Finite State Machines

Table of Contents

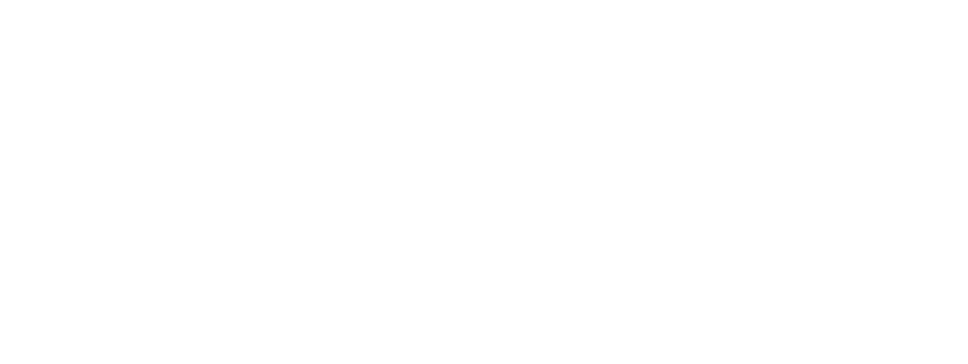
[Symbols 3](#_Toc132740230)

[Examples 4](#_Toc132740231)

[Regular Languages 7](#_Toc132740232)

[Unions of Languages 8](#_Toc132740233)

A **Finite State Machine** is any machine which has a set of **states**, , with the **starting state** of and the **final state** of . The machine takes a set of **alphabets**, , and uses the **current state**, , along with the provided input symbol in a **transition function**, , to decide how to change states. It is acceptable for a finite state machine to have **multiple possible final states**.



In the state diagram above, a given input is said to be **processed** if it manages to cause the machine to change states from the initial state to the final state. For example, the input values 0011 and 001011 will both work for the given state machine, but the input value 001010 will not. Basically, any input which contains the substring ‘11’ will work as intended.

The **alphabets** mentioned in the formal definition refer to the symbols we are providing as the input. For the above example, . A series of alphabets creates a **string**, , and the set of strings that are acceptable for the machine is called the machine’s **language**, . For example, the language for the state machine shown above consists of all strings which contain the sequence of input symbols ‘11’, i.e., .

Suppose we need to build a machine which has a language consisting of strings that start and end with 1. That machine would look like this:



The diagram explicitly shows a **dead state**, which is the state reached when the input has become such that it is impossible for the machine to continue. For our example, if the initial input symbol is 0, it is impossible for whatever string we are providing to be acceptable by this machine. Typically, the dead state is not shown in state diagrams. The arrow is just not there, so the dead state is implied.

## Symbols

– The language contains the symbols and . Additionally, providing no symbols is also valid.

– The machine takes no input. The language does not have any symbols.

– Symbols are interpreted as single digits.

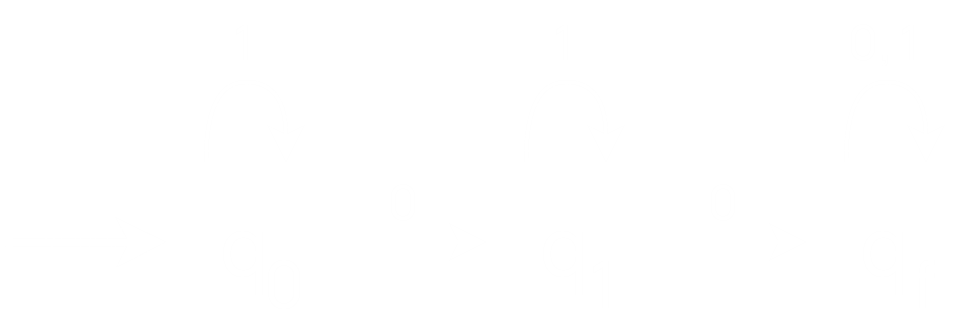
– Symbols are interpreted as double digits.

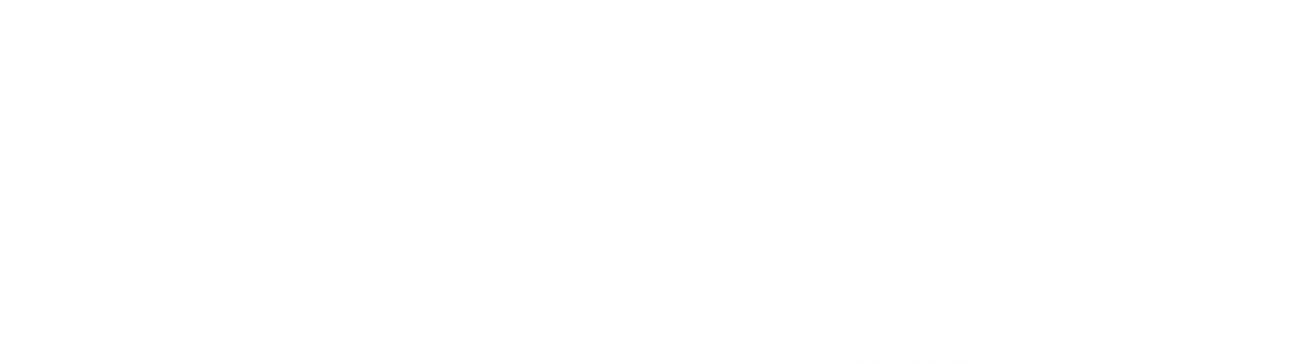
- Symbols can be repeated zero or more times.

- Symbols can be repeated one or more times.

## Examples

defines a language that contains a substring that starts and ends with and can contain any number of s in between. The diagram for such a machine is shown below.





The double circle in the diagram implies a valid final state.



Suppose the current input is . There are 3 possible states for our machine to be in:

* State A, where , where is some unknown
* State B, where
* State C, where

If the next symbol is , this is the same as saying we are multiplying by .

If we are currently at state A,

This means we stay at state A.

If we are currently at state B

which means we go to state C.

If we are currently at state C

which means we go to state B.

If the next symbol is , this is the same as saying we are multiplying by and adding .

If we are currently at state A

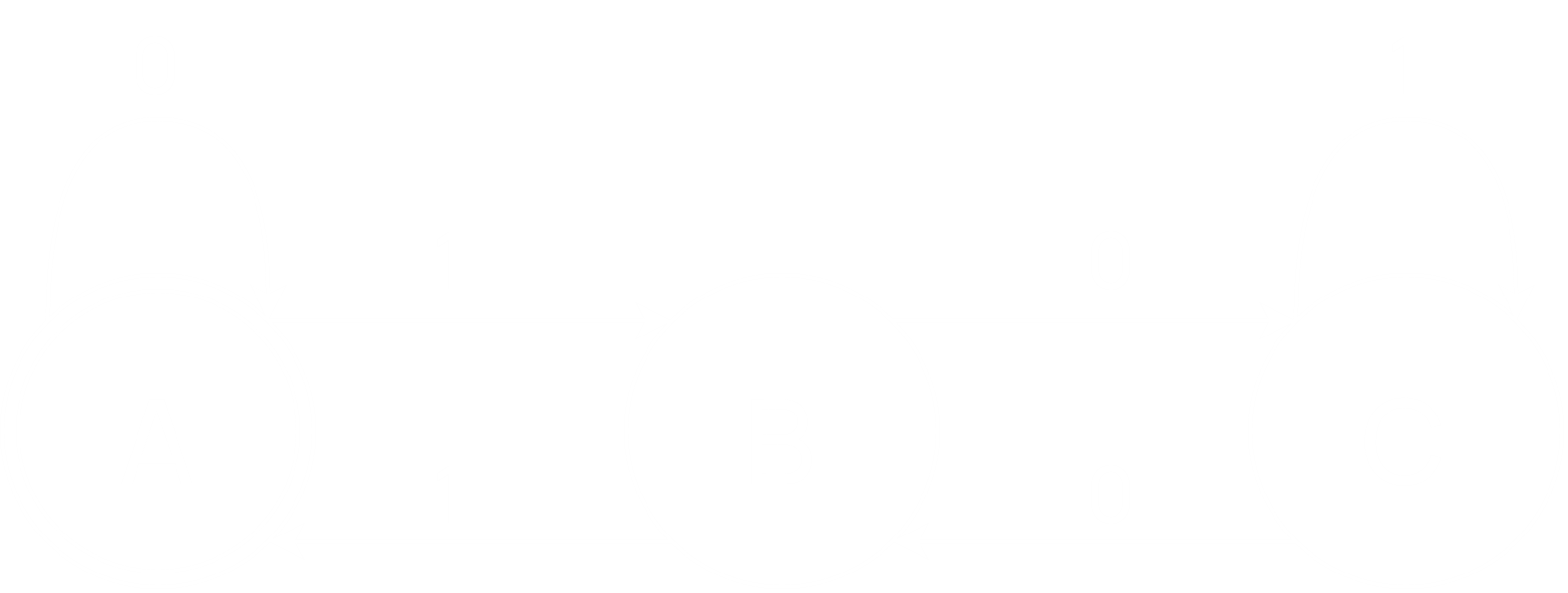
which means we go to state B.

If we are currently at state B

which means we go to state A.

If we are currently at state C

which means we stay at state C.



## Regular Languages

A language is said to be a **regular language** if some finite state machine recognizes it. This is easier to understand if we look at examples of languages that are not regular.

An example of this language could be . This is a repetition of the sequence two times. No finite state machine will be able to accept this because there is no way to know that the second sequence is the second one and not the first. A finite state machine has **no memory**. It only remembers its current state and changes states based on the next symbol given to it.

A better example of the same issue could be , where the finite state machine would be unable to remember whether the same number of s has been given as the number of s.

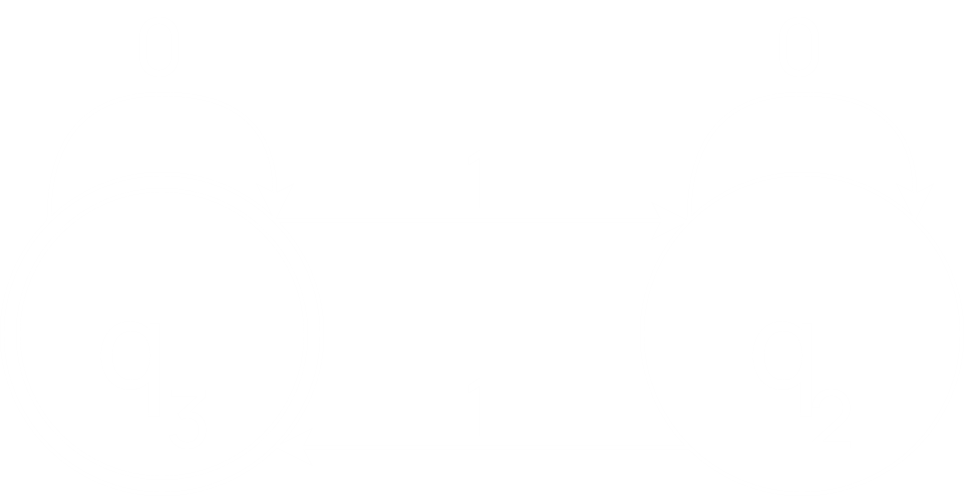
A machine that has a bit more memory and can process languages that are not regular is a **Pushdown Automate** (PDA) machine, which we will study later.

There are a few rules governing regular languages:

* The union of two regular languages is a regular language.
* The intersection of two regular languages is a regular language.
* The concatenation (cartesian product of strings) of two regular languages is a regular language.
* The Kleene closure (all possible finite sequences that can be made from the strings) of a regular language is a regular language.
* The complement of a regular language is a regular language.

## Unions of Languages



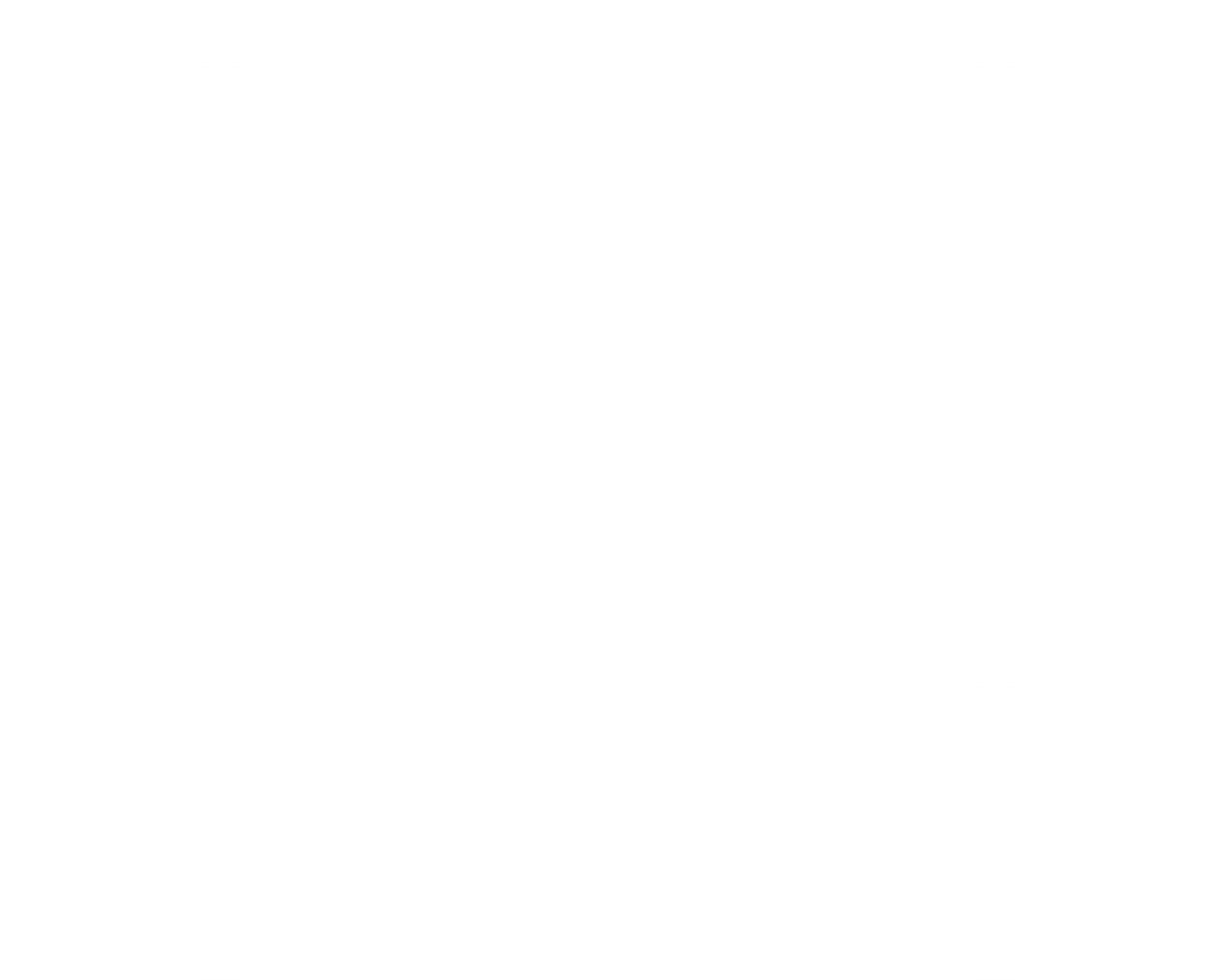


The two machines above, and , work with a language that accepts strings with an even number of 0s in them and a language that accepts strings with an odd number of 1s in them.

We know that should give us a regular language that accepts either of those two cases. Thus, it should be possible to build a machine for this.

One way to solve this might be to run the input on first and then and check if either machine accepts the input. Unfortunately, we do not have memory and will be unable to remember whether we ran successfully or failed.

Instead, we run the input **simultaneously** on both models.



Formally, for the new machine ,

Initial State

, but we assume