6.5 TINY语言的语义分析

这一节我们基于前一章构造的 TINY语法分析程序，开发 TINY语言的语义分析程序代码。语义分析程序所基于的TINY的语法和语法树结构在3.7节描述。

TINY语言在其静态语义要求方面特别简单，语义分析程序也将反映这种简单性。在 TINY中没有明确的说明，也没有命名的常量、数据类型或过程；名字只引用变量。变量在使用时隐含地说明，所有的变量都是整数数据类型。也没有嵌套作用域，因此变量名在整个程序有相同的含义，符号表也不需要保存任何作用域信息。

在TINY中类型检查也特别简单。只有两种简单类型：整型和布尔型。仅有的布尔型值是两个整数值的比较的结果。因为没有布尔型操作符或变量，布尔值只出现在 if或repeat语句的测试表达式中，不能作为操作符的操作数或赋值的值。最后，布尔值不能使用 *write*语句输出。

我们把对TINY语义分析程序的代码的讨论分成两个部分。首先，讨论符号表的结构及其

相关的操作。然后，语义分析程序自身的操作，包括符号表的构造和类型检查。

6.5.1 TINY的符号表

在TINY语义分析程序符号表的设计中，首先确定什么信息需要在符号表中保存。一般情况这些信息包括数据类型和作用域信息。因为 TINY没有作用域信息，并且所有的变量都是整型，TINY符号表不需要保存这些信息。然而，在代码产生期间，变量需要分配存储器地址，并且因为在语法树中没有说明，因此符号表是存储这些地址的逻辑位置。现在，地址可以仅仅看成是整数索引，每次遇到一个新的变量时增加。为使符号表更加有趣和有用，还使用符号表产生一个交叉参考列表，显示被访问变量的行号。

作为符号表产生信息的例子，考虑下列TINY程序的例子(加上了行号)：

```
 1: { Sample program
 2:   in TINY language --
 3:   computes factorial
 4: }
 5: read x; { input an integer }
 6: if 0 < x then { don't compute if x <= 0 }
 7:   fact := 1;
 8:   repeat
 9:     fact := fact * x;
10:     x := x - 1
11:   until x = 0;
12:   write fact { output factorial of x }
13: end
```

这个程序的符号表产生之后，语义分析程序将输出 (**TraceAnalyze= True**)下列信息到列出的文件中：

```
Symbol table:
Variable Name   Location   Line  Numbers
----------------------------------------
x               0          5     6      9     10     10     11
fact            1          7     9      9     12
```

注意，在符号表中同一行的多次引用产生了那一行的多个入口。

符号表的代码包含在**symtab.h**和**symtab.c**文件中，在附录B中列出(分别是第1150到1179行和第1200到1321行)。

符号表使用的结构是在6.3.1节中描述的分离的链式杂凑表，杂凑函数是程序清单 6-2给出的。因为没有作用域信息，所以不需要 *delete*操作，*insert*操作除了标识符之外，也只需要行号和地址参数。需要的其他的两个操作是打印刚才列出的文件中的汇总信息，以及 *lookup*操作，从符号表中取出地址号(后面的代码产生器需要，符号表生成器也要检查是否已经看见了变量)。因此，头文件**symtab.h**包含下列说明：

```
void st_insert ( char * name, int lineno, int loc );
int st_lookup ( char * name );
void printSymTab(FILE * listing);
```

因为只有一个符号表，它的结构不需要在头文件中说明，也无须作为参数在这些过程中出现。

在**symtab.c**中相关的实现代码使用了一个动态分配链表，类型名是**LineList**(第1236行到第1239行)，存储记录在杂凑表中每个标识符记录的相关行号。标识符记录本身保存在一个

“桶”列表中，类型名是**BucketList**(第1247行到第1252行)。**st_insert**过程在每个“桶”列表(第1262行到第1295行)前面增加新的标识符记录，但行号在每个行号列表的尾部增加，以保持行号的顺序(**st_insert**的效率可以通过使用环形列表或行号列表的前/后双向指针来改进；参见练习)。

6.5.2 TINY语义分析程序

TINY的静态语义共享标准编程语言的特性，符号表的继承属性，而表达式的数据类型是合成属性。因此，符号表可以通过对语法树的前序遍历建立，类型检查通过后序遍历完成。虽然这两个遍历能容易地组合成一个遍历，为使两个处理步骤操作的不同之处更加清楚，仍把它们分成语法树上两个独立的遍。因此，语义分析程序与编译器其他部分的接口，放在文件**analyze.h**中(附录B，第1350行到第1370行)，由两个过程组成，通过下列说明给出

```
void buildSymtab(TreeNode *);
void typeCheck(TreeNode *);
```

第1个过程完成语法树的前序遍历，当它遇到树中的变量标识符时，调用符号表**st_insert**过程。遍历完成后，它调用**printSymTab**打印列表文件中存储的信息。第2个过程完成语法树的后序遍历，在计算数据类型时把它们插入到树节点，并把任意的类型检查错误记录到列表文件中。这些过程及其辅助过程的代码包含在**analyze.c**文件中（附录B，第1400行到第1558行）。

为强调标准的树遍历技术，实现**buildSymtab**和**typeCheck**使用了相同的通用遍历函数**traverse**(第1420行到第1441行),它接受两个作为参数的过程(和语法树)，一个完成每个节点的前序处理，一个进行后序处理：

```
static void traverse ( TreeNode * t,
               void (* preProc) (TreeNode * ),
               void (* postProc) (TreeNode * ) )
{ if (t != NULL)
  { preProc(t);
    { int i;
      for (i=0; i < MAXCHILDREN; i++)
        traverse(t->child[i],preProc, postProc);
    }
    postProc(t);
    traverse(t->sibling, preProc, postProc);
  }
}
```

给定这个过程，为得到一次前序遍历，当传递一个“什么都不做”的过程作为**preproc**时，需要说明一个过程提供前序处理并把它作为**preproc**传递到**traverse**。对于TINY符号表的情况，前序处理器称作**insertNode**，因为它完成插入到符号表的操作。“什么都不做”的过程称作**nullProc**，它用一个空的过程体说明(第1438行到第1441行)。然后建立符号表的前序遍历由**buildSymtab**过程(第1488行到第1494行)内的单个调用

```
traverse (syntaxTree, insertNode, nullProc);
```

完成。类似地，**typeCheck**(第1556行到第1558行)要求的后序遍历由单个调用

```
traverse (syntaxTree, nullProc, checkNode);
```

完成。这里**checkNode**是一个适当说明的过程，计算和检查每个节点的类型。现在还剩下描述过程**insertNode**和**checkNode**的操作。

insertNode过程(第1447行到第1483行)必须基于它通过参数 (指向语法树节点的指针)接受的语法树节点的种类，确定何时把一个标识符 (与行号和地址一起)插入到符号表中。对于语句节点的情况，包含变量引用的节点是赋值节点和读节点，被赋值或读出的变量名包含在节点的**attr.name**字段中。对表达式节点的情况，感兴趣的是标识符节点，名字也存储在**attr.name**中。因此，在那 3个位置，如果还没有看见变量 **insertNode**过程包含一个

```
st_insert (t->attr.name, t->lineno, location++);
```

调用(与行号一起存储和增加地址计数器)，并且如果变量已经在符号表中，则

```
st_insert (t->attr.name,t->lineno,0);
```

(存储行号但没有地址)。

最后，在符号表建立之后， **buildSymtab**完成对**printSymTab**的调用，在标志**TraceAnalyze**的控制下(在**main.c**中设置)，在列表文件中写入行号信息。

类型检查遍的**checkNode**过程有两个任务。首先，基于子节点的类型，它必须确定是否出现了类型错误。其次，它必须为当前节点推断一个类型(如果它有一个类型)并且在树节点中为这个类型分配一个新的字段。这个字段在**TreeNode**中称作**type**字段(在**globals.h**中说明，见附录B，第216行)。因为仅有表达式节点有类型，这个类型推断只出现在表达式节点。在TINY中只有两种类型，整型和布尔型，这些类型在全局说明的枚举类型中说明(见附录B，第203行)：

```
typedef enum {Void, Integer, Boolean} ExpType;
```

这里类型 **Void**是“ 无类型 ”类型，仅用于初始化和错误检查。当出现一个错误时，**checkNode**过程调用**typeError**过程，基于当前的节点，在列表文件中打印一条错误消息。

还剩下归类**checkNode**的动作。对表达式节点，节点可以是叶子节点 (常量或标识符，种类是**ConstK**或**IdK**)，或者是 操作符节点(种类**OpK**)。对叶子节点的情况(第1517行到第1520行)，类型总是**Integer** (没有类型检查发生)。对操作符节点的情况(第1508行到第1516行)，两个子孙子表达式的类型必须是**Integer** (因为后序遍历已经完成，已经计算出它们的类型)。然后，**OpK**节点的类型从操作符本身确定 (不关心是否出现了类型错误)：如果操作符是一个比较操作符(<或=)，那么类型是**Boolean**；否则是**Integer**。

对语句节点的情况，没有类型推断，但除了一种情况，必须完成某些类型检查。这种情况是**ReadK**语句，这里被读出的变量必须自动成为**Integer**类型，因此没有必要进行类型检查。所有4种其他语句种类需要一些形式的类型检查： **IfK**和**RepeatK**语句需要检查它们的测试表达式，确保它们是类型**Boolean**(第1527行到第1530行和第1539行到第1542行)，而**WriteK**和**AssignK**语句需要检查(第1531行到第1538行)确定被写入或赋值的表达式不是布尔型的 (因为变量只能是整型值，只有整型值能被写入)：

```
x := 1 < 2; { error - Boolean value
                  cannot be assigned }
write 1 = 2; { also an error }
```