

编译原理实验

实验四：语义分析程序及中间代码生成实验

姓名：陈家豪

班级：2018 级计科 1 班

学号：18308013

目录

1	实验目的	2
2	实验内容	2
3	实验要求	2
4	算法原理与描述	2
4.1	TINY+ 语言定义	2
4.2	语义分析	4
4.3	三地址中间代码与中间代码生成	5
5	实验过程和关键代码说明	6
5.1	前提背景	6
5.2	符号表的设计	6
5.2.1	符号表结构	6
5.2.2	符号表的相关操作	9
5.3	语义分析与中间代码生成	11
5.3.1	high-level program structures	11
5.3.2	statements	14
5.3.3	expressions	17
5.3.4	自动类型转换	23
6	实验结果	24
6.1	语义分析	24
6.1.1	符号表的构建与维护	24
6.1.2	声明检查	25
6.1.3	使用检查	27
6.1.4	类型检查	28
6.1.5	自动类型转换	29
6.1.6	参数检查	30
6.1.7	返回检查	32
6.2	中间代码生成	34
7	实验总结	35

1 实验目的

构造 TINY + 的语义分析程序并生成中间代码。

2 实验内容

构造符号表，用 C 语言扩展 TINY 的语义分析程序，构造 TINY + 的语义分析器，构造 TINY + 的中间代码生成器。

3 实验要求

能检查一定的语义错误，将 TINY + 程序转换成三地址中间代码。

4 算法原理与描述

4.1 TINY+ 语言定义

在上一次实验中，我们已经尝试将原始 TINY 语言进行自定义拓展，得到 TINY+ 语言。我们认为在此重复一遍我们的 TINY + 语言的新特性和相关细节是必要的。此外，上次实验完成后，我发现我定义的 TINY+ 语法存在操作符优先级的問題。在本次实验中我修复了这个问题。

词法上，TINY+ 增加了以下关键字：**WHILE,AND,OR,BOOL,TRUE** 和 **FALSE**。其中 **WHILE** 用于增加对 while 语句的支持，其余 5 个新增关键字用于增加对布尔值的支持。此外，单符操作符中增加了 % 用于求余运算，以及 ! 用于否定；增加了单符比较符 > 和 <，以及多符比较符 >= 和 <=。

语法上，TINY+ 增加了以下语法支持：

- (1) 支持声明定义布尔类型变量和布尔类型函数；
- (2) 支持 while 语句；
- (3) 允许在声明语句中同时声明多个同一类型的变量并赋值；
- (4) 支持求模（%）运算；
- (5) 支持 <,<=,> 和 ≥ 比较运算；
- (6) 支持在判断中使用 **AND** 和 **OR**，含义类似于 C/C++ 的 && 和 ||；
- (7) 支持使用布尔表达式对变量赋值；
- (8) 支持单个符号加分号组成语句，如 “x;” 是允许的（虽然没有意义）；

TINY+ 的语法结构依然分为三个层次。对于 high-level program structures, 其 EBNF 描述如下:

$$\begin{aligned}
Program &\rightarrow FunctionDecl \ FunctionDecl^* \\
FunctionDecl &\rightarrow Type \ [\ \mathbf{MAIN} \] \ \mathbf{id} \ ' (' \ [\ FormalParams \] \)' \ Block \\
FormalParams &\rightarrow FormalParam \ (\ ',' \ FormalParam \)^* \\
FormalParam &\rightarrow Type \ \mathbf{id} \\
Type &\rightarrow \mathbf{INT} \ | \ \mathbf{REAL} \ | \ \mathbf{BOOL}
\end{aligned}$$

相较于 TINY, *Type* 增加了与 **BOOL** 有关的修改。

对于 statements, 其 EBNF 描述如下:

$$\begin{aligned}
Block &\rightarrow \mathbf{BEGIN} \ Statement^* \ \mathbf{END} \\
Statement &\rightarrow Block \\
&\quad | \ LocalVarDecl \\
&\quad | \ AssignStmt \\
&\quad | \ ReturnStmt \\
&\quad | \ IfStmt \\
&\quad | \ WhileStmt \\
&\quad | \ WriteStmt \\
&\quad | \ ReadStmt \\
LocalVarDecl &\rightarrow Type \ AssignExpr \ (\ ',' \ AssignExpr \)^* \ ';' \\
AssignStmt &\rightarrow AssignExpr \ ';' \\
ReturnStmt &\rightarrow \mathbf{RETURN} \ BoolMultiExpr \ ';' \\
IfStmt &\rightarrow \mathbf{IF} \ ' (' \ BoolMultiExpr \)' \ Statement \ [\mathbf{ELSE} \ Statement] \\
WhileStmt &\rightarrow \mathbf{WHILE} \ ' (' \ BoolMultiExpr \)' \ Statement \\
WriteStmt &\rightarrow \mathbf{WRITE} \ ' (' \ BoolMultiExpr \ ',' \ \mathbf{string} \)' \ ';' \\
ReadStmt &\rightarrow \mathbf{READ} \ ' (' \ \mathbf{id} \ ',' \ \mathbf{string} \)' \ ';'
\end{aligned}$$

相较于 TINY, *Statement* 增加了与 *WhileStmt* 有关的修改; 新增了 *WhileStmt* 并对其进行了相关描述; *LocalVarDecl* 进行了修改, 支持多变量声明与赋值; *ReturnStmt* 和 *WriteStmt* 的 *Expression* 升级为 *BoolMultiExpr*。

对于 expressions，其 EBNF 描述如下：

$$\begin{aligned}
 \text{Expression} &\rightarrow \text{MultiplicativeExpr } (('+' \mid '-') \text{ MultiplicativeExpr })^* \\
 \text{MultiplicativeExpr} &\rightarrow \text{PrimaryExpr } (('*' \mid '/' \mid \%) \text{ PrimaryExpr })^* \\
 \text{PrimaryExpr} &\rightarrow ('!' \mid '-') (\text{num} \mid \text{id} \\
 &\quad \mid \text{TRUE} \\
 &\quad \mid \text{FALSE} \\
 &\quad \mid '(' \text{ BoolMultiExpr } ')' \\
 &\quad \mid \text{id } '(' [\text{ActualParams}] ')') \\
 \text{AssignExpr} &\rightarrow \text{id } [':' \text{ BoolMultiExpr }] \\
 \text{BoolMultiExpr} &\rightarrow \text{BoolMidExpr } ('AND' \text{ BoolMidExpr })^* \\
 \text{BoolMidExpr} &\rightarrow \text{BoolExpression } ('OR' \text{ BoolExpression })^* \\
 \text{BoolExpression} &\rightarrow \text{Expression } [('!=' \mid '==' \mid '>=' \mid '<=' \mid '>' \mid '<') \text{ Expression }] \\
 \text{ActualParams} &\rightarrow \text{BoolMultiExpr } (',' \text{ BoolMultiExpr })^*
 \end{aligned}$$

相比于 TINY，*MultiplicativeExpr* 增加了对 % 的支持；*PrimaryExpr* 增加了单目运算符的支持和对 **TRUE**、**FALSE** 的支持，并将 *Epression* 改为 *BoolMultiExpr*；新增 *AssignExpr*，*BoolMultiExpr* 和 *BoolMidExpr*，并对它们进行了相关描述；*BoolExpression* 增加了对新增比较符的支持。

以上就是 TINY+ 的相关描述。

4.2 语义分析

语义分析是词法分析和语法分析后的第三个检查步骤，由语义分析器完成。语义分析器使用语法树和符号表中的信息来检查源程序是否和语言定义的语义一致。其同时也收集类型信息，并把这些信息存放在语法树或符号表中，以便在随后的中间代码生成过程中使用。

一般来说，语义分析包含类型检查、自动类型转换等内容。在本次实验中，我们计划并实现以下语义分析内容：

- (1) **符号表的构建与维护**：在语义检查时，将声明的函数名和变量名以及相关信息作为符号项添加进符号表，并为不同的作用域创建各自的符号表；
- (2) **声明检查**：在声明函数和声明变量时，将通过符号表进行检查，确保当前作用域没有同名的函数或变量。声明检查还包括了在声明函数时对函数参数是否重名的检查。
- (3) **使用检查**：在程序使用到某个变量或函数时，将通过符号表进行检查，确保变量或函数于使用前已在同一作用域或包含所在作用域的父作用域声明。此外我们还要通过符号表获取相关信息，用于后续检查；
- (4) **类型检查**：在生成指令时，确保运算符具有匹配的运算分量，拥有正确的数据类型；
- (5) **自动类型转换**：当要求类型和当前类型不匹配时，将尝试隐式自动类型转换，如果转换失败则报错。我们实现的 TINY+ 相关程序中总共有 INT、REAL 和 BOOL 三种需要检查的类型。我采用和 C 语

言一样的转换形式，即 INT 和 REAL 之间支持自动类型转换，INT 和 REAL 支持自动类型转换至 BOOL，BOOL 不支持自动转换至 INT 和 BOOL；

(6) **参数检查**：在调用函数时，将对参数数量和参数类型进行检查，确保传入参数和函数声明的参数一致；

(7) **返回检查**：函数里面必须包含返回语句，并且返回语句的返回值类型必须与函数的返回类型相匹配；

4.3 三地址中间代码与中间代码生成

在对程序代码执行词法分析、语法分析和语义分析后，我们需要将程序代码转换成中间代码。中间代码是编译器前端的最终输出结果。它将被交给编译器后端生成真正的机器码。一种常用的中间代码是三地址中间代码。

三地址代码有两个基本概念，分别是地址和指令。地址相当于一个可被操作 (运算) 的对象，而指令相当于对地址进行操作的方法命令。在三地址代码中，一条指令的右侧最多有一个运算符。由于一条指令至多还有三个地址 (运算对象)，故这种代码被称为三地址代码。

本次实验我们将采用龙书提供的常见三地址代码指令形式。不过，由于 TINY+ 语言的特殊性，以及龙书的三地址代码指令集缺乏一些关键指令细节、存在部分 TINY+ 无需使用的指令，我们增加了部分必要的指令形式。以下是我们实验将要采用的三地址代码指令集，标黑指令为龙书原有内容，标蓝指令为我们新增或修改的内容：

(1) **$x = y \text{ op } z$** ：三元赋值指令。其中 **op** 是一个双目算术符或逻辑运算符， **x, y, z** 是地址。在本次实验中，三元赋值指令不能应用于逻辑运算上， **$op \in \{+, -, *, /, \%\}$** ；

(2) **$x = \text{op } y$** ：二元赋值指令。其中 **op** 是一个单目运算符或类型转换符。在本次实验中， **$op \in \{!, \text{minus}, (\text{int}), (\text{real})\}$** ；

(3) **$x = y$** ：复制指令；

(4) **$\text{goto } L$** ：无条件转移指令。下一步要执行的指令是带有标号 **L** 的三地址指令，然后顺序执行；

(5) **$\text{if } x \text{ goto } L$** 与 **$\text{if False } x \text{ goto } L$** ：一元条件转移指令，分别当 **x** 为真或为假时，下一步将执行带有标号 **L** 的指令、然后顺序执行，否则下一步继续顺序执行序列中的指令；

(6) **$\text{if } x \text{ relop } y \text{ goto } L$** ：二元条件转移指令。其中 **relop** 是二元比较算术符。当 **$x \text{ relop } y$** 为真时，下一步执行带有标号 **L** 的三地址指令、然后顺序执行，否则下一步继续顺序执行序列中的指令。在本次实验中， **$\text{relop} \in \{!, =, ==, >=, <=, >, <\}$** ；

(7) **$\text{param } x$** ：传递指令。参数的传递通过一个栈来操作。在调用函数前，需要使用该指令将 **x** 的值压栈，以供后面函数调用时使用；

(8) **$x = \text{call } y, n$** ：调用指令。其中 **x** 是函数返回值存储的地址，**y** 是函数名，**n** 是将要传递的函数参数个数。然后，程序会跳转至带有 **y** 标签的指令顺序执行，直到遇到返回指令才会返回到调用指令、将返回值赋值给 **x** 后继续顺序执行。需要注意的是，TINY + 中强制要求函数存在返回值，故“ **$x =$** ”不可缺少；

- (9) **get x** : 接收指令。其一般位于函数的起始位置。该指令将参数栈的一个值弹栈，并存储在地址 x 上；
- (10) **return x** : 返回指令。函数主体部分将通过这个指令，将地址 x 的值进行返回，然后跳转至（时间上）最近使用的调用指令；
- (11) **Label L** : 标签指令。严格意义上其不是一条指令。其会将在序列中紧跟自己的后续指令打上 L 标签。 L 既可以是标号，也可以是函数名。**Label MAIN** 是整个程序的起始执行位置；

使用上述指令，我们就可以把 TINY+ 源程序翻译成三地址码版本。由于翻译过程存在大量细节和注意事项，无法从抽象高度上进行总结，故我们把翻译原理和过程放在章节5.3中，结合实际代码共同说明。

5 实验过程和关键代码说明

5.1 前提背景

在上次的实验中，我们已经实现了 TINY+ 的词法分析和语法分析，并得到了源程序的语法树。本次实验，我们将基于上一次实验的代码和结果进行拓展，在语法树上进行语义分析和中间代码生成，而不是修改上次实验的代码、在生成语法树的同时进行语义分析和中间代码生成。这样，我们只需要实现一个以语法树为输入的语义检查器和中间代码生成器，不需要对原始代码的逻辑进行修改。但这样也意味着**语义检查器和中间代码生成器接收的语法树必须是绝对正确的**（事实上也应该绝对正确，不然会在词法分析或语法分析上报错）。所幸的是我在上次实验完成的语法树生成器质量较高，目前没发现有 bug。

语义分析和中间代码生成都需要对语法树遍历。需要注意的是，语义分析和中间代码生成是可以并行遍历、执行的，中间代码生成只依赖当前语义分析的结果。因此，我将语义分析和中间代码生成结合起来，这样只需要通过一次遍历，就可以实现在语义分析的同时生成中间代码。

具体操作上，我使用一个文件指针 fp 在函数之间传递，在语义检查的同时通过 fp 输出相应的三地址码到指定文件上。当语义检查完成时，也就得到了程序的三地址码文本文件。

5.2 符号表的设计

5.2.1 符号表结构

符号表是一种供编译器用于保存有关源程序构造的各种信息的数据结构，是语义检查和中间代码生成的核心工具。符号表中的每个条目包含一个标识符相关信息，例如其字符串 (名称)、类型、存储位置和相关信息。

考虑到符号表涉及到大量的插入、查询操作，而对符号项的删除操作几乎不存在，因此我采用双向链表的形式实现符号表，相关数据类型和结构的声明定义代码如下所示：

```
1 // 符号项类型
2 typedef enum SymbolType
3 {
4     ST_FUNC, // 函数
5     ST_VAR,  // 变量
```

```

6     ST_TABLE, // block符号表
7     ST_HEAD,  // 符号表头部，无意义
8     ST_ANY    // 未知
9 } SymbolType;
10
11 // 符号值类型
12 typedef enum ValueType
13 {
14     VT_INT,
15     VT_REAL,
16     VT_BOOL,
17     VT_STRING,
18     VT_VOID,
19     VT_ANY // 未知
20 } ValueType;
21
22 // 符号项结构
23 typedef struct Symbol
24 {
25     struct Symbol* prevSymPointer; // 指向当前符号表上一个Symbol的指针
26     struct Symbol* nextSymPointer; // 指向当前符号表下一个Symbol的指针
27     SymbolType symType;           // 符号类型
28     token* tp;                    // token指针
29     struct Symbol* parentSymPointer; // 指向父符号表所属的Symbol的指针
30     struct Symbol* childSymPointer; // 指向属于自己的子符号表的头部Symbol的指针
31     ValueType valType;            // 符号值类型
32     Bool isVariable;              // 符号值是否可变
33     Bool isParam;                 // 是否为参数
34     char* newName;                // 三地址码命名
35     int address;                  // 符号地址
36     int size;                     // 符号字节数
37 } Symbol;

```

SymbolType 定义了一个符号项类型。对于一个符号表，其头部的符号项类型必须为 ST_HEAD；函数名和变量名对应的符号项的类型分别为 ST_FUNC 和 ST_VAR；当出现嵌套作用域时，需要在当前作用域的符号表插入一个类型为 ST_TABLE 的符号项，然后在这个符号项上使用指针 childSymPointer 指向嵌套作用域的符号表。此外，对于声明定义函数时引起的嵌套作用域，不需要创建 ST_TABLE 符号项、而是直接使用相关的 ST_FUNC 符号项作为嵌套作用域的父亲符号项。

ValueType 定义了一个符号的值类型。对于变量而言，ValType 即为自己的类型；对于函数而言，ValType 即为自己的返回值类型。

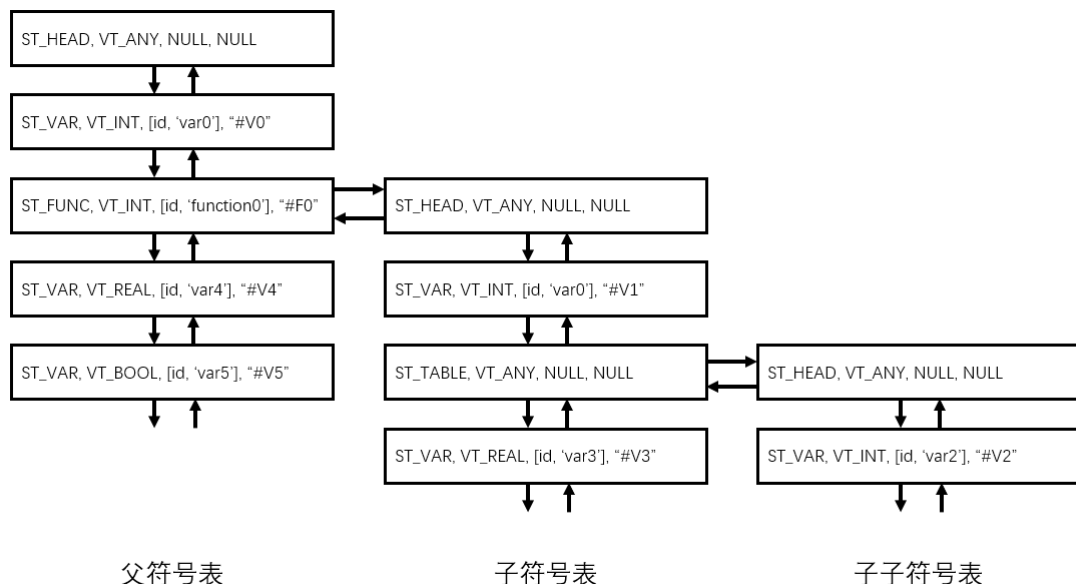


图 1: 符号表结构图例。每个矩形是一个符号项，符号项之间通过指针连接。符号项上注明的四个值从左到右分别为符号项类型、符号值类型、token 和三地址码名。

一个典型的符号表结构如图1所示，这个图清晰地体现出了符号表元素之间的关系。需要注意的是，在子符号表和父符号表中存在相同名字“var0”的变量。这种操作是允许的，正如 C 语言一样，在不同作用域可以声明相同的变量名，对原有变量名进行覆盖。在使用变量时，顺序是从当前作用域开始查找、逐层往外：

```

1  {
2      int x = 1;
3      int y = 2;
4      {
5          int x = 3;
6          int z = 4;
7          printf("%d", x); // output: 3
8          printf("%d", y); // output: 2
9      }
10     printf("%d", x); // output: 1
11     printf("%d", z); // ERROR!
12 }
```

图 2: 不同作用域下变量的声明与使用

显然不同作用域的同名变量是不同的，如何在生成的三地址码中进行区分呢？这就是为什么需要三地址码命名 **newName** 的原因。对于每个声明的变量、函数，我们分配全局唯一的新名字，然后在三地址中间代码中我们使用新名字作为地址、而不是原有的名字。我定义了六种类型的三地址码命名，其中

id 为整数，与前面的字符串进行拼接：

- (1) “#V”+id: 声明变量；
- (2) “#C”+id: 声明常量；
- (3) “#TV”+id: 临时变量；
- (4) “#TC”+id: 临时常量；
- (5) “#F”+id: 声明函数；
- (6) “#L”+id: 标签；

5.2.2 符号表的相关操作

对符号表的操作主要有两种，分别是插入操作和查询操作。

对于插入操作，我们在符号表的末尾进行插入操作，这样可以保证符号表中符号项的排列顺序是与程序的执行顺序是一致的。此外，插入操作有两种情况，一种是在当前符号表的末尾插入、一种是在当前符号表的末尾构建新的子符号表。实现的函数代码如下所示：

```
1 Symbol* buildNewSymbol(Symbol* latedPointer, Bool isChild) {
2     Symbol *latedSymPointer = (Symbol*)calloc(1, sizeof(Symbol));
3     if (isChild) {
4         latedSymPointer->parentSymPointer = latedPointer;
5         latedSymPointer->prevSymPointer = NULL;
6         latedSymPointer->symType = ST_HEAD;
7     }
8     else {
9         latedSymPointer->parentSymPointer = NULL;
10        latedSymPointer->prevSymPointer = latedPointer;
11        latedSymPointer->symType = ST_ANY;
12    }
13    latedSymPointer->nextSymPointer = NULL;
14    latedSymPointer->tp = NULL;
15    latedSymPointer->childSymPointer = NULL;
16    latedSymPointer->valType = VT_ANY;
17    latedSymPointer->isVariable = False;
18    latedSymPointer->address = -1;
19    latedSymPointer->size = -1;
20    latedSymPointer->newName = NULL;
21    if (latedPointer!=NULL) {
22        if (isChild)
23            latedPointer->childSymPointer = latedSymPointer;
24        else
25            latedPointer->nextSymPointer = latedSymPointer;
26    }
27    return latedSymPointer;
28 }
```

对于查询操作，我们有两种查询需求，第一种是在声明变量和函数时，需要判断当前作用域是否已存在同名变量/函数；第二种是在使用变量/调用函数时，需要判断当前作用域和包含当前作用域的作用域是否包含该变量/函数，然后根据就近原则选择。在上面我们说到，我们的符号表的表项顺序是按照程序执行的顺序排序的，而索引指针指向当前符号表的末尾。因此，对于第一种查询，我们只需要检查当前符号表是否存在同名变量/函数（因为当前符号表的都是已经声明的）；对于第二种查询，我们需要从表尾到表头、从子表到父表依次查询，一旦查到即返回。相关代码如下所示，图3清晰地表示了两种查询需求的区别：

```
1 // 判断token是否在当前符号表中存在
2 Bool isSymbolExist(const Symbol* latedPointer, const token* tp, SymbolType symType) {
3     Symbol *p = latedPointer;
4     Bool flag = False;
5     while (p!=NULL&&flag==False) {
6         if (isTokenEqual(p->tp, tp)&&(symType==ST_ANY||symType==p->symType))
7             flag = True;
8         p = p->prevSymPointer;
9     }
10    return flag;
11 }
12
13 // 查找在符号表和父符号表中查找name对应的符号项
14 const Symbol* findSymbol(const Symbol* latedPointer, const char* name, SymbolType symType)
15 {
16     while (latedPointer!=NULL&&(latedPointer->prevSymPointer!=NULL||latedPointer->
17         parentSymPointer!=NULL)) {
18         if (latedPointer->symType==symType) {
19             if (isStringEqual(latedPointer->tp->str, name))
20                 return latedPointer;
21         }
22         if (latedPointer->symType==ST_HEAD)
23             latedPointer = latedPointer->parentSymPointer;
24         else
25             latedPointer = latedPointer->prevSymPointer;
26     }
27    return NULL;
28 }
```

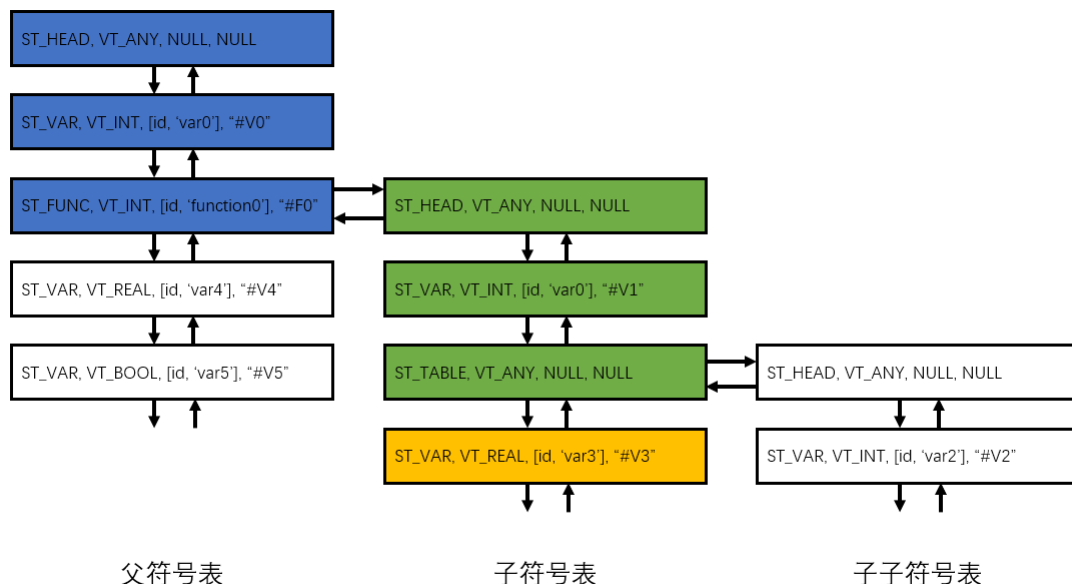


图 3: 假设当前末尾指针指向黄色符号项，声明变量/函数时是在黄色和绿色表项查询是否有同名符号项；在使用变量/函数时是按顺序从下往上、从右往左查询，先后查询黄色表项、绿色表项、蓝色表项，按最近原则选择。

5.3 语义分析与中间代码生成

正如5.1所述，基于语法树，我们同时进行语义分析与中间代码生成。语法树的结构详见 code/myStructure.h，在这里仅展示节点结构代码：

```

1 // 节点
2 typedef struct TreeNode
3 {
4     NodeType nodetype; // 节点类型
5     struct TreeNode** ChildPtrList; // 子节点指针数组的指针
6     int ChildCurrentIndex; // 当前子节点数量
7     int ChildMaxNum; // 子节点指针数组支持的最大数量
8     token* tp; // 叶子节点对应的TOKEN
9
10 } TreeNode;

```

龙书通过使用语法制导的翻译来完成语义分析和中间代码生成。我们参考了其中的一些内容，为语法树的每个节点类型创建不同的函数，参数相当于综合属性和继承属性，通过相互调用来完成语义分析和中间代码生成。在4.1中我们提供了 TINY+ 的 EBNF 形式和语法树层次结构。在这部分，我们将针对不同层次和结构展开说明。

5.3.1 high-level program structures

在正式说明前，我们需要注意的是，每个节点的函数都接收当前符号表末尾指针作为参数（继承属性）、把更新后的符号表末尾指针作为返回值（综合属性）。当返回值为 NULL 时，即说明该函数的语义分析失败，调用该函数的节点函数也应该返回 NULL，最终使得分析程序停止、报出错误信息。

Program 对于 Program，其是多个 FunctionDecl 的排列，我们只需要从左到右依次解析即可。其中 isHasReturn 用于判断 FunctionDecl 是否带有返回语句，若没有则需要报错：

```
1 Symbol* checkProgram(TreeNode* root, Symbol* latedSymPointer, FILE *fp) {
2     for (int i=0; i<root->ChildCurrentIndex; i++) {
3         // 是否有返回语句
4         Bool isHasReturn = False;
5         latedSymPointer = checkFunctionDecl(root->ChildPtrList[i], latedSymPointer, fp, &
6             isHasReturn);
7         if (latedSymPointer==NULL)
8             return NULL;
9         if (isHasReturn == False) {
10             throwError(RowScanIndex, ColScanIndex, "The function doesn't have return
11                 statement.");
12             return NULL;
13         }
14     }
15     return latedSymPointer;
16 }
```

一个函数的三地址码典型结构如图4所示：



图 4: 函数的三地址码结构

FunctionDecl 对于 FunctionDecl，其包含 5~7 个子节点。除了 Block 节点，其他节点均很小，故我们一并在函数中处理它们，避免频繁调用函数。整个函数代码如下所示。需要留意的是，在调用 checkBlock 函数时，我们传递的是子符号表的末尾指针，因为 Block 的作用域为子符号表。而 checkFunctionDecl 执行完毕后返回的是父符号表的末尾指针。父符号表和子符号表的分离操作在这里完成。

```
1 Symbol* checkFunctionDecl(TreeNode* root, Symbol* latedSymPointer, FILE *fp, Bool*
2     isHasReturn) {
3     // 构建函数名符号项
4     latedSymPointer = buildNewSymbol(latedSymPointer, False);
5 }
```

```

4 .....
5 // 设置调用函数标志
6 fprintf(fp, "Label %s\n", latedSymPointer->newName);
7 // 创建子符号表
8 Symbol* newSymPointer = buildNewSymbol(latedSymPointer, True);
9 TreeNode* tempNode;
10 // 遍历各节点
11 for (int childIndex = 0; childIndex < root->ChildCurrentIndex; childIndex++) {
12     tempNode = root->ChildPtrList[childIndex];
13     if (tempNode->nodetype == NT_FACTOR) {
14         updateScanIndex(tempNode->tp);
15         // 存在MAIN, 输出Label MAIN
16         if (tempNode->tp->type == MAIN)
17             fprintf(fp, "Label MAIN\n");
18         else if (tempNode->tp->type == ID) {
19             // 查看是否存在同名函数
20             if (isSymbolExist(latedSymPointer, tempNode->tp, ST_FUNC)) {
21                 throwError(RowScanIndex, ColScanIndex, "A function with the same name
22                     already exists in this scope.");
23                 return NULL;
24             }
25             latedSymPointer->tp = tempNode->tp;
26         }
27     }
28     else if (tempNode->nodetype == NT_TYPE)
29         latedSymPointer->valType = getValueType(tempNode->ChildPtrList[0]->tp->type);
30     else if (tempNode->nodetype == NT_FORMAL_PARAMS) {
31         // 获取参数
32         for (int i = tempNode->ChildCurrentIndex - 1; i >= 0; i -= 2) {
33             newSymPointer = buildNewSymbol(newSymPointer, False);
34             ..... // 更新参数的符号项内容
35             token* idp = tempNode->ChildPtrList[i]->ChildPtrList[1]->tp;
36             updateScanIndex(idp);
37             // 判断参数之间是否重名
38             if (isSymbolExist(latedSymPointer, idp, ST_VAR) == True) {
39                 throwError(RowScanIndex, ColScanIndex, "The parameter with the same
40                     name exists.");
41                 return NULL;
42             }
43             ..... // 更新参数的符号项内容
44             fprintf(fp, "    get %s\n", newSymPointer->newName);
45         }
46     }
47     else if (tempNode->nodetype == NT_BLOCK) {
48         newSymPointer = checkBlock(tempNode, newSymPointer, fp, isHasReturn);
49         if (newSymPointer == NULL)
50             return NULL;
51     }
52 }

```

```

50     else {
51         throwError(RowScanIndex, ColScanIndex, "Unknown error.@1");
52         return NULL;
53     }
54 }
55 return latedSymPointer;
56 }

```

5.3.2 statements

对于 statements 层次, Block 节点的解析较为简单, 从左到右对每个 BEGIN 和 END 之间的节点调用一次 checkStatement 即可。因此在此不赘述 Block 节点的解析。

Statement 节点的解析是最复杂的解析之一。它虽然只有一个子节点, 但子节点有 8 种可能类型。同样, 由于只有 Statement 的解析涉及到这些子节点类型, 故我们将 Statement 节点和其各个子节点的解析整合成一个函数一并处理。

Block 对于 Statement 节点中的 Block 节点, 我们只需要创建一个新的子符号表后调用 checkBlock 即可, 在此不再赘述。

LocalVarDecl 对于 LocalVarDecl, 其子节点 AssignExpr 在这里表示声明变量并赋值, 而在 AssignStmt 中表示执行运算并把值赋给已经声明的变量。故我们分开实现。在 LocalVarDecl 中, 我们需要获取变量的名字和变量, 然后查询符号表、确保在同一作用域不存在同名的已声明变量。然后我们新建符号项, 在调用 checkMultiExpr、输出 BoolMultiExpr 运算过程的三地址码后, 将结果赋给新变量。而在 AssignStmt 中, 我们直接调用 checkAssignExpr 即可:

```

1 Symbol* checkStatement(TreeNode* root, Symbol* latedSymPointer, FILE* fp, Bool*
    isHasReturn) {
2     root = root->ChildPtrList[0];
3     switch (root->nodetype) {
4     .....
5     case NT_LOCAL_VAR_DECL: {
6         ValueType valType = getValueType(root->ChildPtrList[0]->ChildPtrList[0]->tp->type)
            ;
7         for (int i = 1; i < root->ChildCurrentIndex; i += 2) {
8             TreeNode* tempNode = root->ChildPtrList[i];
9             token* tp = tempNode->ChildPtrList[0]->tp;
10            updateScanIndex(tp);
11            // 查看id是否已被声明
12            if (isSymbolExist(latedSymPointer, tp, ST_VAR)) {
13                throwError(RowScanIndex, ColScanIndex, "A variable with the same name
                    already exists in this scope.");
14                return NULL;
15            }
16            ..... //创建并更新符号项
17            // 存在赋值语句, 运行
18            if (tempNode->ChildCurrentIndex > 1)

```

```

19         .....
20     }
21     updateScanIndex(root->ChildPtrList[root->ChildCurrentIndex - 1]->tp);
22     break;
23 }
24 case NT_ASSIGN_STMT: {
25     char* tempName = NULL;
26     ValueType valType;
27     latedSymPointer = checkAssignExpr(root->ChildPtrList[0],
28         latedSymPointer, &tempName, &valType, fp);
29     if (latedSymPointer == NULL)
30         return NULL;
31     updateScanIndex(root->ChildPtrList[root->ChildCurrentIndex - 1]->tp);
32     break;
33 }
34 .....
35 }
36 }

```

ReturnStmt 对于 ReturnStmt, 我们需要获取函数的值类型, 然后传递给 checkBoolMultiExpr, 要求 BoolMultiExpr 得到的这个类型的结果。若执行失败, checkBoolMultiExpr 会返回 NULL 值。此外, 用于判断函数是否有 return 语句的 isHasReturn 标记需要更新。受限于篇幅, 在此不展示相关代码。

IfStmt 对于 IfStmt 语句, 若为 if(BoolMultiExpr)-Statement1-else-Statement2 形式, 其执行逻辑如图5所示:

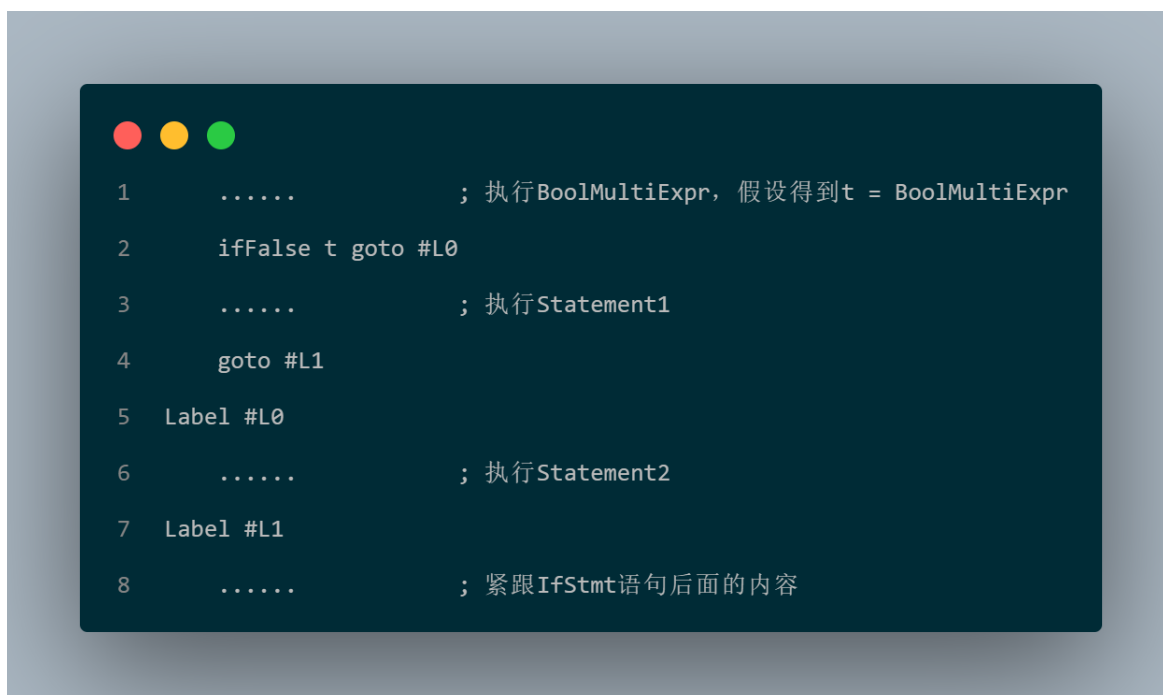


图 5: if-else 的执行逻辑结构

若为 if(BoolMultiExpr)-Statement 形式, 其执行逻辑如图6所示:


```

1      .....      ; 执行BoolMultiExpr, 假设得到t = BoolMultiExpr
2      ifFalse t goto #L0
3      .....      ; 执行Statement
4      goto #L1
5  Label #L0
6      .....      ; 紧跟IfStmt语句后面的内容

```

图 6: if 的执行逻辑结构

因此，我们按照这个顺序进行语义检查和中间代码生成即可：

```

1      .....
2  case NT_IF_STMT: {
3      char* label1 = NULL;
4      buildNewLabel(&label1);
5      // 执行 BoolMultiExpr
6      char* tempName = NULL;
7      ValueType tempValType = VT_BOOL;
8      latedSymPointer = checkBoolMultiExpr(root->ChildPtrList[2],
9          latedSymPointer, &tempName, &tempValType, fp);
10     if (latedSymPointer == NULL)
11         return NULL;
12     // 输出判断
13     fprintf(fp, "    ifFalse %s goto %s\n", tempName, label1);
14     updateScanIndex(root->ChildPtrList[3]->tp);
15     // 执行 statement
16     latedSymPointer = checkStatement(root->ChildPtrList[4], latedSymPointer, fp,
17         isHasReturn);
18     if (latedSymPointer == NULL)
19         return NULL;
20     // 若为if-else形式, 继续执行判断
21     if (root->ChildCurrentIndex > 5) {
22         .....
23     }
24     else
25         fprintf(fp, "Label %s\n", label1);
26     break;
27 }
28 .....

```

WhileStmt 对于 WhileStmt 语句，其执行逻辑如图7所示：

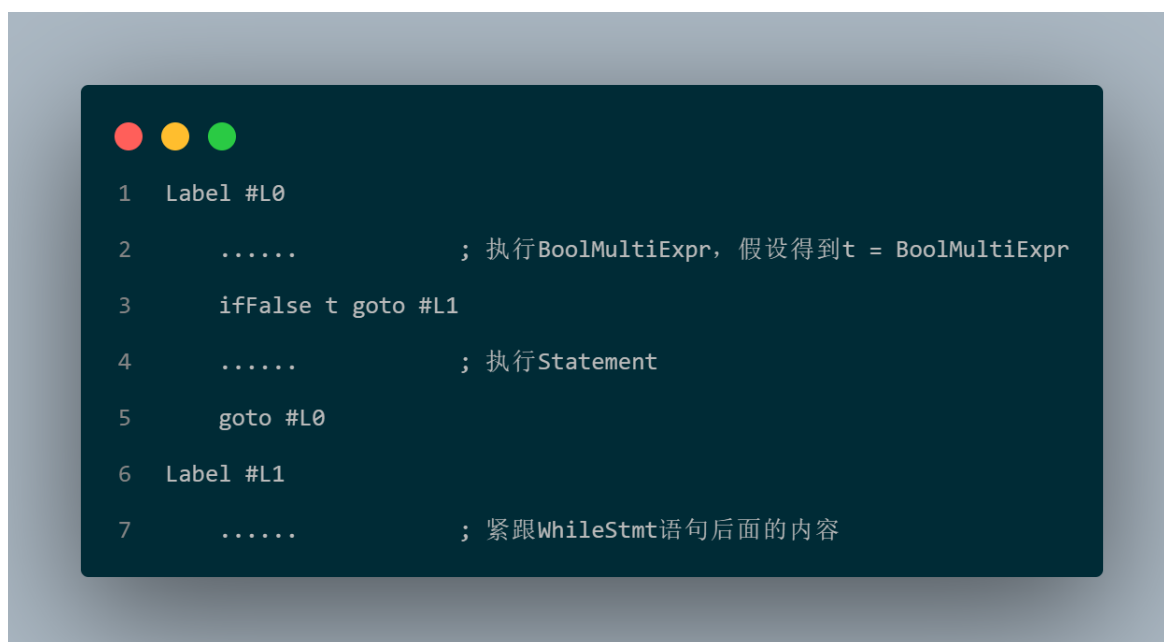


图 7: while 的执行逻辑结构

可以看到，除了具体的中间代码生成逻辑，WhileStmt 的处理方法和 IfStmt 是大同小异的。在此为了节省篇幅，不再展示 WhileStmt 相关代码，具体代码见 code/mySemantics.c。

WriteStmt 和 ReadStmt WriteStmt 和 ReadStmt 是最容易处理的语句，我们只需要获取节点相应内容，然后生成对应的三地址码即可。在此相关代码亦不展示。

5.3.3 expressions

expressions 层次内的语句绝大部分涉及到具体的运算和变量值类型。在此需要详细介绍它们的函数声明格式和参数含义。

以 checkBoolMultiExpr 为例，其声明如下所示：

```
1 Symbol* checkBoolMultiExpr (TreeNode* root ,
2                               Symbol* latedSymPointer ,
3                               char** requiredName ,
4                               ValueType* requiredValType ,
5                               FILE* fp );
```

root、latedSymPointer 和 fp 我们都知道其含义和作用，而参数 requiredName 和 requiredValType 是决定整个程序能否正确执行类型检查和类型转换，并且尽可能在中间代码生成过程中减少临时变量使用的关键。

requiredName 是一个指向字符串地址的指针，含义是要求该函数生成的最终结果名字为 *requiredName。当 *requiredName=NULL 时，说明对最终结果名字没有要求，需要该函数自行处理并提供一个最终结果的名字，将其存储在 *requiredName 上。同样，requiredValType 是一个指向值变量的指针，含义是要求该函数生成的最终结果值类型为 *requiredValType。当 *requiredValType=VT_ANY 时，

说明对最终结果的值类型没有要求，需要该函数将自己计算得到的最终结果的值类型存储在 **required-Name* 上。若 **requiredValType* 与函数自己生成的变量值类型不一致，那么该函数需要执行自动类型转换、得到符合 **requiredValType* 类型的变量输出。换言之，*requiredName* 和 *requiredValType* 充当了一个同时作为继承属性和综合属性的身份，前者旨在减少临时变量的使用，后者旨在对输出值类型作出要求、必要时要自动类型转换。当当前函数无法满足要求时，将 *return NULL*。这时往往意味着程序无法通过类型检查、自动类型转换失败。

BoolMultiExpr 对于 $BoolMultiExpr \rightarrow BoolMidExpr (AND\ BoolMidExpr)^*$ ，当其不包含 **AND** 节点时，说明其对输出的值类型和名字没有要求，全部依赖于子节点 *BoolMidExpr*，故直接传递即可：

```

1 Symbol* checkBoolMultiExpr(.....) {
2     if (root->ChildCurrentIndex==1) {
3         latedSymPointer = checkBoolMidExpr(root->ChildPtrList[0], latedSymPointer,
4             requiredName, requiredValType, fp);
5         if (latedSymPointer==NULL)
6             return NULL;
7     }
8     .....
9 }
```

而当存在 **AND** 节点时，*BoolMultiExpr* 的输出值类型必然为 *VT_BOOL*，此时若 **requiredValType* 不是 *VT_BOOL* 或 *VT_ANY*，那么就无法完成类型转换、报错并 *return NULL*。此外，此时 *BoolMidExpr* 的输出类型也被要求为 *VT_BOOL*。假设 *BoolMultiExpr* 的形式为 *x AND y AND ...*，那么根据短路原则我们有如图8的执行结构：

```

1      .....          ;假设 x=BoolMidExpr
2      ifFalse x goto #L1
3      .....          ;假设 y=BoolMidExpr
4      ifFalse y goto #L1
5      .....
6      t = true          ;t = *requiredName或者临时变量
7      goto #L2
8      Label #L1
9      t = false
10     Label #L2
11     .....          ;BoolMultiExpr之后的语句

```

图 8: AND 语句的执行逻辑结构

对应的实现代码为:

```

1 Symbol* checkBoolMultiExpr(.....) {
2     .....
3     else {
4
5         if (*requiredValType!=VT_BOOL&&*requiredValType!=VT_ANY) {
6             ..... // 出错, return NULL
7         }
8         // 设置要求属性为VT_BOOL
9         *requiredValType = VT_BOOL;
10        char* label1 = NULL;
11        buildNewLabel(&label1);
12        char* label2 = NULL;
13        buildNewLabel(&label2);
14        for (int childIndex = 0; childIndex<root->ChildCurrentIndex; childIndex+=2) {
15            // 通过tempName套取checkBoolMidExpr的输出变量名
16            char* tempName = NULL;
17            // 解析BoolMidExpr
18            latedSymPointer = checkBoolMidExpr(root->ChildPtrList[childIndex],
19                latedSymPointer, &tempName, requiredValType, fp);
20            if (latedSymPointer==NULL)
21                return NULL;
22            fprintf(fp, "    ifFalse %s goto %s\n", tempName, label1);
23        }
24    }
25 }

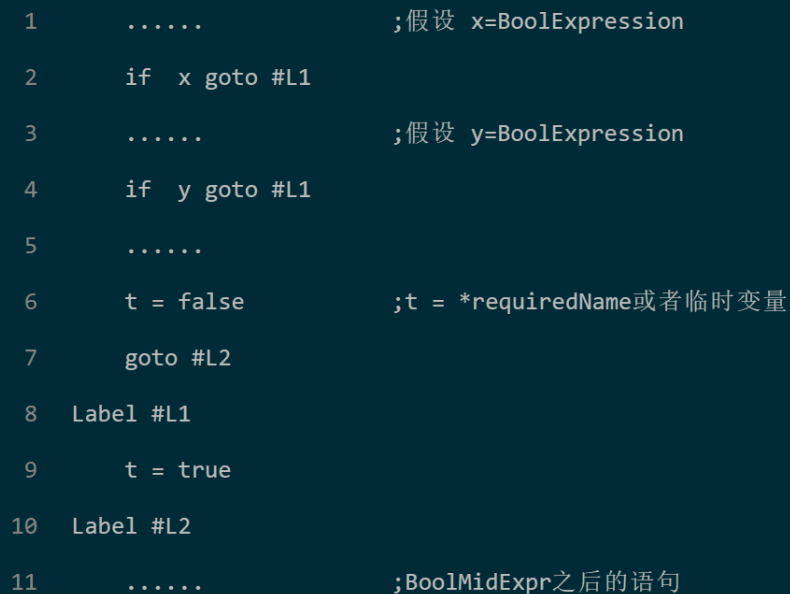
```

```

23 // 名字没要求, 获取一个临时名字
24 if (*requiredName==NULL)
25     buildNewTempValue(requiredName);
26 fprintf(fp, "    %s = true\n", *requiredName);
27 fprintf(fp, "    goto %s\n", label2);
28 fprintf(fp, "Label %s\n", label1);
29 fprintf(fp, "    %s = false\n", *requiredName);
30 fprintf(fp, "Label %s\n", label2);
31 }
32 return latedSymPointer;
33 }

```

BoolMidExpr checkBoolMidExpr 的代码与 checkBoolMultiExpr 极其相似, 区别在于当存在 **OR** 节点时, 其短路运行机制有所不同:



```

1      .....      ;假设 x=BoolExpression
2      if x goto #L1
3      .....      ;假设 y=BoolExpression
4      if y goto #L1
5      .....
6      t = false      ;t = *requiredName或者临时变量
7      goto #L2
8      Label #L1
9          t = true
10     Label #L2
11     .....      ;BoolMidExpr之后的语句

```

图 9: OR 语句的执行逻辑结构

BoolExpression 当 BoolExpression 仅有一个 Expression 节点时, 其最终结果的名字和最终结果的值类型依赖于 Expression, 故和在 checkBoolMultiExpr 中的处理方式相同、直接传递。当 BoolExpression 有两个 Expression 节点时, 说明 checkBoolExpression 的最终结果值类型必须为 VT_BOOL。此外, 除了 == 和 !=, 对于其他比较符, 两个 Expression 节点的最终结果值类型必须为 VT_INT 或 VT_REAL (布尔值之间不存在谁比谁大的说法)。当两个 Expression 节点值类型不一致时, 我们还需要进行自动类型转换才能进行比较。假设经过自动类型转换后, 两个 Expression 节点的最终结果分别为 x 和 y, 比较运算符为 op, 那么我们有如图10的执行结构。



图 10: BoolExpression 语句的执行逻辑结构

Expression 和 MultiplyExpr 两个语句的结构几乎相同，故放在一起讨论。当两个语句只有一个节点时，操作方式与上述函数一样。当这两个语句包含若干个子节点、通过二元运算符连接时，这些子节点的最终输出值类型必须为 VT_INT 或 VT_REAL（难道布尔值还能加减乘除？）。此外，不同于 BoolMultiExpression 和 BoolMidExprssion，数字的连续加减乘除不存在短路机制。因此，我们只能从左往右计算前两个子节点输出的结果、存储到临时变量，然后使用临时变量和下一个子节点输出结果进行计算、存储到下一个临时变量，循环往复。在最后一步运算中，我们不使用临时变量、而是使用 *requiredName 提供的地址存储最终输出结果，并根据 *requiredValType 对结果进行自动类型转换。受限于篇幅，在此我仅提供函数的实现思路，不展示相关代码。

PrimaryExpr PrimaryExpr 是 expressions 乃至整个语法层次中最复杂、需要考虑情况最多的语句。PrimaryExpr 可能包含否定单目运算符或者取负单目运算符。对于前者，PrimaryExpr 的最终输出结果必须为 VT_BOOL。当否定单目运算符后面跟随的值类型为 VT_INT 或 VT_REAL，我们可以不使用类型转换，而是使用如图11所示的结构输出最终结果。

```

1      .....      ;假设得到类型为VT_INT或VT_REAL的x
2      if x==0 goto #L1
3      t = true      ;t = *requiredName或者临时变量
4      goto #L2
5  Label #L1
6      t = false
7  Label #L2
8      .....      ;PrimaryExpr之后的语句

```

图 11: 否定运算符结合数字的执行逻辑结构

对于取负值的运算符，其后面跟随的值类型必须为 VT_INT 或 VT_REAL，而 PrimaryExpr 需要根据 *requiredValType 对取了负数的结果进行进一步自动类型转换。

此外，PrimaryExpr 还包括了函数的调用过程。在确认函数已经声明后，调用函数的执行逻辑结构如图12所示。

```

1  ; 假设调用语句为 a:=func(BoolMultiExpr1, BoolMultiExpr2,...);
2      .....      ;假设得到x=BoolMultiExpr1
3      param x
4      .....      ;假设得到y=BoolMultiExpr1
5      param y
6      .....
7      #V0 = call #F0, n ;#V0是a的三地址命名，#F0是func的三地址命名
8                          ;n为参数个数
9                          ;当#V0和#F0类型不匹配时，需要先做类型转换
10     .....      ;PrimaryExpr之后的语句

```

图 12: 函数调用的执行逻辑结构

5.3.4 自动类型转换

实验过程和实现代码说明的最后一部分是有关自动类型转换的内容。我实现了一个函数 `outputTypeTransformCode`，用于生成类型转换的三地址码。对于 `VT_INT` 和 `VT_REAL`，两者之间的转换十分简单，正如图13所示，只需要使用单目运算符即可。



```
1 ; 假设x为VT_INT, y为VT_REAL
2     #TV0 = (real)x      ; int->real
3     #TV1 = (int)y       ; real->int
```

图 13: `VT_INT` 和 `VT_REAL` 之间的转换

然而对于从 `VT_INT` (`VT_REAL`) 到 `VT_BOOL` 的转换，由于类型完全不同、无法直接通过单目运算符转换。我参考了 C 语言的逻辑，设计了如图14的转换结构。此时的 `#VT0` 就相当于 `#V0` 转换后的值。



```
1 ; 假设#V0为VT_INT或VT_REAL
2     if #V0==0 goto #L0
3     #TV0 = true
4     goto #L1
5 Label #L0
6     #TV0 = false
7 Label #L1
8     .....
```

图 14: `VT_INT` (`VT_REAL`) 到 `VT_BOOL` 的转换

同样为了节省篇幅，在此不展示代码。详细实现过程见 `code/mySemantics.c`。

6 实验结果

在完成上述代码后，我们将对实现的程序展开测试。我们首先对一些正确或错误的程序样本进行语义分析，检查我们的语义分析器的错误检测能力。然后，我们构建一些复杂的程序样本，让我们实现的编译器将其翻译成三地址中间代码，通过直接检查三地址中间代码来检查中间代码生成器是否存在错误。

6.1 语义分析

6.1.1 符号表的构建与维护

语义分析中最重要的部分是符号表的构建与维护。我们尝试使用程序对一份没有 bug 的 TINY+ 代码进行语义分析，将符号表打印出来进行检查。测试的代码 good1.tinyplus 代码如下所示：

```
1  /** good1.tinyplus **/  
2  /** This is a TINY+ program without bug. **/  
3  INT f2(INT x, INT y )  
4  BEGIN  
5      INT a;  
6      a;  
7      a := 10;  
8      WHILE (a >= x)  
9          a := a % 2;  
10     RETURN a+y;  
11 END  
12 INT MAIN f1 ()  
13 BEGIN  
14     BOOL flag:=TRUE;  
15     INT x:=5, y;  
16     REAL c:=4.521;  
17     IF (5>4)  
18         x := (x+3)*4;  
19     ELSE  
20     BEGIN  
21         READ(y, "input y");  
22         c := y - 3;  
23     END  
24     INT z:= f2(x,y) + f2(y,x);  
25     WRITE (z, "output z");  
26     RETURN 0;  
27 END
```

分析完成后，程序打印出来的符号表如图15所示。符号项的四个值分别是符号类型、符号值类型、token 和三地址码命名。父符号表和子符号表通过缩进来进行区分。

```

1  ST_HEAD VT_ANY NULL NULL
2  ST_FUNC VT_INT (ID, f2) #F0
3      ST_HEAD VT_ANY NULL NULL
4      ST_VAR VT_INT (ID, x) #V0
5      ST_VAR VT_INT (ID, y) #V1
6      ST_VAR VT_INT (ID, a) #V2
7  ST_FUNC VT_INT (ID, f1) #F1
8      ST_HEAD VT_ANY NULL NULL
9      ST_VAR VT_BOOL (ID, flag) #V3
10     ST_VAR VT_INT (ID, x) #V4
11     ST_VAR VT_INT (ID, y) #V5
12     ST_VAR VT_REAL (ID, c) #V6
13     ST_TABLE VT_ANY NULL NULL
14         ST_HEAD VT_ANY NULL NULL
15     ST_VAR VT_INT (ID, z) #V7

```

图 15: good1.tinyplus 符号表

经过检查，可以发现符号表的结构完整，各个变量名和函数名的位置、类型、值类型和三地址码命名正确，不同作用域中声明的变量和函数分布在不同的符号表。后续我们又测试了其他程序，均输出正确，说明我们的程序能够很好地构建和维护符号表。

6.1.2 声明检查

声明检查包括两部分，一部分是在声明变量/函数时，检查同一作用域是否已存在同名变量/函数；另一部分是在声明函数时，函数头内的参数是否有重名的情况。

针对第一种情况，我设计了 error2_1.tinyplus:

```

1  /** error2_1.tinyplus */
2  /** This is a TINY+ program with bug(s). */
3  INT MAIN f1 ()
4  BEGIN
5      INT x:=5;
6      BOOL y:=TRUE;
7      READ(y, "symbol");
8      BEGIN
9          BOOL y:=FALSE;

```

```

10     END
11     REAL x:=6.4;
12     RETURN 0;
13 END

```

程序分析结果如图16所示。可以看到，程序没有认为第 9 行中在嵌套作用域声明的 `y` 是错误结果，反而发现了第 11 行的 `x` 是一个重复声明的变量。这与我们的预期结果一致。

```

E:\code\Windows\VS2019\CPPSample\Debug\CPPSample.exe
-----
| TINY+ v1.0 :: 18308013 ChenJiahao |
-----
[INFO] Welcome to use TINY+ compiler!
[INFO] Please input your source file full path:
[INPUT] error2_1.tinyplus
[INFO] Lexical analysis:

[INFO] Result has been generated in output_token.txt.
[INFO] Lexical analysis finished.
[INFO] Parser analysis:

[INFO] Result has been generated in output_tree.txt.
[INFO] Parser analysis finished.
[INFO] Semantic analysis:
[ERROR] 11:10: A variable with the same name already exists in this scope.
[INFO] Semantic analysis interrupted.
请按任意键继续. . .

```

图 16: error2_1.tinyplus 分析结果

至于第二种情况，error2_2.tinyplus 代码如下所示：

```

1  /** error2_2.tinyplus **/
2  /** This is a TINY+ program with bug(s). **/
3  INT MAIN f1 (INT x,BOOL z,REAL x)
4  BEGIN
5      BOOL y:=TRUE;
6      RETURN 0;
7  END

```

程序分析结果如图17所示，可见程序发现了第三行的函数声明中存在同名参数的情况。

```
E:\code\Windows\VS2019\CPPSample\Debug\CPPSample.exe
| TINY+ v1.0 :: 18308013 ChenJiahao |
-----
[INFO] Welcome to use TINY+ compiler!
[INFO] Please input your source file full path:
[INPUT] error2_2.tinyplus
[INFO] Lexical analysis:

[INFO] Result has been generated in output_token.txt.
[INFO] Lexical analysis finished.
[INFO] Parser analysis:

[INFO] Result has been generated in output_tree.txt.
[INFO] Parser analysis finished.
[INFO] Semantic analysis:
[ERROR] 3:31: The parameter with the same name exists.
[INFO] Semantic analysis interrupted.
请按任意键继续. . .
```

图 17: error2_2.tinyplus 分析结果

通过上述实验结果，我们可以认为我们的程序实现了声明检查的能力。

6.1.3 使用检查

使用检查主要用于检查当前使用的变量/函数是否已经在当前作用域或包含当前作用域的作用域中声明。测试代码 error3.tinyplus 如下所示：

```
1  /** error3.tinyplus **/
2  /** This is a TINY+ program with bug(s). **/
3  INT MAIN f1 ()
4  BEGIN
5      INT x:=5;
6      REAL y:=4.3;
7      BEGIN
8          INT z:=x, y;
9          y := x + a;
10     END
11     RETURN 0;
12 END
```

程序分析结果如图18所示。可见，对于第 8、9 行中 x 的使用，语义分析器并没有报错。而对第 9 行中 a 的使用，语义分析器给出了变量未声明的报错。这和预期结果一致。

```
E:\code\Windows\VS2019\CPPSample\Debug\CPPSample.exe

| TINY+ v1.0 :: 18308013 ChenJiahao |
|-----|
[INFO] Welcome to use TINY+ compiler!
[INFO] Please input your source file full path:
[INPUT] error3.tinyplus
[INFO] Lexical analysis:

[INFO] Result has been generated in output_token.txt.
[INFO] Lexical analysis finished.
[INFO] Parser analysis:

[INFO] Result has been generated in output_tree.txt.
[INFO] Parser analysis finished.
[INFO] Semantic analysis:
[ERROR] 9:18: The variable is not defined.
[INFO] Semantic analysis interrupted.
请按任意键继续. . .
```

图 18: error3.tinyplus 分析结果

通过上述实验结果，我们可以认为我们的程序实现了使用检查的能力。

6.1.4 类型检查

类型检查的目的在于检查值类型是否与运算匹配。由于我在本次实验中设计了十分通用的自动类型转换功能，故类型检查不通过的情况仅存在尝试将 BOOL 变量转换为其他类型。error4.tinyplus 代码设计如下：

```
1  /** error4.tinyplus **/
2  /** This is a TINY+ program with bug(s). **/
3  INT MAIN f1 ()
4  BEGIN
5      INT x:=4;
6      BOOL y:=!x;
7      IF (y > 0)
8      BEGIN
9          x := x-1;
10     END
11 END
```

可见第 6 行中出现了一个 INT 值自动转换成 BOOL 值的情况，而第 7 行尝试将 BOOL 值与常数进行大小比较。程序分析结果如图19所示，语义分析器通过了第 6 行的操作，而对第 7 行的操作报错。这与预期结果相符。

```
E:\code\Windows\VS2019\CPPSample\Debug\CPPSample.exe
| TINY+ v1.0 :: 18308013 ChenJiahao |
-----
[INFO] Welcome to use TINY+ compiler!
[INFO] Please input your source file full path:
[INPUT] error4.tinyplus
[INFO] Lexical analysis:

[INFO] Result has been generated in output_token.txt.
[INFO] Lexical analysis finished.
[INFO] Parser analysis:

[INFO] Result has been generated in output_tree.txt.
[INFO] Parser analysis finished.
[INFO] Semantic analysis:
[ERROR] 7:11: The types can't be compared.
[INFO] Semantic analysis interrupted.
请按任意键继续. . .
```

图 19: error4.tinyplus 分析结果

6.1.5 自动类型转换

既然我们设计了通用的自然类型转换,那么我们自然要对这个功能进行测试。测试代码 good5.tinyplus 如下所示:

```
1  /** good5.tinyplus */
2  /** This is a TINY+ program without bug. */
3  INT MAIN f1 ()
4  BEGIN
5      INT x:=4;
6      REAL y:=-x*3;
7      INT z:=x-y;
8      IF (!x)
9          RETURN 0;
10     ELSE
11         RETURN 1;
12 END
```

程序输出的符号表和三地址代码分别如图20和图21所示。可见在三地址码中,无论是对 y 和 z 的赋值、还是将 x 转换成 BOOL 类型,程序都完成了自动类型转换。需要注意的是,在运算过程中,程序会使用优先使用精度更高的浮点数进行计算。此外,由于程序设计的局限性,“IF (!x)”的转换三地址码不是最优的,完全可以使用更少的临时变量、标签和跳转语句实现。

```

1  ST_HEAD VT_ANY NULL NULL
2  ST_FUNC VT_INT (ID, f1) #F0
3      ST_HEAD VT_ANY NULL NULL
4      ST_VAR VT_INT (ID, x) #V0
5      ST_VAR VT_REAL (ID, y) #V1
6      ST_VAR VT_INT (ID, z) #V2

```

图 20: good5.tinyplus 符号表

```

1  Label #F0
2  Label MAIN
3      #V0 = 4
4      #TV0 = minus #V0
5      #TV1 = #TV0 * 3
6      #V1 = (real)#TV1
7      #TV2 = (real)#V0
8      #TV3 = #TV2 - #V1
9      #V2 = (int)#TV3
10     ifFalse #V0==0 goto #L1
11     #TV4 = true
12     goto #L2
13 Label #L1
14     #TV4 = false
15 Label #L2
16     ifFalse #TV4 goto #L0
17     return 0
18     goto #L3
19 Label #L0
20     return 1
21 Label #L3

```

图 21: good5.tinyplus 三地址代码

6.1.6 参数检查

参数检查包括了在调用函数时对传参数量的检查和传参类型的检查。一份传参数量出错的代码如下所示：

```

1  /** error6_1.tinyplus **/
2  /** This is a TINY+ program with bug(s). **/
3  INT f2(INT x, INT y )
4  BEGIN
5      INT a;
6      a;
7      a := 10;
8      WHILE (a >= x)
9          a := a % 2;
10     RETURN a+y;
11 END
12 INT MAIN f1 ()
13 BEGIN
14     BOOL flag:=TRUE;
15     INT x:=5, y;
16     REAL c:=4.521;
17     IF (5>4)
18         x := (x+3)*4;

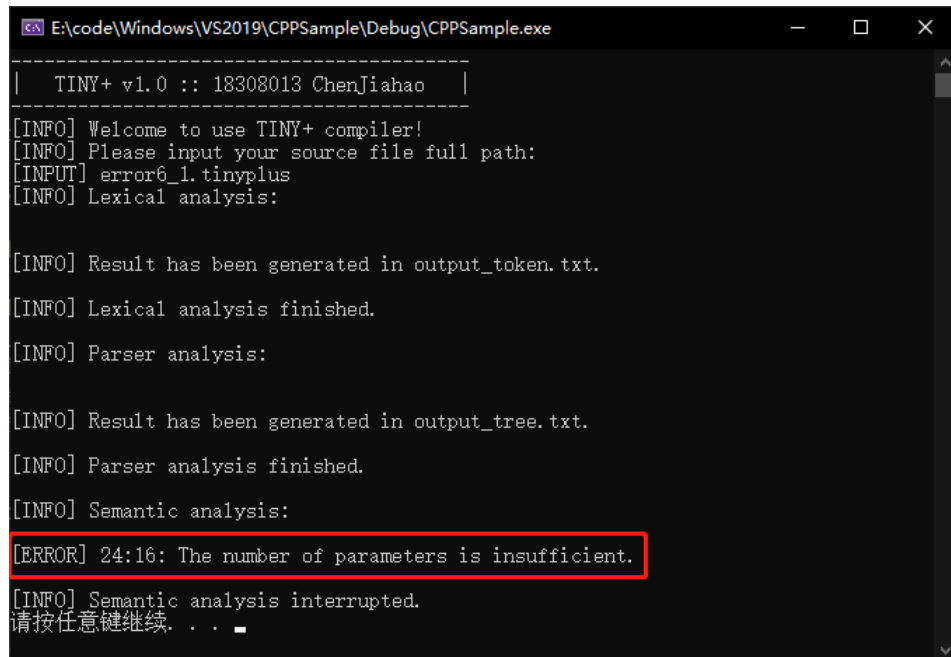
```

```

19 ELSE
20 BEGIN
21     READ(y, "input y");
22     c := y - 3;
23 END
24 INT z:= f2(x) + f2(y,x);
25 WRITE (z, "output z");
26 END

```

在 error6_1.tinyplus 的第 24 行中，我们错误的只传递给 f2() 一个参数。程序发现了这个问题，相关报错信息如图22所示。



```

E:\code\Windows\VS2019\CPPSample\Debug\CPPSample.exe
| TINY+ v1.0 :: 18308013 ChenJiahao |
-----
[INFO] Welcome to use TINY+ compiler!
[INFO] Please input your source file full path:
[INPUT] error6_1.tinyplus
[INFO] Lexical analysis:

[INFO] Result has been generated in output_token.txt.
[INFO] Lexical analysis finished.
[INFO] Parser analysis:

[INFO] Result has been generated in output_tree.txt.
[INFO] Parser analysis finished.
[INFO] Semantic analysis:
[ERROR] 24:16: The number of parameters is insufficient.
[INFO] Semantic analysis interrupted.
请按任意键继续. . .

```

图 22: error6_1.tinyplus 分析结果

另一份传参类型出错的代码如下所示：

```

1 /** error6_2.tinyplus */
2 /** This is a TINY+ program with bug(s). */
3 INT f2(INT x, INT y )
4 BEGIN
5     INT a;
6     a;
7     a := 10;
8     WHILE (a >= x)
9         a := a % 2;
10    RETURN a+y;
11 END
12 INT MAIN f1 ()
13 BEGIN
14     BOOL flag:=TRUE;
15     INT x:=5, y;
16     REAL c:=4.521;

```



```

17 IF (5>4)
18     x := (x+3)*4;
19 ELSE
20 BEGIN
21     READ(y, "input y");
22     c := y - 3;
23 END
24 INT z:= f2(x,y) + f2(flag, x);
25 WRITE (z, "output z");
26 END

```

在 error6_2.tinyplus 的第 24 行中，我们想要把一个 BOOL 变量 flag 作为 INT 参数传递给 f2()。这种操作当然是错误的，程序的报错信息如图23所示。

```

E:\code\Windows\VS2019\CPPSample\Debug\CPPSample.exe
-----
|  TINY+ v1.0 :: 18308013 ChenJiahao  |
-----
[INFO] Welcome to use TINY+ compiler!
[INFO] Please input your source file full path:
[INPUT] error6_2/error6_2.tinyplus
[INFO] Lexical analysis:

[INFO] Result has been generated in output_token.txt.
[INFO] Lexical analysis finished.
[INFO] Parser analysis:

[INFO] Result has been generated in output_tree.txt.
[INFO] Parser analysis finished.
[INFO] Semantic analysis:
[ERROR] 24:26: The type can't be changed to INT.
[INFO] Semantic analysis interrupted.
请按任意键继续. . .

```

图 23: error6_2.tinyplus 分析结果

6.1.7 返回检查

语义检查的最后一部分是对函数进行返回检查，一个函数必须带有返回语句，并且返回值必须与函数数值类型相匹配。

错误代码 error7_1.tinyplus 如下所示，可见其函数没有使用返回语句：

```

1 /** error7_1.tinyplus */
2 /** This is a TINY+ program with bug(s). */
3 INT MAIN f1 ()
4 BEGIN
5     INT x:=5;
6     REAL y:=4.3;
7 END

```

程序报错信息如图24所示：

```
E:\code\Windows\VS2019\CPPSample\Debug\CPPSample.exe
| TINY+ v1.0 :: 18308013 ChenJiahao |
-----
[INFO] Welcome to use TINY+ compiler!
[INFO] Please input your source file full path:
[INPUT] error7_1.tinyplus
[INFO] Lexical analysis:

[INFO] Result has been generated in output_token.txt.
[INFO] Lexical analysis finished.
[INFO] Parser analysis:

[INFO] Result has been generated in output_tree.txt.
[INFO] Parser analysis finished.
[INFO] Semantic analysis:
[ERROR] 6:1: The function doesn't have return statement.
[INFO] Semantic analysis interrupted.
请按任意键继续. . .
```

图 24: error7_1.tinyplus 分析结果

错误代码 error7_2.tinyplus 如下所示，可见返回语句试图使用一个 BOOL 值作为 INT 类型返回：

```
1 /** error7_2.tinyplus **/
2 /** This is a TINY+ program with bug(s). **/
3 INT MAIN f1 ()
4 BEGIN
5     INT x:=5;
6     REAL y:=4.3;
7     RETURN FALSE;
8 END
```

程序报错信息如图25所示：

```
E:\code\Windows\VS2019\CPPSample\Debug\CPPSample.exe
| TINY+ v1.0 :: 18308013 ChenJiahao |
-----
[INFO] Welcome to use TINY+ compiler!
[INFO] Please input your source file full path:
[INPUT] error7_2.tinyplus
[INFO] Lexical analysis:

[INFO] Result has been generated in output_token.txt.
[INFO] Lexical analysis finished.
[INFO] Parser analysis:

[INFO] Result has been generated in output_tree.txt.
[INFO] Parser analysis finished.
[INFO] Semantic analysis:
[ERROR] 6:12: The type can't be changed.
[INFO] Semantic analysis interrupted.
请按任意键继续. . .
```

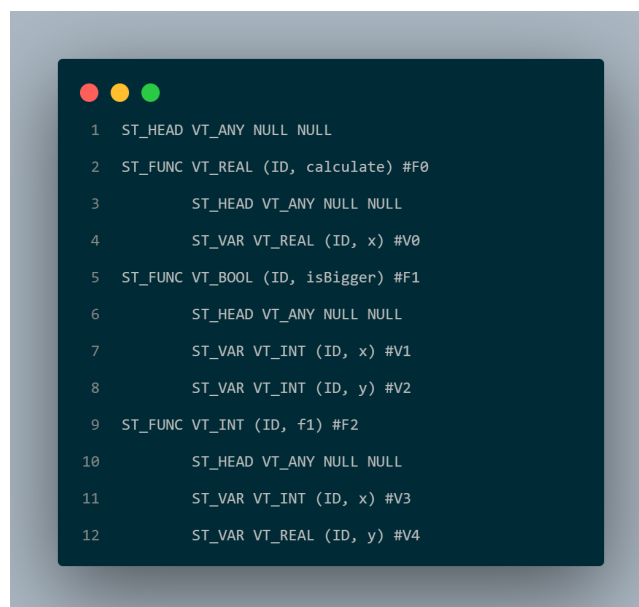
图 25: error7_2.tinyplus 分析结果

6.2 中间代码生成

最后一部分，我们将检查中间代码生成器生成的三地址码质量情况。测试代码如下所示：

```
1  /** good8.tinyplus **/  
2  /** This is a TINY+ program without bug. **/  
3  REAL calculate (REAL x)  
4  BEGIN  
5      IF (!x)  
6          RETURN (x+1)*(-2*x);  
7      RETURN x*x;  
8  END  
9  
10 BOOL isBigger (INT x, INT y)  
11 BEGIN  
12     IF (calculate(x)>calculate(y))  
13         RETURN TRUE;  
14     ELSE  
15         RETURN FALSE;  
16 END  
17  
18 INT MAIN f1 ()  
19 BEGIN  
20     INT x:=4;  
21     REAL y:=-x*3;  
22     WHILE (isBigger(x,y))  
23         x:=x-1;  
24     RETURN x;  
25 END
```

程序生成的符号表如图26所示。三地址码如图27和图28所示。



1	ST_HEAD VT_ANY NULL NULL
2	ST_FUNC VT_REAL (ID, calculate) #F0
3	ST_HEAD VT_ANY NULL NULL
4	ST_VAR VT_REAL (ID, x) #V0
5	ST_FUNC VT_BOOL (ID, isBigger) #F1
6	ST_HEAD VT_ANY NULL NULL
7	ST_VAR VT_INT (ID, x) #V1
8	ST_VAR VT_INT (ID, y) #V2
9	ST_FUNC VT_INT (ID, f1) #F2
10	ST_HEAD VT_ANY NULL NULL
11	ST_VAR VT_INT (ID, x) #V3
12	ST_VAR VT_REAL (ID, y) #V4

图 26: good8.tinyplus 符号表

```

1  Label #F0
2      get #V0
3      ifFalse #V0==0 goto #L1
4      #TV0 = true
5      goto #L2
6  Label #L1
7      #TV0 = false
8  Label #L2
9      ifFalse #TV0 goto #L0
10     #TV2 = (real)1
11     #TV1 = #V0 + #TV2
12     #TV3 = minus 2
13     #TV5 = (real)#TV3
14     #TV4 = #TV5 * #V0
15     #TV6 = #TV1 * #TV4
16     return #TV6
17 Label #L0
18     #TV7 = #V0 * #V0
19     return #TV7
20 Label #F1
21     get #V2
22     get #V1
23     #TV8 = (real)#V1
24     param #TV8
25     #TV9 = call #F0, 1
26     #TV10 = (real)#V2
27     param #TV10
28     #TV11 = call #F0, 1

```

图 27: good8.tinyplus 三地址码 (1)

```

1      if #TV9 > #TV11 goto #L4
2      #TV12 = false
3      goto #L5
4  Label #L4
5      #TV12 = true
6  Label #L5
7      ifFalse #TV12 goto #L3
8      return True
9      goto #L6
10 Label #L3
11     return False
12 Label #L6
13 Label #F2
14 Label MAIN
15     #V3 = 4
16     #TV13 = minus #V3
17     #TV14 = #TV13 * 3
18     #V4 = (real)#TV14
19 Label #L7
20     param #V3
21     #TV15 = (int)#V4
22     param #TV15
23     #TV16 = call #F1, 2
24     ifFalse #TV16 goto #L8
25     #V3 = #V3 - 1
26     goto #L7
27 Label #L8
28     return #V3

```

图 28: good8.tinyplus 三地址码 (2)

经过检查，程序生成的三地址码是准确无误的。然而，在部分语句的翻译上还存在着改进空间。

7 实验总结

本次实验是编译原理课程第 4 次实验，要求我们实现语义分析器和中间代码生成器。我在上一次实验的基础上，通过遍历一次语法树同时实现语义分析和中间代码生成。我最终实现的语义分析器可以检查不同类型的语义错误，而中间代码生成器可以生成高质量的三地址中间代码，取得了令人满意的实验结果。

实话实说，在完成这次实验之前，我对相关内容的了解是匮乏的、不足的，尤其是语法制导翻译的内容处于一知半解的状态。这给我这次实验带来了极大的困难。不过在完成实验的过程中我反复翻阅课本和课件、吸收相关知识，并查阅了大量网上资料，最终还是对相关知识点有了一定的认识。本次实验中我没有完全按照课件和教材的方法来实现语义分析和中间代码生成，而是结合了课本上的部分知识（比如布尔值的转换、while 语句的生成方案等）以及个人的一些想法来完成实验。其中最让我感到满意的是 requiredName 和 requiredValType 两个参数变量的应用。通过这两个参数在函数之间的传递，我们可以实现输出值类型的约束和减少临时变量的使用。这既有利于帮助我们在语义分析上进行类型检查和自动类型转换，也帮助我们生成更高质量的三地址中间代码。

由于时间有限，我没有对本次实验进行更多的尝试与拓展。事实上，在本次实验中我生成的三地址中间代码已经达到了可执行的标准，尤其是设计了一个良好的函数调用和返回机制，基于栈的做法与主流编程语言一致。略有不足的是，对 TINY+ 的拓展还没有达到我当初“类 C”的想法，此外三地址中间代码生成还有进一步的优化空间。希望这次实验能给我未来的学习带来一定帮助。