编译原理实验二报告

2016-11-16

南京大学软件学院

张云飞 141250197

一、 实验题目

语法分析程序的代码实现

二、 实验内容

根据设计的上下文无关文法，提取公共左因子和消除左递归后，利用程序生成 LL(1)文法的预测分析表，然后对输入的 token 序列进行语法检测。

三、 实验步骤

1. 设计上下文无关文法

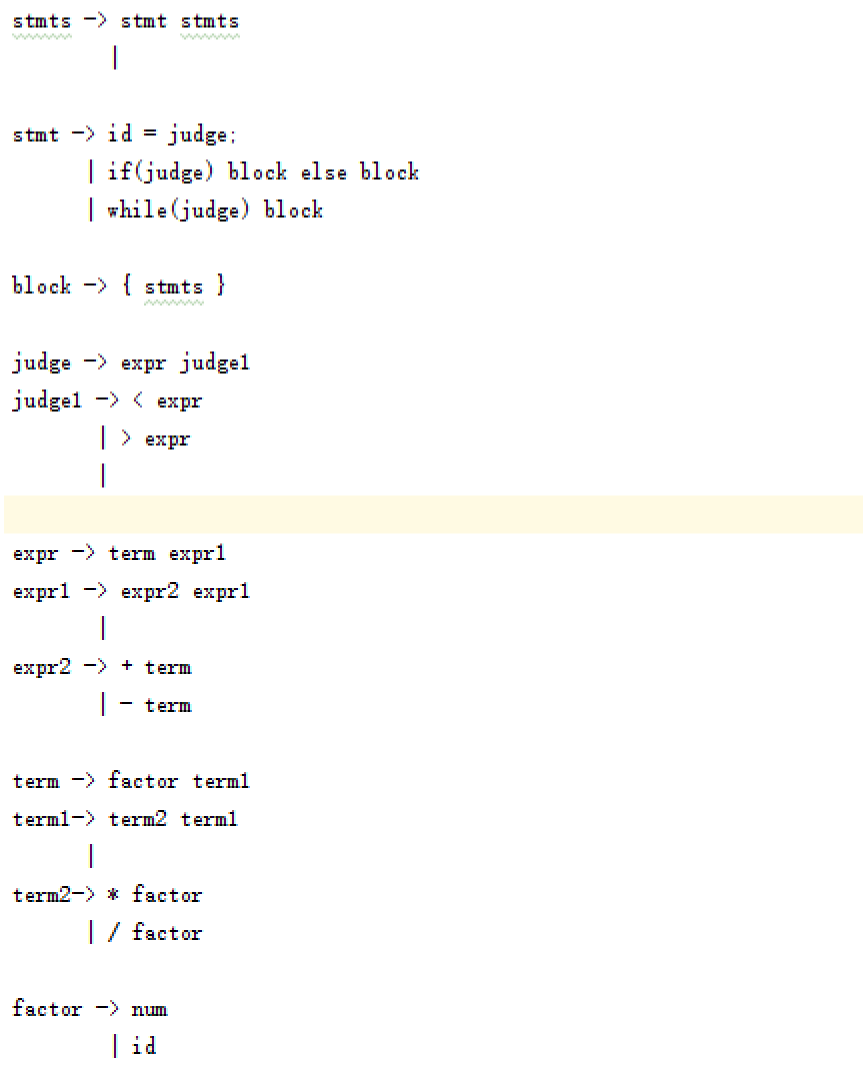
文法支持以下表达式：

a)普通的加减和赋值表达式

b)if else 语句

c)While 语句

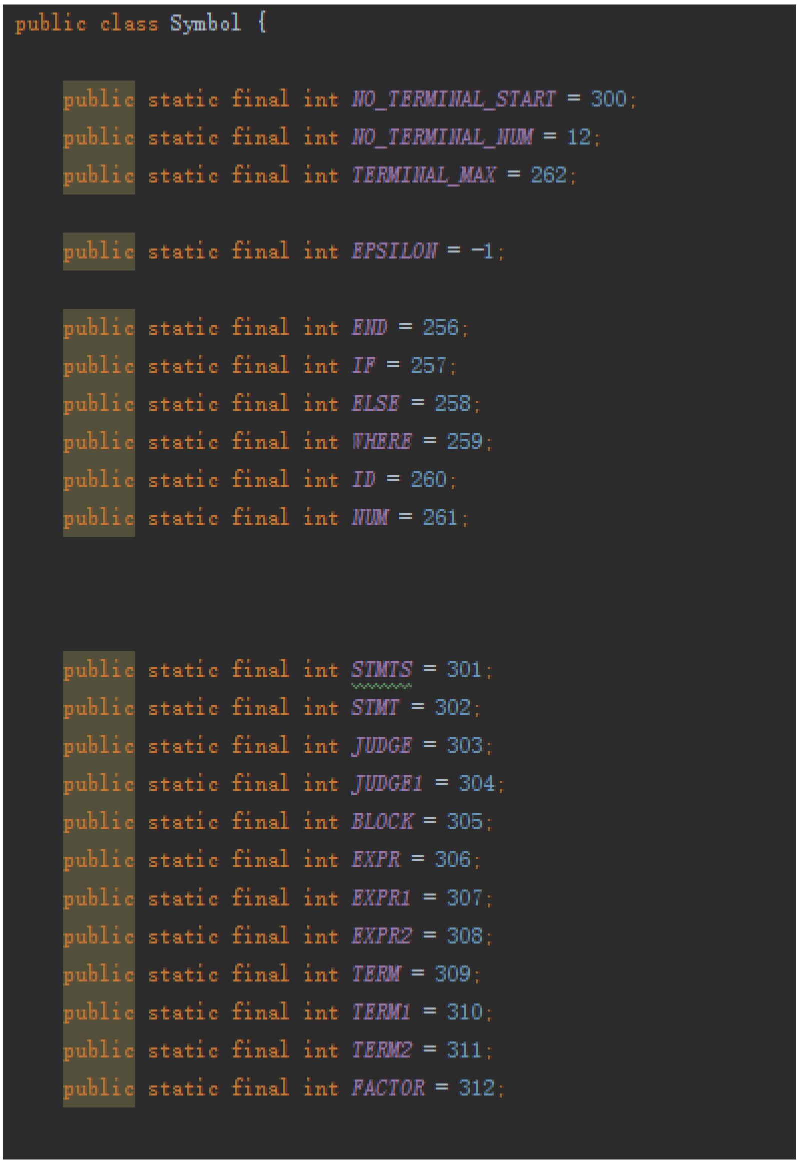
对文法做提取公共左因子和消除左递归后的结果如下：



```
stmts -> stmt stmts
        |

stmt -> id = judge;
      | if(judge) block else block
      | while(judge) block

block -> { stmts }

judge -> expr judge1
judge1 -> < expr
        | > expr
        |

expr -> term expr1
expr1 -> expr2 expr1
        |
expr2 -> + term
        | - term

term -> factor term1
term1-> term2 term1
      |
term2-> * factor
      | / factor

factor -> num
        | id
```

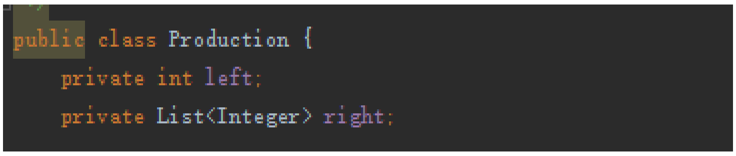
2. 根据文法自动生成 LL(1)的预测分析表

   a) 首先将文法中的终结符和非终结符映射为一个对应的数值。在 Symbol 类中实现符号与数值的映射。



```java
public class Symbol {

    public static final int NO_TERMINAL_START = 300;
    public static final int NO_TERMINAL_NUM = 12;
    public static final int TERMINAL_MAX = 262;

    public static final int EPSILON = -1;

    public static final int END = 256;
    public static final int IF = 257;
    public static final int ELSE = 258;
    public static final int WHERE = 259;
    public static final int ID = 260;
    public static final int NUM = 261;



    public static final int STMTS = 301;
    public static final int STMT = 302;
    public static final int JUDGE = 303;
    public static final int JUDGE1 = 304;
    public static final int BLOCK = 305;
    public static final int EXPR = 306;
    public static final int EXPR1 = 307;
    public static final int EXPR2 = 308;
    public static final int TERM = 309;
    public static final int TERM1 = 310;
    public static final int TERM2 = 311;
    public static final int FACTOR = 312;
```

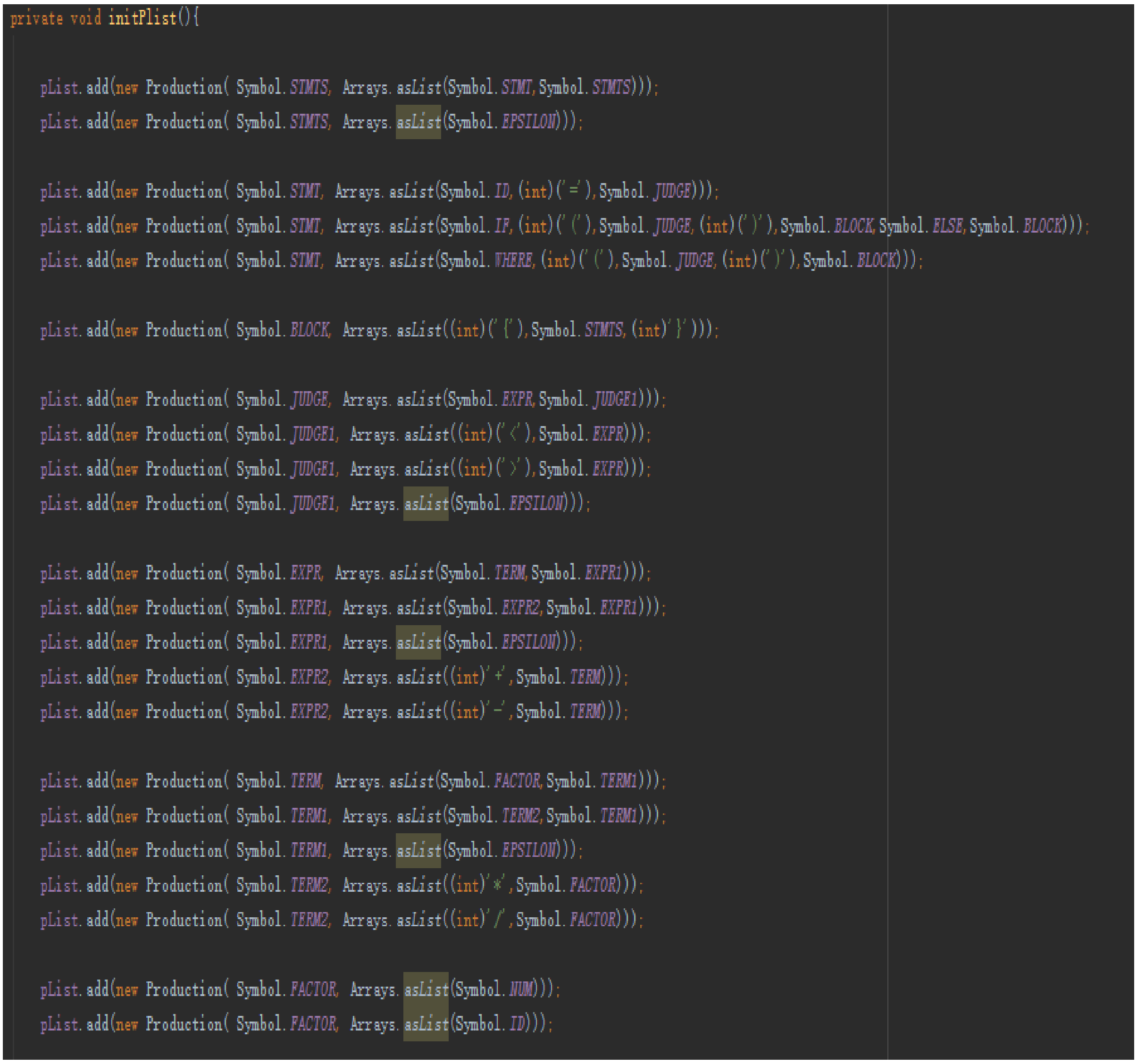
b) 将文法表达式转为数值表示形式写入程序中



```java
public class Production {
    private int left;
    private List<Integer> right;
```

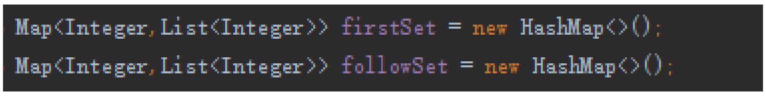
定义 Production 类，左侧为非终结符，右侧为非终结符推导的右部。

在 ProductonTable 类中把之前设计好的文法表达式写入。



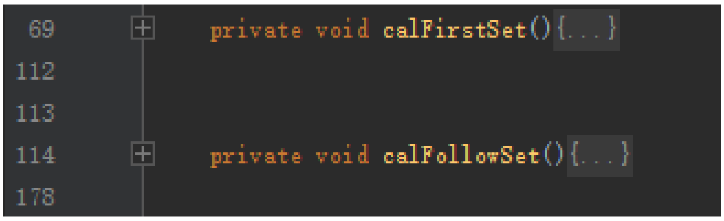
```java
private void initPlist(){

    pList.add(new Production( Symbol.STMTS,  Arrays.asList(Symbol.STMT,Symbol.STMTS)));
    pList.add(new Production( Symbol.STMTS,  Arrays.asList(Symbol.EPSILON)));

    pList.add(new Production( Symbol.STMT,  Arrays.asList(Symbol.ID,(int)('='),Symbol.JUDGE)));
    pList.add(new Production( Symbol.STMT,  Arrays.asList(Symbol.IF,(int)('('),Symbol.JUDGE,(int)(')'),Symbol.BLOCK,Symbol.ELSE,Symbol.BLOCK)));
    pList.add(new Production( Symbol.STMT,  Arrays.asList(Symbol.WHERE,(int)('('),Symbol.JUDGE,(int)(')'),Symbol.BLOCK)));

    pList.add(new Production( Symbol.BLOCK,  Arrays.asList((int)('{'),Symbol.STMTS,(int)'}')));

    pList.add(new Production( Symbol.JUDGE,  Arrays.asList(Symbol.EXPR,Symbol.JUDGE1)));
    pList.add(new Production( Symbol.JUDGE1,  Arrays.asList((int)('<'),Symbol.EXPR)));
    pList.add(new Production( Symbol.JUDGE1,  Arrays.asList((int)('>'),Symbol.EXPR)));
    pList.add(new Production( Symbol.JUDGE1,  Arrays.asList(Symbol.EPSILON)));

    pList.add(new Production( Symbol.EXPR,  Arrays.asList(Symbol.TERM,Symbol.EXPR1)));
    pList.add(new Production( Symbol.EXPR1,  Arrays.asList(Symbol.EXPR2,Symbol.EXPR1)));
    pList.add(new Production( Symbol.EXPR1,  Arrays.asList(Symbol.EPSILON)));
    pList.add(new Production( Symbol.EXPR2,  Arrays.asList((int)'+',Symbol.TERM)));
    pList.add(new Production( Symbol.EXPR2,  Arrays.asList((int)'-',Symbol.TERM)));

    pList.add(new Production( Symbol.TERM,  Arrays.asList(Symbol.FACTOR,Symbol.TERM1)));
    pList.add(new Production( Symbol.TERM1,  Arrays.asList(Symbol.TERM2,Symbol.TERM1)));
    pList.add(new Production( Symbol.TERM1,  Arrays.asList(Symbol.EPSILON)));
    pList.add(new Production( Symbol.TERM2,  Arrays.asList((int)'*',Symbol.FACTOR)));
    pList.add(new Production( Symbol.TERM2,  Arrays.asList((int)'/',Symbol.FACTOR)));

    pList.add(new Production( Symbol.FACTOR,  Arrays.asList(Symbol.NUM)));
    pList.add(new Production( Symbol.FACTOR,  Arrays.asList(Symbol.ID)));
```

c) 计算 first 和 follow 集合



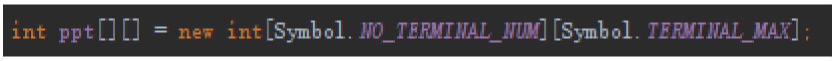
```java
Map<Integer,List<Integer>> firstSet = new HashMap<>();
Map<Integer,List<Integer>> followSet = new HashMap<>();
```

在 ProductionTable 中为每个非终结符建立 first 和 follow 的映射集合。First 和 follow 集合的计算方式按照课件中的流程通过有限次的循环最终生成。



```
private void calFirstSet(){...}


private void calFollowSet(){...}
```

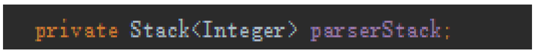
d) 生成预测分析表



```
int ppt[][] = new int[Symbol.NO_TERMINAL_NUM][Symbol.TERMINAL_MAX];
```

在 PPTBuilder 中建立二维数组保存预测分析表。根据之前计算得到的 first,follow 集合和文法推到表达式，最终计算生成预测分析表。

3. 根据生成的预测分析表对输入进行语法推导分析



```
private Stack<Integer> parserStack;
```

在类 ProductionHandler 中，利用在 parserStack 上的一系列操作，完成对语法的推导分析。

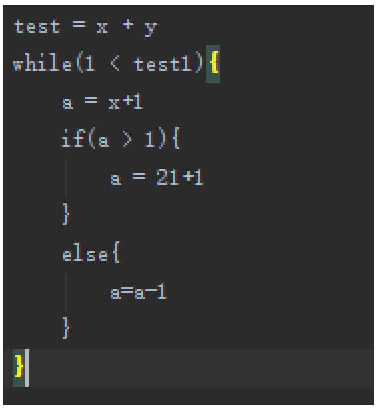
四、 补充说明

1. token 序列的生成

利用实验一中的程序，在 source.txt 中输入语句，实验一中的代码会将输入语句转化为一个 token 序列，提供给语法分析程序使用。

2. 输入和输出

输入放在 source.txt 中，当前的输入为：



```
test = x + y
while(1 < test1){
    a = x+1
    if(a > 1){
        a = 21+1
    }
    else{
        a=a-1
    }
}
```

可以按照当前格式输入运算表达式，if,else 和 while 语句，目前的文法中定义每个 if 后面必须有一个 else，如果没有会提示错误。

输出在控制台下，首先会输出定义的正则表达式，然后输出解析后的 token 序列。最后输出语法的推导解析过程。当前输入下，推导解析过程如下：

自顶向下语法分析过程：

stmts->stmt stmts

stmt->id = judge

judge->expr judge1

expr->term expr1

term->factor term1

factor->id

term1->epsilon

expr1->expr2 expr1

expr2->+ term

```
term->factor term1
factor->id
term1->epsilon
expr1->epsilon
judge1->epsilon
stmts->stmt stmts
stmt->while ( judge ) block
judge->expr judge1
expr->term expr1
term->factor term1
factor->num
term1->epsilon
expr1->epsilon
judge1->< expr
expr->term expr1
term->factor term1
factor->id
term1->epsilon
expr1->epsilon
block->{ stmts }
stmts->stmt stmts
stmt->id = judge
```

```
judge->expr judge1
expr->term expr1
term->factor term1
factor->id
term1->epsilon
expr1->expr2 expr1
expr2->+ term
term->factor term1
factor->num
term1->epsilon
expr1->epsilon
judge1->epsilon
stmts->stmt stmts
stmt->if ( judge ) block else block
judge->expr judge1
expr->term expr1
term->factor term1
factor->id
term1->epsilon
expr1->epsilon
judge1->> expr
expr->term expr1
```

```
term->factor term1
factor->num
term1->epsilon
expr1->epsilon
block->{ stmts }
stmts->stmt stmts
stmt->id = judge
judge->expr judge1
expr->term expr1
term->factor term1
factor->num
term1->epsilon
expr1->expr2 expr1
expr2->+ term
term->factor term1
factor->num
term1->epsilon
expr1->epsilon
judge1->epsilon
stmts->epsilon
block->{ stmts }
stmts->stmt stmts
```

```
    stmt->id = judge
    judge->expr judge1
    expr->term expr1
    term->factor term1
    factor->id
    term1->epsilon
    expr1->expr2 expr1
    expr2->- term
    term->factor term1
    factor->num
    term1->epsilon
    expr1->epsilon
    judge1->epsilon
    stmts->epsilon
    stmts->epsilon
    stmts->epsilon
语法分析完成
```