编译原理课程实验报告

- 语法分析

姓名：焦紫顺

学号：121250063

2014/11/14

目录

1. 目标

本实验是语法分析器的实现，给定一个语法分析表和文法序列，对 Token 序列进行文法推导，得出推导序列，语法分析表由手工完成作为输入，而 Token 序列则来源于上次词法分析实验的词法分析器。主要实现分析程序。

2. 内容与功能描述

本实验工具的实现语言为 Java (SE 1.7)，项目工程 IDE 为 Eclipse 4.3。

目录结构和内容简述如下表。

目录	描述
Document	文档目录
\编译原理实验报告 2.docx	本文档
Input	输入目录
\re.txt	正则表达式定义文件
\grammar.txt	文法产生式定义文件
\form.txt	语法分析表定义文件
\text.txt	待分析文本文件
Output	输出目录
\result.txt	语法分析结果文件(推导序列)
Lex	程序项目目录
\src	源代码目录
\basic	基本数据结构基类
\grammar	语法分析核心类、主函数
\lex	词法分析程序
\structrue	词法分析核心类和算法
\debug	输出调试辅助类
\iohelper	文件输入辅助类

实现功能：

基于输入的预测分析表（LR（1)、LR（0） 或 LALR）和文法序列，给出待测文本的语法推导序列。

3. 原理与设计

设计程序时对分析程序进行组件划分，划分为分析驱动程序(GrammarAnalyzer)、状态栈、

文法符号栈(StateStack)、Token 输入流(TokenStream)、分析预测表(GrammarAlyForm)及分析结果表。

4. 依赖与假设

由于本程序不包含文法分析程序，故需要手工构造预测分析表并依据格式写入文件中，文法序列顺序应与预测分析表有关的文法序列顺序一致。

非终结符号需要和之前的词法分析中定义的词法单元名称一致，词法单元可自定义，详见 re.txt。

5. 数据结构设计

本实验的实现涉及到多种数据结构。所有的详细数据操作实现请参看源代码。

5.1 状态栈

存储状态和待规约的文法符号序列需要运用栈，这里直接利用 Java 内置的栈类。

由于文法符号包括终结符号和非终结符号，为区分两者，可利用继承特性，让 Terminal 类和 Nonterminal 类继承 Sign 基类，利用多态避免繁杂的条件判断。

```
public class StateStack<T> {
    private Stack<T> stateStack;
    /*判断栈空*/
    public boolean isEmpty() {
        return stateStack.empty();
    }

    public void clear() {
        this.stateStack.clear();
    }

    public T getTop() {
        return this.stateStack.peek();
    }

    public void push(T i) {
        this.stateStack.push(i);
    }

    public T pop() {
        return this.stateStack.pop();
    }
```

```
    /*连续出栈count次，返回列表*/
    public List<T> pop(int count) {
        if(this.stateStack.size() < count) {
            count = this.stateStack.size();
        }

        LinkedList<T> l = new LinkedList<T>();
        while(count-- > 0) {
            l.add(this.stateStack.pop());
        }
        return l;
    }
```

详细数据操作实现参看源代码。

5.2 Token 输入流

由于在语法分析中 Token 输入流不需要回退，只需要向前看符号，故输入流类似迭代器，实现迭代器的操作即可，注意要允许多次获取当前的 Token。Token 来自于词法分析器的分析结果，封装了一个单词的细节信息，内有对应的字符串值，便于以后扩展程序时需要获取信息。

```
public class TokenStream {
    private Iterator<Token> tokenItr;
    private Token currentToken;         // 现Token

    public static final String END_SIGN ="$";

    public TokenStream(List<Token> tokenList) {
        addEndSign(tokenList); //为Token序列结尾加入终结符
        this.tokenItr = tokenList.iterator();
        this.currentToken = tokenItr.next();
    }

    public Token getCurrentToken() {
        return this.currentToken;
    }

    public Token goNext() {
        this.currentToken = tokenItr.next();
        return this.currentToken;
    }

    /**为Token序列结尾加入终结符*/
    private void addEndSign(List<Token> tokenList) {
        Token end = new Token();
        end.setType(END_SIGN);
        end.setValue("");
        tokenList.add(end);
```

```
    }
}
```

详细数据操作实现参看源代码。

5.3 分析预测表

在构建预测分析表时，给出查表的功能即可，即 Action 方法和 GOTO 方法，内部可使用数组和 Map 方便查找对应动作。 每一个 State 对应于手工构造的预测分析表中的一行。Action 即动作枚举类以便根据此执行相应动作。

```
/**
 * 语法分析表
 * @author emengjzs
 */
public class GrammarAlyForm {
    // 状态列，下标对应状态序号
    private ArrayList<State> stateList;

    // 根据终结符号给出动作
    public Action getAction(Integer state, Terminal sign) {
        return stateList.get(state).getAction(sign);
    }

    // 根据非终结符号给出跳转状态
    public Integer getGOTO(Integer state, Nonterminal sign) {
        return stateList.get(state).getGOTO(sign);
    }

}
```

```
public class State {
    private Map<String, Action> actionList = new
TreeMap<String, Action>(); // action映射表
    private Map<String, Integer> gotoList = new HashMap<String,
Integer>(); // Goto映射表

    public Action getAction(Terminal t) {
        Action index = this.actionList.get(t.sign);
        if(index != null) {
            return index;
        }
        return new Action();
    }

    public Integer getGOTO(Nonterminal s) {
```

```
        return this.gotoList.get(s.getSign());
    }
}
```

6. 算法设计

6.1 语法分析

分析驱动算法实现上与教材相似，不停地执行循环，根据输入流中当前词法单元和现在状态查预测分析表决定相应动作，直到遇见接受动作为止。

```
        boolean run = true;
        while(run) {
            Integer index = this.stateStack.getTop();
            Terminal searchTerminal = new
Terminal(this.tokenStream.getCurrentToken());
            Action action = this.form.getAction(index,
searchTerminal);
            switch(action.getType()) {
                case MOVE:
                    move(action.getIndex(),  searchTerminal);
                    break;
                case PRODUCE:
                    produce(action.getIndex());
                    break;
                case ACCEPT:
                    info.append(action.toString());
                    run = false;
                    break;
                case ERROR:
                    error();
                    run = false;
                    break;
            }
        }
```

若动作为移入，将 token 对应的终结符压入栈中，并让输入流给出下一个字符，若为规约则根据产生式的长度出栈相应次数，根据 index 输出对应的文法，若为结束，则结束过程，否则提示语法错误。

7. 运行用例

这里只给出部分测试用例信息，详细输入数据请查看 Input 目录并运行程序，运行结果请

查看 Output 目录。

7.1 输入：正则表达式

程序的运行会从分析正则表达式开始，正则表达式包含关键字、标识符、操作符、数值和块符号等，注释不会在语法分析中。

re.txt

```
// 此为注释
// 关键字
KEY_WORDS int float double if else break while do continue for return void char static default
// 一个正则表达一行
// 空白
WHITE (\n|\r|\t|\f|\s)(\n|\r|\t|\f|\s)*
小写字母 (a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z) -1
大写字母 (A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z) -1
数字 (0|1|2|3|4|5|6|7|8|9|0) -1
integer (1|2|3|4|5|6|7|8|9|0)(数字)* use 数字
float (integer).(数字)(数字)* use 数字 integer
id ((小写字母)|(大写字母)|_)((小写字母)|(大写字母)|(数字)|_)* use 小写字母 大写字母 数字
op_cmp ==|>|<|<=|>=|!=
semi ;
+ +
- -
* \*
\ \\
% %
= =
( \(
) \)
{ {
} }
// 注释不会被用作语法分析
COMMIT  //(( 小 写 字 母 )|(\r|\t|\f|\s)|( 大 写 字 母 )|( 数 字 )|(\s|\t|!|@|#|$|%|^|&|\*|\(|\)|+|-|_|.|?|/|,|'|"|[|]|{|}|;|=|<|>))* use 小写字母 大写字母 数字
```

7.2 输入：文法

基本文法，包括表达式、赋值语句、While 循环、条件语句

grammar.txt

```
S->S semi S
S->S semi statement
S->S semi
S->statement
S->semi
statement->IF ( test ) { S }
statement->IF ( test ) { S } ELSE { S }
statement->WHILE ( test ) { S }
statement->id = expr
expr->expr + term
expr->term
term->term * factor
term->factor
factor->( expr )
factor->val
val->id
val->integer
val->float
test->expr op_cmp expr
```

7.3 输入：预测分析表

这里的分析表是 LR(1)分析表，由于表过大，故只给出部分内容。每行分别为状态记号、遇见符、动作（0 移入 1 规约 2 结束）、编号。GOTO 下每一行是状态记号、非终结符号、转移状态。

form.txt

	ACTION						GOTO		
	id	if	while	semi	(	…	S	statement	…
0	S4	S1	S3	S5			2	6	
1					S7				
2				S8					
3					S9				
4									
5				R4					
6				R3					
…….									

7.4 输入：待测文本

text.txt

```
state = (3*5+6)+3.1415;
id=0;
stateStack1=0;
stateStack2=0;
if(stateStack2==0) {
    while(stateStack1<=10) {
          _tmp = 34342 + 1;
        stateStack1 = (3*5+6)+3.1415;
    }
}
else {
    j=314; // This is a comment.
    ;
}
```

7.5 输出：推导序列

因推导过程也很长，所以只给出部分结果。详情看 Output 文件夹下内容。

result.txt

```
......
```

0 2	S	semi	移入：semi
0 2 8	S semi	IF	移入：IF
0 2 8 1	S semi IF	(	移入：(
0 2 8 1 7	S semi IF (	id	移入：id
0 2 8 1 7 16	S semi IF (id	op_cmp	归约 15：val->id
0 2 8 1 7 19	S semi IF (val	op_cmp	归约 14：factor->val
0 2 8 1 7 14	S semi IF (factor	op_cmp	归约 12：term->factor
0 2 8 1 7 18	S semi IF (term	op_cmp	归约 10：expr->term
0 2 8 1 7 13	S semi IF (expr	op_cmp	移入：op_cmp
0 2 8 1 7 13 41	S semi IF (expr op_cmp	integer	移入：integer
0 2 8 1 7 13 41 36	S semi IF (expr op_cmp integer	)	归约 16：val->integer
0 2 8 1 7 13 41 38	S semi IF (expr op_cmp val	)	归约 14：factor->val
0 2 8 1 7 13	S semi IF (expr op_cmp	)	归约 12：term->factor

41 33	factor		
0 2 8 1 7 13 41 37	S semi IF (expr op_cmp term	)	归约 10: expr->term
0 2 8 1 7 13 41 53	S semi IF (expr op_cmp expr	)	归约 18: test->expr op_cmp expr
0 2 8 1 7 12	S semi IF (test	)	移入:)

8. 问题与发现

文法分析器不够灵活，仅能依靠已分析好的预测分析表进行分析，一旦分析表有错误很难知道出错的地方，可考虑为文法自动构建预测分析表。