RA2011026010061

INDEX

Name: Aadarsh Joshi Class: CSE-AIML Q1 Subject: Compiler Design lab

S.No.	Date	Topic	Page No. Marks	Signature
1	13/1/23	Implementation of Lexical Analyzer	1+1.5+1	3.5
2	30/1/23	Implementation of Regular Expression to NFA	5/5	
3	1/2/23	Conversion of NFA to DFA	8/10	1/2/23
4	8/2/23	Elimination of Left Recursion and Left Factoring	4/5	
5	15/2/23	First and Follow	5+3=8	15/2
6	20/2/23	Predictive Parsing Table	5+3	22/2
7	9/3/23	Shift Reduce Parsing	5+4	23/3
8	9/3/23	Computation of Lead and Trail	5+3	
9	16/3/23	Computation of LR[0]		
10	27/3/23	Intermediate Code Generator - Postfix and Prefix.	5+4	27/3
11	3/4/23	Intermediate Code Generation - Quadruples, Triple and Indirect tuple	5+5	3/4
12	10/4/23	Implementation of simple Code Generator	5+4	10/4
13	18/4/23	Implementation of DAG	5+5	

Completed

13/1/23

EXPERIMENT - 1

AIM:

To write a program to implement a lexical analyzer in C++/C/Python

ALGORITHM:

1. Start
2. Get the input program from the file program.txt
3. Read the program line by line and check if each word in a line is a keyword, identifier, math operator, logical operator, numerical value or other symbol.
4. For each lexeme read, generate a token as follows:
   a. If the lexeme ~~read~~ is a keyword, then the token is a keyword itself
   b. If the lexeme is an identifier, then the token generated is printed on the console as identifier
   c. In the same way the math operator, logical operator, numerical value and other symbol are printed on the console
5. The stream of token generated are displayed in the console output
6. Stop

CODE

```
#include <iostream>
#include <cstring>
#include <stdlib.h>
#include <ctype.h>
#include <fstream>
using namespace std;
string arr[] = {"void", "using", "namespace", "int",
        "include", "iostream", "std", "main", "cin",
                "cout", "return", "float", "double"
                "string", "printf");
bool iskeyword (string a){
    for (int i=0, i<14; i++){
            if (arr[i]==a){
                    return true;
            }
    }
    return false;
}
int main(){
    fstream file;
    string s ,filename;
    filename = ".bdd.c");
    file.open (filename.c_str());
    while (file>>s){
        if ( s== '+' || s == "-" || s== " " || s == "/" || s=="^"
            || s== "=" || s == "==" || s == "&" || s=="|"
            || s== "%" || s == "++" || s == "--" || s=="+="
            || s== "-=" || s==" /=" || s== "=" || s=="%=")
            {
                cout << s << " is an operator \n";
                s= " ";
            }
            else if (is keyword (s))
            {
                cout << s << " is a keyword \n";
                s= " ";
            }
```

```
else if (s == "(" || s == "{" || s == "[" || s == ")" || s == "}"
        || s == "]" || s == "<" || s == ">" || s == "()"
        || s == ";" || s == "<<" || s == ">>" || s == ","
        || s == "#") {
            cout << s << " is a symbol \n";
                s = "";
            }
    else if (s == "\n" || s == " " || s == "")
    {
        s = "";
    }
    else if (isdigit(s[0])) {
        int x = 0;
        if (!isdigit(s[x++])) {
                continue;
            }
        else {
            cout << s << "is a constant \n";
                s = "";
            }
        }
    else {
        cout << s << " is an identifier \n";
                s = "";
            }
    return 0;
}
```

RESULT

The ~~following~~ implementation of lexical analyzer in C++ ~~was~~ compiled, executed and verified successfully

OUTPUT

#include is an identifier
<stdio.h> is an identifier
is an identifier
void is a keyword
main is a keyword
(is a punctuation
) is a punctuation
int is a keyword
x is an identifier
= is an operator
6 is a constant
int is a keyword
return is a keyword
0 is constant
} is punctuation

30/1/23

EXPERIMENT - 2

AIM:

To write a program Regular Expression to NFA.

ALGORITHM:

1. Start
2. Get the input from the user
3. Initialize separate variables and function for postfix, Display and NFA
5. By using Switch case Initialize different cases for the input

~~6. For '.' operator initialize different case for the input~~

6. For '.' operator initialize a separate method by using various stack functions do the same for the other operators like '*' and '+'.
7. Regular expression is in the form like a+b
8. Display the output
9. Stop.

PROGRAM:

```
transition_table = [[0]*3 for _ in range(20)]
re = input("Enter the regular expression:")
re += " "
i=0
j=1
while (i< len(re)):
    if re[i] == 'a':
        try
        if re[i+1] = '+' and
```

```
while (i < len(re)):
    if re[i] == 'a':
        try:
            if re[i+1] != '|' and re[i+1] != '*':
                transition_table[j][0] = j+1
                j += 1
                transition_table[j][0] =
            elif re[i+1] == '|' and re[i+2] == 'b':
                transition_table[j][2] = ((j+1)*10)+(j+3)
                j += 1
                transition_table[j][0] = j+1
                j += 1
                transition_table[j][2] = j+3
                j += 1
                transition_table[j][1] = j+1
                j += 1
                transition_table[j][1] = j+1
                j += 1
                transition_table[j][2] = ((j+1)*10)+(j+1)
                j += 1
        except:
            transition_table[j][0] = j+1
    elif re[i] == 'b':
        try:
            if re[i+1] != '|' and re[i+1] != '*':
                transition_table[j][1] = j+1
                j += 1
            elif re[i+1] == '|' and re[i+2] == 'a':
                transition_table[j][2] = ((j+1)*10)+(j+3)
                j += 1
                transition_table[j][1] = j+1
                j += 1
                transition_table[j][2] = j+3
                j += 1
```

```
transition_table [j] [1] = j+1
j += 1
transition_table [j] [2] = j+3
j += 1
transition_table [j][2] = j+1
j += 1
i = i+2

except:
    transition_table [j] [1] = j+1
elif re[i] == 'e' and re[i+1] != '|' and re[i+1] != '*':
    transition_table [j] [2] = j+1
    j += 1
elif re[i] == ')' and re[i+1] == '*':
    transition_table [0][2] = ((j+1) * 10) + 1
    transition_table [j] [2] = ((j+1) * 10) + 1
    j += 1
i += 1
print ("Transition function:")
for i in range (j):
    if (transition_table [i][0] != 0):
        print ("q[{0}] --> {1}",
                format (i, transition_table [i] [0]))
    if (transition_table [i] [2] != 0):
        if (transition_table [i] [2] != 0):
            print ("q[{0}], e] --> {1}",
                format (i, transition_table[i][2]))
        else
            print ("q[{0}, e] --> {1}, {2}",
                format (i, int(transition_table [i][2] /10),
                transition_table [i] [2] % 10))
```

RESULT

The program to convert regular expression to NFA was implemented successfully.

INPUT:

(a|b)* abb

OUTPUT

Transition Function

q [0, e] --> 1 & 7

q [2, a] --> 3

q [3, e] --> 6

q [4, b] --> 5

q [5, e] --> 6

q [6, e] --> 7 & 1

q [7, a] --> 8

q [8, b] --> 9

q [9, b] --> 10

1/2/23

EXPERIMENT - 3

AIM:

To write a program for converting NFA to DFA.

ALGORITHM:

ε NFA

1. Start
2. Get the input from the user
3. Set the only state in SDFA to "unmarked".
4. While SDFA contains an unmarked start do:
   a. Let T be that unmarked state
   b. for each a in I do, S = e-closure (MoveNFA(T,a))
   c. If S is not in SDFA already then, add S to SDFA (as an "unmarked" state)
   d. Set MoveDFA(T,a) to S.
5. For each S in SDFA if any s ∈ S is a final state in the NFA then, mark S an a final state in the DFA.
6. Print the result
7. Stop the program

PROGRAM

```
import pandas as pd
nfa = {}
n = int(input("No of states"))
t = int(input("No of transitions:"))
for i in range(n):
    state = input("state name:")
    nfa[state] = {}
    for j in range(t):
        path = input("path:")
```

INPUT

No of state : 3

No of transition : 2

state name : A

path : 0

Enter end state from A travelling through path 0

A

Path : 1

Enter end state from state A travelling through path 1:

A B

state name : B

path 0

Enter end state from state B travelling through path 0

C

path 1

Enter end state from state B travelling through path 1 :

C

state name : C

path 0

Enter end state from state C travelling through path 0

path 1

End state from C travelling through path 1:

```
        print("Enter end state from {} travelling
                through path {}:".format(state, path))
        reaching_state [x for x in input().split()]
        nfa [state][path] = reaching_state
print ("\n NFA :- \n")
print (nfa)
print ("\n Printing NFA table :-")
nfa_table = pd. DataFrame (nfa)
print (nfa_table. transpose())

print("Enter final state of NFA:")
nfa_final_state = [x for x in input(). split()]
new_state_list = []
dfa = {}
key_list = list(
        list (nfa. keys()) [0])
path_list = list (nfa [keys_list [0]]. key ())
dfa [key_list [0]] = {}
for y in range (l):
    var = "". join (nfa [key_list [0]] [path_list [y]])
    dfa [key_list [0]] [path_list [y]] = var
if var not in keys_list:
        new_states_list. append (var)
        keys_list . append (var)
while len(new_state_list) != 0:
        dfa [new_states_list [0]] = {}
        for l in range (len (new_state_list[0])):
            for i in range (len( path_list)):
                temp = []
```

NFA :-

{ 'A' : {'0' : ['A'], '1' : ['A', 'B']},

'B' : {'0' : ['C'], '1' ['C']}, 'C' : {'0 : [], '1' : []}

Printing NFA table

	0	1
A	[A]	[A, B]
B	[C]	[C]
C	[]	[]

Enter final state of NFA

C

~~Output~~

OUTPUT

DFA:

{'A' : {'0' : 'A', '1' : 'AB'}, 'AB' : {'0' : 'AC', '1' : 'ABC'},

'AC' : {'0' : 'A', '1' : 'AB'}, ~~'ABC' : {'0' : 'AC', '1' : 'ABC'}~~}

Printing DFA table:-

	0	1
A	A	AB
AB	AC	ABC
AC	A	AB
ABC	AC	ABC

Final state of the DFA are : ['AC', 'ABC']

```
    for j in range (len(new_states_list[0])):
        temp += nfa [new_state_list[0][j]][path_list[i]]

    s = " "
    s = s.join (temp)
    if s not in keys_list:
        new_states_list.appends(s)
        key_list.append(s)
    dfa [new_states_list[0]] [path_list[i]] = s
new_states_list.remove (new_states_list[0])
print ("\nDFA :- \n")
print (dfa)
print ("Printing DFA table :- ")
dfa_table = pd.DataFrame(dfa)
print (dfa_table.transpose())
dfa_states_list = list (dfa.keys())
dfa_final_states = []
for x in dfa_state_list:
    for i in x:
        if i in nfa_final_state:
            dfa_final_states.append(x)
            break
print ("\n Final state of the DFA are : ", dfa_final_states)
```

RESULT

The given NFA was converted to a DFA using python successfully.

11/2/22

8/2/23

EXPERIMENT 4

AIM:

To implement a program for elimination of left factoring and left recursion

ALGORITM FOR LEFT RECURSION:

1. Start the program
2. Initialize the array for taking input from the user
3. Prompt the user to input no of non-terminal having left recursion and no of production for these non-terminals
4. Prompt the user to input the production for non-terminals
5. Eliminate left recursion using the following rules.

   $A \rightarrow A\alpha 1 \mid A\alpha 2 \mid \ldots \mid A\alpha m$

   $A \rightarrow \beta 1 \mid \beta 2 \mid \ldots \mid \beta n$

   Then replace it by

   $A \rightarrow \beta i \; A' \; i = 1, 2, 3 \ldots n$

   $A' \rightarrow \alpha j \; A' \; j = 1, 2, 3 \ldots m$

   $A' \rightarrow \epsilon$
6. After eliminating the left recursion by applying these rules, display the ~~problem~~ production without left recursion
7. Stop.

ALGORITHM FOR LEFT FACTORING

1. Start
2. Ask the user to enter the set of production
3. Check for common symbol in the given set of production by comparing with :

$A \rightarrow \alpha\beta 1 \mid \alpha\beta 2$

4. If found replace the particular production with:

$A \rightarrow \alpha A'$

$A' \rightarrow \beta 1 \mid \beta 2 \mid \epsilon$

5. Display the output
6. Exit.

CODE FOR LEFT RECURSION:

```
gram = {
    "E": ["E+T", "T"],
    "T": ["T*F", "F"],

    "F": ["(E)", "i"]
}

def removeDirectLR(gramA, A):
    temp = gramA[A]
    tempCr = []
    tempInCr = []
    for i in temp:
        if i[0] == A:
            tempInCr.append(i[1:] + [A + "'"])
        else
            tempCr.append(i + [A "'"])
    tempInCr.append(["ε"])
    gramA[A + "'"] = tempInCr
    return gramA
```

INPUT

"E" : ["E+T", "T"]

"T" : ["T*F", "F"]

"F" : ["(E)", "i"]

OUTPUT

T → [["F", "T'"]]

F → [["F", "T'"]]

F → [["(", "E", ")"], "i"]]

E' → [["+", "T", "E'"], ["e"]]

T' → [["*", "F", "T'"], ["e"]]

```
def checkFor Indirect (gramA, a, ai):
    if ai not in gramA:
        return False
    if a == ai:
        return True
    if i[0] in gramA:
        return Check For Indirect (gramA, a, i[0])

    return False

def rep (gramA, A)
    temp = gramA[A]
    newTemp = []
    for i in temp:
        if check For Indirect( gramA, A, i[0]):
        t = []
        for K in gramA [i[0]]:
            t = []
            t += K
            t += i[1:]
            newTemp. append (t)
        else:
            new Temp. append (i)
    gramA[A] = newTemp
    return gramA

def rem (gram):
    c = 1
    conv = {}
    gramA = {}
    revconv = {}
    for g in gram:
        conv[g] = "A" + str(c)
        gramA = ["A" + str(c)] = []
```

```
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)

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
```

```
result = rem(gram)
for i in result:
    print(f'{i} -> result[i]')
```

CODE FOR LEFT FACTORING

```
from itertools import takewhile
def groupby(lst):
    d = {}
    l = [y[0] for y in rules]
    initial = list(set(l))
    for y in initial:
        for i in rule:
            if i.startswith(y):
                if y not in d:
                    d[y] = []
                d[y].append(i)
    return d
def prefix(x):
    return len(set(x)) == 1
starting = ""
rules = []
common = []
alphabet set = ["A", "B", "C", "D", "E", "F", "G", "H"
                "M", "N", "O", "P", "R", "S", "T", "U"
                "V", "W", "X", "Y", "Z"]
s = "S->iE+S|iEtSeS|a"
while (True):
    rules = []
    common = []
    split = s.split("->")
    starting = split[0]
```

INPUT

s = "S → iEtS | iEtSeS | a"

OUTPUT

$S \rightarrow aZ'$

$Z' \rightarrow \epsilon$

$S \rightarrow iEtSY'$

$Y' \rightarrow \epsilon \mid eS$

```
for i in split[1].split('|'):
        rules.append(i)

for k,l in groupby(rules).items():
    r = [d[0] for d in takewhile(prefix, zip(*l))]
    common.append(''.join(r))

for i in common:
    newalphabet = alphabetset.pop()
    print(starting +"->" + i + newalphabet)
    index = []
    for k in rules:
        if (k.startswith(i)):
            index.append(k)
    print(newalphabet + "->", end="")
    for j in index[:-1]:
        stringtoprint = j.replace(i,"",1) + "|"
        if stringtoprint == "|":
            print("\u03B5","", end="")
        else
            ~~print("\u03B5","|", end="")~~
            print(j.replace(i,"",1) + "|", end="")
    stringtoprint = index[-1].replace(i,"",1) + "|"
    if stringtoprint == "|":
        print("\u03B5","", end="")
    else
        print(index[-1].replace(i,"",1)+"", end="")
    print("")
```

break

RESULT:

The given programs are successfully executed.

EXPERIMENT - 5

15/2/23

AIM

To write a program to perform first and follow using any language.

ALGORITHM

- For computing the first

1. If X is a terminal then FIRST(X) = {X}

   Example F → I | id

   We can write it as FIRST(F) → { (, id)

2. If X is non terminal like E → T then FIRST substitute T with other production until you get a terminal as the first symbol

3. If X → ∈ then add ∈ to FIRST(X)

- FOR computing the follow:

1. Always check the right side of the production for a non terminal whose FOLLOW set is being found

2. (a) If that non-terminal (S, A, B ...) is followed by any terminal (a, b ..., *, +, (,),) then add that terminal into the FOLLOW set

   b) If that non-terminal is followed by any other non-terminal then add FIRST of the other non terminal into the FOLLOW set

CODE

```
import sys
sys.setrecursionlimit(60)

def first(string):
    first = set()
    if string in non-terminals:
        alternative = production_dict[string
        for alternative in alternatives:
            first_2 = first(alternative)
            first_i = first_i | first_2
    elif string in terminal
        first_ = {string}
    elif string == '' or string == '@':
        first_ = {'@'}
    else:
        first_2 = first(string[0])
        if '@' in first_2:
            i = 1
            while '@' in first_2:
                first_ = first_ | first_2 - {'@'}
                if string[i] in terminals:
                    first_ = first_ | {string[i:]}
                    break
                elif string[i:] == '':
                    first_ = first_ | {'@'}
                    break
                first_2 = first(string[i:])
                first_ = first_ | first_2 - {'@'}
                i += 1
```

INPUT:

Enter no of terminals: 3

Enter the terminal:

a
b
c

Enter the no of non terminals: 3

Enter the non terminals:

A
B
C

Enter the starting symbol: A

Enter no of production: 5

Enter the production

A: → aA

B → bB

C → aC

B → bC

C → a

```
    else:
        first_ = first_ | first_2
return first_

def follow(nT):
    follow_ = set()
    prods = production_dict.items()
    if nT == starting_symbol:
        follow_ = follow_ | {'$'}

    for nt, rhs in prods:
        for alt in rhs:
            for char in alt:
                if char == nT:
                    following_str = alt[alt.index(char)+1:]
                    if following_str == "":
                        if nt == nT:
                            continue
                        else:
                            follow_ = follow_ | follow(nt)
                    else
                        follow_2 = first(following_str)
                        if '@' in follow_2:
                            follow_ = follow_ | follow_2 - {'@'}
                            follow_ = follow_ | follow(nt)
                        else
                            follow_ = follow_ | follow_2
return follow_
```

```
no_of_terminals = int (input ("Enter no. of terminals: "))
terminal = []
print ("Enter the no of terminal: "))
for _ in range (no_of terminals):
        terminal. append ( Input()):

no_of_non_terminal = int (input ( Enter of no. of
                                    non terminal"))

non_terminal = []
print ("Enter the non terminals: ")
for _ in range (no_of non_terminal):
        non_terminal. append ( input())
production_dict = {}

for nT in non_terminals:
        production_dict[nT] = []

for production in productions:
        nonterm_to_prod = production.split ("->")
        alternatives = non term_to_prod [1]. split("/")
        for alternative in alternative
                production_dict [non term_to_prod [0]].
                                        append (alternatives)

FIRST = {}
FOLLOW = {}
```

```
for non-terminal in non-terminals:
    FIRST [non-terminal] = set()

for non-terminal in non-terminal:
    FOLLOW [non-terminal] = set()

for non terminal in non-terminals:
    FIRST [non-terminal] = FIRST [non-terminal] | first
                                                  {non-termine

FOLLOW [starting_symbol] = FOLLOW [starting-symbol] | {$}

print ("{: ^20} {: ^20} {: ^20}".format('Non terminals',
                                        'First', 'Follow'))
for non-terminal in non terminals:
    print ("{: ^20" {: ^20} {: ^20}".format(non termino
                                    str(FIRST(non-terminal),
                                    str(FOLLOW[non terminal]))
```

RESULT

The ~~first~~ FIRST and FOLLOW set of non-terminal of a grammar were found ~~and~~ succesfully using python language

15/2

22-02-2023

EXPERIMENT 6

PREDICTIVE PARSING TABLE

AIM:

To write a program for Predictive Parsing Table

ALGORITHM:

For the production $A \rightarrow \alpha$ of Grammer G

- For each terminal, a is FIRST (α) add $A \rightarrow \alpha$ to M[A, a]
- If ϵ is in FIRST (α) and b in FOLLOW (A) then add $A \rightarrow \alpha$ to M[A, b]
- If ϵ is in FIRST (α) and \$ is in FOLLOW (A) then add $A \rightarrow \alpha$ to M[A, \$]
- All remaining entries in Table M are errors

PROGRAM:

```
#include <stdio.h>
#include <string.h>
char pod [7][10] = {"S", "A","A", "B","B", "C", "C"};
char pror [7][10] = {"A", "Bb", "Cd", "aB", "@", "Cc", "@"
char prod [7][10] = {"S->A", "A->Bb", "A->Cd", "B->aB",
                     "B->@", "C->Cc", "C->@"};

char first[7][10] = {"abcd", "ab", "cd", "a@", "@", "c@", "c
char follow[7][10] = {"$", "$", "$", "$", "b$", "c$", "d$"};
char table[5][6][10];
```

```
int numr ( char c) {
    switch (c) {
        case 'S':
            return 0;
        case 'A':
            return 1;
        case 'B':
            return 2;
        case 'C':
            return 3;
        case 'a':
            return 0;
        case 'b':
            return 1;
        case 'c':
            return 2;
        case 'd'
            return 3;
        case 'f'
            return 4;
        }
        return (2);
    }
int main()
{ int i, j, k;
    for (i=0; i<5; i++)
        for (j=0 ; j<6; j++)
            strcpy (table [i] [j], "");
    printf("The following grammar is used for Parsing
        Table: ");

    for (i=0 ; i<7; i++)
        printf("%s \n", prod [i]);
```

INPUT

enter the no of non-terminals

6

Enter the production in the grammar

~~E → E+T | T~~
~~T → T*F | F~~
~~F → (E) | id~~

S → aBDh
B → cC
C → bC | e
D → EF
E → g | e
F → f | e

OUTPUT

first

FIRST[S] = a
FIRST[B] = c
FIRST[C] = b e
FIRST[D] = g e
FIRST[E] = g e
FIRST[F] = f e

follow

FOLLOW[S] = $
FOLLOW[B] = g e $
FOLLOW[C] = g e $
FOLLOW[D] = h
FOLLOW[E] = f e
FOLLOW[F] = h

```
printf(" Predictive Parsing Table");
fflush (stdin);
for (i=0 ; i<7; i++) {
    k = strlen (first[i]);
    for (j=0; j<10; j++)
        if first[i][j] != '@')
        strcpy (table [numr (prod [i][0]+1]
                        [numr(first [i][j]+1], prod[i]);
    }
for (i=0; i<7; i++) {
    if (prod[i][0] == '@') {
        k = strlen (follow [i]);
        for (j=0; j<k; j++)
            strcpy (table [numr(prod [i][0]) +1]
                    [numr (follow [i][j] + 1], prod[i]);
    }
}
}
strcpy (table [0][0], " ");
strcpy (table[0][1], "a");
strcpy (table [0][2], "b");
strcpy (table [0][3], "c");
strcpy (table[0] [4], "d");
strcpy (table[0][5], "$");
strcpy (table[1][0] , "S");
strcpy (table [2][0], "A");
strcpy (table [3][0], " B");
strcpy (table[4][0], "C");
```

~~M[B,c]=~~

M[S,a] = S → aBDh

M[B,c] = B → cC

M[C,e] = C → e

M[D,g] = D → EF

M[D,e] = D → EF

M[E,g] = E → g

M[E,e] = E → e

M[F,f] = F → f

M[F,e] = F → e

	a	g	c	f	h	b
S	S→aBDh					
B			B→cC			
C					C→e	C→
D		D→EF			D→e	
E		E→g		E→e	E→e	
				F→f	F→e	

o/p verified

```
for (i=0; i<5; i++)
    for (j=0; j<6; j++) {
        printf("%-10s", table[i][j]);
        if (j==5)
            printf("\n - - - - - - - - - \n");
        }
    }
```

RESULT

The implementation and creation of predictive parse table using c was executed successfully

22/2

5/3/23

EXPERIMENT - 7

Shift Reduce Parsing

AIM:

To write a program to implement lexical Analysis using C

ALGORITHM:

- Shift reduce parsing is a process of reducing a string to the start symbol of a grammer
- Shift reduce parsing use a stack to hold the grammer and an input tape to hold the string
- Sift reduce parsing performs the two actions: shift and reduce. That's why it is known as shift reduce parsing
- At the shift action, the current symbol in the input string pushed to a stack.
- At each reduction, the symbols will replaced by the non-terminals.

The symbol is the right side of the production and non terminal is the left side of the production

PROGRAM:

```
#include <stdio.h>
#include<string.h>
int k=0, z=0, n=0, j=0, c=0;
char a[16], ac[20], stk[15], act[10];
void check();
```

INPUT:

"E" : ["E*E", "E+E", "i"]

starting_terminal = "E"

inp = "i+i*i"

OUTPUT:

Stack	Input Buffer	Parsing Action
$i	+i*i	Shift
$E	+i*i	Reduce S→i
$E+	i*i	Shift
$E+i	*i	Shift
$E+E	*i	Reduce S→i
$E*	i	Reduce S→E+E
$E*	i	Shift
$E*	i	Rejected.

```
for(z=0; z<c; z++)
  if (stk[z] == '(' && stk[z+1] == 'E'
                    && stk[z+2] == ')')
  {
    stk[z] = 'E';      stk[z+1] = '\0';
    stk[z+2] = '\0';
    printf("\n $ %s \t %s $ \t %s", stk, a, ac);
      i = i-2
  }
}
```

RESULT:-

The implementation of shift reduce parsing was executed and verified successfully

9/3/23

EXPERIMENT 8

AIM:

To write a program to compute of lead and Trail

ALGORITHM:

1. For leading, check for the first non-terminal
2. If found, print it.
3. Look for next production for the same non-terminal
4. If not found, recursively call the procedure for the single non-terminal present before the comma or End of production string.
5. Include it's results in the result of this non-terminal.
6. For trailing, we compute same as leading but we start from the end of the production to the beginning
7. Stop

PROGRAM:

```
#include<iostream>
#include<string.h>
#include<stdlib.h>
using namespace std;
int vars, terms, i,j,k,m, rep, count, temp=-1;
char var[10], term[10], lead[10][10], trail[10][10];
```

```
struct grammer {
    int prodno;
    char lhs; rhs [20][20];
} gram [50];
void get () {
    cout << "\n LEADING AND TRAILING \n";
    cout << "\n Enter the no. of variables: ";
    cin>> vars;
    cout<< "\n Enter the variable: \n";
  for (i=0; i<vars; i++) {
      cin>> gram [i] .lhs;
      var [i] = gram [i] .lhs;
  }
  cout<<" \n Enter the no. of terminals: ";
  cin>> terms;
  cout<<" \n Enter the terminals: ";
  for (j=0; j<terms; j++)
        cin>> term [j];
   cout<<" \n PRODUCTION DETAILS \n";
  }
void leading () {
    for (i=0; i<vars; i++) {
        for (j=0; j<gram[i].prodno; j++) {
            for (k=0; k<terms; k++) {
                if (gram [i].rhs [j][0] == term [k])
                    lead [i][k] = 1;
                else {
                    if (gram [i]. rhs[j][1] == term [k])
                        lead [i][k] = 1;
                    }
            } }
    }
```

INPUT

Enter the no of Production: 6

Enter the production one by one

$E \rightarrow E+E$

$E \rightarrow T$

$T \rightarrow T*F$

$F \rightarrow (E)$

$T \rightarrow F$

$F \rightarrow i$

```
for ( rep=0 ; rep < vars ; rep++) {
        for (i=0; i< vars; i++) {
            for(j=0; j<gram[i].prod no; j++){
                for ( m=1; m < vars ;m ++) {
                    if (gram[i].rhs[j][0] ==var[m]){
                            term=m
                            goto out;
                    }   }
                out :
                for ( k=0; k < terms; k++) {
                        if (lead[temp][k] ==1)
                                lead [i][k] = 1;
                    }
                }
            }
        }
    }
void trailing() {
        for (i=0; i< vars; i++) {
            for (j=0; j<gram[i].prod no; j++) {
                count =0;
                while (gram[i].rhs [j][count] != '\x0')
                        count ++;
                for (k=0; k < terms; k++) {
                        if (gram [i].rhs [j][count-2]==term[k]
                                trail [i][k] =1;
                    }   }
            }
        }
```

OUTPUT

LEADING[E] = {+, *, (, i}

LEADING[T] = {*, (, i}

LEADING[F] = {(, i}

TRAILING[E] = {+, *,), i}

TRAILING[T] = {*,), i}

TRAILING[F] = {), i}

```
int main () {
    get ();
    leading ();
    trailing ();
    display ();
}
```

RESULT:

The program to find lead and trail was successfully compiled and run

16/3/23

EXPERIMENT -9

Computation of LR(0) item

AIM:

To write a program to implement LR(0) items

ALGORITHM:-

1. Start
2. Create structure for production with RHS and LHS
3. open file and read input file
4. Build state 0 from extra grammer Law $S' \rightarrow S \$$ that is all start symbol of grammer and one dot(.) before S symbol
5. If Dot symbol is before a non-terminal, add grammer law that this non-terminal is left hand side of that Law and set Dot in before of first part of RHS.
6. If state exist, used that instead
7. Now find set of terminal and non-terminal in which Dot exist in before
8. If step 7 set is non empty goto 9, else go to 10
9. For each terminal / non-terminal in set step 7 create new state by using all grammer law that Dot position is before of that terminal / non-terminal
10. Go to step 5
11. End of state building
12. Display the output
13. End

PROGRAM:

```
#include<iostream>
#include<conio.h>
#include<string.h>
using namespace std;
char prod[20][20], listofvar[20] = "ABCDEFGHIJKLMNOPQR";
int novar=1, i=0, k=0, n=0, m=0, arr[30];
int noitem=0;
```

~~Enter~~

INPUT:

ENTER THE PRODUCTIONS OF THE GRAMMER (0 TO END)

E → E+T

E → T

T → T*F

T → F

F → (E)

F → i

OUTPUT

A → E

E → E+T

E → T

T → T*F

F → (E)

F → i

THE SET OF ITEMS ARE

I_0

A → ·E

E → ·E+T

E → ·T

T → ·T*F

F → ·(E)

F → ·i

I_1

$A \to E$

$E \to E. + T$

I_2

$E \to T.$

$T \to T.*F$

I_3

$F \to (.E)$

$E \to .E + T$

$E \to .T$

$T \to .T*F$

I_4

$F \to i.$

I_5

$E \to E + .T$

$T \to .T*F$

I_6

$T \to T*.F$

$F \to .(E)$

$F \to .i$

I_8

$E \to E + T.$

$T \to T.*F$

I_9

$T \to T*F.$

I_{10}

$F \to (E).$

```
for (k=3; k<strlen(prod[n]); k++) {
    if (prod[n][k]=='|')
        g[j].lhs[m++] = prod[n][k];
        if (prod[n][k]=='|') {
            g[j].rhs[m] = '\0';
            m=0;
            j = novar;
            g[novar++].lhs = prod[n][0];
        }
    }
}
for (i=0; i<26; i++)
        if (!isvariable(listofvar[i]))
                break;
g[0].lhs= listofvar[i];
char temp[2] = {g[i].lhs,'\0'};
strcat(g[0].rhs, temp);
cout<<"augumented grammer";
for (i=0; i<novar; i++)
  {
    clos[noitem][i].lhs = g[i].rhs;
    strcpy(clos[noitem][i].rhs, g[i].rhs);
    if(strcmp(clos[noitem][i].rhs,"ε")==0)
            strcpy(clos[noitem][i].rhs ".");

for(int z=0; z<noitem; z++) {
    char list[10];
    int l=0;
    for (y=0; y<car[z]; y++) {
            for (k=0; k<strlen(clos[z][y].rhs)-1;
                                        k++)
                if (clos[z][y].rhs[k]=='.')
```

```
        if (m==1)
                lhs[l+1] = clos[z][g].rhs[k+1];
        }
    }
}
cout << "THE SET OF ITEMS ARE";
for (int z=0; z<noitem; z++) {
    cout << "I" << z << "\n";
    for (g=0; g<arr[z]; g++)
        cout << clos[z][g].lhs << "->" << clos[z][g].rhs
                                << "\n";
    }
}
```

RESULT:

The program for computation of LR[0] was successfully compiled and run

8/23

EXPERIMENT - 10

Intermediate Code Generation - Postfix, Prefix

M!

A program to implement Intermediate Code generation- Postfix, Prefix.

ALGORITHM

Declare set of operators

Initialize an empty stack

To convert INFIX to POSTFIX follow the following steps

4. Scan the infix expression from left to right

5. ~~Scan the infix expression from le~~
If the scanned character is an operand, output it.

6. Else, If the precedence of the scanned operator is greater than the precedence of the operator in the stack

7. Else, Pop all the operators from the stack, which are greater than ~~or~~ equal to in precedence than that of the scanned operator

8. If the scanned character is an '(', push it to the stack.

9. If the scanned character ')', pop the stack and output it until a '(' is encountered and discard both the parathesis

10. Pop and output from the stack until it is not empty

11. To convert INFIX to PREFIX follow the steps

12. Scan the expression from left to right

14. Whenever the operands arrive, print them

15. Repeat step 6 to 9 until the stack is empty

PROGRAM

```
OPERATOR= set ( ['+', '-', '*', '/', '(', ')'])

def infix_to_postfix (formula):-
    stack = []
    output = " "

    for ch in formula:
        if ch not in OPERATORS:
            output += ch
        elif ch == "(":
            stack.append ('(')
        elif ch == ')':
            while stack and stack [-1] != '(':
                output += stack.pop()

        else :
            while stack and stack [-1] != '(' and
                                PRI [ch] <= PRI [stack [-1]]:
                output += stack.pop()
            stack.append (ch)

    while stack:
        output += stack.pop()
    print (f' POSTFIX : {output}')
    return output

def infix_to_prefix (formula):
    op_stack = []
    exp_stack = []
    for ch in formula:
        if not ch in OPERATOR:-
            exp_stack.append(ch)
        elif ch == '(':
            op_stack.append(ch)
        elif ch == ')'
```

INPUT:

INPUT THE EXPRESSION:

(a+b) * (c+d) / r

~~PREFIX: /*+ab+cdr~~

~~POSTFIX: abtcd+*r/~~

OUTPUT:

PREFIX: /*+ab+cdr

POSTFIX: ab+cd+*r/

```
while op_stack [-1] != '('
        op = op_stack.pop()
        a = exp_stack.pop()
        b = exp_stack.pop()
        exp_stack.append(op+b+a)
    op_stack.pop()
print(f'PREFIX : {exp_stack[-1]}')
return exp_stack[-1]
```

pre = infix_to_prefix(expres)
pos = infix_to_postfix(expres)

RESULT

The program was successfully compiled and run.

27/3

3/4/23 EXPERIMENT - 11

Intermediate Code Generation
Quadruples, Triples Indirect Triple

AIM:
To implement Code generation - Quadruple, Triple indirect triple

ALGORITHM:
The algorithm take a sequence of three address statement as input.
for each three address statements as input
For each three address statement of the form
a:=b op perform the various actions

1. Invoke a function get reg to find out the location L where the result of computation b op c should be stored

2. Consult the address description for y to determine y'.
If the value of y currently in memory and register both then prefer the register y'.
If the value of y is not already in L.
then generate the instruction MOV y', L to place a copy of y in L.

3. If the current value of y or z have no next uses or not live on exit from the block or in register then alter the register descriptor to indicate that after execution of x = y op z those register will no longer contain y or z

INPUT

(1) Enter the expression:

a= b+ c*d

2) Enter the expression:

a = b*-c + b*-c

OUTPUT:

(1)

The intermediate code is

t1 = c*d

t2 = b+t1

a = t2

2) The intermediate code is

t1 = b*-

t2 = b*-

t3 = c+t1

t4 = 1-c1

t5 = 2-t3

a = t5

PROCEDURE:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void small();
void clove (int i);
int p[5] = {0,1,2,3,4}, c = l, i, k, l, m, ip;
char sw[5] = {'=','-','+','/','*'},
                g[20], a[5], b[5], ch[2];

void main() {
  printf (" Enter the expression");
 scanf ("%s",g);
 printf("\t The intermediate code is: \n");
 small();
}
void clove (int i){
 a[0] = b[0] = '\0';

if ( !isdigit(g[i+2] && !isdigit(g[i-2])) {
   ~~a[0] = b[0] = '\0';~~
   a[0] = g[i-1];
   b[0] = g[i+1];
}
if (isdigit (g[i+2])) {
   a[0] = g[i-1];
   b[0] = 't';
  b[1] = g[i+2];
}
```

INPUT THE EXPRESSION : a = b * -c + b * -c

PREFIX : - + a * - * = b c b c

POSTFIX : a = b * c - b * + c -

THREE ADDRESS CODE GENERATION

t1 = - * b

t2 = t1 - c

t3 = t2 * b

t4 = a + t3

t5 = t4 - c

t = uminus * c

t = unary minus c

The quadruple for the expression

OP	ARG1	ARG2	RESULT
- *	=	b	- t(1)
-	c	(-)	= t(2)
+	t(1)	t(2)	t(3)
*	t3	b	t(4)
+	a	t(3)	t4
-	c	(-)	t(5)
+	t(5)	t(6)	t(6)
			t(7)

(e

```
if (g[i] == sw[m])
    if (pi <= rp[m]) {
        pi = p[m];
        l=1 }
        k = i;
    }
}
if (l==1)
    dove (k);
else
    exit(0);
}
```

RESULT

The program was successfully compiled and run

31/9/23

10/11/23

EXPERIMENT - 12

AIM:

To write a code for simple code generator

ALGORITHM:

1. Take user inputs for the name of the class and the number of fields.
2. Initialize an empty list called field.
3. Loop through the range of the number of field
   - Take user input for the name and type of each field
   - Append a tuple containing the field name and type to the field list
4. Print generated code to console

CODE

```
class_name = input("Enter the name of your class: ")
num_field = int(input("Enter the number of field for your class"))
fields = []

for i in range(num_fields):
    field_name = input(f"Enter name for field {i+1}: ")
    field_type = input(f"Enter type for field {i+1}: ")
    field.append(field_name, field_type))
```

INPUT:

Enter the name of your class : Person

Enter the number of fields : 4

gen code ('a', 'b', 'ADD', 'c') → a = b + c

gen code ('d', 'a', 'SUB', 'b') → d = a − c

gen_code ('x', 'y', 'MUL', 'z') → x = y * z

OUTPUT

SUB R1, R2

MOV y, R2

MOV z, R3

MUL R2, R3

```
if op == '+'
    machine_instruction.append(f"ADD {arg1}, {arg2}
                                {result}")
elif op == '*':
    machine_instruct.append(f"MUL {arg1}, {arg2}, {result}
elif op == '/':
    machine_instruct.append(f"DIV {arg1}, {arg2}, {result}"
elif op == '=':
    machine_instruction.append(f"MOV {arg1}, {result}")

def read_threeaddress_code()
    three_address_code = []
    while True:
        instruction = input:
        if instruction == 'done':
            break
        part = instruction.split()
        op = part[1]
        arg1 = parts[2]
        arg2 = part[3]
        result = part[4]
        three_address_code.append((op, arg1, arg2, result))

machine_instructions =
                generate_machine_instruction
                (three_address_code)
```

INPUT

2 + 3 = x

x + 3 = y

2 * y = z

~~CFGT~~

OUTPUT

Generated machine instruction

ADD 2 , 3, x

ADD x , 3, y

MUL 2, y , z

```
print ("Generated machine instructions:)
for instruction in machine_instructions
    print (instruction)
```

~~print~~ RESULT

The following code was implemented for a simple code generator.

10/4

18/4/23 EXPERIMENT 13

AIM:

A program to Implementation of DAG

Algorithm:-

1. The leaves of a graph are labeled by a unique identifier can be variable name or constant
2. Interior nodes of the graph are labeled by an operator symbol.
3. Node are also given a sequence of identifier for labels to store the computed value
4. If y operand is un defined then create node(y)
5. If z operand is undefined then for case(i) create node (z)
6. For case(i) create node (OP) whose right child is node (z) and left child is node(y)
7. For case(ii) check whether there is node(OP) which one child node(y)
8. For case (iii), node n will be node(y)
9. For node(x) delete x from the list of identifiers. Append x to attached identifiers list for the node n found in step2.

Finally set node(x) to n

Code:

```
#include <iostream>
#include<string>
#include<unordered_map>
Using namespace std;
Class DAG {
    public:
        char label;   Char data;
        DAG* left;   DAG* right;
```

INPUT

1

A = x+y

B = A* z

C = B/x

OUTPUT

LABEL	Ptr	left Ptr	right Ptr
A	+	x	y
B	*	A	z
C	/	B	x

```
DAG (char x) {
    label = '-';    data = x;
    left = NULL;    right = NULL;
}
DAG ( char lb, char x, DAG* lt, DAG* rt) {
    label = lb;    data = x;
    left = lt;    right = rt;
}
};
int main () {
    int n;
    n = 3;
    string st[n];
    st[0] = "A = x + y";
    st[1] = "B = A * z";
    st[2] = "C = B / x";
    unordered_map < char, DAG* > labelDAGNode;
    for (int i=0; i<3; i++) {
        string stTemp = st[i];
        for (int j=0; j<5; j++) {
            char tempLabel = stTemp[0];
            char tempLeft = stTemp[2];
            char tempData = stTemp[3];
            char tempRight = stTemp[4];
            DAG * leftPtr;
            DAG * rightPtr;
            if (labelDAGNode.count(tempLeft) == 0) {
                leftPtr = new DAG (tempLeft);
            }
            else {
                leftPtr = labelDAGNode [tempLeft];
            }
            if (labelDAGNode.count(tempRight) == 0) {
                rightPtr = new DAG (tempRight);
            }
```

```
    else {
        rightPtr = labelDAGNode[tempRight];
    }

DAG * nn = new DAG (templabel, tempData, leftPtr, rightPtr);
labelDAGNode.insert ( make_pair (templabel, nn));
    }
}

cout << " Label    Ptr    leftPtr    rightPtr "<<endl;
for (int i=0; i<n; i++)
        DAG * x = labelDAGNode[sA[i][0]];
    if (x -> left -label == '_')
            cout<< x -> right -> data;
    else
        cout << x -> left -> label;
    cout << "    ";
    if (x -> right -> label == '_')
                cout << x -> right -> data;
    cout << endl;
    }
return 0;
}
```

RESULT

The program was successfully compiled and run

17/1