

实验四 语义分析和中间代码生成

1. 符号表的构造

1.1 目的

符号表的构造是为了在语义分析时判断一个变量或函数是否存在未定义就使用的错误情形，同时记录变量和函数的属性。由于常量的字符串和属性已经被存储到了语法分析树中，故符号表不存储常量信息。

1.2 符号表的结构

在这个试验中，考虑 TINY 语言只存在局部变量，同时程序的结构被定义为：

```
Program -> MethodDecl MethodDecl*
```

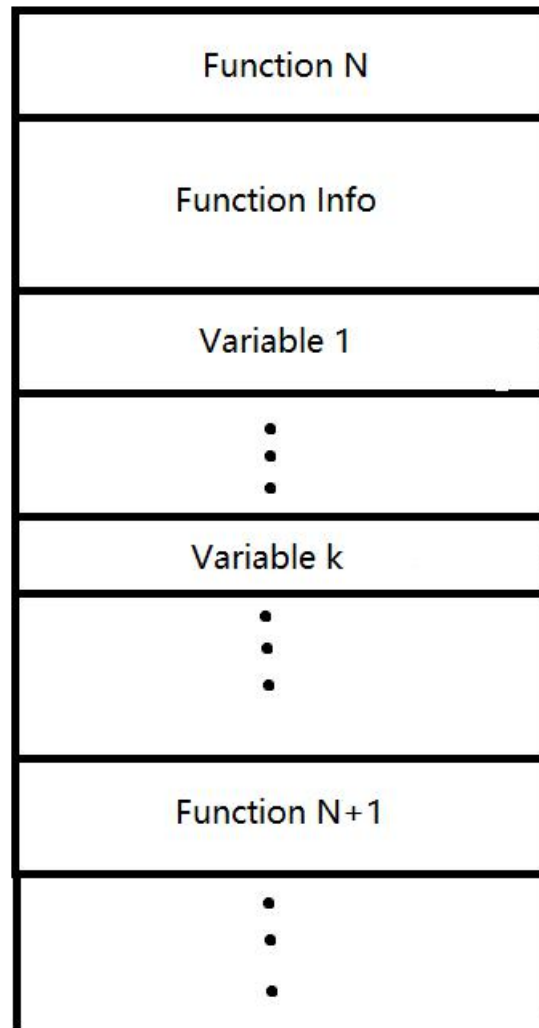
```
MethodDecl -> Type [MAIN] Id '(' FormalParams ')' Block
```

即程序由一个个函数组成且函数之间没有其他文法单元，又考虑到实现的复杂性，故我将一个变量的作用域定义为在其所在的函数内从该变量被定义的行开始到该函数结束为止。

综合上面的思考，我将符号表的结构设计成一个两层结构，即第一层为函数表，其包括所定义函数的属性信息，每个函数表的元组（fTuple）包括一个本地变量表（varTable），包含了在该函数内所定义的变量的属性信息。定义的结构体如下：

```
typedef struct {  
    char* varName;  
    int dataType;  
    unsigned int offset;  
}varTuple;  
  
typedef struct {  
    char* fName;  
    int returnType;  
    int argNum;  
    int argsType[MAX_ARGC];  
    int varNum;  
    varTuple varTable[MAX_LOCVAR];  
}fTuple;
```

该符号表用静态数组实现，可根据实际需求修改宏定义来满足编译所需要的资源数量。结构示意图如下：



1.3 符号表的构造过程

1.3.1 函数表的空间分配和初始化

函数表是一个全局变量，其在 `table.c` 中定义：

```
#define MAX_FUNCNUM 256
fTuple funcTable[MAX_FUNCNUM];
int funcTop = -1;
```

其中的 `funcTop` 指向了当前正在进行语义翻译和代码生成时所属的函数在函数表中的位置。

在产生式：

`MethodDecl : Type MAIN ID LEFTBRACKETS FormalParams RIGHTBRACKETS Block`

的文法单元 `ID` 后，嵌入语义规则：

```
{addFunc($3->nodeContent->tokenStr, getType($1));}
```

`addFunc` 函数会将 `ID` 对应的标识符名称作为函数名参数和 `Type` 所对应的类型作为返回值类型参数。

在 `addFunc` 中，会检查是否与已定义的函数重名：

```
for(int i = 0; i <= funcTop; i++) {
    if(strcmp(fName, funcTable[i].fName) == 0) {
        return 1;//multiple function name
    }
}
```

如果没有，则会将 `funcTop` 加一并更新函数表信息：

```
funcTop += 1;
funcTable[funcTop].fName = fName;
funcTable[funcTop].returnType = returnType;
funcTable[funcTop].argNum = 0;//initialize to 0, might increases later
```

funcTable[funcTop].varNum = 0;//initialize to 0, might increases later

1.3.2 函数形参的记录

在产生式：FormalParam : Type ID 后嵌入语义规则，其包含：

```
addArg($2->nodeContent->tokenStr, getType($1), getOffset(getType($1)));
```

形参的类型会被记录在 argsType 数组中，而形参本身也会作为函数的一个本地变量被记录在变量表中。

每当一个形参被规约时，该形参所包含的标识符和类型信息就会被 addArg() 录入符号表中。特别的由于产生式：

FormalParams : FormalParam | FormalParams COMMA FormalParam

是一个左递归即产生最左推导，最左边的形参在形参类型表中被存储在低位。

1.3.3 本地变量的记录

在产生式：LocalVarDecl : Type ID SEMICOLON 后加上语义规则，其中包括：

```
addVar($2->nodeContent->tokenStr, getType($1), getOffset(getType($1)));
```

该函数会在当前函数表元组所包含的变量表中用实参中的信息，分配并更新一个变量元组，以记录该变量的信息。

1.3.4 符号表的测试

符号表的测试需要将语义分析的基本功能完成后才能进行，故放在后面的部分。

2. 语义分析

2.1 表达式类型判断

在这个语言中，有三种数据类型：整型、实数和布尔。

类型检查主要是对表达式的类型进行赋值和检查，与表达式(Expression)相关的产生式有：

Expression : Expression PLUS MultiplicativeExpr | Expression MINUS MultiplicativeExpr | MultiplicativeExpr

MultiplicativeExpr : MultiplicativeExpr MULTI PrimaryExpr | MultiplicativeExpr DIV PrimaryExpr | PrimaryExpr

PrimaryExpr : INTNUMBER | REALNUMBER | ID | LEFTBRACKETS Expression RIGHTBRACKETS | ID LEFTBRACKETS ActualParams RIGHTBRACKETS

BoolExpression : Expression EQUAL Expression | Expression NOTEQUAL Expression

对于 BoolExpression，其类型恒为布尔型。

对于 PrimaryExpr，其规约后的文法单元的类型属性直接取产生式右边对应词法单元的类型，如

PrimaryExpr : ID {

.....

```
setType($$, getVarType($1->nodeContent->tokenStr));
```

.....

}

进行规约时直接调用 getVarType() 从符号表中取出对应变量的类型。

对于 MultiplicativeExpr 和 Expression，由于其包含两个表达式的计算，故需要综合考虑两个表达式的类型，我对转换规则定义如下

- 1.若两个表达式的类型相同，则运算结果表达式类型为该两个表达式的类型
- 2.若两个表达式至少有一个为实数，则运算结果表达式类型为实数
- 3.其余情况均为整型数类型

执行以上规则的函数是：

```

int operType(synTreenode* r1, synTreenode* r2) {
    if(r1->nodeContent->a.exprA.dataType==real || r2->nodeContent->a.exprA.dataType == real)return real;
    return integer;
}

```

当进行规约时，语义规则中的程序会将两个语法树节点作为参数调用该函数并将返回值作为该节点的类型。

以上便完成了表达式类型判断的功能。在赋值语句和函数返回语句中也需要类型检查，这些工作将结合符号表进行并最终体现在中间代码生成中。

2.2 符号表的应用

2.2.1 将符号表用于错误检测

符号表在错误检测中主要用于检查是否存在未声明或使用变量或函数和实参与形参不匹配的。当分析过程中遇到对变量的引用或函数的调用语句，语义规则会调用相应的检查函数，这些函数会在符号表中进行搜索以尝试匹配对应的标识符或检查实参与形参的对应，如果匹配失败则判定为错误。

2.2.2 符号表和类型检查的测试结果

用例：

```

1  /** this is a comment line in the sample program */
2  INT f2(INT x, INT y )
3  BEGIN
4  REAL z;
5  INT f1Var;
6  z := x*x - y*y;
7  z := a + b;
8  z := ((60.2 - 50) * 2) / 4 * 10 - 6 + (5 - 6 / 3);
9  RETURN z;
10 END
11 INT MAIN f1()
12 BEGIN
13 INT x;
14 READ(x, "A41.input");
15 INT y;
16 READ(y, "A42.input");
17 REAL z;
18 IF (z == 60) BEGIN
19 x := 1;
20 IF (y == 50) BEGIN
21 x := 3;
22 END ELSE BEGIN
23 x := 4;
24 END
25 END ELSE BEGIN
26 x := 2;
27 END
28 z := f2(x,y,f1Var) + f3(y,x) + f2(z, y);
29 WRITE (z, "A4.output");
30 END

```

该用例会测试符号表的构建和错误检测：

1. 引用未声明的变量
2. 返回值类型与函数定义的不同
3. 实参数量不匹配和在作用域外引用变量
4. 调用未声明的函数
5. 实参类型和形参不匹配

运行结果：

```

samchen2@ubuntu:~/tiny/code3$ ./run.sh
error at line 7:Undefined reference to a
error at line 7:Undefined reference to b
error at line 9:Function f2's return value type is integer but you return value with type real
error at line 28:Undefined reference to f1Var
error at line 28:Function f2 expects 2 parameters but pass 3 parameters
error at line 28:Undefined reference to f3
error at line 28:Function f2's 1th parameter expects type integer but pass type real

-----Function Table-----

Function1 : f2
  Argument type : integer integer
  Variable table:
    x integer 0
    y integer 4
    z real 8
    f1Var integer 16
Function2 : f1
  Argument type :
  Variable table:
    x integer 0
    y integer 4
    z real 8

```

语义分析程序运行后找出了 7 个错误：

- 1.变量 a 未定义就使用
- 2.变量 b 未定义就使用
- 3.函数 f2 定义的返回值类型为整型，却返回实数
- 4.变量 f1Var 未定义就使用（超出该变量的作用域）
- 5.f2 函数只有 2 个参数却传进 3 个
- 6.f3 函数未定义就使用
- 7.f2 函数第一个参数类型应为整型却传进实数类型

与该用例所设置的错误点相符。

构造出的符号表也与测试用例所期望相符。

3.中间代码生成

3.1 中间代码生成相关函数

void genTriCode(const char* res, const char* oper1, const char* operator, const char* oper2);//生成一个三地址代码

void genIfCode(const char* boolExpr, const char* gotoElseLabel);//生成一个 IF 条件跳转代码

void genRetCode(const char* expr);//生成一个返回返回值代码

void genGotoCode(const char* gotoLabel);//生成一个跳转代码

void genLabel(const char* labelStr);//生成一个跳转标签

void genCopyCode(const char* dst, const char* src);//生成一个赋值代码

void genParamPassCode(const char* param);//生成一个参数传递代码

void resetTempIndex();//重置临时变量，使自增编号置 0

void genCallCode(const char* procedureName, int paramNum);//生成一个函数调用代码

void genCallAssignCode(const char* dst, const char* procedureName, int paramNum);//生成一个调用并保存函数调用返回值的代码

void genCastCode(const char* dst, int castType, const char* src);//生成一个转换类型并保存转换值的代码

char* getTemp();//获得一个可用的临时变量

char* getLabel();//获得一个标签

char* afterCast(const char* initial, int initialType, int targetType);//将一个变量转换类型，存储到一个临时变量并返回该临时变量;如果 initialType = targetType，则返回 initial

char* getResIndex(synTreeNode*);//获得表达式值所存储的临时变量名称

3.2 赋值语句的代码生成

AssignStmt : ID ASSIGN Expression SEMICOLON {

```

.....
//generate code
setResIndex($3, afterCast(getResIndex($3), getType($3), getVarType($1->nodeContent->tokenStr)));
genCopyCode($1->nodeContent->tokenStr, getResIndex($3));
resetTempIndex();//Expression's value is used, reset temp var
}

```

在赋值语句的代码生成中，先对存储 Expression 值的临时变量进行类型检查和转换，再将检查和转换后的临时变量值通过调用 genCopyCode()生成赋值代码。

特别的，由于 Expression 被规约，其数值已经被赋给了 ID，所以计算该 Expression 所用到的临时变量可以被覆盖，所以将临时变量的自增编号重置为 0。

```

T0 = x * x
T1 = y * y
T2 = T0 - T1
T3 = (real)T2
z = T3

```

z := x*x - y*y 的生成代码

3.3 返回语句的代码生成

```

ReturnStmt : RETURN Expression SEMICOLON {
.....
//generate code
checkReturnType(getType($2));
setResIndex($2, afterCast(getResIndex($2), getType($2), getFuncRetType(getCurrentFuncName())));
genRetCode(getResIndex($2));
resetTempIndex();//Expression's value is used, reset temp var
}

```

整体逻辑与赋值语句的代码生成类似。

3.4 函数调用语句的代码生成

```

PrimaryExpr : | ID LEFTBRACKETS ActualParams RIGHTBRACKETS {
.....
//generate code
for(int i = 0; i <= actualParamsTop; i++) {
    genParamPassCode(actualParamsNameStack[i]);
}
setResIndex($$, getTemp());
genCallAssignCode(getResIndex($$), $1->nodeContent->tokenStr, actualParamsTop + 1);
//reset actual parameters stack
actualParamsTop = -1;
}

```

```

ActualParams : ActualParams COMMA Expression {
.....
//push actual parameter's type and name into stack
actualParamsTop++;
}

```

```

        actualParamsTypeStack[actualParamsTop] = getType($3);
        actualParamsNameStack[actualParamsTop] = getResIndex($3);
    }
    | Expression {
        .....
        //push actual parameter's type and name into stack
        actualParamsTop++;
        actualParamsTypeStack[actualParamsTop] = getType($1);
        actualParamsNameStack[actualParamsTop] = getResIndex($1);
    }

```

函数调用的代码生成包含两个部分，一个是实参的传递，一个是函数的调用。

对于实参传递的代码生成，在对实参文法单元进行规约时，语义规则会将存储实参表达式值的临时变量压入栈中，而后将栈中的变量作为参数调用 `genParamPassCode()` 生成参数传递代码。

对于函数调用的代码生成，直接调用 `genCallAssignCode()` 生成代码。

```

Param x
Param y
T0 = call f2, 2
-
f2(x, y)作为表达式时生成的代码

```

3.5 表达式计算的代码生成

3.5.1 算术表达式的代码生成

```

Expression : Expression PLUS MultiplicativeExpr {
    .....
    //generate code
    setResIndex($$, getTemp());
    setResIndex($1, afterCast(getResIndex($1), getType($1), getType($$)));
    setResIndex($3, afterCast(getResIndex($3), getType($3), getType($$)));
    genTriCode(getResIndex($$), getResIndex($1), "+", getResIndex($3));
}

```

表达式的计算中比较重要的一部分是表达式的类型转换，通过调用 `afterCast()` 转换参与计算的两个子表达式的类型并返回存储转换值的临时变量名称。最后调用 `genTriCode` 生成三地址码。

```

T1 = (real)y
T0 = z * T1
z = T0

```

`z := z * y` 产生的中间代码，`z` 为实数，`y` 为整型

3.5.2 布尔表达式的代码生成

布尔表达式的计算由条件跳转语句完成：

```

BoolExpression : Expression EQUAL Expression {
    .....
    //generate code
    setResIndex($$, getTemp());
    setResIndex($1, afterCast(getResIndex($1), getType($1), operType($1, $3)));
}

```

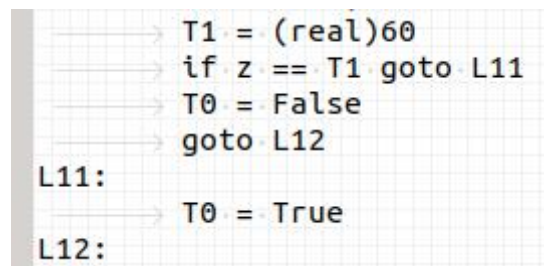


```

        setResIndex($3, afterCast(getResIndex($3), getType($3), operType($1, $3)));
        genCalBoolExpr(getResIndex($$), getResIndex($1), "==", getResIndex($3));
    }
    void genCalBoolExpr(const char* res, const char* oper1, const char* op, const char* oper2) {
        char* labelT = getLabel();
        char* labelO = getLabel();
        fprintf(yyout, "\\tif %s %s %s goto %s\\n", oper1, op, oper2, labelT);
        fprintf(yyout, "\\t%s = False\\n", res);
        genGotoCode(labelO);
        genLabel(labelT);
        fprintf(yyout, "\\t%s = True\\n", res);
        genLabel(labelO);
        free(labelT);
        free(labelO);
    }
}

```

例如表达式 $a == b$ ，如果为真，则将 `True` 赋给存储布尔表达式值的临时变量，否则赋值 `False`。特别的，如果两个表达式的类型不同，还要进行类型转换



```

T1 = (real)60
if z == T1 goto L11
T0 = False
goto L12
L11:
T0 = True
L12:

```

$z == 60$ 生成的中间代码

3.6 if 语句的代码生成

对于一个 if-else 语句：

```

If (bool) begin
    Statements1
end else begin
    Statements2
End

```

，其生成的中间代码格式如下：

```

If false bool goto L1
    Statements1
Goto L2
L1:
    Statements2
L2:

```

.....

根据上述定义，在 IF 语句产生式中插入语义规则：

```

IfStmt : IF LEFTBRACKETS BoolExpression GenIfCode RIGHTBRACKETS Statement ELSE {
    LabelStack[++LabelStackTop]=getLabel();
    genGotoCode(LabelStack[LabelStackTop]);
    genLabel(LabelStack[LabelStackTop - 1]);
} Statement {
    .....
    genLabel(LabelStack[LabelStackTop]);
}

```



```

    free(LabelStack[LabelStackTop]);
    free(LabelStack[LabelStackTop - 1]);
    LabelStackTop -= 2;
};

GenIfCode : {
    LabelStack[++LabelStackTop] = getLabel();
    genIfFalseGoto(getResIndex($0), LabelStack[LabelStackTop]);
};

```

每当识别完 IF 语句中的布尔表达式后，嵌入的文法单元 GenIfCode 所带的语义规则会将该 IF 语句判定为假时跳转到的标签压入栈中储存并调用 genIfFalseGoto() 生成一个条件跳转语句。

在识别 ELSE 后，语义规则会生成一个新标签并压入栈中，这个新标签将会被放在整个 if-else 语句的最后，而后语义规则生成一个跳转到该新标签的跳转语句，使得执行完 Statements1 后不会再执行 Statements2。最后放置一个 IF 语句判定为假时跳转到的标签。

在整个 if-else 语句被规约后，放置跳出整个 if-else 语句的标签，最后将保存的标签全部弹栈。
生成的代码如下：
原码：

```

IF (z == 60) BEGIN
    x := 1;
    IF (y == 50) BEGIN
        x := 3;
    END ELSE BEGIN
        x := 4;
    END
END ELSE BEGIN
    x := 2;
END

```

中间代码：

```

--> T1 = (real)60
--> if z == T1 goto L11
--> T0 = False
--> goto L12
L11:
--> T0 = True
L12:
--> if False T0 goto L13
--> x = 1
--> if y == 50 goto L14
--> T0 = False
--> goto L15
L14:
--> T0 = True
L15:
--> if False T0 goto L16
--> x = 3
--> goto L17
L16:
--> x = 4
L17:
--> goto L18
L13:
--> x = 2
L18:

```

可以根据中间代码得出执行结果：

z	y	x
60	50	3
!60	50	2
60	!50	4
!60	!50	2

与原码逻辑所产生的的结果一致。

以上完成了对 if-else 语句的分析和中间代码生成。

4. 扩展

经过实验一、三和四后，一个 TINY 语言的词法、语法、语义分析器和中间代码生成器的基本框架已经搭好，可以在这个基本框架下进行扩展。

4.1 循环结构

基本的 TINY 语言是没有的循环结构的，于是我在 TINY 的基础上增加了 WHILE 语句。

4.1.1 增加词法分析内容

在实验项目的.l 文件中添加：

```
"WHILE"                {loadYylval(WHILE);return WHILE;}
```

使得词法分析器可以识别 WHILE 关键词并生成相关数据结构。

4.1.2 增加语法、语义分析和中间代码生成内容

对于一个 WHILE 语句：

```
WHILE (BoolExpression) Statement
```

其所生成的中间代码结构如下：

Begin Label:

```
Calculate BoolExpression
If !BoolExpression goto End Label
Statement
goto Begin Label
```

End Label:

.....

根据以上规则在实验项目的.y 文件中，添加：

```
WhileStmt : WHILE
{
LabelStack[++LabelStackTop] = getLabel();
genLabel(LabelStack[LabelStackTop]);/*lay begin label*/
}
LEFTBRACKETS BoolExpression GenIfCode RIGHTBRACKETS Statement {
$$ = mkTreenode(mkTreeNodeContent(writestmt));
insertSonNode($$, $1);
insertSonNode($$, $3);
insertSonNode($$, $4);
insertSonNode($$, $6);
insertSonNode($$, $7);
//generate code
genGotoCode(LabelStack[LabelStackTop - 1]);/*back to begin*/
genLabel(LabelStack[LabelStackTop]);/*jump out while*/
```

```

    free(LabelStack[LabelStackTop]);
    free(LabelStack[LabelStackTop - 1]);
    LabelStackTop -= 2; /*pop out while stmt's label*/
    resetTempIndex();
}
;

```

4.1.3 运行结果

用例：

```

... WHILE (z < 60) BEGIN
...     z := z + 1;
... END

```

运行结果：

```

L8: ← Begin Label
    if z < 60 goto L9
    T0 = False
    goto L10
L9:
    T0 = True
L10:
    if False T0 goto L11
    T2 = (real)1
    T1 = z + T2
    z = T1
    goto L8
L11: ← End Label

```

计算z<60
的值 不符合
条件则
跳出
执行循
环体中
的语句

4.2 布尔表达式的扩展

在基本的 TINY 语言中，布尔表达式只包含对两个表达式值的等值和不等值比较，没有大于、小于、不大于、不小于、且、或这些比较运算符，于是可以对这一部分进行扩展。

4.2.1 增加词法分析内容

在.l 文件中添加：

```

">"          {loadYylval(GT);return GT;}
"<"          {loadYylval(LT);return LT;}
">="         {loadYylval(GE);return GE;}
"<="         {loadYylval(LE);return LE;}
"&&"         {loadYylval(AND);return AND;}
"||"         {loadYylval(OR);return OR;}

```

词法分析器将识别以上新增词法。

4.2.2 增加语法、语义分析和中间代码生成内容

4.2.2.1 大于、小于、不大于、不小于

对于新增的大于、小于、不大于、不小于四种比较，其结构与原有的等值比较一样，只需要修改中间代码生成时比较运算符对应的字符串即可（红色标识）。

```

BoolExpression : Expression GT Expression {
    $$ = mkTreenode(mkTreeNodeContent(bexpression));
    insertSonNode($$, $1);
    insertSonNode($$, $2);
}

```

```

insertSonNode($$, $3);
setType($$, boolean);
//generate code
setResIndex($$, getTemp());
setResIndex($1, afterCast(getResIndex($1), getType($1), operType($1, $3)));
setResIndex($3, afterCast(getResIndex($3), getType($3), operType($1, $3)));
genCalBoolExpr(getResIndex($$), getResIndex($1), ">", getResIndex($3));
}

```

4.2.2.2 且运算和或运算

对于且运算和或运算，则需要增加一个新的文法单元 BoolExpressions，其产生式如下：

```

BoolExpressions -> BoolExpressions AND BoolExpressions
                  | BoolExpressions OR BoolExpressions
                  | BoolExpression

```

同时在.y 文件中加入 token：

```

%left OR
%left AND

```

该代码使得且运算的优先级大于或运算并且两者皆为左结合。

4.2.2.2.1 且运算

且运算的语法、语义分析和中间代码生成的实现如下：

```

BoolExpressions : BoolExpressions AND BoolExpressions {
    $$ = mkTreenode(mkTreeNodeContent(bexpressions));
    insertSonNode($$, $1);
    insertSonNode($$, $2);
    insertSonNode($$, $3);
    setType($$, boolean);
    setResIndex($$, getTemp());
    //generate code
    char* falseLabel = getLabel();
    char* escapeLabel = getLabel();
    genIfFalseGoto(getResIndex($1), falseLabel);
    genIfFalseGoto(getResIndex($3), falseLabel);
    genCopyCode(getResIndex($$), "True");//The result is true
    genGotoCode(escapeLabel);
    genLabel(falseLabel);
    genCopyCode(getResIndex($$), "False");
    genLabel(escapeLabel);
    free(falseLabel);
    free(escapeLabel);
}

```

首先是语法和语义分析部分，该语义规则将产生式右边的三个文法单元已经规约产生的三棵语法树作为新生成树节点子树，完成语法树的递归构造。语义分析部分则将规约成的文法单元的数值类型设置为布尔，运行结果如下：

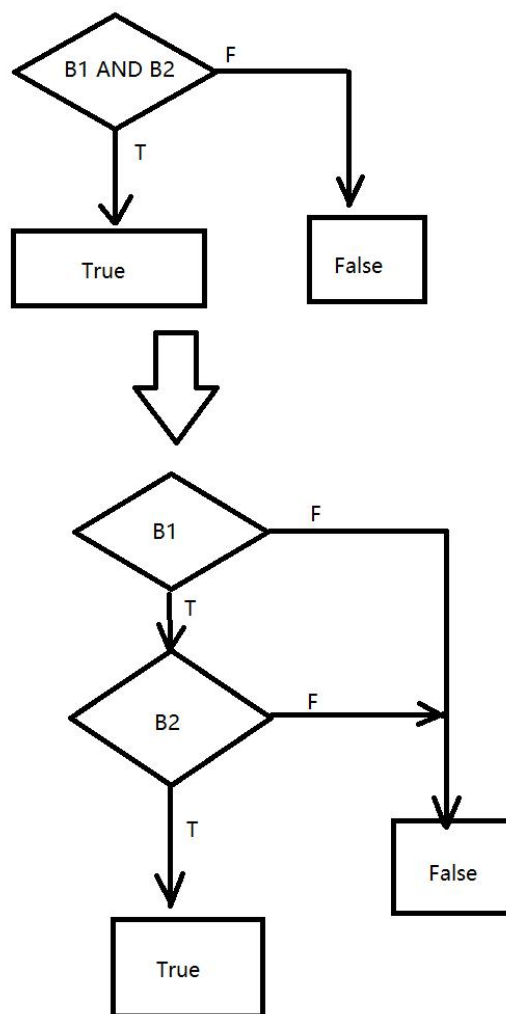
```

|- Nonterminal = BoolExpressions ,dataType = boolean
|
|- Nonterminal = BoolExpressions ,dataType = boolean
|
|   |- Nonterminal = BoolExpression ,dataType = boolean
|   |
|   |   |- Nonterminal = Expression ,dataType = real
|   |   |
|   |   |   |- Nonterminal = MultiplicativeExpr ,dataType = real
|   |   |   |
|   |   |   |   |- Nonterminal = PrimaryExpr ,dataType = real
|   |   |   |   |
|   |   |   |   |   |- Token = z
|   |   |
|   |   |- Token = <
|   |
|   |- Nonterminal = Expression ,dataType = integer
|   |
|   |   |- Nonterminal = MultiplicativeExpr ,dataType = integer
|   |   |
|   |   |   |- Nonterminal = PrimaryExpr ,dataType = integer
|   |   |   |
|   |   |   |   |- Token = 60
|   |
|   |- Token = &&
|
|- Nonterminal = BoolExpressions ,dataType = boolean
|
|   |- Nonterminal = BoolExpression ,dataType = boolean
|   |
|   |   |- Nonterminal = Expression ,dataType = integer
|   |   |
|   |   |   |- Nonterminal = MultiplicativeExpr ,dataType = integer
|   |   |   |
|   |   |   |   |- Nonterminal = PrimaryExpr ,dataType = integer
|   |   |   |   |
|   |   |   |   |   |- Token = y
|   |   |
|   |   |- Token = ==
|   |
|   |- Nonterminal = Expression ,dataType = integer
|   |
|   |   |- Nonterminal = MultiplicativeExpr ,dataType = integer
|   |   |
|   |   |   |- Nonterminal = PrimaryExpr ,dataType = integer
|   |   |   |
|   |   |   |   |- Token = 100

```

z < 60 && y == 100 的语法树

其次是中间代码生成。原理通过程序框图展示：



对于任意一个且运算多条件判断语句（上图中上面的程序框图），都可以分解成单条件判断语句的组合（上图中下面的程序框图）。于是通过语义规则中//generate code 下方的代码可以完成对 AND 多条件布尔表达式的代码生成。运行结果如下：

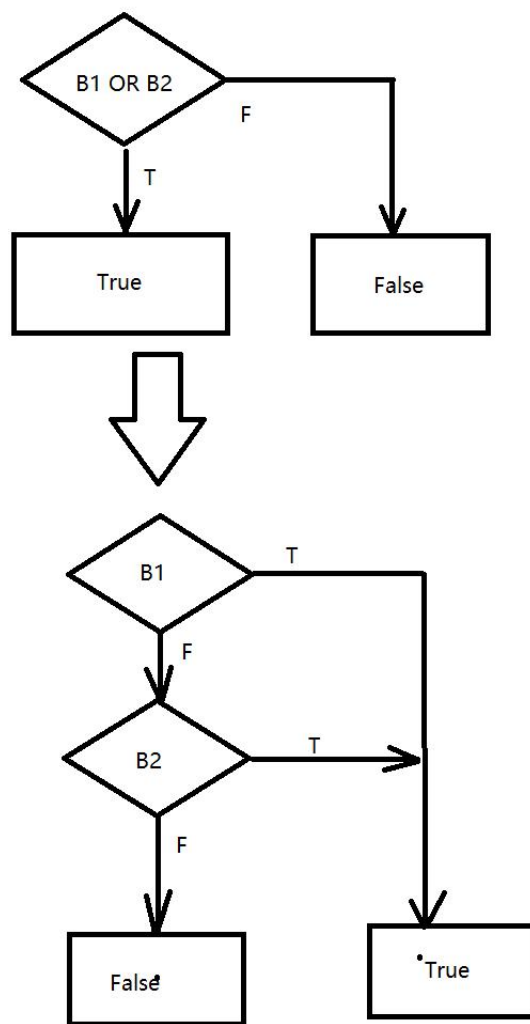
```

T1 = (real)60
if z < T1 goto L20
T0 = False      计算z < 60的布尔值 T0
goto L21
L20:
T0 = True
L21:
if y == 100 goto L22
T2 = False      计算y == 100的布尔值 T2
goto L23
L22:
T2 = True
L23:
if False T0 goto L24
if False T2 goto L24
T3 = True        计算T0 AND T2的布尔值
goto L25
L24:
T3 = False
L25:
  
```

$z < 60 \ \&\& \ y == 100$ 生成的中间代码

4.2.2.2.2 或运算

或运算的语法和语义分析和且运算的一致，只在代码生成部分有差别，其转换的程序框图示意图如下：



中间代码生成的运行结果如下：

```

→ T1 = (real)60
→ if z < T1 goto L20
→ T0 = False 计算z < 60
→ goto L21    的布尔值T0
L20:
→ T0 = True
L21:
→ if y == 100 goto L22
→ T2 = False 计算y == 100
→ goto L23    的布尔值T2
L22:
→ T2 = True
L23:
→ if T0 goto L24
→ if T2 goto L24
→ T3 = False 计算T0 OR
→ goto L25    T2的布尔值
L24:
→ T3 = True T3
L25:

```

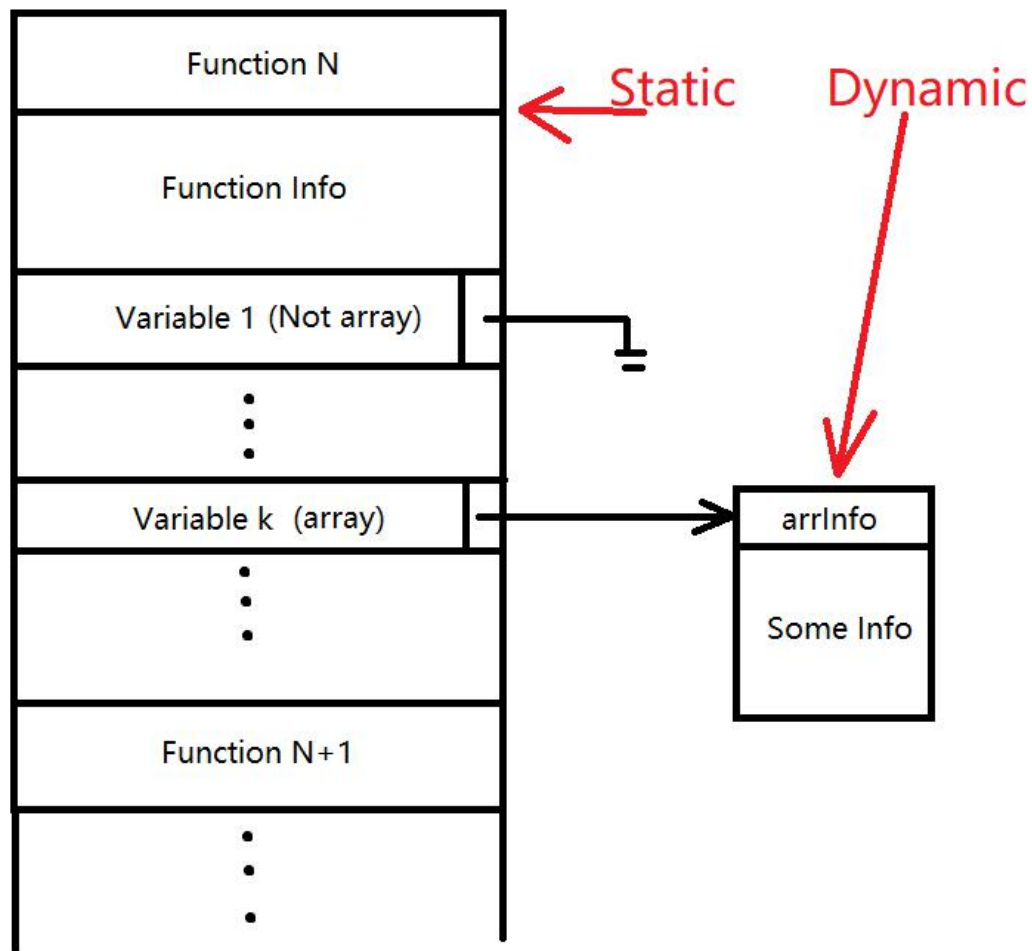
$z < 60 \parallel y == 100$ 生成的中间代码

4.3 数组

4.3.1 符号表的扩展

先前构造的基本符号表只能存储单数值变量的信息，因此需要扩展符号表以存储数组变量的信息。考虑到给每个变量都分配一张数组信息表可能造成空间的浪费，故数组信息表的空间为动态分配，扩

展后符号表结构示意图如下：



其中 arrInfo 的定义如下：

```
typedef struct {  
    int arrDataType;  
    unsigned int dimension;  
    unsigned int dimensionSize[MAX_DIMENSION];  
}arrInfo;
```

arrDataType 记录了数组元素的类型，dimension 记录数组的维数，dimensionSize 记录了每一维的大小。通过这个扩展的符号表，便可以进行数组变量信息的存储和查询。

4.3.2 词法分析的扩展

词法分析部分只加入左右方括号这两个新词法单元：

```
"["    {loadYylval(LEFTSQUAREBRACKETS);return LEFTSQUAREBRACKETS;}  
"]"    {loadYylval(RIGHTSQUAREBRACKETS);return RIGHTSQUAREBRACKETS;}
```

4.3.3 语法、语义分析的扩展

在我所定义的 TINY 扩展语言中，数组在被定义时，只能用整数常量定义每个维度的大小，而不能用变量或者表达式，即：

```
INT arr[3][2];//合法
```

```
INT arr[x][3 + 5];//不合法
```

但是当数组被引用时，下标可以是表达式，即：

```
arr[3][2] := 5;//合法
```

```
y := arr[x][3 + 5];//合法
```

因此，为了对这两种情况进行区分，我新定义两个文法单元。

4.3.3.1 ConstIndexs

该文法单元的产生式如下：

```
ConstIndexs -> ConstIndexs LEFTSQUAREBRACKETS INTNUMBER RIGHTSQUAREBRACKETS  
| LEFTSQUAREBRACKETS INTNUMBER RIGHTSQUAREBRACKETS
```

在用上述产生式进行规约时，其语义规则中的代码：

```
sscanf($3->nodeContent->tokenStr, "%d", &dSize);  
dimensionSizeStack[++dimensionSizeStackTop] = dSize;  
setIndexsDim($$, getIndexsDim($1) + 1);
```

会将整数常量词法单元的值压入栈中，并记录当前的维度，该栈和记录的维度最终会在产生式：

```
LocalVarDecl : Type ID ConstIndexs SEMICOLON {  
    addArr($2->nodeContent->tokenStr, getType($1), getIndexsDim($3), dimensionSizeStack +  
    dimensionSizeStackTop + 1 - getIndexsDim($3));  
    dimensionSizeStackTop -= getIndexsDim($3);  
}
```

的规约中被作为参数，调用 addArr()，完成数组信息的记录。

4.3.3.2 VarIndexs

该文法单元的产生式如下：

```
VarIndexs -> VarIndexs LEFTSQUAREBRACKETS Expression RIGHTSQUAREBRACKETS  
| LEFTSQUAREBRACKETS Expression RIGHTSQUAREBRACKETS
```

再用上述产生式进行规约时，其语义规则中的代码：

```
setResIndex($3, afterCast(getResIndex($3), getType($3), integer));  
dimensionOffsetStack[++dimensionOffsetStackTop] = getResIndex($3);  
setIndexsDim($$, getIndexsDim($1) + 1);
```

会将存储表达式的值的临时变量类型转换为整型（如果不为整型）并压入栈中，同时记录维度。这些信息的使用在下一节中体现。

4.3.3.3 数组的引用

```
PrimaryExpr : ID VarIndexs {
```

```
    .....  
    //check var table  
    //generate code  
    if(checkArr($1->nodeContent->tokenStr) == 0) { //valid  
        char* offsetVar = getTemp();  
        unsigned int size;  
        char t[256];  
        genCopyCode(offsetVar, "0");  
        for(int i = dimensionOffsetStackTop, c = getIndexsDim($2); c >= 1; c--, i--) {  
            sprintf(t, "%s * %u", dimensionOffsetStack[i],  
getSubSpaceSize($1->nodeContent->tokenStr, c - 1));  
            genTriCode(offsetVar, offsetVar, "+", t);  
        }  
        sprintf(t, "%s[%s]", $1->nodeContent->tokenStr, offsetVar);  
        setResIndex($$, strDeepCopy(t));  
        setType($$, getArrType($1->nodeContent->tokenStr));  
    } else {  
        setResIndex($$, strDeepCopy("ERROR"));  
        setType($$, integer); //default type  
    }  
}
```

当用该产生式进行规约时,语义规则会用存储在 dimensionOffsetStack 栈中值和文法单元 VarIndexs 的维度属性进行偏移值的计算:

```
for(int i = dimensionOffsetStackTop, c = getIndexsDim($2); c >= 1; c--, i--) {
    sprintf(t, "%s * %u", dimensionOffsetStack[i], getSubSpaceSize($1->nodeContent->tokenStr, c - 1));
    genTriCode(offsetVar, offsetVar, "+", t);
}
```

其中函数 getSubSpaceSize()会计算数组某一维的子空间的大小, 如数组:

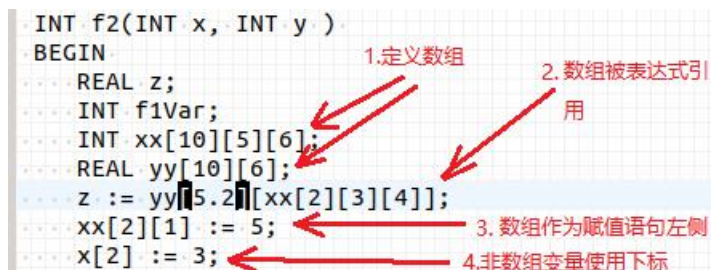
```
INT arr[10][5][6]
```

计算 arr[3]的偏移值时, getSubSpaceSize()会计算并返回 $5 * 6 * \text{sizeof}(\text{INT})$, 最终计算出偏移值为 $3 * 5 * 6 * \text{sizeof}(\text{INT})$ 。

在我所定义的数组中, []运算符只做偏移值计算, 不做类型计算, 即 $\text{Type}(\text{arr}[3]) == \text{Type}(\text{arr}[3][5][6]) == \text{INT}$ 。所以数组所规约出的 PrimaryExpr 的类型为该数组的元素类型。

4.3.4 运行结果

用例:



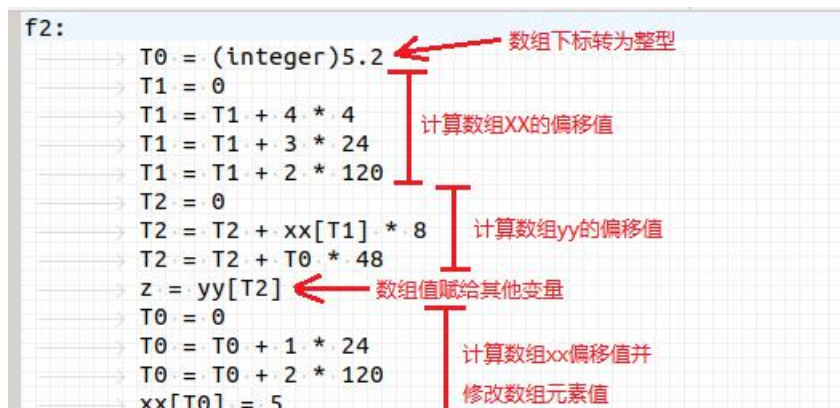
```
INT f2(INT x, INT y )
BEGIN
    REAL z;
    INT f1Var;
    INT xx[10][5][6];
    REAL yy[10][6];
    z := yy[5.2][xx[2][3][4]];
    xx[2][1] := 5;
    x[2] := 3;
```

运行结果:

1:

```
-----Function Table-----
Function1 : f2
Argument type : integer integer
Variable table:
    x integer 0
    y integer 4
    z real 8
    f1Var integer 16
    xx array 20 dimension:10 5 6
    yy array 1224 dimension:10 6
```

2、3:



```
f2:
    T0 = (integer)5.2
    T1 = 0
    T1 = T1 + 4 * 4
    T1 = T1 + 3 * 24
    T1 = T1 + 2 * 120
    T2 = 0
    T2 = T2 + xx[T1] * 8
    T2 = T2 + T0 * 48
    z = yy[T2]
    T0 = 0
    T0 = T0 + 1 * 24
    T0 = T0 + 2 * 120
    xx[T0] = 5
```

4:

```
samchen2@ubuntu:~/tiny/code3 (extend arr)$ ./run.sh  
error at line 10:x is not an array
```