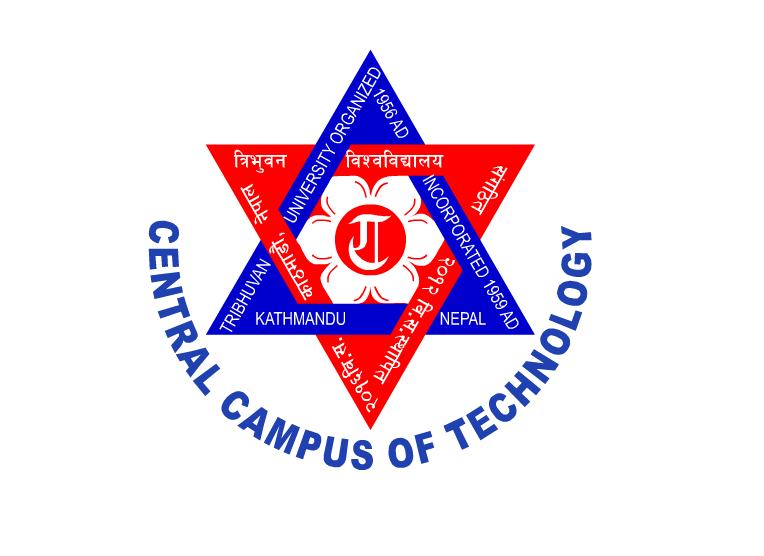
**Central Campus of Technology**



**SUBJECT: THEORY OF COMPUTATION**

**4th Semester**

**Bachelor of Science in Computer Science and Information Technology**

**(Bsc. CSIT)**

**Submitted By Submitted To**

**Department of Information Technology**

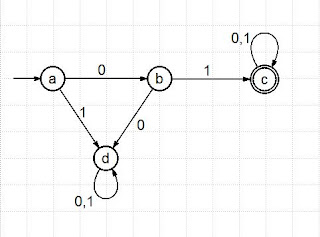
**Roll No: 01 (Prasana Regmi)**

**Table of Content**

|  |  |  |
| --- | --- | --- |
| **S.N** | **Name of Practical** | **Remarks** |
| 1 | C program to implement DFA’s over alphabet ∑ = {0, 1}. The DFA that accepts all the strings that start with 01. |  |
| 2 | C program to implement DFA’s over alphabet ∑ = {0, 1}. The DFA that accepts all the strings that end with 01. |  |
| 3 | C program for a DFA that accepts stings with substring 001 over {0,1}. |  |
| 4 | C program to validate C identifiers and keywords. |  |
| 5 | C program to implement NFA’s over alphabet ∑ = {0, 1}. The NFA that accepts all the strings that starts with 01. |  |
| 6 | C program to implement NFA’s over alphabet ∑ = {0, 1}. The NFA that accepts all the strings that ends with 01. |  |
| 7 | C program for a NFA that accepts stings with substring 001 over {0,1}. |  |
| 8 | C program for a NFA that accepts stings with substring 001 over {0,1}. |  |
| 9 | C program to implement PDA that accepts equal number of 0’s and 1’s with empty stack. |  |
| 10 | C program to implement a Turing Machine (TM) for L = {set of all strings over {0, 1} such that the string has number of 0’s followed by same number of 1’s. |  |

**Practical No: - 1**

**Title:** C program to implement DFA’s over alphabet ∑ = {0, 1}. The DFA that accepts all the strings that start with 01.



**Theory:**

DFA refers to deterministic finite automata. Deterministic refers to the uniqueness of the computation. The finite automata are called deterministic finite automata if the machine is read an input string one symbol at a time. In DFA, there is only one path for specific input from the current state to the next state. DFA does not accept the null move, i.e., the DFA cannot change state without any input character. DFA can contain multiple final states. It is used in Lexical Analysis in Compiler.

The formal definition of DFA:

A DFA can be represented by a 5-tuple (Q, ∑, δ, q0, F) where −

* Q is a finite set of states.
* ∑ is a finite set of symbols called the alphabet.
* δ is the transition function where δ: Q × ∑ → Q
* q0 is the initial state from where any input is processed (q0 ∈ Q).
* F is a set of final state/states of Q (F ⊆ Q).

**Code:**

#include <stdio.h>

#include <string.h>

typedef enum {

q0,

q1,

q2,

DEAD

} State;

State transition(State current\_state, char input) {

switch (current\_state) {

case q0:

if (input == '0') return q1;

else return DEAD;

case q1:

if (input == '1') return q2;

else return DEAD;

case q2:

return q2;

default:

return DEAD;

}

}

int simulateDFA(const char\* input) {

State current\_state = q0;

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

current\_state = transition(current\_state, input[i]);

if (current\_state == DEAD) {

return 0;

}

}

return (current\_state == q2);

}

int main() {

char input[100];

printf("Enter a binary string (only 0s and 1s): ");

scanf("%s", input);

if (simulateDFA(input)) {

printf("The string is accepted by the DFA.\n");

} else {

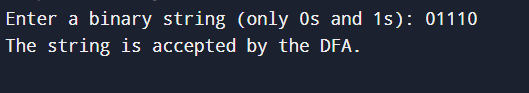
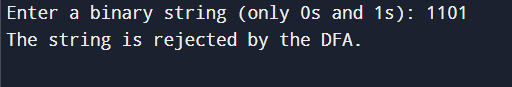
printf("The string is rejected by the DFA.\n");

}

return 0;

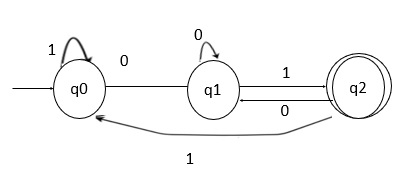
}

**Output:**

**Practical No: - 2**

**Title:** C program to implement DFA’s over alphabet ∑ = {0, 1}. The DFA that accepts all the strings that end with 01.



**Theory:**

DFA refers to deterministic finite automata. Deterministic refers to the uniqueness of the computation. The finite automata are called deterministic finite automata if the machine is read an input string one symbol at a time. In DFA, there is only one path for specific input from the current state to the next state. DFA does not accept the null move, i.e., the DFA cannot change state without any input character. DFA can contain multiple final states. It is used in Lexical Analysis in Compiler.

The formal definition of DFA:

A DFA can be represented by a 5-tuple (Q, ∑, δ, q0, F) where −

* Q is a finite set of states.
* ∑ is a finite set of symbols called the alphabet.
* δ is the transition function where δ: Q × ∑ → Q
* q0 is the initial state from where any input is processed (q0 ∈ Q).
* F is a set of final state/states of Q (F ⊆ Q).

**Code:**

#include <stdio.h>

#include <string.h>

typedef enum {

q0,

q1,

q2,

DEAD

} State;

State transition(State current\_state, char input) {

switch (current\_state) {

case q0:

if (input == '0') return q1;

else return q0;

case q1:

if (input == '1') return q2;

else return q1;

case q2:

if (input == '0') return q1;

else return q0;

default:

return DEAD;

}

}

int simulateDFA(const char\* input) {

State current\_state = q0;

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

current\_state = transition(current\_state, input[i]);

}

return (current\_state == q2);

}

int main() {

char input[100];

printf("Enter a binary string (only 0s and 1s): ");

scanf("%s", input);

if (simulateDFA(input)) {

printf("The string is accepted by the DFA.\n");

} else {

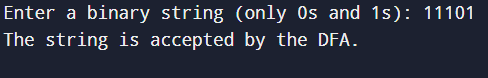
printf("The string is rejected by the DFA.\n");

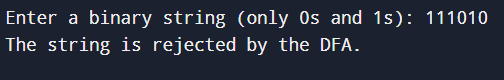
}

return 0;

}

**Output:**

****

****

**Practical No: - 3**

**Title:** C program for a DFA that accepts stings with substring 001 over {0,1}.

**Theory:**

DFA refers to deterministic finite automata. Deterministic refers to the uniqueness of the computation. The finite automata are called deterministic finite automata if the machine is read an input string one symbol at a time. In DFA, there is only one path for specific input from the current state to the next state. DFA does not accept the null move, i.e., the DFA cannot change state without any input character. DFA can contain multiple final states. It is used in Lexical Analysis in Compiler.

The formal definition of DFA:

A DFA can be represented by a 5-tuple (Q, ∑, δ, q0, F) where −

* Q is a finite set of states.
* ∑ is a finite set of symbols called the alphabet.
* δ is the transition function where δ: Q × ∑ → Q
* q0 is the initial state from where any input is processed (q0 ∈ Q).
* F is a set of final state/states of Q (F ⊆ Q).

**Code:**

#include <stdio.h>

#include <string.h>

typedef enum {

q0,

q1,

q2,

q3

} State;

State transition(State current\_state, char input) {

switch (current\_state) {

case q0:

if (input == '0') return q1;

else return q0;

case q1:

if (input == '0') return q2;

else return q0;

case q2:

if (input == '1') return q3;

else return q1;

case q3:

return q3;

default:

return q0;

}

}

int simulateDFA(const char\* input) {

State current\_state = q0;

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

current\_state = transition(current\_state, input[i]);

}

return (current\_state == q3);

}

int main() {

char input[100];

printf("Enter a binary string (only 0s and 1s): ");

scanf("%s", input);

if (simulateDFA(input)) {

printf("The string contains '001', so it is accepted.\n");

} else {

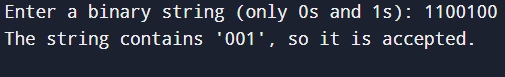
printf("The string does not contain '001', so it is not accepted.\n");

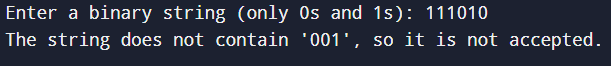
}

return 0;

}

**Output:**

****

****

**Practical No: - 4**

**Title:** C program to validate C identifiers and keywords.

**Theory:**

**C identifiers:** These are the names of variables, functions, arrays, structures and pointers etc. The first character of C identifiers must be letter or underscore and remaining characters might be letters, digits or underscore.

**Keywords:** These are the reserved words having predefined meaning in the language. There are 32 keywords in C. They cannot be used as identifiers.

**Code:**

#include<stdio.h>

#include<string.h>

char keyword[32][10] = {"auto", "double", "int", "struct", "break", "else",

"long", "switch", "case", "enum", "register", "typedef",

"char", "extern", "return", "union", "const", "float",

"short", "unsigned", "continue", "for", "signed", "void",

"default", "goto", "sizeof", "volatile", "do", "if",

"static", "while"};

enum states {q0, qf, qd};

enum states delta(enum states, char);

int iskeyword(char []);

int main(){

enum states curr\_state = q0;

char string[20], ch;

int i = 0;

printf("\n Enter a string \t");

gets(string);

ch = string[i];

if(iskeyword(string))

printf("\n The string %s is keyword.", string);

else {

while(ch != '\0') {

curr\_state = delta(curr\_state, ch);

ch = string[++i];

}

if(curr\_state == qf)

printf("\n The string %s is valid identifier.", string);

else

printf("\n The string %s is neither keyword nor valid identifier.", string);

}

return 0;

}

enum states delta(enum states s, char ch) {

enum states curr\_state;

switch(s) {

case q0:

if(ch >= 'A' && ch <= 'Z' || ch >= 'a' && ch <= 'z' || ch == '\_')

curr\_state = qf;

else

curr\_state = qd;

break;

case qf:

if(ch >= 'A' && ch <= 'Z' || ch >= 'a' && ch <= 'z' || ch == '\_' || ch >= '0' && ch <= '9')

curr\_state = qf;

else

curr\_state = qd;

break;

case qd:

curr\_state = qd;

break;

}

return curr\_state;

}

int iskeyword(char str[]) {

for(int i = 0; i < 32; i++) {

if(strcmp(str, keyword[i]) == 0)

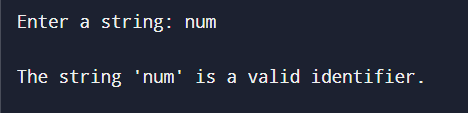
return 1;

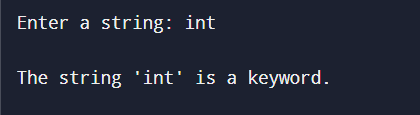
}

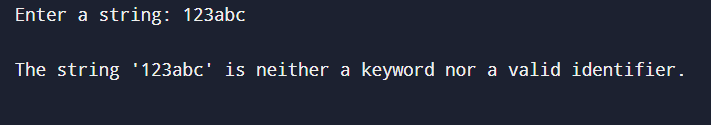
return 0;

}

**Output:**

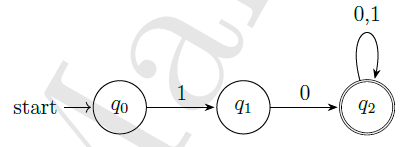






**Practical No: - 5**

**Title:** C program to implement NFA’s over alphabet ∑ = {0, 1}. The NFA that accepts all the strings that starts with 01.



**Theory:**

NFA refers to Nondeterministic Finite Automaton. A Finite Automata (FA) is said to be non-deterministic if there is more than one possible transition from one state on the same input symbol.

The formal definition of NFA:

A NFA can be represented by a 5-tuple (Q, ∑, δ, q0, F) where −

* Q is a finite set of states.
* ∑ is a finite set of symbols called the alphabet.
* δ is the transition function where δ: Q × ∑ ∪ {ε}→2|Q|
* q0 is the initial state from where any input is processed (q0 ∈ Q).
* F is a set of final state/states of Q (F ⊆ Q).

**Code:**

#include <stdio.h>

#include <string.h>

#define STATES 3

#define SYMBOLS 2

int simulate\_nfa(char \*input) {

int current\_states[STATES] = {1, 0, 0};

int next\_states[STATES] = {0, 0, 0};

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

char symbol = input[i];

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

next\_states[j] = 0;

}

for (int state = 0; state < STATES; state++) {

if (current\_states[state]) {

switch (state) {

case 0:

if (symbol == '0') {

next\_states[1] = 1;

}

break;

case 1:

if (symbol == '1') {

next\_states[2] = 1;

}

break;

case 2:

next\_states[2] = 1;

break;

}

}

}

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

current\_states[j] = next\_states[j];

}

}

return current\_states[2];

}

int main() {

char input[100];

printf("Enter a string over alphabet {0, 1}: ");

scanf("%s", input);

if (simulate\_nfa(input)) {

printf("String is accepted (starts with 01).\n");

} else {

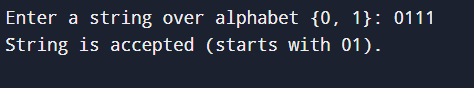
printf("String is rejected (does not start with 01).\n");

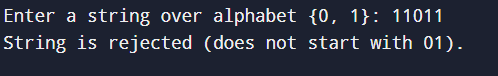
}

return 0;

}

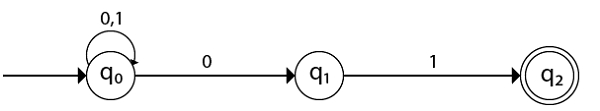
**Output:**

****

****

**Practical No: - 6**

**Title:** C program to implement NFA’s over alphabet ∑ = {0, 1}. The NFA that accepts all the strings that ends with 01.



**Theory:**

NFA refers to Nondeterministic Finite Automaton. A Finite Automata (FA) is said to be non-deterministic if there is more than one possible transition from one state on the same input symbol.

The formal definition of NFA:

A NFA can be represented by a 5-tuple (Q, ∑, δ, q0, F) where −

* Q is a finite set of states.
* ∑ is a finite set of symbols called the alphabet.
* δ is the transition function where δ: Q × ∑ ∪ {ε}→2|Q|
* q0 is the initial state from where any input is processed (q0 ∈ Q).
* F is a set of final state/states of Q (F ⊆ Q).

**Code:**

#include <stdio.h>

#include <string.h>

#define STATES 3

#define SYMBOLS 2

int simulate\_nfa(char \*input) {

int current\_states[STATES] = {1, 0, 0};

int next\_states[STATES] = {0, 0, 0};

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

char symbol = input[i];

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

next\_states[j] = 0;

}

for (int state = 0; state < STATES; state++) {

if (current\_states[state]) {

switch (state) {

case 0:

if (symbol == '0') next\_states[0] = 1;

if (symbol == '0') next\_states[1] = 1;

break;

case 1:

if (symbol == '1') next\_states[2] = 1;

break;

case 2:

if (symbol == '0') next\_states[1] = 1;

if (symbol == '1') next\_states[2] = 1;

break;

}

}

}

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

current\_states[j] = next\_states[j];

}

}

return current\_states[2];

}

int main() {

char input[100];

printf("Enter a string over alphabet {0, 1}: ");

scanf("%s", input);

if (simulate\_nfa(input)) {

printf("String is accepted (ends with 01).\n");

} else {

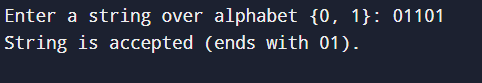
printf("String is rejected (does not end with 01).\n");

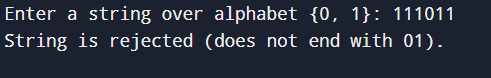
}

return 0;

}

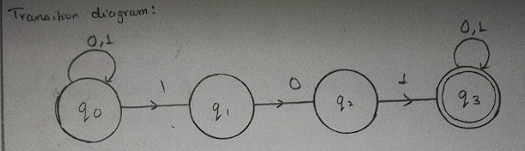
**Output:**



****

**Practical No: - 7**

**Title:** C program for a NFA that accepts stings with substring 001 over {0,1}.



**Theory:**

NFA refers to Nondeterministic Finite Automaton. A Finite Automata (FA) is said to be non-deterministic if there is more than one possible transition from one state on the same input symbol.

The formal definition of NFA:

A NFA can be represented by a 5-tuple (Q, ∑, δ, q0, F) where −

* Q is a finite set of states.
* ∑ is a finite set of symbols called the alphabet.
* δ is the transition function where δ: Q × ∑ ∪ {ε}→2|Q|
* q0 is the initial state from where any input is processed (q0 ∈ Q).
* F is a set of final state/states of Q (F ⊆ Q).

**Code:**

#include <stdio.h>

#include <string.h>

#define STATES 4

int simulate\_nfa(char \*input) {

int current\_states[STATES] = {1, 0, 0, 0};

int next\_states[STATES] = {0, 0, 0, 0};

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

char symbol = input[i];

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

next\_states[j] = 0;

}

for (int state = 0; state < STATES; state++) {

if (current\_states[state]) {

switch (state) {

case 0: // State 0: waiting for the first '0'

if (symbol == '0') {

next\_states[0] = 1; // Stay in state 0 if '0'

next\_states[1] = 1; // Move to state 1 for first '0'

}

break;

case 1: // State 1: waiting for the second '0'

if (symbol == '0') {

next\_states[2] = 1; // Move to state 2 for second '0'

}

break;

case 2: // State 2: waiting for '1' after '00'

if (symbol == '1') {

next\_states[3] = 1; // Move to accepting state 3 if '1'

}

break;

case 3: // State 3: accepting state

next\_states[3] = 1; // Stay in accepting state

break;

}

}

}

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

current\_states[j] = next\_states[j];

}

}

return current\_states[3]; // Check if in accepting state (state 3)

}

int main() {

char input[100];

printf("Enter a string over alphabet {0, 1}: ");

scanf("%s", input);

if (simulate\_nfa(input)) {

printf("String is accepted (contains substring 001).\n");

} else {

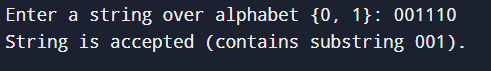
printf("String is rejected (does not contain substring 001).\n");

}

return 0;

}

**Output:**

****

**Practical No: - 8**

**Title:** C program to implement PDA that accepts all strings over alphabet {0, 1} that have equal number of 0’s and 1’s.

**Theory:**

The context free languages have a type of automation that defines them. This automation is called “pushdown automation” which can be thought as a ε-NFA with the addition of stack. The presence of a stack means that, the pushdown automata can remember infinity amount of information. However, the pushdown automation can only access the information on its stack in a Last-in-first-out way.

A PDA is defined by seven tuples (Q, ∑, Γ, δ, q0, Z0, F) where-

* **Q** is the finite number of states
* **∑** is input alphabet
* Γ is stack symbols
* **δ** is the transition function: Q × (∑ ∪ {ε}) × S × Q × S\*
* **q0** is the initial state (q0 ∈ Q)
* Z0 is the initial stack top symbol (Z0 ∈ S)
* **F** is a set of accepting states (F ∈ Q)

A PDA for L = {set of all strings over [0,1] such that equal number of 0s and 1s, acceptance by final state.

For this, let us construct a PDA as;

P = {Q, Σ, Г, δ, q0, z0, F} be the PDA recognizing the given language. where, let us suppose

Q = {q0, q1, q2}

Σ = {0, 1}

Г = {0, 1, z0}

z0 = z0

q0 = q0

F = {q2}

Now δ is defined as;

1. δ(q0, Є, Є) = (q1, z0) //initialize the stack to indicate the bottom of stack.

2. δ(q1, 0, z0) = (q1, 0z0)

3. δ(q1, 1, z0) = (q1, 1z0)

4. δ(q1, 0, 0) = (q1, 00)

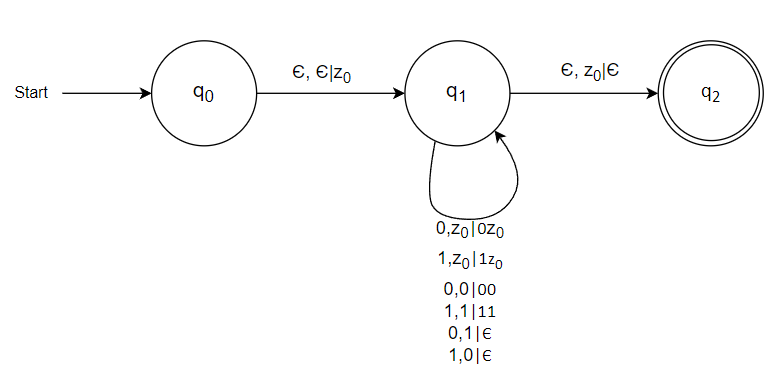
5. δ(q1, 1, 1) = (q1, 11)

6. δ(q1, 0, 1) = (q1, Є)

7. δ(q1, 1, 0) = (q1, Є)

8. δ(q1, Є, z0) = (q2, Є) // indicate the acceptance of string.

So, the graphical notation for the PDA constructed in example 1 can be constructed as;



**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_STACK 100

typedef enum { q0, qf, qError } State;

typedef struct {

char items[MAX\_STACK];

int top;

} Stack;

void initStack(Stack\* s) {

s->top = -1;

}

int isFull(Stack\* s) {

return s->top == MAX\_STACK - 1;

}

int isEmpty(Stack\* s) {

return s->top == -1;

}

void push(Stack\* s, char item) {

if (!isFull(s)) {

s->items[++s->top] = item;

}

}

char pop(Stack\* s) {

if (!isEmpty(s)) {

return s->items[s->top--];

}

return '\0';

}

State delta(State currentState, char inputSymbol, Stack\* stack) {

switch (currentState) {

case q0:

if (inputSymbol == '0') {

push(stack, inputSymbol);

} else if (inputSymbol == '1') {

if (!isEmpty(stack) && stack->items[stack->top] == '0') {

pop(stack);

} else {

return qError;

}

} else {

return qError;

}

break;

case qf:

return qError;

case qError:

return qError;

}

return currentState;

}

int isAccepted(const char\* input) {

Stack stack;

initStack(&stack);

State currentState = q0;

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

currentState = delta(currentState, input[i], &stack);

if (currentState == qError) {

return 0;

}

}

return (currentState == q0 && isEmpty(&stack));

}

int main() {

char input[100];

printf("Enter a string over {0,1}: ");

scanf("%s", input);

if (isAccepted(input)) {

printf("Accepted.\n");

} else {

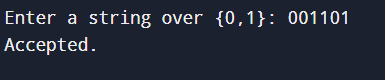
printf("Rejected.\n");

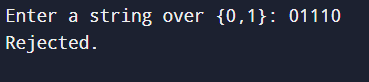
}

return 0;

}

**Output:**

****

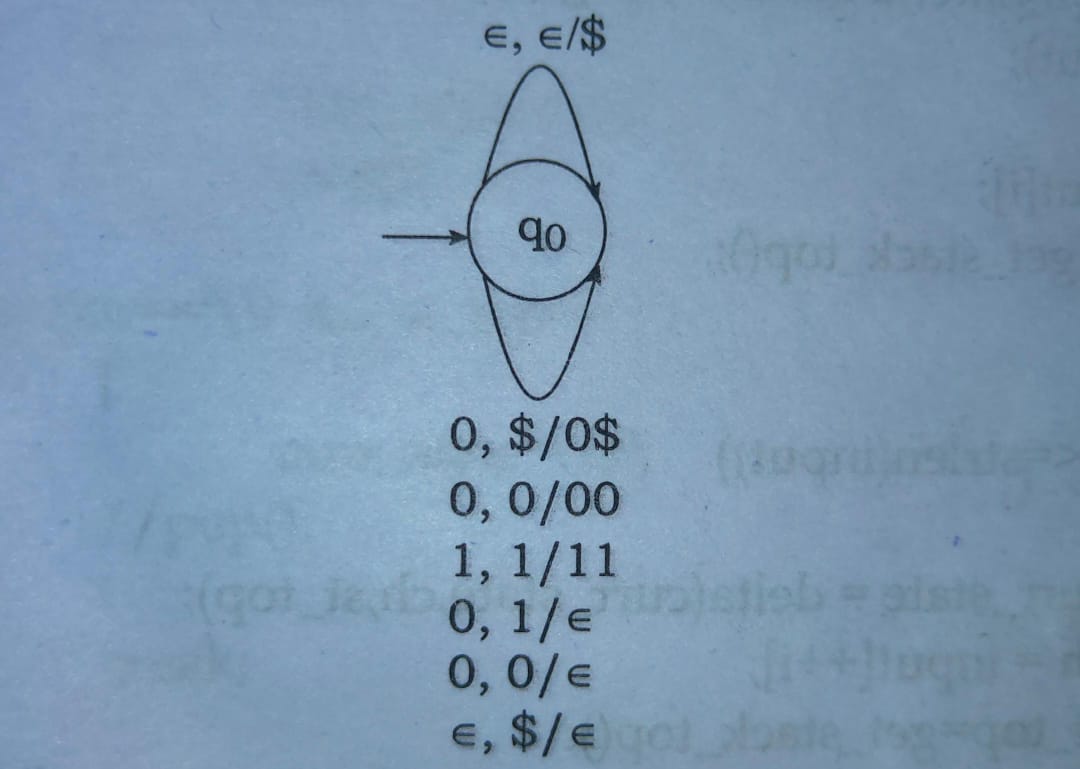
****

**Practical No: - 9**

**Title:** C program to implement PDA that accepts equal number of 0’s and 1’s with empty stack.

**Theory:**

A Pushdown Automaton (PDA) is a type of computational model that extends finite automata by adding a stack. This stack allows the PDA to keep track of an unbounded amount of information, enabling it to recognize context-free languages.



**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_STACK 100

typedef struct {

char items[MAX\_STACK];

int top;

} Stack;

void initStack(Stack\* s) {

s->top = -1;

}

int isFull(Stack\* s) {

return s->top == MAX\_STACK - 1;

}

int isEmpty(Stack\* s) {

return s->top == -1;

}

void push(Stack\* s, char item) {

if (!isFull(s)) {

s->items[++s->top] = item;

}

}

char pop(Stack\* s) {

if (!isEmpty(s)) {

return s->items[s->top--];

}

return '\0';

}

int isAccepted(const char\* input) {

Stack stack;

initStack(&stack);

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

if (input[i] == '0') {

push(&stack, '0');

} else if (input[i] == '1') {

if (!isEmpty(&stack)) {

pop(&stack);

} else {

push(&stack, '1'); // Allow `1` to remain if no `0` to match

}

} else {

return 0; // Invalid character

}

}

return isEmpty(&stack); // Accept if stack is empty

}

int main() {

char input[100];

printf("Enter a string over {0,1}: ");

scanf("%s", input);

if (isAccepted(input)) {

printf("Accepted.\n");

} else {

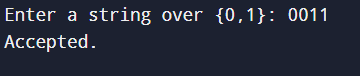
printf("Rejected.\n");

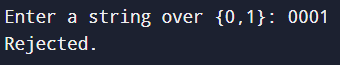
}

return 0;

}

**Output:**

****

****

**Practical No: - 10**

**Title:** C program to implement a Turing Machine (TM) for L = {set of all strings over {0, 1} such that the string has number of 0’s followed by same number of 1’s.

**Theory:**

Turing machine is an abstract machine developed by an English Mathematician Alan Turing in 1936. The model of computation provides a theoretical foundation for modern computers. A

Turing machine will have;

* A finite set of alphabets
* A finite set of states
* A linear tape which is potentially infinite to both end.

A Turing Machine T M is defined by the seven-tuples, M = (Q, Σ, Г, δ, q0, B, F) where,

* Q = the finite set of states of the finite control
* Σ = the finite set of input symbols
* Г = the complete set of tape symbols Σ is always subset of Г.
* q0 = the start state; q0 ε Q
* B = the blank symbol; B ε Г but B does not belong to Σ.
* F = the set of final or accepting states; F is subset of Q
* δ = the transition function defined by

Q × Г →Q × Г × (R, L, S); where R, L, S is the direction of movement of head left, or right or stationary. i.e. δ(q, x) = δ(p, Y, D); which means T M in state q and current tape symbol x, moves to next state P, replacing tape symbol x with Y and move the head either direction or remains at same cell of input tape.

A TM accepting the language [0n1n / n ≥ 1] over alphabet, ∑ = [0, 1].

Given finite sequence of 0’s and 1’s on tape and followed by blanks. The TM starts at state q0 and changes 0 to an X and moves to the right changing its state of q1.

At state q1, TM expects 1 and changes a 1 to Y and moves to the left changing the state to q2. If any number of 0s and Ys are seen, it remains on the state q1 and leaving these symbols unchanged and moving the head position to the right.

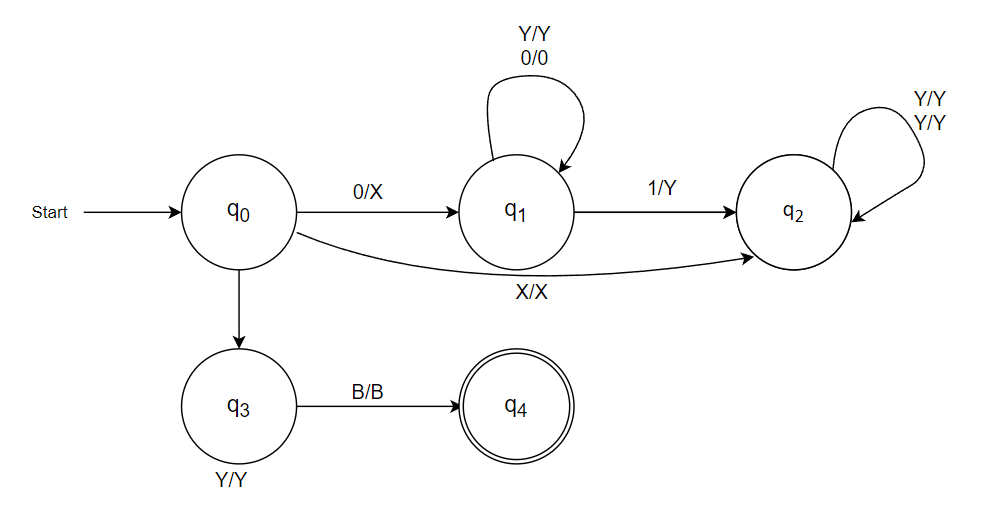
At state q2, if 0s or Y’s are seen, it leaves them as it is and moves to the left staying at the same state q2. If it sees X at state q2, the tape symbol is left unchanged moves to right switching its sate to q0.

At state q0, if it sees Y then the symbol is left unchanged and head is moved right changing to the state q3.

At sate q3, if Y is seen, it is left unchanged and head is moved to the right. If BLANK (here, ‘\0’) is seen at state q3, the string is accepted by switching the state to q4.

At any state, if the machine seen other than the defined symbols, it rejects the string.

The state transition diagram is shown in the figure below q3.



**Code:**

#include <stdio.h>

#include <string.h>

#define MAX\_TAPE 100

typedef struct {

char tape[MAX\_TAPE];

int head;

} TuringMachine;

void initialize\_tape(TuringMachine \*tm, const char \*input) {

strcpy(tm->tape, input);

tm->head = 0;

}

int process\_tape(TuringMachine \*tm) {

int count0 = 0, count1 = 0;

for (int i = 0; i < strlen(tm->tape); i++) {

if (tm->tape[i] == '0') {

count0++;

} else if (tm->tape[i] == '1') {

count1++;

}

}

return count0 == count1;

}

int main() {

TuringMachine tm;

char input[MAX\_TAPE];

printf("Enter a binary string: ");

scanf("%s", input);

initialize\_tape(&tm, input);

if (process\_tape(&tm)) {

printf("Accepted\n");

} else {

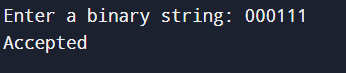
printf("Rejected\n");

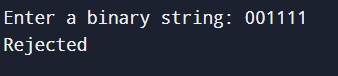
}

return 0;

}

**Output:**

****

****