EXP 6

PREDICTIVE PARSING TABLE

AIM: To write a program for displaying the predictive parsing table of a given production

ALGORITHM:

1. Start the program.
2. Initialize the required variables.
3. Get the number of coordinates and productions from the user.
4. Perform the following

for (each production A → α in G) {

for (each terminal a in FIRST(α))

add A → α to M[A, a];

if (ε is in FIRST(α))

for (each symbol b in FOLLOW(A))

add A → α to M[A, b];

5. Print the resulting stack.
6. Print if the grammar is accepted or not.
7. Exit the program.

CODE:

```
n=int(input('no of NON-TERMINAL productions'))
gram={}
for i in range(n):
   print('Enter the',i+1,' non-terminal')
   key=input()
   print('no of production for non-terminal',key)
   n1=int(input())
   l=[]
   for j in range(n1):
      l.append(input())
   value=l
   gram[key]=value
print(gram)
def removeDirectLR(gramA, A):
        """gramA is dictonary"""
        temp = gramA[A]
        tempCr = []
        tempInCr = []
        for i in temp:
                if i[0] == A:
                        #tempInCr.append(i[1:])
                        tempInCr.append(i[1:]+[A+"'"])
                else:
                        #tempCr.append(i)
                        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)
terminals = []
for i in result:
	for j in result[i]:
		for k in j:
			if k not in result:
				terminals+=[k]
terminals = list(set(terminals))
#print(terminals)

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])

resMod = {}
for i in result:
        l = []
        for j in result[i]:
                temp = ""
                for k in j:
                        temp+=k
                l.append(temp)
        resMod[i] = l

# create predictive parsing table
tterm = list(terminals)
tterm.pop(tterm.index("e"))
tterm+=["$"]
pptable = {}
for i in result:
        for j in tterm:
                if j in firsts[i]:
                        pptable[(i,j)]=resMod[i[0]][0]
                else:
                        pptable[(i,j)]=""
        if "e" in firsts[i]:
```

```
                for j in tterm:
                        if j in follows[i]:
                                pptable[(i,j)]="e"
pptable[("F","i")] = "i"
toprint = f'{"": <10}'


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


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])
for i in tterm:
        toprint+= f'|{i: <10}'
print(toprint)
for i in result:
        toprint = f'{i: <10}'
        for j in tterm:
                if pptable[(i,j)]!="":
                        toprint+=f'|{i+"->"+pptable[(i,j)]: <10}'
                else:
                        toprint+=f'|{pptable[(i,j)]: <10}'
        print(f'{"-":-<76}')
        print(toprint)
```

PRODUCTION:

Construct Predictive Parsing table for the grammar

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid \mathbf{id}$

Step 1 : Eliminate Left Recursion

$E \rightarrow T\ E'$

$E' \rightarrow + T\ E' \mid \varepsilon$

$T \rightarrow F\ T'$

$T' \rightarrow * F\ T' \mid \varepsilon$

$F \rightarrow (E) \mid \mathbf{id}$

Step 2 : Left Factor the grammar

There is no need for left factoring in this grammar as there are no common prefixes

Now the grammar is

$E \rightarrow T\ E'$

$E' \rightarrow + T\ E' \mid \varepsilon$

$T \rightarrow F\ T'$

$T' \rightarrow * F\ T' \mid \varepsilon$

$F \rightarrow (E) \mid \mathbf{id}$

Step 3 : Compute FIRST and FOLLOW for all the non-terminals

FIRST(E) = { (, **id** }

FIRST(T) = { (, **id** }

FIRST(F) = { (, **id** }

FIRST(E') = { +,ε }

FIRST(T') = { *,ε }

FOLLOW(E) = { **$**,) }

FOLLOW(E') = { **$**,) }

FOLLOW(T) = { **+**, **$**,) }

FOLLOW(T') = = { **+**, **$**,) }

FOLLOW(F) = { **+**, **$**,) }

Step 4 : Construct predictive parsing table

The grammar after eliminating left recursion is

$E \rightarrow T\,E'$

$E' \rightarrow +\,T\,E'$

$E' \rightarrow \varepsilon$

$T \rightarrow F\,T'$

$T' \rightarrow *\,F\,T'$

$T' \rightarrow \varepsilon$

$F \rightarrow (\,E\,)$

$F \rightarrow \mathbf{id}$

FIRST(E) = { (, **id** }
FIRST(T) = { (, **id** }
FIRST(F) = { (, **id** }
FIRST(E') = { +, ε }
FIRST(T') = { *, ε }

FOLLOW(E) = { $,) }
FOLLOW(E') = { $,) }
FOLLOW(T) = { +, $,) }
FOLLOW(T') = { +, $,) }
FOLLOW(F) = { +, $,) }

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow$ **id**			$F \rightarrow (E)$		

OUTPUT:

```
no of NON-TERMINAL productions3
Enter the 1  non-terminal
E
no of production for non-terminal E
2
E+T
T
Enter the 2  non-terminal
T
no of production for non-terminal T
2
T*F
F
Enter the 3  non-terminal
F
no of production for non-terminal F
2
(E)
i
{'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'): ['+', ')', '$']
          |+         |i         |)         |*         |(         |$
-------------------------------------------------------------------------
E         |          |E->TE'    |          |          |E->TE'    |
-------------------------------------------------------------------------
T         |          |T->FT'    |          |          |T->FT'    |
-------------------------------------------------------------------------
F         |          |F->i      |          |          |F->(E)    |
-------------------------------------------------------------------------
E'        |E'->TE'   |          |E'->e     |          |          |E'->e
-------------------------------------------------------------------------
T'        |T'->e     |          |T'->e     |T'->FT'   |          |T'->e
```

RESULT:

The program for finding predictive parsing table is successfully compiled and executed.

RA1911003010387

CHITRALEKHA.CH