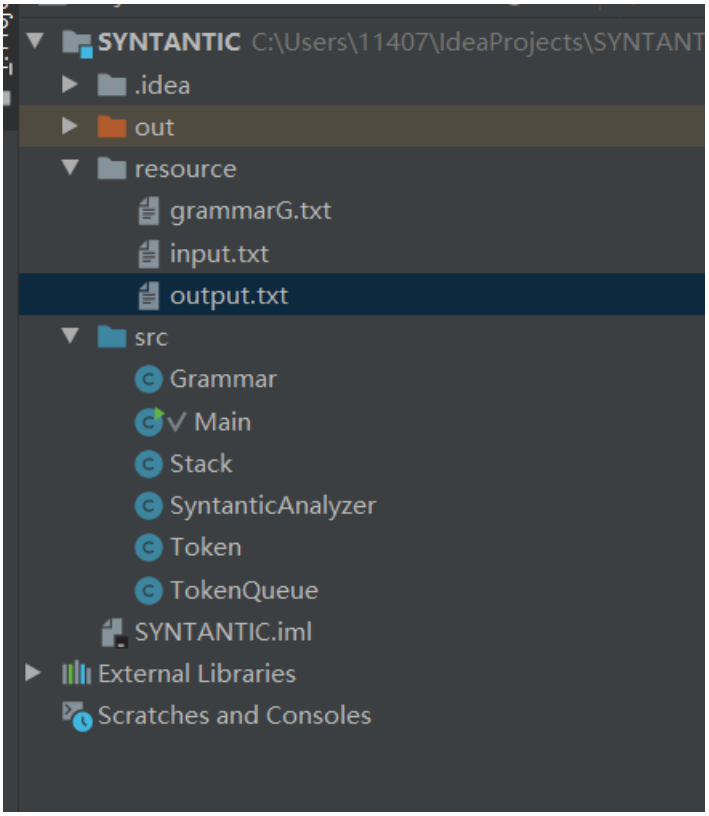
编译原理第二次实验报告

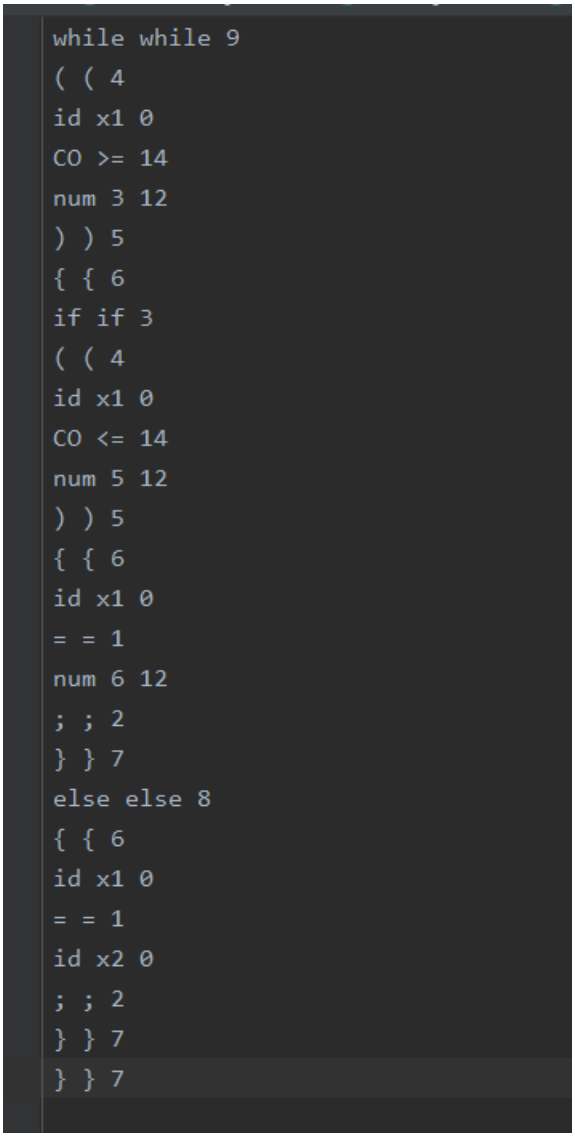
171250535 蔡明卫

零、运行截图

1- 文件结构



2- 输入


while while 9
((4
id x1 0
CO >= 14
num 3 12
)) 5
{ { 6
if if 3
((4
id x1 0
CO <= 14
num 5 12
)) 5
{ { 6
id x1 0
= = 1
num 6 12
; ; 2
} } 7
else else 8
{ { 6
id x1 0
= = 1
id x2 0
; ; 2
} } 7
} } 7

3- 输出

```
S->while(C){S}
C->D C'
D->id CO num
C'->ε
S->if(C){S}else{S}
C->D C'
D->id CO num
C'->ε
S->id = E
E->T E'
T->F T'
F->num
T'->ε
E'->ε
S->id = E
E->T E'
T->F T'
F->id
T'->ε
E'->ε
```

一、 实验目的

本次实验通过设计 LL(1)语法分析程序，对输入的 token 序列分析并生成分析树，并将结果输出至 output.txt，旨在加深对 LL(1)语法分析过程的理解，将编译原理课堂上的内容应用于实践。

二、 内容描述

本次使用我使用 Java 编写程序，读取 input.txt 里面的 token 序列，对其进行自顶向下的 LL(1)语法分析，输出产生式序列并存入 ouput.txt。

三、 处理方法

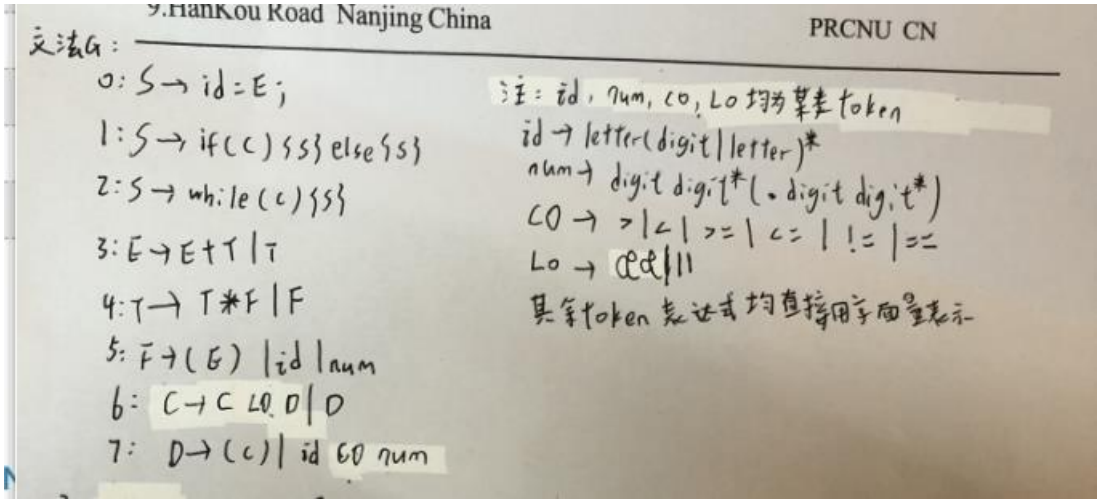
1- 定义文法 G
2- 文法预处理（提取最大左公因子，消除左递归）
3- 利用 first(),follow()构造预测分析表
4- 基于预测分析表编写程序，生成产生式序列
5- 如果能构造分析树，则将产生式序列输出并存入 ouput.txt，如果出错则输出提示

四、 假设

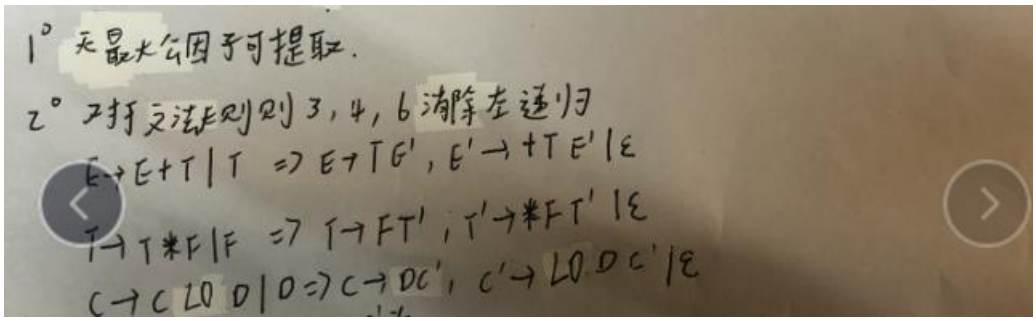
1- 为避免二义性 if 后必存在 else 语句

五、 相关描述

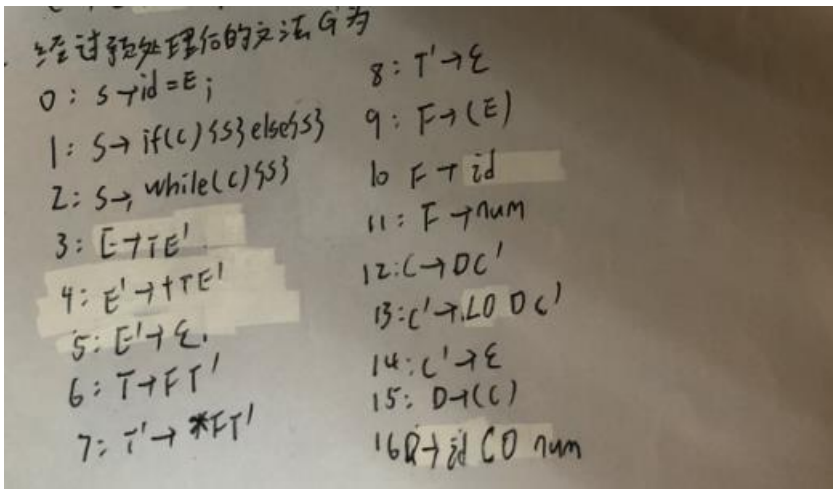
1- 文法 G 的定义



2- 文法预处理



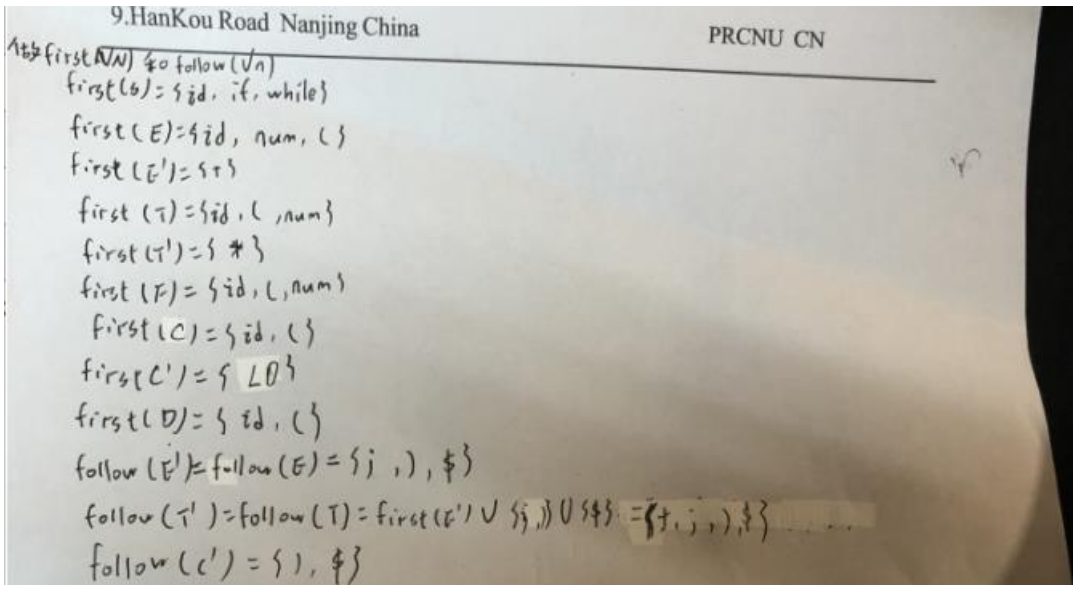
3- 文法 G'



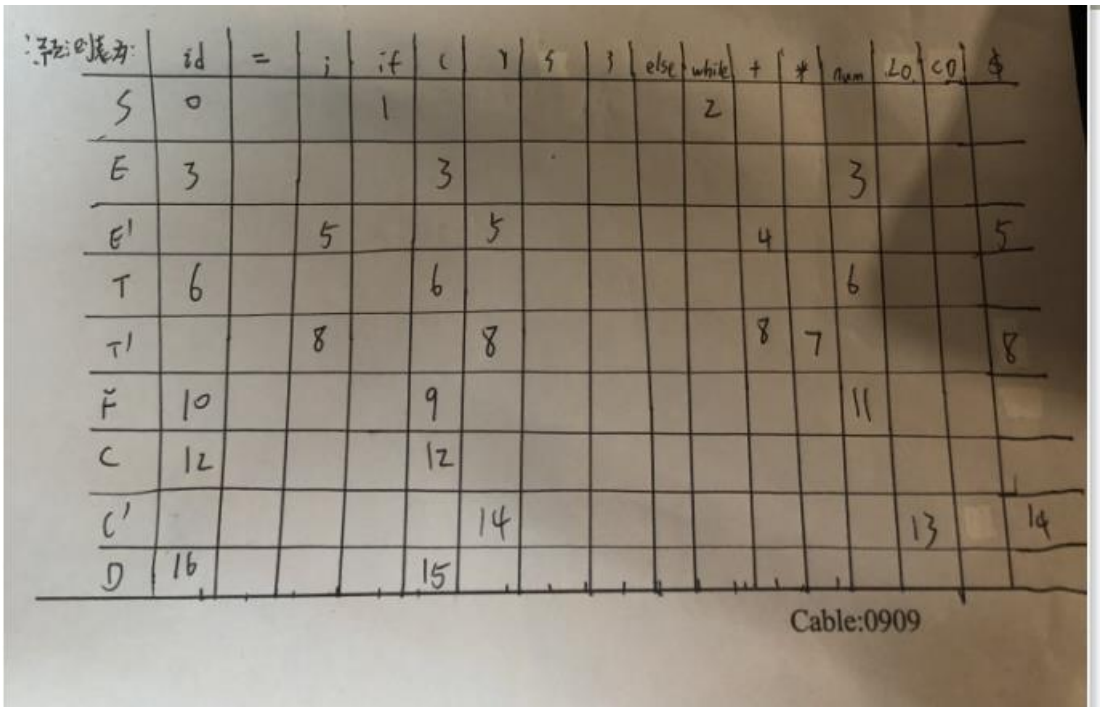
```
0 :S->id = E
1 :S->if(C){S}else{S}
2 :S->while(C){S}
3 :E->T E'
4 :E'->+ T E'
5 :E'->ε
6 :T->F T'
7 :T'->* F T'
8 :T'->ε
9 :F->( E )
10:F->id
11:F->num
12:C->D C'
13:C'->LO D C'
14:C'->ε
15:D->(C)
16:D->id CO num
```

（grammarG.txt）

4- first（Vn）与 follow（Vn）



5- 预测分析表



预测分析表	id	=	;	if	(	)	5	3	else	while	+	*	num	Lo	co	$
S	0			1						2						
E	3				3								3			
E'			5			5					4					5
T	6				6								6			
T'			8			8					8	7				8
F	10				9								11			
C	12				12											
C'						14								13		14
D	16				15											

Cable:0909

```
//LL(1)预测分析表
private int parsingTable[][]={
        {0,-1,-1,1,-1,-1,-1,-1,-1,2,-1,-1,-1,-1,-1,-1},
        {3,-1,-1,-1,3,-1,-1,-1,-1,-1,-1,-1,3,-1,-1,-1},
        {-1,-1,5,-1,-1,5,-1,-1,-1,-1,4,-1,-1,-1,-1,5},
        {6,-1,-1,-1,6,-1,-1,-1,-1,-1,-1,-1,6,-1,-1,-1},
        {-1,-1,8,-1,-1,8,-1,-1,-1,-1,8,7,-1,-1,-1,8},
        {10,-1,-1,-1,9,-1,-1,-1,-1,-1,-1,-1,11,-1,-1,-1},
        {12,-1,-1,-1,12,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1},
        {-1,-1,-1,-1,-1,14,-1,-1,-1,-1,-1,-1,-1,13,-1,14},
        {16,-1,-1,-1,15,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}


};
```

（SyntanticAnalyzer.java）

六、 重要数据结构描述

1- Token

词法分析程序的输出

type 为 1 代表为终结符，-1，代表非终结符

no 代表各终结符，非终结符的种类，即预测分析表中的行/列号

```java
public class Token {
    private int type;
    private int no;
    private String str;

    public Token(int type,int no,String str){
        this.type=type;
        this.no=no;
        this.str=str;
    }

    public Token(){}
    public int getType() { return type; }
    public int getNo() { return no; }
    public String getStr() { return str; }
}
```

2- **Stack**

自定义 Stack，实现压栈，出栈操作

```java
import java.util.ArrayList;

public class Stack {
    private ArrayList<Token> tokenArrayList;
    public Stack(){
        tokenArrayList = new ArrayList<Token>();
        tokenArrayList.add(new Token( type: 1, no: 15, str: "$"));
    }

    public void push(Token t){
        tokenArrayList.add(t);
    }

    public void pop(){
        tokenArrayList.remove( index: tokenArrayList.size() - 1);
    }

    public Token get(){
        return tokenArrayList.get(tokenArrayList.size() - 1);
    }

}
```

3- **Grammar**

对应各条产生式

```java
public class Grammar {
    private int no;
    private ArrayList<Token> tokenArrayList=new ArrayList<>();
    public Grammar(int no){
        this.no=no;
        start(no);
    }
    public void addToStack(Stack stack){
        for(int i=0;i<tokenArrayList.size();i++){
            stack.push(tokenArrayList.get(i));
        }

    }
    public void addGrammarToRes(ArrayList<String> arrayList){...}

    private void start(int no){...}
}
```

4- TokenQueue

将从 input.txt 文件中读取的 token 序列依次存入的队列，实现进队和出队操作

```java
import java.util.ArrayList;

public class TokenQueue {
    private ArrayList<Token> tokenArrayList;

    public TokenQueue() {
        tokenArrayList = new ArrayList<Token>();

    }
    public Token getFirst() { return tokenArrayList.get(0); }

    public void dequeue() { tokenArrayList.remove( index: 0); }

    public void enqueue(Token token) { tokenArrayList.add(token); }

    public int getsize(){
        return tokenArrayList.size();
    }


}
```

七、 核心算法描述

1- readQueue()

从 input.txt 中读取 token 序列，并将其存入 TokenQueue 中

```java
private static TokenQueue readQueue(String path) throws IOException {
    TokenQueue tokenQueue1=new TokenQueue();
    FileInputStream fis = new FileInputStream(path);
    InputStreamReader isr = new InputStreamReader(fis);
    BufferedReader br = new BufferedReader(isr);
    String line = "";
    while ((line = br.readLine()).length()!=0) {
      String[] list1=line.split( regex: " ");
      tokenQueue1.enqueue(new Token( type: 1,Integer.parseInt(list1[2]),list1[0]));
    }
    tokenQueue1.enqueue(new Token( type: 1, no: 15, str: "$"));
    br.close();
    isr.close();
    fis.close();
    return tokenQueue1;

}
```

2- SyntanticAnalyzer.start()

核心算法，分别取出栈顶和队首的元素，如果都是非终结符且两者匹配，栈顶弹出且队伍弹出队首元素，如果栈顶为非终结符，弹出栈顶且查询预测分析表，将相应的产生式从右往左压栈，并将产生式加入结果，如果查询结果为空，则输出错误信息。

```java
public ArrayList<String> start(){
    Token stackTop;
    Token queueTop;
    while(true) {
        stackTop = stack.get();
        queueTop = tokenQueue.getFirst();
        if(stackTop.getNo()==15&&queueTop.getNo()==15){
            return res;
        }
        else if(stackTop.getNo()==15&&queueTop.getNo()!=15){
            res.clear();
            res.add("can not construct a parsing tree");
            return res;
        }
        else if(stackTop.getNo()!=15&&queueTop.getNo()==15){
            res.clear();
            res.add("can not construct a parsing tree");
            return res;
        }
        else{
            if(stackTop.getType()==1){
                //栈顶和队首都是终结符
                if(stackTop.getNo()==queueTop.getNo()){
                    stack.pop();
                    tokenQueue.dequeue();
                    continue;
                }
                else{
                    //错误处理
                    res.clear();
                    res.add("can not construct a parsing tree");
                    return res;
```

```
                }
            }else{
                //栈顶为非终结符
                stack.pop();
                //根据表推导
                if(parsingTable[stackTop.getNo()][queueTop.getNo()]==-1){
                    res.clear();
                    res.add("can not construct a parsing tree");
                    return res;
                }
                grammarlist[parsingTable[stackTop.getNo()][queueTop.getNo()]].addGrammarToRes(res);
                grammarlist[parsingTable[stackTop.getNo()][queueTop.getNo()]].addToStack(stack);
                continue;
            }
        }
    }
```

3- Grammar.addToStack()

本方法用于将查预测分析表获得的产生式从右向左压入栈中(grammar 在创建时会根据序号将产生式包含的 token 序列倒序加入到 Arraylist 中)

```
public void addToStack(Stack stack){
    for(int i=0;i<tokenArrayList.size();i++){
        stack.push(tokenArrayList.get(i));
    }

}
```

八、 测试用例

1- 输入

```
while while 9
( ( 4
id x1 0
CO >= 14
num 3 12
) ) 5
{ { 6
if if 3
( ( 4
id x1 0
CO <= 14
num 5 12
) ) 5
{ { 6
id x1 0
= = 1
num 6 12
; ; 2
} } 7
else else 8
{ { 6
id x1 0
= = 1
id x2 0
; ; 2
} } 7
} } 7
```

2- 输出

```
S->while(C){S}
C->D C'
D->id CO num
C'->ε
S->if(C){S}else{S}
C->D C'
D->id CO num
C'->ε
S->id = E
E->T E'
T->F T'
F->num
T'->ε
E'->ε
S->id = E
E->T E'
T->F T'
F->id
T'->ε
E'->ε
```

九、 问题与解决方案

1- 手工构造预测分析表有点麻烦，容易出错，而且由于 first()和 follow()函数的算法，一个算错其他也会错，算错过一次又花了比重算更长的时间修正错误。

2- 起初试图用实验一一样的 switch 嵌套的形式，发现行数和列数过多，采用了表驱动的方式重构了算法，极大减少了编码量

十、 感受和评论

本次实验加深了我对 LL(1)算法和语法分析这个过程的理解，通过这次实验更是打通了我对编译全过程理解的最后一环，虽然只是学到了编译这门课程的皮毛可也算对编译有了初步的了解，遗憾的是第一次实验我定义的 REs 都没有实际意义，而且 token 的种类太少了，所以只能放弃第一次实验的结果，手动在输入文件中编造 token 序列，遗憾没能将第一次和第二次实验的代码合在一起，假期有机会重构第一次实验的代码，使两者能够结合。