



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

一、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。

二、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。

(2) 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。

(3) 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。

(4) 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）。

三、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 $A \rightarrow A\alpha \mid \beta$ ，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：

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \varepsilon$

```
67     #消除直接左递归
68     def clean_direct_recursion(self, ch_i, grammer, new_vn):
69         ch = ch_i + ""
70         flag = 0
71         rules = copy.deepcopy(grammer)
72         for key in grammer.keys():
73             for item_i in grammer[key]:
74                 if ch_i == key and ch_i == item_i[0]:
75                     flag = 1
76                     #添加新非终结符
77                     if ch not in rules.keys():
78                         rules[ch] = []
79                         rules[ch].append(item_i[1:] + ch)
80                         rules[key].remove(item_i)
81         #不存在左递归，直接返回
82         if flag == 0:
83             return rules, new_vn
84         for key in grammer.keys():
85             for item_i in grammer[key]:
86                 if ch_i == key and ch_i != item_i[0]:
87                     if ch not in rules.keys():
88                         rules[ch] = []
89                         rules[ch].append(item_i + ch)
90                         rules[key].remove(item_i)
91         #添加新非终结符空串产生式
92         rules[ch].append('ε')
93         new_vn.append(ch)
94         # print(rules, ' ', new_vn)
95         return rules, new_vn
```

2. 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 $A \rightarrow A\alpha$ ，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。

```

45     #消除间接左递归
46     def remove_left_recursion(self):
47         new_grammar = copy.deepcopy(self.grammar)
48         new_vn = copy.deepcopy(self.vn)
49         #两层循环暴露直接左递归
50         for i in range(len(self.vn)):
51             for j in range(0, i):
52                 new_grammar = self.convert(self.vn[i], self.vn[j], new_grammar)
53                 new_grammar, new_vn = self.clean_direct_recursion(self.vn[i], new_grammar, new_vn)
54         return new_grammar, new_vn

```

3. 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。

```

56     #产生式右部非终结符转终结符
57     def convert(self, ch_i, ch_j, grammar):
58         rules = copy.deepcopy(grammar)
59         for key in grammar.keys():
60             for item_i in grammar[key]:
61                 if ch_i == key and ch_j == item_i[0]:
62                     rules[key].remove(item_i)
63                     for item_j in grammar[ch_j]:
64                         rules[key].append(item_j + item_i[1:])
65         return rules

```

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 `grammar_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 $A \rightarrow Aa \mid b$ ），会引入新的非终结符（如 A' ）并将空串产生式（ $A' \rightarrow \epsilon$ ）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。

```
103     # 获取最长公共前缀
104     def LCP(self, i, j, rules):
105         strs = [rules[i], rules[j]]
106         res = ''
107         for each in zip(*strs):
108             if len(set(each)) == 1:
109                 res += each[0]
110             else:
111                 return res
112         return res
```

2. 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。

```
114     #获取公共前缀索引
115     def get_lcp_res(self, key):
116         res = {}
117         rules = self.grammar[key]
118         for i in range(len(rules)):
119             for j in range(i+1, len(rules)):
120                 temp = self.LCP(i,j,rules)
121                 if temp not in res.keys():
122                     res[temp] = set()
123                     res[temp].add(i)
124                     res[temp].add(j)
125         #去空串前缀
126         if '' in res.keys():
127             res.pop('')
128         return res
```

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。

```
130     def remove_common_factor(self):
131         keys = list(self.grammar.keys())
132         for key in keys:
133             while (True):
134                 res = self.get_lcp_res(key)
135                 #直到没有公共前缀
136                 if (res == {}):
137                     break
138                 dels = [] #存即将删除的串
139                 lcp = list(res.keys())[0] #每次取一个公共前缀
140                 ch = key+""
141                 if ch not in self.vn:
142                     self.vn.append(ch)
143                 # 遍历要消除公共因子的元素下标
144                 for i in res[lcp]:
145                     string = self.grammar[key][i]
146                     dels.append(string)
147                     string = string.lstrip(lcp)
148                     if string == '':
149                         string += '\epsilon'
150                     if ch not in self.grammar.keys():
151                         self.grammar[ch] = []
152                     #加入新产生式
153                     self.grammar[ch].append(string)
154                     #删去原来产生式
155                     for string in dels:
156                         self.grammar[key].remove(string)
157                     self.grammar[key].append(lcp + ch)
158             return self.grammar, self.vn
```

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 $A \rightarrow \varepsilon$, 则将 ε 加入 $FIRST(A)$ 。

如果 $A \rightarrow B_1B_2...B_n$, 并且 B_1 的 $FIRST$ 集包含终结符, 则将 B_1 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 $A \rightarrow \alpha B \beta$, 则将 $FIRST(\beta)$ (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:

○ 对于每个非终结符 A , 根据其产生式进行计算:

- 如果产生式为 $A \rightarrow a\alpha$, 则将 a 加入 $FIRST(A)$ 。
- 如果产生式为 $A \rightarrow B_1B_2...B_n$, 则依次计算 B_1 的 $FIRST$ 集。若 B_1 的 $FIRST$ 集中包含空串 ε , 则继续计算, B_2 以此类推。

```
322     for str in grammars: # 求first集 -->直接推出第一个字符为终结符
323         part_begin = str.split(">")[0]
324         part_end = str.split(">")[1]
325         if part_end[0]=='ε':
326             FIRST[part_begin] = FIRST.get(part_begin) + part_end[0]
327         elif (match_strings(vm,part_end)[0] in vt) :
328             FIRST[part_begin] = FIRST.get(part_begin) + match_strings(vm,part_end)[0]
```

```

330         for i in range(len(vn)):
331             while True:
332                 test = FIRST
333                 for str in grammars: # 求first集 二 A->B 把B的first集加到A的first集中
334                     part_begin = ''
335                     part_end = ''
336                     part_begin += str.split('->')[0]
337                     part_end += str.split('->')[1]
338                     #B的first集加到A的first集中
339                     if part_end[0]!='ε' :
340                         if match_strings(vm,part_end)[0] in vn:
341                             FIRST[part_begin] = FIRST.get(part_begin) + FIRST.get(match_strings(vm,part_end)[0])

```

3. 去重：对每个非终结符的 First 集去重，确保没有重复的元素。

```

343         # first集去重
344         for i, j in FIRST.items():
345             temp = ""
346             for word in list(set(j)):
347                 temp += word
348             FIRST[i] = temp

```

4. 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。

```

349         if test == FIRST:
350             break

```

（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 $A \rightarrow \alpha B \beta$ ，且 B 是非终结符，则 $\text{Follow}(B)$ 应该包含 $\text{Follow}(A)$ （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 $A \rightarrow \alpha B$ ，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 $\text{Follow}(B)$ 应该包含 $\text{Follow}(A)$ 。

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 $\text{Follow}(B)$ 应该也包含 $\text{Follow}(A)$ 。

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。

```
356 # S->Ab型
357 for str in grammars:
358     part_begin = str.split(">")[0]
359     part_end = str.split(">")[1]
360     # S->a 直接推出终结符则继续
361     if (len(match_strings(vm, part_end)) == 1 and (part_end in vt)):
362         continue
363     # 否则
364     else:
365         temp = match_strings(vm+["ε"], reverse_by_set(vm+["ε"], part_end))
366         # 若非终结符在末端 A->aCB A->aB 若非终结符B在句型的末端则把A加入进去
367         if temp[0] in vn:
368             FOLLOW[temp[0]] = FOLLOW.get(temp[0]) + FOLLOW.get(part_begin)
369             temp1 = temp[0] + B
370             for i in temp[1:]:
371                 # print("1111111111111111")
372                 # print(i)
373                 if i in vt: # A->aB
374                     temp1 = i + a
375                 else:
376                     if temp1 in vn: # A->aCB # i=c 删掉
377                         # 此时temp1是C, CA->aBC, i=B, first(C)-空加入到follow(B)中即i
378                         FOLLOW[i] = FOLLOW.get(i) + FIRST.get(temp1).replace("ε", "")
379                     # A->aBβ (但是β可以推出空串, 即β的first集中有空)
380                     if ('ε' in FIRST.get(temp1)):
381                         FOLLOW[i] = FOLLOW.get(i) + FOLLOW.get(part_begin)
382                     temp1 = i
```

```
383 # 若终结符在末端
384 else:
385     temp1 = temp[0]
386     for i in temp[1:]:
387         if i in vt:
388             temp1 = i
389         else:
390             if temp1 in vn:
391                 FOLLOW[i] = FOLLOW.get(i) + FIRST.get(temp1)
392             else:
393                 FOLLOW[i] = FOLLOW.get(i) + temp1
394     temp1 = i
```

3. 去重：对每个非终结符的 Follow 集去重，确保没有重复的元素。

```
395 # follow集去重
396 for i, j in FOLLOW.items():
397     temp = ""
398     for word in list(set(j)):
399         temp += word
400     FOLLOW[i] = temp
```

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。

```
401         if test == FOLLOW:
402             break
```

四、LL(1)文法的判定

1. 定义：

- 文法为LL(1)的条件是对于每个非终结符A及其产生式 $A \rightarrow \alpha \mid \beta$ ，满足以下条件：
 - $FIRST(\alpha) \cap FIRST(\beta) = \emptyset$
 - 如果 ϵ 在 $FIRST(\alpha)$ 或 $FIRST(\beta)$ 中，则需满足 $FOLLOW(A) \cap FIRST(\beta) = \emptyset$ 或 $FOLLOW(A) \cap FIRST(\alpha) = \emptyset$ 。

2. 算法流程：

- 计算文法的FIRST集和FOLLOW集。
- 对于每个非终结符的所有产生式，检查是否满足LL(1)条件。

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 $A \rightarrow \alpha$ 。

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 。

```

235         if t == 'ε': # 如果是ε, 对应FOLLOW集中的终结符位置填上ε
236             for j in range(len(new_vt)): # 遍历所有的终结符
237                 if new_vt[j] in FOLLOW[new_vn[i]]: # FOLLOW[part_begin]为当前非终结符的FOLLOW集
238                     # 如果分析表该位置为空, 则填入ε
239                     if analysis_table[i + 1][j + 1] is None:
240                         analysis_table[i + 1][j + 1] = '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。

```

226 #分析表
227 analysis_table = [[None] * (1 + len(new_vt)) for row in range(1 + len(new_vn))]
228 analysis_table[0][0] = ' '
229 for i in range(len(new_vt)):
230     analysis_table[0][i + 1] = new_vt[i]
231 for i in range(len(new_vn)):
232     analysis_table[i + 1][0] = new_vn[i]
233 for i in range(len(new_vn)):
234     for t in new_grammar[new_vn[i]]: # 遍历该文法的所有产生式
235         if t == 'ε': # 如果是ε, 对应在FOLLOW集中的终结符位置填上ε
236             for j in range(len(new_vt)): # 遍历所有的终结符
237                 if new_vt[j] in FOLLOW[new_vn[i]]: # FOLLOW[part_begin]为当前非终结符的FOLLOW集
238                     # 如果分析表该位置为空, 则填入ε
239                     if analysis_table[i + 1][j + 1] is None:
240                         analysis_table[i + 1][j + 1] = 'ε'
241             else:
242                 first_found = False # 用于标记是否已经找到有效的FIRST项
243                 for symbol in t: # 遍历产生式右侧的每个符号
244                     if symbol in new_vt: # 如果是终结符
245                         # 将该符号填入对应位置
246                         j = new_vt.index(symbol)
247                         if analysis_table[i + 1][j + 1] is None:
248                             analysis_table[i + 1][j + 1] = t
249                         first_found = True
250                     break # 终结符就直接填入, 并停止检查其他符号
251                 else: # 如果是非终结符
252                     # 使用该非终结符的FIRST集
253                     for first_symbol in FIRST[symbol]:
254                         if first_symbol != 'ε': # 只处理非ε项
255                             j = new_vt.index(first_symbol)
256                             if analysis_table[i + 1][j + 1] is None:
257                                 analysis_table[i + 1][j + 1] = t
258                     # 如果该非终结符的FIRST集包含ε, 需要继续检查后面的符号
259                     if 'ε' in FIRST[symbol]:
260                         continue
261                 else:
262                     first_found = True
263                     break # 如果FIRST集没有包含ε, 停止检查后面的符号
264
265 # 如果右侧符号都能推导出ε, 则检查FOLLOW集并填充
266 if not first_found:
267     for j in range(len(new_vt)):
268         if new_vt[j] in FOLLOW[new_vn[i]]:
269             if analysis_table[i + 1][j + 1] is None:
270                 analysis_table[i + 1][j + 1] = 'ε'

```

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。

```
409     goal_str=match_strings(vm+["ε"],goal_str)
410     stack_str=match_strings(vm+["ε"],stack_str)
411     lookup_table=None
```

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

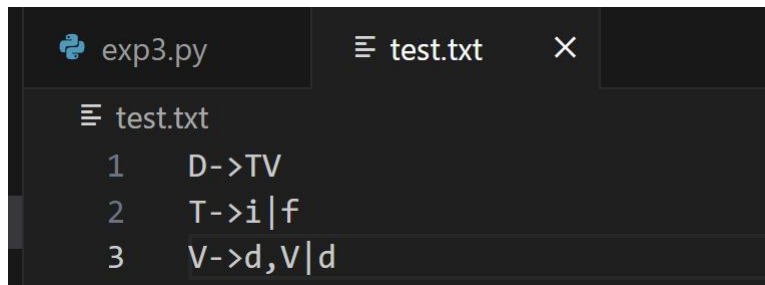
在每一步操作中，分析器都会记录当前栈内容、剩余输入串以及对应

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。

四、 测试结果

测试用例：

A screenshot of a code editor window. The top bar shows two tabs: 'exp3.py' and 'test.txt'. The 'test.txt' tab is active. The editor content shows three lines of text, each preceded by a line number (1, 2, 3). The text represents grammar rules: 'D->TV', 'T->i|f', and 'V->d,V|d'.

```
test.txt
1    D->TV
2    T->i|f
3    V->d,V|d
```

消除左递归：

```
...  <  >  [search]
exp3.py  ×  test.txt
exp3.py > match_strings > [input_str]
1  import re

问题  输出  终端  端口  调试控制台

○ PS D:\gaga\work\编译原理实践\实验三\EXP3
理实践/实验三/EXP3/exp3.py

输入的文法:
+-----+-----+-----+-----+
| 编号 | 左部 | 右部 | 产生式 |
+-----+-----+-----+-----+
| 1    | D    | TV   | D->TV  |
| 2    | T    | i    | T->i   |
| 3    | T    | f    | T->f   |
| 4    | V    | d,V  | V->d,V |
| 5    | V    | d    | V->d   |
+-----+-----+-----+-----+

消除左递归:
+-----+-----+-----+-----+
| 编号 | 左部 | 右部 | 产生式 |
+-----+-----+-----+-----+
| 1    | D    | TV   | D->TV  |
| 2    | T    | i    | T->i   |
| 3    | T    | f    | T->f   |
| 4    | V    | d,V  | V->d,V |
| 5    | V    | d    | V->d   |
+-----+-----+-----+-----+
```

提取左公因子、计算 first 集和 follow 集、预测分析表、预测分析器：

提取公因子：

编号	左部	右部	产生式
1	D	TV	D->TV
2	T	i	T->i
3	T	f	T->f
4	V	dV'	V->dV'
5	V'	,V	V'->,V
6	V'	ϵ	V'-> ϵ

文法的FIRST集：

$\text{FIRST}(D) = \{f, i\}$
 $\text{FIRST}(T) = \{f, i\}$
 $\text{FIRST}(V) = \{d\}$
 $\text{FIRST}(V') = \{\epsilon, ,\}$

文法的FOLLOW集：

$\text{FOLLOW}(D) = \{\$ \}$
 $\text{FOLLOW}(T) = \{d\}$
 $\text{FOLLOW}(V) = \{\$ \}$
 $\text{FOLLOW}(V') = \{\$ \}$

该文法是LL(1)文法

```

...  ← →  EXP3
exp3.py  test.txt
exp3.py > match_strings > input_str
1  import re

问题  输出  终端  端口  调试控制台

预测分析表：
+-----+-----+-----+-----+-----+
| 非终结符 | i | f | d | , | $ |
+-----+-----+-----+-----+
| D | TV | TV | None | None | None |
| T | i | f | None | None | None |
| V | None | None | dV' | None | None |
| V' | None | None | None | ,V |  $\epsilon$  |
+-----+-----+-----+-----+

请输入字符串(q退出):id,d,d
分析成功!

+-----+-----+-----+
| 栈 | 输入串 | 寻找产生式 |
+-----+-----+-----+
| ['$', 'D'] | ['i', 'd', ',', 'd', ',', 'd', '$'] | D->TV |
| ['$', 'V', 'T'] | ['i', 'd', ',', 'd', ',', 'd', '$'] | T->i |
| ['$', 'V', 'i'] | ['i', 'd', ',', 'd', ',', 'd', '$'] |  |
| ['$', 'V'] | ['d', ',', 'd', ',', 'd', '$'] | V->dV' |
| ['$', "V'", 'd'] | ['d', ',', 'd', ',', 'd', '$'] |  |
| ['$', "V'"] | ['d', ',', 'd', ',', 'd', '$'] | V'->,V |
| ['$', 'V', ','] | ['d', ',', 'd', ',', 'd', '$'] |  |
| ['$', 'V'] | ['d', ',', 'd', ',', 'd', '$'] | V->dV' |
| ['$', "V'", 'd'] | ['d', ',', 'd', ',', 'd', '$'] |  |
| ['$', "V'"] | ['d', ',', 'd', ',', 'd', '$'] | V'->,V |
| ['$', 'V', ','] | ['d', ',', 'd', ',', 'd', '$'] |  |
| ['$', 'V'] | ['d', ',', 'd', ',', 'd', '$'] | V->dV' |
| ['$', "V'", 'd'] | ['d', ',', 'd', ',', 'd', '$'] |  |
| ['$', "V'"] | ['d', ',', 'd', ',', 'd', '$'] | V'->,V |
| ['$', "V'"] | ['d', ',', 'd', ',', 'd', '$'] | V'-> $\epsilon$  |
| ['$'] | ['$'] | 分析成功 |
+-----+-----+-----+

请输入字符串(q退出):q
PS D:\gaga\work\编译原理实践\实验三\EXP3>

```

五、源代码

```
1. import re
2. import copy
3. from prettytable import PrettyTable
4.
5. #按终结符和非终结符遍历
6. def match_strings(A, input_str):
7.     # 优先匹配最长(A'和A识别成A')
8.     A = sorted(A, key=lambda x: len(x), reverse=True)
9.     pattern = '|'.join(map(re.escape, A))
10.    matches = re.findall(pattern, input_str)
11.    return matches
12.
13. #按终结符非终结符整体字符串倒序
14. def reverse_by_set(A, input_str):
15.     result = []
16.     i = len(input_str)
17.     while i > 0:
18.         for word in reversed(A):
19.             word_len = len(word)
```

```

20.             if i >= word_len and input_str[i
- word_len:i] == word:
21.                 result.append(word)
22.                 i -= word_len
23.                 break
24.         else:
25.             i -= 1
26.     return ''.join(result)
27.
28. #可视化输出
29. class draw_grammer:
30.     def draw_grammer(grammer, vn, descrpition)
:
31.         print_content = PrettyTable(['编号
', '左部', '右部', '产生式'])
32.         idx = 1
33.         for i in vn:
34.             for j in grammer[i]:
35.                 print_content.add_row([idx,
i, j, i + '->' + j])
36.                 idx += 1

```



```

37.         print('\n\n'+description+':\n', prin
           t_content)
38.
39. #消除左递归
40. class EliminateLeftRecursion:
41.     def __init__(self, grammer, vn):
42.         self.grammer = grammer
43.         self.vn = vn
44.
45.     #消除间接左递归
46.     def remove_left_recursion(self):
47.         new_grammer = copy.deepcopy(self.grammer)
48.         new_vn = copy.deepcopy(self.vn)
49.         #两层循环暴露直接左递归
50.         for i in range(len(self.vn)):
51.             for j in range(0, i):
52.                 new_grammer = self.convert(self.vn[i], self.vn[j], new_grammer)
53.                 new_grammer, new_vn = self.clean_direct_recursion(self.vn[i], new_grammer, new_vn)

```

```

54.         return new_grammer, new_vn
55.
56.     #产生式右部非终结符转终结符
57.     def convert(self, ch_i, ch_j, grammer):
58.         rules = copy.deepcopy(grammer)
59.         for key in grammer.keys():
60.             for item_i in grammer[key]:
61.                 if ch_i == key and ch_j == i
tem_i[0]:
62.                     rules[key].remove(item_i
)
63.                     for item_j in grammer[ch
_j]:
64.                         rules[key].append(it
em_j + item_i[1:])
65.         return rules
66.
67.     #消除直接左递归
68.     def clean_direct_recursion(self, ch_i, g
rammer, new_vn):
69.         ch = ch_i + "'"
70.         flag = 0

```

```

71.         rules = copy.deepcopy(grammer)
72.         for key in grammer.keys():
73.             for item_i in grammer[key]:
74.                 if ch_i == key and ch_i == i
tem_i[0]:
75.                     flag = 1
76.                     #添加新非终结符
77.                     if ch not in rules.keys()
:
78.                         rules[ch] = []
79.                         rules[ch].append(item_i[
1:] + ch)
80.                         rules[key].remove(item_i
)
81.                     #不存在左递归, 直接返回
82.                     if flag == 0:
83.                         return rules, new_vn
84.                     for key in grammer.keys():
85.                         for item_i in grammer[key]:
86.                             if ch_i == key and ch_i != i
tem_i[0]:

```

```

87.             if ch not in rules.keys()
:
88.                 rules[ch] = []
89.                 rules[ch_i].append(item_
i + ch)
90.                 rules[key].remove(item_i
)
91.             #添加新非终结符空串产生式
92.             rules[ch].append('ε')
93.             new_vn.append(ch)
94.             # print(rules, ' ', new_vn)
95.             return rules, new_vn
96.
97. #提取左公因子
98. class ExtractCommonFactors:
99.     def __init__(self, grammer, vn):
100.         self.grammer = grammer
101.         self.vn = vn
102.
103.         # 获取最长公共前缀
104.         def LCP(self, i, j, rules):
105.             strs = [rules[i], rules[j]]

```

```

106.         res = ''
107.         for each in zip(*strs):
108.             if len(set(each)) == 1:
109.                 res += each[0]
110.             else:
111.                 return res
112.         return res
113.
114.     #获取公共前缀索引
115.     def get_lcp_res(self, key):
116.         res = {}
117.         rules = self.grammer[key]
118.         for i in range(len(rules)):
119.             for j in range(i+1, len(rules)
120.                             ):
121.                 temp = self.LCP(i,j,rules
122.                                 )
123.                 if temp not in res.keys():
124.                     res[temp] = set()
125.                     res[temp].add(i)
126.                     res[temp].add(j)

```

```

125.         #去空串前缀
126.         if '' in res.keys():
127.             res.pop('')
128.         return res
129.
130.     def remove_common_factor(self):
131.         keys = list(self.grammer.keys())
132.         for key in keys:
133.             while (True):
134.                 res = self.get_lcp_res(key
135. y)
136.                 #直到没有公共前缀
137.                 if (res == {}):
138.                     break
139.                 dels = [] #存即将删除的串
140.                 lcp = list(res.keys())[0]
141.                 #每次取一个公共前缀
142.                 ch = key+""
143.                 if ch not in self.vn:
144.                     self.vn.append(ch)
145.                 # 遍历要消除公共因子的元素下标
146.                 for i in res[lcp]:

```

```

145.                string = self.grammer
                    [key][i]
146.                dels.append(string)
147.                string = string.lstri
                    p(lcp)
148.                if string == '':
149.                    string += 'ε'
150.                if ch not in self.gra
                    mmer.keys():
151.                    self.grammer[ch]
                        = []
152.                #加入新产生式
153.                self.grammer[ch].appe
                    nd(string)
154.                #删去原来产生式
155.                for string in dels:
156.                    self.grammer[key].rem
                        ove(string)
157.                self.grammer[key].append(
                    lcp + ch)
158.                return self.grammer, self.vn
159.

```



```

160. #文法分析
161. class LL1_analysis:
162.     def __init__(self, Gram):
163.         #终结符 非终结符 分析表元素 $+开始符号
164.         self.vt, self.vn, self.analysis_t
            able, self.stack_str = self.init_all_(g=Gram
            )
165.         self.ptr = 0
166.
167.     def init_all_(self, g):
168.         #读取文法
169.         grammer_list = {} #非终结符: 产生式
170.         vn_list = [] #非终结符
171.         for line in re.split('\n', g):
172.             # 去空格
173.             line = "".join([i for i in li
                ne if i not in [' ', ' ']])
174.             if '->' in line:
175.                 if line.split('->')[0] not
                    in vn_list:
176.                     vn_list.append(line.s
                        plit('->')[0])

```

```

177.                 for i in line.split('->')
                        [1].split('|'):
178.                 if grammer_list.get(l
                        ine.split('->')[0]) is None:
179.                 grammer_list[line.
                        split('->')[0]] = []
180.                 grammer_list[line.
                        split('->')[0]].append(i)
181.                 else:
182.                 grammer_list[line.
                        split('->')[0]].append(i)
183.                 draw_grammer.draw_grammer(grammer
                        =grammer_list, vn=vn_list, descrpition='输入
                        的文法')
184.
185.                 #消除左递归
186.                 # print('产生式: ', grammer_list)
187.                 # print('非终结符: ', vn_list)
188.                 eliminate_left_recursion = Elimin
                        ateLeftRecursion(grammer=grammer_list, vn=vn
                        _list)

```

```

189.         new_grammer, new_vn = eliminate_left_recursion.remove_left_recursion()

190.         draw_grammer.draw_grammer(grammer
        =new_grammer, vn=new_vn, descrpition='消除左
        递归')

191.

192.         #提取左公因子

193.         extractcommonfactors = ExtractCommonFactors(grammer=new_grammer, vn=new_vn)

194.         new_grammer, new_vn = extractcommonfactors.remove_common_factor()

195.         draw_grammer.draw_grammer(grammer
        =new_grammer, vn=new_vn, descrpition='提取公
        因子')

196.

197.         only_grammer = []

198.         new_vt = []

199.         for i in new_vn:

200.             for j in new_grammer[i]:

201.                 only_grammer.append(i + '
        ->' + j)

202.

```

```

203.                 for t in j:  # 获取当前的所
                        有的终结符
204.                 if t not in new_vt an
                        d t not in new_vn and t!="ε" and t!="'":
205.                     # print(t)
206.                     new_vt.append(t)
207.                 new_vt.append('$')
208.                 # print('\n\n 消除文法左递归的文法的非
                        终结符:', new_vn,
209.                 #         '\n\n 消除文法左递归的文法的终
                        结符:', new_vt)
210.
211.                 #FIRST 集和 FOLLOW 集
212.                 FIRST, FOLLOW = self.get_first_and
                        d_follow_set(grammars=only_grammer, vn=new_vn,
                        vt=new_vt)
213.                 print('\n\n 文法的 FIRST 集:')
214.                 for i, j in FIRST.items():
215.                     str = j[0]
216.                     for temp in j[1:]:
217.                         str = str + ',' + temp

```

```

218.             print("FIRST(" + i + ") " + "
= {" + str + "}")
219.             print('\n\n 文法的 FOLLOW 集:')
220.             for i, j in FOLLOW.items():
221.                 str = j[0]
222.                 for temp in j[1:]:
223.                     str = str + ',' + temp
224.             print("FOLLOW(" + i + ") " + "
= {" + str + "}")
225.
226.             #分析表
227.             analysis_table = [[None] * (1 + l
en(new_vt)) for row in range(1 + len(new_vn))
]
228.             analysis_table[0][0] = ' '
229.             for i in range(len(new_vt)):
230.                 analysis_table[0][i + 1] = ne
w_vt[i]
231.             for i in range(len(new_vn)):
232.                 analysis_table[i + 1][0] = ne
w_vn[i]
233.             for i in range(len(new_vn)):

```

```

234.         for t in new_grammar[new_vn[i]
]: # 遍历该文法的所有产生式
235.         if t == 'ε': # 如果是ε，对
           应在 FOLLOW 集中的终结符位置填上ε
236.         for j in range(len(new_vt)): # 遍历所有的终结符
237.         if new_vt[j] in FOLLOW[new_vn[i]]: # FOLLOW[part_begin]为当前
           非终结符的 FOLLOW 集
238.         # 如果分析表该位置为空，则填入ε
239.         if analysis_table[i + 1][j + 1] is None:
240.             analysis_table[i + 1][j + 1] = 'ε'
241.         else:
242.             first_found = False
           # 用于标记是否已经找到有效的 FIRST 项
243.         for symbol in t: # 遍历产生式右侧的每个符号
244.         if symbol in new_vt: # 如果是终结符

```

```

245.                                     # 将该符号填入对
    应位置
246.                                     j = new_vt.in
    dex(symbol)
247.                                     if analysis_t
    able[i + 1][j + 1] is None:
248.                                     analysis_
    table[i + 1][j + 1] = t
249.                                     first_found =
    True
250.                                     break # 终结
    符就直接填入，并停止检查其他符号
251.                                     else: # 如果是非终
    结符
252.                                     # 使用该非终结符
    的 FIRST 集
253.                                     for first_sym
    bol in FIRST[symbol]:
254.                                     if first_
    symbol != ' $\epsilon$ ': # 只处理非 $\epsilon$ 项
255.                                     j = n
    ew_vt.index(first_symbol)

```



```

256.                                     if an
    analysis_table[i + 1][j + 1] is None:
257.                                     a
    analysis_table[i + 1][j + 1] = t
258.                                     # 如果该非终结符
    的 FIRST 集包含  $\epsilon$ , 需要继续检查后面的符号
259.                                     if ' $\epsilon$ ' in FIR
    ST[symbol]:
260.                                     continue
261.                                     else:
262.                                     first_fou
    nd = True
263.                                     break #
    如果 FIRST 集没有包含  $\epsilon$ , 停止检查后面的符号
264.
265.                                     # 如果右侧符号都能推导出  $\epsilon$ ,
    则检查 FOLLOW 集并填充
266.                                     if not first_found:
267.                                     for j in range(1e
    n(new_vt)):
268.                                     if new_vt[j]
    in FOLLOW[new_vn[i]]:

```

```

269.                                     if analys
    is_table[i + 1][j + 1] is None:
270.                                     analy
    sis_table[i + 1][j + 1] = 'ε'
271.        #判断是否为 LL (1) 文法
272.        is_ll1 = True
273.        for i in range(1, len(new_vn) + 1)
            :
274.            for j in range(1, len(new_vt)
                + 1):
275.                if analysis_table[i][j] i
                    s not None: # 如果当前位置有值 检查是否冲突
276.                    for k in range(i + 1,
                        len(new_vn) + 1): # 对比同一非终结符的其他产生
                        式
277.                        if analysis_table
                            [k][j] == analysis_table[i][j]:
278.                            is_ll1 = Fals
                                e
279.                            break
280.                    if not is_ll1:
281.                        break

```

```

282.             if not is_ll1:
283.                 break
284.
285.             if is_ll1:
286.                 print("\n\n 该文法是 LL(1) 文法")
287.             else:
288.                 print("\n\n 该文法不是 LL(1) 文法
289.                 ")
290.             #输出分析表
291.             pretty_table_title = ['非终结符']
292.             for i in new_vt:
293.                 pretty_table_title.append(i)
294.             analysis_pretty_table = PrettyTable(
                pretty_table_title)
295.             for i in range(len(analysis_table)
                - 1):
296.                 analysis_pretty_table.add_row
                (analysis_table[i + 1])
297.             print('\n\n 预测分析
                表:\n', analysis_pretty_table)
298.

```

```

299.         #返回预处理结构
300.         # print("new_vn:",new_vn[0])
301.         return new_vt, new_vn, analysis_t
           able, '$' + new_vn[0]
302.
303.     def get_first_and_follow_set(self,grammars,vn,vt):
304.         FIRST = {}
305.         FOLLOW = {}
306.         index=0
307.         for str in grammars: # 初始化 first、
           follow 集
308.             # print(str)
309.             part_begin = str.split("->")[
               0]
310.             part_end = str.split("->")[1]
311.             FIRST[part_begin] = ""
312.             if index==0:
313.                 FOLLOW[part_begin]="$"
314.             else:
315.                 FOLLOW[part_begin] = ""
316.             index+=1

```

```

317.                # print(part_begin,FOLLOW[par
    t_begin])
318.
319.                #first 集
320.                vm=vt+vn
321.                # print(match_strings(vm,grammars
    [1].split(">") [1]))
322.                for str in grammars: # 求 first
    集 一 ->直接推出第一个字符为终结符
323.                part_begin = str.split(">") [
    0]
324.                part_end = str.split(">") [1]
325.                if part_end[0]=='ε':
326.                    FIRST[part_begin] = FIRST.
    get(part_begin) + part_end[0]
327.                elif (match_strings(vm,part_e
    nd) [0] in vt) :
328.                    FIRST[part_begin] = FIRST.
    get(part_begin) + match_strings(vm,part_end)
    [0]
329.
330.                for i in range(len(vn)) :

```

```

331.         while True:
332.             test = FIRST
333.             for str in grammars: # 求
                first 集 二 A->B 把 B 的 first 集 加到 A 的 first 集
                中
334.                 part_begin = ''
335.                 part_end = ''
336.                 part_begin += str.split
                    it('-',>')[0]
337.                 part_end += str.split
                    ('-',>')[1]
338.                 #B 的 first 集 加到 A 的
                    first 集中
339.                 if part_end[0]!='ε' :
340.                     if match_strings(
                        vm,part_end)[0] in vn:
341.                         FIRST[part_be
                            gin] = FIRST.get(part_begin) + FIRST.get(mat
                                ch_strings(vm,part_end)[0])
342.
343.                 # first 集 去重

```

```

344.         for i, j in FIRST.items()
:
345.             temp = ""
346.             for word in list(set(
j)):
347.                 temp += word
348.                 FIRST[i] = temp
349.             if test == FIRST:
350.                 break
351.
352.         #follow 集
353.         for i in range(len(vn)):
354.             while True:
355.                 test = FOLLOW
356.                 # S->Ab 型
357.                 for str in grammars:
358.                     part_begin = str.split(
t("<->")[0]
359.                     part_end = str.split(
"<->")[1]
360.                     #S->a 直接推出终结符则继

```

续

```

361.             if (len(match_strings
    (vm,part_end)) == 1 and (part_end in vt)):
362.                 continue
363.                 #否则
364.                 else:
365.                     temp = match_strings
    (vm+["ε"],reverse_by_set(vm+["ε"],part_end))
366.                     # 若非终结符在末端
    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
367.                     if temp[0] in vn:
368.                         FOLLOW[temp[0]]
    = FOLLOW.get(temp[0]) + FOLLOW.get(part_begin)
369.                         temp1 = temp[
    0] #B
370.                         for i in temp
    [1:]:
371.                             # print("
    1111111111111111")

```



```

372.                                     # print(i
    )
373.                                     if i in v
    t:#A->aB
374.                                     temp1
    = i#a
375.                                     else:
376.                                     if te
    mp1 in vn:#A->aCB #i=c 删掉
377.                                     #
    此时 temp1 是 C, CA->aBC, i=B, first(C)-空加入到
    follow(B) 中即 i
378.                                     F
    OLLOW[i] = FOLLOW.get(i) + FIRST.get(temp1).
    replace("ε", "")
379.                                     #A->a
    Bβ (但是β可以推出空串, 即β的 first 集中有空)
380.                                     if ('
    ε' in FIRST.get(temp1)):
381.                                     F
    OLLOW[i] = FOLLOW.get(i) + FOLLOW.get(part_b
    egin)

```

```

382.                                     temp1
    = i
383.                                     # 若终结符在末端
384.                                     else:
385.                                     temp1 = temp[
    0]
386.                                     for i in temp
    [1:]:
387.                                     if i in v
    t :
388.                                     temp1
    = i
389.                                     else:
390.                                     if te
    mp1 in vn:
391.                                     F
    FOLLOW[i] = FOLLOW.get(i) + FIRST.get(temp1)
392.                                     else:
393.                                     F
    FOLLOW[i] = FOLLOW.get(i) + temp1
394.                                     temp1
    = i

```

```

395.             #follow 集去重
396.             for i, j in FOLLOW.items():
397.                 temp = ""
398.                 for word in list(set(
j)):
399.                     temp += word
400.                     FOLLOW[i] = temp
401.                     if test == FOLLOW:
402.                         break
403.             return FIRST, FOLLOW
404.
405.     #LL (1) 分析过程
406.     def LL1_analysis_solve(self, goal_str,
ans_table):
407.         vt, vn, analysis_table, stack_str,
ptr = self.vt, self.vn, self.analysis_table,
self.stack_str, self.ptr
408.         vm=vn+vt
409.         goal_str=match_strings(vm+["ε"],g
oal_str)

```

```

410.         stack_str=match_strings(vn+["ε"],
        stack_str)

411.         lookup_table=None

412.         shuchu=''

413.         # o=0

414.         while ptr >= 0 and ptr <= len(goal_str):

415.             stack_top = stack_str[len(stack_str) - 1] # 获取栈顶 $new_vn[0]

416.             goal_pos = goal_str[ptr]

417.             # print(stack_str, ' ', stack_top, ' ', goal_pos)

418.             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

419.                 print('输入不合法! ')

420.                 return

421.             elif stack_top == goal_pos:

422.                 if stack_top == '$': # 栈顶符号=当前输入符号=$

423.                     print('分析成功! ')

```

```

424.                # shuchu="".join(stack_top) + "->" + "".join(lookup_table)

425.                ans_table.add_row([stack_str, goal_str[ptr:len(goal_str)], '分析成功'])

426.                return

427.            else:    # 栈顶符号=当前输入符号但是并不都等于$

428.                # shuchu="".join(stack_top) + "->" + "".join(lookup_table)

429.                ans_table.add_row([stack_str, goal_str[ptr:len(goal_str)], ''])

430.                stack_str = stack_str[0:len(stack_str) - 1] #弹栈

431.                ptr += 1 #指针前移

432.                continue

433.

434.                lookup_table = None

435.                #查找栈顶符号在分析表中索引

436.                if stack_top in vn:

437.                    stack_top_index = vn.index(stack_top)

```

```

438.             elif stack_top in vt:
439.                 stack_top_index = vt.index
440.                 x(stack_top)
441.             else:
442.                 print(f"未知的栈顶符
443.                     号: {stack_top}")
444.                 return
445.             #查找输入符号在分析表中索引
446.             if goal_pos in vn:
447.                 goal_pos_index = vn.index
448.                 (goal_pos)
449.             elif goal_pos in vt:
450.                 goal_pos_index = vt.index
451.                 (goal_pos)
452.             else:
453.                 print(f"未知的输入符
454.                     号: {goal_pos}")
455.                 return
456.             #查找对应产生式

```

```

454.             lookup_table = analysis_table
                [stack_top_index + 1][goal_pos_index + 1]
455.             # print(stack_top, ' ', goal_
                pos, ' ', lookup_table)
456.             if lookup_table is not None:
457.                 #弹栈, 结束
458.                 if lookup_table == 'ε':
459.                     hh="".join(stack_top)
                        + "->" + "".join(lookup_table)
460.                     ans_table.add_row([st
                ack_str, goal_str[ptr:len(goal_str)], hh])
461.                     stack_str = stack_str
                        [0:len(stack_str) - 1]
462.                     # o+=1
463.                     continue
464.                 else:
465.                     #存在对应产生式, 反向压栈
466.                     shuchu="".join(stack_
                top) + "->" + "".join(lookup_table)
467.                     ans_table.add_row([st
                ack_str, goal_str[ptr:len(goal_str)], shuchu]
                )

```

```

468.                                # o+=1
469.                                stack_str = stack_str
                                [0:len(stack_str) - 1] # 弹栈
470.                                stack_str += match_st
                                rings(vm+["ε"],reverse_by_set(vm+["ε"],looku
                                p_table))
471.                                continue
472.                                else:
473.                                print('分析失败! ')
474.                                return
475.
476. if __name__ == '__main__':
477.     with open('test.txt', 'r', encoding='
utf-8') as file:
478.         ll1_analysis = LL1_analysis(Gram=
file.read())
479.         while True:
480.             goal_str = input('请输入字符串(q退
出):') + '$'
481.             if goal_str == 'q$':
482.                 break

```



```
483.         result_table = PrettyTable(['栈', '输入串', '寻找产生式'])
484.         l11_analysis.LL1_analysis_solve(goal_str=goal_str, ans_table=result_table)
485.         print(result_table)
```

测试文件（test.txt）：

D→TV

T→i|f

V→d, V|d