编译设计文档

词法分析器 Lexer

Lexer 将输入的字节流解析成一个一个的 token，并封装在 ArrayList 中。

```
        //词法分析
        Lexer lexer = new Lexer(source);
        TokenStream tokenStream = new TokenStream(lexer.getTokenArrayList());
```

具体实现细节

运行

通过调用 `getTokenArrayList` 函数，如果 `ArrayList` 为空，则使用 `while` 循环不断调用 `getCurToken` 并填入 `ArrayList`。

`read` 封装 `InputStream.read()`，读取下一个字符。

空白符

- `' '` 或 `\t`
- 换行 `\r\n` 或 `\n`

解析 token

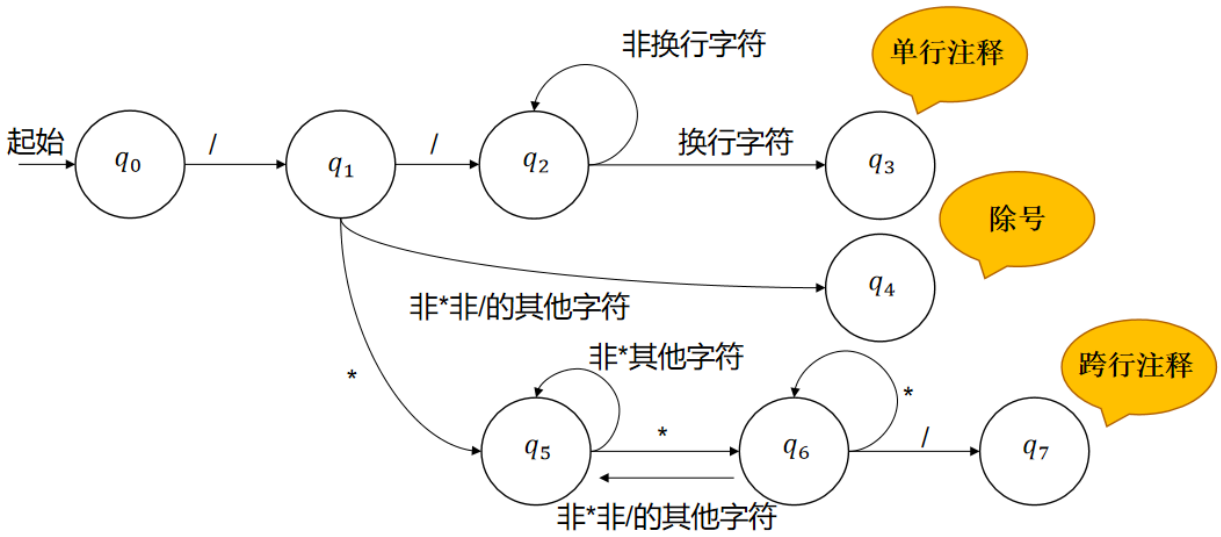
主要分成以下几个类

- skip 掉空白符
- EOF 结束标志
- 符号类
- 数字类
- string类
- ident类（标识符）
- 注释和除法类

注释和除法

这里使用 FSM

状态转移图跟PPT一致




```
        // DIV & note
        else if (curChar == '/') {
            sb.append(curChar);
            read();
            //div (q4)
            if (curChar != '/' && curChar != '*') {
                return new Token(TokenType.DIV,"/",curLine);
            }
            //only one line note (q2)
            else if (curChar == '/') {
                read();
                while (!isNewLine()) {
                    read();
                }
                //q3
                read();
                curLine++;
                return getCurToken();
            }
            //multi lines note (q5)
            //FSM
            else {
                read();
                int state = 5;
                while (state == 5 || state == 6) {
                    state = FSM(state);
                }
                if (state == 7) {
                    return getCurToken();
                } else {
                    return new Token(TokenType.ERROR,"/**/",curLine);
                }
            }

            private int FSM(int state) throws IOException {
```

```
        switch (state) {
            case 5:
                if (curChar == '*') {
                    read();
                    return 6;
                } else {
                    if (isNewLine()) {
                        curLine++;
                    }
                    read();
                    return 5;
                }
            case 6:
                if (curChar == '/') {
                    read();
                    return 7;
                } else if (curChar == '*') {
                    read();
                    return 6;
                } else {
                    if (isNewLine()) {
                        curLine++;
                    }
                    read();
                    return 5;
                }
            case 7:
                read();
                return 7;
            default:
                System.out.println("error in /**/");
                return -1;
        }
    }
```

反思

在 `getToken` 函数里写了很多如果输入错误的判断，并创造 `TokenType.ERROR`，如今看来是不必要的，因为错误处理的词法部分并不涉及这些，属于是当时想多了。

语法分析器 Parser

Parser 将得到的 Token 解析出语法成分，并构建语法树，最后后序遍历输出语法树。

```
        //语法分析
        Parser parser = new Parser(tokenStream);
        Node root = parser.parseCompUnit();
        //打印流
        PrintStream ps = new PrintStream("output.txt");
        System.setOut(ps);
        //后序遍历输出语法树
        visitNode(root);
```

统一约定

- 默认读取到将要解析的 node 的第一个 token
- 调用完 parse 方法，则对应的 node 已经读完并得到
- TokenNode 需要手动 read 读完
- 获得 endLine 不能用 curToken 否则会出现 null pointer

实现细节

辅助方法

- `TokenStream.watch()` 帮助预读或者回读。

Node

- Node 非终结符

```
    private int startLine;
    private int endLine;
    private SyntaxType SyntaxType;
    private ArrayList<Node> children;
```

- TokenNode 终结符

  子节点集合为null，遍历到它时只输出 TokenType 和 内容。

```
public class TokenNode extends Node{
    private Token token;

TokenNode constNode = new
TokenNode(startLine,startLine,SyntaxType.TOKEN,null,curToken);
```

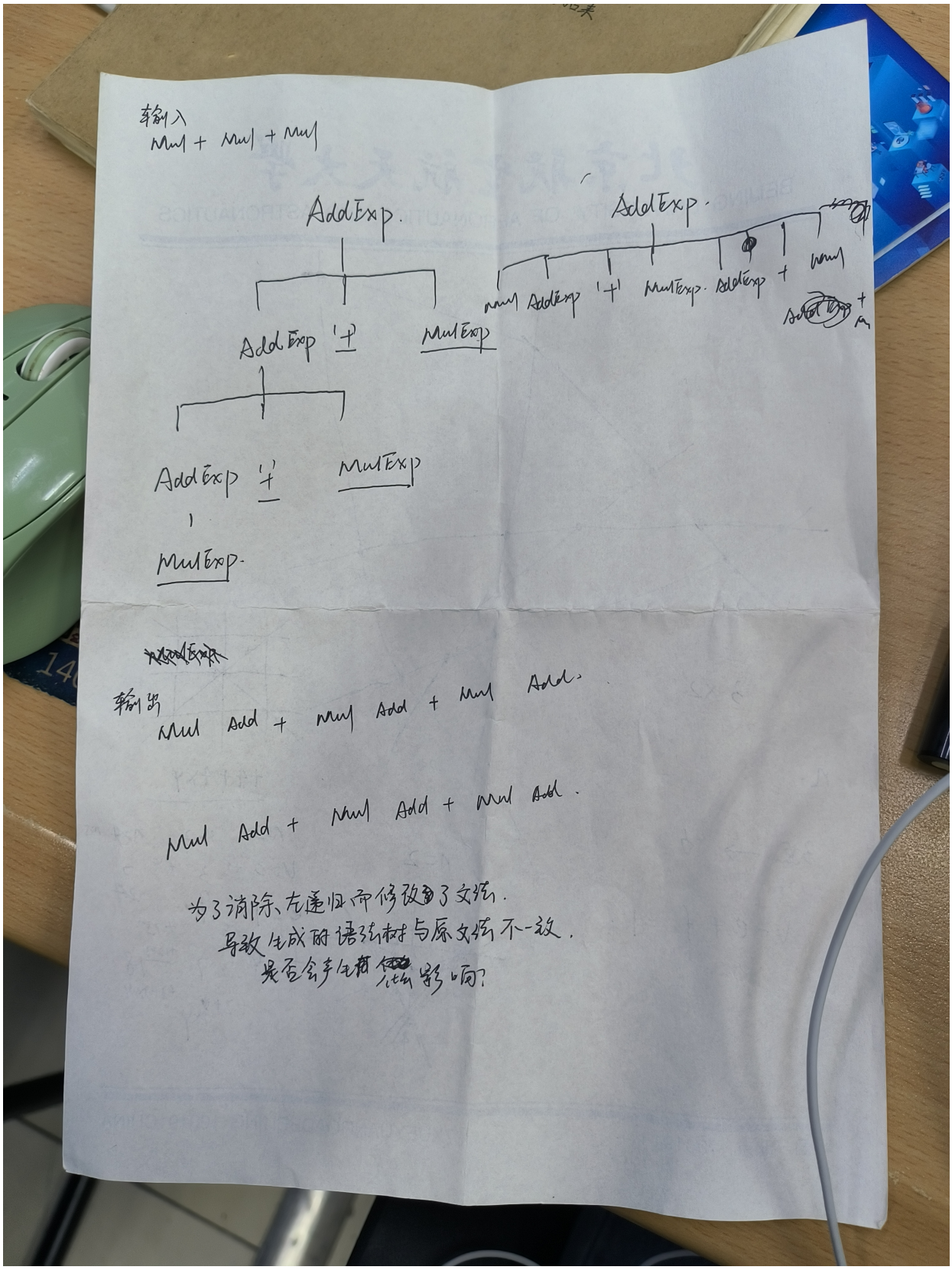
修改左递归文法

```
AddExp → MulExp | AddExp ('+' | '−') MulExp
AddExp -> MulExp {'+' MulExp | '-' MulExp}

MulExp → UnaryExp | MulExp ('*' | '/' | '%') UnaryExp
```

```
MulExp -> UnaryExp {'*' UnaryExp | '/' UnaryExp | '%' UnaryExp}

LOrExp → LAndExp | LOrExp '||' LAndExp
LOrExp -> LAndExp {'||' LAndExp}

RelExp → AddExp | RelExp ('<' | '>' | '<=' | '>=') AddExp
RelExp -> AddExp {'<' AddExp | '>' AddExp | '<=' AddExp | '>=' AddExp}

EqExp → RelExp | EqExp ('==' | '!=') RelExp
EqExp -> RelExp {'==' RelExp | '!=' RelExp}

LAndExp → EqExp | LAndExp '&&' EqExp
LAndExp -> EqExp {'&&' EqExp}
```

虽然修改文法可以消除左递归，但是会对语法树的建立以及语法成分的输出产生影响。因为我是后续遍历语法树输出语法成分，所以我在建立语法树时，添加了一些 children 为 null 的非终结符作为叶子节点，起到矫正语法输出的作用，一个例子如下。

```
public Node parseAddExp() {
    int startLine = curToken.getLine();
    ArrayList<Node> children = new ArrayList<>();
    //parse MulExp
    Node node = parseMulExp();
    children.add(node);
    //parse '+' or '-'
    while (curToken.getType() == TokenType.PLUS || curToken.getType() ==
TokenType.MINU) {
        //防止文法改变导致语法树输出改变
        node = new
Node(startLine,tokenStream.watch(-1).getLine(),SyntaxType.ADD_EXP,null);
        children.add(node);
        //parse '+' or '-'
        TokenNode pmNode = new
TokenNode(curToken.getLine(),curToken.getLine(),SyntaxType.TOKEN,null,curToken);
        children.add(pmNode);
        read();
        //parse MulExp
        node = parseMulExp();
        children.add(node);
    }
```



反思

- 递归下降的方法，通过 FIRST 集来判断该 parse 什么。
- 有些判断（比如 `[Exp] ;` `LVal = ...`）的判断写法并不适合后续的错误处理，还是因为没有严格按照 FIRST 集来判断，用了一些 trick 投机了。
- 很容易多读或者少读 token 导致后续出错，一定要写好注释，知道每一步在干什么。