AIM:

To compute first of the non-terminals.

ALGORITHM:

- Get the number of terminals, terminals, number of non-terminals, non-terminals from the user. Then, get the number of productions and the productions from the user.
- Create a production dictionary with the non-terminals as the keys and add the right side of the production to the dictionary after splitting it at '/'.
- Create an empty dictionary first to store the first of the non-terminals.
- Create a function first to compute the first of the non-terminals.
- In the function, iterate through the productions.
- We create an empty set for first. If the production is a terminal or empty or has an epsilon in it then add it to the set.
- If it's a non-terminal compute the first of the non-terminal. If there is an epsilon in it add the first of that non terminal and remove the epsilon.
- If the rest of the string is in terminals, then add it into first and break out of the loop. Else if it's the end of the production, add epsilon to the first.
- Else, compute the first of the rest of the string and add it to the set, first.
- Else, take union between the first of the non-terminal and the recently computed first.
- Return first.
- Finally print the first of all the non-terminals.

AIM:

To compute follow of the non-terminals.

ALGORITHM:

- Get the number of terminals, terminals, number of non-terminals, non-terminals from the user. Then, get the number of productions and the productions from the user.
- Create a production dictionary with the non-terminals as the keys and add the right side of the production to the dictionary after splitting it at '/'.

- Create an empty dictionary first to store the first of the non-terminals.
- Create a function first to compute the first of the non-terminals.
- In the function, iterate through the productions.
- We create an empty set for first. If the production is a terminal or empty or has an epsilon in it then add it to the set.
- If it's a non-terminal compute the first of the non-terminal. If there is an epsilon in it add the first of that non terminal and remove the epsilon.
- If the rest of the string is in terminals, then add it into first and break out of the loop. Else if it's the end of the production, add epsilon to the first.
- Else, compute the first of the rest of the string and add it to the set, first.
- Else, take union between the first of the non-terminal and the recently computed first.
- Return first.
- Next, we create a function follow to compute the follow of the non-terminals.
- We create an empty set for follow.
- We get the productions for the non-terminals and check if the non-terminal is the starting symbol. If yes, then add $ to the follow.
- Iterate through the productions. If the character is equal to the non-terminal which we passed, then get the string which follows the character. If that's the end of the production and the non-terminal is equal to the non-terminal which we passed, then continue the while loop. Else, compute the follow of the non-terminal and add it to the follow.
- Else, compute the first of the rest of the string. If epsilon is present in the first of the string, add that to the follow and remove the epsilon from it. And add the follow of the non-terminal.
- Else, take union of the follow of that non-terminal and first of rest of the string.
- Return follow.
- Finally print the first and follow of the non-terminals.

In [31]:
```
def input_fun():
    x= input("Enter grammar space seperated: ").split(" ")
    n = len(x)
    key=[]
    value=[]
    for i in x:
        ni = i.split('->')
        value.append(ni[1].split("|"))
        key.append(ni[0])
    res = {key[i]: value[i] for i in range(len(key))}
    return res

gram = input_fun()

def removeDirectLR(gramA, A):
    temp = gramA[A]
    tempCr = []
    tempInCr = []
    for i in temp:
        if i[0] == A:
            tempInCr.append(i[1:]+[A+"'"])
        else:
            tempCr.append(i+[A+"'"])
    tempInCr.append(["e"])
    gramA[A] = tempCr
    gramA[A+"'"] = tempInCr
    return gramA


def checkForIndirect(gramA, a, ai):
    if ai not in gramA:
        return False
    if a == ai:
        return True
    for i in gramA[ai]:
        if i[0] == ai:
            return False
        if i[0] in gramA:
            return checkForIndirect(gramA, a, i[0])
    return False

def rep(gramA, A):
    temp = gramA[A]
    newTemp = []
    for i in temp:
        if checkForIndirect(gramA, A, i[0]):
            t = []
            for k in gramA[i[0]]:
                t=[]
                t+=k
                t+=i[1:]
                newTemp.append(t)

        else:
            newTemp.append(i)
    gramA[A] = newTemp
```

```
    return gramA

def rem(gram):
    c = 1
    conv = {}
    gramA = {}
    revconv = {}
    for j in gram:
        conv[j] = "A"+str(c)
        gramA["A"+str(c)] = []
        c+=1

    for i in gram:
        for j in gram[i]:
            temp = []
            for k in j:
                if k in conv:
                    temp.append(conv[k])
                else:
                    temp.append(k)
            gramA[conv[i]].append(temp)


    #print(gramA)
    for i in range(c-1,0,-1):
        ai = "A"+str(i)
        for j in range(0,i):
            aj = gramA[ai][0][0]
            if ai!=aj :
                if aj in gramA and checkForIndirect(gramA,ai,aj):
                    gramA = rep(gramA, ai)

    for i in range(1,c):
        ai = "A"+str(i)
        for j in gramA[ai]:
            if ai==j[0]:
                gramA = removeDirectLR(gramA, ai)
                break

    op = {}
    for i in gramA:
        a = str(i)
        for j in conv:
            a = a.replace(conv[j],j)
        revconv[i] = a

    for i in gramA:
        l = []
        for j in gramA[i]:
            k = []
            for m in j:
                if m in revconv:
                    k.append(m.replace(m,revconv[m]))
                else:
                    k.append(m)
            l.append(k)
        op[revconv[i]] = l
```

```
    return op

result = rem(gram)


def first(gram, term):
    a = []
    if term not in gram:
        return [term]
    for i in gram[term]:
        if i[0] not in gram:
            a.append(i[0])
        elif i[0] in gram:
            a += first(gram, i[0])
    return a

firsts = {}
for i in result:
    firsts[i] = first(result,i)
    print(f'First({i}):',*firsts[i])


def follow(gram, term):
    a = []
    for rule in gram:
        for i in gram[rule]:
            if term in i:
                temp = i
                indx = i.index(term)
                if indx+1!=len(i):
                    if i[-1] in firsts:
                        a+=firsts[i[-1]]
                    else:
                        a+=[i[-1]]
                else:
                    a+=["e"]
                if rule != term and "e" in a:
                    a+= follow(gram,rule)
    return a

follows = {}
for i in result:
    follows[i] = list(set(follow(result,i)))
    if "e" in follows[i]:
        follows[i].pop(follows[i].index("e"))
    follows[i]+=["$"]
    print(f'Follow({i}):',*follows[i])
```

```
Enter grammar space seperated: E->E+T|T T->T*F|F F->(E)|i
First(E): ( i
First(T): ( i
First(F): ( i
First(E'): + e
First(T'): * e
Follow(E): ) $
Follow(T): + ) $
Follow(F): * ) + $
Follow(E'): ) $
Follow(T'): ) + $
```

In [ ]:
```
E->E+T|T T->T*F|F F->(E)|i
```