

**杭州电子科技大学**

**《编译原理课程实践》**

**实验报告**

题 目： 实验3语法分析核心算法实现

学 院： 计算机学院

专 业： 计算机科学与技术

班 级： 22052312

学 号： 22050201

姓 名： 黄江晔

完成日期： 2024.11.13

1. **实验目的**

（1）理解上下文无关文法中的左递归、左公共因子、FIRST集和FOLLOW集的概念及其对语法分析的影响。

（2）掌握消去左递归、提取左公共因子，以及计算FIRST集和FOLLOW集的算法。

（3）理解LL(1)文法的概念及应用，并学习设计和实现LL(1)预测分析器。

1. **实验内容与实验要求**

实验内容：

（1）消去左递归：

* 对非终结符集合进行排序。
* 按顺序遍历每个非终结符，检查其候选式是否以排在其前面的非终结符开头，并进行代换。
* 消去直接左递归。

1. 提取左公共因子：

* 对每个非终结符的候选式识别最长的公共前缀。
* 构建字典树（Trie）辅助提取公共前缀，将公共前缀提取为新非终结符的候选式。

1. 计算FIRST集和FOLLOW集：

* 输入上下文无关文法。
* 计算每个非终结符的FIRST集和FOLLOW集。

1. LL(1)文法判定与预测分析器：

* 输入上下文无关文法。
* 判断文法是否为LL(1)。
* 构造预测分析表。
* 实现预测分析器，能够根据输入串进行语法分析。

实验要求：

（1）消去左递归：输入一个上下文无关文法，输出消去左递归后的文法，处理直接和间接左递归，确保输出文法与输入文法等价。

（2）提取左公共因子：输入一个上下文无关文法，输出提取左公共因子后的文法，使用适当的数据结构（如Trie树）提高提取效率，确保输出文法无二义性且与输入文法等价。

（3）计算FIRST集和FOLLOW集：输入一个上下文无关文法，输出每个非终结符的FIRST集和FOLLOW集，算法应考虑文法的各种情况，确保输出结果准确。

（4）LL(1)文法判定与预测分析器：在前述任务的基础上判断文法是否为LL(1)，输出文法是否为LL(1)的判断结果，输出预测分析表，并输入一个字符串，输出语法分析结果（是否成功以及分析过程）。

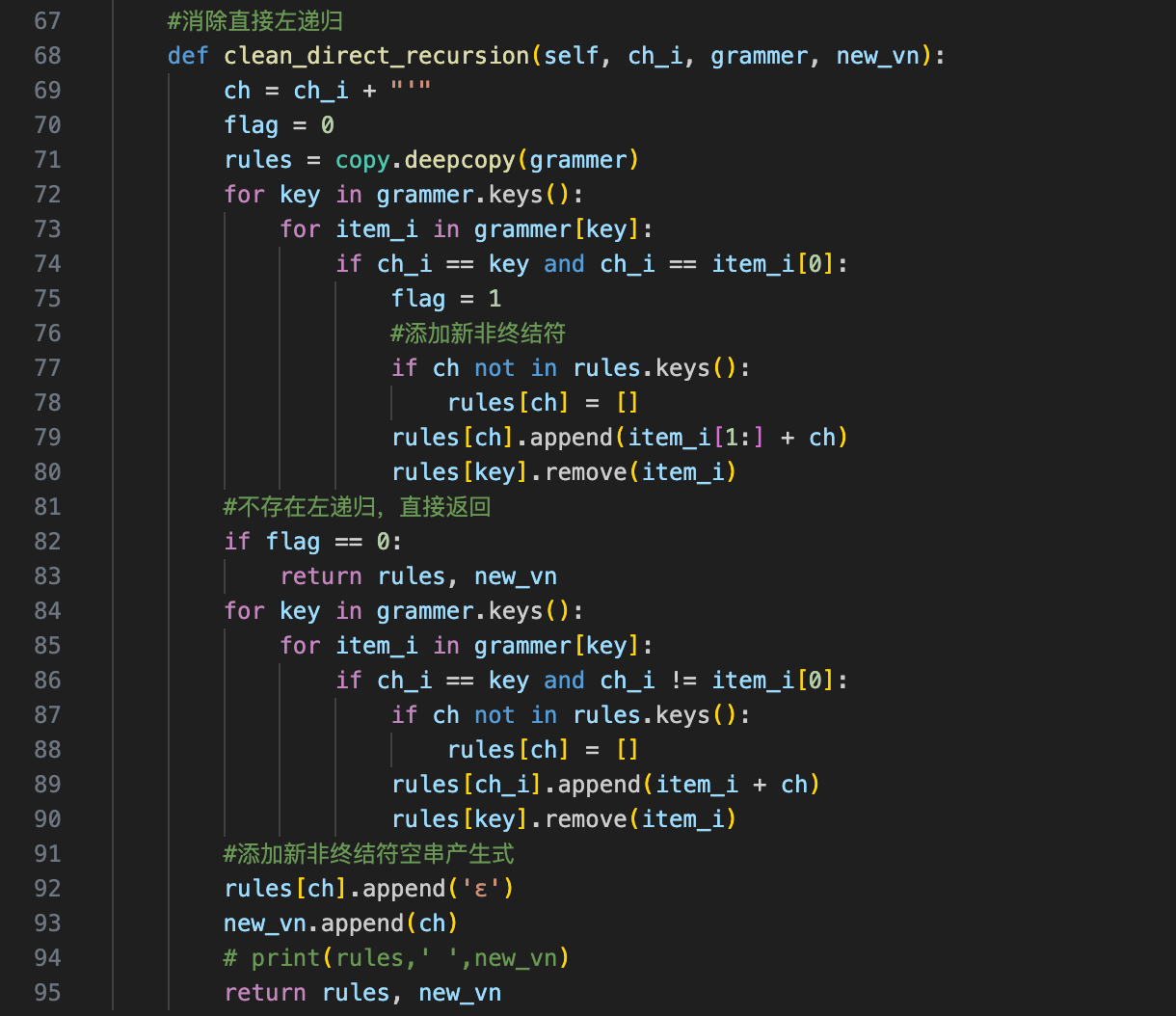
1. **设计方案与算法描述**

一、消除上下文无关文法中的左递归

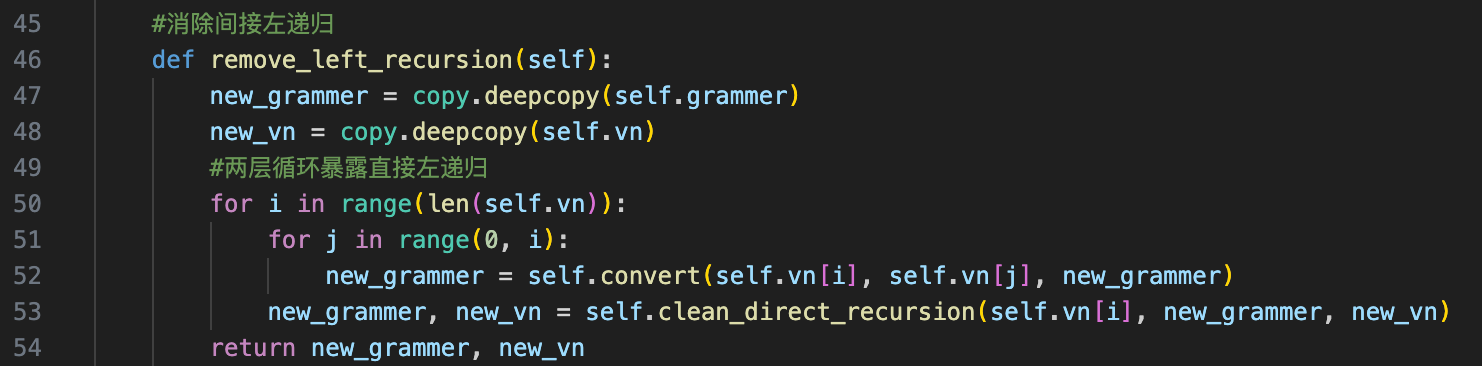
1. 识别直接左递归：⽂法规则形式为 A -> Aα|β ，其中A是⾮终结符，α和β是任意的⽂法符号序列（终结符或⾮终结符），并且⾄少有⼀个β不以 A 开头。要消除直接左递归，可以引⼊⼀个新的⾮终结符A' 来替换原产⽣式的左递归部分，并重新定义 A 的产⽣式如下：

A ->βA'

A' ->αA' |ε



1. 识别间接左递归：间接左递归的⽂法规则可能没有直接的形式 A -> Aα，但是通过⼀系列的推导，A可以推导出以A开头的字符串。消除间接左递归通常需要将间接左递归转换为直接左递归，然后应⽤直接左递归的消除⽅法。



1. 转换产⽣式：对于每个存在左递归的⾮终结符，引⼊新的⾮终结符作为辅助符号，并重新定义产⽣式以消除左递归。



重点难点：

1. 设计⽂法的数据结构，特别是产⽣式规则的左部和右部如何表示，才能更⽅便进⾏替换

文法数据结构设计：文法被表示为字典，键为非终结符，值为该非终结符的所有产生式的列表。每个产生式是由终结符和非终结符组成的字符串或列表。这种设计结构便于进行文法的转换和修改。

产生式规则的左部与右部表示：文法的左部是非终结符（如 A），右部则是由多个候选项（产生式）组成，使用 | 分隔。每个候选项可以是由终结符和非终结符组成的字符串（例如 aB 或 a）。

右部非终结符转终结符：在 convert 方法中，通过匹配产生式右部的非终结符与已有的其他产生式进行替换，确保在消除间接左递归时能够适当合并产生式。

2. ⽂法输⼊形式，并转换为相应的数据结构

文法输入和转换：在 LL1\_analysis 类的 init\_all\_ 方法中，代码读取文法字符串并将其转换为字典数据结构。每一行文法被解析为非终结符和相应的产生式，并存储到字典 grammer\_list 中。

消除左递归：在 init\_all\_ 中，调用 EliminateLeftRecursion 类的 remove\_left\_recursion 方法，首先对文法进行间接左递归的消除，然后再通过 clean\_direct\_recursion 方法消除直接左递归，最终得到一个无左递归的文法。

3. 如何验证转换出来的⽂法与原始⽂法是等价的？【可探讨】

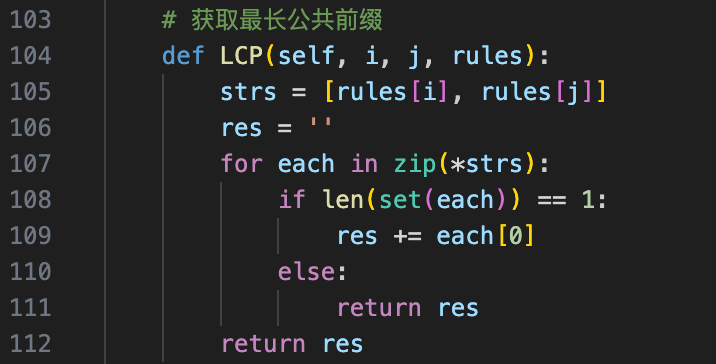
直接左递归与间接左递归的处理：在 EliminateLeftRecursion 类中，消除左递归的过程分为两步：首先是处理间接左递归，之后是消除直接左递归。在消除直接左递归时，新非终结符（如 A'）会被引入，并且通过对原产生式的转换、替换和分离来消除递归。这种方式确保了文法的等价性，因为虽然文法形式发生了变化，但每个文法的语言（可以生成的句子）保持不变。

空串产生式的添加：在直接左递归的处理过程中，如果产生式中有非终结符自递归（如 A -> Aa | b），会引入新的非终结符（如 A'）并将空串产生式（A' -> ε）加入文法，这样保证了文法的等价性，即原文法和新文法能生成相同的字符串

二、提取上下文无关文法左公共因子

1. 计算公共前缀

LCP 方法：计算两个产生式右部的最长公共前缀的函数。接收两个产生式的右部（通过其索引在 rules 列表中获得），逐字符地比较它们，直到找到不相等的字符为止。返回这个公共前缀。



1. 获取公共前缀的索引

get\_lcp\_res 方法：遍历当前非终结符的所有产生式（rules），并尝试找出它们之间的公共前缀。对于每一对产生式，调用 LCP 方法计算它们的公共前缀，并将这些公共前缀存储到字典 res 中，键是公共前缀，值是包含这个前缀的产生式索引的集合。



3. 消除公共因子

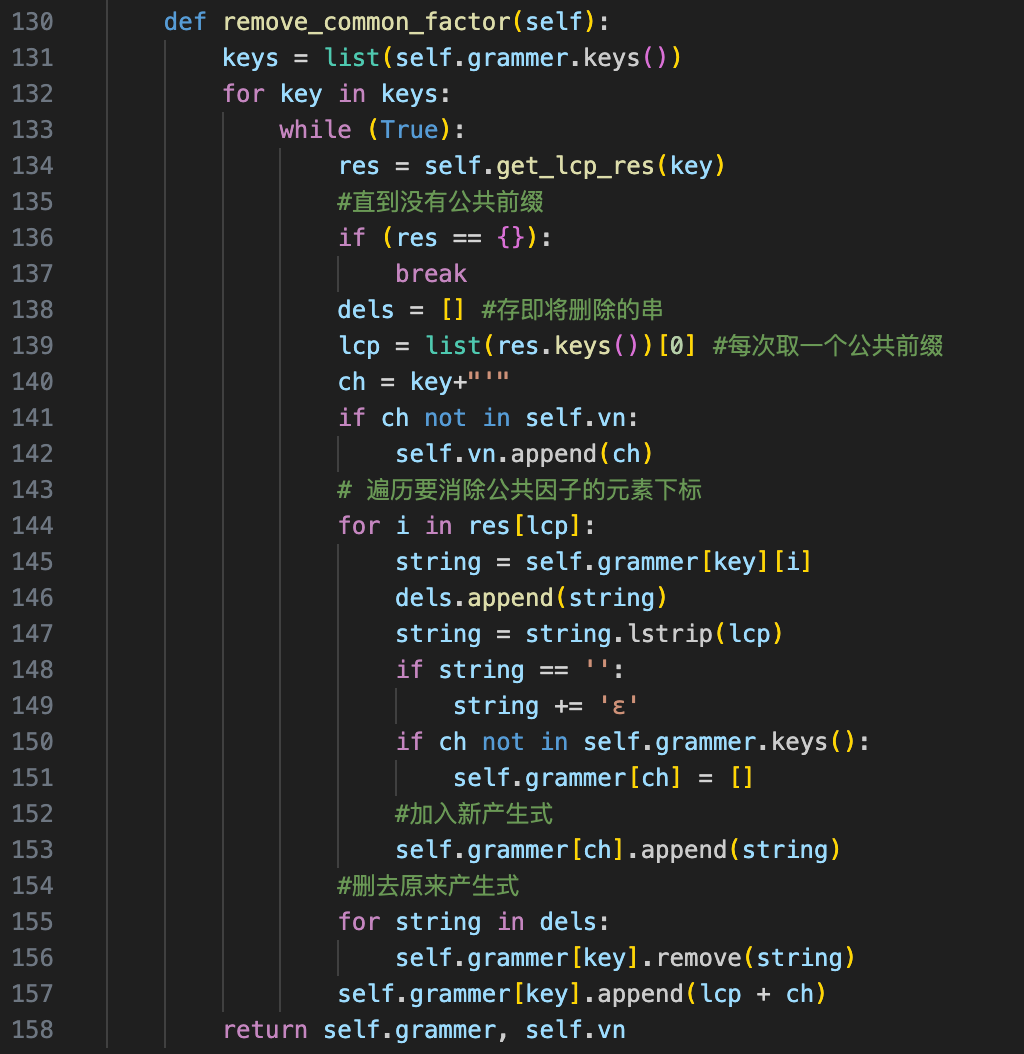
remove\_common\_factor 方法：遍历所有非终结符（key），并通过 get\_lcp\_res 方法查找它们产生式的公共前缀。对于每个公共前缀，执行以下操作：

新非终结符的引入：如果公共前缀存在，则为这个非终结符引入一个新非终结符（key'），并将其添加到非终结符集合中。

创建新产生式：将有公共前缀的产生式右部（去掉公共前缀）作为新产生式加入到新非终结符的产生式列表中。如果右部为空串，则替换为空串（ε）。

修改原产生式：在原非终结符的产生式中，去除有公共前缀的部分，加入新的产生式（即 lcp + key'）。

重复此过程：直到所有公共前缀都被提取和处理。



三、计算FIRST集和FOLLOW集

（1）整体逻辑：

FIRST集

对于文法的每个非终结符A，FIRST(A)是可以从A的产生式推导出的第一个终结符集。

如果A → ε，则将ε加入FIRST(A)。

如果A → B1B2...Bn，并且B1的FIRST集包含终结符，则将B1的FIRST集加入A的FIRST集。

FOLLOW集

对于文法中的每个非终结符A，FOLLOW(A)是所有可以跟随在A之后的终结符的集合。

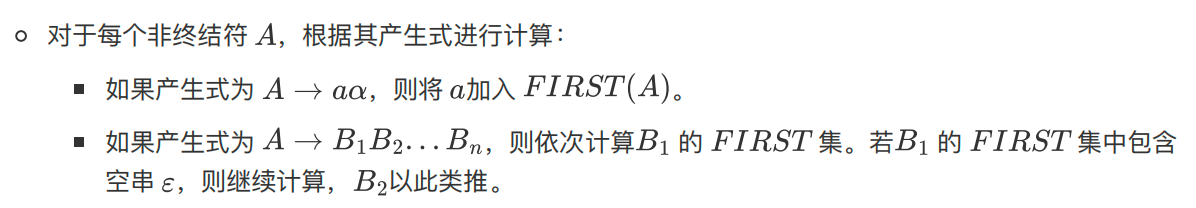
如果A是文法的开始符号，则将$（表示输入结束符）加入FOLLOW(A)。

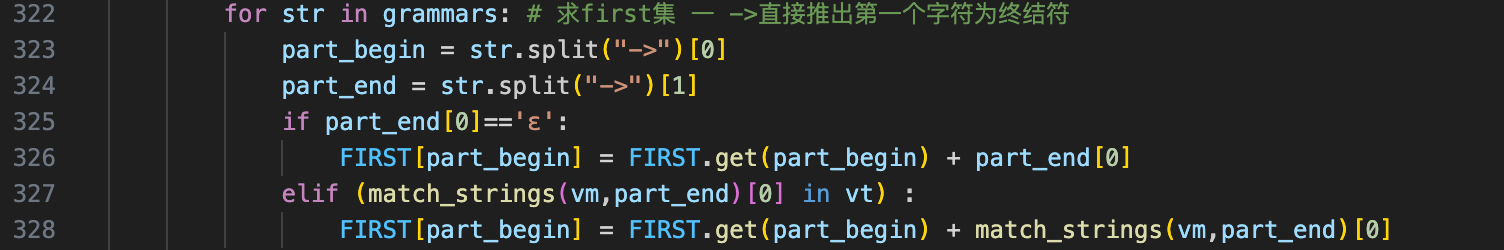
如果A → αBβ，则将FIRST(β)（除去ε）加入FOLLOW(B)。

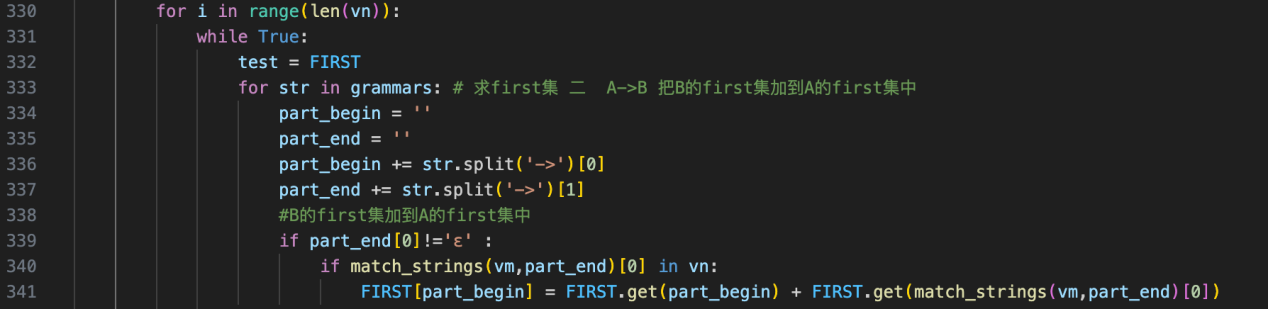
（2）FIRST集的计算：

1.初始化：为每个非终结符初始化一个空的 First 集。

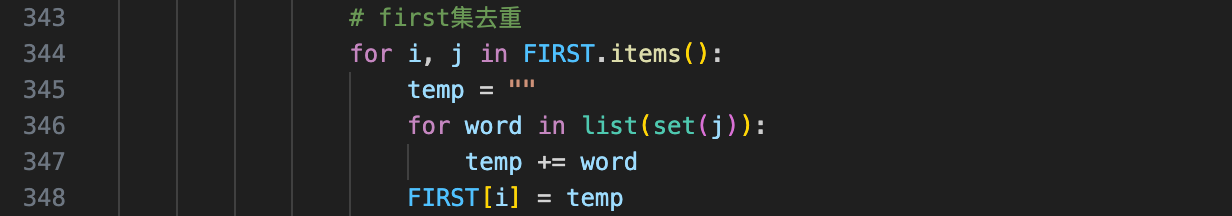
2.按以下两条规则分两部分计算：



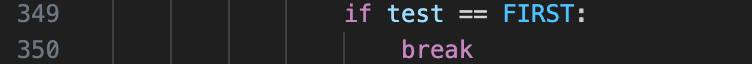




1. 去重：对每个非终结符的First集去重，确保没有重复的元素。



1. 收敛：在递归计算First集时，如果在某一轮计算后First集没有变化（即 test == FIRST），则退出循环，表示已经计算完成。



（3）FOLLOW集的计算：

1. 初始化：对于文法的开始符号，将其Follow集初始化为 "$"（表示输入的结束符号）。其他非终结符的Follow集初始化为空字符串。

2. 递归计算Follow集

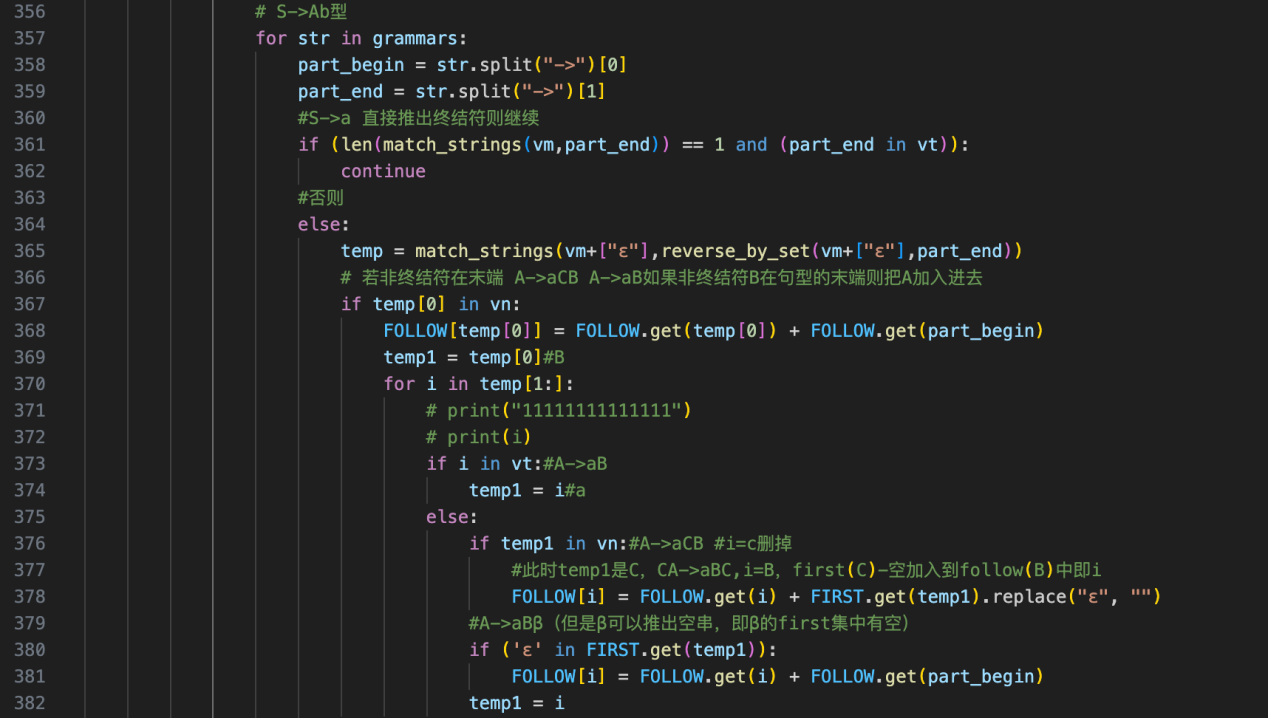
通过迭代法来计算 Follow 集，这个过程依赖于规则：

如果某个非终结符A推导出一个产生式A ->αBβ，且B是非终结符，则Follow(B)应该包含Follow(A)（即A的Follow集被传递给B）。

如果某个非终结符A推导出一个产生式A ->αB，且B是非终结符且其右侧没有符号（即B在产生式的末尾），那么Follow(B)应该包含 Follow(A)。

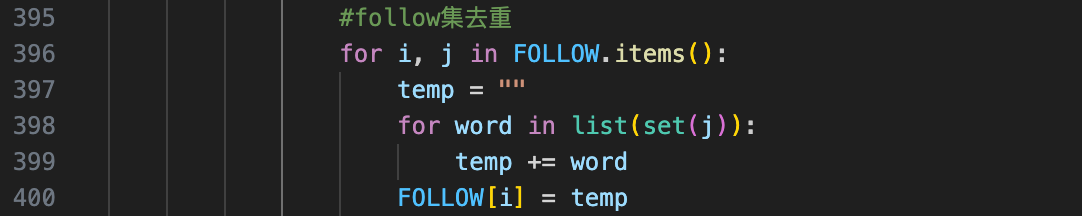
如果B后面紧跟着某个符号β，且β可推导出空串（即β的First集包含ε），则 Follow(B)应该也包含Follow(A)。

若右边的符号是终结符，或者产生式的右部以终结符开始，那么终结符会被加入到 Follow集中。



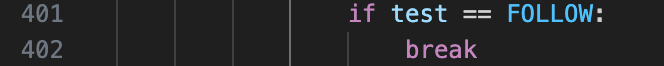


3.去重：对每个非终结符的Follow集去重，确保没有重复的元素。

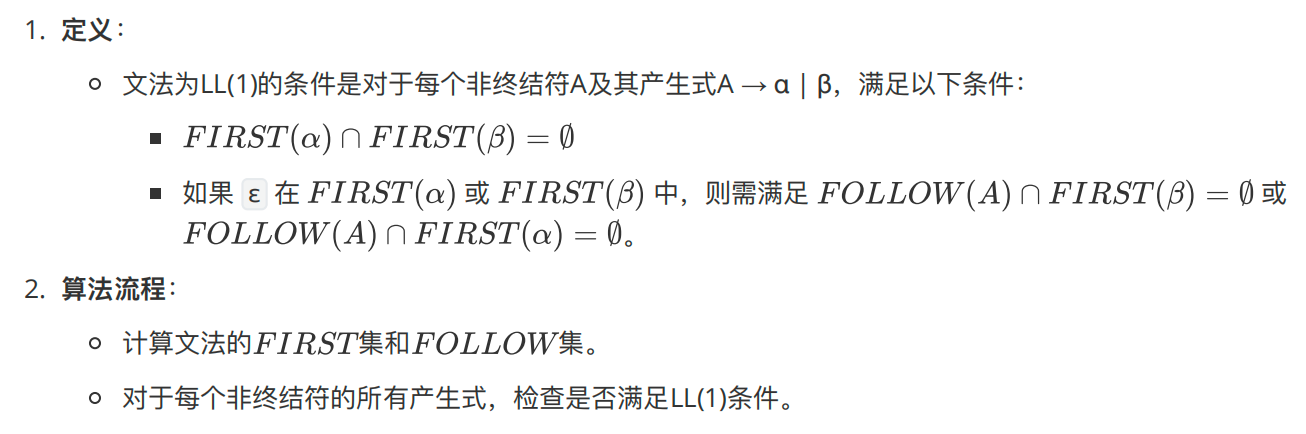


4. 检查是否收敛

在计算过程中，使用一个 test 变量来记录上一轮计算前的 Follow 集。如果某一轮计算后，Follow 集 没有发生变化（即 test == FOLLOW），则退出循环，表示计算已经完成。



四、LL(1)文法的判定



1. 初始化判定标志

首先，定义一个 is\_ll1 标志来表示是否为 LL(1) 文法。初始化为 True，表示默认认为该文法是 LL(1) 文法，直到发现冲突为止。

2. 遍历分析表

然后，遍历 LL(1) 分析表，检查每个位置是否有冲突。分析表是一个二维数组 analysis\_table，其中每个位置可能存放某个非终结符的产生式或 None（表示该位置没有产生式）。

外层循环遍历所有非终结符（通过 new\_vn），内层循环遍历所有终结符（通过 new\_vt）。

如果在某个位置 analysis\_table[i][j] 存在值，表示该位置已经填充了一个产生式，接下来检查是否有冲突。

3. 检查是否存在冲突

当 analysis\_table[i][j] 存在值时，说明该非终结符和终结符的组合已经被填入了分析表。

然后，你需要检查同一非终结符 new\_vn[i] 在相同的终结符 new\_vt[j] 下是否有其他的产生式。如果存在相同位置（即 analysis\_table[k][j] == analysis\_table[i][j]）的其他产生式，则发生冲突，说明该文法不是 LL(1) 文法。

4. 判断并退出

如果发现冲突，即 is\_ll1 被设置为 False，则退出循环，表示文法不是 LL(1) 文法。

外层和内层循环都检查完成后，判断 is\_ll1 的值：

如果 is\_ll1 仍然为 True，说明没有冲突，文法是 LL(1) 文法。

否则，文法不是 LL(1) 文法。

五、构造预测分析表

1.总体逻辑：

构造LL(1)预测分析表，对于每个非终结符A和终结符a，填入合适的产生式A→α。

根据输入串和预测分析表进行语法分析，使用栈结构逐步推导。

2.初始化分析表，遍历文法产生式

遍历每个非终结符 new\_vn[i]，然后针对该非终结符的每个产生式 t，填充分析表中的相应位置。

* 处理 ε 产生式

如果当前产生式是 ε，则表示该非终结符能够推导出空串。此时，需要填充分析表中该非终结符和 FOLLOW 集合中的每个终结符的交集位置为 ε。



* 处理非 ε 产生式

如果产生式 t 不是 ε，则遍历产生式右侧的每个符号，分为以下几种情况：

第一，如果符号是终结符，直接将该符号填入分析表相应的位置。

第二，如果符号是非终结符，则使用该非终结符的 FIRST 集合中的元素：

遍历该非终结符的 FIRST 集合，将其中非 ε 的元素填入相应位置。

如果该非终结符的 FIRST 集合包含 ε，则继续检查后续的符号。

* 处理右侧能推导出 ε 的情况

如果产生式右侧的所有符号都能够推导出 ε，则需要检查该非终结符的 FOLLOW 集合，将 FOLLOW 集合中的符号填入相应的分析表位置。





六、实现预测分析器

1. 分析器的基本结构

LL1\_analysis\_solve 函数是整个预测分析器的核心函数。使用一个栈和一个输入字符串来模拟文法的自顶向下分析，通过 LL(1) 分析表来决定如何进行推导。

2. 参数说明

goal\_str: 目标输入串，通常是要分析的字符串。

ans\_table: 用于记录分析过程和结果的表格。

vt: 终结符集合。

vn: 非终结符集合。

analysis\_table: 预先构造的 LL(1) 分析表，用于查找每个非终结符和终结符的匹配产生式。

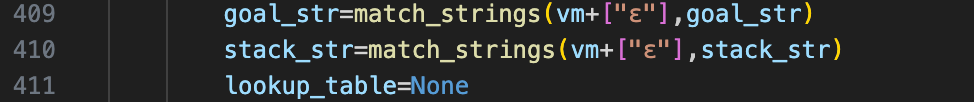
stack\_str: 模拟栈，栈底存放 $ 表示输入结束。

ptr: 指向当前输入符号的位置。

3. 初始化过程

将 goal\_str 和 stack\_str 与终结符、非终结符集合进行匹配（match\_strings(vm + ["ε"], goal\_str)），这应该是确保输入的合法性或做符号处理。

初始化一个空的 lookup\_table，该表格用来存放当前分析状态的匹配信息。



4. 主循环

这个循环是 LL(1) 预测分析的核心，分析器在栈顶符号和输入符号的帮助下逐步分析目标字符串，直到分析成功或失败。

4.1. 检查非法输入

首先，分析器会检查栈顶符号和当前输入符号是否合法：如果栈顶符号和当前输入符号都不在终结符或非终结符中，输入就被视为非法，返回错误信息。

4.2. 栈顶符号与当前输入符号匹配

如果栈顶符号和当前输入符号相同，且栈顶符号为 $（表示字符串结束），则分析成功。

如果栈顶符号和输入符号相同，但不为 $，则从栈中弹出栈顶符号并前进输入指针。

4.3. 查找分析表

如果栈顶符号和当前输入符号不相同，则需要查找分析表，找到匹配的产生式：

查找栈顶符号的索引：根据栈顶符号在非终结符或终结符集合中的位置查找其索引。

查找当前输入符号的索引：根据当前输入符号在终结符集合中的位置查找其索引。

查找分析表中的对应产生式：根据栈顶符号和当前输入符号的索引，查找 LL(1) 分析表中的对应值。

4.4. 分析表匹配成功

如果找到了匹配的产生式（lookup\_table 不是 None），则根据该产生式进行相应的操作：

如果产生式是 ε，弹出栈顶符号。

如果产生式是其他符号，则将该产生式右侧的符号反向压入栈中。

4.5. 分析失败

如果在分析表中找不到匹配的产生式（即 lookup\_table 为 None），则分析失败。

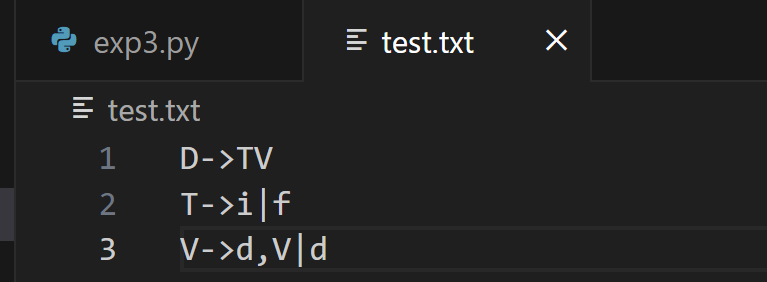
5. 输出分析过程

在每一步操作中，分析器都会记录当前栈内容、剩余输入串以及对应的产生式，并将其保存到 ans\_table 中。

详细代码见第五部分源代码。

1. **测试结果**

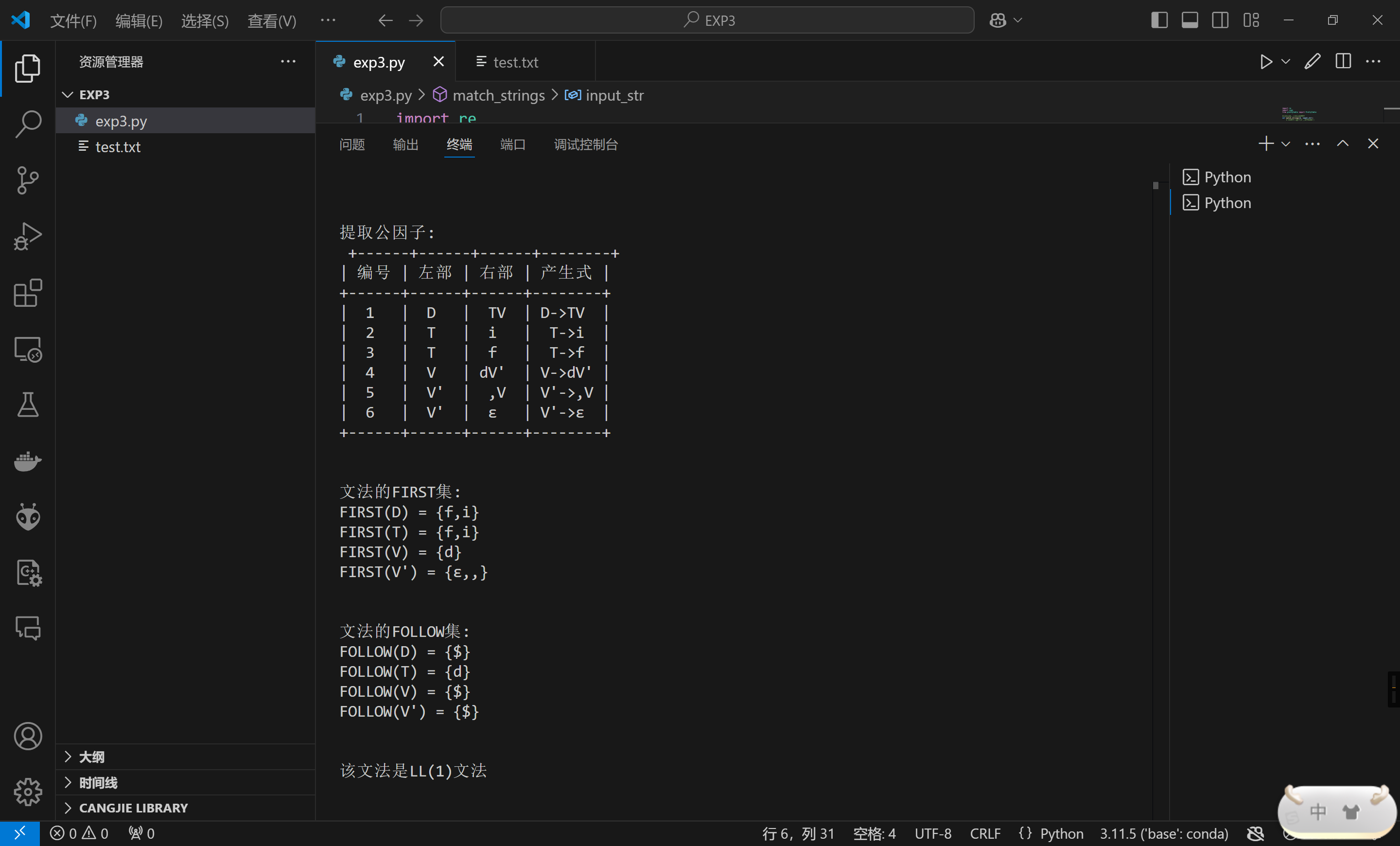
测试用例：

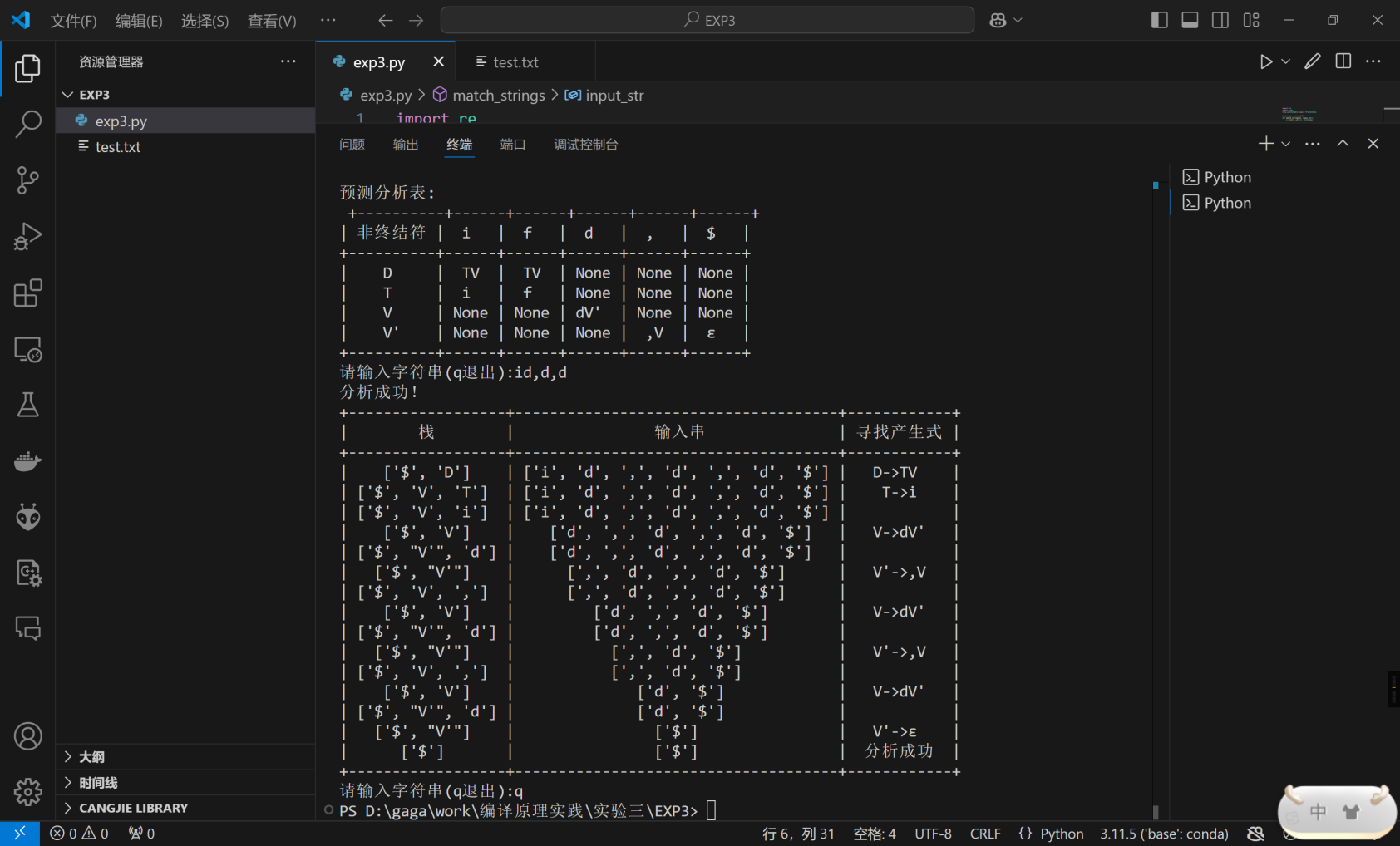


消除左递归：



提取左公因子、计算first集和follow集、预测分析表、预测分析器：





**五、源代码**

1. import re
2. import copy
3. from prettytable import PrettyTable
4. #按终结符和非终结符遍历
5. def match\_strings(A, input\_str):
6. # 优先匹配最长(A'和A识别成A')
7. A = sorted(A, key=lambda x: len(x), reverse=True)
8. pattern = '|'.join(map(re.escape, A))
9. matches = re.findall(pattern, input\_str)
10. return matches
11. #按终结符非终结符整体字符串倒序
12. def reverse\_by\_set(A, input\_str):
13. result = []
14. i = len(input\_str)
15. while i > 0:
16. for word in reversed(A):
17. word\_len = len(word)
18. if i >= word\_len and input\_str[i - word\_len:i] == word:
19. result.append(word)
20. i -= word\_len
21. break
22. else:
23. i -= 1
24. return ''.join(result)
25. #可视化输出
26. class draw\_grammer:
27. def draw\_grammer(grammer,vn,descrpition):
28. print\_content = PrettyTable(['编号', '左部', '右部', '产生式'])
29. idx = 1
30. for i in vn:
31. for j in grammer[i]:
32. print\_content.add\_row([idx, i, j, i + '->' + j])
33. idx += 1
34. print('\n\n'+descrpition+':\n', print\_content)
35. #消除左递归
36. class EliminateLeftRecursion:
37. def \_\_init\_\_(self, grammer, vn):
38. self.grammer = grammer
39. self.vn = vn
40. #消除间接左递归
41. def remove\_left\_recursion(self):
42. new\_grammer = copy.deepcopy(self.grammer)
43. new\_vn = copy.deepcopy(self.vn)
44. #两层循环暴露直接左递归
45. for i in range(len(self.vn)):
46. for j in range(0, i):
47. new\_grammer = self.convert(self.vn[i], self.vn[j], new\_grammer)
48. new\_grammer, new\_vn = self.clean\_direct\_recursion(self.vn[i], new\_grammer, new\_vn)
49. return new\_grammer, new\_vn
51. #产生式右部非终结符转终结符
52. def convert(self, ch\_i, ch\_j, grammer):
53. rules = copy.deepcopy(grammer)
54. for key in grammer.keys():
55. for item\_i in grammer[key]:
56. if ch\_i == key and ch\_j == item\_i[0]:
57. rules[key].remove(item\_i)
58. for item\_j in grammer[ch\_j]:
59. rules[key].append(item\_j + item\_i[1:])
60. return rules
61. #消除直接左递归
62. def clean\_direct\_recursion(self, ch\_i, grammer, new\_vn):
63. ch = ch\_i + "'"
64. flag = 0
65. rules = copy.deepcopy(grammer)
66. for key in grammer.keys():
67. for item\_i in grammer[key]:
68. if ch\_i == key and ch\_i == item\_i[0]:
69. flag = 1
70. #添加新非终结符
71. if ch not in rules.keys():
72. rules[ch] = []
73. rules[ch].append(item\_i[1:] + ch)
74. rules[key].remove(item\_i)
75. #不存在左递归，直接返回
76. if flag == 0:
77. return rules, new\_vn
78. for key in grammer.keys():
79. for item\_i in grammer[key]:
80. if ch\_i == key and ch\_i != item\_i[0]:
81. if ch not in rules.keys():
82. rules[ch] = []
83. rules[ch\_i].append(item\_i + ch)
84. rules[key].remove(item\_i)
85. #添加新非终结符空串产生式
86. rules[ch].append('ε')
87. new\_vn.append(ch)
88. # print(rules,' ',new\_vn)
89. return rules, new\_vn
90. #提取左公因子
91. class ExtractCommonFactors:
92. def \_\_init\_\_(self, grammer, vn):
93. self.grammer = grammer
94. self.vn = vn
96. # 获取最长公共前缀
97. def LCP(self, i, j, rules):
98. strs = [rules[i], rules[j]]
99. res = ''
100. for each in zip(\*strs):
101. if len(set(each)) == 1:
102. res += each[0]
103. else:
104. return res
105. return res
106. #获取公共前缀索引
107. def get\_lcp\_res(self, key):
108. res = {}
109. rules = self.grammer[key]
110. for i in range(len(rules)):
111. for j in range(i+1, len(rules)):
112. temp = self.LCP(i,j,rules)
113. if temp not in res.keys():
114. res[temp] = set()
115. res[temp].add(i)
116. res[temp].add(j)
117. #去空串前缀
118. if '' in res.keys():
119. res.pop('')
120. return res
121. def remove\_common\_factor(self):
122. keys = list(self.grammer.keys())
123. for key in keys:
124. while (True):
125. res = self.get\_lcp\_res(key)
126. #直到没有公共前缀
127. if (res == {}):
128. break
129. dels = [] #存即将删除的串
130. lcp = list(res.keys())[0] #每次取一个公共前缀
131. ch = key+"'"
132. if ch not in self.vn:
133. self.vn.append(ch)
134. # 遍历要消除公共因子的元素下标
135. for i in res[lcp]:
136. string = self.grammer[key][i]
137. dels.append(string)
138. string = string.lstrip(lcp)
139. if string == '':
140. string += 'ε'
141. if ch not in self.grammer.keys():
142. self.grammer[ch] = []
143. #加入新产生式
144. self.grammer[ch].append(string)
145. #删去原来产生式
146. for string in dels:
147. self.grammer[key].remove(string)
148. self.grammer[key].append(lcp + ch)
149. return self.grammer, self.vn
150. #文法分析
151. class LL1\_analysis:
152. def \_\_init\_\_(self, Gram):
153. #终结符 非终结符 分析表元素 $+开始符号
154. self.vt, self.vn, self.analysis\_table, self.stack\_str = self.init\_all\_(g=Gram)
155. self.ptr = 0
156. def init\_all\_(self, g):
157. #读取文法
158. grammer\_list = {}  #非终结符：产生式
159. vn\_list = []  #非终结符
160. for line in re.split('\n', g):
161. # 去空格
162. line = "".join([i for i in line if i not in ['', ' ']])
163. if '->' in line:
164. if line.split('->')[0] not in vn\_list:
165. vn\_list.append(line.split('->')[0])
166. for i in line.split('->')[1].split('|'):
167. if grammer\_list.get(line.split('->')[0]) is None:
168. grammer\_list[line.split('->')[0]] = []
169. grammer\_list[line.split('->')[0]].append(i)
170. else:
171. grammer\_list[line.split('->')[0]].append(i)
172. draw\_grammer.draw\_grammer(grammer=grammer\_list, vn=vn\_list, descrpition='输入的文法')
173. #消除左递归
174. # print('产生式：', grammer\_list)
175. # print('非终结符：', vn\_list)
176. eliminate\_left\_recursion = EliminateLeftRecursion(grammer=grammer\_list, vn=vn\_list)
177. new\_grammer, new\_vn = eliminate\_left\_recursion.remove\_left\_recursion()
178. draw\_grammer.draw\_grammer(grammer=new\_grammer, vn=new\_vn, descrpition='消除左递归')
179. #提取左公因子
180. extractcommonfactors = ExtractCommonFactors(grammer=new\_grammer, vn=new\_vn)
181. new\_grammer, new\_vn = extractcommonfactors.remove\_common\_factor()
182. draw\_grammer.draw\_grammer(grammer=new\_grammer, vn=new\_vn, descrpition='提取公因子')
183. only\_grammer = []
184. new\_vt = []
185. for i in new\_vn:
186. for j in new\_grammer[i]:
187. only\_grammer.append(i + '->' + j)
189. for t in j:  # 获取当前的所有的终结符
190. if t not in new\_vt and t not in new\_vn and t!="ε" and t!="'":
191. # print(t)
192. new\_vt.append(t)
193. new\_vt.append('$')
194. # print('\n\n消除文法左递归的文法的非终结符:',new\_vn,
195. #       '\n\n消除文法左递归的文法的终结符:', new\_vt)
196. #FIRST集和FOLLOW集
197. FIRST, FOLLOW = self.get\_first\_and\_follow\_set(grammars=only\_grammer,vn=new\_vn,vt=new\_vt)
198. print('\n\n文法的FIRST集:')
199. for i, j in FIRST.items():
200. str = j[0]
201. for temp in j[1:]:
202. str = str + ',' + temp
203. print("FIRST(" + i + ")" + " = {" + str + "}")
204. print('\n\n文法的FOLLOW集:')
205. for i, j in FOLLOW.items():
206. str = j[0]
207. for temp in j[1:]:
208. str = str + ',' + temp
209. print("FOLLOW(" + i + ")" + " = {" + str + "}")
210. #分析表
211. analysis\_table = [[None] \* (1 + len(new\_vt)) for row in range(1 + len(new\_vn))]
212. analysis\_table[0][0] = ' '
213. for i in range(len(new\_vt)):
214. analysis\_table[0][i + 1] = new\_vt[i]
215. for i in range(len(new\_vn)):
216. analysis\_table[i + 1][0] = new\_vn[i]
217. for i in range(len(new\_vn)):
218. for t in new\_grammer[new\_vn[i]]:  # 遍历该文法的所有产生式
219. if t == 'ε':  # 如果是ε，对应在FOLLOW集中的终结符位置填上ε
220. for j in range(len(new\_vt)):  # 遍历所有的终结符
221. if new\_vt[j] in FOLLOW[new\_vn[i]]:  # FOLLOW[part\_begin]为当前非终结符的FOLLOW集
222. # 如果分析表该位置为空，则填入ε
223. if analysis\_table[i + 1][j + 1] is None:
224. analysis\_table[i + 1][j + 1] = 'ε'
225. else:
226. first\_found = False  # 用于标记是否已经找到有效的FIRST项
227. for symbol in t:  # 遍历产生式右侧的每个符号
228. if symbol in new\_vt:  # 如果是终结符
229. # 将该符号填入对应位置
230. j = new\_vt.index(symbol)
231. if analysis\_table[i + 1][j + 1] is None:
232. analysis\_table[i + 1][j + 1] = t
233. first\_found = True
234. break  # 终结符就直接填入，并停止检查其他符号
235. else:  # 如果是非终结符
236. # 使用该非终结符的FIRST集
237. for first\_symbol in FIRST[symbol]:
238. if first\_symbol != 'ε':  # 只处理非ε项
239. j = new\_vt.index(first\_symbol)
240. if analysis\_table[i + 1][j + 1] is None:
241. analysis\_table[i + 1][j + 1] = t
242. # 如果该非终结符的FIRST集包含ε，需要继续检查后面的符号
243. if 'ε' in FIRST[symbol]:
244. continue
245. else:
246. first\_found = True
247. break  # 如果FIRST集没有包含ε，停止检查后面的符号
248. # 如果右侧符号都能推导出ε，则检查FOLLOW集并填充
249. if not first\_found:
250. for j in range(len(new\_vt)):
251. if new\_vt[j] in FOLLOW[new\_vn[i]]:
252. if analysis\_table[i + 1][j + 1] is None:
253. analysis\_table[i + 1][j + 1] = 'ε'
254. #判断是否为LL（1）文法
255. is\_ll1 = True
256. for i in range(1, len(new\_vn) + 1):
257. for j in range(1, len(new\_vt) + 1):
258. if analysis\_table[i][j] is not None:  # 如果当前位置有值 检查是否冲突
259. for k in range(i + 1, len(new\_vn) + 1):  # 对比同一非终结符的其他产生式
260. if analysis\_table[k][j] == analysis\_table[i][j]:
261. is\_ll1 = False
262. break
263. if not is\_ll1:
264. break
265. if not is\_ll1:
266. break
267. if is\_ll1:
268. print("\n\n该文法是LL(1)文法")
269. else:
270. print("\n\n该文法不是LL(1)文法")
271. #输出分析表
272. pretty\_table\_title = ['非终结符']
273. for i in new\_vt:
274. pretty\_table\_title.append(i)
275. analysis\_pretty\_table = PrettyTable(pretty\_table\_title)
276. for i in range(len(analysis\_table) - 1):
277. analysis\_pretty\_table.add\_row(analysis\_table[i + 1])
278. print('\n\n预测分析表:\n', analysis\_pretty\_table)
279. #返回预处理结构
280. # print("new\_vn:",new\_vn[0])
281. return new\_vt, new\_vn, analysis\_table, '$' + new\_vn[0]
282. def get\_first\_and\_follow\_set(self,grammars,vn,vt):
283. FIRST = {}
284. FOLLOW = {}
285. index=0
286. for str in grammars: # 初始化first、follow集
287. # print(str)
288. part\_begin = str.split("->")[0]
289. part\_end = str.split("->")[1]
290. FIRST[part\_begin] = ""
291. if index==0:
292. FOLLOW[part\_begin]="$"
293. else:
294. FOLLOW[part\_begin] = ""
295. index+=1
296. # print(part\_begin,FOLLOW[part\_begin])
297. #first集
298. vm=vt+vn
299. # print(match\_strings(vm,grammars[1].split("->")[1]))
300. for str in grammars: # 求first集 一 ->直接推出第一个字符为终结符
301. part\_begin = str.split("->")[0]
302. part\_end = str.split("->")[1]
303. if part\_end[0]=='ε':
304. FIRST[part\_begin] = FIRST.get(part\_begin) + part\_end[0]
305. elif (match\_strings(vm,part\_end)[0] in vt) :
306. FIRST[part\_begin] = FIRST.get(part\_begin) + match\_strings(vm,part\_end)[0]
307. for i in range(len(vn)):
308. while True:
309. test = FIRST
310. for str in grammars: # 求first集 二  A->B 把B的first集加到A的first集中
311. part\_begin = ''
312. part\_end = ''
313. part\_begin += str.split('->')[0]
314. part\_end += str.split('->')[1]
315. #B的first集加到A的first集中
316. if part\_end[0]!='ε' :
317. if match\_strings(vm,part\_end)[0] in vn:
318. FIRST[part\_begin] = FIRST.get(part\_begin) + FIRST.get(match\_strings(vm,part\_end)[0])
319. # first集去重
320. for i, j in FIRST.items():
321. temp = ""
322. for word in list(set(j)):
323. temp += word
324. FIRST[i] = temp
325. if test == FIRST:
326. break
327. #follow集
328. for i in range(len(vn)):
329. while True:
330. test = FOLLOW
331. # S->Ab型
332. for str in grammars:
333. part\_begin = str.split("->")[0]
334. part\_end = str.split("->")[1]
335. #S->a 直接推出终结符则继续
336. if (len(match\_strings(vm,part\_end)) == 1 and (part\_end in vt)):
337. continue
338. #否则
339. else:
340. temp = match\_strings(vm+["ε"],reverse\_by\_set(vm+["ε"],part\_end))
341. # 若非终结符在末端 A->aCB A->aB如果非终结符B在句型的末端则把A加入进去
342. if temp[0] in vn:
343. FOLLOW[temp[0]] = FOLLOW.get(temp[0]) + FOLLOW.get(part\_begin)
344. temp1 = temp[0]#B
345. for i in temp[1:]:
346. # print("11111111111111")
347. # print(i)
348. if i in vt:#A->aB
349. temp1 = i#a
350. else:
351. if temp1 in vn:#A->aCB #i=c删掉
352. #此时temp1是C，CA->aBC,i=B，first(C)-空加入到follow(B)中即i
353. FOLLOW[i] = FOLLOW.get(i) + FIRST.get(temp1).replace("ε", "")
354. #A->aBβ（但是β可以推出空串，即β的first集中有空）
355. if ('ε' in FIRST.get(temp1)):
356. FOLLOW[i] = FOLLOW.get(i) + FOLLOW.get(part\_begin)
357. temp1 = i
358. # 若终结符在末端
359. else:
360. temp1 = temp[0]
361. for i in temp[1:]:
362. if i in vt :
363. temp1 = i
364. else:
365. if temp1 in vn:
366. FOLLOW[i] = FOLLOW.get(i) + FIRST.get(temp1)
367. else:
368. FOLLOW[i] = FOLLOW.get(i) + temp1
369. temp1 = i
370. #follow集去重
371. for i, j in FOLLOW.items():
372. temp = ""
373. for word in list(set(j)):
374. temp += word
375. FOLLOW[i] = temp
376. if test == FOLLOW:
377. break
378. return FIRST, FOLLOW
379. #LL（1）分析过程
380. def LL1\_analysis\_solve(self, goal\_str, ans\_table):
381. vt, vn, analysis\_table, stack\_str, ptr = self.vt, self.vn, self.analysis\_table, self.stack\_str, self.ptr
382. vm=vn+vt
383. goal\_str=match\_strings(vm+["ε"],goal\_str)
384. stack\_str=match\_strings(vm+["ε"],stack\_str)
385. lookup\_table=None
386. shuchu=''
387. # o=0
388. while ptr >= 0 and ptr <= len(goal\_str):
389. stack\_top = stack\_str[len(stack\_str) - 1]  # 获取栈顶 $new\_vn[0]
390. goal\_pos = goal\_str[ptr]
391. # print(stack\_str, '  ', stack\_top, '  ', goal\_pos)
392. if (stack\_top not in vt and stack\_top not in vn) or goal\_pos not in vt:  # 非法输入的情况
393. print('输入不合法！')
394. return
395. elif stack\_top == goal\_pos:
396. if stack\_top == '$':  # 栈顶符号=当前输入符号=$
397. print('分析成功！')
398. # shuchu="".join(stack\_top)+"->"+"".join(lookup\_table)
399. ans\_table.add\_row([stack\_str, goal\_str[ptr:len(goal\_str)], '分析成功'])
400. return
401. else:  # 栈顶符号=当前输入符号但是并不都等于$
402. # shuchu="".join(stack\_top)+"->"+"".join(lookup\_table)
403. ans\_table.add\_row([stack\_str, goal\_str[ptr:len(goal\_str)], ''])
404. stack\_str = stack\_str[0:len(stack\_str) - 1]#弹栈
405. ptr += 1#指针前移
406. continue
407. lookup\_table = None
408. #查找栈顶符号在分析表中索引
409. if stack\_top in vn:
410. stack\_top\_index = vn.index(stack\_top)
411. elif stack\_top in vt:
412. stack\_top\_index = vt.index(stack\_top)
413. else:
414. print(f"未知的栈顶符号: {stack\_top}")
415. return
416. #查找输入符号在分析表中索引
417. if goal\_pos in vn:
418. goal\_pos\_index = vn.index(goal\_pos)
419. elif goal\_pos in vt:
420. goal\_pos\_index = vt.index(goal\_pos)
421. else:
422. print(f"未知的输入符号: {goal\_pos}")
423. return
424. #查找对应产生式
425. lookup\_table = analysis\_table[stack\_top\_index + 1][goal\_pos\_index + 1]
426. # print(stack\_top, ' ', goal\_pos, ' ', lookup\_table)
427. if lookup\_table is not None:
428. #弹栈，结束
429. if lookup\_table == 'ε':
430. hh="".join(stack\_top)+"->"+"".join(lookup\_table)
431. ans\_table.add\_row([stack\_str, goal\_str[ptr:len(goal\_str)], hh])
432. stack\_str = stack\_str[0:len(stack\_str) - 1]
433. # o+=1
434. continue
435. else:
436. #存在对应产生式，反向压栈
437. shuchu="".join(stack\_top)+"->"+"".join(lookup\_table)
438. ans\_table.add\_row([stack\_str, goal\_str[ptr:len(goal\_str)], shuchu])
439. # o+=1
440. stack\_str = stack\_str[0:len(stack\_str) - 1]  # 弹栈
441. stack\_str += match\_strings(vm+["ε"],reverse\_by\_set(vm+["ε"],lookup\_table))
442. continue
443. else:
444. print('分析失败！')
445. return
446. if \_\_name\_\_ == '\_\_main\_\_':
447. with open('test.txt', 'r', encoding='utf-8') as file:
448. ll1\_analysis = LL1\_analysis(Gram=file.read())
449. while True:
450. goal\_str = input('请输入字符串(q退出):') + '$'
451. if goal\_str == 'q$':
452. break
453. result\_table = PrettyTable(['栈', '输入串', '寻找产生式'])
454. ll1\_analysis.LL1\_analysis\_solve(goal\_str=goal\_str, ans\_table=result\_table)
455. print(result\_table)

测试文件（test.txt）：

D->TV

T->i|f

V->d,V|d