Fig. 5.31. Construcción por descenso recursivo de árboles sintácticos.

5.6 EVALUACION ASCENDENTE DE LOS ATRIBUTOS HEREDADOS

En esta sección se introduce un método para implantar definiciones con atributos por la izquierda en el contexto del análisis sintáctico ascendente. El método es capaz de manejar todas las definiciones con atributos por la izquierda considerados en la sección anterior, en el sentido de que puede implantar cualquier definición con atributos por la izquierda basada en una gramática LL(1). También puede implantar muchas (pero no todas) definiciones con atributos por la izquierda basadas en gramáticas LR(1). El método es una generalización de la técnica de traducción ascendente estudiada en la sección 5.3.

Eliminación de los esquemas de traducción de acciones intercaladas

En el método de traducción ascendente de la sección 5.3 se suponía que todas las acciones de traducción estaban en el lado derecho de la producción, mientras que en el método de análisis sintáctico predictivo de la sección 5.5 era necesario intercalar acciones en varios lugares dentro del lado derecho. Para comenzar el estudio de cómo se pueden manejar los atributos heredados de forma ascendente, se introduce una transformación que hace que todas las acciones intercaladas en un esquema de traducción ocurran en los extremos derechos de sus producciones.

La transformación inserta nuevos no terminales *marcadores* que generan ϵ dentro de la gramática básica. Se sustituye cada acción intercalada por un no terminal marcador distinto M y se asocia la acción al final de la producción $M \rightarrow \epsilon$. Por ejemplo, el esquema de traducción

$E \rightarrow T\ R$
$R \rightarrow +\ T\ \{\ print('+')\ \}\ R \mid -\ T\ \{\ print('-')\ \}\ R \mid \epsilon$
$T \rightarrow$ **núm** $\{\ print(\textbf{núm}.val)\ \}$

se transforma utilizando los no terminales marcadores M y N en

$$
\begin{aligned}
&E \rightarrow T\ R\\
&R \rightarrow +\ T\ M\ R \mid -\ T\ N\ R \mid \epsilon\\
&T \rightarrow \textbf{núm}\ \{\ print(\textbf{núm}.val)\ \}\\
&M \rightarrow \epsilon\ \{\ print('+')\ \}\\
&N \rightarrow \epsilon\ \{\ print('-')\ \}
\end{aligned}
$$

Las gramáticas en los dos esquemas de traducción aceptan exactamente el mismo lenguaje y, si se dibuja un árbol de análisis sintáctico con nodos adicionales para las acciones, se puede demostrar que las acciones se realizan en el mismo orden. Las acciones en el esquema de traducción transformado terminan producciones, así que pueden realizarse justo antes de que se reduzca el lado derecho durante el análisis sintáctico ascendente.

Herencia de atributos en la pila del analizador sintáctico

Un analizador sintáctico ascendente reduce el lado derecho de la producción $A \rightarrow XY$ eliminando X e Y del tope de la pila y sustituyéndolas por A. Supóngase que X tiene un atributo sintetizado $X.s$, que la implantación de la sección 5.3 conservaba junto a X en la pila del analizador.

Como el valor de $X.s$ ya está en la pila del analizador antes de que tenga lugar cualquier reducción en el subárbol más abajo de Y, este valor puede ser heredado por Y. Es decir, si el atributo heredado $Y.h$ está definido por la regla de copia $Y.h := X.s$, entonces el valor $X.s$ se puede utilizar donde se llama a $Y.h$. Como se verá, las reglas de copia desempeñan un importante papel en la evaluación de atributos heredados durante el análisis sintáctico ascendente.

Ejemplo 5.17. Se puede pasar el tipo de un identificador mediante reglas de copia usando atributos heredados, como se muestra en la figura 5.32 (adaptada de la Fig. 5.7). Primero se examinarán los movimientos realizados por un analizador sintáctico ascendente con la entrada

```
real p, q, r
```

Después se mostrará cómo se puede acceder al valor del atributo $T.tipo$ cuando se aplican las producciones para L. El esquema de traducción que se desea implantar es

$$
\begin{aligned}
&D \rightarrow T && \{\ L.her := T.tipo\ \}\\
&\quad\ L\\
&T \rightarrow \textbf{int} && \{\ T.tipo := integer\ \}\\
&T \rightarrow \textbf{real} && \{\ T.tipo := real\ \}\\
&L \rightarrow && \{\ L_1.her := L.her\ \}\\
&\quad\ L_1,\ \textbf{id} && \{\ añadetipo(\textbf{id}.entrada,\ L.her)\ \}\\
&L \rightarrow \textbf{id} && \{\ añadetipo(\textbf{id}.entrada,\ L.her)\ \}
\end{aligned}
$$

Si no se tienen en cuenta las acciones en el esquema de traducción anterior, la secuencia de movimientos realizadas por el analizador sintáctico con la entrada de

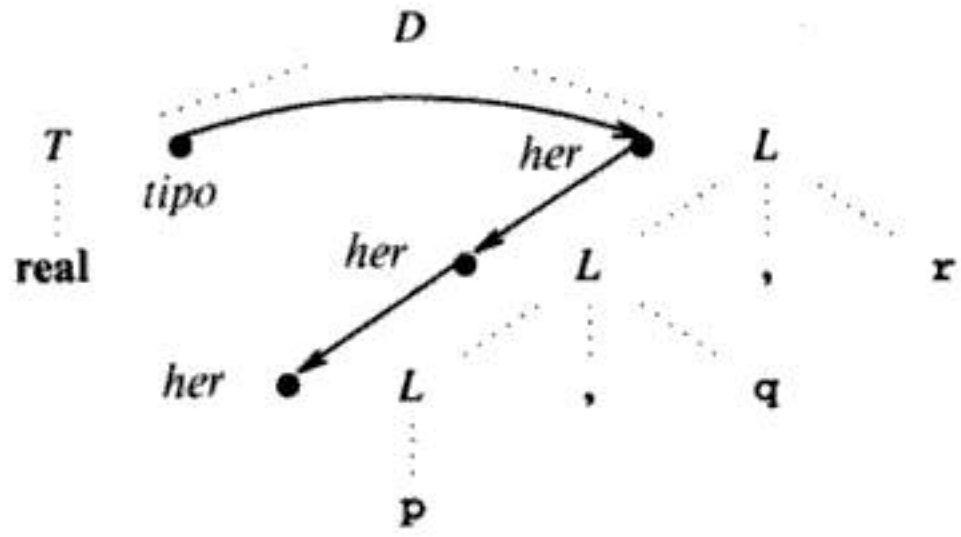


Fig. 5.32. En cada nodo para L, $L.her = T.tipo$.

la figura 5.32 es igual a la de la figura 5.33. Para una mayor claridad, se muestra el símbolo gramatical correspondiente en lugar de un estado de la pila y el identificador real del componente léxico **id**.

Supóngase, como en la sección 5.3, que la pila del analizador sintáctico se implanta como un par de matrices, *estado* y *val*. Si *estado*[i] es para el símbolo gramatical X, entonces *val*[i] guarda un atributo sintetizado $X.s$. En la figura 5.33 se muestran los contenidos de la matriz *estado*. Obsérvese que cada vez que el lado derecho de una producción de L se reduce en la figura 5.33, T está en la pila justo por debajo del lado derecho. Este hecho se puede utilizar para acceder al valor del atributo $T.tipo$.

La implantación de la figura 5.34 utiliza el hecho de que el atributo $T.tipo$ está en un lugar conocido de la pila *val*, relativo al tope. Sean *tope* y *ntope* los índices de la entrada del tope en la pila justo antes y después de que tenga lugar una reducción, respectivamente. Por las reglas de copia que definen $L.her$, se sabe que se puede utilizar $T.tipo$ en lugar de $L.her$.

ENTRADA	*estado*	PRODUCCIÓN UTILIZADA
real p,q,r	–	
p,q,r	**real**	
p,q,r	T	$T \rightarrow$ **real**
,q,r	T p	
,q,r	TL	$L \rightarrow$ **id**
q,r	TL,	
,r	TL, q	
,r	TL	$L \rightarrow L$, **id**
r	TL,	
	TL, r	
	TL	$L \rightarrow L$, **id**
	D	$D \rightarrow TL$

Fig. 5.33. Siempre que se reduce un lado derecho para L, T está justo abajo del lado derecho.

Cuando se aplica la producción $L \rightarrow$ **id**, **id**.*entrada* está en el tope de la pila *val* y *T.tipo* se encuentra justo por debajo de ella. Por tanto, *añadetipo*(*val* [*tope*], *val* [*tope* − 1] cuando es equivalente a *añadetipo* (**id**.*entrada*, *T.tipo*). De manera similar, como el lado derecho de la producción $L \rightarrow L$, **id** tiene tres símbolos, *T.tipo* aparece en *val*[*tope* − 3] cuando tiene lugar esta reducción. Se eliminan las reglas de copia que se refieren a *L.her* porque en su lugar se utiliza el valor de *T.tipo* en la pila. □

PRODUCCIÓN	FRAGMENTO DE CÓDIGO
$D \rightarrow T\ L$;	
$T \rightarrow$ **int**	*val*[*ntope*] := *integer*
$T \rightarrow$ **real**	*val*[*ntope*] := *real*
$L \rightarrow L$, **id**	*añadetipo*(*val*[*tope*], *val*[*tope* − 3])
$L \rightarrow$ **id**	*añadetipo*(*val*[*tope*], *val*[*tope* − 1])

Fig. 5.34. El valor de *T.tipo* se utiliza en lugar de *L.her*.

Simulación de la evaluación de atributos heredados

Alcanzar el valor de un atributo dentro de la pila del analizador sintáctico se puede conseguir inicialmente si la gramática permite predecir la posición del valor del atributo.

Ejemplo 5.18. Como ejemplo de un caso en que no se puede predecir la posición, considérese el siguiente esquema de traducción.

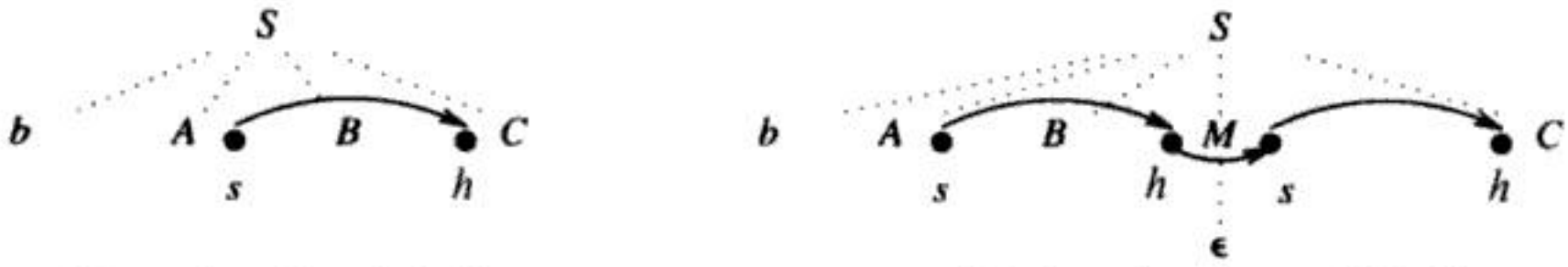
PRODUCCIÓN	REGLAS SEMÁNTICAS
$S \rightarrow aAC$	$C.h := A.s$
$S \rightarrow bABC$	$C.h := A.s$
$C \rightarrow c$	$C.s := g(C.h)$

(5.6)

C hereda el atributo sintetizado $A.s$ por una regla de copia. Obsérvese que puede o no haber una B entre A y C en la pila. Cuando se realiza la reducción por $C \rightarrow c$, el valor de $C.h$ está en $val[tope - 1]$ o en $val[tope - 2]$, pero no está claro qué caso corresponde.

En la figura 5.35 se inserta un no terminal marcador M nuevo justo antes de la C en el lado derecho de la segunda producción de (5.6). Si se está analizando según la producción $S \rightarrow bABMC$, entonces $C.h$ hereda el valor de $A.s$ indirectamente a través de $M.h$ y $M.s$. Cuando se aplica la producción $M \rightarrow \epsilon$, una regla de copia $M.s := M.h$ garantiza que el valor $M.s = M.h = A.s$ aparezca justamente antes de la parte de la pila utilizada para analizar el subárbol de C. Por consiguiente, se puede encontrar el valor de $C.h$ en $val\,[tope - 1]$ cuando se aplica $C \rightarrow c$, independientemente de si se utilizan la primera o la segunda producción en la siguiente modificación de (5.6).

PRODUCCIÓN	REGLAS SEMÁNTICAS
$S \rightarrow aAC$	$C.h := A.s$
$S \rightarrow bABMC$	$M.h := A.s$; $C.h := M.s$
$C \rightarrow c$	$C.s := g(C.h)$
$M \rightarrow \epsilon$	$M.s := M.h$



(a) producción original (b) dependencias modificadas

Fig. 5.35. Copiado de un valor de atributo a través de un marcador M.

Los no terminales marcadores también pueden utilizarse para simular reglas semánticas que no sean reglas de copia. Por ejemplo, considérese

PRODUCCIÓN	REGLAS SEMÁNTICAS	
$S \rightarrow aAC$	$C.h := f(A.s)$	(5.7)

Esta vez la regla que define a $C.h$ no es una regla de copia, de modo que el valor de $C.h$ no está todavía en la pila *val*. También se puede resolver este problema utilizando un marcador.

PRODUCCIÓN	REGLAS SEMÁNTICAS	
$S \rightarrow aANC$	$N.h := A.s$; $C.h := N.s$	(5.8)
$N \rightarrow \epsilon$	$N.s := f(N.h)$	

El no terminal distinto N hereda $A.s$ por una regla de copia. Su atributo sintetizado $N.s$ se asigna a $f(A.s)$; después $C.h$ hereda este valor utilizando una regla de copia. Cuando se reduce por $N \rightarrow \epsilon$, el valor de $N.h$ se encuentra en el lugar para $A.s$, es decir, en $val[tope - 1]$. Cuando se reduce por $S \rightarrow aANC$, también se encuentra el valor de $C.h$ en $val[tope - 1]$, porque es $N.s$. En realidad, $C.h$ no es necesario en este momento; fue necesario durante la reducción de una cadena de terminales a C, cuando su valor estaba almacenado y a salvo en la pila con N. □

Ejemplo 5.19. En la figura 5.36 se utilizan tres no terminales marcadores L, M y N para garantizar que el valor del atributo heredado $C.tp$ aparezca en una posición conocida en la pila del analizador mientras se hace la reducción del subárbol para C. La gramática con atributos original aparece en la figura 5.22 y su importancia para la formación de textos ya se ha explicado en el ejemplo 5.13.

La inicialización se realiza usando L. La producción para S es $S \rightarrow L\,C$ en la figura 5.36, de modo que L permanecerá en la pila mientras se reduce el subárbol debajo de C. El valor 10 del atributo heredado $C.tp = L.s$ se introduce en la pila del analizador mediante la regla $L.s := 10$ asociada con $L \rightarrow \epsilon$.

PRODUCCIÓN	REGLAS SEMÁNTICAS
$S \rightarrow L\ C$	$C.tp := L.s$ $S.al := C.al$
$L \rightarrow \epsilon$	$L.s := 10$
$C \rightarrow C_1\ M\ C_2$	$C_1.tp := C.tp$ $M.h := C.tp$ $C_2.tp := M.s$ $C.al := máx(C_1.al, C_2.al)$
$C \rightarrow C_1$ **sub** $N\ C_2$	$C_1.tp := C.tp$ $N.h := C.tp$ $C_2.tp := desp(C_1.al, C_2.al)$
$C \rightarrow$ **texto**	$C.al :=$ **texto**$.a \times C.tp$
$M \rightarrow \epsilon$	$M.s := M.h$
$N \rightarrow \epsilon$	$N.s := contrae(N.h)$

Fig. 5.36. Todos los atributos heredados son asignados por reglas de copia.

El marcador M en $C \rightarrow C_1\ M\ C_2$ desempeña un papel similar al de M en la figura 5.35; garantiza que el valor de $C.tp$ aparezca justo debajo de C_2 en la pila del analizador. En la producción $C \rightarrow C_1$ **sub** $N\ C_2$, el no terminal N se utiliza como en (5.8). N hereda, por medio de la regla de copia $N.h := C.tp$, el valor de atributo del que depende $C_2.tp$, y sintetiza el valor de $C_2.tp$ por la regla $N.s := contrae(N.h)$. La consecuencia, que se deja como ejercicio, es que el valor de $C.tp$ siempre está inmediatamente debajo del lado derecho cuando se reduce a C.

En la figura 5.37 se muestran los fragmentos de código que implantan la definición dirigida por la sintaxis de la figura 5.36. Todos los atributos heredados son asignados por reglas de copia en la figura 5.36, así que la implantación obtiene sus valores localizando la posición que ocupan en la pila *val*. Como en los ejemplos anteriores, *tope* y *ntope* dan los índices del tope de la pila antes y después de una reducción, respectivamente. □

PRODUCCIÓN	FRAGMENTO DE CÓDIGO
$S \rightarrow L\ C$	$val[ntope] := val[tope]$
$L \rightarrow$	$val[ntope] := 10$
$C \rightarrow C_1\ M\ C_2$	$val[ntope] := máx(val[tope - 2], val[tope])$
$C \rightarrow C_1$ **sub** $N\ C_2$	$val[ntope] := desp(val[tope - 3], val[tope])$
$C \rightarrow$ **texto**	$val[ntope] := val[tope] \times val[tope - 1]$
$M \rightarrow \epsilon$	$val[ntope] := val[tope - 1]$
$N \rightarrow \epsilon$	$val[ntope] := contrae(val[tope - 2])$

Fig. 5.37. Implantación de la definición dirigida por la sintaxis de la figura 5.36.

La introducción sistemática de marcadores, como en la modificación de (5.6) y (5.7), posibilita la evaluación de definiciones con atributos por la izquierda durante el análisis sintáctico LR. Como sólo hay una producción por cada marcador, una gramática sigue siendo LL(1) cuando se añaden los marcadores. Cualquier gramática LL(1) también es una gramática LR(1) de modo que no surgen conflictos de análisis sintáctico cuando se añaden marcadores a una gramática LR(1). Desgraciadamente, no se puede decir lo mismo de todas las gramáticas LR(1); es decir, pueden surgir conflictos de análisis sintáctico si se introducen marcadores en ciertas gramáticas LR(1).

Las ideas de los ejemplos anteriores pueden formalizarse en el siguiente algoritmo.

Algoritmo 5.3. Análisis sintáctico y traducción ascendentes con atributos heredados.

Entrada. Una definición con atributos por la izquierda con una gramática subyacente LL(1).

Salida. Un analizador sintáctico que calcula los valores de todos los atributos en su pila de análisis sintáctico.

Método. Supóngase para simplificar, que todo no terminal a tiene un atributo heredado $A.h$ y que todo símbolo gramatical X tiene un atributo sintetizado $X.s$. Si X es un terminal, entonces su atributo sintetizado es en realidad el valor léxico devuelto con X por el analizador léxico; este valor léxico aparece en la pila, en una matriz *val*, como en los ejemplos anteriores.

Para toda producción $A \rightarrow X_1 \ldots X_n$, introdúzcase n no terminales marcadores nuevos, $M_1, \ldots, M_n$, y sustitúyase la producción por $A \rightarrow M_1X_1 \ldots M_nX_n$[2]. El atributo sintetizado $X_j.s$ irá en la pila del analizador en la entrada de la matriz *val* asociada con X_j. El atributo heredado $X_j.h$, si es que lo hay, aparece en la misma matriz, pero asociado con M_j.

Una propiedad invariante importante es que conforme se realiza el análisis sintáctico, el atributo heredado $A.h$, si es que existe, se encuentra en la posición de la matriz *val* inmediatamente debajo de la posición para M_1. Como el símbolo inicial no tiene ningún atributo heredado, el caso no supone ningún problema cuando el símbolo inicial es A, pero incluso si hubiera un atributo heredado, se podría colocar debajo del fondo de la pila. Se puede demostrar la invariante mediante una sencilla inducción sobre el número de pasos del análisis sintáctico ascendente, observando que los atributos heredados se asocian con los no terminales marcadores M_j y que el atributo $X_j.h$ se calcula en M_j antes de comenzar la reducción a X_j.

Para comprobar si los atributos se pueden calcular como se pretendía durante un análisis sintáctico ascendente, considérense dos casos. Primero, si se reduce a un no terminal marcador M_j, se sabe a qué producción $A \rightarrow M_1X_1 \ldots M_n X_n$ pertenece este

---

[2] Aunque insertar M_1 antes que X_1 simplifica el estudio de los no terminales marcadores, tiene el desafortunado efecto colateral de introducir conflictos de análisis sintáctico en una gramática con recursividad por la izquierda. Véase el ejercicio 5.21. Como se observa más adelante, se puede eliminar M_1.

marcador. Por tanto, se conocen las posiciones de cualesquiera atributos que el atributo heredado $X_j.h$ necesite para su cálculo. $A.h$ está en $val\,[tope - 2j + 2]$, $X_1.h$ está en $val\,[tope - 2j + 3]$, $X_1.s$ está en $val\,[tope - 2j + 4]$, $X_2.h$ está en $val\,[tope - 2j + 5]$, y así sucesivamente. Por tanto, se puede calcular $X_j.h$ y guardarla en $val[tope + 1]$, que se convierte en el nuevo tope de la pila después de la reducción. Obsérvese que es importante que la gramática sea LL(1), porque de otro modo no se podría estar seguro de que se está reduciendo ϵ a un determinado no terminal marcador y por tanto no se podrían localizar los atributos adecuados, o incluso conocer la fórmula a aplicar en general. Se pide al lector que haga un acto de fe o que demuestre el hecho de que toda gramática LL(1) con marcadores siga siendo LR(1).

El segundo caso ocurre cuando se reduce a un símbolo no marcador, por ejemplo por la producción $A \rightarrow M_1X_1 \ldots M_nX_n$. Entonces sólo hay que calcular el atributo sintetizado $A.s$; obsérvese que $A.h$ ya ha sido calculado, y reside en la posición de la pila justo debajo de la posición en la que se inserta la misma A. Los atributos necesarios para calcular $A.s$ están claramente disponibles en posiciones conocidas de la pila, las posiciones de las X_j, durante la reducción.

Las siguientes simplificaciones reducen el número de marcadores; la segunda evita los conflictos de análisis sintáctico en gramáticas con recursividad por la izquierda.

1. Si X_j no tiene atributo heredado, no es necesario utilizar el marcador M_j. Por supuesto, las posiciones previstas de los atributos en la pila cambiarán si se omite M_j, pero este cambio se puede incorporar fácilmente al analizador sintáctico.
2. Si existe $X_1.h$, pero se calcula mediante una regla de copia $X_1.h := A.h$, entonces se puede omitir M_1, puesto que por la constante se sabe que $A.h$ ya estará ubicada allí donde se desea, justo debajo de X_1 en la pila, y este valor también puede, por tanto, servir para X_1. □

Sustitución de atributos heredados por atributos sintetizados

A veces es posible evitar el uso de atributos heredados mediante la modificación de la gramática subyacente. Por ejemplo, una declaración en Pascal puede constar de una lista de identificadores seguida de un tipo, por ejemplo, `m, n : integer`. Una gramática para dichas declaraciones puede incluir producciones de la forma

$$D \rightarrow L : T$$
$$T \rightarrow \textbf{integer} \mid \textbf{char}$$
$$L \rightarrow L\ ,\ \textbf{id} \mid \textbf{id}$$

Como los identificadores son generados por L pero el tipo no está en el subárbol para L, no se puede asociar el tipo con un identificador utilizando únicamente atributos sintetizados. De hecho, si el no terminal L hereda un tipo de T a su derecha en la primera producción, se obtiene una definición dirigida por la sintaxis que no es una definición con atributos por la izquierda, así que las traducciones basadas en ella no se pueden realizar durante el análisis sintáctico.

Una solución a este problema es reestructurar la gramática para incluir el tipo como el último elemento de la lista de identificadores:

$D \rightarrow$ **id** L
$L \rightarrow$ **,** **id** $L \mid$ **:** T
$T \rightarrow$ **integer** $\mid$ **char**

Ahora, el tipo puede ser considerado como un atributo sintetizado *L.tipo*. Conforme cada identificador es generado por L, su tipo se puede introducir en la tabla de símbolos.

Una definición dirigida por la sintaxis difícil

El algoritmo 5.3 para implantar atributos heredados durante el análisis sintáctico ascendente se prolonga a algunas, pero no a todas, las gramáticas LR. La definición con atributos por la izquierda de la figura 5.38 se basa en una gramática LR(1) sencilla pero no se puede implantar durante el análisis sintáctico LR. El no terminal L en $L \rightarrow \epsilon$ hereda la cuenta del número de símbolos 1 generados por S. Como la producción $L \rightarrow \epsilon$ es la primera por la que reduciría un analizador sintáctico ascendente, el traductor todavía no puede conocer el número de símbolos 1 de la entrada.

PRODUCCIÓN	REGLAS SEMÁNTICAS
$S \rightarrow L$	$L.cuenta := 0$
$L \rightarrow L_1$ 1	$L_1.cuenta := L.cuenta + 1$
$L \rightarrow \epsilon$	$print(L.cuenta)$

Fig. 5.38. Difícil definición dirigida por la sintaxis.

5.7 EVALUADORES RECURSIVOS

Se pueden construir las funciones recursivas que evalúan los atributos conforme recorren un árbol de análisis sintáctico a partir de una definición dirigida por la sintaxis usando una generalización de las técnicas para la traducción predictiva de la sección 5.5. Dichas funciones permiten implantar definiciones dirigidas por la sintaxis que no pueden implantarse a la vez que el análisis sintáctico. En esta sección se asocia una función de traducción simple con cada no terminal. La función visita a los hijos de un nodo para el no terminal en un orden determinado por la producción en el nodo; no es necesario que los hijos sean visitados en orden de izquierda a derecha. En la sección 5.10, se verá cómo lograr el efecto de traducción durante más de una pasada asociando múltiples procedimientos con no terminales.

Recorridos de izquierda a derecha

En el algoritmo 5.2 se vio cómo se puede implantar una definicion con atributos por la izquierda basada en una gramática LL(1) mediante la construcción de una función recursiva que analice sintácticamente y traduzca cada no terminal. Se pueden implantar todas las definiciones dirigidas por la sintaxis con atributos por la izquierda si se invoca una función recursiva similar en un nodo para ese no terminal

en un árbol de análisis sintáctico construido previamente. Observando la producción en el nodo, la función puede determinar cuáles son sus hijos. La función para un no terminal A toma como argumentos un nodo y los valores de los atributos heredados para A, y devuelve como resultados los valores de los atributos sintetizados para A.

Los detalles de la construcción son exactamente como en el algoritmo 5.2, excepto el paso 2, donde la función para un no terminal decide qué producción se debe utilizar basada en el símbolo en curso de entrada. Aquí, la función emplea una proposición **case** para determinar la producción utilizada en un nodo. Se da un ejemplo para ilustrar el método.

Ejemplo 5.20. Considérese la definición dirigida por la sintaxis para determinar el tamaño y la altura de fórmulas en la figura 5.22. El no terminal C tiene un atributo heredado *tp* y un atributo sintetizado *al*. Utilizando el algoritmo 5.2, modificado de la forma que se mencionó anteriormente, se construye la función para C que se mostrará en la figura 5.39.

La función C toma como argumentos un nodo n y un valor correspondiente a $C.tp$ en el nodo, y devuelve un valor correspondiente a $C.al$ en el nodo n. La función tiene un caso para cada producción con C a la izquierda. El código correspondiente a cada producción simula las reglas semánticas asociadas con la producción. El orden en que se aplican las reglas debe ser tal que los atributos heredados de un no terminal se calculen antes de que se llame la función para el no terminal.

```
function C(n, tp)
    var tp1, tp2, al1, al2;
begin
    case producción en el nodo n of
    'C → C1 C2':
        tp1 := tp;
        al1 := C(hijo(n, 1), tp1);
        tp2 := tp;
        al2 := C(hijo(n, 2), tp2);
        return máx(al1, al2);
    'C → C1 sub C2':
        tp1 := tp;
        al1 := C(hijo(n, 1), tp1);
        tp2 := contrae(tp);
        al2 := C(hijo(n, 3), tp2);
        return desp(al1, al2);
    'C → texto':
        return tp × texto.a;
    default:
        error
    end
end;
```

Fig. 5.39. Función para el no terminal C de la figura 5.22.

En el código correspondiente a la producción $C \rightarrow C$ **sub** C, las variables *tp*, *tp1* y *tp2* guardan los valores de los atributos heredados $C.tp$, $C_1.tp$ y $C_2.tp$. De manera similar *al*, *al1* y *al2* guardan los valores de $C.al$, $C_1.al$ y $C_2.al$. Se utiliza la función *hijo*(m, i) para referirse al i-ésimo hijo del nodo m. Como C_2 es la etiqueta del tercer hijo del nodo n, el valor de $C_2.al$ viene determinado por la llamada a la función C(*hijo*(n, 3), *tp2*). □

Otros recorridos

Cuando se tiene disponible un árbol de análisis sintáctico explícito, se puede visitar a los hijos de un nodo en cualquier orden. Considérese la definición del ejemplo 5.21, que no es una definición con atributos por la izquierda. En una traducción especificada por esta definición, se deben visitar los hijos de un nodo para una producción de izquierda a derecha, mientras que los hijos de un nodo para la otra producción se deben visitar de derecha a izquierda.

Este ejemplo abstracto ilustra la eficacia de utilizar funciones mutuamente recursivas para evaluar los atributos en los nodos de un árbol de análisis sintáctico. No es necesario que las funciones dependan del orden en que son creados los nodos del árbol. La principal consideración para evaluar durante un recorrido es que los atributos heredados en un nodo se calculen antes de que el nodo sea visitado por primera vez y que los atributos sintetizados se calculen antes de abandonar el nodo por última vez.

Ejemplo 5.21. Cada uno de los no terminales de la figura 5.40 tiene un atributo heredado h y un atributo sintetizado s. También se muestran los grafos de dependencias para las dos producciones. Las reglas asociadas con $A \rightarrow L\ M$ imponen depen-

PRODUCCIÓN	REGLAS SEMÁNTICAS
$A \rightarrow L\ M$	$L.h := l(A.h)$ $M.h := m(L.s)$ $A.s := f(M.s)$
$A \rightarrow Q\ R$	$R.h. := r(A.h)$ $Q.h := q(R.s)$ $A.s := f(Q.s)$

Fig. 5.40. Producciones y reglas semánticas para el no terminal A.

```
function A(n, ah);
begin
    case producción en el nodo n of
    'A → L M':          /* orden de izquierda a derecha */
        lh := l(ah);
        ls := L(hijo(n, 1), lh);
        mh := m(ls);
        ms := M(hijo(n, 2), mh)
        return f(ms);
    'A → Q R':          /* orden de derecha a izquierda */
        rh := r(ah);
        rs := R(hijo(n, 2), rh);
        qh := q(rs);
        qs := Q(hijo(n, 1), qh);
        return f(qs);
    default:
        error
    end
end;
```

Fig. 5.41. Las dependencias de la figura 5.40 determinan el orden en que se visitan los hijos.

dencias de izquierda a derecha y las reglas asociadas con $A \rightarrow Q\,R$ imponen dependencias de derecha a izquierda.

En la figura 5.41 se muestra la función para el no terminal A; se supone que se pueden construir las funciones para L, M, Q y R. Las variables de la figura 5.41 se nombran según los no terminales y sus atributos; por ejemplo, *lh* y *ls* son las variables correspondientes a $L.h$ y $L.s$.

El código correspondiente a la producción $A \rightarrow L\,M$ se construye como en el ejemplo 5.20. Es decir, se determina el atributo heredado de L, se llama la función para L para que determine el atributo sintetizado de L, y se repite el proceso para M. El código correspondiente a $A \rightarrow Q\,R$ visita el subárbol para R antes de visitar el subárbol para Q. En los demás casos, el código para las dos producciones es muy similar. □

5.8 CONSIDERACIONES DE ESPACIO PARA VALORES DE ATRIBUTOS EN EL MOMENTO DE LA COMPILACION

En esta sección se considera la asignación de espacio en el momento de la compilación para los valores de atributos. Se utilizará información del grafo de dependencias para un árbol de análisis sintáctico, de modo que el enfoque de esta sección es adecuado para métodos basados en árboles de análisis sintáctico que determinen el orden de evaluación a partir del grafo de dependencias. En la siguiente sección se

considera el caso en que se puede predecir el orden de evaluación para, cuando se construye el compilador, poder decidir de una vez por todas el espacio para los atributos.

Dado un orden de evaluación para los atributos no necesariamente en profundidad, la *duración* de un atributo comienza cuando se calcula por primera vez el atributo y termina cuando han sido calculados todos los atributos que dependen de él. Se puede ahorrar espacio conservando el valor de un atributo sólo durante su duración.

Para insistir en que las técnicas de esta sección sirven para cualquier orden de evaluación, se considerará la siguiente definición dirigida por la sintaxis, que no es una definición con atributos por la izquierda, para pasar la información de tipos a los identificadores en una declaración.

Ejemplo 5.22. La definición dirigida por la sintaxis de la figura 5.42 es una prolongación de la de la figura 5.4 para permitir declaraciones de la forma

```
real  c[12][31];                (5.9)
int   x[3], y[5];               (5.10)
```

En la figura 5.43(a) se muestra un árbol de análisis sintáctico para (5.10) mediante líneas con puntos. Los números que aparecen en los nodos se aplican en el siguiente ejemplo. Como en el ejemplo 5.3, el tipo que se obtiene de T es heredado por L y trasladados hacia los identificadores de la declaración. Una arista desde *T.tipo* a *L.her* muestra que *L.her* depende de *T.tipo*. La definición dirigida por la sintaxis de la figura 5.42 no es una definición con atributos por la izquierda porque I_1.*her* depende de **núm**.*val* y **núm** está a la derecha de I_1 en $I \rightarrow I_1$ [**núm**]. □

Asignación de espacio para los atributos en el momento de la compilación

Supóngase que se tiene una secuencia de registros para guardar valores de atributos. Por conveniencia, se supone que cada registro puede guardar cualquier valor de atributo. Si los atributos representan tipos distintos, entonces se pueden formar grupos de atributos que tomen la misma cantidad de almacenamiento y considerar cada grupo por separado. Se confía en que la información acerca de las duraciones de los atributos determine los registros en que se evalúan.

Ejemplo 5.23. Supóngase que los atributos se evalúan en el orden dado por los números de los nodos en el grafo de dependencias de la figura 5.43[3], construido en el último ejemplo. La duración de cada nodo comienza cuando se evalúa su atributo y termina cuando se utiliza su atributo por última vez. Por ejemplo, la duración del nodo 1 termina cuando 2 se evalúa porque 2 es el único nodo que depende de 1. La duración de 2 termina cuando se evalúa 6. □

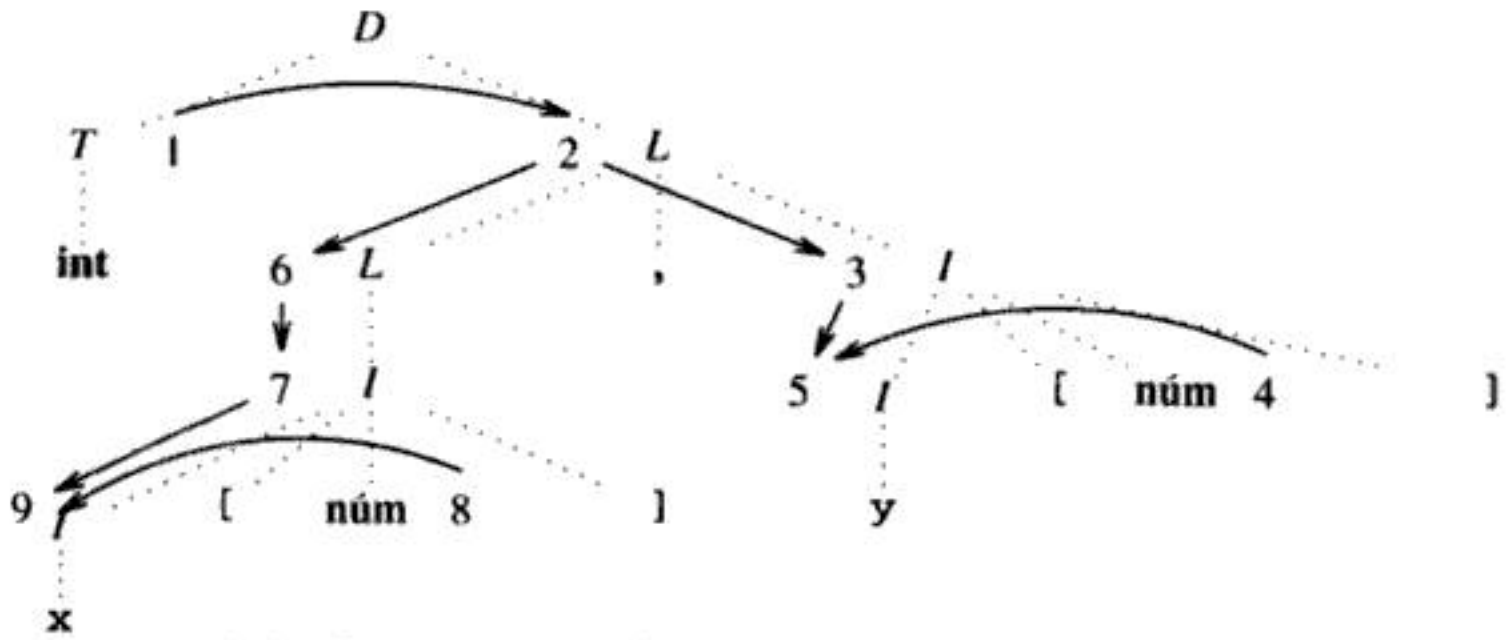
---

[3] El grafo de dependencias de la figura 5.43 no muestra los nodos correspondientes a la regla semántica *añadetipo*(**id**.*entrada*, *I.her*) porque no se asigna espacio a los falsos atributos. Obsérvese, sin embargo, que esta regla semántica no se debe evaluar hasta después de que esté disponible el valor de *I.her*. Un algoritmo para determinar este hecho debe trabajar con un grafo de dependencias que contenga nodos para esta regla semántica.

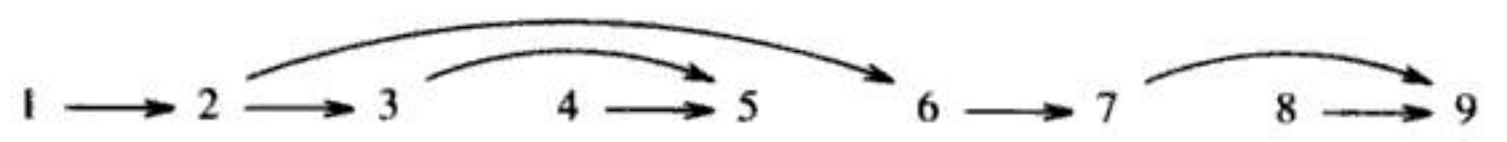
PRODUCCIÓN	REGLAS
$D \rightarrow T\ L$	$L.her := T.tipo$
$T \rightarrow$ **int**	$T.tipo := integer$
$T \rightarrow$ **real**	$T.tipo := real$
$L \rightarrow L_1\ ,\ I$	$L_1.her := L.her$ $I.her := L.her$
$L \rightarrow I$	$I.her := L.her$
$I \rightarrow I_1$ [**núm**]	$I_1.her := array($**núm**$.val, I.her)$
$I \rightarrow$ **id**	$añadetipo($**id**$.entrada, I.her)$

Fig. 5.42. Paso del tipo a los identificadores en una declaración.

En la figura 5.44 se da un método para evaluar atributos que utiliza el menor número posible de registros. Se consideran los nodos del grafo de dependencias D para un árbol de análisis sintáctico en el orden en que deben evaluarse. Inicialmente, se tiene un conjunto de registros $r_1, r_2, \ldots$. Si el atributo b se define para la regla semántica $b := f(c_1, c_2, \ldots, c_k)$, entonces la duración de uno o más de $c_1, c_2, \ldots, c_k$ debe terminar con la evaluación de b; los registros que guardan dichos



(a) Grafo de dependencias para un árbol de análisis sintáctico



(b) Nodos en el orden de evaluación (a)

Fig. 5.43. Determinación de las duraciones de los valores de atributos.

```
for cada nodo m en m1, m2, . . . , mN do begin
    for cada nodo n cuya duración termine con la evaluación de m do
        marcar el registro de n;
    if algún registro r está marcado then begin
        desmarcar r;
        evaluar m en el registro r;
        devolver los registros marcados al conjunto de disponibles
    end
    else /* no se marcaron registros */
        evaluar m en un registro del conjunto de disponibles;
    /* las acciones que usen el valor de m se pueden insertar aquí */
    if la duración de m ha terminado then
        devolver los registros de m al conjunto de disponibles
end
```

Fig. 5.44. Asignación de valores de atributos a registros.

atributos se devuelven después de evaluar b. Siempre que sea posible, b se evalúa en un registro que guarde uno de $c_1, c_2, \ldots, c_k$.

En la figura 5.45 se muestran los registros utilizados durante una evaluación del grafo de dependencias de la figura 5.43. Se comienza por evaluar el nodo 1 dentro del registro r_1. La duración del nodo 1 termina cuando se evalúa 2, de modo que 2 se evalúa dentro de r_1. El nodo 3 recibe un registro nuevo r_2, porque el nodo 6 necesitará el valor de 2.

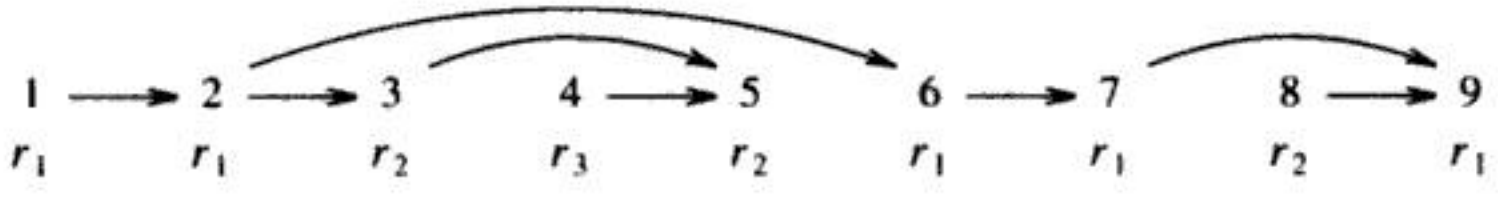


Fig. 5.45. Registros utilizados por los valores de atributos de la figura 5.43.

Evitar las copias

Se puede mejorar el método de la figura 5.44 considerando las reglas de copia como un caso especial. Una regla de copia tiene la forma $b := c$, de modo que si el valor de c está en el registro r, entonces el valor de b aparece ya dentro del registro r. El número de atributos definido por las reglas de copia puede ser significativo, así que es mejor evitar hacer copias explícitas.

Un conjunto de nodos que tengan el mismo valor forma una clase de equivalencia. Se puede modificar el método de la figura 5.44 tal como se indica a continuación para guardar el valor de una clase de equivalencia en un registro. Cuando se considera el nodo m, primero se comprueba si está definido por una regla de copia.

Si así es, entonces su valor debe estar ya en un registro, y m se incorpora a la clase de equivalencia con valores en dicho registro. Es más, un registro es devuelto al conjunto de registros disponibles sólo al final de las duraciones de todos los nodos con valores en el registro.

Ejemplo 5.24. El grafo de dependencias de la figura 5.43 se vuelve a dibujar en la figura 5.46, con un signo igual antes de cada nodo definido por una regla de copia. A partir de la definición dirigida por la sintaxis de la figura 5.42, se ve que el tipo determinado en el nodo 1 se copia a cada elemento de la lista de identificadores, dando como resultado que los nodos 2, 3, 6 y 7 de la figura 5.43 sean copias de 1.

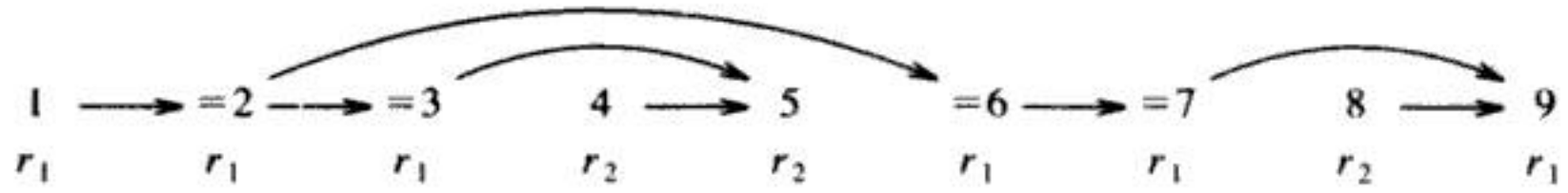


Fig. 5.46. Registros utilizados, tomando en cuenta las reglas de copia.

Como 2 y 3 son copias de 1, sus valores se toman del registro r_1 de la figura 5.46. Obsérvese que la duración de 3 termina cuando se evalúa 5, pero el registro r_1 que guarda el valor de 3 no se devuelve al conjunto porque no ha terminado la duración de 2 en su clase de equivalencia.

El siguiente código muestra cómo la declaración (5.10) del ejemplo 5.22 puede ser procesada por un compilador:

```
r1 := integer;                  /* evalúa los nodos 1, 2, 3, 6, 7 */
r2 := 5;                        /* evalúa el nodo 4 */
r2 := array(r2, r1);            /* tipo de y */
añadetipo(y, r2);
r2 := 3;                        /* evalúa el nodo 8 */
r2 := array(r2, r1);            /* tipo de x */
añadetipo(x, r2);
```

En el código anterior, x e y apuntan a las entradas en la tabla de símbolos para x e y, y el procedimiento *añadetipo* se debe llamar en los momentos adecuados para que añada los tipos de x e y a sus entradas en la tabla de símbolos. □

5.9 ASIGNACION DE ESPACIO EN EL MOMENTO DE LA CONSTRUCCION DEL COMPILADOR

Aunque es posible guardar los valores de todos los atributos en una sola pila durante un recorrido, a veces se puede evitar hacer copias utilizando múltiples pilas. En general, si las dependencias entre los atributos hacen que no sea conveniente poner los valores de algunos atributos en una pila, se pueden guardar en los nodos de un árbol sintáctico construido explícitamente.

En las secciones 5.3 y 5.6 se ha visto el uso de una pila para guardar los valores de atributos durante el análisis sintáctico ascendente. Un analizador sintáctico por descenso recursivo también utiliza implícitamente una pila para seguir las llamadas a procedimientos; se estudiará este aspecto en el capítulo 7.

El uso de una pila se puede combinar con otras técnicas para ahorrar espacio. Las acciones de impresión (*print*) ampliamente utilizadas en los esquemas de traducción del capítulo 2 emiten siempre que es posible atributos con valores de cadenas a un archivo de salida. Mientras se construían los árboles sintácticos en la sección 5.2, se pasaban apuntadores a los nodos en lugar de subárboles completos. En general, en vez de pasar objetos grandes, se puede ahorrar espacio pasando apuntadores. Estas técnicas se aplicarán en los ejemplos 5.27 y 5.28.

Predicción de duraciones a partir de la gramática

Cuando se obtiene el orden de evaluación para los atributos a partir de un determinado recorrido del árbol de análisis sintáctico, se pueden predecir las duraciones de los atributos en el momento de la construcción del compilador. Por ejemplo, supóngase que los hijos se visitan de izquierda a derecha durante un recorrido en profundidad, como en la sección 5.4. Comenzando en un nodo para la producción $A \rightarrow B\ C$, se visita el subárbol para B, se visita el subárbol para C y después se regresa al nodo para A. El padre de A no puede referirse a los atributos de B y C, de modo que sus duraciones deben terminar cuando se regresa a A. Véase que dichas observaciones se basan en la producción $A \rightarrow B\ C$ y en el orden en que se visitan los nodos para dichos no terminales. No es necesario conocer los subárboles en B y C.

Con cualquier orden de evaluación, si la duración del atributo c está contenida en la de b, entonces el valor de c se puede guardar en una pila por encima del valor de b. En este caso, b y c no tienen por qué ser atributos del mismo no terminal. Para la producción $A \rightarrow B\ C$, se puede utilizar una pila durante un recorrido en profundidad de la siguiente forma.

Empiécese en el nodo para A con los atributos heredados de A presentes ya en la pila. Después evalúense y métanse en la pila los valores de los atributos heredados de B. Estos atributos permanecen en la pila conforme se recorre el subárbol de B, regresando con los atributos sintetizados de B por encima de ellos. Este proceso se repite con C; es decir, se meten en la pila sus atributos heredados, se recorre su subárbol y se regresa con sus atributos sintetizados en el tope. Si se escribe $\mathbf{H}(X)$ y $\mathbf{S}(X)$ para los atributos heredados y sintetizados de X, respectivamente, la pila contiene

$$\mathbf{H}(A),\ \mathbf{H}(B),\ \mathbf{S}(B),\ \mathbf{H}(C),\ \mathbf{S}(C) \tag{5.11}$$

Ahora ya están en la pila los valores de los atributos necesarios para calcular los atributos sintetizados de A, así que se puede regresar a A cuando la pila contiene

$$\mathbf{H}(A),\ \mathbf{S}(A)$$

Obsérvese que el número (y presumiblemente el tamaño) de los atributos heredados y sintetizados de un símbolo gramatical es fijo. Por tanto, en cada paso del proceso anterior se sabe hasta qué profundidad de la pila hay que llegar para encontrar un atributo.

Ejemplo 5.25. Supóngase que los valores de los atributos para la traducción de tipografía de la figura 5.22 se guardan en una pila como se explicó anteriormente. Comenzando en un nodo para la producción $C \rightarrow C_1\ C_2$ con $C.tp$ en el tope de la pila, los contenidos de la pila antes y después de visitar un nodo se muestran en la figura 5.47 a la izquierda y a la derecha del nodo, respectivamente. Como de costumbre las pilas crecen hacia abajo.

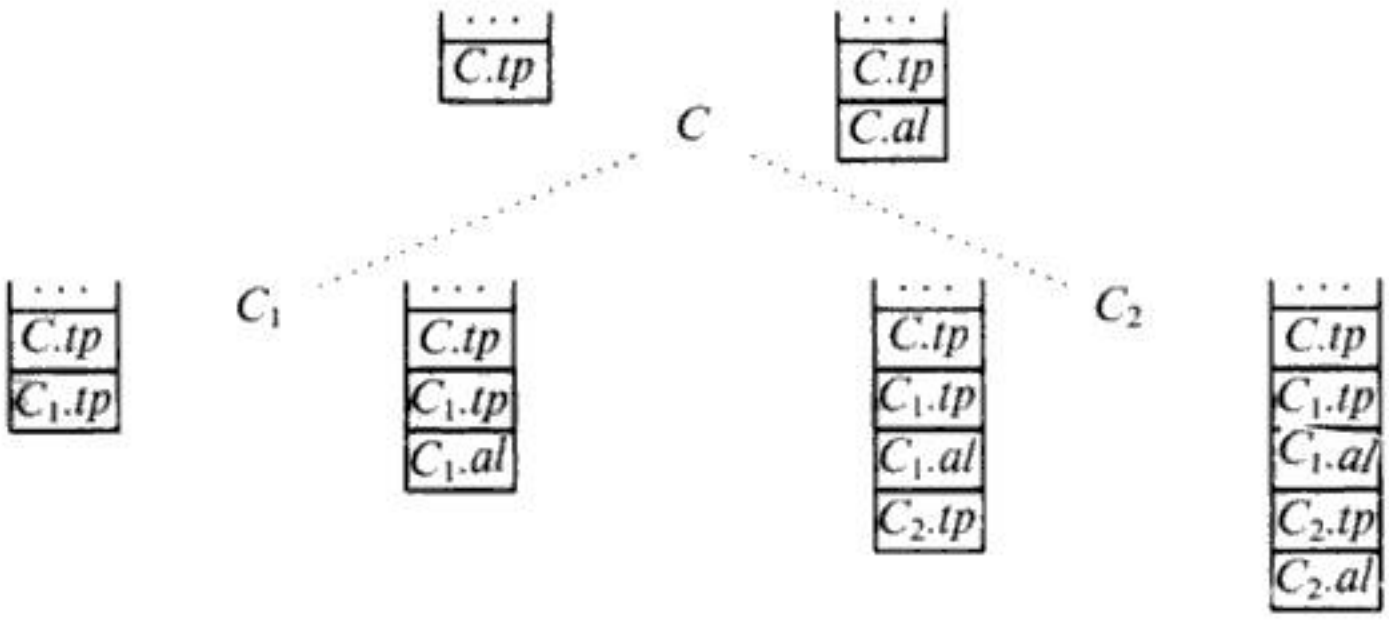


Fig. 5.47. Contenidos de la pila antes y después de visitar un nodo.

Obsérvese que justo antes de que sea visitado por primera vez un nodo para el no terminal C, su atributo tp está en el tope de la pila. Justo después de la última visita, es decir, cuando el recorrido se mueve hacia arriba desde ese nodo, sus atributos al y tp están en las dos posiciones del tope de la pila. □

Cuando un atributo b está definido por una regla de copia $b := c$ y el valor de c está en el tope de la pila de valores de atributo, puede no ser necesario meter una copia de c en la pila. Puede haber más oportunidades para eliminar reglas de copia si se utiliza más de una pila para guardar valores de atributos. En el siguiente ejemplo, se utilizan pilas diferentes para los atributos sintetizados y heredados. Si se compara con el ejemplo 5.25 se ve que se pueden eliminar más reglas de copia si se utilizan pilas diferentes.

Ejemplo 5.26. Con la definición dirigida por la sintaxis de la figura 5.22, supóngase que se utilizan pilas diferentes para el atributo heredado tp y el atributo sintetizado al. Se mantienen las pilas de modo que $C.tp$ está en el tope de la pila de tp justo antes que C sea visitado por primera vez y justo después de que se visite C por última vez. $C.al$ estará en el tope de la pila al justo después que se visite C.

Con pilas diferentes se pueden aprovechar las reglas de copia $C_1.tp := C.tp$ y $C_2.tp := C.tp$ asociadas con $C \rightarrow C_1\ C_2$. Como muestra la figura 5.48, no es necesario meter $C_1.tp$ porque su valor ya está en el tope de la pila como $C.tp$.

En la figura 5.49 se muestra un esquema de traducción basado en la definición dirigida por la sintaxis de la figura 5.22. La operación $mete(v, p)$ mete el valor v en

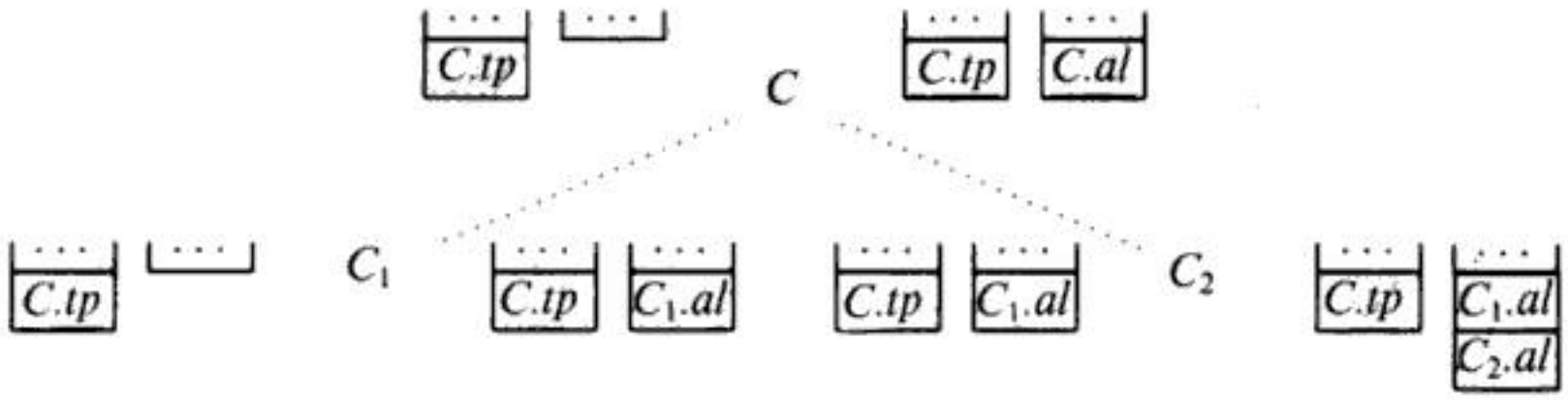


Fig. 5.48. Uso de pilas independientes para los atributos *tp* y *al*.

la pila *p* y *saca*(*p*) saca el valor del tope de la pila *p*. Se utiliza *tope*(*p*) para referirse al elemento del tope de la pila p. □

El siguiente ejemplo combina el uso de una pila para valores de atributos con acciones para emitir código.

$S \rightarrow$	{ *mete*(10, *tp*) }
C	
$C \rightarrow C_1$	
C_2	{ *a*2 := *tope*(*al*); *saca*(*al*);
	*a*1 := *tope*(*al*); *saca*(*al*);
	mete(*máx*(*a*1, *a*2), *al*) }
$C \rightarrow C_1$	
sub	{ *mete*(*contrae*(*tope*(*tp*)), *tp*) }
C_2	{ *saca*(*tp*);
	*a*2 := *tope*(*al*); *saca*(*al*);
	*a*1 := *tope*(*al*); *saca*(*al*);
	mete(*desp*(*a*1, *a*2), *al*) }
$C \rightarrow$ **texto**	{ *mete*(**texto**.*a* × *tope*(*tp*), *al*) }

Fig. 5.49. Esquema de traducción manteniendo las pilas *tp* y *al*.

Ejemplo 5.27. Ahora se consideran algunas técnicas para implantar una definición dirigida por la sintaxis especificando la generación de código intermedio. El valor de una expresión booleana *E* **and** *F* es falsa si *E* es falsa. En C, la subexpresión *F* no se debe evaluar si *E* es falsa. En la sección 8.4 se considera la evaluación de dichas expresiones booleanas.

Las expresiones booleanas en la definición dirigida por sintaxis de la figura 5.50 se construyen a partir de identificadores y el operador **and**. Cada expresión *E* hereda dos etiquetas *E.verdadero* y *E.falso* que marcan los puntos a donde debe saltar el control si *E* es verdadera y falsa, respectivamente.

Supóngase que $E \rightarrow E_1$ **and** E_2. Si la evaluación de E_1 es falsa, entonces el control fluye a la etiqueta heredada *E.falso*; de lo contrario, la evaluación de E_1 da verdadero, así que el control fluye al código para evaluar E_2. Una nueva etiqueta generada por la función *etiqnueva* marca el principio del código para E_2. Las

PRODUCCIÓN	REGLAS SEMÁNTICAS
$E \rightarrow E_1$ **and** E_2	$E_1.verdadero := etiqnueva$ $E_1.falso := E.falso$ $E_2.verdadero := E.verdadero$ $E_2.falso := E.falso$ $E.código := E_1.código \parallel genera(\text{'}\textbf{etiqueta}\text{'}\ E_1.verdadero) \parallel E_2.código$
$E \rightarrow$ **id**	$E.código := genera(\text{'}\textbf{if}\text{'}\ \textbf{id}.lugar\ \text{'}\textbf{goto}\text{'}\ E.verdadero) \parallel genera(\text{'}\textbf{goto}\text{'}\ E.falso)$

Fig. 5.50. Evaluación en «cortocircuito» de expresiones booleanas.

instrucciones individuales se forman utilizando *genera*. Para un mayor análisis de la relevancia de la figura 5.50 para la generación de código intermedio véase la sección 8.4.

La definición dirigida por la sintaxis de la figura 5.50 es una definición con atributos por la izquierda, así que se puede construir un esquema de traducción para ella. El esquema de traducción de la figura 5.51 utiliza al procedimiento *emite* pa-

$E \rightarrow$ { $E_1.verdadero := etiqnueva$;
$E_1.falso := E.falso$ }
E_1
and { $emite(\text{'}\textbf{etiqueta}\text{'}\ E_1.verdadero)$;
$E_2.verdadero := E.verdadero$;
$E_2.falso := E.falso$ }
E_2

$E \rightarrow$ **id** { $emite(\text{'}\textbf{if}\text{'}\ \textbf{id}.lugar\ \text{'}\textbf{goto}\text{'}\ E.verdadero)$;
$emite(\text{'}\textbf{goto}\text{'}\ E.falso)$ }

Fig. 5.51. Emisión de código para expresiones booleanas.

ra generar y emitir instrucciones incrementalmente. En la figura también se muestran las acciones para asignar los valores de los atributos heredados, insertados antes de los símbolos gramaticales apropiados, como se explicó en la sección 5.4.

El esquema de traducción de la figura 5.52 va más allá; utiliza pilas distintas para guardar los valores de los atributos heredados *E.verdadero* y *E.falso*. Al igual que en el ejemplo 5.26, las reglas de copia no tienen efecto sobre las pilas. Para implantar la regla E_1.*verdadero* := *etiqnueva*, se mete una etiqueta nueva dentro de la pila de

$E \rightarrow$ { *mete*(*etiqnueva*, *verdadero*) }
E_1
and { *emite*('**etiqueta**' *tope*(*verdadero*));
saca(*verdadero*) }
E_2

$E \rightarrow$ **id** { *emite*('**if**' **id**.*lugar* '**goto**' *tope*(*verdadero*));
emite('**goto**' *tope*(*falso*)) }

Fig. 5.52. Emisión de código para expresiones booleanas.

verdadero antes de que se visite E_1. La duración de esta etiqueta finaliza con la acción *emite*('**etiqueta**' *tope*(*verdadero*)), correspondiente a *emite*('**etiqueta**' E_1.*verdadero*), de modo que a la pila de verdadero se le aplica la operación *saca* después de la acción. La pila de falso no cambia en este ejemplo, pero es necesaria cuando se admite el operador **or** además del operador **and**. □

Duraciones sin solapamiento

Un registro sólo es un caso especial de una pila. Si cada operación *mete* va seguida de una operación *saca*, entonces no puede haber más de un elemento a la vez en la pila. En este caso se puede utilizar un registro en lugar de una pila. En cuanto a las duraciones, si las duraciones de dos atributos no se solapan, se pueden guardar sus valores en el mismo registro.

Ejemplo 5.28. La definición dirigida por la sintaxis de la figura 5.53 construye árboles sintácticos para expresiones en forma de listas con operadores en un solo nivel de precedencia. Dicha definición se ha tomado del esquema de traducción de la figura 5.28.

La duración de cada atributo R debería finalizar cuando sea evaluado el atributo que depende de él. Se puede demostrar que para cualquier árbol de análisis sintáctico, los atributos de R pueden evaluarse dentro del mismo registro r. El siguiente razonamiento es el típico necesario para analizar gramáticas. La inducción se realiza sobre el tamaño del subárbol asociado a R en el fragmento de árbol de análisis sintáctico de la figura 5.54.

Se obtiene el árbol más pequeño posible si se aplica $R \rightarrow \epsilon$, en cuyo caso $R.s$ es una copia de $R.h$, así que el valor de ambos está en r. Para un subárbol más grande, la producción en la raíz del subárbol debe ser para $R \rightarrow$ **opsuma** $T\ R_1$. La duración de $R.h$ termina cuando se evalúa $R_1.h$, de modo que $R_1.h$ se puede evaluar dentro del registro r. Según la hipótesis de inducción, a todos los atributos para casos del

PRODUCCIÓN	REGLAS SEMÁNTICAS
$E \rightarrow T\ R$	$R.h := T.apn$ $E.apn := R.s$
$R \rightarrow$ **opsuma** $T\ R_1$	$R_1 := haznodo$ (**opsuma**.*lexema*, $R.h$, $T.apn$) $R.s := R_1.s$
$R \rightarrow \epsilon$	$R.s := R.h$
$T \rightarrow$ **núm**	$T.apn := hazhoja$(**núm**, **núm**.*val*)

Fig. 5.53. Una definición dirigida por la sintaxis adaptada de la figura 5.28.

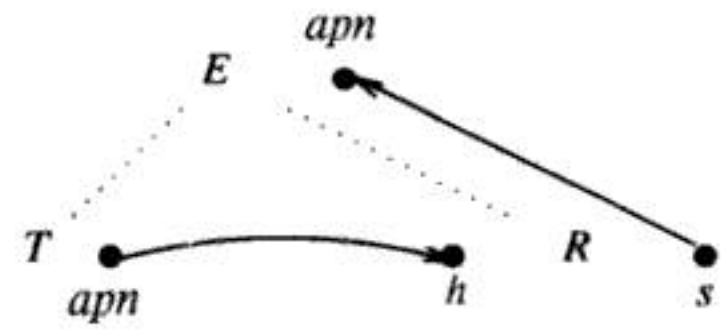


Fig. 5.54. Grafo de dependencias para $E \rightarrow T\ R$.

no terminal R en el subárbol de R_1 se les puede asignar el mismo registro. Por último, $R.s$ es una copia de $R_1.s$, de modo que su valor está ya en r.

El esquema de traducción de la figura 5.55 evalúa los atributos en la gramática con atributos en la figura 5.53, utilizando el registro r para guardar los valores de los atributos $R.h$ y $R.s$ para todos los casos del no terminal R.

```
E → T        { r := T.apn /* r contiene ahora a R.h */ }
    R        { E.apn := r /* se ha devuelto r con R.s */ }
R → opsuma
    T        { r := haznodo(opsuma.lexema, r, T.apn) }
    R
R → ϵ
T → núm      { T.apn := hazhoja(núm, núm.val) }
```

Fig. 5.55. Esquema de traducción transformado para construir árboles sintácticos.

Para terminar, en la figura 5.56 se muestra el código para aplicar el esquema de traducción anterior; se construye según el algoritmo 5.2. El no terminal R ya no tiene atributos, así que R se convierte en un procedimiento en lugar de ser una función.

La variable *r* se hizo local a la función *E*, así que *E* puede ser llamado recursivamente, aunque no es necesario en el esquema de la figura 5.55. Se puede mejorar aún más este código eliminando la recursión por el final y sustituyendo después la llamada restante de *R* por el código del procedimiento resultante, como en la sección 2.5. □

5.10 ANALISIS DE DEFINICIONES DIRIGIDAS POR LA SINTAXIS

En la sección 5.7, los atributos se evaluaron durante el recorrido de un árbol utilizando un conjunto de funciones mutuamente recursivas. La función para un no terminal transformaba los valores de los atributos heredados en un nodo en los valores de los atributos sintetizados en dicho nodo.

```
function E: ↑ nodo_árbol_sintáctico;
    var r: ↑ nodo_árbol_sintáctico;
        lexemaopsuma: char;

    procedure R;
    begin
        if símbolo-anticipación = opsuma then begin
            lexemaopsuma := valex;
            parea(opsuma);
            r := haznodo(lexemaopsuma, r, T);
            R
        end
    end;

begin
    r := T; R
    return r
end;
```

Fig. 5.56. Compárese el procedimiento *R* con el código de la figura 5.31.

El enfoque de la sección 5.7 se extiende a traducciones que no pueden realizarse durante un solo recorrido en profundidad. Aquí se usará una función distinta para cada atributo sintetizado de cada no terminal, aunque una sola función puede evaluar grupos de atributos sintetizados. La construcción de la sección 5.7 considera el caso especial en que todos los atributos sintetizados forman un grupo. El agrupamiento de atributos viene determinado por las dependencias impuestas por las reglas semánticas en una definición dirigida por la sintaxis. El siguiente ejemplo abstracto ilustra la construcción de un evaluador recursivo.

Ejemplo 5.29. La definición dirigida por la sintaxis de la figura 5.57 está motivada por un problema que se considerará en el capítulo 6. Brevemente, el problema es éste: un identificador "sobrecargado" puede tener un conjunto de tipos posibles; como consecuencia, una expresión puede tener un conjunto de tipos posibles. Se utiliza información del contexto para seleccionar uno de los tipos posibles para cada subexpresión. Puede resolverse el problema realizando una pasada ascendente para sintetizar el conjunto de tipos posibles, seguida de una pasada descendente para reducir el conjunto a un solo tipo.

PRODUCCIÓN	REGLAS SEMÁNTICAS
$S \rightarrow E$	$E.h := g(E.s)$ $S.r := E.t$
$E \rightarrow E_1\ E_2$	$E.s := fs(E_1.s, E_2.s)$ $E_1.h := fh1(E.h)$ $E_2.h := fh2(E.h)$ $E.t := ft(E_1.t, E_2.t)$
$E \rightarrow \mathbf{id}$	$E.s := \mathbf{id}.s$ $E.t := i(E.h)$

Fig. 5.57. Los atributos sintetizados s y t no se pueden evaluar juntos.

Las reglas semánticas de la figura 5.57 son una abstracción de este problema. El atributo sintetizado s representa el conjunto de tipos posibles y el atributo heredado h representa la información del contexto. Un atributo sintetizado adicional t, que no se puede evaluar en la misma pasada que s, puede representar el código generado o el tipo seleccionado para una subexpresión. En la figura 5.58 se muestran los grafos de dependencias para las producciones de la figura 5.57. □

Evaluacion recursiva de atributos

El grafo de dependencias para un árbol de análisis sintáctico se forma uniendo grafos más pequeños correspondientes a las reglas semánticas para una producción. El grafo de dependencias D_p para la producción p se basa únicamente en las reglas se-

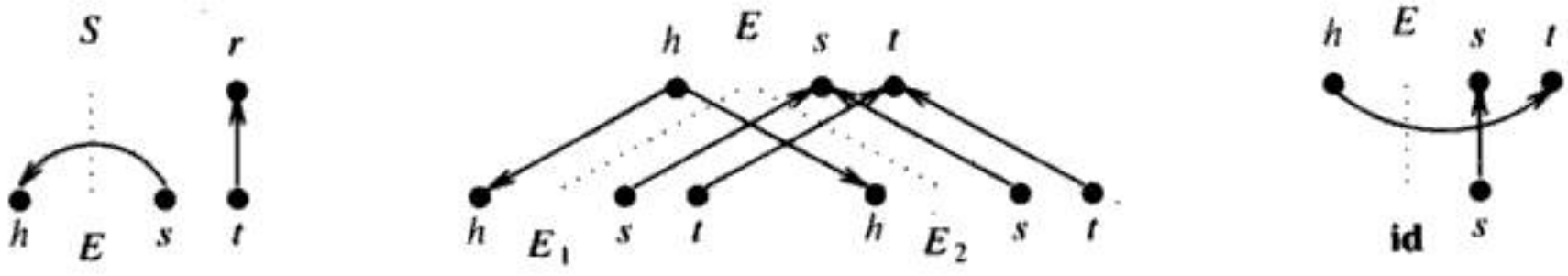


Fig. 5.58. Grafos de dependencias para las producciones de la figura 5.57.

mánticas para una sola producción, es decir, en las reglas semánticas para los atributos sintetizados del lado izquierdo y los atributos heredados de los símbolos gramaticales del lado derecho de la producción. Así, el grafo D_p muestra sólo dependencias locales. Por ejemplo, todas las aristas del grafo de dependencias para $E \rightarrow E_1\ E_2$ de la figura 5.58 están entre casos del mismo atributo. Por este grafo de dependencias, no se puede saber si los atributos s deben calcularse antes que los otros atributos.

Considerando con mayor detalle el grafo de dependencias para el árbol de análisis sintáctico de la figura 5.59, se ve que los atributos de cada caso del no terminal E deben evaluarse en el orden $E.s$, $E.h$, $E.t$. Obsérvese que todos los atributos de la figura 5.59 pueden evaluarse en tres pasadas: una pasada ascendente para evaluar los atributos s, una pasada descendente para evaluar los atributos h y una pasada ascendente final para evaluar los atributos t.

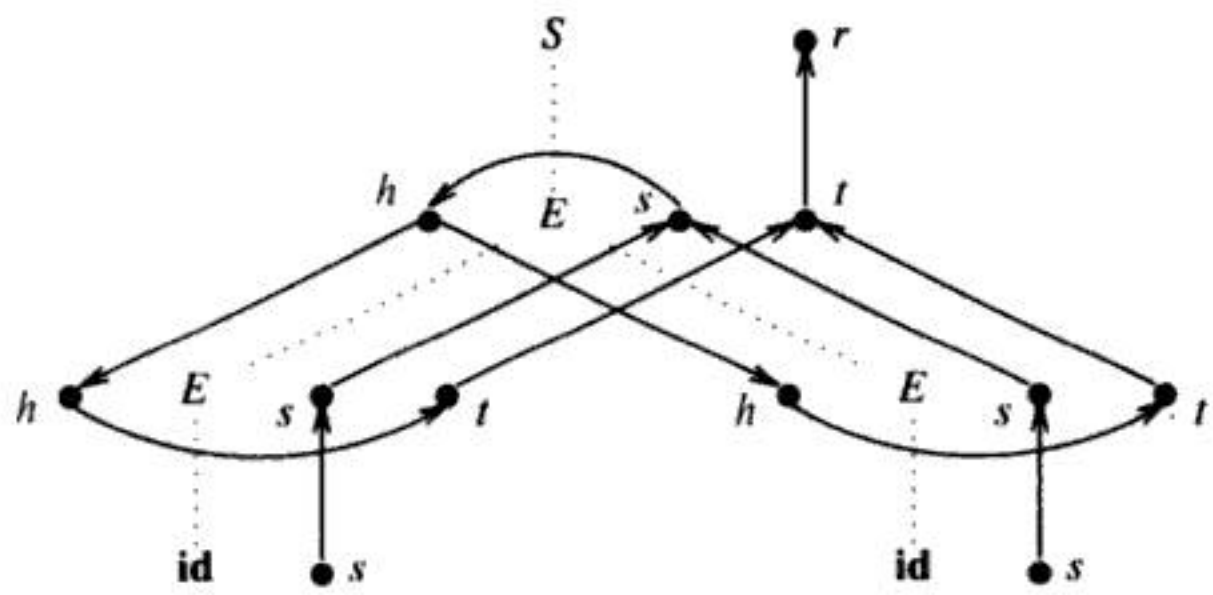


Fig. 5.59. Grafo de dependencias para un árbol de análisis sintáctico.

En un evaluador recursivo, la función para un atributo sintetizado toma los valores de algunos de los atributos heredados como parámetros. En general, si el atributo sintetizado $A.a$ puede depender del atributo heredado $A.b$, entonces la función para $A.a$ toma $A.b$ como parámetro. Antes de analizar las dependencias, se considera un ejemplo que ilustra su uso.

Ejemplo 5.30. Las funciones Es y Et de la figura 5.60 devuelven los valores de los atributos sintetizados s y t en un nodo n etiquetado con E. Como en la sección 5.7, hay un caso para cada producción en la función para un no terminal. El código ejecutado en cada caso simula las reglas semánticas asociadas con la producción de la figura 5.57.

Del estudio anterior del grafo de dependencias de la figura 5.59 se desprende que el atributo $E.t$ en un nodo de un árbol de análisis sintáctico puede depender de $E.h$. Por tanto, se pasa al atributo h como parámetro a la función $E.t$ para el atributo t. Como el atributo $E.s$ no depende de ningún atributo heredado, la función $E.s$ no tiene parámetros correspondientes a valores de atributos. □

Definiciones dirigidas por la sintaxis fuertemente no circulares

Se pueden construir evaluadores recursivos para una clase de definiciones dirigidas por la sintaxis, llamadas definiciones "fuertemente no circulares". Para una definición dentro de esta clase, puede evaluarse según el mismo orden (parcial). Cuando se construye la función para un atributo sintetizado del no terminal, se utiliza este orden para seleccionar los atributos heredados que se convierten en los parámetros de la función.

```
function Es(n);
begin
    case producción en el nodo n of
    'E → E1 E2':
        s1 := Es(hijo(n, 1));
        s2 := Es(hijo(n, 2));
        return fs(s1, s2);
    'E → id';
        return id.s;
    default:
        error
    end
end;

function Et(n, h);
begin
    case producción en el nodo n of
    'E → E1 E2':
        h1 := fh1(h);
        t1 := Et(hijo(n, 1), h1);
        h2 := fh2(h);
        t2 := Et(hijo(n, 2), h2);
        return ft(t1, t2);
    'E → id':
        return i(h);
    default:
        error
    end
end;

function Sr(n);
begin
    s := Es(hijo(n, 1));
    h := g(s);
    t := Et(hijo(n, 1), h);
    return t
end;
```

Fig. 5.60. Funciones para los atributos sintetizados de la figura 5.57.

Se da una definición de esta clase y se demuestra que la definición dirigida por la sintaxis de la figura 5.57 pertenece a dicha clase. Después se da un algoritmo para comprobar si existe circularidad y fuerte no circularidad, y se demuestra cómo la aplicación del ejemplo 5.30 sirve para todas las definiciones fuertemente no circulares.

Considérese el no terminal A en un nodo n de un árbol de análisis sintáctico. El grafo de dependencias para el árbol de análisis sintáctico puede tener generalmente caminos que comiencen en un atributo del nodo n, recorran atributos de otros nodos en el árbol y terminen en otro atributo de n. Para lo que se pretende, basta con considerar los caminos que permanecen dentro de la parte del árbol de análisis sintáctico debajo de A. Una pequeña reflexión revela que dichos caminos van desde algún atributo heredado de A a algún atributo sintetizado de A. Se hace una estimación (quizá demasiado pesimista) del conjunto de dichos caminos considerando órdenes parciales respecto a los atributos de A.

Tenga la producción p los no terminales $A_1, A_2, \ldots, A_n$ en el lado derecho. Sea RA_j un orden parcial respecto a los atributos de A_j, para $1 \leq j \leq n$. Se escribe $D_p[RA_1, RA_2, \ldots, RA_n]$ para el grafo obtenido añadiendo aristas a D_p de la siguiente manera: si RA_j ordena el atributo $A_j.b$ antes del $A_j.c$ entonces añádase una arista de $A_j.b$ a $A_j.c$.

Se dice que una definición dirigida por la sintaxis es *fuertemente no circular* si para cada terminal A se puede encontrar un orden parcial RA respecto a los atributos de A tal que para cada producción p con lado izquierdo A y no terminales $A_1, A_2, \ldots, A_n$ en el lado derecho,

1. $D_p[RA_1, RA_2, \ldots, RA_n]$ es acíclico, y
2. si hay una arista del atributo $A.b$ a $A.c$ en $D_p[RA_1, RA_2, \ldots, RA_n]$, entonces RA ordena $A.b$ antes que $A.c$.

Ejemplo 5.31. Sea p la producción $E \rightarrow E_1\ E_2$ de la figura 5.57, cuyo grafo de dependencias D_p está en el centro de la figura 5.58. Sea RE el orden parcial (orden total en este caso) $s \rightarrow h \rightarrow t$. Existen dos casos de no terminales en el lado derecho de p, que se escriben E_1 y E_2, como de costumbre. Por tanto, RE_1 y RE_2 son los mismos que RE, y el grafo $D_p[RE_1, RE_2]$ se muestra en la figura 5.61.

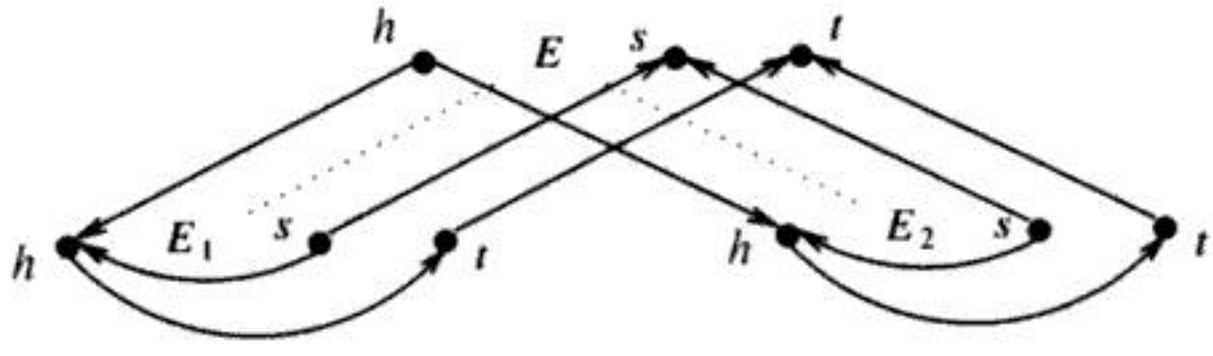


Fig. 5.61. Grafo de dependencias aumentado para una producción.

Entre los atributos asociados con la raíz E de la figura 5.61, los únicos caminos son de h a t. Como RE hace que h preceda a t, no se incumple la condición (2). □

Dada una definición fuertemente no circular y un orden parcial RA para cada no terminal A, la función para el atributo sintetizado s de A toma los argumentos que sigue: si RA ordena el atributo heredado h antes que s, entonces h es un argumento de la función, de lo contrario no.

Una prueba de circularidad

Se dice que una definición dirigida por la sintaxis es circular si el grafo de dependencias para un árbol de análisis sintáctico tiene un ciclo; las definiciones circulares están mal formadas y carecen de significado. Es imposible comenzar a calcular ninguno de los valores de atributos en el ciclo. Calcular los órdenes parciales que garantizan que una definición sea fuertemente no circular tiene mucho que ver con comprobar si una definición es circular. Por tanto, primero se considerará una prueba de la existencia de circularidad.

Ejemplo 5.32. En la siguiente definición dirigida por la sintaxis, los caminos entre los atributos de A dependen de la producción que se aplique. Si se aplica $A \rightarrow 1$, entonces $A.s$ depende de $A.h$; de lo contrario no. Para una información completa sobre las dependencias posibles se deben seguir con atención los conjuntos de órdenes parciales de los atributos de un no terminal.

PRODUCCIÓN	REGLAS SEMÁNTICAS
$S \rightarrow A$	$A.h := c$
$A \rightarrow 1$	$A.s := f(A.h)$
$a \rightarrow 2$	$A.s := d$

□

La idea base del algoritmo de la figura 5.62 es la siguiente: Se representan los órdenes parciales mediante grafos dirigidos acíclicos (GDA). Dados GDA a los atributos de símbolos en el lado derecho de una producción, se puede determinar un GDA para los atributos del lado izquierdo como se ilustra en la figura 5.62.

Sea p la producción $A \rightarrow X_1 X_2 \ldots X_k$ con grafo de dependencias D_p. Sea D_j un GDA para X_j, $1 \leq j \leq k$. Cada arista $b \rightarrow a$ en D_j se añade temporalmente al grafo de dependencias D_p para la producción. Si el grafo obtenido tiene un ciclo, entonces la definición dirigida por la sintaxis es circular. De lo contrario, los caminos en el grafo resultante determinan un nuevo GDA a los atributos del lado izquierdo de la producción, y el GDA obtenido se añade a $\mathcal{F}(A)$.

La prueba de circularidad de la figura 5.62 toma un tiempo exponencial en función del número de grafos en los conjuntos $\mathcal{F}(X)$ para cualquier símbolo gramatical X. Existen definiciones dirigidas por la sintaxis que no admiten una prueba de circularidad en un tiempo polinomial.

Se puede convertir el algoritmo de la figura 5.62 en una prueba más eficiente si una definición dirigida por la sintaxis es fuertemente no circular, de la siguiente manera. En lugar de mantener una familia de grafos $\mathcal{F}(X)$ para cada X, se resume la información de la familia conservando un solo grafo $F(X)$. Obsérvese que cada grafo en $\mathcal{F}(X)$ tiene los mismos nodos para los atributos de X, pero puede tener distintas

```
for el símbolo gramatical X do
    F(x) tiene un solo grafo con los atributos de X y ninguna arista;
repeat
    cambio := false;
    for la producción p dada por A → X1 X2 ... Xk do begin
        for los GDA G1 ∈ F(X1), ..., Gk ∈ F(Xk) do begin
            D := Dp;
            for la arista b → c en Gj, 1 ≤ j ≤ k do
                añadir una arista en D entre los atributos b y c de Xj;
            if D tiene un ciclo then
                falla la prueba de circularidad
            else begin
                G := un nuevo grafo con nodos para los atributos
                de A y ninguna arista;
                for cada par de atributos b y c de A do
                    if hay un camino en D desde b a c then
                        añadir b → c a G;
                if G no está ya en F(A) then begin
                    añadir G a F(A);
                    cambio := true
                end
            end
        end
    end
until cambio = false
```

Fig. 5.62. Una prueba de circularidad.

aristas. $F(X)$ es el grafo de los nodos para los atributos de X que tiene una arista entre $X.b$ y $X.c$ si la tiene un grafo de $\mathscr{F}(X)$. $F(X)$ representa una "estimación del peor caso" de las dependencias entre los atributos de X. Por ejemplo, si $F(X)$ es acíclico, entonces es seguro que la definición dirigida por la sintaxis es no circular. Sin embargo, no siempre ocurre a la inversa; es decir, si $F(X)$ tiene un ciclo, no siempre la definición dirigida por la sintaxis es circular.

La prueba de circularidad modificada construye grafos acíclicos $F(X)$ para cada X si resulta positiva. A partir de dichos grafos se puede construir un evaluador para la definición dirigida por la sintaxis. El método es una clara generalización del ejemplo 5.30. La función para el atributo sintetizado $X.s$ toma como argumentos únicamente todos los atributos heredados, que preceden a s en $F(X)$. La función, llamada en el nodo n, llama a otras funciones que calculan los atributos sintetizados necesarios en los hijos de n. A las rutinas para calcular dichos atributos se les pasan valores para los atributos heredados que necesitan. El hecho de que la prueba de fuerte no circularidad resultara positiva significa que dichos atributos heredados se pueden calcular.

EJERCICIOS

5.1 Para la expresión de entrada `(4*7+1)*2`, constrúyase un árbol de análisis sintáctico con anotaciones según la definición dirigida por la sintaxis de la figura 5.2.

5.2 Constrúyase el árbol de análisis sintáctico y el árbol sintáctico para la expresión `((a)+(b))` según

a) la definición dirigida por la sintaxis de la figura 5.9, y

b) el esquema de traducción de la figura 5.28.

5.3 Constrúyase el GDA e identifíquense los números de valor para las subexpresiones de la siguiente expresión, suponiendo que + asocia por la izquierda:

`a+a+(a+a+a+(a+a+a+a))`.

***5.4** Dése una definición dirigida por la sintaxis para traducir expresiones infijas a expresiones infijas sin paréntesis redundantes. Por ejemplo, puesto que + y * asocian por la izquierda, `((a*(b+c))*(d))` puede reescribirse `a*(b+c)*d`.

5.5 Dése una definición dirigida por la sintaxis para diferenciar expresiones formadas mediante la aplicación de los operadores aritméticos + y * a la variable `x` y constantes; por ejemplo, `x*(3*x+x*x)`. Supóngase que no tiene lugar ninguna simplificación, de modo que `3*x` se traduce a `3*1+0*x`.

5.6 La siguiente gramática genera expresiones formadas mediante la aplicación de un operador aritmético + a constantes enteras y reales. Cuando se suman dos enteros, el tipo obtenido es entero, de lo contrario es real.

$E \rightarrow E + T \mid T$

$T \rightarrow$ **núm . núm** $\mid$ **núm**

a) Dése una definición dirigida por la sintaxis para determinar el tipo de cada subexpresión.

b) Amplíese la definición dirigida por la sintaxis de a) para que traduzca expresiones a notación postfija así como determinar los tipos. Utilícese el operador unitario **entareal** para convertir un valor entero en un valor real equivalente, de manera que ambos operandos de + en la forma postfija tengan el mismo tipo.

5.7 Amplíese la definición dirigida por la sintaxis de la figura 5.22 para tener en cuenta el ancho de las cajas sus alturas correspondientes. Supóngase que el terminal **texto** tiene un atributo sintetizado *a* que proporciona el ancho normalizado del texto.

5.8 Sea el atributo sintetizado *val* que da el valor del número binario generado por S en la siguiente gramática. Por ejemplo, con la entrada `101.101`, $S.val = 5.625$

$S \rightarrow L \ . \ L \mid L$

$L \rightarrow L \ B \mid B$

$B \rightarrow$ `0` $\mid$ `1`

a) Utilícense atributos sintetizados para determinar *S.val*.
b) Determínese *S.val* con una definición dirigida por la sintaxis en la que el único atributo sintetizado de *B* sea *c*, que proporcione la contribución del bit generado por *B* al valor final. Por ejemplo, la contribución del primero y último bits en `101.101` al valor 5.625 es 4 y 0.125, respectivamente.

5.9 Reescríbase la gramática subyacente en la definición dirigida por la sintaxis del ejemplo 5.3 de modo que la información sobre el tipo pueda propagarse utilizando únicamente atributos sintetizados.

***5.10** Cuando las proposiciones generadas por la siguiente gramática se traducen a código de máquina abstracta, una proposición **break** traduce en un salto a la instrucción que va después de la proposición **while** englobadora más cercana. Para simplificar, las expresiones se representan mediante el terminal **expr** y las otras clases de proposiciones con el terminal **otras**. Dichos terminales tienen un atributo sintetizado *código* que hace su traducción.

$S \rightarrow$ **while expr do begin** S **end**
$\mid S\ ;\ S$
$\mid$ **break**
$\mid$ **otras**

Dése una definición dirigida por la sintaxis que traduzca proposiciones a código para la máquina de pila de la sección 2.8. Asegúrese de que se traduzcan correctamente las proposiciones **break** dentro de proposiciones **while** anidadas.

5.11 Elimínese la recursión por la izquierda de las definiciones dirigidas por la sintaxis del ejercicio 5.6 a) y b).

5.12 Las expresiones generadas por la siguiente gramática pueden tener asignaciones dentro de ellas.

$S \rightarrow E$
$E \rightarrow E := E \mid E + E \mid (E) \mid$ **id**

La semántica de las expresiones es como en C. Es decir, `b:=c` es una expresión que asigna el valor de `c` a `b`; el valor de lado derecho de esta expresión es el mismo que el de `c`. Es más, `a:=(b:=c)` asigna el valor de `c` a `b` y después a `a`.

a) Constrúyase una definición dirigida por la sintaxis para comprobar que el lado izquierdo de una expresión sea un valor de lado izquierdo. Utilícese un atributo heredado *lado* del no terminal *E* para indicar si la expresión generada por *E* aparece en el lado izquierdo o en el derecho de una asignación.
b) Amplíese la definición dirigida por la sintaxis de a) para que genere código intermedio para la máquina de pila de la sección 2.8 conforme comprueba la entrada.

5.13 Reescríbase la gramática subyacente del ejercicio 5.12 de manera que agrupe las subexpresiones de := a la derecha y a las subexpresiones de + a la izquierda.

a) Constrúyase un esquema de traducción que simule la definición dirigida por la sintaxis del ejercicio 5.12 b).

b) Modifíquese el esquema de traducción de a) para que emita código incrementalmente a un archivo de salida.

5.14 Dése un esquema de traducción para asegurarse de que el mismo identificador no aparezca dos veces en una lista de identificadores.

5.15 Supónganse las declaraciones generadas por la siguiente gramática

$D \rightarrow \textbf{id}\ L$
$L \rightarrow \textbf{,}\ \textbf{id}\ L \mid \textbf{:}\ T$
$T \rightarrow \textbf{integer} \mid \textbf{real}$

a) Constrúyase un esquema de traducción para introducir el tipo de cada identificador en la tabla de símbolos, como en el ejemplo 5.3.

b) Constrúyase un traductor predictivo a partir del esquema de traducción de a).

5.16 La siguiente gramática es una versión no ambigua de la gramática subyacente de la figura 5.22. Las llaves { } se utilizan sólo para agrupar cajas, y se eliminan durante la traducción.

$S \rightarrow L$
$L \rightarrow L\ B \mid B$
$B \rightarrow B\ \textbf{sub}\ F \mid F$
$F \rightarrow \{\ L\ \} \mid \textbf{texto}$

a) Adáptese la definición dirigida por la sintaxis de la figura 5.22 para que utilice la gramática anterior.

b) Conviértase la definición dirigida por la sintaxis de a) en un esquema de traducción.

***5.17** Amplíese la transformación para eliminar la recursión por la izquierda de la sección 5.5 de modo que permita al no terminal A en (5.2):

a) Atributos heredados definidos por reglas de copia.

b) Atributos heredados.

5.18 Elimínese la recursión por la izquierda del esquema de traducción del ejercicio 5.16(b).

***5.19** Supóngase que se tiene una definición con atributos por la izquierda cuya gramática subyacente es LL(1) o una para la que se pueden resolver ambigüedades y construir un analizador sintáctico predictivo. Demuéstrese que se pueden conservar los atributos heredados y sintetizados en la pila de un analizador sintáctico descendente guiado por la tabla de análisis sintáctico predictivo.

***5.20** Demuéstrese que añadiendo no terminales marcadores únicos en cualquier parte en una gramática LL(1) se obtiene una gramática LR(1).

5.21 Considérese la siguiente modificación de la gramática LR(1), $L \rightarrow Lb \mid a$:

$$L \rightarrow M\ L\ b \mid a$$
$$M \rightarrow \epsilon$$

a) ¿En qué orden aplicaría un analizador sintáctico ascendente las producciones en el árbol de análisis sintáctico para la cadena de entrada *abbb*?

*b) Demuéstrese que la gramática modificada no es LR(1).

***5.22** Demuéstrese que en un esquema de traducción basado en la figura 5.36, el valor del atributo heredado *C.tp* siempre estará inmediatamente debajo del lado derecho cuando se reduzca un lado derecho a *C*.

5.23 El algoritmo 5.3 para el análisis sintáctico ascendente y la traducción con atributos heredados utilizan no terminales marcadores para guardar los valores de atributos heredados en posiciones predecibles de la pila del analizador sintáctico. Pueden ser necesarios menos marcadores si los valores se colocan en una pila independientes de la pila de análisis sintáctico.

a) Conviértase la definición dirigida por la sintaxis de la figura 5.36 en un esquema de traducción.

b) Modifíquese el esquema de traducción construido en a) de modo que el valor del atributo heredado *tp* aparezca en una pila independiente. Elimínese el no terminal marcador *M* en el proceso.

***5.24** Considérese la traducción durante el análisis sintáctico como en el ejercicio 5.23. S. C. Johnson sugiere el siguiente método para simular una pila independiente para los atributos heredados, utilizando marcadores y una variable global para cada atributo heredado. En la siguiente producción, la primera acción mete el valor *v* en la pila *h* y la segunda acción lo saca.

$$A \rightarrow \alpha\ \{\ mete(v, h)\ \}\ \beta\ \{\ saca(h)\ \}$$

La pila *h* se puede simular mediante las siguientes producciones que utilizan una variable global *g* y un no terminal marcador *M* con atributo sintetizado *s*:

$$A \rightarrow \alpha\ M\ \beta\ \{\ g := M.s\ \}$$
$$M \rightarrow \epsilon\ \{\ M.s := g\ ;\ g := v\ \}$$

a) Aplíquese esta transformación al esquema de traducción del ejercicio 5.23(b). Sustitúyanse todas las referencias al tope de la pila independiente por referencias a la variable global.

b) Demuéstrese que el esquema de traducción construido en a) calcula los mismos valores para el atributo sintetizado del símbolo inicial que el del ejercicio 5.23(b).

5.25 Utilícese el enfoque de la sección 5.8 para implantar todos los atributos *E.lado* en el esquema de traducción del ejercicio 5.12(b) mediante una sola variable booleana.

5.26 Modifíquese el uso de la pila durante el recorrido en profundidad del ejemplo 5.26 de modo que los valores en la pila correspondan a los almacenados en la pila del analizador sintáctico del ejemplo 5.19.

NOTAS BIBLIOGRAFICAS

El uso de atributos sintetizados para especificar la traducción de un lenguaje aparece en Irons [1961]. La idea de un analizador sintáctico que llame a las acciones semánticas es estudiada por Samelson y Bauer [1960] y Brooker y Morris [1962]. Junto con los atributos heredados, los grafos de dependencia y una prueba de fuerte no circularidad que aparecen en Knuth [1968] incluye una prueba de circularidad en una corrección a dicho artículo. El ejemplo ampliado en el artículo utiliza efectos colaterales disciplinados sobre los atributos globales asociados a la raíz de un árbol de análisis sintáctico. Si los atributos pueden ser funciones, se pueden eliminar los atributos heredados; como en la semántica denotacional, se puede asociar una función de atributos heredados a atributos sintetizados con un no terminal. Dichas observaciones aparecen en Mayoh [1981].

Una aplicación en la que no se desean efectos colaterales en las reglas semánticas es en la edición dirigida por la sintaxis. Supóngase que se genera un editor a partir de una gramática con atributos para el lenguaje fuente, como en Reps [1984] y considérese una modificación de edición al programa fuente que dé como resultado que se elimine una parte del árbol de análisis sintáctico para el programa. Mientras no haya efectos colaterales, se pueden recalcular incrementalmente los valores de los atributos para el programa modificado.

Ershov [1958] utiliza la dispersión para localizar las subexpresiones comunes.

La definición de gramáticas con atributos por la izquierda hecha por Lewis, Rosenkrantz y Stearns [1974] está motivada por la traducción durante el análisis sintáctico. Existen limitaciones similares en cuanto a las dependencias de los atributos en cada uno de los recorridos en profundidad de izquierda a derecha en Bochmann [1976]. Las gramáticas afijas, tal como fueron introducidas por Koster [1971], están relacionadas con las gramáticas con atributos por la izquierda. Se proponen en Koskimies y Räihä [1983] limitaciones de las gramáticas con atributos por la izquierda para controlar el acceso a atributos globales.

En Bochmann y Ward [1978], se describe la construcción mecánica de un traductor predictivo, similar a los que construye el algoritmo 5.2. La impresión de que el análisis sintáctico descendente permite una mayor flexibilidad en la traducción resultó ser falsa, ya que Brosgol [1974] demostró que un esquema de traducción basado en una gramática LL(1) se puede simular durante el análisis sintáctico LR(1). Independientemente, Watt [1977] utilizó no terminales marcadores para garantizar que los valores de los atributos heredados aparezcan en una pila durante el análisis sintáctico ascendente. Las posiciones en los lados derechos de las producciones donde se pueden insertar los no terminales marcadores sin perder la propiedad LR(1) son estudiadas por Purdom y Brown [1980] (véase Ejerc. 5.21). En Tarhio [1982] se proporcionan condiciones suficientes sobre las reglas semánticas para demostrar que no basta con exigir que los atributos heredados sean definidos por reglas de copia para

garantizar que los atributos puedan evaluarse durante el análisis sintáctico ascendente. Jones y Madsen [1980] proporcionan una caracterización, en cuanto a los estados de un analizador sintáctico, de los atributos que pueden evaluarse durante el análisis sintáctico LR(1). Como ejemplo de traducción que no puede realizarse durante el análisis sintáctico, Giegerich y Wilhelm [1978] consideran la generación de código para las expresiones booleanas. En la sección 8.6 se verá que se puede utilizar el relleno de retroceso para este problema, así que una segunda pasada completa no es necesaria.

Se han desarrollado varias herramientas para implantar definiciones dirigidas por la sintaxis, comenzando con FOLDS de Fang [1972], pero pocas se han visto muy utilizadas. DELTA, de Lorho [1977], construyó un grafo de dependencias en el momento de la compilación. Ahorraba espacio observando las duraciones de los atributos y eliminando reglas de copia. Se estudian los métodos de evaluación de atributos basados en árboles de análisis sintáctico en Kennedy y Ramanathan [1979] y en Cohen y Harry [1979].

Engelfriet [1984] examina los métodos para la evaluación de atributos. Un artículo parecido, de Courcelle [1984], estudia los fundamentos teóricos. HLP, descrito por Răihă y colaboradores [1983], realiza recorridos en profundidad alternativos, como los propuestos por Jazayeri y Walter [1975]. LINGUIST, de Farrow [1984], también hace pasadas alternativas. Ganzinger y colaboradores [1982] escriben que MUG permite que el orden en que se visitan los hijos de un nodo venga determinado por la producción en el nodo. GAG, obra de Kastens, Hutt y Zimmerman [1982], permite visitas repetidas a los hijos de un nodo. GAG implanta la clase de gramáticas con atributos ordenados definida por Kastens [1980]. La idea de las visitas repetidas aparece en el artículo anteriormente citado de Kennedy y Warren [1976], donde se construyen evaluadores para la clase más amplia de gramáticas fuertemente no circulares. Saarinen [1978] describe una modificación del método de Kennedy y Warren que ahorra espacio conservando los valores de los atributos en un pila si no son necesarios durante una visita posterior. Una implantación descrita por Jourdan [1984] construye evaluadores recursivos para dicha clase. También Katayama [1984] construye evaluadores recursivos. En NEATS, de Madsen [1980], se emplea un enfoque bastante diferente, construyendo un GDA para expresiones que representan valores de atributos.

El análisis de las dependencias en el momento de la construcción del compilador puede ahorrar tiempo y espacio en el momento de la compilación. La prueba de circularidad es un típico problema de análisis. Jazayeri, Ogden y Rounds [1975] demuestran que una prueba de circularidad exige una cantidad de tiempo exponencial como función del tamaño de la gramática. Las técnicas para mejorar la implantación de una prueba de circularidad son consideradas por Lorho y Pair [1975], Răihă y Saarinen [1982], y Deransart, Jourdan y Lorho [1984].

El espacio utilizado por evaluadores "ingenuos" ha provocado el desarrollo de técnicas para conservar espacio. Marill [1962] describió el algoritmo para asignar valores de atributos a registros de la sección 5.8 en un contexto bastante diferente. Sethi [1975] demostró que el problema de encontrar un tipo topológico del grafo de dependencias que minimice el número de registros utilizados es NP completo. El análisis en el momento de la compilación de las duraciones en un evaluador de múl-

tiples pasadas se desarrolla en Rähiä [1981] y en Jazayeri y Pozefsky [1981]. Branquart y colaboradores [1976] mencionan el uso de pilas independientes para guardar los atributos sintetizados y heredados durante un recorrido. GAG realiza un análisis de las duraciones y sitúa los valores de atributos en variables globales, pilas y nodos de árboles de análisis sintáctico si son necesarios. Farrow y Yellin [1984] hacen una comparación de las técnicas para ahorro de espacio utilizadas por GAG y LINGUIST.

CAPITULO 6

Comprobación de tipos

Un compilador debe comprobar si el programa fuente sigue tanto las convenciones sintácticas como las semánticas del lenguaje fuente. Esta comprobación, llamada *comprobación estática* (para distinguirla de la comprobación *dinámica* que se realiza durante la ejecución del programa objeto), garantiza la detección y comunicación de algunas clases de errores de programación. Los ejemplos de comprobación estática incluyen:

1. *Comprobaciones de tipos.* Un compilador debe informar de un error si se aplica un operador a un operando incompatible; por ejemplo, si se suman una variable tipo matriz y una variable de función.
2. *Comprobaciones del flujo del control.* Las proposiciones que hacen que el flujo del control abandone una construcción deben tener algún lugar a dónde transferir el flujo de control. Por ejemplo, una proposición **break** en C hace que el control abandone la proposición que la engloba, **while**, **for** o **switch** más cercana; si dicha proposición englobadora no existe, ocurre un error.
3. *Comprobaciones de unicidad.* Hay situaciones en que se debe definir un objeto una vez exactamente. Por ejemplo, en Pascal, un identificador debe declararse de forma única, las etiquetas en una proposición **case** deben ser diferentes y no se pueden repetir los elementos en un tipo escalar.
4. *Comprobaciones relacionadas con nombres.* En ocasiones, el mismo nombre debe aparecer dos o más veces. Por ejemplo, en Ada, un lazo o bloque puede tener un nombre que aparezca al principio y al final de la construcción. El compilador debe comprobar que se utilice el mismo nombre en ambos sitios.

En este capítulo, se trata principalmente la comprobación de tipos. Como indican los ejemplos anteriores, la mayoría de las otras comprobaciones estáticas son rutinarias y se pueden aplicar utilizando las técnicas del último capítulo. Algunas de ellas pueden formar parte de otras actividades. Por ejemplo, conforme se introduce información acerca de un nombre en una tabla de símbolos, se puede comprobar que el nombre esté declarado de una única manera. Muchos compiladores de Pascal combinan la comprobación estática y la generación de código intermedio con el análisis sintáctico. En construcciones más complejas, como las de Ada, puede ser conveniente tener una pasada de comprobación de tipos independiente entre el análisis sintáctico y la generación de código intermedio, como se indica en la figura 6.1.

Un comprobador de tipos se asegura de que el tipo de una construcción coincida con el previsto en su contexto. Por ejemplo, el operador aritmético predefinido `mod` en Pascal exige operandos de tipo entero, de modo que un comprobador de tipos debe asegurarse de que los operandos de `mod` tengan tipo entero. De igual manera, el comprobador de tipos debe asegurarse de que la desreferenciación se aplique sólo a un apuntador, de que la indización se haga sólo sobre una matriz, de que una función definida por el usuario se aplique al número y tipo correctos de argumentos, etcétera. En la sección 6.2 aparece la especificación de un comprobador de tipos simple. En la sección 6.3 se estudia la representación de los tipos y la cuestión de cuándo concuerdan dos tipos.

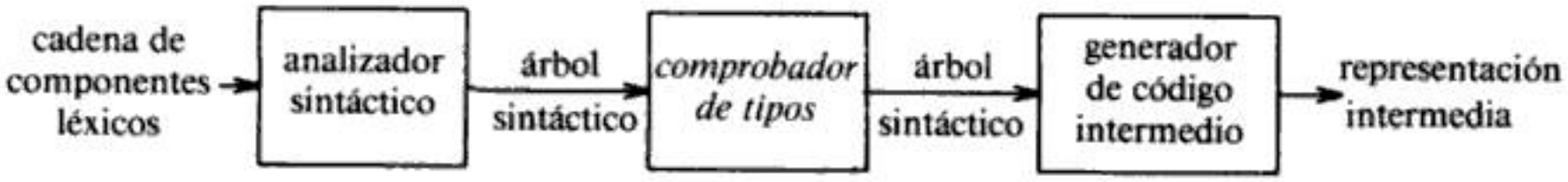


Fig. 6.1. Ubicación de un comprobador de tipos.

Puede necesitarse la información sobre los tipos reunida por un comprobador de tipos cuando se genera el código. Por ejemplo, los operadores aritméticos como + normalmente se aplican tanto a enteros como a reales, tal vez a otros tipos, y se debe examinar el contexto de + para determinar el sentido que se pretende dar. Se dice que un símbolo que puede representar diferentes operaciones en diferentes contextos está "sobrecargado". La sobrecarga puede ir acompañada de coacción de tipos, donde un compilador proporciona un operador para convertir un operando en el tipo esperado por el contexto.

Una noción diferente de la de sobrecarga es la de "polimorfismo". El cuerpo de una función polimórfica puede ejecutarse con argumentos de varios tipos. Este capítulo concluye con un algoritmo de unificación para inferir los tipos de las funciones polimórficas.

6.1 SISTEMAS DE TIPOS

El diseño de un comprobador de tipos para un lenguaje se basa en información acerca de las construcciones sintácticas del lenguaje, la noción de tipos y las reglas para asignar tipos a las construcciones de lenguaje. Los siguientes extractos del informe de Pascal y del manual de referencia de C, respectivamente, son ejemplos de la información con la que el diseñador de un compilador podría verse obligado a comenzar.

- "Si ambos operandos de los operadores aritméticos de suma, sustracción y multiplicación son de tipo entero, entonces el resultado es de tipo entero."
- "El resultado del operador unario `&` es un apuntador hacia el objeto al que se refiere el operando. Si el tipo del operando es '...', el tipo del resultado es 'apuntador a ...'."

En los anteriores extractos se encuentra implícita la idea de que cada expresión tiene asociado un tipo. Además, los tipos tienen estructura; el tipo "apuntador a ..." se construye a partir del tipo al que se refiere "...".

En Pascal y en C, los tipos son básicos o construidos. Los tipos básicos son los tipos atómicos sin estructura interna por lo que concierne al programador. En Pascal, los tipos básicos son *boolean*, *character*, *integer* y *real*. Los tipos de subrango, como `1..10`, y los tipos enumerados, como

```
(violeta, indigo, azul, verde, amarillo, naranja, rojo)
```

se pueden considerar como tipos básicos. Pascal admite que un programador construya tipos a partir de tipos básicos y otros tipos construidos, como matrices (*array*), registros (*record*) y conjuntos (*set*). Además, los apuntadores y las funciones también pueden considerarse como tipos construidos.

Expresiones de tipos

El tipo de una construcción de un lenguaje se denotará mediante una "expresión de tipo". De manera informal, una expresión de tipo es, o bien un tipo básico o se forma aplicando un operador llamado *constructor de tipos* a otras expresiones de tipos. Los conjuntos de tipos y constructores básicos dependen del lenguaje que deba comprobarse.

Este capítulo utiliza la siguiente definición de *expresiones de tipos*:

1. Un tipo básico es una expresión de tipo. Entre los tipos básicos se encuentran *boolean*, *char*, *integer* y *real*. Un tipo básico especial, *error_tipo*, señalará un error durante la comprobación de tipos. Por último, un tipo básico *vacío* que indica "la ausencia de valor", permite que se comprueben las proposiciones.

2. Como se puede dar nombre a las expresiones de tipos, el nombre de un tipo es una expresión de tipo. Un ejemplo de la utilidad de los nombres de tipo aparece en 3(c), más adelante. En la sección 6.3 se estudian las expresiones de tipos que contienen nombres.

3. Un constructor de tipos aplicado a expresiones de tipos es una expresión de tipo. Los constructores incluyen:

   a) *Matrices.* Si T es una expresión de tipo, entonces *array* (I, T) es una expresión de tipo que indica el tipo de una matriz con elementos de tipo T y conjunto de índices I. I es a menudo un rango de enteros. Por ejemplo, la declaración en Pascal

   ```
   var A: array[1..10] of integer;
   ```

   asocia la expresión de tipo *array* (1..10, *integer*) con `A`.

   b) *Productos.* Si T_1 y T_2 son expresiones de tipo, entonces su producto cartesiano $T_1 \times T_2$ es una expresión de tipo. Se supone que $\times$ es asociativa por la izquierda.

   c) *Registros.* La diferencia entre un registro y un producto es que los campos de un registro tienen nombres. El constructor de tipos *record* se aplicará a

una tupla formada con nombres de campos y tipos de campos. (Técnicamente, los nombres de campos deberían formar parte del constructor de tipos, pero es conveniente que los nombres de los campos estén junto a sus tipos asociados. En el Cap. 8, el constructor de tipos *record* se aplica a un apuntador a una tabla de símbolos que contiene entradas para los nombres de los campos.) Por ejemplo, el fragmento de programa en Pascal

```
type fila = record
                dirección: integer;
                lexema: array [1..15] of char
            end;
var tabla: array [1..101] of fila;
```

declara que el nombre de tipo `fila` representa la expresión de tipo

record (`dirección` × *integer*) × (`lexema` × *array* (1..15, *char*)))

y que la variable `tabla` es una matriz de registros de este tipo.

d) *Apuntadores.* Si T es una expresión de tipo, entonces *pointer* (T) es una expresión de tipo que indica el tipo "apuntador a un objeto de tipo T". Por ejemplo, en Pascal, la declaración

```
var p: ↑ fila
```

declara que la variable `p` tiene tipo *pointer* (`fila`).

e) *Funciones.* Matemáticamente, una función transforma elementos de un conjunto, el *dominio*, a elementos de otro conjunto, el *rango*. Se pueden considerar las funciones dentro de los lenguajes de programación como transformaciones de un *dominio tipo D* a un *rango tipo R*. La expresión de tipo $D \rightarrow R$ indicará el tipo de dicha función. Por ejemplo, la función predefinida `mod` de Pascal tiene un dominio de tipo *int* × *int*, es decir, un par de enteros, y rango de tipo *int*. Así, se dice que `mod` tiene el tipo[1]

int × *int* → *int*

Otro ejemplo: la declaración en Pascal

```
function f(a, b: char) : ↑ integer; . . .
```

dice que el tipo del dominio de `f` se indicará mediante *char* × *char* y el tipo del rango por *pointer* (*integer*). Entonces el tipo de `f` será indicado por la expresión de tipo

char × *char* → *pointer* (*integer*)

A menudo existen, por razones de implantación que se estudian en el siguiente capítulo, limitaciones en cuanto al tipo que una función puede devolver; por ejemplo, no se pueden devolver matrices o funciones. Sin embargo, existen lenguajes, entre los que LISP es el ejemplo más destacado, que

---

[1] Se supone que × tiene mayor precedencia que →, de modo que *int* × *int* → *int* es igual que (*int* × *int*) → *int*. Asimismo, → es asociativo por la derecha.

permiten que las funciones devuelvan objetos de tipos arbitrarios, así que se puede definir por ejemplo una función g de tipo

$$(integer \rightarrow integer) \rightarrow (integer \rightarrow integer)$$

Es decir, g toma como argumento una función que transforma un entero en un entero y g produce como resultado otra función del mismo tipo.

4. Las expresiones de tipo pueden contener variables cuyos valores son expresiones de tipos. Las variables de tipos se introducirán en la sección 6.6.

Una manera conveniente de representar expresiones de tipos es utilizando un grafo. Con el enfoque dirigido por la sintaxis de la sección 5.2, se puede construir un árbol o un GDA para una expresión de tipo, con nodos interiores para los constructores de tipos y hojas para los tipos básicos, nombres de tipo y variables de tipo (véase Fig. 6.2). En la sección 6.3 se dan ejemplos de representaciones de expresiones de tipos que se han utilizado en compiladores.

Fig. 6.2. Arbol y GDA, respectivamente, para *char* × *char* → *pointer* (*integer*).

Sistemas de tipos

Un *sistema de tipos* es una serie de reglas para asignar expresiones de tipos a las distintas partes de un programa. Un comprobador de tipos implanta un sistema de tipos. Los sistemas de tipos de este capítulo se especifican a la manera dirigida por la sintaxis, así que se implantan fácilmente usando las técnicas del capítulo anterior.

Diferentes compiladores o procesadores del mismo lenguaje pueden utilizar diferentes sistemas de tipos. Por ejemplo, en Pascal, el tipo de una matriz incluye el conjunto de índices de la matriz, de modo que una función con un argumento de tipo matriz sólo se puede aplicar a matrices con dicho conjunto de índices. Sin embargo, muchos compiladores de Pascal admiten que el conjunto de índices quede sin especificar cuando una matriz se pasa como argumento. Por tanto, estos compiladores usan un sistema de tipos distinto del de la definición del lenguaje Pascal. De manera similar, en el sistema UNIX, el mandato `lint` examina programas en C para buscar posibles errores con un sistema de tipos más detallado que el que utiliza el compilador mismo de C.

Comprobación estática y dinámica de tipos

Se dice que la comprobación realizada por un compilador es estática, mientras que la comprobación hecha al ejecutar el programa objeto se denomina dinámica. En principio, cualquier verificación se puede realizar dinámicamente, si el código objeto carga el tipo de un elemento junto con el valor de dicho elemento.

Un sistema de tipos *seguro* elimina la necesidad de comprobar dinámicamente errores de tipos ya que permite determinar estáticamente que dichos errores no pueden ocurrir cuando se está ejecutando el programa objeto. Es decir, si un sistema de tipos seguro asigna un tipo que no sea *error_tipo* a una parte de un programa, entonces los errores de tipo no pueden producirse cuando se está ejecutando el código objeto de dicha parte del programa. Se dice que un lenguaje es fuertemente tipificado si su compilador puede garantizar que los programas que acepte se ejecutarán sin errores de tipo.

En la práctica, algunas comprobaciones sólo se pueden hacer dinámicamente. Por ejemplo, si primero se declara

```
tabla: array[0..255] of char;
i: integer
```

y después se calcula `tabla[i]`; en general, un compilador no puede garantizar que durante la ejecución, el valor de `i` estará en el rango de 0 a 255[2].

Recuperación de errores

Como la comprobación de tipos tiene la capacidad de descubrir errores en los programas, es importante que un comprobador de tipos haga algo razonable cuando se descubre un error. Como mínimo, el compilador debe informar de la naturaleza y la posición del error. Es mejor que el comprobador de tipos se recupere de los errores, para que pueda comprobar el resto de la entrada. Como el manejo de errores afecta a las reglas de comprobación de tipos, tiene que diseñarse como parte del sistema de tipos desde el principio; las reglas tienen que servir para tratar los errores.

La inclusión del manejo de errores puede dar como resultado un sistema de tipos que vaya más allá del necesario para especificar programas correctos. Por ejemplo, cuando se ha producido un error, es posible que se desconozca el tipo del fragmento de programa formado de manera incorrecta. Tratar con información incompleta exige técnicas similares a las necesarias para lenguajes que no exigen que los identificadores se declaren antes de su uso. Las variables de tipo, que se estudian en la sección 6.6, pueden utilizarse para garantizar el uso adecuado de identificadores no declarados o aparentemente mal declarados.

6.2 ESPECIFICACION DE UN COMPROBADOR DE TIPOS SENCILLO

En esta sección se especifica un comprobador de tipos para un lenguaje simple en el que se debe declarar el tipo de cada identificador antes que el identificador se utilice. El comprobador de tipos es un esquema de traducción que sintetiza el tipo de cada expresión a partir de los tipos de sus subexpresiones. El comprobador de tipos puede manejar matrices, apuntadores, proposiciones y funciones.

---

[2] Se pueden utilizar técnicas de análisis de flujo de datos similares a las del capítulo 10 para deducir si `i` está dentro de los límites en algunos programas. Sin embargo, ninguna técnica puede tomar la decisión correcta siempre.

Un lenguaje simple

La gramática de la figura 6.3 genera programas, representados por el no terminal P, que constan de una secuencia de declaraciones D seguida de una expresión simple E.

$$
\begin{aligned}
&P \rightarrow D\ ;\ E\\
&D \rightarrow D\ ;\ D \mid \textbf{id}\ :\ T\\
&T \rightarrow \textbf{char} \mid \textbf{integer} \mid \textbf{array}\ [\ \textbf{núm}\ ]\ \textbf{of}\ T \mid \uparrow T\\
&E \rightarrow \textbf{literal} \mid \textbf{núm} \mid \textbf{id} \mid E\ \textbf{mod}\ E \mid E\ [\ E\ ] \mid E\uparrow
\end{aligned}
$$

Fig. 6.3. Gramática para el lenguaje fuente.

Un programa generado por la gramática de la figura 6.3 es:

```
clave: integer;
clave mod 1999
```

Antes de estudiar las expresiones, considérense los tipos del lenguaje. El lenguaje ya tiene dos tipos básicos, *char* e *integer*; se utiliza un tercer tipo básico, *error_tipo*, para señalar los errores. Para simplificar, se supone que todas las matrices comienzan en 1. Por ejemplo,

```
array [256] of char
```

conduce a la expresión de tipo *array* (1..256, *char*) que consta del constructor *array* aplicado al subrango 1..256 y el tipo *char*. Como en Pascal, el operador prefijo ↑ en las declaraciones construye un tipo apuntador, de modo que

```
↑ integer
```

conduce a la expresión de tipo *pointer* (*integer*) que consta del constructor *pointer* aplicado al tipo *integer*.

En el esquema de traducción de la figura 6.4, la acción asociada con la producción $D \rightarrow \textbf{id}\ :\ T$ guarda un tipo en una entrada de la tabla de símbolos para un identificador. La acción *añadetipo* (**id**.*entrada*, *T.tipo*) se aplica al atributo sintetizado *entrada* que apunta a la entrada de la tabla de símbolos para **id** y una expresión de tipo representada por el atributo sintetizado *tipo* del no terminal T.

$P \rightarrow D\ ;\ E$	
$D \rightarrow D\ ;\ D$	
$D \rightarrow \textbf{id}\ :\ T$	{ *añadetipo* (**id**.*entrada*, *T.tipo*) }
$T \rightarrow \textbf{char}$	{ *T.tipo* := *char* }
$T \rightarrow \textbf{integer}$	{ *T.tipo* := *integer* }
$T \rightarrow \uparrow T_1$	{ *T.tipo* := *pointer* (T_1.*tipo*) }
$T \rightarrow \textbf{array}\ [\ \textbf{núm}\ ]\ \textbf{of}\ T_1$	{ *T.tipo* := *array* (1..**núm**.*val*, T_1.*tipo*) }

Fig. 6.4. La parte de un esquema de traducción que guarda el tipo de un identificador.

Si T genera **char** o **integer**, entonces $T.tipo$ es *char* o *integer*, respectivamente. El límite superior de una matriz se obtiene del atributo *val* del componente léxico **núm** que da el número entero representado por **núm**. Se supone que las matrices comienzan en 1, así que el constructor de tipos *array* se aplica al subrango 1..**núm**.*val* y al tipo de elemento.

Como D aparece antes que E en el lado derecho de $P \rightarrow D\ ;\ E$, se puede asegurar que se guardarán los tipos de todos los identificadores declarados antes de que se compruebe la expresión generada por E (véase Cap. 5). De hecho, si se modifica de manera adecuada la gramática de la figura 6.3, se pueden implantar los esquemas de traducción de esta sección, si se desea, durante el análisis sintáctico descendente o ascendente.

Comprobación de tipos en las expresiones

En las siguientes reglas, el atributo sintetizado *tipo* para E da la expresión de tipo asignada por el sistema de tipos a la expresión generada por E. Las siguientes reglas semánticas señalan que las constantes representadas por los componentes léxicos **literal** y **núm** tienen tipo *char* e *integer*, respectivamente:

$E \rightarrow$ **literal**	$\{\ E.tipo := char\ \}$
$E \rightarrow$ **núm**	$\{\ E.tipo := integer\ \}$

Se utiliza la función *busca*(*e*) para traer el tipo guardado en la entrada de la tabla de símbolos apuntada por *e*. Cuando un identificador aparece en una expresión, se trae su tipo declarado y se asigna al atributo *tipo*:

$E \rightarrow$ **id**	$\{\ E.tipo := busca\ (\textbf{id}.entrada)\ \}$

La expresión formada aplicando el operador **mod** a dos subexpresiones de tipo *integer* tiene tipo *integer*; de lo contrario, su tipo es *error_tipo*. La regla es

$E \rightarrow E_1$ **mod** E_2	$\{\ E.tipo :=$ **if** $E_1.tipo = integer$ **and** $E_2.tipo = integer$ **then** *integer* **else** *error_tipo* $\}$

En una referencia a una matriz $E_1\ [E_2]$, la expresión de índice E_2 debe tener tipo entero, en cuyo caso el resultado es el tipo de elemento t obtenido del tipo $array\ (s, t)$ de E_1; no se utiliza el conjunto de índices s de la matriz.

$E \rightarrow E_1\ [E_2]$	$\{\ E.tipo :=$ **if** $E_2.tipo = integer$ **and** $E_1.tipo = array\ (s, t) \rightarrow$ **then** t **else** *error_tipo* $\}$

Dentro de las expresiones, el operador postfijo ↑ entrega el objeto apuntado por su operando. El tipo de $E\uparrow$ es el tipo t del objeto apuntado por el apuntador E:

$E \rightarrow E_1\uparrow$	$\{\ E.tipo :=$ **if** $E_1.tipo = pointer\ (t)$ **then** t **else** *error_tipo* $\}$

Se deja al lector que añada producciones y reglas semánticas para admitir tipos adicionales y operaciones dentro de las expresiones. Por ejemplo, para permitir que los identificadores tengan el tipo *boolean*, se puede introducir la producción $T \rightarrow$ **boolean** en la gramática de la figura 6.3. La introducción de operadores de comparación como < y conectivos lógicos como **and** en las producciones de E permitiría la construcción de expresiones de tipo *boolean*.

Comprobación de tipos en las proposiciones

Puesto que las construcciones de los lenguajes, como las proposiciones, carecen típicamente de valores, se les puede asignar el tipo básico especial *vacío*. Si se detecta un error dentro de una proposición, el tipo asignado a la proposición es *error_tipo*.

Las proposiciones que aquí se consideran son las proposiciones de asignación, condicional y de lazo **while**. Las secuencias de proposiciones se separan con símbolos de punto y coma. Las producciones de la figura 6.5 se pueden combinar con las de la figura 6.3 si se cambia de producción de un programa completo a $P \rightarrow D$; S. Ahora un programa consta de declaraciones seguidas de proposiciones; se siguen necesitando las reglas anteriores para comprobar expresiones porque las proposiciones pueden tener expresiones dentro de ellas.

$S \rightarrow$ **id** $:= E$	{ *S.tipo* := **if id**.*tipo* = *E.tipo* **then** *vacío* **else** *error_tipo* }
$S \rightarrow$ **if** E **then** S_1	{ *S.tipo* := **if** *E.tipo* = *boolean* **then** S_1.*tipo* **else** *error_tipo* }
$S \rightarrow$ **while** E **do** S_1	{ *S.tipo* := **if** *E.tipo* = *boolean* **then** S_1.*tipo* **else** *error_tipo* }
$S \rightarrow S_1 ; S_2$	{ *S.tipo* := **if** S_1.*tipo* = *vacío* **and** S_2.*tipo* = *vacío* **then** *vacío* **else** *error_tipo* }

Fig. 6.5. Esquema de traducción para comprobar el tipo de las proposiciones.

En la figura 6.5 se dan las reglas para comprobar proposiciones. La primera regla comprueba que los lados izquierdo y derecho de una proposición de asignación tengan el mismo tipo[3]. La segunda y tercera reglas especifican que las expresiones en las proposiciones condicional y **while** deben tener tipo *boolean*. Los errores son propagados por la última regla de la figura 6.5 porque una secuencia de proposiciones tiene tipo *vacío* sólo si cada subproposición tiene tipo *vacío*. En estas reglas, una discordancia de tipos produce el tipo *error_tipo*; por supuesto, un comprobador de tipos amigable informaría asimismo de la naturaleza y la posición de la discordancia de tipos.

---

[3] Si se admite una expresión en el lado izquierdo de una asignación, entonces también hay que distinguir entre valores de lado izquierdo y valores de lado derecho. Por ejemplo, `1 := 2` es incorrecto porque no se le pueden asignar valores a la constante `1`.

Comprobación de tipos de funciones

La aplicación de una función a un argumento puede ser representada por la producción

$$E \rightarrow E\ (\ E\ )$$

en la que una expresión es la aplicación de una expresión a otra. Se pueden aumentar las reglas para asociar expresiones de tipos con el no terminal T con la siguiente producción y acción para permitir tipos de función en las declaraciones.

$$T \rightarrow T_1\ '\!\rightarrow'\ T_2 \qquad \{\ T.tipo := T_1.tipo \rightarrow T_2.tipo\ \}$$

Las comillas alrededor de la flecha que se usa como constructor de función lo distinguen de la flecha que se usa como metasímbolo en una producción.

La regla para comprobar el tipo de aplicación de una función es

$$E \rightarrow E_1\ (E_2) \qquad \{\ E.tipo := \textbf{if}\ E_2.tipo = s\ \textbf{and}\ E_1.tipo = s \rightarrow t\ \textbf{then}\ t\ \textbf{else}\ error\_tipo\ \}$$

Esta regla indica que en una expresión formada por la aplicación de E_1 a E_2, el tipo de E_1 debe ser una función $s \rightarrow t$ del tipo s de E_2 a algún tipo de rango t; el tipo de $E_1(E_2)$ es t.

Se pueden estudiar muchos aspectos relacionados con la comprobación de tipos en presencia de funciones con respecto a la sencilla sintaxis anterior. La generalización hacia funciones con más de un argumento se realiza construyendo un tipo producto que consta de los argumentos. Obsérvese que n argumentos de tipo $T_1, \ldots, T_n$ se pueden considerar como un solo argumento de tipo $T_1 \times \ldots \times T_n$. Por ejemplo, se puede escribir

```
raíz : (real → real) × real → real                    (6.1)
```

para declarar una función `raíz` que toma una función de reales a reales y un real como argumentos y devuelve un real. La sintaxis tipo Pascal para esta declaración es

```
function raíz (function f (real): real; x: real): real
```

La sintaxis de (6.1) separa la declaración del tipo de una función de los nombres de sus parámetros.

6.3 EQUIVALENCIA DE EXPRESIONES DE TIPOS

Las reglas de comprobación de la última sección tienen la forma, "**if** dos expresiones de tipo son iguales **then** devuelve un cierto tipo **else** devuelve *error_tipo*". Por tanto, es importante tener una definición precisa de cuándo son equivalentes dos expresiones de tipos. Surgen posibles ambigüedades cuando se dan nombres a las expresiones de tipos y dichos nombres se utilizan después en posteriores expresiones de tipos. El aspecto clave es si un nombre en una expresión de tipos se representa a sí mismo o si es la abreviatura de otra expresión de tipo.

Como existe una interacción entre la noción de equivalencia de tipos y la representación de los tipos se tratarán ambos a la vez. Para una mayor eficiencia, los

compiladores utilizan representaciones que permitan determinar rápidamente la equivalencia de tipos. La noción de equivalencia de tipos implantada por un compilador específico se puede explicar a menudo utilizando los conceptos de equivalencia estructural y equivalencia de nombre que se estudian en esta sección. El análisis se hace mediante una representación de grafos de las expresiones de tipos, con hojas para los tipos básicos y nombres de tipos, y nodos interiores para los constructores de tipos, como en la figura 6.2. Como se verá, los tipos definidos de manera recursiva conducen a ciclos en el grafo de tipos si un nombre se considera la abreviatura de una expresión de tipo.

Equivalencia estructural de las expresiones de tipos

Ya que las expresiones de tipos se construyen a partir de tipos básicos y constructores, una noción natural de equivalencia entre dos expresiones de tipos es la *equivalencia estructural*; es decir, dos expresiones son, o bien el mismo tipo básico, o están formadas aplicando el mismo constructor a tipos estructuralmente equivalentes. Por tanto, dos expresiones de tipos son estructuralmente equivalentes si, y sólo si, son idénticas. Por ejemplo, la expresión de tipo *integer* sólo es equivalente a *integer* porque son el mismo tipo básico. De manera similar, *pointer* (*integer*) sólo es equivalente a *pointer* (*integer*) porque los dos se forman aplicando el mismo constructor, *pointer*, a tipos equivalentes. Si se utiliza el método de número de valor del algoritmo 5.1 para construir una representación por medio de un GDA de las expresiones de tipos, entonces las expresiones de tipos idénticas se representarán con el mismo nodo.

En la práctica, a menudo es necesario modificar la noción de equivalencia estructural para reflejar las verdaderas reglas de comprobación de tipos del lenguaje fuente. Por ejemplo, cuando se pasan matrices como parámetros, quizá no se quiera incluir los límites de la matriz como parte del tipo.

Se puede adaptar el algoritmo para comprobar la equivalencia estructrural de la figura 6.6 para examinar nociones de equivalencia modificadas. El algoritmo asume que los únicos constructores de tipos son para matrices, productos, apuntadores y funciones. El algoritmo compara recursivamente la estructura de las expresiones de tipos sin comprobar si existen ciclos, así que puede ser aplicado a un árbol o a un GDA. No hace falta que las expresiones de tipos idénticas sean representadas por el mismo nodo en el GDA. Se puede comprobar la equivalencia estructural de los nodos en los grafos de tipos con ciclos utilizando un algoritmo de la sección 6.7.

Los límites de matriz s_1 y t_1 en

$$s = array(s_1, s_2)$$
$$t = array(t_1, t_2)$$

se pasan por alto si la prueba para la equivalencia de matrices en las líneas 4 y 5 de la figura 6.6 se reformula como sigue:

else if $s = array(s_1, s_2)$ **and** $t = array(t_1, t_2)$ **then**
    return *equivest* (s_2, t_2)

En algunas ocasiones, se puede encontrar una representación para las expresiones de tipos que sea mucho más compacta que la notación de grafos de tipos. En el

siguiente ejemplo, parte de la información sacada de una expresión de tipo se codifica como una secuencia de bits, que entonces puede ser interpretada como un solo entero. La codificación es tal que números enteros distintos representan expresiones de tipos estructuralmente no equivalentes. Se puede acelerar la prueba para la equivalencia estructural comprobando primero la desigualdad estructural mediante la comparación de las representaciones de enteros de los tipos, y aplicando después el algoritmo de la figura 6.6 sólo si los enteros son iguales.

```
(1)  function equivest (s, t): boolean;
     begin
(2)      if s y t son el mismo tipo básico then
(3)          return true
(4)      else if s = array (s2, s2) and t = array (t1, t2) then
(5)          return equivest (s1, t1) and equivest (s2, t2)
(6)      else if s = s1 × s2 and t = t1 × t2 then
(7)          return equivest (s1, t1) and equivest (s2, t2)
(8)      else if s = pointer (s1) and t = pointer (t1) then
(9)          return equivest (s1, t1)
(10)     else if s = s1 → s2 and t = t1 → t2 then
(11)         return equivest (s1, t1) and equivest (s2, t2)
         else
(12)         return false
     end
```

Fig. 6.6. Comprobación de la equivalencia estructural de dos expresiones de tipos s y t.

Ejemplo 6.1. La codificación de las expresiones de tipos de este ejemplo proviene de un compilador de C escrito por D. M. Ritchie. También la utiliza el compilador de C descrito en Johnson [1979].

Considérense expresiones de tipos con los siguientes constructores de tipos para apuntadores, funciones y matrices: *pointer* (t) indica un apuntador al tipo t, *freturns* (t) señala una función de algunos argumentos que devuelve un objeto de tipo t, y *array* (t) indica una matriz (de longitud no determinada) de elementos de tipo t. Obsérvese que se han simplificado los constructores de tipos matriz y función. Se tendrá localizado el número de elementos de una matriz, pero el número se guarda en otro lugar, de modo que no forme parte del constructor de tipo *array*. Asimismo, el único operando del constructor *freturns* es el tipo del resultado de una función; los tipos de los argumentos de la función se guardarán en otro lugar. Por tanto, los objetos con expresiones estructuralmente equivalentes de este sistema de tipos quizá sigan sin pasar la prueba de la figura 6.6 que ahí se aplica a un sistema de tipos más detallado.

Como cada uno de estos constructores es un operador unario, las expresiones de tipos formadas mediante la aplicación de dichos constructores a tipos básicos tienen una estructura muy uniforme. Ejemplos de dichas expresiones de tipos son:

char
freturns (*char*)
pointer (*freturns* (*char*))
array (*pointer* (*freturns* (*char*)))

Cada una de las expresiones anteriores puede representarse con una secuencia de bits mediante un esquema de codificación simple. Como sólo hay tres constructores de tipos, se pueden utilizar dos bits para codificar un constructor, de la siguiente manera:

CONSTRUCTOR DE TIPOS	CODIFICACIÓN
pointer	01
array	10
freturns	11

Los tipos básicos de C se codifican utilizando cuatro bits en Johnson [1979]; los cuatro tipos básicos de este ejemplo se pueden codificar de la siguiente manera:

TIPO BÁSICO	CODIFICACIÓN
boolean	0000
char	0001
integer	0010
real	0011

Las expresiones de tipos limitadas ya se pueden codificar como secuencias de bits. Los cuatro bits situados más a la derecha codifican el tipo básico de una expresión de tipo. Moviéndose de derecha a izquierda, los dos bits siguientes indican el constructor aplicado al tipo básico, los dos bits siguientes describen el constructor que se aplica a éste, y así sucesivamente. Por ejemplo,

EXPRESIÓN DE TIPO	CODIFICACIÓN
char	000000 0001
freturns (*char*)	000011 0001
pointer (*freturns* (*char*))	000111 0001
array (*pointer* (*freturns* (*char*)))	100111 0001

Véase el ejercicio 6.12 si se desean más detalles.

Además de ahorrar espacio, dicha representación mantiene los constructores que aparecen en cualquier expresión de tipo. Dos secuencias de bits distintas no pueden representar el mismo tipo porque, o bien el tipo básico o los constructores en las expresiones de tipos son distintos. Por supuesto, tipos distintos podrían tener la misma secuencia de bits puesto que no se representan el tamaño de las matrices y los argumentos de las funciones.

La codificación de este ejemplo se podría ampliar para incluir tipos registro (*record*). La idea es considerar cada registro como un tipo básico en la codificación; una secuencia de bits independiente codifica el tipo de cada campo del registro. Se examina más detalladamente la equivalencia de tipos en C en el ejemplo 6.4. □

Nombres para expresiones de tipos

En algunos lenguajes se puede dar nombre a los tipos. Por ejemplo, en el fragmento del programa en Pascal

```
type liga      = ↑ nodo;
var  siguiente : enlace;
     último    : enlace;                    (6.2)
     p         : ↑ nodo;
     q, r      : ↑ nodo;
```

el identificador `enlace` está declarado como el nombre del tipo ↑ `nodo`. De inmediato surge la pregunta siguiente, ¿tienen idénticos tipos todas las variables `siguiente`, `último`, `p`, `q` y `r`? Curiosamente, la respuesta depende de la implantación. El problema surgió porque el informe de Pascal no definió el término "tipo idéntico".

Para modelar esta situación, se permitirá dar nombres a las expresiones de tipo y que estos nombres aparezcan en expresiones de tipos donde previamente sólo existían tipos básicos. Por ejemplo, si `nodo` es el nombre de una expresión de tipo, entonces *pointer* (`nodo`) es una expresión de tipo. Por el momento, supóngase que no existen definiciones circulares de expresiones de tipos tal como considerar que `nodo` es el nombre de una expresión de tipo que contenga `nodo`.

Cuando se permiten los nombres en las expresiones de tipos, surgen dos nociones de equivalencia de expresiones de tipos, según el tratamiento de los nombres. La *equivalencia de nombres* considera cada nombre de un tipo como un tipo distinto, de modo que dos expresiones de tipo tienen equivalencia de nombre si, y sólo si, son idénticas. Con la *equivalencia estructural*, los nombres se sustituyen por las expresiones de tipos que definen, así que dos expresiones de tipos son estructuralmente equivalentes si representan dos expresiones de tipos estructuralmente equivalentes cuando todos los nombres han sido sustituidos.

Ejemplo 6.2. En la siguiente tabla se dan las expresiones de tipos que pueden asociarse con las variables de las declaraciones (6.2).

VARIABLE	EXPRESIÓN DE TIPO
`siguiente`	`enlace`
`último`	`enlace`
`p`	*pointer* (`nodo`)
`q`	*pointer* (`nodo`)
`r`	*pointer* (`nodo`)

Con la equivalencia de nombres, las variables `siguiente` y `último` tienen el mismo tipo porque tienen las mismas expresiones de tipos asociadas. Las variables `p`, `q` y `r` también tienen el mismo tipo, pero `p` y `siguiente` no, porque sus expresiones de tipos asociadas son diferentes. Con la equivalencia estructural, las cinco variables tienen el mismo tipo porque `enlace` es un nombre para la expresión de tipo *pointer* (`nodo`). □

Los conceptos de equivalencia estructural y de nombre son útiles para explicar las reglas que utilizan varios lenguajes para asociar tipos con identificadores en las declaraciones.

Ejemplo 6.3. En Pascal surge una confusión ya que muchas implantaciones asocian un nombre de tipo implícito con cada identificador declarado. Si la declaración contiene una expresión de tipo que no sea un nombre, se crea un nombre implícito. Cada vez que aparece una expresión de tipo en una declaración de variables, se crea un nombre implícito nuevo.

Por tanto, se crean nombres implícitos para las expresiones de tipos de las dos declaraciones que contienen `p`, `q` y `r` en (6.2). Es decir, las declaraciones se consideran como si fueran

```
type liga       = ↑ nodo;
     np         = ↑ nodo;
     nqr        = ↑ nodo;
var  siguiente  : liga;
     último     : liga;
     p          : np;
     q          : nqr;
     r          : nqr;
```

Se han introducido los nuevos nombres de tipo `np` y `nqr`. Con la equivalencia de nombres, como `siguiente` y `último` se declaran con el mismo nombre de tipo, se considera que tienen tipos equivalentes. De manera similar, se considera que `q` y `r` tienen tipos equivalentes porque tienen asociado el mismo nombre de tipo implícito. Sin embargo, `p`, `q` y `siguiente` no tienen tipos equivalentes, porque todos tienen tipos con nombres distintos.

La implantación habitual consiste en construir un grafo de tipos para representar a los tipos. Cada vez que se ve un constructor de tipos o un tipo básico, se crea un nuevo nodo. Cada vez que se ve un nombre de tipo nuevo, se crea una hoja. Sin embargo, se conserva la expresión de tipo a la que se refiere el nombre. Con esta representación, dos expresiones de tipos son equivalentes si están representadas por el mismo nodo en el grafo de tipos. En la figura 6.7 se muestra un grafo de tipos para las declaraciones (6.2). Las líneas de puntos indican la asociación entre variables y nodos en el grafo de tipos. Obsérvese que el nombre de tipo `nodo` tiene tres padres, todos etiquetados con *pointer*. Aparece un signo igual entre el nombre de tipo `enlace` y el nodo en el grafo de tipos al que se refiere. □

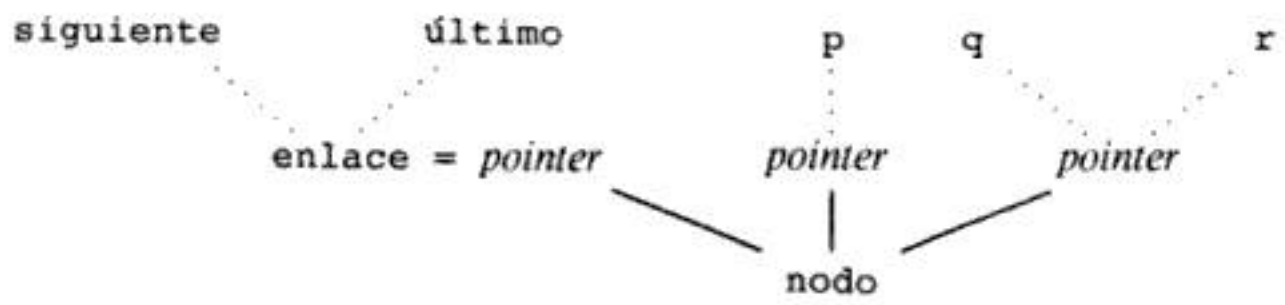


Fig. 6.7. Asociación de variables y nodos en el grafo de tipos.

Ciclos en las representaciones de tipos

Las estructuras de datos básicas, como las listas enlazadas y los árboles, a menudo se definen recursivamente; por ejemplo, una lista enlazada o está vacía o consta de un nodo con un apuntador a una lista enlazada. Dichas estructuras de datos generalmente se implantan mediante registros que contienen apuntadores a registros similares, y los nombres de tipos desempeñan un papel esencial en la definición de tipos de dichos registros.

Considérese una lista enlazada de nodos, donde cada uno contiene alguna información de tipo entero y un apuntador al siguiente nodo de la lista. Las declaraciones en Pascal de nombres de tipos correspondientes a ligas y nodos son:

```
type  enlace = ↑ nodo;
      nodo   = record
                 info      : integer;
                 siguiente : enlace
               end;
```

Obsérvese que el nombre de tipo `enlace` se define mediante `nodo` y `nodo` se define mediante `enlace`, de modo que sus definiciones son recursivas.

Los nombres de tipo definidos recursivamente se pueden sustituir si se quieren introducir ciclos en el grafo de tipos. Si *pointer* (`nodo`) se sustituye por `enlace`, se obtiene para `nodo` la expresión de tipo que se muestra en la figura 6.8(a). Si se utilizan ciclos, como en la figura 6.8(b), se puede eliminar la mención de `nodo` de la parte del grafo de tipo debajo del nodo etiquetado con *pointer*.

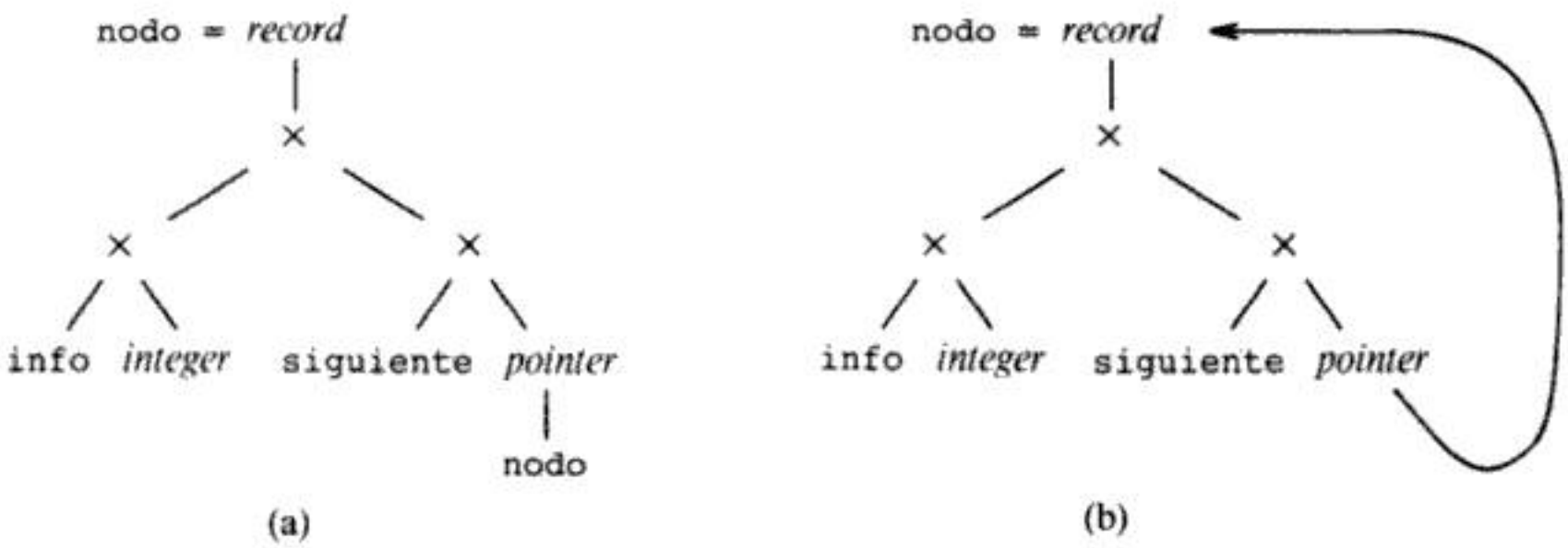


Fig. 6.8. El nombre de tipo `nodo` definido recursivamente.

Ejemplo 6.4. C evita los ciclos en los grafos de tipos utilizando la equivalencia estructural para todos los tipos excepto los registros. En C, la declaración de `nodo` sería así:

```
struct nodo {
    int info;
    struct nodo *siguiente;
};
```

C utiliza la palabra clave `struct` en lugar de `record`, y el nombre `nodo` se convierte en parte del tipo del registro. En efecto, C utiliza la representación acíclica de la figura 6.8(a).

C exige que los nombres de tipos se declaren antes de su uso, con la excepción de que permite apuntadores a tipos de registro no declarados. Por tanto, todos los ciclos potenciales se deben a apuntadores a registros. Como el nombre de un registro es parte de su tipo, la búsqueda de equivalencia estructural se detiene cuando se alcanza un constructor de registros, y los tipos que se están comparando son equivalentes porque tienen el mismo tipo de registro con nombre o bien son no equivalentes. □

6.4 CONVERSIONES DE TIPOS

Considérense expresiones como `x + i` donde `x` es de tipo real e `i` es de tipo entero. Como la representación de enteros y reales es distinta dentro de un computador, y se utilizan instrucciones de máquina distintas para las operaciones sobre enteros y reales, puede que el compilador tenga que convertir primero uno de los operandos de + para garantizar que ambos operandos sean del mismo tipo cuando tenga lugar la suma.

La definición del lenguaje especifica las conversiones necesarias. Cuando un entero se asigna a un real, o viceversa, la conversión es al tipo del lado izquierdo de la asignación. En expresiones, la transformación más común es la de convertir el entero en un número real y después realizar una operación real con el par de operandos reales obtenido. Se puede utilizar el comprobador de tipos en un compilador para insertar estas operaciones de conversión en la representación intermedia del programa fuente. Por ejemplo, la notación postfija para `x + i` puede ser

```
x i entareal real+
```

El operador **entareal** transforma `i` de entero a real y después **real+** realiza la suma real con sus operandos.

La conversión de tipos surge con frecuencia en otro contexto. Se dice que un símbolo que tiene distintos significados dependiendo de su contexto está sobrecargado. La sobrecarga se estudiará en la siguiente sección, pero se menciona porque las conversiones de tipo a menudo acompañan a la sobrecarga.

Coerciones

Se dice que la conversión de un tipo a otro es *implícita* si el compilador la va a realizar automáticamente. Las conversiones de tipo implícitas, también llamadas *coerciones*, se limitan en muchos lenguajes a situaciones donde en principio no se pierde ninguna información; por ejemplo, un entero se puede transformar en un real pero no al contrario. En la práctica, sin embargo, la pérdida es posible cuando un número real debe ajustarse al mismo número de bits que un entero.

Se dice que la conversión es *explícita* si el programador debe escribir algo para motivar la conversión. En la práctica, todas las conversiones en Ada son explícitas. Para un comprobador de tipos, las conversiones explícitas parecen iguales que las aplicaciones de función, así que no presentan problemas nuevos.

Por ejemplo, en Pascal, la función predefinida `ord` transforma un carácter en un entero, y `chr` realiza la transformación inversa de entero a carácter, así que estas conversiones son explícitas. Por otra parte, C coerciona (es decir, convierte implícitamente) los caracteres ASCII a enteros entre `0` y `127` en las expresiones aritméticas.

Ejemplo 6.5. Considérense las expresiones formadas aplicando un operador aritmético **op** a constantes e identificadores, como en la gramática de la figura 6.9. Supóngase que hay dos tipos, real y entero, donde los enteros se convierten en reales cuando sea necesario. El atributo *tipo* del no terminal E puede ser entero o real, y las reglas de comprobación de tipos se muestran en la figura 6.9. Como en la sección 6.2, la función *busca*(*e*) devuelve el tipo guardado en la entrada a la tabla de símbolos apuntada por *e*. □

PRODUCCIÓN	REGLA SEMÁNTICA
$E \rightarrow$ **núm**	$E.tipo :=$ *integer*
$E \rightarrow$ **núm . núm**	$E.tipo :=$ *real*
$E \rightarrow$ **id**	$E.tipo :=$ *busca* (**id**.*entrada*)
$E \rightarrow E_1$ **op** E_2	$E.tipo :=$ **if** $E_1.tipo =$ *integer* **and** $E_2.tipo =$ *integer* **then** *integer* **else if** $E_1.tipo =$ *integer* **and** $E_2.tipo =$ *real* **then** *real* **else if** $E_1.tipo =$ *real* **and** $E_2.tipo =$ *integer* **then** *real* **else if** $E_1.tipo =$ *real* **and** $E_2.tipo =$ *real* **then** *real* **else** *error_tipo*

Fig. 6.9. Reglas de comprobación de tipos para la coerción de entero a real.

La conversión implícita de constantes se puede realizar generalmente en el momento de la compilación, mejorando a menudo el tiempo de ejecución del programa objeto. En los siguientes fragmentos de código, `X` es una matriz de reales a la que se asigna 1 como valor inicial en todos sus elementos. Utilizando un compilador de Pascal, Bentley [1982] vio que el fragmento de código

```
for I := 1 to N do X[I] := 1
```

empleó 48.4N microsegundos, mientras que el fragmento

```
for I := 1 to N do X[I] : = 1.0
```

empleó 5.4N microsegundos. Ambos, asignan el valor uno a los elementos de una matriz de reales. Sin embargo, el código generado (por este compilador) para el primer fragmento contenía una llamada a una rutina en el momento de la ejecución para convertir la representación entera de `1` en la de número real. Como ya se conoce en el momento de la compilación que `X` es una matriz de reales, un compilador más completo convertiría `1` en `1.0` en el momento de la compilación.

6.5 SOBRECARGA DE FUNCIONES Y OPERADORES

Un símbolo *sobrecargado* es el que tiene distintos significados dependiendo de su contexto. En matemáticas, el operador de suma + está sobrecargado, porque + en $A + B$ tiene distintos significados si A y B son enteros, reales, números complejos o matrices. En Ada, los paréntesis `()` están sobrecargados; la expresión `A(I)` puede ser el `I`-ésimo elemento de la matriz `A`, una llamada a la función `A` con argumento `I` o una conversión explícita de la expresión `I` al tipo `A`.

La sobrecarga se *resuelve* cuando se determina un significado único para un caso de un símbolo sobrecargado. Por ejemplo, si + puede denotar una suma real o entera, entonces los dos casos de + en `x+(i+j)` pueden denotar diferentes formas de suma, dependiendo de los tipos de `x`, `i` y `j`. La resolución de la sobrecarga a menudo aparece referida como *identificación de operadores*, porque determina la operación que denota un símbolo de operador.

Los operadores aritméticos están sobrecargados en la mayoría de los lenguajes. Sin embargo, la sobrecarga que afecta a los operadores aritméticos como + se puede resolver observando únicamente los argumentos del operador. El análisis de casos para determinar si se debe utilizar la versión entera o la real de + es similar al de la regla semántica para $E \rightarrow E_1$ **op** E_2 de la figura 6.9, donde el tipo de E se determina observando los tipos posibles de E_1 y E_2.

Conjunto de tipos posibles para una subexpresión

No siempre es posible resolver la sobrecarga observando únicamente los argumentos de una función, como lo muestra el siguiente ejemplo. En lugar de un solo tipo, una subexpresión por sí sola puede tener un conjunto de tipos posibles. En Ada, el contexto debe proporcionar información suficiente para limitar la elección a un solo tipo.

Ejemplo 6.6. En Ada, una de las interpretaciones estándar (es decir, predefinida) del operador * es la de una función que va de un par de enteros a un entero. El operador puede sobrecargarse si se añaden declaraciones como éstas:

```
function "*" ( i, j : integer ) return complex;
function "*" ( x, y : complex ) return complex;
```

Después de estas declaraciones, los tipos posibles para * incluyen:

$$\begin{aligned} \textit{integer} \times \textit{integer} &\rightarrow \textit{integer} \\ \textit{integer} \times \textit{integer} &\rightarrow \textit{complex} \\ \textit{complex} \times \textit{complex} &\rightarrow \textit{complex} \end{aligned}$$

Supóngase que el único tipo posible para `2`, `3` y `5` es entero. Con las declaraciones anteriores, la subexpresión `3*5` tiene tipo entero o complejo, dependiendo de su contexto. Si la expresión completa es `2*(3*5)`, entonces `3*5` debe tener tipo entero porque * toma un par de enteros o un par de números complejos como argumentos. Por otro lado, `3*5` debe tener tipo complejo si la expresión completa es `(3*5)*z` y `z` se declara complejo. □

En la sección 6.2, se consideró que cada expresión tenía un tipo único, de modo que la regla de comprobación de tipos para la aplicación de funciones era

$$E \rightarrow E_1 \ (\ E_2 \ ) \qquad \{ E.tipo := \textbf{if } E_2.tipo = s \textbf{ and } E_1.tipo = s \rightarrow t \textbf{ then } t \textbf{ else } error\_tipo \}$$

En la figura 6.10 aparece la generalización natural de esta regla a conjuntos de tipos. La única operación en la figura 6.10 es la de aplicación de funciones; las reglas para comprobar otros operadores en las expresiones son similares. Pueden existir varias declaraciones de un identificador sobrecargado, así que se supone que una entrada de una tabla de símbolos puede contener un conjunto de tipos posibles; este conjunto es devuelto por la función *busca*. El no terminal inicial E' genera una expresión completa. Su papel se aclara a continuación.

PRODUCCIÓN	REGLA SEMÁNTICA
$E' \rightarrow E$	$E'.tipos := E.tipos$
$E \rightarrow$ **id**	$E.tipos :=$ *busca* (**id**.*entrada*)
$E \rightarrow E_1 \ (\ E_2 \ )$	$E.tipos := \{ t \mid$ existe una s en $E_2.tipos$ tal que $s \rightarrow t$ está en $E_1.tipos \}$

Fig. 6.10. Determinación del conjunto de tipos posibles de una expresión.

En palabras, la tercera regla de la figura 6.10 indica que si s es uno de los tipos de E_2 y uno de los tipos de E_1 puede transformar s en t, entonces t es uno de los tipos de $E_1(E_2)$. Una discordancia de tipos durante la aplicación de la función da como resultado que el conjunto $E.tipos$ se convierta en vacío, condición que se utiliza temporalmente para señalar un error de tipo.

Ejemplo 6.7. Además de ilustrar la especificación de la figura 6.10, este ejemplo sugiere cómo llevar este enfoque a otras construcciones. Por ejemplo, se considera la expresión 3*5. Sean las declaraciones del operador * como en el ejemplo 6.6. Es decir * puede transformar un par de enteros en un entero o en un número complejo, dependiendo del contexto. En la figura 6.11, se muestran el conjunto de tipos posibles para las subexpresiones de 3*5, donde i y c son abreviaturas de *integer* y *complex*, respectivamente.

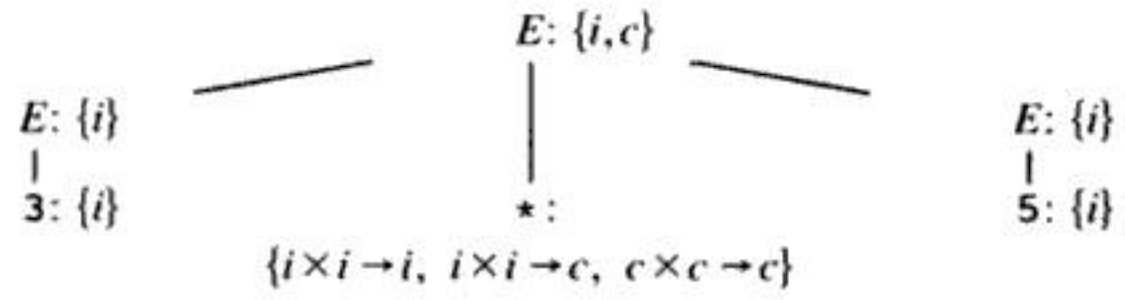


Fig. 6.11. Conjunto de tipos posibles para la expresión 3*5.

De nuevo, supóngase que el único tipo posible para `3` y `5` es *integer*. Por tanto, el operador * se aplica a un par de enteros. Si se considera este par de enteros como una unidad, su tipo viene dado por *integer* × *integer*. Hay dos funciones en el conjunto de tipos para * que se aplican a pares de enteros; una devuelve un entero, mientras que la otra devuelve un número complejo, así que la raíz puede tener tipo *integer* o tipo *complex*. □

Reducción del conjunto de tipos posibles

Ada exige que una expresión completa tenga un tipo único. Dado un tipo único a partir del contexto, se pueden reducir las elecciones de tipo para cada subexpresión. Si este proceso no da como resultado un tipo único para cada subexpresión, entonces se declara un error de tipo para la expresión.

Antes de trabajar de forma descendente de una expresión a sus subexpresiones, se consideran más detalladamente los conjuntos *E.tipos* construidos por las reglas de la figura 6.10. Se demuestra que todo tipo t en *E.tipos* es un tipo *factible*; es decir, es posible elegir entre los tipos sobrecargados de los identificadores que aparecen en E de modo que E obtenga el tipo t. La propiedad se cumple para identificadores por declaración, ya que cada elemento de **id**.*tipos* es factible. Para el paso de inducción, considérese el tipo t en *E.tipos*, donde E es $E_1(E_2)$. Según la regla para la aplicación de funciones de la figura 6.10, para un tipo s, s debe estar en E_2.*tipos* y un tipo $s \rightarrow t$ debe estar en E_1.*tipo*. Por inducción, s y $s \rightarrow t$ son tipos factibles para E_2 y E_1, respectivamente. Se deduce que t es un tipo factible para E.

Puede haber varias maneras de llegar a un tipo factible. Por ejemplo, considérese la expresión `f(x)`, donde `f` puede tener los tipos $a \rightarrow c$ y $b \rightarrow c$, y `x` puede tener los tipos a y b. Entonces, `f(x)` tiene tipo c pero `x` puede tener tipo a o b.

PRODUCCIÓN	REGLAS SEMÁNTICAS
$E' \rightarrow E$	$E'.tipos := E.tipos$ $E.único := \textbf{if } E'.tipos = \{t\} \textbf{ then } t \textbf{ else } error\_tipo$ $E'.código := E.código$
$E \rightarrow \textbf{id}$	$E.tipos := busca(\textbf{id}.entrada)$ $E.código := genera(\textbf{id}.lexema\ ':'\ E.único)$
$E \rightarrow E_1\ (\ E_2\ )$	$E.tipos := \{\ s' \mid$ existe una s en $E_2.tipos$ tal que $s \rightarrow s'$ está en $E_1.tipos\ \}$ $t := E.único$ $S := \{\ s \mid s \in E_2.tipos \textbf{ and } s \rightarrow t \in E_1.tipos\ \}$ $E_2.único := \textbf{if } S = \{s\} \textbf{ then } s \textbf{ else } error\_tipo$ $E_1.único := \textbf{if } S = \{s\} \textbf{ then } s \rightarrow t \textbf{ else } error\_tipo$ $E.código := E_1.código \parallel E_2.código \parallel genera(\textbf{'aplica'}\ ':'\ E.único)$

Fig. 6.12. Reducción del conjunto de tipos para una expresión.

La definición dirigida por la sintaxis de la figura 6.12 se obtiene de la de la figura 6.10 añadiendo reglas semánticas para determinar el atributo heredado *único* de E. A continuación se estudia el atributo sintetizado *código* de E.

Como la expresión completa es generada por E', se desea que $E'.tipos$ sea un conjunto que contenga un solo tipo t. Se hereda este tipo único como el valor de $E.único$. De nuevo, el tipo básico *error_tipo* señala un error.

Si una función $E_1(E_2)$ devuelve el tipo t, entonces puede haber un tipo s que sea factible para el argumento E_2; al mismo tiempo, $s \rightarrow t$ es factible para la función. Se utiliza el conjunto S en la regla semántica correspondiente de la figura 6.12 para asegurarse de que haya un único tipo s con dicha propiedad.

Se puede implantar la definición dirigida por la sintaxis de la figura 6.12 realizando dos recorridos en profundidad de un árbol sintáctico para una expresión. Durante el primer recorrido, el atributo *tipos* se sintetiza de manera ascendente. Durante el segundo recorrido, el atributo *único* se propaga de forma descendente, y como se regresa de un nodo, se puede sintetizar el atributo *código*. En la práctica, el comprobador de tipos puede simplemente asociar un tipo único a cada nodo del árbol sintáctico. En la figura 6.12, se genera una notación postfija para ver cómo se podría generar el código intermedio. En la notación postfija, cada identificador y cada caso del operador **aplica** tiene un tipo asociado a él mediante la función *genera*.

6.6 FUNCIONES POLIMORFICAS

Un procedimiento normal permite que las proposiciones de su cuerpo se ejecuten con argumentos de tipos fijos; cada vez que se llama un procedimiento polimórfico, las proposiciones de su cuerpo pueden ejecutarse con argumentos de tipos distintos. El término "polimórfico" también se aplica a cualquier parte de código que pueda ejecutarse con argumentos de tipos distintos, de modo que se puede hablar de funciones, así como de operadores polimórficos.

Los operadores predefinidos para indicar matrices, aplicar funciones y manipular apuntadores son generalmente polimórficos porque no se limitan a una determinada clase de matriz, función o apuntador. Por ejemplo, el manual de referencia de C establece acerca del operador apuntador `&`: "Si el tipo de operando es '...', el tipo de resultado es 'apuntador a ...'". Puesto que cualquier tipo se puede sustituir por "...", el operador `&` en C es polimórfico.

En Ada, las funciones "genéricas" son polimórficas, pero en Ada el polimorfismo es limitado. Como también se ha utilizado el térmimo "genérico" para referirse a funciones sobrecargadas y a la coerción de argumentos de funciones, se evitará usar dicho término.

En esta sección se estudian los problemas que surgen cuando se diseña un comprobador de tipos para un lenguaje con funciones polimórficas. Para analizar el polimorfismo, se ampliará el conjunto de expresiones de tipos a fin de que incluyan expresiones con variables de tipo. La introducción de variables de tipo hace que surjan algunos aspectos algorítmicos relativos a la equivalencia de expresiones de tipos.

¿Por qué las funciones polimórficas?

Las funciones polimórficas resultan atractivas porque facilitan la implantación de algoritmos que manipulan estructuras de datos, independientemente de los tipos de los elementos en la estructura de datos. Por ejemplo, es conveniente tener un programa que determine la longitud de una lista sin que sea necesario conocer los tipos de los elementos de la lista.

Los lenguajes como Pascal exigen una especificación completa de los tipos de los parámetros de funciones, así que una función para determinar la longitud de una lista enlazada de enteros no puede aplicarse a una lista de reales. El código en Pascal de la figura 6.13 es para listas de enteros. La función `longitud` sigue los siguientes enlaces dentro de la lista hasta que se encuentra un enlace con valor `nil`. Aunque la función no depende en absoluto del tipo de información de un nodo, Pascal exige que se declare el tipo del campo `info` cuando se escriba la función `longitud`.

```
type enlace = ↑ nodo;
     nodo   = record
                  info : integer;
                  siguiente : enlace
     end;

function longitud ( aplista : enlace ) : integer;
var lon : integer;
begin
    lon := 0;
    while aplista <> nil do begin
        lon := lon + 1;
        aplista := aplista↑.siguiente
    end;
    longitud := lon
end;
```

Fig. 6.13. Programa en Pascal para encontrar la longitud de una lista.

En un lenguaje con funciones polimórficas, como ML (Milner [1984]), se puede escribir una función `longitud` de modo que se aplique a cualquier tipo de lista, como se muestra en la figura 6.14. La palabra clave `fun` indica que `longitud` es una función recursiva. Las funciones `null` y `tl` están predefinidas; `null` comprueba si una lista está vacía y `tl` devuelve el resto de la lista después de eliminar el

```
fun longitud(aplista) =
        if null(aplista) then 0
        else longitud(tl(aplista)) + 1;
```

Fig. 6.14. Programa en ML para encontrar la longitud de una lista.

primer elemento. Con la definición de la figura 6.14, las dos aplicaciones siguientes de la función `longitud` producen 3:

```
longitud(["dom", "lun", "mar"]);
longitud([10,9,8]);
```

En la primera, `longitud` se aplica a una lista de cadenas; en la segunda, se aplica a una lista de enteros.

Variables de tipo

Las variables que representan expresiones de tipos permiten considerar tipos desconocidos. En el resto de esta sección, se utilizarán las letras griegas α, β, . . . para variables de tipo en las expresiones de tipos.

Una aplicación importante de las variables de tipo es la comprobación del uso consistente de identificadores en un lenguaje que no exija que los identificadores se declaren antes de ser utilizados. Una variable representa el tipo de un identificador no declarado. Por ejemplo, observando el programa se puede saber si el identificador no declarado se utiliza como un entero en una proposición y como una matriz en otra. Dicho uso inconsistente puede considerarse como un error. Por otra parte, si la variable siempre se utiliza como entero, entonces no sólo se ha garantizado un uso consistente: a partir del proceso se ha llegado a la conclusión de cuál debe ser su tipo.

La *inferencia de tipos* es el problema de determinar el tipo de una construcción de lenguaje a partir del modo en que se usa. Este término se aplica a menudo al problema de inferir el tipo de una función a partir de su cuerpo.

Ejemplo 6.8. Las técnicas para inferencia de tipos pueden aplicarse a programas en lenguajes como C y Pascal para completar la información sobre los tipos que falta en el momento de la compilación. El fragmento de código de la figura 6.15 describe el procedimiento `mlista`, que tiene un parámetro `p` que es en sí un procedimiento. Todo lo que sabe al observar la primera línea del procedimiento `mlista` es que `p` es un procedimiento; en concreto, no se conocen el número ni los tipos de los argumentos que toma `p`. C y el manual de referencia de Pascal admiten dichas especificaciones incompletas del tipo de `p`.

El procedimiento `mlista` aplica el parámetro `p` a todos los nodos de una lista enlazada. Por ejemplo, se puede utilizar `p` para inicializar o imprimir el entero contenido en un nodo. A pesar de que no se especifican los tipos de los argumentos de `p`, se puede inferir del uso de `p` en la expresión `p(aplista)` que el tipo de `p` debe ser:

`enlace` → *vacío*

Cualquier llamada de `mlista` con un parámetro de procedimiento que no tenga este tipo es un error. Se puede considerar un procedimiento como una función que no devuelve un valor, de modo que su tipo resultante es *vacío*. □

Las técnicas para la inferencia de tipos y para la comprobación de tipos tienen mucho en común. En ambos casos, se trata de expresiones de tipos que contienen variables. Un comprobador de tipos utilizará en esta sección un razonamiento similar al del ejemplo siguiente para inferir los tipos representados por variables.

```
type enlace = ↑nodo;

procedure mlista ( aplista : enlace; procedure p ) ;
begin
    while aplista <> nil do begin
        p(aplista);
        aplista := aplista↑.siguiente
    end
end;
```

Fig. 6.15. Procedimiento `mlista` con el parámetro de tipo procedimiento `p`.

Ejemplo 6.9. En el siguiente pseudoprograma se puede inferir un tipo para la función polimórfica `desref`. La función `desref` tiene el mismo efecto que el operador de Pascal ↑ para desreferenciar apuntadores.

```
function desref(p);
begin
    return p↑
end;
```

Cuando se observa la primera línea

```
function desref(p);
```

no se sabe nada del tipo de `p`, de modo que se representa mediante la variable de tipo β. Por definición, el operador postfijo ↑ toma un apuntador a un objeto, y devuelve el objeto. Como el operador ↑ se aplica a `p` en la expresión `p`↑, lógicamente `p` debe ser un apuntador a un objeto de tipo desconocido α, así que

$$\beta = \mathit{pointer}(\alpha)$$

donde α es otra variable de tipo. Además, la expresión `p`↑ tiene tipo α, de modo que se puede escribir la expresión de tipo

$$\text{para cualquier tipo } \alpha,\ \mathit{pointer}(\alpha) \rightarrow \alpha \tag{6.3}$$

para el tipo de la función `desref`. □

Un lenguaje con funciones polimórficas

Todo lo aprendido hasta ahora de las funciones polimórficas es que pueden ejecutarse con argumentos de "tipos distintos". Se establecen proposiciones precisas acerca del conjunto de tipos a los que se puede aplicar una función polimórfica con el símbolo $\forall$, que significa "para cualquier tipo". Por tanto,

$$\forall\alpha.\ \mathit{pointer}(\alpha) \rightarrow \alpha \tag{6.4}$$

es la forma de escribir la expresión de tipo (6.3) para el tipo de la función `desref` del ejemplo 6.9. La función polimórfica `longitud` de la figura 6.14 toma una lista

de elementos de cualquier tipo y devuelve un entero, de modo que su tipo se puede escribir:

$$\forall\alpha.\ list\ (\alpha) \rightarrow integer \tag{6.5}$$

En este caso, *list* es un constructor de tipos. Sin el símbolo $\forall$, sólo se pueden dar ejemplos de tipos posibles para el dominio y el rango de `longitud`:

$$list\ (integer) \rightarrow integer$$
$$list\ (list\ (char)) \rightarrow integer$$

Las expresiones de tipos como la (6.5) son las proposiciones más generales para el tipo de una función polimórfica.

El símbolo $\forall$ es el *cuantificador universal*, y se dice que la variable de tipo a la cual se aplica está *acotada* por él. Se puede dar otro nombre a las variables acotadas, a condición de que se haga lo mismo siempre que aparezca la variable. Por tanto, la expresión de tipo

$$\forall\gamma.\ pointer\ (\gamma) \rightarrow \gamma$$

es equivalente a la (6.4). De manera informal, una expresión de tipo que contenga un símbolo $\forall$ se considerará un "tipo polimórfico".

La gramática de la figura 6.16 genera el lenguaje que se usará para comprobar las funciones polimórficas.

$$\begin{aligned}
P &\rightarrow D\ ;\ E \\
D &\rightarrow D\ ;\ D \mid \textbf{id}\ :\ Q \\
Q &\rightarrow \forall\ \textbf{variable de tipo}\ .\ Q \mid T \\
T &\rightarrow T\ '\!\rightarrow'\ T \\
&\quad \mid T \times T \\
&\quad \mid \textbf{constructor_unario}\ (\ T\ ) \\
&\quad \mid \textbf{tipo_básico} \\
&\quad \mid \textbf{variable_de_tipo} \\
&\quad \mid (\ T\ ) \\
E &\rightarrow E\ (\ E\ ) \mid E\ ,\ E \mid \textbf{id}
\end{aligned}$$

Fig. 6.16. Gramática para un lenguaje con funciones polimórficas.

Los programas generados por dicha gramática constan de una secuencia de declaraciones seguida de la expresión E que se va a comprobar. Por ejemplo,

```
desref : ∀α.pointer (α) → α ;
     q : pointer (pointer (integer));
desref(desref(q))
```
(6.6)

Como el no terminal T genera directamente las expresiones de tipos, se minimiza la notación. Los constructores $\rightarrow$ y $\times$ forman tipos de función y producto. Los constructores unarios, representados por **constructor_unario**, permiten escribir tipos como *pointer* (*integer*) y *list* (*integer*). Los paréntesis sólo se utilizan para agrupar tipos. Las

expresiones cuyos tipos deben comprobarse tienen una sintaxis muy sencilla: pueden ser identificadores, secuencias de expresiones que forman una tupla o la aplicación de una función a un argumento.

Las reglas de comprobación de tipos para las funciones polimórficas difieren de tres maneras de las reglas para funciones ordinarias de la sección 6.2. Antes de introducir las reglas, se ilustran estas diferencias estudiando la expresión `desref(desref(q))` del programa (6.6). En la figura 6.17 se muestra un árbol sintáctico para dicha expresión. A cada nodo se encuentran asociadas dos etiquetas. La primera indica la subexpresión representada por el nodo, y la segunda es una expresión de tipo asignada a la subexpresión. Los subíndices e e i distinguen entre los casos exterior e interior de `desref`, respectivamente.

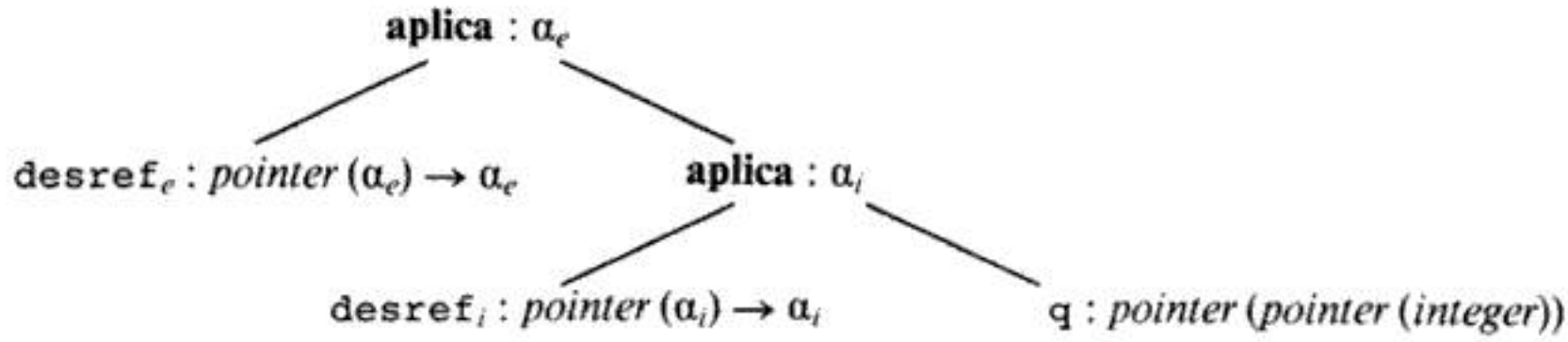


Fig. 6.17. Arbol sintáctico etiquetado para `desref(desref(q))`.

Las diferencias con respecto a las reglas para las funciones ordinarias son:

1. Casos diferentes de una función polimórfica en la misma expresión no deben tener necesariamente argumentos del mismo tipo. En la expresión `desref`$_e$`(desref`$_i$`(q))`, `desref`$_i$ elimina un nivel de indirección de apuntador, así que `desref`$_e$ se aplica a un argumento de un tipo distinto. La implantación de esta propiedad se basa en la interpretación de $\forall\ \alpha$ como "para cualquier tipo α". Cada caso de `desref` tiene su propia visión de lo que representa la variable acotada α de (6.4). Por tanto, a cada caso de `desref` se asigna una expresión de tipo formada por la sustitución de α en (6.4) por una variable nueva y la eliminación del cuantificador $\forall$ en el proceso. En la figura 6.17 las variables nuevas α_e y α_i se utilizan en las expresiones de tipos asignadas a los casos exterior e interior de `desref`, respectivamente.

2. Como pueden aparecer variables en expresiones de tipos, hay que reconsiderar la noción de equivalencia de tipos. Supóngase que E_1 de tipo $s \rightarrow s'$ se aplica a E_2 de tipo t. En lugar de determinar simplemente la equivalencia de s y t, se deben "unificar". La unificación se define más adelante; de manera informal, se determina si s y t se pueden convertir en estructuralmente equivalentes sustituyendo las variables de tipos en s y t por expresiones de tipos. Por ejemplo, en el nodo interior etiquetado con **aplica** en la figura 6.17, la igualdad

   $$\textit{pointer}\ (\alpha_i) = \textit{pointer}\ (\textit{pointer}\ (\textit{integer}))$$

   se cumple si α_i es sustituido por *pointer* (*integer*).

3. Se necesita un mecanismo para registrar el efecto de la unificación de dos expresiones. En general, una variable de tipo puede aparecer en varias expresiones de tipo. Si la unificación de s y s' da como resultado que la variable α represente al tipo t, entonces α debe continuar representando a t durante la comprobación de tipos. Por ejemplo, en la figura 6.17, α_i es el tipo del rango de `desref`$_i$, así que se puede utilizar para el tipo de `desref`$_i$ `(q)`. La unificación del tipo del dominio de `desref`$_i$ con el tipo de `q` afecta por tanto a la expresión de tipo en el nodo interior etiquetado con **aplica**. La otra variable de tipo α_e de la figura 6.17 representa a *integer*.

Sustituciones, casos y unificación

La información acerca de los tipos representados por variables se formaliza definiendo una transformación, de variables de tipo a expresiones de tipos, llamada *sustitución*. La siguiente función recursiva *sust*(t) precisa la noción de aplicar una sustitución S para sustituir todas las variables de tipo en una expresión t. Como de costumbre, se considera el constructor de tipos de función como el constructor "típico".

```
function sust (t:expresión_tipo):expresión_tipo;
begin
    if t es un tipo básico then return t
    else if t es una variable then return S (t)
    else if t es t1 → t2 then return sust (t1) → sust (t2)
end
```

Para simplificar, se escribe $S(t)$ para la expresión de tipo que se obtiene cuando se aplica *sust* a t; el resultado $S(t)$ se denomina *caso* de t. Si la sustitución S no especifica una expresión para la variable α, se supone que $S(\alpha)$ es α; es decir, S es la transformación identidad aplicada a dichas variables.

Ejemplo 6.10. En lo sucesivo, se escribirá $s < t$ para indicar que s es un caso de t:

$$\begin{aligned} \textit{pointer}(\textit{integer}) &< \textit{pointer}(\alpha) \\ \textit{pointer}(\textit{real}) &< \textit{pointer}(\alpha) \\ \textit{integer} \rightarrow \textit{integer} &< \alpha \rightarrow \alpha \\ \textit{pointer}(\alpha) &< \beta \\ \alpha &< \beta \end{aligned}$$

Sin embargo, en el ejemplo siguiente, la expresión de tipo de la izquierda no es un caso de la derecha (por la razón que se indica):

integer	*real*	Las sustituciones no se aplican a tipos básicos.
integer → *real*	$\alpha \rightarrow \alpha$	Sustitución inconsistente para α.
integer → α	$\alpha \rightarrow \alpha$	Se deben sustituir todos los casos de α. □

Dos expresiones de tipo t_1 y t_2 se *unifican* si existe alguna sustitución S tal que $S(t_1) = S(t_2)$. En la práctica, interesa el *unificador más general*, que es la sustitución que menos limitaciones exige a las variables dentro de las expresiones. Más

exactamente, el unificador más general de las expresiones t_1 y t_2 es una sustitución S con las siguientes propiedades:

1. $S(t_1) = S(t_2)$ y
2. para cualquier otra sustitución S' tal que $S'(t_1) = S'(t_2)$, la sustitución S' es un caso de S (es decir, para cualquier t, $S'(t)$ es un caso de $S(t)$).

En lo sucesivo, "unifica" hará referencia al unificador más general.

Comprobación de funciones polimórficas

Las reglas para comprobar expresiones generadas por la gramática de la figura 6.16 se escribirán mediante las siguientes operaciones en una representación con grafos de los tipos.

1. *nuevas* (t) sustituye las variables acotadas en la expresión de tipo t por variables nuevas y devuelve un apuntador a un nodo que representa la expresión de tipo obtenida. En el proceso se eliminan todos los símbolos $\forall$ en t.
2. *unifica* (m, n) unifica las expresiones de tipos representadas por los nodos apuntados por m y n. Como efecto secundario conserva la sustitución que convierte a las expresiones en equivalentes. Si falla el proceso de unificación de las expresiones, falla todo el proceso de comprobación de tipos[4].

Las hojas individuales y los nodos interiores en el grafo de tipos se construyen utilizando las operaciones *hazhoja* y *haznodo* similares a las de la sección 5.2. Es necesario que haya una hoja única para cada variable de tipo, pero no es necesario que otras expresiones estructuralmente equivalentes tengan nodos únicos.

La operación *unifica* se basa en la siguiente formulación de unificación y sustituciones, fundada en la teoría de los grafos. Supóngase que los nodos m y n representan a las expresiones e y f, respectivamente. Se dice que los nodos m y n son *equivalentes bajo* la sustitución S si $S(e) = S(f)$. Puede reformularse el problema de encontrar el unificador más general S como el problema de agrupar, en conjuntos, los nodos que deben ser equivalentes bajo S. Para que las expresiones sean equivalentes, sus raíces deben ser equivalentes. Asimismo, dos nodos m y n son equivalentes si, y sólo si, representan al mismo operador y sus hijos correspondientes son equivalentes.

En la siguiente sección se introducirá un algoritmo para unificar un par de expresiones. El algoritmo conserva los conjuntos de nodos equivalentes bajo las sustituciones realizadas.

En la figura 6.18 se muestran las reglas para la comprobación de tipos para expresiones. No se muestra cómo se procesan las declaraciones. Conforme se examinan las expresiones de tipos generadas por los no terminales T y Q, *hazhoja* y *haznodo* añaden nodos al grafo de tipos, siguiendo la construcción de GDA de la

---

[4] La razón para interrumpir el proceso de comprobación de tipos es que puede que se registren los efectos secundarios de algunas unificaciones antes de que se detecte el fallo. Puede implantarse la recuperación de errores si se retrasan los efectos secundarios de la operación *unifica* hasta que las expresiones se hayan unificado con éxito.

sección 5.2. Cuando se declara un identificador, el tipo de la declaración se guarda en la tabla de símbolos en la forma de un apuntador al nodo que representa al tipo. En la figura 6.18 se hace referencia a este apuntador como el atributo sintetizado **id**.*tipo*. Como se ha mencionado anteriormente, la operación *nuevas* elimina los símbolos $\forall$ al sustituir las variables acotadas por variables nuevas. La acción asociada con la producción $E \rightarrow E_1, E_2$ asigna *E.tipo* al producto de los tipos de E_1 y E_2.

$E \rightarrow E_1$ (E_2)	{ $p :=$ *hazhoja* (*vartiponueva*); *unifica* (E_1.*tipo*, *haznodo* ('$\rightarrow$', E_2.*tipo*, p)); E.*tipo* $:= p$ }
$E \rightarrow E_1$, E_2	{ E.*tipo* $:=$ *haznodo* ('$\times$', E_1.*tipo*, E_2.*tipo*) }
$E \rightarrow$ **id**	{ E.*tipo* $:=$ *nuevas* (**id**.tipo) }

Fig. 6.18. Esquema de traducción para la comprobación de funciones polimórficas.

La regla de comprobación de tipos para la aplicación de función $E \rightarrow E_1$ (E_2) es necesaria al considerar el caso donde tanto E_1.*tipo* como E_2.*tipo* son variables de tipo, por ejemplo, E_1.*tipo* $= \alpha$ y E_2.*tipo* $= \beta$. Aquí, E_1.*tipo* debe ser una función tal que para un tipo desconocido γ, se tenga $\alpha = \beta \rightarrow \gamma$. En la figura 6.18, se crea una variable de tipo nueva correspondiente a γ y E_1.*tipo* se unifica con E_2.*tipo* $\rightarrow \gamma$. Cada llamada a *vartiponueva* devuelve una variable de tipo nueva, *hazhoja* construye una hoja para ella, y *haznodo* construye un nodo que representa a la función que se va a unificar con E_1.*tipo*. Cuando la unificación haya sido un éxito, la nueva hoja representa el tipo del resultado.

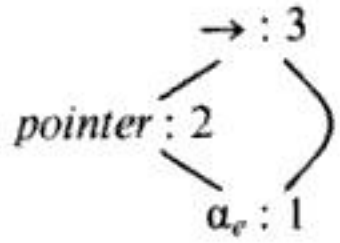
Las reglas de la figura 6.18 se ilustran mediante el análisis detallado de un sencillo ejemplo. Se resume el trabajo del algoritmo escribiendo las expresiones de tipo asignadas a cada subexpresión, como se muestra en la figura 6.19. A cada aplicación de función, la operación *unifica* puede tener el efecto secundario de registrar una expresión de tipo para alguna de las variables de tipo. Dichos efectos secundarios son propuestos en la columna correspondiente a una sustitución en la figura 6.19.

Ejemplo 6.11. La comprobación de tipos para la expresión `desref`$_e$`(desref`$_i$`(q))` en el programa (6.6) avanza de manera ascendente desde las hojas. Una vez más, los

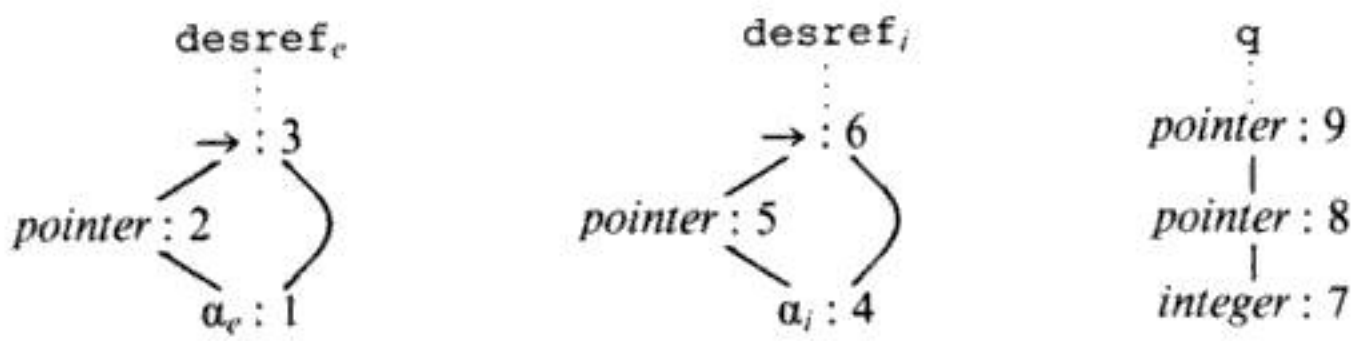
EXPRESIÓN : TIPO	SUSTITUCIÓN
`q` : *pointer* (*pointer* (*integer*))	
`desref`$_i$: *pointer* (α_i) $\rightarrow \alpha_i$	
`desref`$_i$`(q)` : *pointer* (*integer*)	$\alpha_i =$ *pointer* (*integer*)
`desref`$_e$: *pointer* (α_e) $\rightarrow \alpha_e$	
`desref`$_e$`(desref`$_i$`(q))` : *integer*	$\alpha_e =$ *integer*

Fig. 6.19 Resumen de la determinación ascendente de tipos.

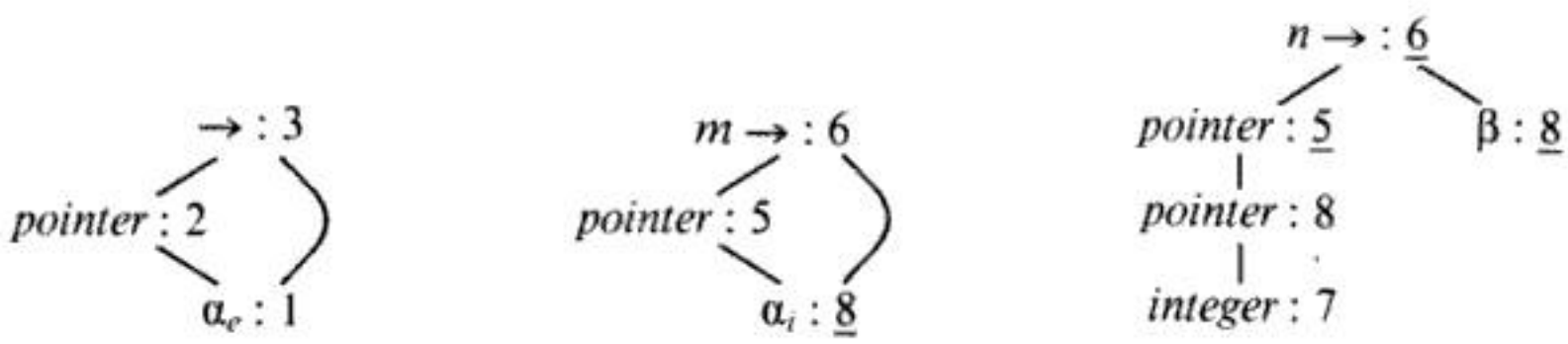
subíndices e e i distinguen entre casos de `desref`. Cuando se considera la subexpresión $\texttt{desref}_e$, *nuevas* construye los siguientes nodos utilizando una nueva variable de tipo α_e.



El número en un nodo indica la clase de equivalencia a la que pertenece el nodo. A continuación se muestra la parte del grafo de tipos para los tres identificadores. Las líneas punteadas indican que los nodos numerados con 3, 6 y 9 son para $\texttt{desref}_e$, $\texttt{desrer}_i$ y `q`, respectivamente.



La aplicación de función $\texttt{desref}_i\texttt{(q)}$ se comprueba construyendo un nodo n para una función que va del tipo de `q` a una nueva variable de tipo β. Esta función se unifica con éxito con el tipo de $\texttt{desref}_i$ representado por el nodo m de más abajo. Antes de que se unifiquen los nodos m y n, cada nodo tiene un número distinto. Después de la unificación, los nodos equivalentes son los nodos de abajo con el mismo número; los números modificados están subrayados:



Obsérvese que tanto el nodo para α_i como para *pointer* (*integer*) están numerados con 8, es decir, que α_i se unifica con esta expresión de tipo, como se muestra en la figura 6.19. Por tanto, α_e se unifica con *integer*. □

El ejemplo siguiente relaciona la inferencia de tipos de las funciones polimórficas en ML con las reglas para la comprobación de tipos de la figura 6.18. La sintaxis de las definiciones de funciones en ML viene dada por

fun id$_0$ (**id**$_1$, . . . **id**$_k$) = E ;

donde **id**$_0$ representa el nombre de la función e **id**$_1$, . . . , **id**$_k$ representan sus parámetros. Para simplificar, se supone que la sintaxis de la expresión E es como en la figura 6.16, y que los únicos identificadores en E son el nombre de la función, sus parámetros y las funciones predefinidas.

El enfoque es una formalización del ejemplo 6.9, donde se infería un tipo polimórfico para `desref`. Las variables de tipo nuevas se construyen para el nombre de la función y sus parámetros. Las funciones predefinidas tienen generalmente tipos polimórficos; todas las variables de tipos que aparezcan en dichos tipos están acotadas por cuantificadores $\forall$. Después se comprueba que los tipos de las expresiones $\mathbf{id}_0(\mathbf{id}_1, \ldots, \mathbf{id}_k)$ y E concuerden. Si concuerdan, se habrá inferido un tipo para el nombre de la función. Por último, todas las variables en el tipo inferido están acotadas por cuantificadores $\forall$ para dar el tipo polimórfico de la función.

Ejemplo 6.12. Recuérdese la función en ML de la figura 6.14 para determinar la longitud de una lista

```
fun longitud(aplista) =
    if null(aplista) then 0
    else longitud(tl(aplista)) + 1;
```

Se introducen las variables de tipo β y γ para los tipos de `longitud` y `aplista`, respectivamente. Se observa que el tipo de `longitud(aplista)` coincide con el de la expresión que forma al cuerpo de la función y que `longitud` debe tener el tipo

para cualquier tipo α, $list(\alpha) \rightarrow integer$

así que el tipo de `longitud` es

$\forall\alpha.\ list(\alpha) \rightarrow integer$

Con mayor detalle, se realiza el programa que se muestra en la figura 6.20, al que se pueden aplicar las reglas para la comprobación de tipos de la figura 6.18. Las declaraciones en el programa asocian las variables de tipo nuevas β y γ con `longitud` y `aplista`, y hacen explícitos los tipos de las operaciones predefinidas. Los condicionales se escriben al estilo de la figura 6.16 aplicando el operador polimór-

```
longitud : β;
 aplista : γ;
      if : ∀α. boolean × α × α → α;
    null : ∀α. list (α) → boolean;
      tl : ∀α. list (α) → list (α);
       0 : integer;
       1 : integer;
       + : integer × integer → integer;
   match : ∀α. α × α → α;

 match(
       longitud(aplista)
       if( null(aplista), 0, longitud(tl(aplista)) + 1 )
      )
```

Fig. 6.20. Declaraciones seguidas de la expresión por comprobar.

LINEA	EXPRESIÓN : TIPO	SUSTITUCIÓN
(1)	`aplista` : γ	
(2)	`longitud` : β	
(3)	`longitud(aplista)` : δ	$\beta = \gamma \rightarrow \delta$
(4)	`aplista` : γ	
(5)	`null` : $list\,(\alpha_n) \rightarrow boolean$	
(6)	`null(aplista)` : $boolean$	$\gamma = list\,(\alpha_n)$
(7)	`0` : $integer$	
(8)	`aplista` : $list\,(\alpha_n)$	
(9)	`tl` : $list\,(\alpha_t) \rightarrow list\,(\alpha_t)$	
(10)	`tl(aplista)` : $list\,(\alpha_n)$	$\alpha_t = \alpha_n$
(11)	`longitud` : $list\,(\alpha_n) \rightarrow \delta$	
(12)	`longitud(tl(aplista))` : δ	
(13)	`1` : $integer$	
(14)	`+` : $integer \times integer \rightarrow integer$	
(15)	`longitud(tl(aplista))+1` : $integer$	$\delta = integer$
(16)	`if` : $boolean \times \alpha_i \times \alpha_i \rightarrow \alpha_i$	
(17)	`if( . . . )` : $integer$	$\alpha_i = integer$
(18)	`match` : $\alpha_m \times \alpha_m \rightarrow \alpha_m$	
(19)	`match( . . . )` : $integer$	$\alpha_m = integer$

Fig. 6.21. Inferencia del tipo $list\,(\alpha_n) \rightarrow integer$ para `longitud`.

fico `if` a tres operandos, que representan las expresiones que van a ser examinadas, la parte de **then** y la parte de **else**; la declaración indica que la parte de **then** y la de **else** pueden ser de cualquier tipo que concuerde, que será, asimismo, el tipo del resultado.

Es evidente que `longitud(aplista)` debe tener el mismo tipo que el cuerpo de la función; esta comprobación se codifica usando el operador `match` (comprobar la concordancia). El uso de `match` permite técnicamente que todas las comprobaciones se realicen con un programa al estilo de la figura 6.16.

En la figura 6.21 se resume el resultado de aplicar las reglas para la comprobación de tipos de la figura 6.18 al programa de la figura 6.20. Las variables nuevas introducidas por la operación *nuevas* aplicadas a los tipos polimórficos de las operaciones predefinidas se distinguen mediante subíndices en α. En la línea (3) se observa que `longitud` debe ser una función de γ a algún tipo desconocido δ. Más adelante, cuando se comprueba la subexpresión `null(aplista)`, se observa en la línea (6) que γ se unifica con $list\,(\alpha_n)$, donde α_n es un tipo desconocido. Llegados a este punto, se sabe que el tipo de `longitud` debe ser

para cualquier tipo α_n, $list(\alpha_n) \rightarrow \delta$

En ocasiones, cuando se compruebe la suma en la línea (15) —se ha tomado la libertad de escribir + entre sus argumentos para una mayor claridad— δ se unifica con *integer*.

Cuando se termina la comprobación, la variable de tipo α_n permanece en el tipo de `longitud`. Como no se hizo ninguna suposición acerca de α_n, se puede sustituir por cualquier tipo cuando se utilice la función. Por tanto, se le considera una variable acotada y se escribe

$$\forall \alpha_n.\ list(\alpha_n) \rightarrow integer$$

para el tipo de `longitud`. □

6.7 UN ALGORITMO PARA LA UNIFICACION

De manera informal, la unificación es el problema de determinar si dos expresiones e y f pueden convertirse en idénticas sustituyendo expresiones por las variables que aparezcan en e y f. Comprobar la igualdad de expresiones es un caso especial de unificación; si e y f tienen constantes pero no variables, entonces e y f se unifican si, y sólo si, son idénticas. El algoritmo de unificación de esta sección puede aplicarse a grafos con ciclos, así que se puede utilizar para comprobar si hay equivalencia estructural en tipos circulares[5].

La unificación fue definida en la última sección en términos de una función S, llamada sustitución, que transforma variables en expresiones. Se escribe $S(e)$ para la expresión que se obtiene al sustituir cada variable α en e por $S(\alpha)$. S es un unificador para e y f si $S(e) = S(f)$. El algoritmo de esta sección determina una sustitución que es el unificador más general de un par de expresiones.

Ejemplo 6.13. Para tener una perspectiva de los unificadores más generales, considérense las dos expresiones de tipos

$$((\alpha_1 \rightarrow \alpha_2) \times list\,(\alpha_3)) \rightarrow list(\alpha_2)$$
$$((\alpha_3 \rightarrow \alpha_4) \times list\,(\alpha_3)) \rightarrow \alpha_5$$

Dos unificadores, S y S', para estas expresiones son:

x	$S(x)$	$S'(x)$
α_1	α_3	α_1
α_2	α_2	α_1
α_3	α_3	α_1
α_4	α_2	α_1
α_5	$list\,(\alpha_2)$	$list\,(\alpha_1)$

Estas sustituciones transforman e y f como sigue:

$$S(e) = S(f) = ((\alpha_3 \rightarrow \alpha_2) \times list\,(\alpha_3)) \rightarrow list\,(\alpha_2)$$
$$S'(e) = S'(f) = ((\alpha_1 \rightarrow \alpha_1) \times list\,(\alpha_1)) \rightarrow list\,(\alpha_1)$$

---

[5] En algunas aplicaciones, es un error unificar una variable con una expresión que contenga dicha variable. El algoritmo 6.1 permite dichas sustituciones.

La sustitución S es el unificador más general de e y f. Obsérvese que $S'(e)$ es un caso de $S(e)$ porque se puede sustituir α_1 por ambas variables en $S(e)$. Sin embargo, a la inversa no se cumple, porque la misma expresión se debe sustituir por cada caso de α_1 en $S'(e)$, así que no se puede obtener $S(e)$ sustituyendo por la variable α_1 en $S'(e)$. □

Cuando las expresiones que van a unificarse están representadas por árboles, el número de nodos en el árbol para la expresión sustituida $S(e)$ puede ser exponencial en función del número de nodos de los árboles para e y f, aunque S sea el unificador más general. Sin embargo, esta explosión de tamaño no tiene por qué ocurrir si se utilizan grafos en lugar de árboles para representar expresiones y sustituciones.

Se implantará la formulación de la teoría de grafos de la unificación, que también se estudia en la última sección. El problema es el de agrupar en conjuntos nodos que deben ser equivalentes bajo el unificador más general de dos expresiones. Las dos expresiones del ejemplo 6.13 se representan con los dos nodos etiquetados con $\rightarrow$:1 de la figura 6.22. Los enteros que aparecen en los nodos indican las clases

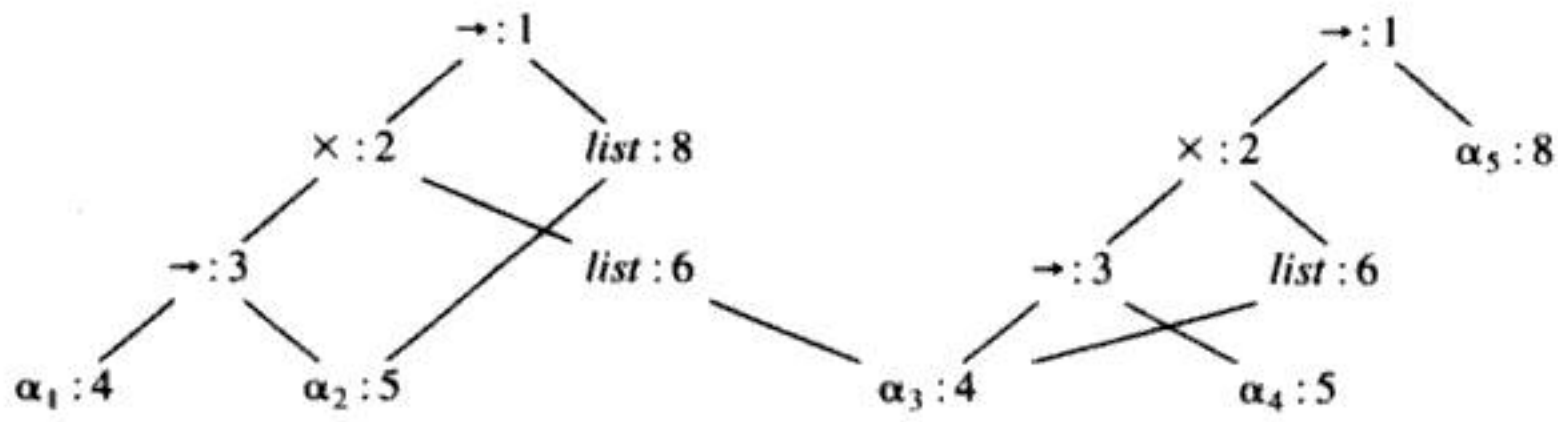


Fig. 6.22. Clases de equivalencia después de la unificación.

de equivalencia a las que pertenecen los nodos después de que se unifiquen los nodos numerados con 1. Estas clases de equivalencia tienen la propiedad de que todos los nodos interiores de la clase son para el mismo operador. Los hijos correspondientes de los nodos interiores en una clase de equivalencia también son equivalentes.

Algoritmo 6.1. Unificación de un par de nodos en un grafo.

Entrada. Un grafo y un par de nodos m y n que deben unificarse.

Salida. El valor booleano **true** si las expresiones representadas por los nodos m y n se unifican; de lo contrario, **false**. La versión de la operación *unifica* necesaria para las reglas de comprobación de tipos de la figura 6.18 se obtiene si la función de este algoritmo se modifica para que falle en lugar de que devuelva **false**.

Método. Un nodo se representa mediante un registro como en la figura 6.23 con campos para un operador binario y apuntadores a los hijos izquierdo y derecho.

Los conjuntos de nodos equivalentes se mantienen utilizando el campo *conjunto*. Se elige un nodo en cada clase de equivalencia como representante único de la clase, haciendo que su campo *conjunto* contenga un apuntador a *nil* (vacío). Los campos *conjunto* de los nodos restantes de la clase de equivalencia apuntarán (tal

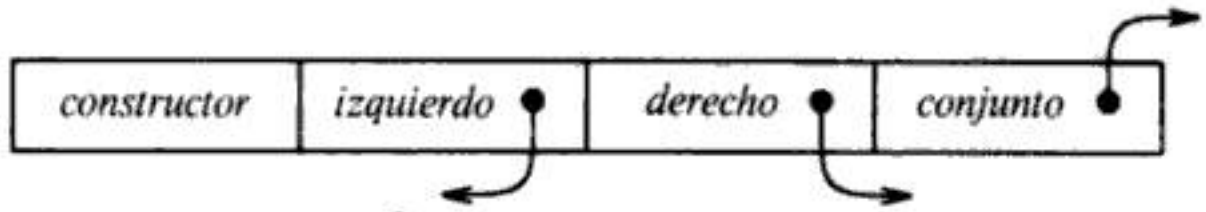


Fig. 6.23. Estructura de datos para un nodo.

vez indirectamente a través de otros nodos del conjunto) al representante. Inicialmente, cada nodo *n* está en una clase de equivalencia solo, y *n* es su propio nodo representante.

```
function unifica (m, n, nodo) : boolean;
begin
    s := encuentra (m);
    t := encuentra (n);
    if s = t then
        return true
    else if s y t son nodos que representan al mismo tipo básico then
        return true
    else if s es un nodo interior con hijos s1 y s2 and
            t es un nodo interior con hijos t1 y t2 then begin
        unión (s, t);
        return unifica (s1, t1) and unifica (s2, t2)
    end
    else if s o t representa a una variable then begin
        unión (s, t);
        return true
    end
    else return false
        /* los nodos interiores con operadores diferentes no se pueden unificar */
end
```

Fig. 6.24. Algoritmo de unificación.

El algoritmo de unificación de la figura 6.24 utiliza las siguientes dos operaciones sobre los nodos:

1. *encuentra* (*n*) devuelve el nodo representante de la clase de equivalencia que en ese momento contenga el nodo *n*.
2. *unión*(*m*, *n*) fusiona las clases de equivalencia que contengan los nodos *m* y *n*. Si uno de los representantes de las clases de equivalencia de *m* y *n* es un nodo no variable, *unión* convierte al nodo no variable en representante de la clase de equivalencia fusionada; de lo contrario, *unión* convierte a uno o al otro de los representantes originales en el nuevo representante. Esta asimetría en la especificación de *unión* es importante porque una variable no se puede utilizar como

representante de una clase de equivalencia para una expresión que contenga un constructor de tipos o un tipo básico. De lo contrario, dos expresiones no equivalentes se podrían unificar por medio de esa variable.

La operación de *unión* en conjuntos se implanta modificando simplemente el campo *conjunto* del representante de una clase de equivalencia para que apunte al representante de la otra. Para encontrar la clase de equivalencia a la que pertenece un nodo, se siguen los apuntadores *conjunto* de los nodos hasta alcanzar el representante (el nodo con un apuntador a **nil** en el campo de conjunto).

Obsérvese que el algoritmo de la figura 6.24 utiliza $s = encuentra\,(m)$ y $t = encuentra\,(n)$ en lugar de m y n, respectivamente. Los nodos representantes s y t son iguales si m y n están en la misma clase de equivalencia. Si s y t representan el mismo tipo básico, la llamada a *unifica* (m, n) devuelve **true**. Si tanto s como t son nodos interiores para un constructor de tipos binario, se prueba a fusionar sus clases de equivalencia y después se comprueba recursivamente que sus hijos respectivos sean equivalentes. Al hacer primero la fusión, se reduce el número de clases de equivalencia antes de comprobar recursivamente los hijos, así que el algoritmo termina.

Se implanta la sustitución de una expresión por una variable añadiendo la hoja para la variable a la clase de equivalencia que contenga el nodo para la expresión. Si m o n es una hoja para una variable que forma parte de una clase de equivalencia que contiene un nodo que representa una expresión con un constructor de tipos o un tipo básico, entonces *encuentra* devolverá a un representante que refleje dicho constructor de tipos o tipo básico, de modo que una variable no se pueda unificar con dos expresiones distintas. □

Ejemplo 6.14. Ya se ha mostrado el grafo inicial para las dos expresiones del ejemplo 6.13 en la figura 6.25 con cada nodo numerado y en su propia clase de equivalencia. Para calcular *unifica* (1, 9), el algoritmo observa que tanto el nodo 1 como el 9 representan el mismo operador, así que fusiona 1 y 9 en la misma clase de equivalencia y llama *unifica* (2, 10) y *unifica* (8, 14). El cálculo de *unifica* (1, 9) es el grafo mostrado en la figura 6.22. □

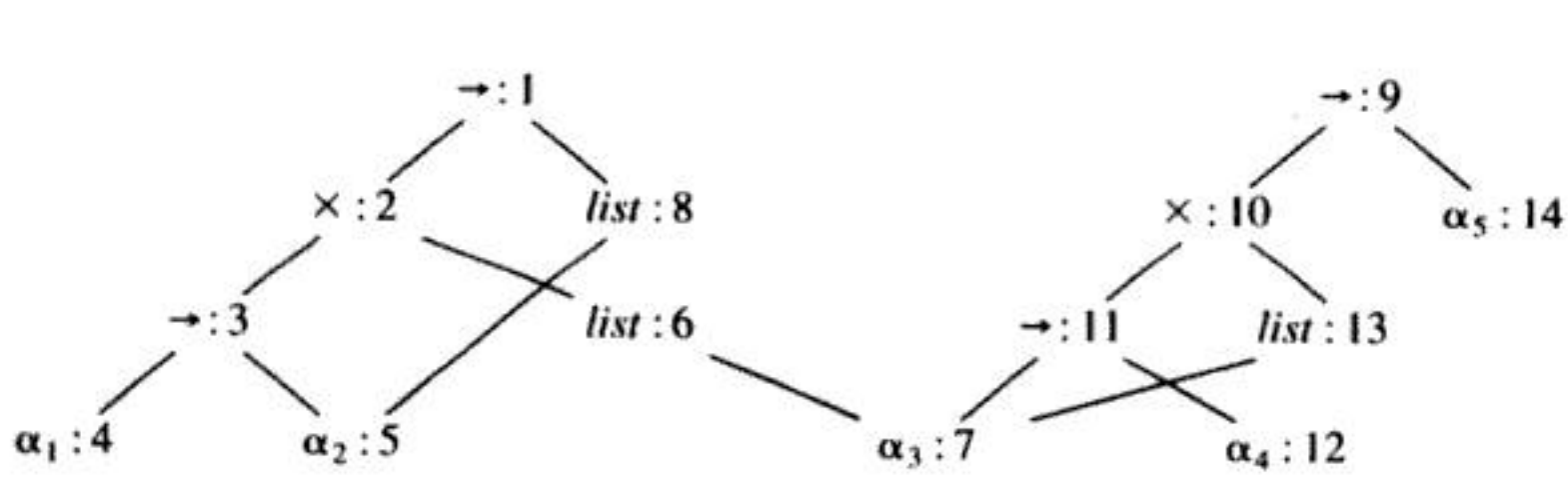


Fig. 6.25. GDA inicial con cada nodo en su propia clase de equivalencia.

Si el algoritmo 6.1 devuelve **true**, se puede construir una sustitución S que actúe de unificador, de la siguiente manera. Cada nodo n del grafo obtenido representará a la expresión asociada con *encuentra* (n). Por tanto, para cada variable α, *encuentra* (α) entrega el nodo n que es el representante de la clase de equivalencia de α. La

expresión representada por n es $S(\alpha)$. Por ejemplo, en la figura 6.22, se ve que el representante para α_3 es el nodo 4, que representa a α_1. El representante para α_5 es el nodo 8 que representa a *list* (α_2).

Ejemplo 6.15. Se puede utilizar el algoritmo 6.1 para comprobar la equivalencia estructural de las dos expresiones de tipos

e : *real* → e
f : *real* → (*real* → f)

En la figura 6.26 se muestran los grafos de tipos para estas expresiones. Por conveniencia, cada nodo ha sido numerado.

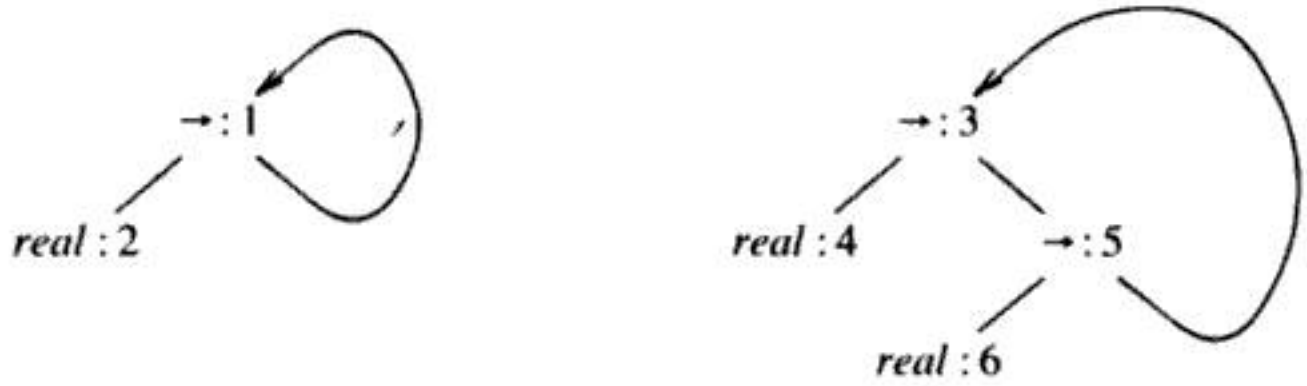


Fig. 6.26. Grafo para dos tipos circulares.

Se llama a *unifica* (1, 3) para que compruebe si existe equivalencia estructural en estas dos expresiones. El algoritmo fusiona los nodos 1 y 3 en una clase de equivalencia, y recursivamente llama a *unifica* (2, 4) y *unifica* (1, 5). Como 2 y 4 representan el mismo tipo básico, la llamada a *unifica* (2, 4) devuelve **true**. La llamada a *unifica* (1, 5) añade 5 a la clase de equivalencia de 1 y 3, y recursivamente llama a *unifica* (2, 6) y *unifica* (1, 3).

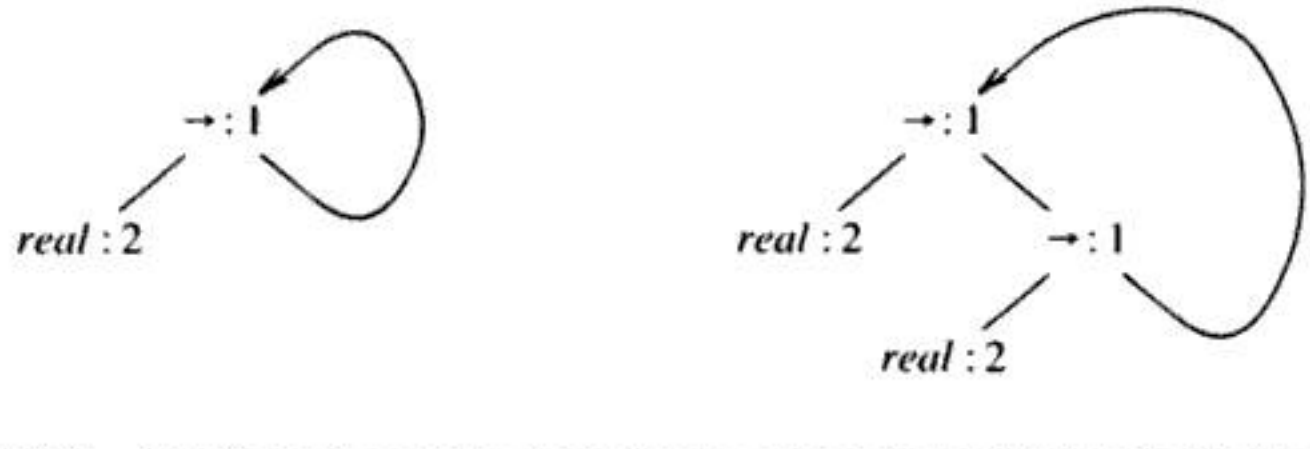


Fig. 6.27. Grafo de tipos que muestra las clases de equivalencia de los nodos.

La llamada a *unifica* (2, 6) devuelve **true** porque 2 y 6 también representan el mismo tipo básico. La segunda llamada de *unifica* (1, 3) termina porque ya se han fusionado los nodos 1 y 3 en la misma clase de equivalencia. El algoritmo por consiguiente termina, y devuelve **true** para indicar que las dos expresiones son equivalentes. En la figura 6.27 se muestran las clases de equivalencia de los nodos obtenidas, donde los nodos con el mismo entero están en la misma clase de equivalencia. □

EJERCICIOS

6.1 Escríbanse expresiones de tipos para los siguientes tipos.

a) Una matriz de apuntadores a reales, donde los índices de la matriz varían de 1 a 100.

b) Una matriz bidimensional de enteros (es decir, una matriz de matrices) cuyas filas estén indizadas de 0 a 9 y cuyas columnas estén indizadas de − 10 a 10.

c) Funciones cuyos dominios sean funciones desde enteros a apuntadores a enteros y cuyos rangos sean registros que consten de un entero y un carácter.

6.2 Supóngase que se tienen las siguientes declaraciones en C:

```
typedef struct {
    int a, b;
    } NODO, *APNODO;
NODO aa[100];
APNODO bb(x, y) int x; NODO y { ... }
```

Escríbanse expresiones de tipo para los tipos de `aa` y `bb`.

6.3 La siguiente gramática define listas de listas de literales. La interpretación de los símbolos es la misma que la de la gramática de la figura 6.3 además del tipo **list**, que indica una lista de elementos del tipo T siguiente.

$$
\begin{array}{lcl}
P & \rightarrow & D \ ; \ E \\
D & \rightarrow & D \ ; \ D \mid \textbf{id} \ : \ T \\
T & \rightarrow & \textbf{list of} \ T \mid \textbf{char} \mid \textbf{integer} \\
E & \rightarrow & (\ L \ ) \mid \textbf{literal} \mid \textbf{num} \mid \textbf{id} \\
L & \rightarrow & E \ , \ L \mid E
\end{array}
$$

Escríbanse reglas de traducción similares a las de la sección 6.2 para determinar los tipos de las expresiones (E) y listas (L).

6.4 Añádase a la gramática del ejercicio 6.3 la producción

$$E \rightarrow \textbf{nil}$$

que indica que una expresión puede ser la lista nula (vacía). Revísense las reglas en su respuesta al ejercicio 6.2 para tener en cuenta que **nil** puede representar una lista vacía de elementos de cualquier tipo.

6.5 Utilizando el esquema de traducción de la sección 6.2, calcúlense los tipos de las expresiones de los siguientes fragmentos de programa. Muéstrense los tipos en cada nodo del árbol de análisis sintáctico.

a)
```
c: char; i: integer;
c mod i mod 3
```

b)
```
p: ↑integer; a: array [10] of integer;
a[p↑]
```

```
c) f: integer → boolean;
   i: integer; j: integer; k: integer;
   while f(i) do
        k := i;
        i := j mod i;
        j := k
```

6.6 Modifíquese el esquema de traducción para la comprobación de expresiones de la sección 6.2 para que imprima un mensaje descriptivo cuando se detecte un error y para que continúe la comprobación como si hubiera aparecido el tipo previsto.

6.7 Reformúlense las reglas para la comprobación de tipos para las expresiones de la sección 6.2 de modo que se refieran a nodos en una representación por medio de grafos de las expresiones de tipos. Las reglas reformuladas deben utilizar estructuras de datos y operaciones basadas en un lenguaje como Pascal. Utilícese la equivalencia estructural de las expresiones de tipos cuando:
a) las expresiones de tipos estén representadas por árboles, como en la figura 6.2, y
b) el grafo de tipos sea un GDA con un nodo único por cada expresión de tipo.

6.8 Modifíquese el esquema de traducción de la figura 6.5 para que sirva para:
a) Proposiciones que tengan valores. El valor de una asignación es el valor de la expresión a la derecha del signo :=. El valor de una proposición condicional o una proposición **while** es el valor del cuerpo de la proposición; el valor de una lista de proposiciones es el valor de la última proposición de la lista.
b) Expresiones booleanas. Añádanse producciones para los operadores lógicos **and**, **or** y **not**, y para operadores de comparación (<, etcétera). Después añádanse reglas de traducción adecuadas que proporcionen el tipo de estas expresiones.

6.9 Generalícense las reglas de comprobación de tipos para las funciones que se dan al final de la sección 6.2 para que maneje funciones *n*-arias.

6.10 Supóngase que los nombres de tipos `enlace` y `nodo` se definen como en la sección 6.3. ¿Cuáles de las siguientes expresiones son estructuralmente equivalentes?
i) `enlace`.
ii) *pointer* (`nodo`).
iii) *pointer* (`enlace`).
iv) *pointer* (*record* ((`info` × *integer*) × (`siguiente` × *pointer* (`nodo`))).

6.11 Reformúlese el algoritmo para comprobar la equivalencia estructural de la figura 6.6 de modo que los argumentos de *equivest* sean apuntadores a nodos de un GDA.

6.12 Considérese la codificación de las expresiones de tipos limitadas como secuencias de bits del ejemplo 6.1. En Johnson [1979], los campos de dos bits

para constructores aparecían en orden inverso, con el campo para el constructor más externo después de los 4 bits del tipo básico; por ejemplo,

EXPRESIÓN DE TIPO	CODIFICACIÓN
char	000000 0001
freturns (*char*)	000011 0001
pointer (*freturns* (*char*))	001101 0001
array (*pointer* (*freturns* (*char*)))	110110 0001

Utilizando los operadores de C escríbase código para construir la representación de *array* (*t*) a partir de la de *t* y viceversa, siendo la codificación como en:

a) Johnson [1979].
b) Ejemplo 6.1.

6.13 Supóngase que el tipo de cada identificador es un subrango de enteros. Para expresiones con los operadores `+`, `-`, `*`, `div` y `mod`, como en Pascal, escríbanse reglas para la comprobación de tipos que asignen a cada subexpresión el subrango en el que debe estar su valor.

6.14 Dése un algoritmo para comprobar la equivalencia de los tipos de C (véase Ejemplo 6.4).

6.15 Algunos lenguajes, como PL/I, coercionarán un valor booleano a un entero, con **true** identificado con 1 y **false** con 0. Por ejemplo, `3<4<5` se agrupa `(3<4)<5`, y tiene el valor **true** (o 1), porque `3<4` tiene el valor 1, y `1<5` es **true**. Escríbanse reglas de traducción para las expresiones booleanas que realicen dicha coerción. Utilícense proposiciones condicionales en el lenguaje intermedio para asignar valores enteros a temporales que representen el valor de una expresión booleana, cuando sea necesario.

6.16 Generalícense los algoritmos de (a) figura 6.9 y (b) figura 6.12 a expresiones con los constructores de tipos *array*, *pointer* y producto cartesiano.

6.17 ¿Cuáles de las siguientes expresiones de tipo recursivas son equivalentes?

`e1` = *integer* → `e1`
`e2` = *integer* → (*integer* → `e2`)
`e3` = *integer* → (*integer* → `e1`)

6.18 Usando las reglas del ejemplo 6.6, determínense cuáles de las siguientes expresiones tienen tipos únicos. Supóngase que `z` es un número complejo.

a) `1 * 2 * 3`
b) `1 * (z * 2)`
c) `(1 * z) * z`

6.19 Supóngase que se permiten las conversiones de tipos del ejemplo 6.6. ¿Imponiendo qué condiciones a los tipos de *a*, *b* y *c* (entero o complejo), tendrá tipo único la expresión (*a* * *b*) * *c*?

6.20 Exprésense, utilizando variables de tipos, los tipos de las siguientes funciones.

a) La función *ref* que toma como argumento un objeto de cualquier tipo y devuelve un apuntador a dicho objeto.
b) Una función que tome como argumento una matriz indizada por enteros, con elementos de cualquier tipo, y devuelva una matriz cuyos elementos sean los objetos apuntados por los elementos de la matriz dada.

6.21 Encuéntrese el unificador más general de las expresiones de tipos

i) $(pointer\,(\alpha)) \times (\beta \rightarrow \gamma)$
ii) $\beta \times (\gamma \rightarrow \delta)$

¿Qué pasaría si δ en ii) fuera α?

6.22 Encuéntrese el unificador más general para cada par de expresiones de la siguiente lista, o determínese que no existe ninguno.

a) $\alpha_1 \rightarrow (\alpha_2 \rightarrow \alpha_1)$
b) $array\,(\beta_1) \rightarrow (pointer\,(\beta_1) \rightarrow \beta_3)$
c) $\gamma_1 \rightarrow \gamma_2$
d) $\delta_1 \rightarrow (\delta_1 \rightarrow \delta_2)$

6.23 Ampliense las reglas para la comprobación de tipos del ejemplo 6.6 para que abarque registros. Utilícese la siguiente sintaxis adicional para las siguientes expresiones de tipo y expresiones:

$$\begin{aligned} T &\rightarrow \textbf{record}\ campos\ \textbf{end} \\ E &\rightarrow E\ .\ \textbf{id} \\ campos &\rightarrow campos\ ;\ campos \mid campo \\ campo &\rightarrow \textbf{id}\ :T \end{aligned}$$

¿Qué limitaciones impone la falta de nombres de tipos a los tipos que se pueden definir?

***6.24** La resolución de la sobrecarga en la sección 6.5 se desarrolla en dos fases: primero se determina el conjunto de tipos posibles para cada subexpresión y después se reduce en una segunda fase a un solo tipo después de que se haya determinado el tipo único de la expresión completa. ¿Qué estructuras de datos se utilizarían para resolver la sobrecarga en una sola pasada ascendente?

****6.25** La resolución de la sobrecarga resulta más difícil si las declaraciones de identificadores son opcionales. En concreto, supóngase que se pueden utilizar las declaraciones para sobrecargar identificadores que representen símbolos de funciones, pero que todos los casos de un identificador no declarado tienen el mismo tipo. Demuéstrese que el problema de determinar si una expresión en este lenguaje tiene un tipo válido es NP completo. Este problema surge durante la comprobación de tipos en el lenguaje experimental Hope (Burstall, MacQueen y Sannella [1980]).

6.26 Siguiendo el ejemplo 6.12, infiéranse los siguientes tipos polimórficos para `trans`:

`trans` : $\forall\alpha.\ \forall\beta.\ ((\alpha\rightarrow\beta)\times \mathit{list}(\alpha)) \rightarrow \mathit{list}\ (\beta)$

La definición en ML de `trans` es:

```
fun trans(f,l) =
                  if null(l) then nil
                  else cons( f(hd(l)), trans(f,tl(l)))
```

Los tipos de los identificadores predefinidos en el cuerpo de la función son:

`null:` $\forall\alpha.\ \mathit{list}\ (\alpha) \rightarrow \mathit{boolean}$;
`nil:` $\forall\alpha.\ \mathit{list}\ (\alpha)$;
`cons:` $\forall\alpha.\ (\alpha \times \mathit{list}\ (\alpha)) \rightarrow \mathit{list}\ (\alpha)$;
`hd:` $\forall\alpha.\ \mathit{list}\ (\alpha) \rightarrow \alpha$;
`tl:` $\forall\alpha.\ \mathit{list}\ (\alpha) \rightarrow \mathit{list}\ (\alpha)$;

****6.27** Demuéstrese que el algoritmo de unificación de la sección 6.7 determina el unificador más general.

***6.28** Modifíquese el algoritmo de unificación de la sección 6.7 para que no unifique una variable con una expresión que contenga dicha variable.

****6.29** Supóngase que las expresiones se representan por medio de árboles. Encuéntrense expresiones e y f tales que para cualquier unificador S, el número de nodos en $S(e)$ sea exponencial en función del número de nodos en e y f.

6.30 Se dice que dos nodos son *congruentes* si representan expresiones equivalentes. Aunque no haya dos nodos congruentes en el grafo de tipo original, después de la unificación es posible que nodos distintos sean congruentes.

a) Dése un algoritmo para fusionar una clase de nodos mutuamente congruentes en un solo nodo.

**b) Amplíese el algoritmo de a) para que fusione nodos congruentes hasta que no haya dos nodos distintos congruentes.

***6.31** La expresión `g(g)` en la línea 9 del programa completo de C de la figura 6.28 es la aplicación de una función a sí misma. La declaración en la línea 3 da *integer* como el tipo del rango `g`, pero no se especifican los tipos de los argumentos de `g`. Pruébese a ejecutar el programa. El compilador puede emitir una advertencia porque `g` es considerado una función en la línea 3, en lugar de un apuntador a una función.

a) ¿Qué se puede decir sobre el tipo de `g`?
b) Utilícense las reglas de comprobación de tipos para funciones polimórficas de la figura 6.18 para inferir un tipo para `g` en el siguiente programa.

`m` : *integer*;
`veces` : $\mathit{integer} \times \mathit{integer} \rightarrow \mathit{integer}$;
`g` : α;
`veces( m, g(g) )`

```
(1)  int n;
(2)  int f(g);
(3)  int g();
(4)  {
(5)      int m;
(6)      m = n;
(7)      if( m == 0 ) return 1;
(8)      else {
(9)          n = n - 1; return m * g(g);
(10)     }
(11) }
(12) main()
(13) {
(14)     n = 5; printf("%d factorial es %d\n", n, f(f) );
(15) }
```

Fig. 6.28. Un programa en C que contiene aplicaciones de una función a sí misma.

NOTAS BIBLIOGRAFICAS

Los tipos básicos y los constructores de tipos de los primeros lenguajes como FORTRAN y ALGOL 60 eran tan limitados que la comprobación de tipos no suponía un problema serio. Por tanto, las descripciones de la comprobación de tipos en sus compiladores se ocultan tras los estudios de generación de código para expresiones. Sheridan [1959] describe la traducción de expresiones en el compilador original de FORTRAN. El compilador registraba si el tipo de una expresión era entero o real, pero el lenguaje no permitía coerciones. Backus [1981, pág. 54] recuerda: "Pienso que sólo porque no nos gustaban las reglas relativas a lo que pasaría con expresiones en modo mezclado, decidimos: 'Olvidémoslo. Es más fácil'". Naur [1965] es uno de los primeros artículos sobre comprobación de tipos en un compilador de ALGOL; las técnicas utilizadas por el compilador son similares a las estudiadas en la sección 6.2.

Los medios para la estructuración de datos, como matrices y registros, ya aparecían en los años 40 en el Plankalkül de Zuse, que tuvo poca influencia directa (Bauer y Wössner [1972]). Uno de los primeros lenguajes de programación que admitió la construcción sistemática de expresiones de tipos fue ALGOL 68. Las expresiones de tipos se pueden definir recursivamente, y se utiliza la equivalencia estructural. En EL1 existe una distancia clara entre equivalencia de nombre y equivalencia estructural, dejando que el programador elija (Wegbreit [1974]). La crítica de Pascal realizada por Welsh, Sneeringer y Hoare [1977] concentraba su interés sobre dicha distinción.

La combinación de coerción y sobrecarga puede causar ambigüedades: coercionar un argumento puede suponer que la sobrecarga se resuelva a favor de un algo-

ritmo distinto. Por tanto, se imponen limitaciones sobre una o la otra. En PL/I se adoptó un enfoque poco riguroso en lo relativo a las coerciones, siendo su primer criterio de diseño "*Todo vale*. Si una determinada combinación de símbolos tiene un significado bastante razonable, este significado se hará oficial (Radin y Rogoway [1965])". A menudo se impone un orden en el conjunto de tipos básicos —por ejemplo, Hext [1967] describe una estructura reticular impuesta sobre los tipos básicos de CPL— y los tipos inferiores se pueden coercionar a favor de tipos superiores.

La resolución de la sobrecarga en el momento de la compilación en lenguajes como APL (Iverson [1962]) y SETL (Schwartz [1973]) tiene la capacidad de mejorar el tiempo de ejecución de los programas (Bauer y Saal [1974]). Tennenbaum [1974] distingue entre resolución "hacia adelante" que determina el conjunto de tipos posibles de un operador a partir de sus operandos, y resolución "hacia atrás" basada en el tipo previsto según el contexto. Con una retícula de tipos, Jones y Muchnick [1976] y Kaplan y Ullman [1980] resuelven limitaciones sobre tipos obtenidos del análisis hacia adelante y hacia atrás. Se puede resolver la sobrecarga en Ada realizando una sola pasada hacia adelante y después una pasada hacia atrás, como en la sección 6.5. Esta observación aparece en varios artículos: Ganzinger y Ripken [1980]; Pennello, DeRemer y Meyers [1980]; Janas [1980]; Persch y colaboradores [1980]. Cormack [1981] proporciona una implantación recursiva y Baker [1982] evita una pasada explícita hacia atrás, arrastrando un GDA de tipos posibles.

La inferencia de tipos fue estudiada por Curry (véase Curry y Feys [1958]), en relación con la lógica combinatoria y el cálculo lambda de Church [1941]. Se ha señalado repetidas veces que el cálculo lambda es el centro de un lenguaje funcional. En este capítulo se ha utilizado repetidamente la aplicación de una función a un argumento para estudiar los conceptos de la comprobación de tipos. Las funciones se pueden definir y aplicar independientemente de los tipos en el cálculo lambda, y Curry se interesó por su "carácter funcional", y por determinar lo que ahora se denomina tipo polimórfico más general, compuesto por una expresión de tipo con cuantificadores universales, como en la sección 6.6. Motivado por Curry, Hindley [1969] observó que se podría utilizar la unificación para inferir los tipos. En su tesis, Morris [1968a] asignó tipos a expresiones lambda estableciendo un conjunto de ecuaciones y resolviéndolas para determinar los tipos asociados con las variables. Milner [1978], que no conocía el trabajo de Hindley, también observó que se podría utilizar la unificación para resolver los conjuntos de ecuaciones, y aplicó la idea para inferir tipos en el lenguaje de programación ML.

Cardelli [1984] describe la pragmática de la comprobación de tipos en ML. Meertens [1983] aplica este enfoque a lenguajes; Suzuki [1981] investiga su aplicación a Smalltalk 1976 (Ingalls [1978]). Mitchell [1984] describe cómo se pueden incluir coerciones.

Morris [1968a] observa que los tipos recursivos o circulares permiten inferir los tipos para expresiones que contengan la aplicación de una función a sí misma. El programa en C de la figura 6.28, que contiene una aplicación de una función a sí misma, está motivado por un programa en ALGOL de Ledgard [1971]. El ejercicio 6.31 es obra de MacQueen, Plotkin y Sethi [1984], y proporciona un modelo semántico para tipos polimórficos recursivos. En McCracken [1979] y Cartwright

[1985] hay distintos enfoques. Reynolds [1985] estudia el sistema de tipos de ML, los principios generales teóricos para evitar anomalías de coerciones y sobrecarga, y funciones polimórficas de orden superior.

Robinson [1965] fue el primero que estudió la unificación. El algoritmo de unificación de la sección 6.7 puede adaptarse fácilmente de algoritmos para comprobar la equivalencia de (1) autómatas finitos y (2) listas enlazadas con ciclos (Knuth [1973a], Sec. 2.3.5, Ejercicio 11). El algoritmo casi lineal para comprobar la equivalencia de autómatas finitos obra de Hopcroft y Karp [1971] se puede considerar la implantación del esbozo de la página 594 de Knuth [1973a]. Utilizando las estructuras de datos inteligentemente, Paterson y Wegman [1978] y Martelli y Montanari [1982] introducen algoritmos lineales para el caso acíclico. En Downey, Sethi y Tarjan [1980] aparece un algoritmo para encontrar nodos congruentes (véase Ejercicio 6.30).

Despeyroux [1984] describe un generador de comprobadores de tipos que utiliza la concordancia de patrones para crear un comprobador de tipos a partir de una especificación semántica operativa basada en reglas de inferencia.

CAPITULO 7

Ambientes para el momento de la ejecución

Antes de considerar la generación de código, hay que relacionar el texto fuente estático de un programa con las acciones que deben ocurrir en el momento de la ejecución para implantar el programa. Durante la ejecución, el mismo nombre en el texto fuente puede denotar distintos objetos de datos en la máquina objeto. Este capítulo examina las relaciones entre los nombres y los objetos de datos.

La asignación y desasignación de objetos de datos es llevada a cabo mediante el paquete de *apoyo durante la ejecución*, que consta de rutinas cargadas con el código objeto generado. El diseño del paquete de apoyo a la de ejecución está influido por la semántica de los procedimientos. Con las técnicas de este capítulo se pueden construir paquetes de apoyo para lenguajes como FORTRAN, Pascal y LISP.

Cada ejecución de un procedimiento se denomina *activación* del procedimiento. Si el procedimiento es recursivo, pueden estar funcionando varias de sus actividades al mismo tiempo. En Pascal, cada llamada a un procedimiento desemboca en una activación que puede manipular objetos de datos asignados para su uso.

La representación de un objeto de datos durante la ejecución está determinada por su tipo. A menudo, los tipos de datos elementales, como caracteres, enteros y reales, pueden estar representados por objetos de datos equivalentes en la máquina objeto. Sin embargo, los agregados, como matrices, cadenas y estructuras, generalmente se representan mediante conjuntos de objetos primitivos; su esquema de distribución se estudia en el capítulo 8.

7.1 ASPECTOS DEL LENGUAJE FUENTE

Como ejemplo, supóngase que un programa está compuesto por procedimientos, como en Pascal. Esta sección distingue entre el texto fuente de un procedimiento y sus actividades durante la ejecución.

Procedimientos

Una *definición de un procedimiento* es una declaración que, en su forma más simple, asocia un identificador con una proposición. El identificador es el *nombre del*

procedimiento, y la proposición es el *cuerpo del procedimiento*. Por ejemplo, el código en Pascal de la figura 7.1 contiene la definición del procedimiento llamado `leematriz` en las líneas 3 a 7; el cuerpo del procedimiento está en las líneas 5 a 7. Los procedimientos que devuelven valores se denominan *funciones* en muchos lenguajes; sin embargo, es mejor llamarlos procedimientos. Un programa completo también se considerará un procedimiento.

```
(1)  program ordenamiento(input,output);
(2)      var a : array [0..10] of integer;

(3)      procedure leematriz;
(4)          var i : integer;
(5)          begin
(6)              for i := 1 to 9 do read(a[i])
(7)          end;

(8)      function partición(y, z: integer) : integer;
(9)          var i, j, x, v: integer;
(10)         begin . . .
(11)         end;

(12)     procedure clasificación_por_particiones(m, n: integer);
(13)         var i : integer;
(14)         begin
(15)             if ( n > m ) then begin
(16)                 i := partición(m,n);
(17)                 clasificación_por_particiones(m,i-1);
(18)                 clasificación_por_particiones(i+1,n)
(19)             end
(20)         end;

(21)     begin
(22)         a[0] := -9999; a[10] := 9999;
(23)         leematriz;
(24)         clasificación_por_particiones(1,9)
(25)     end.
```

Fig. 7.1. Un programa en Pascal para leer y ordenar enteros.

Cuando aparece el nombre de un procedimiento dentro de una proposición ejecutable, se dice que el procedimiento es *llamado* en dicho momento. La idea básica es que la llamada a un procedimiento ejecuta el cuerpo del procedimiento. El programa principal en las líneas 21 a 25 de la figura 7.1 llama al procedimiento `leematriz` en la línea 23 y después llama al procedimiento `clasificación_por_particiones` en la línea 24. Obsérvese que las llamadas a procedimientos también pueden ocurrir dentro de expresiones, como en la línea 16.

Algunos de los identificadores que aparecen en la definición de un procedimiento son especiales, y se denominan *parámetros formales* del procedimiento. (En

C se denominan "argumentos formales" y en FORTRAN "falsos argumentos".) Los identificadores `m` y `n` de la línea 12 son parámetros formales de `clasificación_por_particiones`. Los argumentos, conocidos como *parámetros actuales*, pueden pasarse a un procedimiento llamado; son sustituidos por los parámetros formales del cuerpo. En la sección 7.5 se estudian los métodos para establecer la correspondencia entre los parámetros reales y los parámetros formales. La línea 18 de la figura 7.1 es una llamada de `clasificación_por_particiones` con parámetros actuales `i+1` y `n`.

Arboles de activación

Se tienen en cuenta los siguientes supuestos sobre el flujo de control entre procedimientos durante la ejecución de un programa:

1. El control fluye secuencialmente; es decir, la ejecución de un programa consta en una secuencia de pasos, y el control está en algún punto específico del programa a cada paso.
2. Cada ejecución de un procedimiento comienza al principio del cuerpo del procedimiento y en algún momento devuelve el control al punto situado inmediatamente tras el lugar donde fue llamado el procedimiento. Esto significa que puede describirse el flujo del control entre procedimientos utilizando árboles, como se verá más adelante.

Cada ejecución del cuerpo de un procedimiento se considera una activación del procedimiento. La *duración* de una activación de un procedimiento `p` es la secuencia de pasos entre el primero y último paso de la ejecución del cuerpo del procedimiento, incluido el tiempo que se tarda en ejecutar los procedimientos llamados por `p`, los procedimientos llamados por ellos, y así sucesivamente. En general, el término "duración" se refiere a una secuencia consecutiva de pasos durante la ejecución de un programa.

En lenguajes como Pascal, cada vez que el control se introduce en un procedimiento `q` desde el procedimiento `p`, finalmente regresa a `p` (si no hay un error inevitable). Más concretamente, cada vez que el control fluye desde una activación de un procedimiento `p` a una activación de un procedimiento `q`, se regresa a la misma activación de `p`.

Si *a* y *b* son activaciones de procedimientos, entonces sus duraciones o bien no se solapan o son anidadas. Es decir, si se entra a *b* antes de salir de *a*, entonces el control debe abandonar *b* antes de abandonar *a*.

Esta propiedad de anidamiento de las duraciones de las actividades se puede ilustrar insertando dos proposiciones de impresión en cada procedimiento, una antes de la primera proposición del cuerpo del procedimiento y la otra después de la última. La primera proposición imprime `entra` seguida del nombre del procedimiento y los valores de los parámetros reales; la última proposición imprime `sale` seguida de la misma información. Una ejecución del programa de la figura 7.1 con estas proposiciones de impresión produjo el resultado que se muestra en la figura 7.2. La duración de la activación `clasificación_por_particiones(1,9)` es la secuencia de pasos ejecutados entre la impresión de `entra a clasifica-`

`ción_por_particiones(1,9)` y la impresión de `sale de clasificación_por_particiones(1,9)`. En la figura 7.2, se supone que el valor devuelto por `partición(1,9)` es 4.

Un procedimiento es *recursivo* si puede comenzar una nueva activación antes de que haya terminado una activación anterior del mismo procedimiento. En la figura 7.2 se muestra que el control entra en la activación de `clasificación_por_particiones(1,9)`, de la línea 24, al principio de la ejecución del programa pero abandona dicha activación casi al final. Mientras tanto, hay otras activaciones de `clasificación_por_particiones`, de modo que `clasificación_por_particiones` es recursivo.

Un procedimiento recursivo `p` no necesita llamarse a sí mismo directamente; `p` puede llamar a otro procedimiento `q`, que puede entonces llamar a `p` a través de una secuencia de llamadas a procedimientos. Se puede utilizar un árbol, llamado *árbol de activación*, para representar la forma en que el control entra y sale de las activaciones. En un árbol de activaciones,

1. cada nodo representa una activación de un procedimiento,

2. la raíz representa la activación del programa principal,

3. el nodo para *a* es el padre del nodo para *b* si, y sólo si, el control fluye de la activación *a* a la *b*, y

4. el nodo para *a* está a la izquierda del nodo para *b* si, y sólo si, la duración de *a* ocurre antes que la duración de *b*.

Como cada nodo representa una activación única, y viceversa, conviene decir que el control está en un nodo cuando está en la activación representada por el nodo.

```
La ejecución comienza...
entra a leematriz
sale de leematriz
entra a clasificación_por_particiones(1,9)
entra a partición(1,9)
sale de partición(1,9)
entra a clasificación_por_particiones(1,3)
    . . .
sale de clasificación_por_particiones(1,3)
entra a clasificación_por_particiones(5,9)
    . . .
sale de clasificación_por_particiones(5,9)
sale de clasificación_por_particiones(1,9)
Termina ejecución.
```

Fig. 7.2. Salida que sugiere las activaciones de los procedimientos de la figura 7.1.

Ejemplo 7.1. El árbol de activación de la figura 7.3 se construye a partir del resultado de la figura 7.2[1]. Para ahorrar espacio, sólo se muestra la primera letra de cada procedimiento. La raíz del árbol de activación corresponde a todo el programa `ordenamiento`. Durante la ejecución de `ordenamiento`, existe una activación de `leematriz` representada por el primer hijo de la raíz, con etiqueta `1`. La siguiente activación, representada por el segundo hijo de la raíz, es para `clasificación_por_particiones`, con parámetros reales 1 y 9. Durante esta activación, las llamadas a `partición` y `clasificación_por_particiones` de las líneas 16 a 18 de la figura 7.1 conducen a las activaciones `p(1,9)`, `c(1,3)` y `c(5,9)`. Obsérvese que las activaciones `c(1,3)` y `c(5,9)` son recursivas, y que comienzan y terminan antes de que termine `c(1,9)`. □

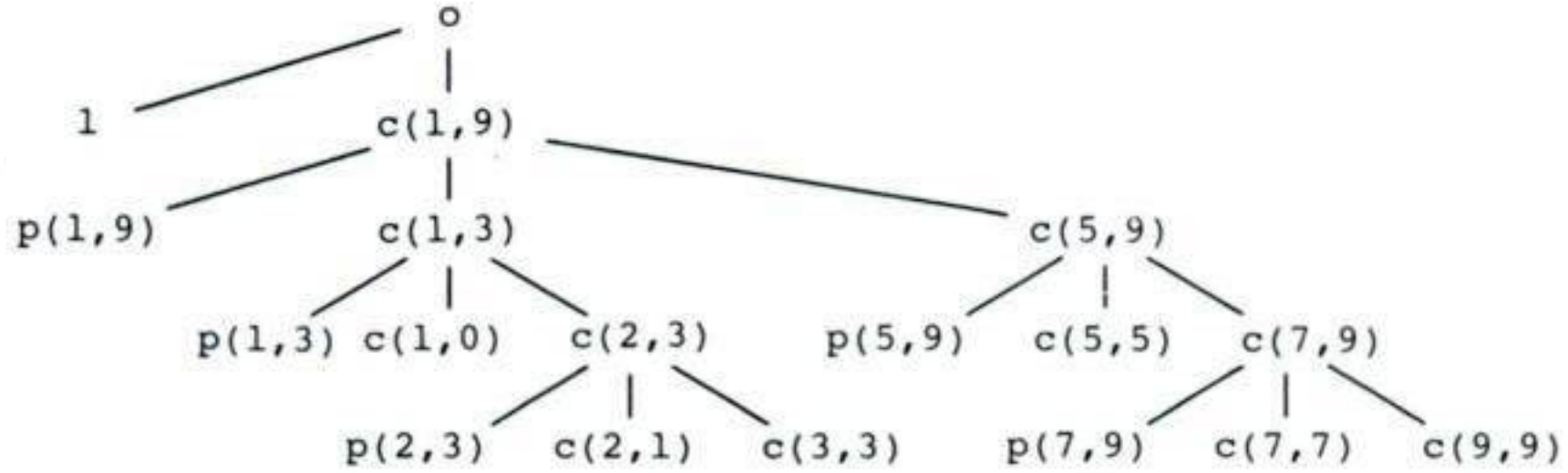


Fig. 7.3. Un árbol de activación correspondiente a la salida de la figura 7.2.

Pilas de control

El flujo de control de un programa corresponde a un recorrido en profundidad del árbol de activación que comienza en la raíz, visita un nodo antes que a sus hijos y visita los hijos en cada nodo recursivamente de izquierda a derecha. Por tanto, el resultado de la figura 7.2 se puede reconstruir recorriendo el árbol de activaciones de la figura 7.3, imprimiendo `entra` cuando se alcanza por primera vez el nodo de una activación e imprimiendo `sale` cuando todo el subárbol del nodo haya sido visitado durante el recorrido.

Se puede utilizar una pila, llamada *pila de control* para llevar un registro de las activaciones de los procedimientos en curso. Se trata de introducir el nodo para una activación en la pila de control cuando comience la activación, y sacarlo cuando termine. Los contenidos de la pila de control se relacionarán con los caminos hacia la raíz del árbol de activaciones. Cuando el nodo *n* esté en el tope de la pila de control, la pila contendrá los nodos situados a lo largo del camino de *n* hasta la raíz.

Ejemplo 7.2. En la figura 7.4 se muestran los nodos del árbol de activaciones de la figura 7.3 que han sido alcanzados cuando el control entra en la activación repre-

---

[1] Las llamadas actuales realizadas por `clasificación_por_participaciones` dependen de lo que devuelva `partición` (véase Aho, Hopcroft y Ullman [1983] para detalles del algoritmo). La figura 7.3 representa un posible árbol de llamadas. Resulta coherente con la figura 7.2 aunque en la figura 7.2 no se muestran algunas llamadas en niveles más bajos del árbol.

sentada por `c(2,3)`. Las activaciones con etiquetas `1`, `p(1,9)`, `p(1,3)` y `c(1,0)` han terminado su ejecución, de modo que la figura contiene líneas punteadas hacia sus nodos. Las otras líneas marcan el camino desde `c(2,3)` hasta la raíz.

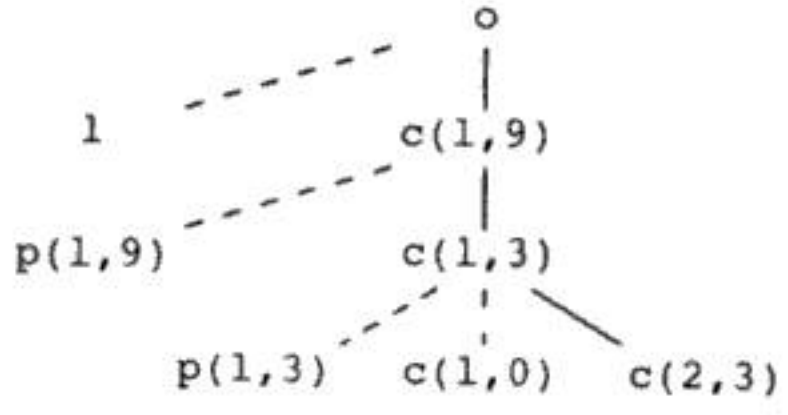


Fig. 7.4. La pila de control contiene los nodos a lo largo de un camino a la raíz.

Llegados a este punto, la pila de control contiene los siguientes nodos a lo largo de dicho camino hacia la raíz (el tope de la pila está a la derecha)

```
o, c(1,9), c(1,3), c(2,3)
```

y ningún otro nodo. □

Las pilas de control se aplican también a la técnica de asignación de memoria por medio de pilas utilizada para implantar lenguajes como Pascal y C. Esta técnica se estudia detalladamente en las secciones 7.3 y 7.4.

El ámbito de una declaración

Una declaración en un lenguaje es una construcción sintáctica que asocia información a un nombre. Las declaraciones pueden ser explícitas, tal como en el fragmento de Pascal

```
var i : integer;
```

o pueden ser implícitas. Por ejemplo, se supone que cualquier nombre de variable que comience con `I` indica un entero en un programa en FORTRAN, a menos que se declare lo contrario.

Puede haber declaraciones independientes del mismo nombre en distintas partes de un programa. Las *reglas de ámbito* de un lenguaje determinan qué declaración de un nombre se utiliza cuando el nombre aparece en el texto de un programa. En el programa en Pascal de la figura 7.1, `i` está declarada tres veces, en las líneas 4, 9 y 13, y los usos del nombre `i` en los procedimientos `leematriz`, `partición` y `clasificación_por_particiones` son independientes entre sí. La declaración de la línea 4 se aplica a los usos de `i` de la línea 6. Es decir, los dos casos de `i` de la línea 6 están dentro del ámbito de la declaración de la línea 4. Los tres casos de `i` de las líneas 16 a 18 están dentro del ámbito de la declaración de `i` de la línea 13.

La parte del programa a la que se aplica una declaración se denomina *ámbito* de dicha declaración. Se dice que el caso de un nombre en un procedimiento es *local* al procedimiento si está dentro del ámbito de una declaración dentro del procedimiento; si no, el caso es *no local*. La distinción entre nombres locales y no locales sirve para cualquier construcción sintáctica que pueda tener incluidas declaraciones.

Aunque el ámbito es una propiedad de la declaración de un nombre, a veces conviene utilizar la abreviatura "el ámbito del nombre `x`" en lugar de "el ámbito de la declaración del nombre `x` que se aplica a este caso de `x`". En este sentido, el ámbito de `i` en la línea 17 de la figura 7.1 es el cuerpo de `clasificación_por_particiones`[2].

En el momento de la compilación, se puede utilizar la tabla de símbolos para encontrar la declaración que se aplica a un caso de un nombre. Cuando aparece una declaración, se crea una entrada en la tabla de símbolos para ella. Mientras se esté situado dentro del ámbito de la declaración, se devuelve su entrada cuando se busca su nombre en la tabla. Las tablas de símbolos se estudian en la sección 7.6.

Enlace de nombres

Aunque cada nombre se declare una sola vez en el programa, el mismo nombre puede indicar distintos objetos de datos durante la ejecución. El término informal "objeto de datos" corresponde a una posición de memoria que puede contener valores.

En la semántica de los lenguajes de programación, el término *ambiente* se refiere a una función que transforma un nombre en una posición de memoria, y el término *estado* se refiere a una función que transforma una posición de memoria en el valor allí almacenado, como se muestra en la figura 7.5. Utilizando los términos *valor de lado izquierdo* y *valor de lado derecho* del capítulo 2, un ambiente transforma un nombre en un valor de lado izquierdo, y un estado transforma el valor de lado izquierdo en un valor de lado derecho.

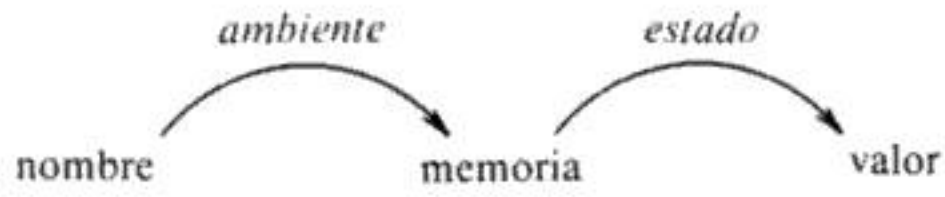


Fig. 7.5. Una transformación de dos etapas de nombres a valores.

Los ambientes y los estados son distintos; una asignación modifica el estado, pero no el ambiente. Por ejemplo, supóngase que la dirección 100 de la memoria, asociada con la variable `pi`, contiene `0`. Después de la asignación `pi := 3.14`, la misma dirección de la memoria sigue estando asociada con `pi`, pero el valor allí contenido es ahora `3.14`.

Cuando un ambiente asocia la posición de memoria *s* con un nombre `x`, se dice que `x` está *enlazado a s*; la asociación en sí misma es considerada un *enlace* de `x`. El término "posición" de memoria debe tomarse en sentido figurado. Si `x` no es un tipo básico, la posición *s* de memoria para `x` puede ser un conjunto de palabras de memoria.

Un enlace es la contrapartida dinámica de una declaración, como se muestra en la figura 7.6. Como ya se sabe, pueden estar funcionando al mismo tiempo más de

---

[2] La mayorías de las veces, los términos nombre, identificador, variable y lexema se pueden intercambiar sin que produzca ninguna confusión acerca de la construcción deseada.

una activación de un procedimiento recursivo. En Pascal, el nombre de una variable local de un procedimiento está enlazado a una posición de memoria distinta en cada activación de un procedimiento. En la sección 7.3 se consideran las técnicas para el enlazado de nombres de variables locales.

NOCIÓN ESTÁTICA	CONTRAPARTE DINÁMICA
definición de un procedimiento	activaciones del procedimiento
declaración de un nombre	enlaces del nombre
ámbito de una declaración	duración de un enlace

Fig. 7.6. Nociones estáticas y dinámicas correspondientes.

Algunas preguntas

La forma en que el compilador de un lenguaje debe organizar su memoria y enlazar nombres viene determinada por las respuestas a las siguientes preguntas:

1. ¿Pueden ser recursivos los procedimientos?
2. ¿Qué ocurre con los valores de los nombres locales cuando el control regresa de una activación de un procedimiento?
3. ¿Puede un procedimiento hacer referencia a nombres no locales?
4. ¿Cómo se pasan los parámetros cuando se llama a un procedimiento?
5. ¿Se pueden pasar procedimientos como parámetros?
6. ¿Se pueden devolver procedimientos como resultados?
7. ¿Se puede asignar dinámicamente la memoria bajo un control del programa?
8. ¿Se debe desasignar explícitamente la memoria?

En el resto de este capítulo se estudia el resultado de estas cuestiones en el sistema de apoyo de la ejecución, necesario para un determinado lenguaje de programación.

7.2 ORGANIZACIÓN DE LA MEMORIA

La organización de la memoria para la ejecución de esta sección se puede utilizar para lenguajes como FORTRAN, Pascal y C.

Subdivisión de la memoria durante la ejecución

Supóngase que el compilador obtiene un bloque de memoria del sistema operativo para que se ejecute el programa compilado. Según el análisis de la última sección, esta memoria para el momento de la ejecución debe estar subdividida de modo que pueda albergar:

1. el código objeto generado,
2. los objetos de datos y
3. una contrapartida de la pila de control para registrar las activaciones de procedimientos.

El código objeto generado tiene un determinado tamaño en el momento de la compilación, así que el compilador puede colocarlo estáticamente en una zona, tal vez en el extremo inferior de la memoria. De manera similar, también se puede conocer el tamaño de algunos de los datos en el momento de la compilación, y por tanto también se pueden colocar en una zona estáticamente, como se muestra en la figura 7.7. Una razón para asignar estáticamente tantos datos como sea posible es que las direcciones de dichos objetos se pueden compilar al código objeto. Todos los objetos de datos en FORTRAN pueden asignarse estáticamente.

Las implantaciones de lenguajes como Pascal y C utilizan ampliaciones de la pila de control para gestionar las activaciones de los procedimientos. Cuando ocurre una llamada, se interrumpe la ejecución de una activación y se guarda en la pila la información sobre el estado de la máquina, como el valor del contador del programa y los registros de dicha máquina. Cuando el control regresa de la llamada, se puede reiniciar la activación después de almacenar los valores de los registros relevantes y colocar el contador del programa en el punto situado inmediatamente después de la llamada. Los datos cuyas duraciones estén contenidas en la de una activación se pueden colocar en la pila, junto con otra información asociada con la activación. Esta estrategia se estudia en la siguiente sección.

Fig. 7.7. Subdivisión típica de la memoria durante la ejecución en áreas de código y de datos.

Un área distinta de la memoria para el momento de la ejecución, llamada *montículo*, guarda el resto de la información. Pascal permite que los datos se asignen durante el control del programa, como se estudia en la sección 7.7; la memoria para dichos datos se toma del montículo. Las implantaciones de lenguajes en los que las duraciones de las activaciones no se pueden representar con un árbol de activaciones pueden utilizar la pila para guardar la información sobre las activaciones. La forma controlada en que se asignan y desasignan los datos en una pila hace que sea menos costoso colocar los datos en la pila que en el montículo.

Los tamaños de la pila y del montículo pueden variar durante la ejecución del programa, así que se colocan en los extremos opuestos de la memoria en la figura 7.7, donde pueden crecer el uno hacia el otro convenientemente. Pascal y C nece-

sitan ambos una pila y un montículo durante la ejecución, pero no así todos los lenguajes.

Por norma, las pilas crecen hacia abajo. Es decir, el "tope" de la pila se dibuja hacia la parte inferior de la página. Como las direcciones de memoria aumentan conforme se recorre la página, el "crecimiento hacia abajo" significa hacia direcciones superiores. Si *tope* marca el tope de la pila, se pueden calcular los desplazamientos desde el tope de la pila restando el desplazamiento a *tope*. Con muchas máquinas este cálculo se puede realizar con eficacia manteniendo el valor de *tope* en un registro. Entonces, las direcciones de la pila se pueden representar como desplazamientos desde el *tope*[3].

Registros de activación

La información necesaria para una sola ejecución de un procedimiento se consigue utilizando un bloque contiguo de memoria llamado *registro de activación* o *marco*, que consta del conjunto de campos que se muestra en la figura 7.8. No todos los lenguajes ni todos los compiladores utilizan la totalidad de estos campos. A menudo los registros pueden sustituir uno o más campos. Para lenguajes como Pascal y C, normalmente se introduce el registro de activación de un procedimiento en la pila de ejecución cuando se llama al procedimiento y se extrae de la pila cuando el control regresa al autor de la llamada.

La finalidad de los campos de un registro de activación es la siguiente, comenzando por el campo para los valores temporales.

1. Los valores temporales, como los que surgen en la evaluación de expresiones, se almacenan en el campo para valores temporales.

2. El campo para datos locales guarda los datos locales a una ejecución de un procedimiento. La disposición interna de este campo se estudia más adelante.

3. El campo para el estado guardado de la máquina contiene información sobre el estado de la máquina justo antes de que sea llamado el procedimiento. Esta información incluye los valores del contador del programa y los registros de la máquina que deben reponerse cuando el control regrese del procedimiento.

4. En al sección 7.4 se utiliza el *enlace de acceso* opcional para hacer referencia a los datos no locales guardados en otros registros de activación. Para un lenguaje como FORTRAN, no se necesitan los enlaces de acceso porque los datos no locales se mantienen en un lugar fijo. Pascal necesita los enlaces de acceso, o el mecanismo de estructura de datos tipo *display*.

---

[3] La organización de la figura 7.7 supone que la memoria para la ejecución consta de un solo bloque contiguo de memoria obtenido al inicio de la ejecución. Esta suposición impone un límite fijo a la suma de los tamaños de la pila y del montículo. Si el límite es lo bastante amplio como para que raramente se sobrepase, puede que resulte demasiado amplio para la mayoría de los programas. La alternativa de enlazar objetos en la pila y el montículo puede encarecer el seguimiento del tope de la pila. Además, puede que la máquina objeto fomente una diferente colocación de áreas. Por ejemplo, algunas máquinas sólo permiten desplazamientos positivos desde una dirección de un registro.

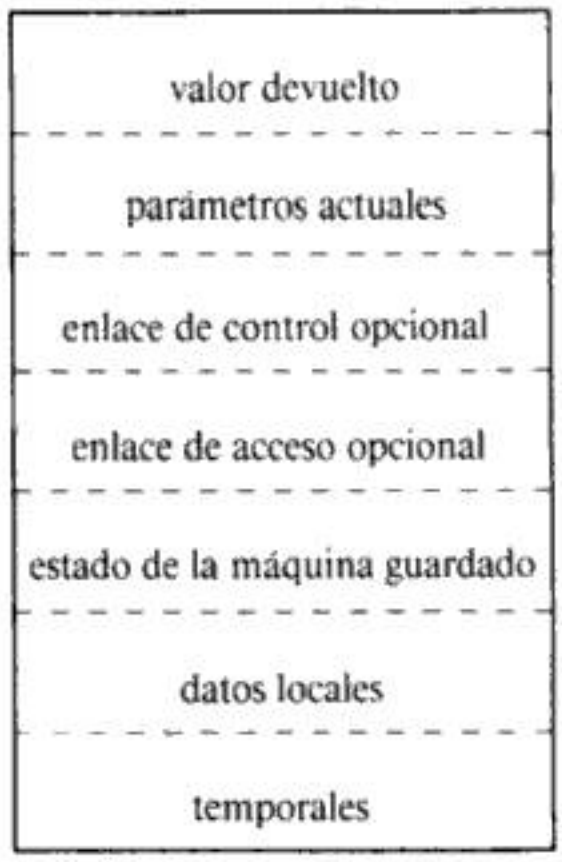


Fig. 7.8. Un registro de activación general.

5. El *enlace de control* opcional apunta al registro de activación del autor de la llamada.
6. El campo para los parámetros actuales es utilizado por el procedimiento autor de la llamada para proporcionar parámetros al procedimiento que recibe la llamada. Existe espacio para los parámetros en el registro de activación, pero en la práctica los parámetros se pasan en los registros de la máquina para una mayor eficiencia.
7. El campo para el valor devuelto es utilizado por el procedimiento que recibe la llamada para devolver un valor al procedimiento autor de la llamada. De nuevo, en la práctica este valor se suele trasladar en un registro para mayor eficiencia.

Los tamaños de cada uno de estos campos se pueden determinar en el momento en que es llamado un procedimiento. De hecho, los tamaños de casi todos los campos se pueden determinar en el momento de la compilación. Existe una excepción cuando un procedimiento tiene una matriz local cuyo tamaño venga determinado por el valor de un parámetro actual, disponible únicamente cuando el procedimiento sea llamado durante la ejecución. Véase la sección 7.3 para la asignación de datos de longitud variable en un registro de activación.

Disposición espacial de los datos locales en el momento de la compilación

Supóngase que la memoria para la ejecución se obtiene en bloques de bytes contiguos, donde un byte es la mínima unidad de memoria direccionable. En muchas máquinas, un byte consta de ocho bits y cierto número de bytes forman una palabra de máquina. Los objetos multibyte se almacenan en bytes consecutivos y se les da la dirección del primer byte.

La cantidad de memoria necesaria para un nombre viene determinada por su tipo. Un tipo de datos elemental, como un carácter, un entero o un real, generalmente se puede almacenar en un número entero de bytes. La memoria para un agregado, como una matriz o un registro, debe ser lo suficientemente grande como para dar cabida a todos sus componentes. Para acceder fácilmente a los componentes, la

memoria para los agregados se coloca generalmente en un bloque contiguo de bytes. Véanse las secciones 8.2 y 8.3 si se quieren detalles.

El campo para los datos locales se perfila conforme se examinan las declaraciones en un procedimiento durante la compilación. Los datos de longitud variable se mantienen fuera de este campo. Se hace un recuento de las posiciones de memoria asignadas a declaraciones anteriores. Según el resultado se determina una dirección *relativa* de la memoria para un valor local con respecto a alguna posición, como el comienzo del registro de activación. La dirección relativa, o *desplazamiento*, es la diferencia entre las direcciones de esa posición y del objeto de datos.

La disposición espacial de la memoria para los datos está muy influida por las limitaciones de direccionamiento de la máquina objeto. Por ejemplo, las instrucciones para sumar enteros pueden suponer que los enteros están *alineados*, es decir, colocados en posiciones de memoria como una dirección divisible por 4. Aunque una matriz de diez caracteres sólo necesita los bytes suficientes para guardar diez caracteres, un compilador puede por tanto colocar 12 bytes, dejando sin utilizar 2 bytes. El espacio que no se usa por consideraciones de alineamiento se llama *relleno*. Cuando este espacio es muy pequeño, un compilador puede *empaquetar* datos de modo que no queda ningún relleno; entonces puede ser necesario practicar instrucciones adicionales durante la ejecución para situar los datos empaquetados de modo que se puedan manipular como si estuvieran alineados de forma apropiada.

Ejemplo 7.3. La figura 7.9 es una simplificación de la disposición de los datos utilizada por los compiladores para C para dos máquinas a las que se llamarán Máquina 1 y Máquina 2. C proporciona tres tamaños de números enteros, que se declaran utilizando las palabras clave `short`, `int` y `long`. Los conjuntos de instrucciones de las dos máquinas indican que el compilador para la Máquina 1 asigna 16, 32 y 32 bits para los tres tamaños de enteros, mientras que el compilador para la Máquina 2 asigna 24, 48 y 64 bits, respectivamente. Para comparar las máquinas, en la figura 7.9 los tamaños se miden en bits, aunque ninguna de las dos máquinas permita direccionar bits directamente.

La memoria de la Máquina 1 está organizada en bytes que constan de 8 bits cada uno. Aunque cada byte tiene una dirección, el conjunto de instrucciones propicia que los enteros de tipo `short` se posicionen hacia bytes cuyas direcciones sean pares, y que los de tipo `int` se posicionen en direcciones divisibles por 4. El compilador coloca los enteros de tipo `short` en direcciones pares, aunque tenga que saltarse un byte de relleno en el proceso. Por tanto, se pueden asignar cuatro bytes, formados por 32 bits para un carácter seguido de un entero tipo `short`.

En la Máquina 2, cada palabra consta de 64 bits, y se permiten 24 bits para la dirección de una palabra. Existen 64 posibilidades para los bits individuales dentro de una palabra, de modo que se necesitan 6 bits adicionales para distinguirlos. Debido al diseño, un apuntador a un carácter en la Máquina 2 utiliza 30 bits (24 para encontrar la palabra y 6 para la posición del carácter dentro de la palabra).

La orientación hacia palabras del conjunto de instrucciones de la Máquina 2 ha obligado al compilador a asignar una sola palabra completa a la vez, aunque bastaría con menos bits para representar todos los posibles valores de ese tipo; por ejemplo, sólo se necesitan 8 bits para representar un carácter. Por tanto, durante la ali-

neación, en la figura 7.9 se muestran 64 bits para cada tipo. Dentro de cada palabra, los bits para cada tipo básico están en posiciones conocidas. Para un carácter seguido de un entero de tipo `short`, se asignarían dos palabras compuestas por 128 bits, y el carácter utilizaría sólo 8 de los bits de la primera palabra y el entero tipo `short` sólo 24 de los bits de la segunda palabra. □

Tipo	Tamaño (Bits)		Alineación (Bits)	
	Máquina 1	Máquina 2	Máquina 1	Máquina 2
`char`	8	8	8	64[a]
`short`	16	24	16	64
`int`	32	48	32	64
`long`	32	64	32	64
`float`	32	64	32	64
`double`	64	128	32	64
apuntador a carácter .	32	30	32	64
otros apuntadores ...	32	24	32	64
estructuras	≥8	≥64	32	64

[a] Los caracteres en una matriz se alinean cada 8 bits.

Fig. 7.9. Distribuciones espaciales de datos utilizados por dos compiladores de C.

7.3 ESTRATEGIAS PARA LA ASIGNACION DE MEMORIA

En cada una de las tres áreas de datos de la figura 7.7 se utiliza una estrategia distinta para la asignación de memoria.

1. La asignación estática dispone la memoria para todos los objetos de datos durante la compilación.
2. La asignación por medio de una pila trata la memoria en ejecución como una pila.
3. La asignación por medio de un montículo asigna y desasigna la memoria conforme se necesita durante la ejecución a partir de un área de datos conocida como montículo.

Estas estrategias de asignación se aplican en esta sección a los registros de activación. También se describe cómo accede el código objeto de un procedimiento a la memoria enlazada a un nombre local.

Asignación estática

En la asignación estática, los nombres se ligan a la memoria durante la compilación del programa, así que no es necesario un paquete de apoyo para la ejecución. Como los enlaces no cambian durante la ejecución, cada vez que se activa un procedi-

miento, sus nombres se enlazan a las mismas posiciones de memoria. Esta propiedad permite que los valores de los nombres locales sean *retenidos* durante las activaciones de un procedimiento. Es decir, cuando el control regresa a un procedimiento, los valores de las variables locales son los mismos que cuando el control salió por última vez.

Según el tipo de un nombre, el compilador determina la cantidad de memoria que debe reservarse para dicho nombre, como se vio en la sección 7.2. La dirección de esta memoria consta de un desplazamiento desde un extremo del registro de activación del procedimiento. El compilador debe decidir a dónde van los registros de activación, con respecto al código objeto y a otro registro de activación. Una vez tomada esta decisión, queda determinada la posición de cada registro de activación, y por tanto de la memoria para cada nombre dentro del registro. Durante la compilación se pueden ahora proporcionar las direcciones en las que el código objeto puede encontrar los datos con los que opera. También se conocen durante la compilación las direcciones en las que se guarda la información cuando se produce la llamada a un procedimiento.

Sin embargo, utilizar únicamente la asignación estática conlleva algunas limitaciones.

1. El tamaño de un objeto de datos y las limitaciones en cuanto a su posición en la memoria deben conocerse en el momento de la compilación.
2. Los procedimientos recursivos son escasos, porque todas las activaciones de un procedimiento utilizan los mismos enlaces para los nombres locales.
3. Las estructuras de datos no se pueden crear dinámicamente ya que no hay un mecanismo para la asignación de memoria durante la ejecución.

FORTRAN fue diseñado para que admitiera la asignación estática de memoria. Un programa en FORTRAN consta de un programa principal, subrutinas y funciones (pueden llamarse todos *procedimientos*), como en el programa en FORTRAN 77 de la figura 7.10. Utilizando la organización de la memoria de la figura 7.7, la disposición del código y los registros de activación para dicho programa se muestran en la figura 7.11. Dentro del registro de activación para `CNSUME` (léase "consume" —a FORTRAN no le gustan los identificadores largos—), hay espacio para las variables locales `BUF`, `SIGTE` y `C`. La memoria enlazada con `BUF` guarda una cadena de cincuenta caracteres. Va seguida de espacio para guardar un valor entero para `SIGTE` y un valor de tipo `CHARACTER` para `C`. El hecho de que `SIGTE` también esté declarado dentro de `PRDUCE` no supone ningún problema, porque las variables locales de los dos procedimientos tienen espacio en sus respectivos registros de activación.

Como se conocen los tamaños del código ejecutable y de los registros de activación en el momento de la compilación, pueden darse organizaciones de memoria diferentes a la de la figura 7.11. Un compilador de FORTRAN puede colocar el registro de activación de un procedimiento junto con el código para dicho procedimiento. En algunos sistemas de computación, se podría no especificar la posición relativa de los registros de activación y permitir al director de enlace que enlace los registros de activación y el código ejecutable.

Ejemplo 7.4. El programa de la figura 7.10 se basa en que los valores de las variables locales quedan retenidos durante las activaciones de los procedimientos. Una proposición SAVE en FORTRAN 77 especifica que el valor de una variable local al principio de una activación debe ser el mismo que al final de la última activación. Se pueden especificar los valores iniciales para dichas variables utilizando la proposición DATA.

```
(1)        PROGRAM CNSUME
(2)             CHARACTER * 50 BUF
(3)             INTEGER SIGTE
(4)             CHARACTER C, PRDUCE
(5)             DATA SIGTE /1/, BUF / ' '/
(6)  6          C = PRDUCE()
(7)             BUF(SIGTE:SIGTE) = C
(8)             SIGTE = SIGTE + 1
(9)             IF (C .NE. ' ' ) GOTO 6
(10)            WRITE (*,'(A)') BUF
(11)            END

(12)       CHARACTER FUNCTION PRDUCE()
(13)            CHARACTER * 80 BUFFER
(14)            INTEGER SIGTE
(15)            SAVE BUFFER, SIGTE
(16)            DATA SIGTE /81/
(17)                IF ( SIGTE .GT. 80 ) THEN
(18)                    READ (*,'(A)') BUFFER
(19)                    SIGTE = 1
(20)                END IF
(21)                PRDUCE = BUFFER(SIGTE:SIGTE)
(22)                SIGTE = SIGTE + 1
(23)            END
```

Fig. 7.10. Un programa en FORTRAN 77.

La proposición de la línea 18 del procedimiento PRDUCE lee sólo una línea de texto a la vez dentro de un *buffer*. El procedimiento produce caracteres sucesivos para cada vez que se activa. El programa principal CNSUME también tiene un *buffer* en el que acumula caracteres hasta que aparezca un espacio en blanco. Con la entrada

```
hola mundo
```

los caracteres devueltos por las activaciones de PRDUCE se muestran en la figura 7.12; el resultado del programa es

```
hola
```

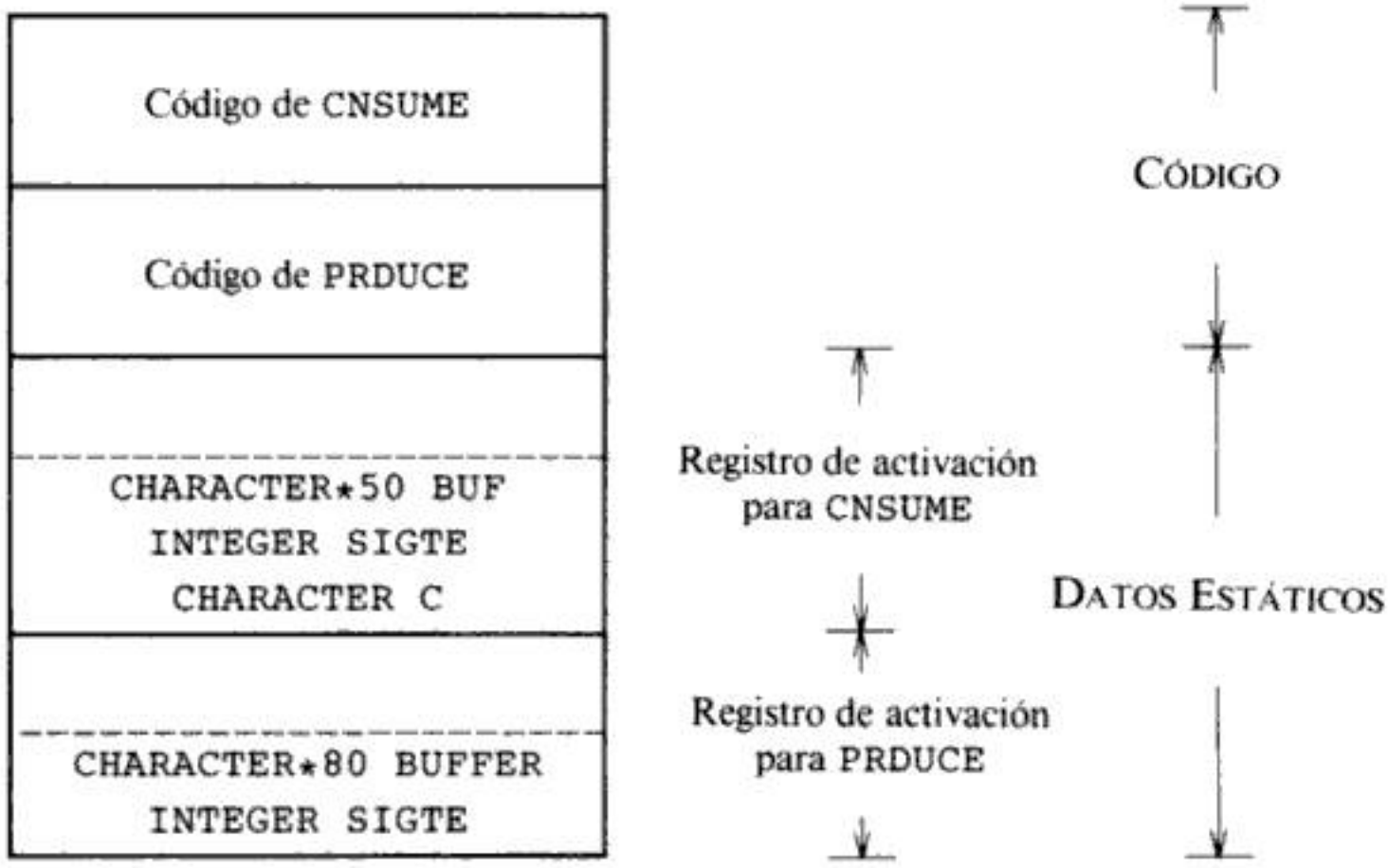


Fig. 7.11. Almacenamiento estático para identificadores locales de un programa en FORTRAN 77.

El *buffer* en el que PRDUCE lee las líneas debe conservar su valor entre varias activaciones. La proposición SAVE de la línea 15 garantiza que cuando el control regrese a PRDUCE, las variables locales BUFFER y SIGTE tengan los mismos valores que tenían cuando el control salió del procedimiento por última vez. La primera vez que el control llega a PRDUCE, se toma el valor de la variable local SIGTE de la proposición DATA de la línea 16. De esta forma, a SIGTE se le asigna el valor inicial 81. □

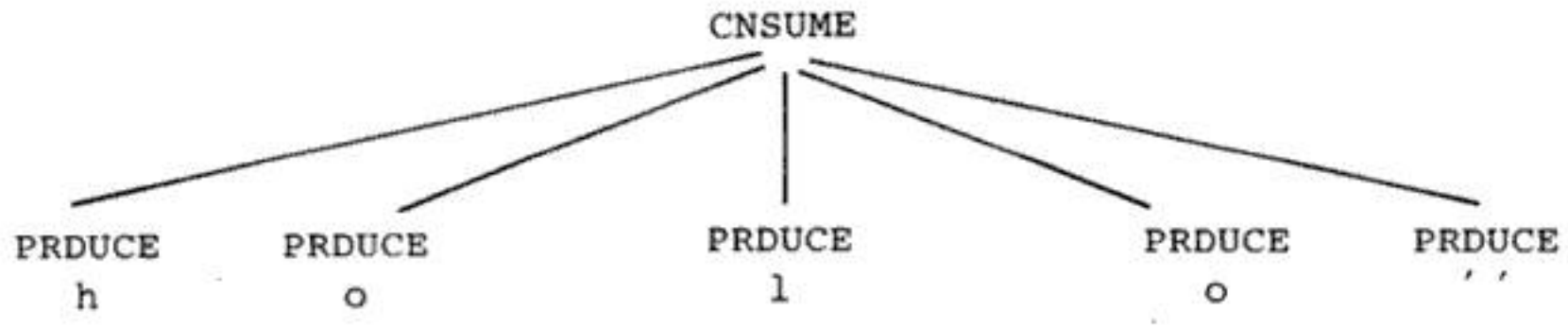


Fig. 7.12. Caracteres devueltos por las activaciones de PRODUCE.

Asignación por medio de una pila

La asignación por medio de una pila se basa en la idea de una pila de control; la memoria se organiza como una pila, y los registros de activación se introducen y se sacan cuando las activaciones comienzan y terminan, respectivamente. La memoria para las variables locales en cada llamada de un procedimiento está contenida en el registro de activación de dicha llamada. De ese modo, en cada activación, las variables locales se enlazan a una memoria nueva, puesto que se introduce un nuevo registro de activación en la pila al realizar una llamada. Además, los valores de las variables locales se *borran* cuando finaliza la activación; es decir, los valores se pier-

den porque la memoria para las variables locales desaparece cuando se extrae el registro de activación.

Primero se describe una forma de asignación por medio de una pila en la que se conocen los tamaños de todos los registros de activación en el momento de la compilación. Más adelante se consideran los casos en que sólo se dispone en el momento de la compilación de información incompleta acerca de los tamaños.

Supóngase que el registro *tope* marca el tope de la pila. Durante la ejecución, un registro de activación se puede asignar y desasignar incrementando y decrementando *tope*, respectivamente, por el tamaño del registro. Si el procedimiento `c` tiene un registro de activación de tamaño *a*, entonces *tope* se incrementa por *a* justo antes que se ejecute el código objeto de `c`. Cuando el control regresa de `c`, *tope* se decrementa por *a*.

Ejemplo 7.5. En la figura 7.13 se muestran los registros de activación que se introducen y se sacan en la pila para la ejecución cuando el control fluye a través del árbol de activaciones de la figura 7.3. Las líneas punteadas en el árbol van a activaciones que han terminado. La ejecución comienza con una activación del procedimiento `o`. Cuando el control alcanza la primera llamada en el cuerpo de `o`, el procedimiento `1` se activa y su registro de activación se mete en la pila. Cuando el control regresa de esta activación, el registro se saca dejando únicamente el registro `o` en la pila. En la activación de `o`, el control llega entonces a una llamada de `c` con parámetros actuales `1` y `9`, y se asigna un registro de activación en el tope de la pila para una activación de `c`. Siempre que el control esté en una activación, su registro de activación estará en el tope de la pila.

Se producen varias activaciónes entre los últimos dos estados de la figura 7.13. En el último estado de la figura 7.13, las activaciones `p(1,3)` y `c(1,0)` han comenzado y han terminado durante la duración de `c(1,3)`, de modo que sus registros de activación han entrado y salido de la pila, dejando el registro de activación para `c(1,3)` en el tope. □

En un procedimiento en Pascal se puede determinar una dirección relativa para los datos locales en un registro de activación, como se vio en la sección 7.2. Durante la ejecución, supóngase que *tope* marca la posición del final de un registro. La dirección de un nombre local `x` en el código objeto para el procedimiento puede escribirse por tanto como *dx* (*tope*), para indicar que los datos enlazados a `x` pueden encontrarse en la posición obtenida al sumar *dx* al valor del registro *tope*. Obsérvese que las direcciones pueden considerarse alternativamente como desplazamientos desde el valor de cualquier otro registro *r* que apunte a una posición fija en el registro de activación.

Secuencia de llamadas

Las llamadas a procedimientos se implantan mediante la generación de lo que se conoce como *secuencias de llamadas* en el código objeto. Una *secuencia de llamada* asigna un registro de activación e introduce información dentro de sus campos. Una *secuencia de retorno* restablece el estado de la máquina para que el procedimiento que efectúa la llamada pueda continuar su ejecución.

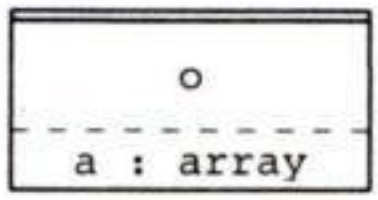
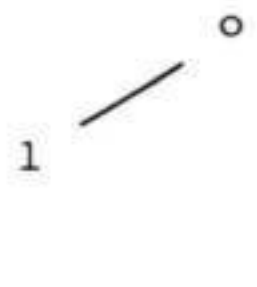
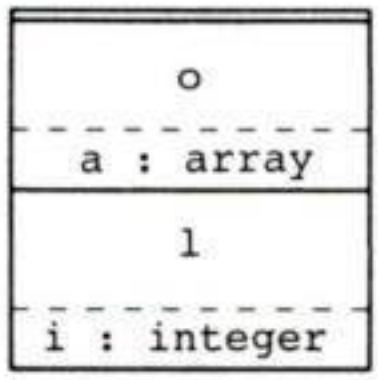
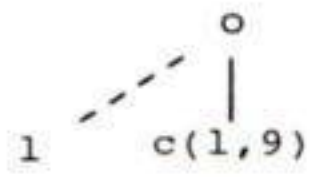
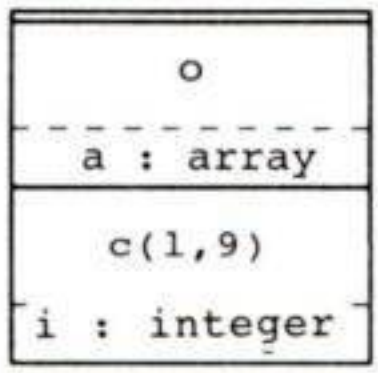
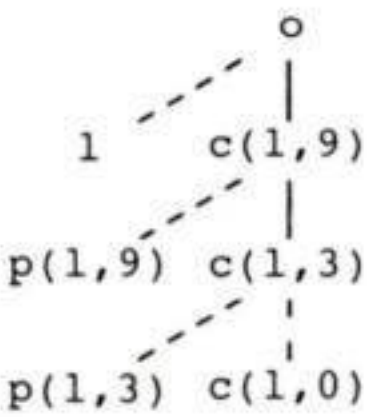
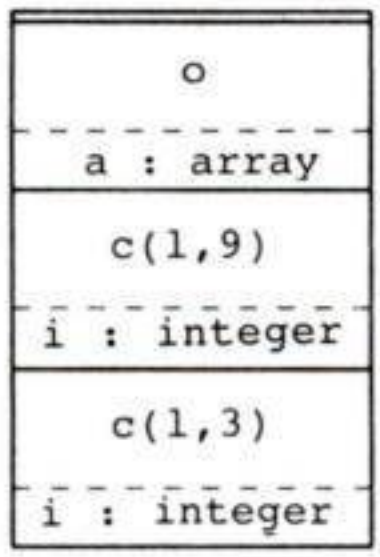
POSICIÓN EN EL ÁRBOL DE ACTIVACIÓN	REGISTROS DE ACTIVACIÓN EN LA PILA	COMENTARIOS
o		Marco para o
		Se activa 1
		Se ha sacado de la pila el marco para 1 y se ha metido c(1,9)
		El control acaba de regresar a c(1,3)

Fig. 7.13. Asignación, por medio de una pila con crecimiento hacia abajo, de registros de activación.

Las secuencias de llamadas y los registros de activación son diferentes, incluso para implantaciones del mismo lenguaje. El código en una secuencia de llamada a menudo está dividido en el procedimiento que hace la llamada y el procedimiento que recibe la llamada. No hay una división exacta entre el llamador y el llamado de

las tareas de ejecución —el lenguaje fuente, la máquina objeto y el sistema operativo imponen requisitos que pueden hacer prevalecer una solución sobre la otra[4]—.

Un principio que ayuda al diseño de secuencias de llamadas y registros de activación es que los campos cuyos tamaños se fijan primero se colocan en el medio. En el registro de activación general de la figura 7.8, el enlace de control, el enlace de acceso y los campos del estado de la máquina aparecen en el medio. La decisión de utilizar o no los enlaces de control y de acceso es parte del diseño del compilador, de modo que estos campos se pueden fijar durante la construcción del compilador. Si se guarda exactamente la misma cantidad de información sobre el estado de la máquina para cada activación, entonces el mismo código puede almacenar y restablecer para todas las activaciones. Además, a los programas como los depuradores les resultará más fácil interpretar el contenido de la pila cuando se produzca un error.

Aunque el tamaño del campo para los valores temporales se fija en un determinado momento de la compilación, este tamaño puede no conocerse en la etapa inicial. La generación u optimación cuidadosa del código puede reducir el número de variables temporales que necesite el procedimiento, de modo que, por lo que se refiere a la etapa inicial, no se conoce el tamaño de este campo. Por tanto, en el registro de activación general, se muestra este campo después de aquél para datos locales, donde los cambios en su tamaño no afectarán a los desplazamientos de los objetos de datos relativos a los campos del medio.

Como cada llamada tiene sus propios parámetros actuales, generalmente el que hace la llamada evalúa los parámetros actuales y los comunica al registro de activación del procedimiento receptor de la llamada. Los métodos para pasar parámetros se estudian en la sección 7.5. En la pila para la ejecución, el registro de activación del llamador se encuentra justo debajo del registro del procedimiento llamado, como se muestra en la figura 7.14. Es mejor que colocar los campos para los parámetros y para un valor devuelto potencial inmediatamente después del registro de activación del que hace la llamada. Así, el llamador puede acceder a estos campos utilizando desplazamientos desde el final de su propio registro de activación, sin necesidad de conocer la disposición completa del registro del procedimiento llamado. Además, el que realiza la llamada no tiene porqué conocer los datos locales o las variables temporales del procedimiento que recibe la llamada. La ventaja de ocultar esta información es que se pueden manejar procedimientos con un número variable de argumentos, como `printf` en C, que se estudiará más adelante.

Los lenguajes como Pascal exigen que las matrices locales a un procedimiento tengan una longitud que pueda determinarse durante la compilación. En la mayoría de los casos, el tamaño de una matriz local puede depender del valor de un parámetro pasado al procedimiento. Si así es, no se puede determinar el tamaño de todos los datos locales al procedimiento hasta que sea llamado el procedimiento. Más adelante en esta sección se estudiarán las técnicas para manejar datos de longitud variable.

---

[4] Si se llama *n* veces a un procedimiento, entonces la parte de la secuencia de llamada en los distintos llamadores se genera *n* veces. Sin embargo, la parte del procedimiento que recibe la llamada es compartida por todas las llamadas, de modo que se genera sólo una vez. Por tanto, es conveniente introducir cuanto sea posible de la secuencia de llamada en el procedimiento receptor de la llamada.

La siguiente secuencia de llamada está motivada por el análisis anterior. Como en la figura 7.14, el registro *tope_sp* apunta al final del campo del estado de la máquina en un registro de activación. El llamado conoce esta posición, así que se le considera responsable de asignar valor a *tope_sp* antes de que el control fluya al procedimiento receptor de la llamada. El código para el procedimiento llamado puede acceder a sus variables temporales y datos locales utilizando desplazamientos desde *tope_sp*. La secuencia de llamada es:

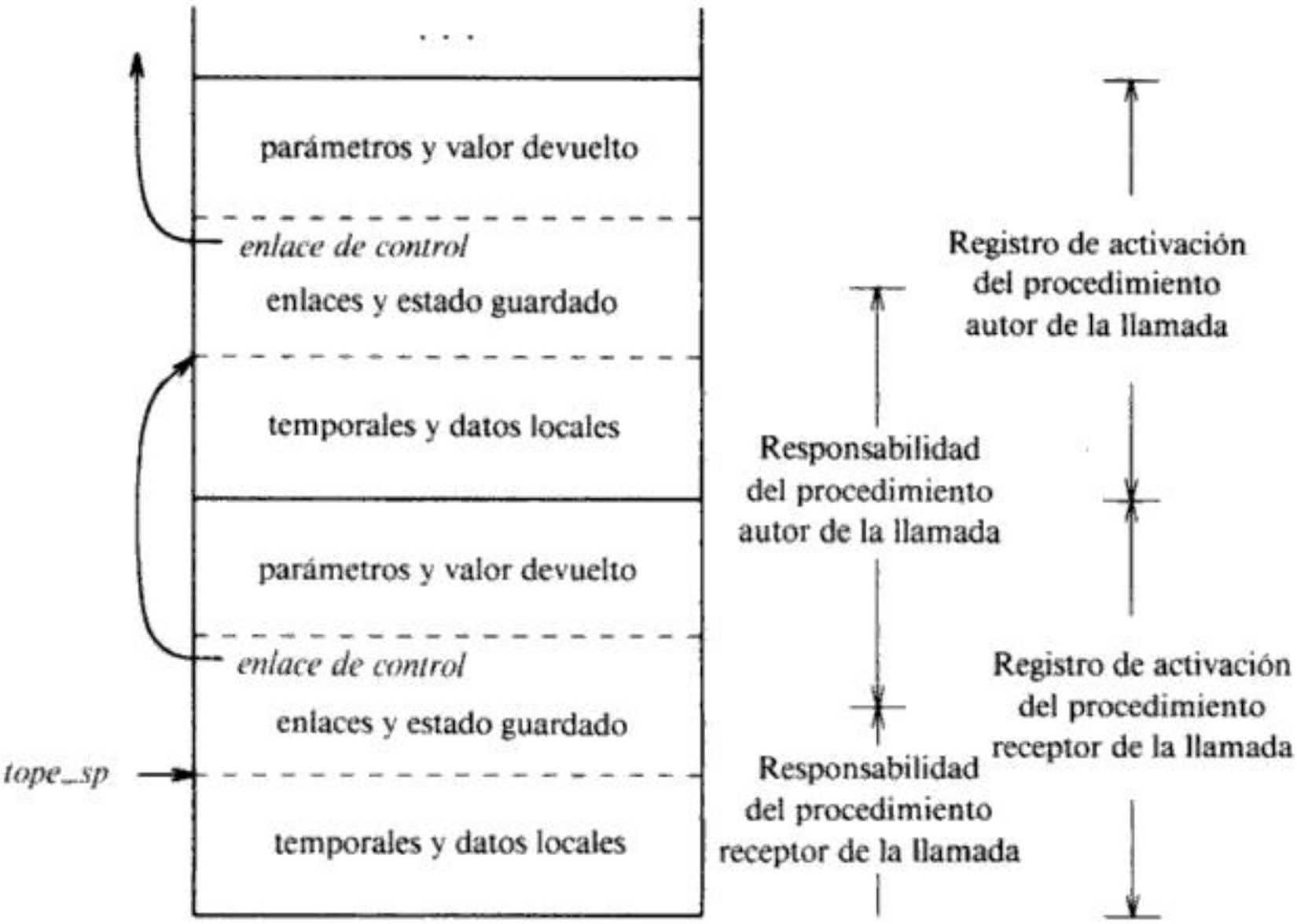


Fig. 7.14. División de tareas entre el procedimiento autor de la llamada y el procedimiento receptor de la llamada.

1. El que hace la llamada evalúa los parámetros reales.
2. El que hace la llamada almacena una dirección de regreso y el valor anterior de *tope_sp* en el registro de activación del procedimiento llamado. Después que efectúa la llamada incrementa *tope_sp* hasta la posición mostrada en la figura 7.14. Es decir, *tope_sp* se traslada más allá de los datos locales y las variables temporales del autor de la llamada y de los campos de parámetros y estado del procedimiento llamado.
3. El procedimiento receptor de la llamada guarda los valores de los registros y otra información sobre el estado.
4. El procedimiento receptor de la llamada asigna valores iniciales a sus datos locales y comienza la ejecución.

Una posible secuencia de retorno sería:

1. El procedimiento receptor de la llamada coloca un valor devuelto a continuación del registro de activación del autor de la llamada.
2. Utilizando la información del campo de estado, el procedimiento llamado restablece *tope_sp* y otros registros y se dirige a una dirección de retorno en el código del que efectúa la llamada.
3. Aunque se haya decrementado *tope_sp*, el autor de la llamada puede copiar el valor devuelto a su propio registro de activación y utilizarlo para evaluar una expresión.

Las secuencias de llamada anteriores permiten que el número de argumentos de procedimiento llamado dependa de la llamada. Obsérvese que durante la compilación, el código objeto del autor de la llamada conoce el número de argumentos que está proporcionando al procedimiento llamado. De este modo, el autor de la llamada conoce el tamaño del campo de parámetros. Sin embargo, el código objeto del procedimiento receptor de la llamada debe prepararse para manejar asimismo otras llamadas, así que espera hasta ser llamado, y después examina el campo de los parámetros. Utilizando la organización de la figura 7.14, la información que describen los parámetros debe colocarse después del campo de estado, de manera que la pueda encontrar el procedimiento llamado. Por ejemplo, considérese la función `printf` de la biblioteca estándar de C. El primer argumento de `printf` especifica la naturaleza de los argumentos restantes, de modo que una vez que `printf` localiza el primer argumento, puede encontrar los restantes.

Datos de longitud variable

En la figura 7.15 se sugiere una estrategia habitual para manejar datos de longitud variable, donde el procedimiento p tiene tres matrices locales. La memoria para estas matrices no es parte del registro de activación de p; en el registro de activación sólo aparece un apuntador al principio de cada matriz. Las direcciones relativas de estos apuntadores se conocen durante la compilación, de modo que el código objeto puede acceder a los elementos de las matrices a través de los apuntadores.

En la figura 7.15 también se muestra un procedimiento c llamado por p. El registro de activación de c comienza después de las matrices de p, y las matrices de longitud variable de c comienzan después del registro de c.

El acceso a los datos en la pila se hace a través de dos apuntadores, *tope* y *tope_sp*. El primero de ellos marca el tope actual de la pila; apunta a la posición en la que comenzará el siguiente registro de activación. El segundo se utiliza para encontrar los datos locales. Para ser coherente con la organización de la figura 7.14, supóngase que *tope_sp* apunta al final del campo del estado de la máquina. En la figura 7.15, *tope_sp* apunta al final de este campo en el registro de activación de c. Dentro del campo hay un enlace de control con el valor previo de *tope_sp* cuando el control estaba en la activación autora de la llamada de p.

El código para reposicionar a *tope* y *tope_sp* se puede generar durante la compilación, utilizando los tamaños de los campos en los registros de activación. Cuando retorna c, el nuevo valor de *tope* es *tope_sp* menos la longitud de los campos del estado de la máquina y de los parámetros en el registro de activación de c. Esta lon-

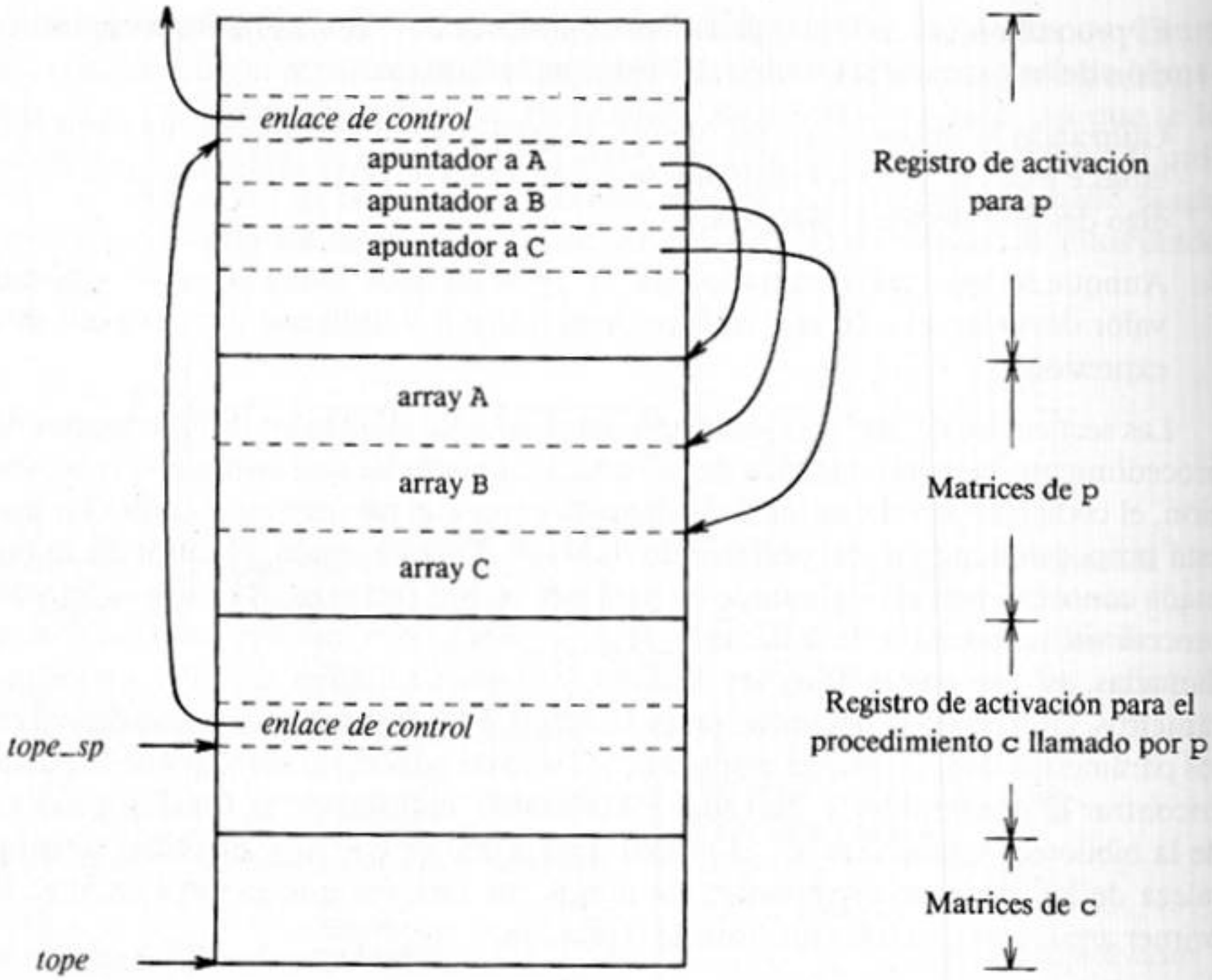


Fig. 7.15. Acceso a matrices asignadas dinámicamente.

gitud es conocida en el momento de la compilación, al menos por el que efectúa la llamada. Después de ajustar *tope*, el nuevo valor de *tope_sp* se puede copiar del enlace de control de c.

Referencias suspendidas

Siempre que se puede realizar una desasignación de memoria, surge el problema de las referencias suspendidas. Una *referencia suspendida* se produce cuando hay una referencia a una memoria que ha sido desasignada. Utilizar referencias suspendidas es un error lógico, puesto que el valor de la memoria desasignada no está definido según la semántica de la mayoría de los lenguajes. Peor aún, porque como esta memoria puede asignarse posteriormente a otro dato, pueden aparecer errores ocultos en programas con referencias suspendidas.

Ejemplo 7.6. El procedimiento `suspende` en el programa en C de la figura 7.16 devuelve un apuntador a la memoria enlazada al nombre local `i`. Se crea el apuntador con el operador `&` aplicado a `i`. Cuando el control regresa a `main` desde `suspende`, se libera la memoria para las variables locales y puede utilizarse para otros propósitos. Como `p` en `main` hace referencia a esta memoria, el uso de `p` constituye una referencia suspendida. □

En el ejemplo en la sección 7.7 se produce la desasignación durante el control del programa.

```
main()
{
    int *p;
    p = suspende();
}
int *suspende()
{
    int i = 23;
    return &i;
}
```

Fig. 7.16. Un programa en C que deja que p apunte a una posición de memoria desasignada.

Asignación por medio de un montículo

No puede utilizarse la estrategia de asignación por medio de una pila, estudiada anteriormente, si ocurre una de estas dos cosas:

1. Se deben retener los valores de los nombres locales cuando finaliza una activación.
2. Una activación llamada sobrevive al autor de la llamada. Este caso no es posible en aquellos lenguajes en que los árboles de activaciones representan correctamente el flujo de control entre los procedimientos.

En cada uno de los casos anteriores, la desasignación de los registros de activación no tiene por qué ocurrir de la forma *último en entrar – primero en salir*, así que la memoria no se puede organizar como una pila.

La asignación por medio de un montículo divide partes de memoria contigua, conforme las necesiten los registros de activación u otros objetos. Las distintas partes se pueden desasignar en cualquier orden, de modo que con el paso del tiempo el montículo constará de áreas alternas, libres y bajo utilización.

Se puede observar la diferencia entre la asignación con pila y con montículo de los registros de activación en las figuras 7.17 y 7.13. En la figura 7.17, se retiene el registro para una activación del procedimiento `1` cuando finaliza la activación. Por tanto, el registro para la nueva activación `c(1,9)` no puede seguir físicamente al registro para `o`, como sucedió en la figura 7.13. Pero si el registro de activación retenido para `1` se desasigna, habrá espacio libre en el montículo entre los registros de activación para `o` y `c(1,9)`. El que maneja montículo puede utilizar dicho espacio.

La gestión eficiente del montículo es un aspecto bastante especializado en la teoría de las estructuras de datos; en la sección 7.8 se revisan algunas técnicas. Por lo general se asocia una pérdida de tiempo y espacio con el uso de un gestor del montículo. Para una mayor eficacia, puede resultar útil manejar registros de activación

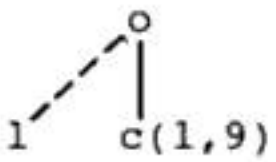
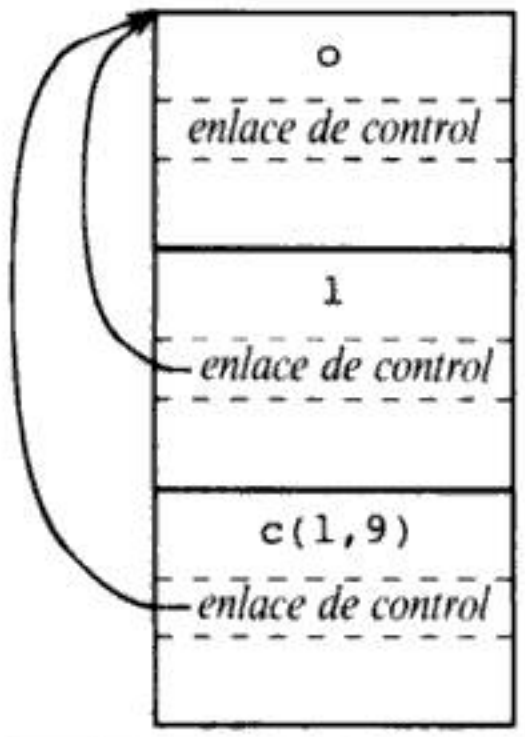
Posición en el Árbol de Activación	Registros de Activación en el Montículo	Comentarios
		Registro de activación retenido para 1

Fig. 7.17. Los registros de las activaciones en curso no necesitan ser adyacentes en un montículo.

pequeños o registros de tamaño previsible como caso especial, de la siguiente manera:

1. Para cada tamaño considerado, hágase una lista enlazada de bloques libres de dicho tamaño.
2. Si es posible, rellénese una solicitud para un tamaño s con un bloque de tamaño s', donde s' sea el tamaño más pequeño mayor o igual a s. Cuando finalmente se desasigne el bloque, se devuelve a la lista enlazada de la que proviene.
3. Para grandes bloques de memoria utilícese el gestor del montículo.

Este enfoque da como resultado una rápida asignación y desasignación en cantidades pequeñas de memoria, puesto que tomar y devolver un bloque de una lista enlazada son operaciones eficientes. Para grandes cantidades de memoria se supone que el cálculo tardará algún tiempo en consumir la memoria, de modo que el tiempo empleado por el asignador es a menudo insignificante comparado con el tiempo utilizado para hacer los cálculos.

7.4 ACCESO A NOMBRES NO LOCALES

Las estrategias para asignación de memoria de la última sección se adaptan en esta sección para permitir el acceso a nombres no locales. Aunque el análisis se basa en la asignación por medio de una pila de los registros de activación, las mismas ideas sirven para la asignación por medio de un montículo.

Las reglas de ámbito de un lenguaje determinan el enfoque de las referencias a los nombres no locales. Una regla común, llamada *regla de ámbito léxico* o *regla de*

ámbito estático, determina la declaración que se aplica a un nombre con sólo examinar el texto del programa. Pascal, C y Ada son algunos de los muchos lenguajes que utilizan ámbito estático, con una estipulación "anidamiento más cercano" añadida que se estudiará más adelante. Otra regla llamada *regla de ámbito dinámico*, determina la declaración aplicable a un nombre durante la ejecución, considerando las actividades en curso. LISP, APL y SNOBOL son algunos lenguajes que utilizan ámbito dinámico.

Se comenzará con bloques y la regla del "anidamiento más cercano". Después se considerarán los nombres no locales en lenguajes como C, donde los ámbitos son léxicos, donde todos los nombres no locales se pueden enlazar a memoria asignada estáticamente, y donde no se permite la declaración de procedimientos anidados.

En lenguajes como Pascal, que tienen procedimientos anidados y ámbito léxico, los nombres que pertenecen a distintos procedimientos pueden ser parte del ambiente en un momento dado. Se estudian dos formas de encontrar los registros de activación que contienen la memoria enlazada a nombres no locales: enlaces de acceso y estructuras de datos tipo *display*.

Una subsección final estudia la implantación del ámbito dinámico.

Bloques

Un bloque es una proposición que contiene sus propias declaraciones de datos locales. El concepto de bloque nació con ALGOL. En C, un bloque tiene la sintaxis

{ *declaraciones proposiciones* }

Una característica de los bloques es su estructura de anidamiento. Los delimitadores marcan el comienzo y el final de un bloque. C utiliza las llaves { y } como delimitadores, mientras que la tradición de ALGOL es utilizar `begin` y `end`. Los delimitadores garantizan que un bloque sea independiente de otro, o que esté anidado dentro de otro. Es decir, no es posible que dos bloques B_1 y B_2 se superpongan de modo que el primero, B_1, comienza, después B_2, pero B_1 termina antes que B_2. Esta propiedad de anidamiento a menudo se denomina *estructura de bloques*.

El ámbito de una declaración en un lenguaje con estructura de bloques viene dado por la regla del *anidamiento más cercano*:

1. El ámbito de una declaración en un bloque B incluye B.
2. Si un nombre x no está declarado en un bloque B, entonces un caso de x en B está en el ámbito de una declaración de x en un bloque abarcador B' tal que
   i) B' tiene una declaración de x, y
   ii) B' está anidado más cerca alrededor de B que cualquier otro bloque con una declaración de x.

Debido al diseño, cada declaración de la figura 7.18 asigna como valor inicial al nombre declarado el número del bloque en el que aparece. El ámbito de la declaración de b en B_0 no incluye a B_1 porque b está declarada en B_1, indicado por $B_0 - B_1$ en la figura. Dicho vacío se llama *excepción* en el ámbito de la declaración.

La regla del ámbito del anidamiento más cercano se refleja en el resultado del programa de la figura 7.18. El control fluye a un bloque desde el punto inmediata-

```
main()
{
    int a = 0;
    int b = 0;
    {
        int b = 1;
        {
            int a = 2;
            printf("%d %d\n", a, b);
        }
        {
            int b = 3;
            printf("%d %d\n", a, b);
        }
    }
    printf("%d %d\n", a, b);
}
```

B_0 B_1 B_2 B_3

DECLARACIÓN	ÁMBITO
`int a = 0;`	$B_0 - B_2$
`int b = 0;`	$B_0 - B_1$
`int b = 1;`	$B_1 - B_3$
`int a = 2;`	B_2
`int b = 3;`	B_3

Fig. 7.18. Bloques en un programa en C.

mente anterior a él, y fluye desde el bloque al punto inmediatamente posterior a él en el texto fuente. Por tanto, las proposiciones de impresión se ejecutan en el orden B_2, B_3, B_1 y B_0, orden en que el control sale de los bloques. Los valores de `a` y `b` en estos bloques son:

```
2 1
0 3
0 1
0 0
```

La estructura de bloques se puede implantar utilizando la asignación por medio de una pila. Como el ámbito de una declaración no se aplica fuera del bloque en que aparece, se puede asignar el espacio para el nombre declarado cuando se entra al bloque y desasignar cuando el control sale del bloque. Este punto de vista considera al bloque como un "procedimiento sin parámetros", llamado únicamente desde el punto inmediatamente anterior al bloque y regresando únicamente al punto inmediatamente posterior al bloque. Se puede mantener el ambiente no local para un bloque utilizando las técnicas para procedimientos que aparecen más adelante en esta sección. Obsérvese, sin embargo, que los bloques son más simples que los procedimientos porque no se pasan parámetros y porque el flujo del control desde y hacia un bloque sigue muy de cerca el texto del bloque estático[5].

---

[5] Se puede implantar un salto fuera de un bloque hasta un bloque abarcador sacando de la pila los registros de activación de los bloques intermedios. Algunos lenguajes permiten un salto adentro de un bloque. Antes de que se transfiera el control de esta manera, se tienen que establecer los registros de activación para los bloques intermedios. La semántica del lenguaje determina cómo se inicializan los datos locales en estos registros de activación.

Una implantación alternativa es asignar de una sola vez memoria para un cuerpo de un procedimiento completo. Si hay bloques dentro del procedimiento, entonces la asignación se hace para la memoria necesaria por las declaraciones dentro de los bloques. Para el bloque B_0 de la figura 7.18, se puede asignar memoria como en la figura 7.19. Los subíndices en las variables locales a y b identifican a los bloques en los que están declaradas las variables locales. Obsérvese que a a_2 y a b_3 se puede asignar la misma memoria porque están en bloques que no se activan a la vez.

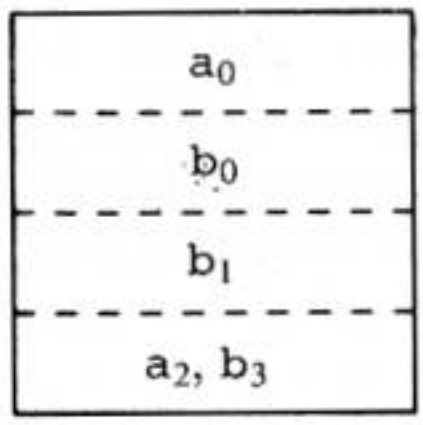


Fig. 7.19. Almacenamiento de los nombres declarados en la figura 7.18.

En ausencia de datos de longitud variable, la cantidad máxima de memoria necesaria durante la ejecución de un bloque puede determinarse durante la compilación. (Se pueden manejar los datos de longitud variable utilizando apuntadores como en la Sec. 7.3.) Al realizar dicha determinación, convencionalmente se supone que se pueden tomar de hecho todas las trayectorias de control en el programa. Es decir, se supone que se pueden ejecutar tanto la parte **then** como la parte **else** de una proposición condicional, y que se pueden alcanzar todas las proposiciones dentro de un bucle **while**.

Ambito léxico sin procedimientos anidados

Las reglas de ámbito léxico para C son más sencillas que las de Pascal, que se estudian a continuación, porque las definiciones de procedimientos no se pueden anidar en C. Es decir, una definición de procedimiento no puede aparecer dentro de otra. Como en la figura 7.20, un programa en C consta de una secuencia de declaraciones de variables y procedimientos (C les llama funciones). Si hay una referencia no local a un nombre a en alguna función, entonces a se debe declarar fuera de cualquier función. El ámbito de una declaración fuera de una función consta de los cuerpos de las funciones que vayan después de la declaración, con agujeros si se declara el nombre dentro de una función. En la figura 7.20, los casos no locales de a en `leematriz`, `partición` y `main` se refieren a la matriz declarada en la línea 1.

```
(1) int a[11];
(2) leematriz() { ... a ... }
(3) int partición(y,z) int y, z; { ... a ... }
(4) clasificación_por_particiones(m,n) int m, n; { ... }
(5) main() { ... a ... }
```

Fig. 7.20. Programa en C con casos no locales de a.

En ausencia de procedimientos anidados, la estrategia de asignación mediante pilas para los nombres locales de la sección 7.3 se puede utilizar directamente para un lenguaje con ámbito léxico como C. La memoria para todos los nombres declarados fuera de cualquier procedimiento se puede asignar estáticamente. La posición de esta memoria se conoce durante la compilación, de modo que si un nombre es no local en el cuerpo de algún procedimiento, se utiliza simplemente la dirección estáticamente determinada. Cualquier otro nombre debe ser local a la activación del tope de la pila, accesible a través del apuntador *tope*. Los procedimientos anidados hacen que falle este esquema porque entonces nombres no locales pueden referirse a datos del fondo de la pila, como se estudiará más adelante.

Una ventaja importante de la asignación estática para los nombres no locales es que los procedimientos declarados se pueden pasar libremente como parámetros y devolverse como resultado (en C se pasa una función pasando un apuntador a ella). Con ámbito léxico y sin procedimientos anidados, cualquier nombre que sea no local a un procedimiento es no local a todos los procedimientos. Su direccion estática puede ser utilizada por todos los procedimientos, independientemente de cómo se activen. Asimismo, si los procedimientos se devuelven como resultados, los nombres no locales en el procedimiento devuelto se refieren a la memoria asignada estáticamente para ellos.

Por ejemplo, considérese el programa en Pascal de la figura 7.21. Todos los casos del nombre `m`, que se muestran dentro de un círculo en la figura 7.21, están dentro del ámbito de la declaración de la línea 2. Como `m` es no local a todos los procedimientos del programa, se puede asignar su memoria estáticamente. Cuando se ejecuten los procedimientos `f` y `g`, pueden utilizar la dirección estática para acceder al valor de `m`. El hecho de que `f` y `g` se pasen como parámetros sólo influye cuando se activan y no influye en la forma de acceder al valor de `m`.

Más concretamente, la llamada `b(f)` en la línea 11 asocia la función `f` con el parámetro formal `h` del procedimiento `b`. De modo que cuando se llama al parámetro formal `h` en la línea 8, en `write(h(2))`, se activa la función `f`. La activación de `f` devuelve `2` porque el nombre no local `m` tiene el valor de `0` y el parámetro

```
(1)  program paso(input, output);
(2)      var ⓜ : integer;
(3)      function f(n : integer) : integer;
(4)          begin f := ⓜ + n end { f };
(5)      function g(n : integer) : integer;
(6)          begin g := ⓜ * n end { g };
(7)      procedure b(function h(n : integer) : integer);
(8)          begin write(h(2)) end { b };
(9)      begin
(10)     ⓜ := 0;
(11)     b(f); b(g); writeln
(12)     end.
```

Fig. 7.21. Programa en Pascal con casos no locales de `m`.

formal `n` tiene el valor de 2. Después, la llamada `b(g)` asocia `g` con `h`; esta vez, una llamada a `h` activa `g`. La salida de este programa es

```
2          0
```

Ambito léxico con procedimientos anidados

Un caso no local de un nombre `a` en un procedimiento en Pascal está dentro del alcance de la declaración anidada más cercana de `a` en el texto del programa estático.

El anidamiento de definiciones de procedimientos en el programa en Pascal de la figura 7.22 está indicado por la siguiente indentación:

```
ordenamiento
    leematriz
    intercambio
    clasificación_por_particiones
        partición
```

El caso de `a` en la línea 15 de la figura 7.22 está dentro de la función `partición`, anidada dentro del procedimiento `clasificación_por_particiones`. La declaración anidada más cercana de `a` está en la línea 2 dentro del procedimiento for-

```
(1)  program ordenamiento(input, output);
(2)      var a : array [0..10] of integer;
(3)          x : integer;
(4)      procedure leematriz;
(5)          var i : integer;
(6)          begin ... a ... end { leematriz };
(7)      procedure intercambio(i, j: integer);
(8)          begin
(9)              x := a[i]; a[i] := a[j]; a[j] := x
(10)         end;
(11)     procedure clasificación_por_particiones(m, n: integer);
(12)         var k, v : integer;
(13)         function partición(y, z: integer) : integer;
(14)             var i, j : integer;
(15)             begin ... a ...
(16)                   ... v ...
(17)                   ... intercambio(i,j); ...
(18)             end { partición } ;
(19)         begin ... end { clasificación_por_particiones };
(20)     begin ... end { ordenamiento } .
```

Fig. 7.22. Un programa en Pascal con procedimientos anidados.

mado por el programa completo. La regla del anidamiento más cercano se aplica también a nombres de procedimientos. El procedimiento `intercambio`, que es llamado por `partición` en la línea 17, es no local a `partición`. Aplicando la regla, primero se comprueba si `intercambio` está definido dentro de `clasificación_por_particiones`; como no lo está, se busca en el programa principal `ordenamiento`.

Profundidad de anidamiento

La noción de *profundidad de anidamiento* de un procedimiento se utiliza más adelante para implantar el ámbito léxico. Se considera que el nombre del programa principal está a profundidad de anidamiento 1; se suma 1 a la profunidad de anidamiento al pasar de un procedimiento abarcador a un procedimiento abarcado. En la figura 7.22, el procedimiento `clasificación_por_particiones` de la línea 11 está a profundidad de anidamiento 2, mientras que `partición` en la línea 13 está a profundidad de anidamiento 3. Con cada caso de un nombre, se asocia la profundidad de anidamiento del procedimiento en el cual está declarado. Los casos de `a`, `v` e `i` en las líneas 15 a 17 en `partición` tienen por tanto profundidades de anidamiento 1, 2 y 3, respectivamente.

Enlaces de acceso

Una implantación directa del ámbito léxico para procedimientos anidados se obtiene añadiendo un apuntador, llamado *enlace de acceso*, a cada registro de activación. Si el procedimiento `p` está anidado inmediatamente dentro de `c` en el texto fuente, entonces el enlace de acceso en un registro de activación para `p` apunta al enlace de acceso para la activación más reciente de `c`.

En la figura 7.23 se muestran algunos estados de la pila para la ejecución durante una ejecución del programa de la figura 7.22. De nuevo, para ahorrar espacio en la figura, sólo se muestra la primera letra del nombre de cada procedimiento. El enlace de acceso para la activación de `ordenamiento` está vacío, porque no hay un procedimiento que la abarque. En enlace de acceso para cada activación de `clasificación_por_particiones` apunta al registro de `ordenamiento`. Obsérvese en la figura 7.23(c) que el enlace de acceso en el registro de activación para `partición(1,3)` apunta al enlace de acceso en el registro de la activación más reciente de `clasificación_por_particiones`, es decir, `clasificación_por_particiones(1,3)`.

Supóngase que el procedimiento `p` a profundidad de anidamiento n_p hace referencia a un nombre no local `a` con profundidad de anidamiento $n_a \leq n_p$. La dirección de memoria para `a` se puede encontrar de la siguiente manera:

1. Cuando el control está en `p`, un registro de activación para `p` está en el tope de la pila. Sígase $n_p - n_a$ enlaces de acceso desde el registro que está en el tope de la pila. El valor de $n_p - n_a$ se puede precalcular durante la compilación. Si el enlace de acceso en un registro apunta al enlace de acceso en otro, entonces se puede seguir un enlace realizando una sola operación de indirección.

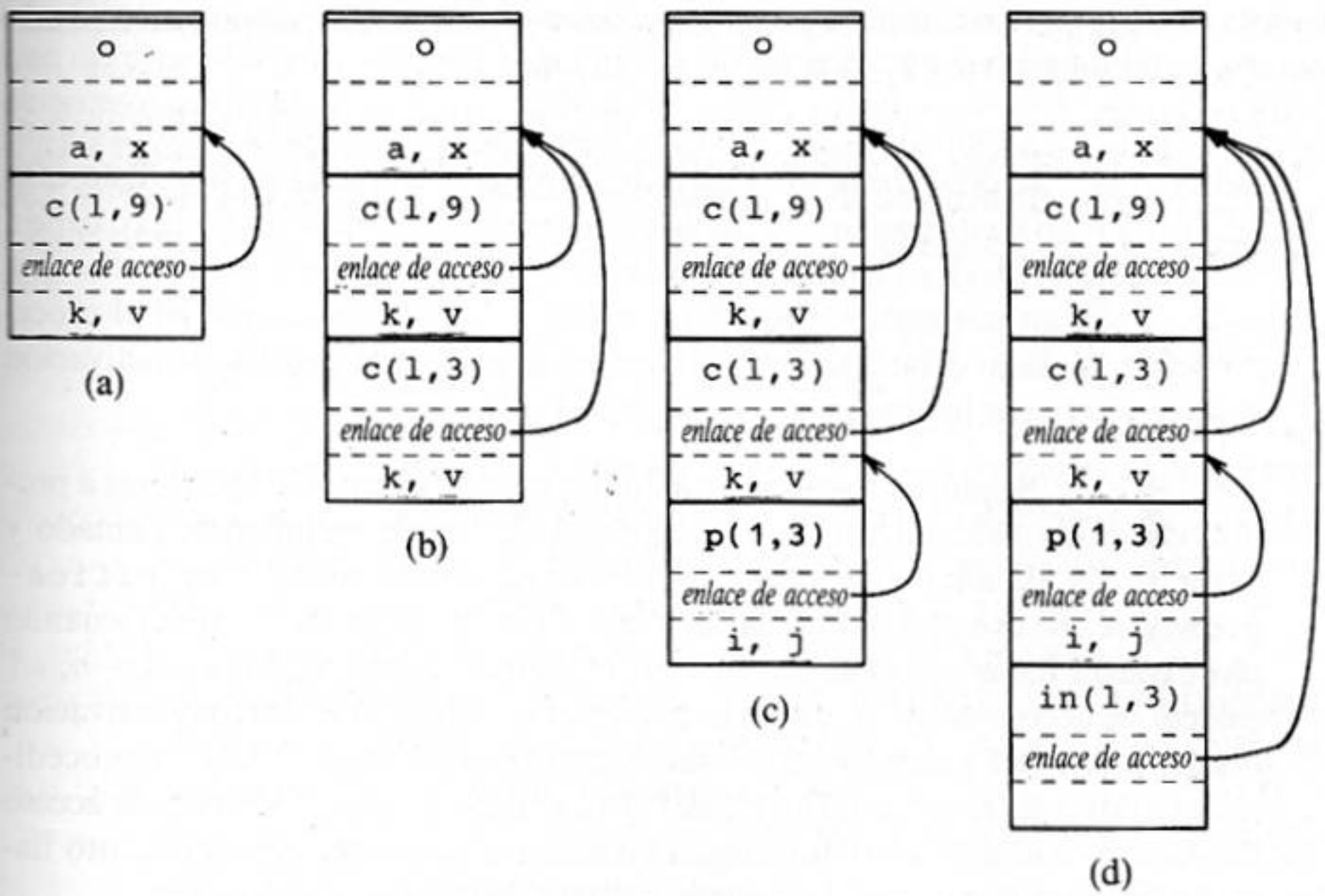


Fig. 7.23. Enlaces de acceso para encontrar las posiciones de memoria para los nombres no locales.

2. Después de seguir $n_p - n_a$ enlaces, se llega a un registro de activación para el procedimiento al que es local a. Como se estudió en la última sección, su ubicación en la memoria está en un desplazamiento determinado con respecto a una posición en el registro. En concreto, el desplazamiento puede ser con respecto al enlace de acceso.

Por tanto, la dirección del nombre no local a en el procedimiento p viene dada por el siguiente par calculado en el momento de la compilación y guardado en la tabla de símbolos:

($n_p - n_a$, desplazamiento dentro del registro de activación que contiene a)

El primer componente proporciona el número de enlaces de acceso que deben recorrerse.

Por ejemplo, en las líneas 15 y 16 de la figura 7.22, el procedimiento `partición` a profundidad de anidamiento 3 hace referencia a los nombres no locales a y v a profundidades de anidamiento 1 y 2, respectivamente. Los registros de activación que contienen las direcciones de memoria para dichos nombres no locales se encuentran siguiendo $3-1=2$ y $3-2=1$ enlaces de acceso, respectivamente, desde el registro correspondiente a `partición`.

El código para establecer los enlaces de acceso es parte de la secuencia de llamada. Supóngase que el procedimiento p a profundidad de anidamiento n_p llama al procedimiento x a profundidad de anidamiento n_x. El código para establecer el en-

lace de acceso en el procedimiento llamado depende de si el procedimiento llamado está anidado o no dentro del que efectúa la llamada.

1. Caso $n_p < n_x$. Como el procedimiento llamado `x` está anidado a mayor profundidad que `p`, se debe declarar dentro de `p`, o no sería accesible a `p`. Este caso ocurre cuando `ordenamiento` llama a `clasificación_por_particiones` en la figura 7.23(a) y cuando `clasificación_por_particiones` llama a `partición` en la figura 7.23(c). En este caso, el enlace de acceso en el procedimiento llamado debe apuntar al enlace de acceso en el registro de activación del que efectúa la llamada justo debajo en la pila.

2. Caso $n_p \geq n_x$. Según las reglas de ámbito, los procedimientos abarcadores a profundidades de anidamiento 1, 2, . . . , $n_x - 1$ de los procedimientos llamado y autor de la llamada deben ser los mismos, como cuando `clasificación_por_particiones` se llama a sí mismo en la figura 7.23(b) y cuando `partición` llama a `intercambio` en la figura 7.23(d). Siguiendo $n_p - n_x + 1$ enlaces de acceso desde el autor de la llamada, se llega al registro de activación más reciente del procedimiento que estáticamente abarca tanto al procedimiento llamado como al autor de la llamada más cercano. El enlace de acceso alcanzado es al que debe apuntar el enlace de acceso del procedimiento llamado. De nuevo, $n_p - n_x + 1$ se puede calcular durante la compilación.

Procedimientos como parámetros

Las reglas de ámbito léxico se aplican aunque un procedimiento anidado se pase como parámetro. La función `f` en las líneas 6 y 7 del programa en Pascal de la figura 7.24 tiene un nombre no local `m`; todos los casos de `m` se muestran en un círculo. En la línea 8, el procedimiento `c` asigna `0` a `m` y después pasa `f` como parámetro a `b`. Obsérvese que el ámbito de la declaración de `m` en la línea 5 no incluye el cuerpo de `b` de las líneas 2 y 3.

Dentro del cuerpo de `b`, la proposición `writeln(h(2))` activa `f` porque el parámetro formal `h` se refiere a `f`. Es decir, `writeln` imprime el resultado de la llamada `f(2)`.

```
(1)  program param(input, output);
(2)      procedure b(function h(n:integer): integer);
(3)          begin writeln(h(2)) end { b };
(4)      procedure c;
(5)          var ⓜ : integer;
(6)          function f(n : integer) : integer;
(7)              begin f := ⓜ + n end { f };
(8)          begin ⓜ := 0; b(f) end { c };
(9)      begin
(10)     c
(11)     end.
```

Fig. 7.24. Se debe pasar un enlace de acceso con el parámetro actual `f`.

¿Cómo se establece el enlace de acceso para la activación de f? La respuesta es que un procedimiento anidado que se pasa como parámetro debe tomar su enlace de acceso junto con él, como se muestra en la figura 7.25. Cuando el procedimiento c pasa f, determina un enlace de acceso para f, si estuviera llamando a f. Este enlace se pasa junto con f a b. Por tanto, cuando f se activa desde dentro de b, el enlace se utiliza para establecer el enlace de acceso en el registro de activación para f.

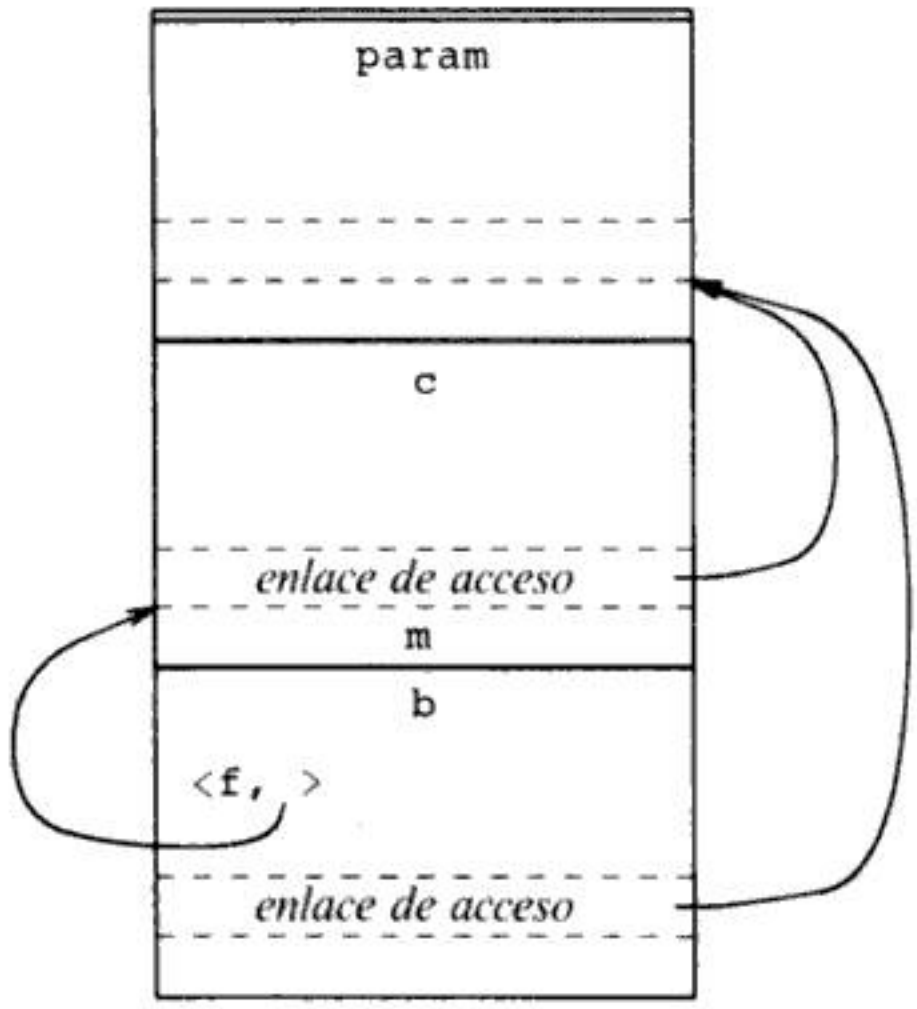


Fig. 7.25. El procedimiento como parámetro actual f lleva su enlace de acceso.

Estructura de datos tipo display

Se puede obtener un acceso más rápido que con los enlaces de acceso a los nombres no locales utilizando una matriz *d* de apuntadores a registros de activación, llamada *display*. Se mantiene el *display* de modo que la dirección de memoria para un nombre no local a a profundidad de anidamiento *i* esté en el registro de activación apuntado por el elemento *d* [*i*] del *display*.

Supóngase que el control está en la activación de un procedimiento p a profundidad de anidamiento *j*. Entonces, los primeros $j-1$ elementos del *display* apuntan a las activaciones más recientes de los procedimientos que lexicográficamente abarcan el procedimiento p, y *d* [*j*] apunta a la activación de p. Utilizar un *display* es generalmente más rápido que seguir los enlaces de acceso porque se encuentra un registro de activación que guarda un nombre no local accediendo a un elemento de *d* y siguiendo después sólo un apuntador.

Una solución sencilla para mantener el *display* es la de utilizar enlaces de acceso además del *display*. Como parte de las secuencias de llamada y de regreso, el *display* se actualiza siguiendo la cadena de enlaces de acceso. Cuando sigue el enlace de acceso a un registro de activación a profundidad de anidamiento *n*, el elemento del

display d [n] apunta a dicho registro de activación. En realidad, el *display* duplica la información de la cadena de enlaces de acceso.

Esta colocación puede mejorarse. El método ilustrado en la figura 7.26 exige menos trabajo en la entrada y salida de un procedimiento si, como es habitual, los procedimientos no se pasan como parámetros. En la figura 7.26 el *display* consta en una matriz global, independiente de la pila. Las situaciones que se muestran en la figura se refieren a una ejecución del texto fuente de la figura 7.22. De nuevo, sólo se muestra la primera letra del nombre de cada procedimiento.

En la figura 7.26(a) se muestra la situación justo antes de que comience la activación `c(1,3)`. Como `clasificación_por_particiones` está a profundidad de anidamiento 2, influye en el elemento del *display* d [2] cuando comienza una nueva activación de `clasificación_por_particiones`. En la figura 7.26(b) se muestra el efecto de la activación `c(1,3)` sobre d [2], donde d [2] apunta ahora al nuevo registro de activación; el valor anterior de d [2] se guarda dentro del nuevo registro de activación[6]. Posteriormente se necesitará este valor para restablecer el *display* a su estado de la figura 7.26(a), cuando el control regrese de la activación `c(1,9)`.

El *display* cambia cuando se produce una nueva activación y se debe restablecer cuando el control retorna de la nueva activación. Las reglas de ámbito de Pascal y otros lenguajes de ámbito léxico permiten que se mantenga el *display* mediante los siguientes pasos. Sólo se estudia el caso más fácil, donde los procedimientos no se pasan como parámetros (véase Ejercicio 7.8). Cuando se establece un nuevo registro de activación para un procedimiento a profundidad de anidamiento i,

1. se guarda el valor de $d[i]$ dentro del nuevo registro de activación y
2. se apunta $d[i]$ al nuevo registro de activación.

Justo antes de que finalice una activación, se restablece $d[i]$ al valor guardado.

Estos pasos tienen la siguiente justificación. Supóngase que un procedimiento a profundidad de anidamiento j llama a un procedimiento que se encuentra a profundidad i. Existen dos casos, que dependen de si el procedimiento llamado está anidado o no dentro del llamado en el texto fuente, como en el análisis de los enlaces de acceso.

1. Caso $j < i$. Entonces, $i=j+1$ y el procedimiento llamado está anidado dentro del que efectúa la llamada. Los primeros j elementos del *display* no necesitan por tanto modificarse, y se asigna la dirección del nuevo registro de activación a d [i]. Este caso se ilustra en la figura 7.26(a) cuando `ordenamiento` llama a `clasificación_por_particiones` y también cuando `clasificación_por_particiones` llama a `partición` en la figura 7.26(c).

2. Caso $j \geq i$. De nuevo, los procedimientos abarcadores a profundidades de anidamiento 1, 2, . . . , $i-1$ de los procedimientos receptor y autor de la llamada deben ser los mismos. En este caso, se guarda el valor anterior de d [i] en el nuevo

---

[6] Obsérvese que `c(1,9)` también guardó d[2], aunque sucede que el segundo elemento del *display* nunca había sido utilizado y no es necesario restituirlo. Para todas las llamadas de `c` es más fácil guardar d[2] que decidir durante la ejecución si es necesario hacerlo o no.

registro de activación, y se apunta $d[i]$ al nuevo registro de activación. El *display* se mantiene correctamente porque los primeros $i-1$ elementos se dejan como están.

Ocurre un ejemplo del caso 2, con $i=j=2$, cuando `clasificación_por_particiones` es llamado recursivamente en la figura 7.26(a). Un ejemplo más interesante es el de la activación `p(1,3)` a profundidad de anidamiento 3 que llama a `in(1,3)`, a profundidad 2, y el procedimiento abarcador de ambas es `o` a profundidad 1, como en la figura 7.26(d). (El programa está en la Fig. 7.22.) Obsérvese que cuando se llama a `in(1,3)`, el valor de $d[3]$ que pertenece a `p(1,3)` está todavía en el *display*, aunque no se puede acceder a él mientras el control esté en `in`. Si `in` llamara a otro procedimiento a profundidad 3, este

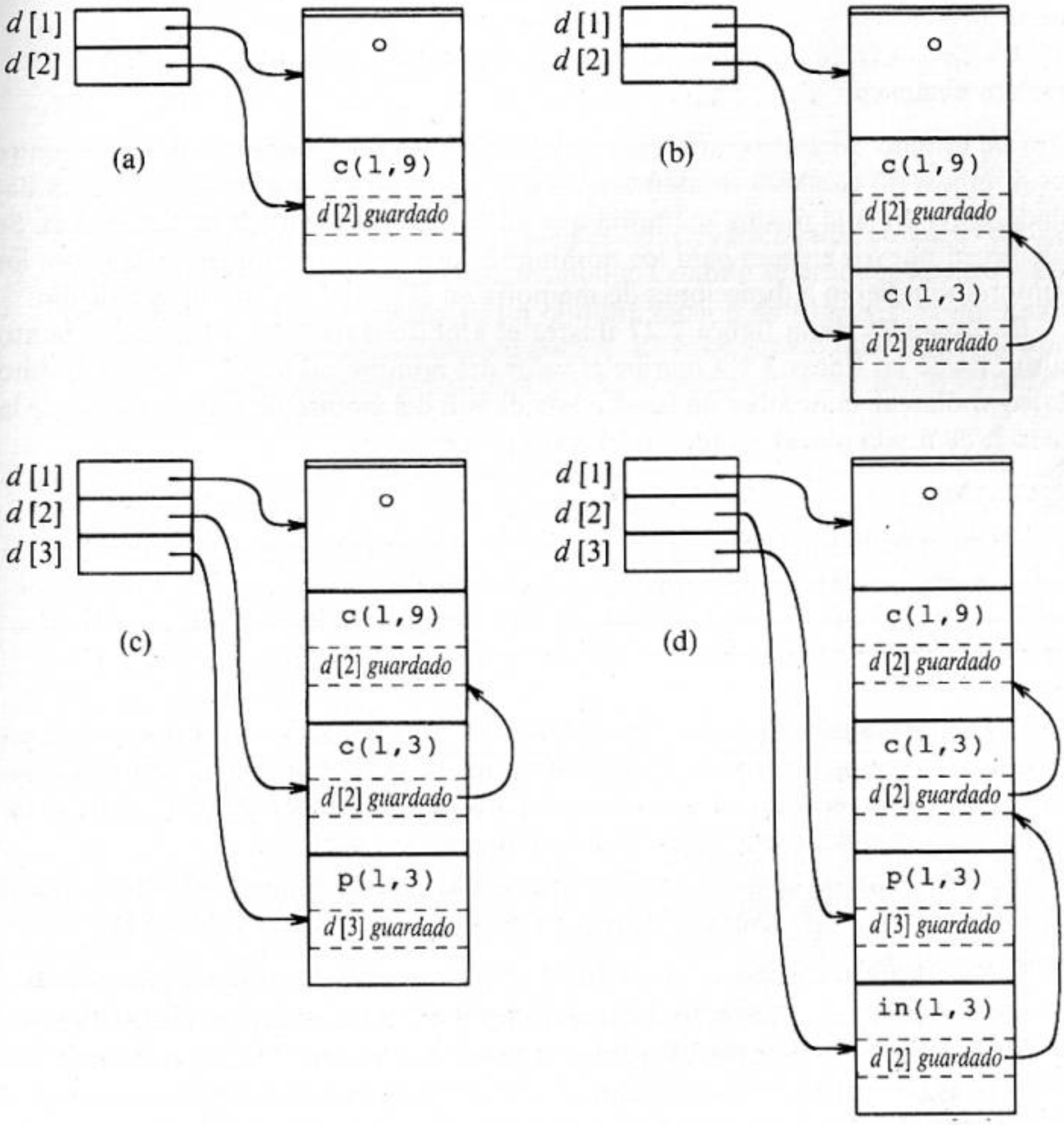


Fig. 7.26. Mantenimiento del *display* cuando no se pasan procedimientos como parámetros.

procedimiento guardaría *d* [3] y lo restablecería al regresar a `in`. Así se demuestra que cada procedimiento observa el *display* correcto para todas las profundidades, hasta su propia profundidad.

Hay varios lugares donde se puede mantener un *display*. Si hay suficientes registros, entonces el *display*, considerado como una matriz, puede ser un conjunto de registros. Obsérvese que el compilador puede determinar la longitud máxima de esta matriz; es la profundidad máxima de anidamiento de los procedimientos en el programa. En caso contrario, el *display* se puede conservar en memoria asignada estáticamente y todas las referencias a los registros de activación comienzan utilizando direccionamiento indirecto por medio del apuntador del *display* adecuado. Este enfoque es el apropiado para una máquina con direccionamiento indirecto, aunque cada indirección cuesta un ciclo de máquina. Otra posibilidad es guardar el *display* en la misma pila de ejecución, y crear una nueva copia a cada entrada de procedimiento.

Ambito dinámico

Con un ámbito dinámico, una nueva activación hereda los enlaces existentes entre los nombres no locales a la memoria. Un nombre no local `a` en la activación llamada se refiere a la misma memoria que en la activación autora de la llamada. Se establecen nuevos enlaces para los nombres locales del procedimiento llamado; los nombres se refieren a direcciones de memoria en el nuevo registro de activación.

El programa de la figura 7.27 ilustra el ámbito dinámico. El procedimiento `muestra` de las líneas 3 y 4 escribe el valor del nombre no local `r`. Con el ámbito léxico en Pascal, el nombre no local `r` está dentro del ámbito de la declaración de la línea 2, de modo que el resultado del programa es

```
0.250 0.250
0.250 0.250
```

Sin embargo, con un ámbito dinámico, el resultado es

```
0.250 0.125
0.250 0.125
```

Cuando se llama a `muestra` en las líneas 10 y 11 del programa principal, se escribe `0.250` porque se utiliza la variable `r` que es local al programa principal. Sin embargo, cuando se llama a `muestra` en la línea 7 desde dentro de `chico`, se escribe `0.125` porque se utiliza la variable `r` que es local a `chico`.

Los dos enfoques siguientes para implantar el ámbito dinámico guardan cierto parecido con el uso de enlaces de acceso y *displays*, respectivamente, en la implantación del alcance léxico.

1. *Acceso profundo.* Conceptualmente, el ámbito dinámico da resultado si los enlaces de acceso apuntan al mismo registro de activación al que apuntan los enlaces de control. Una aplicación sencilla es prescindir de los enlaces de acceso y utilizar el enlace de control para examinar la pila, en busca del primer registro de activación que contenga la memoria para el nombre no local. El término *acceso profundo* se deriva de que la búsqueda en la pila puede hacerse "en pro-

```
(1)  program dinámico(input, output);
(2)      var r : real;
(3)      procedure muestra;
(4)          begin write( r : 5:3 ) end;
(5)      procedure chico;
(6)          var r : real;
(7)          begin r := 0.125; muestra end;
(8)      begin
(9)          r := 0.25;
(10)         muestra; chico; writeln;
(11)         muestra; chico; writeln
(12)     end.
```

Fig. 7.27. El resultado depende de si se utiliza ámbito léxico o dinámico.

fundidad". La profundidad a la que puede llegar la búsqueda depende de la entrada al programa y no se puede determinar durante la compilación.

2. *Acceso superficial.* En este caso, la idea es conservar el valor en curso de cada nombre en memoria asignada estáticamente. Cuando se produce una nueva activación de un procedimiento p, un nombre local n dentro de p se apropia de la memoria asignada estáticamente para n. El valor previo de n se puede guardar en el registro de activación para p y se debe restablecer cuando termine la activación de p.

La diferencia entre los dos enfoques es que el acceso profundo tarda más en acceder a un nombre no local, pero no tiene pérdidas relacionadas con el comienzo y final de una activación. Por otra parte, el acceso superficial permite el acceso directo a los nombres no locales, pero se emplea tiempo en mantener dichos valores cuando las activaciones comienzan y terminan. Cuando las funciones se pasan como parámetros y se devuelven como resultados, se obtiene una implantación más directa con el acceso profundo.

7.5 PASO DE PARAMETROS

Cuando un procedimiento llama a otro, el método habitual de comunicación entre ellos es a través de nombres no locales y a través de parámetros del procedimiento llamado. En la figura 7.28 se utilizan tanto los nombres no locales como los parámetros para intercambiar los valores de `a[i]` y `a[j]`. Aquí la matriz `a` es no local al procedimiento `intercambio`, e `i` y `j` son parámetros.

En esta sección se estudian varios métodos para asociar parámetros actuales y formales: llamada por valor, llamada por referencia, copia y restauración, llamada por nombre y macro expansión. Es importante conocer el método de paso de parámetros que utiliza un lenguaje (o compilador), porque el resultado de un programa puede depender del método empleado.

¿Por qué tantos métodos? Los distintos métodos surgen de diferentes interpretaciones en cuanto a lo que representa una expresión. En una asignación como

```
a[i] := a[j]
```

la expresión `a[j]` representa un valor, mientras que `a[i]` representa una posición de memoria dentro de la que se coloca el valor de `a[j]`. La decisión de si utilizar la posición o el valor representado por una expresión viene determinada por el hecho de que la expresión aparezca en el lado izquierdo o en el derecho, respectivamente, del signo de asignación. Como en el capítulo 2, el término *valor de lado izquierdo* se refiere a la posición de memoria representada por una expresión y el término *valor de lado derecho* se refiere al valor contenido de una posición de memoria.

```
(1)  procedure intercambio(i, j: integer);
(2)      var x : integer;
(3)      begin
(4)          x := a[i]; a[i] := a[j]; a[j] := x
(5)      end
```

Fig. 7.28. El procedimiento en Pascal `intercambia` con nombres no locales y parámetros.

Las diferencias entre los métodos de paso de parámetros se basan principalmente en que un parámetro actual representa un valor de lado izquierdo, un valor de lado derecho o el texto del mismo parámetro actual.

Llamada por valor

En cierto modo, este es el método más sencillo posible de pasar parámetros. Los parámetros actuales se evalúan y sus valores de lado derecho se pasan al procedimiento llamado. La llamada por valor se usa en C, y generalmente los parámetros se pasan en Pascal de esta manera. Hasta ahora todos los programas de este capítulo han seguido este método para pasar parámetros. La llamada por valor se puede implantar como sigue.

1. Un parámetro formal se considera como un nombre local, de modo que las direcciones de memoria para los parámetros formales se encuentran en el registro de activación del procedimiento llamado.
2. El procedimiento autor de la llamada evalúa los parámetros actuales y coloca sus valores de lado derecho en las direcciones de memoria de los parámetros formales.

Una característica distintiva de la llamada por valor es que las operaciones sobre los parámetros formales no afectan a los valores en el registro de activación del autor de la llamada. Si no se tiene en cuenta la palabra clave `var` en la línea 3 de la figura 7.29, Pascal pasará `x` e `y` por valor al procedimiento `permuta`. Entonces la llamada `permuta(a,b)` en la línea 12 no modifica los valores de `a` y `b`. Con la lla-

```
(1)  program referencia(input, output);
(2)  var a, b: integer;
(3)  procedure permuta(var x, y: integer);
(4)      var temp : integer;
(5)      begin
(6)          temp := x;
(7)             x := y;
(8)             y := temp
(9)      end;
(10) begin
(11)     a := 1; b := 2;
(12)     permuta(a,b);
(13)     writeln('a = ',a); writeln('b = ',b)
(14) end.
```

Fig. 7.29. Programa en Pascal con el procedimiento `permuta`.

mada por valor, el efecto de la llamada `permuta(a,b)` es equivalente a la secuencia de pasos

```
   x := a
   y := b
temp := x
   x := y
   y := temp
```

donde `x`, `y` y `temp` son locales a `permuta`. Aunque estas asignaciones cambian los valores de las variables locales `x`, `y` y `temp`, las modificaciones no sirven cuando el control retorna de la llamada y se desasigna el registro de activación para `permuta`. Por tanto, la llamada no afecta al registro de activación del autor de la llamada.

```
(1)  permuta(x,y)
(2)  int *x, *y;
(3)  {   int temp;
(4)      temp = *x; *x = *y; *y = temp;
(5)  }
(6)  main()
(7)  {   int a = 1, b = 2;
(8)      permuta( &a, &b );
(9)      printf("a es ahora %d, b es ahora %d\n",a,b);
(10) }
```

Fig. 7.30. Programa en C que usa apuntadores en un procedimiento llamado por valor.

Un procedimiento llamado por valor puede afectar a su llamador ya sea a través de nombres no locales (véase `intercambio` en la Fig. 7.28) o a través de apuntadores que se pasan explícitamente como parámetros. En el programa en C de la figura 7.30, `x` e `y` son declarados en la línea 2 apuntadores a enteros; el operador `&` en la llamada `permuta(&a,&b)` de la línea 8 hace que los apuntadores a `a` y `b` se pasen a `permuta`. El resultado de este programa es

```
a es ahora 2, b es ahora 1
```

El uso de apuntadores en este ejemplo sugiere cómo intercambiaría valores un compilador que utilice llamada por referencia.

Llamada por referencia

Cuando los parámetros se pasan por *referencia* (conocida también como *llamada por dirección* o *llamada por posición*), el autor de la llamada pasa al procedimiento receptor de la llamada un apuntador a la dirección de memoria de cada parámetro actual.

1. Si un parámetro actual es un nombre o una expresión que tenga un valor de lado izquierdo, entonces se pasa ese mismo valor de lado izquierdo.
2. Sin embargo, si el parámetro actual es una expresión, como `a+b` o `2`, que no tiene ningún valor de lado izquierdo, entonces la expresión se evalúa en una nueva posición, y se pasa la dirección de dicha posición.

Una referencia a un parámetro formal en el procedimiento llamado se convierte, en el código objeto, en una referencia indirecta a través del apuntador pasado al procedimiento llamado.

Ejemplo 7.7. Considérese el procedimiento `permuta` de la figura 7.29. Una llamada a `permuta` con parámetros reales `i` y `a[i]`, es decir, `permuta(i,a[i])`, tendría el mismo efecto que la siguiente secuencia de pasos.

1. Cópiense las direcciones (valores de lado izquierdo) de `i` y `a[i]` dentro del registro de activación del procedimiento llamado, por ejemplo, en las posiciones `arg1` y `arg2` correspondientes a `x` e `y`, respectivamente.
2. Asígnese a `temp` el contenido de la posición apuntada por `arg1` (es decir, iguálese `temp` a I_0, donde I_0 es el valor inicial de `i`). Este paso corresponde a `temp := x` en la línea 6 de la definición de `permuta`.
3. Asígnense los contenidos de la posición apuntada por `arg1` al valor de la posición apuntada por `arg2`; es decir, `i := a`[I_0]. Este paso corresponde a `x := y` en la línea 7 de `permuta`.
4. Asígnese al contenido de la posición apuntada por `arg2` el valor de `temp`; es decir, hágase `a`[I_0] `:= i`. Este paso corresponde a `y := temp`. □

Varios lenguajes utilizan llamada por referencia; los parámetros `var` de Pascal se pasan de esta forma. Las matrices generalmente se pasan por referencia.

Copia y restauración

Un híbrido entre la llamada por valor y la llamada por referencia es el *enlazado de copia y restauración* (también conocido como *copia-dentro, copia-fuera* o *valor y resultado*).

1. Antes de que el control fluya al procedimiento llamado, se evalúan los parámetros actuales. Los valores de lado derecho de los parámetros actuales se pasan al procedimiento llamado como en la llamada por valor. Además, sin embargo, los valores de lado izquierdo de estos parámetros actuales con valores de lado izquierdo se determinan antes de llamada.
2. Cuando el control retorna, los valores de lado derecho en curso de los parámetros formales se copian de retorno en los valores de lado izquierdo de los parámetros reales, utilizando los valores de lado izquierdo calculados antes de la llamada. Sólo se copia en los parámetros actuales con valores de lado izquierdo, por supuesto.

El primer paso "copia dentro" los valores de los parámetros actuales en el registro de activación del procedimiento llamado (en la memoria para los parámetros formales). El segundo paso "copia fuera" los valores finales de los parámetros formales en el registro de activación del que efectúa la llamada (en los valores de lado izquierdo calculados según los parámetros actuales antes de la llamada).

Obsérvese que `permuta(i,a[i])` funciona correctamente utilizando copia y restauración porque la posición de `a[i]` la calcula y preserva el programa que hace la llamada antes de la llamada. Por tanto, el valor final del parámetro formal `y`, que será el valor inicial de `i`, se copia en la posición correcta, aunque la posición de `a[i]` es modificada por la llamada (porque el valor de `i` cambia).

Algunas implantaciones de FORTRAN utilizan el método de copia y restauración. Sin embargo, otras utilizan la llamada por referencia. Pueden aparecer diferencias entre las dos si el procedimiento llamado tiene más de una manera de acceder a una posición en el registro de activación del autor de la llamada. La activación realizada por la llamada `inseguro(a)` en la línea 6 de la figura 7.31 puede acceder a `a` como un nombre no local y a través del parámetro formal `x`. Con la llamada por referencia, las asignaciones a `x` y a `a` afectan inmediatamente a `a`, así que el valor final de `a` es `0`. Sin embargo, con la copia y restauración, el valor `1` del parámetro

```
(1) program copiaafuera(input,output);
(2)     var a : integer;
(3)     procedure inseguro(var x : integer);
(4)         begin x := 2; a := 0 end;
(5)     begin
(6)         a := 1; inseguro(a); writeln(a)
(7)     end.
```

Fig. 7.31. El resultado cambia si la llamada por referencia se cambia por copia y restauración.

real a se copia en el parámetro formal x. El valor final 2 de x se copia hacia fuera en el valor de lado izquierdo de a justo antes de que el control retorne, así que el valor final de a es 2.

Llamada por nombre

La llamada por nombre se define tradicionalmente con la *regla de copia* de ALGOL, que es:

1. El procedimiento se considera como si fuera un macro; es decir, su cuerpo se sustituye por la llamada en el autor de la llamada, y los parámetros actuales se sustituyen literalmente por los parámetros formales. Dicha sustitución literal se denomina *macroexpansión* o *expansión en línea*.

2. Los nombres locales del procedimiento llamado se distinguen de los nombres del procedimiento llamador. Se puede considerar que a cada nombre local del procedimiento llamado se le da sistemáticamente otro nombre distinto antes de realizar la macroexpansión.

3. Si es necesario, los parámetros actuales se encierran entre paréntesis para preservar su integridad.

Ejemplo 7.8. La llamada `permuta(i,a[i])` del ejemplo 7.7 se implantaría como si fuera

```
temp := i
   i := a[i]
a[i] := temp
```

Así, con la llamada por nombre, `permuta` asigna `a[i]` a `i`, como estaba previsto, pero ofrece el resultado sorprendente de asignar I_0 a `a[a[`I_0`]]` (en lugar de asignarlo a `a[`I_0`]`), donde I_0 es el valor inicial de `i`. Este fenómeno se produce porque la posición de `x` en la asignación `x:=temp` de `permuta` no se evalúa hasta ser necesario, y para entonces el valor de `i` ya ha cambiado. Aparentemente no se puede escribir una versión de `permuta` que funcione correctamente si se utiliza la llamada por nombre (véase Fleck [1976]). □

Aunque el interés de la llamada por nombre es fundamentalmente teórico, se ha propuesto la técnica conceptualmente relacionada de expansión en línea para reducir el tiempo de ejecución de un programa. Existe un cierto costo asociado al establecimiento de una activación de un procedimiento —se asigna espacio para el registro de activación, se guarda el estado de la máquina, se establecen los enlaces y después se transfiere el control—. Cuando el cuerpo de un procedimiento es pequeño, el código para las secuencias de llamada puede superar en peso al código del cuerpo del procedimiento. Por tanto, puede ser más eficiente utilizar la expansión en línea del cuerpo dentro del código del autor de la llamada, aunque el tamaño del programa aumente un poco. En el siguiente ejemplo, la expansión en línea se aplica a un procedimiento llamado por valor.

Ejemplo 7.9. Supóngase que la función `f` en la asignación

```
x := f(A) + f(B)
```

sea llamada por valor. Aquí, los parámetros actuales A y B son expresiones. Sustituir las expresiones A y B para cada caso del parámetro formal en el cuerpo de `f` conduce a la llamada por nombre; recuérdese `a[i]` del último ejemplo.

Se pueden utilizar nuevas variables temporales para obligar a que la evaluación de los parámetros actuales se realice antes de la ejecución del cuerpo del procedimiento:

```
t1 := A ;
t2 := B ;
t3 := f(t1);
t4 := f(t2);
 x := t3 + t4;
```

Ahora, la expansión en línea sustituirá todos los casos del parámetro formal por t_1 y t_2 cuando se amplíen la primera y la segunda llamadas, respectivamente[7]. □

La implantación más frecuente de la llamada por nombre consiste en pasar al procedimiento llamado subrutinas sin parámetros, que pueden evaluar el valor de lado izquierdo o de lado derecho del parámetro actual. Como cualquier procedimiento pasado como parámetro en un lenguaje que utiliza alcance léxico, estas subrutinas llevan con ellas un enlace de acceso, que apuntan al registro de activación en curso para el procedimiento que hace la llamada.

7.6 TABLAS DE SIMBOLOS

Un compilador utiliza una tabla de símbolos para llevar un registro de la información sobre el ámbito y el enlace de los nombres. Se examina la tabla de símbolos cada vez que se encuentra un nombre en el texto fuente. Si se descubre un nombre nuevo o nueva información sobre un nombre ya existente, se producen cambios en la tabla.

Un mecanismo de tabla de símbolos debe permitir añadir entradas nuevas y encontrar las entradas existentes eficientemente. Los dos mecanismos para tablas de símbolos presentadas en esta sección son listas lineales y tablas de dispersión. Cada esquema se evalúa basándose en el tiempo necesario para añadir n entradas y realizar e consultas. Una lista lineal es lo más fácil de implantar, pero su rendimiento es pobre cuando e y n se vuelven más grandes. Los esquemas de dispersión proporcionan un mayor rendimiento con un esfuerzo algo mayor de programación y gasto de espacio. Ambos mecanismos pueden adaptarse rápidamente para funcionar con la regla del anidamiento más cercano.

---

[7] Existen costos ocultos asociados con variables temporales. Pueden hacer que se asigne espacio adicional en un registro de activación. Si se inicializan las variables locales en el registro de activación, entonces las variables temporales adicionales también hacen perder tiempo.

Es útil que un compilador pueda aumentar dinámicamente la tabla de símbolos durante la compilación. Si la tabla de símbolos tiene tamaño fijo al escribir el compilador, entonces el tamaño debe ser lo suficientemente grande como para albergar cualquier programa fuente. Es muy probable que dicho tamaño sea demasiado grande para la mayoría de los programas e inadecuado para algunos.

Entradas de la tabla de símbolos

Cada entrada de la tabla de símbolos corresponde a la declaración de un nombre. El formato de las entradas no tiene que ser uniforme porque la información de un nombre depende del uso de dicho nombre. Cada entrada se puede implantar como un registro que conste de una secuencia de palabras consecutivas de memoria. Para mantener uniformes los registros de la tabla de símbolos, es conveniente guardar una parte de la información de un nombre fuera de la entrada de la tabla, almacenando en el registro sólo un apuntador a esta información.

No toda la información se introduce en la tabla de símbolos a la vez. Las palabras clave se introducen, si acaso, al inicio. El analizador léxico de la sección 3.4 busca secuencias de letras y dígitos en la tabla de símbolos para determinar si se ha encontrado una palabra clave reservada o un nombre. Con este enfoque, las palabras clave deben estar en la tabla de símbolos antes de que comience el análisis léxico. En ocasiones, si el analizador léxico reconoce las palabras clave reservadas, entonces no necesitan aparecer en la tabla de símbolos. Si el lenguaje no convierte en reservadas las palabras clave, entonces es indispensable que las palabras clave se introduzcan en la tabla de símbolos advirtiendo su posible uso como palabras clave.

La entrada misma de la tabla de símbolos puede establecerse cuando se aclara el papel de un nombre, y se llenan los valores de los atributos cuando se dispone de la información. En algunos casos, el analizador léxico puede iniciar la entrada en cuanto aparezca un nombre en los datos de entrada. A menudo, un nombre puede indicar varios objetos distintos, quizás incluso en el mismo bloque o procedimiento. Por ejemplo, las declaraciones en C

```
int    x;
struct x { float y, z; };                    (7.1)
```

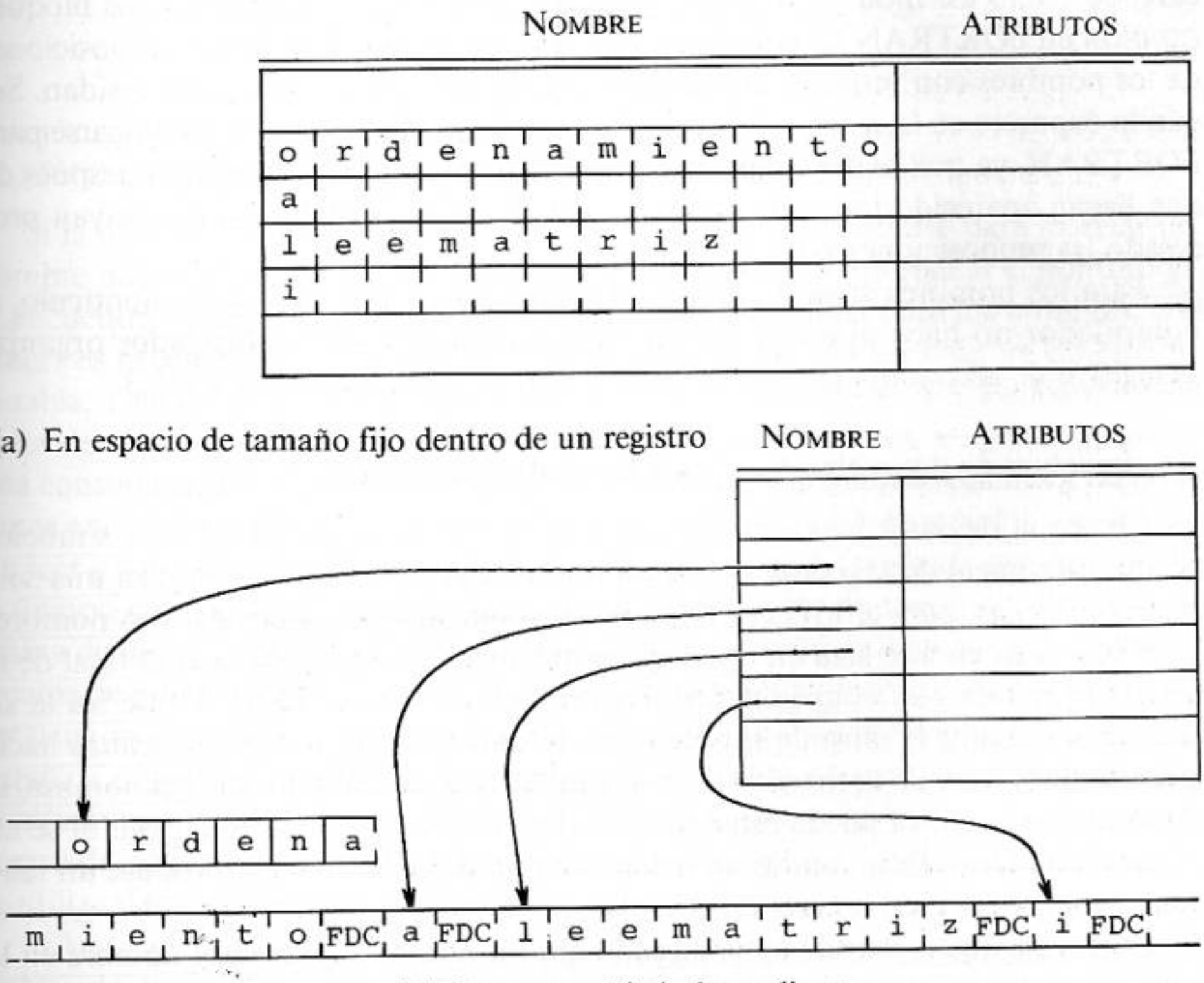
utilizan `x` como entero y como etiqueta de una estructura con dos campos. En dichos casos, el analizador léxico sólo puede devolver al analizador sintáctico el nombre solo (o un apuntador al lexema que forma dicho nombre), en lugar de un apuntador a la entrada en la tabla de símbolos. Se crea el registro en la tabla de símbolos cuando se descubre el papel sintáctico que desempeña este nombre. Para las declaraciones de (7.1), se crearían dos entradas en la tabla de símbolos para `x`; una con `x` como entero y otra como estructura.

Los atributos de un nombre se introducen en respuesta a las declaraciones, que pueden ser implícitas. Las etiquetas son a menudo identificadores seguidos de dos puntos, así que una acción asociada con el reconocimiento de dicho identificador puede ser introducir este hecho en la tabla de símbolos. Asimismo, la sintaxis de las declaraciones de procedimientos especifican que algunos identificadores son parámetros formales.

Caracteres dentro de un nombre

Como se vio en el capítulo 3, existe una distinción entre el componente léxico **id** para un identificador o nombre, el lexema formado por la cadena de caracteres que componen el nombre, y los atributos del nombre. Las cadenas de caracteres pueden ser difíciles de manejar, así que los compiladores utilizan a menudo alguna representación de longitud fija del nombre en lugar del lexema. El lexema es necesario cuando se establece por primera vez una entrada a la tabla de símbolos y cuando se busca un lexema encontrado en los datos de entrada para determinar si es un nombre que ya ha aparecido. Una representación habitual de un nombre es un apuntador a una entrada en la tabla de símbolos para él.

Si hay un límite superior pequeño para la longitud de un nombre, entonces los caracteres del nombre pueden almacenarse en la entrada de la tabla de símbolos, como se muestra en la figura 7.32(a). Si no hay límite para la longitud de un nombre, o si rara vez se alcanza el límite, se puede utilizar el esquema indirecto de la figura 7.32(b). En lugar de asignar la cantidad máxima de espacio para guardar un lexema en cada entrada de la tabla, se puede utilizar el espacio más eficientemente si sólo hay espacio para un apuntador en cada entrada de la tabla. En el registro para un nombre, se coloca un apuntador a una matriz independiente de caracteres (la



(a) En espacio de tamaño fijo dentro de un registro

(b) En una matriz independiente

Fig. 7.32. Almacenamiento de los caracteres de un nombre.

tabla de cadenas) que da la posición del primer carácter del lexema. El esquema indirecto de la figura 7.32(b) permite que el tamaño del campo del nombre de la entrada misma de la tabla de símbolos permanezca constante.

El lexema completo que constituye un nombre debe almacenarse para garantizar que todos los usos del mismo nombre se puedan asociar con el mismo registro en la tabla de símbolos. Sin embargo, se debe distinguir entre casos del mismo lexema que estén dentro de los ámbitos de declaraciones distintas.

Información sobre la asignación de memoria

Se mantiene en la tabla de símbolos la información acerca de las posiciones de memoria que se ligarán a nombres durante la ejecución. Considérense primero los nombres con posiciones de memoria estática. Si el código objeto es lenguaje ensamblador, el ensamblador puede encargarse de las posiciones de memoria para los distintos nombres. Basta con examinar la tabla de símbolos, después de generar código ensamblador para el programa, y generar definiciones de datos en lenguaje ensamblador para añadirlos al programa en lenguaje ensamblador para cada nombre.

Sin embargo, si el compilador genera código de máquina, entonces se debe indagar la posición de cada objeto de datos relativa a un origen fijo, como el principio de un registro de activación. La misma observación sirve para un bloque de datos cargado como un módulo independiente del programa. Por ejemplo, los bloques COMMON en FORTRAN se cargan por separado y se deben determinar las posiciones de los nombres con respecto al principio del bloque COMMON en el que residan. Según lo expuesto en la sección 7.9, el enfoque de la sección 7.3 debe modificarse para FORTRAN, ya que se deben asignar desplazamientos para los nombres después de que hayan aparecido todas las declaraciones de un procedimiento y se hayan procesado las proposiciones EQUIVALENCE.

Para los nombres cuya memoria esté asignada en una pila o un montículo, el compilador no hace ninguna asignación de memoria —el compilador organiza el registro de activación para cada procedimiento, como en la sección 7.3.

La estructura de datos tipo lista para las tablas de símbolos

La estructura de datos más sencilla y fácil de implantar para una tabla de símbolos es una lista lineal de registros, que se muestra en la figura 7.33. Se utiliza una sola matriz, o varias, para almacenar nombres y su información asociada. Los nombres nuevos se añaden a la lista en el orden en que aparecen. La posición del final de la matriz se marca con el apuntador *disponible*, que apunta hacia donde irá la siguiente entrada de la tabla de símbolos. La búsqueda de un nombre se realiza hacia atrás desde el final de la matriz hasta el comienzo. Cuando se localiza el nombre, la información asociada puede estar en las palabras situadas a continuación. Si se alcanza el comienzo de la matriz sin haber encontrado el nombre, se produce un fallo —un nombre previsto no está en la tabla.

Obsérvese que construir una entrada para un nombre y buscar el nombre en la tabla de símbolos son operaciones independientes —es posible realizar sólo una de ellas—. En un lenguaje estructurado por bloques, un caso de un nombre está dentro del ámbito de la declaración de anidamiento más cercana del nombre. Se puede im-

plantar esta regla de ámbito utilizando la estructura de datos tipo lista, construyendo una entrada nueva para un nombre cada vez que se declara. Se construye una entrada nueva en las palabras inmediatamente después del apuntador *disponible*; este apuntador aumenta por el tamaño del registro en la tabla de símbolos. Como las entradas se insertan en orden, comenzando por el principio de la matriz, aparecen en el orden en que se crearon. Buscando desde *disponible* hacia el principio de la matriz, se encuentra la entrada más recientemente creada.

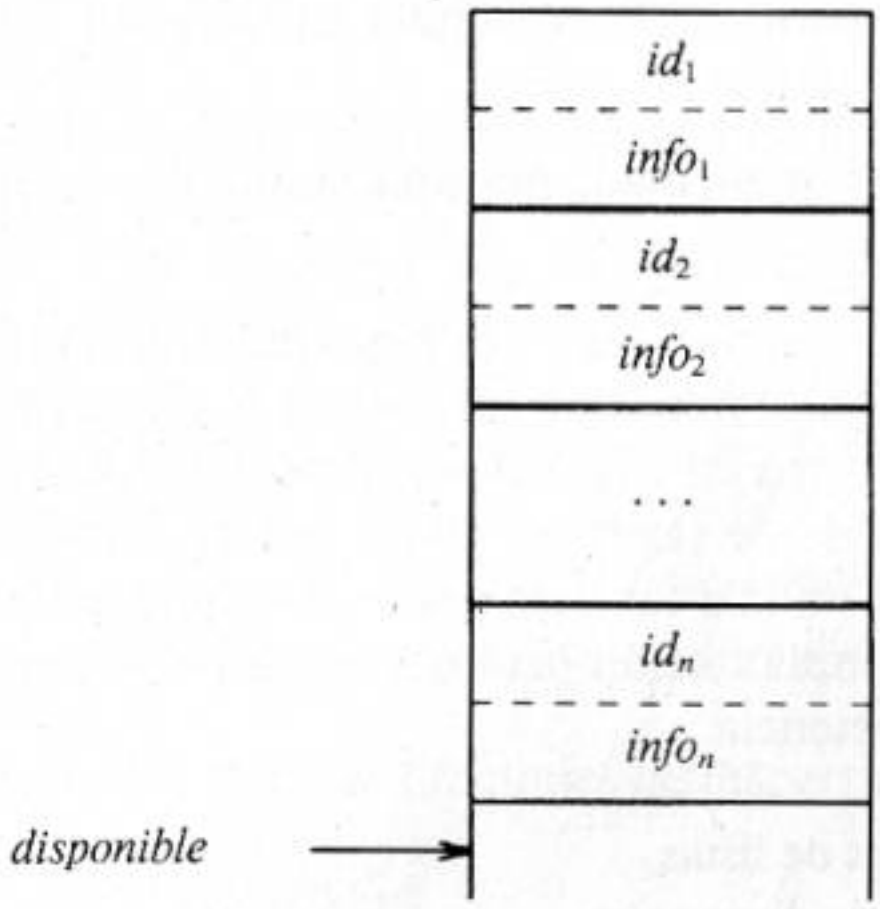


Fig. 7.33. Una lista lineal de registros.

Si la tabla de símbolos contiene n nombres, el trabajo necesario para insertar un nombre nuevo es constante si se realiza la inserción sin comprobar si el nombre ya se encuentra en la tabla. Si no se permiten entradas múltiples para los nombres, entonces es necesario recorrer toda la tabla para descubrir que un nombre no está en la tabla, realizando un trabajo proporcional a n en el proceso. Para encontrar los datos de un nombre, como promedio se examinan $n/2$ nombres, así que el costo de una consulta también es proporcional a n. Por tanto, como las inserciones y las consultas emplean un tiempo proporcional a n, el trabajo total para insertar n nombres y hacer e consultas es a lo sumo $cn\,(n+e)$, donde c es una constante que representa el tiempo necesario para realizar unas pocas operaciones de máquina. En un programa de tamaño medio, se puede tener $n=100$ y $e=1000$, así que se utilizan varios cientos de miles de operaciones de máquina para el mantenimiento de la tabla. Esto no tiene importancia porque se trata de menos de un segundo de tiempo. Sin embargo, si n y e se multiplican por 10, el costo se multiplica por 100, y el tiempo de mantenimiento se vuelve prohibitivo. La estructuración proporciona datos valiosos acerca de dónde emplea su tiempo un compilador y puede utilizarse para decidir si se utiliza demasiado tiempo en buscar en listas lineales.

Tablas de dispersión

En muchos compiladores se han implantado variaciones de la técnica de búsqueda conocida como dispersión. En este caso se considera una variante bastante senci-

lla conocida como *dispersión abierta*, donde "abierta" se refiere a la propiedad de que no hay límite al número de entradas que pueden construirse en la tabla. Incluso este esquema permite la realización de e consultas sobre n nombres en un tiempo proporcional a $n(n+e)/m$, para cualquier constante m elegida. Como m se puede aumentar todo lo que se quiera, hasta n, este método es en general más eficiente que las listas lineales y es el método elegido para las tablas de símbolos en la mayoría de las situaciones. Como es normal, el espacio ocupado por la estructura de datos aumenta con m, así que atañe un arreglo tiempo-espacio.

El esquema de dispersión básico se ilustra en la figura 7.34. Hay dos partes relativas a la estructura de datos:

1. Una *tabla de dispersión* formada por una matriz fija de m apuntadores a entradas de tabla.
2. Entradas de la tabla organizadas en m listas enlazadas independientes llamadas *cubetas* (algunas cubetas pueden estar vacías). Cada registro en la tabla de símbolos aparece en sólo una de estas listas. La memoria para estos registros se puede obtener de una matriz de registros, como se estudiará en la siguiente sección. También se pueden utilizar las ventajas para la asignación dinámica de memoria del lenguaje de implantación para obtener espacio para los registros, aunque se pierde algo de eficiencia.

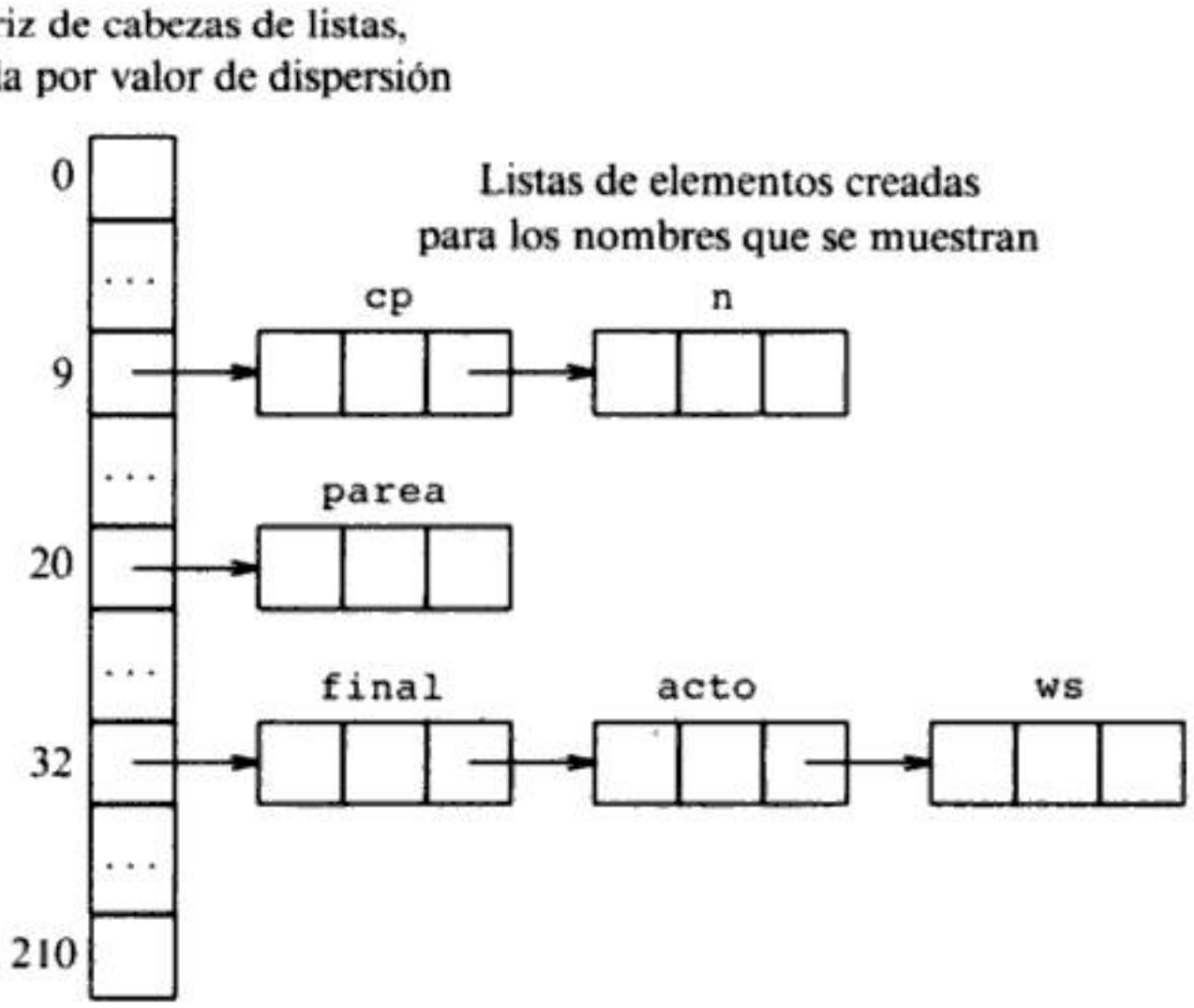


Fig. 7.34. Una tabla de dispersión de tamaño 211.

Para determinar si existe o no una entrada para la cadena s en la tabla de símbolos, se aplica una *función de dispersión* h a s, tal que $h(s)$ devuelve un entero entre 0 y $m-1$. Si s está en la tabla de símbolos, entonces está en la lista numerada

con $h(s)$. Si s todavía no está en la tabla, se introduce creando un registro para s enlazado al principio de la lista numerada con $h(s)$.

En la práctica, la lista promedio consta de n/m registros si hay n nombres en una tabla de tamaño m. Si se elige m de modo que n/m esté limitado por una constante pequeña, por ejemplo 2, el tiempo para acceder a una entrada de la tabla es prácticamente constante. El espacio empleado por la tabla de símbolos consta de m palabras para la tabla de dispersión y cn palabras para las entradas de la tabla, donde c es el número de palabras por entrada de la tabla. Por tanto, el espacio para la tabla de dispersión depende sólo de m, y el espacio para las entradas de la tabla depende sólo del número de entradas.

La elección de m depende de la aplicación que vaya a darse a una tabla de símbolos. Elegir a m como unos cientos convertiría la búsqueda en la tabla una parte insignificante del tiempo total empleado por un compilador, incluso para programas de tamaño medio. Sin embargo, cuando los datos de entrada a un compilador pueden ser generados por otro programa, el número de nombres puede sobrepasar con mucho el de la mayoría de los programas del mismo tamaño generados por una persona, y serían preferibles tamaños mayores para la tabla.

Se ha prestado mucho interés a la cuestión de cómo diseñar una función de dispersión fácil de calcular para cadenas de caracteres y que distribuya cadenas uniformemente entre las m listas.

Un enfoque adecuado para calcular funciones de dispersión es proceder de la siguiente manera:

1. Determínese un entero positivo h a partir de los caracteres $c_1, c_2, \ldots, c_k$ de la cadena s. Convertir caracteres simples en enteros viene apoyado generalmente por el lenguaje de implantación. Pascal proporciona la función *ord* para este propósito; C convierte automáticamente un carácter en un entero si se realiza en él una operación aritmética.
2. Conviértase el entero h determinado anteriormente en el número de una lista, es decir, un entero entre 0 y $m-1$. Dividir simplemente por m y tomar el resto es una idea razonable. Tomar el resto funciona mejor si m es un número primo, de ahí la elección de 211 en lugar de 200 en la figura 7.34.

Se engaña menos fácilmente a las funciones de dispersión que examinan todos los caracteres de una cadena que, por ejemplo, a las funciones que examinan sólo unos cuantos caracteres en los extremos o en la mitad de una cadena. Recuérdese que los datos de entrada para un compilador pudieron ser creados por un programa y por tanto pueden tener una forma estilizada para evitar conflictos con los nombres que pudieran utilizar una persona u otro programa. Las personas tienden asimismo a "agrupar" nombres, eligiendo `valor`, `valornuevo`, `valor1`, etcétera.

Una técnica sencilla para calcular h es sumar los valores enteros de los caracteres de una cadena. Una idea mejor todavía es multiplicar el valor anterior de h por una constante α antes de sumarle el nuevo carácter. Es decir, tomar $h_0 = 0$, $h_i = \alpha h_{i-1} + c_i$, para $1 \le i \le k$, y dejar que $h = h_k$, donde k sea la longitud de la cadena. (Recuérdese que el valor de dispersión que da el número de lista es de h **mod** m.) Sumando los caracteres es el caso $\alpha = 1$. Una estrategia similar es efectuar la operación "o exclusivo" de c_i con αh_{i-1}, en lugar de sumar.

Para enteros de 32 bits, si se toma $\alpha = 65599$, un número primo cercano a 2^{16}, entonces pronto se producirá un desbordamiento durante el cálculo de αh_{i-1}. Como α es un número primo, es mejor no tener en cuenta los desbordamientos y conservar sólo los 32 bits de orden menor.

En una serie de experimentos, la función *dispersión pjw* de la figura 7.35, tomada del compilador de C de P. J. Weinberger, se comportó consistentemente bien con todos los tamaños de tabla analizados (véase Fig. 7.36). Los tamaños incluían los primeros números primos mayores de 100, 200, . . , 1500. En segundo lugar estaba la función que calculó h multiplicando el valor anterior por 65599, sin tener en cuenta los desbordamientos, y sumando el siguiente carácter. La función *dispersión pjw* se calcula comenzando por $h = 0$. Para cada carácter c, se trasladan los bits de h cuatro posiciones a la izquierda y se suma c. Si cualquiera de los cuatro bits de orden mayor de h es 1, se desplazan los cuatro bits 24 posiciones a la derecha. Se les hace la operación de "o exclusivo" en h, y se reasigna a 0 cualquiera de los cuatro bits de orden mayor que fuera 1.

```
(1)  #define PRIMO 211
(2)  #define FDC '\0'
(3)  int dispersión_pjw(c)
(4)  char   *c;
(5)  {
(6)      char *p;
(7)      unsigned h = 0, g;
(8)      for ( p = c; *p != FDC; p = p+1 ) {
(9)          h = (h << 4) + (*p);
(10)         if (g = h&0xf0000000) {
(11)             h = h ^ (g >> 24);
(12)             h = h ^ g;
(13)         }
(14)     }
(15)     return h % PRIMO;
(16) }
```

Fig. 7.35. La función de dispersión *dispersión pjw*, escrita en C.

Ejemplo 7.10. Para obtener mejores resultados, se deben tener en cuenta el tamaño de la tabla de dispersión y los datos de entrada previstos. Por ejemplo, es preferible que los valores de dispersión para los nombres de mayor frecuencia en un lenguaje sean distintos. Si las palabras clave se introducen en la tabla de símbolos, entonces es probable que las palabras clave sean de los nombres que se producen con mayor frecuencia, aunque en un muestreo de programas en C, el nombre `i` aparecía en una proporción de tres a uno con respecto a `while`.

Una manera de poner a prueba una función de dispersión es observar el número de cadenas que forman parte de la lista. Dado un archivo F formado por n cadenas, supóngase que b_j cadenas forman parte de la lista j, para $0 \leq j \leq m-1$. Una medida

de lo uniformemente que se distribuyen las cadenas a lo largo de las listas se obtiene calculando

$$\sum_{j=0}^{m-1} b_j\,(b_j+1)\,/\,2 \tag{7.2}$$

La justificación intuitiva para este término es que hay que examinar 1 elemento de la lista para encontrar la primera entrada en la lista j, 2 para encontrar la segunda, y así sucesivamente hasta b_j para encontrar la última entrada. La suma de 1, 2, . . . , b_j es $b_j\,(b_j+1)\,/\,2$.

Según el ejercicio 7.14, el valor de (7.2) para una función de dispersión que distribuya cadenas aleatoriamente a lo largo de las cubetas es

$$(n\,/\,2m)\,(n\,+\,2m\,-\,1) \tag{7.3}$$

En la figura 7.36 se muestra la razón de los términos (7.2) y (7.3) para varias funciones de dispersión aplicadas a nueve archivos. Los archivos son:

1. Los 50 nombres y palabras clave más frecuentes en una muestra de programas en C.
2. Igual que (1), pero con los 100 nombres y palabras clave más frecuentes.
3. Igual que (1), pero con los 500 nombres y palabras clave más frecuentes.

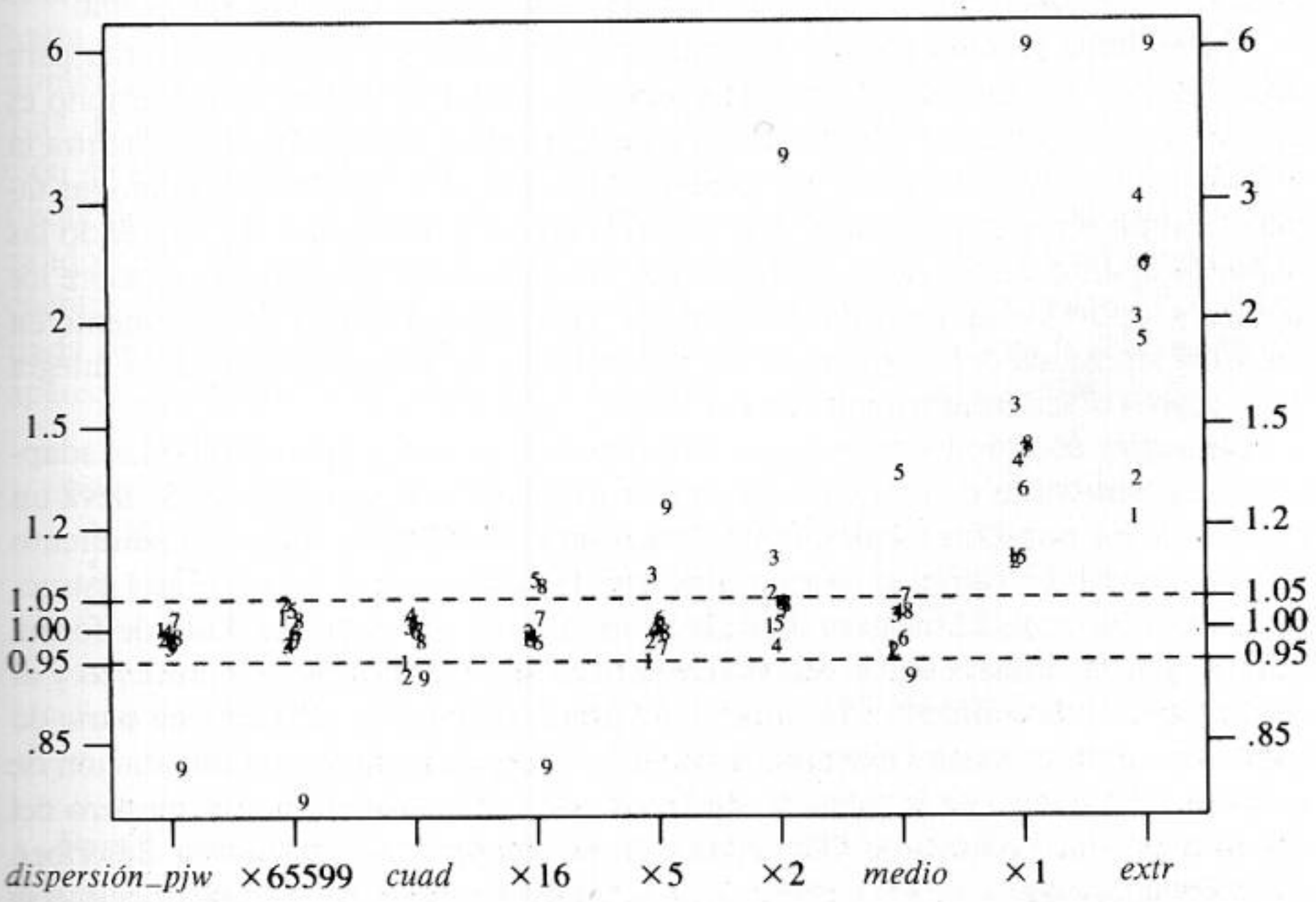


Fig. 7.36: Rendimiento relativo de varias funciones de dispersión para una tabla de tamaño 211.

4. 952 nombres externos en el núcleo del sistema operativo UNIX.
5. 627 nombres de un programa en C generados por C++ (Stroustrup [1986]).
6. 915 cadenas de caracteres generados aleatoriamente.
7. 614 palabras de la sección 3.1 de este libro (en su versión en inglés de junio de 1987).
8. 1201 palabras en inglés con `xxx` añadido como prefijo y sufijo.
9. Los 300 nombres `v100`, `v101`, . . . , `v399`.

La función *dispersión_pjw* es como la de la figura 7.35. Las funciones con nombre $\times\alpha$, donde α es una constante entera, calculan h **mod** m, donde h se obtiene iterativamente comenzando por 0, multiplicando el valor anterior por α, y sumando el siguiente carácter. La función *medio* forma h a partir de los cuatro caracteres del medio de la cadena, en tanto que *extr* suma los primeros y los últimos tres caracteres con la longitud para formar h. Por último, *cuad* agrupa cada cuatro caracteres consecutivos en un entero y después suma los enteros. □

Representación de la información sobre el ámbito

Las entradas de la tabla de símbolos son para declaraciones de nombres. Cuando se busca el caso de un nombre del texto fuente en la tabla de símbolos, se debe devolver la entrada correspondiente a la declaración adecuada de dicho nombre. Las reglas de ámbito del lenguaje fuente determinan qué declaración es la apropiada.

Un enfoque sencillo consiste en mantener una tabla de símbolos distinta para cada ámbito. En realidad, la tabla de símbolos para un procedimiento o ámbito es el equivalente durante la compilación de un registro de activación. Se encuentra la información para los nombres no locales de un procedimiento examinando las tablas de símbolos correspondientes a los procedimientos abarcadores siguiendo las reglas de ámbito del lenguaje. Asimismo, se puede asociar la información sobre los nombres locales de un procedimiento al nodo correspondiente al procedimiento en un árbol sintáctico del programa. Con este enfoque la tabla de símbolos se integra en la representación inmediata de la entrada.

Las reglas de ámbito de anidamiento más cercano se pueden implantar adaptando las estructuras de datos anteriormente estudiadas en esta sección. Se lleva un registro de los nombres locales de un procedimiento dando a cada procedimiento un único nombre. También deben numerarse los bloques si el lenguaje está estructurado por bloques. El número de cada procedimiento se puede calcular de forma dirigida por la sintaxis según reglas semánticas que reconozcan el comienzo y el final de cada procedimiento. El número de procedimiento se convierte en parte de todos los nombres locales declarados en dicho procedimiento; la representación de nombre local dentro de la tabla de símbolos es un par formado por el número del nombre y del procedimiento. (En algunas disposiciones, como las que se describen más adelante, no es necesario que aparezca el número del procedimiento ya que se puede deducir de la posición del registro dentro de la tabla de símbolos.)

Cuando se busca un nombre recién examinado, se produce concordancia sólo si los caracteres del nombre concuerdan con un carácter de entrada por carácter, y el número asociado en la entrada de la tabla de símbolos es el número del procedi-

miento que se está procesando. Las reglas de ámbito del anidamiento más cercano pueden implantarse según las siguientes operaciones con un nombre:

busca:	encuentra la entrada recién creada
inserta:	construye una entrada nueva
borra:	elimina la entrada recién creada

Deben preservarse las entradas "borradas"; sólo se eliminan de la tabla de símbolos activa. En un compilador de una pasada, la información de la tabla de símbolos sobre un ámbito que conste, por ejemplo, de un cuerpo de un procedimiento, no se necesita para la compilación después de que se haya procesado el cuerpo del procedimiento. Sin embargo, puede necesitarse durante la ejecución, sobre todo si se implanta un sistema de diagnóstico para la ejecución. En este caso, se debe añadir la información de la tabla de símbolos al código generado para ser utilizada por el editor de enlaces o por el sistema de diagnóstico para la ejecución. Véase también el tratamiento de los nombres de los campos dentro de los registros en las secciones 8.2 y 8.3.

Cada una de las estructuras de datos presentadas en esta sección —listas y entradas de dispersión— se puede mantener para apoyar las operaciones anteriores.

Cuando se ha descrito antes una lista lineal formada por una matriz de registros, se mencionó cómo se puede implantar *busca* insertando las entradas por un extremo de manera que el orden de las entradas dentro de la matriz sea el mismo que el orden de inserción de las entradas. Un examen que comience por el final y siga hasta el principio de la matriz encuentra la entrada más reciente creada para un nombre. La situación es similar en una lista enlazada, como se muestra en la figura 7.37. El apuntador *frente* apunta a la entrada más reciente de la lista. La implantación de *inserta* emplea un tiempo constante porque se coloca una entrada nueva al frente de la lista. La implantación de *busca* se realiza examinando la lista comenzando en la entrada apuntada por *frente* y siguiendo los enlaces hasta que se encuentre el nombre deseado o se alcance el final de la lista. En la figura 7.37, la entrada correspondiente a a declarada en el bloque B_2, anidado dentro del bloque B_0, aparece más cerca del principio de la lista que la entrada para a declarada en B_0.

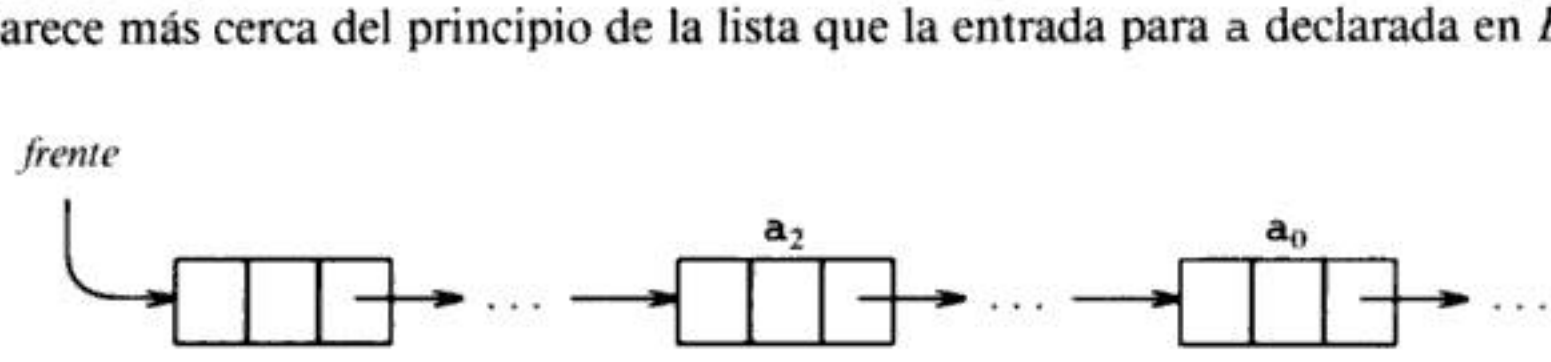


Fig. 7.37. La entrada más reciente de a está cerca del frente.

Para la operación *borra*, obsérvese que las entradas para las declaraciones del procedimiento más profundamente anidado aparecen más cerca del principio de la lista. Por tanto, no es necesario conservar el número de procedimiento con cada entrada —si se anota la primera entrada de cada procedimiento, entonces todas las entradas hasta la primera se pueden borrar de la tabla de símbolos activa cuando se termine de procesar el ámbito de ese procedimiento.

Una tabla de dispersión consta en m listas a las que se accede por medio de una matriz. Como un nombre siempre se direcciona hacia la misma lista por medio de la dispersión, se mantienen listas individuales como en la figura 7.37. Sin embargo, para implantar la operación *borra* no habría que examinar toda la tabla de dispersión buscando las listas con entradas que deban borrarse. Se puede utilizar el siguiente enfoque. Supóngse que cada entrada tiene dos enlaces:

1. un enlace de dispersión que enlaza la entrada con otras entradas cuyos nombres producen el mismo valor de dispersión, y
2. un enlace de ámbito que enlaza todas las entradas que estén dentro del mismo alcance.

Si no se modifica el enlace de ámbito cuando se borra una entrada de la tabla de dispersión, entonces la cadena formada por los enlaces de ámbito constituirá una tabla de símbolos independiente (inactiva) para el ámbito en cuestión.

El borrado de entradas de la tabla de dispersión se debe efectuar con cuidado, porque el borrado de una entrada afecta a la entrada previa de su lista. Recuérdese que se borra la i-ésima entrada apuntando la $(i-1)$-ésima entrada a la $(i+1)$-ésima. Por tanto, utilizar únicamente los enlaces de ámbito para encontrar la i-ésima entrada no es suficiente. La $(i-1)$-ésima entrada se puede encontrar si los enlaces de dispersión forman una lista enlazada circular, en la que la última entrada vuelve a apuntar a la primera. También se puede utilizar una pila para llevar un registro de las listas que contengan entradas que deban borrarse. Cuando se analiza un procedimiento nuevo se coloca un marcador en la pila. Por encima del marcador están los números de las listas que contienen las entradas correspondientes a los nombres declarados en dicho procedimiento. Cuando se termina de procesar el procedimiento, los números de listas pueden sacarse de la pila hasta alcanzar el marcador. En el ejercicio 7.11 se estudiará otro esquema.

7.7 INSTRUMENTOS DE LOS LENGUAJES PARA LA ASIGNACION DINAMICA DE LA MEMORIA

En esta sección, se describen brevemente las facilidades que proporcionan algunos lenguajes para la asignación dinámica de memoria para los datos, durante el control del programa. La memoria para dichos datos se toma generalmente de un montículo. Los datos asignados se retienen a menudo hasta que se desasignan explícitamente. La asignación misma puede ser *explícita* o *implícita*. En Pascal, por ejemplo, la asignación explícita se realiza utilizando el procedimiento estándar `new`. La ejecución de `new(p)` asigna memoria para el tipo de objeto apuntado por `p`, y `p` queda apuntando al objeto recién asignado. La desasignación se realiza llamando a `dispose` en la mayoría de las implantaciones de Pascal.

La asignación implícita se produce cuando la evaluación de una expresión da como resultado la obtención de memoria para guardar los valores de la expresión. LISP, por ejemplo, asigna una celda en una lista cuando se utiliza `cons`; automáticamente se reclaman las celdas que ya no se pueden alcanzar. SNOBOL permite que

la longitud de una cadena varíe durante la ejecución y administra el espacio necesario para guardar la cadena de un montículo.

Ejemplo 7.11. El programa en Pascal de la figura 7.38 construye la lista enlazada de la figura 7.39 e imprime los enteros de las celdas; su resultado es

```
76      3
 4    • 2
 7      1
```

Cuando la ejecución del programa comienza en la línea 15, la dirección de memoria para el apuntador `cabeza` está en el registro de activación correspondiente al programa completo. Cada vez que el control llega a

```
(11)  new(p); p↑.llave := k; p↑.info := i;
```

la llamada `new(p)` da como resultado la asignación de una celda en alguna parte dentro del montículo; `p↑` se refiere a dicha celda en las asignaciones de la línea 11.

Obsérvese por el resultado del programa que las celdas asignadas son accesibles cuando el control regresa al programa principal desde `inserta`. En otras palabras, se retienen las celdas que se asignan utilizando `new` durante una activación de `inserta` cuando el control regresa de la activación al programa principal. □

```
(1)  program tabla(input, output);
(2)  type enlace = ↑ nodo;
(3)      nodo = record
(4)                 llave, info : integer;
(5)                 sigte : enlace
(6)      end;
(7)  var cabeza : enlace;
(8)  procedure inserta(k, i : integer);
(9)      var p : enlace;
(10)     begin
(11)         new(p); p↑.llave := k; p↑.info := i;
(12)         p↑.sigte := cabeza; cabeza := p
(13)     end;
(14) begin
(15)     cabeza := nil;
(16)     inserta(7,1); inserta(4,2); inserta(76,3);
(17)     writeln(cabeza↑.llave, cabeza↑.info);
(18)     writeln(cabeza↑.sigte↑.llave, cabeza↑.sigte↑.info);
(19)     writeln(cabeza↑.sigte↑.sigte↑.llave,
             cabeza↑.sigte↑.sigte↑.info)
(20) end.
```

Fig. 7.38. Asignación dinámica de nodos utilizando `new` en Pascal.

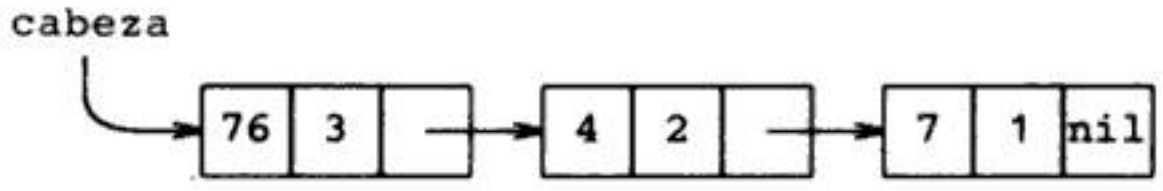


Fig. 7.39. Lista enlazada construida por el programa de la figura 7.38.

Basura

La memoria asignada dinámicamente puede volverse inaccesible. La memoria que un programa asigna, pero a la que no puede referirse, se denomina *basura*. En la figura 7.38, supóngase que se asigna `nil` a `cabeza↑.siguiente` entre las líneas 16 y 17:

```
(16)   inserta(7,1); inserta(4,2); inserta (76,3);
       cabeza↑.siguiente := nil;
(17)   writeln(cabeza↑.llave, cabeza↑.info);
```

La celda situada en el extremo izquierdo de la figura 7.39 contiene ahora un apuntador a `nil` en vez de un apuntador a la celda situada en el medio. Cuando se pierde el apuntador a la celda del centro, ésta y la celda de la derecha se convierten en basura.

LISP realiza *recogida de basura*, un proceso que se estudiará en la siguiente sección y que reclama la memoria inaccesible. Pascal y C no tienen recogida de basura, dejando que el programador desasigne explícitamente la memoria que ya no sirve. En estos lenguajes, se puede reutilizar la memoria desasignada, pero la basura permanece hasta que finaliza el programa.

Referencias suspendidas

Puede surgir una complicación adicional con la desasignación explícita. Puede producirse una referencia suspendida. Como ya se mencionó en la sección 7.3, cuando se hace referencia a memoria desasignada se produce una referencia suspendida. Por ejemplo, considérese el resultado de la ejecución de `dispose(cabeza↑.siguiente)` entre las líneas 16 y 17 de la figura 7.38:

```
(16)   inserta(7,1); inserta(4,2); inserta(76,3);
       dispose(cabeza↑.siguiente);
(17)   writeln(cabeza↑.llave, cabeza↑.info);
```

La llamada a `dispose` desasigna la celda que sigue a la apuntada por `cabeza`, como se muestra en la figura 7.40. Sin embargo, no se ha modificado `cabeza↑.siguiente`, de modo que es un apuntador desactivado que hace referencia a memoria desasignada.

Las referencias suspendidas y la basura son conceptos relacionados; las referencias suspendidas ocurren si se produce la desasignación antes de la última referencia, mientras que existe basura si se produce la última referencia antes de la desasignación.

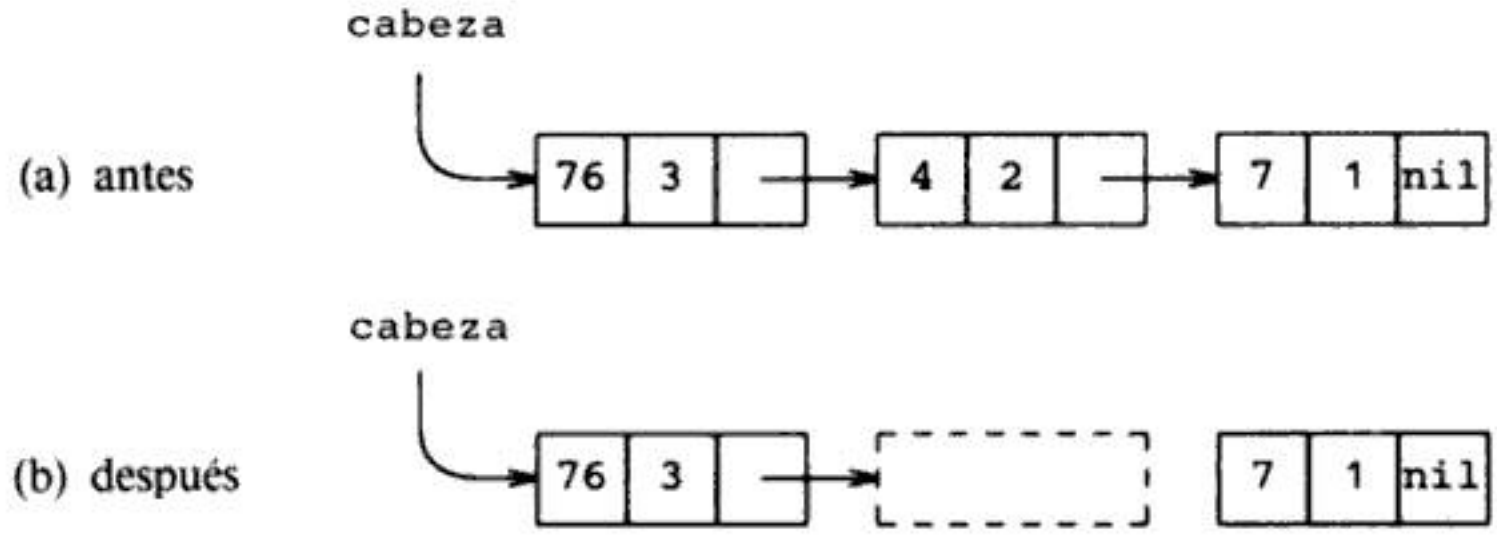


Fig. 7.40. Creación de referencias suspendidas y de basura.

7.8 TECNICAS PARA LA ASIGNACION DINAMICA DE LA MEMORIA

Las técnicas necesarias para implantar la asignación dinámica de la memoria dependen de cómo se desasigne la memoria. Si la desasignación es implícita, entonces el paquete de apoyo para la ejecución es el encargado de determinar cuándo un bloque de memoria ya no es necesario. Si la desasignación se realiza explícitamente por el programador, el compilador tiene menos trabajo. Se considera a continuación la desasignación explícita.

Asignación explícita de bloques de tamaño fijo

La forma más sencilla de asignación dinámica se realiza con bloques de tamaño fijo. Enlazando los bloques en una lista, como en la figura 7.41, se puede hacer rápidamente la asignación y la desasignación con poca o nula pérdida de memoria.

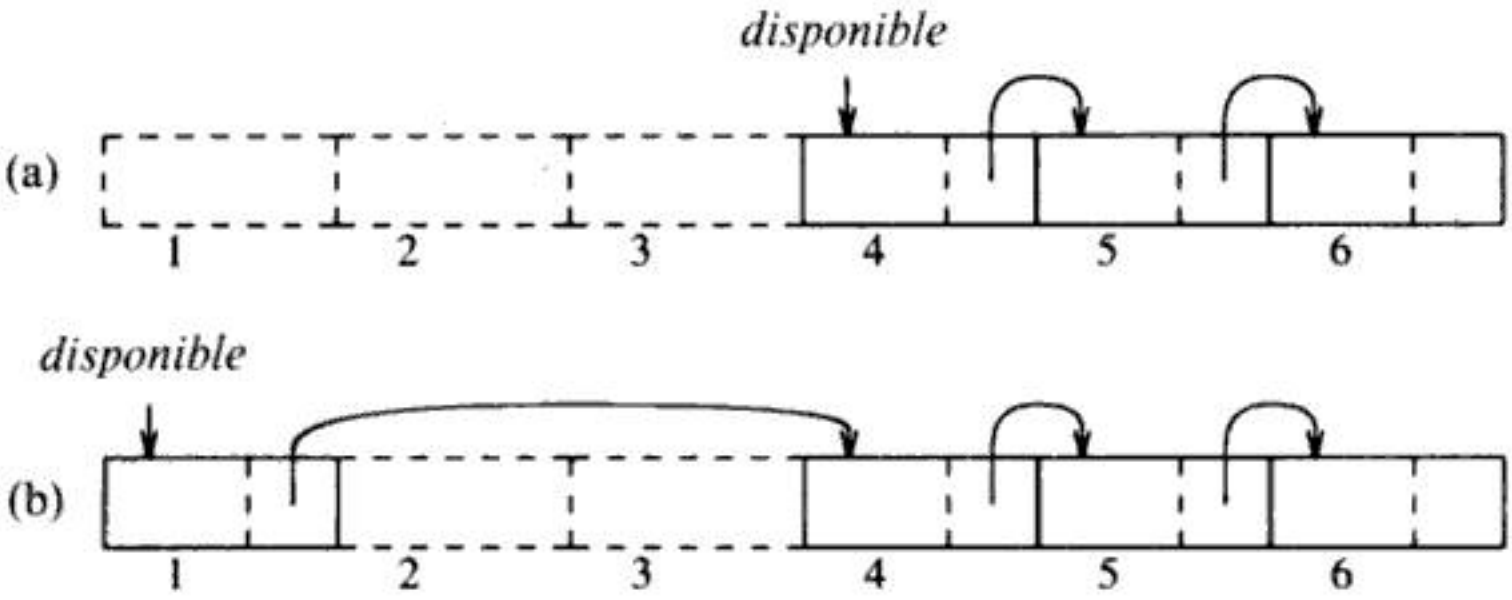


Fig. 7.41. Un bloque desasignado se añade a la lista de bloques disponibles.

Supóngase que los bloques deben obtenerse de un área contigua de memoria. La inicialización del área se realiza utilizando una porción de cada bloque para un enlace al siguiente bloque. El apuntador *disponible* apunta al primer bloque. La asignación consiste en sacar un bloque de la lista y la desasignación en devolver el bloque a la lista.

Las rutinas del compilador que administran los bloques no necesitan conocer el tipo de objeto que guardará en el bloque el programa del usuario. Se puede considerar cada bloque como un registro variable y las rutinas del compilador consideran el bloque como si estuviera compuesto por un enlace al siguiente bloque y el programa de usuario considera el bloque como algún otro tipo. Por tanto, no se pierde espacio porque el programa de usuario puede utilizar el bloque completo para sus propios propósitos. Cuando se devuelve el bloque, entonces las rutinas del compilador utilizan parte del espacio del bloque mismo para enlazarlo en la lista de bloques disponibles, como se muestra en la figura 7.41.

Asignación explícita de bloques de tamaño variable

Cuando se asignan y desasignan bloques, la memoria se puede *fragmentar*; es decir, el montículo puede estar formado por bloques alternos libres y en uso, como en la figura 7.42.

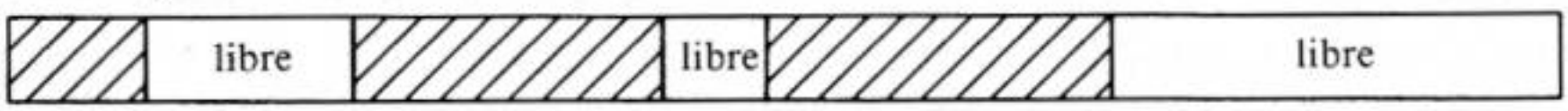


Fig. 7.42. Bloques libres y ocupados en un montículo.

Puede producirse la situación que se muestra en la figura 7.42 si un programa asigna cinco bloques y después desasigna el segundo y el cuarto, por ejemplo. La fragmentación no trae consecuencias si los bloques son de tamaño fijo, pero si son de tamaño variable, una situación como la de la figura 7.42 supone un problema, porque no se puede asignar un bloque mayor que cualquiera de los bloques libres, aunque el espacio esté en principio disponible.

Un método para asignar bloques de tamaño variable es el *método del primer ajuste*. Cuando se asigna un bloque de tamaño s, se busca el primer bloque libre que sea de tamaño $f \geq s$. Entonces este bloque se subdivide en un bloque utilizado de tamaño s, y un bloque libre de tamaño $f-s$. Obsérvese que la asignación incurre en una pérdida de tiempo porque se debe buscar un bloque libre lo suficientemente grande.

Cuando se desasigna un bloque, se comprueba que sea contiguo a un bloque libre. Si es posible, el bloque desasignado se combina con el bloque libre contiguo a él para crear un bloque libre más grande. Combinar bloques libres adyacentes en un bloque libre mayor evita que se produzcan más fragmentaciones. Hay varios detalles sutiles en cuanto a la forma de asignar, desasignar y mantener los bloques en una lista o listas disponibles. También hay algunos compromisos entre tiempo, espacio y disponibilidad de bloques grandes. El lector puede remitirse a Knuth [1973a] o a Aho, Hopcroft y Ullman [1983] para un mayor estudio de estos aspectos.

Desasignación implícita

La desasignación implícita exige cooperación entre el programa de usuario y el paquete para la ejecución, porque este último necesita saber cuándo ha dejado de funcionar un bloque de memoria. Esta cooperación se implanta fijando el formato de

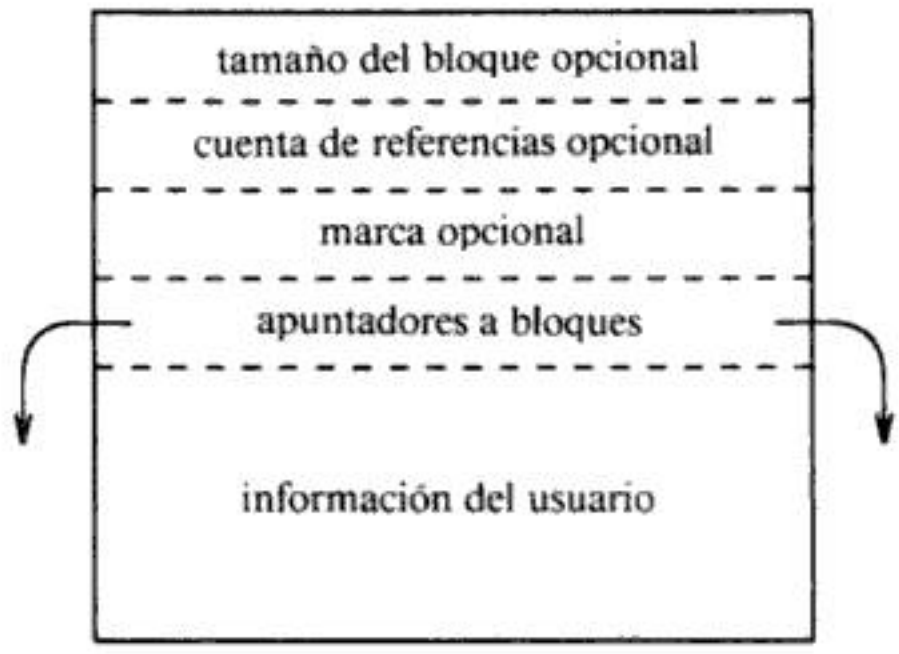


Fig. 7.43. El formato de un bloque.

los bloques de memoria. Para el presente análisis, supóngase que el formato de un bloque de memoria es como el de la figura 7.43.

El primer problema es el de reconocer las fronteras del bloque. Si el tamaño de los bloques es fijo, entonces se puede utilizar la información de la posición. Por ejemplo, si cada bloque ocupa 20 palabras, entonces un bloque nuevo comienza cada 20 palabras. De lo contrario, en la memoria inaccesible asociada a un bloque, se conserva el tamaño del bloque, así que se puede determinar dónde comienza el siguiente bloque.

El segundo problema es el de reconocer si un bloque está en uso. Se supone que un bloque está en uso si es posible que el programa de usuario haga referencia a la información contenida en el bloque. Se puede producir la referencia a través de un apuntador o después de seguir una secuencia de apuntadores, así que el compilador necesita saber la posición en la memoria de todos los apuntadores. Utilizando el formato de la figura 7.43, los apuntadores se guardan en una posición fija dentro del bloque. Más concretamente, se supone que el área de información del usuario de un bloque no contiene ningún apuntador.

Se pueden utilizar dos enfoques para la desasignación implícita. Aquí sólo se esbozan; para más detalles, véase Aho, Hopcroft y Ullman [1983].

1. *Cuenta de referencias.* Se lleva la cuenta del número de bloques que apuntan directamente al presente bloque. Si en algún momento la cuenta disminuye a 0, entonces se puede desasignar el bloque porque no se le puede hacer referencia. En otras palabras, el bloque se ha convertido en basura que puede recogerse. Mantener la cuenta de las referencias puede suponer mucho tiempo; la asignación de apuntadores `p:=q` produce cambios en las cuentas de referencias de los bloques apuntados por `p` y por `q`. La cuenta para el bloque apuntado por `p` disminuye en uno, mientras que la cuenta para el bloque apuntado por `q` aumenta en uno. Las cuentas de referencias se utilizan más adecuadamente cuando los apuntadores entre bloques nunca aparecen en ciclos. Por ejemplo, en la figura 7.44, ningún bloque es accesible desde ningún otro, así que ambos son basura, pero cada uno tiene una cuenta de referencias de uno.

2. *Técnicas de marca.* Un enfoque alternativo es suspender temporalmente la ejecución del programa de usuario y utilizar los apuntadores congelados para determinar los bloques que están siendo utilizados. Este enfoque exige que se conozcan todos los apuntadores del montículo. Conceptualmente, se vierte pintura en el montículo a través de estos apuntadores. Cualquier bloque alcanzado por la pintura está en uso y puede desasignarse el resto. Más concretamente, se recorre el montículo y se marcan todos los bloques como *no usados*. Después se siguen los apuntadores que marcan como *usados* cualquier bloque alcanzado en el proceso. Un examen secuencial final del montículo permite recoger todos los bloques todavía marcados como *no usados*.

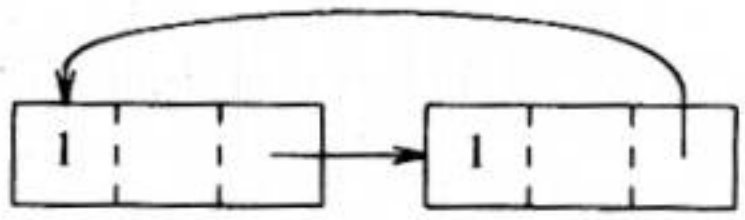


Fig. 7.44. Nodos basura con cuentas de referencias distintas de cero.

Con bloques de tamaño variable, se tiene la posibilidad adicional de desplazar los bloques de memoria utilizados de sus posiciones actuales[8]. Este proceso, llamado *compactación*, traslada todos los bloques utilizados a un extremo del montículo, así que toda la memoria libre se puede reunir en un gran bloque libre. La compactación también necesita información sobre los apuntadores en los bloques porque cuando se traslada un bloque utilizado, hay que ajustar todos los apuntadores a él para reflejar el cambio. Su ventaja es que después de la compactación se elimina la fragmentación de la memoria disponible.

7.9 ASIGNACION DE MEMORIA EN FORTRAN

FORTRAN fue diseñado para permitir la asignación estática de la memoria, como en la sección 7.3. Sin embargo, hay algunos aspectos, como el tratamiento de las declaraciones COMMON y EQUIVALENCE, que son bastante especiales en FORTRAN. Un compilador de FORTRAN puede crear varias *áreas de datos*, es decir, bloques de memoria en los que se pueden almacenar los valores de objetos. En FORTRAN, existe un área de datos para cada procedimiento y un área de datos para cada bloque COMMON con nombre y para COMMON en blanco, si se utiliza. La tabla de símbolos debe registrar para cada nombre el área de datos a la que pertenece y su desplazamiento dentro de dicho área de datos, es decir, su posición relativa al comienzo del área. El compilador debe decidir dónde van las áreas de datos con respecto al código ejecutable y a ellas mismas, pero esta elección es arbitraria porque las áreas de datos son independientes.

El compilador debe calcular el tamaño de cada área de datos. Para las áreas de datos de los procedimientos basta con un solo contador, porque sus tamaños se co-

---

[8] También se podría hacer con bloques de tamaño fijo, pero no supondría ninguna ventaja.

nocen después que se procesa cada procedimiento. Para los bloques COMMON, se debe guardar un registro para cada bloque durante el procesamiento de todos los procedimientos, ya que cada procedimiento que utiliza un bloque puede tener su propia idea del tamaño del bloque, y el tamaño real es el máximo de los tamaños derivados de los distintos procedimientos. Si los procedimientos se compilan por separado, se debe utilizar un editor de enlaces para seleccionar el tamaño del bloque COMMON de manera que sea el máximo de todos esos bloques con el mismo nombre entre las partes de código que se están enlazando.

Para cada área de datos, el compilador crea un *mapa de memoria*, que es una descripción del contenido del área. Este "mapa de memoria" puede ser simplemente una indicación, en la entrada de la tabla de símbolos de entrada de cada nombre del área, del desplazamiento de ese nombre en el área. No es necesario tener una fácil respuesta a la pregunta: "¿Cuáles son todos los nombres en esta área de datos?" Sin embargo, en FORTRAN se conoce la respuesta para las áreas de datos de los procedimientos, porque todos los nombres declarados en un procedimiento que no sean COMMON o equivalentes a un nombre COMMON están en el área de datos del procedimiento. Los nombres COMMON pueden tener sus entradas a la tabla de símbolos enlazadas, con una cadena para cada bloque COMMON, en el orden de su aparición en el bloque. De hecho, como los desplazamientos de los nombres en el área de datos no siempre se pueden determinar hasta que se procese el procedimiento completo (las matrices en FORTRAN se pueden declarar antes de declarar sus dimensiones), es necesario crear estas cadenas de nombres COMMON.

Un programa en FORTRAN consta de un programa principal, subrutinas y funciones (a todas se les llamará *procedimientos*). Cada caso de un nombre tiene un ámbito formado por sólo un procedimiento. Se puede generar código objeto para cada procedimiento al alcanzar el final de dicho procedimiento. Si se hace así, es posible desechar la mayor parte de la información de la tabla de símbolos. Sólo hay que conservar aquellos nombres externos a la rutina que se acaba de procesar. Estos son nombres de otros procedimientos y de bloques comunes. Estos nombres pueden no ser verdaderamente externos al programa completo que se está compilando, pero se deben conservar hasta que se procese la serie completa de procedimientos.

Datos en áreas COMMON

Para cada bloque se crea un registro que proporciona el primer y el último nombre, pertenecientes al procedimiento en curso, que están declarados como pertenecientes a ese bloque COMMON. Cuando se procesa una declaración como

```
COMMON /BLOQ1/ NOMB1, NOMB2
```

el compilador debe hacer lo siguiente:

1. En la tabla para los nombres de bloques COMMON, créese un registro para BLOQ1, si es que no existe ya uno.
2. En las entradas de la tabla de símbolos para NOMB1 y NOMB2, colóquese un apuntador a la entrada en la tabla de símbolos para BLOQ1, indicando que están dentro de COMMON y que pertenecen a BLOQ1.

3. a) Si se acaba de crear el registro para BLOQ1, colóquese un apuntador en dicho registro apuntando a la entrada de la tabla de símbolos correspondiente a NOMB1, indicando el primer nombre dentro de este bloque COMMON. Después, enlácese la entrada de la tabla de símbolos correspondiente a NOMB1 con la del NOMB2, utilizando un campo de la tabla de símbolos reservado para enlazar miembros del mismo bloque COMMON. Por último, colóquese un apuntador dentro del registro para BLOQ1 apuntando a la entrada de la tabla de símbolos correspondiente a NOMB2, indicando el último miembro encontrado de ese bloque.

   b) Sin embargo, si ésta no es la primera declaración de BLOQ1, enlácense simplemente NOMB1 y NOMB2 al final de la lista de nombres para BLOQ1. Por supuesto, se actualiza el apuntador al final de la lista para BLOQ1, que aparece en el registro para BLOQ1.

Después de haber procesado un procedimiento, se aplica el algoritmo de equivalencia, que se analizará en breve. Se puede descubrir que algunos nombres adicionales pertenecen al bloque COMMON, porque se han hecho equivalentes a nombres que están dentro de COMMON. Se verá que no es realmente necesario enlazar un nombre XYZ a la cadena correspondiente a su bloque COMMON. Se asigna un bit en la entrada de la tabla de símbolos para XYZ, indicando que XYZ se ha hecho equivalente a algún otro nombre. Entonces, una estructura de datos que se estudiará más adelante dará la posición de XYZ con respecto a algún nombre declarado como parte de COMMON.

Después de realizar las operaciones de equivalencia, se puede crear un mapa de memoria para cada bloque COMMON examinando la lista de nombres correspondiente a dicho bloque. Se asigna el valor inicial de cero a un contador, y para cada nombre en la lista, se iguala su desplazamiento al valor en curso del contador. Después se añade al contador el número de unidades de memoria empleadas por el objeto de datos indicado por el nombre. Entonces se pueden borrar los registros del bloque COMMON y el siguiente procedimiento puede reutilizar el espacio.

Si un nombre XYZ dentro de un bloque COMMON se hace equivalente a un nombre que no esté en COMMON, hay que determinar el desplazamiento máximo desde el principio de XYZ para toda palabra de memoria necesaria para todo nombre hecho equivalente a XYZ. Por ejemplo, si XYZ es un número real, equivalente a A(5,5), donde A es una matriz de 10×10 de números reales, A(1,1) aparece 44 palabras antes de XYZ y A(10,10) aparece 55 palabras después de XYZ, como se muestra en la figura 7.45. La existencia de A no afecta al contador del bloque COMMON; sólo aumenta en una palabra cuando se considera XYZ, independientemente del nombre al que XYZ se haga equivalente. Sin embargo, el final del área de datos para el bloque COMMON debe estar lo suficiente alejado del principio para albergar a la matriz A. Por tanto, se registra el desplazamiento más largo, desde el inicio del bloque COMMON, de toda palabra utilizada por un nombre hecho equivalente a un miembro de dicho bloque. En la figura 7.45, esta cantidad debe ser al menos el desplazamiento de XYZ más 55. También se comprueba que la matriz A no se prolongue delante del principio del área de datos; es decir, el desplazamiento de XYZ debe ser al menos 44. En caso contrario, hay un error y se debe producir un mensaje diagnóstico.

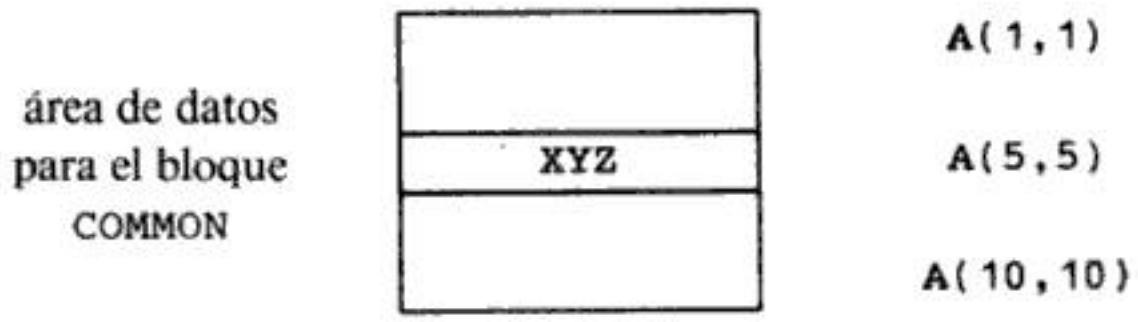


Fig. 7.45. Relación entre proposiciones COMMON y EQUIVALENCE.

Un algoritmo de equivalencia sencillo

Los primeros algoritmos para procesar proposiciones de equivalencia aparecieron en ensambladores, no en compiladores. Como estos algoritmos pueden ser bastante complejos, sobre todo cuando se consideran interacciones entre las proposiciones COMMON y EQUIVALENCE, se estudiará primero una situación típica de un lenguaje ensamblador, donde las únicas proposiciones EQUIVALENCE son de la forma

```
EQUIVALENCE A,B+desplazamiento
```

en donde A y B son nombres de posiciones. Esta proposición hace que A indique la posición que está *desplazamiento* unidades de memoria más allá de la posición B.

Una secuencia de proposiciones EQUIVALENCE agrupa nombres en *conjuntos de equivalencia* cuyas posiciones relativas unas a otras vienen definidas todas por las proposiciones EQUIVALENCE. Por ejemplo, la secuencia de proposiciones

```
EQUIVALENCE A, B+100
EQUIVALENCE C, D-40
EQUIVALENCE A, C+30
EQUIVALENCE E, F
```

agrupa los nombres en los conjuntos {A, B, C, D} y {E, F}, donde E y F indican la misma posición, C está 70 posiciones después de B, A 30 después de C y D 10 después de A.

Para calcular los conjuntos de equivalencia se crea un árbol para cada conjunto. Cada nodo de un árbol representa un nombre y contiene el desplazamiento de dicho nombre en relación con el nombre correspondiente al padre de ese nodo. El nombre que se encuentra en la raíz de un árbol se denomina *líder*. La posición de cualquier nombre en relación con el líder se calcula siguiendo el camino que va desde el nodo para dicho nombre y sumando los desplazamientos a lo largo del camino.

Ejemplo 7.12. El conjunto de equivalencia {A, B, C, D} mencionado anteriormente podría representarse mediante el árbol que se muestra en la figura 7.46. D es el líder, y se puede comprobar que A está situada 10 posiciones antes de D, ya que la suma de los desplazamientos en el camino de A a D es $100+(-110) = -10$. □

Ahora se proporciona un algoritmo para construir árboles para conjuntos de equivalencia. Los campos relevantes en las entradas de la tabla de símbolos son:

1. *padre*, que apunta a la entrada de la tabla de símbolos para el padre, que es nulo si el nombre es una raíz (o no ha sido hecho equivalente a nada), y

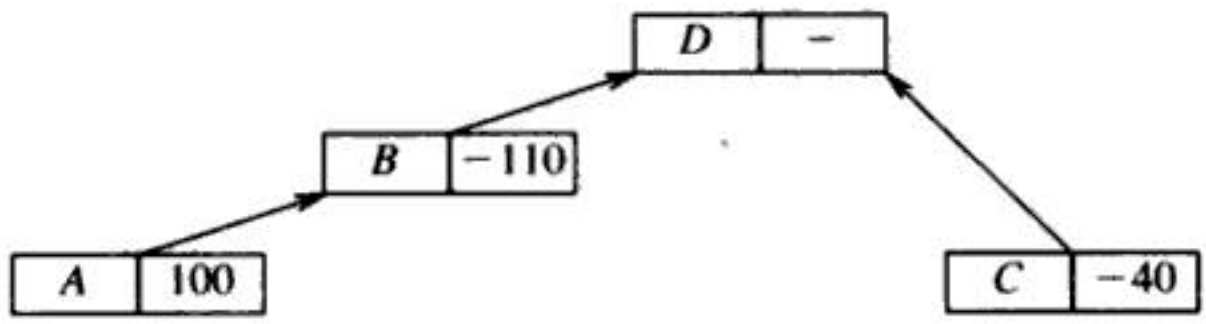


Fig. 7.46. Arbol que representa un conjunto de equivalencia.

2. *desplazamiento*, que da el desplazamiento de un nombre con respecto al nombre del padre.

Este algoritmo asume que cualquier nombre podría ser el líder de un conjunto de equivalencia. En la práctica, en un lenguaje ensamblador, un solo nombre dentro del conjunto tendría una posición real definida por una seudooperación, y este nombre se convertiría en líder. Se supone que el lector sabe cómo modificar el algoritmo para convertir un nombre particular en el líder.

Algoritmo 7.1. Construcción de árboles de equivalencia.

Entrada. Una lista de proposiciones que definen las equivalencias de la forma

```
EQUIVALENCE    A,B+dist
```

Salida. Una serie de árboles tal que se pueda determinar la posición del nombre en relación con el líder, para cualquier nombre mencionado en la lista de equivalencias

```
        begin
(1)         poner a p y q apuntando a los nodos para A y B, respectivamente;
(2)         c := 0; d := 0;  /* c y d calculan los desplazamientos de A y B a partir
                de los líderes de sus respectivos conjuntos */
(3)         while padre(p) ≠ null do begin
(4)             c := c + desplazamiento(p);
(5)             p := padre(p)
            end;        /* se traslada p al líder de a, acumulando
                           desplazamientos conforme se avanza */
(6)         while padre(q) ≠ null do begin
(7)             d := d + desplazamiento(q);
(8)             q := padre(q)
            end;        /* se hace lo mismo para B */
(9)         if p = q then /* A y B ya se hicieron equivalentes */
(10)            if c-d ≠ dist then error;
                    /* a A y B se les han dado dos posiciones relativas distintas */
            else begin      /* fusionar los conjuntos de A y B */
(11)            padre(p) := q          /* al líder de A se le hace hijo del líder de B */
(12)            desplazamiento(p) := d-c+dist
            end
        end
```

Fig. 7.47. Algoritmo de equivalencia.

de entrada, siguiendo el camino desde ese nombre a la raíz y sumando los valores de *desplazamiento* encontrados a lo largo del camino.

Método. Se repiten los pasos de la figura 7.47 para cada proposición de equivalencia `EQUIVALENCE A,B+`*dist*, por turno. La justificación para la fórmula de la línea (12) para el desplazamiento del líder de A en relación con el líder de B es la siguiente. La posición de A, por ejemplo l_A, es igual a c más la posición del líder de A, por ejemplo m_A. La posición de B, por ejemplo l_B, es igual a d más la posición del líder de B, por ejemplo m_B. Pero $l_A = l_B + dist$, de modo que $c + m_A = d + m_B + dist$. Así, $m_A - m_B$ es igual a $d - c + dist$. □

Ejemplo 7.13. Si se procesa

```
EQUIVALENCE    A, B+100
EQUIVALENCE    C, D-40
```

se obtiene la configuración de la figura 7.46, pero sin el desplazamiento −110 en el nodo correspondiente a B y sin enlace de B a D. Cuando se procesa

```
EQUIVALENCE    A, C+30
```

se comprueba que p apunta a B después del lazo **while** de la línea (3) y q apunta a d después del lazo **while** de la línea (6). También se ve que $c = 100$ y $d = -40$. Después, en la línea (11) se convierte D en el padre de B y se asigna 100 al campo de *desplazamiento* de B, que es $(-40)-(100)+30$. □

El algoritmo 7.1 emplearía un tiempo proporcional a n^2 para procesar n equivalencias, ya que en el peor caso los caminos seguidos en la lazos de las líneas (3) y (6) podrían incluir todos los nodos de sus árboles respectivos. Realizar equivalencias exige sólo una pequeña fracción del tiempo empleado en la compilación, así que n^2 pasos no es prohibitivo, y probablemente no se necesita un algoritmo más complejo que el de la figura 7.47. Sin embargo, hay dos cosas sencillas para hacer que el algoritmo 7.1 emplee un tiempo que sea casi lineal en función del número de equivalencias que procesa. Como es poco probable que los conjuntos de equivalencia sean lo bastante grandes, de promedio, como para que sea necesario implantar esas mejoras, conviene observar que hacer equivalencias sirve como paradigma para varios procesos importantes que requieren "fusión de conjuntos". Por ejemplo, varios algoritmos eficientes para el análisis del flujo de datos dependen de rápidos algoritmos de equivalencia; el lector interesado puede remitirse a las notas bibliográficas del capítulo 10.

La primera mejora que se puede realizar es llevar una cuenta, por cada líder, del número de nodos de su árbol. Después, en las líneas (11) y (12), en vez de enlazar arbitrariamente el líder de A con el líder de B, enlazar el que tenga la cuenta menor con el otro. Esto garantiza que el árbol crezca a lo ancho de modo que los caminos serán cortos. Se deja como ejercicio demostrar que n equivalencias realizadas de esta forma no pueden producir caminos más largos que $\log_2 n$ nodos.

La segunda idea se conoce como compresión de caminos. Cuando se sigue un camino hacia la raíz en los lazos de las líneas (3) y (6), hay que hacer que todos los

nodos encontrados sean hijos del líder, si es que todavía no lo son. Es decir, mientras se sigue el camino, hay que registrar todos los nodos $n_1, n_2, \ldots, n_k$ encontrados, donde n es el nodo correspondiente a A o B y n_k es el líder. Después hay que ajustar los desplazamientos y hacer que $n_1, n_2, \ldots, n_{k-2}$ sean hijos de n_k mediante los pasos de la figura 7.48.

```
begin
    h := desplazamiento (n_{k-1});
    for i := k-2 downto 1 do begin
        padre (n_i) := n_k;
        h := h + desplazamiento (n_i);
        desplazamiento (n_i) := h
    end
end
```

Fig. 7.48. Ajuste de desplazamientos.

Un algoritmo de equivalencias para FORTRAN

Hay varias características adicionales que deben añadirse al algoritmo 7.1 para hacer que funcione para FORTRAN. Primero se debe determinar si un conjunto de equivalencia está o no en un bloque COMMON, lo cual se puede hacer registrando para cada líder si alguno de los nombres dentro de su conjunto está en COMMON, y si es así, en qué bloque.

Segundo, en un lenguaje ensamblador, un miembro de un conjunto de equivalencia convertirá el conjunto completo en una aplicacion real siendo una etiqueta de una proposición y permitiendo así que las direcciones indicadas por todos los nombres del conjunto se calculen con respecto a dicha posición. En FORTRAN, sin embargo, le corresponde al compilador determinar las posiciones de memoria, de modo que un conjunto de equivalencia que no esté en un bloque COMMON se puede considerar como "flotante" hasta que el compilador determine la posición del conjunto completo en su área de datos apropiada. Para hacerlo correctamente, el compilador necesita conocer la extensión del conjunto de equivalencia, es decir, el número de posiciones que ocupan colectivamente los nombres del conjunto. Para resolver este problema, se asocian dos campos al líder, *baja*, y *alta*, que proporcionan los desplazamientos con respecto al líder de las posiciones más baja y más alta utilizadas por los miembros del conjunto de equivalencia. Tercero, hay algunos problemas menores derivados del hecho de que los nombres pueden ser matrices y las posiciones intermedias de la matriz se pueden hacer equivalentes a posiciones de otras matrices.

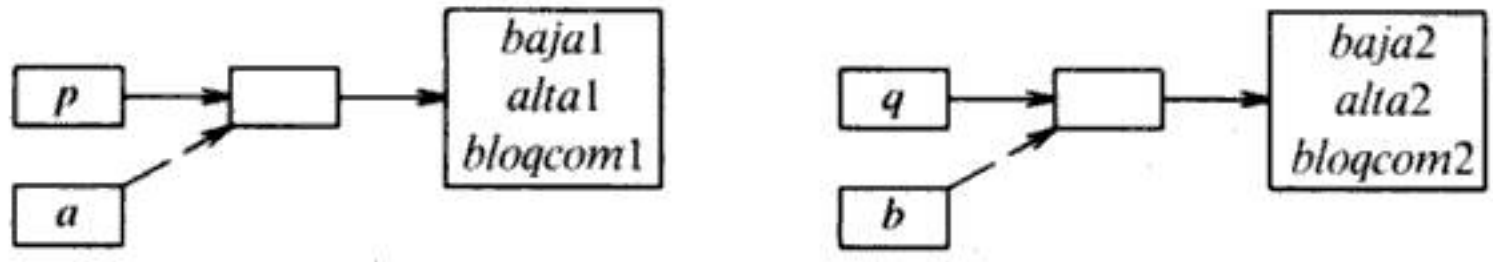
Como hay tres campos (*baja, alta* y un apuntador a un bloque COMMON) que deben asociarse a cada líder, no hay que asignar espacio para estos campos en todas las entradas de la tabla de símbolos. Un curso de acción consiste en utilizar el campo *padre* del algoritmo 7.1 para apuntar, en el caso del líder, a un registro dentro de una tabla nueva con tres campos, *baja*, *alta* y *bloqcom*. Como esta tabla y la tabla de símbolos ocupan áreas disjuntas, se puede saber a qué tabla apunta un apunta-

dor. A veces, la tabla de símbolos contiene un bit que indica si un nombre es o no un líder. Si el espacio es realmente escaso, se muestra en los ejercicios un algoritmo alternativo que evita esta tabla adicional pero que exige un mayor esfuerzo de programación.

Considérense los cálculos que deben sustituir las líneas (11) y (12) de la figura 7.47. En la figura 7.49(a) se presenta la situación en la que dos conjuntos de equivalencia, cuyos líderes apuntan a p y q, se deben fusionar. La estructura de datos que representa a los dos conjuntos aparece en la figura 7.49(b). Primero se debe comprobar que no haya dos miembros entre los dos conjuntos de equivalencia que estén en COMMON. Aunque ambos estén en el mismo bloque, la definición estándar de FORTRAN prohíbe que se hagan equivalentes. Si cualquiera de los bloques COMMON contiene un miembro de uno u otro conjunto de equivalencia, entonces el conjunto fusionado tiene un apuntador al registro correspondiente a ese bloque en *bloqcom*. En la figura 7.50 se muestra el código que realiza esta comprobación, suponiendo que el líder apuntado por q se convierte en el líder del conjunto fusionado. Entre las líneas (11) y (12) de la figura 7.47 se debe calcular también la extensión del conjunto de equivalencia fusionado. La figura 7.49(a) indica las fórmulas para los nuevos valores de *baja* y *alta* con respecto al líder apuntado q.

$$baja = \min(baja2,\ baja1 - c + dist + d)$$
$$alta = \max(alta2,\ alta1 - c + dist + d)$$

(a) posiciones relativas de conjuntos de equivalencia



(b) estructura de datos

Fig. 7.49. Fusión de conjuntos de equivalencia.

```
begin
    bloqcom1 := bloqcom (padre (p));
    bloqcom2 := bloqcom (padre (q));
    if bloqcom1 ≠ null and bloqcom2 ≠ null then
        error;      /* hay dos nombres equivalentes dentro de COMMON */
    else if bloqcom2 = null then
        bloqcom (padre (q)) := bloqcom1
end
```

Fig. 7.50. Cálculo de bloques COMMON.

Por tanto, se debe hacer

```
begin
    baja (padre (q)) := mín(baja (padre (q)), baja (padre (p)) − c+dist+d);
    alta (padre (q)) := máx(alta (padre (q)), alta (padre (p)) − c+dist+d)
end
```

Estas proposiciones van seguidas de las líneas (11) y (12) de la figura 7.47 para efectuar la fusión de los dos conjuntos de equivalencia.

Se deben tener en cuenta dos últimos detalles para que el algoritmo 7.1 funcione para FORTRAN. En FORTRAN se pueden hacer equivalentes posiciones intermedias de matrices a otras posiciones en otras matrices o a nombres simples. El desplazamiento de una matriz A desde su líder supone el desplazamiento de la primera posición de A desde la primera localidad del líder. Si una posición como `A(5,7)` se hace equivalente a `B(20)`, por ejemplo, se debe calcular la posición de `A(5,7)` con respecto a `A(1,1)` e inicializar *c* con el negativo de esta distancia en la línea (2) de la figura 7.47. Asimismo, *d* debe inicializarse con el negativo de la posición de `B(20)` con respecto a `B(1)`. Las fórmulas de la sección 8.3 junto con el conocimiento del tamaño de los elementos de las matrices `A` y `B`, son suficientes para calcular los valores iniciales de *c* y *d*.

El último detalle a considerar es el hecho de que FORTRAN permite una proposición `EQUIVALENCE` que abarca muchas posiciones, como:

```
EQUIVALENCE (A(5,7), B(20), C, D(4,5,6))
```

Estas se pueden considerar como

```
EQUIVALENCE (B(20), A(5,7))
EQUIVALENCE (C, A(5,7))
EQUIVALENCE (D(4,5,6), A(5,7))
```

Obsérvese que si se realizan las equivalencias en este orden, sólo `A` se convierte en el líder de un conjunto que contiene más de un elemento. Se puede utilizar muchas veces un registro con *baja, alta* y *bloqcom* para "conjuntos de equivalencia" de un solo nombre.

Transformaciones de correspondencia de áreas de datos

A continuación se describen las reglas mediante las cuales se asigna espacio en las distintas áreas de datos para los nombres de cada rutina.

1. Para cada bloque COMMON, visiténse todos los nombres declarados como pertenecientes a ese bloque en el orden de sus declaraciones (utilícense las cadenas de nombres del bloque COMMON creadas en la tabla de símbolos para este propósito). Asígnese el número de palabras necesarias para cada nombre por turno, llevando una cuenta del número de palabras asignadas, de modo que se puedan calcular los desplazamientos para cada nombre. Si un nombre A se hace equivalente, la extensión de su conjunto de equivalencia no tiene importancia, pero se debe comprobar que el valor *baja* para el líder de A no se extienda más allá del principio del bloque COMMON. Consúltese el valor *alta* correspondiente al líder para poner un límite inferior a la última palabra del bloque. Se deja al lector que encuentre las fórmulas exactas para realizar estos cálculos.

2. Visítense todos los nombres de la rutina en cualquier orden.

   a) Si un nombre está en un bloque COMMON, no se hace nada. Su espacio ya ha sido asignado en (1).

   b) Si un nombre no está dentro de un bloque COMMON y no se declara equivalente, asígnese el número de palabras necesarias en el área de datos para la rutina.

   c) Si un nombre A se declara equivalente, encuéntrese su líder, por ejemplo L. Si a L ya se le dio una posición en el área de datos de la rutina, calcúlese la posición de A sumándole a esa posición todos los valores *desplazamiento* encontrados en el camino de A a L en el árbol que representa el conjunto de equivalencia de A y L. Si a L no se le ha dado una posición, asígnense las siguientes (*alta* – *baja*) palabras en el área de datos para el conjunto de equivalencia. La posición de L en estas palabras es $-$*baja* palabras del principio, y la posición de A se puede calcular sumando los valores *desplazamiento* como antes.

EJERCICIOS

7.1 Utilizando las reglas de ámbito de Pascal, determínense las declaraciones que se aplican a cada caso de los nombres a y b de la figura 7.51. El resultado del programa consta de los enteros 1 al 4.

```
program a(input, output);
procedure b(u, v, x, y: integer);
    var a : record a, b : integer end;
        b : record b, a : integer end;
    begin
       with a do begin a := u; b := v end;
       with b do begin a := x; b := y end;
       writeln(a.a, a.b, b.a, b.b)
    end;
begin
    b(1, 2, 3, 4)
end.
```

Fig. 7.51. Programa en Pascal con varias declaraciones de a y b.

7.2 Considérese un lenguaje estructurado por bloques en el que un nombre se puede declarar como entero o real. Supóngase que las expresiones se representan mediante el terminal **expr** y que las únicas proposiciones son asignaciones, condicionales, lazos **while** y secuencias de proposiciones. Suponiendo que a los enteros se les asigna una palabra y a los reales dos palabras, proporciónese un algoritmo dirigido por la sintaxis (basado en una gramática razonable para declaraciones y bloques) para determinar los enlaces de nombres a palabras que pueden ser utilizadas por una activación de un bloque. ¿Utiliza la asignación el mínimo número de palabras apropiado para cualquier ejecución del bloque?

***7.3** En la sección 7.4 se vio que el *display* se podía mantener correctamente si cada procedimiento a profundidad i guardara $d[i]$ al principio de una activación y restableciera $d[i]$ al final. Demuéstrese por inducción sobre el número de llamadas que cada procedimiento observa un *display* correcto.

7.4 Una *macro* es una forma de procedimiento, implantada sustituyendo literalmente el cuerpo para cada llamada de procedimiento. En la figura 7.52 se muestra un programa en *pic* y su salida. Las primeras dos líneas definen a las macros `muestra` y `chico`. Los cuerpos de las macros están contenidos entre los dos signos `%` en las líneas. Cada uno de los cuatro círculos de la figura es dibujado con `muestra`; el radio del círculo viene dado por el nombre no local `r`. Los bloques en *pic* están delimitados por [y]. Cada variable a la que se le asigne un valor dentro de un bloque se declara implícitamente dentro del bloque. Según el resultado, ¿qué se puede decir del ámbito de cada caso de `r`?

```
define muestra % { círculo radio r en Aquí } %
define chico % [ r = 1/12; muestra ] %
[
    r = 1/6;
    muestra; chico;
    traslada;
    muestra; chico;
]
```

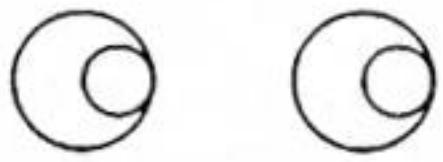


Fig. 7.52. Círculos dibujados por un programa en *pic*.

7.5 Escríbase un procedimiento para insertar un elemento en una lista enlazada pasando un apuntador a la cabeza de la lista. ¿Con qué mecanismos de paso de parámetros funciona este procedimiento?

7.6 ¿Qué imprime el programa de la figura 7.53, suponiendo (a) llamada por valor, (b) llamada por referencia, (c) enlace de copia y restauración y (d) llamada por nombre?

7.7 Cuando se pasa como parámetro un procedimiento en un lenguaje de ámbito léxico, se puede pasar su ambiente no iocal utilizando un enlace de acceso. Proporciónese un algoritmo para determinar dicho enlace.

```
program principal (input, output);
    procedure p (x, y, z);
        begin
            y := y + 1;
            z := z + x;
        end;
    begin
        a := 2;
        b := 3;
        p (a+b, a, a);
        print a
    end.
```

Fig. 7.53. Seudoprograma que ilustra el paso de parámetros.

7.8 En el programa de Pascal de la figura 7.54 se ilustran las tres clases de ambientes que se podrían asociar con un procedimiento pasado como parámetro. Los ambientes *léxico*, *de pasada* y *de activación* de dicho procedimiento constan en los enlaces de identificadores en el punto en que se define el procedimiento, pasadas como parámetro y activadas, respectivamente. Considérese la función `f`, pasada como parámetro en la línea 11. Utilizando los ambientes léxico, de pasada y de activación para `f`, el nombre no local `m` en la línea 8 está en el ámbito de las declaraciones de `m` en las líneas 6, 10 y 3, respectivamente.

a) Dibújese el árbol de activación para este programa.

```
(1)  program param(input, output);
(2)      procedure b (function h(n: integer): integer);
(3)          var m : integer;
(4)      begin m := 3; writeln(h(2)) end { b };
(5)  procedure c;
(6)      var m : integer;
(7)          function f(n : integer) : integer;
(8)              begin f := m + n end { f };
(9)          procedure r;
(10)             var m : integer;
(11)             begin m := 7; b(f) end { r };
(12)     begin m := 0; r end { c };
(13) begin
(14)     c
(15) end.
```

Fig. 7.54. Ejemplo de ambientes léxico, de pasada y de activación.

b) ¿Cuál es el resultado del programa, utilizando los ambientes léxico, de pasada y de activación para `f`?

*c) Modifíquese la aplicación por medio de *display* de un lenguaje con ámbito léxico para construir el ambiente léxico correctamente cuando se active un procedimiento pasado como parámetro.

*7.9 La proposición $f := a$ en la línea 11 del seudoprograma de la figura 7.55 llama a la función a, la cual devuelve como resultado la función *sumam*.

a) Dibújese el árbol de activación para una ejecución de este programa.

b) Supóngase que se utiliza ámbito léxico para los nombres no locales. ¿Por qué fallará el programa si se utiliza asignación por medio de una pila?

c) ¿Cuál es el resultado del programa con asignación con un montículo?

```
(1)  program devuelve (input, output);
(2)  var f: function (integer): integer;
(3)    function a : function (integer) : integer;
(4)      var m : integer;
(5)      function sumam (n : integer) : integer;
(6)        begin return m + n end;
(7)      begin m := 0; return sumam end;
(8)    procedure b (g : function (integer) : integer);
(9)      begin writeln (g (2)) end;
(10)   begin
(11)     f := a; b(f)
(12)   end.
```

Fig. 7.55. Seudoprograma en el que la función *sumam* se devuelve como resultado.

*7.10 Algunos lenguajes, como LISP, tienen la capacidad de devolver procedimientos recién creados durante la ejecución. En la figura 7.56, todas las funciones, definidas en el texto fuente o creadas durante la ejecución, toman a lo sumo un argumento y devuelven un valor, bien sea una función o un número real. El operador $\circ$ representa la composición de funciones; es decir, $(f \circ g)(x) = f(g(x))$.

a) ¿Qué valor imprime *principal*?

*b) Supóngase que siempre que se cree y se devuelva un procedimiento p, su registro de activación se convierte en un hijo del registro de activación de la función que devuelve p. Entonces el ambiente de pasada de p se puede mantener conservando un árbol de registros de activación en lugar de una pila. ¿Cuál es el árbol de registros de activación cuando se calcula a en *principal* en la figura 7.56?

*c) Como alternativa, supóngase que se crea un registro de activación para p cuando se activa p, y se convierte en un hijo del registro de activación

correspondiente al procedimiento que llama a *p*. Se puede utilizar este enfoque para mantener el registro de activación correspondiente a *p*. Dibújense las situaciones de los registros de activación y sus relaciones padre-hijo conforme se ejecutan las proposiciones de *principal*. ¿Es una pila suficiente para guardar los registros de activación cuando se utiliza este enfoque?

```
function f(x: function);
var   y : function;
          y := x ∘ h;     /* crea a y cuando se le ejecuta */
          return y
end {f};
function h( );
     return sin
end { h };
function g (z : function);
var   w : function;
          w := arctan ∘ z;     /* crea a w cuando se le ejecuta */
          return w
end { g };
function principal( );
var   a : real;
          u, v : function;
          v := f(g);
          u := v( );
          a := u(π/2);
          print a
end { principal }.
```

Fig. 7.56. Seudoprograma que crea funciones durante la ejecución.

7.11 Otra forma de llevar a cabo el borrado de tablas de dispersión para los nombres cuyo ámbito haya sido pasado (como en la Sec. 7.6), es dejar los nombres que han expirado en una lista hasta que se examine de nuevo dicha lista. Suponiendo que las entradas incluyen el nombre del procedimiento en el que se hace la declaración, en principio se puede saber si un nombre es o no viejo, y borrarlo si lo es. Proporciónese un esquema de indización para procedimientos que permitan saber en un tiempo $O(1)$ si un procedimiento es "antiguo", es decir, si su ámbito ya ha sido pasado.

7.12 Muchas funciones de dispersión pueden caracterizarse con una secuencia de constantes enteras $\alpha_0, \alpha_1, \ldots$. Si c_i, $1 \leq i \leq n$, es el valor entero del i-ésimo carácter en la cadena s, entonces la cadena se direcciona por la dispersión hacia

$$dispersiôn\ (s) = (\alpha_0 + \sum_{i=1}^{n} \alpha_i c_i)\ \textbf{mod}\ m$$

donde m es el tamaño de la tabla de dispersión. Para cada uno de los siguientes casos, determínese la secuencia de constantes $\alpha_0, \alpha_1, \ldots$ o demuéstrese que no existe dicha secuencia. Cada caso determina un entero; se obtiene un valor de dispersión tomando ese entero **mod** m.

a) Tómese la suma de los caracteres.
b) Tómese la suma del primer y el último caracteres.
c) Tómese h_n, donde $h_0 = 0$ y $h_i = 2h_{i-1} + c_i$.
d) Considérense los bits de los 4 caracteres situados en el medio como un entero de 32 bits.
e) Un entero de 32 bits se puede considerar formado por 4 *bytes*, donde cada *byte* es un dígito que toma 256 valores posibles. Comenzando con 0000, para $1 \leq i \leq n$, sumar c_i al *byte* i **mod** 4, con acarreos permitidos. Es decir, c_1 y c_5 se suman al *byte* 1, c_2 y c_6 al *byte* 2, etc. Devuélvase el valor final.

***7.13** ¿Por qué las funciones de dispersión caracterizadas con una secuencia de enteros $\alpha_0, \alpha_1, \ldots$, como en el ejercicio 7.12, a veces tienen un bajo rendimiento si la entrada consta de cadenas consecutivas, por ejemplo, `v000`, `v001`, . . . ? El sistema es que en alguna parte del proceso, su comportamiento se desvía de lo fortuito y se puede predecir.

****7.14** Cuando n cadenas se dispersan en m listas, el número medio de cadenas por lista es n/m, independientemente de lo uniformemente que se distribuyan las cadenas. Supóngase que d es una "distribución", es decir, una cadena aleatoria se coloca en la i-ésima lista con probabilidad $d(i)$. Supóngase que una función de dispersión con distribución d coloca b_j cadenas coleccionadas al azar en la lista j, $0 \leq j \leq m-1$. Demuéstrese que el valor previsto $W = \sum_{j=0}^{m-1} (b_j)(b_j+1)/2$ se relaciona linealmente con la varianza de la distribución d. Para una distribución uniforme demuéstrese que el valor previsto de W es $(n/2m)(n + 2m - 1)$.

7.15 Supóngase que hay la siguiente secuencia de declaraciones en un programa en FORTRAN.

```
      SUBROUTINE SUB(X,Y)
      INTEGER A, B(20), C(10,15), D, E
      COMPLEX F, G
      COMMON /BLOQC/ D, E
      EQUIVALENCE (G, B(2))
      EQUIVALENCE (D, F, B(1))
```

Muéstrese el contenido de las áreas de datos para `SUB` y `BLOQC` (al menos la porción del área de `BLOQC` accesible desde `SUB`). ¿Por qué no hay espacio para `X` e `Y`?

***7.16** Una estructura de datos útil para cálculos de equivalencias es la *estructura de anillo*. Se utiliza un apuntador y un campo de desplazamiento en cada entrada de la tabla de símbolos para enlazar los miembros de un conjunto de equivalencia. Esta estructura aparece en la figura 7.57, donde `A`, `B`, `C` y `D`

son equivalentes, y E y F son equivalentes con la posición de B, 20 palabras después de la de A, y así sucesivamente.

a) Proporciónese un algoritmo para calcular el desplazamiento de X con respecto a Y, suponiendo que X e Y están en el mismo conjunto de equivalencia.
b) Proporciónese un algoritmo para calcular *baja* y *alta*, como se definieron en la sección 7.9, con respecto a la posición de algún nombre Z.
c) Proporciónese un algoritmo para procesar

```
EQUIVALENCE U, V
```

No hay que asumir que U y V están necesariamente en conjuntos de equivalencia distintos.

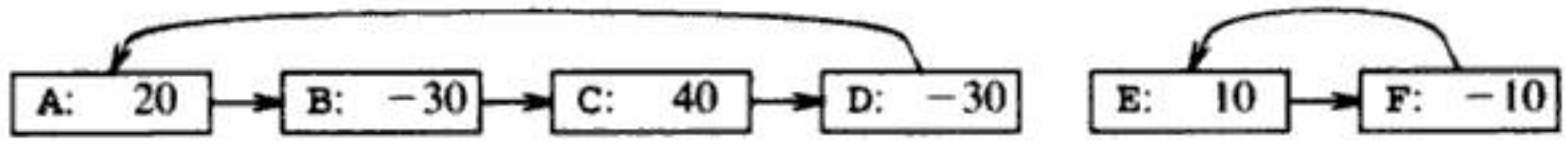


Fig. 7.57. Estructuras de anillo.

*7.17 El algoritmo de la sección 7.9 para transformar áreas de datos exige que se compruebe que *baja* del líder del conjunto de equivalencia de A no haga que el espacio para el conjunto de equivalencia de A se extienda hasta antes del comienzo del bloque COMMON y que se calcule *alta* del líder de A para aumentar el límite superior del bloque COMMON, si fuera necesario. Dénse fórmulas desde el punto de vista de *siguiente*, el desplazamiento de A en el bloque COMMON, y *último*, la última palabra en curso del bloque, para realizar la prueba y actualizar *último* si es necesario.

NOTAS BIBLIOGRAFICAS

Las estructuras de datos tipo pilas han desempeñado un papel fundamental en la implantación de funciones recursivas. McCarthy [1981, pág. 178] recuerda que durante un proyecto de implantación de LISP iniciado en 1958, se decidió "utilizar una sola matriz como pila pública contigua para guardar los valores de las direcciones de las variables devueltas y de retorno de subrutinas devueltas para la implantación de rutinas recursivas". La inclusión de bloques y procedimientos recursivos en ALGOL 60 —véase Naur [1981, Sec. 2.10] para una exposición detallada de su diseño— también estimuló el desarrollo de asignación por medio de pilas. La idea de utilizar una estructura de datos tipo *display* para acceder a los nombres no locales en un lenguaje con ámbito léxico es obra de Dijkstra [1960, 1963]. Aunque LISP utiliza ámbito dinámico, es posible lograr el efecto de ámbito léxico utilizando los denominados "funargs", compuestos por una función y un enlace de acceso; McCarthy [1981] describe el desarrollo de este mecanismo. Los sucesores de LISP como COMMON LISP (Steele [1984]) se han alejado del ámbito dinámico.

En los libros de texto sobre lenguajes de programación se pueden encontrar explicaciones sobre el enlace de nombres; véase, por ejemplo, Abelson y Sussman

[1985], Pratt [1984] o Tennent [1981]. Un enfoque alternativo propuesto en el capítulo 2, consiste en leer la descripción de un compilador. El desarrollo paso a paso en Kernighan y Pike [1984] comienza con una calculadora para expresiones aritméticas y construye un intérprete para un lenguaje sencillo con procedimientos recursivos. Véase también el código para Pascal-S en Wirth [1981]. En Randell y Russell [1964] aparece una descripción detallada de asignación con pilas, el uso de un *display* y la asignación dinámica de matrices.

Johnson y Ritchie [1981] estudian el diseño de una secuencia de llamada que permite que el número de argumentos varíe de llamada a llamada. Un método general para establecer un *display* global consiste en seguir la cadena de enlaces de acceso, estableciendo los elementos del *display* en el proceso. El enfoque de la sección 7.4, que atañe a un solo elemento, parece haber sido bastante conocido durante algún tiempo; una referencia publicada es Rohl [1975]. Moses [1970] analiza las distinciones entre los ambientes que se aplican cuando una función se pasa como parámetro y considera los problemas que surgen cuando dichos ambientes se implantan utilizando acceso superficial y profundo. No puede utilizarse la asignación de la pila para lenguajes con corrutinas o procesos múltiples. Lampson [1982] considera implantaciones rápidas utilizando asignación por montículo.

En lógica matemática, las variables cuantificadas de alcance y sustitución limitados aparecen con el *Begriffsschrift* de Frege [1879]. Las sustituciones y el paso de parámetros han sido objeto de muchas discusiones tanto en el colectivo de la lógica matemática como en el de los lenguajes de programación. Church [1956, pág. 288] escribe: "Es especialmente difícil enunciar correctamente la regla de sustitución para variables funcionales", y relata el desarrollo de dicha regla para el cálculo proposicional. El cálculo lambda de Church [1941] se ha aplicado a los ambientes en los lenguajes de programación, por ejemplo en Landin [1964]. A partir de Landin [1964], a menudo se hace referencia a un par formado por una función y un enlace de acceso como a una *cerradura*.

Las estructuras de datos para tablas de símbolos y los algoritmos para buscarlas se estudian detalladamente en Knuth [1973b] y en Aho, Hopcroft y Ullman [1974, 1983]. La dispersión es analizada en Knuth [1973b] y Morris [1968b]. El artículo original que estudia la dispersión es Peterson [1957]. En McKeeman [1976] se pueden encontrar más estudios sobre las técnicas de organización de tablas de símbolos. El ejemplo 7.10 es obra de Bentley, Cleveland y Sethi [1985]. Reiss [1983] describe un generador de tablas de símbolos.

Algunos algoritmos de equivalencias han sido descritos por Arden, Galler y Graham [1961] y Galler y Fischer [1964]; en este libro se ha adoptado este último enfoque. La eficiencia de los algoritmos de equivalencia se estudia en Fischer [1972], Hopcroft y Ullman [1973] y Tarjan [1975].

CAPITULO 8

Generación de código intermedio

En el modelo de análisis y síntesis de un compilador, la etapa inicial traduce un programa fuente a una representación intermedia a partir de la cual la etapa final genera el código objeto. Los detalles del lenguaje objeto se confinan en la etapa final, si esto es posible. Aunque un programa fuente se puede traducir directamente al lenguaje objeto, algunas ventajas de utilizar una forma intermedia independiente de la máquina son:

1. Se facilita la redestinación; se puede crear un compilador para una máquina distinta uniendo una etapa final para la nueva máquina a una etapa inicial ya existente.

2. Se puede aplicar a la representación intermedia un optimador de código independiente de la máquina. Estos optimadores se estudian con detalle en el capítulo 10.

Este capítulo muestra cómo se pueden utilizar los métodos dirigidos por la sintaxis de los capítulos 2 y 5 para traducir a una forma intermedia construcciones de lenguajes de programación como declaraciones, asignaciones y proposiciones de flujo del control. Para simplificar, se supone que el programa fuente ya ha sido analizado sintácticamente y comprobado estáticamente como en la organización de la figura 8.1. La mayor parte de las definiciones dirigidas por la sintaxis de este capítulo se pueden implantar durante el análisis sintáctico ascendente o descendente utilizando las técnicas del capítulo 5, así que la generación de código intermedio se puede intercalar en el análisis sintáctico, si se desea.

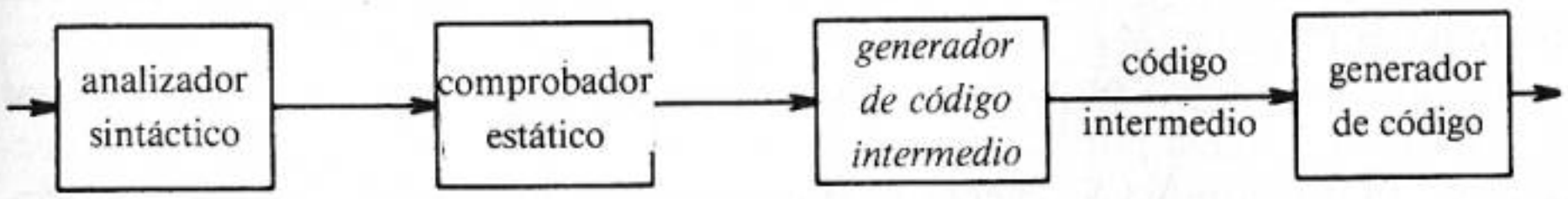


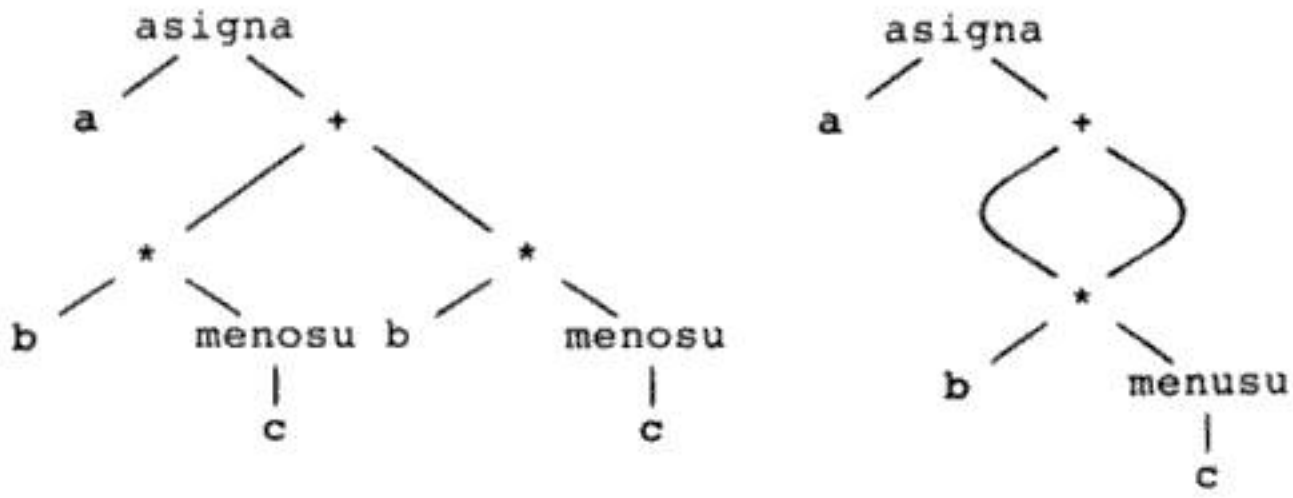
Fig. 8.1. Ubicación del generador de código intermedio.

8.1 LENGUAJES INTERMEDIOS

Los árboles sintácticos y la notación postfija, introducidos en las secciones 5.2 y 2.3, respectivamente, son dos clases de representaciones intermedias. En este capítulo se utilizará una tercera, llamada código de tres direcciones. Las reglas semánticas para generar código de tres direcciones a partir de construcciones de lenguajes de programación comunes son similares a las reglas para construir árboles sintácticos o para generar notación postfija.

Representaciones gráficas

Un árbol sintáctico describe la estructura jerárquica natural de un programa fuente. Un grafo dirigido acíclico (GDA) proporciona la misma información pero de una forma más compacta porque se identifican las subexpresiones comunes. En la figura 8.2 aparecen un árbol sintáctico y un GDA para la proposición de asignación `a := b * - c + b * - c`.



(a) árbol sintáctico (b) GDA

Fig. 8.2. Representaciones gráficas de `a := b * - c + b * - c`.

La representación postfija es una representación linealizada de un árbol sintáctico; es una lista de los nodos de un árbol en la que un nodo aparece inmediatamente después de sus hijos. La notación postfija para el árbol sintáctico de la figura 8.2(a) es

```
a b c menosu * b c menosu * + asigna        (8.1)
```

Las aristas de un árbol sintáctico no aparecen explícitamente en la notación postfija. Se pueden recuperar según el orden en que aparecen los nodos y el número de operandos que espera el operador de un nodo. La recuperación de las aristas es similar a la evaluación, mediante una pila, de una expresión en notación postfija. Véase la sección 2.8 para más detalles y la relación entre notación postfija y código para una máquina de pila.

Los árboles sintácticos para las proposiciones de asignación son producidos por la definición dirigida por la sintaxis de la figura 8.3; es una extensión de la sección 5.2. El no terminal *S* genera una proposición de asignación. Los dos operadores binarios + y * son ejemplos del conjunto completo de operadores de un lenguaje

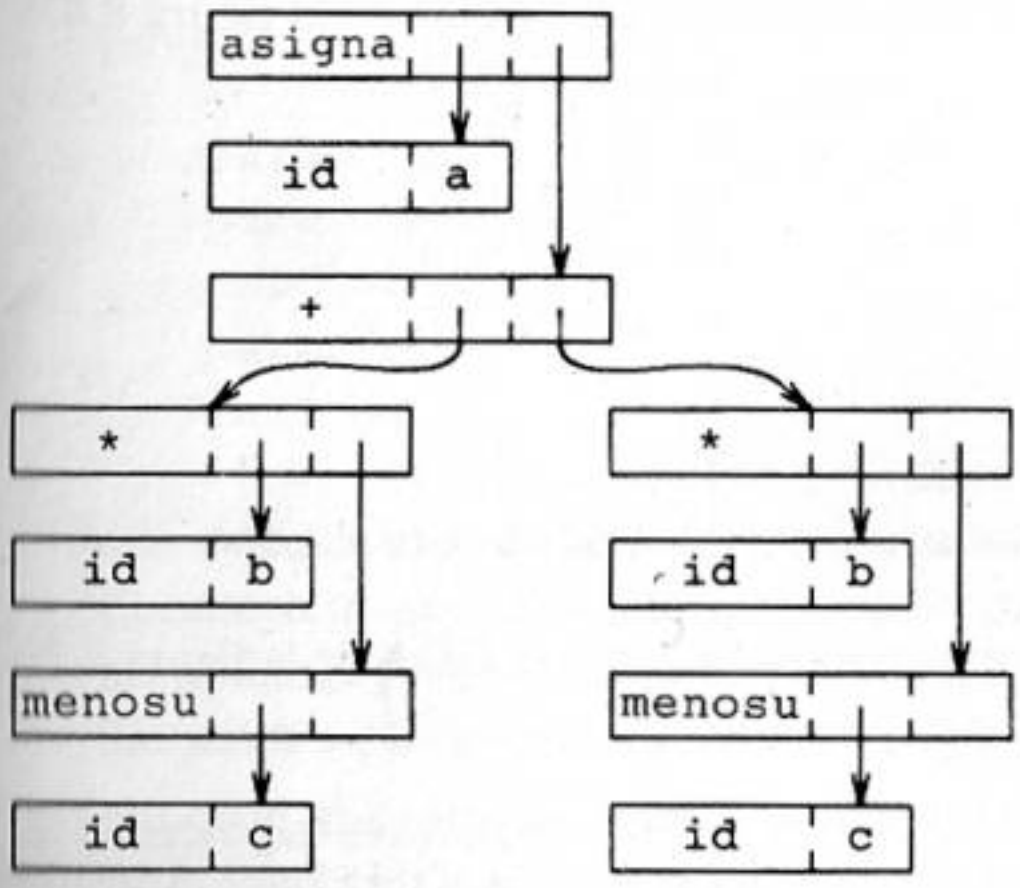
PRODUCCIÓN	REGLA SEMÁNTICA
$S \rightarrow$ **id** := E	$S.apn :=$ *haznodo* ('asigna', *hazhoja* (**id**, **id**.*lugar*), $E.apn$)
$E \rightarrow E_1 + E_2$	$E.apn :=$ *haznodo* ('+', $E_1.apn$, $E_2.apn$)
$E \rightarrow E_1 * E_2$	$E.apn :=$ *haznodo* ('*', $E_1.apn$, $E_2.apn$)
$E \rightarrow - E_1$	$E.apn :=$ *haznodoun* ('menosu', $E_1.apn$)
$E \rightarrow (E_1)$	$E.apn := E_1.apn$
$E \rightarrow$ **id**	$E.apn :=$ *hazhoja* (**id**, **id**.*lugar*)

Fig. 8.3. Definición dirigida por la sintaxis para producir árboles sintácticos para proposiciones de asignación.

típico. Las asociatividades y precedencias de los operadores son las habituales, aunque no se hayan incluido en la gramática. Esta definición construye el árbol de la figura 8.2(a) a partir de la entrada `a := b * - c + b * - c`.

Esta misma definición dirigida por la sintaxis producirá el GDA de la figura 8.2(b) si las funciones *haznodoun* (*op*, *hijo*) y *haznodo* (*op*, *izquierdo*, *derecho*) devuelven un apuntador a un nodo ya existente siempre que sea posible, en lugar de construir nuevos nodos. El componente léxico **id** tiene el atributo *lugar* que apunta a la entrada de la tabla de símbolos correspondiente al identificador. En la sección 8.3 se muestra cómo se puede encontrar una entrada de la tabla de símbolos a partir de un atributo **id**.*nombre*, que representa al lexema asociado con ese caso de **id**. Si el analizador léxico conserva todos los lexemas en una sola matriz de caracteres, entonces el atributo *nombre* puede ser el índice del primer carácter del lexema.

En la figura 8.4 aparecen dos representaciones del árbol sintáctico de la figura 8.2(a). Cada nodo se representa como un registro con un campo para su operador y campos adicionales para apuntadores a sus hijos. En la figura 8.4(b), los nodos se



(a)

0	id	b	
1	id	c	
2	menosu	1	
3	*	0	2
4	id	b	
5	id	c	
6	menosu	5	
7	*	4	6
8	+	3	7
9	id	a	
10	asigna	9	8
11	...		

(b)

Fig. 8.4. Dos representaciones del árbol sintáctico de la figura 8.2(a).

localizan en una matriz de registros y el índice o posición de los nodos sirve como apuntador al nodo. Todos los nodos del árbol sintáctico se pueden visitar siguiendo apuntadores, comenzando desde la raíz en la posición 10.

Código de tres direcciones

El código de tres direcciones es una secuencia de proposiciones de la forma general

```
x := y op z
```

donde x, y y z son nombres, constantes o variables temporales generadas por el compilador; *op* representa cualquier operador, como un operador aritmético de punto fijo o flotante, o un operador lógico sobre datos con valores booleanos. Obsérvese que no se permite ninguna expresión aritmética compuesta, pues sólo hay un operador en el lado derecho de una proposición. Por tanto, una expresión del lenguaje fuente como x+y*z se puede traducir en una secuencia

```
t1 := y * z
t2 := x + t1
```

donde t_1 y t_2 son nombres temporales generados por el compilador. Esta descomposición de expresiones aritméticas complejas y de proposiciones de flujo del control anidadas hace al código de tres direcciones deseable para la generación de código objeto y para la optimación. (Véanse Caps. 10 y 12.) El uso de nombres para los valores intermedios calculados por un programa permite que el código de tres direcciones se reorganice fácilmente —a diferencia de la notación postfija—.

El código de tres direcciones es una representación linealizada de un árbol sintáctico o un GDA en la que los nombres explícitos corresponden a los nodos interiores del grafo. El árbol sintáctico y el GDA de la figura 8.2 están representados por las secuencias de código de tres direcciones de la figura 8.5. Los nombres de las variables aparecen directamente en las proposiciones de tres direcciones, de modo que la figura 8.5(a) no tiene proposiciones correspondientes a las hojas de la figura 8.4.

```
t1 := - c
t2 := b * t1
t3 := - c
t4 := b * t3
t5 := t2 + t4
a  := t5
```

(a) Código para el árbol sintáctico

```
t1 := - c
t2 := b * t1
t5 := t2 + t2
a  := t5
```

(b) Código para el GDA.

Fig. 8.5. Código de tres direcciones correspondiente al árbol y al GDA de la figura 8.2.

La explicación del término "código de tres direcciones" es que cada proposición contiene generalmente tres direcciones, dos para los operandos y una para el resultado. En las aplicaciones del código de tres direcciones que se dan más adelante en esta sección, un nombre definido por el programador se sustituye por un apuntador a la entrada de la tabla de símbolos correspondiente a dicho nombre.

Tipos de proposiciones de tres direcciones

Las proposiciones de tres direcciones son análogas al código ensamblador. Las proposiciones pueden tener etiquetas simbólicas y existen proposiciones para el flujo del control. Una etiqueta simbólica representa el índice de una proposición de tres direcciones en la matriz que contiene el código intermedio. Los índices reales se pueden sustituir por las etiquetas, ya sea realizando una pasada independiente o mediante "relleno en retroceso", que se estudia en la sección 8.6.

A continuación se presentan las proposiciones de tres direcciones comunes que se utilizan en el resto de este libro:

1. Las proposiciones de asignación de la forma `x := y` *op* `z`, donde *op* es una operación binaria aritmética o lógica.

2. Las instrucciones de asignación de la forma `x :=` *op* `y`, donde *op* es una operación unaria. Las operaciones unarias principales incluyen el menos unario, la negación lógica, los operadores de desplazamiento y operadores de conversión que, por ejemplo, convierten un número de punto fijo en un número de punto flotante.

3. Las *proposiciones de copia* de la forma `x := y`, donde el valor de `y` se asigna a `x`.

4. El salto incondicional `goto E`. La proposición de tres direcciones con etiqueta `E` es la siguiente que va a ejecutarse.

5. Los saltos condicionales como `if x` *oprel* `y goto E`. Esta instrucción aplica un operador relacional (<, =, >=, etc.) a `x` e `y`, y a continuación ejecuta la proposición con etiqueta `E` si `x` pone *oprel* en relación con `y`. Si no, a continuación se ejecuta la proposición de tres direcciones que sigue a `if x` *oprel* `y goto E`, como en la secuencia habitual.

6. `param x y call p,` *n* para llamadas a procedimientos y `return y`, donde `y`, que representa un valor devuelto, es opcional. Su uso típico es como secuencia de proposiciones de tres direcciones

```
param x1
param x2
  . . .
param xn
call p, n
```

que se genera como parte de una llamada al procedimiento $p(x_1, x_2, \ldots, x_n)$. El entero *n*, que indica el número de parámetros reales en "`call p,`*n*" no es redundante porque las llamadas pueden estar anidadas. La implantación de las llamadas a procedimientos se esboza en la sección 8.7.

7. Las asignaciones con índices de la forma `x := y[i]` y `x[i] := y`. La primera asigna a `x` el valor de la posición en `i` unidades de memoria más allá de la posición `y`. La proposición `x[i] :=y` asigna al contenido de la posición en `i` unidades de memoria más allá de la posición `x` al valor de `y`. En ambas instrucciones, `x`, `y` e `i` se refieren a objetos de datos.

8. Las asignaciones de direcciones y apuntadores de la forma `x := &y`, `x := *y` y `*x := y`. La primera hace que el valor de `x` sea la dirección de `y`. Presumiblemente, `y` es un nombre, tal vez una variable temporal, que indica una expresión con un valor de lazo izquierdo como `A[i, j]`, y `x` es el nombre de un apuntador o una variable temporal. Es decir, el valor de lado derecho de `x` es el valor de lado izquierdo (posición) de un objeto. En la proposición `x := *y`, se supone que `y` es un apuntador o una variable temporal cuyo valor de lado derecho es una posición. El valor de lado derecho de `x` se iguala al contenido de dicha posición. Por último, `*x := y` hace que el valor de lado derecho del objeto apuntado por `x` sea igual al valor de lado derecho de `y`.

La elección de operadores permisibles es un aspecto importante en el diseño de una forma intermedia. El conjunto de operadores debe ser lo bastante rico como para implantar las operaciones del lenguaje fuente. Un conjunto de operadores pequeño es más fácil de implantar en una nueva máquina objeto. Sin embargo, un conjunto de instrucciones limitado puede obligar a la etapa inicial a generar largas secuencias de proposiciones para algunas operaciones del lenguaje fuente. En tal caso, el optimador y el generador de código tendrán que trabajar más si se desea producir buen código.

Traducción dirigida por la sintaxis a código de tres direcciones

Cuando se genera código de tres direcciones, se construyen nombres temporales para los nodos interiores de un árbol sintáctico. Se calculará el valor del no terminal E en el lado izquierdo de $E \rightarrow E_1 + E_2$ dentro de un nuevo temporal `t`. En general, el código de tres direcciones para **id** `:=` E consta de código para evaluar E en algún nombre temporal `t`, seguido de la asignación **id**.*lugar* `:= t`. Si una expresión consta de un solo identificador, por ejemplo `y`, entonces `y` mismo contiene el valor de la expresión. Por el momento, se crea un nuevo nombre cada vez que se necesite un temporal; en la sección 8.3 se proporcionan técnicas para reutilizar los nombres temporales.

La definición con atributos sintetizados de la figura 8.6 genera código de tres direcciones para proposiciones de asignación. Dada la entrada `a := b * - c + b * - c`, produce el código de la figura 8.5(a). El atributo sintetizado *S.código* representa el código de tres direcciones para la asignación S. El no terminal E tiene dos atributos:

1. *E.lugar*, que es el nombre que contendrá el valor de E, y
2. *E.código*, que es la secuencia de proposiciones de tres direcciones que evalúan E.

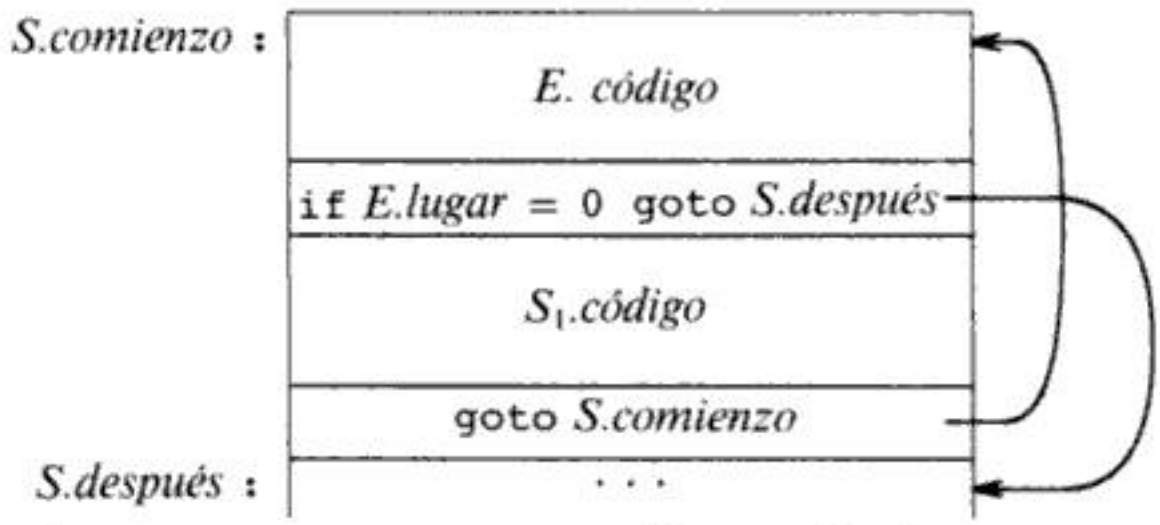
La función *tempnuevo* devuelve una secuencia de nombres distintos t_1, t_2, ..., en respuesta a sucesivas llamadas.

Por conveniencia, se utiliza la notación *gen* (x ':=' y '+' z) en la figura 8.6 para representar la proposición de tres direcciones $x := y + z$. Las expresiones que aparecen en lugar de variables como x, y y z se evalúan cuando se pasan a *gen*, y los operadores entrecomillados, como '+', se toman literalmente. En la práctica, las proposiciones de tres direcciones se pueden enviar a un archivo de salida, en lugar de construirlas en los atributos *código*.

PRODUCCIÓN	REGLA SEMÁNTICA
$S \rightarrow$ **id** := E	*S.código* := *E.código* ‖ *gen* (**id**.*lugar* ':=' *E.lugar*)
$E \rightarrow E_1 + E_2$	*E.lugar* := *tempnuevo*; *E.código* := E_1.*código* ‖ E_2.*código* ‖ *gen* (*E.lugar* ':=' E_1.*lugar* '+' E_2.*lugar*)
$E \rightarrow E_1 * E_2$	*E.lugar* := *tempnuevo*; *E.código* := E_1.*código* ‖ E_2.*código* ‖ *gen* (*E.lugar* ':=' E_1.*lugar* '*' E_2.*lugar*)
$E \rightarrow - E_1$	*E.lugar* := *tempnuevo*; *E.código* := E_1.*código* ‖ *gen* (*E.lugar* ':=' 'menosu' E_1.*lugar*)
$E \rightarrow (E_1)$	*E.lugar* := E_1.*lugar*; *E.código* := E_1.*código*
$E \rightarrow$ **id**	*E.lugar* := **id**.*lugar*; *E.código* := ''

Fig. 8.6. Definición dirigida por la sintaxis para producir código de tres direcciones para las asignaciones.

Se pueden añadir proposiciones de flujo del control al lenguaje de asignaciones de la figura 8.6 mediante producciones y reglas semánticas como las de las proposiciones **while** de la figura 8.7. En la figura, el código para $S \rightarrow$ **while** E **do** S_1 se genera utilizando los atributos nuevos *S.comienzo* y *S.después* para marcar la pri-



PRODUCCIÓN	REGLAS SEMÁNTICAS
$S \rightarrow$ **while** E **do** S_1	*S.comienzo* := *etiqnueva*; *S.después* := *etiqnueva*; *S.código* := *gen* (*S.comienzo* ':') ‖ *E.código* ‖ *gen* ('if' *E.lugar* '=' '0' 'goto' *S.después*) ‖ S_1.*código* ‖ *gen* ('goto' *S.comienzo*) ‖ *gen* (*S.después* ':')

Fig. 8.7. Reglas semánticas que generan código para una proposición **while**.

mera proposición en el código correspondiente a E y la proposición que sigue al código correspondiente a S, respectivamente. Estos atributos representan etiquetas creadas por la función *etiqnueva* que devuelve una etiqueta nueva cada vez que es llamada. Obsérvese que *S.después* se convierte en la etiqueta de la proposición que viene después del código correspondiente a la proposición **while**. Se supone que una expresión distinta de cero representa el valor *true*; es decir, cuando el valor de E se convierte en cero, el control abandona la proposición **while**.

Las expresiones que gobiernan el flujo del control pueden ser generalmente expresiones booleanas que contengan operadores relacionales y lógicos. Las reglas semánticas para las proposiciones **while** de la sección 8.6 difieren de las de la figura 8.7 para admitir flujo de control dentro de las expresiones booleanas.

Se puede obtener notación postfija adaptando las reglas semánticas de la figura 8.6 (o véase Fig. 2.5). La notación postfija para un identificador es el identificador mismo. Las reglas para las otras producciones concatenan sólo al operador después del código de los operandos. Por ejemplo, a la producción $E \rightarrow - E_1$ se asocia la regla semántica

$$E.código := E_1.código \parallel \text{'menosu'}$$

En general, la forma intermedia producida por las traducciones dirigidas por la sintaxis de este capítulo se puede cambiar realizando a las reglas semánticas modificaciones similares.

Implantaciones de proposiciones de tres direcciones

Una proposición de tres direcciones es una forma abstracta de código intermedio. En un compilador, estas proposiciones se pueden implantar como registros con campos para el operador y los operandos. Tres de dichas representaciones son cuádruplos, triples y triples indirectos.

Cuádruplos

Un cuádruplo es una estructura tipo registro con cuatro campos, que se llamarán *op*, *arg*1, *arg*2 y *resultado*. El campo *op* contiene un código interno para el operador. La proposición de tres direcciones `x := y op z` se representa poniendo `y` en *arg*1, `z` en *arg*2 y `x` en *resultado*. Las proposiciones con operadores unarios como `x := -y` o `x := y` no utilizan *arg*2. Los operadores como `param` no utilizan ni *arg*2 ni *resultado*. Los saltos condicionales e incondicionales ponen la etiqueta objeto en *resultado*. Los cuádruplos de la figura 8.8(a) corresponden a la asignación `a := b * -c +b * -c`. Se obtienen a partir del código de tres direcciones de la figura 8.5(a).

Los contenidos de los campos *arg*1, *arg*2 y *resultado* son generalmente apuntadores a las entradas de la tabla de símbolos correspondientes a los nombres representados por dichos campos. En ese caso, los nombres temporales se deben introducir en la tabla de símbolos conforme van siendo creados.

Triples

Para evitar introducir nombres temporales en la tabla de símbolos, se hará referencia a un valor temporal según la posición de la proposición que lo calcula. En ese

	op	*arg1*	*arg2*	*resultado*
(0)	menosu	c		t_1
(1)	*	b	t_1	t_2
(2)	menosu	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	:=	t_5		a

(a) Cuádruplos

	op	*arg1*	*arg2*
(0)	menosu	c	
(1)	*	b	(0)
(2)	menosu	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	asigna	a	(4)

(b) Triples

Fig. 8.8. Representaciones por medio de cuádruplos y triples de proposiciones de tres direcciones.

caso, las proposiciones de tres direcciones se pueden representar mediante registros con sólo tres campos: *op*, *arg*1 y *arg*2, como en la figura 8.8(b). Los campos *arg*1 y *arg*2, para los argumentos de *op*, son apuntadores a la tabla de símbolos (para nombres o constantes definidos por el programador) o apuntadores dentro de la estructura del triple (para valores temporales). Como se utilizan tres campos, este formato del código intermedio se conoce como triple[1]. Excepto el tratamiento de los nombres definidos por el programador, los triples corresponden a la representación de un árbol sintáctico o un GDA mediante una matriz de nodos, como en la figura 8.4.

Los números entre paréntesis representan apuntadores dentro de la estructura de los triples, en tanto que los apuntadores a la tabla de símbolos se representan mediante los nombres mismos. En la práctica, la información necesaria para interpretar las distintas clases de entradas en los campos *arg*1 y *arg*2 se puede codificar dentro del campo *op* o en algunos campos adicionales. Los triples de la figura 8.8(b) corresponden a los cuádruplos de la figura 8.8(a). Obsérvese que la proposición de copia a := t_5 se codifica en la representación con triples poniendo a dentro del campo *arg*1 y utilizando el operador asigna.

	op	*arg1*	*arg2*
(0)	[ ]=	x	i
(1)	asigna	(0)	y

(a) x[i] := y

	op	*arg1*	*arg2*
(0)	=[ ]	y	i
(1)	asigna	x	(0)

(b) x := y[i]

Fig. 8.9. Otras representaciones por medio de triples.

[1] A veces los triples se denominan "código de dos direcciones", y los "cuádruplos", "código de tres direcciones". Sin embargo, aquí se considerará el "código de tres direcciones" como una noción abstracta con varias implantaciones, siendo las principales los triples y los cuádruplos.

Una operación ternaria como `x[i] := y` exige dos entradas en la estructura de triples, como se muestra en la figura 8.9(a), mientras que `x := y[i]` se representa naturalmente como dos operaciones en la figura 8.9(b).

Triples indirectos

Otra implantación del código de tres direcciones que se ha considerado es la de hacer una lista de los apuntadores a triples, en lugar de hacer una lista de los triples mismos. Esta aplicación se llama naturalmente triples indirectos.

	proposición
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

	op	*arg1*	*arg2*
(14)	menosu	c	
(15)	*	b	(14)
(16)	menosu	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	asigna	a	(18)

Fig. 8.10. Representación por medio de triples indirectos de proposiciones de tres direcciones.

Por ejemplo, se utilizará una matriz *proposición* para listar los apuntadores a triples en el orden deseado. Entonces, los triples de la figura 8.8(b) se puede representar como en la figura 8.10.

Comparación de las representaciones: el uso de indirecciones

La diferencia entre triples y cuádruplos se puede considerar como la cuestión de cuánta indirección está presente en la representación. Cuando se produce finalmente código objeto, a cada nombre, temporal o definido por el programador, se le asignará alguna posición en la memoria para la ejecución. Esta posición se pondrá en la entrada de la tabla de símbolos correspondiente al dato. Utilizando una notación de cuádruplos, una proposición de tres direcciones que defina o utilice un temporal puede acceder de inmediato a la posición correspondiente a dicho temporal a través de la tabla de símbolos.

Hay una ventaja más importante de los cuádruplos en un compilador optimador, donde a menudo se trasladan las proposiciones. Utilizando la notación de cuádruplos, la tabla de símbolos interpone un grado adicional de indirección entre el cálculo de un valor y su uso. Si se traslada una proposición que calcule x, la proposición que utiliza x no necesita ninguna modificación. Sin embargo, en la notación de triples, trasladar una proposición que defina a un valor temporal exige que se modifiquen todas las referencias a esa proposición en las matrices *arg*1 y *arg*2. Este problema dificulta la utilización de los triples en un compilador optimador.

Los triples indirectos no presentan dicho problema. Una proposición se puede trasladar reordenando la lista *proposición*. Como los apuntadores o valores temporales se refieren a la matriz (o matrices) *op-arg*1-*arg*2, que no se han modificado, no hay que modificar ninguno de esos apuntadores. Por tanto, los triples indirectos se parecen mucho a los cuádruplos en lo relativo a su utilidad. Las dos notaciones exigen aproximadamente la misma cantidad de espacio y son igual de eficientes para reordenar el código. Como en el caso de los triples ordinarios, se debe posponer la asignación de memoria para aquellos temporales que lo necesiten hasta la fase de generación de código. Sin embargo, los triples indirectos pueden ahorrar algún espacio en comparación con los cuádruplos si el mismo valor temporal se utiliza más de una vez. La razón es que dos o más entradas en la matriz *proposición* pueden apuntar a la misma línea de la estructura *op-arg*1-*arg*2. Por ejemplo, se podrían combinar primero las líneas (14) y (16) de la figura 8.10 y combinar después las líneas (15) y (17).

8.2 DECLARACIONES

Conforme se examina la secuencia de declaraciones dentro de un procedimiento o un bloque, se puede distribuir la memoria para los nombres locales al procedimiento. Para cada nombre local se crea una entrada en la tabla de símbolos con información, por ejemplo, referente al tipo y la dirección relativa de la memoria correspondiente al nombre. La dirección relativa consta de un desplazamiento desde la base del área de datos estática o el campo para los datos locales en un registro de activación.

Cuando la etapa inicial genera direcciones, puede estar pensando en una máquina objeto. Supóngase que las direcciones de enteros consecutivos difieren en 4 en una máquina direccionable por bytes. Los cálculos de las direcciones generados por la etapa inicial pueden por tanto incluir multiplicaciones por 4. El conjunto de instrucciones de la máquina objeto también puede favorecer ciertas distribuciones de los objetos de datos, y por tanto sus direcciones. Aquí se pasa por alto la alineación de los objetos de datos; el ejemplo 7.3 muestra cómo los objetos de datos son alineados por dos compiladores.

Declaraciones dentro de un procedimiento

La sintaxis de lenguajes como C, Pascal y FORTRAN permite que todas las declaraciones en un solo procedimiento se procesen como un grupo. En este caso, una variable global, por ejemplo *desplazamiento*, puede contener la siguiente dirección relativa disponible.

En el esquema de traducción de la figura 8.11, el no terminal P genera una secuencia de declaraciones de la forma **id** : T. Antes de considerar la primera declaración, *desplazamiento* se pone en 0. Conforme aparece un nuevo nombre, ese nombre se introduce en la tabla de símbolos con un desplazamiento igual al valor en curso de *desplazamiento*, y *desplazamiento* aumenta en el ancho del objeto de datos indicado por dicho nombre.

El procedimiento *introduce* (*nombre*, *tipo*, *desplazamiento*) crea una entrada en la tabla de símbolos para *nombre*, le da el tipo *tipo* y la dirección relativa *desplazamiento* en su área de datos. Se utilizan los atributos sintetizados *tipo* y *ancho* para el no terminal T para indicar el tipo y ancho, o número de unidades de memoria empleadas por los objetos de ese tipo. El atributo *tipo* representa una expresión de tipos construida a partir de los tipos básicos *integer* y *real* aplicando los constructores de tipos *pointer* y *array*, como en la sección 6.1. Si las expresiones de tipos se representan por medio de grafos, entonces el atributo *tipo* puede ser un apuntador al nodo que representa una expresión de tipo.

En la figura 8.11, los enteros tienen un ancho de 4 y los reales tienen un ancho de 8. El ancho de una matriz se obtiene multiplicando el ancho de cada elemento por el número de elementos en la matriz[2]. Se supone que el ancho de cada apuntador es 4. En Pascal y C, se puede encontrar un apuntador antes de conocer el tipo del objeto apuntado por él (véase la exposición de tipos recursivos en la Sec. 6.3). La asignación de memoria para dichos tipos es más sencilla si todos los apuntadores tienen el mismo ancho.

$P \rightarrow$ D	{ *desplazamiento* := 0 }
$D \rightarrow D ; D$	
$D \rightarrow$ **id** : T	{ *introduce* (**id**.*nombre*, T.*tipo*, *desplazamiento*); *desplazamiento* := *desplazamiento* + T.*ancho* }
$T \rightarrow$ **integer**	{ T.*tipo* := *integer*; T.*ancho* := 4 }
$T \rightarrow$ **real**	{ T.*tipo* := *real*; T.*ancho* := 8 }
$T \rightarrow$ **array** [**núm**] **of** T_1	{ T.*tipo* := *array* (**núm**.*val*, T_1.*tipo*); T.*ancho* := **núm**.*val* × T_1.*ancho* }
$T \rightarrow \uparrow T_1$	{ T.*tipo* := *pointer* (T_1.*tipo*); T.*ancho* := 4 }

Fig. 8.11. Cálculo de los tipos y direcciones relativas de nombres declarados.

La inicialización de *desplazamiento* en el esquema de traducción de la figura 8.11 es más evidente si la primera producción aparece en una línea como:

$$P \rightarrow \{ \textit{desplazamiento} := 0 \}\ D \tag{8.2}$$

Se pueden utilizar los no terminales que generen ϵ, llamados no terminales marcadores en la sección 5.6, para reescribir producciones de modo que todas las accio-

---

[2] Para matrices cuyo límite inferior no sea 0, se simplifica el cálculo de direcciones para los elementos de la matriz si el desplazamiento introducido en la tabla de símbolos se ajusta, como se vio en la sección 8.3.

nes aparezcan en los extremos derechos. Con un no terminal marcador M, (8.2) se puede replantear como:

$$\begin{array}{ll} P \rightarrow M\ D & \\ M \rightarrow \epsilon & \{\ \mathit{desplazamiento} := 0\ \} \end{array}$$

Registro de la información sobre el ámbito

En un lenguaje con procedimientos anidados, se pueden asignar direcciones relativas a los nombres locales a cada procedimiento utilizando el enfoque de la sección 8.11. Cuando se encuentre un procedimiento anidado, se suspende temporalmente el proceso de las declaraciones del procedimiento abarcador. Este enfoque se ilustrará añadiendo reglas semánticas al siguiente lenguaje.

$$\begin{array}{l} P \rightarrow D \\ D \rightarrow D\ ;\ D \mid \textbf{id}\ :\ T \mid \textbf{proc id}\ ;\ D\ ;\ S \end{array} \qquad (8.3)$$

No se muestran las producciones para los no terminales S, que corresponde a las proposiciones, y T, que corresponde a los tipos, porque sólo se consideran las declaraciones. El no terminal T tiene los atributos sintetizados *tipo* y *ancho*, como en el esquema de traducción de la figura 8.11.

Para simplificar, supóngase que hay una tabla de símbolos distinta para cada procedimiento en el lenguaje (8.3). Una posible implantación de una tabla de símbolos es una lista enlazada de entradas correspondientes a los nombres. Si se desea se pueden usar implantaciones más elaboradas.

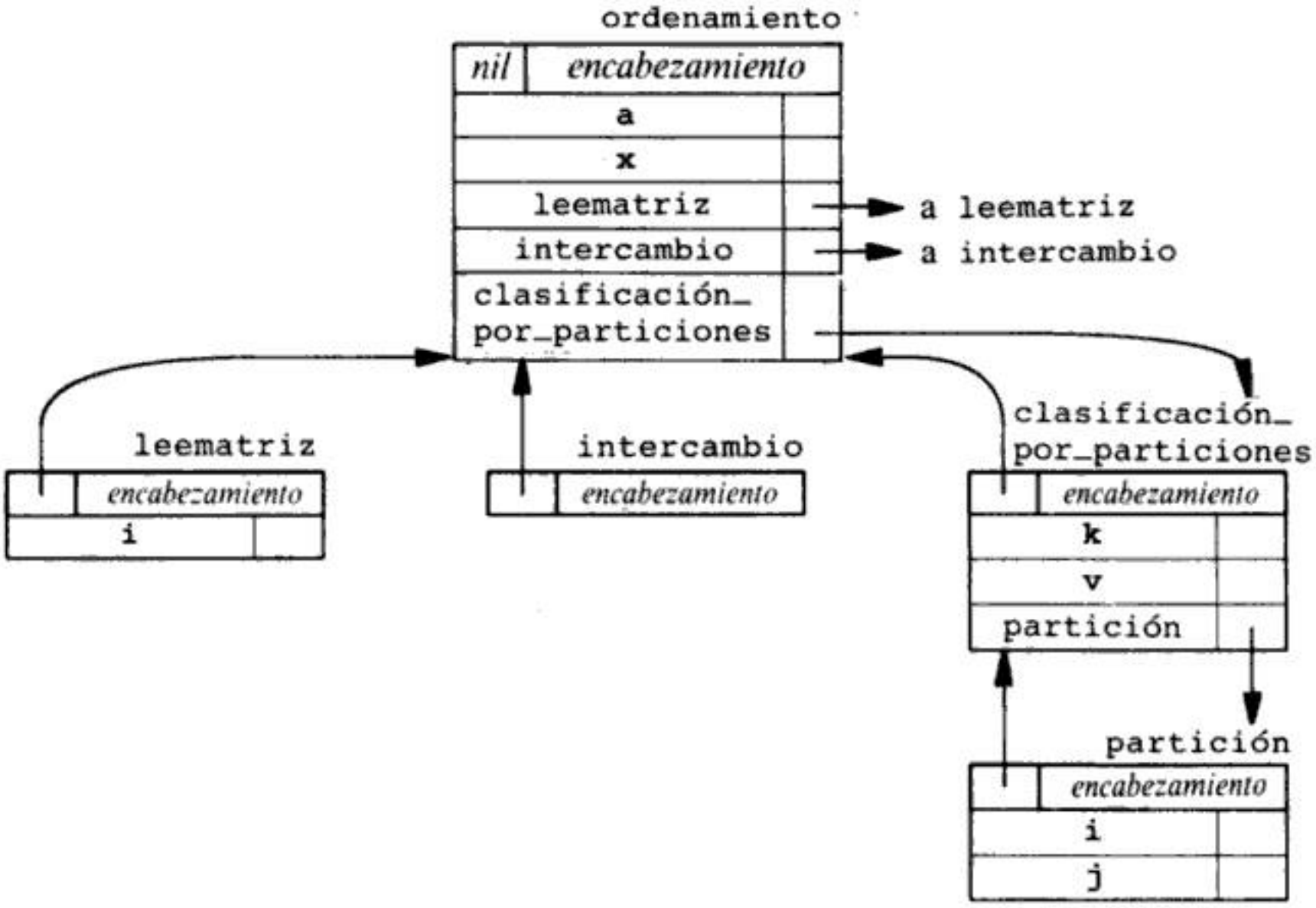


Fig. 8.12. Tablas de símbolos para procedimientos anidados.

Cuando se encuentra una declaración de procedimiento $D \rightarrow$ **proc id** D_1; S, se crea una nueva tabla de símbolos, y se crean las entradas para las declaraciones dentro de D_1 en la nueva tabla. La nueva tabla apunta de vuelta a la tabla de símbolos del procedimiento abarcador; el nombre representado por **id** mismo es local al procedimiento abarcador. El único cambio con respecto al tratamiento de las declaraciones de variables de la figura 8.11 es que se indica al procedimiento *introduce* en qué tabla de símbolos debe realizar una entrada.

Por ejemplo, en la figura 8.12 se muestran las tablas de símbolos para cinco procedimientos. La estructura de anidamiento de los procedimientos se puede deducir de los enlaces entre las tablas de símbolos; el programa es el de la figura 7.22. Las tablas de símbolos para los procedimientos `leematriz`, `intercambio` y `clasificación_por_particiones` apuntan de vuelta a la tabla correspondiente al procedimiento contenedor `ordenamiento`, formado por el programa completo. Como `participación` está declarado dentro de `clasificación_por_particiones`, su tabla apunta a la de `clasificación_por_particiones`.

Las reglas semánticas están definidas respecto a las siguientes operaciones:

1. *creatabla* (*previa*) crea una nueva tabla de símbolos y devuelve un apuntador a la nueva tabla. El argumento *previa* apunta a una tabla de símbolos creada previamente, se supone que la correspondiente al procedimiento abarcador. El apuntador *previa* se coloca en un encabezamiento para la nueva tabla de símbolos, junto con información adicional como la profundidad de anidamiento de un procedimiento. También se pueden numerar los procedimientos en el orden en que se declaran y guardar dicho número en el encabezamiento.

2. *introduce* (*tabla*, *nombre*, *tipo*, *desplazamiento*) crea una nueva entrada correspondiente a *nombre* en la tabla de símbolos apuntada por *tabla*. De nuevo, *introduce* coloca el tipo *tipo* y la dirección relativa *desplazamiento* en campos dentro de la entrada.

3. *añadeancho* (*tabla*, *ancho*) registra el ancho acumulado de todas las entradas de *tabla* en el encabezamiento asociado con esta tabla de símbolos.

4. *introduceproc* (*tabla*, *nombre*, *tablanueva*) crea una entrada nueva para el procedimiento *nombre* dentro de la tabla de símbolos apuntada por *tabla*. El argumento *tablanueva* apunta a la tabla de símbolos correspondiente a este procedimiento *nombre*.

El esquema de traducción de la figura 8.13 muestra cómo se pueden colocar los datos en una pasada, utilizando la pila *tblapn* para guardar apuntadores a las tablas de símbolos de los procedimientos abarcadores. Con las tablas de símbolos de la figura 8.13, *tblapn* contendrá apuntadores a las tablas correspondientes a `ordenamiento`, `clasificación_por_particiones` y `partición` cuando se consideren las declaraciones dentro de `partición`. El apuntador a la tabla de símbolos en curso se encuentra en el tope. La otra pila, *desplazamiento*, es la generalización natural a procedimientos anidados del atributo *desplazamiento* de la figura 8.11. El elemento del tope de *desplazamiento* es la siguiente dirección relativa disponible para un nombre local del procedimiento en curso.

Todas las acciones semánticas en los subárboles correspondientes a B y C en

$$A \rightarrow B\ C\ \{\ acción_A\ \}$$

se realizan antes de que ocurra $acción_A$ al final de la producción. Por tanto, la acción asociada con el marcador M asigna como valor inicial a la pila *tblapn* una tabla de símbolos para el ámbito más externo, creado por la operación *creatabla*(*nil*). La acción también introduce la dirección relativa 0 en la pila *desplazamiento*. El no terminal N desempeña un papel similar cuando aparece la declaración de un procedimiento. Su acción utiliza la operación *creatabla*(*tope*(*tblapn*)) para crear una nueva tabla de símbolos. Aquí, el argumento *tope*(*tblapn*) proporciona el alcance abarcador de la nueva tabla. Se introduce un apuntador a la nueva tabla por encima del apuntador correspondiente al ámbito abarcador. De nuevo, se introduce 0 en *desplazamiento*.

Para cada declaración de variable **id** : T, se crea una entrada para **id** en la tabla de símbolos en curso. Esta declaración no modifica la pila *tblapn*; el tope de la pila *desplazamiento* se incrementa en *T.ancho*. Cuando se produzca la acción del lado derecho de $D \rightarrow$ **proc id** ; $N\ D_1$, el ancho de todas las declaraciones generadas por D_1 estará en el tope de la pila *desplazamiento*; se registra utilizando *añadeancho*. Después sacan las pilas *tblapn* y *desplazamiento*, y se regresa para examinar las declaraciones en el procedimiento abarcador. Llegados a este punto, el nombre del procedimiento abarcado se introduce en la tabla de símbolos de su procedimiento abarcador.

$P \rightarrow M\ D$	{ *añadeancho* (*tope* (*tblapn*),*tope* (*desplazamiento*)); *saca* (*tblapn*); *saca* (*desplazamiento*) }
$M \rightarrow \epsilon$	{ *t* := *creatabla* (*nil*); *mete* (*t*,*tblapn*); *mete* (0, *desplazamiento*) }
$D \rightarrow D_1\ ;\ D_2$	
$D \rightarrow$ **proc id** ; $N\ D_1$; S	{ *t* := *tope* (*tblapn*); *añadeancho* (*t*, *tope* (*desplazamiento*)); *saca* (*tblapn*); *saca* (*desplazamiento*); *introduceproc* (*tope* (*tblapn*), **id**.*nombre*, *t*) }
$D \rightarrow$ **id** : T	{ *introduce* (*tope* (*tblapn*), **id**.*nombre*, *T.tipo*, *tope* (*desplazamiento*)); *tope* (*desplazamiento*) := *tope* (*desplazamiento* + *T.ancho*) }
$N \rightarrow \epsilon$	{ *t* := *creatabla* (*tope* (*tblapn*)); *mete* (*t*, *tblapn*); *mete* (0, *desplazamiento*) }

Fig. 8.13. Proceso de declaraciones en procedimientos anidados.

Nombres de campos dentro de registros

Las siguientes producciones permiten que el no terminal T genere registros además de tipos básicos, apuntadores y matrices:

$T \rightarrow$ **record** D **end**

Las acciones dentro del esquema de traducción de la figura 8.14 ponen de relieve la similitud entre la disposición de los registros como una construcción de un lenguaje y los registros de activación. Como las definiciones de procedimientos no afectan los cálculos de los anchos de la figura 8.13, se pasa por alto el hecho de que la producción anterior también permite que aparezcan definiciones de procedimientos dentro de los registros.

$T \rightarrow$ **record** L D **end**	{ $T.tipo := record\,(tope\,(tblapn))$;
	$T.ancho := tope\,(desplazamiento)$;
	$saca\,(tblapn)$; $saca\,(desplazamiento)$ }
$L \rightarrow \epsilon$	{ $t := creatabla\,(nil)$;
	$mete\,(t, tblapn)$; $mete\,(0, desplazamiento)$ }

Fig. 8.14. Creación de una tabla de símbolos para nombres de campo en un registro.

Después de que aparezca la palabra clave **record**, la acción asociada con el marcador L crea una nueva tabla de símbolos para los nombres de los campos. Se introduce un apuntador en esta tabla de símbolos en la pila *tblapn* y la dirección relativa 0 se introduce en la pila *desplazamiento*. La acción correspondiente a $D \rightarrow$ **id** : T de la figura 8.13 introduce por tanto la información sobre el nombre del campo **id** en la tabla de símbolos correspondiente al registro. Además, el tope de la pila *desplazamiento* contendrá el ancho de todos los objetos de datos dentro del registro después que se hayan examinado los campos. La acción que sigue a **end** en la figura 8.14 devuelve este ancho como el atributo sintetizado *T.ancho*. El tipo *T.tipo* se obtiene aplicando el constructor *record* al apuntador a la tabla de símbolos correspondiente a este registro. Este apuntador se utilizará en la siguiente sección para recuperar los nombres, tipos y anchos de los campos en el registro a partir de *T.tipo*.

8.3 PROPOSICIONES DE ASIGNACION

En esta sección las expresiones pueden ser de tipo entero, real, matriz y registro. Como parte de la traducción de las asignaciones a código de tres direcciones, se muestra cómo se pueden buscar los nombres en la tabla de símbolos y cómo se puede acceder a los elementos de matrices y registros.

Los nombres dentro de la tabla de símbolos

En la sección 8.1, se formaron proposiciones de tres direcciones utilizando los nombres mismos, entendiendo que los nombres representaban apuntadores a sus entradas en la tabla de símbolos. El esquema de traducción de la figura 8.15 muestra cómo

se pueden encontrar dichas entradas a la tabla de símbolos. El lexema correspondiente al nombre representado por **id** viene dado por el atributo **id**.*nombre*. La operación *busca*(**id**.*nombre*) comprueba si existe una entrada para este caso del nombre en la tabla de símbolos. Si así es, se devuelve un apuntador a la entrada; en caso contrario, *busca* devuelve *nil* para indicar que no se ha encontrado la entrada.

Las acciones semánticas de la figura 8.15 usan el procedimiento *emite* para emitir proposiciones de tres direcciones a un archivo de salida, en vez de crear atributos *código* para los no terminales, como en la figura 8.6. En la sección 2.3, se puede hacer la traducción emitiendo un archivo de salida si los atributos *código* de los no terminales de los lados izquierdos de las producciones se forman concatenando los atributos *código* de los no terminales de la derecha, en el mismo orden en que aparecen los no terminales en el lado derecho, tal vez con algunas cadenas adicionales en medio.

Reinterpretando la operación *busca* de la figura 8.15, se puede utilizar el esquema de traducción aunque la regla del anidamiento más cercano se refiera a los nombres no locales, como en Pascal. Más concretamente, supóngase que el contexto en el que aparece una asignación viene dado por la siguiente gramática.

$$
\begin{aligned}
&P \rightarrow M\ D\\
&M \rightarrow \epsilon\\
&D \rightarrow D\ ;\ D \mid \textbf{id}\ :\ T \mid \textbf{proc id}\ ;\ N\ D\ ;\ S\\
&N \rightarrow \epsilon
\end{aligned}
$$

El no terminal P se convierte en el nuevo símbolo inicial cuando estas producciones se añaden a las de la figura 8.15.

Para cada procedimiento generado por esta gramática, el esquema de traducción de la figura 8.13 establece una tabla de símbolos aparte. Cada una de estas tablas de

$S \rightarrow$ **id** := E { $p :=$ *busca* (**id**.*nombre*);
if $p \neq$ *nil* **then**
emite (p ':=' E.*lugar*)
else *error* }

$E \rightarrow E_1 + E_2$ { E.*lugar* := *tempnuevo*;
emite (E.*lugar* ':=' E_1.*lugar* '+' E_2.*lugar*) }

$E \rightarrow E_1 * E_2$ { E.*lugar* := *tempnuevo*;
emite (E.*lugar* ':=' E_1.*lugar* '*' E_2.*lugar*) }

$E \rightarrow - E_1$ { E.*lugar* := *tempnuevo*;
emite (E.*lugar* ':=' 'menosu' E_1.*lugar*) }

$E \rightarrow (E_1)$ { E.*lugar* := E_1.*lugar* }

$E \rightarrow$ **id** { $p :=$ *busca* (**id**.*nombre*);
if $p \neq$ *nil* **then**
E.*lugar* := p
else *error* }

Fig. 8.15. Esquema de traducción para producir códigos de tres direcciones para las asignaciones.

símbolos tiene un encabezamiento que contiene un apuntador a la tabla correspondiente al procedimiento abarcador. (Véase Fig. 8.12 como ejemplo.) Cuando se está examinando la proposición que forma el cuerpo de un procedimiento, en el tope de la pila *tblapn* aparece un apuntador a la tabla de símbolos del procedimiento. Este apuntador se introduce en la pila mediante acciones asociadas con el no terminal marcador N en el lado derecho de $D \rightarrow$ **proc id** ; $N\ D_1$; S.

Sean las producciones para el no terminal S las de la figura 8.15. Los nombres que aparezcan en una asignación generada por S deben haber sido declarados, ya sea dentro del procedimiento en el que aparece S, o en algún procedimiento abarcador. Cuando se aplica a *nombre*, la operación modificada *busca* comprueba primero si *nombre* aparece en la tabla de símbolos en curso, accesible a través de *tope* (*tblapn*). Si no, *busca* utiliza el apuntador del encabezado de la tabla para encontrar la tabla de símbolos correspondiente al procedimiento abarcador y buscar ahí el nombre. Si el nombre no se puede encontrar en ninguno de estos ámbitos, entonces *busca* devuelve *nil*.

Por ejemplo, supóngase que las tablas de símbolos son como en la figura 8.12 y que se está examinando una asignación en el cuerpo del procedimiento `partición`. La operación *busca*(`i`) encontrará una entrada en la tabla de símbolos de `partición`. Como `v` no está en esta tabla de símbolos, *busca*(`v`) utilizará el apuntador del encabezado de esta tabla de símbolos para continuar la búsqueda en la tabla de símbolos del procedimiento abarcador `clasificación_por_particiones`.

Reutilización de los nombres temporales

Hasta ahora se ha supuesto que *tempnuevo* genera un nombre temporal nuevo cada vez que se necesita un temporal. Resulta especialmente útil en compiladores optimadores crear un nombre distinto cada vez que se llame a *tempnuevo*, como ya se verá en el capítulo 10. Sin embargo, los temporales utilizados para guardar valores intermedios en los cálculos de expresiones tienden a obstruir la tabla de símbolos, y hay que asignar espacio para guardar sus valores.

Los nombres temporales se pueden reutilizar modificando *tempnuevo*. En el siguiente capítulo se explora un enfoque alternativo para empaquetar distintos temporales dentro de la misma posición durante la generación de código.

La mayor parte de los temporales que indican datos se genera durante la traducción dirigida por la sintaxis de las expresiones, mediante reglas como las de la figura 8.15. El código generado por las reglas para $E \rightarrow E_1 + E_2$ tiene la forma general:

evaluar E_1 en t_1
evaluar E_2 en t_2
$t := t_1 + t_2$

Según las reglas para el atributo sintetizado *E.lugar* se deduce que t_1 y t_2 no se utilizan en ninguna otra parte del programa. Las duraciones de estos temporales están anidadas como pares correspondientes de paréntesis equilibrados. De hecho, las duraciones de todos los temporales utilizados para la evaluación de E_2 están contenidas en la duración de t_1. Por tanto, es posible modificar *tempnuevo* para que utilice, como si fuera una pila, una pequeña matriz en un área de datos del procedimiento para guardar los temporales.

Supóngase para simplificar que sólo se están considerando enteros. Se lleva una cuenta *c*, inicializada con cero. Siempre que se utilice como operando un nombre temporal, hay que disminuir *c* en 1. Siempre que se genere un nuevo nombre temporal, hay que utilizar `$c` e incrementar *c* en 1. Obsérvese que la "pila" de temporales no se mete y se saca durante la ejecución, aunque el compilador hace que los almacenamientos y cargas de los valores temporales ocurran en el "tope".

Ejemplo 8.1. Considérese la asignación

```
x:= a * b + c * d - e * f
```

En la figura 8.16 se muestra la secuencia de proposiciones de tres direcciones que generarían las reglas semánticas de la figura 8.15 si se modificara *tempnuevo*. La figura también contiene una indicación del valor "en curso" de *c* después de la generación de cada proposición. Obsérvese que cuando se calcula `$0-$1`, *c* se disminuye a cero, de modo que `$0` está de nuevo disponible para guardar el resultado. □

A los temporales que se pueden asignar y/o utilizar más de una vez, por ejemplo, en una proposición condicional, no se les puede asignar nombres de la forma "primero en entrar-primero en salir" descrita anteriormente. Como suelen ser infrecuentes, a todos estos valores temporales se les pueden asignar nombres propios. Se produce el mismo problema que el de los temporales que se definen o utilizan más de una vez cuando se realiza la optimación de código, como combinar subexpresiones comunes o sacar un cálculo de un lazo (véase Cap. 10). Una estrategia razonable es crear un nombre nuevo siempre que se cree una definición adicional o uso de un temporal o que se traslade su cálculo.

PROPOSICIÓN	VALOR DE *c*
	0
`$0 := a * b`	1
`$1 := c * d`	2
`$0 := $0 + $1`	1
`$1 := e * f`	2
`$0 := $0 - $1`	1
`x := $0`	0

Fig. 8.16. Código de tres direcciones con valores temporales guardados en una pila.

Acceso a elementos de matrices

Se puede acceder rápidamente a los elementos de una matriz si se guardan en un bloque de posiciones consecutivas. Si el ancho de cada elemento de la matriz es *a*, entonces el *i*-ésimo elemento de la matriz A comienza en la posición

$$base + (i - inf) \times a \tag{8.4}$$

donde *inf* es el límite inferior de los subíndices y *base* es la dirección relativa de la

posición de memoria asignada a la matriz. Es decir, *base* es la dirección relativa de A[*inf*].

La expresión (8.4) se puede evaluar parcialmente durante la compilación si se reescribe como

$$i \times a + (base - inf \times a)$$

La subexpresión $c = base - inf \times a$ se puede evaluar cuando aparece la declaración de la matriz. Se supone que c se guarda en la entrada de la tabla de símbolos de A, así que la dirección relativa de A[i] se obtiene añadiendo simplemente $i \times a$ a c.

También se puede aplicar el precálculo durante la compilación a los cálculos de direcciones de elementos de matrices de multidimensionales. Una matriz bidimensional se almacena generalmente en una de dos formas, bien en *forma fila* (fila a fila) bien en *forma de columnas* (columna a columna). En la figura 8.17 se muestra la disposición de una matriz A de 2×3 en (a) forma por filas y (b) forma por columnas. FORTRAN utiliza la forma por columnas; Pascal utiliza la forma por filas, porque `A[i,j]` es equivalente a `A[i][j]`, y los elementos de cada matriz `A[i]` se guardan consecutivamente.

En el caso de una matriz bidimensional almacenada en forma por filas, la dirección relativa de A[i_1, i_2] se puede calcular mediante la fórmula

$$base + ((i_1 - inf_1) \times n_2 + i_2 - inf_2) \times a$$

donde inf_1 e inf_2 son los límites inferiores de los valores de i_1 e i_2 y n_2 es el número de valores que puede tomar i_2. Es decir, si sup_2 es el límite superior para el valor de i_2, entonces $n_2 = sup_2 - inf_2 + 1$. Suponiendo que i_1 e i_2 son los únicos valores que no se conocen durante la compilación, la expresión anterior se puede reescribir como

$$((i_1 \times n_2) + i_2) \times a + (base - ((inf_1 \times n_2) + inf_2) \times a) \tag{8.5}$$

El último término de esta expresión se puede determinar durante la compilación.

Se pueden ampliar las formas por filas y por columnas a muchas dimensiones. La generalización de la forma por filas es almacenar los elementos de manera que, cuando se examina un bloque de memoria, los subíndices situados en el extremo

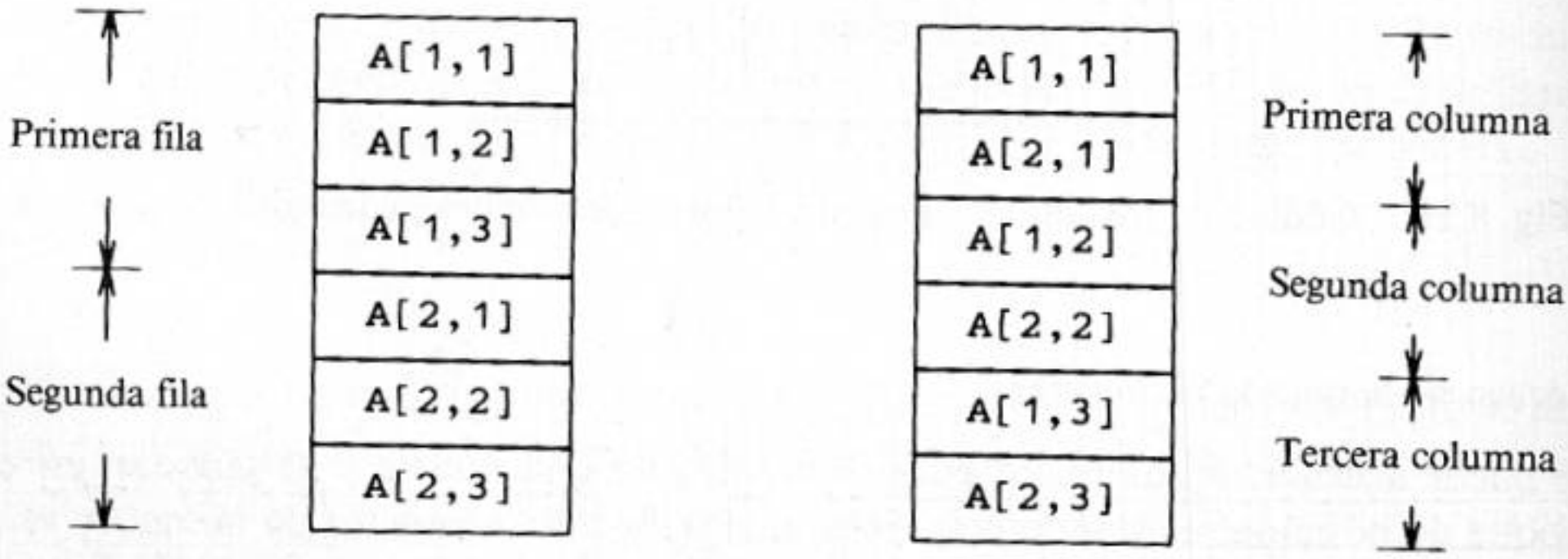


Fig. 8.17. Distribuciones de una matriz bidimensional.

derecho varían más rápidamente como los números en un taxímetro. La expresión (8.5) se aplica a la siguiente expresión para la dirección relativa de A $[i_1, i_2, \ldots, i_k]$

$$((\ldots((i_1 n_2 + i_2)n_3 + i_3)\ldots)n_k + i_k) \times a + base - ((\ldots((inf_1 n_2 + inf_2)n_3 + inf_3)\ldots)n_k + inf_k) \times a \quad (8.6)$$

Como para toda j, $n_j = sup_j - inf_j + 1$ se supone fija, el término de la segunda línea de (8.6) puede ser calculado por el compilador guardado con la entrada de la tabla de símbolos para A[3]. La forma por columnas se aplica a la disposición contraria, son los subíndices de la izquierda los que varían más rápidamente.

Algunos lenguajes permiten que los tamaños de las matrices se especifiquen dinámicamente cuando se llama a un procedimiento durante la ejecución. En la sección 7.3 se estudió la asignación de memoria para dichas matrices en una pila para la ejecución. Las fórmulas para acceder a los elementos de estas matrices son las mismas que para las matrices de tamaño fijo, pero los límites superior e inferior no se conocen en el momento de la compilación.

El problema principal de generar código para referencias de matrices es relacionar los cálculos de (8.6) con una gramática para referencias de matrices. Se pueden permitir las referencias a matrices en las asignaciones si se admite el no terminal L con las siguientes producciones donde **id** aparece en la figura 8.15:

$$L \rightarrow \textbf{id}\ [\ listaE\ ] \mid \textbf{id}$$
$$listaE \rightarrow listaE\ ,\ E \mid E$$

Para que los límites de las distintas dimensiones n_j de la matriz se encuentren disponibles cuando se agrupan las expresiones de índices dentro de una *listaE*, es útil reescribir las producciones de la forma:

$$L \rightarrow listaE\ ] \mid \textbf{id}$$
$$listaE \rightarrow listaE\ ,\ E \mid \textbf{id}\ [\ E$$

Es decir, el nombre de la matriz se asocia a la expresión de índice situada más a la izquierda en lugar de unirse a *listaE* cuando se forma una L. Estas producciones permiten que un apuntador a la entrada de la tabla de símbolos correspondiente al nombre de la matriz se pase como el atributo sintetizado *matriz* de *listaE*[4].

También se utiliza *listaE.ndim* para registrar el número de dimensiones (expresiones de índice) en la *listaE*. La función *límite* (*matriz*, *j*) devuelve n_j, el número de elementos de la j-ésima dimensión de la matriz cuya entrada en la tabla de símbolos viene apuntada por *matriz*. Para finalizar, *listaE.lugar* indica el nombre temporal que guarda un valor calculado según las expresiones de índice que aparecen en *listaE*.

---

[3] En C, se simula una matriz multidimensional definiendo matrices cuyos elementos sean matrices. Por ejemplo, supóngase que `x` es una matriz de matrices de enteros. El lenguaje permite entonces escribir tanto `x[i]` como `x[i][j]`, y los anchos de estas expresiones son distintos. Sin embargo, el límite inferior de todas las matrices es 0, así que el término de la segunda línea de (8.6) se limita a *base* en cada caso.

[4] La transformación es similar a la mencionada al final de la sección 5.6 para eliminar atributos heredados. Aquí también se podría haber resuelto el problema con atributos heredados.

Una *listaE* que produce los primeros m índices de referencia a una matriz k-dimensional A$[i_1, i_2, \ldots, i_k]$ generará código de tres direcciones para calcular

$$(\ldots((i_1 n_2 + i_2)n_3 + i_3)\ldots)n_m + i_m \tag{8.7}$$

utilizando la recurrencia

$$\begin{aligned} e_1 &= i_1 \\ e_m &= e_{m-1} \times n_m + i_m \end{aligned} \tag{8.8}$$

Por tanto, cuando $m = k$, basta con una multiplicación por el ancho a para calcular el término de la primera línea de (8.6). Obsérvese que aquí, las i_j pueden ser realmente valores de expresiones, y el código para evaluar estas expresiones estará entremezclado con el código para calcular (8.7).

Un valor de lado izquierdo I tendrá dos atributos. *I.lugar* e *I.desplazamiento*. En caso de que I sea un nombre simple, *I.lugar* será un apuntador a la entrada de la tabla de símbolos para dicho nombre, e *I.desplazamiento* será **null**, indicando que el valor de lado izquierdo es un nombre simple en lugar de ser una referencia a una matriz. El no terminal E tiene la misma traducción *E.lugar*, con el mismo significado que en la figura 8.15.

El esquema de traducción para acceder a elementos de matrices

Se añadirán acciones semánticas a la gramática:

(1) $S \rightarrow L := E$
(2) $E \rightarrow E + E$
(3) $E \rightarrow (E)$
(4) $E \rightarrow L$
(5) $L \rightarrow listaE\ ]$
(6) $L \rightarrow$ **id**
(7) $listaE \rightarrow listaE\ ,\ E$
(8) $listaE \rightarrow$ **id** $[\ E$

Como en el caso de expresiones sin referencias a matrices, el mismo código de tres direcciones es producido por el procedimiento *emite* invocado en las acciones semánticas.

Se genera una asignación normal si L es un nombre simple; en caso contrario, se genera una asignación indizada dentro de la posición indicada por L:

(1) $S \rightarrow L := E$ { **if** *L.desplazamiento* = **null then** /* L es un **id** simple */
emite (*L.lugar* ' := ' *E.lugar*);
else
emite (*L.lugar* '[' *L.desplazamiento* '] ' ' := ' *E.lugar*) }

El código para las expresiones aritméticas es exactamente el mismo que el de la figura 8.15:

(2) $E \rightarrow E_1 + E_2$ { *E.lugar* := *tempnuevo*;
emite (*E.lugar* ' := ' E_1.*lugar* ' + ' E_2.*lugar*) }

(3) $E \rightarrow (E_1)$ { *E.lugar* := E_1.*lugar* }

Cuando una referencia a matriz L se reduce a E, se desea el valor de lado derecho de L. Por tanto, se utiliza la indización para obtener el contenido de la localidad *L.lugar* [*L.desplamiento*]:

(4) $E \rightarrow L$

```
{ if L.desplazamiento = null then /* L es un id simple */
      E.lugar := L.lugar
  else begin
      E.lugar := tempnuevo;
      emite (E.lugar ':=' L.lugar'['L.desplazamiento']')
  end }
```

A continuación, *L.desplazamiento* es un temporal nuevo que representa el primer término de (8.6); la función *ancho* (*listaE.matriz*) devuelve *a* en (8.6). *L.lugar* representa el segundo término de (8.6), que es devuelto por la función *c* (*listaE.matriz*).

(5) $L \rightarrow lista\,[E]$

```
{ L.lugar := tempnuevo;
  L.desplazmiento := tempnuevo;
  emite (L.lugar ':=' c (listaE.matriz));
  emite (L.desplazamiento ':=' listaE.lugar
        '*' ancho (listaE.matriz)) }
```

Un desplazamiento nulo indica un nombre simple.

(6) $L \rightarrow$ **id**

```
{ L.lugar := id.lugar;
  L.desplazamiento := null }
```

Cuando se observa la siguiente expresión, se aplica la recurrencia (8.8). En la siguiente acción, $listaE_1.lugar$ corresponde a e_{m-1} en (8.8) y *listaE.lugar* a e_m. Obsérvese que si $listaE_1$ tiene $m - 1$ componentes, entonces *listaE* del lado izquierdo de la producción tiene m componentes.

(7) $lista\ E \rightarrow listaE_1\ ,\ E$

```
{ t := tempnuevo;
  m := listaE1.ndim + 1;
  emite (t ':=' listaE1.lugar '*' límite (listaE1.matriz, m));
  emite (t ':=' E. lugar);
  listaE.matriz := listaE1.matriz;
  listaE.lugar := t;
  listaE.ndim := m }
```

E.lugar contiene el valor de la expresión E como el valor de (8.7) para $m = 1$.

(8) $listaE \rightarrow$ **id** [E

```
{ listaE.matriz := id.lugar;
  listaE.lugar := E.lugar;
  listaE.ndim := 1 }
```

Ejemplo 8.2. Sea A una matriz de 10×20 con $inf_1 = inf_2 = 1$. Por tanto, $n_1 = 10$ y $n_2 = 20$. Se asume que a es 4. En la figura 8.18 se muestra un árbol de análisis sin-

táctico con anotaciones para la asignación `x:=A[y,z]`. La asignación se traduce a la siguiente secuencia de proposiciones de tres direcciones:

```
t1 := y * 20
t1 := t1 + z
t2 := c              /* constante c = base_A - 84 */
t3 := 4 * t1
t4 := t2 [t3]
x := t4
```

Para cada variable, se ha utilizado su nombre en lugar de **id**.*lugar*.

Conversiones de tipos dentro de asignaciones

En la práctica, hay muchos tipos diferentes de variables y constantes, así que el compilador debe rechazar algunas operaciones de tipos mixtos o bien generar las instrucciones de coerción (conversión de tipos) apropiadas.

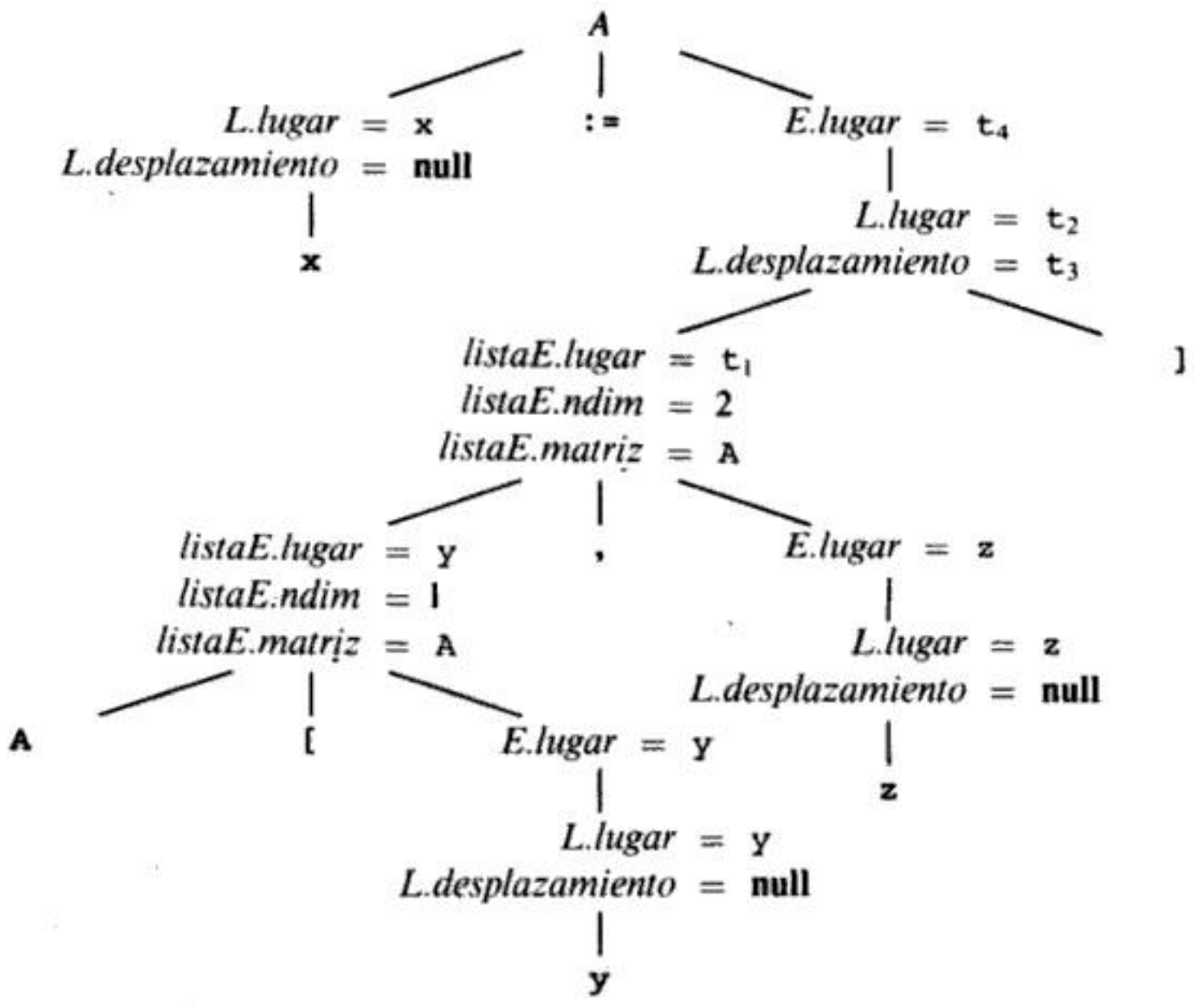


Fig. 8.18. Arbol de análisis sintáctico con anotaciones para `x := A[y, z]`.

Considérese la gramática para las proposiciones de asignación anterior, pero supóngase que hay dos tipos —real y entero, y los enteros se convierten en reales cuando sea necesario.

```
E.lugar := tempnuevo;
if E1.tipo = integer and E2.tipo = integer then begin
    emite(E.lugar ':=' E1.lugar '+ent' E2.lugar);
    E.tipo := integer
end
else if E1.tipo = real and E2.tipo = real then begin
    emite(E.lugar ':=' E1.lugar '+real' E2.lugar);
    E.tipo := real
end
else if E1.tipo = integer and E2.tipo = real then begin
    u := tempnuevo;
    emite(u ':=' 'entareal' E1.lugar);
    emite(E.lugar ':=' u '+real' E2.lugar);
    E.tipo := real
end
else if E1.tipo = real and E2.tipo = integer then begin
    u := tempnuevo;
    emite(u ':=' 'entareal' E2.lugar);
    emite(E.lugar ':=' E1.lugar '+real' u);
    E.tipo := real
end
else
    E.tipo := error tipo;
```

Fig. 8.19. Acción semántica para $E \rightarrow E_1 + E_2$.

Se introduce otro atributo, *E.tipo*, cuyo valor es *real* o *integer*. La regla semántica para *E.tipo* asociada con la producción $E \rightarrow E_1 + E_2$ es:

```
E → E1 + E2    { E.tipo :=
                    if E1.tipo = integer and
                       E2.tipo = integer then integer
                    else real }
```

Esta regla está presente en la sección 6.4; sin embargo, aquí y en todo este capítulo se omiten las comprobaciones de errores de tipo; en el capítulo 6 hay un análisis sobre la comprobación de tipos.

Se debe modificar toda la regla semántica para $E \rightarrow E + E$ y la mayor parte de las otras producciones para generar, cuando sea necesario, proposiciones de tres direcciones de la forma `x := entareal y` cuyo efecto es convertir el entero `y` en un real de igual valor, llamado `x`. También se debe incluir junto con el código de operador una indicación de si se piensa en aritmética de punto fijo o flotante. En la figura 8.19 se lista la acción semántica completa para una producción de la forma $E \rightarrow E_1 + E_2$.

Por ejemplo, para la entrada

```
x := y + i * j
```

suponiendo que x e y tienen tipo *real* y que i y j tienen tipo *integer*, la salida sería como

```
t1 := i int* j
t3 := entareal t1
t2 := y real+ t3
 x := t2
```

La acción semántica de la figura 8.19 utiliza dos atributos, *E.lugar* y *E.tipo* para el no terminal *E*. Conforme se incrementa el número de tipos objeto de conversión, crece cuadráticamente el número de casos (o peor, si hay operadores con más de dos argumentos). Por tanto, si hay un gran número de tipos, es importante la organización cuidadosa de las acciones semánticas.

Acceso a campos dentro de registros

El compilador debe registrar tanto los tipos como las direcciones relativas de los campos de un registro. Una ventaja de conservar esta información en las entradas de la tabla de símbolos correspondientes a los nombres de los campos es que la rutina para buscar nombres en la tabla de símbolos también puede utilizarse para buscar nombres de campos. Teniendo en cuenta esto, con las acciones semánticas de la figura 8.14 de la última sección, se creó en la figura 8.14 una tabla de símbolos diferente para cada tipo. Si *t* es un apuntador a la tabla de símbolos para un tipo registro, entonces el tipo *record*(*t*) que se forma aplicando el constructor *record* al apuntador fue devuelto como *T.tipo*.

Se utiliza la expresión

```
p↑.info + 1
```

para ilustrar cómo se puede extraer un apuntador a la tabla de símbolos de un atributo *E.tipo*. Según las operaciones, en estas expresiones se concluye que p debe ser un apuntador a un registro con un nombre de campo `info` cuyo tipo es aritmético. Si los tipos se construyen como en las figuras 8.13 y 8.14, el tipo de p debe venir dado por una expresión de tipo

pointer (*record*(*t*))

El tipo de p↑ es entonces *record*(*t*), del cual se puede extraer *t*. El nombre de campo `info` se busca en la tabla de símbolos apuntada por *t*.

8.4 EXPRESIONES BOOLEANAS

En los lenguajes de programación, las expresiones booleanas tienen dos propósitos principales. Se utilizan para calcular valores lógicos, pero sobre todo como expresiones condicionales en proposiciones que alteran el flujo del control, como las proposiciones **if-then**, **if-then-else** o **while-do**.

Las expresiones booleanas se componen de los operadores booleanos (**and**, **or** y **not**) aplicados a elementos que son variables booleanas o expresiones relacionales. A su vez, las expresiones relacionales son de la forma E_1 **oprel** E_2, donde E_1 y E_2 son expresiones aritméticas. Algunos lenguajes, como PL/I, permiten expresiones más generales, donde se pueden aplicar operadores booleanos, aritméticos y relacionales a expresiones de cualquier tipo, sin diferenciar valores booleanos de aritméticos; si es necesario se realiza una coerciòn. En esta sección se consideran las expresiones booleanas generadas por la siguiente gramática:

$$E \rightarrow E \textbf{ or } E \mid E \textbf{ and } E \mid \textbf{not } E \mid (E) \mid \textbf{id oprel id} \mid \textbf{true} \mid \textbf{false}$$

Se utiliza el atributo *op* para determinar cuál de los operadores de comparación $<$, $\leq$, $=$, $\neq$, $>$, o $\geq$ está representado por **oprel**. Como de costumbre, se supone que **or** y **and** son asociativos por la izquierda y que **or** tiene la precedencia menor, después **and** y luego **not**.

Métodos para traducir expresiones booleanas

Hay dos métodos principales para representar los valores de una expresión booleana. El primer método consiste en codificar numéricamente los valores **true** (verdadero) y **false** (falso) y evaluar una expresión booleana igual que una expresión aritmética. A menudo se utiliza `1` para indicar **true** y `0` para indicar **false**, aunque son posibles muchas otras codificaciones. Por ejemplo, se podría dejar que cualquier cantidad distinta de cero indicara **true** y que cero indicara **false**, o que cualquier cantidad no negativa indicara **true** y cualquier número negativo indicara **false**.

El segundo método para implantar expresiones booleanas es mediante flujo del control, es decir, representando el valor de una expresión booleana mediante una posición alcanzada en un programa. Este método es especialmente adecuado para la implantación de expresiones booleanas en proposiciones de flujo de control, como las proposiciones **if-then** y **while-do.** Por ejemplo, dada la expresión E_1 **or** E_2, si se determina que E_1 es verdadera, entonces se puede concluir que toda la expresión es verdadera sin tener que evaluar E_2.

La semántica de los lenguajes de programación determina si se deben evaluar todas las partes de una expresión booleana. Si la definición del lenguaje permite (o exige) que queden sin evaluar partes de una expresión booleana, entonces el compilador puede optimar la evaluación de las expresiones booleanas calculando únicamente lo necesario de una expresión como para determinar su valor. Por tanto, en una expresión como E_1 **or** E_2, ni E_1 ni E_2 se evalúan necesariamente por completo. Si E_1 o E_2 es una expresión con efectos secundarios (por ejemplo, que contiene una función que cambia una variable global), entonces se puede obtener una respuesta imprevista.

Ninguno de los métodos anteriores es uniformemente superior al otro. Por ejemplo, el compilador optimador BLISS/11 (Wulf y colaboradores, 1975), entre otros, elige el método apropiado para cada expresión individualmente. Esta sección considera ambos métodos para la traducción de expresiones booleanas a código de tres direcciones.

Representación numérica

Primero se considerará la implantación de expresiones booleanas utilizando `1` para indicar el valor **true** y `0` para indicar **false**. Las expresiones se evaluarán completamente, de izquierda a derecha, de manera similar a las expresiones aritméticas. Por ejemplo, la traducción de

```
a or b and not c
```

es la secuencia de tres direcciones

```
t1 := not c
t2 := b and t1
t3 := a or t2
```

Una expresión relacional como `a<b` es equivalente a la proposición condicional `if a < b then 1 else 0`, que se puede traducir a la secuencia de código de tres direcciones (de nuevo, se comienzan arbitrariamente a numerar las proposiciones por el 100):

```
100:  if a < b goto 103
101:  t := 0
102:  goto 104
103:  t := 1
104:
```

En la figura 8.20 se muestra un esquema de traducción para producir código de tres direcciones para las expresiones booleanas. En este esquema, se supone que *emite* y

$E \rightarrow E_1$ **or** E_2	{ *E.lugar* := *tempnuevo*; *emite* (*E.lugar* ':=' E_1.*lugar* 'or' E_2.*lugar*) }
$E \rightarrow E_1$ **and** E_2	{ *E.lugar* := *tempnuevo*; *emite* (*E.lugar* ':=' E_1.*lugar* 'and' E_2.*lugar*) }
$E \rightarrow$ **not** E_1	{ *E.lugar* := *tempnuevo*; *emite* (*E.lugar* ':=' 'not' E_1*lugar*) }
$E \rightarrow$ (E_1)	{ *E.lugar* := E_1.*lugar* }
$E \rightarrow$ **id**$_1$ **oprel** **id**$_2$	{ *E.lugar* := *tempnuevo*; *emite* ('if' **id**$_1$.*lugar* **oprel**.*op* **id**$_2$.*lugar* 'goto' *sigte prop*+3); *emite* (*E.lugar* ':=' '0'); *emite* ('**goto**' *sigte prop*+2); *emite* (*E.lugar* ':=' '1') }
$E \rightarrow$ **true**	{ *E.lugar* := *tempnuevo*; *emite* (*E.lugar* ':=' '1') }
$E \rightarrow$ **false**	{ *E.lugar* := *tempnuevo*; *emite* (*E.lugar* ':=' '0') }

Fig. 8.20. Esquema de traducción utilizando una representación numérica para los valores booleanos.

coloca proposiciones de tres direcciones en un archivo de salida en el formato correcto, que *sigteprop* da el índice de la siguiente proposición de tres direcciones en la secuencia de salida, y que *emite* incrementa *sigteprop* después de producir cada proposición de tres direcciones.

Ejemplo 8.3. El esquema de la figura 8.20 generaría el código de tres direcciones de la figura 8.21 para la expresión `a < b or c < d and e < f`. □

Código en cortocircuito

También se puede traducir una expresión booleana a código de tres direcciones sin generar código para ninguno de los operadores booleanos y sin que haya que evaluar necesariamente la expresión completa. Este estilo de evaluación a veces se denomina código "en cortocircuito" o código "saltado". Se pueden evaluar las expresiones booleanas sin generar código para los operadores booleanos **and**, **or** y **not** si se representa el valor de una expresión mediante una posición en la secuencia de código. Por ejemplo, en la figura 8.21, se puede saber qué valor tendrá t_1 por el hecho de si se alcanza la proposición 101 o la 103, así que el valor de t_1 es redundante. Para muchas expresiones booleanas, se puede determinar el valor de la expresión sin tener que evaluarla completamente.

```
100:  if a < b goto 103
101:  t1 := 0
102:  goto 104
103:  t1 := 1
104:  if c < d goto 107
105:  t2 := 0
106:  goto 108
107:  t2 := 1
108:  if e < f goto 111
109:  t3 := 0
110:  goto 112
111:  t3 := 1
112:  t4 := t2 and t3
113:  t5 := t1 or t4
```

Fig. 8.21. Traducción de `a<b or c<d and e<f`.

Proposiciones de flujo del control

Ahora se considera la traducción de expresiones booleanas dentro del código de tres direcciones en el contexto de proposiciones **if-then**, **if-then-else** y **while-do** como las generadas por la siguiente gramática:

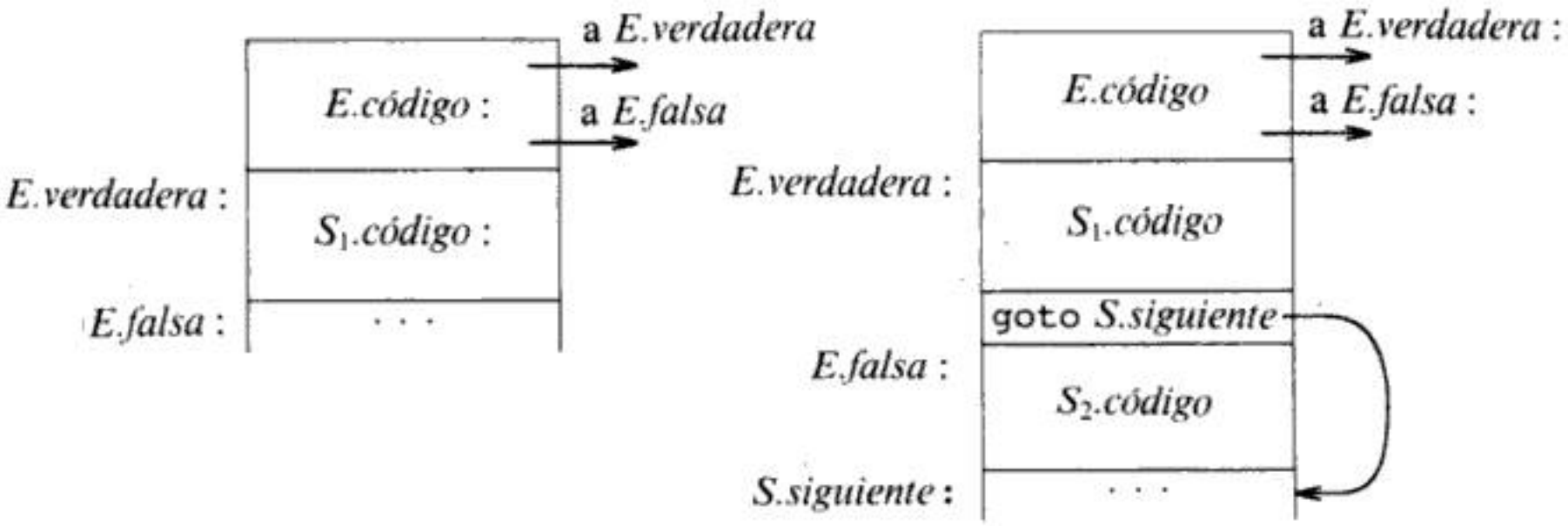
$$
\begin{aligned}
S \rightarrow\ & \textbf{if } E \textbf{ then } S_1 \\
\mid\ & \textbf{if } E \textbf{ then } S_1 \textbf{ else } S_2 \\
\mid\ & \textbf{while } E \textbf{ do } S_1
\end{aligned}
$$

En cada una de estas producciones, E es la expresión booleana que debe traducirse. En la traducción, se supone que una proposición de tres direcciones puede etiquetarse con símbolos y que la función *etiqnueva* devuelve una nueva etiqueta simbólica cada vez que es llamada.

Con una expresión booleana E se asocian dos etiquetas: $E.verdadera$, la etiqueta a la que fluye el control si E es verdadera, y $E.falsa$, la etiqueta a la que fluye el control si E es falsa. Las reglas semánticas para traducir una proposición S de flujo

de control permiten que el control fluya desde la traducción *S.código* a la instrucción de tres direcciones situada inmediatamente después de *S.código*. En algunos casos, la instrucción inmeditamente después de *S.código* es un salto a una etiqueta L. Se evita un salto a un salto a L desde dentro de *S.código* utilizando el atributo heredado *S.siguiente*. El valor de *S.siguiente* es una etiqueta que se asocia a la primera instrucción de tres direcciones que se ejecuta después del código correspondiente a S[5]. No se muestra la inicialización de *S.siguiente*.

Al traducir la proposición **if-then**, $S \rightarrow$ **if** E **then** S_1, se crea una nueva etiqueta *E.verdadera* y se asocia a la primera instrucción de tres direcciones generada para la proposición S_1, como en la figura 8.22(a). En la figura 8.23 aparece una definición dirigida por la sintaxis. El código correspondiente a E genera un salto a *E.verdadera* si E es verdadera y un salto a *S.siguiente* si E es falsa. Por tanto, se asocia *E.falsa* a *S.siguiente*.



(a) **if-then**

(b) **if-then-else**

(c) **while-do**

Fig. 8.22. Código para las proposiciones **if-then**, **if-then-else** y **while-do**.

Al traducir la proposición **if-then-else**, $S \rightarrow$ **if** E **then** S_1 **else** S_2, el código correspondiente a la expresión booleana E salta afuera de él a la primera instrucción del código para S_1 si E es verdadera, y a la primera instrucción del código para S_2 si E es falsa, como se ilustra en la figura 8.22(b). Como la proposición **if-then**, un atri-

[5] Si se implanta literalmente el enfoque de heredar una etiqueta *S.siguiente* puede conducir a una proliferación de etiquetas. El método de relleno de retroceso de la sección 8.6 crea etiquetas sólo cuando son necesarias.

buto heredado *S.siguiente* proporciona la etiqueta de la instrucción de tres direcciones que debe ejecutarse después de ejecutar el código de S. Aparece un `goto` *S.siguiente* explícito después del código de S_1, pero no después de S_2. Se deja demostrar al lector que, con estas reglas semánticas, si *S.siguiente* no es la etiqueta de la instrucción que va inmeditamente después de S_2.*código*, entonces una proposición abarcadora proporcionará el salto a la etiqueta *S.siguiente* después del código correspondiente a S_2.

El código para $S \rightarrow$ **while** E **do** S_1 se forma como en la figura 8.22(c). Se crea y se asocia una nueva etiqueta *S.comienzo* a la primera instrucción generada por E. Otra etiqueta nueva, *E.verdadera*, se asocia a la primera instrucción de S_1. El código para E genera un salto a esta etiqueta si E es verdadera y un salto a *S.siguiente* si E es falsa; de nuevo, se iguala *E.falsa* a *S.siguiente*. Después del código para S_1 se coloca la instrucción `goto` *S.comienzo*, que produce un salto de vuelta al principio del código correspondiente a la expresión booleana. Obsérvese que S_1.*siguiente* se iguala a esta etiqueta *S.comienzo*, así que los saltos desde dentro de S_1.*código* pueden ir directamente a *S.comienzo*.

En la sección 8.6 se estudia más detalladamente la traducción de proposiciones de flujo de control, donde un método alternativo, llamado "relleno de retroceso" emite código para dichas proposiciones en una pasada.

Traducciones a flujo de control de expresiones booleanas

Ahora se estudia *E.código*, el código producido por las expresiones booleanas E en la figura 8.23. Como ya se ha indicado, E se traduce a una secuencia de proposiciones de tres direcciones que evalúan E como una secuencia de saltos condicionales e incondicionales a una de dos posiciones: *E.verdadera*, el lugar que debe alcanzar el flujo del control si E es verdadera, y *E.falsa*, el lugar que debe alcanzar el control si E es falsa.

La idea base de la traducción es la siguiente. Supóngase que E es de la forma `a < b`. Entonces el código generado es de la forma

```
if a < b goto E.verdadera
goto E.falsa
```

Supóngase que E es de la forma E_1 **or** E_2. Si E_1 es verdadera, entonces se sabe inmediatamente que E misma es verdadera, así que E_1.*verdadera* es lo mismo que *E.verdadera*. Si E_1 es falsa, entonces se debe evaluar E_2, así que se convierte E_1.*falsa* en la etiqueta de la primera proposición en el código de E_2. Las salidas con verdadero y falso de E_2 se pueden igualar a las salidas con verdadero y falso de E, respectivamente.

Se aplican consideraciones análogas a la traducción de E_1 **and** E_2. No se necesita código para una expresión E de la forma **not** E_1: sólo se intercambian las salidas con verdadero y falso de E_1 para obtener las salidas con verdadero y falso de E. En la figura 8.24 se muestra una definición dirigida por la sintaxis que genera código de tres direcciones para expresiones booleanas. Obsérvese que los atributos *verdadera* y *falsa* son heredados.

PRODUCCIÓN	REGLAS SEMÁNTICAS
$S \rightarrow$ **if** E **then** S_1	*E.verdadera* := *etiqnueva*; *E.falsa* := *S.siguiente*; S_1.*siguiente* := *S.siguiente*; *S.código* = *E.código* ‖ *gen* (*E.verdadera* ':') ‖ S_1.*código*
$S \rightarrow$ **if** E **then** S_1 **else** S_2	*E.verdadera* := *etiqnueva*; *E.falsa* := *etiqnueva*; S_1.*siguiente* := *S.siguiente* S_2.*siguiente* := *S.siguiente* *S.código* := *E.código* ‖ *gen* (*E.verdadera* ':') ‖ S_1.*código* ‖ *gen* ('goto' *S.siguiente*) ‖ *gen* (*E.falsa* ':') ‖ S_2.*código*
$S \rightarrow$ **while** E **do** S_1	*S.comienzo* := *etiqnueva*; *E.verdadera* := *etiqnueva*; *E.falsa* := *S.siguiente*; S_1.*siguiente* := *S.comienzo*; *S.código* := *gen* (*S.comienzo* ':') ‖ *E.código* ‖ *gen* (*E.verdadera* ':') ‖ S_1.*código* ‖ *gen* ('goto' *S.comienzo*)

Fig. 8.23. Definición dirigida por la sintaxis para proposiciones de flujo del control.

Ejemplo 8.4. Considérese de nuevo la expresión

```
a < b or c < d and e < f
```

Supóngase que las salidas con verdadero y falso para la expresión completa se han igualado a `Lverdadero` y `Lfalso`. Utilizando entonces la definición de la figura 8.24 se obtendría el siguiente código:

```
      if a < b goto Lverdadero
      goto L1
L1:   if c < d goto L2
      goto Lfalso
L2:   if e < f goto Lverdadero
      goto Lfalso
```

Obsérvese que el código generado no es óptimo, en el sentido de que la segunda proposición se puede eliminar sin modificar el valor del código. Las instrucciones redundantes de esta forma se pueden por tanto eliminar con un sencillo optimador local, como se puede ver en el capítulo 9. Otro enfoque que evita generar estos saltos

PRODUCCIÓN	REGLAS SEMÁNTICAS
$E \rightarrow E_1$ **or** E_2	$E_1.verdadera := E.verdadera$; $E_1.falsa := etiqnueva$; $E_2.verdadera := E.verdadera$; $E_2.falsa := E.falsa$; $E.código := E_1.código \parallel gen(E_1.falsa\ ':') \parallel E_2.código$
$E \rightarrow E_1$ **and** E_2	$E_1.verdadera := etiqnueva$; $E_1.falsa := E.falsa$; $E_2.verdadera := E.verdadera$; $E_2.falsa := E.falsa$; $E.código := E_1.código \parallel gen(E_1.verdadera\ ':') \parallel E_2.código$
$E \rightarrow$ **not** E_1	$E_1.verdadera := E.falsa$; $E_1.falsa := E.verdadera$; $E.código := E_1.código$;
$E \rightarrow (\ E_1\ )$	$E_1.verdadera := E.verdadera$; $E_1.falsa := E.falsa$; $E.código := E_1.código$;
$E \rightarrow$ **id**$_1$ **oprel** **id**$_2$	$E.código := gen('\texttt{if}'$ **id**$_1.lugar$ **oprel**$.op$ **id**$_2.lugar$ '`goto`' $E.verdadera) \parallel$ $gen('\texttt{goto}'\ E.falsa)$
$E \rightarrow$ **true**	$E.código := gen('\texttt{goto}'\ E.verdadera)$
$E \rightarrow$ **falsa**	$E.código := gen('\texttt{goto}'\ E.falsa)$

Fig. 8.24. Definición dirigida por la sintaxis para producir código de tres direcciones para expresiones booleanas.

redundantes es traducir una expresión relacional de la forma $\mathbf{id}_1 < \mathbf{id}_2$ a la proposición `if` $\mathbf{id}_1 > \mathbf{id}_2$ `goto` *E.falso* suponiendo que cuando la relación sea verdadera se evita el código.

Ejemplo 8.5. Considérese la proposición

```
while a < b do
    if c < d then
       x := y + z
    else
       x := y - z
```

La anterior definición dirigida por la sintaxis, junto con esquemas para proposiciones de asignación y expresiones booleanas, produciría el siguiente código:

```
L1:   if a < b goto L2
      goto Lsiguiente
L2:   if c < d goto L3
      goto L4
L3:   t1 := y + z
      x := t1
      goto L1
L4:   t2 := y - z
      x := t2
      goto L1
Lsiguiente:
```

Se observa que las dos primeras instrucciones `goto` se pueden eliminar modificando las direcciones de las pruebas. Se puede realizar este tipo de transformación local mediante la optimación local que se estudia en el capítulo 9. □

Expresiones booleanas en modo mixto

Es importante darse cuenta de que se ha simplificado la gramática para las expresiones booleanas. En la práctica, las expresiones booleanas a menudo contienen subexpresiones aritméticas como en `(a+b) < c`. En lenguajes donde el valor de verdad

E.tipo := *aritm*;
if E_1.*tipo* = *aritm* **and** E_2.*tipo* = *aritm* **then begin**
    /* suma aritmética normal */
    E.lugar := *tempnuevo*;
    E.código := E_1.*código* ‖ E_2.*código* ‖
        gen (*E.lugar* ':=' E_1.*lugar* '+' E_2.*lugar*)
end
else if E_1.*tipo* = *aritm* **and** E_2.*tipo* = *bool* **then begin**
    E.lugar := *tempnuevo*;
    E_2.*verdadera* := *etiqnueva*;
    E_2.*falsa* := *etiqnueva*;
    E.código := E_1.*código* ‖ E_2.*código* ‖
        gen (E_2.*verdadera* ':' *E.lugar* ':=' E_1.*lugar* + 1) ‖
        gen ('goto' *sigteprop* + 1) ‖
        gen (E_2.*falsa* ':' *E.lugar* ':=' E_1.*lugar*)
else if . . .

Fig. 8.25. Regla semántica para la producción $E \rightarrow E_1 + E_2$.

falso tiene el valor numérico `0` y verdadero tiene el valor `1`, `(a<b) + (b<a)` se puede considerar una expresión aritmética, con valor `0` si `a` y `b` tienen el mismo valor, y `1` en caso contrario.

Se puede utilizar el método para representar expresiones booleanas mediante el salto de código, aunque las expresiones sean representadas por código para calcular su valor. Por ejemplo, considérese la gramática representativa

$$E \rightarrow E + E \mid E \textbf{ and } E \mid E \textbf{ oprel } E \mid \textbf{id}$$

Se puede suponer que $E + E$ produce un resultado aritmético entero (incluir tipos aritméticos reales u otros complica las cosas pero no añade nada al valor instructivo de este ejemplo), en tanto que las expresiones E **and** E y E **oprel** E producen valores booleanos representados mediante flujo de control. La expresión E **and** E exige que ambos argumentos sean booleanos, pero las operaciones + y **oprel** toman cualquiera de esos tipos de argumento, incluidos los mixtos. $E \rightarrow$ **id** también se considera aritmético, aunque se podría ampliar este ejemplo permitiendo identificadores booleanos.

Para generar código en esta situación, se utiliza el atributo sintetizado *E.tipo*, que será o *aritm* o *bool,* dependiendo del tipo de E. E tendrá los atributos heredados *E.verdadera* y *E.falsa* para expresiones booleanas y el atributo sintetizado *E.lugar* para las expresiones aritméticas. En la figura 8.25 se muestra parte de la regla semántica para $E \rightarrow E_1 + E_2$.

En el caso de modo mixto, se genera el código para E_1, después para E_2, seguido de las tres proposiciones:

```
E2.verdadera: E.lugar := E1.lugar + 1
              goto sigteprop + 1
E2.falsa:     E.lugar := E1.lugar
```

La primera proposición calcula el valor $E_1 + 1$ para E cuando E_2 es verdadera; la tercera, el valor E_1 para E cuando E_2 es falsa. La segunda proposición es un salto sobre la tercera. Las reglas semánticas para los casos restantes y las otras producciones son muy similares, y se dejan como ejercicios.

8.5 PROPOSICIONES CASE

La proposición "**switch**" o "**case**" se encuentra disponible en una variedad de lenguajes; incluso las proposiciones `goto` calculadas y asignadas de FORTRAN se pueden considerar como variaciones de la proposición **switch**. En la figura 8.26 se muestra la sintaxis para nuestra proposición **switch**.

```
switch expresión
    begin
        case valor: proposición
        case valor: proposición
                    . . .
        case valor: proposición
        default:    proposición
    end
```

Fig. 8.26. Sintaxis de la proposición **switch**.

Hay una expresión seleccionadora, que debe evaluarse, seguida de n valores constantes que puede tomar la expresión, incluyendo tal vez un "valor" por *omisión*, que siempre concuerda con la expresión si no lo hace ningún otro valor. La traducción deseada de una proposición **switch** es código para:

1. Evaluar la expresión.
2. Encontrar qué valor de la lista de casos es el mismo que el valor de la expresión. Recuérdese que el valor por omisión concuerda con la expresión si no concuerda ninguno de los valores explícitamente mencionados en los casos.
3. Ejecutar la proposición asociada con el valor encontrado.

El paso 2 es una ramificación de n caminos, que se puede implantar de varias formas. Si el número de casos no es demasiado elevado, por ejemplo unos diez a lo sumo, entonces es razonable utilizar una secuencia de proposiciones `goto` condicionales, cada una de las cuales comprueba si hay un valor individual y hace una transferencia al código correspondiente.

Una forma más compacta de aplicar esta secuencia de proposiciones `goto` condicionales es crear una tabla de pares, donde cada par conste de un valor y una etiqueta para el código de la proposición correspondiente. Se genera código para colocar al final de esta tabla el valor de la expresión misma, emparejada con la etiqueta correspondiente a la proposición por omisión. El compilador puede generar un lazo simple para comparar el valor de la expresión con cada valor de la tabla, asegurándose que si no se encuentra ninguna otra concordancia, es seguro que concuerda la última entrada (el valor por omisión).

Si el número de valores es superior a diez más o menos, resulta más eficiente construir una tabla de dispersión (véase Sec. 7.6) para los valores, y las etiquetas de las distintas proposiciones son las entradas. Si no se encuentra ninguna entrada para el valor poseído por la expresión del **switch**, se puede generar un salto a la proposición por omisión.

Hay un caso especial bastante común en el que existe una implantación incluso más eficiente de la ramificación de n caminos. Si todos los valores están en un rango pequeño, por ejemplo de $i_{mín}$ a $i_{máx}$ y el número de valores distintos es una fracción razonable de $i_{máx} - i_{mín}$, entonces se puede construir una matriz de etiquetas, con la etiqueta de la proposición correspondiente al valor j en la entrada de la tabla con desplazamiento $j - i_{mín}$ y la etiqueta para el valor por omisión para las entradas que no tengan valor. Para realizar la proposición **switch**, se evalúa la expresión para obtener el valor j, se comprueba que esté dentro del rango $i_{mín}$ a $i_{máx}$ y se hace una transferencia indirecta a la entrada de la tabla con el desplazamiento $j - i_{mín}$. Por ejemplo, si la expresión es de tipo carácter, se puede crear una tabla de 128 entradas por ejemplo (dependiendo del conjunto de caracteres) y transferirla sin comprobar el rango.

Traducción dirigida por la sintaxis de proposiciones case

Considérese la siguiente proposición **switch**.

```
switch E
    begin
        case V1:        S1
        case V2:        S2
                ...
        case Vn-1       Sn-1
        default:        Sn
    end
```

Con un esquema de traducción dirigida por la sintaxis, es conveniente traducir esta proposición **case** a un código intermedio que tenga la forma de la figura 8.27.

Todas las comprobaciones aparecen al final, así que un generador de código simple puede reconocer la ramificación de múltiples caminos y generar código eficiente para ella, utilizando la implantación más apropiada propuesta al comienzo de esta sección. Si se genera la secuencia más directa que se muestra en la figura 8.28, el compilador tendría que hacer un análisis extensivo para encontrar la implantación más eficiente. Obsérvese que no es conveniente colocar las proposiciones de ramificación al principio, porque el compilador no podría entonces emitir código para cada una de las S_i conforme las fuera encontrando.

Para traducir a la forma de la figura 8.27, se generan dos etiquetas nuevas, `prueba` y `siguiente`, y un temporal nuevo, `t`, cuando se observa la palabra clave **switch**. Después, conforme se realiza el análisis sintáctico de la expresión E, se genera código para evaluar E en `t`. Después de procesar E, se genera el salto `goto prueba`.

```
                código para evaluar E en t
                goto prueba
L1:             código para S1
                goto siguiente
L2:             código para S2
                goto siguiente
                    . . .
Ln-1:           código para Sn-1
                goto siguiente
Ln:             código para Sn
                goto siguiente
prueba:         if t = V1 goto L1
                if t = V1 goto L2
                    . . .
                if t = Vn-1 goto Ln-1
                goto Ln
siguiente:
```

Fig. 8.27. Traducción de una proposición **case**.

```
                código para evaluar E en t
                if t ≠ V1 goto L1
                código para S1
                goto siguiente
L1:             if t ≠ V2 goto L2
                código para S2
                goto siguiente
L2:                 . . .
Ln-2:           if t ≠ Vn-1 goto Ln-1
                código para Sn-1
                goto siguiente
Ln-1:           código para Sn
siguiente:
```

Fig. 8.28. Otra traducción de una proposición **case**.

Después, cuando aparezca cada palabra clave **case,** se crea una nueva etiqueta L_i y se introduce en la tabla de símbolos. En una pila especial para almacenar los casos se coloca un apuntador a esta entrada de la tabla de símbolos y el valor V_i de la constante del caso. (Si esta proposición **switch** forma parte de una de las proposiciones internas a otra proposición **switch**, se coloca un marcador en la pila para separar los casos del **switch** interior de los de **switch** exterior.)

Cada proposición **case** V_i: S_i se procesa emitiendo la etiqueta recién creada L_i seguida del código para S_i, seguido del salto `goto siguiente`. Luego, cuando se encuentre la palabra clave **end** que finaliza el cuerpo de la proposición **switch**, se está en condiciones de generar el código para la ramificación de n caminos. Leyendo los pares apuntador-valor en la pila de casos de abajo a arriba, se puede generar una secuencia de proposiciones de tres direcciones de la forma

```
caso V1 L1
caso V2 L2
        ...
caso Vn-1 Ln-1
caso t Ln
etiqueta siguiente
```

donde `t` es el nombre que contiene el valor de la expresión seleccionadora E y L_n es la etiqueta para la proposición por omisión. La proposición de tres direcciones `caso` V_i L_i es un sinónimo de `if t =` V_i `goto` L_i de la figura 8.27, pero es más fácil que el generador de código final detecte `caso` como candidato a ser tratado de forma especial. En la fase de generación de código, estas secuencias de proposiciones `caso` se pueden traducir a una ramificación de n caminos del tipo más eficiente, dependiendo de cuántas haya y de si los valores pertenecen a un rango pequeño.

8.6 RELLENO DE RETROCESO

La forma más fácil de implantar las definiciones dirigidas por la sintaxis de la sección 8.4 es utilizar dos pasadas. Primero se construye un árbol sintáctico para la entrada y después se recorre el árbol en profundidad, realizando las traducciones dadas en la definición. El problema principal en la generación de código para las expresiones booleanas y las proposiciones de flujo del control en una sola pasada es que durante una sola pasada es posible que no se conozcan las etiquetas a las que debe ir el control en el momento en que se generan las proposiciones de salto. Se puede evitar este problemas generando una serie de proposiciones de ramificación sin especificar temporalmente los destinos de los saltos. Cada una de dichas proposiciones se colocará en una lista de proposiciones `goto` cuyas etiquetas se rellenarán cuando se pueda determinar la etiqueta adecuada. Este relleno posterior de etiquetas se denomina *relleno de retroceso*.

En esta sección se muestra cómo se puede utilizar el relleno de retroceso para genera código para las expresiones booleanas y las proposiciones de flujo del control en una pasada. Las traducciones que se generan serán de la misma forma que las de la sección 8.4, excpeto en la manera en que se generan las etiquetas. Para una mayor concreción, se generan cuádruplos en una matriz cuádruplo. Las etiquetas serán índices en esta matriz. Para manipular listas de etiquetas, se utilizan tres funciones:

1. *crealista* (i) crea una lista nueva que contiene sólo i, un índice para la matriz de cuádruplos; *crealista* devuelve un apuntador a la lista que ha elaborado.
2. *fusiona* (p_1, p_2) concatena las listas apuntadas por p_1 y p_2, y devuelve un apuntador a la lista concatenada.
3. *completa* (p, i) inserta i como la etiqueta objeto de cada una de las proposiciones de la lista apuntada por p.

Expresiones booleanas

Ahora se construye un equema de traducción adecuado para producir cuádruplos para las expresiones booleanas durante el análisis sintáctico ascendente. Se inserta un no terminal marcado M en la gramática para hacer que una acción semántica recoja, en los momentos apropiados, el índice del siguiente cuádruplo por generar. La gramática que se utiliza es la siguiente:

(1) $E \rightarrow E_1$ **or** $M\ E_2$
(2) $\quad\mid E_1$ **and** $M\ E_2$
(3) $\quad\mid$ **not** E_1
(4) $\quad\mid (\ E_1\ )$
(5) $\quad\mid$ **id**$_1$ **oprel id**$_2$
(6) $\quad\mid$ **true**
(7) $\quad\mid$ **false**
(8) $M \rightarrow \epsilon$

Los atributos sintetizados *listaverdad* y *listafalso* del no terminal E se utilizan para generar código de salto para las expresiones booleanas. Cuando se genera el código

para E, los saltos a las salidas con verdadero y falso se dejan incompletos, sin rellenar el campo de etiqueta. Estos saltos incompletos se colocan en listas apuntadas por *E.listaverdad* y *E.listafalso*, de manera apropiada.

Las acciones semánticas reflejan las condiciones mencionadas anteriormente. Considérese la producción $E \rightarrow E_1$ **and** $M\ E_2$. Si E_1 es falsa, entonces E también es falsa, de modo que las proposiciones de E_1.*listafalso* se convierten en parte de *E.listafalso*. Sin embargo, si E_1 es verdadera se debe comprobar a continuación E_2, así que el destino de las proposiciones E_1.*listaverdad* debe ser el comienzo del código generado por E_2. Este destino se obtiene utilizando el no terminal marcador M. El atributo *M.cuad* registra el número de la primera proposición de E_2.*código*. A la producción $M \rightarrow \epsilon$ se asocia la acción semántica

$\{\ M.\ cuad := sigtecuad\ \}$

La variable *sigtecuad* contiene el índice del siguiente cuádruplo por seguir. Este valor será el relleno con retroceso en la lista *E.listaverdad* cuando haya aparecido el resto de la producción $E \rightarrow E_1$ **and** $M\ E_2$. El esquema de traducción es como sigue.

(1) $E \rightarrow E_1$ **or** $M\ E_2$ { *completa* (E_1.*listafalso*, *M.cuad*);
E.listaverdad := *fusiona* (E_1.*listaverdad*, E_2.*listaverdad*);
E.listafalso := E_2.*listafalso* }

(2) $E \rightarrow E_1$ **and** $M\ E_2$ { *completa* (E_1.*listaverdad*,*M.cuad*);
E.listaverdad := E_2.*listaverdad*;
E.listafalso := *fusiona* (E_1.*listafalso*, E_2.*listafalso*) }

(3) $E \rightarrow$ **not** E_1 { *E.listaverdad* := E_1.*listafalso*;
E.listafalso := E_1.*listaverdad* }

(4) $E \rightarrow (\ E_1\ )$ { *E.listaverdad* := E_1.*listaverdad*;
E.listafalso := E_1.*listafalso* }

(5) $E \rightarrow$ **id**$_1$ **oprel** **id**$_2$ { *E.listaverdad* := *crealista* (*sigtecuad*);
E.listafalso := *crealista* (*sigtecuad*+1);
emite ('`if`' **id**$_1$.*lugar* **oprel**.op
id$_2$.*lugar* '`goto_`');
emite ('`goto_`') }

(6) $E \rightarrow$ **true** { *E.listaverdad* := *crealista* (*sigtecuad*);
emite ('`goto_`') }

(7) $E \rightarrow$ **false** { *E.listafalso* := *crealista* (*sigtecuad*);
emite ('`goto_`') }

(8) $M \rightarrow \epsilon$ { *M.cuad* := *sigtecuad* }

Para simplificar, la acción semántica (5) genera dos proposiciones, un `goto` condicional y uno incondicional. Ninguno tiene su destino completo. El índice de la primera proposición generada se coloca en una lista, y a *E.listaverdad* se le da un apuntador a esta lista. La segunda proposición generada `goto` también se coloca en una lista y se le da a *E.listafalso*.

Ejemplo 8.6. Considérese de nuevo la expresión `a < b or c < d and e < f`. En la figura 8.29 se muestra un árbol de análisis sintáctico con anotaciones. Las acciones se realizan durante un recorrido en profundidad del árbol. Como todas las acciones aparecen al final de los lados derechos, se pueden ejecutar con las reducciones durante un análisis sintáctico ascendente. En respuesta a la reducción de `a < b` a E por la producción (5), se generan los dos cuádruplos

```
100:  if a < b goto _
101:  goto _
```

(De nuevo se comienzan a numerar arbitrariamente las proposiciones con 100.) El no terminal marcador M en las producciones $E \rightarrow E_1$ **or** $M\ E_2$ registra el valor de *sigtecuad*, que en ese momento es 102. La reducción de `c < d` a E por la producción (5) genera los cuádruplos

```
102:  if c < d goto _
103:  goto _
```

Ahora se ha visto E_1 en la producción $E \rightarrow E_1$ **and** $M\ E_2$. El marcador no terminal en esta producción registra el valor en curso de *sigtecuad*, que ahora es 104. Reduciendo `e < f` a E por la producción (5) se genera

```
104:  if e < f goto _
105:  goto _
```

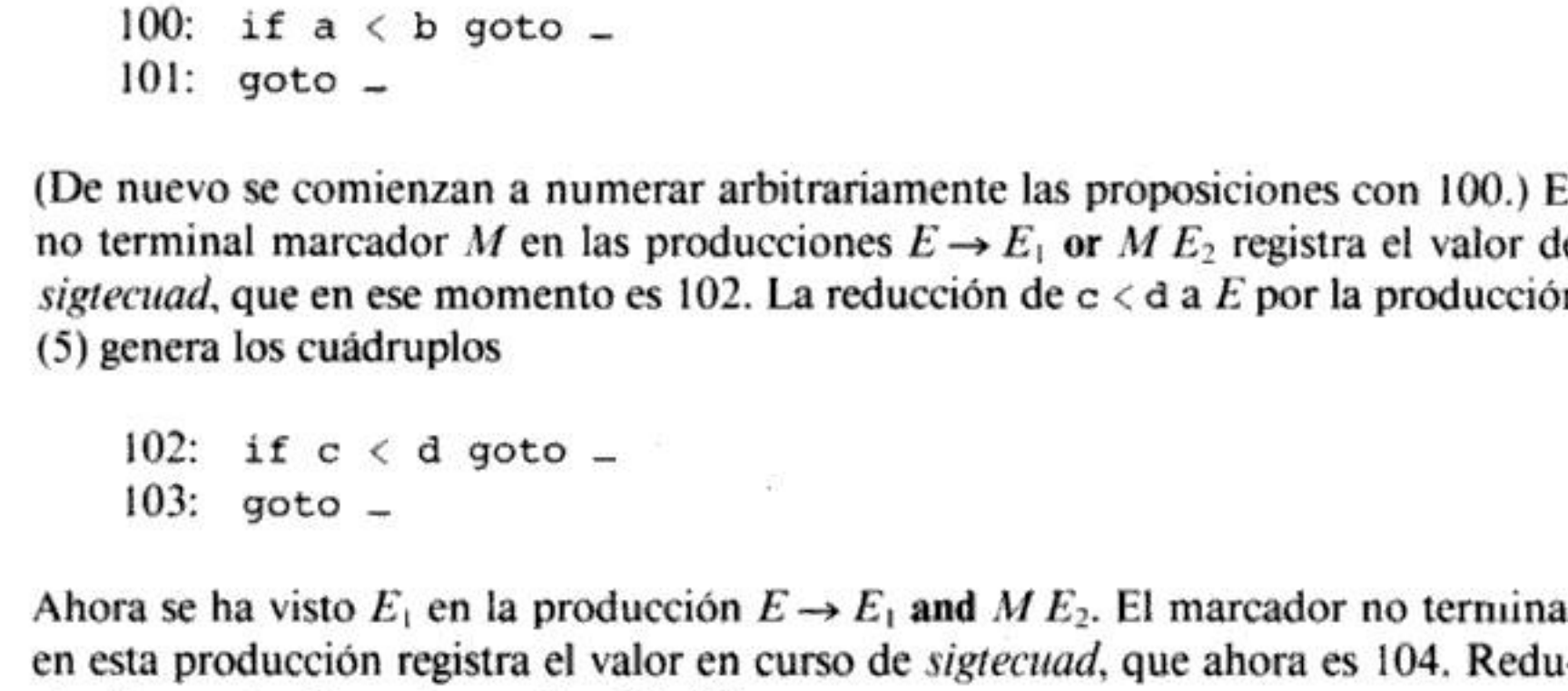


Fig. 8.29. Arbol de análisis sintáctico con anotaciones para `a<b or c<d and e<f`.

Ahora se reduce por $E \rightarrow E_1$ **and** $M\ E_2$. La acción semántica correspondiente llama a *completa*({102},104), donde {102} como argumento indica un apuntador a la lista que contiene sólo a 102, siendo esa la lista apuntada por E_1.*listaverdad*. Esta llamada a *completa* pone 104 en la proposición 102. Por tanto, las seis proposiciones generadas hasta este momento son:

```
100:  if a < b goto _
101:  goto _
102:  if c < d goto 104
103:  goto _
104:  if e < f goto _
105:  goto _
```

La acción semántica asociada con la recucción final por $E \rightarrow E_1$ **or** $M\ E_2$ llama a *completa* ({101},102) que deja a las proposiciones como:

```
100:  if a < b goto _
101:  goto 102
102:  if c < d goto 104
103:  goto _
104:  if e < f goto _
105:  goto _
```

Toda la expresión es verdadera si, y sólo si, se alcanzan los **goto** de las proposiciones 100 ó 104, y es falsa si, y sólo si, se alcanzan los `goto` de las proposiciones 103 ó 105. Estas instrucciones tendrán sus campos objeto llenos más adelante en la compilación, cuando se haya visto lo que se debe hacer según sea la expresión verdadera o falsa.

Proposiciones de flujo del control

Ahora se muestra cómo se puede utilizar el relleno de retroceso para traducir proposiciones de flujo del control en una pasada. Como antes, la atención se concentra en la generación de cuádruplos, y la notación relativa a la traducción de nombres de campos y los procedimientos para el manejo de listas de esa sección sirven asimismo para ésta. A modo de ejemplo más extenso se desarrolla un esquema de traducción para proposiciones generadas por la siguiente gramática:

(1) $S \rightarrow$ **if** E **then** S
(2) $\quad\quad \{$ **if** E **then** S **else** S
(3) $\quad\quad |$ **while** E **do** S
(4) $\quad\quad |$ **begin** L **end**
(5) $\quad\quad |\ A$
(6) $L \rightarrow L$; S
(7) $\quad\quad |\ S$

Aquí, S indica una proposición, L una lista de proposiciones, A una proposición de asignación, y E una expresión booleana. Obsérvese que debe haber otras producciones, como las correspondientes a las proposiciones de asignación. Sin embargo, las proposiciones dadas bastarán para ilustrar la técnicas utilizadas para traducir proposiciones de flujo del control.

Se emplea la misma estructura de código para las proposiciones **if-then**, **if-then-else** y **while-do** de la sección 8.4. Se supone que el código que sigue a una determinada proposición en ejecución también lo sigue físicamente en la matriz de cuádruplos. Si esto no es así, se debe proporcionar un salto explícito.

El enfoque general elegido será rellenar los saltos fuera de las proposiciones cuando se encuentren sus destinos. No sólo las expresiones booleanas necesitan de dos listas de saltos que ocurren cuando la expresión es verdadera y cuando es falsa, sino que las proposiciones también necesitan listas de saltos (dadas por el atributo *sigtelista*) al código que les sigue en la secuencia de ejecución.

Esquema para implantar la traducción

A continuación se describe un esquema de traducción dirigido por la sintaxis para generar traducciones para las construcciones de flujo de control dadas anteriormente. El no terminal E tiene dos atributos, *E.listaverdad* y *E.listafalso*, como antes. L y S también necesitan una lista de cuádruplos vacíos que más tarde habrá que completar con relleno de retroceso. Estas listas son apuntadas por los atributos *L.sigtelista* y *S.sigtelista*. *S.sigtelista* es un apuntador a una lista de todos los saltos condicionales e incondicionales al cuádruplo que sigue a la proposición S en orden de ejecución, y *L.sigtelista* se define de modo similar.

En la disposición del código para $S \rightarrow$ **while** E **do** S_1 de la figura 8.22(c), las etiquetas *S.comienzo* y *E.verdadera* marcan el comienzo del código para la proposición completa S y el cuerpo S_1. Los dos casos del no terminal marcador M en la siguiente producción registran los números de los cuádruplos de estas posiciones:

$$S \rightarrow \textbf{while}\ M_1\ E\ \textbf{do}\ M_2\ S_1$$

De nuevo, la única producción para M es $M \rightarrow \epsilon$ con una acción que asigna el atributo *M.cuad* al número del siguiente cuádruplo. Después de que se haya ejecutado el cuerpo S_1 de la proposición **while**, el control fluye al comienzo. Por tanto, cuando se reduce **while** M_1 E **do** M_2 S_1 a S, se rellena de retroceso S_1.*sigtelista* para que todos los destinos en esa lista sean M_1.*cuad*. Después del código para S_1 se añade un salto explícito al comienzo del código para E, porque el control también puede "salir del fondo". Se rellena *E.listaverdad* para que vaya al comienzo de S_1 haciendo que los saltos en *E.listaverdad* vayan a M_2.*cuad*.

Otra razón para utilizar *S.sigtelista* y *L.sigtelista* surge cuando se genera código para la proposición condicional **if** E **then** S_1 **else** S_2. Si el control "sale del fondo" de S_1, como cuando S_1 es una asignación, se debe incluir al final del código S_1 un salto sobre el código de S_2. Se utiliza otro no terminal marcador para introducir este salto después de S_1. Sea el no terminal N este marcador con producción $N \rightarrow \epsilon$. N tiene el atributo *N.sigtelista*, que será una lista formada por el número de cuádruplo de la proposición `goto _` que se genera por la regla semántica para N. A continuación se dan las reglas semánticas para la gramática revisada.

(1) $S \rightarrow$ **if** E **then** $M_1\ S_1\ N$ **else** $M_2\ S_2$

{ *completa* (*E.listaverdad*, M_1.*cuad*);
completa (*E.listafalso*, M_2.*cuad*);
S.sigtelista := *fusiona* (*S*$_1$.*sigtelista*, *fusiona* (*N.sigtelista*, S_2.*sigtelista*)) }

Se rellenan los saltos cuando E es verdadera para que vayan al cuádruplo M_1.*cuad*, que es el comienzo del código para S_1. De forma similar, se rellenan los saltos cuando E es falsa para que vayan al comienzo del código para S_2. La lista *S.sigtelista* incluye todos los saltos que salen de S_1 y S_2, así como el salto generado por N.

(2) $N \rightarrow \epsilon$ { *N.sigtelista* := *crealista* (*sigtecuad*);
emite ('`goto _`') }

(3) $M \rightarrow \epsilon$ { *M.cuad* := *sigtecuad* }

(4) $S \rightarrow$ **if** E **then** $M\ S_1$ { *completa* (*E.listaverdad*, *M.cuad*);
S.sigtelista := *fusiona* (*E.listafalso*, *S*$_1$.*sigtelista*) }

(5) $S \rightarrow$ **while** $M_1\ E$ **do** $M_2\ S_1$ { *completa* (*S*$_1$.*sigtelista*, M_1.*cuad*);
completa (*E.listaverdad*, M_2.*cuad*);
S.sigtelista := *E.listafalso*;
emite ('`goto _`' M_1.*cuad*) }

(6) $S \rightarrow$ **begin** L **end** { *S.sigtelista* := *L.sigtelista* }

(7) $S \rightarrow A$ { *S.sigtelista* := **nil** }

La asignación *S.sigtelista* := **nil** inicializa *S.sigtelista* con una lista vacía.

(8) $L \rightarrow L_1\ ;\ M\ S$ { *completa* (*L*$_1$.*sigtelista*, *M.cuad*);
L.sigtelista := *S.sigtelista* }

La proposición que sigue a L_1 en orden de ejecución es el comienzo de S. Por tanto, la lista L_1.*sigtelista* se rellena para que vaya al comienzo del código para S, que viene dado por *M.cuad*.

(9) $L \rightarrow S$ { *L.sigtelista* := *S.sigtelista* }

Obsérvese que no se generan nuevos cuádruplos en ninguna parte en estas reglas semánticas, excepto en las reglas (2) y (5). El resto del código se genera mediante las acciones semánticas asociadas con las proposiciones de asignación y las expresiones. Lo que hace el flujo del control es provocar el relleno de retroceso adecuado para que las asignaciones y las evaluaciones de las expresiones booleanas conecten apropiadamente.

Etiquetas y proposiciones goto

La construcción más elemental de un lenguaje de programación para cambiar el flujo del control en un programa es la etiqueta y el salto `goto`. Cuando un compilador encuentra una proposición como `goto L`, debe comprobar que haya exactamente

una proposición con etiqueta L en el ámbito de esta proposición goto. Si la etiqueta ya ha aparecido, en una proposición de declaración de etiqueta o como la etiqueta de alguna proposición fuente, entonces la tabla de símbolos tendrá una entrada que proporciona la etiqueta generada por el compilador para la primera instrucción de tres direcciones asociada con la proposición fuente etiquetada con L. Para la traducción se genera una proposición de tres direcciones goto con una etiqueta generada por el compilador como destino.

Cuando se encuentre una etiqueta L por primera vez en el programa fuente, en una declaración o como destino de una instrucción de salto goto hacia adelante, se introduce L en la tabla de símbolos y se genera una etiqueta simbólica para L.

8.7 LLAMADAS A PROCEDIMIENTOS

El procedimiento[6] es una construcción de programación tan importante y utilizada tan a menudo que es fundamental que un compilador genere buen código para llamadas y retornos de procedimientos. Son parte del paquete de apoyo para la ejecución, las rutinas en tiempo de ejecución que manejan el paso de argumentos a los procedimientos, las llamadas y los retornos. En el capítulo 7 se estudiaron las distintas clases de mecanismos necesarios para implantar el paquete de apoyo para la ejecución. En esta sección se estudia el código que se genera habitualmente para las llamadas y retornos de procedimientos.

Considérese una gramática para una llamada sencilla a un procedimiento.

(1) $S \rightarrow$ **call id** (*listaE*)
(2) $listaE \rightarrow listaE$, E
(3) $listaE \rightarrow E$

Secuencias de llamadas

Como se vio en el capítulo 7, la traducción de una llamada incluye una secuencia de llamada, que es una secuencia de acciones que se toman a la entrada y a la salida de cada procedimiento. Aunque las secuencias de llamada difieren, incluso en aplicaciones del mismo lenguaje, habitualmente tienen lugar las siguiente acciones:

Cuando ocurre la llamada a un procedimiento, se debe asignar espacio para el registro de activación del procedimiento llamado. Los argumentos del procedimiento llamado se deben evaluar y poner a disposición del procedimiento llamado en un lugar conocido. Se deben establecer los apuntadores de ambiente para permitir que el procedimiento llamado tenga acceso a los datos de los procedimientos abarcadores. Se debe guardar el estado del procedimiento que efectúa la llamada para que pueda reanudar la ejecución después de la llamada. También se guarda en un lugar conocido la dirección de retorno, que es la posición a la que la rutina llamada debe transferir el control cuando finalice. La dirección de retorno normalmente es la posición de la instrucción que sigue la llamada en el procedimiento autor de la

---

[6] Aquí, el término procedimiento incluye a la función. Una función es un procedimiento que devuelve un valor.

llamada. Por último, se debe generar un salto al principio del código del procedimiento llamado.

Cuando vuelve un procedimiento, deben tener lugar varias acciones. Si el procedimiento llamado es una función, el resultado se debe guardar en un lugar conocido. Se debe restablecer el registro de activación del procedimiento que hace la llamada y hay que generar un salto a la dirección de retorno del procedimiento autor de la llamada.

No existe una división exacta de las tareas en el momento de la ejecución entre el procedimiento que hace la llamada y el procedimiento que recibe la llamada. A menudo, el lenguaje fuente, la máquina objeto y el sistema operativo imponen requisitos que favorecen una solución sobre otra.

Un ejemplo sencillo

Considérese un ejemplo sencillo en el que los parámetros se pasan por referencia y la memoria se asigna estáticamente. En esta situación, se utilizan las proposiciones `param` como depositarias de los argumentos. Al procedimiento receptor de la llamada se le pasa un apuntador en un registro a la primera de las proposiciones `param`, y puede obtener un apuntador a cualquiera de sus argumentos utilizando el desplazamiento apropiado desde este apuntador de base. Cuando se genera código de tres direcciones para este tipo de llamada, basta con generar las proposiciones de tres direcciones necesarias para evaluar los argumentos que sean expresiones distintas de nombres simples, después tiene que haber una lista de proposiciones de tres direcciones `param`, una por cada argumento. Si no se quieren mezclar las proposiciones evaluadoras de argumentos con las proposiciones `param`, hay que guardar el valor de *E.lugar* para cada expresión *E* dentro de **id**(*E*, *E*, . . ., *E*)[7].

Una estructura de datos conveniente para guardar estos valores es una cola, una lista "primero en entrar-primero en salir". La rutina semántica para *listaE* → *listaE*, *E* incluirá un paso para guardar *E.lugar* en la cola llamada *cola*. Después, la rutina semántica para *S* → **call id** (*listaE*) generará una proposición `param` para cada elemento dentro de *cola*, lo cual hace que estas proposiciones sigan a las proposiciones que evalúan las expresiones como argumentos. Esas proposiciones fueron generadas cuando los argumentos mismos se redujeron a *E*. La siguiente traducción dirigida por la sintaxis incorpora estas ideas.

(1) *S* → **call id** (*listaE*)
{ **for** cada elemento *p* en *cola* **do**
emite('`param`' *p*);
emite('`call`' **id**.*lugar*) }

El código para *S* es el código para *listaE*, que evalúa los argumentos, seguido de una proposición `param` *p* para cada argumento, seguido de una proposición **call**. No se genera una cuenta del número de parámetros con la proposición `call` pero se

---

[7] Si los parámetros se pasan al procedimiento que recibe la llamada poniéndolos en una pila, como sería el caso habitual para datos asignados dinámicamente, no hay razón para mezclar proposiciones evaluadoras y proposiciones `param`. La proposición `param` se sustituye en el momento de la generación de código por código que introduzca un parámetro en la pila.

podría calcular del mismo modo en que se calculó *listaE.ndim* en la sección anterior.

(2) *listaE* → *listaE* , *E*
{ añádase *E.lugar* al final de *cola* }

(3) *listaE* → *E*
{ inicialícese *cola* para que contenga sólo *E.lugar* }

En este caso, *cola* se vacía y entonces obtiene un solo apuntador a la localidad de la tabla de símbolos correspondiente al nombre que indica el valor de *E*.

EJERCICIOS

8.1 Tradúzcase la expresión aritmética `a* - (b+c) a`
a) un árbol sintáctico
b) notación postfija
c) código de tres direcciones

8.2 Tradúzcase la expresión `- (a+b) * (c+d) + (a+b+c) a`
a) cuádruplos
b) triples
c) triples indirectos.

8.3 Tradúzcanse a
a) un árbol sintáctico
b) notación postfija
c) código de tres direcciones.
las proposiciones ejecutables del siguiente programa en C

```
main()
{
    int i:
    int a[10];
    i = 1;
    while (i <= 10) {
        a[i] = 0; i = i + 1;
    }
}
```

***8.4** Demuéstrese que si todos los operadores son binarios, entonces una cadena de operadores y operandos es una expresión postfija si, y sólo si, (1) hay exactamente un operador menos que operandos, y (2) todo prefijo no vacío de la expresión tiene menos operadores que operandos.

8.5 Modifíquese el esquema de traducción de la figura 8.11 para calcular los tipos y direcciones relativas de los nombres declarados para permitir listas de nombres en lugar de nombres simples en las declaraciones de la forma $D \rightarrow \mathbf{id} : T$.

8.6 La forma *prefija* de una expresión en la que el operador θ se aplica a expresiones $e_1, e_2, \ldots, e_k$ es $\theta p_1 p_2 \ldots p_k$, donde p_i es la forma prefija de e_i.

a) Genérese la forma prefija de `a*-(b+c)`.

**b) Demuéstrese que las expresiones infijas no se pueden traducir a la forma prefija con esquemas de traducción en que todas las acciones sean de impresión y todas ellas aparezcan al final de los lados derechos de las producciones.

c) Dése una definición dirigida por la sintaxis para traducir expresiones infijas a forma prefija. ¿Qué métodos del capítulo 5 se pueden utilizar?

8.7 Escríbase un programa para implantar la definición dirigida por la sintaxis para traducir expresiones booleanas al código de tres direcciones dado en la figura 8.24.

8.8 Modifíquese la definición dirigida por la sintaxis de la figura 8.24 para generar código para la máquina de pila de la sección 2.8.

8.9 La definición dirigida por la sintaxis de la figura 8.24 traduce $E \rightarrow \mathbf{id}_1 < \mathbf{id}_2$ al par de proposiciones

if $\mathbf{id}_1 < \mathbf{id}_2$ **goto** . . .
goto . . .

En lugar de esto se podría traducir a la proposición simple

if $\mathbf{id}_1 \geq \mathbf{id}_2$ **goto** _

y saltar el código cuando E sea verdadera. Modifíquese la definición de la figura 8.24 para generar código de esta naturaleza.

8.10 Escríbase un programa para implantar la definición dirigida por la sintaxis para las proposiciones de flujo del control dada en la figura 8.23.

8.11 Escríbase un programa para implantar el algoritmo de relleno de retroceso dado en la sección 8.6.

8.12 Tradúzcanse las siguientes proposiciones de asignación a código de tres direcciones usando el esquema de traducción de la sección 8.3.

```
A[i,j] := B[i.j] + C[A[k,l]] + D[i+j]
```

***8.13** Algunos lenguajes, como PL/I, permiten que a una lista de nombres se le proporcione una lista de atributos y también permiten que las declaraciones aniden una dentro de otra. La siguiente gramática hace una abstracción del problema:

$$
\begin{aligned}
D &\rightarrow \textit{listanombres listaatr} \\
&\mid \text{(} D \text{)} \textit{ listaatr} \\
\textit{listanombres} &\rightarrow \mathbf{id} \text{ , } \textit{listanombres} \\
&\mid \mathbf{id} \\
\textit{listaatr} &\rightarrow A \textit{ listaatr} \\
&\mid A \\
A &\rightarrow \mathbf{decimal} \mid \mathbf{fixed} \mid \mathbf{float} \mid \mathbf{real}
\end{aligned}
$$

El significado de $D \rightarrow (D)$ *listaatr* es que a todos los nombres mencionados en la declaración dentro del paréntesis se les dan los atributos que aparecen en *listaatr*, independientemente de cuántos niveles de anidamiento existan. Obsérvese que una declaración de n nombres y m atributos pueden dar lugar a que se introduzcan nm piezas de información en la tabla de símbolos. Proporciónese una definición dirigida por la sintaxis para las declaraciones definidas por esta gramática.

8.14 En C, la proposición `for` tiene la siguiente forma:

`for` (e_1 ; e_2 ; e_3) *prop*

Entendiendo que tiene el siguiente significado:

```
e1;
while (e2 ) {
    prop;
    e3;
}
```

construyase una definición dirigida por la sintaxis para traducir proposiciones `for` del estilo de C a código de tres direcciones.

8.15 El estándar de Pascal define que la proposición

for $v :=$ *inicial* **to** *final* **do** *prop*

tiene el mismo significado que la secuencia de código:

```
begin
    t1 := inicial; t2 := final;
    if t1 ⩽ t2 then begin
        v:= t1;
        prop;
        while v ≠ t2 do begin
            v := succ(v);
            prop
        end
    end
end
```

a) Considérese el siguiente programa en Pascal:

```
program ciclofor(input, output);
    var i, inicial, final: integer;
    begin
        read(inicial, final);
        for i:= inicial to final do
            writeln(i)
    end.
```

¿Qué comportamiento tiene este programa con inicial = `MAXINT` − 5 y *final* = `MAXINT`, donde `MAXINT` es el mayor número entero en la máquina objeto?

*b) Constrúyase una definición dirigida por la sintaxis que genere código de tres direcciones correcto para las proposiciones `for` de Pascal.

NOTAS BIBLIOGRAFICAS

UNCOL (por sus siglas en inglés *Universal Compiler Oriented Language*, Lenguaje orientado a un compilador universal) es un lenguaje intermedio universal mítico, buscado desde mediados de la década de 1950. Dado un UNCOL, el informe de comité de Strong y colaboradores [1958] demostró cómo se podían construir compiladores ensamblando una etapa inicial para un determinado lenguaje fuente con una etapa final para una determinada máquina objeto. Las técnicas de arranque del informe se utilizan rutinariamente para redestinar compiladores (véase Sec. 11.2). En Steel [1961] aparece una propuesta original para UNCOL.

Un compilador redestinable consta de una etapa inicial que puede unirse a varias etapas finales para implantar un lenguaje dado a varias máquinas. NELIAC es uno de los primeros ejemplos de un lenguaje con un compilador redestinable (Huskey, Halstead y McArthur [1960] escrito en su propio lenguaje. Véase también Richards [1971] y su descripción de un compilador redestinable para BCPL, Nori y colaboradores [1981] para Pascal y Johnson [1979] para C. Newey, Poole y Waite [1972] aplican la idea de cambiar la etapa final a un macroprocesador, un editor de textos y un compilador de BASIC.

El ideal de UNCOL de aplicar n lenguajes en m máquinas mediante la escritura de n etapas iniciales y m etapas finales, en oposición a $n \times m$ compiladores distintos, se ha enfocado de varias formas. Un enfoque consiste en retroadaptar una etapa inicial para un nuevo lenguaje a un compilador existente. Feldman [1979b] describe cómo se añade una etapa inicial de FORTRAN 77 a los compiladores de C de Johnson [1979] y Ritchie [1979]. Las organizaciones de compiladores diseñadas para adaptar múltiples etapas iniciales y etapas finales son descritas por Davidson y Fraser [1984b], Leverett y colaboradores [1980] y Tanenbaum y colaboradores [1983].

Los términos máquinas abstractas de "unión" e "intersección" utilizados por Davidson y Fraser [1984b] resaltan el papel del conjunto de operadores permitidos en una representación intermedia. El conjunto de instrucciones y modos de direccionamiento de una máquina de interesección son limitados, así que la etapa inicial no tiene que elegir muchas opciones cuando genera código intermedio. Las máquinas de unión proporcionan formas alternativas de implantar construcciones de nivel fuente. Como no todas las alternativas se pueden implantar directamente por todas las máquinas objeto, el conjunto de instrucciones más rico de la máquina de unión puede aliviar la dependencia con respecto a la máquina destino. Estos comentarios también sirven para otras clases de código intermedio, como los árboles sintácticos y el código de tres direcciones. Fraser y Hanson [1982] consideran formas de expresar el acceso a la pila de ejecución utilizando operaciones independientes de la máquina.

La implantación de ALGOL 60 es estudiada detalladamente por Randell y Russell [1964] y Grau, Hill y Langmaack [1967]. Freiburghouse [1969] estudia PL/I, Wirth [1971] Pascal y Branquart y colaboradores [1976] ALGOL 68.

Minker y Minker [1980] y Giegerich y Wilhelm [1978] estudian la generación de código óptimo para expresiones booleanas. El ejercicio 8.15 es obra de Newey y Waite [1985].

CAPITULO 9

Generación de código

La fase final del modelo de compilador de este libro es el generador de código. Toma como entrada una representación intermedia del programa fuente y produce como salida un programa objeto equivalente, como se indica en la figura 9.1. Las técnicas de generación de código de este capítulo se pueden utilizar con independencia de si se produce o no una fase de optimación antes de la generación de código, como en algunos compiladores denominados "de optimación". Dicha fase intenta transformar el código intermedio en una forma de la que se pueda producir código objeto más eficiente. En el siguiente capítulo se tratará con detalle la optimación de código.

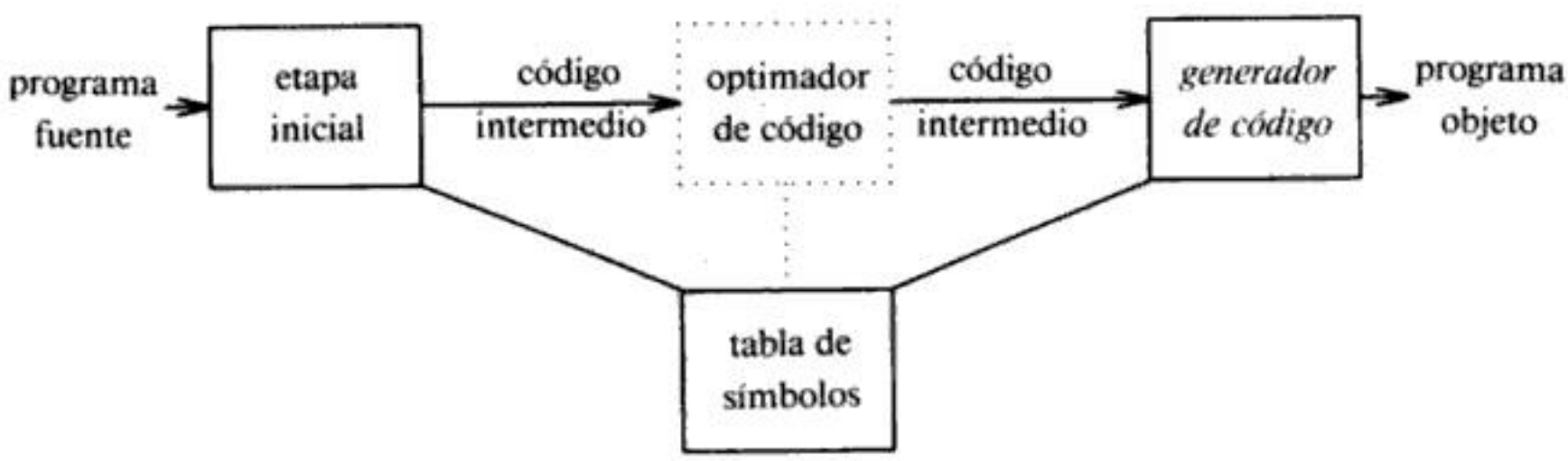


Fig. 9.1. Posición del generador de código.

Las exigencias tradicionalmente impuestas a un compilador son duras. El código de salida debe ser correcto y de gran calidad, lo que significa que debe utilizar de forma eficaz los recursos de la máquina objeto. Además, el generador de código mismo debe ejecutarse eficientemente.

Matemáticamente, el problema de generar código óptimo es indecidible. En la práctica, hay que conformarse con técnicas heurísticas que generan código bueno pero no siempre óptimo. La elección de las heurísticas es importante, ya que un algoritmo de generación de código cuidadosamente diseñado puede producir fácilmente código que sea varias veces más rápido que el producido por un algoritmo diseñado precipitadamente.

9.1 ASPECTOS DEL DISEÑO DE UN GENERADOR DE CODIGO

En tanto que los detalles dependen de la máquina objeto y del sistema operativo, aspectos como el manejo de la memoria, la selección de instrucciones, la asignación de registros y el orden de evaluación son inherentes en casi todos los problemas de generación de código. En esta sección, se examinarán los aspectos genéricos del diseño de generadores de código.

Entrada al generador de código

La entrada para el generador de código consta de la representación intermedia del programa fuente producida por la etapa inicial, junto con información de la tabla de símbolos que se utiliza para determinar las direcciones durante la ejecución de los objetos de datos denotados por los nombres de la representación intermedia.

Como se vio en el capítulo anterior, hay varias opciones para el lenguaje intermedio: representaciones lineales como la notación postfija, representaciones de tres direcciones como los cuádruplos, representaciones de una máquina virtual como el código para una máquina de pila y representaciones gráficas como los árboles sintácticos y los GDA. Aunque los algoritmos de este capítulo se expresan desde el punto de vista de código de tres direcciones, árboles y GDA, muchas de las técnicas también se aplican a otras representaciones intermedias.

Se asume que antes de la generación de código, la etapa inicial ha hecho los análisis léxico y sintáctico, y traducido el programa fuente a una representación intermedia razonablemente detallada, así que los valores de los nombres que aparecen en el lenguaje intermedio pueden ser representados por cantidades que la máquina objeto puede manipular directamente (bits, enteros, reales, apuntadores, etc.). También se supone que ya ha tenido lugar la comprobación de tipos necesaria, de modo que los operadores de conversión de tipos ya se han insertado donde fuera necesario y ya se han detectado los errores semánticos obvios (por ejemplo, intentar usar como índice de una matriz un número de punto flotante). Por tanto, la fase de generación de código puede proseguir con la hipótesis de que su entrada no contiene errores. En algunos compiladores, esta clase de comprobación semántica se realiza junto con la generación de código.

Programas objeto

La salida del generador de código es el programa objeto. Al igual que el código intermedio, esta salida puede adoptar una variedad de formas: lenguaje de máquina absoluto, lenguaje de máquina relocalizable o lenguaje ensamblador.

Producir como salida un programa en lenguaje de máquina absoluto tiene la ventaja de que se puede colocar en una posición fija de memoria y ejecutarse inmediatamente. Un programa pequeño se puede compilar y ejecutar rápidamente. Varios compiladores para "trabajos de estudiantes", como WATFIV o PL/C, producen código absoluto.

Producir como salida un programa en lenguaje de máquina relocalizable (módulo objeto) permite que los subprogramas se compilen por separado. Un conjunto de módulos objeto relocalizables se puede enlazar y cargar para su ejecución me-

diante un cargador enlazador. Aunque se tenga que pagar el costo añadido de enlazar y cargar si se producen módulos objeto relocalizables, se gana mucha flexibilidad al poder compilar subrutinas por separado y llamar desde un módulo objeto a otros programas previamente compilados. Si la máquina objeto no maneja relocalización automáticamente, el compilador debe proporcionar al cargador información de relocalización explícita para que enlace los segmentos de programa compilados por separado.

Producir como salida un programa en lenguaje ensamblador facilita el proceso de generación de código. Se pueden generar instrucciones simbólicas y utilizar las macros del ensamblador para ayudar a generar el código. El precio que se paga es el paso de ensamble después de la generación de código. Como producir código ensamblador no duplica la tarea completa del compilador, esta elección es otra alternativa razonable, especialmente para una máquina con memoria pequeña, donde un compilador debe utilizar varias pasadas. En este capítulo se usa código ensamblador como lenguaje objeto para una lectura más clara. Sin embargo, se debe insistir en que mientras las direcciones se puedan calcular según los desplazamientos y otra información almacenada en la tabla de símbolos, el generador de código puede producir direcciones relocalizables o absolutas para nombres al igual que direcciones simbólicas.

Administración de la memoria

La correspondencia entre los nombres del programa fuente con direcciones de objetos de datos en la memoria durante la ejecución la realiza la etapa inicial en cooperación con el generador de código. En el último capítulo se supuso que un nombre en una proposición de tres direcciones se refiere a una entrada en la tabla de símbolos para el nombre. En la sección 8.2, las entradas de la tabla de símbolos se iban creando conforme se examinaban las declaraciones de un procedimiento. El tipo en una declaración determina el ancho, es decir, la cantidad de memoria necesaria para el nombre declarado. Según la información de la tabla de símbolos, se puede determinar una dirección relativa para el nombre dentro de un área de datos para el procedimiento. En la sección 9.3 se proponen implantaciones mediante asignación estática y por medio de una pila de áreas de datos, y se muestra cómo se pueden convertir los nombres en una representación intermedia en direcciones en el código objeto.

Si se está generando código de máquina, hay que convertir las etiquetas de las proposiciones de tres direcciones en direcciones de instrucciones. Este proceso es análogo a la técnica de "relleno de retroceso" de la sección 8.6. Supóngase que las etiquetas se refieren a números de cuádruplos en una matriz de cuádruplos. Conforme se examina cada cuádruplo por turno se puede deducir la localidad de la primera instrucción de máquina generada para dicho cuádruplo, llevando simplemente la cuenta del número de palabras utilizadas para las instrucciones generadas hasta entonces. Esta cuenta se puede conservar en la matriz de cuádruplos (en un campo adicional), así que si se encuentra una referencia como *j*: `goto` *i*, e *i* es menor que *j*, el número del cuádruplo en curso, se puede generar simplemente una instrucción de salto con la dirección objeto igual a la localidad de máquina de la primera instrucción en el código para el cuádruplo *i*. Sin embargo, si el salto es hacia

adelante, de modo que *i* supera a *j*, hay que guardar en una lista para el cuádruplo *i* la posición de la primera instrucción de máquina generada para el cuádruplo *j*. Después, cuando se procese el cuádruplo *i*, se rellena la localidad de máquina apropiada para todas las instrucciones que sean saltos hacia adelante a *i*.

Selección de instrucciones

La naturaleza del conjunto de instrucciones de la máquina objeto determina la dificultad de la selección de instrucciones. Es importante que el conjunto de instrucción sea uniforme y completo. Si la máquina objeto no apoya cada tipo de datos de una manera uniforme, entonces cada excepción a la regla general exige un tratamiento especial.

Las velocidades de las instrucciones y las expresiones particulares de la máquina son otros factores importantes. Si no se tiene en cuenta la eficiencia del programa objeto, la selección de instrucciones es sencilla. Para cada tipo de proposición de tres direcciones, se puede diseñar un esqueleto de código que perfila el código objeto que ha de generarse para esa construcción. Por ejemplo, cada proposición de tres direcciones de la forma `x:=y+z`, donde `x`, `y` y `z` son asignadas estáticamente, se puede traducir a la secuencia de código

```
MOV y,R0  /* cargar y en el registro R0 */
ADD z,R0  /* sumar z a R0 */
MOV R0,x  /* almacenar R0 en x */
```

Desgraciadamente, esta clase de generación de código, proposición a proposición, a menudo produce código de mala calidad. Por ejemplo, la secuencia de proposiciones

```
a := b + c
d := a + e
```

se podría traducir a

```
MOV b,R0
ADD c,R0
MOV R0,a
MOV a,R0
ADD e,R0
MOV R0,d
```

Aquí, la cuarta proposición es redundante, y también la tercera si `a` no se utiliza posteriormente.

La calidad del código generado viene determinada por su velocidad y tamaño. Una máquina objeto con un conjunto de instrucciones rico puede proporcionar varios modos de aplicar una determinada operación. Como puede haber grandes diferencias de costos entre distintas implantaciones, una traducción ingenua del código intermedio puede conducir a un código objeto correcto pero inaceptablemente ineficaz. Por ejemplo, si la máquina objeto tiene una instrucción de "incremento" (`INC`), entonces la proposición de tres direcciones `a:=a+1` se puede implantar más eficientemente mediante la instrucción simple `INC a`, en lugar de

mediante una secuencia más obvia que cargue a en un registro, añade uno al registro, y después vuelva a almacenar el resultado en a:

```
MOV   a, R0
ADD   #1, R0
MOV   R0, a
```

Las velocidades de las instrucciones son necesarias para diseñar buenas secuencias de código pero, desgraciadamente, es difícil obtener información exacta de los tiempos de ejecución. Decidir cuál es la mejor secuencia de código de máquina para una determinada construcción de tres direcciones también puede exigir conocer el contexto en el que aparece esa construcción. En la sección 9.12 se estudian herramientas para construir selectores de instrucciones.

Asignación de registros

Las instrucciones que implican operandos en registros son generalmente más cortas y rápidas que las de operandos en memoria. Por tanto, utilizar eficientemente los registros es fundamental para generar un buen código. El uso de registros se divide a menudo en dos subproblemas:

1. Durante la *asignación de los registros*, se selecciona el conjunto de variables que residirá en los registros en un momento del programa.
2. Durante una fase posterior de *asignación a los registros*, se escoge el registro específico en el que residirá una variable.

Es difícil encontrar una asignación óptima de registros a variables, incluso con valores en un solo registro. Matemáticamente, el problema es NP completo. Este problema se complica aún más porque el *hardware*, el sistema operativo o ambos, en la máquina objeto pueden exigir que se cumplan ciertas convenciones del uso de registros.

Algunas máquinas exigen *parejas de registros* (un registro con número par y el siguiente con número impar) para algunos operandos y resultados. Por ejemplo, en las máquinas IBM Sistema/370 la multiplicación y la división de enteros incluyen parejas de registros. La instrucción de multiplicación es de la forma

```
M    x, y
```

donde x, el multiplicando, es el registro par de una pareja de registros par-impar. El valor del multiplicando se toma del registro impar de la pareja. El multiplicador y es un solo registro. El producto ocupa toda la pareja del registro.

La instrucción de división es de la forma

```
D    x, y
```

donde el dividendo de 64 bits ocupa una pareja de registros par e impar cuyo registro par es x; y representa al divisor. Después de la división, el registro par contiene el resto y el registro impar el cociente.

Considérense ahora las secuencias de código de tres direcciones de la figura 9.2(a) y (b), en las que la única diferencia es el operador de la segunda proposición. En la figura 9.3 se dan las secuencias más cortas en código ensamblador para (a) y (b).

```
t := a + b
t := t * c
t := t / d
```

(a)

```
t := a + b
t := t + c
t := t / d
```

(b)

Fig. 9.2. Dos secuencias de código de tres direcciones.

R*i* representa al registro *i*. (SRDA[1] R0,32 desplaza al dividendo dentro de R1 y elimina R0 de modo que todos los bits son iguales al bit de signo.) L, ST y A significan carga (*load*), almacena (*store*) y suma (*add*), respectivamente. Obsérvese que la opción óptima para el registro en el que debe cargarse a depende de lo que finalmente le ocurra a t. En la sección 9.7 se estudian estrategias para la asignación de los registros.

```
L    R1, a
A    R1, b
M    R0, c
D    R0, d
ST   R1, t
```

(a)

```
L     R0, a
A     R0, b
A     R0, c
SRDA  R0, 32
D     R0, d
ST    R1, t
```

(b)

Fig. 9.3. Secuencias óptimas de código de máquina.

Elección del orden de evaluación

El orden en el que se realicen los cálculos puede variar la eficiencia del código objeto. Algunos ordenamientos de los cálculos necesitan menos registros que otros para guardar resultados intermedios, como se verá más adelante. Elegir un orden mejor es otro problema difícil, NP completo. Al principio se evitará el problema generando código para las proposiciones de tres direcciones en el orden en que hayan sido producidas por el generador de código intermedio.

Enfoques en la generación de código

Sin duda, el criterio más importante para un generador de código es que produzca código correcto, dado el número de casos especiales con que puede encontrarse. Suponiendo que sea correcto, diseñar un generador de código para que sea fácil de aplicar, comprobar y mantener es un importante objetivo de diseño.

La sección 9.6 contiene un algoritmo directo para generación de código que utiliza información sobre los usos posteriores de un operando para generar código para

---

[1] Shift Right Double Arithmetic.

una máquina de registros. El algoritmo considera cada proposición por turno, conservando los operandos en registros mientras sea posible. Se puede mejorar el resultado de dicho generador de código mediante técnicas de optimación global tales como las estudiadas en la sección 9.9.

La sección 9.7 introduce algunas técnicas para mejorar el uso de los registros considerando el flujo del control en el código intermedio. El énfasis se concentra en la asignación de registros para operandos muy utilizados en los lazos internos.

Las secciones 9.10 y 9.11 presentan algunas técnicas de selección de código dirigidas por árboles que facilitan la construcción de generadores de código redireccionable. Se han trasladado a numerosas máquinas algunas versiones de PCC, el compilador transportable de C, producidas con dichos generadores de código. La disponibilidad del sistema operativo UNIX en una variedad de máquinas debe mucho a la transportabilidad de PCC. En la sección 9.12 se muestra cómo la generación de código se puede considerar como un proceso de reescritura de árboles.

9.2 LA MAQUINA OBJETO

La familiaridad con la máquina objeto y su conjunto de instrucciones es un prerrequisito para diseñar un buen generador de código. Desgraciadamente, en un estudio general de la generación de código no es posible describir los matices de ninguna máquina objeto tan detalladamente como para poder generar buen código para un lenguaje completo en dicha máquina. En este capítulo, se utilizará como computador objeto una máquina de registros representativa de varios minicomputadores. Sin embargo, las técnicas de generación de código de este capítulo también se han utilizado en muchas otras clases de máquinas.

Este computador objeto es una máquina direccionable por bytes, con palabra de cuatro bytes y n registros generales, `R0`, `R1`, . . . , `R`$n-1$. Tiene instrucciones de dos direcciones de la forma

op *fuente*, *destino*

donde *op* es un código de operador, y *fuente* y *destino* son campos de datos. Tiene los siguientes códigos de operaciones (entre otros):

`MOV`	(mueve *fuente* a *destino*)
`ADD`	(suma *fuente* a *destino*)
`SUB`	(resta *fuente* de *destino*)

Cuando sean necesarias, se introducirán otras instrucciones.

Los campos fuente y destino no son lo suficientemente largos como para guardar direcciones de memoria, así que algunos patrones de bits en estos campos especifican que las palabras situadas después de una instrucción contienen operandos o direcciones o ambos. La fuente y el destino de una instrucción se especifican combinando registros y posiciones de memoria con modos de direccionamiento. En la siguiente descripción, *contenido*(a) indica el contenido del registro o dirección de memoria representado por a.

Los modos de direccionamiento junto con sus formas en lenguaje ensamblador y costos asociados son como sigue:

MODO	FORMA	DIRECCIÓN	COSTO AÑADIDO
absoluto	M	M	1
registro	R	R	0
indizado	*c* (R)	*c* + *contenido* (R)	1
registro indirecto	*R	*contenido* (R)	0
indizado indirecto	**c* (R)	*contenido* (*c* + *contenido* (R))	1

Una posición de memoria M o un registro R se representa a sí mismo cuando se utiliza como fuente o como destino. Por ejemplo, la instrucción

```
MOV R0,M
```

guarda el contenido del registro R0 en la posición de memoria M.

Un desplazamiento de dirección *c* desde el valor del registro R se escribe como *c* (R). Por tanto,

```
MOV 4(R0),M
```

almacena el valor

contenido (4 + *contenido* (R0))

en la posición de memoria M.

Las versiones indirectas de los dos últimos modos se indican mediante el prefijo *. Así,

```
MOV *4(R0),M
```

almacena el valor

contenido (*contenido* (4 + *contenido* (R0)))

en la posición de memoria M.

Un modo final de direccionamiento permite que la fuente sea una constante:

MODO	FORMA	CONSTANTE	COSTO AÑADIDO
literal	#*c*	*c*	1

Por tanto, la instrucción

```
MOV #1,R0
```

carga la constante 1 en el registro R0.

Costos de las instrucciones

Se considera que el costo de una instrucción es uno más los costos asociados con los modos de dirección fuente y destino (que se indican como "costo añadido" en las tablas de modos de direccionamiento anteriores). Este costo corresponde a la longitud (en palabras) de la instrucción. Los modos de direccionamiento que implican

registros tienen costo cero, mientras que los de una posición de memoria o una literal tienen costo uno, porque dichos operandos tienen que almacenarse con la instrucción.

Si el espacio es importante, entonces evidentemente hay que reducir al mínimo la longitud de las instrucciones. Sin embargo, esto tiene una ventaja adicional importante. En la mayoría de las máquinas y de las instrucciones, el tiempo empleado en traer una instrucción de la memoria excede al tiempo empleado en ejecutar la instrucción. Por tanto, si se minimiza la longitud de las instrucciones también se tiende a minimizar asimismo el tiempo empleado en ejecutar las instrucciones[2]. A continuación se presentan algunos ejemplos.

1. La instrucción `MOV R0,R1` copia el contenido del registro `R0` en el registro `R1`. Esta instrucción tiene costo uno, porque ocupa sólo una palabra de memoria.
2. La instrucción (de almacenamiento) `MOV R5,M` copia el contenido del registro `R5` en la localidad de memoria `M`. Esta instrucción tiene costo dos, porque la dirección de la posición de memoria `M` está en la palabra que sigue a la instrucción.
3. La instrucción `ADD #1,R3` suma la constante 1 al contenido del registro 3, y tiene costo dos, porque la constante 1 debe aparecer en la siguiente palabra después de la instrucción.
4. La instrucción `SUB 4(R0),*12(R1)` almacena el valor

   contenido (*contenido* (12 + *contenido* (`R1`))) − *contenido* (*contenido* (4 + `R0`))

   en el destino `*12(R1)`. El costo de esta instrucción es tres, porque las constantes 4 y 12 se almacenan en las siguientes dos palabras después de la instrucción.

Algunas de las dificultades de generar código para esta máquina se ven cuando se considera el código que se debe generar para una proposición de tres direcciones de la forma `a:=b+c`, donde `b` y `c` son variables simples en diferentes posiciones de memoria indicadas por estos nombres. Esta proposición se puede implantar con muchas secuencias distintas de instrucciones. Aquí se presentan unos cuantos ejemplos:

1.
```
MOV   b, R0
ADD   c, R0
MOV   R0, a
```
costo = 6

2.
```
MOV   b, a
ADD   c, a
```
costo = 6

---

[2] El criterio de costo debe ser más instructivo que realista. Permitir una palabra completa para una instrucción simplifica la regla para determinar el costo. Una estimación más exacta del tiempo empleado por una instrucción tomaría en cuenta si una instrucción exige o no que el valor de un operando, además de su dirección (que se encuentra con la instrucción), se traiga de la memoria.

Suponiendo que R0, R1 y R2 contienen las direcciones de a, b y c, respectivamente, se puede utilizar:

```
3.  MOV     *R1, *R0
    ADD     *R2, *R0
```
costo = 2

Suponiendo que R1 y R2 contienen los valores de b y c, respectivamente, y que no se necesita el valor de b después de la asignación, se puede utilizar:

```
4.  ADD     R2, R1
    MOV     *R1, a
```
costo = 3

Se observa que para generar buen código para esta máquina, hay que utilizar sus capacidades eficientemente. Es preferible conservar el valor de lado izquierdo o de lado derecho de un nombre en un registro, si es posible, si se va a utilizar en un futuro cercano.

9.3 ADMINISTRACION DE LA MEMORIA DURANTE LA EJECUCION

Como se vio en el capítulo 7, la semántica de los procedimientos en un lenguaje determina cómo se enlazan los nombres con la memoria durante la ejecución. La información necesaria durante una ejecución de un procedimiento se conserva en un bloque de memoria llamado registro de activación; la memoria correspondiente a los nombres locales al procedimiento también aparece en el registro de activación.

En esta sección se estudia qué código debe generarse para administrar los registros de activación en el momento de la ejecución. En la sección 7.3 se estudiaron dos estrategias estándar para la asignación de memoria: la asignación estática y la asignación por medio de una pila. En la asignación estática, se fija la posición de un registro de activación en la memoria durante la compilación. En la asignación por medio de una pila, para cada ejecución de un procedimiento se introduce un nuevo registro de activación en la pila. El registro se saca cuando la activación finaliza. Más adelante se considerará cómo el código objeto de un procedimiento puede hacer referencia a los objetos de datos de los registros de activación.

Como se vio en la sección 7.2, un registro de activación para un procedimiento tiene campos para guardar parámetros, resultados, información del estado de la máquina, datos locales, valores temporales, etc. En esta sección, se ilustran las estrategias de asignación utilizando el campo del estado de la máquina para conservar la dirección de retorno y el campo para los datos locales. Se supone que los otros campos se manejan como se vio en el capítulo 7.

Como la asignación y la desasignación de los registros de activación durante la ejecución se produce como parte de las secuencias de llamada y de retorno de un procedimiento, la atención se concentra en las siguientes proposiciones de tres direcciones:

1. `call`,
2. `return`,
3. `halt`, y
4. `acción`, que contiene otras proposiciones.

Por ejemplo, el código de tres direcciones para los procedimientos `c` y `p` de la figura 9.4 contiene precisamente estas clases de proposiciones. El tamaño y la disposición de los registros de activación se comunican al generador de código a través de la información de la tabla de símbolos sobre los nombres. Para verlo con claridad, en la figura 9.4 se muestra la disposición en lugar de la forma de las entradas de la tabla de símbolos.

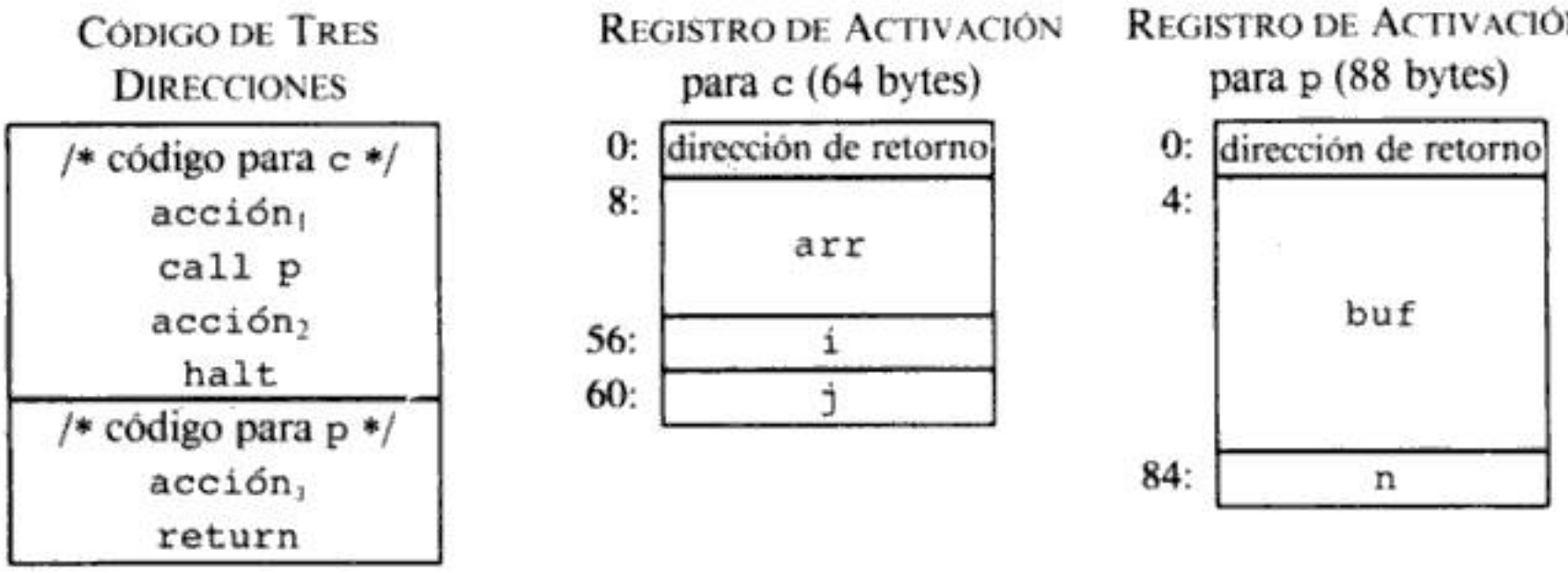


Fig. 9.4. Entrada para un generador de código.

Se asume que la memoria durante la ejecución se divide en áreas para el código, datos estáticos y una pila, como en la sección 7.2 (aquí no se utiliza el área adicional para un montículo de esa sección).

Asignación estática

Considérese el código necesario para aplicar la asignación estática. Una proposición `call` en el código intermedio se implanta mediante una secuencia de dos instrucciones de la máquina objeto. Una instrucción `MOV` guarda la dirección de retorno, y una instrucción `GOTO` transfiere el control al código objeto del procedimiento llamado:

```
MOV    #aquí + 20, llamado.área estática
GOTO   llamado.área código
```

Los atributos *llamado.área estática* y *llamado.área código* son constantes que se refieren a la dirección del registro de activación y a la primera instrucción del procedimiento llamado, respectivamente. La fuente *#aquí* + 20 en la instrucción `MOV` es la dirección de retorno literal y es la dirección de la instrucción que sigue a la instrucción `GOTO`. (Según el estudio de la Sec. 9.2, las tres constantes más las dos instrucciones en la secuencia de llamada cuestan cinco palabras a 20 bytes.)

El código para un procedimiento finaliza con un retorno al procedimiento que efectúa la llamada, excepto que el primer procedimiento no recibe ninguna llamada, así que su instrucción final es HALT, que seguramente devuelve el control al sistema operativo. Un retorno del procedimiento *llamado* se implanta con

GOTO **llamado.área estática*

que transfiere el control a la dirección guardada al principio del registro de activación.

Ejemplo 9.1. El código de la figura 9.5 se construye a partir de los procedimientos c y p de la figura 9.4. Se utiliza la seudoinstrucción ACCION para implantar la proposición acción, que representa el código de tres direcciones que no es relevante para este estudio. Arbitrariamente inicia el código para estos procedimientos en las direcciones 100 y 200, respectivamente, y se supone que cada instrucción ACCION ocupa 20 bytes. Los registros de activación para el procedimiento se asignan estáticamente comenzando en la localidad 300 y 364, respectivamente.

Las instrucciones que comienzan en la dirección 100 implantan las proposiciones

```
acción1; call p; acción2; halt
```

del primer procedimiento c. Por tanto, la ejecución comienza con la instrucción $ACCION_1$ en la dirección 100. La instrucción MOV en la dirección 120 guarda la ins-

```
                        /* código para c */
100:  ACCION1
120:  MOV #140, 364     /* guarda la dirección de retorno 140 */
132:  GOTO 200          /* llamada a p */
140:  ACCION2
160:  HALT
        . . .
                        /* código para p */
200:  ACCION3
220:  GOTO *364         /* regresa a la dirección guardada en la localidad 364 */
        . . .

                        /* 300 a 363 contienen el registro de activación para c */
300:                    /* dirección de retorno */
304:                    /* datos locales para c */
        . . .

                        /* 364 a 451 contienen el registro de activación para p */
364:                    /* dirección de retorno */
368:                    /* datos locales para p */
```

Fig. 9.5. Código objeto para la entrada de la figura 9.4.

trucción de retorno 140 en el campo del estado de la máquina, que es la primera palabra en el registro de activación de p. La instrucción GOTO en la dirección 132 transfiere el control a la primera instrucción del código objeto del procedimiento llamado.

Como la secuencia de llamada anterior guardó 140 en la dirección 364, *364 representa a 140 cuando se ejecuta la proposición GOTO en la dirección 220. Por tanto, el control regresa a la dirección 140 y se reanuda la ejecución del procedimiento c. □

Asignación por medio de una pila

La asignación estática se puede convertir en asignación por medio de una pila, utilizando direcciones relativas de memoria en los registros de activación. La posición del registro para una activación de un procedimiento no se conoce hasta el momento de la ejecución. En la asignación por medio de una pila, esta posición se guarda generalmente en un registro, así que se puede acceder a las palabras del registro de activación mediante desplazamientos desde el valor de dicho registro. El modo de direccionamiento indizado de la máquina objeto descrita en este capítulo sirve para este propósito.

Las direcciones relativas en un registro de activación se pueden considerar como desplazamientos desde cualquier posición conocida en el registro de activación, como se vio en la sección 7.3. Conviene utilizar desplazamientos positivos manteniendo en un registro SP un apuntador al comienzo del registro de activación del tope de la pila. Cuando ocurre una llamada de procedimiento, el procedimiento que hace la llamada incrementa SP y transfiere el control al procedimiento que recibe la llamada. Después de que el control regresa al autor de la llamada, decrementa SP, desasignando por tanto el registro de activación del procedimiento llamado[3].

El código para el primer procedimiento inicializa la pila haciendo que SP apunte al comienzo del área de la pila en la memoria:

```
MOV #comienzopila, SP          /* inicializa la pila */
código para el primer procedimiento
HALT                           /* termina la ejecución */
```

Una secuencia de llamada a un procedimiento incrementa SP, guarda la dirección de retorno y transfiere el control al procedimiento llamado:

```
ADD #llamador.tamañoregistro, SP
MOV #aquí + 16, *SP            /* guarda la dirección de retorno */
GOTO llamado.área código
```

El atributo *llamador.tamañoregistro* representa el tamaño de un registro de activación, de modo que la instrucción ADD deja SP apuntando al comienzo del siguiente registro de activación. La fuente *#aquí* + 16 en la instrucción MOV es la dirección de la instrucción después de GOTO; se guarda en la dirección apuntada por SP.

---

[3] Con desplazamientos negativos, SP podría apuntar al final de la pila y el procedimiento llamado podría incrementar SP.

CÓDIGO DE TRES DIRECCIONES

```
/* código para s */
acción1
call q
acción2
halt
/* código para p */
acción3
return
/* código para q */
acción4
call p
acción5
call q
acción6
call q
return
```

Fig. 9.6. Código de tres direcciones para ilustrar la asignación por medio de una pila.

La secuencia de regreso consta de dos partes. El procedimiento llamado transfiere el control a la dirección de retorno utilizando

```
GOTO *0(SP)      /* retorno al llamador */
```

La razón para utilizar `*0(SP)` en la instrucción `GOTO` es que se necesitan dos niveles de indirección: `0(SP)` es la dirección de la primera palabra en el registro de activación y `*0(SP)` es la dirección de regreso allí guardada.

La segunda parte de la secuencia de regreso está en el autor de la llamada, que decrementa `SP` restableciendo así `SP` a su valor previo. Es decir, después de la substracción, `SP` apunta al comienzo del registro de activación del autor de la llamada:

`SUB` #*llamador.tamañoregistro*, `SP`

En la sección 7.3 hay un estudio más amplio de las secuencias de llamada y de los compromisos en la división del trabajo entre los procedimientos autor y receptor de la llamada.

Ejemplo 9.2. El programa de la figura 9.6 es una condensación del código de tres direcciones para el programa en Pascal estudiado en la sección 7.1. El procedimiento `c` es recursivo, de modo que puede haber más de una activación activa de `c` a la vez.

Supóngase que se han determinado los tamaños de los registros de activación de los procedimientos `o`, `p` y `c` durante la compilación como *otam*, *ptam* y *ctam*, respectivamente. La primera palabra en cada registro de activación contiene una dirección de retorno. Arbitrariamente se asume que el código para estos procedimientos comienza en las direcciones 100, 200 y 300, respectivamente, y que la pila comienza en 600. El código objeto para el programa de la figura 9.6 es como sigue:

```
                                /* código para o */
100:  MOV #600, SP              /* inicia la pila */
108:  ACCION1
128:  ADD #otam, SP             /* comienza la secuencia de llamada */
136:  MOV #152, *SP             /* introduce la dirección de retorno */
144:  GOTO 300                  /* llama a c */
152:  SUB #otam, SP             /* restablece SP */
160:  ACCION2
180:  HALT
      ...
                                /* código para p */
220:  GOTO *0(SP)
220:  GOTO *0(SP)               /* retorno */
      ...
                                /* código para c */
300:  ACCION4                   /* salto condicional a 456 */
320:  ADD #ctam, SP
328:  MOV #344, *SP             /* introduce la dirección de retorno */
336:  GOTO 200                  /* llama a p */
344:  SUB #ctam, SP
352:  ACCION5
372:  AND #ctam, SP
380:  MOV #396, *SP             /* introduce la dirección de retorno */
388:  GOTO 300                  /* llama a c */
396:  SUB #ctam, SP
404:  ACCION6
424:  ADD #ctam, SP
432:  MOV #448, *SP             /* introduce la dirección de retorno */
440:  GOTO 300                  /* llama a c */
448:  SUB #ctam, SP
456:  GOTO *0(SP)               /* retorno */
      ...
600:                            /* aquí comienza la pila */
```

Se supone que $\texttt{ACCION}_4$ contiene un salto condicional a la dirección 456 de la secuencia de retorno de c; de lo contrario, el procedimiento recursivo c está condenado a llamarse a sí mismo para siempre. Más adelante, se considera un ejemplo de una ejecución del programa en el que la primera llamada de c no retorna inmediatamente, pero sí las llamadas posteriores.

Si *otam*, *ptam* y *ctam* son 20, 40 y 60, respectivamente, entonces SP se inicializa con 600, que es la primera dirección de la pila, mediante la primera instrucción en la dirección 100. SP contiene 620 justo antes de que el control se transfiera de o a c, porque *otam* es 20. Por tanto, cuando c llama a p, la instrucción en la dirección 320 incrementa SP a 680, donde comienza el registro de activación para p; SP vuelve a 620 después de que el control retorne a c. Si las siguientes dos llamadas recursivas de c retornan inmediatamente, el valor máximo de SP durante esta ejecución es 680.

Obsérvese, sin embargo, que la última posición de la pila utilizada es 739, puesto que el registro de activación para c que comienza en la posición 680 ocupa 60 bytes. □

Direcciones para los nombres en tiempo de ejecución

La estrategia de asignación de memoria y la distribución de los datos locales en un registro de activación para un procedimiento determinan cómo se accede a las posiciones de memoria de los nombres. En el capítulo 8 se supuso que un nombre en una proposición de tres direcciones es realmente un apuntador a una entrada de la tabla de símbolos para el nombre. Este enfoque tiene una ventaja significativa; hace más transportable el compilador, ya que no hay que modificar la etapa inicial aunque el compilador se traslade a una máquina distinta donde se necesite una organización distinta durante la ejecución (por ejemplo, el *display* se puede conservar en registros en lugar de hacerlo en memoria). Por otra parte, generar la secuencia específica de pasos de acceso mientras se genera el código intermedio puede suponer una ventaja importante en un compilador optimador, ya que permite al compilador aprovechar detalles que ni siquiera vería en la proposición de tres direcciones simples.

En cualquier caso, los nombres deben sustituirse finalmente por código para acceder a posiciones de memoria. Por tanto, se consideran algunas elaboraciones de la proposición de copia de tres direcciones simple `x:=0`. Después de procesar las declaraciones de un procedimiento, supóngase que la entrada de la tabla de símbolos para `x` contiene una dirección relativa 12 para `x`. Primero considérese el caso en el que `x` está en un área asignada estáticamente y que comienza en la dirección *estática*. Entonces, la dirección real durante la ejecución de `x` es *estática* + 12. Aunque el compilador puede finalmente determinar el valor de *estática* + 12 durante la compilación, la posición del área estática puede no conocerse cuando se genera el código intermedio para acceder al nombre. En este caso, es razonable generar código de tres direcciones para "calcular" *estática* + 12, sabiendo que este cálculo se efectuará durante la fase de generación de código, o posiblemente por el cargador, antes de que se ejecute el programa. Entonces la asignación `x:=0` traduce a

```
estática[12] := 0
```

Si el área estática comienza en la dirección 100, el código objeto para esta proposición es

```
MOV #0, 112
```

Por otra parte, supóngase que el lenguaje es como Pascal y que se utiliza un *display* para acceder a los nombres no locales, como se estudió en la sección 7.4. Supóngase también que el *display* reside en registros y que `x` es local a un procedimiento activo cuyo apuntador al *display* está en el registro `R3`. Entonces se puede traducir la copia `x:=0` a las proposiciones de tres direcciones

```
t1 := 12 + R3
*t1 := 0
```

donde t_1 contiene la dirección de x. Esta secuencia se puede implantar mediante la instrucción de máquina

```
MOV #0, 12(R3)
```

Obsérvese que el valor del registro R3 no se puede determinar en la compilación.

9.4 BLOQUES BASICOS Y GRAFOS DE FLUJO

Una representación de grafos de proposiciones de tres direcciones, llamada grafo de flujo, es útil para entender los algoritmos de generación de código, incluso aunque el grafo no esté explícitamente construido por un algoritmo de generación de código. Los nodos del grafo de flujo representan cálculos y las aristas representan el flujo de control. En el capítulo 10 se utiliza mucho el grafo de flujo de un programa como vehículo para recoger información sobre el programa intermedio. Algunos algoritmos de asignación de registros utilizan grafos de flujo para encontrar los lazos internos donde se supone que un programa emplea la mayor parte de su tiempo.

Bloques básicos

Un *bloque básico* es una secuencia de proposiciones consecutivas en las que el flujo de control entra al principio y sale al final sin detenerse y sin posibilidad de saltar excepto al final. La siguiente secuencia de proposiciones de tres direcciones forma un bloque básico:

$$\begin{aligned} t_1 &:= a * a \\ t_2 &:= a * b \\ t_3 &:= 2 * t_2 \\ t_4 &:= t_1 + t_3 \\ t_5 &:= b * b \\ t_6 &:= t_4 + t_5 \end{aligned} \tag{9.1}$$

Una proposición de tres direcciones x := y+z *define* x y *usa* (o *se refiere a*) y y z. Se dice que un nombre en un bloque básico está *activo* en un punto dado si su valor se utiliza después de ese punto en el programa, tal vez en otro bloque básico.

El siguiente algoritmo se puede utilizar para particionar una secuencia de proposiciones de tres direcciones en bloques básicos.

Algoritmo 9.1. Partición en bloques básicos.

Entrada. Una secuencia de proposiciones de tres direcciones.

Salida. Una lista de bloques básicos donde cada proposición de tres direcciones está en un bloque exactamente.

Método.

1. Primero se determina el conjunto de *líderes*, la primera proposición de cada bloque básico. Las reglas que se utilizan son las siguientes:

i) La primera proposición es un líder.
ii) Cualquier proposición que sea el destino de un salto **goto** condicional o incondicional es un líder.
iii) Cualquier proposición que vaya inmediatamente después de un salto **goto** condicional o incondicional es un líder.

2. Para cada líder, su bloque básico consta del líder y de todas las proposiciones hasta, pero sin incluirlo, el siguiente líder o el fin del programa. □

Ejemplo 9.3. Considérese el fragmento de código fuente que se muestra en la figura 9.7; calcula el producto punto de dos vectores a y b de longitud 20. En la figura 9.8 se muestra una lista de proposiciones de tres direcciones que realizan estos cálculos en la máquina objeto que se está considerando.

```
begin
    prod := 0;
    i := 1;
    do begin
        prod := prod + a[i] * b[i];
        i := i + 1
    end
    while i <= 20
end
```

Fig. 9.7. Programa para calcular el producto punto.

Ahora se aplica el algoritmo 9.1 al código de tres direcciones de la figura 9.8 para determinar sus bloques básicos. La proposición (1) es un líder según la regla i) y la proposición (3) es un líder según la regla ii), puesto que la última proposición puede saltar a ella. Según la regla iii), la proposición que sigue a (12) (recuérdese que la

```
(1)   prod := 0
(2)   i  := 1
(3)   t1 := 4 * i
(4)   t2 := a [ t1 ]    /* calcula a[i] */
(5)   t3 := 4 * i
(6)   t4 := b [ t3 ]    /* calcula b[i] */
(7)   t5 := t2 * t4
(8)   t6 := prod + t5
(9)   prod := t6
(10)  t7 := i + 1
(11)  i  := t7
(12)  if i <= 20 goto (3)
```

Fig. 9.8. Código de tres direcciones para calcular el producto punto.

Fig. 9.13 es sólo un fragmento de un programa) es un líder. Por tanto, las proposiciones (1) y (2) forman un bloque básico. El resto del programa, comenzando por la proposición (3), forma un segundo bloque básico. □

Transformaciones en bloques básicos

Un bloque básico calcula un conjunto de expresiones. Estas expresiones son los valores de los nombres activos al salir del bloque. Se dice que dos bloques básicos son *equivalentes* si calculan el mismo conjunto de expresiones.

Se pueden aplicar varias transformaciones a un bloque básico sin modificar el conjunto de expresiones calculadas por el bloque. Muchas de estas transformaciones son útiles para mejorar la calidad del código que finalmente será generado a partir de un bloque básico. En el capítulo siguiente, se muestra cómo un "optimador" de código global intenta utilizar dichas transformaciones para ordenar los cálculos de un programa para reducir el tiempo total de ejecución o la exigencia de espacio del programa objeto final. Hay dos clases importantes de transformaciones locales que se pueden aplicar a los bloques básicos; las transformaciones que preservan la estructura y las transformaciones algebraicas.

Transformaciones que preservan la estructura

Las principales transformaciones que preservan la estructura en bloques básicos son:

1. eliminación de subexpresiones comunes
2. eliminación de código inactivo
3. renombramiento de variables temporales
4. intercambio de dos proposiciones adyacentes independientes

Ahora se examinan estas transformaciones más detalladamente. Por el momento, se supone que los bloques básicos no tienen matrices apuntadoras o llamadas a procedimientos.

1. *Eliminación de subexpresiones comunes*. Considérese el bloque básico

```
a := b + c
b := a - d
c := b + c
d := a - d
```
(9.2)

La segunda y la cuarta proposiciones calculan la misma expresión, `b+c-d`, y por tanto este bloque básico se puede transformar en el bloque equivalente

```
a := b + c
b := a - d
c := b + c
d := b
```
(9.3)

Obsérvese que aunque la primera y la tercera proposiciones en (9.2) y (9.3) parecen tener la misma expresión a la derecha, la segunda proposición redefine `b`. Por tanto, el valor de `b` en la tercera proposición es distinto al valor de `b` en la primera, y la primera y la segunda proposiciones no calculan la misma expresión.

2. *Eliminación de código inactivo.* Supóngase que x está inactivo, es decir, que no se vuelve a utilizar posteriormente, en el punto en que la proposición `x:=y+z` aparece en un bloque básico. Entonces esta proposición se puede eliminar sin modificar el valor del bloque básico.

3. *Renombramiento de variables temporales.* Supóngase que se tiene una proposición `t:=b+c`, donde `t` es un nombre temporal. Si se cambia esta proposición por `u:=b+c`, donde `u` es una variable temporal nueva, y se cambian todos los usos de este ejemplo de `t` por `u`, entonces no se cambia el valor del bloque básico. De hecho, siempre se puede transformar un bloque básico en un bloque equivalente en el que cada proposición que define un temporal define un temporal nuevo. Dicho bloque básico se denomina bloque en *forma normal.*

4. *Intercambio de proposiciones.* Supóngase que se tiene un bloque con las dos proposiciones adyacentes

```
t1 := b + c
t2 := x + y
```

Entonces se pueden intercambiar las dos proposiciones sin que esto afecte al valor del bloque si, y sólo si, ni x ni y son t_1 y ni b ni c son t_2. Obsérvese que un bloque básico en forma normal permite todos los intercambios de proposiciones que sean posibles.

Transformaciones algebraicas

Se pueden utilizar innumerables transformaciones algebraicas para cambiar el conjunto de expresiones calculadas por un bloque básico por un conjunto algebraicamente equivalente. Las transformaciones útiles son las que simplifican las expresiones o sustituyen operaciones caras por otras más baratas. Por ejemplo, las proposiciones como

```
x := x + 0
```

o

```
x := x * 1
```

se pueden eliminar de un bloque básico sin cambiar el conjunto de expresiones que calcula. El operador de exponenciación en la proposición

```
x := y ** 2
```

exige generalmente la llamada a una función para implantarlo. Utilizando una transformación algebraica, esta proposición se puede sustituir por la proposición más sencilla pero equivalente

```
x := y * y
```

Las transformaciones algebraicas se estudian más detalladamente en la sección 9.9 sobre optimación local y en la sección 10.3 sobre optimación de bloques básicos.

Grafos de flujo

Se puede añadir la información sobre el flujo del control al conjunto de bloques básicos que componen un programa, construyendo un grafo dirigido llamado *grafo de flujo*. Los nodos del grafo de flujo son los bloques básicos. Un nodo se distingue como *inicial*; es el bloque cuyo líder es la primera proposición. Hay una arista dirigida del bloque B_1 al bloque B_2 si B_2 puede ir inmediatamente después de B_1 en una secuencia de ejecución; es decir, si

1. hay un salto condicional o incondicional desde la última proposición de B_1 a la primera proposición de B_2, o
2. B_2 sigue inmediatamente a B_1 en el orden del programa, y B_1 no termina con un salto incondicional.

Se dice que B_1 es un *predecesor* de B_2, y que B_2 es un *sucesor* de B_1.

Ejemplo 9.4. En la figura 9.9 se muestra el grafo de flujo del programa de la figura 9.7. B_1 es el nodo inicial. Obsérvese que en la última proposición se sustituyó el salto a la proposición (3) por un salto equivalente al comienzo del bloque B_2. □

Representación de bloques básicos

Los bloques básicos se pueden representar mediante una variedad de estructuras de datos. Por ejemplo, después de particionar las proposiciones de tres direcciones con el algoritmo 9.1, cada bloque básico puede ser representado por un registro que consta de una cuenta del número de cuádruplos dentro del bloque, seguida de un apuntador al líder (primer cuádruplo) del bloque, y de las listas de predecesores y sucesores del bloque. Como alternativa se puede hacer una lista enlazada de los cuádruplos de cada bloque. Las referencias explícitas a los números de los cuádruplos en las proposiciones de salto al final de los bloques básicos pueden causar algún problema si se cambian de lugar los cuádruplos durante la optimación del código. Por ejemplo,

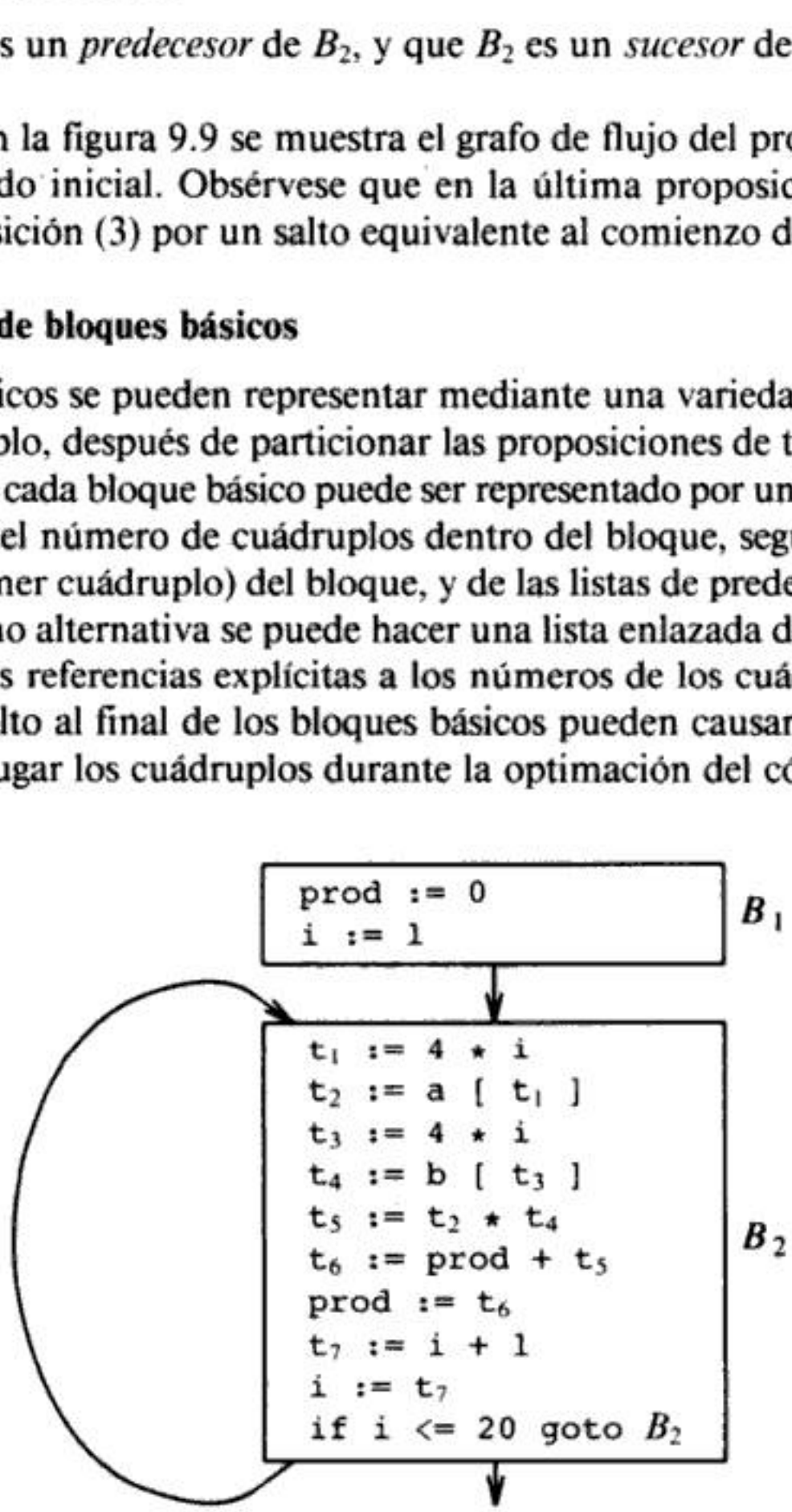


Fig. 9.9. Grafo de flujo para un programa.

si el bloque B_2, que va de las proposiciones (3) a la (12) en el código intermedio de la figura 9.9, se trasladara a otro lugar en la matriz de cuádruplos o si se contrajera, habría que modificar el (3) en `if i <= 20 goto` (3). Por tanto, es preferible que los saltos apunten a bloques en lugar de a cuádruplos, como se hizo en la figura 9.9.

Es importante tener en cuenta que una arista del grafo de flujo desde el bloque B al bloque B' no especifica las condiciones bajo las cuales el control fluye de B a B'. Es decir, la arista no indica si el salto condicional al final de B (si es que hay un salto condicional) va al líder de B' cuando la condición se cumple o cuando no se cumple. Esta información se puede recuperar si se necesita por la proposición de salto en B.

Lazos

En un grafo de flujo, ¿qué es un lazo, y cómo se pueden encontrar todos los lazos? Generalmente es fácil contestar a estas preguntas. Por ejemplo, en la figura 9.9 hay un lazo, formado por el bloque B_2. Sin embargo, las respuestas generales a estas preguntas son bastantes sutiles, y se examinarán detalladamente en el siguiente capítulo. Por el momento, basta con indicar que un lazo es una serie de nodos en un grafo de flujo tales que

1. Todos los nodos en la serie están *fuertemente conectados*; es decir, desde cualquier nodo dentro del lazo a cualquier otro, hay un camino de longitud uno o más, siempre dentro del lazo, y
2. La serie de nodos tiene una *entrada* única, es decir, un nodo dentro del lazo tal que la única forma de alcanzar un nodo del lazo desde un nodo fuera del lazo es atravesar primero la entrada.

Un lazo que no contiene otros lazos se denomina lazo *interno*.

9.5 INFORMACION SOBRE EL SIGUIENTE USO

En esta sección se reúne la información del siguiente uso sobre nombres en los bloques básicos. Si ya no se necesita el nombre en un registro, entonces el registro se puede asignar a algún otro nombre. Esta idea de conservar un nombre en memoria, sólo si va a utilizarse posteriormente, se puede aplicar en varios contextos. Se utilizó en la sección 5.8 para asignar espacio a los valores de atributos. El generador de código simple de la siguiente sección la aplica a la asignación de registros. Como última aplicación, se considera la asignación de memoria para nombres temporales.

Cálculo de los siguientes usos

El *uso* de un nombre en una proposición de tres direcciones se define de la siguiente manera. Supóngase que la proposición de tres direcciones i asigna un valor a x. Si la proposición j tiene a x como un operando y el control puede fluir de la proposición i a la j por un camino que no tiene asignaciones intermedias a x, entonces se dice que la proposición j *usa* el valor de x calculado en i.

Se desea determinar para cada proposición de tres direcciones x := y *op* z cuáles son los siguientes usos de x, y y z. Por el momento, no interesan los usos fuera del bloque básico que contiene esta proposición de tres direcciones pero se puede intentar determinar si existe uso mediante la técnica de análisis de variables activas del capítulo 10.

El algoritmo que aquí se presenta para determinar los usos siguientes realiza una pasada hacia atrás sobre cada bloque básico. Se puede examinar fácilmente una cadena de proposiciones de tres direcciones para encontrar los finales de los bloques básicos como en el algoritmo 9.1. Como los procedimientos pueden tener efectos secundarios arbitrarios, se supone por conveniencia que cada llamada a un procedimiento inicia un nuevo bloque básico.

Habiendo encontrado el final de un bloque básico, se inspecciona hacia atrás hasta el comienzo, registrando (en la tabla de símbolos) para cada nombre x si x tiene o no un siguiente uso en el bloque y si no lo tiene, indicando si está activo a la salida de ese bloque. Si se ha hecho el análisis del flujo de datos que se presenta en el capítulo 10, se sabe qué nombres están activos a la salida de cada bloque. Si no se ha hecho el análisis de variables activas, se puede suponer que todas las variables no temporales están activas a la salida. Si los algoritmos que generan el código intermedio o que optimizan el código permiten que ciertos temporales se utilicen a través de bloques, éstos también se deben considerar activos. Sería buena idea marcar dichos temporales, de modo que no haya que considerar activos todos los temporales.

Supóngase que se alcanza la proposición de tres direcciones *i*: x := y *op* z en el examen hacia atrás. Entonces se hace lo siguiente:

1. Se asocia a la proposición *i* la información encontrada en la tabla de símbolos relativa al siguiente uso y actividad de x, y y z[4].
2. En la tabla de símbolos, se asigna a x "no activo" y "sin uso siguiente".
3. En la tabla de símbolos, se indica que y y z están "activos" y se igualan los siguientes usos y y z a *i*. Obsérvese que no se puede intercambiar el orden de los pasos 2 y 3 porque x puede ser y o z.

Si la proposición de tres direcciones *i* es de la forma x := y o x := *op* y, los pasos son los mismos que antes, sin tener en cuenta z.

Asignación de memoria para los nombres temporales

Aunque puede ser útil en un compilador optimador crear un nombre diferente cada vez que se necesite un temporal (véase Cap. 10), hay que asignar espacio para guardar los valores de dichos temporales. El tamaño del campo para los temporales en el registro de activación general de la sección 7.2 aumenta con el número de temporales.

---

[4] Si x no está activo, entonces esta proposición se puede borrar; dichas transformaciones se consideran en la sección 9.8

En general, se pueden empaquetar dos temporales en la misma posición si no están activos simultáneamente. Como casi todos los temporales se definen y se utilizan dentro de bloques básicos, se puede aplicar la información del siguiente uso para empaquetar temporales. Para los temporales que se utilizan a través de los bloques, el capítulo 10 estudia el análisis del flujo de datos necesarios para calcular su actividad.

Se pueden asignar posiciones de memoria para los temporales examinando cada uno por turno y asignando un temporal a la primera posición dentro del campo para los temporales que no contengan un temporal activo. Si no se puede asignar un temporal a ninguna posición previamente creada, se añade una nueva posición al área de datos del procedimiento en curso. En muchos casos, los temporales se pueden empaquetar dentro de registros en lugar de en posiciones de memoria, como ocurre en la siguiente sección.

Por ejemplo, los seis temporales del bloque básico (9.1) se pueden empaquetar dentro de dos posiciones. Estas posiciones corresponden a t_1 y t_2 en:

```
t1 := a * a
t2 := a * b
t2 := 2 * t2
t1 := t1 + t2
t2 := b * b
t1 := t1 + t2
```

9.6 UN GENERADOR DE CODIGO SIMPLE

La estrategia de generación de código de esta sección genera código objeto para una secuencia de proposiciones de tres direcciones. Considera cada proposición por turno, teniendo en cuenta si los operandos de la proposición están en registros, y aprovechando esa situación si es posible. Para simplificar, se supone que para cada operador dentro de una proposición hay un operador correspondiente en el lenguaje objeto. También se supone que los resultados calculados se pueden dejar en registros mientras sea posible, almacenándolos en la memoria sólo si (a) su registro es necesario para otro cálculo, o (b) se está justo antes de una llamada a un procedimiento, un salto o una proposición etiquetada[5].

La condición (b) implica que todo debe ser almacenado en la memoria justo antes del final de un bloque básico[6] porque después de abandonar un bloque básico se puede ir a varios bloques diferentes, o a un bloque determinado que pueda alcan-

---

[5] Sin embargo, para producir un *volcado simbólico*, que hace disponibles los valores de las posiciones de memoria y los registros referentes a los nombres definidos en el programa fuente para dichos valores, puede ser preferible tener las variables definidas por el programador (pero no necesariamente temporales generados por el compilador) almacenadas en la memoria en el momento del cálculo, por si un error de programa produce de repente la interrupción precipitada y la salida de la ejecución.

[6] Obsérvese que no se supone que los cuádruplos fueron realmente particionados en bloques básicos por el compilador; la noción de un bloque básico es útil conceptualmente en cualquier caso.

zarse desde otros bloques. En cualquier caso, no se puede suponer sin un esfuerzo adicional que un dato utilizado por un bloque aparece en el mismo registro independientemente de cómo alcanzó el control dicho bloque. Por tanto, para evitar un posible error, el algoritmo de generación de código simple guarda todo en memoria cuando se traslada a través de los límites de los bloques básicos así como cuando se hacen llamadas a procedimientos. Más adelante se consideran formas de conservar algunos datos dentro de los registros a través de los límites de los bloques.

Se puede producir un código razonable para una proposición de tres direcciones `a:=b+c` si se genera la instrucción simple `ADD Rj,Ri` con costo uno, dejando el resultado `a` en el registro `Ri`. Está secuencia sólo es posible si el registro `Ri` contiene `b`, `Rj` contiene `c` y `b` no está activa después de la proposición; es decir, `b` no se utiliza después de la proposición.

Si `Ri` contiene `b` pero `c` está en una posición de memoria (llamada `c` por conveniencia), se puede generar la secuencia

```
ADD    c, Ri          costo = 2
```

o

```
MOV    c, Rj
ADD    Rj, Ri         costo = 3
```

suponiendo que `b` no está activa posteriormente. La segunda secuencia se vuelve atractiva si este valor de `c` se utiliza posteriormente porque entonces se puede tomar su valor del registro `Rj`. Se pueden considerar muchos más casos, dependiendo de dónde se encuentren localizados en ese momento `b` y `c` y de si el valor en curso de `b` se utiliza posteriormente. También se deben considerar los casos donde uno o ambos de `b` y `c` son constantes. El número de casos que hay que considerar se incrementa aún más si se supone que el operador + es conmutativo. Por tanto, se ve que la generación de código implica examinar una gran cantidad de casos, y el caso que debe prevalecer depende del contexto en el que aparezca una proposición de tres direcciones.

Descriptores de registros y direcciones

El algoritmo de generación de código utiliza descriptores para seguir de cerca el contenido de los registros y las direcciones para los nombres.

1. Un descriptor de registros sabe lo que hay en cada registro. Es consultado siempre que se necesite un nuevo registro. Se supone que inicialmente el descriptor de registros muestra que todos los registros están vacíos. (Si los registros se asignan entre los bloques, este no sería el caso.) Conforme avanza la generación de código para el bloque, cada registro contiene siempre el valor de cero o más nombres.

2. Un descriptor de direcciones conoce la posición (o posiciones) donde se puede encontrar el valor en curso del nombre durante la ejecución. La posición puede ser un registro, una posición en la pila, una dirección de memoria o conjunto de éstos porque cuando se copia, un valor también permanece donde estaba. Esta

información se puede almacenar en la tabla de símbolos y se utiliza para determinar el método de acceso a un nombre.

Un algoritmo para generación de código

El algoritmo para la generación de código toma como entrada una secuencia de proposiciones de tres direcciones que constituyen un bloque básico. Para cada proposición de tres direcciones de la forma `x := y` *op* `z` se realizan las siguientes operaciones:

1. Se invoca la función *obtenreg* para determinar la posición `L` donde se debe guardar el resultado del cálculo `y` *op* `z`. Generalmente `L` será un registro, pero también puede ser una posición de memoria. Dentro de poco se describirá *obtenreg*.
2. Se consulta el descriptor de direcciones de `y` para determinar `y'`, (una de) la(s) posición(es) en curso de `y`. Se prefiere el registro para `y'` si el valor de `y` está en ese momento en memoria y en un registro. Si el valor de `y` no está todavía en `L`, se genera la instrucción `MOV y', L` para colocar una copia de `y` en `L`.
3. Se genera la instrucción `OP z', L` donde `z'` es una posición en curso de `x`. De nuevo, se prefiere un registro a una posición de memoria si `z` se encuentra en ambos. Se actualiza el descriptor de direcciones de `x` para indicar que `x` está en la posición `L`. Si `L` es un registro, se actualiza su descriptor para indicar que contiene el valor de `x`, y se elimina `x` de todos los otros descriptores de registros.
4. Si los valores en curso de `y` o `z`, o ambos, no tienen usos siguientes, no están activos a la salida del bloque, y están en registros, se altera el descriptor de registros para indicar que después de la ejecución de `x := y` *op* `z`, estos registros ya no contendrán `y` o `z`, o ambos, respectivamente.

Si la proposición de tres direcciones en curso tiene un operador unario, los pasos son análogos a los anteriores, y se omiten los detalles. Un caso especial importante es una proposición de tres direcciones `x := y`. Si `y` está en un registro, simplemente se cambian los descriptores de registros y de direcciones para consignar que el valor de `x` ahora sólo se encuentra en el registro que contiene el valor de `y`. Si `y` no tiene uso siguiente y no está activo a la salida de un bloque, el registro ya no contiene el valor de `y`.

Si `y` sólo se encuentra en memoria, en principio se podría hacer constar que el valor de `x` está en la posición de `y`, pero esta opción complicaría el algoritmo, porque entonces no se podría modificar el valor de `y` sin preservar el valor de `x`. Por tanto, si `y` se encuentra en memoria, se utiliza *obtenreg* para encontrar un registro en el que cargar `y` y convertir ese registro en la posición de `x`.

También se puede generar una instrucción `MOV y, x`, que sería preferible si el valor de `x` no tuviera uso siguiente en el bloque. Vale la pena resaltar que casi todas, si no todas, las instrucciones de copia quedarán eliminadas si se utiliza el algoritmo de mejora de bloques y de copia y propagación del capítulo 10.

Una vez que se hayan procesado todas las proposiciones de tres direcciones en los bloques básicos, mediante instrucciones `MOV`, se almacenan aquellos nombres que estén activos a la salida y no en sus posiciones de memoria. Para ello se utiliza el

descriptor de registros para determinar qué nombres se dejan en los registros, el descriptor de direcciones para determinar que el mismo nombre no está ya en su posición de memoria y la información de las variables activas para determinar si se va a almacenar el nombre. Si no se ha calculado la información sobre variables activas mediante el análisis del flujo de datos entre los bloques, se debe asumir que todos los nombres definidos por el usuario están activos al final del bloque.

La función *obtenreg*

La función *obtenreg* devuelve la posición L para guardar el valor de x para la asignación x := y *op* x. Se puede emplear un gran esfuerzo para implantar esta función de modo que produzca una opción inteligente para L. En esta sección se estudia un esquema simple y fácil de implantar basado en la información sobre el uso siguiente recogida en la última sección.

1. Si el nombre y está en un registro que no contiene el valor de otros nombres (recuérdese que las instrucciones de copia como x := y podrían hacer que un registro guardara el valor de dos o más variables simultáneamente), e y no está activa y no tiene uso siguiente después de la ejecución de x := y *op* z, entonces se devuelve el registro de y para L. Se actualiza el descriptor de direcciones de y para indicar que y ya no se encuentra en L.
2. Si falla (1), devuélvase un registro vacío para L si hay alguno.
3. Si falla (2), si x tiene un uso siguiente en el bloque, u *op* es un operador, como indizar, que exige un registro, encuéntrese un registro ocupado R. Almacénese el valor de R en una posición de memoria (mediante MOV R,M) si es que todavía no está en una posición de memoria apropiada M, actualícese el descriptor de direcciones de M y devuélvase R. Si R contiene el valor de varias variables, se debe generar una instrucción MOV para cada variable que haya que almacenar. Un registro ocupado adecuado puede ser uno cuyo dato sea el referenciado más lejos en el futuro, o uno cuyo valor también esté en memoria. No se especifica la opción exacta porque no se conoce mejor forma de hacer la selección.
4. Si no se utiliza x en el bloque, o no se puede encontrar ningún registro ocupado adecuado, selecciónese la posición de memoria de x como L.

Una función *obtenreg* más sofisticada también consideraría los usos subsiguientes de x y la conmutatividad del operador *op* al determinar el registro que contendrá al valor de x. Se dejan como ejercicios prácticos dichas ampliaciones de *obtenreg*.

Ejemplo 9.5. La asignación d := (a-b) + (a-c) + (a-c) se puede traducir a la siguiente secuencia de código de tres direcciones:

```
t := a - b
u := a - c
v := t + u
d := v + u
```

con d activa al final. El algoritmo de generación de código anterior produciría la secuencia de código de la figura 9.10 para esta secuencia de proposiciones de tres

direcciones. A un lado se muestran los valores de los descriptores de registros y de direcciones conforme avanza la generación de código. En el descriptor de direcciones no se indica que a, b y c están siempre en memoria. También se asume que t, u y v, siendo temporales, no están en memoria a menos que se almacenen explícitamente sus valores con una instrucción MOV.

PROPOSICIONES	CÓDIGO GENERADO	DESCRIPTOR DE REGISTROS	DESCRIPTOR DE DIRECCIONES
		registros vacíos	
t := a - b	MOV a, R0 SUB b, R0	R0 contiene a t	t en R0
u := a - c	MOV a, R1 SUB c, R1	R0 contiene a t R1 contiene a u	t en R0 u en R1
v := t + u	ADD R1, R0	R0 contiene a v R1 contiene a u	u en R1 v en R0
d := v + u	ADD R1, R0	R0 contiene a d	d en R0
	MOV R0, d		d en R0 y en memoria

Fig. 9.10. Secuencia de código.

La primera llamada a *obtenreg* devuelve R0 como la posición en la que calcular t. Como a no está en R0, se generan las instrucciones MOV a,R0 y SUB b,R0. Ahora se actualiza el descriptor de registros para indicar que R0 contiene t.

La generación de código continúa de esta manera hasta que se haya procesado la última proposición de tres direcciones d := v + u. Obsérvese que R1 queda vacío porque u no tiene uso siguiente. Entonces se genera MOV R0,d para almacenar la variable activa d al final del bloque.

El costo del código generado en la figura 9.10 es 12. Se podría reducir a 11 generando MOV R0,R1 inmediatamente después de la primera instrucción y eliminando la instrucción MOV a,R1 pero hacer esto exige un algoritmo de generación de código más complicado. La razón de los ahorros es que resulta más barato cargar R1 desde R0 que desde memoria. □

Generación de código para otros tipos de proposiciones

Las operaciones de indización y de apuntamiento en las proposiciones de tres direcciones se tratan de la misma manera que las operaciones binarias. La tabla de la figura 9.11 muestra las secuencias de código generadas por las proposiciones de asignaciones indizadas a := b[i] y a[i] := b, suponiendo que b está asignada estáticamente.

PROPOSICIÓN	i EN EL REGISTRO Ri		i EN MEMORIA Mi		i EN LA PILA	
	CÓDIGO	COSTO	CÓDIGO	COSTO	CÓDIGO	COSTO
a := b[i]	MOV b(Ri),R	2	MOV Mi,R MOV b(R),R	4	MOV Si(A),R MOV b(R),R	4
a[i] := b	MOV b,a(Ri)	3	MOV Mi,R MOV b,a(R)	5	MOV Si(A),R MOV b,a(R)	5

Fig. 9.11. Secuencias de código para asignaciones con índices.

La posición en curso de i determina la secuencia de código. Se consideran tres casos dependiendo de si i está en el registro Ri, en la posición de memoria M, o en la pila con desplazamiento Si y de si el apuntador al registro de activación de i está en el registro A. El registro R es el registro devuelto cuando se llama a la función *obtenreg*. Para la primera asignación, se preferiría dejar a en el registro R si a tiene un uso siguiente en el bloque y el registro R está disponible. En la segunda asignación, se supone que a tiene asignación estática.

La tabla de la figura 9.12 muestra las secuencias de código generadas por las asignaciones de apuntadores a := *p y *p := a. En este caso, la localidad en curso de p determina la secuencia de código.

PROPOSICIÓN	p EN EL REGISTRO Rp		p EN MEMORIA Mp		p EN LA PILA	
	CÓDIGO	COSTO	CÓDIGO	COSTO	CÓDIGO	COSTO
a := *p	MOV *Rp,a	2	MOV Mp,R MOV *R,R	3	MOV Sp(A),R MOV *R,R	3
*p := a	MOV a,*Rp	2	MOV Mp,R MOV a,*R	4	MOV a,R MOV R,*Sp(A)	4

Fig. 9.12. Secuencias de código para asignaciones con apuntadores.

Se consideran tres casos dependiendo de si p se encuentra inicialmente en el registro Rp, en la posición de memoria Mp, o en la pila con desplazamiento Sp y de si el apuntador al registro de activación de p está en el registro A. El registro R es el registro devuelto cuando se llama a la función *obtenreg*. En la segunda asignación se supone que a tiene asignación estática.

Proposiciones condicionales

Las máquinas implantan los saltos condicionales en una de dos formas. Una manera es saltar si el valor de un registro designado cumple una de seis condiciones: negativo, cero, positivo, no negativo, no cero y no positivo. En dicha máquina se puede implantar una proposición de tres direcciones como if x<y goto z restando y a x en el registro R, y saltando después a z si el valor del registro R es negativo.

Un segundo enfoque, común a muchas máquinas, utiliza un conjunto de *códigos de condición* para indicar si la última cantidad calculada o cargada en un registro es negativa, cero o positiva. A menudo una instrucción de comparación (CMP en la máquina de este capítulo) tiene la ventaja de que asigna el código de condición sin calcular un valor. Es decir, `CMP x,y` pone el código de condición en positivo si `x>y`, y así sucesivamente. Una instrucción de máquina de salto condicional realiza el salto si se cumple una condición designada $<$, $=$, $>$, $\leqslant$, $\neq$, o $\geqslant$. Se utiliza la instrucción `CJ<= z` para indicar que "salta a `z` si el código de condición es negativo o cero". Por ejemplo, `if x<y goto z` se podría implantar mediante

```
CMP    x, y
CJ<    z
```

Si se está generando código para una máquina con códigos de condición es útil mantener un descriptor de códigos de condiciones cuando se genera el código. Este descriptor indica el último nombre en asignar el código de condición, o el par de nombres comparados, si el código de condición se asignó por última vez de esta forma. Por tanto, se podría implantar

```
x := y + z
if x < 0 goto z
```

mediante

```
MOV    y, R0
ADD    z, R0
MOV    R0, x
CJ<    z
```

si se supiera que el código de condición fue determinado por `x` después de `ADD z,R0`.

9.7 DISTRIBUCION Y ASIGNACION DE REGISTROS

Las instrucciones que implican sólo operandos en registros son más cortas y rápidas que las de operandos en memoria. Por tanto, es importante utilizar eficientemente los registros para generar de buen código. Esta sección propone varias estrategias para decidir los valores de un programa que deben residir en registros (asignación de los registros) y en qué registro debe residir cada valor (asignación a registros).

Un aspecto de la distribución y asignación de los registros es asignar valores específicos en un programa objeto a algunos registros. Por ejemplo, se puede tomar la decisión de asignar las direcciones base a un grupo de registros, los cálculos aritméticos a otro, el tope de la pila de ejecución a un registro fijo, y así sucesivamente.

Este enfoque tiene la ventaja de que simplifica el diseño de un compilador. El inconveniente es que, si se aplica con demasiada rigidez, no utiliza eficientemente los registros; algunos registros pueden quedar sin utilizar durante partes substanciales de código, mientras que se generan cargas y almacenamientos innecesarios. Sin embargo, en la mayoría de los entornos de cálculo es razonable reservar algunos re-

gistros para los registros base, apuntadores a la pila, etc., y permitir que el compilador utilice los restantes registros como crea conveniente.

Distribución de registros globales

El algoritmo de generación de código de la sección 9.6 utilizaba registros para guardar valores en el tiempo de duración de un bloque básico. Sin embargo, todas las variables activas se almacenaban en memoria al final de cada bloque. Para evitar algunos de estos almacenamientos y cargas correspondientes, se puede decidir asignar los registros a las variables utilizadas con frecuencia y mantener consistentes estos registros a través de los límites de los bloques (*globalmente*). Como los programas pasan la mayor parte de su tiempo en lazos internos, un enfoque natural de la distribución global de registros es int ntar conservar un valor utilizado con frecuencia en un registro fijo durante todo un lazo. Por el momento, supóngase que se conoce la estructura de lazo de un grafo de flujo y los valores calculados en un bloque básico que se utilizan fuera de ese bloque. El siguiente capítulo estudia las técnicas para calcular esta información.

Una estrategia para la distribución global de registros es asignar un número fijo de registros para guardar los valores más activos de cada lazo interno. Los valores seleccionados pueden ser distintos en lazos distintos. Se pueden utilizar los registros que todavía no se hayan asignado para conservar los valores locales a un bloque, como en la sección 9.6. Este enfoque tiene el defecto de que el número fijo de registros no siempre es el número correcto de registros disponibles para la asignación global de registros. Sin embargo, el método es sencillo de implantar y se ha usado en FORTRAN H, el compilador optimador de FORTRAN para las máquinas de la serie IBM-360 (Lowry y Medlock [1969]).

En lenguajes como C y BLISS, un programador puede realizar una asignación de registros directamente utilizando declaraciones de registros para conservar algunos valores dentro de los registros para la duración de un procedimiento. Un uso inteligente de las declaraciones de registros puede acelerar muchos programas, pero un programador no debe realizar la asignación de los registros sin perfilar primero su programa.

Cuentas de uso

Un método sencillo para determinar los ahorros obtenidos al conservar la variable x en un registro durante el tiempo del lazo L es reconocer que en este modelo de máquina se ahorra una unidad de costo por cada referencia a x si x está en un registro. Sin embargo, si se utiliza el enfoque de la sección anterior para generar código para un bloque, es muy probable que después de haber calculado x en un bloque permanecerá en un registro si hay usos posteriores de x en dicho bloque. Por tanto, se contabiliza un ahorro de uno por cada uso de x dentro del lazo L que no venga precedido de una asignación a x en el mismo bloque. Así, si se asigna un registro a x, se contabiliza un ahorro de dos por cada bloque de L para el que x esté activo a la salida y en el que a x se le asigna un valor.

Por parte del débito, si x está activa en la entrada al encabezamiento del lazo, se debe cargar x en un registro justo antes de entrar en el lazo L. Esta carga cuesta dos

unidades. De manera similar, para cada salida de un bloque B del lazo L en el que x esté activo en la entrada a algún sucesor de B fuera de L, se debe almacenar en memoria x con un costo de dos. Sin embargo, en el supuesto de que el lazo se repita muchas veces, se pueden ignorar estos débitos porque ocurren sólo una vez cuando se entra al lazo. Por tanto, una fórmula aproximada para la ventaja obtenida al asignar un registro a x dentro del lazo L es:

$$\sum_{\text{bloques } B \text{ en } L} (\mathit{uso}(x, B) + 2*\mathit{activo}(x, B)) \tag{9.4}$$

donde *uso*(x,B) es el número de veces que se utiliza x en B antes de cualquier definición de x; *activo*(x,B) es 1 si x está activo a la salida de B y se le asigna un valor en B, y en caso contrario es 0. Obsérvese que (9.4) es aproximada, porque no todos los bloques en un lazo se ejecutan con la misma frecuencia y también porque (9.4) se basaba en el supuesto de que un lazo se repite "muchas" veces. En otras máquinas, habrá que desarrollar una fórmula análoga a (9.4), aunque posiblemente bastante distinta de ella.

Ejemplo 9.6. Considérense los bloques básicos dentro del lazo interno representado en la figura 9.13, donde se han omitido proposiciones de salto y salto condicional. Supóngase que se asignan los registros `R0`, `R1` y `R2` para guardar valores a través del lazo. Las variables activas a la entrada y a la salida de cada bloque se muestran por conveniencia en la figura 9.13, inmediatamente por encima y por debajo de cada bloque, respectivamente. En el capítulo 10 se señalan algunos puntos sutiles sobre las variables activas. Por ejemplo, obsérvese que tanto `e` como `f` están activos al final de B_1, pero sólo `e` está activo a la entrada de B_2 y sólo `f` está activo a la entrada de B_3. En general, las variables activas al final de un bloque son la unión de las variables activas al comienzo de cada uno de sus bloques sucesores.

Para evaluar (9.4) para $x =$ a se observa que a está activa a la salida de B_1 y se le asigna un valor allí, pero no está activa a la salida de B_2, B_3 o B_4. Por tanto, $\sum_{B \text{ en } L} 2*\mathit{activo}(\mathrm{a},B) = 2$. Asimismo, *uso* $(\mathrm{a},B_1) = 0$, porque a se define en B_1 antes

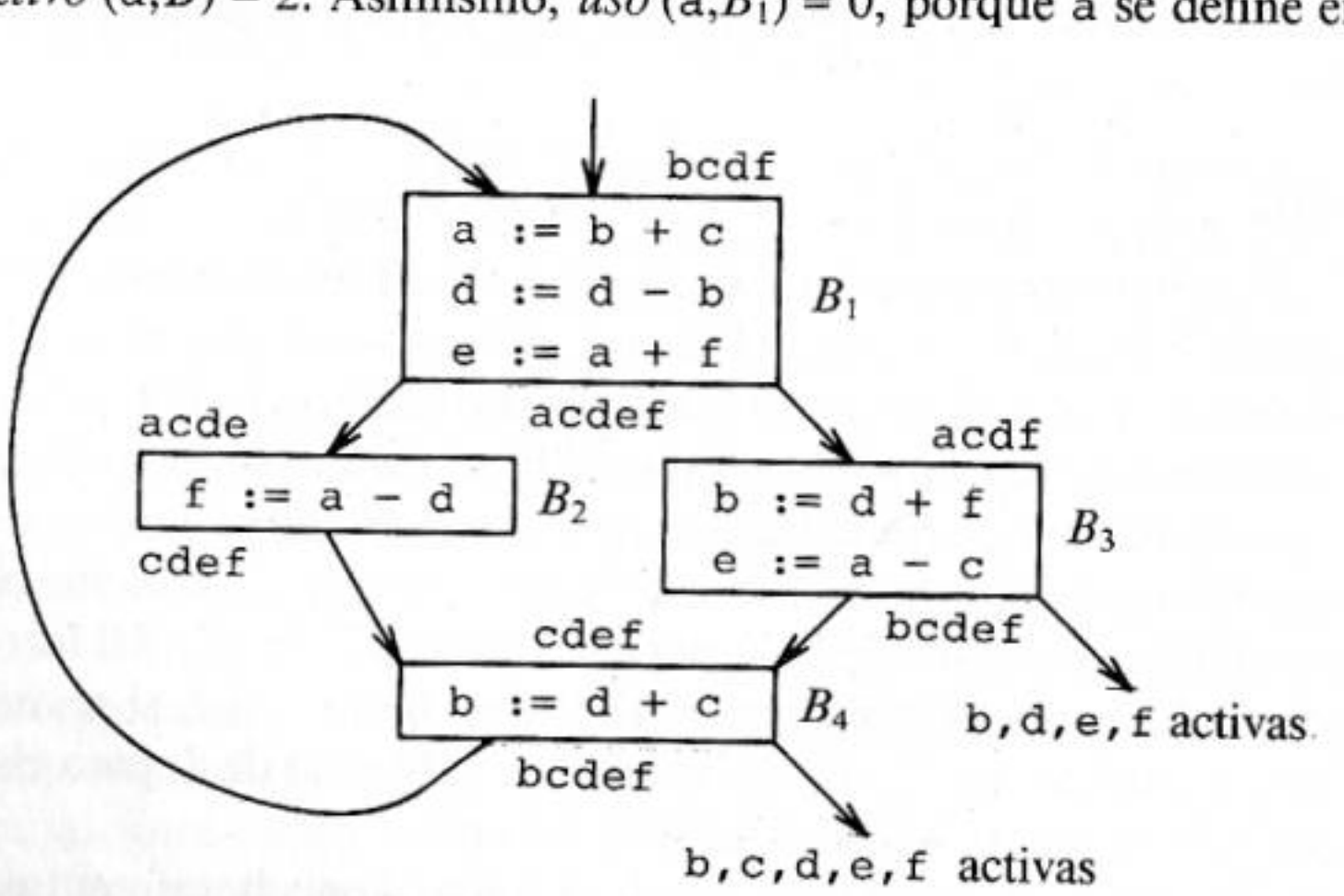


Fig. 9.13. Grafo de flujo para un lazo interno.

de cualquier uso. También, $uso\,(a,B_2) = uso\,(a,B_3) = 1$ y $uso\,(a,B_4) = 0$. Por tanto, $\sum_{B \text{ en } L} uso\,(a,B) = 2$. Así, el valor de (9.4) para x = a es 4. Es decir, se pueden ahorrar cuatro unidades de costo seleccionando a para uno de los registros globales. Los valores de (9.4) para b, c, d, e y f son 6, 3, 6, 4 y 4, respectivamente. De ese modo se pueden seleccionar a, b y d para los registros R0, R1 y R2, respectivamente. Utilizar R0 para e o f en lugar de para a sería otra opción con la misma ventaja aparente. En la figura 9.14 se muestra el código ensamblador generado a partir de la figura 9.13, suponiendo que se utiliza la estrategia de la sección 9.6 para generar código para cada bloque. No se muestra el código generado para los saltos condicionales o incondicionales que finalizan cada bloque de la figura 9.13, y por tanto no se muestra el código generado como secuencia simple, como aparecería en la práctica. Se hace observar que, si no se ciñe uno a la estrategia de reservar R0, R1 y R2, se podría utilizar

```
SUB    R2, R0
MOV    R0, f
```

para B_2, ahorrando una unidad ya que a no está activo a la salida de B_2. Se podría realizar en B_3 un ahorro similar. □

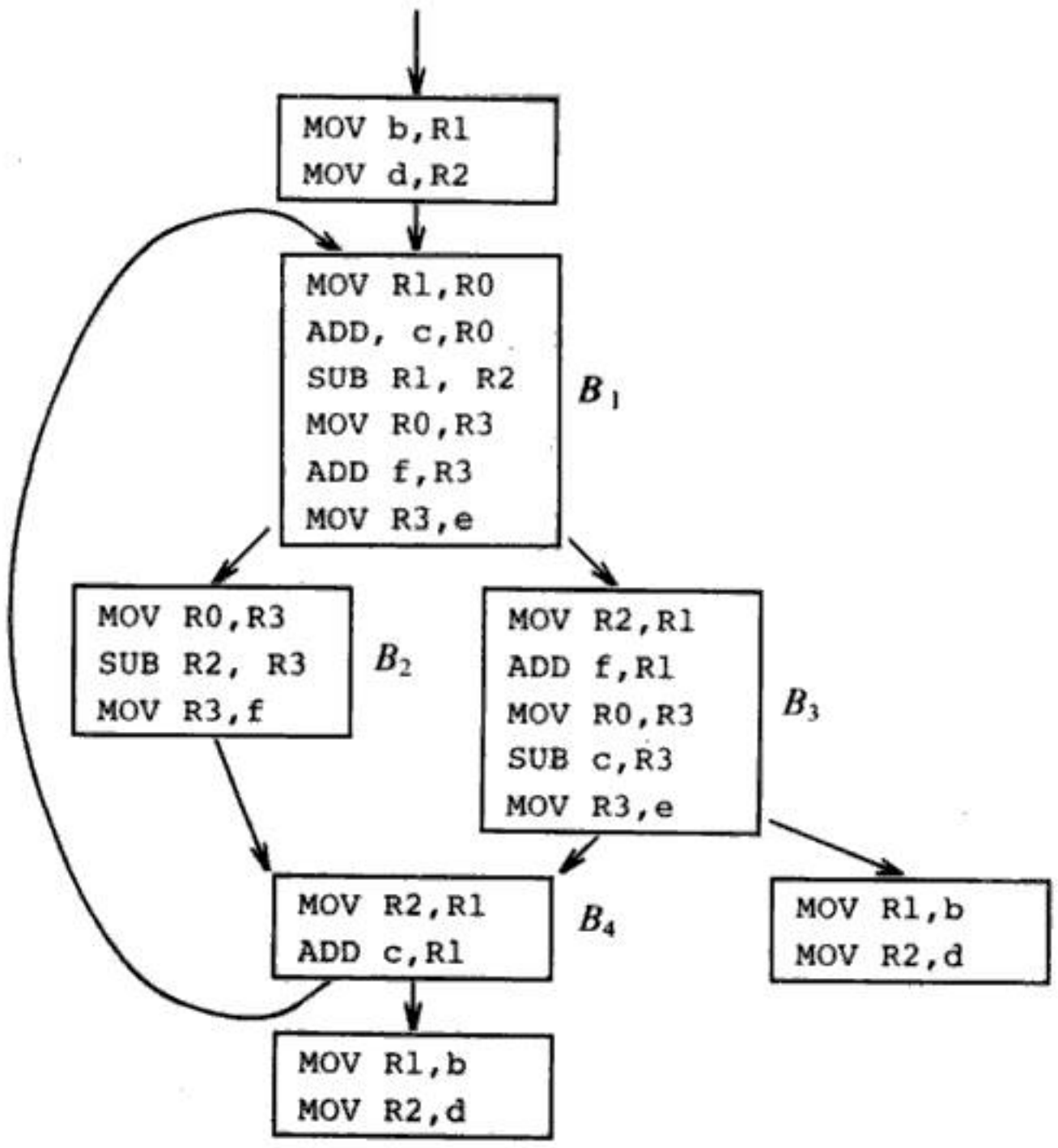


Fig. 9.14. Secuencia de código que usa asignación a registros globales.

Asignación a registros para lazos externos

Habiendo asignado registros y generado código para los lazos internos, se puede aplicar la misma idea para lazos cada vez más grandes. Si un lazo externo L_1 contiene un lazo interno L_2, no hay que asignar registros en $L_1 - L_2$ a los nombres a los que se les asignaron registros en L_2. Sin embargo, si al nombre x se le asignó un registro en el lazo L_1 pero no en L_2, se debe almacenar x a la entrada de L_2 y cargar x si se abandona L_2 y se entra a un bloque de $L_1 - L_2$. De manera similar, si se elige asignar a x un registro en L_2 pero no en L_1, se debe cargar x a la entrada de L_2 y almacenar x en memoria a la salida de L_2. Se deja como ejercicio práctico la obtención de un criterio para seleccionar los nombres a los que se les asignarán registros en un lazo externo L, dado que ya se ha hecho la selección para todos los lazos anidados dentro de L.

Asignación de los registros mediante coloración de grafos

Cuando se necesita un registro para un cálculo pero todos los registros disponibles están siendo utilizados, se debe almacenar (*vaciar*) el contenido de uno de los registros utilizados en una posición de memoria para dejar libre un registro. La coloración de grafos es una técnica sistemática y sencilla para asignar registros y administrar el vaciado de los registros.

En este método se utilizan dos pasadas. En la primera, se seleccionan instrucciones de la máquina objeto como si hubiera un número infinito de registros simbólicos; en efecto, los nombres utilizados en el código intermedio se convierten en nombres de registros y las proposiciones de tres direcciones se convierten en proposiciones en el lenguaje de máquina. Si el acceso a las variables exige instrucciones que utilicen apuntadores a la pila, apuntadores al *display*, registros de base u otras cantidades que faciliten el acceso, entonces se supone que estas cantidades se conservan en registros reservados para cada propósito. Normalmente su uso es directamente traducible a un modo de acceso para una dirección mencionada en una instrucción de máquina. Si el acceso es más complejo, éste se puede dividir en varias instrucciones de máquina, y puede ser necesario crear un registro simbólico temporal (o varios).

Una vez seleccionadas las instrucciones, una segunda pasada asigna registros físicos a los registros simbólicos. El objetivo es encontrar una asignación que minimice el costo de los vaciados de registros.

En la segunda pasada se construye para cada procedimiento un *grafo de interferencia entre registros* en el que los nodos son registros simbólicos y una arista conecta dos nodos si uno está activo en un punto donde se define el otro. Por ejemplo, un grafo de interferencia entre registros para la figura 9.13 tendría nodos para los nombres a y d. En el bloque B_1, a está activa en la segunda proposición, que define a d; por tanto, en el grafo habría una arista entre los nodos de a y d.

Se intenta colorear el grafo de interferencia entre registros utilizando k colores, donde k es el número de registros asignables. (Se dice que un grafo está *coloreado* si a cada nodo se le ha asignado un color de modo que no haya dos nodos adyacentes con el mismo color.) Un color representa un registro y la coloración garantiza que no se asigna el mismo registro físico a dos registros simbólicos que pueden interferir uno con el otro.

Aunque el problema de determinar si un grafo es k-coloreable es generalmente NP-completo, se puede utilizar la siguiente técnica heurística en la práctica para realizar el coloreado rápidamente. Supóngase que un nodo n en un grafo G tiene menos que k vecinos (nodos conectados a n por una arista). Elimínese n y sus aristas de G para obtener un grafo G'. Un k-coloreado de G' se puede ampliar a un k-coloreado de G asignando a n un color no asignado a ninguno de sus vecinos.

Eliminando repetidamente el grafo de interferencias entre registros los nodos con menos de k aristas, se obtiene el grafo vacío, en cuyo caso se puede producir un k-coloreado del grafo original coloreando los nodos en orden inverso al que fueron eliminados o bien se obtiene un grafo en el que cada nodo tiene k o más nodos adyacentes. En este último caso, ya no es posible un k-coloreado. Llegados a este punto, un nodo se vacía introduciendo código para almacenar en memoria y recargar el registro. Entonces se modifica adecuadamente el grafo de interferencias y se continúa el proceso de coloreado. Chaitin [1982] y Chaitin y colaboradores [1981] describen varias heurísticas para elegir el nodo que se va a vaciar. Como regla general se evita introducir código para vaciar registros en lazos internos.

9.8 REPRESENTACION DE BLOQUES BASICOS POR MEDIO DE GDA

Los grafos acíclicos dirigidos (GDA) son estructuras de datos útiles para implantar transformaciones en bloques básicos. Un GDA proporciona una imagen de cómo el valor calculado por cada proposición en un bloque básico se usa en subsiguientes proposiciones del bloque. Construir un GDA a partir de proposiciones de tres direcciones es una buena manera de determinar subexpresiones comunes (expresiones calculadas más de una vez) dentro de un bloque, determinando los nombres que se utilizan dentro del bloque pero que se evalúan fuera de él, y determinando qué proposiciones del bloque podrían utilizar su valor calculado fuera del bloque.

Un *GDA para un bloque básico* (o simplemente *GDA*) es un grafo dirigido acíclico con las siguientes etiquetas en los nodos:

1. Las hojas se etiquetan con identificadores únicos, ya sean nombres de variables o constantes. Según el operador aplicado a un nombre, se determina si se necesita el valor de lado izquierdo o de lado derecho; la mayoría de las hojas representan valores de lado derecho. Las hojas representan valores iniciales de nombres, y se les da como subíndice 0 para evitar confundirlas con las etiquetas que indican valores "en curso" de nombres como en (3), más adelante.
2. Los nodos interiores se etiquetan con un símbolo de operador.
3. Opcionalmente, también se les asigna a los nodos una secuencia de identificadores para las etiquetas. La intención es que los nodos interiores representen valores calculados y se considera que los identificadores que etiquetan un nodo tienen ese valor.

Es importante no confundir los GDA con grafos de flujo. Cada nodo de un grafo se puede representar mediante un GDA, porque cada nodo del grafo de flujo representa un bloque básico.

```
(1)    t1 := 4 * i
(2)    t2 := a [ t1 ]
(3)    t3 := 4 * i
(4)    t4 := b [ t3 ]
(5)    t5 := t2 * t4
(6)    t6 := prod + t5
(7)    prod := t6
(8)    t7 := i + 1
(9)    i := t7
(10)   if i <= 20 goto (1)
```

Fig. 9.15. Código de tres direcciones para el bloque B_2.

Ejemplo 9.7. En la figura 9.15 se muestra el código de tres direcciones que corresponde al bloque B_2 de la figura 9.9. Se han utilizado por conveniencia los números de las proposiciones, comenzando por (1). En la figura 9.16 se muestra el GDA correspondiente. Se analiza el significado del GDA después de dar un algoritmo para construirlo. Por el momento, obsérvese que cada nodo del GDA representa una fórmula referente a las hojas, es decir, los valores que poseen las variables y constantes al entrar al bloque. Por ejemplo, el nodo etiquetado con t_4 de la figura 9.16 representa la fórmula

```
b [4 * i]
```

es decir, el valor de la palabra cuya dirección está desplazada `4*i` bytes de la dirección `b`, que es el valor que corresponde a t_4. □

Construcción de un GDA

Para construir un GDA para un bloque básico, se procesa cada proposición del bloque por turno. Cuando aparece una proposición de la forma `x := y+z` se buscan los nodos que representan los valores "en curso" de `y` y `z`. Estos podrían ser hojas, o nodos interiores del GDA si `y` o `z` o ambos hubieran sido evaluados por proposiciones anteriores del bloque. Después se crea un nodo etiquetado con + y se le dan

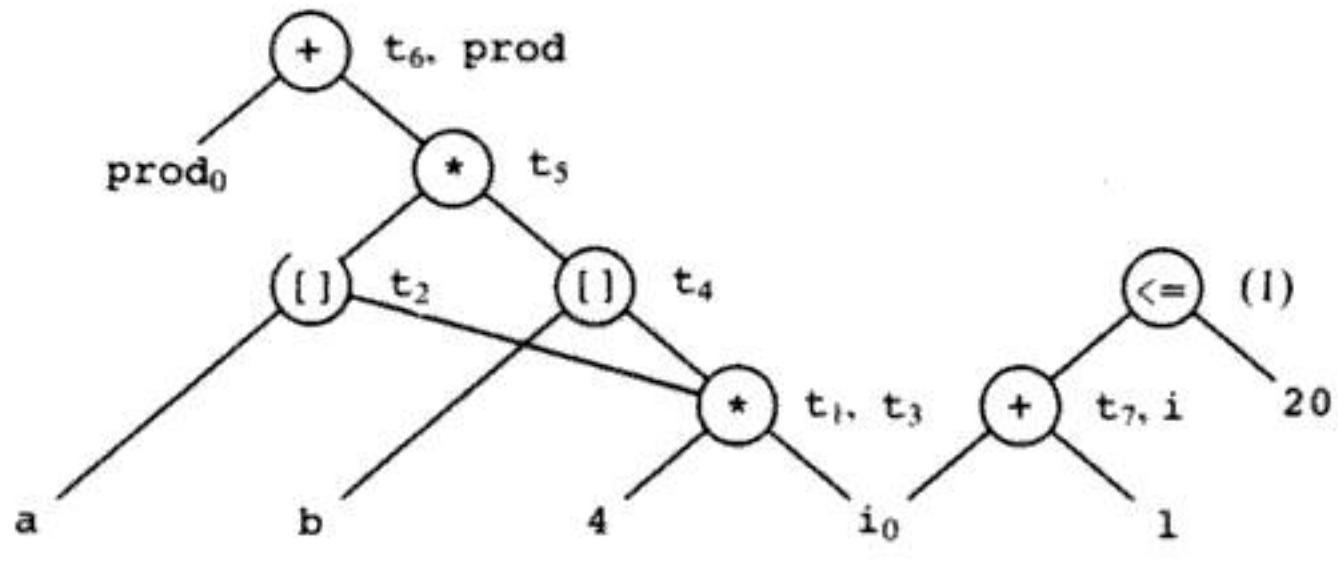


Fig. 9.16. GDA para el bloque de la figura 9.15.

dos hijos; el hijo izquierdo es el nodo correspondiente a y, el derecho es el nodo correspondiente a z. Después se etiqueta ese nodo con x. Sin embargo, si ya existe un nodo que indique el mismo valor que y+z, no se añade el nuevo nodo al GDA, sino que al nodo existente se le proporciona la etiqueta adicional x.

Se deben mencionar dos detalles. Primero, si x (no x_0) había etiquetado previamente algún otro nodo, se elimina dicha etiqueta, puesto que el valor "en curso" de x es el nodo recién creado. Segundo, para una asignación como x := y no se crea un nuevo nodo sino que se añade la etiqueta x a la lista de nombres en el nodo creado para el valor "en curso" de y.

Ahora se proporciona el algoritmo para calcular un GDA a partir de un bloque. El algoritmo es casi el mismo que el algoritmo 5.1, excepto la lista adicional de identificadores que se asocia aquí a cada nodo. Se debe advertir al lector que este algoritmo puede no operar correctamente si hay asignaciones a matrices, si hay asignaciones indirectas a través de apuntadores, o si se puede hacer referencia a una posición de memoria con dos o más nombres, debido a proposiciones EQUIVALENCE o a las correspondencias entre parámetros reales y formales de la llamada a un procedimiento. Al final de esta sección se estudian las modificaciones necesarias para tratar estas situaciones.

Algoritmo 9.2. Construcción de un GDA.

Entrada. Un bloque básico.

Salida. Un GDA para el bloque básico que contiene la siguiente información:

1. Una *etiqueta* para cada nodo. Para las hojas, la etiqueta es un identificador (se permiten constantes) y para los nodos interiores, un símbolo de operador.
2. Para cada nodo, una lista (posiblemente vacía) de identificadores asociados (aquí no se permiten constantes).

Método. Se supone que están disponibles las estructuras de datos adecuadas para crear nodos con uno o dos hijos, distinguiendo entre el hijo "izquierdo" y "derecho" en el último caso. También hay un lugar disponible en la estructura para una etiqueta para cada nodo y la posibilidad de crear una lista enlazada de los identificadores asociados a cada nodo.

Además de estos componentes, hay que mantener el conjunto de todos los identificadores (incluidas constantes) para los cuales hay un nodo asociado. El nodo podría ser una hoja etiquetada con un identificador o un nodo interior con dicho identificador en su lista de identificadores asociados. Se supone la existencia de una función, *nodo*(*identificador*) que, conforme se construye el GDA, devuelve el nodo más recientemente creado asociado con *identificador*. Intuitivamente, *nodo*(*identificador*) es el nodo del GDA que representa el valor que tiene *identificador* en el punto en curso en el proceso de construcción del GDA. En la práctica, una entrada en el registro de la tabla de símbolos para *identificador* indicaría el valor de *nodo*(*identificador*).

El proceso de construcción del GDA consiste en realizar los siguientes pasos, 1 al 3, para cada proposición del bloque por turno. Al inicio se supone que no hay nodos, y la función *nodo* está sin definir para todos los argumentos. Supóngase que

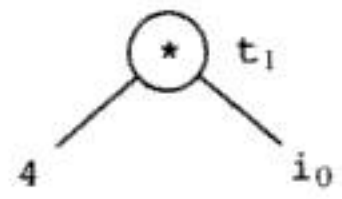
la proposición de tres direcciones "en curso" es (i) `x := y` *op* `z`, (ii) `x :=` *op* `y`, o (iii) `x := y`[7]. Se citarán como los casos (i), (ii) y (iii). Un operador relacional como `if i <= 20 goto` se considera como caso (i), con `x` indefinida.

1. Si *nodo*(`y`) está indefinida, créese una hoja etiquetada con `y`, y que *nodo*(`y`) sea este nodo. En el caso (i), si *nodo*(`z`) está indefinida, créese una hoja etiquetada con `z` y que esa hoja sea *nodo*(`z`).

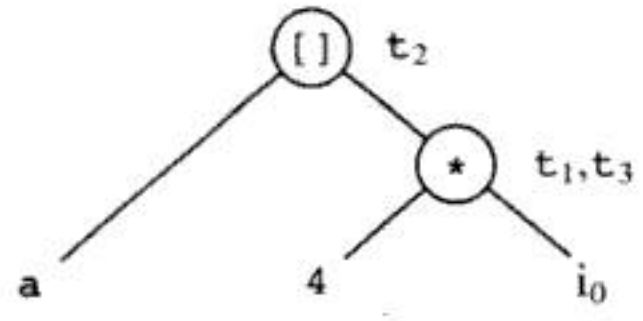
2. En el caso (i), determínese si hay un nodo etiquetado con *op*, cuyo hijo izquierdo sea *nodo*(`y`) y cuyo hijo derecho sea *nodo*(`z`). (Esta comprobación es para detectar las subexpresiones comunes.) Si no es así, créese dicho nodo. En cualquier caso, sea *n* el nodo encontrado o creado. En el caso (ii), determínese si hay un nodo etiquetado con *op*, cuyo único hijo sea *nodo*(`y`). Si no, créese dicho nodo, y que *n* sea el nodo encontrado o creado. En el caso (iii), que *n* sea el *nodo*(`y`).

3. Bórrese `x` de la lista de identificadores asociados a *nodo* (`x`). Añádase `x` a la lista de identificadores asociados al nodo *n* encontrado en el paso 2 e iguálese *nodo* (`x`) a *n*. □

Ejemplo 9.8. Considérese de nuevo el bloque de la figura 9.15 para ver cómo se construye para él el GDA de la figura 9.16. La primera proposición es t_1 := 4*i. En el paso 1 se deben crear las hojas etiquetadas con 4 e i_0. (Se utiliza el subíndice 0, como antes, para ayudar a distinguir las etiquetas de los identificadores asociados en los dibujos, pero el subíndice no forma parte realmente de la etiqueta.) En el paso 2 se crea un nodo etiquetado con *, y en el paso 3 se le asocia el identificador t_1. En la figura 9.17(a) se muestra el GDA en esta etapa.



(a)



(b)

Fig. 9.17. Pasos en el proceso de construcción de un GDA.

---

[7] Se supone que los operadores tienen a lo sumo dos argumentos. La generalización a tres o más argumentos es fácil.

Para la segunda proposición, t_2 := a[t_1] se crea una nueva hoja etiquetada con a y se encuentra el nodo anteriormente creado *nodo*(t_1). También se crea un nuevo nodo etiquetado con [ ] al cual se asocian como hijos los nodos correspondientes a a y t_1.

Para la proposición (3), t_3 := 4*i, se determina que ya existen *nodo*(4) y *nodo*(i). Como el operador es *, no se crea un nuevo nodo para la proposición (3), sino que se asocia t_3 a la lista de identificadores para el nodo t_1. En la figura 9.17(b) se muestra el GDA obtenido. Se puede utilizar el método del número de valor de la sección 5.2 para descubrir rápidamente que ya existe el nodo correspondiente a 4*i.

Se invita al lector a completar la construcción del GDA. Sólo se mencionan los pasos que se toman para la proposición (9), i := t_7. Antes de la proposición (9), *nodo*(i) es la hoja etiquetada con i_0. La proposición (9) es un ejemplo del caso (iii); por tanto, se encuentra *nodo*(t_7), se añade i a su lista de identificadores, y se *nodo*(i) a *nodo*(t_7). Esta es una de sólo dos proposiciones [la otra es la proposición (7)] donde se cambia el valor de *nodo* por un identificador. Este cambio garantiza que el nuevo nodo para i sea el hijo izquierdo del nodo para el operador <= construido para la proposición (10). □

Aplicaciones de los GDA

Se pueden obtener informaciones útiles cuando se ejecuta el algoritmo 9.2. Primero, obsérvese que las subexpresiones comunes se detectan automáticamente. Segundo, se puede determinar los identificadores cuyos valores han sido utilizados en un bloque; son exactamente aquellos para los que en algún momento se creó una hoja en el paso 1. Tercero, se puede determinar las proposiciones que calculan valores que podrían utilizarse fuera del bloque. Son precisamente aquellas proposiciones S cuyo nodo n construido o encontrado en el paso 2 aún tiene *nodo*(x) = n al final de la construcción del GDA, donde x es el identificador al que la proposición S le asigna un valor. (Equivalentemente, x todavía es un identificador asociado a n.)

Ejemplo 9.9. En el ejemplo 9.8, todas las proposiciones cumplen la limitación anterior porque siempre que *nodo* se redefine para prod e i el valor anterior de *nodo* era una hoja. Por tanto, todos los valores de los nodos interiores pueden ser utilizados fuera del bloque. Ahora, supóngase que antes de la proposición (9) se inserta una proposición nueva s que le asigna un valor a i. En la proposición s se crearía un nodo m y se haría *nodo*(i) = m. Sin embargo, en la proposición (9) se redefiniría *nodo*(i). Por tanto, el valor calculado en la proposición s no se podría utilizar fuera del bloque. □

Otro uso importante del GDA es reconstruir una lista simplificada de cuádruplos aprovechando las subexpresiones comunes y no realizando asignaciones de la forma x := y, a menos que fuera absolutamente necesario. Es decir, siempre que un nodo tenga más de un identificador en su lista asociada, se comprueba si son necesarios identificadores fuera del bloque y cuáles. Como ya se ha mencionado, encontrar las variables activas para el final de un bloque requiere un análisis del flujo de datos llamado "análisis de variables activas" que se estudia en el capítulo 10. Sin

embargo, en muchos casos se puede suponer que no se necesita ningún nombre temporal como $t_1, t_2, \ldots, t_7$ en la figura 9.15 fuera del bloque. (Pero hay que tener cuidado con la traducción de las expresiones lógicas; una expresión se puede extender a varios bloques básicos.)

En general, se pueden evaluar los nodos interiores del GDA en cualquier orden que sea un ordenamiento topológico del GDA. En un ordenamiento topológico, un nodo no se evalúa hasta que hayan sido evaluados todos sus hijos que sean nodos interiores. Conforme se evalúa un nodo, se asigna su valor a uno de sus identificadores asociados x, dando preferencia a uno cuyo valor sea necesario fuera del bloque. Sin embargo, no se puede elegir x si hay otro nodo m cuyo valor también guardara x tal que m ha sido evaluado y aún está "activo". Aquí se define que m está activo si se necesita su valor fuera del bloque o si m tiene un padre todavía sin evaluar.

Si hay identificadores asociados adicionales $y_1, y_2, \ldots, y_k$ para un nodo n cuyos valores también son necesarios fuera del bloque, se les hacen asignaciones mediante las proposiciones $y_1 := x, y_2 := x, \ldots, y_k := x$. Si n no tiene identificadores asociados (esto podría pasar si, por ejemplo, n fuera creado por una asignación a x, pero posteriormente a x se le reasignó otro valor), se crea un nuevo nombre temporal para conservar el valor de n. El lector debe saber que en la presencia de asignaciones de apuntadores o matrices, no se permite ningún ordenamiento topológico de un GDA; este tema se tratará en breve.

Ejemplo 9.10. Se reconstruye un bloque básico del GDA de la figura 9.16, ordenando los nodos en el mismo orden en que se crearon: t_1, t_2, t_4, t_5, t_6, t_7, (1). Obsérvese que las proposiciones (3) y (7) del bloque original no crearon nuevos nodos, sino que añadieron las etiquetas t_3 y prod a las listas de identificadores de los nodos t_1 y t_6, respectivamente. Se supone que ninguno de los temporales t_i es necesario fuera del bloque.

Se comienza con el nodo que representa a 4*i. Este nodo tiene dos identificadores asociados, t_1 y t_3. Se elige t_1 para guardar el valor 4*i, de modo que la primera proposición reconstruida es

```
t1 := 4 * i
```

igual que en el bloque básico original. El segundo nodo considerado se etiqueta con t_2. La proposición construida a partir de este nodo es

```
t2 := a [ t1 ]
```

también como antes. El nodo que se considera a continuación se etiqueta con t_4 a partir del cual se genera la proposición

```
t4 := b [ t1 ]
```

La última proposición utiliza t_1 como argumento en lugar de t_3 como en el bloque básico original, porque t_1 es el nombre elegido para retener el valor 4*i.

A continuación se considera el nodo etiquetado con t_5 y se genera la proposición

```
t5 := t2 * t4
```

Para el nodo etiquetado con t_6, prod, se selecciona prod para guardar el valor,

puesto que es ese identificador, y no t_6, el que se necesitará (presumiblemente) fuera del bloque. Al igual que t_3, el temporal t_6 desaparece. La siguiente proposición generada es

```
prod := prod + t5
```

De manera similar, se elige `i` en lugar de t_7 para retener el valor `i+1`. Las dos últimas proposiciones generadas son

```
i := i + 1
if i <= 20 goto (1)
```

Obsérvese que las diez proposiciones de la figura 9.15 se han reducido a siete aprovechando las subexpresiones comunes expuestas durante el proceso de construcción del GDA, y eliminando las asignaciones innecesarias. □

Matrices, apuntadores y llamadas a procedimientos

Considérese el bloque básico:

```
x := a[i]
a[j] := y          (9.5)
z := a[i]
```

Si se utiliza el algoritmo 9.2 para construir el GDA para (9.5), `a[i]` se convertiría en una subexpresión común, y el bloque "optimado" se convertiría en

```
x := a[i]
z := x             (9.6)
a[j] := y
```

Sin embargo, (9.5) y (9.6) calculan valores diferentes para `z` en el caso `i = j` e `y` $\neq$ `a[i]`. El problema es que cuando se hace una asignación a una matriz `a`, se puede estar modificando el valor de lado derecho de la expresión `a[i]`, aunque `a` e `i` no se modifiquen. Por tanto, es necesario cuando se procese una asignación a una matriz `a`, se *desactiven* todos los nodos etiquetados con [ ], cuyo argumento de la izquierda sea `a` más o menos una constante (posiblemente cero)[8]. Es decir, esos nodos se hacen inelegibles para recibir una etiqueta de identificador adicional, impidiendo que sean erróneamente reconocidos como subexpresiones comunes. Por tanto, es necesario tener un bit para cada nodo que indique si ha sido desactivado o no. Además, para cada matriz `a` mencionada en el bloque, es conveniente tener una lista de todos los nodos todavía no desactivados pero que deben desactivarse si se hace una asignación a un elemento de `a`.

Existe un problema similar si se tiene una asignación como `*p := w`, donde `p` es un apuntador. Si se sabe a lo que puede apuntar `p`, se debe desactivar todo nodo que se encuentre en ese momento en el GDA que se está construyendo. Si se desactiva el nodo *n* etiquetado con `a` y hay una asignación posterior a `a`, hay que crear una

---

[8] Obsérvese que el argumento de [ ] que indica el nombre de la matriz podría ser `a` mismo, o una expresión como `a - 4`. En este último caso, el nodo `a` sería un nieto, en lugar de un hijo, del nodo [ ].

nueva hoja para a y utilizar dicha hoja en lugar de *n*. Más adelante se consideran las limitaciones en el orden de evaluación debidas a la desactivación de nodos.

En el capítulo 10 se estudian algunos métodos mediante los cuales se podría descubrir que p sólo puede apuntar a un subconjunto de los identificadores. Si p pudiera apuntar sólo a r o s, entonces sólo deberían desactivarse *nodo*(r) y *nodo*(s). También es posible que se descubriera que `i = j` es imposible en el bloque (9.5), en cuyo caso `a[j] := y` no tendría que desactivar el nodo para `a[i]`. Sin embargo, este último tipo de descubrimiento no vale la pena.

Una llamada a un procedimiento en un bloque básico desactiva todos los nodos, puesto que si no se conoce el procedimiento llamado, se debe asumir que cualquier variable puede ser modificada como efecto secundario. El capítulo 10 estudia cómo se puede establecer que algunos identificadores no son modificados por una llamada a un procedimiento, y entonces no hay que desactivar los nodos para dichos identificadores.

Si se quiere reunir el GDA dentro de un bloque básico pero no se desea utilizar el orden en que se crearon los nodos del GDA, entonces se debe indicar en el GDA que algunos nodos aparentemente independientes deben evaluarse en un orden determinado. Por ejemplo, en (9.5), la proposición `z := a[i]` debe seguir a `a[j] := y`, que debe seguir a `x := a[i]`. Se introducirán ciertas aristas $n \rightarrow m$ en el GDA que no indican que m es un argumento de n, sino que la evaluación de n debe seguir a la evaluación de m en cualquier cálculo del GDA. Las reglas que deben cumplirse son las siguientes:

1. Cualquier evaluación o asignación a un elemento de la matriz a debe seguir a la anterior asignación a un elemento de dicha matriz si es que lo hay.
2. Cualquier asignación a un elemento de la matriz a debe seguir a cualquier evaluación previa de a.
3. Cualquier uso de un identificador debe seguir a la anterior llamada a un procedimiento o a una asignación indirecta por medio de un apuntador si es que lo hay.
4. Cualquier llamada a un procedimiento o asignación indirecta por medio de un apuntador debe seguir a todas las evaluaciones previas de cualquier identificador.

Es decir, cuando se reordena el código, los usos de una matriz a no se pueden cruzar entre sí, y ninguna proposición puede cruzar una llamada a un procedimiento o una asignación por medio de un apuntador.

9.9 OPTIMACION MEDIANTE "MIRILLA"

Una estrategia de generación de código proposición a proposición a menudo produce código objeto que contiene instrucciones redundantes y construcciones subóptimas. La calidad de dicho código objeto se puede mejorar aplicando transformaciones "optimadoras" al programa objeto. El término "optimador" es engañoso porque no está garantizado que el código resultante sea óptimo bajo ninguna medida ma-

temática. Sin embargo, muchas transformaciones simples pueden mejorar significativamente el tiempo de ejecución o las exigencias de espacio del programa objeto, de manera que es importante saber qué tipo de transformaciones son útiles en la práctica.

Una técnica sencilla pero efectiva para mejorar localmente el código objeto es la *optimación mediante "mirilla"*, un método para intentar mejorar el rendimiento del proyecto objeto examinando una secuencia corta de instrucciones objeto (llamada *mirilla*) y sustituyendo estas instrucciones por una secuencia más corta o más rápida, si es posible. Aunque se estudia la optimación mediante mirilla como una técnica para mejorar la calidad del código objeto, la técnica también se puede aplicar directamente después de la generación de código intermedio para mejorar la representación intermedia.

La mirilla es una ventana pequeña que se mueve en el programa objeto. No hace falta que el código dentro de la mirilla sea contiguo, aunque algunas aplicaciones exigen que lo sea. Una característica de la optimación mediante mirilla es que cada mejora puede brindar oportunidades para mejoras adicionales. En general, son necesarias repetidas pasadas sobre el código objeto para obtener las mayores ventajas. En esta sección se darán los siguientes ejemplos de transformaciones de programas característicos de las optimaciones mediante mirilla:

- eliminación de instrucciones redundantes
- optimaciones del flujo del control
- simplificaciones algebraicas
- uso de instrucciones especiales de la máquina.

Cargas y almacenamientos redundantes

Si se encuentra la siguiente secuencia de instrucciones

```
(1)  MOV    R0, a
(2)  MOV    a, R0
```
(9.7)

se puede borrar la instrucción (2) porque siempre que se ejecute (2), (1) garantizará que el valor de `a` ya está en el registro `R0`. Obsérvese que si (2) tuviera una etiqueta[9], no se podría estar seguro de que (1) siempre se ejecutó justo antes de (2), y entonces no se podría eliminar (2). Dicho de otro modo, (1) y (2) tienen que estar en el mismo bloque básico para que esta transformación sea válida.

Aunque no se generaría el código objeto como (9.7) si se utilizara el algoritmo propuesto en la sección 9.6, podría serlo si se utilizara un algoritmo más ingenuo como el que se menciona al comienzo de la sección 9.1.

Código inalcanzable

Otra oportunidad para la optimación local es la eliminación de instrucciones inalcanzables. Se puede eliminar una instrucción sin etiqueta que siga inmediatamente

---

[9] Una ventaja de generar código ensamblador es que estarán presentes las etiquetas, facilitando las optimaciones locales como ésta. Si se genera código de máquina y se desea optimación local, se puede utilizar un bit para marcar las instruciones con etiquetas.

a un salto incondicional. Esta operación se puede repetir para eliminar una secuencia de instrucciones. Por ejemplo, para una mayor depuración, un programa grande puede tener dentro de él algunos segmentos que se ejecutan sólo si una variable llamada `depura` vale 1. En C, el código fuente se parecería a:

```
#define depura 0
...
if ( depura ) {
        imprime información para la depuración
}
```

En la representación intermedia la proposición `if` se puede traducir como:

```
        if depura = 1 goto L1
        goto L2
L1:     imprime información para la depuración          (9.8)
L2:
```

Una optimación mediante mirilla obvia consiste en eliminar los saltos sobre saltos. Por tanto, independientemente del valor de `depura`, (9.8) se puede sustituir por:

```
        if depura ≠ 1 goto L2
        imprime información para la depuración          (9.9)
L2:
```

Ahora, como a `depura` se asigna 0 al principio del programa[10], la propagación de constantes debe sustituir (9.9) por

```
        if 0 ≠ 1 goto L2
        imprime información para la depuración          (9.10)
L2:
```

Como el argumento de la primera proposición de (9.10) se evalúa como la constante **true**, se puede sustituir por `goto L2`. Entonces todas las proposiciones que impriman ayudas para la depuración son manifiestamente inalcanzables y se pueden eliminar de una en una.

Optimaciones del flujo del control

A menudo, los algoritmos del capítulo 8 para generación de código intermedio producen saltos hacia saltos, saltos hacia saltos condicionales, o saltos condicionales hacia saltos. Estos saltos innecesarios se pueden eliminar, ya sea del código intermedio o del código objeto, mediante los siguientes tipos de optimaciones locales. Se puede sustituir la secuencia de saltos

```
        goto L1
        ...
L1:     goto L2
```

[10] Para afirmar que `depura` tiene el valor 0, hay que hacer un análisis del flujo de datos global de "definiciones de alcance", como se presenta en el capítulo 10.

por la secuencia

```
        goto L2
        ...
    L1: goto L2
```

Si ahora no hay saltos a `L1`[11], entonces se puede eliminar la proposición `L1: goto L2` siempre que vaya precedida de un salto incondicional. De manera similar, la secuencia

```
        if a < b goto L1
        ...
    L1: goto L2
```

se puede sustituir por

```
        if a < b goto L2
        ...
    L1: goto L2
```

Por último, supóngase que sólo hay un salto a `L1` y que `L1` va precedida de un salto **goto** incondicional. Entonces la secuencia.

```
        goto L1
        ...
    L1: if a < b goto L2
    L3:
```
(9.11)

se puede sustituir por

```
        if a < b goto L2
        goto L3
        ...
    L3:
```
(9.12)

Aunque el número de instrucciones en (9.11) y (9.12) es el mismo, a veces se puede evitar el salto incondicional en (9.12), pero nunca en (9.11). Por tanto, (9.12) es superior a (9.11) en tiempo de ejecución.

Simplificación algebraica

No hay límite al número de simplificaciones algebraicas que se pueden intentar con la optimación mediante mirilla. Sin embargo, sólo unas pocas identidades algebraicas ocurren con la frecuencia suficiente como para que valga la pena considerar su implantación. Por ejemplo, las proposiciones como

```
    x := x + 0
```

o

```
    x := x * 1
```

---

[11] Si se intenta esta optimación mediante mirilla, se puede contar el número de saltos hacia cada etiqueta en la entrada de la tabla de símbolos para esta etiqueta; no es necesario examinar el código.

a menudo son producidas por algoritmos directos para la generación de código intermedio, y se pueden eliminar fácilmente con la optimación mediante mirilla.

Reducción de intensidad

La reducción de intensidad sustituye operaciones caras por otras equivalentes más baratas en la máquina objeto. Algunas instrucciones de máquina son considerablemente más baratas que otras y se pueden utilizar como casos especiales de operadores más caros. Por ejemplo, x^2 es invariablemente más barata de implantar como `x*x` que como una llamada a una rutina de exponenciación. La división o multiplicación de punto fijo por una potencia de dos es más barata de implantar como un desplazamiento. La división de punto flotante por una constante se puede implantar (de manera aproximada) como multiplicación por una constante, que puede ser más barata.

Uso de instrucciones especiales de la máquina

La máquina objeto puede tener instrucciones de *hardware* para implantar ciertas operaciones específicas eficientemente. Detectar las situaciones que permitan el uso de estas instrucciones puede reducir significativamente el tiempo de ejecución. Por ejemplo, algunas máquinas tienen modos de direccionamiento de autoincremento y autodecremento. Estas suman o restan uno a un operando antes o después de utilizar su valor. El uso de estos modos mejora mucho la calidad del código cuando se introduce o se saca una pila, como en el paso de parámetros. Estos modos también se pueden utilizar en el código para proposiciones como `i:=i+1`.

9.10 GENERACION DE CODIGO A PARTIR DE LOS GDA

En esta sección se muestra cómo generar código para un bloque básico a partir de su representación en GDA. La ventaja de hacer esto es que a partir de un GDA se puede ver más fácilmente cómo reorganizar el orden de la secuencia de cálculos final que empezando a partir de una secuencia lineal de proposiciones de tres direcciones o cuádruplos. El caso más importante para este análisis es cuando el GDA es un árbol. En este caso se puede generar código demostrado como óptimo desde el punto de vista de la longitud del programa o el menor número de temporales utilizados. Este algoritmo para la generación de código óptimo a partir de un árbol también es útil cuando el código intermedio es un árbol de análisis sintáctico.

Reorganización del orden

Se considera brevemente cómo el orden en que se hacen los cálculos puede influir en el costo del código objeto resultante. Considérese el siguiente bloque básico cuya representación en GDA se muestra en la figura 9.18 (el GDA es un árbol).

```
t1 := a + b
t2 := c + d
t3 := e - t2
t4 := t1 - t3
```

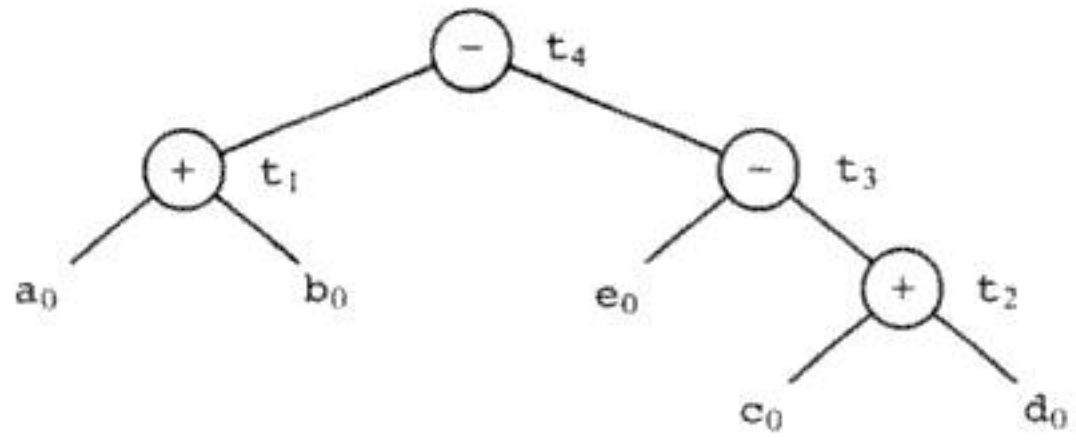


Fig. 9.18. GDA para un bloque básico.

Obsérvese que el orden es el que se obtendría de forma natural de una traducción dirigida por la sintaxis de la expresión `(a+b)-(e-(c+d))` mediante el algoritmo de la sección 8.3.

Si se genera código para las proposiciones de tres direcciones utilizando el algoritmo de la sección 9.6, se obtiene la secuencia de código de la figura 9.19 (suponiendo que dos registros, `R0` y `R1`, están disponibles y que sólo t_4 está activo a la salida).

```
MOV a, R0
ADD b, R0
MOV c, R1
ADD d, R1
MOV R0, t1
MOV e, R0
SUB R1, R0
MOV t1, R1
SUB R0, R1
MOV R1, t4
```

Fig. 9.19. Secuencia de código.

Por otra parte, supóngase que se reorganizó el orden de las proposiciones de modo que el cálculo de t_1 tiene lugar inmediatamente antes que el de t_4 como:

```
t2 := c + d
t3 := e - t2
t1 := a + b
t4 := t1 - t3
```

Entonces, utilizando el algoritmo de generación de código de la sección 9.6, se obtiene la secuencia de código de la figura 9.20. (De nuevo, sólo `R0` y `R1` están disponibles.) Realizando los cálculos por este orden, se han podido ahorrar dos instrucciones, `MOV R0,` t_1 (que almacena el valor de `R0` en la localidad de memoria t_1) y `MOV` t_1`, R1` (que recarga el valor de t_1 en el registro `R1`).

```
MOV c, R0
ADD d, R0
MOV e, R1
SUB R0, R1
MOV a, R0
ADD b, R0
SUB R1, R0
MOV R0, t4
```

Fig. 9.20. Secuencia de código revisada.

Un ordenamiento heurístico de los GDA

El reordenamiento anterior mejoró el código porque se dispuso que el cálculo de t_4 fuera inmediatamente después del cálculo de t_1, su operando izquierdo en el árbol. Resulta claro que esta medida es beneficiosa. El argumento izquierdo para el cálculo de t_4 debe estar en un registro para calcular eficientemente t_4, y calcular t_1 inmediatamente antes que t_4 garantiza este hecho.

Al seleccionar un ordenamiento para los nodos de un GDA hay que asegurarse de que el orden preserve las relaciones de aristas del GDA. Recuérdese (Sec. 9.8) que esas aristas pueden representar la relación operador-operando o las limitaciones debidas a posibles interacciones entre las llamadas a procedimientos, asignaciones con matrices, o asignaciones con apuntadores. Se propone el siguiente algoritmo de ordenamiento heurístico, que intenta en lo posible hacer que la evaluación de un nodo vaya inmediatamente después de la evaluación de su argumento de la izquierda. El algoritmo de la figura 9.21 proporciona el ordenamiento inverso.

Ejemplo 9.11. El algoritmo de la figura aplicado al árbol de la figura 9.18 proporciona el orden a partir del cual se produjo el código de la figura 9.20. Para un examen más completo, considérese el GDA de la figura 9.22.

```
(1)  while queden nodos interiores sin listar do begin
(2)      seleccionar un nodo no listado n, para el que todos sus
             padres hayan sido listados;
(3)      listar n;
(4)      while el hijo de más a la izquierda m de n no tenga padres
             sin listar y no sea una hoja do
             /* ya que n se acaba de listar, m aún no se ha listado */
             begin
(5)              lista (m);
(6)              n := m
             end
     end
```

Fig. 9.21. Algoritmo para listar nodos.

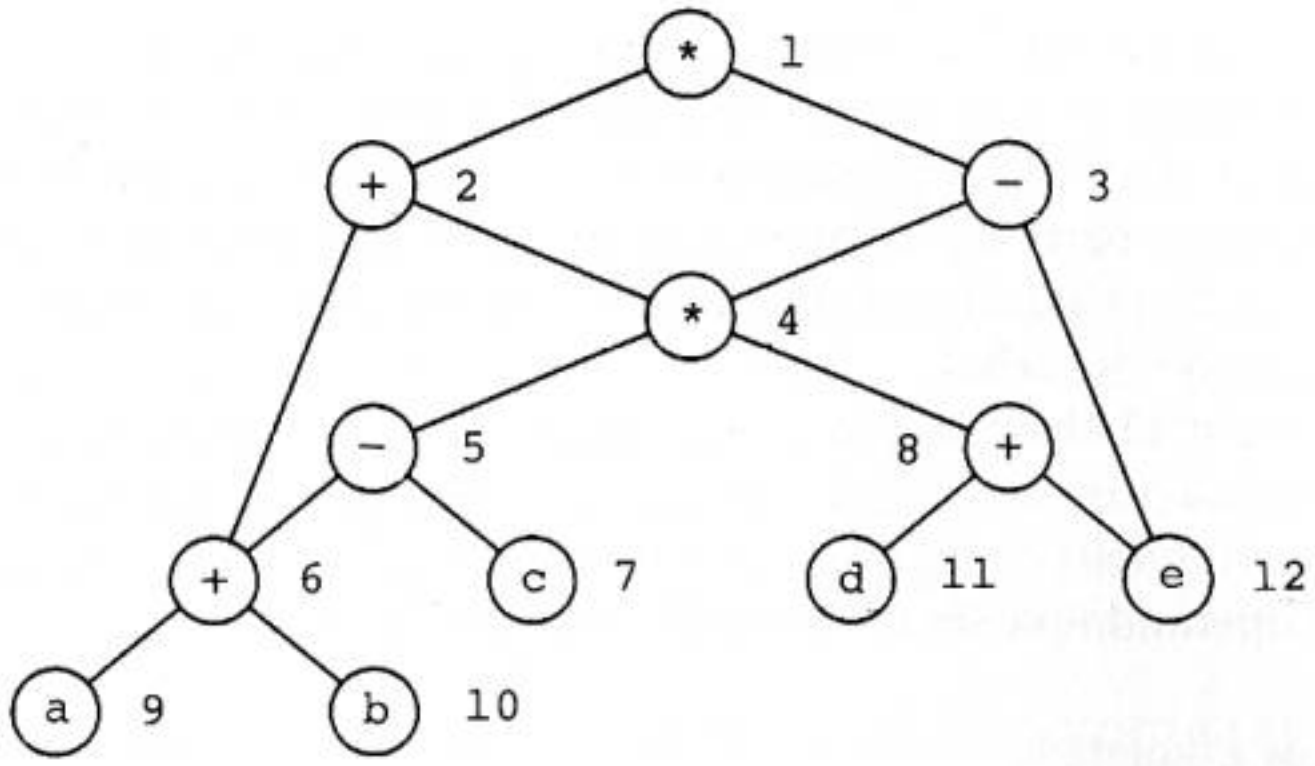


Fig. 9.22. Un GDA.

Al inicio, el único nodo que no tiene padres sin listar es 1, así que se hace $n = 1$ en la línea (2) y se lista 1 en la línea (3). Ahora el argumento izquierdo de 1, que es 2, ya tiene en lista a sus padres, así que se lista 2 y se hace $n = 2$ en la línea (6). Ahora, en la línea (4) se ve que el hijo de la izquierda de 2, que es 6, tiene un padre sin listar, 5. Por tanto se selecciona una nueva n en la línea (2), y el nodo 3 es el único candidato. Se lista 3 y después se avanza a lo largo de su cadena izquierda, listando 4, 5 y 6. Ya sólo queda 8 entre los nodos interiores, así que se lista 8. La lista obtenida es 1234568 así que el orden de evaluación propuesto es 8654321. Este ordenamiento corresponde a la secuencia de proposiciones de tres direcciones:

```
t8 := d + e
t6 := a + b
t5 := t6 - c
t4 := t5 * t8
t3 := t4 - e
t2 := t6 + t4
t1 := t2 * t3
```

que producirá código óptimo para el GDA en la máquina descrita en este capítulo cualquiera que sea el número de registros, si se utiliza el algoritmo de generación de código de la sección 9.6. Debe observarse que en este ejemplo, la heurística de ordenamiento no tuvo que elegir ninguna opción en el paso (2), pero en general puede tener muchas alternativas. □

Ordenamiento óptimo para árboles

Resulta que para el modelo de máquina de la sección 9.2 se puede dar un algoritmo sencillo para determinar el orden óptimo en el que evaluar las proposiciones de un bloque básico cuando la representación en GDA del bloque es un árbol. En este caso, orden óptimo significa el orden que produce la secuencia de instrucciones más corta, de todas las secuencias de instrucciones que evalúan el árbol. Este algoritmo, modificado para que tenga en cuenta pares de registros y otras particularidades de la máquina objeto, ha sido utilizado en compiladores para ALGOL, BLISS y C.

El algoritmo consta de dos partes. La primera parte etiqueta cada nodo del árbol, en forma ascendente, con un entero que indica el número mínimo de registros exigido para evaluar el árbol sin almacenamientos en memoria de los resultados intermedios. La segunda parte del algoritmo es un recorrido de árbol cuyo orden viene determinado por las etiquetas calculadas de los nodos. El código resultante se genera durante el recorrido del árbol.

Intuitivamente, el algoritmo funciona, dados los dos operandos de un operador binario, evaluando primero el operando que requiera un número mayor de registros (el operando más difícil). Si las exigencias de registros de ambos operandos son iguales, cualquier operando puede ser evaluado primero.

El algoritmo de etiquetado

Se utiliza el término "hoja izquierda" para un nodo que es una hoja y el descendiente de más a la izquierda de su padre. Todas las otras hojas se denominan como "hojas derechas".

Se puede realizar el etiquetado visitando los nodos en orden ascendente de modo que no se visite un nodo hasta que se hayan etiquetado todos sus hijos. El orden en que se crean los nodos del árbol sintáctico es el adecuado si el árbol de análisis sintáctico se utiliza como código intermedio, así que en este caso las etiquetas se pueden calcular como una traducción dirigida por la sintaxis. La figura 9.23 contiene el

```
(1)  if n es una hoja then
(2)      if n es el hijo de más a la izquierda de su padre then
(3)          etiqueta (n) := 1;
(4)      else etiqueta (n) := 0
     else begin /* n es un nodo interior */
(5)      sean n1, n2, . . . , nk los hijos de n ordenados por etiqueta,
             de modo que etiqueta (n1) ≥ etiqueta (n2) ≥ . . . ≥ etiqueta (nk);
(6)      etiqueta (n) := máx (etiqueta (ni)+i−1)
                         1≤i≤k
     end
```

Fig. 9.23. Cálculo de etiquetas.

algoritmo para calcular la etiqueta en el nodo n. En el importante caso especial en que n es un nodo binario y sus hijos tienen etiquetas l_1 y l_2, la fórmula de la línea (6) se reduce a

$$etiqueta(n) = \begin{cases} \text{máx}(l_1, l_2) & \text{si } l_1 \neq l_2 \\ l_1 + 1 & \text{si } l_1 = l_2 \end{cases}$$

---

[12] Un recorrido en orden posterior visita recursivamente los subárboles con raíces en los hijos n_1, n_2, . . . , n_k de un nodo n, y después visita n. Este es el orden en que se crean los nodos de un árbol de análisis sintáctico en un análisis sintáctico ascendente.

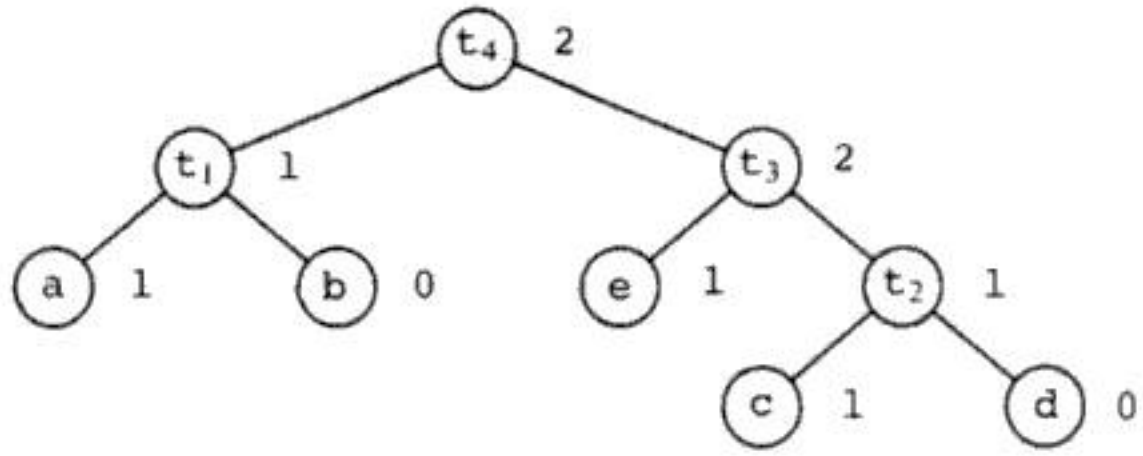


Fig. 9.24. Arbol etiquetado.

Ejemplo 9.12. Considérese el árbol de la figura 9.18. Un recorrido en orden posterior[12] de los nodos visita los nodos en el orden a b t_1 e c d t_2 t_3 t_4. El orden posterior siempre es un orden apropiado para hacer los cálculos de las etiquetas. El nodo a se etiqueta con 1 porque es una hoja izquierda. El nodo b se etiqueta con 0 porque es una hoja derecha. El nodo t_1 se etiqueta con 1 porque las etiquetas de sus hijos no son iguales y la etiqueta máxima de un hijo es 1. La figura 9.24 muestra el árbol etiquetado así obtenido. Implica que se necesitan dos registros para evaluar t_4 y de hecho, se necesitan dos registros sólo para evaluar t_3. □

Generación de código a partir de un árbol etiquetado

A continuación se estudia el algoritmo que toma como entrada un árbol etiquetado T y produce como salida una secuencia de código de máquina que evalúa T en R0. (Entonces R0 se puede almacenar en la posición de memoria adecuada.) Se supone que T tiene sólo operadores binarios. No es difícil generalizar a operadores con un número arbitrario de operandos, y se deja como ejercicio práctico.

El algoritmo utiliza el procedimiento recursivo *gencódigo*(n) para producir código de máquina que evalúe el subárbol de T con raíz n en un registro. El procedimiento *gencódigo* utiliza una pila, *pilar*, para asignar los registros. Al inicio, *pilar* contiene todos los registros disponibles, es decir, R0, R1, . . . , R(r−1), por este orden. Una llamada de *gencódigo* puede encontrar un subconjunto de los registros, quizás en un orden distinto, en *pilar*. Cuando retorna de *gencódigo*, deja los registros en *pilar* en el mismo orden en que los encontró. El código obtenido calcula el valor del árbol T en registro del tope en *pilar*.

La función *permuta*(*pilar*) intercambia los dos registros del tope en *pilar*. El uso de *permuta* garantiza que un hijo izquierdo y su padre se evalúen en el mismo registro.

El procedimiento *gencódigo* utiliza una pila, *pilat*, para asignar posiciones de memoria temporales. Se supone que *pilat* al inicio contiene T0, T1, T2, ... En la práctica, no hace falta implantar *pilat* como una lista, si se lleva un registro de i tal que Ti se encuentre normalmente en el tope. El contenido de *pilat* es siempre un sufijo de T0, T1, ...

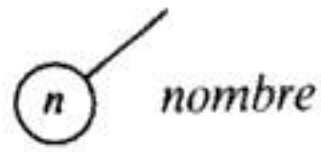
La proposición X := *saca*(*pila*) significa "sacar el elemento del tope de *pila* y asignar su valor a X". A la inversa, se utiliza *mete* (*pila*, X) con el significado "meter X en el tope de *pila*; *tope*(*pila*) se refiere al valor del tope de *pila*.

El algoritmo de generación de código consiste en llamar a *gencódigo* en la raíz

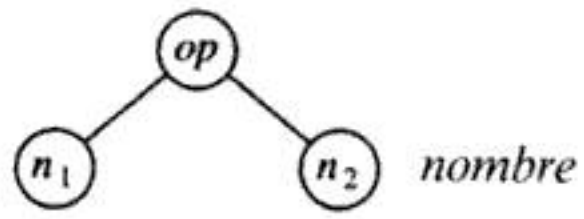
```
procedure gencódigo (n);
begin
/* caso 0 */
if n es una hoja izquierda que representa el operando nombre and
        n es el hijo más a la izquierda de su padre then
    print 'MOV' ‖ nombre ‖ ',' ‖ tope (pilar)
else if n es un nodo interior con operador op, hijo izquierdo n1,
        e hijo derecho n2 then
/* caso 1 */
    if etiqueta (n2) = 0 then begin
        sea nombre el operando representado por n2;
        gencódigo (n1);
        print op ‖ nombre ‖ ',' ‖ tope (pilar)
    end
/* caso 2 */
    else if 1 ≤ etiqueta (n1) < etiqueta (n2) and etiqueta (n1) < r then begin
        permuta (pilar);
        gencódigo (n2);
        R := saca (pilar); /* n2 se evaluó en el registro R */
        gencódigo (n1);
        print op ‖ R ‖ ',' ‖ tope (pilar);
        mete (pilar, R);
        permuta (pilar)
    end
/* caso 3 */
    else if 1 ≤ etiqueta (n2) ≤ etiqueta (n1) and etiqueta (n2) < r then begin
        gencódigo (n1);
        R := saca (pilar); /* n1 se evaluó en el registro R */
        gencódigo (n2);
        print op ‖ tope (pilar) ‖ ',' ‖ R;
        mete (pilar, R)
    end
/* caso 4, ambas etiquetas ≥ r, el número total de registros */
    else begin
        gencódigo (n2);
        T := saca (pilat);
        print 'MOV' ‖ tope (pilar) ‖ ',' ‖ T;
        gencódigo (n1);
        mete (pilat, T);
        print op ‖ T ‖ ',' ‖ tope (pilar)
    end
end
```

Fig. 9.25. La función *gencódigo*.

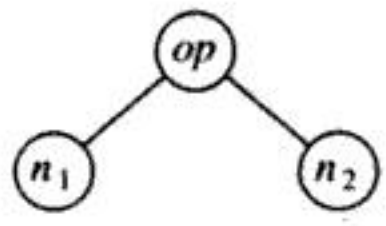
de T, donde *gencódigo* es el procedimiento de la figura 9.25. Se puede explicar examinando cada uno de los cinco casos. Para el caso 0, se tiene un subárbol de la forma



Es decir, n es una hoja y el hijo más a la izquierda de su padre. Por tanto, sólo se genera una instrucción de carga. En el caso 1, se tiene un subárbol de la forma



para el que se genera código para evaluar n_1 en el registro R=*tope*(*pilar*) seguido de la instrucción *op nombre*, R. En el caso 2, se tiene un subárbol de la forma



donde n_1 se puede evaluar sin almacenamientos en memoria pero n_2 es más difícil de evaluar (es decir, requiere más registros) que n_1. Para este caso, se permutan los dos registros del tope en *pilar*, después se evalúa n_2 en R = *tope*(*pilar*). Se elimina R de *pilar* y se evalúa n_1 en S = *tope*(*pilar*). Obsérvese que S era el registro que estaba inicialmente en el tope de *pilar* al comienzo del caso 2. Después se genera la instrucción *op* R, S, que produce el valor de n (el nodo etiquetado con *op*) en el registro S. Otra llamada a *permuta* deja *pilar* como estaba cuando comenzó esta llamada de *gencódigo*.

El caso 3 es similar al caso 2 excepto en que aquí, el subárbol izquierdo es más difícil y se evalúa primero. Aquí no es necesario intercambiar registro con *permuta*.

El caso 4 ocurre cuando ambos subárboles necesitan r o más registros para evaluar sin almacenamientos en memoria. Como hay que utilizar una posición de memoria temporal, primero se evalúa el subárbol derecho en la posición temporal T, después el subárbol izquierdo y por último la raíz.

Ejemplo 9.13. Se generará código para el árbol etiquetado de la figura 9.24 con *pilar* = `R0, R1` al inicio. En la figura 9.26 se muestran la secuencia de llamadas a *gencódigo* y los pasos de impresión de código. A un lado se muestra entre corchetes el contenido de *pilar* en el momento de cada llamada, con el tope en el extremo derecho. Aquí la secuencia de código es una permutación de la de la figura 9.20. □

Se puede demostrar que *gencódigo* produce código óptimo en expresiones para el modelo de máquina de este capítulo, suponiendo que no se tienen en cuenta las propiedades algebraicas de los operadores y que no hay subexpresiones comunes. La prueba, que se deja como ejercicio práctico, se basa en demostrar que cualquier secuencia de código debe realizar

1. una operación para cada nodo interior,

```
gencódigo(t4)        [R1R0]                    /* caso 2 */
    gencódigo(t3)        [R0R1]                /* caso 3 */
        gencódigo(e)         [R0R1]            /* caso 0 */
            print MOV e,R1
        gencódigo(t2)        [R0]              /* caso 1 */
            gencódigo(c)         [R0]          /* caso 0 */
                print MOV c,R0
            print ADD d,R0
        print SUB R0,R1
    gencódigo(t1)        [R0]                  /* caso 1 */
        gencódigo(a)         [R0]              /* caso 0 */
            print MOV a,R0
        print ADD b,R0
    print SUB R1,R0
```

Fig. 9.26. Rastreo de la rutina *gencódigo*.

2. una carga para cada hoja que es el hijo más a la izquierda de su padre, y
3. un almacenamiento para cada nodo cuyos dos hijos tienen etiquetas iguales o mayores que *r*.

Como *gencódigo* produce exactamente estos pasos, es óptima.

Operaciones en varios registros

Se puede modificar el algoritmo de etiquetado para manejar operaciones como multiplicación, división o llamada a una función, que exigen generalmente más de un registro para llevarse a cabo. Basta con modificar el paso (6) de la figura 9.23, el algoritmo de etiquetado, así que *etiqueta*(*n*) es siempre por lo menos el número de registros necesarios para la operación. Por ejemplo, si se supone que la llamada a una función requiere todos los *r* registros, se sustituye la línea (6) por *etiqueta*(*n*) = *r*. Si la multiplicación necesita dos registros, en el caso binario se utiliza

$$etiqueta(n) = \begin{cases} \text{máx}(2, l_1, l_2) & \text{si } l_1 \neq l_2 \\ l_1 + 1 & \text{si } l_1 = l_2 \end{cases}$$

donde l_1 y l_2 son las etiquetas de los hijos de *n*.

Desgraciadamente, esta modificación no garantizará que un par de registros esté disponible para una multiplicación o división o para operaciones de múltiple precisión. Un truco útil en algunas máquinas es simular que la multiplicación y la división requieren tres registros. Si *permuta* nunca se utiliza en *gencódigo*, entonces *pilar* contendrá siempre registros con números altos consecutivos, i, $i+1$, . . . , $r-1$ para una i. Por tanto, los tres primeros incluyen sin ninguna duda un par de registros. Aprovechando que muchas operaciones son conmutativas, a menudo se puede evitar utilizar el caso 2 de *gencódigo*, el caso que llama a *permuta*. Asimismo, aunque *pilar* no contenga tres registros consecutivos en el tope, es muy probable que se encuentren un par de registros en *pilar*.

Propiedades algebraicas

Si se asumen leyes algebraicas para varios operadores, existe la posibilidad de sustituir un árbol T dado por uno con etiquetas más pequeñas (para evitar almacenamientos en el caso 4 de *gencódigo*) o menos hojas izquierdas (para evitar cargas en el caso 1) o ambas cosas. Por ejemplo, como + se considera conmutativo generalmente, se puede sustituir el árbol de la figura 9.27(a) por el de la figura 9.27(b), reduciendo el número de hojas izquierdas en uno y posiblemente reduciendo asimismo los valores de algunas etiquetas.

Como + también se considera asociativo además de conmutativo, se puede tomar un grupo de nodos etiquetado con + como en la figura 9.27(c) y sustituirlo por una cadena izquierda como en la figura 9.27(d). Para minimizar el valor de la etiqueta de la raíz, sólo hay que disponer que T_{i_1} sea uno de T_1, T_2, T_3 y T_4 con la mayor etiqueta y que T_{i_1} no sea una hoja a menos que lo sean todas las $T_1, \ldots, T_4$.

Subexpresiones comunes

Cuando hay subexpresiones comunes en un bloque básico, el GDA correspondiente ya no será un árbol. Las subexpresiones comunes corresponderán a nodos con más de un padre, llamados *nodos compartidos*. Ya no se puede aplicar directamente el algoritmo de etiquetado o *gencódigo*. De hecho, las subexpresiones comunes dificultan mucho la generación de código desde el punto de vista matemático. Bruno y Sethi [1976] demostraron que la generación de código óptimo para GDA en una máquina de un registro es NP completo. Aho, Johnson y Ullman [1977a] demostraron que incluso con un número ilimitado de registros, el problema continúa siendo NP completo. La dificultad surge al intentar determinar un orden óptimo en que evaluar un GDA de la manera más barata.

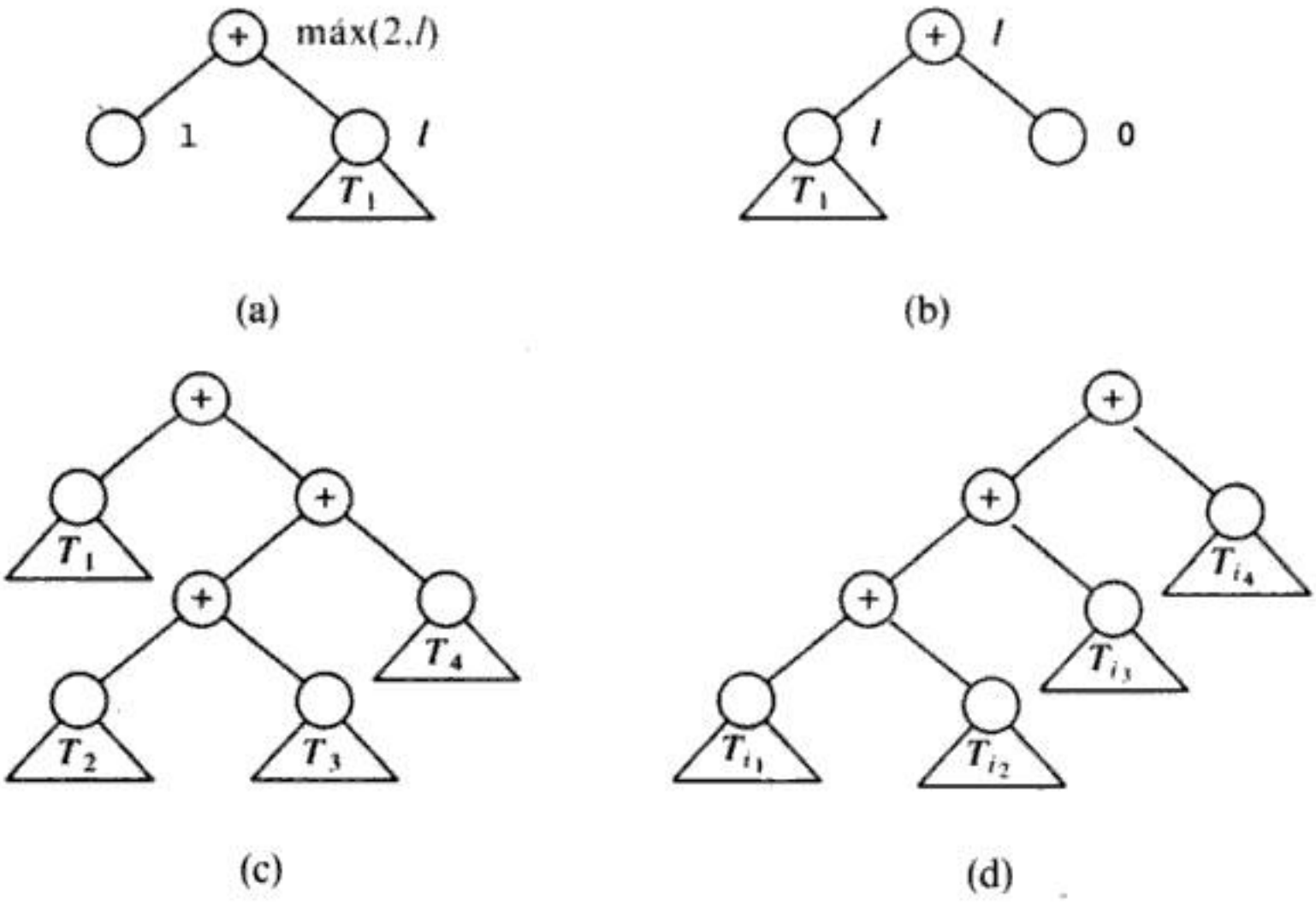


Fig. 9.27. Transformaciones conmutativa y asociativa.

En la práctica se puede obtener una solución razonable si se particiona el GDA en un conjunto de árboles, encontrando para cada raíz o nodo compartido n el subárbol máximo con n como raíz que no incluya otros nodos compartidos, excepto como hojas. Por ejemplo, se puede particionar el GDA de la figura 9.22 en los árboles de la figura 9.28. Cada nodo compartido con p padres aparece como una hoja en a lo sumo p árboles. Los nodos con más de un padre en el mismo árbol se pueden convertir en tantas hojas como sea necesario, de modo que ninguna hoja tenga múltiples padres.

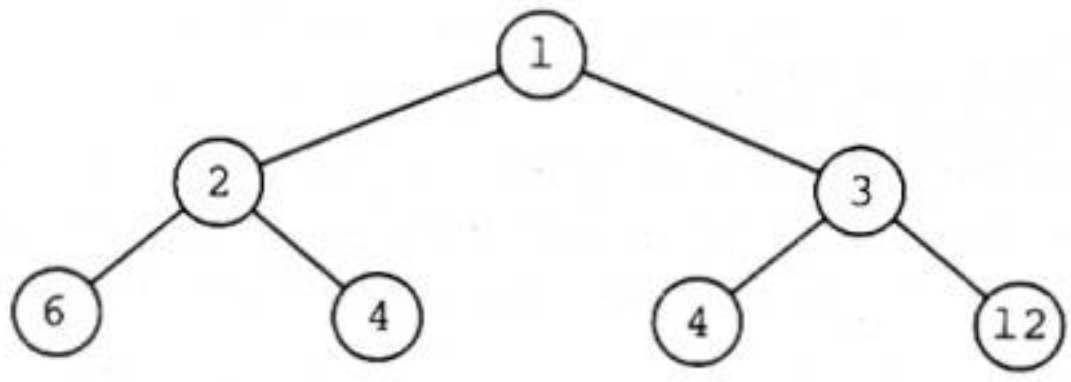


Fig. 9.28. Partición en árboles.

Una vez así particionado el GDA en árboles, se puede ordenar la evaluación de los árboles y utilizar cualquiera de los algoritmos precedentes para generar código para cada árbol. El orden de los árboles debe ser tal que los valores compartidos que sean hojas de un árbol deben estar disponibles cuando se evalúe el árbol. Las cantidades compartidas se pueden calcular y almacenar en memoria (o conservar en registros si hay suficientes registros disponibles). Si bien este proceso no necesariamente genera código óptimo, con frecuencia será satisfactorio.

9.11 ALGORITMO PARA GENERACION DE CODIGO CON PROGRAMACION DINAMICA

En la sección anterior, el procedimiento *gencódigo* produce código óptimo a partir de un árbol de expresiones utilizando una cantidad de tiempo que es una función lineal del tamaño del árbol. Este procedimiento es válido para máquinas en las que

todos los cálculos se hacen en registros y en las que las instrucciones constan de un operador aplicado a dos registros o a un registro y a una posición de memoria.

Se puede utilizar un algoritmo basado en el principio de la programación dinámica para ampliar la clase de máquinas para las que se puede generar código óptimo a partir de árboles de expresiones en un tiempo lineal. El algoritmo de programación dinámica se aplica a una amplia clase de máquinas de registros con conjuntos de instrucciones complejas.

Una clase de máquinas de registros

Se puede utilizar el algoritmo de programación dinámica para generar código para cualquier máquina con r registros intercambiables `R0`, `R1`, . . . , `R`$r-1$ e instrucciones de la forma `R`i := E donde E es cualquier expresión que contenga operadores, registros y posiciones de memoria. Si E implica a uno o más registros, entonces `R`i debe ser uno de dichos registros. Este modelo de máquina incluye la máquina presentada en la sección 9.2.

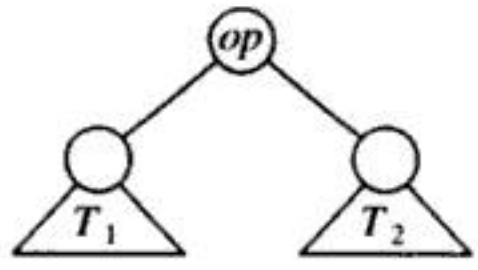
Por ejemplo, la instrucción `ADD R0,R1` se representaría como `R1 := R1 + R0`. La instrucción `ADD *R0,R1` se representaría como `R1 := R1 + ind R0`, donde `ind` representa el operador de indirección.

Se supone que una máquina tiene una instrucción de carga `R`i := `M`, una instrucción de almacenamiento `M := R`i, y una instrucción de copia de registro a registro `R`i := `R`j. Para simplificar, se supone también que toda instrucción cuesta una unidad, aunque el algoritmo de programación dinámica se puede modificar fácilmente para que funcione incluso si cada instrucción tiene su propio costo.

El principio de la programación dinámica

El algoritmo de programación dinámica particiona el problema de generar código óptimo para una expresión en subproblemas generación de código óptimo para las subexpresiones de la expresión dada. Como ejemplo, considérese una expresión E de la forma $E_1 + E_2$. Un programa óptimo para E se forma combinando programas óptimos para E_1 y E_2, en uno u otro orden, seguido de código para evaluar el operador +. Los subproblemas de generar código óptimo para E_1 y E_2 se resuelven de manera similar.

Un programa óptimo producido por el algoritmo de programación dinámica tiene una propiedad importante. Evalúa una expresión $E = E_1\ op\ E_2$ "contiguamente". Observando el árbol sintáctico T para E, se averigua lo que esto significa.



Aquí, T_1 y T_2 son árboles para E_1 y E_2, respectivamente.

Evaluación contigua

Se dice que un programa P evalúa un árbol T *contiguamente* si evalúa primero aquellos subárboles de T que necesiten calcularse en la memoria. Después evalúa el resto de T, ya sea en el orden T_1, T_2 y después la raíz, o en el orden T_2, T_1 y después la raíz, utilizando en ambos casos los valores anteriormente calculados de la memoria siempre que sea necesario. Como ejemplo de evaluación no contigua, P puede evaluar primero parte de T_1 dejando el valor en un registro (en lugar de en la memoria), a continuación evaluar T_2 y después regresar para evaluar el resto de T_1.

En cuanto a la máquina de registros definida anteriormente, se puede demostrar que dado un programa P en lenguaje de máquina para evaluar un árbol de expresiones T, se puede encontrar un programa equivalente P' tal que

1. P' no tiene un costo más alto que P,
2. P' no utiliza más registros que P y
3. P' evalúa el árbol en forma contigua.

Este resultado implica que todo árbol de expresiones se puede evaluar óptimamente mediante un programa contiguo.

A modo de contraste, las máquinas con parejas de registros par-impar como las máquinas IBM Sistema/370 no siempre tienen evaluaciones contiguas óptimas. Para estas máquinas se pueden dar ejemplos de árboles de expresiones en los que un programa óptimo en lenguaje de máquina debe evaluar primero en un registro una parte del subárbol izquierdo de la raíz, luego, una parte del subárbol derecho, a continuación, otra parte del subárbol izquierdo, y después, otra parte del derecho, y así sucesivamente. Este tipo de oscilación es innecesaria para una evaluación óptima de un árbol de expresiones que utilice la máquina de registros generales.

La propiedad de evaluación contigua definida anteriormente indica que para cualquier árbol de expresiones T, siempre existe un programa óptimo que consta de programas óptimos para subárboles de la raíz, seguidos de una instrucción para evaluar la raíz. Esta propiedad permite utilizar el algoritmo de programación dinámica para generar un programa óptimo para T.

El algoritmo de programación dinámica

El algoritmo de programación dinámica actúa en tres fases. Supóngase que la máquina objeto tiene r registros. En la primera fase, para cada nodo n del árbol de expresiones T se calcula en forma ascendente una matriz C de costos, en la que el i-ésimo componente $C[i]$ es el costo óptimo de calcular el subárbol S con raíz en n en un registro, suponiendo que están disponibles i registros para el cálculo, $1 \leq i \leq r$. El costo incluye todas las cargas y almacenamientos necesarios para evaluar S en el número de registros dado. También incluye el costo de calcular el operador en la raíz de S. El componente número cero del vector de costos es el costo óptimo de calcular el subárbol S en la memoria. La propiedad de evaluación contigua asegura la generación de un programa óptimo para S si se consideran combinaciones de programas óptimos sólo para los subárboles de la raíz de S. Esta limitación reduce el número de casos que hay que considerar.

Para calcular $C[i]$ en el nodo n, considérese cada instrucción de máquina R := E

cuya expresión E concuerda con la subexpresión con raíz en el nodo n. Examinando los vectores de costos en los descendientes correspondientes de n, se determinan los costos de evaluar los operandos de E. Para aquellos operandos de E que sean registros, considérense todos los órdenes posibles en que se puedan evaluar en registros los subárboles correspondientes de T. En cada ordenamiento, el primer subárbol correspondiente a un operando registro se puede evaluar utilizando i registros disponibles, el segundo utilizando $i - 1$ registros, y así sucesivamente. En cuanto al nodo n, añádase el costo de la instrucción R := E utilizado para concordar con el nodo n. El valor $C[i]$ es entonces el costo mínimo de todos los órdenes posibles.

Los vectores de costos para el árbol completo T se pueden calcular en forma ascendente en un tiempo linealmente proporcional al número de nodos de T. Es conveniente almacenar en cada nodo la instrucción utilizada para lograr el mejor costo para $C[i]$ para cada valor de i. El costo más pequeño en el vector para la raíz de T proporciona el costo mínimo de evaluar T.

La segunda fase del algoritmo, recorre T usando los vectores de costos para determinar los subárboles de T que se deben calcular en la memoria. La tercera, se recorre cada árbol utilizando los vectores de costos y las instrucciones asociadas para generar el código objeto final. Se genera primero el código para los subárboles calculados en posiciones de memoria. Ambas fases también se pueden implantar para ser ejecutadas en un tiempo linealmente proporcional al tamaño del árbol de expresiones.

Ejemplo 9.14. Considérese una máquina que tenga dos registros, R0 y R1, y las siguientes instrucciones, cada una con costo unitario:

R*i* := M*j*
R*i* := R*i* *op* R*j*
R*i* := R*i* *op* M*j*
R*i* := R*j*
M*i* := R*i*

En estas instrucciones, R*i* es R0 o R1, y M*j* es una posición de memoria.

Se aplicará el algoritmo de programación dinámica para generar código óptimo para el árbol sintáctico de la figura 9.29. En la primera fase se calculan los vectores de costos que se muestran en cada nodo. Para ilustrar este cálculo de costos, considérese el vector de costos en la hoja a. $C[0]$, el costo de calcular a en memoria, es 0, puesto que ya está allí. $C[1]$, el costo de calcular a en un registro, es 1 porque se puede cargar en un registro con la instrucción R0 := a. $C[2]$, el costo de cargar a en un registro cuando hay dos registros disponibles, es el mismo que con un registro disponible. El vector de costos en la hoja a es por tanto (0, 1, 1).

Considérese el vector de costos de la raíz. Primero se determina el costo mínimo de calcular la raíz con uno y dos registros disponibles. La instrucción de máquina R0 := R0 + M concuerda con la raíz porque la raíz está etiquetada con el operador +. Utilizando esta instrucción, el costo mínimo de evaluar la raíz con un registro disponible es el costo mínimo de calcular su subárbol derecho en la memoria, más el costo mínimo de calcular su subárbol izquierdo en el registro, más 1 por la instrucción. No existe otra manera. Los vectores de costos en los hijos derecho e izquierdo de la raíz muestran que el costo mínimo de calcular la raíz con un registro disponible es 5+2+1 = 8.

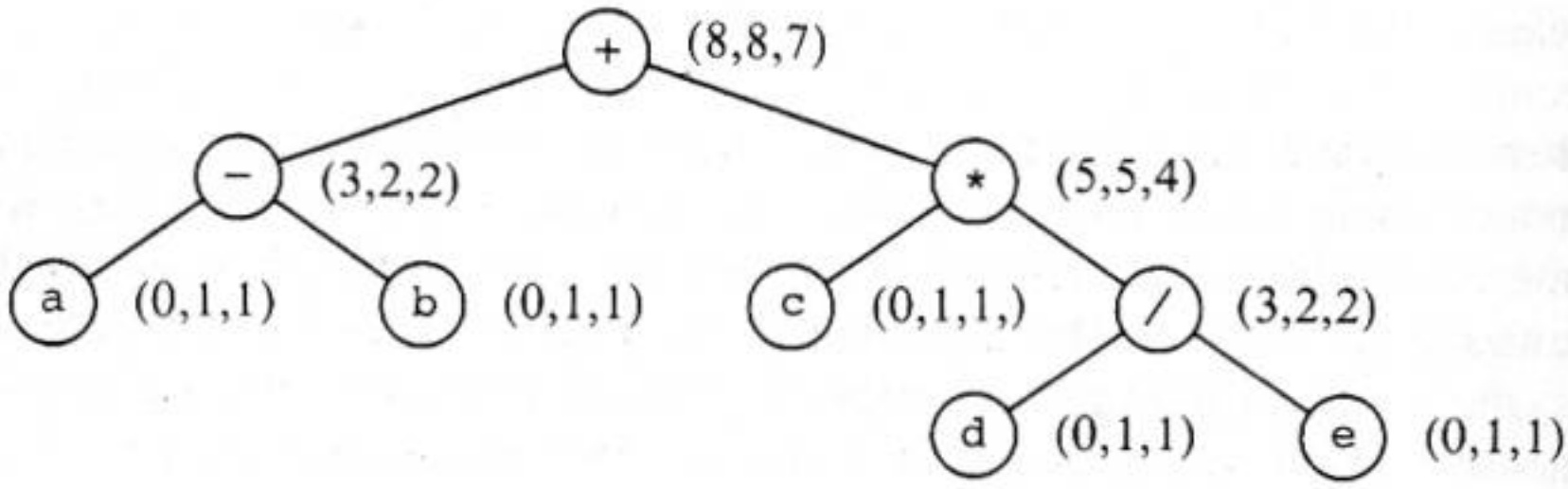


Fig. 9.29. Arbol sintáctico para `(a-b)+c*(d/e)` con vector de costos en cada nodo.

Ahora considérese el costo mínimo de evaluar la raíz con dos registros disponibles. Surgen tres casos dependiendo de la instrucción que se utilice para calcular la raíz y el orden en que se evalúen los subárboles izquierdo y derecho de la raíz.

1. Calcúlese el subárbol izquierdo con dos registros disponibles en el registro `R0`, calcúlese el subárbol derecho con un registro disponible en el registro `R1`, y utilícese la instrucción `R0 := R0 + R1` para calcular la raíz. Esta secuencia tiene un costo de 2+5+1 = 8.

2. Calcúlese el subárbol derecho con dos registros disponibles en `R1`, calcúlese el subárbol izquierdo con un registro disponible en `R0`, y utilícese la instrucción `R0 := R0 + R1`. Esta secuencia tiene un costo de 4+2+1 = 7.

3. Calcúlese el subárbol derecho en la localidad de memoria `M`, calcúlese el subárbol izquierdo con dos registros disponibles en el registro `R0`, y utilícese la instrucción `R0 := R0 + M`. Esta secuencia tiene un costo de 5+2+1 = 8.

La segunda opción da el costo mínimo de 7.

El costo mínimo de calcular la raíz en memoria se determina sumando uno al costo mínimo de calcular la raíz con todos los registros disponibles; es decir, se calcula la raíz en un registro y después se almacena el resultado en la memoria. El vector de costos en la raíz es por tanto (8,8,7).

A partir de los vectores de costos se puede fácilmente construir la secuencia de código recorriendo el árbol. Según el árbol de la figura 9.29, suponiendo que están disponibles dos registros, una secuencia de código óptima es

```
R0 := c
R1 := d
R1 := R1 / e
R0 := R0 * R1
R1 := a
R1 := R1 - b
R1 := R1 + R0
```

□

Desarrollada originalmente en Aho y Johnson [1976], esta técnica se ha utilizado en varios compiladores, incluida la segunda versión del compilador transportable de C de S. C. Johnson, PCC2. La técnica facilita la redestinación, dada la aplicabilidad de la técnica de programación dinámica en muchas clases de máquinas.

9.12 GENERADORES DE GENERADORES DE CODIGO

La generación de código supone elegir un orden de evaluación para las operaciones, asignando valores a los registros, y seleccionando las instrucciones apropiadas en el lenguaje objeto para implantar los operadores en la representación intermedia. Incluso suponiendo que el orden de evaluación viene dado y que los registros son asignados por un mecanismo independiente, la cuestión de decidir las instrucciones que se van a utilizar puede ser una gran tarea combinatoria, especialmente con una máquina con muchos modos de direccionamiento. En esta sección se estudian técnicas de reescritura de árboles que se pueden utilizar para construir la fase de selección de instrucciones de un generador de código automáticamente a partir de una especificación de alto nivel de la máquina objeto.

Generación de código mediante reescritura de árboles

A lo largo de esta sección, la entrada al proceso de generación de código será una secuencia de árboles en el nivel semántico de la máquina objeto. Los árboles se pueden obtener después de insertar las direcciones para el momento de la ejecución en la representación intermedia, como se describió en la sección 9.3.

Ejemplo 9.15. La figura 9.30 contiene un árbol para la proposición de asignación `a[i]:=b+1` en la que `a` e `i` son nombres locales cuyas direcciones en el momento de la ejecución vienen dadas como desplazamientos $const_a$ y $const_i$ a partir de `SP`, el registro que contiene el apuntador al principio del registro de activación en curso. La matriz `a` se almacena en la pila de ejecución. La asignación a `a[i]` es una asignación indirecta en la que el valor de lado derecho de la posición para `a[i]` se iguala al valor de lado derecho de la expresión `b+1`. La dirección de la matriz `a` viene dada añadiendo el valor de la constante $const_a$ al contenido del registro `SP`; el valor de `i` está en la posición que se obtiene sumando el valor de la constante $const_i$ al contenido del registro `SP`. La variable `b` es un nombre global en la posición de memoria mem_b. Para simplificar, se supone que todas las variables son de tipo carácter.

En el árbol, el operador `ind` considera su argumento como una dirección de memoria. Como el hijo izquierdo de un operador de asignación, el nodo `ind` propor-

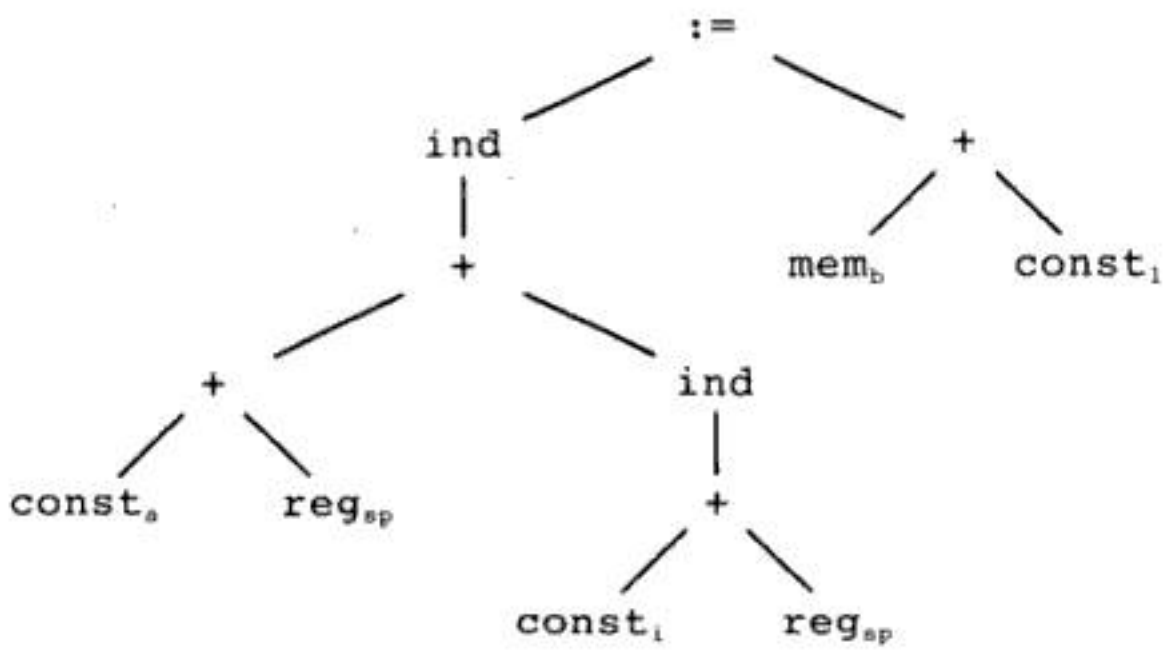


Fig. 9.30. Arbol con código intermedio para `a[i]:=b+1`.

ciona la posición en la cual debe almacenarse el valor de lado derecho que está en el lado derecho del operador de asignación. Si un argumento de un operador + o `ind` es una posición de memoria o un registro, entonces el contenido de dicha posición de memoria o registro se considera como el valor. Las hojas en el árbol son atributos de tipo con subíndices; el subíndice indica el valor del atributo. □

El código objeto se genera durante un proceso en el que el árbol de entrada se reduce a un solo nodo, aplicando al árbol una secuencia de reglas de reescritura de árboles. Cada regla de reescritura de árboles es una proposición de la forma

sustitución ← *plantilla* { *acción* }

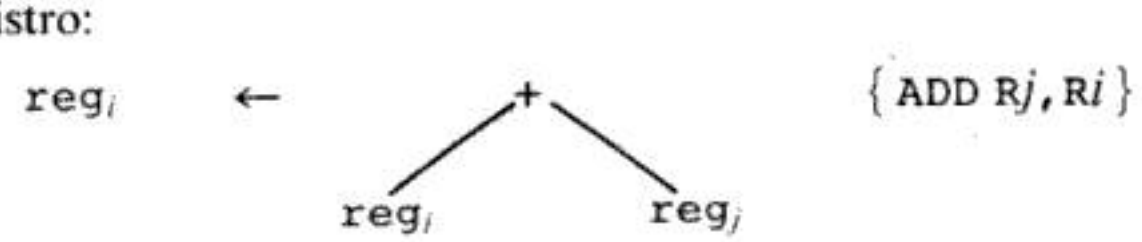
donde

1. *sustitución* es un solo nodo,
2. *plantilla* es un árbol, y
3. *acción* es un fragmento de código, como en un esquema de traducción dirigida por la sintaxis.

Un conjunto de reglas de reescritura de árboles se denomina *esquema de traducción de árboles*.

Cada plantilla de árbol representa un cálculo realizado por la secuencia de instrucciones de máquina emitida por la acción asociada. Generalmente, una plantilla corresponde a una sola instrucción de máquina. Las hojas de la plantilla son atributos con subíndices, como en el árbol de entrada. A menudo se aplican algunas limitaciones a los valores de los subíndices en las plantillas; estas limitaciones se especifican como predicados semánticos que deben cumplirse antes de que la plantilla concuerde. Por ejemplo, un predicado puede especificar que el valor de una constante pertenece a un rango determinado.

Un esquema de traducción de árboles es una forma adecuada para representar la fase de selección de instrucciones de un generador de código. Como ejemplo de regla de reestructura de árboles, considérese la regla para la instrucción de suma de registro a registro:



Esta regla se utiliza de la siguiente manera. Si el árbol de entrada contiene un subárbol que coincida con esta plantilla de árbol, es decir, un subárbol cuya raíz esté etiquetada con el operador + y cuyos hijos izquierdo y derecho sean cantidades en los registros *i* y *j*, entonces se puede sustituir ese subárbol pór un solo nodo etiquetado con $\texttt{reg}_i$ y emitir la instrucción `ADD Rj, Ri` como salida. Es posible que más de una plantilla concuerde con un subárbol en un momento dado; pronto se describirán algunos mecanismos para decidir la regla a aplicar en casos de conflicto. Se asume que la asignación de los registros se realiza antes de la selección de código.

Ejemplo 9.16. La figura 9.31 contiene reglas de reescritura de árboles para algunas instrucciones del modelo de máquina objeto de este capítulo. Estas reglas se utili-

zarán como ejemplo a lo largo de esta sección. Las primeras dos reglas corresponden a instrucciones de carga, las dos siguientes a instrucciones de almacenamiento, y el resto a cargas y sumas indizadas. Obsérvese que la regla (8) exige que el valor de la constante sea uno. Esta condición se especificaría mediante un predicado semántico. □

Un esquema de traducción de árboles funciona de la siguiente manera. Dado un árbol de entrada, las plantillas de las reglas de reescritura de árboles se aplican a sus subárboles. Si una plantilla concuerda, el subárbol concordante del árbol de entrada se sustituye por el nodo de sustitución de la regla y se realiza la acción asociada con la regla. Si la acción contiene una secuencia de instrucciones de máquina, se emiten las instrucciones. Este proceso se repite hasta que el árbol se reduce a un solo nodo, o hasta que no concuerden más plantillas. La secuencia de instrucciones generada cuando se reduce el árbol de entrada a un solo nodo constituye la salida del esquema de traducción de árboles en el árbol de entrada dado.

El proceso de especificar un generador de código es similar al de utilizar un esquema de traducción dirigida por la sintaxis para especificar un traductor. Se escribe un esquema de traducción de árboles para describir el conjunto de instrucciones de una máquina objeto. En la práctica, es mejor encontrar un esquema que posibilite la generación de una secuencia de instrucciones de costo mínimo para cada árbol de entrada. Hay varias herramientas disponibles para ayudar a construir un generador de código automáticamente a partir de un esquema de traducción de árboles.

Ejemplo 9.17. Se usará el esquema de traducción de árboles de la figura 9.31 para generar código para el árbol de entrada de la figura 9.30. Supóngase que la primera regla

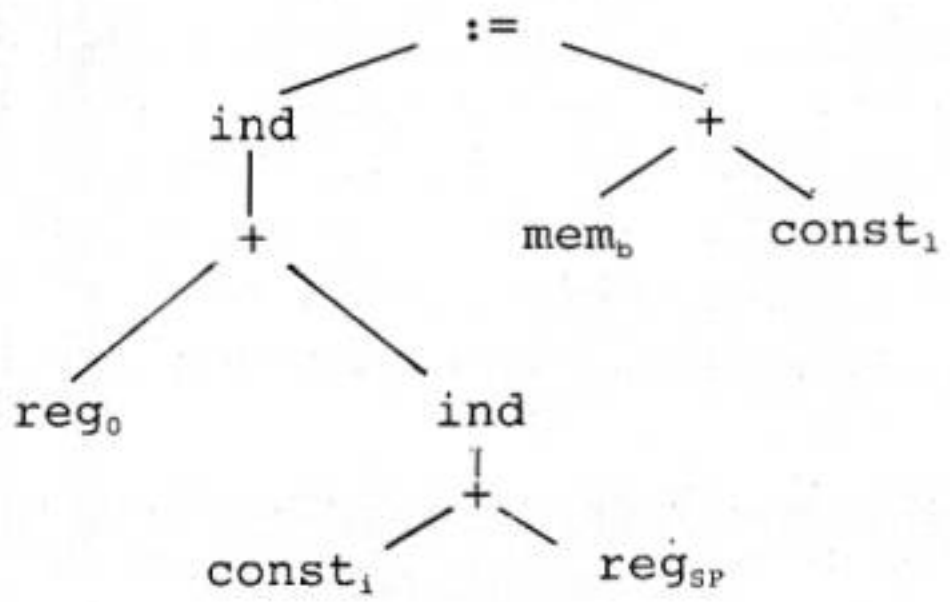
(1) reg_0 ← $const_a$ { MOV #a,R0 }

se aplica para cargar la constante a en el registro R0. La etiqueta de la hoja de la izquierda cambia entonces de $const_a$ a reg_0 y se genera la instrucción MOV #a,R0. La séptima regla

(7) reg_0 ← + { ADD SP,R0 }

(reg_0, reg_{SP})

concuerda ahora con el subárbol de la izquierda con raíz etiquetada con +. Utilizando esta regla, se reescribe este subárbol como un solo nodo etiquetado con reg_0 y se genera la instrucción ADD SP,R0. Ahora el árbol es:



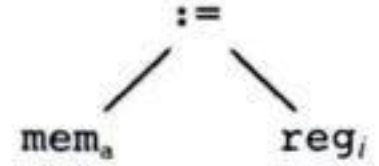
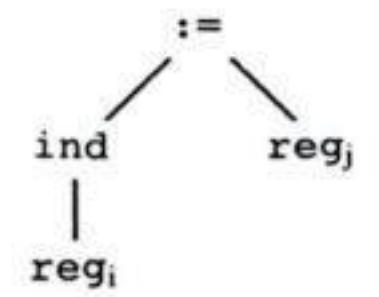
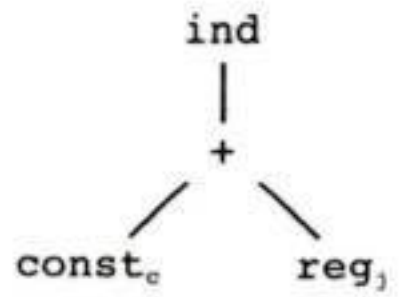
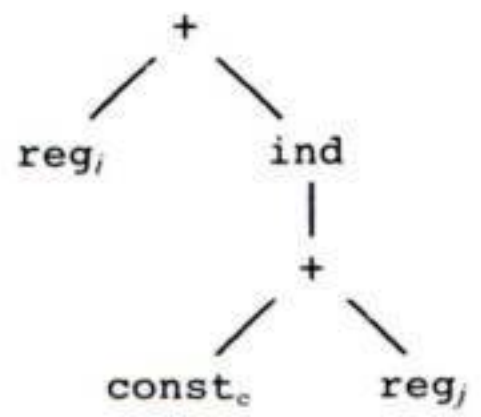
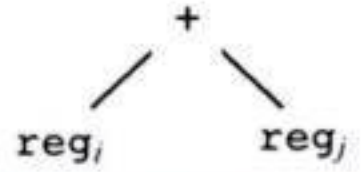
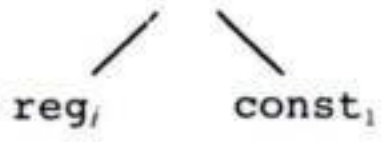
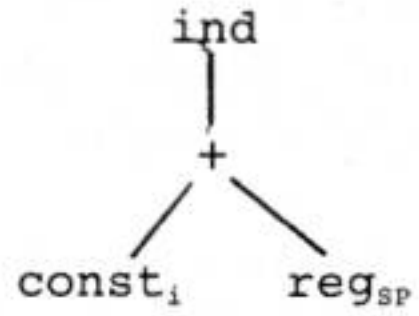
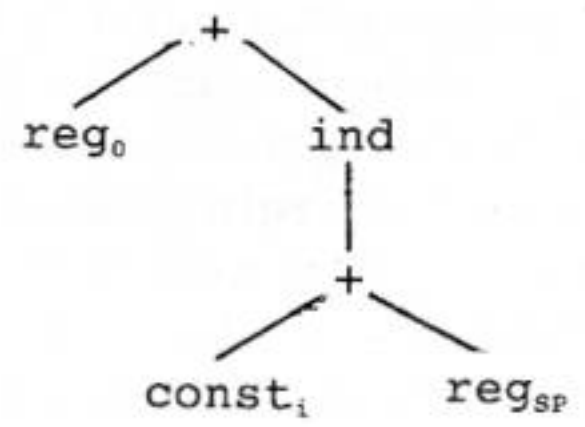
(1)	$reg_i \leftarrow const_c$	{ MOV #c,R*i* }
(2)	$reg_i \leftarrow mem_a$	{ MOV a,R*i* }
(3)	mem ← 	{ MOV R*i*, a }
(4)	mem ← 	{ MOV R*j*, *R*i }
(5)	$reg_i \leftarrow$ 	{ MOV c(R*j*),R*i* }
(6)	$reg_i \leftarrow$ 	{ ADD c(R*j*),R*i* }
(7)	$reg_i \leftarrow$ 	{ ADD R*j*,R*i* }
(8)	$reg_i \leftarrow$ 	{ INC R*i* }

Fig. 9.31. Reglas de reescritura de árboles para algunas instrucciones de la máquina objeto.

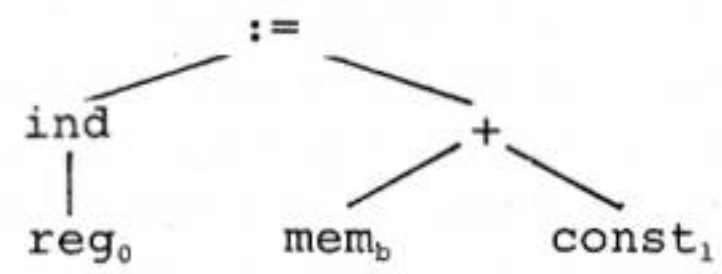
En este punto, se podría aplicar la regla (5) para reducir el subárbol



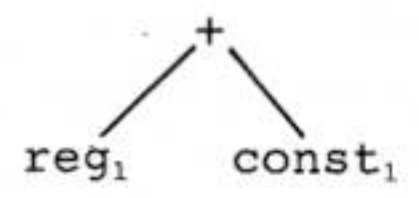
a un solo nodo etiquetado con $\texttt{reg}_1$. Sin embargo, también se puede utilizar la regla (6) para reducir el subárbol mayor



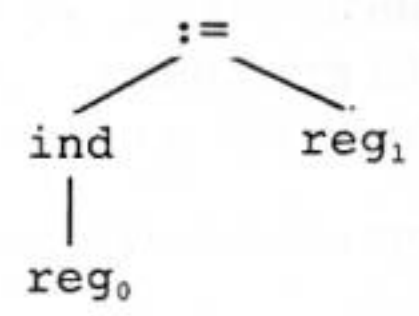
a un solo nodo etiquetado con $\texttt{reg}_0$ y generar la instrucción `ADD i(SP),R0`. Suponiendo que sea más eficiente utilizar una sola instrucción para calcular el subárbol más grande en lugar del más pequeño, se elige la última reducción para obtener



En el subárbol derecho, la regla (2) se aplica a la hoja $\texttt{mem}_b$. Esta regla genera una instrucción para cargar b en el registro 1, por ejemplo, Ahora, la regla (8) puede concordar con el subárbol



y generar la instrucción de incremento `INC R1`. En este punto, el árbol de entrada se ha reducido a



Este árbol restante concuerda con la regla (4), que reduce el árbol a un solo nodo y genera la instrucción `MOV R1,*R0`.

En el proceso de reducción del árbol a un solo nodo, se genera la siguiente secuencia de código:

```
MOV #a,R0
ADD SP,R0
ADD i(SP),R0
MOV b,Rl
INC Rl
MOV Rl,*R0
```

□

Algunos aspectos de este proceso de reducción necesitan una mayor explicación. No se ha especificado cómo se realiza la concordancia de patrones de los árboles. Tampoco se ha especificado el orden en el que se concuerdan las plantillas o qué hacer si concuerda más de una plantilla en un momento dado. Asimismo, obsérvese que si ninguna plantilla concuerda, entonces el proceso de generación de código se bloquea. En el otro extremo, puede que un solo nodo se reescriba indefinidamente, generando una secuencia infinita de instrucciones de movimiento de registros o una secuencia infinita de cargas y almacenamientos.

Una forma de realizar eficamente la concordancia de patrones es ampliando el algoritmo de concordancia de patrones de múltiples palabras clave del ejercicio 3.32 a un algoritmo de concordancia de patrones en árboles en forma descendente. Cada plantilla se puede representar mediante un conjunto de cadenas, es decir, el conjunto de caminos desde la raíz hasta las hojas. Con estas series de cadenas, se puede construir un comprobador de concordancias de patrones en árboles como en el ejercicio 3.32.

Los problemas de ordenamiento y de concordancia múltiple se pueden resolver utilizando la concordancia de patrones junto con el algoritmo de programación dinámica de la sección anterior. Un esquema de traducción de árboles se puede aumentar con información de costos, asociando a cada regla de reescritura de árboles el costo de la secuencia de instrucciones de máquina generada si se aplica dicha regla.

En la práctica, se puede aplicar el proceso de reescritura de árboles ejecutando el comprobador de concordancia de patrones durante un recorrido en profundidad del árbol de entrada y realizando las reducciones cuando los nodos sean visitados por última vez. Si se ejecuta el algoritmo de programación dinámica concurrentemente, se puede seleccionar una secuencia óptima de concordancia utilizando la información de costos asociada a cada regla. Puede que haya que retrasar la decisión relativa a una concordancia hasta conocer el costo de todas las alternativas. Con este enfoque, se puede construir rápidamente un generador de código pequeño y eficiente a partir de un esquema de reescritura de árboles. Además, con el algoritmo de programación dinámica, el diseñador del generador de código no tiene que resolver problemas de concordancias o decidir un orden para la evaluación.

Concordancia de patrones mediante análisis sintáctico

Otro enfoque consiste en utilizar un analizador sintáctico LR para realizar la concordancia de patrones. El árbol de entrada se puede considerar como una cadena utilizando su representación prefija. Por ejemplo, la representación prefija del árbol de la figura 9.30 es

$$:= \ \text{ind} + + \ \text{const}_a \ \text{reg}_{SP} \ \text{ind} + \ \text{const}_i \ \text{reg}_{SP} + \ \text{mem}_b \ \text{const}_1$$

El esquema de traducción de árboles se puede convertir en un esquema de traducción dirigida por la sintaxis sustituyendo las reglas de reescritura de árboles por las producciones de una gramática independiente del contexto en la que los lados derechos sean representaciones prefijas de las plantillas de instrucciones.

Ejemplo 9.18. El esquema de traducción dirigido por la sintaxis de la figura 9.32 se basa en el esquema de traducción de árboles de la figura 9.31. □

A partir de las producciones del esquema de traducción se construye un analizador sintáctico LR utilizando una de las técnicas de construcción de analizadores sintácticos LR del capítulo 4. El código objeto se genera emitiendo la instrucción de máquina correspondiente a cada reducción.

Una gramática para la generación de código es generalmente muy ambigua y se debe tener cuidado de cómo se resuelven los conflictos en las acciones del analizador sintáctico cuando se construya el analizador. Si no hay información sobre los costos, como regla general se prefieren reducciones grandes a pequeñas. Esto significa que en un conflicto de reducción-reducción, se favorece la reducción más grande; en un conflicto de desplazamiento reducción, se elige el desplazamiento. Este enfoque hace que se realice un número mayor de operaciones con una sola instrucción de máquina.

Existen varios aspectos en el uso del análisis sintáctico LR para la generación de código. Primero, el método de análisis sintáctico es eficiente y se comprende fácilmente, de modo que se pueden producir generadores de código fiables y eficientes utilizando los algoritmos descritos en el capítulo 4. Segundo, es relativamente fácil redestinar el generador de código obtenido; un selector de código para una máquina nueva se puede construir escribiendo una gramática para describir las instrucciones de la nueva máquina. Tercero, la calidad del código generado se puede mejorar añadiendo producciones de casos especiales para aprovechar las particularidades de la máquina.

Sin embargo, también hay algunas dificultades. Con el método de análisis sintáctico queda fijado un orden de evaluación de izquierda a derecha. Asimismo, para algunas máquinas con muchos modos de direccionamiento, la gramática que describe la máquina y el analizador sintáctico resultante pueden resultar extraordinariamente grandes. Por tanto, se necesitan técnicas especializadas para codificar y

(1)	$reg_i \rightarrow const_c$	{ MOV #c,R*i* }
(2)	$reg_i \rightarrow mem_a$	{ MOV a,R*i* }
(3)	$mem \rightarrow := mem_a\ reg_i$	{ MOV R*i*, a }
(4)	$mem \rightarrow := ind\ reg_i\ reg_j$	{ MOV Rj,*Ri }
(5)	$reg_i \rightarrow ind + const_c\ reg_i$	{ MOV c(R*j*),R*i* }
(6)	$reg_i \rightarrow + reg_i\ ind + const_c\ reg_j$	{ ADD c(R*j*),R*i* }
(7)	$reg_i \rightarrow + reg_i\ reg_j$	{ ADD R*j*,*R*i* }
(8)	$reg_i \rightarrow + reg_i\ const_1$	{ INC R*i* }

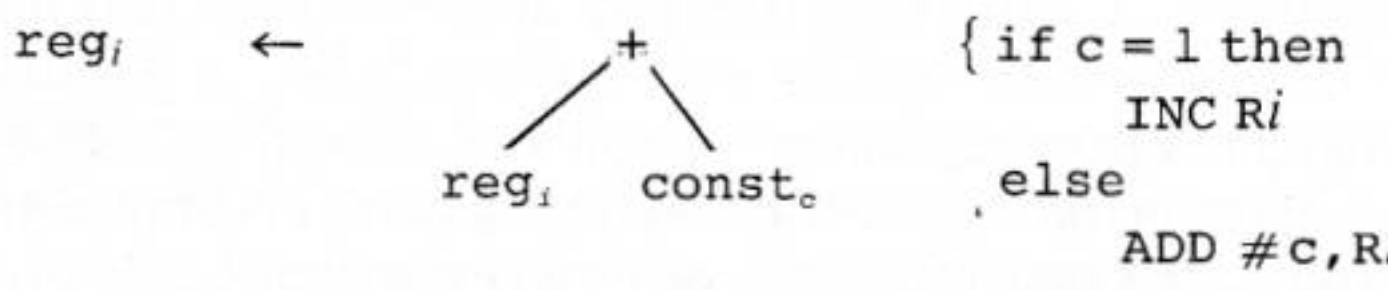
Fig. 9.32. Esquema de traducción dirigida por la sintaxis construida a partir de la figura 9.31.

procesar las gramáticas para la descripción de máquinas. También hay que asegurarse de que el analizador sintáctico resultante no se bloquee (no tenga más movimientos) mientras se hace el análisis sintáctico de un árbol de expresiones, porque la gramática no haya considerado algunos patrones de operadores o porque el analizador sintáctico haya tomado la resolución equivocada en algún conflicto de acciones del análisis sintáctico. También hay que asegurarse de que el analizador sintáctico no caiga en un lazo infinito de reducciones de producciones con símbolos simples en el lado derecho. El problema del lazo se puede resolver utilizando una técnica de división de estados en el momento en que se generen las tablas del analizador sintáctico (véase Glanville [1977]).

Rutinas para la comprobación semántica

Las hojas del árbol de entrada son atributos de tipo con subíndices, donde un subíndice asocia un valor con un atributo. En un esquema de traducción para la generación de código, aparecen los mismos atributos, pero a menudo con limitaciones en cuanto a los valores que pueden tener los subíndices. Por ejemplo, una instrucción de máquina puede exigir que el valor de un atributo pertenezca a un rango determinado o que los valores de dos atributos estén relacionados.

Estas limitaciones en cuanto a los valores de los atributos se pueden especificar como predicados que se invocan antes de realizar una reducción. De hecho, el uso general de las acciones semánticas y los predicados puede proporcionar una mayor flexibilidad y facilidad de descripción que una especificación puramente gramatical de un generador de código. Se pueden utilizar plantillas genéricas para representar clases de instrucciones y entonces las acciones semánticas se pueden utilizar para elegir instrucciones para los casos específicos. Por ejemplo, dos formas de la instrucción de suma se pueden representar con una plantilla:



Los conflictos en acciones de análisis sintáctico se pueden resolver mediante predicados que eliminan ambigüedades y que admiten varias estrategias de selección que serán utilizadas en diferentes contextos. Una descripción más pequeña de una máquina objeto es posible porque algunos aspectos de la arquitectura de la máquina, como los modos de direccionamiento, se pueden factorizar en los atributos. La complicación de este enfoque es que puede resultar difícil comprobar la validez de la gramática con atributos como una descripción fidedigna de la máquina objeto, aunque este problema es compartido en mayor o menor grado por todos los generadores de código.

EJERCICIOS

9.1 Genérese código para las siguientes proposiciones en C para la máquina objeto de la sección 9.2 suponiendo que todas las variables son estáticas. Supóngase que hay tres registros disponibles.

a) `x = 1`
b) `x = y`
c) `x = x + 1`
d) `x = a + b * c`
e) `x = a/(b+c) - d*(e+f)`

9.2 Repítase el ejercicio 9.1 suponiendo que todas las variables son automáticas (asignadas en la pila).

9.3 Genérese código para las siguientes proposiciones en C para la máquina objeto de la sección 9.2 suponiendo que todas las variables son estáticas. Supóngase que hay tres registros disponibles.

a) `x = a[i] + 1`
b) `a[i] = b[c[i]]`
c) `a[i][j] = b[i][k] * c[k][j]`
d) `a[i] = a[i] + b[j]`
e) `a[i] += b[j]`

9.4 Hágase el ejercicio 9.1 utilizando

a) el algoritmo de la sección 9.6
b) el procedimiento *gencódigo*
c) el algoritmo de programación dinámica de la sección 9.11.

9.5 Genérese código para las siguientes proposiciones en C

a) `x = f(a) + f(a) + f(a)`
b) `x = f(a)/g(b,c)`
c) `x = f(f(a))`
d) `x = ++f(a)`
e) `*p++ = *q++`

9.6 Genérese código para el siguiente programa en C

```
main()
{
    int i;
    int a[10];
    while (i <= 10)
        a[i] = 0;
}
```

9.7 Supóngase que para el lazo de la figura 9.13 se elige asignar tres registros que guarden `a`, `b` y `c`. Genérese código para los bloques de este lazo. Compárese el costo de ese código con el de la figura 9.14.

9.8 Constrúyase el grafo de interferencia entre registros para el programa de la figura 9.13.

9.9 Supóngase que para simplificar se almacenan automáticamente todos los registros en la pila (o en la memoria si no se utiliza una pila) antes de cada llamada a un procedimiento y se restablecen después del regreso. ¿Cómo in-

fluye esto en la fórmula (9.4) utilizada para evaluar la utilidad de asignar un registro a una variable dada en un lazo?

9.10 Modifíquese la función *obtenreg* de la sección 9.6 para que devuelva parejas de registros cuando sea necesario.

9.11 Dése un ejemplo de un GDA para el que la heurística de ordenamiento de los nodos de un GDA dada en la figura 9.21 no proporcione el mejor ordenamiento.

***9.12** Genérese código óptimo para las siguientes proposiciones de asignación:

a) `x := a + b * c`
b) `x := (a * - b) + (c - (d + e))`
c) `x := (a/b - c)/d`
d) `x := a + (b + c/d*e)/(f*g - h*i)`
e) `a[i,j] := b[i,j] - c[a[k,l]] * d[i+j]`

9.13 Genérese código para el siguiente programa en Pascal:

```
programa lazofor(input, output);
  var i, inicial, final: integer;
  begin
      read(inicial, final);
      for i:= inicial to final do
        writeln(i)
  end.
```

9.14 Constrúyase el GDA para el siguiente bloque básico:

```
d := b * c
e := a + b
b := b * c
a := e - d
```

9.15 ¿Cuáles son los órdenes de evaluación y los nombres legales para los valores en los nodos para el GDA del ejercicio 9.14

a) suponiendo que `a`, `b` y `c` están activos al final del bloque básico?
b) suponiendo que sólo `a` está activo al final?

9.16 En el ejercicio 9.15(b), si se va a generar código para una máquina con sólo un registro, ¿qué orden de evaluación es mejor? ¿Por qué?

9.17 Se puede modificar el algoritmo de construcción de un GDA para que tenga en cuenta asignaciones a matrices y a través de apuntadores. Cuando se hace una asignación a un elemento de una matriz se supone que se crea un nuevo valor para dicha matriz. Este nuevo valor se representa mediante un nodo cuyos hijos son el valor anterior de la matriz, el valor del índice dentro de la matriz y el valor asignado. Cuando ocurre una asignación por medio de un apuntador, se asume que se ha creado un nuevo valor para cada variable a la que pudo haber apuntado el apuntador; los hijos del nodo para cada nuevo valor son el valor del apuntador y el valor anterior de la variable que pudo

haber sido asignada. Utilizando estos supuestos, constrúyase el GDA para el siguiente bloque básico:

```
a[i] := b
*p := c
d := a[j]
e := *p
*p := a[i]
```

Supóngase que a) p puede apuntar a cualquier parte, b) p apunta sólo a b o a d. No olvide mostrar las limitaciones de orden implícitas.

9.18 Si se hace una asignación a una expresión de apuntadores o de matrices como a[i] o *p y después se utiliza sin que exista la posibilidad de que su valor haya cambiado en el ínterin, se puede reconocer y aprovechar la situación para simplificar el GDA. Por ejemplo, en el código del ejercicio 9.17, como a p no se le hace ninguna asignación entre la segunda y la cuarta proposiciones, la proposición e := *p se puede sustituir por e := c, ya que sea lo que apunte p, tendrá el mismo valor que c, aunque no se sabe a qué apunta p. Revísese el algoritmo de construcción de un GDA para aprovechar dichas conclusiones. Aplique su algoritmo al código del ejercicio 9.17.

9.19 Diséñese un algoritmo para generar código óptimo para una secuencia de proposiciones de tres direcciones de la forma a := b + c en la máquina de *n* registros del ejercicio 9.14. Las proposiciones se deben ejecutar en el orden dado. ¿Cuál es la complejidad temporal de su algoritmo?

NOTAS BIBLIOGRAFICAS

El lector interesado en las investigaciones sobre generación de código debe consultar Waite [1976a,b], Aho y Sethi [1977], Graham [1980 y 1984], Ganapathi, Fischer y Hennessy [1982], Lunell [1983] y Henry [1984]. Wulf y colaboradores [1975] analizan la generación de código para BLISS, Ammann [1977] para Pascal, y Auslander y Hopkins [1982] para PL.8.

Las estadísticas sobre el uso de los programas son útiles para el diseño de compiladores. Knuth [1971b] realizó un estudio empírico de programas en FORTRAN. Elshoff [1976] proporciona algunas estadísticas sobre el uso de PL/I, y Shimasaki y colaboradores [1980] y Carter [1982] analizan programas en Pascal. Lunde [1977], Shustek [1978] y Ditzel y McLellan [1982] analizan el rendimiento de varios compiladores en distintos conjuntos de instrucciones de computadores.

Gran parte de la heurística para la generación de código propuesta en este capítulo ha sido utilizada en varios compiladores. Freiburghouse [1974] estudia las cuentas de uso como ayuda para generar buen código para los bloques básicos. Belady [1966], en el contexto de intercambio de páginas, demostró que la estrategia utilizada en *obtenreg* de crear un registro libre expulsando de un registro la variable cuyo valor quedará sin utilizar por más tiempo es óptima. La estrategia de asignar un número fijo de registros para guardar variables dentro de la duración de un lazo

fue mencionada por Marill [1962] y Lowry y Medlock [1969] la utilizaron en la implantación de FORTRAN H.

Horwitz y colaboradores [1966] proporcionan un algoritmo para optimar el uso de registros de índice en FORTRAN. La coloración de grafos como técnica para la asignación de registros fue propuesta por J. Cocke, Ershov [1971] y Schwartz [1973]. El tratamiento de la coloración de grafos de la sección 9.7 sigue el de Chaitin y colaboradores [1981] y Chaitin [1982]. Chow y Hennessy [1984] describen un algoritmo de coloración de grafos basado en prioridades para la asignación de registros. Kennedy [1972], Johnsson [1975], Harrison [1975], Beatty [1974] y Leverett [1982] analizan otros enfoques para la asignación de registros.

Los algoritmos de etiquetado para árboles de la sección 9.10 evocan un algoritmo para nombrar ríos: la confluencia de un río mayor y un tributario menor continúa utilizando el nombre del río mayor; la confluencia de dos ríos del mismo tamaño recibe un nuevo nombre. El algoritmo de etiquetado apareció originalmente en Ershov [1958]. Anderson [1964], Nievergelt [1965], Nakata [1967], Redziejowski [1969] y Beatty [1972] han propuesto algoritmos para la generación de código utilizando este método. Sethi y Ullman [1970] utilizaron el método de etiquetado en un algoritmo que ellos demostraron que generaba código óptimo para árboles de expresiones en una amplia gama de situaciones. El procedimiento *gencódigo* de la sección 9.10 es una modificación del algoritmo de Sethi y Ullman, obra de Stockhausen [1973]. Bruno y Lassagne [1975] y Coffman y Sethi [1983] proporcionan algoritmos para la generación de código óptimo para árboles de expresiones si la máquina objeto tiene registros que deben ser utilizados como una pila.

Aho y Johnson [1976] diseñaron el algoritmo de programación dinámica descrito en la sección 9.11. Se utilizó este algoritmo como base para el generador de código del compilador transportable de C de S. C. Johnson, PCC2, y también lo utilizó Ripken [1977] en un compilador para la máquina IBM 370. Knuth [1977] generalizó el algoritmo de programación dinámica para máquinas con clases de registros asimétricos, como la IBM 7090 y la CDC 6600. Al desarrollar esta generalización, Knuth consideró la generación de código como un problema de análisis sintáctico para gramáticas independientes del contexto.

Floyd [1961] proporciona un algoritmo para manejar subexpresiones comunes en expresiones aritméticas. La partición de los GDA en árboles y el uso de un procedimiento como *gencódigo* en los árboles independientemente es obra de Waite [1976a]. Sethi [1975] y Bruno y Sethi [1976] demostraron que el problema de generación de código para GDA es NP completo. Aho, Johnson y Ullman [1977a] demuestran que el problema continúa siendo NP completo incluso con máquinas de un solo registro o de infinitos registros. Aho, Hopcroft y Ullman [1974] y Garey y Johnson [1979] analizan la importancia de lo que significa que un problema sea NP completo.

Las transformaciones sobre bloques básicos han sido estudiadas por Aho y Ullman [1972a] y por Downey y Sethi [1978]. McKeeman [1965], Fraser [1979], Davidson y Fraser [1980 y 1984a,b], Lamb [1981] y Giegerich [1983] estudian la optimación mediante mirilla. Tanenbaum, van Staveren y Stevenson [1982] propugnan el uso de la optimación mediante mirilla también en el código intermedio.

Wasilew [1971], Weingart [1973], Johnson [1978] y Cattell [1980] tratan la ge-

neración de código como un proceso de reescritura de árboles. El ejemplo de reescritura de árboles de la sección 9.12 proviene de Henry [1984]. Aho y Ganapathi [1985] propusieron la combinación de una concordancia eficiente de patrones de árboles con la programación dinámica que se menciona en la misma sección. Tjiang [1986] implantó un lenguaje para la generación de código llamado Twig, basado en los esquemas de traducción de la sección 9.12. Kron [1975], Huet y Levy [1979] y Hoffman y O'Donnell [1982] describen algoritmos generales para la concordancia de patrones de árboles.

El enfoque de Graham y Glanville para la generación de código mediante el uso de un analizador sintáctico LR para la selección de instrucciones se describe y evalúa en Glanville [1977], Glanville y Graham [1978], Graham [1980 y 1984], Henry [1984] y Aigrain y colaboradores [1984]. Ganapathi [1980] y Ganapathi y Fischer [1982] han utilizado gramáticas con atributos para especificar e implantar generadores de código.

Otras técnicas para automatizar la construcción de generadores de código han sido propuestas por Fraser [1977], Cattell [1980] y Leverett y colaboradores [1980]. La transportabilidad de compiladores también es analizada por Richards [1971 y 1977]. Szymanski [1978] y Leverett y Szymanski [1980] describen técnicas para encadenar instrucciones de salto dependientes de la extensión. Yannakakis [1985] tiene un algoritmo de tiempo polinomial para el ejercicio 9.19.

CAPITULO 10

Optimación de código

Idealmente, los compiladores deberían producir código objeto que fuera tan bueno como para ser escrito a mano. La realidad es que este objetivo sólo se alcanza en pocos casos y difícilmente. Sin embargo, a menudo se puede lograr que el código directamente producido por los algoritmos de compilación se ejecute más rápidamente o que ocupe menos espacio, o ambas cosas. Esta mejora se consigue mediante transformaciones de programas que tradicionalmente se denominan *optimaciones*, aunque el término "optimación" no es adecuado porque rara vez existe la garantía de que el código resultante sea el mejor posible. Los compiladores que aplican transformaciones para mejorar el código se denominan *compiladores optimadores*.

En este capítulo se consideran principales las optimaciones independientes de la máquina, que son transformaciones de programas que mejoran el código objeto sin tener en cuenta las propiedades de la máquina objeto. Las optimaciones dependientes de la máquina, como la asignación de registros y la utilización de secuencias de instrucciones de máquina especiales ("expresiones idiomáticas" o "modismos" de la máquina) se estudiaron en el capítulo 9.

Se obtiene el mayor beneficio con el mínimo esfuerzo si se pueden identificar las partes de un programa ejecutadas frecuentemente y después conseguir que esas partes sean lo más eficientes posible. Se dice que la mayoría de los programas emplea el noventa por ciento de su tiempo de ejecución en el diez por ciento del código. Aunque los porcentajes reales pueden variar, ocurre a menudo que una pequeña fracción de un programa contabilice la mayor parte del tiempo de ejecución. Perfilar el tiempo de ejecución de un programa con datos de entrada representativos identifica con exactitud las regiones de un programa por las que se ha pasado muchas veces. Desgraciadamente, un compilador no tiene la ventaja de contar con una muestra de datos de entrada, así que debe acertar dónde se encuentran las partes críticas del programa.

En la práctica, los lazos internos del programa son buenos candidatos para realizar mejoras. En un lenguaje que resalta las construcciones de control como las proposiciones **while** y **for**, los lazos pueden ser evidentes según la sintaxis del programa; en general, un proceso, llamado análisis del flujo del control, identifica los lazos dentro del grafo de flujo de un programa.

Este capítulo es una cornucopia de transformaciones de optimación útiles y de técnicas para implantarlas. La mejor técnica para decidir las transformaciones que valen la pena poner en un compilador consiste en reunir estadísticas sobre los programas fuente y evaluar la ventaja de un conjunto dado de optimaciones sobre una muestra representativa de programas fuente reales. El capítulo 12 describe transformaciones que han demostrado ser útiles en la optimación de compiladores para varios lenguajes.

Uno de los temas de este capítulo es el análisis del flujo de datos, un proceso para recopilar información sobre el modo en que se utilizan las variables en un programa. La información recopilada en varios puntos de un programa se puede relacionar utilizando sencillas ecuaciones de conjuntos. Se presentan varios algoritmos para recopilar la información utilizando análisis del flujo de datos para utilizar de manera efectiva esta información en la optimación. También se considera el impacto de construcciones del lenguaje como los procedimientos y los apuntadores en la optimación.

Las últimas cuatro secciones de este capítulo se ocupan de material más avanzado. Abarcan algunas ideas de teoría de grafos relevantes para el análisis de flujo de control y se aplican estas ideas al análisis de flujo de datos. El capítulo concluye con un estudio sobre herramientas de propósito general para el análisis del flujo de datos y técnicas para depurar código optimado. La parte más importante del capítulo se concentra en las técnicas de optimación aplicadas a los lenguajes en general. En el capítulo 12 se revisan algunos compiladores que utilizan estas ideas.

10.1 INTRODUCCION

Para crear un programa en lenguaje objeto eficiente, un programador necesita más que un compilador optimador. En esta sección se revisan las opciones de que dispone el programador y el compilador para crear programas objeto eficientes. Se mencionan los tipos de transformaciones para mejorar el código que supone que el programador y quien escribe un compilador utilizarán para mejorar el rendimiento de un programa. También se considera la representación de programas sobre los que se aplicarán las transformaciones.

Criterios para las transformaciones para mejorar el código

Dicho de una manera sencilla, las mejores transformaciones de programas son las que producen el mayor beneficio con el menor esfuerzo. Las transformaciones realizadas por un compilador optimador deben tener varias propiedades.

Primero, una transformación debe preservar el significado de los programas. Es decir, una "optimación" no debe cambiar el resultado producido por un programa para una entrada dada, o causar un error, como una división por cero, que no estuviera presente en la versión original del programa fuente. La influencia de este criterio impregna este capítulo. En todo momento se toma el enfoque "seguro" de desaprovechar la oportunidad de aplicar una transformación en lugar de arriesgarse a cambiar lo que hace el programa.

Segundo, una transformación debe, como promedio, acelerar los programas en una cantidad mensurable. En ocasiones interesa reducir el espacio que ocupa el código compilado, aunque el tamaño del código tiene menos importancia que la que tenía antes. Por supuesto, no toda transformación consigue mejorar todo programa y, ocasionalmente, una "optimación" puede ralentizar ligeramente un programa, mientras en general mejore las cosas.

Tercero, una transformación debe valer la pena. No tiene sentido que el escritor de un compilador haga el esfuerzo intelectual de aplicar una transformación que mejore el código y que el compilador gaste el tiempo adicional compilando programas fuente si este esfuerzo no es recompensado cuando se ejecutan los programas objeto. Algunas transformaciones locales o "de mirilla" del tipo estudiado en la sección 9.9 son lo bastante sencillas y ventajosas como para ser incluidas en un compilador.

Algunas transformaciones sólo se pueden aplicar después de un análisis detallado y que lleva su tiempo del programa fuente, de modo que tiene poco sentido aplicarlas a programas que sólo se ejecutarán pocas veces. Por ejemplo, es probable que un compilador no optimador rápido sea más útil durante la depuración o para "trabajos de estudiantes" que se ejecutarán con éxito pocas veces y luego se desecharán. Sólo cuando el programa en cuestión toma una fracción significativa de los ciclos de la máquina, la calidad mejorada del código justifica el tiempo empleado en ejecutar un compilador optimador sobre el programa.

Obtención de un mayor rendimiento

Generalmente se obtienen mejoras espectaculares en el tiempo de ejecución de un programa —como reducir el tiempo de ejecución de unas horas a unos segundos— mejorando el programa a todos los niveles, desde el nivel fuente hasta el nivel objeto, como se sugiere en la figura 10.1. En cada nivel, las opciones disponibles están entre los dos extremos de encontrar un algoritmo mejor y de implantar un algoritmo dado, así que se realizan menos operaciones.

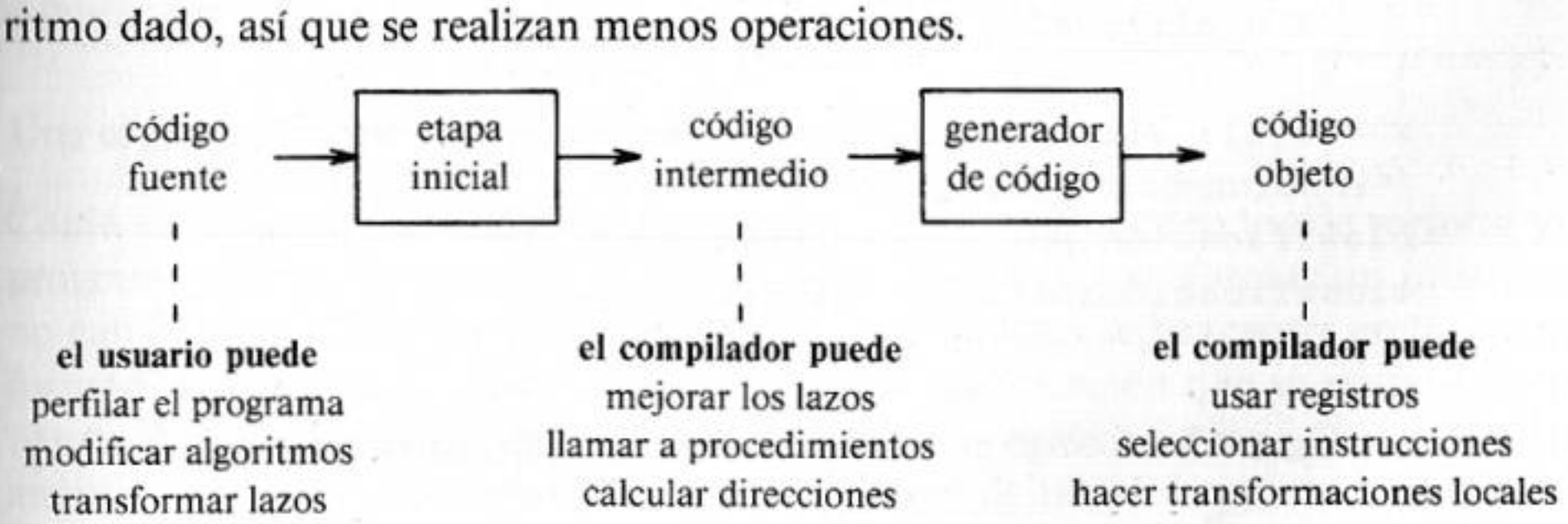


Fig. 10.1. Lugares en que el usuario y el compilador pueden hacer mejoras potenciales.

Las transformaciones de los algoritmos producen ocasionalmente mejoras espectaculares en el tiempo de ejecución. Por ejemplo, Bentley [1982] dice que el tiempo de ejecución de un programa para clasificar N elementos se redujo de $2.02N^2$ microsegundos a $12N\log_2 N$ microsegundos cuando una "clasificación por inserción" se

sustituyó por una "clasificación por particiones" (*quicksort*)[1]. Para $N = 100$, la sustitución acelera el programa en un factor de 2.5. Para $N = 100\,000$, la mejora es mucho mayor: la sustitución acelera el programa en un factor de más de mil.

Lamentablemente, ningún compilador puede encontrar el mejor algoritmo para un problema dado. Sin embargo, a veces un compilador puede sustituir una secuencia de operaciones por una secuencia algebraicamente equivalente, y con ello reducir significativamente el tiempo de ejecución de un programa. Dichos ahorros son más habituales cuando se aplican transformaciones algebraicas a los programas en lenguajes de muy alto nivel; por ejemplo, lenguajes de consulta para bases de datos (véase Ullman [1982]).

En esta sección y en la siguiente, se utilizará un programa de clasificación llamado clasificación por particiones para ilustrar el efecto de varias transformaciones para mejorar el código. El programa en C de la figura 10.2 viene de Sedgewick [1978], donde se analiza la optimación a mano de dicho programa. Aquí no se analizarán los aspectos algorítmicos de este programa —de hecho, `a[0]` debe contener el menor y `a[max]` el mayor elemento por clasificarse para que el programa funcione—.

```
void clasificación_por_particiones(m,n)
int m,n;
{
    int i,j;
    int v,j;
    if ( n <= ) return;
    /* el fragmento comienza aquí */
    i = m-1; j = n; v = a[n];
    while(1)
        do i = 1+1; while ( a[i] < v );
        do j = j-1; while ( a[j] > v );
        if ( i >= j ) break;
        x = a[i]; a[i] = a[j]; a[j] = x;
    }
    x = a[i]; a[i] = a[n]; a[n] = x;
    /* el fragmento termina aquí */
    clasificación_por_particiones(m,j);
    clasificación_por_particiones(i+1,n);
}
```

Fig. 10.2. Código en C para clasificación_por_particiones.

Puede que sea posible realizar algunas transformaciones de mejora del código a nivel del programa fuente. Por ejemplo, en un lenguaje como Pascal o FORTRAN, un programador sólo puede hacer referencia a elementos de una matriz de la manera habitual; por ejemplo, como `b[i,j]`. En el nivel del lenguaje intermedio, sin

[1] Véase Aho, Hopcroft y Ullman [1983] para un estudio de estos algoritmos de clasificación y de sus velocidades.

embargo, pueden aparecer nuevas oportunidades para mejorar el código. El código de tres direcciones, por ejemplo, proporciona muchas oportunidades para mejorar los cálculos de las direcciones, especialmente en lazos. Considérese el código de tres direcciones para determinar el valor de `a[i]`, suponiendo que cada elemento de la matriz ocupa cuatro bytes:

```
t1 := 4*i;   t2 := a[t1]
```

Un código intermedio ingenuo recalculará `4*i` cada vez que `a[i]` aparezca en el programa fuente, y el programador no controla los cálculos redundantes de las direcciones porque están implícitas en la implantación del lenguaje, en lugar de ser explícitas en el código escrito por el usuario. En estas situaciones, corresponde al compilador depurarlas. Sin embargo, en un lenguaje como C, esta transformación puede ser realizada al nivel fuente por el programador, ya que las referencias a los elementos de una matriz se pueden reescribir sistemáticamente utilizando apuntadores para hacerlas más eficientes. Esta reescritura es similar a las transformaciones que tradicionalmente aplican los compiladores optimadores de FORTRAN.

En el nivel de la máquina objeto, corresponde al compilador utilizar adecuadamente los recursos de la máquina. Por ejemplo, conservar las variables más utilizadas dentro de registros puede reducir significativamente el tiempo incluso a la mitad. De nuevo, C permite a un programador aconsejar al compilador que conserve algunas variables en registros, pero la mayoría de los lenguajes no lo permite. De manera similar, el compilador puede acelerar los programas de forma significativa eligiendo instrucciones que aprovechan los modos de direccionamiento de la máquina para hacer en una instrucción lo que ingenuamente se supondría que necesitaran dos o tres, como se estudió en el capítulo 9.

Aunque es posible que el programador mejore el código, puede ser más conveniente que el compilador realice algunas de las mejoras. Si se puede confiar en que el compilador genere código eficiente, entonces el usuario puede concentrarse en escribir código claro.

Una organización para un compilador optimador

Como ya se ha mencionado, existen varios niveles en los que se puede mejorar un programa. Como las técnicas necesarias para analizar y transformar un programa no cambian significativamente con el nivel, este capítulo se concentra en las transformaciones de código intermedio utilizando la organización que se muestra en la figura 10.3. La fase de mejora del código consta del análisis de flujo de control y el análisis de flujo de datos seguidos de la aplicación de transformaciones. El generador de código, que se estudia en el capítulo 9, produce el programa objeto a partir del código intermedio transformado.

Para una mejor presentación, se supone que el código intermedio consta de proposiciones de tres direcciones. El código intermedio, del tipo producido por las técnicas del capítulo 8, para una parte del programa de la figura 10.2, se muestra en la figura 10.4. Con otras representaciones intermedias, las variables temporales $t_1, t_2, \ldots, t_{15}$ de la figura 10.4 no tienen que aparecer explícitamente, como se estudia en el capítulo 8.

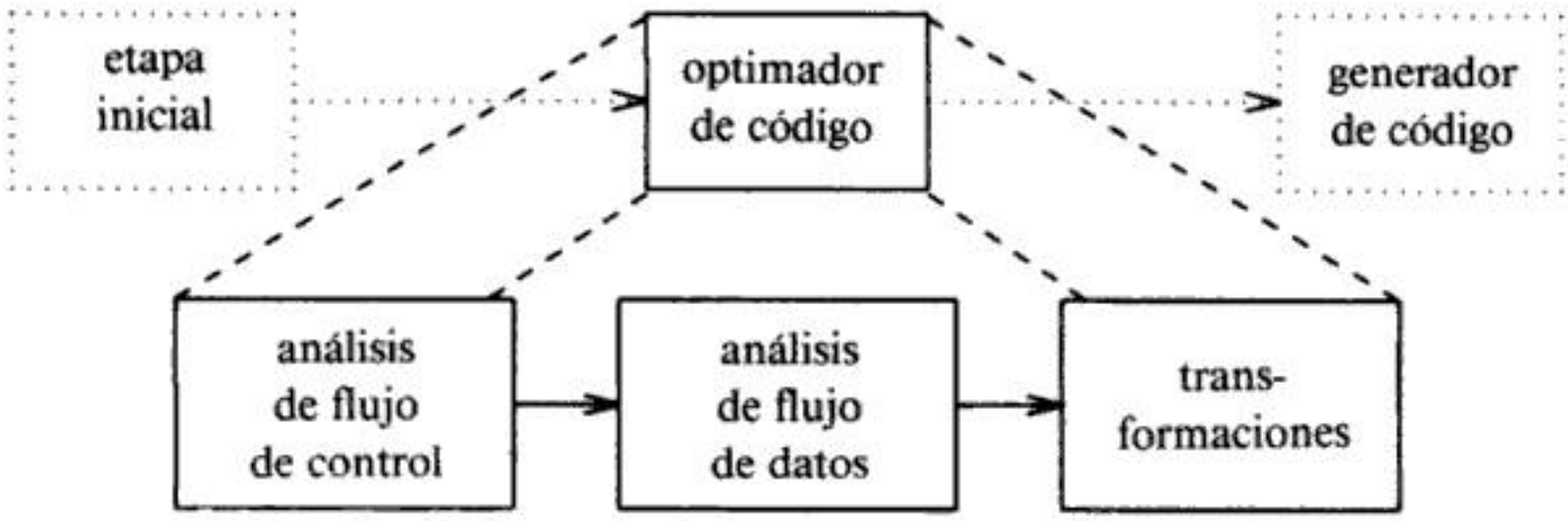


Fig. 10.3. Organización del optimador de código.

La organización de la figura 10.3 tiene las siguientes ventajas:

1. Las operaciones necesarias para implantar construcciones de alto nivel se hacen explícitas en el código intermedio, de modo que es posible optimarlas. Por ejemplo, los cálculos de direcciones para `a[i]` se hacen explícitos en la figura 10.4, de modo que el recálculo de expresiones como `4*i` se puede eliminar como se estudia en la siguiente sección.

```
(1)    i := m-1
(2)    j := n
(3)    t1 := 4*n
(4)    v := a[t1]
(5)    i := i+1
(6)    t2 := 4*i
(7)    t3 := a[t2]
(8)    if t3 < v goto (5)
(9)    j := j-1
(10)   t4 := 4*j
(11)   t5 := a[t4]
(12)   if t5 > v goto (9)
(13)   if i >= j goto (23)
(14)   t6 := 4*i
(15)   x := a[t6]
(16)   t7 := 4*i
(17)   t8 := 4*j
(18)   t9 := a[t8]
(19)   a[t7] := t9
(20)   t10 := 4*j
(21)   a[t10] := x
(22)   goto (5)
(23)   t11 := 4*i
(24)   x := a[t11]
(25)   t12 := 4*i
(26)   t13 := 4*n
(27)   t14 := a[t13]
(28)   a[t12] := t14
(29)   t15 := 4*n
(30)   a[t15] := x
```

Fig. 10.4. Código de tres direcciones para el fragmento de la figura 10.2.

2. El código intermedio puede ser (relativamente) independiente de la máquina objeto, de modo que el optimador no tiene que cambiar mucho si el generador de código se sustituye por uno para una máquina diferente. El código intermedio de la figura 10.4 asume que cada elemento de la matriz a ocupa cuatro bytes. Algunos códigos intermedios, como el código P para Pascal, dejan al generador de código que proporcione el tamaño de los elementos de las matrices, así que el código intermedio es independiente del tamaño de la palabra de una má-

quina. Se podría haber hecho lo mismo en este código intermedio si se sustituyera 4 por una constante simbólica.

En el optimador de código, los programas se representan mediante grafos de flujo, en los que las aristas indican el flujo del control y los nodos representan bloques básicos, como se vio en la sección 9.4. A menos que se especifique lo contrario, un programa significa un solo procedimiento. En la sección 10.8 se analiza la optimación entre procedimientos.

Ejemplo 10.1. La figura 10.5 contiene el grafo de flujo para el programa de la figura 10.4. B_1 es el nodo inicial. Todos los saltos condicionales e incondicionales hacia proposiciones en la figura 10.4 se han sustituido en la figura 10.5 por saltos al bloque del cual son líderes las proposiciones.

En la figura 10.5 hay tres lazos, B_2 y B_3 son lazos en sí mismos. Los bloques B_2, B_3, B_4 y B_5 juntos forman lazo, con entrada B_2. □

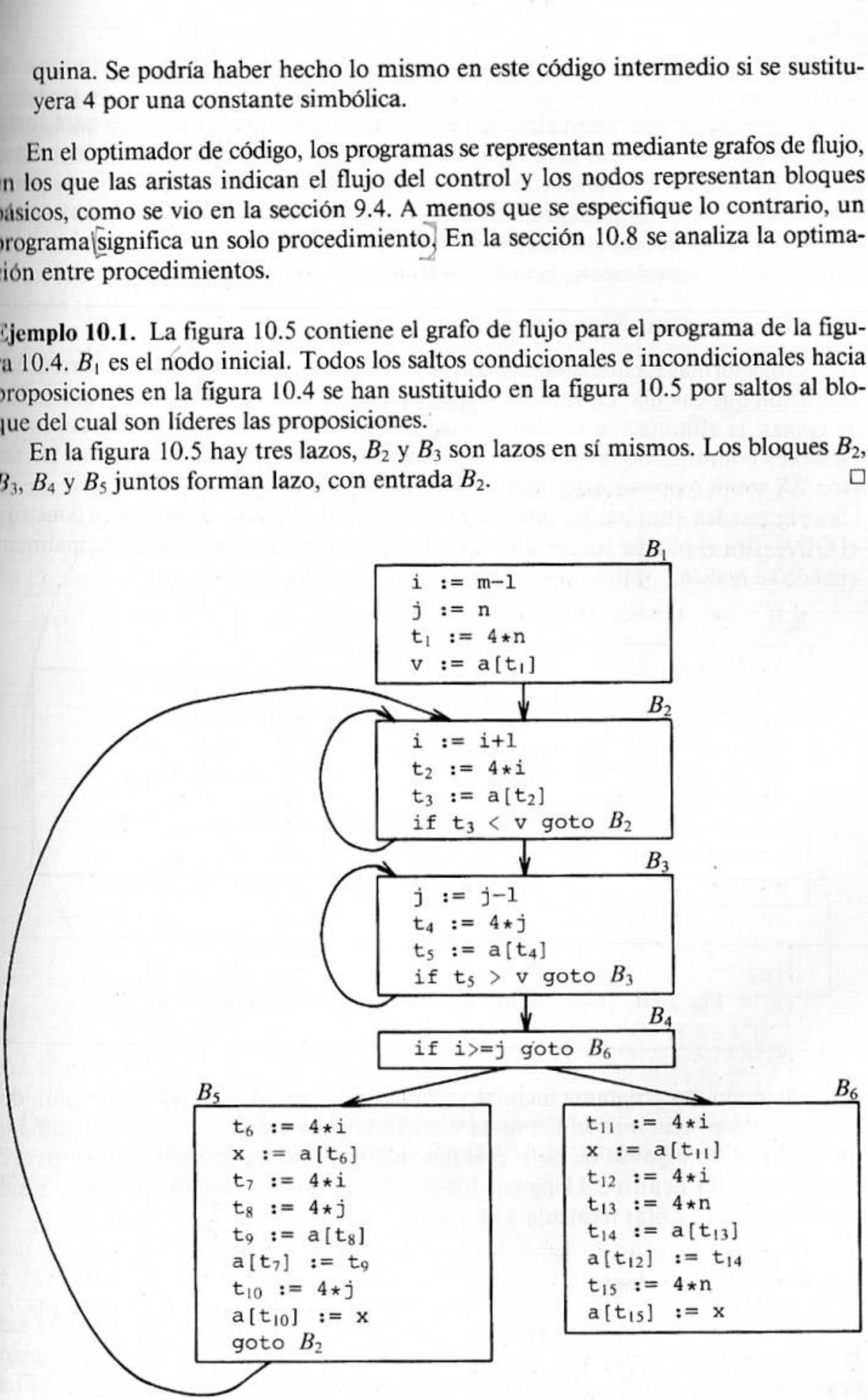


Fig. 10.5. Grafo de flujo.

10.2 LAS PRINCIPALES FUENTES PARA LA OPTIMACION

En esta sección se presentan algunas de las transformaciones más útiles para mejorar el código. Las técnicas para aplicar estas transformaciones se presentan en posteriores secciones. Una transformación de un programa se denomina *local* si se puede realizar observando sólo las proposiciones de un bloque básico; en caso contrario se denomina *global*. Muchas transformaciones se pueden realizar tanto a nivel local como global. Generalmente, primero se realizan las transformaciones locales.

Transformaciones que preservan la función

Hay varias formas en que un compilador puede mejorar un programa sin modificar la función que calcula. La eliminación de subexpresiones comunes, la programación de copias, la eliminación de código inactivo y el cálculo previo de constantes son ejemplos comunes de dichas transformaciones que preservan la función. La sección 9.8 sobre representación por medio de GDA para los bloques básicos mostró cómo se pueden eliminar las subexpresiones comunes locales conforme se construye el GDA para el bloque básico. Las otras transformaciones aparecen principalmente cuando se realizan optimaciones globales, y se estudiarán una a una.

B_5

```
t6 := 4*i
x := a[t6]
t7 := 4*i
t8 := 4*j
t9 := a[t8]
a[t7] := t9
t10 := 4*j
a[t10] := x
goto B2
```

(a) Antes

B_5

```
t6 := 4*i
x := a[t6]
t8 := 4*j
t9 := a[t8]
a[t6] := t9
a[t8] := x
goto B2
```

(b) Después

Fig. 10.6. Eliminación de subexpresiones comunes locales.

A menudo, un programa incluirá varios cálculos del mismo valor, como un desplazamiento en una matriz. Como se mencionó en la sección 10.1, el programador no puede evitar algunos de estos cálculos duplicados porque están bajo el nivel de detalle accesible dentro del lenguaje fuente. Por ejemplo, el bloque B_5 que se muestra en la figura 10.6(a) recalcula `4*i` y `4*j`.

Subexpresiones comunes

Una ocurrencia de una expresión E se denomina *subexpresión común* si E ha sido previamente calculada y los valores de las variables dentro de E no han cambiado desde el cálculo anterior. Se puede evitar recalcular la expresión si se puede utilizar el valor calculado previamente. Por ejemplo, las asignaciones a t_7 y a t_{10} tienen las

subexpresiones comunes `4*i` y `4*j`, respectivamente, en el lado derecho de la figura 10.6(a). Han sido eliminados en la figura 10.6(b) utilizando t_6 en lugar de t_7 y t_8 en lugar de t_{10}. Se obtendría este cambio si se reconstruyera el código intermedio a partir del GDA para el bloque básico.

Ejemplo 10.2. En la figura 10.7 se muestra el resultado de eliminar subexpresiones comunes globales y locales de los bloques B_5 y B_6 en el grafo de flujo de la figura 10.5. Primero se analiza la transformación de B_5 y después se mencionan algunas sutilezas relativas a matrices.

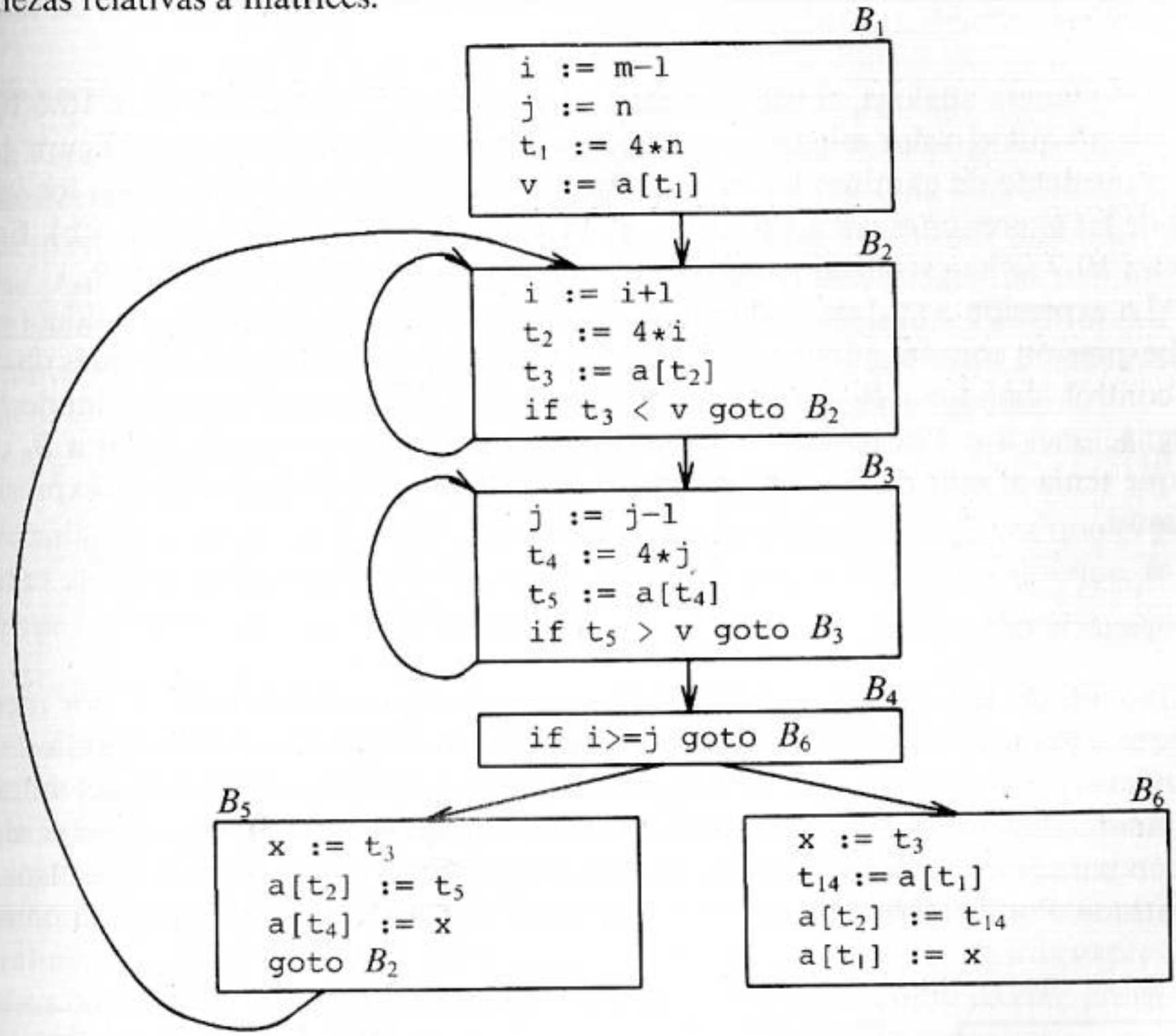


Fig. 10.7. B_5 y B_6 después de la eliminación de subexpresiones comunes.

Después de haber eliminado las subexpresiones comunes, B_5 todavía tiene que evaluar `4*i` y `4*j`, como se muestra en la figura 10.6(b). Ambas son subexpresiones comunes; en particular, las tres proposiciones

```
t8 := 4*j;   t9 := a[t8];   a[t8] := x
```

en B_5 se pueden sustituir por

```
t9 := a[t4];   a[t4] := x
```

utilizando t_4 calculado en el bloque B_3. En la figura 10.7, obsérvese que a medida que el control pasa desde la evaluación de `4*j` en B_3 hasta B_5, no hay cambio en `j`, de modo que se puede utilizar t_4 si se necesita `4*j`.

Otra subexpresión común aparece en B_5 después de que t_4 sustituye a t_8. La nueva expresión a[t_4] corresponde al valor de a[j] en el nivel fuente. No sólo j mantiene su valor a medida que el control sale de B_3 y después entra a B_5, sino que también lo mantiene a[j], un valor calculado dentro de un temporal t_5, porque no hay asignaciones a elementos de la matriz a en el ínterin. Las proposiciones

```
t9 := a[t4];   a[t6] := t9
```

en B_5 se pueden por tanto sustituir por

```
a[t6] := t5
```

De manera análoga, el valor asignado a x en el bloque B_5 de la figura 10.6(b) es el mismo que el valor asignado a t_3 en el bloque B_2. El bloque B_5 de la figura 10.7 es el resultado de eliminar las subexpresiones comunes correspondientes a los valores de las expresiones del nivel fuente a[i] y a[j] de B_5 en la figura 10.6(b). En la figura 10.7 se han realizado una serie de transformaciones similares con B_6.

La expresión a[t_1] en los bloques B_1 y B_6 de la figura 10.7 no se considera una subexpresión común, aunque t_1 se puede utilizar en ambos lugares. Después de que el control abandona B_1 y antes de que llega a B_6 puede pasar por B_5, donde hay asignaciones a a. Por tanto. a[t_1] puede no tener el mismo valor al llegar a B_6 que el que tenía al salir de B_1, y no es seguro considerar a[t_1] como una subexpresión común. □

Propagación de copias

El bloque B_5 de la figura 10.7 se puede mejorar aún más eliminando x por medio de dos nuevas transformaciones. Una concierne las asignaciones de la forma f:=g llamadas *proposiciones de copia*, o *copias* simplemente. Si se estudia más detalladamente el ejemplo 10.2, las copias habrían surgido mucho antes porque el algoritmo para eliminar las subexpresiones comunes las introduce, al igual que otros algoritmos. Por ejemplo, cuando en la figura 10.8 se elimina la subexpresión común

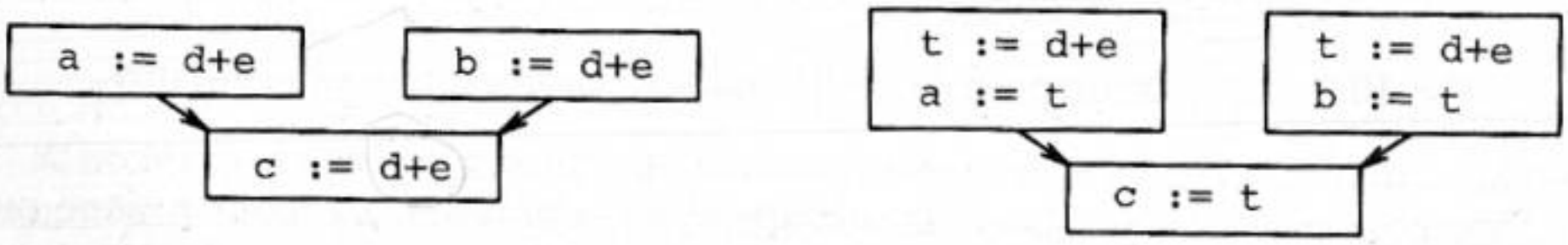


Fig. 10.8. Copias introducidas durante la eliminación de subexpresiones comunes.

en c:=d+e, el algoritmo utiliza una nueva variable t para guardar el valor de d+e. Como el control puede llegar a c:=d+e, ya sea después de hacer la asignación a a o después de hacer la asignación a b, sería incorrecto sustituir c:=d+e por c:=a o por c:=b.

La idea en que se basa la transformación de propagación de copias es utilizar g por f, siempre que sea posible después de la proposición de copia f:=g. Por ejem-

plo, la asignación `x:=t3` en el bloque B_5 de la figura 10.7 es una copia. La propagación de copias aplicada a B_5 produce:

```
x := t3
a[t2] := t5
a[t4] := t3                    (10.1)
goto B2
```

Esto puede no parecer una mejora, pero como ya se verá, da la oportunidad de eliminar las asignaciones a x.

Eliminación de código inactivo

Una variable está activa en un punto de un programa si su valor puede ser utilizado posteriormente; en caso contrario está inactiva en ese punto. Una idea afín es el código inactivo o inútil, proposiciones que calculan valores que nunca llegan a utilizarse. Aunque es improbable que el programador introduzca código inactivo intencionadamente, puede aparecer como resultado de transformaciones anteriores. Por ejemplo, en la sección 9.9 se estudió el uso de `depura` que se asigna a falso o verdadero en varios puntos del programa, y se utiliza en proposiciones como

```
if (depura) print . . .        (10.2)
```

Mediante un análisis del flujo de datos, es posible concluir que cada vez que el programa alcanza dicha proposición, el valor de `depura` es falso. Generalmente, lo es porque hay una proposición determinada

```
depura := false
```

que se puede considerar la última asignación a `depura` antes de hacer la comprobación (10.2), independientemente de la secuencia de ramificaciones que tome en realidad el programa. Si la propagación de copias sustituye `depura` por el valor de verdad **false**, entonces la proposición de impresión **print** está inactiva porque no se puede alcanzar. Se puede eliminar la prueba y la impresión del código objeto. Generalmente, deducir en el momento de la compilación que el valor de una expresión es una constante y utilizar la constante en su lugar se conoce como *cálculo previo de constantes.*

Una ventaja de la propagación de copias es que a menudo convierte la proposición de copia en código inactivo. Por ejemplo, la propagación de copias seguida de la eliminación de código inactivo elimina la asignación a x y transforma (10.1) en:

```
a[t2] := t5
a[t4] := t3
goto B2
```

Este código es una mejora adicional del bloque B_5 de la figura 10.7.

Optimaciones de lazos

Ahora se da una breve introducción a un lugar muy importante para las optimaciones, que son los lazos, especialmente los lazos internos donde los programas tienden a emplear la mayor parte de su tiempo. El tiempo de ejecución de un programa se

puede mejorar si se disminuye la cantidad de instrucciones en un lazo interno, incluso si se incrementa la cantidad de código fuera del lazo. Hay tres técnicas importantes para la optimación de lazos: el *traslado de código*, que traslada código fuera del lazo; la *eliminación de variables de inducción*, que se aplica para eliminar `i` y `j` de los lazos internos B_2 y B_3 de la figura 10.7; y la *reducción de intensidad*, que sustituye una operación cara por una más barata, como una multiplicación por una suma.

Traslado de código

Una modificación importante que disminuye la cantidad de código en un lazo es el traslado de código. Esta transformación toma una expresión que produce el mismo resultado independientemente del número de veces que se ejecute un lazo (el *cálculo de una invariante del ciclo*) y coloca la expresión antes del lazo. Obsérvese que la noción "antes del lazo" supone la existencia de una entrada al lazo. Por ejemplo, la evaluación de `límite - 2` es un cálculo de una invariante de lazo en la siguiente proposición **while**:

```
while ( i <= límite-2 ) /* la proposición no cambia límite */
```

El traslado de código resultará equivalente a

```
t = límite-2;
while ( i <= t ) /* la proposición no cambia límite ni t */
```

Variables de inducción y reducción de intensidad

Aunque el traslado de código no es aplicable al ejemplo de la clasificación por particiones que se ha venido considerando, las otras dos transformaciones sí lo son. Los lazos se procesan generalmente de dentro hacia afuera. Por ejemplo, considérese el lazo alrededor de B_3. Sólo se muestra en la figura 10.9 la parte del grafo de flujo relevante para la transformación de B_3.

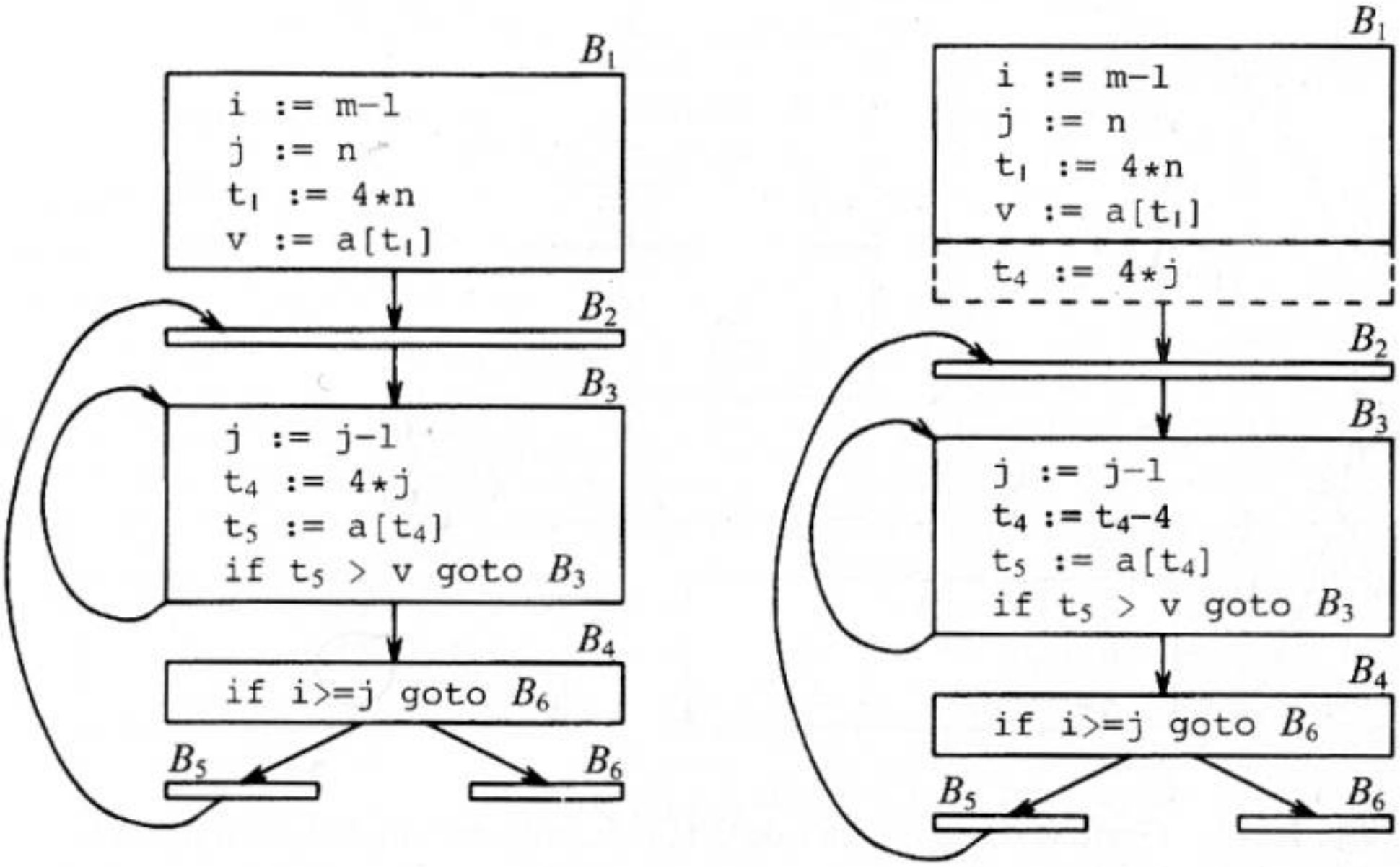
Obsérvese que los valores de `j` y t_4 permanecen atados; cada vez que el valor de `j` disminuye en 1, el de t_4 disminuye en 4 porque `4*j` se asigna a t_4. Dichos identificadores se denominan *variables de inducción.*

Cuando hay dos o más variables de inducción en un lazo, es posible suprimirlos todos menos uno, mediante el proceso de eliminación de variables de inducción. Para el lazo interno alrededor de B_3 en la figura 10.9(a), no se puede suprimir completamente `j` o t_4; t_4 se utiliza en B_3 y `j` en B_4. Sin embargo, se puede ilustrar la reducción de intensidad y una parte del proceso de eliminación de variables de inducción. Finalmente, `j` será eliminado cuando se considere el lazo externo de $B_2 - B_5$.

Ejemplo 10.3. Como la relación t_4 `= 4*j` se cumple siempre después de dicha asignación a t_4 en la figura 10.9(a) y t_4 no cambia en ninguna otra parte del lazo interno alrededor de B_3, se deduce que justo después de la proposición `j:=j-1` se debe cumplir la relación t_4 `= 4*j-4`. Por tanto, se puede sustituir la asignación t_4`:= 4*j` por t_4`:=`t_4`-4`. El único problema es que t_4 no tiene un valor cuando se entra al bloque B_3 por primera vez. Como se debe conservar la relación t_4 `= 4*j`

a la entrada al bloque B_3, se coloca una inicialización de t_4 al final del bloque en donde se inicializa j misma, mostrada por la suma punteada al bloque B_1 en la figura 10.9(b).

La sustitución de una multiplicación por una resta acelerará el código objeto si la multiplicación emplea más tiempo que la suma o la resta, como es el caso en muchas máquinas. □



(a) Antes

(b) Después

Fig. 10.9. Reducción de intensidad aplicada a 4*j en el bloque B_3.

La sección 10.7 analiza cómo detectar las variables de inducción y las transformaciones que se pueden aplicar. Esta sección concluye con un ejemplo más de eliminación de variables de inducción que considera i y j en el contexto del lazo exterior que contiene B_2, B_3, B_4 y B_5.

Ejemplo 10.4. Después de que la reducción de intensidad se aplica a los lazos internos alrededor de B_2 y B_3, el único uso de i y j es determinar el resultado de la prueba en el bloque B_4. Se sabe que los valores de i y t_2 cumplen la relación $t_2 = 4*i$, en tanto que los de j y t_4 satisfacen la relación $t_4 = 4*j$, así que la prueba $t_2>=t_4$ es equivalente a i>=j. Una vez hecha esta sustitución, i en el bloque B_2 y j en el bloque B_3 se convierten en variables inactivas y las asignaciones a ellas en estos bloques se convierten en código inactivo que puede ser eliminado, obteniéndose el grafo de flujo que se muestra en la figura 10.10. □

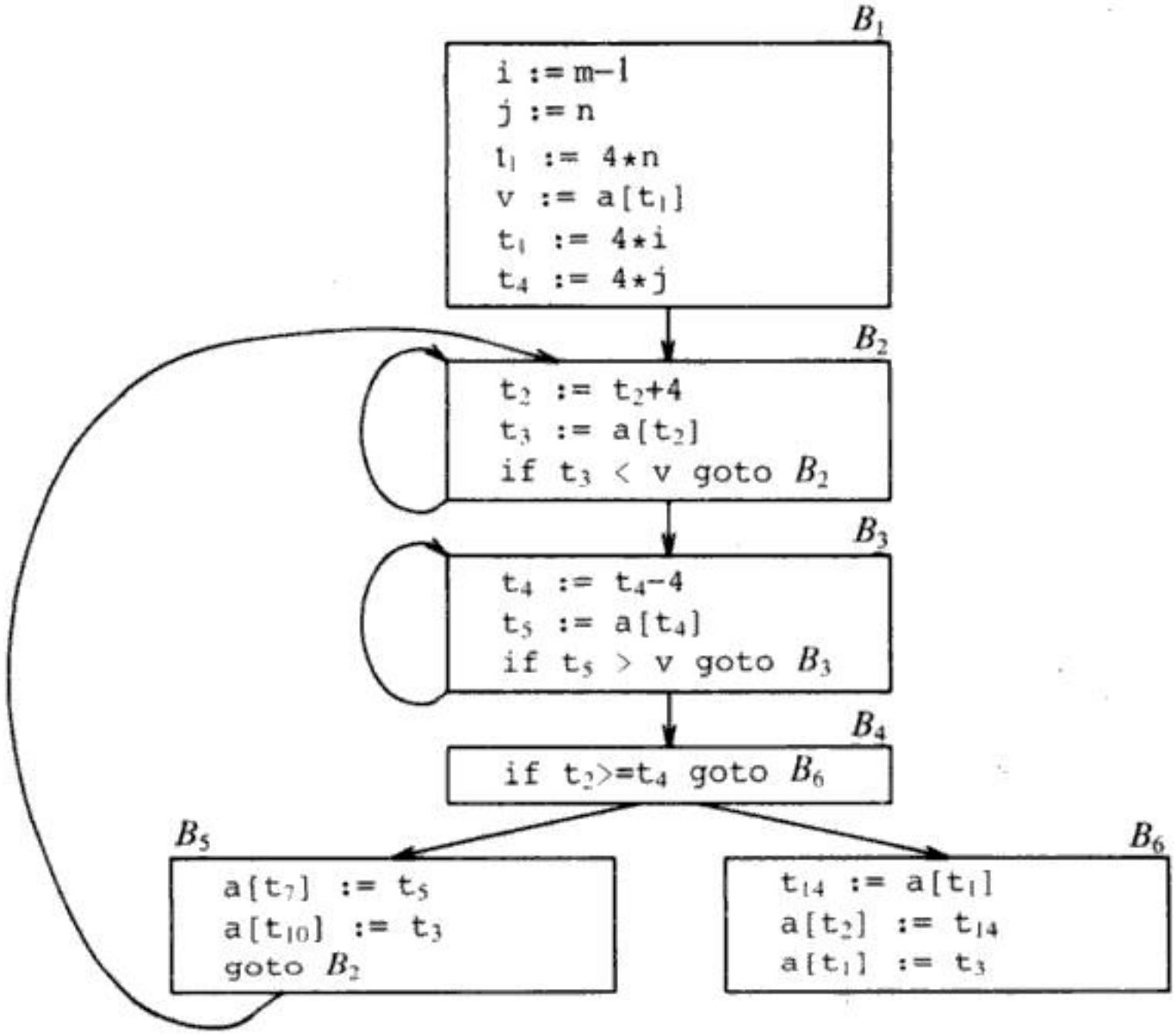


Fig. 10.10. Grafo de flujo después de la eliminación de variables de inducción.

Las transformaciones para mejorar el código han sido efectivas. En la figura 10.10, el número de instrucciones en los bloques B_2 y B_3 se ha reducido de 4 a 3 a partir del grafo de flujo original de la figura 10.5, en B_5 se ha reducido de 9 a 3, y en B_6 de 8 a 3. En realidad, B_1 ha aumentado de cuatro instrucciones a seis, pero B_1 se ejecuta sólo una vez en el fragmento, así que el tiempo de ejecución total está rara vez influido por el tamaño de B_1.

10.3 OPTIMACION DE BLOQUES BASICOS

En el capítulo 9 se vieron varias transformaciones para mejorar el código para los bloques básicos. Estas incluyen transformaciones que preservan la estructura, como la eliminación de subexpresiones comunes y la eliminación de código inactivo, y las transformaciones algebraicas como la reducción de intensidad.

Muchas de las transformaciones que preservan la estructura se pueden implantar mediante la construcción de un GDA para un bloque básico. Recuérdese que hay un nodo en el GDA para cada uno de los valores iniciales de las variables que aparecen en el bloque básico, y hay un nodo n asociado a cada proposición s dentro del bloque. Los hijos de n son los nodos correspondientes a proposiciones que son las últimas definiciones anteriores a s de los operandos utilizados por s. El nodo n se

etiqueta con el operador aplicado a s, y también se asocia a n la lista de variables para las cuales es la última definición dentro del bloque. También se tienen en cuenta los nodos, si los hay, cuyos valores están activos a la salida del bloque; estos son los nodos de salida.

Las subexpresiones comunes se pueden detectar observando, cuando un nuevo nodo m está a punto de añadirse, si existe un nodo n con los mismos hijos, en el mismo orden, y con el mismo operador. Si es así, n calcula el mismo valor que m y puede ser utilizado en su lugar.

Ejemplo 10.5. En la figura 10.11 se muestra un GDA para el bloque (10.3)

```
a := b + c
b := a - d                    (10.3)
c := b + c
d := a - d
```

Cuando se construye el nodo para la tercera proposición `c:=b+c`, se sabe que el uso de `b` en `b+c` se refiere al nodo de la figura 10.11 etiquetado con −, porque esa es la definición más reciente de `b`. Por tanto, no se confunden los valores calculados en las proposiciones una y tres.

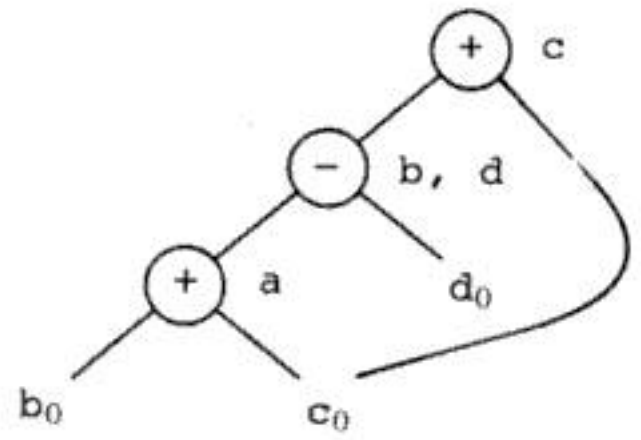


Fig. 10.11. GDA para el bloque básico (10.3).

Sin embargo, el nodo correspondiente a la cuarta proposición `d:=a-d` tiene el operador − y los nodos etiquetados con `a` y d_0 como hijos. Como el operador y los hijos son los mismos que para el nodo correspondiente a la proposición dos, no se crea este nodo sino que se añade `d` a la lista de definiciones para el nodo etiquetado con −. □

Puede suceder que, como sólo hay tres nodos en el GDA de la figura 10.11, el bloque (10.3) se puede sustituir por un bloque con sólo tres proposiciones. De hecho, si `b` o `d` no está activa a la salida del bloque, entonces no hay que calcular esa variable, y se puede utilizar la otra para recibir el valor representado por el nodo etiquetado con − en la figura 10.11. Por ejemplo, si `b` no está activa a la salida, se podría utilizar:

```
a := b + c
d := a - d
c := d + c
```

Sin embargo, si tanto b como d están activas a la salida, entonces se debe utilizar una cuarta proposición para copiar el valor de una a la otra[2].

Obsérvese que cuando se buscan subexpresiones comunes, en realidad se están buscando expresiones que vayan a calcular el mismo valor, independientemente de cómo se calcule dicho valor. Por tanto, el método del GDA no tendrá en cuenta el hecho de que la expresión calculada por la primera y la cuarta proposiciones en la secuencia

```
a := b + c
b := b - d
c := c + d                    (10.4)
e := b + c
```

es la misma, es decir, b+c. Sin embargo, las identidades algebraicas aplicadas al GDA, como se estudia a continuación, pueden exponer la equivalencia. En la figura 10.12 se muestra el GDA para esta secuencia.

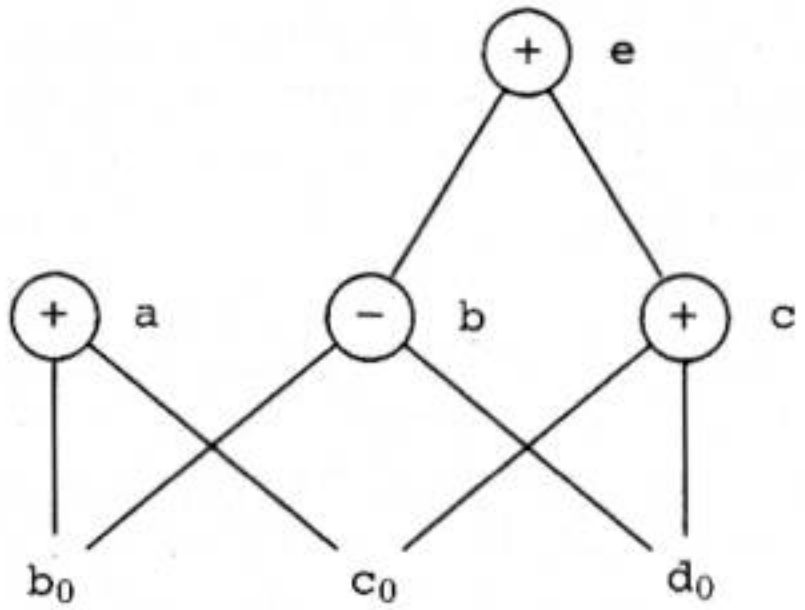


Fig. 10.12. GDA para el bloque básico (10.4).

La operación sobre GDA que corresponde a la eliminación del código inactivo es bastante directa de implantar. Se borra de un GDA cualquier raíz (nodo sin ancestros) que no tenga variables activas. La aplicación repetida de esta transformación eliminará todos los nodos del GDA que correspondan a código inactivo.

El uso de identidades algebraicas

Las identidades algebraicas representan otra clase importante de optimaciones sobre bloques básicos. En la sección 9.9 se introdujeron algunas sencillas transformaciones algebraicas que se pueden intentar durante la optimación. Por ejemplo, se pueden aplicar identidades aritméticas, como

---

[2] En general, hay que tener cuidado cuando se reconstruye código a partir de los GDA para elegir los nombres de las variables que corresponden a nodos. Si una variable x se define dos veces, o se le hace una asignación una vez y también se utiliza el valor inicial x_0, entonces hay que asegurarse de que no se modifique el valor de x hasta que se hayan hecho todos los usos del nodo cuyo valor contenía previamente x.

```
x + 0 = 0 + x = x
x - 0 = x
x * 1 = 1 * x = x
x / 1 = x
```

Otra clase de optimaciones algebraicas incluyen la reducción de intensidad, es decir, sustituir un operador más caro por otro más barato como en

```
x ** 2 = x * x
2.0 * x = x + x
x / 2 = x * 0.5
```

Una tercera clase de optimaciones afines es el cálculo previo de constantes. Aquí se evalúan las expresiones constantes durante la compilación y se sustituyen las expresiones constantes por sus valores[3]. Por tanto, la expresión `2*3.14` se sustuiría por `6.28`. Muchas expresiones constantes surgen con el uso de constantes simbólicas.

El proceso de construcción de un GDA puede ayudar a aplicar estas y otras transformaciones algebraicas más generales, como la conmutatividad y la asociatividad. Por ejemplo, supóngase que * es conmutativo; es decir, `x*y = y*x`. Antes de crear un nuevo nodo etiquetado con * con hijo izquierdo *m* e hijo derecho *n*, se revisa si ya existe dicho nodo. Después se busca un nodo que tenga operador *, hijo izquierdo *n* e hijo derecho *m*.

Los operadores relacionales <=, >=, <, >, = y ≠ a veces generan subexpresiones comunes imprevistas. Por ejemplo, la condición `x>y` también se puede comprobar restando los argumentos y realizando una prueba del código de condición establecido por la resta. (Sin embargo, la resta puede introducir desbordamientos y desbordamientos negativos, mientras que una instrucción de comparación no lo haría.) Por tanto, sólo hay que generar un nodo del GDA para `x-y` y `x>y`.

Las leyes asociativas también se pueden aplicar para evidenciar las subexpresiones comunes. Por ejemplo, si el código fuente tiene las asignaciones

```
a := b + c
e := c + d + b
```

se podría generar el siguiente código intermedio:

```
a := b + c
t := c + d
e := t + b
```

Si no se necesita `t` fuera de este bloque, se puede cambiar esta secuencia por

```
a := b + c
e := a + d
```

utilizando la asociatividad y la conmutatividad de +.

---

[3] Las expresiones aritméticas se deben evaluar del mismo modo durante la compilación que durante la ejecución. K. Thompson ha propuesto una solución elegante al cálculo previo de constantes: compilar la expresión constante, ejecutar el código objeto allí mismo y sustituir la expresión con el resultado. Por tanto, el compilador no necesita contener un intérprete.

El escritor del compilador debe examinar cuidadosamente la especificación del lenguaje para determinar los ordenamientos de cálculos que están permitidos, puesto que la aritmética del computador no siempre obedece las identidades algebraicas de la matemática. Por ejemplo, el estándar para FORTRAN 77 establece que un compilador puede evaluar cualquier expresión matemáticamente equivalente si no se viola la integridad de los paréntesis. Por tanto, un compilador puede evaluar `x*y-x*z` como `x*(y-z)` pero no puede evaluar `a+(b-c)` como `(a+b)-c`. Un compilador de FORTRAN debe por tanto llevar un registro de dónde estaban los paréntesis en las expresiones del lenguaje fuente, si va a optimar programas según la definición del lenguaje.

10.4 LAZOS EN LOS GRAFOS DE FLUJO

Antes de considerar optimaciones de lazos, hay que definir lo que constituye un lazo en un grafo de flujo. Se utilizará la noción de que un nodo "domina" a otro para definir los "lazos naturales" y la importante clase especial de grafos de flujo "reducibles". En la sección 10.9 se dará un algoritmo para encontrar dominantes y comprobar la reducibilidad de los grafos de flujo.

Dominantes

Se dice que el nodo d de un grafo de flujo *domina* al nodo n, que se escribe d *dom* n, si todo camino desde el nodo inicial del grafo de flujo a n pasa por d. Con esta definición, todo nodo se domina a sí mismo, y la entrada de un lazo (como se definió en la Sec. 9.4) domina todos los nodos del lazo.

Ejemplo 10.6. Considérese el grafo de flujo de la figura 10.13, con nodo inicial 1. El nodo inicial domina a todos los nodos. El nodo 2 sólo se domina a sí mismo,

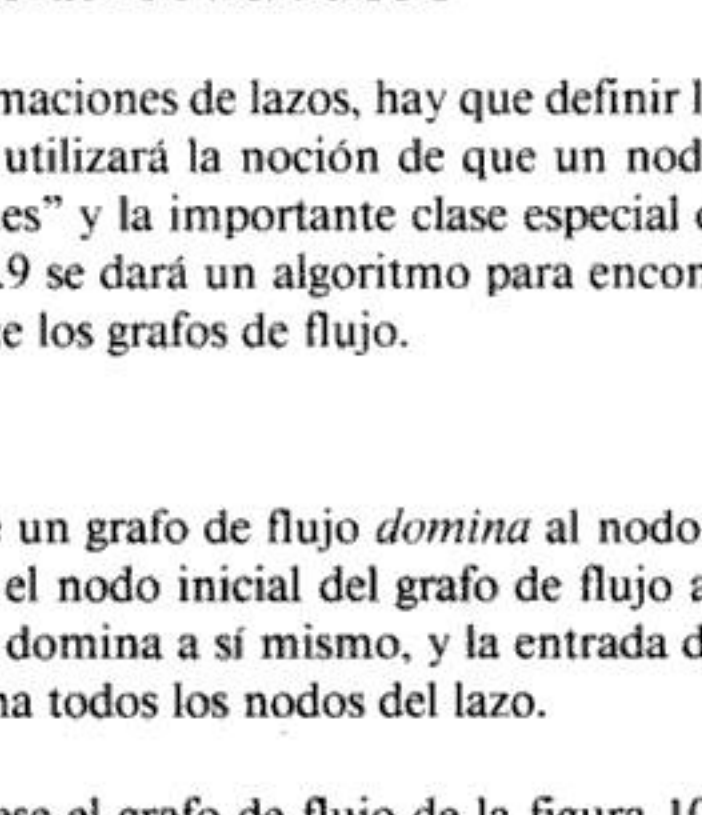


Fig. 10.13. Grafo de flujo.

puesto que el control puede llegar a cualquier otro nodo a través de un camino que comience con $1 \rightarrow 3$. El nodo 3 domina a todos excepto a 1 y a 2. El nodo 4 domina a todos excepto a 1, 2 y 3, porque todos los caminos a partir de 1 deben comenzar con $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$ o con $1 \rightarrow 3 \rightarrow 4$. Los nodos 5 y 6 sólo se dominan a ellos mismos, porque el flujo del control puede saltar a cualquiera de ellos a través del otro. Por último, 7 domina a 7, 8, 9, 10; 8 domina a 8, 9, 10; 9 y 10 sólo se dominan a ellos mismos. □

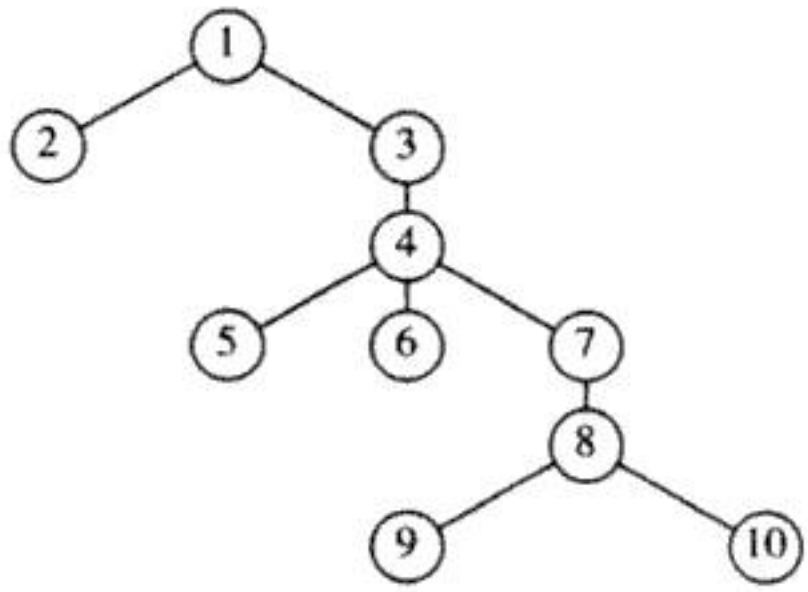


Fig. 10.14. Arbol de dominación para el grafo de flujo de la figura 10.13.

Una forma útil de presentar la información sobre dominadores es en un árbol, llamado *árbol de dominación*, en el que el nodo inicial es la raíz, y cada nodo d domina sólo a sus descendientes en el árbol. Por ejemplo, en la figura 10.14 se muestra el árbol de dominación para el grafo de flujo de la figura 10.13.

La existencia de árboles de dominación se deriva de una propiedad de los dominantes; cada nodo n tiene un *dominador inmediato* m que es el último dominador de n en cualquier camino desde el nodo inicial a n. Respecto a la relación *dom*, el dominador inmediato m tiene la propiedad de que si $d \neq n$ y d *dom* n, entonces d *dom* m.

Lazos naturales

Una aplicación importante de la información sobre dominadores es determinar los lazos de un grafo de flujo objeto de mejora. Estos lazos tienen dos propiedades importantes:

1. Un lazo debe tener un solo punto de entrada, llamado "encabezamiento". Este punto de entrada domina todos los nodos dentro del lazo, o no sería la única entrada al lazo.
2. Debe haber al menos una forma de iterar el lazo, es decir, al menos un camino de regreso al encabezamiento.

Una buena forma de encontrar todos los lazos en un grafo de flujo es buscar aristas en el grafo de flujo cuyas cabezas dominen sus colas. (Si $a \rightarrow b$ es una arista, b es la *cabeza* y a es la *cola*.) Dichas aristas se denominan *aristas de retroceso.*

Ejemplo 10.7. En la figura 10.13 hay una arista $7 \rightarrow 4$, y 4 *dom* 7. De manera similar, $10 \rightarrow 7$ es una arista, y 7 *dom* 10. Las otras aristas con esta propiedad son $4 \rightarrow 3$, $8 \rightarrow 3$ y $9 \rightarrow 1$. Obsérvese que éstas son exactamente las aristas que intuitivamente parecen formar lazos en el grafo de flujo. □

Dada una arista de retroceso $n \rightarrow d$, se define el *lazo natural* de la arista como d más el conjunto de nodos que puede alcanzar n sin ir a través de d. El nodo d es el encabezamiento del lazo.

Ejemplo 10.8. El lazo natural de la arista $10 \rightarrow 7$ consta de los nodos 7, 8 y 10, puesto que 8 y 10 son todos los nodos que pueden alcanzar 10 sin pasar por 7. El lazo natural de $9 \rightarrow 1$ es el grafo de flujo completo. (No hay que olvidar el camino $10 \rightarrow 7 \rightarrow 8 \rightarrow 9$.) □

Algoritmo 10.1. Construcción del lazo natural de una arista de retroceso.

Entrada. Un grafo de flujo G y una arista de retroceso $n \rightarrow d$.

Salida. El conjunto *lazo* que consta de todos los nodos dentro del lazo natural de $n \rightarrow d$.

Método. Comenzando con el nodo n, se considera cada nodo $m \neq d$ que se sabe que está en *lazo* para asegurarse de que los predecesores de m también se colocan en *lazo*. El algoritmo se da en la figura 10.15. Cada nodo en *lazo*, excepto d, se coloca una vez en *pila*, así que sus predecesores serán examinados. Obsérvese que como d se coloca en el lazo al inicio, nunca se examinan sus precedesores, y por tanto se encuentran sólo aquellos nodos que alcanzan n sin pasar por d. □

Lazos internos

Si se utilizan los lazos naturales como "los lazos", entonces existe la propiedad útil de que, a menos que dos lazos tengan el mismo encabezamiento, son disjuntos o

```
procedure inserta (m);
if m no está en lazo then begin
    lazo := lazo ∪ {m};
    mete m en pila
end;

/* sigue el programa principal */

pila := vacía;
lazo := {d};
inserta (n);
while pila no sea vacía do begin
    saca m, el primer elemento de pila, de pila;
    for cada predecesor b de m do inserta (p)
end
```

Fig. 10.15. Algoritmo para construir el lazo natural.

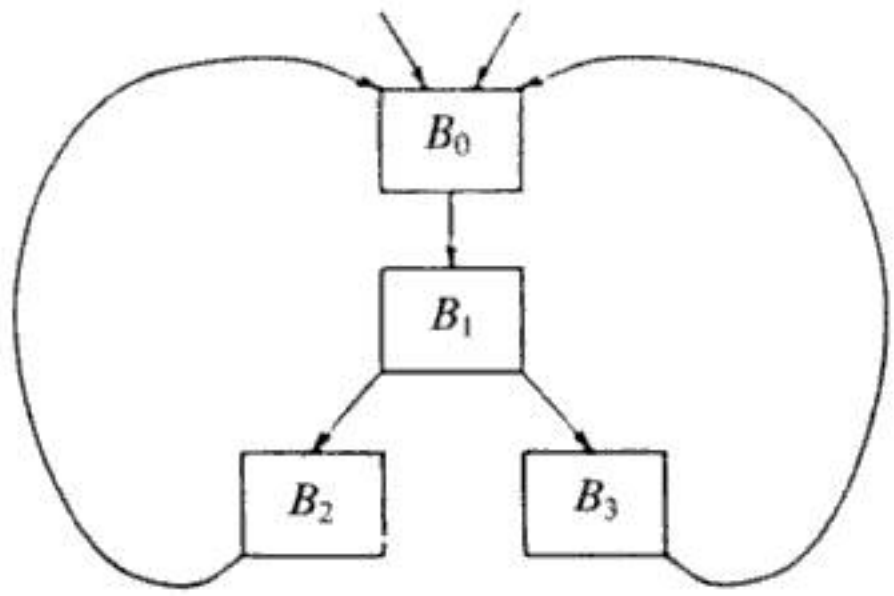


Fig. 10.16. Dos lazos con el mismo encabezamiento.

uno está completamente contenido (*anidado dentro*) en el otro. Por tanto, sin tener en cuenta por el momento los lazos con el mismo encabezamiento, se tiene una noción natural de *lazo interno*: uno que no contiene otros lazos.

Cuando dos lazos tienen el mismo encabezamiento, como en la figura 10.16, es difícil saber cuál es el lazo interno. Por ejemplo, si la condicional al final de B_1 fuera

```
if a = 10 goto B2
```

probablemente el lazo $\{B_0, B_1, B_3\}$ sería el lazo interno. Sin embargo, no es posible estar seguro sin un examen detallado del código. Quizás `a` casi siempre es `10`, y es habitual iterar el lazo $\{B_0, B_1, B_2\}$ muchas veces antes de ir a B_3. Por tanto, se supondrá que cuando dos lazos naturales tienen el mismo encabezamiento pero ninguno está anidado dentro del otro, se combinan y consideran como un solo lazo.

Preencabezamientos

Varias transformaciones exigen trasladar las proposiciones "antes del encabezamiento". Por tanto, se comienza el tratamiento de un lazo L creando un nuevo bloque, llamado *preencabezamiento*. El preencabezamiento sólo tiene el encabezamiento como sucesor y todas las aristas que antes entraban al encabezamiento de L desde fuera de L ahora entran al preencabezamiento. No se modifican las aristas desde dentro del lazo L al encabezamiento. El ordenamiento se muestra en la figura 10.17. Al inicio, el preencabezamiento está vacío pero las transformaciones sobre L pueden colocar proposiciones en él.

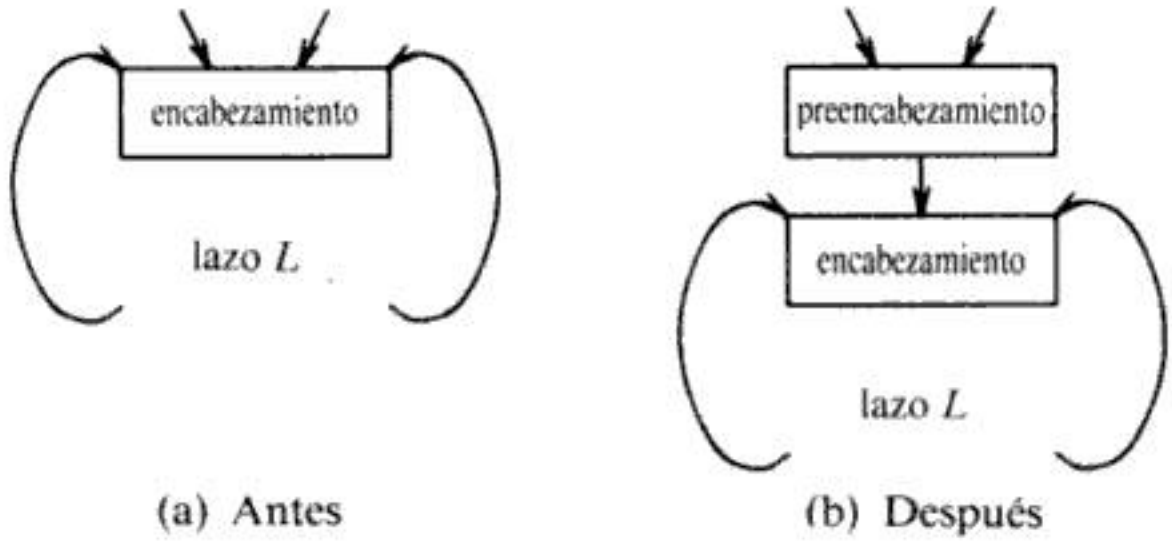


Fig. 10.17. Introducción del preencabezamiento.

Grafos de flujo reducibles

Los grafos de flujo que ocurren en la práctica, a menudo forman parte de la clase de grafos de flujo reducibles que se define más adelante. El uso exclusivo de proposiciones de flujo de control estructuradas como las proposiciones **if-then-else**, **while-do**, **continue** y **break** produce programas cuyos grafos de flujo siempre son reducibles. Incluso los programas escritos utilizando proposiciones **goto** por programadores sin conocimiento previo de diseño de programas estructurados son reducibles casi siempre.

Se han propuesto muchas definiciones de "grafo de flujo reducible". La que aquí se adopta tiene una de las propiedades más importantes de los grafos de flujo reducibles: que no hay saltos hacia dentro de los lazos desde el exterior; la única entrada a un lazo es a través de su encabezamiento. Los ejercicios y las notas bibliográficas contienen una breve historia del concepto.

Un grafo de flujo G es *reducible* si, y sólo si, se pueden particionar las aristas en dos grupos disjuntos, llamadas con frecuencia aristas *de avance* y aristas *de retroceso*, con las siguientes dos propiedades:

1. Las aristas de avance forman un grafo acíclico en el que cada nodo se puede alcanzar desde el nodo inicial de G.
2. Las aristas de retroceso constan sólo de las aristas cuyas cabezas dominan sus colas.

Ejemplo 10.9. El grafo de flujo de la figura 10.13 es reducible. En general, si se conoce la relación *dom* para un grafo de flujo, se pueden encontrar y eliminar todas las aristas de retroceso. Las aristas restantes deben ser las aristas de avance si el grafo es reducible, y para comprobar si un grafo de flujo es reducible, basta con comprobar si las aristas de avance forman un grafo acíclico. En el caso de la figura 10.13, es fácil comprobar que si se eliminan las cinco aristas de retroceso $4 \rightarrow 7$, $7 \rightarrow 4$, $8 \rightarrow 3$, $9 \rightarrow 1$ y $10 \rightarrow 7$, cuyas cabezas dominan a sus colas, el grafo restante es acíclico. □

Ejemplo 10.10. Considérese el grafo de flujo de la figura 10.18, cuyo nodo inicial es 1. Este grafo de flujo no tiene aristas de retroceso, puesto que ninguna cabeza de una

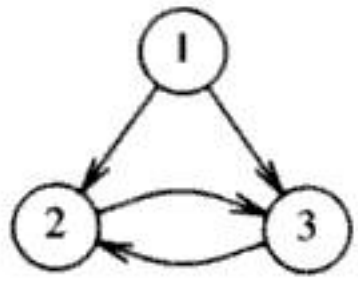


Fig. 10.18. Un grafo de flujo no reducible.

arista domina a la cola de dicha arista. Por tanto, sólo podría ser reducible si el grafo completo fuera acíclico. Pero como no lo es, el grafo de flujo no es reducible. De manera intuitiva, la razón de que este grafo de flujo no sea reducible es que se puede entrar al lazo 2-3 por dos sitios diferentes, los nodos 2 y 3.

La propiedad clave de los grafos de flujo reducibles para el análisis de lazos es que en dichos grafos de flujo cada conjunto de nodos que informalmente podría ser considerado como un lazo debe contener una arista de retroceso. De hecho, basta con examinar los lazos naturales de aristas de retroceso para encontrar todos los lazos en un programa cuyo grafo de flujo sea reducible. Por el contrario, el grafo de flujo de la figura 10.18 tiene un "lazo" que consta de los nodos 2 y 3, pero no hay ninguna arista de retroceso de la cual sea éste su lazo natural. De hecho, ese "lazo" tiene dos encabezamientos, 2 y 3, lo que hace que la aplicación de muchas técnicas de optimación de código, como las presentadas en la sección 10.2 para traslado de código y eliminación de variables de inducción, no sean aplicables directamente.

Por fortuna, las estructuras de flujo de control no reducibles, como la de la figura 10.18, aparecen tan raramente en la mayoría de los programas que convierten el estudio de lazos con más de un encabezamiento en poco importante. Incluso hay lenguajes, como BLISS y MODULA 2, que permiten sólo programas con grafos de flujo reducibles, y muchos otros lenguajes producirán sólo grafos de flujo reducibles mientras no se utilicen proposiciones **goto.**

Ejemplo 10.11. Volviendo a la figura 10.13, se observa que el único "lazo interno", es decir, un lazo sin sublazos, es {7, 8, 10}, el lazo natural con arista de retroceso $10 \rightarrow 7$. El conjunto {4, 5, 6, 7, 8, 10} es el lazo natural de $7 \rightarrow 4$. (Obsérvese que 8 y 10 pueden alcanzar 7 a través de la arista $10 \rightarrow 7$.) La intuición de que {4, 5, 6, 7} forma un lazo es equivocada, ya que 4 y 7 serían entradas desde el exterior, incumpliendo el requisito de una sola entrada. Dicho de otro modo, no hay razón para suponer que el control emplea mucho tiempo en ir alrededor del conjunto de nodos {4, 5, 6, 7}; es tan plausible que el control pase a 8 desde 7 más veces que a 4. Incluyendo 8 y 10 en el lazo, se tiene mayor certeza de haber aislado una región muy transitada del programa.

Sin embargo, hay que reconocer el peligro de hacer suposiciones sobre la frecuencia de las ramificaciones. Por ejemplo, si se saca una proposición invariante de 8 ó 10 en el lazo {7, 8, 10} y de hecho, el control siguiera la arista $7 \rightarrow 4$ en más ocasiones que $7 \rightarrow 8$, se incrementaría realmente el número de veces que se ejecutara la proposición trasladada. En la sección 10.7 se estudiarán métodos para evitar este problema.

Después, el lazo más grande es {3, 4, 5, 6, 7, 8, 10}, que es el lazo natural de ambas aristas $4 \rightarrow 3$ y $8 \rightarrow 3$. Como antes, la intuición de que {3, 4} debería considerarse como un lazo incumple el requisito de un solo encabezamiento. El último lazo, el de la arista de retroceso $9 \rightarrow 1$, es el grafo de flujo completo. □

Hay varias propiedades útiles adicionales de los grafos de flujo reducibles, que se presentarán cuando se estudien los temas de búsqueda en profundidad y análisis de intervalos en la sección 10.9.

10.5 INTRODUCCION AL ANALISIS GLOBAL DEL FLUJO DE DATOS

Para realizar la optimación de código y un buen trabajo de generación de código, un compilador necesita reunir información sobre el programa como un todo y distribuir esta información a cada bloque en el grafo de flujo. Por ejemplo, en la sección 9.7 se vio que el conocimiento de las variables que están activas a la salida de cada bloque puede optimar el uso de los registros. En la sección 10.2 se sugirió cómo utilizar el conocimiento de las subexpresiones comunes para eliminar cálculos redundantes. Asimismo, en las secciones 9.9 y 10.3 se estudió cómo un compilador puede aprovechar las "definiciones de alcance", como saber dónde se definió por última vez una variable como `depura` antes de llegar a un bloque dado, para poder realizar transformaciones como el cálculo previo de constantes y la eliminación de código inactivo. Estos hechos son sólo unos cuantos ejemplos de la *información de flujo de datos* que un compilador optimador recopila mediante un proceso llamado *análisis de flujo de datos*.

La información del flujo de datos se puede recopilar estableciendo y resolviendo sistemas de ecuaciones que relacionan la información en varios puntos de un programa. Una ecuación típica tiene la forma

$$sal\,[S] = gen\,[S] \cup (ent\,[S] - desact\,[S]) \tag{10.5}$$

y se puede leer como "la información al final de una proposición se genera dentro de la proposición o se introduce al comienzo y no se desactiva cuando el control fluye por la proposición". Dichas ecuaciones se denominan *ecuaciones de flujo de datos*.

Los detalles sobre cómo se plantean y resuelven las ecuaciones de flujo de datos dependen de tres factores:

1. Las nociones de generar y desactivar dependen de la información deseada, es decir, del problema de análisis del flujo de datos que debe resolverse. Además, para algunos problemas, en lugar de avanzar junto con el flujo del control y definir *sal* [*S*] según *ent* [*S*], hay que hacerlo hacia atrás y definir *ent* [*S*] según *sal* [*S*].

2. Como los datos fluyen a lo largo de caminos de control, el análisis del flujo de datos está influido por las construcciones de control en un programa. De hecho, cuando se escribe *sal*[*S*], se asume implícitamente que hay un punto final único donde el control sale de la proposición; en general, las ecuaciones se establecen al nivel de bloques básicos en lugar de proposiciones, porque los bloques tienen puntos finales únicos.

3. Algunas sutilezas acompañan dichas proposiciones como las llamadas a procedimientos, las asignaciones por medio de variables tipo apuntador e incluso las asignaciones a variables de tipo matriz.

En esta sección se considera el problema de determinar el conjunto de definiciones que alcanzan un punto en un programa y su uso para encontrar oportunidades de hacer cálculo previo de constantes. Más adelante en este capítulo, los algoritmos

para traslado de código y la eliminación de variables de inducción utilizarán también esta información.

Inicialmente se consideran programas construidos utilizando proposiciones **if** y **do-while**. El flujo de control previsible en estas proposiciones permite concentrarse en las ideas necesarias para establecer y resolver las ecuaciones de flujo de datos. Las asignaciones en esta sección son proposiciones de copia o de la forma a:=b+c. En este capítulo con frecuencia se utiliza "+" como operador típico. Todo lo que se indique sirve directamente para otros operadores, incluidos aquellos con un operando o con más de dos operandos.

Puntos y caminos

Dentro de un bloque básico, se habla del *punto* entre dos proposiciones adyacentes, al igual que del punto antes de la primera proposición y después de la última. Por tanto, el bloque B_1 de la figura 10.19 tiene cuatro puntos: uno antes de todas las asignaciones y uno después de cada una de las tres asignaciones.

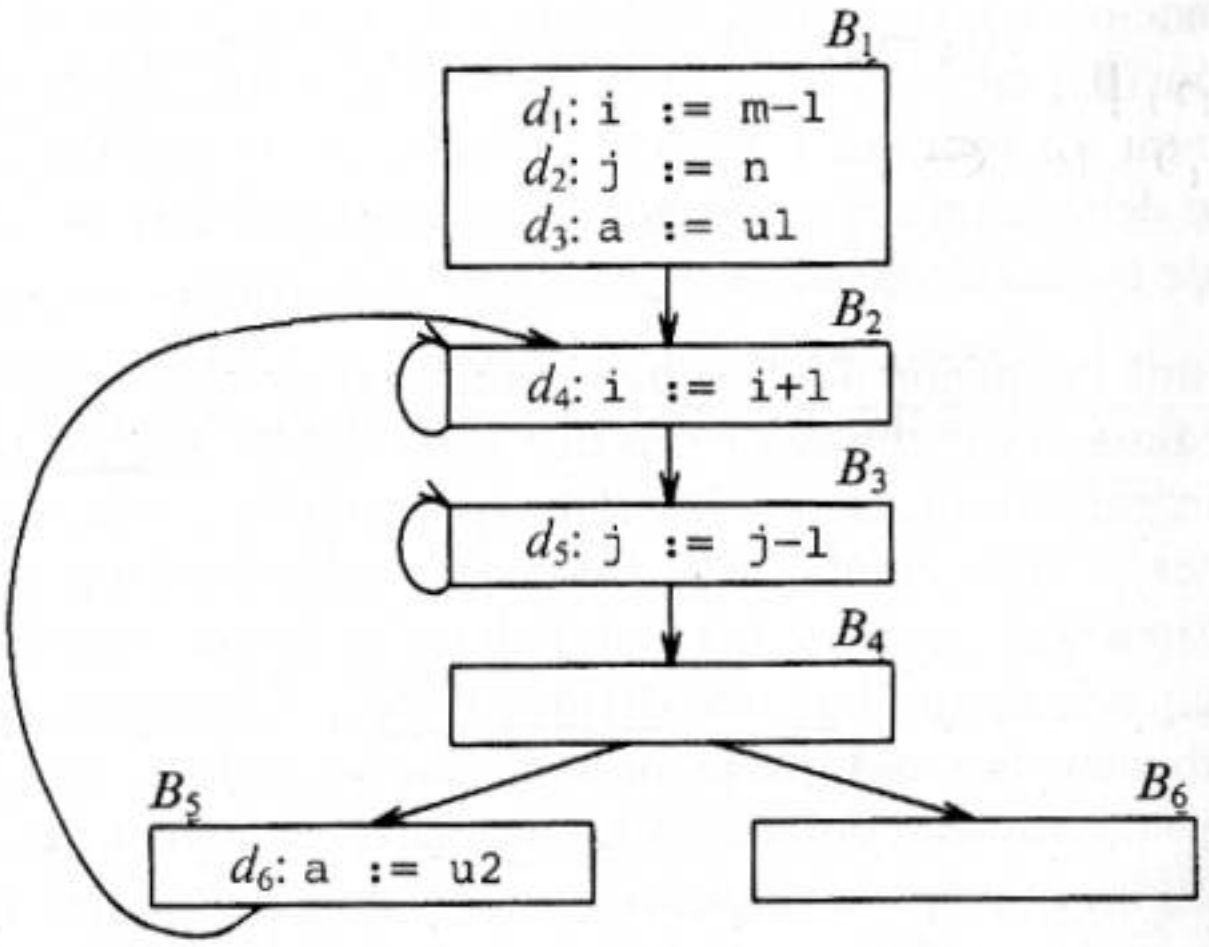


Fig. 10.19. Un grafo de flujo.

A continuación se considera una visión global y todos los puntos en la totalidad de los bloques. Un *camino* desde p_1 a p_n es una secuencia de puntos $p_1, p_2, \ldots, p_n$ tal que para cada i entre 1 y $n - 1$,

1. p_i es el punto que precede inmediatamente a una proposición y p_{i+1} es el punto que sigue inmediatamente a dicha proposición en el mismo bloque, o
2. p_i es el final de un bloque y p_{i+1} es el comienzo de un bloque sucesor.

Ejemplo 10.12. En la figura 10.19 hay un camino desde el comienzo del bloque B_5 hasta el comienzo del bloque B_6. Pasa por el punto final de B_5 y después por todos los puntos en B_2, B_3 y B_4, en orden, antes de alcanzar el comienzo de B_6. □

Definiciones de alcance

Una *definición* de una variable x es una proposición que asigna, o puede asignar, un valor a x. Las formas más comunes de definición son las asignaciones a x y las proposiciones que leen un valor de un dispositivo de E/S y lo almacenan en x. Estas proposiciones ciertamente definen un valor para x, y se consideran definiciones *no ambiguas* de x. Otras clases de proposiciones pueden definir un valor para x; se denominan definiciones *ambiguas*. Las formas más habituales de definiciones ambiguas de x son:

1. Una llamada a un procedimiento con x como parámetro (que no sea un parámetro por valor) o un procedimiento que puede acceder a x porque x está dentro del ámbito del procedimiento. También hay que considerar la posibilidad de los "sinónimos", donde x no está dentro del ámbito del procedimiento, pero x se ha identificado con otra variable que se pasa como parámetro o que está dentro del ámbito. Estos aspectos se estudian en la sección 10.8.

2. Una asignación por medio de un apuntador que pudiera referirse a x. Por ejemplo, la asignación `*q:=y` es una definición de x si es posible que q apunte a x. En la sección 10.8 también se estudian los métodos para determinar hacia dónde podría apuntar un apuntador, pero en ausencia de un conocimiento en sentido contrario, se debe asumir que una asignación por medio de un apuntador es una definición de todas las variables.

Se dice que una definición *d alcanza* un punto *p* si hay un camino desde el punto que sigue inmediatamente *d* hasta *p*, tal que *d* no se "desactive" a lo largo del camino. Intuitivamente, si una definición *d* de una variable a alcanza el punto *p*, entonces *d* debe ser el lugar en el que el valor de a utilizado en *p* puede haber sido definido por última vez. Se *desactiva* una definición de una variable a si entre dos puntos a lo largo del camino hay una definición de a. Obsérvese que sólo las definiciones no ambiguas de a desactivan otras definiciones de a. Por tanto, un punto puede ser alcanzado por una definición no ambigua y por una definición ambigua de la misma variable que aparezca posteriormente a lo largo de un camino.

Por ejemplo, ambas definiciones, `i:=m-1` y `j:=n`, en el bloque B_1 de la figura 10.19 alcanzan el principio del bloque B_2, al igual que la definición `j:=j-1` suponiendo que no haya asignaciones o lecturas de `j` en B_4, B_5, o en la parte de B_3 que sigue a esta definición. Sin embargo, la asignación `j` en B_3 desactiva la definición `j:=n`, así que esta última no alcanza B_4, B_5 o B_6.

Si se formulan así las definiciones de alcance, a veces surgen imprecisiones. Sin embargo, todas van en dirección "segura" o "conservadora". Por ejemplo, obsérvese nuestra suposición de que se pueden recorrer todas las aristas de un grafo de flujo. Esto puede no ser cierto en la práctica. Por ejemplo, el control puede realmente alcanzar la asignación `a:=4` en el siguiente fragmento de programa para ningún valor de a y de b.

```
if a = b then a := 2
else if a = b then a := 4
```

Decidir en general si se puede tomar cada camino en un grafo de flujo es un problema indecidible, y no se intentará resolverlo.

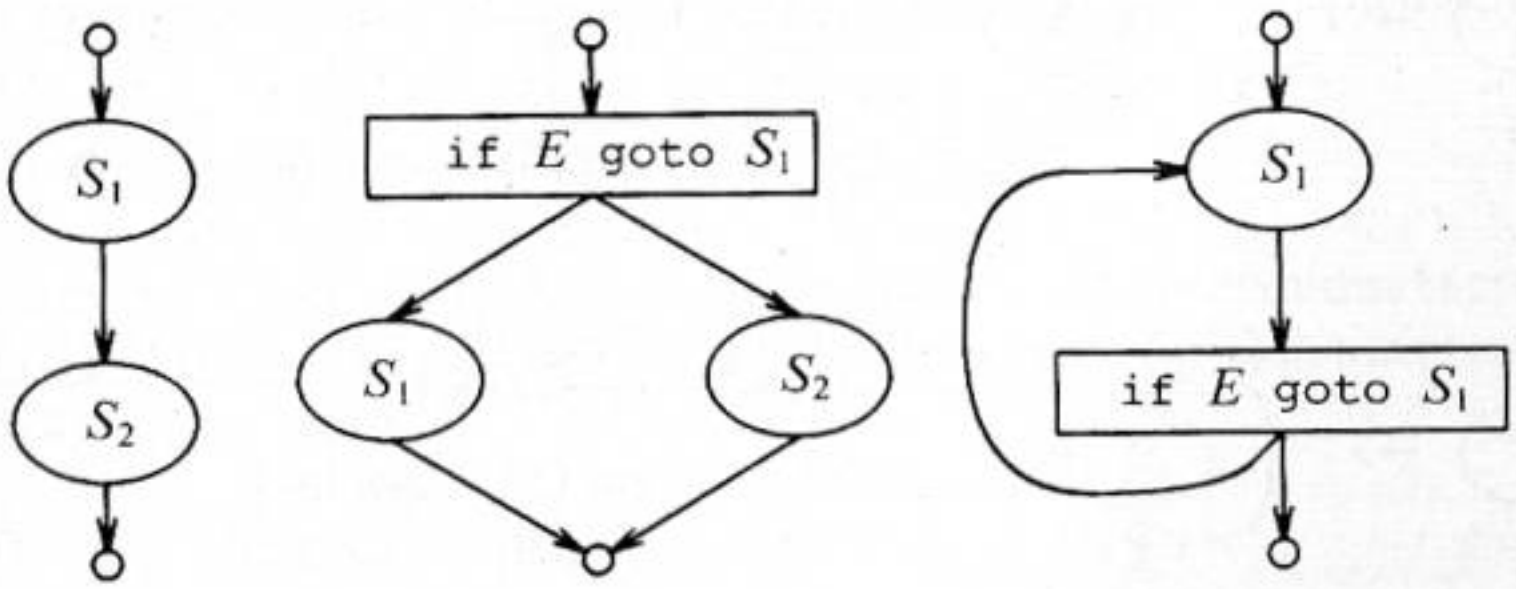
Un tema recurrente en el diseño de las transformaciones para mejorar el código es que, ante la duda, sólo se deben tomar decisiones conservadoras, aunque las estrategias conservadoras pueden impedir la realización de transformaciones que no suponen ningún peligro. Una decisión es *conservadora* si nunca conlleva un cambio en lo que calcula el programa. En aplicaciones de definiciones de alcance, es conservador asumir que una definición puede alcanzar un punto, aunque no lo haga. Por tanto, se permiten caminos que pueden no ser nunca recorridos durante la ejecución de un programa y se permite que las definiciones atraviesen definiciones ambiguas de la misma variable.

Análisis de flujo de datos de programas estructurados

Los grafos de flujo para las construcciones de flujo del control como las proposiciones **do-while** tienen una propiedad útil: hay un solo punto de comienzo en el que el control entra y un solo punto final por el que sale el control cuando termina la ejecución de la proposición. Esta propiedad se explota cuando se habla de que las definiciones alcanzan el comienzo y el final de proposiciones con la siguiente sintaxis:

$S \rightarrow$ **id** $:= E \mid S \; ; \; S \mid$ **if** E **then** S **else** $S \mid$ **do** S **while** E
$E \rightarrow$ **id** + **id** $\mid$ **id**

Las expresiones de este lenguaje son similares a las del código intermedio, pero los grafos de flujo para las proposiciones tienen formas limitadas sugeridas por los diagramas de la figura 10.20. Un propósito principal de esta sección es estudiar las ecuaciones de flujo de datos resumidas en la figura 10.21.



$S_1 \; ; \; S_2$ **if** E **then** S_1 **else** S_2 **do** S_1 **while** E

Fig. 10.20. Algunas construcciones de control estructuradas.

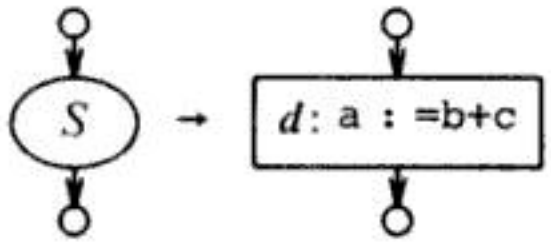
Se define una parte de un grafo de flujo denominada *región* como un conjunto de nodos N que incluye un *encabezamiento*, que domina a todos los otros nodos de la región. Todas las aristas entre los nodos de N están en la región, excepto (tal vez) alguno que entra al encabezamiento[4]. La parte de un grafo de flujo correspondiente a una proposición S es una región que cumple la limitación adicional de que el control puede fluir sólo a un bloque exterior cuando abandona la región.

Por conveniencia técnica se supone que hay bloques falsos sin proposiciones (indicados por círculos abiertos en la Fig. 10.20) a través de los cuales el control fluye

justo antes de entrar y justo antes de abandonar la región. Se dice que los puntos de comienzo de los bloques ficticios a la entrada y la salida de la región de una proposición son los puntos *comienzo* y *fin*, respectivamente, de la proposición.

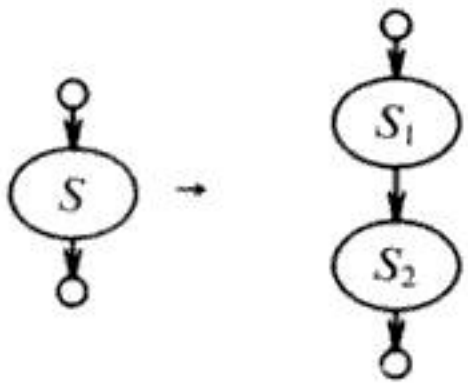
Las ecuaciones de la figura 10.21 son una definición inductiva, o dirigida por la sintaxis, de los conjuntos *ent* [*S*], *sal* [*S*] y *desact* [*S*] para todas las proposiciones *S*. Los conjuntos *gen* [*S*] y *desact* [*S*] son atributos sintetizados; se calculan en forma ascendente, desde las proposiciones más pequeñas a las más grandes. Se desea que

(a)



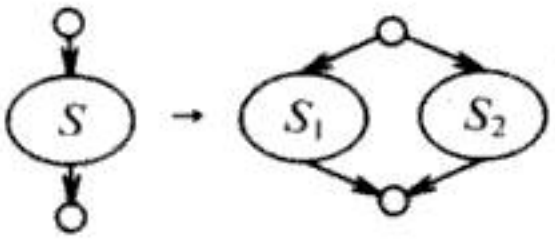
$$
\begin{aligned}
gen\,[S] &= \{d\} \\
desact\,[S] &= D_a - \{d\} \\
sal\,[S] &= gen\,[S] \cup (in\,[S] - desact\,[S])
\end{aligned}
$$

(b)



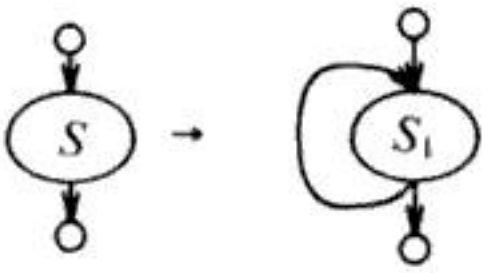
$$
\begin{aligned}
gen\,[S] &= gen\,[S_2] \cup (gen\,[S_1] - desact\,[S_2]) \\
desact\,[S] &= desact\,[S_2] \cup (desact\,[S_1] - gen\,[S_2]) \\
ent\,[S_1] &= ent\,[S] \\
ent\,[S_2] &= sal\,[S_1] \\
sal\,[S] &= sal\,[S_2]
\end{aligned}
$$

(c)



$$
\begin{aligned}
gen\,[S] &= gen\,[S_1] \cup gen\,[S_2] \\
desact\,[S] &= desact\,[S_1] \cap desact\,[S_2] \\
ent\,[S_1] &= ent\,[S] \\
ent\,[S_2] &= ent\,[S] \\
sal\,[S] &= sal\,[S_1] \cup sal\,[S_2]
\end{aligned}
$$

(d)



$$
\begin{aligned}
gen\,[S] &= gen\,[S_1] \\
desact\,[S] &= desact\,[S_1] \\
ent\,[S_1] &= ent\,[S] \cup gen\,[S_1] \\
sal\,[S] &= sal\,[S_1]
\end{aligned}
$$

Fig. 10.21. Ecuaciones de flujo de datos para definiciones de alcance.

la definición *d* esté en *gen* [*S*] si *d* alcanza el fin de *S*, independientemente de que alcance o no el comienzo de *S*. Dicho de otra forma, *d* debe aparecer en *S* y alcanzar el fin de *S* a través de un camino que no salga de *S*. Esta es la justificación para señalar que *gen* [*S*] es el conjunto de definiciones "generadas por *S*".

De manera similar, se pretende que *desact* [*S*] sea el conjunto de definiciones que nunca alcanzan el fin de *S*, aunque alcancen el comienzo. Por tanto, tiene sen-

---

[4] Un lazo es un caso especial de una región fuertemente conectada e incluye todas sus aristas de retroceso dentro del encabezamiento.

tido considerar estas definiciones como "desactivadas por S". Para que la definición d esté en *desact* [S], cada camino desde el comienzo al fin de S debe tener una definición no ambigua de la misma variable definida por d, y si d aparece en S, entonces debe haber otra definición de la misma variable siguiendo cada ocurrencia de d a lo largo de cualquier camino[5].

Las reglas para *gen* y *desact*, siendo traducciones sintetizadas, son relativamente fáciles de entender. Para comenzar, obsérvense las reglas de la figura 10.21(a) para una asignación simple a la variable a. Evidentemente, esta asignación es una definición de a, por ejemplo, la definición d. Entonces d es la única definición que alcanza el fin de la proposición independientemente de que alcance el comienzo. Por tanto,

$$gen\,[S] = \{d\}$$

Por otra parte, d "desactiva" todas las otras definiciones de a, así que se escribe

$$desact\,[S] = D_a \quad \{d\}$$

donde D_a es el conjunto de todas las definiciones en el programa para la variable a.

La regla para una cascada de proposiciones, que se ilustra en la figura 10.21(b), es un poco más sutil. ¿En qué circunstancias la definición d es generada por $S = S_1 ; S_2$? Primero, si es generada por S_2, entonces es generada por S. Si d es generada por S_1, alcanzará el fin de S suponiendo que S_2 no la desactive. Así, se escribe

$$gen\,[S] = gen\,[S_2] \cup (gen\,[S_1] - desact\,[S_2])$$

Se aplican razonamientos similares a la desactivación de una definición, de modo que se tiene

$$desact\,[S] = desact[S_2] \cup (desact\,[S_1] - gen\,[S_2])$$

Para la proposición **if**, ilustrada en la figura 10.21(c), se observa que si cualquiera de las ramas del "**if**" genera una definición, entonces esa definición alcanza el fin de la proposición S. Por tanto,

$$gen\,[S] = gen\,[S_1] \cup gen\,[S_2]$$

Sin embargo, para "desactivar" una definición d, la variable definida por d se debe desactivar a lo largo de cualquier camino que vaya desde el comienzo al final de S. Además, se debe desactivar a lo largo de una de las ramas, de modo que

$$desact\,[S] = desact\,[S_1] \cap desact\,[S_2]$$

Por último, considérense las reglas para los lazos de la figura 10.21(d). Puesto de manera sencilla, el lazo no afecta a *gen* o *desact*. Si la definición d se genera dentro de S_1, entonces alcanza el final de S_1 y el final de S. A la inversa, si d se genera dentro de S, sólo puede ser generada dentro de S_1. Si S_1 desactiva d, entonces recorrer

---

[5] En esta sección introductoria se supone que todas las definiciones son no ambiguas. La sección 10.8 estudia las modificaciones necesarias para manejar definiciones ambiguas.

el lazo no servirá de nada; se redefine la variable de d dentro de S_1 cada vez que se hace un recorrido. A la inversa, si S desactiva d, entonces debe ser desactivada por S_1. La conclusión es:

$$gen\,[S] = gen\,[S_1]$$
$$desact\,[S] = desact\,[S_1]$$

Estimación conservadora de la información sobre el flujo de datos

Hay un error de cálculo sutil en las reglas para *gen* y *desact* de la figura 10.21. Se ha dado por supuesto que la expresión condicional E en las proposiciones **if** y **do** no se interpretan; es decir, existen entradas al programa que hacen que sus ramas vayan a una de las dos partes. Dicho de otra forma, se supone que cualquier camino de la teoría de grafos en el diagrama de flujo es también un *camino de ejecución*, es decir, un camino que se recorre cuando se ejecuta el programa con al menos una entrada posible.

Este no siempre es el caso, y de hecho no se puede decidir en general si se puede o no tomar una ramificación. Supóngase, por ejemplo, que la expresión E en una proposición **if** fuera siempre verdadera. Entonces nunca podría tomarse el camino a través de S_2 en la figura 10.21(c). Esto tiene dos consecuencias. Primero, una definición generada por S_2 no realmente generada por S, porque no hay forma de llegar desde el comienzo de S a la proposición S_2. Segundo, ninguna definición dentro de *desact* [S] puede alcanzar el fin de S. Por tanto, cada definición de este tipo debería estar lógicamente en *desact* [S], aunque no esté en *desact* [S_2].

Cuando se compara el conjunto *gen* calculado con el "verdadero" *gen*, se descubre que el *gen* verdadero es siempre un subconjunto del *gen* calculado. Por otra parte, el *desact* verdadero es siempre un supraconjunto del *desact* calculado. Estas contenciones se aplican incluso después de considerar las otras reglas de la figura 10.21. Por ejemplo, si la expresión E en una proposición **do-**S**-while-**E nunca puede ser falsa, entonces nunca se saldrá del lazo. Por tanto, el *gen* verdadero es $\varnothing$, y toda definición es desactivada por el lazo. El caso de una cascada de proposiciones, en la figura 10.21(b), donde se debe tener en cuenta la imposibilidad de salir de S_1 o de S_2 debido a un lazo infinito, se deja como ejercicio práctico.

Es natural preguntarse si estas diferencias entre los conjuntos *gen* y *desact* verdaderos y calculados obstaculizan el análisis del flujo de datos. La respuesta está en el uso que se haga de esos datos. En el caso de las definiciones de alcance, normalmente se utiliza la información para inferir que el valor de una variable `x` en un punto se limita a un número pequeño de posibilidades. Por ejemplo, si resulta que las únicas definiciones de `x` que alcanzan este punto son de la forma `x := 1`, se puede inferir que `x` tiene el valor 1 en ese punto. Por tanto, se puede decidir sustituir las referencias a `x` por referencias a 1.

Como consecuencia, no parece serio sobreestimar el conjunto de definiciones que alcanzan un punto; simplemente impide realizar una optimación que se podría hacer legítimamente. Por otra parte, subestimar el conjunto de definiciones es un error fatal; podría conducir a un cambio en el programa que modifique lo que el programa calcula. Por ejemplo, se puede pensar que todas las definiciones de alcance de `x` dan a `x` el valor de 1, y por tanto se sustituye `x` por 1; pero existe otra definición

de alcance no detectada que da a x el valor de 2. Entonces, para el caso de las definiciones de alcance, se dice que un conjunto de definiciones es *seguro* o *conservador* si la estimación es un supraconjunto (no necesariamente un supraconjunto propio) del conjunto verdadero de definiciones de alcance. La estimación se denomina *insegura* si no es necesariamente un supraconjunto del verdadero.

Para cada problema de flujo de datos se debe examinar el efecto de las estimaciones imprecisas sobre los tipos de cambios que puedan causar en los programas. Generalmente se aceptan discrepancias que sean *seguras* en el sentido de que puedan prohibir optimaciones que se podrían hacer legalmente, pero no se aceptan discrepancias que sean *inseguras* en el sentido de que puedan causar "optimaciones" que no preserven el comportamiento observado desde el exterior del programa. En cada problema de flujo de datos, normalmente un conjunto o un supraconjunto (pero no ambos) de la respuesta verdadera es seguro.

Volviendo a las implicaciones de seguridad sobre la estimación de *gen* y *desact* para las definiciones de alcance, obsérvese que las discrepancias, supraconjuntos para *gen* y subconjuntos para *desact* están ambas en la dirección segura. Intuitivamente, incrementar *gen* aumenta el conjunto de definiciones que pueden alcanzar un punto, y no puede impedir que una definición alcance un lugar que realmente alcanzó. Asimismo, reducir *desact* sólo puede incrementar el conjunto de definiciones que alcanzan un punto dado.

Cálculo de *ent* y *sal*

Muchos problemas de flujo de datos se pueden resolver mediante traducciones sintetizadas similares a las utilizadas para calcular *gen* y *desact*. Por ejemplo, se puede pretender determinar, para cada proposición S, el conjunto de variables definidas dentro de S. Esta información se puede calcular con ecuaciones análogas a las de *gen*, sin ni siquiera ser necesarios conjuntos análogos a *desact*. Se puede utilizar, por ejemplo, para determinar cálculos de lazo invariante.

Sin embargo, hay otros tipos de información sobre el flujo de datos, como el problema de las definiciones de alcance que fue utilizado como ejemplo, donde también hay que calcular algunos atributos heredados. Resulta que *ent* es un atributo heredado y que *sal* es un atributo sintetizado que depende de *ent*. Se pretende que *ent* [S] sea el conjunto de definiciones que alcanzan el comienzo de S teniendo en cuenta el flujo del control a lo largo de todo el programa, incluidas las proposiciones fuera de S o dentro de la que S está anidada. El conjunto *sal* [S] se define de manera similar para el final de S. Es importante observar la distinción entre *sal*[S] y *gen*[S]. Este último es el conjunto de definiciones que alcanzan el final de S sin seguir caminos fuera de S.

Como ejemplo de la diferencia, considérese la cascada de proposiciones en la figura 10.21(b). Una proposición d puede ser generada en S_1 y por tanto alcanzar el comienzo de S_2. Si d no está en *desact* [S_2], d alcanzará el final de S_2, y por tanto estará en *sal* [S_2]. Sin embargo, d no está en *gen* [S_2].

Después de calcular *gen* [S] y *desact* [S] de forma ascendente, para todas las proposiciones S se puede calcular *ent* y *sal* comenzando por la proposición que representa el programa completo, sabiendo que *ent* [S_0] = $\varnothing$ si S_0 es el programa com-

pleto. Es decir, ninguna definición alcanza el principio del programa. Para cada uno de los cuatro tipos de proposiciones de la figura 10.21, se puede suponer que se conoce *ent* [*S*]. Se debe utilizar para calcular *ent* para cada una de las subproposiciones de *S* [lo cual es trivial en los casos (b) a (d) e irrelevante en el caso (a)]. Después, recursivamente (de forma descendente) se calcula *sal* para cada una de las subproposiciones S_1 o S_2, y se utilizan estos conjuntos para calcular *sal* [*S*].

El caso más sencillo es la figura 10.21(a), donde la proposición es una asignación. Suponiendo que se conoce *ent* [*S*], se calcula *sal* mediante la ecuación (10.5), es decir

$$sal\,[S] = gen\,[S] \cup (ent\,[S] - desact\,[S])$$

En palabras, una definición alcanza el final de *S* si es generada por *S* (es decir, es la definición *d* que es la proposición), o alcanza el comienzo de la proposición y no es desactivada por la proposición.

Supóngase que se ha calculado *ent* [*S*] y que *S* es la cascada de dos proposiciones S_1; S_2, como en el segundo caso de la figura 10.21. Se comienza por observar que $ent\,[S_1] = ent\,[S_2]$. Después, recursivamente se calcula $sal\,[S_1]$, que da $ent\,[S_2]$, puesto que una definición alcanza el comienzo de S_2 si, y sólo si, alcanza el final de S_1. Ahora se puede calcular recursivamente $sal\,[S_2]$, y este conjunto es igual a *sal* [*S*].

A continuación considérese la proposición **if** de la figura 10.21(c). Como se asumió conservadoramente que el control puede fluir por cualquier rama, una definición alcanza el comienzo de S_1 o S_2 exactamente cuando alcanza el comienzo de *S*. Es decir,

$$ent\,[S_1] = ent\,[S_2] = ent\,[S]$$

También se deriva del diagrama de la figura 10.21(c) que una definición alcanza el final de *S* si, y sólo si, alcanza el final de una o de ambas subproposiciones; es decir,

$$sal\,[S] = sal\,[S_1] \cup sal\,[S_2]$$

Por tanto, se pueden utilizar estas ecuaciones para calcular $ent\,[S_1]$ y $ent\,[S_2]$ a partir de *ent* [*S*], calcular recursivamente $sal\,[S_1]$ y $sal\,[S_2]$, y después utilizar éstos para calcular *sal* [*S*].

Utilización de lazos

El último caso, la figura 10.21(d), presenta problemas específicos. Supóngase de nuevo que se tienen $gen\,[S_1]$ y $desact\,[S_1]$, habiéndolos calculado de forma ascendente, y supóngase que se tiene $ent\,[S_1]$ y que se está en el proceso de realizar un recorrido en profundidad del árbol de análisis sintáctico. A diferencia de los casos (b) y (c), no se puede utilizar simplemente *ent* [*S*] como $ent\,[S_1]$, porque las definiciones dentro de S_1 que alcanzan el final de S_1 pueden seguir el arco de regreso al comienzo de S_1, y por tanto, estas definiciones también están en $ent\,[S_1]$. En lugar de eso, se tiene

$$ent\,[S_1] = ent\,[S] \cup sal\,[S_1] \tag{10.6}$$

También se tiene la ecuación obvia para *sal* [*S*]:

$$sal\,[S] = sal\,[S_1]$$

que se puede utilizar una vez que se haya calculado *sal* $[S_1]$. Sin embargo, parece que no se puede calcular *ent* $[S_1]$ por medio de (10.6) hasta que se haya calculado *sal* $[S_1]$, y el plan general ha sido calcular *sal* para una proposición, calculando primero *ent* para dicha proposición.

Por fortuna, hay una manera directa de expresar *sal* en términos de *ent*; viene dada por (10.5), o en este caso concreto:

$$sal\,[S_1] = gen\,[S_1] \cup (ent\,[S_1] - desact\,[S_1]) \tag{10.7}$$

Es importante comprender lo que está ocurriendo en este caso. En realidad no se sabe que (10.7) es verdadera con respecto a una proposición arbitraria S_1; sólo se sospecha que debe ser verdadera porque "tiene sentido" que una definición alcance el final de una proposición si, y sólo si, es generada dentro de la proposición o alcanza el comienzo y no es desactivada. Sin embargo, la única forma que se conoce de calcular *sal* para una proposición es mediante las ecuaciones dadas en la figura 10.21(a) a la (c). Se asume (10.7) y se derivan las ecuaciones para *ent* y *sal* de la figura 10.21(d). Después se pueden utilizar las ecuaciones de la figura 10.21(a) a la (d) para demostrar que (10.7) se cumple para una S_1 arbitraria. Después se podrían reunir estas pruebas para hacer una demostración válida por inducción sobre el tamaño de una proposición S que la ecuación (10.7) y todas las ecuaciones de la figura 10.21 se cumplen para S y todas sus subproposiciones. No se hará así; se dejan las demostraciones como ejercicio, pero el razonamiento que se ha adoptado aquí debe resultar instructivo.

Incluso asumiendo (10.6) y (10.7) no se han resuelto todos los problemas. Estas dos ecuaciones definen una recurrencia para *ent* $[S_1]$ y *sal* $[S_1]$ simultáneamente. Las ecuaciones se reescribirán como

$$\begin{aligned} E &= F \cup A \\ A &= G \cup (E - D) \end{aligned} \tag{10.8}$$

donde E, A, F, G y D corresponden a *ent* $[S_1]$, *sal* $[S_1]$, *ent* $[S]$, *gen* $[S_1]$ y *desact* $[S_1]$, respectivamente. Las dos primeras son variables, las otras tres son constantes.

Para resolver (10.8), supóngase que $A = \varnothing$. Entonces se podría utilizar la primera ecuación de (10.8) para calcular una estimación de E, es decir,

$$E^1 = F$$

A continuación se puede utilizar la segunda ecuación para obtener una mejor estimación de A:

$$A^1 = G \cup (E^1 - D) = G \cup (F - D)$$

Aplicando la primera ecuación a esta nueva estimación de A se obtiene:

$$E^2 = F \cup A^1 = F \cup G \cup (F - D) = F \cup G$$

Si se reaplica la segunda ecuación, la siguiente estimación de A es:

$$A^2 = G \cup (E^2 - D) = G \cup (F \cup G - D) = G \cup (F - D)$$

Obsérvese que $A^2 = A^1$. Por tanto, si se calcula la siguiente estimación de E, será igual a E^1, lo cual dará otra estimación de A igual a A^1, y así sucesivamente. De ese modo, los valores limitadores para E y A son los dados anteriormente por E^1 y A^1. Por tanto, se han obtenido las ecuaciones de la figura 10.21(d), que son

$$ent\,[S_1] = ent\,[S] \cup gen\,[S_1]$$
$$sal\,[S] = sal\,[S_1]$$

La primera de estas ecuaciones proviene del cálculo anterior; la segunda proviene del examen del grafo de la figura 10.21(d).

Queda el detalle de por qué se pudo empezar por la estimación de $A = \varnothing$. Recuérdese que en el estudio de las estimaciones conservadoras se sugirió que los conjuntos como $sal\,[S_1]$, al que representa A, deberían ser sobreestimados en lugar de subestimados. De hecho, si hubiera que comenzar con $A = \{d\}$, donde d es una definición que no aparece ni en F, G o D, entonces d concluiría en los valores limitadores de E y de A.

En este caso, se deben invocar los significados para *ent* y *sal*. Si tal d realmente perteneciera a $ent\,[S_1]$, tendría que haber un camino desde donde estuviera d al comienzo de S_1 que mostrara cómo alcanza d ese punto. Si d estuviera fuera de S, entonces D tendría que estar en $ent\,[S]$, en tanto que si d estuviera dentro de S (y por tanto dentro de S_1) tendría que estar en $gen\,[S_1]$. En el primer caso, d estaría en F y por tanto se colocaría en E mediante (10.8). En el segundo caso, d estaría en G y de nuevo se transmitiría a E a través de A en (10.8). La conclusión es que comenzar con una estimación demasiado pequeña y construir de forma ascendente añadiendo más definiciones a E y A es una forma segura de estimar $ent\,[S_1]$.

Representación de conjuntos

Los conjuntos de definiciones, como $gen\,[S]$ y $desact\,[S]$, se pueden representar de forma compacta utilizando vectores de bits. Se asigna un número a cada definición de interés en el grafo de flujo. Entonces el vector de bits que representa un conjunto de definiciones tendrá un 1 en la posición i si, y sólo si, la definición numerada con i está en el conjunto.

El número de una proposición de definición se puede tomar como el índice de la proposición en una matriz que contenga apuntadores a proposiciones. Sin embargo, no todas las definiciones pueden ser de interés durante el análisis global de flujo de datos. Por ejemplo, no hace falta asignar números a las definiciones de temporales utilizadas únicamente dentro de un solo bloque, como la mayoría de temporales generados para la evaluación de expresiones. Por tanto, los números de las definiciones de interés se registrarán normalmente en una tabla aparte.

Una representación de conjuntos mediante un vector de bits permitirá también que las operaciones de conjuntos se implanten eficientemente. La unión y la intersección de dos conjuntos se puede implantar mediante las operaciones lógicas **or** y **and**, respectivamente, que son operaciones básicas en la mayoría de los lenguajes de programación orientados a sistemas. La diferencia $A - B$ de los conjuntos A y B se puede implantar tomando el complemento de B y utilizando después **and** lógico para calcular $A \wedge \neg B$.

Ejemplo 10.13. En la figura 10.22 se muestra un programa con siete definiciones, indicadas por d_1 hasta d_7 en los comentarios a la izquierda de las definiciones. Los vectores de bits que representan los conjuntos *gen* y *desact* para las proposiciones de la figura 10.22 se muestran a la izquierda de los nodos del árbol sintáctico de la figura 10.23. Los mismos conjuntos fueron calculados aplicando las ecuaciones para flujo de datos de la figura 10.21 a las proposiciones representadas por los nodos del árbol sintáctico.

```
/* d1 */      i := m-1;
/* d2 */      j := n;
/* d3 */      a := u1;
              do
/* d4 */           i := i+1;
/* d5 */           j := j-1;
                   if e1 then
/* d6 */                a := u2
                   else
/* d7 */                i := u3
          while e2
```

Fig. 10.22. Programa para ilustrar definiciones de alcance.

Considérese el nodo para d_7 en el ángulo inferior derecho de la figura 10.23. El conjunto *gen* $\{d_7\}$ viene representado por 000 0001 y el conjunto *desact* $\{d_1, d_4\}$ por 100 1000. Es decir, d_7 desactiva todas las otras definiciones de i, su variable.

El segundo y el tercer hijo del nodo **if** representan las partes **then** y **else**, respectivamente, del condicional. Obsérvese que el conjunto *gen* 000 0011 en el nodo **if** es

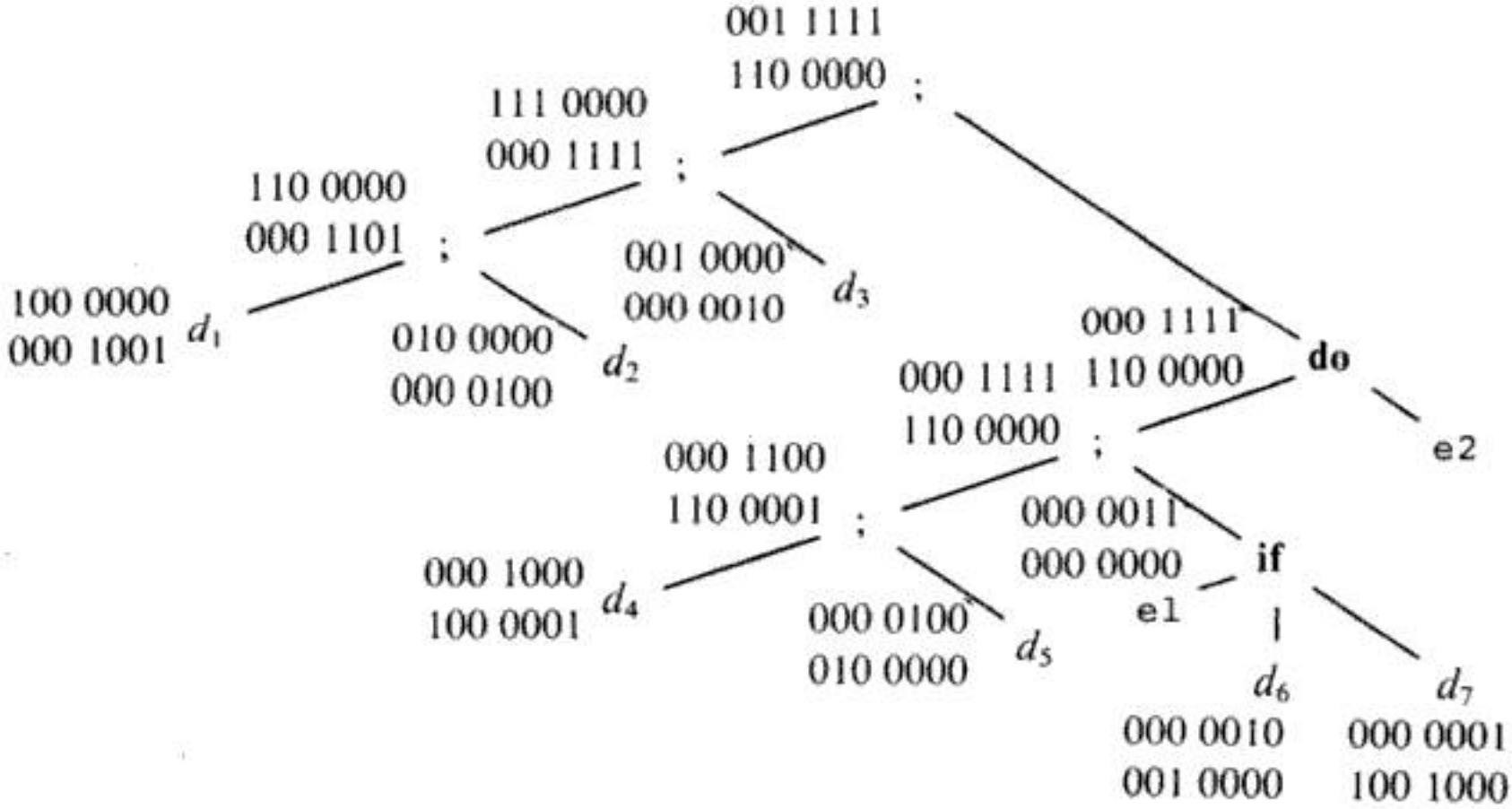


Fig. 10.23. Los conjuntos *gen* y *desact* en los nodos de un árbol sintáctico.

la unión de los conjuntos 000 0010 y 000 0001 en el segundo y tercer hijos. El conjunto *desact* está vacío porque las definiciones desactivadas por las partes **then** y **else** son disjuntas.

Las ecuaciones de flujo de datos para una cascada de proposiciones se aplican al padre del nodo **if**. El conjunto *desact* en este nodo se obtiene por

$$000\ 0000 \cup (110\ 0001 - 000\ 0011) = 110\ 0000$$

En palabras, nada es desactivado por el condicional y d_7, desactivado por la proposición d_4, es generado por el condicional, de modo que sólo d_1 y d_2 están en el conjunto *desact* del padre del nodo **if**.

Ahora se puede calcular *ent* y *sal*, comenzando desde arriba en el árbol de análisis sintáctico. Se supone que el conjunto *ent* en la raíz del árbol sintáctico está vacío. Por tanto, el conjunto *sal* para el hijo izquierdo de la raíz es el conjunto *gen* para dicho nodo, o 111 0000. Este es también el valor del conjunto *ent* en el nodo **do**. Según las ecuaciones de flujo de datos asociadas con la producción **do** de la figura 10.21, el conjunto *ent* para la proposición dentro del lazo **do** se obtiene tomando la unión del conjunto *ent* 111 0000 en el nodo **do** y el conjunto *gen* 000 1111 en la proposición. La unión produce 111 1111, de modo que todas las definiciones pueden alcanzar el comienzo del cuerpo del lazo. Sin embargo, en el punto justo anterior a la definición d_5, el conjunto *ent* es 011 1110, ya que la definición d_4 desactiva d_1 y d_7. El equilibrio de los cálculos *ent* y *sal* se deja como ejercicio. □

Definiciones de alcance local

El espacio para la información sobre el flujo de datos se puede cambiar por tiempo, guardando información sólo en ciertos puntos y recalculando la información en los puntos intermedios cuando sea necesario. Los bloques básicos se consideran generalmente como una unidad durante el análisis global de flujo, limitando la atención sólo a aquellos puntos que estén al comienzo de los bloques. Como generalmente hay muchos más puntos que bloques, limitar el esfuerzo a los bloques es un ahorro significativo. Cuando se necesiten, las definiciones de alcance para todos los puntos de un bloque se pueden calcular a partir de las definiciones de alcance para el comienzo del bloque.

Con más detalle, considérese una secuencia de asignaciones $S_1 ; S_2 ; \ldots ; S_n$ en un bloque básico B. El comienzo de B es el punto p_0, el punto entre las proposiciones S_i y S_{i+1}, p_i y el final del bloque el punto p_n. Las definiciones que alcanzan el punto p_j se pueden obtener de *ent* $[B]$ considerando las proposiciones $S_1 ; S_2 ; \ldots ; S_j$, comenzando con S_1 y aplicando las ecuaciones de flujo de datos de la figura 10.21 para cascadas de proposiciones. Al inicio, sea $D = ent\ [B]$. Cuando se considera S_i, se eliminan de D las definiciones desactivadas por S_i y se añaden las definiciones generadas por S_i. Al final, D está formado por las definiciones que alcanzan p_j.

Cadenas de uso y definición

A menudo es conveniente almacenar la información de las definiciones de alcance como "cadenas de uso y definición", que son listas, para cada uso de una variable, de todas las definiciones que alcanzan dicho uso. Si un uso de una variable a en el

bloque B no viene precedido por ninguna definición no ambigua de a, entonces la cadena de definición y uso para dicho uso de a es el conjunto de definiciones en *ent* [B] que son definiciones de a. Si hay definiciones no ambiguas de a dentro de B que preceden este uso de a, entonces sólo la última de dichas definiciones de a estará en la cadena de uso y definición, y no se coloca *ent* [B] en ella. Además, si hay definiciones ambiguas de a, entonces todas aquellas para las que no haya ninguna definición ambigua de a entre ella y el uso de a estarán en la cadena de uso y definición para este uso de a.

Orden de evaluación

Las técnicas para conservar espacio durante la evaluación de atributos, estudiadas en el capítulo 5, también se aplican al cálculo de la información sobre el flujo de datos utilizando especificaciones como la de la figura 10.21. Específicamente, la única limitación en cuanto al orden de evaluación para los conjuntos de proposiciones *gen*, *desact*, *ent* y *sal* es el impuesto por las dependencias entre estos conjuntos. Habiendo elegido un orden de evaluación, se puede desocupar el espacio para un conjunto después de que se hayan producido todos sus usos.

Las ecuaciones de flujo de datos de esta sección difieren en un aspecto de las reglas semánticas para atributos del capítulo 5: en el capítulo 5 no se permitían las dependencias circulares entre atributos, pero no se ha visto que las ecuaciones de flujo de datos pueden tener dependencias circulares; por ejemplo, *ent* [S_1] y *sal* [S_1] dependen el uno del otro en 10.8. En el caso del problema de las definiciones de alcance, se pueden reescribir las ecuaciones de flujo de datos para eliminar la circularidad —compárense las ecuaciones no circulares de la Fig. 10.21 con las ecuaciones de 10.8—. Una vez que se obtiene una especificación no circular, se pueden aplicar las técnicas del capítulo 5 para obtener soluciones eficientes de ecuaciones de flujo de datos.

Flujo de control general

El análisis de flujo de datos debe tener en cuenta todos los caminos del control. Si los caminos del control resultan evidentes a partir de la sintaxis, entonces las ecuaciones de flujo de datos se pueden establecer y resolver de manera dirigida por la sintaxis, como en esta sección. En el caso de aquellos programas que pueden contener proposiciones **goto**, o incluso las proposiciones más disciplinadas **break** y **continue**, el enfoque elegido debe modificarse para que tenga en cuenta los caminos reales del control.

Se pueden adoptar varios enfoques. El método iterativo de la siguiente sección trabaja para grafos de flujo arbitrarios. Como los grafos de flujo obtenidos en la presencia de proposiciones **break** y **continue** son reducibles, se pueden considerar sistemáticamente dichas construcciones utilizando los métodos basados en intervalos, que se estudiará en la sección 10.10.

Sin embargo, no es necesario abandonar el enfoque dirigido por la sintaxis cuando se permiten proposiciones **break** y **continue**. Antes de terminar esta sección se considera un ejemplo que propone cómo alojar las proposiciones **break**. Estas ideas se desarrollarán en la sección 10.10.

Ejemplo 10.14. La proposición **break** dentro del lazo **do-while** de la figura 10.24 es equivalente a un salto al final del lazo. ¿Cómo hay que definir entonces el conjunto *gen* para la siguiente proposición?

```
if e3 then a := u2
else begin i := u3; break end
```

Se define el conjunto *gen* como $\{d_6\}$, donde d_6 es la definición `a:=u2`, porque d_6 es la única definición generada a lo largo de los caminos del control desde el punto inicial al final de la proposición. La definición d_7, es decir, `i:=u3`, se tomará en cuenta al considerar el lazo **do-while** completo.

```
/* d1 */    i := m-1;
/* d2 */    j := n;
/* d3 */    a := u1;
            do
/* d4 */        i := i+1;
/* d5 */        j := j-1;
               if e3 then
/* d6 */            a := u2
               else begin
/* d7 */                i := u3;
                       break
                end
            while e2
```

Fig. 10.24. Programa que contiene una proposición **break**.

Hay un truco de programación que permite ignorar el salto causado por la proposición **break** mientras se procesan las proposiciones dentro del cuerpo del lazo: se consideran los conjuntos *gen* y *desact* para una proposición **break** como el conjunto vacío y U, el conjunto universal de todas las definiciones, respectivamente, como se muestra en la figura 10.25. Los conjuntos restantes *gen* y *desact* que se muestran en la figura 10.25 se determinan por medio de las ecuaciones de flujo de datos de la figura 10.21, y el conjunto *gen* se muestra encima del conjunto *desact*. Las proposiciones S_1 y S_2 representan secuencias de asignaciones. Los conjuntos *gen* y *desact* en el nodo **do** quedan por determinar.

No se puede alcanzar el punto final de una secuencia de proposiciones que finalice con una proposición **break**, así que se puede considerar el conjunto *gen* para la secuencia como $\varnothing$ y el conjunto *desact* como U; el resultado seguirá siendo una estimación conservadora de *ent* y *sal*. De manera similar, el punto final de la proposición **if** sólo se puede alcanzar a través de la parte **then**, y los conjuntos *gen* y *desact* de la figura 10.25 son los mismos que los de su segundo hijo.

Los conjuntos *gen* y *desact* en el nodo **do** deben tener en cuenta todos los caminos desde el comienzo hasta el final de la proposición **do**, así que se ven afectados por la proposición **break**. Ahora se calcularán dos conjuntos, G y D, inicialmente

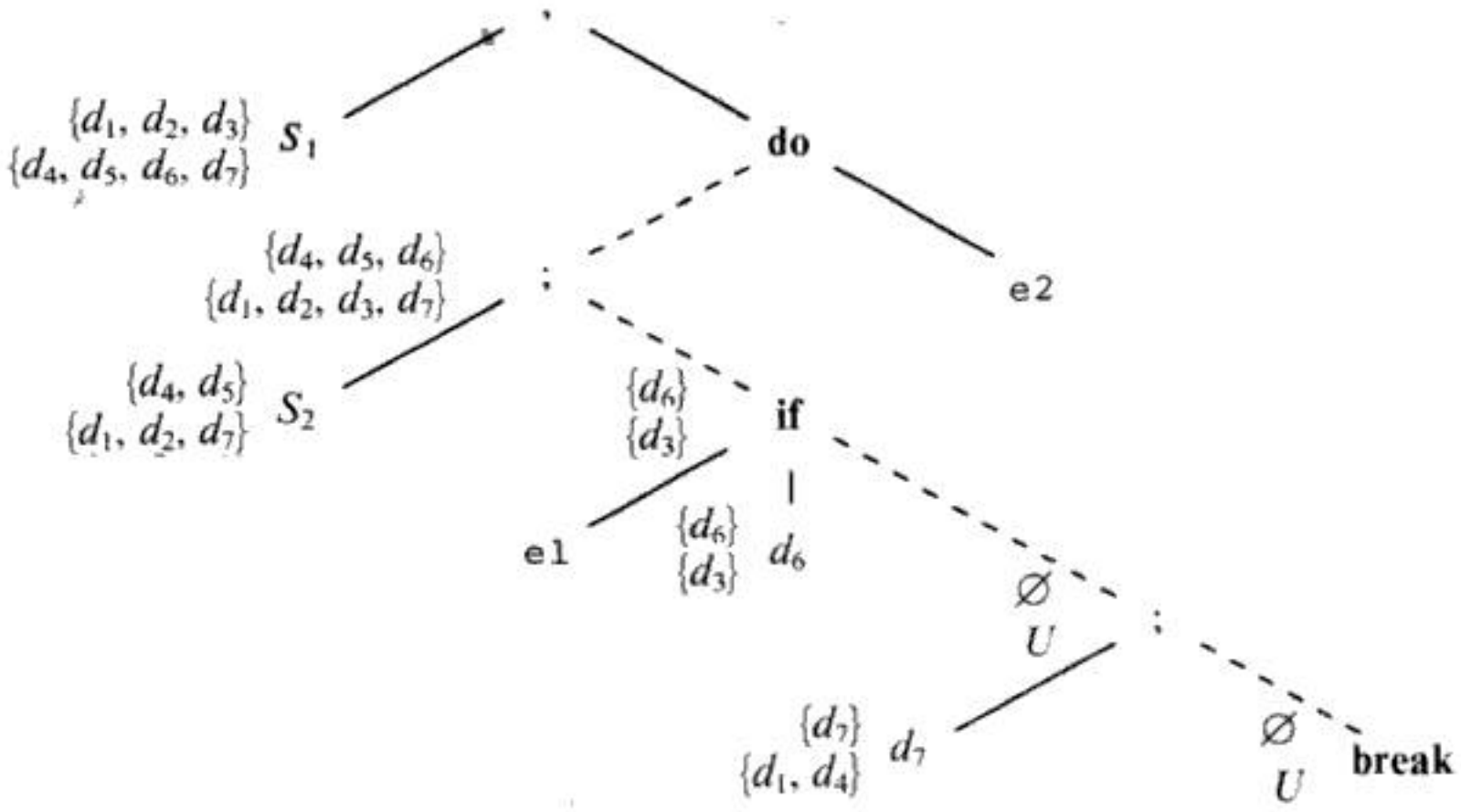


Fig. 10.25. Efecto de una proposición **break** en los conjuntos *gen* y *desact*.

vacíos, a medida que se recorre el camino de puntos desde el nodo **do** hasta el nodo **break**. La intuición es que G y D representan las definiciones generadas y desactivadas conforme el control fluye a la proposición **break** desde el comienzo del cuerpo del lazo. Entonces se puede determinar el conjunto *gen* para la proposición **do-while** tomando la unión de G y el conjunto *gen* para el cuerpo del lazo, porque el control puede alcanzar el final de **do** desde la proposición **break** o cayendo por el cuerpo del lazo. Por la misma razón, se determina el conjunto *desact* para el **do** tomando la intersección de D y el conjunto *desact* del cuerpo del lazo.

Antes de alcanzar el nodo **if** se tiene $G = gen\,[S_2] = \{d_4,d_5\}$ y $D = desact\,[S_2] = \{d_1,d_2,d_7\}$. En el nodo **if** interesa el caso en que el control fluye a la proposición **break**, de modo que la parte **then** del condicional no afecta a G y D. El siguiente nodo a lo largo del camino de puntos es para una secuencia de proposiciones, así que se calculan nuevos valores de G y D. Escribiendo S_3 para la proposición representada por el hijo izquierdo del nodo de secuencia (el etiquetado con d_7), se utiliza

$$G := gen\,[S_3] \cup (G - desact\,[S_3])$$
$$D := desact\,[S_3] \cup (D - gen\,[S_3])$$

Por tanto, los valores de G y D al llegar a la proposición **break** son $\{d_5, d_7\}$ y $\{d_1, d_2, d_4\}$, respectivamente. □

10.6 SOLUCION ITERATIVA DE LAS ECUACIONES DE FLUJO DE CONTROL

El método de la última sección es sencillo y eficiente cuando se puede aplicar, pero para lenguajes como FORTRAN o Pascal que admiten grafos de flujo arbitrarios, no es lo bastante general. La sección 10.10 estudia el "análisis de intervalos", una forma de obtener las ventajas del enfoque dirigido por la sintaxis en el análisis de

flujo de datos para grafos de flujo en general, a costa de una complejidad conceptual considerablemente mayor.

Aquí se estudiará otro enfoque importante para resolver problemas de flujo de datos. En lugar de intentar utilizar el árbol de análisis sintáctico para obtener el cálculo de los conjuntos *ent* y *sal*, primero se construye el grafo de flujo y después se calcula *ent* y *sal* para cada nodo simultáneamente. Mientras se analiza este nuevo método, también se aprovechará la oportunidad de mostrar al lector muchos problemas distintos de análisis del flujo de datos, algunas de sus aplicaciones y las diferencias entre los problemas.

Las ecuaciones para muchos problemas de flujo de datos son similares en la forma en que la información se "genera" y se "desactiva". Sin embargo, hay dos formas principales en que las ecuaciones difieren en detalle.

1. Las ecuaciones de la última sección para definiciones de alcance son ecuaciones *de avance* en el sentido de que los conjuntos *sal* se calculan según los conjuntos *ent*. También hay problemas de flujo de datos que son *de retroceso* en que los conjuntos *ent* se calculan a partir de los conjuntos *sal*.
2. Cuando hay más de una arista que entra a un bloque B, las definiciones que alcanzan el comienzo de B son la unión de las definiciones que llegan a lo largo de cada una de las aristas. Se dice por tanto que la unión es el *operador de confluencia*. Por el contrario, se considerarán problemas como las expresiones globales disponibles, donde la intersección es el operador de confluencia, porque una expresión sólo está disponible al comienzo de B si está disponible al final de cada predecesor de B. En la sección 10.11 se verán otros ejemplos de operadores de confluencia.

En esta sección se estudiarán ejemplos tanto de ecuaciones de avance y de retroceso, donde la unión y la intersección se turnan como operador de confluencia.

Algoritmo iterativo para definiciones de alcance

Para cada bloque básico B se definen *sal* [B], *gen* [B], *desact* [B] y *ent* [B], como en la última sección, observando que cada bloque B puede ser considerado como una proposición que es la cascada de una o más proposiciones de asignación. Suponiendo que se han calculado *gen* y *desact* para cada bloque, se pueden crear dos grupos de ecuaciones, que se muestran en (10.9) más adelante, que relacionan *ent* y *sal*. El primer grupo de ecuaciones se deriva de la observación de que *ent* [B] es la unión de las definiciones que llegan desde todos los predecesores de B. El segundo grupo son casos especiales de la ley general (10.5) que se desea ver cumplida para todas las proposiciones. Estos dos grupos son:

$$\begin{aligned} ent\,[B] &= \bigcup_{\substack{P \text{ un prede-}\\ \text{cesor de } B}} sal\,[P] \\ sal\,[B] &= gen\,[B] \cup (ent\,[B] - desact\,[B]) \end{aligned} \tag{10.9}$$

Si un grafo de flujo tiene n bloques básicos, se obtienen $2n$ ecuaciones de (10.9). Las $2n$ ecuaciones se pueden resolver considerándolas como recurrencias para calcular los conjuntos *ent* y *sal*, igual que se resolvieron las ecuaciones de flujo de datos (10.6)

y (10.5) para proposiciones **do-while** en la última sección. En la última sección se comenzó con el conjunto vacío de definiciones como la estimación inicial para todos los conjuntos *sal*. Aquí se comenzará con conjuntos *ent* vacíos porque se vio en (10.9) que los conjuntos *ent*, siendo la unión de conjuntos *sal*, estarán vacíos si lo están los conjuntos *sal*. Mientras se podía afirmar que las ecuaciones (10.6) y (10.7) sólo necesitaban una iteración, en el caso de estas ecuaciones más complejas no se puede limitar *a priori* el número de iteraciones.

Algoritmo 10.2. Definiciones de alcance.

Entrada. Un grafo de flujo para el que se han calculado *desact* [*B*] y *gen* [*B*] para cada bloque *B*.

Salida. *ent* [*B*] y *sal* [*B*] para cada bloque *B*.

Método. Se utiliza un enfoque iterativo, comenzando por la "estimación" *ent* [*B*] $= \varnothing$ para todo *B* y convergiendo a los valores deseados de *ent* y *sal*. Como hay que iterar hasta que converjan los conjuntos *ent* (y por tanto los conjuntos *sal*), se utiliza una variable booleana, *cambio*, para registrar en cada pasada por los bloques si ha cambiado cualquiera de los conjuntos *ent*. El algoritmo se perfila en la figura 10.26. □

```
    /* inicializar sal sobre la suposición de que ent [B] = ∅ para todo bloque B */
(1) for cada bloque B do sal [B] := gen [B];
(2) cambio := true;     /* para que avance el lazo while */
(3) while cambio do begin
(4)     cambio := false;
(5)     for cada bloque B do begin
(6)         ent[B] := ∪ sal [P];
                      P un predecesor de B
(7)         salant := sal [B];
(8)         sal [B] := gen [B] ∪ (ent [B] − desact [B]);
(9)         if sal [B] ≠ salant then cambio := true
        end
end
```

Fig. 10.26. Algoritmo para calcular *ent* y *sal*.

Intuitivamente, el algoritmo 10.2 propaga las definiciones tan lejos como fueran sin ser desactivadas, simulando en un sentido todas las posibles ejecuciones del programa. Las notas bibliográficas contienen referencias donde se pueden encontrar pruebas formales de la exactitud de este y otros problemas de análisis de flujo de datos.

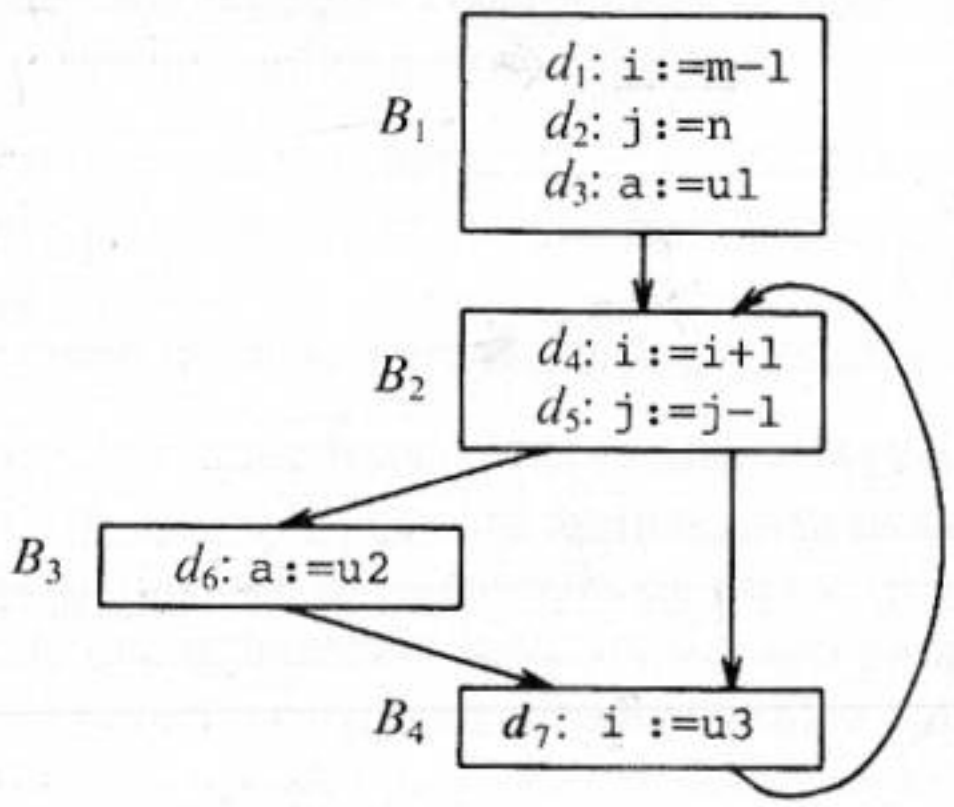
Se ve que el algoritmo se detendrá finalmente porque *sal* [*B*] nunca disminuye de tamaño para cualquier *B*; una vez que se añade una definición, ésta permanece para siempre. (La demostración de este hecho se deja como un ejercicio inductivo.) Como el conjunto de todas las definiciones es finito, en algún momento debe haber una pasada del lazo **while** en la que *salant* = *sal* [*B*] para cada *B* de la línea (9).

Entonces *cambio* seguirá igual a **false** y el algoritmo termina. Se puede terminar en ese momento porque si los conjuntos *sal* no han cambiado, los conjuntos *ent* no cambiarán en la siguiente pasada. Y si los conjuntos *ent* no cambian, los conjuntos *sal* no pueden cambiar, de modo que no puede haber más cambios en las pasadas posteriores.

Se puede demostrar que un límite superior en cuanto al número de veces que se itera el lazo **while** es el número de nodos en el grafo de flujo. Intuitivamente, la razón es que si una definición alcanza un punto, lo puede hacer a lo largo de un camino sin lazos y el número de nodos en un grafo de flujo es un límite superior en cuanto al número de nodos en un camino sin lazos. Cada vez que se itera el lazo **while**, la definición progresa en por lo menos un nodo a lo largo del camino en cuestión.

De hecho, si se ordenan adecuadamente los bloques en el lazo **for** de la línea (5), hay una evidencia empírica de que el promedio de iteraciones sobre programas reales es menor que 5 (véase Sec. 10.10). Ya que los conjuntos se pueden representar mediante vectores de bits y las operaciones de estos conjuntos se pueden implantar mediante operaciones lógicas en vectores de bits, el algoritmo 10.2 es sorprendentemente eficiente en la práctica.

Ejemplo 10.15. Se obtuvo el grafo de flujo de la figura 10.27 del programa de la figura 10.22 de la última sección. El algoritmo 10.2 se aplicará a este grafo de flujo para poder comparar los enfoques de las dos secciones.



$gen\,[B_1] = \{d_1, d_2, d_3\}$
$desact\,[B_1] = \{d_4, d_5, d_6, d_7\}$

$gen\,[B_2] = \{d_4, d_5\}$
$desact\,[B_2] = \{d_1, d_2, d_7\}$

$gen\,[B_3] = \{d_6\}$
$desact\,[B_3] = \{d_3\}$

$gen\,[B_4] = \{d_7\}$
$desact\,[B_4] = \{d_1, d_4\}$

Fig. 10.27. Grafo de flujo para ilustrar definiciones de alcance.

Sólo son interesantes las definiciones $d_1, d_2, \ldots, d_7$ que definen `i`, `j` y `a` en la figura 10.27. Como en la última sección, los conjuntos de definiciones se representarán por medio de vectores de bits, donde el bit i de la izquierda representa la definición d_i.

El lazo de la línea (1) en la figura 10.26 inicializa $sal\,[B] = gen\,[B]$ para cada B, y esos valores iniciales de $sal\,[B]$ se muestran en la tabla de la figura 10.28. No se calculan ni utilizan los valores iniciales ($\varnothing$) de cada conjunto $ent\,[B]$ pero se mues-

tran para ser más completos. Supóngase que el lazo for de la línea (5) se ejecuta con $B = B_1, B_2, B_3, B_4$, por ese orden. Con $B = B_1$, no hay predecesores para el nodo inicial de modo que *ent* $[B_1]$ sigue siendo el conjunto vacío, representado por 000 0000; como resultado, *sal* $[B_1]$ sigue siendo igual a *gen* $[B_1]$. Este valor no difiere de *salant* calculado en la línea (7), así que todavía se asigna *cambio* a **true**.

Después se considera $B = B_2$ y se calcula

$$\begin{aligned} ent\,[B_2] &= sal\,[B_1] \cup sal\,[B_3] \cup sal\,[B_4] \\ &= 111\ 0000 + 000\ 0010 + 000\ 0001 = 111\ 0011 \\ sal\,[B_2] &= gen\,[B_2] \cup (ent\,[B_2] - desact\,[B_2]) \\ &= 000\ 1100 + (111\ 0011 - 110\ 0001) = 001\ 1110 \end{aligned}$$

Este cálculo se resume en la figura 10.28. Al final de la primera pasada, *sal* $[B_4]$ = 001 0111, reflejando el hecho de que se genera d_7 y de que d_3, d_5 y d_6 alcanzan B_4 y no se desactivan en B_4. A partir de la segunda pasada no hay cambios en ninguno de los conjuntos *sal*, así que el algoritmo termina. □

BLOQUE B	INICIAL		PASADA 1		PASADA 2	
	ent $[B]$	*sal* $[B]$	*ent* $[B]$	*sal* $[B]$	*ent* $[B]$	*sal* $[B]$
B_1	000 0000	111 0000	000 0000	111 0000	000 0000	111 0000
B_2	000 0000	000 1100	111 0011	001 1110	111 1111	001 1110
B_3	000 0000	000 0010	001 1110	000 1110	001 1110	000 1110
B_4	000 0000	000 0001	001 1110	001 0111	001 1110	001 0111

Fig. 10.28. Cálculo de *ent* y *sal*.

Expresiones disponibles

Una expresión `x+y` está *disponible* en un punto p si todo camino (no necesariamente sin lazos) desde el nodo inicial hasta p evalúa `x+y`, y después de la última de dichas evaluaciones antes de alcanzar p, no hay asignaciones posteriores a `x` o `y`. Para expresiones disponibles se dice que un bloque *desactiva* una expresión `x+y` si asigna (o puede asignar) `x` o `y` y no recalcula posteriormente `x+y`. Un bloque *genera* la expresión `x+y` si efectivamente evalúa `x+y` y no redefine posteriormente `x` o `y`.

Obsérvese que la noción de "desactivar" o "generar" una expresión disponible no es exactamente la misma que para las definiciones de alcance. Sin embargo, estas nociones de "desactivar" y "generar" obedecen a las mismas leyes que para las definiciones de alcance. Se podrían calcular exactamente como se hizo en la sección 10.5 si se modificaran las reglas de 10.21(a) para una sola proposición de asignación.

El principal uso de la información sobre proposiciones disponibles es para detectar subexpresiones comunes. Por ejemplo, en la figura 10.29, la expresión `4*i` en el bloque B_3 será una subexpresión común si `4*i` está disponible en el punto de entrada del bloque B_3. Estará disponible si a `i` no se le asigna un nuevo valor en el bloque B_2 o si, como en la figura 10.29(b), `4*i` se recalcula después de que se asigna `i` en B_2.

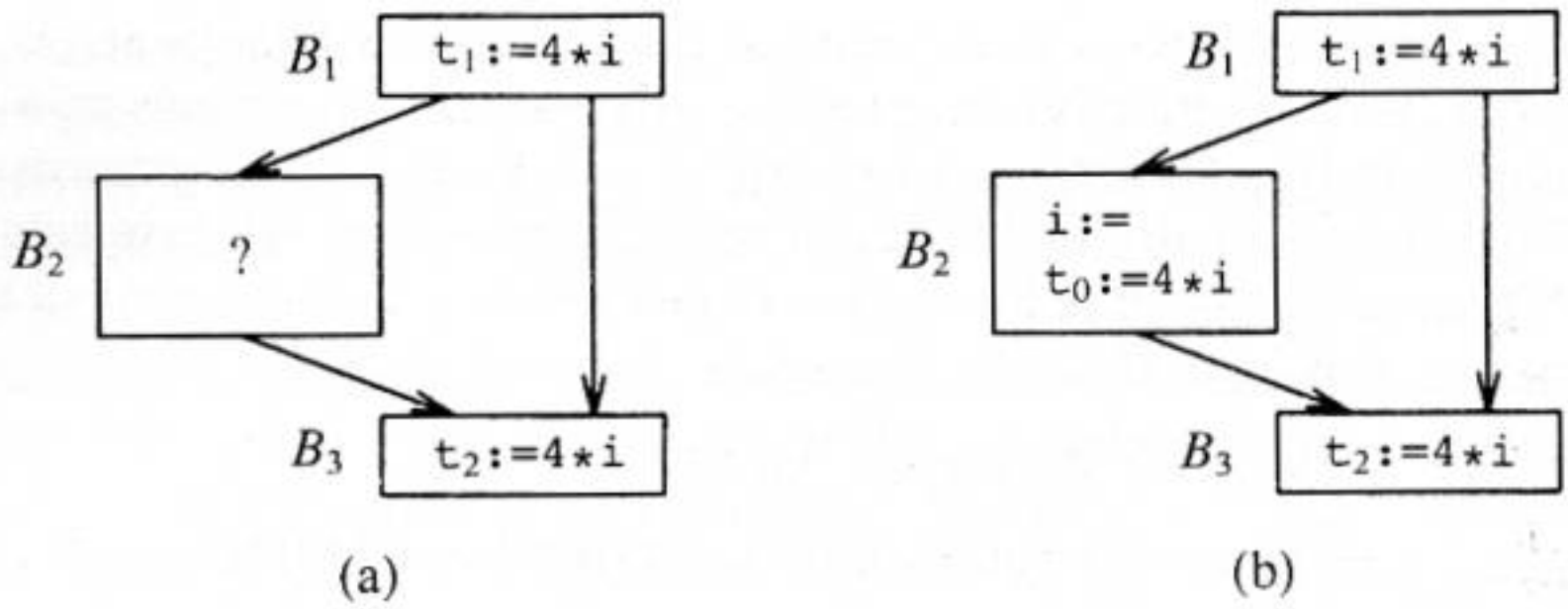


Fig. 10.29. Subexpresiones comunes potenciales a través de bloques.

Se puede calcular fácilmente el conjunto de expresiones generadas para cada punto en un bloque, procediendo desde el comienzo hasta el final del bloque. En el punto anterior al bloque, se supone que no hay expresiones disponibles. Si en el punto p está disponible el conjunto A de expresiones, y q es el punto después de p, con la proposición `x:=y+z` entre ellos, entonces se forma el conjunto de expresiones disponibles en q mediante los siguientes dos pasos:

1. Añadir a A la expresión `y+z`.
2. Borrar de A cualquier expresión que tenga una `x`.

Obsérvese que los pasos se deben tomar en el orden correcto, ya que `x` podría ser la misma que `y` o `z`. Después de llegar al final del bloque, A es el conjunto de expresiones generadas por el bloque. El conjunto de expresiones desactivadas consta de todas las expresiones, por ejemplo, `y+z`, tales que `y` o `z` se define en el bloque, e `y+z` no es generada por el bloque.

Ejemplo 10.16. Considérense las cuatro proposiciones de la figura 10.30. Después de la primera, `b+c` está disponible. Después de la segunda, `a-d` se vuelve disponible, pero `b+c` ya no está disponible porque se ha redefinido `b`. La tercera no vuelve a hacer disponible `b+c` porque el valor de `c` se modifica inmediatamente. Después de la última proposición, `a-d` ya no está disponible porque `d` ha cambiado. Por tanto, no se generan proposiciones y se desactivan todas las proposiciones que involucran a `a`, `b`, `c` o `d`. □

Las expresiones disponibles se pueden encontrar de una forma que recuerda al modo de calcular las definiciones de alcance. Supóngase que U es el conjunto "universal" de todas las expresiones que aparecen a la derecha de una o más proposiciones del programa. Para cada bloque B, sea *ent* [B] el conjunto de expresiones en U disponibles en el punto justo antes del comienzo de B. Para cada bloque B sea *ent* [B] el conjunto de expresiones en U disponible en el punto justo antes del comienzo de B. Sea *sal* [B] el mismo para el punto después del final de B. Se define *e_gen* [B] como las expresiones generadas por B y *e_desact* [B] como el conjunto de expresiones en U desactivadas en B. Obsérvese que *ent*, *sal*, *e_gen* y *e_desact* se pueden re-

presentar mediante vectores de bits. Las siguientes ecuaciones relacionan las incógnitas *ent* y *sal* una con otra y las cantidades conocidas *e_gen* y *e_desact*.

$$\begin{aligned} sal\,[B] &= e\_gen\,[B] \cup (ent\,[B] - e\_desact\,[B]) \\ ent\,[B] &= \bigcap_{\substack{P \text{ un prede-} \\ \text{cesor de } B}} sal\,[P]; \text{ para } B \text{ no inicial} \\ ent\,[B_1] &= \varnothing, \text{ donde } B_1 \text{ es el bloque inicial.} \end{aligned} \tag{10.10}$$

Las ecuaciones (10.10) son casi idénticas a las ecuaciones (10.9) para las definiciones de alcance. La primera diferencia es que *ent* para el nodo inicial se considera como un caso especial. Esto se justifica basándose en que nada está disponible si el programa acaba de comenzar en el nodo inicial, aunque alguna expresión pudiera estar disponible a lo largo de todos los caminos hacia el nodo inicial desde cualquier otra parte del programa. Si no se obligara a *ent* $[B_1]$ a estar vacío, se podría deducir erróneamente que ciertas expresiones estaban disponibles antes de que el programa comenzara.

PROPOSICIONES	EXPRESIONES DISPONIBLES
	..... ninguna
`a := b+c`	
	..... sólo `b+c`
`b := a-d`	
	..... sólo `a-d`
`c := b+c`	
	..... sólo `a-d`
`d := a-d`	
	..... ninguna

Fig. 10.30. Cálculo de expresiones disponibles.

La segunda y más importante diferencia es que el operador de confluencia es la intersección en lugar de la unión. Este operador es el apropiado porque una expresión está disponible al comienzo de un bloque sólo si está disponible al final de todos sus predecesores. Por el contrario, una definición alcanza el comienzo de un bloque siempre que alcanza el fin de uno o más de sus predecesores.

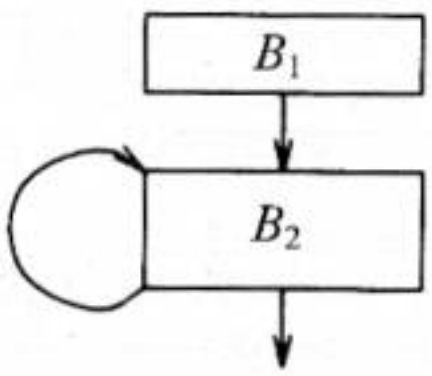
El uso de $\cap$ en lugar de $\cup$ hace que las ecuaciones (10.10) se comporten de manera distinta a las (10.9). Aunque ningún conjunto tiene una única solución, para (10.9) es la menor solución que corresponde a la definición de "alcance", y esa solución se obtuvo comenzando con el supuesto de que nada iba a ninguna parte, y avanzando hacia la solución. De ese modo, nunca se supuso que una definición *d* podía alcanzar un punto *p* a menos que se encontrara un camino real que propagara *d* a *p*. Por el contrario, para las ecuaciones (10.10) se desea la mayor solución posible, así que se comienza con una aproximación que es demasiado larga y que va disminuyendo.

Puede no resultar obvio que comenzando con el supuesto "todo, es decir, el conjunto U, está disponible en todas partes" y eliminando sólo aquellas expresiones para las que se pueda descubrir un camino a lo largo del cual no esté disponible, se alcanza un conjunto de expresiones verdaderamente disponibles. En el caso de expresiones disponibles, es conservador producir un subconjunto del conjunto exacto de expresiones disponibles, y esto es lo que se hace. La razón por la que los subconjuntos son conservadores es que el uso que se va a dar a la información es sustituir el cálculo de una expresión disponible por un valor previamente calculado (véase Algoritmo 10.5 de la siguiente sección), y no saber que una expresión está disponible sólo impide cambiar el código.

Ejemplo 10.17. Se centrará la atención en un solo bloque, B_2 en la figura 10.31, para ilustrar el efecto de la aproximación inicial de *ent* $[B_2]$ en *sal* $[B_2]$. Sean G y D las abreviaturas de *gen* $[B_2]$ y *desact* $[B_2]$, respectivamente. Las ecuaciones de flujo de datos para el bloque B_2 son:

$$ent\,[B_2] = sal\,[B_1] \cap sal\,[B_2]$$
$$sal\,[B_2] = G \cup (ent\,[B_2] - D)$$

Estas ecuaciones se han reescrito como recurrencias en la figura 10.31, siendo E^j y A^j las j-ésimas aproximaciones de *ent* $[B_2]$ y de *sal* $[B_2]$, respectivamente. La figura también muestra que comenzando con $E^0 = \varnothing$ se obtiene $A^1 = A^2 = G$, en tanto que comenzando con $E^0 = U$ se obtiene un conjunto mayor para A^2. Resulta que *sal*$[B_2]$ es igual a A^2 en cada caso, porque las iteraciones convergen todas en los puntos mostrados.



$$A^{j+1} = G \cup (E^j + D)$$
$$E^{j+1} = sal\,[B_1] \cap A^{j+1}$$

$E^0 = \varnothing$	$E^0 = U$
$A^1 = G$	$A^1 = U - D$
$E^1 = sal\,[B_1] \cup G$	$E^1 = sal\,[B_1] - D$
$A^2 = G$	$A^2 = G \cup (sal\,[B_1] - D)$

Fig. 10.31. Inicializar con $\varnothing$ los conjuntos *ent* es demasiado restrictivo.

Intuitivamente, la solución obtenida si se comienza con $E^0 = U$ utilizando

$$sal\,[B_2] = G \cup (sal\,[B_1] - D)$$

es más deseable, porque refleja correctamente el hecho de que las expresiones en *sal* $[B_1]$ que no son desactivadas por B_2 están disponibles al final de B_2, así como las expresiones generadas por B_2. □

Algoritmo 10.3. Expresiones disponibles.

Entrada. Un grafo de flujo G con *e_desact* [B] y *e_gen* [B] calculadas para cada bloque B. El bloque inicial es B_1.

Salida. El conjunto *ent* [B] para cada bloque B.

Método. Ejecútese el algoritmo de la figura 10.32. La explicación de los pasos es similar a la de la figura 10.26. □

```
ent [B1] := ∅;
sal [B1] := e_gen [B1]; /* int y sal nunca cambian para el nodo inicial, B1 */
for B ≠ B1 do sal [B] := U − e_desact [B]; /* la estimación inicial
                                              es demasiado grande */
cambio := true;
while cambio do begin
    cambio := false;
    for B ≠ B1 do begin
        ent [B] := ⋂ (P un predecesor de B) sal [P];
        salant := sal [B];
        sal [B] := e_gen [B] ∪ (ent [B] − e_desact [B]);
        if sal [B] ≠ salant then cambio := true
    end
end
```

Fig. 10.32. Cálculo de las expresiones disponibles.

Análisis de variables activas

Varias transformaciones para mejorar el código dependen de la información calculada en la dirección opuesta al flujo del control de un programa; a continuación se consideran algunas de ellas. En el análisis de *variables activas* se desea conocer para la variable x y el punto p si se pudiera utilizar el valor de x en p a lo largo de algún camino del grafo de flujo que comienza en p. Si es así, se dice que x está *activa* en p; en caso contrario, x está *inactiva* en p.

Como se vio en la sección 9.7, un uso importante de la información sobre variables activas aparece cuando se genera código objeto. Después de que un valor se calcula dentro de un registro y se utiliza presumiblemente dentro de un bloque, no es necesario almacenar ese valor si está inactivo al final del bloque. Asimismo, si todos los registros están llenos y se necesita otro registro, prevalece el uso de un registro con un valor inactivo, puesto que ese valor no tiene que almacenarse en memoria.

Se define *ent* [B] como el conjunto de variables activas en el punto inmediatamente anterior al bloque B y se define *sal* [B] como el conjunto de variables activas en el punto inmediatamente después del bloque. Sea *def* [B] el conjunto de variables a las que se les ha asignado definitivamente un valor en B antes de cualquier uso de dicha variable en B, y sea *uso* [B] el conjunto de variables cuyos valores se pueden

utilizar antes que cualquier definición de la variable. Entonces, las ecuaciones que relacionan *def* y *uso* con las incógnitas *ent* y *sal* son:

$$ent\,[B] = uso\,[B] \cup (sal\,[B] - def\,[B])$$

$$sal\,[B] = \bigcup_{S \text{ un sucesor de } B} ent\,[S]; \qquad (10.11)$$

El primer grupo indica que una variable está activa al entrar a un bloque si se utiliza antes de una redefinición en el bloque o si está activa al salir del bloque y no se redefine en el bloque. El segundo grupo de ecuaciones indica que una variable está activa al salir de un bloque si, y sólo si, está activa al entrar en uno de sus sucesores.

Debe tenerse en cuenta la relación entre (10.11) y las ecuaciones de definiciones de alcance (10.9). Aquí, *ent* y *sal* han intercambiado sus papeles, y *uso* y *def* sustituyen a *gen* y *desact*, respectivamente. Como (10.9), la solución a (10.11) no es necesariamente única, y lo que se desea es la solución más pequeña. El algoritmo que se utiliza para la solución mínima es fundamentalmente una versión hacia atrás del algoritmo 10.2. Como el mecanismo para detectar cambios en cualquiera de los conjuntos *ent* es tan similar a la forma en que se detectan cambios en los conjuntos *sal* de los algoritmos 10.2 y 10.3, se omiten los detalles de la comprobación de la terminación.

Algoritmo 10.4. Análisis de variables activas.

Entrada. Un grafo de flujo con *def* y *uso* calculados para cada bloque.

Salida. sal [*B*], el conjunto de variables activas a la salida de cada bloque *B* del grafo de flujo.

Método. Ejecútese el programa de la figura 10.33.

for cada bloque *B* **do** *ent* [*B*] := $\varnothing$;
while ocurren cambios a cualquiera de los conjuntos *ent* **do**
for cada bloque *B* **do begin**
$sal\,[B] := \bigcup_{S \text{ un sucesor de } B} sal\,[S]$;
$ent\,[B] := uso\,[B] \cup (sal\,[B] - def\,[B])$
end

Fig. 10.33. Cálculo de variables activas.

Cadenas de definición y uso

Un cálculo que se realiza virtualmente de la misma forma que el análisis de variables activas es el *encadenamiento de definición y uso*. Se dice que una variable se *usa* en la proposición *s* si puede que su valor de lado derecho sea necesario. Por ejemplo, se utilizan `b` y `c` (pero no `a`) en cada una de las proposiciones `a:=b+c` y `a[b]:=c`. El problema del encadenamiento de definición y uso consiste en calcular para un

punto p el conjunto de usos s de una variable, por ejemplo x, tal que haya un camino desde p a s que no redefina x.

Como con las variables activas, si se puede calcular *sal* [B], el conjunto de usos alcanzable desde el final del bloque B, entonces se pueden calcular las definiciones alcanzadas desde cualquier punto p dentro del bloque B examinando la parte del bloque B que sigue a p. En concreto, si hay una definición de la variable x en el bloque, se puede determinar la *cadena de definición y uso* para dicha definición, que es la lista de todos los usos posibles de esa definición. El método es análogo al de la sección 10.5 para calcular las cadenas de uso y definición, y se deja al lector.

Las ecuaciones para calcular la información del encadenamiento de definición y uso parecen exactamente iguales que las (10.11) sustituyendo *def* y *uso*. En lugar de *uso* [B], se toma el conjunto de usos *expuestos hacia arriba* en B, es decir, el conjunto de pares (s, x) tales que s es una proposición en B que utiliza la variable x y tales que no ocurre ninguna definición anterior de x en B. En lugar de *def* [B] se toma el conjunto de pares (s, x) tales que s es una proposición que utiliza x, s no está en B, y B tiene una definición de x. Estas ecuaciones se resuelven por el análogo obvio del algoritmo 10.4, y no se estudiará este tema más en profundidad.

10.7 TRANSFORMACIONES PARA MEJORAR EL CODIGO

Los algoritmos para realizar las transformaciones para mejorar el código introducidos en la sección 10.2 dependen de la información sobre el flujo de datos. En las últimas dos secciones se ha visto cómo se puede reunir esta información. Aquí se considera la eliminación de subexpresiones comunes, la propagación de copias y las transformaciones para trasladar los cálculos invariantes de los lazos fuera de ellos y para eliminar las variables de inducción. Para muchos lenguajes se pueden lograr mejoras significativas en el tiempo de ejecución mejorando el código de los lazos. Cuando se implantan dichas transformaciones en un compilador, es posible hacer varias a la vez. Sin embargo, las ideas en que se basan las transformaciones se estudiarán individualmente.

El énfasis de esta sección se centra en las transformaciones globales que utilizan información sobre el programa como un todo. Como se vio en las últimas dos secciones, el análisis global del flujo de datos no considera generalmente los puntos dentro de bloques básicos. Por tanto, las transformaciones globales no son un sustituto de las transformaciones locales; se deben realizar ambas. Por ejemplo, cuando se realiza la eliminación de subexpresiones comunes globales sólo se debe tener en cuenta si una expresión es generada por un bloque y no si se recalcula varias veces dentro de un bloque.

Eliminación de subexpresiones comunes globales

El problema de flujo de datos de las expresiones disponibles estudiado en la última sección permite determinar si una expresión en el punto p en un grafo de flujo es una subexpresión común. El siguiente algoritmo formaliza las ideas intuitivas analizadas en la sección 10.2 para eliminar las subexpresiones comunes.

Algoritmo 10.5. Eliminación de subexpresiones comunes globales.

Entrada. Un grafo de flujo con información sobre las expresiones disponibles.

Salida. Un grafo de flujo revisado.

Método. Para cada proposición s de la forma `x:=y+z`[6] tal que `y+z` esté disponible al comienzo del bloque de s y ni `y` ni `z` se definan antes de la proposición s en ese bloque, hágase lo siguiente:

1. Para descubrir las evaluaciones de `y+z` que alcanzan el bloque de s, se siguen las aristas del grafo de flujo, buscando hacia atrás desde el bloque de s. Sin embargo, no se pasa por ningún bloque que evalúe `y+z`. La última evaluación de `y+z` en cada bloque encontrado es una evaluación de `y+z` que alcanza s.
2. Créese una nueva variable `u`.
3. Sustitúyase cada proposición `w:=y+z` encontrada en 1 por

```
u := y+z
w := u
```

4. Sustitúyase la proposición s por `x:=u`.

Algunos comentarios a este algoritmo son, por orden:

1. La búsqueda en el paso 1 del algoritmo de las evaluaciones de `y+z` que alcanzan la proposición también se puede formular como un problema de análisis de flujo de datos. Sin embargo, no tiene sentido resolverlo para todas las expresiones `y+z` y todas las proposiciones o bloques, porque se reúne demasiada información irrelevante. En lugar de ello, se debería realizar una búsqueda en el grafo de flujo de cada proposición y expresión relevante.
2. No todos los cambios realizados por el algoritmo 10.5 son mejoras. Se puede desear limitar el número de evaluaciones distintas que alcanzan a s encontradas en el paso 1, quizás a una. Sin embargo, la propagación de copias, que se estudiará a continuación, a menudo permite obtener una ventaja aunque varias evaluaciones de `y+z` alcancen s.
3. El algoritmo 10.5 no se dará cuenta de que `a*z` y `c*z` deben tener el mismo valor en

```
a := x+y          c := x+y
b := a*z    vs.   d := c*z
```

porque este sencillo enfoque para las subexpresiones comunes sólo considera las expresiones literales, en lugar de los valores calculados por las expresiones. Kildall [1973] proporciona un método para capturar dichas equivalencias en una pasada; en la sección 10.11 se analizarán estas ideas. Sin embargo, se pueden capturar con múltiples pasadas del algoritmo 10.5, y se puede considerar su repetición hasta que ya no se produzcan más cambios. Si `a` y `c` son variables tem-

---

[6] Recuérdese que se sigue utilizado + como un operador genérico.

porales que no se utilizan fuera del bloque en el que aparecen, entonces la subexpresión común `(x+y)*z` se puede capturar considerando de forma especial las variables temporales, como en el siguiente ejemplo.

Ejemplo 10.18. Supóngase que no hay asignaciones a la matriz `a` en el grafo de flujo de la figura 10.34(a), así que se puede decir sin lugar a dudas que `a[t2]` y `a[t6]` son subexpresiones comunes. El problema consiste en eliminar estas subexpresiones comunes.

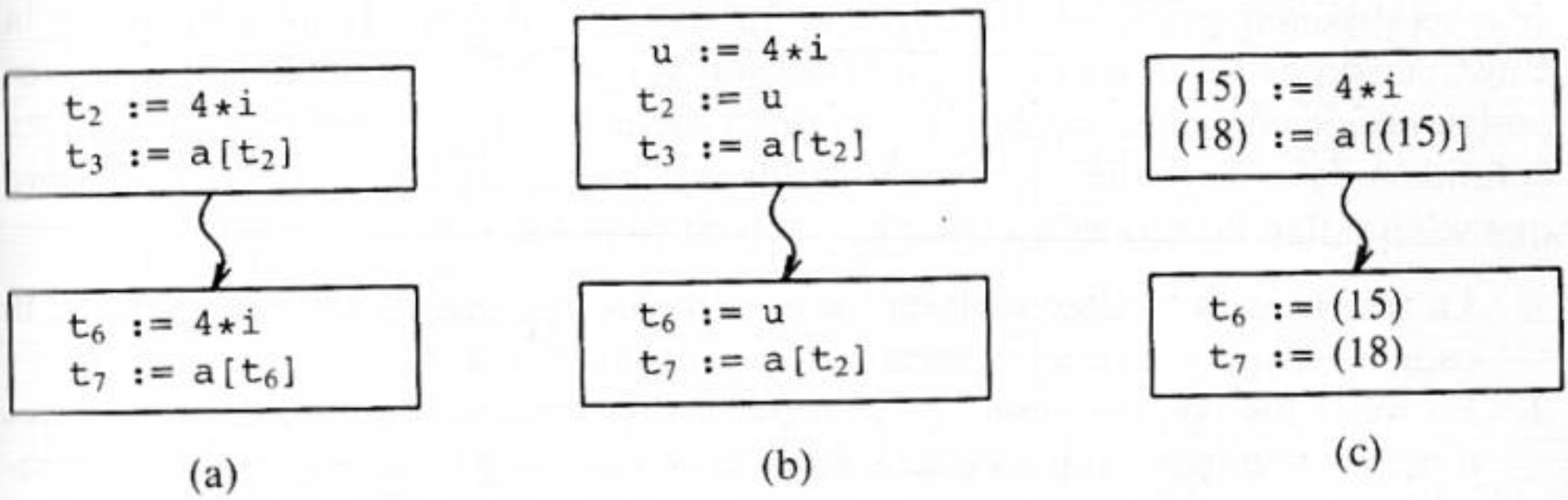


Fig. 10.34. Eliminación de la subexpresión común `4*i`.

La subexpresión común `4*i` de la figura 10.34(a) se ha eliminado en la figura 10.34(b). Una forma de determinar que `a[t2]` y `a[t6]` también son subexpresiones comunes es sustituir `t2` y `t6` por `u` utilizando propagación de copias (que se estudiará a continuación); ambas expresiones se convierten entonces en `a[u]`, que se puede eliminar reaplicando el algoritmo 10.5. Obsérvese que la misma variable nueva `u` se inserta en ambos bloques de la figura 10.34(b), así que basta con la propagación de copias local para convertir `a[t2]` y `a[t6]` en `a[u]`.

Hay otra forma que tiene en cuenta que es el compilador el que inserta las variables temporales y sólo se utilizan dentro de los bloques en los que aparecen. Se considerará más detalladamente la forma en que se representan las expresiones durante el cálculo de las expresiones disponibles para comprender el hecho de que distintas variables temporales pueden representar la misma expresión. La técnica recomendada para representar conjuntos de expresiones es asignar un número a cada expresión y utilizar vectores de bits donde el bit *i* represente a la expresión con número *i*. Las técnicas para numeración de valores de la sección 5.2 se pueden aplicar durante la numeración de expresiones para tratar las variables temporales de forma especial.

Más concretamente, supóngase que `4*i` tiene el número de valor 15. Las expresiones `a[t2]` y `a[t6]` tendrán el mismo número de valor si se utiliza el número de valor 15 en lugar de los nombres de las variables temporales `t2` y `t6`. Supóngase que el número de valor resultante es 18. Entonces el bit 18 representará tanto `a[t2]` como `a[t6]` durante el análisis del flujo de datos y se puede determinar que `a[t6]` está disponible y puede ser eliminado. El código resultante se indica en la figura 10.34(c). Se utilizan (15) y (18) para representar los temporales correspondientes a las expresiones con dichos valores. En realidad, `t6` es inútil y se podría eliminar durante el

análisis de variables activas locales. Asimismo t_7, siendo un temporal, no se calcularía en sí mismo; al contrario, los usos de t_7 se sustituirían por usos de (18). □

Propagación de copias

El algoritmo 10.5 recién estudiado, y varios otros algoritmos como el de la eliminación de variables de inducción que se presenta más adelante en esta sección, introducen proposiciones de copia de la forma `x:=y`. Las copias también pueden ser generadas directamente por el generador de código intermedio, aunque la mayoría de ellas suponen temporales locales a un bloque y se pueden eliminar mediante la construcción de GDA estudiada en la sección 9.8. A veces es posible eliminar proposiciones de copia *s*:`x:=y` si se determinan todos los lugares donde se utiliza esta definición de `x`. Entonces se puede sustituir `y` por `x` en todos estos lugares, siempre que se cumplan las siguientes condiciones para cada uso *u* de `x`.

1. La proposición *s* debe ser la única definición de `x` que alcance *u* (es decir, la cadena de uso y definición para el uso *u* consta sólo de *s*).
2. En todos los caminos desde *s* a *u*, incluidos los caminos que pasan varias veces a través de *u* (pero que no pasan a través de *s* una segunda vez), no hay asignaciones a `y`.

La condición 1 se puede comprobar utilizando la información del encadenamiento de uso y definición, pero ¿qué hay de la condición 2? Se establecerá un nuevo problema de análisis de flujo de datos en el que *ent* [*B*] es el conjunto de copias *s*:`x:=y` tal que todo camino desde el nodo inicial hasta el comienzo de *B* contiene la proposición *s*, y después de la última ocurrencia de *s* no hay asignaciones a `y`. El conjunto *sal* [*B*] se puede definir en consonancia, pero con respecto al final de *B*. Se dice que la proposición de copia *s*:`x:=y` se *genera* en el bloque *B* si *s* ocurre en *B* y no hay asignaciones posteriores a `y` dentro de *B*. Se dice que *s* : `x:=y` se *desactiva* en *B* si se hace una asignación a `x` o `y` allí y *s* no está en *B*. La noción de que las asignaciones a `x` desactivan `x:=y` también apareció en las definiciones de alcance, pero la idea de que las asignaciones a `y` también lo hacen es exclusivo de este problema. Obsérvese la importante consecuencia de que distintas asignaciones `x:=y` se desactivan una a la otra; *ent* [*B*] puede contener sólo una proposición de copia con `x` en el lado izquierdo.

Sea *U* el conjunto universal de todas las proposiciones de copia en el programa. Es importante observar que distintas proposiciones `x:=y` son distintas en *U*. Se define *c_gen* [*B*] como el conjunto de todas las copias generadas en el bloque *B* y *c_desact* [*B*] como el conjunto de copias en *U* que se desactivan en *B*. Entonces, las siguientes ecuaciones relacionan las cantidades definidas:

$$\begin{aligned} sal\,[B] &= c\_gen\,[B] \cup (ent\,[B] - c\_desact\,[B]) \\ ent\,[B] &= \bigcap_{\substack{P \text{ un predecesor de } B}} sal\,[P] \text{ para } B \text{ no inicial} \\ ent\,[B_1] &= \varnothing \text{ donde } B_1 \text{ es el bloque inicial} \end{aligned} \tag{10.12}$$

Las ecuaciones 10.12 son idénticas a las ecuaciones 10.10, si *c_desact* se sustituye por *e_desact* y *c_gen* por *e_gen*. Por tanto, 10.12 se puede resolver mediante

el algoritmo 10.3, y no se analizará más este tema. Sin embargo, se dará un ejemplo que muestra algunas de las sutilezas de la optimación de copias.

Ejemplo 10.19. Considérese el grafo de flujo de la figura 10.35. Aquí, *c_gen* $[B_1]$ $= \{\texttt{x:=y}\}$ y *c_gen* $[B_3] = \{\texttt{x:=z}\}$. También, *c_desact* $[B_2] = \{\texttt{x:=y}\}$, puesto que y es asignada en B_2. Por último, *c_desact* $[B_1] = \{\texttt{x:=z}\}$ puesto que x es asignada en B_1, y *c_desact* $[B_3] = \{\texttt{x:=y}\}$ por la misma razón.

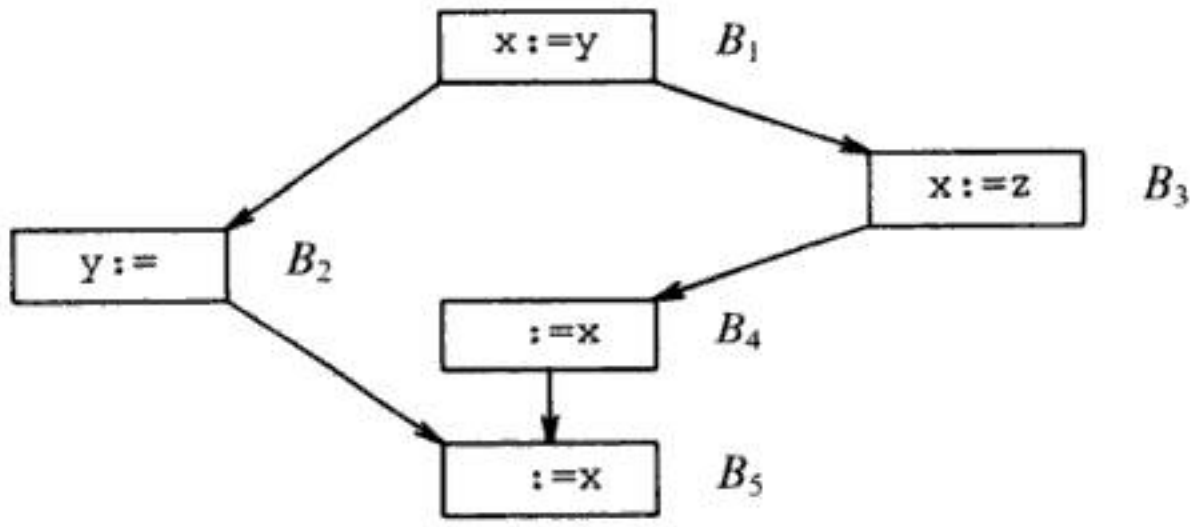


Fig. 10.35. Grafo de flujo del ejemplo 10.19.

Los otros conjuntos *c_gen* y *c_desact* son $\varnothing$. Asimismo, *ent* $[B_1] = \varnothing$ por las ecuaciones 10.12. El algoritmo 10.3 en una pasada determina que

$$ent\,[B_2] = ent\,[B_3] = sal\,[B_1] = \{\texttt{x:=y}\}$$

Del mismo modo, *sal* $[B_2] = \varnothing$ y

$$sal\,[B_3] = ent\,[B_4] = sal\,[B_4] = \{\texttt{x:=z}\}$$

Por último, *ent* $[B_5] = sal\,[B_2] \cap sal\,[B_4] = \varnothing$.

Se observa que ninguna de las copias, x:=y o x:=z, "alcanza" el uso de x en B_5, en el sentido del algoritmo 10.5. Es cierto pero irrelevante que ambas definiciones de x "alcanzan" B_5 en el sentido de las definiciones de alcance. Por tanto, no se puede propagar ninguna de las copias, ya que no es posible sustituir y (respectivamente z) por x en todos los usos de x que alcanza la definición x:=y (respectivamente x:=z). Se podría sustituir z por x en B_4 pero esto no mejoraría el código. □

Ahora se especifican los detalles del algoritmo para eliminar las proposiciones de copia.

Algoritmo 10.6. Propagación de copias.

Entrada. Un grafo de flujo G, con cadenas de uso y definición que den las definiciones que alcanzan el bloque B, y con *c_ent* $[B]$ que represente la solución a las ecuaciones 10.12, es decir, el conjunto de copias x:=y que alcanzan el bloque B a lo largo de todos los caminos, sin asignación a x o y después de la última ocurrencia de x:=y en el camino. También se necesitan cadenas de definición y uso que proporcionen los usos de cada definición.

Salida. Un grafo de flujo revisado.

Método. Para cada copia *s*:x:=y, hágase lo siguiente:

1. Determínense aquellos usos de x alcanzados por esta definición de x, a saber, *s*:x:=y.
2. Determínese si para todo uso de x que se encuentre en 1, *s* está en *c_ent* [*B*], donde *B* es el bloque de este uso particular, y, además, ninguna definición de x o y ocurre antes de este uso de x dentro de *B*. Recuérdese que si *s* está en *c_ent* [*B*], entonces *s* es la única definición de x que alcanza *B*.
3. Si *s* cumple la condición de 2, entonces elimínese *s* y sustitúyanse todos los usos de x encontrados por y en 1. □

Detección de cálculos invariantes de lazos

Se utilizarán las cadenas de uso y definición para detectar aquellos cálculos dentro de un lazo que son *invariantes del lazo*, es decir, cuyos valores no cambian mientras el control permanece dentro del lazo. Como se estudió en la sección 10.4, un lazo es una región que consta de un conjunto de bloques con un encabezamiento que domina a todos los otros bloques, así que la única forma de entrar al lazo es a través del encabezamiento. También es necesario que un lazo tenga al menos una forma de regresar al encabezamiento desde cualquier bloque dentro del lazo.

Si una asignación x:=y+z está en una posición en el lazo donde todas las definiciones posibles de y y z están fuera del lazo (incluido el caso especial en que y, z o ambas son constantes), entonces y+z es un invariante del lazo porque su valor será el mismo cada vez que se encuentre x:=y+z siempre que el control permanezca dentro del lazo. Todas estas asignaciones se pueden detectar a partir de las cadenas de uso y definición, es decir, una lista de todos los puntos de definición de y y z que alcancen la asignación x:=y+z.

Habiendo reconocido que el valor de x calculado en x:=y+z no cambia dentro del lazo, supóngase que hay otra proposición v:=x+w, donde w sólo podría haberse definido fuera del lazo. Entonces x+w también es un invariante del lazo.

Se pueden utilizar estas ideas para hacer repetidas pasadas por el lazo, descubriendo más y más cálculos cuyos valores sean invariantes del lazo. Si se tiene ambas cadenas de uso y definición y de definición y uso, ni siquiera hay que hacer varias pasadas por el código. La cadena de definición y uso para la definición x:=y+z indicará dónde podría usarse este valor de x, y sólo hay que comprobar entre estos usos de x, dentro del lazo, que no usen otra definición de x. Estas asignaciones invariantes del lazo se pueden trasladar al preencabezamiento, siempre que sus operandos además de x también sean invariantes del lazo, como se estudia en el siguiente algoritmo.

Algoritmo 10.7. Detección de cálculos invariantes de lazos.

Entrada. Un lazo *L* formado por un conjunto de bloques básicos, donde cada bloque contenga una secuencia de proposiciones de tres direcciones. Se supone que las cadenas de uso y definición para las proposiciones individuales están disponibles como se les calcula en la sección 10.5.

Salida. El conjunto de proposiciones de tres direcciones que calculan el mismo valor cada vez que se ejecutan, desde que el control entra al lazo L hasta el momento en que el control sale de L.

Método. Se dará una especificación más bien informal del algoritmo, esperando que los principios resulten claros.

1. Márquese "invariante" aquellas proposiciones cuyos operandos sean todos constantes o tengan sus definiciones de alcance fuera de L.
2. Repítase el paso 3 hasta que en alguna repetición no se marquen nuevas proposiciones como "invariante".
3. Márquense "invariante" todas aquellas proposiciones que no hayan sido así marcadas previamente cuyos operandos sean constantes, tengan todas sus definiciones de alcance fuera de L, o tengan exactamente una definición de alcance, y que esa definición sea una proposición dentro de L marcada como invariante. □

Realización de traslado de código

Habiendo encontrado las proposiciones invariantes dentro de un lazo, se puede aplicar a algunas de ellas una optimación conocida como *traslado de código*, en la que las proposiciones se trasladan al preencabezamiento del lazo. Las tres siguientes condiciones garantizan que el traslado de código no modifique la función del programa. Ninguna de las condiciones es absolutamente fundamental; se han seleccionado estas condiciones porque son fáciles de comprobar y porque se aplican a situaciones que ocurren en programas reales. Más adelante se analizará la posibilidad de relajar las condiciones.

Las condiciones para la proposición *s*:`x:=y+z` son:

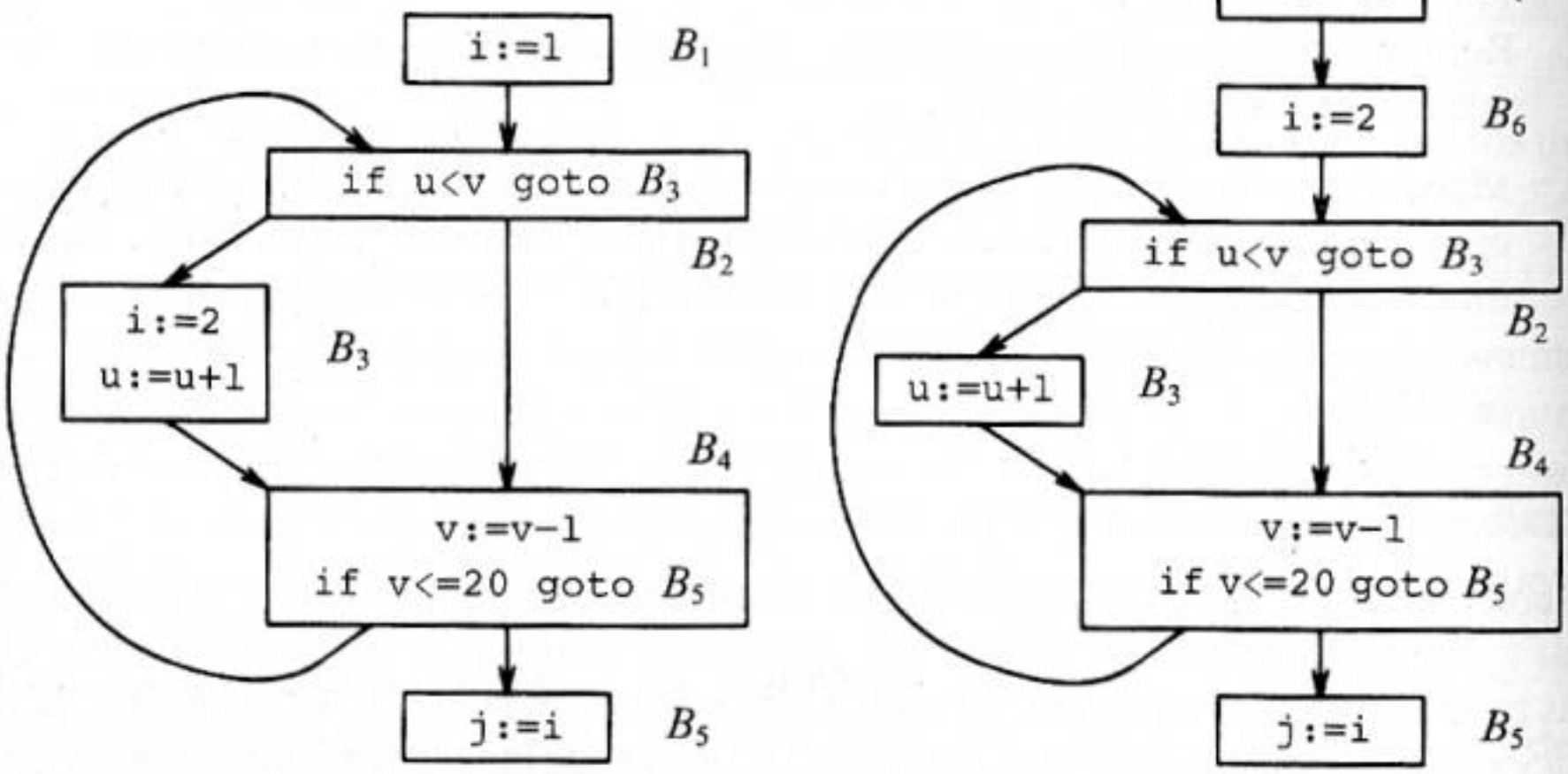
1. El bloque que contenga *s* domina todos los nodos *salida* del lazo, donde una salida de un lazo es un nodo con un sucesor que no esté en el lazo.
2. No hay otra proposición en el lazo que haga una asignación a `x`. De nuevo, si `x` es un temporal asignado sólo una vez, esta condición siempre se cumple y no necesita comprobación.
3. Ningún uso de `x` en el lazo es alcanzado por ninguna definición de `x` que no sea *s*. Esta condición también se cumplirá, generalmente, si `x` es un temporal.

Los siguientes tres ejemplos motivan las condiciones anteriores.

Ejemplo 10.20. Trasladar una proposición que no necesita ejecutarse dentro de un lazo a una posición fuera del lazo puede cambiar la función que realiza el programa, como se demuestra utilizando la figura 10.36. Esta observación motiva la condición 1, ya que una proposición que domina todas las salidas no puede dejar de ejecutarse, suponiendo que el lazo no funciona indefinidamente.

Considérese el grafo de flujo que se muestra en la figura 10.36(a). B_2, B_3 y B_4 forman un lazo con encabezamiento B_2. La proposición `i:=2` en B_3 es evidentemente un invariante del lazo. Sin embargo, B_3 no domina a B_4, la única salida del

lazo. Si se traslada `i:=2` a un preencabezamiento recién creado B_6, como se muestra en la figura 10.36(b), se puede cambiar el valor asignado a `j` en B_5, en casos en que nunca se ejecuta B_3. Por ejemplo, si `u=30` y `v=25` cuando se entra por primera vez a B_2, la figura 10.36(a) asigna `1` a `j` en B_5, ya que nunca se entra a B_3, en tanto que la figura 10.36(b) asigna `2` a `j`. □



(a) Antes (b) Después

Fig. 10.36. Ejemplo de traslado ilegal de código.

Ejemplo 10.21. La condición 2 es necesaria cuando hay más de una asignación a `x` en el lazo. Por ejemplo, la estructura del grafo de flujo de la figura 10.37 es la misma que la de la figura 10.36(a), y existe la posibilidad de crear un preencabezamiento B_6 como en la figura 10.36(b).

Como B_2 en la figura 10.37 domina la salida B_4, la condición 1 no impide que `i:=3` se traslade al preencabezamiento B_6. Sin embargo, si se hace así, se igualará `i` a `2` siempre que se ejecute B_3, y cuando se llegue a B_5 `i` tendrá el valor `2`, aunque se siga una secuencia como $B_2 \rightarrow B_3 \rightarrow B_4 \rightarrow B_2 \rightarrow B_4 \rightarrow B_5$. Por ejemplo, considérese lo que ocurre si `v` es `22` y `u` es `21` cuando se llega por primera vez a B_2. Si `i:=3` está en B_2, se iguala `j` a `3` en B_5, pero si `i:=3` se traslada al preencabezamiento se iguala `j` a `2`. □

Ejemplo 10.22. Ahora se considerará la regla 3. El uso `k:=i` en el bloque B_4 de la figura 10.38 es alcanzado por `i:=1` en el bloque B_1, así como por `i:=2` en B_3. Por tanto, no se podría trasladar `i:=2` al preencabezamiento, porque el valor de `k` que alcanza B_5 cambiaría en el caso `u>=v`. Por ejemplo, si `u=v=0` entonces `k` se iguala a `1` en el grafo de flujo de la figura 10.38, pero si `i:=2` se traslada al preencabezamiento, `k` se hace igual a `2` una vez y para siempre. □

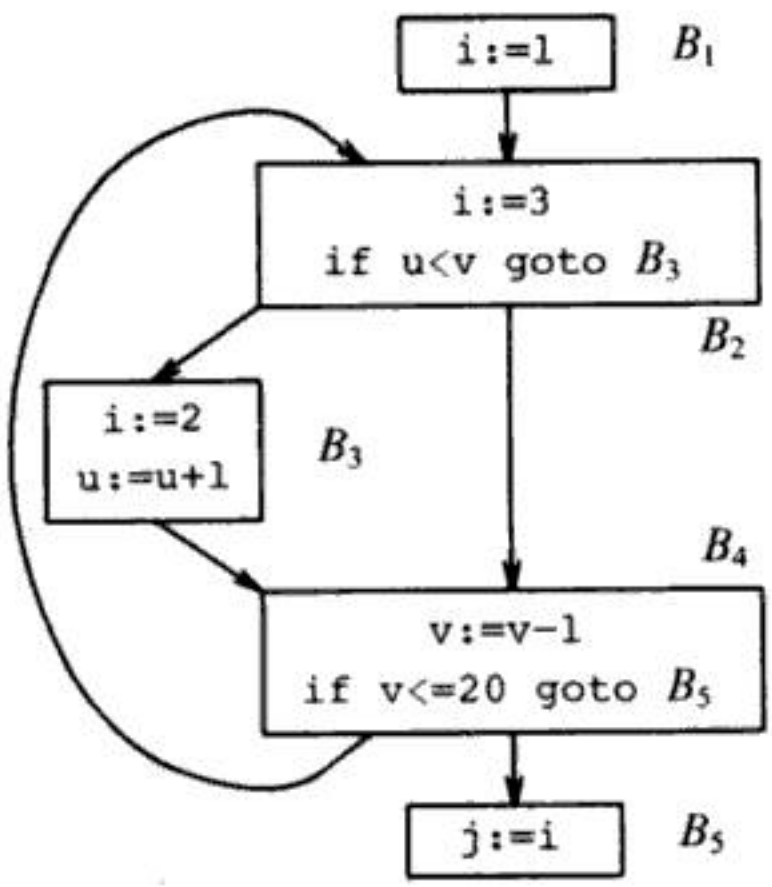


Fig. 10.37. Condición 2.

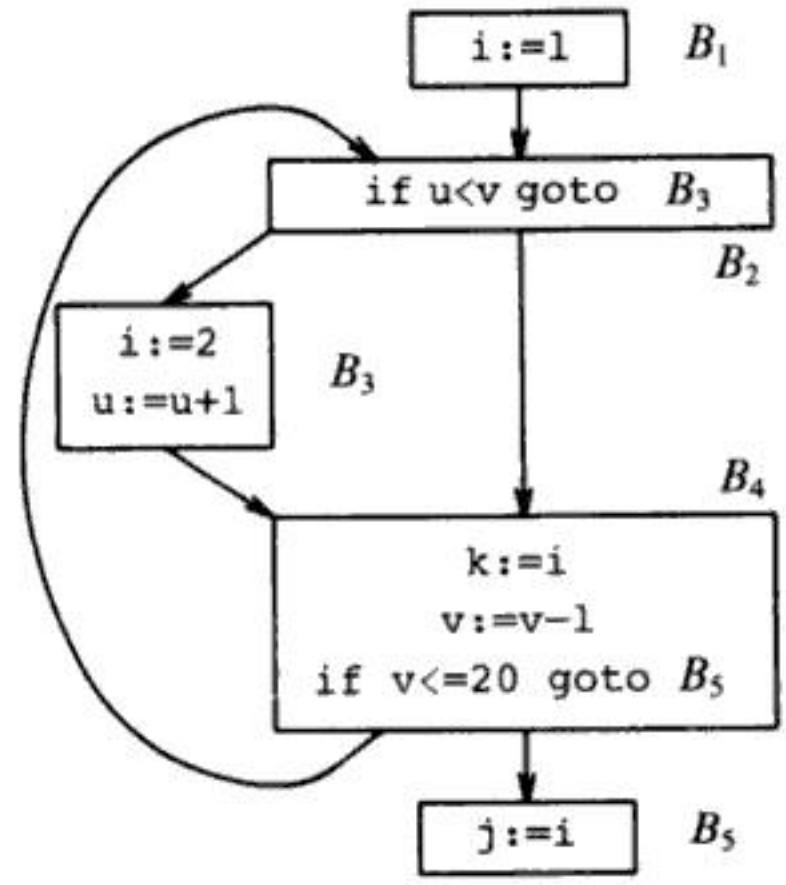


Fig. 10.38. Condición 3.

Algoritmo 10.8. Traslado de código.

Entrada. Un lazo L con información sobre el encabezamiento de uso y definición y sobre los dominantes.

Salida. Una versión revisada del lazo con un preencabezamiento y (posiblemente) algunas proposiciones trasladadas al preencabezamiento.

Método.

1. Utilícese el algoritmo 10.7 para encontrar las proposiciones invariantes del lazo.
2. Para cada proposición s que defina x encontrada en el paso 1, verifíquese:
   i) que esté en un bloque que domine todas las salidas de L,
   ii) que x no se defina en otra parte de L, y
   iii) que todos los usos en L de x sólo puedan ser alcanzados por la definición de x en la proposición s.
3. Trasládese, en el orden encontrado por el algoritmo 10.7, cada proposición s encontrada en 1 y que haya cumplido las condiciones 2i), 2ii) y 2iii), a un preencabezamiento nuevo recién creado, siempre que las proposiciones de definición de los operandos de s que estén definidos en la lazo L (en caso de que s haya sido encontrado en el paso 3 del Algoritmo 10.7) hayan sido trasladadas previamente al preencabezamiento. □

Para comprender por qué no puede ocurrir ningún cambio en la función que realiza el programa, las condiciones 2i) y 2ii) del algoritmo 10.8 garantizan que el valor de x calculado en s debe ser el valor de x después de cualquier bloque de salida de L. Cuando s se traslada al preencabezamiento, s seguirá siendo la definición de x que alcance el final de cualquier bloque de salida de L. La condición 2iii) garantiza que los usos de x dentro de L utilizaron y utilizarán el valor de x calculado por s.

Para comprobar por qué la transformación no puede aumentar el tiempo de ejecución del programa, obsérvese que la condición 2i) garantiza que *s* se ejecute al menos una vez cada vez que el control entra a *L*. Después del traslado del código, se ejecutará exactamente una vez en el preencabezamiento y ninguna en *L* cada vez que el control entre a *L*.

Estrategias alternativas para el traslado de código

Se puede relajar un poco la condición 1 si quiere correr el riesgo de incrementar un poco el tiempo de ejecución de un programa. Por supuesto, no se modifica el cálculo del programa. La versión relajada de la condición 1 para el traslado de código [es decir, el elemento 2i) del Algoritmo 10.8] consiste en que se pueda trasladar una proposición *s* que asigne x sólo si:

1. El bloque que contenga *s* domina todas las salidas del lazo, o x no se utiliza fuera del lazo. Por ejemplo, si x es una variable temporal, se puede estar seguro (en muchos compiladores) de que el valor se utilizará sólo en su propio bloque. En general, se necesita el análisis de variables activas para saber si x está activa en cualquier salida del lazo.

Si se modifica el algoritmo 10.8 para que utilice la condición 1′, ocasionalmente el tiempo de ejecución se incrementará ligeramente, pero en general no habrá por qué preocuparse. El algoritmo modificado puede trasladar al preencabezamiento algunos cálculos que podrían no ejecutarse en lazo. Este riesgo ralentiza de manera significativa el programa, y también puede ocasionar un error en algunas circunstancias. Por ejemplo, la evaluación de una división `x/y` en un lazo puede estar precedida de una prueba para ver si `y=0`. Si se traslada `x/y` al preencabezamiento, puede ocurrir una división por 0. Por esta razón, no conviene utilizar la condición 1′ a menos que la optimación pueda ser inhibida por el programador o que se aplique la condición 1 más estricta para las proposiciones de división.

Aunque la asignación `x:=y+z` nunca cumpla ninguna de las condiciones 2i), 2ii) y 2iii) del algoritmo 10.8, todavía se puede sacar el cálculo `y+z` del ciclo. Se crea un nuevo temporal `t`, y se coloca `t:=y+z` en el preencabezamiento. Después se sustituye `x:=y+z` por `x:=t` en el lazo. En muchos casos se puede entonces propagar la proposición de copia `x:=t` como se estudió antes en esta sección. Obsérvese que si se cumple la condición 2iii) del algoritmo 10.8, es decir, si todos los usos de x en el lazo *L* se definen en `x:=y+z` (ahora `x:=t`), entonces se puede eliminar la proposición `x:=t` sustituyendo los usos de x en *L* por usos de `t` y colocando `x:=t` después de cada salida del lazo.

Mantenimiento de la información del flujo de datos después del traslado de código

Las transformaciones del algoritmo 10.8 no alteran la información del encadenamiento de uso y definición, ya que según las condiciones 2i), 2ii) y 2iii), todos los usos de la variable asignada por una proposición trasladada *s* que fuera alcanzada por *s* sigue siendo alcanzada por *s* desde su nueva posición. Las definiciones de las variables utilizadas por *s* están fuera de *L*, en cuyo caso alcanzan al preencabezamiento, o están dentro de *L*, en cuyo caso por el paso 3 fueron trasladadas al preencabezamiento por delante de *s*.

Si se representan las cadenas de uso y definición mediante listas de apuntadores a apuntadores a proposiciones (en lugar de por listas de apuntadores a proposiciones), se pueden mantener las cadenas de uso y definición cuando se traslade la proposición *s*, cambiando simplemente el apuntador a *s* cuando se traslade. Es decir, para cada proposición *s* se crea un apuntador p_s, que siempre apunte a *s*. Se sitúa p_s en cada cadena de uso y definición que contenga *s*. Entonces, independientemente de adonde se traslade *s*, sólo hay que modificar p_s, sin tener en cuenta cuántas cadenas de uso y definición encuentre *s*. Por supuesto, el nivel adicional de indirecciones emplea algo de tiempo y espacio del compilador.

Si se representan las cadenas de uso y definición mediante una lista de direcciones de proposiciones (apuntadores a proposiciones) todavía se pueden mantener las cadenas de uso y definición cuando se trasladen las proposiciones. Pero entonces se necesitan asimismo las cadenas de definición y uso, para una mayor eficiencia. Cuando se traslada *s*, se puede recorrer su cadena de definición y uso, modificando la cadena de uso y definición en todos los usos que se refieran a *s*.

La información sobre dominación viene modificada ligeramente por el traslado de código. El preencabezamiento es ahora el dominador inmediato del encabezamiento y el dominador inmediato del preencabezamiento es el nodo que antes era el dominador inmediato del encabezamiento. Es decir, el preencabezamiento se inserta en el árbol de dominación como padre del encabezamiento.

Eliminación de variables de inducción

Una variable `x` se denomina *variable de inducción* de un lazo *L* si cada vez que la variable `x` cambie de valor, se incrementa o decrementa por una constante. A menudo, una variable de inducción se incrementa por la misma constante cada vez que se itera el lazo, como `i` en un lazo encabezado por **for** `i` := 1 **to** 10. Sin embargo, los métodos aquí considerados se refieren a variables que se incrementan o decrementan cero, una, dos o más veces cuando se itera un lazo. El número de modificaciones en una variable de inducción puede diferir incluso en distintas iteraciones.

Una situación habitual es cuando una variable de inducción, por ejemplo, `i`, es el índice de una matriz, y alguna otra variable de inducción, por ejemplo, `t`, cuyo valor es una función lineal de `i`, es el desplazamiento real utilizado para acceder a la matriz. A menudo, el único uso que se hace de `i` es para comprobar la terminación del lazo. Entonces se puede prescindir de `i` sustituyendo su prueba por una sobre `t`.

Los siguientes algoritmos se refieren a una clase limitada de variables de inducción para simplificar la presentación. Se pueden ampliar los algoritmos añadiendo más casos, pero otras exigen que se demuestren los teoremas sobre expresiones con los habituales operadores aritméticos.

Se buscarán *variables básicas de inducción*, que son aquellas variables `i` cuyas únicas asignaciones dentro del lazo *L* son de la forma `i:=i` $\pm c$, donde *c* es una constante[7]. Después se buscan variables de inducción adicionales `j` que estén defi-

---

[7] En este análisis de las variables de inducción, "+" representa sólo el operador de adición, no un operador genérico, y lo mismo los otros operadores aritméticos estándar.

nidas sólo una vez dentro de L, y cuyo valor sea una función lineal de una variable básica de inducción `i` donde se defina `j`.

Algoritmo 10.9. Detección de variables de inducción.

Entrada. Un lazo L con información sobre las definiciones de alcance e información sobre los cálculos invariantes del lazo (del Algoritmo 10.7).

Salida. Un conjunto de variables de inducción. Asociado a cada variable de inducción `j` hay un triple (`i`, c, d), donde `i` es una variable básica de inducción, y c y d son constantes tales que el valor de `j` viene dado por c`*i+`d en el punto donde se define `j`. Se dice que `j` pertenece a la *familia* de `i`. La variable básica de inducción `i` pertenece a su propia familia.

Método

1. Encuéntrense todas las variables básicas de inducción examinando las proposiciones de L. Aquí se utiliza la información sobre cálculos invariantes del lazo. Asociado con cada variable básica de inducción está el triple (`i`, 1, 0).

2. Búsquense las variables `k` con una sola asignación a `k` dentro de L y que tiene una de las siguientes formas:

   `k:=j*`b, `k:=`b`*j`, `k:=j/`b, `k:=j`$\pm b$, `k:=b`$\pm$`j`

   donde b es una constante, y `j` es una variable de inducción, básica o de otro tipo.

   Si `j` es básica, entonces `k` está en la familia de `j`. El triple correspondiente a `k` depende de la instrucción que la defina. Por ejemplo, si `k` está definida por `k:=j*`b, entonces el triple para `k` es (`j`, b, 0). Los triples para los casos restantes se pueden determinar de manera similar.

   Si `j` no es básica, supóngase que `j` está en la familia de `i`. Entonces los requisitos adicionales son que

   (a) no haya asignación a `i` entre el único punto de asignación a `j` en L y la asignación a `k`, y

   (b) ninguna definición de `j` fuera de L alcance `k`.

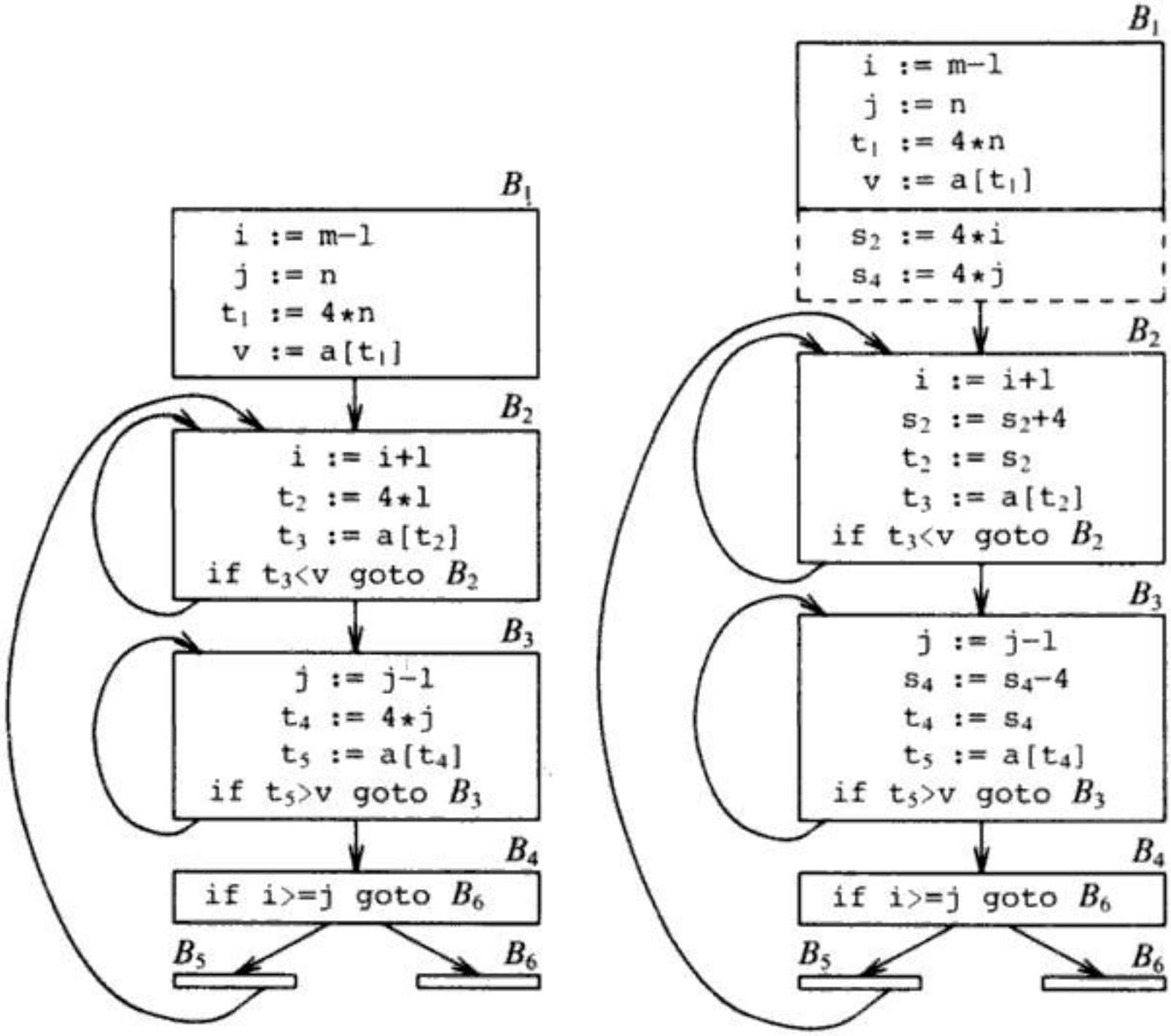
   El caso habitual será que las definiciones de `k` y `j` estén en temporales en el mismo bloque, en cuyo caso es fácil de comprobar. En general, la información sobre las definiciones de alcance proporcionará las comprobaciones necesarias si se analiza el grafo de flujo del lazo L para determinar los bloques (y por tanto las definiciones) que estén en los caminos entre la asignación a `j` y la asignación a `k`.

   El triple para `k` se calcula a partir del triple (`i`, c, d) para `j` y la instrucción que defina `k`. Por ejemplo, la definición `k:=`b`*j` conduce a (`i`, $b*c$, $b*d$) para `k`. Obsérvese que las multiplicaciones en $b*c$ y $b*d$ se pueden realizar a medida que avanza el análisis porque b, c y d son constantes. □

Una vez encontradas las familias de las variables de inducción, se modifican las instrucciones que calculan una variable de inducción para que utilice sumas o restos

en lugar de multiplicaciones. La sustitución de una instrucción más cara por otra más barata se llama *reducción de intensidad.*

Ejemplo 10.23. El lazo formado por el bloque B_2 de la figura 10.39(a) tiene la variable básica de inducción `i` porque la única asignación a `i` en el lazo incrementa su valor en `1`. La familia de `i` contiene t_2 porque hay una sola asignación a t_2, con lado derecho `4*i`. Por tanto, el triple para t_2 es (`i`,4,0). De manera similar, `j` es la única variable básica de inducción en el lazo que consta de B_3 y t_4, con triple (`j`,4,0), está en la familia de `j`.



(a) Antes

(b) Después

Fig. 10.39. Reducción de intensidad.

También se pueden buscar variables de inducción en el lazo exterior con encabezamiento B_2 y bloques B_2, B_3, B_4, B_5. Tanto `i` como `j` son variables básicas de inducción en este lazo más grande. De nuevo, t_2 y t_4 son variables de inducción con triples (`i`,4,0) y (`j`,4,0), respectivamente.

El grafo de flujo de la figura 10.39(b) se obtiene del de la figura 10.39(a) aplicando el siguiente algoritmo. Más adelante se analizará esta transformación. □

Algoritmo 10.10. Reducción de intensidad aplicada a variables de inducción.

Entrada. Un lazo L con información sobre las definiciones de alcance y las familias de inducción calculadas por medio del algoritmo 10.9.

Salida. Un lazo revisado.

Método. Considérese cada variable básica de inducción `i` por turno. Para cada variable de inducción `j` en la familia de `i` con triple (`i`, c, d):

1. Créese una nueva variable `s` (pero si dos variables j_1 y j_2 tienen los mismos triples, créese sólo una nueva variable para ambas).
2. Sustitúyanse las asignaciones a `j` por `j:=s`.
3. Inmediatamente después de cada asignación `i:=i+`n en L, donde n es una constante, añádase

```
s := s + c*n
```

donde la expresión $c*n$ da como resultado una constante porque c y n son constantes. Colóquese `s` en la familia de `i`, con triple (`i`,c,d).

4. Queda por asegurarse que `s` se inicialice con $c*$`i`$+d$ al entrar al lazo. La inicialización se puede colocar al final del preencabezamiento. La inicialización consta de

```
s := c*i          /* sólo s:=i si c es 1 */
s := s+d              /* omítase si d es 0 */
```

Obsérvese que `s` es una variable de inducción en la familia de `i`. □

Ejemplo 10.24. Supóngase que se consideran los lazos de la figura 10.39(a) de dentro hacia fuera. Como el tratamiento de los lazos internos que contienen B_2 y B_3 es muy similar, sólo se hablará del lazo que rodea a B_3. En el ejemplo 10.23 se observó que la variable básica de inducción en el lazo que rodea a B_3 es `j` y que la otra variable de inducción es t_4 con triple (`j`,4,0). En el paso 1 del algoritmo 10.10, se construye una nueva variable s_4. En el paso 2, la asignación t_4`:=4*j` se sustituye por t_4`:=`s_4. El paso 4 inserta la asignación s_4`:=`s_4`-4` después de la asignación `j:=j-1`, donde `-4` se obtiene multiplicando el `-1` en la asignación de `j` y el `4` en el triple (`j`,4,0) para t_4.

Como B_1 sirve de preencabezamiento para el lazo, se puede poner la inicialización de s_4 al final del bloque B_1 que contiene la definición de `j`. Las instrucciones añadidas se muestran en la ampliación punteada al bloque B_1.

Cuando se considera el lazo externo, el grafo de flujo resulta como en la figura 10.39(b). Hay cuatro variables, `i`, s_2, `j` y s_4, que se podrían considerar variables de inducción. Sin embargo, el paso 3 del algoritmo 10.10 coloca las variables recién creadas en las familias de `i` y `j`, respectivamente, para facilitar la eliminación de `i` y `j`, utilizando el siguiente algoritmo. □

Después de la reducción de intensidad se ve que el único uso de algunas variables de inducción es para realizar pruebas. Se puede sustituir una prueba de una variable de inducción de ese tipo por la de otra. Por ejemplo, si `i` y `t` son variables de

inducción tales que el valor de `t` siempre sea cuatro veces el valor de `i`, entonces la prueba `i>=j` es equivalente a `t>=4*j`. Después de esta sustitución se puede eliminar `i`. Obsérvese, sin embargo, que si `t=-4*i`, entonces también hay que cambiar el operador relacional porque `i>=j` es equivalente a `t<=-4*j`. En el siguiente algoritmo se considera el caso en que la constante multiplicativa es positiva, dejando como ejercicio la generalización a constantes negativas.

Algoritmo 10.11. Eliminación de variables de inducción.

Entrada. Un lazo L con información sobre las definiciones de alcance, sobre cálculos invariantes del lazo (del Algoritmo 10.7) y sobre variables activas.

Salida. Un lazo revisado.

Método.

1. Considérese cada variable de inducción `i` cuyos únicos usos sean calcular otras variables de inducción en su familia y en saltos condicionales. Tómese una `j` dentro de la familia de `i`, preferiblemente una tal que c y d en su triple (`i`, c, d) sean tan simples como sea posible (es decir, se prefiere $c=1$ y $d=0$), y se modifica cada comprobación en que aparezca `i` para utilizar `j` en su lugar. Se supondrá que c es positiva. Una prueba de la forma `if i` **oprel** `x goto` B, donde `x` es una variable de inducción, se sustituye por

```
r := c*x        /* r := x si c es 1 */
r := r+d        /* se omite si d es 0 */
if j oprel r goto B
```

   donde `r` es un temporal nuevo. El caso `if x` **oprel** `i goto` B se trata de forma análoga. Si hay dos variables de inducción i_1 e i_2 en la prueba `if` i_1 **oprel** i_2 `goto` B, entonces se comprueba si se pueden sustituir las dos i_1 e i_2. El caso más fácil es cuando se tiene j_1 con triple (i_1, c_1, d_1) y j_2 con triple (i_2, c_2, d_2), y $c_1 = c_2$ y $d_1 = d_2$. Entonces, i_1 **oprel** i_2 es equivalente a j_1 **oprel** j_2. En casos más complejos, puede que no valga la pena sustituir la prueba, porque quizás haya que introducir dos pasos multiplicativos y una suma, en tanto que sólo se podrían ahorrar dos pasos eliminando i_1 e i_2.

   Por último, bórrense todas las asignaciones a las variables de inducción eliminadas del lazo L, porque ahora ya no se utilizarán.

2. Considérese a continuación cada variable de inducción `j` para la cual el algoritmo 10.10 introdujo una proposición `j:=s`. Compruébese primero que no puede haber asignaciones a `s` entre la proposición introducida `j:=s` y cualquier uso de `j`. En la situación habitual, `j` se utiliza en el bloque en el que se define, simplificando esta comprobación; en caso contrario, se necesita la información sobre las definiciones de alcance, más un análisis de grafos para implantar esta comprobación. Después, sustitúyanse todos los usos de `j` por usos de `s` y bórrese la proposición `j:=s`. □

Ejemplo 10.25. Considérese el grafo de flujo de la figura 10.39(b). El lazo interno que rodea a B_2 contiene dos variables de inducción, `i` y s_2, pero no se pueden eli-

minar porque s_2 se utiliza como índice para la matriz a, e i se utiliza en una prueba fuera del lazo. De manera similar, el lazo alrededor de B_3 contiene las variables de inducción j y s_4, pero no se puede eliminar ninguna.

Se aplicará el algoritmo 10.11 al lazo externo. Cuando el algoritmo 10.10 creó las variables nuevas s_2 y s_4, como se analizó en el ejemplo 10.24, s_2 se situó en la familia de i y s_4 en la familia de j. Considérese la familia de i. El único uso de i es en la prueba de terminación del lazo en el bloque B_4, de modo que i es candidato a ser eliminado en el paso 1 del algoritmo 10.11. La prueba en el bloque B_4 incluye las dos variables de inducción i y j. Por fortuna, las familias de i y j contienen s_2 y s_4 con las mismas constantes en sus triples, porque los triples son (i,4,0) y (j,4,0), respectivamente. Por tanto, la prueba i>=j se puede sustituir por s_2>=s_4, permitiendo que se eliminen i y j.

El paso 2 del algoritmo 10.11 aplica la propagación de copias a las variables recién creadas, sustituyendo t_2 y t_4 por s_2 y s_4, respectivamente. □

Variables de inducción con expresiones invariantes de lazos

En los algoritmos 10.9 y 10.10 se pueden permitir expresiones invariantes de lazos en lugar de constantes. Sin embargo, el triple (i,*c*,*d*) para una variable de inducción j puede entonces contener expresiones invariantes de lazos en lugar de expresiones. La evaluación de estas expresiones se debe realizar fuera del lazo *L*, en el preencabezamiento. Además, como el código intermedio exige que haya a lo sumo un operador por instrucción, hay que estar preparado para generar proposiciones en código intermedio para la evaluación de expresiones. La sustitución de pruebas en el algoritmo 10.11 exige que se conozca el signo de la constante multiplicativa *c*. Por este motivo es razonable concentrarse en los casos en que *c* sea una constante conocida.

10.8 TRATAMIENTO CON SINONIMOS (ALIAS)

Si dos o más expresiones denotan la misma dirección de memoria, se dice que las expresiones son *sinónimas* (o alias) una de otra. En esta sección se considerará el análisis del flujo de datos en presencia de apuntadores y procedimientos que introducen sinónimos.

La presencia de apuntadores complican el análisis del flujo de datos, porque no se sabe a ciencia cierta lo que se define y utiliza. Lo único que se puede suponer si no se sabe dónde puede apuntar el apuntador p es que una asignación indirecta por medio de un apuntador puede cambiar potencialmente (es decir, definir) cualquier variable. También hay que suponer que cualquier uso de los datos apuntados por un apuntador, por ejemplo, x:=*p, puede utilizar potencialmente cualquier variable. Estas suposiciones dan como resultado más variables activas y definiciones de alcance que las que hay en realidad y menos expresiones disponibles que las que hay en realidad. Por fortuna, se puede utilizar el análisis del flujo de datos para averiguar adónde puede apuntar un apuntador, pudiendo así obtener información más precisa de los otros análisis de flujo de datos.

Como en el caso de las asignaciones con variables de tipo apuntador, cuando se llega a una llamada a un procedimiento, no hay que hacer necesariamente la supo-

sición de peor caso (que se puede cambiar todo) siempre que se pueda calcular el conjunto de variables que un procedimiento podría cambiar. Al igual que con las optimaciones de código, se pueden cometer errores en el lado conservador. Es decir, los conjuntos de variables cuyos valores "pueden ser" cambiados o utilizados podrían incluir las variables que fueran cambiadas realmente o utilizadas en alguna ejecución del programa. Como de costumbre, se intentará simplemente aproximarse a los conjuntos verdaderos de variables modificadas y utilizadas sin trabajar demasiado y sin cometer un error que altere la función del programa.

Un sencillo lenguaje de apuntadores

Para una mayor especificidad, se considerará un lenguaje en el que hay tipos de datos elementales (por ejemplo, enteros y reales) que requieren una palabra cada uno, y matrices de estos tipos. También habrá apuntadores a estos elementos y a matrices, pero no a otros apuntadores. Habrá que conformarse con saber que un apuntador `p` apunta a algún lugar en la matriz `a`, sin preocuparse del elemento de `a` al que está apuntando. Es razonable agrupar todos los elementos de una matriz, por lo que se refiere a los objetivos del apuntador. Normalmente, los apuntadores se utilizarán como cursores que recorren una matriz completa, así que si se pudiera realizar un análisis de flujo más detallado, indicaría a menudo que en un punto determinado del programa, `p` podría estar apuntando a cualquiera de los elementos de `a`.

También se deben hacer ciertas suposiciones sobre qué operaciones aritméticas sobre apuntadores tienen un significado semántico. Primero, si el apuntador `p` apunta a un elemento de datos primitivo (de una palabra), entonces cualquier operación aritmética sobre `p` produce un valor que puede ser un entero, pero no un apuntador. Si `p` apunta a una matriz, entonces la suma o resta de un entero deja a `p` apuntando a un lugar de la misma matriz, en tanto que otras operaciones aritméticas sobre apuntadores producen un valor que no es un apuntador. Aunque no todos los lenguajes prohíben, por ejemplo, trasladar un apuntador desde una matriz `a` a una matriz `b` sumando al apuntador, esta acción dependería de la implantación particular para asegurarse que la matriz `b` siguiera a `a` en la memoria. El punto de vista aquí adoptado es que un compilador optimador debe tener en cuenta únicamente la definición del lenguaje para decidir las optimaciones que se van a realizar. Cada implantador del compilador, sin embargo, debe emitir un juicio sobre qué optimaciones específicas se deben permitir al compilador.

Efectos de asignaciones con apuntadores

Con estas suposiciones, las únicas variables que podrían utilizarse como apuntadores son las que se declaran como apuntadores y los temporales que reciben un valor que es un apuntador más o menos una constante. Se hará referencia a todas estas variables como apuntadores. Las reglas para determinar dónde puede apuntar un apuntador `p` son las siguientes:

1. Si hay una proposición de asignación *s*:`p:=&a`, entonces inmediatamente después de *s*, `p` sólo apunta a `a`. Si `a` es una matriz, entonces `p` puede apuntar sólo a `a` después de cualquier asignación a `p` de la forma `p:=&a` $\pm$ `c`, donde `c` es una

constante[8]. Como siempre, se considera que &a se refiere a la posición del primer elemento de la matriz a.

2. Si hay una proposición de asignación *s*:p:=q±c, donde c es un entero que no sea cero, y p y q son apuntadores, entonces inmediatamente después de *s*, p puede apuntar a cualquier matriz a la que podría apuntar q antes de *s*, pero a nada más.

3. Si hay una asignación *s*:p:=q, entonces inmediatamente después de *s*, p puede apuntar a cualquier cosa a la que pudiera apuntar q antes de *s*.

4. Después de cualquier otra asignación a p, no hay ningún objeto al que pudiera apuntar p; dicha asignación probablemente (dependiendo de la semántica del lenguaje) no tenga significado.

5. Después de cualquier asignación a una variable diferente de p, p apunta a cualquier cosa a la que apuntaba antes de la asignación. Obsérvese que esta regla supone que ningún apuntador puede apuntar a un apuntador. Relajar esta suposición no dificulta mucho las cosas y la generalización se deja al lector.

Se definirá *ent* [*B*] para el bloque *B* como la función que proporciona para cada apuntador p el conjunto de variables a las que p podría apuntar al comienzo de *B*. Formalmente, *ent* [*B*] es un conjunto de pares de la forma (p, a), donde p es un apuntador y a es una variable, lo cual significa que p podría apuntar a a. En la práctica, *ent* [*B*] se puede representar como una lista para cada apuntador, y la lista para p da el conjunto de todas las a tales que (p, a) está en *ent* [*B*]. Se define *sal* [*B*] de manera similar para el fin de *B*.

Se especifica una función de transferencia, $trans_B$ que define el efecto del bloque *B*. Es decir, $trans_B$ es una función que toma como argumento un conjunto de pares *S*, donde cada par es de la forma (p, a) siendo p un apuntador y a una variable distinta de apuntador, y produce otro conjunto *T*. Presumiblemente, el conjunto al que se aplica $trans_B$ será *ent* [*B*] y el resultado de la aplicación será *sal* [*B*]. Sólo hay que indicar cómo se calcula *trans* para proposiciones simples; $trans_B$ será entonces la composición de $trans_s$ para cada proposición *s* del bloque *B*. Las reglas para calcular *trans* son las siguientes:

1. Si *s* es p:=&a o p:=&a±c en el caso en que a sea una matriz, entonces

$$trans_s(S) = (S - \{(p,b) \mid \text{cualquier variable b}\}) \cup \{(p,a)\}$$

2. Si *s* es p:=q±c para el apuntador q y un entero no cero c, entonces

$$trans_s(S) = (S - \{(p,b) \mid \text{cualquier variable b}\}) \cup \{(p,b) \mid (q,b) \text{ está en } S \text{ y b es una variable de tipo matriz}\}$$

Obsérvese que esta regla tiene sentido incluso si p = q.

3. Si *s* es p:=q, entonces

$$trans_s(S) = (S - (S - \{(p,b) \mid \text{cualquier variable b}\})) \cup \{(p,b) \mid (q,b) \text{ está en } S\}$$

---

[8] En esta sección, + se representa a sí mismo, en lugar de a un operador genérico.

4. Si s asigna al apuntador p cualquier otra expresión, entonces

$$trans_s = S - \{(\mathrm{p},\mathrm{b}) \mid \text{cualquier variable b}\}$$

5. Si s no es una asignación a un apuntador, entonces $trans_s(S) = S$.

Ahora se pueden escribir las ecuaciones que relacionan *ent*, *sal* y *trans* como sigue:

$$\begin{aligned} sal[B] &= trans_B(ent[B]) \\ ent[B] &= \bigcup_{\substack{P \text{ un prede-} \\ \text{cesor de } B}} sal[P] \end{aligned} \tag{10.13}$$

donde si B consta de las proposiciones $s_1, s_2, \ldots, s_k$, entonces

$$trans_B(S) = trans_{s_k}(trans_{s_{k-1}}(\cdots(trans_{s_2}(trans_{s_1}(S)))\cdots)).$$

Las ecuaciones 10.13 se pueden resolver fundamentalmente como las definiciones de alcance del algoritmo 10.2. Por tanto, no se considerará detalladamente el algoritmo, sino que sólo se dará un ejemplo.

Ejemplo 10.26. Considérese el grado de flujo de la figura 10.40. Se supone que a es una matriz y que c es un entero; p y q son apuntadores. Al inicio, se iguala $ent[B_1]$ a $\varnothing$. Después, $trans_{B_1}$ tiene el efecto de eliminar los pares con primer componente q, y después añade el par (q, c). Es decir, se hace que q apunte a c. Por tanto,

$$sal[B_1] = trans_{B_1}(\varnothing) = \{(\mathrm{q}, \mathrm{c})\}$$

Entonces, $ent[B_2] = sal[B_1]$. El efecto de `p:=&c` es sustituir todos los pares con primer componente p por el par (p, c). El efecto de `q:=&(a[2])` es sustituir los pares con primer componente q por (q, a). Obsérvese que `q:=&(a[2])` es en realidad una asignación de la forma `q:=&a+c` para una constante c. Ahora se puede calcular

$$sal[B_2] = trans_{B_2}(\{(\mathrm{q}, \mathrm{c})\}) = \{(\mathrm{p}, \mathrm{c}), (\mathrm{q}, \mathrm{a})\}$$

De manera similar, $ent[B_3] = \{(\mathrm{q}, \mathrm{c})\}$ y $sal[B_3] = \{(\mathrm{p}, \mathrm{a}), (\mathrm{q}, \mathrm{c})\}$.

A continuación, se encuentra que $ent[B_4] = sal[B_2] \cup sal[B_3] \cup sal[B_5]$. Presumiblemente, $sal[B_5]$ se inicializó con $\varnothing$ y en este paso todavía no ha cambiado. Sin embargo, $sal[B_2] = \{(\mathrm{p}, \mathrm{c}), (\mathrm{q}, \mathrm{a})\}$ y $sal[B_3] = \{(\mathrm{p}, \mathrm{a}), (\mathrm{q}, \mathrm{c})\}$, de modo que

$$ent[B_4] = \{(\mathrm{p}, \mathrm{a}), (\mathrm{p}, \mathrm{c}), (\mathrm{q}, \mathrm{a}), (\mathrm{q}, \mathrm{c})\}$$

El efecto de `p:=p+1` en B_4 es el de eliminar la posibilidad de que p no apunte a una matriz. Es decir,

$$sal[B_4] = trans_{B_4}(ent[B_4]) = \{(\mathrm{p}, \mathrm{a}), (\mathrm{q}, \mathrm{a}), (\mathrm{q}, \mathrm{c})\}$$

Obsérvese que siempre que se ejecute B_2, haciendo que p apunte a c, tiene lugar una acción semánticamente sin sentido si se utiliza p indirectamente después de `p:=p+1` en B_4. Por tanto, este grafo de flujo no es "realista", pero ilustra las inferencias que se pueden hacer sobre los apuntadores.

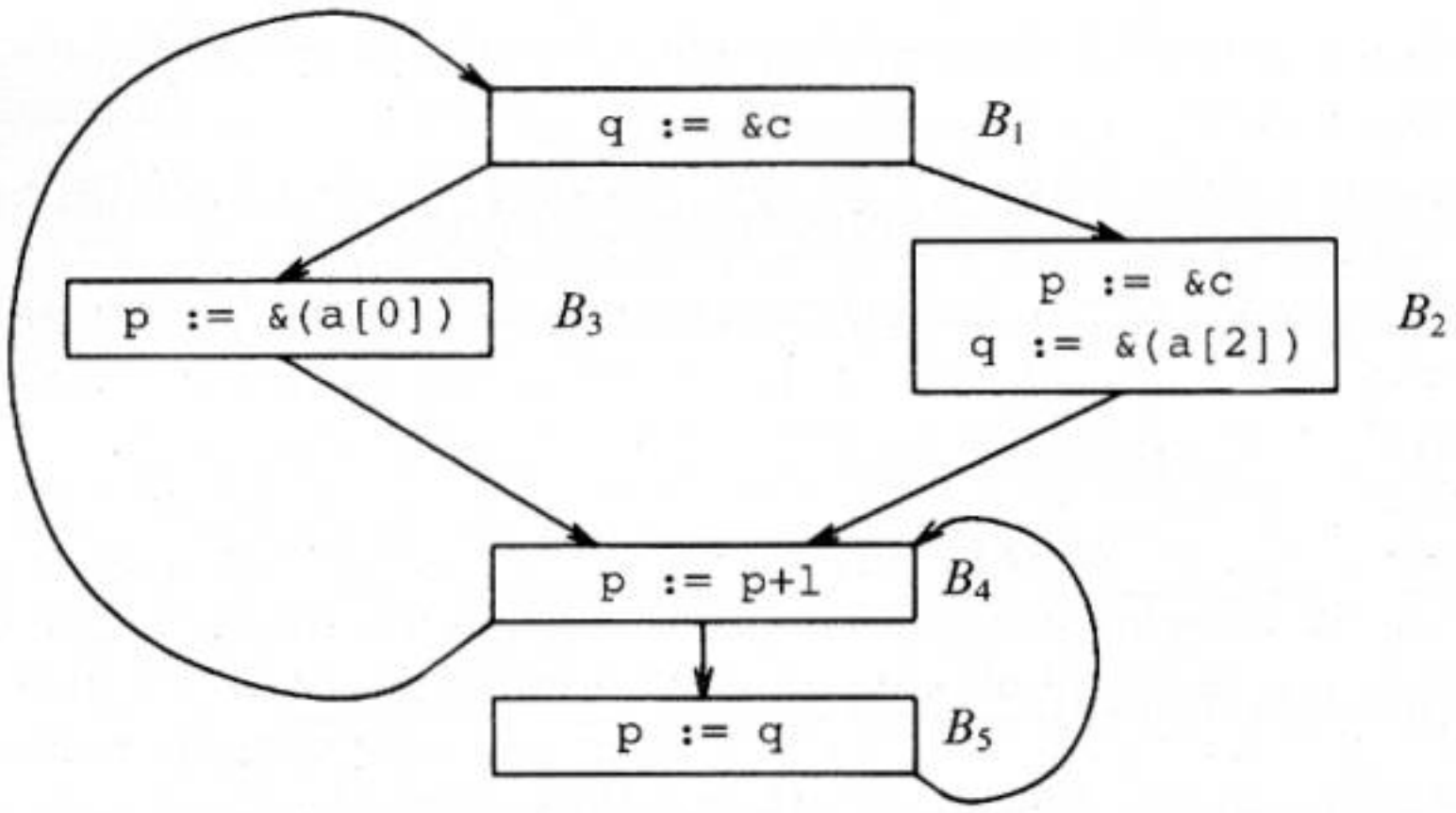


Fig. 10.40. Grafo de flujo que muestra operaciones con apuntadores.

Continuando, *ent* $[B_5]$ = *sal* $[B_4]$, y $trans_{B_5}$ copia los objetivos de q y se los da asimismo a p. Como q puede apuntar a a o a c en *ent* $[B_5]$,

$$sal\,[B_5] = \{(p, a), (p, c), (q, a), (q, c)\}$$

En la siguiente pasada se encuentra que *ent* $[B_1]$ = *sal* $[B_4]$, de modo que *sal* $[B_1]$ = $\{(p, a), (q, c)\}$. Este valor también es el nuevo *ent* $[B_2]$ y *ent* $[B_3]$, pero estos nuevos valores no cambian a *sal* $[B_2]$ o a *sal* $[B_3]$, ni se modifica *ent* $[B_4]$. Por tanto, se ha llegado a la solución deseada. □

Uso de la información acerca de los apuntadores

Supóngase que *ent* $[B]$ es el conjunto de variables apuntadas por cada apuntador al comienzo del bloque B y que se tiene una referencia del apuntador p dentro del bloque B. Comenzando con *ent* $[B]$, aplíquese $trans_s$ para cada proposición s del bloque B que preceda la referencia a p. Este cálculo indicará hacia dónde podría apuntar p en la proposición donde esa información sea importante.

Supóngase ahora que se ha determinado hacia dónde podría apuntar cada apuntador cuando ese apuntador se utiliza en una referencia indirecta, ya sea a la izquierda o a la derecha del símbolo de asignación. ¿Cómo se puede utilizar esta información para obtener soluciones más acertadas a los problemas comunes de flujo de datos? En cada caso se debe considerar en qué dirección son conservadores los errores y se debe utilizar la información de los apuntadores de manera que sólo se cometan errores conservadores. Para ver cómo se realiza esta elección, considérense dos ejemplos: las definiciones de alcance y el análisis de variables activas.

Para calcular las definiciones de alcance se puede utilizar el algoritmo 10.2, pero es necesario conocer los valores de *desact* y *gen* correspondientes a un bloque. Estas últimas cantidades se calculan como siempre para las proposiciones que no son asignaciones indirectas por medio de apuntadores. Se considera que una asignación indirecta *p:=a genera una definición de toda variable tal que p pudiera apuntar a b. Esta suposición es conservadora, porque como se estudió en la sección 10.5, por lo

general es conservador suponer que las definiciones alcanzan un punto cuando en realidad no lo hacen.

Cuando se calcula *desact*, se supondrá que `*p=a` desactiva las definiciones de `b` sólo si `b` no es una matriz y es la única variable a la que podría apuntar `p`. Si `p` pudiera apuntar a dos o más variables, entonces no se supone que se desactivan las definiciones de ninguna de ellas. De nuevo, se está siendo conservador porque se permite que las definiciones de `b` pasen por `*p:=a`, y por tanto que alcancen todas las partes que pueden, a menos que se pueda demostrar que `*p:=a` redefinió `b`. En otras palabras, cuando existe la duda, se supone que una definición alcanza.

Para las variables activas se puede utilizar el algoritmo 10.4, pero se debe reconsiderar cómo se van a definir *def* y *uso* para proposiciones de la forma `*p:=a` y `a:=*p`. La proposición `*p:=a` sólo utiliza `a` y `p`. Se dice que define `b` sólo si `b` es la única variable a la que puede apuntar `p`. Esta suposición permite que los usos de `b` pasen por la proposición a menos que sean bloqueados por la asignación `*p:=a`. Por tanto, nunca se puede decir que `b` está inactiva en un punto cuando en realidad está activa. La proposición `a:=*p` representa siempre una definición de `a`. También representa un uso de `p` y un uso de cualquier variable a la que pudiera apuntar `p`. Maximizando los posibles usos, de nuevo se maximiza la estimación de las variables activas. Al maximizar las variables activas, normalmente se está siendo conservador. Por ejemplo, se puede generar código para almacenar una variable inactiva, pero nunca se dejará de almacenar una que estuviera activa.

Análisis del flujo de datos entre procedimientos

Hasta ahora, se ha hablado de "programas" que son procedimientos simples y por tanto grafos de flujo simples. Ahora se verá cómo reunir información procedente de muchos procedimientos interactuantes. La idea básica consiste en determinar cómo influye cada procedimiento en la información sobre los conjuntos *gen*, *desact*, *uso* o *def* de los otros procedimientos, y después calcular la información del flujo de datos para cada procedimiento independientemente como antes.

Durante el análisis de flujo de datos habrá que considerar sinónimos impuestos por los parámetros en las llamadas a procedimientos. Como no es posible que dos variables globales denoten la misma posición de memoria, al menos uno de un par de sinónimos debe ser un parámetro formal. Como se pueden pasar los parámetros formales a los procedimientos, es posible que dos parámetros formales sean sinónimos.

Ejemplo 10.27. Supóngase que se tiene un procedimiento `p` con dos parámetros formales `x` e `y` pasados por referencia. En la figura 10.41, se observa una situación en la que `b+x` se calcula en B_1 y B_3. Supóngase que los únicos caminos desde los bloques B_1 a B_3 pasan por B_2, y no hay asignaciones a `b` o a `x` a lo largo de ninguno de dichos caminos. Entonces, ¿está disponible `b+x` en B_3? La respuesta depende de si `x` e `y` pueden denotar la misma dirección de memoria. Por ejemplo, podría haber una llamada `p(z,z)`, o tal vez una llamada de `p(u,v)`, donde `u` y `v` son parámetros formales de otro procedimiento `q(u,v)`, y es posible una llamada de `q(z,z)`.

De manera similar, es posible que `x` e `y` sean sinónimos si `x` es un parámetro formal, por ejemplo de `p(x,w)`, e `y` es una variable con un alcance accesible a algún

procedimiento q que llame a p, por ejemplo con llamada p(y,t). Situaciones aún más complicadas podrían hacer que x e y fueran sinónimos de otra variable, y dentro de poco se estudiarán algunas reglas generales para determinar todos esos pares de sinónimos. □

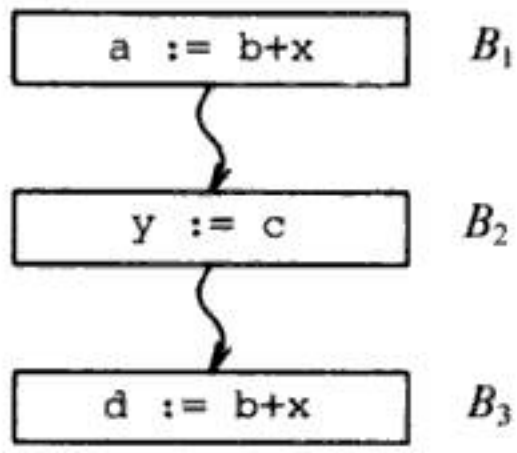


Fig. 10.41. Ilustración de problemas de sinónimos (alias).

En algunas situaciones resultará conservador no considerar algunas variables como sinónimos de otras. Por ejemplo, en las definiciones de alcance, si se desea afirmar que una definición de a es desactivada por una definición de b, es mejor asegurarse de que a y b son realmente sinónimos siempre que se ejecute la definición de b. Otras veces resulta conservador considerar las variables como sinónimos una de otra siempre que exista alguna duda. El ejemplo 10.27 es uno de esos casos. Si la expresión disponible b+x no va a ser desactivada por una definición de y, hay que asegurarse de que ni b ni x puedan ser un sinónimo de y.

Un modelo de código con llamadas a procedimientos

Para ilustrar cómo se pueden considerar los sinónimos, se toma un lenguaje que permita procedimientos recursivos, y cualquiera de ellos pueda hacer referencia tanto a variables locales como globales. Los datos disponibles para un procedimiento constan sólo de los datos globales y de sus propios datos locales; es decir, no hay estructura de bloques para el lenguaje. Los parámetros se pasan por referencia. Todos los procedimientos deben tener un grafo de flujo con una sola *entrada* (el nodo inicial) y un solo nodo de *retorno* que hace que el control vuelva a la rutina que efectúa la llamada. Se supone por conveniencia que todos los nodos están en un camino desde la entrada hasta el regreso.

Ahora supóngase que se está en un procedimiento p y que se encuentra una llamada al procedimiento q(u,v). Si se quiere calcular las definiciones de alcance, las expresiones disponibles o cualquiera de un número de otros análisis de flujo de datos, se debe saber si q(u,v) puede cambiar el valor de alguna variable. Obsérvese que se dice "puede cambiar" en lugar de decir "cambiará". Al igual que todos los problemas de flujo de datos, es imposible saber con certeza si el valor de una variable se modifica o no. Sólo se puede encontrar un conjunto que incluya todas las variables cuyos valores cambien y quizás algunas que no. Con cuidado, se puede reducir la última clase de variables, obteniendo una buena aproximación al conjunto verdadero y fallando sólo del lado conservador.

Las únicas variables cuyos valores podría definir la llamada `q(u,v)` son las globales y las variables `u` y `v`, que pueden ser locales a `p`. Las definiciones de variables locales de `q` no tienen ningún efecto después de que regresa la llamada. Aun cuando `p = q`, cambiarán otras copias de las locales de `q` y esas locales desaparecen después del retorno. Es fácil determinar las locales definidas explícitamente por `q`; sólo hay que comprobar cuáles tienen definiciones en `q`, o están definidas en una llamada a procedimiento hecha por `q`. Además, `u`, `v` o ambas, que pueden ser globales, cambian si `q` tiene una definición de su primero o segundo parámetro, respectivamente, o si `q` pasa estos parámetros formales como parámetros reales a otro procedimiento que los defina. Sin embargo, no todas las variables modificadas por una llamada a `q` tienen que ser definidas explícitamente por `q` o por uno de los procedimientos que llama, porque las variables pueden tener sinónimos.

Cálculo de los sinónimos

Antes de responder a la pregunta de qué variables pueden cambiar en un procedimiento dado, se debe desarrollar un algoritmo para encontrar sinónimos. El enfoque que se utilizará aquí es muy sencillo. Se calcula una relación $\equiv$ sobre variables que formalice la noción "puede ser sinonimo de". Al hacer esto no se establecen diferencias entre ocurrencias de una variable en diferentes llamadas al mismo procedimiento, aunque se distinguen variables locales a distintos procedimientos pero que tengan el mismo identificador.

Para facilitar las cosas, no se intenta diferenciar los conjuntos de sinónimos en distintos puntos del programa, sino que si dos variables pudieran ser sinónimos una de otra se supondrá que siempre lo pueden ser. Por último, se hará la suposición conservadora de que $\equiv$ es transitiva, así que las variables se agrupan en clases de equivalencias, y dos variables podrían ser sinónimo una de la otra si, y sólo si, están en la misma clase.

Algoritmo 10.12. Cálculo simple de sinónimos.

Entrada. Una serie de procedimientos y variables globales.

Salida. Una relación de equivalencia $\equiv$ con la propiedad de que siempre que haya una posición en el programa donde `x` e `y` sean sinónimo una de otra, $\mathtt{x} \equiv \mathtt{y}$; la inversa no siempre se cumple.

Método.

1. Dése otro nombre a las variables, si es necesario, para que dos procedimientos no utilicen el mismo parámetro formal o identificador de variable local, ni para que las variables locales, los parámetros formales o los variables globales compartan un identificador.

2. Si hay un procedimiento $\mathtt{p}(\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n)$ y una invocación $\mathtt{p}(\mathtt{y}_1, \mathtt{y}_2, \ldots, \mathtt{y}_n)$ de ese procedimiento, asígnese $\mathtt{x}_i \equiv \mathtt{y}_i$ para toda i. Es decir, cada parámetro formal puede ser un sinónimo de cualquiera de sus parámetros actuales correspondientes.

3. Tómese la cerradura transitiva y reflexiva de las correspondencias actual-formal añadiendo

   a) $x \equiv y$ siempre que $y \equiv x$.
   b) $x \equiv z$ siempre que $x \equiv y$ e $y \equiv z$ para alguna y. □

Ejemplo 10.28. Considérese el esbozo de los tres procedimientos que se muestran en la figura 10.42, donde se supone que los parámetros se pasan por referencia. Hay dos variables globales, `g` y `h`, y dos variables locales, `i` para el procedimiento `principal` y `k` para el procedimiento `dos`. El procedimiento `uno` tiene los parámetros formales `w` y `x`, el procedimiento `dos` tiene los parámetros formales `y` y `z`, y `principal` no tiene parámetros formales. Por tanto, no hay que dar otro nombre a las variables. Primero se calculan los sinónimos debido a las correspondencias actual-formal.

La llamada a `uno` mediante `principal` hace $h \equiv w$ e $i \equiv x$. La primera llamada a `dos` hecha por `uno` hace $w \equiv y$ y $w \equiv z$. La segunda llamada hace $g \equiv y$ y $x \equiv z$.

La llamada a `uno` por `dos` hace $k \equiv w$ e $y \equiv x$. Cuando se toma la cerradura transitiva de las relaciones de sinónimos representada por $\equiv$, entonces se ve en este ejemplo que todas las variables son sinónimos posibles una de otra. □

```
global g, h;
    procedure principal( );
        local i;
        g := . . . ;
        uno(h, i)
    end;

    procedure uno(w, x);
        x := . . . ;
        dos(w, w);
        dos(g, x)
end;

procedure dos(y, z);
    local k;
    h := . . . ;
    uno(k, y)
end;
```

Fig. 10.42. Procedimientos de muestra.

El cálculo de sinónimos del algoritmo 10.12 no da a menudo como resultado grupos tan extensos de sinónimos como en el ejemplo 10.28. Intuitivamente, no se pensaría frecuentemente que dos variables distintas con tipos diferentes sean sinónimos. Además, el programador tiene indudablemente tipos conceptuales para sus variables. Por ejemplo, si el primer parámetro formal de un procedimiento `p` representa una velocidad, se puede pensar que el programador considerará el primer argumento en cualquier llamada a `p` como una velocidad. Por tanto, se espera intui-

tivamente que la mayoría de los programas produzca pequeños grupos de posibles sinónimos.

Análisis de flujo de datos en presencia de llamadas a procedimientos

Considérese, como ejemplo, cómo se pueden calcular las expresiones disponibles en presencia de llamadas a procedimientos, donde los parámetros se pasan por referencia. Al igual que en la sección 10.6, hay que determinar cuándo podría ser definida una variable, desactivando por tanto una definición, y cuándo se generan (evalúan) las expresiones.

Se puede definir, para cada procedimiento p, un conjunto *cambio* [p], cuyo valor es ser el conjunto de variables globales y parámetros formales de p que pueden ser modificados durante una ejecución de p. En este punto, no se considera que se ha modificado una variable si se ha modificado un miembro de su clase de equivalencia de sinónimos.

Sea *def* [p] el conjunto de parámetros formales y variables globales con definiciones explícitas dentro de p (sin incluir los definidos dentro de procedimientos llamados por p). Para escribir las ecuaciones para *cambio* [p], sólo hay que relacionar las variables globales y los parámetros formales de p utilizados como parámetros actuales en llamadas hechas por p con los parámetros formales correspondientes de los procedimientos llamados. Se puede escribir:

$$cambio\,[\mathtt{p}] = def\,[\mathtt{p}] \cup A \cup G \tag{10.14}$$

donde

1. $A = \{\mathtt{a} \mid \mathtt{a}$ es una variable global o parámetro formal de p tal que, para algún procedimiento q y entero i, p llama a q con a como i-ésimo parámetro actual y el i-ésimo parámetro formal de q está en *cambio* [q]$\}$

2. $G = \{\mathtt{g} \mid \mathtt{g}$ es una variable global en *cambio* [q] y p llama a q$\}$.

No debe sorprender el hecho de que la ecuación (10.14) se pueda resolver para un conjunto de procedimientos mediante una técnica iterativa. Aunque la solución no es única, sólo se necesita la más pequeña. Se puede llegar a esa solución comenzando por una aproximación demasiado pequeña e iterando. Por supuesto, la aproximación demasiado pequeña por la que comenzar es *cambio* [p] = *def* [p]. Los detalles de la iteración se dejan al lector como ejercicio.

Vale la pena considerar el orden en el que se deberían visitar los procedimientos de la iteración anterior. Por ejemplo, si los procedimientos no son mutuamente recursivos, entonces se pueden visitar primero los procedimientos que no llaman a ningún otro (debe haber al menos uno). Para estos procedimientos, *cambio* = *def*. A continuación se puede calcular *cambio* para los procedimientos que llaman sólo a procedimientos que no llaman a nadie. Se puede aplicar directamente (10.14) para este siguiente grupo de procedimientos, puesto que se conocerá *cambio* [q] para cualquier q en (10.14).

Esta idea se puede precisar más de la siguiente manera. Se dibuja un *grafo de llamadas*, cuyos nodos son procedimientos, con una arista de p a q si p llama a q[9]. Una serie de procedimientos que no sean mutuamente recursivos tendrá un grafo de llamadas acíclico. En este caso se puede visitar cada nodo una vez.

Ahora se proporciona un algoritmo para calcular *cambio*.

Algoritmo 10.13. Análisis entre procedimientos de variables modificadas.

Entrada. Una serie de procedimientos $p_1, p_2, \ldots, p_n$. Si el grafo de llamadas es acíclico, se supone que p_i llama a p_j sólo si $j<i$. En caso contrario, no se hacen suposiciones acerca de qué procedimientos llaman a qué otros.

Salida. Para cada procedimiento p, se produce *cambio* [p], el conjunto de variables globales y parámetros formales de p que pueden ser cambiados explícitamente por p sin sinónimos.

Método.

1. Calcúlese *def* [p] para cada procedimiento p mediante inspección.
2. Ejecútese el programa de la figura 10.43 para calcular *cambio*. □

```
(1)  for cada procedimiento p do cambio [p] := def [p]; /*inicializar */
(2)  while ocurren cambios a cualquier cambio [p] do
(3)      for i := 1 to n do
(4)          for cada procedimiento q llamado por p_i do begin
(5)              añadir las variables globales de cambio [q] a cambio [p_i];
(6)                  for cada parámetro formal x (el j-ésimo) de q do
(7)                      if x está en cambio [q] then
(8)                          for cada llamada a q por p_i do
(9)                              if a, el j-ésimo parámetro actual de la llamada,
                                     es un parámetro global o formal de p_i
(10)                                 then añadir a a cambio [p_i]
              end
```

Fig. 10.43. Algoritmo iterativo para calcular *cambio*.

Ejemplo 10.29. Se considera de nuevo la figura 10.42. Por inspección, *def* [principal] = {g}, *def* [uno] = {x} y *def* [dos] = {h}. Estos son los valores iniciales de *cambio*. El grafo de llamadas de los procedimientos se muestra en la figura 10.44. Se tomará dos, uno, principal como el orden en que se visitan los procedimientos.

Considérese p_i = dos en el programa de la figura 10.42. Entonces q sólo puede ser el procedimiento uno en la línea (4). Como *cambio* [uno] = {x} inicialmente,

---

[9] Se supone aquí que no hay variables de tipo procedimiento. Estas complican la construcción del grafo de llamadas, ya que hay que determinar los posibles parámetros actuales correspondientes a los parámetros formales de tipo procedimiento al mismo tiempo que se construyen las aristas del grafo de llamadas.

no se añade nada a *cambio* [dos] en la línea (5). En las líneas (6) y (7) sólo hay que considerar el segundo parámetro formal del procedimiento uno, ya que el primer parámetro actual es local a dos. En la única llamada a uno de dos, el segundo parámetro actual es y, y x su correspondiente parámetro formal, se modifica, así que *cambio* [dos] se iguala a {h, y} en la línea (10).

Ahora se considera p_i = uno. En la línea (4), q sólo puede ser el procedimiento dos.

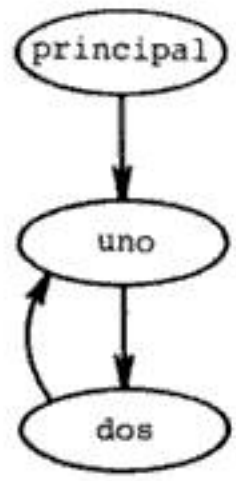


Fig. 10.44. Grafo de llamadas.

En la línea (5), h es una variable global en *cambio* [dos], de modo que se hace *cambio* [uno] = {h, x}. En las líneas (6) y (7), sólo el primer parámetro formal de dos está en *cambio* [dos], así que se debe añadir g y w a *cambio* [uno] en la línea (10), siendo éstos los primeros dos parámetros actuales en las llamadas al procedimiento dos. Por tanto, *cambio* [uno] = {g, h, w, x}.

Considérese ahora principal. El procedimiento uno cambia sus dos parámetros formales, así que tanto h como i cambiarán en la llamada a uno hecha por principal. Sin embargo, i es una variable local y no es necesario considerarla. Por tanto, se hace *cambio* [principal] = {g, h}. Por último, se repite el lazo **while** de la línea (2). Al considerar dos, se ve que uno modifica la variable global g. Por consiguiente, la llamada a uno(k,y) hace que g se modifique, así que *cambio* [dos] = {g, h, y}. No ocurren más cambios en la iteración. □

Uso de la información de cambio

Como ejemplo de cómo se puede utilizar *cambio*, considérese el cálculo de subexpresiones comunes globales. Supóngase que se están calculando las expresiones disponibles para un procedimiento p, y que se quiere calcular *d_desact* [*B*] para un bloque *B*. Se debe considerar que una definición de la variable a desactiva cualquier expresión que implique a o una x que pudiera ser un sinónimo de a. Sin embargo, una llamada en *B* al procedimiento q no puede desactivar una expresión que involucre a a a menos que a sea una sinónimo (recuérdese que a es un sinónimo de sí misma) de alguna variable en *cambio* [q]. Por tanto, se puede utilizar la información calculada por los algoritmos 10.12 y 10.13 para construir una aproximación segura al conjunto de expresiones desactivadas.

Para calcular las expresiones disponibles en programas con llamadas a procedimientos, también hay que tener una forma conservadora de estimar el conjunto de expresiones generadas por una llamada a un procedimiento. Para ser conservadores, se puede suponer que a+b es generada por una llamada a q si, y sólo si, en cada

camino desde la entrada de q hasta su regreso, se encuentra a+b sin una definición subsiguiente de a o b. Cuando se buscan ocurrencias de a+b no se debe aceptar x+y como tal ocurrencia a menos que se tenga la seguridad que, en cada llamada a q, x sea un sinónimo de a e y un sinónimo de b.

Se exige este requisito porque es conservador equivocarse al asumir que una expresión no está disponible cuando lo está. Por la misma razón se debe suponer que a+b es desactivada por una definición de cualquier z que pudiera ser un sinónimo de a o b. Por tanto, la forma más sencilla de calcular las expresiones disponibles para todos los nodos de todos los procedimientos es suponer que una llamada no genera nada y que *d_desact* [*B*] para todos los bloques *B* se calcula como antes. Como no se espera que muchas expresiones sean generadas por el procedimiento habitual, este enfoque es suficientemente bueno para la mayoría de los propósitos.

Un enfoque alternativo más complicado y más exacto para el cálculo de expresiones disponibles consiste en calcular *gen* [p] para cada procedimiento p de forma iterativa. Se puede inicializar *gen* [p] como el conjunto de expresiones disponibles al final del nodo de regreso de p según el método anterior. Es decir, no se permiten sinónimos para las expresiones generadas; a+b sólo se representa a sí misma, aunque otras variables puedan ser sinónimos de a o b.

Ahora se calculan de nuevo las expresiones disponibles para todos los nodos de todos los procedimientos. Sin embargo, una llamada a q(a,b) genera aquellas expresiones en *gen* [q] con a y b sustituidas por los parámetros formales correspondientes de q. *d_desact* permanece como antes. Se puede encontrar un nuevo valor de *gen* [p] para cada procedimiento p, comprobando las expresiones que están disponibles al final del regreso de p. Esta iteración se puede repetir hasta que no se obtengan más cambios en las expresiones disponibles en ningún nodo.

10.9 ANALISIS DE FLUJO DE DATOS DE GRAFOS DE FLUJO ESTRUCTURADOS

Los programas sin proposiciones **goto** tienen grafos de flujo reducibles al igual que los programas inspirados por varias metodologías de programación. Varios estudios de amplias clases de programas han revelado que casi todos los programas escritos por personas tienen grafos de flujo reducibles[10]. Esta observación es relevante para la optimación porque se pueden encontrar algoritmos de optimación que se ejecuten significativamente más rápidamente con grafos de flujo reducibles. En esta sección se analiza una variedad de conceptos de grafos de flujo, como el "análisis de intervalos", relevantes sobre todo para los grafos de flujo estructurados. Fundamentalmente, se aplicarán las técnicas dirigidas por la sintaxis desarrolladas en la sección 10.5 al planteamiento más general donde no es la sintaxis la que proporciona necesariamente la estructura, sino el grafo de flujo.

---

[10] "Escritos por personas" no es redundante porque sabe de varios programas que generan código con "nidos de ratas" de proposiciones **goto**. Esto no es incorrecto; la estructura está en la entrada a dichos programas.

Búsqueda en profundidad

Hay un ordenamiento útil de los nodos de un grafo de flujo, conocido como ordenamiento *en profundidad*, que es una generalización del recorrido en profundidad de un árbol que se estudió en la sección 2.3. Un ordenamiento en profundidad se puede utilizar para detectar lazos en cualquier grafo de flujo; también ayuda a acelerar los algoritmos de flujo de datos como los estudiados en la sección 10.6. El ordenamiento en profundidad se crea comenzando en el nodo inicial y examinando el grafo completo, intentando visitar los nodos muy alejados del nodo inicial tan rápidamente como sea posible (*en profundidad*). La ruta de la búsqueda forma un árbol. Antes de proporcionar el algoritmo, considérese un ejemplo.

Ejemplo 10.30. En la figura 10.46 se ilustra una posible búsqueda en profundidad del grafo de flujo de la figura 10.45. Las aristas sólidas forman un árbol; las aristas punteadas son las otras aristas del grafo de flujo. La búsqueda en profundidad del grafo de flujo corresponde a un recorrido anterior al orden del árbol, $1 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 10$, después se regresa a 8, después a 9. Se vuelve de nuevo a 8, retrocediendo a 7, 6 y 4 y después avanzando hacia 5. Se retrocede desde 5 a 4, después vuelta a 3 y 1. Desde 1 se va a 2, después se regresa de 2, de vuelta a 1, y se ha recorrido el árbol completo en orden previo. Obsérvese que todavía no se ha explicado cómo se selecciona este árbol a partir del grafo de flujo. □

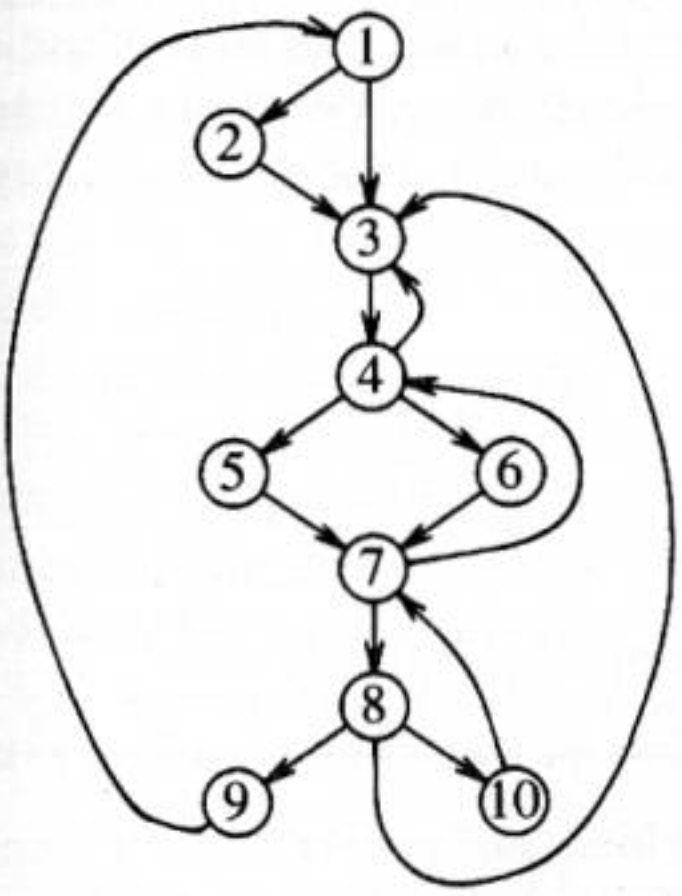


Fig. 10.45. Grafo de flujo.

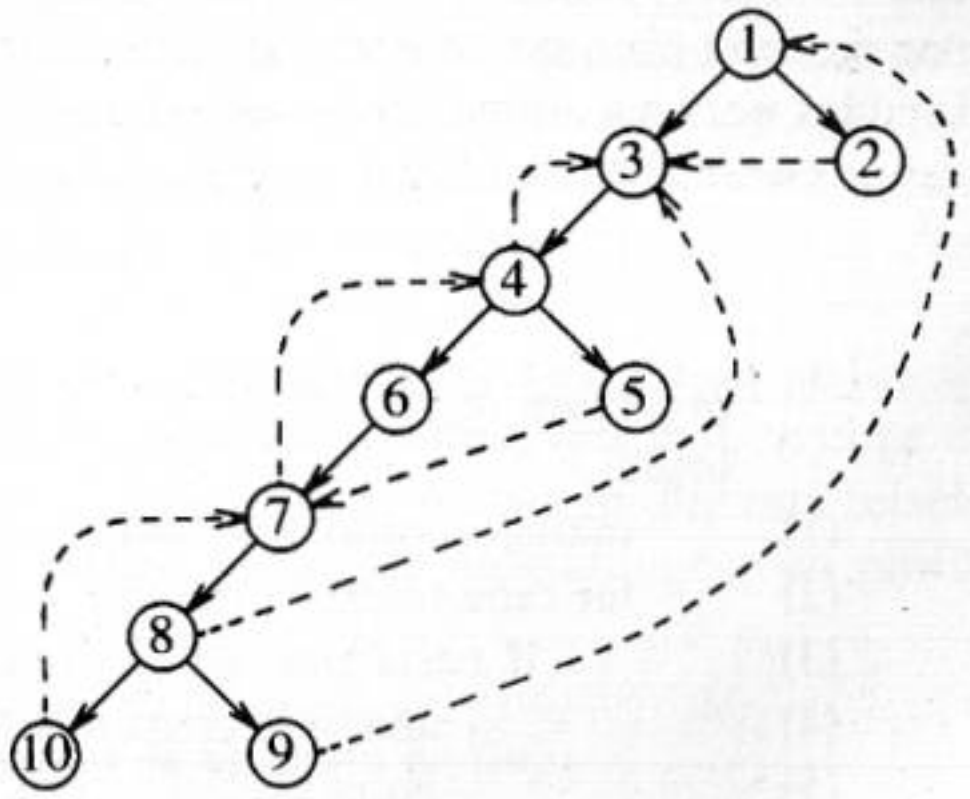


Fig.10.46. Presentación en profundidad.

El *ordenamiento en profundidad* de los nodos es el contrario del orden en que se visitaron los nodos en el recorrido en orden previo.

Ejemplo 10.31. En el ejemplo 10.30, la secuencia completa de nodos visitados a medida que se recorre el árbol es

1, 3, 4, 6, 7, 8, 10, 8, 9, 8, 7, 6, 4, 5, 4, 3, 1, 2, 1.

En esta lista, se marca la última ocurrencia de cada número para obtener

1, 3, 4, 6, 7, 8, 10, 8, 9, 8, 7, 6, 4, 5, 4, 3, 1, 2, 1.

El ordenamiento en profundidad es la secuencia de números marcados en orden inverso. Aquí esta secuencia resulta ser 1, 2, . . . , 10. Es decir, al inicio los nodos se numeraron en orden de profundidad. □

Ahora se da un algoritmo que calcula un ordenamiento en profundidad de un grafo de flujo construyendo y recorriendo un árbol con raíz en el nodo inicial, intentando construir caminos en el árbol siempre que sea posible. Dicho árbol se denomina *árbol de expansión en profundidad* (*aep*). Este algoritmo se utilizó para construir la figura 10.46 a partir de la figura 10.45.

Algoritmo 10.14. Arbol de expansión en profundidad y ordenamiento en profundidad.

Entrada. Un grafo de flujo G.

Salida. Un *aep* T de G y un ordenamiento de los nodos de G.

Método. Se utiliza el procedimiento recursivo *busca*(n) de la figura 10.47; el algoritmo consiste en inicializar todos los nodos de G como "no visitados", después llamar *busca*(n_0), donde n_0 es el nodo inicial. Cuando se llama *busca*(n), primero se marca n como "visitado", para evitar añadir dos veces n al árbol. Se utiliza i para contar desde el número de nodos de G hasta 1, asignando números en profundidad $np\,[n]$ a los nodos n a medida que se avanza. El conjunto de aristas T forma el árbol de expansión en profundidad correspondiente a G, y se le denominan *aristas de árbol*. □

```
        procedure busca (n);
        begin
(1)         marca n como "visitado";
(2)         for cada sucesor s de n do
(3)             if s está "no visitado" then begin
(4)                 añadir la arista n → s a T;
(5)                 busca (s)
                end;
(6)         np [n] := i;
(7)         i := i − 1
        end;

        /* sigue el programa principal */

(8)     T := vacío; /* conjunto de aristas */
(9)     for cada nodo n de G do marca n como "no visitado";
(10)    i := número de nodos de G;
(11)    busca (n0)
```

Fig. 10.47. Algoritmo de búsqueda en profundidad.

Ejemplo 10.32. Considérese la figura 10.47. Se asigna 10 a i y se llama a *busca*(1). En la línea (2) de *busca* se debe considerar cada sucesor del nodo 1. Supóngase que primero se considera $s = 3$. Después se añade la arista $1 \rightarrow 3$ al árbol y se llama a *busca*(3). En *busca*(3) se añade la arista $3 \rightarrow 4$ a T y se llama a *busca*(4).

Supóngase que en *busca*(4) se elige $s = 6$ primero. Entonces se añade la arista $4 \rightarrow 6$ a T y se llama a *busca*(6). Esta a su vez hace que se añada $6 \rightarrow 7$ a T y se llame a *busca*(7). El nodo 7 tiene dos sucesores, 4 y 8. Pero 4 ya había sido marcado como "visitado" por *busca*(4), de modo que no se hace nada cuando $s = 4$. Cuando $s = 8$ se añade la arista $7 \rightarrow 8$ a T y se llama a *busca*(8). Supóngase que entonces se escoge $s = 10$. Se añade la arista $8 \rightarrow 10$ y se llama a *busca*(10).

Ahora 10 tiene un sucesor, 7, pero 7 ya se ha marcado como visitado, así que en *busca*(10), se cae hasta el paso (6) de la figura 10.47, haciendo $np[10] = 10$ e $i = 9$. Esto completa la llamada a *busca*(10), así que se vuelve a *busca*(8). Ahora se hace $s = 9$ en *busca*(8), se añade la arista $8 \rightarrow 9$ a T y se llama a *busca*(9). El único sucesor del nodo 9, el nodo 1, ya ha sido "visitado", así que se hace $np[9] = 9$ e $i = 8$. Después se regresa a *busca*(8). El último sucesor de 8, el nodo 3, es "visitado" así que no se hace nada para $s = 3$. En este punto se han considerado todos los sucesores de 8, de modo que se hace $np[8] = 8$ e $i = 7$, regresando a *busca*(7).

Ya han sido considerados todos los sucesores de 7, de modo que se hace $np[7] = 7$ e $i = 6$, regresando a *busca*(6). De manera similar, ya han sido considerados los sucesores de 6, así que se hace $np[6] = 6$ e $i = 5$, y se regresa a *busca*(4). El sucesor 3 de 4 ya ha sido "visitado", pero 5 todavía no, de modo que se añade $4 \rightarrow 5$ al árbol y se llama a *busca*(5), que ya no genera más llamadas, puesto que el sucesor 7 de 5 ya ha sido "visitado". Por tanto, $np[5] = 5$, i se hace igual a 4, y se regresa a *busca*(4). Se ha terminado de considerar los sucesores de 4, así que se hace $np[4] = 4$ e $i = 3$, regresando a *busca*(3). Después se hace $np[3] = 3$ e $i = 2$ y se regresa a *busca*(1).

Los pasos finales consisten en llamar a *busca*(2) desde *busca*(1), hacer $np[2] = 2$, $i = 1$, regresar a *busca*(1), hacer $np[1] = 1$ e $i = 0$. Obsérvese que se elige una numeración de los nodos tal que $np[i] = i$, pero no es necesario que esta relación se cumpla para un grafo cualquiera, o incluso para otro ordenamiento en profundidad del grafo de la figura 10.45. □

Aristas en una presentación en profundidad de un grafo de flujo

Cuando se construye un *aep* para un grafo de flujo, las aristas del grafo de flujo pertenecen a tres categorías.

1. Hay aristas que van de un nodo m a un ancestro de m en el árbol (posiblemente a m mismo). Estas aristas se denominarán *aristas de retroceso*. Por ejemplo, $7 \rightarrow 4$ y $9 \rightarrow 1$ son aristas de retirada en la figura 10.46. Es un hecho interesante y útil que si el grafo de flujo es reducible, entonces las aristas de retirada son exactamente las aristas de retroceso del grafo de flujo[11], independientemente del

[11] Recuérdese que las aristas de retroceso de un grafo de flujo son aquellas cuyas cabezas dominan a sus colas.

orden en que se visiten los sucesores en el paso (2) de la figura 10.47. Para cualquier grafo de flujo, toda arista de retroceso es de retirada, aunque si el grafo es no reducible habrá algunas aristas de retirada que no sean aristas de retroceso.

2. Hay aristas, llamadas *aristas de avance*, que van desde un nodo m a un descendiente de m en el árbol. Todas las aristas del *aep* son aristas de avance. No hay más aristas de avance en la figura 10.46 pero, por ejemplo, si $4 \rightarrow 8$ fuera una arista, estaría en esta categoría.

3. Hay aristas $m \rightarrow n$ tales que ni m ni n son un ancestro del otro en el *aep*. Las aristas $2 \rightarrow 3$ y $5 \rightarrow 7$ son los únicos ejemplos de este tipo en la figura 10.46. Estas aristas se denominan *aristas cruzadas*. Una propiedad importante de las aristas cruzadas es que si se dibuja el *aep* de modo que los hijos de un nodo se dibujen de izquierda a derecha en el orden en que fueron añadidas al árbol, entonces todas las aristas cruzadas viajan de derecha a izquierda.

Se debe observar que $m \rightarrow n$ es una arista de retroceso si, y sólo si, $np[m] \geq np[n]$. Para comprobar por qué, obsérvese que si m es un descendiente de n en el *aep*, entonces *busca*(m) termina antes que *busca*(n), de modo que $np[m] \geq np[n]$. A la inversa, si $np[m] \geq np[n]$, entonces *busca*(m) termina antes que *busca*(n), o $m = n$. Pero *busca*(n) debe haber comenzado antes que *busca*(m) si hay una arista $m \rightarrow n$, o si no, el hecho de que n sea un sucesor de m haría que n fuera un descendiente de m en el *aep*. Por tanto, el tiempo en que esté activo *busca*(m) es un subintervalo del tiempo que esté activo *busca*(n), de lo cual se deduce que n es un ancestro de m en el *aep*.

Profundidad de un grafo de flujo

Hay un parámetro importante de los grafos de flujo llamado *profundidad*. Dado un árbol de expansión en profundidad para el grafo, la profundidad es el número mayor de aristas de retirada en cualquier camino sin ciclos.

Ejemplo 10.33. En la figura 10.46, la profundidad es 3 porque hay un camino

$$10 \rightarrow 7 \rightarrow 4 \rightarrow 3$$

con tres aristas de retroceso, pero no hay ningún camino sin ciclos con cuatro o más aristas de retroceso. Es una coincidencia que el camino "más profundo" aquí sólo tenga aristas de retroceso; en general, se puede tener una mezcla de aristas de retroceso, de avance, y cruzadas en un camino "más profundo". □

Se puede demostrar que la profundidad nunca es mayor que la que intuitivamente uno llamaría profundidad de anidamiento de lazos en el grafo de flujo. Si un grafo de flujo es reducible, se puede sustituir "de retroceso" por "retirada" en la definición de "profundidad", porque las aristas de retirada en cualquier *aep* son exactamente las aristas de retroceso. La noción de profundidad entonces se vuelve independiente del *aep* elegido.

Intervalos

La division de un grafo de flujo en intervalos sirve para imponer una estructura jerárquica en el grafo de flujo. Esta estructura ayuda a su vez a aplicar las reglas para el análisis de flujo de datos dirigido por la sintaxis cuyo desarrollo comenzó en la sección 10.5.

Intuitivamente, un "intervalo" en un grafo de flujo es un lazo natural más una estructura acíclica que cuelga de los nodos de un lazo. Una propiedad importante de los intervalos es que tienen nodos *encabezamiento* que dominan todos los nodos en el intervalo; es decir, cada intervalo es una región. Formalmente, dado un grafo de flujo G con nodo inicial n, y un nodo n de G, el *intervalo con encabezamiento n*, indicado como $\mathrm{I}(n)$, se define como sigue.

1. n está en $\mathrm{I}(n)$.
2. Si todos los predecesores de algún nodo $m \neq n_0$ están en $\mathrm{I}(n)$, entonces m está en $\mathrm{I}(n)$.
3. No hay nada más en $\mathrm{I}(n)$.

Por tanto, se puede construir $\mathrm{I}(n)$ comenzando por n y añadiendo nodos m mediante la regla 2. No importa en qué orden se añadan dos candidatos m porque una vez que los predecesores de un nodo estén todos en $\mathrm{I}(n)$, permanecen en $\mathrm{I}(n)$, y se añadirá finalmente cada candidato por la regla 2. Llegará un momento, en que no se puedan añadir más nodos a $\mathrm{I}(n)$, y el conjunto resultante de nodos será el intervalo con encabezamiento n.

Particiones de intervalos

Dado cualquier grafo de flujo G, se puede particionar en intervalos disjuntos como sigue.

Algoritmo 10.15. Análisis de intervalos de un grafo de flujo.

Entrada. Un grafo de flujo G con nodo inicial n_0.

Salida. Una partición de G en un conjunto de intervalos disjuntos.

Método. Para cualquier nodo n, se calcula $\mathrm{I}(n)$ por el método esbozado antes:

```
I(n) := {n};
while existe un nodo m ≠ n0,
        cuyos predecesores están todos en I(n) do
            I(n) := I(n) ∪ {m}
```

Los nodos que sean encabezamientos de intervalos en la partición se eligen de la siguiente manera. Al inicio, ningún nodo está "seleccionado".

```
construyanse I(n0) y "selecciónense" todos los nodos en ese intervalo;
while hay un nodo m, no "seleccionado" todavía,
        pero con un predecesor seleccionado do
            constrúyase I(m) y "selecciónense" todos los nodos en ese intervalo
```

□

Una vez que un candidato m tiene un predecesor p seleccionado, nunca se podrá añadir m a un intervalo que no contenga p. Por tanto, los candidatos m permanecen candidatos hasta que se seleccionan para encabezar su propio intervalo. Por tanto, el orden en que se seleccionan los encabezamientos de intervalos m en el algoritmo 10.15 no afecta la partición final en intervalos. Asimismo, mientras todos los nodos sean alcanzables desde n_0, se puede demostrar por inducción sobre la longitud de un camino desde n_0 a n que el nodo n será puesto finalmente en el intervalo de algún otro nodo, o se convertirá en un encabezamiento de su propio intervalo, pero no ambas. Así, el conjunto de intervalos construido en el algoritmo 10.15 particiona G realmente.

Ejemplo 10.34. Encuéntrese la partición de intervalos para la figura 10.45. Se comienza por cosntruir I(1), porque el nodo 1 es el nodo inicial. Se puede añadir 2 a I(1) porque el único predecesor de 2 es 1. Sin embargo, no se puede añadir 3 porque tiene predecesores, 4 y 8, que todavía no están en I(1), y de manera similar, todos los otros nodos excepto 1 y 2 tienen predecesores que todavía no están en I(1). Así, $I(1) = \{1,2\}$.

Ahora se puede calcular I(3) porque 3 tiene algunos predecesores "seleccionados", 1 y 2, pero 3 no es el mismo un intervalo. Sin embargo, no se puede añadir ningún nodo a I(3), así que $I(3) = \{3\}$. Ahora 4 es un encabezamiento porque tiene un predecesor, 3, en un intervalo. Se puede añadir 5 y 6 a I(4) porque sólo tienen a 4 como predecesor pero no se pueden añadir más nodos; por ejemplo, 7 tiene como predecesor a 10.

A continuación, 7 se convierte en encabezamiento y se puede añadir 8 a I(7). Después se pueden añadir 9 y 10, porque éstos tienen sólo a 8 como predecesor. Por tanto, los intervalos en la partición de la figura 10.45 son:

$I(1) = \{1,2\}$ $\quad$ $I(4) = \{4,5,6\}$
$I(3) = \{3\}$ $\quad$ $I(7) = \{7,8,9,10\}$ □

Grafos de intervalos

Siguiendo los intervalos de un grafo de flujo G, se puede construir un nuevo grafo de flujo I(G) mediante las siguientes reglas:

1. Los nodos de I(G) corresponden a los intervalos en la partición de intervalos de G.
2. El nodo inicial de I(G) es el intervalo de G que contiene el nodo inicial de G.
3. Hay una arista del intervalo I a un intervalo distinto J si, y sólo si, hay en G una arista desde algún nodo en I al encabezamiento de J. Obsérvese que no podría ser una arista que entrara a algún nodo n de J que fuera el encabezamiento, desde fuera de J, porque entonces n no podría haberse añadido a J en el algoritmo 10.15.

Se puede aplicar el algoritmo 10.15 y la construcción de grafos de intervalos por turno, produciéndose la secuencia de grafos G, I(G), I(I(G)), ... Finalmente, se llegará a un grafo en el que cada uno de sus nodos sea un intervalo en sí mismo. Este

grafo se denomina *grafo de flujo límite* de G. Un hecho interesante es que un grafo de flujo es reducible si, y sólo si, su grafo de flujo límite es un solo nodo[12].

Ejemplo 10.35. En figura 10.48 se muestra el resultado de la aplicación repetida de la construcción de intervalos a la figura 10.45. En el ejemplo 10.34 se dieron los intervalos de este grafo y el grafo de intervalos construido a partir de ellos se muestra en la figura 10.48(a). Obsérvese que la arista $10 \rightarrow 7$ de la figura 10.45 no hace que

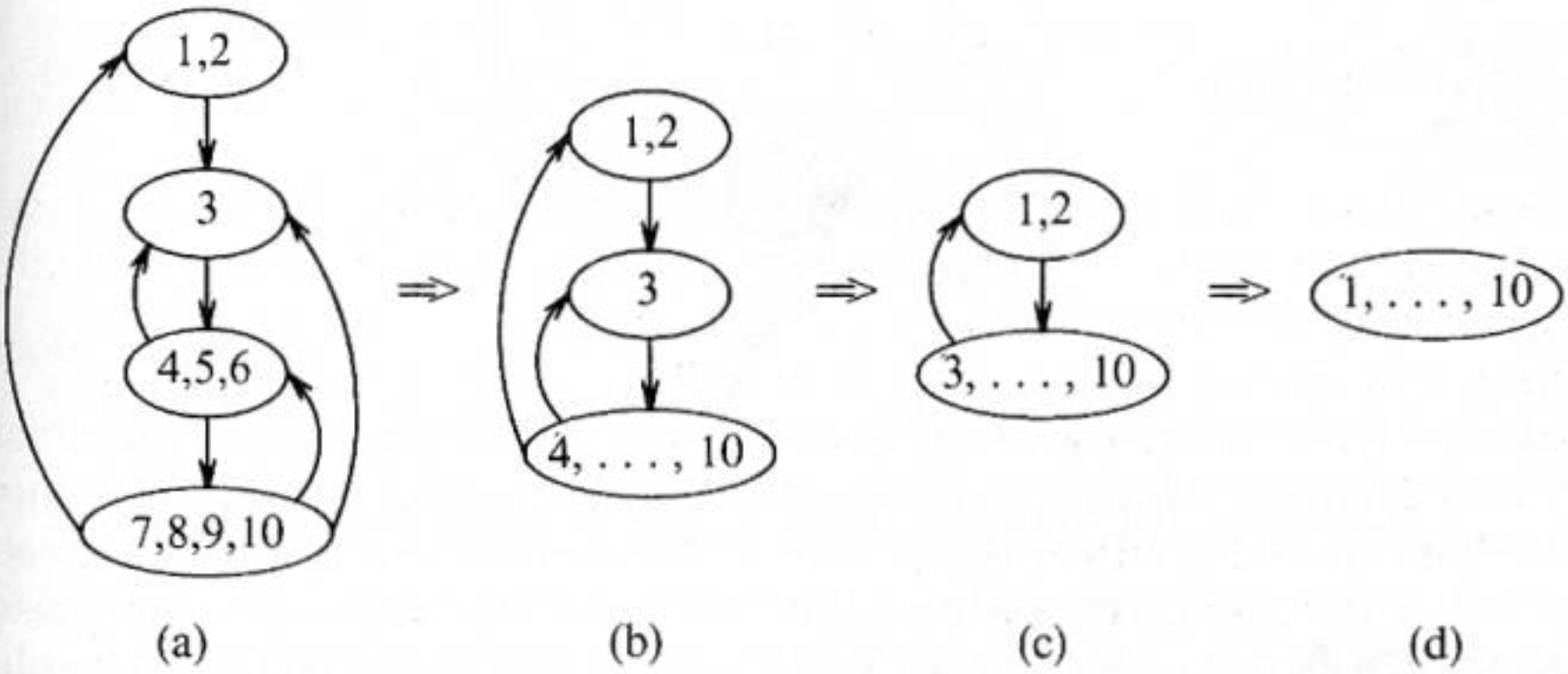


Fig. 10.48. Secuencia de grafos de intervalos.

se forme una arista desde el nodo que representa a {7,8,9,10} a sí mismo en la figura 10.48(a) porque la construcción de grafos de intervalos excluye explícitamente dichos lazos. Obsérvese también que el grafo de flujo de la figura 10.45 es reducible porque su grafo de flujo límite es un solo nodo. □

Separación de nodos

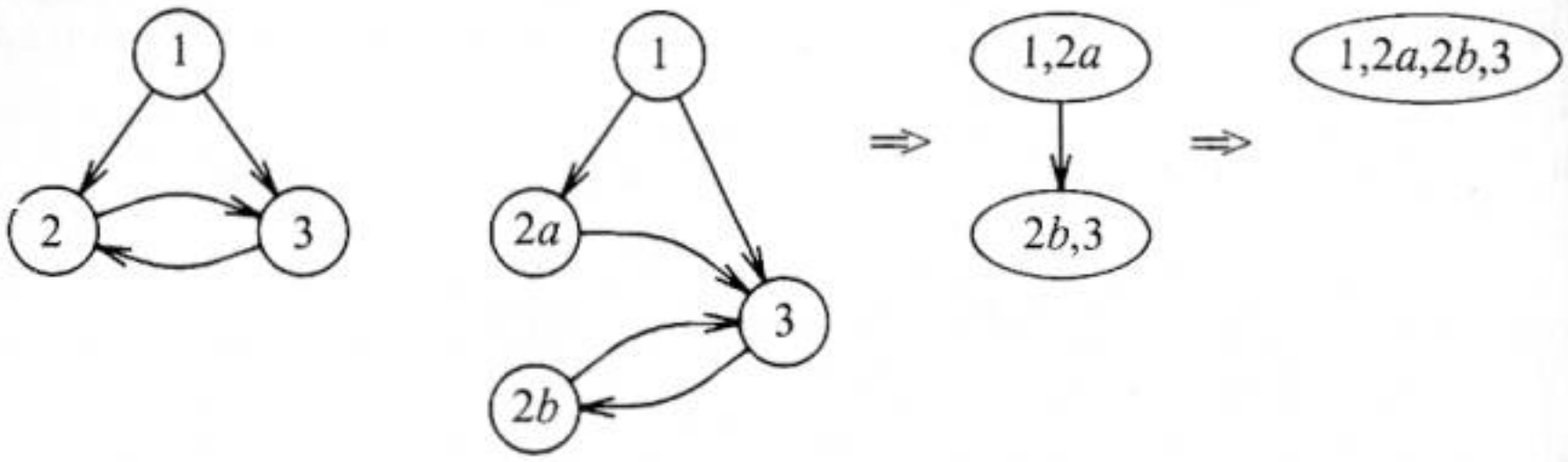
Si se llega a un grafo de flujo límite que no sea un solo nodo, se puede seguir avanzando sólo si se separan uno o más nodos. Si un nodo n tiene k predecesores, se puede sustituir n por k nodos, $n_1, n_2, \ldots, n_k$. El i-ésimo predecesor de n se convierte en el predecesor de n_i sólo, mientras que todos los sucesores de n se convierten en sucesores de todos los n_i.

Si se aplica el algoritmo 10.15 al grafo resultante, cada n_i tiene un predecesor único y por tanto se convertirá en parte del intervalo de dicho predecesor. Así, una división de nodos más una ronda de particionamiento de intervalos dará como resultado un grafo con menos nodos. Como consecuencia, la construcción de grafos de intervalos intercalada con una separación de nodos cuando sea necesario, debe finalmente lograr un grafo de un solo nodo. El significado de esta observación se verá en la siguiente sección, cuando se diseñen los algoritmos de análisis de flujo de datos que se guían por estas dos operaciones con grafos.

---

[12] De hecho, ésta es históricamente la definición original.

Ejemplo 10.36. Considérese el grafo de flujo de la figura 10.49(a), que es su propio grafo de flujo límite. Se puede separar el nodo 2 en $2a$ y $2b$, con predecesores 1 y 3, respectivamente. Este grafo se muestra en la figura 10.49(b). Si se aplica dos veces el particionamiento de intervalos, se obtiene la secuencia de grafos que se muestra en la figura 10.49(c) y (d), acabando en un solo nodo. □



(a) (b) (c) (d)

Fig. 10.49. Separación de nodos seguida de partición de intervalos.

Análisis T_1 - T_2

Una manera adecuada de lograr el mismo efecto que el análisis de intervalos consiste en aplicar dos transformaciones sencillas a los grafos de flujo.

T_1: Si n es un nodo con un lazo, es decir, una arista $n \rightarrow n$, bórrese esa arista.

T_2: Si hay un nodo n, que no sea el nodo inicial, con un predecesor único, m, entonces m puede *consumir* n borrando n y haciendo que todos los sucesores de n (incluido m, posiblemente) sean sucesores de m.

Algunos hechos interesantes acerca de las transformaciones T_1 y T_2 son:

1. Si se aplica T_1 y T_2 a un grafo de flujo G en cualquier orden hasta que se obtenga un grafo de flujo para el que no sean posibles aplicaciones de T_1 y T_2, entonces se obtiene un grafo de flujo único. La razón es que un candidato para la eliminación del lazo por medio de T_1 o el consumo por T_2 sigue siendo un candidato, aunque primero se haga alguna otra aplicación de T_1 o T_2.
2. El grafo de flujo resultante de la aplicación exhaustiva de T_1 y T_2 a G es el grafo de flujo límite de G. La prueba es bastante sutil y se deja como ejercicio. Como consecuencia, otra definición de "grafo de flujo reducible" es un grafo que puede ser convertido por T_1 y T_2 en un solo nodo.

Ejemplo 10.37. En la figura 10.50 se ve una secuencia de las reducciones T_1 y T_2 comenzando por un grafo de flujo que es un renombramiento de la figura 10.49(b). En la figura 10.50(b), c ha consumido d. Obsérvese que el lazo en cd en la figura 10.50(b) se obtiene de la arista $d \rightarrow c$ de la figura 10.50(a). Ese lazo es eliminado por T_1 en la figura 10.50(c). Obsérvese también que cuando a consume b en la figura 10.50(d), las aristas que van de a y b al nodo cd se convierten en una sola arista. □

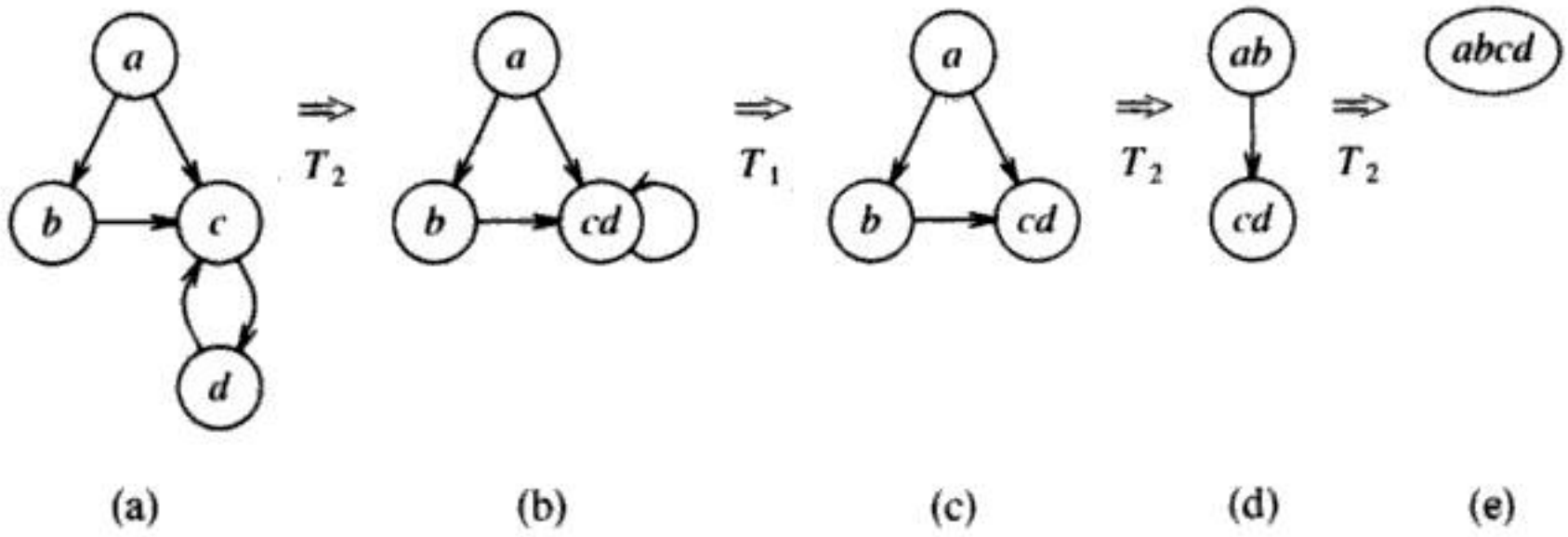


Fig. 10.50. Reducción por T_1 y T_2.

Regiones

Recuérdese que en la sección 10.5 se ha visto que una región en un grafo de flujo es un conjunto de nodos N que incluye un encabezamiento que domina todos los otros nodos en la región. Todas las aristas entre los nodos de N están en la región, excepto (posiblemente) algunas de ellas que entren al encabezamiento. Por ejemplo, todo intervalo es una región, pero hay regiones que no son intervalos porque, por ejemplo, pueden omitir algunos nodos que incluiría un intervalo, o pueden omitir algunas aristas de vuelta al encabezamiento. También hay regiones mucho más grandes que cualquier intervalo, como ya se verá.

A medida que se reduce un grafo de flujo G por T_1 y T_2, en todo momento se cumplen las siguientes condiciones:

1. Un nodo representa una región G.
2. Una arista desde a a b representa un conjunto de aristas. Cada una de dichas aristas es de un nodo en la región representada por a al encabezamiento de la región representada por b.
3. Cada nodo y arista de G está representado exactamente por un nodo o arista del grafo en curso.

Para saber por qué se cumplen estas observaciones, obsérvese primero que se cumplen de manera trivial para G misma. Cada nodo es una región por sí mismo, y cada arista se representa sólo a sí misma. Supóngase que se aplica T_1 a un nodo n que represente un región R, en tanto que el lazo $n \rightarrow n$ representa un conjunto de aristas E y todas deben entrar al encabezamiento de R. Si se añaden las aristas E a la región R, ésta sigue siendo una región, de modo que después de eliminar la arista $n \rightarrow n$, el nodo n representa a R y a las aristas de E, lo cual preserva las condiciones 1 a 3 anteriores.

Si se hubiera utilizado T_2 para consumir el nodo b por el nodo a, supóngase que a y b representan las regiones R y S, respectivamente. Asimismo, sea E el conjunto de aristas representadas por la arista $a \rightarrow b$. R, S y E forman juntos una región cuyo encabezamiento es el encabezamiento de R. Para demostrarlo hay que comprobar que el encabezamiento de R domina a todos los nodos de S. Si no, debe haber algún

camino al encabezamiento de S que no termine con una arista de E. Entonces habría que representar la última arista de este camino en el grafo de flujo en curso mediante alguna otra arista que entre a b. Pero no puede haber dicha arista, o T_2 no se puede utilizar para consumir b.

Ejemplo 10.38. El nodo etiquetado con cd de la figura 10.51(b) representa la región mostrada en la figura 10.51(a), que se formó al haber consumido c a d. Obsérvese que la arista $d \rightarrow c$ no es parte de la región; en la figura 10.50(b) esa arista se representa mediante el lazo en cd. Sin embargo, en la figura 10.50(c), se ha eliminado la arista $cd \rightarrow cd$ y el nodo cd representa ahora la región que se muestra en la figura 10.51(b).

En la figura 10.50(d), el nodo cd sigue representando la región de la figura 10.51(b), en tanto que el nodo ab representa la región de la figura 10.51(c). La arista $ab \rightarrow cd$ en la figura 10.50(d) representa las aristas $a \rightarrow c$ y $b \rightarrow c$ del grafo de flujo original de la figura 10.50(a). Cuando se aplica T_2 para alcanzar la figura 10.50(e), el nodo restante representa todo el grafo de flujo, figura 10.50(a). □

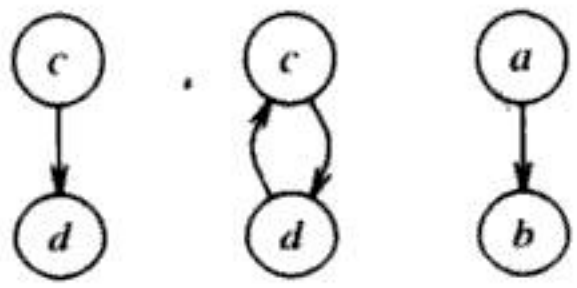


Fig. 10.51. Algunas regiones.

Se debe observar que la propiedad de la reducción T_1 y T_2 anteriormente mencionada se cumple también para el análisis de intervalos. Se deja como ejercicio el hecho de que a medida que se construye I(G), I(I(G)), y así sucesivamente, cada nodo en cada uno de estos grafos representa una región y cada arista representa un conjunto de aristas que cumplen la propiedad 2 anterior.

Encuentro de dominadores

Se cierra esta sección con un algoritmo eficiente para un concepto que se ha utilizado mucho y se seguirá utilizando al desarrollar la teoría de grafos de flujo y análisis de flujo de datos. Se dará un algoritmo simple para calcular los dominadores en cada nodo n en un grafo de flujo, basado en el principio de que si $p_1, p_2, \ldots, p_k$ son todos los predecesores de n, y $d \neq n$, entonces d *dom* n si, y sólo si, d *dom* p_i para cada i. El método es semejante al análisis de flujo de datos de avance con intersección como el operador de confluencia (por ejemplo, las expresiones disponibles), en que se toma una aproximación del conjunto de dominantes de n y se refina visitando repetidamente por turno todos los nodos.

En este caso, la aproximación inicial que se elija tiene el nodo inicial dominado únicamente por el nodo inicial, y todo domina a todo además del nodo inicial. In-

tuitivamente, la razón por la que este enfoque funciona es que los candidatos a dominadores son descartados sólo cuando se encuentra un camino que demuestre que, por ejemplo, *m dom n* es falso. Si no se puede encontrar dicho camino desde el nodo inicial a *n* evitando *m*, entonces *m* es en realidad un dominador de *n*.

Algoritmo 10.16. Encuentro de dominadores.

Entrada. Un grafo de flujo G con conjunto de nodos N, conjunto de aristas E y nodo inicial n_0.

Salida. La relación *dom*.

Método. Se calcula $D(n)$, el conjunto de dominadores de n, iterativamente mediante el procedimiento de la figura 10.52. Al final, d está en $D(n)$ si, y sólo si, *d dom n*. El lector puede proporcionar los detalles relativos a cómo se detectan los cambios a $D(n)$; el algoritmo 10.2 servirá como modelo.

```
(1)  D(n0) := {n0};
(2)  for n en N − {n0} do D(N) := N;
     /* termina la inicialización */
(3)  while ocurran cambios a cualquiera D(n) do
(4)      for n en N − {n0} do
(5)          D(n) := {n} ∪ ( ∩ D(p) );
                          p un predecesor de n
```

Fig. 10.52. Algoritmo para el cálculo de dominadores.

Se puede demostrar que el $D(n)$ calculado en la línea (5) de la figura 10.52 es siempre un subconjunto del $D(n)$ en curso. Como $D(n)$ no puede hacerse más pequeño indefinidamente, en algún momento se debe terminar el lazo **while**. Se deja al lector interesado la demostración de que, después de la convergencia, $D(n)$ es el conjunto de dominadores de n. El algoritmo de la figura 10.52 es bastante eficiente, ya que $D(n)$ se puede representar mediante un vector de bits y el conjunto de operaciones de la línea (5) se puede realizar con las operaciones lógicas **and** y **or**. □

Ejemplo 10.39. Considérese de nuevo el grafo de flujo de la figura 10.45, y supóngase que en el lazo **for** de la línea (4) los nodos se visitan en orden numérico. El nodo 2 sólo tiene a 1 como predecesor, así que $D(2) := \{2\} \cup D(1)$. Como 1 es el nodo inicial, a $D(1)$ se le asignó $\{1\}$ en la línea (1). Por tanto, a $D(2)$ se le asigna $\{1,2\}$ en la línea (5).

Después se considera el nodo 3, con predecesores 1, 2, 4 y 8. La línea (5) da $D(3) = \{3\} \cup (\{1\} \cap \{1,2\} \cap \{1,2,\ldots,10\}) = \{1,3\}$. Los cálculos restantes son:

$$D(4) = \{4\} \cup (D(3) \cap D(7)) = \{4\} \cup (\{1,3\} \cap \{1,2,\ldots,10\}) = \{1,3,4\}$$

$$D(5) = \{5\} \cup D(4) = \{5\} \cup \{1,3,4\} = \{1,3,4,5\}$$

$$D(6) = \{6\} \cup D(4) = \{6\} \cup \{1,3,4\} = \{1,3,4,6\}$$

$$D(7) = \{7\} \cup (D(5) \cap D(6) \cap D(10))$$
$$= \{7\} \cup (\{1,3,4,5\} \cap \{1,3,4,6\} \cap \{1,2, \ldots, 10\}) = \{1,3,4,7\}$$
$$D(8) = \{8\} \cup D(7) = \{8\} \cup \{1,3,4,7\} = \{1,3,4,7,8\}$$
$$D(9) = \{9\} \cup D(8) = \{9\} \cup \{1,3,4,7,8\} = \{1,3,4,7,8,9\}$$
$$D(10) = \{10\} \cup D(8) = \{10\} \cup \{1,3,4,7,8\} = \{1,3,4,7,8,10\}$$

Se ve que la segunda pasada por el lazo **while** no produce cambios, así que los valores anteriores producen la relación *dom*. □

10.10 ALGORITMOS EFICIENTES PARA EL FLUJO DE DATOS

En esta sección se considerarán dos formas de utilizar la teoría de grafos para acelerar el análisis de flujo de datos. La primera es una aplicación del ordenamiento en profundidad para reducir el número de pasadas que realizan los algoritmos iterativos de la sección 10.6, y la segunda utiliza los intervalos o las transformaciones T_1 y T_2 para generalizar el enfoque dirigido por la sintaxis de la sección 10.5.

Ordenamiento en profundidad en algoritmos iterativos

En todos los problemas estudiados hasta ahora, como las definiciones de alcance, las expresiones disponibles o las variables activas, cualquier suceso significativo en un nodo se propagará a ese nodo a lo largo de un camino acíclico. Por ejemplo, si una definición *d* está en *ent* [*B*], entonces hay algún camino acíclico desde el bloque que contiene *d* hasta *B* tal que *d* está en todos los conjuntos *ent* y *sal* a lo largo de ese camino. De manera similar, si una expresión x+y no está disponible a la entrada al bloque *B*, entonces hay un camino acíclico que demuestre ese hecho; o el camino va desde el nodo inicial y no incluye proposiciones que desactiven o generen x+y, o el camino va desde un bloque que desactiva x+y y a lo largo del camino no hay generación posterior de x+y. Por último, para las variables activas, si x está activa a la salida del bloque *B*, entonces hay un camino acíclico desde *B* hasta un uso de x, a lo largo del cual no hay definiciones de x.

El lector debe comprobar que en cada uno de estos casos, los caminos con lazos no añadan nada. Por ejemplo, si se alcanza un uso de x desde el fin del bloque *B* a lo largo de un camino con un lazo, se puede eliminar dicho lazo para encontrar un camino más corto a lo largo del cual el uso de x todavía se alcance desde *B*.

Si toda la informacion útil se propaga a lo largo de caminos acíclicos, existe la posibilidad de adaptar el orden en que se visiten los nodos en los algoritmos de flujo de datos iterativos de modo que, después de relativamente pocas pasadas a través de los nodos, se pueda estar seguro de que la información ha pasado a lo largo de todos los caminos acíclicos. En concreto, las estadísticas recogidas en Knuth [1971b] muestran que los grafos del flujo típicos tienen una *profundidad de intervalo* muy baja, que es el número de veces que uno debe aplicar la partición de intervalos para alcanzar el grafo de flujo límite; se encontró un promedio de 2.75. Además, se puede demostrar que la profundidad de intervalo de un grafo de flujo nunca es menor que

lo que aquí se ha llamado "profundidad", el número máximo de aristas de retroceso en cualquier camino acíclico. (Si el grafo de flujo no es reducible, la profundidad puede depender del árbol de expansión en profundidad elegido.)

Recordando el análisis del árbol de expansión en profundidad de la sección anterior, se observa que si $a \rightarrow b$ es una arista, entonces el número en profundidad de b es menor que el de a sólo cuando la arista es una arista de retroceso. Por tanto, se sustituye la línea (5) de la figura 10.26, que indica que se visite cada bloque B del grafo de flujo para el cual se estén calculando las definiciones de alcance, por:

for cada bloque B en orden en profundidad **do**

Supóngase que se tiene un camino a lo largo del cual se propaga una definición d, como

$$3 \rightarrow 5 \rightarrow 19 \rightarrow 35 \rightarrow 16 \rightarrow 23 \rightarrow 45 \rightarrow 4 \rightarrow 10 \rightarrow 17$$

donde los enteros representan los números en profundidad de los bloques a lo largo del camino. Entonces la primera vez que se pase por el lazo de las líneas (5) a (9) de la figura 10.26, d se propagará desde *sal* [3] a *ent* [5] a *sal* [5], y así sucesivamente, hasta *sal* [35]. No se alcanzará *ent* [16] en ese recorrido porque como 16 precede a 35, ya se había calculado *ent* [16] cuando d se puso en *sal* [35]. Sin embargo, la próxima vez que se recorre el lazo de las líneas (5) a (9), cuando se calcule *ent* [16], se incluirá d porque está en *sal* [35]. La definición d también se propagará a *sal* [16], *ent* [23], y así sucesivamente, hasta *sal* [45], donde debe esperar porque ya se ha calculado *ent* [4]. En la tercera pasada, d viaja a *ent* [4], *sal* [4], *ent* [10], *sal* [10] y *ent* [17], de modo que después de tres pasadas se ha establecido que d alcanza el bloque 17[13].

No debería ser difícil extraer el principio general de este ejemplo. Si se utiliza orden en profundidad en la figura 10.26, entonces el número de pasadas necesarias para propagar cualquier definición de alcance a lo largo de cualquier camino acíclico no es sólo uno más que el número de aristas a lo largo del camino que va desde un bloque con un número mayor a uno con un número menor. Estas aristas son exactamente las aristas de retroceso, así que el número de pasadas necesarias es uno más la profundidad. Por supuesto, el algoritmo 10.2 no detecta el hecho de que todas las definiciones han alcanzado todo aquello que pueden alcanzar con una pasada más, así que el límite superior sobre el número de pasadas empleadas por el algoritmo con ordenamiento de bloques en profundidad es realmente dos más la profundidad, o 5 si se cree que los resultados de Knuth [1971b] son típicos.

El ordenamiento en profundidad es ventajoso también para las expresiones disponibles (Algoritmo 10.3), o cualquier problema de flujo de datos que se resolvió mediante la propagación hacia adelante. Para problemas como las variables activas, donde se propaga hacia atrás, se puede lograr el mismo promedio de cinco pasadas si se elige el contrario del orden en profundidad. Por tanto, se puede propagar un uso de una variable en el bloque 17 hacia atrás a lo largo del camino

$$3 \rightarrow 5 \rightarrow 19 \rightarrow 35 \rightarrow 16 \rightarrow 23 \rightarrow 45 \rightarrow 4 \rightarrow 10 \rightarrow 17$$

---

[13] La definición d también alcanza *sal* [17], pero esto es irrelevante para el camino en cuestión.

en una pasada a *ent* [4], donde hay que esperar hasta la siguiente pasada para alcanzar en orden *sal* [45]. En la segunda pasada alcanza *ent* [16] y en la tercera pasada va desde *sal* [35] hasta *sal* [3]. En general, uno más las pasadas de profundidad basta para llevar el uso de una variable de vuelta hacia atrás, a lo largo de un camino acíclico, si se elige el orden en profundidad en sentido inverso para visitar los nodos en una pasada, porque entonces, los usos se propagan a lo largo de cualquier secuencia decreciente en una sola pasada.

Análisis de flujo de datos basado en la estructura

Con un poco más de esfuerzo, se pueden implantar algoritmos de flujo de datos que visiten nodos (y apliquen ecuaciones de flujo de datos) no más veces que la profundidad de intervalo del grafo de flujo, y con frecuencia el nodo promedio será visitado incluso menos veces. No se ha establecido firmemente si el esfuerzo adicional da como resutlado un ahorro real de tiempo, pero se ha utilizado una técnica como ésta, basada en el análisis de intervalos, en varios compiladores. Además, las ideas que aquí se exponen se aplican a algoritmos de flujo de datos dirigidos por la sintaxis para todos los tipos de proposiciones de control estrucutradas, no sólo las proposiciones **if** ... **then** y **do** ... **while** que se estudiaron en la sección 10.5, y han aparecido también en varios compiladores.

Los algoritmos se basarán en la estructura inducida sobre los grafos de flujo por las transformaciones T_1 y T_2. Como en la sección 10.5, se tratan las definiciones que se generan y se desactivan cuando el control fluye a través de una región. A diferencia de las regiones definidas por las proposiciones **if** o **while**, una región general puede tener múltiples salidas, de modo que para cada bloque B en la región R se calcularán los conjuntos $gen_{R,B}$ y $desact_{R,B}$ de definiciones generadas y desactivadas, respectivamente, a lo largo de los caminos dentro de la región desde el encabezamiento hasta el final del bloque B. Estos conjuntos se utilizarán para definir una *función de* transferencia $trans_{R,B}(S)$ que indica para cualquier conjunto S de definiciones, el conjunto de definiciones que alcanza el final del bloque B viajando a lo largo de caminos totalmente dentro de R, dado que todas y sólo las definiciones en S alcancen el encabezamiento de R.

Como se vio en las secciones 10.5 y 10.6, las definiciones que alcanzan el final del bloque B pertenecen a dos clases:

1. Aquellas generadas dentro de R y que se propagan al final de B independiente de S.

2. Aquellas que no se generan en R, pero que tampoco son desactivadas a lo largo de un camino desde el encabezamiento de R al final de B, y por tanto están en $trans_{R,B}(S)$ si, y sólo si, están en S.

Por tanto, se puede escribir *trans* de la forma

$$trans_{R,B}(S) = gen_{R,B} \cup (S - desact_{R,B})$$

El núcleo del algoritmo es una forma de calcular $trans_{R,B}$ para regiones cada vez mayores definidas por alguna descomposición $(T_1 - T_2)$ de un grafo de flujo. Por el momento, se supone que el grafo de flujo es reducible, aunque una simple modificación permite que el algoritmo funcione también para grafos no reducibles.

La base es una región que consta de un solo bloque, B. Aquí, la función de transferencia de la región es la función de transferencia del bloque mismo, ya que una definición alcanza el final del bloque si, y sólo si, es generada por el bloque o está en el conjunto S y no está desactivada. Es decir,

$$gen_{B,B} = gen\,[B]$$

$$desact_{B,B} = desact\,[B]$$

Considérese ahora la construcción de una región R por T_2; es decir, R se forma cuando R_1 consume R_2, como se sugiere en la figura 10.53. Primero, obsérvese que dentro de la región R no hay aristas desde R_2 de vuelta a R_1 puesto que las aristas desde R_2 al encabezamiento de R_1 no son parte de R. Por tanto, cualquier camino que esté totalmente dentro de R va primero (opcionalmente) a través de R_1, después (opcionalmente) a través de R_2, pero después no puede regresar a R_1. Asimismo, obsérvese que el encabezamiento de R es el encabezamiento de R_1. Se puede concluir que dentro de R, R_2 no afecta a la función de nodos en R_1; es decir,

$$gen_{R,B} = gen_{R_1,B}$$

$$desact_{R,B} = desact_{R_1,B}$$

para toda B en R_1.

Para B en R_2, una definición puede alcanzar el final de B si se cumple cualquiera de las siguientes definiciones.

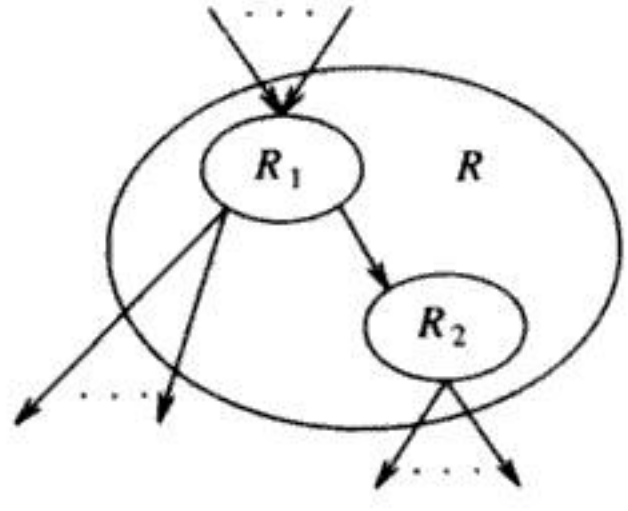


Fig. 10.53. Construcción de regiones por medio de T_2.

1. La definición se genera dentro de R_2.
2. La definición se genera dentro de R_1, alcanza el final de algún predecesor del encabezamiento de R_2, y no se desactiva yendo del encabezamiento de R_2 a B.
3. La definición está en el conjunto S disponible en el encabezamiento de R_1, no se desactiva yendo a algún predecesor del encabezamiento de R_2, y no se desactiva yendo desde el encabezamiento de R_2 a B.

Por tanto, las definiciones que alcanzan los fines de aquellos bloques de R_1 que sean predecesores del encabezamiento de R_2 desempeñan un papel especial. Fundamentalmente, se ve lo que sucede a un conjunto S que entra al encabezamiento de R_1 cuando sus definiciones intentan alcanzar el encabezamiento de R_2 a través de uno

de sus predecesores. El conjunto de definiciones que alcanzan uno de los predecesores del encabezamiento de R_2 se convierte en el conjunto de entrada para R_2, y se aplican las funciones de transferencia para R_2 a dicho conjunto.

Por tanto, sea G la unión de $gen_{R_1,P}$ para todos los predecesores P del encabezamiento de R_2, y sea K la intersección de $desact_{R_1,P}$ para todos esos predecesores P. Entonces, si S es el conjunto de definiciones que alcanzan el encabezamiento de R_1, el conjunto de definiciones que alcanzan el encabezamiento de R_2 a lo largo de caminos que se encuentren totalmente dentro de R es $G \cup (S-K)$. Por tanto, la función de transferencia en R para aquellos bloques B en R_2 se puede calcular mediante

$$gen_{R,B} = gen_{R_2,B} \cup (G - desact_{R_2,B})$$

$$desact_{R,B} = desact_{R_2,B} \cup (K - gen_{R_2,B})$$

A continuación, considérese lo que sucede cuando una región R se construye a partir de una región R_1 utilizando la transformación T_1. La situación general se muestra en la figura 10.54; obsérvese que R consta de R_1 más algunas aristas de retroceso hacia el encabezamiento de R_1 (que también es el encabezamiento de R, por supuesto). Un camino que vaya dos veces a través del encabezamiento debe ser cíclico y como ya se vio, no necesita analizarse. Así, todas las definiciones generadas al final del bloque B se generan de una de dos maneras.

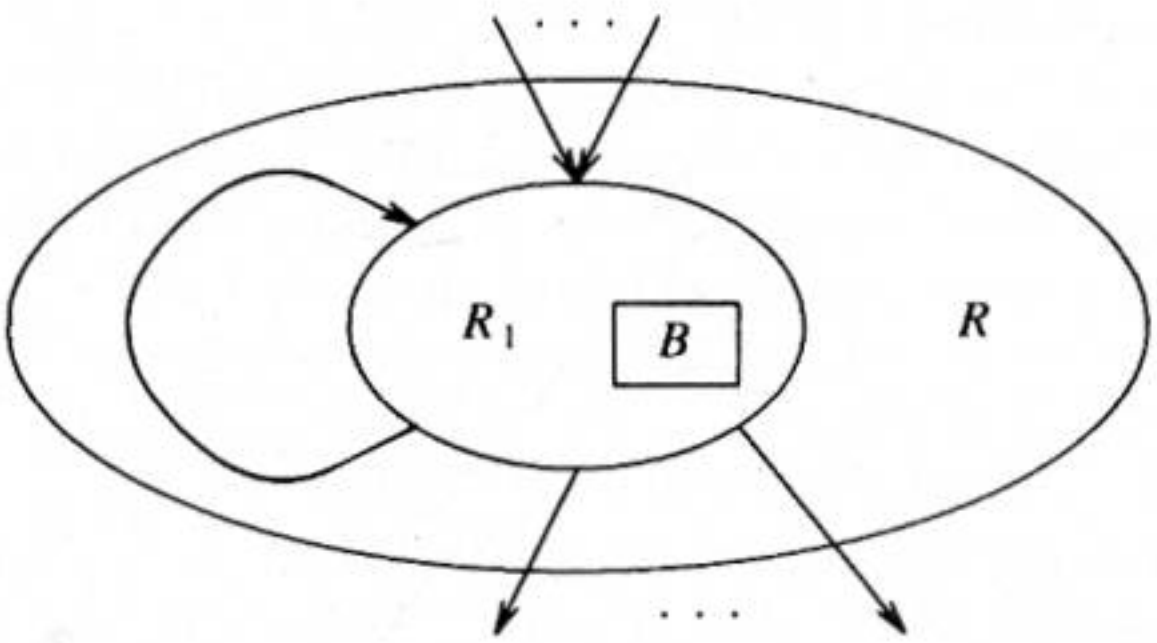


Fig. 10.54. Construcción de regiones por medio de T_1.

1. La definición se genera dentro de R_1 y no necesita las aristas de retroceso incorporadas a R para alcanzar el fin de B.
2. La definición se genera en algún lugar dentro de R_1, alcanza un predecesor del encabezamiento, sigue una arista de retroceso, y no se desactiva yendo desde el encabezamiento a B.

Si G es la unión de $gen_{R_1,P}$ para todos los predecesores del encabezamiento en R, entonces

$$gen_{R,B} = gen_{R_1,B} \cup (G - desact_{R_1,B})$$

Una definición se desactiva yendo desde el encabezamiento a B si, y sólo si, se

desactiva a lo largo de todos los caminos acíclicos, de modo que las aristas de retroceso incorporadas a R no hacen que se desactiven más definiciones. Es decir,

$$desact_{R,B} = desact_{R_1,B}$$

Ejemplo 10.40. Reconsidérese el grafo de flujo de la figura 10.50, cuya descomposición (T_1, T_2) se muestra en la figura 10.55, con nombres para las regiones de la descomposición. También se muestran en la figura 10.56 algunos vectores de bits hipotéticos que representan tres definiciones y si son generados o desactivados por cada uno de los bloques de la figura 10.55.

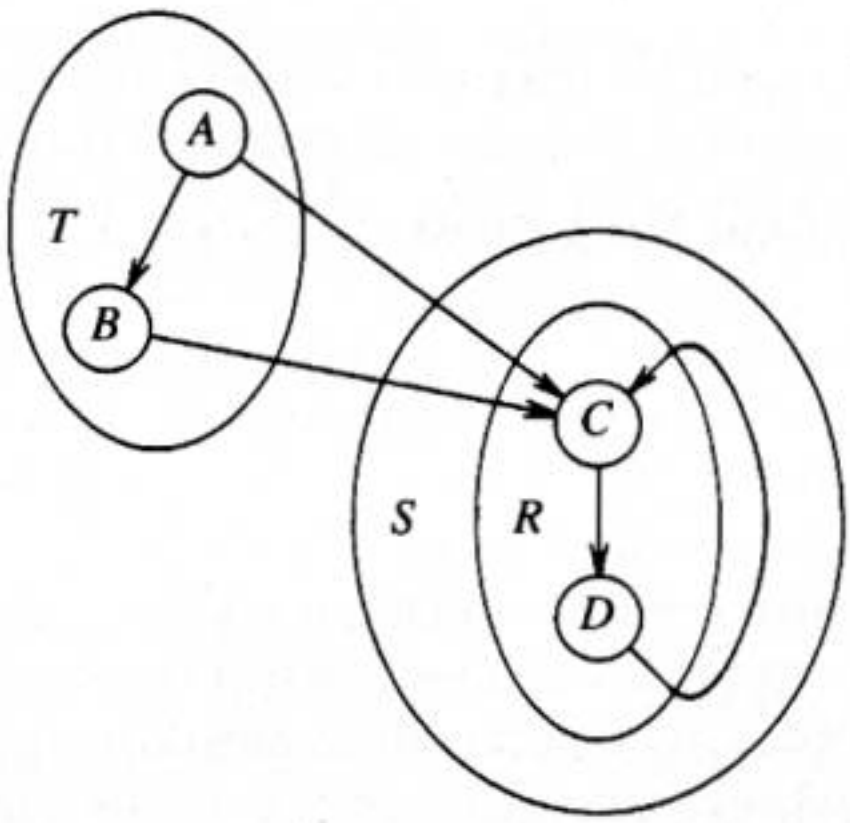


Fig. 10.55. Descomposición de un grafo de flujo.

Comenzando de dentro hacia fuera, se observa que para regiones de un solo nodo, que se llaman A, B, C y D, *gen* y *desact* vienen dados por la tabla de la figura 10.56. Después se puede pasar a la región R, que se forma cuando C consume D por medio de T_2. Siguiendo las reglas anteriores para T_2, se observa que *gen* y *desact* no cambian para C, es decir,

$$gen_{R,C} = gen_{C,C} = 000$$
$$desact_{R,C} = desact_{C,C} = 010$$

Para el nodo D, hay que encontrar en la región C la unión de *gen* para todos los predecesores del encabezamiento de la región D. Por supuesto, el encabezamiento de la región D es el nodo D, y sólo hay un predecesor de ese nodo en la región C: el nodo C. Por tanto,

BLOQUE	*gen*	*desact*
A	100	010
B	010	101
C	000	010
D	001	000

Fig. 10.56. Información de *gen* y *desact* para los bloques de la figura 10.55.

$$gen_{R,D} = gen_{D,D} \cup (gen_{C,C} - desact_{D,D}) = 001 + (000 - 000) = 001$$
$$desact_{R,D} = desact_{D,D} \cup (desact_{C,C} - gen_{D,D}) = 000 + (010 - 001) = 010$$

Ahora se construye la región S a partir de la región R por medio de T_1. Los conjuntos *desact* no cambian, así que se tiene

$$desact_{S,C} = desact_{R,C} = 010$$
$$desact_{S,D} = desact_{R,D} = 010$$

Para calcular los conjuntos *gen* para S se observa que la única arista de retroceso al encabezamiento de S que se incorpora yendo desde R a S es la arista $D \rightarrow C$. Por tanto,

$$gen_{S,C} = gen_{R,C} \cup (gen_{R,D} - desact_{R,C}) = 000 + (001 - 010) = 001$$
$$gen_{S,D} = gen_{R,D} \cup (gen_{R,D} - desact_{R,D}) = 001 + (001 - 010) = 001$$

El cálculo para la región T es análogo al de la región R y se obtiene

$$gen_{T,A} = 100$$
$$desact_{T,A} = 010$$
$$gen_{T,B} = 010$$
$$desact_{T,B} = 101$$

Por último, se calculan *gen* y *desact* para la región U, el grafo de flujo completo. Como U se construye cuando T consume S por la transformación T, los valores de *gen* y *desact* para los nodos A y B no son diferentes de los que se acaban de dar. Para C y D, se observa que el encabezamiento de S, el nodo C, tiene dos predecesores en la región T: A y B. Por tanto, se calcula

$$G = gen_{T,A} \cup gen_{T,B} = 110$$
$$D = desact_{T,A} \cap desact_{T,B} = 000$$

Después se puede calcular

$$gen_{U,C} = gen_{S,C} \cup (G - desact_{S,C}) = 101$$
$$desact_{U,C} = desact_{S,C} \cup (K - gen_{S,C} = 010$$
$$gen_{U,D} = gen_{S,D} \cup (G - desact_{S,D}) = 101$$
$$desact_{U,D} = desact_{S,D} \cup (K - gen_{S,D}) = 010$$ □

Habiendo calculado $gen_{U,B}$ y $desact_{U,B}$ para cada bloque B, donde U es la región que consta del grafo de flujo completo, se ha calculado fundamentalmente *sal* [B] para cada bloque B. Es decir, si se observa la definición de $trans_{U,B}(S) = gen_{U,B} \cup \cup (S - desact_{U,B})$, se ve que $trans_{U,B}(\varnothing)$ es exactamente *sal* [B]. Pero $trans_{U,B}(\varnothing) = = gen_{U,B}$. Por tanto, terminar el algoritmo de definiciones de alcance basado en la estructura consiste en utilizar los conjuntos *gen* como los *sal*, y calcular los conjuntos *ent* tomando la unión de los conjuntos *sal* de los predecesores. Estos pasos se resumen el el siguiente algoritmo.

Algoritmo 10.17. Definiciones de alcance basadas en la estructura.

Entrada. Un grafo de flujo reducible G y los conjuntos de definiciones *gen* [B] y *desact* [B] para cada bloque B de G.

Salida. *ent* [*B*] para cada bloque *B*.

Método.

1. Encuéntrese la descomposición $(T_1 - T_2)$ para G.
2. Para cada región R en la descomposición, de dentro hacia fuera, calcúlese $gen_{R,B}$ y $desact_{R,B}$ para cada bloque B en R.
3. Si U es el nombre de la región que consta del grafo completo, entonces para cada bloque B, asígnese *ent* [*B*] a la unión, entre todos los predecesores P del bloque B, de $gen_{U,P}$. □

Algunos aumentos de velocidad para el algoritmo basado en la estructura

Primero, obsérvese que si se tiene una función de transferencia $G \cup (S-K)$, la función no se modifica si se eliminan de K algunos miembros de G. Así, cuando se aplique T_2, en lugar de utilizar las fórmulas

$$gen_{R,B} = gen_{R_2,B} \cup (G - desact_{R_2,B})$$
$$desact_{R,B} = desact_{R_2,B} \cup (K - gen_{R_2,B})$$

se puede sustituir la segunda por

$$desact_{R,B} = desact_{R_2,B} \cup K$$

ahorrando una operación para cada bloque en la región R_2.

Otra idea útil es observar que la única vez que se aplica T_1 es después de haber consumido alguna región R_2 por R_1, y haya algunas aristas de retroceso desde R_2 al encabezamiento de R_1. En lugar de realizar primero cambios en R_2 debidos a la operación T_2, y después hacer los cambios en R_1 y R_2 debidos a la operación T_1, se pueden combinar los dos conjuntos de cambios si se hace lo siguiente:

1. Utilizando la regla T_2, calcúlese la nueva función de transferencia para aquellos nodos de R_2 que sean predecesores del encabezamiento de R_1.
2. Utilizando la regla T_1, calcúlese la nueva función de transferencia para todos los nodos de R_1.
3. Utilizando la regla T_2, calcúlese la nueva función de transferencia para todos los nodos de R_2. Obsérvese que la regeneración debida a la aplicación de T_1 ha alcanzado los predecesores de R_2 y se pasan a todos los de R_2 mediante la regla T_2; no hay necesidad de aplicar la regla T_1 para R_2.

Manejo de grafos de flujo no reducibles

Si la reducción $(T_1 - T_2)$ de un grafo de flujo se detiene en un grafo de flujo límite que no sea un solo nodo, entonces se debe realizar una separación de nodos. Separar un nodo del grafo de flujo límite corresponde a duplicar toda la región representada por ese nodo. Por ejemplo, en la figura 10.57 se sugiere el efecto que la separación de nodos puede tener sobre un grafo de flujo original de nueve nodos que fue particionado por T_1 y T_2 en tres regiones conectadas por algunas aristas.

Como se mencionó en la sección anterior, alternando separaciones con secuencias de reducciones, se tiene la seguridad de que el grafo de flujo se reduce a un solo

nodo. El resultado de las separaciones es que algunos de los nodos del grafo original tendrán más de una copia en la región representada por el grafo de un nodo. Se puede aplicar el algoritmo 10.17 a esta región con pocos cambios. La única diferencia es que cuando se separe un nodo, se deben duplicar los conjuntos *gen* y *desact* correspondientes a los nodos del grafo original en la región representada por el nodo separado. Por ejemplo, cualesquiera que sean los valores de *gen* y *desact* para los nodos de la región de dos nodos de la figura 10.57 de la izquierda se convierten en *gen* y *desact* para cada uno de los nodos correspondientes en ambas regiones de dos nodos de la derecha. En el paso final, cuando se calculen los conjuntos *ent* para todos los nodos, los nodos del grafo original que tengan varios representantes en la región final tendrán sus conjuntos *ent* calculados tomando la unión de los conjuntos *ent* de todos sus representantes.

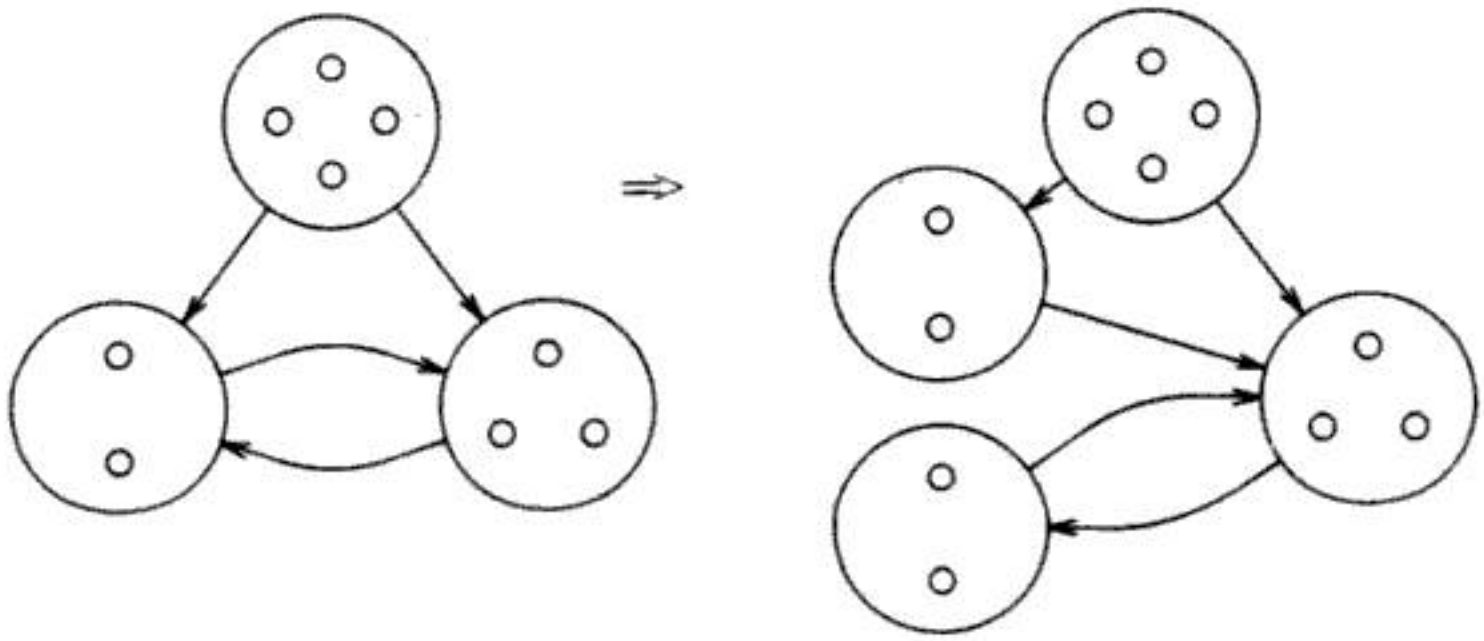


Fig. 10.57. Separación de un grafo de flujo no reducible.

En el peor caso, la separación de nodos podría exponenciar el número total de nodos representados por todas las regiones. Por tanto, si se espera que muchos grafos de flujo sean no reducibles, probablemente no se deberían utilizar métodos basados en la estructura. Por fortuna, los grafos de flujo no reducibles son lo suficientemente raros como para no tener en cuenta el costo de la separación de nodos.

10.11 UNA HERRAMIENTA PARA EL ANALISIS DEL FLUJO DE DATOS

Como ya se ha mencionado, existen grandes similitudes entre los distintos problemas de flujo de datos estudiados. Se vio que las ecuaciones de flujo de datos de la sección 10.6 se distinguían por:

1. La función de transferencia utilizada, que era en cada caso estudiado de la forma $f(X) = A \cup (X - B)$. Por ejemplo, $A = desact$ y $B = gen$ para las definiciones de alcance.
2. El operador de confluencia utilizado, que en cada caso estudiado era la unión o la intersección.
3. La dirección de propagación de la información: hacia adelante o hacia atrás.

Como estas distinciones no son grandes, no debe sorprender que todos estos problemas se consideren de forma unificada. Dicho enfoque fue descrito por Kildall [1973], y Kildall aplicó una herramienta para simplificar la implantación de problemas de flujo de datos y la utilizó para varios proyectos de compiladores. No ha sido muy utilizada, probablemente porque la cantidad de trabajo ahorrada por el sistema no es tan grande como la ahorrada por herramientas como los generadores de analizadores sintácticos. Sin embargo, hay que saber lo que se puede hacer no sólo porque sugiere una simplificación a los implantadores de compiladores optimadores, sino también porque ayuda a unificar las distintas ideas que se han visto antes en este capítulo. Además, esta sección sugiere cómo se pueden desarrollar estrategias de análisis del flujo de datos más poderosas que proporcionan información más precisa que los algoritmos mencionados hasta ahora.

Marcos para análisis de flujo de datos

Se describirán marcos que modelan los problemas de propagación hacia adelante. Si sólo se considera el tipo iterativo de solución a los problemas de flujo de datos, entonces no importa la dirección del flujo; se puede invertir la dirección de las aristas y hacer algunos ajustes menores para el nodo inicial, y entonces considerar un problema de propagación hacia atrás como si fuera hacia adelante. Los algoritmos basados en la estructura son algo distintos; los problemas de propagación hacia adelante y hacia atrás no se resuelven de la misma forma porque el inverso de un grafo de flujo reducible no tiene por qué ser reducible. Sin embargo, el tratamiento de los problemas de propagación hacia atrás se dejará como ejercicio y la atención se concentrará en problemas de propagación hacia adelante.

Un *marco para análisis de flujo de datos* consta de:

1. Un conjunto V de *valores* que deben ser propagados. Los valores de *ent* y *sal* son miembros de V.
2. Un conjunto F de *funciones de transferencia* de V a V.
3. Una *operación de reunión* binaria $\wedge$, sobre V, para representar al operador de confluencia.

Ejemplo 10.41. Para las definiciones de alcance, V consta de todos los subconjuntos del conjunto de definiciones en el programa. El conjunto F es el conjunto de todas las funciones de la forma $f(X) = A \cup (X - B)$, donde A y B son conjuntos de definiciones, es decir, miembros de V; A y B son lo que se llaman *gen* y *desact*, respectivamente. Por último, la operación $\wedge$ es la unión.

Para las expresiones disponibles, V consta de todos los subconjuntos del conjunto de expresiones calculadas por el programa, y F es el conjunto de expresiones de la misma forma que antes pero donde A y B son ahora conjuntos de expresiones. El operador de reunión es la intersección, por supuesto.

Ejemplo 10.42. El enfoque de Kildall no se limita a los ejemplos sencillos que se han venido considerando, aunque surge la complejidad, tanto en términos de tiempo de computación como de dificultad intelectual. Los ejercicios proponen un ejemplo

muy poderoso, donde la información del flujo de datos calculada indica en definitiva todos los pares de expresiones que tienen el mismo valor en un punto. Sin embargo, se apreciará la esencia de este ejemplo al proporcionar un método que indique las variables con valores constantes de forma que capta más información que las definiciones de alcance. El nuevo marco entiende, por ejemplo, que cuando x es definida por *d*: `x:=x+1`, y x tenía un valor constante antes de la asignación, lo seguirá teniendo después.

Por el contrario, si se utilizan las definiciones de alcance para la propagación de constante, se vería que la proposición *d* era una posible definición de x, y se supone por tanto que x no tenía un valor constante. Por supuesto, en una pasada, el lado derecho de *d*: `x:=x+1` puede ser sustituido por una constante y entonces otra ronda de propagación de constantes detectaría que los usos de x definidos en *d* eran en realidad usos de una constante.

En el nuevo marco, el conjunto V es el conjunto de todas las transformaciones de variables del programa a un conjunto particular de valores. Ese conjunto de valores consta de:

1. Todas las constantes.

2. El valor *noconst*, que significa que se ha determinado que la variable en cuestión no tiene un valor constante. El valor *noconst* se asignaría a la variable x si, por ejemplo, durante el análisis de flujo de datos se descubrieran dos caminos a lo largo de los cuales 2 y 3, respectivamente, se asignaran a x, o un camino a lo largo del cual la definición previa de x fuera una proposición de lectura, **read**.

3. La variable *indef*, que significa que no se puede afirmar nada todavía acerca de la variable en cuestión, presumiblemente porque al principio de la ejecución del análisis del flujo de datos no se ha descubierto ninguna definición de la variable que alcance el punto en cuestión.

Obsérvese que *noconst* e *indef* no son lo mismo; son fundamentalmente opuestas. La primera indica que se han visto tantas formas en que podría ser definida una variable que se sabe que no es constante. La segunda indica que se ha visto tan poco acerca de la variable, que no se puede decir nada en absoluto.

La operación de reunión se define mediante la siguiente tabla. Se define a μ y ν como dos miembros de V; es decir, μ y ν hacen corresponder cada variable a una constante, con *indef* o con *noconst*. Entonces la función $\rho = \mu \wedge \nu$ se define en la figura 10.58, donde se da el valor de $\rho(\mathbf{x})$ en términos de los valores de $\mu(\mathbf{x})$ y $\nu(\mathbf{x})$ para cada variable x. En la tabla, *c* es una constante arbitraria y *d* es otra constante distinta de *c*. Por ejemplo, si $\mu(\mathbf{x}) = c$ y $\nu(\mathbf{x}) = d$, una constante distinta, entonces aparentemente x toma los valores *c* y *d* a lo largo de dos caminos distintos y en la confluencia de esos caminos, x no tiene un valor constante; de aquí la elección de $\rho(\mathbf{x}) = noconst$. Otro ejemplo: si a lo largo de un camino no se sabe nada acerca de x, lo cual se refleja por $\mu(\mathbf{x}) = indef$ y a lo largo de otro camino, se cree que x tiene el valor *c*, entonces después de la confluencia de estos caminos sólo se puede afirmar que x tiene el valor *c*. Por supuesto, el descubrimiento posterior de otro camino al

punto de confluencia, a lo largo del cual x tenía un valor además de c, cambiará el valor asignado para x después de la confluencia de *noconst*.

Por último, se debe diseñar el conjunto de funciones F que refleje la transferencia de información del comienzo al final de cualquier bloque. La descripción de este conjunto de funciones es complicada, aunque las ideas son sencillas. Por tanto, se dará una "base" para el conjunto de funciones describiendo las funciones que representan proposiciones de definición simples, y el conjunto completo de funciones se puede construir entonces componiendo funciones a partir de este conjunto base para reflejar los bloques con más de una proposición de definición.

$\nu(x)$	$\mu(x)$			
	noconst	c	$d\ (\neq c)$	*indef*
noconst	*noconst*	*noconst*	*noconst*	*noconst*
c	*noconst*	c	*noconst*	c
indef	*noconst*	c	d	*indef*

Fig. 10.58. $\rho(x)$ en términos de $\mu(x)$ y $\nu(x)$.

1. La función identidad está en F; esta función refleja cualquier bloque que no tenga proposiciones de definición. Si I es la función identidad, y μ es cualquier correspondencia de variables a valores, entonces $I(\mu) = \mu$. Obsérvese que μ misma no tiene por qué ser la identidad; es arbitraria.

2. Para cada variable x y constante c hay una función f en F tal que para cada correspondencia μ en V, se tiene $f(\mu) = \nu$, donde $\nu(w) = \mu(w)$ para toda w distinta de x, y $\nu(x) = c$. Estas funciones reflejan la acción de una proposición de asignación x:=c.

3. Para las tres variables x, y y z (no necesariamente distintas), hay una función f en F tal que para cada correspondencia μ en V, se tiene $f(\mu) = \nu$. La correspondencia ν se define por: para cada w distinta de x se tiene $\nu(w) = \mu(w)$ y $\nu(x) = \mu(y)+\mu(z)$. Si cualquiera de $\mu(y)$ o $\mu(z)$ es *noconst*, entonces la suma es *noconst*. Si cualquiera de $\mu(y)$ o $\mu(z)$ es *indef*, pero ninguna es *noconst*, entonces el resultado es *indef*. Esta función expresa el efecto de la asignación x:=y+z. Como es habitual en este capítulo, + se puede considerar como un operador genérico; aquí es necesaria una modificación obvia si el operador es unitario, ternario, o superior, y se necesita otra modificación obvia para tener en cuenta el efecto de una proposición de copia x:=y.

4. Para cada variable x hay una función f en F tal que para cada μ, $f(\mu) = \nu$, donde $\nu(w) = \mu(w)$ para toda w diferente de x, y $\nu(x) = noconst$. Esta función refleja una definición leyendo x, puesto que después de una proposición **read**, hay que asumir que x no tiene ningún valor constante en concreto.

Los axiomas de los marcos para análisis de flujo de datos

Para hacer que los tipos de algoritmos de flujo de datos que se han visto hasta ahora funcionen para un marco arbitrario, hay que suponer algunas cosas acerca del conjunto V, el conjunto de funciones F y el operador de reunión $\wedge$. Las suposiciones básicas se listan más abajo, aunque algunos algoritmos de flujo de datos necesitan suposiciones adicionales.

1. F tiene una función de identidad I, tal que $I(\mu) = \mu$ para toda μ en V.

2. F es cerrado bajo la composición; es decir, para cualesquiera dos funciones f y g en F, la función h definida por $h(\mu) = g(f(\mu))$ está en F.

3. $\wedge$ es una operación asociativa, conmutativa e idempotente. Estas tres propiedades se expresan algebraicamente como

$$\mu \wedge (\nu \wedge \rho) = (\mu \wedge \nu) \wedge \rho$$
$$\mu \wedge \nu = \nu \wedge \mu$$
$$\mu \wedge \mu = \mu$$

   para toda μ, ν y ρ en V.

4. Hay un *elemento tope* $\top$ en V, que satisface la ley

$$\top \wedge \mu = \mu$$

   para toda μ en V.

Ejemplo 10.43. Considérense las definiciones de alcance. Con seguridad F tiene la identidad, la función donde *gen* y *desact* son el conjunto vacío. Para mostrar la cerradura bajo la composición, supóngase que se tienen dos funciones

$$f_1(X) = G_1 \cup (X - K_1)$$
$$f_2(X) = G_2 \cup (X - K_2)$$

Entonces

$$f_2(f_1(X)) = G_2 \cup ((G_1 \cup (X - K_1)) - K_2)$$

Se puede comprobar que el lado derecho de lo anterior es algebraicamente igual a

$$(G_2 \cup (G_1 - K_2)) \cup (X - (K_1 \cup K_2))$$

Si se hace $K = K_1 \cup K_2$ y $G = (G_2 \cup (G_1 - K_2))$, entonces se ha demostrado que la composición de f_1 y f_2, que es $f(X) = G \cup (X - K)$, es de la forma que la convierte en un miembro de F.

Como el operador de reunión, que es la unión, es fácil comprobar que la unión es asociativa, conmutativa e idempotente. El elemento "tope" resulta ser el conjunto vacío en este caso, puesto que $\varnothing \cup X = X$ para cualquier conjunto X.

Cuando se consideran las expresiones disponibles, se ve que los mismos argumentos utilizados para las definiciones de alcance también demuestran que F tiene una intensidad y que es cerrado bajo la composición. El operador de reunión es ahora la intersección, pero este operador también es asociativo, conmutativo e idempo-

tente. El elemento tope tiene un sentido más intuitivo esta vez; es el conjunto E de todas las expresiones en el programa, puesto que para cualquier conjunto X de expresiones, $E \cap X = X$. □

Ejemplo 10.44. Considérese el marco para cálculos constantes que se presentó en el ejemplo 10.42. El conjunto de funciones F se diseñó para que incluyera la identidad y para que fuera cerrado bajo la composición. Para comprobar las leyes algebraicas para $\wedge$ basta con demostrar que se aplican a cada variable $\mathbf{x}$. Como ejemplo, se comprobará la idempotencia. Sea $\nu = \mu \wedge \mu$, es decir, para toda $\mathbf{x}$, $\nu(\mathbf{x}) = \mu(\mathbf{x}) \wedge \mu(\mathbf{x})$. Es sencillo comprobar por casos que $\nu(\mathbf{x}) = \mu(\mathbf{x})$. Por ejemplo, si $\mu(\mathbf{x}) = \textit{noconst}$, entonces $\nu(\mathbf{x}) = \textit{noconst}$, puesto que el resultado de emparejar *noconst* consigo mismo en la figura 10.58 es *noconst*.

Por último, el elemento tope es la correspondencia τ definida por $\tau(\mathbf{x}) = \textit{indef}$ para todas las variables $\mathbf{x}$. Se puede comprobar por la figura 10.58 que para cualquier correspondencia μ y cualquier variable $\mathbf{x}$, si ν es la función $\tau \wedge \mu$, entonces $\nu(\mathbf{x}) = \mu(\mathbf{x})$, puesto que el resultado de emparejar *indef* con cualquier valor en la tabla de la figura 10.58 es este otro valor. □

Monotonicidad y distributividad

Se necesita otra condición para hacer que funcione el algoritmo iterativo para análisis de flujo de datos. Esta condición, llamada monotonicidad, indica informalmente que si se toma cualquier función f del conjunto F y si se aplica f a dos miembros de V, uno "más grande" que el otro, entonces el resultado de aplicar f al más grande no es menos que lo que se obtiene al aplicar f al menor.

Para precisar la noción de "más grande", se define una relación $\leq$ en V mediante

$$\mu \leq \nu \text{ si, y sólo si, } \mu \wedge \nu = \mu$$

Ejemplo 10.45. En el marco de las definiciones de alcance, donde el operador de reunión es la unión y los miembros de V son conjuntos de definiciones, $X \leq Y$ significa que $X \cup Y = X$, es decir, X es un supraconjunto de Y. Así, $\leq$ mira "hacia atrás"; los elementos más pequeños de V son supraconjuntos de los más grandes.

Para las expresiones disponibles, donde el operador de reunión es la intersección, las cosas funcionan de la manera "correcta", y $X \leq Y$ significa que $X \cap Y = X$; es decir, X es un subconjunto de Y. □

Obsérvese en el ejemplo 10.45 que $\leq$, en el sentido dado aquí no tiene por qué tener todas las propiedades de $\leq$ en los enteros. Es cierto que $\leq$ es transitivo; el lector puede demostrar, como ejercicio sobre el uso de los axiomas para $\wedge$, que $\mu \leq \nu$ y $\nu \leq \rho$ implican que $\mu \leq \rho$. Sin embargo, $\leq$ en el sentido dado aquí no es un orden total. Por ejemplo, en el marco de las expresiones disponibles, se pueden tener dos conjuntos X e Y, ninguno de los cuales sea un subconjunto del otro, en cuyo caso ni $X \leq Y$ ni $Y \leq X$ sería verdadera.

A menudo ayuda dibujar el conjunto V en un *diagrama de retículo*, que es un grafo cuyos nodos son los elementos de V, y cuyas aristas están dirigidas hacia abajo, desde X a Y, si $Y \leq X$. Por ejemplo, en la figura 10.59 se muestra el conjunto V para un problema de flujo de datos de definiciones de alcance donde hay tres defi-

niciones, d_1, d_2 y d_3. Puesto que $\leq$ es "supraconjunto de", una arista se dirige hacia abajo desde cualquier subconjunto de estas tres definiciones a cada uno de sus supraconjuntos. Como $\leq$ es transitivo, se omite convencionalmente la arista de X a Y si queda otro camino de X a Y en el diagrama. Por tanto, aunque $\{d_1, d_2, d_3\} \leq \{d_1\}$, no se dibuja esta arista porque está representada por el camino a través de $\{d_1, d_2\}$, por ejemplo.

También es útil observar que se puede leer la reunión de estos diagramas, puesto que $X \wedge Y$ es siempre la Z más alta para la que hay caminos hacia abajo hasta Z desde X e Y. Por ejemplo, si X es $\{d_1\}$ e Y es $\{d_2\}$, entonces Z en la figura 10.59 es $\{d_1, d_2\}$, lo cual tiene sentido, porque el operador de reunión es la unión. También es cierto que el elemento tope aparecerá en el tope del diagrama de retículo; es decir, hay un camino hacia abajo desde $\top$ a cada elemento.

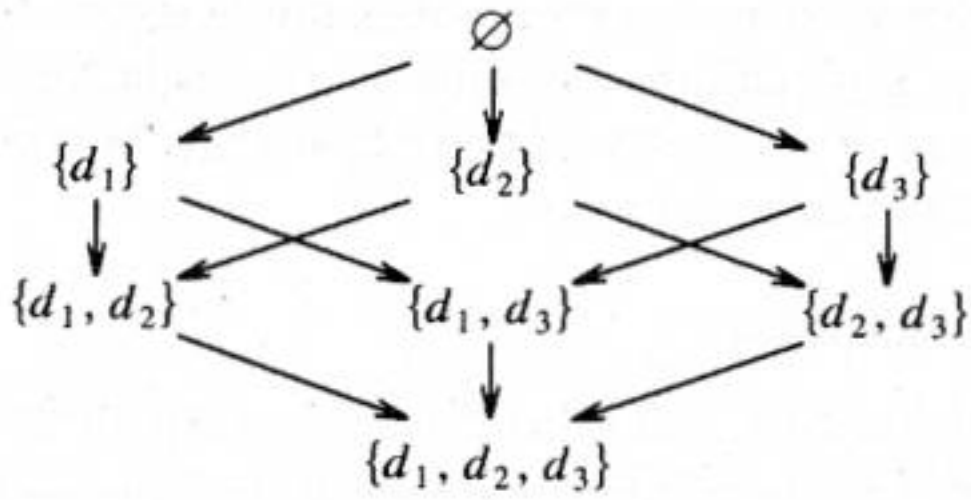


Fig. 10.59. Retículo de subconjuntos de definiciones.

Ahora se puede definir que un marco $(F, V, \wedge)$ es *monótono* si

$$\mu \leq \nu \text{ implica } f(\mu) \leq f(\nu) \tag{10.15}$$

para toda μ y ν en V y f en F.

Hay una forma equivalente de definir la monotonicidad:

$$f(\mu \wedge \nu) \leq f(\mu) \wedge f(\nu) \tag{10.16}$$

para toda μ y ν en V y f en F. Es útil saltar de una a otra en estas dos definiciones equivalentes, de modo que se esbozará una demostración de su equivalencia, dejando al lector que compruebe algunas observaciones sencillas, utilizando la definición de $\leq$ y las leyes asociativa, conmutativa y de idempotencia para $\wedge$.

Se asumirá (10.15) y se demostrará por qué se cumple (10.16). Primero, obsérvese que para cualesquiera μ y ν, se cumplen ambas $\mu \wedge \nu \leq \mu$ y $\mu \wedge \nu \leq \nu$; es una prueba sencilla, que se deja al lector, demostrar estos hechos comprobando que, para cualesquiera x e y,$(x \wedge y) \wedge y = x \wedge y$. Por tanto, por (10.15), $f(\mu \wedge \nu) \leq f(\mu)$ y $f(\mu \wedge \nu) \leq f(\nu)$. Se deja al lector comprobar la ley general

$$x \leq y \text{ y } x \leq z \text{ implican } x \leq y \wedge z$$

Definiendo $x = f(\mu \wedge \nu)$, $y = f(\mu)$ y $z = f(\nu)$, se tiene (10.16).

A la inversa, se asume (10.16) y se demuestra (10.15). Se supone que $\mu \leq \nu$ y se utiliza (10.16) para concluir $f(\mu) \leq f(\nu)$, demostrando así (10.15). La ecuación (10.16) indica que $f(\mu \wedge \nu) \leq f(\mu) \wedge f(\nu)$. Pero ya que se asume que $\mu \leq \nu$, $\mu \wedge \nu = \mu$,

por definición. Así, (10.16) indica que $f(\mu) \leq f(\mu) \wedge f(\nu)$. Como regla general, el lector puede demostrar que

$$\text{si } x \leq y \wedge z \text{ entonces } x \leq z$$

Por tanto, (10.16) implica que $f(\mu) \leq f(\nu)$, y se ha demostrado (10.15).

A menudo un marco obedece una condición más fuerte que (10.16), que se denomina *condición de distributividad*:

$$f(\mu \wedge \nu) = f(\mu) \wedge f(\nu)$$

para todas μ y ν en V y f en F. Con certeza, si $x = y$, entonces $x \wedge y = x$ por idempotencia, de modo que $x \leq y$. Por tanto, la distributividad implica monotonicidad.

Ejemplo 10.46. Considérese el marco de las definiciones de alcance. Sean X e Y conjuntos de definiciones, y sea f una función definida por $f(Z) = G \cup (Z - K)$ para algunos conjuntos de definiciones G y K. Entonces se puede comprobar que el marco de las definiciones de alcance satisface la condición de distributividad, mediante la comprobación de que

$$G \cup ((X \cup Y) - K) = (G \cup (X - K)) \cup (G \cup (Y - K))$$

Dibujar un diagrama de Venn hace transparente la demostración de la relación anterior, aunque parezca complicado. □

Ejemplo 10.47. Ahora se demostrará que el marco para el cálculo de constantes es monótono pero no distributivo. Primero, sirve de ayuda aplicar la operación $\wedge$ y la relación $\leq$ a los elementos que aparecen en la tabla de la figura 10.58. Es decir, se define

$$\begin{aligned}
\mathit{noconst} \wedge c &= \mathit{noconst} && \text{para cualquier constante } c \\
c \wedge d &= \mathit{noconst} && \text{para constantes } c \neq d \\
c \wedge \mathit{indef} &= c && \text{para cualquier constante } c \\
\mathit{noconst} \wedge \mathit{indef} &= \mathit{noconst} && \\
x \wedge x &= x && \text{para cualquier valor } x
\end{aligned}$$

Entonces la figura 10.58 se puede interpretar como si se indicara que $\rho(\mathtt{a}) = \mu(\mathtt{a}) \wedge \wedge \nu(\mathtt{a})$.

Se puede determinar cuál es la relación $\leq$ sobre valores a partir de la operación $\wedge$. Se encuentra

$$\begin{aligned}
\mathit{noconst} &\leq c && \text{para cualquier constante } c \\
c &\leq \mathit{indef} && \text{para cualquier constante } c \\
\mathit{noconst} &\leq \mathit{indef} &&
\end{aligned}$$

Se muestra esta relación en el diagrama de malla de la figura 10.60, donde las c_i sirven para sugerir todas las constantes posibles. Obsérvese que esa figura no es $\leq$ sobre elementos de V; más bien es una relación sobre el conjunto de valores para $\mu(\mathtt{a})$ para variables individuales a. Se pueden considerar los elementos de V como vectores de dichos valores, un componente para cada variable, y el diagrama de malla para V se puede extrapolar de la figura 10.60 si se recuerda que $\mu \leq \nu$ se cumple si,

y sólo si, $\mu(a) \leq \nu(a)$ para toda a; es decir, si los vectores que representan μ y ν tienen a todos los componentes relacionados por $\leq$, y la relación está en la misma dirección en cada componente.

Por tanto, decir que $\mu \leq \nu$ es decir que, siempre que $\mu(a)$ sea una constante c, $\nu(a)$ es esa constante o es *indef*, y siempre que $\mu(a)$ sea *indef*, también lo será $\nu(a)$. Un examen detallado de las distintas funciones f asociadas con los distintos tipos de proposiciones de definición permite comprobar que si $\mu \leq \nu$, entonces $f(\mu) \leq f(\nu)$, demostrando así que cumple (10.15) y mostrando monotonicidad. Por ejemplo, si f está asociada con la asignación a:=b+c, sólo $\mu(a)$ y $\nu(a)$ cambian, de modo que hay que comprobar que si $\mu \leq \nu$ (es decir, $\mu(x) \leq \nu(x)$, para toda x), entonces $[f(\mu)](a) \leq [f(\nu)](a)$[14]. Se deben considerar todos los posibles valores de $\mu(b)$, $\mu(c)$, $\nu(b)$ y $\nu(c)$, sujetos a las limitaciones que $\mu(b) \leq \nu(b)$ y $\mu(c) \leq \nu(c)$. Por ejemplo, si

$$\begin{aligned} \mu(b) &= \textit{noconst} \\ \nu(b) &= 2 \\ \mu(c) &= 3 \\ \nu(c) &= \textit{indef} \end{aligned}$$

entonces $[f(\mu)](a) = \textit{noconst}$ y $[f(\nu)](a) = \textit{indef}$. Como $\textit{noconst} \leq \textit{indef}$, se ha hecho la comprobación en un caso. Los otros casos se dejan como ejercicio al lector.

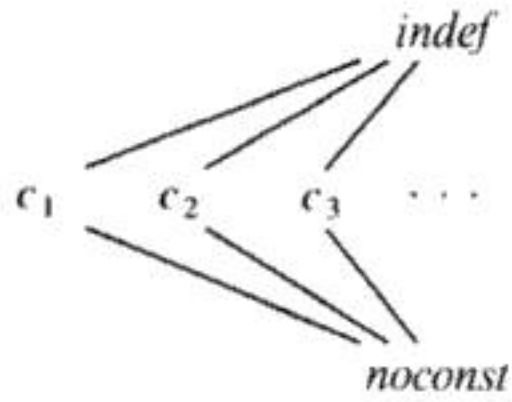


Fig. 10.60. Diagrama de retículo para valores de variables.

Ahora se debe comprobar que el marco para el cálculo de constantes no es distributivo. Para esta parte, sea f la función asociada con la asignación a:=b+c, y sea $\mu(b) = 2$, $\mu(c) = 3$, $\nu(b) = 3$ y $\nu(c) = 2$. Sea $\rho = \mu \wedge \nu$. Entonces $\mu(b) \wedge \nu(b) = 2 \wedge 3 = \textit{noconst}$. De manera similar, $\mu(c) \wedge \nu(c) = \textit{noconst}$. De forma equivalente, $\rho(b) = \rho(c) = \textit{noconst}$. Se deduce que $[f(\rho)](a) = \textit{noconst}$, puesto que se presume que la suma de dos valores no constantes es no constante.

Por otra pare, $[f(\mu)](a) = 5$, ya que dado que $b = 2$ y $c = 3$, la asignación a:=b+c iguala a a 5. De manera similar, $[f(\nu)](a) = 5$. Así, $[f(\mu) \wedge f(\nu)](a) = 5$. Se observa ahora que $\rho(a)\ [f(\mu \wedge \nu)](a) \neq [f(\mu) \wedge f(\nu)](a)$, de modo que se incumple la condición de distributividad.

Intuitivamente, la razón por la que se incumple la distributividad es que el marco para el cálculo de constantes no es lo bastante poderoso como para recordar todos los invariantes, en concreto, el hecho de que a lo largo de los caminos cuyos efectos

---

[14] Se debe tener cuidado al leer una expresión como $[f(\mu)](a)$. Dice, se aplica f a μ para obtener una correspondencia $f(\mu)$, que se llamará μ'. Después se aplica μ' a (a), y el resultado es uno de los valores en el diagrama de la figura 10.60.

sobre las variables son descritos por μ o ν, se cumple la ecuación b+c = 5, aunque ni b ni c sean por sí mismas constantes. Se podrían diseñar marcos más complicados para evitar este problema, aunque no está claro que suponga una ventaja. Por fortuna, la monotonicidad es adecuada para que "trabaje" el algoritmo de flujo de datos iterativo, como se verá a continuacion. □

Soluciones de reunión sobre caminos a problemas de flujo de datos

Imagínese que un grafo de flujo tiene asociado con cada uno de sus nodos una función de transferencia, una de las funciones del conjunto F. Para cada bloque B, sea f_B la función de transferencia para B.

Considérese cualquier camino $P = B_0 \rightarrow B_1, \ldots, \rightarrow B_k$ desde el nodo inicial B_0 a un bloque B_k. Se puede definir la *función de transferencia para P* como la composición de $f_{B_0}, f_{B_1}, \ldots, f_{B_k-1}$. Obsérvese que f_{B_k} no es parte de la composicion, reflejando el punto de vista de que este camino se toma para alcanzar el comienzo del bloque B_k, no su final.

Se ha supuesto que los valores dentro de V representan información sobre los datos utilizados por el programa, y que el operador de confluencia $\wedge$ indica cómo se combina esta información cuando convergen los caminos. También tiene sentido ver el elemento tope como que no representa "ninguna información", puesto que un camino que lleve el elemento tope conduce a cualquier otro camino, en lo que se refiere a la información que es llevada después de la confluencia de operaciones. Por tanto, si B es un bloque en el grafo de flujo, la información que entre a B debe ser calculable considerando todo camino posible desde el nodo inicial a B y viendo lo que sucede a lo largo de ese camino, comenzando sin información. Es decir, para cada camino P desde B_0 a B, se calcula $f_P(\mathsf{T})$ y se toma la reunión de todos los elementos resultantes.

En principio, esta reunión podría ser sobre un número infinito de valores distintos, puesto que hay un número infinito de caminos diferentes. En la práctica, a menudo resulta adecuado considerar sólo caminos acíclicos, y aunque no lo sea, como en el caso del marco para el cálculo de constantes anteriormente analizado, generalmente hay otras razones para hacer que esta reunión infinita sea finita para cualquier grafo de flujo determinado.

Formalmente, se define la *solución de reunión sobre caminos* para un grafo de flujo como

$$rsc(B) = \bigwedge_{\substack{\text{caminos } P \\ \text{entre } B_0 \text{ a } B}} f_P(\mathsf{T})$$

La solución de *rsc* a un grafo de flujo tiene sentido cuando se comprende que, en lo que se refiere a la información que alcanza el bloque B, el grafo de flujo también puede ser el propuesto en la figura 10.61, donde a la función de transferencia asociada con cada uno de (un número posiblemente infinito) los caminos $P_1, P_2, \ldots,$ en el grafo de flujo original se le ha dado un camino completamente independiente hacia B. En la figura 10.61, la información que alcanza B viene dada por la reunión sobre todos los caminos.

Soluciones conservadoras a los problemas de flujo

Cuando se intentan resolver las ecuaciones de flujos de datos que se obtienen de un marco arbitrario, se puede o no ser capaz de conseguir fácilmente la solución de *rsc*. Por fortuna, como en los ejemplos concretos de marcos de flujo de datos de las secciones 10.5 y 10.6, hay una dirección segura en la cual se puede fallar y el algoritmo de flujo de datos iterativo estudiado en esas secciones siempre proporciona una solución segura. Se dice que una solución $ent[B]$ es una *solución segura* si $ent[B] \leq \leq rsc(B)$ para todos los bloques B.

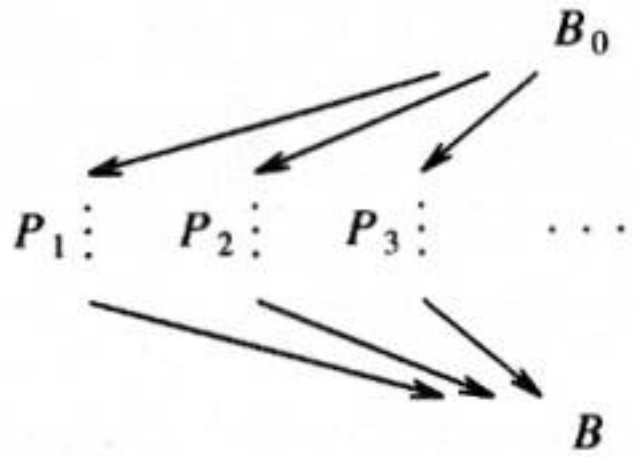


Fig. 10.61. Grafo que muestra el conjunto de todos los posibles caminos a B.

A pesar de lo que el lector pueda imaginar, esta definición no ha sido "sacada de la manga". Recuérdese que en cualquier grafo de flujo, el conjunto de caminos *aparentes* a un nodo (aquellos que son caminos en el grafo de flujo) pueden ser un subconjunto propio de los caminos *reales*, aquellos que se toman en alguna ejecución del programa correspondiente a dicho grafo de flujo. Para que el resultado del análisis de flujo de datos sea utilizable cualquier propósito, los datos deben seguir siendo fiables si se modifica el grafo de flujo eliminando algunos caminos, puesto que en general no se pueden distinguir los caminos reales de los aparentes que no son reales.

Supóngase que entre el conjunto infinito de caminos propuesto en la figura 10.61, x es la reunión de $f_P(\mathrm{T})$ tomada de entre todos los caminos reales P que son seguidos en alguna ejecución. Asimismo, sea y y la reunión de $f_P(\mathrm{T})$ sobre todos los otros caminos P. Por tanto, $rsc(B)$ es $x \wedge y$. Entonces, la respuesta cierta al problema de flujo de datos en el nodo B es x, pero la solución de *rsc* es $x \wedge y$. Recuérdese que $x \wedge y \leq y$, puesto que $(x \wedge y) \wedge y = x \wedge y$. Así, la solución de *rsc* es $\leq$ la solución verdadera.

Aunque se prefiere la solución "verdadera" al problema de flujo de datos, casi con seguridad no habrá una forma eficiente de saber exactamente las partes que son reales y las que no, así que hay que aceptar la solución de *rsc* como la mejor solución factible. Por tanto, cualquier uso que se haga de la información sobre el flujo de datos deberá ser consistente con la posibilidad de que la solución que se obtenga sea $\leq$ la solución verdadera.

Una vez aceptado esto, también se debería poder aceptar una solución que sea $\leq$ la solución de *rsc* (y por tanto $\leq$ la solución verdadera). Dichas soluciones son más fáciles de obtener que la solución de *rsc* para aquellos marcos que sean monótonos pero no distributivos. Para marcos distributivos, como los de la sección 10.6, el algoritmo iterativo simple calcula la solución de *rsc*.

El algoritmo iterativo para marcos generales

Hay una generalización obvia al algoritmo 10.2 que sirve para una gran variedad de marcos. El algoritmo iterativo exige que el marco sea monótono, y tiene que ser finito, en el sentido que la reunión sobre el conjunto infinito de caminos sugerido en la figura 10.61 es equivalente a una reunión sobre un subconjunto finito. Se dará el algoritmo y después se analizarán las formas de garantizar que fuera finito. Sin embargo, una garantía habitual para que sea finito es la que se ha tenido todo el tiempo: basta con la propagación a lo largo de caminos acíclicos.

Algoritmo 10.18. Solución iterativa a marcos generales de flujo de datos.

Entrada. Un grafo de flujo de datos, un conjunto de "valores" V, un conjunto de funciones F, una operación de reunión $\wedge$, y una asignación de un miembro de F a cada nodo del grafo de flujo.

Salida. Un valor *ent* [B] en V para cada nodo del grafo de flujo.

Método. Se da el algoritmo en la figura 10.62. Al igual que los algoritmos de flujo de datos iterativos, se calcula *ent* y *sal* para cada nodo mediante aproximaciones sucesivas. Se supone que f_B es la función en F asociada con el bloque B; esa función desempeña el papel de *gen* y *desact* de la sección 10.6. □

```
(1)  for cada nodo B do /* inicialización, suponiendo que ent [B] = ⊤ */
(2)      sal [B] := f_B [⊤];
(3)  while ocurran cambios a cualquier sal do
(4)      for cada bloque B, en orden en profundidad do begin
(5)          sal [B] := ⋀ (predecesores P de B) sal [P];
             /* antes, la reunión de un conjunto vacío es ⊤ */
(6)          sal [B] := f_B (ent [B])
         end
```

Fig. 10.62. Algoritmo iterativo para marcos generales.

Una herramienta para el análisis de flujo de datos

Ahora se puede ver cómo se pueden aplicar las ideas de esta sección a una herramienta para el análisis de flujo de datos. El algoritmo 10.18 depende para su funcionamiento de las siguientes subrutinas:

1. Una rutina para aplicar una determinada f_B en F a un determinado valor en V. Esta rutina se utiliza en las líneas (2) y (6) de la figura 10.62.
2. Una rutina para aplicar el operador de reunión a dos valores en V; esta rutina se necesita cero o más veces en la línea (5).
3. Una rutina para saber si dos valores son iguales o no. Esta prueba no se realiza de manera explícita en la figura 10.62, pero está implícita en la comprobación de si ha habido un cambio en cualquiera de los valorse de *sal*.

También hay que especificar declaraciones de tipos de datos para F y V para poder pasar argumentos a las rutinas antes mencionadas. Los valores de *ent* y *sal* de la figura 10.62 también son del tipo declarado para V. Por último, se necesita una rutina que tome la representación ordinaria del contenido de un bloque básico, es decir, una lista de proposiciones, y produzca un elemento de F, la función de transferencia para ese bloque.

Ejemplo 10.48. Para el marco para las definiciones de alcance, primero se puede construir una tabla que identificara cada proposición del grafo de flujo dado con un entero único desde 1 hasta un máximo m. Después, el tipo de V podría ser vectores de bits de longitud m. F podría representarse por medio de pares de vectores de bits de ese tamaño, es decir, por medio de los conjuntos *gen* y *desact*. La rutina para construir los vectores de bits *gen* y *desact*, dadas las proposiciones de un bloque y la tabla que asocia las proposiciones de definición con posiciones en los vectores de bits, es directa, como lo son las rutinas para calcular las reuniones (operación o lógica de vectores de bits), comparar vectores de bits en términos de igualdad y aplicar funciones definidas por un par *gen-desact* a vectores de bits.

La herramienta para el análisis de flujo de datos es por tanto poco más que una implantación de la figura 10.62 con llamadas a las subrutinas dadas siempre que se necesite una reunión, la aplicación de una función o una comparación. La herramienta apoyará una representación fija de grafos de flujo, y por tanto tendrá que realizar funciones como encontrar todos los predecesores de un nodo o el ordenamiento en profundidad del grafo de flujo, y aplicar a cada bloque la rutina que calcula la función en F asociada con ese bloque. La ventaja de utilizar dicha herramienta es que los aspectos de manipulación de grafos y comprobaciones de convergencia del algoritmo 10.18 no tienen que reescribirse para cada análisis de flujo de datos que se realice.

Propiedades del algoritmo 10.18

Se deben aclarar los supuestos bajo los que funciona el algoritmo 10.18 y exactamente a qué converge el algoritmo cuando converge. Primero, si el marco es monótono y converge, entonces se afirma que el resultado del algoritmo es que $ent\,[B] \leq rsc(B)$ para todos los bloques B. La razón intuitiva es que a lo largo de cualquier camino $P = B_0, B_1, \ldots, B_k$ desde el nodo inicial a $B = B_k$, se puede demostrar por inducción sobre i que el efecto del camino desde B_0 hasta B_i se percibe después de a lo sumo i iteraciones del lazo **while** de la figura 10.62. Es decir, si P_i es el camino $B_0, \ldots, B_i$, entonces después de i iteraciones, $ent\,[B_i] \leq f_{P_i}(\mathsf{T})$. Por tanto, cuando, y si, el algoritmo converge, $ent\,[B]$ será $\leq f_P(\mathsf{T})$ para todo camino P desde B_0 a B. Utilizando la regla que si $x \leq y$ y $x \leq z$, entonces $x \leq y \wedge z$[15], se puede demostrar que $ent\,[B] \leq rsc(B)$.

---

[15] Existe la exigencia técnica de que se debe mostrar en principio esta regla no sólo para los valores, y y z (de donde se deriva la regla de que si $x \leq y_i$ para cualquier conjunto finito de y_i entonces $x \leq \wedge_i y_i$) sino que la misma regla se cumple para un número infinito de y_i. Sin embargo, en la práctica, siempre que se obtenga convergencia del algoritmo 10.18, se encontrará un número finito de caminos tal que la reunión sobre todos los caminos sea igual a la reunión sobre este conjunto finito.

Cuando el marco es distributivo, se puede demostrar que el algoritmo 10.18 de hecho converge a la solución *rsc*. La idea fundamental es demostrar que en todo momento durante la ejecución del algoritmo, *ent* [*B*] y *sal* [*B*] son iguales a la reunión de $f_P(\mathsf{T})$ para un conjunto de caminos P al comienzo y final de B, respectivamente. Sin embargo, en el siguiente ejemplo se demuestra que éste no tiene por qué ser el caso cuando el marco sea monótono, pero no distributivo.

Ejemplo 10.49. Se explotará el ejemplo de la no distributividad del marco para el cálculo de constantes que se estudió en el ejemplo 10.47; el grafo de flujo relevante se muestra en la figura 10.63. Las correspondencias μ y ν que salen de B_2 y B_4 son las del ejemplo 10.47. La correspondencia ρ, que entra a B_5, es $\mu \wedge \nu$, y δ es la correspondencia que sale de B_5, asignándole *noconst* a a, aunque cada camino real (y cada camino aparente), calcule a = 5 después de B_5.

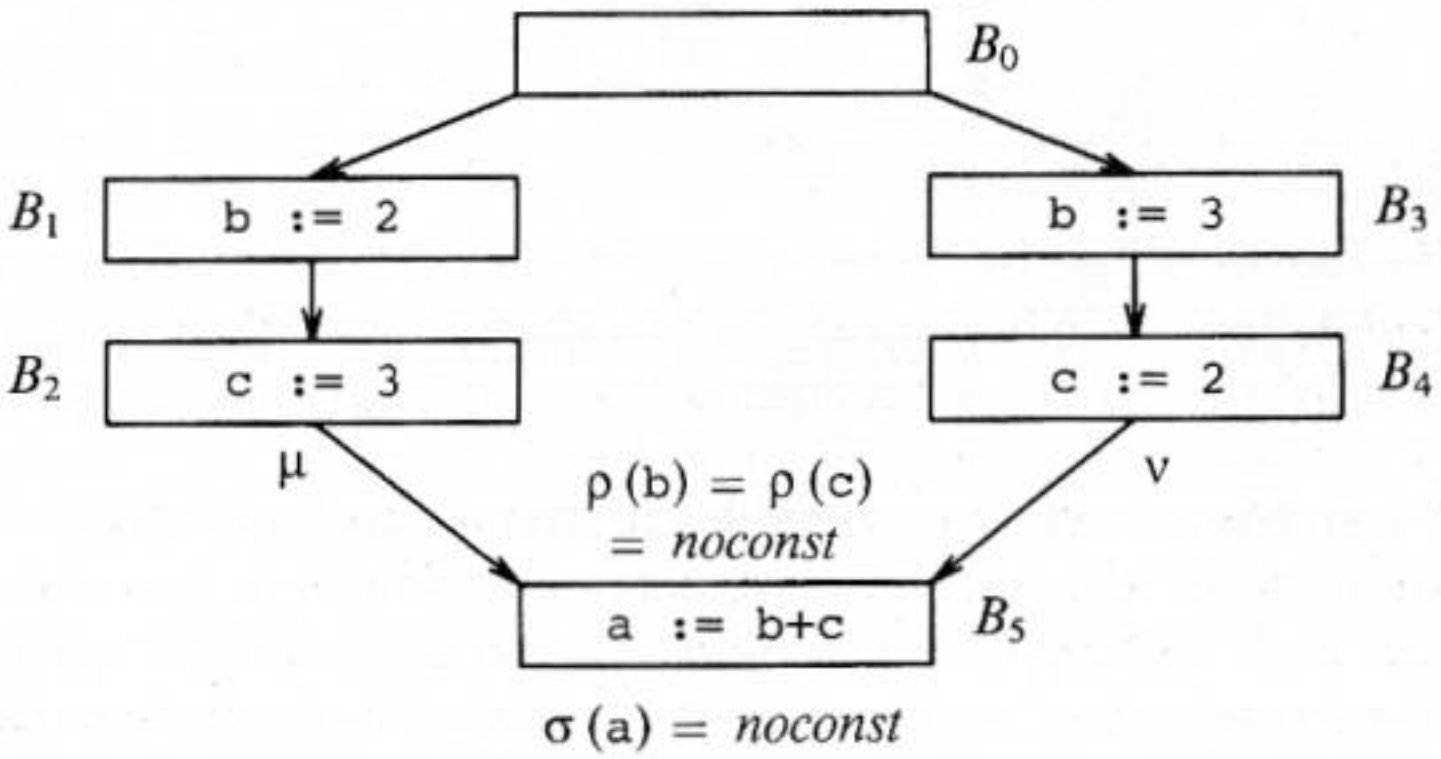


Fig. 10.63. Ejemplo de una solución menor que la solución de *rdc*.

El problema, intuitivamente, es que el algoritmo 10.18, trabajando con un marco no distributivo, se comporta como si algunas secuencias de nodos que ni siquiera son caminos aparentes (caminos en grafo de flujo), fueran caminos reales. Por tanto, en la figura 10.63, el algoritmo se comporta como si caminos como $B_0 \rightarrow B_1 \rightarrow$ $\rightarrow B_4 \rightarrow B_5$ o $B_0 \rightarrow B_3 \rightarrow B_2 \rightarrow B_5$ fueran caminos reales, colocando a b y c una combinación de valores que no suman 5. □

Convergencia del algoritmo 10.18

Hay varias formas de demostrar que el algoritmo 10.18 converge para un marco particular. Probablemente el caso más común es cuando sólo se necesitan caminos acíclicos, es decir, se puede demostrar que la reunión sobre caminos acíclicos es la misma que la solución de *rsc*, sobre todos los caminos. Si es ése el caso, no sólo converge el algoritmo, y normalmente lo hará muy rápidamente, en dos pasadas más que la profundidad del grafo de flujo, como se analizó en la sección 10.10.

Por otra parte, los marcos como el del ejemplo de cálculo de constantes exigen que se consideren más elementos además de los caminos acíclicos. Por ejemplo, en

la figura 10.64 se muestra un sencillo grafo de flujo donde hay que considerar el camino $B_1 \rightarrow B_2 \rightarrow B_2 \rightarrow B_3$ para darse cuenta de que x no tiene un valor constante al entrar a B_3.

Sin embargo, para cálculos constantes se puede razonar que el algoritmo 10.18 converge de la siguiente forma. Primero, es fácil demostrar para un marco monótono arbitrario que *ent* [*B*] y *sal* [*B*], para cualquier bloque *B*, forman una secuencia no creciente, en el sentido de que el nuevo valor para una de esas variables es siempre $\leq$ el valor anterior. Si se recuerda la figura 10.60, el diagrama de retículo para los valores de las correspondencias aplicadas a variables, se ve que para cualquier variable, el valor de *ent* [*B*] o *sal* [*B*] sólo puede disminuir dos veces, una de *indef* a una constante y una de esa constante a *noconst*.

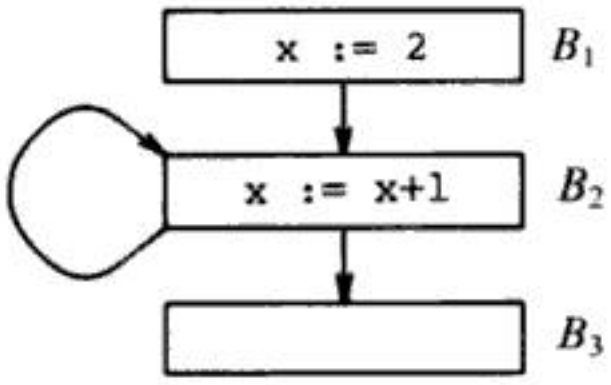


Fig. 10.64. Grafo de flujo que exige que un camino cíclico se incluya en *rsc*.

Supóngase que hay *n* nodos y *v* variables. Entonces en cada iteración del lazo **while** de la figura 10.62, al menos una variable debe disminuir su valor en algún *sal* [*B*] o el algoritmo converge, y ni siquiera una iteración infinita del lazo **while** modificará los valores en los *ent* y los *sal*. Por tanto, el número de iteraciones se limita a $2nv$; si ese número de modificaciones ocurre, entonces cada variable debe haber alcanzado *noconst* en cada bloque del grafo de flujo.

Ajuste de la inicialización

En algunos problemas de flujo de datos, hay una discrepancia entre lo que el algoritmo 10.18 da como solución y lo que intuitivamente se desea. Recuérdese que para las expresiones disponibles, $\wedge$ es la intersección, así que T debe ser el conjunto de todas las expresiones. Ya que el algoritmo 10.18 inicialmente asume que *ent* [*B*] es T para cada bloque *B*, incluido el nodo inicial, la solución de *rsc* producida por el algoritmo 10.18 es realmente el conjunto de expresiones que, suponiendo que estén disponibles en el nodo inicial (que no lo están), estarían disponibles a la entrada al bloque *B*.

La diferencia, por supuesto, es que podría haber caminos desde el nodo inicial a *B* a lo largo del cual ni se genera ni se desactiva una expresión x+y. El algoritmo 10.18 indicaría que x+y está disponible, cuando de hecho no lo está, porque no se puede encontrar ninguna variable a lo largo de ese camino que contenga su valor. El ajuste es sencillo. Se puede modificar el algoritmo 10.18 de modo que para el marco de expresiones disponibles, *ent* [B_0] se asigne y se iguale al conjunto vacío, o se puede modificar el grafo de flujo introduciendo un nodo inicial ficticio, un predecesor del nodo inicial real, que desactive todas las expresiones.

10.12 ESTIMACION DE TIPOS

Ahora se aborda un problema de flujo de datos más complejo que los marcos de la sección anterior. Algunos lenguajes, desde APL a SETL hasta los diversos dialectos de LISP, no exigen que se declaren los tipos de las variables, y hasta permiten que la misma variable contenga valores de distintos tipos en diferentes momentos. Intentos serios de compilar dichos lenguajes para producir código eficiente han utilizado análisis de flujo de datos para inferir los tipos de las variables, ya que el código para, por ejemplo, sumar dos enteros, es mucho más eficiente que una llamada a una rutina general que pueda sumar dos objetos de una variedad de tipos (por ejemplo, enteros, reales, vectores).

La primera idea es que calcular tipos de variables es como calcular las definiciones de alcance. Se puede asociar un conjunto de tipos posibles con cada variable en cada punto. El operador de confluencia es la unión de conjuntos de tipos, ya que si la variable x tiene el conjunto S_1 de posibles tipos en un camino y el conjunto S_2 en otro, entonces x tiene cualquiera de los tipos en $S_1 \cup S_2$ después de la confluencia de caminos. A medida que el control pasa a través de una proposición, se pueden hacer algunas inferencias relativas a los tipos de las variables basándose en los operadores que aparezcan en la proposición, los tipos posibles de sus operandos y los tipos que produzcan como resultados. El ejemplo 6.6, relativo a un operador que podía multiplicar enteros y números complejos, era un ejemplo de este tipo de inferencia.

Desgraciadamente, hay al menos dos problemas con este enfoque.

1. El conjunto de tipos posibles para una variable puede ser infinito.
2. La determinación de tipos exige generalmente propagación de la información hacia adelante y hacia atrás para obtener estimaciones precisas de los tipos posibles. Por tanto, ni siquiera el marco de la sección 10.11 es lo bastante general como para juzgar el problema.

Antes de considerar el punto 1, se examinarán algunos de los tipos de inferencias sobre tipos que se pueden hacer en lenguajes familiares.

Ejemplo 10.50. Considérense las proposiciones

```
i := a[j]
k := a[i]
```

Supóngase que al principio no se sabe nada sobre los tipos de las variables `a`, `i`, `j` y `k`. Sin embargo, supóngase que el operador de acceso a matrices [ ] exige un argumento entero. Si se examina la primera proposición se puede inferir que `j` es un entero en ese punto, y que `a` es una matriz de elementos de algún tipo. Entonces, la segunda proposición indica que `i` es un entero.

Ahora se pueden propagar las inferencias hacia atrás. Si se calculó `i` como un entero en la primera proposición, entonces el tipo de la expresión `a[i]` debe ser entero, lo cual significa que `a` debe ser una matriz de enteros. Después se puede seguir razonando hacia adelante para descubrir que el valor asignado a `k` por la segunda

proposición también debe ser un entero. Obsérvese que es imposible descubrir que los elementos de a son enteros razonando sólo hacia adelante o sólo hacia atrás. □

Conjuntos de tipos infinitos

Hay numerosos ejemplos de casos patológicos donde el conjunto de tipos posibles para una variable es en realidad infinito. Por ejemplo, SETL permite que una proposición como

```
x := { x }
```

se ejecute dentro de un lazo. Si se comienza sabiendo sólo que x podría ser un entero, entonces después de considerar una iteración del lazo se descubre que x podría ser un entero o un conjunto de enteros. Después de considerar una segunda iteración, se ve que x podría ser también un conjunto de conjuntos de enteros, y así sucesivamente.

Un problema similar podría ocurrir en una versión sin tipos de un lenguaje convencional cómo C, donde la proposición

```
x = &x
```

con la posibilidad inicial de que x sea un entero conduce al descubrimiento de que x puede tener cualquier tipo de la forma

apuntador a apuntador a . . . apuntador a entero

La forma tradicional de solucionar dichos problemas es reducir el conjunto de tipos posibles a un número finito. La idea general consiste en agrupar el número infinito de tipos posibles en un número finito de clases, dejando generalmente solos a los tipos más simples, y agrupando en clases más grandes los más complicados, que se espera sean los más escasos. Los siguientes ejemplos sugieren lo que se puede hacer.

Ejemplo 10.51. Se sigue con el ejemplo del capítulo 6, donde se utilizó el operador $\rightarrow$ como un constructor de tipos para funciones. Aquí, el conjunto de tipos incluirá el tipo básico *ent*, y todos los tipos de la forma $\tau \rightarrow \sigma$, que representan el tipo de una función con dominio tipo τ y rango tipo σ, donde τ y σ son tipos del conjunto. Por tanto, el conjunto de tipos es infinito, incluidos tipos como

$$(ent \rightarrow ent) \rightarrow ((ent \rightarrow ent) \rightarrow ent)$$

Para reducir este conjunto a un número finito de clases, una expresión de tipos se limitará a tener un constructor de tipos de función $\rightarrow$, sustituyendo las subexpresiones en una expresión de tipos que contengan al menos una ocurrencia de $\rightarrow$ por el nombre *func*. Por tanto, hay cinco tipos distintos:

$$\begin{aligned} &ent \\ &ent \rightarrow ent \\ &ent \rightarrow func \\ &func \rightarrow ent \\ &func \rightarrow func \end{aligned}$$

Los conjuntos de tipos se representarán como vectores de bits de longitud cinco, con las posiciones correspondientes a los cinco tipos en el orden anteriormente listado. Así, 01111 representa el tipo para cualquier aplicación de función, es decir, para cualquiera excepto *ent*. Obsérvese que éste es en un sentido el tipo de *func*, puesto que *func* no puede ser un entero.

La proposición de asignación básica para este modelo es

```
x := f(y)
```

Conociendo los tipos posibles de `f` e `y`, se pueden determinar los tipos posibles de `x` buscando el tipo en la tabla de la figura 10.65. Si `f` puede ser de cualquier tipo en el conjunto S_1 e y de cualquier tipo en el conjunto S_2, se toma cada par τ en S_1 y σ en S_2 y se busca la entrada en la fila correspondiente a τ y la columna de σ, que se llamará $\tau(\sigma)$. Después se toma la unión de los resultados de todas esas búsquedas para obtener el conjunto de tipos posibles para `x`.

τ	σ				
	ent	*ent* → *ent*	*ent* → *func*	*func* → *ent*	*func* → *func*
ent	00000	00000	00000	00000	00000
ent → *ent*	10000	00000	00000	00000	00000
ent → *func*	01111	00000	00000	00000	00000
func → *ent*	00000	10000	10000	10000	10000
func → *func*	00000	01111	01111	01111	01111

Fig. 10.65. El valor de $\tau(\sigma)$.

Por ejemplo, si $\tau = ent \rightarrow func$ y $\sigma = ent$, entonces $\tau(\sigma) = 01111$. Es decir, el resultado de aplicar una correspondencia de tipo *ent* → *func* a un *ent* es una *func*, lo cual significa una transformación de cualquiera de los cuatro tipos además de *ent*. No se puede saber cuál porque repartir un número infinito de tipos en cinco clases lo impide.

Como segundo ejemplo, sea τ como antes y $\sigma = ent \rightarrow ent$. Entonces $\tau(\sigma) = 0000$ porque el tipo del dominio de τ es definitivamente distinto al tipo σ, y por tanto no se puede aplicar la correspondencia. □

Un sistema de tipos sencillo

Para ilustrar las ideas en que se basan los algoritmos para inferencia de tipos, se introduce un sistema de tipos sencillo y un lenguaje basado en el ejemplo 10.51. Los tipos son los cinco ilustrados en este ejemplo. Las proposiciones del lenguaje son de tres clases.

1. `read x`. Un valor de `x` se lee desde la entrada y presumiblemente, no se sabe nada de su tipo.

2. `x := f(y)`. El valor de `x` se iguala al obtenido mediante la aplicación de la función `f` al valor `y`. En la figura 10.65 está resumido lo que se conoce del tipo de `x` después de la asignación.

3. `usar x como` τ. Cuando se pasa a través de dicha proposición, se puede suponer que el problema es correcto y por tanto, el tipo de `x` sólo puede ser τ antes y después de la proposición. El valor y el tipo de `x` no se ven afectados por la proposición.

Los tipos se infieren realizando un análisis de flujo de datos sobre un grafo de flujo de un programa que consta de proposiciones de estos tres tipos. Para simplificar, se supone que todos los bloques constan de una sola asignación. Los valores de *ent* y *sal* para bloques son correspondencias de variables a conjuntos de los cinco tipos del ejemplo 10.51.

Al inicio, cada *ent* y *sal* hacen una correspondencia de cada variable al conjunto de todos los cinco tipos. Cuando se propaga la información, se reduce el conjunto de tipos asociados con algunas variables en algunos puntos, hasta que en algún momento ya no se puedan reducir más ninguno de esos conjuntos. Se supondrá que los conjuntos resultantes indican los tipos posibles de cada variable en cada punto. Esta suposición es conservadora, puesto que un tipo se elimina sólo si se puede demostrar (dado que el programa sea correcto) que el tipo es imposible. Normalmente, se espera sacar ventaja de que algunos tipos son imposibles, no de que son posibles, así que "demasiado grande" es la dirección segura para los errores.

Se utilizan dos esquemas para modificar los conjuntos *ent* y *sal*: un esquema "hacia adelante" y un esquema "hacia atrás". El esquema hacia adelante utiliza la proposición que está en el bloque B y el valor de *ent* [B] para limitar *sal* [B][16], y el esquema hacia atrás hace lo contrario. En cada esquema, el operador de confluencia es la "unión orientada a variables", en el sentido de que la confluencia de dos correspondencias α y β es aquella correspondencia γ tal que para todas las variables x,

$$\gamma[x] = \alpha[x] \cup \beta[x]$$

El esquema de avance

Supóngase que se tiene un bloque B siendo *ent* [B] la correspondencia μ y *sal* [B] la correspondencia ν. El esquema de avance permite limitar ν. Las reglas para limitar ν dependen de la instrucción que se encuentre en el bloque B, naturalmente.

1. Si la proposición es `read x`, entonces se podría leer cualquier tipo. Si ya se sabe algo acerca del tipo de `x` después de la lectura, no se debe olvidar durante esta

---

[16] Vale la pena observar que en los esquemas tradicionales de flujo de datos hacia adelante, no se limitó *sal*, sino que más bien se volvió a calcular a partir de *ent* cada vez. Esto se pudo hacer porque los conjuntos *ent* y *sal* siempre cambiaron en una dirección, ya fuera siempre creciendo o siempre disminuyendo. Sin embargo, en un problema como la inferencia de tipos, donde se realizan pasadas alternativamente hacia adelante y hacia atrás, se puede dar el caso de que la pasada hacia atrás haya dejado *sal* mucho más pequeño que lo que se puede justificar aplicando las reglas hacia adelante a *ent*. Por tanto, no se debe aumentar accidentalmente *sal* en la pasada hacia adelante, sólo para volver a reducirlo (pero quizá no tanto) en la pasada hacia atrás. Un comentario similar sirve para la pasada hacia atrás; se debe limitar *ent*, no recalcularlo.

pasada hacia adelante, de modo que simplemente no se cambia $\nu(x)$ en la pasada hacia adelante. Para todas las otras variables y, se hace

$$\nu(y) := \nu(y) \cap \mu(y)$$

2. Supóngase ahora que la proposición es `usar x como` τ. Después de esta proposición, τ es el único tipo posible para x. Si ya se sabe que el tipo τ es imposible para x, entonces no hay ningún tipo posible para x después de la proposición. Estas observaciones se pueden resumir mediante:

$$\nu(x) := \nu(x) \cap \{\tau\}$$
$$\nu(y) := \nu(y) \cap \mu(y) \text{ para } y \neq x$$

3. Ahora considérese el caso en que la proposición es `x := f(y)`. Los únicos tipos posibles para x después de la proposición son aquellos que

   i) son posibles según el valor presente de ν, y

   ii) son el resultado de aplicar una correspondencia de un tipo τ al tipo σ, y τ y σ son tipos que f e y, respectivamente, podrían tener antes de que se ejecute la proposición.

Formalmente,

$$\nu(x) := \nu(x) \cap \{\rho \mid \rho = \tau(\sigma), \tau \text{ está en } \mu(f), \text{ y } \sigma \text{ está en } \mu(y)\}$$

También se pueden hacer algunas inferencias sobre los tipos de f e y, puesto que dando por supuesta la corrección del programa, f no puede tener un tipo que no se aplique a algún tipo de y e y no puede tener un tipo que no pueda servir como el tipo del argumento para algunos tipos posibles de f. Es decir, si $f \neq x$, entonces

$$\nu(f) := \nu(f) \cap \{\tau \text{ en } \mu(f) \mid \text{para alguna } \sigma \text{ en } \mu(y), \tau(\sigma) \neq \varnothing\}$$

si $y \neq x$, entonces

$$\nu(y) := \nu(y) \cap \{\sigma \text{ en } \mu(y) \mid \text{para alguna } \tau \text{ en } \mu(f), \tau(\sigma) \neq \varnothing\}$$

para todas las otras z,

$$\nu(z) := \nu(z) \cap \mu(z)$$

El esquema de retroceso

Ahora se considerará cómo, en una pasada hacia atrás, se puede limitar μ basándose en lo que indique ν y las proposiciones.

1. Si la proposición es `read x`, es fácil ver que no se pueden hacer nuevas inferencias sobre tipos imposibles antes de la proposición, así que $\mu(x)$ no cambia. Sin embargo, para toda $y \neq x$, se puede propagar información hacia atrás haciendo $\mu(y) := \mu(y) \cap \nu(y)$.

2. Si se tiene la proposición `usar x como` τ, entonces se puede hacer la misma clase de inferencia que en la dirección hacia adelante; x sólo puede tener tipo τ

antes de la proposición, y los tipos de las otras variables son los que se consideran posibles antes y después de la proposición. Es decir:

$$\mu(x) := \mu(x) \cap \{\tau\}$$
$$\mu(y) := \mu(y) \cap \nu(y) \text{ para } y \neq x$$

3. Como antes, el caso más complejo es una proposición de la forma `x := f(y)`. Para empezar, no se puede inferir nada nuevo sobre `x` antes de la proposición, a menos que `x` sea una de `f` o `y`. A continuación, obsérvese que, como con las reglas hacia adelante, se pueden hacer inferencias a partir del hecho de que los tipos de `f` e `y` deben ser compatibles antes de la proposición. Sin embargo, si $f \neq x$, también se puede limitar $\mu(f)$ a los tipos en $\nu(f)$, y se cumple una afirmación análoga acerca de `y`. Por otra parte, si $f = x$, entonces los tipos de `f` después de la proposición no están relacionados con los tipos de `f` antes de la proposición, de modo que no se permite dicha limitación. De nuevo se cumple una afirmación análoga si $y = x$. Es útil definir una correspondencia especial, sólo para `f` e `y`, para reflejar esta decisión. Por tanto, se define:

$$\text{si } f = x \text{ entonces } \mu_1(f) := \mu(f), \text{ de lo contrario } \mu_1(f) := \mu(f) \cap \nu(f)$$
$$\text{si } y = x \text{ entonces } \mu_1(y) := \mu(y), \text{ de lo contrario } \mu_1(y) := \mu(y) \cap \nu(y)$$

Ahora, se pueden limitar `f` e `y` a aquellos tipos que sean compatibles con los conjuntos de tipos del otro. Al mismo tiempo, se pueden limitar los tipos de `f` e `y` basándose en el hecho de que no sólo deben ser compatibles, sino que deben producir un tipo que indique que puede tener `x`. Por tanto, se define:

$$\mu(f) := \{\tau \text{ en } \mu_1(f) \mid \text{para alguna } \sigma \text{ en } \mu_1(y),\ \tau(\sigma) \cap \nu(x) \neq \varnothing\}$$
$$\mu(y) := \{\sigma \text{ en } \mu_1(y) \mid \text{para alguna } \tau \text{ en } \mu_1(f),\ \tau(\sigma) \cap \nu(x) \neq \varnothing\}$$
$$\mu(z) := \mu(z) \cap \nu(z) \text{ para } z \text{ no igual a } x, y \text{ o } f$$

Antes de pasar al algoritmo para determinación de tipos, recuérdese el estudio de las definiciones de alcance en la sección 10.5 que indicaba que si se comienza con la falsa suposición de que una definición *d* está disponible en algún punto en un lazo, se puede propagar erróneamente este hecho a lo largo del lazo, dejando un conjunto de definiciones de alcance mayor de lo necesario. Puede ocurrir un problema similar con la determinación de tipos, donde la suposición de que una variable puede tener un tipo determinado "se demuestra" a sí misma cuando se itera el lazo. Por tanto, se introducirá un 33° valor, además de los 32 conjuntos de tipos del ejemplo 10.51 que una correspondencia μ pueda asignar a una variable, el valor *indef*. Este uso de *indef* es similar a su uso en el marco de propagación de constantes de la sección anterior.

Durante la confluencia, el valor *indef* admite cualquier otro valor, es decir, actúa como el tipo 00000. Por otro lado, cuando se intersectan conjuntos de tipos, por ejemplo, calcular $\mu(y) \cap \nu(x)$, el valor *indef* también admite cualquier otro conjunto de tipos; es decir, funciona como el tipo 11111. Así, cuando se lee por ejemplo un valor de una variable `x`, se niega que se haya pensado que el "tipo" de `x` era *indef* después de la lectura, y el tipo de `x` se vuelve 11111.

Algoritmo 10.19.

Entrada. Un grafo de flujo cuyos bloques son proposiciones simples de los tres tipos (lectura, asignación y utilizar como) mencionados anteriormente.

Salida. Un conjunto de tipos para cada variable en cada punto. El conjunto es conservador, en el sentido de que cualquier cálculo real debe conducir a un tipo dentro del conjunto.

Método. Se calcula una correspondencia *ent* [*B*] y una correspondencia *sal* [*B*] para cada bloque *B*. Cada correspondencia envía las variables del programa a conjuntos de tipos en el sistema de tipos presentado en el ejemplo 10.51. Al inicio, todas las correspondencias envía todas las variables a *indef*.

Después se hacen alternativamente pasadas hacia atrás y adelante a través del grafo de flujo, hasta que tanto las pasadas consecutivas hacia adelante como hacia atrás no puedan producir más cambios. La pasada hacia adelante se realiza mediante:

for cada bloque *B* en orden en profundidad **do begin**
    $ent\,[B] := \bigcup_{pred.\ P\ de\ B} sal\,[P]$;
    sal [*B*] := función de *ent* [*B*] y *sal* [*B*] como se definió anteriormente
end

La pasada hacia atrás es:

for cada bloque *B* en orden en profundidad inverso **do begin**
    $sal\,[B] := \bigcup_{sucesor\ S\ de\ B} ent\,[S]$;
    ent [*B*] := función de *ent* [*B*] y *sal* [*B*] como se definió anteriormente
end □

Ejemplo 10.52. Considérese el sencillo programa que se muestra en la figura 10.66. Se consideran cuatro correspondencias que se designarán μ_1 hasta μ_4. Cada μ_i es *sal* [B_i] y *ent* [B_{i+1}]. Técnicamente, B_1 no debería tener dos proposiciones, porque se

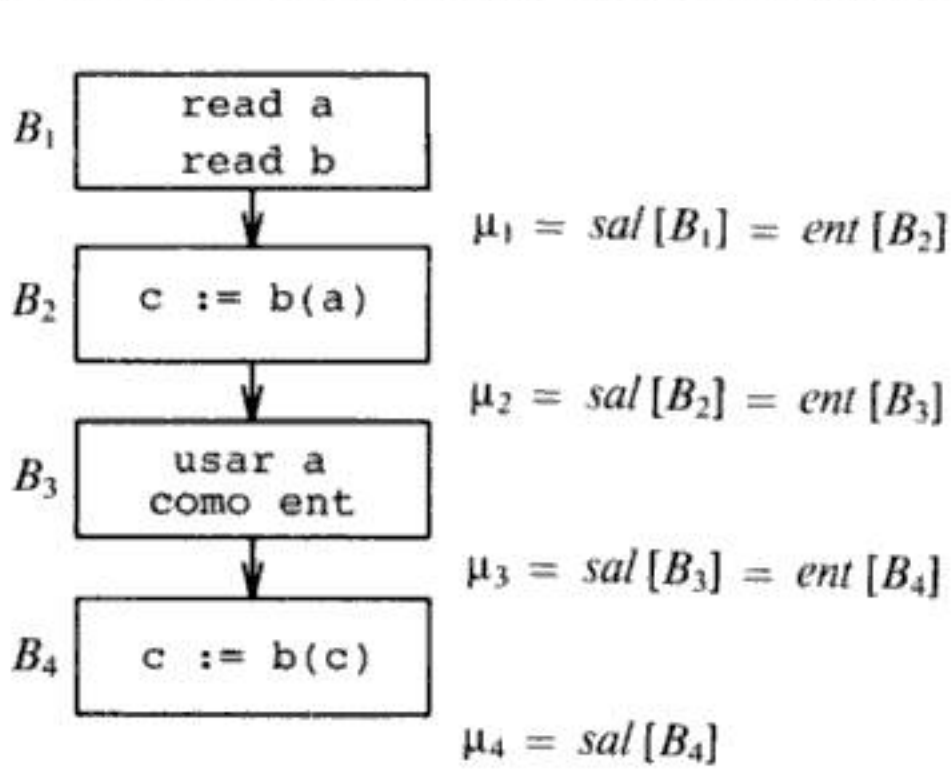


Fig. 10.66. Programa de ejemplo.

ha supuesto que los bloques constan de una sola proposición en esta sección. Sin embargo, no se considera lo que ocurre antes del final de B_1 porque todas las variables pueden tener allí cualquier tipo.

Resulta que se necesitan cinco pasadas antes de que ocurra la convergencia y otras dos para detectar que la convergencia ha ocurrido. Estas pasadas se resumen en la figura 10.67(a) a (e). La primera pasada es hacia adelante. Cuando se considera B_2, se descubre que b no puede ser un entero, porque se utiliza como una correspondencia. También se descubre que a se utiliza como entero en B_3, y por tanto sólo se puede hacer corresponder con *ent* en μ_3 y μ_4. Estas observaciones se resumen en la figura 10.67(a).

	a	b	c
μ_1	11111	11111	*indef*
μ_2	11111	01111	11111
μ_3	10000	01111	11111
μ_4	10000	01111	11111

(a) HACIA ADELANTE

	a	b	c
μ_1	10000	01100	*indef*
μ_2	10000	01111	11111
μ_3	10000	01111	11111
μ_4	10000	01111	11111

(b) HACIA ATRAS

	a	b	c
μ_1	10000	01100	*indef*
μ_2	10000	01100	11111
μ_3	10000	01100	11111
μ_4	10000	01100	11111

(c) HACIA ADELANTE

	a	b	c
μ_1	10000	01100	*indef*
μ_2	10000	01100	10000
μ_3	10000	01100	10000
μ_4	10000	01100	11111

(d) HACIA ATRAS

	a	b	c
μ_1	10000	01000	*indef*
μ_2	10000	01000	10000
μ_3	10000	01000	10000
μ_4	10000	01000	10000

(e) HACIA ADELANTE

Fig. 10.67. Simulación del algoritmo 10.19 sobre el grafo de flujo de la figura 10.66.

La segunda pasada, que se muestra en la figura 10.67(b), es hacia atrás. En esta pasada, cuando se considera a B_2, se sabe que a debe ser un entero cuando se le aplica b. Por tanto, el tipo de b sólo podría ser *ent* → *ent* o *ent* → *func*. En la tercera pasada, que es hacia adelante, esta limitación del tipo de b se propaga a lo largo de todo el descenso por el grafo de flujo, como se muestra en la figura 10.67(c).

La cuarta pasada, hacia atrás, se muestra en la figura 10.67(d). Aquí, el hecho de que c sea un argumento de b en B_3 indica que c sólo puede ser un entero. Asi-

mismo, cuando se considera B_2, se ve que el resultado de `b(a)` sólo puede ser del tipo de `c`, que es `ent`. Este hecho invalida la posibilidad de que `b` sea de tipo *ent* → *func*. Por último, en la figura 10.67(e), se ve cómo en la quinta pasada, hacia adelante, se propagan estos hechos sobre `b` y `c`. En posteriores pasadas no se pueden realizar nuevas inferencias. En este caso se han reducido los conjuntos de tipos posibles a un solo tipo para cada variable en cada punto; `a` y `c` son enteros y `b` es una correspondencia de enteros a enteros. En general, pueden haber varios tipos posibles para una variable en un punto. □

10.13 DEPURACION SIMBOLICA DE CODIGO OPTIMADO

Un *depurador simbólico* es un sistema que permite observar los datos del programa mientras se ejecuta el programa. Generalmente se llama al depurador cuando ocurre un error en el programa, como un desbordamiento, o cuando se alcanzan determinadas proposiciones, indicadas por el programador en el código fuente. Una vez invocado, el depurador simbólico permite al programador examinar, y tal vez modificar, cualquiera de las variables que suelen ser accesibles al programa en ejecución.

Para que el depurador entienda una orden del usuario como "muéstrame el valor en curso de a", tiene que disponer de cierta información.

1. Debe haber una forma de asociar un identificador como `a` con la posición que representa. Por tanto, la parte de la tabla de símbolos que asigna una posición a cada variable, por ejemplo, un lugar en un área global de datos o en un registro de activación para un procedimiento, debe ser registrada por el compilador y preservada para que el depurador la utilice. Por ejemplo, esta información puede codificarse en el módulo de carga para el programa.

2. Debe haber información sobre el ámbito, de modo que se puedan eliminar ambigüedades de las referencias a un identificador que se declare más de una vez, y de modo que se pueda saber, si se está en algún procedimiento `p`, qué otros datos de procedimientos son accesibles y cómo se encuentran estos datos en la pila o en otra estructura de ejecución. De nuevo, esta información debe tomarse de la tabla de símbolos del compilador y se debe preservar para el uso futuro que de él haga el depurador.

3. El usuario debe saber dónde se está en el programa cuando se invoca el depurador. Esta información es intercalada por el compilador en la llamada al depurador cuando el compilador maneja una invocación del depurador declarada por el usuario. También se obtiene del manejador de excepciones cuando un error de ejecución hace que se llame al depurador.

4. Para que la información sobre la localización en el programa, que se mencionó en 3, tenga sentido para el usuario, debe haber una tabla que asocie cada instrucción en lenguaje de máquina con la proposición fuente de la que procede. El compilador puede preparar esta tabla a medida que genera código.

Aunque el diseño de un depurador simbólico es en sí interesante, sólo se considerarán las dificultades al intentar escribir un depurador simbólico para un compi-

lador optimador. A primera vista, puede parecer que no hace falta depurar un programa optimado. En el ciclo de desarrollo normal, a medida que el usuario depura un programa se utiliza un compilador rápido no optimador hasta que el usuario sepa que el programa fuente es correcto. Sólo entonces se utiliza un compilador optimador.

Desgraciadamente, un programa puede ejecutarse correctamente con un compilador no optimador y después fallar, con los mismos datos de entrada, cuando se compila con el compilador optimador. Por ejemplo, puede existir un error no detectado en el compilador optimador o, reordenando las operaciones, el compilador optimador puede introducir un desbordamiento (*overflow*) o un desbordamiento negativo (*underflow*). Asimismo, incluso los compiladores "no optimadores" pueden realizar algunas transformaciones sencillas, como la eliminación de subexpresiones comunes locales o el reordenamiento de código dentro de un bloque básico, que modifican en gran medida la dificultad de diseñar un depurador simbólico. Por tanto, hay que considerar los algoritmos y las estructuras de datos que se deben utilizar en un depurador simbólico para un compilador optimador que transforme bloques básicos de forma arbitraria.

Deducción de valores de variables en los bloques básicos

Para simplificar, supóngase que tanto el código fuente como el código objeto son secuencias de proposiciones intermedias. Considerar el código fuente como código intermedio no presenta problemas porque este último es más general que el primero. Por ejemplo, sólo se puede permitir al usuario que coloque puntos de ruptura (llamadas al depurador) entre proposiciones fuente, pero aquí se permiten puntos de ruptura después de cualquier proposición intermedia. Considerar el código objeto como código intermedio es cuestionable sólo si el optimador parte una proposición intermedia en varias proposiciones de máquina que quedan separadas. Por ejemplo, por alguna razón, se pueden compilar las dos proposiciones intermedias

```
u := v + w
x := y + z
```

a código donde las dos sumas se realizan en registros diferentes e intercalados. Si este es el caso, se pueden considerar las cargas y almacenamientos de registros como si los registros fueran temporales en el código intermedio, por ejemplo:

```
r1 := v
r2 := y
r1 := r1 + w
r2 := r2 + z
 u := r1
 x := r2
```

Ocurren varios problemas cuando se interactúa con el usuario acerca de un bloque, cuando el usuario piensa que se está ejecutando el bloque fuente pero de hecho se está ejecutando una versión optimada de ese bloque:

1. Supóngase que se está ejecutando el programa que se obtiene al "optimar" un bloque básico del programa fuente y mientras se está ejecutando la proposición `a:=b+c`, ocurre un desbordamiento. Se debe decir al usuario que ha ocurrido un error en una de las proposiciones fuente. Como `b+c` puede ser una subexpresión común que aparezca en dos o más de las proposiciones fuente, ¿a qué proposición se debe atribuir el error?

2. Se plantea un problema más difícil si el usuario del depurador quiere ver el valor "en curso" de alguna variable `d`. En el programa optimado, se pudo haber asignado `d` en alguna proposición *s*. Pero en el programa fuente, *s* puede ir después de la proposición en la que se invocó al depurador, de modo que el valor de `d` disponible para el depurador no es el que el usuario piensa que es el valor "en curso" de `d` según el listado del código fuente. De manera similar, *s* puede preceder la proposición que invoca al depurador, pero en el código fuente hay otra asignación a `d` entre ellas, de modo que el valor de `d` disponible para el depurador está anticuado. ¿Es posible poner a disposición del usuario el valor correcto de `d`? Por ejemplo, ¿podría ser el valor de alguna otra variable en la versión optimada, o se podría calcular a partir de los valores de otras variables?

3. Por último, si el usuario coloca un punto de ruptura después de una proposición del código fuente, ¿cuándo se debe entregar el control al depurador durante la ejecución del código optimado?

Una solución puede ser ejecutar la versión sin optimar del bloque junto con la versión optimada, para hacer disponibles los valores correctos de cada variable en todo momento. Se rechaza esta "solución", porque las sutilezas de los errores ocultos, especialmente los errores introducidos por el compilador, pueden desaparecer cuando las instrucciones que provocaron el problema están separadas una de otras en tiempo o espacio.

La solución que aquí se adopta es proporcionar suficiente información acerca de cada bloque al depurador para que finalmente pueda contestar la pregunta: ¿es posible proporcionar el valor correcto de la variable `a`?, y si lo es, ¿cómo? La estructura utilizada para abarcar esta información es el GDA del bloque básico, con anotaciones sobre información acerca de las variables que guardan el valor correspondiente a un nodo en el GDA, y en qué momentos en el programa fuente y en el optimado. La notación

$$\mathtt{a}: i - j$$

asociada a un nodo significa que el valor representado por ese nodo está almacenado en la variable `a` desde el comienzo de la proposición *i* a lo largo de la parte de proposición *j* justo antes de que se produzca su asignación. Si $j = \infty$, entonces `a` contiene este valor hasta el final del bloque.

Ejemplo 10.53. En la figura 10.68(a) se observa un bloque básico de código fuente, y en la figura 10.68(b) hay una posible versión "optimada" de ese código. En la figura 10.69 se muestra el GDA para uno u otro bloque, con indicaciones de los rangos en los que las variables contienen esos valores en el código fuente y en el opti-

```
(1)  c := a+b        (1′)  d := a+b
(2)  d := c          (2′)  t := b*e
(3)  c := c-e        (3′)  a := d-e
(4)  a := d-e        (4′)  b := d/t
(5)  b := b*e        (5′)  c := a
(6)  b := d/b
```

(a) (b)

Fig. 10.68. Código fuente y código optimado.

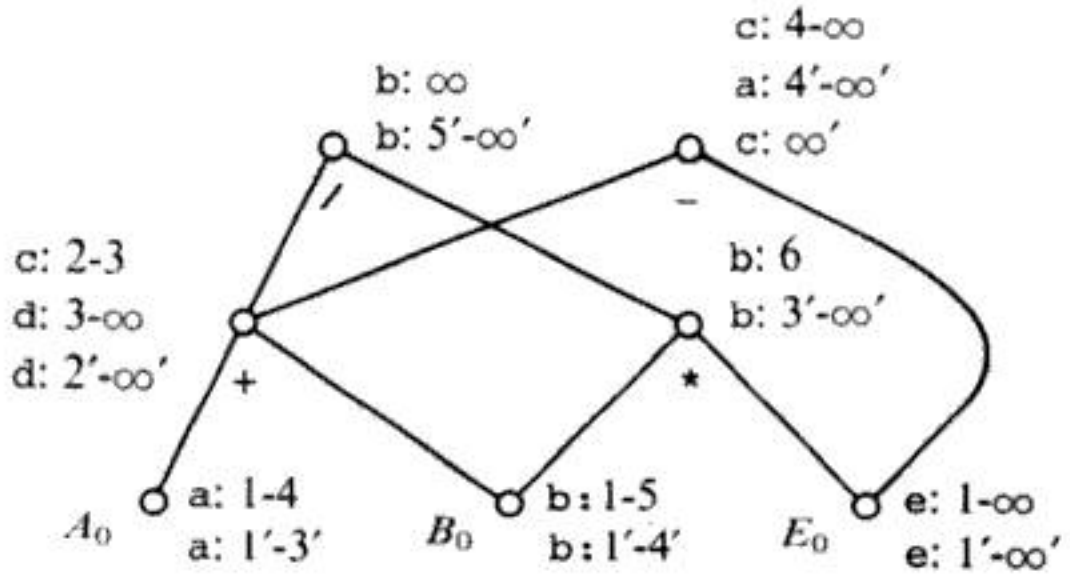


Fig. 10.69. GDA con anotaciones.

mado. Las primas se utilizan para indicar que el rango de una proposición está en el código optimado. Por ejemplo, el nodo etiquetado con + es el valor de c en el código fuente desde el comienzo de la proposición (2) hasta justo antes de la asignación en la proposición (3). También es el valor de d en el código fuente desde el comienzo de la proposición (3) hasta el fin. Además, el mismo nodo es el valor de d en el código optimado desde la proposición (2′) hasta el fin. □

Ahora se puede contestar a la primera pregunta de antes. Supóngase que ocurre un error, como un desbordamiento, mientras se está ejecutando la proposición j' del código optimado. Como el mismo valor sería calculado por cualquier proposición fuente que calcula el mismo nodo del GDA que la proposición j', tiene sentido informar al usuario que el error ocurrió en la primera proposición fuente que calcula este nodo. Por tanto, en el ejemplo 10.53, si el error ocurrió en la proposición 1′, 2′, 3′ ó 4′, habría que informar que ocurrió en la proposición 1, 5, 3 y 6, respectivamente. No puede ocurrir ningún error en la proposición 5′, porque no se calcula ningún valor. Se posponen los detalles de cómo se calculan las proposiciones correspondientes hasta el ejemplo 10.54, más adelante.

También se puede responder a la segunda pregunta. Supóngase que se está en la proposición j' del código optimado y que se informa al usuario que el control se encuentra en la proposición i del código fuente, donde se produjo un error. Si el usuario pide ver el valor de una variable x, se debe encontrar una variable y (con

frecuencia, pero no siempre, y es x) tal que el valor de x en la proposición i del código fuente sea el mismo nodo del GDA que y en la proposición j' en el código optimado. Se inspecciona el GDA para ver el nodo que representa el valor de x en i y se puede leer a partir de ese nodo todas las variables del programa objeto que alguna vez tuvieron ese valor, para comprobar si una de ellas conserva este valor en j'.

Si lo conserva, el proceso habrá terminado; si no, se puede calcular el valor de x en i a partir de otras variables en j'. Sea n el nodo para x en el momento i. Entonces se pueden considerar los hijos de n, m y p, para ver si estos dos nodos representan el valor de alguna variable en el momento j. Si, por ejemplo, hay una variable para m, pero ninguna para p, se pueden considerar los hijos de p, recursivamente. Finalmente, se encuentra una forma de calcular el valor de x en el momento i o se deduce que no existe tal forma. Si se encuentra una forma de calcular los valores de m y p, entonces se calculan y se aplica el operador en n para calcular el valor de x en el momento i[17].

Ejemplo 10.54. Supóngase que cuando se ejecuta el código de la figura 10.68(b), ocurre un error en la proposición 2′. La proposición estaba calculando el nodo etiquetado con $*$ en la figura 10.69, y la primera proposición fuente que calcula ese valor es la proposición 5. Así que se informa de un error en la proposición 5.

En la figura 10.70 se ha tabulado el nodo del GDA que corresponde a cada variable en el código fuente y en el optimado, en el comienzo de las proposiciones 5 y 2′, respectivamente; los nodos se indican mediante su etiqueta, ya sea un símbolo de operador o un símbolo de valor inicial como A_0. También se indica cómo calcular el valor en el momento 5 a partir de los valores de las variables en el momento 2′. Por ejemplo, si el usuario pide el valor de a, se le da el valor del nodo etiquetado con $-$. Ninguna variable tiene ese valor en el momento 2′ pero por fortuna hay variables, d y e, que contienen el valor de cada uno de los hijos del nodo $-$ en el momento 2′, así que se puede imprimir el valor de a calculando el valor de d−e. □

VARIABLE	VALOR EN		OBTENER POR
	MOMENTO 2′	MOMENTO 5	
a	A_0	−	d−e
b	B_0	B_0	b
c	no definido	−	d−e
d	+	+	d
e	E_0	E_0	e
t	no definido		

Fig. 10.70. Valores de variables en los momentos 2′ y 5.

[17] Ocurre una sutileza si el cálculo del valor del nodo n ocasiona otro error. Entonces se debe informar al usuario de que el error ocurrió en realidad antes, en la primera proposición fuente que calcula el valor de n.

Ahora se responderá a la tercera pregunta: cómo tratar las llamadas al depurador insertadas por el usuario. En un sentido, la respuesta es trivial; si el usuario pide una ruptura después de la proposición i en el programa fuente, se puede detener la ejecución al comienzo del bloque. Si el usuario quiere ver el valor de una variable `x` después de la proposición i, se consulta el GDA con anotaciones para ver qué nodo representa el valor deseado de `x` y se calcula ese valor a partir de los valores iniciales de las variables para dicho bloque.

Por otra parte, se puede dejar menos trabajo al depurador y evitar algunas situaciones donde los intentos de calcular un valor conducen a errores que deben ser anunciados al usuario, si se pospone la llamada al depurador cuanto sea posible. Es fácil calcular la última proposición j' en el programa optimado, de modo que se llama al depurador después de j' y se simula ante el usuario que la llamada se hizo después de la proposición i del programa fuente. Para encontrar j', sea S el conjunto de nodos del GDA que corresponden al valor de alguna variable del programa fuente inmediatamente después de la proposición i. El usuario puede pedir que se calcule cualquier valor en S. Por tanto, se puede hacer la ruptura después de la proposición j' del código optimado sólo si para cada nodo n en S hay alguna $k' > j'$ tal que se asocie alguna variable con el nodo n en el momento k' en el código optimado. Para entonces se sabe que el valor de n está disponible inmediatamente después de la proposición j' o se calculará en algún momento después de la proposición j'. En el primer caso, resulta trivial calcular el valor de n si se hace la ruptura después de j', mientras que en el segundo caso se sabe que los valores disponibles después de j' son suficientes para calcular n de algún modo.

Ejemplo 10.55. Considérense de nuevo el código fuente y el optimado de la figura 10.68 y supóngase que el usuario inserta un punto de ruptura después de la proposición (3) del código fuente. Para encontrar el conjunto S, se inspecciona el GDA de la figura 10.69, y se ve qué nodos tienen asociadas variables del programa fuente en el momento 4. Estos nodos son los etiquetados con A_0, B_0, E_0, + y − en esa figura.

Ahora se observa de nuevo el GDA para encontrar la mayor j' tal que cada uno de los nodos de S tenga asociada alguna variable del código optimado en un momento estrictamente mayor que j'. Los nodos etiquetados con +, − y E_0, no plantean problemas, puesto que sus valores son transportados por las variables `d`, `a` y `e`, respectivamente, en el momento ∞'. Los nodos A_0 y B_0 limitan el valor de j' y el primero de ellos en perder su valor es A_0, cuyo valor es destruido por la proposición 3′. Por tanto, $j' = 2'$ es el mayor valor posible de j'; es decir, si el usuario pide una ruptura después de la proposición fuente 3, se le puede dar el punto de ruptura después de la proposición 2′.

El lector debe conocer una sutileza en el ejemplo 10.55, para lo cual no existe una solución realmente buena. Si se ejecuta el código optimado a través de la proposición 2′ antes de llamar al depurador, un error en el cálculo de `b*e` en la proposición 2′ (por ejemplo, un desbordamiento negativo) puede ocasionar que se invoque al depurador antes de la llamada que se pensaba realizar. Sin embargo, como el cálculo correspondiente a la proposición 2′ no se produce hasta la proposición 5 en el programa fuente, se dirá al usuario que el error ocurrió en la proposición 5. El usuario se preguntará cómo se llegó a la proposición 5 sin llamar al depurador en la

proposición 3. Probablemente la mejor solución a este problema sea no permitir que j' sea tan grande que haya una proposición k' del código optimado, con $k' \leqslant j'$, tal que el código fuente no calcule el valor calculado por k' hasta después de la proposición i en la que se colocó el punto de ruptura.

Efectos de la optimación global

Cuando el compilador realiza optimaciones globales, hay problemas más difíciles que el depurador simbólico debe resolver y a menudo no hay manera de encontrar el valor correcto de una variable en un punto. Dos transformaciones importantes que no ocasionan problemas significativos son la eliminación de variables de inducción y la eliminación de subexpresiones comunes globales; en ambos casos el programa se puede localizar en unos pocos bloques y considerar de la forma anteriormente estudiada.

Eliminación de variables de inducción

Si se elimina una variable de inducción `i` de un programa en favor de un miembro de la familia de `i`, por ejemplo `t`, entonces hay una función lineal que relaciona `i` y `t`. Además, si se siguen los métodos de la sección 10.7, el código optimado modificará `t` en exactamente aquellos bloques en que se modifica `i`, de modo que siempre se cumple la relación lineal entre `i` y `t`. Por tanto, después de tener en cuenta el reordenamiento de las proposiciones dentro de un bloque que asigna a `t` (y en el programa fuente, asigna a `i`) se puede proporcionar al usuario el valor "en curso" de `i` mediante una transformación lineal sobre `t`.

Hay que tener cuidado si `i` no se define antes del lazo, puesto que `t` siempre será asignado antes de la entrada al lazo, y se puede por tanto proporcionar un valor para `i` en un punto del programa donde el usuario espera que `i` esté indefinida. Por fortuna, es habitual que una variable del programa fuente que sea una variable de inducción que deba inicializarse antes del lazo, y sólo las variables generadas por el compilador (cuyos valores no puede pedir el usuario) estarán indefinidas a la entrada del lazo. Si este no es el caso para alguna variable de inducción `i`, entonces el problema es similar al de traslado de código que se estudiará más adelante.

Eliminación de subexpresiones comunes globales

Si se realiza una eliminación de subexpresiones comunes globales para la expresión `a+b`, también afectan un número limitado de bloques de formas simples. Si `t` es la variable utilizada para contener el valor de `a+b`, entonces en algunos bloques que calculen `a+b` se puede sustituir

```
c := a+b
```

por

```
t := a+b
c := t
```

Esta clase de modificación se puede llevar a cabo con los métodos para bloques básicos que ya se han estudiado.

En otros bloques, un uso como `d:=a+b` puede ser sustituido por un uso `d:=t`. Para manejar esta situación con los métodos anteriores, sólo hay que observar en el GDA para este bloque que el valor de `t` permanece todo el tiempo en el valor del nodo para `a+b` (que aparecerá en el GDA para el código fuente, pero que, por lo demás, no aparecería en el GDA para el código optimado).

Traslado de código

Hay otras transformaciones que no son fáciles de realizar. Por ejemplo, supóngase que se saca una proposición

s: `a:=b+c`

de un lazo porque es un invariante del lazo. Si se llama al depurador dentro del lazo, no se sabe si la proposición *s* ya ha sido ejecutada en el programa fuente y por tanto no se puede saber si el valor en curso de `a` es el que el usuario vería en el programa fuente.

Una posibilidad es insertar en el código optimado una nueva variable que, dentro del lazo, indique si se ha asignado `a` en el lazo (lo que sólo puede ocurrir en la posición anterior de la proposición *s*). Sin embargo, esta estrategia puede no siempre ser adecuada, porque para una fiabilidad absoluta, se debe utilizar sólo el código real, no una versión del código construida especialmente para la depuración.

Sin embargo, hay un caso especial común que se puede mejorar. Supóngase que el bloque *B* que contiene a la proposición *s* en el programa fuente divide el lazo en dos conjuntos de nodos: los que dominan a *B* y los dominados por él. Además, supóngase que todos los predecesores del encabezamiento son dominados por *B*, como se sugiere en la figura 10.71. Entonces, la primera vez que se pase por los bloques

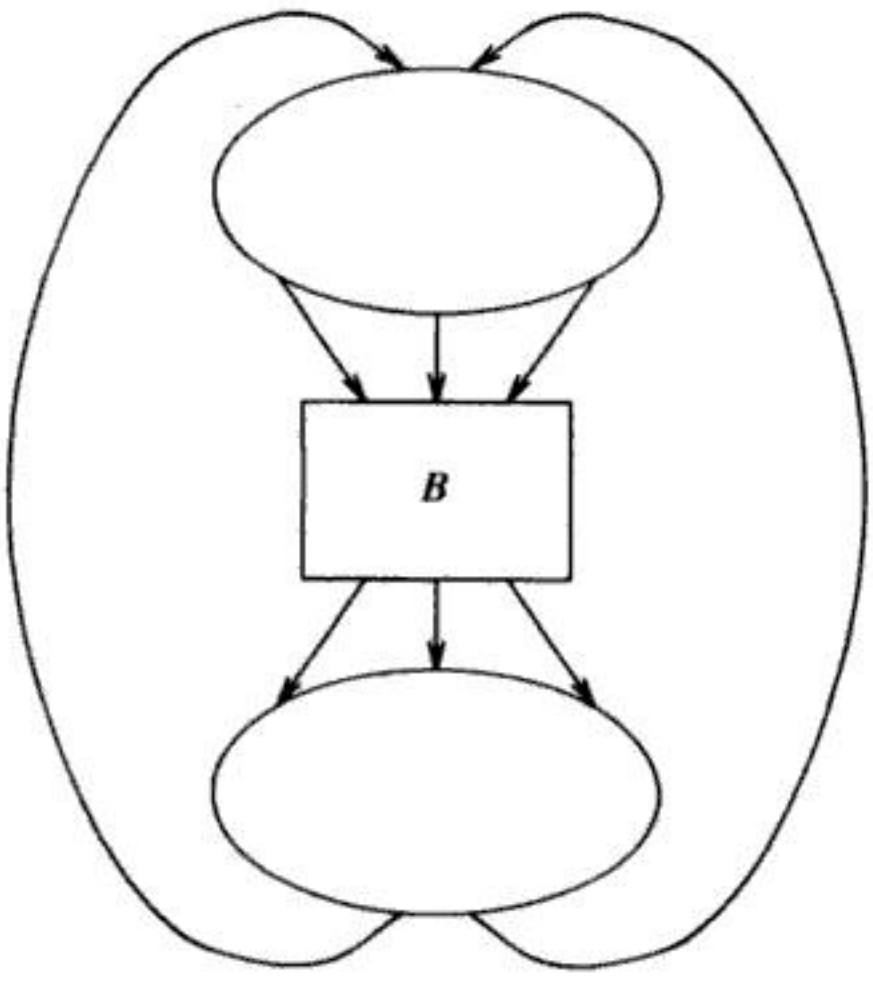


Fig. 10.71. Un bloque que divide un lazo en dos partes.

que dominan a B, se puede suponer que a no ha sido asignada en el lazo, mientras que la primera vez que se pase por los bloques que domina B, a ha sido asignada en la proposición s. Por supuesto, a partir de la segunda vez inclusive que se pase por lazo, a habrá sido asignada en s.

Si la llamada al depurador es ocasionada por un error durante la ejecución, es muy probable que el error quede descubierto la primera vez que se recorre el lazo. Si ese es el caso, entonces sólo hay que saber si se está encima o debajo de B en la figura 10.71. Entonces, se sabe si el valor de a es el definido en s, en cuyo caso basta imprimir el valor de a producido por el código optimado, o si el valor de a es el que tenía a a la entrada del lazo en la versión fuente del programa. En el segundo caso, poco se puede hacer, excepto si

1. el depurador tiene a su disposición la información sobre las definiciones de alcance, tanto para el programa fuente como para el programa optimado;
2. hay una definición única de a que alcanza el encabezamiento en el programa fuente, y
3. esa definición es también la única definición de una variable x que alcanza el punto donde se llamó al depurador.

Si se cumplen todas estas condiciones, entonces se puede imprimir el valor de x y decir que es el valor de a.

Se debe advertir al lector que esta línea de razonamiento no se cumple si el depurador fue llamado por un punto de ruptura insertado por el usuario, por tanto, no hay razón para sospechar que se está recorriendo el lazo por primera vez. Sin embargo, si se están utilizando puntos de ruptura insertados por el usuario, entonces también puede ser razonable insertar código dentro del programa depurado para ayudar al depurador a saber si ésta es la primera vez que se recorre el lazo, una solución que ya se mencionó pero que puede no ser adecuada porque altera el código optimado.

EJERCICIOS

10.1 Considérese el programa para la multiplicación de matrices de la figura 10.72.

```
begin
    for i := 1 to n do
        for j := 1 to n do
            c[i,j] := 0;
    for i := 1 to n do
        for j := 1 to n do
            for k := 1 to n do
                c[i,j] := c[i,j] + a[i,k] * b[k,j]
end
```

Fig. 10.72. Programa para multiplicación de matrices.

a) Suponiendo que a a, b y c se les ha asignado memoria estática y que hay cuatro bytes por palabra en una memoria direccionada por bytes, prodúzcanse proposiciones de tres direcciones para el programa de la figura 10.72.
b) Genérese código de máquina objeto para las proposiciones de tres direcciones.
c) Constrúyase un grafo de flujo a partir de las proposiciones de tres direcciones.
d) Elimínense las subexpresiones comunes de cada bloque básico.
e) Encuéntrense los lazos en el grafo de flujo.
f) Sáquense los cálculos invariantes de los lazos.
g) Encuéntrense las variables de inducción de cada lazo y elimínense cuando sea posible.
h) Genérese código de máquina objeto a partir del grafo de flujo de g). Compárese el código con el producido en b).

10.2 Calcúlense las definiciones de alcance y las cadenas de uso y definición para el grafo de flujo original del ejercicio 10.1c) y el grafo de flujo final de 10.1g).

10.3 El programa de la figura 10.73 cuenta los números primos desde 2 a n utilizando el método de la criba sobre una matriz debidamente grande.

```
begin
    read n;
    for i := 2 to n do
        a[i] := true;    /(*) inicialización */
    cuenta := 0;
    for i := 2 to n ** .5 do
        if a[i] then    /* i es primo */
            begin
                cuenta := cuenta + 1;
                for j := 2 * i to n by i do
                    a[j] := false
                /* j es divisible por i */
            end;
    print cuenta
end
```

Fig. 10.73. Programa para calcular números primos.

a) Tradúzcase el programa de la figura 10.73 a proposiciones de tres direcciones suponiendo que a tiene asignada memoria estática.
b) Genérese código de máquina objeto a partir de las proposiciones de tres direcciones.
c) Constrúyase un grafo de flujo a partir de las proposiciones de tres direcciones.
d) Muéstrese el árbol de dominación para el grafo de flujo de a).

e) Para el grafo de flujo de c), indíquense las aristas de retroceso y sus lazos naturales.
f) Trasládense los cálculos invariantes fuera de los lazos utilizando el algoritmo 10.7.
g) Elimínense las variables de inducción cuando sea posible.
h) Propáguense las proposiciones de copia cuando sea posible.
i) ¿Es posible la compresión de lazos? Si lo es, hágase.
j) En el supuesto de que n siempre sea par, despliéguense lazos internos una vez cada uno. ¿Qué nuevas optimaciones son posibles ahora?

10.4 Repítase el ejercicio 10.3 suponiendo que a a se le ha asignado memoria dinámica, siendo apn un apuntador a la primera palabra de a.

10.5 Para el grafo de flujo de la figura 10.74, calcúlense:
a) Las cadenas de uso y definición y de definición y uso.
b) Las variables activas al final de cada bloque.
c) Las expresiones disponibles.

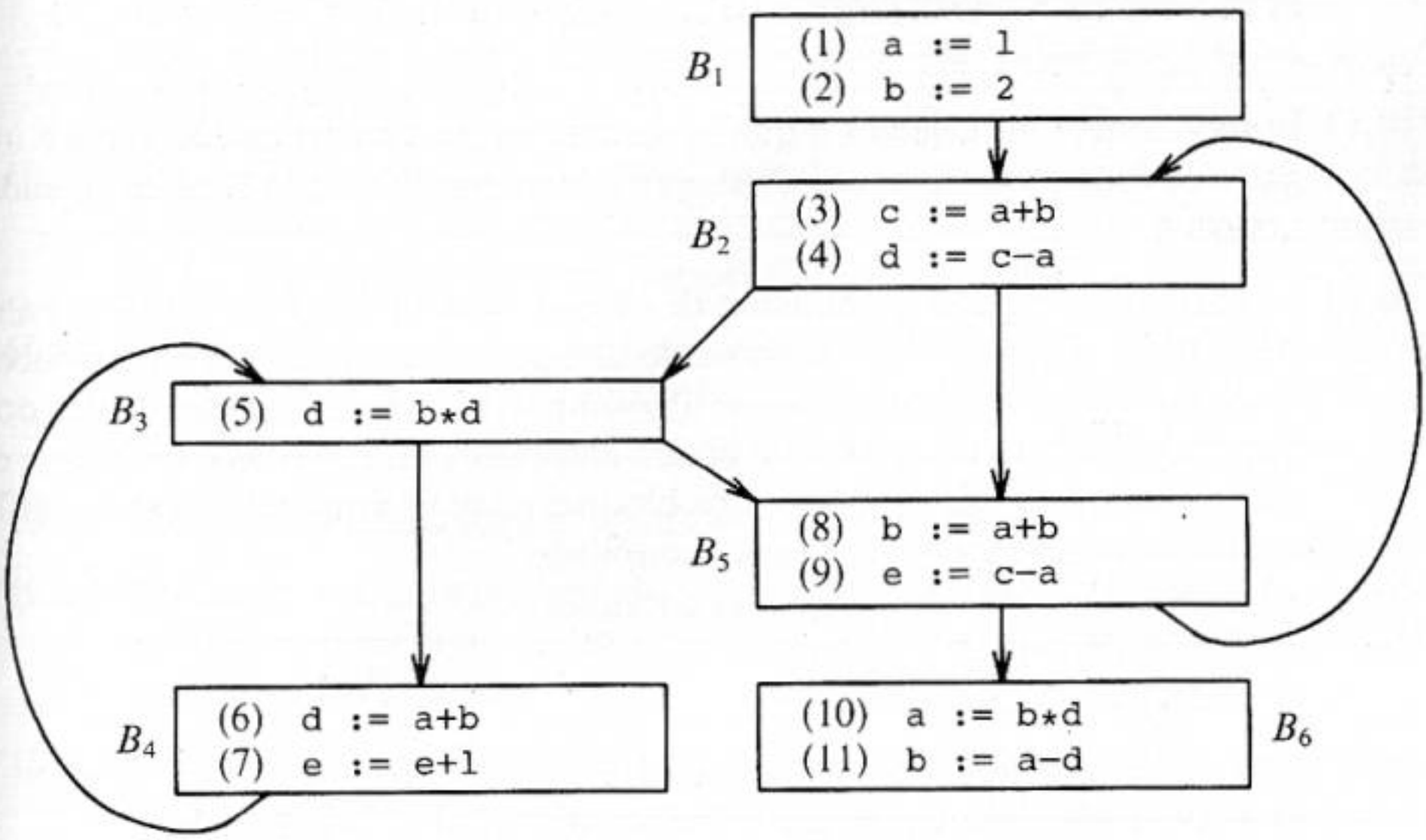


Fig. 10.74. Grafo de flujo.

10.6 ¿Es posible realizar un cálculo previo de una constante en la figura 10.74? Si lo es, hágase.

10.7 ¿Hay subexpresiones comunes en la figura 10.74? Si es así, elimínense.

10.8 Se dice que una expresión *e* está *muy ocupada* en el punto *p* si, independientemente del camino que se tome desde *p*, la expresión *e* será evaluada antes de que se definan cualquiera de sus operandos. Proporciónese un algoritmo de flujo de datos al estilo de la sección 10.6 para encontrar todas las expresiones muy ocupadas. ¿Qué operador de confluencia utiliza usted?, y ¿la

propagación se hace hacia adelante o hacia atrás? Aplique su algoritmo al grafo de flujo de la figura 10.74.

***10.9** Si la expresión *e* está muy ocupada en el punto *p*, se puede *elevar e* calculándola en *p* y preservando su valor para que pueda ser utilizado posteriormente. (*Nota*: Esta optimación generalmente no ahorra tiempo, pero puede ahorrar espacio.) Proporciónese un algoritmo para elevar las expresiones muy ocupadas.

10.10 ¿Existen expresiones que se puedan elevar en la figura 10.74? Si las hay, elévelas.

10.11 Cuando sea posible propáguense los pasos de copia introducidos en las modificaciones de los ejercicios 10.6, 10.7 y 10.10.

10.12 Un *bloque básico ampliado* es una secuencia de bloques $B_1, \ldots, B_k$ tal que para $1 \leq i \leq k$, B_i es el único predecesor de B_{i+1} y B_1 no tiene un predecesor único. Encuéntrese el bloque básico ampliado finalizando cada nodo en
a) la figura 10.39,
b) el grafo de flujo construido en el ejercicio 10.1c),
c) la figura 10.74.

***10.13** Proporciónese un algoritmo que se ejecute en un tiempo de $O(n)$ sobre un grafo de flujo para encontrar el bloque básico ampliado que finaliza en cada nodo.

10.14 Se puede realizar una optimación de código entre bloques sin ningún análisis de flujo de datos considerando cada bloque básico ampliado como si fuera un bloque básico. Proporciónense algoritmos para realizar las siguientes optimaciones dentro de un bloque básico ampliado. En cada caso, indíquese el efecto que puede tener sobre otros bloques básicos ampliados una modificación dentro de un bloque básico ampliado.
a) Eliminación de subexpresiones comunes.
b) Cálculo previo de constantes.
c) Propagación de copias.

10.15 Para el grafo de flujo del ejercicio 10.1c):
a) Encuéntrese una secuencia de reducción $T_1 - T_2$ para el grafo.
b) Encuéntrese la secuencia de grafos de intervalo.
c) ¿Cuál es el grafo de flujo límite? ¿Es reducible el grafo de flujo?

10.16 Repítase el ejercicio 10.15 para el grafo de flujo de la figura 10.74.

****10.17** Demuéstrese que las siguientes condiciones son equivalentes (son definiciones alternativas de "grafo de flujo reducible").
a) El límite de la reducción $T_1 - T_2$ es un solo nodo.
b) El límite del análisis de intervalo es un solo nodo.
c) El grafo de flujo puede tener sus aristas divididas en dos clases; una forma un grafo acíclico y la otra, las aristas "de retroceso", consta de aristas cuyas cabezas dominan a sus colas.
d) El grafo de flujo no tiene subgrafos con forma como la que se muestra en

la figura 10.75. Aquí, n_0 es el nodo inicial, y n_0, a, b y c son todos distintos, con la excepción de que es posible que $a = n_0$. Las flechas representan caminos de nodos disjuntos (excepto los puntos extremos, por supuesto).

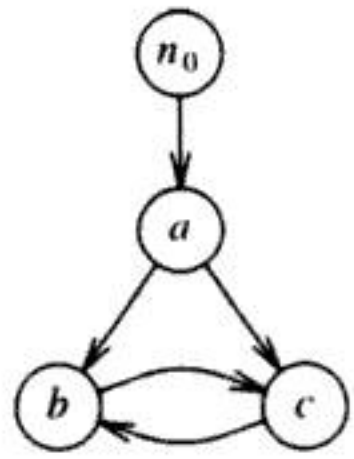


Fig. 10.75. Subgrafos prohibidos para grafos de flujo reducibles.

10.18 Proporciónense algoritmos para calcular (a) expresiones disponibles y (b) variables activas para el lenguaje con apuntadores que se estudió en la sección 10.8. Asegúrese que hace suposiciones conservadoras acerca de *gen*, *desact*, *uso* y *def* en (b).

10.19 Proporciónese un algoritmo para calcular las definiciones de alcance entre procedimientos utilizando el modelo de la sección 10.8. De nuevo, asegúrese de que hace aproximaciones conservadoras a la verdad.

10.20 Supóngase que los parámetros se pasan por valor en lugar de por referencia. ¿Pueden dos nombres ser sinónimo uno de otro? ¿Qué pasa si se utiliza enlace de copia y restauración?

10.21 ¿Cuál es la profundidad del grafo de flujo del ejercicio 10.1c)?

****10.22** Demuéstrese que la profundidad de un grafo de flujo reducible nunca es menor que el número de veces que se debe realizar el análisis de intervalos para producir un solo nodo.

***10.23** Generalícese el algoritmo de la sección 10.8 de análisis de flujo de datos basado en la estructura para un marco de flujo de datos general en el sentido de la sección 10.11. ¿Qué suposiciones sobre F y $\wedge$ hay que hacer para asegurarse de que un algoritmo funcione?

***10.24** Se obtiene un poderoso e interesante marco para análisis de flujo de datos imaginando que los "valores" que van a ser propagados son todas las particiones de expresiones posibles así que dos expresiones están en la misma clase sólo si tienen el mismo valor. Para evitar listar toda la infinidad de expresiones posibles, se pueden representar dichos valores listando sólo el mínimo de expresiones equivalentes a alguna otra expresión. Por ejemplo, si se ejecutan las proposiciones

```
A := B
C := A + D
```

entonces se tienen las siguientes equivalencias mínimas: `A` ≡ `B` y `C` = `A` + `D`. De aquí se derivan otras equivalencias, como `C` ≡ `B` + `D` y `A` + `E` ≡ `B` + `E`, pero no hay por qué listarlas explícitamente.

a) ¿Cuál es el operador de reunión o de confluencia adecuado para este marco?
b) Proporciónese una estructura de datos para representar valores y un algoritmo para implantar el operador de reunión.
c) ¿Cuáles son las funciones adecuadas para asociarlas con las proposiciones? Explíquese el efecto que debería tener la función asociada con asignaciones como `A:=B+C` en una partición.
d) ¿Es este marco distributivo? ¿Es monótono?

10.25 ¿Cómo se utilizarían los datos reunidos por el marco del ejercicio 10.24 para realizar lo siguiente?

a) Eliminación de subexpresiones comunes.
b) Propagación de copias.
c) Cálculo previo de constantes.

***10.26** Proporciónense pruebas formales de lo siguiente acerca de la relación $\leq$ sobre mallas.

a) $a \leq b$ y $a \leq c$ implica que $a \leq (b \wedge c)$.
b) $a \leq (b \wedge c)$ implica que $a \leq b$.
c) $a \leq b$ y $b \leq c$ implica que $a \leq c$.
d) $a \leq b$ y $b \leq a$ implica que $a = b$.

****10.27** Demuéstrese que la siguiente condición es necesaria y suficiente para que el algoritmo de flujo de datos iterativo, con ordenamiento en profundidad, converja dentro de 2 más las pasadas en profundidad. Para todas las funciones f y g y valor a:

$$f(g(a)) \geq f(a) \wedge g(a) \wedge a$$

10.28 Demuéstrese que los marcos para definiciones de alcance y para expresiones disponibles cumplen la condición del ejercicio 10.27. *Nota*: De hecho, estos marcos convergen en 1 más las pasadas en profundidad.

****10.29** ¿Se deduce la condición del ejercicio 10.27 de la monotonicidad? ¿De la distributividad? ¿A la inversa?

10.30 En la figura 10.76 se observan dos bloques básicos, el primero es código "inicial" y el segundo es una versión optimada.

a) Constrúyanse los GDA para los bloques de la figura 10.76(a) y (b). Compruébese que, suponiendo que sólo `J` está activa a la salida, estos dos bloques son equivalentes.
b) Anótese el GDA con los momentos en que se conoce el valor de cada variable en cada nodo.
c) Indíquese, en caso de que ocurra un error en cada una de las proposiciones (1′) hasta (4′), en qué proposición de la figura 10.76(a) se indicaría que se produjo un error.

```
1)  E := A+B
2)  F := E-C
3)  G := F*D
4)  H := A+B
5)  I := I-C
6)  J := I+G
```

(a) INICIAL

```
1')  E := A+B
2')  E := E-C
3')  F := E*D
4')  J := E+F
```

(b) OPTIMADO

Fig. 10.76. Código inicial y código optimado.

d) Para cada uno de los errores de la parte c), indíquense las variables de la figura 10.76(a) para las que es posible calcular un valor, y cómo se hace.
e) Supóngase que se pueden utilizar leyes algebraicas válidas como "si $a + b = c$ entonces $a = c - b$". ¿Modificaría su respuesta a d)?

10.31 Generalícese el ejemplo 10.14 para tener en cuenta un conjunto arbitrario de proposiciones de ruptura. Generalícese asimismo para que permita proposiciones de continuación (**continue**), que no interrumpen el lazo interno sino que pasan directamente a la siguiente iteración del lazo. *Sugerencia*: Utilícense las técnicas desarrolladas en la sección 10.10 para grafos de flujo reducibles.

10.32 Demuéstrese que en el algoritmo 10.3, los conjuntos *ent* y *sal* de definiciones nunca se reducen. De manera similar, demuéstrese que en el algoritmo 10.4 estos conjuntos de expresiones nunca aumentan.

10.33 Generalícese el algoritmo 10.9 para la eliminación de variables de inducción al caso en que las constantes multiplicativas puedan ser negativas.

10.34 Tómese el algoritmo de determinación a dónde pueden señalar los apuntadores de la sección 10.8 y generalícese al caso en que se permite que apunten a otros apuntadores.

***10.35** Cuando se estiman cada uno de los siguientes conjuntos, indíquese si las estimaciones demasiado grandes o demasiado pequeñas son conservadoras. Explíquese su respuesta en términos del uso previsto para la información.
a) Expresiones disponibles.
b) Las variables modificadas por un procedimiento.
c) Las variables no modificadas por un procedimiento.
d) Las variables de inducción que pertenecen a una determinada familia.
e) Las proposiciones de copia que alcanzan un punto dado.

***10.36** Refínase el algoritmo 10.12 para que calcule los sinónimos de una variable dada en un punto dado.

***10.37** Modifíquese el algoritmo 10.12 para los casos en que los parámetros se pasan
a) por valor,
b) por copia y restauración.

*10.38 Demuéstrese que el algoritmo 10.33 converge en un supraconjunto (no necesariamente propio) de las variables realmente modificadas.

*10.39 Generalícese el algoritmo 10.13 para determinar las variables modificadas en caso de que se permitan variables con procedimientos como valores.

*10.40 Demuéstrese que en cada grafo de intervalos cada nodo representa una región del grafo de flujo original.

10.41 Demuéstrese que el algoritmo 10.16 calcula correctamente el conjunto de dominadores de cada nodo.

*10.42 Modifíquese el algoritmo 10.17 (definiciones de alcance basadas en la estructura) para que calcule las definiciones de alcance sólo para pequeñas regiones designadas, sin que sea necesario que todo el grafo de flujo esté presente en memoria inmediatamente. Asegúrese de que su resultado sea conservador. Adapte su algoritmo a las expresiones disponibles. ¿Cuál de ellos podrá proporcionar información útil?

*10.43 En la sección 10.10 se propuso un aceleramiento al algoritmo 10.17 basado en la combinación de una reducción T_1 y una reducción T_2. Demuéstrese que la modificación se realizó correctamente.

10.44 Generalícese el método iterativo de la sección 10.11 para problemas de flujo hacia atrás.

**10.45 Demuéstrese que cuando converge el algoritmo 10.18, la solución resultante es $\leq$ la solución de *rsc*, demostrando que para cada camino P de longitud i, después de i iteraciones, $ent\ [B_i] \leq f_P\ [\mathrm{T}]$.

10.46 En la figura 10.77 hay un grafo de flujo de un programa en el lenguaje hipotético presentado en la sección 10.12. Encuéntrese la mejor estimación de los tipos de cada variable utilizando el algoritmo 10.19.

NOTAS BIBLIOGRAFICAS

Se puede encontrar información adicional sobre optimación de código en las referencias Cocke y Schwartz [1970], Abel y Bell [1972], Schaefer [1973], Hecht [1977] y Muchnick y Jones [1981]. Allen [1975] proporciona una bibliografía sobre optimación de programas.

En los libros se describen muchos compiladores optimadores. Ershov [1966] estudia uno de los primeros compiladores que utilizaba sofisticadas técnicas de optimación. Lowry y Medlock [1969] y Scarborough y Kolsky [1980] detallan la construcción de compiladores optimadores para optimación de FORTRAN. Busam y Englund [1969] y Metcalf [1982] describen técnicas adicionales para la optimación de FORTRAN. Wulf y colaboradores [1975] estudian el diseño de un compilador optimador influyente para BLISS.

Allen y colaboradores [1980] describen un sistema construido para hacer experimentos con optimaciones de programas. Cocke y Markstein [1980] señalan la

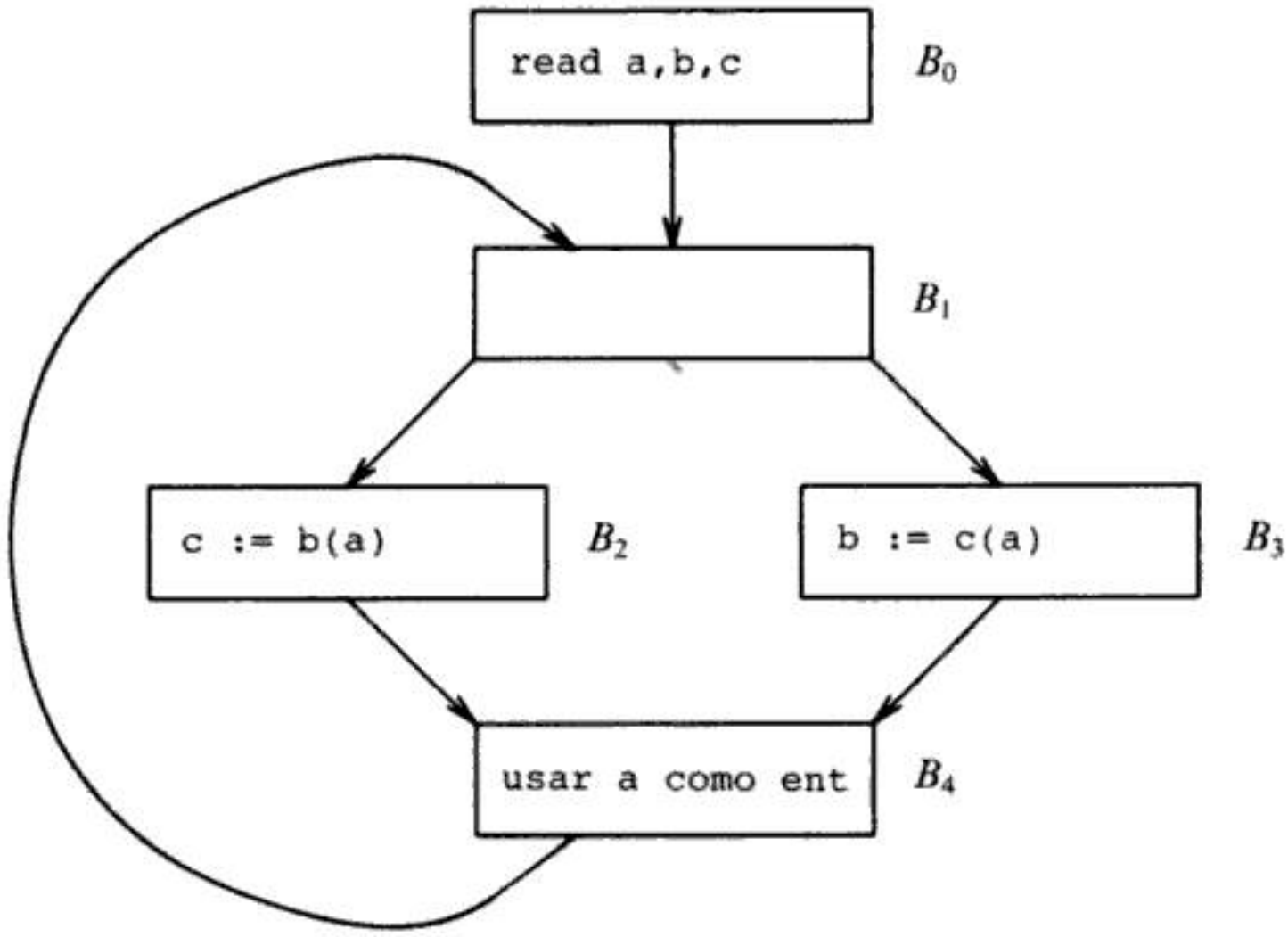


Fig. 10.77. Programa de ejemplo para inferencia de tipos.

efectividad de varias optimaciones para un lenguaje del tipo de PL/I. Anklam, Cutler, Heinen y MacLaren [1982] describen la implantación de transformaciones de optimación utilizadas en compiladores para PL/I y C. Auslander y Hopkins [1982] informan de un compilador para una variante de PL/I que utiliza un algoritmo sencillo para producir código intermedio de bajo nivel que después se mejora con transformaciones de optimación globales. Freudenberger, Schwartz y Sharir [1983] describen experiencias con un optimador para SETL. Chow [1983] informa sobre experimentos con un optimador transportable e independiente de la máquina para Pascal. Powell [1984] describe un compilador de optimación transportable independiente de la máquina para MODULA-2.

El estudio sistemático de las técnicas para análisis de flujo de datos comienza con Allen [1970] y Cocke [1970], desde que publicaron juntos como Allen y Cocke [1976], aunque varios métodos de análisis de flujo de datos ya habían sido utilizados.

El análisis de flujo de datos dirigido por la sintaxis, como se introdujo en la sección 10.5, ha sido empleado en BLISS (Wulf y colaboradores [1975], Geschke [1972]), SIMPL (Zelkowitz y Bail [1974]), y MODULA-2 (Powell [1984]). En las referencias Hecht y Schaffer [1975], Hecht [1977] y Rosen [1977] aparecen estudios adicionales sobre esta familia de algoritmos.

El enfoque iterativo del análisis de flujo de datos presentado en la sección 10.6 se remonta a Vyssotsky (véase Vyssotsky y Wegner [1963]), que utilizó el método en 1962 en un compilador de FORTRAN. El uso de ordenamiento en profundidad para mejorar la eficiencia proviene de Hecht y Ullman [1975].

El enfoque de análisis de intervalos aplicado al análisis de flujo de datos fue iniciado por Cocke [1970]. Kennedy [1971] origina el uso de análisis de intervalos para problemas de flujo de retroceso, como variables activas. Hay motivo para creer, según Kennedy [1976], que los métodos basados en intervalos son más eficientes que los iterativos, si el lenguaje que se está optimando tiende, si acaso, a producir pocos grafos de flujo no reducibles. La variante aquí utilizada, basada en las transformaciones T_1 y T_2, es de Ullman [1973]. Una versión un poco más rápida que aprovecha que la mayoría de las regiones tienen una sola salida apareció en Graham y Wegman [1976].

De Allen [1970] es la definición original de un grafo de flujo reducible, uno que se convierte en un solo nodo después de repetidos análisis de intervalos. Se encuentran caracterizaciones equivalentes en Hecht y Ullman [1972, 1974], Kasyanov [1973], y Tarjan [1974b]. La separación de nodos para grafos de flujo no reducibles se debe a Cocke y Miller [1969].

La idea de que el flujo de control estructruado se modela mediante grafos de flujo reducibles viene expresada en Kosaraju [1974], Kasami, Peterson y Tokura [1973], y Cherniavsky, Henderson y Keohane [1976]. Baker [1977] describe su uso en un algoritmo para estructuración de programas.

El enfoque de la teoría de retículos al análisis de flujo de datos iterativo comenzó en Kildall [1973]. Tennembaum [1974] y Wegbreit [1975] presentan formulaciones similares. A Kam y Ullman [1976] se debe la versión eficiente del algoritmo de Kildall, donde se utiliza orden en profundidad.

Si bien Kildall dio por supuesta la condición de distributividad (que no satisface realmente sus marcos como el marco para el cálculo de constantes del Ejemplo 10.42), la validez de la monotonicidad fue percibida en varios artículos que proporcionaban algoritmos de flujo de datos, como Tennenbaum [1974], Schwartz [1975a, b], Graham y Wegman [1976], Jones y Muchnick [1976], Kam y Ullman [1977], y Cousot y Cousot [1977].

Como algoritmos distintos exigen suposiciones distintas acerca de los datos, la teoría de las propiedades necesarias para ciertos algoritmos fue desarrollada por Kam y Ullman [1977], Rosen [1980] y Tarjan [1981].

Otra indicación que se dedujo del artículo de Kildall fue mejorar los algoritmos para solucionar los problemas de flujo de datos (por ejemplo, el Ejemplo 10.42) que planteó. Una idea clave es que los elementos del retículo no tienen por qué considerarse como atómicos, sino que se puede explotar el hecho de que en realidad son transformaciones de variables a valores. Véase Reif y Lewis [1977] y Wegman y Zadeck [1985]. Kou [1977] explota la idea para problemas más convencionales.

La referencia Kennedy [1981] es una evaluación de las técnicas de análisis de flujo de datos, y Cousot [1981] analiza las ideas de la teoría de retículos.

Gear [1965] introdujo las optimaciones de lazos básicas de traslado de código y una forma limitada de eliminación de variables de inducción. Allen [1969] es un artículo fundamental sobre optimación de lazos; Allen y Cocke [1972] y Waite [1976b] son estudios más amplios de técnicas en el área. Morel y Renvoise [1979] describen un algoritmo que elimina simultáneamente de los lazos cálculos redundantes e invariantes.

El estudio de la eliminación de variables de inducción de la sección 10.7 se basa en Lowry y Medlock [1969]. Véase Allen, Cocke y Kennedy [1981] para algoritmos más poderosos.

Un algoritmo para algunos de los problemas de lazos no estudiados detalladamente, como averiguar si hay un camino desde *a* a *b* que no pase por *c*, se pueden resolver con un algoritmo eficiente de Wegman [1983].

Los pioneros del uso de dominadores, tanto para la detección de lazos como para realizar traslado de código, fueron Lowry y Medlock [1969], aunque ellos atribuyen la idea general a Prosser [1959]. El algoritmo 10.16 para encontrar dominadores fue descubierto independientemente por Purdom y Moore [1972] y Aho y Ullman [1973a]. El uso del ordenamiento en profundidad para acelerar los algoritmos es de Hecht y Ullman [1975], pero la manera asintóticamente más eficiente de realizar el trabajo es de Tarjan [1974a]. Lengauer y Tarjan [1979] describen un algoritmo eficiente para encontrar dominadores que es adecuado para uso práctico.

El estudio de sinónimos y el análisis entre prodedimientos comienza con Spillman [1971] y Allen [1974]. Se han desarrollado algunos métodos más poderosos que los de la sección 10.8. En general, consideran la relación sinónimo a cada punto del programa para evitar algunos de los pares de sinónimos imposibles que el algoritmo "descubre". Estos trabajos incluyen Barth [1978], Banning [1979] y Weihl [1980]. Véase también Ryder [1979] y su construcción de grafos de llamadas.

Hennessy [1981] estudia un aspecto similar al del análisis entre procedimientos, el efecto de las excepciones en el análisis de flujo de datos de programas.

El artículo fundamental de la determinación de tipos mediante análisis de flujo de datos es Tennenbaum [1974], sobre el que se basa nuestro estudio de la sección 10.12. Kaplan y Ullman [1980] proporcionan un algoritmo más poderoso para la detección de tipos.

Hennessy [1982] analiza la depuración simbólica de código optimado de la sección 10.13. Varios artículos han intentado evaluar las mejoras gracias a distintas optimaciones. El valor de una optimación parece depender en gran medida del lenguaje que se está compilando. Quizás el lector quiera consultar el estudio clásico de optimación de FORTRAN en Knuth [1971b], o los artículos de Gajewska [1975], Palm [1975], Cocke y Kennedy [1976], Cocke y Markstein [1980], Chow [1983], Chow y Hennessy [1984] y Powell [1984].

Otro aspecto de la optimación que no ha sido tratado aquí es la optimación de lenguajes de "muy alto nivel", como el lenguaje de teoría de conjuntos SETL, donde se están modificando realmente los algoritmos subyacentes y las estructuras de datos. Una optimación central en este área es la eliminación generealizada de variables de inducción, como en Earley [1975b], Fong y Ullman [1976], Paige y Schwartz [1977] y Fong [1979].

Otro paso clave en la optimación para lenguajes de muy alto nivel es la selección de estructuras de datos; este tema se analiza en Schwartz [1975a, b], Low y Rovner [1976] y Schonberg, Schwartz y Sharir [1981].

Tampoco se han tocado los aspectos sobre la optimación incremental de código, donde pequeñas modificaciones al programa no exigen una reoptimación completa. Ryder [1983] estudia el análisis de flujo de datos incremental, en tanto que Pollock y Soffa [1985] intentan realizar optimación incremental de bloques básicos.

Por último, deben mencionarse algunas de las muchas otras formas en que se han utilizado las técnicas de análisis de flujo de datos. Backhouse [1984] lo utiliza en los grafos de transiciones asociados con analizadores sintácticos para la recuperación de errores. Harrison [1977] y Suzuki e Ishihata [1977] estudian su uso para comprobar los límites de matrices durante la compilación.

Uno de los usos más significativos del análisis de flujo de datos fuera de la optimación de código está en el área de búsqueda estática (durante la compilación) de errores del programa. Fosdick y Osterweil [1976] es un artículo fundamental, en tanto que Osterweil [1981], Adrion, Bronstad y Cherniavsky [1982], y Freudenberger [1984] proporcionan algunos desarrollos más recientes.

CAPITULO 11

¿Quiere escribir un compilador?

Después de haber visto los principios, técnicas y herramientas para el diseño de compiladores, supóngase que se quiere escribir un compilador. Si se tiene un planteamiento previo, la implantación resulta más rápida y sencilla. Este breve capítulo resalta algunos aspectos de la construcción de compiladores. Gran parte del análisis se concentra en la estructura de compiladores utilizando el sistema operativo UNIX y sus herramientas.

11.1 PROYECTANDO UN COMPILADOR

Un compilador nuevo puede ser para un nuevo lenguaje fuente o para producir nuevo código objeto, o para ambos propósitos. Si utiliza el marco de este libro, se obtiene finalmente un diseño para un compilador que consta de un conjunto de módulos. Diversos factores impactan el diseño e implantación de estos módulos.

Aspectos del lenguaje fuente

El "tamaño" de un lenguaje influye en el tamaño y número de módulos. Aunque no hay una definición precisa para el tamaño de un lenguaje, es evidente que un compilador para un lenguaje como Ada o PL/I es más grande y difícil de implantar que un compilador para un lenguaje pequeño como RATFOR (un preprocesador de FORTRAN "racional", Kernighan [1975]) o EQN (un lenguaje para tipografía de matemáticas).

Otro factor importante es hasta qué punto cambiará el lenguaje fuente durante el curso de la construcción del compilador. Aunque la especificación del lenguaje fuente puede parecer inmutable, pocos lenguajes permanecen iguales durante la vida del compilador. Incluso un lenguaje maduro evoluciona, aunque sea lentamente. FORTRAN, por ejemplo, ha cambiado considerablemente desde 1957; las proposiciones de iteración, Hollerith y condicionales en FORTRAN 77 son bastante distintas de las del lenguaje original. Rosler [1984] informa sobre la evolución de C.

Por otro lado, un lenguaje nuevo, experimental, puede sufrir cambios drásticos al ser implantado. Una forma de crear un nuevo lenguaje consiste en hacer evolu-

cionar un compilador para un prototipo operativo del lenguaje hasta convertirse en uno que satisfaga las necesidades de cierto grupo de usuarios. Muchos de los "pequeños" lenguajes desarrollados con el sistema UNIX como AWK o EQN se crearon de esta forma.

Por tanto, el escritor de un compilador puede prever al menos cierto cambio en la definición del lenguaje fuente durante la vida de un compilador. El diseño modular y el uso de herramientas puede ayudar al escritor del compilador a enfrentarse a este cambio. Por ejemplo, utilizar generadores para implantar el analizador léxico y el analizador sintáctico de un compilador le permite al escritor del compilador acomodar más fácilmente modificaciones sintácticas en la definición del lenguaje que si el analizador léxico y el sintáctico están escritos directamente en código.

Aspectos del lenguaje objeto

La naturaleza y limitaciones del lenguaje objeto y el entorno de ejecución deben ser tratados con cuidado porque también tienen una fuerte influencia en el diseño del compilador y en las estrategias de generación en el diseño del compilador y en las estrategias de generación de código que debe utilizar. Si el lenguaje objeto es nuevo, quien escribe el compilador deberá asegurarse de que sea correcto y de que se comprendan sus secuencias de tiempos. Una máquina nueva o un ensamblador nuevo pueden tener errores no detectados que quizás descubra el compilador. Los errores en el lenguaje objeto pueden agravar la tarea de depurar el compilador mismo.

Un lenguaje fuente de éxito se implantará probablemente en varias máquinas objeto. Si un lenguaje persiste, los compiladores para el lenguaje necesitarán producir código para varias generaciones de máquinas objeto. Parece seguro que se producirá una mayor evolución en el hardware de las máquinas, así que es probable que los compiladores redestinables tengan futuro. Por tanto, el diseño del lenguaje intermedio es importante, ya que confina los detalles específicos de la máquina a un número reducido de módulos.

Criterios de rendimiento

Hay varios aspectos del rendimiento de los compiladores: la velocidad del compilador, la calidad del código, los diagnósticos de error, la transportabilidad y el mantenimiento. La separación entre estos criterios no es muy clara, y la especificación del compilador puede dejar muchos de estos parámetros sin especificar. Por ejemplo, ¿es más importante la velocidad de compilación que la velocidad del código objeto? ¿Cuánta importancia tienen los buenos mensajes de error y la recuperación de errores?

La velocidad del compilador se puede lograr reduciendo el número de módulos y pasadas cuanto sea posible, quizás hasta el punto de generar lenguaje de máquina en una pasada. Sin embargo, este enfoque puede no producir un compilador que genere código objeto de alta calidad, ni uno que sea particularmente fácil de mantener.

Hay dos aspectos referentes a la transportabilidad: la redestinación y la relocalización. Un compilador redestinable es el que se puede modificar fácilmente para que genere código para un nuevo lenguaje objeto. Un compilador reubicable es el que se

puede transportar fácilmente para su ejecución en una máquina nueva. Un compilador transportable puede no ser tan eficiente como un compilador diseñado para una máquina específica, porque el compilador para una sola máquina puede hacer suposiciones específicas acerca de la máquina objeto que un compilador transportable no puede.

11.2 ASPECTOS DEL DESARROLLO DE COMPILADORES

Hay varios enfoques generales que puede adoptar quien escribe un compilador para implantar un compilador. El más sencillo consiste en redestinar o reubicar un compilador ya existente. Si no existe ya un compilador adecuado, el escritor del compilador puede adoptar la organización de un compilador conocido para un lenguaje similar e implantar los componentes correspondientes, utilizando herramientas para la generación de componentes o implantándolos a mano. No es muy frecuente que se requiera una organización completamente nueva para un compilador.

Independientemente del enfoque que se adopte, la escritura de compiladores es un ejercicio de ingeniería de *software*. Las lecciones de otros esfuerzos de *software* (véase por ejemplo Brooks [1975]) pueden aplicarse para mejorar la fiabilidad y mantenimiento del producto final. Un diseño que asimile con facilidad las modificaciones permitirá que el compilador evolucione con el lenguaje. El uso de herramientas para la construcción de compiladores puede ser una ayuda significativa a este respecto.

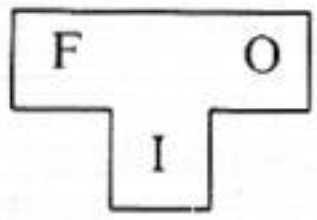
Arranque

Un compilador es un programa lo bastante complejo que se desearía escribir en un lenguaje más asequible que el lenguaje ensamblador. En el entorno de programación UNIX, los compiladores se escriben generalmente en C. Incluso los compiladores de C se escriben en C. La base del *arranque* es utilizar las ventajas de un lenguaje para compilarse a sí mismo. Se estudiará el uso del arranque para crear compiladores y trasladarlos de una máquina a otra modificando la etapa final. Las ideas basicas para el arranque se conocen desde mediados de los años cincuenta. (Strong y colaboradores [1958]).

El arranque puede dar lugar a la pregunta: "¿Cómo se compiló el primer compilador?" algo así como ¿qué fue antes, el huevo o la gallina? pero la primera pregunta es más fácil de contestar. Para ello considérese cómo LISP se convirtió en un lenguaje de programación. McCarthy [1981] señala que a finales de 1958 LISP se utilizó como una notación para escribir funciones; después se traducían à mano a lenguaje ensamblador y se ejecutaban. La implantación de un intérprete para LISP se produjo inesperadamente. McCarthy quería demostrar que LISP era una notación para describir funciones "mucho más clara que las máquinas de Turing o las definiciones recursivas generales utilizadas en la teoría de funciones recursivas", de modo que escribió una función *eval*[*e*, *a*] en LISP que tomara una expresión *e* en LISP como argumento. S. R. Russell observó que *eval* podía servir como un intérprete para LISP, la codificó a mano y así creó un lenguaje de programación con un

intérprete. Como se mencionó en la sección 1.1, en lugar de generar código objeto, un intérprete realiza de hecho las operaciones del programa fuente.

A efectos de arranque, un compilador se caracteriza por tres lenguajes: el lenguaje fuente F que compila, el lenguaje objeto O para el que genera código y el lenguaje de implantación I en el que está escrito. Los tres lenguajes se representan utilizando el siguiente diagrama, llamado un *diagrama T*, por su forma (Bratman [1961]).



Dentro del texto, el diagrama T anterior se abrevia: F_IO. Los tres lenguajes, F, I y O pueden ser todos muy diferentes. Por ejemplo, un compilador puede ejecutarse en una máquina y producir código objeto para otra máquina. Dicho compilador se denomina a menudo un *compilador cruzado.*

Supóngase que se escribe un compilador cruzado para un nuevo lenguaje L en el lenguaje de implantación S para que genere código para la máquina N; es decir, se crea L_SN. Si un compilador ya existente para S se ejecuta en la máquina M y genera código para M, se caracteriza por S_MM. Si L_SN se ejecuta a través de S_MM, se obtiene un compilador L_MN, es decir, un compilador de L a N que se ejecuta en M. Este proceso se ilustra en la figura 11.1 juntando los diagramas T para estos compiladores.

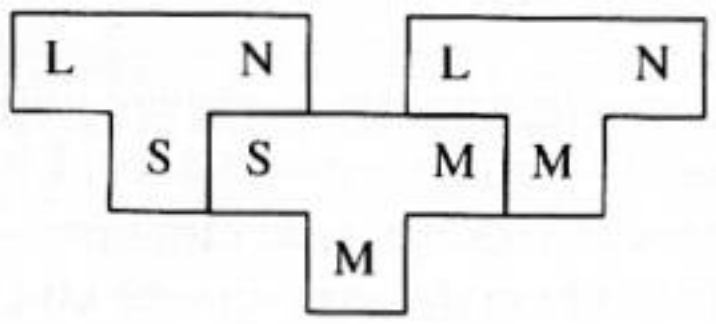


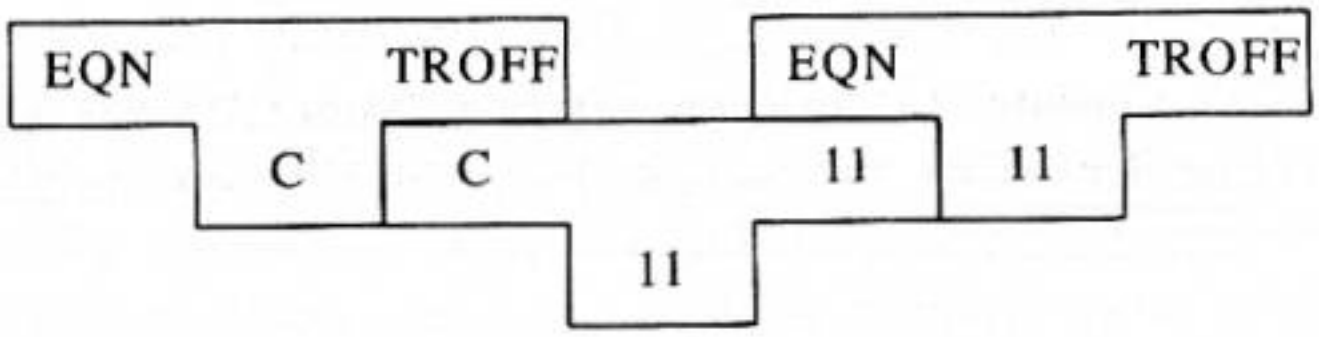
Fig. 11.1. Compilación de un compilador.

Cuando se juntan los diagramas T como en la figura 11.1, obsérvese que el lenguaje de implantación S del compilador L_SN debe ser el mismo que el lenguaje fuente del compilador existente S_MM y que el lenguaje objeto M del compilador existente debe ser el mismo que el lenguaje de implantación de la forma traducida L_MN. Un trío de diagramas T como el de la figura 11.1 se puede considerar como una ecuación

$$L_SN + S_MM = L_MN$$

Ejemplo 11.1. La primera versión del compilador de EQN (véase Sec. 12.1) tuvo a C como lenguaje de implantación y generó órdenes para el formador de textos TROFF. Como se muestra en el siguiente diagrama, se obtuvo un compilador cru-

zado para EQN, ejecutándose en un PDP-11, ejecutando EQN$_C$TROFF en el compilador de C, C1111, en el PDP-11.

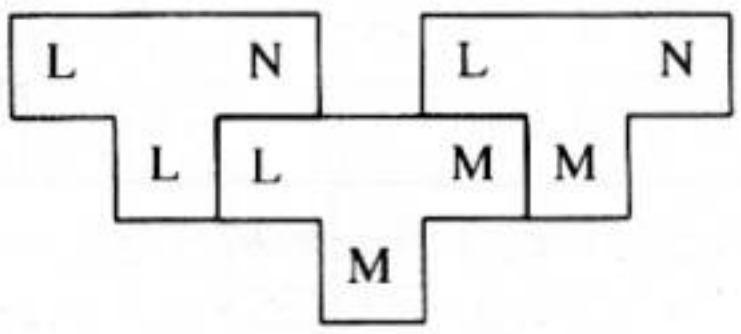


□

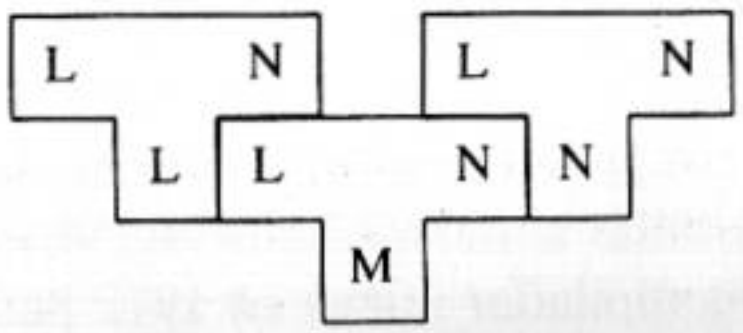
Una forma de arranque construye un compilador para subconjuntos cada vez mayores de un lenguaje. Supóngase que va a aplicarse un nuevo lenguaje L en la máquina M. Como primer paso se puede escribir un pequeño compilador que traduzca un subconjunto S de L a código objeto para M; es decir, un compilador $S_M M$. Después se utiliza el subconjunto S para escribir un compilador $L_S M$ para L. Cuando $L_S M$ se ejecuta a través de $S_M M$, se obtiene una implantación de L: $L_M M$. ELIAC fue uno de los primeros lenguajes implantados en su propio lenguaje (Huskey, Halstead y McArthur [1960]).

Wirth [1971] señala que Pascal se implantó por primera vez escribiendo un compilador en Pascal mismo. El compilador se tradujo después "a mano" a un lenguaje de bajo nivel disponible sin ningún intento de optimación. El compilador era para un subconjunto "(>60 por ciento)" de Pascal; después de varias etapas de arranque, se obtuvo un compilador para la totalidad de Pascal. Lecarme y Peyrolle-Thomas [1978] resumen los métodos utilizados para arrancar compiladores de Pascal.

Para aprovechar mejor las ventajas del arranque, un compilador tiene que escribirse en el lenguaje que compila. Supóngase que se escribe un compilador $L_L N$ para el lenguaje L en L para generar código para una máquina N. El desarrollo tiene lugar en una máquina M, donde un compilador existente $L_M M$ para L se ejecuta y genera código para M. Si se compila primero $L_L N$ con $L_M M$ se obtiene un compilador cruzado $L_M N$ que se ejecuta en M pero produce código para N:



El compilador $L_L N$ se puede compilar por segunda vez, utilizando esta vez el compilador cruzado generado:



El resultado de la segunda compilación es un compilador $L_N N$ que se ejecuta en N y genera código para N. Hay varias aplicaciones útiles de este proceso de dos pasos, de modo que se escribirá como en la figura 11.2.

Ejemplo 11.2. Este ejemplo está motivado por el desarrollo del compilador de FORTRAN H (véase Sec. 12.4). "El compilador mismo fue escrito en FORTRAN y arrancado tres veces. La primera vez fue para pasar de ejecutarse en la IBM 7094 a System/360 —un procedimiento arduo. La segunda vez fue para optimarse a sí mismo, lo cual redujo el tamaño del compilador de aproximadamente 550K a aproximadamente 400K bytes" (Lowry y Medlock [1969]).

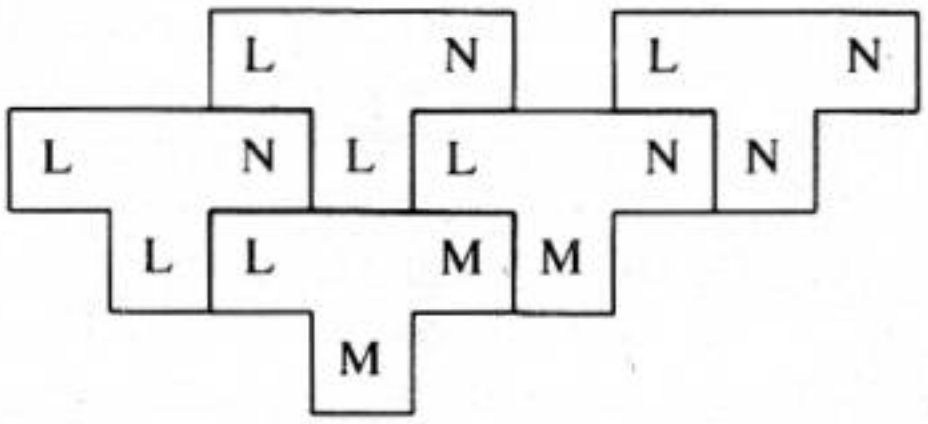
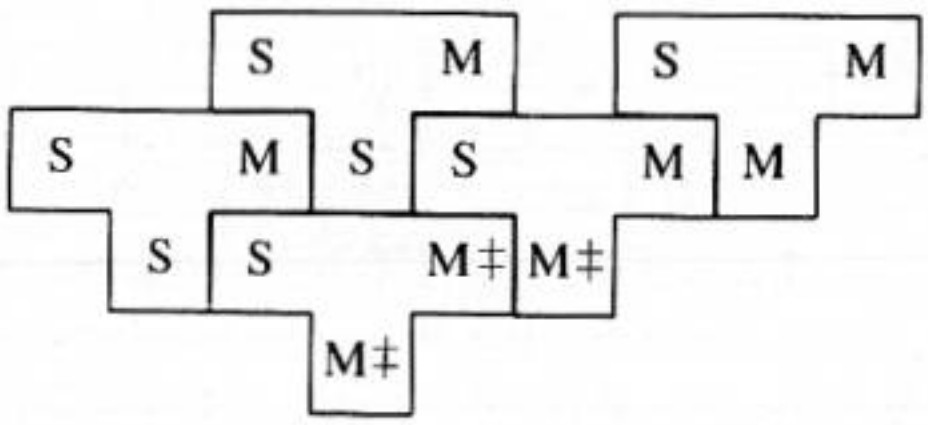


Fig. 11.2. Arranque de un compilador.

Utilizando las técnicas de arranque, un compilador optimador se puede optimar a sí mismo. Supóngase que todo el desarrollo se hace en la máquina M. Se tiene $S_S M$, un buen compilador optimador para un lenguaje S escrito en S, y se quiere $S_S M$, un buen compilador optimador para S escrito en M.

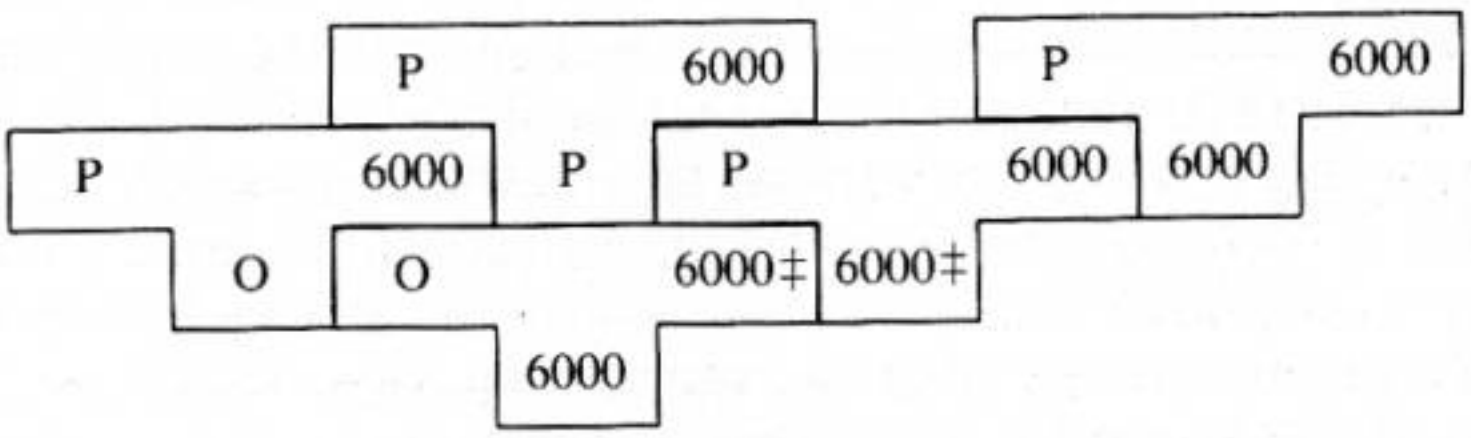
Se puede crear $S_M \ddagger M \ddagger$, un compilador "rápido y sucio" para S en M que no sólo genera código de mala calidad sino que también emplea mucho tiempo en hacerlo. ($M\ddagger$ indica una mala implantación en M. $S_M \ddagger M \ddagger$ es una mala implantación de un compilador que genera código malo.) Sin embargo, se puede utilizar el compilador indiferente $S_M \ddagger M \ddagger$ para obtener un buen compilador para S en dos pasos:



Primero, el compilador optimador $S_S M$ es traducido por el compilador rápido y sucio para producir $S_M \ddagger M$, una mala implantación del compilador optimador, pero que produce buen código. El compilador optimador bueno $S_M M$ se obtiene recompilando $S_S M$ a través de $S_M \ddagger M$. □

Ejemplo 11.3. Ammann [1981] describe cómo se obtuvo una implantación limpia de Pascal por un proceso similar al del ejemplo 11.2. Las revisiones de Pascal produjeron la escritura de un compilador nuevo en 1972 para las máquinas de la serie

CDC 6000. En el siguiente diagrama, O representa el lenguaje Pascal "antiguo" y P el lenguaje revisado.



Se escribió un compilador para Pascal revisado en Pascal antiguo y se tradujo a $P_{6000}‡6000$. Como en el ejemplo 11.2, el símbolo ‡ marca una fuente de ineficacia. El compilador antiguo no generaba código lo suficientemente eficiente. "Por tanto, la velocidad del compilador de $[P_{6000}‡6000]$ era bastante moderada y sus requisitos de memoria muy elevados (Ammann [1981])". Las revisiones de Pascal fueron lo suficientemente pequeñas para que el compilador P_O6000 pudiera traducirse a mano con poco esfuerzo a Pp6000 y ejecutarse a través del compilador ineficiente $P_{6000}‡6000$ para obtener una implantación limpia. □

11.3 EL ENTORNO PARA DESARROLLO DE COMPILADORES

En realidad, un compilador sólo es un programa. El entorno en que se desarrolle este programa puede afectar la velocidad y la fiabilidad de la implantación del compilador. El lenguaje en el que se implante el compilador es igualmente importante. Aunque se han escrito compiladores en lenguajes como FORTRAN, la mayoría de las personas que escriben compiladores elegirán un lenguaje orientado a sistemas como C.

Si el lenguaje fuente mismo es un nuevo lenguaje orientado a sistemas, entonces es razonable escribir el lenguaje en su propio lenguaje. Utilizando las técnicas de arranque estudiadas en la sección anterior, compilar el compilador ayuda a depurarlo.

Las herramientas para la construcción de *software* del entorno de programación facilitan la creación de un compilador eficiente. Al escribir un compilador, generalmente se particiona todo el programa en módulos, donde cada módulo se puede procesar de muchas maneras. Un programa que lleve a cabo el procesamiento de estos módulos es una ayuda indispensable para quien escribe el compilador. El sistema UNIX contiene una orden llamada *make* (Feldman [1979a]) que administra y mantiene los módulos que constituyen un programa de computador; *make* conoce las relaciones entre los módulos del programa y emite sólo las órdenes necesarias para mantener consistentes los módulos después de las modificaciones.

Ejemplo 11.4. La orden *make* lee la especificación de las funciones que hay que realizar en un archivo llamado `makefile`. En la sección 2.9 se construyó un traductor compilando siete archivos con un compilador de C, dependiendo cada uno de ellos de un archivo encabezador global, `global.h`. Para mostrar cómo *make* puede rea-

lizar la tarea de unir el compilador, supóngase que al compilador resultante se le llama `trad`. La especificación `makefile` puede aparecer como:

```
OBJS = analizléx.o analizsint.o emisor.o símbolos.o\
       inic.o error.o principal.o

trad:  $(OBJS)
       cc $(OBJS) -o trad

analizléx.o analizsint.o emisor.o símbolos.o\
       inic.o error.o principal.o: global.h
```

El signo igual en la primera línea hace que `OBJS` del lado izquierdo represente los siete archivos objeto de la derecha. (Las líneas largas se pueden dividir colocando una diagonal invertida al final de la parte continua.) Los dos puntos de la segunda línea indican que `trad` a su izquierda depende de todos los archivos de `OBJS`. Dicha línea de dependencia puede ir seguida de una orden para construir (*make*) el archivo que está a la izquierda de los dos puntos. Por tanto, la tercera línea indica que el programa objeto `trad` se crea enlazando los archivos objeto `analizléx.o`, `analizsint.o`, ..., `principal.o`. Sin embargo, *make* sabe que debe crear primero los archivos objeto; lo hace automáticamente buscando los archivos fuente correspondientes `analizléx.o`, `analizsint.o`, ... `principal.o`, y compilando cada uno con el compilador de C para crear los archivos objeto correspondientes. La última línea de `makefile` dice que todos los siete archivos objeto dependen del archivo de encabezamiento global, `global.h`.

El traductor se crea con sólo teclear la orden *make*, que hace que se emitan las siguientes órdenes:

```
cc -c analizléx.c
cc -c analizsin.c
cc -c emisor.c
cc -c símbolos.c
cc -c inic.c
cc -c error.c
cc -c principal.c
cc analizléx.o analizsint.o emisor.o símbolos.o\
       inic.o error.o principal.o -o trad
```

Posteriormente, se volverá a hacer una compilación sólo si se modifica un archivo fuente dependiente después de la última compilación. La referencia Kernighan y Pike [1984] contiene ejemplos del uso de *make* para facilitar la construcción de un compilador. □

Un perfilador es otra herramienta útil para la construcción de compiladores. Una vez escrito el compilador, se puede utilizar un perfilador para determinar dónde emplea el compilador su tiempo cuando compila un programa fuente. La identificación y modificación de los puntos muy transitados del compilador puede acelerar el compilador en un factor de dos o tres.

Además de las herramientas para el desarrollo de *software*, se han desarrollado varias herramientas específicamente para el proceso de desarrollo de compiladores. En la sección 3.5 se describió el generador LEX que puede utilizarse para producir automáticamente un analizador léxico a partir de una especificación por medio de expresiones regulares de un analizador léxico; en la sección 4.9 se describió el generador YACC que puede utilizarse para producir automáticamente un analizador sintáctico LR a partir de una descripción gramatical de la sintaxis del lenguaje. La orden *make* anteriormente descrita invocará automáticamente a LEX y YACC cuando los necesite. Además de los generadores léxicos y sintácticos, se han creado generadores de gramáticas con atributos y generadores de generadores de código para facilitar la construcción de los componentes de un compilador. Muchas de estas herramientas para la construcción de compiladores tienen la ventaja de que atraparán los errores de la especificación del compilador.

Ha habido discusiones sobre la eficiencia y conveniencia de los generadores de programas en la construcción de compiladores (Waite y Carter [1985]). El hecho comprobado es que los generadores de programas bien implantados son una ayuda significativa en la producción de componentes fiables de un compilador. Es mucho más fácil producir un analizador sintáctico correcto utilizando una descripción gramatical del lenguaje y un generador de analizadores sintácticos que implantar un analizador sintáctico directamente a mano. Sin embargo, un aspecto importante es cómo estos generadores sirven de interfaz respecto uno del otro y respecto de otros programas. Un error común en el diseño de un generador es suponer que es el centro del diseño. En un diseño mejor el generador produce subrutinas con interfaces limpias que puedan ser llamadas por otros programas (Johnson y Lesk[1978]).

11.4 PRUEBAS Y MANTENIMIENTO

Un compilador debe generar código correcto. Idealmente, se desearía tener un computador que comprobara mecánicamente que un compilador implanta fielmente su especificación. Varios artículos estudian la corrección de algunos algoritmos de compilación pero desgraciadamente, los compiladores casi nunca se especifican de forma que una implantación arbitraria se pueda comprobar mecánicamente en vez de una especificación formal. Como los compiladores son funciones bastante complejas, también hay que comprobar que la especificación misma sea correcta.

En la práctica, se debe recurrir a un método sistemático de comprobar el compilador para incrementar la confianza de que funcionará satisfactoriamente en el campo. Un enfoque que han utilizado con éxito muchos escritores de compiladores es la prueba de "regresión". Se mantiene un conjunto de programas de prueba y siempre que se modifique el compilador, los programas de prueba se compilan utilizando tanto la versión nueva como la antigua del compilador. Al escritor del compilador se le informa de las diferencias en los programas objeto producidos por los dos compiladores. También se puede utilizar la orden del sistema UNIX *make* para automatizar las pruebas.

Elegir los programas que van a ser incluidos en el conjunto de prueba es un problema difícil. Como meta, sería deseable que los programas ejercitaran todas las pro-

posiciones del compilador por lo menos una vez. Hace falta mucho ingenio para encontrar un conjunto de prueba así. Se han construido conjuntos de pruebas exhaustivas para varios lenguajes. (FORTRAN, TEX, C, etc.). Muchos escritores de compiladores añaden a las pruebas de regresión programas que han descubierto errores en anteriores versiones de su compilador; es frustrante que un error antiguo reaparezca debido a una nueva corrección.

Comprobar el rendimiento también es importante. Algunos escritores de compiladores se aseguran de que las nuevas versiones del compilador generen código que sea casi tan bueno como la versión previa realizando estudios de tiempos como parte de la prueba de regresión.

El mantenimiento de un compilador es otro problema a tener en cuenta, sobre todo si el compilador va a ejecutarse en distintos entornos o si cambia la gente involucrada en el proyecto del compilador. Un elemento crucial para poder mantener a un compilador es un buen estilo de programación y una buena documentación. Los autores conocen un compilador que fue escrito utilizando tan sólo siete comentarios, uno de los cuales decía "Este código está maldito". No es necesario decir que es bastante difícil que nadie, excepto quizá el escritor, pueda mantener este programa.

Knuth [1984b] desarrolló un sistema llamado WEB que apunta el problema de documentar grandes programas escritos en Pascal. WEB facilita la documentación de programas; la documentación se desarrolla al mismo tiempo que el código, no después. Muchas de las ideas de WEB se pueden aplicar asimismo a otros lenguajes.

CAPITULO 12

Una mirada a algunos compiladores

Este capítulo analiza la estructura de algunos compiladores existentes para un lenguaje de formación de textos, Pascal, FORTRAN, BLISS y MODULA 2. El propósito no es favorecer los diseños aquí representados y excluir otros sino ilustrar la variedad posible en la implantación de un compilador.

Los compiladores para Pascal se eligieron porque influyeron en el diseño del lenguaje mismo. Los compiladores para C se eligieron porque C es el lenguaje principal del sistema operativo UNIX. El compilador de FORTRAN H se eligió porque tiene una influencia significativa sobre el desarrollo de técnicas de optimación. BLISS/11 se eligió para ilustrar el diseño de un compilador cuyo objetivo es optimizar espacio. El compilador MODULA 2 de DEC se eligió porque utiliza téenicas relativamente sencillas para producir código excelente y fue escrito por una persona en pocos meses.

12.1 EQN, UN PREPROCESADOR PARA TIPOGRAFIA DE MATEMATICAS

El conjunto de entradas posibles para varios programas de computadora se puede considerar como un pequeño lenguaje. La estructura del conjunto se puede escribir por medio de una gramática y se puede utilizar la traducción dirigida por la sintaxis para especificar de manera precisa lo que hace el programa. Entonces la tecnología de compiladores puede servir para implantar el programa.

Uno de los primeros compiladores para lenguajes pequeños en el entorno de programación UNIX fue EQN de Kernighan y Cherry [1975]. Como se describe brevemente en la sección 1.2, EQN toma una entrada como "`E sub 1`" y genera órdenes para el formador de textos TROFF para producir una salida de la forma "E_1".

La implantación de EQN se esboza en la figura 12.1. El preprocesamiento de macros (véase Sec. 1.4) y el análisis léxico se realizan a la vez. La cadena de componentes léxicos después del análisis léxico se traduce durante el análisis sintáctico a órdenes para la formación de textos. El traductor se construye utilizando el generador de analizadores sintácticos YACC, descrito en la sección 4.9.

Considerar la entrada para EQN como un lenguaje y aplicar tecnología de compiladores para construir un traductor tiene varias ventajas señaladas por los autores.

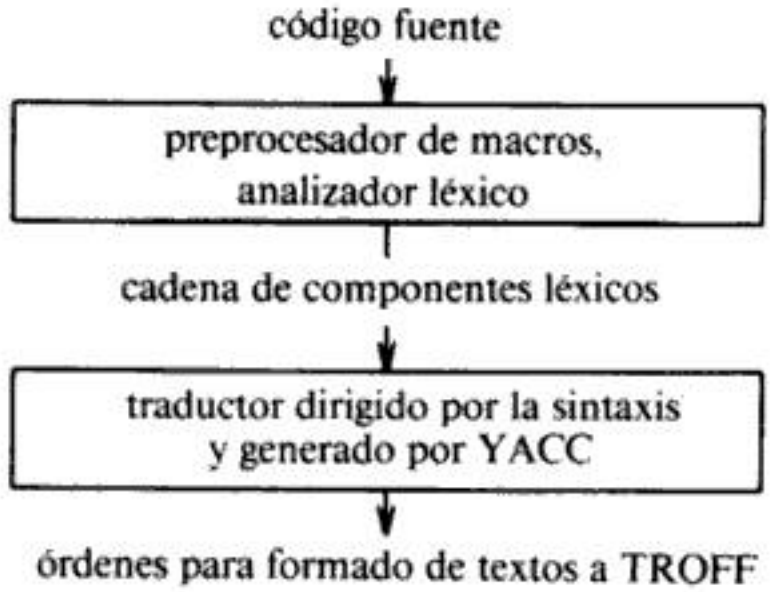


Fig. 12.1. Implantación de EQN.

1. *Facilidad de implantación.* "La construcción de un sistema que funcione, suficiente para probar ejemplos significativos exigió tal vez una persona-mes".
2. *Evolución del lenguaje.* Una definición dirigida por la sintaxis facilita cambios en el lenguaje de entrada. A través de los años EQN ha evolucionado en respuesta a las necesidades de los usuarios.

Los autores concluyen observando que "definir un lenguaje y construir un compilador para él utilizando un compilador de compiladores parece la única forma sensata de hacer las cosas".

12.2 COMPILADORES PARA PASCAL

El diseño de Pascal y el desarrollo del primer compilador para él "fueron independientes", según comenta Wirth [1971]. Por tanto, conviene examinar la estructura de los compiladores para el lenguaje escrito por Wirth y sus colegas. El primero (Wirth [1971]) y el segundo compilador (Ammann [1981, 1977]) generaban código de máquina absoluto para las máquinas de la serie CDC 6000. Los experimentos sobre transportabilidad con el segundo compilador llevaron al compilador Pascal-P que genera código, llamado código P, para una máquina de pila abstracta (Nori y colaboradores [1981]).

Cada uno de los compiladores anteriores es un compilador de una pasada organizado alrededor de un analizador sintáctico descendente recursivo, como la etapa inicial "bebé" del capítulo 2. Wirth [1971] observa que "resultó relativamente fácil moldear el lenguaje según [las limitaciones del método de análisis sintáctico]". En la figura 12.2 se muestra la organización del compilador Pascal-P.

Las operaciones básicas de la máquina de pila abstracta utilizada por el compilador Pascal-P refleja las necesidades de Pascal. La memoria para la máquina se organiza en cuatro áreas:

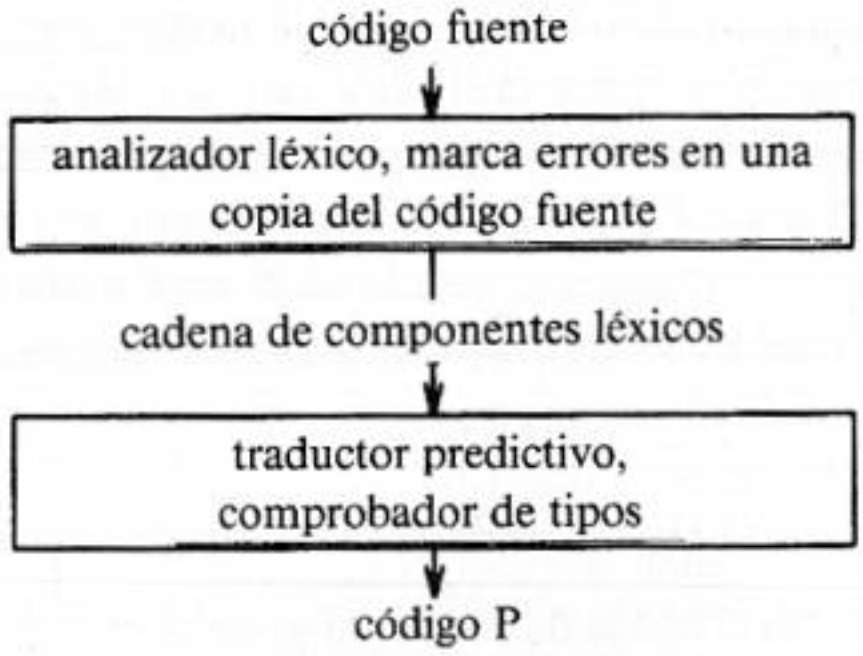


Fig. 12.2. Compilador de Pascal P.

1. código para los procedimientos,
2. constantes,
3. una pila para los registros de activación, y
4. un montículo para los datos asignados aplicando el operador **new**[1].

Como los procedimientos se pueden anidar en Pascal, el registro de activación para un procedimiento contiene enlaces de acceso y de control. Una llamada a procedimiento se traduce a una instrucción de "marcar la pila" para la máquina abstracta, con los enlaces de acceso y de control como parámetros. El código para un procedimiento se refiere a la memoria para un nombre local utilizando un desplazamiento desde el final de un registro de activación. Se hace referencia a la memoria para nombres no locales mediante un par, que consta del número de enlaces de acceso que van a ser recorridos y un desplazamiento, como en la sección 7.4. El primer compilador usó un *display* para acceder eficientemente a los nombres no locales.

Ammann [1981] saca las siguientes conclusiones de la experiencia al escribir el segundo compilador. Por una parte, el compilador de una pasada era fácil de implantar y generaba poca actividad de entrada y salida (el código para el cuerpo de un procedimiento se compila en memoria y se escribe como una unidad a memoria secundaria). Por otra parte, la organización de una pasada "impone severas limitaciones a la calidad del código generado y soporta requisitos de memoria" relativamente altos.

12.3 LOS COMPILADORES PARA C

C es un lenguaje de programación de propósito general diseñado por D. M. Ritchie y se utiliza como el principal lenguaje de programación en el sistema operativo UNIX (Ritchie y Thompson [1974]). UNIX mismo está escrito en C y ha sido trasladado a varias máquinas, desde microprocesadores hasta macrocomputadores, trasladando primero un compilador de C. Esta sección describe brevemente la estructura global

[1] El arranque se facilita porque el compilador, escrito en el subconjunto que compila, utiliza el montículo como pila, así que al principio se puede utilizar un sencillo administrador del montículo.

del compilador para el computador PDP-11 por Ritchie [1979] y la familia PCC de compiladores transportables por Johnson [1979]. Tres cuartos del código en PCC es independiente de la máquina objeto. Todos estos compiladores son básicamente de dos pasadas; el compilador para PDP-11 tiene una tercera pasada opcional que realiza optimación sobre la salida en lenguaje ensamblador, como se indica en la figura 12.3. Esta fase de optimación local elimina las proposiciones redundantes o inaccesibles.

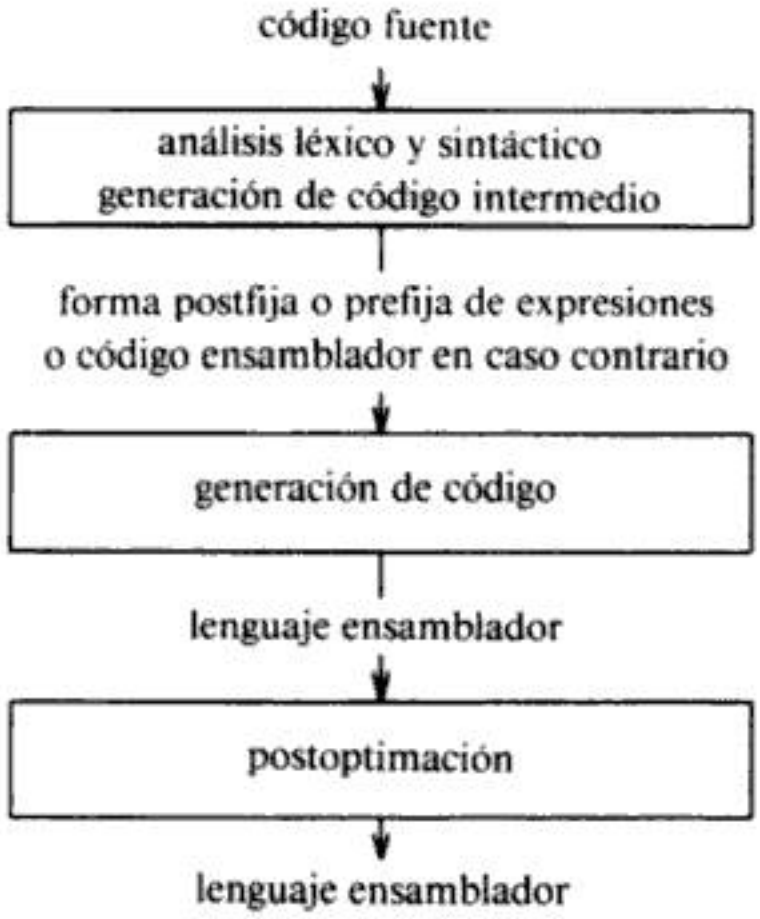


Fig. 12.3. Estructura de pasadas de los compiladores de C.

La pasada I de cada compilador realiza el análisis lexicográfico, el análisis sintáctico y la generación de código intermedio. El compilador para PDP-11 utiliza descenso recursivo para analizar sintácticamente todo excepto las expresiones, para las que se utiliza precedencia de operadores. El código intermedio consta de notación postfija para expresiones y código ensamblador para proposiciones de flujo de control. PCC utiliza un analizador sintáctico LALR(1) generado por YACC. Su código intermedio consta de notación prefija para expresiones y código ensamblador para otras construcciones. En cada caso, la asignación de memoria para los nombres locales se realiza durante la primera pasada, así que se puede hacer referencia a los nombres utilizando desplazamientos dentro de un registro de activación.

Dentro de la etapa final, las expresiones se representan mediante árboles sintácticos. En el compilador para PDP-11, la generación de código se implantó mediante un recorrido de árboles, utilizando una estrategia similar a la del algoritmo de etiquetado de la sección 9.10. Se han hecho modificaciones a ese algoritmo para garantizar que estén disponibles pares de registros para operaciones que los necesiten y para aprovechar operandos que sean constantes.

Johnson [1978] reconsidera la influencia de la teoría en PCC. En PCC y en PCC2, una versión posterior del compilador, se genera código para las expresiones reescribiendo los árboles. El generador de código en PCC examina de una en una las proposiciones del lenguaje fuente, encontrando repetidamente subárboles máximos que se pueden calcular sin almacenamientos, utilizando los registros disponibles. Las eti-

quetas calculadas como en la sección 9.10 identifican las subexpresiones que van a ser calculadas y almacenadas en memoria. El compilador genera el código para evaluar y almacenar los valores representados por estos subárboles a medida que se seleccionan los subárboles. La reescritura es más evidente en PCC2, cuyo generador de código se basa en el algoritmo de programación dinámica de la sección 9.11.

Johnson y Ritchie [1981] describen la influencia de la máquina objeto en el diseño de los registros de activación y las secuencias de llamada y retorno de procedimientos. La función de biblioteca **printf** puede tener un número variable de argumentos, así que el diseño de la secuencia de llamada en algunas máquinas se ve dominado por la necesidad de permitir listas de argumentos de longitud variable.

12.4 LOS COMPILADORES DE FORTRAN H

El compilador original de FORTRAN H escrito por Lowry y Medlock [1969] era un compilador optimador extenso y bastante poderoso construido con métodos que preceden a los descritos en este libro. Se han realizado varios intentos de mejorar el rendimiento; se desarrolló una versión "ampliada" del compilador para el IBM/370, y una versión "mejorada" fue desarrollada por Scarborough y Kolsky [1980]. FORTRAN H ofrece al usuario la posibilidad de elegir entre no utilizar optimación, utilizar sólo optimación de registros o utilizar optimación completa. En la figura 12.4 aparece un esbozo del compilador en caso de que se realice optimación completa.

El texto fuente se considera en cuatro pasadas. Las primeras dos realizan el análisis léxico y sintáctico, produciendo cuádruplos. La siguiente pasada incorpora optimación de código y optimación de registros y la pasada final genera código objeto a partir de los cuádruplos y las asignaciones a los registros.

La fase de análisis léxico es poco frecuente, puesto que su salida no es una cadena de componentes léxicos sino una cadena de "pares operador-operando", que son más o menos equivalentes a un componente léxico operando junto con el componente léxico no operando precedente. Se debe observar que en FORTRAN, como en la mayoría de los lenguajes, nunca hay dos componentes léxicos operandos consecutivos como identificadores o constantes; más bien, dos componentes léxicos de ese tipo siempre están separados al menos por un componente léxico de puntuación.

Por ejemplo, la proposición de asignación

```
A = B(I) + C
```

se traduciría a la secuencia de pares:

"proposición de asignación"	A
=	B
(s	I
)	-
+	C

La fase de análisis léxico distingue entre un paréntesis izquierdo cuya misión es introducir una lista de parámetros o subíndices de uno cuya misión es agrupar ope-

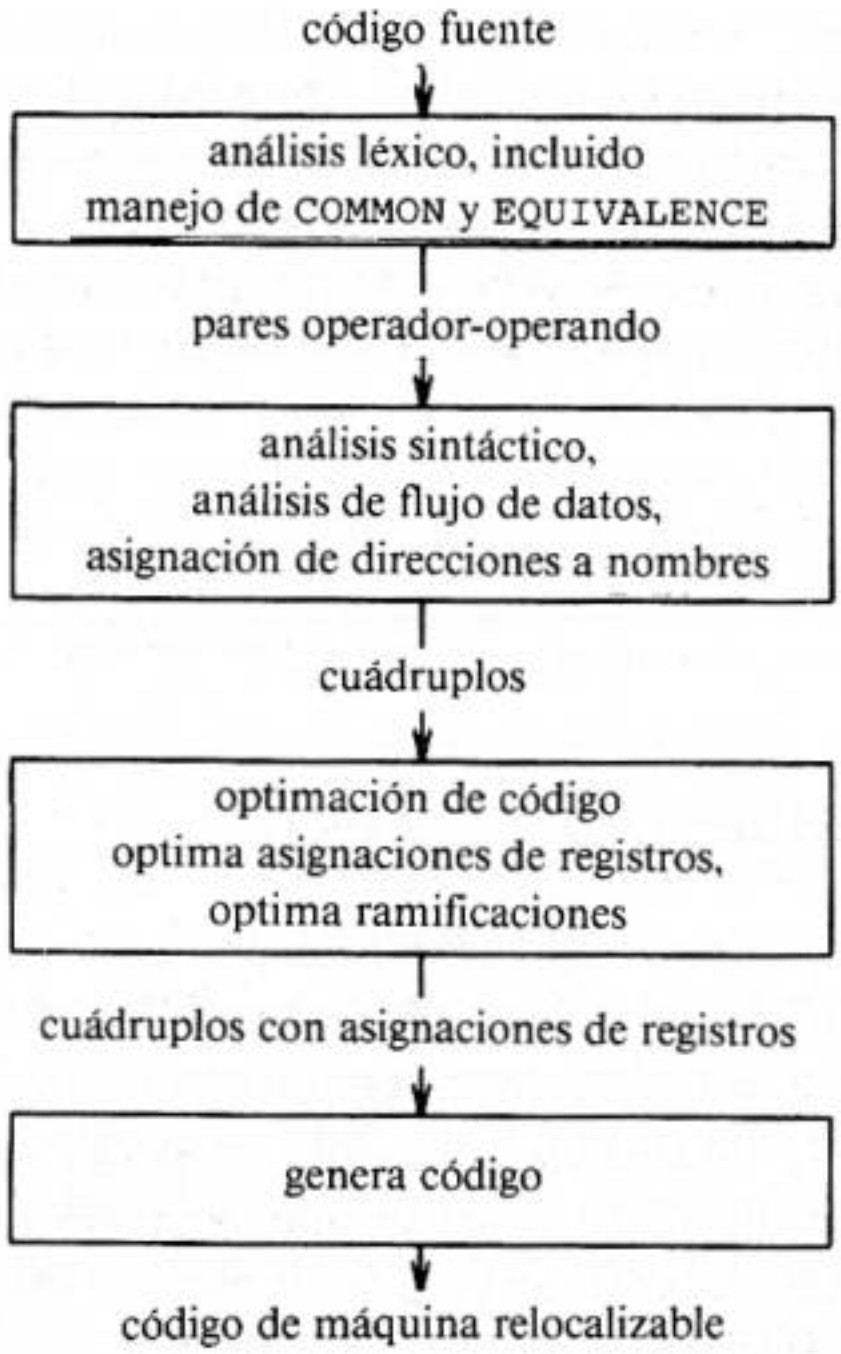


Fig. 12.4. Esbozo del compilador FORTRAN H.

randos. Por tanto, el símbolo "(s" sirve para representar un paréntesis izquierdo utilizado como operador de subíndice. Los paréntesis derechos nunca tienen un operando a continuación y FORTRAN H no distingue las dos funciones de los paréntesis derechos.

Asociado con el análisis léxico está el proceso de proposiciones COMMON y EQUIVALENCE. En esta etapa es posible localizar cada bloque de memoria COMMON, así como los bloques de memoria asociados con las subrutinas, y determinar la localización de cada variable mencionada por el programa en una de estas áreas de memoria estáticas.

Ya que FORTRAN no tiene proposiciones de control estructuradas como las proposiciones **while**, el análisis sintáctico, excepto para las expresiones, es bastante directo y FORTRAN H utiliza simplemente un analizador sintáctico por precedencia de operadores para expresiones. Algunas optimaciones locales muy simples se realizan durante la generación de cuádruplos; por ejemplo, las multiplicaciones por potencias de dos se sustituyen por operaciones de desplazamientos hacia la izquierda.

Optimación de código en FORTRAN H

Cada subrutina se particiona en bloques básicos y la estructura de los lazos se deduce encontrando aristas de grafo de flujo cuyas cabezas dominen a sus colas, como se describe en la sección 10.4. El compilador realiza las siguientes optimaciones.

1. *Eliminación de subexpresiones comunes.* El compilador busca subexpresiones comunes locales y expresiones comunes a un bloque B y uno o más bloques a los que B domine. Otros casos de subexpresiones comunes no son detectados. Además, la detección de subexpresiones comunes se realiza expresión a expresión, en vez de utilizar el método de vectores de bits descrito en la sección 10.6. Al desarrollar la versión "mejorada" del compilador, los autores vieron que era posible un mayor aumento de velocidad utilizando métodos de vectores de bits.
2. *Traslado de código.* Las proposiciones invariantes de los lazos se eliminan de ellos básicamente como se describe en la sección 10.7.
3. *Propagación de copias.* De nuevo, se realiza no más de una proposición de copia a la vez.
4. *Eliminación de variables de inducción.* Esta optimación se realiza sólo para variables asignadas una vez dentro del lazo. En lugar de utilizar el enfoque de "familia" descrito en la sección 10.7, se hacen múltiples pasadas a través del código para detectar las variables de inducción que pertenecen a la familia de alguna otra variable de inducción.

Aunque el análisis de flujo de datos se realiza al estilo de uno a uno, se almacenan como vectores de bits los valores correspondientes a lo que se ha llamado *ent* y *sal*. Sin embargo, en el compilador original se impuso un límite de longitud 127 a dichos vectores, así que los programas grandes sólo tienen optimación para sus variables más utilizadas. La versión mejorada incrementa el límite pero no lo elimina.

Optimaciones algebraicas

Como FORTRAN se utiliza a menudo para cálculos numéricos, la optimación algebraica es peligrosa, puesto que las transformaciones de expresiones pueden, en la aritmética del computador, introducir desbordamientos o pérdidas de precisión que no son visibles si se tiene una visión idealizada de la simplificación algebraica. Sin embargo, las transformaciones algebraicas en las que toman parte los enteros son generalmente seguras y la versión mejorada del compilador realiza parte de esta optimación únicamente en caso de las referencias a matrices.

En general, una referencia a matriz como `A(I,J,K)` implica un cálculo de desplazamiento en el que se calcula una expresión de la forma $a\mathtt{I} + b\mathtt{J} + c\mathtt{K} + d$; los valores exactos de las constantes dependen de la posición de `A` y de las dimensiones de la matriz. Si, por ejemplo, `I` y `K` fueran constantes, ya sean constantes numéricas o variables invariantes de un lazo, entonces el compilador aplica la ley asociativa y conmutativa para obtener una expresión $b\mathtt{J} + e$, donde $e = a\mathtt{I} + c\mathtt{K} + d$.

Optimación de registros

FORTRAN H divide los registros en tres clases. Estos conjuntos de registros se utilizan para optimación de registros locales, optimación de registros globales y "optimación de ramificaciones". El número exacto de registros en cada clase puede ser ajustado por el compilador, dentro de unos límites.

Los registros globales se asignan sobre una base lazo a lazo a las variables con referencias más frecuentes en ese lazo. Una variable que califique a un registro en

un lazo L, pero no en el lazo que contenga inmediatamente a L, se carga en el preencabezamiento de L y se almacena en memoria a la salida de L.

Los registros locales se utilizan dentro de un bloque básico para guardar los resultados de una proposición hasta que se utiliza en una proposición o proposiciones posteriores. Sólo se almacena el valor de un temporal si no existen los suficientes registros locales. El compilador intenta calcular nuevos valores en el registro que contenga uno de sus operandos, si ese operando se desactiva posteriormente. En la versión mejorada, se intenta reconocer la situación donde los registros globales se pueden intercambiar con otros registros para incrementar el número de veces que puede tener lugar una operación en el registro que guarda uno de sus operandos.

La optimación de ramificaciones es un artefacto del conjunto de instrucciones del IBM/370, que asigna una gran importancia significativa a saltar sólo a posiciones que se pueden expresar como el contenido de algún registro, más una constante en el rango de 0 a 4095. Por tanto, FORTRAN H asigna algunos registros para guardar direcciones en el espacio de código, a intervalos de 4096 bytes, para permitir saltos eficientes en todos excepto en programas demasiado grandes.

12.5 EL COMPILADOR BLISS/11

Este compilador implanta el lenguaje de programación de sistemas BLISS en un PDP-11 (Wulf y colaboradores [1975]). En cierto sentido, es un compilador optimador de un mundo que ha dejado de existir, un mundo donde el espacio de memoria estaba tan solicitado que tenía sentido realizar optimaciones cuyo solo propósito era reducir espacio en lugar de tiempo. Sin embargo, la mayoría de las optimaciones realizadas por el compilador también ahorran tiempo y hoy en día se utilizan descendientes de este compilador.

El compilador debe ser considerado por varias razones. Su rendimiento en la optimación es fuerte, y realiza varias transformaciones que no se encuentran en casi ninguna otra parte. Además, encabezó el enfoque "dirigido por la sintaxis" para la optimación, como se estudió en la sección 10.5. Es decir, el lenguaje BLISS se diseñó para producir sólo grafos de flujo reducibles (no tiene instrucciones **goto**). Por tanto, fue posible realizar análisis de flujo de datos directamente sobre el árbol de análisis sintáctico, en lugar de hacerlo en un grafo de flujo.

El compilador trabaja en una sola pasada, con un procedimiento completamente procesado antes de leer el siguiente. Los diseñadores consideraron que el compilador estaba compuesto por cinco módulos, como se muestra en la figura 12.5.

LEXSYNFLO realiza análisis léxico y análisis sintáctico. Se utiliza un analizador sintáctico descendente recursivo. Como BLISS no permite proposiciones **goto**, todos los grafos de flujo de los procedimientos de BLISS son reducibles. De hecho, la sintaxis del lenguaje permite construir el grafo de flujo, y determinar lazos y entradas a lazos, a medida que se hace el análisis sintáctico. LEXSYNFLO lo hace así y también determina subexpresiones comunes y una variante de las cadenas de uso y definición y de definición y uso, aprovechando la estructura de los grafos de flujo reducibles. Otra tarea importante de LEXSYNFLO es detectar grupos de expresiones similares. Estas son candidatas a ser sustituidas por una sola subrutina. Obsér-

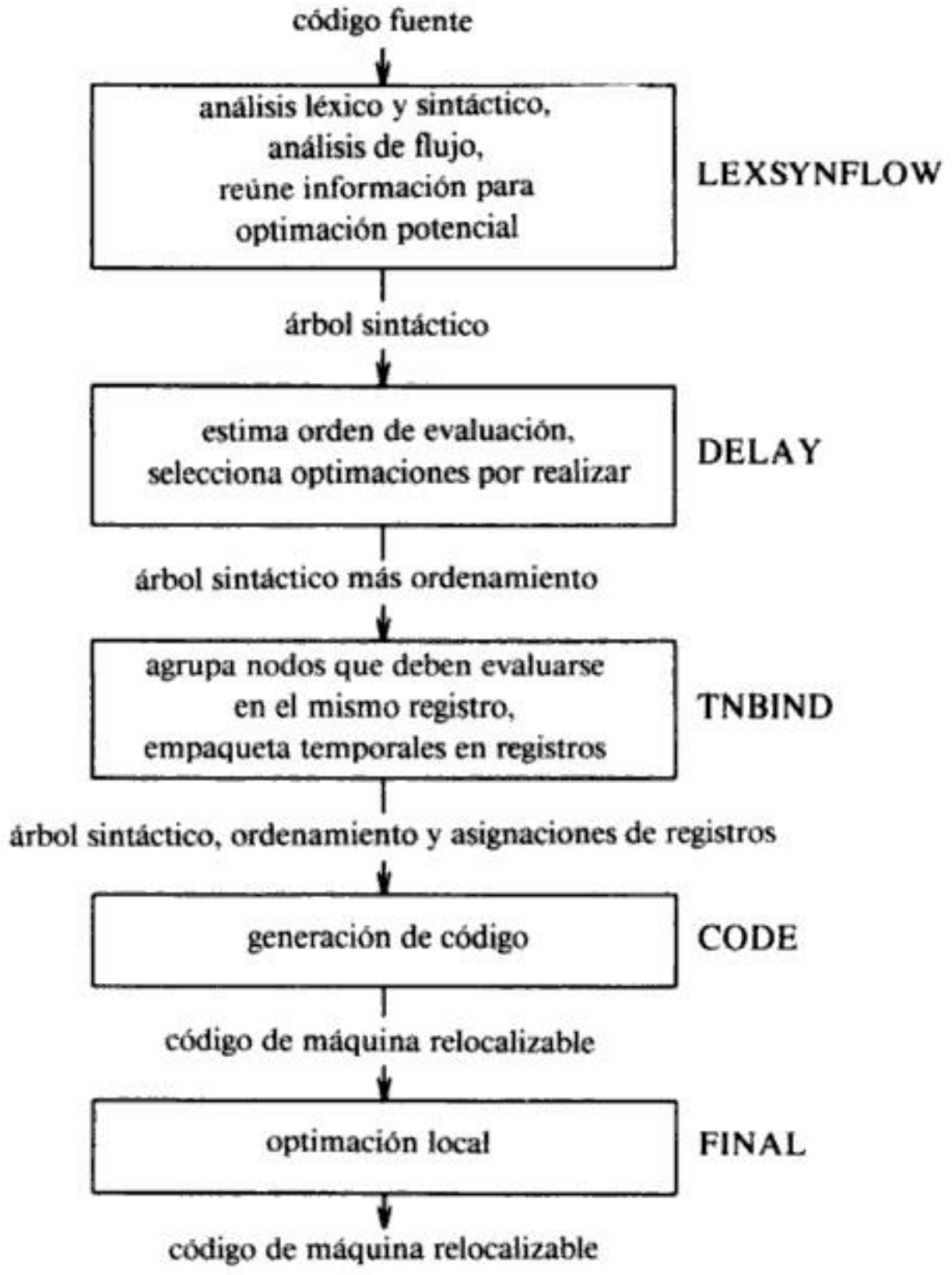


Fig. 12.5. El compilador BLISS/11.

vese que esta sustitución hace que el programa se ejecute más lentamente pero puede ahorrar espacio.

El módulo DELAY examina el árbol sintáctico para determinar qué ejemplos particulares de las optimaciones habituales, como el traslado de código invariante y la eliminación de subexpresiones comunes suponen probablemente una ventaja. El orden de evaluación de las expresiones se determina en este momento, basado en la estrategia de etiquetado de la sección 9.10, modificada para tener en cuenta registros que no estén disponibles porque se utilizan para preservar los valores de subexpresiones comunes. Las leyes algebraicas se utilizan para determinar si se debe hacer reordenamiento de los cálculos. Las expresiones condicionales se evalúan numéricamente o mediante flujo de control, como se estudió en la sección 8.4, y DELAY decide el modo más barato en cada caso.

TNBIND considera los nombres de temporales que deben enlazarse a registros. Se asignan tanto registros como localidades de memoria. La estrategia empleada

consiste en agrupar primero los nodos del árbol sintáctico a los que se debe asignar el mismo registro. Como se estudió en la sección 9.6, es mejor evaluar un nodo en el mismo registro que uno de sus padres. A continuación, la ventaja que se obtiene si se conserva un temporal en un registro se estima mediante un cálculo que favorece los que se utilizan varias veces en espacios cortos. Después los registros se asignan hasta que se utilizan, empaquetando los nodos más beneficiosos primero en los registros. CODE convierte el árbol, con su información sobre ordenamiento y asignación de registros en código de máquina relocalizable.

Este código es examinado repetidamente por FINAL que realiza optimación local hasta que no pueda obtener una mayor mejoría. Las mejoras realizadas incluyen la eliminación de saltos (condicionales o incondicionales) a saltos y complementación de condicionales, tal como se estudia en la sección 9.9.

Las instrucciones redundantes o inalcanzables se eliminan (podrían haber sido ocasionadas por otras optimaciones de FINAL). Se intenta la fusión de secuencias de código similares en los dos caminos de una ramificación, como en la propagación local de constantes. Se intentan otras optimaciones locales, algunas muy dependientes de la máquina. Una muy importante es la sustitución, donde sea posible, de instrucciones de salto por "ramificaciones" de PDP-11, que necesitan una palabra pero están limitadas en su rango a 128 palabras.

12.6 COMPILADOR OPTIMADOR DE MODULA-2

Este compilador, descrito en Powell [1984], fue desarrollado con la intención de producir buen código, utilizando optimaciones que proporcionen alto rendimiento con poco esfuerzo; el autor describe su estrategia como búsqueda de las optimaciones "mejores y más sencillas". Dicha filosofía puede ser difícil de seguir; sin experimentación ni medidas, es difícil decidir por anticipado cuáles son las optimaciones "mejores y más sencillas" y algunas de las decisiones tomadas en el compilador de MODULA-2 probablemente son inadecuadas para un compilador que proporcione optimación máxima. Sin embargo, la estrategia consiguió el propósito del autor de producir excelente código con un compilador que se escribió por una persona en pocos meses. En la figura 12.6 se esbozan las cinco pasadas de la etapa inicial del compilador.

El analizador sintáctico se generó utilizando YACC, y produce árboles sintácticos en dos pasadas, puesto que las variables de MODULA no tienen que declararse antes de ser utilizadas. Se intentó hacer este compilador compatible con los dispositivos existentes. El código intermedio es código P para ser compatible con muchos compiladores de Pascal. El formato de llamada a procedimiento para este compilador coincide con los compiladores de Pascal y C que se ejecutan bajo Berkeley UNIX, de modo que los procedimientos escritos en los tres lenguajes se pueden integrar fácilmente.

El compilador no realiza análisis de flujo de datos. En su lugar, MODULA-2, como BLISS, es un lenguaje que sólo puede producir grafos de flujo reducibles, así que aquí también se puede utilizar la metodología de la sección 10.5. De hecho, el compilador de MODULA va más lejos que el compilador de BLISS-11 en la forma

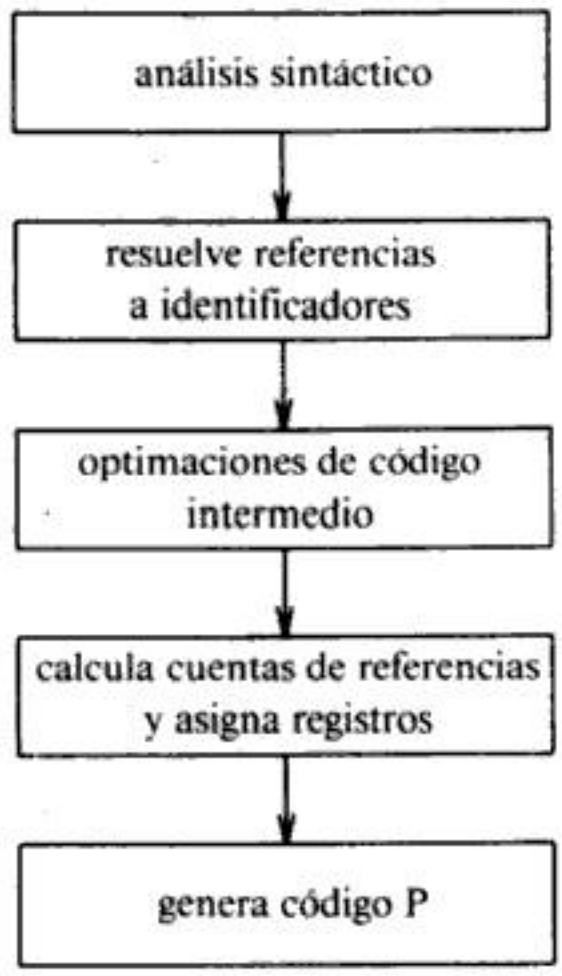


Fig. 12.6. Pasadas del compilador de MODULA-2.

en que se aprovecha de la sintaxis. Los lazos se identifican por su sintaxis; es decir, el compilador busca construcciones **while** y **for**. Las expresiones invariantes se detectan por el hecho de que ninguna de sus variables se define en el lazo y se trasladan a un encabezamiento del lazo. Las únicas variables de inducción que se detectan son las de la familia de un índice del lazo **for**. Las subexpresiones comunes globales se detectan cuando una está en un bloque que domine el bloque de la otra, pero este análisis se realiza expresión a expresión, en lugar de con vectores de bits.

La estrategia de asignación de registros se diseñó de manera similar para hacer cosas razonables sin ser exhaustivo. En concreto, considera como candidatos para la asignación de un registro sólo a:

1. temporales utilizados durante la evaluación de una expresión (éstas reciben prioridad absoluta),
2. valores de subexpresiones comunes,
3. valores de índice y límites en lazos **for**,
4. la dirección de E en una expresión de la forma `with` E `do`, y
5. variables simples (caracteres, enteros, etc.) locales al procedimiento en curso.

Se intenta estimar el valor de conservar cada variable en las clases (2) a (5) de un registro. Se supone que una proposición se ejecuta 10^d veces si está unida dentro de d lazos. Sin embargo, las variables referenciadas dos veces a lo sumo no se consideran elegibles; las otras se clasifican en orden de uso estimado y se asignan a un registro, si hay uno disponible, después de asignar los temporales de expresiones y las variables con alto rango.

APENDICE

Un proyecto de programación

A.1 INTRODUCCION

Este apéndice propone ejercicios de programación que se pueden utilizar en un laboratorio de programación junto con un curso de diseño de compiladores basado en este libro. Los ejercicios consisten en implantar los componentes básicos de un compilador para un subconjunto de Pascal. El subconjunto es mínimo, pero permite expresar programas como la clasificación recursiva de la sección 7.1. Ser un subconjunto de un lenguaje existente posee cierta utilidad. El significado de los programas en el subconjunto viene determinado por la semántica de Pascal (Jensen y Wirth [1975]). Si hay un compilador de Pascal disponible, se puede utilizar para comprobar el comportamiento del compilador escrito como ejercicio. Las construcciones en el subconjunto aparecen en la mayoría de los lenguajes de programación, de modo que los ejercicios correspondientes se pueden formular utilizando un lenguaje distinto si no hay un compilador de Pascal disponible.

A.2 ESTRUCTURA DEL PROGRAMA

Un programa consta de una secuencia de declaraciones de datos globales, una secuencia de declaraciones de procedimientos y funciones y una sola proposición compuesta que es el "programa principal". A los datos globales se les asigna memoria estática. A los datos locales a procedimientos y funciones se les asigna memoria en una pila. Se permite la recursividad y los parámetros se pasan por referencia. Se supone que el compilador proporciona los procedimientos `read` y `write`.

La figura A.1 da un ejemplo de un programa. El nombre del programa es `ejemplo`, e `input` y `output` son los nombres de los archivos utilizados por `read` y `write`, respectivamente.

A.3 SINTAXIS DE UN SUBCONJUNTO DE PASCAL

Más abajo se lista una gramática LALR(1) para un subconjunto de Pascal. La gramática puede modificarse para el análisis descendente recursivo eliminando la re-

```
program ejemplo(input, output);
var x, y: integer;
function mcd(a, b: integer): integer;
begin
    if b = 0 then mcd := a
    else mcd := mcd(b, a mod b)
end;

begin
    read(x, y);
    write(mcd(x, y))
end.
```

Fig. A.1. Programa de ejemplo.

cursividad por la izquierda, como se describe en las secciones 2.4 y 4.3. Se puede construir un analizador sintáctico por precedencia de operadores para las expresiones sustituyendo **oprel**, **opsuma** y **pmult**, y eliminando las producciones ϵ.

La suma de la producción

proposición → **if** *expresión* **then** *proposición*

introduce la ambigüedad del **else**, que se puede eliminar como se estudió en la sección 4.3 (véase también el Ejemplo 4.19 si se utiliza un analizador sintáctico predictivo).

No hay distinción sintáctica entre una variable simple y la llamada a una función sin parámetros. Ambas se generan por medio de la producción

factor → **id**

Por tanto, la asignación `a:=b` le asigna a `a` el valor devuelto por la función `b`, si `b` ha sido declarado como una función.

programa →
    programa id (*lista_identificadores*) ;
    declaraciones
    declaraciones_subprogramas
    proposición_compuesta
    .

lista_identificadores →
    id
    | *lista_identificadores* , **id**

declaraciones →
    declaraciones **var** *lista_identificadores* : *tipo* ;
    | ϵ

tipo →
    tipo_estándar
    | **array** [**núm** . . **núm**] **of** *tipo estándar*

tipo_estándar →
    integer
    | **real**

declaraciones_subprogramas →
    declaraciones_subprogramas declaración_subprograma ;
    | ϵ

declaración_subprograma →
    encab_subprograma declaraciones proposición_compuesta

encab_subprograma →
    function id *argumentos* : *tipo_estándar* ;
    | **procedure id** *argumentos* ;

argumentos →
    (*lista_parámetros*)
    | ϵ

lista_parámetros →
    lista_identificadores : *tipo*
    | *lista_parámetros* ; *lista_identificadores* : *tipo*

proposición_compuesta →
    begin
    proposiciones_optativas
    end

proposiciones_optativas →
    lista_proposiciones
    | ϵ

lista_proposiciones →
    proposición
    | *lista_proposiciones* ; *proposición*

proposición →
    variable **opasigna** *expresión*
    | *proposición_procedimiento*
    | *proposición_compuesta*
    | **if** *expresión* **then** *proposición* **else** *proposición*
    | **while** *expresión* **do** *proposición*

variable →
    id
    | **id** [*expresión*]

proposición_procedimiento →
    id
    | **id** (*lista_expresiones*)

lista_expresiones →
    expresión
    | *lista_expresiones* , *expresión*

expresión →
    expresión_simple
    | *expresión_simple* **oprel** *expresión_simple*

expresión_simple →
    término
    | *signo término*
    | *expresión_simple* **opsuma** *término*

término →
    factor
    | *término* **opmult** *factor*

factor →
    id
    | **id** (*lista_expresiones*)
    | **núm**
    | (*expresión*)
    | **not** *factor*

signo →
    + | −

A.4 CONVENCIONES LEXICOGRAFICAS

La notación para especificar los componentes léxicos proviene de la sección 3.3.

1. Los comentarios se encierran entre { y }. No pueden contener un {. Los comentarios pueden aparecer después de cualquier componente léxico.

2. Los espacios en blanco entre los componentes léxicos son opcionales, con la excepción de que las palabras clave deben ir encerradas entre espacios en blanco, caracteres de nueva línea, el comienzo del programa o el punto final.

3. El componente léxico **id** para los identificadores concuerda con una letra seguida de letras o dígitos:

    letra → `[a-zA-Z]`
    dígito → `[0-9]`
    id → **letra** (**letra** | **dígito**)*

    El implantador puede desear poner un límite a la longitud de un identificador.

4. El componente léxico **núm** concuerda con enteros sin signo (véase Ejemplo 3.5):

$$\begin{aligned}
\textbf{dígitos} &\rightarrow \textbf{dígito dígito}^* \\
\textbf{fracción_optativa} &\rightarrow \textbf{.}\ \textbf{dígitos} \mid \epsilon \\
\textbf{exponente_optativo} &\rightarrow (\ \texttt{E}\ (\ \texttt{+} \mid \texttt{-} \mid \epsilon\ )\ \textbf{dígitos}\ ) \mid \epsilon \\
\textbf{núm} &\rightarrow \textbf{dígitos fracción_optativa exponente_optativo}
\end{aligned}$$

5. Las palabras clave son reservadas y aparecen en **negritas** en la gramática.

6. Los operadores de relación (**oprel**) son =, <>, <, <=, >=, y >. Obsérvese que <> indica $\neq$.

7. Los operadores **opsuma** son +, − y `or`.

8. Los operadores **opmult** son *, /, `div`, `mod`, y `and`.

9. El lexema para el componente léxico **opasigna** es :=.

A.5 EJERCICIOS PROPUESTOS

Un ejercicio de programación adecuado para un curso de un semestre es escribir un intérprete para el lenguaje anteriormente definido, o para un subconjunto similar de otro lenguaje de alto nivel. El proyecto implica traducir el programa fuente a una representación intermedia como cuádruplos o código para máquina de pila y después interpretar la representación intermedia. Se propondrá un orden para la construcción de los módulos. El orden es distinto del orden en el que los módulos se ejecutan en el compilador porque es conveniente tener un intérprete operativo para depurar los otros componentes del compilador.

1. *Diséñese un mecanismo para la tabla de símbolos.* Decídase la organización de la tabla de símbolos. Téngase en cuenta la información reunida sobre nombres, pero déjese la estructura de registros de la tabla de símbolos flexible en ese momento. Escríbanse rutinas para:

i) Buscar en la tabla de símbolos un determinado nombre, crear una nueva entrada para ese nombre si no hay ninguna y devolver en cualquier caso un apuntador al registro correspondiente a dicho nombre.

ii) Borrar de la tabla de símbolos todos los nombres locales a un procedimiento dado.

2. *Escríbase un intérprete de cuádruplos.* El conjunto exacto de cuádruplos se puede dejar abierto en este momento, pero deberá incluir las proposiciones aritméticas y de saltos condicionales correspondientes al conjunto de operadores del lenguaje. Inclúyanse también operaciones lógicas si las condiciones se evalúan aritméticamente en lugar de hacerlo por posición en el programa. Además, hay que prever la necesidad de "cuádruplos" para conversión de entero a real, para marcar el comienzo y el final de procedimientos, y para pasar parámetros y llamadas a procedimientos.

También es necesario en este momento diseñar la secuencia de llamadas y la organización en el momento de la ejecución para los programas que están siendo interpretados. La organización de pila sencilla estudiada en la sección 7.3 es adecuada

para el lenguaje de ejemplo, porque no se permiten declaraciones de procedimientos anidadas en el lenguaje; es decir, las variables son globales (declaradas al nivel del programa completo) o son locales a un solo procedimiento.

Para simplificar, se podría utilizar otro lenguaje de alto nivel en lugar del intérprete. Cada cuádruplo puede ser una proposición de un lenguaje de alto nivel como C o incluso Pascal. Entonces la salida del compilador es una secuencia de proposiciones en C que se pueden compilar en un compilador de C existente. Este enfoque permite que el implantador se concentre en la organización para la ejecución.

3. *Escríbase el analizador léxico.* Selecciónense códigos internos para los componentes léxicos. Decídase cómo se representarán las constantes en el compilador. Cuéntense las líneas para un uso posterior por parte del manejador de mensajes de error. Hágase un listado del programa fuente si se desea. Escríbase un programa para introducir las palabras reservadas en la tabla de símbolos. Diséñese el analizador léxico para que sea una subrutina llamada por el analizador sintáctico, devolviendo un par (componente léxico, valor de atributo). Por el momento, los errores detectados por su analizador léxico pueden tratarse llamando a una rutina para impresión de errores y suspendiendo la ejecución.

4. *Escríbanse las acciones semánticas.* Escríbanse rutinas semánticas para generar los cuádruplos. Habrá que modificar la gramática en algunos lugares para hacer más fácil la traducción. Consúltense las secciones 5.5 y 5.6 si se quieren ejemplos de cómo modificar de manera útil la gramática.

5. *Escríbase el analizador sintáctico.* Si hay un generador de analizadores sintácticos LALR disponible, simplificará considerablemente el trabajo. Si está disponible un generador de analizadores sintácticos que maneje gramáticas ambiguas, como YACC, entonces se pueden combinar los no terminales que indiquen expresiones. Además, el problema del "**else** ambiguo" se puede resolver desplazando cada vez que ocurra un conflicto de desplazamiento/reducción.

6. *Escríbanse las rutinas para el manejo de errores.* Hay que estar preparado para recuperarse de errores léxicos y sintácticos. Imprímanse diagnósticos de errores para los errores léxicos, sintácticos y semánticos.

7. *Evaluación* El programa de la figura A.1 puede servir como una simple rutina de prueba. Otro programa de prueba se puede basar en el programa en Pascal de la figura 7.1. El código para la función `partición` de la figura corresponde al fragmento marcado en el programa en C de la figura 10.2. Ejecute su programa en un compilador, si hay alguno disponible. Determínense las rutinas en las que se emplea la mayor parte del tiempo. ¿Qué módulos habría que modificar para incrementar la velocidad de su compilador?

A.6 EVOLUCION DEL INTERPRETE

Un enfoque alternativo para construir un intérprete para el lenguaje es comenzar implantando una calculadora de bolsillo, es decir, un intérprete de expresiones. Añádanse gradualmente construcciones al lenguaje hasta que se obtenga un intér-

prete para todo el lenguaje. En Kernighan y Pike [1984] se adopta un enfoque similar. Un orden propuesto para añadir construcciones es:

1. *Tradúzcanse las expresiones a la notación postfija.* Utilizando un analizador sintáctico descendente, como en el capítulo 2, o un generador de analizadores sintácticos, familiarícese con el entorno de programación escribiendo un traductor de expresiones aritméticas simples a notación postfija.

2. *Añádase un analizador léxico.* Permítanse que en el traductor anteriormente construido aparezcan palabras clave, identificadores y números. Redestínese el traductor para que produzca código para una máquina de pila o cuádruplos.

3. *Escríbase un intérprete de la representación intermedia.* Como se estudió en la sección A.5, se puede utilizar un lenguaje de alto nivel en lugar del intérprete. Por el momento, el intérprete sólo tiene que admitir operaciones aritméticas, asignaciones, y entrada y salida. Amplíese el lenguaje permitiendo declaraciones de variables globales, asignaciones y llamadas a procedimientos `read` y `write`. Estas construcciones permiten que se compruebe el intérprete.

4. *Añádanse proposiciones.* Un programa en el lenguaje consta ahora de un programa principal sin declaraciones de subprogramas. Compruébense tanto el traductor como el intérprete.

5. *Añádanse procedimientos y funciones.* La tabla de símbolos debe permitir ahora que los ámbitos de los identificadores se limiten a cuerpos de procedimientos. Diséñese una secuencia de llamadas. De nuevo, la organización de pila simple de la sección 7.3 es adecuada. Amplíese el intérprete para que admita la secuencia de llamada.

A.7 AMPLIACIONES

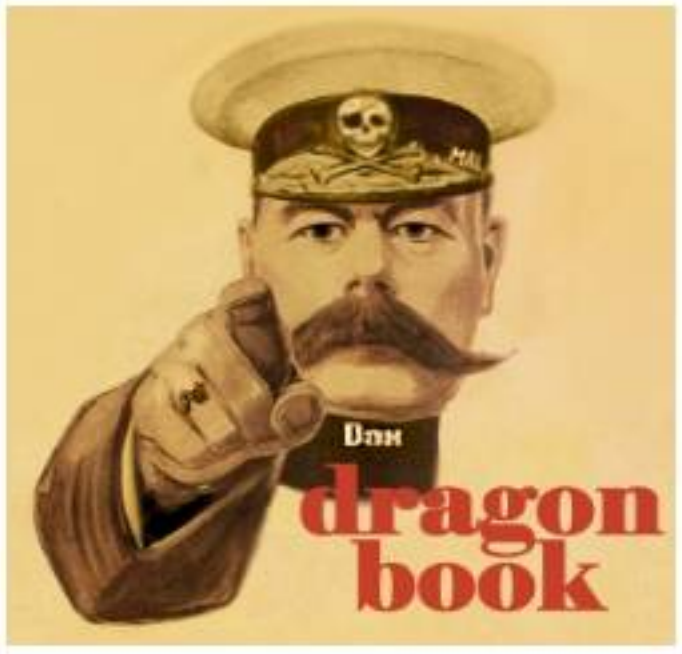
Se pueden añadir varias características al lenguaje sin complicar demasiado la compilación. Entre éstas están:

1. matrices multidimensionales
2. proposiciones **for** y **case**
3. estructuras de bloque
4. estructuras de registros, **record**

Si el tiempo lo permite, añádanse una o más de estas ampliaciones a su compilador.

Bibliografía

Páginas 771 a 790
No incluidas en
Spanish Edition ©



Indice de materias

Vocabulario bilingüe de términos técnicos (inglés-español)

absolute machine code
código de máquina absoluto, 5, 19, 530, 531
abstract machine
máquina abstracta, 64
(*véase también stack machine*)
abstract syntax tree
árbol de sintaxis abstracta, 49
(*véase también syntax tree*)
acceptance
aceptación, 117, 118, 205
accepting state
estado de aceptación, 117
access link
enlace de acceso, 410, 411, 430-34, 436
action table
tabla de acciones, 223
activation
activación, 401
activation environment
entorno de activación, 471, 472
activation record
registro de activación, 410-24, 538, 539, 543, 544
activation tree
árbol de activación, 403-05
actual parameter
parámetro actual, 402-12
acyclic graph
grafo acíclico
(*véase directed acyclic graph*)
adress descriptor
descriptor de direcciones, 553, 554
address mode
modo de direccionamiento, 18, 19, 535, 536, 595
adjacency list
lista de adyacencias, 117
advancing edge
arista de avance, 682
affix grammar
gramática afija, 351
algebraic transformation
transformación algebraica, 548, 573, 583, 618, 620, 757
alias
alias, 666-78, 739
alignment, of data
alineación de datos, 411, 412, 487
alphabet
alfabeto, 94
alternative
alternativa, 171
ambiguity
ambigüedad, 30, 176, 179, 180, 189, 197, 206, 207, 235, 236, 254-61, 268-71, 594
anbiguous definition
definición ambigua, 628
analysis
análisis, 2-10
(*véanse también lexical analysis; parsing*)
annotated parse tree
árbol de análisis sintáctico con anotaciones, 34, 288
arithmetic operator
operador aritmético, 373
array
matriz, 357, 361, 440
array reference
referencia a una matriz, 208, 498, 499, 569, 598, 606, 667
assembly code
código ensamblador, 4, 15, 17-19, 91, 531, 535
assignment statement
proposición de asignación, 65, 481, 492-503
associativity
asociatividad, 30, 32, 97, 98, 213, 254-56, 270, 271, 702

attribute
atributo, 11, 33, 89, 267, 288
(*véanse también inherited attribute; lexical value; syntax directed definition; synthesized attribute*)
atribute grammar
gramática con atributos, 289, 596
augmented dependency graph
grafo de dependencias aumentado, 344
augmented grammar
gramática aumentada, 228
automatic code generator
generador automático de código, 23
available expression
expresión disponible, 645-49, 677, 678, 702, 712

back edge
arista de retroceso, 622, 624, 682
back end
etapa final, 20, 64
backpatching
relleno de retroceso, 21, 22, 515-20, 531
bactracking
búsqueda de retroceso, 186
Backus-Naur form
forma de Backus-Naur
backward data, flow equations
ecuaciones de flujo de datos inverso, 642, 717-20
basic block
bloque básico, 545-49, 609, 616-20, 627, 722, 723
(*véase también extended basic block*)
basic induction variable
variable básica de inducción, 661
basic symbol
símbolo básico, 97, 124
basic type
tipo básico, 357
binary alphabet
alfabeto binario, 94
binding, of names
enlazado de nombres, 408
block
bloque
(*véanse basic block; block structure; common block*)
block structure
estructura de bloques, 425, 426, 452-54
body
cuerpo
(*véase procedure body*)
boolean expression
expresión booleana, 336-38, 502-11, 515-18

bootstrapping
arranque, 743-47
bottom-up parsing
análisis sintáctico ascendente, 41, 200, 299, 302-04, 317-25, 477
(*véanse también LR parsing; operator precedence parsing; shift-reduce parsing*)
bounded context parsing
análisis sintáctico por acotamiento de contexto, 285
branch optimization
optimación de saltos, optimación de ramificaciones, 758, 760
break statement
proposición BREAK, 639-41
bucket
cubeta, 448
(*véase también hashing*)
buffer
buffer, 58, 63, 90-94, 131, 132
byte
byte, 411, 412

call
llamada
(*véase procedure call*)
call-by-address
llamada por dirección
(*véase call-by-reference*)
call-by-location
llamada por dirección
(*véase call-by-reference*)
call-by-name
llamada por nombre, 442, 443
call-by-reference
llamada por referencia, 440
call-by-value
llamada por valor, 438-40, 443
calling sequence
secuencia de llamada, 417-21, 521-23
canonical collection of sets of items
colección canónica de conjuntos de elementos, 228, 230, 237, 238
canonical derivation
derivación canónica, 173
canonical LR parsing
análisis sintáctico LR canónico, 236, 240, 242, 262
canonical LR parsing table
tabla de análisis sintáctico LR canónico, 236, 240, 242
cartesian product
producto cartesiano, 357, 358
case statement
proposición CASE, 511, 514

CFG
CFG
(*véase context-free grammar*)
changed variable
variable modificada, 676-78
character class
clase de caracteres, 94, 98, 99, 150, 151
(*véase también alphabet*)
child
hijo, 29
Chomsky normal form
forma normal de Chomsky, 284
circular syntax-directed definition
definición circular dirigida por sintaxis, 295, 345, 346, 352
closure
cerradura, 95, 96
closure, of set of items
cerradura de un conjunto de elementos, 237-39, 230-32
CNF
CNF
(*véase Chomsky normal form*)
Cocke-Younger-Kasami algorithm
algoritmo de Cocke-Younger-Kasami, 164, 284
code generation
generación de código, 15, 16, 529-600, 754, 755
code hoisting
elevación de código, 732
code motion
traslado de código, 614, 657-61, 728, 729, 739
code optimization
optimación de código, 14, 15, 477, 486, 495, 529, 547, 570-75, 757, 758
coercion
coacción, coerción, 356, 371, 372, 398
coloring
coloración
(*véase graph coloring*)
column-major form
forma por columnas, 496, 497
comment
comentario, 86, 87
common block
bloque COMMON, 446, 460-62, 469
common lisp
COMMON LISP, 475
common subexpression
subexpresión común, 299, 547, 563, 567, 583, 583, 610-13, 617, 618, 651-53, 727, 757-61
(*véase también available expression*)
commutativity
conmutatividad, 702
compactation, of storage
compactación de la memoria, 460
compiler-compiler
compilador de compiladores, 22
composition
composición, 702
compression
compresión
(*véase encoding of types*)
concatenation
concatenación, 35, 96-98
concrete syntax tree
árbol de sintaxis concreta, 49
(*véase también syntax tree*)
condition code
código de condición, 558
conditional expression
expresión condicional
(*véase boolean expression*)
confliguration
configuración, 223
conflict
conflicto
(*véanse disambiguation rule; reduce/reduce conflict; shift/reduce conflict*)
confluence operator
operador de confluencia, 642, 698, 713
congruence closure
cerradura de congruencia
(*véase congruent nodes*)
congruent nodes
nodos congruentes, 397, 400
conservative approximation
aproximación conservadora, 629, 632, 633, 648, 670-72, 677, 678, 707, 708
constant folding
cálculo previo de constantes, 610, 613, 619, 700-07
context-free grammar
gramática independiente del contexto, 25, 26, 29, 41, 82, 169-85, 288
(*véanse también LL grammar; LR grammar; operator grammar*)
context-free language
lenguaje independiente del contexto, 172, 177, 183-85
contiguous evaluation
evaluación contigua, 586
control flow
flujo de control, 66, 67, 484-86, 506, 522, 572, 624, 629, 639, 707, 708, 738
control link
enlace de control, 410, 411, 419-21, 436

control stack
pila de control, 405, 406, 408, 409
copy propagation
propagación de copias, 610, 612, 613, 654-56
copy rule
regla de copia, 332, 333, 335, 442, 443
copy statement
proposición de copia, 481, 612
copy-restore-linkage
enlazado de copia y restauración, 441
core, of set of items
corazón de un conjunto de elementos, 243
cross compiler
compilador cruzado, 744
cross edge
arista cruzada, 682
cycle
ciclo, 181
cycle, in type graphs
ciclo en el grafo de tipos, 370, 371
cycle-free grammar
gramática sin ciclos, 278
CYK algorithm
algoritmo de CYK
(*véase Cocke-Younger-Kasami algorithm*)

DAG
GDA
(*véase directed acyclic graph*)
dangling else
else ambiguo, 178, 197, 207, 208, 257-59, 270, 275
dangling reference
referencia desactivada, 456, 457
data area
área de datos, 460, 461, 468, 469
data layout
disposición de los datos 411, 412, 487
data object
objeto de datos, dato
(*véase objet*)
data statement
proposición DATA, 415, 416
data-flow analysis
análisis de flujo de datos, 604, 626-40
data flow analysis framework
marco para análisis de flujo de datos, 699-713
data-flow engine
dispositivo para análisis de flujo de datos, 24, 708-12
data-flow equation
ecuación de flujo de datos, 626, 642, 698
dead code
código inactivo, 548, 610, 613
debugger
depurador, 419
debugging
depuración, 572
(*véase también symbolic debugging*)
declaration
declaración, 276, 406-08, 487-91, 524, 525
deep access
acceso profundo, 436, 437
default value
valor por omisión, 512
definition
definición, 545, 628, 629, 650
definition-use chain
cadena de definición y uso
(*véase du-chain*)
deletion, of locals
borrado de valores locales, 416
denotational semantics
semántica denotacional, 351
dependency graph
grafo de dependencias, 287, 288, 292-95, 341-45
depth, of a flow graph
profundidad de un grafo de flujo, 682, 690-92, 734
depth-first ordering
ordenamiento en profundidad, 306, 679, 680, 690-92
depth-first search
búsqueda en profundidad, 679-82
depth-first spanning tree
árbol de expansión en profundidad, 680
depth-first traversal
recorrido en profundidad, 37, 334, 405
derivation
derivación, 29, 171-76
deterministic finite automaton
autómata finito determinista, 115-24, 129-31, 135-39, 143-48, 152, 153, 185, 223, 232, 233
deterministic transition diagram
diagrama de transiciones determinista, 102
DFA
AFD
(*véase deterministic finite automaton*)
diagnostic
diagnóstico
(*véase error message*)
directed acyclic graph
grafo dirigido acíclico, 299, 300, 359, 478-80, 485, 563-70, 574-77, 598-600, 616-20, 624, 723-26

disambiguating rule
regla para eliminar ambigüedades, 176, 254, 261, 268-71
display
estructura de datos tipo *display*, 433-36
distance, between strings
distancia entre cadenas, 159
distributive framework
esquema distributivo, 704-07, 710, 711
distributivity
distributividad, 738
do statement
proposición DO, 88, 114
dominator
dominador, 621, 657, 688-90, 739
dominator tree
árbol de dominación, 620
du-chain
cadena de definición y uso, 650, 651, 661
dummy argument
argumento ficticio
(*véase formal parameter*)
dynamic allocation
asignación dinámica, 454-60
dynamic checking
comprobación dinámica, 355, 359
dynamic programming
programación dinámica, 284, 584-88, 600
dynamic scope
ámbito dinámico, 425

Earley's algorithm
algoritmo de Earley, 164
edit distance
distancia de edición, 158, 159
efficency
eficiencia, 87, 128-30, 145-48, 246-50, 288, 372, 447, 449-52, 465, 636-38, 743
(*véase también code optimization*)
egrep
egrep, 160
emitter
emisor, 67-69, 72
empty set
conjunto vacío, 94
empty string
cadena vacía, 27, 92, 96
encoding of types
codificación de tipos, 366, 367
entry, to a loop
entrada a un ciclo, 550
environment
ambiente, 407, 471
ϵ -closure
cerradura ϵ, 120-22, 232
ϵ -free grammar
gramática sin producciones ϵ, 277
ϵ -production
producción ϵ, 181, 182, 194, 278
ϵ -transition
transición ϵ, 116, 118, 137
equivalence, of basic blocks
equivalencia entre bloques básicos, 547
equivalence, of finite automata
equivalencia de autómatas finitos, 400
equivalence, of grammars
equivalencia entre gramáticas, 172
equivalence, of regular expressions
equivalencia entre expresiones regulares, 97, 153
equivalence, of syntax-directed definitions
equivalencia entre definiciones dirigidas por sintaxis, 311-14
equivalence, of type expressions
equivalencia entre expresiones de tipos, 364-71
(*véase también unification*)
equivalence statement
proposición EQUIVALENCE, 446, 462-69
equivalence, under a substitution
equivalencia bajo una sustitución, 383, 389-91
error handling
manejo de errores, 12, 73, 90, 165, 166
(*véanse también lexical error; logical error; semantic error; syntax error*)
error message
mensaje de error, 199, 217-21, 263, 264
error productions
producciones de error, 168, 169, 272, 273
evaluation order, for basic blocks
orden de evaluación para bloques básicos, 534, 574-78
evaluation order, for syntax-directed definitions
orden de evaluación para definiciones dirigidas por sintaxis, 294, 295, 307-09, 325-47
evaluation order, for trees
orden de evaluación para árboles, 577-96
explicit allocation
asignación explícita, 454, 458
explicit type conversion
conversión explícita de tipos, 371
expression
expresión, 6, 7, 31, 32, 170, 299-301, 362, 363
(*véase también postfix expression*)
extended basic blocks
bloques básicos ampliados, 732

external reference
referencia externa, 19

failure function
función de fallo, 154, 156, 157
family, of an induction variable
familia, de una variable de inducción, 662
feasible type
tipo factible, 375
fgrep
fgrep, 160
Fibonacci string
cadena de Fibonacci, 155
field, of a record
campo de un registro, 491-93, 502, 503
final state
estado final
(*véase accepting state*)
find
encuentra, 390, 391
finite automaton
autómata finito, 115-47
(*véase también transition diagram*)
FIRST
PRIMERO, 46, 193, 194, 198, 199
firstpos
primerapos, 138-43
flow graph
grafo de flujo, 545, 549, 550, 563, 609, 620, 624
(*véase también reducible flow graph*)
flow of control
flujo de control, 545
(*véase también control flow*)
flow-of-control check
comprobación de flujo de control, 355
FOLLOW
SIGUIENTE, 193-96, 198, 199, 236, 237
followpos
siguientepos, 138-43
formal parameter
parámetro formal, 402, 403
Fortran
FORTRAN, 2, 88, 114, 115, 159-61, 214, 398, 408, 409, 414-16, 441, 446, 460-69, 496, 620, 736, 741
Fortran H
FORTRAN H, 559, 599, 746, 755-58
forward data-flow equations
ecuaciones de flujo de datos hacia adelante, 642, 716-21
forward edge
arista de avance, 624
fragmentation
fragmentación, 458

frame
marco
(*véase activation record*)
front end
etapa inicial, 20, 64
function
función
(*véase procedure*)
function type
tipo de una función, 358, 359, 364, 373-76
(*véase también polymorphic function*)

garbage collection
recogida de basura, 456
generation, of a string
generación de una cadena, 29
generic function
función genérica, 376
(*véase también polymorphic function*)
global error correction
corrección global de errores, 168, 169
global name
nombre global, 671
(*véase también nonlocal name*)
global optimization
optimación global, 610, 651
global register allocation
asignación global de registros, 559-63
GNF
FNG
(*véase Greibach normal form*)
goto, of set of items
función *ir_a* de un conjunto de elementos, 228, 230, 231, 237, 238, 245, 246
goto statement
proposición **goto**, 520, 521
goto table
tabla de la función *ir_a*, 222
graph
grafo
(*véanse dependency; directed acyclic graph; flow graph; interval graph; reducible flow graph; register interference graph; search; transition graph; tree; type graph*)
graph coloring
coloración de grafos, 562, 563
Greibach normal form
forma normal de Greibach, 279
grep
grep, 160

handle
mango, 201-06, 211, 212, 216, 233

handle pruning
poda, 203, 204
hash function
función de dispersión, 448-52
hans table
tabla de dispersión, 512
hashing
dispersión, 301, 302, 447-54, 473, 474
hashpjw
dispersión pjw, 450-52
head
cabeza, 622
header
encabezamiento, 621, 622, 630, 683
heap
montículo, 409, 410, 753
heap allocation
asignación por medio de un montículo, 415, 416, 423, 424, 454-60
Helsinki Language Processor
(*véase HLP*)
hierarchical analysis
análisis jerárquico, 5
(*véase también parsing*)
HLP
HLP, 286, 352
Hole in scope
excepción en el ámbito, 425
Hollerith string
cadena Hollerith, 100
Hope
Hope, 396

idempotence
idempotente, 98, 702
identifier
identificador, 56, 57, 88, 89, 183, 184
identity function
función identidad, 701, 702
if statement
proposición **if**, 114, 115, 505-07, 518-20
inmediate dominator
dominador inmediato, 621
immediate left recursion
recursión directa por la izquierda, 180
implicit allocation
asignación implícita, 454, 458-60
implicit type conversion
conversión implícita de tipos, 371
important state
estado significativo, 137
indexed addressing
direccionamiento indizado, 535, 536, 557
indirect addressing
direccionamiento indirecto, 535-37
indirect triples
triples indirectos, 486, 487
indirection
indirección, 486
induction variable
variable de inducción, 614-16, 661-66, 727, 739, 757
infix expression
expresión infija, 33
inherited attribute
atributo heredado, 34, 288, 289, 291, 292, 308, 309, 317-19, 321-26, 334, 335, 350, 351
(*véase también attribute*)
initial node
nodo inicial, 549
initial state
estado inicial
(*véase state*)
in-line expansion
expansión en línea, 442, 443
(*véase también macro*)
inner loop
lazo interno, 550, 623
input symbol
símbolo de entrada, 116
instance, of a polymorphic type
caso de un tipo polimórfico, 382
instruction selection
selección de instrucciones, 532
intermediate code
código intermedio, 12, 14, 477-527, 530, 607, 608, 722
(*véanse también abstract machine; postfix expression; quadruple; three address code; tree*)
interpreter
intérprete, 3, 4
interprocedural data flow analysis
análisis del flujo de datos entre procedimientos, 671-78
interval
intervalo, 683-86
interval analysis
análisis de intervalos, 641, 678, 686, 737, 738
interval depth
profundidad de intervalo, 690
interval graph
grafo de intervalos, 684, 685
interval partition
partición de intervalos, 684
item
elemento
(*véanse kernel item; LR(1) item; LR(0) item*)

iterative data-flow analysis
análisis iterativo de flujo de datos, 641-51, 690-92, 709-12

kernel item
elemento nuclear, 230, 248
keyword
palabra clave, 56, 57, 88, 89, 444
kill
eliminar, 626, 630-32, 645, 654
Kleene closure
cerradura de Kleene
(*véase closure*)
KMP algorithm
algoritmo de MP, 154, 155
Knuth-Morris-Pratt algorithm
algoritmo de Knuth-Morris-Pratt, 161
(*véase también KMP algorithm*)

label
etiqueta, 67, 68, 481, 520, 521, 531
LALR collection of sets of items
colección de conjuntos de elementos LALR, 245
LALR grammar
gramática LALR, 245, 246
LALR parsing
análisis sintáctico LALR
(*véase lookahead LR parsing*)
LALR parsing table
tabla de análisis sintáctico LALR, 242-52
lambda calculus
cálculo lambda, 399
language
lenguaje, 28, 94, 118, 172, 209
lastpos
últimapos, 138-41
lattice
retículo, 399
l-attributed definition
definición con atributos por la izquierda, 288, 305-23, 350
lazy state construction
construcción diferida de estados, 130, 160
leader
bloque inicial, 545
leaf
hoja, 29, 30
left associativity
asociatividad por la izquierda, 30, 31, 213, 270
left factoring
factorización por la izquierda, 182
left leaf
hoja izquierda, 578
left recursion
recursividad por la izquierda, 47-50, 72, 180-82, 196, 197, 312-14
leftmost derivation
derivacion por la izquierda, 173
left-sentential form
forma de frase izquierda, 173
lex
LEX, 85, 109-15, 131-32, 150, 151, 160, 749
lexeme
lexema, 12, 56, 57, 61, 62, 87, 444-46
lexical analysis
análisis léxico, 5, 12, 26, 54-60, 71, 85-161, 164, 176, 269, 272, 756
lexical environment
ambiente léxico, 471, 472
lexical error
error léxico, 90, 165
lexical scope
ámbito léxico, 424-36
lexical value
valor léxico, 12, 113, 289
library
biblioteca, 4, 5, 54
lifetime, of a temporary
duración de un valor temporal, 494, 495
lifetime, of an activation
duración de una activación, 403, 423
lifetime, of an attribute
duración de un atributo, 330, 331, 334-39
limit flow graph
grafo de flujo límite, 685, 686
linear analysis
análisis lineal, 4
(*véase también lexical analysis*)
link editor
editor de enlace, director de enlace, 19, 414
linked list
lista enlazada, 453
lint
lint, 359
lisp
LISP, 425, 454, 456, 475, 713, 743
literal string
cadena literal, 88
live variable
variable activa, 551, 560, 567, 613, 617, 618, 649, 650, 660, 671
LL grammar
gramática LL, 164, 166, 196, 197, 227, 277, 280, 285, 317-19
loader
cargador, 19
local name
nombre local, 408, 410-12, 425

local optimization
optimación local, 610, 651
logical error
error lógico, 165
longest common subsequence
subsecuencia común más larga, 158
lookahead
preanálisis, 92, 114, 115, 136, 221, 237
lookahead LR parsing
análisis sintáctico LR con examen por anticipado, 222, 242-50, 262, 285
(*véase también yacc*)
lookahead symbol
símbolo de preanálisis, 42
loop
lazo, 550, 562, 620-25, 634-36, 697
loop header
encabezamiento de un lazo
(*véase header*)
loop optimization
optimación de lazos, 613, 614
(*véanse también code motion; induction variable; reduction in strength*)
loop-invariant computation
cálculo de invariantes de ciclos
(*véase code motion*)
LR grammar
gramática LR, 164, 166, 207, 226, 227, 254, 280, 281, 285, 318
LR parsing
análisis sintáctico LR, 221-74, 352, 595
LR (1) grammar
gramática LR (1), 242
LR (1) item
elemento del análisis sintáctico LR (1), 237
LR (0) item
elemento del análisis sintáctico LR (0), 228
l-value
valor de lado izquierdo, 65, 66, 236, 407, 438-43, 563

machine code
código de la máquina, 5, 18, 19, 585, 593
machine status
estado de la máquina, 410, 419-21
macro
macro, 16, 17, 86, 444, 470
make
make, 747-49
manifest constant
constante manifiesta, 110
marker nonterminal
no terminal marcador, 318, 321-25, 351
meet operator
operador de reunión, 699
meet-over-paths solution
solución de reunión sobre caminos, 707, 708, 710, 712
memory map
mapa de memoria, 461
memory organization
organización de la memoria
(*véase storage organization*)
minimum-distance error correction
corrección de errores de distancia mínima, 90
mixed strategy precedence
precedencia de estrategia mixta, 285
mixed-mode expression
expresión de modo mixto, 510-12
modula
MODULA, 625, 737, 760, 761
monotone framework
marco monótono, 704-06, 710, 711
monotonicity
monotonicidad, 738
most closely nested rule
regla del anidamiento más cercano, 425, 429
most general unifier
unificador más general, 382, 383, 388, 389

name
nombre, 401
name equivalence, of type expressions
equivalencia entre nombres en las expresiones de tipos, 368, 369
name-related check
comprobaciones relacionadas con nombres, 355
natural loop
lazo natural, 621, 622
neliac
NELIAC, 526
nested procedures
procedimientos anidados, 429-36, 489-91
nesting depth
profundidad de anidamiento, 430
nesting, of activations
anidamiento de activaciones, 403
NFA
AFN
(*véase nondeterministic finite automaton*)
node splitting
separación de nodos, 685, 686, 697, 698
nondeterministic finite automaton, 115, 116, 119-30, 132-35, 138, 139
nondeterministic transition diagram
diagrama de transiciones no determinista, 189

nonlocal name
nombre no local, 424-38, 544
nonreducible flow graph
grafo de flujo no reducible, 625, 697, 698
nonregular set
conjunto no regular, 185, 186
(*véase también regular set*)
nonterminal
no terminal, 27, 170-72, 210-12
(*véase también marker nonterminal*)
nullable expression
expresión anulable, 138-41

object
objeto, 401, 407
object code
código objeto, 722
object language
lenguaje objeto
(*véase target language*)
offset
desplazamiento, 410, 412, 464, 477, 478, 541
one-pass compiler
compilador de una pasada
(*véase single-pass translation*)
operator grammar
gramática de operadores, 209, 210
operator identification
identificación de operadores, 373
(*véase también overloading*)
operator precedence grammar
gramática de precedencia de operadores, 278, 279
operator precedence parsing
análisis sintáctico por precedencia de operadores, 209-21, 285, 754
operator precedence relations
relaciones de precedencia de operadores, 209, 210
optimizing compiler
compilador optimizador
(*véase code optimization*)
overloading
sobrecarga, 341, 345, 373-76, 397, 398

packed data
datos empaquetados, 412
padding
relleno, 412
panic mode
modo de pánico, 90, 168, 198, 262
parameter passing
paso de parámetros, 427-29, 437-39, 671, 672
parentheses
paréntesis, 97, 100, 177, 178
parse tree
árbol de análisis sintáctico, 6-8, 28-30, 40-43, 49, 50, 164, 173-76, 201, 202, 287, 305, 306
parser generator
generador de analizadores sintácticos, 749
parsing
análisis sintáctico, 6, 7, 12, 30, 31, 40, 49, 57, 58, 73, 86, 87, 163
(*véase también bottom-up parsing; bounded context parsing; top-down parsing*)
partial order
orden parcial, 343, 344
partition
partición, 144, 145
pass
pasada, 20-22
passing environment
ambiente de pasada, 471, 472
pattern matching
concordancia de patrones, 87, 131-33, 594, 600
P-code
código P, 761
peephole optimization
optimación local, 571-75, 600
period, of a string
periodo de una cadena, 155
phase
fase, 10, 11
(*véanse también code generation; code optimization; error handling; lexical analysis; parsing; semantic analysis*)
phrase-level error recovery
recuperación de errores a nivel de frase, 169, 199, 200, 262
pic
pic, 470
pig Latin
seudolatín, 80
point
punto, 627, 628
pointer
apuntador, 361, 422, 482, 557, 569, 570, 598, 599, 666-71
pointer type
tipo apuntador, 358
polymorphic function
función polimórfica, 356, 376-88
polymorphic type
tipo polimórfico, 380
pop
sacar, 65

portability
transportabilidad, 87, 742
portable C compiler
compilador transportable del lenguaje C
positive closure
cerradura positiva, 98
(*véase también closure*)
postfix expression
rotación postfija, 25, 33, 478, 480, 484, 523
postorder traversal
recorrido en orden posterior, 578
precedence
precedencia, 31, 32, 97, 213, 254-56, 270, 271
(*véase también operator precedence grammar*)
precedence function
función de precedencia, 214-16
precedence relations
relaciones de precedencia
(*véase operator precedence relations*)
predecessor
predecesor, 549
predictive parsing
análisis sintáctico predictivo, 44-48, 187-93, 198, 199, 221, 311-18
predictive translator
traductor predictivo, 316-18
prefix
prefijo, 95
prefix expression
forma prefija de una expresión, 524
preheader
preencabezamiento, 623, 625
preprocessor
preprocesador, 4, 5, 16
pretty printer
impresora estética, 3
procedure
procedimiento, 401, 402
procedure body
cuerpo de un procedimiento, 402
procedure call
llamada a un procedimiento, 208, 408, 410, 417-24, 481, 521, 522, 538-43, 570, 666
(*véase también interprocedural data flow analysis*)
procedure definition
definición de un procedimiento, 402
procedure parameter
procedimientos como parámetros, 428, 432, 433
product
producto
(*véase cartesian product*)
production
producción, 26, 170
profile
perfilar, 605
programming language
lenguaje de programación
projet, programming
proyecto de programación, 763-69
propagation, of lookahead
propagación de símbolos de anticipación, 249
push
insertar, 65

quadruples
cuádruplos, 484, 486, 487
query interpreter
intérprete de consultas, 4
queue
cola, 522
quicksort
clasificación por particiones, 402, 606

raftor
RATFOR, 741
reaching definition
definición de alcance, 628-39, 642-45, 670, 671, 692-98, 702
recognizer
reconocedor, 115
record type
tipo record, 358, 371, 492
recursión
recursión, 6, 7, 169, 326, 327, 339, 340, 404, 414
(*véanse también left recursion; right recursion; tail recursion*)
recursive-descent parsing
análisis sintáctico descendente recursivo, 44, 186, 187, 754, 758
reduce/reduce conflict
conflicto de reducción/reducción, 207, 244, 270, 595
reducible flow graph
grafo de flujo reducible, 624-26, 682, 685, 686, 732, 733, 758
reduction
reducción, 200, 201, 204, 205, 217-19, 222, 263
reduction in strength
reducción de intensidad, 574, 614-16, 619, 663, 664
redundant code
código redundante, 571

reference
referencia
(*véase use*)
reference count
cuenta de referencia, 459
region
región, 630, 687, 688, 692-98
register
registro, 535-38
register allocation
asignación de registros, 533, 558-63, 579-82, 739-43
register assignment
asignación a registros, 15, 533, 553-57, 561, 562
register descriptor
descriptor de registros, 553, 554
register pair
par de registros, 533, 582
register-interference graph
grafo de interferencia entre registros, 562, 563
regression test
prueba de regresión, 749
regular definition
definición regular, 98, 110
regular expression
expresión regular, 85, 96-100, 109, 110, 115, 116, 123-27, 131, 137, 138, 142, 150, 151, 177, 276
regular set
conjunto regular, 100
rehostability
relocalización (de un compilador), 742
relative address
dirección relativa
(*véase offset*)
relocatable machine code
código de máquina relocalizable, 5, 18, 530, 531
relocation bit
bit de relocalización, 19
renaming
renombramiento, 548
reserved word
palabra reservada, 57, 89
retargetability
redestinabilidad, 742
retargeting
redestinación (de un compilador), 477
retention, of locals
conservación de valores locales, 414, 415, 423
retreating edge
arista de retroceso, 681, 682
return address
dirección de retorno, 421, 538-44
return node
nodo de retorno, 672
return sequence
secuencia de retorno, 417-21
return value
valor devuelto, 411, 419, 420
right associativity
asociatividad por la derecha, 31, 213, 270
right leaf
hoja derecha, 578
right recursion
recursión por la derecha, 48
rightmost derivation
derivación por la derecha, 173, 200-02
right-sentential form
forma de frase derecha, 173, 200-02
root
raíz, 29, 30
row-major form
forma por filas, 496
run-time support
apoyo durante la ejecución, 401
(*véanse también heap allocation; stack allocation*)
r-value
valor de lado derecho, 65, 235, 407, 438-43, 563

safe approximation
aproximación segura
(*véase conservative approximation*)
s-attributed definition
definición con atributos sintetizados, 290, 302, 303
save statement
proposición SAVE, 415, 416
scanner
analizador léxico, 86
scanner generator
generador de analizadores léxicos, 23
(*véase también lex.*)
scanning
análisis léxico
(*véase lexical analysis*)
scope
ámbito, 406, 407, 424, 425, 452-54, 473, 489-92
search, of a graph
búsqueda en un grafo, 121
(*véase también depth-first search*)
semantic action
acción semántica, 38, 39, 267, 268

semantic analysis
análisis semántico, 5, 8
semantic error
error semántico, 165, 360
semantic rule
regla semántica, 34, 287-95
semantics
semántica, 25
sentence
frase, 94, 172
sentential form
forma de frase, 172
sentinel
centinela, 93
shallow access
acceso superficial, 437
shared node
nodo compartido, 583, 584
shell
shell, 152
shift
desplazamiento, 204, 205, 222
shift/reduce conflict
conflicto de desplazamiento/reducción, 207, 219-21, 244, 270, 271, 595
shift-reduce parsing
análisis sintáctico por desplazamiento y reducción, 204-08
(*véanse también LR parsing; operator precedence parsing*)
short-circuit code
código en cortocircuito, 505
side effect
efecto colateral, 289
signature, of a DAG node
signatura de un nodo de un GDA, 300
silicon compiler
compilador de circuitos, 4
simple LR parsing
análisis sintáctico LR sencillo, 222, 227-36, 262, 285
simple precedence
precedencia simple, 285
simple syntax-directed translation
traducción simple dirigida por sintaxis, 39, 40, 307
single production
producción simple, 255, 278
single-pass translation
traducción de una pasada, 288, 753
skeletal parse tree
estructura de árbol de análisis sintáctico, 212
SLR grammar
gramática SLR, 234
SLR parsing
análisis sintáctico SLR
(*véase simple LR parsing*)
SLR parsing table
tabla de análisis sintáctico SLR, 233-37
snobol
SNOBOL, 425
sound type system
sistema de tipos seguro, 360
source language
lenguaje fuente, 1
spontaneous generation, of lookahead
generación espontánea del examen por anticipado, 248
stack
pila, 128, 129, 191, 204, 223, 262, 263, 283, 284, 299, 303, 304, 319-25, 334-38, 405, 406, 409, 410, 490-94, 579, 753
(*véase también control stack*)
stack allocation
asignación de la pila, 415, 417-26, 538, 541-44
stack machine
máquina de pila, 64-68, 478, 600
start state
estado inicial, 102
start symbol
símbolo inicial, 27-29, 170, 290
state
estado, 102, 103, 116, 117, 156, 222, 223, 303
state minimization
minimización de estados, 144-47
state (of storage)
estado (de la memoria), 407
statement
proposición, 26, 28, 32, 67, 68
(*véanse también assignment statement; case statement; copy statement; do statement; equivalence statement; goto statement; if statement; while statement*)
static allocation
asignación estática, 413-16, 538-41, 544
static checking
comprobación estática, 3, 355, 359
static scope
ámbito estático
(*véase lexical scope*)
storage
memoria, 407, 408
storage allocation
asignación de memoria, 413-24, 455-60
storage organization
organización de memoria, 408-12
string
cadena, 94, 171, 172

string table
tabla de cadenas, 445
strongly noncircular syntax-directed definition
definición dirigida por sintaxis fuertemente no circular, 342-46, 351
strongly typed language
lenguaje fuertemente tipificado, 360
structural equivalence, of type expressions
equivalencia estructural de las expresiones de tipos, 365-67, 388, 392
structure editor
editor de estructuras, 3
subsequence
subsecuencia, 95
subset construction
construcción de subconjuntos, 120-23, 137
substitution
sustitución, 382, 383, 388-91
substring
subcadena, 95
successor
sucesor, 549
suffix
sufijo, 95
switch statement
proposición **switch**
(*véase case statement*)
symbol table
tabla de símbolos, 7, 11, 61-63, 86, 164, 443-54, 484, 487,94, 721
symbolic debugging
depuración simbólica, 721-29
symbolic dump
volcado simbólico, 552
symbolic register
registro simbólico, 562
synchronizing token
componente léxico de sincronización, 198, 199
syntax
sintaxis, 25
(*véase también context-free grammar*)
syntax analysis
análisis sintáctico
(*véase parsing*)
syntax error
error sintáctico, 165-69, 198-200, 205, 212, 216, 221, 224, 262-64, 272-74, 282, 283, 286
syntax tree
árbol sintáctico, 2, 8, 50, 295-99, 478-80, 485
(*véanse también abstract syntax tree; concrete syntax tree; parse tree*)
syntax-directed definition
definición dirigida por la sintaxis, 33, 34, 287-94
(*véanse también annotated parse tree; syntax-directed translation*)
syntax-directed translation
traducción dirigida por la sintaxis, 8, 25, 33-40, 287-353, 478, 479, 482-84
syntax-directed translation engine
dispositivo de traducción dirigida por la sintaxis, 23
synthesized attribute
atributo sintetizado, 34, 288-90, 308, 309, 325, 326, 335
(*véase también attribute*)

table compression
compresión de tablas, 147, 148, 153, 252-54
table-driven parsing
análisis sintáctico guiado por tablas, 191, 196-98, 222-26
(*véanse también canonical LR parsing; LALR parsing; operator precedence parsing; SLR parsing*)
tail
cola, 621
tail recursion
recursión por el final, 53
target language
lenguaje objeto, 1
target machine
máquina objeto, 742, 743
T-diagram
diagrama T, 744-47
temporary
valores temporales, 410, 484, 494, 495, 551, 552, 653, 657
terminal
terminal, 26, 169-72, 289, 290
testing
prueba, 749, 750
text editor
editor de textos, 160
text formatter
formador de textos, 4, 8-10
three-address code
código de tres direcciones, 14, 480-86
token
componente léxico, 4, 5, 12, 26, 27, 56, 57, 86-88, 100, 169, 170, 184
T_1-T_2 analysis
análisis T_1-T_2, 686, 687, 692-98
tools
herramientas, 742

value-result linkage
enlace por valor y resultado
(*véase copy-restore-linkage*)
variable
variable
(*véanse identifier; type variable*)
variable-length data
datos de longitud variable, 419, 422, 427
very busy expression
expresión muy ocupada, 731, 732
viable prefix
prefijo viable, 206, 223, 230, 231, 236-38
void type
tipo vacío, 357; 363

weat precedence
precedencia débil, 285
while statement
proposición **while**, 505-07, 518-20
white space
espacio en blanco, 56, 86, 87, 101
word
palabra, 94

yacc
YACC, 264-74, 749, 754, 760
yield
producción, 29

COMPILADORES

Principios, Técnicas y Herramientas

A cualquier persona interesada en el diseño de compiladores le resulta familiar el libro del dragón, Principles of Compiler, de Alfred V. Aho y Jeffrey D. Ullman. El libro del dragón fue un texto memorable sobre el diseño de compiladores, pero este campo ha evolucionado con rapidez y se encuentra en un estado muy avanzado respecto al que tenía cuando se publicó el libro.

El texto comienza con una introducción de las ideas principales que subyacen al proceso de la compilación y posteriormente ilustra esas ideas construyendo un compilador sencillo de una pasada. El resto del libro amplía los conceptos presentados en los dos primeros capítulos y trata temas más avanzados como el análisis sintáctico, la verificación de tipos y la generación y optimización de código.

El nuevo "libro del dragón" presenta material básico que caracterizó al antiguo, pero también profundiza en los avances recientes de esta área. Entre las características de esta nueva versión se encuentran:

* **Aspectos prácticos en el desarrollo de compiladores.**
* **Más material sobre la traducción dirigida por sintaxis, la verificación de tipos, la organización durante la ejecución, la generación automática de código y la optimización del código.**

AHO - SETHI - ULLMAN

dragon book