Experiment no – 07

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 $A \rightarrow B_1, B_2, \ldots \ldots \ldots . B_n$

  for $i = 1$ to $n - 1$

  If B_i and B_{i+1} are both terminals then

  set $B_i = B_{i+1}$

  If $i \leq n - 2$ and B_i and B_{i+2} are both terminals and B_{i+1} is non-terminal then

  set $B_i = B_{i+1}$

  If B_i is terminal & B_{i+1} is non-terminal then for all a in LEADING (B_{i+1})

  set $B_i <. a$

  If B_i is non-terminal & B_{i+1} is terminal then for all a in TRAILING (B_i)

  set $a .> B_{i+1}$

- 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:-

```
================ RESTART: C:\Users\ya
LEADING(T): {'i', '*', '('}
LEADING(E): {'i', '+', '(', '*'}
LEADING(F): {'i', '('}
TRAILING(T): {'i', ')', '*'}
TRAILING(E): {'i', '+', ')', '*'}
TRAILING(F): {'i', ')'}

Operator Precedance Matrix
`   (   )   *   +   i
(   <   =   0   0   <
)   0   >   >   >   0
*   <   >   <   >   <
+   <   >   <   <   <
i   0   >   >   >   0
```

Conclusion:-

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