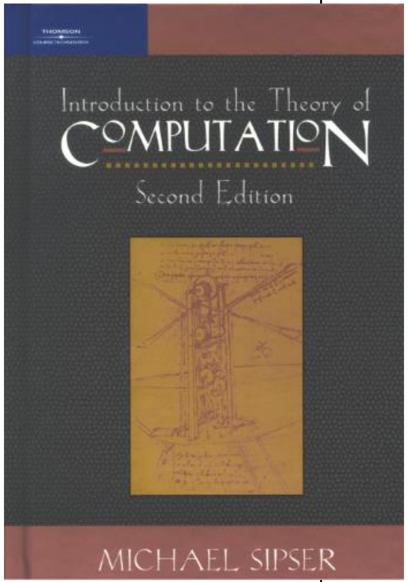
0

INTRODUCTION


THOMSON
Introduction to the Theory of
COMPUTATION
Second Edition
MICHAEL SIPSER

We begin with an overview of those areas in the theory of computation that we present in this course. Following that, you'll have a chance to learn and/or review some mathematical concepts that you will need later.

0.1

AUTOMATA, COMPUTABILITY, AND COMPLEXITY

This book focuses on three traditionally central areas of the theory of computation: automata, computability, and complexity. They are linked by the question:

What are the fundamental capabilities and limitations of computers?

This question goes back to the 1930s when mathematical logicians first began to explore the meaning of computation. Technological advances since that time have greatly increased our ability to compute and have brought this question out of the realm of theory into the world of practical concern.

In each of the three areas—automata, computability, and complexity—this question is interpreted differently, and the answers vary according to the interpretation. Following this introductory chapter, we explore each area in a separate part of this book. Here, we introduce these parts in reverse order because starting from the end you can better understand the reason for the beginning.

lem of determining whether a mathematical statement is true or false. This task is the bread and butter of mathematicians. It seems like a natural for solution by computer because it lies strictly within the realm of mathematics. But no computer algorithm can perform this task.

Among the consequences of this profound result was the development of ideas concerning theoretical models of computers that eventually would help lead to the construction of actual computers.

The theories of computability and complexity are closely related. In complexity theory, the objective is to classify problems as easy ones and hard ones, whereas in computability theory the classification of problems is by those that are solvable and those that are not. Computability theory introduces several of the concepts used in complexity theory.

AUTOMATA THEORY

Automata theory deals with the definitions and properties of mathematical models of computation. These models play a role in several applied areas of computer science. One model, called the *finite automaton*, is used in text processing, compilers, and hardware design. Another model, called the *context-free grammar*, is used in programming languages and artificial intelligence.

Automata theory is an excellent place to begin the study of the theory of computation. The theories of computability and complexity require a precise definition of a *computer*. Automata theory allows practice with formal definitions of computation as it introduces concepts relevant to other nontheoretical areas of computer science.

0.2

MATHEMATICAL NOTIONS AND TERMINOLOGY

As in any mathematical subject, we begin with a discussion of the basic mathematical objects, tools, and notation that we expect to use.

SETS

A ***set*** is a group of objects represented as a unit. Sets may contain any type of object, including numbers, symbols, and even other sets. The objects in a set are called its ***elements*** or ***members***. Sets may be described formally in several ways. One way is by listing a set's elements inside braces. Thus the set

$$\{7, 21, 57\}$$

contains the elements 7, 21, and 57. The symbols $\in$ and $\notin$ denote set membership and nonmembership. We write $7 \in \{7, 21, 57\}$ and $8 \notin \{7, 21, 57\}$. For two sets A and B, we say that A is a ***subset*** of B, written $A \subseteq B$, if every member of

Formal Languages

- Alphabet: a finite set of symbols $\Sigma = \{a,b\}$
- String: a finite sequence of symbols ababbaab
- Language: a (possibly ∞) set of strings $L=\{a,aa,aaa,\ldots\}$
- String length: number of symbols in it $|aba|=3$
- Empty string: ε or ^ $(|\varepsilon|=0)$ $\forall w \quad w \bullet \varepsilon = \varepsilon \bullet w = w$
- String concatenation: w_1w_2 $ab \cdot ba = abba$
- Language concatenation: $\{1,2\} \cdot \{a,b,\ldots\}$

  $L_1L_2=\{w_1w_2 \mid w_1 \in L_1, w_2 \in L_2\}$ $=\{1a,2a,1b,2b,\ldots\}$
- String exponentiation: $w^k = ww\ldots w$ (k times) $a^3=aaa$
- Language exponentiation: $L^k = LL\ldots L$ (k times) $\{0,1\}^{32}$

  $LL = L^2$ $L^k=LL^{k-1}$ $L^0=\{\varepsilon\}$

Formal Languages

- String reversal: w^R $(aabc)^R=cbaa$
- Language reversal: $L^R=\{w^R \mid w\in L\}$ $\{ab,cd\}^R=\{ba,dc\}$
- Language union: ⇒ set union
  $L_1 \cup L_2=\{w \mid w\in L_1 \text{ or } w\in L_2\}$ $\{a\}\cup\{b,aa\}=\{a,b,aa\}$
- Language intersection: ⇒ set intersection
  $L_1 \cap L_2=\{w \mid w\in L_1 \text{ and } w\in L_2\}$ $\{a,b\}\cap\{b,c\}=\{b\}$
- Language difference: ⇒ set difference
  $L_1 - L_2=\{w \mid w\in L_1 \text{ and } w\notin L_2\}$ $\{a,b\}-\{b,d\}=\{a\}$
- Kleene closure: $L^* = L^0 \cup L^1 \cup L^2 \cup \ldots$ $\{a\}^*=\{\varepsilon,a,aa,\ldots\}$
  $L^+ = L^1 \cup L^2 \cup L^3 \cup \ldots$ $\{a\}^+=\{a,aa,\ldots\}$
- All finite strings (over Σ): Σ^* $L\subseteq\Sigma^* \ \forall L$ $\{\varepsilon,a,aa,aaa,\ldots\}$

Theorem: Σ^* contains no ∞ strings. only finite strings in Σ^i

Formal Languages

Language complementation: $L' = \Sigma^* - L$ “negation” w.r.t. Σ^*

Theorem: $(L^*)^* = L^*$ $L^* \subseteq (L^*)^*$ & $(L^*)^* \subseteq L^*$

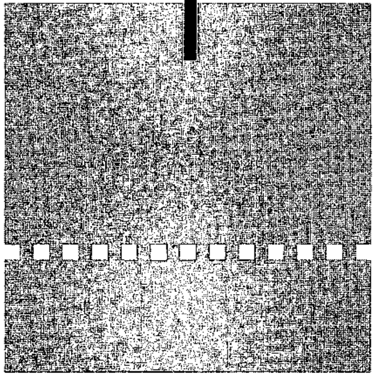
Theorem: $L^+ = LL^*$

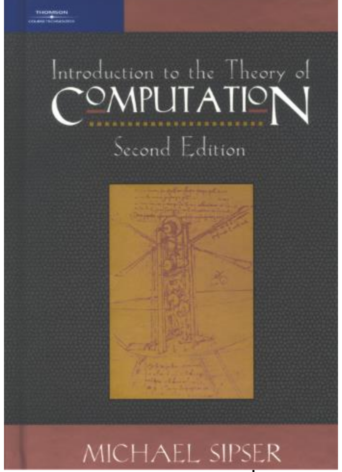
- “Trivial” language: $\{\varepsilon\}$ $\{\varepsilon\} \bullet L = L \bullet \{\varepsilon\} = L$
- Empty language: $\emptyset$ $\emptyset^* = \{\varepsilon\}$

Theorem: Σ^* is countable, $|\Sigma^*| = |\mathbb{N}|$ dovetailing

Theorem: 2^{Σ^*} is uncountable. diagonalization

1





Introduction to the Theory of
COMPUTATION
Second Edition

MICHAEL SIPSER

REGULAR LANGUAGES

The theory of computation begins with a question: What is a computer? It is perhaps a silly question, as everyone knows that this thing I type on is a computer. But these real computers are quite complicated—too much so to allow us to set up a manageable mathematical theory of them directly. Instead we use an idealized computer called a ***computational model***. As with any model in science, a computational model may be accurate in some ways but perhaps not in others. Thus we will use several different computational models, depending on the features we want to focus on. We begin with the simplest model, called the ***finite state machine*** or ***finite automaton***.

1.1

FINITE AUTOMATA

Finite automata are good models for computers with an extremely limited amount of memory. What can a computer do with such a small memory? Many useful things! In fact, we interact with such computers all the time, as they lie at the heart of various electromechanical devices.

The controller for an automatic door is one example of such a device. Often found at supermarket entrances and exits, automatic doors swing open when sensing that a person is approaching. An automatic door has a pad in front to

FORMAL DEFINITION OF A FINITE AUTOMATON

In the preceding section we used state diagrams to introduce finite automata. Now we define finite automata formally. Although state diagrams are easier to grasp intuitively, we need the formal definition, too, for two specific reasons.

First, a formal definition is precise. It resolves any uncertainties about what is allowed in a finite automaton. If you were uncertain about whether finite automata were allowed to have 0 accept states or whether they must have exactly one transition exiting every state for each possible input symbol, you could consult the formal definition and verify that the answer is yes in both cases. Second, a formal definition provides notation. Good notation helps you think and express your thoughts clearly.

The language of a formal definition is somewhat arcane, having some similarity to the language of a legal document. Both need to be precise, and every detail must be spelled out.

A finite automaton has several parts. It has a set of states and rules for going from one state to another, depending on the input symbol. It has an input alphabet that indicates the allowed input symbols. It has a start state and a set of accept states. The formal definition says that a finite automaton is a list of those five objects: set of states, input alphabet, rules for moving, start state, and accept states. In mathematical language a list of five elements is often called a 5-tuple. Hence we define a finite automaton to be a 5-tuple consisting of these five parts.

We use something called a ***transition function***, frequently denoted δ, to define the rules for moving. If the finite automaton has an arrow from a state x to a state y labeled with the input symbol 1, that means that, if the automaton is in state x when it reads a 1, it then moves to state y. We can indicate the same thing with the transition function by saying that $\delta(x, 1) = y$. This notation is a kind of mathematical shorthand. Putting it all together we arrive at the formal definition of finite automata.

DEFINITION 1.5

A ***finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the ***states***,
2. Σ is a finite set called the ***alphabet***,
3. $\delta \colon Q \times \Sigma \longrightarrow Q$ is the ***transition function***,[1]
4. $q_0 \in Q$ is the ***start state***, and
5. $F \subseteq Q$ is the ***set of accept states***.[2]

---

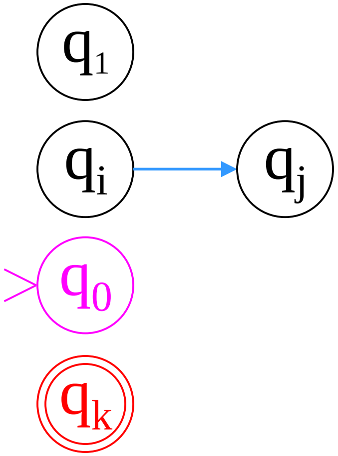
[1] Refer back to page 7 if you are uncertain about the meaning of $\delta \colon Q \times \Sigma \longrightarrow Q$.

[2] Accept states sometimes are called ***final states***.

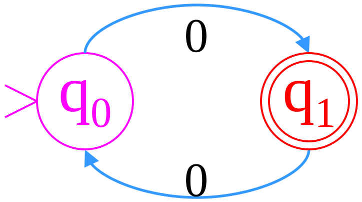
Finite Automata

Basic idea: a FA is a “machine” that changes states while processing symbols, one at a time.

- Finite set of states: $Q = \{q_0, q_1, q_3, \ldots, q_k\}$
- Transition function: $\delta: Q \times \Sigma \rightarrow Q$
- Initial state: $q_0 \in Q$
- Final states: $F \subseteq Q$
- Finite automaton is $M=(Q, \Sigma, \delta, q_0, F)$



Ex: an FA that accepts all odd-length strings of zeros:



$M=(\{q_0,q_1\}, \{0\}, \{((q_0,0),q_1), ((q_1,0),q_0)\}, q_0, \{q_1\})$

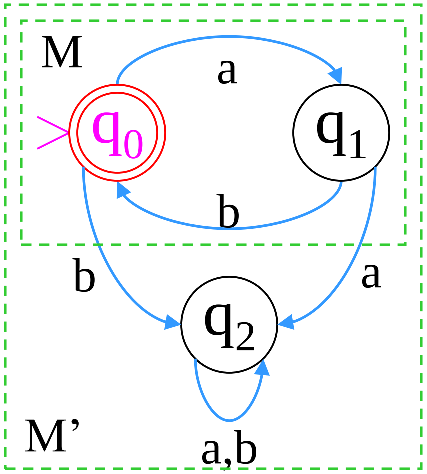
Finite Automata

FA operation: consume a string $w \in \Sigma^*$ one symbol at a time while changing states

Acceptance: end up in a final state

Rejection: anything else (including hang-up / crash)

Ex: FA that accepts all strings of form ababab…= $(ab)^*$



$M=(\{q_0,q_1\}, \{a,b\}, \{((q_0,a),q_1), ((q_1,b),q_0)\}, q_0, \{q_0\})$

But M "crashes" on input string "abba"!
Solution: add dead-end state to fully specify M

$M'=(\{q_0,q_1,q_2\}, \{a,b\}, \{ ((q_0,a),q_1), ((q_1,b),q_0), ((q_0,b),q_2), ((q_1,a),q_2). ((q_2,a),q_2), ((q_2,b),q_2) \}, q_0, \{q_0\})$

Finite Automata

Transition function δ extends from symbols to strings:

$\delta: Q \times \Sigma^* \to Q$ $\qquad \delta(q_0, wx) = \delta(\delta(q_0, w), x)$

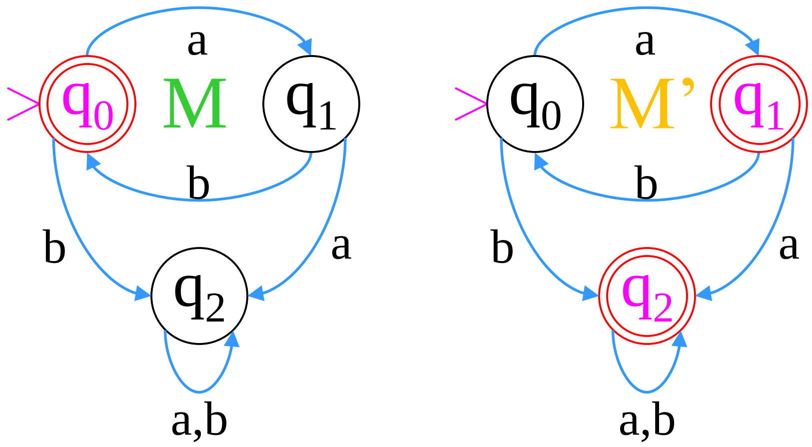
where $\delta(q_i, \varepsilon) = q_i$

Language of M is $L(M)=\{w \in \Sigma^* \mid \delta(q_0, w) \in F\}$

Definition: language is regular iff it is accepted by some FA.

Theorem: Complementation preserves regularity.

Proof: Invert final and non-final states in fully specified FA.



$L=L(M)=(ab)^*$

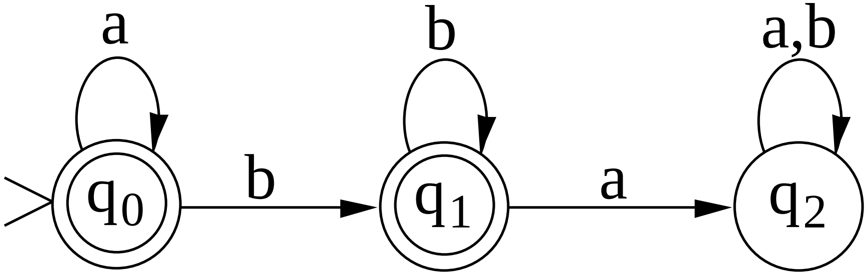
$L'=L(M')= b(a+b)^* + (a+b)^*a$
$\qquad + (a+b)^*(aa+bb)(a+b)^*$

Why are these complements?

M' "simulates" M and
does the opposite!

Problem: design a DFA that accepts all strings over {a,b} where any a’s precede any b’s.

Idea: skip over any contiguous a’s, then skip over any b’s, and then accept iff the end is reached.



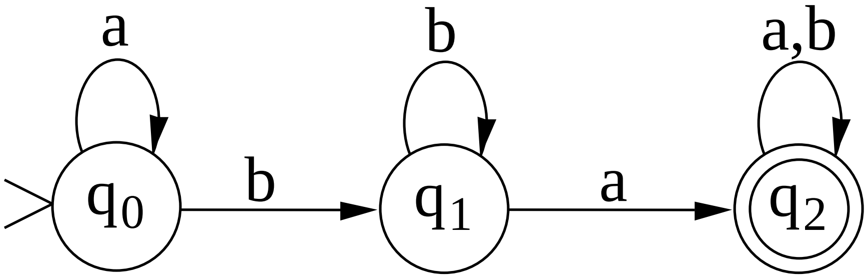
L = a*b*

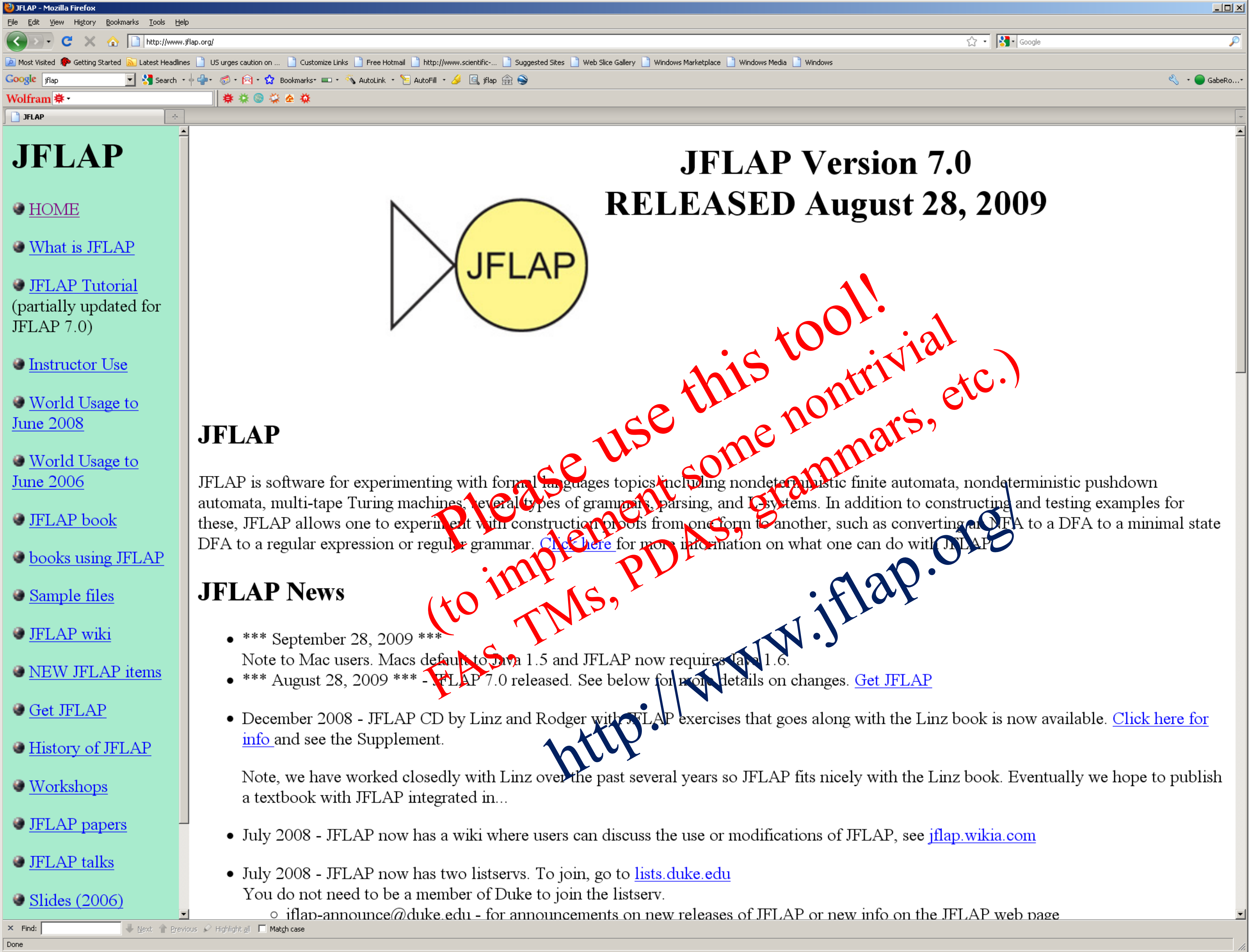
Q: What is the complement of L?

Problem: what is the complement of $L = a^*b^*$?

Idea: write a regular expression and then simplify.

$$\begin{aligned} L' &= (a+b)^*b^+(a+b)^*a^+(a+b)^* \\ &= (a+b)^*b(a+b)^*a(a+b)^* \\ &= (a+b)^*b^+a(a+b)^* \\ &= (a+b)^*ba(a+b)^* \\ &= a^*b^+a(a+b)^* \end{aligned}$$





JFLAP

- HOME
- What is JFLAP
- JFLAP Tutorial (partially updated for JFLAP 7.0)
- Instructor Use
- World Usage to June 2008
- World Usage to June 2006
- JFLAP book
- books using JFLAP
- Sample files
- JFLAP wiki
- NEW JFLAP items
- Get JFLAP
- History of JFLAP
- Workshops
- JFLAP papers
- JFLAP talks
- Slides (2006)

JFLAP Version 7.0
RELEASED August 28, 2009

JFLAP

JFLAP is software for experimenting with formal languages topics including nondeterministic finite automata, nondeterministic pushdown automata, multi-tape Turing machines, several types of grammars, parsing, and L-systems. In addition to constructing and testing examples for these, JFLAP allows one to experiment with construction proofs from one form to another, such as converting an NFA to a DFA to a minimal state DFA to a regular expression or regular grammar. Click here for more information on what one can do with JFLAP.

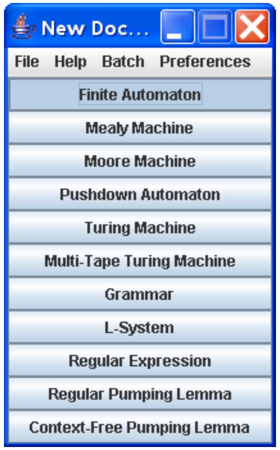
JFLAP News

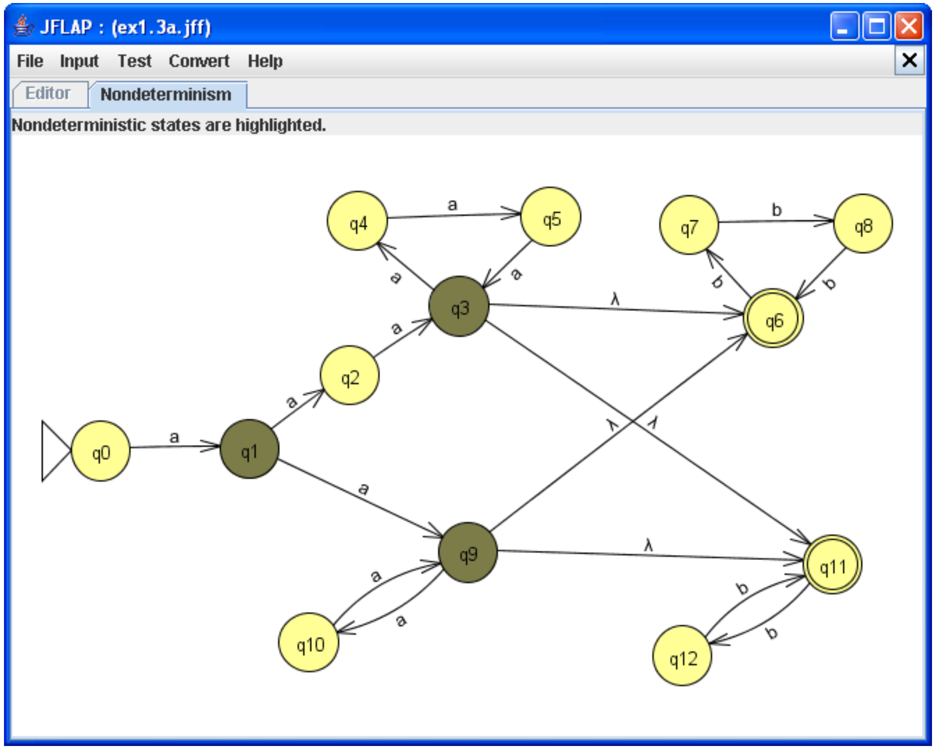
- *** September 28, 2009 ***
  Note to Mac users. Macs default to Java 1.5 and JFLAP now requires Java 1.6.
- *** August 28, 2009 *** - JFLAP 7.0 released. See below for more details on changes. Get JFLAP
- December 2008 - JFLAP CD by Linz and Rodger with JFLAP exercises that goes along with the Linz book is now available. Click here for info and see the Supplement.

  Note, we have worked closedly with Linz over the past several years so JFLAP fits nicely with the Linz book. Eventually we hope to publish a textbook with JFLAP integrated in...
- July 2008 - JFLAP now has a wiki where users can discuss the use or modifications of JFLAP, see jflap.wikia.com
- July 2008 - JFLAP now has two listservs. To join, go to lists.duke.edu
  You do not need to be a member of Duke to join the listserv.
  - jflap-announce@duke.edu - for announcements on new releases of JFLAP or new info on the JFLAP web page

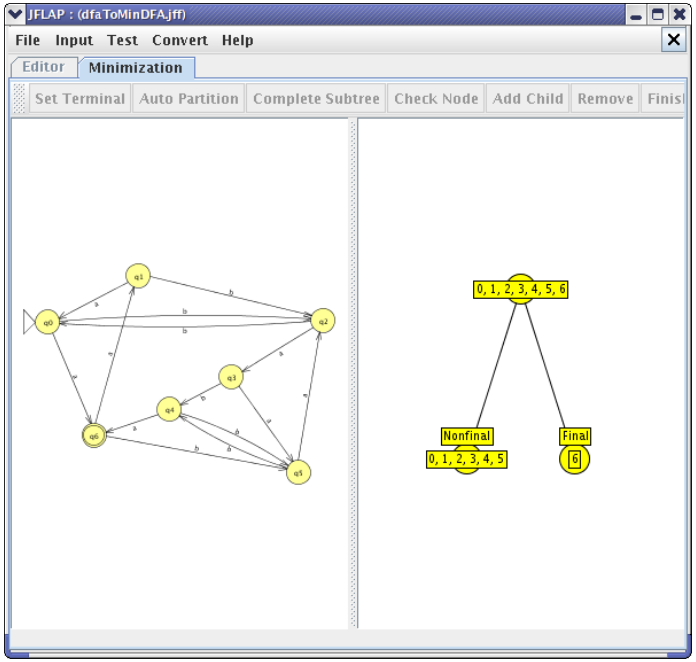
Please use this tool!
(to implement some nontrivial FAs, TMs, PDAs, grammars, etc.)

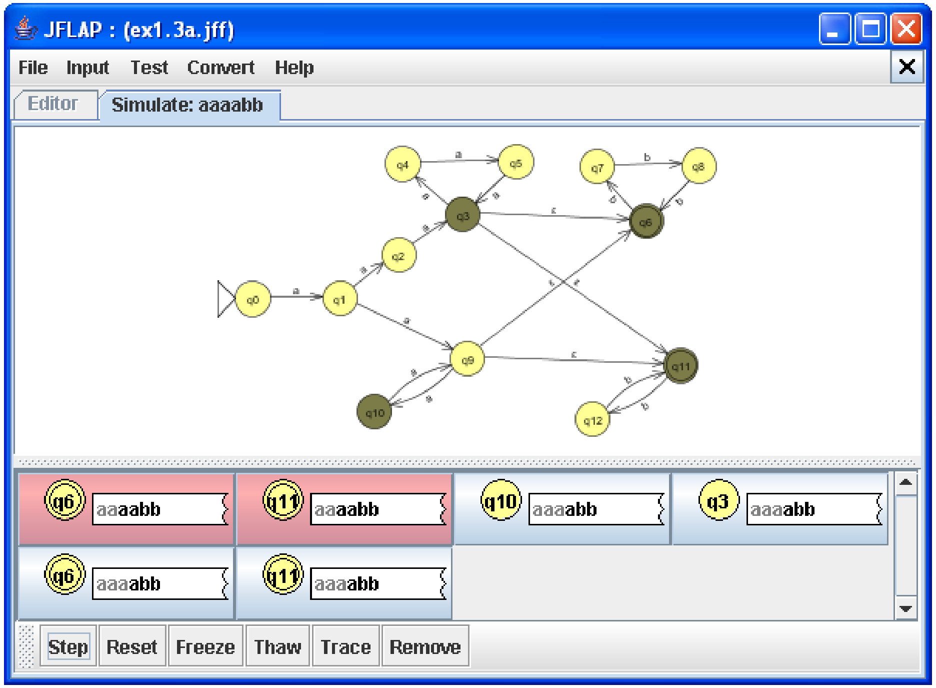
http://www.jflap.org/

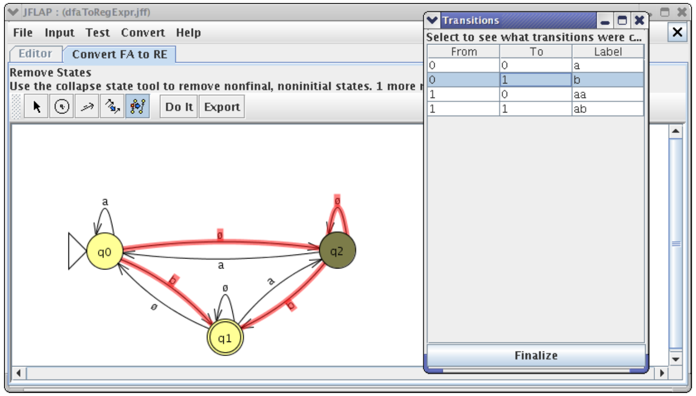




Nondeterministic states are highlighted.





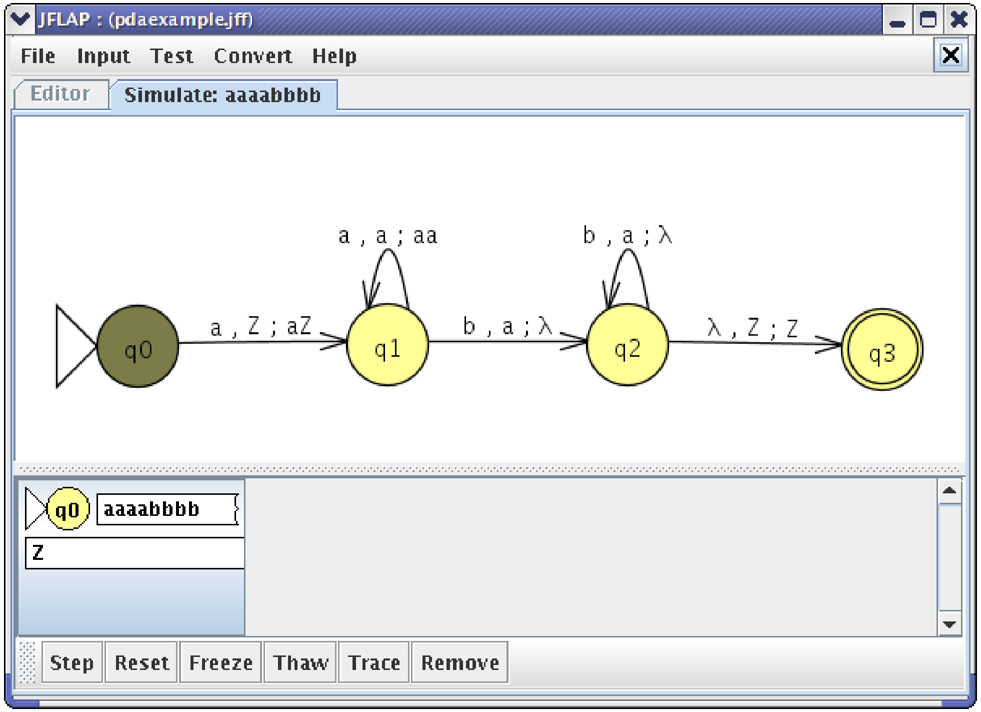


Remove States

Use the collapse state tool to remove nonfinal, noninitial states. 1 more r

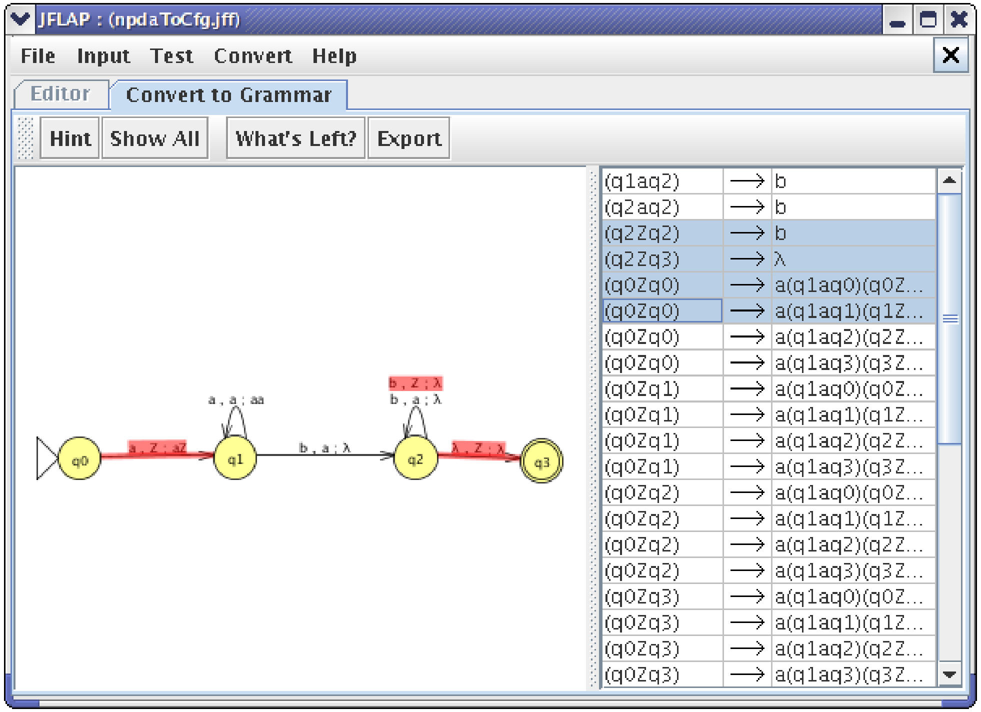
Select to see what transitions were c...

From	To	Label
0	0	a
0	1	b
1	0	aa
1	1	ab

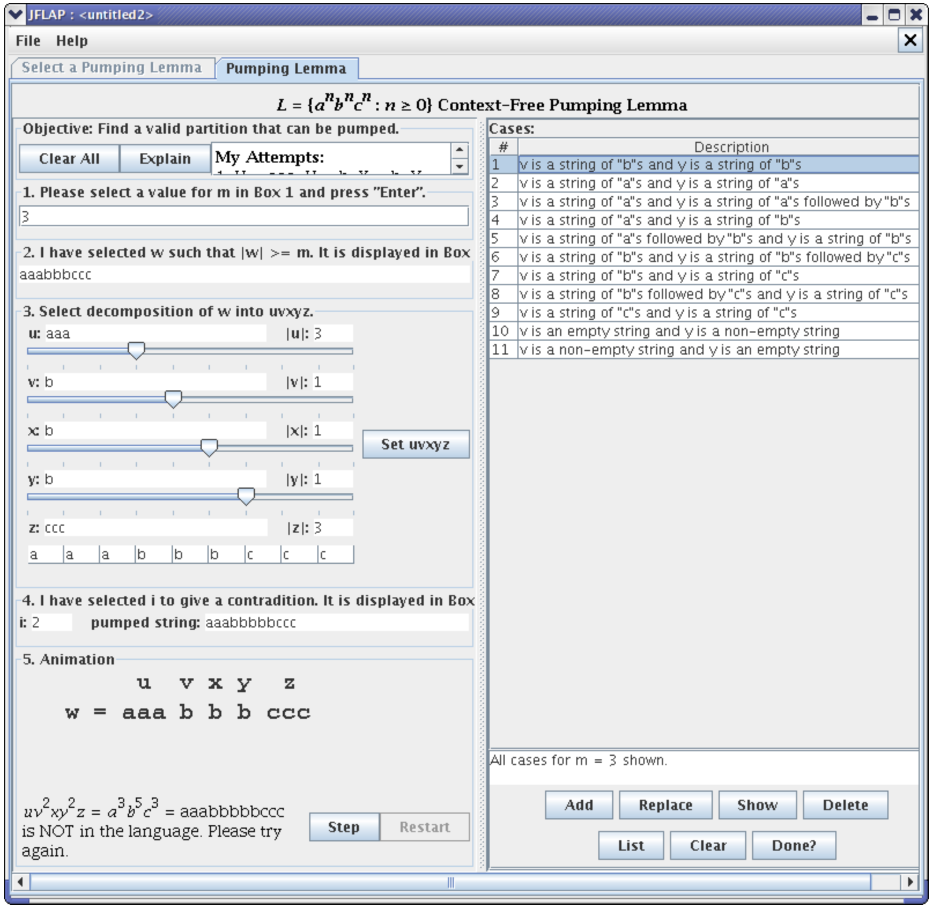


q0 aaaabbbb

Z



(q1aq2)	⟶	b
(q2aq2)	⟶	b
(q2Zq2)	⟶	b
(q2Zq3)	⟶	λ
(q0Zq0)	⟶	a(q1aq0)(q0Z...
(q0Zq0)	⟶	a(q1aq1)(q1Z...
(q0Zq0)	⟶	a(q1aq2)(q2Z...
(q0Zq0)	⟶	a(q1aq3)(q3Z...
(q0Zq1)	⟶	a(q1aq0)(q0Z...
(q0Zq1)	⟶	a(q1aq1)(q1Z...
(q0Zq1)	⟶	a(q1aq2)(q2Z...
(q0Zq1)	⟶	a(q1aq3)(q3Z...
(q0Zq2)	⟶	a(q1aq0)(q0Z...
(q0Zq2)	⟶	a(q1aq1)(q1Z...
(q0Zq2)	⟶	a(q1aq2)(q2Z...
(q0Zq2)	⟶	a(q1aq3)(q3Z...
(q0Zq3)	⟶	a(q1aq0)(q0Z...
(q0Zq3)	⟶	a(q1aq1)(q1Z...
(q0Zq3)	⟶	a(q1aq2)(q2Z...
(q0Zq3)	⟶	a(q1aq3)(q3Z...



$L = \{a^n b^n c^n : n \geq 0\}$ Context-Free Pumping Lemma

Objective: Find a valid partition that can be pumped.

My Attempts:

1. Please select a value for m in Box 1 and press "Enter".

3

2. I have selected w such that |w| >= m. It is displayed in Box

aaabbbccc

3. Select decomposition of w into uvxyz.

u: aaa |u|: 3

v: b |v|: 1

x: b |x|: 1

y: b |y|: 1

z: ccc |z|: 3

a a a b b b c c c

4. I have selected i to give a contradition. It is displayed in Box

i: 2 pumped string: aaabbbbbccc

5. Animation

u v x y z

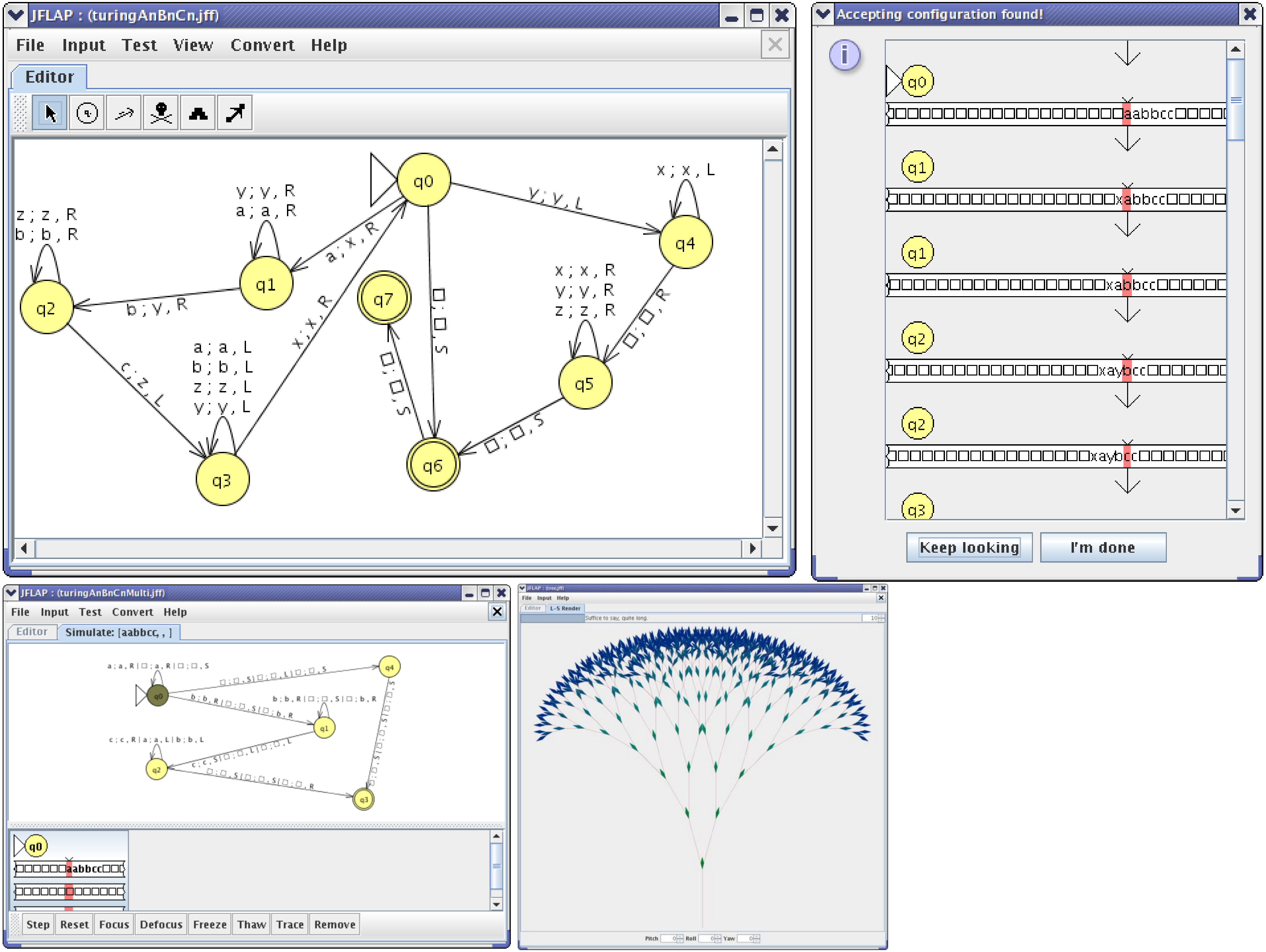
w = aaa b b b ccc

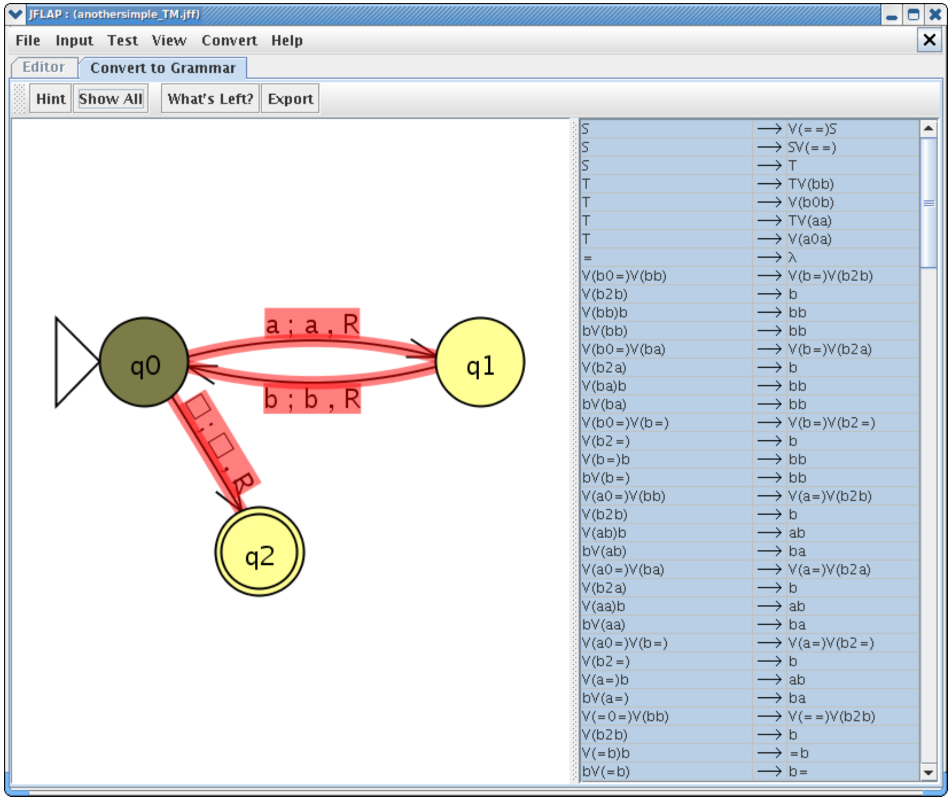
$uv^2xy^2z = a^3b^5c^3$ = aaabbbbbccc is NOT in the language. Please try again.

Cases:

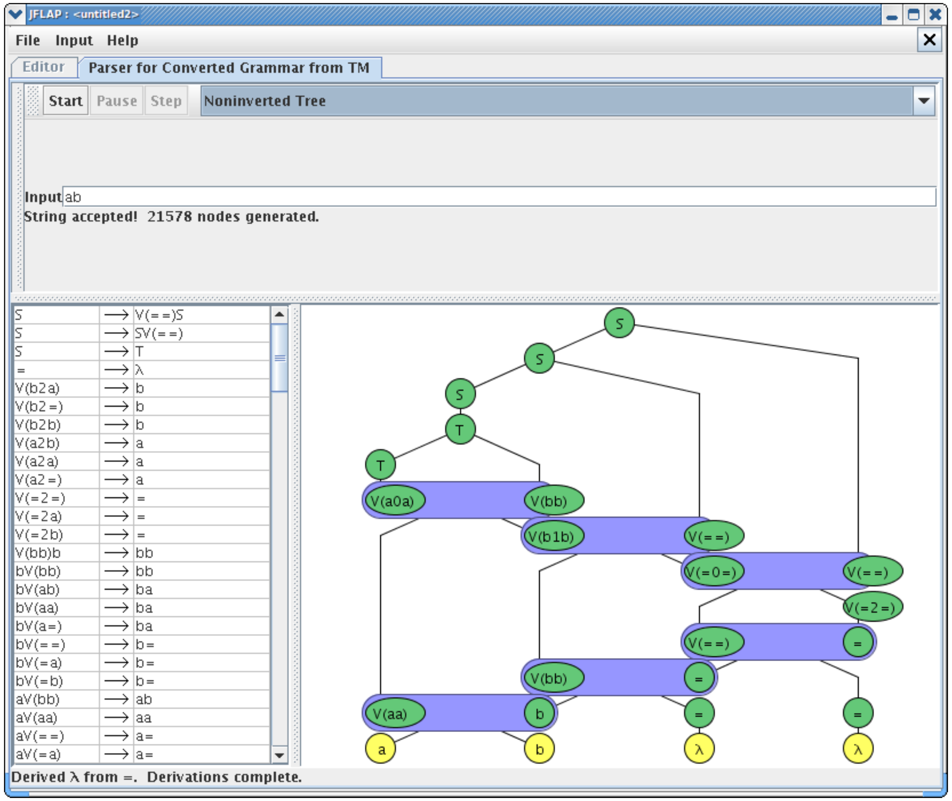
#	Description
1	v is a string of "b"s and y is a string of "b"s
2	v is a string of "a"s and y is a string of "a"s
3	v is a string of "a"s and y is a string of "a"s followed by "b"s
4	v is a string of "a"s and y is a string of "b"s
5	v is a string of "a"s followed by "b"s and y is a string of "b"s
6	v is a string of "b"s and y is a string of "b"s followed by "c"s
7	v is a string of "b"s and y is a string of "c"s
8	v is a string of "b"s followed by "c"s and y is a string of "c"s
9	v is a string of "c"s and y is a string of "c"s
10	v is an empty string and y is a non-empty string
11	v is a non-empty string and y is an empty string

All cases for m = 3 shown.


JFLAP : (turingAnBnCn.jff)
File Input Test View Convert Help
Editor
q0
q1
q2
q3
q4
q5
q6
q7
y ; y , R
a ; a , R
z ; z , R
b ; b , R
a ; x , R
b ; y , R
c ; z , L
a ; a , L
b ; b , L
z ; z , L
y ; y , L
x ; x , R
y ; y , L
x ; x , L
□ ; □ , S
□ ; □ , R
x ; x , R
y ; y , R
z ; z , R
□ ; □ , S
□ ; □ , S
Accepting configuration found!
Keep looking
I'm done
JFLAP : (turingAnBnCnMulti.jff)
File Input Test Convert Help
Editor
Simulate: [aabbcc, ,]
Step
Reset
Focus
Defocus
Freeze
Thaw
Trace
Remove
JFLAP : (tree.jff)
File Input Help
Editor
L-S Render
Suffice to say, quite long.
Pitch
Roll
Yaw



S	⟶	V(==)S
S	⟶	SV(==)
S	⟶	T
T	⟶	TV(bb)
T	⟶	V(b0b)
T	⟶	TV(aa)
T	⟶	V(a0a)
=	⟶	λ
V(b0=)V(bb)	⟶	V(b=)V(b2b)
V(b2b)	⟶	b
V(bb)b	⟶	bb
bV(bb)	⟶	bb
V(b0=)V(ba)	⟶	V(b=)V(b2a)
V(b2a)	⟶	b
V(ba)b	⟶	bb
bV(ba)	⟶	bb
V(b0=)V(b=)	⟶	V(b=)V(b2=)
V(b2=)	⟶	b
V(b=)b	⟶	bb
bV(b=)	⟶	bb
V(a0=)V(bb)	⟶	V(a=)V(b2b)
V(b2b)	⟶	b
V(ab)b	⟶	ab
bV(ab)	⟶	ba
V(a0=)V(ba)	⟶	V(a=)V(b2a)
V(b2a)	⟶	b
V(aa)b	⟶	ab
bV(aa)	⟶	ba
V(a0=)V(b=)	⟶	V(a=)V(b2=)
V(b2=)	⟶	b
V(a=)b	⟶	ab
bV(a=)	⟶	ba
V(=0=)V(bb)	⟶	V(==)V(b2b)
V(b2b)	⟶	b
V(=b)b	⟶	=b
bV(=b)	⟶	b=

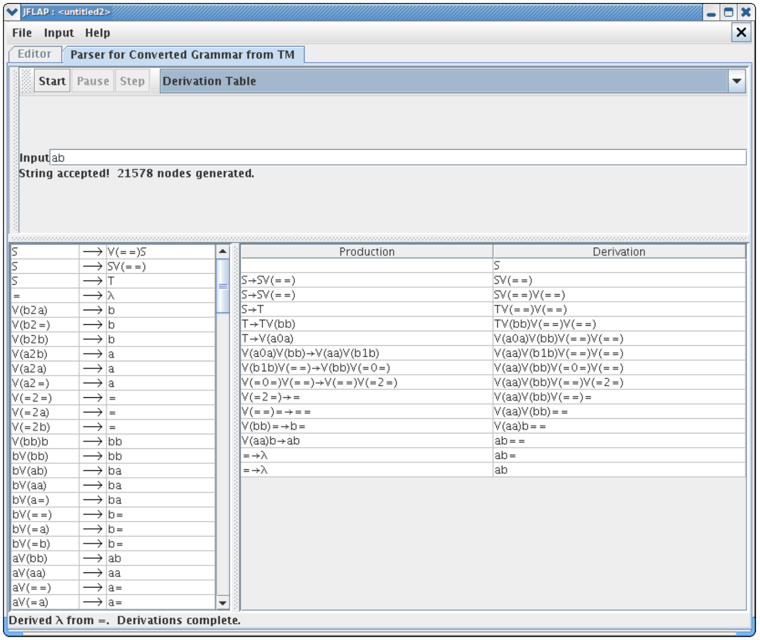


Input ab

String accepted! 21578 nodes generated.

S	⟶	V(==)S
S	⟶	SV(==)
S	⟶	T
=	⟶	λ
V(b2a)	⟶	b
V(b2=)	⟶	b
V(b2b)	⟶	b
V(a2b)	⟶	a
V(a2a)	⟶	a
V(a2=)	⟶	a
V(=2=)	⟶	=
V(=2a)	⟶	=
V(=2b)	⟶	=
V(bb)b	⟶	bb
bV(bb)	⟶	bb
bV(ab)	⟶	ba
bV(aa)	⟶	ba
bV(a=)	⟶	ba
bV(==)	⟶	b=
bV(=a)	⟶	b=
bV(=b)	⟶	b=
aV(bb)	⟶	ab
aV(aa)	⟶	aa
aV(==)	⟶	a=
aV(=a)	⟶	a=

Derived λ from =. Derivations complete.



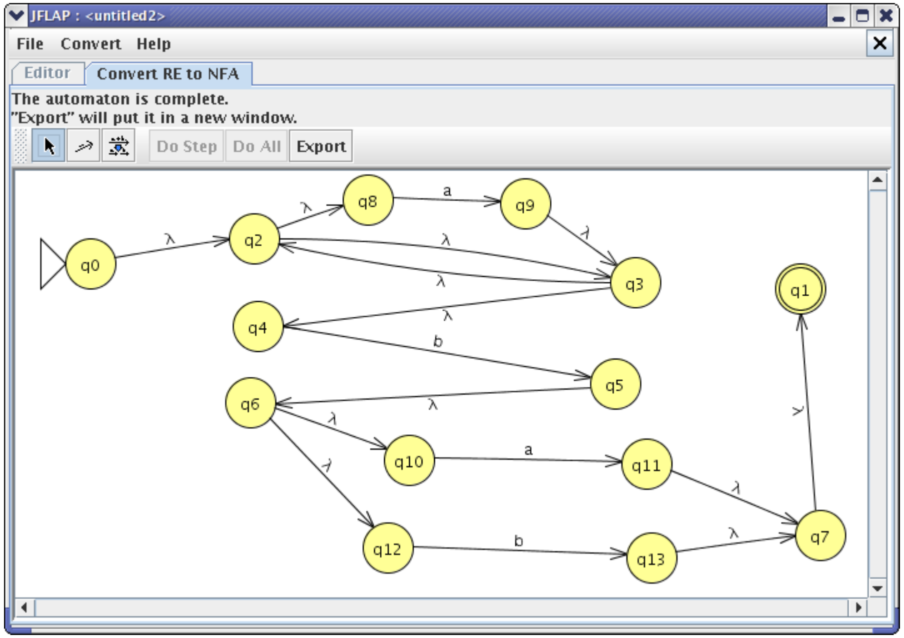
Input ab

String accepted! 21578 nodes generated.

S	⟶	V(==)S
S	⟶	SV(==)
S	⟶	T
=	⟶	λ
V(b2a)	⟶	b
V(b2=)	⟶	b
V(b2b)	⟶	b
V(a2b)	⟶	a
V(a2a)	⟶	a
V(a2=)	⟶	a
V(=2=)	⟶	=
V(=2a)	⟶	=
V(=2b)	⟶	=
V(bb)b	⟶	bb
bV(bb)	⟶	bb
bV(ab)	⟶	ba
bV(aa)	⟶	ba
bV(a=)	⟶	ba
bV(==)	⟶	b=
bV(=a)	⟶	b=
bV(=b)	⟶	b=
aV(bb)	⟶	ab
aV(aa)	⟶	aa
aV(==)	⟶	a=
aV(=a)	⟶	a=

Production	Derivation
	S
S→SV(==)	SV(==)
S→SV(==)	SV(==)V(==)
S→T	TV(==)V(==)
T→TV(bb)	TV(bb)V(==)V(==)
T→V(a0a)	V(a0a)V(bb)V(==)V(==)
V(a0a)V(bb)→V(aa)V(b1b)	V(aa)V(b1b)V(==)V(==)
V(b1b)V(==)→V(bb)V(=0=)	V(aa)V(bb)V(=0=)V(==)
V(=0=)V(==)→V(==)V(=2=)	V(aa)V(bb)V(==)V(=2=)
V(=2=)→=	V(aa)V(bb)V(==)=
V(==)=→==	V(aa)V(bb)==
V(bb)=→b=	V(aa)b==
V(aa)b→ab	ab==
=→λ	ab=
=→λ	ab

Derived λ from =. Derivations complete.



The automaton is complete.
"Export" will put it in a new window.

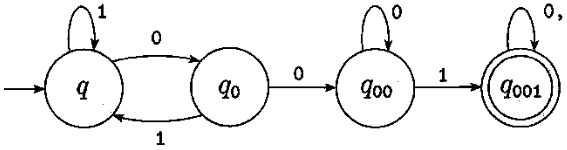
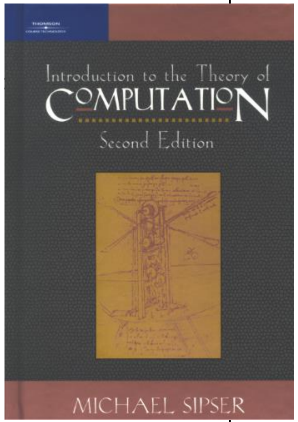


FIGURE 1.22
Accepts strings containing 001



THE REGULAR OPERATIONS

In the preceding two sections we introduced and defined finite automata and regular languages. We now begin to investigate their properties. Doing so will help develop a toolbox of techniques to use when you design automata to recognize particular languages. The toolbox also will include ways of proving that certain other languages are nonregular (i.e., beyond the capability of finite automata).

In arithmetic, the basic objects are numbers and the tools are operations for manipulating them, such as + and ×. In the theory of computation the objects are languages and the tools include operations specifically designed for manipulating them. We define three operations on languages, called the ***regular operations***, and use them to study properties of the regular languages.

DEFINITION 1.23

Let A and B be languages. We define the regular operations ***union***, ***concatenation***, and ***star*** as follows.

- **Union:** $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$.
- **Concatenation:** $A \circ B = \{xy \mid x \in A \text{ and } y \in B\}$.
- **Star:** $A^* = \{x_1 x_2 \ldots x_k \mid k \geq 0 \text{ and each } x_i \in A\}$.

You are already familiar with the union operation. It simply takes all the strings in both A and B and lumps them together into one language.

The concatenation operation is a little trickier. It attaches a string from A in front of a string from B in all possible ways to get the strings in the new language.

The star operation is a bit different from the other two because it applies to a single language rather than to two different languages. That is, the star operation is a ***unary operation*** instead of a ***binary operation***. It works by attaching any number of strings in A together to get a string in the new language. Because "any number" includes 0 as a possibility, the empty string ε is always a member of A^*, no matter what A is.

EXAMPLE 1.24

Let the alphabet Σ be the standard 26 letters $\{\texttt{a}, \texttt{b}, \ldots, \texttt{z}\}$. If $A = \{\texttt{good}, \texttt{bad}\}$ and $B = \{\texttt{boy}, \texttt{girl}\}$, then

$$A \cup B = \{\texttt{good}, \texttt{bad}, \texttt{boy}, \texttt{girl}\},$$

$$A \circ B = \{\texttt{goodboy}, \texttt{goodgirl}, \texttt{badboy}, \texttt{badgirl}\}, \text{ and}$$

$$A^* = \{\varepsilon, \texttt{good}, \texttt{bad}, \texttt{goodgood}, \texttt{goodbad}, \texttt{badgood}, \texttt{badbad}, \texttt{goodgoodgood}, \texttt{goodgoodbad}, \texttt{goodbadgood}, \texttt{goodbadbad}, \ldots\}.$$

Let $\mathcal{N} = \{1, 2, 3, \ldots\}$ be the set of natural numbers. When we say that $\mathcal{N}$ is ***closed under multiplication*** we mean that, for any x and y in $\mathcal{N}$, the product $x \times y$ also is in $\mathcal{N}$. In contrast $\mathcal{N}$ is not closed under division, as 1 and 2 are in $\mathcal{N}$ but $1/2$ is not. Generally speaking, a collection of objects is ***closed*** under some operation if applying that operation to members of the collection returns an object still in the collection. We show that the collection of regular languages is closed under all three of the regular operations. In Section 1.3 we show that these are useful tools for manipulating regular languages and understanding the power of finite automata. We begin with the union operation.

THEOREM 1.25

The class of regular languages is closed under the union operation.

In other words, if A_1 and A_2 are regular languages, so is $A_1 \cup A_2$.

PROOF IDEA We have regular languages A_1 and A_2 and want to show that $A_1 \cup A_2$ also is regular. Because A_1 and A_2 are regular, we know that some finite automaton M_1 recognizes A_1 and some finite automaton M_2 recognizes A_2. To prove that $A_1 \cup A_2$ is regular we demonstrate a finite automaton, call it M, that recognizes $A_1 \cup A_2$.

This is a proof by construction. We construct M from M_1 and M_2. Machine M must accept its input exactly when either M_1 or M_2 would accept it in order to recognize the union language. It works by *simulating* both M_1 and M_2 and accepting if either of the simulations accept.

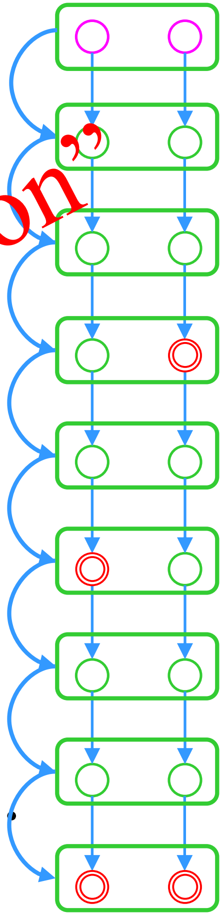
How can we make machine M simulate M_1 and M_2? Perhaps it first simulates M_1 on the input and then simulates M_2 on the input. But we must be careful here! Once the symbols of the input have been read and used to simulate M_1, we can't "rewind the input tape" to try the simulation on M_2. We need another approach.

Finite Automata

Theorem: Intersection preserves regularity.

Proof: ("parallel" simulation):

- Construct all super-states, one per each state pair.
- New super-transition function jumps among super-states, simulating both old transition functions
- Initial super state contains both old initial states.
- Final super states contains pairs of old final states.
- Resulting DFA accepts ∩ of languages of original 2 DFAs (but new size can be the product of their sizes).



"cross product construction"

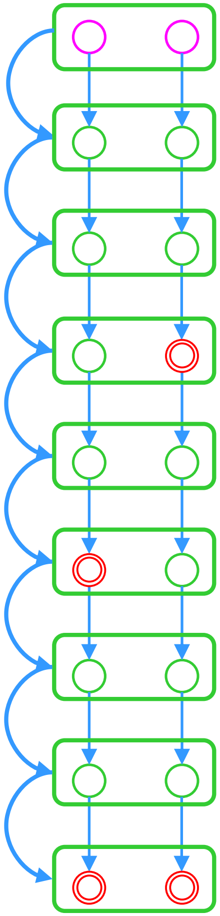
Given $M_1=(Q_1, \Sigma, \delta_1, q', F_1)$ and $M_2=(Q_2, \Sigma, \delta_2, q'', F_2)$

construct $M=(Q, \Sigma, \delta, q, F)$ $Q = \boxed{Q_1 \times Q_2}$

$F = F_1 \times F_2$ $q=(q',q'')$

$\delta : Q \times \Sigma \rightarrow Q$ $\delta((q_i,q_j),x) = (\delta_1(q_i,x),\delta_2(q_j,x))$

Finite Automata



Theorem: Union preserves regularity.

Proof: De Morgan's law: $L_1 \cup L_2 = \overline{\overline{L_1} \cap \overline{L_2}}$

Or cross-product construction, i.e.,
parallel simulation with $F = (F_1 \times Q_2) \cup (Q_1 \times F_2)$

Theorem: Set difference preserves regularity.

Proof: Set identity $L_1 - L_2 = L_1 \cap \overline{L_2}$

Or cross-product construction, i.e.,
parallel simulation with $F = (F_1 \times (Q_2 - F_2))$

Theorem: XOR preserves regularity.

Proof: Set identity $L_1 \oplus L_2 = (L_1 \cup L_2) - (L_1 \cap L_2)$

Or cross-product construction, i.e.,
parallel simulation with $F = (F_1 \times (Q_2 - F_2)) \cup ((Q_1 - F_1) \times F_2)$

Meta-Theorem: Identity-based proofs are easier!

This concludes the construction of the finite automaton M that recognizes the union of A_1 and A_2. This construction is fairly simple, and thus its correctness is evident from the strategy described in the proof idea. More complicated constructions require additional discussion to prove correctness. A formal correctness proof for a construction of this type usually proceeds by induction. For an example of a construction proved correct, see the proof of Theorem 1.54. Most of the constructions that you will encounter in this course are fairly simple and so do not require a formal correctness proof.

We have just shown that the union of two regular languages is regular, thereby proving that the class of regular languages is closed under the union operation. We now turn to the concatenation operation and attempt to show that the class of regular languages is closed under that operation, too.

THEOREM 1.26

The class of regular languages is closed under the concatenation operation.

In other words, if A_1 and A_2 are regular languages then so is $A_1 \circ A_2$.

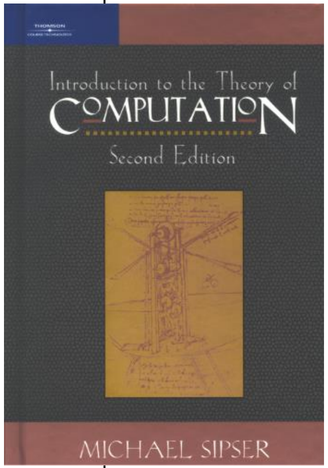
To prove this theorem let's try something along the lines of the proof of the union case. As before, we can start with finite automata M_1 and M_2 recognizing the regular languages A_1 and A_2. But now, instead of constructing automaton M to accept its input if either M_1 or M_2 accept, it must accept if its input can be broken into two pieces, where M_1 accepts the first piece and M_2 accepts the second piece. The problem is that M doesn't know where to break its input (i.e., where the first part ends and the second begins). To solve this problem we introduce a new technique called nondeterminism.

1.2

NONDETERMINISM

Nondeterminism is a useful concept that has had great impact on the theory of computation. So far in our discussion, every step of a computation follows in a unique way from the preceding step. When the machine is in a given state and reads the next input symbol, we know what the next state will be—it is determined. We call this ***deterministic*** computation. In a ***nondeterministic*** machine, several choices may exist for the next state at any point.

Nondeterminism is a generalization of determinism, so every deterministic finite automaton is automatically a nondeterministic finite automaton. As Figure 1.27 shows, nondeterministic finite automata may have additional features.



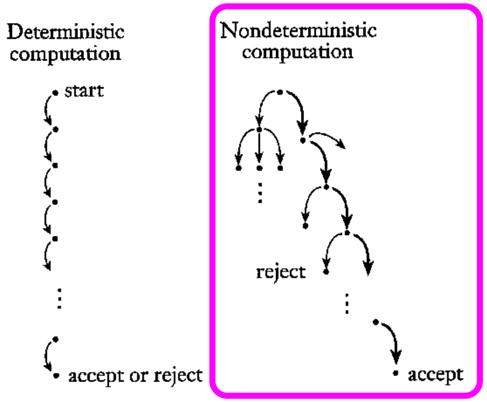


FIGURE 1.28
Deterministic and nondeterministic computations with an accepting branch

Let's consider some sample runs of the NFA N_1 shown in Figure 1.27. The computation of N_1 on input 010110 is depicted in the following figure.

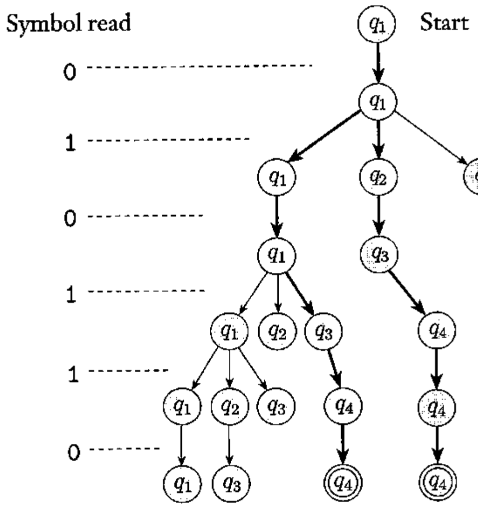


FIGURE 1.29
The computation of N_1 on input 010110

Finite Automata

Non-determinism: generalizes determinism, where many "next moves" are allowed at each step:

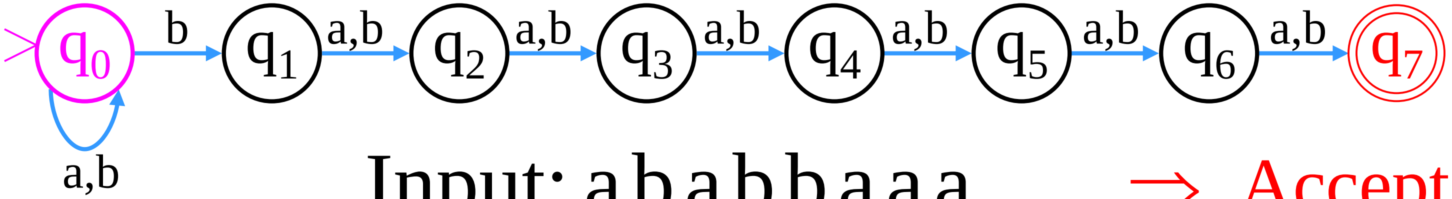
Old $\delta:Q\times\Sigma \to Q$

New $\delta:2^Q\times\Sigma \to 2^Q$

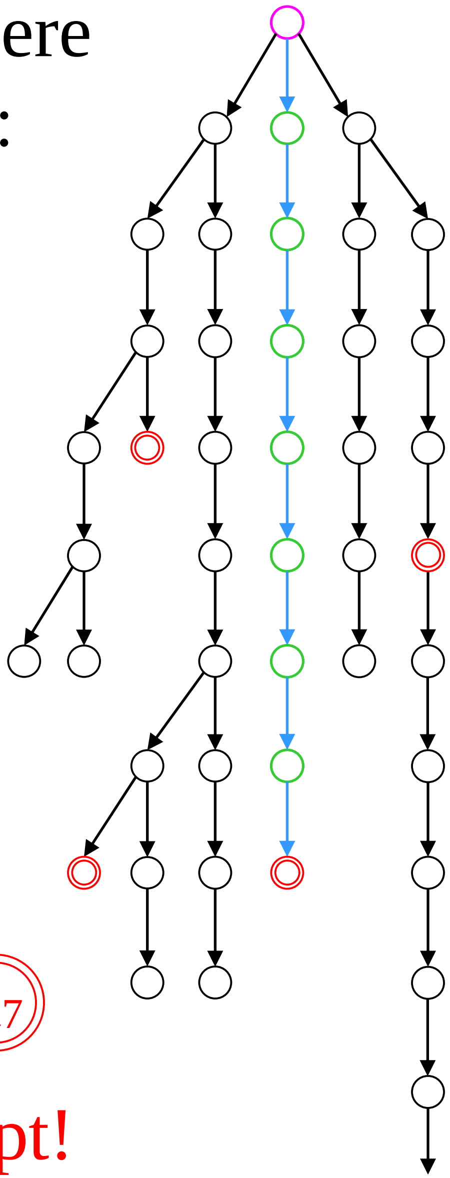
Computation becomes a "tree".

Acceptance: ∃ a path from root (start state) to some leaf (a final state)

Ex: non-deterministically accept all strings where the 7th symbol before the end is a "b":



Input: a b a b b a a a ⇒ Accept!

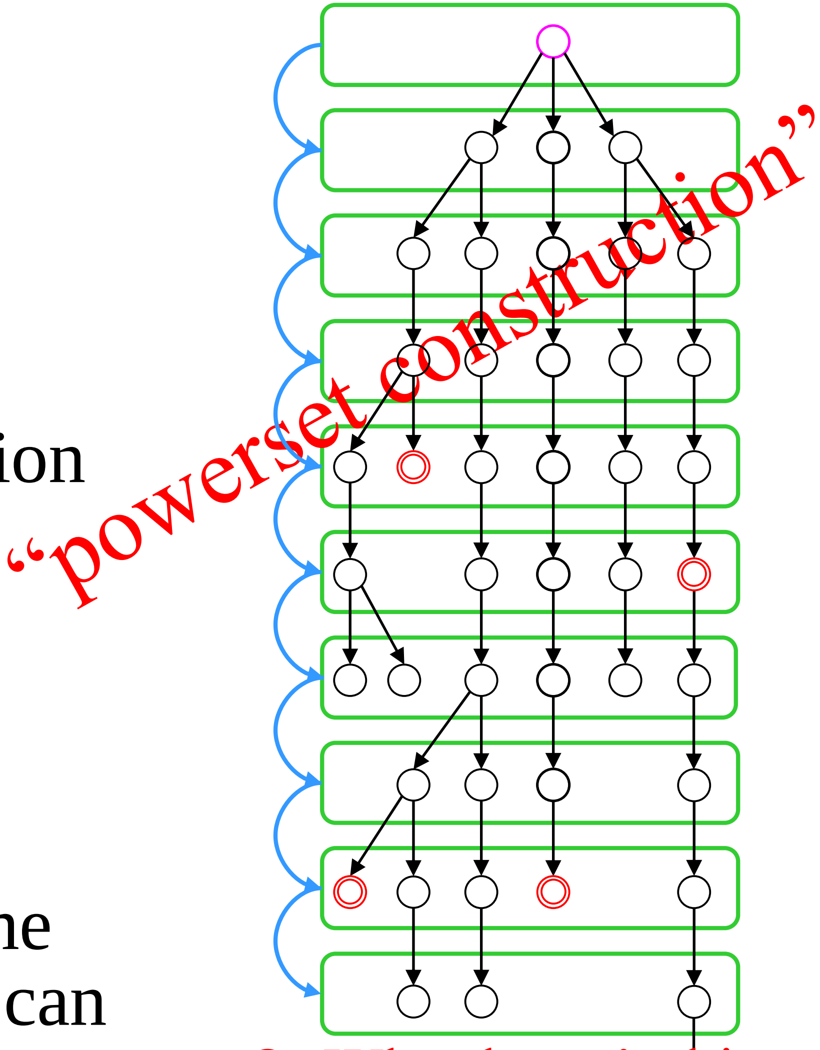


Finite Automata

Theorem: Non-determinism in FAs doesn't increase power.

Proof: by simulation:

- Construct all super-states, one per each state subset.
- New super-transition function jumps among super-states, simulating old transition function
- Initial super state are those containing old initial state.
- Final super states are those containing old final states.
- Resulting DFA accepts the same language as original NFA, but can have exponentially more states.

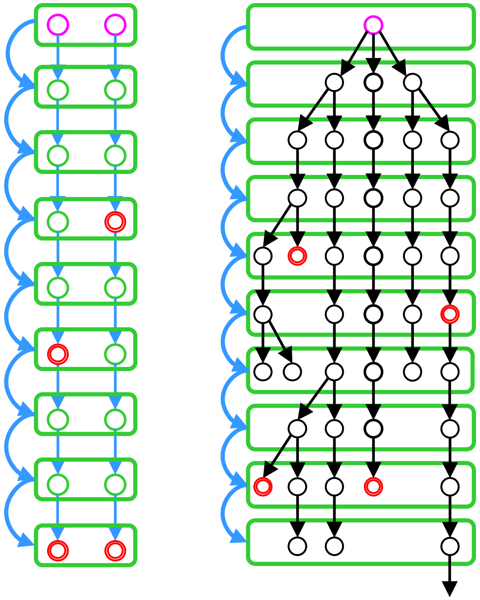


Q: Why doesn't this work for PDAs or TMs?

Finite Automata

Note: Powerset construction generalizes the cross-product construction. More general constructions are possible.

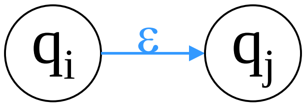
EC: Let HALF(L)=$\{v \mid \exists\ v,w \in \Sigma^* \ni |v|=|w| \text{ and } vw\ \varepsilon\ L\}$
Show that HALF preserves regularity.

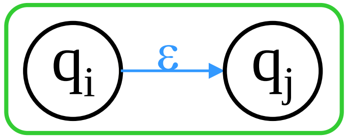


A two way FA can move its head backwards on the input: $\delta:Q\times\Sigma \rightarrow Q\times\{\text{left},\text{right}\}$

EC: Show that two-way FA are not more powerful than ordinary one-way FA.

ε-transitions:





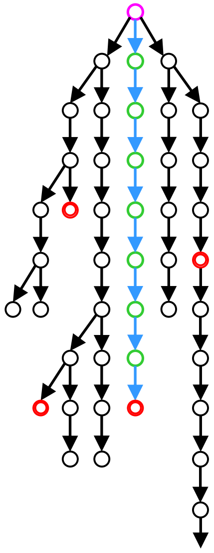
One super-state!

Theorem: ε-transitions don't increase FA recognition power.

Proof: Simulate ε-transitions FA without using ε-transitions.
i.e., consider ε-transitions to be a form of non-determinism.

Extra credit!

The movie “Next” (2007)
Based on the science fiction story “The Golden Man” by Philip Dick



Premise: a man with the super power of non-determinism!

At any given moment his reality branches into multiple directions, and he can choose the branch that he prefers!

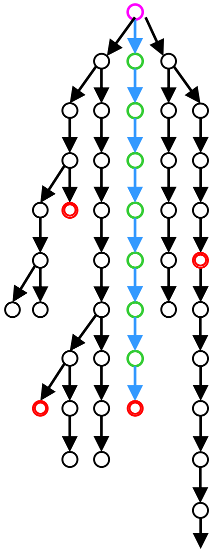
Transition function!

Top-10 Reasons to Study Non-determinism

1. Helps us understand the ubiquitous concept of *parallelism* / concurrency;
2. Illuminates the structure of problems;
3. Can help save time & effort by solving intractable problems more efficiently;
4. Enables vast, deep, and general studies of “completeness” theories;
5. Helps explain why verifying proofs & solutions seems to be easier than constructing them;

Why Study Non-determinism?

6. Gave rise to new and novel mathematical approaches, proofs, and analyses;
7. Robustly decouples / abstracts complexity from underlying computational models;
8. Gives disciplined techniques for identifying "hardest" problems / languages;
9. Forged new unifications between computer science, math & logic;
10. Non-determinism is interesting fun, and cool!

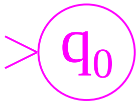


Regular Expressions

Regular expressions are defined recursively as follows:

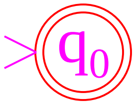
$\varnothing$

empty set



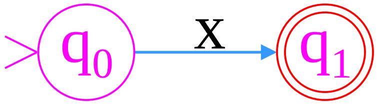
$\{\varepsilon\}$

trivial language



$\{x\}\ \forall\ x \in \Sigma$

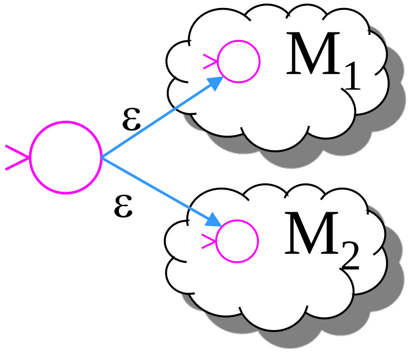
singleton language



Inductively, if R and S are regular expressions, then so are:

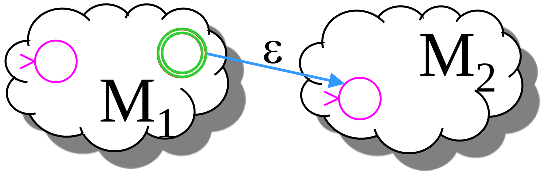
$(R+S)$

union



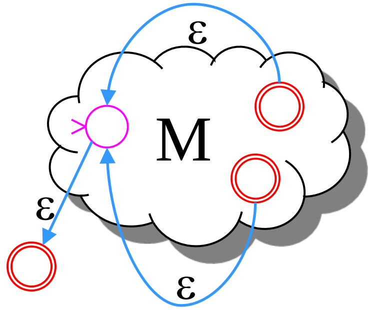
RS

concatenation



R^*

Kleene closure



Compositions!

Examples: $aa(a+b)^*bb$ $(a+b)^*b(a+b)^*a(a+b)^*$

Theorem: Any regular expression is accepted by some FA.

First, let's consider again closure under union. Earlier we proved closure under union by simulating deterministically both machines simultaneously via a Cartesian product construction. We now give a new proof to illustrate the technique of nondeterminism. Reviewing the first proof, appearing on page 45, may be worthwhile to see how much easier and more intuitive the new proof is.

THEOREM 1.45

The class of regular languages is closed under the union operation.

PROOF IDEA We have regular languages A_1 and A_2 and want to prove that $A_1 \cup A_2$ is regular. The idea is to take two NFAs, N_1 and N_2 for A_1 and A_2, and combine them into one new NFA, N.

Machine N must accept its input if either N_1 or N_2 accepts this input. The new machine has a new start state that branches to the start states of the old machines with ε arrows. In this way the new machine nondeterministically guesses which of the two machines accepts the input. If one of them accepts the input, N will accept it, too.

We represent this construction in the following figure. On the left, we indicate the start and accept states of machines N_1 and N_2 with large circles and some additional states with small circles. On the right, we show how to combine N_1 and N_2 into N by adding additional transition arrows.

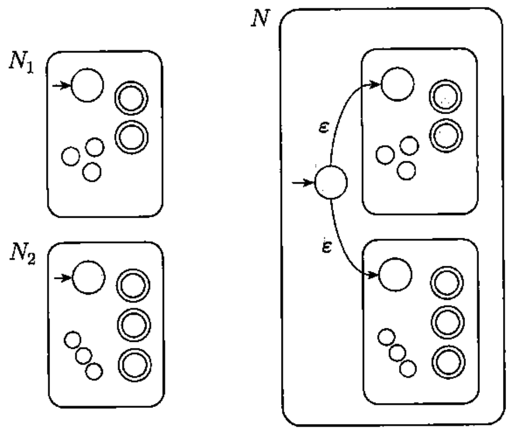
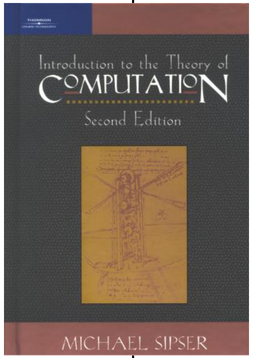


FIGURE 1.46
Construction of an NFA N to recognize $A_1 \cup A_2$


Introduction to the Theory of
COMPUTATION
Second Edition
MICHAEL SIPSER

THEOREM 1.49

The class of regular languages is closed under the star operation.

PROOF IDEA We have a regular language A_1 and want to prove that A_1^* also is regular. We take an NFA N_1 for A_1 and modify it to recognize A_1^*, as shown in the following figure. The resulting NFA N will accept its input whenever it can be broken into several pieces and N_1 accepts each piece.

We can construct N like N_1 with additional ε arrows returning to the start state from the accept states. This way, when processing gets to the end of a piece that N_1 accepts, the machine N has the option of jumping back to the start state to try to read in another piece that N_1 accepts. In addition we must modify N so that it accepts ε, which always is a member of A_1^*. One (slightly bad) idea is simply to add the start state to the set of accept states. This approach certainly adds ε to the recognized language, but it may also add other, undesired strings. Exercise 1.15 asks for an example of the failure of this idea. The way to fix it is to add a new start state, which also is an accept state, and which has an ε arrow to the old start state. This solution has the desired effect of adding ε to the language without adding anything else.

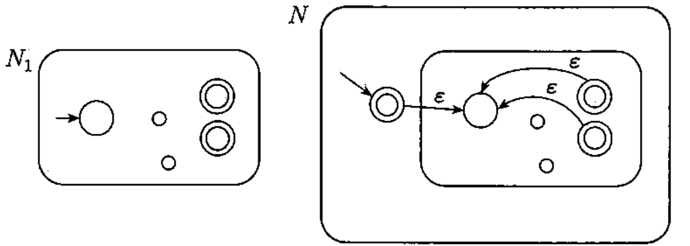


FIGURE 1.50
Construction of N to recognize A^*

PROOF Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognize A_1. Construct $N = (Q, \Sigma, \delta, q_0, F)$ to recognize A_1^*.

1. $Q = \{q_0\} \cup Q_1$.
   The states of N are the states of N_1 plus a new start state.
2. The state q_0 is the new start state.
3. $F = \{q_0\} \cup F_1$.
   The accept states are the old accept states plus the new start state.

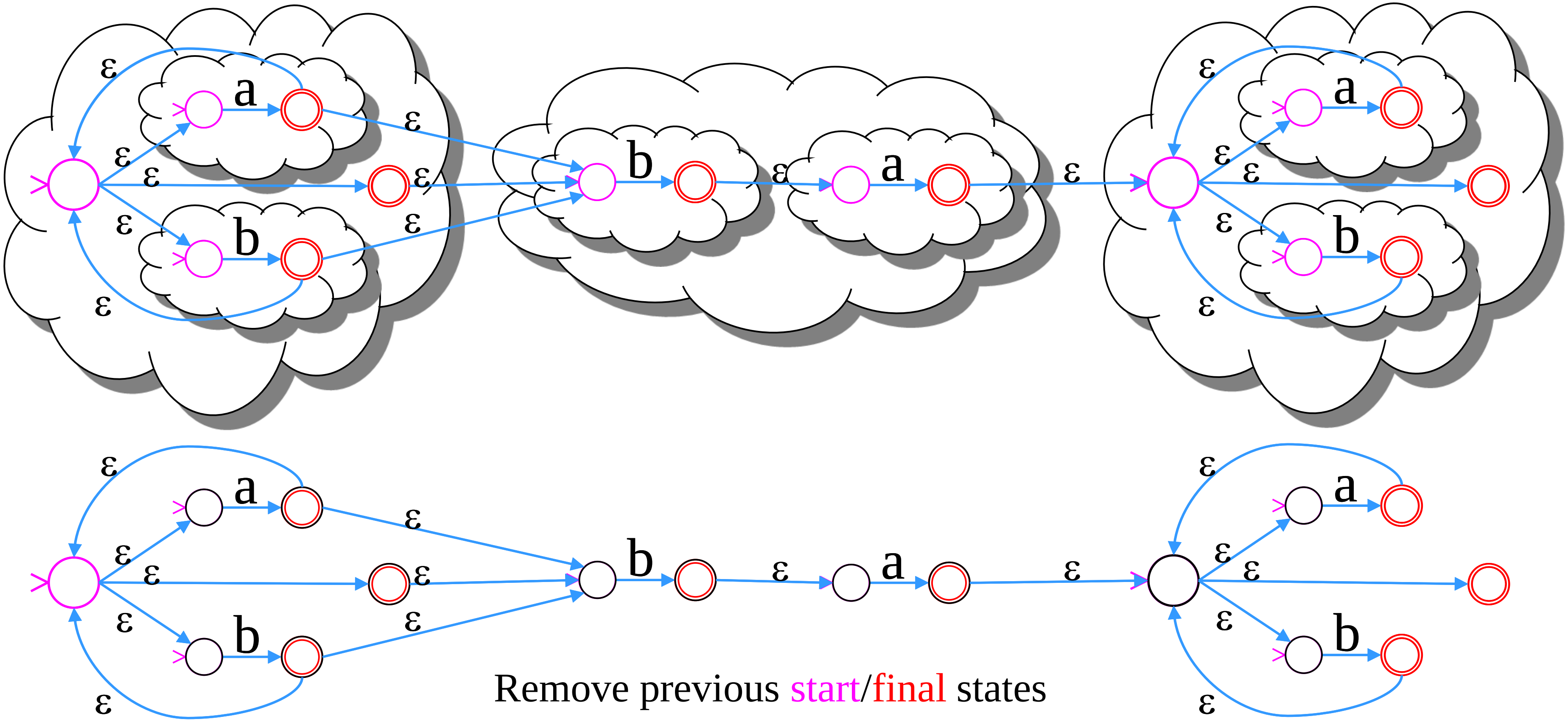
Regular Expressions

A FA for a regular expressions can be built by composition:

Ex: all strings over S={a,b} where ∃ a "b" preceding an "a"

$(a+b)^*b(a+b)^*a(a+b)^*$

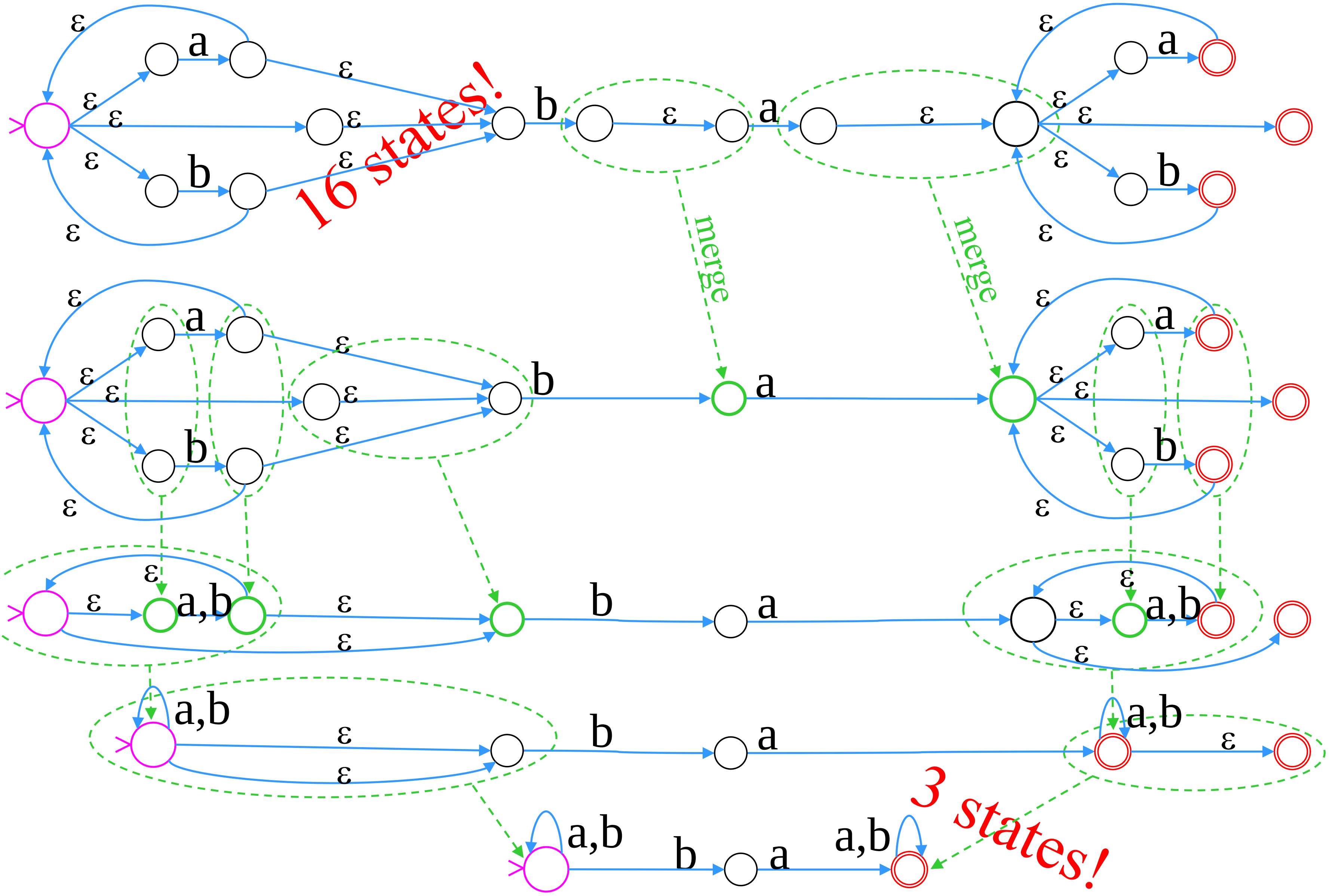
Why? → $= (a+b)^*ba(a+b)^*$



Remove previous start/final states

FA Minimization

Idea: “Equivalent” states can be merged:



FA Minimization

Theorem [Hopcroft 1971]: the number N of states in a FA can be minimized within time O(N log N).

Based on earlier work [Huffman 1954] & [Moore 1956].

Conjecture: Minimizing the number of states in a nondeterministic FA can not be done in polynomial time.

Theorem: Minimizing the number of states in a pushdown automaton (or TM) is undecidable.

Idea: implement a finite automaton minimization tool

- Try to design it to run reasonably efficiently
- Consider also including:
    - A regular-expression-to-FA transformer
    - A non-deterministic-to-deterministic FA converter

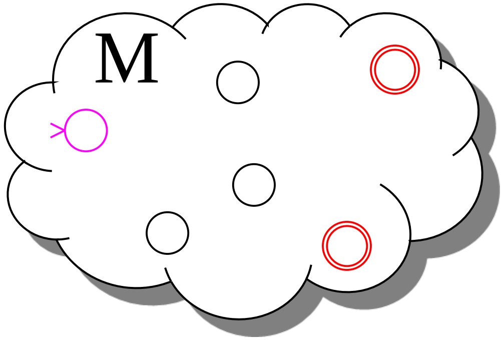
FAs and Regular Expressions

Theorem: Any FA accepts a language denoted by some RE.

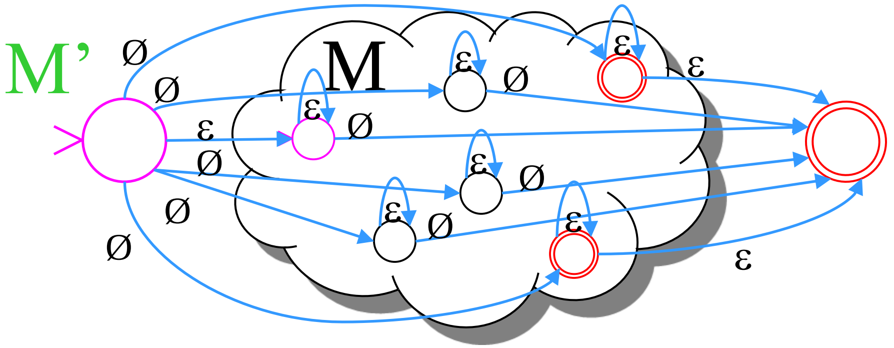
Proof: Use “generalized finite automata” where a transition can be a regular expression (not just a symbol), and:

Only 1 super start state and 1 (separate) super final state.

Each state has transitions to all other states (including itself), except the super start state, with no incoming transitions, and the super final state, which has no outgoing transitions.



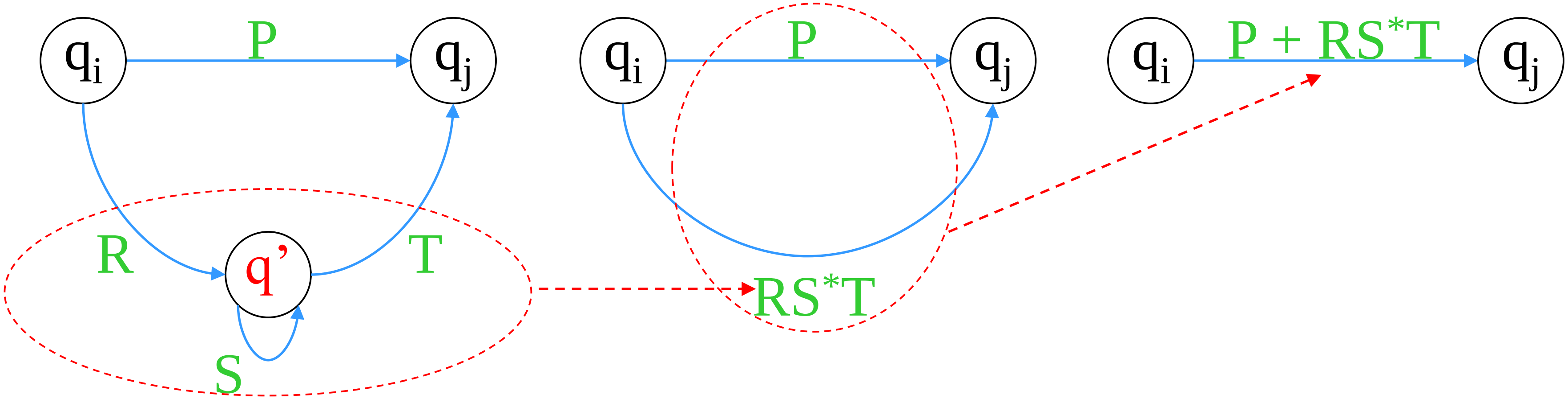
Original FA M



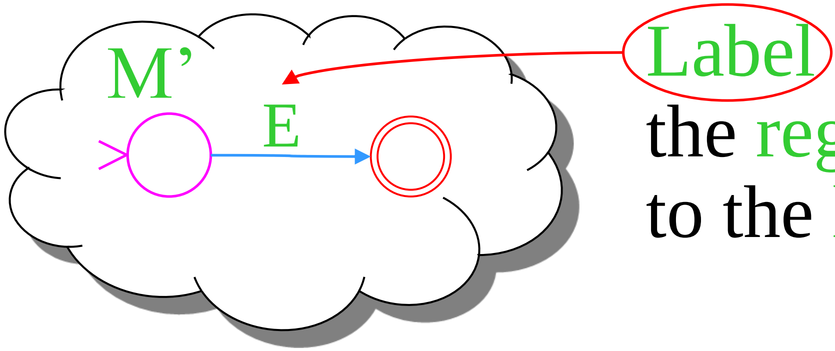
Generalized FA (GFA) M’

FAs and Regular Expressions

Now reduce the size of the GFA by one state at each step.
A transformation step is as follows:



Such a transformation step is always possible, until the GFA has only two states, the super-start and super-final states:



Label of last remaining transition is the regular expression corresponding to the language of the original FA!

Corollary: FAs and REs denote the same class of languages.

We break this procedure into two parts, using a new type of finite automaton called a ***generalized nondeterministic finite automaton***, GNFA. First we show how to convert DFAs into GNFAs, and then GNFAs into regular expressions.

Generalized nondeterministic finite automata are simply nondeterministic finite automata wherein the transition arrows may have any regular expressions as labels, instead of only members of the alphabet or ε. The GNFA reads blocks of symbols from the input, not necessarily just one symbol at a time as in an ordinary NFA. The GNFA moves along a transition arrow connecting two states by reading a block of symbols from the input, which themselves constitute a string described by the regular expression on that arrow. A GNFA is nondeterministic and so may have several different ways to process the same input string. It accepts its input if its processing can cause the GNFA to be in an accept state at the end of the input. The following figure presents an example of a GNFA.

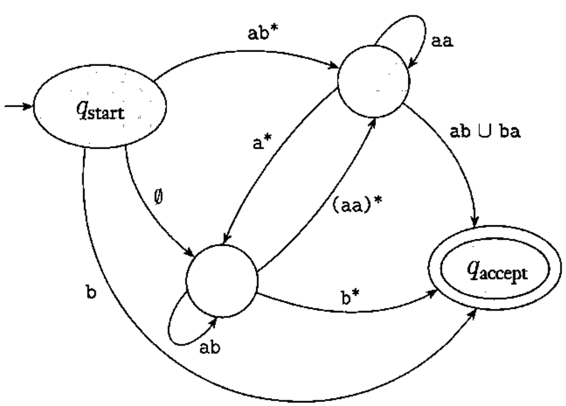
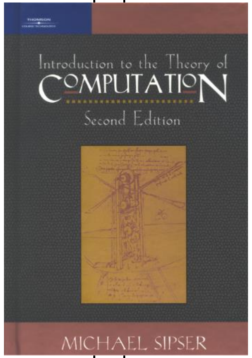


FIGURE 1.61
A generalized nondeterministic finite automaton

For convenience we require that GNFAs always have a special form that meets the following conditions.

- The start state has transition arrows going to every other state but no arrows coming in from any other state.
- There is only a single accept state, and it has arrows coming in from every other state but no arrows going to any other state. Furthermore, the accept state is not the same as the start state.
- Except for the start and accept states, one arrow goes from every state to every other state and also from each state to itself.



take the machine from q_i to q_j either directly or via q_{rip}. We illustrate this approach in Figure 1.63.

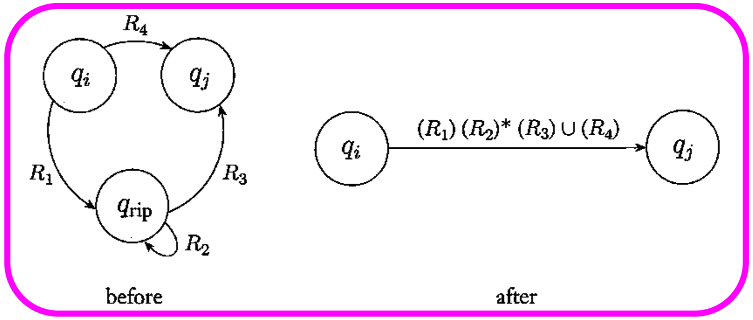


FIGURE 1.63
Constructing an equivalent GNFA with one fewer state

In the old machine if q_i goes to q_{rip} with an arrow labeled R_1, q_{rip} goes to itself with an arrow labeled R_2, q_{rip} goes to q_j with an arrow labeled R_3, and q_i goes to q_j with an arrow labeled R_4, then in the new machine the arrow from q_i to q_j gets the label

$$(R_1)(R_2)^*(R_3) \cup (R_4).$$

We make this change for each arrow going from any state q_i to any state q_j, including the case where $q_i = q_j$. The new machine recognizes the original language.

PROOF Let's now carry out this idea formally. First, to facilitate the proof, we formally define the new type of automaton introduced. A GNFA is similar to a nondeterministic finite automaton except for the transition function, which has the form

$$\delta\colon \big(Q - \{q_{\text{accept}}\}\big) \times \big(Q - \{q_{\text{start}}\}\big) \longrightarrow \mathcal{R}.$$

The symbol $\mathcal{R}$ is the collection of all regular expressions over the alphabet Σ, and q_{start} and q_{accept} are the start and accept states. If $\delta(q_i, q_j) = R$, the arrow from state q_i to state q_j has the regular expression R as its label. The domain of the transition function is $\big(Q - \{q_{\text{accept}}\}\big) \times \big(Q - \{q_{\text{start}}\}\big)$ because an arrow connects every state to every other state, except that no arrows are coming from q_{accept} or going to q_{start}.

Regular Expressions Identities

- $R+S = S+R$
- $R(ST) = (RS)T$
- $R(S+T) = RS+RT$
- $(R+S)T = RT+ST$
- $\emptyset^* = \varepsilon^* = \varepsilon$
- $R+\emptyset = \emptyset+R = R$
- $R\varepsilon = \varepsilon R = R$
- $(R^*)^* = R^*$
- $(\varepsilon + R)^* = R^*$
- $(R^*S^*)^* = (R+S)^*$

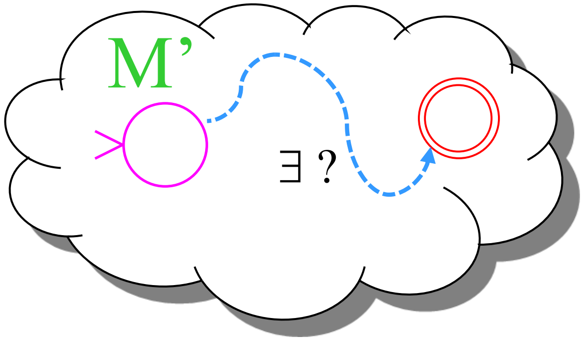
$R+\varepsilon \neq R$

$R\emptyset \neq R$

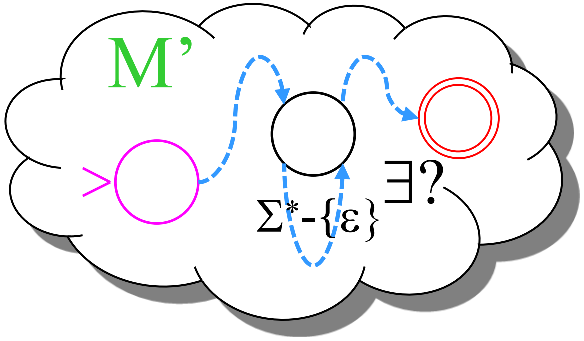
Decidable Finite Automata Problems

Def: A problem is decidable if ∃ an algorithm which can determine (in finite time) the correct answer for any instance.

Given a finite automata M_1 and M_2:

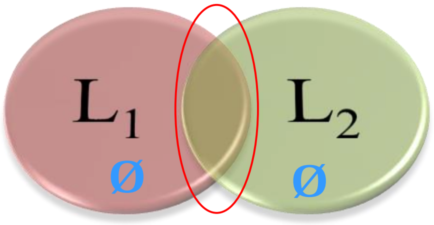


Q_1: Is $L(M_1) = Ø$?
Hint: graph reachability



Q_2: Is $L(M_2)$ infinite ?
Hint: cycle detection

Q_3: Is $L(M_1) = L(M_2)$?
Hint: consider L_1-L_2 and L_2-L_1



Regular Experssion Minimization

Problem: find smallest equivalent regular expression

- Decidable (why?)
- Hard: PSPACE-complete

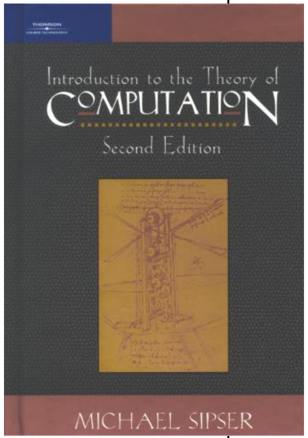
Turing Machine Minimization

Problem: find smallest equivalent Turing machine

- Not decidable (why?)
- Not even recognizable (why?)

2

CONTEXT-FREE LANGUAGES



In Chapter 1 we introduced two different, though equivalent, methods of describing languages: *finite automata* and *regular expressions*. We showed that many languages can be described in this way but that some simple languages, such as $\{0^n1^n \mid n \geq 0\}$, cannot.

In this chapter we present ***context-free grammars***, a more powerful method of describing languages. Such grammars can describe certain features that have a recursive structure, which makes them useful in a variety of applications.

Context-free grammars were first used in the study of human languages. One way of understanding the relationship of terms such as *noun*, *verb*, and *preposition* and their respective phrases leads to a natural recursion because noun phrases may appear inside verb phrases and vice versa. Context-free grammars can capture important aspects of these relationships.

An important application of context-free grammars occurs in the specification and compilation of programming languages. A grammar for a programming language often appears as a reference for people trying to learn the language syntax. Designers of compilers and interpreters for programming languages often start by obtaining a grammar for the language. Most compilers and interpreters contain a component called a ***parser*** that extracts the meaning of a program prior to generating the compiled code or performing the interpreted execution. A number of methodologies facilitate the construction of a parser once a context-free grammar is available. Some tools even automatically generate the parser from the grammar.

The collection of languages associated with context-free grammars are called the ***context-free languages***. They include all the regular languages and many additional languages. In this chapter, we give a formal definition of context-free grammars and study the properties of context-free languages. We also introduce ***pushdown automata***, a class of machines recognizing the context-free languages. Pushdown automata are useful because they allow us to gain additional insight into the power of context-free grammars.

2.1

CONTEXT-FREE GRAMMARS

The following is an example of a context-free grammar, which we call G_1.

$$\begin{aligned} A &\rightarrow 0A1 \\ A &\rightarrow B \\ B &\rightarrow \# \end{aligned}$$

A grammar consists of a collection of ***substitution rules***, also called ***productions***. Each rule appears as a line in the grammar, comprising a symbol and a string separated by an arrow. The symbol is called a ***variable***. The string consists of variables and other symbols called ***terminals***. The variable symbols often are represented by capital letters. The terminals are analogous to the input alphabet and often are represented by lowercase letters, numbers, or special symbols. One variable is designated as the ***start variable***. It usually occurs on the left-hand side of the topmost rule. For example, grammar G_1 contains three rules. G_1's variables are A and B, where A is the start variable. Its terminals are 0, 1, and #.

You use a grammar to describe a language by generating each string of that language in the following manner.

1. Write down the start variable. It is the variable on the left-hand side of the top rule, unless specified otherwise.
2. Find a variable that is written down and a rule that starts with that variable. Replace the written down variable with the right-hand side of that rule.
3. Repeat step 2 until no variables remain.

For example, grammar G_1 generates the string 000#111. The sequence of substitutions to obtain a string is called a ***derivation***. A derivation of string 000#111 in grammar G_1 is

$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000B111 \Rightarrow 000\#111$$

You may also represent the same information pictorially with a ***parse tree***. An example of a parse tree is shown in Figure 2.1.

Context-Free Grammars

Basic idea: set of production rules induces a language

- Finite set of variables: $V = \{V_1, V_2, \ldots, V_k\}$
- Finite set of terminals: $T = \{t_1, t_2, \ldots, t_j\}$
- Finite set of productions: P
- Start symbol: S
- Productions: $V_i \rightarrow \Delta$ where $V_i \in V$ and $\Delta \in (V \cup T)^*$

Applying $V_i \rightarrow \Delta$ to $\alpha V_i \beta$

yields: $\alpha \Delta \beta$

Note: productions do not depend on "context"

- hence the name "context free"!

Context-Free Grammars

Example: G:

$S \rightarrow Sa$

$S \rightarrow Sb$

$S \rightarrow \varepsilon$

G can be denoted more succinctly as:

G: $S \rightarrow Sa \mid Sb \mid \varepsilon$

Def: A derivation in a grammar G is a sequence of productions applied to the start symbol, ending with a final derived string (of terminals).

Ex:

$S \rightarrow Sa \rightarrow \boxed{a}$

$S \rightarrow Sa \rightarrow Sba \rightarrow Saba \rightarrow Saaba \rightarrow \boxed{aaba}$

$S \rightarrow Sa \rightarrow Saa \rightarrow Saaa \rightarrow Sbaaa \rightarrow Sbbaaa \rightarrow \boxed{bbaaa}$

$S \rightarrow \boxed{\varepsilon}$

strings in the language

Context-Free Grammars

Def: A string w is generated by a grammar G
if some derivation in G yields w.

Example: $S \rightarrow Sa \rightarrow Sba \rightarrow Saba \rightarrow Saaba \rightarrow aaba$

Def: The language L(G) generated by a context-free
grammar G is the set of all strings that G generates.

Example: G: $S \rightarrow Sa \mid Sb \mid \varepsilon$

$\{\varepsilon, a, aaba, bbaaa, \dots\} \subseteq L(G)$

moreover $\{a,b\}^* \subseteq L(G) \Rightarrow L(G)=\{a,b\}^*$

i.e., $L(G)=\Sigma^*$ where $\Sigma=\{a,b\}$

Def: A language is context-free if there exists a
context-free grammar that generates it.

Example: $L=\{a,b\}^*$ is context-free (and it is also regular).

Context-Free Grammars

Def: a palindrome reads the same forwards and backwards.
e.g., “noon”, “civic”, “level”, “rotor”, “madam”, “kayak”, “radar”, “reviver”, “racecar”, “step on no pets”, etc.

Example: design a context-free grammar that generates all palindromic strings over $\Sigma=\{a,b\}$
i.e., $L = \{w \mid w \in \Sigma^* \text{ and } w = w^R \}$

Idea: generate both ends of w simultaneously, from middle.

G: $S \rightarrow aSa \mid bSb \mid a \mid b \mid \varepsilon$

Derivations:

$S \rightarrow aSa \rightarrow abSba \rightarrow abba$

$S \rightarrow bSb\ S \rightarrow baSab \rightarrow baaSaab \rightarrow baabaab$

$L(G) = \{w \mid w \in \Sigma^* \text{ and } w = w^R\}$

Context-Free Grammars

Example: design a context-free grammar for strings representing all well-balanced parenthesis.

Idea: create rules for generating nesting & juxtaposition.

G_1: $S \to SS \mid (S) \mid \varepsilon$

Ex: $S \to SS \to (S)(S) \to (\varepsilon)(\varepsilon) \to ()()$

$S \to (S) \to ((S)) \to ((\varepsilon)) \to (())$

$S \to (S) \to (SS) \to \ldots \to (()((()))())$

Another grammar:

G_2: $S \to (S)S \mid \varepsilon$

Q: Is $L(G_1) = L(G_2)$?

Context-Free Grammars

Example: design a context-free grammar that generates all valid regular expressions.

Idea: embed the regular expression rules in a grammar.

G: $S \rightarrow x_i$ for each $x_i \in \Sigma_L$

$S \rightarrow (S) \mid SS \mid S^* \mid S+S$

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

Derivations:

$S \rightarrow S^* \rightarrow (S)^* \rightarrow (S+S)^* \rightarrow (a+b)^*$

$S \rightarrow SS \rightarrow SSSS \rightarrow abS^*b \rightarrow aba^*a$

Theorem: The set of regular expressions is context-free.

Ambiguity

Def: A statement /sentence is ambiguous if it has multiple syntactic / semantic interpretations.

Example: “I like $\boxed{\text{dominating}}$ people”

verb or adjective?

Example: a-b+c (a-b)+c $\neq$ a-(b+c)

Example: if p then if q then S else T

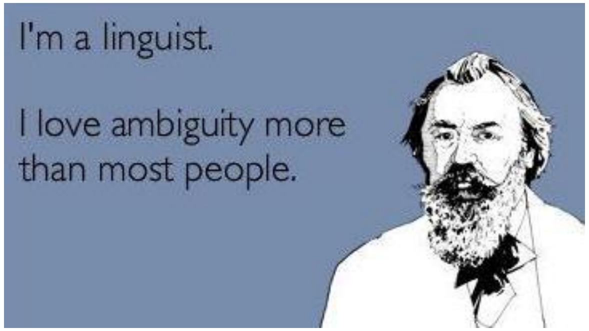
$\text{if p then (if q } \underbrace{\text{then S else}} \text{ T)}$

or:

$\text{if p } \underbrace{\text{then (if q then S) else}} \text{ T}$

Ambiguity in programs should be avoided!

Ambiguity in Language



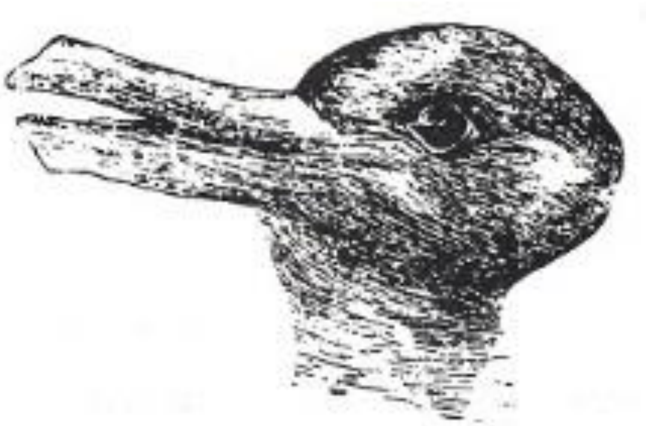
I'm a linguist.

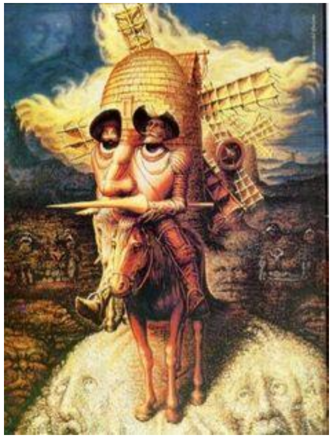
I love ambiguity more than most people.

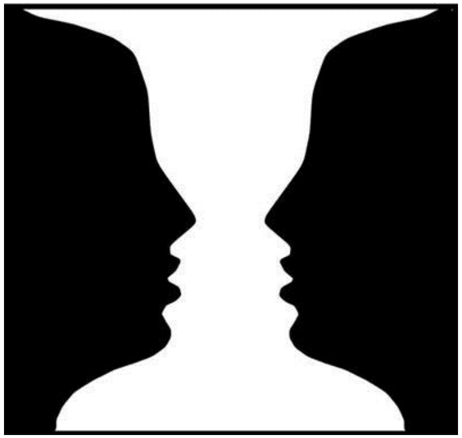
- Call me a cab.
- OK, you're a cab.

“I'm glad I'm a man, and so is Lola.” - Last line of song “Lola” by The Kinks

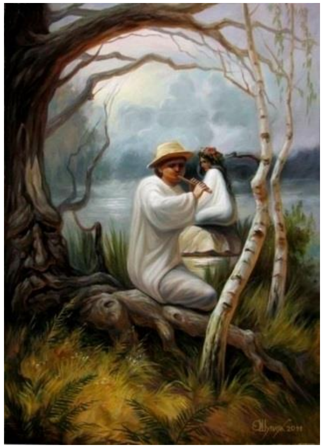
Ambiguity in Art

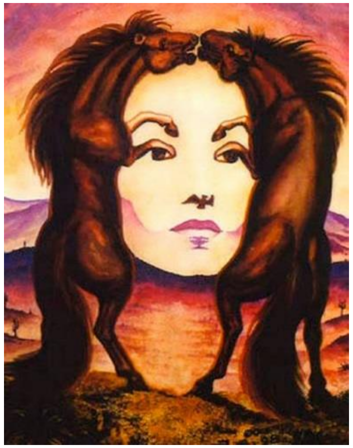


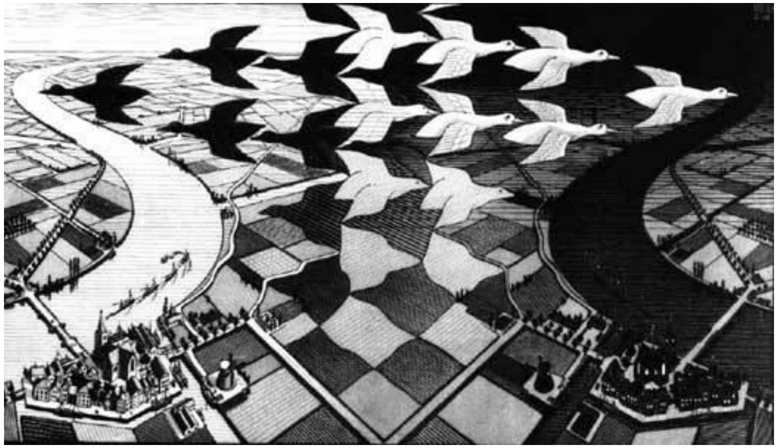


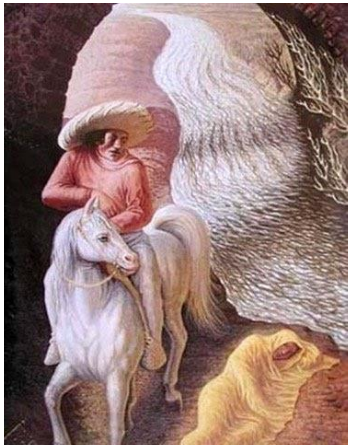


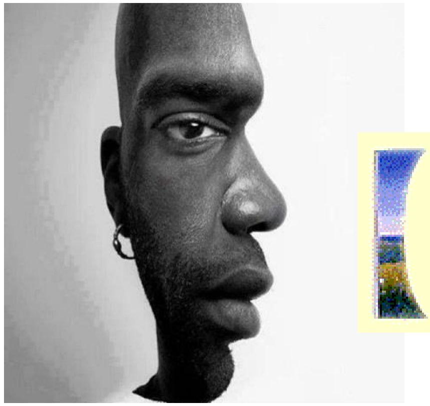
Ambiguity in Art











Ambiguity

Def: A grammar is ambiguous if some string in its language has two non-isomorphic derivations.

Theorem: Some context-free grammars are ambiguous.

Example: $L = \{\varepsilon\}$

G_1: $S \rightarrow SS \mid \varepsilon$

Derivation 1: $S \rightarrow \varepsilon$

Derivation 2: $S \rightarrow SS \rightarrow SSS \rightarrow \varepsilon\varepsilon\varepsilon = \varepsilon$

G_1 is ambiguous!

G_2: $S \rightarrow \varepsilon$

$L(G_1) = L(G_2) = \{\varepsilon\}$

G_2 is not ambiguous!

Ambiguity

Def: A grammar is ambiguous if some string in its language has two non-isomorphic derivations.

Theorem: Some context-free grammars are ambiguous.

Example: $L = a^*$

G_3: $S \rightarrow SS \mid a \mid \varepsilon$

Derivation 1: $S \rightarrow SS \rightarrow aa$

Derivation 2: $S \rightarrow SS \rightarrow SSS \rightarrow aa\varepsilon = aa$

G_3 is ambiguous!

G_4: $S \rightarrow Sa \mid \varepsilon$

$L(G_3) = L(G_4) = a^*$

G_4 is not ambiguous!

Ambiguity

Def: A grammar is ambiguous if some string in its language has two non-isomorphic derivations.

Theorem: Some context-free grammars are ambiguous.

Example: well-balanced parenthesis:

G_5: $S \rightarrow SS \mid (S) \mid \varepsilon$

Derivation 1: $S \rightarrow (S) \rightarrow (\varepsilon) \rightarrow (\,)$

Derivation 2: $S \rightarrow SS \rightarrow (S)S \rightarrow (\varepsilon)\varepsilon \rightarrow (\,)$

G_5 is ambiguous!

G_6: $S \rightarrow (S)S \mid \varepsilon$

$L(G_5) = L(G_6)$

G_6 is not ambiguous!

Ambiguity

Def: A grammar is ambiguous if some string in its language has two non-isomorphic derivations.

Theorem: Some context-free grammars are ambiguous. (but non-ambiguous grammars can be found)

grammar's fault

Def: A context-free language is inherently ambiguous if every context-free grammar for it is ambiguous.

language's fault

Theorem: Some context-free languages are inherently ambiguous (i.e., no non-ambiguous CFG exists).

Ex: $\{a^n b^n c^m d^m \mid m>0, n>0\} \cup \{a^n b^m c^m d^n \mid m>0, n>0\}$ is an inherently ambiguous CF language, and so is $\{a^n b^m c^k \mid n=m \text{ or } m=k\}$

4. Convert the remaining rules into the proper form by adding additional variables and rules. The final grammar in Chomsky normal form is equivalent to G_6, which follows. (Actually the procedure given in Theorem 2.9 produces several variables U_i along with several rules $U_i \rightarrow \mathtt{a}$. We simplified the resulting grammar by using a single variable U and rule $U \rightarrow \mathtt{a}$.)

$$
\begin{aligned}
S_0 &\rightarrow AA_1 \mid UB \mid \mathtt{a} \mid SA \mid AS \\
S &\rightarrow AA_1 \mid UB \mid \mathtt{a} \mid SA \mid AS \\
A &\rightarrow \mathtt{b} \mid AA_1 \mid UB \mid \mathtt{a} \mid SA \mid AS \\
A_1 &\rightarrow SA \\
U &\rightarrow \mathtt{a} \\
B &\rightarrow \mathtt{b}
\end{aligned}
$$

2.2

PUSHDOWN AUTOMATA

In this section we introduce a new type of computational model called ***pushdown automata***. These automata are like nondeterministic finite automata but have an extra component called a ***stack***. The stack provides additional memory beyond the finite amount available in the control. The stack allows pushdown automata to recognize some nonregular languages.

Pushdown automata are equivalent in power to context-free grammars. This equivalence is useful because it gives us two options for proving that a language is context free. We can give either a context-free grammar generating it or a pushdown automaton recognizing it. Certain languages are more easily described in terms of generators, whereas others are more easily described in terms of recognizers.

The following figure is a schematic representation of a finite automaton. The control represents the states and transition function, the tape contains the input string, and the arrow represents the input head, pointing at the next input symbol to be read.

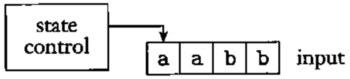
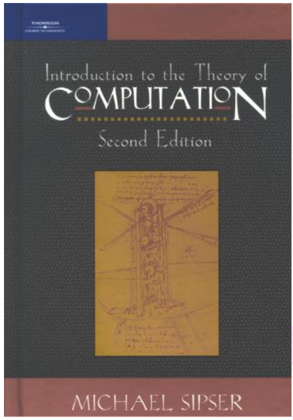


FIGURE 2.11
Schematic of a finite automaton



Nondeterministic pushdown automata recognize certain languages which no deterministic pushdown automata can recognize, though we will not prove this fact. We give languages requiring nondeterminism in Examples 2.16 and 2.18. Recall that deterministic and nondeterministic finite automata do recognize the same class of languages, so the pushdown automata situation is different. We focus on nondeterministic pushdown automata because these automata are equivalent in power to context-free grammars.

FORMAL DEFINITION OF A PUSHDOWN AUTOMATON

The formal definition of a pushdown automaton is similar to that of a finite automaton, except for the stack. The stack is a device containing symbols drawn from some alphabet. The machine may use different alphabets for its input and its stack, so now we specify both an input alphabet Σ and a stack alphabet Γ.

At the heart of any formal definition of an automaton is the transition function, which describes its behavior. Recall that $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$ and $\Gamma_\varepsilon = \Gamma \cup \{\varepsilon\}$. The domain of the transition function is $Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon$. Thus the current state, next input symbol read, and top symbol of the stack determine the next move of a pushdown automaton. Either symbol may be ε, causing the machine to move without reading a symbol from the input or without reading a symbol from the stack.

For the range of the transition function we need to consider what to allow the automaton to do when it is in a particular situation. It may enter some new state and possibly write a symbol on the top of the stack. The function δ can indicate this action by returning a member of Q together with a member of Γ_ε, that is, a member of $Q \times \Gamma_\varepsilon$. Because we allow nondeterminism in this model, a situation may have several legal next moves. The transition function incorporates nondeterminism in the usual way, by returning a set of members of $Q \times \Gamma_\varepsilon$, that is, a member of $\mathcal{P}(Q \times \Gamma_\varepsilon)$. Putting it all together, our transition function δ takes the form $\delta\colon Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$.

DEFINITION 2.13

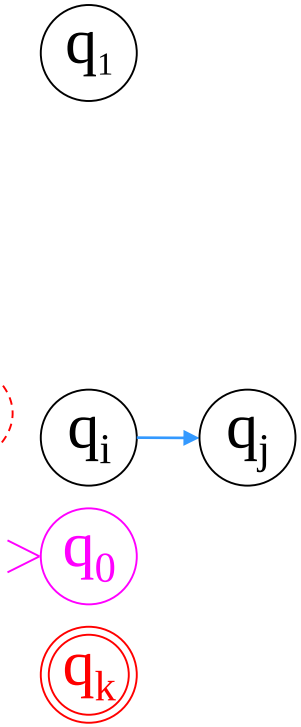
A ***pushdown automaton*** is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where Q, Σ, Γ, and F are all finite sets, and

1. Q is the set of states,
2. Σ is the input alphabet,
3. Γ is the stack alphabet,
4. $\delta\colon Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$ is the transition function,
5. $q_0 \in Q$ is the start state, and
6. $F \subseteq Q$ is the set of accept states.

Pushdown Automata

Basic idea: a pushdown automaton is a finite automaton that can optionally write to an unbounded stack.

- Finite set of states: $Q = \{q_0, q_1, q_3, \ldots, q_k\}$
- Input alphabet: Σ
- Stack alphabet: Γ
- Transition function: $\delta: Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow 2^{Q \times \Gamma^*}$
- Initial state: $q_0 \in Q$
- Final states: $F \subseteq Q$



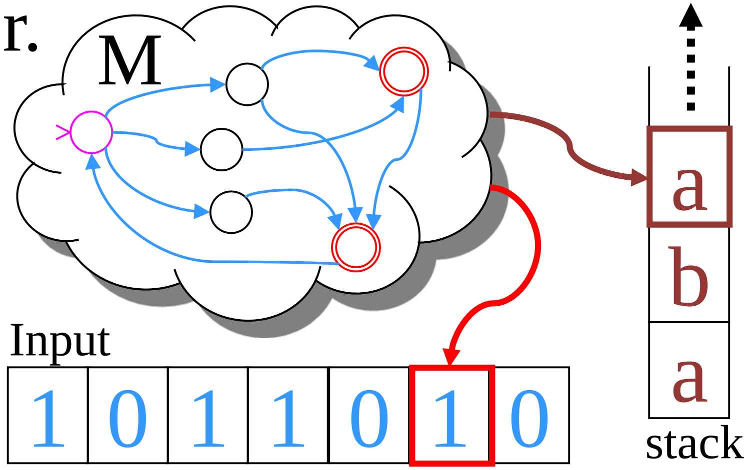
Pushdown automaton is $M=(Q, \Sigma, \Gamma, \delta, q_0, F)$

Note: pushdown automata are non-deterministic!

Pushdown Automata

A pushdown automaton can use its stack as an unbounded but access-controlled (last-in/first-out or LIFO) storage.

- A PDA accesses its stack using “push” and “pop”
- Stack & input alphabets may differ.
- Input read head only goes 1-way.
- Acceptance can be by final state or by empty-stack.



Note: a PDA can be made deterministic by restricting its transition function to unique next moves:

$$\delta: Q\times(\Sigma\cup\{\varepsilon\})\times\Gamma \rightarrow Q\times\Gamma^*$$

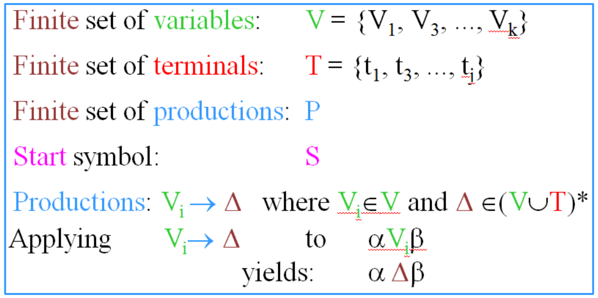
Pushdown Automata

Theorem: If a language is accepted by some context-free grammar, then it is also accepted by some PDA.

Theorem: If a language is accepted by some PDA, then it is also accepted by some context-free grammar.

Corrolary: A language is context-free iff it is also accepted by some pushdown automaton.

I.E., context-free grammars and PDAs have equivalent "computation power" or "expressiveness" capability.



Finite set of variables: $V = \{V_1, V_3, \ldots, V_k\}$

Finite set of terminals: $T = \{t_1, t_3, \ldots, t_j\}$

Finite set of productions: P

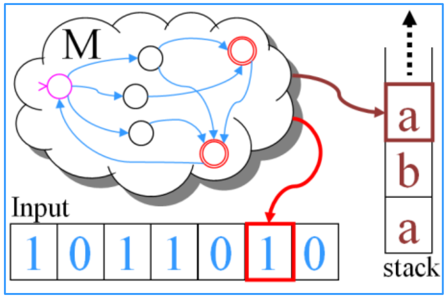
Start symbol: S

Productions: $V_i \rightarrow \Delta$ where $V_i \in V$ and $\Delta \in (V \cup T)^*$

Applying $V_i \rightarrow \Delta$ to $\alpha V_i \beta$

yields: $\alpha \Delta \beta$

$\equiv$



Closure Properties of CFLs

Theorem: The context-free languages are closed under union.

Hint: Derive a new grammar for the union.

Theorem: The CFLs are closed under Kleene closure.

Hint: Derive a new grammar for the Kleene closure.

Theorem: The CFLs are closed under $\cap$ with regular langs.

Hint: Simulate PDA and FA in parallel.

Theorem: The CFLs are not closed under intersection.

Hint: Find a counter example.

Theorem: The CFLs are not closed under complementation.

Hint: Use De Morgan’s law.

Decidable PDA / CFG Problems

Given an arbitrary pushdown automata M (or CFG G)
the following problems are decidable (i.e., have algorithms):

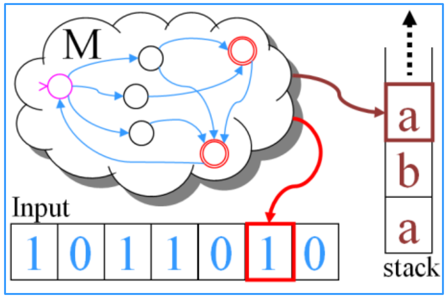
Q_1: Is L(M) = Ø ?
Q_5: Is L(G) = Ø ?

Q_2: Is L(M) finite ?
Q_6: Is L(G) finite ?

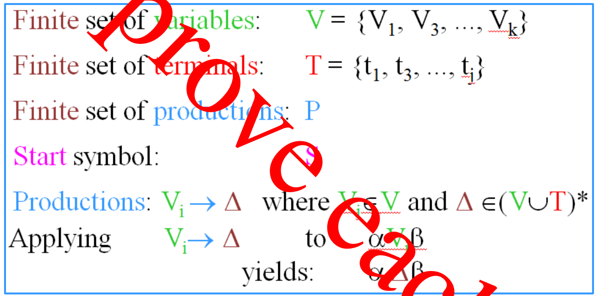
Q_3: Is L(M) infinite ?
Q_7: Is L(G) infinite ?

Q_4: Is w $\in$ L(M) ?
Q_8: Is w $\in$ L(G) ?

Extra-credit: prove each!



|||



Finite set of variables: $V = \{V_1, V_3, \ldots, V_k\}$

Finite set of terminals: $T = \{t_1, t_3, \ldots, t_j\}$

Finite set of productions: P

Start symbol: S

Productions: $V_i \rightarrow \Delta$ where $V_i \in V$ and $\Delta \in (V \cup T)^*$

Applying $V_i \rightarrow \Delta$ to $\alpha V_i \beta$
yields: $\alpha \Delta \beta$

Undecidable PDA / CFG Problems

Theorem: the following are undecidable (i.e., there exist no algorithms to answer these questions):

Q: Is PDA M minimal ?

Q: Are PDAs M_1 and M_2 equivalent ?

Q: Is CFG G minimal ?

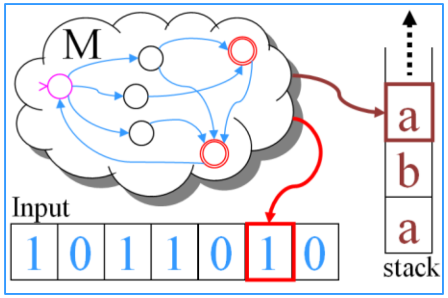
Q: Is CFG G ambiguous ?

Q: Is $L(G_1) = L(G_2)$?

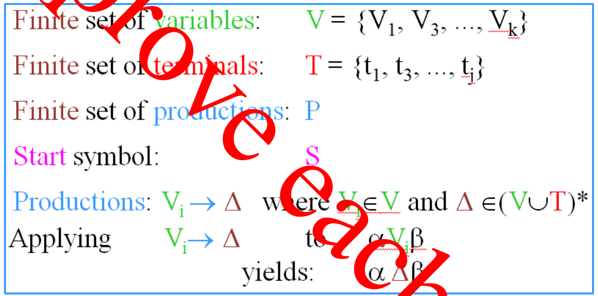
Q: Is $L(G_1) \cap L(G_2) = \emptyset$?

Q: Is CFL L inherently ambiguous ?

Extra-credit: prove each!



|||



Finite set of variables: $V = \{V_1, V_3, \ldots, V_k\}$

Finite set of terminals: $T = \{t_1, t_3, \ldots, t_j\}$

Finite set of productions: P

Start symbol: S

Productions: $V_i \rightarrow \Delta$ where $V_i \in V$ and $\Delta \in (V \cup T)^*$

Applying $V_i \rightarrow \Delta$ to $\alpha V_i \beta$

yields: $\alpha \Delta \beta$

PDA Enhancements

Theorem: 2-way PDAs are more powerful than 1-way PDAs.

Hint: Find an example non-CFL accepted by a 2-way PDA.

Theorem: 2-stack PDAs are more powerful than 1-stack PDAs.

Hint: Find an example non-CFL accepted by a 2-stack PDA.

Theorem: 1-queue PDAs are more powerful than 1-stack PDAs.

Hint: Find an example non-CFL accepted by a 1-queue PDA.

Theorem: 2-head PDAs are more powerful than 1-head PDAs.

Hint: Find an example non-CFL accepted by a 2-head PDA.

Theorem: Non-determinism increases the power of PDAs.

Hint: Find a CFL not accepted by any deterministic PDA.

Extra-credit: prove each!

1.4

NONREGULAR LANGUAGES

To understand the power of finite automata you must also understand their limitations. In this section we show how to prove that certain languages cannot be recognized by any finite automaton.

Let's take the language $B = \{0^n1^n | n \geq 0\}$. If we attempt to find a DFA that recognizes B, we discover that the machine seems to need to remember how many 0s have been seen so far as it reads the input. Because the number of 0s isn't limited, the machine will have to keep track of an unlimited number of possibilities. But it cannot do so with any finite number of states.

Next, we present a method for proving that languages such as B are not regular. Doesn't the argument already given prove nonregularity because the number of 0s is unlimited? It does not. Just because the language appears to require unbounded memory doesn't mean that it is necessarily so. It does happen to be true for the language B, but other languages seem to require an unlimited number of possibilities, yet actually they are regular. For example, consider two languages over the alphabet $\Sigma = \{0,1\}$:

$$C = \{w | \text{ } w \text{ has an equal number of 0s and 1s}\}, \text{ and}$$

$$D = \{w | \text{ } w \text{ has an equal number of occurrences of 01 and 10 as substrings}\}.$$

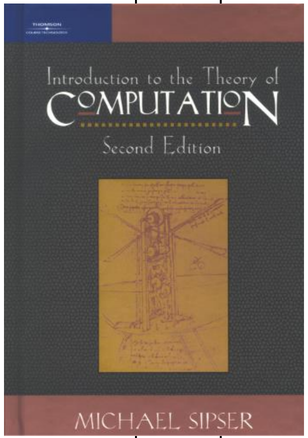
At first glance a recognizing machine appears to need to count in each case, and therefore neither language appears to be regular. As expected, C is not regular, but surprisingly D is regular![6] Thus our intuition can sometimes lead us astray, which is why we need mathematical proofs for certainty. In this section we show how to prove that certain languages are not regular.

THE PUMPING LEMMA FOR REGULAR LANGUAGES

Our technique for proving nonregularity stems from a theorem about regular languages, traditionally called the ***pumping lemma***. This theorem states that all regular languages have a special property. If we can show that a language does not have this property, we are guaranteed that it is not regular. The property states that all strings in the language can be "pumped" if they are at least as long as a certain special value, called the ***pumping length***. That means each such string contains a section that can be repeated any number of times with the resulting string remaining in the language.

---

[6] See Problem 1.48.



appearing before q_9, piece y is the part between the two appearances of q_9, and piece z is the remaining part of s, coming after the second occurrence of q_9. So x takes M from the state q_1 to q_9, y takes M from q_9 back to q_9 and z takes M from q_9 to the accept state q_{13}, as shown in the following figure.

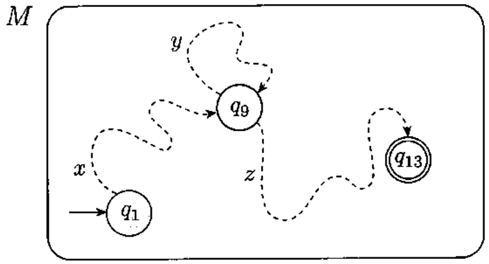


FIGURE 1.72
Example showing how the strings x, y, and z affect M

Let's see why this division of s satisfies the three conditions. Suppose that we run M on input $xyyz$. We know that x takes M from q_1 to q_9, and then the first y takes it from q_9 back to q_9, as does the second y, and then z takes it to q_{13}. With q_{13} being an accept state, M accepts input $xyyz$. Similarly, it will accept xy^iz for any $i > 0$. For the case $i = 0$, $xy^iz = xz$, which is accepted for similar reasons. That establishes condition 1.

Checking condition 2, we see that $|y| > 0$, as it was the part of s that occurred between two different occurrences of state q_9.

In order to get condition 3, we make sure that q_9 is the first repetition in the sequence. By the pigeonhole principle, the first $p+1$ states in the sequence must contain a repetition. Therefore $|xy| \leq p$.

PROOF Let $M = (Q, \Sigma, \delta, q_1, F)$ be a DFA recognizing A and p be the number of states of M.

Let $s = s_1s_2 \cdots s_n$ be a string in A of length n, where $n \geq p$. Let $r_1, \ldots, r_{n+1}$ be the sequence of states that M enters while processing s, so $r_{i+1} = \delta(r_i, s_i)$ for $1 \leq i \leq n$. This sequence has length $n + 1$, which is at least $p + 1$. Among the first $p + 1$ elements in the sequence, two must be the same state, by the pigeonhole principle. We call the first of these r_j and the second r_l. Because r_l occurs among the first $p+1$ places in a sequence starting at r_1, we have $l \leq p+1$. Now let $x = s_1 \cdots s_{j-1}$, $y = s_j \cdots s_{l-1}$, and $z = s_l \cdots s_n$.

As x takes M from r_1 to r_j, y takes M from r_j to r_j, and z takes M from r_j to r_{n+1}, which is an accept state, M must accept xy^iz for $i \geq 0$. We know that $j \neq l$, so $|y| > 0$; and $l \leq p+1$, so $|xy| \leq p$. Thus we have satisfied all conditions of the pumping lemma.

Context-Free Grammars

Def: A language is context-free if it is generated by some context-free grammar.

Theorem: All regular languages are context-free.

Proof idea: construct a grammar that "simulates" a DFA, where variables correspond to states, etc.

Theorem: Some context-free languages are not regular.

Ex: $\{0^n1^n \mid n > 0\}$

Proof by "pumping" argument: long strings in a regular language contain a pumpable substring.

$\exists\, N\in\mathbb{N} \ni \forall z\in L, |z|\geq N\ \exists\ u,v,w\in\Sigma^* \ni z=uvw,$

$|uv|\leq N, |v|\geq 1, uv^iw\in L\ \forall\ i\geq 0.$

Context-Free Grammars

Def: A language is context-free if it is generated by some context-free grammar.

Theorem: Some languages are not context-free .

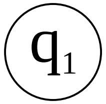
Ex: $\{0^n 1^n 2^n \mid n > 0\}$

Proof by "pumping" argument for CFL's.

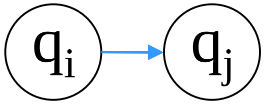
Turing Machines

Basic idea: a Turing machine is a finite automaton that can optionally write to an unbounded tape.

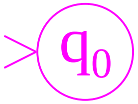
- Finite set of states: $Q = \{q_0, q_1, q_3, \ldots, q_k\}$



- Tape alphabet: Γ
- Blank symbol: $\beta \in \Gamma$
- Input alphabet: $\Sigma \subseteq \Gamma - \{\beta\}$
- Transition function: $\delta: (Q-F)\times\Gamma \to Q\times\Gamma\times\{L,R\}$



- Initial state: $q_0 \in Q$



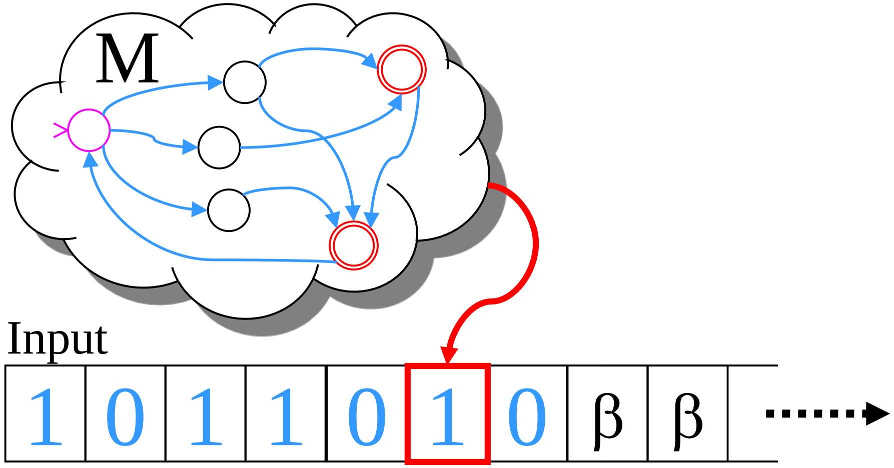
- Final states: $F \subseteq Q$

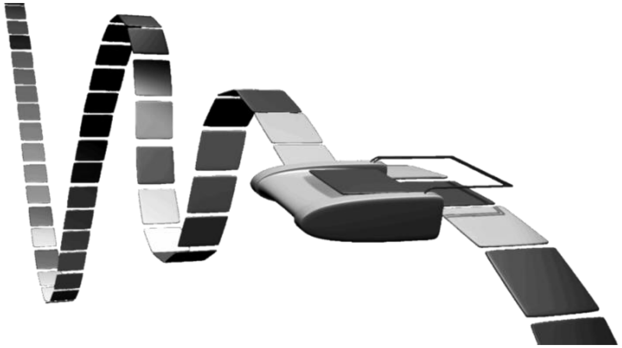
Turing machine is $M=(Q, \Gamma, \beta, \Sigma, \delta, q_0, F)$

Turing Machines

A Turing machine can use its tape as an unbounded storage but reads / writes only at head position.

- Initially the entire tape is blank, except the input portion
- Read / write head goes left / right with each transition
- Input string acceptance is by final state(s)
- A Turing machine is usually deterministic





3.2
VARIANTS OF TURING MACHINES

Alternative definitions of Turing machines abound, including versions with multiple tapes or with nondeterminism. They are called ***variants*** of the Turing machine model. The original model and its reasonable variants all have the same power—they recognize the same class of languages. In this section we describe some of these variants and the proofs of equivalence in power. We call this invariance to certain changes in the definition ***robustness***. Both finite automata and pushdown automata are somewhat robust models, but Turing machines have an astonishing degree of robustness.

To illustrate the robustness of the Turing machine model let's vary the type of transition function permitted. In our definition, the transition function forces the head to move to the left or right after each step; the head may not simply stay put. Suppose that we had allowed the Turing machine the ability to stay put. The transition function would then have the form $\delta\colon Q\times\Gamma\longrightarrow Q\times\Gamma\times\{\mathrm{L},\mathrm{R},\mathrm{S}\}$. Might this feature allow Turing machines to recognize additional languages, thus adding to the power of the model? Of course not, because we can convert any TM with the "stay put" feature to one that does not have it. We do so by replacing each stay put transition with two transitions, one that moves to the right and the second back to the left.

This small example contains the key to showing the equivalence of TM variants. To show that two models are equivalent we simply need to show that we can simulate one by the other.

MULTITAPE TURING MACHINES

A ***multitape Turing machine*** is like an ordinary Turing machine with several tapes. Each tape has its own head for reading and writing. Initially the input appears on tape 1, and the others start out blank. The transition function is changed to allow for reading, writing, and moving the heads on some or all of the tapes simultaneously. Formally, it is

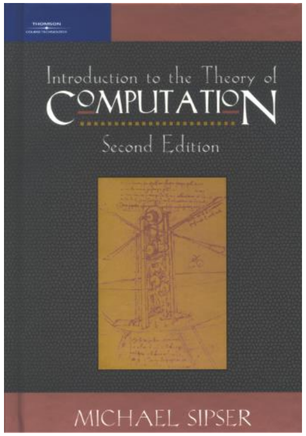
$$\delta\colon Q\times\Gamma^k\longrightarrow Q\times\Gamma^k\times\{\mathrm{L},\mathrm{R},\mathrm{S}\}^k,$$

where k is the number of tapes. The expression

$$\delta(q_i,a_1,\ldots,a_k)=(q_j,b_1,\ldots,b_k,\mathrm{L},\mathrm{R},\ldots,\mathrm{L})$$

means that, if the machine is in state q_i and heads 1 through k are reading symbols a_1 through a_k, the machine goes to state q_j, writes symbols b_1 through b_k, and directs each head to move left or right, or to stay put, as specified.

Multitape Turing machines appear to be more powerful than ordinary Turing machines, but we can show that they are equivalent in power. Recall that two machines are equivalent if they recognize the same language.



PROOF The simulating deterministic TM D has three tapes. By Theorem 3.13 this arrangement is equivalent to having a single tape. The machine D uses its three tapes in a particular way, as illustrated in the following figure. Tape 1 always contains the input string and is never altered. Tape 2 maintains a copy of N's tape on some branch of its nondeterministic computation. Tape 3 keeps track of D's location in N's nondeterministic computation tree.

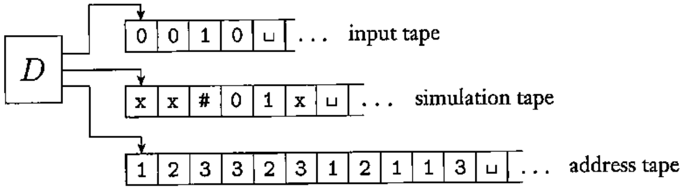


FIGURE 3.17
Deterministic TM D simulating nondeterministic TM N

Let's first consider the data representation on tape 3. Every node in the tree can have at most b children, where b is the size of the largest set of possible choices given by N's transition function. To every node in the tree we assign an address that is a string over the alphabet $\Sigma_b=\{1,2,\ldots,b\}$. We assign the address 231 to the node we arrive at by starting at the root, going to its 2nd child, going to that node's 3rd child, and finally going to that node's 1st child. Each symbol in the string tells us which choice to make next when simulating a step in one branch in N's nondeterministic computation. Sometimes a symbol may not correspond to any choice if too few choices are available for a configuration. In that case the address is invalid and doesn't correspond to any node. Tape 3 contains a string over Σ_b. It represents the branch of N's computation from the root to the node addressed by that string, unless the address is invalid. The empty string is the address of the root of the tree. Now we are ready to describe D.

1. Initially tape 1 contains the input w, and tapes 2 and 3 are empty.
2. Copy tape 1 to tape 2.
3. Use tape 2 to simulate N with input w on one branch of its nondeterministic computation. Before each step of N consult the next symbol on tape 3 to determine which choice to make among those allowed by N's transition function. If no more symbols remain on tape 3 or if this nondeterministic choice is invalid, abort this branch by going to stage 4. Also go to stage 4 if a rejecting configuration is encountered. If an accepting configuration is encountered, *accept* the input.
4. Replace the string on tape 3 with the lexicographically next string. Simulate the next branch of N's computation by going to stage 2.

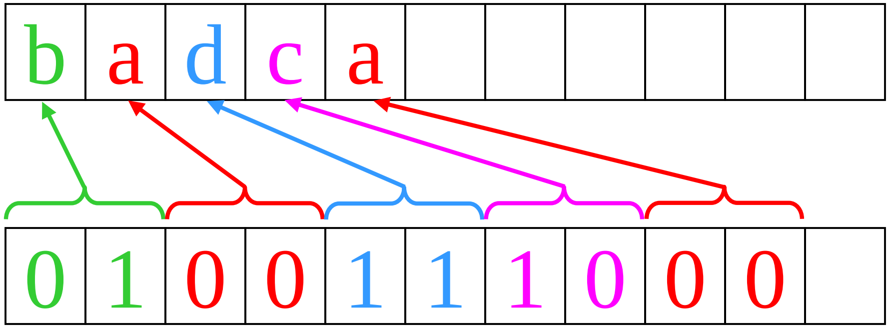
Turing Machine “Enhancements”

Larger alphabet:

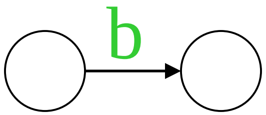
old: Σ={0,1}   new: Σ’ ={a,b,c,d}

Idea: Encode larger alphabet using smaller one.

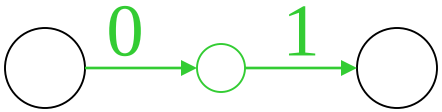
Encoding example: a=00, b=01, c=10, d=11



old: δ

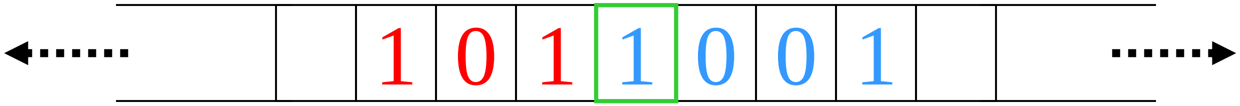


new: δ'

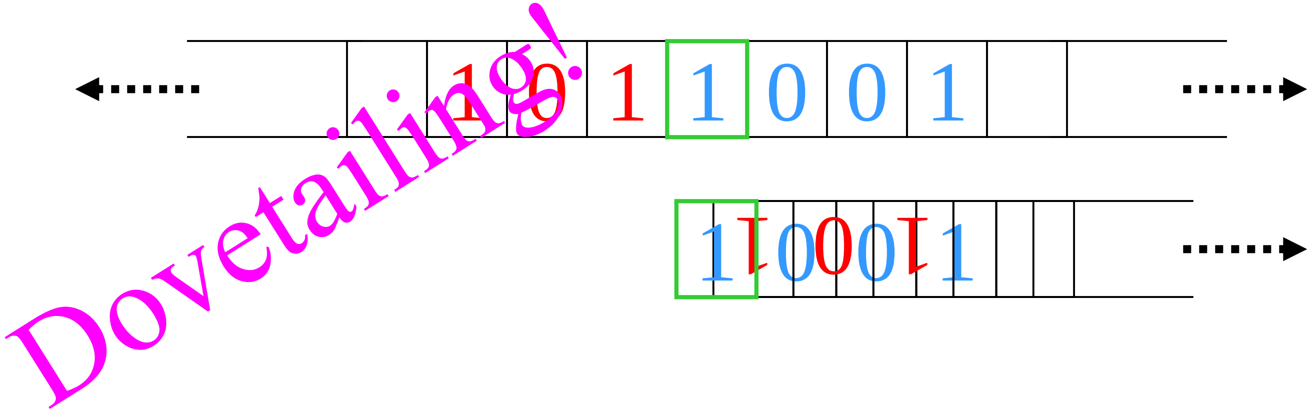


Turing Machine “Enhancements”

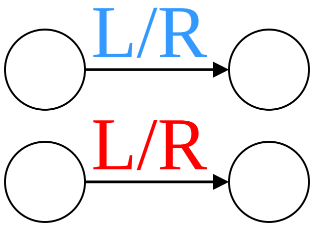
Double-sided infinite tape:



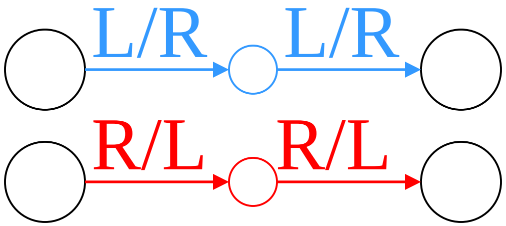
Idea: Fold into a normal single-sided infinite tape



old: δ

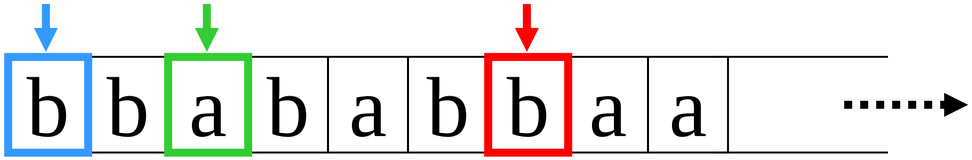


new: δ'

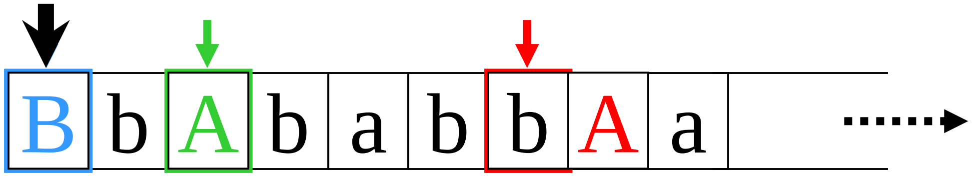


Turing Machine "Enhancements"

Multiple heads:



Idea: Mark heads locations on tape and simulate

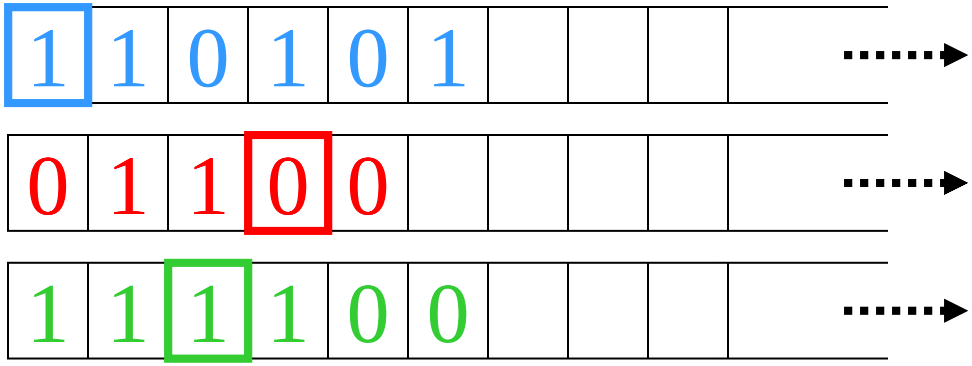


Modified δ' processes each "virtual" head independently:

- Each move of δ is simulated by a long scan & update
- δ' updates & marks all "virtual" head positions

Turing Machine “Enhancements”

Multiple tapes:



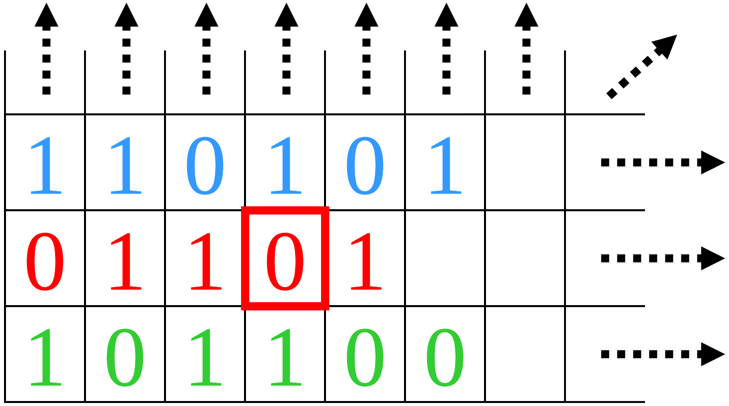
Idea: Interlace multiple tapes into a single tape

Modified δ' processes each “virtual” tape independently:

- Each move of δ is simulated by a long scan & update
- δ' updates R/W head positions on all “virtual tapes”

Turing Machine "Enhancements"

Two-dimensional tape:



1	1	0	1	0	1		
0	1	1	0	1			
1	0	1	1	0	0		

This is how compilers implement 2D arrays!

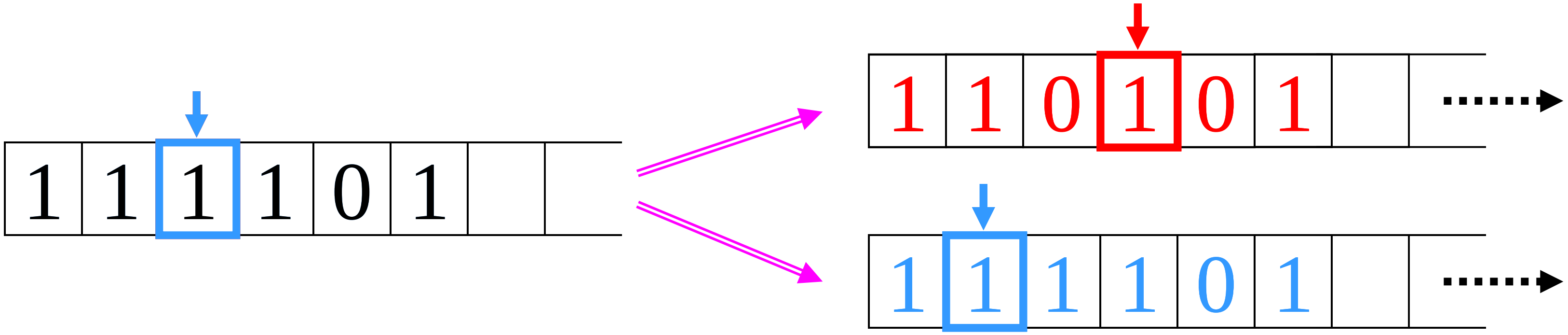
Idea: Flatten 2-D tape into a 1-D tape

Modified 1-D δ' simulates the original 2-D δ:

- Left/right δ moves: δ' moves horizontally
- Up/down δ moves: δ' jumps between tape sections

Turing Machine “Enhancements”

Non-determinism:



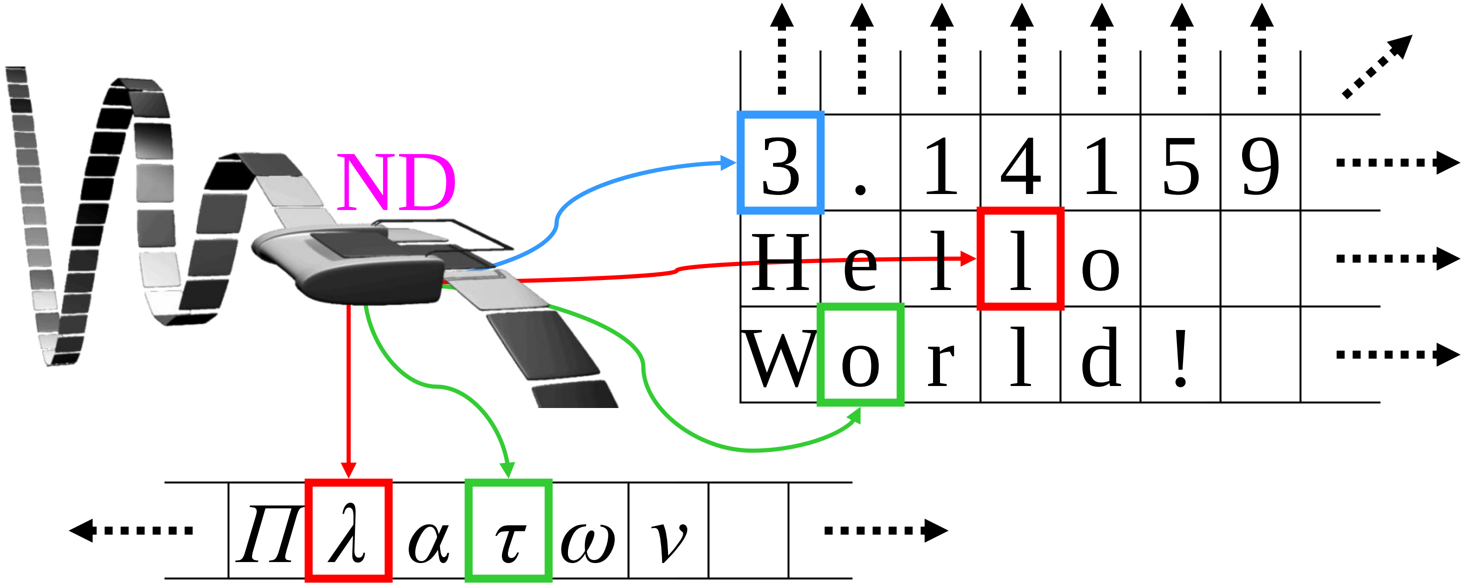
Idea: Parallel-simulate non-deterministic threads

Modified deterministic δ' simulates the original ND δ:

- Each ND move by δ spawns another independent “thread”
- All current threads are simulated “in parallel”

Turing Machine "Enhancements"

Combinations:

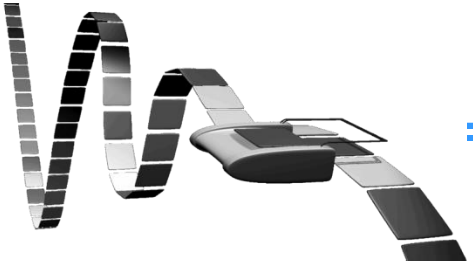


Idea: "Enhancements" are independent (and commutative with respect to preserving the language recognized).

Theorem: Combinations of "enhancements" do not increase the power of Turing machines.

Turing -Recognizable vs. -Decidable

w→
Input



⟹

√
Accept
& halt

×
Reject
& halt

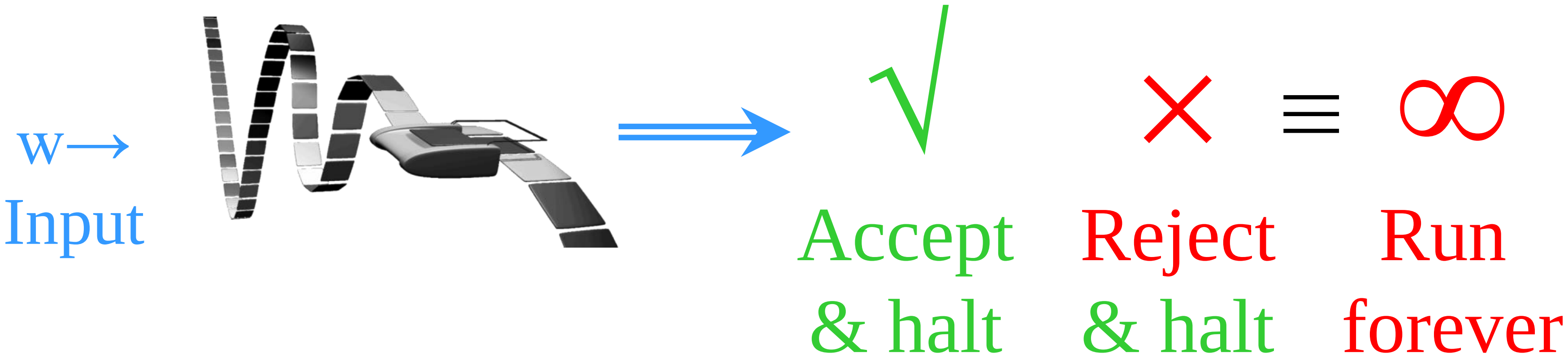
Never
runs
forever

Def: A language is Turing-decidable iff it is exactly the set of strings accepted by some always-halting TM.

$w \in \Sigma^*$ =	a	b	aa	ab	ba	bb	aaa	aab	aba	abb	baa	bab	bba	bbb	aaaa	…
M(w)⟹	√	×	√	×	×	×	√	×	×	×	×	×	×	×	√	…
L(M) =	{ a,		aa,				aaa,								aaaa	…}

Note: M must always halt on every input.

Turing -Recognizable vs. -Decidable


w→
Input

√ Accept & halt

× Reject & halt

≡

∞ Run forever

Def: A language is Turing-recognizable iff it is exactly the set of strings accepted by some Turing machine.

$w \in \Sigma^* =$	a	b	aa	ab	ba	bb	aaa	aab	aba	abb	baa	bab	bba	bbb	aaaa	…
M(w)⇒	√	×	√	∞	×	∞	√	∞	∞	×	×	×	∞	×	√	…
L(M) =	{ a,		aa,				aaa,								aaaa	…}

Note: M can run forever on an input, which is implicitly a reject (since it is not an accept).

Recognition vs. Enumeration

Def: “Decidable” means “Turing-decidable”
“Recognizable” means “Turing-recognizable”

Theorem: Every decidable language is also recognizable.

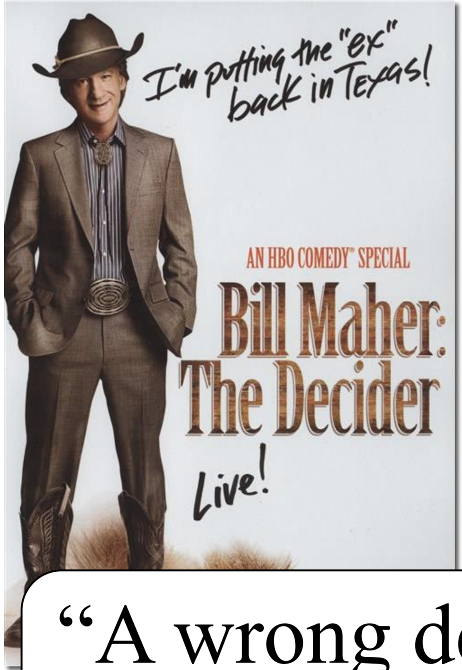
Theorem: Some recognizable languages are not decidable.

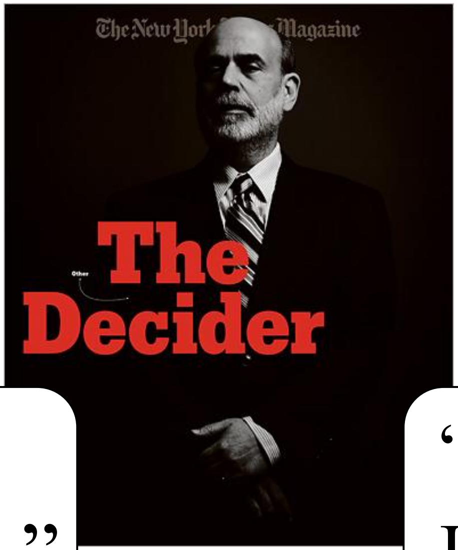
Ex: The halting problem is recognizable but not decidable.

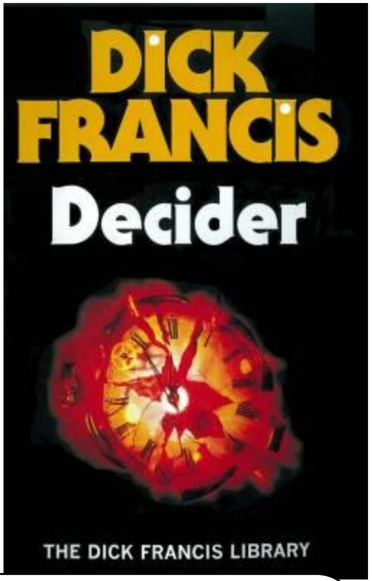
Note: Decidability is a special case of recognizability.

Note: It is easier to recognize than to decide.

Famous Deciders

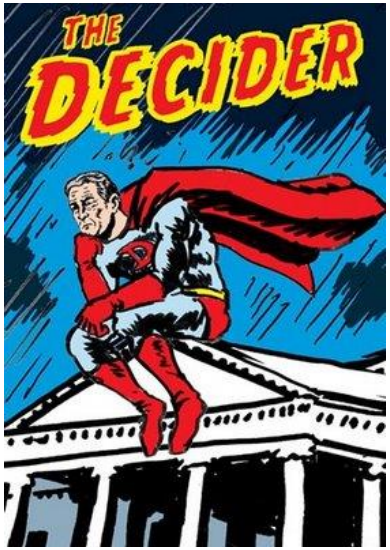




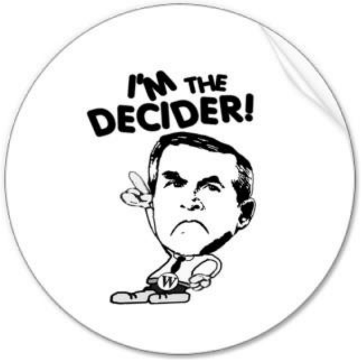


"A wrong decision is better than indecision."

"I'm the decider, and I decide what is best."



Famous Deciders



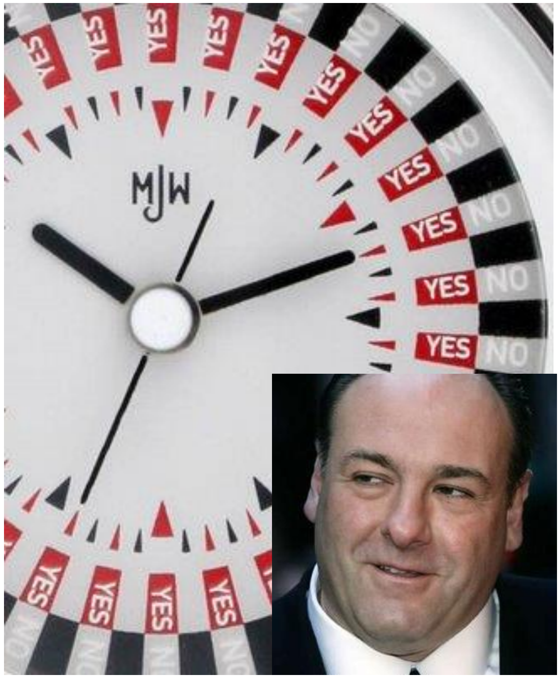

the DECIDER party

FREE

with The Wooden Birds
Evangelicals
The Lemurs
Low Line Caller
and Eugene Mirman
+ DJ Sean O'Neal from The A.V. Club

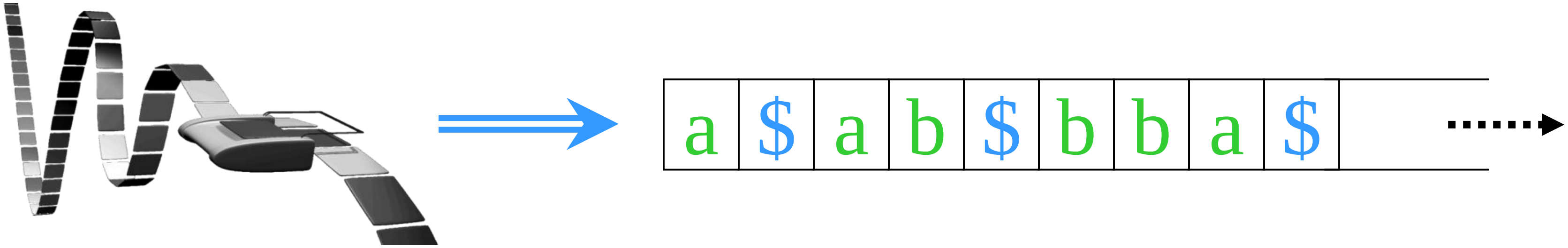
Monday, March 16, 8PM
The Mohawk, 912 Red River

Hosted by:
DECIDER.com
the ONION
thebarbariangroup



Recognition and Enumeration

Def: An "enumerator" Turing machine for a language L prints out precisely all strings of L on its output tape.



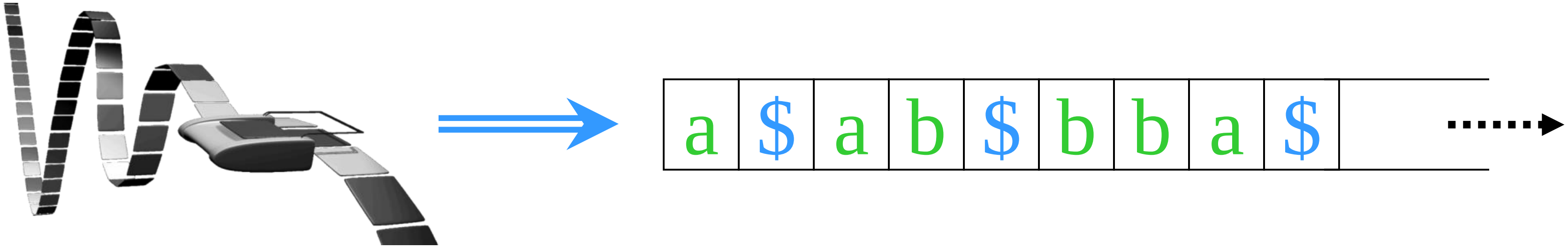
Note: The order of enumeration may be arbitrary.

Theorem: If a language is decidable, it can be enumerated in lexicographic order by some Turing machine.

Theorem: If a language can be enumerated in lexicographic order by some TM, it is decidable.

Recognition and Enumeration

Def: An "enumerator" Turing machine for a language L prints out precisely all strings of L on its output tape.



Note: The order of enumeration may be arbitrary.

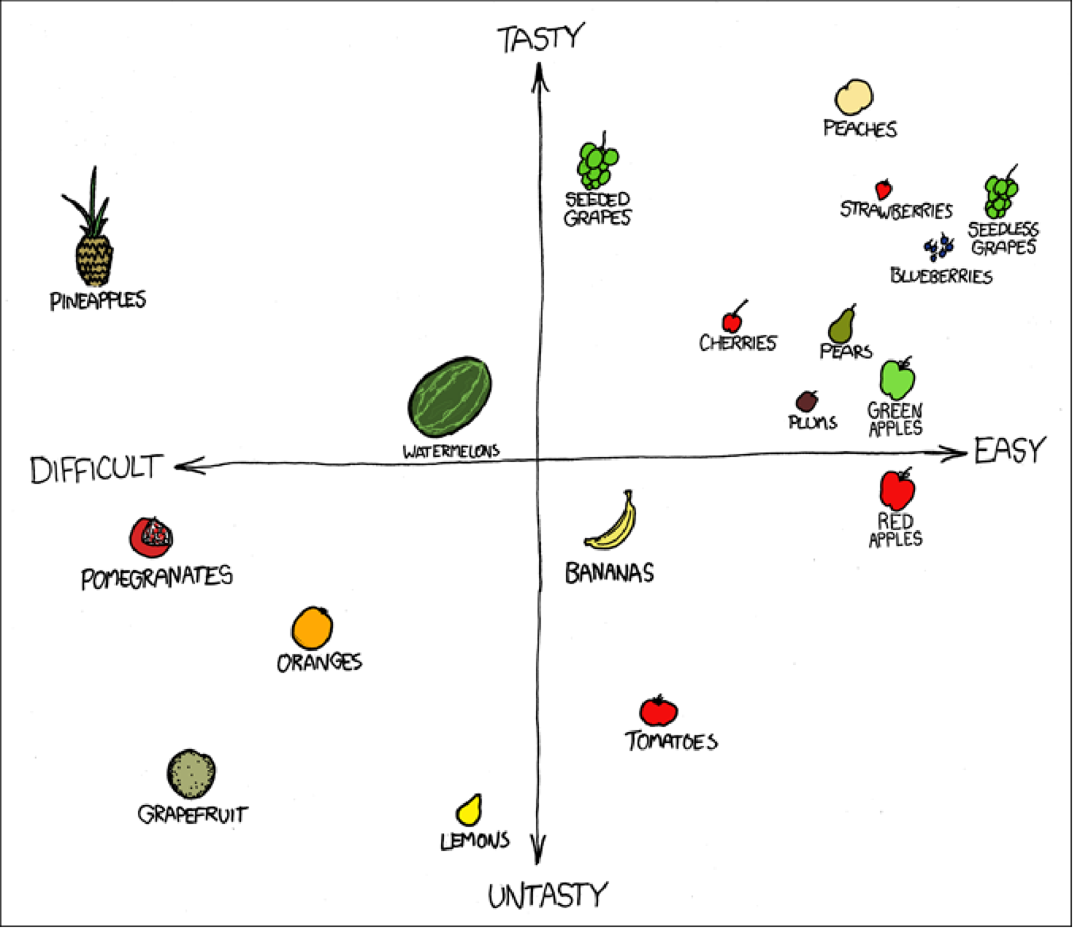
Theorem: If a language is recognizable, then it can be enumerated by some Turing machine.

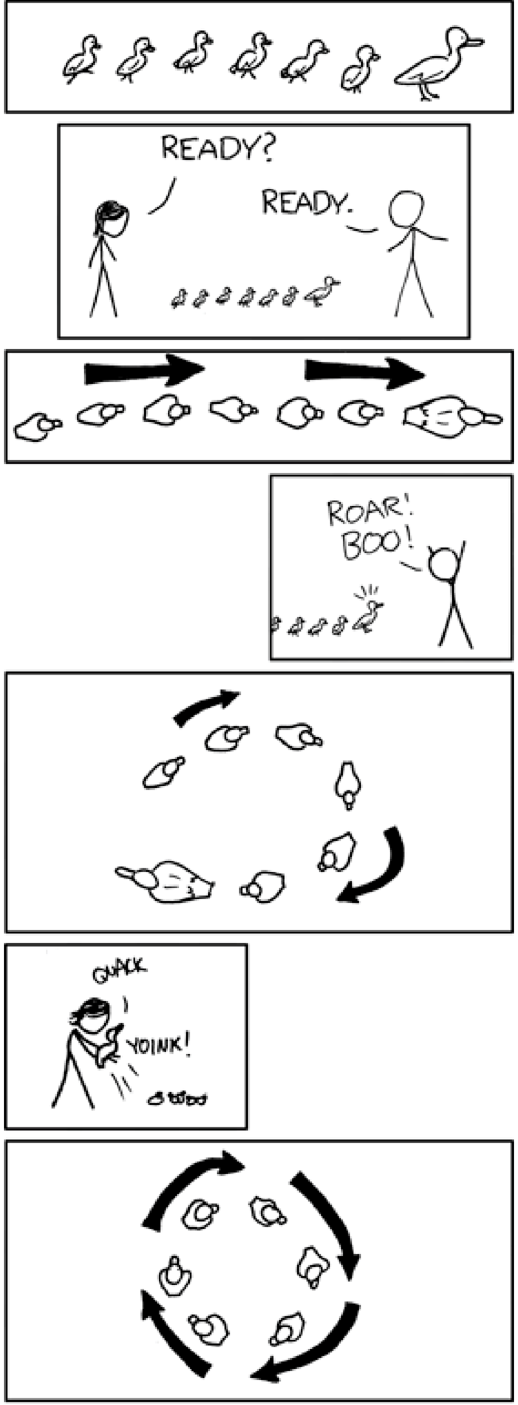
Theorem: If a language can be enumerated by some TM, then it is recognizable.

THE ALPHABET IN ALPHABETICAL ORDER

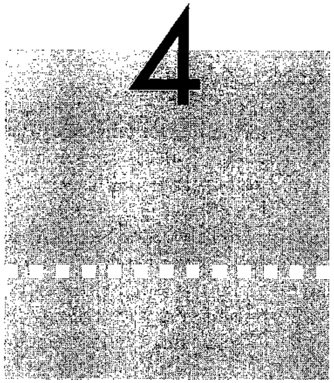
- Aitch
- Are
- Ay
- Bee
- Cue
- Dee
- Double u
- Ee
- Ef
- El
- Em
- En
- Ess
- Ex
- Eye
- Gee
- Jay
- Kay
- Oh
- Pea
- See
- Tee
- Vee
- Wy
- Yu
- Zee

S. Harris





OPERATION: DUCKLING LOOP

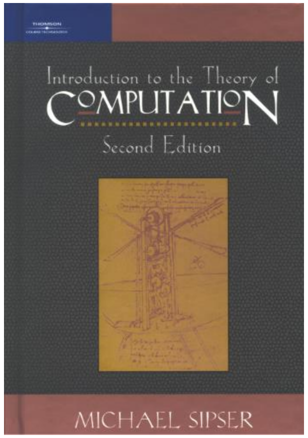


DECIDABILITY

In Chapter 3 we introduced the Turing machine as a model of a general purpose computer and defined the notion of algorithm in terms of Turing machines by means of the Church–Turing thesis.

In this chapter we begin to investigate the power of algorithms to solve problems. We demonstrate certain problems that can be solved algorithmically and others that cannot. Our objective is to explore the limits of algorithmic solvability. You are probably familiar with solvability by algorithms because much of computer science is devoted to solving problems. The unsolvability of certain problems may come as a surprise.

Why should you study unsolvability? After all, showing that a problem is unsolvable doesn't appear to be of any use if you have to solve it. You need to study this phenomenon for two reasons. First, knowing when a problem is algorithmically unsolvable *is* useful because then you realize that the problem must be simplified or altered before you can find an algorithmic solution. Like any tool, computers have capabilities and limitations that must be appreciated if they are to be used well. The second reason is cultural. Even if you deal with problems that clearly are solvable, a glimpse of the unsolvable can stimulate your imagination and help you gain an important perspective on computation.


Introduction to the Theory of
COMPUTATION
Second Edition
MICHAEL SIPSER

COROLLARY 3.18

A language is Turing-recognizable if and only if some nondeterministic Turing machine recognizes it.

PROOF Any deterministic TM is automatically a nondeterministic TM, and so one direction of this theorem follows immediately. The other direction follows from Theorem 3.16.

We can modify the proof of Theorem 3.16 so that if N always halts on all branches of its computation, D will always halt. We call a nondeterministic Turing machine a ***decider*** if all branches halt on all inputs. Exercise 3.3 asks you to modify the proof in this way to obtain the following corollary to Theorem 3.16.

COROLLARY 3.19

A language is decidable if and only if some nondeterministic Turing machine decides it.

ENUMERATORS

As we mentioned earlier, some people use the term *recursively enumerable language* for Turing-recognizable language. That term originates from a type of Turing machine variant called an ***enumerator***. Loosely defined, an enumerator is a Turing machine with an attached printer. The Turing machine can use that printer as an output device to print strings. Every time the Turing machine wants to add a string to the list, it sends the string to the printer. Exercise 3.4 asks you to give a formal definition of an enumerator. The following figure depicts a schematic of this model.

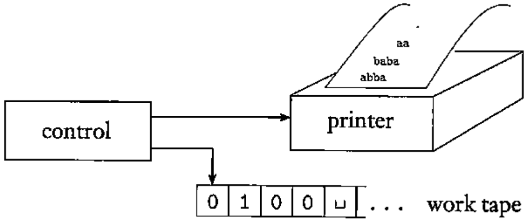
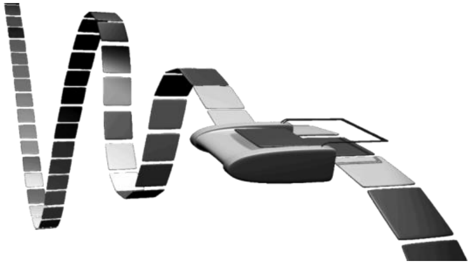


FIGURE 3.20
Schematic of an enumerator

Decidability

w→
Input



⟹

√
Accept
& halt

×
Reject
& halt

Never
runs
forever

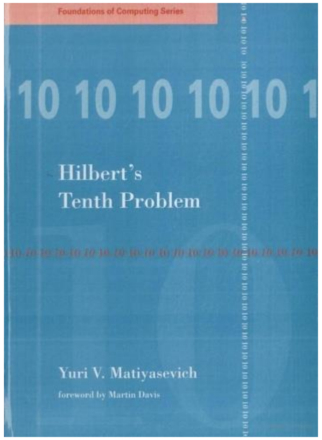
Def: A language is Turing-decidable iff it is exactly the set of strings accepted by some always-halting TM.

Theorem: The finite languages are decidable.

Theorem: The regular languages are decidable.

Theorem: The context-free languages are decidable.

A “Simple” Example

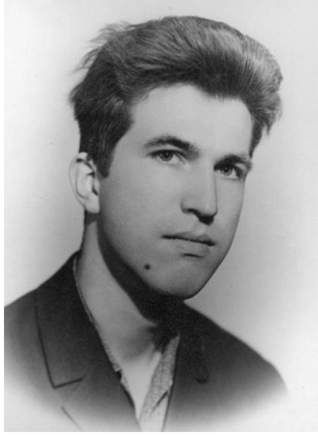


Foundations of Computing Series

Hilbert's Tenth Problem

Yuri V. Matiyasevich

foreword by Martin Davis



Let S = $\{x^3 + y^3 + z^3 \mid x, y, z \in \mathbb{Z}\}$

Q: Is S infinite?
A: Yes, since S contains all cubes.

Q: Is S Turing-recognizable?
A: Yes, since dovetailing TM can enumerate S.

Q: Is S Turing-decidable?
A: Unknown!

Q: Is $29 \in S$?
A: Yes, since $3^3+1^3+1^3=29$

Q: Is $30 \in S$?
A: Yes, since $(2220422932)^3+(-2218888517)^3+(-283059965)^3=30$

Q: Is $33 \in S$?
A: Unknown!

Theorem [Matiyasevich, 1970]: Hilbert’s 10th problem (1900), namely of determining whether a given Diophantine (i.e., multi-variable polynomial) equation has any integer solutions, is not decidable.

Closure Properties of Decidable Languages

Theorem: The decidable languages are closed under union.

Hint: use simulation.

Theorem: The decidable languages are closed under $\cap$.

Hint: use simulation.

Theorem: The decidable langs are closed under complement.

Hint: simulate and negate.

Theorem: The decidable langs are closed under concatenation.

Hint: guess-factor string and simulate.

Theorem: The decidable langs are closed under Kleene star.

Hint: guess-factor string and simulate.

Closure Properties of Recognizable Languages

Theorem: The recognizable languages are closed under union.

Hint: use simulation.

Theorem: The recognizable languages are closed under $\cap$.

Hint: use simulation.

Theorem: The recognizable langs are not closed under compl.

Hint: reduction from halting problem.

Theorem: The recognizable langs are closed under concat.

Hint: guess-factor string and simulate.

Theorem: The recognizable langs are closed under Kleene star.

Hint: guess-factor string and simulate.

5.3

MAPPING REDUCIBILITY

We have shown how to use the reducibility technique to prove that various problems are undecidable. In this section we formalize the notion of reducibility. Doing so allows us to use reducibility in more refined ways, such as for proving that certain languages are not Turing-recognizable and for applications in complexity theory.

The notion of reducing one problem to another may be defined formally in one of several ways. The choice of which one to use depends on the application. Our choice is a simple type of reducibility called ***mapping reducibility***.[2]

Roughly speaking, being able to reduce problem A to problem B by using a mapping reducibility means that a computable function exists that converts instances of problem A to instances of problem B. If we have such a conversion function, called a *reduction*, we can solve A with a solver for B. The reason is that any instance of A can be solved by first using the reduction to convert it to an instance of B and then applying the solver for B. A precise definition of mapping reducibility follows shortly.

COMPUTABLE FUNCTIONS

A Turing machine computes a function by starting with the input to the function on the tape and halting with the output of the function on the tape.

DEFINITION 5.17

A function $f\colon \Sigma^* \longrightarrow \Sigma^*$ is a ***computable function*** if some Turing machine M, on every input w, halts with just $f(w)$ on its tape.

EXAMPLE 5.18

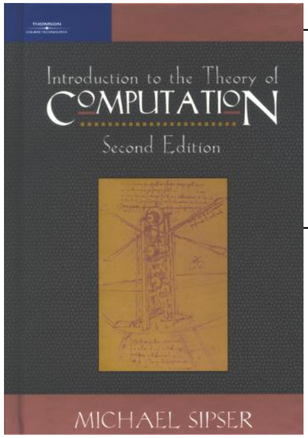
All usual arithmetic operations on integers are computable functions. For example, we can make a machine that takes input $\langle m, n\rangle$ and returns $m + n$, the sum of m and n. We don't give any details here, leaving them as exercises.

EXAMPLE 5.19

Computable functions may be transformations of machine descriptions. For example, one computable function f takes input w and returns the description of a Turing machine $\langle M'\rangle$ if $w = \langle M\rangle$ is an encoding of a Turing machine M.

---

[2] It is called ***many–one reducibility*** in some other textbooks.



The machine M' is a machine that recognizes the same language as M, but never attempts to move its head off the left-hand end of its tape. The function f accomplishes this task by adding several states to the description of M. The function returns ε if w is not a legal encoding of a Turing machine.

FORMAL DEFINITION OF MAPPING REDUCIBILITY

Now we define mapping reducibility. As usual we represent computational problems by languages.

DEFINITION 5.20

Language A is ***mapping reducible*** to language B, written $A \leq_m B$, if there is a computable function $f\colon \Sigma^* \longrightarrow \Sigma^*$, where for every w,

$$w \in A \iff f(w) \in B.$$

The function f is called the ***reduction*** of A to B.

The following figure illustrates mapping reducibility.

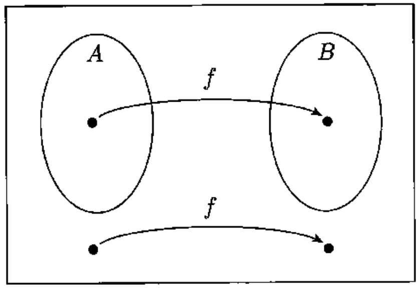


FIGURE 5.21
Function f reducing A to B

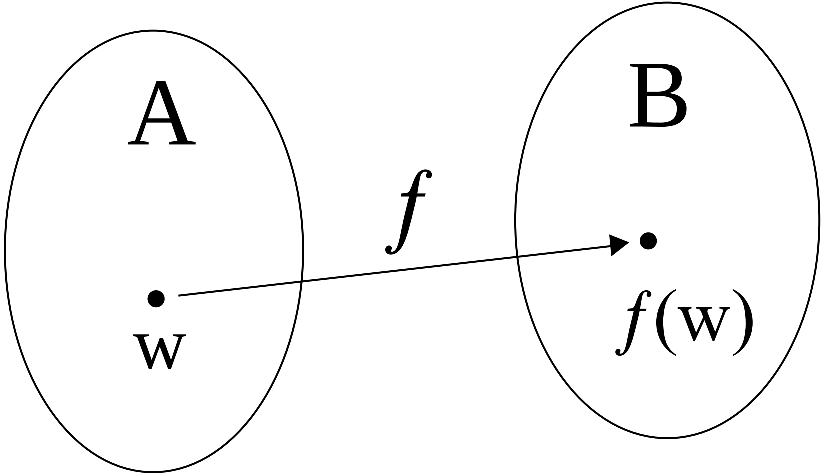
A mapping reduction of A to B provides a way to convert questions about membership testing in A to membership testing in B. To test whether $w \in A$, we use the reduction f to map w to $f(w)$ and test whether $f(w) \in B$. The term *mapping reduction* comes from the function or mapping that provides the means of doing the reduction.

If one problem is mapping reducible to a second, previously solved problem, we can thereby obtain a solution to the original problem. We capture this idea in the following theorem.

Reducibilities

Def: A language A is reducible to a language B if $\exists$ computable function/map $f:\Sigma^*\rightarrow\Sigma^*$ where

$\forall w \quad w\in A \Leftrightarrow f(w)\in B$



Note: f is called a "reduction" of A to B

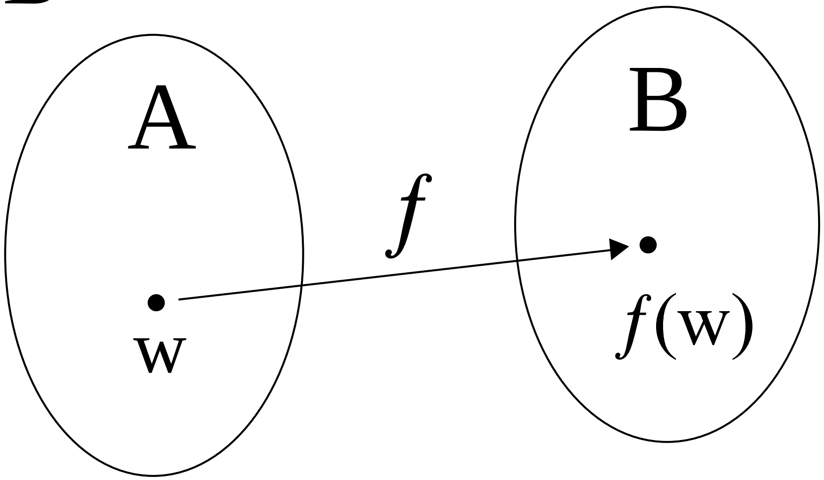
Denotation: $A \le B$

Intuitively, A is "no harder" than B

Reducibilities

Def: A language A is reducible to a language B if $\exists$ computable function/map $f:\Sigma^*\rightarrow\Sigma^*$ where

$$\forall w \quad w\in A \Leftrightarrow f(w)\in B$$



Theorem: If $A \leq B$ and B is decidable then A is decidable.

Theorem: If $A \leq B$ and A is undecidable then B is undecidable.

Note: be very careful about the mapping direction!

Reduction Example 1

Def: Let H_ε be the halting problem for TMs running on $w=\varepsilon$.

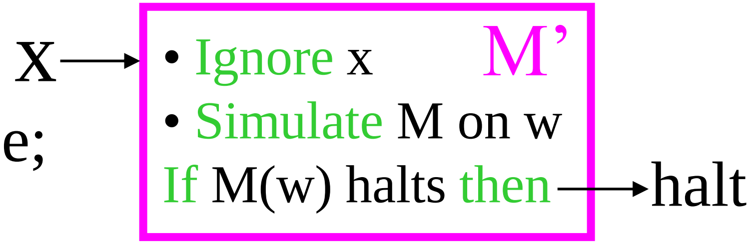
"Does TM M halt on ε?" $H_\varepsilon = \{ <M> \in \Sigma^* \mid M(\varepsilon) \text{ halts} \}$

Theorem: H_ε is <u>not</u> decidable.

Proof: Reduction from the Halting Problem H:

Given an arbitrary TM M and input w, construct new TM M'

that <u>if it ran</u> on input x, it would:



x →

- Ignore x
- Simulate M on w

If M(w) halts then → halt

1. Overwrite x with the <u>fixed</u> w on tape;
2. Simulate M on the <u>fixed</u> input w;
3. Accept ⇔ M accepts w.

Note: M' is not run!

Note: M' halts on ε (and on any $x \in \Sigma^*$) ⇔ M halts on w.

A decider (oracle) for H_ε can thus be used to decide H!

Since H is undecidable, H_ε must be undecidable also. ■

Reduction Example 2

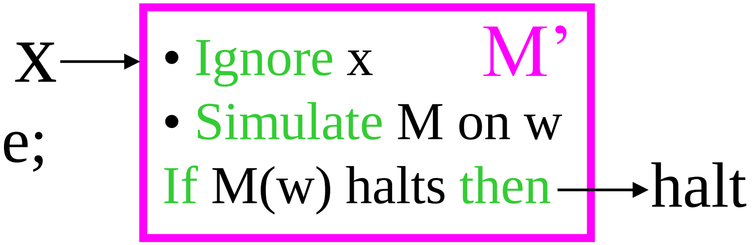
Def: Let $L_\emptyset$ be the emptyness problem for TMs.

“Is L(M) empty?” $L_\emptyset = \{ \langle M \rangle \in \Sigma^* \mid L(M) = \emptyset \}$

Theorem: $L_\emptyset$ is <u>not</u> decidable.

Proof: Reduction from the Halting Problem H:

Given an arbitrary TM M and input w, construct new TM M’ that <u>if it ran</u> on input x, it would:



x →

M’

- Ignore x
- Simulate M on w

If M(w) halts then → halt

1. Overwrite x with the <u>fixed</u> w on tape;
2. Simulate M on the <u>fixed</u> input w;
3. Accept ⇔ M accepts w.

Note: M’ is not run!

Note: M’ halts on every $x \in \Sigma^*$ ⇔ M halts on w.

A decider (oracle) for $L_\emptyset$ can thus be used to decide H!

Since H is undecidable, $L_\emptyset$ must be undecidable also. ■

Reduction Example 3

Def: Let L_{reg} be the regularity problem for TMs.

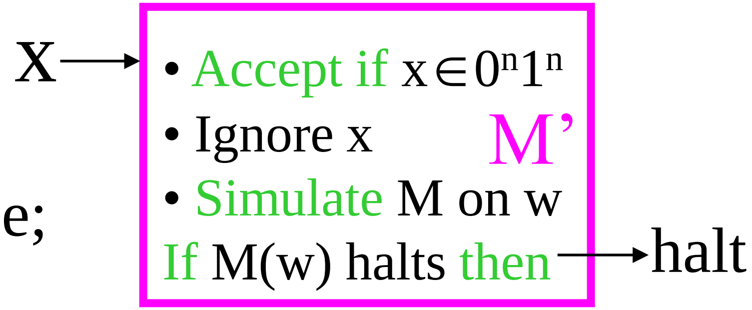
"Is L(M) regular?" $L_{reg} = \{ \langle M\rangle \in \Sigma^* \mid L(M) \text{ is regular } \}$

Theorem: L_{reg} is <u>not</u> decidable.

Proof: Reduction from the Halting Problem H:

Given an arbitrary TM M and input w, construct new TM M'

that <u>if it ran</u> on input x, it would:



1. Accept if $x \in 0^n1^n$
2. Overwrite x with the <u>fixed</u> w on tape;
3. Simulate M on the <u>fixed</u> input w;
4. Accept $\Leftrightarrow$ M accepts w.

Note: M' is not run!

Note: $L(M') = \Sigma^* \Leftrightarrow$ M halts on w

$L(M') = 0^n1^n \Leftrightarrow$ M does not halt on w

■

A decider (oracle) for L_{reg} can thus be used to decide H!

Rice's Theorem

Def: Let a "property" P be a set of recognizable languages.

Ex: $P_1=\{L \mid L \text{ is a decidable language}\}$

$P_2=\{L \mid L \text{ is a context-free language}\}$

$P_3=\{L \mid L = L^*\}$

$P_4=\{\{\varepsilon\}\}$

$P_5= \emptyset$

$P_6=\{L \mid L \text{ is a recognizable language}\}$

The two trivial properties

L is said to "have property P" iff $L \in P$

Ex: $(a+b)^*$ has property P_1, P_2, P_3 & P_6 but not P_4 or P_5

$\{ww^R\}$ has property P_1, P_2, & P_6 but not P_3, P_4 or P_5

Def: A property is "trivial" iff it is empty or it contains all recognizable languages.

Rice’s Theorem

Theorem: The two trivial properties are decidable.

Proof:

$P_{none} = \varnothing$

x $\rightarrow$

- Ignore x
- Say “no”
- Stop

M_{none} $\rightarrow$ no

M_{none} decides P_{none}

$P_{all}=\{L \mid L \text{ is a recognizable language}\}$

x $\rightarrow$

- Ignore x
- Say “yes”
- Stop

M_{all} $\rightarrow$ yes

M_{all} decides P_{all}

Q: What other properties (other than P_{none} and P_{all}) are decidable?

A: None!

Rice's Theorem

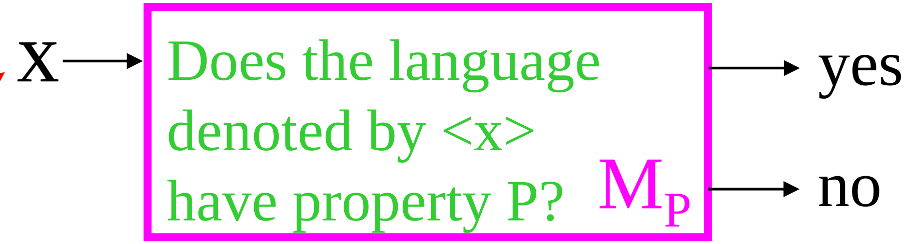
Theorem [Rice, 1951]: All non-trivial properties of the Turing-recognizable languages are not decidable.

Proof: Let P be a non-trivial property.

Without loss of generality assume $\emptyset \notin P$, otherwise substitute P's complement for P in the remainder of this proof.

Select $L \in P$ (note that $L \neq \emptyset$ since $\emptyset \notin P$), and let M_L recognize L (i.e., $L(M_L) = L \neq \emptyset$).

Assume (towards contradiction) that $\exists$ some TM M_P which decides property P:



x

Does the language denoted by <x> have property P?

M_P

yes

no

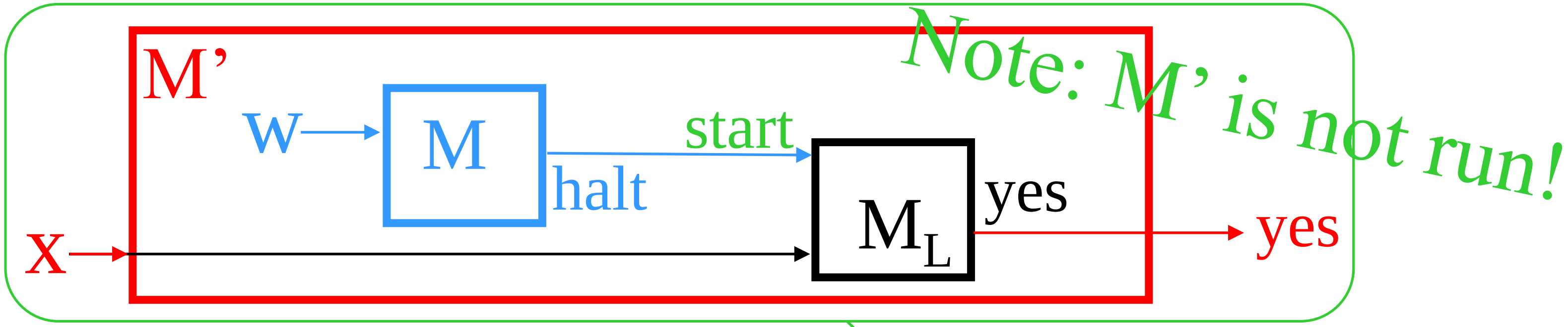
Note: x can be e.g., a TM description.

Rice’s Theorem

Reduction strategy: use M_p to “solve” the halting problem.

Recall that $L \in P$, and let M_L recognize L (i.e., $L(M_L)=L \neq Ø$).

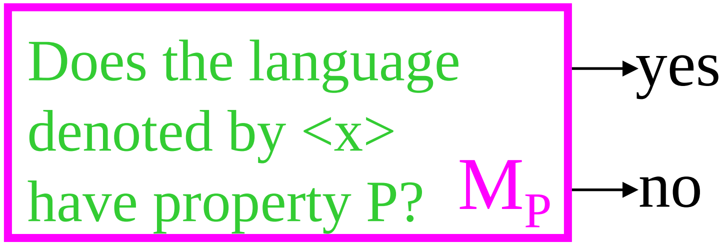
Given an arbitrary TM M & string w, construct M’:



What is the language of M’?

L(M’) is either Ø or $L(M_L)=L$

If M halts on w then L(M’)$=L(M_L)= L$



If M does not halt on w then L(M’)= Ø since M_L never starts

=> M halts on w iff L(M’) has property P

“Oracle” M_P can determine if L(M’) has property P,

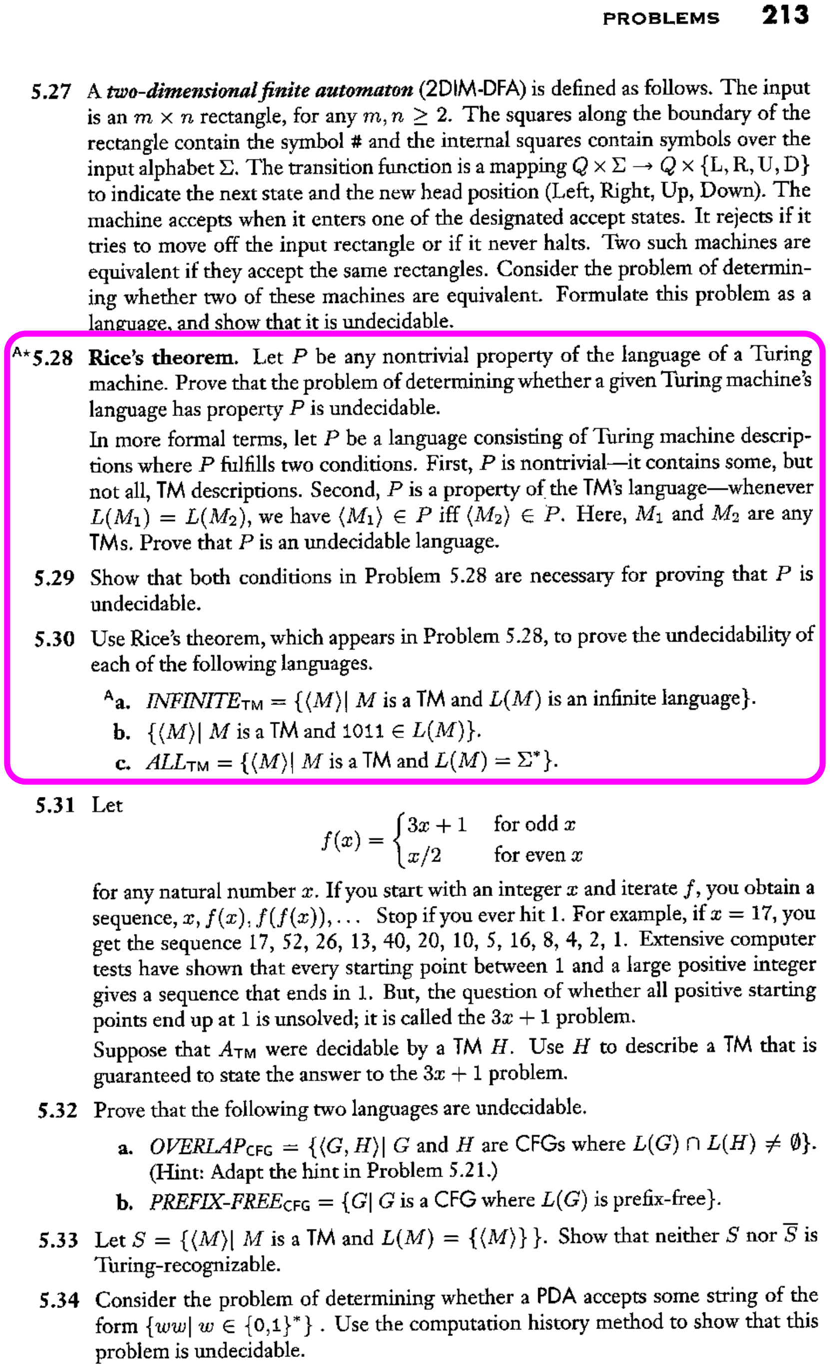
and thereby “solve” the halting problem, a contradiction! ■

Rice’s Theorem

Corollary: The following questions are not decidable: given a TM, is its language L:

- Empty?
- Finite?
- Infinite?
- Co-finite?
- Regular?
- Context-free?
- Inherently ambiguous?
- Decidable?
- $L = \Sigma^*$?
- L contains an odd string?
- L contains a palindrome?
- L = {Hello, World} ?
- L is NP-complete?
- L is in PSPACE?

Warning: Rice’s theorem applies to properties (i.e., sets of languages), not (directly to) TM’s or other object types!



5.27 A ***two-dimensional finite automaton*** (2DIM-DFA) is defined as follows. The input is an $m \times n$ rectangle, for any $m, n \geq 2$. The squares along the boundary of the rectangle contain the symbol # and the internal squares contain symbols over the input alphabet Σ. The transition function is a mapping $Q \times \Sigma \rightarrow Q \times \{\mathrm{L}, \mathrm{R}, \mathrm{U}, \mathrm{D}\}$ to indicate the next state and the new head position (Left, Right, Up, Down). The machine accepts when it enters one of the designated accept states. It rejects if it tries to move off the input rectangle or if it never halts. Two such machines are equivalent if they accept the same rectangles. Consider the problem of determining whether two of these machines are equivalent. Formulate this problem as a language, and show that it is undecidable.

A★**5.28** **Rice's theorem.** Let P be any nontrivial property of the language of a Turing machine. Prove that the problem of determining whether a given Turing machine's language has property P is undecidable.

In more formal terms, let P be a language consisting of Turing machine descriptions where P fulfills two conditions. First, P is nontrivial—it contains some, but not all, TM descriptions. Second, P is a property of the TM's language—whenever $L(M_1) = L(M_2)$, we have $\langle M_1 \rangle \in P$ iff $\langle M_2 \rangle \in P$. Here, M_1 and M_2 are any TMs. Prove that P is an undecidable language.

5.29 Show that both conditions in Problem 5.28 are necessary for proving that P is undecidable.

5.30 Use Rice's theorem, which appears in Problem 5.28, to prove the undecidability of each of the following languages.

A**a.** $INFINITE_{\mathsf{TM}} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) \text{ is an infinite language}\}$.

b. $\{\langle M \rangle \mid M \text{ is a TM and } 1011 \in L(M)\}$.

c. $ALL_{\mathsf{TM}} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) = \Sigma^*\}$.

5.31 Let

$$f(x) = \begin{cases} 3x + 1 & \text{for odd } x \\ x/2 & \text{for even } x \end{cases}$$

for any natural number x. If you start with an integer x and iterate f, you obtain a sequence, $x, f(x), f(f(x)), \ldots$ Stop if you ever hit 1. For example, if $x = 17$, you get the sequence 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1. Extensive computer tests have shown that every starting point between 1 and a large positive integer gives a sequence that ends in 1. But, the question of whether all positive starting points end up at 1 is unsolved; it is called the $3x + 1$ problem.

Suppose that A_{TM} were decidable by a TM H. Use H to describe a TM that is guaranteed to state the answer to the $3x + 1$ problem.

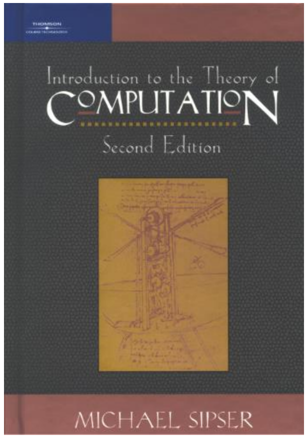
5.32 Prove that the following two languages are undecidable.

a. $OVERLAP_{\mathsf{CFG}} = \{\langle G, H \rangle \mid G \text{ and } H \text{ are CFGs where } L(G) \cap L(H) \neq \emptyset\}$. (Hint: Adapt the hint in Problem 5.21.)

b. $PREFIX\text{-}FREE_{\mathsf{CFG}} = \{G \mid G \text{ is a CFG where } L(G) \text{ is prefix-free}\}$.

5.33 Let $S = \{\langle M \rangle \mid M \text{ is a TM and } L(M) = \{\langle M \rangle\}\}$. Show that neither S nor $\overline{S}$ is Turing-recognizable.

5.34 Consider the problem of determining whether a PDA accepts some string of the form $\{ww \mid w \in \{0,1\}^*\}$. Use the computation history method to show that this problem is undecidable.



Introduction to the Theory of
COMPUTATION
Second Edition

MICHAEL SIPSER

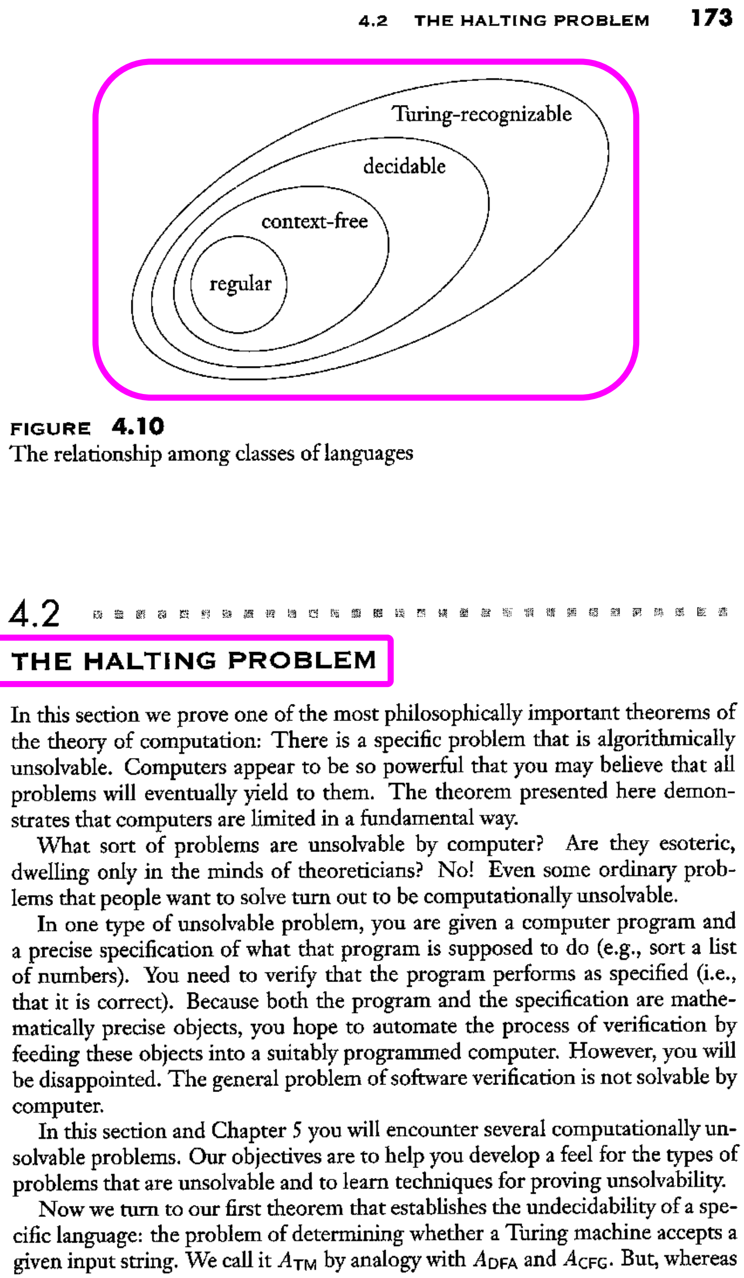


FIGURE 4.10
The relationship among classes of languages

4.2
THE HALTING PROBLEM

In this section we prove one of the most philosophically important theorems of the theory of computation: There is a specific problem that is algorithmically unsolvable. Computers appear to be so powerful that you may believe that all problems will eventually yield to them. The theorem presented here demonstrates that computers are limited in a fundamental way.

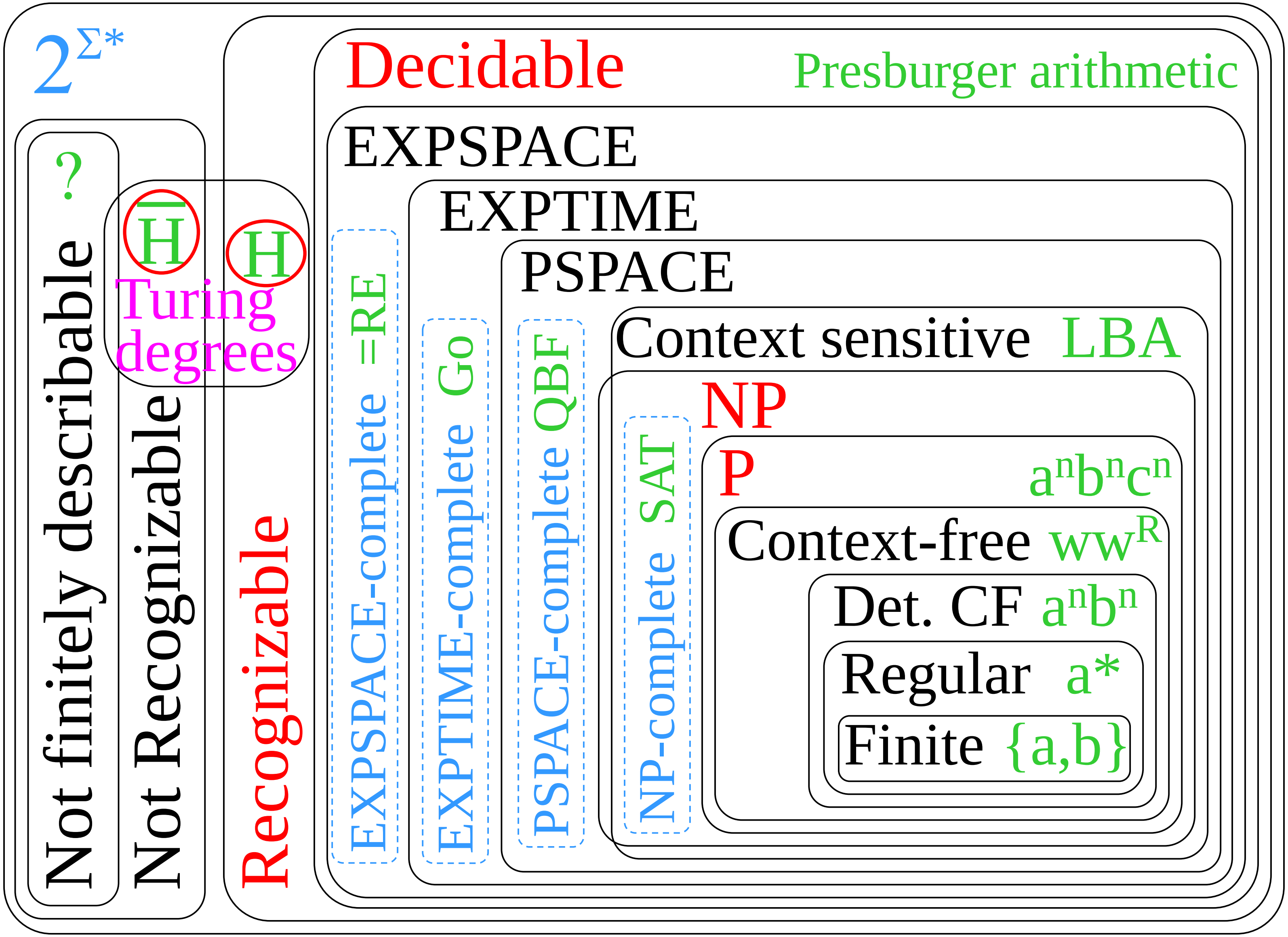
What sort of problems are unsolvable by computer? Are they esoteric, dwelling only in the minds of theoreticians? No! Even some ordinary problems that people want to solve turn out to be computationally unsolvable.

In one type of unsolvable problem, you are given a computer program and a precise specification of what that program is supposed to do (e.g., sort a list of numbers). You need to verify that the program performs as specified (i.e., that it is correct). Because both the program and the specification are mathematically precise objects, you hope to automate the process of verification by feeding these objects into a suitably programmed computer. However, you will be disappointed. The general problem of software verification is not solvable by computer.

In this section and Chapter 5 you will encounter several computationally unsolvable problems. Our objectives are to help you develop a feel for the types of problems that are unsolvable and to learn techniques for proving unsolvability.

Now we turn to our first theorem that establishes the undecidability of a specific language: the problem of determining whether a Turing machine accepts a given input string. We call it A_{TM} by analogy with A_{DFA} and A_{CFG}. But, whereas

The Extended Chomsky Hierarchy



2^{Σ^*}

Not finitely describable ?

Not Recognizable $\overline{H}$

Turing degrees

H

Recognizable

Decidable Presburger arithmetic

EXPSPACE

EXPSPACE-complete =RE

EXPTIME

EXPTIME-complete Go

PSPACE

PSPACE-complete QBF

Context sensitive LBA

NP

NP-complete SAT

P $a^n b^n c^n$

Context-free ww^R

Det. CF $a^n b^n$

Regular a^*

Finite $\{a,b\}$

Context-Sensitive Grammars

Problem: design a context-sensitive grammar to generate the (non-context-free) language $\{1^n\$1^{2^n} \mid n\geq 1\}$

Idea: generate n 1’s to the left & to the right of $; then double n times the # of 1’s on the right.

$S \rightarrow 1ND1E$ /* Base case; E marks end-of-string */

$N \rightarrow 1ND \mid \$$ /* Loop: n 1’s and n D’s; end with $ */

$D1 \rightarrow 11D$ /* Each D doubles the 1’s on right */

$DE \rightarrow E$ /* The E “cancels” out the D’s */

$E \rightarrow \varepsilon$ /* Process ends when the E vanishes */

Context-Sensitive Grammars

Example: Generating strings in $\{1^n\$1^{2^n} \mid n \geq 1\}$

$S \rightarrow 1ND1E$ $\qquad$ $D1 \rightarrow 11D$ $\qquad$ $E \rightarrow \varepsilon$

$N \rightarrow 1ND \mid \$$ $\qquad$ $DE \rightarrow E$

$S \rightarrow 1\underline{N}D1E$

$\rightarrow 11ND\underline{D1}E$

$\rightarrow 11\underline{N}D11DE$

$\rightarrow 111ND\underline{D1}1DE$

$\rightarrow 111N\underline{D1}1D1DE$

$\rightarrow 111N11D1D1\underline{DE}$

$\rightarrow 111\underline{N}11D1D1E$

$\rightarrow 111\$11\underline{D1}D1E$

$\rightarrow 111\$1111D\underline{D1}E$

$\rightarrow 111\$1111\underline{D1}1DE$

$\rightarrow 111\$111111\underline{D1}DE$

$\rightarrow 111\$11111111D\underline{DE}$

$\rightarrow 111\$11111111\underline{DE}$

$\rightarrow 111\$11111111\underline{E}$

$\rightarrow 111\$11111111\varepsilon$

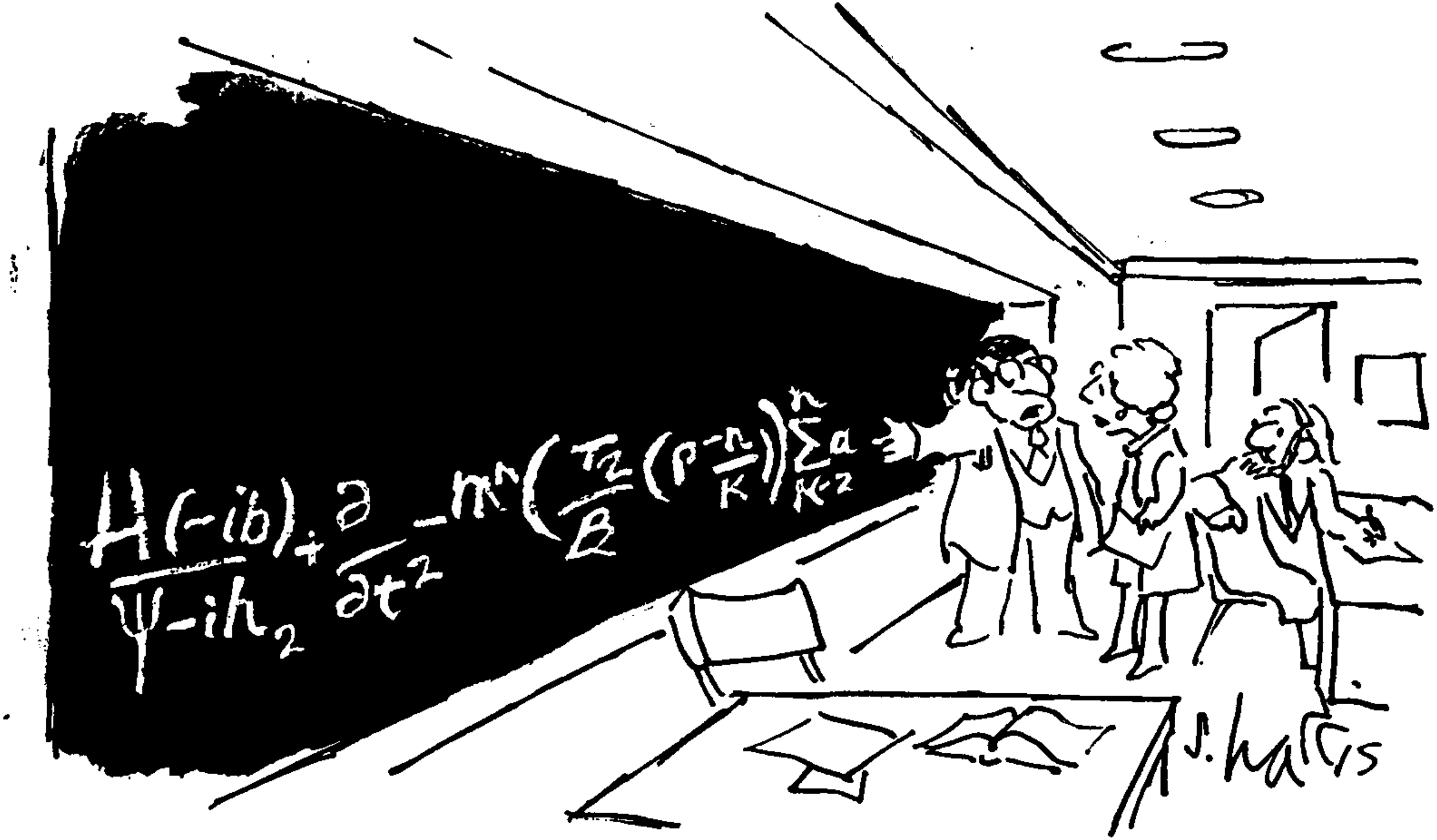
$= 1^3\$1^8 = 1^3\$1^{2^3}$

Context-Sensitive Grammars

Theorem: Context-free grammars are equivalent to arbitrary Turing machines.

Idea: a context-free grammar can “simulate” an arbitrary Turing machine / algorithm.

Details: grammar rules can implement the Turing machine’s read/write head & transition function.



"But this *is* the simplified version for the general public."