PA1-B 实验报告

姓名：杨雅儒　　班级：无 73　　学号：2017011071

一．本阶段工作

本阶段工作主要分为两部分：**错误恢复**部分和**语法分析**部分。

1. 错误恢复部分

这部分的策略同步骤二中提出的错误处理方法，这里还是作一下归纳和介绍：

在 parse 方法中，对于当前 symbol 这个即将进行匹配的非终结符，首先这个非终结符的 follow 集合（代码中为 followNow）加入到 follow 集合中。然后筛掉所有不在它的 begin 集合也不在 follow 集合中的 lookahead（如果存在被筛掉的则报错）。

对于当前的 lookahead，如果在 begin 集合中，则可以恢复匹配该非终结符【恢复匹配时，同正常匹配一样，不用特地考虑匹配失败，因为会在递归中处理】；如果在 follow 集合中，我们认为该非终结符已经匹配失败了，所以直接 return null。

另外修改了 Parser.java 的 parse 方法，并且加入了一个 Boolean 变量 fail，当有语法错误时不再触发 act 方法——这是调试时发现的一个 bug，否则会出现空指针错误。

另外注意两点：

* return 时要从 follow 中删掉由该非终结符新添进去的
* 匹配终结符时如果失败，并不会消耗掉该终结符，这样才可以回溯到正确处。

这里贴出代码如下:

```
93    private SemValue parse(int symbol, Set<Integer> follow) {
94        // add new follow
95        Set<Integer> newAdded = new HashSet<Integer> () ;
96        Set<Integer> begin = beginSet(symbol) ;
97        Set<Integer> followNow = followSet(symbol) ;
98      //  boolean fail = false ;
99        for(Integer t : followNow) {
100           if(!follow.contains(t)) {
101               follow.add(t) ;
102               newAdded.add(t) ;
103           }
104       }
105       //debugInfo("lookahead: "+name(lookahead)+  "   now: " + name(symbol));
106       if(!begin.contains(lookahead)) {
107       //  debugInfo("LALALA  "+ name(lookahead) + "  now: " + name(symbol));
108           error() ;
109           while(!begin.contains(lookahead) && !follow.contains(lookahead)) {
110               lookahead=lex() ;
111           }
112           if(!begin.contains(lookahead) && follow.contains(lookahead)) {   //fail to recover
113               follow.removeAll(newAdded) ;
114               return null ;
115           }
116       }
117
118       Map.Entry<Integer, List<Integer>> result = query(symbol, lookahead); // get production by lookahead symbol
119       int actionId = result.getKey(); // get user-defined action
120
121       List<Integer> right = result.getValue(); // right-hand side of production
122       int length = right.size();
123       SemValue[] params = new SemValue[length + 1];
124
125
126       for (int i = 0; i < length; i++) { // parse right-hand side symbols one by one
127           int term = right.get(i);
128           params[i + 1] = isNonTerminal(term)
129                   ? parse(term, follow) // for non terminals: recursively parse it
130                   : matchToken(term) // for terminals: match token
131                   ;
132           if(!isNonTerminal(term)) {
133               debugInfo("matchToken"+ name(term)+ " in "+name(symbol));
134           }
135           if(params[i + 1] == null) fail=true ;
136       }
137       params[0] = new SemValue(); // initialize return value
138       if(!fail) act(actionId, params); // do user-defined action
139       follow.removeAll(newAdded);
140       return params[0];
141   }
```

2. 语法分析部分

- **特性 1**

加入 Scopy。

同 PA1_A 的特性 1。Parser.spec 中新增对应的 token 以及文法，Tree.java 中新增 Scopy 类型节点，其中加入一个字符串变量以及一个 Expr 类变量，Tree.java 中新增 SCOPY 常量。

- **特性 2**

加入 Sealed。

同 PA_1A 的特性 2。Parser.spec 中新增 token 和 SealedOpt 以及其对应的文法，通过 SealedOpt 返回的代表码的 code 判定是否有 sealed 参数，若是则将 ClassDef 中新增的 isSealed 变量赋值为 true。并且在 tree.java 中修改了 ClassDef 节点，向其中加入 boolean 变量 isSealed，并修改了构造函数。

（下面的特性中与 PA1-A 一样的就一笔带过不再赘述了，**主要叙述差别之处**）

- **特性 3**

加入条件卫士语句。

在 PA_1A 的特性 3 的基础上，由于发生了冲突，将 IF 作为左公因子提出，剩下的以 IfSuf 代替。

并且由于 PA_1A 需要尽早规约，所以会出现直接左递归和其它一些 conflict。在这里，IfBranches 的产生式中将右侧的两个符号互换位置，在 GuardedCont 中同样互换位置。另外注意一下输出顺序即可。

- 特性 4

支持简单的自动类型推导。

只用注意一下与 PA_1A 语法上有区别。在 Tree.java 中的 Ident 节点上新增 isVar 变量并修改构造函数，修改其 printTo 函数，其余就是正常的加入 token 和相应常量等。

- **特性 5**

(1) 数组常量。

与 PA1-A 相比需要对 parser.spec 进行修改以改为 LL(1)文法，另外新增常量和 token 同 PA1-A，还有在 Tree.java 中新增 ArrayConstant 节点，并在 SemValue.java 中增加该类变量 acons。

(2)、(3) 数组初始化常量表达式和数组拼接表达式

形如 E%%n 和 E1++E2。

这部分主要需要处理的就是优先级与结合性，做法如下：

在 Expr4 和 Expr5 之间依次插入 ExprDAdd 和 ExprDMod，以及对应的 ExprDAddT 和 ExprDModT，另外再加入 OperDAdd 和 OperDMod。对于 ExprDMod 同其它的 Expr 类似都是左结合的，而对于 ExprDAdd 是右结合，所以在 ExprDAdd 的产生式中应当从右向左使用 for 循环构建 AST。这样做我们的 AST 就成功地构建了。

另外还需要在 Binary 节点上增加对两个操作符的输出。另外其它常量的添加不再赘述。

(4) 子数组表达式

原本做法：
由于 ExprT8 处有数组下标表达式，所以将其括号中的 Expr 后加上一个 SubArrayExpr，这个非终结符的产生式有空和 ':' Expr。对于 Tree.java 需要新增节点 ArrayRange，另外新增 token 和相应常量。

但是这样写会和(5)中的部分导致一些 conflict，所以在做(5)的时候进行了修正。

(5) 支持 default

调试地有些久了。

主要工作是在 ExprT8 中增加了内容，一并将上面的(4)进行了整体修改。主要是利用提取的方式，首先判断是子数组表达式还是普通数组（通过判断有无冒号确定），然后对于普通数组则考虑后面是否有 default。对于普通数组则还可以继续使用 ExprT8 跟随其后，而对于 default 则后面不再有东西。实际上这样的话，冲突就在一次次讨论中被细化而解决了。

(6) comprehension 表达式

基本上同 PA1-A，同样加入了 IFExpr 非终结符，以及加入了 CompArrayExpr 节点进行处理。唯一的区别就是将'['和']'改为了 LOR 和 ROR 两个词法符号。

(7) 数组迭代语句

也基本上完全与 PA1-A 相同。不再赘述。

二．

可以发现，在预测集合中，发生冲突的部分优先处理了 else 语句。而在给出的工具指示中也写明：“The default setting is unstrict mode. When conflicts appear, we assign higher priority to former defined productions”

也就是说，对于 E：else S | /* empty */ ，在非严格模式下，出现冲突时我们优先选择 else S 对应的产生式，这样就可以处理了。

例如出现如下代码时：

```
1  class Main {
2      static void main() {
3          if(a){
4          }
5          if(b){
6          }
7          else {
8          }
9      }
10 }
```

则会由于优先级，使得 else 对应后者。

三． 为什么把原先的 comprehension 表达式文法改写成 LL(1)比较困难？

Emm 其实在我的实现里面似乎不太难改，也可能是因为我忽略了一些问题，总之在先提取出 ‘[‘ 和 Expr 之后，我使用了如下方式：

```
ExprT8_2        :   DEFAULT Expr9
                    {
                        $$.expr=$2.expr ;
                        $$.myType=2 ;
                    }
                |   ExprT8
                    {
                        $$.vec=$1.vec;
                    }
                ;

ExprT8_1        :   ':' Expr ']' ExprT8
                    {
                        $$.myType = 1 ;
                        $$.expr = $2.expr ;
                    }
                |   ']' ExprT8_2
                    {
                        $$.vec = $2.vec ;
                        $$.expr = $2.expr ;
                        if($2.myType==2) $$.myType=3 ;
                        else $$.myType=4 ;
                    }
                |   FOR IDENTIFIER IN Expr IfExpr ']'
                ;
```

测试时并没有产生其它警告。

四．无论何种错误处理方法，都无法完全避免误报的问题。请举出一个语法错误的 Decaf 程序例子， 用你实现的 Parser 进行语法分析会带来误报。 根据你用的错误处理方法，这些误报为什么会产生？

例如：

```
1 class Main {
2     static void main() {
3         A = [-1,2,3] ;
4     }
5 }
```

报错：

```
*** Error at (3,8): syntax error
*** Error at (3,10): syntax error
*** Error at (3,12): syntax error
*** Error at (3,13): syntax error
*** Error at (3,14): syntax error
```

分析如下：

在语法分析过程中，当进入 = 后面时即进入 Expr 的匹配，到负号处时，由于负号不应该出现，所以正常报错。但是此时回到 Oper5 时将负号成功匹配了，接下来则进入了 ExprT5: Oper5 Expr6 ExprT5 的匹配，在将 1 作为 Expr6 直到 Constant 匹配完毕之后，接下来的逗号无法匹配，从而导致了误报。