编译原理PA1-B

实验过程

改写为LL(1)文法

改动涉及三处文法。

1. 抽象类和抽象方法。这一部分很简单，只需要向`ClassDef`与`FieldList`添加产生式即可（基本照搬`Decaf.jacc`）。因为语法规定`abstract`关键字必须在最前，而且此关键字在之前的语法规则中从未出现过，因此新添加的产生式自动符合LL(1)文法。
2. `var`语句。与上面类似，对于非终结符`SimpleStmt`添加一条产生式，产生式内容和代码部分同样可以照搬`Decaf.jacc`。
3. Lambda函数类型、Lambda函数定义和新的函数调用语法。观察已有的文法，注意到`Type : AtomType ArrayType`，其中`ArrayType`已经规定了数组类型的语法，因此Lambda类型的语法可以在此基础上进行扩展。我们把它改名为`TypeFollow`，更改后的产生式应该为：

   ```
   TypeFollow : '[' ']' TypeFollow
              | '(' TypeList ')' TypeFollow
              | <empty>
   ```

   我们可以举出有一点点复杂的例子：`int()[](int)`，这是一个Lambda函数类型，其参数类型为`int`，返回类型也是一个Lambda函数（此函数再返回一个`int`类型）。显然，依次运用第(2, 1, 2, 3)条产生规则可以推导出我们想要的这个例子。哦对了，我们还缺一个`TypeList`，不过它的LL(1)文法产生式也比较好写，这里略去。

   然后改写函数定义的语法。我们规定非终结符`ExprLambda`产生一个Lambda函数定义，形如`ExprLambda : FUN '(' VarList ')' Block | FUN '(' VarList ')' '=>' Expr`。注意到有公共前缀，于是如下改写为LL(1)文法：

   ```
   ExprLambda : FUN '(' VarList ')' LambdaBody
   LambdaBody : DOUBLE_ARROW Expr | Block
   ```

   最后改写函数调用的文法。查阅文档得知，新文法中`(`与`)`、`[`与`]`、`.`的优先级相同，因此不妨把函数调用的文法也写进非终结符`ExprT8`的生成规则中，然后扩展产生式`Expr8 : Expr9 ExprT8`下面那段代码。注意去掉原来的版本中同样是函数调用的“重叠”部分文法，在这几处中，函数调用的传参列表前必须是ID（或者是访问对象的字段），于这两处定义把`ExprListOpt`去掉。如果忘记做修改，会在build时候得到一堆Warning。

以上就是文法的主要修改内容。注意生成代码部分的修改，以适合扩展后的Java代码。

错误恢复

错误恢复依照指导上的标准算法实现，其算法结构如下：

1. 判断`token`是否属于$First(Symbol)$（即：查找表`M[Symbol, token]`是否为空）。如果属于，那么可以正常进行分析流程。否则继续步骤2。

2. 重复取出下一个`token`，直到拿到的`token`属于$First(Symbol) \cup End(Symbol)$。对于其中的$End(Symbol)$可以递归地求出：利用`parseSymbol`函数的第二个参数`Set<Integer> follow`传递祖先节点的所有的`Follow`符号，再向`follow`中添加$Follow(Symbol)$。

   如果先取到了$First(Symbol)$以外的字符，那么认为`Symbol`的管辖范围结束，此时结束`Symbol`的递归推导，返回`null`。否则，取到了$First(Symbol)$中的字符，就按照产生新的`Symbol`进行推导，正常进行分析流程。

在原有代码上还需要进行修改，在正常进行语法解析时，需判断`parseSymbol`返回值是否为空，避免递归中下层处理的语法错误导致上层发生运行时错误。文档里没说，需要观察代码知道的一个点时：语法错误提示通过`yyerror(msg)`函数产生。旧有框架没有处理错误重复，需要把指导书中的代码粘贴过来。

问题思考

Q1

> 本阶段框架是如何解决空悬 else (dangling-else) 问题的？

本阶段框架的文法没有明确解决空悬else问题，因此`gradle build`的时候提示Warning，如下：

```
Warning: conflict productions at line 260:
ElseClause -> ELSE Stmt
ElseClause -> <empty>
```

虽然不严格符合LL(1)文法，但是ll1pg隐式地通过给同一非终结符号的不同产生式赋予不同的优先级，解决了这个问题。例如，对于文法：

```
ElseClause : ELSE Stmt | <empty>
```

出现在前面的`Else Stmt`具有更高的优先级，从而解决了冲突。

Q2

> 使用 LL(1) 文法如何描述二元运算符的优先级与结合性？请结合框架中的文法，举例说明。

部分相关产生式如下：

```
Expr1           :   Expr2 ExprT1
ExprT1          :   Op1 Expr2 ExprT1 | \<empty\>

Expr2           :   Expr3 ExprT2
ExprT2          :   Op2 Expr3 ExprT2 | \<empty\>
...
Expr6           :   Expr7 ExprT6
ExprT6          :   Op6 Expr7 ExprT6 | \<empty\>
```

其中二元运算符按照优先级从低到高分属于`Op1`，`Op2`，...，`Op6`，文法保证优先级>=2的运算符能由`Expr2`推出，之后再与`Op1`类运算符结合成为`Expr1`。这样，运算符的优先级得到规定。

本实验中没有右结合的运算符，因此结合性并没有实例可对比佐证。观察文法得知，在解析`Expr2`时，先解析右侧的运算符，一路下来，最左边的操作数在生成的语法树最顶端，那岂不是应该是右结合吗？且慢，这与对应语法规则下代码的处理紧密相连。实际上，`ExprT2`的解析结果是一个从左到右顺序的操作符与操作数（表达式）列表，然后交由`buildBinaryExpr`函数处理，构造左结合的语法树。如果

要实现右结合，需要重载`buildBinaryExpr`函数，由于LL(1)的限制，改写文法以规定结合性是难以做到的。

Q3

> 无论何种错误恢复方法，都无法完全避免误报的问题。 请举出一个具体的 Decaf 程序（显然它要有语法错误），用你实现的错误恢复算法进行语法分析时会带来误报。 并说明该算法为什么无法避免这种误报。

举例如下：

```
class Main {
    static void main() {
        int x = 1;
    { // error: expected '}', got '{'
    void func1() {
        int x = 2;
    }
    void func2() {
        int x = 3;
    }
}
```

只有一处(4, 5)有语法错误，但是语法分析器看起来觉得像内部代码块，导致貌似无关的位置产生了各种错误：

```
*** Error at (5,15): syntax error
*** Error at (5,16): syntax error
*** Error at (6,15): syntax error
*** Error at (8,5): syntax error
```

个人认为，主要原因是一种错误代码有多种可能的正确改正方式，但是LL(1)文法的语法分析器无法综合上下文判断，他会根据眼前的`token`决定采取那种解析式，结果却使用了错误的产生式。显然，人类可以通过观察上下文猜到第四行的`{`是错的，但是让LL(1)文法的分析器选取出这种最优解释绝非易事。

总结

本阶段将语法改写成LL(1)的类型，熟悉了LL(1)文法；同时借助框架熟悉了LL(1)文法分析的过程，并且运用一定的错误恢复。

顺利完成本章作业需要理解LL(1)文法的诸多概念，如$PS(S \to \alpha)$，$Begin(S)$, $Follow(S)$；掌握消除左递归和公因子的方法，并且能阅读代码以在框架基础上加以实现。