

编译原理

LL(1)语法分析设计原理与实现

学 院：

专 业：

学生姓名：

学 号：

北京交通大学

2023年10月

1. **程序功能描述**

1、将二元式序列文件作为程序的输入；

2、构造文法的first集，follow集，select集，构造LL（1）分析表，并通过分析表队对输入的字符串做出判断，判断其是否是给定文法的算术表达式；

3、可以给出文法相应的LL（1）分析表

4、能发现简单的错误，例如非法字符输入和合法输入但其中某处推导中出现的语法错误等；

1. **主要数据结构**

|  |  |
| --- | --- |
| 变量及类型 | 用途 |
| follow\_table | 存放follow集 |
| input\_str | 存放输入串 |
| first | 存放first集 |
| output | 存放输出内容 |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

1. **程序结构描述**
2. **设计方法**

提前定义好数组保存相关的内容，如图3.2所示。

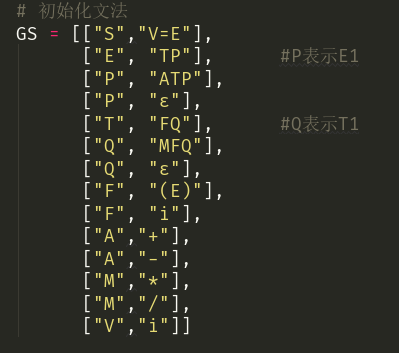


图3.2

实验原理分析如下：

（1）求FIRST集的算法思想

如果产生式右部第一个字符为终结符，则将其计入左部first集

如果产生式右部第一个字符为非终结符

->求该非终结符的first集

->将该非终结符的非$first集计入左部的first集

->若存在$，则将指向产生式的指针右移

->若不存在$，则停止遍历该产生式，进入下一个产生式

->若已经到达产生式的最右部的非终结符，则将$加入左部的first集

处理数组中重复的first集中的终结符

（2）求FOLLOW集的算法思想

对于文法G中每个非终结符A构造FOLLOW(A)的办法是,连续使用下面的规则,直到每个FOLLOW不在增大为止.

(1) 对于文法的开始符号S,置#于FOLLOW(S)中;

(2) 若A->aBb是一个产生式,则把FIRST(b)\{ε}加至FOLLOW(B)中;

(3) 若A->aB是一个产生式,或A->aBb是一个产生式而b=>ε(即ε∈FIRST(b))则把FOLLOW(A)加至FOLLOW(B)中

（3）生成预测分析表的算法思想

构造分析表M的算法是：

(1) 对文法G的每个产生式A->a执行第二步和第三步;

(2) 对每个终结符a∈FIRST(a),把A->a加至M[A,a]中;

(3) 若ε∈FIRST(a),则把任何b∈FOLLOW(A)把A->a加至M[A,b]中;

(4) 把所有无定义的M[A,a]标上出错标志.

（4）对符号串的分析过程

预测分析程序的总控程序在任何时候都是按STACK栈顶符号X和当前的输入符号行事的,对于任何(X,a),总控程序

每次都执行下述三种可能的动作之一;

(1) 若X=a=”#”,则宣布分析成功,停止分析过程.

(2) 若X=a≠”#”,则把X从STACK栈顶逐出,让a指向下一个输入符号.

(3) 若X是一个非终结符,则查看分析表M,若M[A,a]中存放着关于X的一个产生式,那么,首先把X逐出STACK栈顶,然后

把产生式的右部符号串按反序一一推进STACK栈(若右部符号为ε,则意味着不推什么东西进栈).在把产生式的右部

符号推进栈的同时应做这个产生式相应得语义动作,若M[A,a]中存放着”出错标志”,则调用出错诊察程序ERROR.

1. **函数定义及函数间的调用关系**

主要函数及实现功能如下，函数功能见注释：



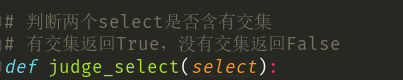






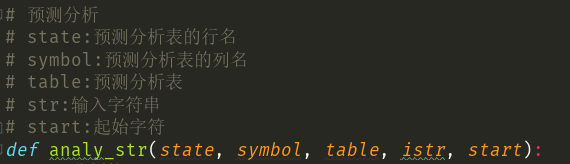


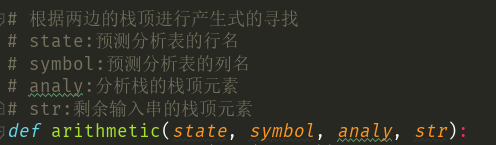


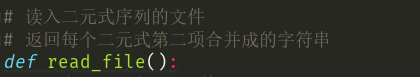




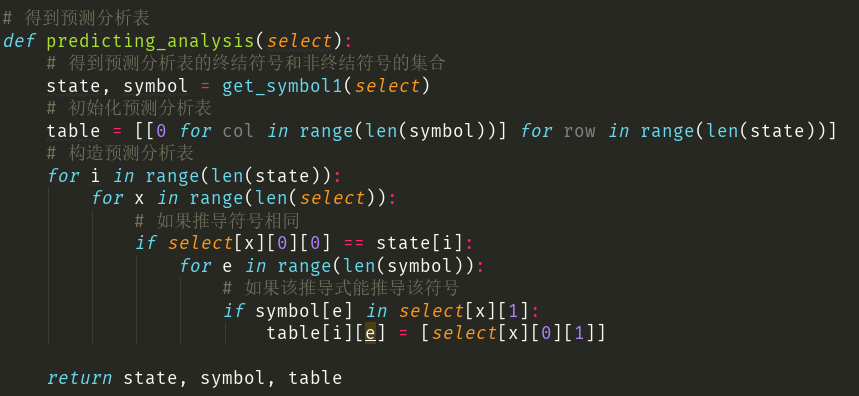








其中，分析表的构建函数如下：



1. **程序测试**

测试用例1：test1.txt

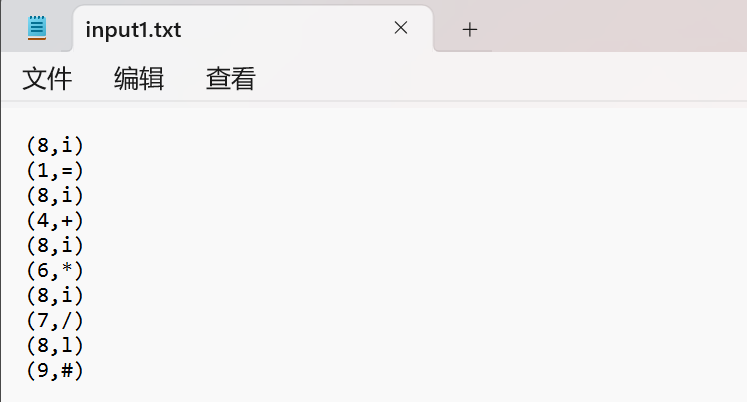
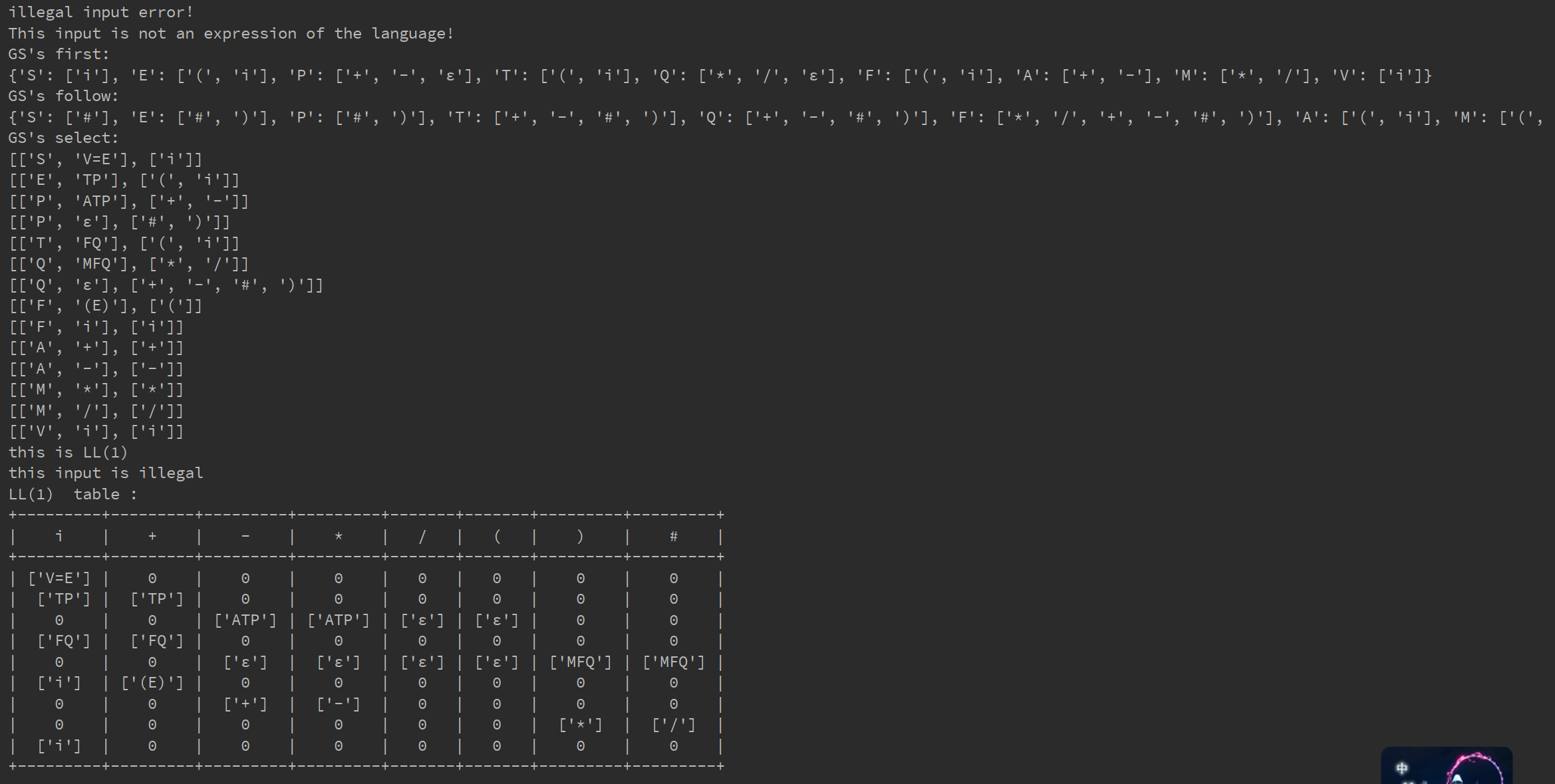


图4.1 测试用例1

程序执行结果：



测试用例2：test2.txt

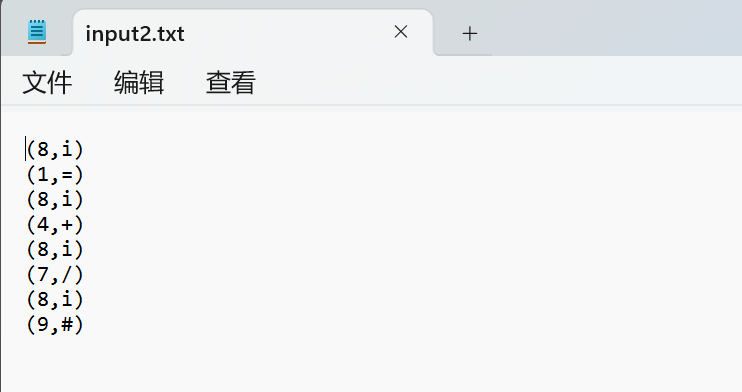


图4.2 测试用例2

程序执行结果：

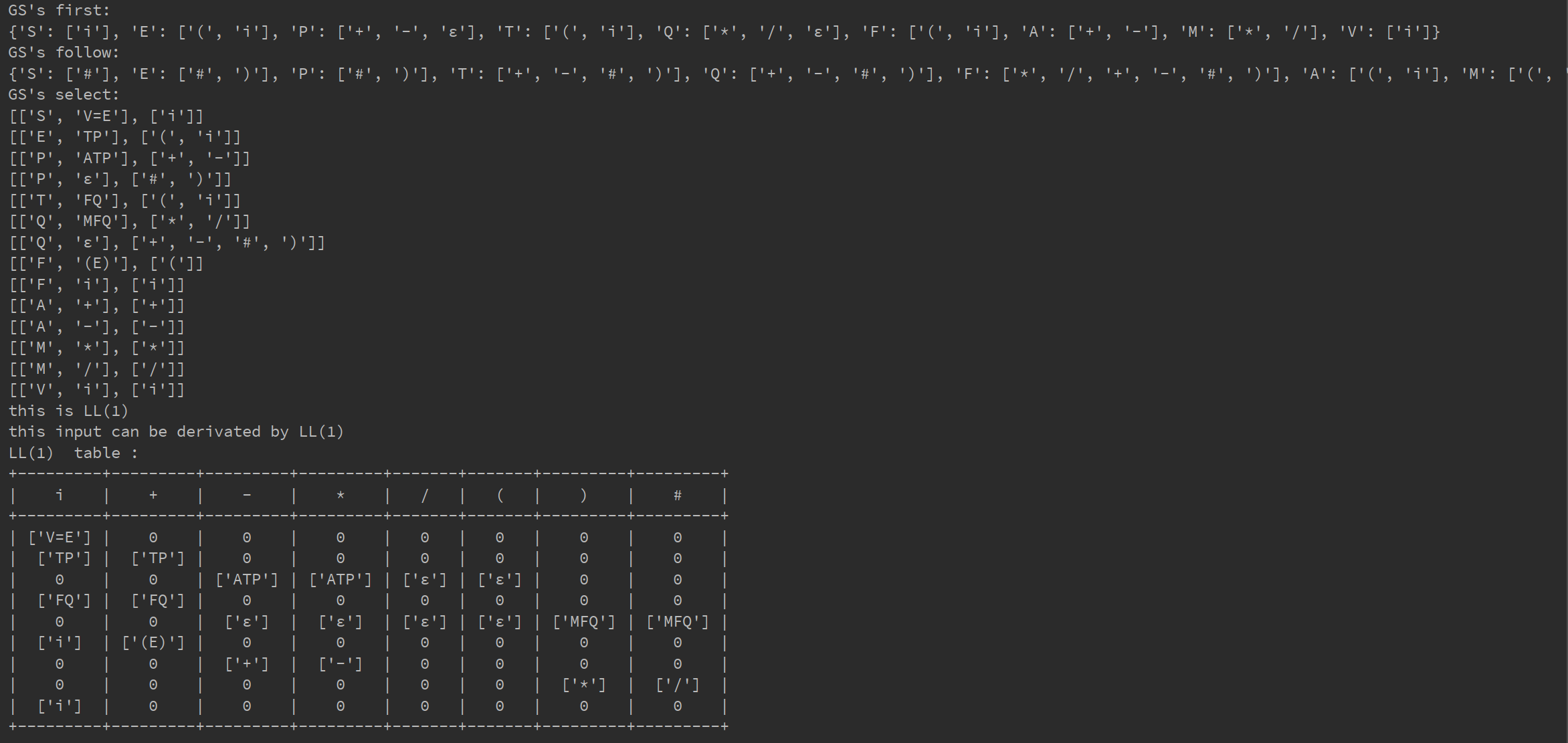


图4.3 测试结果

1. **实验结果分析**

由测试结果可见，我编写的LL(1)语法分析程序可以正确的输出first集、follow集、select集和LL(1)语法的分析表，并且可以正确判断输入的语言是否符合LL(1)语法的规则。而且，我加入了错误判断，并且函数实现是模块化的，便于之后的扩展。

1. **研究性教学实验感受**

刚开始写实验的时候有些无从下手，后面通过看书和查阅资料逐步了解需要完成任务和实现的方法，然后开始逐步实现应有的功能函数。通过这次实验让我对词法LL(1)语法分析的理解更深了一步，这也是做实验和学习这门课程的目的所在。一步步完善实验的过程也是在增进自己的知识。

1. **附件（源代码列表）**

import copy  
import prettytable as pt  
# 终结符号用ε表示，栈顶用#来进行表示  
'''  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*单词符号及其分类编码\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
单词符号 种别码  
= 1  
( 2  
) 3  
+ 4  
- 5  
\* 6  
/ 7  
i 8  
# 9  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
'''  
  
# 用于存储follow表  
follow\_table = {}  
  
  
# 返回FIRST所有的状态  
def get\_state(GS):  
 first = []  
 for i in GS:  
 if i[0] not in first:  
 first.append(i[0])  
 return first  
  
  
# 返回运算后的FIRST集  
def get\_FIRST(GS):  
 # 获取FIRST中的状态集合  
 state = get\_state(GS)  
 # 初始化FIRST表  
 first\_table = [[i] for i in state]  
 for i in range(len(first\_table)):  
 result = recur\_get\_this\_state(first\_table[i][0], GS)  
 first\_table[i].extend(result)  
 return first\_table  
  
  
# 递归循环得到FIRST集  
def recur\_get\_this\_state(target, GS):  
 result = []  
 for i in range(len(GS)):  
 # 当前推导符号和要进行递归的符号相同  
 if target == GS[i][0]:  
 # 判断是否为终结符  
 if GS[i][1][0].islower() or GS[i][1][0].isalpha() == False:  
 add\_into\_result(result, GS[i][1][0])  
  
 else:  
 # 如果为非终结符号  
 temp = recur\_get\_this\_state(GS[i][1][0], GS)  
 # 用于标记  
 index = 0  
 # 若有非终结符存在  
 while ("ε" in temp):  
 for q in range(0, len(temp)):  
 if temp[q] == "ε":  
 del temp[q]  
 break  
 add\_into\_result(result, temp)  
 index += 1  
 if index == len(str(GS[i][1])):  
 add\_into\_result(result, "ε")  
 break  
 else:  
 temp = recur\_get\_this\_state(GS[i][1][index], GS)  
 add\_into\_result(result, temp)  
  
 return result  
  
  
# 向list中添加不重复的元素  
def add\_into\_result(old, add):  
 for i in add:  
 if i not in old:  
 old.extend(i)  
  
  
# 得到follow集  
def get\_FOLLOW(GS, first):  
 global follow\_table  
 # 获取FIRST中的状态集合  
 state = get\_state(GS)  
 # 初始化FIRST表  
 for i in range(len(state)):  
 follow\_table[str(state[i])] = []  
 follow\_table[str(state[0])].extend("#")  
 for i in range(len(follow\_table)):  
 result = recur\_get\_follow(str(state[i]), GS, first)  
 add\_into\_result(follow\_table[str(state[i])], result)  
 return follow\_table  
  
  
# 递归得到follow集  
def recur\_get\_follow(target, GS, first):  
 result = []  
 for i in range(len(GS)):  
 for a in range(1, len(GS[i])):  
 # 查看是否包含当前字符  
 if str(GS[i][a]).find(target) != -1:  
 # 获得当前字符所在的位置，指向当前字符  
 index = str(GS[i][a]).find(target)  
 # 当前字符后面跟有其他字符  
 if index != len(GS[i][a]) - 1:  
 # 如果为终结符号的话  
 index += 1  
 if GS[i][a][index].islower() or GS[i][a][index].isalpha() == False:  
 add\_into\_result(result, GS[i][a][index])  
 else:  
 # 如果为非终结符号，则进行判断是FIRST还是FOLLOW集加入  
 while (index != len(GS[i][a])):  
 # 如果在FIRST集中有空符号出现，则该FIRST集加入到FOLLOW集当中，并且读取下一个符号  
 if "ε" in first[str(GS[i][a][index])]:  
 new = first[str(GS[i][a][index])]  
 for x in new:  
 if x != "ε":  
 add\_into\_result(result, x)  
 else:  
 # 当前FIRST集中没有空符号，则该FOLLOW集添加完毕，退出循环  
 add\_into\_result(result, first[str(GS[i][a][index])])  
 break  
 index += 1  
 # 如果index等于当前符号跳转位的长度，说明该符号跳转位到最后一位为止都在进行判别，则此时把  
 # 跳转符号的FOLLOW集加入到目标FOLLOW集当中  
 if index == len(GS[i][a]):  
 if follow\_table[str(GS[i][0])] == []:  
 rea = recur\_get\_follow(str(GS[i][0]), GS, first)  
 add\_into\_result(result, rea)  
 else:  
 add\_into\_result(result, follow\_table[str(GS[i][0])])  
 else:  
 # 当前字符是最后一个字符  
 # 根据该推导字符是否已经被推导过了来决定是否进行循环  
 if follow\_table[str(GS[i][0])] == []:  
 rea = recur\_get\_follow(str(GS[i][0]), GS, first)  
 add\_into\_result(result, rea)  
 else:  
 add\_into\_result(result, follow\_table[str(GS[i][0])])  
 add\_into\_result(follow\_table[target], result)  
 return result  
  
  
# 计算得到select集  
def get\_SELECT(GS, first, follow):  
 # 初始化select表  
 select = [[[a, b], []] for a, b in GS]  
 # 用来进行标记FIRST合并的符号  
 flag = False  
 # 存储非终结符号里面的结果  
 new\_add = []  
 # 依次查看跳转的每一条语句  
 for i in range(len(select)):  
 # 查看跳转的每个字符  
 for a in select[i][0][1]:  
 # 如果为非终结符号，则加入到select集中，并不再进行接下来的判断  
 if str(a).islower() or str(a).isalpha() == False:  
 add\_into\_result(new\_add, a)  
 break  
 else:  
 # 如果为非终结符号  
 # 查看非终结符号所含有的FIRST集中的集合是否含有空符号  
 # 若有则读取跳转符号的follow集  
  
 # 挨个去除FIRST集，并按规定去除里面含有的空集  
 if new\_add == []:  
 new\_add = copy.deepcopy(first[str(a)])  
 if "ε" not in new\_add:  
 break  
 else:  
 # 假如是第二次进行FIRST集合并则需要去除其中一个的ε符号  
 if flag == False:  
 if "ε" in new\_add:  
 for x in range(len(new\_add)):  
 if new\_add[x] == "ε":  
 del new\_add[x]  
 break  
 add\_into\_result(new\_add, first[str(a)])  
 elif "ε" in first[str(a)]:  
 for i in first[str(a)]:  
 if i != "ε":  
 add\_into\_result(new\_add, i)  
 flag = True  
 elif "ε" in first[str(a)]:  
 for i in first[str(a)]:  
 if i != "ε":  
 add\_into\_result(new\_add, i)  
 # 如果FIRST集中有空符号，则去除空符号，并将FOLLOW集加入  
 if "ε" in new\_add:  
 for f in range(len(new\_add)):  
 if new\_add[f] == "ε":  
 del new\_add[f]  
 break  
 add\_into\_result(new\_add, follow[str(select[i][0][0])])  
  
 # 将结果加入到select集中  
 select[i][1].extend(new\_add)  
 # 重置变量初始值  
 new\_add = []  
 flag = False  
  
 return select  
  
  
# 判断两个select是否含有交集  
# 有交集返回True，没有交集返回False  
def judge\_select(select):  
 for a1 in select:  
 for a2 in select:  
 # 判断两个select集是有否相同左部产生式的  
 if a1[0][0] == a2[0][0] and a1[0][1] != a2[0][1]:  
 intersection = list(set(a1[1]).intersection(set(a2[1])))  
 if intersection != []:  
 return True  
 return False  
  
  
# 返回select集合中的终结符号集合  
def get\_symbol1(select):  
 result1 = []  
 result2 = []  
 for i in select:  
 for x in i[0][0]:  
 if x not in result1:  
 result1.extend(x)  
 for x in i[1]:  
 if x not in result2:  
 result2.extend(x)  
 return result1, result2  
  
  
# 得到预测分析表  
def predicting\_analysis(select):  
 # 得到预测分析表的终结符号和非终结符号的集合  
 state, symbol = get\_symbol1(select)  
 # 初始化预测分析表  
 table = [[0 for col in range(len(symbol))] for row in range(len(state))]  
 # 构造预测分析表  
 for i in range(len(state)):  
 for x in range(len(select)):  
 # 如果推导符号相同  
 if select[x][0][0] == state[i]:  
 for e in range(len(symbol)):  
 # 如果该推导式能推导该符号  
 if symbol[e] in select[x][1]:  
 table[i][e] = [select[x][0][1]]  
  
 return state, symbol, table  
  
  
# 预测分析  
# state:预测分析表的行名  
# symbol:预测分析表的列名  
# table:预测分析表  
# str:输入字符串  
# start:起始字符  
def analy\_str(state, symbol, table, istr, start):  
 # 分析栈  
 analy\_shed = []  
 # 输入串栈  
 str\_shed = []  
  
 # 初始化栈  
 analy\_shed.append("#")  
 analy\_shed.append(start)  
 str\_shed.append("#")  
 str\_shed.extend(list(istr[::-1]))  
 while(str\_shed[-1] != "#" or analy\_shed[-1] != "#"):  
 # 查看栈顶是否相同，相同则匹配消去  
 if str\_shed[-1] == analy\_shed[-1]:  
 str\_shed.pop()  
 analy\_shed.pop()  
 else:  
 # 当栈顶不同的时候，看能否进行产生式推导  
 y,x = arithmetic(state,symbol,analy\_shed[-1],str\_shed[-1])  
 if table[y][x] == 0:  
 return "this input is illegal"  
 elif table[y][x][0] == "ε":  
 analy\_shed.pop()  
 else:  
 analy\_shed.pop()  
 # 推导的表达式入栈  
 analy\_shed.extend(list((table[y][x][0])[::-1]))  
 return "this input can be derivated by LL(1)"  
  
  
# 根据两边的栈顶进行产生式的寻找  
# state:预测分析表的行名  
# symbol:预测分析表的列名  
# analy:分析栈的栈顶元素  
# str:剩余输入串的栈顶元素  
def arithmetic(state, symbol, analy, str):  
 for a in range(len(state)):  
 for b in range(len(symbol)):  
 # 如果根据栈顶可以推导出新的产生式，则进行推导  
 if state[a] == analy and symbol[b] == str:  
 # 返回产生式在预测分析表里面的位置  
 return a, b  
 return -1,-1  
  
def error():  
 print("Syntax error")  
 print("This input is not an expression of the language!")  
  
def error1():  
 print("illegal input error！")  
 print("This input is not an expression of the language!")  
  
  
def judge(symbol):  
 chart = {  
 '1': "=",  
 '2': "(",  
 '3': ")",  
 '4': "+",  
 '5': "-",  
 '6': "\*",  
 '7': "/",  
 '8': "i",  
 '9': "#",  
 }  
 return chart.get(symbol)  
  
# 返回每个二元式第二项合并成的字符串  
def read\_file():  
 #读取二元式txt文件  
 with open('input2.txt', 'r') as f1:  
 list1 = f1.readlines()  
  
 list1\_symbol = []  
 list1\_new = []  
 for i in range(0, len(list1)):  
 list1[i] = list1[i].rstrip('\n') #去除每行结尾的回车符  
 list1\_symbol.append(list1[i].\_\_getitem\_\_(1)) # 提取每个二元式的第二项  
 list1\_new.append(list1[i].\_\_getitem\_\_(3)) #提取每个二元式的第二项  
  
 #判断文件读取内容是否合法，包括检查非法字符和不匹配现象  
 for symbol,value in zip(list1\_symbol,list1\_new):  
 if judge(symbol) != value:  
 error1() #出现非法字符则报错  
  
 # 返回每个二元式第二项合并成的字符串  
 return "".join(list1\_new)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 # 初始化文法  
 GS = [["S","V=E"],  
 ["E", "TP"], #P表示E1  
 ["P", "ATP"],  
 ["P", "ε"],  
 ["T", "FQ"], #Q表示T1  
 ["Q", "MFQ"],  
 ["Q", "ε"],  
 ["F", "(E)"],  
 ["F", "i"],  
 ["A","+"],  
 ["A","-"],  
 ["M","\*"],  
 ["M","/"],  
 ["V","i"]]  
  
 # 输入串，即词法分析的输出二元式序列  
 input\_str = read\_file()  
  
 # 得到FIRST集  
 first = {}  
 output = get\_FIRST(GS)  
 for i in range(len(output)):  
 first[str(output[i][0][0])] = output[i][1:]  
 # 得到FOLLOW集  
 follow = get\_FOLLOW(GS, first)  
 print("GS's first:")  
 print(first)  
 print("GS's follow:")  
 print(follow)  
 # 得到select集  
 select = get\_SELECT(GS, first, follow)  
 print("GS's select:")  
 for i in select:  
 print(i)  
  
 # 根据select的交集进行判断是否为LL(1)文法  
 if judge\_select(select):  
 print("this is not LL(1)") #因为相同左部产生式的select集的交集不为空  
 else:  
 print("this is LL(1)")  
  
 # 得到预测分析表  
 state, symbol, table = predicting\_analysis(select)  
  
 # 预测分析  
 result = analy\_str(state, symbol, table, input\_str, GS[0][0])  
 print(result)  
  
 #输出LL(1)分析表  
 tb = pt.PrettyTable()  
 tb.field\_names = ["i", "+", "-", "\*", "/", "(", ")", "#"]  
  
 table\_new = []  
 for i in table:  
 tb.add\_row(i)  
 print("LL(1) table :")  
 print(tb)