编译原理语法分析程序实现报告

141250179 袁阳阳

南京大学软件学院

目录

1.目标

本次实验的目的是通过自行定义文法，对编译器进行语法分析的过程进行模拟，加深对语法分析过程的理解。

2.内容描述

本程序使用 java 编写。程序读取一个文本文件，运用实验一的词法分析程序得到 token 序列，再对 token 序列进行语法分析。这里使用 LR（1）的方法进行自底向上分析，最后产生规约序列。报告以 LR(1)文法为例，程序对于 SLR(1)的文法和分析表同样适用，只要在输入文件中给出合法的 parsing table 和上下文无关文法。

3.假设和依赖

3.1 实验环境

操作系统	Windows 10
编译器	Jdk1.8
其他工具	IntelliJ Idea, Git

3.2 其他假设

1）用户确保自己作为输入的 parsing table 和文法都是正确的，并且是非二义性的。
2）输入的程序中的符号包含于词法分析器可分析的符号集合中
3）所有的终结符和非终结符都是单个字符。
4）用户给出的输入文件是合乎系统定义的，具体格式在下文叙述。

4.思路和方法

1）自定义文法
2）基于该文法构造状态机
3）基于状态机和该文法构造分析表
4）基于分析表编写语法分析程序
5）程序编写思路：
a.构建输入带，符号栈，状态栈，建立 parsing table 在程序中的映射，方便查询。

b.通过实验一的词法分析程序获取待分析程序段的 TOKEN 序列，将 TOKEN 序列转换为表达式，依次移动读头，根据当前状态栈栈顶的状态号和读头下的符号查询 parsing table：

b.1 如果查到 Si 则为移点操作，将读头下的符号压入符号栈，将状态号 i 压入状态栈；

b.2 如果查到 ri(i 不等于 0)则为规约操作，将栈顶与文法产生式 i 右端相同的部分全部弹出，同时将状态栈一起弹出，并将产生式 i 左端的非终结符 A 压入符号栈，查当前状态栈栈顶的状态 X 在 GOTO 表中的 GOTO(X,A)=状态 Y，将状态号 Y 压入状态栈；

b.3 如果查到 r0，则语法分析成功；

c.如果查找不到 Action 或者 GOTO 项，则当前表达式不符合所定义的文法，分析失败，终止程序。

d.具体的输入文件格式参见 6.3 分析表，8.运行截图中的 input.txt 和 cfg.txt。

5.相关分析过程描述

5.1 自定义的文法

0. S=E
1. E=E+T
2. E=T
3. T=T*F
4. T=F
5. F=(E)
6. F=i

5.2 构建预测分析表

state	ACTION						GOTO		
	i	+	*	(	)	$	E	T	F
0	S_5			S_4			1	2	3
1		S_6				accept			
2		r_2	S_7		r_2	r_2			
3		r_4	r_4		r_4	r_4			
4	S_5			S_4			8	2	3
5		r_6	r_6		r_6	r_6			
6	S_5			S_4				9	3
7	S_5			S_4					10
8		S_6			S_{11}				
9		r_1	S_7		r_1	r_1			
10		r_3	r_3		r_3	r_3			
11		r_5	r_5		r_5	r_5			

6.重要的数据结构

6.1 输入带

直接用 char 数组来作为输入带，用下标来表示读头的改变

6.2 状态栈和符号栈

使用 java 库提供的 stack 来实现状态栈和符号栈：

```
Stack<Integer> stateSta = new Stack<Integer>();
Stack<String> symbolSta = new Stack<String>();
```

6.3 分析表

分析表以固定的文件格式存储（如下图，0 i%S5|(%S4 表示状态 0 经过 shift 操作通过符号 i 到达状态 5，%%%%以下为 GOTO 的情况，如 0 E%1|T%2|F%3 表示状态 0 通过 E 到达状态 1，通过 T 到达状态 2，通过 F 到达状态 3），同时建立类 state 来对分析表进行查询以及获取下一状态

```
0 i%S5|(%S4
1 +%S6|$%r0
2 +%r2|*%S7|)%r2|$%r2
3 +%r4|*%r4|)%r4|$%r4
4 i%S5|(%S4
5 +%r6|*%r6|)%r6|$%r6
6 i%S5|(%S4|
7 i%S5|(%S4
8 +%S6|)%S11
9 +%r1|*%S7|)%r1|$%r1
10 +%r3|*%r3|)%r3|$%r3
11 +%r5|*%r5|)%r5|$%r5
%%%%
0 E%1|T%2|F%3
1
2
3
4 E%8|T%2|F%3
5
6 T%9|F%3
7 F%10
8
9
10
11
```

7.核心算法

程序主要的方法是：

state.shiftState()——获取当前状态通过读头下的符号要返回的移点状态或者规约情况，返回为 Si 或者 ri

state.gotoState()——用来获取进行规约时，将符号栈和状态栈栈顶的若干元素出栈后，应当到达的新的将被压进状态栈的状态

Analyzer.syntaxAnalyze()——主程序用来调度以上两个函数，进行初始化，压栈，出栈，移动读头等操作

下面是 shiftState()方法：

```
//此方法用来获取当前状态通过读头下的符号要返回的移点状态或者规约情况，返回为 Si 或者 ri
public String shiftState(int init, char input){
        BufferedReader br = null;
        try {
            br = new BufferedReader(new InputStreamReader(
                    new FileInputStream(new File(pptFileName))));
            String line ="";
            while(!((line=br.readLine()).equals("%%%%"))){//%%%%为分析表中 Action 与 GOTO 的分界线
                String[] whole = line.split(" ");
                int nowState = Integer.parseInt(whole[0]);
                if(nowState==init){//找到目前状态
//                    System.out.println(whole[1]);//TEST TEST TEST
                    String[] changeArray = whole[1].split("\\|");
                    for(int i=0;i<changeArray.length;i++){
//                        System.out.println(changeArray[i]);//TEST TEST TEST
                        String[] change = changeArray[i].split("%");//change[0]为要与输入带读头指向的字符相比较的字符,change[1]为即将到达的状态
                        char toCmp = (change[0].toCharArray())[0];
                        if(toCmp==input){//字符与读头指向的字符相符合
                            return change[1];//返回相应的 Si 或者 ri
                        }
                    }
                }
            }
            br.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
```

```
        return null;
    }
```

下面是 gotoState()方法：

```
//  此方法用来获取进行规约时，将符号栈和状态栈栈顶的若干元素出栈后，应当到达的新的将被压进状态栈的状态
    public int gotoState(int initState,int reduceIndex){
        int result=-1;
        BufferedReader brPpt = null;//读 ppt 文件
        String[] product=getProduct(reduceIndex).split("%");
        try {
            brPpt = new BufferedReader(new InputStreamReader(new FileInputStream(new File(pptFileName))));
            String left= product[0];
            String linePpt="";
            while(!(brPpt.readLine().equals("%%%%"))){}//%%%%为分析表中 Action 与 GOTO 的分界线
            while((linePpt = brPpt.readLine())!=null){
                String[] whole = linePpt.split(" ");
                int nowState = Integer.parseInt(whole[0]);
                if(nowState==initState){//找到了对应目前状态的状态
                    String[] change = whole[1].split("\\|");
                    for(int i=0;i<change.length;i++){
                        String toCmp = change[i].split("%")[0];
                        if(toCmp.equals(left)){//找到了 initState 通过 leftCode 要 GOTO 的状态

result=Integer.parseInt(change[i].split("%")[1]);
                            return result;
                        }
                    }
                }

            }
            brPpt.close();

        } catch (IOException e) {
            e.printStackTrace();
        }
        return result;
    }
```

下面是 syntaxAnalyze()方法：

```
public void syntaxAnalyze(char[] inputBelt,String pptFileName,String cfgFileName){
    initialize();
```

```
    State stateOp = new State(pptFileName,cfgFileName);
    String action="";
    for(int i=0;i<inputBelt.length;i++){
        int nowState = stateSta.peek();
        String tmp = stateOp.shiftState(nowState,inputBelt[i]);
        if(tmp==null){
            System.out.println("analyze failed!");
            return;
        }
        char[] op = tmp.toCharArray();
        if(op[0]=='S'){//遇到 S 为移点操作，op[1]为移点后到达的状态
            action = "shift";
            myPrint(inputBelt,i,action);
            stateSta.push(Integer.parseInt(op[1]+""));//新状态进状
态栈
            symbolSta.push(inputBelt[i]+"");//读头下的字符进符号栈
            nowState = stateSta.peek();
        }else{//遇到 r 为规约操作，op[1]为规约要使用的产生式序号
            //accept
            if(Integer.parseInt(op[1]+"")==0){
                action = "accept";
                myPrint(inputBelt,i,action);
                System.out.println("analyze successfully!");
                return;
            }
            String productStr =
stateOp.getProduct(Integer.parseInt(op[1]+""));//规约对应的产生式
            if(productStr==null){
                System.out.println("analyze failed!");
            }
            String[] product = productStr.split("%");
            action = "reduced by
"+op[1]+":"+product[0]+"->"+product[1];
            myPrint(inputBelt,i,action);
            String topElement="";
            int len = product[1].length();
            for(int j=0;j<len;j++){
                topElement=symbolSta.pop()+topElement;//获取栈顶的若
干元素
            }
            if(topElement.equals(product[1])){//比较栈顶的若干元素与
产生式的右边是否相同，相同则进行规约
                for(int k=0;k<len;k++){
                    stateSta.pop();
```

```
                }
                symbolSta.push(product[0]);//将产生式的左边压栈
                int newState =
stateOp.gotoState(stateSta.peek(),Integer.parseInt(op[1]+""));//
获取新状态
                stateSta.push(newState);//将状态压栈
                if(newState==-1){
                    System.out.println("analyze failed!");
                }
                nowState = newState;
                i--;
            }else{
                System.out.println("analyze failed!");
            }

        }
    }
}
```

8.运行截图

Input.txt

```
a*b+c
```

cfg.txt

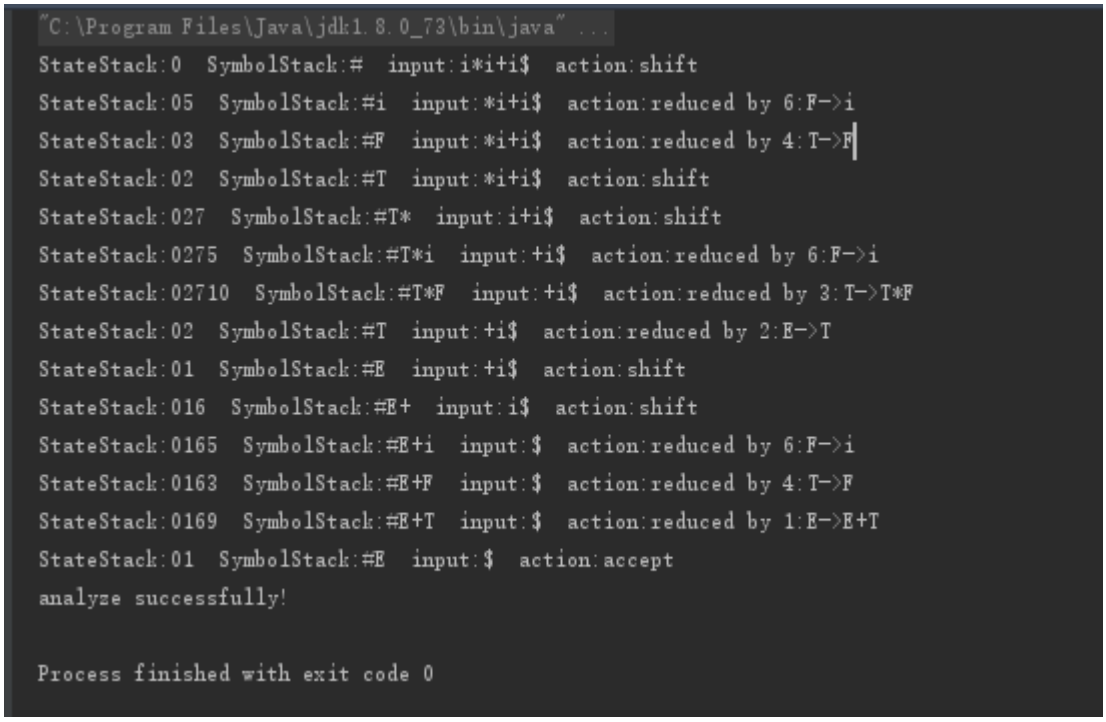
```
S=E
E=E+T
E=T
T=T*F
T=F
F=(E)
F=i
```

词法分析产生的 tokenSequence.txt

```
<ID,a>
<OPERATOR_3,*>
<ID,b>
<OPERATOR_1,+>
<ID,c>

```

语法分析规约过程：



9.问题与解决

从 token 序列到语法分析这一步，刚开始的时候有些不知该如何下手，思考之后，决定将每一个 token 都简化为一个终结符，例如 ID 类型的 token 用 i 来表示，+即用+来表示，这样简化了对输入的处理，并使得词法分析程序和语法分析程序串联起来。

10. 感受与总结

对 LR 的分析过程有了更加清楚的认识，将词法分析和语法分析程序也初步串联起来，虽然分析的内容只是简单的程序段，但是很大程度地增加了对编译的理解。