CS203 AfternoonLab Lab 3 – State Machines in a Simple C Program

Assigned: Monday September 8nd, 2025
Due: Sunday September 14th, 2025, to the Moodle by 11:59.

General Problem.

For this lab you will be provided a tar-zipped file on the Lab Moodle page. In this directory, a Makefile is provided to help you with your lab. To setup the lab:

1. download the file,
2. untar-zip it (using "`tar xvfz lab3_yourid.tgz`"),
3. then modifying the directory to have your id, and
4. modify the `Makefile` to reflect the new directory name.

You are required to use the `main.c` file provide with the project to implement several different Finite State Machines. To do this, everything you might need to know can be found in this document and in your textbook in the chapters on C. These chapters are: 1, 2, and appendix 1. If something appears to be missing, please do not hesitate to talk with the faculty member or the lab assistants and aid will be provided.

A submission has been added to the course Moodle. Using the Makefile provided, you will create a tar-zipped file and load the submission to the Moodle.

The goal here is to become more familiar with coding in C, so you will not need to do anything fancy. You will not need any special libraries, just work with the basic constructs of C. Also, it is advised to rely on the provided help materials first. This also means you should not use the C string handling library.

What is a Finite State Machine.

For theoretical computers, there are three basic types of abstract machines: Finite State Machines (FSM), Push Down Automata (PDA), and Turing Machines (TM). Each machine type has a more powerful memory than the one before it. FSMs can only remember one thing at a time, indicating the current system state. PDAs can remember multiple things, but the information is stored in a stack, so only the most recent information is available and to get at older information new things need to be forgotten. TMs have infinite memories and are simple models of modern-day computers, thus all information encountered is stored in a central memory (or computer tape).

For this lab you will be constructing four FSMs that accept an input string to determine if it is valid or invalid, these are called **acceptors**. There are other types of state machines that produce outputs based on the current state and input, these are called **transducer** machines.

As these are abstract machines, we will define them mathematically. These machines do not use calculus in any way, instead, these machines rely on discrete math. A Finite State Machine is defined as a set of sets and a function:

$$FSM = \{S, I, g, s_0, A\}$$

S – is a set of states, given as {$\mathbf{S_0}$, $\mathbf{S_1}$, $\mathbf{S_2}$, $\mathbf{S_3}$, $\mathbf{S_4}$, … }

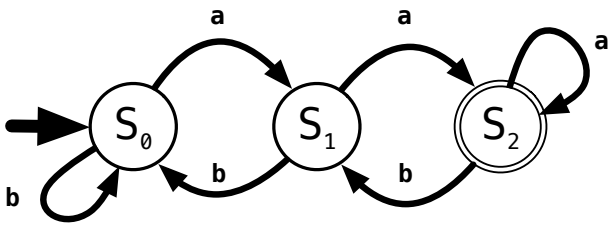
I – input alphabet, given as a set of characters {**a**, **b**, **c**, **d**, … }

g – a transition function that accepts the current state, the current input, and determines the next state. Often this is provided as a transition table (also called a lookup table).

s_0 – the initial state where the FSM begins.

A – a set of acceptor states that when the FSM ends in to indicate that the input string was valid. So if the FSM runs through the entire sequence of inputs and it ends in one of these acceptor states, then the machine returns that the input string is **valid**. If the machine does not end in an acceptor state, then the input string will be considered **invalid**.

An example of a simple FSM is the following diagram.



In the diagram, each circle is a specific state, defined by S. The large arrow pointing to state-zero (s_0) indicates the starting state. State-two (s_2), with the double ring, is the accepting state. Each labeled arrow indicates the transition taken in the graph for that given input, with the labels specified by the input-alphabet. This graph can be translated into a transition table given below.

	Inputs	
Current-State	a	b
s_0	s_1	s_0
s_1	s_2	s_0
s_2	s_2	s_1

Now, consider a simple input string of "**aa**", this will take the transition sequence of state-zero (s_0) to state-one (s_1), and then state-one (s_2) to state-two (s_2). Once these two transitions have occurred, the machine will stop and finish in the accepting state. Thus, the input string of "**aa**" is valid.

The following two examples would also be valid "**ababaaba**" and "**bbbbaaaaaa**" because they both end in an accepting state of s_2. But if the sequence "**aabbaabb**" where input, then this string would be invalid because the machine would not finish in an accepting state.

Coding a state machine can be done with either nested-conditionals or switch statements in a while loop. In the assignment you will need to use both to trace through a character string (or C-String) as the input string.

The humble C-String.

A character string in C is just an array of **`char`** and can be defined as: **`char string1[5];`**

A C-string literal can be specified in the following way: **`char *string2 = "This is a string.";`**

It is important to understand that the length of **`string1`** is five bytes long, and the length of **`string2`** is eighteen bytes long, with 17 bytes for the visible characters and the last byte being what is called a null-terminator.

The null-terminator is a null-value (or zero) that is used to indicate the termination of a string. So, you can force a null-terminator with the literal '\0'. This is demonstrated below.

```
int main() {

    char *str2 = "This is a string.";  // this is a literal string stored as read-only
    char str1[18];                     // an array to copy the literal into

    for(int cnt = 0; cnt < 18; cnt++) { // the mechanism for copying the string
        str1[cnt] = str2[cnt];
    }

    puts(str2);  // these are system calls that will print a string
    puts(str1);

    str1[6] = '\0'; // modifying the copied string with a null-terminator

    puts(str1); // print the new string to show it is truncated

    return 0;
}
```

In the code above, there is one string literal and two variables containing the string. In the code above **str2** stores the literal and **str1** stores a copy of that literal. The system function **puts** will print the passed string. The third to last statement introduces a null-terminator into the copied string. When this program is run, the following is printed out.

This is a string.
This is a string.
This i

The terminator is placed in array location 6, or the seventh position, allowing only the first six characters to be printed. Thus, the literal and copied strings look like the following when the program terminates.

str1	T	h	i	s		i	s		a		s	t	r	i	n	g	\0
str2	T	h	i	s		i	\0		a		s	t	r	i	n	g	\0

You might notice that both single and double quotes are used in the code above. Double quotes can be zero or more characters in length, but there will always be one extra array location for the null-terminator. So, the string literal **""**, representing an empty string will still be one byte for the terminator. When there are single quotes, this represents a single character, and a null-terminator is not used.

The different ways to manipulate a C-String.

Pointers can be very useful for manipulating strings, but you will find that instead of pointers array brackets can be used. Below are examples of a length function using both array and pointer operations.

```
int len1(char *str) {

    int cnt = 0;

    while(str[cnt]) { // here we are looking for the null-terminator to generate
        cnt++;        // a false condition
    }

    return cnt;
}

int len2(char *str) {

    int cnt = 0;
```

```
    while(*(str + cnt)) { // here pointer arithmetic is used to index the string
        cnt++;            // once the pointer is determined it the dereferenced for the value
    }

    return cnt;
}
```

In both cases, the string is being iterated using the **cnt** variable as the index. In the first situation the value at that location is determined using the array operation. In the second situation the index is an offset from the start of the array generating a pointer location, then the * operator is used to obtain the value. In both cases, the while loop will terminate when the null-terminator is found. A third approach is given below.

```
int len3(char *str) {

    int cnt = 0;

    while(*str) {
        cnt++;
        str++;
    }

    return cnt;
}
```

In this approach, the variable **str** (which is the pointer) is being incremented. Using this code, the pointer is stored locally to the function and the value is incremented to find the next pointer in the sequence.

If each of these different cases, to following code will run each situation:

```
printf("%d\n", len1("123456"));
printf("%d\n", len2("123456"));
printf("%d\n", len3("123456"));
```

producing the same results:

```
6
6
6
```

From these three examples, it is clear there are different approaches. For the lab, choose one, and stick with that approach. As we move forward, we will reexamine these different approaches for what is happening in detail.

To examine specific values in a string you can use the character literal, as shown below.

```
char *str2 = "This is a string.";
char str1[18];

if(str1[0] == 'T')
    puts("That is a T.");

if(*(str2 + 11) == 't')
    puts("That is definitely a t.");
```

This code will produce the following:

```
That is a T.
That is definitely a t.
```

You can also use switch statements to determine the value of a particular character.

```
char *str2 = "This is a string.";

switch (str2[4]) {

    case ' ':
        puts("location 4 is a space");
        break;

    case 't':
        puts("location 4 is a t character value");
        break;

    default:
        puts("I do not know");
}
```

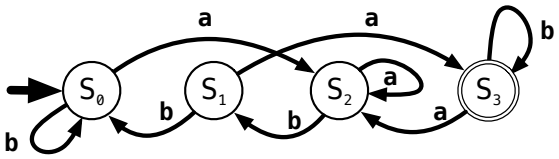
With this code, the output will be “location 4 is a space”.

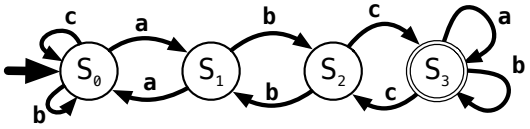
Implementing the lab.

For this lab implement the following four different state machines and run them for eight different input strings, four strings that produce a valid result and four that produce an invalid result. For the first two machines use only conditionals, then for the second two machines use only switch-statements. Each machine will be implemented as a separate function, which is called from the main. This is already set up in the provided program file.

In the comments before each function, specify the lookup table in the comments for the associated FSM that you are implementing in the associated function.

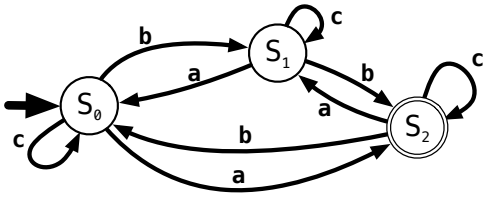
FSM1

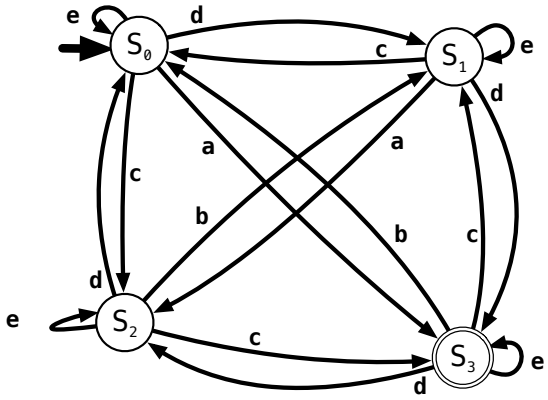




FSM2

FSM3





FSM4

Requirements.

- In the provided source file, there will be implemented states (as enum data types) and input alphabets as global char variables. Use these global tools to implement your code.
- The program general structure is provided to you, please use this as a starting point, but you should only need to fill in the existing functions and enhance the main.
- You will not use any libraries other than standard-io to help you, which is already included in the provided file.
- Do not use the string library; implement the program using only core C functionality.
- Provide the lookup table for each FSM in the comments above the function that implements that machine.
- FSM1 and FSM2 will use nested-conditionals, while FSM3 and FSM4 will use switch-statements to implement these machines.