COP 4620/5625

PROGRAMMING LANGUAGE TRANSLATORS

Instructor: Manuel Bermudez

FALL 2005

FINAL EXAM

Take-home, Drop-dead deadline: 12 noon Friday Dec. 16

Turn in at my office or at E301 CSE

Problem 1 _____ (10p.)

Problem 2 _____ (20p.)

Problem 3 _____ (15p.)

Problem 4 _____ (15p.)

Problem 5 _____ (15p.)

Problem 6 _____ (15p.)

Problem 7 _____ (15p.)

SCORE _____ (105p.)

NAME _____

PROBLEM 1. (10 points)

Write a string-to-tree transduction grammar, with the following characteristics.

1. The language is composed of expressions, with operators and the atomic symbol 'a'.
2. The language has the following operator precedence hierarchy.

Level	Operator	Type of Op.	Associativity
Least binding	'&','*'	Binary Infix	Left
	'^','%','~'	Binary Infix	Right
	'#','+'	Binary Infix	Left
	'|'	Unary Prefix	None
	'@'	Unary Postfix	None
Most binding	'(' ')'	Embedding	None

Show the derivation tree and the abstract syntax tree of the following expression:

```
a & a * | a # a ^ | | a @ + a % a ~ a
```

PROBLEM 2. (20 points)

The following string-to-tree transduction grammar describes the language of regular expressions over alphabet {a, b, c}.

```
E0 -> E0 '|' E1     => '|'
   -> E1
E1 -> E1 E2         => 'cat'
   -> E2
E2 -> E2 'list' E3  => 'list'
   -> E3
E3 -> E4 '+'        => '+'
   -> E4 '*'        => '*'
   -> E4 '?'        => '?'
   -> E4
E4 -> '(' E0 ')'
   -> 'a'           => 'a'
   -> 'b'           => 'b'
   -> 'c'           => 'c'
```

This language consists of regular expressions. Here are a couple of sample regular expressions:

1) `(a | b)* c? (b a)+`
2) `(a* | b)* list c a+`

The language has the following operator precedence hierarchy.

Level	Operator	Type of Op.	Associativity
Least binding	'|'	Binary Infix	Left
	''	Binary Infix	Left
	'list'	Binary Infix	Left
	'+','*','?'	Unary Postfix	None
Most binding	'(' ')'	Embedding	None

1) Show that the above gramamr is not LL(1).
2) Transform the grammar to make it LL(1), calculate Select sets and show the modified grammar is LL(1).
3) Show the parse table for the modified grammar.
4) Write a recursive descent parser (in pseudo-code) for this language, adding 'BuildTree' statements to build the Abstract Syntax Tree for the original grammar.
5) Show a step-by-step parse for each of the two sample regular expressions shown above. In each step, show the stack, the remaining input, and any BuildTree invocations.
6) Construct the LR(0) item sets, and build the ACTION and GOTO tables of the LR(0) parser.
7) Show that the grammar is NOT LR(0).
8) Is the grammar SLR(1) ? Show the necessary analysis.
9) Is the grammar LALR(1) ? Show the necessary analysis.
10) Using your ACTION and GOTO tables, show a step-by-step LR parse for each of the two sample regular expressions shown above. In each step, show the stack, the remaining input, and the action taken (shift/reduce). Then take the sequence of reductions performed by the parser, and use them to build (bottom-up) the Abstract Syntax tree for the input string.

PROBLEM 3. (15 points)

Consider the following context-free grammar.

$$
\begin{aligned}
S &\rightarrow Q\perp \\
Q &\rightarrow R\ r\ T \\
&\rightarrow R \\
R &\rightarrow s\ T \\
&\rightarrow t \\
T &\rightarrow R
\end{aligned}
$$

1) Show that the grammar is NOT LL(1).
2) Modify the grammar to make it LL(1).
3) Construct First, Follow and Select sets (as necessary) to prove that the new grammar is LL(1).
4) Construct the corresponding LL(1) (a.k.a. OPF) parse table.
5) Hand-simulate the execution of the classical top-down parsing algorithm, using your parse table, for the input string "sstrst$\perp$ ". For each step, show the stack, the input, and the result of OPF. Then take the sequence of productions produced by OPF and use them to build (top-down) the derivation tree for the input string.

PROBLEM 4. (15 points)

Consider the same grammar as in Problem 3:

$$
\begin{aligned}
S &\rightarrow Q\perp \\
Q &\rightarrow R\ r\ T \\
&\rightarrow R \\
R &\rightarrow s\ T \\
&\rightarrow t \\
T &\rightarrow R
\end{aligned}
$$

a) Construct the LR(0) item sets, and build the ACTION and GOTO tables of the LR(0) parser.

b) Show that the grammar is NOT LR(0).

c) Is the grammar SLR(1) ? Show the necessary analysis.

d) Is the grammar LALR(1) ? Show the necessary analysis.

e) Hand-simulate the execution of the LR parsing algorithm, using your ACTION and GOTO tables (modified in c), for the input string "sstrst$\perp$ ". For each step, show the stack, the input, and the action taken (shift/reduce). Then take the sequence of reductions performed by the parser, and use them to build (bottom-up) the derivation tree for the input string.

PROBLEM 5. (15 points)

You are about to (partially) write a compiler-interpreter for roman numerals. Below is a (slightly incomplete) grammar that recognizes a list of roman numerals, separated by commas. Each numeral is between 1 and 3999. In case you don't remember how roman numerals work, the following rules should refresh your memory:

1) Roman numerals are composed of symbols I (one), V (five), X (ten), L (fifty), C (one hundred), D (five hundred), and M (one thousand).

2) A smaller number placed in front of a larger number is subtracted from it. (e.g. IV means four). Only one such smaller number can be subtracted at a time (e.g. IIV does not mean 3). Other restrictions apply (see the grammar).

3) A number placed after a greater or equal number is added to it (e.g. DC means 600, XIII means 13, and MMM means 3000). Again, other restrictions apply.

Romans	→	Roman (',' Roman)*	=> 'romans'
Roman	→	Thous Hunds Tens Ones	=> ' '
Thous	→	M? M? M?	=> ' '
Hunds	→	C C? C?	=> ' '
	→	C (D + M)	=> ' '
	→	D C? C? C?	=> ' '
	→		
Tens	→	X X? X?	=> ' '
	→	X (L + C)	=> ' '
	→	L X? X? X?	=> ' '
	→		
Ones	→	I I? I?	=> ' '
	→	I (V + X)	=> ' '
	→	V I? I? I?	=> ' '
	→		
M	→	'M'	=> '1000'
D	→	'D'	=> '500'
C	→	'C'	=> '100'
L	→	'L'	=> '50'
X	→	'X'	=> '10'
V	→	'V'	=> '5'
I	→	'I'	=> '1'

a) Is the above language of Roman numerals a context-free language, or is it a regular language ? Justify your answer.

b) Some of the tree transduction parts are blank. Each one should be either "+" or "-". YOU FILL THEM IN.

Now, assume someone has already undertaken the task of developing a recursive descent parser for this grammar.

c) Show the derivation tree and the abstract syntax tree for the following input:

DLVII, MI, MCMXCIV

d) Below is a (slightly incomplete) roman compiler back-end program, written in pseudo code. The code for cases "Plus" and "Minus" is missing. YOU FILL THEM IN.

```
program RomanBackEnd(input,output);
const
        One = '1';
        Five = '5';
        Ten  = '10';
        Fifty = '50';
        Hundr = '100';   { These are Node Names }
        FiveH = '500';
        Thous = '1000';
        Plus  = '+';
        Minus = '-';
var i : integer;
function Traverse (T:TreeNode): integer;
var N,i: integer;
begin
        case NodeName(T) of
                Plus:
                Minus:
                One   : Traverse := 1;
                Five  : Traverse := 5;
                Ten   : Traverse := 10;
                Fifty : Traverse := 50;
                Hundr : Traverse := 100;
                FiveH : Traverse := 500;
                Thous : Traverse := 1000;
        end
end;
begin {main}
        ReadTree;
        for i := 1 to Rank(RootOfTree(1)) do
                writeln('Value Is ',Traverse (Child(Root,i)));
end.
```

e) Annotate each "+" and "-" in your AST with the value returned by function "Traverse", when applied to that node. Show the output of the roman compiler back-end.

PROBLEM 6. (15 points)

The Algol "for" loop has the following form:

for i := flexpr1 , flexpr2 , ..., flexprn do S

Each flexpri (for-list expression i) is either an integer expression, or a range of the form

lower-exprn .. upper-exprn

For example, assuming j=3 and k=4, the statement

for i:= j+1, 3, j .. k, k .. j, 7, 1 .. k do statement

will execute the statement with the following values of i: 4,3,3,4,7,1,2,3,4. Notice that the ranges are upwards, so the range k .. j causes the statement to be executed zero times. Notice also that the loop control variable is incremented automatically.

At the end of this exam you will find the original Tiny. Specifically, you will find the parser specification, the code for the constrainer, and the code for the code generator. Mark the listings with the changes required to implement the Algol loop construct.

PROBLEM 7. (15 points)

Consider the addition of conditional expressions to Tiny. A conditional expression is similar to an ‘‘if’’ statement, except that instead of chosing between two statements, one choses between two expressions. A conditional expression can appear anywhere that an ordinary expression can, the same way that an ‘‘if’’ statement can appear anywhere that an ordinary statement can. The general form is E1 ? E2 : E3. The ‘‘else’’ part (or ‘‘:’’ part) is mandatory. Here are some examples:

Example	Meaning
f(a<b ? 0 : 1)	f(0), if a<b f(1), otherwise
X := eof ? true : false	X := true, if eof X := false, otherwise
f := n>1 ? n*f(n-1) : 1	f := n*f(n-1), if n>1 f := 1, otherwise

(a) Design the abstract syntax for the conditional expression, i.e. draw a schematic for the AST for the conditional expression. Hint: the answer is VERY simple.

(b) At the end of this exam you will find the original Tiny. Specifically, you will find the parser specification, the code for the constrainer, and the code for the code generator. Mark the listings with the changes required to implement the conditional expression.

TINY'S GRAMMAR:

```
%%
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>";
```

```
/*************************************************************
             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 TREE\n");
   for (i=1;i<=SizeOf(Tree);i++)
      printf("%2d: %d\n",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;

   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 ("\n");
}


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, "\n");
   }



   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 INTEGER\n");
            printf ("\n");
         }
         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 INTEGER\n");
            printf ("\n");
         }
         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 ("\n");

   }  /* 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, "\n");
   }


   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 MATCH\n");
           printf ("\n");
         }

         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++)
         {
            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 MATCH\n");
            printf ("\n");
         }
         break;
```

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

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

         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 BOOLEAN\n");
            printf ("\n");
         }
         ProcessNode (Child(T,2));
         break;


      case NullNode :
         break;

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

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

```
/*****************************************************************
          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
   {
      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, "\n");
   }

   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 ");
         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);


        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 ("\n");

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