Name: Apurba Koirala

Reg no: 22BCE3799

Subject Code: BCSE307P

Course Title: Compiler Design Lab

Lab Slot: L49 + L50

Guided By: Dr. Kannadasan R Lab

Assessment 4.

Problem Statement: Design a LALR Bottom Up Parser for the given grammar

Design and implement an LALR bottom up Parser for checking the syntax of the statements in the given language.

Aim: Design LALR Bottom Up Parser for a grammar

Code:

```
MAX = 100

NUM_STATES = 5

NUM_SYMBOLS = 4


action = [

    [2, 3, -1, -1],

    [-1, -1, -1, 0],

    [2, 3, -1, -1],

    [-2, -2, -2, -2],

    [-1, -1, -1, -1],

]


goto_table = [

    [1],

    [-1],

    [4],

    [-1],

    [-1],

]
```

```python
C, D, B, DOLLAR = 0, 1, 2, 3

class Stack:
    def __init__(self):
        self.items = []

    def push(self, item):
        if len(self.items) < MAX:
            self.items.append(item)
        else:
            print("Stack overflow")

    def pop(self):
        if self.items:
            return self.items.pop()
        else:
            print("Stack underflow")
            return -1

    def peek(self):
        if self.items:
            return self.items[-1]
        else:
            return -1

    def display(self):
        print("Stack:", self.items)
```

```
def print_parsing_table():
    print("Parsing Action Table:")
    print("State | c | d | b | $")
    print("--------------------")
    for i in range(NUM_STATES):
        print(f"{i:2d}    ", end="")
        for j in range(NUM_SYMBOLS):
            if action[i][j] == -1:
                print("   . ", end="")
            elif action[i][j] < 0:
                print(f" R{-action[i][j]} ", end="")
            else:
                print(f" S{action[i][j]} ", end="")
        print()

    print("\nParsing Goto Table:")
    print("State | A")
    print("-------------")
    for i in range(NUM_STATES):
        print(f"{i:2d}    ", end="")
        if goto_table[i][0] == -1:
            print("   . ")
        else:
            print(f" {goto_table[i][0]:2d} ")

def LALR_parser(input_string):
```

```
    s = Stack()

    s.push(0)


    i = 0

    while i < len(input_string):

        symbol = C if input_string[i] == 'c' else D if input_string[i] == 'd' else B if input_string[i] == 'b' else
DOLLAR

        state = s.peek()


        action_code = action[state][symbol]

        if action_code > 0:

            print(f"Shift: {input_string[i]}")

            s.push(action_code)

            i += 1

        elif action_code < 0:

            prod = -action_code

            print("Reduce by ", end="")

            if prod == 1:

                print("S -> A b")

            elif prod == 2:

                print("A -> c A")

            elif prod == 3:

                print("A -> d")

            if prod == 1:

                s.pop()

            s.pop()

            s.push(goto_table[s.peek()][0])

        elif action_code == 0:
```

```
            print("Input accepted.")
            return
        else:
            print(f"Error: Unexpected symbol {input_string[i]}")
            return
        s.display()

    if s.peek() == 0:
        print("Input accepted.")
    else:
        print("Input rejected.")

if __name__ == "__main__":
    print_parsing_table()

    input_string = input("Enter the input string (e.g., cd or cdcdb): ")

    input_string += "$"

    LALR_parser(input_string)
```

Output:

```
Parsing Action Table:
State | c | d | b | $
---------------------
 0     S2  S3    .    .
 1       .    .    .  S0
 2     S2  S3    .    .
 3     R2  R2  R2  R2
 4       .    .    .    .

Parsing Goto Table:
State | A
-------------
 0      1
 1       .
 2      4
 3       .
 4       .
Enter the input string (e.g., cd or cdcdb): cd
Shift: c
Stack: [0, 2]
Shift: d
Stack: [0, 2, 3]
Reduce by A -> c A
Stack: [0, 2, 4]
Reduce by S -> A b
Stack: [0, 1]
Input accepted.
```

Question 19 B

Problem Statement : Design SLR Parser

Design SLR bottom up parser for the above language

ALGORITHM

SStep1:	Start
Step2:	Initially the parser has s0 on the stack where s0 is the initial state and w$ is in buffer
Step3:	Set ip point to the first symbol of w$
Step4:	repeat forever, begin
Step5:	Let S be the state on top of the stack and a symbol pointed to by ip
Step6:	If action [S, a] =shift S then begin Push S1 on to the top of the stack Advance ip to next input symbol
Step7:	Else if action [S, a], reduce A->B then begin Pop 2* |B| symbols of the stack Let S1 be the state now on the top of the stack
Step8:	Output the production A→B End
Step9:	else if action [S, a]=accepted, then return Else Error() End
Step10:	Stop

AIM: To design SLR bottom up parser for a language

Code:

```
import copy


def grammarAugmentation(rules, nonterm_userdef,
                                        start_symbol):


    newRules = []
```

```
    newChar = start_symbol + "'"
    while (newChar in nonterm_userdef):
        newChar += "'"
    newRules.append([newChar,
                    ['.', start_symbol]])

    for rule in rules:

        k = rule.split("->")
        lhs = k[0].strip()
        rhs = k[1].strip()

        multirhs = rhs.split('|')
        for rhs1 in multirhs:
            rhs1 = rhs1.strip().split()

            rhs1.insert(0, '.')
            newRules.append([lhs, rhs1])
    return newRules


def findClosure(input_state, dotSymbol):
    global start_symbol, \
        separatedRulesList, \
        statesDict

    closureSet = []
```

```
if dotSymbol == start_symbol:
    for rule in separatedRulesList:
        if rule[0] == dotSymbol:
            closureSet.append(rule)
else:
    closureSet = input_state

prevLen = -1
while prevLen != len(closureSet):
    prevLen = len(closureSet)

    tempClosureSet = []
    for rule in closureSet:
        indexOfDot = rule[1].index('.')
        if rule[1][-1] != '.':
            dotPointsHere = rule[1][indexOfDot + 1]
            for in_rule in separatedRulesList:
                if dotPointsHere == in_rule[0] and \
                        in_rule not in tempClosureSet:
                    tempClosureSet.append(in_rule)

    for rule in tempClosureSet:
        if rule not in closureSet:
            closureSet.append(rule)
return closureSet
```

```
def compute_GOTO(state):
	global statesDict, stateCount

	generateStatesFor = []
	for rule in statesDict[state]:

		if rule[1][-1] != '.':
			indexOfDot = rule[1].index('.')
			dotPointsHere = rule[1][indexOfDot + 1]
			if dotPointsHere not in generateStatesFor:
				generateStatesFor.append(dotPointsHere)

	if len(generateStatesFor) != 0:
		for symbol in generateStatesFor:
			GOTO(state, symbol)
	return


def GOTO(state, charNextToDot):
	global statesDict, stateCount, stateMap

	newState = []
	for rule in statesDict[state]:
		indexOfDot = rule[1].index('.')
		if rule[1][-1] != '.':
			if rule[1][indexOfDot + 1] == \
						charNextToDot:
```

```
                        shiftedRule = copy.deepcopy(rule)
                        shiftedRule[1][indexOfDot] = \
                            shiftedRule[1][indexOfDot + 1]
                        shiftedRule[1][indexOfDot + 1] = '.'
                        newState.append(shiftedRule)

addClosureRules = []
for rule in newState:
    indexDot = rule[1].index('.')
    if rule[1][-1] != '.':
        closureRes = \
            findClosure(newState, rule[1][indexDot + 1])
        for rule in closureRes:
            if rule not in addClosureRules \
                    and rule not in newState:
                addClosureRules.append(rule)

for rule in addClosureRules:
    newState.append(rule)

stateExists = -1
for state_num in statesDict:
    if statesDict[state_num] == newState:
        stateExists = state_num
        break

if stateExists == -1:
```

```
        stateCount += 1
        statesDict[stateCount] = newState
        stateMap[(state, charNextToDot)] = stateCount
    else:

        stateMap[(state, charNextToDot)] = stateExists
    return


def generateStates(statesDict):
    prev_len = -1
    called_GOTO_on = []

    while (len(statesDict) != prev_len):
        prev_len = len(statesDict)
        keys = list(statesDict.keys())

        for key in keys:
            if key not in called_GOTO_on:
                called_GOTO_on.append(key)
                compute_GOTO(key)
    return

def first(rule):
    global rules, nonterm_userdef, \
        term_userdef, diction, firsts
```

```
if len(rule) != 0 and (rule is not None):
    if rule[0] in term_userdef:
        return rule[0]
    elif rule[0] == '#':
        return '#'

if len(rule) != 0:
    if rule[0] in list(diction.keys()):

        fres = []
        rhs_rules = diction[rule[0]]

        for itr in rhs_rules:
            indivRes = first(itr)
            if type(indivRes) is list:
                for i in indivRes:
                    fres.append(i)
            else:
                fres.append(indivRes)

        if '#' not in fres:
            return fres
        else:

            newList = []
            fres.remove('#')
```

```
                    if len(rule) > 1:
                        ansNew = first(rule[1:])
                        if ansNew != None:
                            if type(ansNew) is list:
                                newList = fres + ansNew
                            else:
                                newList = fres + [ansNew]
                        else:
                            newList = fres
                        return newList


                    fres.append('#')
                    return fres


def follow(nt):
    global start_symbol, rules, nonterm_userdef, \
        term_userdef, diction, firsts, follows

    solset = set()
    if nt == start_symbol:

        solset.add('$')

    for curNT in diction:
        rhs = diction[curNT]
```

```
for subrule in rhs:
    if nt in subrule:

        while nt in subrule:
            index_nt = subrule.index(nt)
            subrule = subrule[index_nt + 1:]

            if len(subrule) != 0:

                res = first(subrule)
                if '#' in res:
                    newList = []
                    res.remove('#')
                    ansNew = follow(curNT)
                    if ansNew != None:
                        if type(ansNew) is list:
                            newList = res + ansNew
                        else:
                            newList = res + [ansNew]
                    else:
                        newList = res
                    res = newList
            else:
                if nt != curNT:
                    res = follow(curNT)
            if res is not None:
```

```
                    if type(res) is list:
                        for g in res:
                            solset.add(g)
                    else:
                        solset.add(res)
    return list(solset)



def createParseTable(statesDict, stateMap, T, NT):
    global separatedRulesList, diction
    rows = list(statesDict.keys())
    cols = T+['$']+NT
    Table = []
    tempRow = []
    for y in range(len(cols)):
        tempRow.append('')
    for x in range(len(rows)):
        Table.append(copy.deepcopy(tempRow))
    for entry in stateMap:
        state = entry[0]
        symbol = entry[1]
        a = rows.index(state)
        b = cols.index(symbol)
        if symbol in NT:
            Table[a][b] = Table[a][b]\
                + f"{stateMap[entry]} "
        elif symbol in T:
```

```
                Table[a][b] = Table[a][b]\
                    + f"S{stateMap[entry]} "
numbered = {}
key_count = 0
for rule in separatedRulesList:
    tempRule = copy.deepcopy(rule)
    tempRule[1].remove('.')
    numbered[key_count] = tempRule
    key_count += 1
addedR = f"{separatedRulesList[0][0]} -> " \
    f"{separatedRulesList[0][1][1]}"
rules.insert(0, addedR)
for rule in rules:
    k = rule.split("->")
    k[0] = k[0].strip()
    k[1] = k[1].strip()
    rhs = k[1]
    multirhs = rhs.split('|')
    for i in range(len(multirhs)):
        multirhs[i] = multirhs[i].strip()
        multirhs[i] = multirhs[i].split()
    diction[k[0]] = multirhs
for stateno in statesDict:
    for rule in statesDict[stateno]:
        if rule[1][-1] == '.':
            temp2 = copy.deepcopy(rule)
            temp2[1].remove('.')
```

```
                        for key in numbered:
                            if numbered[key] == temp2:
                                follow_result = follow(rule[0])
                                for col in follow_result:
                                    index = cols.index(col)
                                    if key == 0:
                                        Table[stateno][index] = "Accept"
                                    else:
                                        Table[stateno][index] =\
                                            Table[stateno][index]+f"R{key} "

    print("\nSLR(1) parsing table:\n")
    frmt = "{:>8}" * len(cols)
    print(" ", frmt.format(*cols), "\n")
    ptr = 0
    j = 0
    for y in Table:
        frmt1 = "{:>8}" * len(y)
        print(f"{{:>3}} {frmt1.format(*y)}"
            .format('I'+str(j)))
        j += 1

def printResult(rules):
    for rule in rules:
        print(f"{rule[0]} ->"
            f" {' '.join(rule[1])}")
```

```
def printAllGOTO(diction):
	for itr in diction:
		print(f"GOTO ( I{itr[0]} ,"
			f" {itr[1]} ) = I{stateMap[itr]}")
rules = ["E -> E + T | T",
		"T -> T * F | F",
		"F -> ( E ) | id"
		]
nonterm_userdef = ['E', 'T', 'F']
term_userdef = ['id', '+', '*', '(', ')']
start_symbol = nonterm_userdef[0]
print("\nOriginal grammar input:\n")
for y in rules:
	print(y)
print("\nGrammar after Augmentation: \n")
separatedRulesList = \
	grammarAugmentation(rules,
						nonterm_userdef,
						start_symbol)
printResult(separatedRulesList)
start_symbol = separatedRulesList[0][0]
print("\nCalculated closure: I0\n")
I0 = findClosure(0, start_symbol)
printResult(I0)
statesDict = {}
stateMap = {}
statesDict[0] = I0
```

```
stateCount = 0

generateStates(statesDict)

print("\nStates Generated: \n")

for st in statesDict:

	print(f"State = I{st}")

	printResult(statesDict[st])

	print()


print("Result of GOTO computation:\n")

printAllGOTO(stateMap)

diction = {}

createParseTable(statesDict, stateMap,

					term_userdef,

					nonterm_userdef)
```

OUTPUT:

```
Original grammar input:

E -> E + T | T
T -> T * F | F
F -> ( E ) | id

Grammar after Augmentation:

E' -> . E
E -> . E + T
E -> . T
T -> . T * F
T -> . F
F -> . ( E )
F -> . id

Calculated closure: I0

E' -> . E
E -> . E + T
E -> . T
T -> . T * F
T -> . F
F -> . ( E )
F -> . id

States Generated:

State = I0
E' -> . E
E -> . E + T
E -> . T
T -> . T * F
T -> . F
F -> . ( E )
F -> . id
```

```
State = I1
E' -> E .
E -> E . + T

State = I2
E -> T .
T -> T . * F

State = I3
T -> F .

State = I4
F -> ( . E )
E -> . E + T
E -> . T
T -> . T * F
T -> . F
F -> . ( E )
F -> . id

State = I5
F -> id .

State = I6
E -> E + . T
T -> . T * F
T -> . F
F -> . ( E )
F -> . id

State = I7
T -> T * . F
F -> . ( E )
F -> . id
```

```
State = I8
F -> ( E . )
E -> E . + T

State = I9
E -> E + T .
T -> T . * F

State = I10
T -> T * F .

State = I11
F -> ( E ) .

Result of GOTO computation:

GOTO ( I0 , E ) = I1
GOTO ( I0 , T ) = I2
GOTO ( I0 , F ) = I3
GOTO ( I0 , ( ) = I4
GOTO ( I0 , id ) = I5
GOTO ( I1 , + ) = I6
GOTO ( I2 , * ) = I7
GOTO ( I4 , E ) = I8
GOTO ( I4 , T ) = I2
GOTO ( I4 , F ) = I3
GOTO ( I4 , ( ) = I4
GOTO ( I4 , id ) = I5
GOTO ( I6 , T ) = I9
GOTO ( I6 , F ) = I3
GOTO ( I6 , ( ) = I4
GOTO ( I6 , id ) = I5
GOTO ( I7 , F ) = I10
GOTO ( I7 , ( ) = I4
GOTO ( I7 , id ) = I5
GOTO ( I8 , ) ) = I11
GOTO ( I8 , + ) = I6
GOTO ( I9 , * ) = I7
```

SLR(1) parsing table:

	id	+	*	(	)	$	E	T	F
I0	S5			S4			1	2	3
I1		S6				Accept			
I2		R2	S7		R2	R2			
I3		R4	R4		R4	R4			
I4	S5			S4			8	2	3
I5		R6	R6		R6	R6			
I6	S5			S4				9	3
I7	S5			S4					10
I8		S6			S11				
I9		R1	S7		R1	R1			
I10		R3	R3		R3	R3			
I11		R5	R5		R5	R5			

Question 23

write a C program to implement the shift-reduce parsing algorithm.

TOOLS/APPARATUS: Turbo C or gcc / gprof compiler in linux.

Algorithm:

Grammar:

E->E+E

E->E*E

E->E/E

E->a/b

Method:

Stack	Input Symbol	Action
$	id1*id2$	shift
$id1	*id2 $	shift *
$*	id2$	shift id2
$id2	$	shift
$	$	accept

Shift: Shifts the next input symbol onto the stack.

Reduce: Right end of the string to be reduced must be at the top of the stack. Accept: Announce successful completion of parsing.

Error: Discovers a syntax error and call an error recovery routine.

AIM: Implement Shift Reduce Parser using the given algorithm

```
a = "a*a/b"

stk = []

act = "SHIFT"


def check():

    global stk, a

    ac = "REDUCE TO E -> "


    if len(stk) >= 1 and stk[-1] == 'a':

        print(f"${''.join(stk[:-1])}a\t{a}$\t{ac}a")

        stk[-1] = 'E'

        print(f"${''.join(stk)}\t{a}$")


    if len(stk) >= 1 and stk[-1] == 'b':

        print(f"${''.join(stk[:-1])}b\t{a}$\t{ac}b")

        stk[-1] = 'E'

        print(f"${''.join(stk)}\t{a}$")


    i = 0

    while i < len(stk) - 2:

        if stk[i] == 'E' and stk[i + 1] == '+' and stk[i + 2] == 'E':

            print(f"${''.join(stk[:i])}E+E{''.join(stk[i+3:])}\t{a}$\t{ac}E+E")

            stk[i] = 'E'

            del stk[i + 1:i + 3]

            print(f"${''.join(stk)}\t{a}$")

            i = max(i - 2, 0)

        else:
```

```
            i += 1

    i = 0
    while i < len(stk) - 2:
        if stk[i] == 'E' and stk[i + 1] == '*' and stk[i + 2] == 'E':
            print(f"${''.join(stk[:i])}E*E{''.join(stk[i+3:])}\t{a}$\t{ac}E*E")
            stk[i] = 'E'
            del stk[i + 1:i + 3]
            print(f"${''.join(stk)}\t{a}$")
            i = max(i - 2, 0)
        else:
            i += 1

    i = 0
    while i < len(stk) - 2:
        if stk[i] == 'E' and stk[i + 1] == '/' and stk[i + 2] == 'E':
            print(f"${''.join(stk[:i])}E/E{''.join(stk[i+3:])}\t{a}$\t{ac}E/E")
            stk[i] = 'E'
            del stk[i + 1:i + 3]
            print(f"${''.join(stk)}\t{a}$")
            i = max(i - 2, 0)
        else:
            i += 1

def main():
    global stk, a, act
    print("stack     input     action")
    print(f"${''.join(stk)}\t{a}$\t{act}")
```

```
    for char in a:
        stk.append(char)
        a = a[1:]
        print(f"${''.join(stk)}\t{a}$\t{act}")
        check()

    check()

    if len(stk) == 1 and stk[0] == 'E':
        print(f"${''.join(stk)}\t{a}$\tAccept")
    else:
        print(f"${''.join(stk)}\t{a}$\tReject")

if __name__ == "__main__":
    main()
```

OUTPUT:

```
stack      input      action
$       a*a/b$  SHIFT
$a      *a/b$   SHIFT
$a      *a/b$   REDUCE TO E -> a
$E      *a/b$
$E*     a/b$    SHIFT
$E*a    /b$     SHIFT
$E*a    /b$     REDUCE TO E -> a
$E*E    /b$
$E*E    /b$     REDUCE TO E -> E*E
$E      /b$
$E/     b$      SHIFT
$E/b    $       SHIFT
$E/b    $       REDUCE TO E -> b
$E/E    $
$E/E    $       REDUCE TO E -> E/E
$E      $
$E      $       Accept
```