『编译技术』SysY-Mips编译器设计——语义分析(符号表管理与错误处理)

章节目录

零. 作业要求

设计并实现错误处理程序，能够诊断出题目要求的常见语法语义错误，并进行适当的局部化容错处理，输出程序中所有错误信息，包括类型与位置。

一. 错误编码

本次作业只针对如下数条错误情况予以考虑。

- **a 非法符号**：在字符串中出现的非法符号
- **b 名字重定义**：变量名在当前作用域下的重复定义，内层覆盖外层定义是合法的，必须是同层作用域才出错
- **c 未定义的名字**：使用未定义的标识符
- **d 函数参数个数不匹配**：调用语句中传参个数与符号表中函数定义时参数个数不一致
- **e 函数参数类型不匹配**：同上，参数类型不匹配
- **f 无返回值的函数存在不匹配的return语句**：void函数中任意地方出现return <Exp>。
- **g 有返回值的函数缺少return语句**：只需要判断函数末尾有无return，报错行号为}所在行号
- **h 改变常量的值**：对const定义的常量进行修改
- **i 缺少分号**
- **j 缺少右小括号**
- **k 缺少右中括号**：以上三个报错行号为前一个非终结符所在行号
- **l 输出语句格式字符与个数不匹配**
- **m 非循环块使用continue和break**

上述错误可大致分为两种，**语法错误**和**语义错误**，语法错误包括缺少符号(i,j,k)与输出语句中存在非法字符(a)，这些错误不满足文法要求；其余皆是语义错误，这些错误满足文法，然而不满足SysY语义限制。

进一步分析，其中b,c,d,e,h五种错误需要用到**符号表**，其余错误无需符号表辅助，可以通过在语法分析器基础上增添功能找寻错误。

二. 符号表

建立符号表

无论错误处理是否需要，建立起符号表都是必须的，符号表中表项结构如下所示。

BasicSymbol：

属性名	ID	名称	类型	是否常量	行号
描述	自增ID，Symbol的唯一标识符	符号名	符号类型，规定 0->int,1->int[],2->int[][],3->func	规定 0->变量，1->常量	符号的行号，错误处理需要用到

FuncSymbol：

属性名	函数返回参数类型	函数形参个数	函数形参类型列表
描述	规定 0->int, 1->void	记录函数形参个数	记录函数每个形参类型，规定如上

24.10.11补充：使用数字作为类型标记是不推荐的，易混淆也不方便他人理解，使用枚举类解决会更好！

每一张符号表存储`HashMap<String, Symbol>`，设计成哈希表一方面是由于每一张符号表的符号名不会重复，另一方面提高用符号名查询到对应`Symbol`的效率。

作用域

每一个作用域对应一张符号表，开始一个新的作用域有两种情况，程序开始时的全局作用域以及每个`<Block>`都会开启新的作用域，因此要在这两处位置新建符号表。特别注意的是，**函数定义时的形参应当放到紧接着函数主体开启的符号表当中。**

作用域存在**树状结构**关系，符号表同理，查找符号时应当从当前作用域的符号表向全局作用域的符号表由底至上的搜索，这便需要我们记录下符号表的父节点。

因此一张符号表的结构如下所示。

属性	ID	父节点ID	包含的所有符号
描述	自增ID，SymbolTable唯一标识符	父符号表ID	`HashMap<String, Symbol>`

在`Parser`中我们存储`HashMap<Integer(id),SymbolTable>`，便于我们通过`id`快速查找到对应的符号表。

三. 错误处理

建立起符号表后，便可进行符号表相关的错误处理。符号表在当前需要提供如下两个功能。

- 给定一标识符名，在当前符号表中查找是否出现过同名符号。
- 给定一标识符名，在树状符号表集合中由底至上查找是否出现过同名符号，直至全局作用域的符号表。

第一个功能适用于判断是否出现**标识符重定义**；
第二个功能适用于判断是否出现**标识符未定义**，然而查到距离当前层最近的符号后，并不能直接下出标识符已定义的结论，还需要比较查到的符号类型和解析当前语法下需要的标识符类型，才能下定结论。

举例而言，`LVal`需要常变量类型，`UnaryExp`的函数调用分支需要函数类型。

```
// 如下所示 符号表查到的是函数类型的fun，而我们需要的是变量类型的fun，此时会出现“名字未定义”的错误
void fun() {
}
int main() {
    fun = 1;
    return 1;
}
```

其他错误的判断逻辑不过多赘述。

四. 错误处理存在的Bug

- 当处理`[<Exp>];`中分号漏写的情况时产生问题。

问题原因：由于`Stmt`在判断`AssignStmt`,`InputStmt`,`ExpStmt`时使用的预读法会读走一个`Exp`，接着判断下一个单词是否是`';'`来判断是属于哪类语句，然而若`';'`不存在，则判断会产生异常。
解决方法：将判断下一个字符是否为`';'`改为判断是否为`'='`即可，`'='`不会出现缺失的情况。

- `<UnaryExp> -> Ident '(' {<FuncRParams>} ')'`缺少右小括号的错误处理，即缺失某些符号时会在语法解析时引起二义性，例如：

  ```
  // 正常
  b = a() + 3;
  // 缺失')'
  b = a( + 3;
  ```

问题原因：之前只需预读两位后判断当前单词是否是`')'`即可判断是否存在实参，而存在缺失括号后，这个方法便不再适用

解决方法：预读两位后判断当前单词是否是`<FuncRParams>`的FIRST集来判断是否存在实参。

前提假设：由于`<FuncRParams>`的 FIRST 集包括`{'+', '-'}`，而`<UnaryExp>`的 FOLLOW 集中也包括`{'+', '-'}`，一旦去掉右小括号后会产生歧义。前提假设不会出现这种情况，即出现`{'+', '-'}`时将其看作解析`<FuncRParam>`处理，这也符合C语言编译器的行为。

- 主函数也要考虑无返回值的问题。
- `'%'` ascii码为37； `'\'` ascii码92 但是当且仅当和`\n`一起出现才算做合法，语法定义问题。
- 实参形参类型匹配中，应考虑到当实参是函数调用存在无返回值的错误情况。

解决方法：遍历`Exp`树，将所有函数调用的函数名取出，查**符号表**依次判断其**返回类型**，若至少有一个`void`则需错误处理。

- `printf`语句中同时出现非法字符和参数数目不相同。

解决方法：为减少字符串遍历次数，我在词法分析时便记录了"%d"的个数与是否存在非法字符，并将其打包进`wordInfo`的实体中`value`属性。我们尝试将两部分信息合二为一，我们规定`value`记录"%d"的数量，若存在非法字符，将`value`取负再减一，如此语法分析器可通过`value`的正负信息及大小信息得到上述两种信息。

- 处理注释时要保留换行符，不然行数会被打乱！

五. 总结

- **加入错误处理后语法解析思路需要改变**：语法解析不能再过度依赖可能被删减的单词`')', '}', ']', ';'`，需要另寻他路。

  例如，在给`<Stmt>`分类时，赋值语句，表达式语句和输入语句需要预读判断，之前的思路是试探解析`Exp`后判断此时栈顶的单词是否为`';'`，若是，则判定为表达式语句，反之则在另外两种语句中继续判断。而现在面临`';'`可能缺失的问题，因此我们需要转换思路，输入语句和赋值语句在解析完Exp后下一个单词一定为`'='`，反之则为表达式语句，颠倒了判断思路后即可解决。总之，**不能依赖于可能丢失的单词作为解析后续文法的依据。**
- 未考虑到的错误情况仍存在很多，且该任务已经为我们做了很多简化，每多考虑一种错误情况难度便会更上升一截。
- 学会使用C语言编译器(gcc)与我们的编译器做对照，观察gcc在错误处理时的行为，包括报错位置以及报错类型，尤其是存在多个报错的争议部分(比如名字未定义和函数参数类型不匹配同时出现)，并尽可能使我们的错误处理逻辑与其保持一致！