FORMAN CHRISTIAN COLLEGE
(A CHARTERED UNIVERSITY)

COMP 451 B

SP24

Lab 7

Hafsah Shahbaz – 251684784

Daim Bin Khalid - 251686775

INTRODUCTION

This lab report delves into the structure of a Deterministic Finite Automaton (DFA) implemented in a C program. The DFA is designed to validate strings based on a specific pattern defined by states and transitions. The program serves as a fundamental tool as demonstrating being part of a compiler that makes sure only valid strings are accepted and invalid strings are prompted as errors.

Objectives:

- Implement a DFA to validate strings according to a predetermined pattern.
- Validate input strings provided as command-line arguments.
- Demonstrate state transitions and actions taken during the validation process.

The program begins by accepting a command-line argument specifying the string to be validated. Upon execution, it initializes the DFA with an initial state and iterates through the characters of the input string. At each step, it follows predefined transitions based on the current state and the input character. If the input string satisfies the pattern defined by the DFA, the program terminates with a success message. Otherwise, it terminates with a rejection message.

Standard C library functions serve as the backbone of this program, facilitating various tasks such as input validation, and character manipulation. These functions streamline development, ensuring efficient and reliable program execution.

1. **`<stdio.h>`** The standard input-output library in C provides functions for file handling and console input and output operations. In our program, it's extensively used for file handling and console output.

    - `printf()`: Used to print formatted output to the standard output stream.

2. **`<string.h>:`** This library is used to make string manipulation easier.

    - `strlen()`: Returns the int size of the string passed to it.

ALGORITHM and LOGIC

The program implements a Deterministic Finite Automaton (DFA) to validate strings based on a predefined pattern. Below is a detailed explanation of the algorithm and its logic:

1. **Initialization:**

   - The program begins by checking if the correct number of command-line arguments is provided. It expects only one argument, which is the string to be validated.
   - If the number of arguments is incorrect, the program displays a usage message and exits with an error code.

2. **Input Validation:**

   - After receiving the input string from the command line, the program checks if the string ends with the '$' symbol, which serves as an end-of-string marker.
   - If the string does not end with '$', indicating an incomplete or invalid input, the program displays an error message and exits.

3. **State Transition:**

   - The DFA implementation defines a set of states ('q0', 'q1', 'q2', 'q3', 'q4') and transitions between them based on the input symbols ('a', 'b') and the end-of-string marker ('$').
   - The DFA transitions between states according to the rules defined by the DFA's state transition table. Each state represents a specific stage in the validation process, and each transition represents the acceptance or rejection of the input string based on the current state and the next input symbol.
   - The program utilizes a series of 'goto' statements to simulate the DFA's state transitions. While 'goto' statements are generally discouraged in modern programming due to readability concerns, they provide a straightforward way to represent state transitions in this context.

4. **Validation Result:**

   - Upon reaching the end of the input string ('$' symbol), the program evaluates the final state to determine the acceptance or rejection of the input string.
   - If the final state corresponds to an accepting state, the input string is accepted, and the program terminates with a success message.
   - If the final state corresponds to a non-accepting state, the input string is rejected, and the program terminates with a rejection message.

Further breakdown of code:

1 . Arguments & Initialization:

Functionality:

Entry point of the program. It processes command-line arguments, checks for correct number of arguments, makes sure the string ends with the delimeter ‘$’, sets up state tracker and its flags.

Arguments:

- argc: An integer representing the number of command-line arguments passed to the program.
- argv: An array of strings containing the command-line arguments.

Variables:

- input_str: Stores the string input to be checked by the state machine.
- current_state: The current state of the DFA is kept track of here. Initialized to ‘q0’.
- total_char: Total size of the input string.
- i: The iterator to track characters in the string array.
- accept: Flag to indicate if the input string will be accepted or not.

Code Snippet:

```
if (argc != 2)
{
    printf("Usage: %s <string>\n", argv[0]);
    return 1;
}

char *input_str = argv[1];
if (input_str[strlen(input_str) - 1] != '$')
{
    printf("String does not end with $\nExiting...\n");
    return 1;
}

printf("Input String is: %s\n", input_str);
printf("State transitions are shown below:\n");

char *current_state = "q0";
int total_char = strlen(input_str);
int i = -1;
int accept = 0;
```

2. State Definitions & Transitions:

Functionality:

This section of the program iterates through each character of the string and based on the input of either ‘a’ or ‘b’ moves to the defined corresponding state. If any other character is encountered the program rejects the string and terminates it. If ‘$’ ending delimiter is found, then conclude the program.

States:

- q0: if ‘a’ is encountered move to ‘q1’, if ‘b’ is encounctered move to ‘q3’.
- q2: if ‘a’ is encountered move to ‘q2’, if ‘b’ is encounctered move to ‘q3’.
- q3: if ‘a’ is encountered move to ‘q1’, if ‘b’ is encounctered move to ‘q4’.
- q4: if ‘a’ is encountered move to ‘q1’, if ‘b’ is encounctered move to ‘q4’.
- finish: Concludes if accept flag is 1 then accepts the string and if 0 then reject.

Code Snippet:

```
// start of the state machine

goto q0;


q0:

    current_state = "q0";

    i += 1;


    if (input_str[i] == 'a')

    {

        printf("Received %c on state %s ---- Moving to state q1\n", 
input_str[i], current_state);

        goto q1;

    }

    else if (input_str[i] == 'b')

    {

        printf("Received %c on state %s ---- Moving to state q3\n", 
input_str[i], current_state);

        goto q3;

    }

    else if (input_str[i] == '$')
    {
        accept = 0;
        goto finish;
    }
    else
    {
        printf("Received unknown character %c on state %s\n", input_str[i],
current_state);
        accept = -1;
        goto finish;
    }

.
.
.

q4:
    current_state = "q4";
    i += 1;

    if (input_str[i] == 'a')
    {
        printf("Received %c on state %s ---- Moving to state q1\n",
input_str[i], current_state);
        goto q1;
    }
    else if (input_str[i] == 'b')
    {
        printf("Received %c on state %s ---- Moving to state q4\n",
input_str[i], current_state);
        goto q4;
    }
    else if (input_str[i] == '$')
    {
        accept = 1;
        goto finish;
    }
    else
    {
        printf("Received unknown character %c on state %s\n", input_str[i],
current_state);
        accept = -1;
        goto finish;
    }

finish:
    if (accept == -1)
    {
        printf("Terminating the process...\n");
        return 1;
    }

    printf("End of string.\n");

    if (accept == 0)
    {
        printf("String rejected.\n");
        return 1;
    }
    else if (accept == 1)
    {
        printf("String accepted\n");
        return 0;
    }
}
```

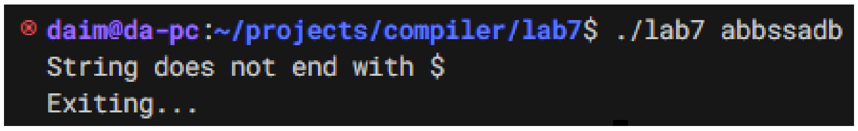
OUTPUT

1. Sample Run 1:

Input String:

- abbssadb

Result:



```
daim@da-pc:~/projects/compiler/lab7$ ./lab7 abbssadb
String does not end with $
Exiting...
```

Comments:

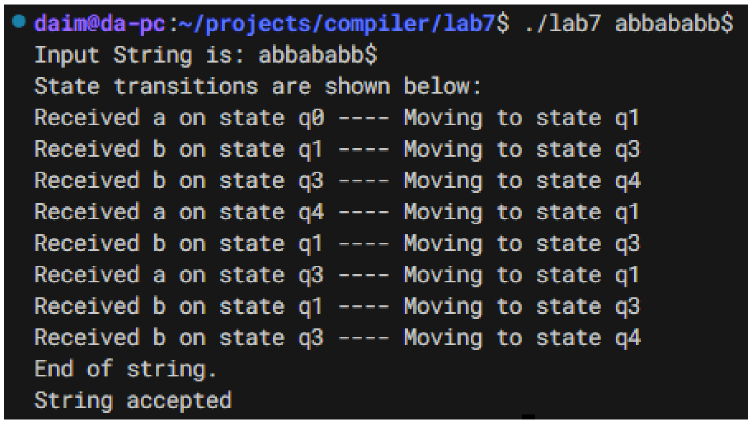
Demonstrates how ‘$’ is necessary.

2. Sample Run 2:

Input String:

- abbababb$

Result:



```
daim@da-pc:~/projects/compiler/lab7$ ./lab7 abbababb$
Input String is: abbababb$
State transitions are shown below:
Received a on state q0 ---- Moving to state q1
Received b on state q1 ---- Moving to state q3
Received b on state q3 ---- Moving to state q4
Received a on state q4 ---- Moving to state q1
Received b on state q1 ---- Moving to state q3
Received a on state q3 ---- Moving to state q1
Received b on state q1 ---- Moving to state q3
Received b on state q3 ---- Moving to state q4
End of string.
String accepted
```

Comments:

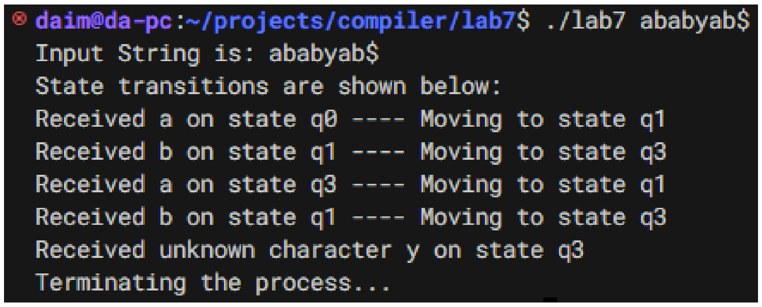
Demonstration of a valid string being checked in the DFA state machine.

3. Sample Run 2:

Input String:

- ababyab$

Result:



```
⊗ daim@da-pc:~/projects/compiler/lab7$ ./lab7 ababyab$
  Input String is: ababyab$
  State transitions are shown below:
  Received a on state q0 ---- Moving to state q1
  Received b on state q1 ---- Moving to state q3
  Received a on state q3 ---- Moving to state q1
  Received b on state q1 ---- Moving to state q3
  Received unknown character y on state q3
  Terminating the process...
```

Comments:

Demonstration of an invalid string being rejected by the DFA state machine.