实现类型检查

徐子越

November 2019

目录

1 作业要求

实现类型检查模块, 在上一次作业构造的语法树上进行后序遍历:

1. 实现类型检查、结果类型赋值等
2. 然后打印出语法树，对每个结点打印类型信息（类型错误）

2 作业思路

2.1 大体流程

根据作业要求，本次实验主要是在之前的基础上实现类型检查、结果类型赋值等操作，在打印一颗语法树的同时, 打印包括每个结点的类型信息以及错误信息。具体操作可以参考讲义 6 中的 TinyC 类型检查的代码。

首先我们需要明确类型检查的具体流程即在整个语法树建立完成后，按照后续遍历对每个节点进行类型检查、结果类型赋值等操作。我们不妨参考 TinyC 的实现代码，如下所示，由于 typeCheck 函数是在语法树建立完成后开始扫描节点并打印错误信息等操作，所以该函数的主要目的就是对语法树进行遍历，在本例中利用 traverse 函数实现。

traverse 函数利用其参数传入的函数指针继而完成指定的函数操作，同时可以指定前序遍历和后续遍历完成不同的操作。在本次实验中，我们只需要后续遍历，所以将前序遍历的函数 nullProc 置为空即可。

当然，这只是一种实现方法，由于在打印节点的时候也是后续遍历，所以你也可以在打印节点信息之前进行类型检查等操作。

frame

```
1  void typeCheck(TreeNode * syntaxTree)
2  {
3          traverse(syntaxTree,nullProc,checkNode);
4  }
5  static void traverse( TreeNode * t, void (* preProc)
6          (TreeNode *),void (* postProc) (TreeNode *) )
7  { if (t != NULL)
8    { preProc(t);
9      { int i;
10       for (i=0; i < MAXCHILDREN; i++)
11         traverse(t->child[i],preProc,postProc);
12     }
13     postProc(t);
14     traverse(t->sibling,preProc,postProc);
```

```
15    }
16 }
```

typeChecktraverse

2.2 类型检查函数

在明确大体流程之后，本次实验的重点就是完成类型检查函数 CheckNode 的具体实现，下面为 TinyC 实现的函数：

frame

```
1  static void checkNode(TreeNode * t)
2  { switch (t->nodekind)
3    { case ExpK:
4        switch (t->kind.exp)
5        { case OpK:
6            if ((t->child[0]->type != Integer) ||
7                (t->child[1]->type != Integer))
8              typeError(t,"Op applied to non-integer");
9            if ((t->attr.op == EQ) || (t->attr.op == LT))
10             t->type = Boolean;
11           else
12             t->type = Integer;
13           break;
14         case ConstK:
15         case IdK:
16           t->type = Integer;
17           break;
18         default:
19           break;
20       }
21       break;
22     case StmtK:
23       switch (t->kind.stmt)
24       { case IfK:
25           if (t->child[0]->type == Integer)
26             typeError(t->child[0],"if test is not Boolean");
```

```
27            break;
28          case AssignK:
29            if (t->child[0]->type != Integer)
30              typeError(t->child[0],
31              "assignment of non-integer value");
32            break;
33          case WriteK:
34            if (t->child[0]->type != Integer)
35              typeError(t->child[0],"write of non-integer value");
36            break;
37          case RepeatK:
38            if (t->child[1]->type == Integer)
39              typeError(t->child[1],"repeat test is not Boolean");
40            break;
41          default:
42            break;
43        }
44        break;
45      default:
46        break;
47
48    }
49  }
```

checkNode

代码中给大家提供的思路与上一次作业中构造语法树的思路如出一辙，主要为根据节点类型以及其子类型确定了该节点的具体操作。下面将对不同类型节点进行分析：

首先是表达式类型的节点，具体为：

1. 符号节点（OpK）：首先保证符号两边的节点类型皆为整形（由于 TinyC 不支持字符类型，若实现的编译器支持字符类型，类型检查应当更为复杂），之后将该节点的类型进行赋值。
   TinyC 的实现代码较为简单，大家应当积极思考，根据自己编译器所支持的变量以及不同的操作符进行更为细致的检查。值得一提的是，赋值操作节点类型可以不需要进行赋值，但是在对变量的节点类型进行赋值时需要进行查表的操作，故本次实验中，符号表的实现是必须的。此外，可以在符号表中设置两个标志位，标志该变量是否定义以及是否初始化，如果出现未初始化就进行各种操作的情况，需要打印“该变量未初始化”的错误。

2. 标识符节点（IdK）：将标识符的类型进行赋值，由于编译器中不止支持 int 型变量，不能盲目赋

值，需要进行查表确定变量的类型。

然后是语句类型的节点，具体为：

1. If 语句（IfK）：需要保证第一个子节点为布尔型、整型，TinyC 支持变量类型较少，所以实现较为简单。
2. 赋值语句（AssignK）：如果将表达式中的赋值表达式处理完整，则不需要在此部分进行过多处理。
3. While 语句（WhileK）以及 For 语句（ForK）：需要检查第一个子节点为布尔型、整型等（取决于你的编译器支持哪些变量）。

最后是声明类型的节点，在 TinyC 中没有做任何处理，但是可以实现较多的类型检查操作，例如在变量声明时检查该标识符是否已经存在于符号表中，若不在符号表，则将其添加至符号表，更为复杂一些的，在声明的同时对变量进行赋值操作，也需要检查重定义问题。

以上就是本次实验的全部内容