Inicio: 11:06
Final: 12:13

Universidad de Costa Rica

Escuela de Ciencias de la Computación e Informática

CI-2700

TÓPICOS ESPECIALES - COMPILADORES

II Ciclo 2014

Profesor: Manuel E. Bermúdez

EXAMEN PARCIAL I

100 minutos

18 exámenes:
MAX: 98
MIN: 43
PROMEDIO: 75.0

Problema 1 30 (30p.)

Problema 2 20 (20p.)

Problema 3 30 (30p.)

Problema 4 20 (20p.)

TOTAL 100 (100p.)

NOMBRE SOLUCION

NOTA: Favor entregar todo su trabajo en este examen.

PROBLEMA 1. (30 pts.)

Agregar la instrucción **`for`** del lenguaje C a la implementación de Tiny. La instrucción **`for`** en C tiene la siguiente forma:

```
for ( Inicio ; Expresion ; Incremento ) S
```

`Inicio`, **`Incremento`** y **`S`** son instrucciones (Statement); **`Expresion`** es una expresión booleana.

IMPORTANTE: **`Inicio`**, **`Expresion`**, e **`Incremento`** son TODOS opcionales, pero los **`";"`** que los separan son obligatorios.

Un ejemplo de la instrucción **`for`** en Tiny:

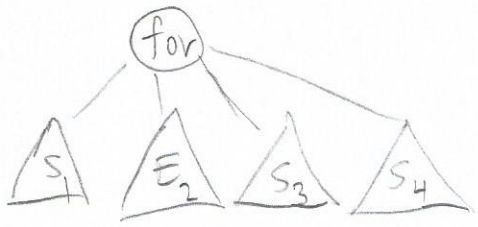
```
for (i:=1; i <= 10; i:=i+1) output(i)
```

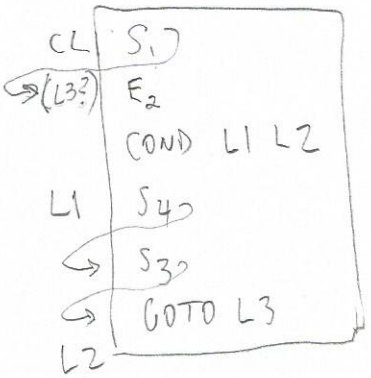
(a) Haga un dibujo de la sintaxis abstracta de la instrucción **`for`**.

(b) Haga un dibujo del código generado por esta instrucción.

(c) En el Apéndice encontrará los cuatro archivos originales, conocidos como los cuatro jinetes del Apocalipsis de Tiny: **`lex.tiny`**, **`parse.tiny`**, **`Constrainer.c`**, y **`CodeGenerator.c`**. Marque los listados con todos los cambios necesarios para la implementación de la instrucción **`for`**.

(a)





ESPACIO DE TRABAJO

PROBLEMA 2. (20 pts.)

Abajo está la especificación de análisis sintáctico de un lenguaje de expresiones regulares, escrito en notación del TWS (i.e. esto de hecho funciona!)

```
%%
Expression -> Term ('|' Term)+          => "|"
           -> Term;
Term       -> List List+                => "cat"
           -> List;
List       -> Factor 'list'  List       => "list"
           -> Factor;
Factor     -> Primary '*'               => "*"
           -> Primary '+'               => "+"
           -> Primary '?'               => "?"
           -> Primary;
Primary    -> IDENTIFIER                => "<identifier>"
           -> '(' Expression ')';
```

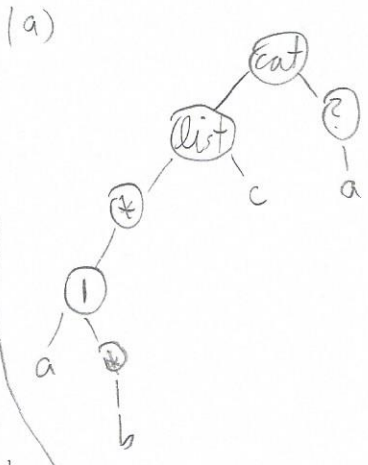
El operador de alternación es |. El operador de "concatenación" es vacío. + es el operador unario postfijo que indica "una o más instancias del operando. * es el operador unario postfijo que indica "cero o más" instancias del operando. ? es el operador unario postfijo que indica que el operando es opcional. **list** es el operador binario infijo que indica una "lista de **a**'s separadas por **b**'s", donde **a** es el operando izquierdo, y **b** es el operando derecho.

Considere la expresión regular **(a | b*)* list c a?**.

a) Dibuje el árbol que produciría el TWS para esta expresión.

b) Escriba la derivación de la expresión, a partir de Expression, completando esta secuencia:

```
Expression => Term
           => List List
           ...
           =>  (a | b*)* list c IDENTIFIER ?
           =>  (a | b*)* list c a ?
```



⇒ F list L L
⇒ P * list L L
⇒ (E) * list L L
⇒ (T | T)* list L L
⇒ (F | T)* list L L
⇒ (P | T) * list L L
⇒ (a | T) * list L L
⇒ (a | L) * list L L

⇒ (a | F) * list L L
⇒ (a | P *) * list L L
⇒ (a | b *) * list L L
⇒ (a | b *) * list F L
⇒ (a | b *) * list P L
⇒ (a | b*) * list c L
⇒ (a | b*) * list c F
⇒ (a | b*) * list c P ?

ESPACIO DE TRABAJO

PROBLEMA 3. (30 pts.)

Agregar la instrucción **`loop-while`** a la implementación de Tiny. La instrucción **`loop-while`** tiene la siguiente forma:

```
loop S while B : T repeat
```

`S` y **`T`** son instrucciones; **`B`** es una expresión booleana. **`S`** se ejecuta al menos una vez. Después de cada ejecución de `S`, la expresión **`B`** se prueba: si es verdadera, se ejecutan **`T`** y **`S`**, y se vuelve a examinar **`B`**; si es falsa, la iteración termina.

Un ejemplo de la instrucción **`loop-while`**:

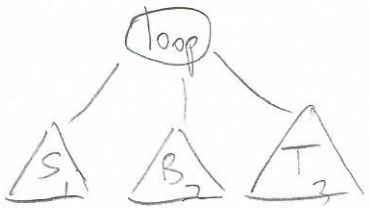
```
loop output(i) while i <= 10 : i:=i+1 repeat
```

(a) Haga un dibujo de la sintaxis abstracta de la instrucción **`loop-while`**.

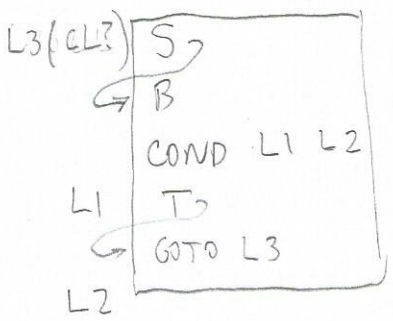
(b) Haga un dibujo del código generado por esta instrucción.

(c) Marque los listados en el Apéndice con todos los cambios necesarios para la implementación de la instrucción **`loop-while`**.

(a)



(b)



PROBLEMA 4. (20 pts.)

Agregar a la implementacón de Tiny (en el Apéndice) las siguientes cuatro construcciones.

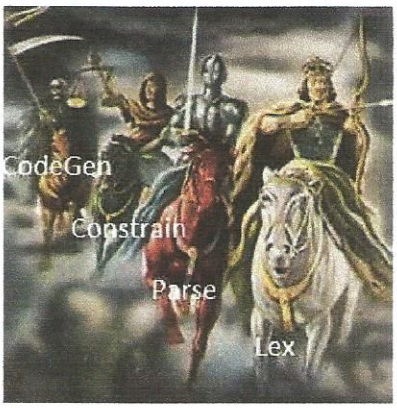
a) El operador prefijo unario de auto-incremento, como una instrucción.
Ejemplo: `++n;`. Favor usar el nombre de nodo de árbol "`pre++`".

b) El operador postfijo unario de auto-incremento, como una instrucción.
Ejemplo: `n++;`. Favor usar el nombre de nodo de árbol "`post++`".

c) El operador prefijo unario de auto-incremento, como una expresión.
Ejemplo: `output(++n);`. Favor usar el nombre de nodo de árbol "`pre++`".

d) El operador postfijo unario de auto-incremento, como una expresión.
Ejemplo: `output(n++);`. Favor usar el nombre de nodo de árbol "`post++`".

APÉNDICE (ARCHIVOS DE TINY)



lex.tiny:

```
%{
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include <Tokenizer.h>
#include "y.tab.h"

static int line = 1;
static int column = 1;
int rule(int token);
int node(int token);
void yyerror(char* message);
int debug_tokenizer = 0;

%}
%x COMM1
COMM2    #.*\n
IDENT    [_a-zA-Z][_a-zA-Z0-9]*
INT      [0-9]+
WHITE    [ \t\v\f]*
LIT      '[^\']+'
STR
%%

{WHITE}       { column += yyleng; }
\n            { column = 1; line++; }
"program"     { return rule(PROGRAM); }
"var"         { return rule(VAR); }
"integer"     { return rule(INTEGER); }
"boolean"     { return rule(BOOLEAN); }
"begin"       { return rule(BEGINX); }
"end"         { return rule(END); }
":="          { return rule(ASSIGNMENT); }
"output"      { return rule(OUTPUT); }
"if"          { return rule(IF); }
"then"        { return rule(THEN); }
"else"        { return rule(ELSE); }
"while"       { return rule(WHILE); }
"do"          { return rule(DO); }
"<="          { return rule(LTE); }
"read"        { return rule(READ); }
{INT}         { return node(INTEGER_NUM); }
{IDENT}       { return node(IDENTIFIER); }
"{"           { column += yyleng; BEGIN( COMM1 ); }
<COMM1>[^}\n]+   { column += yyleng; }
<COMM1>"}"    { column += yyleng; BEGIN( INITIAL ); }
<COMM1>\n     { column = 1; line++; }
{COMM2}       { column = 1; line++; }
":"           { return rule(yytext[0]); }
";"           { return rule(yytext[0]); }
"."           { return rule(yytext[0]); }
","           { return rule(yytext[0]); }
"("           { return rule(yytext[0]); }
")"           { return rule(yytext[0]); }
"+"           { return rule(yytext[0]); }
"-"           { return rule(yytext[0]); }
.             { yyerror("unrecognized char");
                printf("-->%s<--0,yytext);
                column++; }
%%

int rule(int token)
{
```

"for" { return rule (FOR); }
"loop" { return rule (LOOP); }
"repeat" { return rule (REPEAT); }
"++" { return rule (MM); }

```
    if (debug_tokenizer) {
    printf("string:%s, token: %d, line: %d, column: %d0,
          yytext,token,line,column);
    }

    yylval.info.line = line;
    yylval.info.column = column;
    column += yyleng;

    yylval.info.string = malloc(yyleng+1);
    assert(yylval.info.string);
    strcpy(yylval.info.string, yytext);

    yylval.info.makenode = 0;

    return token;
}

int node(int token)
{
    int tok = rule(token);
    yylval.info.makenode = 1;
    return tok;
}

void yyerror(char* message)
{
    printf("****** %s @line %d, column %d0,
       message,line,column);
}
```

parse.tiny:

```
%%
Tiny        -> PROGRAM Name ':' Dclns Body Name '.'     => "program";
Dclns       -> VAR (Dcln ';')*                          => "dclns"
            =>                                          => "dclns";
Dcln        -> Name list ',' ':' Type                   => "dcln";
Type        -> INTEGER                                  => "integer"
            -> BOOLEAN                                  => "boolean";
Body        -> BEGINX Statement list ';' END            => "block";
Statement   -> Name ASSIGNMENT Expression               => "assign"
            -> OUTPUT '(' Expression ')'                => "output"
            -> IF Expression THEN Statement
                          ELSE Statement                => "if"
            -> WHILE Expression DO Statement            => "while"
            -> Body
            ->                                          => "<null>";

Expression  -> Term
            -> Term LTE Term                            => "<=";

Term        -> Primary
            -> Primary '+' Term                         => "+";

Primary     -> '-' Primary                              => "-"
            -> READ                                     => "read"
            -> Name
            -> INTEGER_NUM                              => "<integer>"
            -> '(' Expression ')';

Name        -> IDENTIFIER                               => "<identifier>";
```

→ MM None ⇒ "pre++"
→ None MM ⇒ "post++"

→ FOR '(' Statenet ';' ForExp ';' Statenet ')' Statement ⇒ "for"
→ LOOP Statenet WHILE Expression ';' statenet REPEAT ⇒ "loop"
→ MM None ⇒ "pre ++"
→ None MM ⇒ "post ++"

ForExp → Expression
→ ⇒ "<null>";

Constrainer.c:

```
/*****************************************************************
              Copyright (C) 1986 by Manuel E. Bermudez
              Translated to C - 1991
*****************************************************************/

#include <stdio.h>
#include <header/Open_File.h>
#include <header/CommandLine.h>
#include <header/Table.h>
#include <header/Text.h>
#include <header/Error.h>
#include <header/String_Input.h>
#include <header/Tree.h>
#include <header/Dcln.h>
#include <header/Constrainer.h>

#define ProgramNode     1
#define TypesNode       2
#define TypeNode        3
#define DclnsNode       4
#define DclnNode        5
#define IntegerTNode    6
#define BooleanTNode    7
#define BlockNode       8
#define AssignNode      9
#define OutputNode      10
#define IfNode          11
#define WhileNode       12
#define NullNode        13
#define LENode          14
#define PlusNode        15
#define MinusNode       16
#define ReadNode        17
#define IntegerNode     18
#define IdentifierNode  19

typedef TreeNode UserType;

/*****************************************************************
 Add new node names to the end of the array, keeping in strict
  order with the define statements above, then adjust the i loop
  control variable in InitializeConstrainer().
*****************************************************************/
char *node[] = { "program", "types", "type", "dclns",
                 "dcln", "integer", "boolean", "block",
                 "assign", "output", "if", "while",
                 "<null>", "<=", "+", "-", "read",
                 "<integer>", "<identifier>"
               };

UserType TypeInteger, TypeBoolean;
boolean TraceSpecified;
FILE *TraceFile;
char *TraceFileName;
int NumberTreesRead, index;


void Constrain(void) {
   int i;
   InitializeDeclarationTable();
   Tree_File = Open_File("_TREE", "r");
   NumberTreesRead = Read_Trees();
   fclose (Tree_File);

   AddIntrinsics();

#if 0
   printf("CURRENT TREE0);
   for (i=1;i<=SizeOf(Tree);i++)
      printf("%2d: %d0,i,Element(Tree,i));
#endif

   ProcessNode(RootOfTree(1));


   Tree_File = fopen("_TREE", "w");
   Write_Trees();
   fclose (Tree_File);

   if (TraceSpecified)
      fclose(TraceFile);
}


void InitializeConstrainer (int argc, char *argv[]) {
   int i, j;
```

#define ForNode 20
#define LoopNode 21
#define PrePP 22
#define PosPP 23

, "for", "loop", "pre++",
"post++"

```
   InitializeTextModule();
   InitializeTreeModule();

   for (i=0, j=1; i<19; i++, j++)
      String_Array_To_String_Constant (node[i], j);

   index = System_Flag ("-trace", argc, argv);

   if (index) {
      TraceSpecified = true;
      TraceFileName = System_Argument ("-trace", "_TRACE", argc, argv);
      TraceFile = Open_File(TraceFileName, "w");
   }
   else TraceSpecified = false;
}

void AddIntrinsics (void) {
   TreeNode TempTree;

   AddTree (TypesNode, RootOfTree(1), 2);

   TempTree = Child (RootOfTree(1), 2);
   AddTree (TypeNode, TempTree, 1);

   TempTree = Child (RootOfTree(1), 2);
   AddTree (TypeNode, TempTree, 1);

   TempTree = Child (Child (RootOfTree(1), 2), 1);
   AddTree (BooleanTNode, TempTree, 1);

   TempTree = Child (Child (RootOfTree(1), 2), 2);
   AddTree (IntegerTNode, TempTree, 1);
}


void ErrorHeader (TreeNode T) {
   printf ("<<< CONSTRAINER ERROR >>> AT ");
   Write_String (stdout,SourceLocation(T));
   printf (" : ");
   printf ("0);
}

int NKids (TreeNode T) {
   return (Rank(T));
}


UserType Expression (TreeNode T) {
   UserType Type1, Type2;
   TreeNode Declaration, Temp1, Temp2;
   int NodeCount;

   if (TraceSpecified) {
      fprintf (TraceFile, "<<< CONSTRAINER >>> : Expn Processor Node ");
      Write_String (TraceFile, NodeName(T));
      fprintf (TraceFile, "0);
   }

   switch (NodeName(T)) {
      case LENode :
         Type1 = Expression (Child(T,1));
         Type2 = Expression (Child(T,2));

         if (Type1 != Type2) {
            ErrorHeader(T);
            printf ("ARGUMENTS OF '<=' MUST BE TYPE INTEGER0);
            printf ("0);
         }
         return (TypeBoolean);


      case PlusNode :
      case MinusNode :
         Type1 = Expression (Child(T,1));

         if (Rank(T) == 2)
            Type2 = Expression (Child(T,2));
         else
            Type2 = TypeInteger;

         if (Type1 != TypeInteger || Type2 != TypeInteger) {
            ErrorHeader(T);
            printf ("ARGUMENTS OF '+', '-', '*', '/', mod ");
            printf ("MUST BE TYPE INTEGER0);
            printf ("0);
         }
```

```
      return (TypeInteger);

   case ReadNode :
      return (TypeInteger);

   case IntegerNode :
      return (TypeInteger);

   case IdentifierNode :
      Declaration = Lookup (NodeName(Child(T,1)), T);
      if (Declaration != NullDeclaration)
      {
         Decorate (T, Declaration);
         return (Decoration(Declaration));
      }
      else
         return (TypeInteger);

   default :
      ErrorHeader(T);
      printf ( "UNKNOWN NODE NAME ");
      Write_String (stdout,NodeName(T));
      printf ("0);

} /* end switch */
} /* end Expression */


void ProcessNode (TreeNode T) {
   int Kid, N;
   String Name1, Name2;
   TreeNode Type1, Type2, Type3;

   if (TraceSpecified) {
      fprintf (TraceFile,
               "<<< CONSTRAINER >>> : Stmt Processor Node ");
      Write_String (TraceFile, NodeName(T));
      fprintf (TraceFile, "0);
   }

   switch (NodeName(T)) {
      case ProgramNode :
         OpenScope();
         Name1 = NodeName(Child(Child(T,1),1));
         Name2 = NodeName(Child(Child(T,NKids(T)),1));

         if (Name1 != Name2) {
           ErrorHeader(T);
           printf ("PROGRAM NAMES DO NOT MATCH0);
           printf ("0);
         }

         for (Kid = 2; Kid <= NKids(T)-1; Kid++)
            ProcessNode (Child(T,Kid));
         CloseScope();
         break;


      case TypesNode :
         for (Kid = 1; Kid <= NKids(T); Kid++)
            ProcessNode (Child(T,Kid));
         TypeBoolean = Child(T,1);
         TypeInteger = Child(T,2);
         break;


      case TypeNode :
         DTEnter (NodeName(Child(T,1)),T,T);
         break;


      case DclnsNode :
         for (Kid = 1; Kid <= NKids(T); Kid++)
            ProcessNode (Child(T,Kid));
         break;


      case DclnNode :

         Name1 = NodeName (Child(T, NKids(T)));

         Type1 = Lookup (Name1,T);

         for (Kid  = 1; Kid < NKids(T); Kid++) {
```

NullNode: return Boolean;

PrePP:

PostPP: Type1 = Expression (Child (T,1);
if (Type1 != TypeInteger) {
ErrorHeader(T);
printf ("Auto-inc variable not integer");
printf("\n");
}
return (TypeInteger);

```
        DTEnter (NodeName(Child(Child(T,Kid),1)), Child(T,Kid), T);
        Decorate (Child(T,Kid), Type1);
      }
      break;

   case BlockNode :
      for (Kid = 1; Kid <= NKids(T); Kid++)
         ProcessNode (Child(T,Kid));
      break;

   case AssignNode :
      Type1 = Expression (Child(T,1));
      Type2 = Expression (Child(T,2));

      if (Type1 != Type2) {
         ErrorHeader(T);
         printf ("ASSIGNMENT TYPES DO NOT MATCH0);
         printf ("0);
      }
      break;

   case OutputNode :
      for (Kid = 1; Kid <= NKids(T); Kid++)
         if (Expression (Child(T,Kid)) != TypeInteger) {
            ErrorHeader(T);
            printf ("OUTPUT EXPRESSION MUST BE TYPE INTEGER0);
            printf ("0);
         }
      break;

   case IfNode :
      if (Expression (Child(T,1)) != TypeBoolean) {
         ErrorHeader(T);
         printf ("CONTROL EXPRESSION OF 'IF' STMT");
         printf (" IS NOT TYPE BOOLEAN0);
         printf ("0);
      }

      ProcessNode (Child(T,2));
      if (Rank(T) == 3)
         ProcessNode (Child(T,3));
      break;


   case WhileNode :
      if (Expression (Child(T,1)) != TypeBoolean) {
         ErrorHeader(T);
         printf ("WHILE EXPRESSION NOT OF TYPE BOOLEAN0);
         printf ("0);
      }
      ProcessNode (Child(T,2));
      break;


   case NullNode :
      break;

   default :
      ErrorHeader(T);
      printf ("UNKNOWN NODE NAME ");
      Write_String (stdout,NodeName(T));
      printf ("0);

  } /* end switch */
} /* end ProcessNode */
```

PrePP:

PostPP: if(Expression (Child (T,1)) != TypeInteger) {
    ErrorHeader(T);
    printf ("auto-increment variable not integer")
    printf("\n");
}
break;

LoopNode:
    ProcessNode (Child (T,1));
    if (Expression (Child (T,2)) != TypeBoolean)
    { ErrorHeader(T);
      printf ("loop Exp not Boo")
      printf("\n");
    }
    ProcessNode (Child (T,3);
    break;

ForNode: ProcessNode (Child (T,1));
    if(Expression (Child (T,2)) != TypeBoolean)
    { ErrorHeader(T);
      printf ("for Exp Not Bool");
      printf ("\n");
    ProcessNode (Child (T,3));
    ProcessNode (Child (T,4))
    break;

CodeGenerator.c:

```
/*************************************************************************
          Copyright (C) 1986 by Manuel E. Bermudez
          Translated to C - 1991
*************************************************************************/

#include <stdio.h>
#include <header/CommandLine.h>
#include <header/Open_File.h>
#include <header/Table.h>
#include <header/Text.h>
#include <header/Error.h>
#include <header/String_Input.h>
#include <header/Tree.h>
#include <header/Code.h>
#include <header/CodeGenerator.h>
#define LeftMode 0
#define RightMode 1

    /*  ABSTRACT MACHINE OPERATIONS  */
#define    NOP          1   /* 'NOP'       */
#define    HALTOP       2   /* 'HALT'      */
#define    LITOP        3   /* 'LIT'       */
#define    LLVOP        4   /* 'LLV'       */
#define    LGVOP        5   /* 'LGV'       */
#define    SLVOP        6   /* 'SLV'       */
#define    SGVOP        7   /* 'SGV'       */
#define    LLAOP        8   /* 'LLA'       */
#define    LGAOP        9   /* 'LGA'       */
#define    UOPOP       10   /* 'UOP'       */
#define    BOPOP       11   /* 'BOP'       */
#define    POPOP       12   /* 'POP'       */
#define    DUPOP       13   /* 'DUP'       */
#define    SWAPOP      14   /* 'SWAP'      */
#define    CALLOP      15   /* 'CALL'      */
#define    RTNOP       16   /* 'RTN'       */
#define    GOTOOP      17   /* 'GOTO'      */
#define    CONDOP      18   /* 'COND'      */
#define    CODEOP      19   /* 'CODE'      */
#define    SOSOP       20   /* 'SOS'       */
#define    LIMITOP     21   /* 'LIMIT'     */

    /* ABSTRACT MACHINE OPERANDS */

        /* UNARY OPERANDS */
#define    UNOT        22   /* 'UNOT'      */
#define    UNEG        23   /* 'UNEG'      */
#define    USUCC       24   /* 'USUCC'     */
#define    UPRED       25   /* 'UPRED'     */
        /* BINARY OPERANDS */
#define    BAND        26   /* 'BAND'      */
#define    BOR         27   /* 'BOR'       */
#define    BPLUS       28   /* 'BPLUS'     */
#define    BMINUS      29   /* 'BMINUS'    */
#define    BMULT       30   /* 'BMULT'     */
#define    BDIV        31   /* 'BDIV'      */
#define    BEXP        32   /* 'BEXP'      */
#define    BMOD        33   /* 'BMOD'      */
#define    BEQ         34   /* 'BEQ'       */
#define    BNE         35   /* 'BNE'       */
#define    BLE         36   /* 'BLE'       */
#define    BGE         37   /* 'BGE'       */
#define    BLT         38   /* 'BLT'       */
#define    BGT         39   /* 'BGT'       */
        /* OS SERVICE CALL OPERANDS */
#define    TRACEX      40   /* 'TRACEX'    */
#define    DUMPMEM     41   /* 'DUMPMEM'   */
#define    OSINPUT     42   /* 'INPUT'     */
#define    OSINPUTC    43   /* 'INPUT'     */
#define    OSOUTPUT    44   /* 'OUTPUT'    */
#define    OSOUTPUTC   45   /* 'OUTPUT'    */
#define    OSOUTPUTL   46   /* 'OUTPUTL'   */
#define    OSEOF       47   /* 'EOF'       */

        /* TREE NODE NAMES */
#define    ProgramNode  48   /* 'program'   */
#define    TypesNode    49   /* 'types'     */
#define    TypeNode     50   /* 'type'      */
#define    DclnsNode    51   /* 'dclns'     */
#define    DclnNode     52   /* 'dcln'      */
#define    IntegerTNode 53   /* 'integer'   */
#define    BooleanTNode 54   /* 'boolean'   */
#define    BlockNode    55   /* 'block'     */
#define    AssignNode   56   /* 'assign'    */
#define    OutputNode   57   /* 'output'    */
#define    IfNode       58   /* 'if'        */
#define    WhileNode    59   /* 'while'     */
#define    NullNode     60   /* '<null>'    */
```

```
#define     LENode        61   /* '<='        */
#define     PlusNode      62   /* '+'         */
#define     MinusNode     63   /* '-'         */
#define     ReadNode      64   /* 'read'      */
#define     IntegerNode   65   /* '<integer>'*/
#define     IdentifierNode 66  /* '<identifier>'*/

typedef int Mode;

FILE *CodeFile;
char *CodeFileName;
Clabel HaltLabel;

char *mach_op[] =
    {"NOP","HALT","LIT","LLV","LGV","SLV","SGV","LLA","LGA",
     "UOP","BOP","POP","DUP","SWAP","CALL","RTN","GOTO","COND",
     "CODE","SOS","LIMIT","UNOT","UNEG","USUCC","UPRED","BAND",
     "BOR","BPLUS","BMINUS","BMULT","BDIV","BEXP","BMOD","BEQ",
     "BNE","BLE","BGE","BLT","BGT","TRACEX","DUMPMEM","INPUT",
     "INPUTC","OUTPUT","OUTPUTC","OUTPUTL","EOF"};

/*****************************************************************
   add new node names to the end of the array, keeping in strict order
   as defined above, then adjust the j loop control variable in
   InitializeNodeNames().
*****************************************************************/
char *node_name[] =
    {"program","types","type","dclns","dcln","integer",
     "boolean","block","assign","output","if","while",
     "<null>","<=","+","-","read","<integer>","<identifier>"};


void CodeGenerate(int argc, char *argv[]) {
   int NumberTrees;

   InitializeCodeGenerator(argc,argv);
   Tree_File = Open_File("_TREE", "r");
   NumberTrees = Read_Trees();
   fclose (Tree_File);

   HaltLabel = ProcessNode (RootOfTree(1), NoLabel);
   CodeGen0 (HALTOP, HaltLabel);

   CodeFile = Open_File (CodeFileName, "w");
   DumpCode (CodeFile);
   fclose(CodeFile);

   if (TraceSpecified)
      fclose (TraceFile);

/*****************************************************************
  enable this code to write out the tree after the code generator
  has run.  It will show the new decorations made with MakeAddress().
*****************************************************************/

   Tree_File = fopen("_TREE", "w");
   Write_Trees();
   fclose (Tree_File);
}

void InitializeCodeGenerator(int argc,char *argv[]) {
   InitializeMachineOperations();
   InitializeNodeNames();
   FrameSize = 0;
   CurrentProcLevel = 0;
   LabelCount = 0;
   CodeFileName = System_Argument("-code", "_CODE", argc, argv);
}

void InitializeMachineOperations(void) {
   int i,j;
   for (i=0, j=1; i < 47; i++, j++)
      String_Array_To_String_Constant (mach_op[i],j);
}

void InitializeNodeNames(void) {
   int i,j;
   for (i=0, j=48; j <= 66; i++, j++)
      String_Array_To_String_Constant (node_name[i],j);
}

String MakeStringOf(int Number) {
   Stack Temp;
   Temp = AllocateStack (50);
   ResetBufferInTextTable();
   if (Number == 0)
      AdvanceOnCharacter ('0');
   else {
```

#define ForNode 67
#define LoopNode 68
#define PrePP 69
#define PostPP 70

, "for", "loop", "pre++", "post++"

70

```
      while (Number > 0) {
         Push (Temp,(Number % 10) + 48);
         Number /= 10;
      }
      while ( !(IsEmpty (Temp)))
         AdvanceOnCharacter ((char)(Pop(Temp)));
   }
   return (ConvertStringInBuffer());
}

void Reference(TreeNode T, Mode M, Clabel L) {
   int Addr,OFFSET;
   String  Op;

   Addr = Decoration(Decoration(T));
   OFFSET = FrameDisplacement (Addr) ;
   switch (M) {
      case LeftMode  :  DecrementFrameSize();
                        if (ProcLevel (Addr) == 0)
                           Op = SGVOP;
                        else
                           Op = SLVOP;
                        break;
      case RightMode :  IncrementFrameSize();
                        if (ProcLevel (Addr) == 0)
                           Op = LGVOP;
                        else
                           Op = LLVOP;
                        break;
   }
   CodeGen1 (Op,MakeStringOf(OFFSET),L);
}

int NKids (TreeNode T) {
   return (Rank(T));
}

void Expression (TreeNode T, Clabel CurrLabel) {
   int Kid;
   Clabel Label1;

   if (TraceSpecified) {
      fprintf (TraceFile, "<<< CODE GENERATOR >>> Processing Node ");
      Write_String (TraceFile, NodeName (T) );
      fprintf (TraceFile, " , Label is  ");
      Write_String (TraceFile, CurrLabel);
      fprintf (TraceFile, "0);
   }

   switch (NodeName(T)) {
      case LENode :
      case PlusNode :
         Expression ( Child(T,1) , CurrLabel);
         Expression ( Child(T,2) , NoLabel);
         if (NodeName(T) == LENode)
            CodeGen1 (BOPOP, BLE, NoLabel);
         else
            CodeGen1 (BOPOP, BPLUS, NoLabel);
         DecrementFrameSize();
         break;

      case MinusNode :
         Expression ( Child(T,1) , CurrLabel);
         if (Rank(T) == 2) {
            Expression ( Child(T,2) , NoLabel);
            CodeGen1 (BOPOP, BMINUS, NoLabel);
            DecrementFrameSize();
         }
         else
            CodeGen1 (UOPOP, UNEG, NoLabel);
         break;

      case ReadNode :
         CodeGen1 (SOSOP, OSINPUT, CurrLabel);
         IncrementFrameSize();
         break;

      case IntegerNode :
         CodeGen1 (LITOP, NodeName (Child(T,1)), CurrLabel);
         IncrementFrameSize();
         break;

      case IdentifierNode :
         Reference (T,RightMode,CurrLabel);
         break;

      default :
         ReportTreeErrorAt(T);
         printf ("<<< CODE GENERATOR >>> : UNKNOWN NODE NAME ");
```

(1) PrePP: Reference(Child(T,1), RightMode, CurrLabel);
(2) CodeGen1(UOPOP, USUCC, NoLabel);
(3) Reference(Child(T,1), LeftMode, NoLabel);
(4) Reference(Child(T,1), RightMode, NoLabel);
(5) break;

PostPP: los mismos 4, pero en orden

(1)
(4)
(2)
(3)
(5)

RuleNode: CodeGen1(LITOP, MakeStringOf(1), CurrLabel);
break;

```
        Write_String (stdout,NodeName(T));
        printf ("\n");

    } /* end switch */
} /* end Expression */


Clabel ProcessNode (TreeNode T, Clabel CurrLabel) {
   int Kid, Num;
   Clabel Label1, Label2, Label3;

   if (TraceSpecified) {
      fprintf (TraceFile, "<<< CODE GENERATOR >>> Processing Node ");
      Write_String (TraceFile, NodeName (T) );
      fprintf (TraceFile, " , Label is  ");
      Write_String (TraceFile, CurrLabel);
      fprintf (TraceFile, "\n");
   }

   switch (NodeName(T)) {
      case ProgramNode :
         CurrLabel = ProcessNode (Child(T,NKids(T)-2),CurrLabel);
         CurrLabel = ProcessNode (Child(T,NKids(T)-1),NoLabel);
         return (CurrLabel);

      case TypesNode :
         for (Kid = 1; Kid <= NKids(T); Kid++)
            CurrLabel = ProcessNode (Child(T,Kid), CurrLabel);
         return (CurrLabel);

      case TypeNode :
         return (CurrLabel);

      case DclnsNode :
         for (Kid = 1; Kid <= NKids(T); Kid++)
            CurrLabel = ProcessNode (Child(T,Kid), CurrLabel);
         if (NKids(T) > 0)
            return (NoLabel);
         else
            return (CurrLabel);

      case DclnNode :
         for (Kid = 1; Kid < NKids(T); Kid++)
         {
            if (Kid != 1)
               CodeGen1 (LITOP,MakeStringOf(0),NoLabel);
            else
               CodeGen1 (LITOP,MakeStringOf(0),CurrLabel);
            Num = MakeAddress();
            Decorate ( Child(T,Kid), Num);
            IncrementFrameSize();
         }
         return (NoLabel);

      case BlockNode :
         for (Kid = 1; Kid <= NKids(T); Kid++)
            CurrLabel = ProcessNode (Child(T,Kid), CurrLabel);
         return (CurrLabel);

      case AssignNode :
         Expression (Child(T,2), CurrLabel);
         Reference (Child(T,1), LeftMode, NoLabel);
         return (NoLabel);

      case OutputNode :
         Expression (Child(T,1), CurrLabel);
         CodeGen1 (SOSOP, OSOUTPUT, NoLabel);
         DecrementFrameSize();
         for (Kid = 2; Kid <= NKids(T); Kid++)
         {
            Expression (Child(T,Kid), NoLabel);
            CodeGen1 (SOSOP, OSOUTPUT, NoLabel);
            DecrementFrameSize();
         }
         CodeGen1 (SOSOP, OSOUTPUTL, NoLabel);
         return (NoLabel);

      case IfNode :
         Expression (Child(T,1), CurrLabel);
         Label1 = MakeLabel();
         Label2 = MakeLabel();
         Label3 = MakeLabel();
         CodeGen2 (CONDOP,Label1,Label2, NoLabel);
         DecrementFrameSize();
         CodeGen1 (GOTOOP, Label3, ProcessNode (Child(T,2), Label1) );
         if (Rank(T) == 3)
            CodeGen0 (NOP, ProcessNode (Child(T,3),Label2));
         else
            CodeGen0 (NOP, Label2);
         return (Label3);
```

Pre PP

Post PP: Reference (Child(T,1), RightMode, CurrLabel);
CodeGen1 (UOPOP, USUCC, NoLabel);
Reference (Child(T,1), LeftMode, NoLabel);
return NoLabel;

```
BopNode:
   if ( CurrLabel == NoLabel )
      Label3 = MakeLabel();
   else Label3 = CurrLabel;
   CurrLabel = Label3;
   CurrLabel = ProcessNode ( Child(T,1),
                             CurrLabel );
   Expression ( Child(T,2), CurrLabel);
   Label1 = MakeLabel();
   Label2 = MakeLabel();
   CodeGen2( CONDOP, Label1, Label2,
             NoLabel );
   CodeGen1( GOTOOP, Label3,
             ProcessNode ( Child(T,3), Label1 ) );
   return Label2;
```

```
case WhileNode :
   if (CurrLabel == NoLabel)
      Label1 = MakeLabel();
   else
      Label1 = CurrLabel;
   Label2 = MakeLabel();
   Label3 = MakeLabel();
   Expression (Child(T,1), Label1);
   CodeGen2 (CONDOP, Label2, Label3, NoLabel);
   DecrementFrameSize();
   CodeGen1 (GOTOOP, Label1, ProcessNode (Child(T,2), Label2) );
   return (Label3);

case NullNode : return(CurrLabel);

default :
        ReportTreeErrorAt(T);
        printf ("<<< CODE GENERATOR >>> : UNKNOWN NODE NAME ");
        Write_String (stdout,NodeName(T));
        printf ("0);

} /* end switch */
} /* end ProcessNode */
```

For Node:

```
CurLabel = ProcessNode ( Ch(T,1), CurrLabel));
if (CurLabel = NoLabel)
    Label3 = CurrLabel;
else Label3 = MakeLabel();
    Expression ( child(T,2), Label3));
    Label2 = MakeLabel ();
    Label1 = MakeLabel ();
    CodeGen2 ( CONDOP, Label1, Label2, NoLabel);
    Label1 = ProcessNode ( child(T,4), Label1);
    Label1 = ProcessNode ( child(T,3), Label1);
    CodeGen1 ( GOTOOP, Label3, Label1);
    return Label2;
```