**AMITY SCHOOL OF ENGINEERING AND TECHNOLOGY**

**NOIDA, UTTAR PRADESH**



**COMPILER CONSTRUCTION LAB FILE**

**(CSE304)**

**Submitted To - Submitted By-**

Dr. Juhi Singh Ananya Gupta

Associate Professor A2305222350

ASET 6CSE5-Y

**INDEX**

|  |  |  |  |
| --- | --- | --- | --- |
| **S.No.** | **Experiment** | **Date** | **Sign** |
| 1. | Write a program in C to check if given string is accepted or rejected by production rules. S → aS | Sb | ab. |  |  |
| 2. | WAP to convert infix equation to postfix equation using stack implementation |  |  |
| 3. | WAP in C/C++ to count number of tokens in the given program. |  |  |
| 4. | WAP in C/C++ to remove the left recursion from the given grammar. |  |  |
| 5. | WAP in C/C++ to convert infix expression to prefix expression. |  |  |
| 6. | WAP in C/C++ to convert regular expressions to Finite Automata. |  |  |
| 7. | Write a program in C/C++ to convert NFA to DFA. |  |  |
| 8. | Write a program in C/C++ to find first and follow of Given CFG. |  |  |
| 9. | Write a program in C/C++ to design shift reduce parser for: E -> E + E | E \* E | (E) | Id. |  |  |
| 10. | Write a program which accepts regular grammar with no left-recursion, and no null- production rules, and then it accepts a string and reports whether the string is accepted by the grammar or not. |  |  |
| 11. | Consider the following regular expressions:   1. (0 + 1)\* + 0\*1\* 2. (ab\*c + (def)+ + a\*d+e)+ 3. ((a + b)\*(c + d)\*)+ + ab\*c\*d   Write separate programs for each of the regular expressions mentioned above |  |  |
| 12. | Write a program for Recursive Descent Calculator. |  |  |
| 13. | Consider the following grammar: S --> ABC A--> abA | ab B--> b | BC C--> c | cC  Following any suitable parsing technique (prefer top-down), design a parser which accepts a string and tells whether the string is accepted by above grammar or not. |  |  |
| 14. | Design a parser which accepts a mathematical expression (containing integers only). If the expression is valid, then evaluate the expression else report that the expression is invalid. |  |  |
| 15. | Open Ended program: Designing of various types of parser. |  |  |

# ***LAB ASSIGNMENT-1***

## **Aim:** Write a program in C to check if given string is accepted or rejected by production rules. S → aS | Sb | ab.

**Theory:**

Understanding the Grammar Rules:

1. Base Case:
   * The simplest valid string is "ab".
2. Recursive Production:
   * S → aS: Means 'a' can be followed by another valid S (another "ab" structure or a recursive pattern).
   * S → Sb: Means 'S' can be followed by 'b', which allows multiple trailing 'b's.
   * S → ab: This is the base structure.

## **Production Rules:**

1. S → aS
2. S → Sb
3. S → ab

## **Program:**

#include <stdio.h>

#include <string.h>

#include <stdbool.h>

bool isValidProduction(char \*str, int start, int end) {

if (start >= end - 1) {

return (str[start] == 'a' && str[end] == 'b');

}

if (str[start] == 'a' && isValidProduction(str, start + 1, end)) {

return true;

}

if (str[end] == 'b' && isValidProduction(str, start, end - 1)) {

return true;

}

return false;

}

int main() {

char str[100];

printf("Enter the string to check: ");

scanf("%s", str);

int length = strlen(str);

if (isValidProduction(str, 0, length - 1)){

printf("Accepted\n");

}

else {

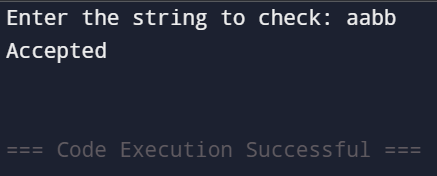
printf("Rejected\n");

}

return 0;

}

**Output:**



# ***LAB ASSIGNMENT-2***

## **Aim:** WAP to convert infix equation to postfix equation using stack implementation.

## **Theory:**

Infix Notation:

An infix expression is a mathematical expression where the operator is placed between operands (e.g., A + B). While humans find this easy to read, computers struggle with operator precedence and associativity.

Postfix Notation (Reverse Polish Notation - RPN):

A postfix expression places the operator after operands (e.g., A B +). This format eliminates the need for parentheses and follows a clear left-to-right evaluation.

## **Program:**

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <string.h>

#define MAX 100

char stack[MAX];

int top = -1;

void push(char c) {

stack[++top] = c;

}

char pop() {

return stack[top--];

}

int precedence(char c) {

if (c == '^')

return 3;

if (c == '\*' || c == '/')

return 2;

if (c == '+' || c == '-')

return 1;

return -1;

}

void infixToPostfix(char \*infix, char \*postfix) {

int i, j = 0;

for (i = 0; infix[i] != '\0'; i++) {

if (isalnum(infix[i])) {

postfix[j++] = infix[i];

}

else if (infix[i] == '(') {

push(infix[i]);

}

else if (infix[i] == ')') {

while (top != -1 && stack[top] != '(') {

postfix[j++] = pop();

}

pop();

}

else {

while (top != -1 && precedence(stack[top]) >= precedence(infix[i])) {

postfix[j++] = pop();

}

push(infix[i]);

}

}

while (top != -1) {

postfix[j++] = pop();

}

postfix[j] = '\0';

}

int main() {

char infix[MAX], postfix[MAX];

printf("Enter infix expression: ");

scanf("%s", infix);

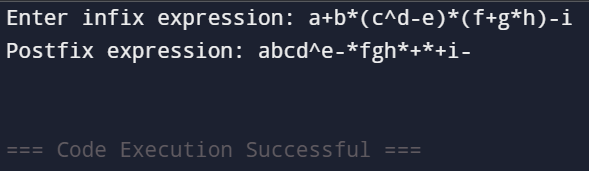
infixToPostfix(infix, postfix);

printf("Postfix expression: %s\n", postfix);

return 0;

}

**Output:**



# ***LAB ASSIGNMENT-3***

## **Aim:** WAP in C/C++ to count number of tokens in the given program.

## **Theory:**

Definition of a Token:

A token is the smallest meaningful unit in a programming language. These can include keywords, identifiers, operators, literals, punctuation, and special symbols.

Types of Tokens in a Program

1. Keywords: Reserved words like int, return, if, while, etc.
2. Identifiers: Variable names, function names (main, sum, num1, etc.).
3. Operators: +, -, \*, /, =, ==, etc.
4. Literals: Numeric constants (10, 3.14) or string constants ("Hello").
5. Special Symbols: {, }, (, ), ;, ,, etc.

## **Program:**

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <string.h>

#define MAX 100

int isDelimiter(char ch) {

return (ch == ' ' || ch == '+' || ch == '-' || ch == '\*' || ch == '/' ||

ch == ',' || ch == ';' || ch == '>' || ch == '<' || ch == '=' ||

ch == '(' || ch == ')' || ch == '[' || ch == ']' ||

ch == '{' || ch == '}');

}

int countTokens(const char \*str) {

int count = 0;

int i = 0;

int length = strlen(str);

while (i < length) {

while (i < length && str[i] == ' ') {

i++;

}

if (isalpha(str[i])) {

while (i < length && (isalnum(str[i]) || str[i] == '\_')) {

i++;

}

count++;

}

else if (isdigit(str[i])) {

while (i < length && isdigit(str[i])) {

i++;

}

count++;

}

else if (isDelimiter(str[i])) {

if ((str[i] == '=' || str[i] == '>' || str[i] == '<' || str[i] == '!') && str[i + 1] == '=') {

i += 2;

} else {

i++;

}

count++;

}

else {

i++;

}

}

return count;

}

int main() {

char input[MAX];

printf("Enter a program string: ");

fgets(input, MAX, stdin);

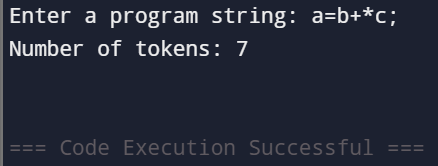
input[strcspn(input, "\n")] = '\0';

printf("Number of tokens: %d\n", countTokens(input));

return 0;

}

**Output:**



# ***LAB ASSIGNMENT- 4***

## **Aim:** WAP in C/C++ to remove the left recursion from the given grammar.

## **Theory:**

Understanding Left Recursion

A grammar is said to have left recursion if a non-terminal symbol (e.g., A) appears as the first symbol on the right-hand side of its own production rule.

Problems with Left Recursion

* Left recursion causes infinite recursion in top-down parsers (like recursive descent parsing).
* It must be removed before implementing such parsers. Approach in C/C++

1. Parse the input grammar rule.
2. Identify left recursion (if A appears first on the right-hand side).
3. Apply transformation to remove recursion.
4. Display the modified grammar.

## Program:

#include <stdio.h>

#include <string.h>

void removeLeftRecursion(char nonTerminal, char alpha[], char beta[]) {

printf("Grammar after removing left recursion:\n");

printf("%c -> %s%c'\n", nonTerminal, beta, nonTerminal);

printf("%c' -> %s%c' | ε\n", nonTerminal, alpha, nonTerminal);

}

int main() {

char nonTerminal, alpha[10], beta[10], grammar[20];

printf("Enter the grammar (Format: A->Aα|β): ");

scanf("%s", grammar);

nonTerminal = grammar[0];

int i = 3, j = 0, k = 0;

char firstProd[10], secondProd[10];

while (grammar[i] != '|' && grammar[i] != '\0') {

firstProd[j++] = grammar[i++];

}

firstProd[j] = '\0';

if (grammar[i] == '|') {

i++;

}

while (grammar[i] != '\0') {

secondProd[k++] = grammar[i++];

}

secondProd[k] = '\0';

if (firstProd[0] == nonTerminal) {

removeLeftRecursion(nonTerminal, firstProd + 1, secondProd);

}

else {

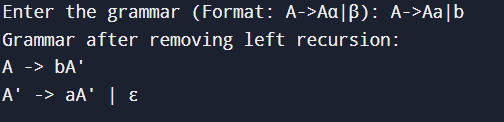
printf("No left recursion detected.\n");

}

return 0;

}

**Output:**



# ***LAB ASSIGNMENT-5***

## **Aim:** WAP in C/C++ to convert infix expression to prefix expression.

## **Theory:**

Understanding Infix, Postfix, and Prefix Notations

* Infix Notation: Operators are placed between operands.
  + Example: A + B
* Postfix Notation (Reverse Polish Notation - RPN): Operators are placed after operands.
  + Example: A B +
* Prefix Notation (Polish Notation): Operators are placed before operands.
  + Example: + A B

***Steps for Converting Infix to Prefix Using Stack***

1. Reverse the infix expression (left-to-right becomes right-to-left).
   * Example: (A + B) \* C → C \* (B + A)
2. Replace ( with ) and vice versa in the reversed expression.
   * Example: C \* (B + A) → C \* ) B + A (
3. Convert the modified expression to Postfix using the standard stack-based method.
4. Reverse the resulting Postfix expression to get the Prefix expression.

**Program:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#define MAX 100

char stack[MAX];

int top = -1;

void push(char c) {

stack[++top] = c;

}

char pop() {

return stack[top--];

}

int precedence(char c) {

if (c == '^') return 3;

if (c == '\*' || c == '/') return 2;

if (c == '+' || c == '-') return 1;

return -1;

}

void reverseString(char \*str) {

int i, j;

char temp;

int len = strlen(str);

for (i = 0, j = len - 1; i < j; i++, j--) {

temp = str[i];

str[i] = str[j];

str[j] = temp;

}

for (i = 0; i < len; i++) {

if (str[i] == '(')

str[i] = ')';

else if (str[i] == ')')

str[i] = '(';

}

}

void infixToPostfix(char \*infix, char \*postfix) {

int i, j = 0;

for (i = 0; infix[i] != '\0'; i++) {

if (isalnum(infix[i])) {

postfix[j++] = infix[i];

}

else if (infix[i] == '(') {

push(infix[i]);

}

else if (infix[i] == ')') {

while (top != -1 && stack[top] != '(') {

postfix[j++] = pop();

}

pop();

}

else {

while (top != -1 && precedence(stack[top]) >= precedence(infix[i])) {

postfix[j++] = pop();

}

push(infix[i]);

}

}

while (top != -1) {

postfix[j++] = pop();

}

postfix[j] = '\0';

}

void infixToPrefix(char \*infix, char \*prefix) {

reverseString(infix);

char postfix[MAX];

infixToPostfix(infix, postfix);

reverseString(postfix);

strcpy(prefix, postfix);

}

int main() {

char infix[MAX], prefix[MAX];

printf("Enter infix expression: ");

scanf("%s", infix);

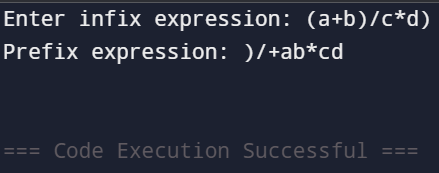
infixToPrefix(infix, prefix);

printf("Prefix expression: %s\n", prefix);

return 0;

}

**Output:**

****

# ***LAB ASSIGNMENT-6***

**Aim:** WAP in C/C++ to convert regular expressions to Finite Automata.

## **Theory:**

***Introduction to Regular Expressions (RE)***

A Regular Expression (RE) is a symbolic representation used to describe patterns in strings. It consists of literals, operators, and metacharacters. REs is used in pattern matching, lexical analysis, and automata theory.

***Types of Finite Automata (FA)***

* + Deterministic Finite Automaton (DFA): Each state has exactly one transition per input symbol.
  + Non-Deterministic Finite Automaton (NFA): A state can have multiple transitions for the same input, including epsilon (ε) transitions.

***Steps to Convert RE to FA Step 1: Convert RE to an ε-NFA***

Using Thompson’s Construction Algorithm, we break the RE into smaller units and build an ε-NFA using:

1. Concatenation (AB)
2. Union (A | B)
3. Kleene Star (A\*)
4. Single character (a, b, etc.) Step 2: Convert ε-NFA to DFA
5. Remove ε-transitions by finding ε-closures.
6. Construct the DFA by grouping equivalent states.

**Program:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_STATES 100

#define MAX\_TRANSITIONS 100

typedef struct {

int from;

char symbol[3];

int to;

} Transition;

typedef struct {

int start;

int end;

int numTransitions;

Transition transitions[MAX\_TRANSITIONS];

} NFA;

int stateCounter = 0;

NFA createBasicNFA(char symbol) {

NFA nfa;

nfa.start = stateCounter++;

nfa.end = stateCounter++;

nfa.numTransitions = 1;

strcpy(nfa.transitions[0].symbol, (char[]){symbol, '\0'});

nfa.transitions[0].from = nfa.start;

nfa.transitions[0].to = nfa.end;

return nfa;

}

NFA concatenate(NFA first, NFA second) {

NFA result = first;

for (int i = 0; i < second.numTransitions; i++) {

result.transitions[result.numTransitions++] = second.transitions[i];

}

strcpy(result.transitions[result.numTransitions].symbol, "ε");

result.transitions[result.numTransitions++] = (Transition){first.end, "ε", second.start};

result.end = second.end;

return result;

}

NFA unionNFA(NFA first, NFA second) {

NFA result;

result.start = stateCounter++;

result.end = stateCounter++;

result.numTransitions = 0;

strcpy(result.transitions[result.numTransitions].symbol, "ε");

result.transitions[result.numTransitions++] = (Transition){result.start, "ε", first.start};

strcpy(result.transitions[result.numTransitions].symbol, "ε");

result.transitions[result.numTransitions++] = (Transition){result.start, "ε", second.start};

for (int i = 0; i < first.numTransitions; i++) {

result.transitions[result.numTransitions++] = first.transitions[i];

}

for (int i = 0; i < second.numTransitions; i++) {

result.transitions[result.numTransitions++] = second.transitions[i];

}

strcpy(result.transitions[result.numTransitions].symbol, "ε");

result.transitions[result.numTransitions++] = (Transition){first.end, "ε", result.end};

strcpy(result.transitions[result.numTransitions].symbol, "ε");

result.transitions[result.numTransitions++] = (Transition){second.end, "ε", result.end};

return result;

}

NFA kleeneStar(NFA nfa) {

NFA result;

result.start = stateCounter++;

result.end = stateCounter++;

result.numTransitions = 0;

strcpy(result.transitions[result.numTransitions].symbol, "ε");

result.transitions[result.numTransitions++] = (Transition){result.start, "ε", nfa.start};

strcpy(result.transitions[result.numTransitions].symbol, "ε");

result.transitions[result.numTransitions++] = (Transition){result.start, "ε", result.end};

for (int i = 0; i < nfa.numTransitions; i++) {

result.transitions[result.numTransitions++] = nfa.transitions[i];

}

strcpy(result.transitions[result.numTransitions].symbol, "ε");

result.transitions[result.numTransitions++] = (Transition){nfa.end, "ε", nfa.start};

strcpy(result.transitions[result.numTransitions].symbol, "ε");

result.transitions[result.numTransitions++] = (Transition){nfa.end, "ε", result.end};

return result;

}

void printNFA(NFA nfa) {

printf("\nNFA Transitions:\n");

for (int i = 0; i < nfa.numTransitions; i++) {

printf("q%d --(%s)--> q%d\n", nfa.transitions[i].from, nfa.transitions[i].symbol, nfa.transitions[i].to);

}

printf("\nStart State: q%d\nEnd State: q%d\n", nfa.start, nfa.end);

}

int main() {

char regex[100];

printf("Enter postfix regular expression: ");

scanf("%s", regex);

NFA stack[MAX\_STATES];

int top = -1;

for (int i = 0; i < strlen(regex); i++) {

char c = regex[i];

if (c == 'a' || c == 'b') {

stack[++top] = createBasicNFA(c);

} else if (c == '.') {

NFA second = stack[top--];

NFA first = stack[top--];

stack[++top] = concatenate(first, second);

} else if (c == '|') {

NFA second = stack[top--];

NFA first = stack[top--];

stack[++top] = unionNFA(first, second);

} else if (c == '\*') {

NFA nfa = stack[top--];

stack[++top] = kleeneStar(nfa);

}

}

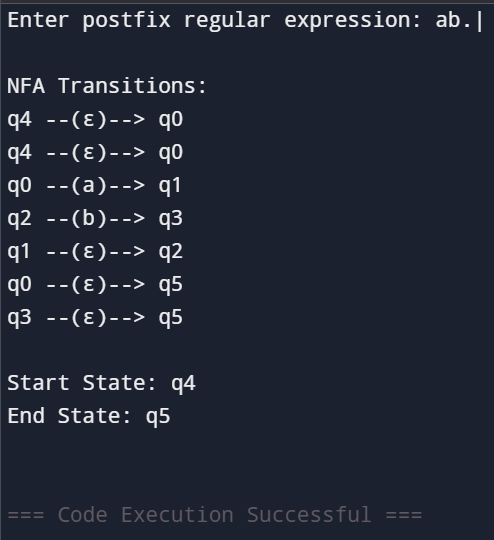
NFA result = stack[top];

printNFA(result);

return 0;

}

**Output:**

****

# ***LAB ASSIGNMENT-7***

## **Aim:** Write a program in C/C++ to convert NFA to DFA.

## **Theory:**

**What is an NFA?**

* Non-Deterministic Finite Automaton (NFA) allows multiple transitions for the same input symbol, including ε (epsilon) transitions.
* It is more flexible but harder to implement directly in hardware.

**What is a DFA?**

* + Deterministic Finite Automaton (DFA) has exactly one transition per input symbol from each state.
  + Easier to implement but may have more states than the NFA.

**Why Convert NFA to DFA?**

* + Many compilers, text processors, and automata applications require DFA because it runs in linear time O(n) while NFA runs in exponential time O(2^n) in the worst case.

**Program:**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_STATES 10

#define MAX\_SYMBOLS 2

typedef struct {

int transitions[MAX\_STATES][MAX\_SYMBOLS][MAX\_STATES];

int num\_states;

int num\_symbols;

int start\_state;

int final\_states[MAX\_STATES];

int num\_final\_states;

} NFA;

typedef struct {

int transitions[MAX\_STATES][MAX\_SYMBOLS];

int num\_states;

int num\_symbols;

int start\_state;

int final\_states[MAX\_STATES];

int num\_final\_states;

} DFA;

void find\_epsilon\_closure(NFA \*nfa, int state, int closure[], int \*count) {

closure[\*count] = state;

(\*count)++;

for (int i = 0; i < nfa->num\_states; i++) {

if (nfa->transitions[state][0][i]) {

int already\_in\_closure = 0;

for (int j = 0; j < \*count; j++) {

if (closure[j] == i) {

already\_in\_closure = 1;

break;

}

}

if (!already\_in\_closure) {

find\_epsilon\_closure(nfa, i, closure, count);

}

}

}

}

void nfa\_to\_dfa(NFA \*nfa, DFA \*dfa) {

int dfa\_states[MAX\_STATES][MAX\_STATES];

int dfa\_state\_count = 0;

for (int i = 0; i < MAX\_STATES; i++) {

for (int j = 0; j < MAX\_STATES; j++) {

dfa\_states[i][j] = -1;

}

}

int start\_closure[MAX\_STATES], start\_count = 0;

find\_epsilon\_closure(nfa, nfa->start\_state, start\_closure, &start\_count);

for (int i = 0; i < start\_count; i++) {

dfa\_states[0][i] = start\_closure[i];

}

dfa->num\_states = 1;

dfa->start\_state = 0;

dfa->num\_symbols = nfa->num\_symbols - 1;

for (int i = 0; i < dfa->num\_states; i++) {

for (int j = 1; j < nfa->num\_symbols; j++) {

int move[MAX\_STATES], move\_count = 0;

for (int k = 0; k < MAX\_STATES && dfa\_states[i][k] != -1; k++) {

int nfa\_state = dfa\_states[i][k];

for (int l = 0; l < nfa->num\_states; l++) {

if (nfa->transitions[nfa\_state][j][l]) {

move[move\_count++] = l;

}

}

}

int new\_closure[MAX\_STATES], new\_count = 0;

for (int k = 0; k < move\_count; k++) {

find\_epsilon\_closure(nfa, move[k], new\_closure, &new\_count);

}

int existing\_state = -1;

for (int k = 0; k < dfa->num\_states; k++) {

int match = 1;

for (int l = 0; l < MAX\_STATES; l++) {

if (dfa\_states[k][l] != new\_closure[l]) {

match = 0;

break;

}

}

if (match) {

existing\_state = k;

break;

}

}

if (existing\_state == -1) {

for (int k = 0; k < MAX\_STATES; k++) {

dfa\_states[dfa->num\_states][k] = new\_closure[k];

}

existing\_state = dfa->num\_states;

dfa->num\_states++;

}

dfa->transitions[i][j - 1] = existing\_state;

}

}

dfa->num\_final\_states = 0;

for (int i = 0; i < dfa->num\_states; i++) {

for (int j = 0; j < nfa->num\_final\_states; j++) {

for (int k = 0; k < MAX\_STATES && dfa\_states[i][k] != -1; k++) {

if (dfa\_states[i][k] == nfa->final\_states[j]) {

dfa->final\_states[dfa->num\_final\_states++] = i;

break;

}

}

}

}

}

void print\_dfa(DFA \*dfa) {

printf("\nDFA Transitions:\n");

for (int i = 0; i < dfa->num\_states; i++) {

for (int j = 0; j < dfa->num\_symbols; j++) {

printf("q%d --(%c)--> q%d\n", i, 'a' + j, dfa->transitions[i][j]);

}

}

printf("\nStart State: q%d\n", dfa->start\_state);

printf("Final States: ");

for (int i = 0; i < dfa->num\_final\_states; i++) {

printf("q%d ", dfa->final\_states[i]);

}

printf("\n");

}

int main() {

NFA nfa;

DFA dfa;

nfa.num\_states = 3;

nfa.num\_symbols = 3;

nfa.start\_state = 0;

nfa.final\_states[0] = 2;

nfa.num\_final\_states = 1;

for (int i = 0; i < MAX\_STATES; i++) {

for (int j = 0; j < MAX\_SYMBOLS; j++) {

for (int k = 0; k < MAX\_STATES; k++) {

nfa.transitions[i][j][k] = 0;

}

}

}

nfa.transitions[0][1][1] = 1;

nfa.transitions[0][2][0] = 1;

nfa.transitions[1][1][2] = 1;

nfa.transitions[1][2][0] = 1;

nfa.transitions[2][1][2] = 1;

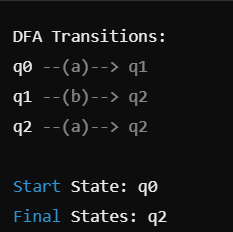
nfa\_to\_dfa(&nfa, &dfa);

print\_dfa(&dfa);

return 0;

}

**Output:**



# ***LAB ASSIGNMENT-8***

## **Aim:** Write a program in C/C++ to find first and follow of Given CFG.

## **Theory:**

***First Set***

The First set of a non-terminal in a grammar contains the set of terminals that appear as the first

symbols in some string derived from that non-terminal.

Rules for Finding First(X):

1. If X is a terminal → First(X) = {X}
2. If X → ε (epsilon) → ε is included in First(X).
3. If X → Y1 Y2 ... Yk, then:
   * If Y1 is a terminal → First(Y1) is in First(X).
   * If Y1 is a non-terminal:
     + Add First(Y1) (except ε) to First(X).
     + If First(Y1) contains ε, then check First(Y2), and so on.
     + If all Yi contain ε, then ε is added to First(X).

***Follow Set***

The Follow set of a non-terminal contains the set of terminals that can appear immediately after

that non-terminal in some derivation.

Rules for Finding Follow(X):

1. Start Symbol Rule: If X is the start symbol, add $ (end of input marker) to Follow(X).
2. If A → α B β, then add First(β) - {ε} to Follow(B).
3. If A → α B, or A → α B β where First(β) contains ε, then add Follow(A) to Follow(B).
4. Repeat until no new elements can be added to any Follow set.

**Program:**

#include <stdio.h>

#include <ctype.h>

#include <string.h>

#define MAX 10

char production[MAX][MAX];

char first[MAX][MAX], follow[MAX][MAX];

int n;

void findFirst(char, int, int);

void findFollow(char);

void addToResult(char[], char);

int isNonTerminal(char);

int main() {

int i;

char c, choice;

printf("Enter the number of productions: ");

scanf("%d", &n);

printf("Enter the grammar productions (e.g., E=E+T,T->T\*F,F->(E)|id):\n");

for (i = 0; i < n; i++) {

scanf("%s", production[i]);

}

for (i = 0; i < n; i++) {

first[i][0] = '\0';

follow[i][0] = '\0';

}

for (i = 0; i < n; i++) {

findFirst(production[i][0], i, 0);

}

for (i = 0; i < n; i++) {

findFollow(production[i][0]);

}

printf("\nFirst and Follow sets:\n");

for (i = 0; i < n; i++) {

printf("First(%c) = { %s }\n", production[i][0], first[i]);

printf("Follow(%c) = { %s }\n", production[i][0], follow[i]);

}

return 0;

}

void findFirst(char c, int index, int level) {

int i, j;

char subFirst[MAX];

if (!isNonTerminal(c)) {

addToResult(first[index], c);

return;

}

for (i = 0; i < n; i++) {

if (production[i][0] == c) {

if (!isNonTerminal(production[i][2])) {

addToResult(first[index], production[i][2]);

}

else {

findFirst(production[i][2], i, level + 1);

strcpy(subFirst, first[i]);

for (j = 0; subFirst[j] != '\0'; j++) {

addToResult(first[index], subFirst[j]);

}

}

}

}

}

void findFollow(char c) {

int i, j, k;

if (production[0][0] == c) {

addToResult(follow[0], '$');

}

for (i = 0; i < n; i++) {

for (j = 2; j < strlen(production[i]); j++) {

if (production[i][j] == c) {

if (production[i][j + 1] != '\0') {

findFirst(production[i][j + 1], i, 0);

for (k = 0; first[i][k] != '\0'; k++) {

if (first[i][k] != 'ε') {

addToResult(follow[i], first[i][k]);

}

}

}

if (production[i][j + 1] == '\0' || first[i][j + 1] == 'ε') {

findFollow(production[i][0]);

for (k = 0; follow[i][k] != '\0'; k++) {

addToResult(follow[i], follow[i][k]);

}

}

}

}

}

}

void addToResult(char result[], char val) {

int i;

for (i = 0; result[i] != '\0'; i++) {

if (result[i] == val) {

return;

}

}

int len = strlen(result);

result[len] = val;

result[len + 1] = '\0';

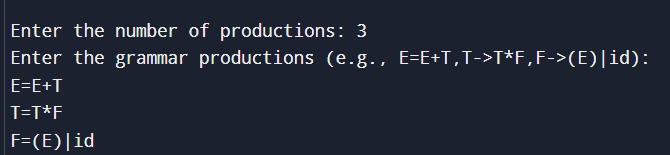
}

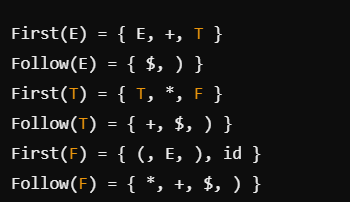
int isNonTerminal(char c) {

return (c >= 'A' && c <= 'Z');

}

**Output:**





# ***LAB ASSIGNMENT-9***

# **Aim:** Write a program in C/C++ to design shift reduce parser for: E -> E + E | E \* E | (E) | id.

**Theory:** Shift-Reduce Parsing is a bottom-up parsing technique that processes the input from left to right and reduces it to the start symbol using reductions based on grammar rules. The process consists of two main actions:

1. Shift → Push the next input symbol onto the stack.
2. Reduce → Replace a sequence of symbols on the stack with a non-terminal based on grammar rules.

Parsing Process

* + A stack is used to store symbols.
  + The parser shifts symbols onto the stack until a pattern matches a grammar rule.
  + When a match is found, a reduction is performed by replacing the matched symbols with the corresponding non-terminal (E).
  + The process continues until the entire input is reduced to the start symbol E or an error is detected.

## **Program:**

#include <stdio.h>

#include <string.h>

#include <ctype.h>

#define MAX 100

char stack[MAX];

int top = -1;

void push(char c) {

stack[++top] = c;

}

void pop(int count) {

top -= count;

}

int isOperator(char c) {

return (c == '+' || c == '\*');

}

void shiftReduce(char \*input) {

int i = 0;

printf("\nStack\t\tInput\t\tAction\n");

printf("\t\t\t\t\n");

while (i < strlen(input)) {

push(input[i]);

input[i] = ' ';

printf("%s\t\t%s\t\tShift\n", stack, input + i + 1);

while (top >= 0) {

if (top >= 2 && stack[top] == 'E' && isOperator(stack[top - 1]) && stack[top - 2] == 'E') {

pop(2);

push('E');

printf("%s\t\t%s\t\tReduce: E → E op E\n", stack, input + i + 1);

} else if (top >= 2 && stack[top] == ')' && stack[top - 1] == 'E' && stack[top - 2] == '(') {

pop(2);

push('E');

printf("%s\t\t%s\t\tReduce: E → (E)\n", stack, input + i + 1);

} else if (top >= 0 && isalnum(stack[top])) {

pop(0);

push('E');

printf("%s\t\t%s\t\tReduce: E → id\n", stack, input + i + 1);

} else {

break;

}

}

i++;

}

if (top == 0 && stack[top] == 'E') {

printf("\nInput accepted. Successfully parsed!\n");

} else {

printf("\nInput rejected. Parsing failed!\n");

}

}

int main() {

char input[MAX];

printf("Enter the input expression (e.g., id+id\*id): ");

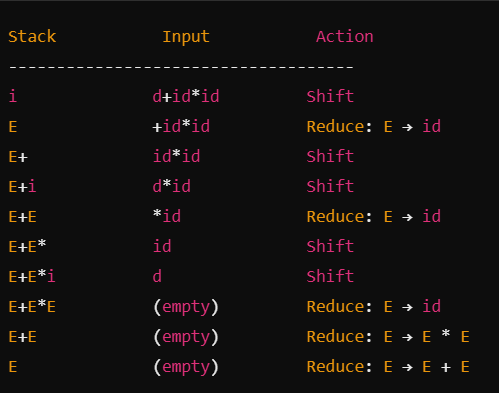
scanf("%s", input);

shiftReduce(input);

return 0;

}

**Output:**



# ***LAB ASSIGNMENT-10***

# **Aim:** Write a program in C/C++ to design shift reduce parser for: E -> E + E | E \* E | (E) | id.

**Theory:** A regular grammar is a type of context-free grammar where the production rules conform to one of the following forms:

1. Right-linear grammar:
   * A→aBA \rightarrow aBA→aB
   * A→aA \rightarrow aA→a
   * A→ϵA \rightarrow \epsilonA→ϵ (not allowed in our case)
2. Left-linear grammar (not considered in this implementation):
   * A→BaA \rightarrow BaA→Ba
   * A→aA \rightarrow aA→a The given grammar does not contain:

* Left recursion (e.g., A→AaA \rightarrow A aA→Aa)
* Null production rules (e.g., A→ϵA \rightarrow \epsilonA→ϵ)

## **Program:**

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#define MAX\_STATES 10

#define MAX\_TRANSITIONS 10

typedef struct {

char nonTerminal;

char terminal;

char nextState;

}

Transition;

int main() {

int numTransitions, i;

char startState;

char finalStates[MAX\_STATES];

int numFinalStates;

char inputString[100];

Transition transitions[MAX\_TRANSITIONS];

printf("Enter the number of transitions: ");

scanf("%d", &numTransitions);

printf("Enter the transitions (Format: A a B for A -> aB or A a - for A -> a):\n");

for (i = 0; i < numTransitions; i++) {

char nonTerm, term, nextState;

scanf(" %c %c %c", &nonTerm, &term, &nextState);

transitions[i].nonTerminal = nonTerm;

transitions[i].terminal = term;

transitions[i].nextState = (nextState == '-') ? '\0' : nextState;

}

printf("Enter the start state: ");

scanf(" %c", &startState);

printf("Enter the number of final states: ");

scanf("%d", &numFinalStates);

printf("Enter the final states: ");

for (i = 0; i < numFinalStates; i++) {

scanf(" %c", &finalStates[i]);

}

printf("Enter the string to check: ");

scanf("%s", inputString);

char currentState = startState;

int accepted = 1;

for (i = 0; i < strlen(inputString); i++) {

char symbol = inputString[i];

int foundTransition = 0;

for (int j = 0; j < numTransitions; j++) {

if (transitions[j].nonTerminal == currentState && transitions[j].terminal == symbol) {

currentState = transitions[j].nextState;

foundTransition = 1;

break;

}

}

if (!foundTransition) {

accepted = 0;

break;

}

}

accepted = accepted && (strchr(finalStates, currentState) != NULL);

if (accepted)

printf("String is ACCEPTED by the given regular grammar.\n");

else

printf("String is REJECTED by the given regular grammar.\n");

return 0;

}

## **Output:**