

# 编译原理 PA1B 实验报告

赵浩宇, 2016012390

November 2017

## 1 实验内容

首先阅读自顶向下语法分析, 并在自顶向下分析方法中加入相应的错误恢复代码, 并且加入新支持的语法特性。

## 2 实验过程

### 2.1 阅读 parser 函数的实现

这个部分比较简单, 读懂了就行了。比较坑的地方在于不知道 `query` 和 `act` 函数在哪个文件里面, 最后发现继承了 `Table`。所以建议助教以后在 `Readme` 里面加入要阅读 `Table.java` 这么一条。

### 2.2 加入错误恢复

这个部分我认为还是有一定的难度的。我采用了 `Readme` 中说明的方法。即遇到错误之后, 就往后查找, 看是否属于 `Begin` 与 `End` 集合的并。如果属于 `Begin`, 那么就恢复分析。如果属于 `End` 集合, 那么就舍弃这个正在分析的, 并且返回 `null`。下面稍微详细的说明一下实现的方法。

首先, 在错误恢复中, 由于 `End` 集合是 `parser` 函数的第二个参数 `follow` 和 `LL(1)` 文法中 `Follow` 集合的并, 所以要不断得更新这个 `parser` 的第二个参数 `follow`。每次分析时, 如果该非终结符的 `Follow` 集合不为空, 那么把它全部加到 `follow` 里面, 并且在之后递归地将这个 `follow` 参数传递下去。而

递归下去的结果就是这个 End 集合成为了所有父节点的 End 集合的并集。

当遇到询问的结果为空的情况时。就按照规则进行分析即可。但是需要注意的是，如果对与当前的 lookahead 出现了语法错误，那么需要先判断这个 lookahead 是否出现 End 集合里面。这个地方容易出现一些错误。

之后在递归调用的地方，如果有一个部分返回了 null，那么在该节点就返回 null，并且不进行 act 等其他操作，不然会出现异常。用这种方法，按照 Readme 中的指示，便可以通过所有的测试样例中的语法错误的测试。

## 2.3 加入自定义语法

这个部分基本与 PA1-A 中的相同。这次的 PA 中的 parser.spec 语法与之前的 parser.y 基本相同，但是有一些不同的地方。不同的地方仿照 parser.spec 中的已经给出的代码框架即可写出。但是需要注意的是优先级的处理。由于复数的三个运算符有最高的优先级，所以不能将他们直接视为一元运算符。我的做法是单独加入了一个新的 Oper8 来表示这三个复数运算符，之后在 Expr9 中加入相应的代码，使得这三个运算符有最高的优先级。

## 3 总结与收获

通过该次编程作业，更好的理解了自顶向下的分析方法，同时也对 LL(1) 文法有了更加深入的体会，因为在写 parser.spec 文件的时候，虽然加入的部分大体上和之前的 parser.y 相同，但是框架部分还是有较大的改动。阅读框架代码，让我加深了对 LL(1) 文法的一些理解。同时，自己实现一个简单的错误恢复机制，可以加深对 First,Follow 以及自顶向下分析方法的理

## 4 思考题

### 4.1

允许 else 语句为空的情况，的确使得语法不是严格的 LL(1) 文法。但是构造工具采用了优先级处理的方法，是的 else 语句的优先级比/\*empty\*/语句的优先级要高，这样总是会先进行 else 语句的展开。

下面是一个 decaf 小程序的例子。

---

```
class Main {
    static int main() {
        if (1)
            if (2)
                return 0;
        else
            return 1;
        return 2;
    }
}
```

---

下面是分析出的结果

---

```
program
    class Main <empty>
        static func main inttype
            formals
            stmtblock
                if
                    intconst 1
                    if
                        intconst 2
                        return
                            intconst 0
                    else
                        return
                            intconst 1
                return
                    intconst 2
```

---

可以看出，当在处理内层的 if-else 语句时，遇到了 else，则由优先级高的进行展开，于是将 else 语句和内层的 if 语句放在了一块。

## 4.2

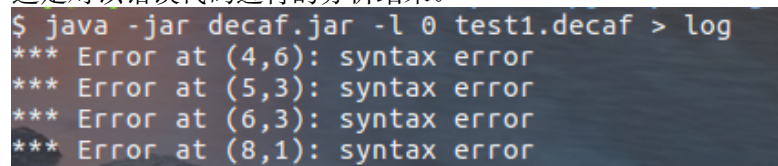
下面是一段错误代码

---

```
class Main {  
    static int main() {  
        return case(1) {  
            1:)4;  
            default:5;  
        };  
    }  
}
```

---

这是对该错误代码进行的分析结果。



```
$ java -jar decaf.jar -l 0 test1.decaf > log  
*** Error at (4,6): syntax error  
*** Error at (5,3): syntax error  
*** Error at (6,3): syntax error  
*** Error at (8,1): syntax error
```

这是因为在自顶向下分析的过程中，‘)’ 符号基本处于所有表达式的 End 集合中。由于 case 语句是一个表达式，那么 case 也就是一个 Expr，而且 case 也为所有其他与 case 相关的表达式的父节点。所以当遇到 ‘)’ 时，由于处于 End 集合中，会跳出所有与 Expr 判断有关的，尤其是会跳出 ACaseExpr, DefaultExpr, 以及 CaseExpr 这种非终结符，从而造成其它的误报。