MODULE I:

1. Introduction

The term "Automata" is derived from the Greek word "αὐτόματα" which means "self-acting". An **automaton** (Automata in plural) is an abstract self-propelled computing device which follows a predetermined sequence of operations automatically.

An automaton with a finite number of states is called a **Finite Automaton (FA)** or **Finite State Machine (FSM).**

2. Why to study Theory of Computation?

Theory of computation is mainly concerned with the study of how problems can be solved using algorithms. It is the study of mathematical properties both of problems and of algorithms for solving problems that depend on neither the details of today's technology nor the programming language.

It is still useful in two key ways:

- It provides a set of ***abstract structures*** that are useful for solving certain classes of problems. These abstract structures can he implemented on whatever hardware/software platform is available
- It defines provable limits to ***what can be computed*** regardless of processor speed or memory size. An understanding of these limits helps us to focus our design effort in areas in which it can pay off, rather than on the computing equivalent of the search for a perpetual motion machine.

The goal is to discover fundamental properties of the problems like:

- Is there any computational ***solution*** to the problem? 1f not. is there a restricted but useful variation of the problem for which a solution does exist?
- If a solution exists, can it be implemented using some ***fixed amount of memory***?
- If a solution exists. how ***efficient*** is it? More specifically. how do its time and space requirements grow as the size of the problem grows?
- Are there groups of problems that are ***equivalent*** in the sense that if there is an efficient

solution to one member of the group there is an efficient solution to all the others?

Applications of theory of computation:

- ***Development of Machine Languages***: Enables both machine-machine and person-machine communication. Without them, none of today's applications of computing could exist. Example: Network communication protocols, HTML etc.
- ***Development of modern programming languages:*** Both the design and the implementation of modern programming languages rely heavily on the theory of context-free languages. Context- free grammars are used to document the languages syntax and they form the basis for the parsing techniques that all compilers use.
- ***Natural language processing***: It is a field of computer science, artificial intelligence, and computational linguistics concerned with the interactions between computers and human (natural) languages.
- ***Automated hardware systems:*** Systems as diverse as parity checkers, vending machines, communication protocols, and building security devices can be straightforwardly described as finite state machines, which is a part of theory of computation.
- ***Video Games:*** Many interactive video games are use large nondeterministic finite state machines.
- ***Security*** is perhaps the most important property of many computer systems. The undecidability results of computation show that there cannot exist a general-purpose method for automatically verifying arbitrary security properties of programs.
- ***Artificial intelligence:*** Artificial intelligence programs solve problems in task domains ranging from medical diagnosis to factory scheduling. Various logical frameworks have been proposed for representing and reasoning with the knowledge that such programs exploit.
- ***Graph Algorithms:*** Many natural structures, including ones as different as organic molecules and computer networks can be modeled as graphs. The theory of complexity tells us that, is there exist efficient algorithms for answering some important questions about graphs. Some questions are "hard", in the sense that no efficient algorithm for them is known nor is one likely to be developed.

3. Strings

Alphabet

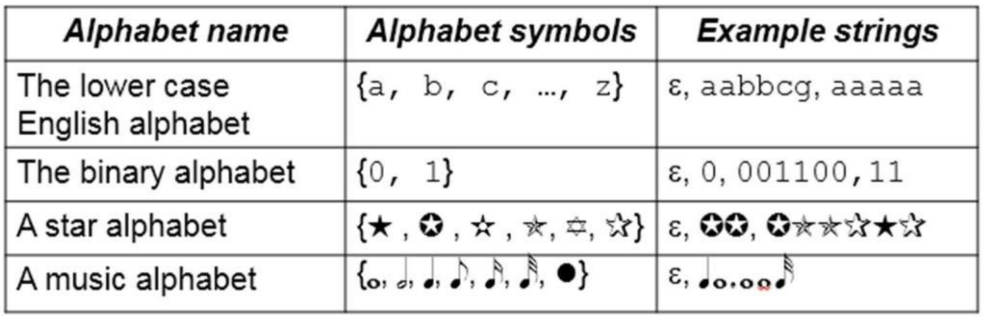
Definition: An **alphabet** is any finite set of symbols denoted by Σ (Sometimes also called as **characters** or **symbols**).

Example: Σ = {a, b, c, d} is an alphabet set where ‘a’, ‘b’, ‘c’, and ‘d’ are symbols.

String

Definition: A **string** is a finite sequence of symbols taken from Σ.

Example: 'cabcad' is a valid string on the alphabet set Σ = {a, b, c, d}



Alphabet name	*Alphabet symbols*	*Example strings*
The lower case English alphabet	{a, b, c, …, z}	ε, aabbcg, aaaaa
The binary alphabet	{0, 1}	ε, 0, 001100, 11
A star alphabet	{★ , ✪ , ☆ , ✯, ✡, ✰}	ε, ✪✪, ✪✯✯✰★✰
A music alphabet	{𝅝, 𝅗𝅥, ♩, ♪, 𝅘𝅥𝅯, 𝅘𝅥𝅰, ●}	ε, ♩𝅝.𝅝𝅝𝅘𝅥𝅰

3.1. Functions on Strings

Length of a String

Definition: It is the number of symbols present in a string. (Denoted by |.|).

Examples: If ***s*** ='cabcad', | ***s*** |= 6; Also |11001101| = 7

If | ***s*** |= 0, it is called an empty string, denoted by **ε.** |ε| = 0

Concatenation of strings: The *concatenation* of two strings *s* and *t*, written s||*t* or simply *st*, is the string formed by appending t to *s*. For example, if *x* = *good* and *y* = *bye*, then *xy* = *goodbye*. So |xy| = |x| + |y|.

The empty string, *e,* is the **identity** for concatenation of strings. *(xe = ex = x).* Concatenation, as a function defined on strings is associative. *(st)w = s (tw).*

String Replication

For each string *w* and each natural number *i*, the string $\boldsymbol{w^i}$ is defined as:

$$w^0 = \varepsilon$$
$$w^{i+1} = w^i w$$

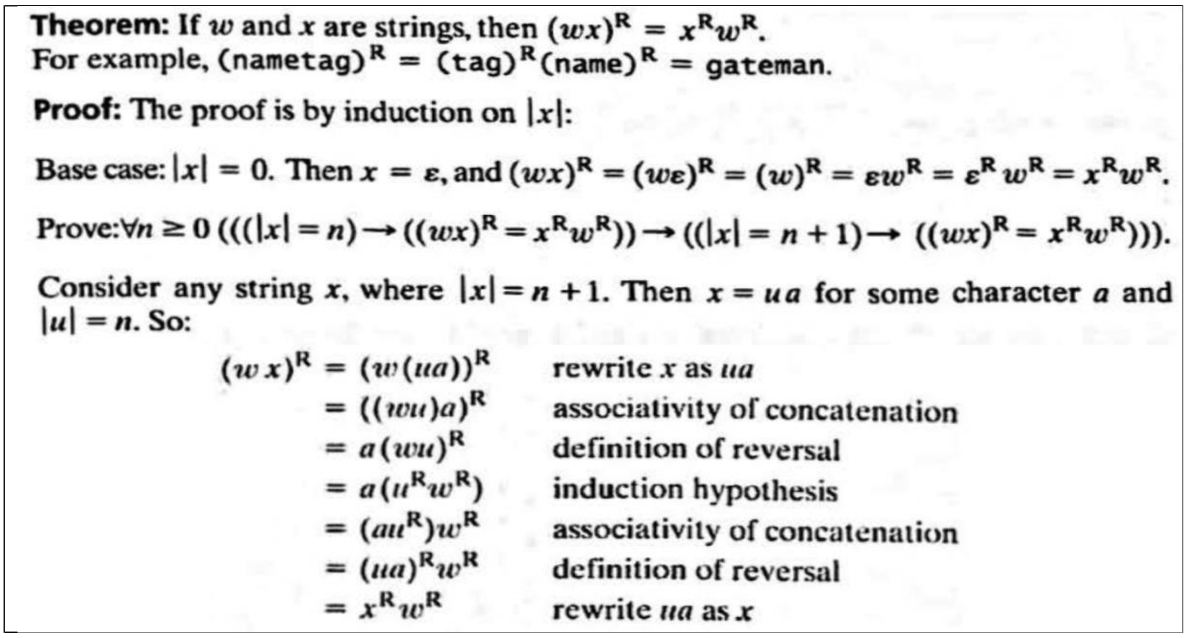
Example: a^3 = aaa, $(bye)^2$ = byebye, a^0b^3 = bbb

String Reversal: For each string ***w***, the reverse of ***w***, written as $\boldsymbol{w^R}$, is defined as:

If $|w| = 0$ then $w^R = w = \varepsilon$.

If $|w| \geq 1$ then $\exists a \in \Sigma\ (\exists u \in \Sigma^*\ (w = ua))$, (i.e., the last character of w is a.)

Then define $w^R = au^R$.



Theorem: If w and x are strings, then $(wx)^R = x^R w^R$.
For example, $(\text{nametag})^R = (\text{tag})^R(\text{name})^R = \text{gateman}$.

Proof: The proof is by induction on $|x|$:

Base case: $|x| = 0$. Then $x = \varepsilon$, and $(wx)^R = (w\varepsilon)^R = (w)^R = \varepsilon w^R = \varepsilon^R w^R = x^R w^R$.

Prove: $\forall n \geq 0\ (((|x| = n) \rightarrow ((wx)^R = x^R w^R)) \rightarrow ((|x| = n+1) \rightarrow ((wx)^R = x^R w^R)))$.

Consider any string x, where $|x| = n + 1$. Then $x = ua$ for some character a and $|u| = n$. So:

$(wx)^R = (w(ua))^R$	rewrite x as ua
$= ((wu)a)^R$	associativity of concatenation
$= a(wu)^R$	definition of reversal
$= a(u^R w^R)$	induction hypothesis
$= (au^R)w^R$	associativity of concatenation
$= (ua)^R w^R$	definition of reversal
$= x^R w^R$	rewrite ua as x

3.2. Relations on strings

Substring: A string ***s*** is a substring of a string of t iff *s* occurs contiguously as part of t.
For example: aaa is a substring of aaabbbaaa, aaaaaa is not a substring of aaabbbaaa

Proper Substring: A string r is a proper substring of a string t, iff t is a substring of t and s $\neq$ t. Every string is a substring (although not a proper substring) of itself. The empty string. e. is a substring of every string.

Prefix: A string s is a prefix of t, iff $\exists x \in \sum^*(t = sx)$. A string s is a proper prefix of a string t iff s is a prefix of t and s≠t. Every string is a prefix (although not a proper prefix) of itself. The empty string ε, is a prefix of every string. For example. the prefixes of abba are: ε, a, ab, abb, abba.

Suffix: A string s is a suffix of t, iff $\exists x \in \sum^*(t = xs)$. A string s is a proper suffix of a string t iff s is a suffix of t and s≠t. Every string is a suffix (although not a proper suffix) of itself. The empty string ε, is a suffix of every string. For example. the prefixes of abba are: ε, a, ba, bba, abba.

4. Languages

A language is a (finite or infinite) set of strings over a finite alphabet ∑. When we are talking about more than one language, we will use the notation $\sum_L$, to mean the alphabet from which the strings in the language L are formed.

Let ∑ = {a, b}. ∑* = {ε, a, b, aa, ab , ba, bb, aaa, aab }.

Some examples of languages over ∑ are:

Φ, {ε}, {a, b}, {ε, a, aa, aaa, aaaa, aaaaa}, {ε, a, aa, aaa, aaaa, aaaaa, ...}

4.1. Techniques for Defining Languages

There are many ways. Since languages are sets. we can define them using any of the set-defining techniques

Ex-1: *All a's Precede All b's,*

L = {w ϵ {a,b}*: an a's precede all b's in w}. The strings ε, a, aa, aabbb, and bb are in *L* . The strings aba, ba, and abc are not in *L*.

Ex-2: ***Strings that end in 'a'***

L = {x : ∃yϵ {a, b}*, (*x* = ya)}. The strings a, aa, aaa, bbaa and ba are in *L*. The strings ε, bab, and bca are not in *L*. *L* consists of all strings that can be formed by taking some string in {a, b}* and concatenating a single a onto the end of it.

Ex-3: ***Empty language***

L = { } = Φ, the language that contains no strings. ***Note:*** L = { ε } the language that contains a single string, ε. Note that *L* is different from Φ.

Ex-4: *Strings of all 'a' s containing zero or more 'a's*

Let *L* = { a^n : $n \geq 0$}. *L* = (ε, a, aa, aaa, aaaa, aaaaa,........)

Ex-5: We define the following languages in terms of the prefix relation on strings:

L1 = {w∊{a, b}* : no prefix of *w* contains b}= { *e* , a, aa, aaa, aaaa, aaaaa, aaaaaa, } .

L2 ={w∊ {a, b}*: no prefix of *w* starts with b}={*w* ∊{a,b}*: the first character of w is a }U{ε}.

L3= {w∊ {a, b}*; every prefix of w starts with b} =Φ. L3 is equal to Φ because ε is a prefix of every string. Since ε does not start with b, no strings meet L3 's requirement.

Languages are sets. So, a computational definition of a language can be given in two ways;

- a **language generator**, which enumerates (lists) the elements of the language, or
- a **language recognizer**, which decides whether or not a candidate string is in the language and returns *True* if it is and *False* if it isn't.

For example, the logical definition. *L* = {x: ∃*y* ∊{a, b}* *(x* = ya)} can be turned into either a language generator (enumerator) or a language recognizer.

In some cases, when considering an enumerator for a language, we may care about the order in which the elements of *L* are generated. If there exists n total order *D* of the elements of $\sum_{L,}$ then we can use *D* to define on *L* a total order called ***lexicographic order*** (written $<_L$.):

• Shorter strings precede longer ones: ∀x (∀y ((|x| < |y|) ⓪ (x $<_L$ y))) and

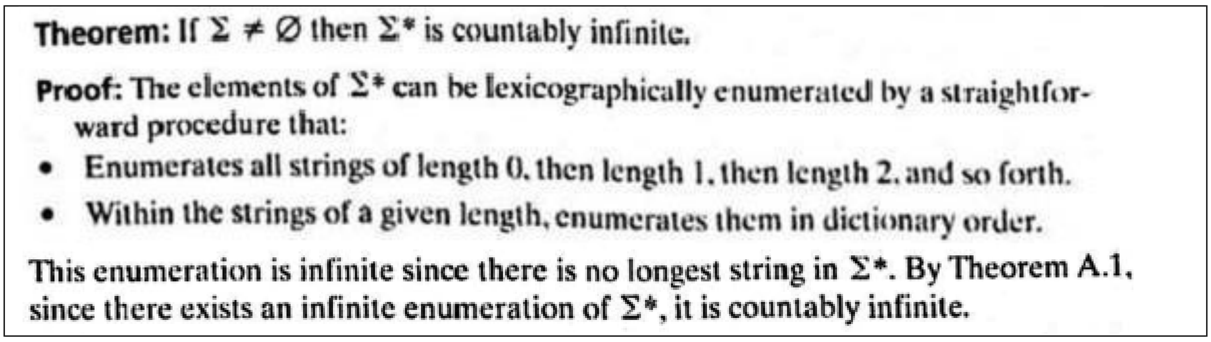
• Of strings that are the same length sort them in dictionary order using *D.*

Let L= {w ∊ {a, b}*; all a's precede all b's}. The lexicographic enumeration of Lis:

ε, a. b. aa. ab. bb. aaa. aab. abb. bbb. aaaa, aaab. aabb. abbb. bbbb. aaaaa ....

4.2. Cardinality of a Language

- Cardinality refers to the number of strings in the language.
- The smallest language over any alphabet is ϕ, whose cardinality is 0.
- The largest language over any alphabet ∑ is ∑*. Suppose that ∑ = Φ, then ∑* = {ε} and |∑*| = l. In general, |∑*| is infinite.



Theorem: If Σ ≠ ∅ then Σ* is countably infinite.

Proof: The elements of Σ* can be lexicographically enumerated by a straightforward procedure that:

- Enumerates all strings of length 0, then length 1, then length 2, and so forth.
- Within the strings of a given length, enumerates them in dictionary order.

This enumeration is infinite since there is no longest string in Σ*. By Theorem A.1, since there exists an infinite enumeration of Σ*, it is countably infinite.

4.4. Functions on Languages

Since languages are sets. all of the standard set operations are well-defined on languages. **Union**, **intersection**, **difference** and **complement** are quite useful

Let: $\Sigma = \{a, b\}$.
$L_1 = \{\text{strings with an even number of a's}\}$.
$L_2 = \{\text{strings with no b's}\} = \{\varepsilon, a, aa, aaa, aaaa, aaaaa, aaaaaa, \ldots\}$.

$L_1 \cup L_2 = \{$all strings of just a's plus strings that contain b's and an even number of a's$\}$.
$L_1 \cap L_2 = \{\varepsilon, aa, aaaa, aaaaaa, aaaaaaaa, \ldots\}$.
$L_2 - L_1 = \{a, aaa, aaaaa, aaaaaaa, \ldots\}$.
$\neg(L_2 - L_1) = \{$strings with at least one b$\} \cup \{$strings with an even number of a's$\}$.

Concatenation

Let L_1 and L_2 be two languages defined over some alphabet ∑. Then their concatenation. written L_1L_2 is:

$$L_1L_2 = \{w \in \Sigma^* : \exists s \in L_1 (\exists t \in L_2 (w = st))\}.$$

Example: Let: L_1 = {cat, dog, mouse, bird}. L_2 = { bone, food}.
L_1L_2 = { catbone, catfood, dogbane, dogfood, mousebone, mousefood, birdbone, birdfood}.

The language {ε} is the **identity for concatenation** of languages. So for all languages L,
L{ε} = {ε}L = L.
The language Φ is a zero for concatenation of languages. So, for all languages *L*, *LΦ* = *ΦL* = Φ. That Φ is a zero follows from the definition of the concatenation of two languages as the set consisting of all strings that can he formed by selecting some string 's' from the first language and some string 'l' from the second language and then concatenating them together. There are no ways to select a string from the empty set.

Concatenation on languages is **associative**. So, for all languages L_1L_2 and L_3:

$$((L_1L_2)L_3 = L_1 (L_2L_3)).$$

Reverse

Let L be a language defined over some alphabet ∑. Then the reverse of L , written L^R is:
$L^R = \{w \in \Sigma^*: w = x^R \text{ for some } x \in L\}$.
In other words, L^R is the set of strings that can be formed by taking some string in L and reversing it

Theorem: If L_1 and L_2 are languages, then $(L_1L_2)^R = L_2{}^R L_1{}^R$.

Proof: If x and y are strings, then $\forall x (\forall y ((xy)^R = y^R x^R))$ Theorem 2.1

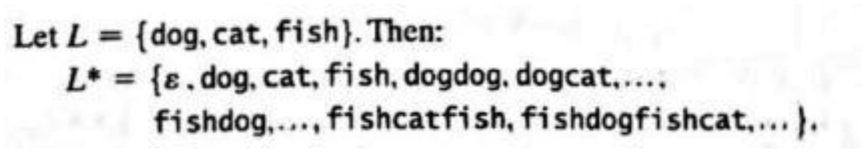
$(L_1L_2)^R = \{(xy)^R : x \in L_1 \text{ and } y \in L_2\}$	Definition of concatenation of languages
$= \{y^R x^R : x \in L_1 \text{ and } y \in L_2\}$	Lines 1 and 2
$= L_2{}^R L_1{}^R$	Definition of concatenation of languages

Kleene Star

Definition: The **Kleene star** denoted by **Σ^*,** is a unary operator on a set of symbols or strings, Σ, that gives the infinite set of all possible strings of all possible lengths over Σ including ε.

Representation: $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \ldots\ldots$ where Σ^p is the set of all possible strings of length p.

Example: If $\Sigma = \{a, b\}$, $\Sigma^* = \{\lambda, a, b, aa, ab, ba, bb, \ldots\ldots\ldots\ldots\}$



Let $L = \{\text{dog}, \text{cat}, \text{fish}\}$. Then:

$L^* = \{\varepsilon, \text{dog}, \text{cat}, \text{fish}, \text{dogdog}, \text{dogcat}, \ldots,$
$\text{fishdog}, \ldots, \text{fishcatfish}, \text{fishdogfishcat}, \ldots\}$.

Kleene Closure / Plus

Definition: The set **Σ^+** is the infinite set of all possible strings of all possible lengths over Σ excluding ε. $\Sigma^+ = \Sigma^* - \{\varepsilon\}$ Representation:

$\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \ldots\ldots$

Example: If $\Sigma = \{a, b\}$, $\Sigma^+ = \{ a, b, aa, ab, ba, bb, \ldots\ldots\ldots\ldots\ldots \}$

Closure: A set S is closed under the operation @ if for every element x & y in S, x@y is also an element of S.

4.5. A Language Hierarchy

A Machine-Based Hierarchy of Language Classes are shown in the diagram.

We have four language classes:

1. **Regular languages**, which can be accepted by some finite state machine.
2. **Context-free languages**, which can be accepted by some pushdown automaton.
3. **Decidable** (or simply D) languages. which can decided by some Turing machine that always halts.
4. **Semi-decidable** (or SD) languages, which can be semi-decided by some Turing machine that halts on all strings in the language.

Each of these classes is a proper subset of the next class, as illustrated in the Figure. As we move outward in the language hierarchy, we have access to tools with greater and expressive power. We can define $A^nB^nC^n$ as a decidable language but not as a context-free or a regular one. These matters because expressiveness generally comes at a price. The price may be: Computational efficiency, decidability and clarity.

- *Computational efficiency:* Finite state machines run in time that is linear in the length of the input string. A general context-free parser based on the idea of a pushdown automaton requires time that grows as the cube of the length of the input string. A

Turing machine may require time that grows exponentially (or faster) with the length of the input string.

- *Decidability:* There exist procedures to answer many useful questions about finite state machines. For example, does an FSM accept some particular string? Is an FSM minimal? Are two FSMs identical? A subset of those questions can be answered for pushdown automata. None of them can be answered for Turing machines.
- *Clarity:* There exist tools that enable designers to draw and analyze finite state machines. Every regular language can also be described using the regular expression pattern language. Every context-free language, in addition to being recognizable by some pushdown automaton, can be described with a context-free grammar

5. Finite State Machines (FSM)

A ***finite state machines*** (or FSM) is a computational device whose input is a string and whose output is one of two values; Accept and Reject. FSMs are also sometimes called ***finite state automata*** or ***FSAs*.**

5.1. Deterministic FSM

- We begin by defining the class of FSMs whose behavior is **deterministic**.
- These machines, makes exactly one move at each step
- The move is determined by the current state and the next input character.

Definition: Deterministic Finite State Machine (DFSM) is M: $M = (K, \Sigma, \delta, s, A)$, where:
K is a finite set of states
Σ is an alphabet
$s \in K$ is the initial state
$A \subseteq K$ is the set of accepting states, and
δ is the transition function from $(K \times \Sigma)$ to K

Configuration: A ***Configuration*** of a DFSM M is an element of $K \times \Sigma^*$. Configuration captures the two things that make a difference to M's future behavior: i) its current state, the input that remains to be read.

The ***Initial Configuration*** of a DFSM M, on input w, is (s_M, w) , where s_M is start state of M

The transition function δ defines the operation of a DFSM M one step at a time. δ is set of all pairs of states in M & characters in Σ. (Current State, Current Character) © New State

Relation 'yields': Yields-in-one-step relates configuration, to configuration-1to configuration- 2 iff M can move from canfiguration-1, to configuration-2 in one step. Let c be any element of Σ and let ***w*** be any element of Σ*, then,

$$(q_1, cw) \vdash_M (q_2, w) \text{ iff } ((q_1, c), q_2) \in \delta$$

$\vdash_M{}^*$ is the reflexive, transitive closure of $\vdash_M$

Complete vs Incomplete FSM

Complete FSM: A transition is defined for every possible state and every possible character in the alphabet. Note: This can cause FSM to be larger than necessary, but ALWAYS processes the entire string

Incomplete FSM: One which defines a transition for every possible state & every possible character in the alphabet which can lead to an accepting state Note: If no transition is defined, the string is *Rejected*

Computation: A Computation by M is a finite sequence of configurations $C_0, C_1, \ldots, C_n$ for some $n \geq 0$ such that:

- C_0 is an initial configuration,
- C_n is of the form (q, ε), for some state $q \in K_M$
  - ε indicates empty string, entire string is processed & implies a complete DFSM
- $C_0 \mid\!-_M C_1 \mid\!-_M C_2 \mid\!-_M \ldots \mid\!-_M C_n$.

However, M **Halts** when the last character has to be processed or a next transition is not defined

Acceptance / Rejection

A DFSM M, **Accepts** a string w iff $(s, w) \mid\!-_M{}^* (q, \varepsilon)$, for some $q \in A_M$.

A DFSM M, **Rejects** a string w iff $(s, w) \mid\!-_M{}^* (q, \varepsilon)$, for some $q \notin A_M$.

The **language accepted by M**, denoted *L*(*M*), is the set of all strings accepted by *M*.

Regular languages

A language is regular iff it is accepted by some DFSM. Some examples are listed below.

- $\{w \in \{a, b\}^* \mid$ every a is immediately followed by b $\}$.
- $\{w \in \{a, b\}^* \mid$ every **a region** in w is of even length$\}$
- binary strings with odd parity.

Designing Deterministic Finite State Machines

Given some language L. how should we go about designing a DFSM to accept L? In general. as in any design task. There is no magic bullet. But there are two related things that it is helpful to think about:

- Imagine any DFSM M that accepts L. As a string w is being read by M, what properties of the part of w that has been seen so far are going to have any bearing on the ultimate answer that M needs to produce? Those are the properties that M needs to record.
- If L is infinite but M has a finite number of states, strings must "cluster". In other words, multiple different strings will all drive M to the same state. Once they have done that, none of their differences matter anymore. If they've driven M to the same state, they share a fate. No matter what comes next, either all of them cause M to accept or all of them cause M to reject.

6. Finite State Transducers

So far, we have used finite state machines as language recognizers. All we have cared about, in analyzing a machine *M*, is whether or not *M* ends in an accepting state. But it is a simple matter to augment our finite state model to allow for output at each step of a machine's operation. Often, once we do that, we may cease to care about whether *M* actually accepts any strings. Many finite state transducers are loops that simply run forever, processing inputs.

An automaton that produces outputs based on current input and/or previous state is called a **transducer**. Transducers can be of two types:

- **Moore Machine** The output depends only on the current state.

Mealy Machine The output depends both on the current state and the current input.

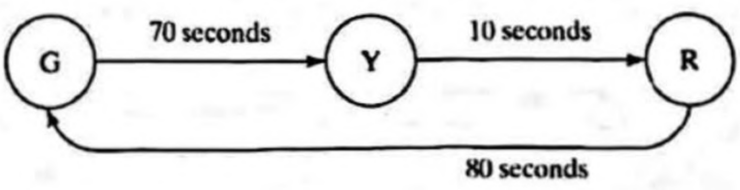
Moore Machine

One simple kind of finite state transducer associates an output with each state of a machine *M*. That output is generated whenever *M* enters the associated state. Deterministic finite state transducers of this sort are called Moore machines, after their inventor Edward Moore. A ***Moore machine M*** is a seven-tuple $(K, \Sigma, O, \delta, D, s, A)$, where:

- K is a finite set of states,
- Σ is an input alphabet,
- O is an output alphabet,
- $s \in K$ is the start state,
- $A \subseteq K$ is the set of accepting states (although for some applications this designation is not important),
- δ is the transition function. It is function from $(K \times \Sigma)$ to (K), and
- D is the display or output function. It is a function from (K) to (O^*).

A Moore machine *M* computes a function $f(w)$ iff, when it reads the input string w, its output sequence is $f(w)$.

Example: Traffic light



Mealy Machine

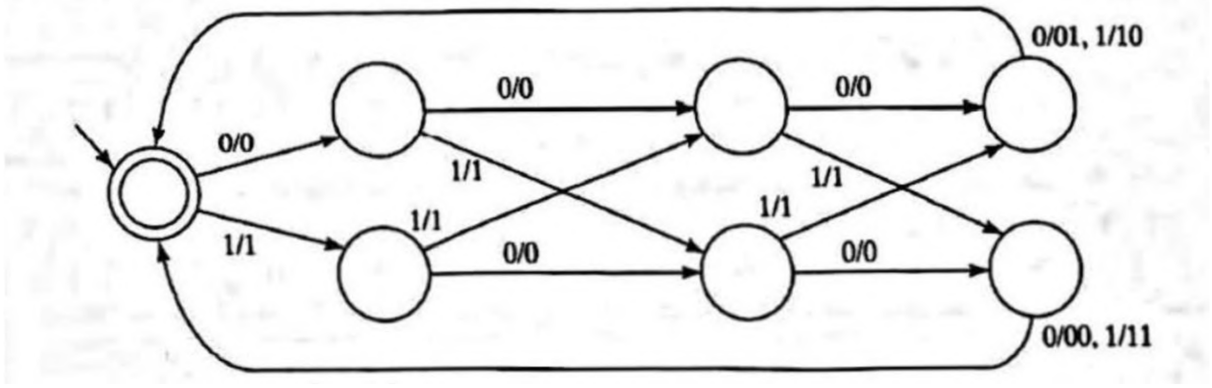
A different definition for a deterministic finite state transducer permits each machine to output any finite sequence of symbols as it makes each transition (in other words, as it reads each symbol of its input). FSMs that associate outputs with transitions are called Mealy machines, after their inventor George Mealy. A ***Mealy machine M*** is a six-tuple $(K, \Sigma, O, \delta, s, A)$, where:

- K is a finite set of states,
- Σ is an input alphabet,
- O is an output alphabet,
- $s \in K$ is the start state,
- $A \subseteq$ is the set of accepting states, and
- δ is the transition function. It is a function from $(K \times \Sigma)$ to $(K \times O^*)$.

A Mealy machine M computes a function $f(w)$ iff, when it reads the input string w, its output sequence is $f(w)$.

Example: Generating Parity bits

The following Mealy machine adds an odd parity bit after every four binary digits that it reads. We will use the notation a/b on an arc to mean that the transition may be followed if the input character is a. If it is followed, then the string b will be generated.



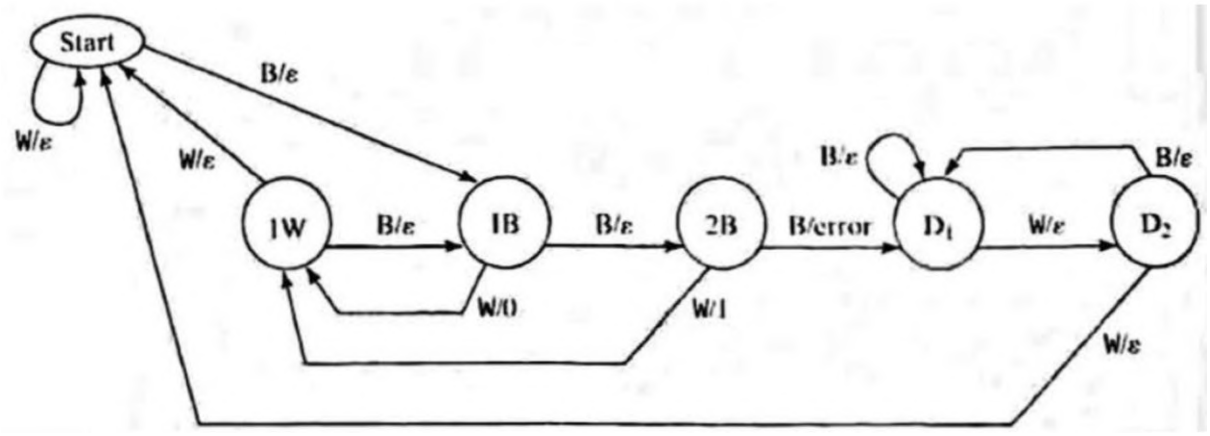
Example: Bar Code reader

Bar codes are ubiquitous. We consider here a Simplification: a bar code system that encodes just binary numbers. Imagine a bar code such as:

It is composed of columns, each of the same width. A column can he either white or black. If two black columns occur next to each other, it will look to us like a single-wide-black column, but the reader will see two adjacent black columns of the standard width. The job of the white columns is to delimit the black ones. A single black column encodes 0. A double black column encodes 1.

We can build a finite state transducer to read such a bar code and output a string of binary digits. We will represent a black bar with the symbol B and a white bar with the symbol W. The input to the transducer will be a sequence of those symbols corresponding to reading the bar code left to right. We'll assume that every correct bar code starts with a black column, so

white space ahead of the first black column is ignored. We will also assume that after every complete bar code there are at least two white columns. So, the reader should, at that point, reset to be ready to read the next code. If the reader sees three or more black columns in a row, it must indicate an error and stay in its error state until it is reset by seeing two while columns.



Interpreters for finite stale transducers can be built using techniques similar to the ones that we used to interpreters for finite state machines.

7. Bidirectional Transducers

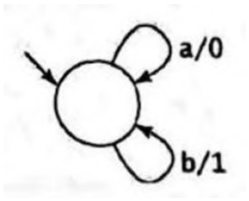
A process that reads an input string and constructs a corresponding output string can be described in a variety of different ways. Why should we choose the finite state transducer model? One reason is that it provides a declarative, rather than a procedural, way to describe the relationship between inputs and outputs, such a declarative model can then be run in two directions.

For example, to read an English text requires transforming a word like "liberties" into the root word "liberty" and the affix PLURAL. To generate an English text requires transforming a root word like "liberty" and the semantic marker “PLURAL” into the surface word "liberties". If we could specify, in a single declarative model, the relationship between surface words (the ones we see in text) and underlying root words and affixes, we could use it for either application.

The facts about English spelling rules and morphological analysis can be described with a bidirectional finite state transducer.

If we expand the definition of a Mealy machine to allow non-determinism, then any of these bidirectional processes can be represented. A nondeterministic Mealy machine can be thought of as defining a relation between one set of strings (for example, English surface words) and a second set of strings (for example. English underlying root words. along with affixes). It is possible that we will need a machine that is nondeterministic in one or both directions because the relationship between the two sets may not be able to be described as a function.

Example: When we define a regular language, it doesn't matter what alphabet we use. Anything that is true of a language L defined over the alphabet {a,b} will also be true of the language L' that contains exactly the strings in L except that every a has been replaced by a 0 and every b has been replaced by a L We can build a simple bidirectional transducer that can convert strings in L to strings in L' and vice-versa.



Of course. the real power of bidirectional finite state transducers comes from their ability to model more complex processes.