编译原理实验PA1-B报告

王力
计63
2016011297

2018 年 11 月 4 日

1 总述

本次是实验第二阶段，主要任务是建立在上次一部分实现和对于语法了解的基础上，实现自顶向下语法分析程序，实现对于LL(1)语句的分析。

2 实现细节

2.1 错误恢复部分

我使用的错误处理方法，和本次说明文件中方法类似。采用的主题为应急恢复，借鉴了短语层恢复中endSym集合的思想，相当于加入了一定的上下文信息。同时，我的实现细节是在Parser类中，使用一个check布尔变量来检测是否已经出现语法错误，如果已经出现语法错误，那么我就将布尔变量置为false，然后跳过所有执行动作的过程。具体代码在源文件中，简单附在下文中。

```
private SemValue parse(int symbol, Set<Integer> follow) {
    //System.out.print("checking for "+name(symbol));

    //establish EndSym set
    Set<Integer> endSym = new HashSet<Integer>();
    endSym.addAll(follow);
    endSym.addAll(followSet(symbol));
```

```
//check if lookahead is in beginSet
  Map.Entry<Integer, List<Integer>> result = query(symbol,
      lookahead); // get production by lookahead symbol
if (result==null) { //lookahead not in beginSet
   //release error
   error();
   //skip
   while (result==null && !endSym.contains(lookahead)) {
      //skip all the symbols here
      lookahead = lex();
      result = query(symbol, lookahead);
   }
} //while out, lookahead is in begin+end set

while (result!=null) { //if lookahead is in beginSet
   /*System.out.print(name(symbol)+" ->   ");
   printSymbolList(result.getValue());
   System.out.println();*/

   int actionId = result.getKey(); // get user-defined action

   List<Integer> right = result.getValue(); // right-hand side of
       production
   int length = right.size();
   SemValue[] params = new SemValue[length + 1];

   //match all the right symbols
   for (int i = 0; i < length; i++) { // parse right-hand side
       symbols one by one
      int term = right.get(i);
      params[i + 1] = isNonTerminal(term)
            ? parse(term, endSym) // for non terminals: recursively
                parse it
            : matchToken(term) // for terminals: match token
            ;
      if (check && params[i + 1] == null) {
         check = false;
      }
   }
```

2.2 测试结果

这里就不附带图片了，本地给出的样例测试是全部通过的。

3 冲突处理

3.1 if-else语句处理原理

我们从理论学习中知道，如果两个产生式的预测集合PS有交集，那么对于lookahead只有一个字符的时候，必然会出现不知道选哪一个产生式的冲突情况出现。对于这种情况，我在阅读了Table源代码的query函数后，结合wiki工具明白了它的自动处理方法。

就比如源代码中的if-else语句，$PS(ElseClause \Rightarrow ELSE \quad Stmt) \cap PS(ElseClause \Rightarrow /*empty*/) \supset \{ELSE\}$，所以可以很明显看出来有冲突。但是对于pg工具而言，它会自动选择顺序在前面的一个产生式作为优先级高的产生式，然后将发生冲突交叉的后一个产生式的预测集合相关元素删除。体现在Table源代码中就是，当你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，但是在switch case语句中就只返回顺序在前面的那一个产生式，另一个就忽略了。

所以很多冲突的时候也会报unreachable，也就是如果某两个产生式预测集合相交字符完全覆盖了预测集合本身，那么就会有一个产生式在这种自动的优先级选择中永远无法被选择了。

3.2 冲突举例和解决办法

可以举一个我最开始遇到的例子，部分简略代码附在下面。因为是错误代码，所以提交的源代码中是肯定没有的。

```
Expr10        :  Expr11 ExprT10
              ;

ExprT10       :   '[' AfterBracket
              |   '.' IDENTIFIER AfterIdentExpr ExprT10
              |   /* empty */
              ;
```

```
AfterBracket  :  Expr AfterExpr
              ;

AfterExpr     :  ']' AfterReverseBracket
              ;

AfterReverseBracket : DEFAULT Expr ExprT10
                    ;
```

上面这个片段可以很明显看出，因为ExprT10可以为空串，同时Expr很显然可以通过一系列的推导得到$Expr \Rightarrow Expr10 \Rightarrow Expr11ExprT10$，那么对于上面最后一个产生式而言，可以看出，将Epxr替换掉之后就可能有ExprT10 ExprT10这样的结构，结合ExprT10可能为空串，可知ExprT10的Follow集合和First集合必定有交，那么已经不满足LL(1)文法性质了。所以必然会有冲突，而且几乎很那改。

然而，本身这个冲突错误的产生不是本质的，是因为我对于语法理解的错误。因为最后一个产生式中的Expr实际上应该是Expr11。因为DEFAULT的优先级相对来说是很高的，所以后面接的应该是一个“原子表达式”，也就是一个作为表达式中的一个不可分割的整体来处理的表达式，而不应该是一个Expr这样的最高级的表达式。

4 为什么原来comprehension文法改写困难

根据我按照原来的语法实现来分析原因的话，我的分析如下。首先，我们前面实现了ArrayConst语法，它是由Constant非终结符推出来的，Constant又是由Expr11，上文提到过，Expr11是表示原子表达式，是作为不可分割的整体，内部内容相当于一个黑箱子，不能给外部任何东西结合。

根据这个属性，我们知道数组comprehension语句其实也是这样的一个原子语句，也应该通过Expr11推出来。那么也就是说，这两个语句是可以归为同一个非终结符推导出来的，所以我们必须要对于它们处理左公因子。但是问题就来了，虽然左中括号比较方便提出来，但是对于ArrayConst来

说，中括号之后会遇到Constant，而对于数组comprehension来说，左中括号之后会遇到Expr。可见，Expr和Constant既不方便提取公因子，又由于其推导关系，必定有相交的first集合，所以必定会报冲突。

5 误报问题

首先我找到的例子是将某些关键字拼写错误的误报情况，比如return一不小心写成retrn，或者static一不小心写成satic。在这种情况之下，有时候会报错位置正确，有时候错误，取决于后面跟着什么。大部分情况之下，对于拼写不对的关键字，分析器会自然当做是某种用户约定的标识符IDENTIFIRE，然后往后找下一个字符，对于return语句来说，就会将错误定位到return关键字后面的一个字符，如果相距较远的话，定位就会十分不准确。

还有就是对于某些由关键字开头引导的语句块问题，比如if ———串行卫士语法，如果误将if写错了，写成iif。分析器就会把这个词当做某个用户自定义的标识符，然后往后分析，由于也没有设置lexer的回退机制，所以等到发现这个可能不是标识符的时候也已经无法回退，然后就是在下一个字符那里报一个标识符错误。更糟糕的是，如果这样一来，错误恢复机制会导致语法分析器继续往下检查错误，会把串行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卫士语法分析，后面就全部误报了。