Java 枚举

enum：参考TokenType定义

列表

```
// 1. 声明和初始化
List<Token> tokens = new ArrayList<>();
List<StmtNode> statements = new ArrayList<>();

// 2. 添加元素
tokens.add( newToken );
statements.add( parseStmt() );

// 3. 批量添加元素
params.addAll( parseFuncFParams() );

// 4. 获取元素 (在解析器中不常用，因为我们通常是顺序处理)
Token t = tokens.get(index);

// 5. 获取大小 (常用于 peek 的边界检查)
int size = tokens.size();
```

SWITCH 记得case加break

注意左递归！！！

可以理解为LAndExp -> EqExp && EqExp && EqExp && EqExp..... 所以每读一个EqExp也等于读了一个LAndExp

可以用 a && b && c模拟一下

```
// LAndExp → EqExp | LAndExp '&&' EqExp
    // LAndExp -> EqExp LAndExp'
    // LAndExp' -> '&&' EqExp LAndExp' | null
    private ExprNode parseLAndExp() {
        ExprNode left = parseEqExp();
        while (currentToken().getType() == TokenType.AND) {
            printComponent("LAndExp");
            consumeToken();
            ExprNode right = parseEqExp();
            BinaryOpExp node = new BinaryOpExp();
            node.op = Operator.AND;
            node.left = left;
            node.right = right;
            left = node;
        }
```

```
        printComponent("LAndExp");
        return left;
    }
```

字符串拼接

StringBuilder

```
StringBuilder sb = new StringBuilder();
sb.append(); // 追加字符
sb.toString(); // 转成String
int len = sb.length() //字符串的长度
sb.setLength(k); // 把字符串设置到指定长度，如果长了就截断，如果短了会用空字符填充
```

Character

```
Character.isDigit(char ch);
Character.isLetter(char ch); // 包含大小写
Character.isLetterOrDigit(char ch);
Character.isWhitespace(char ch);
Character.isUpperCase(char ch);
Character.isLowerCase(char ch);

// 字符大小写转换
Character.toLowerCase(char ch);
Character.toUpperCase(char ch);
```

String

```
s.length();
s.charAt(index);
s.equals(String str);
String s = string.substring(start, end);  // start to end-1位置的子字符串
String s_lower = s.toLowerCase(); //toUpperCase同理
String.valueOf(Char ch) // char to string
```

注意区分预读和回溯的两种方法（尽量都可以用回溯

```
// 可能是 LVal = Exp; 或 Exp;
// LVal → Ident ['[' Exp ']']
// Exp也可以是ident开头的 还包含了 ident(FuncRParam) 函数调用
// 先用预读看看当前是不是一个赋值表达式
```

```
private boolean isAssignStmt() {
        // LVal 必须是标识符开头的
        // 可能是 LVal = Exp; 或 Exp;
        if (peek(0).getType() != TokenType.IDENFR) {
            return false;
        }
        // 模拟向后扫描，计数偏移
        int offset = 1;
        // 不是数组的情况
        if (peek(offset).getType() == TokenType.ASSIGN) {
            return true;
        }

        // 跳过'['
        if (peek(offset).getType() == TokenType.LBRACK) {
            offset++;

            int cnt = 1;
            while(peek(offset).getType() != TokenType.EOFSY
            && cnt > 0) {
                if (peek(offset).getType() == TokenType.ASSIGN){
                    return true;
                }
                if (peek(offset).getType() == TokenType.LBRACK) {
                    cnt++;
                }
                if (peek(offset).getType() == TokenType.RBRACK) {
                    cnt--;
                }
                offset++;
            }
            if (cnt!=0 || peek(offset).getType() == TokenType.EOFSY) {
                return false;
            }
            //offset++; //跳过']'
        }

        return peek(offset).getType() == TokenType.ASSIGN;
    }


if (isAssignStmt()) {
                    //consumeToken();
                    AssignStmt assignnode = new AssignStmt();
                    assignnode.lVal = (LVal) parseLVal();
                    expect(TokenType.ASSIGN, "assign_error", false);
                    assignnode.rValue = parseExp();
```

```
                    expect(TokenType.SEMICN, "i", true);
                    node = assignnode;
                } else {
                    ExprStmt exprnode = new ExprStmt();
                    exprnode.expr = parseExp();
                    expect(TokenType.SEMICN, "i", true);
                    node = exprnode;
                }
```

回溯:

```
// 尝试回溯的逻辑
                int pos = currentPos;
                // 记录当时的输出位置和错误数量
                int outputLen = outputBuilder.length();
                int errorSize = errors.size();
                // 先解析一个LVal
                ExprNode lvalNode = parseLVal();
                // 再判断接下来是不是等号
                if (currentToken().getType() == TokenType.ASSIGN) {
                    consumeToken();
                    AssignStmt assignnode = new AssignStmt();
                    assignnode.lVal = (LVal) lvalNode;
                    assignnode.rValue = parseExp();
                    expect(TokenType.SEMICN, "i", true);
                    node = assignnode;
                } else {
                    // 如果失败的话就要回滚
                    currentPos = pos;
                    outputBuilder.setLength(outputLen);
                    while(errors.size() > errorSize) {
                        errors.remove(errors.size() - 1);
                    }
                    ExprStmt exprnode = new ExprStmt();
                    exprnode.expr = parseExp();
                    expect(TokenType.SEMICN, "i", true);
                    node = exprnode;
                }
                break;
```

注意语法分析树的输出是后序遍历的

前序 根-左-右

中序 左-根-右

后序 左-右-根

Lexer

判断读入的每个字符，每一个currentPos++都要加hasNext()判断

```
" " skip
_ || isLetter -> ident/关键字
digit -> intconst
+,-,*,%,;,,,(,),[,],{,} 直接读一位，没有其他情况
! -> ! or !=
= -> = or ==
< -> < or <=
& -> & or &&
/ -> 三种情况 div 行级注释 块级注释
" -> stringconst
```

Parser

首先判断首字符集，然后做语义的推导