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: $\Rightarrow$ 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: $\Rightarrow$ 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: $\Rightarrow$ 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^* $\quad L \subseteq \Sigma^* \; \forall L$ — $\{\varepsilon,a,aa,aaa,\ldots\}$

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: $\emptyset$ $\emptyset^* = \{\varepsilon\}$

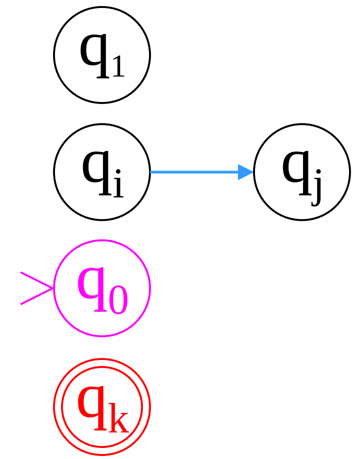
Theorem: Σ^* is countable, $|\Sigma^*| = |\mathbb{N}|$ dovetailing

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

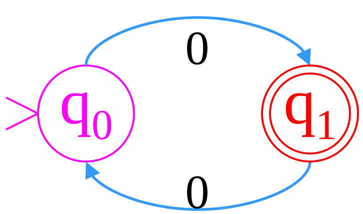
Finite Automata

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:



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

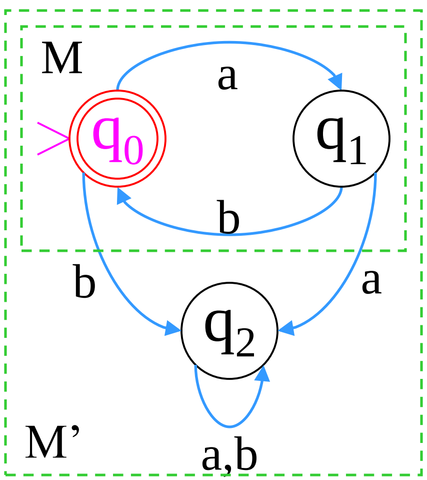
Finite Automata

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 abababab…= $(ab)^*$



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

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:

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

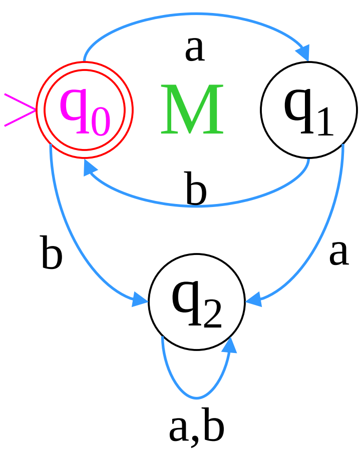
where $\delta(q_i, \varepsilon) = q_i$

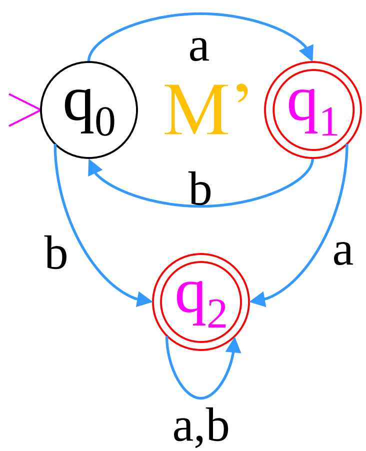
Language of M is $L(M) = \{w \in \Sigma^* \mid \delta(q_0, w) \in F\}$

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.





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

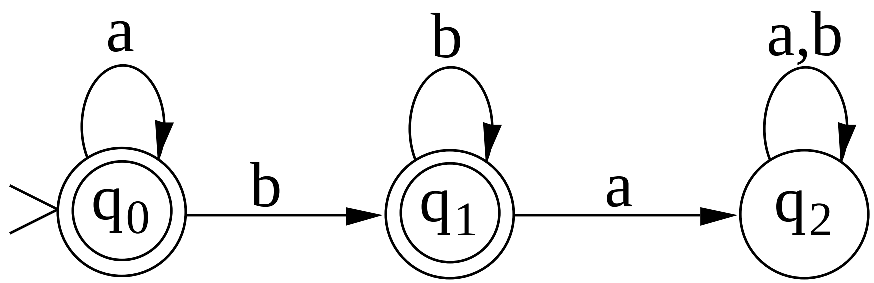
$L' = L(M') = b(a+b)^* + (a+b)^*a$
$\qquad + (a+b)^*(aa+bb)(a+b)^*$

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.



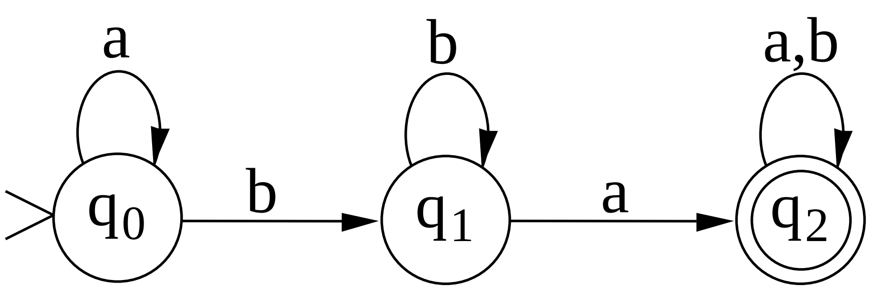
$L = a^*b^*$

Q: What is the complement of L?

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

Idea: write a regular expression and then simplify.

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



Please use this tool!

(to implement some nontrivial FAs, TMs, PDAs, grammars, etc.)

http://www.jflap.org/

JFLAP - Mozilla Firefox

http://www.jflap.org/

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

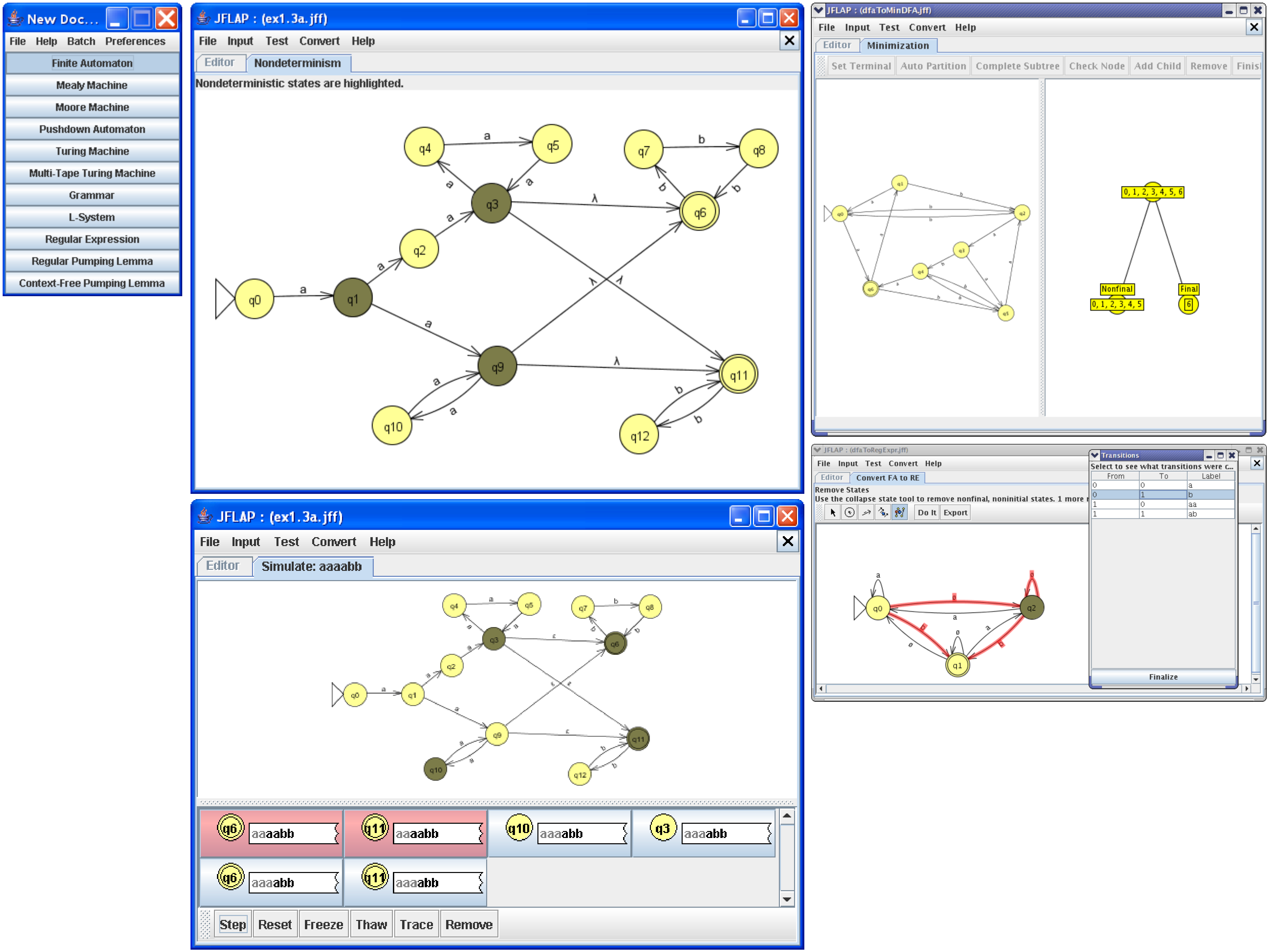
JFLAP

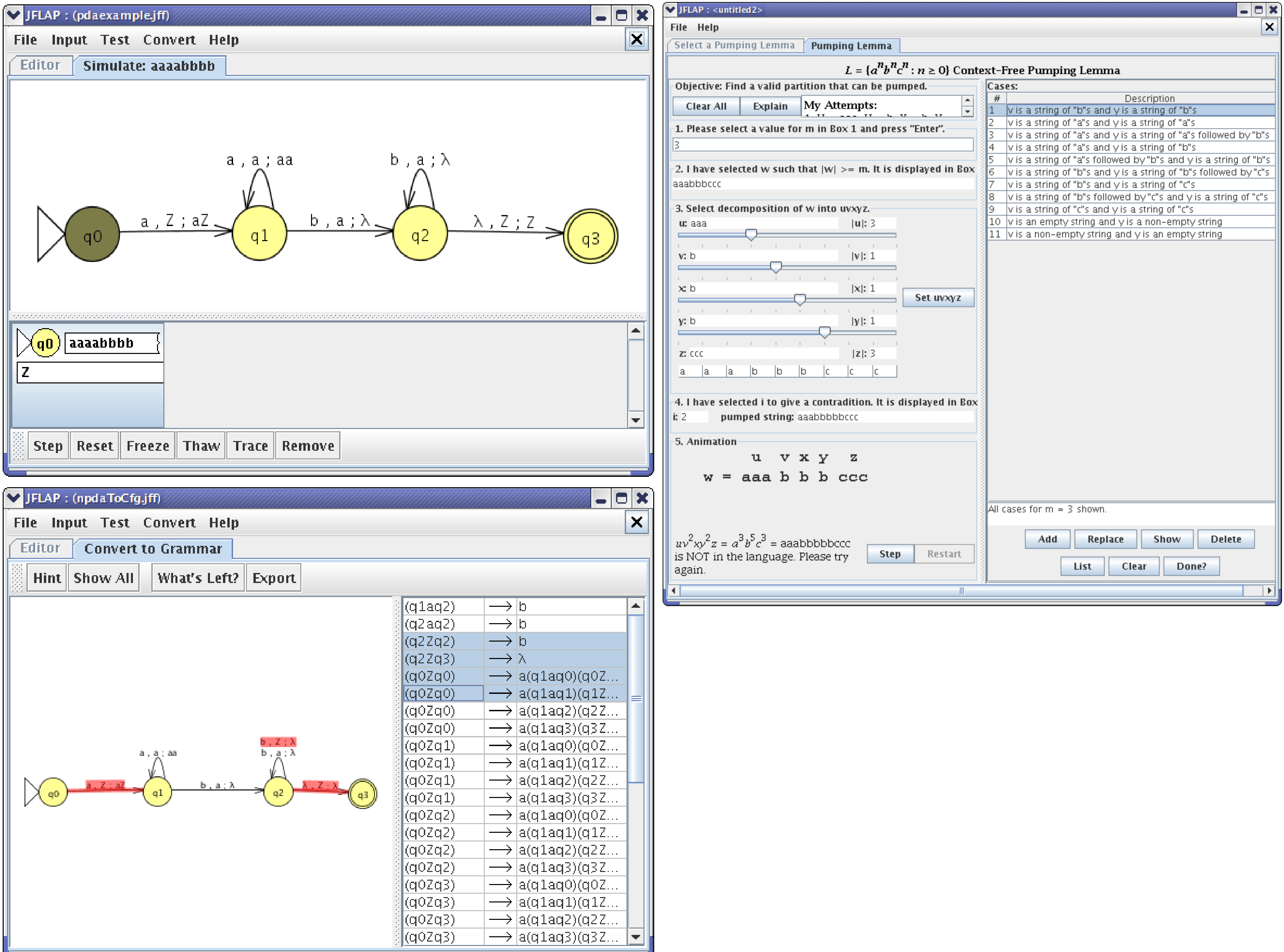
JFLAP is software for experimenting with formal languages topics including nondeterministic finite automata, nondeterministic pushdown automata, multi-tape Turing machines, several types of grammars, parsing, and L-systems. In addition to constructing and testing examples for these, JFLAP allows one 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.

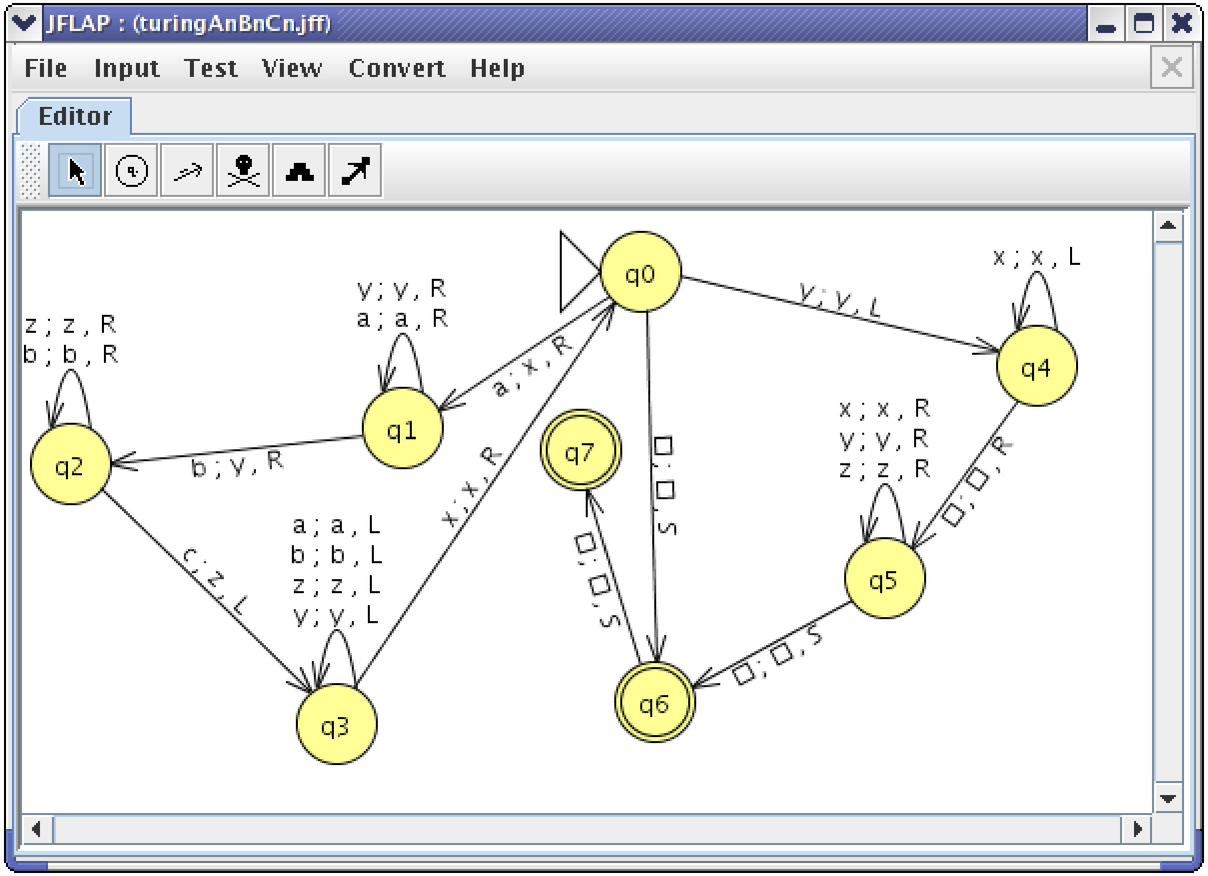
JFLAP News

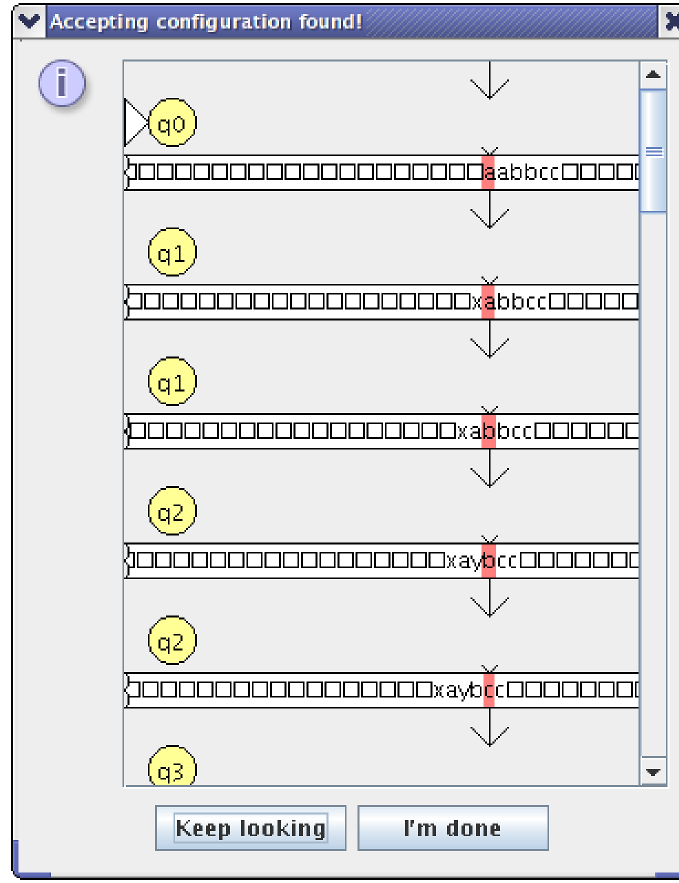
- *** September 28, 2009 ***
  Note to Mac users. Macs default to Java 1.5 and JFLAP now requires Java 1.6.
- *** August 28, 2009 *** - JFLAP 7.0 released. See below for more details on changes. Get JFLAP
- December 2008 - JFLAP CD by Linz and Rodger with JFLAP exercises that goes along with the Linz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

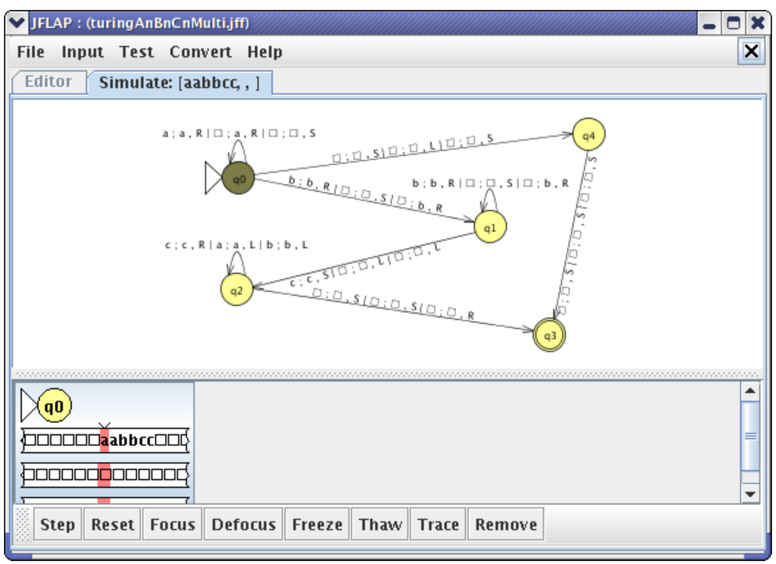
  Note, we have worked closedly with Linz over the past several years so JFLAP fits nicely with the Linz book. Eventually we hope to publish a textbook with JFLAP integrated in...
- July 2008 - JFLAP now has a wiki where users can discuss the use or modifications of JFLAP, see jflap.wikia.com
- July 2008 - 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

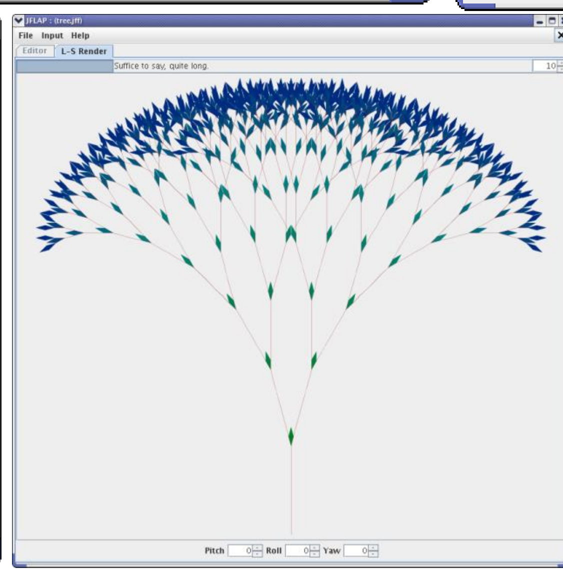

New Doc...
File Help Batch Preferences
Finite Automaton
Mealy Machine
Moore Machine
Pushdown Automaton
Turing Machine
Multi-Tape Turing Machine
Grammar
L-System
Regular Expression
Regular Pumping Lemma
Context-Free Pumping Lemma
JFLAP : (ex1.3a.jff)
File Input Test Convert Help
Editor Nondeterminism
Nondeterministic states are highlighted.
q0 q1 q2 q3 q4 q5 q6 q7 q8 q9 q10 q11 q12
a b λ
JFLAP : (dfaToMinDFA.jff)
File Input Test Convert Help
Editor Minimization
Set Terminal Auto Partition Complete Subtree Check Node Add Child Remove
0, 1, 2, 3, 4, 5, 6
Nonfinal
0, 1, 2, 3, 4, 5
Final
6
JFLAP : (ex1.3a.jff)
File Input Test Convert Help
Editor Simulate: aaaabb
aaaabb
Step Reset Freeze Thaw Trace Remove
JFLAP : (dfaToRegExpr.jff)
File Input Test Convert Help
Editor Convert FA to RE
Remove States
Do It Export
Transitions
From To Label
0 0 a
0 1 b
1 0 aa
1 1 ab
Finalize

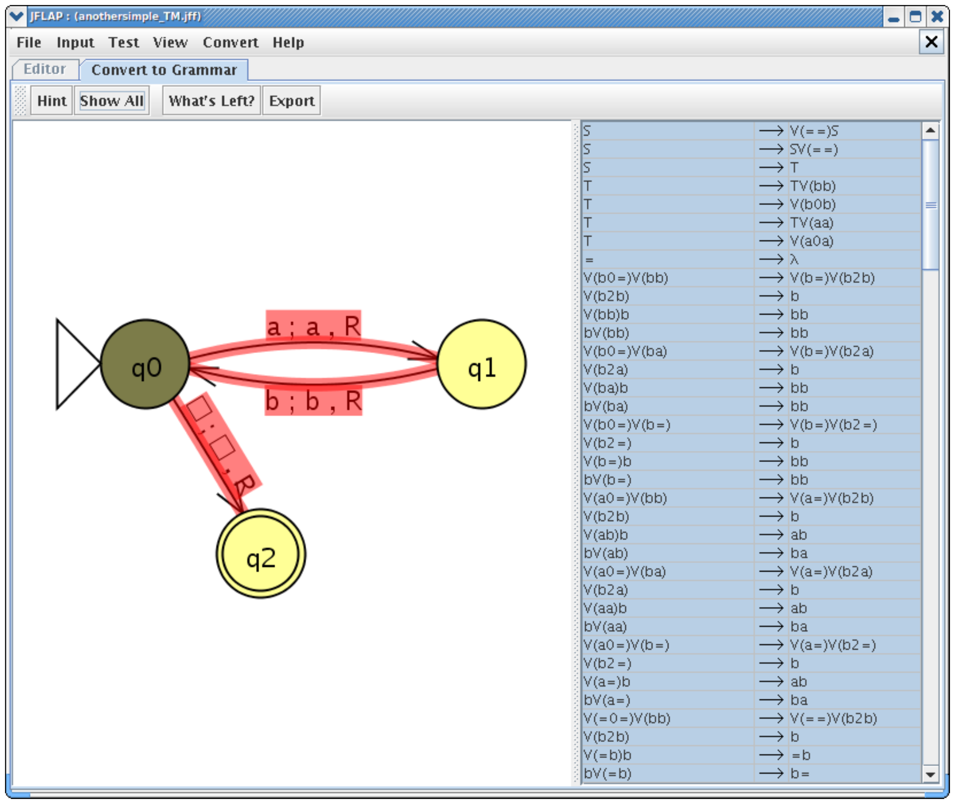

JFLAP : (pdaexample.jff)
File Input Test Convert Help
Editor
Simulate: aaaabbbb
a , a ; aa
b , a ; λ
a , Z ; aZ
b , a ; λ
λ , Z ; Z
q0
q1
q2
q3
aaaabbbb
Z
Step Reset Freeze Thaw Trace Remove
JFLAP : (npdaToCfg.jff)
File Input Test Convert Help
Editor
Convert to Grammar
Hint Show All What's Left? Export
(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...
JFLAP : <untitled2>
File Help
Select a Pumping Lemma
Pumping Lemma
L = {a^n b^n c^n : n ≥ 0} Context-Free Pumping Lemma
Objective: Find a valid partition that can be pumped.
Clear All Explain 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
Set uvxyz
4. I have selected i to give a contradiction. 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.
Step Restart
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.
Add Replace Show Delete
List Clear Done?


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
y ; y , L
x ; x , L
c ; z , L
a ; a , L
b ; b , L
z ; z , L
y ; y , L
x ; x , R
x ; x , R
y ; y , R
z ; z , R
□ ; □ , S
□ ; □ , R
□ ; □ , S
□ ; □ , S

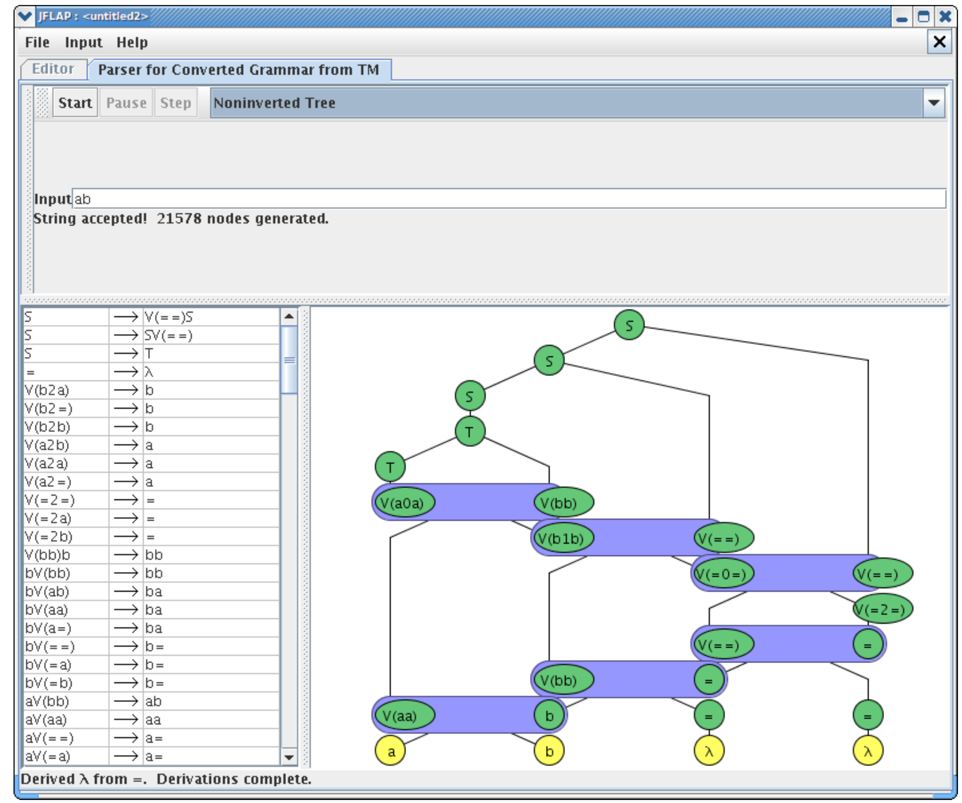

Accepting configuration found!
q0
q1
q1
q2
q2
q3
Keep looking
I'm done


JFLAP : (turingAnBnCnMulti.jff)
File Input Test Convert Help
Editor
Simulate: [aabbcc, ,]
q0
q1
q2
q3
q4
Step Reset Focus Defocus Freeze Thaw Trace Remove

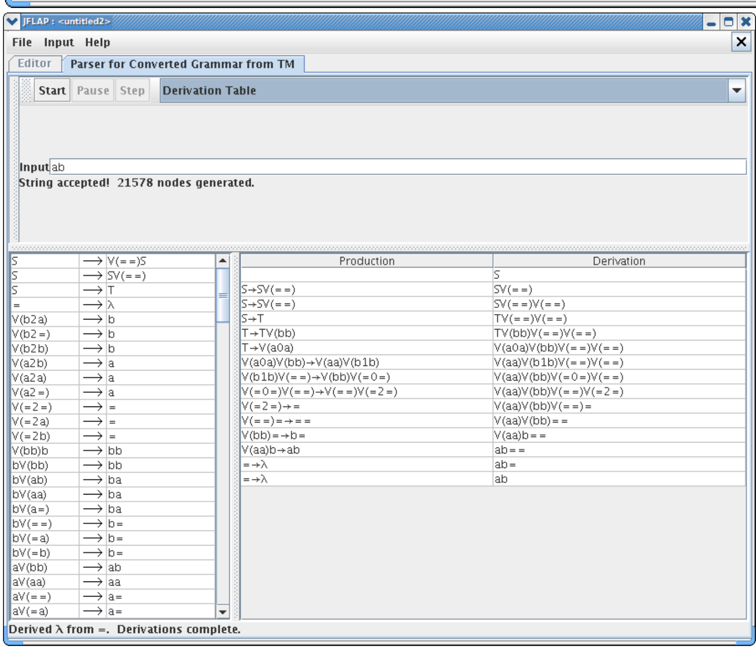

File Input Help
Editor
L-S Render
Suffice to say, quite long.
Pitch
Roll
Yaw



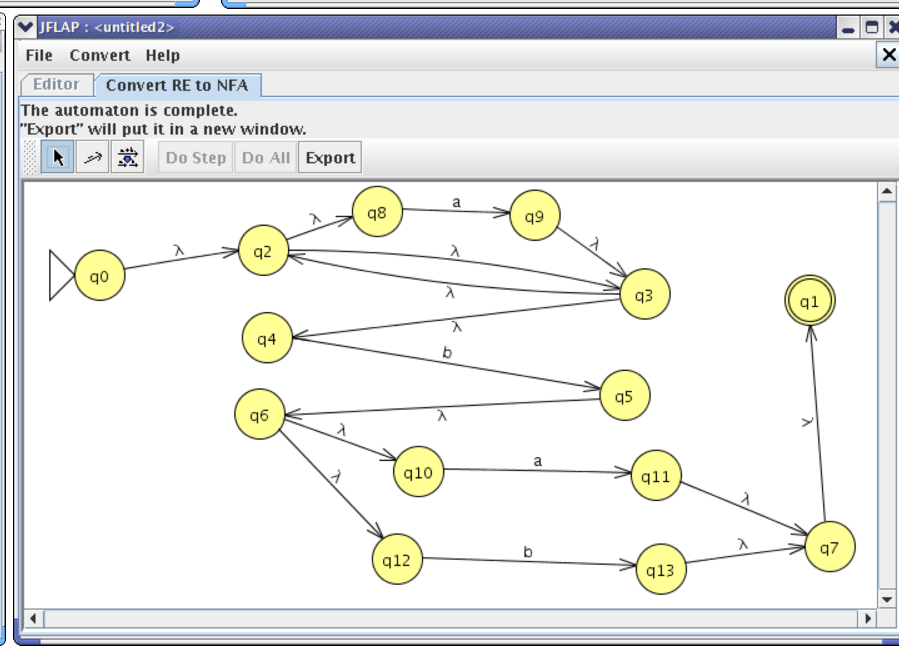
JFLAP : (anothersimple_TM.jff)
File Input Test View Convert Help
Editor Convert to Grammar
Hint Show All What's Left? Export
a ; a , R
b ; b , R
q0
q1
q2
S ⟶ V(==)S
S ⟶ SV(==)
S ⟶ T
T ⟶ TV(bb)
T ⟶ V(b0b)
T ⟶ TV(aa)
T ⟶ V(a0a)
= ⟶ λ
V(b0=)V(bb) ⟶ V(b=)V(b2b)
V(b2b) ⟶ b
V(bb)b ⟶ bb
bV(bb) ⟶ bb
V(b0=)V(ba) ⟶ 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=



JFLAP : <untitled2>
File Input Help
Editor Parser for Converted Grammar from TM
Start Pause Step Noninverted Tree
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=
S
S
S
T
T
V(a0a)
V(bb)
V(b1b)
V(==)
V(=0=)
V(==)
V(=2=)
V(==)
=
V(bb)
=
V(aa)
b
=
=
a
b
λ
λ
Derived λ from =. Derivations complete.



JFLAP : <untitled2>
File Input Help
Editor Parser for Converted Grammar from TM
Start Pause Step Derivation Table
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.



JFLAP : <untitled2>
File Convert Help
Editor Convert RE to NFA
The automaton is complete.
"Export" will put it in a new window.
Do Step Do All Export
q0
q1
q2
q3
q4
q5
q6
q7
q8
q9
q10
q11
q12
q13
a
b
a
b
λ

0

INTRODUCTION

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

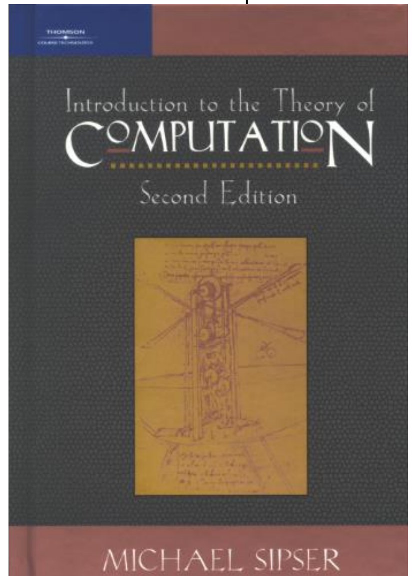
AUTOMATA, COMPUTABILITY, AND COMPLEXITY

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.



1

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

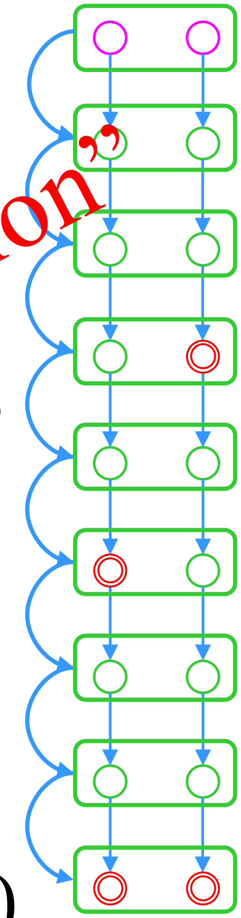
Finite Automata

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 same language as original NFA (but size can be the product of two old sizes).

"cross product construction"



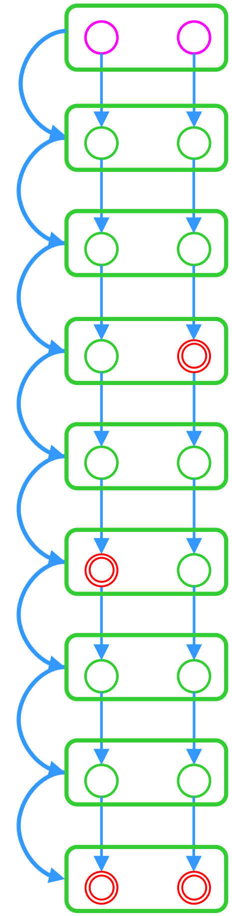
Given $M_1=(Q_1, \Sigma, \delta_1, q', F_1)$ and $M_2=(Q_2, \Sigma, \delta_2, q'', F_2)$

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

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

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

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!

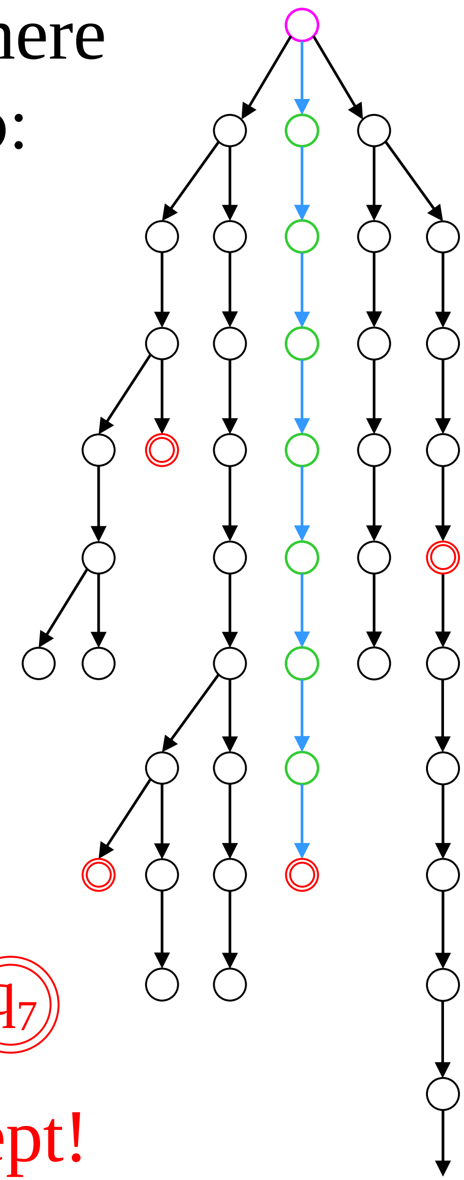
Finite Automata

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

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

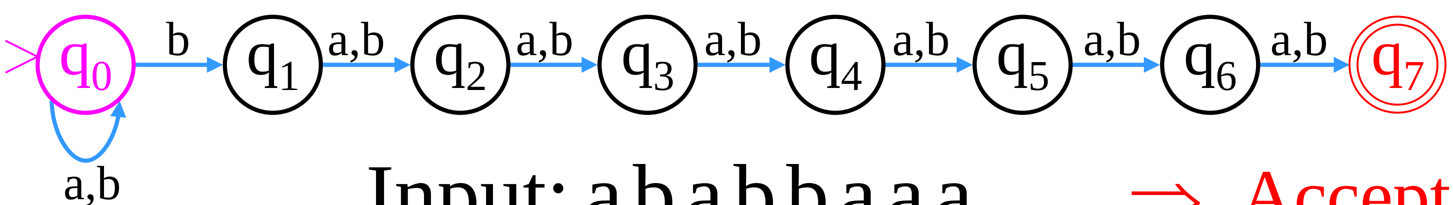
New $\delta: 2^Q \times \Sigma \rightarrow 2^Q$

Computation becomes a "tree".



Acceptance: $\exists$ 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":



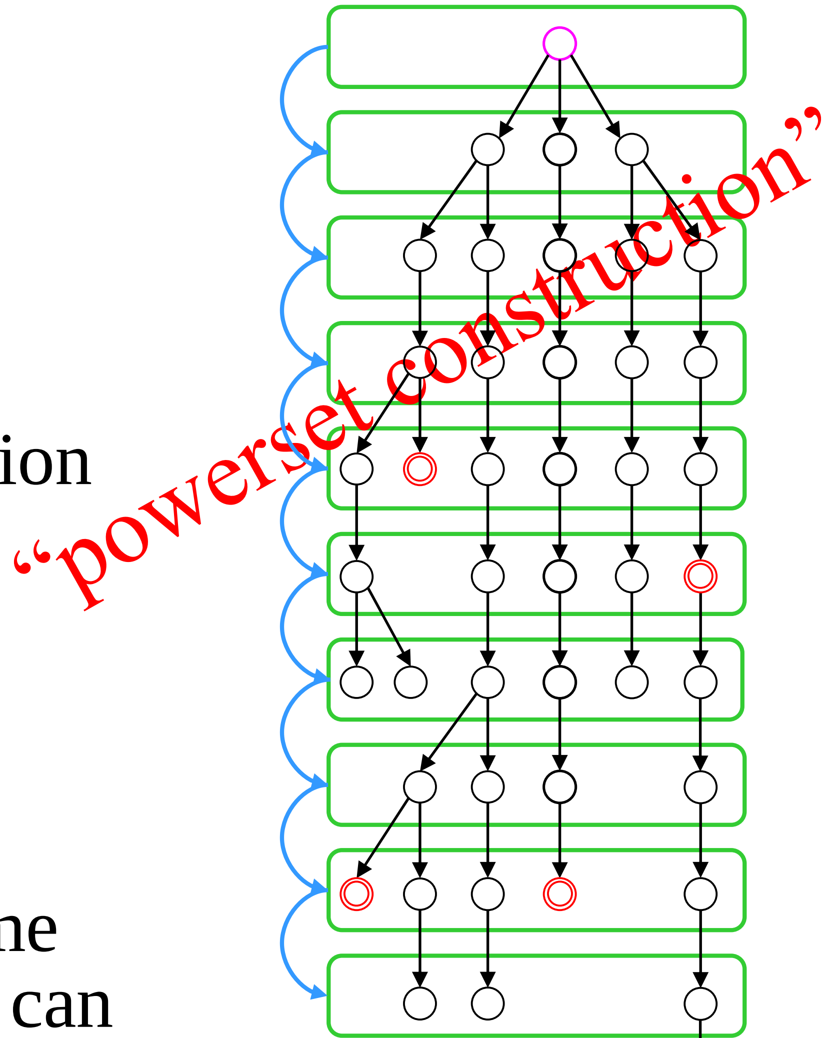
Input: a b a b b a a a $\Rightarrow$ Accept!

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.

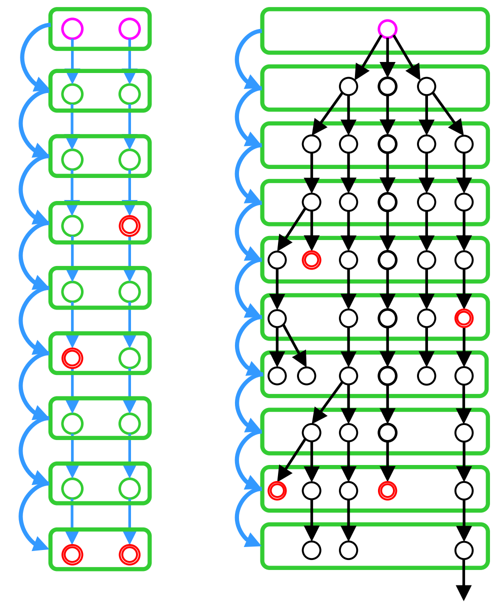


Q: Why doesn't this work for PDAs or TMs?

Finite Automata

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

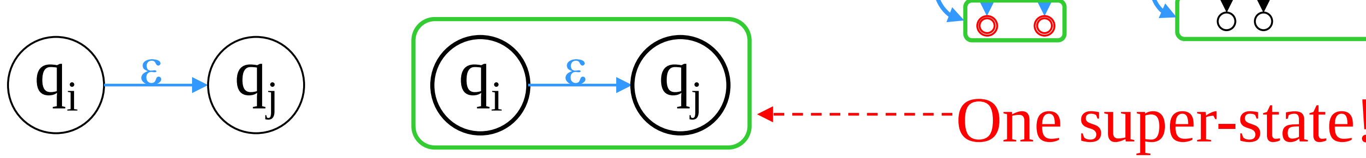
EC: Let HALF(L)=$\{v \mid \exists\ v,w \in \Sigma^* \ni |v|=|w| \text{ and } vw \in L\}$
Show that HALF preserves regularity.



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.

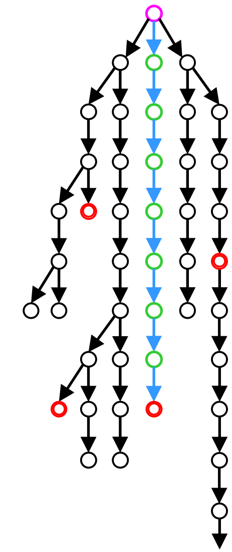
ε-transitions:



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.

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!

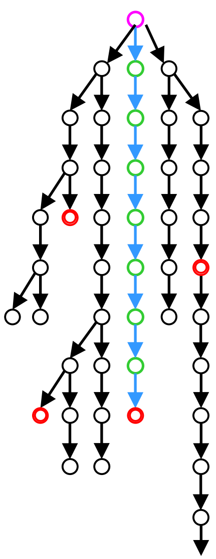
Transition function!

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!



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 $A_1 \circ A_2$.

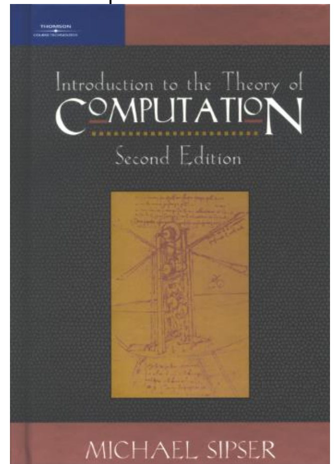
To prove this theorem let's try something along the lines of the proof of the union case. As before, we can start with finite automata M_1 and M_2 recognizing the regular languages A_1 and A_2. But now, instead of constructing automaton M to accept its input if either M_1 or M_2 accept, it must accept 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.



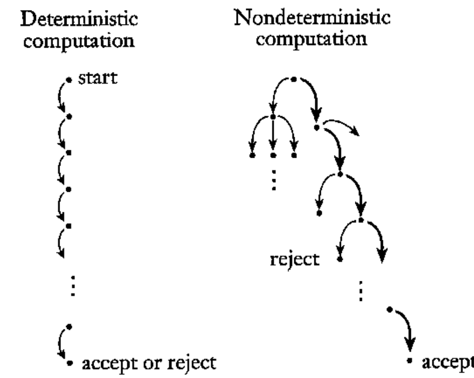


FIGURE 1.28
Deterministic and nondeterministic computations with an accepting branch

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.

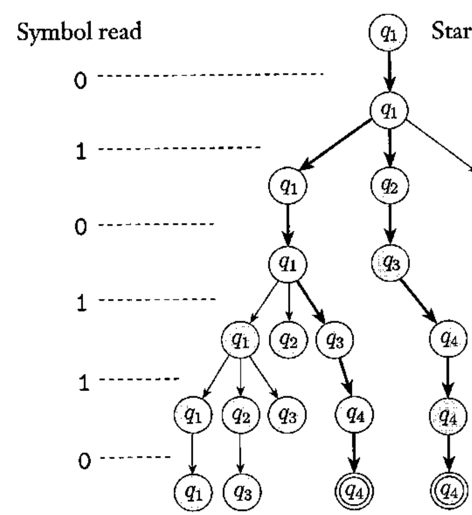
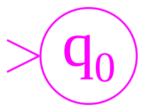


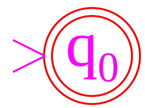
FIGURE 1.29
The computation of N_1 on input `010110`

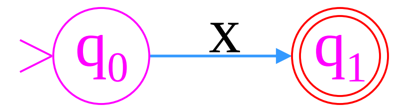
Regular Expressions

Regular expressions are defined recursively as follows:

$\emptyset$	$\{\varepsilon\}$	$\{x\}\ \forall\ x \in \Sigma$
empty set	trivial language	singleton language

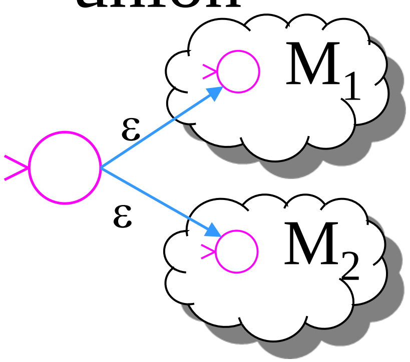


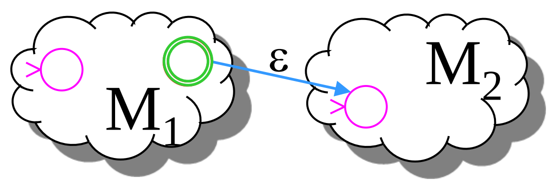


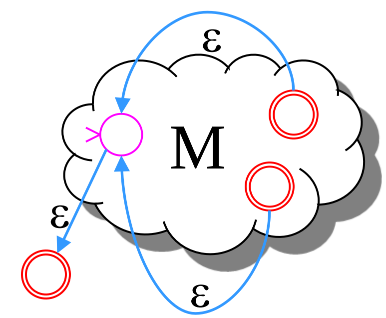


Inductively, if R and S are regular expressions, then so are:

$(R+S)$	RS	R^*
union	concatenation	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.

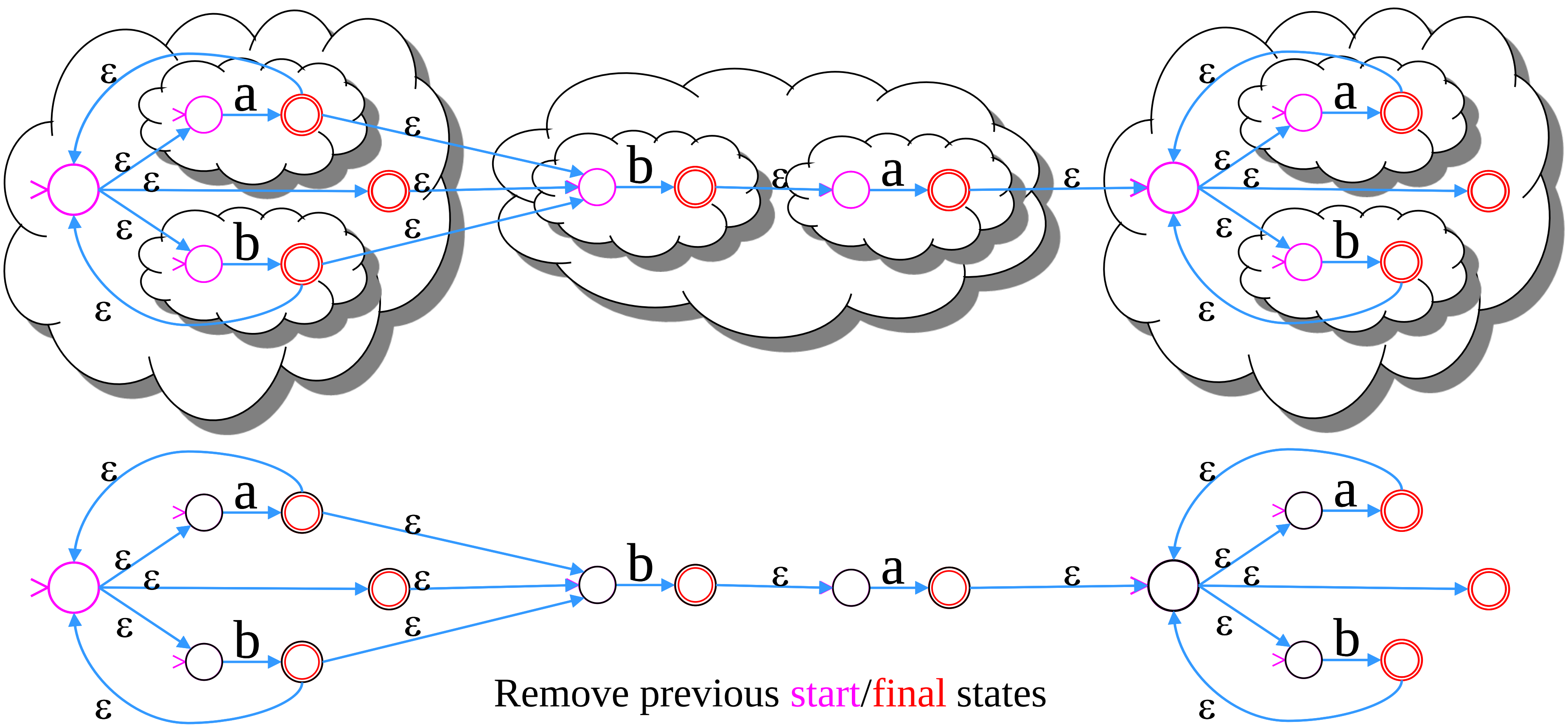
Regular Expressions

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

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

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

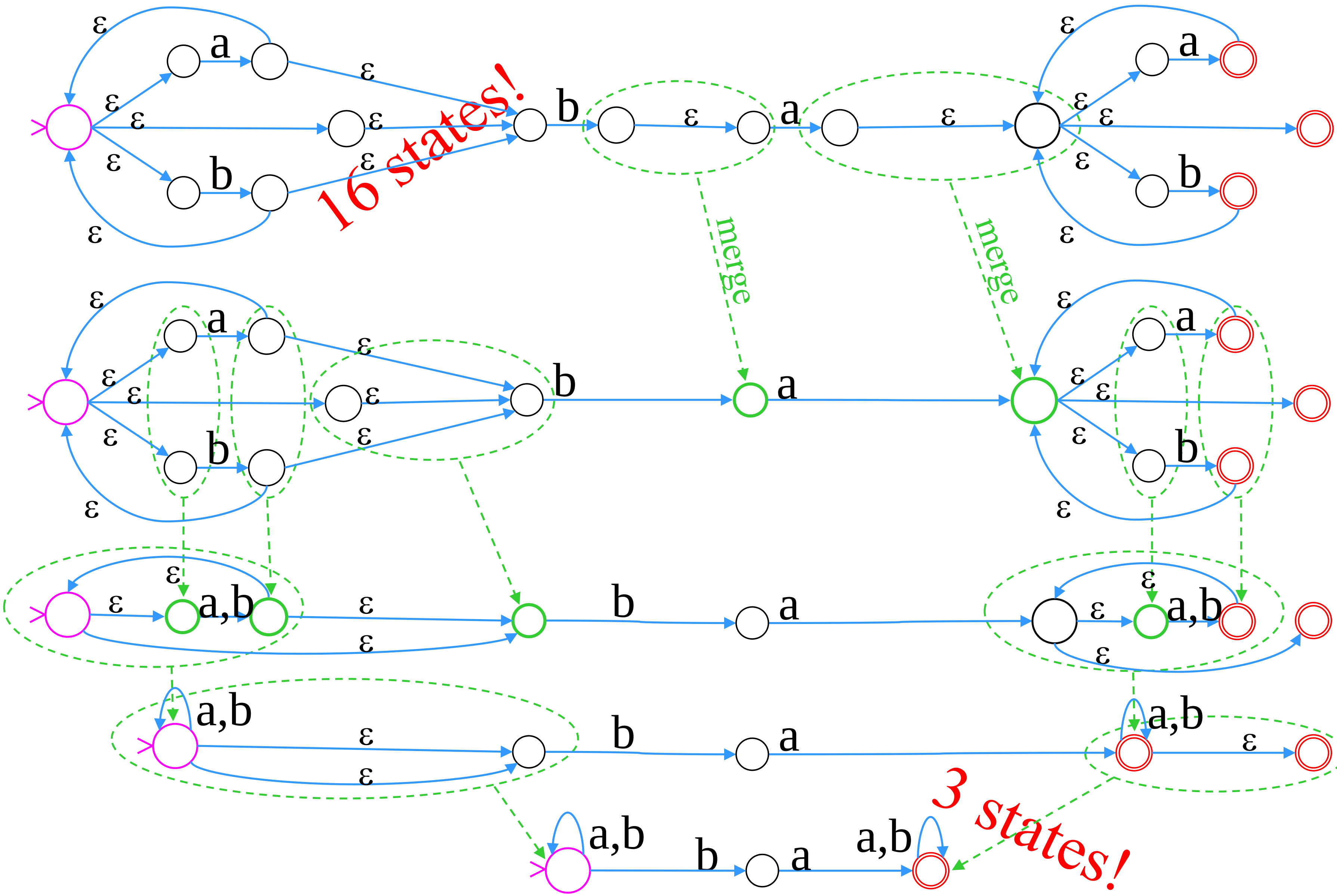
Why? $= (a+b)^* ba (a+b)^*$



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

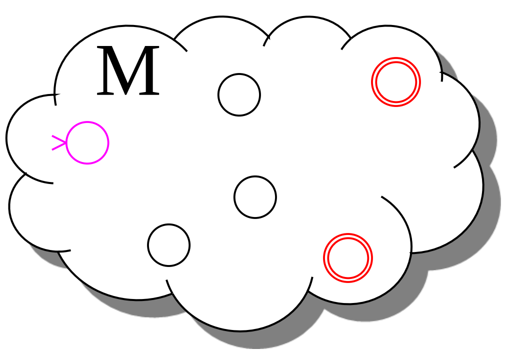
FAs and Regular Expressions

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

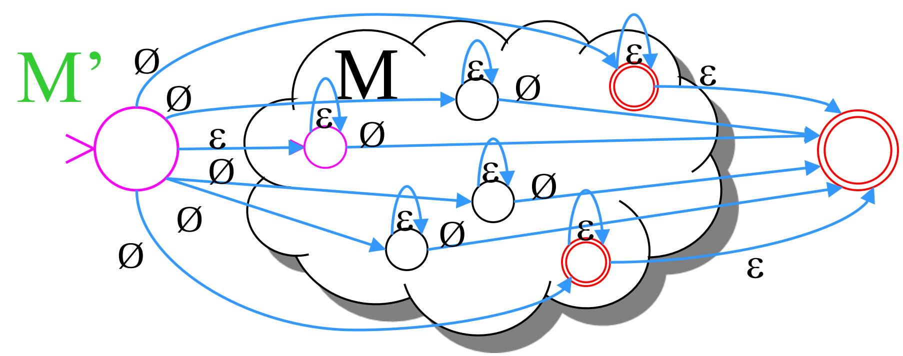
Proof: Use "generalized finite automata" where a transition can be a regular expression (not just a symbol), and:

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.



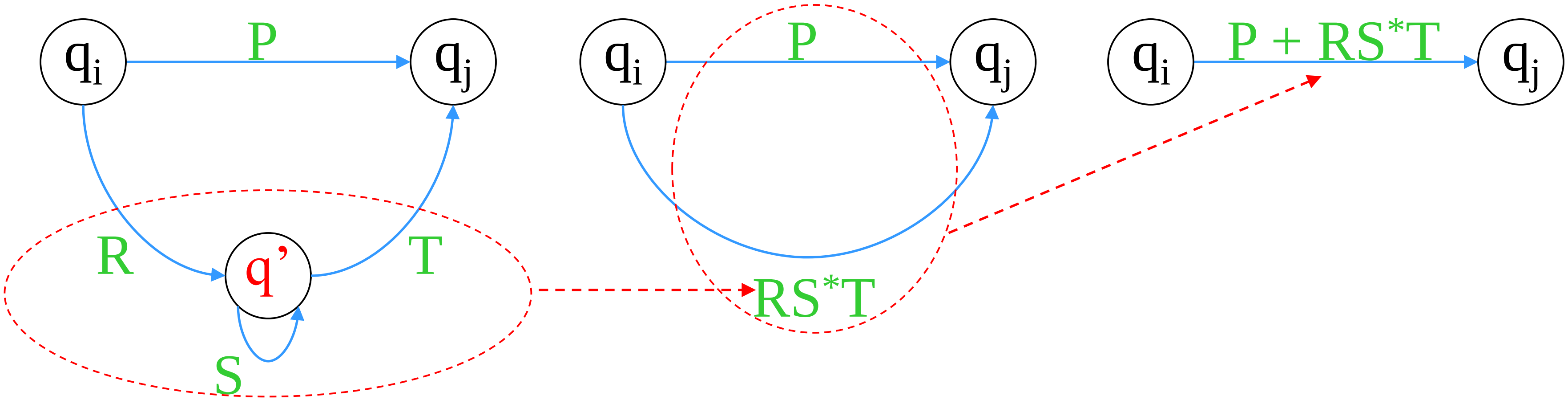
Original FA M



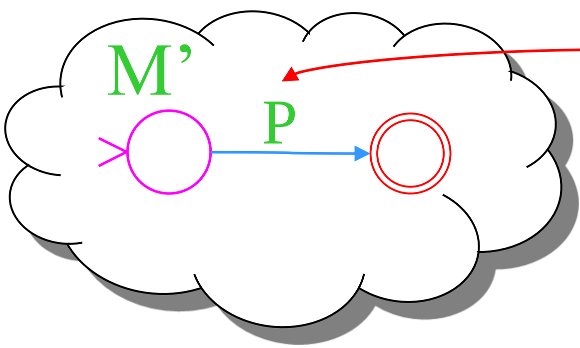
Generalized FA (GFA) M'

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.

Regular Expressions Identities

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

$R+\varepsilon \neq R$

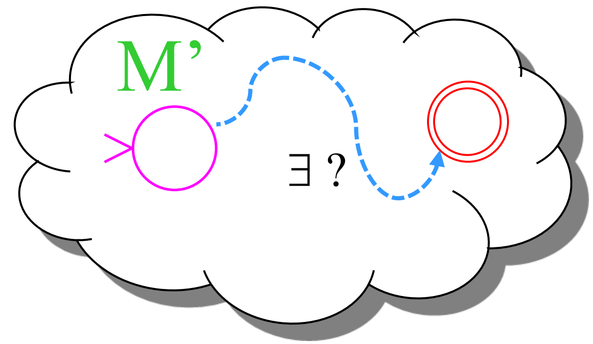
$R\emptyset \neq R$

Decidable Finite Automata Problems

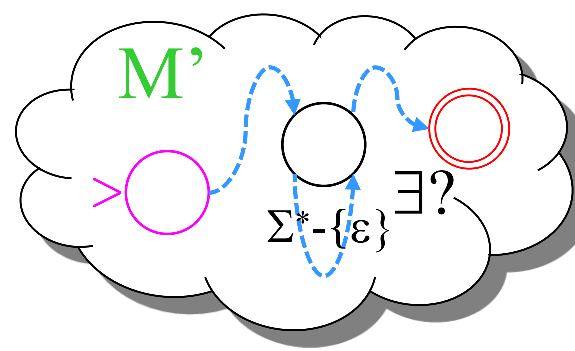
Def: A problem is decidable if ∃ an algorithm which can determine (in finite time) the correct answer for any instance.

Given a finite automata M_1 and M_2:

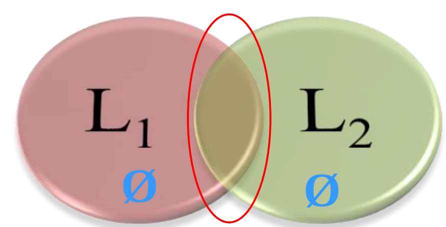
Q_1: Is $L(M_1) = Ø$?
Hint: graph reachability



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?)

Context-Free Grammars

Basic idea: set of production rules induces a language

- Finite set of variables: $V = \{V_1, V_2, ..., V_k\}$
- Finite set of terminals: $T = \{t_1, t_2, ..., t_j\}$
- 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

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

Theorem: All regular languages are context-free.

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.

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

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

Theorem: Some languages are not context-free .

Ex: $\{0^n1^n2^n \mid n > 0\}$

Proof by "pumping" argument for CFL's.

Ambiguity

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

Theorem: Some context-free grammars are ambiguous.

Fault of the grammar

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

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

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

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

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

Fault of the language!

Ex: $\{a^n b^n c^m d^m \mid m>0, n>0\} \cup \{a^n b^m c^n d^m \mid m>0, n>0\}$

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

Idea: create rules for generating nesting & juxtaposition.

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

Ex: $S \rightarrow SS \rightarrow (S)(S) \rightarrow ()()$

$S \rightarrow (S) \rightarrow ((S)) \rightarrow (())$

$S \rightarrow (S) \rightarrow (SS) \rightarrow \ldots \rightarrow (()((())()))$

Q: Is G_1 ambiguous?

Another grammar:

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

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

Q: Is G_2 ambiguous?

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

Idea: embedd the RE rules in a grammar.

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

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

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

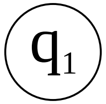
$S \rightarrow SS \rightarrow SSSS \rightarrow abS^*b \rightarrow aba^*a$

Q: Is G ambiguous?

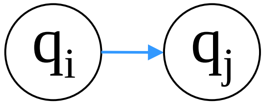
Pushdown Automata

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

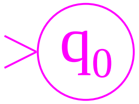
- Finite set of states: $Q = \{q_0, q_1, q_3, \ldots, q_k\}$



- 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: $F \subseteq Q$

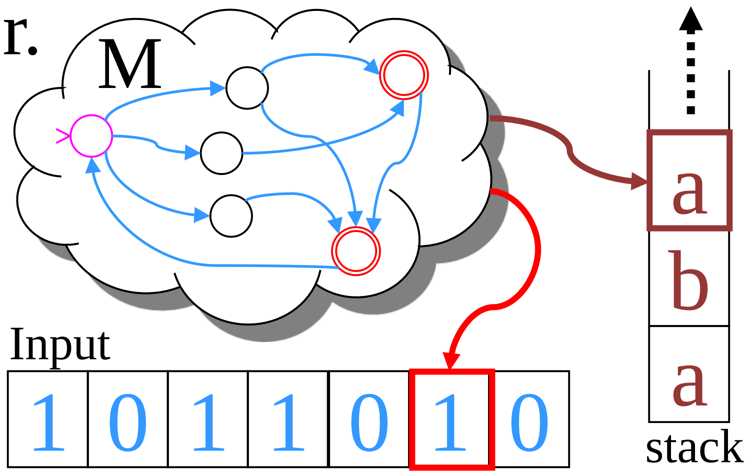
Pushdown automaton is $M=(Q, \Sigma, \Gamma, \delta, q_0, F)$

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:

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

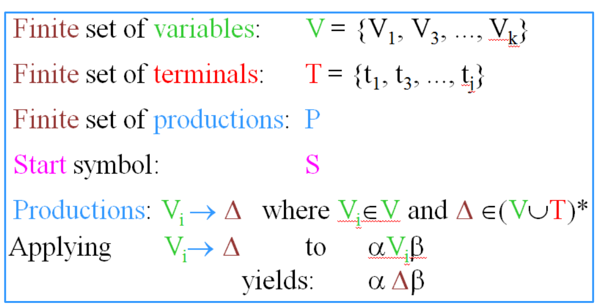
Pushdown Automata

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: $T = \{t_1, t_3, \ldots, t_j\}$

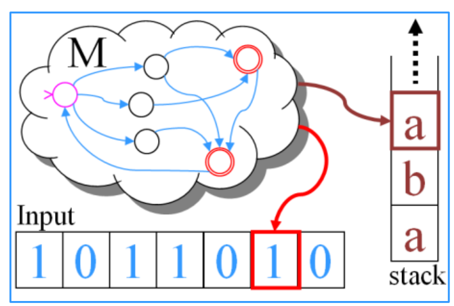
Finite set of productions: P

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):

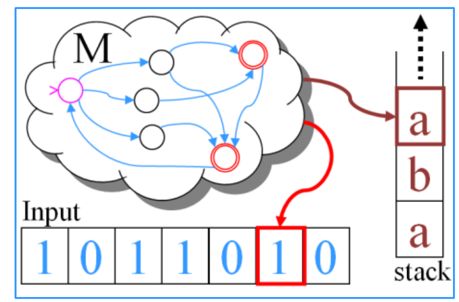
Q_1: Is $L(M) = \emptyset$?
Q_5: Is $L(G) = \emptyset$?

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

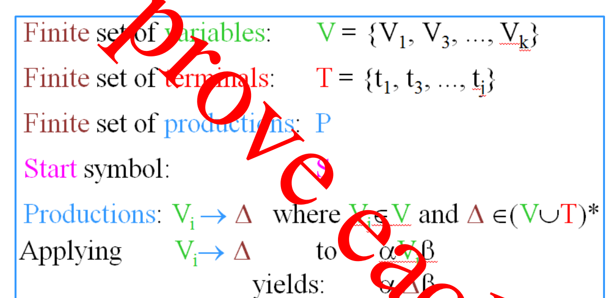
Q_3: Is $L(M)$ infinite ?
Q_7: Is $L(G)$ infinite ?

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

Extra-credit: prove each!



$\equiv$



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 ?

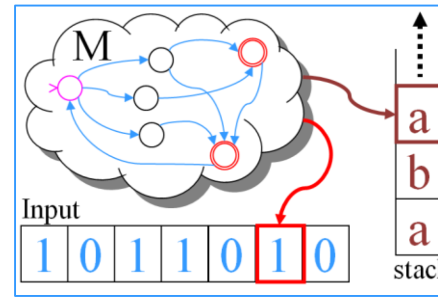
Q: Is CFG G ambiguous ?

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.

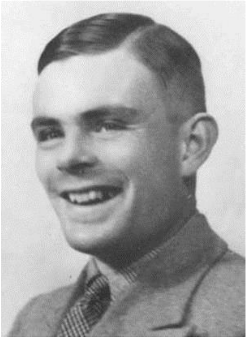
Hint: Find an example non-CFL accepted by a 2-head PDA.

Theorem: Non-determinism increases the power of PDAs.

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

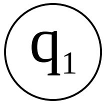
Extra-credit: prove each!

Turing Machines

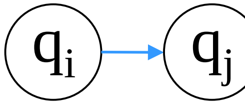


Basic idea: a Turing machine is a finite automaton that can optionally write to an unbounded tape.

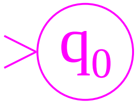
- Finite set of states: $Q = \{q_0, q_1, q_3, \ldots, q_k\}$



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



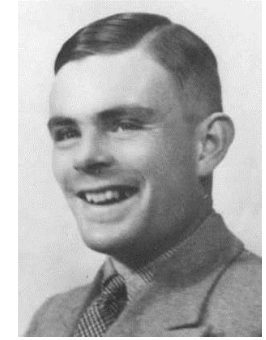
- Initial state: $q_0 \in Q$



- Final states: $F \subseteq Q$

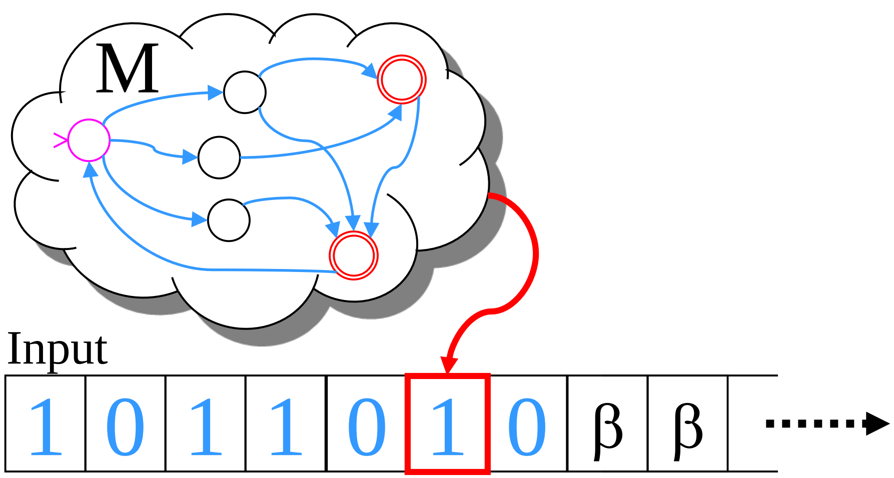
Turing machine is $M = (Q, \Gamma, \beta, \Sigma, \delta, q_0, F)$

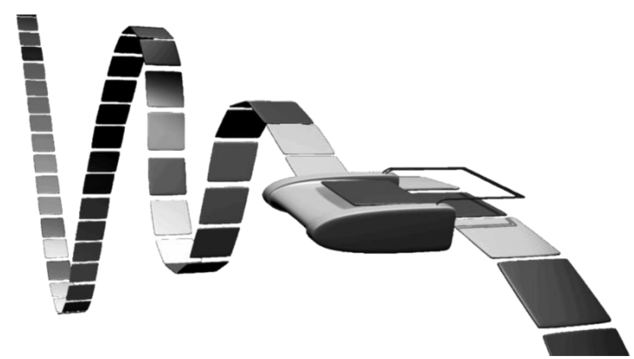
Turing Machines



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





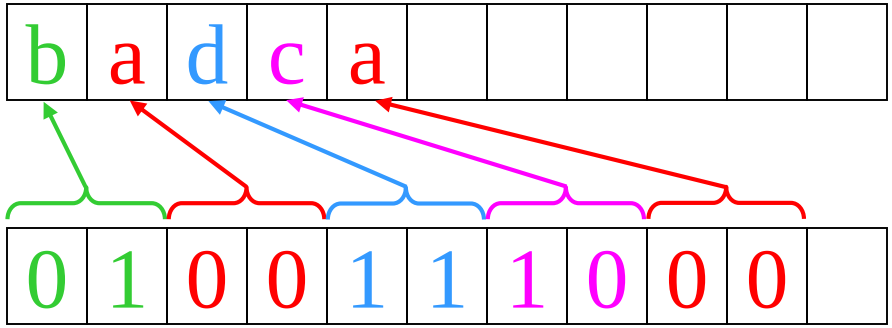
Turing Machine "Enhancements"

Larger alphabet:

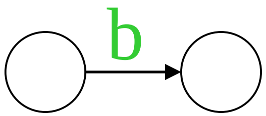
old: $\Sigma=\{0,1\}$ new: $\Sigma' = \{a,b,c,d\}$

Idea: Encode larger alphabet using smaller one.

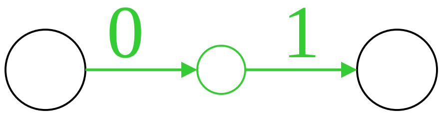
Encoding example: a=00, b=01, c=10, d=11



old: δ

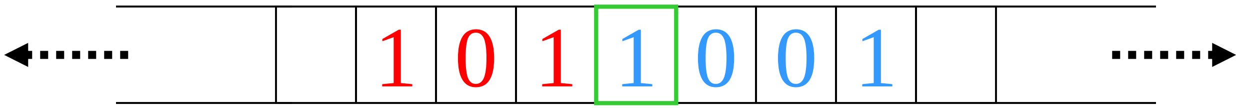


new: δ'

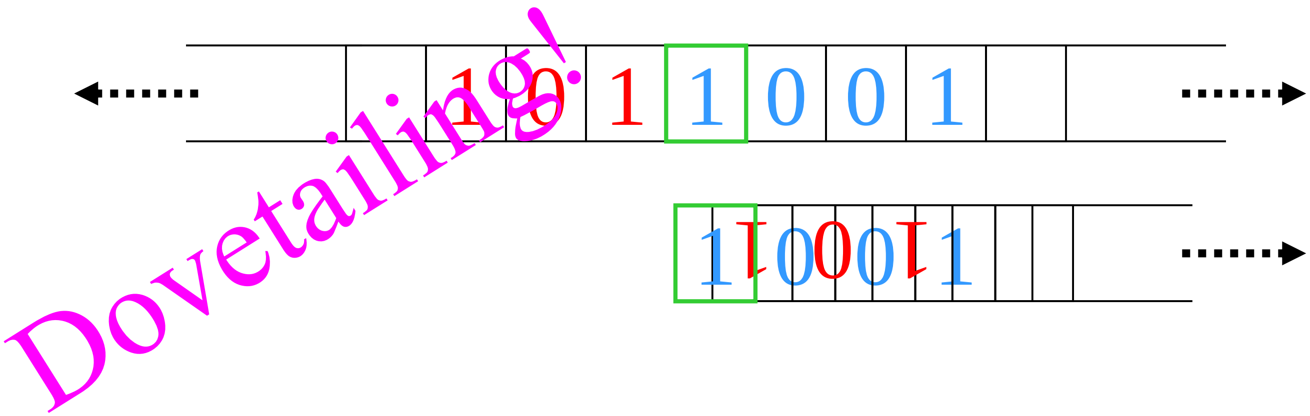


Turing Machine “Enhancements”

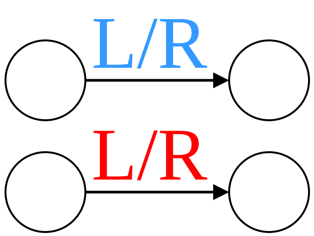
Double-sided infinite tape:



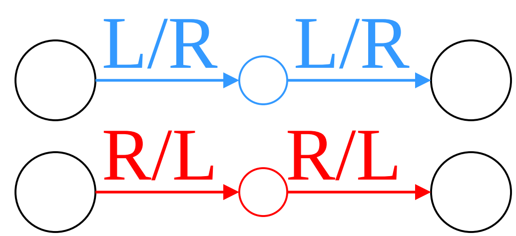
Idea: Fold into a normal single-sided infinite tape



old: δ

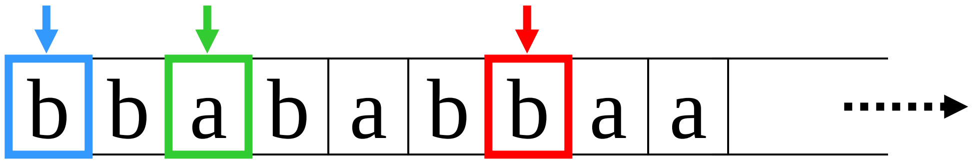


new: δ'

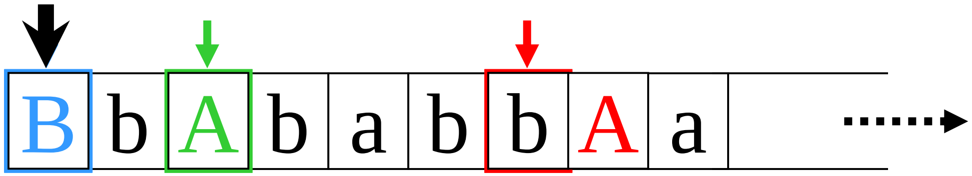


Turing Machine “Enhancements”

Multiple heads:



Idea: Mark heads locations on tape and simulate

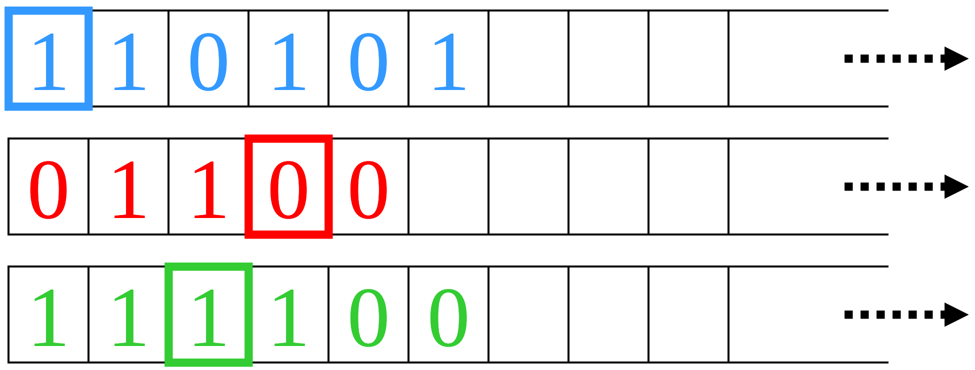


Modified δ' processes each “virtual” head independently:

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

Turing Machine "Enhancements"

Multiple tapes:



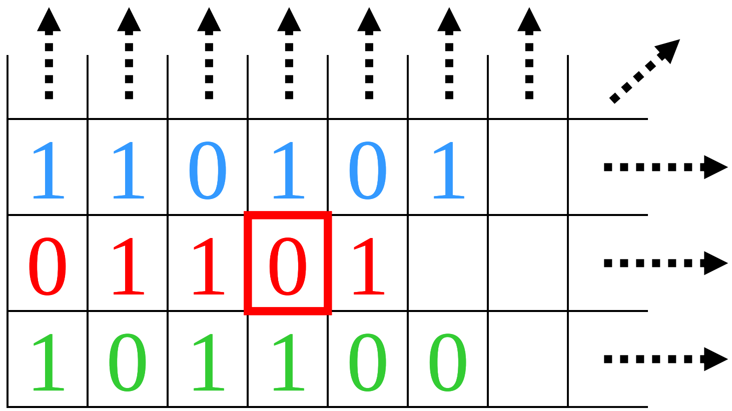
Idea: Interlace multiple tapes into a single tape

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:



This is how compilers implement 2D arrays!

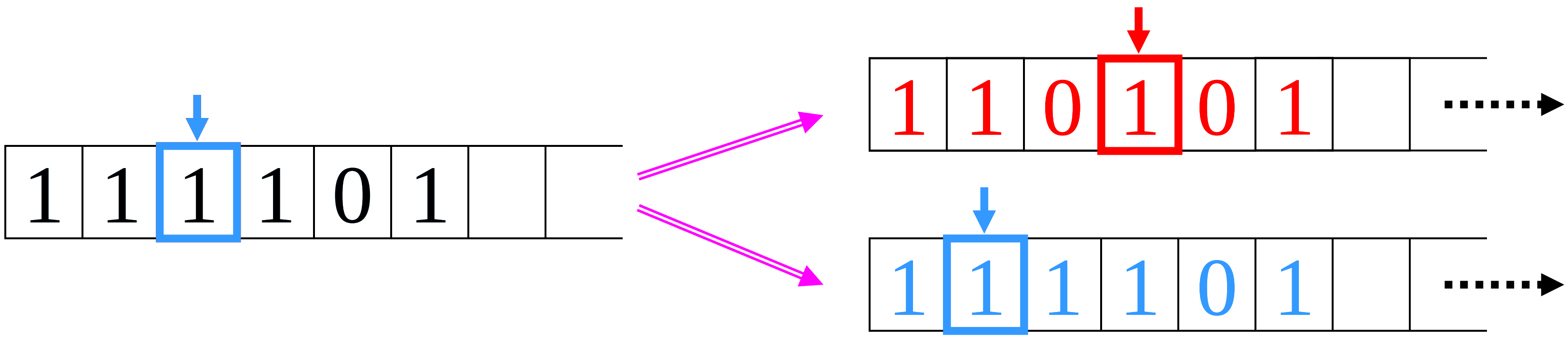
Idea: Flatten 2-D tape into a 1-D tape

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

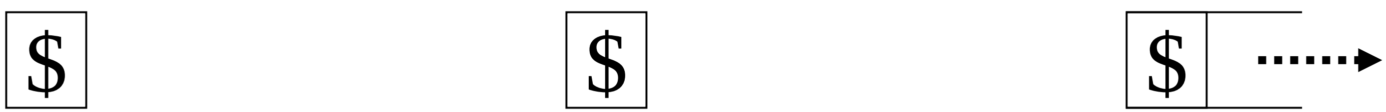
- Left/right δ moves: δ' moves horizontally
- Up/down δ moves: δ' jumps between tape sections

Turing Machine "Enhancements"

Non-determinism:



Idea: Parallel-simulate non-deterministic threads

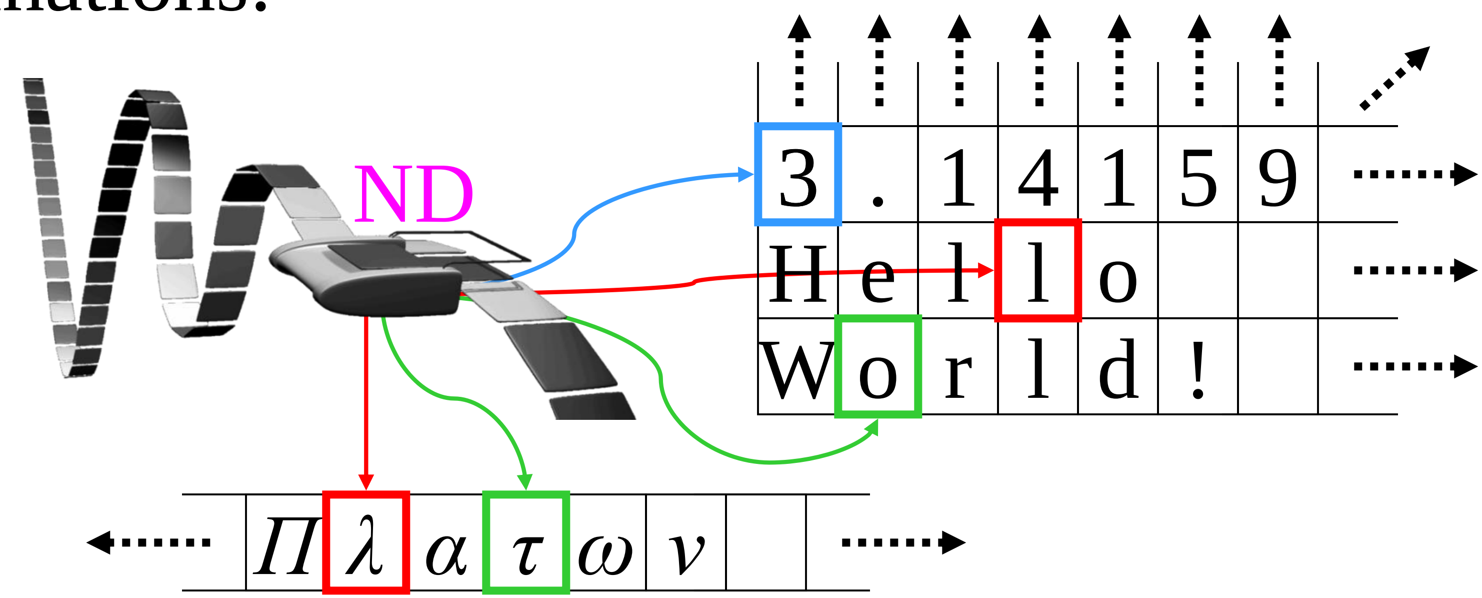


Modified deterministic δ' simulates the original ND δ:

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

Turing Machine "Enhancements"

Combinations:

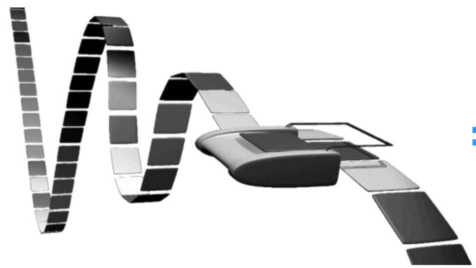


Idea: "Enhancements" are independent (and commutative with respect to preserving the language recognized).

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

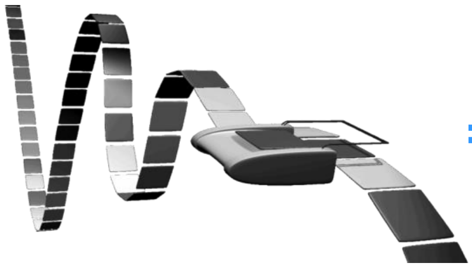
Def: A language is Turing-decidable iff it is exactly the set of strings accepted by some always-halting TM.

$w \in \Sigma^* =$	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.

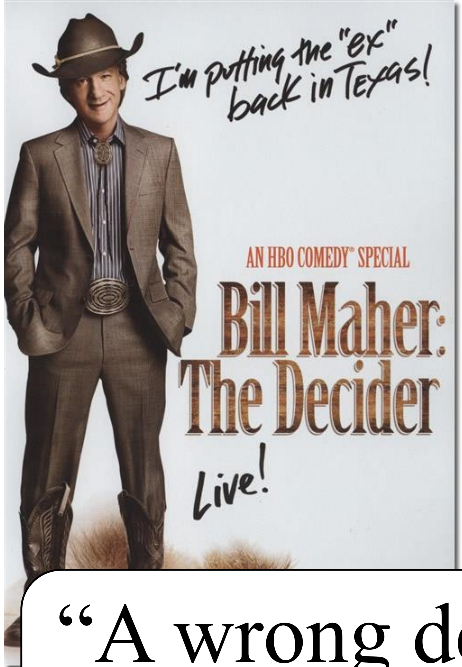
Theorem: Some recognizable languages are not decidable.

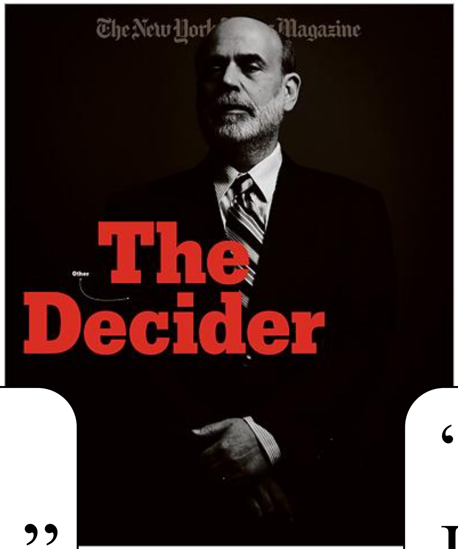
Ex: The halting problem is recognizable but not decidable.

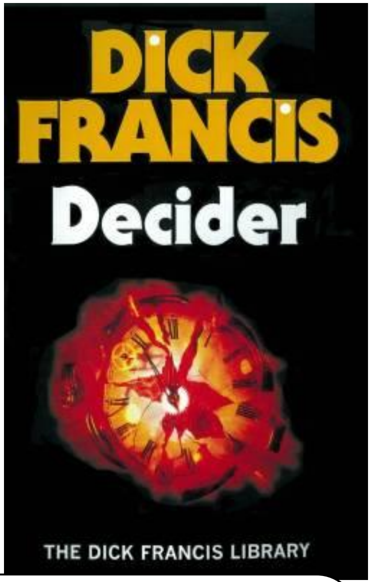
Note: Decidability is a special case of recognizability.

Note: It is easier to recognize than to decide.

Famous Deciders

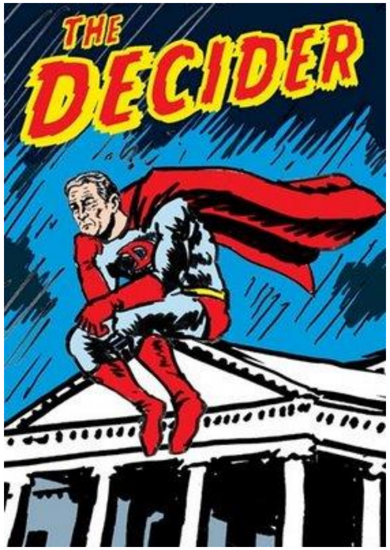




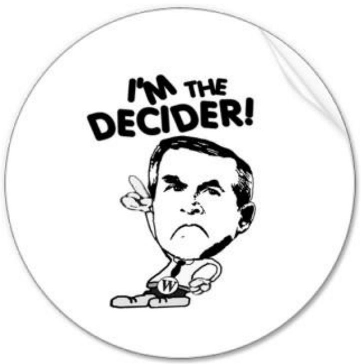


"A wrong decision is better than indecision."

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

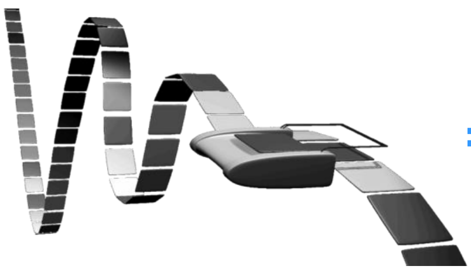


Famous Deciders



Recognition and Enumeration

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



a	$	a	b	$	b	b	a	$	

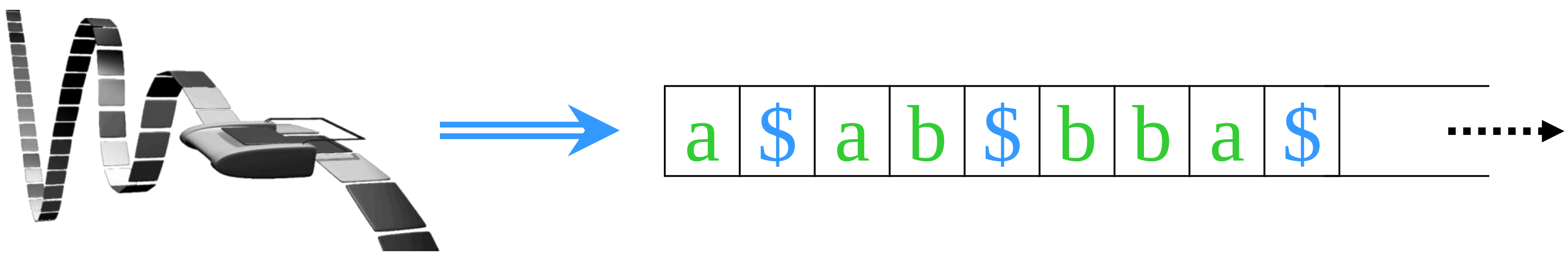
Note: The order of enumeration may be arbitrary.

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

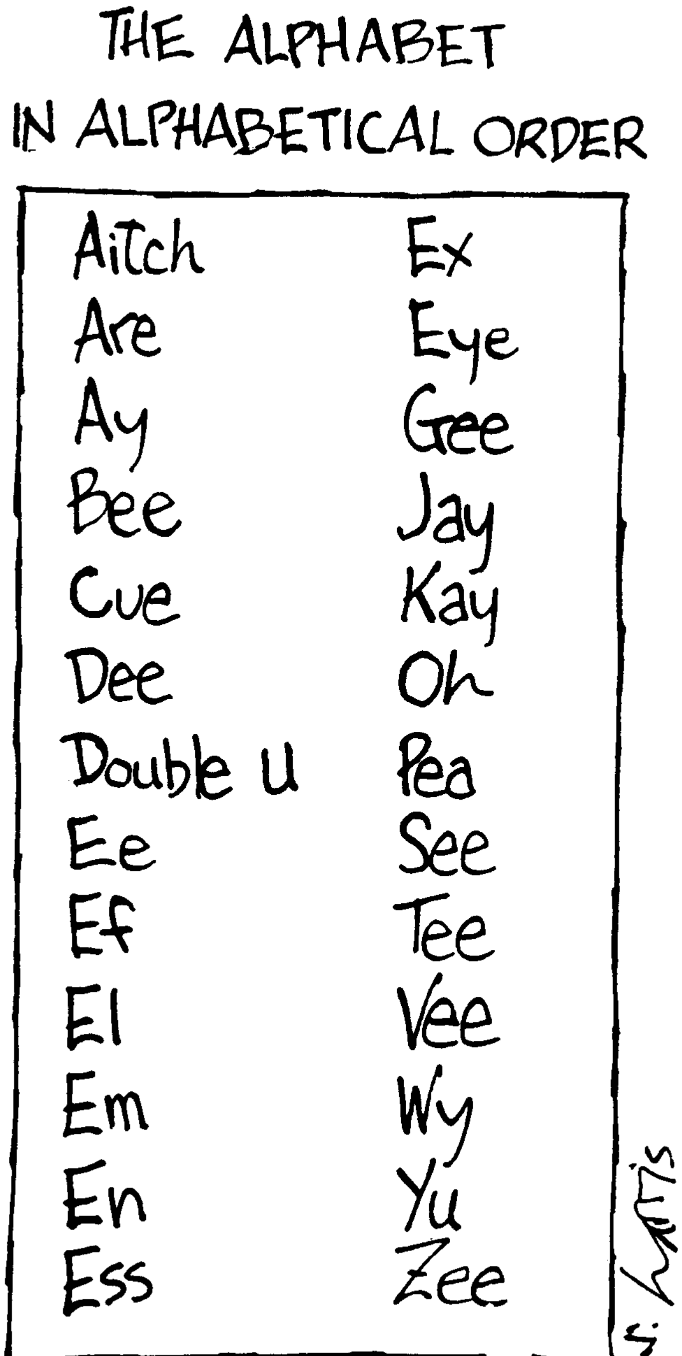
Def: An “enumerator” Turing machine for a language L prints out precisely all strings of L on its output tape.

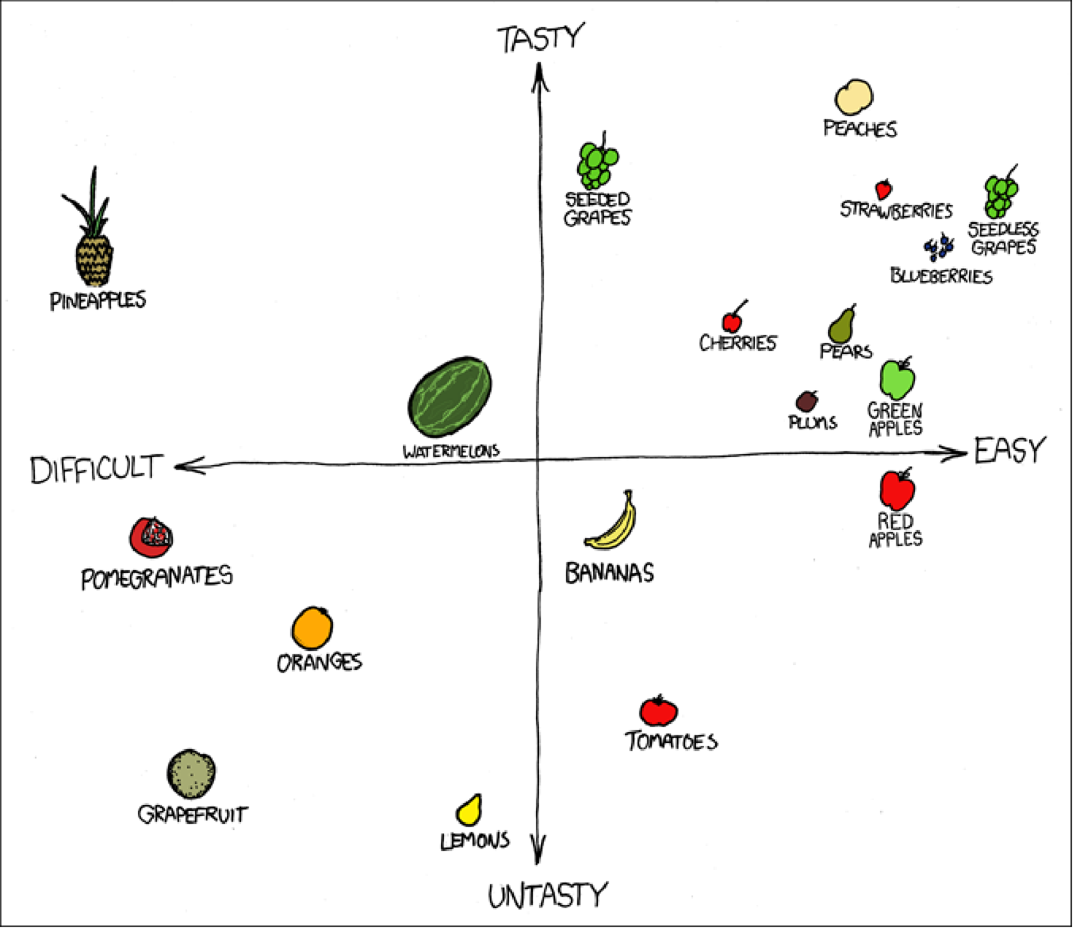


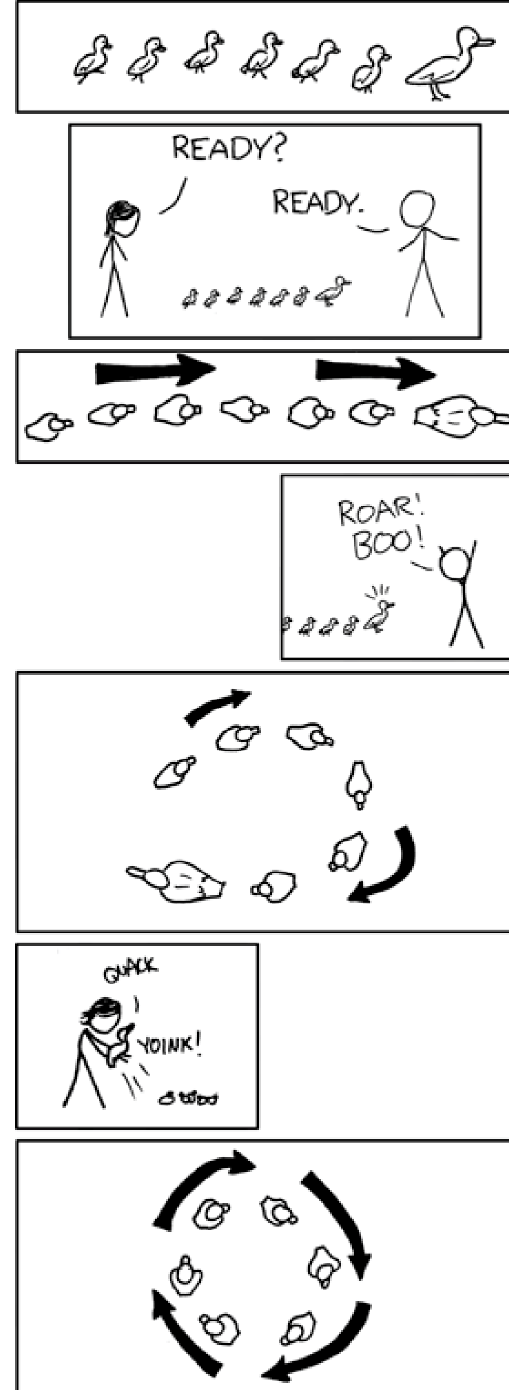
Note: The order of enumeration may be arbitrary.

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

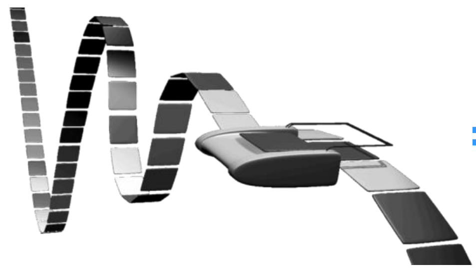




OPERATION: DUCKLING LOOP

Decidability

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.

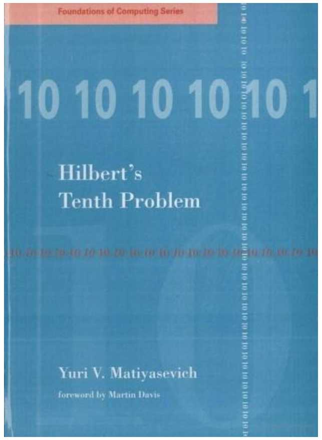
Theorem: The context-free languages are decidable.

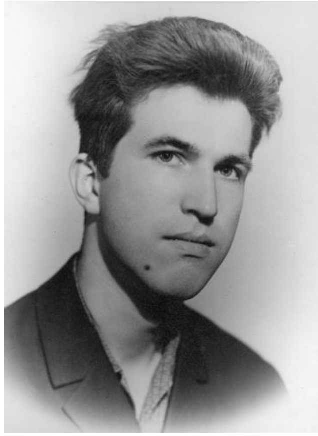
A "Simple" Example

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.

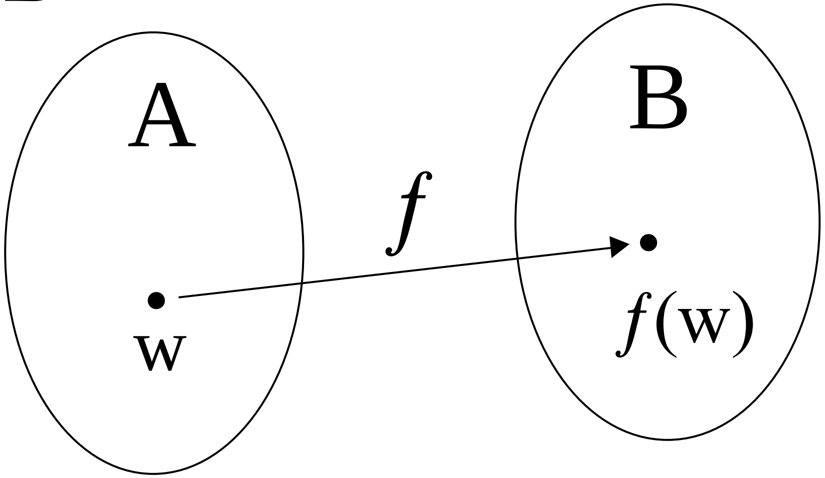
Hint: guess-factor string and simulate.

Reducibilities

Def: A language A is reducible to a language B if

$\exists$ computable function/map $f{:}\Sigma^*\to\Sigma^*$ where

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



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

Denotation: $A \le B$

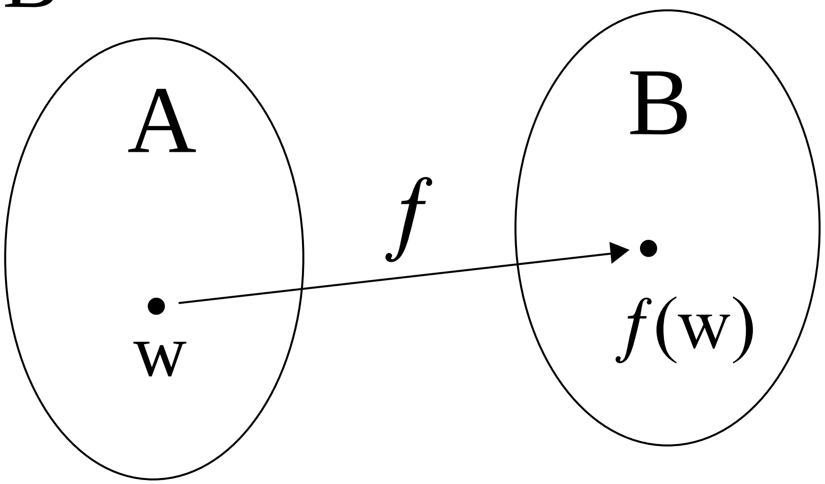
Intuitively, A is "no harder" than B

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=ε.

“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:

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.

x → M’:
- Ignore x
- Simulate M on w

If M(w) halts then → halt

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 $L_\varnothing$ be the emptyness problem for TMs.

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

Theorem: $L_\varnothing$ is not decidable.

Proof: Reduction from the Halting Problem H:

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

that if it ran on input x, it would:

1. Overwrite x with the fixed w on tape;
2. Simulate M on the fixed input w;
3. Accept $\Leftrightarrow$ M accepts w.

x → M'
- Ignore x
- Simulate M on w

If M(w) halts then → halt

Note: M' is not run!

Note: M' halts on every $x \in \Sigma^* \Leftrightarrow$ M halts on w.

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

Since H is undecidable, $L_\varnothing$ 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 not decidable.

Proof: Reduction from the Halting Problem H:

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

that if it ran on input x, it would:

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

x →

M'

- Accept if $x \in 0^n 1^n$
- Ignore x
- Simulate M on w

If M(w) halts then → halt

Note: M' is not run!

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

$L(M') = 0^n 1^n \Leftrightarrow$ 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 →

- Ignore x
- Say "no"
- Stop

M_{none} → no

M_{none} decides P_{none}

P_{all}={L | L is a recognizable language}

x →

- Ignore x
- Say "yes"
- Stop

M_{all} → 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

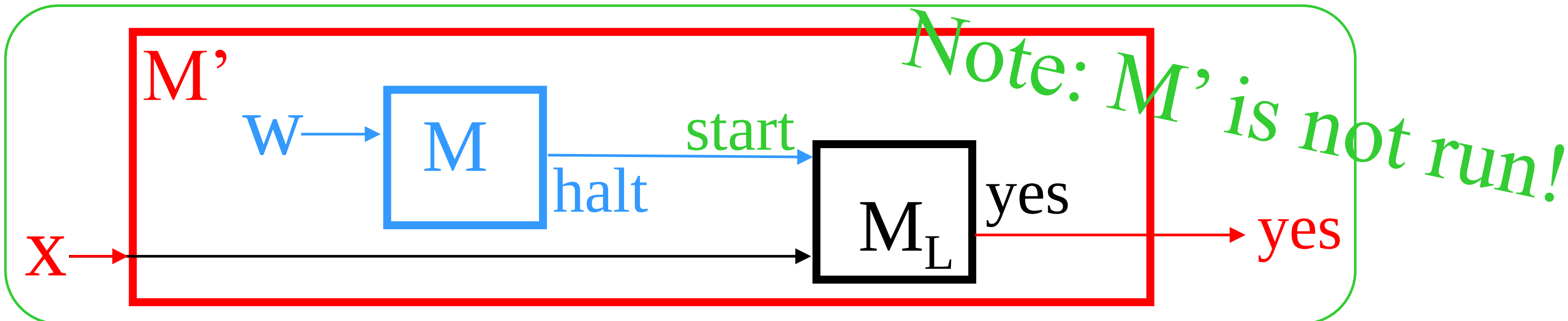
Note: x can be e.g., a TM description.

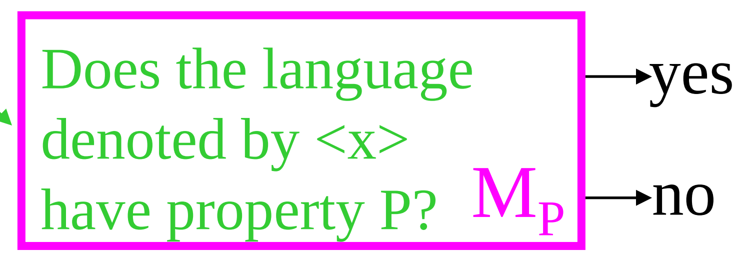
Rice's Theorem

Reduction strategy: use M_p to "solve" the halting problem.

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

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





What is the language of M'?

$L(M')$ is either $\emptyset$ 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 $L(M')= \emptyset$ 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!

Context-Sensitive Grammars

Problem: design a context-sensitive grammar to accept 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 */

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

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

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

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

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

$\rightarrow 11\underline{N}D11DE$

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

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

$\rightarrow 111N11D1D1\underline{DE}$

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

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

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

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

$\rightarrow 111\$11111\underline{D1}DE$

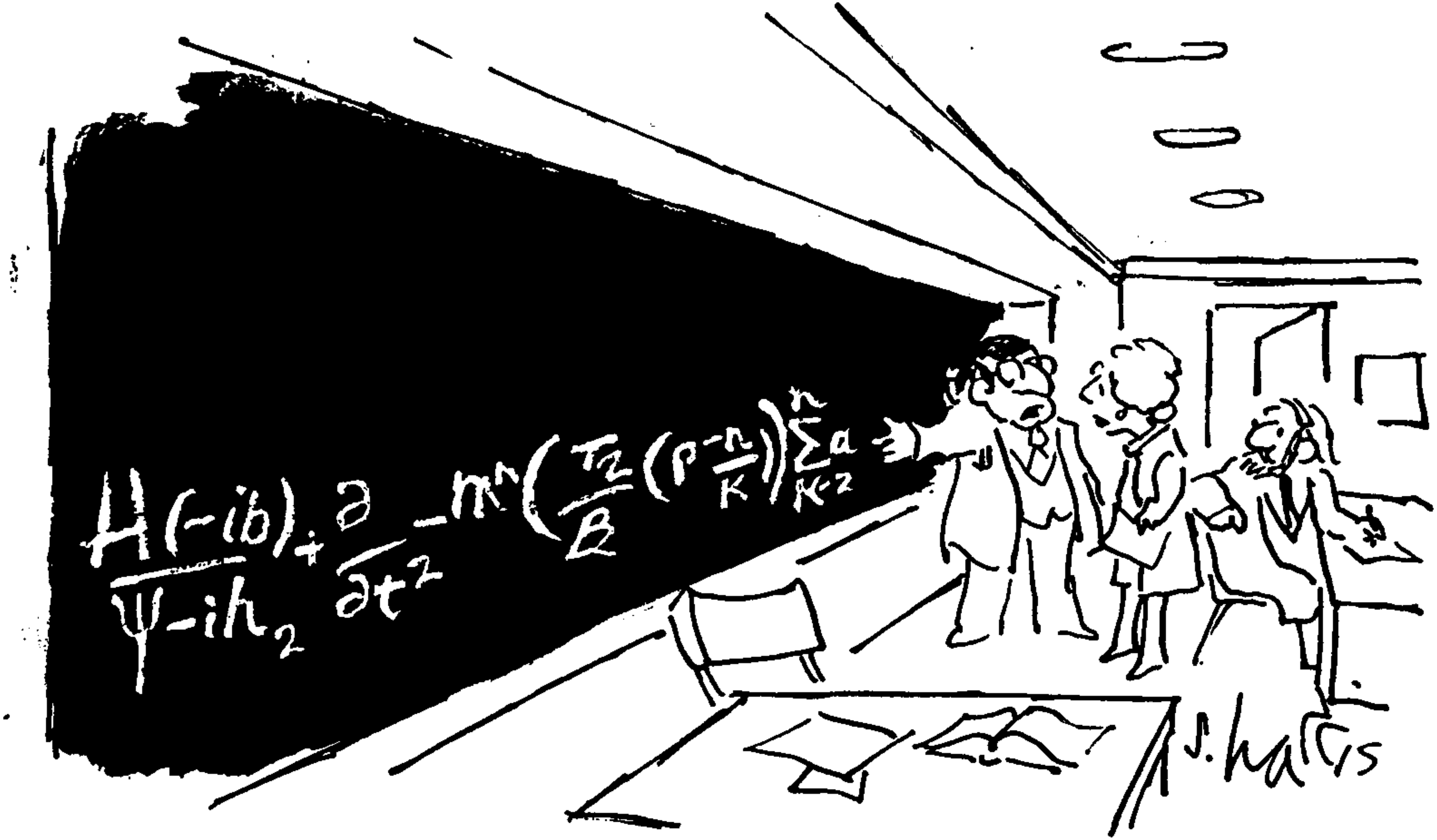
$\rightarrow 111\$1111111D\underline{DE}$

$\rightarrow 111\$1111111\underline{DE}$

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

$\rightarrow 111\$11111111\varepsilon$

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



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