Decaf PA 1-B

王世因 2016011246

1. 本阶段工作

步骤一:阅读 LL(1)分析算法的实现

第一天拿到作业代码的时候，我发现`parser.spec`和PA1-A的`parser.y`十分相似，于是就直接从那里粘贴了很多代码过来，然后遇到了错误。经过仔细看实验报告和ppt，我从现成代码入手看LL(1)分析的语法规则，照猫画虎开始写新增加的特性。

在增加条件卫士特性的时候，我按照PA1A的习惯改代码后，总是报错，而且是以下这种情况，让我十分困惑。后来发现每次修改`parser.spec`后需要`clean`一下再`build`。

```
*** Error at (1,1): syntax error
```

步骤二：增加错误恢复功能

在刚开始运行初始程序的时候，我看到报错NullPointerError，不知道怎么入手。参加了班级的编译原理讲座后理解了大致的写法，之后就根据报告中的算法描述写了。这样我们可以在遇到一个报错的时候，继续继续分析之后的语法，直到遇到了语法的终结符退出这个分支。

```
Set<Integer> begin = beginSet(symbol);
Set<Integer> end = followSet(symbol);
end.addAll(follow);

if (!begin.contains(lookahead)) {
    error();
    while(true) {
        if (begin.contains(lookahead)) {
            return parse(symbol, follow);
        }
        else if (end.contains(lookahead)) {
            return null;
        }
        lookahead = lex(); // get the next input
    }
}
```

我在样例中报错`stackoverflow`，是因为`parser.java`代码中的纠错部分陷入了死循环。正确的写法是每次把当前节点的follow set加入到总的follow set中，包含它所有的父节点的follow set，这样我们可以在遇到文件的结尾的时候退出循环。

步骤三：增加新特性对应的 LL(1) 文法

`foreach`、`scopy`和`sealed`语句因为First集合独特，可以按照上一次的作业那样直接写入。需要额外判断的是`guarded`和`array`。这里我在写完书面作业后，对LL(1)文法有了深入的了解，选择了把一个语句从不同处拆分成小语句的形式进行分析。比如条件卫士语句和普通的IF语句是在括号处开始不一样的，所以我就在这里新建一个node造成两个First集合不同的分支。在上一个PA中，我写的都是左递归，在这次的作业中都改成了右递归。

```
IfStmt          :   IF IfContent ;

IfContent       :   '(' Expr ')' Stmt ElseClause
                |   '{' IfGContent ;

IfGContent      :   IfG IfBranch '}'
                |   '}' ;

IfG             :   Expr ':' Stmt;

IfBranch        :   DIVIDER IfG IfBranch
                |   /* empty */ ;
```

数组的处理是所有样例中最困难的，我根据“Specification on New Features of Decaf”中定义的符号优先级，照猫画虎，增加了`Oper31`和`Oper32`。

```
Oper31          :   ARRAY_CONCAT
                    {
                        $$.counter = Tree.ARRAYCONCAT;
                        $$.loc = $1.loc;
                    }
                ;

Oper32          :   ARRAY_REPEAT
                    {
                        $$.counter = Tree.ARRAYREPEAT;
                        $$.loc = $1.loc;
                    }
                ;
```

```
ExprT3          :   Oper3 Expr31 ExprT3;
Expr31          :   Expr32 ExprT31;
ExprT31         :   Oper31 Expr32 ExprT31;
Expr32          :   Expr4 ExprT32;
ExprT32         :   Oper32 Expr4 ExprT32;
```

2. if 语句的 else 分支为空冲突处理

考察IF ELSE语句：

```
S -> if C then S E
E -> else S | <empty>
```

因为$PS(E \rightarrow else \quad S) \cap PS(E \rightarrow < empty >) = \{else\}$，所以这不是一个LL(1)文法。在文档中我们可以采用人为设定优先级的方式来解决，这一知识点也在书面作业中涉及了。例如下面的语句有两种可能：

```
if (true) if (false) Print("T"); else Print("F");
```

```
if (true){
    if (false) Print("T");
    else Print("F");
}

if (true){
    if (false) Print("T");
}
else Print("F");
```

通过设置不同的优先级，我们可以得到上面的任何一种情况：

```
IfStmt → if Expr Stmt else Stmt | if Expr Stmt (A)
IfStmt → if Expr Stmt | if Expr Stmt else Stmt (B)
```

3. 数组comprehension表达式文法

```
Expr9           :   LISTFORL Expr FOR IDENTIFIER IN Expr AfterList
                |   Constant
                ......

AfterList       :   IF Expr LISTFORR
                |   LISTFORR
                ;

Constant        :   LITERAL
                |   NULL
                |   '[' ARRAY
                ;
```

因为Constant的First集合中有`[`，而且因为Array中允许的常量也是Expr中的一部分，所以如果要把这两个混淆解除的话会花大量的时间和精力，十分困难。

4. 误报

在下面这个例子中，我的程序会检测到两个错误，一个是`-`一个是`]`。第一个是对的，但是第二个错误其实是正确的。这是因为根据实验指南写的代码中，一旦发现错误就会直接返回`null`结束搜索，在这个过程中会遗忘之前的负号。因为`-`在EndSet中，所以程序在这里终止，忘记了之前曾经到访过`[`，因而造成匹配错误。

```
class Main {
    static void main() {
        int[] xs;
        xs = [1,0,-1]; // -1 is an expression, not a constant!
    }
}
```

实验总结和体会

这次的PA和上次的PA目标是一致的，只不过不能用上次的YACC工具了，需要自己写LL(1)文法进行转化。在这个过程中，我对于理论部分有了深入的了解，也进一步锻炼了自己的代码能力。