VIT

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

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 3.

Q16

Question 16

Problem Statement : Design a Predictive Parser for the following grammar
G: { E-> TE' , E' -> +TE' | 0, T-> FT' , T'-> *FT' | 0 , F-> (E) | id }

write a 'C' Program to implement for the Predictive Parser (Non Recursive Descent-parser) for the given grammar.

Given the parse Table:

	id	+	*	(	)	$
E	E-> TE'			E-> TE'		
E'		E' -> +TE'			E'->0	E'->0
T	T-> FT'			T-> FT'		
T'		T'->0	T'-> *FT'		T'->0	T'->0
F	F-> id			F->(E)		

Code:

```
import copy


def removeLeftRecursion(rulesDiction):
    store = {}
    for lhs in rulesDiction:
        alphaRules = []
        betaRules = []
        allrhs = rulesDiction[lhs]
        for subrhs in allrhs:
            if subrhs[0] == lhs:
                alphaRules.append(subrhs[1:])
            else:
```

```
            betaRules.append(subrhs)
        if len(alphaRules) != 0:
            lhs_ = lhs + "'"
            while (lhs_ in rulesDiction.keys()) or (lhs_ in store.keys()):
                lhs_ += "'"
            for b in range(len(betaRules)):
                betaRules[b].append(lhs_)
            rulesDiction[lhs] = betaRules
            for a in range(len(alphaRules)):
                alphaRules[a].append(lhs_)
            alphaRules.append(['0'])
            store[lhs_] = alphaRules
    for left in store:
        rulesDiction[left] = store[left]
    return rulesDiction


def LeftFactoring(rulesDiction):
    newDict = {}
    for lhs in rulesDiction:
        allrhs = rulesDiction[lhs]
        temp = dict()
        for subrhs in allrhs:
            if subrhs[0] not in list(temp.keys()):
                temp[subrhs[0]] = [subrhs]
            else:
                temp[subrhs[0]].append(subrhs)
```

```
        new_rule = []
        tempo_dict = {}
        for term_key in temp:
            allStartingWithTermKey = temp[term_key]
            if len(allStartingWithTermKey) > 1:
                lhs_ = lhs + "'"
                while (lhs_ in rulesDiction.keys()) or (lhs_ in tempo_dict.keys()):
                    lhs_ += "'"
                new_rule.append([term_key, lhs_])
                ex_rules = []
                for g in temp[term_key]:
                    ex_rules.append(g[1:])
                tempo_dict[lhs_] = ex_rules
            else:
                new_rule.append(allStartingWithTermKey[0])
        newDict[lhs] = new_rule
        for key in tempo_dict:
            newDict[key] = tempo_dict[key]
    return newDict


def first(rule):
    global diction, firsts
    if len(rule) != 0 and (rule is not None):
        if rule[0] in term_userdef:
            return [rule[0]]
        elif rule[0] == '0':
```

```
            return ['0']

    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 '0' in indivRes:
                    indivRes.remove('0')
                    if len(rule) > 1:
                        ansNew = first(rule[1:])
                        if ansNew != None:
                            if type(ansNew) is list:
                                indivRes += ansNew
                            else:
                                indivRes.append(ansNew)
                if type(indivRes) is list:
                    fres.extend(indivRes)
                else:
                    fres.append(indivRes)
            return fres
        else:
            return []


def follow(nt):
```

```
global start_symbol, 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:
            index_nt = subrule.index(nt)
            subrule = subrule[index_nt + 1:]
            if len(subrule) != 0:
                res = first(subrule)
                if '0' in res:
                    res.remove('0')
                    if len(subrule) > 1:
                        res += first(subrule[1:])
                    res.append('0')
                solset.update(res)
            else:
                if nt != curNT:
                    res = follow(curNT)
                    if res is not None:
                        solset.update(res)
return list(solset)
```

```
def computeAllFirsts():
  global diction, firsts
  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

  diction = removeLeftRecursion(diction)
  diction = LeftFactoring(diction)

  for y in list(diction.keys()):
    t = set()
    for sub in diction.get(y):
      res = first(sub)
      if res != None:
        if type(res) is list:
          t.update(res)
        else:
          t.add(res)
    firsts[y] = t
```

```
def computeAllFollows():
    global start_symbol, diction, firsts, follows
    for NT in diction:
        solset = set()
        sol = follow(NT)
        if sol is not None:
            solset.update(sol)
        follows[NT] = solset


def createParseTable():
    import copy
    global diction, firsts, follows, term_userdef
    mx_len_first = 0
    mx_len_fol = 0
    for u in diction:
        k1 = len(str(firsts[u]))
        k2 = len(str(follows[u]))
        if k1 > mx_len_first:
            mx_len_first = k1
        if k2 > mx_len_fol:
            mx_len_fol = k2


    print(f"{{:<{10}}} {{:<{mx_len_first + 5}}} {{:<{mx_len_fol + 5}}}".format("Non-T", "FIRST",
"FOLLOW"))
    for u in diction:
```

```
        print(f"{{:<{10}}} {{:<{mx_len_first + 5}}} {{:<{mx_len_fol + 5}}}".format(u, str(firsts[u]),
str(follows[u])))

    ntlist = list(diction.keys())
    terminals = copy.deepcopy(term_userdef)
    terminals.append('$')

    mat = []
    for x in diction:
        row = []
        for y in terminals:
            row.append('')
        mat.append(row)

    grammar_is_LL = True

    for lhs in diction:
        rhs = diction[lhs]
        for y in rhs:
            res = first(y)
            if '0' in res:
                if type(res) == str:
                    res = [res]
                res.remove('0')
                res += follows[lhs]
            if type(res) is str:
```

```
            res = [res]
        for c in res:
            if c in terminals:
                xnt = ntlist.index(lhs)
                yt = terminals.index(c)
                if mat[xnt][yt] == '':
                    mat[xnt][yt] = f"{lhs} -> {' '.join(y)}"
                else:
                    if f"{lhs} -> {' '.join(y)}" in mat[xnt][yt]:
                        continue
                    else:
                        grammar_is_LL = False
                        mat[xnt][yt] += f", {lhs} -> {' '.join(y)}"

print("\nGenerated parsing table:\n")
frmt = "{:>12}" * len(terminals)
print(frmt.format(*terminals))

j = 0
for y in mat:
    frmt1 = "{:>12}" * len(y)
    print(f"{ntlist[j]} {frmt1.format(*y)}")
    j += 1

return (mat, grammar_is_LL, terminals)
```

```
rules = ["E -> T E'",
        "E' -> + T E' | 0",
        "T -> F T'",
        "T' -> * F T' | 0",
        "F -> ( E ) | id"]

term_userdef = ['id', '+', '*', '(', ')']
diction = {}
firsts = {}
follows = {}

computeAllFirsts()
start_symbol = list(diction.keys())[0]
computeAllFollows()
(parsing_table, result, tabTerm) = createParseTable()
```

Output:

```
Non-T      FIRST            FOLLOW
E          {'id', '('}      {')', '$'}
E'         {'0', '+'}       {')', '$'}
T          {'id', '('}      {'+'}
T'         {'0', '*'}       {'+'}
F          {'id', '('}      {'*'}

Generated parsing table:

          id           +           *           (           )           $
E    E -> T E'                              E -> T E'
E'            E' -> + T E'                                 E' -> 0     E' -> 0
T    T -> F T'                              T -> F T'
T'                 T' -> 0T' -> * F T'
F      F -> id                             F -> ( E )
```

Q 22

Question 22

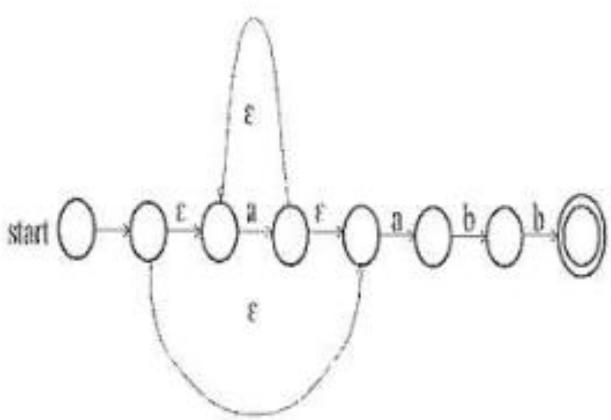
write a C program to construct a Non Deterministic Finite Automata (NFA) from Regular Expression.

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

Algorithm:

1. Start the Program.
2. Enter the regular expression R over alphabet E.
3. Decompose the regular expression R into its primitive components
4. For each component construct finite automata.
5. To construct components for the basic regular expression way that corresponding to that way compound regular expression.
6. Stop the Program.

Example:



Code:

```
def print_transition_table(q):
   print("\n\tTransition Table \n")
   print("_____________________________________\n")
   print("Current State |\tInput |\tNext State")
   print("\n_____________________________________\n")
```

```
    for i in range(len(q)):
        if q[i][0] != 0:
            print(f"  q[{i}]\t  |  a  | q[{q[i][0]}]")
        if q[i][1] != 0:
            print(f"  q[{i}]\t  |  b  | q[{q[i][1]}]")
        if q[i][2] != 0:
            if q[i][2] < 10:
                print(f"  q[{i}]\t          |  e  | q[{q[i][2]}]")
            else:
                print(f"  q[{i}]\t          |  e  | q[{q[i][2] // 10}] , q[{q[i][2] % 10}]")
    print("\n___________________________________\n")


def convert_regex_to_dfa(regex):
    q = [[0, 0, 0] for _ in range(20)]  # Transition table initialized to 20 states
    j = 0  # Tracks current state
    i = 0  # Tracks position in regex
    len_reg = len(regex)


    while i < len_reg:
        if regex[i] == 'a' and (i + 1 >= len_reg or regex[i + 1] not in ['|', '*']):
            q[j][0] = j + 1
            j += 1
        elif regex[i] == 'b' and (i + 1 >= len_reg or regex[i + 1] not in ['|', '*']):
            q[j][1] = j + 1
            j += 1
        elif regex[i] == 'e' and (i + 1 >= len_reg or regex[i + 1] not in ['|', '*']):
```

```
    q[j][2] = j + 1
    j += 1
elif regex[i] == 'a' and i + 2 < len_reg and regex[i + 1] == '|' and regex[i + 2] == 'b':
    q[j][2] = ((j + 1) * 10) + (j + 3)
    j += 1
    q[j][0] = j + 1
    j += 1
    q[j][2] = j + 3
    j += 1
    q[j][1] = j + 1
    j += 1
    q[j][2] = j + 1
    j += 1
    i += 2
elif regex[i] == 'b' and i + 2 < len_reg and regex[i + 1] == '|' and regex[i + 2] == 'a':
    q[j][2] = ((j + 1) * 10) + (j + 3)
    j += 1
    q[j][1] = j + 1
    j += 1
    q[j][2] = j + 3
    j += 1
    q[j][0] = j + 1
    j += 1
    q[j][2] = j + 1
    j += 1
    i += 2
```

```
        elif regex[i] == 'a' and i + 1 < len_reg and regex[i + 1] == '*':
            q[j][2] = ((j + 1) * 10) + (j + 3)
            j += 1
            q[j][0] = j + 1
            j += 1
            q[j][2] = ((j + 1) * 10) + (j - 1)
            j += 1
        elif regex[i] == 'b' and i + 1 < len_reg and regex[i + 1] == '*':
            q[j][2] = ((j + 1) * 10) + (j + 3)
            j += 1
            q[j][1] = j + 1
            j += 1
            q[j][2] = ((j + 1) * 10) + (j - 1)
            j += 1
        elif regex[i] == ')' and i + 1 < len_reg and regex[i + 1] == '*':
            q[0][2] = ((j + 1) * 10) + 1
            q[j][2] = ((j + 1) * 10) + 1
            j += 1
        i += 1

    print(f"Given regular expression: {regex}")
    print_transition_table(q)

regex = "(a|b)*"
convert_regex_to_dfa(regex)
```

Output:

```
Given regular expression: (a|b)*

        Transition Table

_____________________________________________

Current State | Input | Next State

_____________________________________________

  q[0]            |   e    |  q[6] , q[1]
  q[1]            |   a    |  q[2]
  q[2]            |   e    |  q[5]
  q[3]            |   b    |  q[4]
  q[4]            |   e    |  q[5]
  q[5]            |   e    |  q[6] , q[1]

_____________________________________________
```

Q 24

write a C program to implement Recursive Descent Parser.

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

Algorithm:

Input: Context Free Grammar without last recursion and an input string from the grammar.

Output: Sequence of productions rules used to derive the sentence.

Method:

Consider the grammar
E->TE
E'->+TE'/e
T->FT
T->*FT/e
F->(E)/Id

To recursive decent parser for the above grammar is given below

Code:

```
SUCCESS = 1
FAILED = 0

cursor = 0
string = ""

def E():
    global cursor
    print(f"{string[cursor:]}        E -> T E'")
```

```
        if T():
        if Edash():
        return SUCCESS
        return FAILED

def Edash():
        global cursor
        if cursor < len(string) and string[cursor] == '+':
        print(f"{string[cursor:]}        E' -> + T E'")
        cursor += 1
        if T():
        if Edash():
        return SUCCESS
        else:
        print(f"{string[cursor:]}        E' -> ε")
        return SUCCESS
        return FAILED

def T():
        global cursor
        print(f"{string[cursor:]}        T -> F T'")
        if F():
        if Tdash():
        return SUCCESS
        return FAILED
```

```
def Tdash():
    global cursor
    if cursor < len(string) and string[cursor] == '*':
    print(f"{string[cursor:]}         T' -> * F T'")
    cursor += 1
    if F():
    if Tdash():
    return SUCCESS
    else:
    print(f"{string[cursor:]}         T' -> ε")
    return SUCCESS
    return FAILED

def F():
    global cursor
    if cursor < len(string) and string[cursor] == '(':
    print(f"{string[cursor:]}         F -> ( E )")
    cursor += 1
    if E():
    if cursor < len(string) and string[cursor] == ')':
    cursor += 1
    return SUCCESS
    elif cursor < len(string) and string[cursor] == 'i':
    print(f"{string[cursor:]}         F -> i")
    cursor += 1
    return SUCCESS
```

```
        return FAILED


if __name__ == "__main__":
        string = input("Enter the string: ")
        cursor = 0
        print("\nInput\t\tAction")
        print("--------------------------------")

        if E() and cursor == len(string):
        print("--------------------------------")
        print("String is successfully parsed")
        else:
        print("--------------------------------")
        print("Error in parsing String")
```

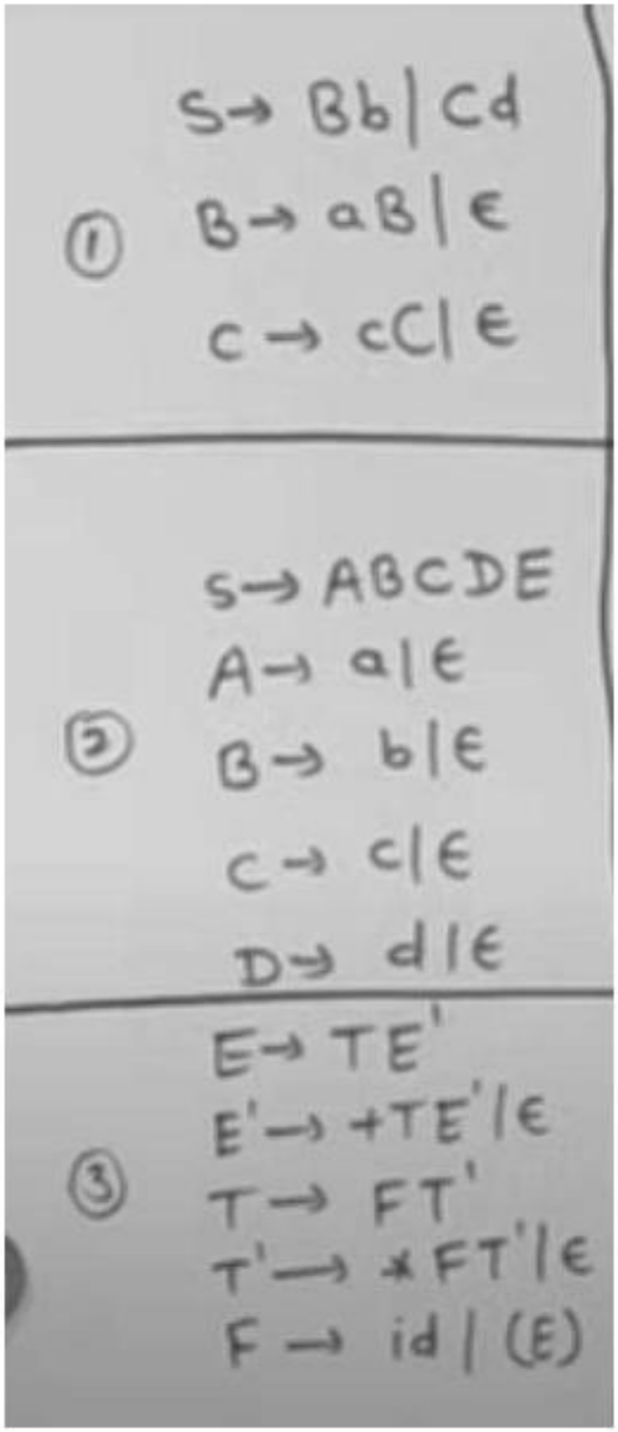
Output:

```
Enter the string: i+i*i

Input            Action
--------------------------------
i+i*i  E -> T E'
i+i*i  T -> F T'
i+i*i  F -> i
+i*i  T' -> ε
+i*i  E' -> + T E'
i*i  T -> F T'
i*i  F -> i
*i  T' -> * F T'
i  F -> i
  T' -> ε
  E' -> ε
--------------------------------
String is successfully parsed
```

Q. 30

Write a C program to find first and follow for the given grammar.



Code:

AIM: To compute first and follow for a grammar

```
productions = {}
firsts = {}
follows = {}


def find_first(symbol):
    if not symbol.isupper():
        return {symbol}


    if firsts[symbol]:
        return firsts[symbol]


    first = set()


    for production in productions[symbol]:
        if production == 'ε':
            first.add('ε')
        else:
            for char in production:
                first_of_char = find_first(char)
                first.update(first_of_char - {'ε'})
                if 'ε' not in first_of_char:
                    break
            else:
                first.add('ε')
```

```
    firsts[symbol] = first
    return first

def find_follow(symbol):
    if follows[symbol]:
        return follows[symbol]

    follow = set()

    if symbol == start_symbol:
        follow.add('$')

    for lhs, rhs in productions.items():
        for production in rhs:
            for i, char in enumerate(production):
                if char == symbol:
                    if i + 1 < len(production):
                        follow_of_next = find_first(production[i + 1])
                        follow.update(follow_of_next - {'ε'})
                        if 'ε' in follow_of_next:
                            follow.update(find_follow(lhs))
                    else:
                        if lhs != symbol:
                            follow.update(find_follow(lhs))

    follows[symbol] = follow
```

```
    return follow


def compute_firsts_and_follows():
    for non_terminal in productions:
        find_first(non_terminal)
        find_follow(non_terminal)


num_productions = int(input("Enter the number of productions: "))


for _ in range(num_productions):
    lhs, rhs = input("Enter production (A=BC|d): ").split('=')
    productions[lhs] = rhs.split('|')
    firsts[lhs] = set()
    follows[lhs] = set()


start_symbol = list(productions.keys())[0]


compute_firsts_and_follows()


for non_terminal in productions:
    print(f"First({non_terminal}) = {{ {', '.join(sorted(firsts[non_terminal]))} }}")
    print(f"Follow({non_terminal}) = {{ {', '.join(sorted(follows[non_terminal]))} }}")
```

Output:

```
Enter the number of productions: 3
Enter production (A=BC|d): S=Bb|Cd
Enter production (A=BC|d): B=aB|ε
Enter production (A=BC|d): C=cC|ε
First(S) = { a, b, c, d }
Follow(S) = { $ }
First(B) = { a, ε }
Follow(B) = { b }
First(C) = { c, ε }
Follow(C) = { d }
```