语义分析及中间代码生成实验报告

王若琪　18340166

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. 算法描述

4.1 语义分析

语义分析是在语法分析的基础上完成的，语法分析完成前一步的语法树分析的构建，之后语义分析器接着调用响应的产生式配备的语义动作或子程序，完成属性文法所要求的的语义处理的意义。

我设计的语义分析的输入是一个语法树，输出是符号表和报错信息。符号表能够统计代码中所有变量的声明及出现位置等信息，报错信息能够检测出一定的语义错误，例如：1. 使用一个没有声明过的标识符，或者重复声明该标识符。2. 二元操作符的两个操作数类型不匹配。3. 赋值语句两边的类型不匹配。

4.1.1 首先构建符号表

符号表有三列，分别是符号名称、符号位置、以及符号出现的行号。这样在之后分析的过程中，就能够通过查阅符号表，来判断语义的正确性。符号表基于哈希表实现，定义元素的数据结构如下：

```
// 哈希表元素数据结构
typedef struct hashNode
{
    char* name; // 符号名
    int type; // 类型
    int memloc; // 变量的位置
    PosList rows; // 行号数据结构
    struct hashNode* next;
} *HT;
// 哈希表
static HT hashTable[HashSize];
```

插入符号表时，可以分为两种情况，即这个符号已经在符号表内或者这个符号不在符号表内，如果这个符号不在符号表内，那么就添加该符号进符号表，如果这个符号已经在符号表内了，就只添加它本次出现的位置即可。此部分算法实现代码如下：

```
// 符号表插入函数
void insert_table(char* name, int Row, int loc, int exp_type)
```

```
{
    int h = hash(name);
    HT li = hashTable[h];
    while ((li != NULL) && (strcmp(name, li->name) != 0))
        li = li->next;
    if (li == NULL) // 如果这个符号不在符号表内，就添加该符号
    {
        li = (HT)malloc(sizeof(struct hashNode));
        li->name = name;
        li->rows = (PosList)malloc(sizeof(struct PosListRec));
        li->rows->Row = Row;
        li->memloc = loc;
        li->type = exp_type;
        li->rows->next = NULL;
        li->next = hashTable[h];
        hashTable[h] = li;
    }
    else // 如果这个符号在符号表内，只加入行号即可
    {
        PosList t = li->rows;
        while (t->next != NULL) t = t->next;
        t->next = (PosList)malloc(sizeof(struct PosListRec));
        t->next->Row = Row;
        t->next->next = NULL;
    }
}
```

4.1.2 语义错误检查

构建完符号表结构之后，就可以开始语义分析了，语义分析主要是递归检查语法树的节点信息，如果有错误就报错输出。能够检查出的语义错误类型以及实现方法如下所示：

- 在构建符号表的过程中，可以检查出有关变量声明和使用的错误：
  - 当给一个变量赋值时，如果他还没有被声明过，那么它就不在符号表内，于是报错，如果在符号表内，那么就正常插入：

```
        case AssignK:
            lookup_temp = lookup_table_type(t->attr.name);
            if (lookup_temp == -1) // 如果他还不在符号表内，报错
            {
                printf("Symb error! %s has not been declared!
line:%d\n", p_temp->attr.name, p_temp->Row);
            }
            else  // 如果他在符号表内，正常插入
            {
                insert_table(t->attr.name, t->Row, 0, lookup_temp);
                t->kind.stmt = lookup_temp;
            }
            break;
```

  - 当声明一个变量时，如果它还没被声明过，那么他就不在符号表内，正常插入，如果它已经在符号表内了，那就说明重复声明，报错。以声明 int 型变量为例：

```
        case IntDeclareK:
            lookup_temp = lookup_table_type(t->attr.name);

            if (lookup_temp == -1) // 如果他还不在符号表内，正常插入
            {
                insert_table(t->attr.name, t->Row, location++,t->attr.name);

            }
            else // 如果他在符号表内，报错
            {
                printf("Symb error! %s has already been declared! line:%d\n", p_temp->attr.name, p_temp->Row);
            }
            break;
```

- 构建好符号表后，再检查一下类型错误。
  - 操作符两边的符号类型要相同，如果不相同就报错 “Different types in this operation!”。除此之外，如果是比较操作，则式子的返回值设为 Boolean 类型。此过程的实现代码如下：

```
        case OpK:
            if ((t->child[0]->attr.token != t->child[1]->attr.token))
            {
                typeError(t, "Different types in this operation!");
            }
            if ((t->attr.token == EQUAL) || (t->attr.token == LT) || (t->attr.token == GT) || (t->attr.token == UNEQUAL))
                t->kind.stmt = Boolean;
            break;
```

  - 如果是赋值语句，那么检查两边的类型是否相同，不相同则报错：

```
        case AssignK: // 如果赋值语句，则检查两边类型是否相同
            if (t->child[0]->kind.stmt != t->kind.stmt)
                typeError(t, "Assignment is not the same type!");
            break;
```

  - 如果是 if，则检查 if 语句的值是否为 boolean 类型，如果不是，则报错：

```
        case IfK: // 如果是if，则检查是否为bool类型
            if (t->child[0]->kind.stmt != Boolean)
                typeError(t->child[0], "If is not Boolean!");
            break;
```

  - 如果是 while 语句，同理检查值是否为 boolean 类型或者 int 类型，如果不是，则报错：

```
        case WhileK: // 如果是while，则检查是否为bool类型或整数类型
            if (t->child[0]->kind.stmt != Boolean && t->child[0]->kind.stmt != IntTypeK)
                typeError(t->child[0], "While is not Boolean and not Integer!");
            break;
```

到这里语义分析的算法流程就结束了，实验测试结果请见第五部分。

4.2 中间代码生成

先设定程序计数器、内存指针、全局指针、内存 offset 等：

```
// 程序计数器
#define  pc 12
// 内存指针
#define  mp 6
// global pointer
#define gp 5
// accumulator
#define  ac 0
#define  ac1 1
// 临时的 memory offset
static int tempOffset = 0;
```

将中间代码的指令分为如下两种，并设计中间代码表达形式：

- 只有寄存器的指令，形式为“操作类型，目标寄存器，源寄存器1，源寄存器2”，用c语言实现代码如下：

```
// 只有寄存器的指令 op是操作类型，*r 是目标寄存器， s是第一个源寄存器， t是第二个源寄存器
void genRO(char* op, int r, int s, int t)
{
    printf("%3d:  %5s  %d,%d,%d ", genLoc++, op, r, s, t);
    printf("\n");
    if (highgenLoc < genLoc) highgenLoc = genLoc;
}
```

- 从寄存器到内存的指令，r 是目标寄存器，*d 是 offset*， s 是基准寄存器，表示程度语句为“ 操作类型，目标寄存器， （offset）基准寄存器”，用c语言代码实现如下：

```
// 从寄存器到内存的指令，r 是目标寄存器，*d 是 offset，* s 基准寄存器
void genRM(char* op, int r, int d, int s)
{
    printf("%3d:  %5s  %d,%d(%d) ", genLoc++, op, r, d, s);
    printf("\n");
    if (highgenLoc < genLoc)  highgenLoc = genLoc;
}
```

除此之外，还需要直到分支指令中，需要跳过多少代码：

```
// 关于跳过多少代码
int genSkip(int howMany)
{
    int i = genLoc;
    genLoc += howMany;
    if (highgenLoc < genLoc)  highgenLoc = genLoc;
    return i;
}
```

有跳过的部分，就要有返回和恢复的部分：

```
// 返回
void genBackup(int loc)
{
    genLoc = loc;
}
// 恢复
void genRestore(void)
{
    genLoc = highgenLoc;
}
```

接下来，递归遍历语法树，生成中间代码，下面以 IF 语句为例来描述代码生成过程：

```
    case IfK:
        p1 = tree->child[0];
        p2 = tree->child[1];
        p3 = tree->child[2];
        recursive_gen(p1); // 生成子节点1部分的中间代码
        savedLoc1 = genSkip(1); // 跳过 1 行
        recursive_gen(p2); // 生成子节点2部分的中间代码
        savedLoc2 = genSkip(1); // 跳过 1 行
        currentLoc = genSkip(0); // 跳过 0 行
        genBackup(savedLoc1); // 返回到 savedLoc1
        genRM_Abs("JEQ", ac, currentLoc); // 生成中间代码
        genRestore(); // 恢复
        recursive_gen(p3); // 生成子节点3部分的中间代码
        currentLoc = genSkip(0);  // 跳过 0 行
        genBackup(savedLoc2); // 返回到 savedLoc2
        genRM_Abs("LDA", pc, currentLoc); // 生成中间代码
        genRestore(); // 恢复
        break;
```

其他语句也同理，故不做赘述。

采用递归的方式来生成中间代码：

```
// 递归生成中间代码
void recursive_gen(struct Node* tree)
{
    if (tree != NULL)
    {
        switch (tree->nodekind) {
        case StmtK:
            genStmt(tree);
            break;
        case ExpK:
            genExp(tree);
            break;
        default:
            break;
        }
        recursive_gen(tree->sibling);
    }
}
```

试验测试结果请见第五部分。

5. 测试结果

5.1 语义分析结果

故意将程序写成如下有错误的形式，测试语义分析的报错：

```
INT MAIN f1()
BEGIN
    INT x;
    INT x;
    x := 64;
    REAL y;
    y := 6;
    z := 6;
    IF(x != y)
    BEGIN
        y:=x;
        x:=y;
    END
END
```

经语义分析，发现了出现的四种类型的错误，一是重复声明变量 x，二是使用未被声明过的符号 z，三是赋值语句两边的符号类型不相同，四是运算符两边的符号类型不相同。

```
Check errors:
-------------------------------------------------------
Symb error! x has already been declared! line:4
Symb error! z has not been declared! line:8
Type error! Assignment is not the same type! line 8
Type error! Different types in this operation! line 9
-------------------------------------------------------
```

生成的符号表为：

```
Symbol table:
-------------  --------   ------------
Variable Name  Location   Line Numbers
-------------  --------   ------------
x              0           3    5    9   11   12
y              1           6    7    9   11   12
-------------  --------   ------------
```

将测试程序简化成如下形式，进行测试：

```
INT MAIN f1()
BEGIN
    INT x;
    x := 64;
    REAL y;
    y := 6;
    IF(x != y)
    BEGIN
        y:=x;
        x:=y;
    END
END
```

生成符号表:

```
Symbol table:
-------------  --------  ------------
Variable Name  Location  Line Numbers
-------------  --------  ------------
x              0           3    4    7    9   10
y              1           5    6    7    9   10
-------------  --------  ------------
```

生成中间代码:

```
 0:     LD  6,0(0)
 1:     ST  0,0(0)
 2:     ST  0,0(6)
 3:     LD  1,0(6)
 4:    SUB  0,1,0
 5:    JEQ  0,2(12)
 6:    LDC  0,0(0)
 7:    LDA  12,1(12)
 8:    LDC  0,1(0)
 9:    JEQ  0,1(12)
10:    LDA  12,0(12)
11:   HALT  0,0,0
```