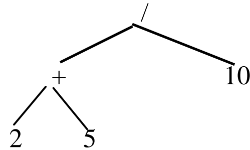
The Three Hour Tour Through Automata Theory

Read Supplementary Materials: The Three Hour Tour Through Automata Theory
Read Supplementary Materials: Review of Mathematical Concepts
Read K & S Chapter 1
Do Homework 1.

Let's Look at Some Problems

```
int alpha, beta;
alpha = 3;
beta = (2 + 5) / 10;
```

(1) **Lexical analysis**: Scan the program and break it up into variable names, numbers, etc.
(2) **Parsing**: Create a tree that corresponds to the sequence of operations that should be executed, e.g.,



(3) **Optimization**: Realize that we can skip the first assignment since the value is never used and that we can precompute the arithmetic expression, since it contains only constants.
(4) **Termination**: Decide whether the program is guaranteed to halt.
(5) **Interpretation**: Figure out what (if anything) it does.

A Framework for Analyzing Problems

We need a single framework in which we can analyze a very diverse set of problems.
The framework we will use is **Language Recognition**

A ***language*** is a (possibly infinite) set of finite length strings over a finite alphabet.

Languages

(1) $\Sigma = \{0,1,2,3,4,5,6,7,8,9\}$

L $= \{w \in \Sigma^*: \text{w represents an odd integer}\}$
$= \{w \in \Sigma^*: \text{the last character of w is 1,3,5,7, or 9}\}$
$= (0\cup1\cup2\cup3\cup4\cup5\cup6\cup7\cup8\cup9)^*$
$(1\cup3\cup5\cup7\cup9)$

(2) $\Sigma = \{(,)\}$

L $= \{w \in \Sigma^*: \text{w has matched parentheses}\}$
= the set of strings accepted by the grammar:

$S \rightarrow (S)$
$S \rightarrow SS$
$S \rightarrow \varepsilon$

(3) L = {w: w is a sentence in English}

Examples: Mary hit the ball.
Colorless green ideas sleep furiously.
The window needs fixed.

(4) L = {w: w is a C program that halts on all inputs}

Encoding Output in the Input String

(5) Encoding multiplication as a single input string

L = {w of the form: <integer>x<integer>=<integer>, where <integer> is any well formed integer, and the third integer is the product of the first two}

12x9=108 12=12 12x8=108

(6) Encoding prime decomposition

L = {w of the form: <integer1>/<integer2>,<integer3> …, where integers 2 - n represent the prime decomposition of integer 1.

15/3,5 2/2

More Languages

(7) Sorting as a language recognition task:

L = {w_1 # w_2: $\exists n \geq 1$,

w_1 is of the form int_1, int_2, … int_n,

w_2 is of the form int_1, int_2, … int_n, and

w_2 contains the same objects as w_1 and w_2 is sorted}

Examples:

1,5,3,9,6#1,3,5,6,9 $\in$ L

1,5,3,9,6#1,2,3,4,5,6,7 $\notin$ L

(8) Database querying as a language recognition task:

L = {d # q # a:

d is an encoding of a database,

q is a string representing a query, and

a is the correct result of applying q to d}

Example:

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

The Traditional Problems and their Language Formulations are Equivalent

By equivalent we mean:

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.

Consider the multiplication example:

L = {w of the form:

<integer>x<integer>=<integer>, where

<integer> is any well formed integer, and

the third integer is the product of the first two}

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

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

A Framework for Describing Languages

Clearly, if we are going to work with languages, each one must have a finite description.

Finite Languages: Easy. Just list the elements of the language.

L = {June, July, August}

Infinite Languages: Need a finite description.

Grammars let us use recursion to do this.

Grammars 1

(1) The Language of Matched Parentheses

S → (S)
S → SS
S → ε

(2) The Language of Odd Integers

S → 1
S → 3
S → 5
S → 7
S → 9
S → 0 S
S → 1 S
S → 2 S
S → 3 S
S → 4 S
S → 5 S
S → 6 S
S → 7 S
S → 8 S
S → 9 S

Grammars 2

S → O
S → A O
A →A D
A → D
D → O
D → E
O → 1
O → 3
O → 5
O → 7
O → 9
E→ 0
E→ 2
E→ 4
E→ 6
E→ 8

Grammars 3

(3) The Language of Simple Arithmetic Expressions

S → <exp>
<exp> → <number>
<exp> → (<exp>)
<exp> → - <exp>
<exp> → <exp> <op> <exp>
<op> → + | - | * | /
<number> → <digit>
<number> → <digit> <number>
<digit > → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Grammars as Generators and Acceptors

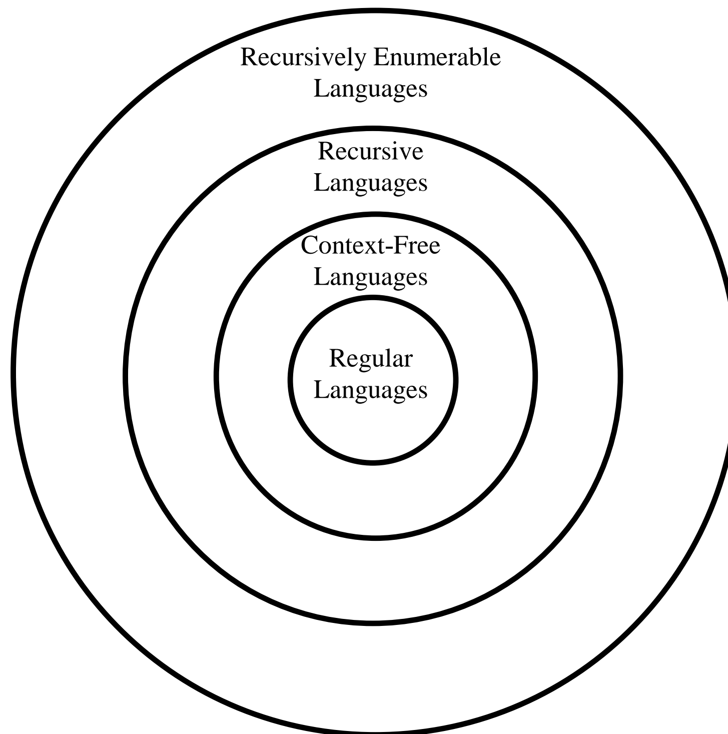
Top Down Parsing

4 + 3

Bottom Up Parsing

4 + 3

The Language Hierarchy



Regular Grammars

In a regular grammar, all rules must be of the form:

<one nonterminal> → <one terminal> or ε

or

<one nonterminal> → <one terminal><one nonterminal>

So, the following rules are okay:

S → ε
S → a
S → aS

But these are not:

S → ab
S → SS
aS → b

Regular Expressions and Languages

Regular expressions are formed from ∅ and the characters in the target alphabet, plus the operations of:

- Concatenation: $\alpha\beta$ means α followed by β
- Or (Set Union): $\alpha \cup \beta$ means α Or (Union) β
- Kleene *: α^* means 0 or more occurrences of α concatenated together.
- At Least 1: α^+ means 1 or more occurrences of α concatenated together.
- (): used to group the other operators

Examples:

(1) Odd integers:
$(0\cup1\cup2\cup3\cup4\cup5\cup6\cup7\cup8\cup9)^*(1\cup3\cup5\cup7\cup9)$

(2) Identifiers:
$(A\text{-}Z)^+((A\text{-}Z) \cup (0\text{-}9))^*$

(3) Matched Parentheses

Context Free Grammars

(1) The Language of Matched Parentheses

S → (S)
S → SS
S → ε

(2) The Language of Simple Arithmetic Expressions

S → <exp>
<exp> → <number>
<exp> → (<exp>)
<exp> → - <exp>
<exp> → <exp> <op> <exp>
<op> → + | - | * | /
<number> → <digit>
<number> → <digit> <number>
<digit > → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Not All Languages are Context-Free

English:

$S \rightarrow NP\ VP$
$NP \rightarrow the\ NP1 \mid NP1$
$NP1 \rightarrow ADJ\ NP1 \mid N$
$N \rightarrow boy \mid boys$
$VP \rightarrow V \mid V\ NP$
$V \rightarrow run \mid runs$

What about "boys runs"

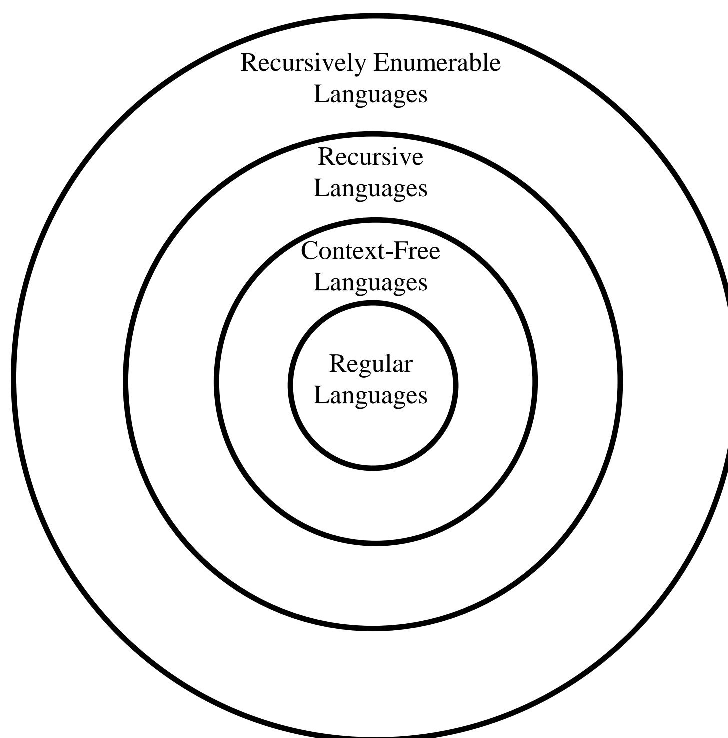
A much simpler example: $a^nb^nc^n$, $n \geq 1$

Unrestricted Grammars

Example: A grammar to generate all strings of the form $a^nb^nc^n$, $n \geq 1$

$S \rightarrow aBSc$
$S \rightarrow aBc$
$Ba \rightarrow aB$
$Bc \rightarrow bc$
$Bb \rightarrow bb$

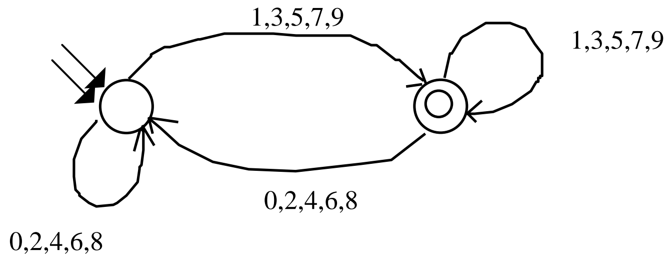
The Language Hierarchy



A Machine Hierarchy

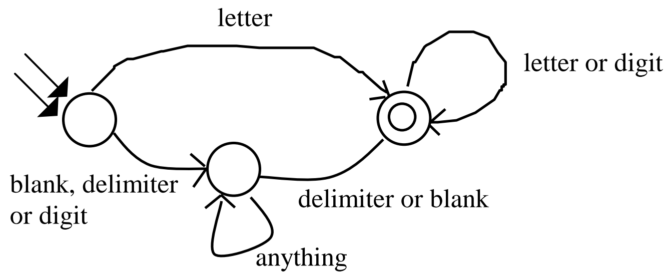
Finite State Machines 1

An FSM to accept odd integers:



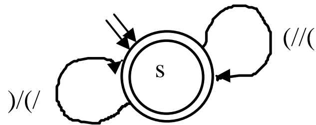
Finite State Machines 2

An FSM to accept identifiers:



Pushdown Automata

A PDA to accept strings with balanced parentheses:

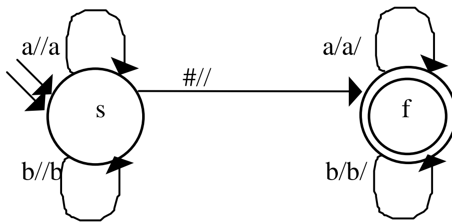


Example: (())()

Stack:

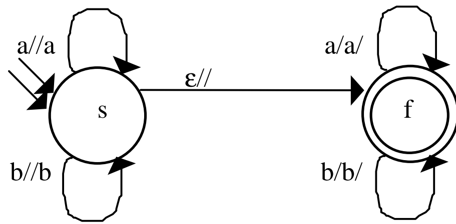
Pushdown Automaton 2

A PDA to accept strings of the form $w\#w^R$:



A Nondeterministic PDA

A PDA to accept strings of the form ww^R

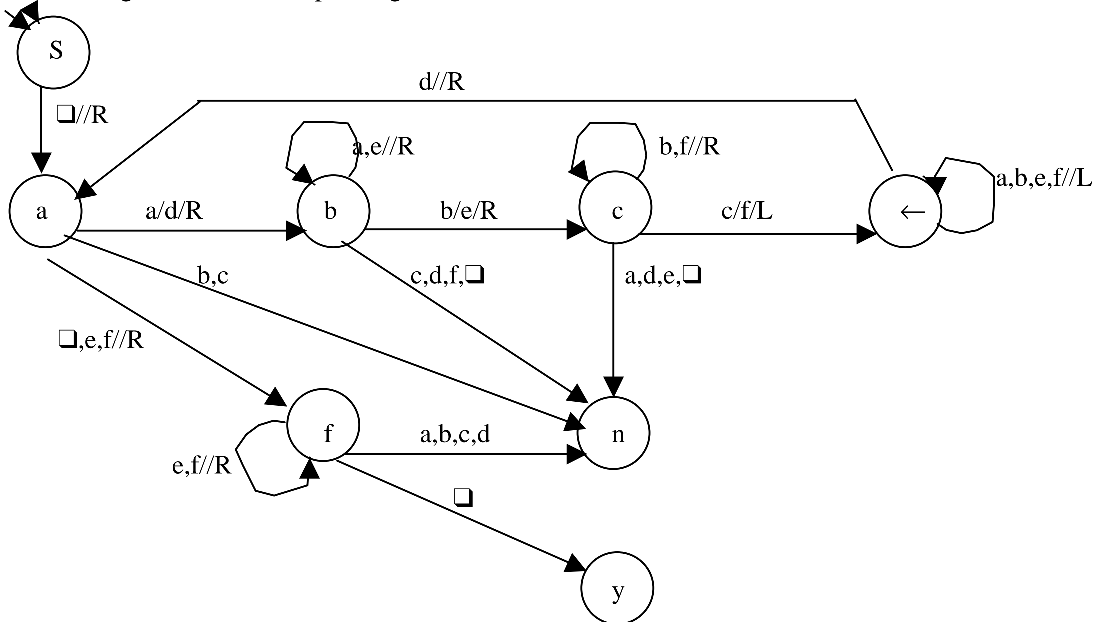


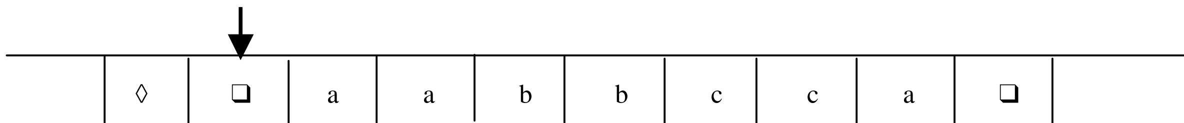
PDA 3

A PDA to accept strings of the form $a^nb^nc^n$

Turing Machines

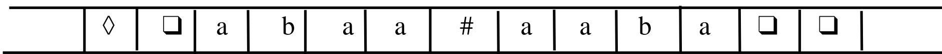
A Turing Machine to accept strings of the form $a^nb^nc^n$



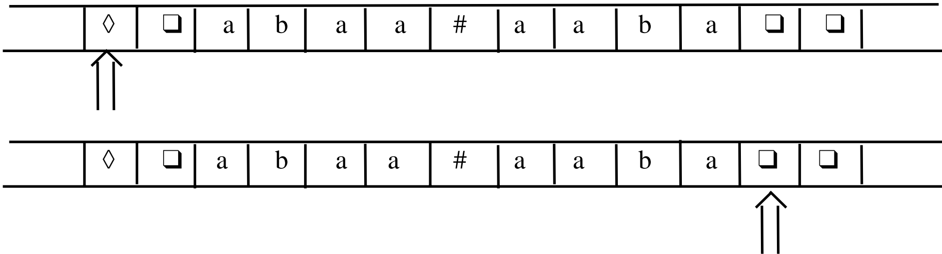


A Two Tape Turing Machine

A Turing Machine to accept $\{w\#w^R\}$



A Two Tape Turing Machine to do the same thing



Simulating k Tapes with One

A multitrack tape:

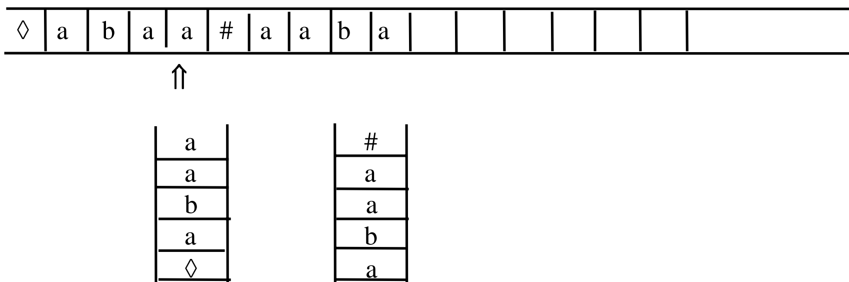
◊	◊	❑	a	b	a	❑	❑	❑	❑
	0	0	1	0	0	0	0		
	◊	a	b	b	a	b	a		
	0	1	0	0	0	0	0		

Can be encoded on a single tape with an alphabet consisting of symbols corresponding to :

{{◊,a,b,#,❑} x {0,1} x
{◊,a,b,#,❑} x {0,1}}

Example: 2nd square: (❑,0,a,1))

Simulating a Turing Machine with a PDA with Two Stacks



The Universal Turing Machine
Encoding States, Symbols, and Transitions

Suppose the input machine M has 5 states, 4 tape symbols, and a transition of the form:

(s,a,q,b), which can be read as:

in state s, reading an a, go to state q, and write b.

We encode this transition as:

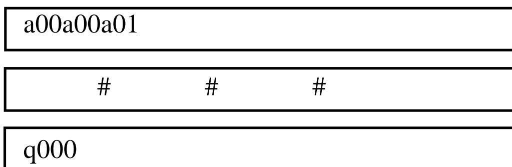
q000,a00,q010,a01

A series of transitions that describe an entire machine will look like

q000,a00,q010,a01#q010,a00,q000,a00

The Universal Turing Machine

a a b



Church's Thesis
(Church-Turing Thesis)

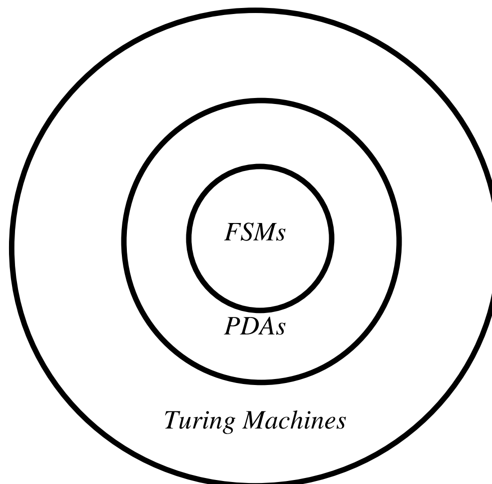
An algorithm is a formal procedure that halts.

The Thesis: Anything that can be computed by any algorithm can be computed by a Turing machine.

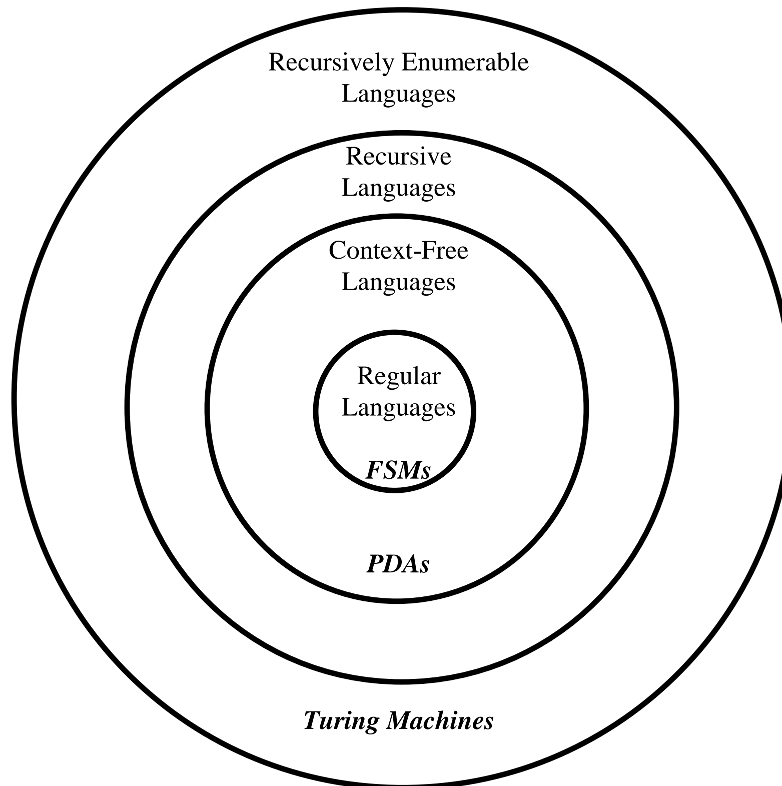
Another way to state it: All "reasonable" formal models of computation are equivalent to the Turing machine. This isn't a formal statement, so we can't prove it. But many different computational models have been proposed and they all turn out to be equivalent.

Example: unrestricted grammars

A Machine Hierarchy



Languages and Machines



Where Does a Particular Problem Go?

Showing what it is -- generally by construction of:

- A grammar, or a machine

Showing what it isn't -- generally by contradiction, using:

- Counting
  - Example: $a^n b^n$
- Closure properties
- Diagonalization
- Reduction

Closure Properties

Regular Lanugages are Closed Under:

- Union
- Concatenation
- Kleene closure
- Complementation
- Reversal
- Intersection

Context Free Languages are Closed Under:

- Union
- Concatenation
- Kleene Closure
- Reversal
- Intersection with regular languages

Etc.

Using Closure Properties

Example:
$L = \{a^n b^m c^p : n \neq m \text{ or } m \neq p\}$ is not deterministic context-free.

Two theorems we'll prove later:

Theorem 3.7.1: The class of deterministic context-free languages is closed under complement.

Theorem 3.5.2: The intersection of a context-free language with a regular language is a context-free language.

If L were a deterministic CFL, then the complement of L (L') would be a deterministic CFL.

But $L' \cap a^*b^*c^* = \{a^n b^n c^n\}$, which we know is not context-free, much less deterministic context-free. Thus a contradiction.

Diagonalization

The power set of the integers is not countable.
Imagine that there were some enumeration:

	1	2	3	4	5
Set 1	1				
Set 2		1		1	
Set 3	1		1		
Set 4		1			
Set 5	1	1	1	1	1

But then we could create a new set

New Set				1	

But this new set must necessarily be different from all the other sets in the supposedly complete enumeration. Yet it should be included. Thus a contradiction.

More on Cantor

Of course, if we're going to enumerate, we probably want to do it very systematically, e.g.,

	1	2	3	4	5	6	7
Set 1	1						
Set 2		1					
Set 3	1	1					
Set 4			1				
Set 5	1		1				
Set 6		1	1				
Set 7	1	1	1				

Read the rows as bit vectors, but read them backwards. So Set 4 is 100. Notice that this is the binary encoding of 4.
This enumeration will generate all **finite** sets of integers, and in fact the set of all finite sets of integers is countable.
But when will it generate the set that contains all the integers except 1?

The Unsolvability of the Halting Problem

Suppose we could implement

HALTS(M,x)

M: string representing a Turing Machine

x: string representing the input for M

If M(x) halts then True

else False

Then we could define

TROUBLE(x)

x: string

If HALTS(x,x) then loop forever

else halt

So now what happens if we invoke TROUBLE(TROUBLE), which invokes HALTS(TROUBLE,TROUBLE)

If HALTS says that TROUBLE halts on itself then TROUBLE loops. IF HALTS says that TROUBLE loops, then TROUBLE halts.

Viewing the Halting Problem as Diagonalization

First we need an enumeration of the set of all Turing Machines. We'll just use lexicographic order of the encodings we used as inputs to the Universal Turing Machine. So now, what we claim is that HALTS can compute the following table, where 1 means the machine halts on the input:

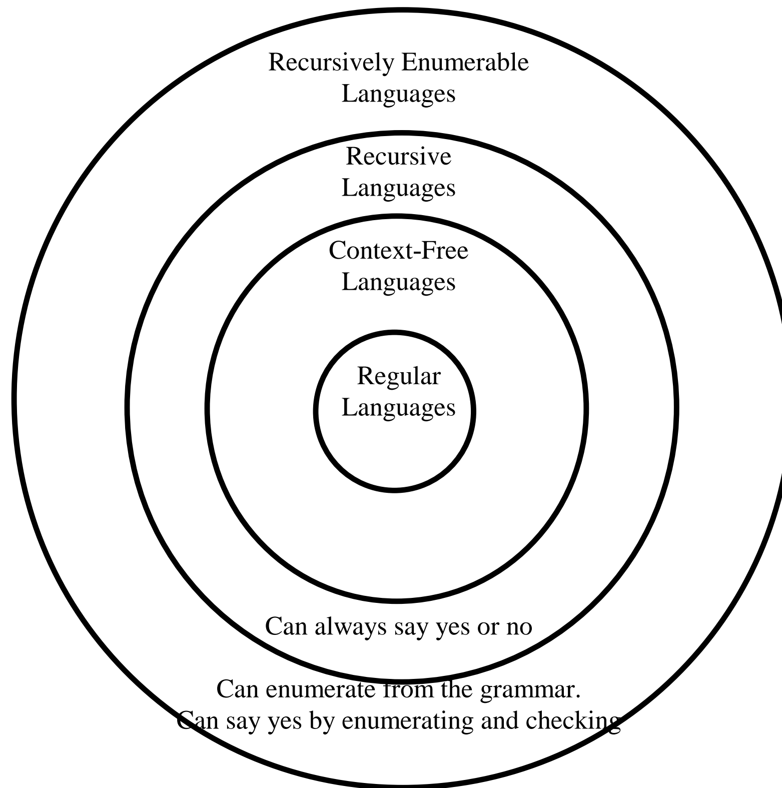
	I1	I2	I3	TROUBLE	I5
Machine 1	1				
Machine 2		1		1	
Machine 3					
TROUBLE			1		1
Machine 5	1	1	1	1	

But we've defined TROUBLE so that it will actually behave as:

TROUBLE			1	1	1

Or maybe HALT said that TROUBLE(TROUBLE) would halt. But then TROUBLE would loop.

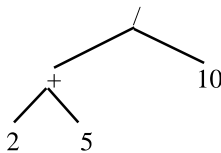
Decidability



Let's Revisit Some Problems

int alpha, beta;
alpha = 3;
beta = (2 + 5) / 10;

(1) **Lexical analysis**: Scan the program and break it up into variable names, numbers, etc.
(2) **Parsing**: Create a tree that corresponds to the sequence of operations that should be executed, e.g.,



(3) **Optimization**: Realize that we can skip the first assignment since the value is never used and that we can precompute the arithmetic expression, since it contains only constants.
(4) **Termination**: Decide whether the program is guaranteed to halt.
(5) **Interpretation**: Figure out what (if anything) useful it does.

So What's Left?

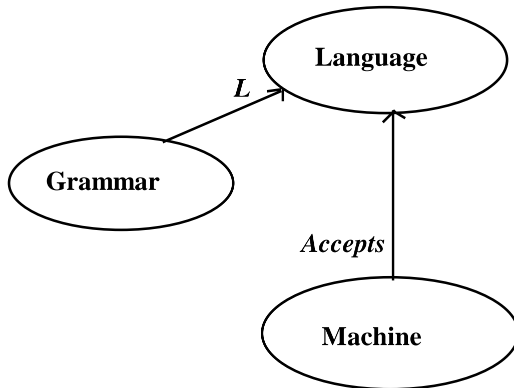
- Formalize and Prove Things

- Regular Languages and Finite State Machines
    - FSMs
        - Nondeterminism
        - State minimization
        - Implementation
    - Equivalence of regular expressions and FSMs
    - Properties of Regular Languages
- Context-Free Languages and PDAs
    - Equivalence of CFGs and nondeterministic PDAs
    - Properties of context-free languages
    - Parsing and determinism
- Turing Machines and Computability
    - Recursive and recursively enumerable languages
    - Extensions of Turing Machines
    - Undecidable problems for Turing Machines and unrestricted grammars

What Is a Language?

Do Homework 2.

Grammars, Languages, and Machines



Strings: the Building Blocks of Languages

An **alphabet** is a finite set of **symbols**: English alphabet: {A, B, C, …, Z}
Binary alphabet: {0, 1}

A **string** over an alphabet is a finite sequence of symbols drawn from the alphabet.

English string: happynewyear
binary string: 1001101

We will generally omit " " from strings unless doing so would lead to confusion.

The set of all possible strings over an alphabet Σ is written Σ*.
binary string: 1001101 ∈ {0,1}*

The shortest string contains no characters. It is called the **empty string** and is written " " or ε (epsilon).

The set of all possible strings over an alphabet Σ is written Σ*.

More on Strings

The **length** of a string is the number of symbols in it.
$|\varepsilon| = 0$
$|1001101| = 7$

A string a is a **substring** of a string b if a occurs contiguously as part of b.

aaa	is a substring of	aaabbbaaa
aaaaaa	is not a substring of	aaabbbaaa

Every string is a substring (although not a proper substring) of itself.

ε is a substring of every string. Alternatively, <u>we can match ε anywhere</u>.

Notice the analogy with sets here.

Operations on Strings

Concatenation: The **concatenation** of two strings x and y is written $x \| y$, $x \cdot y$, or xy and is the string formed by appending the string y to the string x.

$|xy| = |x| + |y|$

If $x = \varepsilon$ and y = "food", then xy =

If x = "good" and y = "bye", then |xy| =

Note: $x \cdot \varepsilon = \varepsilon \cdot x = x$ for all strings x.

Replication: For each string w and each natural number i, the string w^i is defined recursively as

$w^0 = \varepsilon$

$w^i = w^{i-1} w$ for each $i \geq 1$

Like exponentiation, the replication operator has a high precedence.

Examples:

$a^3 =$

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

$a^0 b^3 =$

String Reversal

An inductive definition:

(1) If $|w| = 0$ then $w^R = w = \varepsilon$

(2) If $|w| \geq 1$ then $\exists\, a \in \Sigma$: $w = u \cdot a$

(a is the last character of w)

and

$w^R = a \cdot u^R$

Example:

$(abc)^R =$

More on String Reversal

Theorem: If w, x are strings, then $(w \cdot x)^R = x^R \cdot w^R$

Example: $(\text{dogcat})^R = (\text{cat})^R \cdot (\text{dog})^R$ = tacgod

Proof (by induction on |x|):

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

Induction Hypothesis: If $|x| \leq n$, then $(w \cdot x)^R = x^R \cdot w^R$

Induction Step: Let $|x| = n + 1$. Then $x = u\, a$ for some character a and $|u| = n$

$(w \cdot x)^R$	$= (w \cdot (u \cdot a))^R$	
	$= ((w \cdot u) \cdot a)^R$	associativity
	$= a \cdot (w \cdot u)^R$	definition of reversal
	$= a \cdot u^R \cdot w^R$	induction hypothesis
	$= (u \cdot a)^R \cdot w^R$	definition of reversal
	$= x^R \cdot w^R$	

d o g c a t

w *x*

u *a*

Defining a Language

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

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

Some languages over Σ: $\varnothing$, $\{\varepsilon\}$, $\{a, b\}$, $\{\varepsilon, a, aa, aaa, aaaa, aaaaa\}$

The language Σ^* contains an infinite number of strings, including: ε, a, b, ab, ababaaa

Example Language Definitions

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

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

$L = \{a^n, n \geq 0\}$

$L = a^n$ (If we say nothing about the range of n, we will assume that it is drawn from N, i.e., $n \geq 0$.)

$L = \{x\#y: x,y \in \{0\text{-}9\}^* \text{ and square}(x) = y\}$

$L = \{\} = \varnothing$ (the empty language—not to be confused with $\{\varepsilon\}$, the language of the empty string)

Techniques for Defining Languages

Languages are sets. Recall that, for sets, it makes sense to talk about ***enumerations*** and ***decision procedures***. So, if we want to provide a computationally effective definition of a language we could specify either 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.

Example: The logical definition: $L = \{x : \exists y \in \{a, b\}^* : x = ya\}$ can be turned into either a language generator or a language recognizer.

How Large are Languages?

- The smallest language over any alphabet is $\varnothing$. $|\varnothing| = 0$
- The largest language over any alphabet is Σ^*. $|\Sigma^*| = ?$
  - If $\Sigma = \varnothing$ then $\Sigma^* = \{\varepsilon\}$ and $|\Sigma^*| = 1$
  - If $\Sigma \neq \varnothing$ then $|\Sigma^*|$ is countably infinite because its elements can be enumerated in 1 to 1 correspondence with the integers as follows:
    1. Enumerate all strings of length 0, then length 1, then length 2, and so forth.
    2. Within the strings of a given length, enumerate them lexicographically. E.g., aa, ab, ba, bb

- So all languages are either finite or countably infinite. Alternatively, all languages are *countable*.

Operations on Languages 1

Normal set operations: **union**, **intersection**, **difference**, **complement**…

Examples: $\Sigma = \{a, b\}$ L_1 = strings with an even number of a's
L_2 = strings with no b's

$L_1 \cup L_2 =$
$L_1 \cap L_2 =$
$L_2 - L_1 =$
$\neg(L_2 - L_1) =$

Operations on Languages 2

Concatenation: (based on the definition of concatenation of strings)

If L_1 and L_2 are languages over Σ, their concatenation $L = L_1 L_2$, sometimes $L_1 \cdot L_2$, is
$\{w \in \Sigma^*: w = x\,y \text{ for some } x \in L_1 \text{ and } y \in L_2\}$

Examples:

$L_1 = \{\text{cat, dog}\}$ $L_2 = \{\text{apple, pear}\}$ $L_1 L_2 = \{\text{catapple, catpear, dogapple, dogpear}\}$

$L_1 = \{a^n: n \geq 1\}$ $L_2 = \{a^n: n \leq 3\}$ $L_1 L_2 =$

Identities:

$L\varnothing = \varnothing L = \varnothing \quad \forall L$ (analogous to multiplication by 0)

$L\{\varepsilon\} = \{\varepsilon\}L = L \quad \forall L$ (analogous to multiplication by 1)

Replicated concatenation:

$L^n = L \cdot L \cdot L \cdot \ldots \cdot L$ (n times)

$L^1 = L$

$L^0 = \{\varepsilon\}$

Example:

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

$L^0 = \{\varepsilon\}$

$L^1 = \{\text{dog, cat, fish}\}$

$L^2 = \{\text{dogdog, dogcat, dogfish, catdog, catcat, catfish, fishdog, fishcat, fishfish}\}$

Concatenating Languages Defined Using Variables

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

$L_1 L_2 = \{a^n : n \geq 0\}\{b^n : n \geq 0\} = \{a^n b^m : n,m \geq 0\}$ (common mistake:) $\neq a^n b^n = \{a^n b^n : n \geq 0\}$

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

$L = 1^n 2^m$

$L = a^n b^m a^n$

Operations on Languages 3

Kleene Star (or Kleene closure): $\mathbf{L^*} = \{w \in \Sigma^* : w = w_1 w_2 \ldots w_k \text{ for some } k \geq 0 \text{ and some } w_1, w_2, \ldots w_k \in L\}$

Alternative definition: $\mathbf{L^*} = L^0 \cup L^1 \cup L^2 \cup L^3 \cup \ldots$

Note: $\forall L, \varepsilon \in L^*$

Example:

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

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

Another useful definition: $\mathbf{L^+} = L\,L^*$ (L^+ is the **closure** of L under concatenation)

Alternatively, $\mathbf{L^+} = L^1 \cup L^2 \cup L^3 \cup \ldots$

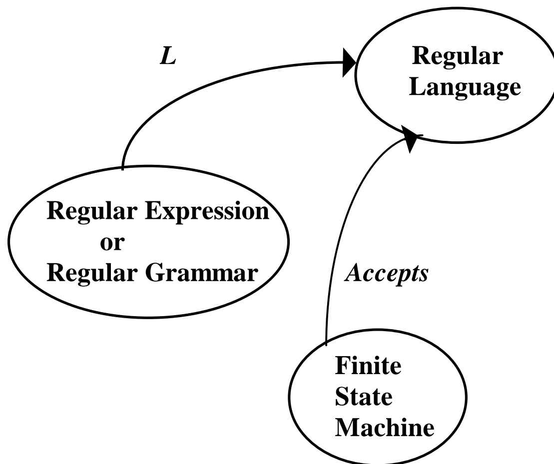
$L^+ = L^* - \{\varepsilon\}$ if $\varepsilon \notin L$

$L^+ = L^*$ if $\varepsilon \in L$

Regular Languages

Read Supplementary Materials: Regular Languages and Finite State Machines: Regular Languages
Do Homework 3.

Regular Grammars, Languages, and Machines



"Pure" Regular Expressions

The **regular expressions** over an alphabet Σ are all strings over the alphabet $\Sigma \cup \{$"(", ")", $\varnothing$, $\cup$, $*\}$ that can be obtained as follows:

1. $\varnothing$ and each member of Σ is a regular expression.
2. If α , β are regular expressions, then so is $\alpha\beta$
3. If α , β are regular expressions, then so is $\alpha\cup\beta$.
4. If α is a regular expression, then so is α^*.
5. If α is a regular expression, then so is (α).
6. Nothing else is a regular expression.

If $\Sigma = \{a,b\}$ then these are regular expressions: $\varnothing$, a, bab, $a\cup b$, $(a\cup b)^*a^*b^*$

So far, regular expressions are just (finite) strings over some alphabet, $\Sigma \cup \{$"(", ")", $\varnothing$, $\cup$, $*\}$.

Regular Expressions Define Languages

Regular expressions define languages via a **semantic interpretation function** we'll call L:
1. $L(\varnothing) = \varnothing$ and $L(a) = \{a\}$ for each $a \in \Sigma$
2. If α , β are regular expressions, then
   $L(\alpha\beta) = L(\alpha)\cdot L(\beta)$
   = all strings that can be formed by concatenating to some string from $L(\alpha)$ some string from $L(\beta)$.
   Note that if either α or β is $\varnothing$, then its language is $\varnothing$, so there is nothing to concatenate and the result is $\varnothing$.
3. If α , β are regular expressions, then $L(\alpha\cup\beta) = L(\alpha) \cup L(\beta)$
4. If α is a regular expression, then $L(\alpha^*) = L(\alpha)^*$
5. $L((\alpha)) = L(\alpha)$

A language is **regular** if and only if it can be described by a regular expression.

A regular expression is always finite, but it may describe a (countably) infinite language.

Regular Languages

An equivalent definition of the class of regular languages over an alphabet Σ:
The ***closure*** of the languages

$\{a\}\ \forall a \in \Sigma$ and $\varnothing$ [1]

with respect to the functions:

- concatenation, [2]
- union, and [3]
- Kleene star. [4]

In other words, the class of regular languages is the smallest set that includes all elements of [1] and that is closed under [2], [3], and [4].

"Closure" and "Closed"

Informally, a set can be defined in terms of a (usually small) starting set and a group of functions over elements from the set. The functions are applied to members of the set, and if anything new arises, it's added to the set. The resulting set is called the **closure** over the initial set and the functions. Note that the functions(s) may only be applied a **finite** number of times.

Examples:

The set of natural numbers **N** can be defined as the closure over $\{0\}$ and the successor (succ(n) = n+1) function.
Regular languages can be defined as the closure of $\{a\}\ \forall a \in \Sigma$ and $\varnothing$ and the functions of concatenation, union, and Kleene star.

We say a set is **closed** over a function if applying the function to arbitrary elements in the set does not yield any new elements.

Examples:

The set of natural numbers **N** is closed under multiplication.
Regular languages are closed under intersection.

See *Supplementary Materials—Review of Mathematical Concepts* for more formal definitions of these terms.

Examples of Regular Languages

$L(\ a^*b^*\ ) =$
$L(\ (a \cup b)\ ) =$
$L(\ (a \cup b)^*\ ) =$
$L(\ (a \cup b)^*a^*b^*) =$
$L = \{w \in \{a,b\}^* : |w| \text{ is even}\}$
$L = \{w \in \{a,b\}^* : w \text{ contains an odd number of a's}\}$

Augmenting Our Notation

It would be really useful to be able to write ε in a regular expression.

Example: $(a \cup \varepsilon)\ b$ (Optional a followed by b)

But we'd also like a minimal definition of what constitutes a regular expression. Why?

Observe that

$\varnothing^0 = \{\varepsilon\}$ (since 0 occurrences of the elements of any set generates the empty string), so
$\varnothing^* = \{\varepsilon\}$

So, without changing the set of languages that can be defined, we can add ε to our notation for regular expressions if we specify that

$L(\varepsilon) = \{\varepsilon\}$

We're essentially treating ε the same way that we treat the characters in the alphabet.
Having done this, you'll probably find that you rarely need $\varnothing$ in any regular expression.

More Regular Expression Examples

L((aa*) $\cup$ ε) =
L((a $\cup$ ε)*) =
L = { w $\in$ {a,b}* : there is no more than one b}
L = { w $\in$ {a,b}* : no two consecutive letters are the same}

Further Notational Extensions of Regular Expressions

- A fixed number of concatenations: α^n means αααα…α (n times).
- At Least 1: α^+ means 1 or more occurrences of α concatenated together.
- Shorthands for denoting sets, such as ranges, e.g., (A-Z) or (letter-letter)
  Example: L = $(A\text{-}Z)^+((A\text{-}Z)\cup(0\text{-}9))^*$

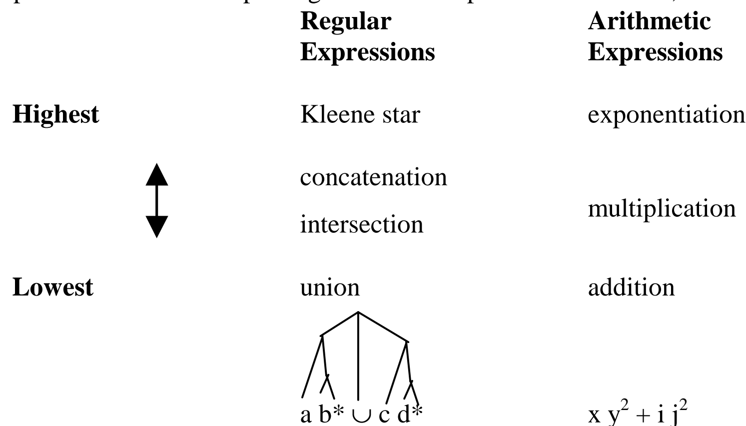
- A replicated regular expression α^n, where n is a constant.
  Example: L = $(0 \cup 1)^{20}$

- Intersection: $\alpha\cap\beta$ (we'll prove later that regular languages are closed under intersection)
  Example: L = $(a^3)^* \cap (a^5)^*$

Operator Precedence in Regular Expressions

Regular expressions are strings in the language of regular expressions. Thus to interpret them we need to:
1. Parse the string
2. Assign a meaning to the parse tree

Parsing regular expressions is a lot like parsing arithmetic expressions. To do it, we must assign precedence to the operators:



	Regular Expressions	Arithmetic Expressions
Highest	Kleene star	exponentiation
↕	concatenation	multiplication
	intersection	
Lowest	union	addition
	a b* ∪ c d*	$x\ y^2 + i\ j^2$

Regular Expressions and Grammars

Recall that grammars are language generators. A grammar is a recipe for creating strings in a language.

Regular expressions are analogous to grammars, but with two special properties:

1. The have limited power. They can be used to define only regular languages.
2. They don't look much like other kinds of grammars, which generally are composed of sets of production rules.

But we can write more "standard" grammars to define exactly the same languages that regular expressions can define. Specifically, any such grammar must be composed of rules that:

- have a left hand side that is a single nonterminal
- have a right hand side that is ε, or a single terminal, or a single terminal followed by a single nonterminal.

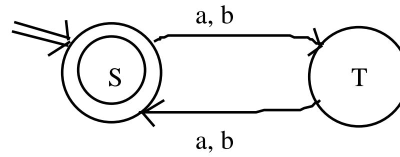
Regular Grammar Example

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

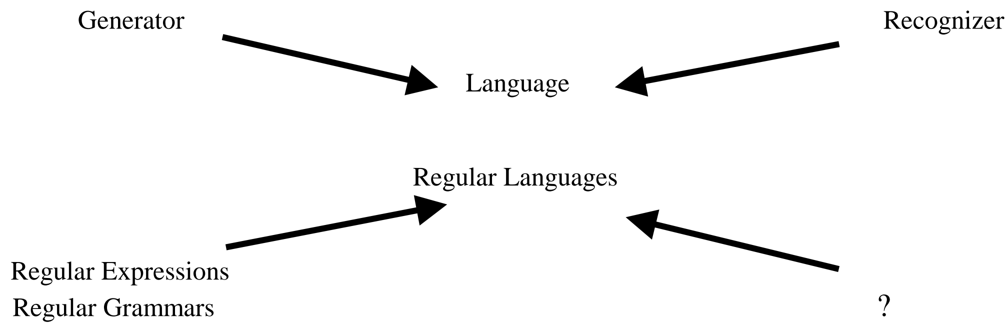
$((aa) \cup (ab) \cup (ba) \cup (bb))^*$

Notice how these rules correspond naturally to a FSM:

$S \rightarrow \varepsilon$
$S \rightarrow aT$
$S \rightarrow bT$
$T \rightarrow a$
$T \rightarrow b$
$T \rightarrow aS$
$T \rightarrow bS$



Generators and Recognizers



Finite State Machines

Read K & S 2.1
Do Homeworks 4 & 5.

Finite State Machines 1

A DFSM to accept odd integers:

Definition of a Deterministic Finite State Machine (DFSM)

$M = (K, \Sigma, \delta, s, F)$, where
- K is a finite set of states
- Σ is an alphabet
- $s \in K$ is the initial state
- $F \subseteq K$ is the set of final states, and
- δ is the transition function. It is function from $(K \times \Sigma)$ to K

i.e., each element of δ maps from: a state, input symbol pair to a new state.

Informally, M **accepts** a string w if M winds up in some state that is an element of F when it has finished reading w (if not, it **rejects** w).

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

Deterministic finite state machines (DFSMs) are also called deterministic finite state automata (DFSAs or DFAs).

Computations Using FSMs

A **computation** of A FSM is a sequence of configurations, where a configuration is any element of $K \times \Sigma^*$.
The **yields** relation $|\!-_M$:
$(q, w) \,|\!-_M\, (q', w')$ iff
- $w = a\,w'$ for some symbol $a \in \Sigma$, and
- $\delta(q, a) = q'$

(The yields relation effectively runs M one step.)

$|\!-_M{}^*$ is the reflexive, transitive closure of $|\!-_M$.
(The $|\!-_M{}^*$ relation runs M any number of steps.)

Formally, a FSM M **accepts** a string w iff
$(s, w) \,|\!-_M{}^*\, (q, \varepsilon)$, for some $q \in F$.

An Example Computation

A DFSM to accept odd integers:
On input 235, the configurations are:

(q0, 235) $|\!-_M$ (q0, 35)
$|\!-_M$
$|\!-_M$

Thus $(q0, 235) \,|\!-_{M^*}\, (q1, \varepsilon)$. (What does this mean?)

Finite State Machines 2

A DFSM to accept $.50 in change:

More Examples

$((aa) \cup (ab) \cup (ba) \cup (bb))^*$

$(b \cup \varepsilon)(ab)^*(a \cup \varepsilon)$

More Examples

L1 = {w ∈ {a, b}* : every a is immediately followed a b}

A regular expression for L1:

A DFSM for L1:

L2 = {w ∈ {a, b}* : every a has a matching b somewhere before it}

A regular expression for L2:

A DFSM for L2:

Another Example: Socket-based Network Communication

<u>Client</u>	<u>Server</u>	
open socket		Σ = {Open, Req, Reply, Close}
send request		
	send reply	L = Open (Req Reply)* (Req $\cup$ ε) Close
send request		
	send reply	
…		M =
close socket		

Definition of a Deterministic Finite State Transducer (DFST)

M = (K, Σ, O, δ, s, F), where

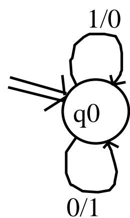
K is a finite set of states
Σ is an input alphabet
O is an output alphabet
$s \in K$ is the initial state
$F \subseteq K$ is the set of final states, and
δ is the transition function. It is function from
$(K \times \Sigma)$ to $(K \times O^*)$
i.e., each element of δ maps from: a state, input symbol pair
to : a new state and zero or more output symbols (an output string)

M **computes a function M(w)** if, when it reads w, it outputs M(w).

Theorem: The output language of a deterministic finite state transducer (on final state) is regular.

A Simple Finite State Transducer

Convert 1's to 0's and 0's to 1's (this isn't just a finite state task -- it's a one state task)



An Odd Parity Generator

After every three bits, output a fourth bit such that each group of four bits has odd parity.

Nondeterministic Finite State Machines

Read K & S 2.2, 2.3
Read Supplementary Materials: Regular Languages and Finite State Machines: Proof of the Equivalence of Nondeterministic and Deterministic FSAs.
Do Homework 6.

Definition of a Nondeterministic Finite State Machine (NDFSM/NFA)

M = (K, Σ, Δ, s, F), where

K is a finite set of states
Σ is an alphabet
$s \in K$ is the initial state
$F \subseteq K$ is the set of final states, and
Δ is the transition *relation*. It is a finite subset of

$(K \times (\Sigma \cup \{\varepsilon\})) \times K$

i.e., each element of Δ contains:

a configuration (state, input symbol or ε), and a new state.

M accepts a string w if there exists *some path* along which w drives M to some element of F.

The language accepted by M, denoted L(M), is the set of all strings accepted by M, where computation is defined analogously to DFSMs.

A Nondeterministic FSA

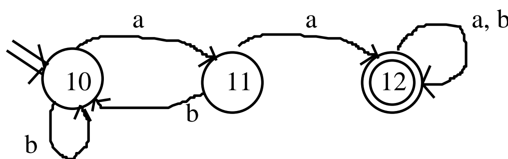
L= {w : there is a symbol $a_i \in \Sigma$ not appearing in w}

The idea is to guess (nondeterministically) which character will be the one that doesn't appear.

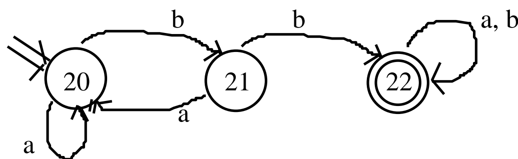
Another Nondeterministic FSA

L_1= {w : aa occurs in w}
L_2= {x : bb occurs in x}
L_3= {y : $\in L_1$ or L_2 }

M_1 =

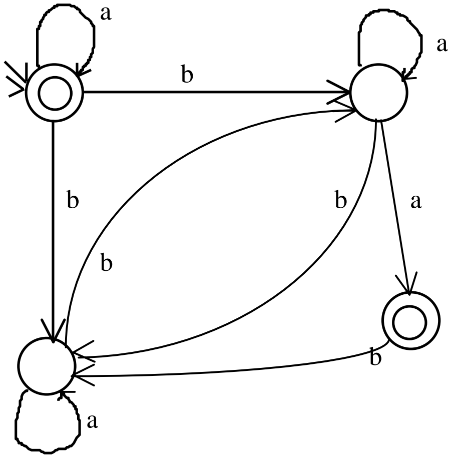


M_2=



M_3=

Analyzing Nondeterministic FSAs



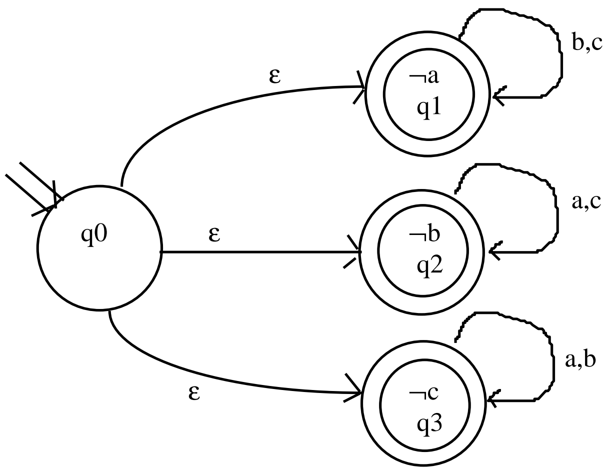
Does this FSA accept: baaba

Remember: we just have to find one accepting path.

Nondeterministic and Deterministic FSAs

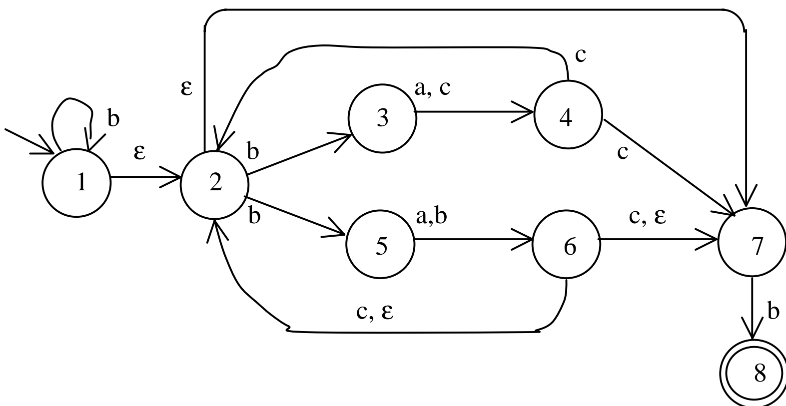
Clearly, {Languages accepted by a DFSA} ⊆ {Languages accepted by a NDFSA}
(Just treat δ as Δ)

More interestingly, **Theorem**: For each NDFSA, there is an equivalent DFSA.
Proof: By construction

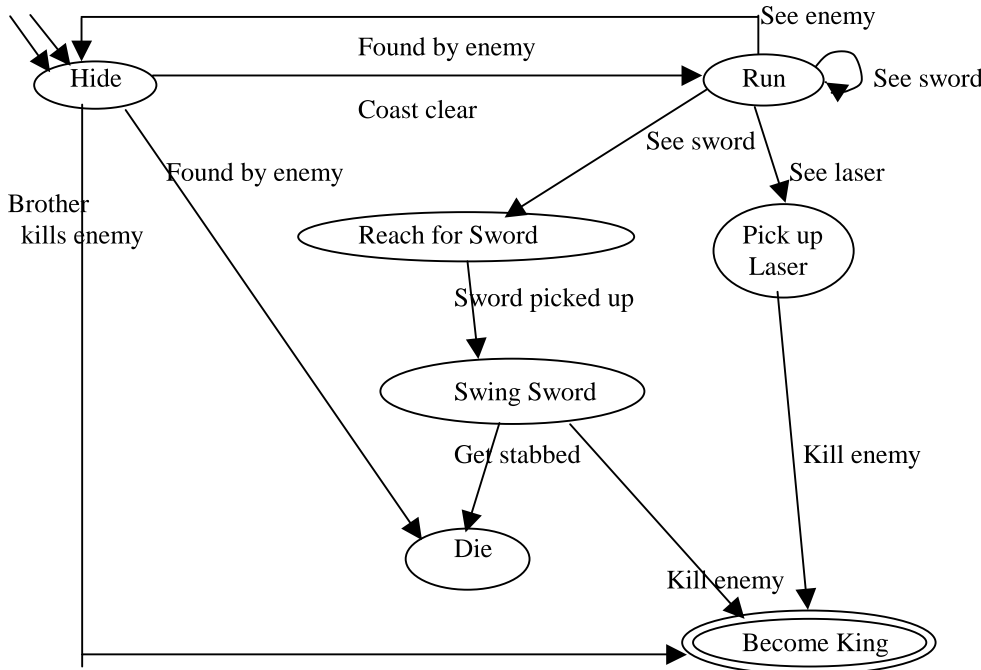


Another Nondeterministic Example

b* (b(a ∪ c)c ∪ b(a ∪ b) (c ∪ ε))* b



A "Real" Example



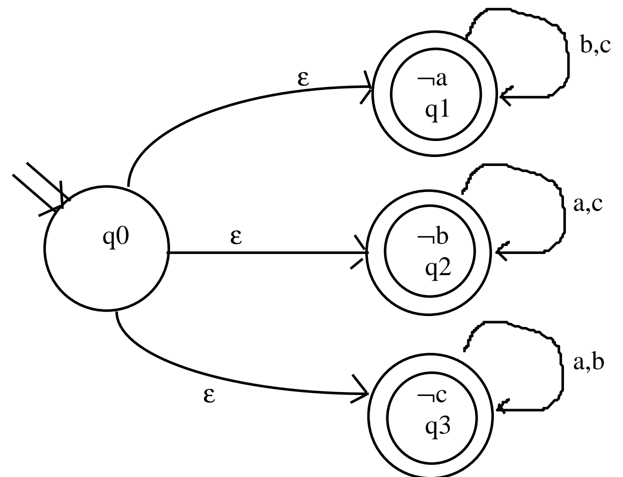
Dealing with ε Transitions

$E(q) = \{p \in K : (q,w) \vdash^*_M (p, w\}$. E(q) is the closure of {q} under the relation $\{(p,r) :$ there is a transition $(p, \varepsilon, r) \in \Delta\}$
An algorithm to compute E(q):

Defining the Deterministic FSA

Given a NDFSA $M = (K, \Sigma, \Delta, s, F)$,
we construct $M' = (K', \Sigma, \delta', s', F')$, where
$K' = 2^K$
$s' = E(s)$
$F' = \{Q \subseteq K : Q \cap F \neq \emptyset\}$
$\delta'(Q, a) = \cup\{E(p) : p \in K$ and $(q, a, p) \in \Delta$
for some $q \in Q\}$

Example: computing δ' for the missing letter machine
s' = {q0, q1, q2, q3}
δ' = { ({q0, q1, q2, q3}, a, {q2, q3}),
({q0, q1, q2, q3}, b, {q1, q3}),
({q0, q1, q2, q3}, c, {q1, q2}),
({q1, q2}, a, {q2}), ({q1, q2}, b, {q1}), ({q1, q2}, c, {q1, q2})
({q1, q3}, a, {q3}), ({q1, q3}, b, {q1, q3}), ({q1, q3}, c, {q1})
({q2, q3}, a, {q2, q3}), ({q2, q3}, b, {q3}), ({q2, q3}, c, {q2})
({q1}, b, {q1}), ({q1}, c, {q1})
({q2}, a, {q2}), ({q2}, c, {q2})
({q3}, a, {q3}), ({q3}, b, {q3}) }

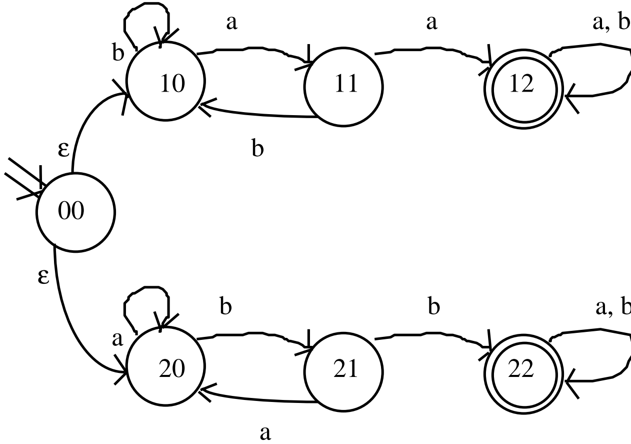


An Algorithm for Constructing the Deterministic FSA

1. Compute the E(q)s:
2. Compute s' = E(s)
3. Compute δ':
   $\delta'(Q, a) = \cup\{E(p) : p \in K \text{ and } (q, a, p) \in \Delta \text{ for some } q \in Q\}$
4. Compute K' = a subset of 2^K
5. Compute $F' = \{Q \in K' : Q \cap F \neq \emptyset\}$

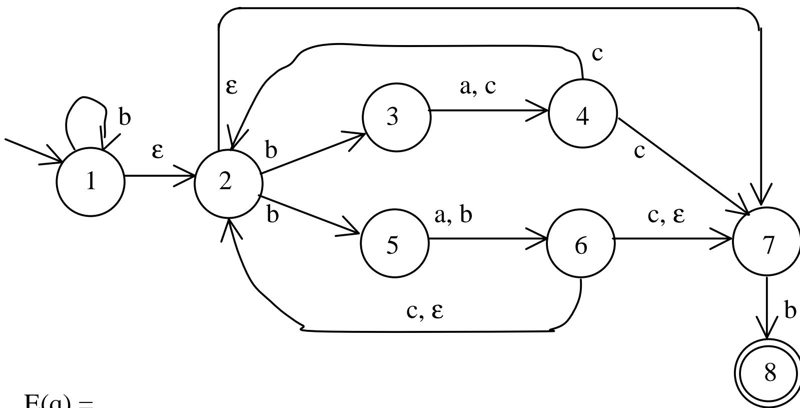
An Example - The Or Machine

$L_1 = \{w : aa \text{ occurs in } w\}$
$L_2 = \{x : bb \text{ occurs in } x\}$
$L_3 = \{y : \in L_1 \text{ or } L_2\}$



Another Example

b* (b(a ∪ c)c ∪ b(a ∪ b) (c ∪ ε))* b



E(q) =

δ' =

Sometimes the Number of States Grows Exponentially

Example: The missing letter machine, with $|\Sigma| = n$

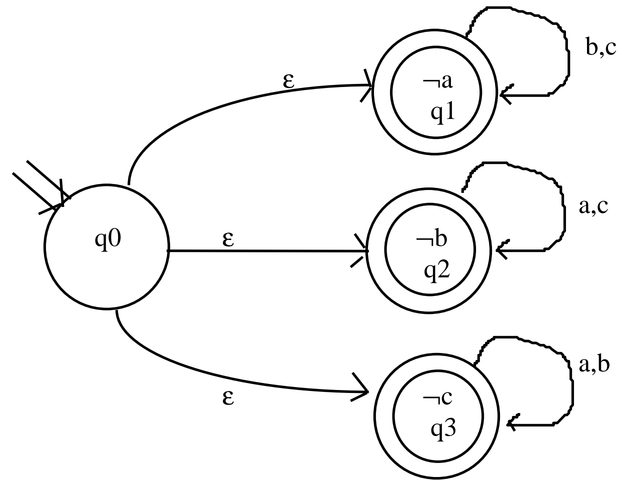
No. of states after 0 chars: 1

No. of new states after 1 char: $\binom{n}{n-1} = n$

No. of new states after 2 chars: $\binom{n}{n-2} = n(n-1)/2$

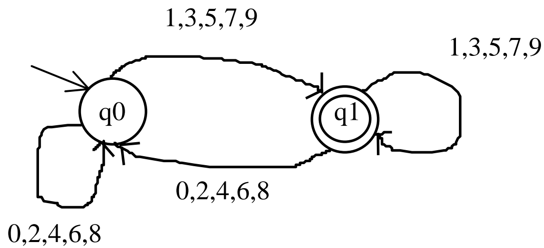
No. of new states after 3 chars: $\binom{n}{n-3} = n(n-1)(n-2)/6$

Total number of states after n chars: 2^n



What If The Original FSA is Deterministic?

M=



1. Compute the E(q)s:
2. s' = E(q0) =
3. Compute δ'
   ({q0}, odd, {q1})
   ({q0}, even, {q0})
   ({q1}, odd, {q1})
   ({q1}, even, {q0})
4. K' = {{q0}, {q1}}
5. F' = {{q1}}

M' = M

The real meaning of "determinism"

A FSA is **deterministic** if, for each input and state, there is at most one possible transition.

DFSAs are always deterministic. Why?

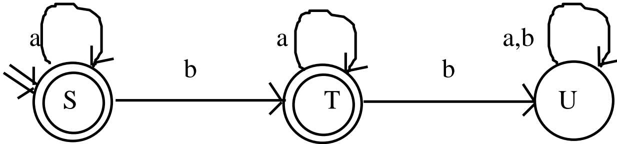
NFSAs can be deterministic (even with ε-transitions and implicit dead states), but the formalism allows nondeterminism, in general.

Determinism implies uniquely defined machine behavior.

Interpreters for Finite State Machines

Deterministic FSAs as Algorithms

Example: No more than one b




```
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 go to U;
etc.
```

Length of Program: $|K| \times (|\Sigma| + 2)$
Time required to analyze string w: $O(|w| \times |\Sigma|)$

We have to write new code for every new FSM.

Until accept or reject do:

A Deterministic FSA Interpreter

To simulate $M = (K, \Sigma, \delta, s, F)$:

Simulate the no more than one b machine on input: aabaa

```
ST := s;
Repeat
        i := get-next-symbol;
        if i ≠ end-of-string then
                ST := δ(ST, i)
Until i = end-of-string;
If ST ∈ F then accept else reject
```

Nondeterministic FSAs as Algorithms

Real computers are deterministic, so we have three choices if we want to execute a nondeterministic FSA:

1. Convert the NDFSA to a deterministic one:
    - Conversion can take time and space 2^K.
    - Time to analyze string w: $O(|w|)$

2. Simulate the behavior of the nondeterministic one by constructing sets of states "on the fly" during execution
    - No conversion cost
    - Time to analyze string w: $O(|w| \times K^2)$

3. Do a depth-first search of all paths through the nondeterministic machine.

A Nondeterministic FSA Interpreter

To simulate $M = (K, \Sigma, \Delta, s, F)$:

```
SET ST;
ST := E(s);
Repeat
        i := get-next-symbol;
        if  i ≠ end-of-string then
                ST1 := ∅
                For all q ∈ ST do
                        For all r ∈ Δ(q, i) do
                                ST1 := ST1 ∪ E(r);
                ST := ST1;

Until i = end-of-string;
If  ST ∩ F ≠ ∅ then accept else reject
```

A Deterministic Finite State Transducer Interpreter

To simulate $M = (K, \Sigma, O, \delta, s, F)$, given that:

δ_1(state, symbol) returns a single new state (i.e., M is deterministic), and

δ_2(state, symbol) returns an element of O*, the string to be output.

```
ST := s;
Repeat:
        i := get-next-symbol;
        if  i ≠ end-of-string then
                write(δ2(ST, i));
                ST := δ1(ST, i)
Until i = end-of-string;
If  ST ∈ F then accept else reject
```

Equivalence of Regular Languages and FSMs

Read K & S 2.4
Read Supplementary Materials: Regular Languages and Finite State Machines: Generating Regular Expressions from Finite State Machines.
Do Homework 8.

Equivalence of Regular Languages and FSMs

Theorem: The set of languages expressible using regular expressions (the regular languages) equals the class of languages recognizable by finite state machines. Alternatively, a language is regular if and only if it is accepted by a finite state machine.

Proof Strategies

Possible Proof Strategies for showing that two sets, *a* and *b* are equal (also for iff):

1. Start with *a* and apply valid transformation operators until *b* is produced.

Example:
Prove:
$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$

$A \cap (B \cup C)$	$= (B \cup C) \cap A$	commutativity
	$= (B \cap A) \cup (C \cap A)$	distributivity
	$= (A \cap B) \cup (A \cap C)$	commutativity

2. Do two separate proofs: (1) a $\Rightarrow$ b, and (2) b $\Rightarrow$a, possibly using totally different techniques. In this case, we show first (by construction) that for every regular expression there is a corresponding FSM. Then we show, by induction on the number of states, that for every FSM, there is a corresponding regular expression.

For Every Regular Expression There is a Corresponding FSM

We'll show this by construction.

Example:

$a^*(b \cup \varepsilon)a^*$

Review - Regular Expressions

The regular expressions over an alphabet Σ^* are all strings over the alphabet $\Sigma \cup \{(,), \varnothing, \cup, *\}$ that can be obtained as follows:

1. $\varnothing$ and each member of Σ is a regular expression.
2. If α , β are regular expressions, then so is $\alpha\beta$.
3. If α , β are regular expressions, then so is $\alpha\cup\beta$.
4. If α is a regular expression, then so is α^*.
5. If α is a regular expression, then so is (α).
6. Nothing else is a regular expression.

We also allow ε and α^+, etc. but these are just shorthands for $\varnothing^*$ and $\alpha\alpha^*$, etc. so they do not need to be considered for completeness.

For Every Regular Expression There is a Corresponding FSM

Formalizing the Construction: The class of regular languages is the smallest class of languages that contains ∅ and each of the singleton strings drawn from Σ, and that is closed under

- Union
- Concatenation, and
- Kleene star

Clearly we can construct an FSM for any finite language, and thus for ∅ and all the singleton strings. If we could show that the class of languages accepted by FSMs is also closed under the operations of union, concatenation, and Kleene star, then we could recursively construct, for any regular expression, the corresponding FSM, starting with the singleton strings and building up the machine as required by the operations used to express the regular expression.

FSMs for Primitive Regular Expressions

An FSM for ∅:

An FSM for ε (∅*):

An FSM for a single element of Σ:

Closure of FSMs Under Union

To create a FSM that accepts the union of the languages accepted by machines M1 and M2:

1. Create a new start state, and, from it, add ε-transitions to the start states of M1 and M2.

Closure of FSMs Under Concatenation

To create a FSM that accepts the concatenation of the languages accepted by machines M1 and M2:

1. Start with M1.
2. From every final state of M1, create an ε-transition to the start state of M2.
3. The final states are the final states of M2.

Closure of FSMs Under Kleene Star

To create an FSM that accepts the Kleene star of the language accepted by machine M1:

1. Start with M1.
2. Create a new start state S0 and make it a final state (so that we can accept ε).
3. Create an ε-transition from S0 to the start state of M1.
4. Create ε-transitions from all of M1's final states back to its start state.
5. Make all of M1's final states final.

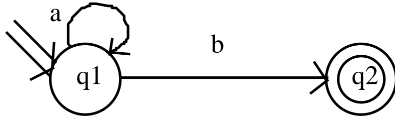
Note: we need a new start state, S0, because the start state of the new machine must be a final state, and this may not be true of M1's start state.

Closure of FSMs Under Complementation

To create an FSM that accepts the complement of the language accepted by machine M1:

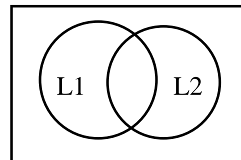
1. Make M1 deterministic.
2. Reverse final and nonfinal states.

A Complementation Example



Closure of FSMs Under Intersection

$L1 \cap L2 =$



Write this in terms of operations we have already proved closure for:

- Union
- Concatenation
- Kleene star
- Complementation

An Example

$(b \cup ab^*a)^*ab^*$

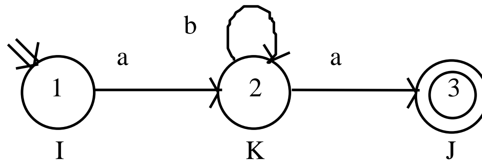
For Every FSM There is a Corresponding Regular Expression

Proof:
(1) There is a trivial regular expression that describes the strings that can be recognized in going from one state to itself ({ε} plus any other single characters for which there are loops) or from one state to another directly (i.e., without passing through any other states), namely all the single characters for which there are transitions.

(2) Using (1) as the base case, we can build up a regular expression for an entire FSM by induction on the number assigned to possible intermediate states we can pass through. By adding them in only one at a time, we always get simple regular expressions, which can then be combined using union, concatenation, and Kleene star.

Key Ideas in the Proof

Idea 1: Number the states and, at each induction step, increase by one the states that can serve as intermediate states.

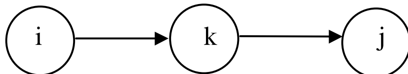


Idea 2: To get from state I to state J without passing through any intermediate state numbered greater than K, a machine may either:
1. Go from I to J without passing through any state numbered greater than K-1 (which we'll take as the induction hypothesis), or
2. Go from I to K, then from K to K any number of times, then from K to J, in each case without passing through any intermediate states numbered greater than K-1 (the induction hypothesis, again).

So we'll start with no intermediate states allowed, then add them in one at a time, each time building up the regular expression with operations under which regular languages are closed.

The Formula

Adding in state k as an intermediate state we can use to go from i to j, described using paths that don't use k:



R(i, j, k) = R(i, j, k - 1) /* what you could do without k

$\cup$

R(i, k, k-1) /* go from i to the new intermediate state without using k or higher

$\circ$

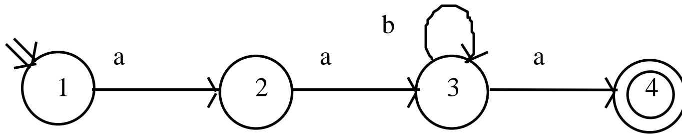
R(k, k, k-1)* /* then go from the new intermediate state back to itself as many times as you want

$\circ$

R(k, j, k-1) /* then go from the new intermediate state to j without using k or higher

Solution: $\cup$ R(s, q, N) $\forall q \in F$

An Example of the Induction



Going through no intermediate states:

$(1,1,0) = \varepsilon$ $(1,2,0) = a$ $(1, 3, 0) = \varnothing$ $(2,3,0) = a$ $(3,3,0) = \varepsilon \cup b$ $(3,4,0) = a$

Allow 1 as an intermediate state:

Allow 2 as an intermediate state:

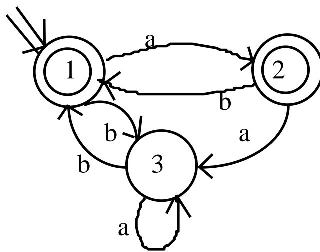
$(1, 3, 2) = (1, 3, 1) \cup (1, 2, 1)(2, 2, 1)^*(2, 3, 1)$
$= \varnothing \cup a\ \varepsilon^*\ a$
$= aa$

Allow 3 as an intermediate state:

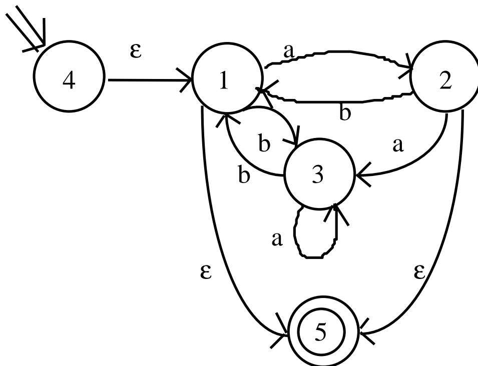
$(1, 3, 3) = (1, 3, 2) \cup (1, 3, 2)(3, 3, 2)^*(3, 3, 2)$
$= aa \cup aa\ (\varepsilon \cup b)^*\ (\varepsilon \cup b)$
$= aab^*$

$(1, 4, 3) = (1, 4, 2) \cup (1, 3, 2)(3, 3, 2)^*(3, 4, 2)$
$= \varnothing \cup aa\ (\varepsilon \cup b)^*\ a$
$= aab^*a$

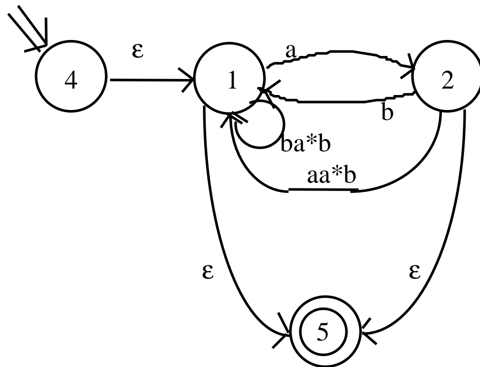
An Easier Way - See Packet



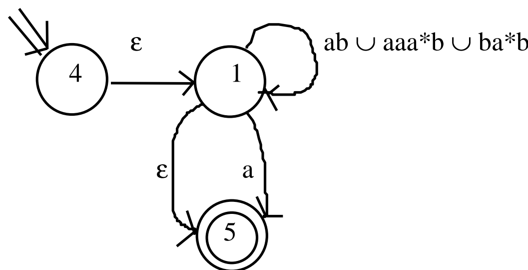
(1) Create a new initial state and a new, unique final state, neither of which is part of a loop.

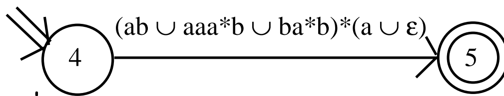


(2) Remove states and arcs and replace with arcs labelled with larger and larger regular expressions. States can be removed in any order, but don't remove either the start or final state.



(Notice that the removal of state 3 resulted in two new paths because there were two incoming paths to 3 from another state and 1 outgoing path to another state, so 2×1 = 2.) The two paths from 2 to 1 should be coalesced by unioning their regular expressions (not shown).





Thus, the equivalent regular expression is:

(ab ∪ aaa*b ∪ ba*b)*(a ∪ ε)

Using Regular Expressions in the Real World (PERL)

Matching floating point numbers:

-? ([0-9]+(\.[0-9]*)? | \.[0-9]+)

Matching IP addresses:

([0-9]+ (\ . [0-9]+) {3})

Finding doubled words:

\< ([A-Za-z]+) \s+ \1 \>

From Friedl, J., Mastering Regular Expressions, O'Reilly,1997.

Note that some of these constructs are more powerful than regular expressions.

Regular Grammars and Nondeterministic FSAs

Any regular language can be defined by a regular grammar, in which all rules

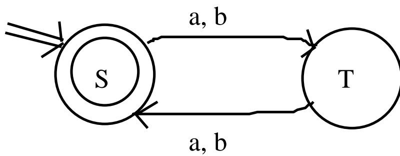
- have a left hand side that is a single nonterminal
- have a right hand side that is ε, a single terminal, a single nonterminal, or a single terminal followed by a single nonterminal.

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

$((aa) \cup (ab) \cup (ba) \cup (bb))^*$

$S \rightarrow \varepsilon$
$S \rightarrow aT$
$S \rightarrow bT$

$T \rightarrow a$
$T \rightarrow b$
$T \rightarrow aS$
$T \rightarrow bS$



An Algorithm to Generate the NDFSM from a Regular Grammar

1. Create a nonterminal for each state in the NDFSM.
2. s is the start state.
3. If there are any rules of the form $X \rightarrow w$, for some $w \in \Sigma$, then create an additional state labeled #.
4. For each rule of the form $X \rightarrow w\ Y$, add a transition from X to Y labeled w ($w \in \Sigma \cup \varepsilon$).
5. For each rule of the form $X \rightarrow w$, add a transition from X to # labeled w ($w \in \Sigma$).
6. For each rule of the form $X \rightarrow \varepsilon$, mark state X final.
7. Mark state # final.

Example 1 - Even Length Strings

$S \rightarrow \varepsilon$
$S \rightarrow aT$
$S \rightarrow bT$

$T \rightarrow a$
$T \rightarrow b$
$T \rightarrow aS$
$T \rightarrow bS$

Example 2 - One Character Missing

$S \rightarrow \varepsilon$
$S \rightarrow aB$
$S \rightarrow aC$
$S \rightarrow bA$
$S \rightarrow bC$
$S \rightarrow cA$
$S \rightarrow cB$

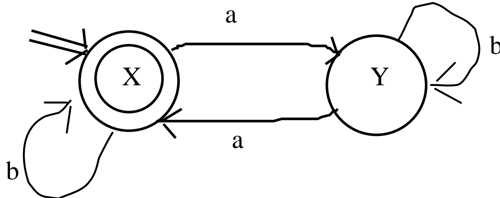
$A \rightarrow bA$
$A \rightarrow cA$
$A \rightarrow \varepsilon$
$B \rightarrow aB$
$B \rightarrow cB$
$B \rightarrow \varepsilon$

$C \rightarrow aC$
$C \rightarrow bC$
$C \rightarrow \varepsilon$

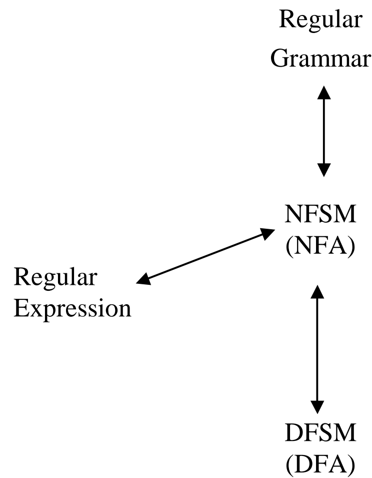
An Algorithm to Generate a Regular Grammar from an NDFSM

1. Create a nonterminal for each state in the NDFSM.
2. The start state becomes the starting nonterminal
3. For each transition $\delta(T, a) = U$, make a rule of the form $T \rightarrow aU$.
4. For each final state T, make a rule of the form $T \rightarrow \varepsilon$.

Example:



Conversion Algorithms between Regular Language Formalisms



Languages That Are and Are Not Regular

Read L & S 2.5, 2.6
Read Supplementary Materials: Regular Languages and Finite State Machines: The Pumping Lemma for Regular Languages.
Do Homework 9.

Deciding Whether a Language is Regular

Theorem: There exist languages that are not regular.
Lemma: There are an uncountable number of languages.
Proof of Lemma:
Let: Σ be a finite, nonempty alphabet, e.g., {a, b, c}.

Then Σ* contains all finite strings over Σ.
e.g., {ε, a, b, c, aa, ab, bc, abc, bba, bbaa, bbbaac}

Σ* is countably infinite, because its elements can be enumerated one at a time, shortest first.

Any language L over Σ is a subset of Σ*, e.g., L1 = {a, aa, aaa, aaaa, aaaaa, …}
L2 = {ab, abb, abbb, abbbb, abbbbb, …}
The set of all possible languages is thus the power set of Σ*.

The power set of any countably infinite set is not countable. So there are an uncountable number of languages over Σ*.

Some Languages Are Not Regular

Theorem: There exist languages that are not regular.
Proof:
(1) There are a countably infinite number of regular languages. This true because every description of a regular language is of finite length, so there is a countably infinite number of such descriptions.
(2) There are an uncountable number of languages.

Thus there are more languages than there are regular languages. So there must exist some language that is not regular.

Showing That a Language is Regular

Techniques for showing that a language L is regular:
1. Show that L has a finite number of elements.
2. Exhibit a regular expression for L.
3. Exhibit a FSA for L.
4. Exhibit a regular grammar for L.
5. Describe L as a function of one or more other regular languages and the operators ·, ∪, ∩, *, -, ¬. We use here the fact that the regular languages are closed under all these operations.
6. Define additional operators and prove that the regular languages are closed under them. Then use these operators as in 5.

Example

Let Σ = {0, 1, 2, … 9}
Let L ⊆ Σ* be the set of decimal representations for nonnegative integers (with no leading 0's) divisible by 2 or 3.

L_1 = decimal representations of nonnegative integers without leading 0's.
$L_1 = 0 \cup \{1, 2, \ldots 9\}\{0 - 9\}^*$
So L_1 is regular.

L_2 = decimal representations of nonnegative integers without leading 0's divisible by 2
$L_2 = L_1 \cap \Sigma^*\{0, 2, 4, 6, 8\}$
So L_2 is regular.

Example, Continued

$L_3 = L_1$ and divisible by 3

Recall that a number is divisible by 3 if and only if the sum of its digits is divisible by 3. We can build a FSM to determine that and accept the language L_{3a}, which is composed of strings of digits that sum to a multiple of 3.

$L_3 = L_1 \cap L_{3a}$

Finally, $L = L_2 \cup L_3$

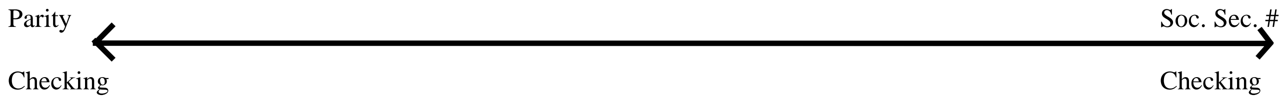
Another Example

$\Sigma = \{0 - 9\}$
L = {w : w is the social security number of a living US resident}

Finiteness - Theoretical vs. Practical

Any finite language is regular. The size of the language doesn't matter.

Parity Checking



Soc. Sec. # Checking

But, from an implementation point of view, it very well may.

When is an FSA a good way to encode the facts about a language?

What are our alternatives?

FSA's are good at looking for repeating patterns. They don't bring much to the table when the language is just a set of unrelated strings.

Showing that a Language is Not Regular

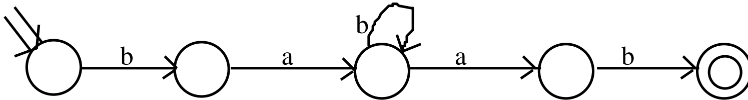
The argument, "I can't find a regular expression or a FSM", won't fly. (But a proof that there cannot exist a FSM is ok.)

Instead, we need to use two fundamental properties shared by regular languages:

1. We can only use a finite amount of memory to record essential properties.
   Example:
   $a^n b^n$ is not regular

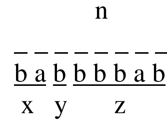
2. The only way to generate/accept an infinite language with a finite description is to use Kleene star (in regular expressions) or cycles (in automata). This forces some kind of simple repetitive cycle within the strings.
   Example:
   ab*a generates aba, abba, abbba, abbbba, etc.
   Example:
   $\{a^n : n \geq 1 \text{ is a prime number}\}$ is not regular.

Exploiting the Repetitive Property



If a FSM of n states accepts any string of length ≥ n, how many strings does it accept?

L = bab*ab



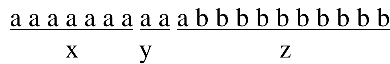
xy*z must be in L.

So L includes: baab, babab, babbab, babbbbbbbbbbab

The Pumping Lemma for Regular Languages

If L is regular, then
$\exists$ N ≥ 1, such that
$\forall$ strings w ∈ L, where |w| ≥ N,
$\exists$ x, y, z, such that w = xyz
and |xy| ≤ N,
and y ≠ ε,
and $\forall$ q ≥ 0, xy^qz is in L.

Example: L = a^nb^n

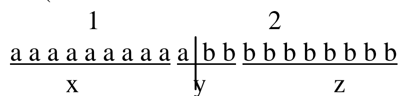


$\exists$ N ≥ 1	Call it N
$\forall$ long strings w	We pick one
$\exists$ x, y, z	We show no x, y, z

Example: a^nb^n is not Regular

N is the number from the pumping lemma (or one more, if N is odd).

Choose w = $a^{N/2}b^{N/2}$. (Since this is what it takes to be "long enough": |w| ≥ N)



We show that there is no x, y, z with the required properties:
|xy| ≤ N,
y ≠ ε,
$\forall$ q ≥ 0, xy^qz is in L.

Three cases to consider:

- y falls in region 1:
- y falls across regions 1 and 2:
- y falls in region 3:

Example: $a^n b^n$ is not Regular

Second try:

Choose w to be be $a^N b^N$. (Since we get to choose any w in L.)

```
                1              |          2
        a a a a a a a a a a a | b b b b b b b b b b
            x          y      |          z
```

We show that there is no x, y, z with the required properties:

$|xy| \leq N$,
$y \neq \varepsilon$,
$\forall\, q \geq 0$, $xy^q z$ is in L.

Since $|xy| \leq N$, y must be in region 1. So $y = a^g$ for some $g \geq 1$. Pumping in or out (any q but 1) will violate the constraint that the number of a's has to equal the number of b's.

A Complete Proof Using the Pumping Lemma

Proof that $L = \{a^n b^n\}$ is not regular:

Suppose L is regular. Since L is regular, we can apply the pumping lemma to L. Let N be the number from the pumping lemma for L. Choose $w = a^N b^N$. Note that $w \in L$ and $|w| \geq N$. From the pumping lemma, there exists some x, y, z where $xyz = w$ and $|xy| \leq N$, $y \neq \varepsilon$, and $\forall\, q \geq 0$, $xy^q z \in L$. Because $|xy| \leq N$, $y = a^{|y|}$ (y is all a's). We choose $q = 2$ and $xy^q z = a^{N+|y|} b^N$. Because $|y| > 0$, then $xy^2 z \notin L$ (the string has more a's than b's). Thus for all possible x, y, z: $xyz = w$, $\exists q$, $xy^q z \notin L$. Contradiction. $\therefore$ L is not regular.

Note: the underlined parts of the above proof is "boilerplate" that can be reused. A complete proof should have this text or something equivalent.

You get to choose w. Make it a single string that depends only on N. Choose w so that it makes your proof easier.
You may end up with various cases with different q values that reach a contradiction. You have to show that all possible cases lead to a contradiction.

Proof of the Pumping Lemma

Since L is regular it is accepted by some DFSA, M. Let N be the number of states in M. Let w be a string in L of length N or more.

```
                    N
a a a a a a a a a a a b b b b b b b b b b
       x         y
       x      y
```

Then, in the first N steps of the computation of M on w, M must visit N+1 states. But there are only N different states, so it must have visited the same state more than once. Thus it must have looped at least once. We'll call the portion of w that corresponds to the loop y. But if it can loop once, it can loop an infinite number of times. Thus:

- M can recognize $xy^q z$ for all values of $q \geq 0$.
- $y \neq \varepsilon$ (since there was a loop of length at least one)
- $|xy| \leq N$ (since we found y within the first N steps of the computation)

.

Another Pumping Example

$L = \{w=a^J b^K : K > J\}$ (more b's than a's)

Choose $w = a^N b^{N+1}$

```
                   N
a a a a a a a a a a b b b b b b b b b b b
      x          y            z
```

We are guaranteed to pump only a's, since $|xy| \leq N$. So there exists a number of copies of y that will cause there to be more a's than b's, thus violating the claim that the pumped string is in L.

A Slightly Different Example of Pumping

$L = \{w=a^J b^K : J > K\}$ (more a's than b's)

Choose $w = a^{N+1} b^N$

```
               N
a a a a a a a a a a b b b b b b b b b b b
     x       y          z
```

We are guaranteed that y is a string of at least one a, since $|xy| \leq N$. But if we pump in a's we get even more a's than b's, resulting in strings that are in L.

What can we do?

Another Slightly Different Example of Pumping

$L = \{w=a^J b^K : J \geq K\}$

Choose $w = a^{N+1} b^N$

```
               N
 a a a a a a a a a a b b b b b b b b b b b
x             y             z
```

We are guaranteed that y is a string of at least one a, since $|xy| \leq N$. But if we pump in a's we get even more a's than b's, resulting in strings that are in L.

If we pump out, then if y is just a then we still have a string in L.

What can we do?

Another Pumping Example

$L = aba^nb^n$

Choose $w = aba^Nb^N$

N
a b a a a a a a a a a a a b b b b b b b b b b b
x y z

What are the choices for (x, y):
(ε, a)
(ε, ab)
(ε, aba^+)
(a, b)
(a, ba^+)
(aba*, a^+)

What if L is Regular?

Given a language L that is regular, pumping will work: L = (ab)* Choose $w = (ab)^N$

There must exist an x, y, and z where y is pumpable.

abababab ababab abababababab
x y z

Suppose y = ababab Then, for all $q \geq 0$, $x y^q z \in L$

Note that this does not prove that L is regular. It just fails to prove that it is not.

Using Closure Properties

Once we have some languages that we can prove are not regular, such as a^nb^n, we can use the closure properties of regular languages to show that other languages are also not regular.

Example: $\Sigma = \{a, b\}$
L = {w : w contains an equal number of a's and b's }
a*b* is regular. So, if L is regular, then $L_1 = L \cap a^*b^*$ is regular.

But L_1 is precisely a^nb^n. So L is not regular.

Don't Try to Use Closure Backwards

One Closure Theorem:
If L_1 and L_2 are regular, then so is $L_3 = L_1 \cap L_2$.

But what if L_3 and L_1 are regular? What can we say about L_2?

$L_3 = L_1 \cap L_2$.

Example: $ab = ab \cap a^nb^n$

A Harder Example of Pumping

$\Sigma = \{a\}$
$L = \{w = a^K : K \text{ is a prime number}\}$

N
a a a a a a a a a a a a a a
x y z

$|x| + |z|$ is prime.
$|x| + |y| + |z|$ is prime.
$|x| + 2|y| + |z|$ is prime.
$|x| + 3|y| + |z|$ is prime, and so forth.

Distribution of primes:
| | | | | | | | | | | | | |

Distribution of $|x| + q|y| + |z|$:
| | | | | | | | | | |

But the Prime Number Theorem tells us that the primes "spread out", i.e., that the number of primes not exceeding x is asymptotic to x/ln x.

Note that when $q = |x| + |z|$, $|xy^qz| = (|y| + 1)\times(|x| + |z|)$, which is composite (non-prime) if both factors are > 1. If you're careful about how you choose N in a pumping lemma proof, you can make this true for both factors.

Automata Theory is Just the Scaffolding

Our results so far give us tools to:

- Show a language is regular by:
  - Showing that it has a finite number of elements,
  - Providing a regular expression that defines it,
  - Constructing a FSA that accepts it, or
  - Exploiting closure properties
- Show a language is not regular by:
  - Using the pumping lemma, or
  - Exploiting closure properties.

But to use these tools effectively, we may also need domain knowledge (e.g., the Prime Number Theorem).

More Examples

$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7\}$
L = {w = the octal representation of a number that is divisible by 7}

Example elements of L:
7, 16 (14), 43 (35), 61 (49), 223 (147)

More Examples

Σ = {W, H, Q, E, S, T, B (measure bar)}
L = {w = w represents a song written in 4/4 time}

Example element of L:
WBWBHHBHQQBHHBQEEQEEB

More Examples

$\Sigma = \{0 - 9\}$
L = {w = is a prime Fermat number}

The Fermat numbers are defined by

$$F_n = 2^{2^n} + 1, n = 1, 2, 3, \ldots$$

Example elements of L:
$F_1 = 5, F_2 = 17, F_3 = 257, F_4 = 65{,}537$

Another Example

$\Sigma = \{0 - 9, *, =\}$
L = {w = a*b=c: a, b, c $\in$ $\{0\text{-}9\}^+$ and int(a) * int(b) = int(c)}

The Bottom Line

A language is regular if:

OR

The Bottom Line (Examples)

- The set of decimal representations for nonnegative integers divisible by 2 or 3
- The social security numbers of living US residents.
- Parity checking
- $a^n b^n$
- $a^j b^k$ where k>j
- a^k where k is prime
- The set of strings over {a, b} that contain an equal number of a's and b's.
- The octal representations of numbers that are divisible by 7
- The songs in 4/4 time
- The set of prime Fermat numbers

Decision Procedures

A **decision procedure** is an algorithm that answers a question (usually "yes" or "no") and terminates. The whole idea of a decision procedure itself raises a new class of questions. In particular, we can now ask,

1. Is there a decision procedure for question X?
2. What is that procedure?
3. How efficient is the best such procedure?

Clearly, if we jump immediately to an answer to question 2, we have our answer to question 1. But sometimes it makes sense to answer question 1 first. For one thing, it tells us whether to bother looking for answers to questions 2 and 3.

Examples of Question 1:

Is there a decision procedure, given a regular expression E and a string S, for determining whether S is in L(E)?

Is there a decision procedure, given a Turing machine T and an input string S, for determining whether T halts on S?

Decision Procedures for Regular Languages

Let M be a deterministic FSA. There is a decision procedure to determine whether:

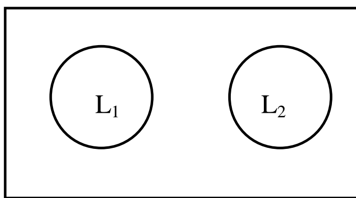
- $w \in L(M)$ for some fixed w
- L(M) is empty
- L(M) is finite
- L(M) is infinite

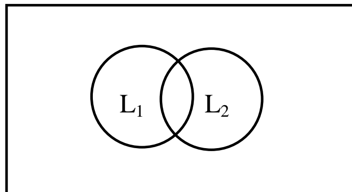
Let M_1 and M_2 be two deterministic FSAs. There is a decision procedure to determine whether M_1 and M_2 are equivalent. Let L_1 and L_2 be the languages accepted by M_1 and M_2. Then the language

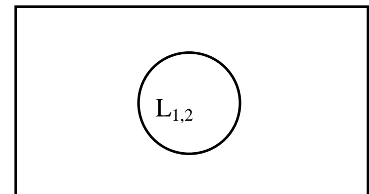
$$L = (L_1 \cap \neg L_2) \cup (\neg L_1 \cap L_2)$$

$$= (L_1 - L_2) \cup (L_2 - L_1)$$

must be regular. L is empty iff $L_1 = L_2$. There is a decision procedure to determine whether L is empty and thus whether $L_1 = L_2$ and thus whether M_1 and M_2 are equivalent.







A Review of Equivalence Relations

Do Homework 7.

A Review of Equivalence Relations

A relation R is an equivalence relation if it is: reflexive, symmetric, and transitive.

Example: R = the reflexive, symmetric, transitive closure of:
(Bob, Bill), (Bob, Butch), (Butch, Bud),
(Jim, Joe), (Joe, John), (Joe, Jared),
(Tim, Tom), (Tom, Tad)

An equivalence relation on a nonempty set A creates a partition of A. We write the elements of the partition as $[a_1]$, $[a_2]$, …
Example:

Another Equivalence Relation

Example: R = the reflexive, symmetric, transitive closure of:
(apple, pear), (pear, banana), (pear, peach),
(peas, mushrooms), (peas, onions), (peas, zucchini)
(bread, rice), (rice, potatoes), (rice, pasta)

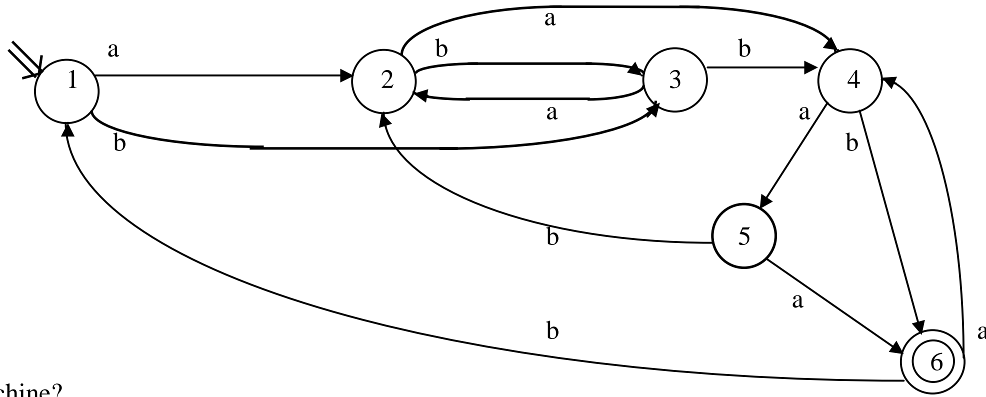
Partition:

State Minimization for DFAs

Read K & S 2.7
Do Homework 10.

State Minimization

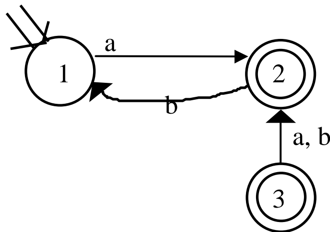
Consider:



Is this a minimal machine?

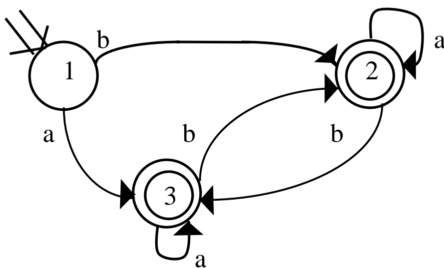
State Minimization

Step (1): Get rid of unreachable states.



State 3 is unreachable.

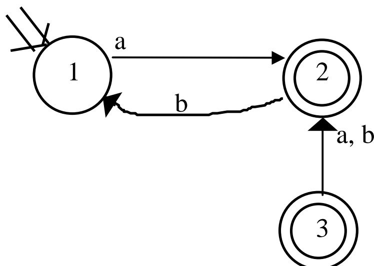
Step (2): Get rid of redundant states.



States 2 and 3 are redundant.

Getting Rid of Unreachable States

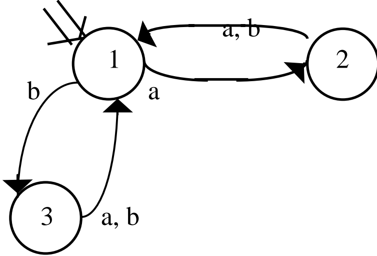
We can't easily find the unreachable states directly. But we can find the reachable ones and determine the unreachable ones from there. An algorithm for finding the reachable states:



Getting Rid of Redundant States

Intuitively, two states are equivalent to each other (and thus one is redundant) if all strings in Σ* have the same fate, regardless of which of the two states the machine is in. But how can we tell this?

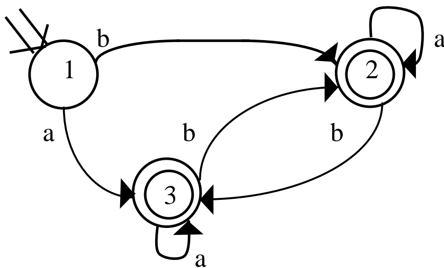
The simple case:



Two states have identical sets of transitions out.

Getting Rid of Redundant States

The harder case:



The outcomes are the same, even though the states aren't.

Finding an Algorithm for Minimization

Capture the notion of equivalence classes of strings with respect to a language.

Capture the (weaker) notion of equivalence classes of strings with respect to a language and a particular FSA.

Prove that we can always find a deterministic FSA with a number of states equal to the number of equivalence classes of strings.

Describe an algorithm for finding that deterministic FSA.

Defining Equivalence for Strings

We want to capture the notion that two strings are equivalent with respect to a language L if, no matter what is tacked on to them on the right, either they will both be in L or neither will. Why is this the right notion? Because it corresponds naturally to what the states of a recognizing FSM have to remember.

Example:

(1)	a	b	|	b	a	b
(2)	b	a	|	b	a	b

Suppose L = {w ∈ {a,b}* : |w| is even}. Are (1) and (2) equivalent?

Suppose L = {w ∈ {a,b}* : every a is immediately followed by b}. Are (1) and (2) equivalent?

Defining Equivalence for Strings

If two strings are equivalent with respect to L, we write $x \approx_L y$. Formally, $x \approx_L y$ if, $\forall z \in \Sigma^*$,

$xz \in L$ iff $yz \in L$.

Notice that $\approx_L$ is an equivalence relation.

Example:

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

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

ε	aa	bbb
a	bb	baa
b	aba	
	aab	

The equivalence classes of $\approx_L$:

$|\approx_L|$ is the number of equivalence classes of $\approx_L$.

Another Example of $\approx_L$

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

$L = \{w \in \Sigma^* : |w|$ is even$\}$

ε	bb	aabb
a	aba	bbaa
b	aab	aabaa
aa	bbb	
	baa	

The equivalence classes of $\approx_L$:

Yet Another Example of $\approx_L$

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

$L = aab^*a$

ε	ba	aabb
a	bb	aabaa
b	aaa	aabbba
aa	aba	aabbaa
ab	aab	
	bab	

The equivalence classes of $\approx_L$:

An Example of $\approx_L$ Where All Elements of L Are Not in the Same Equivalence Class

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

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

ε	bb	aabaa
a	aba	aabbba
b	aab	aabbaa
aa	baa	
	aabb	

The equivalence classes of $\approx_L$:

Is $|\approx_L|$ Always Finite?

$\Sigma = \{a, b\}$
$L = a^n b^n$

ε	aa	aaaa
a	aba	aaaaa
b	aaa	

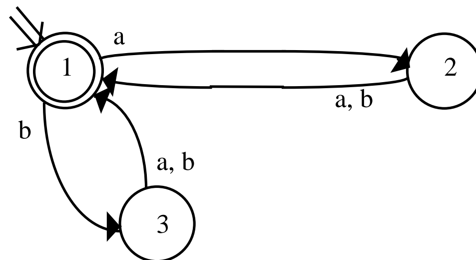
The equivalence classes of $\approx_L$:

Bringing FSMs into the Picture

$\approx_L$ is an ideal relation.

What if we now consider what happens to strings when they are being processed by a real FSM?

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



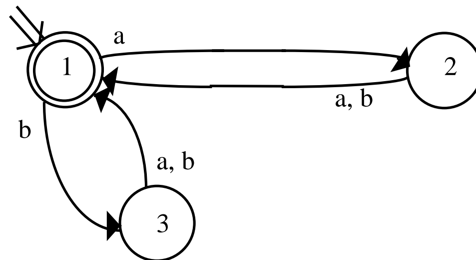
Define $\sim_M$ to relate pairs of strings that drive M from s to the same state.

Formally, if M is a deterministic FSM, then $x \sim_M y$ if there is some state q in M such that $(s, x) |\text{-}^*_M (q, \varepsilon)$ and $(s, y) |\text{-}^*_M (q, \varepsilon)$.

Notice that M is an equivalence relation.

An Example of ~M

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

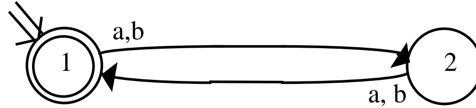


ε	bb	aabb
a	aba	bbaa
b	aab	aabaa
aa	bbb	
	baa	

The equivalence classes of $\sim_M$: $|\sim_M| =$

Another Example of ~M

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



ε	bb	aabb
a	aba	bbaa
b	aab	aabaa
aa	bbb	
	baa	

The equivalence classes of $\sim_M$: $|\sim_M| =$

The Relationship Between $\approx_L$ and $\sim_M$

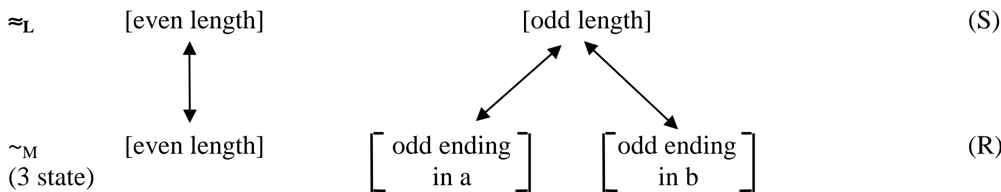
$\approx_{L:}$ [ε, aa, bb, aabb, bbaa] |w| is even
[a, b, aba, aab, bbb, baa, aabaa] |w| is odd

$\sim_M$, 3 state machine:
q1: [ε, aa, bb, aabb, bbaa] |w| is even
q2: [a, aba, baa, aabaa] $(ab \cup ba \cup aa \cup bb)^*a$
q3: [b, aab, bbb] $(ab \cup ba \cup aa \cup bb)^*b$

$\sim_M$, 2 state machine:
q1: [ε, aa, bb, aabb, bbaa] |w| is even
q2: [a, b, aba, aab, bbb, baa, aabaa] |w| is odd

$\sim_M$ is a refinement of $\approx_L$.

The Refinement



An equivalence relation R is a refinement of another one S iff

$xRy \rightarrow xSy$

In other words, R makes all the same distinctions S does, plus possibly more.

$|R| \geq |S|$

$\sim_M$ is a Refinement of $\approx_L$.

Theorem: For any deterministic finite automaton M and any strings x, y $\in \Sigma^*$, if x $\sim_M$ y, then x $\approx_L$ y.

Proof: If x $\sim_M$ y, then x and y drive m to the same state q. From q, any continuation string w will drive M to some state r. Thus xw and yw both drive M to r. Either r is a final state, in which case they both accept, or it is not, in which case they both reject. But this is exactly the definition of $\approx_L$.

Corollary: $|\sim_M| \geq |\approx_L|$.

Going the Other Way

When is this true?

If x $\approx_{L(M)}$ y then x $\sim_M$ y.

Finding the Minimal FSM for L

What's the smallest number of states we can get away with in a machine to accept L?

Example: $L = \{w \in \Sigma^* : |w| \text{ is even}\}$

The equivalence classes of $\approx_L$:

Minimal number of states for M(L) =

This follows directly from the theorem that says that, for any machine M that accepts L, $|\sim_M|$ must be at least as large as $|\approx_L|$.

Can we always find a machine with this minimal number of states?

The Myhill-Nerode Theorem

Theorem: Let L be a regular language. Then there is a deterministic FSA that accepts L and that has precisely $|\approx_L|$ states.

Proof: (by construction)

M = K states, corresponding to the equivalence classes of $\approx_L$.
s = [ε], the equivalence class of ε under $\approx_L$.
F = {[x] : x ∈ L}
δ([x], a) = [xa]

For this construction to prove the theorem, we must show:

1. K is finite.
2. δ is well defined, i.e., δ([x], a) = [xa] is independent of x.
3. L = L(M)

The Proof

(1) K is finite.
Since L is regular, there must exist a machine M, with $|\sim_M|$ finite. We know that

$$|\sim_M| \geq |\approx_L|$$

Thus $|\approx_L|$ is finite.
(2) δ is well defined.
This is assured by the definition of $\approx_L$, which groups together precisely those strings that have the same fate with respect to L.

The Proof, Continued

(3) L = L(M)
Suppose we knew that ([x], y) $|\text{-}_M^*$ ([xy], ε).
Now let [x] be [ε] and let s be a string in Σ^*.
Then

$$([\varepsilon], s) \vdash_M^* ([s], \varepsilon)$$

M will accept s if $[s] \in F$.
By the definition of F, $[s] \in F$ iff all strings in [s] are in L.
So M accepts precisely the strings in L.

The Proof, Continued

Lemma: ([x], y) $|\text{-}_M^*$ ([xy], ε)
By induction on |y|:
Trivial if |y| = 0.
Suppose true for |y| = n.
Show true for |y| = n+1

Let y = y'a, for some character a. Then,

|y'| = n

([x], y'a) $\vdash_M^*$ ([xy'], a)	(induction hypothesis)
([xy',] a) $\vdash_M^*$ ([xy'a], ε)	(definition of δ)
([x], y'a) $\vdash_M^*$ ([xy'a], ε)	(trans. of $\vdash_M^*$)
([x], y) $\vdash_M^*$ ([xy], ε)	(definition of y)

Another Version of the Myhill-Nerode Theorem

Theorem: A language is regular iff $|\approx_L|$ is finite.
Example:

Consider: $L = a^n b^n$

a, aa, aaa, aaaa, aaaaa …

Equivalence classes:

Proof:
Regular → $|\approx_L|$ is finite: If L is regular, then there exists an accepting machine M with a finite number of states N. We know that $N \geq |\approx_L|$. Thus $|\approx_L|$ is finite.

$|\approx_L|$ is finite → regular: If $|\approx_L|$ is finite, then the standard DFSA M_L accepts L. Since L is accepted by a FSA, it is regular.

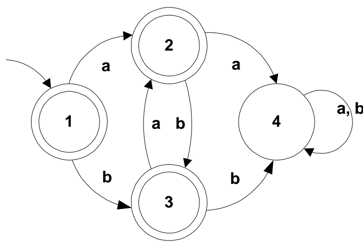
Constructing the Minimal DFA from $\approx_L$

$\Sigma = \{a, b\}$
L = {w ∈ {a, b}* : no two adjacent characters are the same}

The equivalence classes of $\approx_L$:

1: [ε]	ε
2: [a, ba, aba, baba, ababa, ...]	(b∪ε)(ab)*a
3: [b, ab, bab, abab, ...]	(a∪ε)(ba)*b
4: [bb, aa, bba, bbb, ...]	the rest

- Equivalence classes become states
- Start state is [ε]
- Final states are all equivalence classes in L
- δ([x], a) = [xa]



Using Myhill-Nerode to Prove that L is not Regular

L = {a^n : n is prime}

Consider:
ε
a
aa
aaa
aaaa

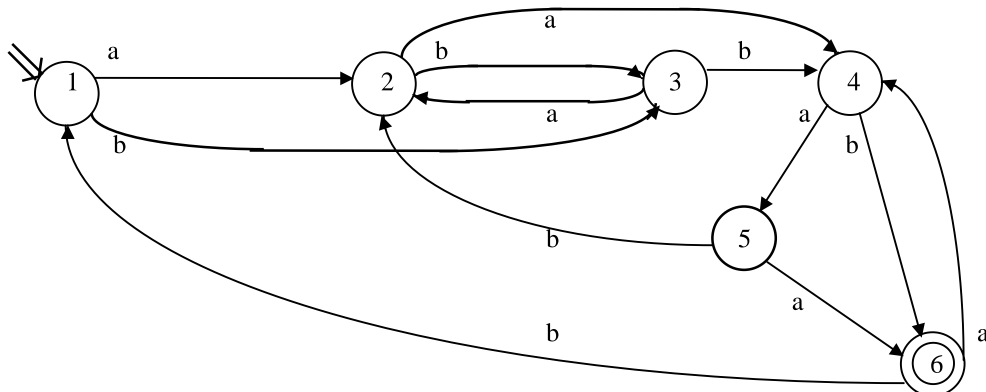
Equivalence classes:

So Where Do We Stand?

1. We know that for any regular language L there exists a minimal accepting machine M_L.
2. We know that |K| of M_L equals $|\approx_L|$.
3. We know how to construct M_L from $\approx_L$.

But is this good enough?

Consider:



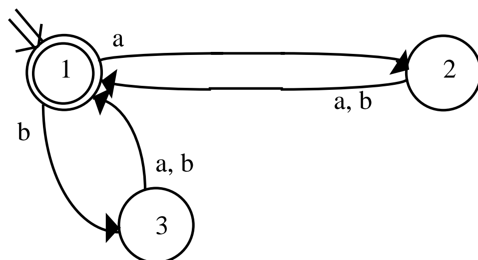
Constructing a Minimal FSA Without Knowing $\approx_L$

We want to take as input any DFSA M' that accepts L, and output a minimal, equivalent DFSA M.

What we need is a definition for "equivalent", i.e., mergeable states.

Define $q \equiv p$ iff for all strings $w \in \Sigma^*$, either w drives M to an accepting state from both q and p or it drives M to a rejecting state from both q and p.

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



Constructing $\equiv$ as the Limit of a Sequence of Approximating Equivalence Relations $\equiv_n$

(Where n is the length of the input strings that have been considered so far)

We'll consider input strings, starting with ε, and increasing in length by 1 at each iteration. We'll start by way overgrouping states. Then we'll split them apart as it becomes apparent (with longer and longer strings) that their behavior is not identical.

Initially, $\equiv_0$ has only two equivalence classes: [F] and [K - F], since on input ε, there are only two possible outcomes, accept or reject.

Next consider strings of length 1, i.e., each element of Σ. Split any equivalence classes of $\equiv_0$ that don't behave identically on all inputs. Note that in all cases, $\equiv_n$ is a refinement of $\equiv_{n-1}$.

Continue, until no splitting occurs, computing $\equiv_n$ from $\equiv_{n-1}$.

Constructing $\equiv$, Continued

More precisely, for any two states p and $q \in K$ and any $n \geq 1$, $q \equiv_n p$ iff:

1. $q \equiv_{n-1} p$, AND
2. for all $a \in \Sigma$, $\delta(p, a) \equiv_{n-1} \delta(q, a)$

The Construction Algorithm

The equivalence classes of $\equiv_0$ are F and K-F.

Repeat for n = 1, 2, 3 …

For each equivalence class C of $\equiv_{n-1}$ do

For each pair of elements p and q in C do

For each a in Σ do

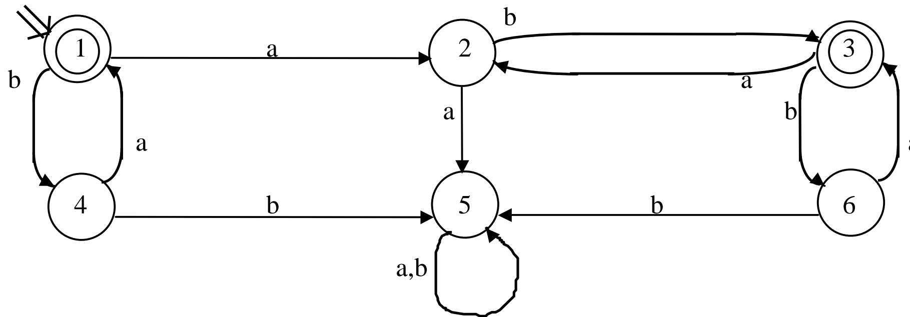
See if $\delta(p, a) \equiv_{n-1} \delta(q, a)$

If there are any differences in the behavior of p and q, then split them and create a new equivalence class.

Until $\equiv_n = \equiv_{n-1}$. $\equiv$ is this answer. Then use these equivalence classes to coalesce states.

An Example

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



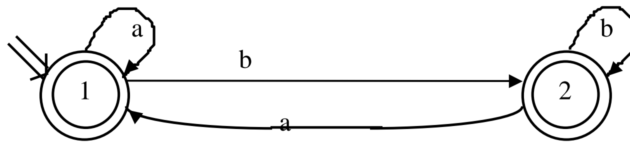
$\equiv_0 =$

$\equiv_1 =$

$\equiv_2 =$

Another Example

(a*b*)*



$\equiv_0 =$

$\equiv_1 =$

Minimal machine:

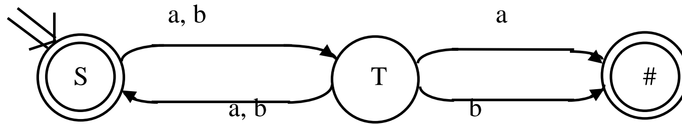
Another Example

Example: L={w ∈ {a, b}* : |w| is even}

((aa) ∪ (ab) ∪ (ba) ∪ (bb))*

S → ε
S → aT
S → bT

T → a
T → b
T → aS
T → bS



Convert to deterministic:
S = {s}
δ =

Another Example, Continued

Minimize:

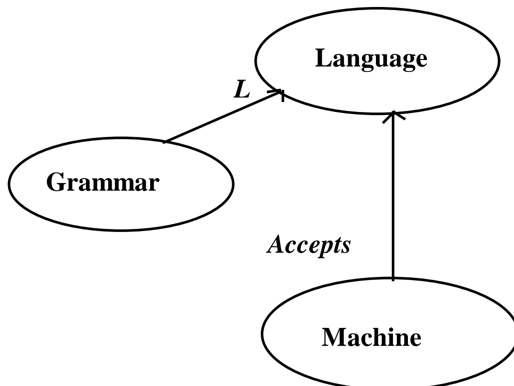
$\equiv_0$ =

$\equiv_1$ =

Minimal machine:

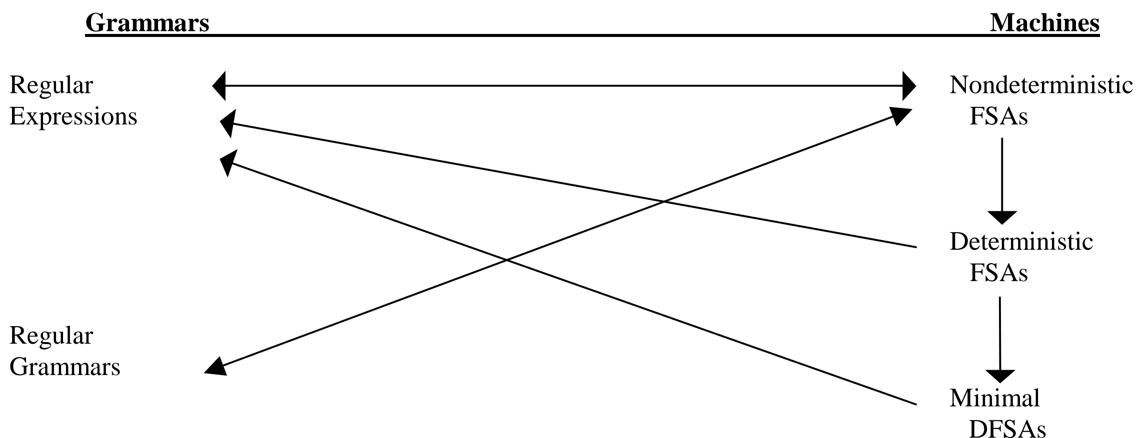
Summary of Regular Languages and Finite State Machines

Grammars, Languages, and Machines



Regular Grammars, Languages, and Machines

Most interesting languages are infinite. So we can't write them down. But we can write down finite grammars and finite machine specifications, and we can define algorithms for mapping between and among them.



What Does "Finite State" Really Mean?

There are two kinds of finite state problems:

- Those in which:
    - Some history matters.
    - Only a finite amount of history matters. In particular, it's often the case that we don't care what order things occurred in.

  Examples:
    - Parity
    - Money in a vending machine
    - Seat belt buzzer
- Those that are characterized by patterns.

  Examples:
    - Switching circuits:
        - Telephone
        - Railroad
    - Traffic lights
    - Lexical analysis
    - grep