|  |
| --- |
| 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 3](#_Toc150348431)

[1.1 问题描述 3](#_Toc150348432)

[1.2 实现方法 3](#_Toc150348433)

[2 实验环境 3](#_Toc150348434)

[2.1编辑器选择 3](#_Toc150348435)

[3 文法预处理 4](#_Toc150348436)

[3.1 消除左递归 4](#_Toc150348437)

[3.2 构造FIRST 首符集 4](#_Toc150348438)

[3.3 构造FOLLOW 随附集 4](#_Toc150348439)

[4 文法分析过程 5](#_Toc150348440)

[4.1 构造分析预测表 5](#_Toc150348441)

[4.2 分析程序说明 6](#_Toc150348442)

[5 程序设计 6](#_Toc150348443)

[5.1 变量定义说明 6](#_Toc150348444)

[5.2 功能函数定义说明 7](#_Toc150348445)

[5.2.1 is\_epsilon() 7](#_Toc150348446)

[5.2.2 is\_list\_epsilon()函数 7](#_Toc150348447)

[5.2.3 cal\_list\_first() 8](#_Toc150348448)

[5.3 消除左递归 9](#_Toc150348449)

[5.3.1基本思路 9](#_Toc150348450)

[5.3.2 算法设计 9](#_Toc150348451)

[5.3.3 算法实现 9](#_Toc150348452)

[5.4 构造FIRST集 11](#_Toc150348453)

[5.4.1 基本思路 11](#_Toc150348454)

[5.4.2 算法设计 11](#_Toc150348455)

[5.4.3 算法实现 12](#_Toc150348456)

[5.5 构造FOLLOW集 14](#_Toc150348457)

[5.5.1 基本思路 14](#_Toc150348458)

[5.5.2 算法设计 14](#_Toc150348459)

[5.5.3 算法实现 15](#_Toc150348460)

[5.6 构造分析表 16](#_Toc150348461)

[5.6.1 基本思路 16](#_Toc150348462)

[5.6.2 算法设计 16](#_Toc150348463)

[5.6.3 算法实现 16](#_Toc150348464)

[5.7 构造分析程序 18](#_Toc150348465)

[5.7.1 基本思路 18](#_Toc150348466)

[5.7.2 算法设计 18](#_Toc150348467)

[5.7.3 算法实现 19](#_Toc150348468)

[6 测试报告 21](#_Toc150348469)

[6.1 运行程序，得到新文法及FIRST、FOLLOW集合和预测分析表 21](#_Toc150348470)

[6.2 测试用例1 22](#_Toc150348471)

[6.3 测试用例2 24](#_Toc150348472)

[6.4 测试用例3 28](#_Toc150348473)

[7 程序源代码 29](#_Toc150348474)

# 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。

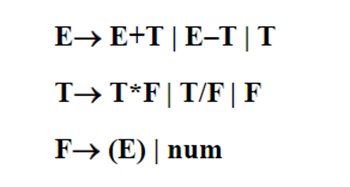


图1-1 文法示意图

## 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→ɛ也是一个产生式，则把ɛ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ɛ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ɛ (即Y1…Y(i-1) =>(\*)ɛ)，则把 FIRST(Yj)中的所有非ɛ元素都加到FIRST(X)中；特别是，若所有的FIRST(Yj , j=1,2,…,K)均含有ɛ，则把 ɛ 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ɛ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=>(\*)ɛ (即 ɛ∈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

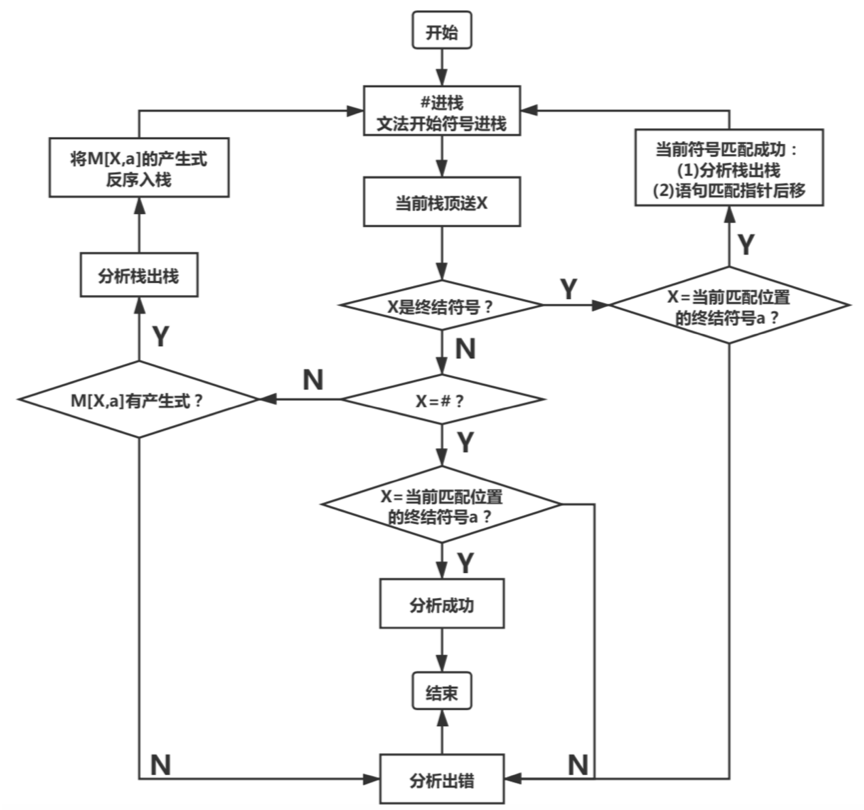


图4-1 LL(1)分析程序流程说明图

# 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→ɛ也是一个产生式，则把ɛ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ɛ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ɛ (即Y1…Y(i-1) =>(\*)ɛ)，则把 FIRST(Yj)中的所有非ɛ元素都加到FIRST(X)中；特别是，若所有的FIRST(Yj , j=1,2,…,K)均含有ɛ，则把 ɛ 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ɛ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=>(\*)ɛ (即 ɛ∈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

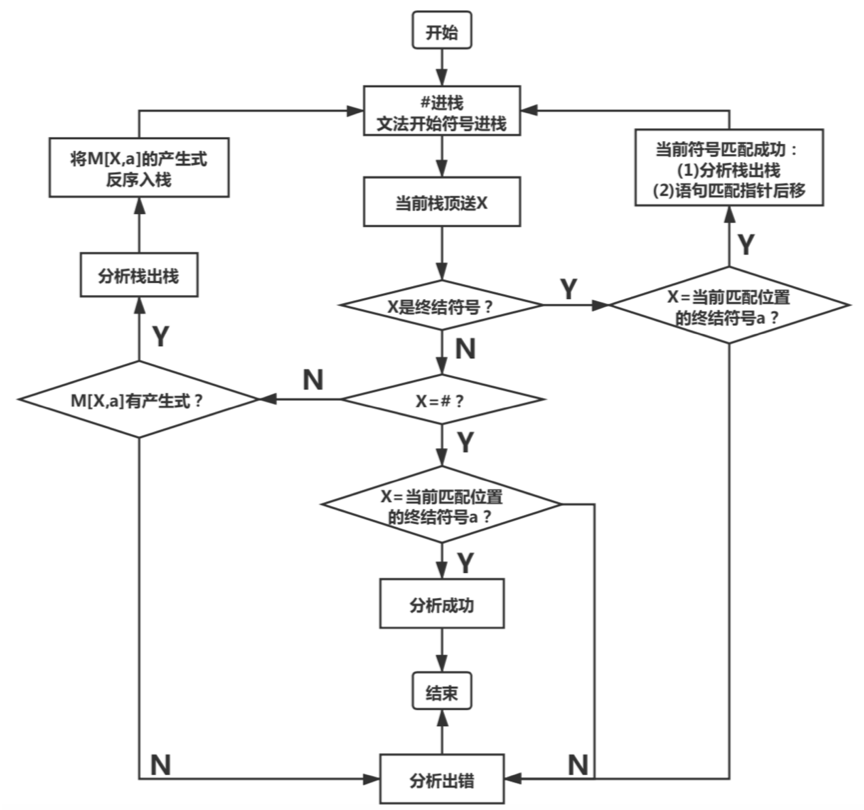


图5-1 LL(1)程序思路

### 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：

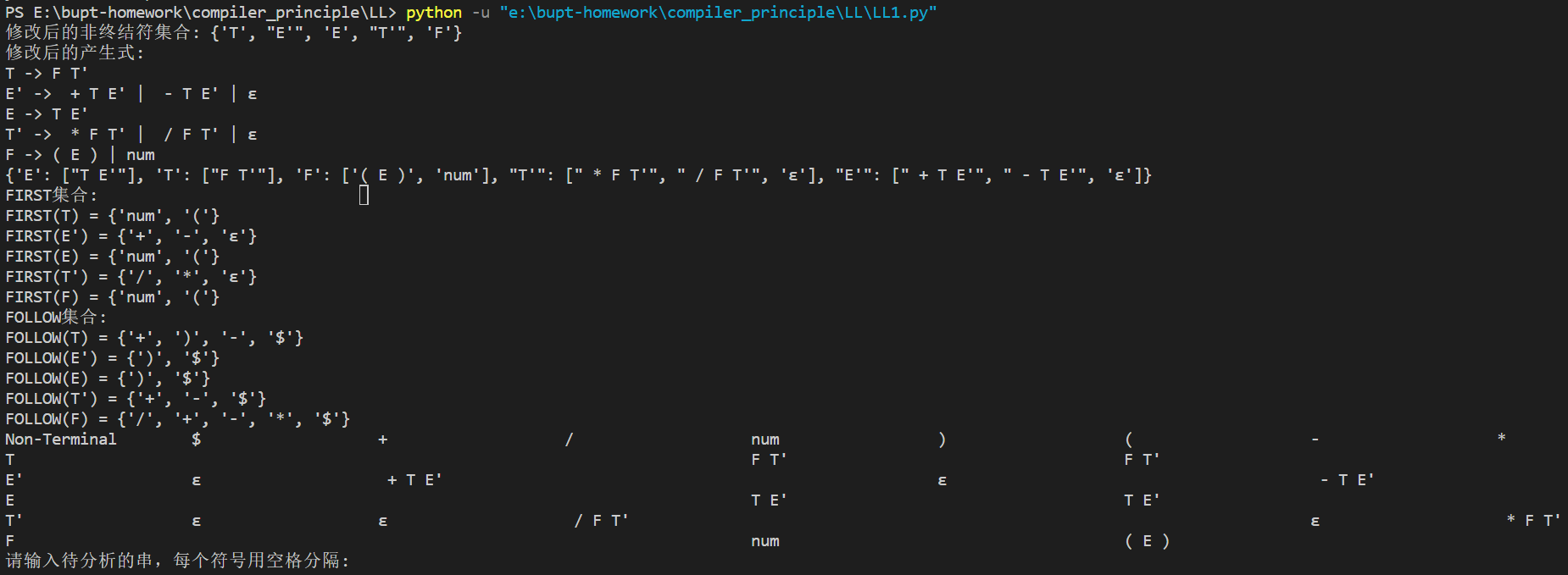


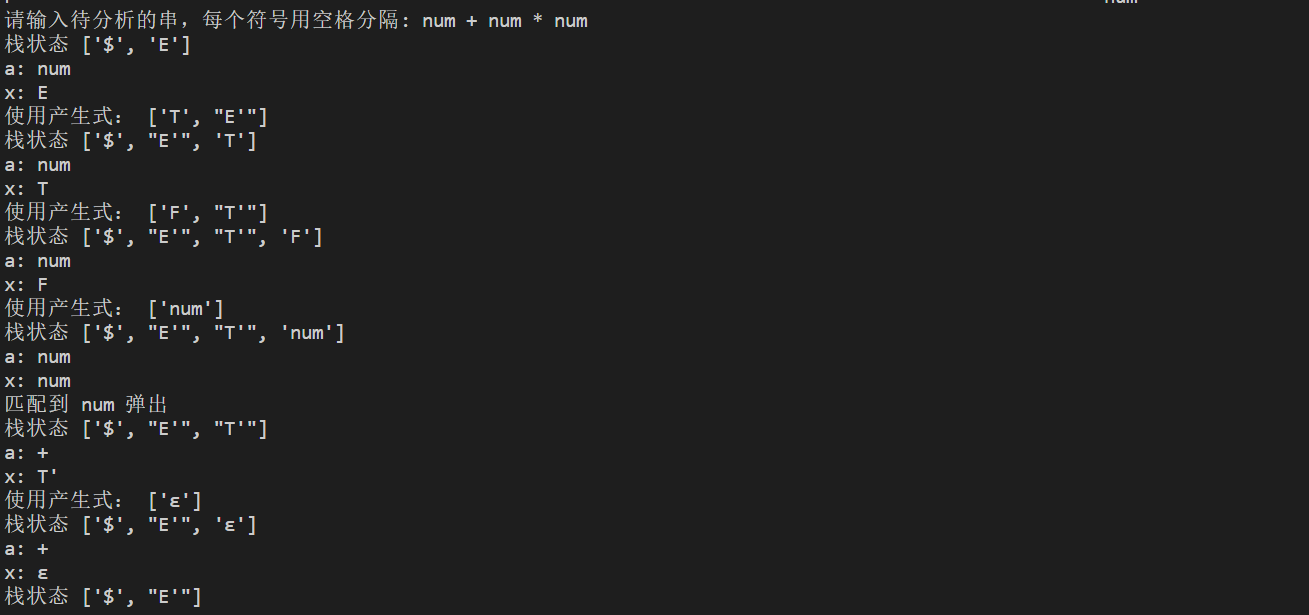
图6-1 程序初始化阶段

经过核验，初始化结果均没出错。

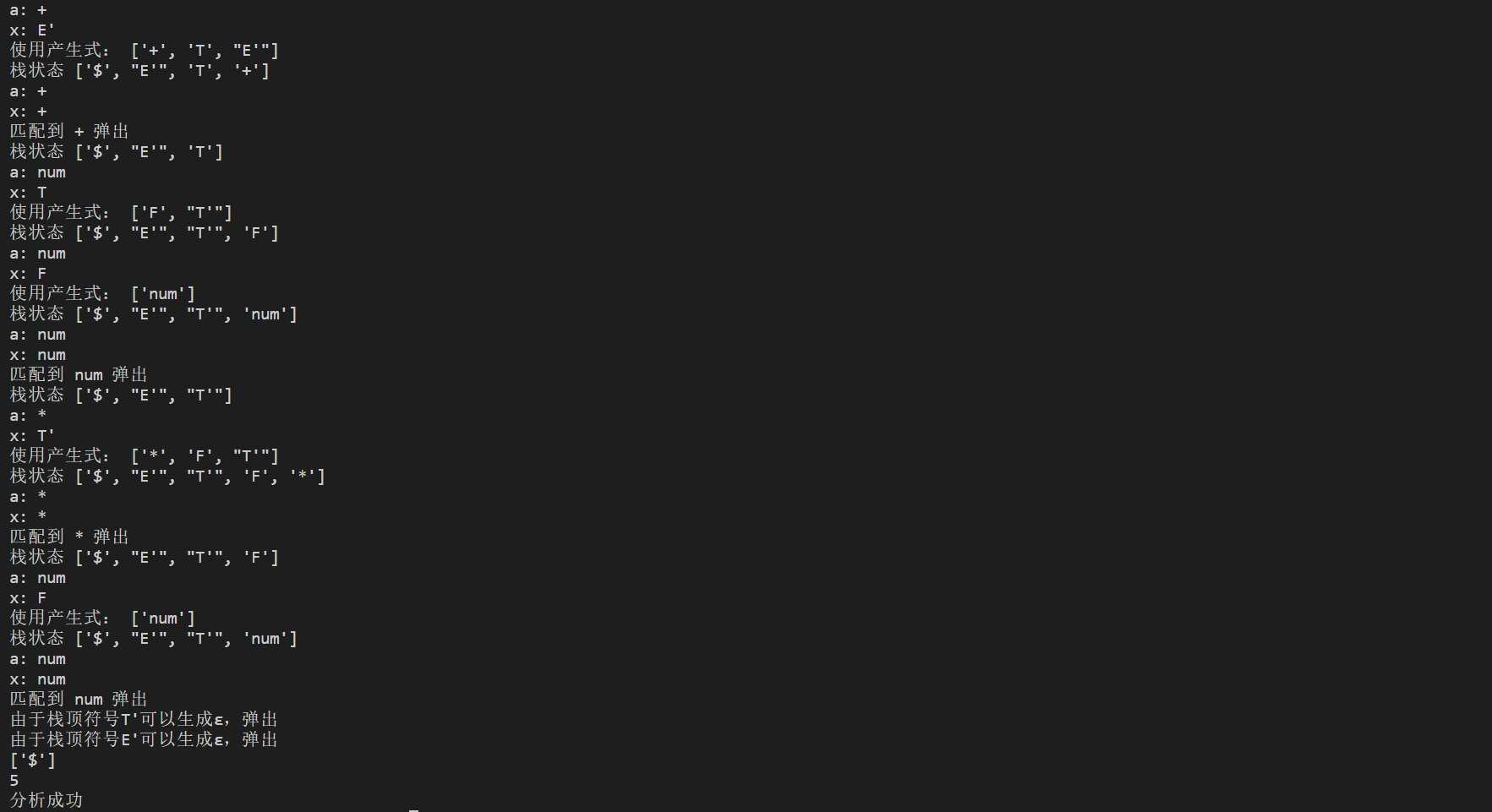
## 6.2 测试用例1

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。

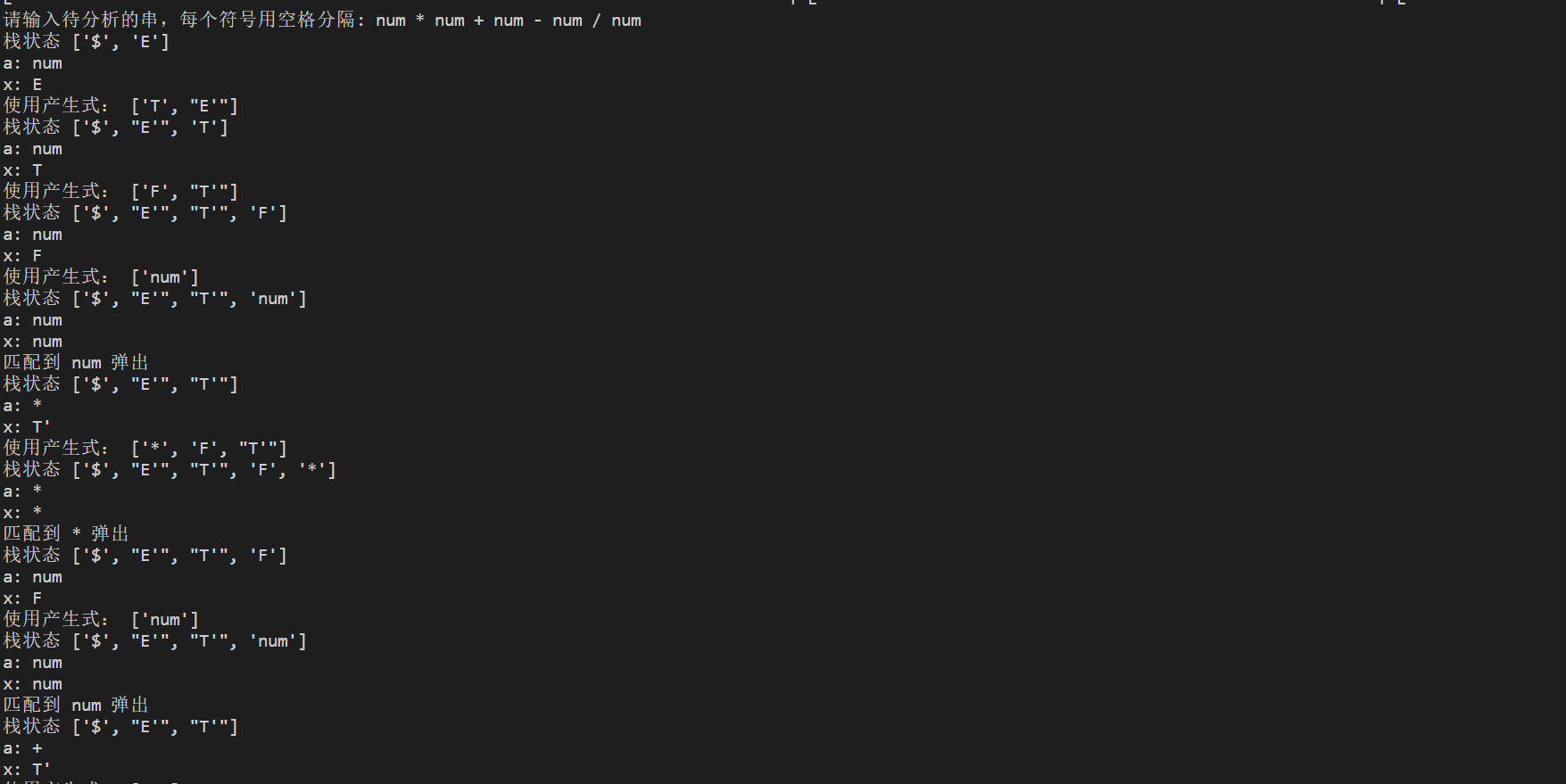
执行过程文字：

|  |
| --- |
| 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

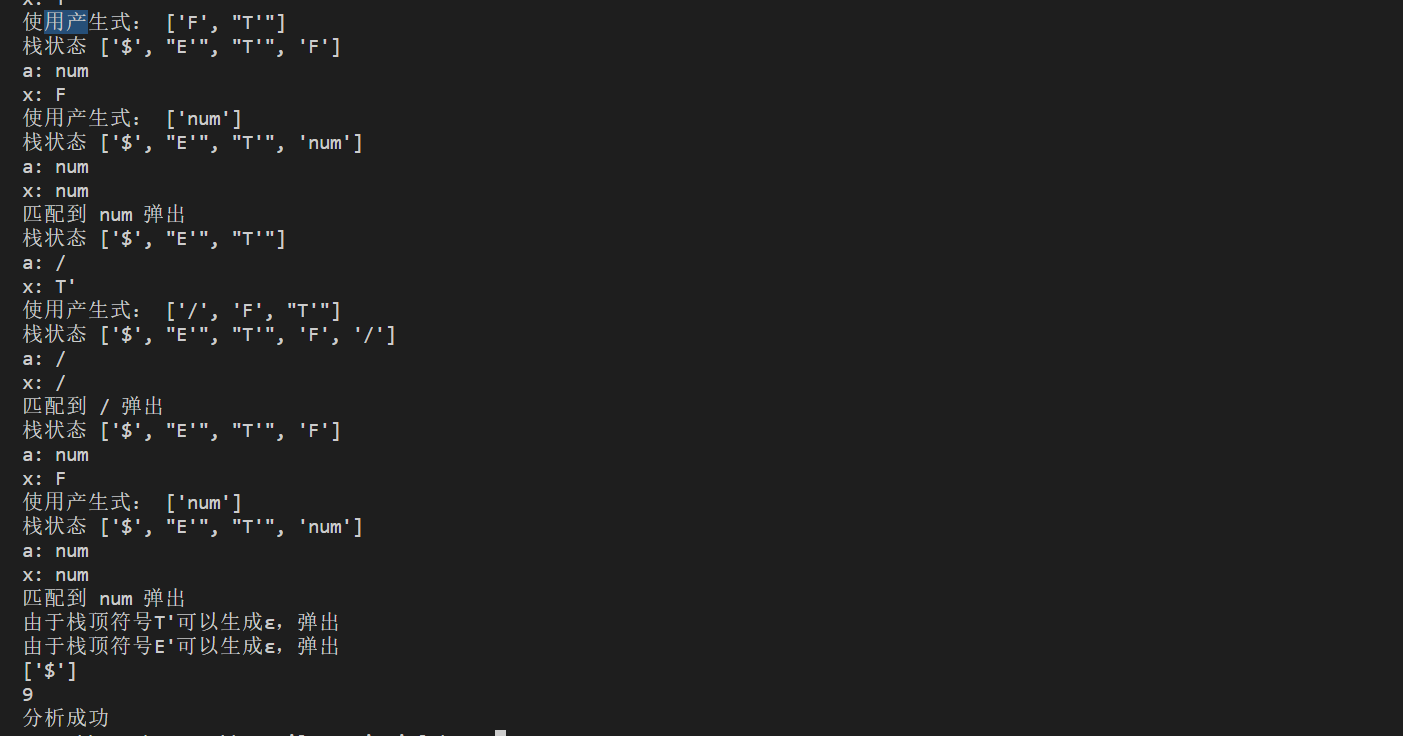
执行过程截图：



6-4 测试用例2 执行过程1



6-5 测试用例2执行过程2



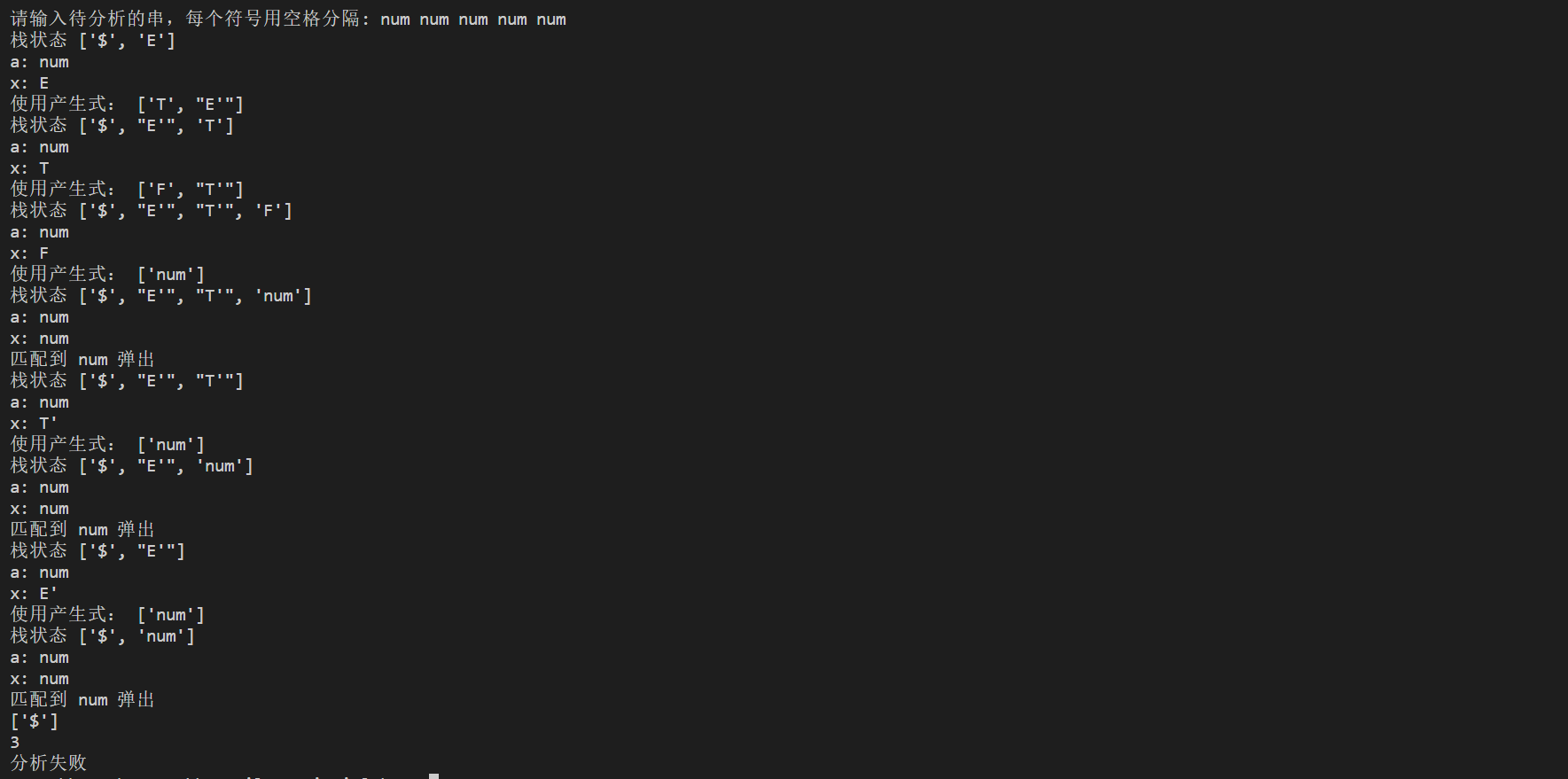
6-6 测试用例3 执行过程3

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) |