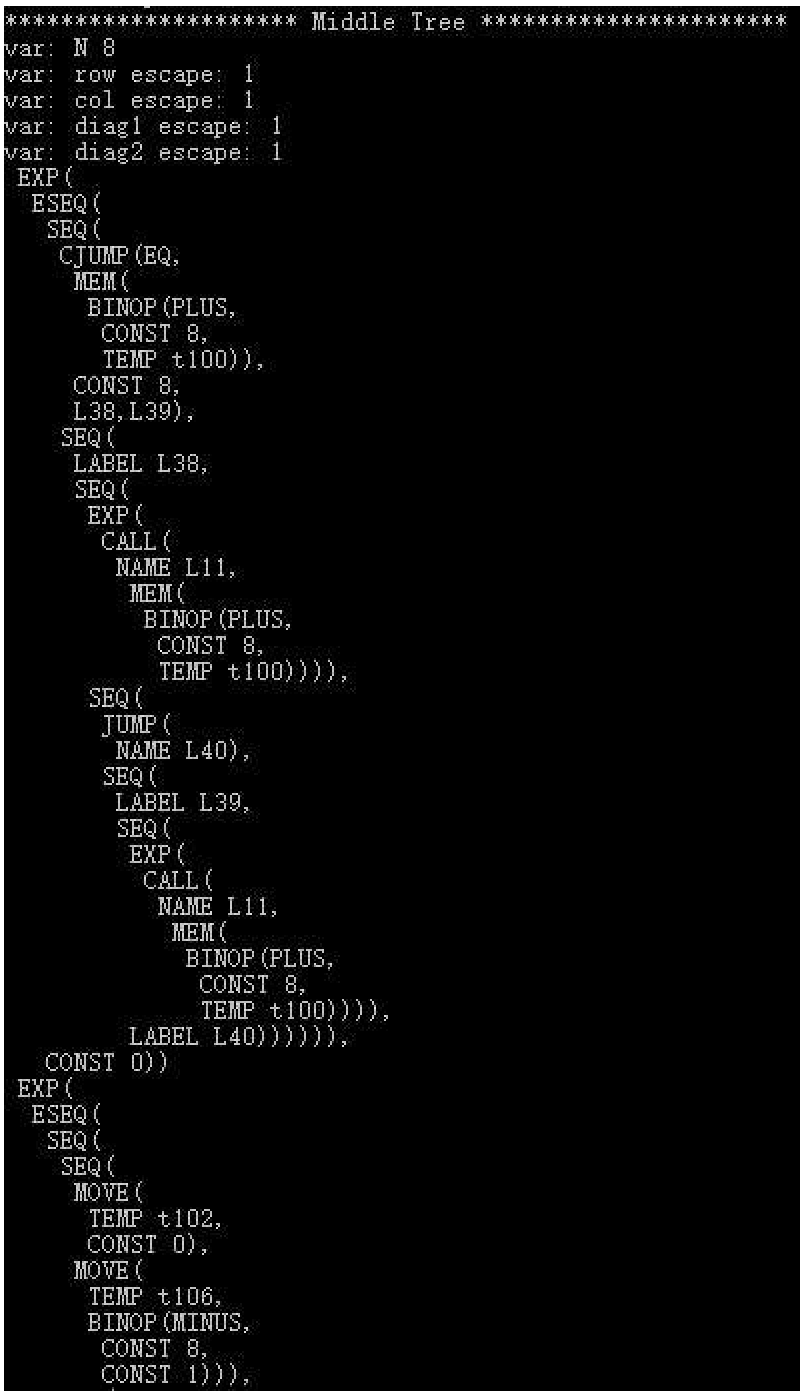
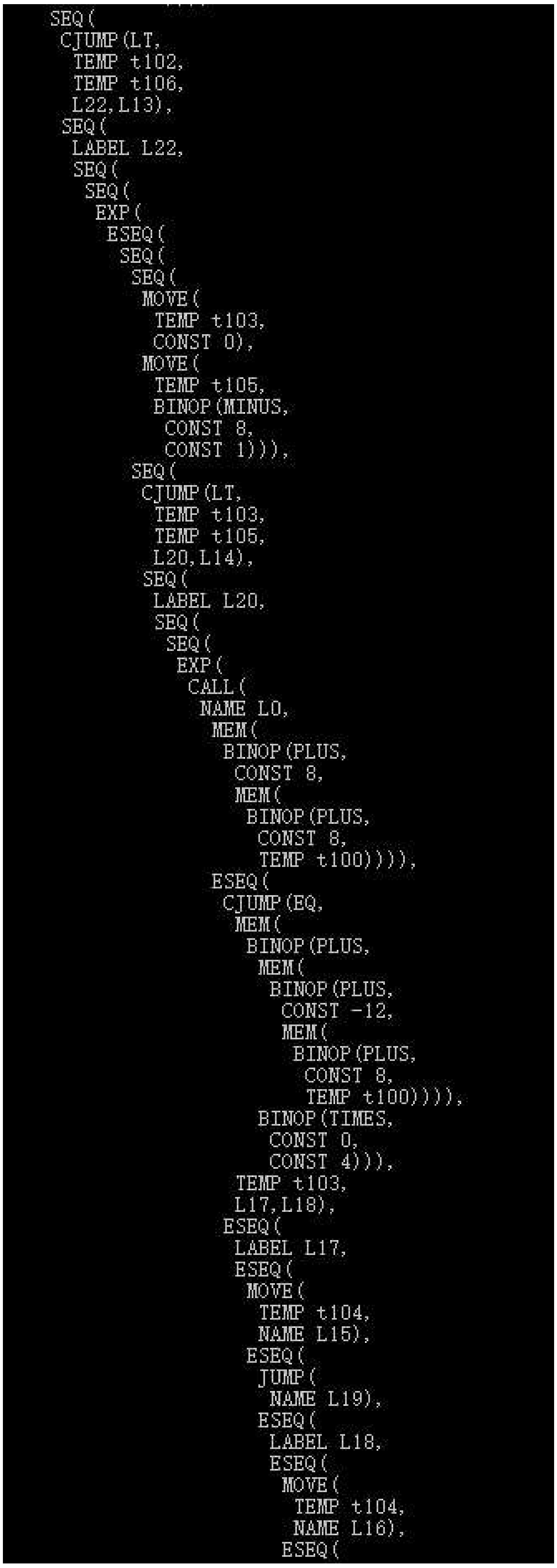
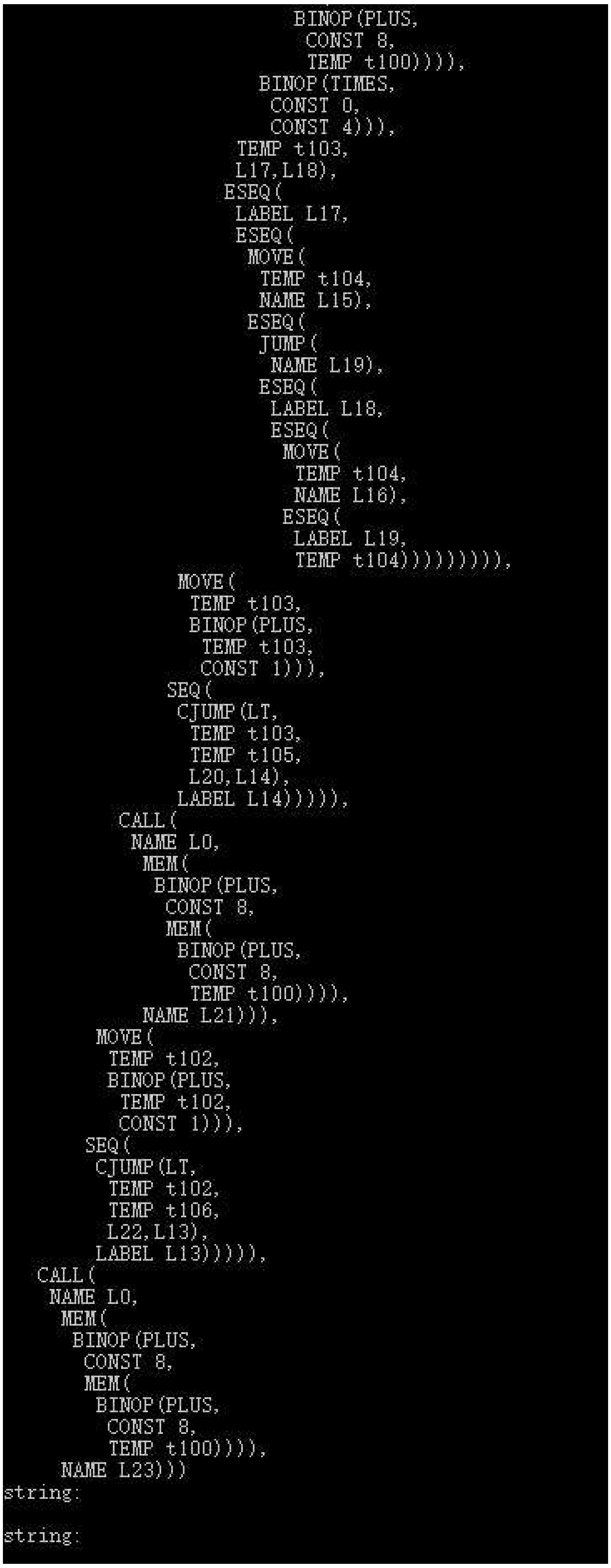
Contents



```
********************* Middle Tree **********************
var: N 8
var: row escape: 1
var: col escape: 1
var: diag1 escape: 1
var: diag2 escape: 1
 EXP(
  ESEQ(
   SEQ(
    CJUMP(EQ,
     MEM(
      BINOP(PLUS,
       CONST 8,
       TEMP t100)),
     CONST 8,
     L38,L39),
    SEQ(
     LABEL L38,
     SEQ(
      EXP(
       CALL(
        NAME L11,
         MEM(
          BINOP(PLUS,
           CONST 8,
           TEMP t100)))),
      SEQ(
       JUMP(
        NAME L40),
       SEQ(
        LABEL L39,
        SEQ(
         EXP(
          CALL(
           NAME L11,
            MEM(
             BINOP(PLUS,
              CONST 8,
              TEMP t100)))),
         LABEL L40)))))),
   CONST 0))
 EXP(
  ESEQ(
   SEQ(
    SEQ(
     MOVE(
      TEMP t102,
      CONST 0),
     MOVE(
      TEMP t106,
      BINOP(MINUS,
       CONST 8,
       CONST 1))),
```

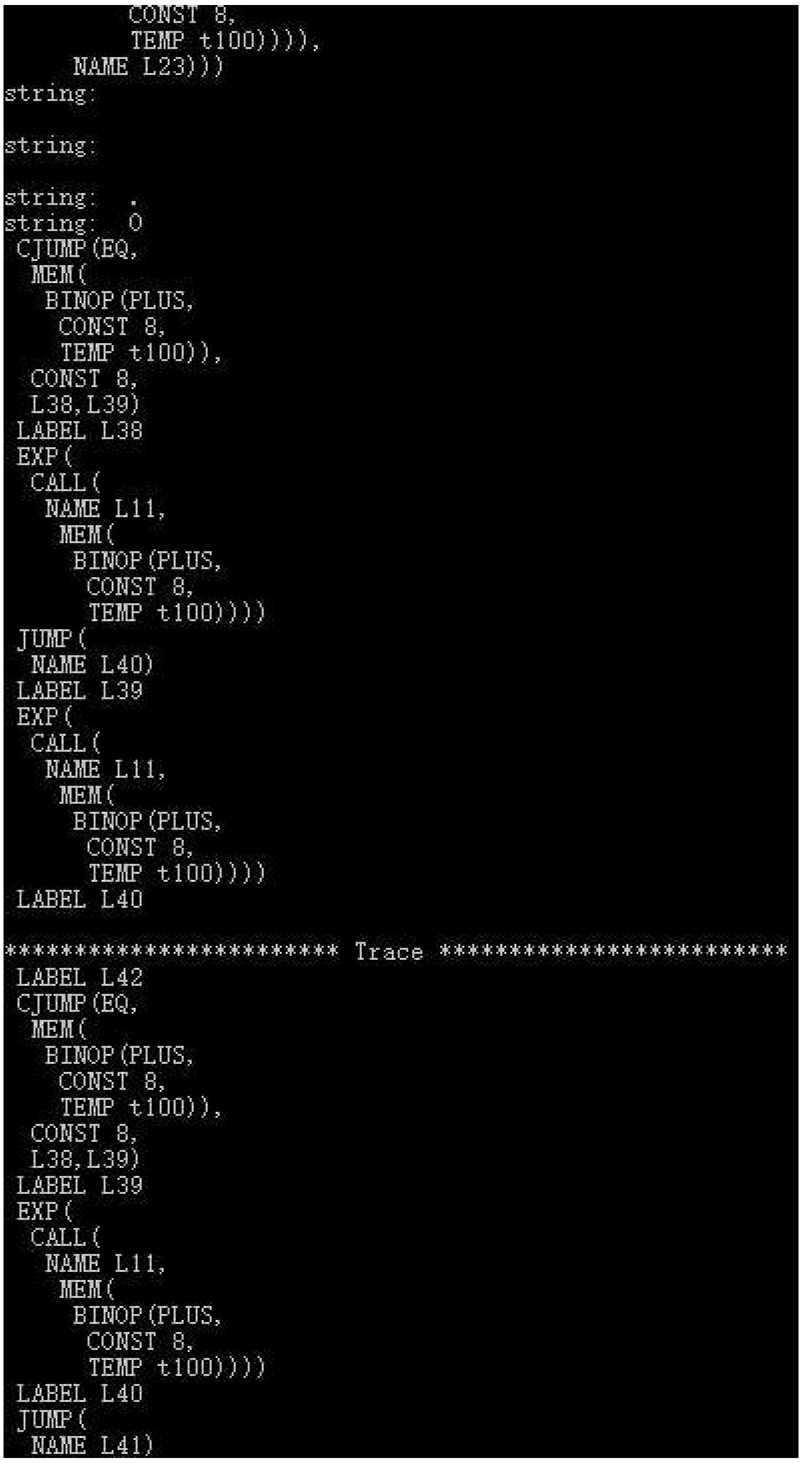


```
SEQ(
 CJUMP(LT,
  TEMP t102,
  TEMP t106,
  L22,L13),
 SEQ(
  LABEL L22,
  SEQ(
   SEQ(
    EXP(
     ESEQ(
      SEQ(
       SEQ(
        MOVE(
         TEMP t103,
         CONST 0),
        MOVE(
         TEMP t105,
         BINOP(MINUS,
          CONST 8,
          CONST 1))),
       SEQ(
        CJUMP(LT,
         TEMP t103,
         TEMP t105,
         L20,L14),
        SEQ(
         LABEL L20,
         SEQ(
          SEQ(
           EXP(
            CALL(
             NAME L0,
              MEM(
               BINOP(PLUS,
                CONST 8,
                MEM(
                 BINOP(PLUS,
                  CONST 8,
                  TEMP t100)))),
              ESEQ(
               CJUMP(EQ,
                MEM(
                 BINOP(PLUS,
                  MEM(
                   BINOP(PLUS,
                    CONST -12,
                    MEM(
                     BINOP(PLUS,
                      CONST 8,
                      TEMP t100)))),
                  BINOP(TIMES,
                   CONST 0,
                   CONST 4))),
                TEMP t103,
                L17,L18),
               ESEQ(
                LABEL L17,
                ESEQ(
                 MOVE(
                  TEMP t104,
                  NAME L15),
                 ESEQ(
                  JUMP(
                   NAME L19),
                  ESEQ(
                   LABEL L18,
                   ESEQ(
                    MOVE(
                     TEMP t104,
                     NAME L16),
                    ESEQ(
```



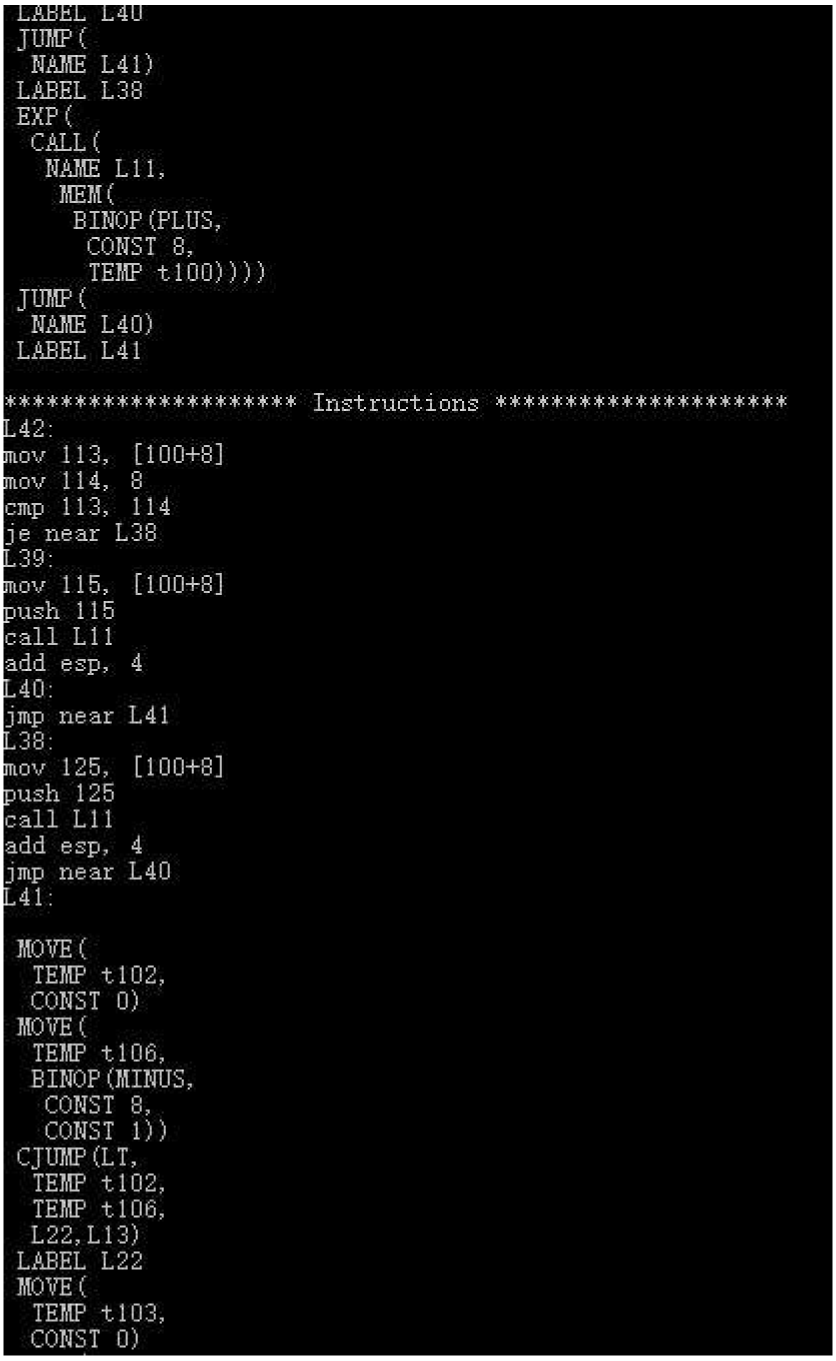
```
                         BINOP(PLUS,
                          CONST 8,
                          TEMP t100)))),
                      BINOP(TIMES,
                       CONST 0,
                       CONST 4))),
                    TEMP t103,
                    L17,L18),
                   ESEQ(
                    LABEL L17,
                    ESEQ(
                     MOVE(
                      TEMP t104,
                      NAME L15),
                     ESEQ(
                      JUMP(
                       NAME L19),
                      ESEQ(
                       LABEL L18,
                       ESEQ(
                        MOVE(
                         TEMP t104,
                         NAME L16),
                        ESEQ(
                         LABEL L19,
                         TEMP t104))))))))),
               MOVE(
                TEMP t103,
                BINOP(PLUS,
                 TEMP t103,
                 CONST 1))),
              SEQ(
               CJUMP(LT,
                TEMP t103,
                TEMP t105,
                L20,L14),
               LABEL L14))))),
          CALL(
           NAME L0,
            MEM(
             BINOP(PLUS,
              CONST 8,
              MEM(
               BINOP(PLUS,
                CONST 8,
                TEMP t100)))),
            NAME L21))),
        MOVE(
         TEMP t102,
         BINOP(PLUS,
          TEMP t102,
          CONST 1))),
       SEQ(
        CJUMP(LT,
         TEMP t102,
         TEMP t106,
         L22,L13),
        LABEL L13))))),
   CALL(
    NAME L0,
     MEM(
      BINOP(PLUS,
       CONST 8,
       MEM(
        BINOP(PLUS,
         CONST 8,
         TEMP t100)))),
     NAME L23)))
string:

string:
```

....50

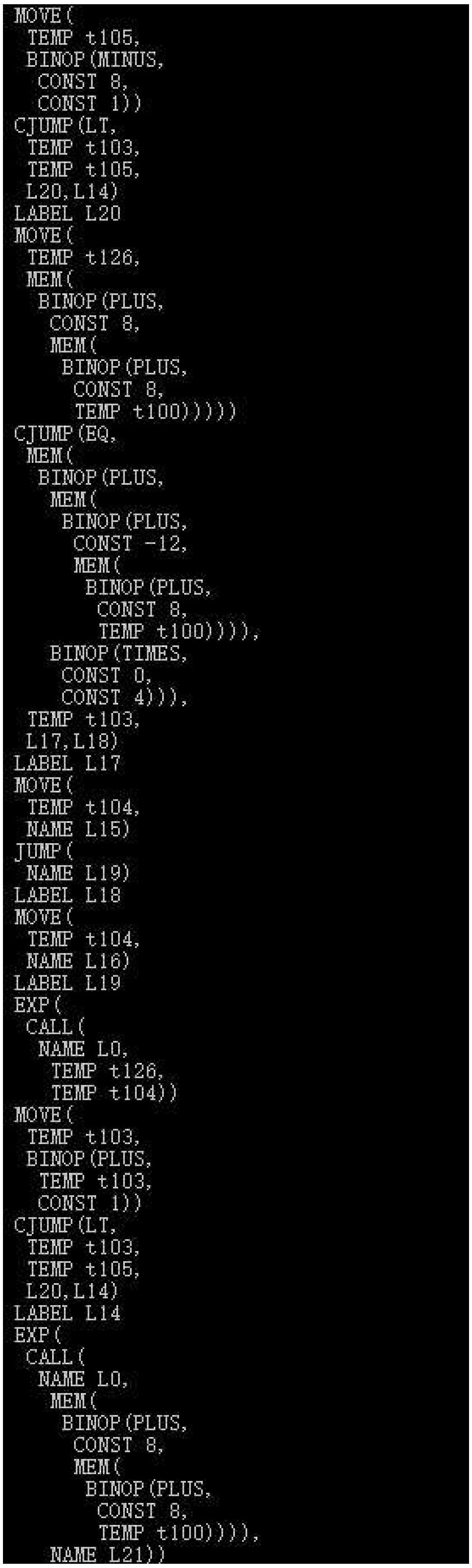


```
         CONST 8,
         TEMP t100)))),
     NAME L23)))
string:

string:

string:  .
string:  0
 CJUMP(EQ,
  MEM(
   BINOP(PLUS,
    CONST 8,
    TEMP t100)),
  CONST 8,
  L38,L39)
 LABEL L38
 EXP(
  CALL(
   NAME L11,
    MEM(
     BINOP(PLUS,
      CONST 8,
      TEMP t100))))
 JUMP(
  NAME L40)
 LABEL L39
 EXP(
  CALL(
   NAME L11,
    MEM(
     BINOP(PLUS,
      CONST 8,
      TEMP t100))))
 LABEL L40

************************ Trace *************************
 LABEL L42
 CJUMP(EQ,
  MEM(
   BINOP(PLUS,
    CONST 8,
    TEMP t100)),
  CONST 8,
  L38,L39)
 LABEL L39
 EXP(
  CALL(
   NAME L11,
    MEM(
     BINOP(PLUS,
      CONST 8,
      TEMP t100))))
 LABEL L40
 JUMP(
  NAME L41)
```

....51

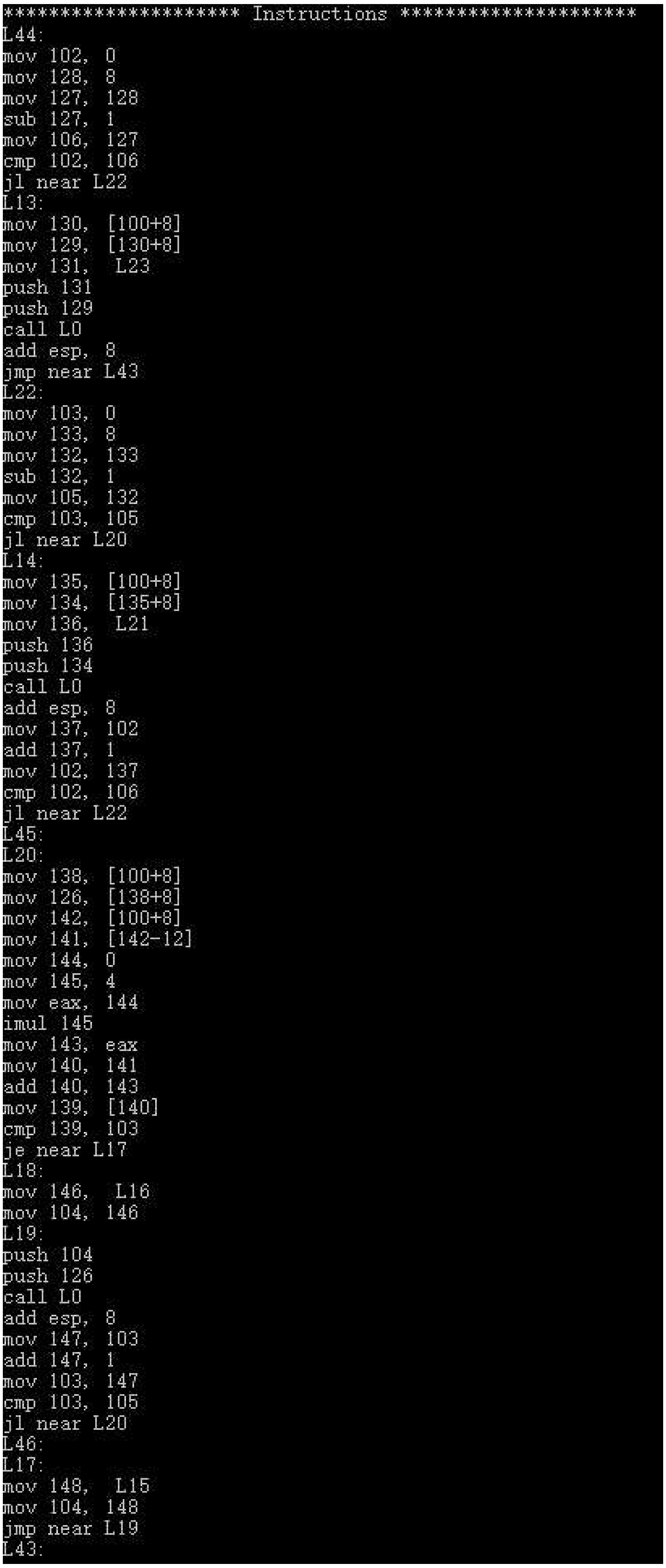


```
 LABEL L40
 JUMP(
  NAME L41)
 LABEL L38
 EXP(
  CALL(
   NAME L11,
    MEM(
     BINOP(PLUS,
      CONST 8,
      TEMP t100))))
 JUMP(
  NAME L40)
 LABEL L41

********************** Instructions **********************
L42:
mov 113, [100+8]
mov 114, 8
cmp 113, 114
je near L38
L39:
mov 115, [100+8]
push 115
call L11
add esp, 4
L40:
jmp near L41
L38:
mov 125, [100+8]
push 125
call L11
add esp, 4
jmp near L40
L41:

 MOVE(
  TEMP t102,
  CONST 0)
 MOVE(
  TEMP t106,
  BINOP(MINUS,
   CONST 8,
   CONST 1))
 CJUMP(LT,
  TEMP t102,
  TEMP t106,
  L22,L13)
 LABEL L22
 MOVE(
  TEMP t103,
  CONST 0)
```



```
MOVE(
 TEMP t105,
 BINOP(MINUS,
  CONST 8,
  CONST 1))
CJUMP(LT,
 TEMP t103,
 TEMP t105,
 L20,L14)
LABEL L20
MOVE(
 TEMP t126,
 MEM(
  BINOP(PLUS,
   CONST 8,
   MEM(
    BINOP(PLUS,
     CONST 8,
     TEMP t100)))))
CJUMP(EQ,
 MEM(
  BINOP(PLUS,
   MEM(
    BINOP(PLUS,
     CONST -12,
     MEM(
      BINOP(PLUS,
       CONST 8,
       TEMP t100)))),
   BINOP(TIMES,
    CONST 0,
    CONST 4))),
 TEMP t103,
 L17,L18)
LABEL L17
MOVE(
 TEMP t104,
 NAME L15)
JUMP(
 NAME L19)
LABEL L18
MOVE(
 TEMP t104,
 NAME L16)
LABEL L19
EXP(
 CALL(
  NAME L0,
   TEMP t126,
   TEMP t104))
MOVE(
 TEMP t103,
 BINOP(PLUS,
  TEMP t103,
  CONST 1))
CJUMP(LT,
 TEMP t103,
 TEMP t105,
 L20,L14)
LABEL L14
EXP(
 CALL(
  NAME L0,
   MEM(
    BINOP(PLUS,
     CONST 8,
     MEM(
      BINOP(PLUS,
       CONST 8,
       TEMP t100)))),
   NAME L21))
```

....53



```
********************* Instructions *********************
L44:
mov 102, 0
mov 128, 8
mov 127, 128
sub 127, 1
mov 106, 127
cmp 102, 106
jl near L22
L13:
mov 130, [100+8]
mov 129, [130+8]
mov 131,  L23
push 131
push 129
call L0
add esp, 8
jmp near L43
L22:
mov 103, 0
mov 133, 8
mov 132, 133
sub 132, 1
mov 105, 132
cmp 103, 105
jl near L20
L14:
mov 135, [100+8]
mov 134, [135+8]
mov 136,  L21
push 136
push 134
call L0
add esp, 8
mov 137, 102
add 137, 1
mov 102, 137
cmp 102, 106
jl near L22
L45:
L20:
mov 138, [100+8]
mov 126, [138+8]
mov 142, [100+8]
mov 141, [142-12]
mov 144, 0
mov 145, 4
mov eax, 144
imul 145
mov 143, eax
mov 140, 141
add 140, 143
mov 139, [140]
cmp 139, 103
je near L17
L18:
mov 146,  L16
mov 104, 146
L19:
push 104
push 126
call L0
add esp, 8
mov 147, 103
add 147, 1
mov 103, 147
cmp 103, 105
jl near L20
L46:
L17:
mov 148,  L15
mov 104, 148
jmp near L19
L43:
```

1. Lexical Analysis

The main task of lexical analysis is to break the input into individual tokens, and keep the position of every token. To achieve a lexical analyzer, we use Lex, an automatic lexical analyzer generator, to translate regular expressions into a DFA.

There are two kinds of input words to deal with: one is the lexical tokens, including reserved words and punctuation symbols; the other is the special character sequences, such as string constants, comments, escape sequences and so on. To process the former kind, we write regular expression to specify the lexical tokens; while for the latter one, we use state machines to deal with them.

1.1 Regular Expression

Regular expressions are convenient for specifying lexical tokens. The lexical tokens in tiger language can be divided into two categories: punctuation symbols and reserved words.

1.1.1 Punctuation Symbols

The punctuation symbols in tiger language and their corresponding token types are as follows:

Regular expression	Token type
,	COMMA
:	COLON
;	SEMICOLON
(	LPAREN
)	RPAREN
[	LBRACK
]	RBRACK
{	LBRACE
}	RBRACE
.	DOT
+	PLUS
-	MINUS
*	TIMES
/	DIVIDE
=	EQ
<>	NEQ
<	LT
<=	LE
>	GT
>=	GE
&	AND
|	OR
:=	ASSIGN

Figure 1.1 Punctuation symbols

1.1.2 Reserved Words

The regular expressions of reserved words in tiger language and their corresponding token types are as follows:

Regular expression	Token type
array	ARRAY
if	IF
then	THEN
else	ELSE
while	WHILE
for	FOR
to	TO
do	DO
let	LET
in	IN
end	END
of	OF
break	BREAK
nil	NIL
function	FUNCTION
var	VAR
type	TYPE
[a-zA-Z]+[a-zA-Z0-9_]*	ID
[0-9]+	INT
[\t]*	
\n	
\r	
.	“illegal token”

Figure 1.2 Reserved words

1.2 State Machine

Although regular expressions are clear and accurate for specifying punctuation symbols and reserved words, they can be weak in identifying special character sequences like comments. Under such circumstance, we need to find a more powerful method to deal with them, that is state machine. Since Lex provides a mechanism to use regular expressions in state machines, we could combine them together to process special character sequences more efficiently.

In this project, we use two state machines: one is for specifying comments, the other is for dealing with string constants and escape sequences.

1.2.1 Comments

Comments begin with “/*” and end up with “*/”, probably have many lines and nesting levels. The state machine for comments is as follows:

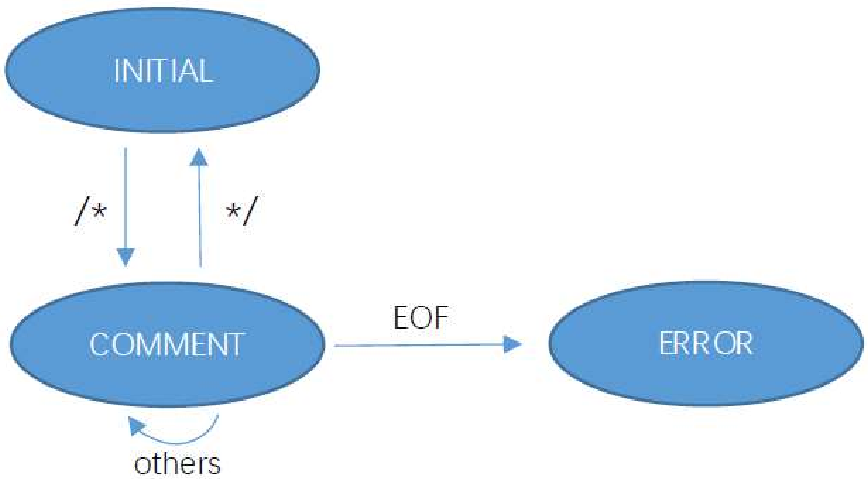


Figure 1.3 State machine for comments

1.2.2 Strings and Escape sequences

A string constant is a sequence, between quotes ("), of zero or more printable characters, spaces, or escape sequences. Each escape sequence is introduced by the escape character \, and stands for a character sequence.

The allowed escape sequences are as follows (all other uses of \ being illegal):

Escape sequence	Meaning
\n	A character interpreted by the system as end-of-line.
\t	Tab.
\ˆc	The control character c, for any appropriate c.
\ddd	The single character with ASCII code ddd (3 decimal digits).
\"	The double-quote character (").
\\	The backslash character (\).
\f__f\	This sequence is ignored, where f__f stands for a sequence of one or more formatting characters (a subset of the non-printable characters including at least space, tab, newline, formfeed). This allows one to write long strings on more than one line, by writing \f at the end of one line and f\ at the start of the next.

Figure 1.4 Escape sequences

The state machine for identifying the strings and escape sequences is as follows:

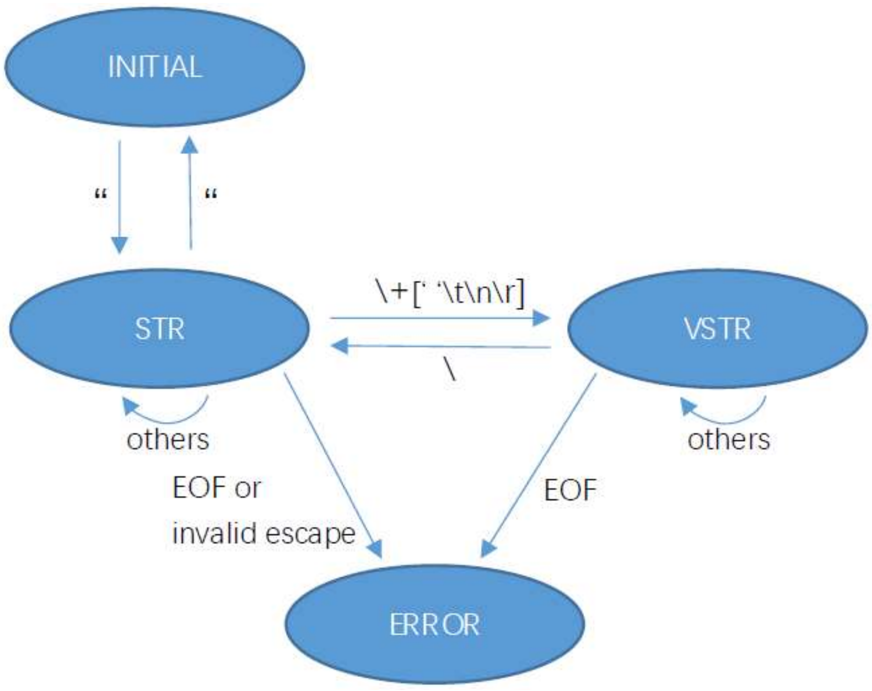


Figure 1.5 State machine for strings and escape sequences

The STR state is for ordinary characters and escape sequences, while the VSTR state is for the escape sequence “/f__f/”. Note that the “f” here is not a letter, but stands for formatting character, including space, tab, newline and formfeed.

2. Syntax analysis

The main task of syntax analysis is to parse the phrase structure of the program. To achieve a syntax parser, we use Yacc(Yet another compiler-compiler), a classic and widely used parser generator, to complete LR parsing for us.

2.1 Nonterminal symbols

In the input file for Yacc, we need to write some CFG to describe the tiger language. CFG consists of terminal and nonterminal symbols. Terminal symbols are the tokens identified by Lex, while nonterminal symbols and their corresponding meaning are as follows:

Nonterminal Symbol	Meaning
program	Input program
exp	Expression
lvalue	Left value
expseq	Expression sequence
funcall	Function call
paraseq	Parameter sequence
asseq	Field sequence
decs	Declarations
dec	Declaration
tydecs	Type declarations
tydec	Type declaration
ty	Type

tyfields	Type fields
tyfield	Type field
vardec	Variable declaration
fundecs	Function declarations
fundec	Function declaration

Figure 2.1 Nonterminal symbols

2.2 CFG

Using the terminal and nonterminal symbols above, we can write CFG to describe tiger language as follows:

```
program  → exp
exp → lvalue
            | lvalue ASSIGN exp
            | INT
            | STRING
            | NIL

            | LPAREN RPAREN
            | LPAREN expseq RPAREN

            | MINUS exp %prec UMINUS
            | exp PLUS exp
            | exp MINUS exp
            | exp TIMES exp
            | exp DIVIDE exp

            | exp EQ exp
            | exp NEQ exp
            | exp LT exp
            | exp LE exp
            | exp GT exp
            | exp GE exp

            | exp AND exp
            | exp OR exp

            | funcall

            | ID LBRACK exp RBRACK OF exp
            | ID LBRACE RBRACE
            | ID LBRACE asseq RBRACE

            | IF exp THEN exp
            | IF exp THEN exp ELSE exp
            | WHILE exp DO exp
            | FOR ID ASSIGN exp TO exp DO exp
            | BREAK

            | LET decs IN END
            | LET decs IN expseq END
```

```
          | LPAREN error RPAREN
          | error SEMICOLON exp
lvalue → ID
            | lvalue DOT ID
            | lvalue LBRACK exp RBRACK
            | ID LBRACK exp RBRACK
expseq    → exp
             | exp SEMICOLON expseq
funcall    → ID LPAREN RPAREN
             | ID LPAREN paraseq RPAREN
paraseq → exp
             | exp COMMA paraseq
asseq → ID EQ exp
          | ID EQ exp COMMA asseq
decs → dec decs
          |
dec  → tydecs
        | vardec
        | fundecs
tydecs → tydec
            | tydec tydecs
tydec → TYPE ID EQ ty
ty → ID
       | LBRACE tyfields RBRACE
       | ARRAY OF ID
tyfields → tyfield
             |
tyfield → ID COLON ID
            | ID COLON ID COMMA tyfield
vardec → VAR ID ASSIGN exp
            | VAR ID COLON ID ASSIGN exp
fundecs   → fundec
             | fundec fundecs
fundec → FUNCTION ID LPAREN tyfields RPAREN EQ exp
            | FUNCTION ID LPAREN tyfields RPAREN COLON ID EQ exp
```

Figure 2.2   CFG

2.3 Priority and Associativity

Since the grammar we write above is ambiguous, we need to define priority and associativity of the tokens, in order to eliminate ambiguity of the grammar. “%left” means left-associative, “%right” means right-associative, and “%nonassoc” means non-associative. The later the associativity is defined in the code, the higher priority the token has.

The priority and associativity of the tokens are as follows:

```
%left SEMICOLON
%right THEN ELSE DOT DO OF
%right ASSIGN
%left OR
%left AND
%nonassoc EQ NEQ LT LE GT GE
%left PLUS MINUS
%left TIMES DIVIDE
%left UMINUS
```

Figure 2.3 Priority and associativity of the tokens

2.4 Abstract Syntax Tree

An abstract syntax makes a clean interface between the parser and the later phases of a compiler. The abstract syntax tree conveys the phrase structure of the source program, with all parsing issues resolved but without any semantic interpretation.

2.4.1 Tree Nodes

The nodes of abstract syntax tree can be divided into the following four categories:

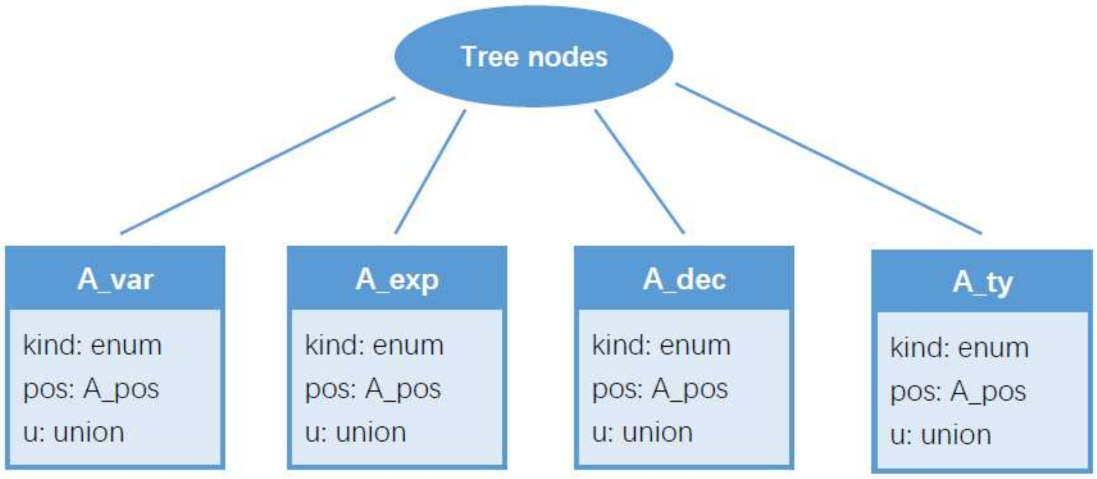


Figure 3.1 Tree nodes

The four kinds of tree nodes above all have three member variables: kind, pos and u. Kind indicates the specific kind of the tree node; pos points to the position of tree node in the source file; u is a union that stores specific information of the tree node, different kinds of tree nodes have different members in union.

2.4.2 A_var

A_var nodes represent variables. There are 3 kinds of A_var nodes:

Kind	Meaning	Example
A_simpleVar	Simple variable	a
A_fieldVar	Field variable	a.value
A_subscriptVar	Subscript variable	a[i+1]

Figure 3.2 Kinds of A_var nodes

The structures of the three kinds of A_var nodes are as follows:

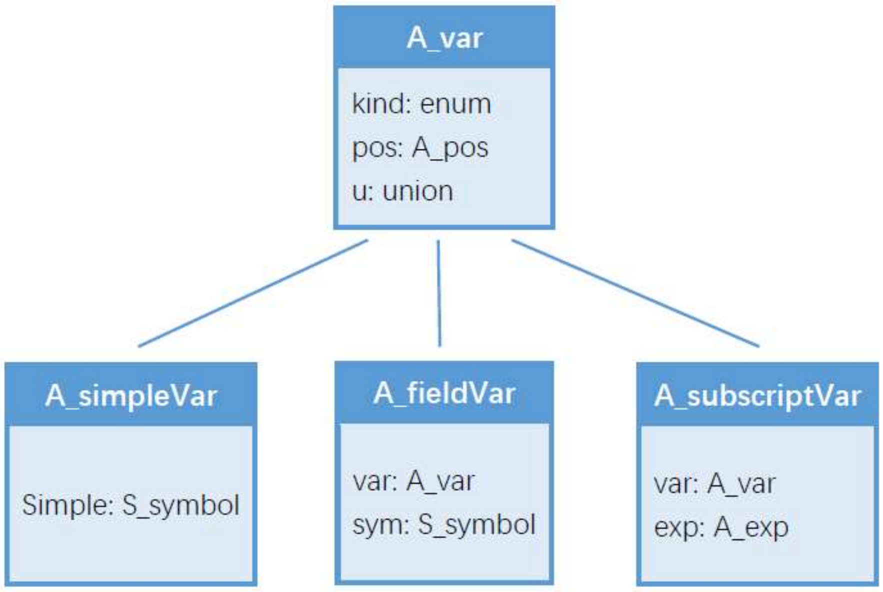


Figure 3.3 Structures of A_var nodes

2.4.3 A_exp

A_exp nodes represent expressions. There are 15 kinds of A_exp nodes:

Kind	Meaning	Example
A_varExp	Variable expression	a
A_nilExp	Nil expression	nil
A_intExp	Int expression	520
A_stringExp	String expression	iloveu
A_callExp	Call expression	print(“iloveu”)
A_opExp	Operation expression	520+1314
A_recordExp	Record expression	love{you=dhd, me=xbd}
A_seqExp	Sequence expression	print(“miss”);print(“you”)
A_assignExp	Assignment expression	love:=520+1314
A_ifExp	If expression	if 1<2 then print(“iloveu”)
A_whileExp	While expression	while 1>0 do print(“iloveu”)
A_forExp	For expression	for a:=520 to 1314 do print(“iloveu”)
A_breakExp	Break expression	break
A_letExp	Let expression	let love:=520 in print(“iloveu”) end
A_arrayExp	Array expression	love [520] of 1314

Figure 3.4 Kinds of A_exp nodes

The structures of the 15 kinds of A_exp nodes are as follows:

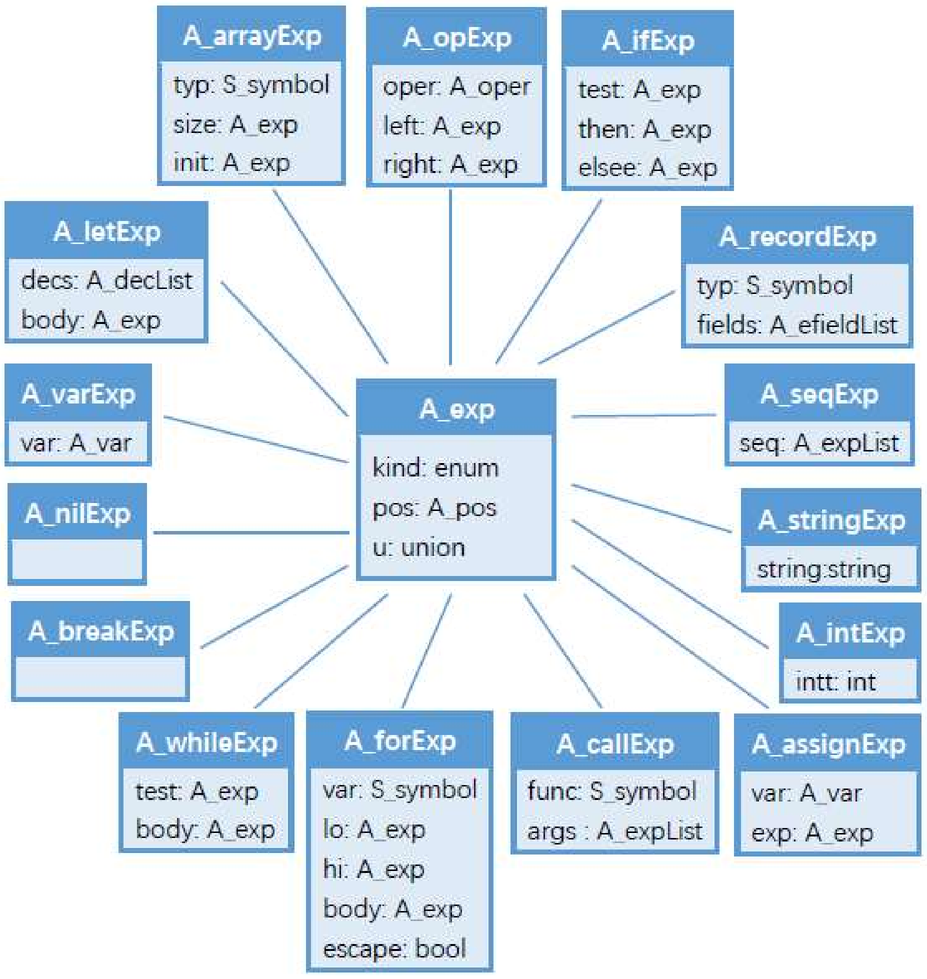


Figure 3.5 Structures of A_exp nodes

2.4.4 A_dec

A_dec nodes represent declarations. There are 3 kinds of A_dec nodes:

Kind	Meaning	Example
A_functionDec	Function declaration	function love(name: string) : int =520
A_varDec	Variable declaration	var love:=520+1314
A_typeDec	Type declaration	type love = {boy: string, girl: string}

Figure 3.6 Kinds of A_dec nodes

The structures of the three kinds of A_dec nodes are as follows:

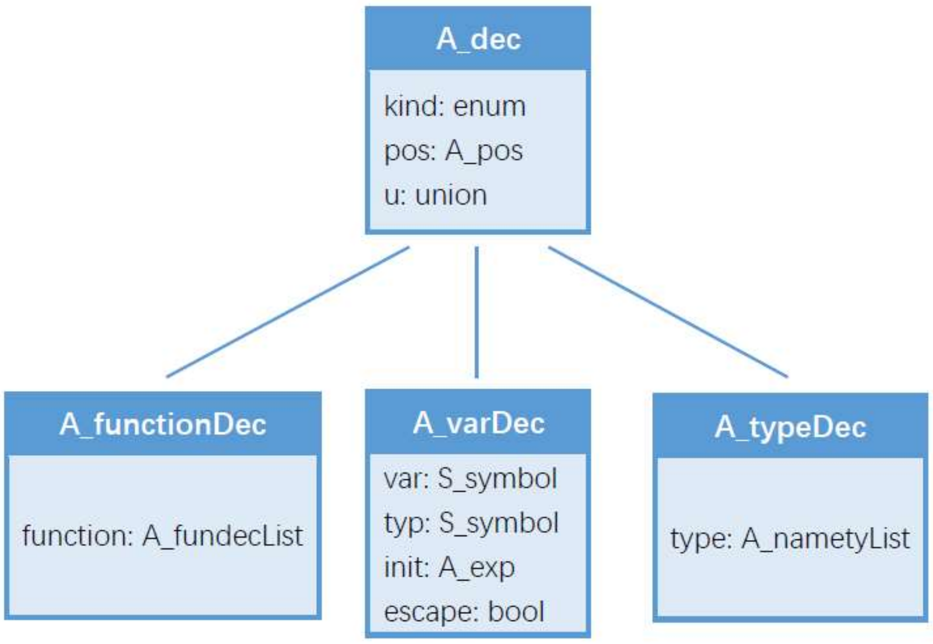


Figure 3.7 Structures of A_dec nodes

2.4.5 A_ty

A_ty nodes represent types. There are 3 kinds of A_ty nodes:

Kind	Meaning	Example
A_nameTy	Type name(type-id)	int
A_recordTy	Record type	{key: int, value: string}
A_arrayTy	Array type	array of int

Figure 3.8 Kinds of A_ty nodes

The structures of the three kinds of A_ty nodes are as follows:

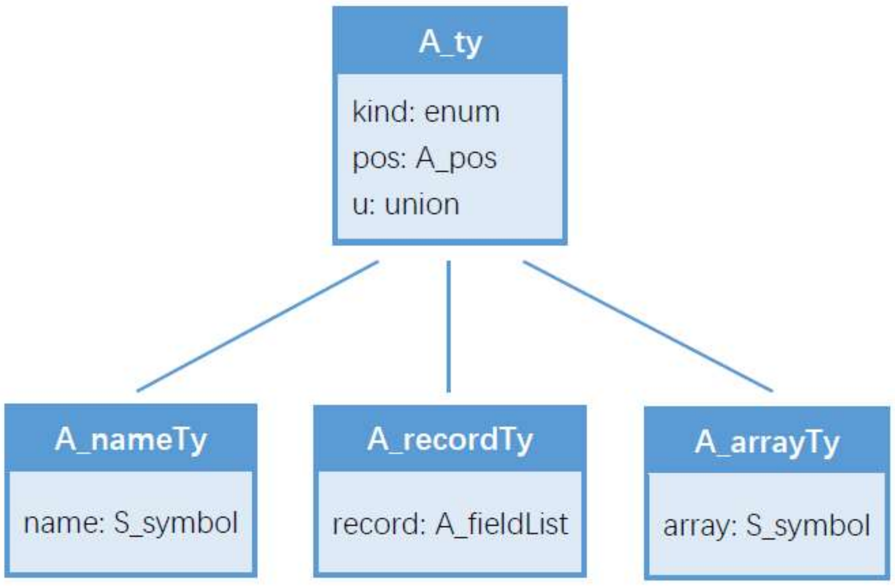


Figure 3.9 Structures of A_ty nodes

2.5 Generating Abstract Syntax Tree

When use Yacc to complete syntax analyzing, we could add some semantic actions behind every production. In these semantic actions, we call the constructor of corresponding kind of tree node, to generate an abstract syntax tree.

The semantic actions of the productions are as follows:

Production	Semantic action	Explanation
program → exp	absyn_root=$1;	The root of the tree
exp → lvalue	$$ = A_VarExp(EM_tokPos, $1);	a
exp → lvalue ASSIGN exp	$$ = A_AssignExp(EM_tokPos, $1, $3);	a:=1
exp → INT	$$ = A_IntExp(EM_tokPos, $1);	520
exp → STRING	$$ = A_StringExp(EM_tokPos, $1);	“iloveu”
exp → NIL	$$ = A_NilExp(EM_tokPos);	nil
exp →LPAREN RPAREN	$$ = A_SeqExp(EM_tokPos, NULL);	()
exp → LPAREN expseq RPAREN	LPAREN expseq RPAREN	(print(“hello”); print(“iloveu”))
exp → MINUS exp %prec UMINUS	$$ = A_OpExp(EM_tokPos, A_minusOp, A_IntExp(EM_tokPos, 0), $2);	-520
exp → exp PLUS exp	$$ = A_OpExp(EM_tokPos, A_plusOp, $1, $3);	x+y
exp → exp MINUS exp	$$ = A_OpExp(EM_tokPos, A_minusOp, $1, $3);	x-y
exp → exp TIMES exp	$$ = A_OpExp(EM_tokPos, A_timesOp, $1, $3);	x*y
exp → exp DIVIDE exp	$$ = A_OpExp(EM_tokPos, A_divideOp, $1, $3);	x/y
exp → exp EQ exp	$$ = A_OpExp(EM_tokPos, A_eqOp, $1, $3);	x=y
exp → exp NEQ exp	$$ = A_OpExp(EM_tokPos, A_neqOp, $1, $3);	x<>y
exp → exp LT exp	$$ = A_OpExp(EM_tokPos, A_ltOp, $1, $3);	x<y
exp → exp LE exp	$$ = A_OpExp(EM_tokPos, A_leOp, $1, $3);	x<=y
exp → exp GT exp	$$ = A_OpExp(EM_tokPos, A_gtOp, $1, $3);	x>y
exp → exp GE exp	$$ = A_OpExp(EM_tokPos, A_geOp, $1, $3);	x>=y
exp → exp AND exp	$$ = A_IfExp(EM_tokPos, $1, $3, A_IntExp(EM_tokPos, 0));	x&y
exp → exp OR exp	$$ = A_IfExp(EM_tokPos, $1, A_IntExp(EM_tokPos, 1), $3);	x|y
exp → funcall	$$ = $1;	print(“iloveu”)
exp → ID LBRACK exp RBRACK OF exp	$$ = A_ArrayExp(EM_tokPos, S_Symbol($1), $3, $6);	a[10] of int
exp → ID LBRACE RBRACE	$$ = A_RecordExp(EM_tokPos, S_Symbol($1), NULL);	list{}
exp → ID LBRACE asseq RBRACE	$$ = A_RecordExp(EM_tokPos, S_Symbol($1), $3);	list{a=1,b=2}
exp → IF exp THEN exp	$$ = A_IfExp(EM_tokPos, $2, $4, NULL);	if 1<2 then print(“iloveu”)
exp → IF exp THEN exp ELSE exp	$$ = A_IfExp(EM_tokPos, $2, $4, $6);	if 1<2 then a:=1 else a:=2
exp → WHILE exp DO exp	$$ = A_WhileExp(EM_tokPos, $2, $4);	while 1<2 do

		print(“iloveu”)
exp → FOR ID ASSIGN exp TO exp DO exp	$$ = A_ForExp(EM_tokPos, S_Symbol($2), $4, $6, $8);	for a:=520 to 1314 do print(“iloveu”)
exp → BREAK	$$ = A_BreakExp(EM_tokPos);	break
exp → LET decs IN END	$$ = A_LetExp(EM_tokPos, $2, A_SeqExp(EM_tokPos, NULL));	let love:=520 in end
exp → LET decs IN expseq END	$$ = A_LetExp(EM_tokPos, $2, A_SeqExp(EM_tokPos, $4));	let love:=520 in print(“iloveu”) end
exp → LPAREN error RPAREN	$$ = A_SeqExp(EM_tokPos, NULL);	
exp → error SEMICOLON exp	$$ = $3;	
lvalue → ID	$$ = A_SimpleVar(EM_tokPos,S_Symbol($1));	a
lvalue → lvalue DOT ID	$$ = A_FieldVar(EM_tokPos, $1, S_Symbol($3));	a.value
lvalue → lvalue LBRACK exp RBRACK	$$ = A_SubscriptVar(EM_tokPos, $1, $3);	a[i+1]
lvalue → ID LBRACK exp RBRACK	$$ = A_SubscriptVar(EM_tokPos, A_SimpleVar(EM_tokPos, S_Symbol($1)), $3);	x[i]
expseq → exp	$$ = A_ExpList($1, NULL);	a+b
expseq → exp SEMICOLON expseq	$$ = A_ExpList($1, $3);	a+2; x+y
funcall → ID LPAREN RPAREN	$$ = A_CallExp(EM_tokPos, S_Symbol($1), NULL);	myprint()
funcall → ID LPAREN paraseq RPAREN	$$ = A_CallExp(EM_tokPos, S_Symbol($1), $3);	print(a)
paraseq → exp	$$ = A_ExpList($1, NULL);	a
paraseq → exp COMMA paraseq	$$ = A_ExpList($1, $3);	x, y
asseq → ID EQ exp	$$ = A_EfieldList(A_Efield(S_Symbol($1), $3), NULL);	a=520
asseq → ID EQ exp COMMA asseq	$$ = A_EfieldList(A_Efield(S_Symbol($1), $3), $5);	a=2, b=3
decs → dec decs	$$ = A_DecList($1, $2);	
decs →	$$ = NULL;	
dec → tydecs	$$ = A_TypeDec(EM_tokPos, $1);	
dec → vardec	$$ = $1;	
dec → fundecs	$$ = A_FunctionDec(EM_tokPos, $1);	
tydecs → tydec	$$ = A_NametyList($1, NULL);	
tydecs → tydec tydecs	$$ = A_NametyList($1, $2);	
tydec → TYPE ID EQ ty	$$ = A_Namety(S_Symbol($2), $4);	type love = int
ty → ID	$$ = A_NameTy(EM_tokPos, S_Symbol($1));	love
ty → LBRACE tyfields RBRACE	$$ = A_RecordTy(EM_tokPos, $2);	{a:int, b:string}
ty → ARRAY OF ID	$$ = A_ArrayTy(EM_tokPos, S_Symbol($3));	array of int
tyfields → tyfield	$$ = $1;	
tyfields →	$$ = NULL;	
tyfield → ID COLON ID	$$ = A_FieldList(A_Field(EM_tokPos, S_Symbol($1), S_Symbol($3)), NULL);	key: int
tyfield → ID COLON ID COMMA tyfield	$$ = A_FieldList(A_Field(EM_tokPos, S_Symbol($1), S_Symbol($3)), $5);	a:int, b: string
vardec → VAR ID ASSIGN exp	$$ = A_VarDec(EM_tokPos,	var love:=520

	S_Symbol($2), NULL, $4);	
vardec → VAR ID COLON ID ASSIGN exp	$$ = A_VarDec(EM_tokPos, S_Symbol($2), S_Symbol($4), $6);	var a: int := 520
fundecs → fundec	$$ = A_FundecList($1, NULL);	
fundecs → fundec fundecs	$$ = A_FundecList($1, $2);	
fundec → FUNCTION ID LPAREN tyfields RPAREN EQ exp	$$ = A_Fundec(EM_tokPos, S_Symbol($2), $4, NULL, $7);	function love(time: int) = print("iloveu")
fundec → FUNCTION ID LPAREN tyfields RPAREN COLON ID EQ exp	$$ = A_Fundec(EM_tokPos, S_Symbol($2), $4, S_Symbol($7), $9);	function love(time: int) : int = print("iloveu")

Figure 3.10 Semantic actions of productions

3. Semantic Analysis

After the syntax analysis, we get an abstract tree. This tree will be transmitted to "Semant" module to do the "semantic analysis" whose main jobs are to construct environment tables and type check. In "Semant" module, we will call the "Translate" module to generate the intermediate code.

3.1 Data Structure

The main data structure used in this period is the symbol table TAB_table which is an encapsulation of the S_table. There exists two type of tables: value table and type table. Type environment stores the mapping relationship from S_Symbol to Ty_ty, value environment stores the mapping relationship from S_Symbol to E_enventry (E_eventry represents the structure of a variable or a function stored in the symbol table, Ty_ty represents the structure of type stored in symbol table).

3.2 Related module description

The main job of semantic analysis period is to manage the symbol table which is equal to the management of environment. Its role is to mapping identifiers to their types and the storing place. Environment consists of some bindings which is used to described the mapping relationship between identifiers and its meaning. Every local variables have their own scopes, they are only visible in their scopes.

Table module:

This module implements a general table "TAB_table" which maps a pointer key to a binding type. The structure "binder_" ,the basic structure to store the binding, is as follows: "key" is the symbol pointer key, "value" is the actual value of symbol, "binder" is a structure pointer pointing to the next binder_ structure, "prevtop" is the most recent inserted key. "key" and "value" domains are both type "void*" because type and variable will both be stored in table.

```
struct binder_ {
    void *key;
    void *value;
    binder next;
    void *prevtop;
};
```
This module implements some basic operations of table including creating a new table, inserting binding relationship, delete binding relationship, loop up binding relationship in table and call a function on all the items in this table.

Symbol module:

This module transfers every variable or type into a symbol whose definition is as follows: 'name' is the name of this symbol and 'S_symbol' is a structure pointing to the next symbol.

```
struct S_symbol_ {
    string name;
    S_symbol next;
};
```

This module is based on "table" module: S_Symbol will be stored in the table structure. Some operations including creating new table, inserting a new binding, deleting a new binding and looping up in the table are encapsulated. Besides this basic operations, function "S_beginScope()" and function "S_endScope()" are defined to implement the action scope. The way is inserting a "mark" pointer into table when a new scope starts and popping all the pointers when a scope finishes.

Types module:

This module implements "type" object. There are two basic types: int and string. Other types includes array, record and types implemented by basic types. The structure "Ty_ty_" representing each type is defined as follows: "kind" indicates the kind of this type, "u" is the detailed attribution of this type. The mentioned types all has their corresponding function to create a new type object.

```
struct Ty_ty_ {enum {Ty_record, Ty_nil, Ty_int, Ty_string, Ty_array,
                Ty_name, Ty_void} kind;  /*kind of this type*/

          /*different types have different attributions*/
          union {Ty_fieldList record;
              Ty_ty array;
              struct {S_symbol sym; Ty_ty ty;} name;
            } u;
       };
```

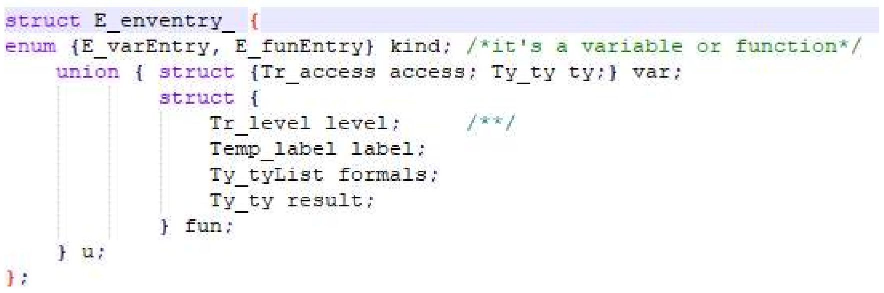
In order to deal with the recursive declaration problem, there is a manually designed type "Ty_Name" to act as a placeholder. Type "Ty_Nil" represents the expression "nil" in record type, type "Ty_Void" represents the void return type of expression.

Temp module:

This module is used to store temporary variables before they enter the registers. The "table"structure defined in "table" module is needed in this module. Some basic operations of this table are provided including creating an empty table storing temporary variables, insert new temporary variables into table and looking up in this table.

Env module:

Type environment and value environment are defined in this module. Structure "E_eventry_" is defined as follows: "kind" indicates it is a variable or function, u contains the detailed attributions of variables or functions. Attribution of variables contains the binding symbol and value; attribution of functions contains the level of function, symbol of this function, formal parameters and return type.

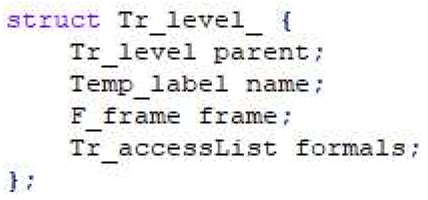


```
struct E_enventry_ {
enum {E_varEntry, E_funEntry} kind; /*it's a variable or function*/
    union { struct {Tr_access access; Ty_ty ty;} var;
            struct {
                Tr_level level;     /**/
                Temp_label label;
                Ty_tyList formals;
                Ty_ty result;
            } fun;
    } u;
};
```

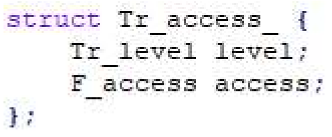
Predefined types “int” and “string” are defined in this module, predefined functions “print”、”flush” and “getchar” are also defined in this module.

Translate module:

This module is responsible for the generation of middle tree. Initialization functions for different types of nodes of middle tree are defined. Basic node structure in “tree” module is used to create middle tree. Two important structures are defined like follows. “Tr_level_”represents the nested level, “Tr_access_” represents a variable indicating its level and storage type and place.



```
struct Tr_level_ {
    Tr_level parent;
    Temp_label name;
    F_frame frame;
    Tr_accessList formals;
};
```



```
struct Tr_access_ {
    Tr_level level;
    F_access access;
};
```

Semant module:

Function of this module is semantics analysis(type examination). functions “transVar”, “transExp” and “transDec” are responsible for semantics analysis and transformation to intermediate code. These three functions needs type environment、value environment and the corresponding source code. Function “transVar” checks if the type simple variable、object and array is consonant with the value. Function “transDec” examines the declaration of variables and functions. Function “transExp” will call beginScoe() firstly to mark current state(type environment and value environment). Then “transDec” are called to enlarge the environment and function body expressions are transformed. At last, function endscope() is called to restore the state.

3.3 Type check

Abstract Tree Node	Check rules
A_SimpleVar	1. loop up the value environment， variable needs to exist and itsreturn value is E_varEntry.
A_FieldVar	1. loop up the value environment， variable needs to exist and itsreturn value is E_varEntry. 2. The type is Ty_record 3. Look up every domain, at least one domain matches the current requested domain.

A_SubscriptVar	1. loop up the value environment， variable needs to exist and itsreturn value is E_varEntry. 2. The type is Ty_array 3. Type of subscript is Ty_int
A_VarExp	No need
A_NilExp	No need
A_VoidExp	No need
A_IntExp	No need
A_SeqExp	No need
A_AssignExp	1. types of left value and right value are the same 2. Left value is not Ty_record 3. Type of right value is not Ty_nil
A_IfExp	1. If there is no else statements, the return type of statement “then” is not Ty_void 2. If there exists else, the return types of then and else need to be the same.
A_WhileExp	1. The return type of while body should be Ty_void
A_ForExp	1. The types of initial and final values should both be Ty_int
A_BreakExp	1. Set a stack, whenever enter a loop push a Label, pop it out when jump out of the loop. 2. Stack should not be empty
A_LetExp	No need
A_ArrayExp	1. Look up the type environment, the return variable should exist and its type should be Ty_array. 2. The size should be Ty_int. 3. The array type in type environment should be the same as actual type.
A_FunctionDec	1. Loop up the value environment, the variable should exist and its type is E_FuncEntry 2. The actual return type of function should be the same as the declaration.
A_VarDec	1. If there is no type declaration, the initial value should not be Ty_nil 2. If there exists declaration, lookup the type environment table to check if it’s matched.

A_TypeDec	1. Look up the type environment table to check if it’s already been declared.
A_NameTy	1. Look up the type environment, the variable should exist and its return type is not Ty_name.
A_RecordTy	1. Look up the type environment, every domain should exist in type environment.
A_ArrayTy	1. Look up the type environment, the variable should exist and its return type is not Ty_name.

4.Middle Code

4.1 Main Data Structure

4.1.1 Frame

F_frame structure is used to described a stack frame, whenever a function is called a new frame will be generated. Every frame should include the initial address, Temp_label, size of frame, and every arguments.

```
struct F_frame_
{
Temp_label              name;
F_accessList             formals;
int                     frame_size;

};
```

F_access structure is used to described variables in stack frame. Kind Is used to described the variable is stored in registers or in frame, u stores the number of registers or the offset in frame.

```
struct F_access_
{
enum {inFrame, inReg} kind;
union {
int                     offset;  /* InFrame, at offset X from the fp */
Temp_temp                reg;    /* InReg, the register name */
} u;
```

4.1.2 Tr_level

Tr_level is an encapsulation of F_frame, it's mainly used to solve the static link. To solve it,there is a parent pointer in Tr_level structure pointing to the parent pointer. When function needs to look up variables by static link, Tr_level can provide enough information to find this variable.

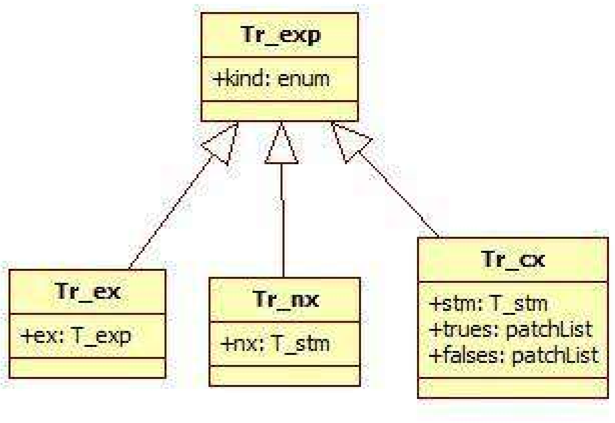
```
struct Tr_level_
{
Temp_label              name;
Tr_level               parent;
Tr_accessList          formals;
F_frame                frame;
};
```

Tr_access represents a variable in Tr_level. Tr_access structure consists of F_access and the level information of this variable.

```
    struct Tr_access_
{
Tr_level            level;
F_access            access;
};
```

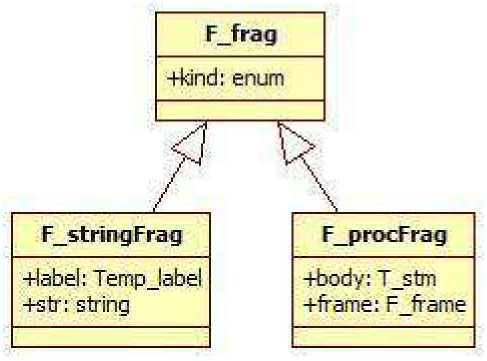
4.1.3 Tr_exp

In Translate module we define a structure Tr_exp to express A_ecp in abstract statements. There are three types of Tr_exp: 1. Exression with return value Tr_exp; 2. Statement without return value Tr_nx; Condition statements Tr_ex.



4.1.4 F_frag

Translate periods generate description for every function including the frame information and body of function, that is the F_frocFrag in F_frag. In the same way, information of string will be encapsulated in F_stringFrag including the name and initial address of label. Translate preserves a private table, fragments can be accessed by Tr_getResult().

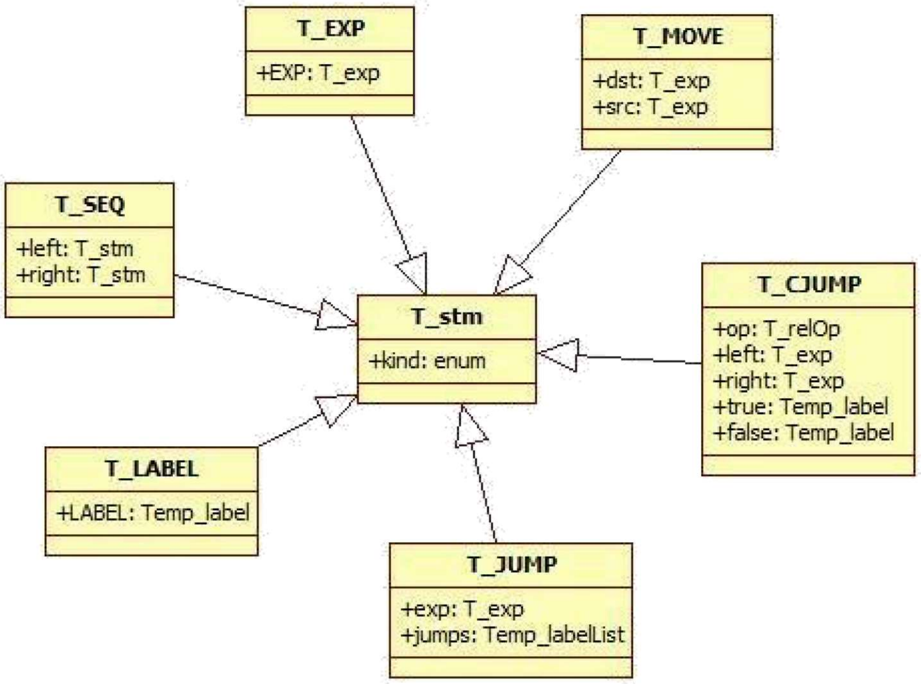


4.2 IR Tree

IR is an abstract machine language, it can describe the machine operations without too much contact With machine details. It is a bridge between the machine code and abstract tree. The main job of Translate Is to generate IR tree. There are two kinds of nodes in IR tree: 1. T_exp with return value; 2.T_stm without Return value.

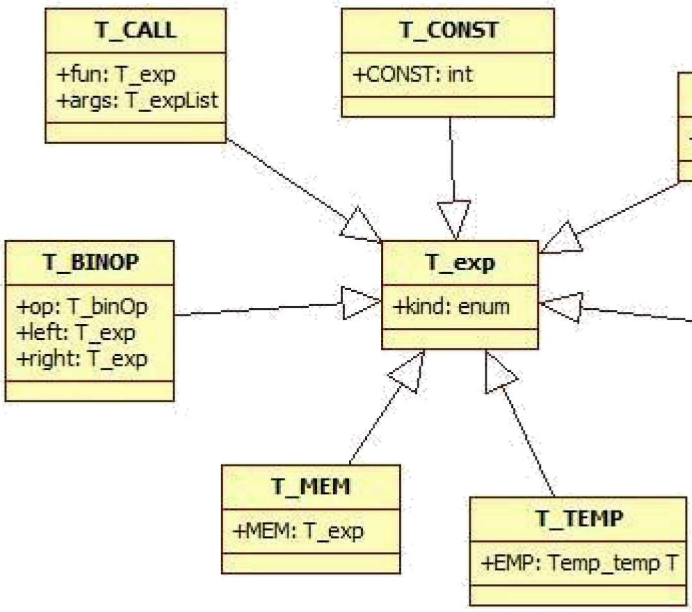
T_stm can be divided into T_SEQ, T_LABEL, T_JUMP, T_CJUMP, T_MOVE

Middle code	Description	Node
MOVE (e1, e2)	Move value of e2 into the Memory pointed by e1	T_Move(T_exp, T_exp)
EXP (e)	compute e	T_stm T_Exp(T_exp)
JUMP (e, labs)	Jump to the first label in labels without condition	T_Jump(T_exp exp, Temp_labelList labels)
CJUMP (o, e1, e2, t, f)	Compute the value of e1 and e2, Compute them with operator o, If it's true jump to t, otherwise f	T_Cjump(T_relOp op, T_exp left, T_exp right, Temp_label true, Temp_label false)
SEQ (s1, s2)	Execute s1 then s2	T_Seq(T_stm left, T_stm right)
LABEL (n)	Define label n as the current address	T_Label(Temp_label)



Structure T_exp can be divided into T_BINOP, T_MEM, T_TEMP, T_ESEQ, T_NAME, T_CONST, T_CALL

Middle code	Description	Node
CONST (i)	Const value i	T_Const(int)
NAME (n)	Label n	T_Name(Temp_label)
TEMP (t)	Temporary variable	T_Temp(Temp_temp)
BINOP(o,t1,t2)	Compute t1 and t2 with the operator o	T_Binop(T_binOp, T_exp, T_exp)
MEM (e)	Extract the e bytes contents from Memory address e	T_Mem(T_exp)
CALL (f, l)	Call function f with l as the argument list	T_Call(T_exp, T_expList)
ESEQ (s, e)	Compute statements, then compute expression e	T_Eseq(T_stm, T_exp)



4.3 Generate Middle Tree

The whole program of tiger language can be viewed as an expression, we transmit the abstract tree to transExp() , this function will execute type check and call Translate module to translate it into middle code. The middle code in main body will returned directly, and the middle code of function and string will be saved in F_frag structure.

In translate module ,we transformed the node in abstract tree into Tr_exp structure, Then retranslate it into Node in IR tree, that is T_exp and T_stm. Then printree module will output the IR tree.

Function in Translate module:

Function	Abstract tree node	Implementation method
Tr_simpleVar()	A_simpleVar	Find fp and the offset in frame of thisvariable.
Tr_fieldVar()	A_fieldVar	Get byte offset by base , word offset and word size

Tr_subscriptVar()	A_subscriptVar	Get byte offset by base , index and word size
Tr_noExp()	A_nilExp; A_voidExp;…	return T_Const(0)
Tr_intExp()	A_intExp	Construct T_CONST Node
Tr_stringExp()	A_stringExp	Add new F_StringFrag in table
Tr_callExp()	A_callExp	Get the static link of function，transform Tr_explist into T_explist, then call T_Call() to construct IR Tree.
Tr_arithopExp()	A_opExp	Construct T_BINOP node by（+-*/）
Tr_logicExp()	A_opExp	Transform logic operation into condition statement.Construct node Tr_Cx by (^&|)
Tr_eqopExp()	A_opExp	With operators（ =,≠,<,>,<=,>= ）, construct T_CJUMP node
Tr_eqstringExp()	A_opExp	Call stringEqual()
Tr_recordExp()	A_recordExp	Allocate storage space, temporary variable will be stored in memory. The temporary variable will be returned to indicate the record address.
Tr_seqExp()	A_seqExp; A_letExp	Construct　T_Seq node，return Tr_Nx or Tr_Ex According to if it has return value.
Tr_assignExp()	A_assignExp; A_varDec	Call unEx() to deal with left value and right value to aconstruct Tr_Nx node.
Tr_ifExp()	A_ifExp	Distinguish if else exists and if then statement and else statement has return value
Tr_doneExp()	A_whileExp	Allocate a Label and return，record the end address of

		while loop
Tr_whileExp()	A_whileExp	Construct T_Jump node, record loop starting address and end address, and put into loop body.
Tr_breakExp()	A_breakExp	Construct T_Jump node and jump to the start address of start address of loop
Tr_arrayExp()	A_arrayExp	call initArray ()
Tr_procEntryExit()	A_functionDec	Add F_ProcFrag in fragment table

5. Trace Produce

After a few previous steps, we finally get a middle tree. What's more has to be done is to convert the Intermediate Representation Tree to assembly language or machine language. However, there are certain aspects aspects of the Tree language that do not correspond exactly with machine languages, and some aspects of the Tree language interfere with compile-time optimization analyses.

To deal with this problem, we should perform an extra step, to translate the original Tree to another form, which is easy to translate to machine language.

5.1 Data Structure

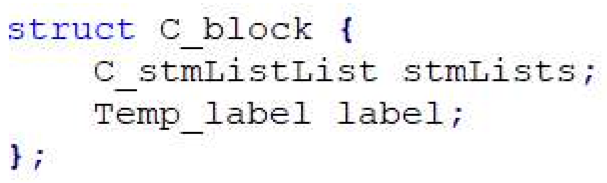
The main data structure used in this period is the following.

```
typedef struct expRefList_ *expRefList;
struct expRefList_ {
    T_exp *head;
    expRefList tail;
};
```

This structure expRefList_ is used to hold a list of several expressions. The implement for this is extremly simple, since we only need to get the pointer to a specific expression, with no any other information. The T_exp *head member is a pointer headed to the head of the list, and the expRefList tail member is pointed to the remaining of the list, which is still a expRefList_.

```
typedef struct C_stmListList_ *C_stmListList;
struct C_stmListList_ {
    T_stmList head;
    C_stmListList tail;
};
```

This structure C_stmListList_ is used to hold a list of handles of T_stmList_, which is already defined in tree.h. This structure is used for generating the basic blocks. For a block contains several statements and a procedure contains several blocks. So if we try to divide a whole procedure into blocks, we need a structure to hold a list of T_stmList.



```
struct C_block {
    C_stmListList stmLists;
    Temp_label label;
};
```

The structure to hold a basic block, contains a C_stmListList and a label to identify the block.

5.2 Related module description

The process of generating a trace can be divided into two steps: first we generate a canonial tree and then process the conditional branch.

Build canonial tree:

A "canonial tree" is a tree without ESEQs and SEQs, at the mean time the parent node of a CALL is either EXP(…) or MOVE(TEMP, …). We use the following method do_stm to processes stm so that it contains no ESEQ nodes.

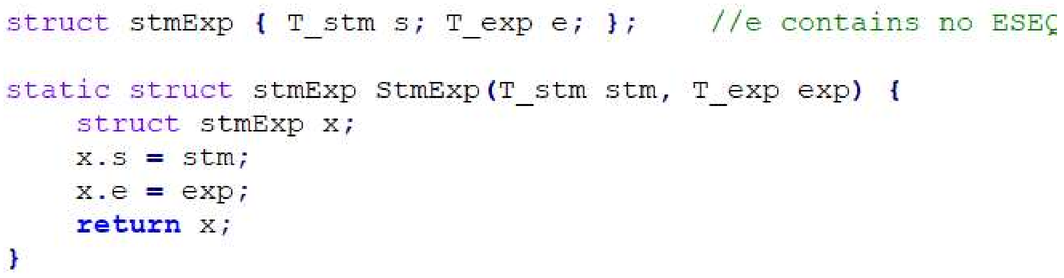
We follow the algrithom in the textbook.

- Step 1. make a "subexpression-extraction" method for each kind.

function record:

```
/*
* INPUT: a exp list,
* PROCESS:extract ESEQs and merge the results
* OUTPUT: a large T_stm
*/
static T_stm reorder(expRefList rlist)
{
   if (!rlist)
      return T_Exp(T_Const(0)); // nop, return a CONST 0

   if ((*rlist->head)->kind == T_CALL) {    //A CALL
      Temp_temp t = Temp_newtemp();
      *rlist->head = T_Eseq(T_Move(T_Temp(t), *rlist->head), T_Temp(t)); //ESEQ(MOVE(TEMP t, CALL(fun, args)), Temp t)
      return reorder(rlist);
   }
   else
   {
      struct stmExp hd = do_exp(*rlist->head); //extract EXEQ from the head, get a (stm, exp) ,exp with no ESEQ
      T_stm s = reorder(rlist->tail);  //do the same thing to the tail, at last we get a stm
      if (commute(s, hd.e))
      {       //if exchangable
         *rlist->head = hd.e; //exchange
         return seq(hd.s, s);    //merge results
      }
      else
      {
         Temp_temp t = Temp_newtemp();
         *rlist->head = T_Temp(t);
         return seq(hd.s, seq(T_Move(T_Temp(t), hd.e), s));  //SEQ(s, SEQ(MOVE(TEMP t, e), s))
      }
   }
}
```

This function gets a expression list, extract the ESEQs out of the list and merge the statement parts of these ESEQ into a big T_stm. The input of the function is the alias (referrences) of all the child node of a certain node. The implement of recorder function is simple. It calls unon an auxiliary function do_exp on each expression in the list. The auxiliary function pull the ESEQs out of a expression and return a struct stmExp.



```
struct stmExp { T_stm s; T_exp e; };     //e contains no ESE

static struct stmExp StmExp(T_stm stm, T_exp exp) {
     struct stmExp x;
     x.s = stm;
     x.e = exp;
     return x;
}
```

Struct stmExp.s is the statement parts of the origin expression, and stmExp.e is a rewritten expression without ESEQs.

- Step 2. make a “subexpression-insertion” method

function do_exp:

```
/* change an exp to an (stm, exp), where exp contains no ESEQ*/
static struct stmExp do_exp(T_exp exp)
```

The implement of the function do_exp is the following. First, check the kind of the expression. Second, if the kind is not ESEQ, make a list of the subexpression references and calls recorder function.

function do_stm:

```
/*
* extract all ESEQs in a stm
*/
static T_stm do_stm(T_stm stm);
```

When we already have record function, we can implete do_stm function to pull ESEQs in a stm. In this function, we simply perform the same method in the do_exp function. Note that for ESEQ node as a expression header, we should perform a do_stm method to the statement part.

```
static struct stmExp do_exp(T_exp exp)
{
   switch (exp->kind) {
   case T_BINOP:
      ...
   case T_MEM:
      ...
   case T_ESEQ:
   {
       struct stmExp x = do_exp(exp->u.ESEQ.exp);
    return StmExp(seq(do_stm(exp->u.ESEQ.stm), x.s), x.e);
   }
    ...
}
```

- Step 3. linearize

After processed do_stm function, we successfully get a tree where all the SEQ nodes are near the top. Based on the following rewrite rule, we can change the tree to a simple list.

```
SEQ(SEQ(a, b), c) = SEQ(a, SEQ(b,c))
```

The linearize function simply perform the rule repeatly.

```
/* build a statement list */
static T_stmList linear(T_stm stm, T_stmList right)
{
   if (stm->kind == T_SEQ)
      return linear(stm->u.SEQ.left,
            linear(stm->u.SEQ.right,
            right));
   else return T_StmList(stm, right);
}
```

Taming conditional branches:

Analyzing the program’s control flow is an important method for determining where the jumps go in a program. We can lump together any sequence of no-branch instuctions into a basic block and analyze the control flow between basic blocks.

● Step 1. form basic blocks

Function Blocks make the beginning of a basic block as following.

```c
/* make the beginning of a basic block a LABEL */
static C_stmListList Blocks(T_stmList stms, Temp_label lb)
{
    if (!stms) {
        return NULL;
    }
    if (stms->head->kind == T_LABEL) //if the head is already a LABEL
    {
        return StmListList(stms, next(stms, stms->tail, lb)); //we simply put to the StmListList, and get the next

    }

    return Blocks(
        T_StmList(
            T_Label(Temp_newlabel()),   //get a new LABEL
            stms),
        lb);
}
```

The algorithem is: if the head is already a LABEL, we go down the list and search for the branch node, and use the structure stmListList to hold the address of the next Block; if the head is not a LABEL, we create a new LABEL as the head, and recursively calls function Blocks itself.

● Step 2. order the basic blocks into a trace

We can choose an ordering of the blocks satisfying the condition that each CJUMP is followed by its false label.

function getNext:

```c
/*
* get the next block from the list of stmLists, using only those that have
* not been traced yet
*/
static T_stmList getNext()
{
    if (!global_block.stmLists)
        return T_StmList(T_Label(global_block.label), NULL);
    else {
        T_stmList s = global_block.stmLists->head;
        if (S_look(block_env, s->head->u.LABEL)) {/* label exists in the table */
            trace(s);
            return s;
        }
        else {
            global_block.stmLists = global_block.stmLists->tail;
            return getNext();
        }
    }
}
```

The function impletes the algorithm in the textbook.

6.Target Code Generation

6.1 Abstract assembly-language instructions

We define a series of abstract assembly-language instuctions as following, and each type a constuction function.

Type	Description
AS_targets	the abstract form of the target label
AS_instr	the abstract form of an instruction in machine language, containing OPER, LABEL and MOVE three kinds
AS_instrList	the list of AS_instrLists
AS_proc	the abstract form of a procedure

6.2 i386 instuction generation

We assume 386 as our target machine, so we generate x86 code.

We implement the Maximal Munch algorithm in the textbook. For each of the abstract assembly-language types, we attached it with x86 instructions.

The following is the Maximal Munch performed on the T_stm.

```
static void munchStm(T_stm s) {
    assert(s != NULL);
    switch (s->kind) {
    case T_MOVE: {
        if (s->u.MOVE.dst->kind == T_TEMP
            && s->u.MOVE.src->kind == T_MEM
            && s->u.MOVE.src->u.MEM->kind == T_BINOP
            && s->u.MOVE.src->u.MEM->u.BINOP.op == T_plus
            && s->u.MOVE.src->u.MEM->u.BINOP.right->kind == T_CONST) {
            Temp_temp r1, r2;
            int con;
            char buf[MAXINSTRLEN];
            r1 = s->u.MOVE.dst->u.TEMP;
            r2 = munchExp(s->u.MOVE.src->u.MEM->u.BINOP.left);
            con = s->u.MOVE.src->u.MEM->u.BINOP.right->u.CONST;
            if (con < 0) {
                con = -con;
                sprintf(buf, "mov `d0, [`s0-%d]\n", con);
            }
            else {
                sprintf(buf, "mov `d0, [`s0+%d]\n", con);
            }
            emit(AS_Oper(String(buf), Temp_TempList(r1, NULL),
                Temp_TempList(r2, NULL), NULL));
            return;
        }
        else if (s->u.MOVE.dst->kind == T_TEMP
            && s->u.MOVE.src->kind == T_MEM
            && s->u.MOVE.src->u.MEM->kind == T_BINOP
            && s->u.MOVE.src->u.MEM->u.BINOP.op == T_plus
            && s->u.MOVE.src->u.MEM->u.BINOP.left->kind == T_CONST) {
            Temp_temp r1, r2;
            int con;
            char buf[MAXINSTRLEN];
            r1 = s->u.MOVE.dst->u.TEMP;
            r2 = munchExp(s->u.MOVE.src->u.MEM->u.BINOP.right);
            con = s->u.MOVE.src->u.MEM->u.BINOP.left->u.CONST;
            if (con < 0) {
                con = -con;
                sprintf(buf, "mov `d0, [`s0-%d]\n", con);
            }
            else {
                sprintf(buf, "mov `d0, [`s0+%d]\n", con);
            }
            emit(AS_Oper(String(buf), Temp_TempList(r1, NULL),
                Temp_TempList(r2, NULL), NULL));
            return;
        }
        else if (s->u.MOVE.dst->kind == T_TEMP
            && s->u.MOVE.src->kind == T_MEM) {
            Temp_temp r1, r2;
            r1 = s->u.MOVE.dst->u.TEMP;
            r2 = munchExp(s->u.MOVE.src->u.MEM);
            emit(AS_Oper("mov `d0, [`s0]\n", Temp_TempList(r1, NULL),
                Temp_TempList(r2, NULL), NULL));
            return;
        }
        else if (s->u.MOVE.dst->kind == T_TEMP
            && s->u.MOVE.src->kind == T_CONST) {
            Temp_temp r1;
            int con;
            char buf[MAXINSTRLEN];
            r1 = s->u.MOVE.dst->u.TEMP;
            con = s->u.MOVE.src->u.CONST;
            sprintf(buf, "mov `d0, %d\n", con);
            emit(AS_Oper(String(buf), Temp_TempList(r1, NULL), NULL, NULL));
            return;
        }
        else if (s->u.MOVE.dst->kind == T_TEMP) {
            Temp_temp r1, r2;
            r1 = s->u.MOVE.dst->u.TEMP;
            r2 = munchExp(s->u.MOVE.src);
            emit(AS_Move("mov `d0, `s0\n", Temp_TempList(r1, NULL),
                Temp_TempList(r2, NULL)));
            return;
        }
        else if (s->u.MOVE.dst->kind == T_MEM
            && s->u.MOVE.dst->u.MEM->kind == T_BINOP
            && s->u.MOVE.dst->u.MEM->u.BINOP.op == T_plus
            && s->u.MOVE.dst->u.MEM->u.BINOP.right->kind == T_CONST) {
            Temp_temp r1, r2;
            char buf[MAXINSTRLEN];
            int con;
            r1 = munchExp(s->u.MOVE.dst->u.MEM->u.BINOP.left);
            r2 = munchExp(s->u.MOVE.src);
            con = s->u.MOVE.dst->u.MEM->u.BINOP.right->u.CONST;
            if (con < 0) {
                con = -con;
                sprintf(buf, "mov [`s0-%d], `s1\n", con);
            }
            else {
                sprintf(buf, "mov [`s0+%d], `s1\n", con);
            }
            emit(AS_Oper(String(buf), NULL, Temp_TempList(r1,
                Temp_TempList(r2, NULL)), NULL));
            return;
        }
        else if (s->u.MOVE.dst->kind == T_MEM
            && s->u.MOVE.dst->u.MEM->kind == T_BINOP
            && s->u.MOVE.dst->u.MEM->u.BINOP.op == T_plus
            && s->u.MOVE.dst->u.MEM->u.BINOP.left->kind == T_CONST) {
            Temp_temp r1, r2;
            char buf[MAXINSTRLEN];
            int con;
            r1 = munchExp(s->u.MOVE.dst->u.MEM->u.BINOP.right);
            r2 = munchExp(s->u.MOVE.src);
            con = s->u.MOVE.dst->u.MEM->u.BINOP.left->u.CONST;
```

```c
            if (con < 0) {
                con = -con;
                sprintf(buf, "mov [`s0-%d], `s1\n", con);
            }
            else {
                sprintf(buf, "mov [`s0+%d], `s1\n", con);
            }
            emit(AS_Oper(String(buf), NULL, Temp_TempList(r1,
                Temp_TempList(r2, NULL)), NULL));
            return;
        }
        else if (s->u.MOVE.dst->kind == T_MEM) {
            Temp_temp r1, r2;
            r1 = munchExp(s->u.MOVE.dst->u.MEM);
            r2 = munchExp(s->u.MOVE.src);
            emit(AS_Oper("mov [`s0], `s1\n", NULL, Temp_TempList(r1,
                Temp_TempList(r2, NULL)), NULL));
            return;
        }
        else {
            assert(0);
        }
    }
    case T_EXP: {
        munchExp(s->u.EXP);
        return;
    }
    case T_LABEL: {
        char buf[MAXINSTRLEN];
        sprintf(buf, "%s:\n", S_name(s->u.LABEL));
        emit(AS_Label(String(buf), s->u.LABEL));
        return;
    }
    case T_JUMP: {
        char buf[MAXINSTRLEN];
        Temp_label lab;
        assert(s->u.JUMP.exp->kind == T_NAME);
        lab = s->u.JUMP.exp->u.NAME;
        sprintf(buf, "jmp near %s\n", S_name(lab));
        emit(AS_Oper(String(buf), NULL, NULL, AS_Targets(Temp_LabelList(lab, NULL))));
        return;
    }
    case T_CJUMP: {
        char buf[MAXINSTRLEN];
        Temp_temp r1, r2;
        Temp_label lab;
        Temp_labelList labs;
        lab = s->u.CJUMP.true;
        labs = Temp_LabelList(s->u.CJUMP.true, Temp_LabelList(s->u.CJUMP.false, NULL));
        r1 = munchExp(s->u.CJUMP.left);
        r2 = munchExp(s->u.CJUMP.right);
        switch (s->u.CJUMP.op) {
        case T_eq: {
            emit(AS_Oper("cmp `s0, `s1\n", NULL, Temp_TempList(r1,
                Temp_TempList(r2, NULL)), NULL));
            sprintf(buf, "je near %s\n", S_name(lab));
            emit(AS_Oper(String(buf), NULL, NULL, AS_Targets(labs)));
            return;
        }
        case T_ne: {
            emit(AS_Oper("cmp `s0, `s1\n", NULL, Temp_TempList(r1,
                Temp_TempList(r2, NULL)), NULL));
            sprintf(buf, "jne near %s\n", S_name(lab));
            emit(AS_Oper(String(buf), NULL, NULL, AS_Targets(labs)));
            return;
        }
        case T_lt: {
            emit(AS_Oper("cmp `s0, `s1\n", NULL, Temp_TempList(r1,
                Temp_TempList(r2, NULL)), NULL));
            sprintf(buf, "jl near %s\n", S_name(lab));
            emit(AS_Oper(String(buf), NULL, NULL, AS_Targets(labs)));
            return;
        }
        case T_le: {
            emit(AS_Oper("cmp `s0, `s1\n", NULL, Temp_TempList(r1,
                Temp_TempList(r2, NULL)), NULL));
            sprintf(buf, "jle near %s\n", S_name(lab));
            emit(AS_Oper(String(buf), NULL, NULL, AS_Targets(labs)));
            return;
        }
        case T_gt: {
            emit(AS_Oper("cmp `s0, `s1\n", NULL, Temp_TempList(r1,
                Temp_TempList(r2, NULL)), NULL));
            sprintf(buf, "jg near %s\n", S_name(lab));
            emit(AS_Oper(String(buf), NULL, NULL, AS_Targets(labs)));
            return;
        }
        case T_ge: {
            emit(AS_Oper("cmp `s0, `s1\n", NULL, Temp_TempList(r1,
                Temp_TempList(r2, NULL)), NULL));
            sprintf(buf, "jge near %s\n", S_name(lab));
            emit(AS_Oper(String(buf), NULL, NULL, AS_Targets(labs)));
            return;
        }
        case T_ult: {
            emit(AS_Oper("cmp `s0, `s1\n", NULL, Temp_TempList(r1,
                Temp_TempList(r2, NULL)), NULL));
            sprintf(buf, "jb near %s\n", S_name(lab));
            emit(AS_Oper(String(buf), NULL, NULL, AS_Targets(labs)));
            return;
        }
        case T_ule: {
            emit(AS_Oper("cmp `s0, `s1\n", NULL, Temp_TempList(r1,
                Temp_TempList(r2, NULL)), NULL));
            sprintf(buf, "jbe near %s\n", S_name(lab));
            emit(AS_Oper(String(buf), NULL, NULL, AS_Targets(labs)));
            return;
        }
        case T_ugt: {
            emit(AS_Oper("cmp `s0, `s1\n", NULL, Temp_TempList(r1,
                Temp_TempList(r2, NULL)), NULL));
            sprintf(buf, "ja near %s\n", S_name(lab));
            emit(AS_Oper(String(buf), NULL, NULL, AS_Targets(labs)));
            return;
        }
        case T_uge: {
            emit(AS_Oper("cmp `s0, `s1\n", NULL, Temp_TempList(r1,
                Temp_TempList(r2, NULL)), NULL));
            sprintf(buf, "jae near %s\n", S_name(lab));
            emit(AS_Oper(String(buf), NULL, NULL, AS_Targets(labs)));
            return;
        }
        default: {
            assert(0);
        }
        }
    }
    default: {
        assert(0);
    }
    }
}
```

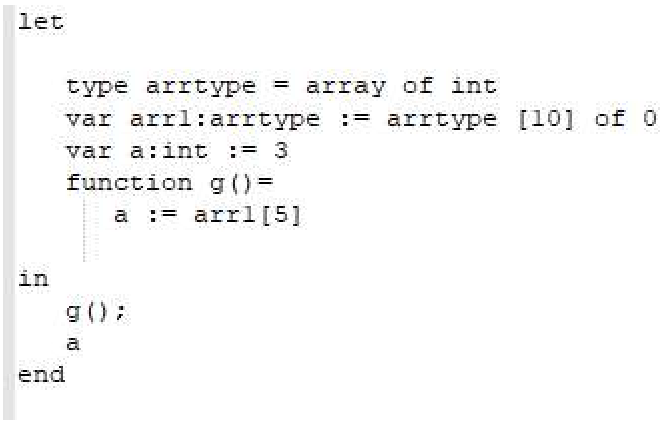
The extra difficulty in implementing x86 instructions is that there are 8 kinds of registers in different length. Currently, I simply place the registers in a list. And allocate the first free register in the list. That needs further optimization.

7. Test cases

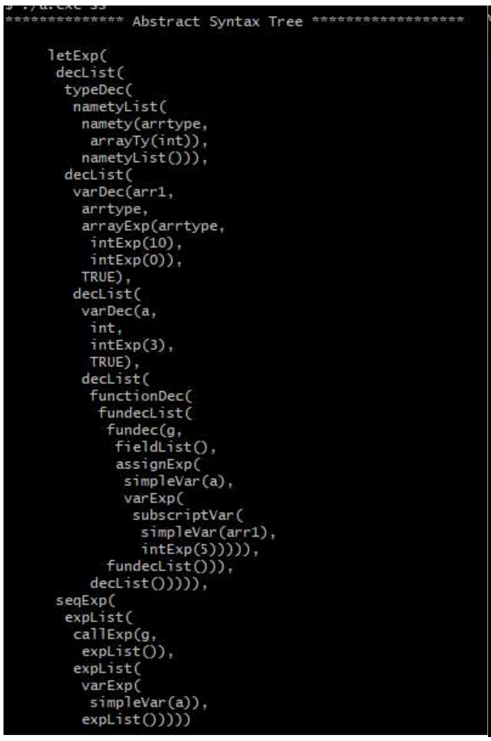
7.1 Array

The main job of this test case is the definition and usage of array.

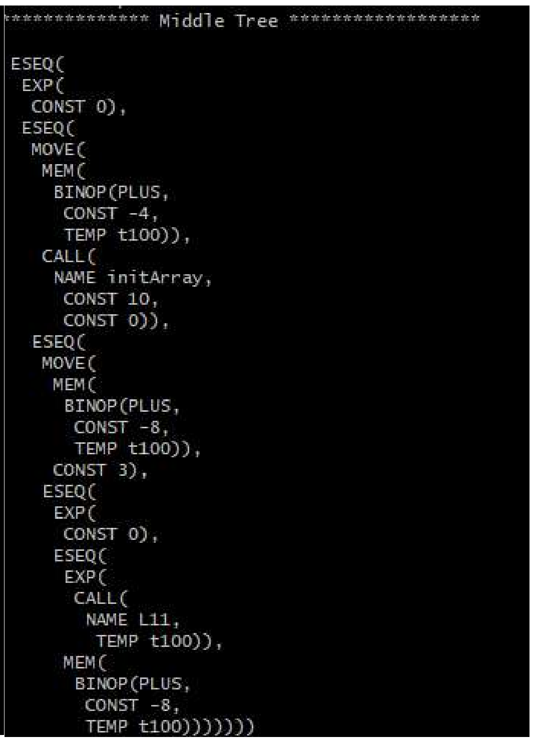
Test code:



```
let

   type arrtype = array of int
   var arr1:arrtype := arrtype [10] of 0
   var a:int := 3
   function g()=
      a := arr1[5]

in
   g();
   a
end
```

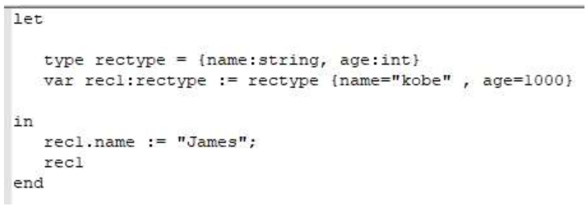


```
************** Abstract Syntax Tree ******************

     letExp(
      decList(
       typeDec(
        nametyList(
         namety(arrtype,
          arrayTy(int)),
         nametyList())),
       decList(
        varDec(arr1,
         arrtype,
         arrayExp(arrtype,
          intExp(10),
          intExp(0)),
         TRUE),
        decList(
         varDec(a,
          int,
          intExp(3),
          TRUE),
         decList(
          functionDec(
           fundecList(
            fundec(g,
             fieldList(),
             assignExp(
              simpleVar(a),
              varExp(
               subscriptVar(
                simpleVar(arr1),
                intExp(5))))),
            fundecList())),
          decList())))),
      seqExp(
       expList(
        callExp(g,
         expList(),
        expList(
         varExp(
          simpleVar(a)),
         expList()))))
```

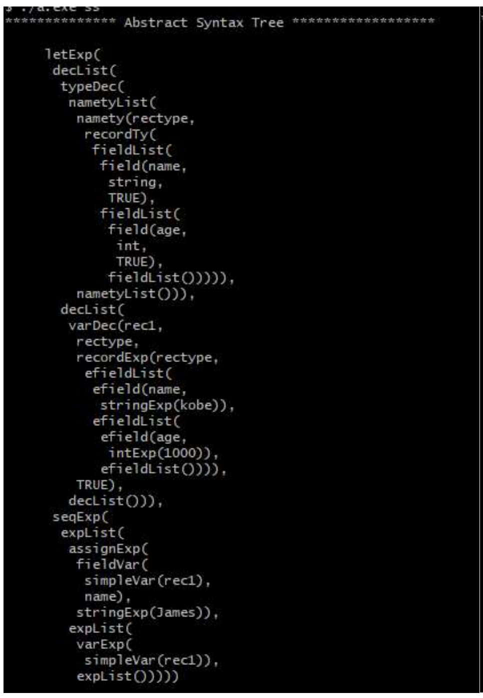


```
*************** Middle Tree ********************

ESEQ(
 EXP(
  CONST 0),
 ESEQ(
  MOVE(
   MEM(
    BINOP(PLUS,
     CONST -4,
     TEMP t100)),
   CALL(
    NAME initArray,
     CONST 10,
     CONST 0)),
  ESEQ(
   MOVE(
    MEM(
     BINOP(PLUS,
      CONST -8,
      TEMP t100)),
    CONST 3),
   ESEQ(
    EXP(
     CONST 0),
    ESEQ(
     EXP(
      CALL(
       NAME L11,
        TEMP t100)),
     MEM(
      BINOP(PLUS,
       CONST -8,
       TEMP t100)))))))
```

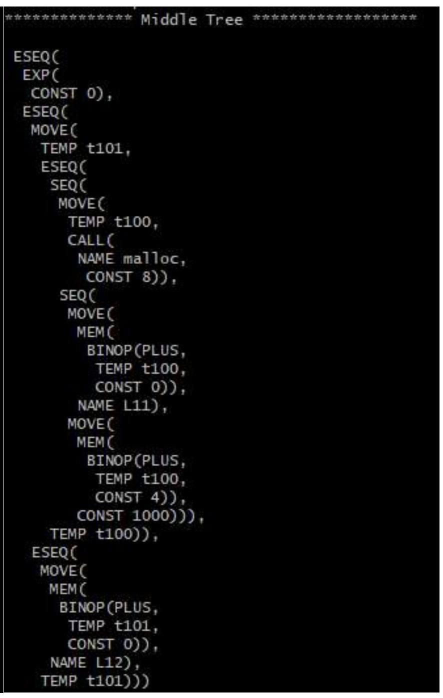
7.2 record



```
let

   type rectype = {name:string, age:int}
   var rec1:rectype := rectype {name="kobe" , age=1000}

in
   rec1.name := "James";
   rec1
end
```



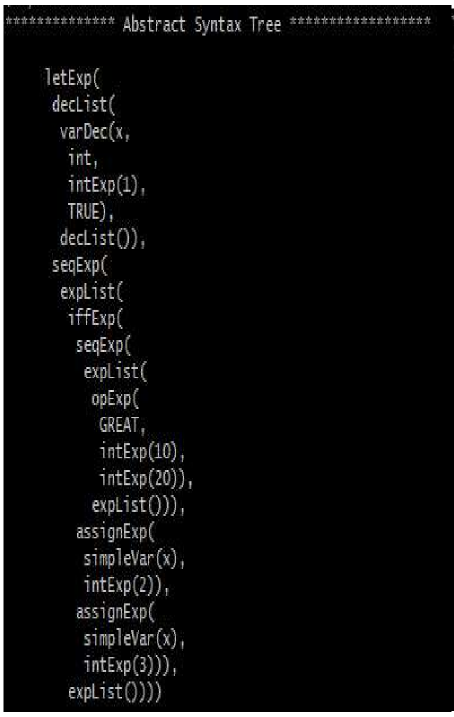
```
************** Abstract Syntax Tree ******************

     letExp(
      decList(
       typeDec(
        nametyList(
         namety(rectype,
          recordTy(
           fieldList(
            field(name,
             string,
             TRUE),
            fieldList(
             field(age,
              int,
              TRUE),
             fieldList())))),
         nametyList())),
       decList(
        varDec(rec1,
         rectype,
         recordExp(rectype,
          efieldList(
           efield(name,
            stringExp(kobe)),
           efieldList(
            efield(age,
             intExp(1000)),
            efieldList()))),
         TRUE),
        decList())),
      seqExp(
       expList(
        assignExp(
         fieldVar(
          simpleVar(rec1),
          name),
         stringExp(James)),
        expList(
         varExp(
          simpleVar(rec1)),
         expList()))))
```



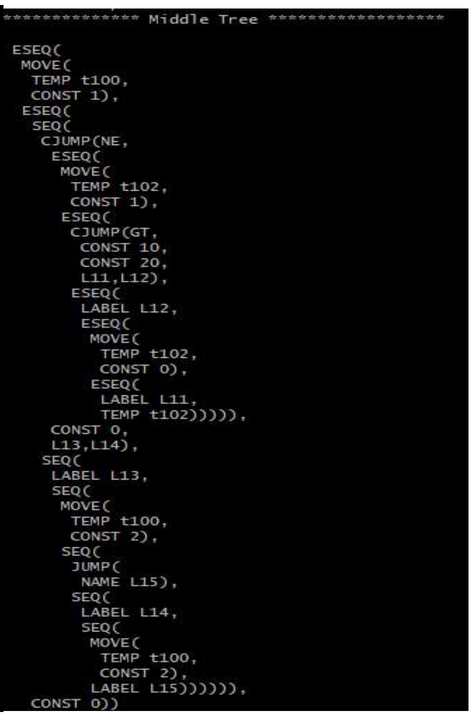
```
************** Middle Tree ******************

ESEQ(
 EXP(
  CONST 0),
 ESEQ(
  MOVE(
   TEMP t101,
   ESEQ(
    SEQ(
     MOVE(
      TEMP t100,
      CALL(
       NAME malloc,
        CONST 8)),
     SEQ(
      MOVE(
       MEM(
        BINOP(PLUS,
         TEMP t100,
         CONST 0)),
       NAME L11),
      MOVE(
       MEM(
        BINOP(PLUS,
         TEMP t100,
         CONST 4)),
       CONST 1000))),
    TEMP t100)),
  ESEQ(
   MOVE(
    MEM(
     BINOP(PLUS,
      TEMP t101,
      CONST 0)),
    NAME L12),
   TEMP t101)))
```

7.3 if statement

```
let

   var x:int := 1

in
   if (10 > 20) then x := 2 else x := 3
end
```

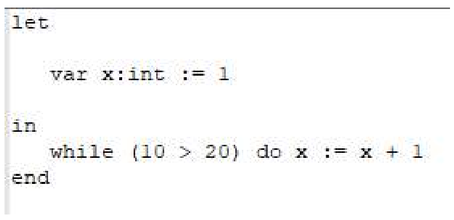


```
************** Abstract Syntax Tree ******************

 letExp(
  decList(
   varDec(x,
    int,
    intExp(1),
    TRUE),
   decList()),
  seqExp(
   expList(
    iffExp(
     seqExp(
      expList(
       opExp(
        GREAT,
        intExp(10),
        intExp(20)),
       expList())),
     assignExp(
      simpleVar(x),
      intExp(2)),
     assignExp(
      simpleVar(x),
      intExp(3))),
    expList()))
```

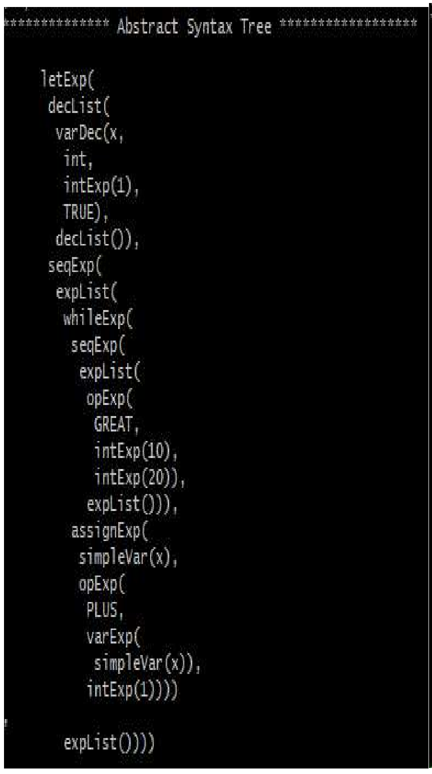


```
************** Middle Tree *******************

 ESEQ(
  MOVE(
   TEMP t100,
   CONST 1),
  ESEQ(
   SEQ(
    CJUMP(NE,
     ESEQ(
      MOVE(
       TEMP t102,
       CONST 1),
      ESEQ(
       CJUMP(GT,
        CONST 10,
        CONST 20,
        L11,L12),
       ESEQ(
        LABEL L12,
        ESEQ(
         MOVE(
          TEMP t102,
          CONST 0),
         ESEQ(
          LABEL L11,
          TEMP t102))))),
     CONST 0,
     L13,L14),
    SEQ(
     LABEL L13,
     SEQ(
      MOVE(
       TEMP t100,
       CONST 2),
      SEQ(
       JUMP(
        NAME L15),
       SEQ(
        LABEL L14,
        SEQ(
         MOVE(
          TEMP t100,
          CONST 2),
         LABEL L15)))))),
   CONST 0))
```

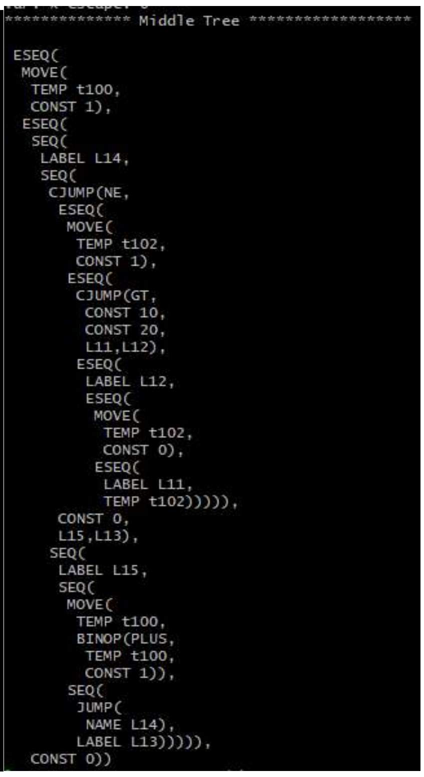
7.4 while loop



```
let

   var x:int := 1

in
   while (10 > 20) do x := x + 1
end
```

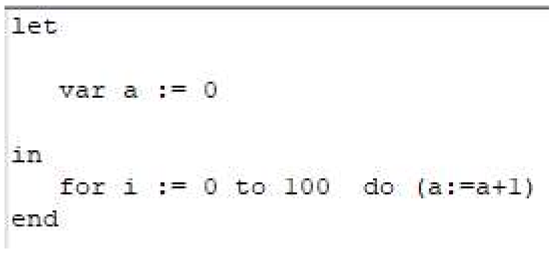


```
************* Abstract Syntax Tree *****************

    letExp(
     decList(
      varDec(x,
       int,
       intExp(1),
       TRUE),
      decList()),
     seqExp(
      expList(
       whileExp(
        seqExp(
         expList(
          opExp(
           GREAT,
           intExp(10),
           intExp(20)),
          expList())),
        assignExp(
         simpleVar(x),
         opExp(
          PLUS,
          varExp(
           simpleVar(x)),
          intExp(1))))

       expList())))
```

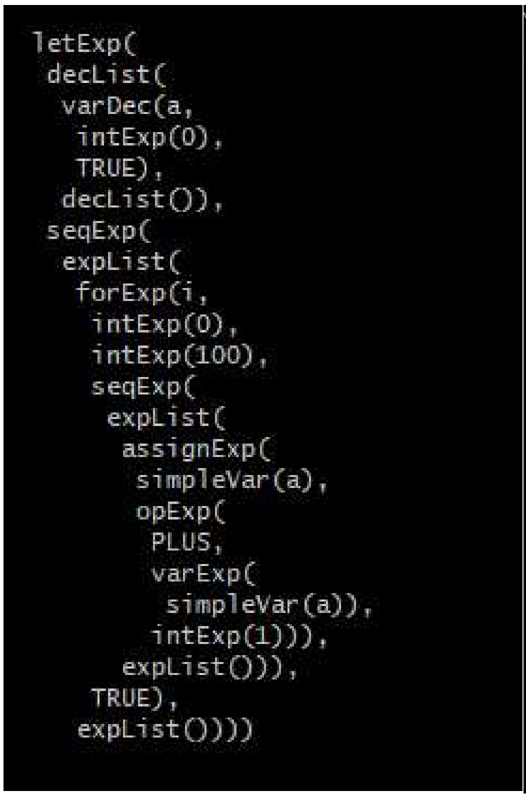


```
************* Middle Tree *****************

 ESEQ(
  MOVE(
   TEMP t100,
   CONST 1),
  ESEQ(
   SEQ(
    LABEL L14,
    SEQ(
     CJUMP(NE,
      ESEQ(
       MOVE(
        TEMP t102,
        CONST 1),
       ESEQ(
        CJUMP(GT,
         CONST 10,
         CONST 20,
         L11,L12),
        ESEQ(
         LABEL L12,
         ESEQ(
          MOVE(
           TEMP t102,
           CONST 0),
          ESEQ(
           LABEL L11,
           TEMP t102))))),
      CONST 0,
      L15,L13),
     SEQ(
      LABEL L15,
      SEQ(
       MOVE(
        TEMP t100,
        BINOP(PLUS,
         TEMP t100,
         CONST 1)),
       SEQ(
        JUMP(
         NAME L14),
        LABEL L13))))),
   CONST 0))
```

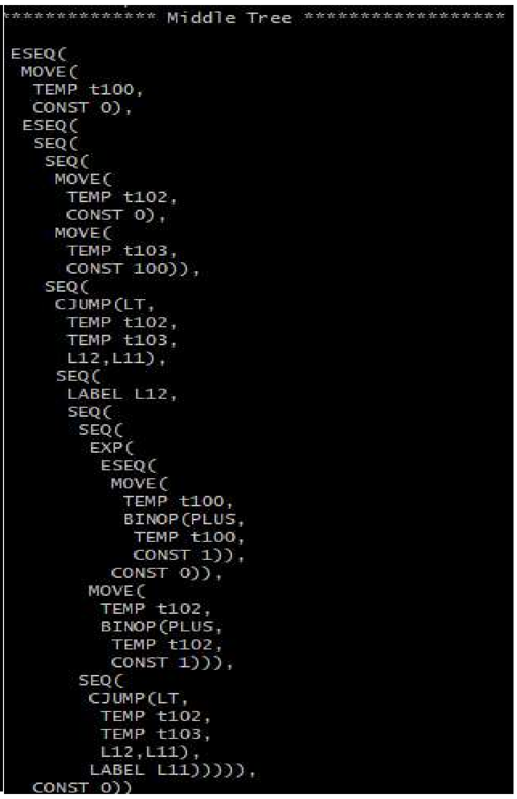
7.5 for statement



```
let

   var a := 0

in
   for i := 0 to 100  do (a:=a+1)
end
```

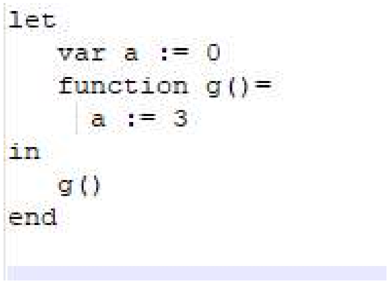


```
letExp(
 decList(
  varDec(a,
   intExp(0),
   TRUE),
  decList()),
 seqExp(
  expList(
   forExp(i,
    intExp(0),
    intExp(100),
    seqExp(
     expList(
      assignExp(
       simpleVar(a),
       opExp(
        PLUS,
        varExp(
         simpleVar(a)),
        intExp(1))),
      expList())),
    TRUE),
   expList())))
```

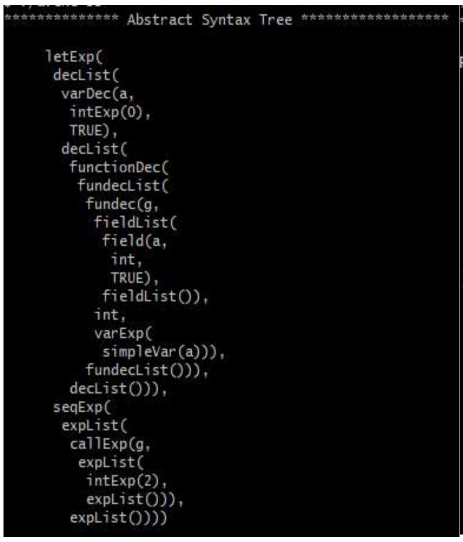


```
*************** Middle Tree *******************

ESEQ(
 MOVE(
  TEMP t100,
  CONST 0),
 ESEQ(
  SEQ(
   SEQ(
    MOVE(
     TEMP t102,
     CONST 0),
    MOVE(
     TEMP t103,
     CONST 100)),
   SEQ(
    CJUMP(LT,
     TEMP t102,
     TEMP t103,
     L12,L11),
    SEQ(
     LABEL L12,
     SEQ(
      SEQ(
       EXP(
        ESEQ(
         MOVE(
          TEMP t100,
          BINOP(PLUS,
           TEMP t100,
           CONST 1)),
         CONST 0)),
       MOVE(
        TEMP t102,
        BINOP(PLUS,
         TEMP t102,
         CONST 1))),
      SEQ(
       CJUMP(LT,
        TEMP t102,
        TEMP t103,
        L12,L11),
       LABEL L11))))),
  CONST 0))
```

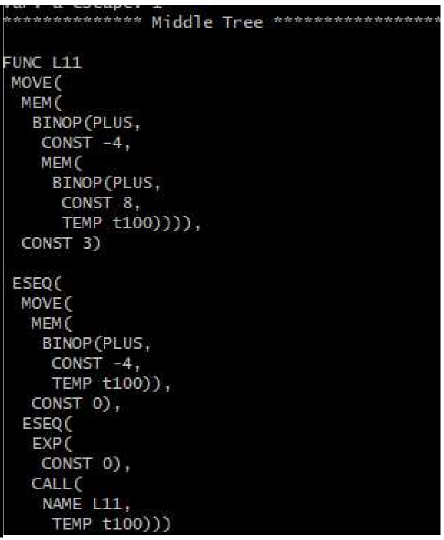
7.6 function



```
let
   var a := 0
   function g()=
     a := 3
in
   g()
end
```

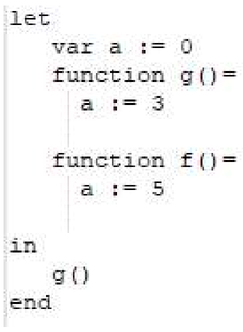


```
************* Abstract Syntax Tree *****************

    letExp(
     decList(
      varDec(a,
       intExp(0),
       TRUE),
      decList(
       functionDec(
        fundecList(
         fundec(g,
          fieldList(
           field(a,
            int,
            TRUE),
           fieldList()),
          int,
          varExp(
           simpleVar(a))),
         fundecList())),
       decList())),
     seqExp(
      expList(
       callExp(g,
        expList(
         intExp(2),
         expList())),
       expList())))
```

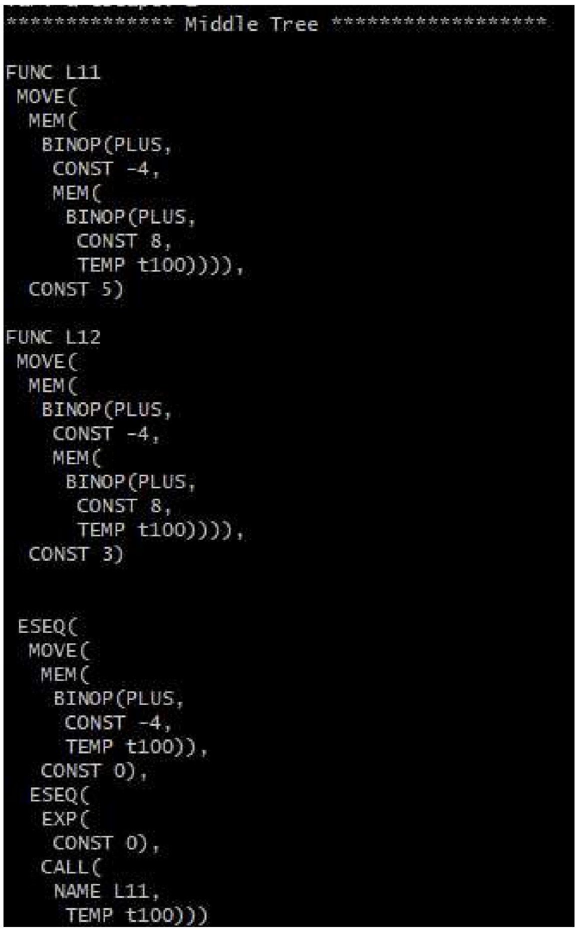


```
*************** Middle Tree *******************

FUNC L11
 MOVE(
  MEM(
   BINOP(PLUS,
    CONST -4,
    MEM(
     BINOP(PLUS,
      CONST 8,
      TEMP t100)))),
  CONST 3)

 ESEQ(
  MOVE(
   MEM(
    BINOP(PLUS,
     CONST -4,
     TEMP t100)),
   CONST 0),
  ESEQ(
   EXP(
    CONST 0),
   CALL(
    NAME L11,
     TEMP t100)))
```

In this module ,the content of function is stored in the frame. The frame will be transmitted to the next period to generate target code. In this period, the blocks before the last part are the body of functions. Information about return type or arguments is stored in frame.
To illustrate this problem, next case illustrates two declarations of function.



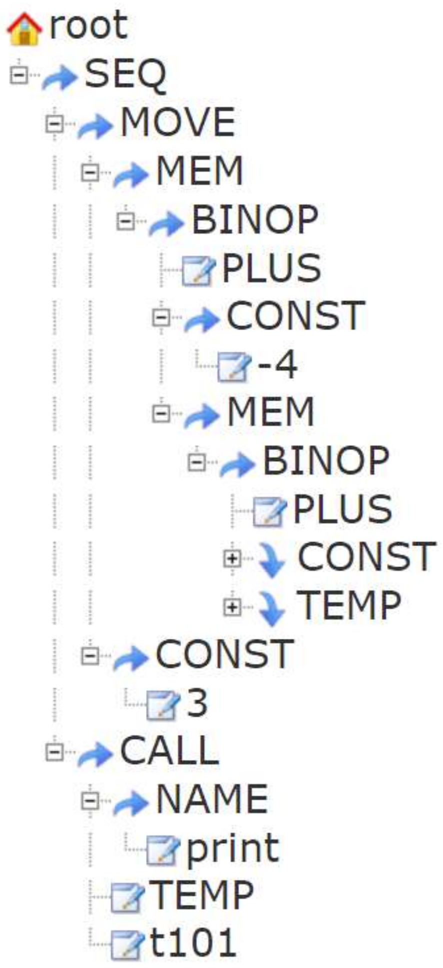
```
let
   var a := 0
   function g()=
     a := 3

   function f()=
     a := 5

in
   g()
end
```



```
************** Middle Tree ******************

FUNC L11
 MOVE(
  MEM(
   BINOP(PLUS,
    CONST -4,
    MEM(
     BINOP(PLUS,
      CONST 8,
      TEMP t100)))),
  CONST 5)

FUNC L12
 MOVE(
  MEM(
   BINOP(PLUS,
    CONST -4,
    MEM(
     BINOP(PLUS,
      CONST 8,
      TEMP t100)))),
  CONST 3)


 ESEQ(
  MOVE(
   MEM(
    BINOP(PLUS,
     CONST -4,
     TEMP t100)),
   CONST 0),
  ESEQ(
   EXP(
    CONST 0),
   CALL(
    NAME L11,
     TEMP t100)))
```

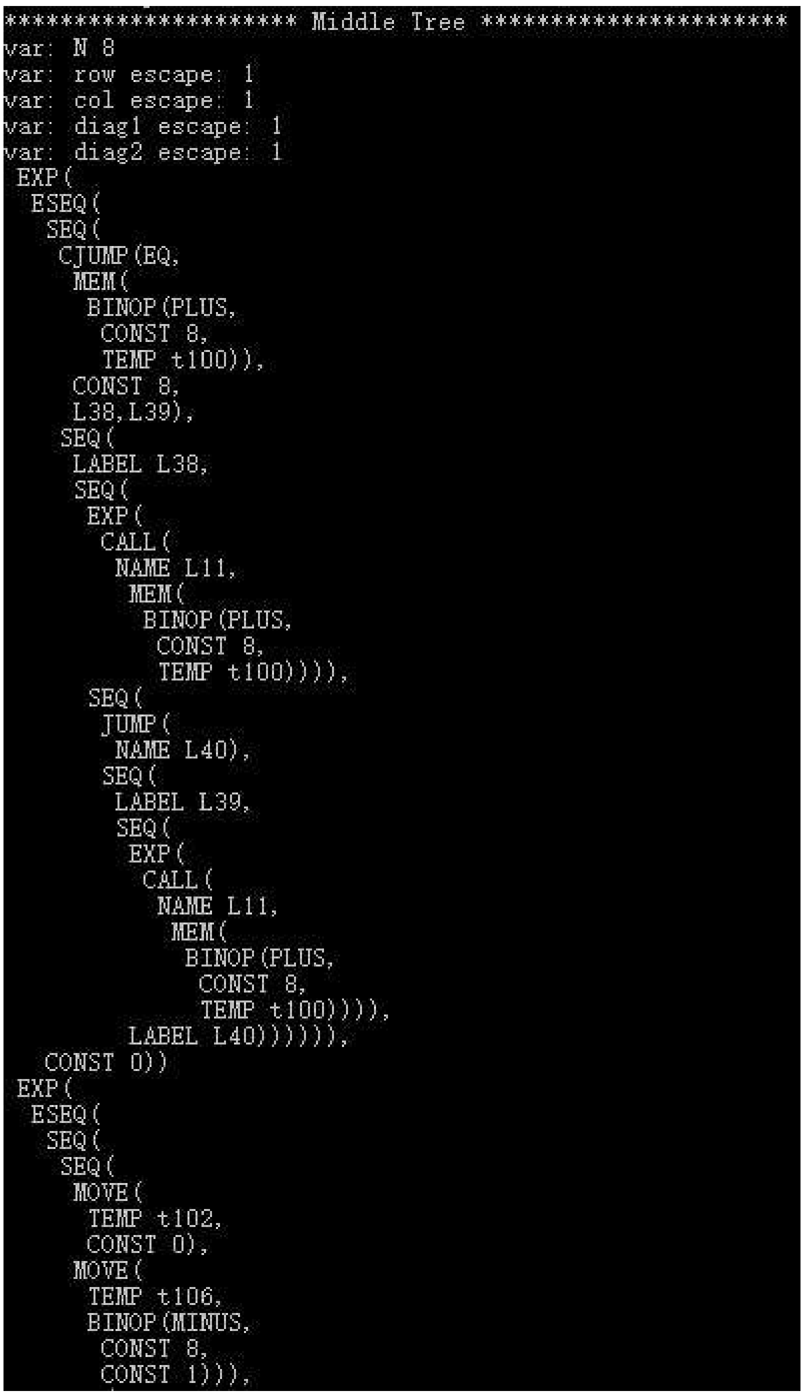
To visualize the middle tree, we develop a web page to demonstrate it with html and js. The page is like the following one. The node can be stretched out or hidden.

Middel Tree:

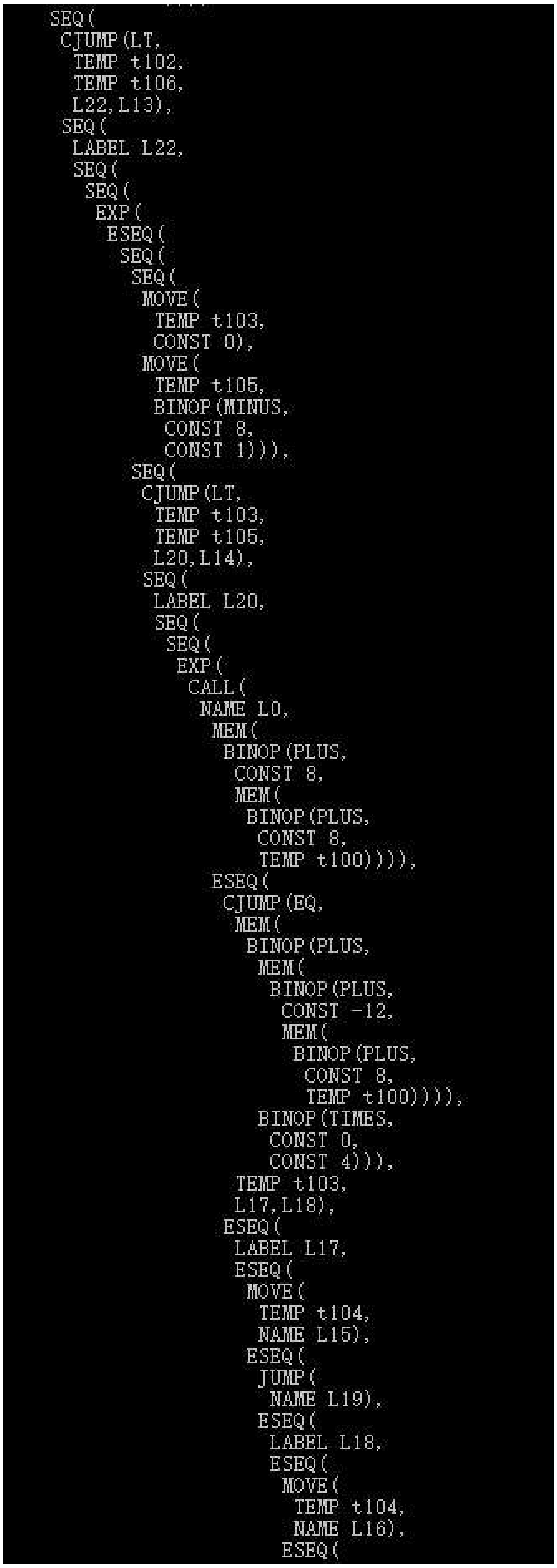


```
root
  SEQ
    MOVE
      MEM
        BINOP
          PLUS
          CONST
            -4
          MEM
            BINOP
              PLUS
              CONST
              TEMP
      CONST
        3
    CALL
      NAME
        print
      TEMP
      t101
```

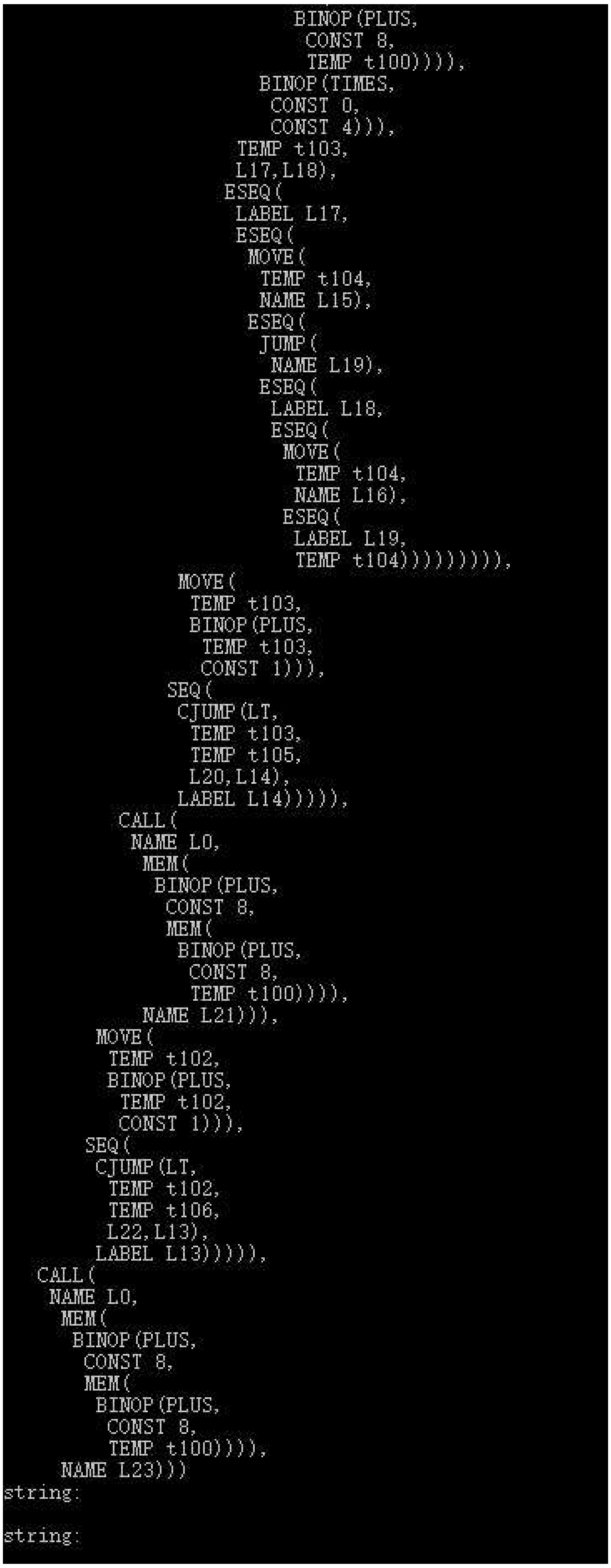
7.7 Object code test



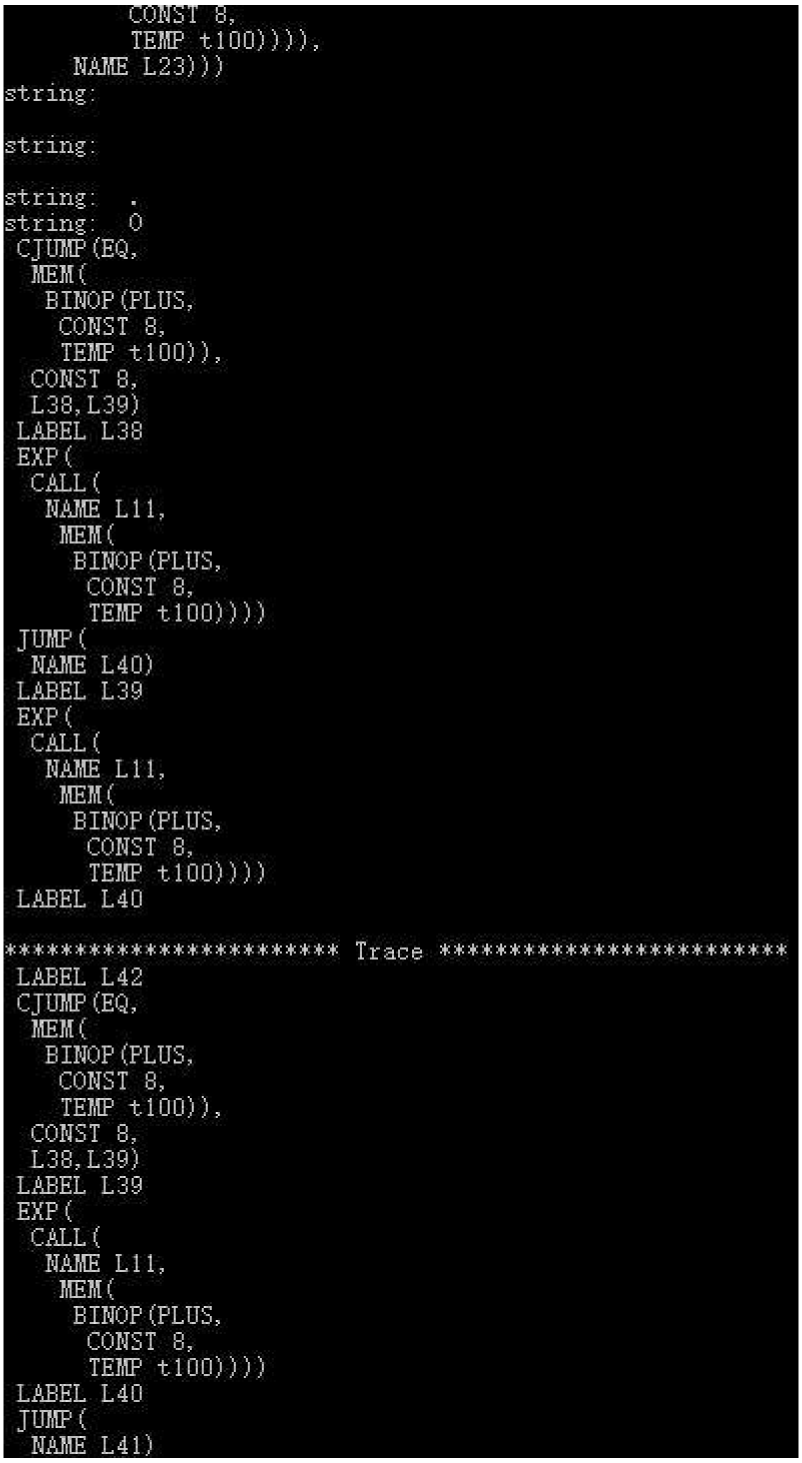
```
********************** Middle Tree ***********************
var: N 8
var: row escape: 1
var: col escape: 1
var: diag1 escape: 1
var: diag2 escape: 1
 EXP(
  ESEQ(
   SEQ(
    CJUMP(EQ,
     MEM(
      BINOP(PLUS,
       CONST 8,
       TEMP t100)),
     CONST 8,
     L38,L39),
    SEQ(
     LABEL L38,
     SEQ(
      EXP(
       CALL(
        NAME L11,
         MEM(
          BINOP(PLUS,
           CONST 8,
           TEMP t100)))),
      SEQ(
       JUMP(
        NAME L40),
       SEQ(
        LABEL L39,
        SEQ(
         EXP(
          CALL(
           NAME L11,
            MEM(
             BINOP(PLUS,
              CONST 8,
              TEMP t100)))),
         LABEL L40)))))),
   CONST 0))
 EXP(
  ESEQ(
   SEQ(
    SEQ(
     MOVE(
      TEMP t102,
      CONST 0),
     MOVE(
      TEMP t106,
      BINOP(MINUS,
       CONST 8,
       CONST 1))),
```



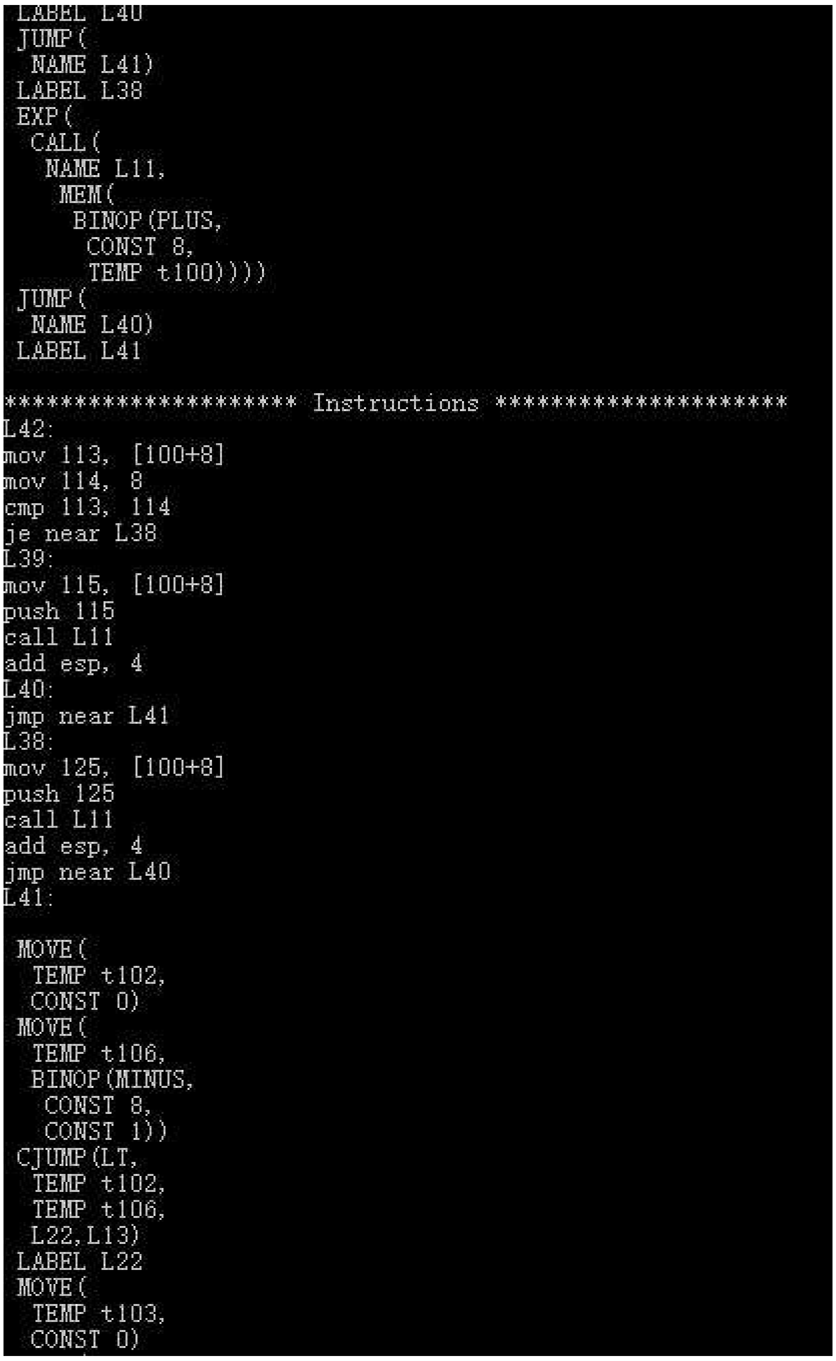
```
    SEQ(
     CJUMP(LT,
      TEMP t102,
      TEMP t106,
      L22,L13),
     SEQ(
      LABEL L22,
      SEQ(
       SEQ(
        EXP(
         ESEQ(
          SEQ(
           SEQ(
            MOVE(
             TEMP t103,
             CONST 0),
            MOVE(
             TEMP t105,
             BINOP(MINUS,
              CONST 8,
              CONST 1))),
           SEQ(
            CJUMP(LT,
             TEMP t103,
             TEMP t105,
             L20,L14),
            SEQ(
             LABEL L20,
             SEQ(
              SEQ(
               EXP(
                CALL(
                 NAME L0,
                  MEM(
                   BINOP(PLUS,
                    CONST 8,
                    MEM(
                     BINOP(PLUS,
                      CONST 8,
                      TEMP t100)))),
                  ESEQ(
                   CJUMP(EQ,
                    MEM(
                     BINOP(PLUS,
                      MEM(
                       BINOP(PLUS,
                        CONST -12,
                        MEM(
                         BINOP(PLUS,
                          CONST 8,
                          TEMP t100)))),
                      BINOP(TIMES,
                       CONST 0,
                       CONST 4))),
                    TEMP t103,
                    L17,L18),
                   ESEQ(
                    LABEL L17,
                    ESEQ(
                     MOVE(
                      TEMP t104,
                      NAME L15),
                     ESEQ(
                      JUMP(
                       NAME L19),
                      ESEQ(
                       LABEL L18,
                       ESEQ(
                        MOVE(
                         TEMP t104,
                         NAME L16),
                        ESEQ(
```



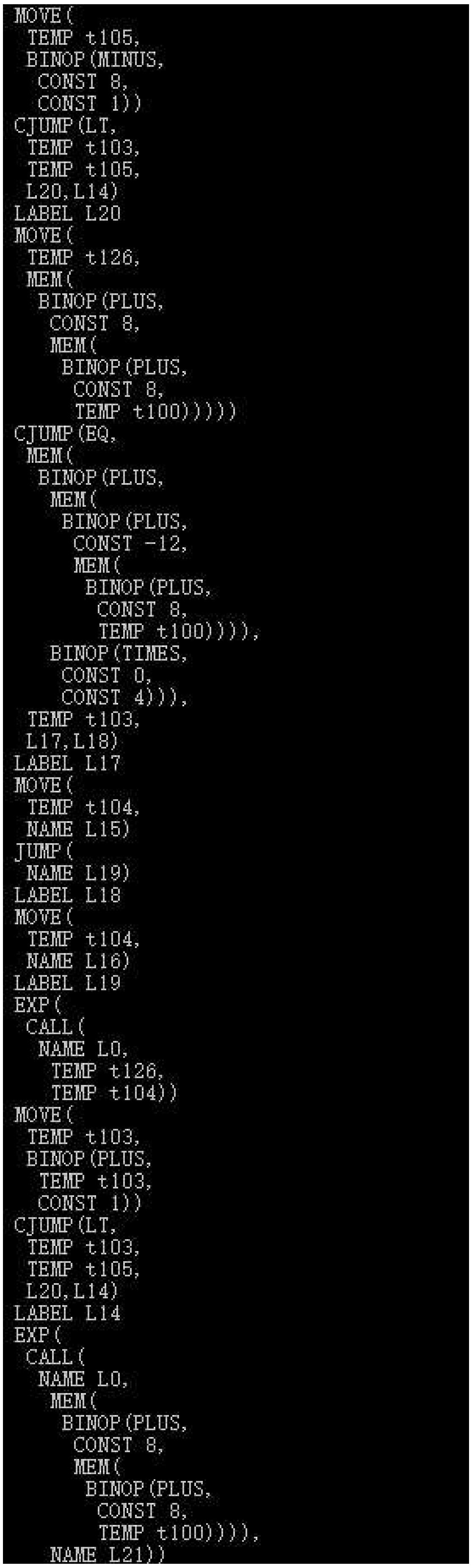
```
                         BINOP(PLUS,
                          CONST 8,
                          TEMP t100)))),
                      BINOP(TIMES,
                       CONST 0,
                       CONST 4))),
                    TEMP t103,
                    L17,L18),
                   ESEQ(
                    LABEL L17,
                    ESEQ(
                     MOVE(
                      TEMP t104,
                      NAME L15),
                     ESEQ(
                      JUMP(
                       NAME L19),
                      ESEQ(
                       LABEL L18,
                       ESEQ(
                        MOVE(
                         TEMP t104,
                         NAME L16),
                        ESEQ(
                         LABEL L19,
                         TEMP t104))))))))),
               MOVE(
                TEMP t103,
                BINOP(PLUS,
                 TEMP t103,
                 CONST 1))),
              SEQ(
               CJUMP(LT,
                TEMP t103,
                TEMP t105,
                L20,L14),
               LABEL L14))))),
          CALL(
           NAME L0,
            MEM(
             BINOP(PLUS,
              CONST 8,
              MEM(
               BINOP(PLUS,
                CONST 8,
                TEMP t100)))),
            NAME L21))),
        MOVE(
         TEMP t102,
         BINOP(PLUS,
          TEMP t102,
          CONST 1))),
       SEQ(
        CJUMP(LT,
         TEMP t102,
         TEMP t106,
         L22,L13),
        LABEL L13))))),
   CALL(
    NAME L0,
     MEM(
      BINOP(PLUS,
       CONST 8,
       MEM(
        BINOP(PLUS,
         CONST 8,
         TEMP t100)))),
     NAME L23)))
string:

string:
```



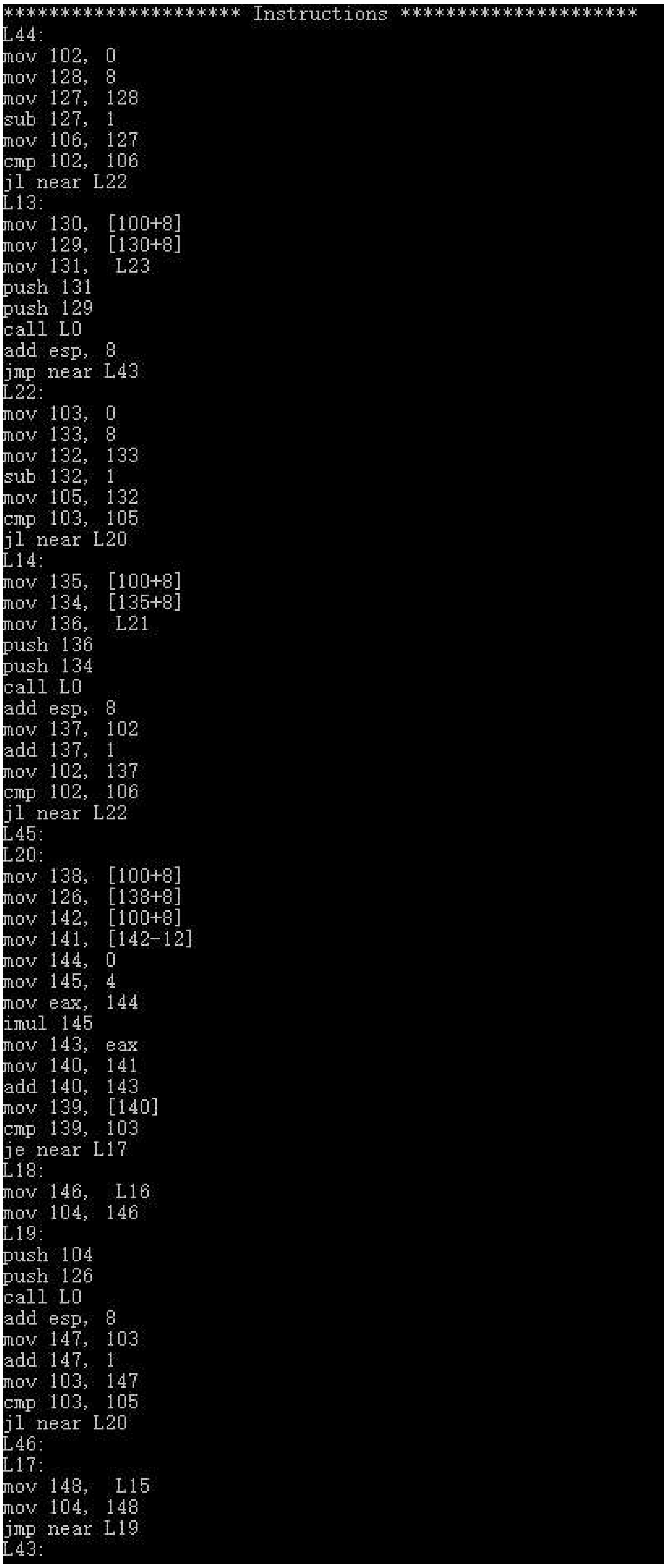
```
         CONST 8,
         TEMP t100)))),
     NAME L23)))
string:

string:

string:  .
string:  0
 CJUMP(EQ,
  MEM(
   BINOP(PLUS,
    CONST 8,
    TEMP t100)),
  CONST 8,
  L38,L39)
 LABEL L38
 EXP(
  CALL(
   NAME L11,
    MEM(
     BINOP(PLUS,
      CONST 8,
      TEMP t100))))
 JUMP(
  NAME L40)
 LABEL L39
 EXP(
  CALL(
   NAME L11,
    MEM(
     BINOP(PLUS,
      CONST 8,
      TEMP t100))))
 LABEL L40

************************* Trace **************************
 LABEL L42
 CJUMP(EQ,
  MEM(
   BINOP(PLUS,
    CONST 8,
    TEMP t100)),
  CONST 8,
  L38,L39)
 LABEL L39
 EXP(
  CALL(
   NAME L11,
    MEM(
     BINOP(PLUS,
      CONST 8,
      TEMP t100))))
 LABEL L40
 JUMP(
  NAME L41)
```



```
 LABEL L40
 JUMP(
  NAME L41)
 LABEL L38
 EXP(
  CALL(
   NAME L11,
    MEM(
     BINOP(PLUS,
      CONST 8,
      TEMP t100))))
 JUMP(
  NAME L40)
 LABEL L41

********************** Instructions **********************
L42:
mov 113, [100+8]
mov 114, 8
cmp 113, 114
je near L38
L39:
mov 115, [100+8]
push 115
call L11
add esp, 4
L40:
jmp near L41
L38:
mov 125, [100+8]
push 125
call L11
add esp, 4
jmp near L40
L41:

 MOVE(
  TEMP t102,
  CONST 0)
 MOVE(
  TEMP t106,
  BINOP(MINUS,
   CONST 8,
   CONST 1))
 CJUMP(LT,
  TEMP t102,
  TEMP t106,
  L22,L13)
 LABEL L22
 MOVE(
  TEMP t103,
  CONST 0)
```



```
MOVE(
 TEMP t105,
 BINOP(MINUS,
  CONST 8,
  CONST 1))
CJUMP(LT,
 TEMP t103,
 TEMP t105,
 L20,L14)
LABEL L20
MOVE(
 TEMP t126,
 MEM(
  BINOP(PLUS,
   CONST 8,
   MEM(
    BINOP(PLUS,
     CONST 8,
     TEMP t100)))))
CJUMP(EQ,
 MEM(
  BINOP(PLUS,
   MEM(
    BINOP(PLUS,
     CONST -12,
     MEM(
      BINOP(PLUS,
       CONST 8,
       TEMP t100)))),
   BINOP(TIMES,
    CONST 0,
    CONST 4))),
 TEMP t103,
 L17,L18)
LABEL L17
MOVE(
 TEMP t104,
 NAME L15)
JUMP(
 NAME L19)
LABEL L18
MOVE(
 TEMP t104,
 NAME L16)
LABEL L19
EXP(
 CALL(
  NAME L0,
   TEMP t126,
   TEMP t104))
MOVE(
 TEMP t103,
 BINOP(PLUS,
  TEMP t103,
  CONST 1))
CJUMP(LT,
 TEMP t103,
 TEMP t105,
 L20,L14)
LABEL L14
EXP(
 CALL(
  NAME L0,
   MEM(
    BINOP(PLUS,
     CONST 8,
     MEM(
      BINOP(PLUS,
       CONST 8,
       TEMP t100)))),
   NAME L21))
```



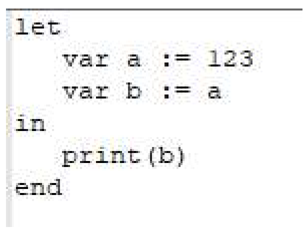
```
********************** Instructions **********************
L44:
mov 102, 0
mov 128, 8
mov 127, 128
sub 127, 1
mov 106, 127
cmp 102, 106
jl near L22
L13:
mov 130, [100+8]
mov 129, [130+8]
mov 131,  L23
push 131
push 129
call L0
add esp, 8
jmp near L43
L22:
mov 103, 0
mov 133, 8
mov 132, 133
sub 132, 1
mov 105, 132
cmp 103, 105
jl near L20
L14:
mov 135, [100+8]
mov 134, [135+8]
mov 136,  L21
push 136
push 134
call L0
add esp, 8
mov 137, 102
add 137, 1
mov 102, 137
cmp 102, 106
jl near L22
L45:
L20:
mov 138, [100+8]
mov 126, [138+8]
mov 142, [100+8]
mov 141, [142-12]
mov 144, 0
mov 145, 4
mov eax, 144
imul 145
mov 143, eax
mov 140, 141
add 140, 143
mov 139, [140]
cmp 139, 103
je near L17
L18:
mov 146,  L16
mov 104, 146
L19:
push 104
push 126
call L0
add esp, 8
mov 147, 103
add 147, 1
mov 103, 147
cmp 103, 105
jl near L20
L46:
L17:
mov 148,  L15
mov 104, 148
jmp near L19
L43:
```

8. Optimization

8.1 Description

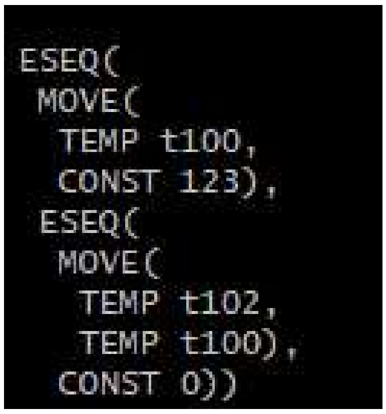
Constant propagation: When the value of a variable is fixed in program, then the value of this variable used in following places can be replaced directly by the value, instead of the variable itself.

For example, the source program is like the following:



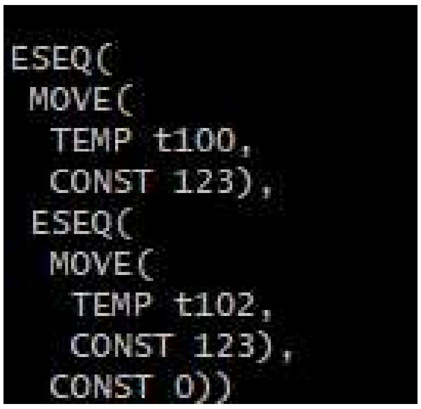
```
let
   var a := 123
   var b := a
in
   print(b)
end
```

Before the optimization, the middle tree will reference variable a when initialize variable b.



```
ESEQ(
 MOVE(
  TEMP t100,
  CONST 123),
 ESEQ(
  MOVE(
   TEMP t102,
   TEMP t100),
  CONST 0))
```

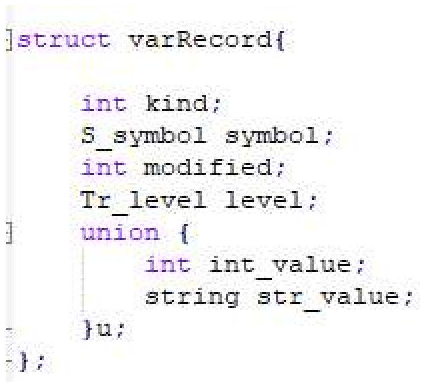
After the optimization, the middle tree will directly use the value 123 to initialize variable b. So the number of referencing variable is decreased.



```
ESEQ(
 MOVE(
  TEMP t100,
  CONST 123),
 ESEQ(
  MOVE(
   TEMP t102,
   CONST 123),
  CONST 0))
```

8.2 Implementation method

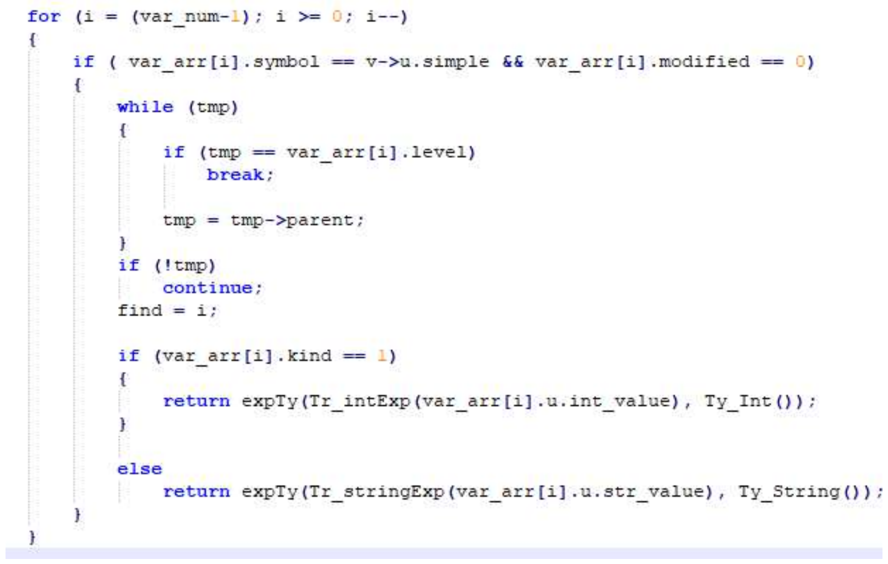
To finish this job, a new structure is defined as follows.



```
struct varRecord{

    int kind;
    S_symbol symbol;
    int modified;
    Tr_level level;
    union {
        int int_value;
        string str_value;
    }u;
};
```

In our program, only basic type variable will be propagation, that is “int” or “string”. Kind is used to indicate the type of variable, symbol represents the declared symbol, modified indicates if the variable has been modified, level indicates the frame level of this variable. union u stores the actual value. If the variable is used in afterwards code, and the variable hasn’t been modified yet, its value will be directly used.

When a simple variable is met when transforming the abstract code, the following code will be executed to determined if constant can be propagated. There is one thing to pay attention to: the loop variable i should be from the maximum to 0, because the inner layer of declaration will be used firstly.



```
for (i = (var_num-1); i >= 0; i--)
{
    if ( var_arr[i].symbol == v->u.simple && var_arr[i].modified == 0)
    {
        while (tmp)
        {
            if (tmp == var_arr[i].level)
                break;

            tmp = tmp->parent;
        }
        if (!tmp)
            continue;
        find = i;

        if (var_arr[i].kind == 1)
        {
            return expTy(Tr_intExp(var_arr[i].u.int_value), Ty_Int());
        }

        else
            return expTy(Tr_stringExp(var_arr[i].u.str_value), Ty_String());
    }
}
```