Lab3实验报告

191850205 夏宇

1 实现功能

在Lab2生成的语法分析树的基础上，利用`listener`模式遍历语法分析树，进行类型检查和语义分析，识别源代码中存在的错误类型，并输出错误信息。

2 设计与实现

实现思路主要依据助教提供的Antlr-L3实验说明。

2.1 定义类型

首先创建一个枚举类，其保存了本次实验的所有可能的变量类型。添加一个抽象父类，该抽象父类中定义了一个枚举类型的成员变量用来表示此对象是哪种类型的变量，每一个具体的类型都需要继承该抽象父类。Array类型需要额外成员变量保存其长度与成员类型；Structure类型需要额外成员变量保存其域中变量（以链表实现）；Funtion类型需要额外两个成员变量，一个保存其返回值类型，另一个保存其参数列表（以链表实现）。

2.2 设计符号表

添加了`SymbolTable`类来表示符号表，类中有一个`Hashtable`类型的成员变量，用来保存变量名（String）和变量类型的对应关系。每定义一个新的函数或者变量都要添加到符号表中，一遍判断变量是否重定义或未定义就使用。

2.3 遍历语法分析树

采用`listener`模式遍历语法分析树。因为我主要在exit时处理语法单元，所以此遍历方式整体上可以视作深度优先遍历，在访问完子节点的exit方法之后再访问父节点的exit方法。这就需要一种机制在子节点和父节点之间传递信息，于是添加一个`ParseTreeProperty<Type> values = new ParseTreeProperty<>();`成员变量，用于将子节点的类型信息传回给父节点。

由于语法单元exp较为复杂且会自递归，为了区别于`values`承担的传递子节点类型作用同时帮助我自己理解，我还添加了一个新的变量`IdentityHashMap<ParserRuleContext, Type> expValues = new IdentityHashMap<>();`,该变量的作用与`values`类似，不过专门用于在`exp`节点之间传递信息。

为了区分结构体定义与结构体变量，添加新的成员变量`HashMap<String, Structure> structureDef = new HashMap<>();`，该变量专门用来保存结构体的定义，每创建一种新的结构体类型都要添加到该表中。当创建新的结构体变量时，只需查阅此表，便能判断该结构体类型是否已经定义。由于此表的添加，当创建新的结构体、普通变量或者函数时，则需要同时查询结构体定义表与符号表，来判断是否重名。

2.4 小设计

除了`INT、FLOAT、FUNCTION、STRUCTURE、FUNCTION`这五种类型，我自己还添加了一个`ERROR`类型，当某一个语法单元出错时，可以将其标识为ERROR类型并添加到`expValues`表中，当父节点需要用到子节点时，如果查表发现子节点为`ERROR`类型，则可以在父节点层次进行错误处理，避免该错误一直向上传播而引起连锁反应输出过多的报错信息。

3 遇到的部分bug

3.1 区分结构体定义与结构体创建带来的bug

由于结构体定义需要将结构体加入结构体定义表中，而结构体创建则不需要，所以我想要在`exitStructSpecifier`方法中区分结构体定义与创建新的结构体变量，从而将结构体定义留在`exitStructSpecifier`方法中处理，而将结构体变量的创建留到`exitDef`方法中处理。但是我一开始错误地认为结构体定义只会在`extDef: specifier SEMI`出现，于是我直接依据当前structSpecifier节点的向上两层父节点的类型来判断是否是在添加新的结构体定义。但是我忽略了一个重要的点，就是在结构体或者函数内部，依据语法规则`def : specifier decList SEMI;`，可以在添加新的结构体定义的同时创建结构体变量，这就导致在函数或结构体内部新增的结构体定义没有被添加到结构体定义表`structureDef`中，导致bug。

解决方法：

仔细观察`structSpecifier`的语法规则可以发现，结构体定义无论是只定义还是在定义的同时创建结构体变量，都含有`opTag`这一节点，于是可以依据`structSpecifier`节点是否含有`opTag`这一子节点来判断是否是结构体的定义。

说明：这里的基本思想是尽可能将任务留给底层节点处理，因为结构体定义`structSpecifier : STRUCT optTag LC defList RC`时，`structSpecifier`节点已经了解了所有需要创建新的结构体类型的所有信息，包括结构体名（opTag），结构体域中变量（defList），所以将创建结构体定义的任务留在`exitStructSpecifier`中执行；而对于创建新的结构体变量，则需要同时知道结构体类型与变量名信息，能掌握这些信息的最底层节点就是`def`节点了`def : specifier decList SEMI;`所以将创建新的结构体变量的任务交给`exitDef`方法执行。

3.2 判断结构体或数组是否等价

初始我判断结构体是否等价直接根据结构体的域中变量类型是否一致，若域中变量数量和类型一致，则认为结构体类型一致。这无法区分如下两个结构体，因为我直接判断a变量和b变量类型都是数组，则认为结构体结构等价，这是存在的bug。

```
struct s{
    int a[2][2];
};
struct t{
    int b[2];
}
```

对于数组类型，情形与结构体类型差不多，即数组为多维数组，要一个维度一个维度判断，而且数组基类型可能是结构体类型，因此还需要判断基类型是否一致。

以判断数组类型是否一致为例：首先粗糙判断基类型是否一致，若不一致直接返回false；如果基类型也是数组类型，则需要递归判断此数组类型是否一致，若不一致直接返回false；若基类型是结构体类型，则需要判断结构体类型是否一致，不一致则返回false。

说明：其实本次实验我还碰到了许多其它bug，但是由于实验周期以及代码复杂度的原因，许多bug已经遗忘。不过剩下的bug大多无关紧要，是一些细节上出现的问题。

4 总结与反思

本次lab我尚未完全ac。事实上在我写完OJ上提供的17中类型错误样例后仅仅只有1000分出头，后面大量时间在修改代码中bug，最终也只得到2600分。因为考试周的原因，本次lab我基本上是在两个间隔较大的时间段内进行的，这导致在时间段2中我没有办法掌控整个框架。而且虽然我一开始想着职责的划分，把能完成的任务交给尽可能底层的节点完成，底层节点没有足够信息处理的才交给父层节点处理，但是在实际编码过程中，我并没有很好的贯彻这一思路，导致代码逻辑较为混乱，以致于出现bug时需要同时修改多个节点对应的方法。同时代码耦合度高，甚至达到了内容耦合的地步，一个条件语句中的代码和另一个条件分支语句中的代码几乎完全一致（复制粘贴而已）。

在以后的编码过程中，可以多思考，明确不同层次、不同方法之间的职责划分，最好是以文档的形式记录下来，这样一方面能避免遗忘，另一方法记录的过程也是思考的过程，能让我更加体会框架的合理与不合理之处；其次，要进行拆分，当一段代码在多处被调用时，应该考虑独立使其成为一个新的方法，降低内容耦合，也减少后续修改代码的复杂度。希望我能贯彻落实，继续进步！