北京邮电大学

编译原理实验报告

LL(1)语法分析程序设计

姓名：陈朴炎　　学号：2021211138

2023-11-7

目录

1 概述

1.1 问题描述

语法分析程序的设计与实现：
编写语法分析程序，实现对算术表达式的语法分析。要求所分析算数表达式由如下的文法产生。

$$E \rightarrow E+T \mid E-T \mid T$$

$$T \rightarrow T*F \mid T/F \mid F$$

$$F \rightarrow (E) \mid num$$

图 1-1 文法示意图

1.2 实现方法

下述两种方法二选一
方法 1：编写递归调用程序实现自顶向下的分析。
方法 2：编写 LL(1)语法分析程序，要求如下。
(1)编程实现算法 4.2，为给定文法自动构造预测分析表。
(2)编程实现算法 4.1，构造 LL(1)预测分析程序。
本次实验我采用方法 2，在实现算法 4.2 和算法 4.1 的基础上，外加了实现了消除左递归算法，还有计算 FIRST、FOLLOW 集算法。

2 实验环境

在 windows11 下，采用 python 3.10.7-64 bit

2.1 编辑器选择

Vscode 版本：1.84.1 x64
插件：
ms-python.isort
ms-python.python
ms-python.vscode-pylance
ms-toolsai.jupyter
ms-toolsai.jupyter-keymap
ms-toolsai.jupyter-renderers

ms-toolsai.vscode-jupyter-cell-tags
ms-toolsai.vscode-jupyter-slideshow
编码：utf-8

3 文法预处理

3.1 消除左递归

对于P→Pα1 / Pα2 /…/ Pαn / β1 / β2 /…/βm
其中，αi（I＝1，2，…，n）都不为 ε，而每个 βj（j＝1，2，…，m）都不以P开头，将上述规则改写为如下形式即可消除P的直接左递归：
P→β1 P’ / β2 P’ /…/βm P’
P’ →α1P’ / α2 P’ /…/ αn P’ /ε

3.2 构造 FIRST 首符集

FIRST集的定义如下：
FIRST(α)={a|α=>aβ， a∈Vt， α， β∈V*}，若 α=>(*) ε 则规定 ε∈FRIST(α)。
先看这个定义的主干部分，首符集里面的组成元素都是终结符，什么样的终结符？产生式右边第一个位置上的终结符，它会被纳入产生式左边的非终结符的首符集。
这个定义的补充部分意思是，如果非终结符可经多步推到得到ε，那么ε也纳入它的首符集。
FIRST集的构造步骤如下：
1.若X ∈Vt，则FIRST(X)={X}
2.若X∈Vn，且有产生式X→a…，则把a加入到FIRST(X)中；
若X→ε也是一个产生式，则把ε也加到FIRST(X)中。
3.若 X→Y…是一个产生式且 Y∈Vn，则把 FIRST(Y)中的所有非ε元素都加到FIRST(X)中;若X → Y1Y2…YK 是一个产生式,Y1,Y2,…,Y(i-1)都是非终结符，而且，对于任何 j,1≤j ≤i-1, FIRST(Yj)都含有ε (即Y1…Y(i-1) =>(*)ε)，则把 FIRST(Yj)中的所有非ε元素都加到 FIRST(X)中；特别是，若所有的FIRST(Yj ， j=1,2,…,K)均含有ε，则把 ε 加到FRIST(X)中。

3.3 构造 FOLLOW 随附集

FOLLOW集的定义如下：
FOLLOW(A)={a | S=>()μ A β 且a ∈ FRIST(β)， μ ∈V， β∈V+ }
若S=>()u A β ，且 β=>() ε，则#∈FOLLOW(A)
同样，我们先看这个定义的主干部分，FOLLOW集里面的元素都来自首符集，说明也都是非终结符，与首符集不同的是，A的FOLLOW集，A出现在产生式的右侧，而不是左侧。
这个定义的补充部分意思是，如果一个产生式经过多部推导后得到u A β，而

跟在 A 后面的 β 又能够经过多步推导得到 ε，则把#加到 FOLLOW(A)。
FOLLOW 集的构造步骤如下：
1 对于文法的开始符号 S，置#于 FOLLOW(S) 中。
2 若 A→α B β 是一个产生式,则把 FIRST(β)中的所有非ε元素加至 FOLLOW(B)中，把 FIRST(β)中的 ε 换成#加至 FOLLOW(B)。
3 若 A→α B 是一个产生式，或 A→ αBβ 是 一个产生式而 β=>(*)ε (即 ε∈FIRST(β))， 则把 FOLLOW(A)加至 FOLLOW(B)中。

4 文法分析过程

4.1 构造分析预测表

1、对 G 中任意一个产生式 A --> α 执行第 2 步和第 3 步
2、for 任意 a ∈ First(α)，将 A --> α 填入 M[A,a]
3、if ε ∈ First(α) then 任意 a ∈ Follow(A)，将 A --> α 填入 M[A,a]
if ε ∈ First(α) & # ∈Follow(A)， then 将 A --> α 填入 M[A,#]
4、将所有没有定义的 M[A,b] 标上出错标志 （留空也可以）

4.2 分析程序说明

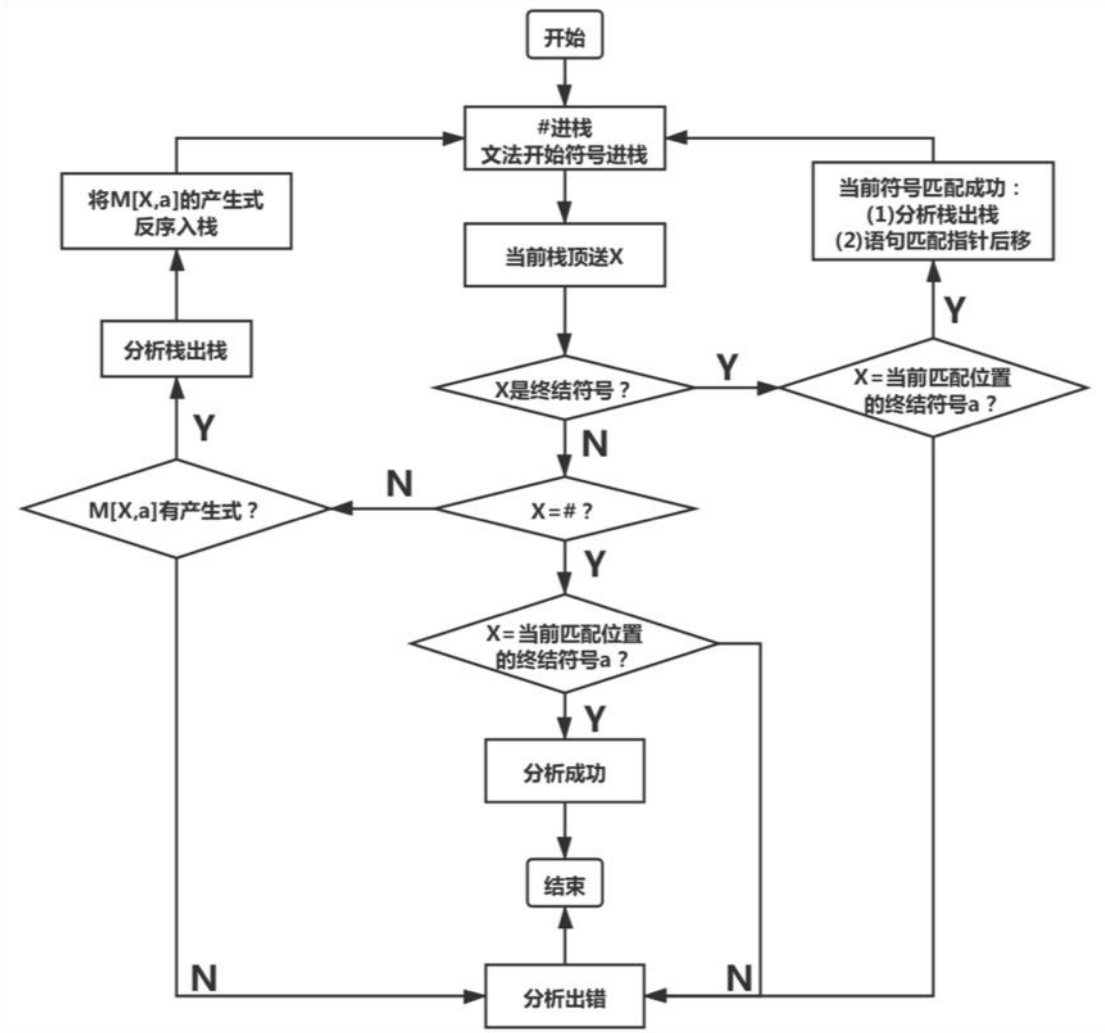


图 4-1 LL(1)分析程序流程说明图

5 程序设计

5.1 变量定义说明

```
Python
T = {"(", ")", "+", "-", "*", "/", "num"}
N = {"E", "T", "F"}
P = {
        'E':['E + T', 'E - T', 'T'],
        'T':['T * F', 'T / F', 'F'],
        'F':['( E )','num']
    }
S = 'E'
```

程序启动初始，我先将该文法定义为如上四个集合和符号，N 表示非终结符，T

表示终结符，P 表示产生式，S 表示文法开始符号。其中文法产生式 P 是一个字典，P 中键值对的键对应文法产生式的左端，是一个非终结符；P 中键值对的值对应文法产生式的右端，它是一个列表，列表中的每个元素都代表了一个产生式右端，如 E->E+T|E-T|T，每个元素存储一个像”E+T”或”E-T”这样的值。

5.2 功能函数定义说明

5.2.1 is_epsilon()

Python

```
# 判断单个字符可不可以推导为空
def is_epsilon(A, N, first):
    if A == 'ε':
        return True
    if A not in N:
        return False
    result = False
    for symbol in first[A]:
        if symbol == 'ε':
            result = True
            break
    return result
```

该函数对 A 这个字符进行判断，判断该字符能否推导成空串 ε，如果 A 直接就是 ε，那就返回 True；如果 A 不在非终结符里，说明它是终结符，就返回 False；如果 A 在终结符里,并且 A 的首符集里有 ε,那就说明 A 能推导成空,返回 True。

5.2.2 is_list_epsilon()函数

Python

```
# 判断字符串可不可以推导成空

def is_list_epsilon(list, N, first):

    result = True

    for item in list:

        if not is_epsilon(item, N, first):

            return False

    return result
```

该函数判断字符串 list 能否推导成空，调用了 is_epsilon()函数，来从头逐一判断 list 的前缀中的每个元素能否推导出 ε，如果全都能，就返回 True；否则，若其中某一个元素不能成空，则返回 False

5.2.3 cal_list_first()

Python

```
# 找出 list 串的 first 集

def cal_list_first(list, N, first):

    FIRST = set()

    for i, item in enumerate(list):

        if item not in N:

            FIRST = {item}

            break

        FIRST.update(first[item])

        if not is_epsilon(item, N, first):

            break
```

```
    return FIRST
```

该函数通过遍历字符串 list，如果遇到终结符，就直接返回其本手，否则返回非终结符的首符集，如果该非终结符能推导出空，那么还得加上去掉该非终结符之后字符串的首符集。

5.3 消除左递归

5.3.1 基本思路

对于 P→Pα1 / Pα2 /…/ Pαn / β1 / β2 /…/βm
其中，αi（I＝1，2，…，n）都不为 ε，而每个 βj（j＝1，2，…，m）都不以 P 开头，将上述规则改写为如下形式即可消除 P 的直接左递归：
P→β1 P’ / β2 P’ /…/βm P’
P’ →α1P’ / α2 P’ /…/ αn P’ /ε

5.3.2 算法设计

鉴于此，我在算法设计实现时先复制了原有的产生式，然后对每个产生式进行遍历分析，收集对应非终结符的产生式的α和β列表，将α1、α2…αn 收集到 alpha_productions 里，同理把β列表收集到 beta_productions 里。
对于有左递归的文法产生式，首先创建一个新的文法非终结符号，比如 E 对应的就构造出 E’，然后对于原来的 E 的产生式，消除完左递归之后就变成如下形式：
E’ -> + T E’ | - T E’ | ε
E -> T E’
同理，还有 T 的文法产生式
T -> F T’
T’ -> * F T’ | / F T’ | ε

5.3.3 算法实现

Python

```
def eliminate_left_recursion(N, P):

    new_non_terminals = []

    new_productions = P.copy()  # 复制原有的产生式字典，以便修改



    for non_terminal in N:
```

```
            productions = P[non_terminal]
            alpha_productions = []
            beta_productions = []

            for production in productions:
                if production.startswith(non_terminal):

alpha_productions.append(production[len(non_terminal):])
                else:
                    beta_productions.append(production)

            if alpha_productions:
                new_non_terminal = non_terminal + "'"
                new_non_terminals.append(new_non_terminal)

                new_alpha_productions = [p +" "+ new_non_terminal for
p in alpha_productions] + ['ε']
                new_productions[non_terminal]    =    [p    +    "    "
+new_non_terminal for p in beta_productions]
                new_productions[new_non_terminal]                    =
new_alpha_productions
```

```
N = N.union(new_non_terminals)


return N, new_productions
```

5.4 构造 FIRST 集

5.4.1 基本思路

FIRST 集的定义如下：
FIRST(α)={a|α=>aβ， a∈Vt， α， β∈V*}，若 α=>(*)ε 则规定 ε∈FRIST(α)。
先看这个定义的主干部分，首符集里面的组成元素都是终结符，什么样的终结符？产生式右边第一个位置上的终结符，它会被纳入产生式左边的非终结符的首符集。
这个定义的补充部分意思是，如果非终结符可经多步推到得到ε，那么ε也纳入它的首符集。
FIRST 集的构造步骤如下：
1.若 X ∈Vt，则 FIRST(X)={X}
2.若 X∈Vn，且有产生式 X→a…，则把 a 加入到 FIRST(X)中；
若 X→ε也是一个产生式，则把ε也加到 FIRST(X)中。
3.若 X→Y…是一个产生式且 Y∈Vn，则把 FIRST(Y)中的所有非ε元素都加到 FIRST(X)中;若 X → Y1Y2…YK 是一个产生式,Y1,Y2,…,Y(i-1)都是非终结符，而且，对于任何 j,1≤j ≤i-1, FIRST(Yj)都含有ε (即 Y1…Y(i-1) =>(*)ε)，则把 FIRST(Yj)中的所有非ε元素都加到 FIRST(X)中；特别是，若所有的 FIRST(Yj ， j=1,2,…,K)均含有ε，则把 ε 加到 FRIST(X)中。

5.4.2 算法设计

首先，初始化一个字典 FIRST，其中键是所有非终结符和终结符，以及 ε（空串），值是对应符号的 FIRST 集，初始时都为空集。
然后，将所有终结符（包括 ε）的 FIRST 集初始化为它们自身，因为终结符的 FIRST 集就是它们本身。
接下来，使用循环来逐步计算文法的 FIRST 集。通过不断遍历文法规则，直到不再发生变化为止。
在每个非终结符的产生式中，分析每个符号。如果是终结符，直接将它加入到对应非终结符的 FIRST 集中。
如果是非终结符，将非终结符的 FIRST 集中的符号加入到对应非终结符的 FIRST 集中，同时检查是否包含 ε，如果不包含 ε，则不再往后推导。
最后，如果产生式中的所有符号的 FIRST 集都包含 ε，那么 ε 也要加入到对

应非终结符的 FIRST 集中。

5.4.3 算法实现

Python

```
def compute_first(N, T, P):
    print(P)

    # 初始化 FIRST 集，将所有非终结符的 FIRST 集初始化为空集
    FIRST = {symbol: set() for symbol in N.union(T).union({'ε'})}

    # 终结符的 FIRST 集就是其自身
    for terminal in T.union({'ε'}):
        FIRST[terminal].add(terminal)

    # 根据文法规则逐步计算 FIRST 集
    changed = True
    while changed:
        changed = False
        for non_terminal in N:
            for production in P[non_terminal]:
                symbols = production.split()
                for symbol in symbols:
```

```
                    if symbol in T.union({"ε"}):
                        # 如果是终结符，直接加入 FIRST 集
                        if symbol not in FIRST[non_terminal]:
                            FIRST[non_terminal].add(symbol)
                            changed = True
                        break
                    else:
                        # 如果是非终结符，加入其 FIRST 集
                        for sym in FIRST[symbol]:
                            if sym != 'ε' and sym not in
FIRST[non_terminal]:
                                FIRST[non_terminal].add(sym)
                                changed = True
                        if 'ε' not in FIRST[symbol]:
                            # 如果该非终结符不包含 ε，不再往后推导
                            break
                    # 如果所有符号的 FIRST 集都包含 ε，那么 ε 也加入
FIRST 集
                    if all(sym == 'ε' or sym in FIRST[sym] for sym in
production):
                        if 'ε' not in FIRST[non_terminal]:
```

```
                    FIRST[non_terminal].add('ε')

                    changed = True



return FIRST
```

5.5 构造 FOLLOW 集

5.5.1 基本思路

FOLLOW 集的定义如下：
FOLLOW(A)={a | S=>() μ A β 且 a ∈ FRIST(β)， μ ∈V， β∈V+ }
若 S=>()u A β ，且 β=>() ε，则#∈FOLLOW(A)
同样，我们先看这个定义的主干部分，FOLLOW 集里面的元素都来自首符集，说明也都是非终结符，与首符集不同的是，A 的 FOLLOW 集，A 出现在产生式的右侧，而不是左侧。
这个定义的补充部分意思是，如果一个产生式经过多部推导后得到 u A β，而跟在 A 后面的 β 又能够经过多步推导得到 ε，则把#加到 FOLLOW(A)。
FOLLOW 集的构造步骤如下：
1 对于文法的开始符号 S，置#于 FOLLOW(S) 中。
2 若 A→ α B β 是一个产生式,则把 FIRST(β)中的所有非ε元素加至 FOLLOW(B)中，把 FIRST(β)中的 ε 换成#加至 FOLLOW(B)。
3 若 A→ α B 是一个产生式，或 A→ αBβ 是 一个产生式而 β=>(*)ε (即 ε∈FIRST(β))， 则把 FOLLOW(A)加至 FOLLOW(B)中。

5.5.2 算法设计

首先，初始化一个字典 FOLLOW，其中键是所有非终结符，值是对应非终结符的 FOLLOW 集，初始时都为空集。
然后，设定起始符号 S 的 FOLLOW 集为 '$'，即文法的结束标志。
使用循环来进行 FOLLOW 集的计算，你使用了两轮循环，可能是因为 FOLLOW 集的计算需要多次迭代以确保每个非终结符的 FOLLOW 集都被正确计算。
在每一轮循环中，遍历文法的每个产生式。对于每个产生式中的非终结符，分析其后的符号。
如果后续符号不是非终结符，就跳过。如果是非终结符，计算后续符号列表的 FIRST 集（可能是 cal_list_first 函数的功能），然后将这些符号添加到当前非终结符的 FOLLOW 集中。
如果后续符号列表的 FIRST 集包含 ε，那么还需要将当前非终结符的 FOLLOW 集添加到它的 FOLLOW 集中。然后，去除 FOLLOW 集中的 ε 符号。
循环迭代这些步骤，直到 FOLLOW 集不再发生变化。

5.5.3 算法实现

Python

```
def compute_follow(N, P, S, first):
    # 初始化 FOLLOW 集，将所有非终结符的 FOLLOW 集初始化为空集
    FOLLOW = {non_terminal: set() for non_terminal in N}

    # 设定起始符号 S 的 FOLLOW 集为$
    FOLLOW[S].add('$')
    for i in range(2):
        for non_terminal in N:
            for production in P[non_terminal]:
                symbols = production.split()
                for i, symbol in enumerate(symbols):
                    if symbol not in N:
                        continue
                    list = symbols[i+1:]
                    if is_list_epsilon(list, N, first):
                        FOLLOW[symbol].update(cal_list_first(list,  N,
first))

FOLLOW[symbol].update(FOLLOW[non_terminal])
```

```
                    else:
                        FOLLOW[symbol].update(cal_list_first(list,  N,
first))
                    FOLLOW[symbol].discard("ε")


    return FOLLOW
```

5.6 构造分析表

5.6.1 基本思路

1、对 G 中任意一个产生式 A --> α 执行第 2 步和第 3 步
2、for 任意 a ∈ First(α)，将 A --> α 填入 M[A,a]
3、if ε ∈ First(α) then 任意 a ∈ Follow(A)，将 A --> α 填入 M[A,a]
if ε ∈ First(α) & # ∈Follow(A)， then 将 A --> α 填入 M[A,#]
4、将所有没有定义的 M[A,b] 标上出错标志 （留空也可以）

5.6.2 算法设计

首先，初始化一个 predict_map 字典，其中键是非终结符，值是一个嵌套字典，内部字典的键是终结符和结束符（$），值是一个空列表，用于存储产生式。
然后，对文法的每个非终结符进行迭代，为每个非终结符创建一个空字典，用于存储该非终结符在不同终结符下的产生式。
针对每个非终结符，遍历其产生式。对于每个产生式，首先计算产生式右侧符号串的 FIRST 集（可能是 cal_list_first 函数的功能）。
遍历 FIRST 集中的终结符，将该终结符对应的产生式添加到 predict_map 中。这样，就构建了文法的预测分析表的部分内容。
如果 FIRST 集中包含 ε（空串），则还需要考虑 FOLLOW 集中的终结符，将对应的产生式也添加到 predict_map 中。这是为了处理可能的 ε 推导。
最后，返回构建好的 predict_map 预测分析表。

5.6.3 算法实现

Python
def build_predict_map(N, T, P, first, follow):

```
    predict_map = {}
    for non_terminal in N:
        predict_map[non_terminal] = {}
        for terminal in T.union({"$"}):
            predict_map[non_terminal][terminal] = []

    for non_terminal in N:
        for production in P[non_terminal]:
            symbols = production.split()
            FIRST = cal_list_first(symbols, N, first)
            for a in FIRST:
                if a not in T:
                    break
                predict_map[non_terminal][a].append(production)
            if "ε" in FIRST:
                for b in follow[non_terminal]:

predict_map[non_terminal][b].append(production)

    return predict_map
```

5.7 构造分析程序

5.7.1 基本思路

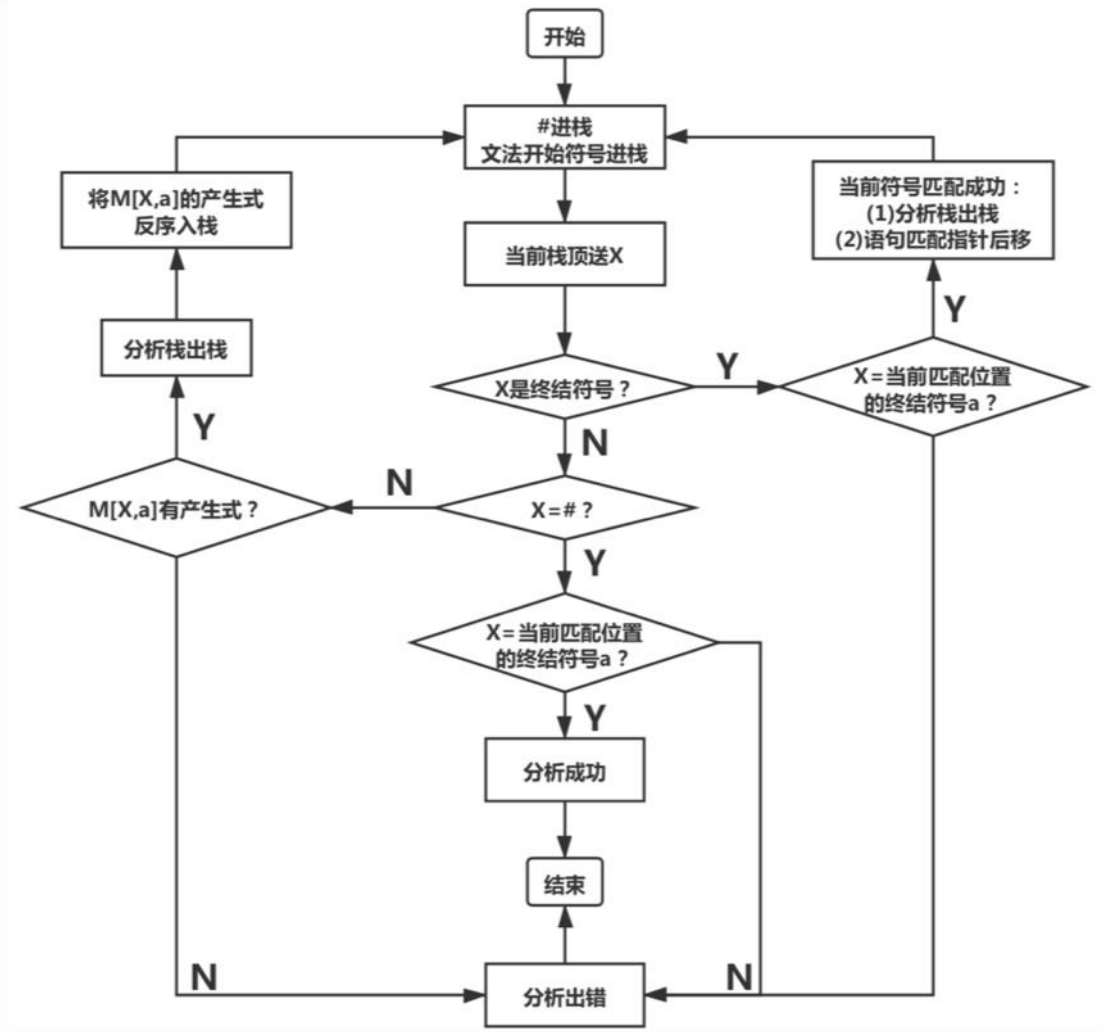


图 5-1 LL(1)程序思路

5.7.2 算法设计

初始化栈（stack）和指向输入串的指针（ip）。栈初始包含结束符$和文法的开始符号 S，指针指向输入串的第一个符号。

接受用户输入的待分析的输入串，将其分割成符号，并存储在 input_string 中。

进入主循环，直到栈顶符号为$且输入串被完全分析。

在每次迭代中，首先检查栈顶符号 x 和输入串中的当前符号 a。

如果 x 是终结符或结束符$，则检查它是否与当前输入符号 a 匹配。如果匹配，则弹出栈顶符号 x，并将指针 ip 前进一步。

如果 x 是空串 ε，则只需将其从栈中弹出。

否则，x 是非终结符。根据预测分析表 predict_map，查找非终结符 x 和输入符号 a 对应的产生式。如果找到匹配的产生式，将其应用于栈顶符号，并将其反向推入栈中。

在每次循环中，还会处理可能的多个产生式的选择，因此会显示哪个产生式被使用。
在分析完成后，还会检查栈中是否存在可以生成空串 ε 的非终结符，将它们从栈中弹出。

最后，检查栈是否只包含结束符$，以及指针 ip 是否指向输入串的末尾。如果满足这两个条件，则分析成功，否则分析失败。

5.7.3 算法实现

Python

```
def ll_parser(T, S, predict_map, first):
    stack = ["$", S]      # 初始化栈
    ip = 0                  # 向前指针指向第一个符号
    input_string = input("请输入待分析的串，每个符号用空格分隔:
").split()

    while stack[-1] != "$" and ip < len(input_string):

        print("栈状态",stack)
        x = stack[-1]
        a = input_string[ip]
        print("a:",a)
        print("x:",x)
        if x in T or x == "$":
            if x == a:
```

```
                stack.pop()
                ip += 1
                print("匹配到",a,"弹出")
            else:
                print("Error: 无法匹配输入符号", a)
                break
        elif x == "ε":
            stack.pop()
        else:
            production = predict_map[x][a]
            for proc in production:
                proc_list = proc.split()
            print("使用产生式：", proc_list)
            stack.pop()
            if production != 'ε':
                stack.extend(reversed(proc_list))
            else:
                print("Error: 找不到合适的产生式")
                break

while len(stack) >= 2 and "ε" in first[stack[-1]]:
```

```
        print("由于栈顶符号"+stack[-1]+"可以生成 ε，弹出")
        stack.pop()



print(stack)
print(ip)
if stack == ['$'] and ip == len(input_string):
        print("分析成功")
else:
        print("分析失败")
```

6 测试报告

6.1 运行程序，得到新文法及 FIRST、FOLLOW 集合和预测分析表

运行程序，首先，会显示出文法在经过消除左递归后的新文法；然后计算 FIRST 集合，并打印；接着计算 FOLLOW 集合并打印；再构造预测分析表，并可视化出来；最后提示用户输入预测的字符串信息，效果如下图 6-1：

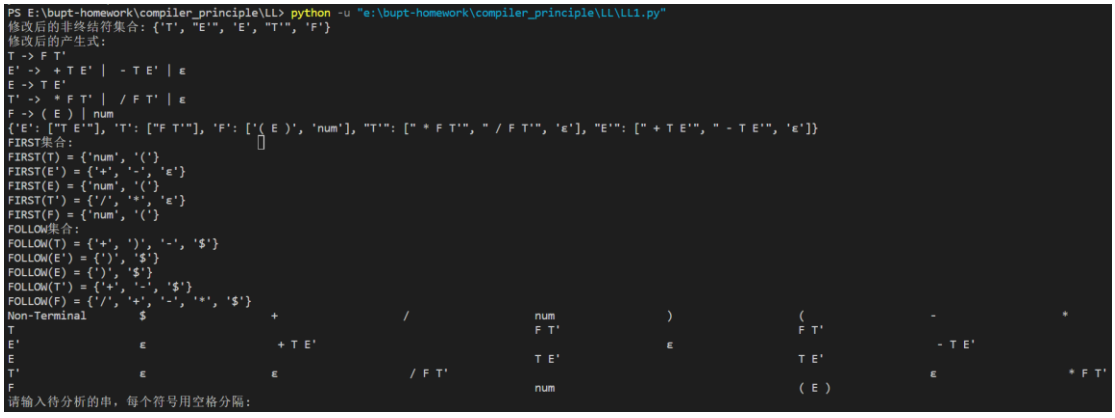
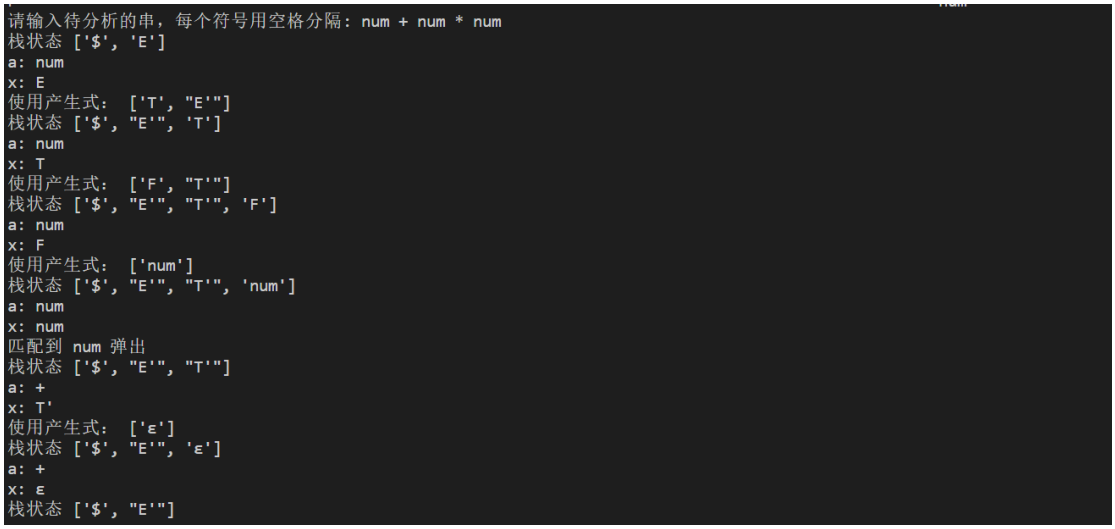


图 6-1 程序初始化阶段

经过核验，初始化结果均没出错。

6.2 测试用例 1

输入：num + num * num
执行过程截图：



6-2 测试用例 1 执行过程 1

6-3 测试用例 1 执行过程 2

最后，由于剩下两个可以推出 ε 的非终结符，因此两个符号都弹出，栈里只剩下$符号，分析成功。
执行过程文字：

Python
请输入待分析的串，每个符号用空格分隔：num + num * num 栈状态 ['$', 'E'] a: num x: E 使用产生式： ['T', "E'"]

```
栈状态 ['$', "E'", 'T']
a: num
x: T
使用产生式： ['F', "T'"]
栈状态 ['$', "E'", "T'", 'F']
a: num
x: F
使用产生式： ['num']
栈状态 ['$', "E'", "T'", 'num']
a: num
x: num
匹配到 num 弹出
栈状态 ['$', "E'", "T'"]
a: +
x: T'
使用产生式： ['ε']
栈状态 ['$', "E'", 'ε']
a: +
x: ε
栈状态 ['$', "E'"]
a: +
x: E'
使用产生式： ['+', 'T', "E'"]
栈状态 ['$', "E'", 'T', '+']
a: +
x: +
匹配到 + 弹出
栈状态 ['$', "E'", 'T']
a: num
x: T
使用产生式： ['F', "T'"]
栈状态 ['$', "E'", "T'", 'F']
a: num
x: F
使用产生式： ['num']
栈状态 ['$', "E'", "T'", 'num']
a: num
x: num
匹配到 num 弹出
栈状态 ['$', "E'", "T'"]
a: *
x: T'
```

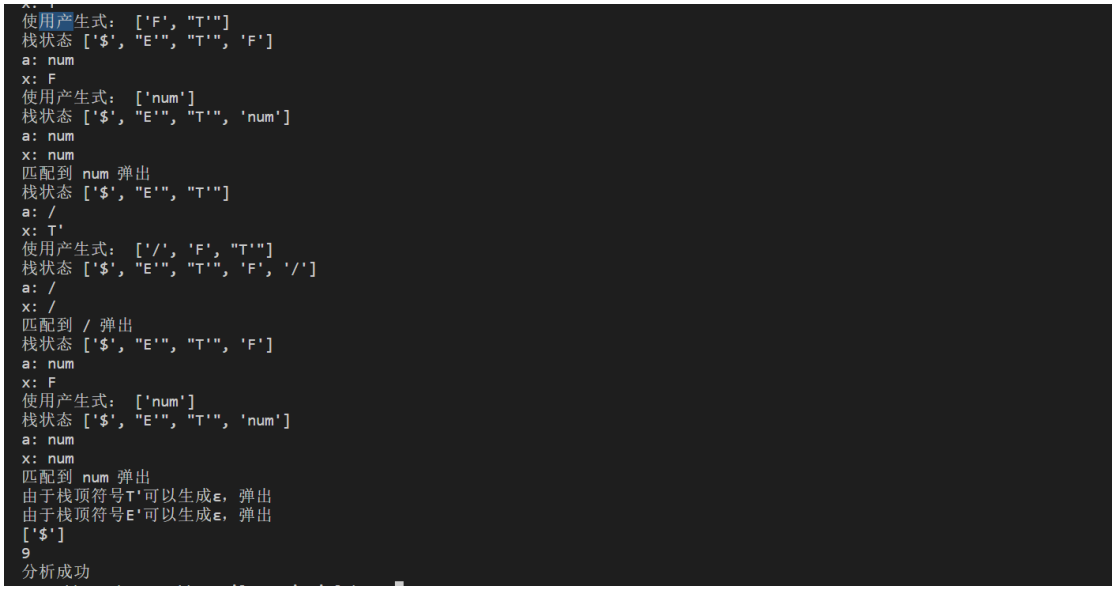
```
使用产生式： ['*', 'F', "T'"]
栈状态 ['$', "E'", "T'", 'F', '*']
a: *
x: *
匹配到 * 弹出
栈状态 ['$', "E'", "T'", 'F']
a: num
x: F
使用产生式： ['num']
栈状态 ['$', "E'", "T'", 'num']
a: num
x: num
匹配到 num 弹出
由于栈顶符号 T'可以生成 ε，弹出
由于栈顶符号 E'可以生成 ε，弹出
['$']
5
分析成功
```

6.3 **测试用例** 2

输入：num * num + num - num / num
执行过程截图：

6-4 测试用例 2 执行过程 1

6-5 测试用例 2 执行过程 2



6-6 测试用例 3 执行过程 3

执行过程文字描述：

Python
请输入待分析的串，每个符号用空格分隔：num * num + num - num / num 栈状态 ['$', 'E'] a: num x: E 使用产生式： ['T', "E'"] 栈状态 ['$', "E'", 'T'] a: num

```
x: T
使用产生式： ['F', "T'"]
栈状态 ['$', "E'", "T'", 'F']
a: num
x: F
使用产生式： ['num']
栈状态 ['$', "E'", "T'", 'num']
a: num
x: num
匹配到 num 弹出
栈状态 ['$', "E'", "T'"]
a: *
x: T'
使用产生式： ['*', 'F', "T'"]
栈状态 ['$', "E'", "T'", 'F', '*']
a: *
x: *
匹配到 * 弹出
栈状态 ['$', "E'", "T'", 'F']
a: num
x: F
使用产生式： ['num']
栈状态 ['$', "E'", "T'", 'num']
a: num
x: num
匹配到 num 弹出
栈状态 ['$', "E'", "T'"]
a: +
x: T'
使用产生式： ['ε']
栈状态 ['$', "E'", 'ε']
a: +
x: ε
栈状态 ['$', "E'"]
a: +
x: E'
使用产生式： ['+', 'T', "E'"]
栈状态 ['$', "E'", 'T', '+']
a: +
x: +
匹配到 + 弹出
栈状态 ['$', "E'", 'T']
```

```
a: num
x: T
使用产生式： ['F', "T'"]
栈状态 ['$', "E'", "T'", 'F']
a: num
x: F
使用产生式： ['num']
栈状态 ['$', "E'", "T'", 'num']
a: num
x: num
匹配到 num 弹出
栈状态 ['$', "E'", "T'"]
a: -
x: T'
使用产生式： ['ε']
栈状态 ['$', "E'", 'ε']
a: -
x: ε
栈状态 ['$', "E'"]
a: -
x: E'
使用产生式： ['-', 'T', "E'"]
栈状态 ['$', "E'", 'T', '-']
a: -
x: -
匹配到 - 弹出
栈状态 ['$', "E'", 'T']
a: num
x: T
使用产生式： ['F', "T'"]
栈状态 ['$', "E'", "T'", 'F']
a: num
x: F
使用产生式： ['num']
栈状态 ['$', "E'", "T'", 'num']
a: num
x: num
匹配到 num 弹出
栈状态 ['$', "E'", "T'"]
a: /
x: T'
使用产生式： ['/', 'F', "T'"]
```

```
栈状态 ['$', "E'", "T'", 'F', '/']
a: /
x: /
匹配到 / 弹出
栈状态 ['$', "E'", "T'", 'F']
a: num
x: F
使用产生式： ['num']
栈状态 ['$', "E'", "T'", 'num']
a: num
x: num
匹配到 num 弹出
由于栈顶符号 T'可以生成 ε，弹出
由于栈顶符号 E'可以生成 ε，弹出
['$']
9
分析成功
```

6.4 **测试用例** 3

输入：num num num num num　　结果：失败

6-7 测试用例 3 执行结果

执行结果文字描述

结果
请输入待分析的串，每个符号用空格分隔: num num num num num 栈状态 ['$', 'E'] a: num x: E

```
使用产生式： ['T', "E'"]
栈状态 ['$', "E'", 'T']
a: num
x: T
使用产生式： ['F', "T'"]
栈状态 ['$', "E'", "T'", 'F']
a: num
x: F
使用产生式： ['num']
栈状态 ['$', "E'", "T'", 'num']
a: num
x: num
匹配到 num 弹出
栈状态 ['$', "E'", "T'"]
a: num
x: T'
使用产生式： ['num']
栈状态 ['$', "E'", 'num']
a: num
x: num
匹配到 num 弹出
栈状态 ['$', "E'"]
a: num
x: E'
使用产生式： ['num']
栈状态 ['$', 'num']
a: num
x: num
匹配到 num 弹出
['$']
3
分析失败
```

7 程序源代码

Python

```
def eliminate_left_recursion(N, P):

    new_non_terminals = []
```

```
    new_productions = P.copy()  # 复制原有的产生式字典，以便修改

    for non_terminal in N:
        productions = P[non_terminal]
        alpha_productions = []
        beta_productions = []

        for production in productions:
            if production.startswith(non_terminal):

alpha_productions.append(production[len(non_terminal):])
            else:
                beta_productions.append(production)

        if alpha_productions:
            new_non_terminal = non_terminal + "'"
            new_non_terminals.append(new_non_terminal)

            new_alpha_productions = [p +" "+ new_non_terminal for
p in alpha_productions] + ['ε']
            new_productions[non_terminal]    =    [p    +    "    "
```

```
+new_non_terminal for p in beta_productions]
                new_productions[new_non_terminal]                    =
new_alpha_productions


    N = N.union(new_non_terminals)


    return N, new_productions


def compute_first(N, T, P):
    print(P)

    # 初始化 FIRST 集，将所有非终结符的 FIRST 集初始化为空集
    FIRST = {symbol: set() for symbol in N.union(T).union({'ε'})}

    # 终结符的 FIRST 集就是其自身
    for terminal in T.union({'ε'}):
        FIRST[terminal].add(terminal)

    # 根据文法规则逐步计算 FIRST 集
    changed = True
    while changed:
```

```
        changed = False
        for non_terminal in N:
            for production in P[non_terminal]:
                symbols = production.split()
                for symbol in symbols:
                    if symbol in T.union({"ε"}):
                        # 如果是终结符，直接加入 FIRST 集
                        if symbol not in FIRST[non_terminal]:
                            FIRST[non_terminal].add(symbol)
                            changed = True
                        break
                    else:
                        # 如果是非终结符，加入其 FIRST 集
                        for sym in FIRST[symbol]:
                            if sym != 'ε' and sym not in
FIRST[non_terminal]:
                                FIRST[non_terminal].add(sym)
                                changed = True
                        if 'ε' not in FIRST[symbol]:
                            # 如果该非终结符不包含 ε，不再往后推导
                            break
```

```
                        # 如果所有符号的 FIRST 集都包含 ε，那么 ε 也加入
FIRST 集
                        if all(sym == 'ε' or sym in FIRST[sym] for sym in
production):
                            if 'ε' not in FIRST[non_terminal]:
                                FIRST[non_terminal].add('ε')
                                changed = True

    return FIRST

def compute_follow(N, P, S, first):
    # 初始化 FOLLOW 集，将所有非终结符的 FOLLOW 集初始化为空集
    FOLLOW = {non_terminal: set() for non_terminal in N}

    # 设定起始符号 S 的 FOLLOW 集为$
    FOLLOW[S].add('$')
    for i in range(2):
        for non_terminal in N:
            for production in P[non_terminal]:
                symbols = production.split()
                for i, symbol in enumerate(symbols):
```

```
                        if symbol not in N:
                            continue
                        list = symbols[i+1:]
                        if is_list_epsilon(list, N, first):
                            FOLLOW[symbol].update(cal_list_first(list, N,
first))

FOLLOW[symbol].update(FOLLOW[non_terminal])
                        else:
                            FOLLOW[symbol].update(cal_list_first(list, N,
first))
                        FOLLOW[symbol].discard("ε")

    return FOLLOW



# 判断单个字符可不可以推导为空
def is_epsilon(A, N, first):
    if A == 'ε':
        return True
    if A not in N:
```

```
            return False
        result = False
        for symbol in first[A]:
            if symbol == 'ε':
                result = True
                break
        return result



# 判断字符串可不可以推导成空
def is_list_epsilon(list, N, first):
        result = True
        for item in list:
            if not is_epsilon(item, N, first):
                return False
        return result


# 找出 list 串的 first 集
def cal_list_first(list, N, first):
        FIRST = set()
        for i, item in enumerate(list):
```

```
            if item not in N:
                FIRST = {item}
                break
            FIRST.update(first[item])
            if not is_epsilon(item, N, first):
                break
        return FIRST



def build_predict_map(N, T, P, first, follow):
        predict_map = {}
        for non_terminal in N:
            predict_map[non_terminal] = {}
            for terminal in T.union({"$"}):
                predict_map[non_terminal][terminal] = []

        for non_terminal in N:
            for production in P[non_terminal]:
                symbols = production.split()
                FIRST = cal_list_first(symbols, N, first)
                for a in FIRST:
```

```
                    if a not in T:
                        break
                    predict_map[non_terminal][a].append(production)
                if "ε" in FIRST:
                    for b in follow[non_terminal]:

predict_map[non_terminal][b].append(production)

    return predict_map



def ll_parser(T, S, predict_map, first):
    stack = ["$", S]      # 初始化栈
    ip = 0                  # 向前指针指向第一个符号
    input_string = input("请输入待分析的串，每个符号用空格分隔:
").split()

    while stack[-1] != "$" and ip < len(input_string):

        print("栈状态",stack)
        x = stack[-1]
```

```
        a = input_string[ip]
        print("a:",a)
        print("x:",x)
        if x in T or x == "$":
            if x == a:
                stack.pop()
                ip += 1
                print("匹配到",a,"弹出")
            else:
                print("Error: 无法匹配输入符号", a)
                break
        elif x == "ε":
            stack.pop()
        else:
            production = predict_map[x][a]
            for proc in production:
                proc_list = proc.split()
            print("使用产生式：", proc_list)
            stack.pop()
            if production != 'ε':
                stack.extend(reversed(proc_list))
```

```
                else:
                    print("Error: 找不到合适的产生式")
                    break

    while len(stack) >= 2 and "ε" in first[stack[-1]]:
        stack.pop()

    print(stack)
    print(ip)
    if stack == ['$'] and ip == len(input_string):
        print("分析成功")
    else:
        print("分析失败")



def print_predict_map(predict_map, N, T):
    # 打印表头（终结符）
    header = ["Non-Terminal"]
    header.extend(T.union({"$"}))
    print("".join(f"{cell:<20}" for cell in header))
```

```
    for non_terminal in N:
        # 打印非终结符行
        row = [non_terminal]
        for terminal in T.union({"$"}):
            productions = predict_map[non_terminal][terminal]
            cell = " | ".join(productions)
            row.append(cell)
        print("".join(f"{cell:<20}" for cell in row))



T = {"(", ")", "+", "-", "*", "/", "num"}
N = {"E", "T", "F"}
P = {
        'E':['E + T', 'E - T', 'T'],
        'T':['T * F', 'T / F', 'F'],
        'F':['( E )','num']
    }
S = 'E'


N, P = eliminate_left_recursion(N, P)
print("修改后的非终结符集合:", N)
```

```
print("修改后的产生式:")
for non_terminal in N:
    print(f"{non_terminal} -> {' | '.join(P[non_terminal])}")



first = compute_first(N, T, P)
print("FIRST 集合:")
for non_terminal in N:
    print(f"FIRST({non_terminal}) = {first[non_terminal]}")


follow = compute_follow(N, P, S, first)
print("FOLLOW 集合:")
for non_terminal in N:
    print(f"FOLLOW({non_terminal}) = {follow[non_terminal]}")


predict_map = build_predict_map(N, T, P, first, follow)
print_predict_map(predict_map, N, T)


ll_parser(T, S, predict_map, first)
```