COP 5555 - PRINCIPLES OF PROGRAMMING LANGUAGES

NOTES ON ATTRIBUTE GRAMMARS.

I. Introduction.

These notes are intended to supplement class lectures on attribute grammars. We will begin with functional graphs, and describe methods of evaluating them. Then we will introduce attribute grammars, show an example of their use, and proceed to give an attribute grammar for Tiny.

II. Functional graphs.

Definition: A functional graph is a directed graph in which

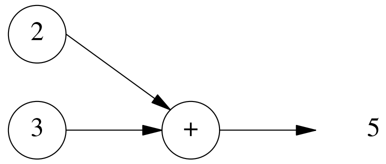
i) Nodes represent functions,

  a. Incoming edges are parameters.

  b. Outgoing edges represent functional values.

ii) Edges represent transmission of data among functions.

Example:



Here we have three functions:

i- Constant function 2.

ii- Constant function 3.

iii- Binary function + (addition).

After some delay, the value of the addition function is five.

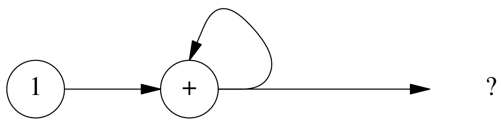
Question:

What effect does a cycle in a functional graph have?

Answer:

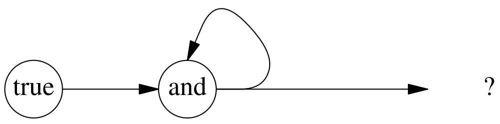
A cycle can make sense only if the graph can achieve a steady state, in which no more changes in the values occur.

Example:



No steady state is achieved, because the value of ? keeps incrementing.

Example:



A steady state is achieved, because regardless of the initial value of the feedback loop, it does not change after the first AND operation, and remains with that value. However, if the "AND" were changed to a "NAND", there would be no steady state.

In general, it is impossible to detect whether a steady state will ever occur (the problem is undecidable, i.e. as difficult as deciding whether a Turing machine will halt).

If we assume that each node in the graph is a single processor, then the graph can be thought of as a network of parallel processors. We will also assume that all functional graphs are acyclic.

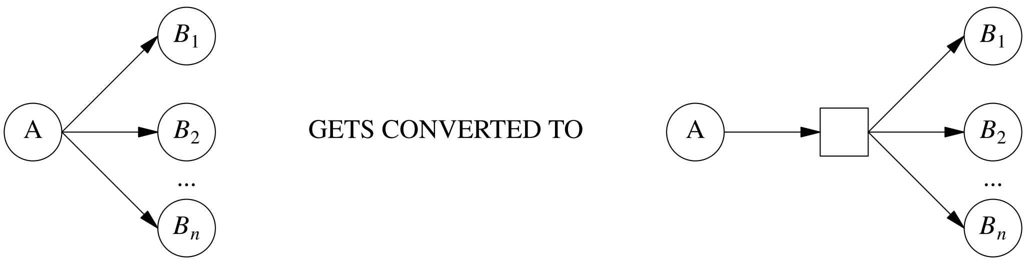
Evaluation of Functional Graphs:

Functional graphs are evaluated by first inserting registers, and then propagating values along the edges.

Register insertion:

Given edges $E_1 \ldots En$ from a node A to nodes $B_1 \ldots B_n$, we insert a register so that there is one edge from A to the register, and n edges from the register to $B_1 \ldots B_n$ respectively.

Example:



All registers must be initialized to some "undefined" value, which must be outside of the domains of all functions.

For an outgoing edge from a "top-most" node, a register is simply added to the graph, so that the edge goes from the node to the register. Note that this is the same as the first case, with n=0.

Functional value propagation:

There are two ways to propagate values along the functional graph. The first, called "data flow analysis", makes several passes on the graph, evaluating functions whose inputs (registers) are defined. The other, called "lazy evaluation", uses a stack to perform a depth-first search, backwards (i.e. the opposite direction of the edges), for the "bottom-most" nodes of the graph. The functions are then evaluated in the order specified by the contents of the stack.

Data Flow Driver:

```
while any top-most register is undefined do
    for each node N in the graph do
        if all inputs of N come from a defined register then
            evaluate function and update N's output register
        fi
    od
od
```

Lazy Evaluation:

```
for each top-most register R do
    push (stack, R)
od
while stack not empty do
    current := top (stack)
    if current = undefined
        then
            computable := true
            for each R in dependency (current), while computable do
                if R = undefined then
                    computable := false
                    push (stack,R)
                fi
            od
            if computable then
                compute_value(current)
                pop_stack
            fi
        else pop (stack)
    fi
od
```

Data Flow Analysis starts at constants and propagate values forward. There is no stack. The algorithm computes ALL values, whether they are needed or not. Lazy evaluation starts at the target nodes and chases the dependencies backwards. The algorithm evaluates functions ONLY if they are needed. It is more expensive in storage, because of the use of a stack.

III. Attribute grammars.

Attribute grammars are used to associate constructs in an AST with segments of a functional graph. An attribute grammar is essentially a context-free grammar, in which each rule has been augmented with a set of axioms. The axioms specify segments of the functional graph. As the AST is built (typically bottom-up), the segments of the functional graph are "pasted" together. Upon completion of the AST, the functional graph's construction also concludes, and one proceeds to evaluate it, using one of the two methods described above. After the evaluation, the top-most register(s) of the functional graph will presumably contain the output of the translation process.

Definition: An attribute grammar consists of

1) A context-free grammar describing the structure of the parse tree.

2) A set of attributes ATT. Each attribute "decorates" some node in the AST, and later in fact becomes a register in the functional graph.

3) A set of axioms that express relationships among attributes.

Example: Binary Numbers.

String-to-tree transduction grammar:

```
S → N        => .
  → N . N    => .
N → N D      => cat
  → D
D → 0        => 0
D → 1        => 1
```

Abstract Syntax Tree Grammar:

```
S → < . N N >
  → < . N >
N → < cat N D >
  → D
D → 0
  → 1
```

The first grammar specifies the transduction from strings to AST's. The second one (which we will use for the attribute grammar) generates trees in prefix form. There are two types of attributes:

1) Synthesized: used to pass information from the sibling to the parent node in the tree.

2) Inherited: used to pass information from the parent to the sibling in the tree.

In general, the type of attribute defines the type of tree walking required (bottom-up or top-down). If the tree is processed recursively, via say, a single procedure called ProcessNode, inherited attributes are input parameters to ProcessNode (to be used by it), whereas synthesized attributes are output parameters of ProcessNode (i.e. to be produced by it).

For binary numbers, we have ATT = { value, length, exp }, where

value: the decimal value of the binary number denoted by the subtree.

length: the number of binary digits occurring to the left of the right-most binary digit in the subtree. This will be used to generate negative exponents, for the fractional part (if any) of the binary number.

exp: the exponent (of 2) that is to be multiplied by the right-most binary digit in the subtree.

The synthesized and inherited attributes are specified by two subsets of ATT:

SATT = { value, length }
IATT = { exp }

Note: In this particular case, SATT and IATT are disjoint. This is not always the case, as we shall see later.

Attributes are associated with nodes in the AST via two functions:

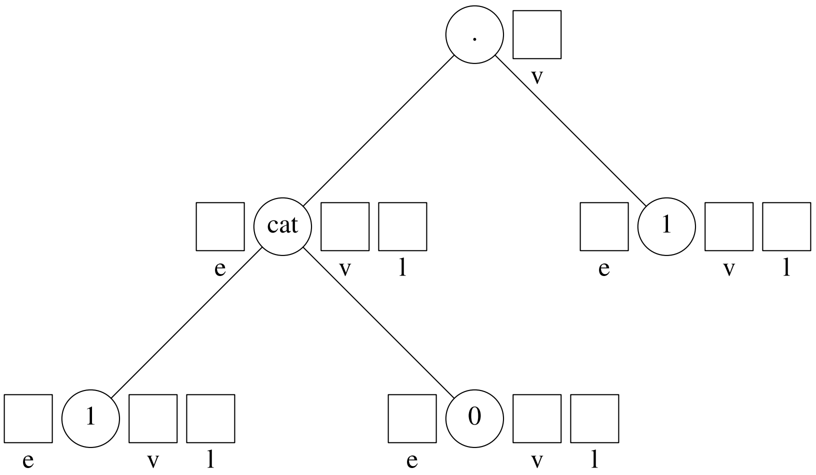
$$S: \Sigma \rightarrow \text{PowerSet (SATT)}$$
$$I: \Sigma \rightarrow \text{PowerSet (IATT)}$$

Σ is the vocabulary in the grammar. In our case, Σ = { 0, 1, cat, . }. Note that meta-symbols '<' and '>' are left out of Σ. Functions S and I map each symbol to a subset of the synthesized and inherited attributes, respectively. In our case,

$$\begin{aligned} S(0) &= \{ \text{value, length} \} \\ S(1) &= \{ \text{value, length} \} \\ S(\text{cat}) &= \{ \text{value, length} \} \\ S(.) &= \{ \text{value} \} \\ I(0) &= \{ \text{exp} \} \\ I(1) &= \{ \text{exp} \} \\ I(\text{cat}) &= \{ \text{exp} \} \\ I(.) &= \{ \} \end{aligned}$$

Recall that attributes are "attached" to tree nodes, and that they become registers in the functional graph.

Example: Suppose the input binary number is '10.1'. Here's the AST, with the corresponding attributes attached:



The inherited attributes are depicted on the left of each node, and the synthesized attributes are depicted on the right. This is simply a convention. However, it aids in understanding the flow of information: top-down on the left, and bottom-up on the right.

For a given tree node, a certain attribute may occur both at that node and at some of its siblings. In the above example, attribute "value" occurs "locally" at the root node and at both its first and second kid. To prevent confusion, we will use a tree addressing scheme, as follows:

Definition: Given a tree node T, with kids $T_{1,} \ldots T_n$,

i- a(ε) refers to attribute "a" at node T, and

ii- a(i) refers to attribute "a" at the i'th kid of T (i.e. node T_i).

In the above example, given that we are referring to the root node, v(ε) refers to the "value" attribute at the root node, while v(1) and v(2) refer to the "value" attribute at the first and second kids of the root, respectively.

All that is left is to specify the functions that connect the registers in the functional graph. This is done with the axioms.

Axiom Specification:

Definition: For each production of the form A $\rightarrow$ < r $K_1 \ldots K_n$ > in the AST grammar, a set of axioms is specified as follows:

Let $r_1, \ldots, r_n$ be the roots of subtrees $K_1, \ldots, K_n$.

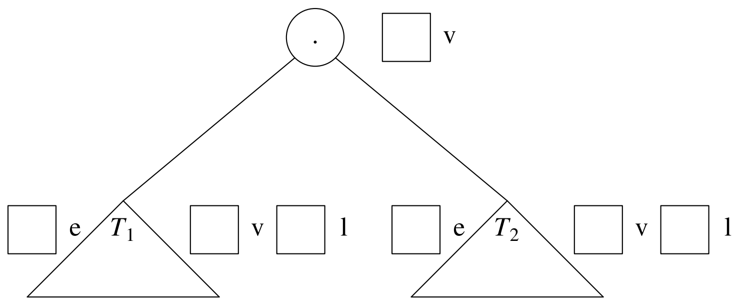
(1) For each attribute 'a' in S(r),
(2) for each sibling r_k of r, and for each attribute 'a' in I(r_k),
(3) there is exactly one axiom of the form
(4) a = f ($w_1, w_2, \ldots, w_m$), where
(5) i- f is a pre-defined semantic function
(6) ii- for all $1 \le i \le m$, either
(7) w_i = z(ε), with z in I(r), or
(8) w_i = z(j), with z in S(r_j), for some j.

Let's make some sense out of this. There are two separate issues here. The first issue is the number of axioms required. There must be exactly one axiom for each synthesized attribute at the root (line 1). There must also be exactly one axiom for each inherited attribute at each of the root's kids (line 2). Another way of phrasing this is to say that we (i.e. the attribute grammar writers) must specify what goes up from the current node (line 1), and what goes down to each kid of the current node (line 2).

The second issue is the form of each of the required axioms. Every axiom is of the form shown (line 4), i.e. a single predefined semantic function (line 5), all of whose arguments (line 6), must come from either inherited attributes at the root (line 7), or synthesized attributes at one of the kids (line 8). It would be most illustrative for the reader to draw some pictures, to understand the restrictions that these rules impose. For example, one may not inherit an attribute from two levels above in the tree, nor synthesize from two levels below. Further, one may not inherit from inherited attributes at the same level in the tree, nor may one synthesize from synthesized attributes at the same level in the tree.

The Attribute Grammar for Binary Numbers:

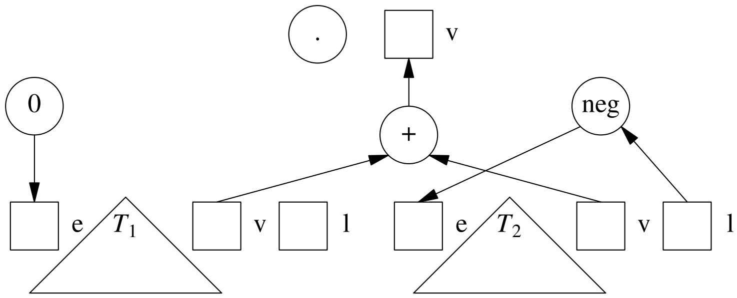
Let's begin with the first production in the AST grammar: S $\rightarrow$ <.NN>. The general form of the tree is shown below. Each subtree may have either 'cat', '0', or '1' as its root. In all three cases, the attributes for the root of the subtree are the same. According to the axiom specification rules stated above, we need three axioms for this production: one for value(ε), one for exp(1), and one for exp(2).



The axioms are:

$$
\begin{aligned}
\text{value}(\varepsilon) &= \text{value}(1) + \text{value}(2) \\
\text{exp}(1) &= 0 \\
\text{exp}(2) &= -\ \text{length}(2)
\end{aligned}
$$

The segment of functional graph is shown below. The lines connecting trees nodes have been omitted.

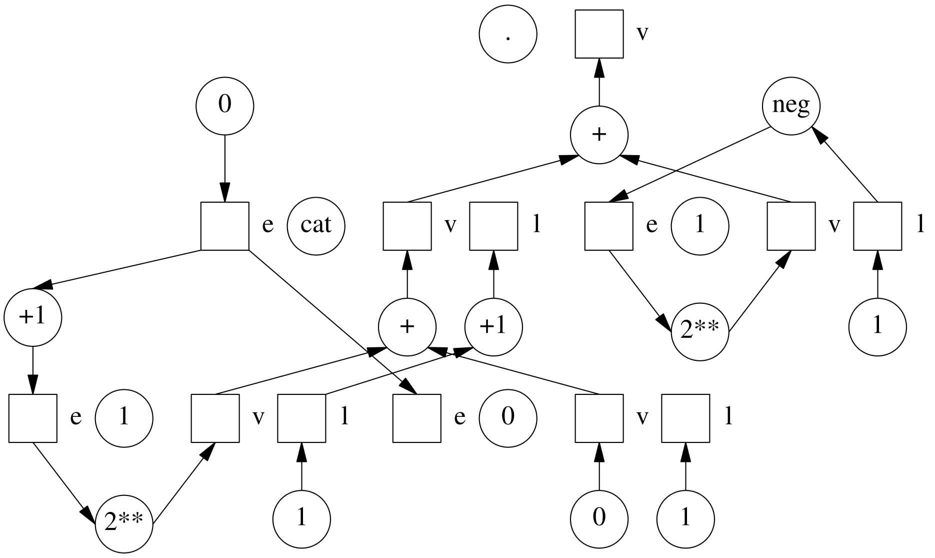


Note that the length attribute from the first kid is ignored, while the length from the second kid is negated and sent down the exponent of the second kid. The length attribute is used only for the second kid of the root (if any), so that negative exponents may be computed. At the bottom of T_2 (the right-hand side tree), length will originate with value 1. It will be incremented by one on the way up, negated at the top, and sent down via the inherited exponent attribute (and incremented along the way), to be used as the correct (negative) power of 2, for the binary digits. The two synthesized "value" attributes (the integral and the fractional part of the binary number) are added together to form the decimal result.

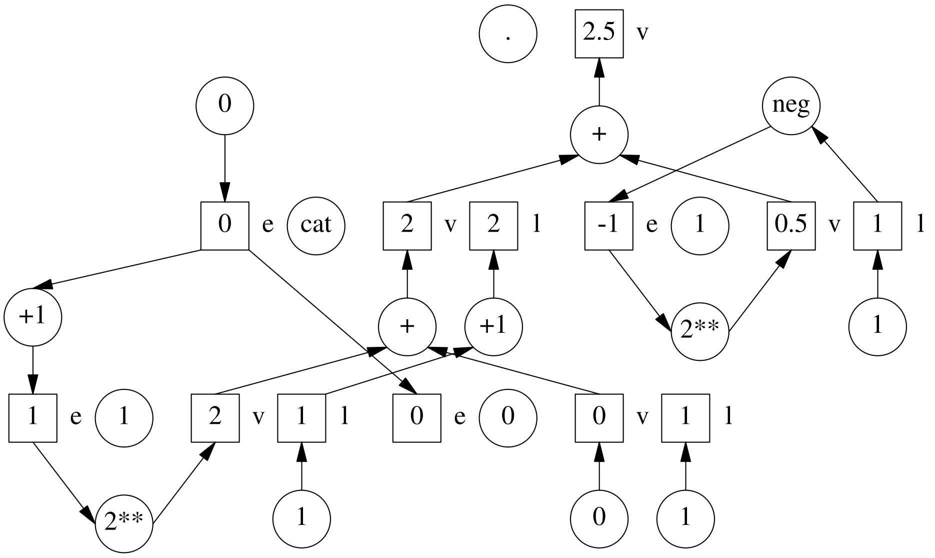
Here's the entire set of axioms:

S → < . N N >		
	value(ε) = value(1) + value(2)	
	exp(1) = 0	
	exp(2) = - length(2)	# explained above
S → < . N >		
	value(ε) = value(1)	# no fraction; copy value up.
	exp(1) = 0	
N → < cat N D >		
	value(ε) = value(1) + value (2)	# add two values.
	length(ε) = length(1) + 1	# increment length up left.
	exp(1) = exp (ε) + 1	# increment exp down left
	exp(2) = exp (ε)	# copy exp down right
N → D		
		# No axioms !
D → 0		
	value(ε) = 0	# zero no matter what.
	length(ε) = 1	# initial length.
D → 1		
	value(ε) = 2 ** exp(ε)	# compute value.
	length(ε) = 1	# initial length.

Here's a complete example, for input "10.1".



Once the functional graph is evaluated:



GENERATION OF FUNCTIONAL GRAPHS.

So far we've seen how a human would build a functional graph from an attribute grammar. Here we describe how functional graphs can be generated automatically from the attribute grammar. The starting point is an AST, produced by a string-to-tree transducer. To guarantee that the tree will be recognized, we must first transform the tree to fully-parenthesized postfix notation, and use an LR parser to recognize it. Fortunately, an LR(0) parser (that's right, no look-ahead) can always do the job. In the case of binary numbers:

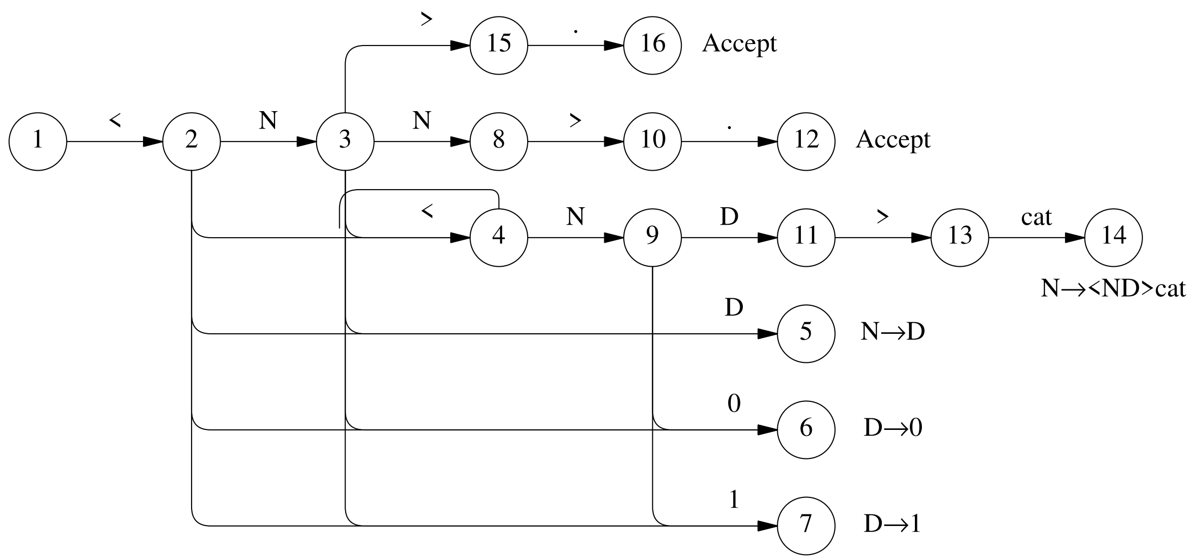
Before:

$$
\begin{array}{lll}
S & \rightarrow & <\ .\ N\ N\ > \\
  & \rightarrow & <\ .\ N\ > \\
N & \rightarrow & <\ cat\ N\ D\ > \\
  & \rightarrow & D \\
D & \rightarrow & 0 \\
  & \rightarrow & 1
\end{array}
$$

After:

$$
\begin{array}{lll}
S & \rightarrow & <\ N\ N\ >\ . \\
  & \rightarrow & <\ N\ >\ . \\
N & \rightarrow & <\ N\ D\ >\ cat \\
  & \rightarrow & D \\
D & \rightarrow & 0 \\
  & \rightarrow & 1
\end{array}
$$

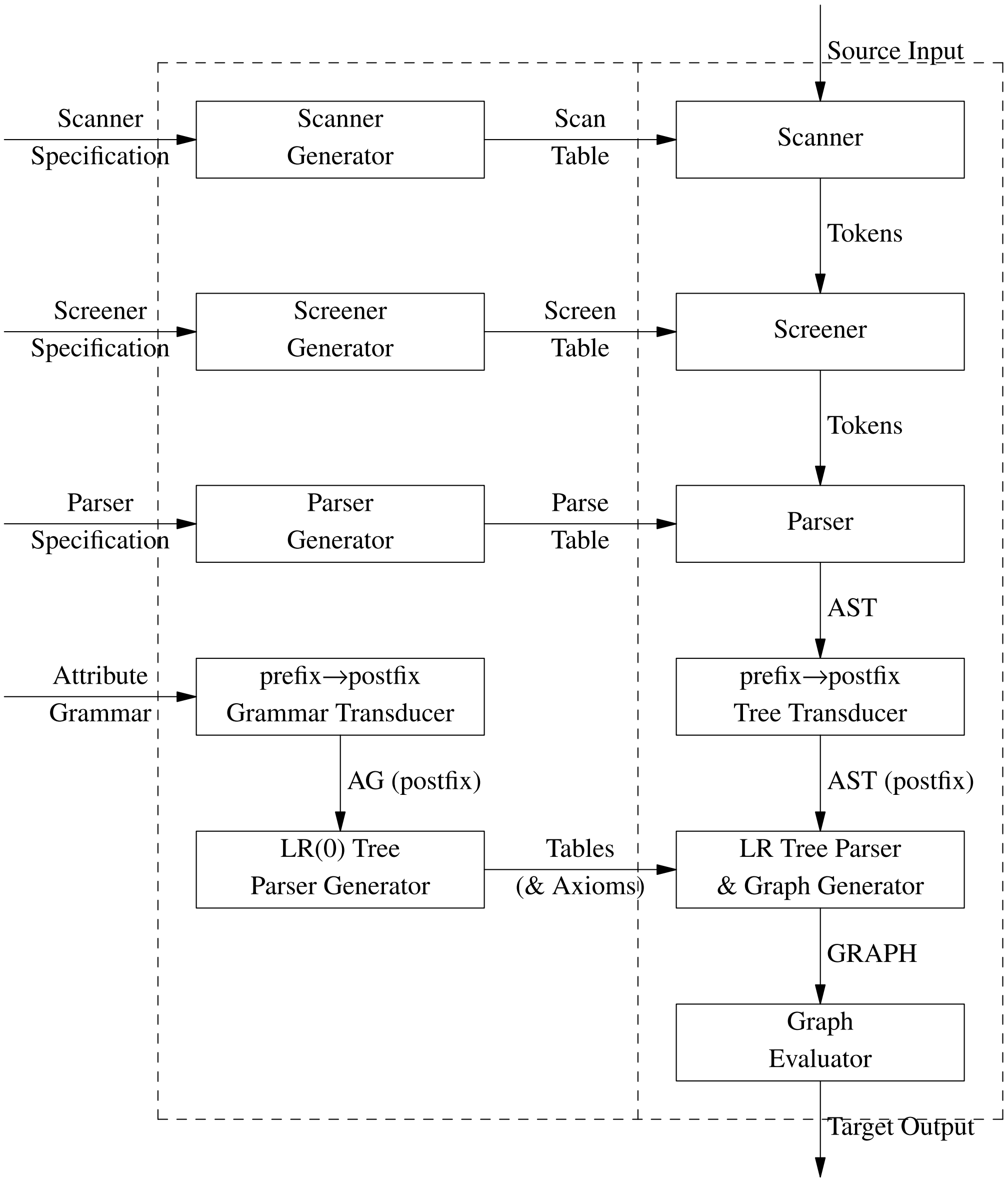
An LR(0) parser can now be built:



With this LR(0) parser, the tree is recognized bottom-up. Each time a reduction is performed on a given production, a segment of the functional graph is built, and is connected to those segments built recursively for the subtrees already recognized.

The Architecture of an Attribute Grammar System.

Illustrated below is the architecture of an attribute grammar-based translator-generation system.



The larger dashed box is the translator genration system; the smaller is the translator being generated. The phases of the translation (scanning, screening, parsing, and back-end processing) are shown. In the case of binary numbers, the user of the system provides the three front-end specifications and the attribute grammar. The system generates a parser from the front-end specifications, and a graph generator from the attribute grammar. The user then provides a sample source input, which is transduced to the AST, transformed into postfix notation, and recognized by the LR(0) tree parser, which in turn builds the functional graph, and passes it to the graph evaluator to produce the translated output. In our case the source language is that of binary numbers, and the target language is that of decimal numbers. The correctness of the translation, of course, depends on ALL four specifications.

Now, finally, the real world. In the case of binary numbers, all attributes were integers. However, attributes can be whatever we want them to be, as long as we have well-defined semantic functions to manipulate them. In the next section, we use attribute grammars to specify (completely) the semantics of a (very) small language, called Tiny for lack of a better name. In this attribute grammar, the most important attribute will be a "file" attribute -- the code file in which instructions of the target machine will be placed. This code file attribute will propagate around the entire abstract syntax tree, collecting instructions as the evaluation of the graph (and the translation) proceeds.

The result of the evaluation of the functional graph will be this code file, i.e. the sequence of machine instructions to which the source program has been translated. To execute the translated program, of course, the code file must be loaded into the target machine and executed.

An Attribute Grammar for Tiny

Tiny is a very small Pascal-like language, with the following characteristics:

1) Every program is given a name.

2) There are user-defined variables, of type integer.

3) Legal expressions are composed of variables, integers, the intrinsic function "read", boolean operator "not", equality operator "=", binary operators "+" and "-" and unary "-". "not" and unary "-' are the most binding operators, "+" and "-" are next, and "=" is the least binding. "+" and "-" are left associative; parentheses override both precedence and associativity.

4) There are assignment statements, and variables must be assigned a value before they are used in expressions. There are no declarations.

5) There is an if-then-else construct, a while-loop, statement sequencing, and an intrinsic procedure "output".

Here's Tiny's syntax:

```
Tiny        →  'program' Name ':' Statement 'end' Name '.'          => 'program';
Statement   →  'assign' Name ':=' Expression                       => 'assign'
            →  'output' Expression                                 => 'output'
            →  'if' Expression 'then' Statement 'else' Statement 'fi'  => 'if'
            →  'while' Expression 'do' Statement 'od'              => 'while'
            →  Statement list ';'                                  => ';' ;
Expression  →  Term '=' Term                                       => '='
            →  Term;
Term        →  Term '+' Factor                                     => '+'
            →  Term '-' Factor                                     => '-'
            →  Factor;
Factor      →  '-' Factor                                          => '-'
            →  'not' Factor                                        => 'not'
            →  Name
            →  'read'                                              => 'read'
            →  '<integer>'
            →  '(' Expression ')';
Name        →  '<identifier>';
```

Here's a sample Tiny program, that copies 10 integers from the input to the output:

```
program copy:
        assign i := 1;
        while not (i=11) do
                output read;
                assign i := i + 1
        od
end copy.
```

Here's the AST grammar for Tiny:

```
P  → < 'program' '<identifier>' E '<identifier>' >
E  → < 'assign' '<identifier>' E >
   → < 'output' E >
   → < 'if' E E E >
   → < 'while' E E >
   → < ';' E* >
   → < 'not' E >
   → < '=' E E >
   → < '+' E E >
   → < '-' E E >
   → < '-' E >
   → '<identifier>'
   → '<integer>'
   → 'read'
```

Notice that the grammar has been simplified. There is no distinction between expressions and statements. The AST grammar generates a superset of the AST's generated by the phrase-structure grammar above. This is in fact a healthy practice. In general it is a good idea to write a compiler back-end (or in our case an attribute grammar that specifies the compiler back-end) that is capable of handling more trees than the parser (as it stands now) will ever give it. If the concrete syntax of the language ever changes, the back-end will be general enough (we hope) to handle the changes without a major upheaval.

The compiler for Tiny generates code for a very simple (again) target machine, whose instruction set is described by the algorithm below:

```
Algorithm Tiny Target Machine:
    I := 1
    Next_Instruction:
        case Code[I] of

            save n:     stack[n] := stack[top--]
            load n:     stack[++top] := stack[n]
            negate:     stack[top] := -stack[top]
            not:        stack[top] := not stack[top]
            add:        t := stack[top--]; stack[top] := stack[top] + t
            subtract:   t := stack[top--]; stack[top] := stack[top] - t
            equal:      t := stack[top--]; stack[top] := stack[top] = t
            read:       stack[++top] := getinteger(input)
            print:      putinteger(stack[top--])
            lit n:      stack[++top] := n
            goto n:     I := n; goto Next_Instruction
            iffalse n:  if stack[top--] = 0 then I:=n; goto Next_Instruction fi
            iftrue n:   if stack[top--] = 1 then I:=n; goto Next_Instruction fi
            stop:       halt

    end;
    ++I; goto Next_Instruction;
```

For the sample Tiny program shown above, the target code is as follows:

```
1:   lit 1          # i := 1
2:   load 1         # i = 11
3:   lit 11
4:   equal
5:   not
6:   iffalse 14     # while
7:   read
8:   print
9:   load 1         # i := i + 1
10:  lit 1
11:  add
12:  save 1
13:  goto 2         # od
14:  stop           # end
```

Our compiler is not ambitious. It recognizes the following semantic errors:

1) Variable used before being initialized.
2) Non-boolean expression in "while", "not", or "if".
3) Non-integer expression in "=", "-", or "+".
4) Program names do not match.

Data structures required:

We will need two data structures: the declaration table and files.

1) **The Declaration Table:** We will assume the following semantic functions:
   a) function enter(name,l). Associates the given "name" with stack location "l". Returns location "l".
   b) function lookup(name). Returns the location previously "entered" for "name". Returns 0 if "name" is not found.
2) **Files**: We will assume the following semantic functions:
   a) function gen(file, $arg_1, \ldots arg_n$). Writes a new line on the given "file". The line contains $arg_1, \ldots arg_n$. The function returns the new, modified file.
   b) function Open. Creates a new file.
   c) function Close. Closes a file.

Note that we assume that files are generated sequentially.

The Attributes:

code: File of code generated.
error: File of semantic errors.
next: Label of the next instruction to be written on the code file.
top: Current size of run-time stack.
type: Type of subtree. Used for type-checking.

ALL attributes are both synthesized and inherited. Hence, to prevent confusion, we will use the following convention:

a↑ is the synthesized attribute a.
a↓ is the inherited attribute a.

Synthesized and Inherited Attributes:

$$
\begin{aligned}
S(\text{program}) &= \{\ \text{code}\uparrow,\ \text{error}\uparrow\ \} \\
I(\text{program}) &= \{\ \} \\
S(\text{assign}) &= \{\ \text{code}\uparrow,\ \text{error}\uparrow,\ \text{next}\uparrow,\ \text{top}\uparrow,\ \text{type}\uparrow\ \} \\
I(\text{assign}) &= \{\ \text{code}\downarrow,\ \text{error}\downarrow,\ \text{next}\downarrow,\ \text{top}\downarrow\ \}
\end{aligned}
$$

All other nodes have the same synthesized and inherited attributes as the "assign" node.

Tree Walking Assumptions:

In general, the tree walk will be as discussed before, i.e. top-down on the left, and bottom-up on the right. On the left of each tree node we will have four inherited attributes (all except type↓). On the right of each tree node we will have all five synthesized attributes. The only exception is the "program" node, which inherits no attributes and only synthesizes the final results of the translation: the code and error files. It should be clear that there are a lot of axioms to write, many of which will be repetitive. Hence the following convention:

Whenever an axiom is omitted, we will assume the following:

If the tree node has no kids, then

$$a\uparrow(\varepsilon) = a\downarrow(\varepsilon)$$

If the tree node has "n" kids, then

$$
\begin{aligned}
a\downarrow(1) &= a\downarrow(\varepsilon) \\
a\downarrow(i) &= a\uparrow(i\text{-}1), \text{ for } 1 < i \le n \\
a\uparrow(\varepsilon) &= a\uparrow(n)
\end{aligned}
$$

In short, when axioms are omitted, we assume a "pipeline" that simply passes the attributes along, beginning with the inherited attribute at the the root of the subtree, down to the first kid, up from the first kid, down to the second kid, etc., and finally up from the last kid and up from the root. This effectively means that we are making extensive (and implicit) use of the identity function.

TINY's ATTRIBUTE GRAMMAR:

Here's the attribute grammar, in all its gory detail.

E → '<integer>:n'

code↑(ε) = gen (code↓(ε), "lit", "n")
next↑(ε) = next↓(ε) + 1
top↑(ε) = top↓(ε) + 1
type↑(ε) = "integer"

E → '<identifier>:x'

code↑(ε) = gen (code↓(ε), "load", lookup("x"))
next↑(ε) = next↓(ε) + 1
top↑(ε) = top↓(ε) + 1
type↑(ε) = "integer"
error↑(ε) = if lookup("x") = 0
    then gen (error↓(ε), "identifier un-initialized")
    else error↓(ε)

E $\rightarrow$ read

code$\uparrow(\varepsilon)$ = gen (code$\downarrow(\varepsilon)$, "read")
next$\uparrow(\varepsilon)$ = next$\downarrow(\varepsilon)$ + 1
top$\uparrow(\varepsilon)$ = top$\downarrow(\varepsilon)$ + 1
type$\uparrow(\varepsilon)$ = "integer"

E $\rightarrow$ < - E >

code$\uparrow(\varepsilon)$ = gen (code$\uparrow(1)$, "negate")
next$\uparrow(\varepsilon)$ = next$\uparrow(1)$ + 1
type$\uparrow(\varepsilon)$ = "integer"
error$\uparrow(\varepsilon)$ = if type$\uparrow(1)$ = "integer"
    then error $\uparrow(1)$
    else gen (error$\uparrow(1)$, "Illegal type for minus")

E $\rightarrow$ < + E E >

code$\uparrow(\varepsilon)$ = gen (code$\uparrow(2)$, "add")
next$\uparrow(\varepsilon)$ = next$\uparrow(2)$ + 1
top$\uparrow(\varepsilon)$ = top$\uparrow(2)$ - 1
type$\uparrow(\varepsilon)$ = "integer"
error$\uparrow(\varepsilon)$ = if type$\uparrow(1)$ = type$\uparrow(2)$ = "integer"
    then error$\uparrow(2)$
    else gen (error$\uparrow(2)$, "Illegal type for plus")

E $\rightarrow$ < - E E >

code$\uparrow(\varepsilon)$ = gen (code$\uparrow(2)$, "subtract")
next$\uparrow(\varepsilon)$ = next$\uparrow(2)$ + 1
top$\uparrow(\varepsilon)$ = top$\uparrow(2)$ - 1
type$\uparrow(\varepsilon)$ = "integer"
error$\uparrow(\varepsilon)$ = if type$\uparrow(1)$ = type$\uparrow(2)$ = "integer"
    then error$\uparrow(2)$
    else gen (error$\uparrow(2)$, "Illegal type for minus")

E $\rightarrow$ < not E >

code$\uparrow(\varepsilon)$ = gen (code$\uparrow(1)$, "not")
next$\uparrow(\varepsilon)$ = next$\uparrow(1)$ + 1
type$\uparrow(\varepsilon)$ = "boolean"
error$\uparrow(\varepsilon)$ = if type$\uparrow(1)$ = "boolean"
    then error$\uparrow(1)$
    else gen (error$\uparrow(1)$, "Illegal type for not")

E $\rightarrow$ < = E E >

code$\uparrow(\varepsilon)$ = gen (code$\uparrow(2)$, "equal")
next$\uparrow(\varepsilon)$ = next$\uparrow(2)$ + 1
type$\uparrow(\varepsilon)$ = "boolean"
top$\uparrow(\varepsilon)$ = top$\uparrow(2)$ - 1
error$\uparrow(\varepsilon)$ = if type$\uparrow(1)$ = type$\uparrow(2)$
    then error$\uparrow(2)$
    else gen (error$\uparrow(2)$, "Type clash in equal comparison")

```
E → < assign '<identifier>:x' E >
    code↑(ε) = if lookup("x") = 0
        then enter("x",top↑(2)); code↑(2)
        else gen ( code↑(2), "save", lookup("x") )
    next↑(ε) = if lookup("x") = 0
        then next↑(2)
        else next↑(2) + 1
    top↑(ε) = if lookup ("x") = 0
        then top↑(2)
        else top↑(2) - 1
    error↑(ε) = if type↑(2) = "integer"
        then error↑(2)
        else gen ( error↑(2), "Assignment type clash" )
    type↑(ε) = "statement"
E → < output E >
    code↑(ε) = gen ( code↑(1), "print" )
    next↑(ε) = next↑(1) + 1
    top↑(ε) = top↑(1) - 1
    type↑(ε) = "statement"
    error↑(ε) = if type↑(1) = "integer"
        then error↑(1)
        else gen ( error↑(1), "Illegal type for output" )
E → < ; E* >
    Use Defaults !
E → < if E E E >
    code↓(2) = gen ( code↑(1), "iffalse", next↑(2) + 1 )
    next↓(2) = next↑(1) + 1
    top↓(2) = top↑(1) - 1
    code↓(3) = gen ( code↑(2), "goto", next↑(3) )
    next↓(3) = next↑(2) + 1
    error↓(2) = if type↑(1) = "boolean"
        then error↑(1)
        else gen ( error↑(1), "Illegal expression for if" )
    error↓(3) = if type↑(2) = "statement"
        then error↑(2)
        else gen ( error↑(2), "Statement required for if" )
    error↑(ε) = if type↑(3) = "statement"
        then error↑(3)
        else gen ( error↑(3), "Statement required for if" )
E → < while E E >
    code↓(2) = gen ( code↑(1), "iffalse", next↑(2) + 1 )
    next↓(2) = next↑(1) + 1
    top↓(2) = top↑(1) - 1
```

```
    code↑(ε) = gen ( code↑(2), "goto", next↓(ε) )
    next↑(ε) = next↑(2) + 1
    type↑(ε) = "statement"
    error↓(2) = if type↑(1) = "boolean"
            then error↑(1)
            else gen ( error↑(1), "Illegal expression in while" )
    error↑(ε) = if type↑(2) = "statement"
            then error↑(2)
            else gen ( error↑(2), "Statement required in while" )

Tiny → < program '<identifier>:x' E '<identifier>:y' >
    code↓(2) = Open
    error↓(2) = Open
    next↓(2) = 1
    top↓(2) = 0
    code↑(ε) = close ( gen ( code↑(2), "stop" ) )
    error↑(ε) = close (
                if x = y
                    then error↑(2)
                    else gen ( error↑(2), "program names don't match" )
                )
```

It would be useful to draw pictures of the functional graph segments, and maybe even the entire AST and functional graph for the sample Tiny program shown above.