语法分析

1、正则文法，按照一定规则顺序匹配。

例子：分析int： IntDeclare()方法

**SimpleASTNode node = null;**

**Token token = tokens.peek(); // 预读**

**if (token != null && token.getType() == TokenType.Int) { // 匹配 Int**

**token = tokens.read(); // 消耗掉 int**

**if (tokens.peek().getType() == TokenType.Identifier) { // 匹配标识符**

**token = tokens.read(); // 消耗掉标识符**

**// 创建当前节点，并把变量名记到 AST 节点的文本值中，**

**// 这里新建一个变量子节点也是可以的**

**node = new SimpleASTNode(ASTNodeType.IntDeclaration, token.getText());**

**token = tokens.peek(); // 预读**

**if (token != null && token.getType() == TokenType.Assignment) {**

**tokens.read(); // 消耗掉等号**

**SimpleASTNode child = additive(tokens); // 匹配一个表达式**

**if (child == null) {**

**throw new Exception("invalide variable initialization, expecting an expression");**

**}**

**else{**

**node.addChild(child);**

**}**

**}**

**} else {**

**throw new Exception("variable name expected");**

**}**

**}**

2、上下文无关语法，允许递归调用。

例子：计算 2+3\*5

**Calculating: AdditiveExp // 计算根节点**

**Calculating: IntLiteral // 计算第一个子节点**

**Result: 2 // 结果是 2**

**Calculating: MulticativeExp // 递归计算第二个子节点**

**Calculating: IntLiteral**

**Result: 3**

**Calculating: IntLiteral**

**Result: 5**

**Result: 15 // 忽略递归的细节，得到结果是 15**

**Result: 17 // 根节点的值是 17**

3、巴科斯范式（BNF）Antlr和Yacc这两个工具都是用这种写法。

写法如下：

**add ::= mul | add + mul**

**mul ::= pri | mul \* pri**

**pri ::= Id | Num | (add)**

很多时候 ::=简化成一个:

**add -> mul (+ mul)\* \*表示能能够重复，等价于通过递归来推导。**

刚刚我们是用加法来推导的乘法，保证了AST中的乘法节点一定会在加法节点的下层，保证了乘法计算优先于加法计算。

像 (>、=、<)放在加法的上面

**exp -> or | or = exp**

**or -> and | or || and**

**and -> equal | and && equal**

**equal -> rel | equal == rel | equal != rel**

**rel -> add | rel > add | rel < add | rel >= add | rel <= add**

**add -> mul | add + mul | add - mul**

**mul -> pri | mul \* pri | mul / pri**

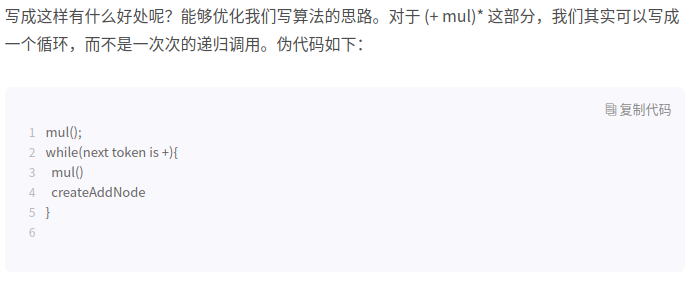
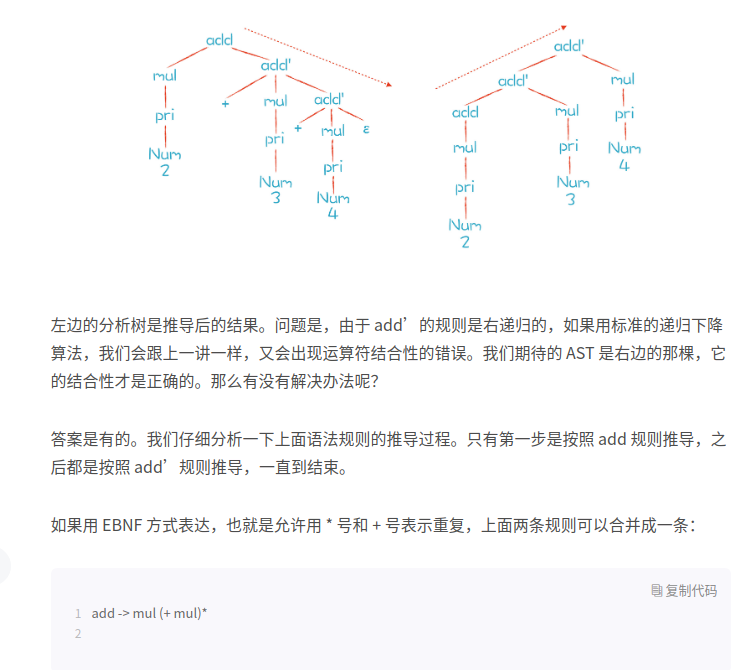
****

4、消除左递归

**add -> mul add'**

**add' -> + mul add' | ε**

****

****

代码如下：

**private SimpleASTNode additive(TokenReader tokens) throws Exception {**

**SimpleASTNode child1 = multiplicative(tokens); // 应用 add 规则**

**SimpleASTNode node = child1;**

**if (child1 != null) {**

**while (true) { // 循环应用 add'**

**Token token = tokens.peek();**

**if (token != null && (token.getType() == TokenType.Plus || token.getType() == TokenType.Minus)) {**

**token = tokens.read(); // 读出加号**

**SimpleASTNode child2 = multiplicative(tokens); // 计算下级节点**

**node = new SimpleASTNode(ASTNodeType.Additive, token.getText());**

**node.addChild(child1); // 注意，新节点在顶层，保证正确的结合性**

**node.addChild(child2);**

**child1 = node;**

**} else {**

**break;**

**}**

**}**

**}**

**return node;**

**}**

结果如下：

**Programm Calculator**

**AdditiveExp +**

**AdditiveExp +**

**AdditiveExp +**

**IntLiteral 2**

**IntLiteral 3**

**IntLiteral 4**

**IntLiteral 5**