50.051 Programming Language Concepts

W9-S1 Some practice about coding FSMs in C

Matthieu De Mari

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Finite State Machine

Definition (Finite State Machine):

A **Finite State Machine (FSM)**, or **finite automaton**, is a mathematical model used to represent systems

- that have a **finite number of possible states**,
- and **can transition between these states based on given inputs**.

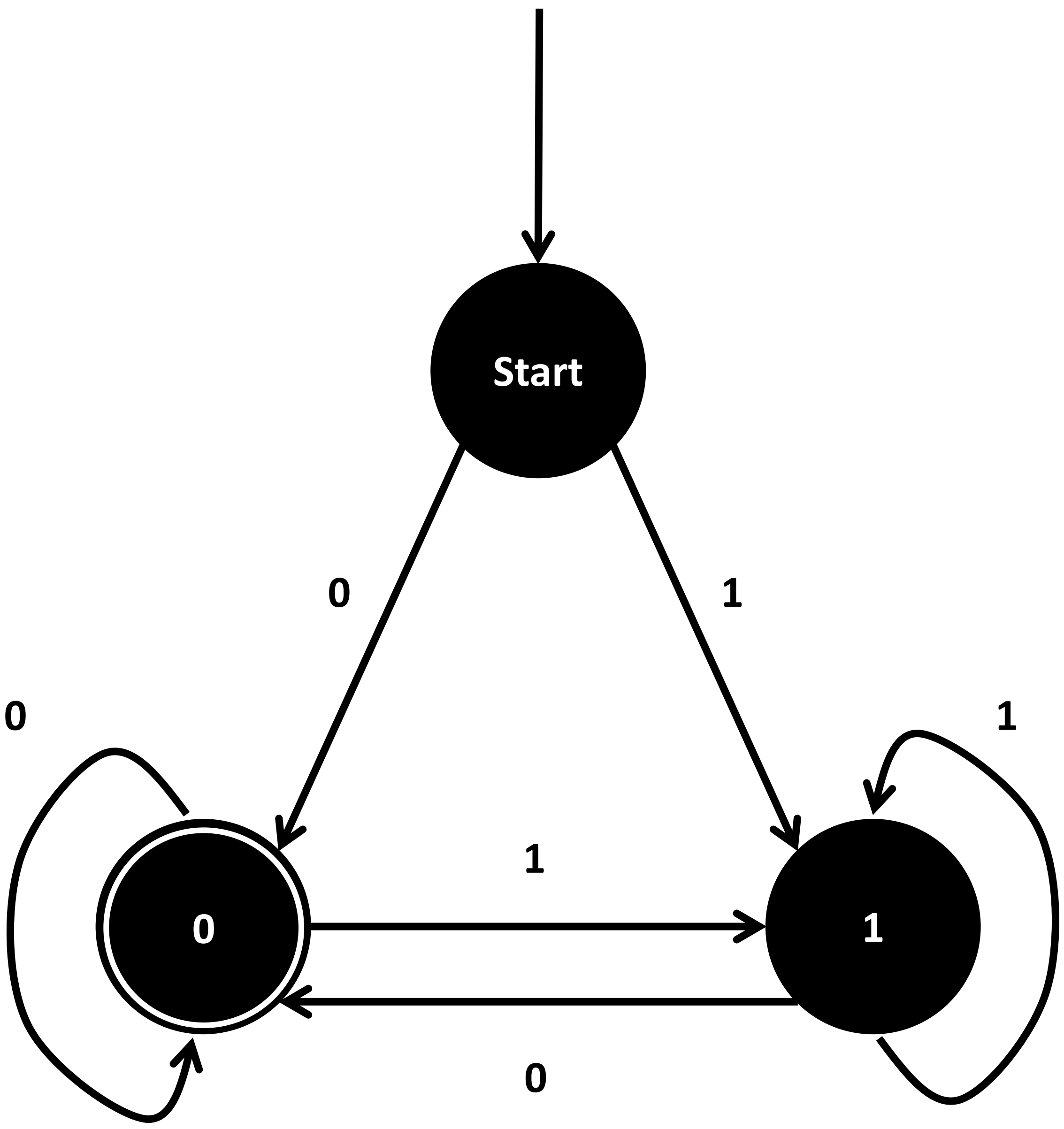
An FSM can be represented using a **graph** representation, known as a **state diagram**, which shows the possible states of the system and the transitions between them.

FSMs are used in a wide variety of applications (control systems, communication protocols, digital circuits, etc.). **In our case, FSMs are at the center of the compiling process.**

Elements of an FSM with stopping states

In order to define a FSM with stopping state, we keep the previous FSM elements:

1. A **finite set of states *S***.
2. A **finite set of inputs or actions *A***.
3. A **starting state s_0** $\in$ ***S***.
4. A **transition function f**, or **transition table**, which describe the **transition logic** in the FSM.



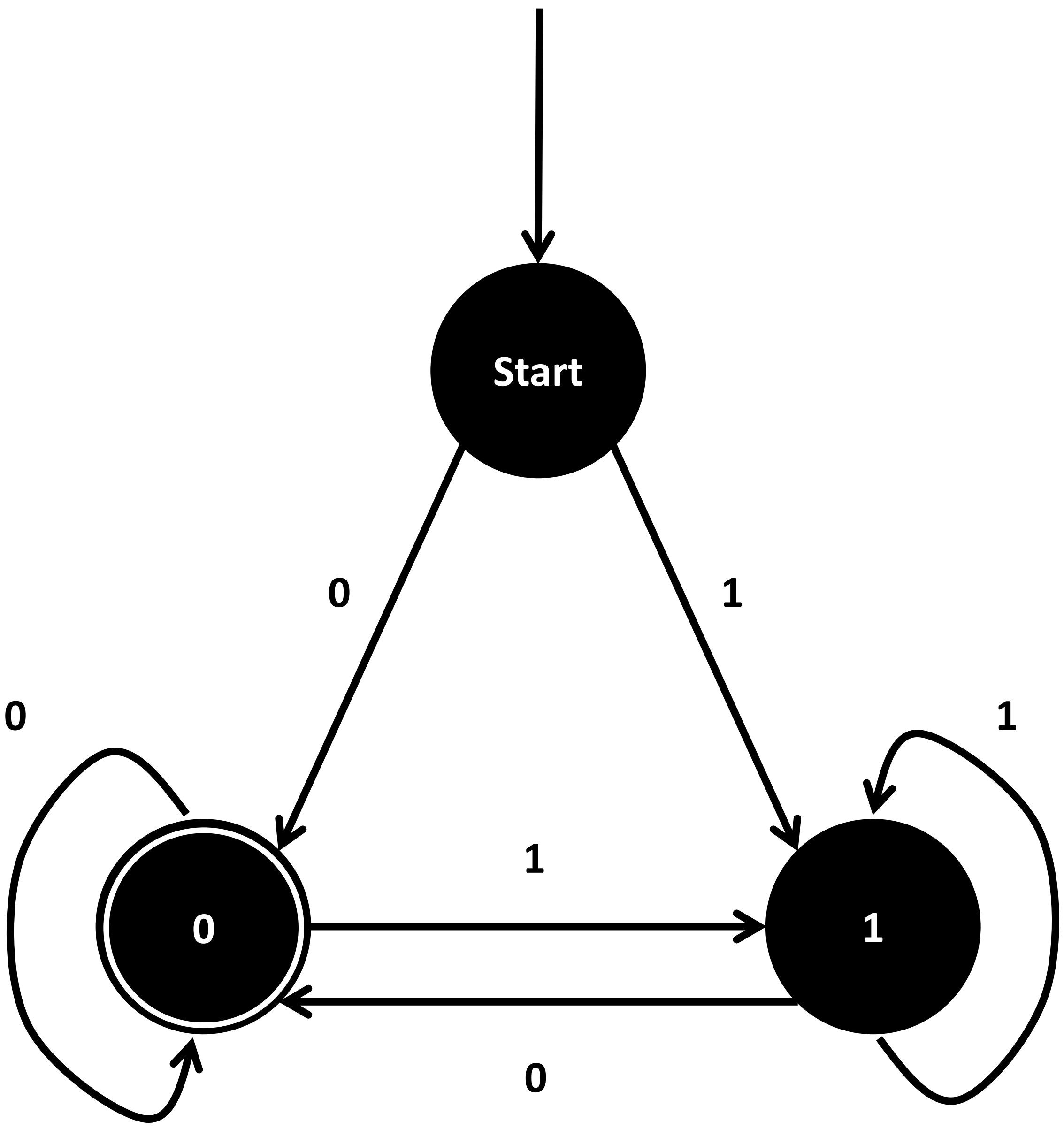
Elements of an FSM with stopping states

In order to define a FSM with stopping state, we keep the previous FSM elements.

5. And we add **a finite set of stopping states F**, defined as a subset of **all possible states S**, i.e. $F \subseteq S$.

In our example, we simply have

$$F = \{0\}$$

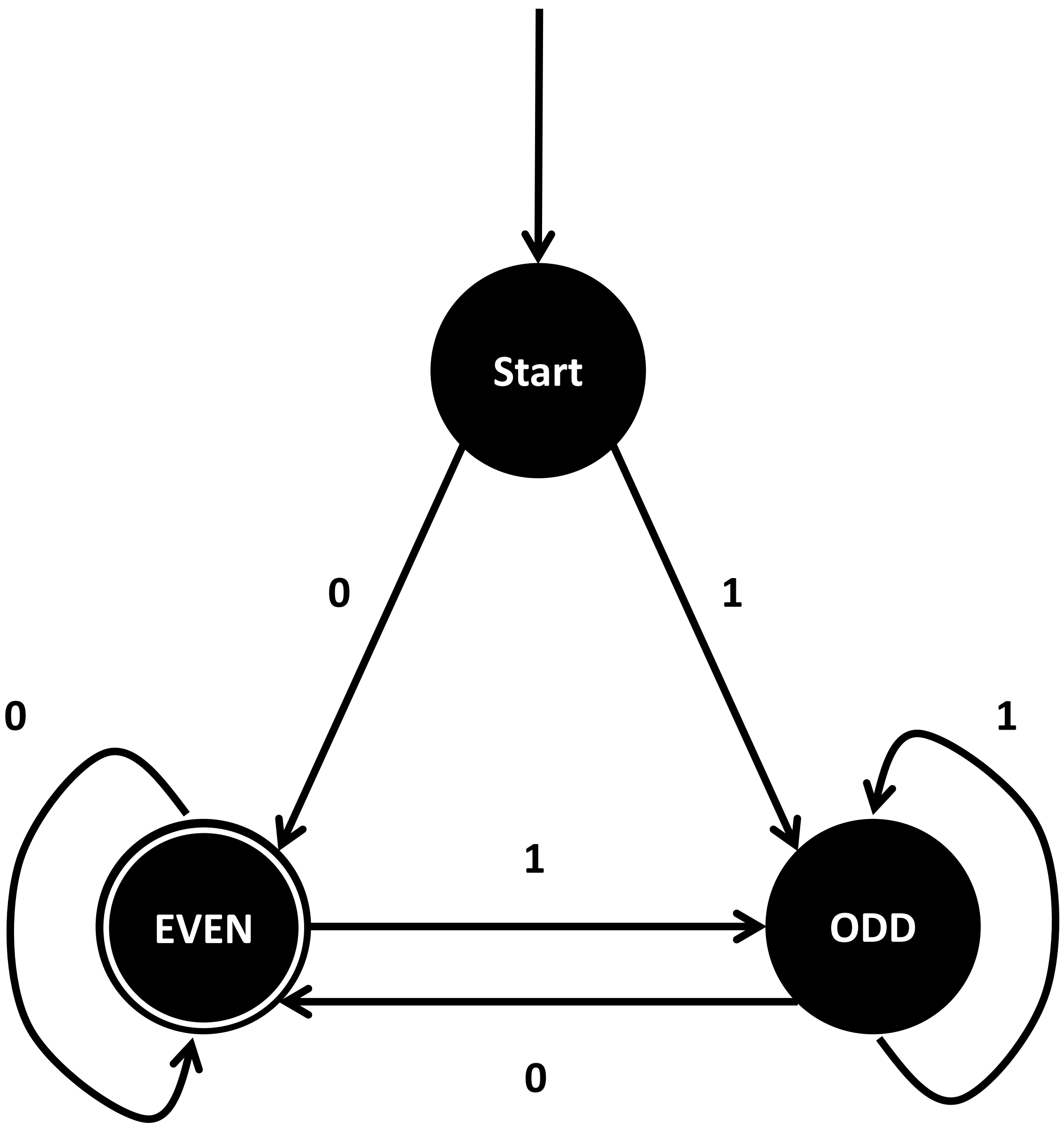


Practice FSM
(a blast from the past!)

Let us consider this FSM, defined on the right.

We have seen that it considers as acceptable inputs the binary strings *s* whose bit of least importance is 0.

This means that the integer number input represented by the string *s* needs to be even to be acceptable.

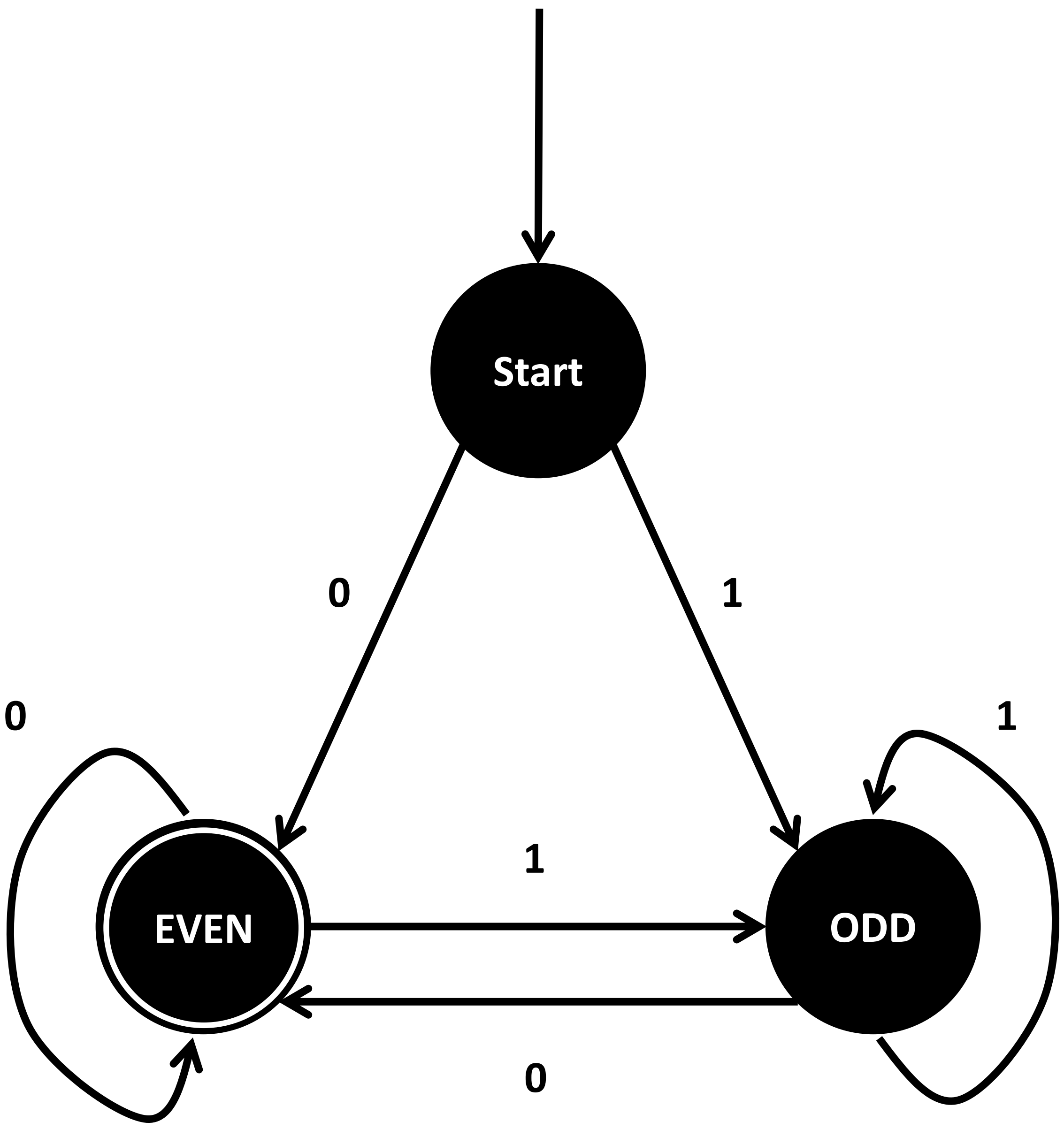


Restricted

Practice FSM
(a blast from the past!)

Let us try to implement this FSM in C now!

(A guided implementation, first.)



Coding a basic FSM step-by-step

First of all, we need to define:

- A **finite set of states *S***.

Here, we have decided to assemble all three possible states (START, EVEN, ODD) for our FSM into an **enum**.

```c
#include <stdio.h>

// Define possible states as an enum
typedef enum {
    START,
    EVEN,
    ODD
} State;

```

Coding a basic FSM step-by-step

Then, we need to define:

- An FSM, which keeps track of the different states it is in, and has a **starting state** $s_0 \in S$.

Here, we have decided to define our FSM object as a **struct**, with only one attribute, being the current state of the FSM.

```c
#include <stdio.h>

// Define possible states as an enum
typedef enum {
    START,
    EVEN,
    ODD
} State;

```

Coding a basic FSM step-by-step

Then, we need to define:

- An FSM, which keeps track of the different states it is in, and has a **starting state** $s_0 \in S$.

Later on, we will initialize our FSM, using START as the starting state.

```
// Main
int main() {
    // Initialize FSM
    FSM f = {START};
```

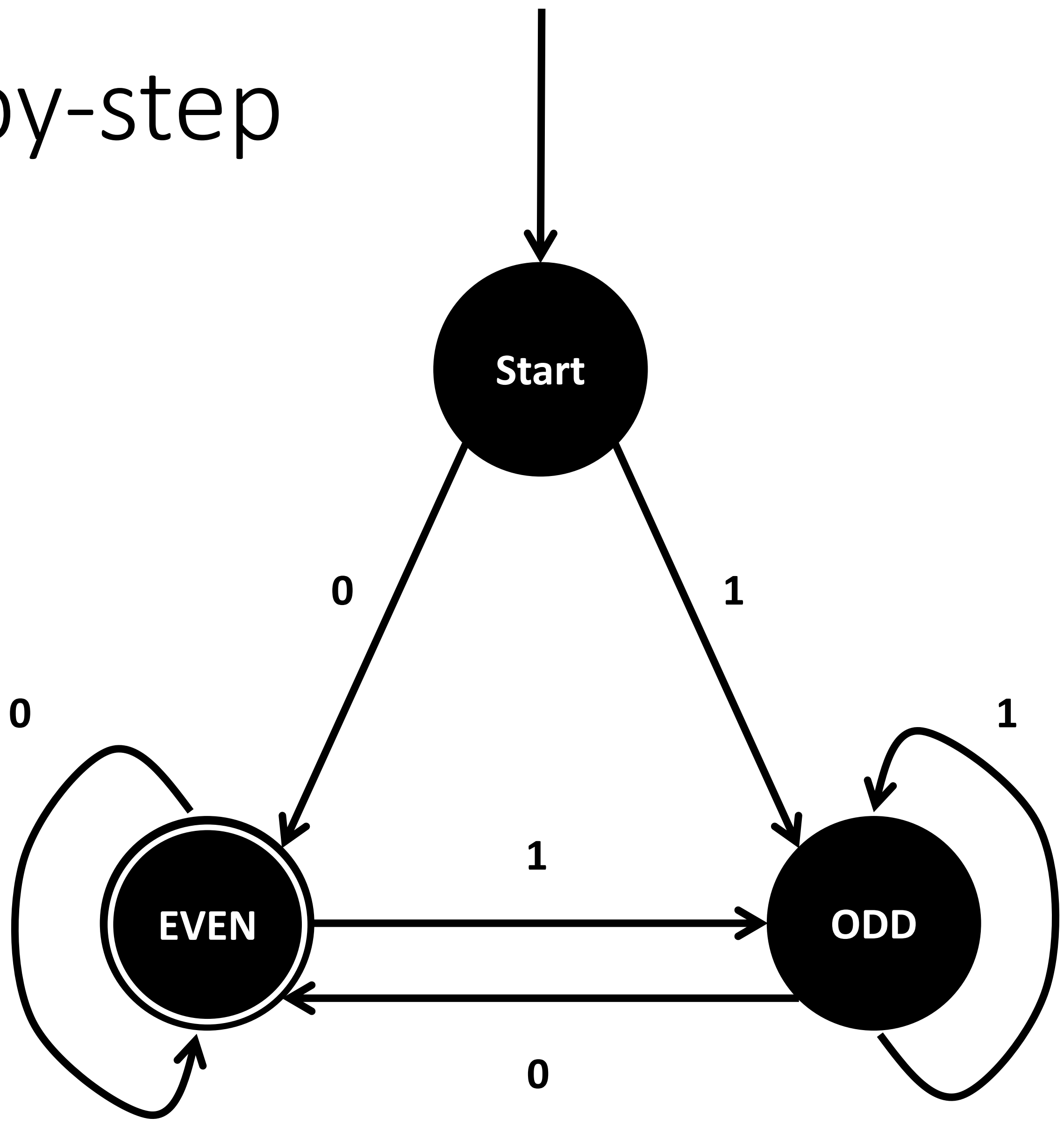
Coding a basic FSM step-by-step

Then, we need to define:

- A **transition function *f***, which describe the **transition logic** in the FSM.

Here, the transition logic is simple,

- Go to EVEN if you see action 0,
- Go to ODD if you see action 1,
- No matter what the current state is.



Coding a basic FSM step-by-step

We implement this logic using a simple *if/else* structure in an update function.

- It takes our FSM,
- It also takes our input character or action,
- It then updates the current state attribute of our FSM struct, following the logic we have established.

```c
// Define our FSM transition function.
void update_state(FSM *f, int input) {
    if (input == 1) {
        f->current_state = ODD;
        printf("New state is odd.\n");
    }
    else if (input == 0) {
        f->current_state = EVEN;
        printf("New state is even.\n");
    }
}
```

Coding a basic FSM step-by-step

We can then define our input string *s* in our main function for testing.

(Or we could ask the user for an input string using a *scanf()*…)

```c
// Main
int main() {
    // Initialize FSM
    FSM f = {START};

    // Decimal representation of an int (our input)
    int input[] = {1, 0, 1, 1, 0};
```

Restricted

Coding a basic FSM step-by-step

We will then use a for loop for the appropriate amount of iterations *n* to browse through all the characters in the input string one at a time, updating the state of our FSM every time.

```
// Run for loop on each character of our input
int n = sizeof(input) / sizeof(input[0]);
for (int i = 0; i < n; i++) {
    // Update state for each possible input value
    update_state(&f, input[i]);
}
```

Coding a basic FSM step-by-step

Eventually, a final display showing a print corresponding to the final state (using a switch this time, because why not).

```
    // Final display
    switch (f.current_state) {
        case EVEN:
            printf("Our final state tells us the number is even.\n");
            break;
        case ODD:
            printf("Our final state tells us the number is odd.\n");
            break;
        default:
            break;
    }
```

Using a transition table instead

In the previous implementation, we have used a transition, which implements the transition logic using an *if/else* statement.

```c
// Define our FSM transition function.
void update_state(FSM *f, int input) {
    if (input == 1) {
        f->current_state = ODD;
        printf("New state is odd.\n");
    }
    else if (input == 0) {
        f->current_state = EVEN;
        printf("New state is even.\n");
    }
}
```

Restricted

Using a transition table instead

We could have, equivalently used a transition table as well.

Define it as a 3 by 2 table (with 3 possible states, 2 possible actions).

```
// Define our transition table for new states
// 3 states (START, EVEN, ODD) and two actions (0, 1)
const State transition_table[3][2] = {
    // 0     1
    {EVEN, ODD}, // START
    {EVEN, ODD}, // EVEN
    {EVEN, ODD}  // ODD
};
```

Using a transition table instead

We could have, equivalently used a transition table as well.

Define it as a 3 by 2 table (with 3 possible states, 2 possible actions).

Use the transition table to find the next state directly.

```
// Define our FSM transition function.
void update_state(FSM *f, int input) {
    // Get the current state from the FSM object
    State current_state = f->current_state;

    // Look up the next state based on the current state and input
    State next_state = transition_table[current_state][input];

    // Update the FSM object with the new state
    f->current_state = next_state;
```

Implementing acceptable states/inputs

Would probably simply require

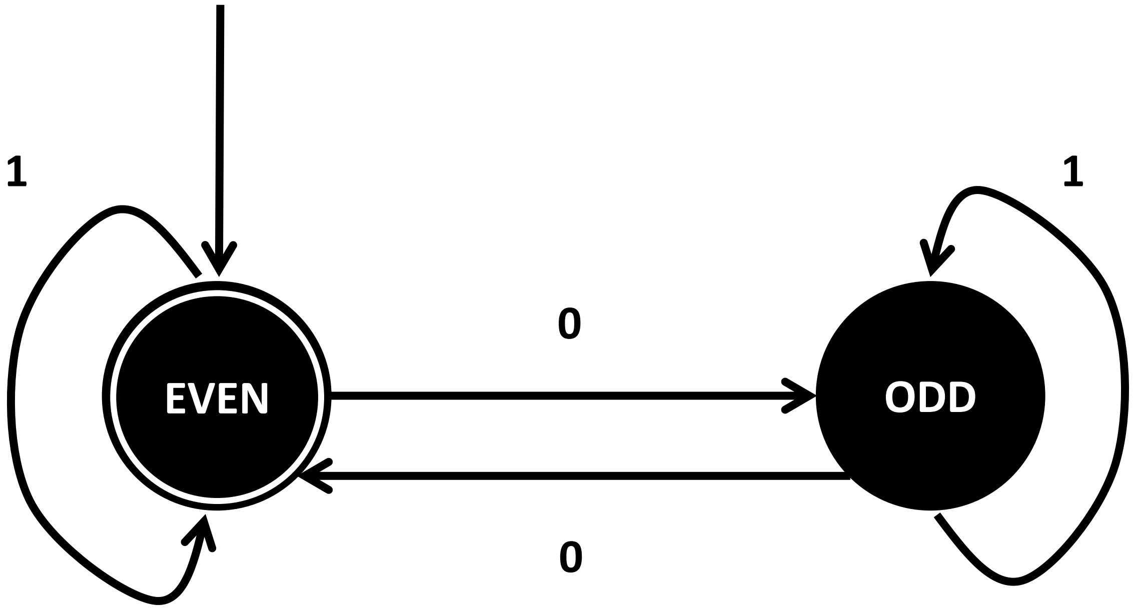
- To define a list of acceptable states,
- To amend the check of the final state and check if the final state falls in the list of acceptable states.

```
    // Final display
    switch (f.current_state) {
        case EVEN:
            printf("Our final state tells us the number is even.\n");
            break;
        case ODD:
            printf("Our final state tells us the number is odd.\n");
            break;
        default:
            break;
    }
```

Practice 1
(another blast from the past)

This used to be practice 7 in the previous lecture.

It is an FSM with a single stopping state, that considers as acceptable inputs any string *x* of 0 and 1, that have an even number of zeroes.

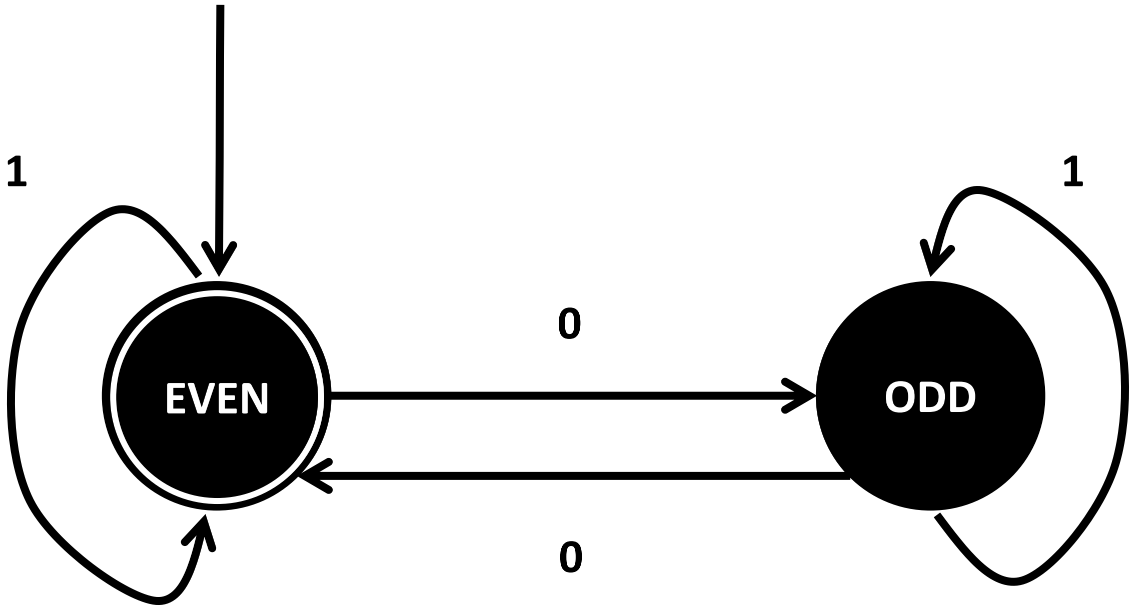


Practice 1
(another blast from the past)

Check the ***main.c*** file in the *“3. Practice 1 template”* folder.

Modify the code to implement this FSM !

(Solution is in folder 4., but no cheating…!)

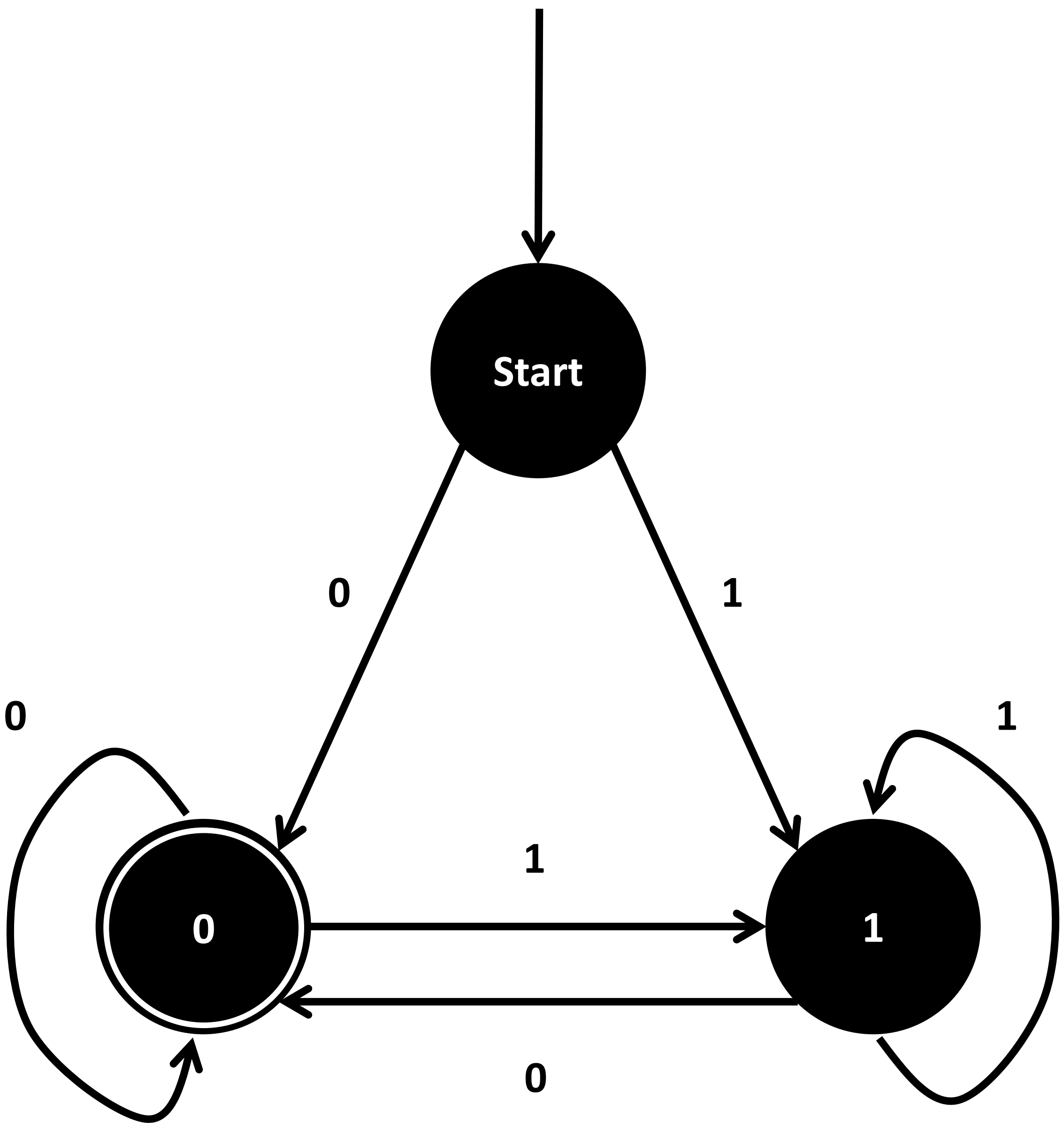


Elements of an FSM with outputs

In general, outputs with stopping or accepting states are useful, but limited in terms of applications.

A stronger version of the FSM consists of the FSM with **outputs**.

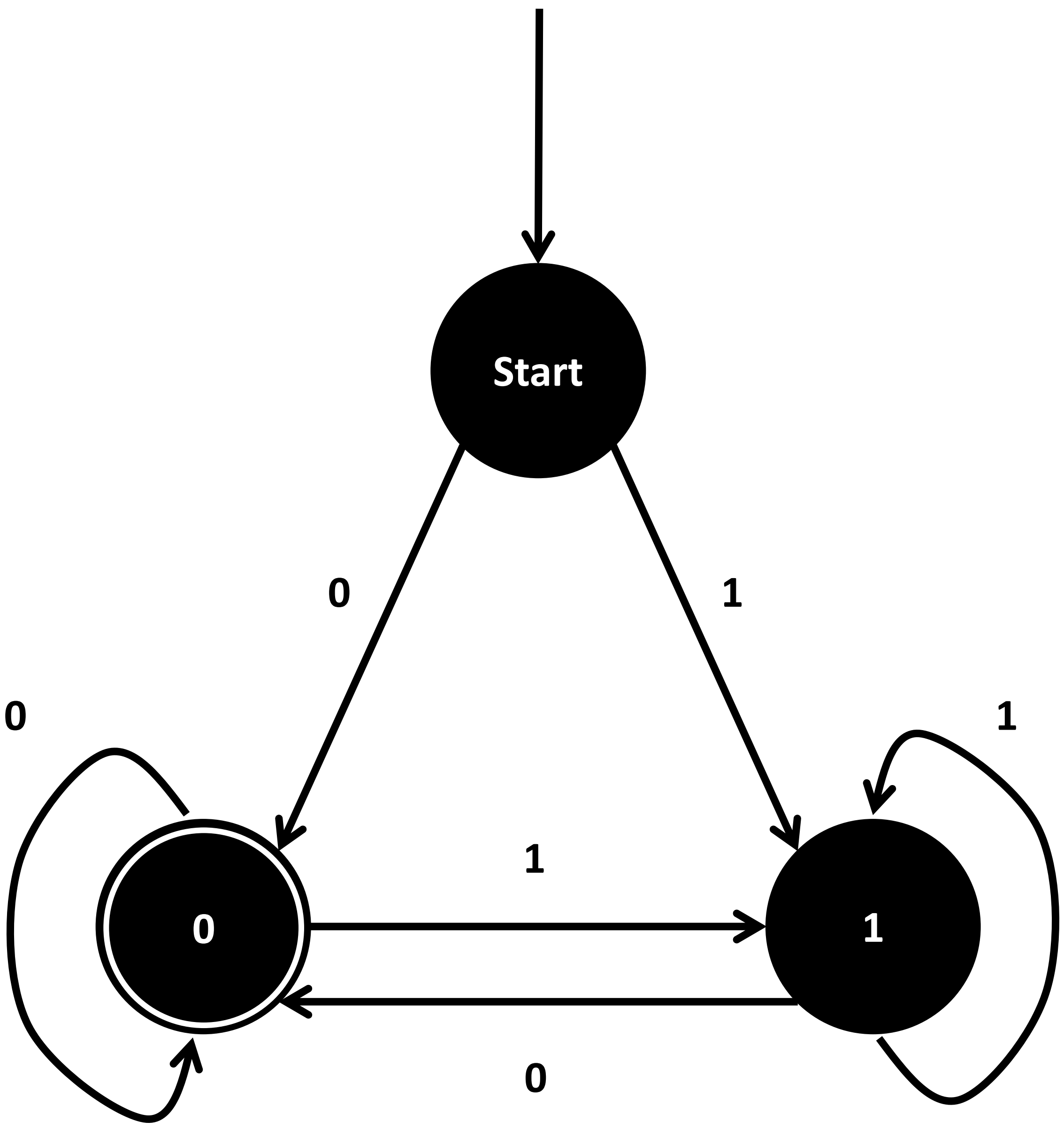
It simply replaces the stopping states with outputs being produced every time an action is taken.



Elements of an FSM with outputs

In order to define a FSM with outputs, we keep the previous FSM elements:

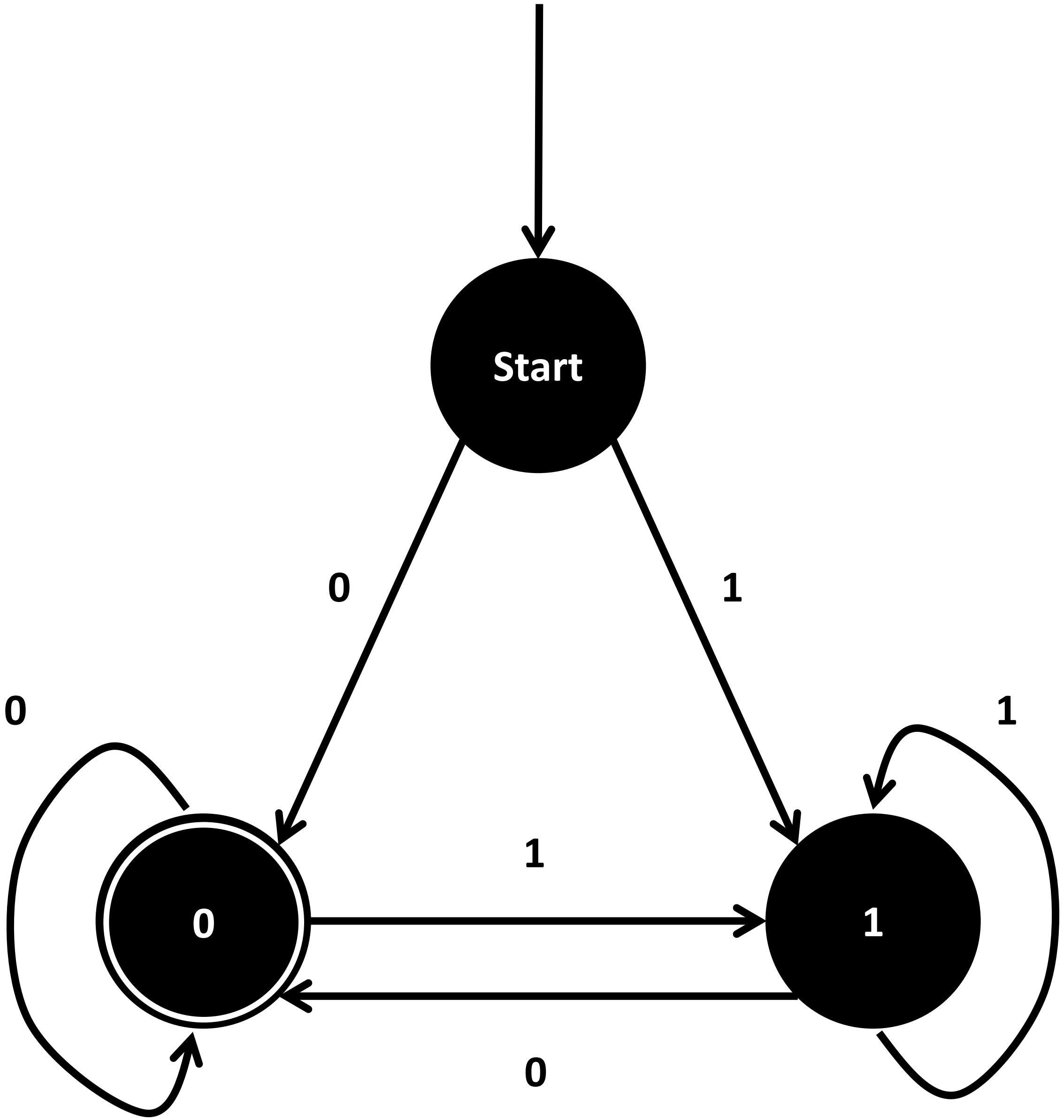
1. A **finite set of states *S***.
2. A **finite set of inputs or actions *A***.
3. A **starting state** $s_0 \in S$.
4. A **transition function** f, or **transition table**, which describe the **transition logic** in the FSM.



Elements of an FSM with outputs

5. And we add **a finite set of possible outputs Y**,
6. And an **output function g**, which decides on an output $y \in Y$ to produce given any action $a \in A$ taken in any given state $s \in S$.

$$g: S \times A \rightarrow Y$$
$$g(s, a) = y$$



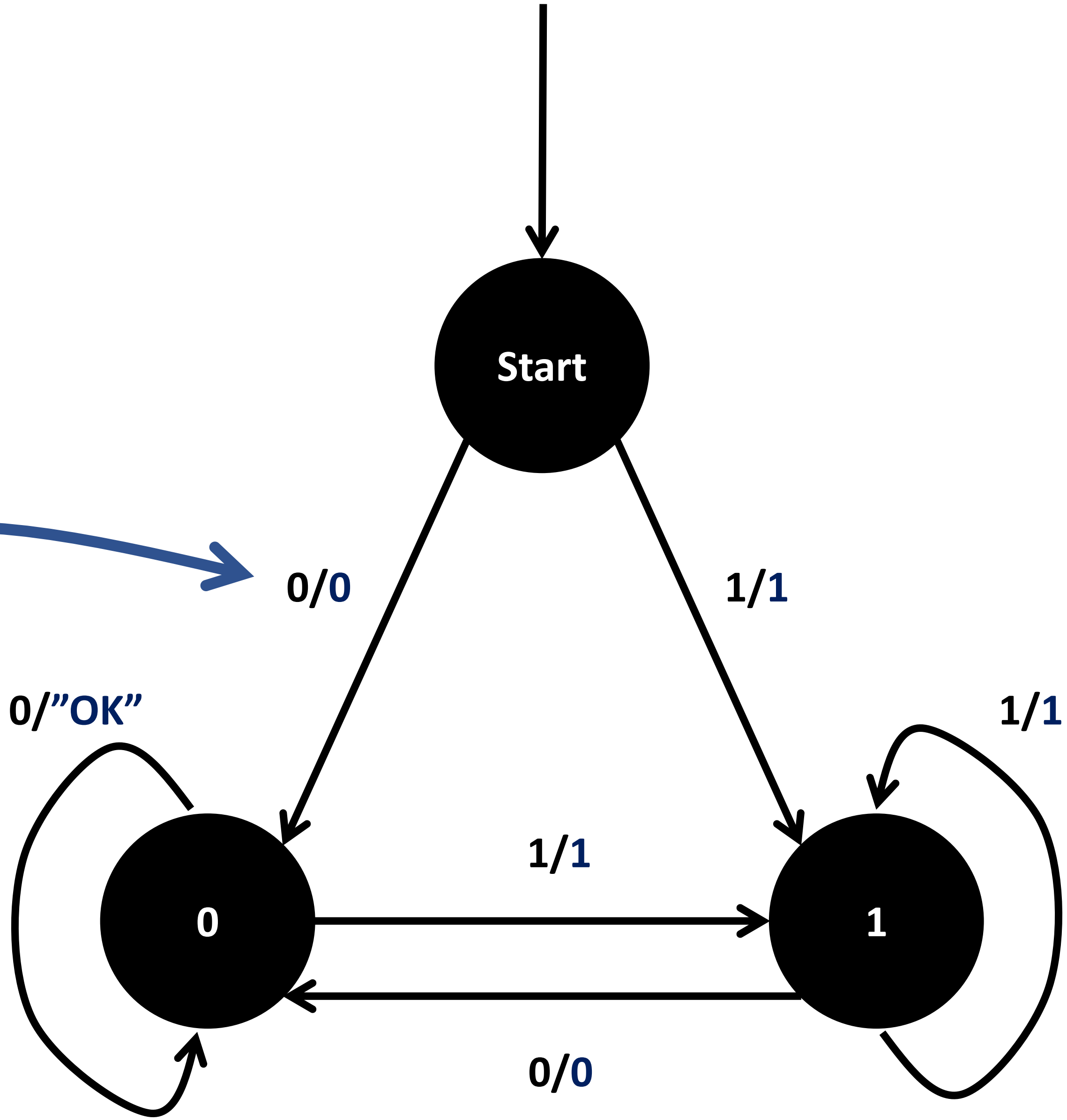
Elements of an FSM with outputs

Outputs are then added using the "a/y" notation on each of the links of the FSM.

In the FSM on the right, the output set Y is defined as

$$Y = \{0, 1, OK\}$$

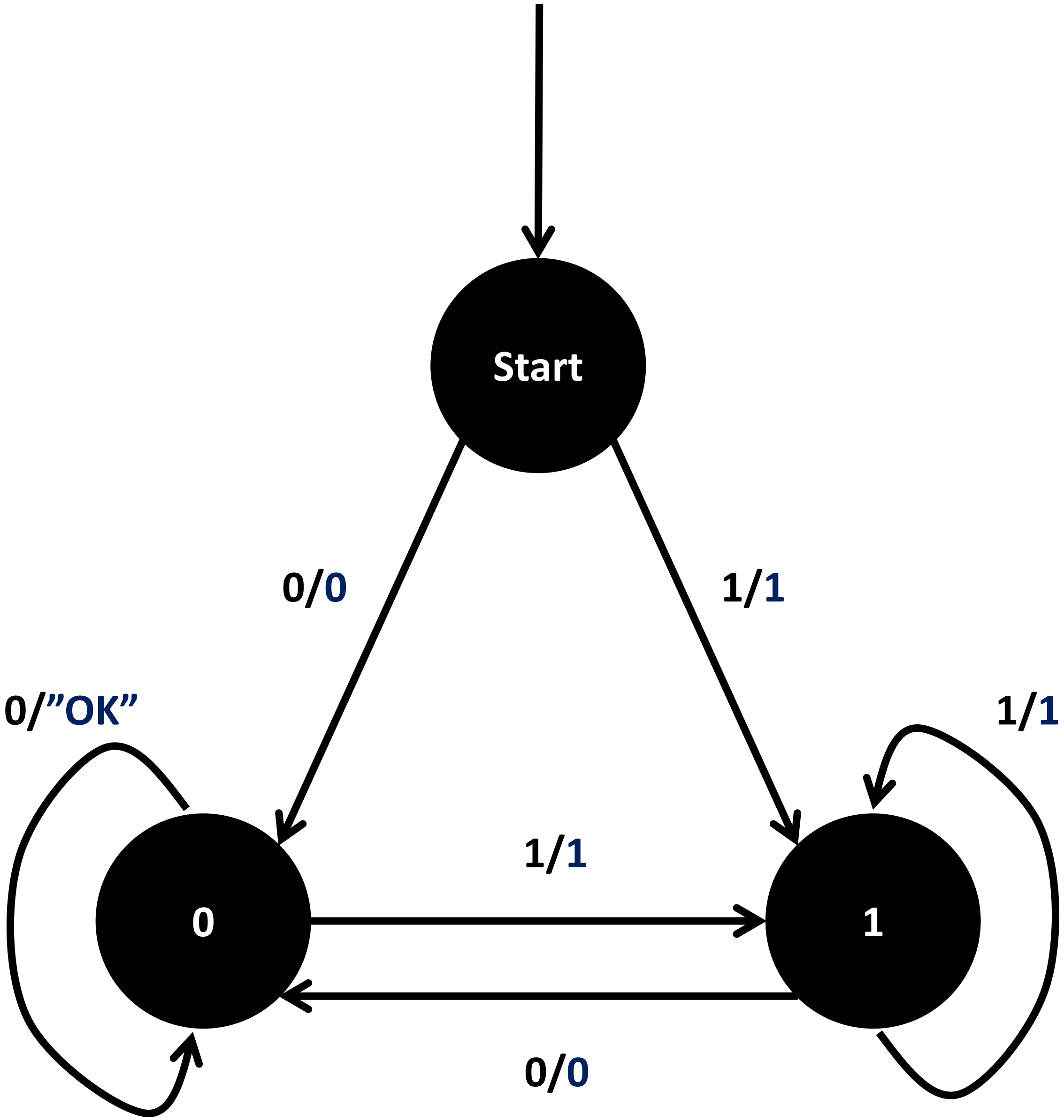
When on start node, using action 0 produces an output 0.



Elements of an FSM with outputs

Could also define outputs in the form of a **table of values** to be produced if a given action *a*, is taken in a state *s*.

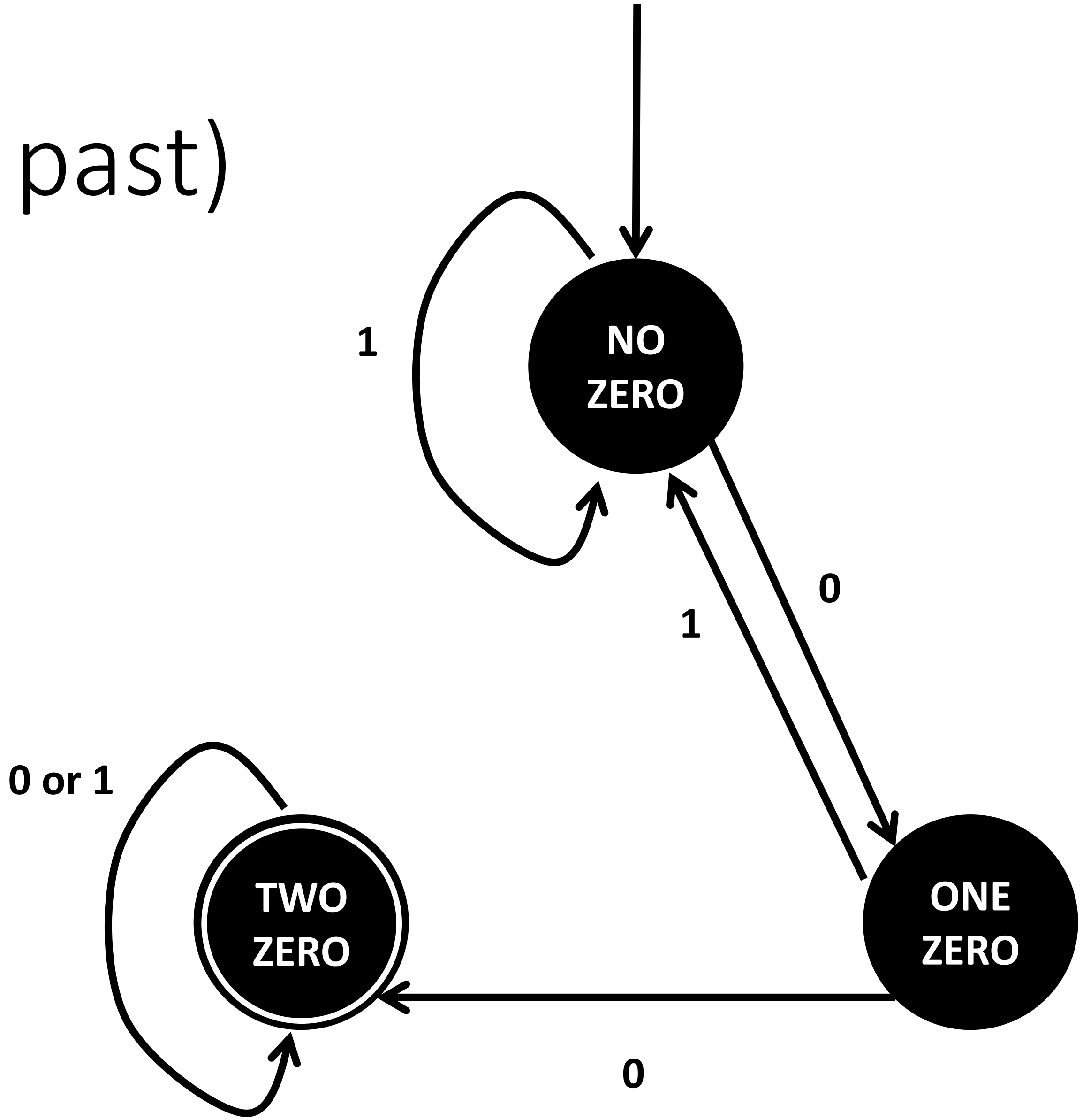
Similar to the **transition table** from earlier, which gave us the new state *s'* if a given action *a*, is taken in a state *s*.



Practice 2
(another blast from the past)

We want to design an FSM which checks if two successive zeroes appear in the binary input string *s*.

- It can be implemented as a simple FSM with an acceptable state being TWO_ZEROES.
- The code for this FSM is shown in Folder 5.

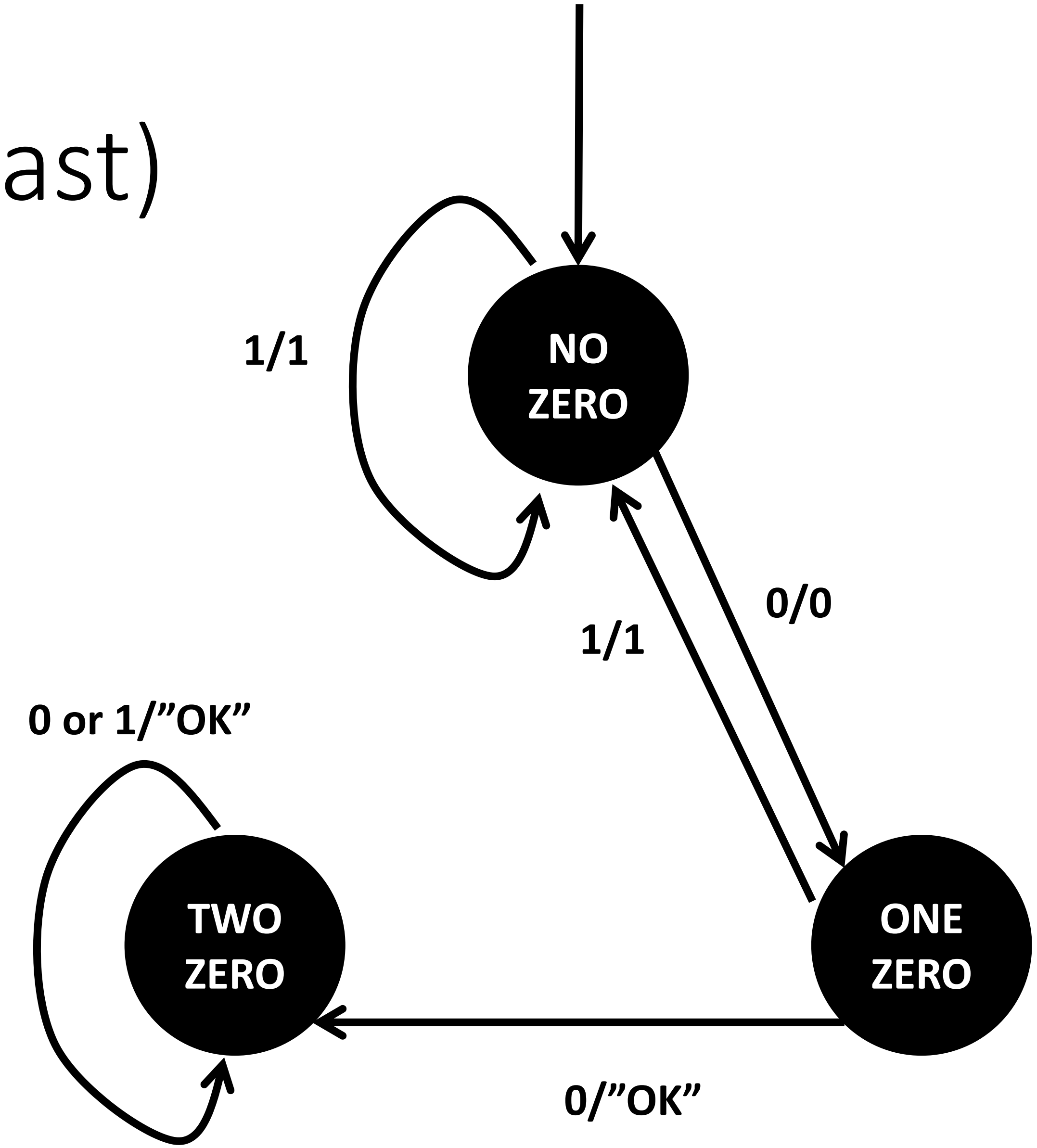


Practice 2
(another blast from the past)

Your objective is to rewrite this FSM so that it produces outputs.

More specifically,

- It will produce 0 (resp. 1) as output if the input is 0 (resp. 1),
- With the exception of the state ONE ZERO and input 1, which produces an output “OK”.
- It also produces “OK” when state is already TWO ZERO.

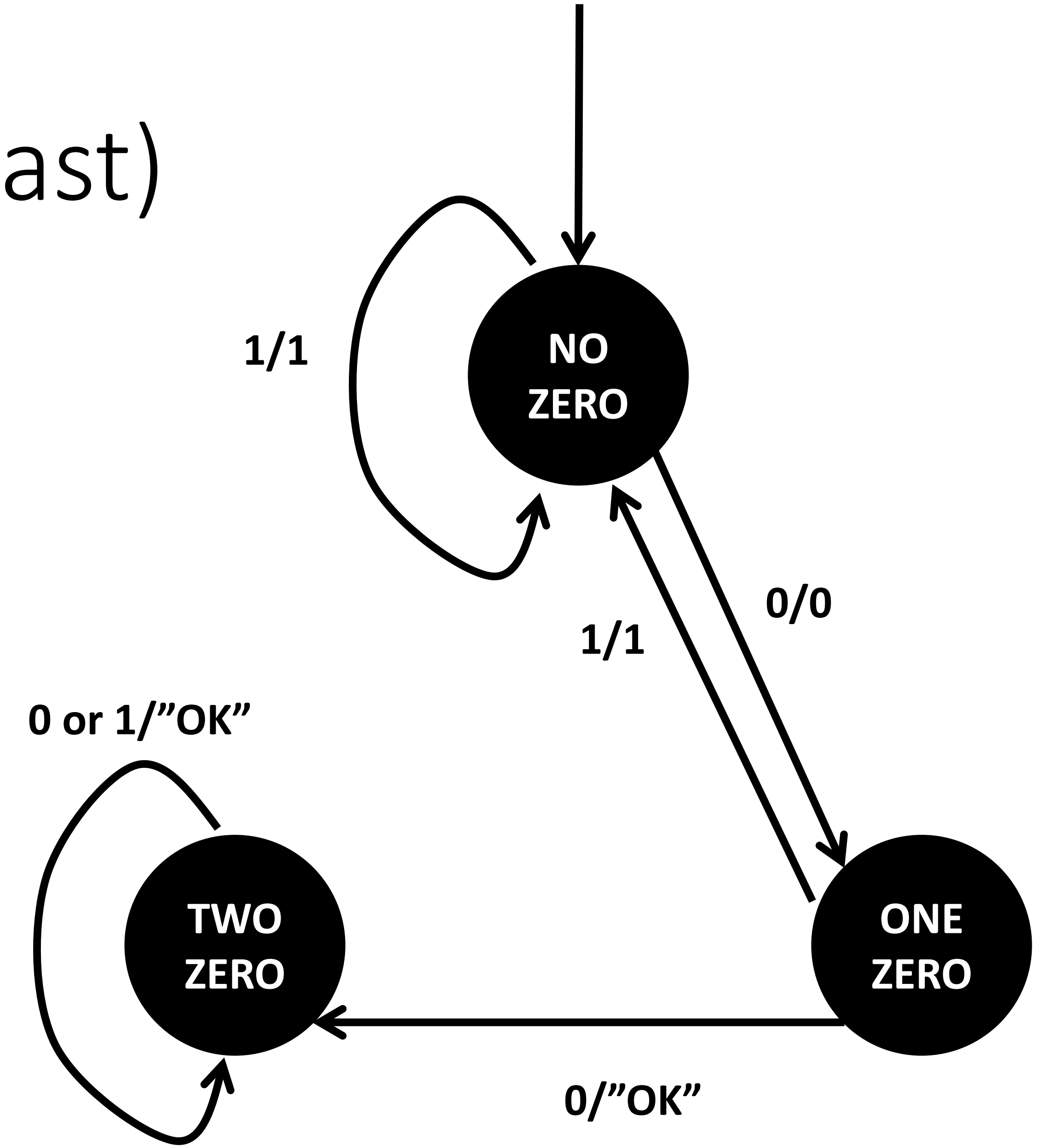


Restricted

Practice 2
(another blast from the past)

You are free to implement these outputs as an extra function or a table of some sort.

Extra kudos if your FSM stops early (i.e. it stops checking characters in the input string *s*, whenever we have seen two zeroes in succession).



Technically, all the other activities can be used as practice!

For those of you who are faster than the rest of the pack.
(Might release the answers to those one day…)

Practice 2: a simple FSM for word recognition

We would like to write an FSM with stopping states that will take strings *x* consisting of combinations of four characters: S, U, T and D.

Possible combinations for the string *x* include, among many others, “USD”, “SUUUUTD”, and the only acceptable input “SUTD”.

Draw a FSM state diagram, which:

- Has 7 possible States (Start, S, U, T, D, Valid, Invalid),
- Has 4 possible Actions (S, U, T, D),
- Has the Start state defined as the starting state,
- Has the Valid state defined as the only stopping state,
- Has the FSM stop in this state, <u>if and only *x* is SUTD</u>; otherwise, it stops in another state (Invalid or something else).

Practice 5

We would like to write an FSM with stopping states and no outputs that will take strings *x* consisting of combinations of four characters: Z, A, and M.

Possible combinations include "MAZ", "AMAZ" and the only acceptable input "ZAMZAM".

Draw a FSM state diagram, which:

- Has possible States, which you are free to decide,
- Has 3 possible Actions (Z, A, M),
- Has the Start state defined as the starting state,
- Has one stopping state,
- Has the FSM stop in this state, <u>if and only *x* is "ZAMZAM"</u>; otherwise, it stops in another state.

Practice 6

We would like to write an FSM with stopping states and no outputs that will take strings *x* consisting of combinations of four characters: Z, A, and M.

Possible combinations include “MAZ”, “AMAZ” and the only acceptable input “ZAMZAM”.

Draw a FSM state diagram, which:

- Has possible States, which you are free to decide,
- Has 3 possible Actions (Z, A, M),
- Has the Start state defined as the starting state,
- Has one stopping state,
- Has the FSM stop in this state, if and only *x* **contains** the string “ZAM”; otherwise, it stops in another state.

Practice 8

Design an FSM that detects whether a binary input string *x* has an alternating bit pattern (e.g., "01010101" or "10101010").

The FSM should end in an accepting state if and only if the input string follows an alternating pattern.

Practice 4

Consider a vending machine and describe it as a FSM with outputs.

It takes three possible actions.

- “0.5”: insert a 50 cents coin,
- “1”: insert a 1 dollar coin,
- “B”: press the machine button.

It also has four possible outputs:

- “0.5”: give back a 50 cent coin to the user,
- “1”: give back a 1 dollar coin to the user,
- “B”: give a chocolate bar to the user,
- “N”: do nothing.

Practice 4

We would like to define a vending machine that has the following logic.

- Whenever a coin is inserted by the user, the total balance is updated.
- If the user has insert 1.5 dollars in total and presses the button, a bar will be given and the balance will return to 0.
- If the user presses the button but the balance is not yet 1.5 dollars, nothing happens.
- If the user inserts a coin and the new balance exceeds the maximal allowed balance of 1.5 dollars, then the machine will return the last coin the user has inserted.

Question: What could the possible states for this FSM be? Draw a state diagram for this FSM.

Practice 10 (Final boss)

Create an FSM that accepts a string input *x* and checks if it meets specific password criteria, define below.

- It should have at least a minimum length of 8,
- At least one uppercase letter,
- At least one lowercase letter,
- And at least one digit.

The FSM should have an "accepted" state if the input string meets all the criteria.