

AUTOMATA THEORY AND COMPUTABILITY

Module – 1

Why study the Theory of Computation, Languages and Strings:

The Theory of Computation are mathematical properties. The problems and of algorithms for solving problems depend on neither the details of today's technology nor the programming fashion i.e. implementations come and go. Instead they:

- Provides a set of abstract structures that are useful for solving certain classes of problems. These abstract structures can be implemented on whatever hardware/software platform is available.
- 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 of Theory of Computation will be to discover fundamental properties of the problems themselves:

- Is there any computational solution to the problem? If 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 the Theory

Some of the applications of Theory of Computation are:

- **Languages.** Enable both machine/machine and person/machine communication. Without them, none of today's applications of computing could exist.
- 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.
- People use natural languages, such as English, to communicate with each other. Since the advent of word processing, and then the Internet, we now type or speak our words to computers. So we would like to build programs to manage our words, check our grammar, search the World Wide Web, and translate from one language to another.
- Systems as diverse as parity checkers, vending machines, communication protocols, and building security devices can be straightforwardly described as finite state machines.
- Many interactive video games are (large, often nondeterministic) finite state machines.
- DNA is the language of life. DNA molecules, as well as the proteins that they describe, are strings that are made up of symbols drawn from small alphabets (nucleotides and amino acids, respectively). So computational biologists exploit many of the same tools that computational linguists use. For example, they rely on techniques that are based on both finite state machines and context-free grammars.

- Security is perhaps the most important property of many computer systems. The undecidability results show that there cannot exist a general purpose method for automatically verifying arbitrary security properties of programs. The complexity results serve as the basis for powerful encryption techniques.
- 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. The undecidability results that show that then: cannot exist a general theorem prover that can decide, given an arbitrary statement in first order logic, whether or not that statement follows from the system's axioms. The complexity results show that, if we back off to the far less expressive system of Boolean (propositional) logic, while it becomes possible to decide the validity of a given statement, it is not possible to do so, in general, in a reasonable amount of time.
- Clearly documented and widely accepted standards play a pivotal role in modern computing systems. Getting a diverse group of users to agree on a single standard is never easy. But the undecidability and complexity results mean that, for some important problems, there is no single right answer for all uses. Expressively weak standard languages may be tractable and decidable, but they may simply be inadequate for some tasks. For those tasks, expressively powerful languages, that give up some degree of tractability and possibly decidability, may be required. The provable lack of a one-size-fits-all language makes the standards process even more difficult and may require standards that allow alternatives.
- Many natural structures, including ones as different as organic molecules and computer networks, can be modeled as graphs. The theory of complexity that tells us that, while there exist efficient algorithms for answering some important questions about graphs, other questions are "hard", in the sense that no efficient algorithm for them is known nor is one likely to be developed.
- The complexity results are problems that matter yet for which no efficient algorithm is likely ever to be found. But practical solutions to some of these problems exist. They rely on a variety of approximation techniques that work pretty well most of the time.

Languages and Strings

- An alphabet is a non-empty, finite set of characters/symbols
 - ❖ Use Σ to denote an alphabet

Examples

$$\Sigma = \{ a, b \}$$

$$\Sigma = \{ 0, 1, 2 \}$$

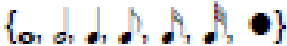

$$\Sigma = \{ a, b, c, \dots, z, A, B, \dots, Z \}$$

$$\Sigma = \{ \#, \$, *, @, \& \}$$

Strings

- A *string* is a finite sequence, possibly empty, of characters drawn from some alphabet Σ .
- ϵ is the empty string
- Σ^* is the set of all possible strings over an alphabet Σ .

Example Alphabets & Strings

<i>Alphabet name</i>	<i>Alphabet symbols</i>	<i>Example strings</i>
The lower case English alphabet	{a, b, c, ..., z}	ϵ , aabbcbg, aaaaa
The binary alphabet	{0, 1}	ϵ , 0, 001100, 11
A star alphabet	{★, ☆, ★, ☆, ☆, ☆}	ϵ , ☆☆, ☆☆☆☆☆
A music alphabet	{ 	ϵ , 

Functions on Strings

Length:

- $|s|$ is the length of string s
- $|s|$ is the number of characters in string s .

$$|\epsilon| = 0$$

$$|1001101| = 7$$

$\#_c(s)$ is defined as the number of times that c occurs in s .

$$\#_a(\text{abbaaa}) = 4.$$

Concatenation: the concatenation of 2 strings s and t is the string formed by appending t to s ; written as $s||t$ or more commonly, st

Example:

If $x = \text{good}$ and $y = \text{bye}$, then $xy = \text{goodbye}$
and $yx = \text{byegood}$

- Note that $|xy| = |x| + |y|$ -- Is it always??
- ϵ is the identity for concatenation of strings. So,

$$\forall x (x \epsilon = \epsilon x = x)$$

- Concatenation is associative. So,

$$\forall s, t, w ((st)w = s(tw))$$

Replication: For each string w and each natural number k , the string w^k is:

$$w^0 = \epsilon$$

$$w^{k+1} = w^k w$$

Examples:

$$a^3 = \text{aaa}$$

$$(\text{bye})^2 = \text{byebye}$$

$$a^0 b^3 = \text{bbb}$$

$$b^2 y^2 e^2 = ??$$

Natural Numbers $\{0, 1, 2, \dots\}$

Reverse: For each string w , w^R is defined as:

if $|w| = 0$ then $w^R = w = \varepsilon$

if $|w| = 1$ then $w^R = w$

if $|w| > 1$ then:

$\exists a \in \Sigma (\exists u \in \Sigma^* (w = ua))$

{ \exists There exist atleast one a }

So define $w^R = a u^R$

OR

if $|w| > 1$ then:

$\exists a \in \Sigma \ \& \ \exists u \in \Sigma^* \ \ni \ w = ua$

So define $w^R = a u^R$

Theorem: If w and x are strings, then $(wx)^R = x^R w^R$.

Example: $(\text{nametag})^R = (\text{tag})^R (\text{name})^R = \text{gateman}$

Proof: By induction on $|x|$:

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

$\forall n \geq 0 ((|x| = n) \rightarrow ((wx)^R = x^R w^R)) \rightarrow$

$((|x| = n + 1) \rightarrow ((wx)^R = x^R w^R))$:

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

Relations on Strings - Substrings

- **Substring:** string s is a *substring* of string t if s occurs contiguously in t
 - Every string is a substring of itself
 - ε is a substring of every string
- Proper Substring: s is a proper substring of t iff $s \neq t$
- Suppose $t = \text{aabbcc}$.
 - Substrings: $\varepsilon, a, aa, ab, bbcc, b, c, \text{aabbcc}$
 - Proper substrings?

The Prefix Relations

s is a *prefix* of t iff $\exists x \in \Sigma^* (t = sx)$.

s is a *proper prefix* of t iff s is a prefix of t and $s \neq t$.

Examples:

The *prefixes* of abba are: $\varepsilon, a, ab, abb, \text{abba}$.

The *proper prefixes* of abba are: ε, a, ab, abb .

- Every string is a prefix of itself.
- ε is a prefix of every string.

The Suffix Relations

s is a *suffix* of t iff $\exists x \in \Sigma^* (t = xs)$.

s is a *proper suffix* of t iff s is a suffix of t and $s \neq t$.

Examples:

The *suffixes* of abba are: ϵ , a, ba, bba, abba.

The *proper suffixes* of abba are: ϵ , a, ba, bba.

- Every string is a suffix of itself.
- ϵ is a suffix of every string.

Defining a Language

A *language* is a (finite or infinite) set of strings over a (finite) alphabet Σ .

Examples: Let $\Sigma = \{a, b\}$

Some languages over Σ :

$\emptyset = \{ \}$ // the empty language, no strings
 $\{\epsilon\}$ // language contains only the empty string
 $\{a, b\}$
 $\{\epsilon, a, aa, aaa, aaaa, aaaaa\}$

- Σ^* is defined as the set of all possible strings that can be formed from the alphabet Σ
 - Σ^* is a language
- Σ^* contains an *infinite* number of strings
 - Σ^* is *countably infinite*

Let $\Sigma = \{a, b\}$

$\Sigma^* = \{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, \dots\}$

Later, we will spend some more time studying Σ^* .

Defining Languages

defining a set Notation:

$L = \{ w \in \Sigma^* \mid \text{description of } w \}$

$L = \{ w \in \{a,b,c\}^* \mid \text{description of } w \}$

- “description of w ” can take many forms but must be precise
- Notation can vary, but must precisely define

$L = \{x \in \{a, b\}^* \mid \text{all } a\text{'s precede all } b\text{'s}\}$

- aab, aaabb, and aabbb are in L .
- aba, ba, and abc are not in L .
- What about ϵ , a, aa, and bb?

$L = \{x : \exists y \in \{a, b\}^* \mid x = ya\}$

- Give an English description.

Let $\Sigma = \{a, b\}$

- $L = \{ w \in \Sigma^* : |w| < 5 \}$
- $L = \{ w \in \Sigma^* \mid w \text{ begins with } b \}$
- $L = \{ w \in \Sigma^* \mid \#_b(w) = 2 \}$
- $L = \{ w \in \Sigma^* \mid \text{each } a \text{ is followed by exactly 2 } b\text{'s} \}$
- $L = \{ w \in \Sigma^* \mid w \text{ does not begin with } a \}$

The Perils of Using English

$L = \{x\#y : x, y \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}^* \text{ and, when } x \text{ \& } y \text{ are viewed as decimal representations of natural numbers, } \text{square}(x) = y\}$.

Examples:

3#9, 12#144

3#8, 12, 12#12#12

#

A Halting Problem Language

$L = \{w \mid w \text{ is a C++ program that halts on all inputs}\}$

- Well specified.
- Can we decide what strings it contains?
- Do we want a generator or recognizer?

Examples

What strings are in the following languages?

- $L = \{w \in \{a, b\}^* : \text{no prefix of } w \text{ contains } b\}$
- $L = \{w \in \{a, b\}^* : \text{no prefix of } w \text{ starts with } a\}$
- $L = \{w \in \{a, b\}^* : \text{every prefix of } w \text{ starts with } a\}$
- $L = \{a^n : n \geq 0\}$
- $L = \{ba^{2^n} : n \geq 0\}$
- $L = \{b^n a^n : n \geq 0\}$

Enumeration

Enumeration: to list all strings in a language (set)

- Arbitrary order
- More useful: **lexicographic order**
- Shortest first
- Within a length, dictionary order
- Define linear order of arbitrary symbols

Lexicographic Enumeration

$\{w \in \{a, b\}^* : |w| \text{ is even}\}$

$\{\epsilon, aa, ab, bb, aaaa, aaab, \dots\}$

What string is next?

How many strings of length 4?

How many strings of length 6?

Cardinality of a Language

- Cardinality of a Language: the number of strings in the language
- $|L|$
- Smallest language over any Σ is \emptyset , with cardinality 0.
- The largest is Σ^* .
 - Is this true?
 - How big is it?
- Can a language be uncountable?

Countably Infinite

Theorem: If $\Sigma \neq \emptyset$ then Σ^* is countably infinite.

Proof: The elements of Σ^* can be lexicographically enumerated by the following procedure:

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

This enumeration is infinite since there is no longest string in Σ^* . Since there exists an infinite enumeration of Σ^* , it is countably infinite.

How Many Languages Are There?

Theorem: If $\Sigma \neq \emptyset$ then the set of languages over Σ is uncountably infinite (uncountable).

Proof: The set of languages defined on Σ is $P(\Sigma^*)$. Σ^* is countably infinite. By Theorem A.4, if S is a countably infinite set, $P(S)$ is uncountably infinite. So $P(\Sigma^*)$ is uncountably infinite.

Functions on Languages

Set (Language) functions

Have the traditional meaning

- Union
- Intersection
- Complement
- Difference

Language functions

- Concatenation
- Kleene star

Concatenation of Languages

If L_1 and L_2 are languages over Σ :

$$L_1 L_2 = \{ w : \exists s \in L_1 \ \& \ \exists t \in L_2 \ni w = st \}$$

Examples:

$$L_1 = \{ \text{cat, dog} \}$$

$$L_2 = \{ \text{apple, pear} \}$$

$$L_1 L_2 = \{ \text{catapple, catpear, dogapple, dogpear} \}$$

$$L_2 L_1 = \{ \text{applecat, appledog, pearcat, peardog} \}$$

$\{\epsilon\}$ is the identity for concatenation:

$$L\{\epsilon\} = \{\epsilon\}L = L$$

\emptyset is a zero for concatenation:

$$L\emptyset = \emptyset L = \emptyset$$

Concatenating Languages Defined Using Variables

The scope of any variable used in an expression that invokes replication will be taken to be the entire expression.

$$L_1 = \{ a^n : n \geq 0 \}$$

$$L_2 = \{ b^n : n \geq 0 \}$$

$$L_1 L_2 = \{ a^n b^m : n, m \geq 0 \}$$

$$L_1 L_2 \neq \{ a^n b^n : n \geq 0 \}$$

Kleene Star *

L^* - language consisting of 0 or more concatenations of strings from L

$$L^* = \{ \epsilon \} \cup \{ w \in \Sigma^* : w = w_1 w_2 \dots w_k, k \geq 1 \ \& \ w_1, w_2, \dots w_k \in L \}$$

Examples:

$$L = \{ \text{dog, cat, fish} \}$$

$$L^* = \{ \epsilon, \text{dog, cat, fish, dogdog, dogcat, dogfish, fishcatfish, fishdogdogfishcat, ...} \}$$

$$L_1 = a^*$$

$$L_2 = b^*$$

What is a^*b^* ?

$$L_1 L_2 =$$

$$L_2 L_1 =$$

$$L_1 L_1 =$$

The + Operator

L^+ = language consisting of 1 or more concatenations of strings from L

$L^+ = L L^*$

$L^+ = L^* - \{\epsilon\}$ iff $\epsilon \notin L$

Explain this definition!!

When is $\epsilon \in L^+$?

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
- A set S is closed under the operation $@$ if for every element $x \in S$ & $y \in S$, $x@y \in S$

Semantics: Assigning Meaning to Strings

When is the meaning of a string important?

A semantic interpretation function assigns meanings to the strings of a language.

Can be very complex.

Example from English:

I am in class.

I am listening to class.

Uniqueness???

- Chocolate, please.
- I'd like chocolate.
- I'll have chocolate today.
- I guess I'll have chocolate.

They all have the same meaning!

Hand

- Give me a hand.
- I smashed my hand in the door.
- Please hand me that book.

These all have different meanings

Uniqueness in CS

- `int x = 4; x++;`
- `int x = 4; ++x;`
- `int x = 4; x = x + 1;`
- `int x = 4; x=x - -1;`
- `int x = 5;`

These all have the same result/meaning!

Semantic Interpretation Functions

For formal languages:

- Programming languages
- Network protocol languages
- Database query languages
- HTML
- BNF

For other kinds of “natural” languages:

- DNA

Other computing

- Genetic algorithm solutions
- Other problem solutions

The Big Picture

Examining Computational Problems

We want to examine a given computational problem and see how difficult it is.

- Then we need to compare problems
- Problems appear different

We want to cast them into the same kind of problem

- decision problems
- in particular, language recognition problem

Decision Problems

A decision problem is simply a problem for which the answer is yes or no (True or False). A decision procedure answers a decision problem.

Examples:

- ❖ Given an integer n , does n have a pair of consecutive integers as factors?
- ❖ The language recognition problem: Given a language L and a string w , is w in L ?

The Power of Encoding

- For problem already stated as decision problems.
- ❖ encode the inputs as strings and then define a language that contains exactly the set of inputs for which the desired answer is yes.
- For other problems, must first reformulate the problem as a decision problem, then encode it as a language recognition task

Everything is a String

Pattern matching on the web:

- Problem: Given a search string w and a web document d , do they match? In other words, should a search engine, on input w , consider returning d ?
- The language to be decided: $\{ \langle w, d \rangle : d \text{ is a candidate match for the query } w \}$

Does a program always halt?

- Problem: Given a program p , written in some standard programming language, is p guaranteed to halt on all inputs?
- The language to be decided:
HPALL = $\{ p : p \text{ halts on all inputs} \}$
Everything is a String

What If we're not working with strings?

Anything can be encoded as a string.

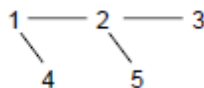
$\langle X \rangle$ is the string encoding of X .

$\langle X, Y \rangle$ is the string encoding of the pair X, Y .

Primality Testing

- Problem: Given a nonnegative integer n , is it prime?
- An instance of the problem: Is 9 prime?
- To encode the problem we need a way to encode each instance: We encode each nonnegative integer as a binary string.
- The language to be decided:
PRIMES = $\{ w : w \text{ is the binary encoding of a prime number} \}$.

- Problem: Given an undirected graph G , is it connected?
- Instance of the problem:



- Encoding of the problem: Let V be a set of binary numbers, one for each vertex in G . Then we construct $\langle G \rangle$ as follows:
 - Write $|V|$ as a binary number,
 - Write a list of edges,
- Separate all such binary numbers by “/”.
101/1/10/10/11/1/100/10/101
- The language to be decided: $\text{CONNECTED} = \{w \in \{0, 1, /\}^* : w = n_1/n_2/\dots/n_i, \text{ where each } n_i \text{ is a binary string and } w \text{ encodes a connected graph, as described above}\}$.

Turning Problems Into Decision Problems

Casting multiplication as decision:

- Problem: Given two nonnegative integers, compute their product.
- Reformulation: Transform computing into verification.
- The language to be decided:
 $L = \{w \text{ of the form: } \langle \text{integer1} \rangle x \langle \text{integer2} \rangle = \langle \text{integer3} \rangle, \text{ where:}$
 $\text{integer } n \text{ is any well formed integer, and } \text{integer3} = \text{integer1} * \text{integer2}\}$
 12x9=108
 12=12
 12x8=108

Casting sorting as decision:

- Problem: Given a list of integers, sort it.
- Reformulation: Transform the sorting problem into one of examining a pair of lists.
- The language to be decided:
 $L = \{w_1 \# w_2 : \exists n \geq 1 (w_1 \text{ is of the form } \langle \text{int}_1, \text{int}_2, \dots, \text{int}_n \rangle,$
 $w_2 \text{ is of the form } \langle \text{int}_1, \text{int}_2, \dots, \text{int}_n \rangle, \text{ and}$
 $w_2 \text{ contains the same objects as } w_1 \text{ and } w_2 \text{ is sorted})\}$
 Examples: 1,5,3,9,6#1,3,5,6,9
 Example of a string not in L : 1,5,3,9,6#1,2,3,4,5,6,7

Casting database querying as decision:

- Problem: Given a database and a query, execute the query.
- Reformulation: Transform the query execution problem into evaluating a reply for correctness.
- The language to be decided:

$L = \{d \# q \# a : d \text{ is an encoding of a database, } q \text{ is a string representing a query, and } a \text{ is the correct result of applying } q \text{ to } d\}$

Example:

(name, age, phone), (John, 23, 567-1234) (Mary, 24, 234-9876)#(select name
age=23)#
(John)

The Traditional Problems and their Language Formulations are Equivalent

By equivalent we mean that either problem can be reduced to the other. If we have a machine to solve one, we can use it to build a machine to do the other using just the starting machine and other functions that can be built using a machine of equal or lesser power.

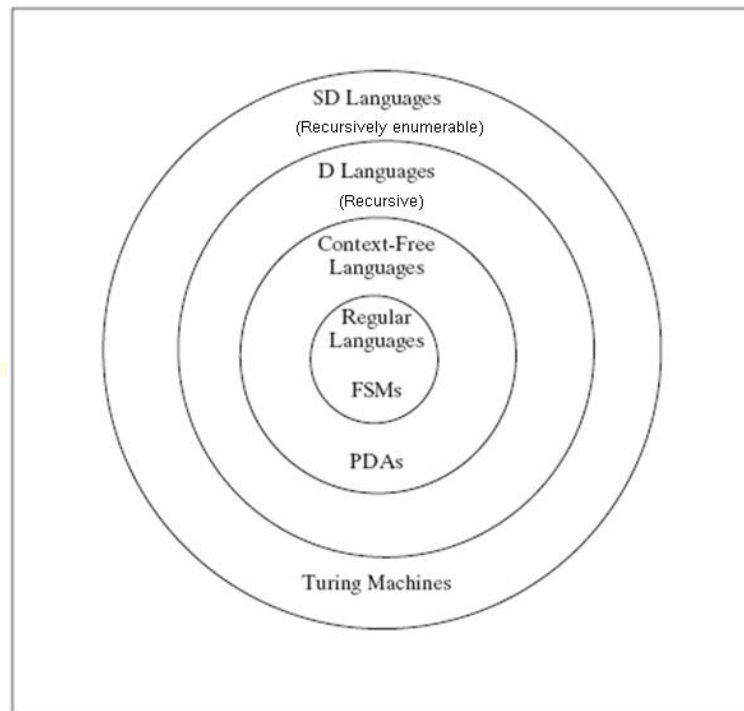
An Example

Consider the multiplication example:

$L = \{w \text{ of the form: } \langle \text{integer1} \rangle x \langle \text{integer2} \rangle = \langle \text{integer3} \rangle, \text{ where: integer } n \text{ is any well formed integer, and } \text{integer3} = \text{integer1} * \text{integer2}\}$

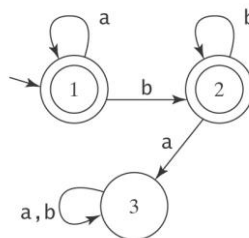
Given a multiplication machine, we can build the language recognition machine:
 Given the language recognition machine, we can build a multiplication machine:

Languages and Machines



Finite State Machines

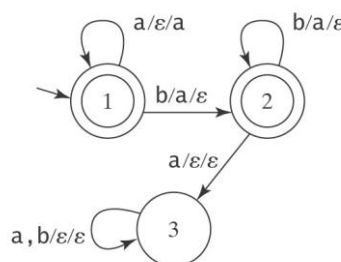
An FSM to accept a^*b^* :



- We call the class of languages acceptable by some FSM **regular**
 - There are simple useful languages that are not regular:
 - An FSM to accept $A^nB^n = \{a^n b^n : n \geq 0\}$
 - How can we compare numbers of a's and b's?
 - The only memory in an FSM is in the states and we must choose a fixed number of states in building it. But no bound on number of a's

Pushdown Automata

Build a PDA (roughly, FSM + a single stack) to accept $A^nB^n = \{a^n b^n : n \geq 0\}$



Bal, the language of balanced parentheses

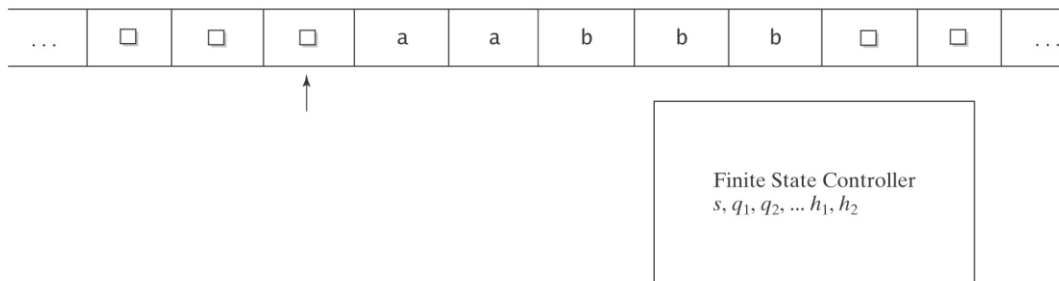
- contains strings like $(())$ or $()()$, but not $(())()$
- important, almost all programming languages allow parentheses, need checking
- PDA can do the trick, not FSM
- We call the class of languages acceptable by some PDA

context-free.

- There are useful languages not context free.
- $A^n B^n C^n = \{a^n b^n c^n : n \geq 0\}$
 - a stack wouldn't work. All popped out and get empty after counting b

Turing Machines

A Turing Machine to accept $A^n B^n C^n$



FSM and PDA (exists some equivalent PDA) are guaranteed to halt.

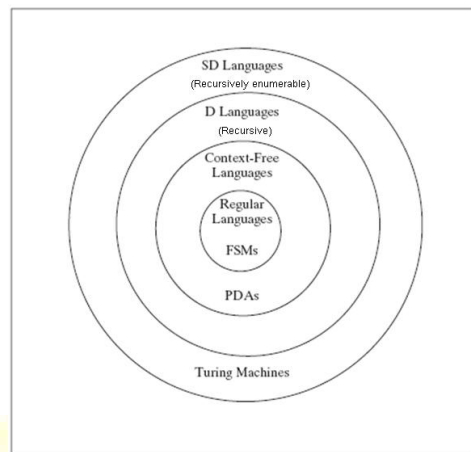
- But not TM. Now use TM to define new classes of languages, D and SD
- A language L is in D iff there exists a TM M that halts on all inputs, accepts all strings in L , and rejects all strings not in L .
 - in other words, M can always say yes or no properly
- A language L is in SD iff there exists a TM M that accepts all strings in L and fails to accept every string not in L . Given a string not in L , M may reject or it may loop forever (no answer).
- in other words, M can always say yes properly, but not no.
 - give up looking? say no?
- $D \subset SD$
- Bal, $A^n B^n$, $A^n B^n C^n$... are all in D
- how about regular and context-free languages?
- In SD but D : $H = \{ \langle M, w \rangle : \text{TM } M \text{ halts on input string } w \}$
- Not even in SD : $Hall = \{ \langle M \rangle : \text{TM } M \text{ halts on all inputs} \}$

Languages and Machines:

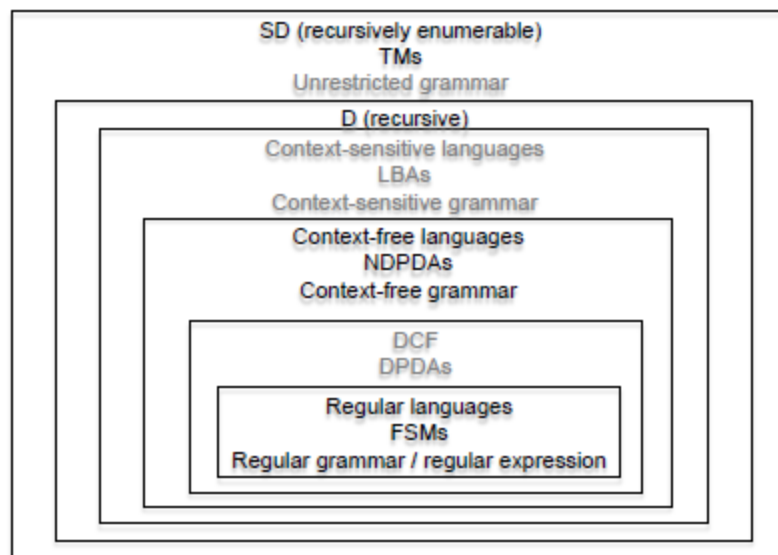
Hierarchy of language classes

Rule of Least Power: "Use the least powerful language suitable for expressing information, constraints or programs on the World Wide Web."

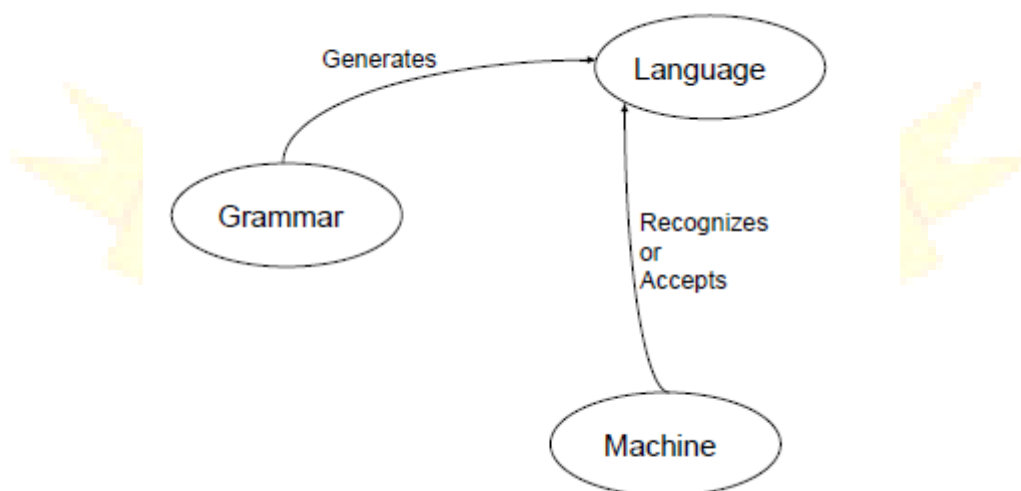
- Applies far more broadly.
- Expressiveness generally comes at a price
 - computational efficiency, decidability, clarity



Languages, Machines, and Grammars



Grammars, Languages, and Machines



A Tractability Hierarchy

- **P** : contains languages that can be decided by a TM in polynomial time
- **NP** : contains languages that can be decided by a nondeterministic TM (one can conduct a search by guessing which move to make) in polynomial time
- **PSPACE**: contains languages that can be decided by a machine with polynomial space

$$P \subseteq NP \subseteq PSPACE$$

- $P = NP$? Biggest open question for theorists

Decision Procedures

An **algorithm** is a detailed procedure that accomplishes some clearly specified task.

A **decision procedure** is an algorithm to solve a decision problem.

Decision procedures are programs and must possess two correctness properties:

- must halt on all inputs
- when it halts and returns an answer, it must be the correct answer for the given input

Decidability

- A decision problem is **decidable** iff there exists a decision procedure for it.
- A decision problem is **undecidable** iff there exists no a decision procedure for it.
- A decision problem is **semidecidable** iff there exists a semidecision procedure for it.
- a semidecision procedure is one that halts and returns *True* whenever *True* is the correct answer. When *False* is the answer, it may either halt and return *False* or it may loop (no answer).
- Three kinds of problems:
 - decidable (recursive)
 - not decidable but semidecidable (recursively enumerable)
 - not decidable and not even semidecidable
- **Note: Usually defined w.r.t. Turing machines**
 - most powerful formalism for algorithms
 - decidable = Turing-decidable

Decidable

Checking for even numbers: Is the integer x even?

Let / perform truncating integer division, then consider the following program:

$\text{even}(x:\text{integer}) = \text{If}(x/2)*2 = x \text{ then return } \text{True} \text{ else return } \text{False}$

Undecidable but Semidecidable

Halting Problem: For any Turing machine M and input w , decide whether M halts on w .

- w is finite
- $H = \{ \langle M, w \rangle : \text{TM } M \text{ halts on input string } w \}$
- asks whether M enters an infinite loop for a particular input w

Java version: Given an arbitrary Java program p that takes a string w as an input parameter. Does p halt on some particular value of w ?

$\text{haltsOnw}(p:\text{program}, w:\text{string}) =$

1. simulate the execution of p on w .
2. if the simulation halts return *True* else return *False*.

Not even Semidecidable

Halting-on-all (totality) Problem: For any Turing machine M , decide whether M halts on all inputs.

- $HALL = \{ \langle M \rangle : \text{TM } M \text{ halts on all inputs} \}$
- If it does, it computes a total function
- equivalent to the problem of whether a program can ever enter an infinite loop, for any input
- differs from the halting problem, which asks whether M enters an infinite loop for a particular input

Java version: Given an arbitrary Java program p that takes a single string as input parameter. Does p halt on all possible input values?

$\text{haltsOnAll}(p:\text{program}) =$

1. for $i = 1$ to infinity do:
 simulate the execution of p on all possible input strings of length i .
2. if all the simulations halt return *True* else return *False*.

Finite State Machines

A computational device whose input is a string, and whose output is one of the two values:

Accept and *Reject*. Also called FSA (finite state automata)

- Input string w is fed to M (an FSM) one symbol at a time, left to right
- Each time it receives a symbol, M considers its current state and the new symbol and chooses a next state
- One or more states maybe marked as accepting states
- Other stats are rejecting states
- If M runs out of input and is in an accepting state, it accepts
- Begin defining the class of FSMs whose behavior is deterministic.
- move is determined by current state and the next input character

A deterministic *FSM* (or *DFSM*) M is a quintuple

$M = (K, \Sigma, \delta, s, A)$, 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

K \times Σ to K .
 state input symbol state

Configurations of DFSMs

A *configuration* of a DFSM M is an element of: $K \times \Sigma^*$

It captures the two things that decide M 's future behavior:

- its current state
- the remaining, unprocessed input

The *initial configuration* of a DFSM M , on input w , is: (s, w)

The Yields Relations

The *yields-in-one-step* relation \vdash_M

$(q, w) \vdash_M (q', w')$ iff

- $w = a w'$ for some symbol $a \in \Sigma$, and
- $\delta(q, a) = q'$

The relation *yields*, \vdash_M^* , is the reflexive, transitive closure of \vdash_M

If $C_i \vdash_M^* C_j$, iff M can go from C_i to C_j in zero (due to "reflexive") or more (due to "transitive") steps.

Notation: \vdash and \vdash^* are also used.

Path

A *path* by M is a **maximal** sequence of configurations $C_0, C_1, C_2 \dots$ such that:

- C_0 is an initial configuration,
- $C_0 \vdash_M C_1 \vdash_M C_2 \vdash_M \dots$

- In other words, a path is just a sequence of steps from the start configuration going as far as possible
- A path ends when it enters an **accepting configuration**, or it has no where to go (no transition defined for C_n)
- This definition of path is applicable to all machines (FSM, PDA, TM, deterministic or nondeterministic).
- A path P can be infinite. For FSM, DPDA, or NDFSM and NDPDA without ϵ - transitions, P always ends. For NDFSM and NDPDA with ϵ -transitions, P can be infinite. For TM (deterministic or nondeterministic), P can be infinite.
- For deterministic machines, there is only one path (ends or not).
- A path accepts w if it ends at an accepting configuration
 - Accepting configuration varies for different machines
- A path rejects w if it ends at a non-accepting configuration

Accepting and Rejecting

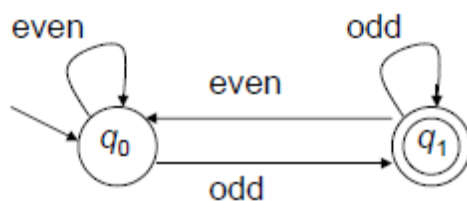
A DFSM M **accepts** a string w iff the path accepts it.

- i.e., $(s, w) \vdash_M^* (q, \epsilon)$, for some $q \in A$.
 - For DFSM, (q, ϵ) where $q \in A$ is an **accepting configuration**
- A DFSM M **rejects** a string w iff the path rejects it.
- **The** path, because there is only one.

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

Theorem: Every DFSM M , on input w , halts in at most $|w|$ steps.

Accepting Example



On input 235, the configurations are:

$(q_0, 235) \vdash_M (q_0, 35)$
 \vdash_M
 \vdash_M
 \vdash_M

Thus $(q_0, 235) \vdash_M^* (q_1, \epsilon)$

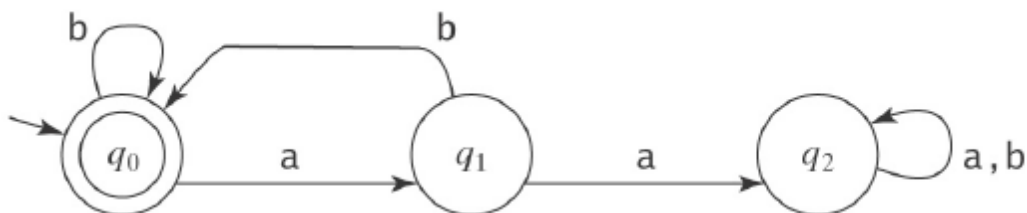
- If M is a DFSM and $\epsilon \in L(M)$, what simple property must be true of M ?
 - The start state of M must be an accepting state

Regular Languages

A language is **regular** iff it is accepted by some FSM.

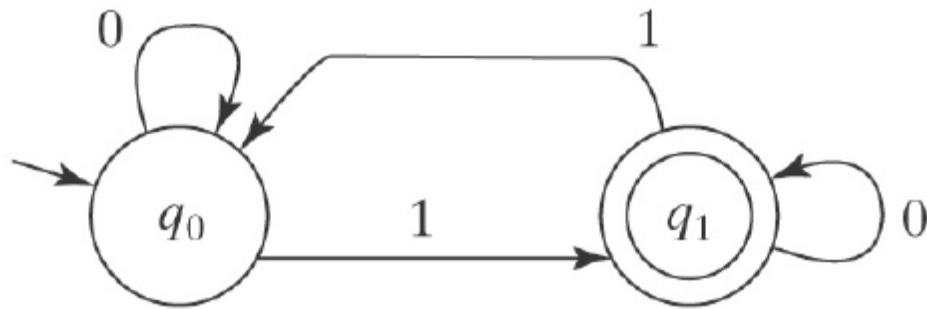
Example:

$L = \{w \in \{a, b\}^* : \text{every } a \text{ is immediately followed by a } b\}.$



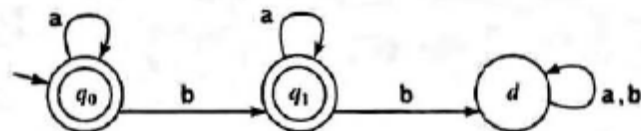
$L = \{w \in \{0, 1\}^* : w \text{ has odd parity}\}$.

A binary string has odd parity iff the number of 1's is odd



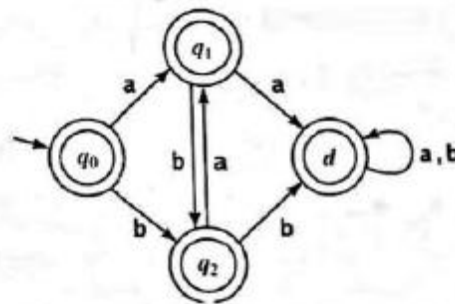
No More Than One b

$L = \{w \in \{a, b\}^* : w \text{ contains no more than one } b\}$.



Checking Consecutive Characters

$L = \{w \in \{a, b\}^* : \text{no two consecutive characters are the same}\}$.



Floating Point Numbers

Let $\text{FLOAT} = \{w : w \text{ is the string representation of a floating point number}\}$.

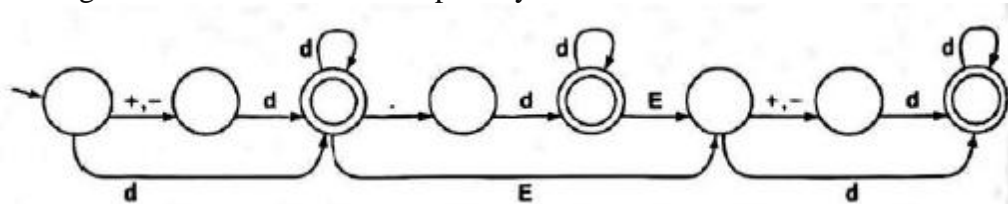
Assume the following syntax for floating point numbers:

- A floating point number is an optional sign, followed by a decimal number, followed by an optional exponent.
- A decimal number may be of the form x or $x.y$, where x and y are nonempty strings of decimal digits.
- An exponent begins with E and is followed by an optional sign and then an integer.
- An integer is a nonempty string of decimal digits.

So, for example, these strings represent floating point numbers:

+3.0, 3.0, 0.3E1, 0.3E+1, -0.3E+1, -3E8

FLOAT is regular because it can be accepted by the DFSM:



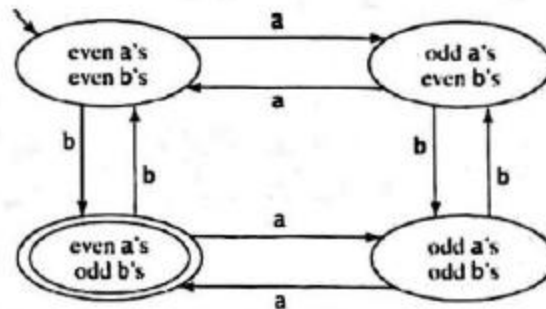
In this diagram, we have used the shorthand d to stand for any one of the decimal digits (0 - 9). And we have omitted the dead state to avoid arrows crossing over each other.

Designing Deterministic Finite State Machines

- 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?
- 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. The smallest DFSM for any language L is the one that has exactly one state for every group of initial substrings that share a common fate.

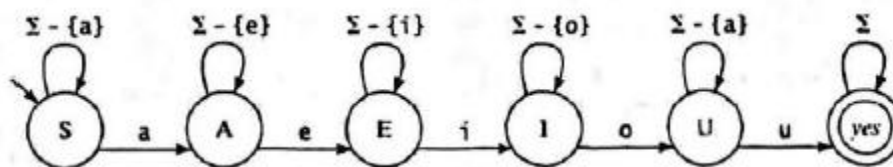
Even a's, Odd b's

Let $L = \{w \in \{a, b\}^* : w \text{ contains an even number of a's and an odd number of b's}\}$.



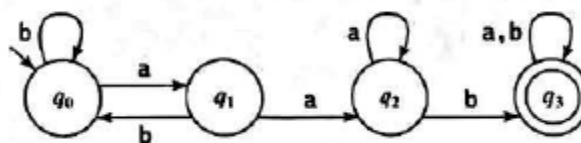
All the Vowels in Alphabetical Order

Let $L = \{w \in \{a-z\}^* : \text{all five vowels, a, e, i, o, and u, occur in } w \text{ in alphabetical order}\}$.



A Substring that Doesn't Occur

Let $L = \{w \in \{a, b\}^* : w \text{ does not contain the substring } aab\}$.



It is straightforward to design an FSM that looks for the substring aab . So we can begin building a machine to accept L by building the following machine to accept $\neg L$. Then we can convert this machine into one that accepts L by making states q_0 , q_1 , and q_2 accepting and state q_3 nonaccepting.

The Missing Letter Language

Let $\Sigma = \{a, b, c, d\}$.

Let $L_{\text{Missing}} = \{w : \text{there is a symbol } a_i \in \Sigma \text{ not appearing in } w\}$.

Try to make a DFSM for L_{Missing}

- Doable, but complicated. Consider the number of accepting states
 - all missing (1)
 - 3 missing (4)
 - 2 missing (6)
 - 1 missing (4)

Nondeterministic FSMs

• In the theory of computation, a **nondeterministic finite state machine** or **nondeterministic finite automaton (NFA)** is a finite state machine where for each pair of state and input symbol there may be several possible next states.

- This distinguishes it from the deterministic finite automaton (DFA), where the next possible state is uniquely determined.
- Although the DFA and NFA have distinct definitions, it may be shown in the formal theory that they are equivalent, in that, for any given NFA, one may construct an equivalent DFA, and vice-versa
- Both types of automata recognize only regular languages.
- Nondeterministic machines are a key concept in computational complexity theory, particularly with the description of complexity classes P and NP .

Definition of an NDFSM

$M = (K, \Sigma, \Delta, s, A)$, 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 relation. It is a finite subset of $(K \times (\Sigma * \{\epsilon\})) \times K$

NDFSM and DFSM

Δ is the transition relation. It is a finite subset of

$(K \times (\Sigma * \{\epsilon\})) \times K$

Recall the definition of DFSM:

$M = (K, \Sigma, \delta, s, A)$, 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

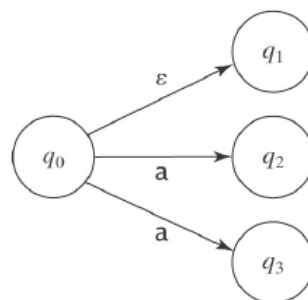
$\Delta: (K \times (\Sigma * \{\epsilon\})) \times K$

$\delta: (K \times \Sigma)$ to K

Key difference:

- In every configuration, a DFSM can make exactly one move; this is not true for NDFSM
- M may enter a config. from which two or more competing moves are possible. This is due to (1) ϵ -transition (2) relation, not function

Sources of Nondeterminism

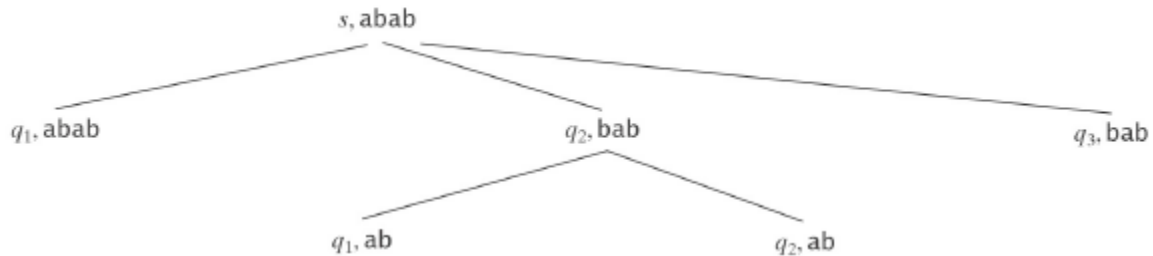


- Nondeterminism is a generalization of determinism
- Every DFSM is automatically an NDFSM

- Can be viewed as a kind of parallel computation
 - Multiple independent threads run concurrently

Envisioning the operation of M

- Explore a search tree (depth-first):
 - Each node corresponds to a configuration of M
 - Each path from the root corresponds to the *path* we have defined



- Alternatively, imagine following all paths through M in parallel (breadth-first):
 - Explain later in “Analyzing Nondeterministic FSMs”

Accepting

Recall: a path is a maximal sequence of steps from the start configuration.

M accepts a string w iff there exists *some path* that accepts it.

- Same as DFSM, (q, ϵ) where $q \in A$ is an **accepting configuration**

M halts upon acceptance.

Other paths may:

- Read all the input and halt in a nonaccepting state,
- Reach a dead end where no more input can be read.
- Loop forever and never finish reading the input

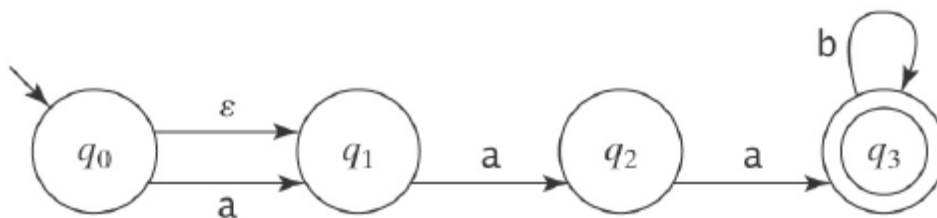
The language accepted by M , denoted $L(M)$, is the set of all strings accepted by M .

M rejects a string w iff all paths reject it.

- It is possible that, on input $w \notin L(M)$, M neither accepts nor rejects. In that case, no path accepts and some path does not reject.

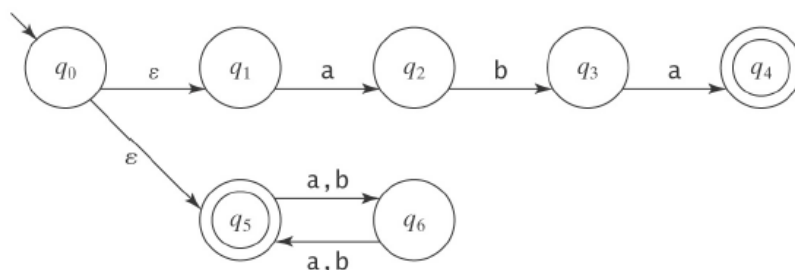
Optional Initial a

$L = \{w \in \{a, b\}^* : w \text{ is made up of an optional } a \text{ followed by } aa \text{ followed by zero or more } b\text{'s}\}.$



Two Different Sublanguages

$L = \{w \in \{a, b\}^* : w = aba \text{ or } |w| \text{ is even}\}.$

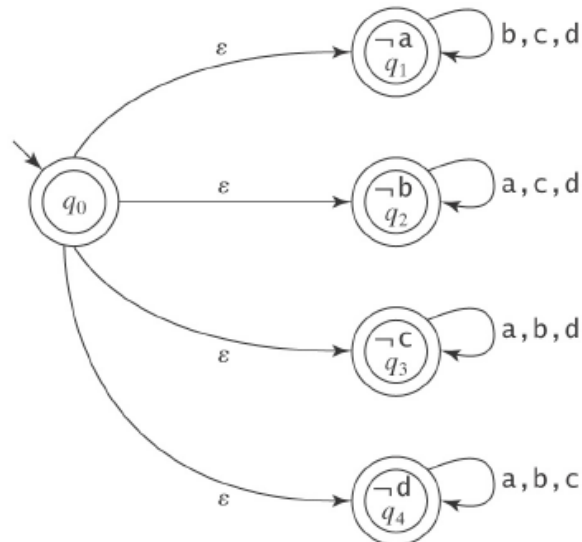


Why NDFSM?

- High level tool for describing complex systems
- Can be used as the basis for constructing efficient practical DFSMs
 - Build a simple NDFSM
 - Convert it to an equivalent DFSM
 - Minimize the result

The Missing Letter Language

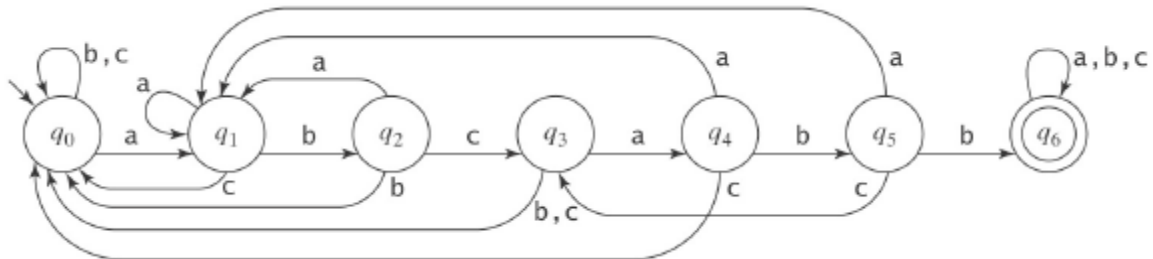
Let $\Sigma = \{a, b, c, d\}$. Let $L_{Missing} = \{w : \text{there is a symbol } ai \in \Sigma \text{ not appearing in } w\}$



Pattern Matching

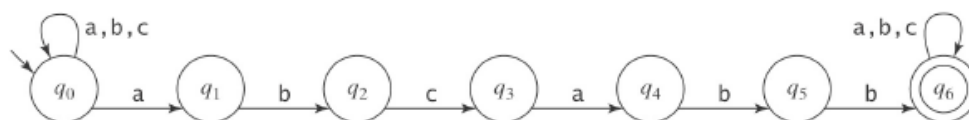
$L = \{w \in \{a, b, c\}^* : \exists x, y \in \{a, b, c\}^* (w = x \text{abcabb} y)\}$.

A DFSM:



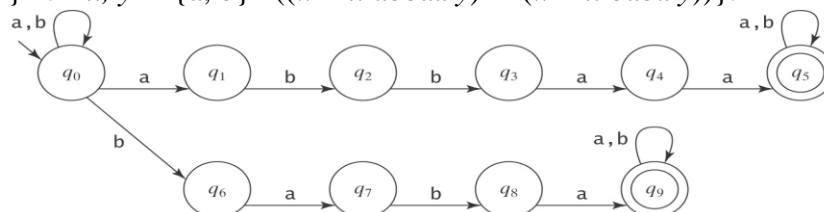
Works, but complex to design, error prone

An NDFSM:



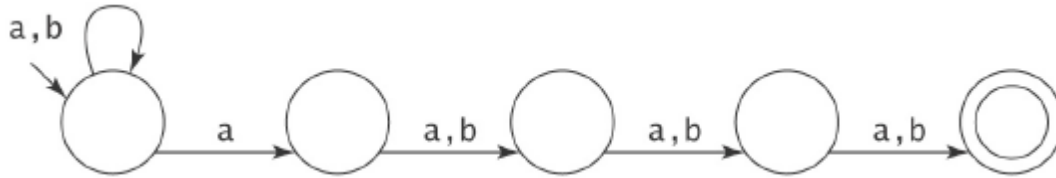
Multiple Keywords

$L = \{w \in \{a, b\}^* : \exists x, y \in \{a, b\}^* ((w = x \text{abbaa} y) \vee (w = x \text{baba} y))\}$.



Checking from the End

$L = \{w \in \{a, b\}^* : \text{the fourth to the last character is } a\}$

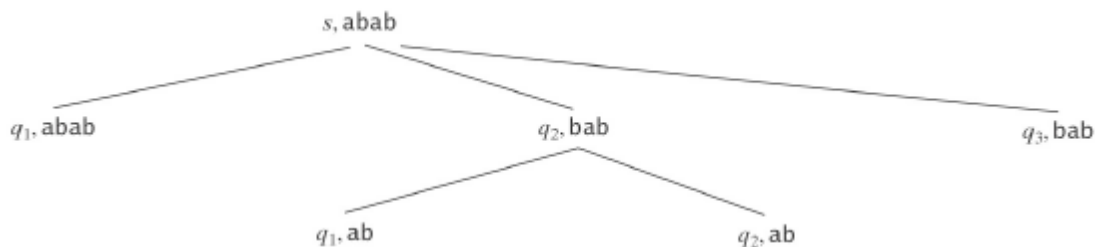


Analyzing Nondeterministic FSMs

Given an NDFSM M , how can we analyze it to determine if it accepts a given string?

Two approaches:

- Depth-first explore a search tree:



- Follow all paths in parallel (breadth-first)

Nondeterministic and Deterministic FSMs

Clearly: $\{\text{Languages accepted by a DFMS}\} \subseteq \{\text{Languages accepted by an NDFSM}\}$

Theorem:

For each DFMS M , there is an equivalent NDFSM M' .

- $L(M') = L(M)$

Theorem:

For each NDFSM, there is an equivalent DFMS.

Theorem: For each NDFSM, there is an equivalent DFMS.

Proof: By construction:

Given an NDFSM $M = (K, \Sigma, \Delta, s, A)$,

we construct $M' = (K', \Sigma, \delta', s', A')$, where

$K' = P(K)$

$s' = \text{eps}(s)$

$A' = \{Q \subseteq K : Q \cap A \neq \emptyset\}$

$\delta'(Q, a) = \bigcup \{\text{eps}(p) : p \in K \text{ and } (q, a, p) \in \Delta \text{ for some } q \in Q\}$

An Algorithm for Constructing the Deterministic FSM

1. Compute the $\text{eps}(q)$'s.
2. Compute $s' = \text{eps}(s)$.
3. Compute δ' .
4. Compute $K' =$ a subset of $P(K)$.
5. Compute $A' = \{Q \in K' : Q \cap A \neq \emptyset\}$.

$\text{ndfsmto fsm}(M: \text{NDFSM}) =$

1. For each state q in KM do:
 - 1.1 Compute $\text{eps}(q)$.
2. $s' = \text{eps}(s)$
3. Compute δ' :
 - 3.1 $\text{active-states} = \{s'\}$.
 - 3.2 $\delta' = \emptyset$.

3.3 While there exists some element Q of *active-states* for which δ' has not yet been computed do:

For each character c in ΣM do:

$new_state = \phi$.

For each state q in Q do:

For each state p such that $(q, c, p) \in \Delta$ do:

$new_state = new_state \cup eps(p)$.

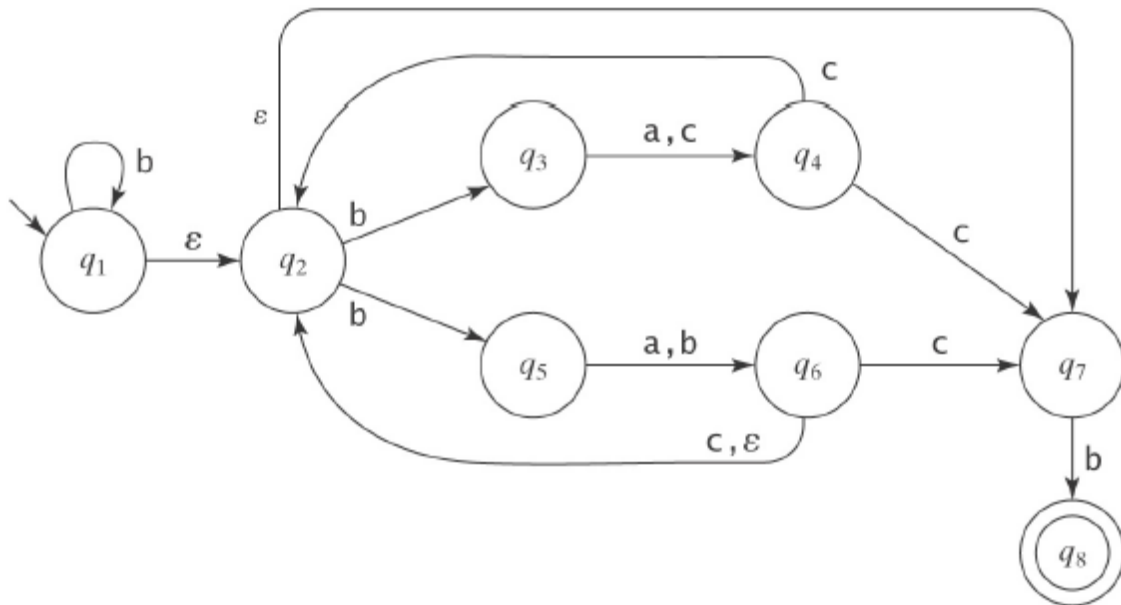
Add the transition (Q, c, new_state) to δ' .

If $new_state \notin active_states$ then insert it.

4. $K' = active_states$.

5. $A' = \{Q \in K' : Q \cap A \neq \phi\}$.

Example:



1. Compute $eps(q)$ for each state q in K_M :

$eps(q_1) = \{q_1, q_2, q_7\}$, $eps(q_2) = \{q_2, q_7\}$, $eps(q_3) = \{q_3\}$, $eps(q_4) = \{q_4\}$,
 $eps(q_5) = \{q_5\}$, $eps(q_6) = \{q_2, q_6, q_7\}$, $eps(q_7) = \{q_7\}$, $eps(q_8) = \{q_8\}$.

2. $s' = eps(s) = \{q_1, q_2, q_7\}$.

3. Compute δ' :

$active_states = \{\{q_1, q_2, q_7\}\}$. Consider $\{q_1, q_2, q_7\}$:

$((\{q_1, q_2, q_7\}, a), \emptyset)$.

$((\{q_1, q_2, q_7\}, b), \{q_1, q_2, q_3, q_5, q_7, q_8\})$.

$((\{q_1, q_2, q_7\}, c), \emptyset)$.

$active_states = \{\{q_1, q_2, q_7\}, \emptyset, \{q_1, q_2, q_3, q_5, q_7, q_8\}\}$. Consider \emptyset :

$((\emptyset, a), \emptyset)$. /* \emptyset is a dead state and we will generally omit it.

$((\emptyset, b), \emptyset)$.

$((\emptyset, c), \emptyset)$.

$active_states = \{\{q_1, q_2, q_7\}, \emptyset, \{q_1, q_2, q_3, q_5, q_7, q_8\}\}$. Consider

$\{q_1, q_2, q_3, q_5, q_7, q_8\}$:

$((\{q_1, q_2, q_3, q_5, q_7, q_8\}, a), \{q_2, q_4, q_6, q_7\})$.

$((\{q_1, q_2, q_3, q_5, q_7, q_8\}, b), \{q_1, q_2, q_3, q_5, q_6, q_7, q_8\})$.

$((\{q_1, q_2, q_3, q_5, q_7, q_8\}, c), \{q_4\})$.

$active-states = \{\{q_1, q_2, q_7\}, \emptyset, \{q_1, q_2, q_3, q_5, q_7, q_8\}, \{q_2, q_4, q_6, q_7\}, \{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}, \{q_4\}\}$. Consider $\{q_2, q_4, q_6, q_7\}$:

$((\{q_2, q_4, q_6, q_7\}, a), \emptyset)$.

$((\{q_2, q_4, q_6, q_7\}, b), \{q_3, q_5, q_8\})$.

$((\{q_2, q_4, q_6, q_7\}, c), \{q_2, q_7\})$.

$active-states = \{\{q_1, q_2, q_7\}, \emptyset, \{q_1, q_2, q_3, q_5, q_7, q_8\}, \{q_2, q_4, q_6, q_7\}, \{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}, \{q_4\}, \{q_3, q_5, q_8\}, \{q_2, q_7\}\}$.

Consider $\{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}$:

$((\{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}, a), \{q_2, q_4, q_6, q_7\})$.

$((\{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}, b), \{q_1, q_2, q_3, q_5, q_6, q_7, q_8\})$.

$((\{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}, c), \{q_2, q_4, q_7\})$.

$active-states = \{\{q_1, q_2, q_7\}, \emptyset, \{q_1, q_2, q_3, q_5, q_7, q_8\}, \{q_2, q_4, q_6, q_7\}, \{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}, \{q_4\}, \{q_3, q_5, q_8\}, \{q_2, q_7\}, \{q_2, q_4, q_7\}\}$. Consider $\{q_4\}$:

$((\{q_4\}, a), \emptyset)$.

$((\{q_4\}, b), \emptyset)$.

$((\{q_4\}, c), \{q_2, q_7\})$.

$active-states$ did not change. Consider $\{q_3, q_5, q_8\}$:

$((\{q_3, q_5, q_8\}, a), \{q_2, q_4, q_6, q_7\})$.

$((\{q_3, q_5, q_8\}, b), \{q_2, q_6, q_7\})$.

$((\{q_3, q_5, q_8\}, c), \{q_4\})$.

$active-states = \{\{q_1, q_2, q_7\}, \emptyset, \{q_1, q_2, q_3, q_5, q_7, q_8\}, \{q_2, q_4, q_6, q_7\},$

$\{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}, \{q_4\}, \{q_3, q_5, q_8\}, \{q_2, q_7\}, \{q_2, q_4, q_7\}, \{q_2, q_6, q_7\}\}$.

Consider $\{q_2, q_7\}$:

$((\{q_2, q_7\}, a), \emptyset)$.

$((\{q_2, q_7\}, b), \{q_3, q_5, q_8\})$.

$((\{q_2, q_7\}, c), \emptyset)$.

$active-states$ did not change. Consider $\{q_2, q_4, q_7\}$:

$((\{q_2, q_4, q_7\}, a), \emptyset)$.

$((\{q_2, q_4, q_7\}, b), \{q_3, q_5, q_8\})$.

$((\{q_2, q_4, q_7\}, c), \{q_2, q_7\})$.

$active-states$ did not change. Consider $\{q_2, q_6, q_7\}$:

$((\{q_2, q_6, q_7\}, a), \emptyset)$.

$((\{q_2, q_6, q_7\}, b), \{q_3, q_5, q_8\})$.

$((\{q_2, q_6, q_7\}, c), \{q_2, q_7\})$.

active-states did not change. δ has been computed for each element of active-states.

$$4. K' = \{\{q_1, q_2, q_7\}, \emptyset, \{q_1, q_2, q_3, q_5, q_7, q_8\}, \{q_2, q_4, q_6, q_7\}, \{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}, \{q_4\}, \{q_3, q_5, q_8\}, \{q_2, q_7\}, \{q_2, q_4, q_7\}, \{q_2, q_6, q_7\}\}.$$

$$5. A' = \{\{q_1, q_2, q_3, q_5, q_7, q_8\}, \{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}, \{q_3, q_5, q_8\}\}.$$

From FSMs to Operational Systems

An FSM is an abstraction. FSM describe that solves a problem without worrying about many kinds of implementation details. In fact, we don't even need to know whether it will be etched into silicon or implemented in software. FSMs for real problems can be turned into operational systems in any of a number of ways:

- An FSM can be translated into a circuit design and implemented directly in hardware. For example, it makes sense to implement the parity checking FSM of in hardware.
- An FSM can be simulated by a general purpose interpreter. We will describe designs for such interpreters in the next section. Sometimes all that is required is a simulation. In other cases, a simulation can be used to check a design before it is translated into hardware.
- An FSM can be used as a specification for some critical aspect of the behavior of a complex system. The specification can then be implemented in software just as any specification might be. And the correctness of the implementation can be shown by verifying that the implementation satisfies the specification (i.e. .. that it matches the FSM).

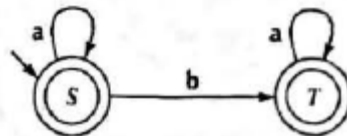
Simulators for FSMs:

Once we have created an FSM to solve a problem, we may want to simulate its execution.

Simulating Deterministic FSMs

Hardcoding a Deterministic FSM

Consider the following deterministic FSM M that accepts the language $L = \{ w \in \{a, b\}^* : w \text{ contains no more than one } b \}$.



We could view M as a specification for the following program:

Until accept or reject do:

```

S:   s = get-next-symbol.
      If s = end-of-file then accept.
      Else if s = a then go to S.
      Else if s = b then go to T.
T:   s = get-next-symbol.
      If s = end-of-file then accept.
      Else if s = a then go to T.
      Else if s = b then reject.
  
```

End.

Interpreter for a deterministic FSM $M = (K, \Sigma, \delta, s, A)$:

dfssimulate(M : DFMSM, w : string) =

1. $st = s$.
2. Repeat:
 - 2.1. $c = \text{get-next-symbol}(w)$.
 - 2.2. If $c \neq \text{end-of-file}$ then:

- 2.2.1 $st = \delta(st, c)$.
 until $c = \text{end-of-file}$.
 3. If $st \in A$ then accept else reject.

Simulating Nondeterministic FSMs

To execute an NDFSM M . One solution is:

$ndfsmconvertandsimulate(M: \text{NDFSM}) = dfsmsimulate(nfdstodfsm(M))$.

We give here a more detailed description of $ndfmsimulate$, which simulates an NDFSM $M = (K, \Sigma, \Delta, s, A)$ running on an input string w :

$ndfmsimulate(M: \text{NDFSM}, w: \text{string}) =$

1. Declare the set st . /* st will hold the current state (a set of states from K). */
2. Declare the set $st1$. /* $st1$ will be built to contain the next state. */
3. $st = \text{eps}(s)$. /* Start in all states reachable from s via only ϵ -transitions. */
4. Repeat:

$c = \text{get-next-symbol}(w)$.
 If $c \neq \text{end-of-file}$ then do:

$st1 = \emptyset$.
 For all $q \in st$ do:

For all $r: (q, c, r) \in \Delta$ do:

$st1 = st1 \cup \text{eps}(r)$.

/* Follow paths from all states M is currently in. /* Find all states reachable from q via a transition labeled c . /* Follow all ϵ -transitions from there. /* Done following all paths. So st becomes M 's new state. /* If all paths have died, quit.

If $st = \emptyset$ then exit.

until $c = \text{end-of-file}$.

5. If $st \cap A \neq \emptyset$ then accept else reject.

Minimizing FSMs

DFSM M is *minimal* iff there is no other DFSM M' such that $L(M) = L(M')$ and M' has fewer states than M does.

Building a Minimal DFSM for a Language

The x and y are *indistinguishable* with respect to L , which we will write as $x \approx_L y$ iff:

$\forall z \in \Sigma^*$ (either both xz and $yz \in L$ or neither is).

In other words, \approx_L is a relation that is defined so that $x \approx_L y$ precisely in case, if x and y are viewed as prefixes of some longer string, no matter what continuation string z comes next, either both xz and yz are in L or both are not.

Example:

If $L = \{a\}^*$, then $a \approx_L aa \approx_L aaa$. But if $L = \{w \in \{a, b\}^* : |w| \text{ is even}\}$, then $a \approx_L aaa$, but it is not the case that $a \approx_L aa$ because, if $z = a$, we have $aa \in L$ but $aaa \notin L$.

We will say that x and y are *distinguishable* with respect to L , iff they are not indistinguishable. So, if x and y are distinguishable, then there exists at least one string z such that one but not both of xz and yz is in L .

Note that \approx_L is an equivalence relation because it is:

- Reflexive: $\forall x \in \Sigma^* (x \approx_L x)$, because $\forall x, z \in \Sigma^* (xz \in L \leftrightarrow xz \in L)$.
- Symmetric: $\forall x, y \in \Sigma^* (x \approx_L y \rightarrow y \approx_L x)$, because $\forall x, y, z \in \Sigma^* ((xz \in L \leftrightarrow yz \in L) \leftrightarrow (yz \in L \leftrightarrow xz \in L))$.
- Transitive: $\forall x, y, z \in \Sigma^* (((x \approx_L y) \wedge (y \approx_L w)) \rightarrow (x \approx_L w))$, because:
 $\forall x, y, z \in \Sigma^* (((xz \in L \leftrightarrow yz \in L) \wedge (yz \in L \leftrightarrow wz \in L)) \rightarrow (xz \in L \leftrightarrow wz \in L))$.

\approx_L Imposes a Lower Bound on the Minimum Number of States of a DFSM for L

Theorem: Let L be a regular language and let $M = (K, \Sigma, \delta, s, A)$ be a DFSM that accepts L . The number of states in M is greater than or equal to the number of equivalence classes of \approx_L .

Proof: Suppose that the number of states in M were less than the number of equivalence classes of \approx_L . Then, by the pigeonhole principle, there must be at least one state q that contains strings from at least two equivalence classes of \approx_L . But then M 's future behavior on those strings will be identical, which is not consistent with the fact that they are in different equivalence classes of \approx_L .

Minimizing an Existing DFSM

$\text{minDFSM}(M: \text{DFSM}) =$

1. $\text{classes} = \{A, K-A\}$. /* Initially, just two classes of states, accepting and rejecting.
2. Repeat until a pass at which no change to classes has been made:
 - 2.1. $\text{newclasses} = \emptyset$. /* At each pass, we build a new set of classes, splitting the old ones as necessary. Then this new set becomes the old set, and the process is repeated.
 - 2.2. For each equivalence class e in classes , if e contains more than one state, see if it needs to be split:

For each state q in e do: /* Look at each state and build a table of what it does. Then the tables for all states in the class can be compared to see if there are any differences that force splitting.

For each character c in Σ do:

Determine which element of classes q goes to if c is read.

If there are any two states p and q such that there is any character c such that, when c is read, p goes to one element of classes and q goes to another, then p and q must be split. Create as many new equivalence classes as are necessary so that no state remains in the same class with a state whose behavior differs from its. Insert those classes into newclasses .

If there are no states whose behavior differs, no splitting is necessary. Insert e into newclasses .

2.3. $classes = newclasses$.

/* The states of the minimal machine will correspond exactly to the elements of $classes$ at this point. We use the notation $[q]$ for the element of $classes$ that contains the original state q .

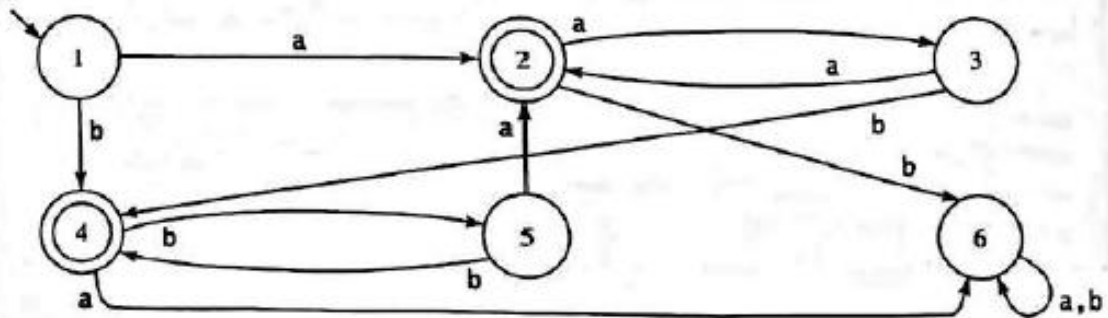
3. Return $M' = (classes, \Sigma, \delta, [s_M], \{[q] : \text{the elements of } q \text{ are in } A_M\})$, where $\delta_{M'}$ is constructed as follows:

if $\delta_M(q, c) = p$, then $\delta_{M'}([q], c) = [p]$.

Example:

Using *minDFSM* to Find a Minimal Machine

Let $\Sigma = \{a, b\}$. Let $M =$



We will show the operation of *minDFSM* at each step:

Initially, $classes = \{[2, 4], [1, 3, 5, 6]\}$.

At step 1:

$((2, a), [1, 3, 5, 6])$	$((4, a), [1, 3, 5, 6])$	No splitting required here.
$((2, b), [1, 3, 5, 6])$	$((4, b), [1, 3, 5, 6])$	

$((1, a), [2, 4])$	$((3, a), [2, 4])$	$((5, a), [2, 4])$	$((6, a), [1, 3, 5, 6])$
$((1, b), [2, 4])$	$((3, b), [2, 4])$	$((5, b), [2, 4])$	$((6, b), [1, 3, 5, 6])$

There are two different patterns, so we must split into two classes, $[1, 3, 5]$ and $[6]$. Note that, although $[6]$ has the same behavior as $[2, 4]$ after reading a single character, it cannot be combined with $[2, 4]$ because they do not share behavior after reading no characters.

$Classes = \{[2, 4], [1, 3, 5], [6]\}$.

At step 2:

$((2, a), [1, 3, 5])$	$((4, a), [6])$	These two must be split.
$((2, b), [6])$	$((4, b), [1, 3, 5])$	

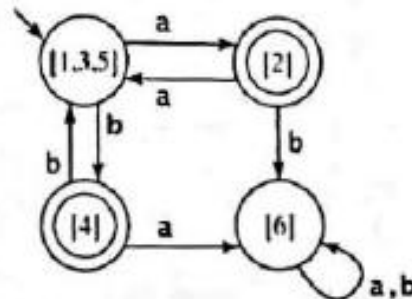
$((1, a), [2, 4])$	$((3, a), [2, 4])$	$((5, a), [2, 4])$	No splitting required.
$((1, b), [2, 4])$	$((3, b), [2, 4])$	$((5, b), [2, 4])$	

$Classes = \{[2], [4], [1, 3, 5], [6]\}$.

At step 3:

$((1, a), [2])$	$((3, a), [2])$	$((5, a), [2])$	No splitting required.
$((1, b), [4])$	$((3, b), [4])$	$((5, b), [4])$	

So *minDFSM* returns $M' =$



A Canonical Form for Regular Languages

A *canonical form* for some set of objects C assigns exactly one representation to each class of "equivalent" objects in C . Further, each such representation is distinct, so two objects in C share the same representation iff they are "equivalent" in the sense for which we define the form.

buildFSMcanonicalform(M : FSM) =

1. $M' = ndfsmtofdsm(M)$.
2. $M\# = minDFSM(M')$.
3. Create a unique assignment of names to the states of $M\#$ as follows:

3.1. Call the start state q_0 .

3.2. Define an order on the elements of Σ .

3.3 Until all states have been named do:

Select the lowest numbered named state that has not yet been selected. Call it q .

Create an ordered list of the transitions out of q by the order imposed on their labels.

Create an ordered list of the as yet unnamed states that those transitions enter by doing the following: If the first transition is (q, c_1, p_1) , then put p_1 first. If the second transition is (q, c_2, p_2) and p_2 is not already on the list, put it next. If it is already on the list, skip it. Continue until all transitions have been considered. Remove from the list any states that have already been named.

Name the states on the list that was just created: Assign to the first one the name q_k , where k is the smallest index that hasn't yet been used. Assign the next name to the next state and so forth until all have been named.

4. Return $M\#$.

Given two FSMs M_1 and M_2 , *buildFSMcanonicalform*(M_1) = *buildFSMcanonicalform*(M_2) iff $L(M_1) = L(M_2)$.

one important use for this canonical form; It provides the basis for a simple way to test whether an FSM accepts any strings or whether two FSMs are equivalent.

Finite State Transducers

Many finite state transducers are loops that simply run forever, processing inputs.

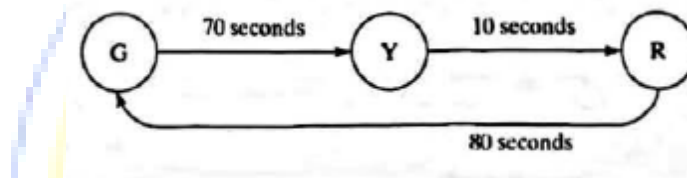
One simple kind of finite state transducer associates an output with each state of a machine M . That output is generated whenever M enters the associated state. Deterministic finite state transducers of this sort are called Moore machines, after their inventor Edward Moore. A **Moore machine** M is 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:

A Typical United States Traffic Light



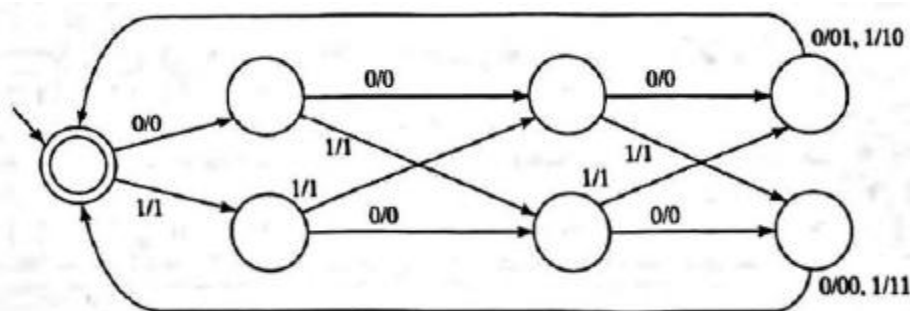
A different definition for a deterministic finite state transducer permits each machine to output any finite sequence of symbols as it makes each transition (in other words, as it reads each symbol of its input). FSMs that associate outputs with transitions are called Mealy machines, after their inventor 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,
- $A \subseteq K$ 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 $f(w)$.

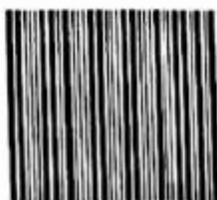
Generating Parity Bits

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.



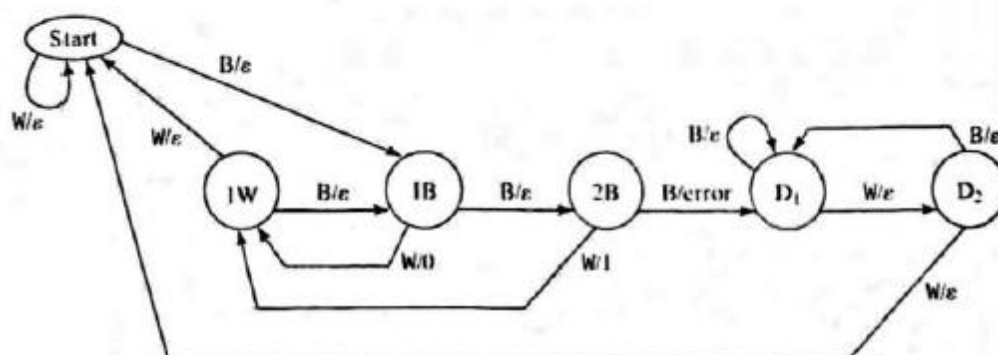
EXAMPLE 5.31 A Bar Code Reader

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 be 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'll 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'll 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 white columns.



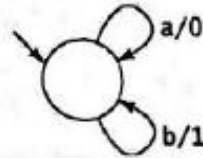
Bidirectional Transducers

Translators need to run in both directions. For example, consider translating between Roman numerals and Arabic ones.

If we expand the definition of a Mealy machine to allow nondeterminism, 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 and a second set of strings. 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 5.32 Letter Substitution

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 1 . 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.

