《编译原理》

实验指导书

张志远　编 写

适用专业：　　　计算机科学与技术

中国民航大学计算机综合实验中心

二〇二二年四月

前　　言

本次在2018版编译原理实验的基础上进行较大修订，首先为更好地服务于教学，增添了LL(1)预测分析及LR语法分析两个小型实验；其次摒弃了eclipse JDT中的AST抽象语法树，使用递归下降分析程序基于L-翻译模式生成三地址代码，更加贴合理论教学内容。

目　　录

实验一　词法分析实验

实验学时：2
实验类型：（验证、√综合、设计）

一、　实验目的

1. 掌握利用状态转换图进行词法分析的方法。
2. 掌握用程序实现状态转换图的方法。
3. 了解词法分析和语法分析的接口。

二、　实验环境

1. Jdk1.8 + Eclipse

三、　实验要求

1. 手工为 simpleBlock 语言编写一个词法分析器
2. 能正确分析以下两个程序：

```
/*comment lines
* until here */
{
   int i1, i2, i3;
   i1 = 014;
   i2 = i1+0x20AF*3;
   i3 = i1-5*(i2%2)+5;
   if(i1==14 && i2>=20){
      i3=i3+1;}
   else{
      i3=i3+2;}
}
```

```
{
   //define two variables m and n
   int m,n;
   m = 12; n = 21;
   if(m<n){
      int t;
      t=m; m=n; n=t;
   }
   int r;
   r = m%n;
   while(r!=0){m=n; n=r; r=m%n;}
}
```

四、　实验步骤

1、simpleBlock 语言的词法特点

注释：注释同 java 语言，包括单行注释//和多行注释，/*　　　*/。

标识符：字母开头，由字母、数字和下划线组成的字符序列，区分大小写。

关键字：int boolean if else while

运算符和分隔符：

✧　分隔符：(　)　{　}　=　;　,
✧　关系运算符：>　<　==　<=　>=　!=
✧　逻辑运算符：!　&&　||
✧　算术运算符：+　-　*　/　%

字面常量：包括整形常量和布尔常量。其中整型常量只包含十进制数，布尔常量有true和false两个。

2. 常量统一采用以下符号定义

int: KEY_INT	boolean: KEY_BOOLEAN	if: KEY_IF	else: KEY_ELSE	while: KEY_WHILE
(:LPARAM	):RPARAM	{:LBRACE	}:RBRACE	=:ASSIGN
;:SEMICOLON	,:COMMA	.:DOT		
>:GREATER	<:LESS	==:EQUAL	<=:LESS_EQUAL	>=:GREATER_EQUAL
!=:NOT_EQUAL		<<:LEFTSHIFT	>>:RIGHTSHIFT	
!:LOGICAL_NOT	&&:LOGICAL_AND	||:LOGICAL_OR		
+:PLUS	++:PLUSPLUS	+=:PLUSEQUAL		
-:MINUS	--:MINUSMINUS	-=:MINUSEQUAL		
*:TIMES	*=:TIMESEQUAL	/:DIVIDE	/=:DIVIDEEQUAL	

%:REMAINDER	%:REMAINDEREQUAL			
整形常量: NUMBER_LITERAL	true: BOOL_TRUE	false: BOOL_FALSE	标识符: IDENTIFIER	文件结束符: EOF

3. 手工编写词法分析单位类

3.1 TokenType.java

TokenType.java 中的 TokenType 枚举了 simpleBlock 语言中所有的单词类型，如表 2 所示。以及一些后续语法分析中涉及到的非终结符等信息。完整的 TokenType.java 文件内容如下（**本文件内容不做修改**）：

```
package lab1;
public enum TokenType{
    /** 忽略的词法单位 **/
    IGNORE,

    /** 变量 **/
    IDENTIFIER,         //标识符

    /** 常量 **/
    NUMBER_LITERAL, //整形常量
    BOOL_TRUE,          //true
    BOOL_FALSE,         //false

    /** 保留字 */
    KEY_INT,  //int
    KEY_BOOLEAN,//boolean
    KEY_WHILE,     //while
    KEY_IF,            //if
    KEY_ELSE,        //else

    /** 算术运算符 */
    PLUS,              //+
    PLUSPLUS,        //++
    PLUSEQUAL,     //+=
    MINUS,            //-
    MINUSMINUS,//--
    MINUSEQUAL,//-=
    TIMES,             //*
    TIMESEQUAL,  //*=
    DIVIDE,            ///
    DIVIDEEQUAL,//*=
    REMAINDER,   //%
    REMAINDEREQUAL,         //%=

    /** 位运算符 */
    LEFTSHIFT,            //<<
    RIGHTSHIFT,          //>>

    /** 关系运算符 */
    LESS,                     //<
    GREATER,         //>
    LESS_EQUAL,          //<=
    GREATER_EQUAL,   //>=
    NOT_EQUAL,           //!=
    EQUAL,              //==

    /**  逻辑运算符  */
    LOGICAL_NOT,//!
    LOGICAL_AND,        //&&
    LOGICAL_OR,         //||

    /**  赋值符号  */
    ASSIGN,             //=

    /**  括号  */
    LPAREN,         //(
    RPAREN,         //)
    LBRACKET,       //{
    RBRACKET,       //}

    /**  界符  */
    COMMA,          //逗号,
    SEMICOLON,  //分号;
    DOT,        //圆点.

    /**  文件结尾符  */
    EOF,        //end of file

    /**  非终结符号以及一些特殊的符号，语法分析时使用  */
    Epsilon,            //空
    Start,              //总的开始符号
    Simpleblock,        //{****}
    Sequence,           //语句序列
    assignmentStatement,//赋值语句

    Expression,                 //E
    Expression_1,               //E'
    Term,                       //T
    Term_1,                     //T'
    Factor,                     //F

    Boolexpression,             //布尔表达式
    Boolexpression_1,
    Boolterm,
    Boolterm_1,
    Boolfactor,
    relationalExpression,       //关系表达式
    relationalOperator,         //关系运算符

    ifStatement,                //if 语句
    OptionalElse,               //else 语句（可选）
    whileStatement              //while 语句
}
```

3.2 Token.java

Token.java 中的 Token 类标记了一个完整的词法分析单位，包括单词类型、单词的字面值以及其所在的行列信息。完整的 Token.java 文件内容如下（**本文件内容不做修改**）：

```
package lab1;
public class Token {
    private TokenType type;
    private String token;
    private int line;
    private int column;

    public Token(TokenType type, String token, int line, int column) {
        this.type = type;
        this.token = token;
        this.line = line;
        this.column = column;
    }

    public TokenType getType() {
        return type;
    }

    public int getLine(){
        return line;
    }

    public int getColumn(){
        return column;
    }

    public String getLexeme(){
        return token;
    }

    public String toString() {
        return type + " " + token + "      (" + line + ", " + column + ")";
    }
}
```

4. 手工编写词法分析类

BlockLexer.java 是词法分析的主要实现类，其中的 nextToken()方法用来获取下一个词法单位，示例中已经给出了基本的词法单位分析过程，同学们需补充/**begin**/和/***end***/之间的代码，包括**去掉注释，识别关系运算符和逻辑运算符等**。

nextToken 方法使用课堂所讲状态转换图方法获取下一个词法单位，用于在后续实验中将其反馈给语法分析使用。在识别出一个词法单位后，使用 getToken 方法返回这个词法单位并清空用于词法记号值的 lexeme 变量。

nextChar 方法取得下一个字符，取得的字符放在成员变量 c 当中；pushbackChar 方法在输入流中回退一个多读入的字符（例如在读取数字常量如 32+45 时，当遇到非数字字符“+”时需要将该字符回退到输入流，否则加号就被忽略过去了）。

dropChar 方法删除多余的字符，如空格、换行等，对于注释同样也应该删除。

BlockLexer.java 示例文件如下：

```
package lab1;
import java.io.*;

public class BlockLexer{
    private PushbackReader in = null;
    private StringBuffer lexeme = new StringBuffer();
private char c;
private int line = 0;
private int column = 0;

public BlockLexer(String infile) {
	PushbackReader reader = null;
	try {
		reader = new PushbackReader(new FileReader(infile));
	} catch(IOException e) {
		e.printStackTrace();
		System.exit(-1);
	}
	in = reader;
}

//取得下一个字符
private void nextChar() {
	try {
		c = (char)in.read();
		lexeme.append(c);
		column++;
	} catch (IOException e) {
		e.printStackTrace();
		System.exit(1);
	}
}

//回退一个字符（多读入的）
private void pushbackChar() {
	try {
		in.unread(lexeme.charAt(lexeme.length() - 1));
		lexeme.deleteCharAt(lexeme.length() - 1);
		column--;
	} catch (IOException e) {
		e.printStackTrace();
		System.exit(1);
	}
}

//取得词法记号，并重置状态变量
private Token getToken(TokenType type) {
	String t = lexeme.toString();
	lexeme.setLength(0);
	return new Token(type, t, line + 1, column - t.length() + 1);
}

//扔掉一个字符（此时单词应该还未开始，只需把长度设为 0 即可）
private void dropChar() {
	lexeme.setLength(0);
}

//去空格、换行、回车等
private void removeSpace() {
	this.nextChar();
```

```
    while (Character.isWhitespace(c)) {
        if (this.c == '\n') {
            this.line++;
            this.column = 0;
        }
        this.dropChar();
        this.nextChar();
    }
    this.pushbackChar();
}

//识别标识符
private Token getID_or_Keywords() {
    int s = 0;
    while(true) {
        switch(s) {
        case 0:
            nextChar();
            if(Character.isLetterOrDigit(c) || c=='_') s = 0;
            else s = 1;
            break;
        case 1:
            this.pushbackChar();
            String t = this.lexeme.toString();
            if (t.equalsIgnoreCase("int")){
                return getToken(TokenType.KEY_INT);
            } else if(t.equalsIgnoreCase("boolean")) {
                return getToken(TokenType.KEY_BOOLEAN);
            } else if(t.equalsIgnoreCase("if")) {
                return getToken(TokenType.KEY_IF);
            } else if(t.equalsIgnoreCase("else")) {
                return getToken(TokenType.KEY_ELSE);
            } else if(t.equalsIgnoreCase("while")) {
                return getToken(TokenType.KEY_WHILE);
            } else if(t.equalsIgnoreCase("true")) {
                return getToken(TokenType.BOOL_TRUE);
            } else if(t.equalsIgnoreCase("false")) {
                return getToken(TokenType.BOOL_FALSE);
            } else {
                return getToken(TokenType.IDENTIFIER);
            }
        }
    }
}

/*****************************begin******************************/
//识别整形常数，可能是十进制、八进制或十六进制
private Token getIntConst() {
    return null;
}

//识别/,/=
//去多行注释/*      */
//去单行注释//
```

```
private Token getDivide_or_removeComment() {
        return null;
}

//识别+,++,+=
private Token getPlus() {
        return null;
}
//识别-,--,-=
private Token getMinus() {
        return null;
}
//识别*,*=
private Token getTimes() {
        return null;
}
//识别%,%=
private Token getRemainder() {
        return null;
}
//识别>,>>,>=
private Token getGreater() {
        return null;
}
//识别<,<<,<=
private Token getLess() {
        return null;
}
//识别=,==
private Token getAssign_or_Equal() {
        return null;
}
//识别!,!=
private Token getNot_or_NotEqual() {
        return null;
}

//识别&&
private Token getAnd() {
        return null;
}

//识别||
private Token getOr() {
        return null;
}
/*****************************end*******************************/
//获取下一个 token
public Token nextToken() {
        Token token = null;
        while(null == token) {
                this.removeSpace();
                this.nextChar();
}
```

```
            if ( Character.isDigit(c) ) {
                token = this.getIntConst();
            } else if ( Character.isLetter(c) || c == '_'){
                token = this.getID_or_Keywords();
            } else if ( c == '+'){
                token = this.getPlus();
            } else if ( c == '-'){
                token = this.getMinus();
            } else if ( c == '*'){
                token = this.getTimes();
            } else if ( c == '/'){
                token = this.getDivide_or_removeComment();
            } else if ( c == '%'){
                token = this.getRemainder();
            } else if ( c == '!'){
                token = this.getNot_or_NotEqual();
            } else if ( c == '&'){
                token = this.getAnd();
            } else if ( c == '|'){
                token = this.getOr();
            } else if ( c == '='){
                token = this.getAssign_or_Equal();
            } else if ( c == '>'){
                token = this.getGreater();
            } else if ( c == '<'){
                token = this.getLess();
            } else if ( c == '('){
                token = this.getToken(TokenType.LPAREN);
            } else if ( c == ')'){
                token = this.getToken(TokenType.RPAREN);
            } else if ( c == '{'){
                token = this.getToken(TokenType.LBRACKET);
            } else if ( c == '}'){
                token = this.getToken(TokenType.RBRACKET);
            } else if ( c == ';'){
                token = this.getToken(TokenType.SEMICOLON);
            } else if (c == ','){
                token = this.getToken(TokenType.COMMA);
            } else if (c == '.'){
                token = this.getToken(TokenType.DOT);
            } else if ((c & 0xff) == 0xff) {
                token = this.getToken(TokenType.EOF);
            } else {
                System.out.println(" get nextToken error!");
                System.out.println(" find illegal character " + c);
                System.out.println(" at line " + (line + 1) + ",colum " + column);
                System.exit(1);
            }
        }
        return token;
    }
}
```

5. 编写 main 函数

右键单击 Compiler2022 项目，选择 new->folder，新建一个 test 文件夹。

右键单击 test 文件夹，选择 new->file，新建测试文件 lab1test1.txt 以及 lab1test2.txt，内容分别为实验要求中的两个程序。

编写 Lab1Main.java 文件内容如下，用于测试词法分析程序。

```
import java.io.*;
public class Lab1Main {
    public static void main(String args[]) {
        BlockLexer l = new BlockLexer("test/lab1test1.txt");
        Token s = l.nextToken();
        while (s != null && s.getType() != TokenType.EOF) {
            System.out.println(s);
            s = l.nextToken();
        }
    }
}
```

Lab1test1.txt 和 lab1test2.txt 输出结果如下：

```
LBRACKET {     (3, 1)
KEY_INT int     (4, 1)
IDENTIFIER i1     (4, 5)
COMMA ,     (4, 7)
IDENTIFIER i2     (4, 9)
COMMA ,     (4, 11)
IDENTIFIER i3     (4, 13)
SEMICOLON ;     (4, 15)
IDENTIFIER i1     (5, 1)
ASSIGN =     (5, 4)
NUMBER_LITERAL 014     (5, 6)
SEMICOLON ;     (5, 9)
IDENTIFIER i2     (6, 1)
ASSIGN =     (6, 4)
IDENTIFIER i1     (6, 6)
PLUS +     (6, 8)
NUMBER_LITERAL 0x20AF     (6, 9)
TIMES *     (6, 15)
......
......
SEMICOLON ;     (11, 8)
RBRACKET }     (11, 9)
RBRACKET }     (12, 1)
```

```
LBRACKET {     (1, 1)
KEY_INT int     (3, 1)
IDENTIFIER m     (3, 5)
COMMA ,     (3, 6)
IDENTIFIER n     (3, 7)
SEMICOLON ;     (3, 8)
IDENTIFIER m     (4, 1)
ASSIGN =     (4, 3)
NUMBER_LITERAL 12     (4, 5)
SEMICOLON ;     (4, 7)
IDENTIFIER n     (4, 9)
ASSIGN =     (4, 11)
NUMBER_LITERAL 21     (4, 13)
SEMICOLON ;     (4, 15)
KEY_IF if     (5, 1)
LPAREN (     (5, 3)
IDENTIFIER m     (5, 4)
LESS <     (5, 5)
......
......
SEMICOLON ;     (11, 28)
RBRACKET }     (11, 29)
RBRACKET }     (12, 1)
```

五、 实验总结

实验二 LL(1)语法分析实验

实验学时：2

实验类型：（验证、√综合、设计）

一、 实验目的

掌握利用预测分析表进行自上而下语法分析的方法

二、 实验环境

Jdk1.8 + Eclipse

三、 实验要求

本次实验必须在实验一词法分析正确完成的情况下才能进行。

要求能对以下代码进行语法分析。

```
{
position = init + rate * 60;
}
```

```
{
   i1 = 014;
   i2 = i1+0x20AF*3;
   i3 = i1-5*(i2%2)+5;
}
```

四、 实验步骤

1、本次实验使用的 LL（1）文法

Simpleblock → { Sequence }
Sequence → AssignmentStatement Sequence | ε
AssignmentStatement → IDENTIFIER = Expression;
Expression → Term Expression_1
Expression_1 → + Term Expression_1 | -Term Expression_1 | ε
Term → Factor Term_1
Term_1 → * Factor Term_1 | / Factor Term_1 | %Factor Term_1 | ε
Factor → (Expression) | IDENTIFIER | NUMBER_LITERAL

产生式	select
Simpleblock → { sequence }	{
Sequence→ assignmentStatement sequence	ID
Sequence→ ε	}
assignmentStatement → IDENTIFIER = expression ;	ID
Expression → term expression_1	(ID NUM
Expression_1 → + term expression_1	+
Expression_1 → - term expression_1	-
Expression_1 → ε	;)
Term → factor term_1	(ID NUM
Term_1 → * factor term_1	*
Term_1 → / factor term_1	/
Term_1 → % factor term_1	%
Term_1 → ε	+ - ;)
Factor → (expression)	(
Factor → IDENTIFIER	ID
Factor → NUMBER_LITERAL	NUM

2. 新建 LL1Table.java。LL1Table 类使用成员变量 table 存储 LL1 预测分析表，具体实现采用两层哈希表。Java 中的哈希表由 key 和 value 组成， 使用 put(key,value)向哈希表中添加内容，使用 get(key)从哈希表中读取。

Java 中的 HashMap 是一个模板类， 需要指定 key 和 value 的类型， table 第一层的 key 是 Tokentype 类型，value 是第二层哈希表。第二层的 key 同样是 TokenType，vaue 是 TokenType[]数组。为方便编程， TokenType 类中含有所有可能用到的终结符和非终结符号。LL1Table 类中提供了添加和查询 LL1 分析表的两个函数 addItem 和 getItem， 可以直接使用。

请在/**begin**/和/**end**/之间补充完整的 LL1 分析表内容。

```
package lab2;
import java.util.*;
import lab1.TokenType;
public class LL1Table {
    HashMap<TokenType, HashMap<TokenType, TokenType[]>> table = null;
    public LL1Table() {
        this.table = new HashMap<TokenType, HashMap<TokenType, TokenType[]>>();

        //select(Simpleblock-> {Sequence}) = {{}
        TokenType[] BP1 = {TokenType.LBRACKET, TokenType.Sequence, TokenType.RBRACKET};
        this.addItem(TokenType.Simpleblock, TokenType.LBRACKET, BP1);

        //select(Sequence -> AssignmentStatement    Sequence) = {IDENTIFIER}
        TokenType[] SP1 = {TokenType.assignmentStatement, TokenType.Sequence};
        this.addItem(TokenType.Sequence, TokenType.IDENTIFIER, SP1);

        //select(Sequence -> epsilon) = {}}
        TokenType[] SP2 = {TokenType.Epsilon};
        this.addItem(TokenType.Sequence, TokenType.RBRACKET, SP2);

        /***********************begin***************************/
        /***********************end***************************/
    }

    private void addItem(TokenType row, TokenType column, TokenType[] list) {
        HashMap<TokenType, TokenType[]> map;
        map = this.table.get(row);
        if(map == null) map = new HashMap<TokenType, TokenType[]>();
        map.put(column, list);
        this.table.put(row, map);
    }

    public TokenType[] getItem(TokenType row, TokenType column){
        HashMap<TokenType, TokenType[]> tmp = this.table.get(row);
        if(tmp == null) return null;
        TokenType[] list = tmp.get(column);
        return list;
    }

    public String toString() {
        StringBuffer buffer = new StringBuffer();
        for(TokenType row : this.table.keySet()) {
            for(TokenType column : this.table.get(row).keySet()) {
                buffer.append("(" + row + "," + column + ") = " + this.getItem(row,column));
                buffer.append("\n");
            }
        }
        return buffer.toString();
    }
}
```

3. 新建 LL1.java。LL1 类中的 lexer 用来指定词法分析器，可以使用 lexer.nextToken()函数获取下一个 token 并存储在 lookAhead 中，stack 用来存储分析栈的内容。请在/**begin**/和/**end**/之间补充完整的 LL1 预测分析方法，并输出预测分析的步骤，具体输出结果见后面。

```
package lab2;
import java.util.*;
import lab1.*;

public class LL1 {
    private BlockLexer lexer = null;
    private Token lookAhead = null;
    private Stack<TokenType> stack;
    private LL1Table table = null;
    public LL1() {
        this.table = new LL1Table();
    }
    public void doParse(String filePath){
        this.stack = new Stack<TokenType>();
        this.lexer = new BlockLexer(filePath);
        this.parse();
    }
    public void parse() {
        /****************begin******************/
        /****************end******************/
    }

    private String array2String(TokenType[] product) {
        String ret = "";
        for(TokenType type : product) {
            ret += type + ",";
        }
        ret = ret.substring(0, ret.length() - 1);
        return ret;
    }
    public boolean isTerminal(TokenType type) {
        if(type.compareTo(TokenType.EOF) <= 0)
            return true;
        else
            return false;
    }
}
```

4.新建 Lab2Main.java，内容如下：

在 test 文件夹下新建两个文件 lab2test1.txt 和 lab2test2.txt，分别存储实验要求当中的两个程序，并分别进行测试。

```
package lab2;
public class Lab2Main {
    public static void main(String[] args) {
        LL1 parser = new LL1();
        parser.doParse("test/lab2test1.txt");
    }
}
```

其中针对 lab2test1.txt 文件的测试结果如下：(列之间以\t 分隔，EOF 相当于课本中的#)

```
步骤 分析栈      当前符号  动作
1    [EOF, Simpleblock]      LBRACKET  LBRACKET,Sequence,RBRACKET
2    [EOF, RBRACKET, Sequence, LBRACKET]  LBRACKET  LBRACKET 匹配
3    [EOF, RBRACKET, Sequence]  IDENTIFIER assignmentStatement,Sequence
4    [EOF, RBRACKET, Sequence, assignmentStatement]  IDENTIFIER IDENTIFIER,ASSIGN,Expression,SEMICOLON
5    [EOF, RBRACKET, Sequence, SEMICOLON, Expression, ASSIGN, IDENTIFIER]  IDENTIFIER IDENTIFIER 匹配
6    [EOF, RBRACKET, Sequence, SEMICOLON, Expression, ASSIGN]   ASSIGN      ASSIGN 匹配
7    [EOF, RBRACKET, Sequence, SEMICOLON, Expression]       IDENTIFIER Term,Expression_1
......
31   [EOF, RBRACKET, Sequence]  RBRACKET  Epsilon
32   [EOF, RBRACKET] RBRACKET  RBRACKET 匹配
33   [EOF] EOF   success
```

五、　实验总结

实验三 LR 语法分析实验

实验学时：2

实验类型：（验证、√综合、设计）

一、 实验目的

掌握利用 LR 分析其进行语法分析的方法

二、 实验环境

Jdk1.8 + Eclipse

三、 实验要求

本次实验必须在实验一词法分析正确完成的情况下才能进行，要求能分析简单的算数表达式：

1. init + rate * time
2. (a + b) * c

注意：受限于 LR 分析表，本次实验仅分析具有加乘及小括号的算数表达式，后面没有分号。

四、 实验步骤

1. 文法及 LR 分析表

文法：

0) S’ -> E
1) E -> E + T
2) E -> T
3) T -> T * F
4) T -> F
5) F -> (E)
6) F -> id

状态	ACTION					GOTO			
	id	+	*	(	)	#	E	T	F
0	S5			S4			1	2	3
1		S6				acc			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5	R6		R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

2. 添加 Production.java 用于定义产生式类，代码如下（无需修改）：

```
package lab3;
import lab1.TokenType;

public class Production {
	TokenType left;
	TokenType[] right;
	public Production(TokenType left, TokenType[] right) {
		this.left = left;
		this.right = right;
	}
	public TokenType getLeft() {
		return this.left;
	}
	public TokenType[] getRight() {
		return this.right;
	}
}
```

3. 添加文法类 Grammar.java 并根据文法补全/**begin***/和/**end***/之间的内容。

```
package lab3;
import java.util.*;
import lab1.TokenType;
public class Grammar {
	public static TokenType startSymbol;
	public static HashMap<Integer,Production> productions;
	static {
		startSymbol = TokenType.Expression;
		productions = new HashMap<Integer,Production>();

		Production p = null;
		//(0)S'->E
		TokenType[] S1 = {TokenType.Expression};
		p = new Production(TokenType.Start, S1);
		productions.put(0, p);

		//(1)E->E + T
		TokenType[] E1 = {TokenType.Expression, TokenType.PLUS, TokenType.Term};
		p = new Production(TokenType.Expression, E1);
		productions.put(1, p);

		//(2)E->T
		TokenType[] E2 = {TokenType.Term};
		p = new Production(TokenType.Expression, E2);
		productions.put(2, p);

		/************************begin******************************/
		/************************end******************************/
	}
}
```

4. 添加 LR 表项中的内容类 LRTableEntry.java，代码如下（无需修改）。

注意成员变量 action 可以取 s,r,g,a 四种，分别表示移进、归约、goto 以及 acc

```
package lab3;
public class LRTableEntry {
	private char action; //'s', shift;   'r',reduce;   'g',goto;    'a',acc
	                     //if action = acc, state will be ignored
	private int state;   //state for shift and goto, product No. for reduce
	public LRTableEntry(char action, int state) {
		this.action = action;
		this.state = state;
	}
	public char getAction() {
		return this.action;
	}
	public int getState() {
		return this.state;
	}
	public String toString() {
		if('a' == this.action) return "acc";
		if('g' == this.action) return this.state + "";
		return this.action + "" + this.state;
	}
}
```

5. 添加 LR 分析表类 LRTable.java，并根据 LR 分析表补全/**begin***/和/**end***/之间的内容。

```
package lab3;
import java.util.*;
import lab1.TokenType;

public class LRTable {
	private static HashMap<Integer, HashMap<TokenType, LRTableEntry>> table = null;

	static {
		table = new HashMap<Integer, HashMap<TokenType, LRTableEntry>>();

		addItem(0, TokenType.IDENTIFIER, new LRTableEntry('s',5));
		addItem(0, TokenType.LPAREN, new LRTableEntry('s',4));
		addItem(0, TokenType.Expression, new LRTableEntry('g',1));
		addItem(0, TokenType.Term, new LRTableEntry('g',2));
		addItem(0, TokenType.Factor, new LRTableEntry('g',3));

		addItem(1, TokenType.PLUS, new LRTableEntry('s',6));
		addItem(1, TokenType.EOF, new LRTableEntry('a',0));

		addItem(2, TokenType.TIMES, new LRTableEntry('s',7));
		addItem(2, TokenType.PLUS, new LRTableEntry('r',2));
		addItem(2, TokenType.RPAREN, new LRTableEntry('r',2));
		addItem(2, TokenType.EOF, new LRTableEntry('r',2));

		/************************begin******************************/
		/************************end******************************/
	}

	private static void addItem(int row, TokenType column, LRTableEntry entry) {
		HashMap<TokenType, LRTableEntry> tmp = null;
		tmp = table.get(row);
		if(tmp == null) tmp = new HashMap<TokenType, LRTableEntry>();
		tmp.put(column, entry);
		table.put(row, tmp);
	}

	public static LRTableEntry get(int row, TokenType column) {
		HashMap<TokenType, LRTableEntry> tmp = null;
		tmp = table.get(row);
```

```
        if(tmp == null) return null;
        return tmp.get(column);
    }
}
```

6. 添加 LR 分析类 LR.java，补全/**begin***/和/**end***/之间的内容实现 LR 分析，并输出分析结果（参见后面）。

```
package lab3;
import java.util.*;
import lab1.*;
public class LR {
    private BlockLexer lexer = null;
    private Token lookAhead = null;
    private Stack<Integer> stateStack;
    private Stack<TokenType> symbolStack;

    public void doParse(String filePath){
        this.stateStack = new Stack<Integer>();
        this.symbolStack = new Stack<TokenType>();
        this.lexer = new BlockLexer(filePath);
        this.parse();
    }

    public void parse() {
    /****************begin*******************/
    /****************end********************/
    }
}
```

7. 添加 Lab3Main.java，在 test 文件夹中添加以下 lab3test1.txt 和 lab3test2.txt，测试 LR 分析结果是否正确。

```
package lab3;
public class Lab3Main {
    public static void main(String[] args) {
        LR parser = new LR();
        parser.doParse("test/lab3test1.txt");
    }
}
```

针对 lab3test1.txt 的测试结果如下：

```
步骤 状态栈    符号栈    当前符号  Action    Goto
1    [0]    [EOF] IDENTIFIER s5
2    [0, 5] [EOF, IDENTIFIER]       PLUS r6     3
3    [0, 3] [EOF, Factor]      PLUS r4     2
4    [0, 2] [EOF, Term] PLUS r2     1
5    [0, 1] [EOF, Expression] PLUS s6
6    [0, 1, 6]     [EOF, Expression, PLUS]       IDENTIFIER s5
7    [0, 1, 6, 5]  [EOF, Expression, PLUS, IDENTIFIER]       TIMES      r6     3
8    [0, 1, 6, 3]  [EOF, Expression, PLUS, Factor]     TIMES      r4     9
9    [0, 1, 6, 9]  [EOF, Expression, PLUS, Term]       TIMES      s7
10   [0, 1, 6, 9, 7]      [EOF, Expression, PLUS, Term, TIMES]      IDENTIFIER s5
11   [0, 1, 6, 9, 7, 5]   [EOF, Expression, PLUS, Term, TIMES, IDENTIFIER]   EOF   r6     10
12   [0, 1, 6, 9, 7, 10]  [EOF, Expression, PLUS, Term, TIMES, Factor]  EOF   r3     9
13   [0, 1, 6, 9]  [EOF, Expression, PLUS, Term]       EOF   r1     1
14   [0, 1] [EOF, Expression] EOF   acc
```

五、　实验总结

实验四 基于 S 翻译模式的语义计算

实验学时：2

实验类型：（验证、 √综合、设计）

一、 实验目的

掌握在 LR 分析的基础上嵌入语义动作进行语法制导翻译的方法

二、 实验环境

Jdk1.8 + Eclipse

三、 实验要求

本次实验必须在实验三 LR 语法分析正确完成的情况下才能进行，要求能分析一条简单的表达式，并生成相应的三地址代码。

1. init + rate * time
2. (a + b) * c

注意：受限于 LR 分析表，本次实验仅分析具有加乘及小括号的算数表达式，后面没有分号。

四、 实验步骤

1. 属性文法，其中 newTemp()表示生成一个新的临时变量，||表示串的连接操作。

0)	S’ -> E	{}

1)	E -> E1 + T	{E.place = newTemp();
		E.code = E1.code || T.code || E.place “=” E1.place “+” T.place;}

2)	E -> T	{E.place = T.place; E.code = T.code;}

3)	T -> T1 * F	{T.place = newTemp();
		T.code = T1.code || F.code || T.place “=” T1.place “*” F.place}

4)	T -> F	{T.place = F.place; T.code = F.code;}
5)	F -> (E)	{F.place = E.place; F.code = E.code;}
6)	F -> id	{F.place = id.lexeme; F.code=””}

2. 新建属性类 Attributes.java，代码如下，无需修改。

```
package lab4;
public class Attributes{
	private String place;
	private String code;
	public Attributes() {}
	public Attributes(String name, String code) {
		this.place = name;
		this.code = code;
	}
	public void setName(String name) {
		this.place = name;
	}
	public void setCode(String code) {
		this.code = code;
	}
	public String getName() {
		return this.place;
	}
	public String getCode() {
		return this.code;
	}
	public String toString() {
		return "[" + this.place + "," + this.code + "]";
	}
}
```

3. 新建三地址代码生成类 LRTAC.java，在实验四 LR.java 的基础上进行修改，补充完整/**begin**/和/**end**/之间的代码，能够将算数表达式翻译为三地址代码。其中成员变量 ATCStack 作为分析中的值栈存放变量名称(place)以及生成的三地址代码(code)。函数 newTemp 用于生成临时变量 T1，T2 等。

```
package lab4;
import lab1.*;
import lab3.*;
import java.util.*;

public class LRTAC {
	private BlockLexer lexer = null;
	private Token lookAhead = null;
	private Stack<Integer> stateStack;
	private Stack<TokenType> symbolStack;
	private Stack<Attributes> ATCStack;
	private static int CNT;
	public void doParse(String filePath){
		this.stateStack = new Stack<Integer>();
		this.symbolStack = new Stack<TokenType>();
		this.ATCStack = new Stack<Attributes>();
		this.lexer = new BlockLexer(filePath);
		this.CNT = 0;
		this.parse();
	}
	//生成临时变量
	private String newTemp() {
		CNT++;
		return "T" + CNT;
	}
	/****************begin*******************/
	public void parse() {
	}
	/****************end*********************/
}
```

4. 添加 Lab4Main.java，测试 lab3test1.txt 和 lab3test2.txt 的分析结果是否正确。

```
package lab4;
public class Lab4Main {
	public static void main(String[] args) {
		LRTAC parser = new LRTAC();
		parser.doParse("test/lab3test1.txt");
	}
}
```

针对 lab3test1.txt 的测试结果如下：

```
	T1=rate*time
	T2=init+T1
```

五、 实验总结

实验五 基于 L 翻译模式的中间代码生成

实验学时：2

实验类型：（验证、√综合、设计）

一、 实验目的

掌握利用递归下降分析的同时进行基于 L 翻译模式的中间代码生成。

二、 实验环境

Jdk1.8 + Eclipse

三、 实验要求

本次实验必须在实验一词法分析正确完成的情况下才能进行，要求能将如下两个 SimpleBlock 程序正确翻译为三地址代码。

```
/*comment lines
* until here */
{
        if(a>b && c>d || e>f){
                x = x - 1 - y;
        }else{
                x = x + 2;
        }
}
```

```
{
    //define two variables m and n
    m = 12; n = 21;
    if(m<n){
        t=m; m=n; n=t;
    }
    r = m%n;
    while(r!=0){m=n; n=r; r=m%n;}
}
```

四、 实验步骤

1、SimpleBlock 语言的 LL（1）文法（其中红色部分文法需要同学们自己编程实现）

```
Simpleblock             → { sequence }
Sequence                →  assignmentStatement sequence
                         | ifStatement sequence
                         | whileStatement sequence
                         | ε
assignmentStatement     →  IDENTIFIER = expression ;
Expression              →  term expression_1
Expression_1            →  + term expression_1 | - term expression_1 |  ε
Term                    →  factor term_1
Term_1                  → * factor term_1 | / factor term_1 | % factor term_1 | ε
Factor                  →  ( expression ) | IDENTIFIER | NUMBER_LITERAL
whileStatement          →  while ( boolexpression ) { sequence }
ifStatement             →  if ( boolexpression ) { sequence } OptionalElse
OptionalElse            →  else { sequence } | ε
Boolexpression          →  boolterm boolexpression_1
Boolexpression_1        →  OR boolterm boolexpression_1 | ε
Boolterm                →  boolfactor boolterm_1
Boolterm_1              →  AND boolfactor boolterm_1 | ε
Boolfactor              →  true | false
                         | relationalExpression
relationalExpression    →  expression relationalOperator expression
relationalOperator      →  < | > | <= | >= | == | !=
```

2、文法的 select 集合（若有错误请同学们自行改正）

产生式	select
Simpleblock → { sequence }	{
Sequence→ assignmentStatement sequence	ID
Sequence→ ifStatement sequence	if
Sequence→ whileStatement sequence	while
Sequence→ ε	}
assignmentStatement → IDENTIFIER = expression ;	ID
Expression → term expression_1	(ID NUM
Expression_1 → + term expression_1	+
Expression_1 → - term expression_1	-
Expression_1 → ε	;) < > <= >= == != && ||
Term → factor term_1	(ID NUM
Term_1 → * factor term_1	*
Term_1 → / factor term_1	/
Term_1 → % factor term_1	%
Term_1 → ε	+ - ;) < > <= >= == != && ||
Factor → (expression)	(
Factor → IDENTIFIER	ID
Factor → NUMBER_LITERAL	NUM
whileStatement → while (boolexpression) { sequence }	while
ifStatement → if (boolexpression) { sequence } OptionalElse	if
OptionalElse → else { sequence }	else
OptionalElse → ε	ID if while }
Boolexpression → boolterm boolexpression_1	TRUE FALSE (ID NUM
Boolexpression_1 → OR boolterm boolexpression_1	OR
Boolexpression_1 → ε	)
Boolterm → boolfactor boolterm_1	TRUE FALSE (ID NUM
Boolterm_1 → AND boolfactor boolterm_1	AND
Boolterm_1 → ε	) OR
Boolfactor → true	TRUE
Boolfactor → false	FALSE
Boolfactor → relationalExpression	(ID NUM
relationalExpression → expression relationalOperator expression	(ID NUM
relationalOperator → <	<
relationalOperator → >	>
relationalOperator → <=	<=
relationalOperator → >=	>=
relationalOperator → ==	==
relationalOperator → !=	!=

3. 语法 L-翻译模式

注意若产生式中有相同的符号，则用下角标以示区别（大小写请忽略）。

例如 Expression_1 → + term expression_1$_a$

另外 makelist 函数用来生成只含一个节点的链表，merge 函数用来合并两个链表，backpatch 函数用于控制语句的代码回填，具体请参见课本 217 页。

gen 函数用于将三地址代码存入 TACList 列表，newTemp()函数用来生成新的临时变量，nextstm 返回下一条将要产生的三地址代码的编号。

L-翻译模式如下：

Simpleblock -> { sequence }{gen(halt) ; print(TACList);}
Sequence-> assignmentStatement sequence
Sequence-> ifStatement sequence
Sequence-> whileStatement sequence
Sequence-> ε
assignmentStatement -> IDENTIFIER = expression ; {gen(IDENTIFIER.lexeme "=" expression.name) ;}
Expression -> term {expression_1.in = term.name;}
expression_1 {expression.name = expression_1.name;}
Expression_1->+ term
{expression_1$_a$.in = newTemp(); gen(expression_1$_a$.in "=" expression_1.in "+" term.name);}
expression_1$_a$ {expression_1.name = expression_1$_a$.name;}
Expression_1-> - term
{expression_1$_a$.in = newTemp(); gen(expression_1$_a$.in "=" expression_1.in "-" term.name);}
expression_1$_a$ {expression_1.name = expression_1$_a$.name;}
Expression_1 -> ε {expression_1.name = expression_1.in;}
Term-> factor {term_1.in = factor.name;}
term_1 {term.name = term_1.name;}
Term_1->* factor {term_1$_a$.in = newTemp(); gen(term_1$_a$.in "=" term_1.in "*" factor.name);}
term_1$_a$ {term_1.name = term_1$_a$.name;}
Term_1 -> / factor {term_1$_a$.in = newTemp(); gen(term_1$_a$.in "=" term_1.in "/" factor.name);}
term_1$_a$ {term_1.name = term_1$_a$.name;}
Term_1-> % factor {term_1$_a$.in = newTemp(); gen(term_1$_a$.in "=" term_1.in "%" factor.name);}
term_1$_a$ {term_1.name = term_1$_a$.name;}
Term_1-> ε {term_1.name = term_1$_a$.in;}
Factor-> (expression) {factor.name = expression.name;}
Factor-> IDENTIFIER {factor.name = id.lexeme;}
Factor-> NUMBER_LITERAL {factor.name = num.lexeme;}
whileStatement ->while ({whileBegin = makelist(nextstm);}
boolexpression) {backpatch(boolexpression.trueList, nextstm);}
{ sequence } {gen("goto" + whileBegin);
backpatch(boolexpression.falseList, nextstm);}
ifStatement -> if (boolexpression) {backpatch(boolexpression.trueList, nextstm);}
{ sequence } {optionalElse.in.falseList = boolexpression.falseList;}
OptionalElse
OptionalElse-> else {elseNext = makelist(nextstm);
gen("goto");
backpatch(optionalElse.in.falseList, nextstm);}
{ sequence }
{backpatch(elseNext , nextstm);}
OptionalElse-> ε {backpatch(optionalElse.in.falseList, nextstm);}
Boolexpression -> boolterm {boolexpression_1.in.trueList = boolterm.trueList;
boolexpression_1.in.falseList = boolterm.falseList;}
boolexpression_1 {boolexpression.trueList = boolexpression_1.trueList;
boolexpression.falseList = boolexpression_1.falseList;}
Boolexpression_1->
OR {backpatch(boolexpression_1.in.falseList, nextstm);}
boolterm {boolexpression_1$_a$.in.trueList = merge(boolterm.trueList , boolexpression_1.in.trueList);
boolexpression_1$_a$.in.falseList = bollterm.falseList;}
boolexpression_1$_a$ {boolexpression_1.trueList = boolexpression_1$_a$.trueList;
boolexpression_1.falseList = boolexpression_1$_a$.falseList ;}
Boolexpression_1 -> ε {boolexpression_1.trueList = boolexpression_1.in.trueList;
boolexpression_1.falseList = boolexpression_1.in.falseList;}
Boolterm ->boolfactor {boolterm_1.in.trueList = boolfactor.trueList;

boolterm_1.in.falseList = boolfactor.falseList;}

boolterm_1 {boolterm.trueList = boolterm_1.trueList;
boolterm.falseList = boolterm_1.falseList;}

Boolterm_1-> AND {backpatch(boolterm_1.in.trueList , nextstm);}
boolfactor {boolterm_1$_a$.in.trueList = boolfactor.trueList;
boolterm_1$_a$.in.falseList = merge(boolfactor.falseList , boolterm_1.in.falseList);}
boolterm_1$_a$ {boolterm_1.trueList = boolterm_1$_a$.trueList ;
boolterm_1.falseList = boolterm_1$_a$.falseList ;}

Boolterm_1 -> ε {boolterm_1.trueList = boolterm_1.in.trueList;
boolterm_1.falseList = boolterm_1.in.falseList;}

Boolfactor-> true {Boolfactor.trueList= makelist(nextstm);
gen(“goto”);}

Boolfactor -> false {Boolfactor.falseList = makelist(nextstm);
gen(“goto”);}

Boolfactor ->relationalExpression
{boolfactor.trueList = relationalExpression.trueList;
boolfactor.falseList = relationalExpression.falseList;}

relationalExpression -> expression$_a$ relationalOperator expression$_b$
{relationalExpression.trueList = makelist(nextstm);
relationalExpression.falseList = makelist(nextstm + 1);
gen(“if” expression$_a$.name relationOperator.op expression$_b$.name “goto”);
gen(“goto”)}

relationalOperator -> < | > | <= | >= | == | != {relationOperator.op= token.lexeme;}

4. 添加包 lab5，并新建 AddressList.java，文件内容如下（无需修改）

```
package lab5;
import java.util.*;
public class AddressList {
	public ArrayList<Integer> trueList;
	public ArrayList<Integer> falseList;
	public AddressList() {
		this.trueList = null;
		this.falseList = null;
	}
}
```

5. 新建 RecursionDescendParser.java，文件内容如下。请根据文法及翻译模式，补全/**begin**/和/**end**/之前的代码。

RecursionDescendParser 类采用不带回溯的递归下降子程序方法进行预测分析，关于递归下降子程序的方法同学们可以参考（清华第三版）课本 87 页例 4.12。

递归下降子程序方法为每一个非终结符编写一段函数。其中 lookAhead 成员变量存储输入串中目前待匹配的词法单位记号，matchToken 方法匹配词法单位，若匹配成功则 lookAhead 继续读入下一个待匹配符号，否则匹配失败程序报错。为了程序调试方便，在 matchToken 方法中增加了一个 functionName 参数，当发生错误时，可以快速定位到底是在哪个函数中发生了错误匹配。

parsingError 方法输出语法错误信息，并定位出错的行和列。

在每一个非终结符的函数体中，首先查看 lookAhead 属于该非终结符哪一个产生式的 first 集合，然后就选用相应的产生式进行分析；另外若 first 集合中含有空，则查看 lookAhead 是否属于该非终结符的 follow 集合，若是，则自动匹配；否则报错。

语法制导翻译

因为语法分析采用 LL 分析，因此语法制导翻译采用 L-翻译模式。相关内容参考课本（清华第 3 版）174 页 7.2.3。为此，我们需要改造递归下降语法分析程序为**递归下降语义计算程序**或**递归下降翻译程序**，改造方法为：假设已经为非终结符 A 构造了一个分析子函数。现在，以 A 的每个继承属性为形参，以 A 的综合属性为返回值：

✧ 若遇到一个终结符 X，若 matchToken 匹配成功，则将其综合属性 x 的值保存至专为 X.x 而声明的变量中；

- 若遇到一个非终结符B，利用对应于B的子函数ParseB产生的赋值语句c=ParseB(b1,b2…)，其中的参数b1,b2…对应B的各继承属性，变量c对应B的综合属性。若有多个综合属性，则可以使用记录类型的变量。
- 若遇到一个语义动作集合，则直接复制其中每一语义动作所对应的代码，只是需要注意将属性的访问替换为相应变量的访问。

```
package lab5;
import lab1.*;
import java.util.*;
/**
 * SimpleBlock 语言的递归下降分析器.
 */
public class RecursionDescendParser {
    private BlockLexer lexer = null;
    private Token lookAhead = null;
    private static int CNT;
    private ArrayList<String> TACList;  //存放 TAC 的列表

    public RecursionDescendParser() {}

    public void doParse(String filePath){
        lexer = new BlockLexer(filePath);
        CNT = 0;
        TACList = new ArrayList<String>();
        this.parse();
    }

    private void printTAC() {
        for(int i=0; i<this.TACList.size(); i++) {
            System.out.println(i + ":" + this.TACList.get(i));
        }
    }

    //创建只有一个节点的链表
    private ArrayList<Integer> makeList(int index){
        ArrayList<Integer> list = new ArrayList<Integer>();
        list.add(index);
        return list;
    }

    //将两个链表合并成一个
    private ArrayList<Integer> merge(ArrayList<Integer> p1, ArrayList<Integer> p2){
        ArrayList<Integer> list = new ArrayList<Integer>();
        if(p1 != null) list.addAll(p1);
        if(p2 != null) list.addAll(p2);
        return list;
    }
    //回填
    private void backPatch(ArrayList<Integer> list, int value) {
        if(list == null) return;
        for(int item : list) {
            if(item >= this.TACList.size()) {
    System.out.println("backpatch error, found illegal pointer:" + item + "with value="+ value);
    continue;
            }
            String code = this.TACList.get(item);
            code = code + " " + value;
            this.TACList.set(item, code);
        }
    }

    private Token matchToken(TokenType type, String functionName){
        if(lookAhead.getType() != type){
            parsingError(type.toString(), functionName);
        }
        Token matchedSymbol = lookAhead;
        lookAhead = lexer.nextToken();
        return matchedSymbol;
    }

    private void parsingError(String types, String functionName){
        printTAC();
        System.out.println("Parsing Error! in " + functionName);
        System.out.println("encounter " + lookAhead.getLexeme());
        System.out.println("at line " + lookAhead.getLine() + ",column " +
lookAhead.getColumn());
        System.out.println("while expecting " + types);
        System.exit(1);
    }

    /**
     * 调用开始符号对应的方法，进行语法分析。
     * @return 返回分析是否成功。
     */
    private void parse() {
        lookAhead = lexer.nextToken();
        simpleblock();
        printTAC();
    }

    /**
     * simpleblock = LBRACE sequence RBRACE
     * B   -> { S }
     */
    private void simpleblock() {
        if(lookAhead.getType() == TokenType.LBRACKET){
            matchToken(TokenType.LBRACKET, "simpleblock");
            sequence();
            this.TACList.add("halt");
            matchToken(TokenType.RBRACKET, "simpleblock");
        }else{
            parsingError(TokenType.LBRACKET.toString(), "simpleblock");
        }
    }

    /**
     * sequence = assignmentStatement sequence |
     *                  ifStatement sequence |
     *                  whileStatement sequence |
     *                  epsilon
     * S -> AS | IS | WS | ε
     */
    private void sequence(){
        if(lookAhead.getType() == TokenType.IDENTIFIER){
            assignmentStatement();
            sequence();
        }else if(lookAhead.getType() == TokenType.KEY_IF){
            ifStatement();
            sequence();
        }else if(lookAhead.getType() == TokenType.KEY_WHILE){
            whileStatement();
            sequence();
        }else if(lookAhead.getType() == TokenType.RBRACKET){
            //match epsilon
        }else{
            String errorTypes = TokenType.IDENTIFIER.toString() + "," +
                      TokenType.RBRACKET.toString();
            parsingError(errorTypes, "sequence");
        }
    }

    /**********************begin**********************/
    //whileStatement → while ( boolexpression ) { sequence }
    private void whileStatement() {
    }

    //ifStatement  →   if ( boolexpression ) { sequence } OptionalElse
    private void ifStatement() {
    }

    //OptionalElse →   else { sequence }
    //OptionalElse →   ε
    private void optionalElse(AddressList inh) {
    }

    //Boolexpression   →   boolterm boolexpression_1   select=TRUE   FALSE   (   ID   NUM
    private AddressList boolexpression() {
    return null;
    }

    //Boolexpression_1   →   OR boolterm boolexpression_1                 select=OR
    //Boolexpression_1   →   ε                                            select=)
    private AddressList boolexpression_1(AddressList inh) {
        return null;
    }
    //Boolterm      →   boolfactor boolterm_1      select=TRUE   FALSE   (   ID   NUM
    private AddressList boolterm() {
        return null;
    }

    //Boolterm_1 →   AND boolfactor boolterm_1        select = AND
```

```
    //Boolterm_1    →   ε                               select = )    OR
    private AddressList boolterm_1(AddressList inh) {
        return null;
    }

    //Boolfactor     →   true                       select = TRUE
    //Boolfactor     →   false                      select = FALSE
    //Boolfactor     →   relationalExpression    select = (      ID      NUM
    private AddressList boolfactor() {
        return null;
    }

//relationalExpression → expression relationalOperator expression    select = (      ID      NUM
    private AddressList relationalExpression() {
        return null;
    }

    //relationalOperator      →   <        select = <
    //relationalOperator      →   >        select = >
    //relationalOperator      →   <=       select = <=
    //relationalOperator      →   >=       select = >=
    //relationalOperator      →   ==       select = ==
    //relationalOperator      →   !=       select = !=
    private String relationalOperator() {
        return null;
    }

    /**********************end***********************/

    /**
     * assignmentStatement = IDENTIFIER ASSIGN expression SEMICOLON
     * A -> id = E;
     */
    private void assignmentStatement(){
        if(lookAhead.getType() == TokenType.IDENTIFIER){
            Token id = matchToken(TokenType.IDENTIFIER, "assignmentStatement");
            matchToken(TokenType.ASSIGN, "assignmentStatement");
            String eName = expression();
            matchToken(TokenType.SEMICOLON, "assignmentStatement");
            this.TACList.add(id.getLexeme() + "=" + eName);
        }else{
            String errorTypes = TokenType.IDENTIFIER.toString();
            parsingError(errorTypes, "assignmentStatement");
        }
    }

    /**
     * expression = term expression_1
     * E -> TE'
     * @return
     */
    private String expression(){
        if(lookAhead.getType() == TokenType.IDENTIFIER
            || lookAhead.getType() == TokenType.LPAREN
            || lookAhead.getType() == TokenType.NUMBER_LITERAL){
            String tName = term();
            String eName = expression_1(tName);
            return eName;
        }else{
            String errorTypes = TokenType.IDENTIFIER.toString()
                    + "," + TokenType.NUMBER_LITERAL.toString()
                    + "," + TokenType.LPAREN.toString();
            parsingError(errorTypes, "expression");
            return null;
        }
    }

    /**
     * expression_1 = PLUS term expression_1 |        select = +
     *         MINUS term expression_1 |              select = -
     *         epsilon                           select = ;   )   <   >   <=   >=    ==   != && ||
     * E' -> +TE' | -TE' | ε
     */
    private String expression_1(String inh){
        if(lookAhead.getType() == TokenType.PLUS){
            matchToken(TokenType.PLUS, "expression_1");
            String tName = term();
            String e1Inh = this.newTemp();
            this.TACList.add(e1Inh + "=" + inh + "+" + tName);
            String e1Syn = expression_1(e1Inh);
            return e1Syn;
        }else if(lookAhead.getType() == TokenType.MINUS){
            matchToken(TokenType.MINUS, "expression_1");
            String tName = term();
            String e1Inh = this.newTemp();
            this.TACList.add(e1Inh + "=" + inh + "-" + tName);
            String e1Syn = expression_1(e1Inh);
            return e1Syn;
        }else if(lookAhead.getType() == TokenType.SEMICOLON
                || lookAhead.getType() == TokenType.RPAREN
                || lookAhead.getType() == TokenType.LESS
                || lookAhead.getType() == TokenType.LESS_EQUAL
                || lookAhead.getType() == TokenType.GREATER
                || lookAhead.getType() == TokenType.GREATER_EQUAL
                || lookAhead.getType() == TokenType.EQUAL
                || lookAhead.getType() == TokenType.NOT_EQUAL
                || lookAhead.getType() == TokenType.LOGICAL_AND
                || lookAhead.getType() == TokenType.LOGICAL_OR){
            //match epsilon
            //select = ;   )   <   >   <=   >=    ==   != && ||
            return inh;
        }else{
            String errorTypes = TokenType.PLUS.toString()
                    + "," + TokenType.MINUS.toString()
                    + "," + TokenType.SEMICOLON.toString()
                    + "," + TokenType.LESS.toString()
                    + "," + TokenType.LESS_EQUAL.toString()
                    + "," + TokenType.GREATER.toString()
                + "," + TokenType.GREATER_EQUAL.toString()
                + "," + TokenType.EQUAL.toString()
                + "," + TokenType.NOT_EQUAL.toString()
                + "," + TokenType.LOGICAL_AND.toString()
                + "," + TokenType.LOGICAL_OR.toString();
        parsingError(errorTypes, "expression_1");
        return null;
    }
}
/**
 * term = factor term_1
 * T -> FT'
 */
private String term(){
    if(lookAhead.getType() == TokenType.IDENTIFIER
        || lookAhead.getType() == TokenType.LPAREN
        || lookAhead.getType() == TokenType.NUMBER_LITERAL){
        String fName = factor();
        String tName = term_1(fName);
        return tName;
    }else{
        String errorTypes = TokenType.IDENTIFIER.toString()
                + "," + TokenType.NUMBER_LITERAL.toString()
                + "," + TokenType.LPAREN.toString();
        parsingError(errorTypes, "term");
        return null;
    }
}
/**
 * term_1 = MULT factor term_1 |       select = *
 *         DIV factor term_1 |         select = /
 *         MOD factor term_1 |         select = %
 *         epsilon                     select = +   -   ;   ) <   >   <=   >=    ==   != && ||
 * T' -> *FT' | /FT' | %FT' | ε
 */
private String term_1(String inh){
    if(lookAhead.getType() == TokenType.TIMES){
        matchToken(TokenType.TIMES, "term_1");
        String fName = factor();
        String t1Inh = this.newTemp();
        this.TACList.add(t1Inh + "=" + inh + "*" + fName);
        String t1Syn = term_1(t1Inh);
        return t1Syn;
    }else if(lookAhead.getType() == TokenType.DIVIDE){
        matchToken(TokenType.DIVIDE, "term_1");
        String fName = factor();
        String t1Inh = this.newTemp();
        this.TACList.add(t1Inh + "=" + inh + "/" + fName);
        String t1Syn = term_1(t1Inh);
        return t1Syn;
    }else if(lookAhead.getType() == TokenType.REMAINDER){
        matchToken(TokenType.REMAINDER, "term_1");
        String fName = factor();
        String t1Inh = this.newTemp();
```

```
            this.TACList.add(t1Inh + "=" + inh + "%" + fName);
            String t1Syn = term_1(t1Inh);
            return t1Syn;
        }else if(lookAhead.getType() == TokenType.PLUS
                    || lookAhead.getType() == TokenType.MINUS
                    || lookAhead.getType() == TokenType.SEMICOLON
                    || lookAhead.getType() == TokenType.RPAREN
                    || lookAhead.getType() == TokenType.LESS
                    || lookAhead.getType() == TokenType.LESS_EQUAL
                    || lookAhead.getType() == TokenType.GREATER
                    || lookAhead.getType() == TokenType.GREATER_EQUAL
                    || lookAhead.getType() == TokenType.EQUAL
                    || lookAhead.getType() == TokenType.NOT_EQUAL
                    || lookAhead.getType() == TokenType.LOGICAL_AND
                    || lookAhead.getType() == TokenType.LOGICAL_OR){
            //match epsilon
            //follow(T') = +    -    ;    ) <    >    <=    >=     ==    != && ||
            return inh;
        }else{
            String errorTypes = TokenType.TIMES.toString()
                        + "," + TokenType.DIVIDE.toString()
                        + "," + TokenType.REMAINDER.toString()
                        + "," + TokenType.PLUS.toString()
                        + "," + TokenType.MINUS.toString()
                        + "," + TokenType.RPAREN.toString()
                        + "," + TokenType.SEMICOLON.toString()
                        + "," + TokenType.LESS.toString()
                        + "," + TokenType.LESS_EQUAL.toString()
                        + "," + TokenType.GREATER.toString()
                        + "," + TokenType.GREATER_EQUAL.toString()
                        + "," + TokenType.EQUAL.toString()
                        + "," + TokenType.NOT_EQUAL.toString()
                        + "," + TokenType.LOGICAL_AND.toString()
                        + "," + TokenType.LOGICAL_OR.toString();
            parsingError(errorTypes, "term_1");
            return null;
        }
    }
    /**
     * factor = LPAREN expression RPAREN |
     *                IDENTIFIER |
     *                NUMBER_LITERAL
     * F -> (E) | id | number
     */
    private String factor() {
        if(lookAhead.getType() == TokenType.LPAREN){
            matchToken(TokenType.LPAREN, "factor");
            String eName = expression();
            matchToken(TokenType.RPAREN, "factor");
            return eName;
        }else if(lookAhead.getType() == TokenType.IDENTIFIER){
            Token id = matchToken(TokenType.IDENTIFIER, "factor");
            return(id.getLexeme());
        }else if(lookAhead.getType() == TokenType.NUMBER_LITERAL){
            Token id = matchToken(TokenType.NUMBER_LITERAL, "factor");
            return(id.getLexeme());
        }else{
            String errorTypes = TokenType.LPAREN.toString()
                    + "," + TokenType.IDENTIFIER.toString()
                    + "," + TokenType.NUMBER_LITERAL.toString();
            parsingError(errorTypes, "factor");
            return null;
        }
    }
    private String newTemp() {
        CNT++;
        return "T" + CNT;
    }
}
```

8. 测试输出

新建 Lab5Main.java 文件，代码如下。在 test 文件夹下新建 lab5test1.txt 和 lab5test2.txt 两个文件，内容分别如实验要求所示，测试输出结果。

```
    package lab5;
    public class Lab5Main {
        public static void main(String[] args) {
            RecursionDescendParser parser = new RecursionDescendParser();
            parser.doParse("test/lab5test1.txt");
        }
    }
```

其中 lab5test1.txt 和 lab5test2.txt 的输出结果如下：

```
0:if a>b goto 2
1:goto 4
2:if c>d goto 6
3:goto 4
4:if e>f goto 6
5:goto 10
6:T1=x-1
7:T2=T1-y
8:x=T2
9:goto 12
10:T3=x+2
11:x=T3
12:halt
```

```
0:m=12
1:n=21
2:if m<n goto 4
3:goto 7
4:t=m
5:m=n
6:n=t
7:T1=m%n
8:r=T1
9:if r!=0 goto 11
10:goto 16
11:m=n
12:n=r
13:T2=m%n
14:r=T2
15:goto 9
16:halt
```

五、 实验总结