PA1-B 实验报告

李潇
计 63
学号: 2016011303

2018 年 11 月 3 日

1 实验总述

本次实验主要任务是在上次对于框架和词法语法分析了解的基础上，手工实现自顶向下语法分析程序，实现对于 LL(1) 语句的分析，并支持一定的错误恢复。

2 具体实现

2.1 错误恢复部分

我采用的错误处理方法，和本次说明文件中方法类似，即借鉴短语层恢复中 endSym 集合的思想，相当于加入了一定的上下文信息。具体流程可如下所示：

1. 计算 A 的 Begin 和 End 集合。

2. 若 a 位于 A 的 Begin 集合中，则正常处理；若 a 不在 A 的 Begin 集合中，则在当前位置报错，然后跳过一系列终结符。

3. 当遇到 Begin 或者 End 集合中所包含的终结符时，停止跳过。若是 Begin 中的终结符，则恢复分析 A；若是 End 中的终结符，则 A 分析失败，继续分析 A 后面的下一个非终结符。

实现源代码如下：

```
1 Set<Integer> endset = new HashSet<Integer>();
2 endset.addAll(followSet(symbol));
3 endset.addAll(follow);
4 if (!beginset.contains(lookahead)) {
5     error();
6     while (true) { //向后跳过运算符
```

```
7           if (beginset.contains(lookahead))
8               return parse(symbol, follow);
9           else if (endset.contains(lookahead)) return null;
10          else lookahead = lex();
11      }
12  }
13  int actionId = result.getKey();
14  List<Integer> right = result.getValue();
15  int length = right.size();
16  SemValue[] params = new SemValue[length + 1];
17  for (int i = 0; i < length; i++) {
18      int term = right.get(i);
19      params[i + 1] = isNonTerminal(term)
20              ? parse(term, endset)
21              : matchToken(term)
22              ;
23  }
24  params[0] = new SemValue();
25  try {   act(actionId, params);
26          return params[0];
27      } catch (NullPointerException e) {
28          return null;
29  }
```

2.2 LL(1) 文法实现新特性

2.2.1 对象浅复制语句和禁止类继承

由于提供的文法本身就不会出现左公因子问题，故这两个的实现基本和第一次实验相同，此处就不再赘述了。

2.2.2 串行条件卫士语句

串行条件语句是以终结符 IF 开始，若直接接在 IfStmt 上，会和 if 语句产生左公因子，这可以通过将其左公因子提出解决；此外，在 GuardStmt 中，需要消除左递归，最终可以将其转化为 LL(1) 文法。值得注意的是，这次实验和上次消除递归表示式顺序上是相反的，因为上次是通过压栈的方式进行解决。

2.2.3 简单自动类型推导

由于该产生式右侧的第一个终结符为 VAR，所以不会产生左公因子，符合 LL(1) 条件。直接按照参考语法写即可。

2.2.4 数组相关操作

对于数组常量，需要注意左递归的消除，方法和上述串行条件卫士语句相同。

对于数组初始化和数组拼接，其均可看作一个二元操作表达式，但要注意优先级的定义。具体地需要仿照 Expr 的拆分方式，在 Expr4A 处利用文法显式定义优先级。

对于取子数组表达式和下表动态访问，由于上面两个新特性中均使用到了‘[’和‘]’，为了构造 LL(1) 文法，必须整合这些公共的因子。将文法拆分后可以如下：

Expr8 -> Expr9 Expr81

Expr81 -> ‘[’Expr Expr811

Expr811 -> ‘]’Expr812 | ‘:’Expr ‘]’

Expr812 -> Expr81 | DEFAULT Expr9

使用如上产生式，即可满足是 LL(1) 的。

2.2.5 Python 风格数组和数组迭代

直接使用参考文法就是 LL(1) 的，故沿用上次的代码即可。

3 冲突处理及解决方案

通过 wiki 上的描述，可以看到在 Unstrict 模式之下，产生式是存在默认优先级的。先定义的产生式具有更高的优先级，故在产生矛盾时，可以根据优先级选择产生式。

对于产生式 ElseClause –> ELSE Stmt | /* empty */，ELSE 显然属于其预测集合的交，故照理应该会发生冲突。但是，按照上述模式，ElseClause –> ELSE Stmt 的优先级要高于 ElseClause -> /* empty */，故在遇到矛盾的情况时，会优先采用前者。

体现在 Table 代码中就表现为，当 query 某个 symbol 和 lookahead 对应的产生式的时候，，本应该会有多个产生式都可以返回，但事实上只返回 switch-case 句中顺序靠前的那一个产生式，另外的就忽略了。

这样的处理方式虽然能 cover 住大多数的情况，但终究是不完善的，比如当遇到如下程序片段时:

```
1 if (true)
2 if (true)
3     print(1);
```

```
4 else
5     print(2);
```

按照之前的处理方式，其代码将会被理解为:

```
1 if (true) {
2     if (true) {
3         print(1);
4     } else { print(2);}
5 }
```

但实际上其语义还存在另一种可能:

```
1 if (true) {
2     if (true) {
3         print(1);
4     }
5 } else { print(2); }
```

要想解决这种冲突，可以如上所示，利用括号进行分块，明确对应 if 和 else 的关系。同时，为了避免这种冲突的情况，可以通过设置 pg 工具为 Strict 模式来使这种冲突情况通不过编译。

4 为什么 comprehension 表达式文法改写困难

按照原来的语法实现来分析原因可知: 在之前实现了 ArrayConst 的语法，它是由 Constant 非终结符推出，而 Constant 又由 Expr9 推出。而数组 comprehension 表达式也应该由 Expr9 推出，即这两个语句为同一个非终结符推出，故必须要处理它们的左公因子，但事实上，虽然'[' 可以比较方便的提取出来，但 ArrayConst 在'[' 之后会遇到 Constant, 而 comprehension 表达式会遇到 Expr, 而事实上，由于 Expr 和 Constant 存在推导关系，故必定有相交的 first 集合，故必定会报冲突。

5 误报分析

对于某些由关键字开头引导的语句块问题，当引导关键词发生拼写错误时很容易产生误报。

如串行条件卫士语法，如果误将 if 拼写错误，分析器就会把这个词当做某个用户自定义的标识符 IDENTIFIER，再往后分析，由于也没有词法分析器的回退机制，故在下一个

标识符那里会报一个标识符错误。但更为严重的是，错误恢复机制会导致语法分析器继续往下检查错误，这会将串行条件卫士语句后面整个语句块当做一个普通的语句块，所以会完全不按串行条件卫士语法分析，后面几乎 IfSubStmt 都会产生一个误报。