# PA1-B实验报告

###### 计61 潘庆霖 2016011388

### 1、简要描述

#### （1）实现错误恢复

* 实现了实验要求中给出的思路，代码如下

private SemValue parse(int symbol, Set<Integer> follow) {  
 Set<Integer> begin = beginSet(symbol);  
 Set<Integer> end = followSet(symbol);  
 Set<Integer> allFollow = new HashSet<>(follow);  
 allFollow.addAll(end);  
  
 if (!begin.contains(lookahead)) {  
 error();  
 while (!begin.contains(lookahead) &&!allFollow.contains(lookahead))  
 lookahead = lex();  
 if (!begin.contains(lookahead) && allFollow.contains(lookahead))  
 return null;  
 }  
   
 Pair<Integer, List<Integer>> result = query(symbol,lookahead);  
 int actionId = result.getKey();   
   
 List<Integer> right = result.getValue();   
 int length = right.size();  
 SemValue[] params = new SemValue[length + 1];  
   
 boolean isError = false;  
   
 for (int i = 0; i < length; i++) {   
 int term = right.get(i);  
 params[i + 1] = isNonTerminal(term)  
 ? parse(term, allFollow)   
 : matchToken(term)   
 ;  
 if (params[i + 1] == null)  
 isError = true;  
 }  
   
 if (isError)  
 return null;  
   
 params[0] = new SemValue();   
 act(actionId, params);   
 return params[0];  
 }

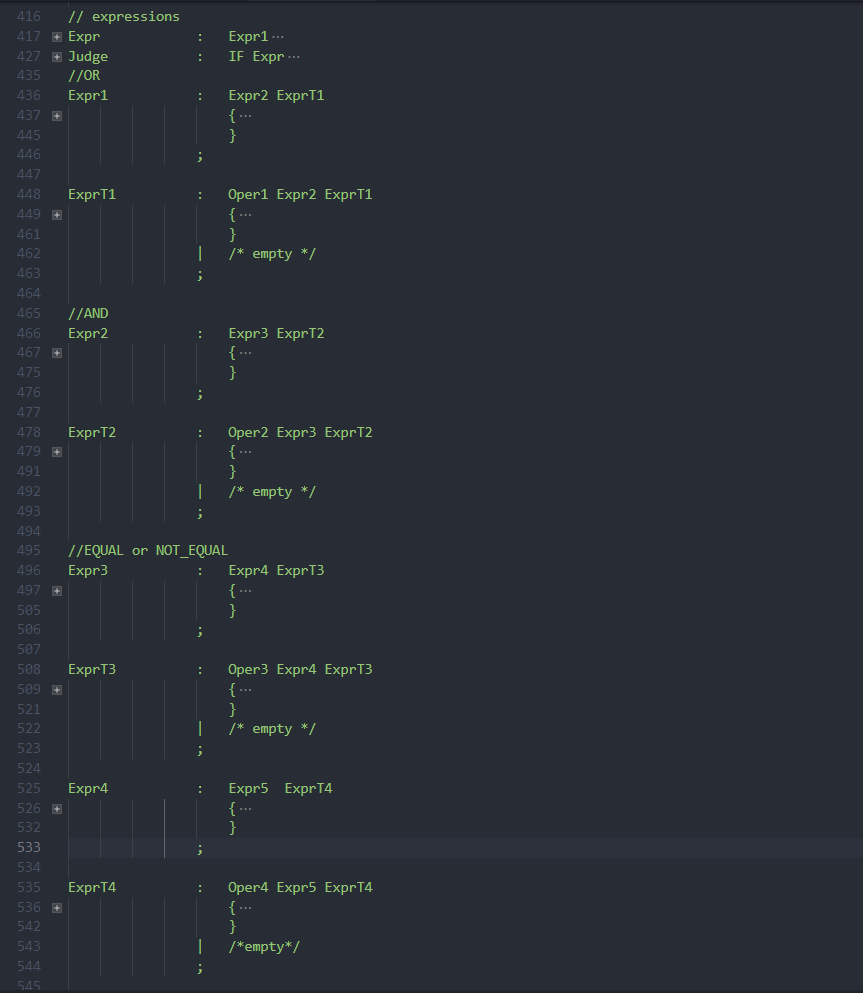
#### （2）添加新增语法特性

##### 根据所需要的关键词和操作符，Parse.spec中新增的终结符如下

%tokens  
':'   
SCOPY GUARD SEALED MOMO IN PLUSPLUS  
VAR FOREACH DEFAULT LeftBra RightBra

其中，LeftBra、RightBra分别代表'[|'、'|]'。

* 对象浅复制语句：
  + 语句格式：scopy(id, E)
  + 对应语法：
  + Stmt : VariableDef  
     | SimpleStmt ';'  
     | IfStmt  
     | WhileStmt  
     | ForStmt  
     | ReturnStmt ';'  
     | PrintStmt ';'  
     | BreakStmt ';'  
     | StmtBlock  
     | OCStmt ';'  
     | ForeachStmt  
     ;  
    OCStmt : SCOPY '(' IDENTIFIER ',' Expr ')' ;
* sealed关键词修饰，使类无法被继承
  + 语句格式：sealed class identifier < extends identifier > { field\* }
  + 对应语法：
  + ClassDef : CLASS IDENTIFIER ExtendsClause '{' FieldList '}'  
     | SEALED CLASS IDENTIFIER ExtendsClause '{' FieldList'}'   
     ;
* 串行条件卫士语句：
  + 语句格式：if {E1:S1 ||| E2:S2 ||| ...}
  + 对应语法：
  + Stmt : IfStmt | ... ;  
    IfStmt : IF IfHelper ;  
    IfHelper : Expr '(' Stmt ')' ElseClause | '{' IfBranchList '}' ;  
    IfBranchList : IfSubStmt IfBranches | /\*empty\*/ ;  
    IfBranches : GUARD IfSubStmt IfBranches | /\*empty\*/ ;  
    IfSubStmt : Expr : Stmt ;
  + 实现过程
    - 思路分析:
    - 对于这一特性的添加，一开始没有考虑到已经存在的IfStmt终结符，直接添加了PA1-A的对应部分，产生了错误，后来通过观察发现，关键词IF不能作为产生式右侧的开头，所以想到了建立一个IfHelper，用来综合原有的if语句与新添加的条件卫士特性。
* 支持简单的自动类型推导
  + 语句格式： var x = y；
  + 对应语法：
    - Stmt ::= SimpleStmt
    - SimpleStmt ::= var identifier = Expr | ...
* 数组常量表达式
  + 语句格式： 形如 [c1, c2, c3, ...]
  + 对应文法：
  + Constant : ArrayConstant | ... ;  
    ArrayConstant : '[' Constants ']' ;  
    Constants : Constant ArrConstList | /\* empty \*/ ;  
    ArrConstList : ',' Constant ArrConstList | /\* empty \*/ ;
* 数组常量化表达式、数组拼接表达式的实现
  + 对应的语句：
    - 常量化表达式： E %% n
    - 拼接表达式： E ++ E
  + 对应文法：
    - 常量化表达式：
    - Expr4 : Expr5 ExprT4 ;  
      ExprT4 : PLUSPLUS Expr5 ExprT4 | /\* empty \*/ ;
    - 拼接表达式：
    - Expr5 : Expr6 ExprT5 ;  
      ExprT5 : MOMO Expr6 ExprT5 | /\* empty \*/ ;
  + 实现过程：
    - 实现时将这两个操作符同时考虑，注意到的点有关于操作符优先级的差异以及结合特性的分析。
    - 观察发现，本次实验与上次不同的一点在于，LL（1）文法的手动实现中，操作符优先级的前后关系体现在不同的Expr层次上，如下图



* + - 如上图中结构层次所示，不同优先级的操作符被有序地安插在多层的Expr代码结构中
    - 通过查看spec*new*features中的优先级定义，我在原先定义的Oper1-Oper7中加入了%% ++ 两个操作符，添加的位置为原来的Oper3与Oper4之间且%%优先级高于++，重新命名得到Oper1-Oper9。之后仿照原有的文法风格，将原有的Expr1-Expr9、ExprT1-ExprT11，拓展为Expr1-Expr11、ExprT1-ExprT11。
    - 然后是关于左结合与右结合的区分与实现。对于右结合的++，正好可以从右到左依次构造语法节点，较为自然地实现了右结合的特点。而对于左结合的%%，我才用的方法是，从对应文法部分的递归过程中，由下至上返回一个有序列表，直到到达顶层，再对该列表从左到右构建语法节点，达到左结合的效果。
* 取子数组表达式、数组下标动态访问表达式
  + 语句格式：
    - 取子数组表达式：E [ E1 : E2]
    - 数组下标动态访问表达式：E [ E1 ] default E`
  + 对应语法：
    - 同时考虑这两个特性的主要原因是，如果分开实现，会影响到带 [ 的原有产生式，导致LL(1)文法特性被破坏，因此应该修改原有文法，使原有文法兼容这两个新的特性，
      * 旧的文法如下:
      * Expr10 : Expr11 ExprT10 ;  
        ExprT10 : '[' Expr ']' ExprT10 | '.' IDENTIFIER AfterIdentExpr ExprT10 | ;
      * 修改文法为：
      * Expr10 : Expr11 ExprT10 ;  
        ExprT10 : '[' Expr ']' ExprT10   
         | '.' IDENTIFIER AfterIdentExpr ExprT10   
         | DEFAULT Expr11  
         |   
         ;
* Python风格的comprehension表达式、foreach数组迭代语句：
  + 语句格式：
    - comprehension表达式：[| E' for x in E < if B > |]
    - foreach数组迭代： foreach ( var x in E while B ) 或者
    - foreach ( Type x in E while B)
  + 对应文法：
    - comprehension表达式：
    - Expr : Expr1|LeftBra Expr FOR IDENTIFIER IN Expr Judge RightBra ;  
      Judge : IF Expe ;
    - foreach数组迭代：
    - Stmt : ForeachStmt ;  
      ForeachStmt : FOREACH '(' BoundVariable IN Expr WhileJudge ')' Stmt ;  
      BoundVariable : VAR IDENTIFIER | Type IDENTIFIER ;

### 2、 if语句的else分支冲突分析

#### 问题描述

假定有两个产生式A——>α和A——>β以这种方式产生冲突，则具体表现为：PS（A——>α)与PS(A——>β)有交集，设这一交集为D。那么，当某一时刻，当前栈顶的非终结符为A，向前查看一个非终结符得到的结果为lookahead，当lookahead包含在D中时，将无法确定该使用哪一个产生式，因为两个产生式都是可能的。

#### 本工具的解决方法

通过定义优先级来区分产生式。在这次实验的代码中，如上文分析Expr的过程可见，我们通过定义的先后顺序来确定不同产生式之间的优先级，产生式低的优先级只有当产生式高的优先级无法匹配的时候，才会被匹配，因此，就上面的例子来说，可以通过定义优先级使得：PS(A——>α) = PS（A——>β）-D。这样就可以使两个预测集合之间没有交集，预测表的表项唯一。

#### 实例检验

构造下列实例

class Main{  
 void test(){  
 if(true)  
 if(false)  
 x = 0;  
 else  
 x = 1;  
 }  
}

得到的分析结果

program  
 class Main <empty>  
 func test voidtype  
 formals  
 stmtblock  
 if  
 boolconst true  
 if  
 boolconst false  
 assign  
 varref x  
 intconst 0  
 else  
 assign  
 varref x  
 intconst 1

### 3、comprehension表达式的有关思考

因为在当前的文法中，Expr可以推导得到数组常量ArrayConstant,而数组常量的第一个非终结符即为'['，这就导致了如果想要实现这一文法：

Expr ::= [ Expr FOR identifier IN Expr < if B >]

就必须处理产生式Expr --> Expr1 ; 与 Expr --> '[' Expr FOR identifier IN Expr < if B > ']' 的冲突。这一冲突主要原因在于Expr最终可能推导出数组常量，所以会变成处理Constant和Expr的右侧表达式左公因子的问题，修改的复杂度较高。

### 4、语法错误的误报例子

例子

class TestClass {   
 void err(){   
 if(;)   
 x = 0;   
 }   
}

运行结果

\*\*\* Error at (3,12): syntax error  
\*\*\* Error at (3,13): syntax error

结果分析：

对于这个例子，多报了（3，13）的错误，主要是因为：

在匹配IF '('后面的表达式的时候，拿到的lookahead是';',并不包含在非终结符Expr的Begin集合,却包含在Expr的End集合中，因此报（3，12）的错误之后，parse（Expr，follow）返回null；

接下去尝试匹配的是')',终结符')'匹配';'失败又会报错一次。再然后等到开始匹配ElseClause的时候，才略去了';'、')'、'x'、'='、'0'、';'。