**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.

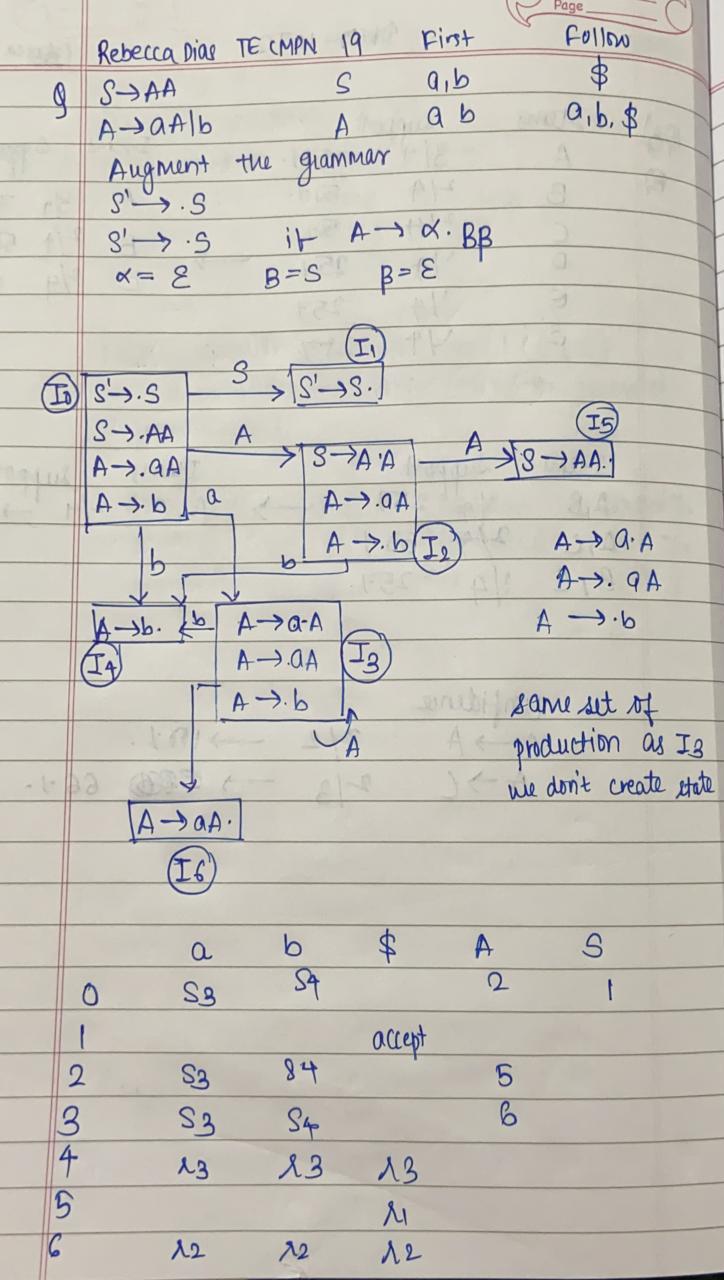
Construction of SLR parsing table –

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.

1. 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.
2. All entries not defined by rules 2 and 3 are made error.

## Construct SLR parser for a given grammar



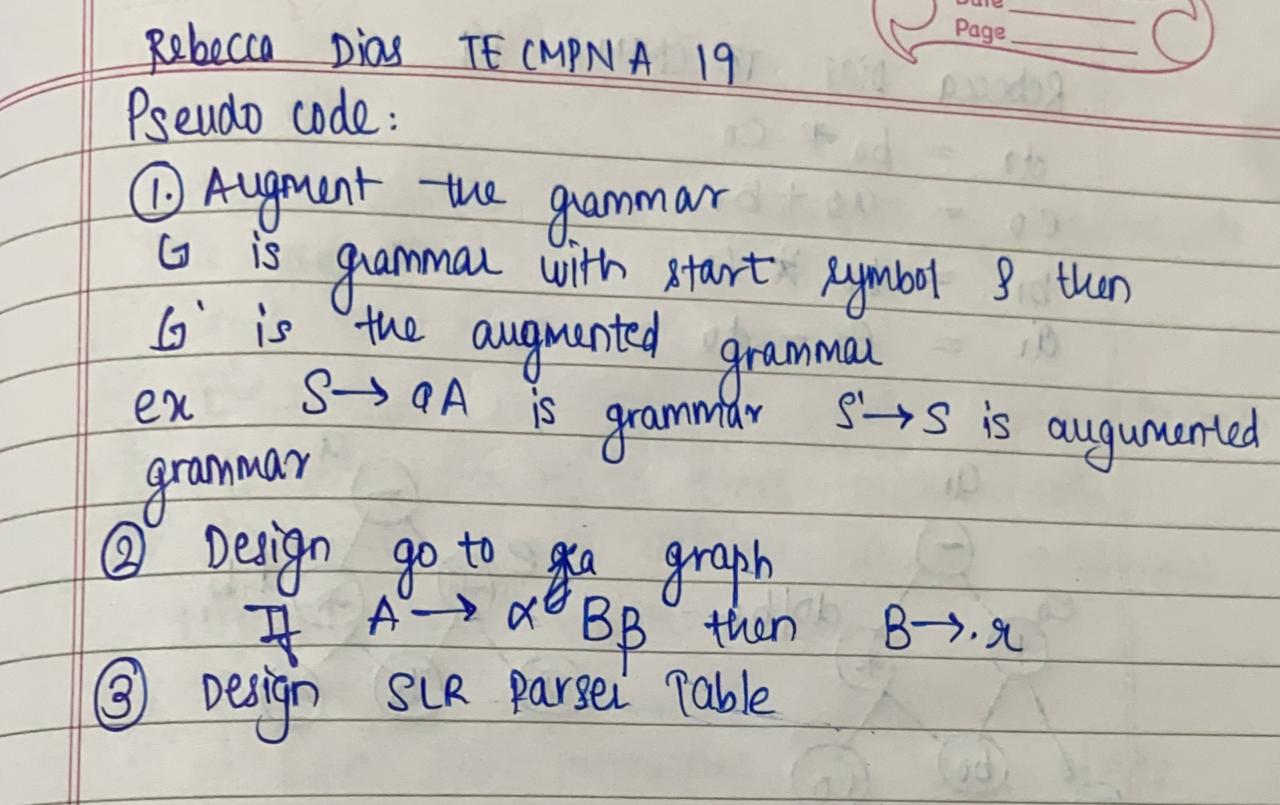
**Implementation**

**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 to 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"

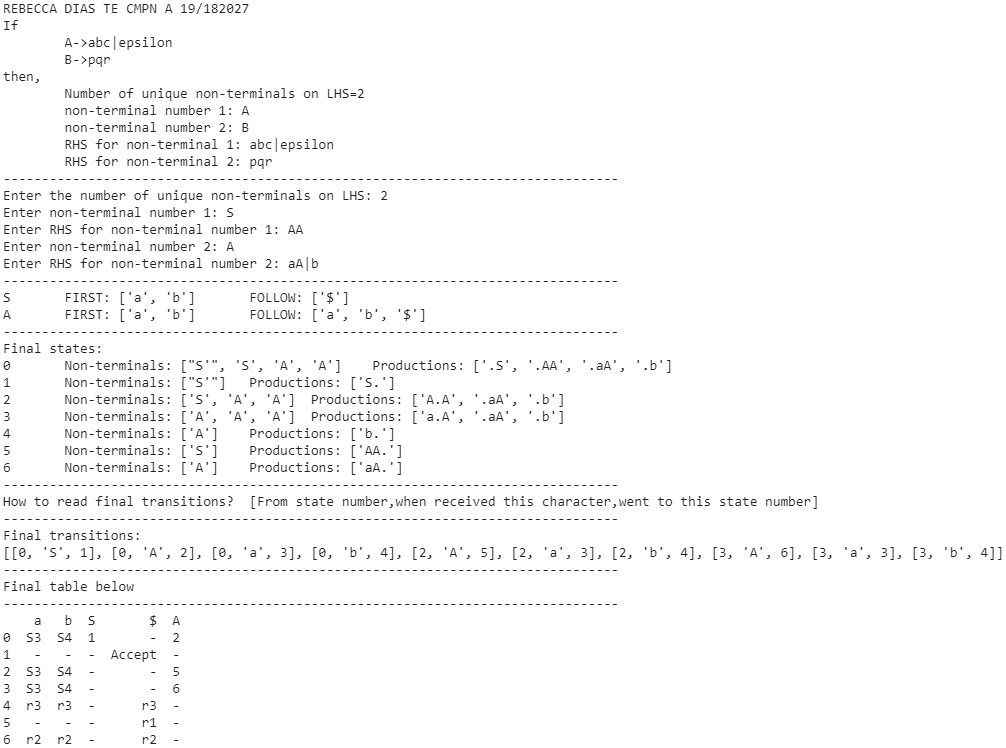
print("-"\*n\_d)

print("Final table below")

print("-"\*n\_d)

print(df)

## Output:



**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 .