

50.051 Programming Language Concepts

W5-S1 Reminders on Finite State Machines (FSMs)

Matthieu De Mari



SINGAPORE UNIVERSITY OF
TECHNOLOGY AND DESIGN

Finite State Machine (as seen in DDW)

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.**

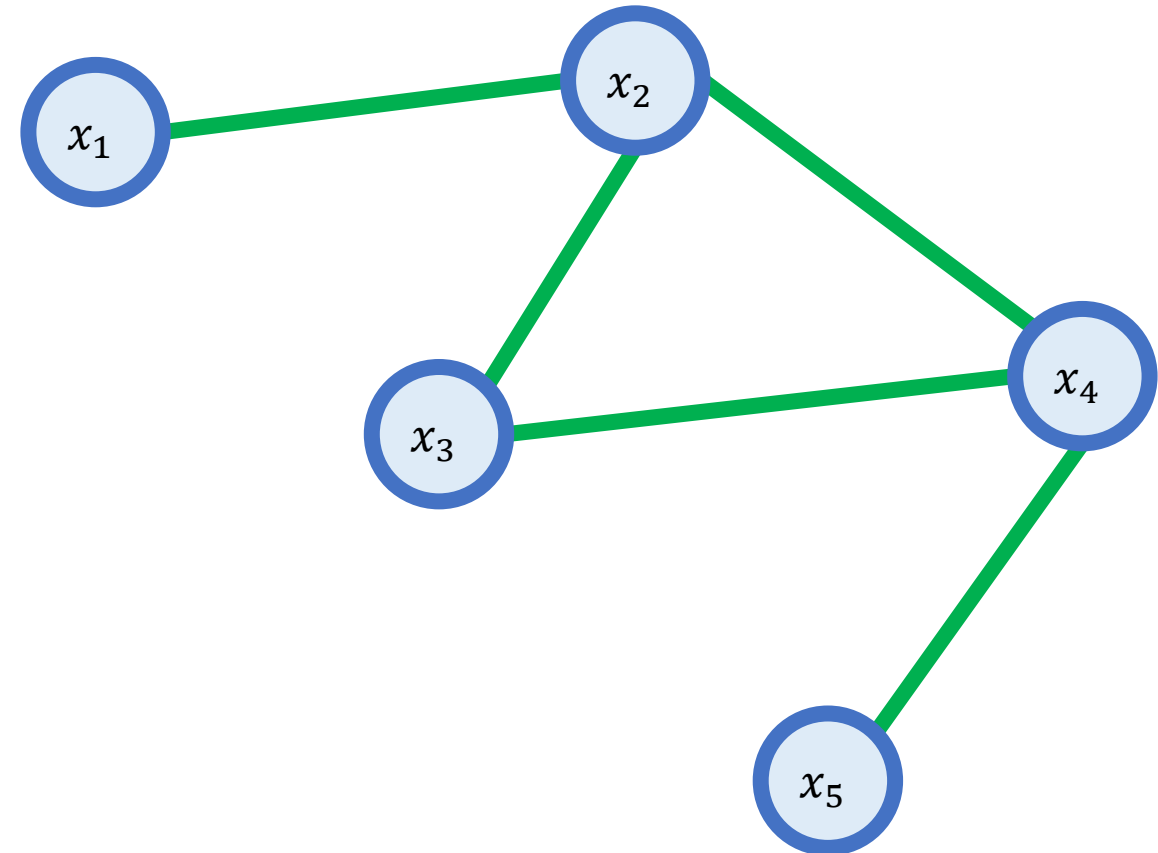
Graph theory

Definition (graph theory):

In mathematics, **graph theory** is the study of graphs objects, which are mathematical structures used to model pairwise relations between objects.

Graphs are one of the principal objects of study in **discrete mathematics**.

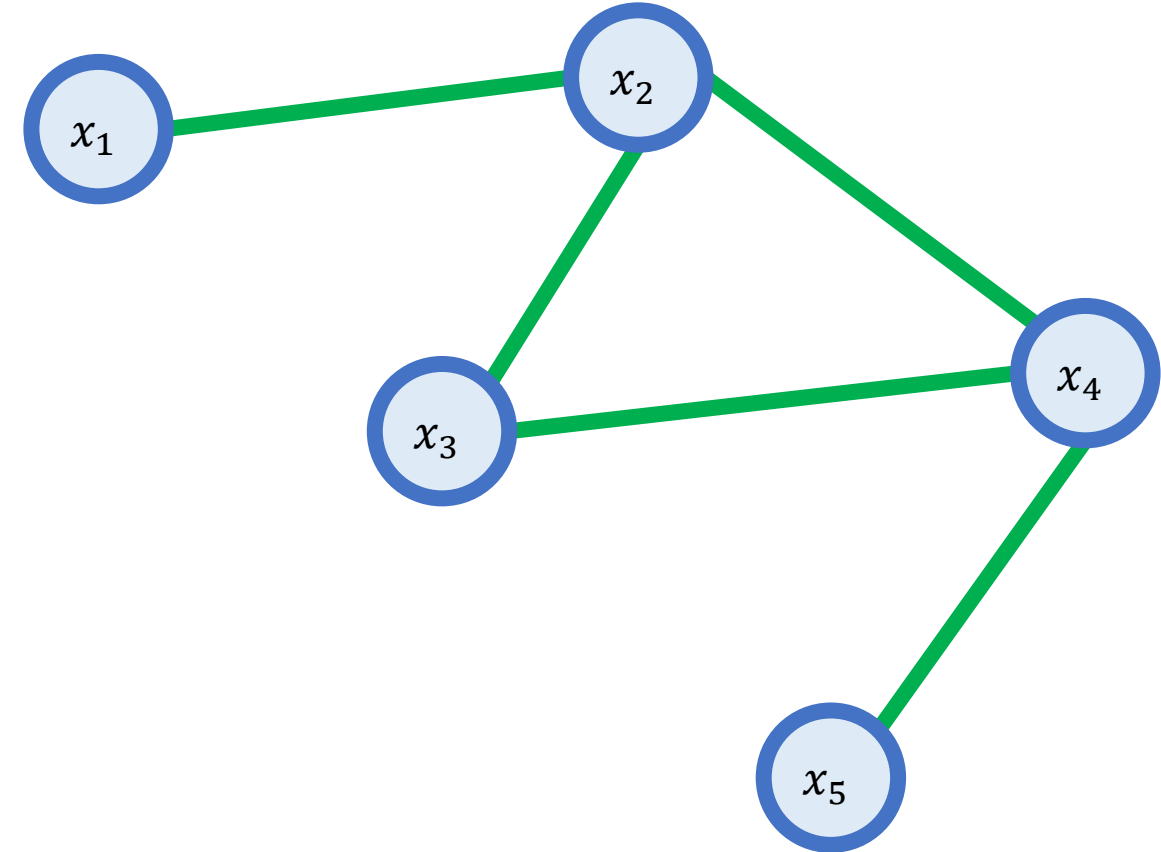
(Need a refresher? Check additional materials!)



Graphs: a general and minimal definition

Definition (graph): A **graph** is a mathematical object, defined by an ordered pair $G = (V, E)$, with

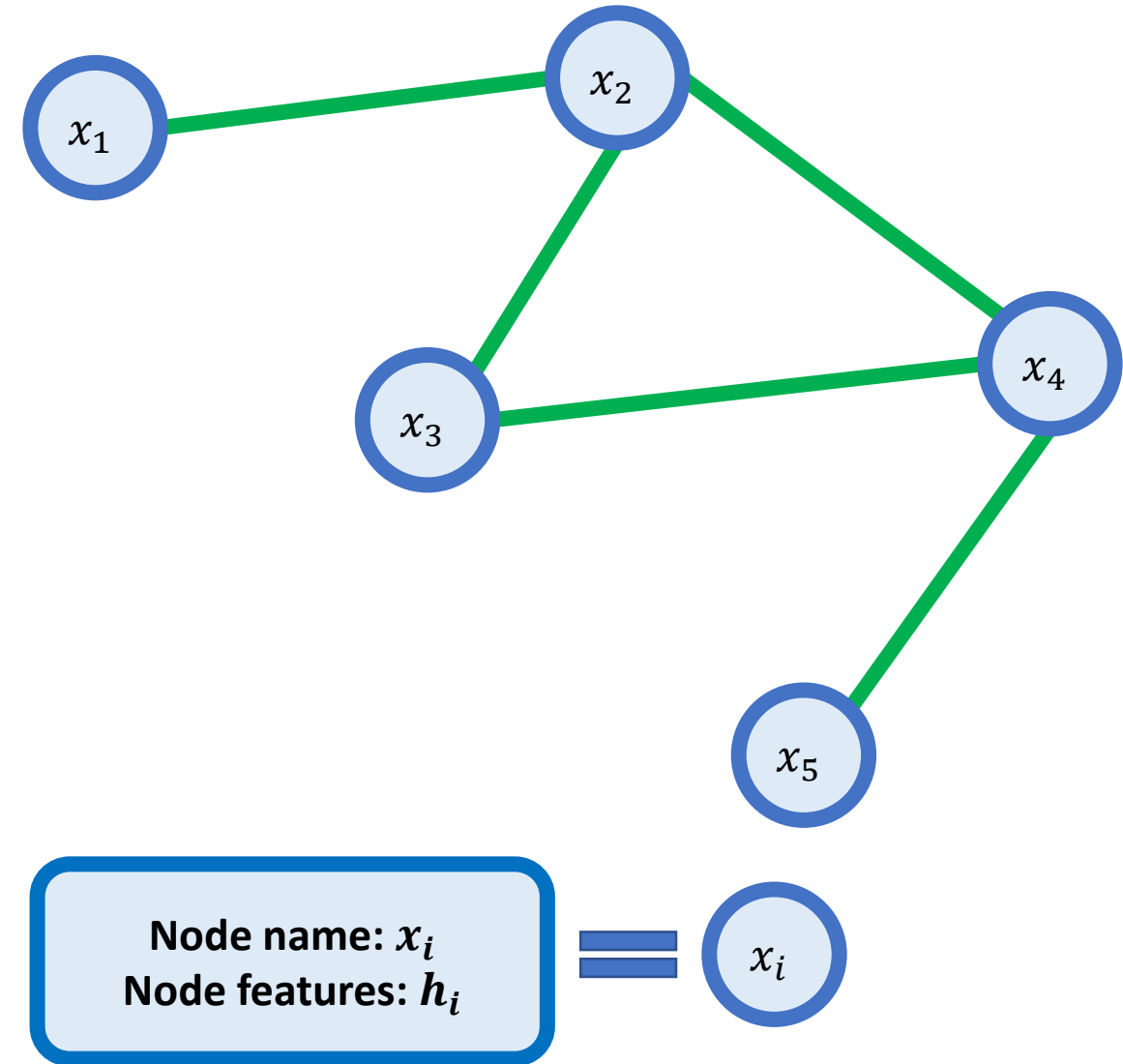
- $V = \{x_1, x_2, \dots, x_N\}$ a set of N **vertices** (also called **nodes** or **points**),
- And E a set of **edges** (also called **links** or **lines**), defined as a subset of $\{(i, j) \mid \forall i \in [1, N], \forall j \in [1, N]\}$.



Nodes definition and attributes

Definition (nodes): A **node** x_i is a point in the graph.

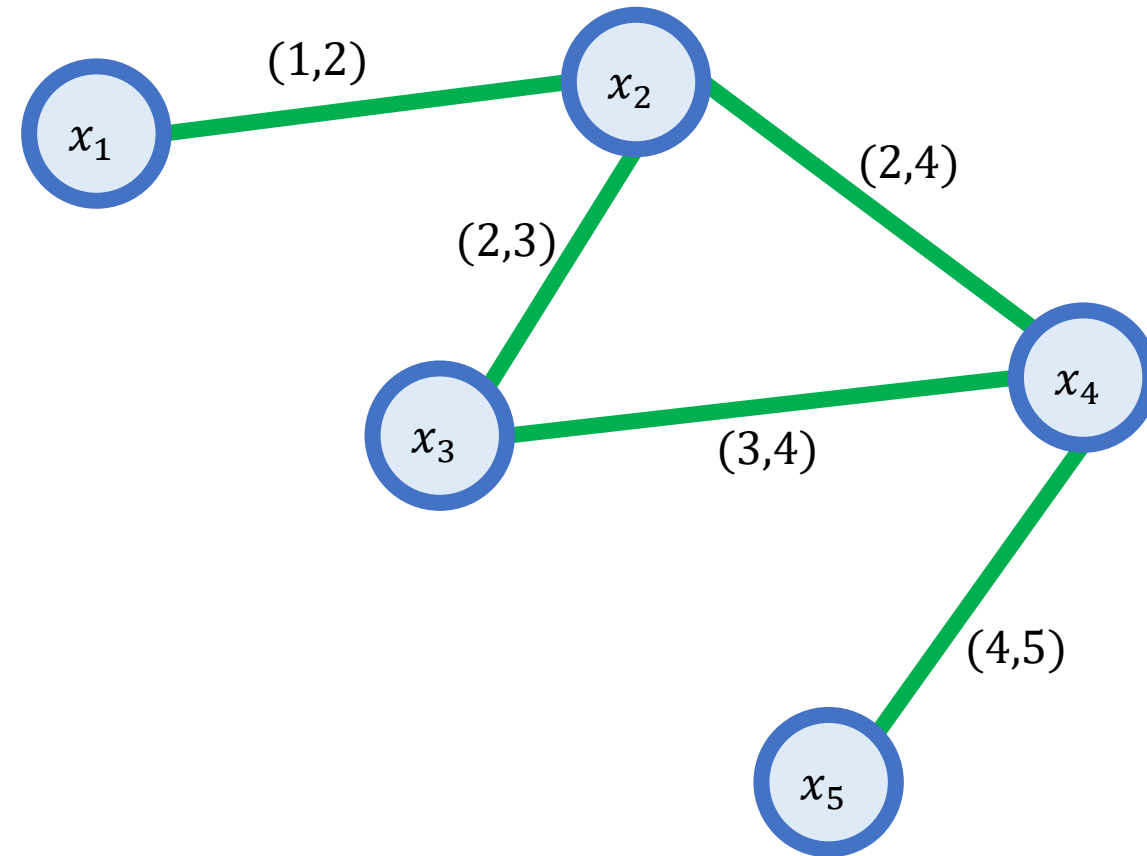
- A **node** has a **name** x_i , which is used for indexing and differs from one node to another.
- A node may also have **attributes**, or **node features**, defined, for instance, as a vector $\mathbf{h}_i \in \mathbb{R}^F$, with F elements



Edges definition

Definition (edges): An **edge** (i, j) defines a connection from **node** x_i to **node** x_j .

- If **edge** $(i, j) \in E$, then nodes x_i and x_j are connected in the **graph** G .
- In our example, we have
$$E = \{(1, 2), (2, 3), (2, 4), (3, 4), (4, 5)\}$$

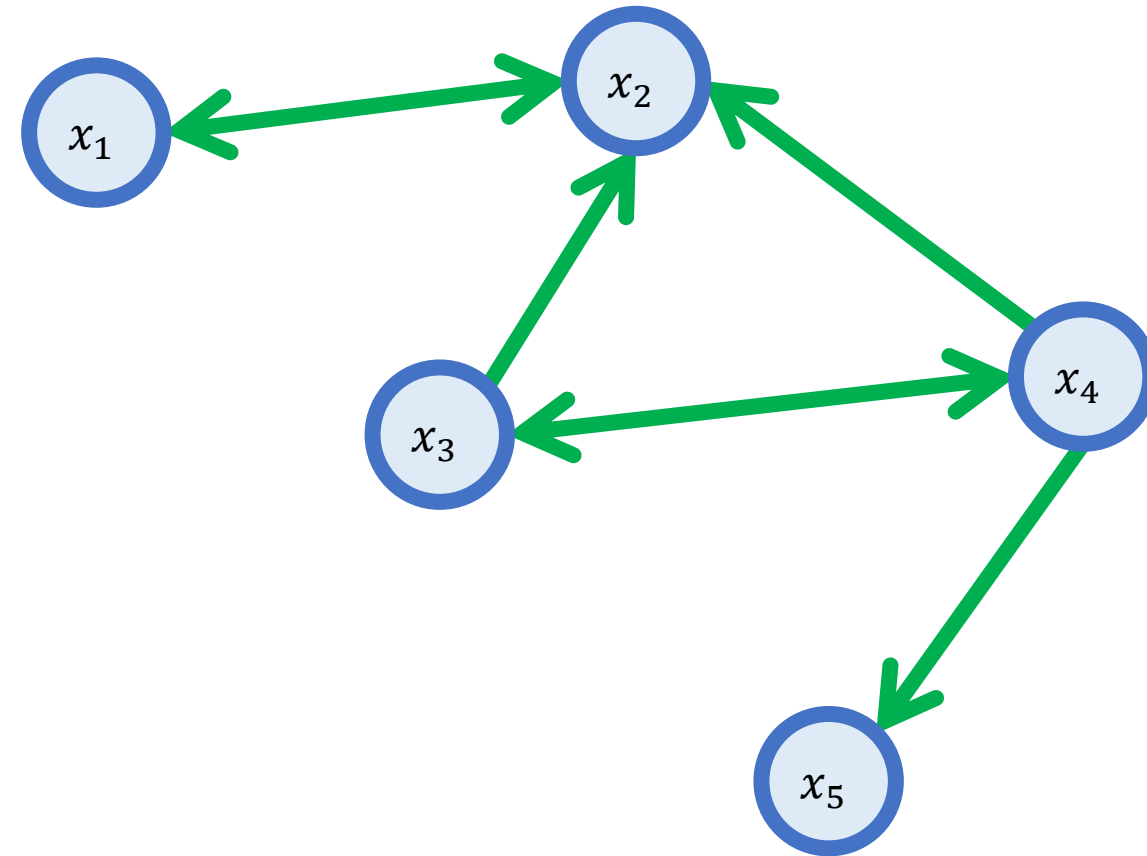


Directed graph definition

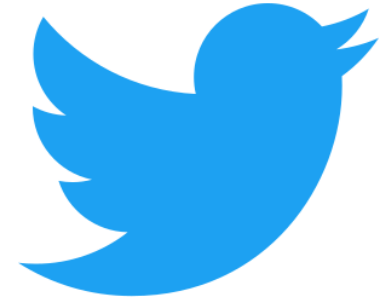
Definition (directed graph): A graph G is **directed**, if the undirected property below does not hold.

- **Undirected property:** “if node x_i is connected to node x_j , then node x_j is also connected to node x_i ”.
- In that case, draw arrows for edges.
- Our example is a directed graph, and our edges set writes as

$$E = \{(1, 2), (2, 1), (3, 2), (4, 2), (3, 4), (4, 3), (4, 5)\}$$



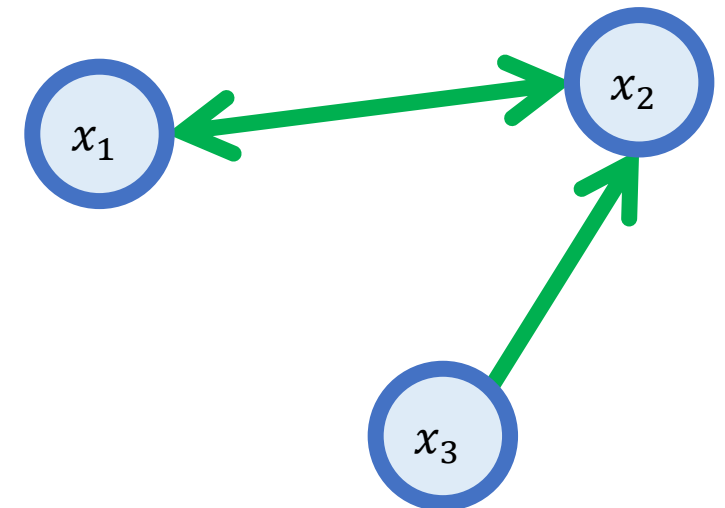
Examples of a directed graph



- An examples of a directed graph is... **Twitter/Instagram!**
- As before, a **node x_i** consists of a Twitter user, and its features are user data.
- On Twitter, the **undirected property** does not hold: you can follow people, but they do not have to follow you back.

Node name: x_i = User ID

Node features: h_i = (user first name, user family name, date of birth, age, etc.)



Back to our 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**.

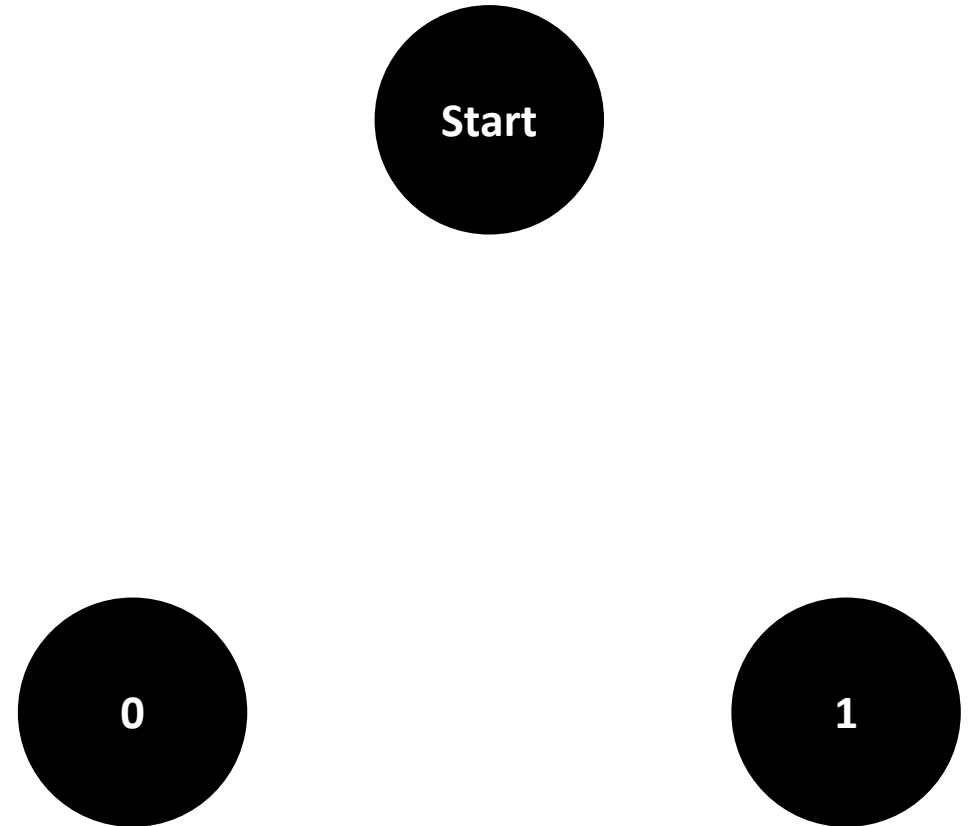
An FSM can be represented using a **graph** representation, known as a **state diagram**, which shows the possible states of the system and the transitions between them.

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.**

An example of an FSM

Consider the **FSM** and its **state diagram** on the right, represented as a **directed graph**.

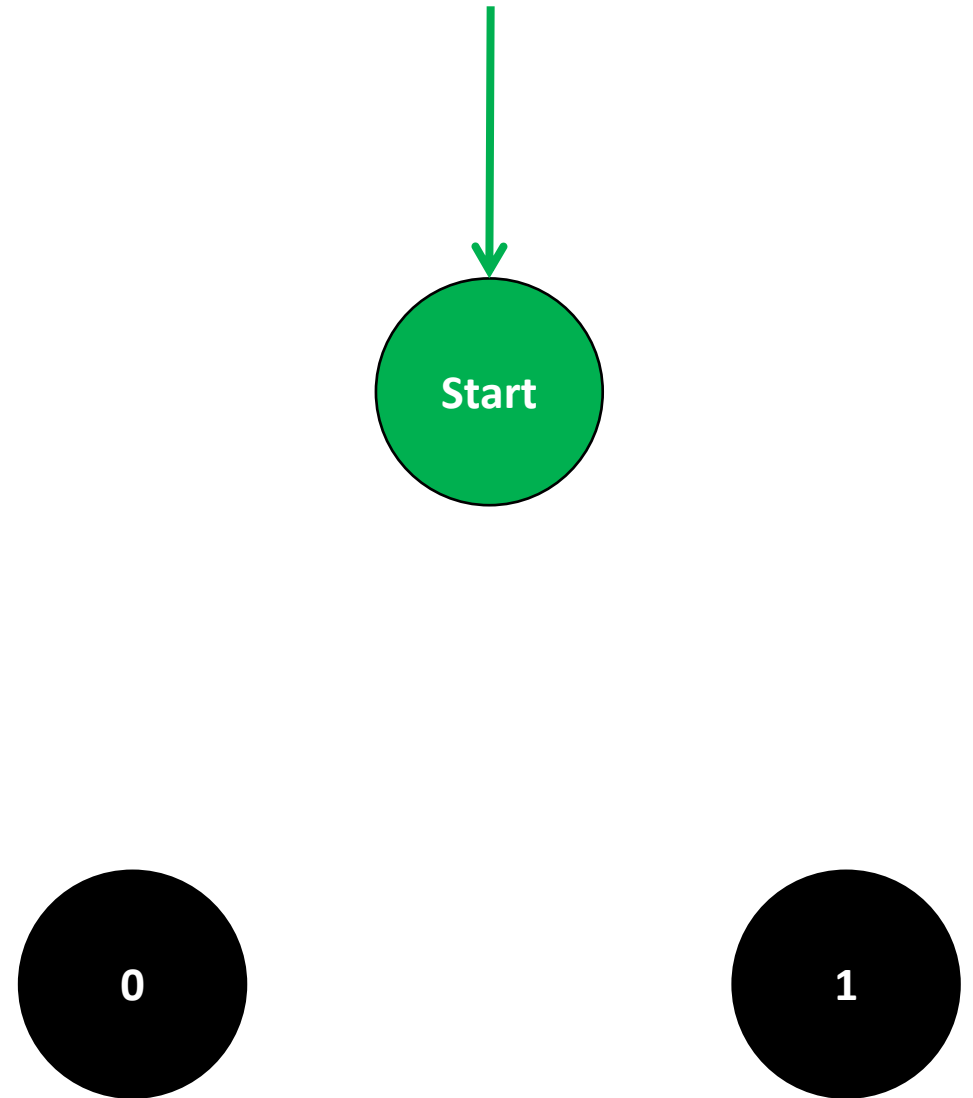
- Given a **string**, e.g. $x = "00110"$,
- We will check all elements in the string, one at a time, from left to right, i.e.:
 $"0" \rightarrow "0" \rightarrow "1" \rightarrow "1" \rightarrow "0"$.
- These five consecutive values will serve as **states** and **inputs**.



An example of an FSM

Consider the **FSM** and its **state diagram** on the right, represented as a **directed graph**.

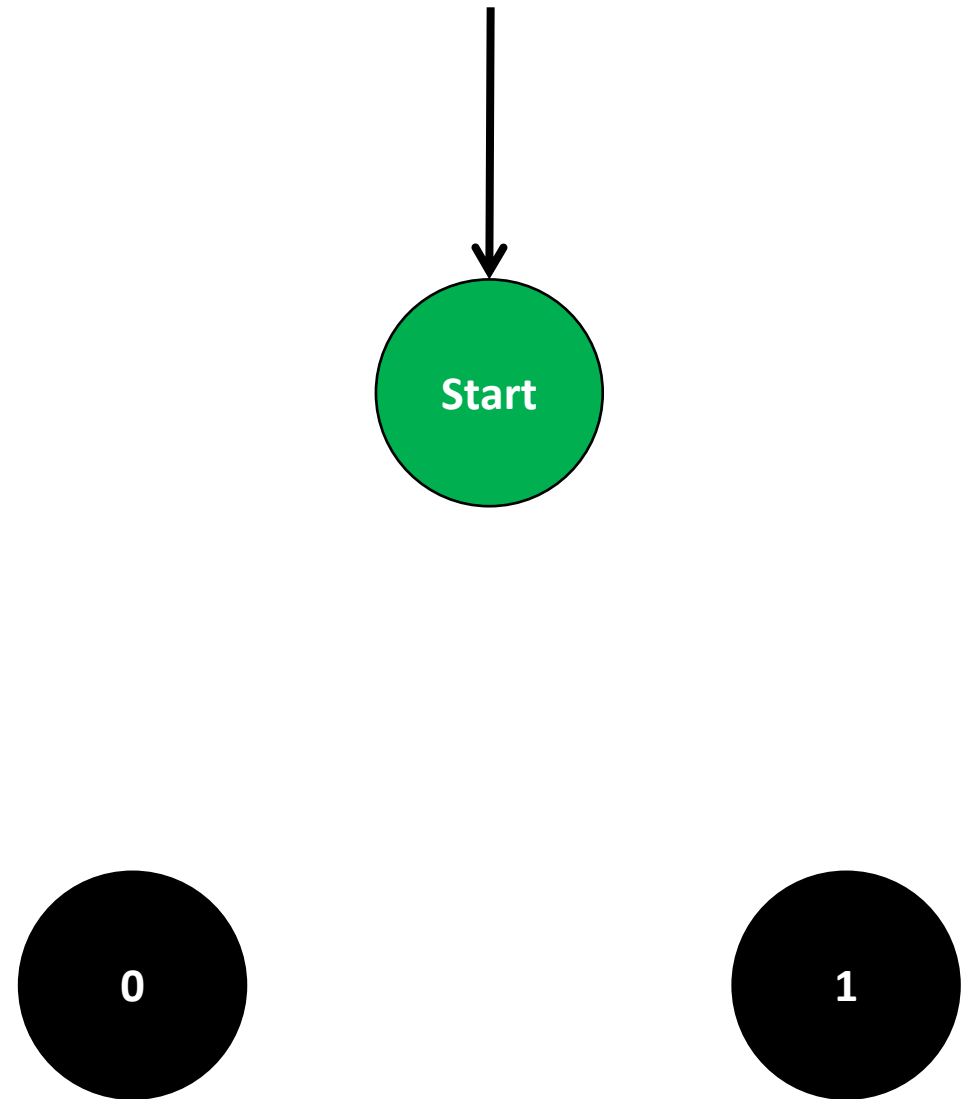
- Begin at **start node**, which is defined as a node, receiving an incoming arrow with no source.



An example of an FSM

Consider the **FSM** and its **state diagram** on the right, represented as a **directed graph**.

- We will check all elements in the string x , one at a time, from left to right, i.e.:
 “0” → “0” → “1” → “1” → “0”.
- We start with “0”, and we are currently on the **start node**.



An example of an FSM

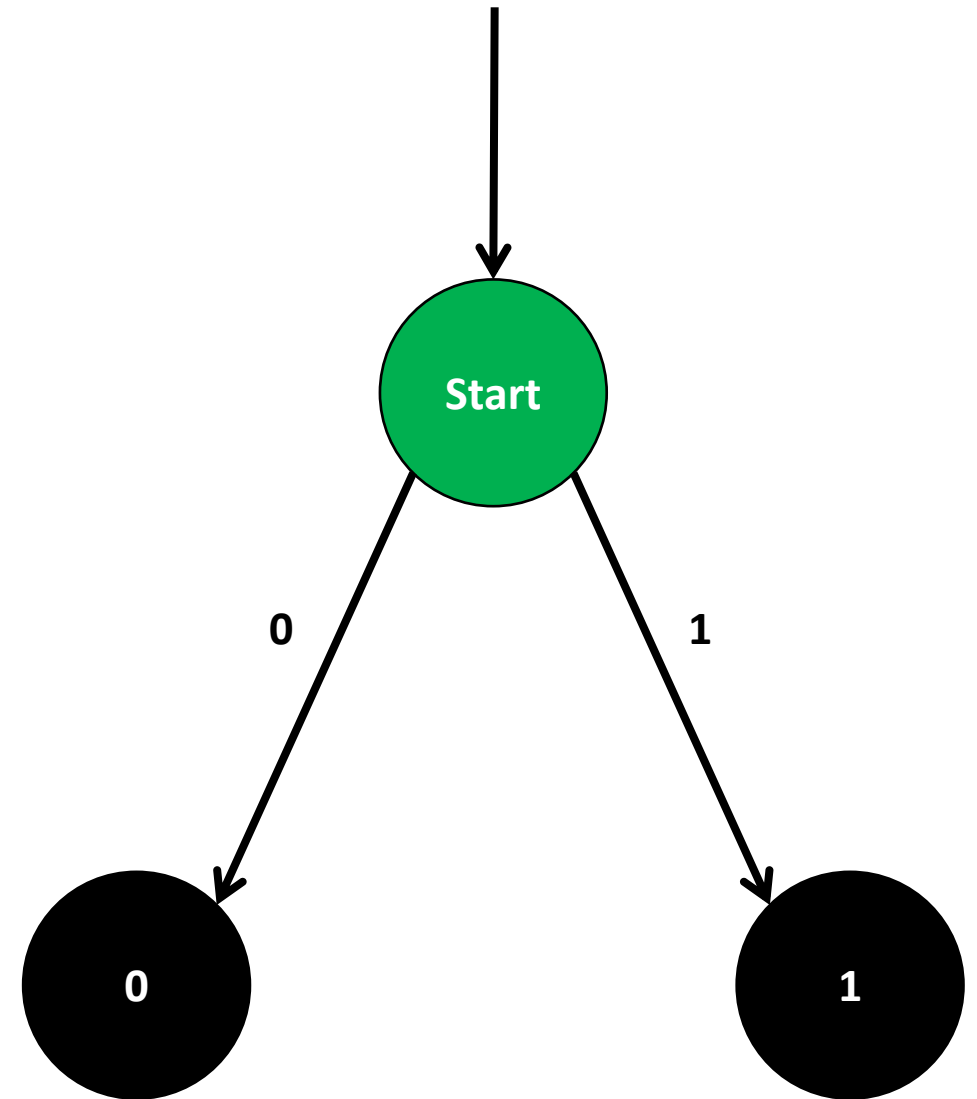
And then, move in the FSM state diagram, following a **logic**.

- We first consider that this value “0” serves as **input** (or **action**) for this turn.
- **Logic**, defined by the **directed graph**, for this FSM (could differ from one FSM to another):

*“On every **input**, your **next state** will be matching your **input**.*

See 0 as input, move to state 0.

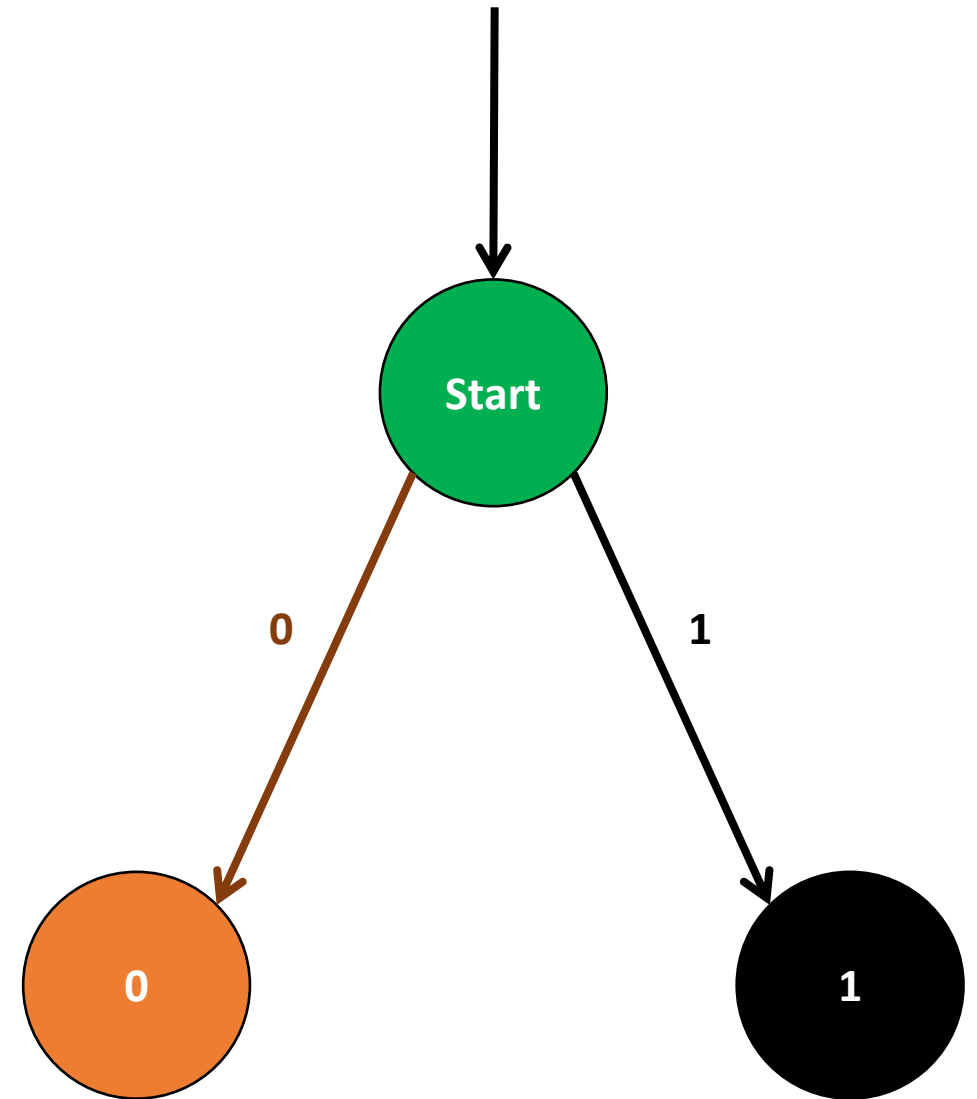
And vice-versa.”



An example of an FSM

For instance,

- Following this logic, our first turn sees the **input value** “0”.
- Moves from **our current state**, the **start node**,
- To the **next state**, i.e. **node “0”**,
- Following the **directed edge**, starting from the **start node** and labelled with “0”.

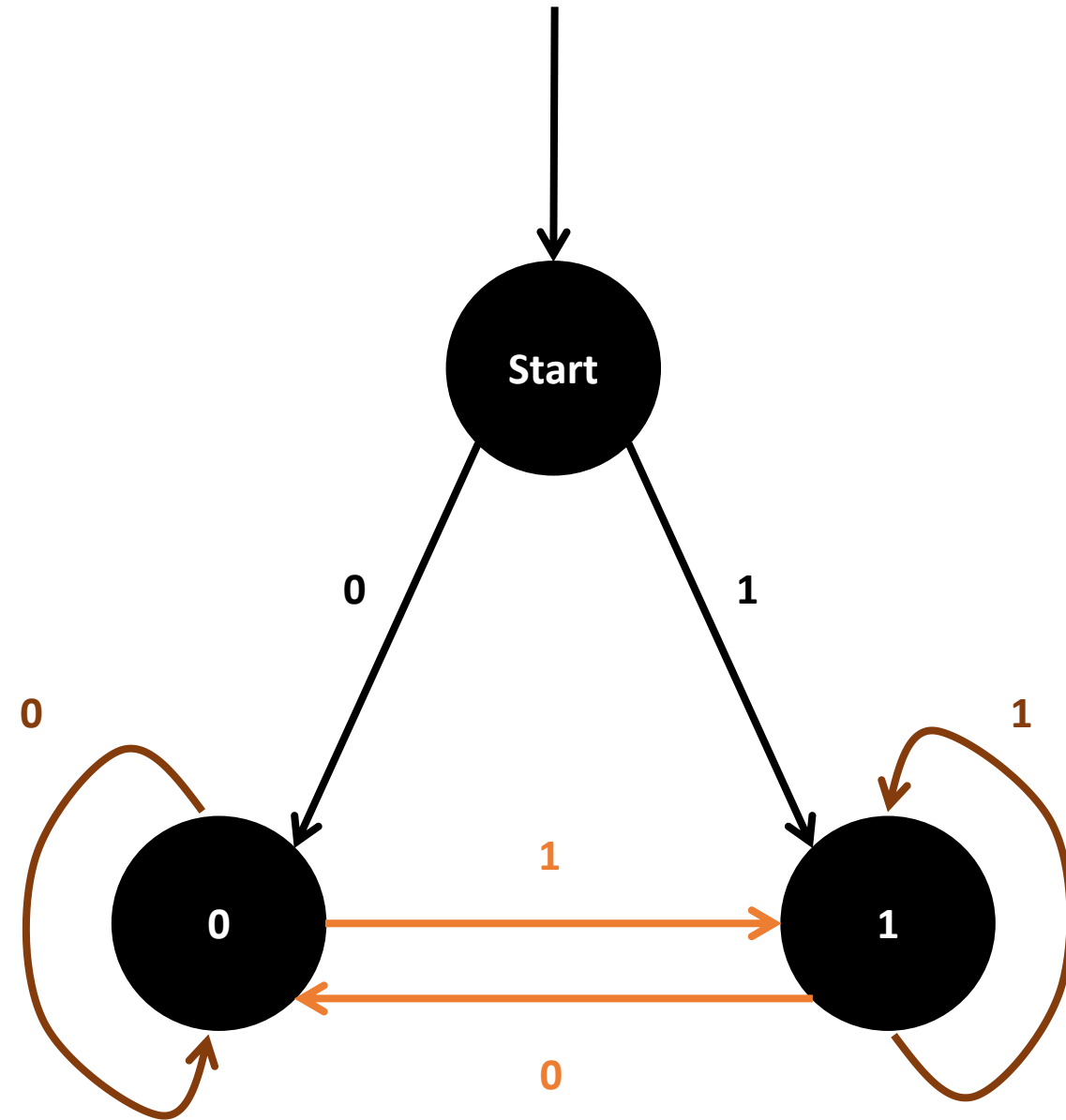


An example of an FSM

And then, we repeat this logic for each element in our string x ,

$"0" \rightarrow "0" \rightarrow "1" \rightarrow "1" \rightarrow "0"$.

- To do so, we will need to draw **additional links**.
- In general, we want to have exactly two links departing from each state node, one for each possible action.
- Some of these links **might point to themselves!**



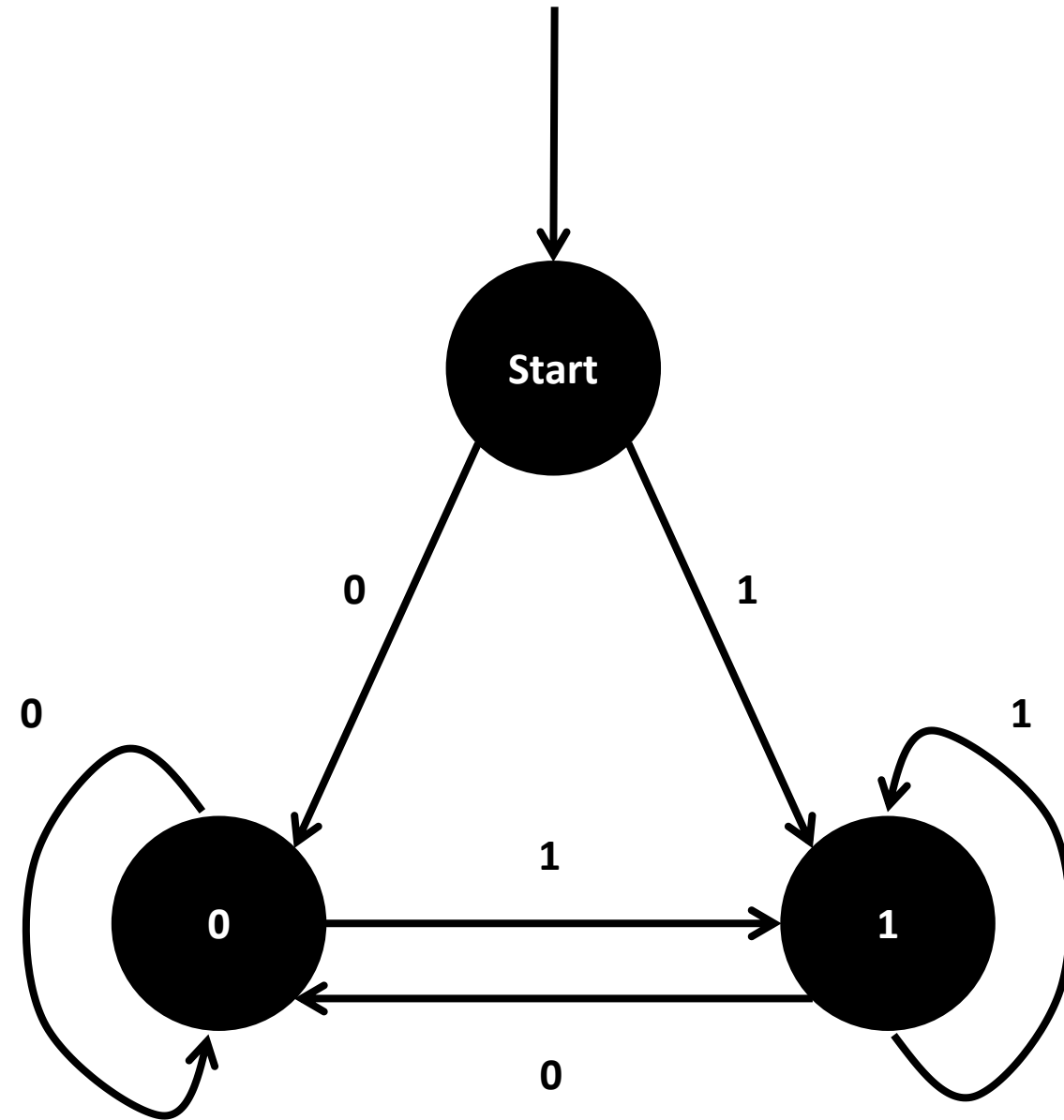
Elements of an FSM

Based on the FSM definition from earlier, we understand that several elements must be defined.

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

In our example, we have

$$S = \{start, 0, 1\}$$



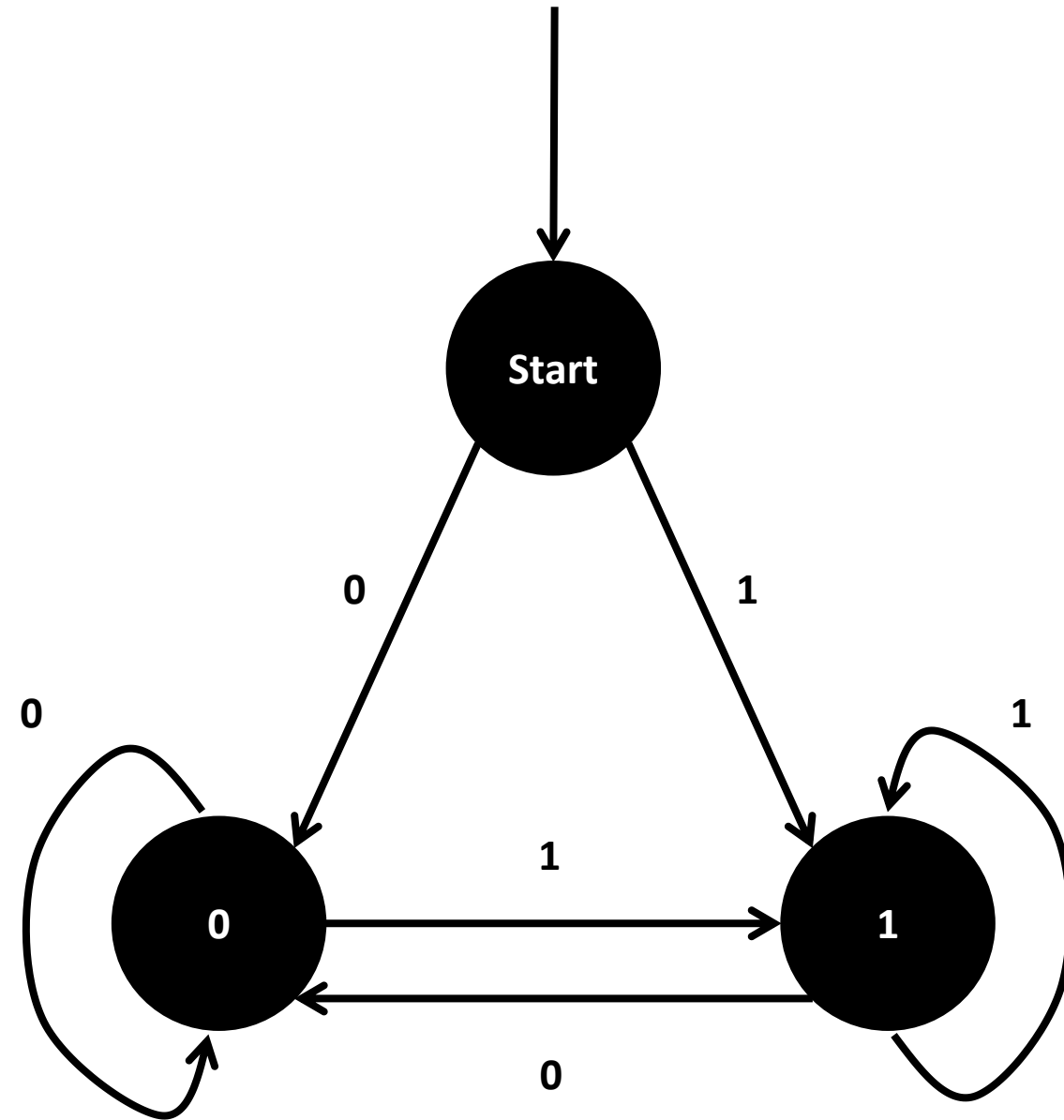
Elements of an FSM

Based on the FSM definition from earlier, we understand that several elements must be defined.

2. A finite set of **inputs** or **actions** A . Note that it does not have to use the same notations as S !

In our example, we have

$$A = \{0, 1\}$$



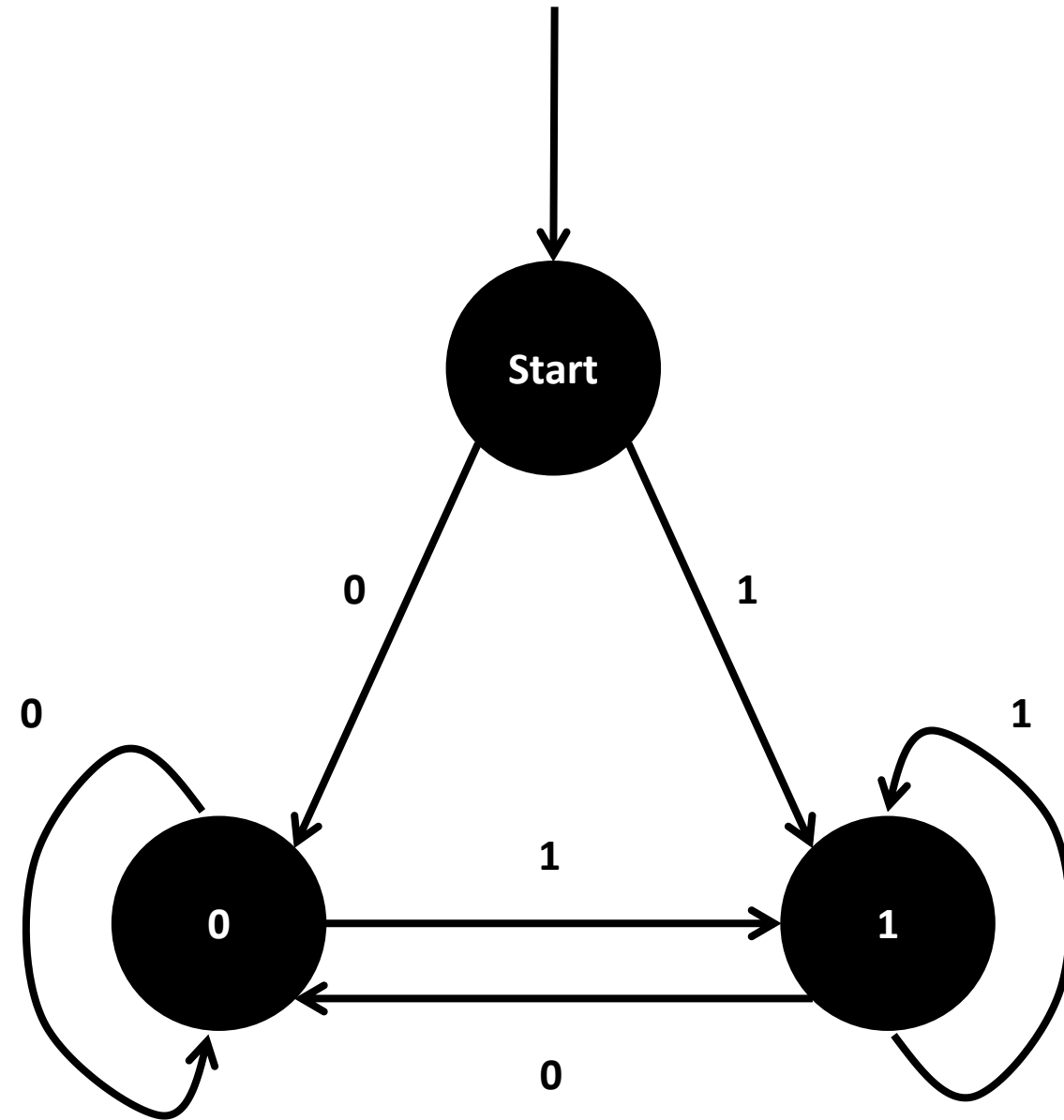
Elements of an FSM

Based on the FSM definition from earlier, we understand that several elements must be defined.

3. A **starting state** $s_0 \in S$.

In our example, we have

$$s_0 = \text{"start"}$$



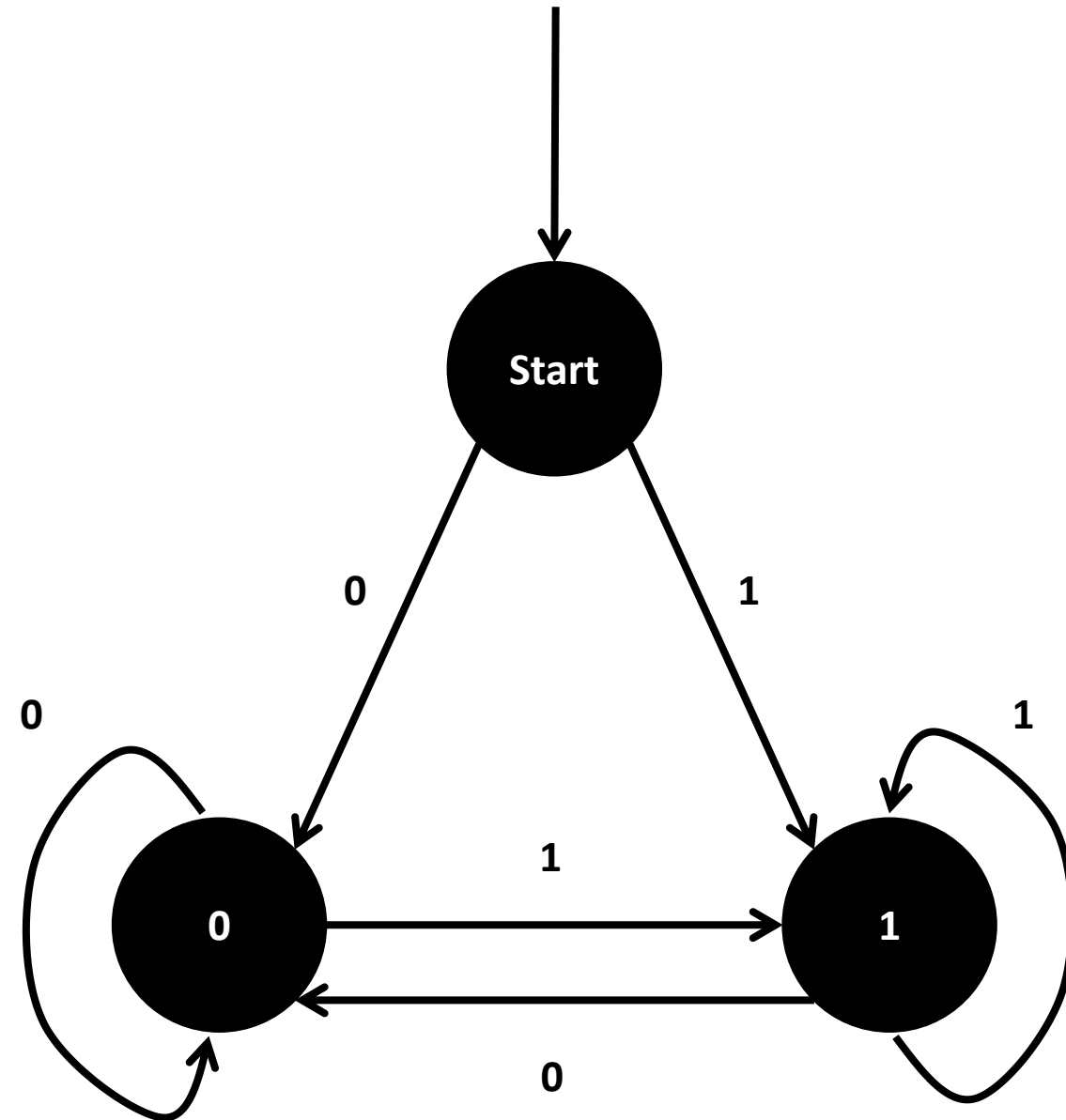
Elements of an FSM

Based on the FSM definition from earlier, we understand that several elements must be defined.

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

This function takes a **current state** $s \in S$, and a **current action** $a \in A$ as parameters. It returns a **new state** $s' \in S$ as output.

$$f: (s, a) \rightarrow s'$$



Elements of an FSM

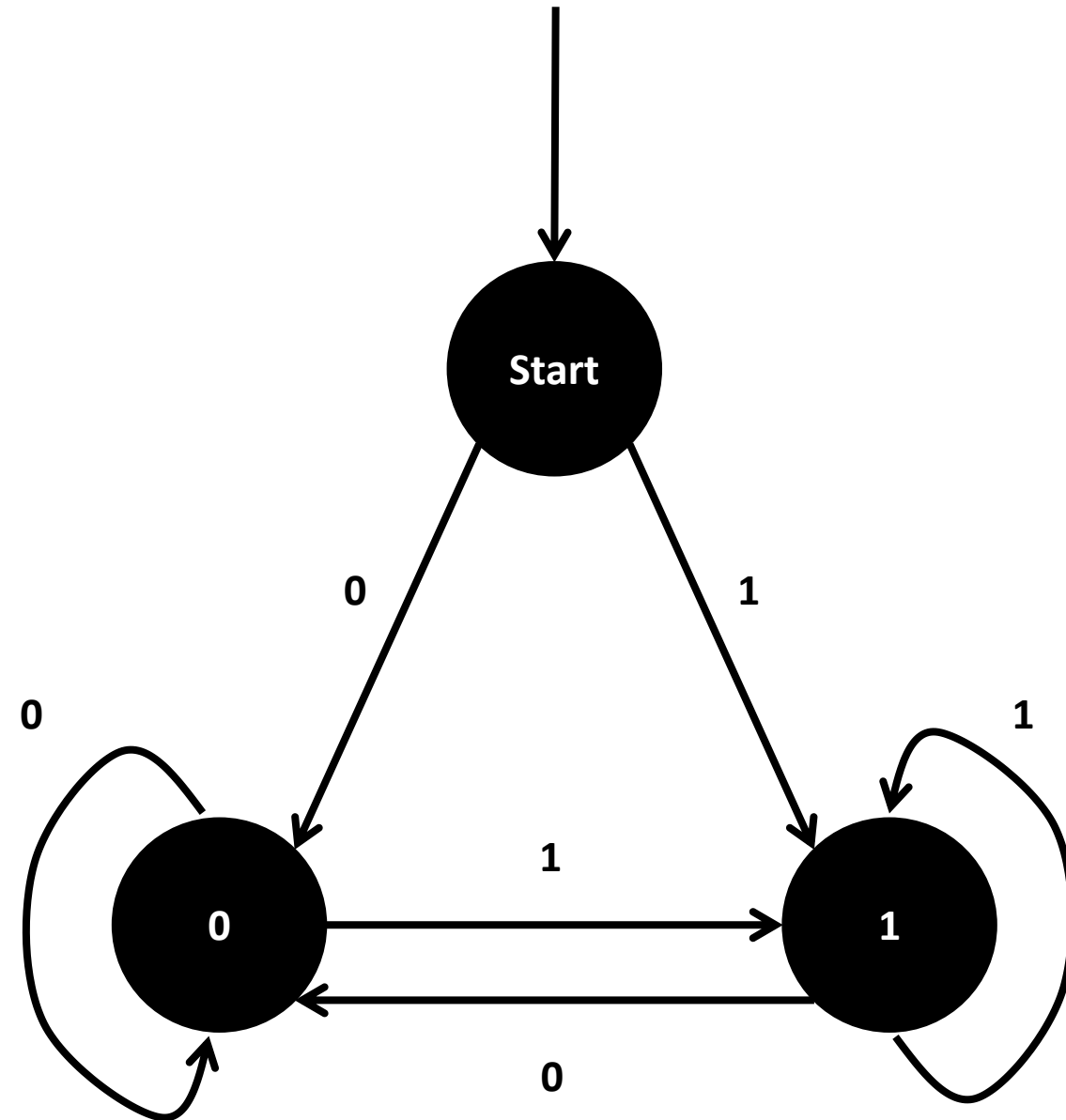
Based on the FSM definition from earlier, we understand that several elements must be defined.

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

In our case, the function is simply:

$$f: S \times A \rightarrow S$$
$$f(s, a) = s' = a$$

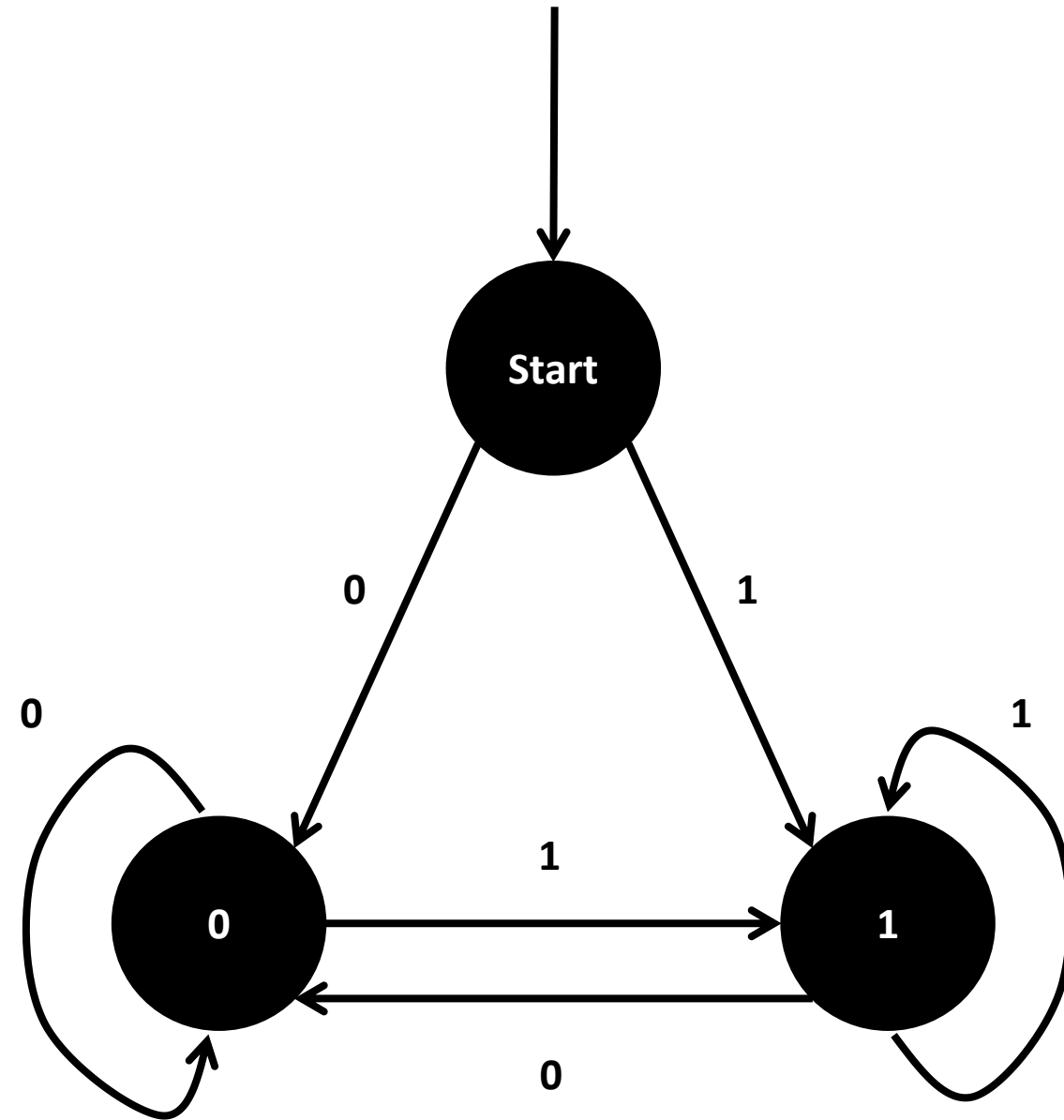
(Other logics for FSMs can be implemented)



Elements of an FSM

Note: the **transition function** can also be expressed in the form of a **transition table**, as shown below.

Current state	Input	Next state
Start	0	0
Start	1	1
0	0	0
0	1	1
1	0	0
1	1	1

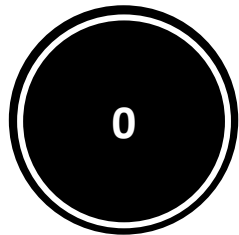


Adding a stopping state

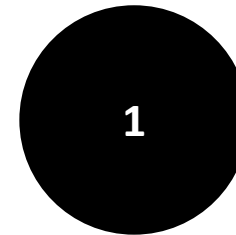
Definition (**stopping states** or **accepting states**):

In a finite state machine (FSM), a **stopping state** or **accepting state** is a **special type of state node** in which the FSM can terminate.

Stopping states can be represented in the state transition diagram by drawing a double circle around the state.



0 is an accepting state



1 is not an accepting state

Adding a stopping state

Definition (**acceptable input**):

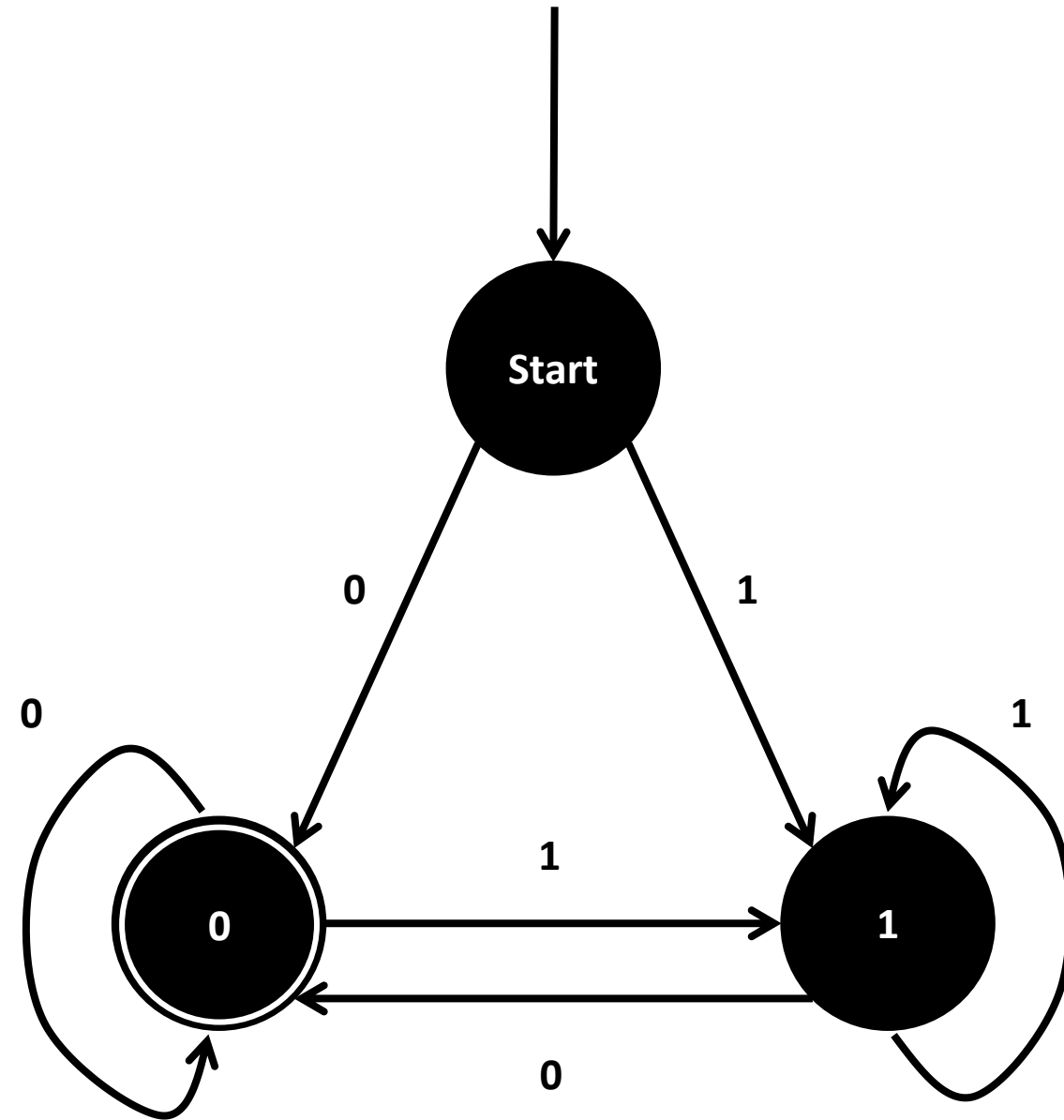
In general, we will consider that the inputs string x (e.g. our “00110” from earlier) is an **acceptable input**, if and only if the **final state** after processing the entirety of the input string s with the FSM happens to be a **stopping state** or **accepting state**.

Important: Passing through an accepting state and ending on a non-accepting one, means your input string s is not acceptable!

Adding a stopping state

Stopping states can be represented in the state transition diagram by drawing a double circle around the state, or by adding a label to the state to indicate that it is a stopping state.

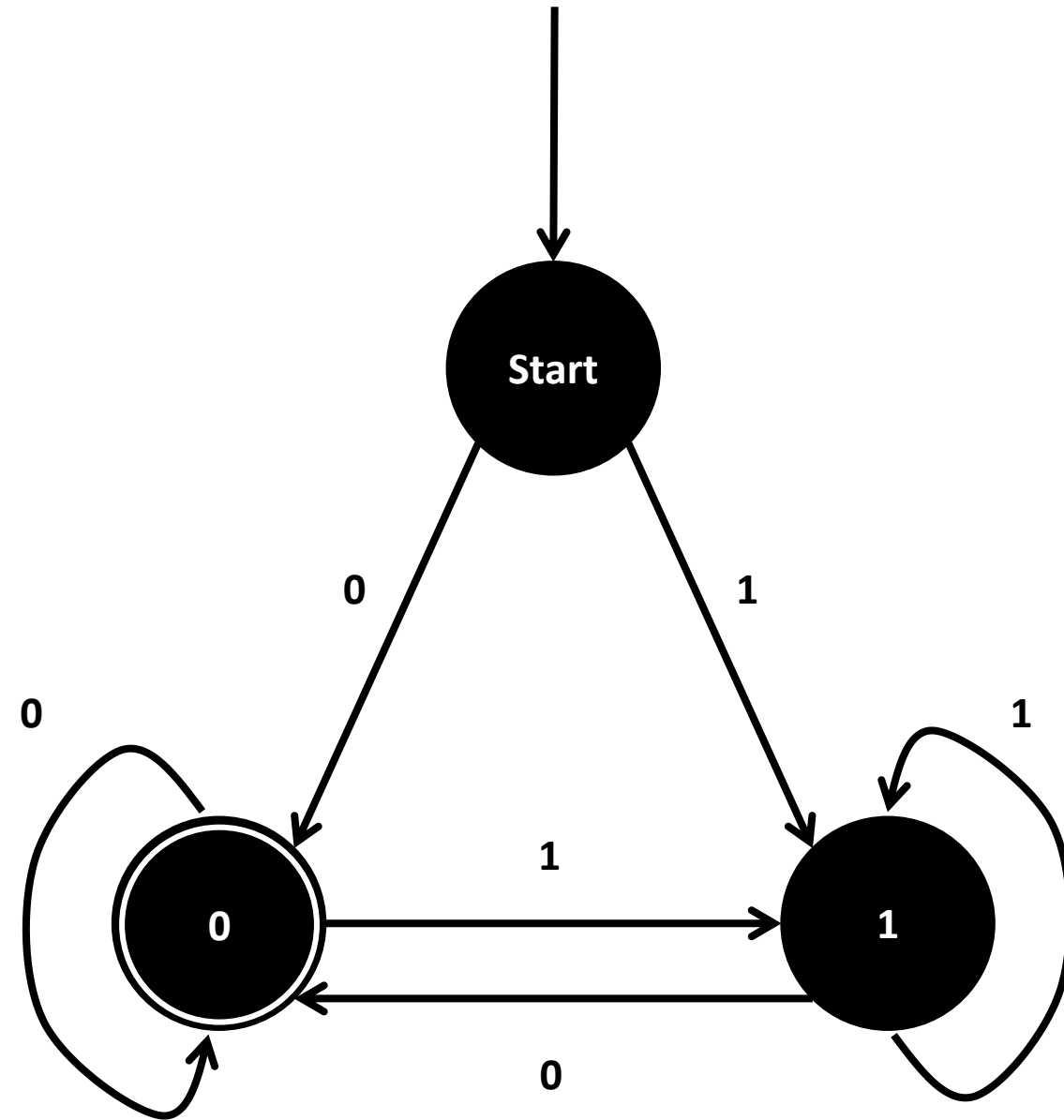
In our example, **the state 0 is now a stopping state**, but 1 and Start are not.



Adding a stopping state

As we will see later through practice, an FSM with stopping states can be used to perform specific tasks, such as recognizing patterns or processing data.

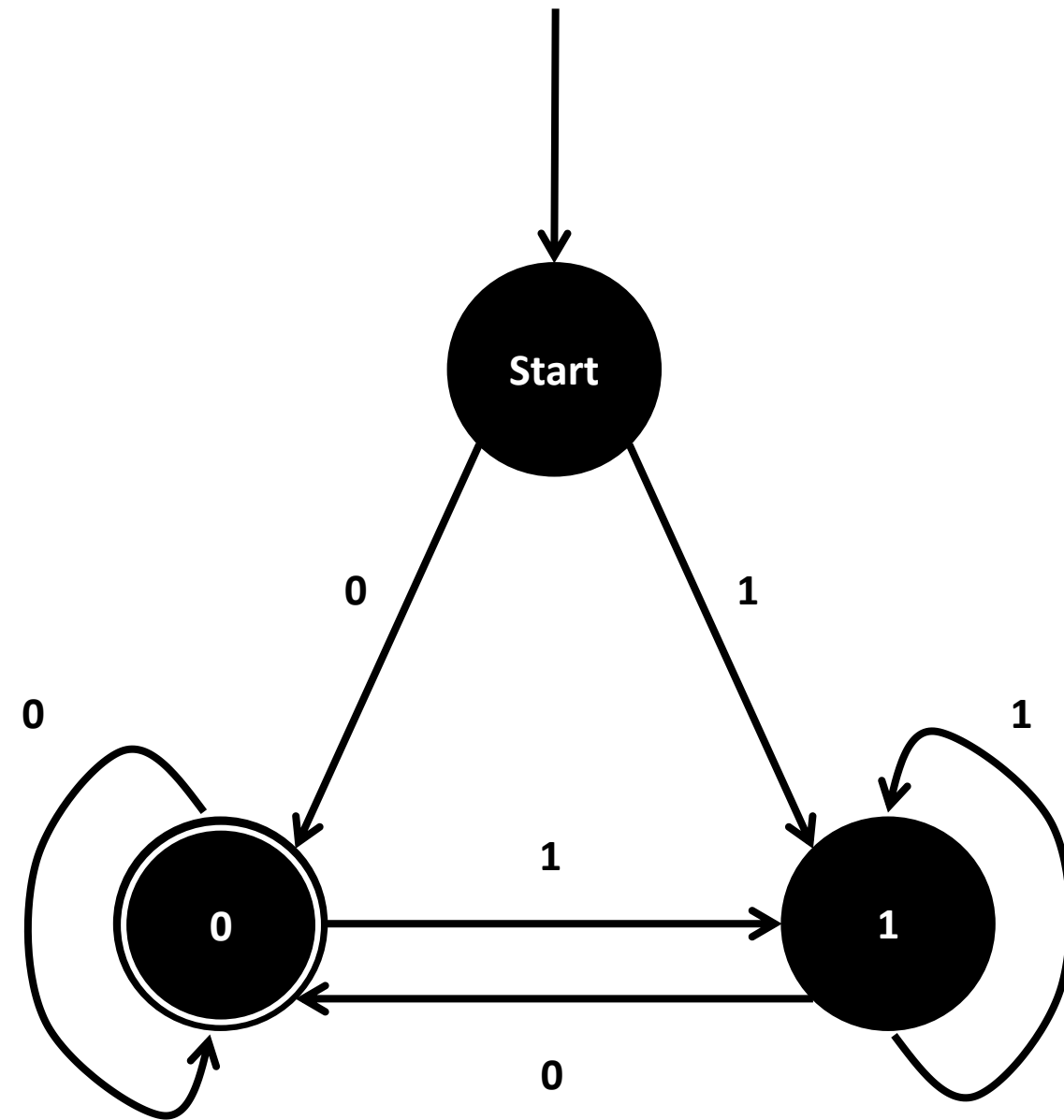
Typically, in a language recognizer FSM, the stopping state will be used to indicate that the FSM has recognized a certain substring in the input string x .



Elements of an FSM with stopping states

In order to define a FSM with stopping states, 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.



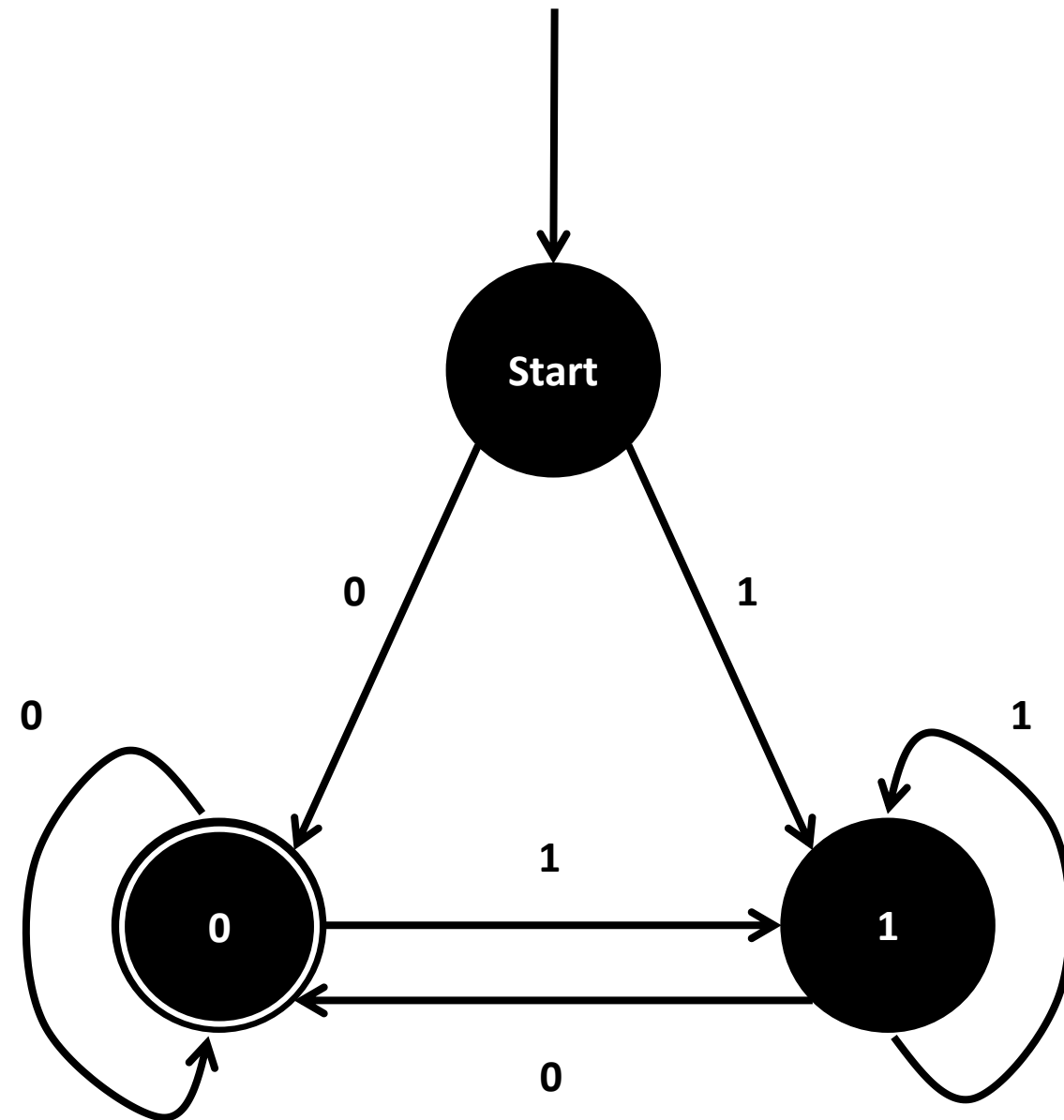
Elements of an FSM with stopping states

In order to define a FSM with stopping states, 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

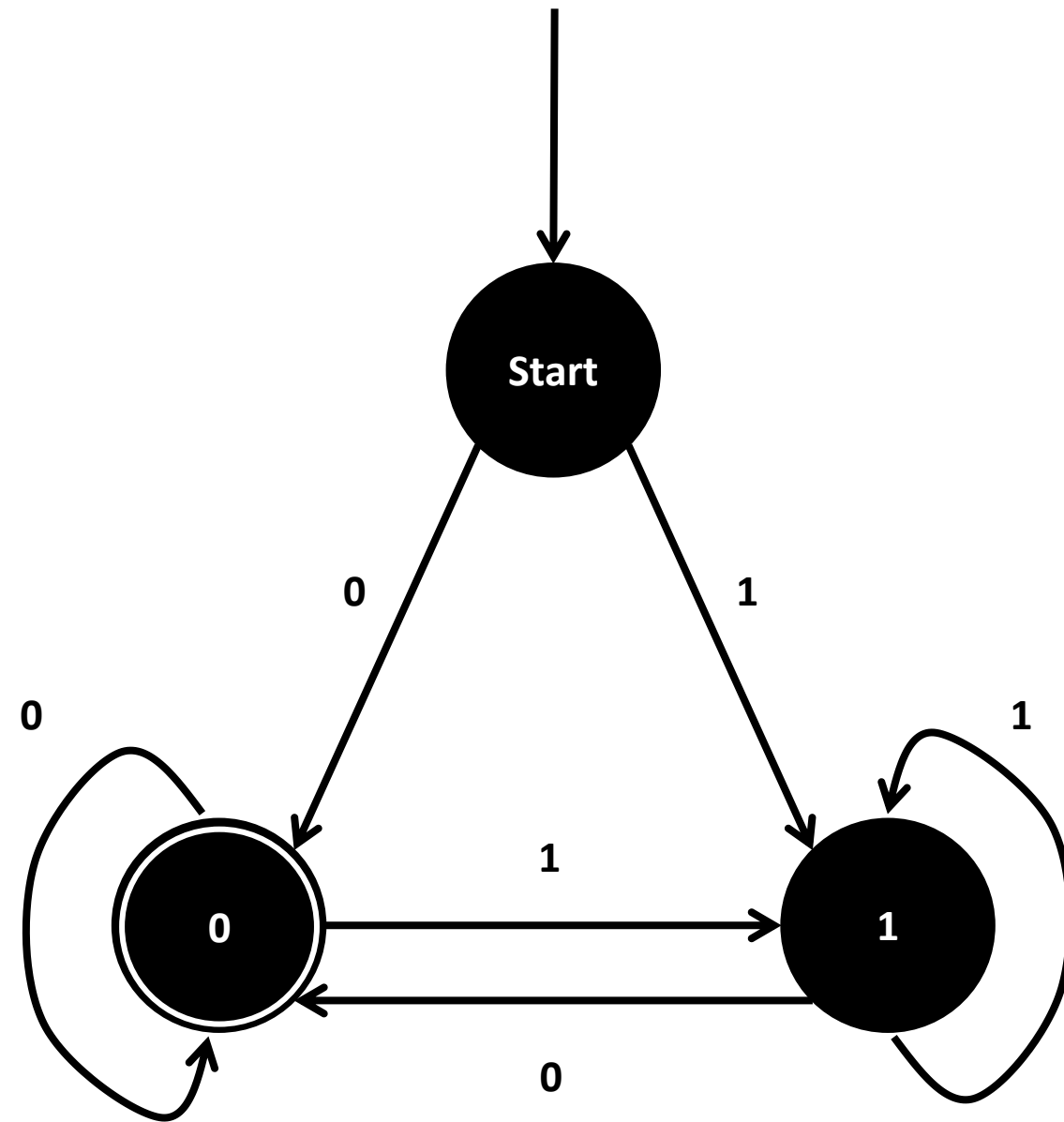
$$F = \{0\}$$



Practice 1: Analysing our example FSM

Question 1: Looking at the FSM state diagram on the right, what are the **acceptable inputs x** that our FSM is checking?

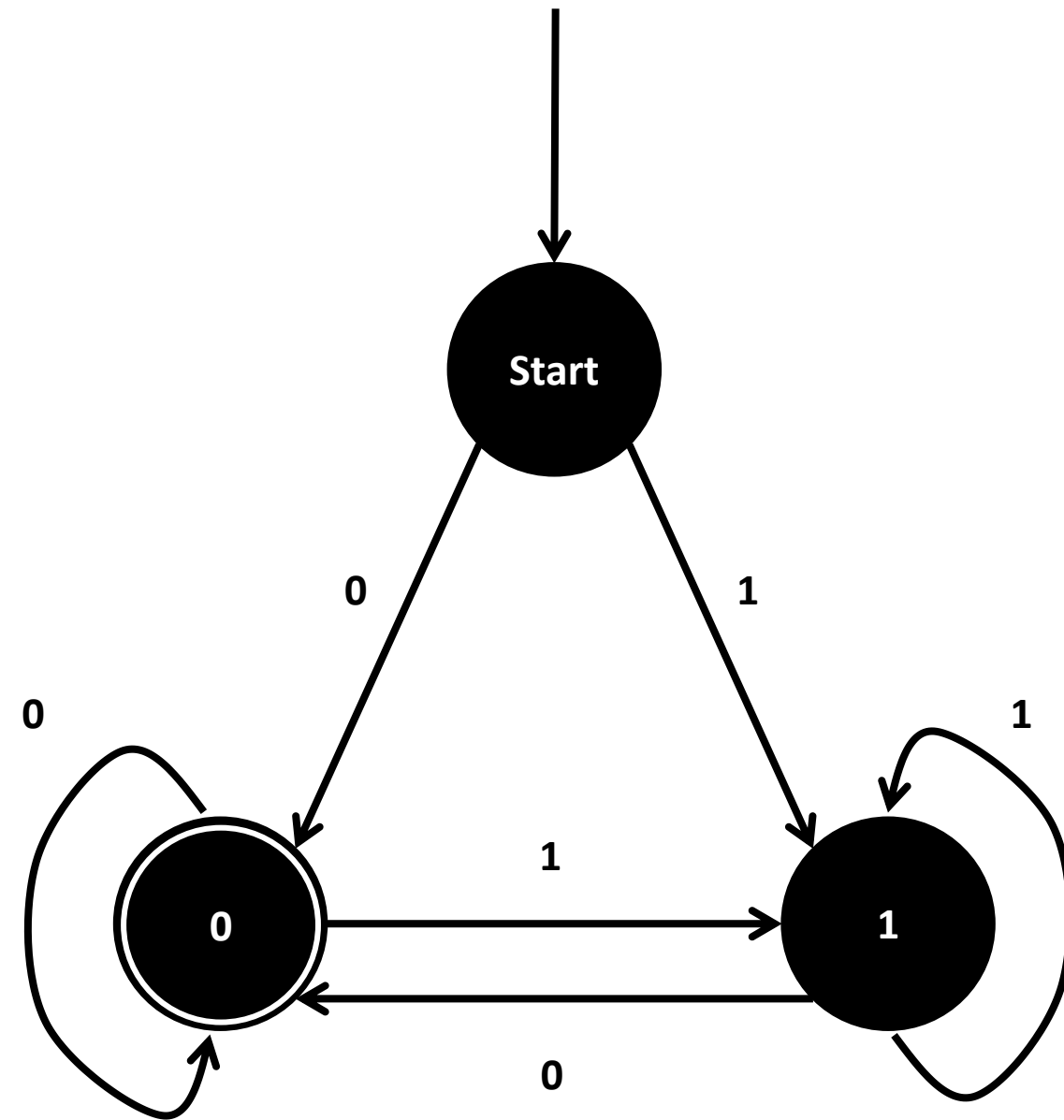
Question 2: If the input strings are binary strings encoding an unsigned integer number, what **numbers** does this FSM consider as **acceptable inputs**?



Practice 1: Analysing our example FSM

Question 1: Looking at the FSM state diagram on the right, what are the **acceptable inputs x** that our FSM is checking?

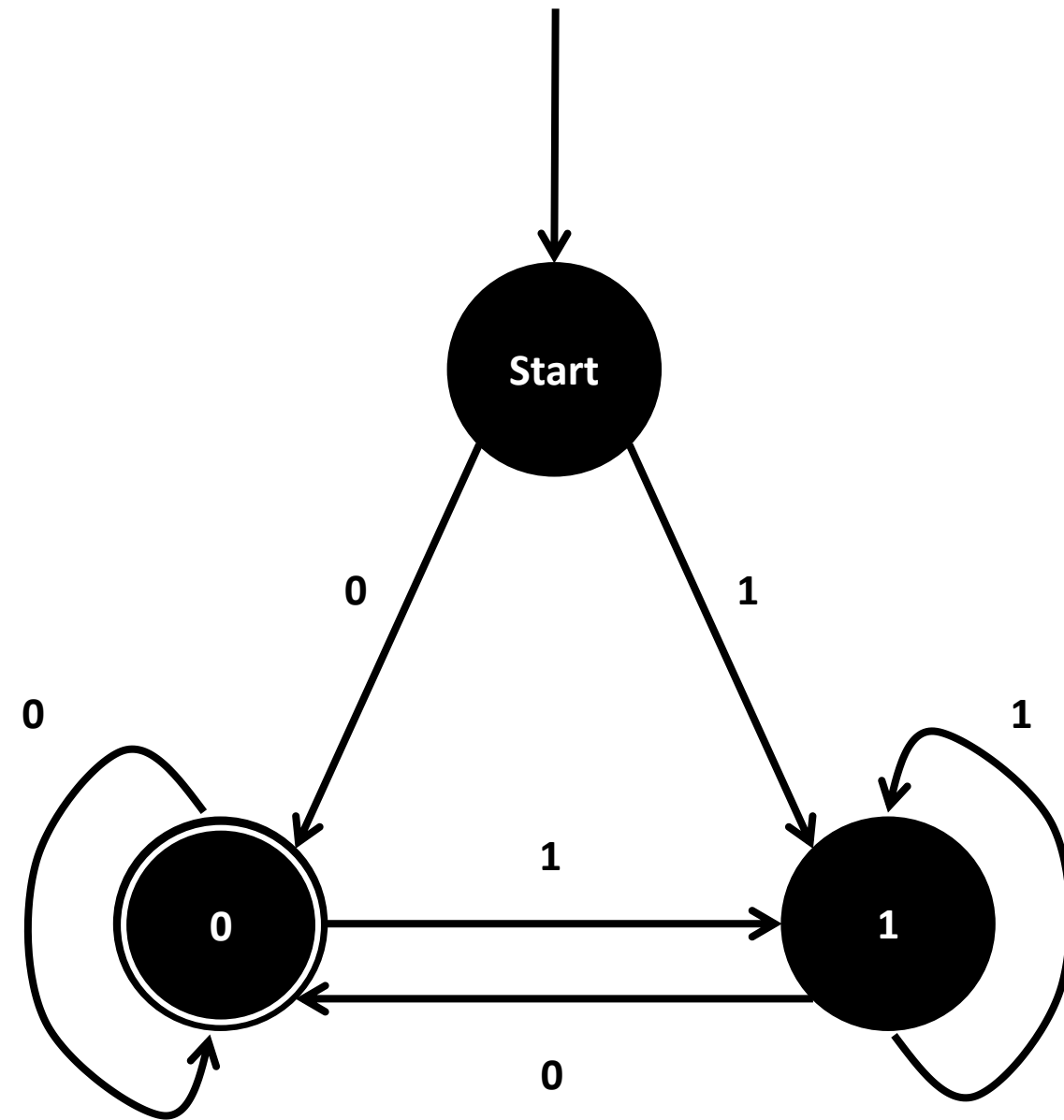
Answer: We need to end on a final state being 0. In our case, this means that an acceptable string s simply consists of any binary string that ends with 0.



Practice 1: Analysing our example FSM

Question 2: If the input strings are binary strings encoding an unsigned integer number, what **numbers** does this FSM consider as **acceptable inputs**?

Answer: We need the bit of least importance to be 0. This means that the integer number input represented by the string s needs to be even to be acceptable.



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:

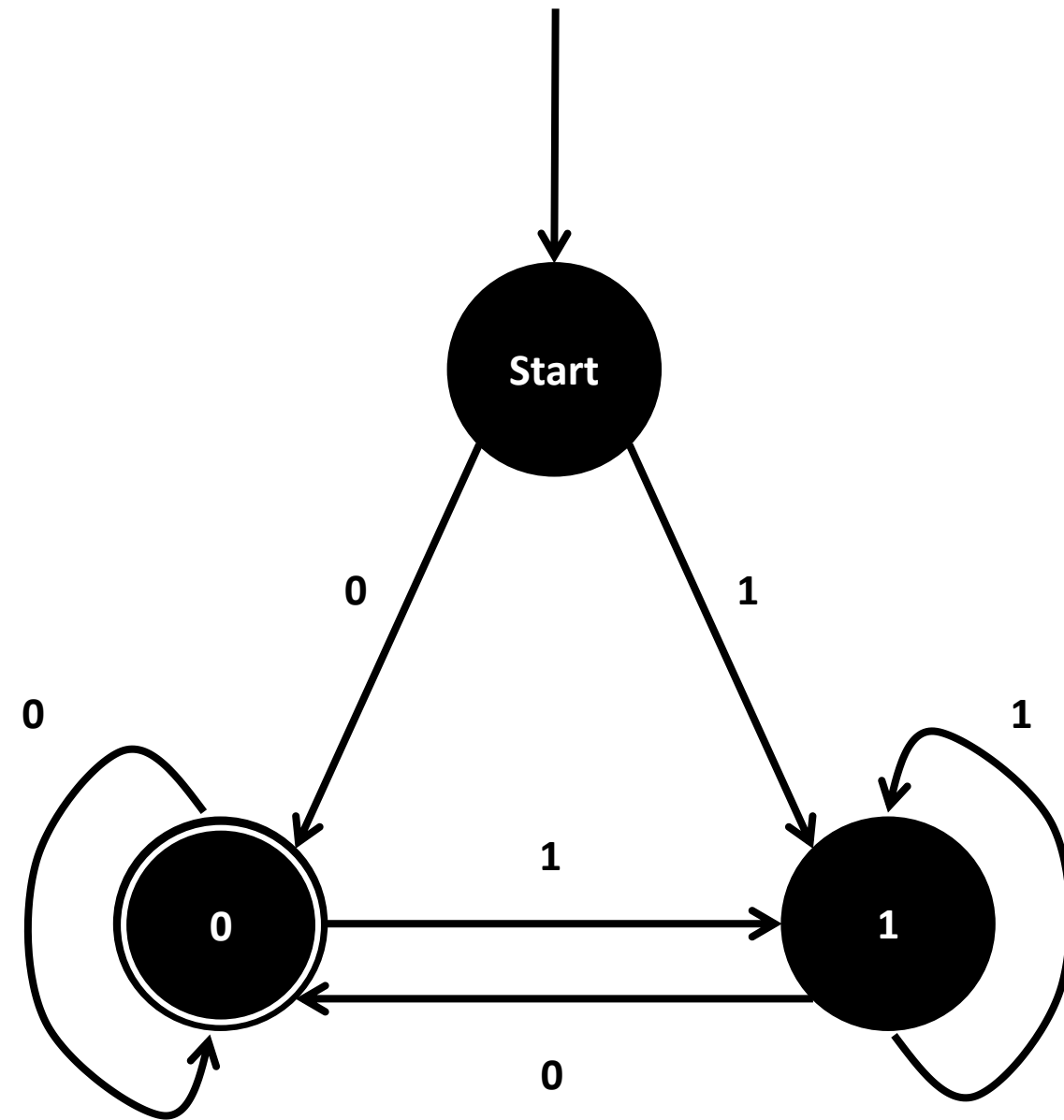
- Has 6 possible States (Start, S, U, T, D, Invalid),
- Has 4 possible Actions (S, U, T, D),
- Has the Start state defined as the starting state,
- Has the D state defined as the only stopping state,
- Has the FSM stop in this state D, if and only x is exactly “SUTD”; otherwise, it stops in another state (Invalid or something else).

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**.

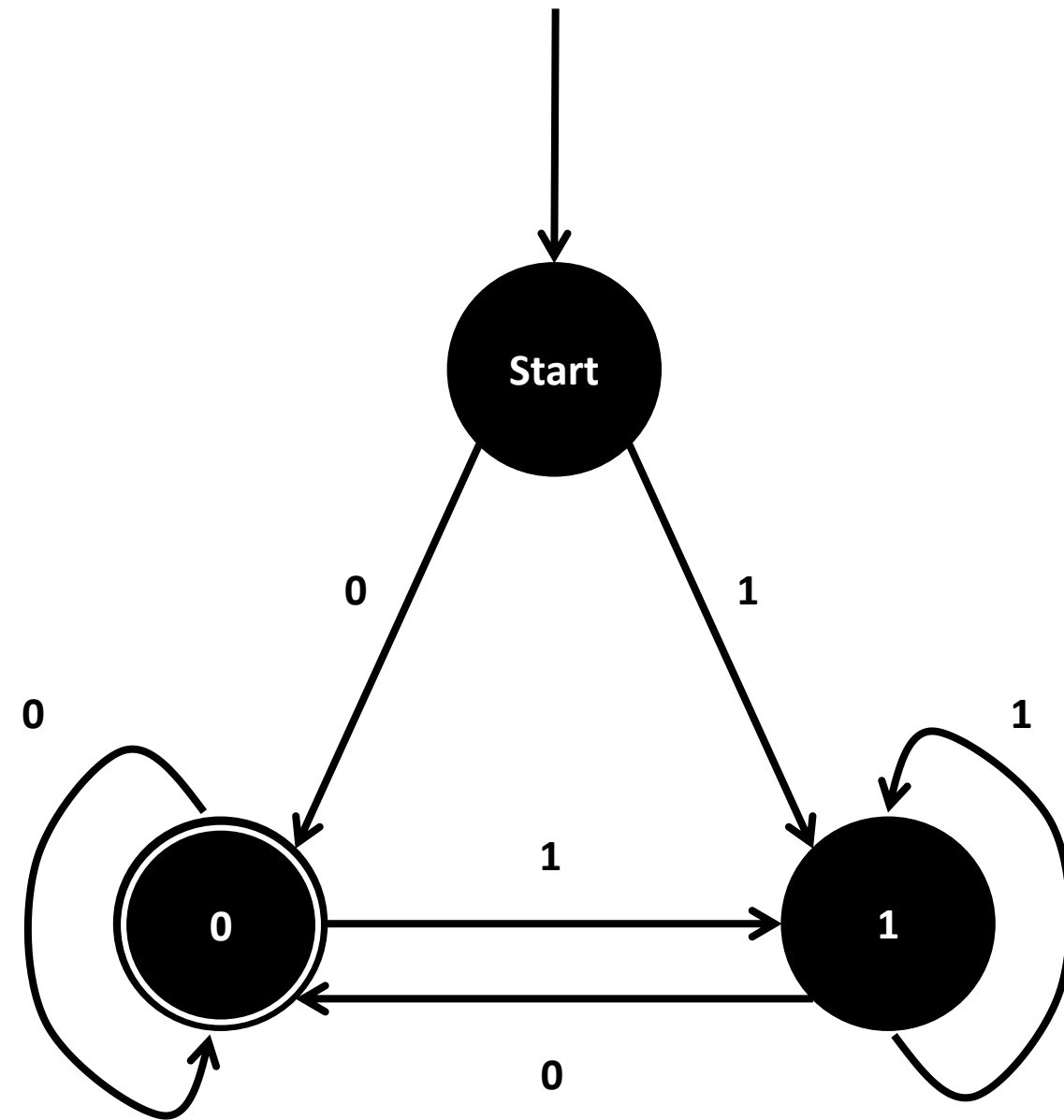
It simply replaces the stopping states with outputs being produced every time an action is taken.



Elements of an FSM with outputs

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

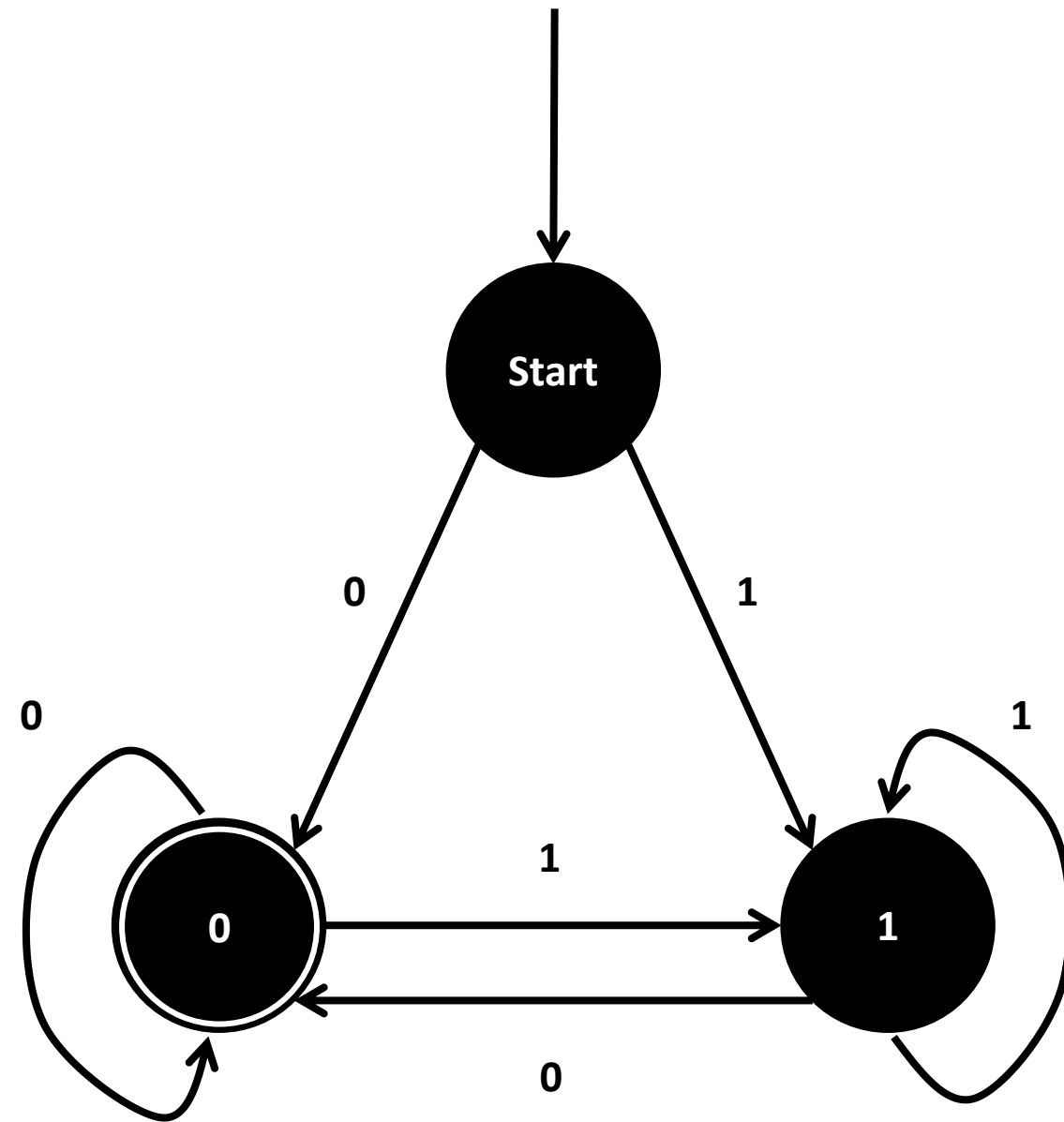
1. A finite set of **states S** .
2. A finite set of **inputs or actions A** .
3. A **starting state $s_0 \in S$** .
4. A **transition function f** , or **transition table**, which describe the **transition logic** in the FSM.



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 $s \in S$.

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



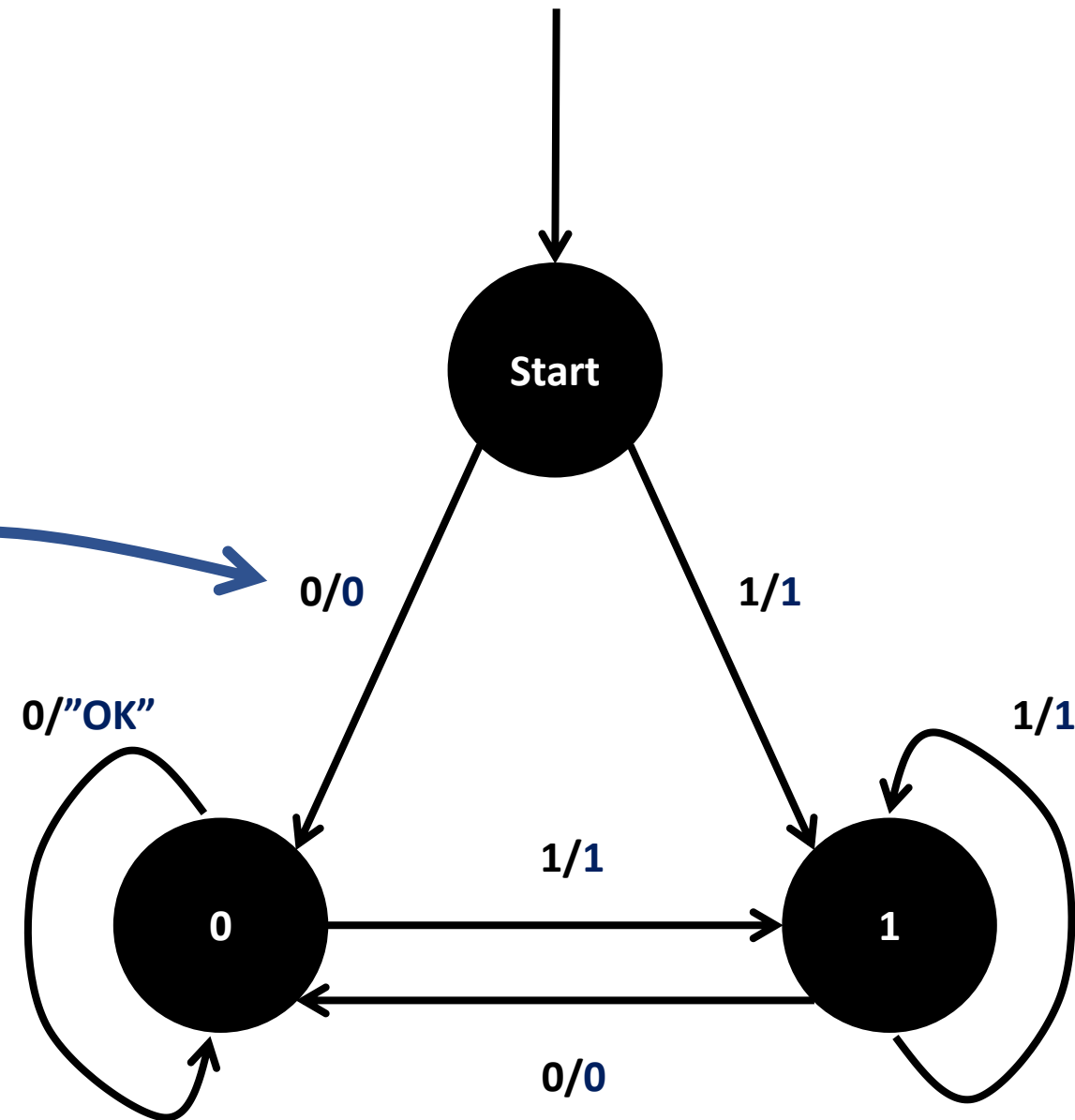
Elements of an FSM with outputs

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

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

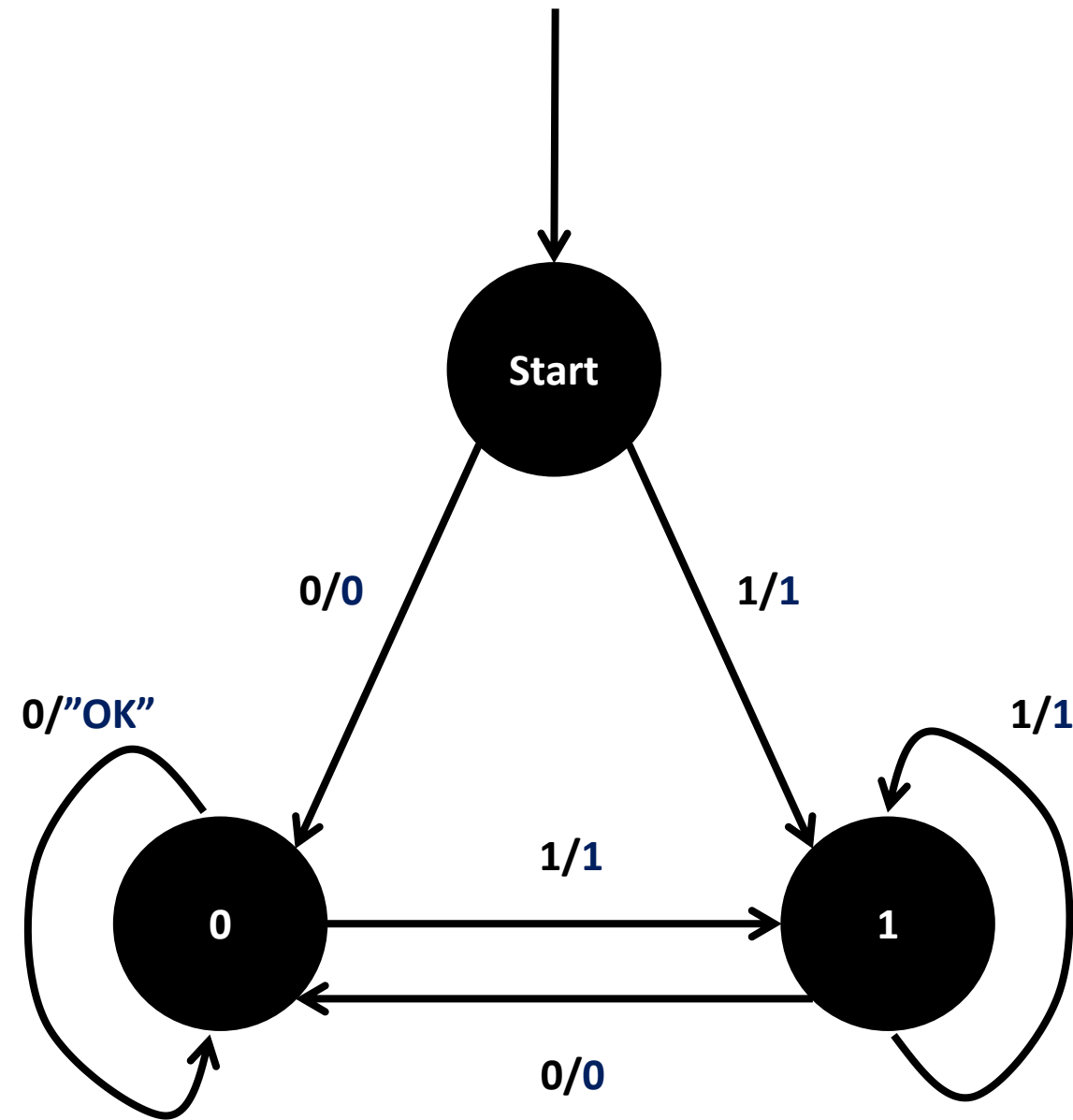
When on start node, using action 0 produces an output 0.



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 .



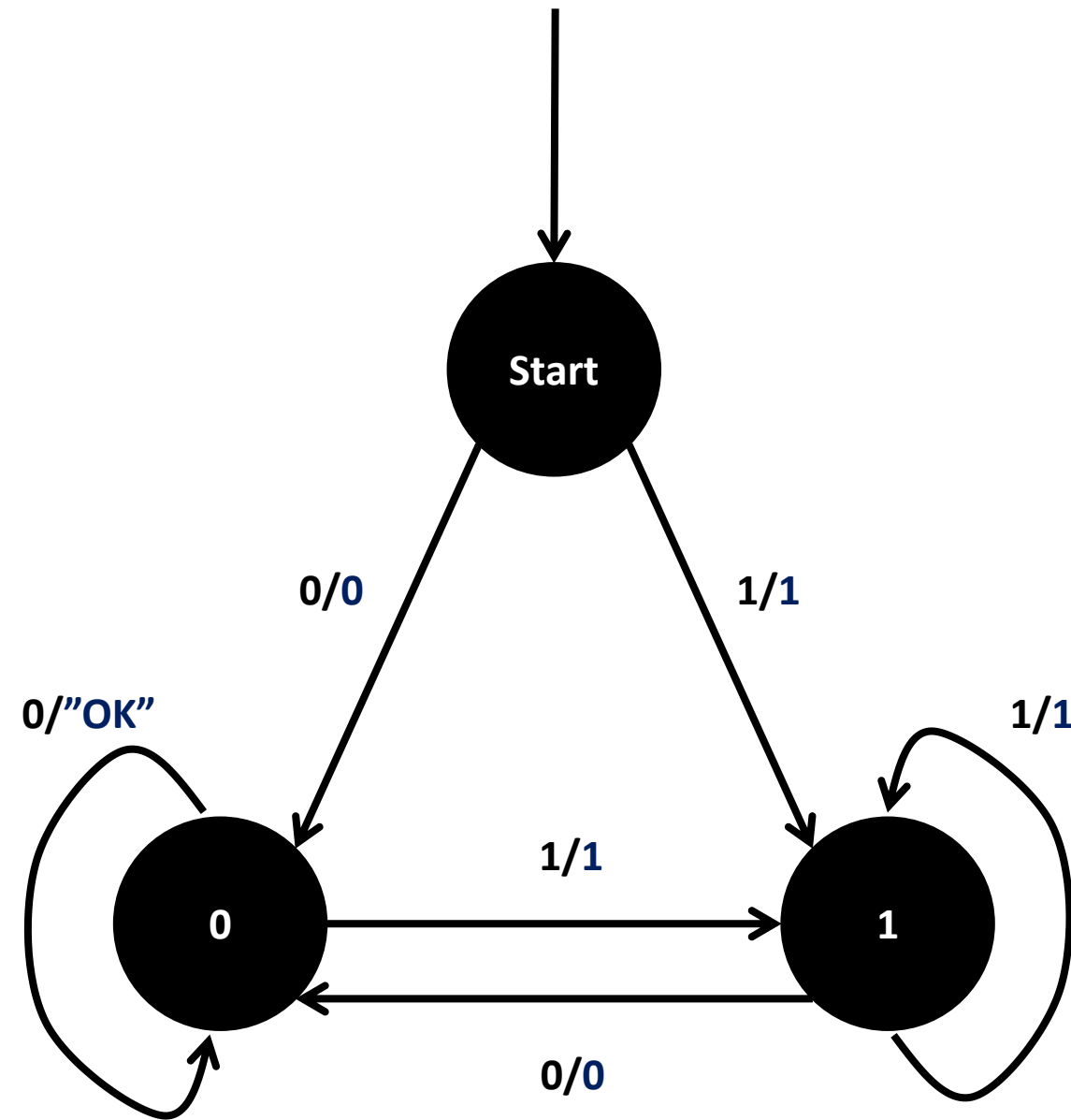
Practice 3

In the FSM on the right, the output set Y is defined as $Y = \{0, 1, OK\}$, and we use the “a/y” notation on our graph.

Question: Let us assume that the FSM stops when an output “OK” is seen or the string x runs out of characters.

The FSM considers as acceptable inputs x any input that produces “OK” as an output at some point.

Which input strings x are then considered acceptable?

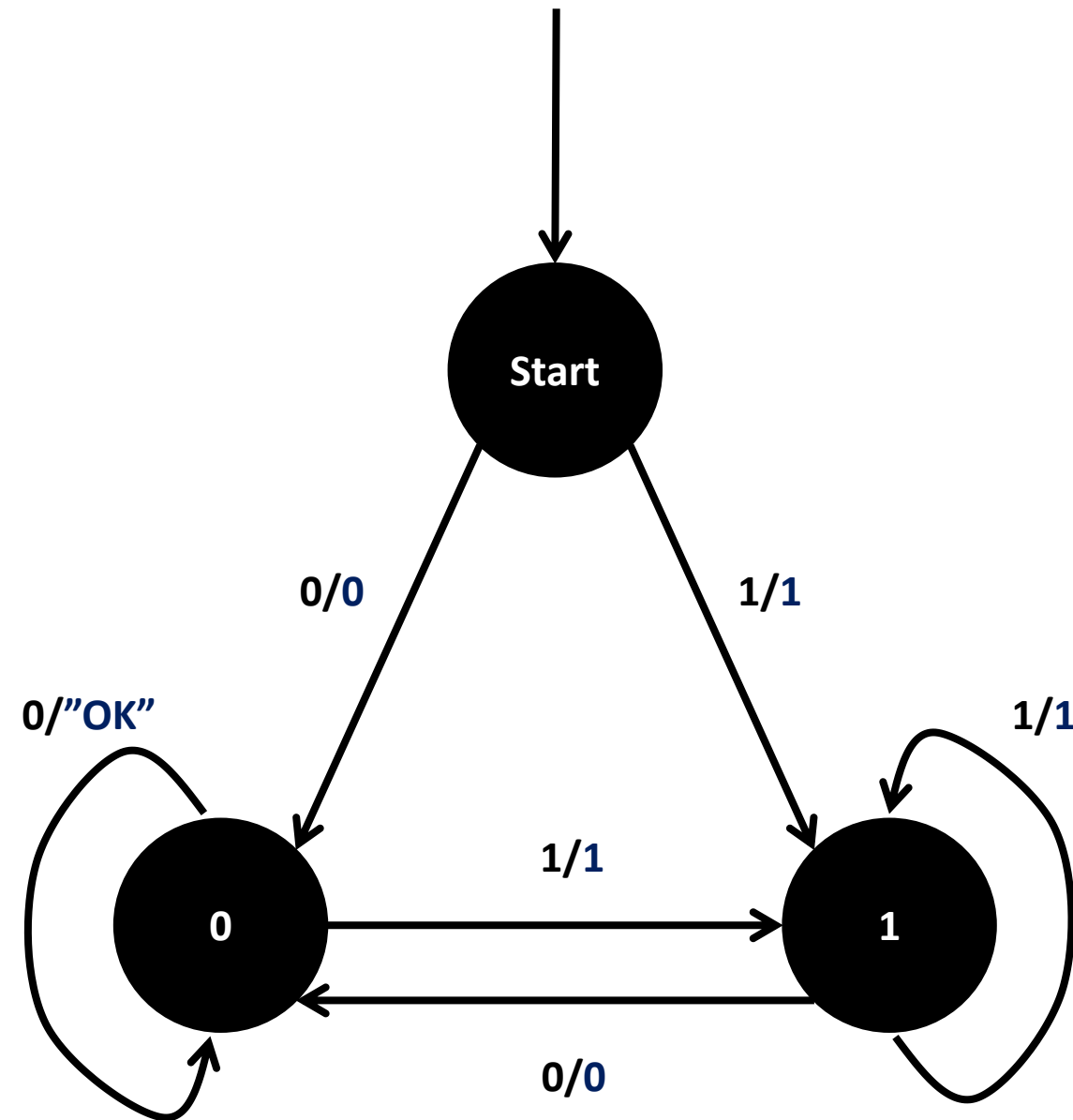


Practice 3

In the FSM on the right, the output set Y is defined as $Y = \{0, 1, OK\}$, and we use the “a/y” notation on our graph.

Answer: The “OK” output is produced if and only if there is an action $a = 0$ being taken while in state $s = 0$.

This is equivalent to having two 0s in succession somewhere in the string x . For instance, 100, 10100, 1001, 1001111110101, etc...



More advanced FSMs?

There are many variations to the state machines we have described.

For instance,

- **Deterministic Finite State Machines (DFSMs):** A deterministic finite state machine is a mathematical model used to represent a system that has a finite number of states and can transition between those states based on input.

It is basically another name for the FSMs (with no outputs) we have been playing with.

More advanced FSMs?

There are many variations to the state machines we have described.

For instance,

- **Mealy Machines:** A Mealy machine is a finite state machine where the output is dependent on both the current state and the input. That is basically another name for our FSM with outputs!
- **Moore Machines:** A Moore machine is a finite state machine where the output is dependent only on the current state. A special case of the Mealy machine.

More advanced FSMs?

There are many variations to the state machines we have described.
For instance,

- **Non-Deterministic Finite State Machines (NDFSMs):** In a non-deterministic finite state machine, the machine can transition from one state to multiple states based on the input.
- This means that there can be multiple possible paths through the machine for a given input.

More advanced FSMs?

There are many variations to the state machines we have described.

For instance,

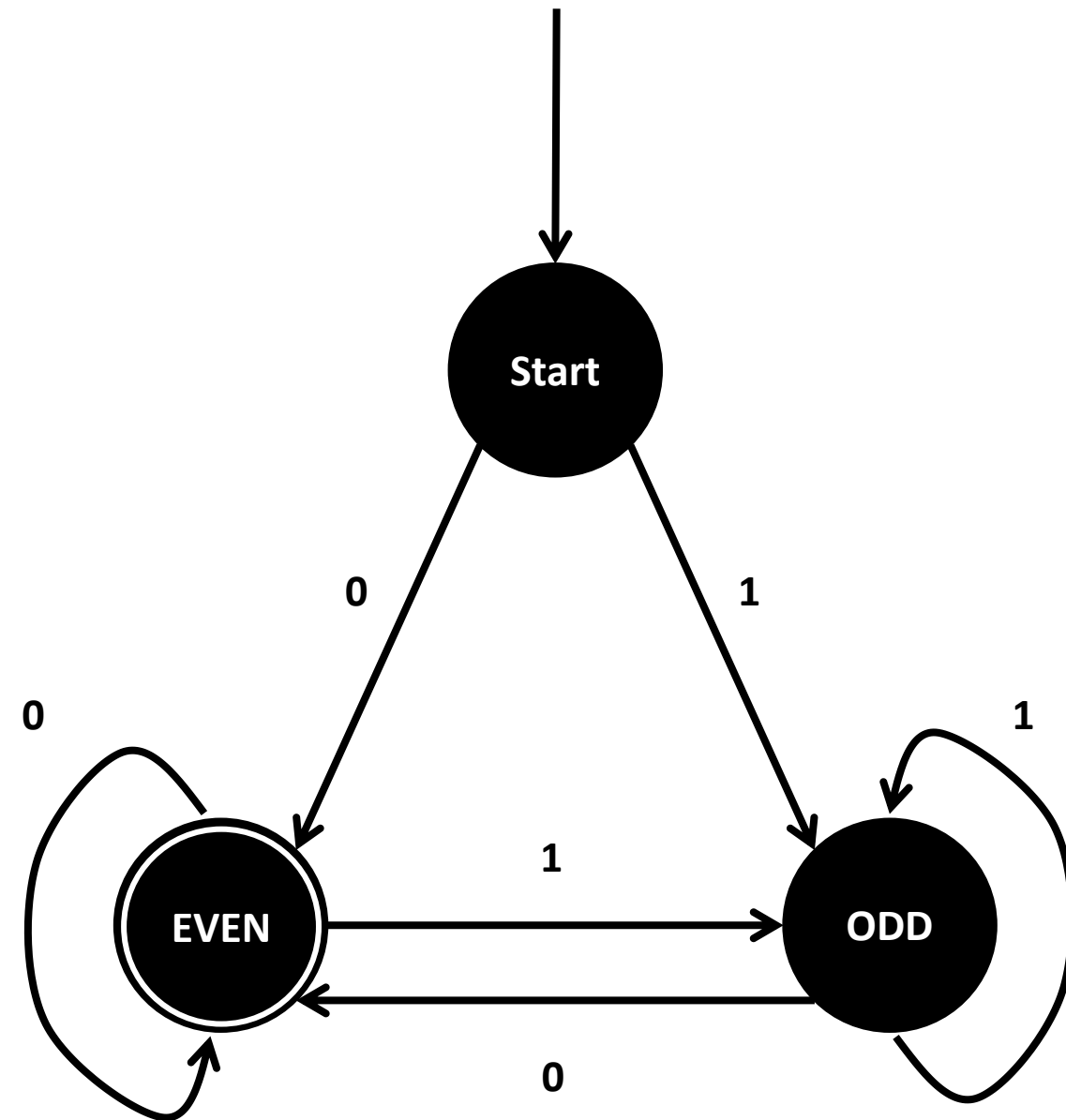
- **Stochastic Finite State Machines (SFSMs):** A stochastic finite state machine is a model used to represent a system that has a finite number of states, but where the transitions have randomness aspects included.
- In **Stochastic Finite State Machines**, there will be probabilities of transitioning between states.

Practice FSM

Let us consider this FSM, defined on the right.

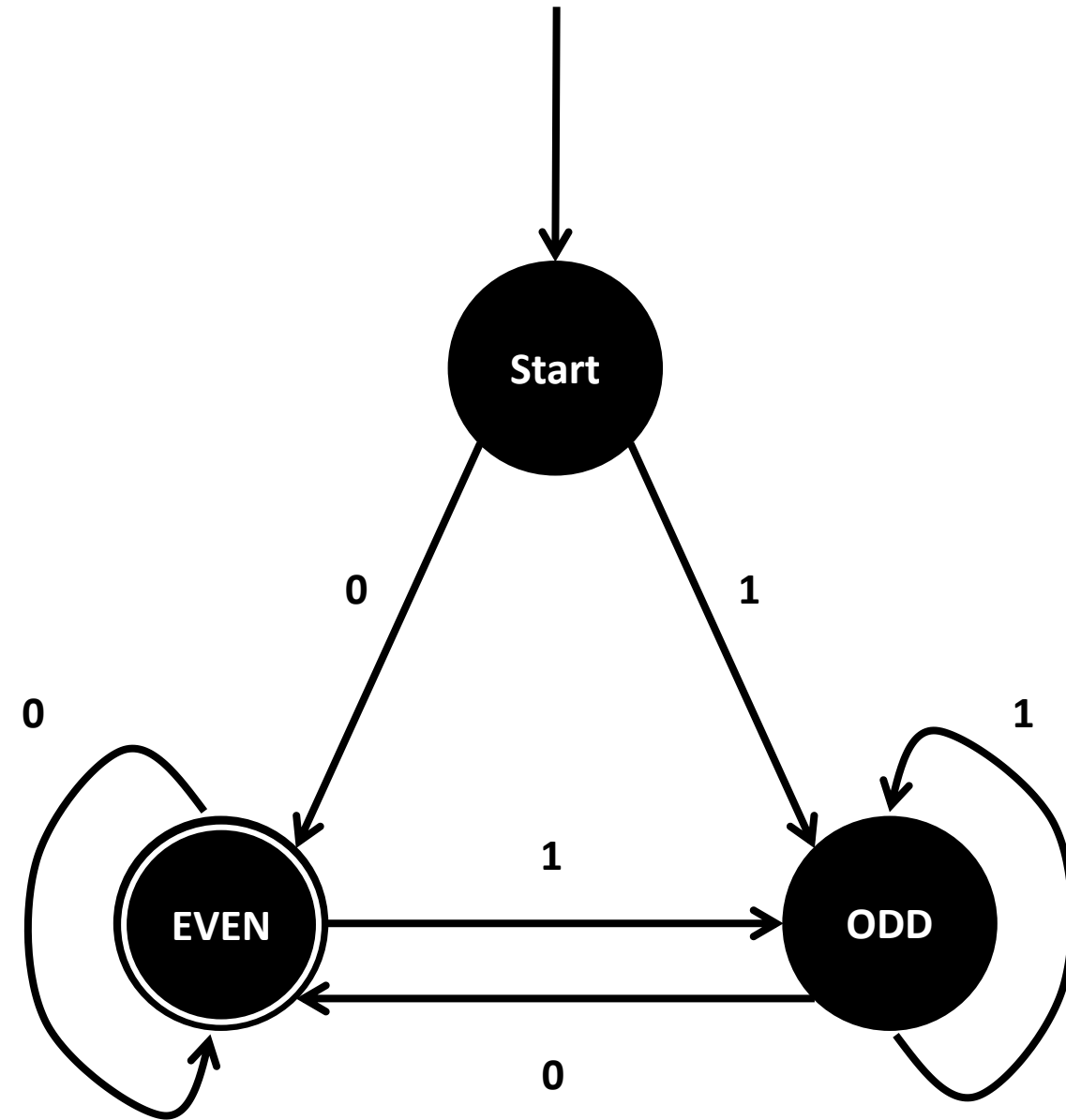
We have seen that it considers as acceptable inputs the binary strings s whose bit of least importance is 0.

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



Practice FSM

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**.

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

```
1  #include <stdio.h>
2
3  // Define possible states as an enum
4  typedef enum {
5      START,
6      EVEN,
7      ODD
8  } State;
9
```

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** $s_0 \in S$.

```
10
11 // Define our FSM object, it has only one
12 // attribute of interest, its current state.
13 typedef struct {
14     State current_state;
15 } FSM;
```

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

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.

```
10
11 // Define our FSM object, it has only one
12 // attribute of interest, its current state.
13 typedef struct {
14     State current_state;
15 } FSM;
```

```
31 // Main
32 int main() {
33     // Initialize FSM
34     FSM f = {START};
```

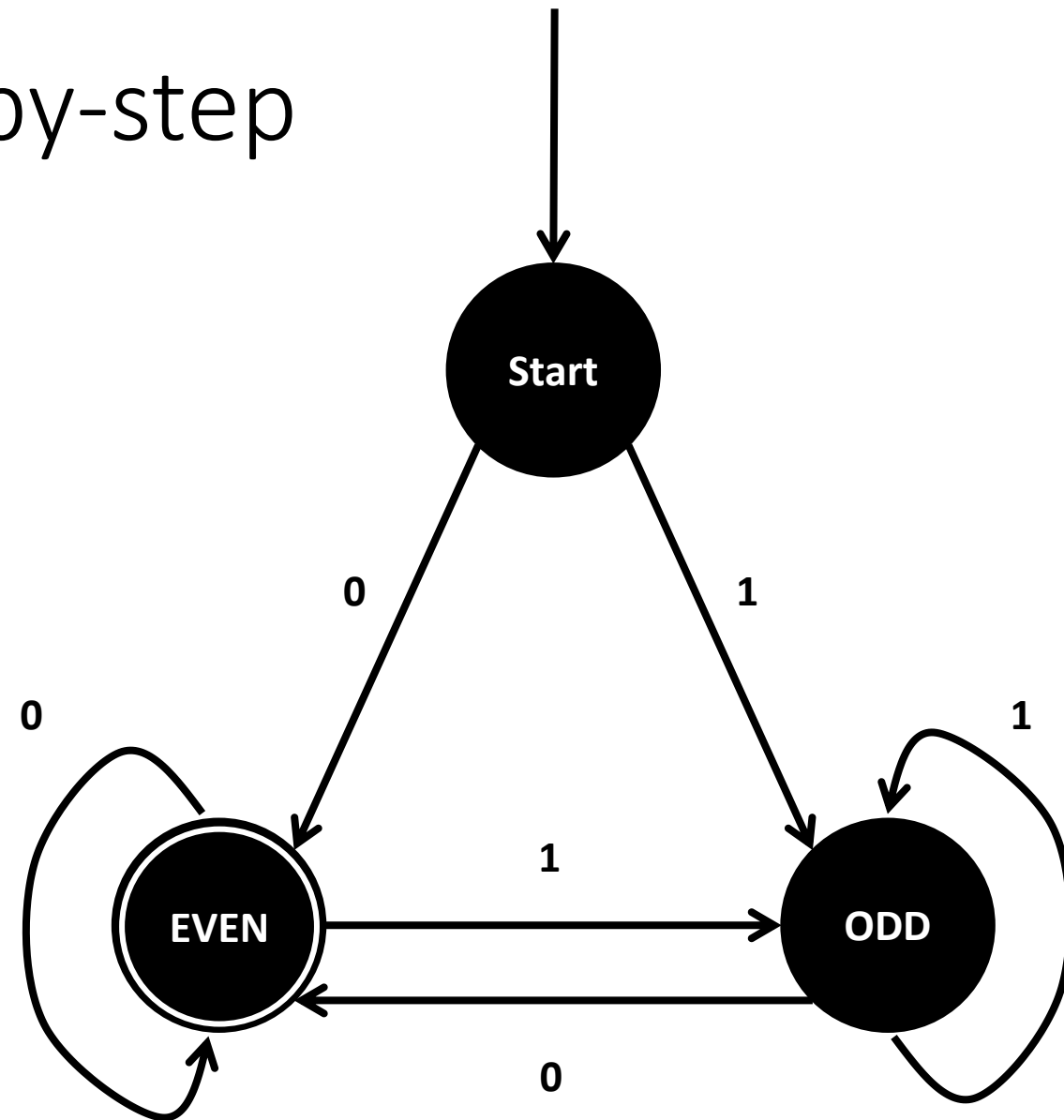
Coding a basic FSM step-by-step

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.

```
18 // Define our FSM transition function.
19 void update_state(FSM *f, int input) {
20     if (input == 1) {
21         f->current_state = ODD;
22         printf("New state is odd.\n");
23     }
24     else if (input == 0) {
25         f->current_state = EVEN;
26         printf("New state is even.\n");
27     }
28 }
```

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()*...)

```
31 // Main
32 int main() {
33     // Initialize FSM
34     FSM f = {START};
35
36     // Decimal representation of an int (our input)
37     int input[] = {1, 0, 1, 1, 0};
```

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.

```
39  
40  
41   
42  
43  
44  
45
```

```
// 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).

```
46 // Final display
47 switch (f.current_state) {
48     case EVEN:
49         printf("Our final state tells us the number is even.\n");
50         break;
51     case ODD:
52         printf("Our final state tells us the number is odd.\n");
53         break;
54     default:
55         break;
56 }
```

Using a transition table instead

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

```
18 // Define our FSM transition function.
19 void update_state(FSM *f, int input) {
20     if (input == 1) {
21         f->current_state = ODD;
22         printf("New state is odd.\n");
23     }
24     else if (input == 0) {
25         f->current_state = EVEN;
26         printf("New state is even.\n");
27     }
28 }
```

Using a transition table instead

Equivalently, we could have used a transition table as well.

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

```
18 // Define our transition table for new states
19 // 3 states (START, EVEN, ODD) and two actions (0, 1)
20 const State transition_table[3][2] = {
21     // 0    1
22     {EVEN, ODD}, // START
23     {EVEN, ODD}, // EVEN
24     {EVEN, ODD}  // ODD
25 };
```

Using a transition table instead

Equivalently, we could have used a transition table as well.

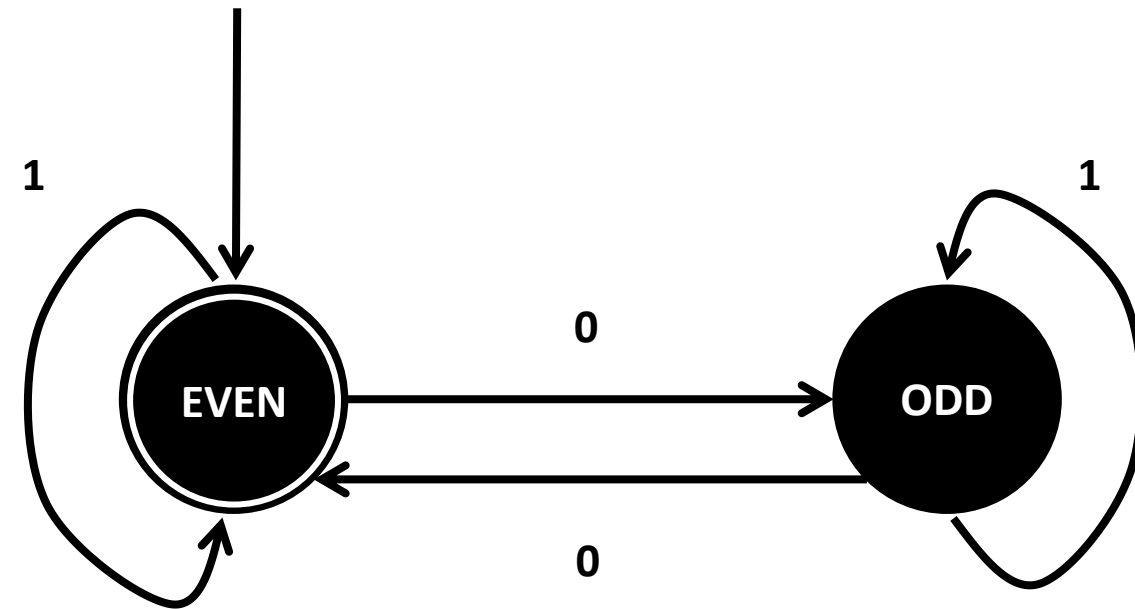
Define it as a 3 by 2 table (with 3 possible states, 2 possible actions).

Use the transition table to find the next state directly.

```
28 // Define our FSM transition function.
29 void update_state(FSM *f, int input) {
30     // Get the current state from the FSM object
31     State current_state = f->current_state;
32
33     // Look up the next state based on the current state and input
34     State next_state = transition_table[current_state][input];
35
36     // Update the FSM object with the new state
37     f->current_state = next_state;
38 }
```

Practice 4

Consider the 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**.

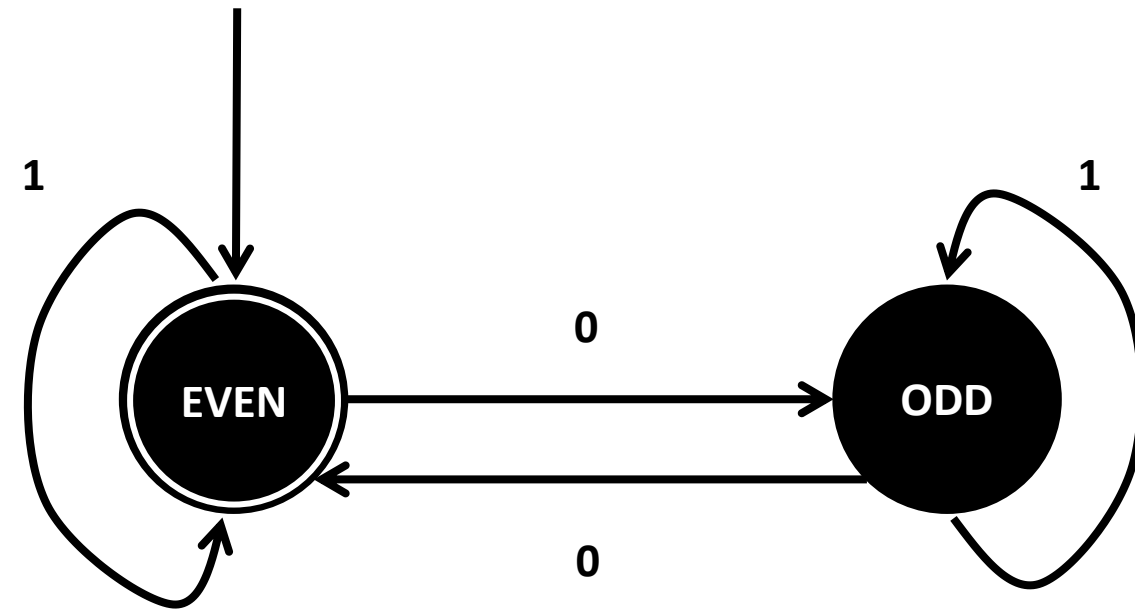


Practice 4

Check the ***main.c*** file in the “3. *Practice 4 template*” folder.

Modify the code to implement this FSM !

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



Conclusion

In this lecture,

- A reminder on Finite State Machines, with and without outputs.
- A guided implementation of FSMs without outputs.
- (We leave the implementation of FSM with outputs as extra practice!)

Let us use the remaining time of today's session to try and implement the FSMs in Practice Activities 1-4!