**Practical 7:**

**Aim: - Write a program to illustrate the generation of OPM for a given grammar.**

**Theory: -**

Operator precedence grammar is kinds of shift reduce parsing method. It is applied to a small class of operator grammars.

A grammar is said to be operator precedence grammar if it has two properties:

* No R.H.S. of any production has a∈.
* No two non-terminals are adjacent.

Operator precedence can only established between the terminals of the grammar. It ignores the non-terminal.

**There are the three operator precedence relations:**

a ⋗ b means that terminal "a" has the higher precedence than terminal "b".

a ⋖ b means that terminal "a" has the lower precedence than terminal "b".

a ≐ b means that the terminal "a" and "b" both have same precedence.

**We first calculate leading and trailing sets for the given grammer:**

**LEADING**

If production is of form A → aα or A → Ba α where B is Non-terminal, and α can be any string, then the first terminal symbol on R.H.S is

**Leading(A) = {a}**

If production is of form A → Bα, if a is in LEADING (B), then a will also be in LEADING (A).

**TRAILING**

If production is of form  A→ αa or A → αaB where B is Non-terminal, and α can be any string then,

**TRAILING (A) = {a}**

If production is of form  A → αB. If a is in TRAILING (B), then a will be in TRAILING (A).

***Algorithms:-***

**LEADING**

* begin
* For each non-terminal A and terminal a

                           L [A, a] = false ;

* For each production of form A ⟶ aα or A → B a α

                            Install (A, a) ;

* While the stack not empty

                            Pop top pair (B, a) form Stack ;

                             For each production of form A → B α

                             Install (A, a);

* end

**TRAILING**

* begin
* For each non-terminal A and terminal a

                 T [A, a] = false ;

* For each production of form A ⟶ αa or A → α a B

                  Install (A, a) ;

* While the stack not empty

                Pop top pair (B, a) form Stack ;

                For each production of form A → αB

                Install (A, a);

* End

**Procedure Install (A, a)**

* begin
* If not T [A, a] then

                 T [A, a] = true

                 push (A, a) onto stack.

* End

**Operator Precedence Relations**

* begin
* For each production A → B1, B2, … … … . Bn

                       for i = 1 to n – 1

              If Bi and Bi+1 are both terminals then

                      set Bi = Bi+1

             If i ≤ n − 2 and Bi and Bi+2are both terminals and Bi+1 is non-terminal then

                      set Bi = Bi+1

             If Biis terminal & Bi+1is non-terminal then for all a in LEADING (Bi+1)

                       set Bi <. a

            If Biis non-terminal & Bi+1 is terminal then for all a in TRAILING (Bi)

                      set a . > Bi+1

* end

**Code**:-

a = ["E=E+T","E=T","T=T\*F","T=F","F=(E)","F=i"]

rules = {}

terms = []

for i in a:

temp = i.split("=")

terms.append(temp[0])

try:

rules[temp[0]] += [temp[1]]

except:

rules[temp[0]] = [temp[1]]

terms = list(set(terms))

#========================================================#

x = list(rules.values())

prod\_rules = []

for i in x:

for j in i:

prod\_rules.append(j)

opr = []

list\_oprs = ["+","-","\*","/","(",")","i"]

for i in prod\_rules:

for x in range(0,len(i)):

if i[x] in list\_oprs:

opr.append(i[x])

opm= []

for i in range(0,len(opr)+1):

x = []

for j in range(0,len(opr)+1):

x.append("0")

opm.append(x)

#========================================================#

def leading(gram, rules, term, start):

s = []

if gram[0] not in terms:

return gram[0]

elif len(gram) == 1:

return [0]

elif gram[1] not in terms and gram[-1] is not start:

for i in rules[gram[-1]]:

s+= leading(i, rules, gram[-1], start)

s+= [gram[1]]

return s

def trailing(gram, rules, term, start):

s = []

if gram[-1] not in terms:

return gram[-1]

elif len(gram) == 1:

return [0]

elif gram[-2] not in terms and gram[-1] is not start:

for i in rules[gram[-1]]:

s+= trailing(i, rules, gram[-1], start)

s+= [gram[-2]]

return s

leads = {}

trails = {}

for i in terms:

s = [0]

for j in rules[i]:

s+=leading(j,rules,i,i)

s = set(s)

s.remove(0)

leads[i] = s

s = [0]

for j in rules[i]:

s+=trailing(j,rules,i,i)

s = set(s)

s.remove(0)

trails[i] = s

for i in terms:

print("LEADING("+i+"):",leads[i])

for i in terms:

print("TRAILING("+i+"):",trails[i])

#========================================================#

print("\nOperator Precedance Matrix")

opr = sorted(opr)

opm[0][0] = "`"

for i in range(1,len(opm)):

opm[0][i] = opr[i-1]

opm[i][0] = opr[i-1]

for i in a:

temp = i.split("=")

cur\_prod = temp[1]

for j in range (0,len(cur\_prod)-1):

if cur\_prod[j] in opr and cur\_prod[j+1] in opr:

opm[opr.index(cur\_prod[j]) +1][opr.index(cur\_prod[j+1])+1] = "="

if j < (len(cur\_prod)-2):

if cur\_prod[j] in opr and cur\_prod[j+2] in opr:

if cur\_prod[j+1] in terms:

opm[opr.index(cur\_prod[j])+1][opr.index(cur\_prod[j+2])+1] = "="

if cur\_prod[j] in opr and cur\_prod[j+1] in terms:

for k in leads[temp[0]]:

opm[opr.index(cur\_prod[j])+1][opr.index(k)+1] = "<"

if cur\_prod[j] in terms and cur\_prod[j+1] in opr:

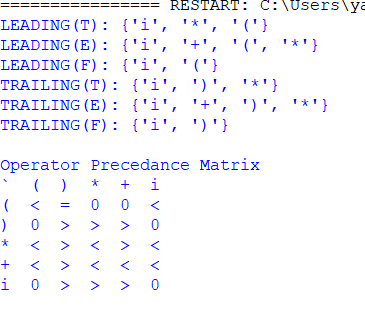
for k in trails[cur\_prod[j]]:

opm[opr.index(k)+1][opr.index(cur\_prod[j+1])+1] = ">"

for i in opm:

print (' '.join(map(str, i)))

**Output:-**



**Conclusion**:-

We successfully constructed the operator precedence matrix for the given grammar.