EXPERIMENT 08

CLASS: TE CMPN A ROLL NO. : 19
NAME: REBECCA DIAS PID: 182027

Aim: To design and implement a program for SLR Parser.

Theory:

Role of a Syntax analyzer

- Takes input a string of tokens form lexical analyzer
- Verifies that string of token names can be generated by the grammar for the source language
- Report syntax errors in an intelligible fashion and recover from commonly occurring errors to continue processing remainder of the program

Compare Top-down Parsing and Bottom-up Parsing

S.No	Top-Down Parsing	Bottom-Up Parsing
1.	It is a parsing strategy that first looks at the highest level of the parse tree and works down the parse tree by using the rules of grammar.	It is a parsing strategy that first looks at the lowest level of the parse tree and works up the parse tree by using the rules of grammar.
2.	Top-down parsing attempts to find the left most derivations for an input string.	Bottom-up parsing can be defined as an attempts to reduce the input string to start symbol of a grammar.
3.	In this parsing technique we start parsing from top (start symbol of parse tree) to down (the leaf node of parse tree) in top-down manner.	In this parsing technique we start parsing from bottom (leaf node of parse tree) to up (the start symbol of parse tree) in bottom-up manner.
4.	This parsing technique uses Left Most Derivation.	This parsing technique uses Right Most Derivation.
5.	It's main decision is to select what production rule to use in order to construct the string.	It's main decision is to select when to use a production rule to reduce the string to get the starting symbol.

Design of SLR parser

The SLR parser is similar to LR(0) parser except that the reduced entry. The reduced productions are written only in the FOLLOW of the variable whose production is reduced.

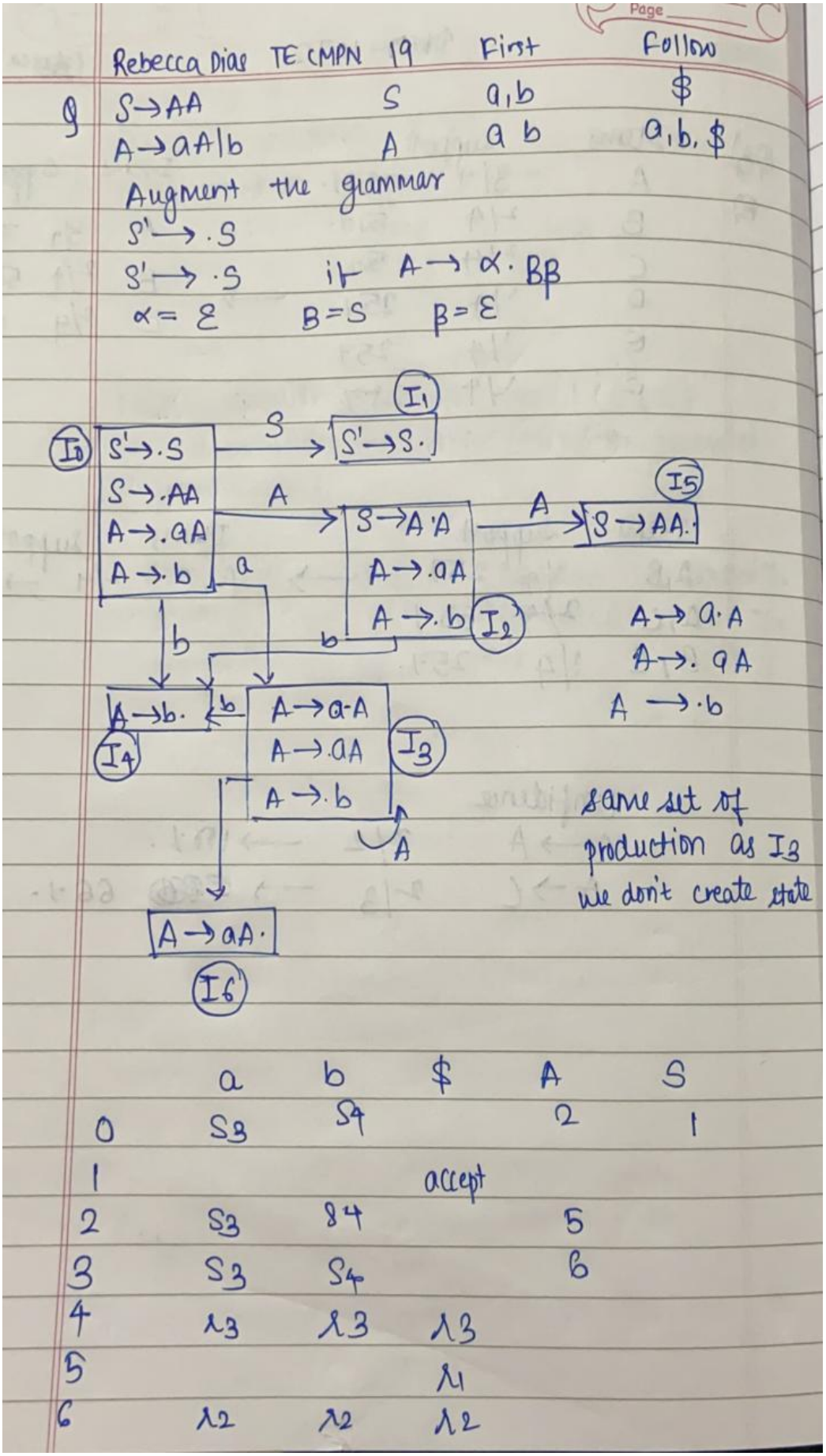
Construction of SLR parsing table –

1. Construct C = { I0, I1, ……. In}, the collection of sets of LR(0) items for G'.
2. State i is constructed from Ii. The parsing actions for state i are determined as follow :

- If [A -> ?.a?] is in Ii and GOTO(Ii , a) = Ij , then set ACTION[i, a] to "shift j". Here a must be terminal.
- If [A -> ?.] is in Ii, then set ACTION[i, a] to "reduce A -> ?" for all a in FOLLOW(A); here A may not be S'.
- Is [S -> S.] is in Ii, then set action[i, $] to "accept". If any conflicting actions are generated by the above rules we say that the grammar is not SLR.

3. The goto transitions for state i are constructed for all nonterminals A using the rule: if GOTO(Ii , A) = Ij then GOTO [i, A] = j.
4. All entries not defined by rules 2 and 3 are made error.

Construct SLR parser for a given grammar

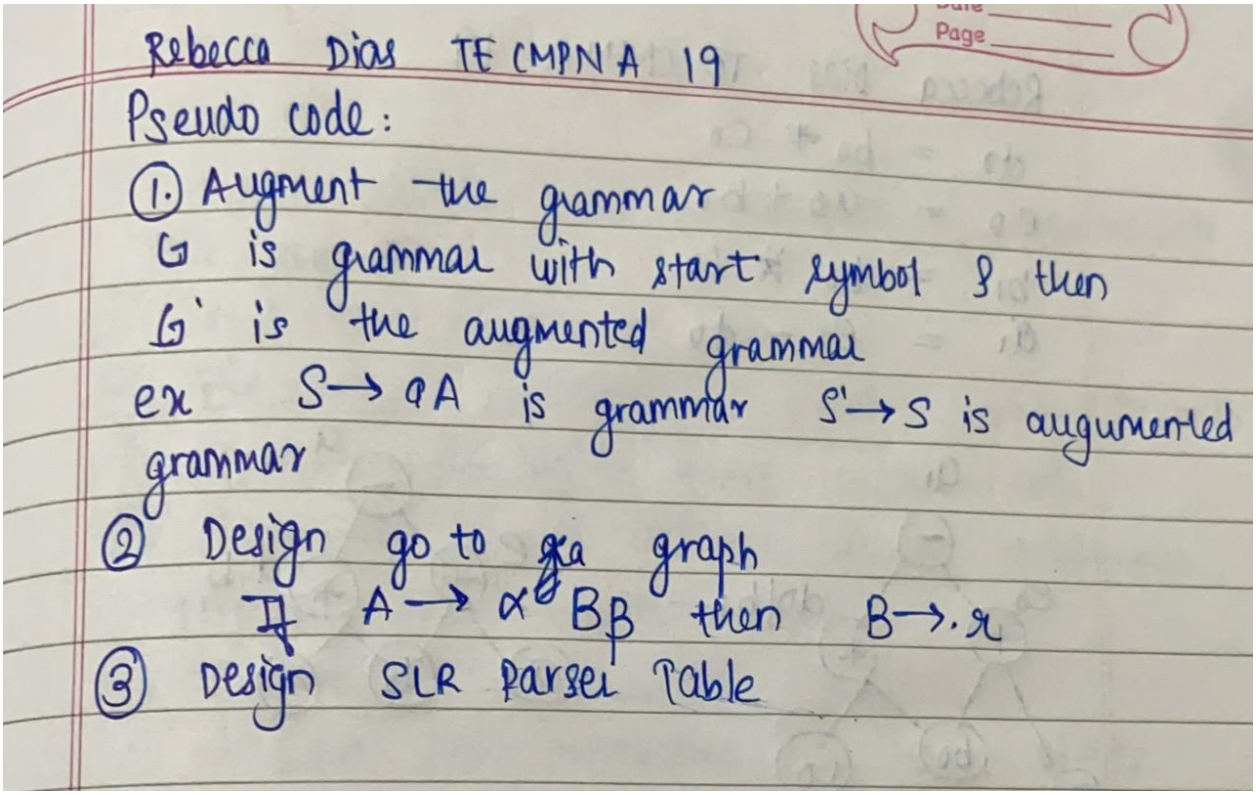


Implementation

Given Grammar G

S AA
A aA/ b

Pseudo Code



Code:

```
import pandas as pd

n_d=80
print('REBECCA DIAS TE CMPN A 19/182027')

print("If\n\tA->abc|epsilon\n\tB->pqr\nthen,\n\tNumber of unique non-
terminals on LHS=2\n\tnon-terminal number 1: A\n\tnon-terminal number 2: B\n\tRHS for non-
terminal 1: abc|epsilon\n\tRHS for non-terminal 2: pqr")
print("-"*n_d)
num=int(input("Enter the number of unique non-terminals on LHS: "))
nt_list=[]
production_list=[]
first_list=[]
follow_list=[]
all_ch=set()

for i in range(num):
    nt=input(f"Enter non-terminal number {i+1}: ")
    production=input(f"Enter RHS for non-terminal number {i+1}: ")
    all_ch.add(nt)
    nt_list.append(nt)
```

```
    production_list.append(production)
    first_list.append([])
    follow_list.append([])
follow_list[0].append('$')

r=1
later_use=[]
for i in range(len(nt_list)):
    if ("|" not in production_list[i]):
        later_use.append([nt_list[i],production_list[i],r])
        r+=1
    else:
        for p in production_list[i].split("|"):
            later_use.append([nt_list[i],p,r])
            r+=1

for production in production_list:
    if ("|" in production):
        for prod in production.split("|"):
            for ch in prod:
                all_ch.add(ch)
    else:
        for ch in production:
            all_ch.add(ch)
all_ch.add("$")

def nt_present(f_list,production,index,j):
    for first_value in first_list[index]:
        if (first_value=="epsilon"):
            if (j==len(production)-1):
                f_list.append("epsilon")
            elif (production[j+1] in nt_list):
                nt_present(f_list,production,nt_list.index(production[j+1]),j+1)
            elif (production[j+1] not in nt_list):
                f_list.append(production[j+1])
        else:
            f_list.append(first_value)
    return


for i in reversed(range(num)):
    nt=nt_list[i]
    productions=production_list[i]
    t_productions=productions.split("|")
    j=0
    for production in t_productions:
        if (production=="epsilon"):
            first_list[i].append(production)
            continue
        elif (production[j] not in nt_list):
            first_list[i].append(production[j])
        else:
            index=nt_list.index(production[j])
```

```
            nt_present(first_list[i],production,index,j)

def follow_nt_present(p_idx,f_list,production,j):
    nt_index=nt_list.index(production[j])

    for first in first_list[nt_index]:
        if (first=="epsilon"):
            if (j==len(production)-1):
                for follow in follow_list[p_idx]:
                    f_list.append(follow)
            elif (production[j+1] not in nt_list):
                f_list.append(production[j+1])
            else:
                follow_nt_present(p_idx,f_list,production,j+1)
        else:
            f_list.append(first)
    return

for prod_idx,productions in enumerate(production_list):
    for production in productions.split("|"):
            if (production=="epsilon"):
                continue
            else:
                for j in range(len(production)):
                    if (production[j] in nt_list):
                        nt_index=nt_list.index(production[j])
                        if (j==len(production)-1):
                            if (nt_index==prod_idx):
                                continue
                            for follow_value in follow_list[prod_idx]:
                                follow_list[nt_index].append(follow_value)
                        elif (production[j+1] not in nt_list):
                            follow_list[nt_index].append(production[j+1])
                        else:
                            follow_nt_present(prod_idx,follow_list[nt_index],production,j+1)

print("-"*n_d)
for i in range(num):
    print(f"{nt_list[i]}\tFIRST: {first_list[i]}\tFOLLOW: {follow_list[i]}")
print("-"*n_d)
for idx,production in enumerate(production_list):
    if("|" in production):
        production_list[idx]=production.split("|")
    if (type(production_list[idx])==list):
        for ele_idx in range(len(production_list[idx])):
            production_list[idx][ele_idx]="."+production_list[idx][ele_idx]
    else:
        production_list[idx]="."+production_list[idx]

production_list=["."+nt_list[0]]+production_list
nt_list=[nt_list[0]+"'"]+nt_list

#print("All non-terminals:",nt_list)
```

```
#print("All productions:",production_list)

state_list=[[[nt_list[0],nt_list[1]],[production_list[0],production_list[1]]]]
for p in state_list[0][1]:
    if (p[1] in nt_list and p[1] not in state_list[0][0]):
        nt_idx=nt_list.index(p[1])
        if (type(production_list[nt_idx])==list):
            num_repeat=len(production_list[nt_idx])
            for _ in range(num_repeat):
                state_list[0][0].append(nt_list[nt_idx])
            for production in production_list[nt_idx]:
                state_list[0][1].append(production)

#print("Initial state list:",state_list)
cpy_state_list=[]
for state in state_list:
    cpy_s=[[],[]]
    for nts in state[0]:
        cpy_s[0].append(nts)
    for prods in state[1]:
        cpy_s[1].append(prods)
    cpy_state_list.append(cpy_s)

transitions=[]
def create_states(state_list,new_state_list):
    temp_state_list=[]
    for state in state_list:
        temp_list=[[],[]]
        for nts in state[0]:
            temp_list[0].append(nts)
        for prods in state[1]:
            temp_list[1].append(prods)
    temp_state_list.append(temp_list)
    for state_idx,_ in enumerate(state_list):
        for p_idx,p in enumerate(state_list[state_idx][1]):
            new_state=[[],[]]
            dot_idx=p.index(".")
            if (dot_idx+1==len(p)):
                continue
            else:
                if (dot_idx+2==len(p)):
                    sg=p[:dot_idx]+p[dot_idx+1]+p[dot_idx]
                else:
                    sg=p[:dot_idx]+p[dot_idx+1]+p[dot_idx]+p[dot_idx+2:]
                if (p[dot_idx+1] in nt_list):
                    new_state[0].append(state_list[state_idx][0][p_idx])
                    new_state[1].append(sg)
                    if (dot_idx+2!=len(p) and sg[dot_idx+2] in nt_list):
                        nt_idx=nt_list.index(p[dot_idx+1])
                        if (type(production_list[nt_idx])==list):
                            num_repeat=len(production_list[nt_idx])
                            for _ in range(num_repeat):
                                new_state[0].append(nt_list[nt_idx])
```

```
                        for production in production_list[nt_idx]:
                            new_state[1].append(production)
                    else:
                        new_state[0].append(nt_list[nt_idx])
                        new_state[1].append(production_list[nt_idx])
                else:
                    new_state[0].append(state_list[state_idx][0][p_idx])
                    new_state[1].append(sg)
                    if (dot_idx+2!=len(p) and sg[dot_idx+2] in nt_list):
                        nt_idx=nt_list.index(sg[dot_idx+2])
                        if (type(production_list[nt_idx])==list):
                            num_repeat=len(production_list[nt_idx])
                            for _ in range(num_repeat):
                                new_state[0].append(nt_list[nt_idx])
                            for production in production_list[nt_idx]:
                                new_state[1].append(production)
                        else:
                            new_state[0].append(nt_list[nt_idx])
                            new_state[1].append(production_list[nt_idx])

            #temp_state_list.append(state_list[state_idx])
            state_list[state_idx][1][p_idx]=p[dot_idx+1]+p[dot_idx]+p[dot_idx+2:]
            if (new_state not in new_state_list):
                new_state_list.append(new_state)
            transitions.append([state_idx,p[dot_idx+1],new_state_list.index(new_state)+1])
    final_list=temp_state_list+new_state_list
    return final_list,transitions,new_state_list

semi_final_list_1,transitions_1,new_state_list=create_states(state_list,[])
cpy_semi_final_list_1=[]
for state in semi_final_list_1:
    cpy_list=[[],[]]
    for nts in state[0]:
        cpy_list[0].append(nts)
    for prods in state[1]:
        cpy_list[1].append(prods)
    cpy_semi_final_list_1.append(cpy_list)

semi_final_list_2,transitions_2,final_new_state_list=create_states(cpy_semi_final_list_1,new_
state_list)
final_list=[]
for l1 in semi_final_list_1:
    if (l1 not in final_list):
        final_list.append(l1)
for l2 in semi_final_list_2:
    if (l2 not in final_list):
        final_list.append(l2)
final_transitions=[]
for t1 in transitions_1:
    if (t1 not in final_transitions):
        final_transitions.append(t1)
for t2 in transitions_2:
    if (t2 not in final_transitions):
```

```
        final_transitions.append(t2)
print("Final states:")
for fi,f in enumerate(final_list):
    print(fi,"\tNon-terminals:",f[0],"\tProductions:",f[1])
print("-"*n_d)
print("How to read final transitions?\t[From state number,when received this character,went t
o this state number]")
print("-"*n_d)
print("Final transitions:")
print(final_transitions)
state_nt=[]
for state_idx,state in enumerate(final_list):
    for p_id,p in enumerate(state[1]):
        if (p[-1]=="."):
            state_nt.append([final_list[state_idx][0][p_id],state_idx,p])

all_ch_list=list(all_ch)
final_new_state_list=cpy_state_list+final_new_state_list

display_list=[]
for _ in range(len(final_new_state_list)):
    l=[]
    for _ in range(len(all_ch_list)):
        l.append("-")
    display_list.append(l)

df=pd.DataFrame(display_list,columns=all_ch_list)

for t in final_transitions:
    if (t[1] in nt_list):
        df.at[t[0],t[1]]=t[2]
    else:
        df.at[t[0],t[1]]="S"+str(t[2])

final=[]
for sn in state_nt:
    for l in later_use:
        if (l[0]==sn[0] and l[1]==sn[2].replace(".","")):
            final.append([l[0],sn[1],l[2]])

for i in range(len(final)):
    nt_idx=nt_list.index(final[i][0])-1
    for f in follow_list[nt_idx]:
        df.at[final[i][1],f]="r"+str(final[i][2])
df.at[state_nt[0][1],"$"]="Accept"
print("-"*n_d)
print("Final table below")
print("-"*n_d)
print(df)
```

Output:

```
REBECCA DIAS TE CMPN A 19/182027
If
        A->abc|epsilon
        B->pqr
then,
        Number of unique non-terminals on LHS=2
        non-terminal number 1: A
        non-terminal number 2: B
        RHS for non-terminal 1: abc|epsilon
        RHS for non-terminal 2: pqr
--------------------------------------------------------------------------------
Enter the number of unique non-terminals on LHS: 2
Enter non-terminal number 1: S
Enter RHS for non-terminal number 1: AA
Enter non-terminal number 2: A
Enter RHS for non-terminal number 2: aA|b
--------------------------------------------------------------------------------
S       FIRST: ['a', 'b']       FOLLOW: ['$']
A       FIRST: ['a', 'b']       FOLLOW: ['a', 'b', '$']
--------------------------------------------------------------------------------
Final states:
0       Non-terminals: ["S'", 'S', 'A', 'A']    Productions: ['.S', '.AA', '.aA', '.b']
1       Non-terminals: ["S'"]   Productions: ['S.']
2       Non-terminals: ['S', 'A', 'A']  Productions: ['A.A', '.aA', '.b']
3       Non-terminals: ['A', 'A', 'A']  Productions: ['a.A', '.aA', '.b']
4       Non-terminals: ['A']    Productions: ['b.']
5       Non-terminals: ['S']    Productions: ['AA.']
6       Non-terminals: ['A']    Productions: ['aA.']
--------------------------------------------------------------------------------
How to read final transitions?  [From state number,when received this character,went to this state number]
--------------------------------------------------------------------------------
Final transitions:
[[0, 'S', 1], [0, 'A', 2], [0, 'a', 3], [0, 'b', 4], [2, 'A', 5], [2, 'a', 3], [2, 'b', 4], [3, 'A', 6], [3, 'a', 3], [3, 'b', 4]]
--------------------------------------------------------------------------------
Final table below
--------------------------------------------------------------------------------
    a   b  S       $  A
0  S3  S4  1       -  2
1   -   -  -  Accept  -
2  S3  S4  -       -  5
3  S3  S4  -       -  6
4  r3  r3  -      r3  -
5   -   -  -      r1  -
6  r2  r2  -      r2  -
```

Conclusion

SLR Parser is designed to create Parse Tree using bottom-up approach. In this experiment we implemented SLR parser to create a parse tree for the given grammar and obtained desired results .