Tejas Tripathi

Finding Follow

Aim:

Find Follow Set of given grammar:

Input: E->TX

X->+TX| ε

T->FY

Y->*FY| ε

F->(E) | z

Expected Output: Follow (E)= [$,)]

Follow (X)= [$,)]

Follow (T)= [+,$,)]

Follow (Y)= [+,$,)]

Follow (F)= [*,+,$,)]

Theory/Logic:

FOLLOW: FOLLOW is used only if the current non-terminal can derive ε; then we're interested in what could have followed it in a sentential form. (NB: A string can derive ε if and only if ε is in its FIRST set.)

1. If S is the start symbol, then put $ into FOLLOW(S)

2. Examine all rules of the form A→αXβ; then

1. FIRST (β) is in FOLLOW(X)

2. If β can derive the empty string then put FOLLOW (A) into FOLLOW(X)

The $ is a symbol which is used to mark the end of the input tape. FOLLOW can be applied to a single non-terminal only, and returns a set of terminals.

Thus, if X is the current non-terminal, a is the next symbol on the input, and we have a production rule for X which allows it to derive ϵ, then we apply this rule only if a is in the FOLLOW set for X. FIRST and FOLLOW help us to pick a rule when we have a choice between two or more right hand side by predicting the first symbol that each right hand side can derive. Even if there is only one right hand side we can still use them to tell us whether or not we have an error - if the current

input symbol cannot be derived from the only right hand side available, then we know immediately that the sentence does not belong to the grammar, without having to (attempt to) finish the parse.

Why FOLLOW?

The parser faces one more problem. Let us consider below grammar to understand this problem. A→aBb

B→c|ε

And suppose the input string is "ab" to parse.

As the first character in the input is a, the parser applies the rule A→aBb.

```
A
/ | \
a B b
```

Now the parser checks for the second character of the input string which is b, and the Non-Terminal to derive is B, but the parser can't get any string derivable from B that contains b as first character. But the Grammar does contain a production rule B→ε, if that is applied then B will vanish, and the parser gets the input "ab", as shown below. But the parser can apply it only when it knows that the character that follows B is same as the current character in the input.

In RHS of A→aBb, b follows Non-Terminal B, i.e. FOLLOW (B) = {b}, and the current input character read is also b. Hence the parser applies this rule. And it is able to get the string "ab" from the given grammar.

```
A A
/ | \ / \
a B b => a b
 |
 ε
```

So FOLLOW can make a Non-terminal to vanish out if needed to generate the string from the parse tree. The conclusions is, we need to find FIRST and FOLLOW sets for a given grammar, so that the parser can properly apply the needed rule at the correct position.

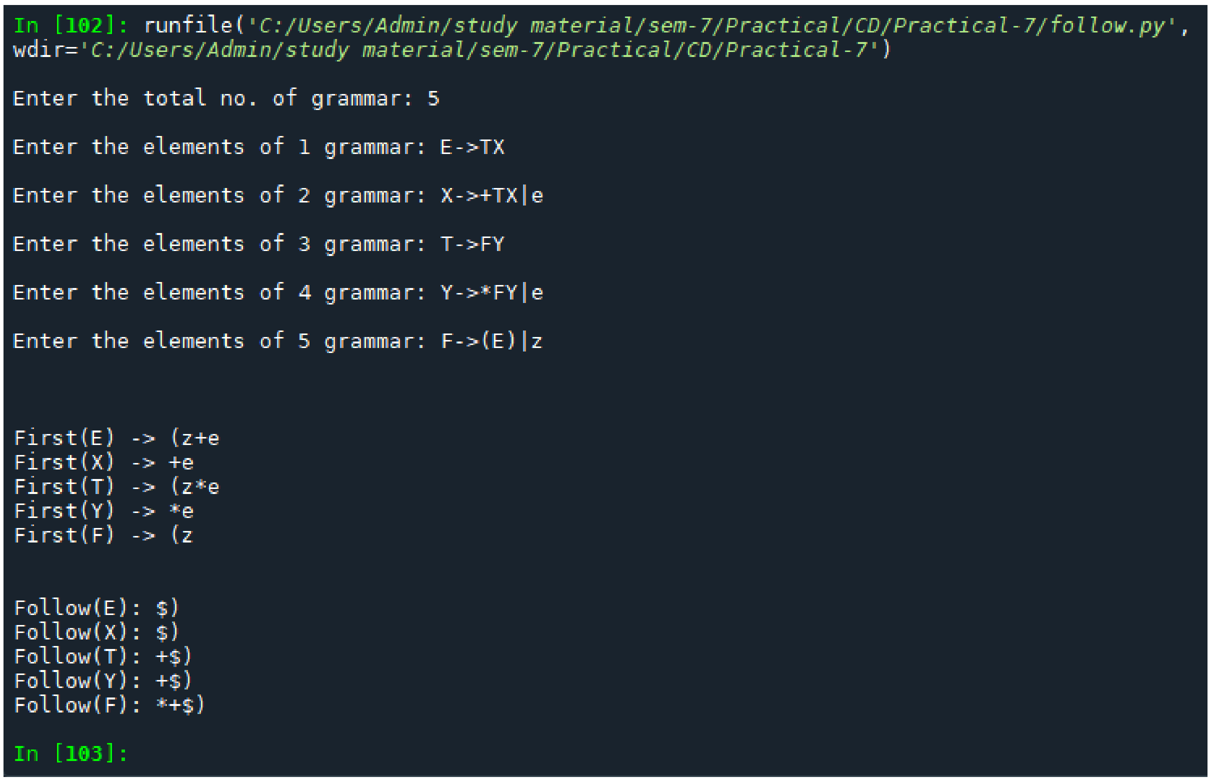
CODE:

```
fterminals=[]

diction={}


def pterminals(chars):

   global fterminals

   fterminals.append(chars)


def getterminal(cha):

   global diction

   global non_terminals

   att=''

   a=diction[cha]

   if a[0] in non_terminals:

      return getterminal(a[0])

   else:

      if '|' in a:

         ind1=a.index('|')

         att=a[0]+a[ind1+1:]

      else:

         att=a[0]

      return att
```

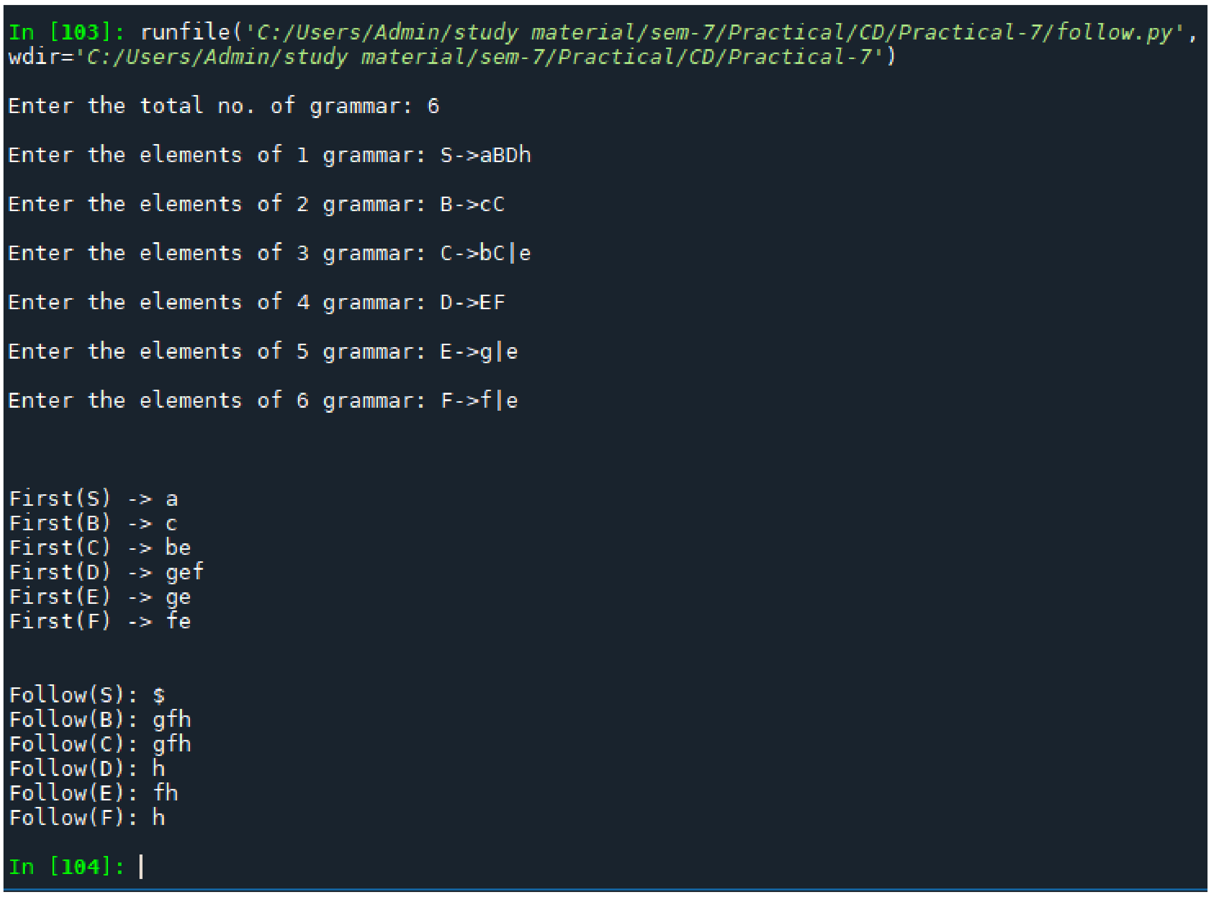
```
def firstof(gra):
    global diction
    global non_terminals
    # gra=['E->TX', 'X->+TX|e', 'T->FY', 'Y->*FY|e', 'F->(E)|z']
    non_terminals=[]
    for i in gra:
        temp=i[0]
        non_terminals.append(temp)
        temp=''

    diction={}

    for i in range(len(gra)):
        diction[non_terminals[i]]=gra[i][3:]

    tstr=''
    for i in range(len(gra)):
        if gra[i][3] not in non_terminals:
            tstr=gra[i][3]
            if '|' in gra[i]:
                ind=gra[i].index('|')
                tstr+=gra[i][ind+1:]
                pterminals(tstr)
            else:
                pterminals(tstr)
            else:
                if len(gra[i]) >= 4:
                    if gra[i][4] in non_terminals:
                        aa=getterminal(gra[i][3])
                        aa+=getterminal(gra[i][4])
                        aaa=''
                        for lst in aa:
                            if lst not in aaa:
                                aaa+=lst
                        pterminals(aaa)
                    else:
                        aa=getterminal(gra[i][3])
                        pterminals(aa)

    for i in range(len(fterminals)):
        print(f'First({non_terminals[i]}) -> {fterminals[i]}')

    return fterminals, non_terminals;

t=int(input("Enter the total no. of grammar: "))
gra=[]
temp=''
for i in range(t):
    temp=input(f"Enter the elements of {i+1} grammar: ")
    gra.append(temp)
    temp=''
print('\n\n')

# gra=['E->TX', 'X->+TX|e', 'T->FY', 'Y->*FY|e', 'F->(E)|z']
# gra=['S->aBDh', 'B->cC', 'C->bC|e', 'D->EF', 'E->g|e', 'F->f|e']
rhs=[]
strin=''
first=[]
firsto={}
terminals=[]
terminal=[]
follow={}
ffirst, non_term=firstof(gra)
for i in range(len(non_term)):
    follow[non_term[i]]=''
follow[non_term[0]]+='$'

for i in ffirst:
    stri2=''
    if 'e' in i:
        for j in i:
            if j!='e':
                stri2+=j
        stri2+='e'
    else:
        stri2=i
    first.append(stri2)

for i in range(len(non_term)):
    firsto[non_term[i]]=first[i]



for i in gra:
    strin=i[3:]
    rhs.append(strin)

for i in rhs:
    for j in i:
        if j not in non_term:
            terminals.append(j)

for i in terminals:
    if i=='|' or i=='e':
        terminals.remove(i)

for i in terminals:
    if i not in terminal:
        terminal.append(i)




for i in non_term:
    for j in rhs:


        if i in j:
            gi=j.index(i)
            gii=gi+1
            if gii==len(j):
                inde=rhs.index(j)
                ab=follow[non_term[inde]]
                follow[i]+=ab
            elif j[gii] in terminal:
                follow[i]+=j[gii]
            elif j[gii] in non_term:
                this=firsto[j[gii]]
                if 'e' in this:
                    if gii==len(j)-1:
                        indii=rhs.index(j)
                        this+=follow[non_term[indii]]
                    elif j[gii+1] in terminal:
                        this+=j[gii+1]
                    elif j[gii+1] in non_term:
                        this+=firsto[j[gii+1]]

                follow[i]+=this



for i in non_term:
    tstri1=''
    if 'e' in follow[i]:
        for j in follow[i]:
            if j!='e':
                tstri1+=j
        follow[i]=tstri1
for i in range(len(non_term)):
    st2=''
    st1=follow[non_term[i]]
    for j in st1:
        if j not in st2:
            st2+=j
    follow[non_term[i]]=st2

print('\n')
for i in non_term:
    print(f'Follow({i}): {follow[i]}')
```

Tejas Tripathi

OUTPUT:



```
In [102]: runfile('C:/Users/Admin/study material/sem-7/Practical/CD/Practical-7/follow.py',
wdir='C:/Users/Admin/study material/sem-7/Practical/CD/Practical-7')

Enter the total no. of grammar: 5

Enter the elements of 1 grammar: E->TX

Enter the elements of 2 grammar: X->+TX|e

Enter the elements of 3 grammar: T->FY

Enter the elements of 4 grammar: Y->*FY|e

Enter the elements of 5 grammar: F->(E)|z


First(E) -> (z+e
First(X) -> +e
First(T) -> (z*e
First(Y) -> *e
First(F) -> (z


Follow(E): $)
Follow(X): $)
Follow(T): +$)
Follow(Y): +$)
Follow(F): *+$)

In [103]:
```



```
In [103]: runfile('C:/Users/Admin/study material/sem-7/Practical/CD/Practical-7/follow.py',
wdir='C:/Users/Admin/study material/sem-7/Practical/CD/Practical-7')

Enter the total no. of grammar: 6

Enter the elements of 1 grammar: S->aBDh

Enter the elements of 2 grammar: B->cC

Enter the elements of 3 grammar: C->bC|e

Enter the elements of 4 grammar: D->EF

Enter the elements of 5 grammar: E->g|e

Enter the elements of 6 grammar: F->f|e


First(S) -> a
First(B) -> c
First(C) -> be
First(D) -> gef
First(E) -> ge
First(F) -> fe


Follow(S): $
Follow(B): gfh
Follow(C): gfh
Follow(D): h
Follow(E): fh
Follow(F): h

In [104]: |
```