中国地质大学

题　　目:	编译原理实验报告
姓　　名:	常文瀚
院　　系:	计算机学院
班　　级:	191181
学　　号:	20181001095

2020 年 12 月 1 日

目录

一、实验题目

应用 LR（0）方法实现语法分析

二、问题描述

用 LR（0）分析法编制语法分析程序，实现可以采用任何一种编程工具。

可用如下文法进行测试：

文法 G（E）：

E->TA

A->+TA

A->e

T->FB

B->*FB

B->e

F->（E）

F->i

三、基本要求

- 构造识别活前缀的 DFA
- LR(0)分析表构造
- 构造 LR(0)语法分析器

四、小组分工

- 王泽清 高天翔：负责总体规划，系统整合以及构造识别活前缀的 DFA。
- 苏浩文 常文瀚：负责在活前缀 DFA 的基础上 LR（0）分析表的构造。
- 刘乐菲 刘宇洁：负责在 LR（0）分析表的基础上构造 LR(0)语法分析器。

五、整体设计

1. 数据结构

本文使用 python 编程语言，使用列表的数据结构进行文法，DFA 等结构的表示，使用字典的数据结构表示终结符和非终结符的映射，使用字符形式表示栈顶的状态和栈顶字符

等结构。具体如下：

- VN = [] # 非终结符
- VT = [] # 终结符
- NFA = [] # NFA 表
- DFA = [] # DFA 表
- grammar = [] # 读入的文法
- doted_grammar = [] # 加点后的文法
- VN2Int = {} # 非终结符映射
- VT2Int = {} # 终结符映射
- action = [] # action 表
- goto = [] # goto 表
- DFA_node = [] # DFA 节点表
- status_stack = [] # 状态栈
- symbol_stack = [] # 符号栈
- now_state = '' # 栈顶状态
- input_ch = '' # 栈顶字符
- input_str = '' # 输入串
- location = 0 # 输入位置

- now_step = 0 # 当前步骤

2. 文件结构

本实验共有三个文件：

- FiniteAutomata.py 和 LR.py：主要用于识别活前缀的 DFA 的可视化功能的实现
- output.py：主要用于进行主程序的实现，包括构造识别活前缀的 DFA，LR（0）分析表的构造以及 LR(0)语法分析器的构建

3. 基本思想

LR 语法分析技术

- LR(k)的语法分析概念

○ L 表示最左扫描，R 表示反向构造出最右推导

- k 表示最多向前看 k 个符号
- LR 语法分析器的优点：
- 由表格驱动
- 对于几乎所有的程序设计语言，只要写出 CFG，就能够构造出识别该语言的 LR 语法分析器
- 最通用的无回溯移入-规约分析技术
- 能分析的文法比 LL(k)文法更多
- 项：文法的一个产生式加上在其中某处的一个点
- A→α·β 表示已经扫描/规约到了 α，并期望在接下来的输入中经过扫描/规约得到 β,然后把 α β 规约到 A
- 如果 β 为空，表示我们可以把 α 规约到 A

LR(0)项集规范族的构造

- 三个相关定义：
- 增广文法
- 项集闭包 CLOSURE
- GOTO 函数
- 增广文法：
- GG 的增广文法 G'是在 G 中增加新开始符号 S'，并加入产生式 S'→S 而得到的
- 显然 G'和 G 接受相同的语言，且按照 S'→S 进行规约实际上就表示已经将输入的符号串规约成开始符号

- 项集闭包 CLOSURE：
- 如果 I 是文法 G 的一个项集，CLOSURE(I)就是根据下列两条规则从 I 构造得到的项集：
  - 将 I 的各项加入到 CLOSURE(I) 中
  - 如果 A→α·Bβ 在 CLOSURE(I)中，而 B→γ 是一个产生式，且项 B→·γ 不在 CLOSURE(I)中，就讲该项加入其中
  - 不断应用以上规则直到没有新项可加入
- 项集闭包的含义：
- 希望看到由 Bβ 推导出的串，要先看到由 B 推导出的串，因此加上 B 的各个产生式对应的串
- GOTO 函数：
  - I 是一个项集，X 是一个文法符号，则 GOTO(I,X)定义为 I 中所有形如 [A→α·

Xβ］的项所对应的［A→αX·β］的项的集合的闭包

- 求LR(0)项集规范族的方法：
  - 从初始项集)CLOSURE({S’→·S})开始，不断计算各种可能的后继，直到生成所有的项集。

LR(0)自动机的构造

- 规范LR(0)项集族中的每个项集对应于LR(0)自动机的一个状态
- 如果GOTO(I,X)=J，则从I到J有一个标号为X的转换
- 开始状态为 CLOSURE({S’→·S}) 对应的项集

LR(0)自动机的作用

假设文法符号串 γ 使LR(0)自动机从开始状态运行到状态j

- 如果jj中存在项A→α·，那么在 γ 之后添加一些终结符号可以得到一个最右句型
  - α 是 γ 的后缀，且是该句型的句柄（对应于A→α）
  - 表示可能找到了当前最右句型的句柄，可以规约
- 如果j中存在项 B→α·Xβ，那么在 γ 之后添加Xβ 和一些终结符号可以得到一个最右句型
  - 该句型中 αXβ 是句柄，但还没有找到，需要继续移入

LR语法分析器的结构

- 栈中存放状态序列，可求出对应的符号序列
- 分析程序根据栈顶状态和当前输入，通过分析表确定下一步动作

LR语法分析表的结构：

- 动作ACTION[i][a]：i是状态，a是下一个符号移入 j：将新状态j压入栈，同时移入a
  - 规约A→β：把栈顶的 β 规约为A（并根据GOTO表项压入新状态）
  - 接受：接受输入，完成分析
  - 报错：在输入中发现语法错误
- 转换GOTO[i][A]：
  - 如果GOTO(Ii,A)=Ij，那么表项GOTO[i][A] = j

LR语法分析器的格局包含了栈中内容和余下输入，

在任何时候栈对应的符号串和剩下的输入组成一个最右句型。

LR语法分析器的行为：对于当前格局，查询当前栈顶状态和下一个符号对应的动作：移入、规约、接受或拒绝。

4. 构造识别活前缀的 DFA

- 识别活前缀的 DFA 首先需要读取文法，构造所有项目，之后需要依据项目集构造 NFA ，最后将 NFA 确定化，构造出识别活前缀的 DFA。
- DFA 的可视化

本程序通过 Python 下调用 Graphviz 来实现 DFA 的绘制，关于 Graphviz 的介绍如下：Graphviz (Graph Visualization Software) 使用 DOT 语言来描述图形的节点、连线及属性等，可以大大减少手绘图形调整图形格式的时间，而将主要精力放在图形的逻辑关系上。而且可以根据需要设置生成图像的格式，如 PDF、JPG 等。

- 具体步骤：

1 ）读取文法，函数：def read_grammar(file_name)。

2 ）找到终结符和非终结符，函数：def find_term_non()。

3 ）给文法加点，函数：def add_dot()。

4 ）构造 NFA，函数：def make_nfa()。

5） 构造 DFA，函数：def make_dfa()。

5. LR（0）分析表的构造

（1） LR(0)分析表构造算法

假设已构造出 LR(0)项目集规范族为：C={I0,I1, … , In}，其中 Ik 为项目集的名字，k 为状态名，令包含 S′ →·S 项目的集合 Ik 的下标 k 为分析器的初始状态。那么分析表的 ACTION 表和 GOTO 表构造步骤为：

① 若项目 A→α·aβ 属于 Ik 且转换函数 GO(Ik,a)= Ij，当 a 为终结符时则置 ACTION[k,a]为 Sj。

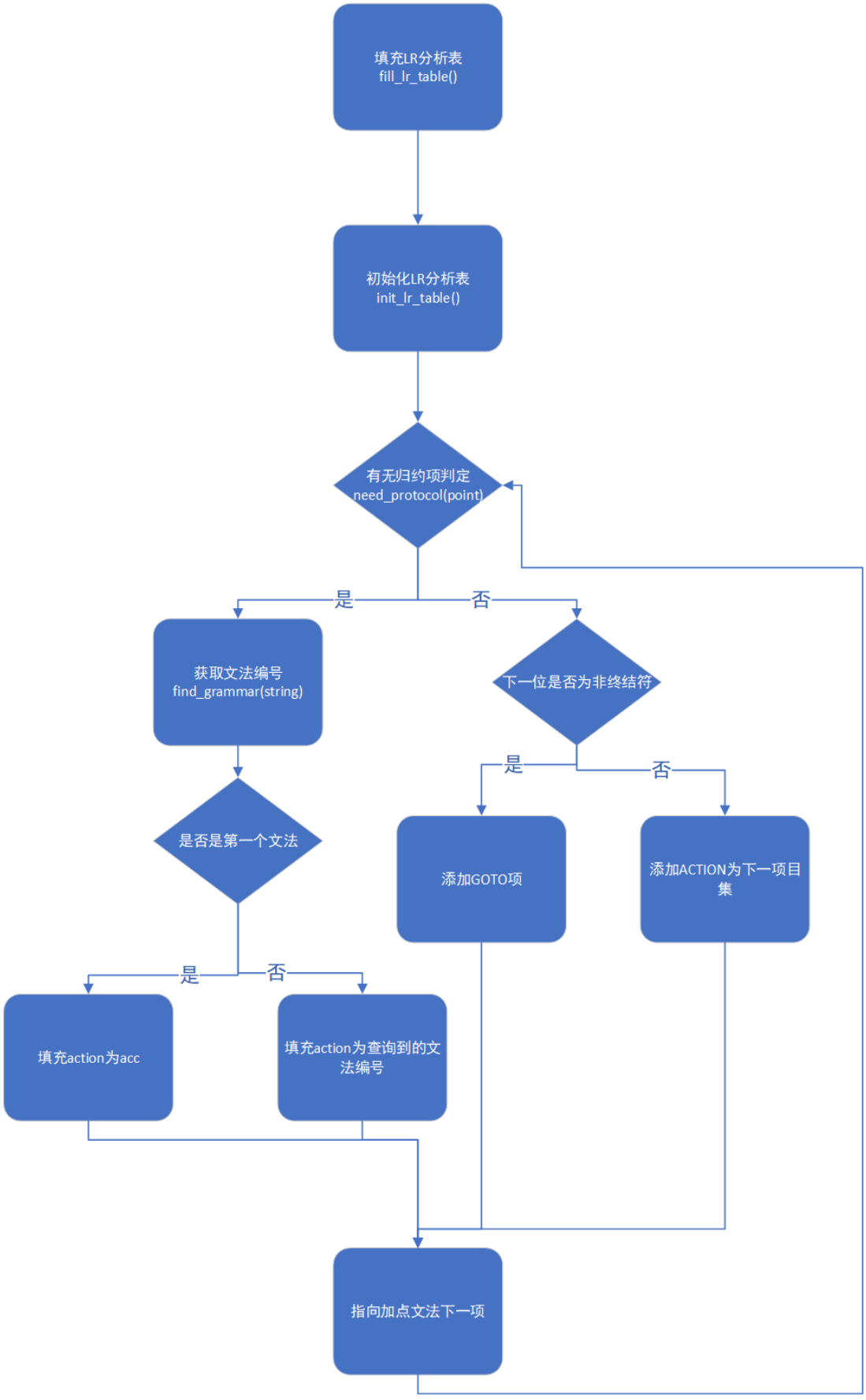
② 若项目 A→α· 属于 Ik，则对任何终结符 a 和’#’号置 ACTION[k,a]和 ACTION[k,#为”rj”，j 为在文法 G′ 中某产生式 A→α 的序号。

③ 若 GO(Ik,A)＝Ij，则置 GOTO[k,A]为”j”，其中 A 为非终结符。

④ 若项目 S′ →S·属于 Ik，则置 ACTION[k,#]为”acc”，表示接受。

⑤ 凡不能用上述方法填入的分析表的元素，均应填上”报错标志”。为了表的清晰我们仅用空白表示错误标志。

(2) LR(0)分析表构造流程



6. 分析表的基础上构造 LR(0)语法分析器

一、一个 LR 分析器由 3 个部分组成

（1）总控程序，也可以称为驱动程序。对所有的 LR 分析器，总控程序都是相同的。

（2）分析表或分析函数。不同的文法分析表将不同，同一个文法采用的 LR 分析器不同时，分析表也不同，分析表又可以分为动作（ACTION）表和状态转换（GOTO）表两个部分，它们都可用二维数组表示。

（3）分析栈，包括文法符号和相应的状态栈。它们均是先进后出栈。

分析器的动作由栈顶状态和当前输入符号所决定（LR(0)分析器不需向前查看输入符号）。

解决问题的基本思路

1、 用构造一个状态转换函数实现状态转换。

2、 再通过函数构造一个移进－归约函数实现移进规约动作。

3、 采用构造一个打印 LR 分析器的工作过程函数实现输出。

二、在规范规约的过程中，一方面记住已移进和规约出的整个符号串，另一方面根据所用的产生式推测可能碰到的输入符号。每一项 ACTION(s，a)所规定的动作不外是下述四种可能之一

（1）移进：把（s，a）的下一个转态 s’ = GOTO(s，X)和输入符号 a 推进栈，下一输入符号变成现行输入符号。

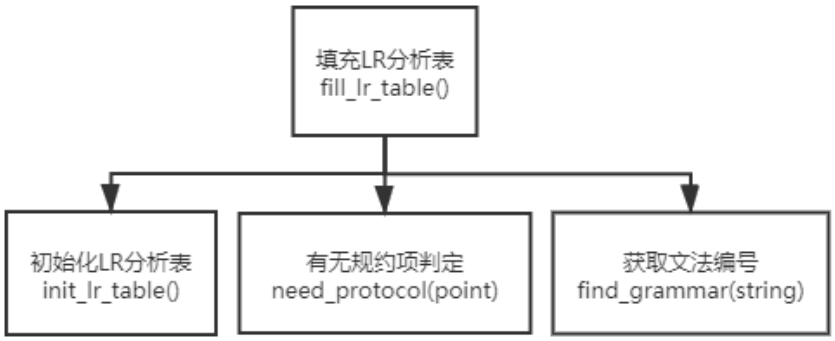
（2）规约：指用某一产生式 A→β 进行规约。假若 β 的长度为 r，规约的动作是 A，去除栈顶的 r 个项，使状态 Sm-r 变成栈顶状态，然后把（Sm-r，A）的下一状态 s’ = GOTO(Sm-r,A)和文法符号 A 推进栈。规约动作不改变现行输入符号。执行规约动作意味着 β（= Xm-r+1…Xm）已呈现于栈顶而且是一个相对于 A 的句柄。

（3）接受：宣布分析成功，停止分析器的工作。

（4）报错：发现源程序含有错误，报错，停止分析器的工作。

六、自己负责的模块设计

1、过程或函数调用关系图



2、模块接口说明

```
class LR:
def __init__(self):
    # e.g. 1. S' -> ·S: projects[S'][·S] = 1 记录所有项目及编号
    self.projects = defaultdict(defaultdict)
    # e.g. 1. S' -> ·S: sorted_projects[1] = (S', ·S) 根据编号查找项目
    self.sorted_projects = dict()
    self.projectSet = dict()    # 项目集
    self.production = []    # 产生式

    # e.g. 3. A -> ·aA, 1. A -> aA: production_numdict[A][·aA] = 1  项目对应的产生式编号
    self.production_numdict = defaultdict(defaultdict)

    self.dfa = FA()
    self.projects_num = 0
    self.startsymbol = None
    self.nonterminals = []
```

3、函数的功能实现

```
# 显示读入文法
def print_read_grammar()
# 初始化 NFA
def init_nfa()
```

```
# 找到点的位置
def find_pos_point(one_grammar)
# 文法是否以 start 开头，以'.'开始
def is_start(grammar_i, start)
# 查找以 start 开头，以'.'开始的文法,返回个数
def find_node(start, grammar_id)
# 构造 NFA
def make_nfa()
# 查找关联
def add_more(i, j)
# 初始化 DFA
def init_dfa()
# 连接
def add_state(to, fro)
# 构造 DFA
def make_dfa()
# 初始化 LR 分析表
def init_lr_table()
# 有无规约项
def need_protocol(point)
# 根据文法内容找到文法编号
def find_grammar(string)
# 填充 LR 分析表
def fill_lr_table()
# 显示 LR 分析表
def print_lr_table()
# 判断分析是否完成
def is_end()
# 输出
def output()
# 统计产生式右部的个数
def count_right_num(grammar_i)
```

七、算法设计

构造识别活前缀的 DFA：

```
    # 读取文法
    def read_grammar(file_name):
        global grammar
        with open(file_name, 'r') as file:
            for line in file:
                line = line.replace('\n', "")
                grammar.append(line)
            file.close()
    # 找到终结符和非终结符
    def find_term_non():
        global grammar
        n = int(len(grammar))
        temp_vt = []
        l = 0
        for i in range(n):
            X, Y = grammar[i].split('->')
            if X not in VN:
                VN.append(X)
                VN2Int.update({X: l})
                l += 1
            for Yi in Y:
                temp_vt.append(Yi)
        m = 0
        for i in temp_vt:
            if i not in VN and i not in VT:
                VT.append(i)
                VT2Int.update({i: m})
```

```
            m += 1
    VT.append('#')
    VT2Int.update({'#': m})
# 给文法加点
def add_dot():
    global doted_grammar
    j = 0
    n = 0
    for i in grammar:
        for k in range(len(i) - 2):
            doted_grammar.append([])
            doted_grammar[n].append(add_char2str(i, k + 3))
            doted_grammar[n].append('false')
            n += 1
        j += 1
# 初始化NFA
def init_nfa():
    global NFA
    for row in range(len(doted_grammar)):
        NFA.append([])
        for col in range(len(doted_grammar)):
            NFA[row].append('')
# 构造NFA
def make_nfa():
    global NFA
    grammar_id = []
for i in range(10):
    grammar_id.append('')
init_nfa()
i = 0
```

```
for grammar_i in doted_grammar:
    pos_point = find_pos_point(grammar_i[0])  # 找到点的位置
    if not pos_point + 1 == len(grammar_i[0]):
        NFA[i][i + 1] = grammar_i[0][pos_point + 1]
        if grammar_i[0][pos_point + 1] in VN:  # 点后面跟着非终结符
            j = find_node(grammar_i[0][pos_point + 1], grammar_id)
            for k in range(j):
                NFA[i][grammar_id[k]] = '*'
                add_more(i, grammar_id[k])
    i += 1
# 初始化 DFA
def init_dfa():
    global DFA
    for row in range(len(doted_grammar)):
        DFA.append([])
        for col in range(len(doted_grammar)):
            DFA[row].append('')
# 构造 DFA
def make_dfa():
    global NFA, doted_grammar, DFA_node
    init_dfa()
    for i in range(len(doted_grammar)):
        DFA_node.append([])
        for j in range(len(doted_grammar)):
            DFA_node[i].append("")
    for i in range(len(doted_grammar)):
        if doted_grammar[i][1] == 'false':
            k = 0
            DFA_node[i][k] = doted_grammar[i][0]
            k += 1
```

```
                doted_grammar[i][1] = 'true'
                for j in range(len(doted_grammar)):
                    if NFA[i][j] == '*':  # 有 ε 弧
                        DFA_node[i][k] = doted_grammar[j][0]
                        k += 1
                        doted_grammar[j][1] = 'true'
                        add_state(i, j)
```

八、调试分析

（1） 运行结果

i）读入的文法

```
E:\python\python.exe E:/projects/bianyi/Syntax-Analysis-master/output.py
----读入的文法----
(0)S->E
(1)E->TA
(2)A->+TA
(3)A->e
(4)T->FB
(5)B->*FB
(6)B->e
(7)F->(E)
(8)F->i
```

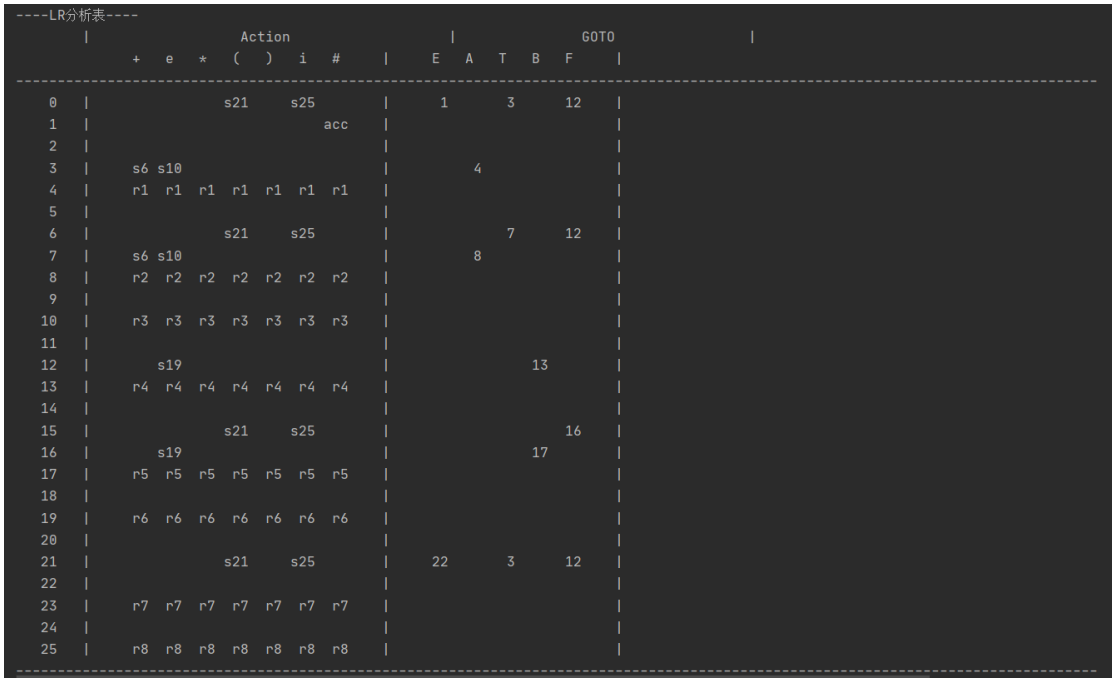
ii）加点后的文法

```
----加点后的文法----
1.S->.E
2.S->E.
3.E->.TA
4.E->T.A
5.E->TA.
6.A->.+TA
7.A->+.TA
8.A->+T.A
9.A->+TA.
10.A->.e
11.A->e.
12.T->.FB
13.T->F.B
14.T->FB.
15.B->.*FB
16.B->*.FB
17.B->*F.B
18.B->*FB.
19.B->.e
20.B->e.
21.F->.(E)
22.F->(.E)
23.F->(E.)
24.F->(E).
25.F->.i
26.F->i.
```

iii）语法分析结果

```
请输入字符串进行语法分析1：  i#+i#e
是LR(0)方法
正确
```

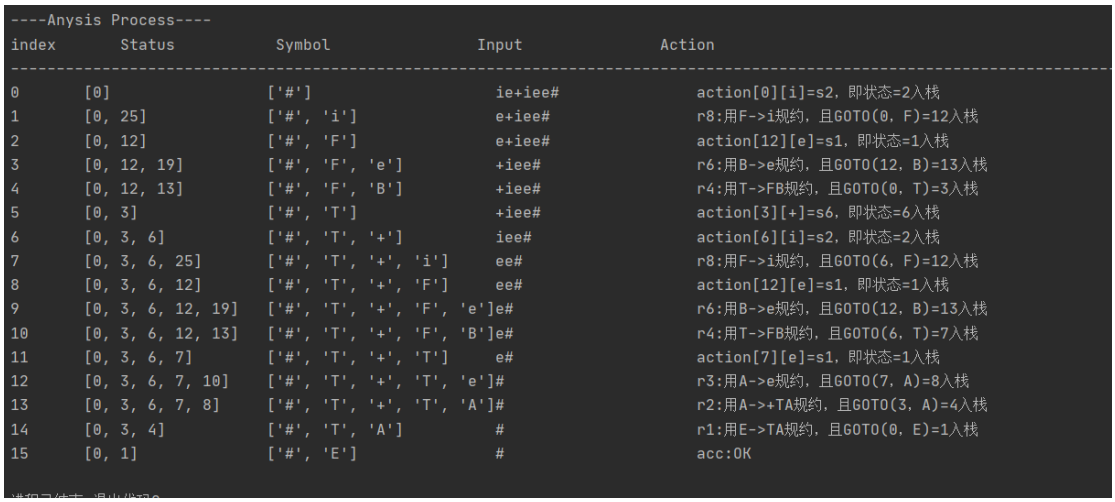
iV）LR 分析表



----LR分析表----

	Action							GOTO				
	+	e	*	(	)	i	#	E	A	T	B	F
0				s21		s25		1		3		12
1							acc					
2												
3	s6	s10							4			
4	r1	r1	r1	r1	r1	r1	r1					
5												
6				s21		s25				7		12
7	s6	s10							8			
8	r2	r2	r2	r2	r2	r2	r2					
9												
10	r3	r3	r3	r3	r3	r3	r3					
11												
12		s19									13	
13	r4	r4	r4	r4	r4	r4	r4					
14												
15				s21		s25						16
16		s19									17	
17	r5	r5	r5	r5	r5	r5	r5					
18												
19	r6	r6	r6	r6	r6	r6	r6					
20												
21				s21		s25			22		3	12
22												
23	r7	r7	r7	r7	r7	r7	r7					
24												
25	r8	r8	r8	r8	r8	r8	r8					

v）正确语句的分析过程

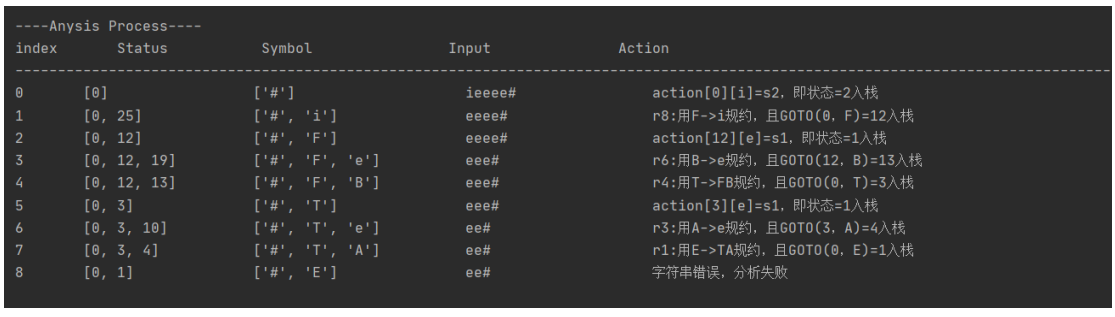


----Anysis Process----

index	Status	Symbol	Input	Action
0	[0]	['#']	ie+iee#	action[0][i]=s2，即状态=2入栈
1	[0, 25]	['#', 'i']	e+iee#	r8:用F->i规约，且GOTO(0，F)=12入栈
2	[0, 12]	['#', 'F']	e+iee#	action[12][e]=s1，即状态=1入栈
3	[0, 12, 19]	['#', 'F', 'e']	+iee#	r6:用B->e规约，且GOTO(12，B)=13入栈
4	[0, 12, 13]	['#', 'F', 'B']	+iee#	r4:用T->FB规约，且GOTO(0，T)=3入栈
5	[0, 3]	['#', 'T']	+iee#	action[3][+]=s6，即状态=6入栈
6	[0, 3, 6]	['#', 'T', '+']	iee#	action[6][i]=s2，即状态=2入栈
7	[0, 3, 6, 25]	['#', 'T', '+', 'i']	ee#	r8:用F->i规约，且GOTO(6，F)=12入栈
8	[0, 3, 6, 12]	['#', 'T', '+', 'F']	ee#	action[12][e]=s1，即状态=1入栈
9	[0, 3, 6, 12, 19]	['#', 'T', '+', 'F', 'e']	e#	r6:用B->e规约，且GOTO(12，B)=13入栈
10	[0, 3, 6, 12, 13]	['#', 'T', '+', 'F', 'B']	e#	r4:用T->FB规约，且GOTO(6，T)=7入栈
11	[0, 3, 6, 7]	['#', 'T', '+', 'T']	e#	action[7][e]=s1，即状态=1入栈
12	[0, 3, 6, 7, 10]	['#', 'T', '+', 'T', 'e']	#	r3:用A->e规约，且GOTO(7，A)=8入栈
13	[0, 3, 6, 7, 8]	['#', 'T', '+', 'T', 'A']	#	r2:用A->+TA规约，且GOTO(3，A)=4入栈
14	[0, 3, 4]	['#', 'T', 'A']	#	r1:用E->TA规约，且GOTO(0，E)=1入栈
15	[0, 1]	['#', 'E']	#	acc:OK

进程已结束，退出代码0

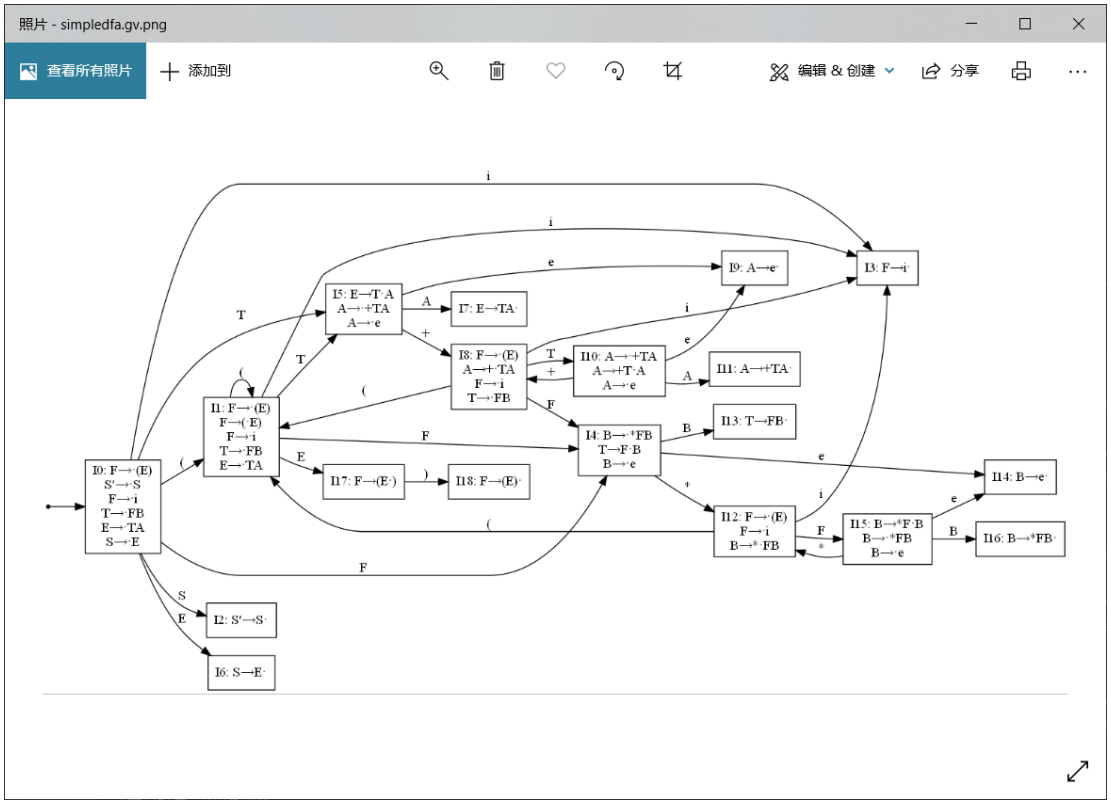
vi）错误语句的分析过程



----Anysis Process----

index	Status	Symbol	Input	Action
0	[0]	['#']	ieeee#	action[0][i]=s2，即状态=2入栈
1	[0, 25]	['#', 'i']	eeee#	r8:用F->i规约，且GOTO(0，F)=12入栈
2	[0, 12]	['#', 'F']	eeee#	action[12][e]=s1，即状态=1入栈
3	[0, 12, 19]	['#', 'F', 'e']	eee#	r6:用B->e规约，且GOTO(12，B)=13入栈
4	[0, 12, 13]	['#', 'F', 'B']	eee#	r4:用T->FB规约，且GOTO(0，T)=3入栈
5	[0, 3]	['#', 'T']	eee#	action[3][e]=s1，即状态=1入栈
6	[0, 3, 10]	['#', 'T', 'e']	ee#	r3:用A->e规约，且GOTO(3，A)=4入栈
7	[0, 3, 4]	['#', 'T', 'A']	ee#	r1:用E->TA规约，且GOTO(0，E)=1入栈
8	[0, 1]	['#', 'E']	ee#	字符串错误，分析失败

vii）生成的 DFA 结果图



（2） 优点分析

对于项目集规范族的构造使用了可视化库 Graphviz (Graph Visualization Software)，能够以图片的形式直观准确的显示程序构造出的自动机，且各模块相互独立，衔接紧密。同时，数据结构的使用极大的减少了空间复杂度，对输入语句的分析速度快，程序性能好，加点后的文法、LR 分析表以及分析过程以表格形式呈现给用户，非常直观。

（3） 缺点分析

当输入错误的语句时，构造分析过程表的模块不能准确的显示出现错误的位置，只能单一的给出语句正确与否的结果，但是经过改进，已解决这一问题，能将前半部分正确的分析过程显示，同时提示错误地分析位置，具体效果可参考运行结果。

九、使用方式

该程序的使用方式如下：

1）将所要分析的文法放置在“syntax-demo.txt”文件中。

2）运行程序即可。

十、总结

通过编译原理实验课，我掌握了什么是编译程序，编译程序工作的基本过程及其各阶段的基本任务，熟悉了编译程序总流程框图，了解了编译程序的生成过程、构造工具及其相关的技术对课本上的知识有了更深的理解，课本上的知识是机械的，表面的。通过把该算法的内容，算法的执行顺序在计算机上实现，把原来以为很深奥的书本知识变的更为简单，对实验原理有更深的理解。

然而，我认为自己在实验课上做的并不够好。首先，在总体实验内容来说，我只完成了词法分析中 LR 分析表的构造与输出;其次，就完成的两个实验来说，功能上还是不够完善，有一些 bug。最后，运行界面上过于简陋，不够美观。由于时间的有限性，这些不足是无法继续改善了。这也让我意识到对于做每一件事，你的付出时间和你的成果是成正比的。所以，对于做一件事，要舍得花时间，肯花时间，这样最后的效果才会更好。对于编译原理，整体看下来自己做得不算好。意识到不足，我更加地明白自己能力不够强，需要在今后的学习中更加深入的理解学习过的知识。

十一、致谢

感谢刘远兴老师对我们的悉心教导以及提供我们这样一个学以致用的机会。通过这一阶段对编译原理课程的学习，特别是通过这次对于词法分析器的编程实践，我对于编译原理中的词法分析这一过程理解的更加清楚明了，收获很大。同时更要感谢小组内成员与我进行积极的沟通，互相帮助解决实验中的难题，没有他们一定不可能顺利完成这次试验。

十二、参考文献

[1] 王一宾. 几种不同 LR 分析表构造方法的分析和比较[J]. 安庆师范学院学报: 自然科学版, 1999, 5(4): 72-73.

[2] 温敬和, 庞艳霞, 王娜. 使用 LR 分析表的词法分析器与分析表最小化[J]. 上海第二工業大學學報, 2007, 24(3): 201-209.

[3] 袁一平. 基于 VC++ 平台的 LR 语法分析器的分析与实现[J]. 长春理工大学学报 (自然科学版), 2007, 30(4).

[4] 金毅, 陆蓓, 王小华. 一种较少状态数的 LR 分析器[J]. 杭州电子科技大学学报, 2006, 26(3): 74-77.

[5] 肖洋, 姜淑娟. 一种 LR 语法分析中的错误恢复方法[J]. 计算机工程, 2007, 33(4): 193-195.