# 编译原理PA1-B实验报告

计52 周京汉2015011245

## 一、实验任务完成

### 1、增添LL(1)对应文法

#### Lexer.java:

直接复用上次PA1-A的代码，将上次的程序复制过来。

#### SemValue.java:

依照上次的代码添加对应的list和对象，用于spec文件使用。

#### BaseLexer.java:

按照第一次作业的复数的错误处理情况写对于复数Complex的识别和整数过大的错误处理，即可完成本次作业中的对于int过大的问题的处理。

#### Parser.spec:

在这次的语法分析器中加入LL（1）文法。首先将全部token都在最开始声明出来，以对应之前Lexer.java中的声明的token。

完成Super，copy，printcomp，作为Expr中的最高优先级，在Exper9中增加判断。

完成对于虚数进行操作的operator:@,#和$。作为operator，对应最高的优先级，但是与之前的oper7重合的话会出现优先级判断的问题，因此，新添加了oper8，来创建最新的优先级：

```
Oper8           :   RE
                        {
                        $$.counter = Tree.RE;
                            $$.loc = $1.loc;
                        }
                |   IM
                        {
                        $$.counter = Tree.IM;
                            $$.loc = $1.loc;
                        }
                |   COMPCAST
                        {
                            $$.counter = Tree.COMPCAST;
                            $$.loc = $1.loc;
                        }
                ;
```

然后在根据Expr9的写法，写出中间的优先级的expr，表示为expr8_9。因此最终对于这三个运算符的优先级计算为最高级别。

对于Case和Do的书写，起先完全按照上一次那么写会出现不是LL（1）文法，从而出现error。因此，将其调换顺序之后，即可消除冲突，如下为一个样例：

```
DoStmtList      :   DOBLOCK DoStmt DoStmtList
                    {
                        $$.doeslist = new ArrayList<Tree.Do>();
                            $$.doeslist.add($2.does);
                            if ($3.doeslist != null) {
                                $$.doeslist.addAll($3.doeslist);
                            }
                    }
                |   /* empty */
                    {
                        $$.doeslist = new ArrayList<Tree.Do>();
                    }
                ;
```

按照如上的顺序，将中间分隔的符号放在最前面进行判断，即可消除二义性。

## 2、错误修复功能

按照readme中写的，先获取Begin与End集合来获取First和Follow集合。然后再对符号进行判定。在act(actionId, params);函数处需要进行try判定，因为有可能会遇见程序的结尾，就会报出错误，增加了try之后，在结尾处就返回Null就可以正确的报错了。

# 二、思考题

## 1、关于if和else的问题

我认为，问题的关键，就在于else与empty的优先级。因为else的优先级会高于empty，因此，每一个if在后面进行匹配的时候，就会自动先匹配最近的那个else，而不是empty，因此最终的匹配结果是固定的，就不会出现二义性。下面的这个代码可以展现出这个问题：

```
class Main {
    static void main() {
        int a;
        if(a>66)
            a = 30;
        if (a < 30)
        {
            int b;
            b = 40;
        }
        else
        {
            int b;
            b = 50;
        }
    }
}
```

上面这段代码中间有两个if，但是却只有一个else，在最终判定的时候，这个else会跟后面的if匹配，而前面的if会和empty匹配，最终编译出来的程序如下：

```
program
    class Main <empty>
        static func main voidtype
            formals
            stmtblock
                vardef a inttype
                if
                    gtr
                        varref a
                        intconst 66
                    assign
                        varref a
                        intconst 30
                if
                    les
                        varref a
                        intconst 30
                    stmtblock
                        vardef b inttype
                        assign
                            varref b
                            intconst 40
                else
                    stmtblock
                        vardef b inttype
                        assign
                            varref b
                            intconst 50
```

最终的编译结果符合预期。

## 2、错误报错

对于错误误报，我认为就是对于一个已经出现的错，应该是报这里出错，而不是，报后面的错。因此，我认为，在我们的方法之中，存在着一些漏洞，可能出现漏报。比如对于一些判定，如果缺少，或增加了部分的括号，分号，就会将一些语句从一个类型，变成判定另一个类型，因此报出的错就是另一个类型的错误了。

对于如下这个样例，我去掉了最前面的大括号，最终导致后面的判定发生变化，导致误报。

```
class Main
    static void main() {
        return case (n) {
            1: "Monday";
            default: "miao";
        };
    }
}
```

最终报出的错误为：

```
*** Error at (2,5): syntax error
*** Error at (3,9): syntax error
*** Error at (6,10): syntax error
```

很明显，本来只有这一个错误，但是却报出了这么多的后面的错误，并且没有一个相关的。

这是因为在去掉大括号之后，后面的static无法match正确的token，因此报错；而后面大括号match后，后面的return又无法满足class中的要求，又无法match；因此也无法识别最后的一个分号，因为根本没有识别出这个return，所以再次报错。

# 三、实验感悟

本次实验将原来的yacc工具化成了自己手写的一套工具，需要自己去手写规定优先级，以及需要满足LL（1）文法来消除二义性，让我更加深刻的理解了优先级的意义和二义性的影响。