杭州电子科技大学

《编译原理课程实践》

实验报告

题　　目：语法分析相关算法的实现

学　　院：计算机

专　　业：计算机科学与技术

班　　级：23052312

学　　号：22050233

姓　　名：郑方昊

完成日期：2024.12.29

一、 实验目的

1.理解上下文无关文法中左递归的概念及其对语法分析的影响。
2.掌握消去上下文无关文法中直接和间接左递归的算法。
3.培养运用编程语言实现文法变换的能力。
4.理解上下文无关文法中的左公共因子的概念及其对语法分析的影响。
5.掌握从上下文无关文法中提取左公共因子的算法，形成无二义性的语法结构。
6.熟练运用数据结构（如 Trie 树）处理和优化文法。
7.理解上下文无关文法中 FIRST 集和 FOLLOW 集的概念及其在语法分析中的重要性。
8.掌握计算文法中 FIRST 集和 FOLLOW 集的算法及其实现。
9.培养分析和解决文法问题的能力。
10.理解 LL(1)文法的概念及其在语法分析中的应用。
11.掌握判定文法是否为 LL(1)的方法。
12.学习设计和实现 LL(1)预测分析器的过程。
13.培养运用编程语言实现自顶向下语法分析的能力。

二、 实验内容与实验要求

2.1 上下文无关文法的左递归消除

2.1.1 实验内容：

实现消去上下文无关文法中所有左递归的算法。具体步骤包括：

1. 对非终结符集合进行排序。
2. 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。
3. 消去直接左递归。

2.1.2 实验要求：

1. 输入：一个上下文无关文法，包括非终结符、终结符和产生式。【为了跟后续实验贯穿，建议仔细设计良好的数据结构来表示文法】
2. 输出：消去左递归后的文法。
3. 算法：应处理直接和间接左递归，确保输出文法与输入文法等价。
4. 测试：提供测试用例，验证算法的正确性。
5. 文档：编写文档，说明如何使用你的程序，包括输入格式和输出解释。

2.2 文法左公共因子的提取

2.2.1 实验内容：

实现从上下文无关文法中提取左公共因子的算法，具体步骤包括：

1. 对每个非终结符的候选式，识别最长的公共前缀。
2. 构建字典树（Trie），辅助提取最长公共前缀，将公共前缀提取为新非终结符的候选式。
3. 输出去除左公共因子的等价文法。

2.2.2 实验要求：

1. 输入一个上下文无关文法，包括非终结符、终结符和产生式。
2. 输出提取左公共因子后的文法。

3. 使用适当的数据结构（如 Trie 树）提高提取效率。
4. 确保输出文法无二义性，且与输入文法等价。

2.3 FIRST 集和 FOLLOW 集的求解和实现

2.3.1 实验内容：

实现求解上下文无关文法的 FIRST 集和 FOLLOW 集的算法。具体步骤包括：

1. 输入上下文无关文法。
2. 计算每个非终结符的 FIRST 集。
3. 计算每个非终结符的 FOLLOW 集。

2.3.2 实验要求：

1. 输入一个上下文无关文法，包括非终结符、终结符和产生式。
2. 输出每个非终结符的 FIRST 集和 FOLLOW 集。
3. 算法应考虑文法的各种情况，确保输出结果准确。

2.4 LL（1）文法判定与预测分析器

2.4.1 实验内容：

实现 LL(1)文法的判定算法和预测分析器的设计与实现。具体步骤包括：

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

2.4.2 实验要求：

1. 输入一个上下文无关文法，包括非终结符、终结符和产生式。
2. 在任务 3.1-3.3 基础上来实现判断
3. 输出文法是否为 LL(1)的判断结果。
4. 输出预测分析表。
5. 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

3.1 上下文无关文法的左递归消除

3.1.1 设计方案：

目标：消除文法中的左递归，以便能够进行 LL(1)分析。

步骤：

（1）识别文法规则中存在左递归的非终结符。

（2）对每个左递归的规则进行转换，消除其左递归。

（3）将产生式改写成两部分：

一部分包含递归调用，但不以该非终结符开头。

另一部分用于处理递归部分，确保不会导致无限递归。

3.1.2 算法描述：

（1）给定文法规则：A → Aα | β，其中 A 是非终结符，α 和 β 是终结符和非终结符的串。

（2）创建一个新的非终结符 A’（A 的副本），然后进行如下转换：

将 A → βA’ 作为新的产生式。

将 A’ → αA’ | ε 作为 A’的产生式（即递归部分的消除）。

3.2 文法左公共因子的提取

3.2.1 设计方案：

目标：提取文法规则中的左公共因子，以消除由于相同前缀导致的冲突，便于进行预测分析。

步骤：

（1）对每个产生式，识别出公共前缀部分。

（2）将这些前缀提取出来，并为每个产生式创建新的分支。

（3）将产生式改为带有新非终结符的形式，确保没有共同前缀。

3.2.2 算法描述：

（1）给定文法规则：A → αβ1 | αβ2 | αβ3，其中 α 为公共前缀。

（2）提取公共前缀 α，并构造新的产生式：

A → αA’。

A’ → β1 | β2 | β3。

3.3 FIRST 集和 FOLLOW 集的求解和实现

3.3.1 设计方案：

目标：计算文法中每个非终结符的 FIRST 集和 FOLLOW 集，以便进行 LL(1)分析。

步骤：

（1）FIRST 集：对于每个非终结符，找到能够推导出该符号的终结符集合。

（2）FOLLOW 集：对于每个非终结符，找到可以出现在该符号之后的终结符集合。

反复迭代直到 FIRST 集和 FOLLOW 集不再发生变化。

3.3.2 算法描述：

（1）FIRST 集：

如果 X 是终结符，则 FIRST(X) = {X}。

如果 X → Y1 Y2 ... Yn，则将 FIRST(Y1) 中的元素加入 FIRST(X)，如果 Y1 可以推导空串，则继续加入 FIRST(Y2)，以此类推。

（2）FOLLOW 集：

如果 A → αBβ，则将 FIRST(β) 中的元素加入 FOLLOW(B)，如果 β 可以推导空串，则将 FOLLOW(A) 中的元素加入 FOLLOW(B)。

（3）反复迭代，直到所有 FIRST 集和 FOLLOW 集稳定。

3.4 LL（1）文法判定与预测分析器

3.4.1 设计方案：

目标：判断一个文法是否为 LL(1)文法，并根据 FIRST 集和 FOLLOW 集设计一个预测分析器。

步骤：

（1）判断文法是否为 LL(1)文法，即对于任意两个产生式 A → α1 和 A → α2，它们的 FIRST 集必须不相交。

（2）构建一个预测分析表，根据文法规则、FIRST 集和 FOLLOW 集填充该表。

（3）使用该分析表进行语法分析。

3.4.2 算法描述：

（1）对于每个非终结符 A，检查其所有产生式的 FIRST 集：

如果两个产生式 A → α1 和 A → α2 具有相同的 FIRST 集，则文法不是 LL(1)文法。

如果文法满足此条件，则可以生成预测分析表。

（2）使用栈进行 LL(1)分析：
将输入符号和栈顶符号进行匹配，如果匹配则继续，如果不匹配则查找预测分析表中的相应产生式进行替换。
如果栈顶符号是非终结符且在分析表中有多个可能的产生式，则无法进行 LL(1)分析。

四、 测试结果

4.1 上下文无关文法的左递归消除

4.2 文法左公共因子的提取

采用实验任务书上的例子进行测试
S -> S+T | T　　T -> T*F | F　　F -> (E) | id

消除左递归:

编号	左部	右部	产生式
1	S	TS'	S->TS'
2	T	FT'	T->FT'
3	F	(E)	F->(E)
4	F	id	F->id
5	S'	+TS'	S'->+TS'
6	S'	ε	S'->ε
7	T'	*FT'	T'->*FT'
8	T'	ε	T'->ε

提取公因子:

编号	左部	右部	产生式
1	S	TS'	S->TS'
2	T	FT'	T->FT'
3	F	(E)	F->(E)
4	F	id	F->id
5	S'	+TS'	S'->+TS'
6	S'	ε	S'->ε
7	T'	*FT'	T'->*FT'
8	T'	ε	T'->ε

4.3 FIRST 集和 FOLLOW 集的求解和实现

采用课后作业第一题的例子来测试

S→MH | a　　H→LSo | ε　　K→dML | ε　　L→eHf　　M→K | bLM

```
文法的FIRST集:
FIRST(S) = {a,b,d,e,ε}
FIRST(H) = {e,ε}
FIRST(K) = {d,ε}
FIRST(L) = {e}
FIRST(M) = {b,d,ε}


文法的FOLLOW集:
FOLLOW(S) = {$,o}
FOLLOW(H) = {$,f,o}
FOLLOW(K) = {$,e,o}
FOLLOW(L) = {$,a,b,d,e,o}
FOLLOW(M) = {$,e,o}
```

4.4 LL（1）文法判定与预测分析器

采用课后作业第二题的例子来测试

S→a | ^ | (T)　　T→T,S | S

```
消除左递归:
 +------+------+------+----------+
| 编号 | 左部 | 右部 |  产生式  |
+------+------+------+----------+
|  1   |  S   |  a   |   S->a   |
|  2   |  S   |  ^   |   S->^   |
|  3   |  S   | (T)  |  S->(T)  |
|  4   |  T   | ST'  |  T->ST'  |
|  5   |  T'  | ,ST' | T'->,ST' |
|  6   |  T'  |  ε   |  T'->ε   |
+------+------+------+----------+


提取公因子:
 +------+------+------+----------+
| 编号 | 左部 | 右部 |  产生式  |
+------+------+------+----------+
|  1   |  S   |  a   |   S->a   |
|  2   |  S   |  ^   |   S->^   |
|  3   |  S   | (T)  |  S->(T)  |
|  4   |  T   | ST'  |  T->ST'  |
|  5   |  T'  | ,ST' | T'->,ST' |
|  6   |  T'  |  ε   |  T'->ε   |
+------+------+------+----------+
```

（消除左递归和提取左公共因子）

FIRST 集和 FOLLOW 集的求解以及 LL（1）文法判定与预测分析器

```
文法的FIRST集:
FIRST(S) = {(,^,a}
FIRST(T) = {(,^,a}
FIRST(T') = {,,ε}


文法的FOLLOW集:
FOLLOW(S) = {$,),,}
FOLLOW(T) = {)}
FOLLOW(T') = {)}


该文法是LL(1)文法


预测分析表:
 +----------+------+------+------+------+------+------+
| 非终结符 |  a   |  ^   |  (   |  )   |  ,   |  $   |
+----------+------+------+------+------+------+------+
|    S     |  a   |  ^   | (T)  | None | None | None |
|    T     | ST'  | ST'  | ST'  | None | None | None |
|    T'    | None | None | None |  ε   | ,ST' | None |
+----------+------+------+------+------+------+------+
```

然后采用(a,a),(a,a,a),(a,,a)来检测语法分析结果

这是（a,,a），显示语法分析结果失败

```
请输入字符串(end退出):(a,,a)
分析失败!没有找到对应的产生式。
+--------------------------+-----------------------------------+-----------+
|           栈              |              输入串               | 寻找产生式 |
+--------------------------+-----------------------------------+-----------+
|        ['$', 'S']        | ['(', 'a', ',', ',', 'a', ')', '$'] |   S->(T)  |
|    ['$', ')', 'T', '(']  | ['(', 'a', ',', ',', 'a', ')', '$'] |           |
|      ['$', ')', 'T']     |    ['a', ',', ',', 'a', ')', '$']   |   T->ST'  |
|   ['$', ')', "T'", 'S']  |    ['a', ',', ',', 'a', ')', '$']   |    S->a   |
|   ['$', ')', "T'", 'a']  |    ['a', ',', ',', 'a', ')', '$']   |           |
|      ['$', ')', "T'"]    |      [',', ',', 'a', ')', '$']      | T'->,ST'  |
| ['$', ')', "T'", 'S', ','] |     [',', ',', 'a', ')', '$']     |           |
+--------------------------+-----------------------------------+-----------+
```

(a,a)的语法分析结果：

```
请输入字符串(end退出):(a,a)
分析成功!
+--------------------------+--------------------------------+-----------+
|            栈             |             输入串             | 寻找产生式 |
+--------------------------+--------------------------------+-----------+
|         ['$', 'S']         | ['(', 'a', ',', 'a', ')', '$'] |   S->(T)   |
|    ['$', ')', 'T', '(']    | ['(', 'a', ',', 'a', ')', '$'] |            |
|      ['$', ')', 'T']       |   ['a', ',', 'a', ')', '$']    |   T->ST'   |
|   ['$', ')', "T'", 'S']    |   ['a', ',', 'a', ')', '$']    |    S->a    |
|   ['$', ')', "T'", 'a']    |   ['a', ',', 'a', ')', '$']    |            |
|      ['$', ')', "T'"]      |      [',', 'a', ')', '$']      |  T'->,ST'  |
| ['$', ')', "T'", 'S', ','] |      [',', 'a', ')', '$']      |            |
|   ['$', ')', "T'", 'S']    |        ['a', ')', '$']         |    S->a    |
|   ['$', ')', "T'", 'a']    |        ['a', ')', '$']         |            |
|      ['$', ')', "T'"]      |           [')', '$']           |   T'->ε    |
|         ['$', ')']         |           [')', '$']           |            |
|           ['$']            |             ['$']             |  分析成功  |
+--------------------------+--------------------------------+-----------+
```

(a,a,a)的语法分析结果：

```
请输入字符串(end退出):(a,a,a)
分析成功!
+--------------------------+----------------------------------------+-----------+
|            栈             |                 输入串                 | 寻找产生式 |
+--------------------------+----------------------------------------+-----------+
|         ['$', 'S']         | ['(', 'a', ',', 'a', ',', 'a', ')', '$'] |   S->(T)   |
|    ['$', ')', 'T', '(']    | ['(', 'a', ',', 'a', ',', 'a', ')', '$'] |            |
|      ['$', ')', 'T']       |   ['a', ',', 'a', ',', 'a', ')', '$']    |   T->ST'   |
|   ['$', ')', "T'", 'S']    |   ['a', ',', 'a', ',', 'a', ')', '$']    |    S->a    |
|   ['$', ')', "T'", 'a']    |   ['a', ',', 'a', ',', 'a', ')', '$']    |            |
|      ['$', ')', "T'"]      |      [',', 'a', ',', 'a', ')', '$']      |  T'->,ST'  |
| ['$', ')', "T'", 'S', ','] |      [',', 'a', ',', 'a', ')', '$']      |            |
|   ['$', ')', "T'", 'S']    |        ['a', ',', 'a', ')', '$']         |    S->a    |
|   ['$', ')', "T'", 'a']    |        ['a', ',', 'a', ')', '$']         |            |
|      ['$', ')', "T'"]      |           [',', 'a', ')', '$']           |  T'->,ST'  |
| ['$', ')', "T'", 'S', ','] |           [',', 'a', ')', '$']           |            |
|   ['$', ')', "T'", 'S']    |             ['a', ')', '$']              |    S->a    |
|   ['$', ')', "T'", 'a']    |             ['a', ')', '$']              |            |
|      ['$', ')', "T'"]      |                [')', '$']                |   T'->ε    |
|         ['$', ')']         |                [')', '$']                |            |
|           ['$']            |                  ['$']                   |  分析成功  |
+--------------------------+----------------------------------------+-----------+
```

实验收获：

通过这次实验，我深入理解了文法分析的基本概念和过程，特别是与 LL(1) 分析相关的算法。首先，通过左递归消除和左公共因子提取，我学会了如何改写文法，使其适用于 LL(1) 分析。这是构建语法分析器的重要步骤，因为只有消除左递归和提取公共因子后，文法才能被简化和规范化，从而进行高效的分析。其次，我在计算 FIRST 集和 FOLLOW 集的过程中，掌握了如何推导出每个非终结符的可达符号，这为后续的预测分析器构建提供了基础。通过实际编码实现这些算法，我对 LL(1) 文法判定的过程有了更深的理解，尤其是如何通过预测分析表来驱动分析过程。整体而言，这次实验不仅加深了我对文法和语法分析的理解，还提升了我在编译原理中解决实际问题的能力。

实验挑战：

在实验过程中，最大的挑战之一是理解和实现 LL(1) 文法的判定。由于 LL(1) 分析依赖于 FIRST 集和 FOLLOW 集的准确计算，因此如何正确地处理不同产生式的 FIRST 集交集问题，避免文法中的冲突是一个难点。在左递归消除时，尤其是在面对嵌套递归和复杂文法时，需要仔细区分递归的不同层级，并确保新产生式的正确性。左公共因子提取时，识别出公共前缀并进行合适的转换也需要小心，特别是对于具有多个相同前缀的产生式，如何有效地提取前缀并生成新的非终结符是一个难题。此外，在构建预测分析表和执行语法分析时，需要确保表格的完整性和准确性，否则会导致分析失败或错误。总体来说，这些挑战让我对文法变换和分析的细节有了更深的理解，并促使我加强了对编译原理中复杂算法的掌握。

五、 源代码

```
import re
import copy
from prettytable import PrettyTable


# 按终结符和非终结符遍历
def match_strings(A, input_str):
    # 优先匹配最长(A'和A识别成A')
    A = sorted(A, key=lambda x: len(x), reverse=True)
    pattern = '|'.join(map(re.escape, A))
    matches = re.findall(pattern, input_str)
    return matches


# 按终结符非终结符整体字符串倒序
def reverse_by_set(A, input_str):
    result = []
    i = len(input_str)
    while i > 0:
        for word in reversed(A):
            word_len = len(word)
            if i >= word_len and input_str[i - word_len:i] == word:
                result.append(word)
                i -= word_len
                break
        else:
            i -= 1
    return ''.join(result)


# 可视化输出
class draw_grammer:
```

```
    def draw_grammer(grammer, vn, descrpition):
        print_content = PrettyTable(['编号', '左部', '右部', '产生式'])
        idx = 1
        for i in vn:
            for j in grammer[i]:
                print_content.add_row([idx, i, j, i + '->' + j])
                idx += 1
        print('\n\n' + descrpition + ':\n', print_content)


# 消除左递归
class EliminateLeftRecursion:
    def __init__(self, grammer, vn):
        self.grammer = grammer
        self.vn = vn

    # 消除间接左递归
    def remove_left_recursion(self):
        new_grammer = copy.deepcopy(self.grammer)
        new_vn = copy.deepcopy(self.vn)

        # 两层循环暴露直接左递归
        for i in range(len(self.vn)):
            for j in range(0, i):
                # 检查是否有间接左递归
                if   self.has_indirect_left_recursion(self.vn[i],   self.vn[j],
new_grammer):
                    new_grammer    =    self.convert(self.vn[i],    self.vn[j],
new_grammer)

            new_grammer,   new_vn   =   self.clean_direct_recursion(self.vn[i],
new_grammer, new_vn)

        return new_grammer, new_vn

    # 检查是否存在间接左递归
    def has_indirect_left_recursion(self, ch_i, ch_j, grammer):
        # 通过深度优先搜索来检查 ch_i 是否通过 ch_j 间接递归回自身
        visited = set()
        stack = [ch_j]  # 从 ch_j 开始，检查是否能回到 ch_i

        while stack:
            current = stack.pop()
            if current == ch_i:  # 如果发现间接左递归
```

```
                return True
            if current not in visited:
                visited.add(current)
                if current in grammer:  # 查找所有生成 current 的非终结符
                    for item in grammer[current]:
                        if item:  # 不为空产生式
                            stack.append(item[0])  # 只考虑产生式的第一个符号，
判断是否是非终结符

        return False

    # 产生式右部非终结符转终结符
    def convert(self, ch_i, ch_j, grammer):
        rules = copy.deepcopy(grammer)
        for key in grammer.keys():
            for item_i in grammer[key]:
                if ch_i == key and ch_j == item_i[0]:
                    rules[key].remove(item_i)
                    for item_j in grammer[ch_j]:
                        rules[key].append(item_j + item_i[1:])
        return rules

    # 消除直接左递归
    def clean_direct_recursion(self, ch_i, grammer, new_vn):
        ch = ch_i + "'"
        flag = 0
        rules = copy.deepcopy(grammer)

        for key in grammer.keys():
            for item_i in grammer[key]:
                if ch_i == key and ch_i == item_i[0]:
                    flag = 1
                    # 添加新非终结符
                    if ch not in rules.keys():
                        rules[ch] = []
                    rules[ch].append(item_i[1:] + ch)
                    rules[key].remove(item_i)

        # 不存在左递归，直接返回
        if flag == 0:
            return rules, new_vn

        # 处理剩余的产生式
        for key in grammer.keys():
```

```
            for item_i in grammer[key]:
                if ch_i == key and ch_i != item_i[0]:
                    if ch not in rules.keys():
                        rules[ch] = []
                    rules[ch_i].append(item_i + ch)
                    rules[key].remove(item_i)

        # 添加新非终结符空串产生式
        rules[ch].append('ε')
        new_vn.append(ch)

        return rules, new_vn


# 提取左公因子
class ExtractCommonFactors:
    def __init__(self, grammer, vn):
        self.grammer = grammer
        self.vn = vn

    # 获取最长公共前缀
    def LCP(self, i, j, rules):
        strs = [rules[i], rules[j]]
        res = ''
        for each in zip(*strs):
            if len(set(each)) == 1:
                res += each[0]
            else:
                return res
        return res

    # 获取公共前缀索引
    def get_lcp_res(self, key):
        res = {}
        rules = self.grammer[key]
        for i in range(len(rules)):
            for j in range(i + 1, len(rules)):
                temp = self.LCP(i, j, rules)
                if temp not in res.keys():
                    res[temp] = set()
                res[temp].add(i)
                res[temp].add(j)
        # 去空串前缀
        if '' in res.keys():
```

```
            res.pop('')
        return res

    def remove_common_factor(self):
        keys = list(self.grammer.keys())
        for key in keys:
            while (True):
                res = self.get_lcp_res(key)
                # 直到没有公共前缀
                if (res == {}):
                    break
                dels = []  # 存即将删除的串
                lcp = list(res.keys())[0]  # 每次取一个公共前缀
                ch = key + "'"
                if ch not in self.vn:
                    self.vn.append(ch)
                # 遍历要消除公共因子的元素下标
                for i in res[lcp]:
                    string = self.grammer[key][i]
                    dels.append(string)
                    string = string.lstrip(lcp)
                    if string == '':
                        string += 'ε'
                    if ch not in self.grammer.keys():
                        self.grammer[ch] = []
                    # 加入新产生式
                    self.grammer[ch].append(string)
                # 删去原来产生式
                for string in dels:
                    self.grammer[key].remove(string)
                self.grammer[key].append(lcp + ch)
        return self.grammer, self.vn


# 文法分析
class LL1_analysis:
    def __init__(self, Gram):
        # 终结符 非终结符 分析表元素 $+开始符号
        self.vt,     self.vn,     self.analysis_table,     self.stack_str     =
self.init_all_(g=Gram)
        self.ptr = 0

    def init_all_(self, g):
        # 读取文法
```

```
        grammer_list = {}  # 非终结符：产生式
        vn_list = []  # 非终结符
        for line in re.split('\n', g):
            # 去空格
            line = "".join([i for i in line if i not in ['', ' ']])
            if '->' in line:
                if line.split('->')[0] not in vn_list:
                    vn_list.append(line.split('->')[0])
                for i in line.split('->')[1].split('|'):
                    if grammer_list.get(line.split('->')[0]) is None:
                        grammer_list[line.split('->')[0]] = []
                        grammer_list[line.split('->')[0]].append(i)
                    else:
                        grammer_list[line.split('->')[0]].append(i)
        draw_grammer.draw_grammer(grammer=grammer_list,              vn=vn_list,
descrpition='输入的文法')

        # 消除左递归
        # print('产生式：', grammer_list)
        # print('非终结符：', vn_list)
        eliminate_left_recursion = EliminateLeftRecursion(grammer=grammer_list,
vn=vn_list)
        new_grammer, new_vn = eliminate_left_recursion.remove_left_recursion()
        draw_grammer.draw_grammer(grammer=new_grammer, vn=new_vn, descrpition='
消除左递归')

        # 提取左公因子
        extractcommonfactors    =    ExtractCommonFactors(grammer=new_grammer,
vn=new_vn)
        new_grammer, new_vn = extractcommonfactors.remove_common_factor()
        draw_grammer.draw_grammer(grammer=new_grammer, vn=new_vn, descrpition='
提取公因子')

        only_grammer = []
        new_vt = []
        for i in new_vn:
            for j in new_grammer[i]:
                only_grammer.append(i + '->' + j)

                for t in j:  # 获取当前的所有的终结符
                    if t not in new_vt and t not in new_vn and t != "ε" and t !=
"":
                        # print(t)
                        new_vt.append(t)
```

```
        new_vt.append('$')
        # print('\n\n 消除文法左递归的文法的非终结符:',new_vn,
        #       '\n\n 消除文法左递归的文法的终结符:', new_vt)

        # FIRST 集和 FOLLOW 集
        FIRST, FOLLOW = self.get_first_and_follow_set(grammars=only_grammer,
vn=new_vn, vt=new_vt)
        print('\n\n 文法的 FIRST 集:')
        for i, j in FIRST.items():
            if j:  # 检查 j 是否为空
                str = j[0]
                for temp in j[1:]:
                    str = str + ',' + temp
                print("FIRST(" + i + ")" + " = {" + str + "}")
            else:
                print("FIRST(" + i + ")" + " = {}")

        print('\n\n 文法的 FOLLOW 集:')
        for i, j in FOLLOW.items():
            if j:  # 检查 j 是否为空
                str = j[0]
                for temp in j[1:]:
                    str = str + ',' + temp
                print("FOLLOW(" + i + ")" + " = {" + str + "}")
            else:
                print("FOLLOW(" + i + ")" + " = {}")

        # 分析表
        analysis_table = [[None] * (1 + len(new_vt)) for row in range(1 +
len(new_vn))]
        analysis_table[0][0] = ' '
        for i in range(len(new_vt)):
            analysis_table[0][i + 1] = new_vt[i]
        for i in range(len(new_vn)):
            analysis_table[i + 1][0] = new_vn[i]
        for i in range(len(new_vn)):
            for t in new_grammer[new_vn[i]]:  # 遍历该文法的所有产生式
                if t == 'ε':  # 如果是ε，对应在 FOLLOW 集中的终结符位置填上ε
                    for j in range(len(new_vt)):  # 遍历所有的终结符
                        if new_vt[j] in FOLLOW[new_vn[i]]:  # FOLLOW[part_begin]
为当前非终结符的 FOLLOW 集
                            # 如果分析表该位置为空，则填入ε
                            if analysis_table[i + 1][j + 1] is None:
                                analysis_table[i + 1][j + 1] = 'ε'
```

```
                else:
                    first_found = False  # 用于标记是否已经找到有效的FIRST项
                    for symbol in t:  # 遍历产生式右侧的每个符号
                        if symbol in new_vt:  # 如果是终结符
                            # 将该符号填入对应位置
                            j = new_vt.index(symbol)
                            if analysis_table[i + 1][j + 1] is None:
                                analysis_table[i + 1][j + 1] = t
                            first_found = True
                            break  # 终结符就直接填入，并停止检查其他符号
                        else:  # 如果是非终结符
                            # 使用该非终结符的FIRST集
                            for first_symbol in FIRST[symbol]:
                                if first_symbol != 'ε':  # 只处理非ε项
                                    j = new_vt.index(first_symbol)
                                    if analysis_table[i + 1][j + 1] is None:
                                        analysis_table[i + 1][j + 1] = t
                            # 如果该非终结符的FIRST集包含ε,需要继续检查后面的符号
                            if 'ε' in FIRST[symbol]:
                                continue
                            else:
                                first_found = True
                                break  # 如果FIRST集没有包含ε，停止检查后面的符号

                    # 如果右侧符号都能推导出ε，则检查FOLLOW集并填充
                    if not first_found:
                        for j in range(len(new_vt)):
                            if new_vt[j] in FOLLOW[new_vn[i]]:
                                if analysis_table[i + 1][j + 1] is None:
                                    analysis_table[i + 1][j + 1] = t
        # 判断是否为LL（1）文法
        is_ll1 = True
        for i in range(1, len(new_vn) + 1):
            for j in range(1, len(new_vt) + 1):
                if analysis_table[i][j] is not None and analysis_table[i][j] != 'ε':  # 如果当前位置有值且不是空串
                    for k in range(i + 1, len(new_vn) + 1):  # 对比同一非终结符的其他产生式
                        if analysis_table[k][j] == analysis_table[i][j] and analysis_table[k][j] != 'ε':  # 排除空串
                            is_ll1 = False
                            break
```

```
                if not is_lll:
                    break
            if not is_lll:
                break

        if is_lll:
            print("\n\n该文法是LL(1)文法")
        else:
            print("\n\n该文法不是LL(1)文法")

        # 输出分析表
        pretty_table_title = ['非终结符']
        for i in new_vt:
            pretty_table_title.append(i)
        analysis_pretty_table = PrettyTable(pretty_table_title)
        for i in range(len(analysis_table) - 1):
            analysis_pretty_table.add_row(analysis_table[i + 1])
        print('\n\n预测分析表:\n', analysis_pretty_table)

        # 返回预处理结构
        # print("new_vn:",new_vn[0])
        return new_vt, new_vn, analysis_table, '$' + new_vn[0]

    def get_first_and_follow_set(self, grammars, vn, vt):
        FIRST = {}
        FOLLOW = {}
        index = 0

        # 初始化 first 和 follow 集合
        for str in grammars:
            part_begin = str.split("->")[0]
            part_end = str.split("->")[1]
            FIRST[part_begin] = ""
            FOLLOW[part_begin] = ""
            index += 1

        # 设置开始符号的 FOLLOW 集合为 $
        start_symbol = grammars[0].split("->")[0]  # 假设第一个产生式的左部是开
始符号
        FOLLOW[start_symbol] = "$"

        # first 集
        # 1. 处理文法中的所有终结符
        vm = vt + vn  # 合并终结符和非终结符
```

```
        for rule in grammars:
            part_begin, part_end = rule.split("->")
            if part_end[0] != 'ε' and part_end[0] in vt:  # 如果第一个字符是终
结符
                FIRST[part_begin] += part_end[0]  # 将第一个终结符加入到 FIRST
集合

        # 2. 处理文法中的非终结符，递归添加其 FIRST 集合
        changed = True
        while changed:
            changed = False
            for rule in grammars:
                part_begin, part_end = rule.split("->")
                # 处理 A -> B 形式
                can_add_epsilon = True  # 用来判断右边的所有符号是否都可以推出
ε
                for i in range(len(part_end)):
                    first_symbol = part_end[i]
                    # 如果是终结符，直接加入到 FIRST 集合
                    if first_symbol in vt:
                        if first_symbol not in FIRST[part_begin]:
                            FIRST[part_begin] += first_symbol
                            changed = True
                        can_add_epsilon = False  # 遇到终结符后，不能推导出 ε
                        break
                    # 如果是非终结符
                    elif first_symbol in vn:
                        # 如果该非终结符可以推出 ε，则继续添加其 FIRST 集合
                        for symbol in FIRST[first_symbol]:
                            if symbol != 'ε' and symbol not in FIRST[part_begin]:
                                FIRST[part_begin] += symbol
                                changed = True
                        # 如果非终结符无法推出 ε，跳出循环
                        if 'ε' not in FIRST[first_symbol]:
                            can_add_epsilon = False
                            break
                    # 如果遇到 ε，则继续处理
                    elif first_symbol == 'ε':
                        if 'ε' not in FIRST[part_begin]:
                            FIRST[part_begin] += 'ε'
                            changed = True
                # 如果右侧所有符号都可以推出 ε，则加入 ε 到 part_begin 的
FIRST 集
                if can_add_epsilon and 'ε' not in FIRST[part_begin]:
```

```
                    FIRST[part_begin] += 'ε'
                    changed = True

        # 去重，确保每个集合中的字符都是唯一的
        for non_terminal in FIRST:
            FIRST[non_terminal] = ''.join(sorted(set(FIRST[non_terminal])))
        # follow 集
        for i in range(len(vn)):
            while True:
                test = FOLLOW.copy()  # 使用 copy 来判断是否有变化
                for rule in grammars:
                    part_begin, part_end = rule.split("->")
                    # S -> a 直接推出终结符则继续
                    if (len(match_strings(vm, part_end)) == 1 and part_end in vt):
                        continue
                    else:
                        temp = match_strings(vm + ["ε"], reverse_by_set(vm + ["
ε"], part_end))
                        if temp[0] in vn:
                            FOLLOW[temp[0]]   =   FOLLOW.get(temp[0],   '')   +
FOLLOW.get(part_begin, '')
                            temp1 = temp[0]
                            for i in temp[1:]:
                                if i in vt:
                                    temp1 = i
                                else:
                                    if temp1 in vn:
                                        FOLLOW[i]   =   FOLLOW.get(i,   '')   +
FIRST.get(temp1, '').replace("ε", "")
                                        first_set = FIRST.get(temp1, set())
                                        if 'ε' in first_set:
                                            FOLLOW[i]   =   FOLLOW.get(i,   '')   +
FOLLOW.get(part_begin, '')
                                        temp1 = i
                        else:
                            temp1 = temp[0]
                            for i in temp[1:]:
                                if i in vt:
                                    temp1 = i
                                else:
                                    if temp1 in vn:
                                        FOLLOW[i]   =   FOLLOW.get(i,   '')   +
FIRST.get(temp1, '')
                                        else:
```

```
                                    FOLLOW[i] = FOLLOW.get(i, '') + temp1
                                temp1 = i
            # follow 集去重
            for i, j in FOLLOW.items():
                FOLLOW[i] = ''.join(sorted(set(j)))

            # 去除 FOLLOW 集中的 'ε'
            for non_terminal in FOLLOW:
                FOLLOW[non_terminal] = FOLLOW[non_terminal].replace('ε', '')

            if test == FOLLOW:
                break

        return FIRST, FOLLOW

    # LL（1）分析过程
    def LL1_analysis_solve(self, goal_str, ans_table):
        vt, vn, analysis_table, stack_str, ptr = self.vt, self.vn, self.analysis_table, self.stack_str, self.ptr
        vm = vn + vt
        goal_str = match_strings(vm + ["ε"], goal_str)
        stack_str = match_strings(vm + ["ε"], stack_str)
        lookup_table = None
        shuchu = ''

        while ptr >= 0 and ptr < len(goal_str):  # 确保指针在输入字符串范围内
            if not stack_str:
                print("分析失败！栈为空，输入串未完全匹配。")
                return

            stack_top = stack_str[len(stack_str) - 1]  # 获取栈顶
            goal_pos = goal_str[ptr]  # 获取当前输入符号

            # 非法输入的情况
            if (stack_top not in vt and stack_top not in vn) or goal_pos not in vt:
                print('输入不合法！')
                return
            elif stack_top == goal_pos:  # 栈顶符号 = 当前输入符号
                if stack_top == '$':  # 栈顶符号 = 当前输入符号 = '$'
                    print('分析成功！')
                    ans_table.add_row([stack_str, goal_str[ptr:], '分析成功'])
                    return
                else:  # 栈顶符号 = 当前输入符号，但是不等于 $
```

```
                    ans_table.add_row([stack_str, goal_str[ptr:], ''])
                    stack_str = stack_str[0:len(stack_str) - 1]  # 弹栈
                    ptr += 1  # 输入指针前移
                    continue

            # 如果栈顶是非终结符
            if stack_top in vn:
                stack_top_index = vn.index(stack_top)
                goal_pos_index = vt.index(goal_pos)
                # 防止索引越界
                if stack_top_index < len(analysis_table) and goal_pos_index < len(analysis_table[0]):
                    lookup_table = analysis_table[stack_top_index + 1][goal_pos_index + 1]
                    if lookup_table is not None:  # 如果找到对应的产生式
                        # 弹栈处理 ε
                        if lookup_table == 'ε':
                            shuchu = "".join(stack_top) + "->" + "".join(lookup_table)
                            ans_table.add_row([stack_str, goal_str[ptr:], shuchu])
                            stack_str = stack_str[0:len(stack_str) - 1]  # 弹栈
                            continue
                        else:
                            # 存在对应产生式，反向压栈
                            shuchu = "".join(stack_top) + "->" + "".join(lookup_table)
                            ans_table.add_row([stack_str, goal_str[ptr:], shuchu])
                            stack_str = stack_str[0:len(stack_str) - 1]  # 弹栈
                            stack_str += match_strings(vm + ["ε"], reverse_by_set(vm + ["ε"], lookup_table))
                            continue
                    else:
                        print('分析失败！没有找到对应的产生式。')
                        return
                else:
                    print(f"分析失败！索引越界，stack_top_index: {stack_top_index}, goal_pos_index: {goal_pos_index}")
                    return

            # 如果栈顶是终结符，直接匹配
            elif stack_top in vt:
                if stack_top == goal_pos:  # 栈顶符号和输入符号匹配
```

```
                    stack_str = stack_str[0:len(stack_str) - 1]  # 弹栈
                    ptr += 1  # 输入指针前移
                    continue
                else:
                    print(f"分析失败!栈顶符号 {stack_top} 与输入符号 {goal_pos} 不匹配。")
                    return


if __name__ == '__main__':
    with open(r'D:\bianyiyuanli3\test2', 'r', encoding='utf-8') as file:
        lll_analysis = LL1_analysis(Gram=file.read())
    while True:
        goal_str = input('请输入字符串(end退出):') + '$'
        if goal_str == 'end$':
            break
        result_table = PrettyTable(['栈', '输入串', '寻找产生式'])
        lll_analysis.LL1_analysis_solve(goal_str=goal_str, ans_table=result_table)
        print(result_table)
```