Welcome to the Course

Hi! Welcome to CSC280. I’m glad you’re taking it, and (although I realize it is a B.S. requirement and—worse still!—it covers what certainly is the most abstract material in our undergraduate core), I hope that you’ll be glad you took it!

After all, most of you are computer science majors, and this course is about getting a firm, rigorous handle on what it means to be a computer of a given type, i.e., we will try to capture what it means to be a model of computation. And we will see how to prove that models have, or lack, the ability to accept certain problems.
Welcome to the Course, cont’d

Indeed, computer science—and the challenges computer scientists (and real world programmers) face—is to a surprisingly large extent about hard or easy problems are. (My life is particularly much about that; I’m a complexity theorist!Ψ) Here are some examples (by the end of the course we’ll know the answers to the red ones as well as anyone on Earth, although that is a trick claim in the third case):

• Can a machine with finite memory recognize the palindromes?
• Is \( \{ww \mid w \in \{0, 1\}^*\} \) a context-free language?
• Can we in polynomial time tell whether a given Boolean formula is satisfiable?
• Does there exist a Turing machine that, given as input another Turing machine \( B \), decides whether \( B \) accepts at least one string?
Questions of the difficulty of problems are everywhere. They touch all aspects of our life and our profession, from helping us build compilers to telling us why perfect compiler optimization is impossible; from (and this is my own work’s current focus) helping us protect elections from manipulative attacks to helping us mount those attacks; from scheduling things well, to building small computer circuits, to searching for patterns, to securely encoding messages.

And as CS people, to better understand problems, and see their commonalities, we group them into classes, typically based on the type/amount/power of computation that suffices to solve the problems.
Welcome to the Course, cont’d

As you might imagine, there is a huge array of classes by now, capturing all sorts of approaches to computation and all sorts of models of computation—ranging from DNA-based computing to quantum computing to probabilistic computing to unambiguous computing to nondeterministic computing to things so strange that they should be drawn as giant green animals with three heads in Dr. Seuss books.

Even experts don’t know all the models. But in this course, we’ll learn how to formalize and study some of the most beautiful and important models and classes.
Welcome to the Course, cont’d

In particular, we’ll intensely study the regular languages (and their machine model, finite automata—both deterministic and nondeterministic), the context-free languages (and their machine model, pushdown automata), and the recursive and recursively enumerable languages (and their machine models, respectively, total Turing machines and Turing machines). We’ll even learn a bit about P and NP—deterministic and nondeterministic polynomial time. (If you are the first to prove that the classes P and NP differ, or that they collapse, you’ll win a million dollars. Let’s see your English course cough up that kind of incentive!)
Welcome to the Course, cont’d

Proving that a problem is not contained in a class is can be hugely challenging. You typically have to show that every single one of the infinite number of computers/programs/automata modeling the class doesn’t solve the problem. Since infinity is sort of big, this could take your lifetime and more, unless you have the right tools, knowledge, and skills.

So where are you going to get those tools, knowledge, and skill?
Welcome to the Course, cont’d

Surprise—you’re there! CSC280! Here, we’ll learn about tools to let us prove that problems are not in various classes, and tools and approaches that let us prove that problems are in various classes, and we’ll get hands-on practice using them. Some of the tools look like tools (e.g., the so-called pumping lemmas), but many are simply about understanding the machine models for a class, so we can show something is in the class by building the machine (i.e., “program”) that by its existence puts the problem into the class.
Welcome to the Course, cont’d

So buckle up, get ready for a demanding course, but with luck also one where you are challenged, have some fun, and get a real feel for how to prove that problems are, or are not, in some of the most important classes. (And if you really like doing this, keep in mind that CSC286/CSC486 is in effect a continuation of course, and among other things it will intensely study such resource-bounded classes as P, NP, and the polynomial hierarchy—triple yummy!)

Or, to come back to where we started:

Welcome to CSC 280!

Cheers,
Lane
Credit Where Credit Is Due: A Note on The Slides

It is always important to give credit where credit is due. Mitsunori Ogihara was a professor here for a long time, and even chaired our department for eight years and coauthored a complexity book with me, before the sunshine and alligators lured him to Florida’s University of Miami. His slides are by far the best I know of for this course, and even after leaving he very kindly agreed to let us at UR use them. (Of course, if something is an obvious insertion from me, such as all the slides so far, it would be wrong to blame Mitsu. Also, over the years that I’ve used them, I’ve edited/modified many of the slides, and so even problems in things that are not obvious insertions-by-me could well be due to me.)–Cheers, Lane
Chapter 0: Fundamental Concepts

Fundamental Concepts
What Is Computation?

Computation is a systematic way of obtaining an answer to a problem.

The systematic nature of computation allows the use of computing devices for actual computation.

The abstraction of devices of the “same kind” is a model:

- “Is this model ‘different’ from that model?”
- “Can we solve this problem in that model?”
Problem Classes

“Computation model” + “Concept of Solving Problems (Typically Already Built into the Model’s Definitions Through the Notion of Acceptance of Inputs)” = “Class of Problems (Namely, Those Accepted by Some Machine of that Model)”

• “Is this model different from that model?” can in a certain sense of “different” be addressed via “Is class $A$ equal to class $B$?”

• “Can we solve this problem in that model?” can be understood by getting a handle on “Exactly what is class $A$?... Is this problem in $A$?”
Class Overview: Classes of Computation

The Universe of Languages

The Recursively Enumerable Languages

The Recursive Languages

The Context-free Languages

The Regular Languages
(and Now on We Go to...) Fundamental Concepts
Sets

- $a \in S$: $a$ is an element of $S$; $a$ is a member of $S$.
  
  **Example**  $1 \in \{1, 2, 3\}$.

- $S \subseteq T$: $S$ is a subset of $T$; $S$ is contained in $T$. This means that every member of $S$ is a member of $T$.
  
  **Example**  $\{1, 2\} \subseteq \{1, 2, 3\}$,  $\{1, 2\} \subseteq \{1, 2\}$,  and  $\{1, 4\} \not\subseteq \{1, 2, 3\}$.

- $S \subset T$: $S$ is a proper subset of $T$; $S$ is properly contained in $T$. This means that $S \neq T$ and $S \subseteq T$.
  
  **Example**  $\{1, 2\} \subset \{1, 2, 3\}$ and $\{1, 2\} \not\subset \{1, 2\}$.

- $\emptyset$ is the empty set, the set without elements.

- $2^S$ is a power set of $S$; i.e., the set of all subsets of $S$. 
**Set Operations**

- $S \cap T$: the **intersection** of $S$ and $T$; the set of all common members between $S$ and $T$.
  
  **Example** $\{1, 2, 3\} \cap \{1, 2, 4\} = \{1, 2\}$.

- $S \cup T$: the **union** of $S$ and $T$; the set of all members of $S$ or $T$.
  
  **Example** $\{1, 2, 3\} \cup \{1, 2, 4\} = \{1, 2, 3, 4\}$.

- $S \setminus T$: the **set difference** of $S$ and $T$; i.e., the set consisting of all members of $S$ that are nonmembers of $T$.
  
  **Example** $\{1, 2, 3\} \setminus \{1, 2, 4\} = \{3\}$.

- If $T \subseteq S$ (or even if not), we write $S - T$ to mean $S \setminus T$.
  
  **Example** $\{1, 2, 3\} - \{1, 2\} = \{3\}$.

- $S \triangle T$: the **disjoint union** of $S$ and $T$. $S \triangle T = (S \setminus T) \cup (T \setminus S)$.
  
  **Example** $\{1, 2, 3\} \triangle \{1, 2, 4\} = \{3, 4\}$.
Set Operations (cont’d)

- \( S \times T \): the **Cartesian product** of \( S \) and \( T \); i.e., \( \{(a, b) \mid a \in S \text{ and } b \in T\} \).

**Example** \( \{a, b, c\} \times \{1, 2\} = \{(a, 1), (a, 2), (b, 1), (b, 2), (c, 1), (c, 2)\} \).

- \( \|S\| \): the **cardinality** of the set \( S \); i.e., the number of elements in \( S \).

**Example** \( \|\{a, b, c\}\| = 3 \).

- Quite often \( |\cdot| \) is used for the cardinality. (But it is wiser to reserve it for “length of” strings and so on.)
Alphabet, Strings, Languages, etc.

- An **alphabet** is any finite set, whose members are called **symbols**.

- A **string (or word)** **over an alphabet** is a sequence of symbols from the alphabet written one after another.
  
  **Example** $aba$ is a word over an alphabet $\{a, b\}$

- The **length** of a word $w$, denoted by $|w|$, is the number of symbols in it.
  
  **Example** If $w = aba$, then $|w| = 3$.

- The **empty string or null string**, denoted by $\epsilon$, is the string with no symbols in it.
Alphabet, Strings, Languages, etc. (cont’d)

• A string $z$ is a substring of $w$ if $z$ appears consecutively within $w$.

  Example Let $z = 00111010$. Then 1111 is a substring of $z$ while 11111 is not.

• The concatenation of strings $x$ and $y$ is the string constructed by appending $y$ after $x$.

  Example The concatenation of $a = 000$ and $b = 111$ is 000111.

• A language is a collection of strings.

• A class is a collection of languages.
For an alphabet $\Sigma$, $\Sigma^*$ is the set of all strings over $\Sigma$.

The complement of a language is the collection of all nonmembers; for a language $L$ over an alphabet $\Sigma$, its complement is $\Sigma^* - L$ and is denoted by $L^c$ or $\overline{L}$.

**Example** If $\Sigma = \{a, b\}$ and $L$ is the set of all strings over $\Sigma$ having an even number of $a$’s, then $\overline{L}$ is the set of all strings over $\Sigma$ having an odd number of $a$’s.
• If $\Sigma$ is a single-letter alphabet with $a$ as its unique symbol, we often write $a^*$ for $\Sigma^*$.

• For a language $L$, $L^*$ is the set of all strings constructed by concatenating any strings from $L$ in any order. That is, $L^* = \{\varepsilon\} \cup \{x_1 \cdots x_m \mid m \geq 1, x_1, \ldots, x_m \in L\}$.

**Example**  $\{a, ab\}^*$ is the set of all strings $w$ over $a$ and $b$ such that either $w$ is empty or ($w$ begins with an $a$ and has no $bb$ as a substring).
A **Boolean variable** takes on one of 0 (FALSE) and 1 (TRUE). The **negation** of $x$, denoted by $\overline{x}$ or $\neg x$, is $1 - x$.

We will be using six **binary Boolean operators**: 

<table>
<thead>
<tr>
<th>$(x, y)$</th>
<th>$(0, 0)$</th>
<th>$(0, 1)$</th>
<th>$(1, 0)$</th>
<th>$(1, 1)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\land$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$\lor$</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$\to$</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$\leftarrow$</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$\leftrightarrow$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$\oplus$</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Boolean Logic (cont’d)

A **predicate** is a **function** whose **range** is \{ TRUE, FALSE \}. A **relation** is a predicate whose number of arguments is fixed to a constant.

**Properties of binary relation** $R$ **over domain** $D$.

- **Reflexive**: For all $x \in D$, $xRx$.
- **Symmetric**: For all $x, y \in D$, $xRy \iff yRx$.
- **Transitive** For all $x, y, z \in D$, $(xRy \land yRz) \rightarrow xRz$.

An **equivalence relation** is a binary relation that is reflexive, symmetric, and transitive.
Proof by Induction

A method for proving a statement $P$. Divide the statement $P$ into cases $P(n), n = a, a+1, a+2, \ldots$ For the base case, prove $P(a)$. For the induction step, assume that $P(n)$ is true for all values of $n \leq k$ and show that $P(k+1)$ holds.
Graphs

A graph consists of nodes (vertices) and edges. A path is a sequence of edges (or a sequence of nodes) that connects from a node to another. A tree is a connected, undirected graph without cycles.
Chapter 1, Part 1

Regular Languages
Finite Automata

A finite automaton is a system for processing any finite sequence of symbols, where the symbols are chosen from a finite set of symbols.

The goal is to determine whether the sequence has a certain property by simply reading the symbols of the sequence from the beginning to the end.
An Illustrating Example of Finite Automata

A coin exchanger takes nickels or dimes and delivers quarters. It takes coins one at a time. When the deposited amount reaches or goes beyond 25 cents it delivers a quarter. There is no “change” button and any change is carried over as a deposit.

For example, if the deposit is currently 20 cents, upon receiving a dime, the machine delivers a quarter and the deposit becomes 5 cents.
Question

If you have a bag full of nickels and dimes and use this machine to change for quarters, do you break even?
Relationship Between Deposit and Coin Inserted

The deposit amount in cents is one of 0, 5, 10, 15, and 20. Upon receiving a coin, the deposit changes as follows:

<table>
<thead>
<tr>
<th>Current Deposit</th>
<th>Coin Inserted</th>
<th>Nickel</th>
<th>Dime</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>15</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>20</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
Visual Representation of the Relationship

Let $N$ stand for “nickel” and $D$ for “dime”. For $a \in \{0, 5, 10, 15, 20\}$, let $q_a$ represent the status in which the deposit is $a$ cents.
Finite Automata

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

1. $Q$ is a finite set called the states,
2. $\Sigma$ is a finite set called the alphabet,
3. $\delta : Q \times \Sigma \rightarrow Q$ is the transition function,
4. $q_0 \in Q$ is the initial state, and
5. $F \subseteq Q$ is the set of accepting states or the set of final states.

Let $M = (Q, \Sigma, \delta, q_0, F)$ be an FA. A string $w = w_1 \cdots w_n$ is accepted by $M$ if there exists a sequence $(p_0, \ldots, p_n)$ of states in $Q$ such that $p_0 = q_0$, $p_n \in F$, and for every $i, 1 \leq i \leq n$, $\delta(p_{i-1}, w_i) = p_i$. 
The Language Decided by a Finite Automaton

The language **decided** by \( M \), denoted \( L(M) \), is the language over \( \Sigma \) such that

\[(*)\text{ for every string } w \text{ over } \Sigma, \ w \in L(M) \iff M \text{ accepts } w.\]
An FA for the Coin Changer

Let $\Sigma = \{N, D\}$.
Let $Q = \{q_0, q_5, q_{10}, q_{15}, q_{20}\}$.
The transition function is:

<table>
<thead>
<tr>
<th>state</th>
<th>$N$</th>
<th>$D$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>$q_5$</td>
<td>$q_{10}$</td>
</tr>
<tr>
<td>$q_5$</td>
<td>$q_{10}$</td>
<td>$q_{15}$</td>
</tr>
<tr>
<td>$q_{10}$</td>
<td>$q_{15}$</td>
<td>$q_{20}$</td>
</tr>
<tr>
<td>$q_{15}$</td>
<td>$q_{20}$</td>
<td>$q_0$</td>
</tr>
<tr>
<td>$q_{20}$</td>
<td>$q_0$</td>
<td>$q_5$</td>
</tr>
</tbody>
</table>

$F = \{q_0\}$.

Our FA accepts $NNND$ and $DDDDDD$ but not $NNN$. 
The Coin Changer As a Finite Automaton
Regular Languages

The regular languages is the class of languages accepted by finite automata.
Example 1

An FA that accepts the strings over 0 and 1 with either (an even number of 0s and an even number of 1s) or (an odd number of 0s and an odd number of 1s)

Drawing rules:
- The initial state has an incoming edge from outside.
- Accept states are represented with double circles.
- Every node has one outgoing edge for each symbol.
Example 2

An FA that accepts the set of all words over \( \{a, b\} \) having odd length
Example 3

An FA that accepts the set of all words over \( \{a, b\} \) containing \( ababb \) as a substring.
Example 4

An FA that accepts the set of all words over \{a, b\} containing at least three \(a\)’s and at least two \(b\)’s
Example 4

An FA that accepts the set of all words over \( \{a, b\} \) containing at least three \( a \)'s and at least two \( b \)'s

“At least two \( b \)'s” and “at least three \( a \)'s”
Example 4

An FA that accepts the set of all words over \( \{a, b\} \) containing at least three \( a \)'s and at least two \( b \)'s
Example 5

An FA that accepts the set of all words over \( \{a, b\} \) containing either as least three \( a \)'s or at least two \( b \)'s
Example 5

An FA that accepts the set of all words over \( \{a, b\} \) containing either as least three \( a \)'s or at least two \( b \)'s

![FA Diagram]
Chapter 1, Part 2

Nondeterministic Finite Automata
Fundamental set operations

Let $A$ and $B$ be two languages. In addition to union and intersection, we consider the following set operations:

- **Concatenation** of $A$ and $B$, $A \circ B$, is \{xy \mid x \in A \text{ and } y \in B\},

- **Star** of $A$, $A^*$, is \{x_1x_2\cdots x_k \mid k \geq 0 \text{ and } x_1, \ldots, x_k \in A\}. \quad \text{(Note: For each set } A, \epsilon \in A^*. \text{ In particular, and many find this surprising and counterintuitive, } \emptyset^* = \{\epsilon\}. \text{ One way to view this is that the concatenation of zero strings from any set—even if the set is } \emptyset \text{—is taken to be } \{\epsilon\}.\}

- **Complement** of $A$, $\overline{A}$, is $\Sigma^* - A$, i.e., \{w \mid w \in \Sigma^* \land w \notin A\}.
Fundamental set operations

Let $A$ and $B$ be two languages. In addition to union and intersection, we consider the following set operations:

- **Concatenation** of $A$ and $B$, $A \circ B$, is $\{xy \mid x \in A \text{ and } y \in B\}$.

- **Star** of $A$, $A^*$, is $\{x_1x_2\cdots x_k \mid k \geq 0 \text{ and } x_1, \ldots, x_k \in A\}$. (Note: For each set $A$, $\epsilon \in A^*$. In particular, and many find this surprising and counterintuitive, $\emptyset^* = \{\epsilon\}$. One way to view this is that the concatenation of zero strings from any set—even if the set is $\emptyset$—is taken to be $\{\epsilon\}$.)

- **Complement** of $A$, $\overline{A}$, is $\Sigma^* - A$, i.e., $\{w \mid w \in \Sigma^* \land w \not\in A\}$.

We will prove that the regular languages are closed under union, intersection, concatenation, star, and complement; i.e., that if $A$ and $B$ are regular, then $A \cup B$, $A \cap B$, $A \circ B$, $A^*$, and $\overline{A}$ are each regular.
Nondeterministic Finite Automata

The **nondeterministic finite automaton** is a variant of finite automaton with two characteristics:

- **ε-transition**: state transition can be made *without reading a symbol*;
- **nondeterminism**: zero or more than one possible value may exist for state transition.
An Example Nondeterministic Finite Automaton

An NFA that accepts all strings over \( \{0,1\} \) that contain a 1 either at the third position from the end or at the second position from the end.

There are two edges labeled 1 coming out of \( q_1 \).

There are no edges coming out of \( q_4 \).

The edge from \( q_2 \) is labeled with \( \epsilon \), in addition to 0 and 1.
What the Diagram Says

- If the node you are in has an outgoing edge labeled $\epsilon$, you may choose to follow it (even if there are still symbols waiting to be read).
- When you choose to read the next symbol,
  - IF the node you are in has one or more outgoing edges labeled with the symbol, you nondeterministically choose one of them and cross it ELSE (if there are no outgoing edges with that symbol) our current computation path dies.

The above is informal; see the formal definitions to come for what is really going on in nondeterministic finite automata.
Nondeterministic Finite Automata, Formally

A **nondeterministic finite automaton** is a 5-tuple $N = (Q, \Sigma, \delta, q_0, F)$, where $\delta$ is a mapping of $Q \times \Sigma_\epsilon$ to $2^Q$ (alternatively, written $\mathcal{P}(Q)$), and $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$.

For each $p \in Q$ and each $a \in \Sigma_\epsilon$, $\delta(s, a) = R$ means:

“upon reading an $a$ the automaton $N$ may transition from state $s$ to any state in $R$.”

For $\epsilon$ in particular, $\delta(s, \epsilon) = R$ means:

“without reading a symbol the automaton $N$ may transition from state $s$ to any state in $R$.”
Acceptance by Nondeterministic Finite Automata

$N$ accepts a word $w = w_1 \cdots w_n, \forall w_1, \ldots, w_n \in \Sigma$ if: there exist $(p_0, \ldots, p_m), p_0, \ldots, p_m \in Q$ and $y_1, \ldots, y_m \in \Sigma_\epsilon$ such that

- $w = y_1 \cdots y_m,$
- $p_0 = q_0,$
- $p_m \in F,$ and
- for every $i, 1 \leq i \leq m, p_i \in \delta(p_{i-1}, y_i).$
## Transition Function for the Previous Example

<table>
<thead>
<tr>
<th>state</th>
<th>symbol</th>
<th>0</th>
<th>1</th>
<th>$\epsilon$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_1$</td>
<td>${q_1}$</td>
<td>${q_1, q_2}$</td>
<td>$\emptyset$</td>
<td></td>
</tr>
<tr>
<td>$q_2$</td>
<td>${q_3}$</td>
<td>${q_3}$</td>
<td>${q_3}$</td>
<td></td>
</tr>
<tr>
<td>$q_3$</td>
<td>${q_4}$</td>
<td>${q_4}$</td>
<td>$\emptyset$</td>
<td></td>
</tr>
<tr>
<td>$q_4$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td></td>
</tr>
</tbody>
</table>
Transition Function for the Previous Example

This NFA accepts $y = 11$ with respect to state sequence $(q_1, q_2, q_3, q_4)$ and decomposition $y = 1\epsilon 1$. 

<table>
<thead>
<tr>
<th>state</th>
<th>symbol</th>
<th>0</th>
<th>1</th>
<th>$\epsilon$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_1$</td>
<td></td>
<td>${q_1}$</td>
<td>${q_1, q_2}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$q_2$</td>
<td></td>
<td>${q_3}$</td>
<td>${q_3}$</td>
<td>${q_3}$</td>
</tr>
<tr>
<td>$q_3$</td>
<td></td>
<td>${q_4}$</td>
<td>${q_4}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$q_4$</td>
<td></td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>
Nondeterministic computation can be thought of as a self-reproducing agent traveling in the state space.

1. At the start of computation the agent is in the initial state.
Nondeterministic Choices

Nondeterministic computation can be thought of as a self-reproducing agent traveling in the state space.

1. At the start of computation the agent is in the initial state.
**Nondeterministic Choices**

2. Both before and after receiving an input symbol, the agent follows each $\epsilon$-labeled outgoing edge (or edge-chain) by producing its own clones and sending them along each edge (or edge-chain). The original remains in the current location.

(Warning and reminder: Despite the informal text above, the formal definition actually uses multiple $\epsilon$’s to run through chains of $\epsilon$’s, one at a time.)
Nondeterministic Choices

2. Both before and after receiving an input symbol, the agent follows each $\epsilon$-labeled outgoing edge (or edge-chain) by producing its own clones and sending them along each edge (or edge-chain). The original remains in the current location.
Nondeterministic Choices

3. On receiving an input symbol, say \( a \),
   (a) If there is only one \( a \)-labeled outgoing edge, the agent follows the edge.
   
   (b) If there is no \( a \)-labeled outgoing edge, the agent evaporates.
**Nondeterministic Choices**

3. On receiving an input symbol, say $a$,
   (a) If there is only one $a$-labeled outgoing edge, the agent follows the edge.
   (b) If there is no $a$-labeled outgoing edge, the agent evaporates.

![I am gone.](image)
Nondeterministic Choices

3. On receiving an input symbol, say $a$,
   (c) If there are $k \geq 2$ $a$-labeled outgoing edges, the agent produces $k - 1$ clones, make them cross $k - 1$ of the edges, and cross the remaining one by himself.
Nondeterministic Choices

3. On receiving an input symbol, say \( a \),
   (c) If there are \( k \geq 2 \) \( a \)-labeled outgoing edges, the agent
       produces \( k - 1 \) clones, make them cross \( k - 1 \) of the
       edges, and cross the remaining one by himself.
Nondeterministic Choices

4. When two agents collide in a state, they merge themselves into one.
Nondeterministic Choices

4. When two agents collide in a state, they merge themselves into one.
Why Use NFA?

For some languages construction is much easier.

Below is a DFA that accepts the same language by remembering the last three symbols.
Comparison
Example 1

An NFA for the language of all strings over \( \{a, b, c\} \) that end with one of \( ab, bc, \) and \( ca. \)
Example 1

An NFA for the language of all strings over \( \{a, b, c\} \) that end with one of \( ab, bc, \) and \( ca. \)
A DFA Version of the Same Language
Example 2

An NFA for the language of all strings over \( \{0, 1\} \) that end with one of 0110, 010, and 00.
Example 2

An NFA for the language of all strings over \(\{0, 1\}\) that end with one of 0110, 010, and 00.
Example 3

An NFA for the language of all strings over \( \{a, b, c\} \) for which one of (the number of occurrences of \( a \)), (the number of occurrences of \( b \)), and (the number of occurrences of \( c \)) is a multiple of 3.
Example 3

An NFA for the language of all strings over \( \{a, b, c\} \) for which one of (the number of occurrences of \( a \)), (the number of occurrences of \( b \)), and (the number of occurrences of \( c \)) is a multiple of 3.
Example 4

An NFA for the language of all strings over \{a, b\} that contain \textit{ababb}.
Example 4

An NFA for the language of all strings over \( \{a, b\} \) that contain \( ababb \).
The NFA simplifies computational design, but the use of nondeterministic selections and $\epsilon$-transitions makes it look very different from FA.

Is that really so?
The FA Model Versus NFA Model

The NFA simplifies computational design, but the use of nondeterministic selections and \( \varepsilon \)-transitions makes it look very different from FA.

Is that really so?

**No, the FA model is equivalent to the NFA model.** That is, every language accepted by an DFA is accepted by an NFA, vice versa.

Obviously, we have:

**Theorem.** Every FA is already an NFA.

We will show:

**Theorem.** Every NFA can be converted to an equivalent FA.
Proof

The principle is that just before the first symbol is received and after each symbol is read, there will be at most one agent in any state.
Proof

The principle is that just before the first symbol is received and after each symbol is read, there will be at most one agent in any state.

We will thus consider the “set of all states an agent can be in.”
Proof (cont’d)

Let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA.

We will construct a DFA $M = (S, \Sigma, \gamma, s_0, G)$.

- The state set $S$ is $\mathcal{P}(Q)$.
- The initial state $s_0$ is the set consisting of $q_0$ and all the states reachable from $q_0$ by following only $\epsilon$ transitions.
- The final state set $G$ is $\{A \in S \mid A \cap F \neq \emptyset\}$; i.e., the set of all subsets of $Q$ containing an element of $F$. 

Transition Function

The transition function $\gamma$ is defined as follows:

For each $A \in S$ and for each $b \in \Sigma$, $\gamma(A, b) = \bigcup_{p \in A} \delta(p, \epsilon^* b \epsilon^*)$, the collection of all states $r$ that can be reached from a state $p$ in $A$ by following

- any number of $\epsilon$-arrows,
- a $b$-arrow, and then
- any number of $\epsilon$-arrows.

For all $w$ over $\Sigma$, $w$ is accepted by $N$ if and only if the new DFA transitions from $s_0$ to a state in $G$ on input $w$. 
Example

An NFA that recognizes the language consisting of all strings over \{0, 1\} that contain a 1 at either the third to last position or the second to last position.

![NFA Diagram]

CSC280, Chapter 1, Part 2 © 2012 Mitsunori Ogihara. Edits/updates by Lane A. Hemaspaandra, 20{13,14,17}. 38
Conversion to DFA

The state set consists of: $\emptyset$, \{q\textsubscript{1}\}\,\{q\textsubscript{2}\}, \{q\textsubscript{3}\}, \{q\textsubscript{4}\},
\{q\textsubscript{1}, q\textsubscript{2}\}, \{q\textsubscript{1}, q\textsubscript{3}\}, \{q\textsubscript{1}, q\textsubscript{4}\}, \{q\textsubscript{2}, q\textsubscript{3}\}, \{q\textsubscript{2}, q\textsubscript{4}\}, \{q\textsubscript{3}, q\textsubscript{4}\},
\{q\textsubscript{1}, q\textsubscript{2}, q\textsubscript{3}\}, \{q\textsubscript{1}, q\textsubscript{2}, q\textsubscript{4}\}, \{q\textsubscript{1}, q\textsubscript{3}, q\textsubscript{4}\}, \{q\textsubscript{2}, q\textsubscript{3}, q\textsubscript{4}\},
\{q\textsubscript{1}, q\textsubscript{2}, q\textsubscript{3}, q\textsubscript{4}\}.

$F$ consists of: \{q\textsubscript{4}\}, \{q\textsubscript{1}, q\textsubscript{4}\}, \{q\textsubscript{2}, q\textsubscript{4}\}, \{q\textsubscript{3}, q\textsubscript{4}\},
\{q\textsubscript{1}, q\textsubscript{2}, q\textsubscript{4}\}, \{q\textsubscript{1}, q\textsubscript{3}, q\textsubscript{4}\}, \{q\textsubscript{2}, q\textsubscript{3}, q\textsubscript{4}\}, \{q\textsubscript{1}, q\textsubscript{2}, q\textsubscript{3}, q\textsubscript{4}\}.

The initial state is \{q\textsubscript{1}\}.
### Transition

<table>
<thead>
<tr>
<th>State</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>${q_1}$</td>
<td>${q_1}$</td>
<td>${q_1, q_2, q_3}$</td>
</tr>
<tr>
<td>${q_1, q_2, q_3}$</td>
<td>${q_1, q_3, q_4}$</td>
<td>${q_1, q_2, q_3, q_4}$</td>
</tr>
<tr>
<td>${q_1, q_3, q_4}$</td>
<td>${q_1, q_4}$</td>
<td>${q_1, q_2, q_3, q_4}$</td>
</tr>
<tr>
<td>${q_1, q_4}$</td>
<td>${q_1}$</td>
<td>${q_1, q_2, q_3}$</td>
</tr>
<tr>
<td>${q_1, q_2, q_3, q_4}$</td>
<td>${q_1, q_3, q_4}$</td>
<td>${q_1, q_2, q_3, q_4}$</td>
</tr>
</tbody>
</table>

The other states are unreachable from the initial state.
A Greedy Conversion Algorithm

**Step 1** For each state \( p \in Q \) and for each symbol \( a \in \Sigma \), compute the set \( R(p, a) \) of all states that can be reached from state \( p \) by:

(a) any number of \( \epsilon \) transitions,
(b) one transition labeled by \( a \), and then
(c) any number of \( \epsilon \) transitions.

**Step 2** Initialize the state collection \( S \) as \( \{p_0\} \), where \( p_0 \) is the set of all states that can be reached from \( q_0 \) by following any number of \( \epsilon \) transitions.

**Step 3** While \( S \) contains a state with no outgoing edges, select an arbitrary member, say \( r \), of \( S \), and do the following:

- For each symbol \( a \), compute the state \( r_a \) as \( \bigcup_{q \in r} R(q, a) \), add \( r_a \) to \( S \) if \( r_a \) is not already in it, and then draw an arc from \( r \) to \( r_a \).
Algorithm Execution Example

Use the previous NFA.
**Step 1**

<table>
<thead>
<tr>
<th>state</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_1$</td>
<td>$q_1$</td>
<td>$q_1, q_2, q_3$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>$q_3, q_4$</td>
<td>$q_3, q_4$</td>
</tr>
<tr>
<td>$q_3$</td>
<td>$q_4$</td>
<td>$q_4$</td>
</tr>
<tr>
<td>$q_4$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

**Step 2**

Initially $S = \{\{q_1\}\}$. 
Step 3

\{q_1\} on 0 changes the state to \{q_1\}
\{q_1\} on 1 changes the state to \{q_1, q_2, q_3\}. New!
\{q_1, q_2, q_3\} on 0 changes the state to \{q_1, q_3, q_4\}. New!
\{q_1, q_2, q_3\} on 1 changes the state to \{q_1, q_2, q_3, q_4\}. New!
\{q_1, q_3, q_4\} on 0 changes the state to \{q_1, q_4\}. New!
\{q_1, q_2, q_3\} on 1 changes the state to \{q_1, q_2, q_3, q_4\}.
\{q_1, q_2, q_3, q_4\} on 0 changes the state to \{q_1, q_3, q_4\}.
\{q_1, q_2, q_3, q_4\} on 1 changes the state to \{q_1, q_2, q_3, q_4\}.
\{q_1, q_4\} on 0 changes the state to \{q_1, q_3\}.
\{q_1, q_4\} on 1 changes the state to \{q_1, q_2, q_3\}. 
Regular Expression

The regular expressions are equivalent to the finite automata.
An expression \( R \) is a **regular expression** if \( R \) is

1. \( a \) for some \( a \) in some alphabet \( \Sigma \),
2. \( \epsilon \),
3. \( \emptyset \),
4. \( (R_1 \cup R_2) \) for some regular expressions \( R_1 \) and \( R_2 \),
5. \( (R_1 \circ R_2) \) for some regular expressions \( R_1 \) and \( R_2 \), or
6. \( (R_1)^* \) for some regular expression \( R_1 \).

When the meaning is clear from the context, \( () \) and \( \circ \) can be removed from the expression.
The Language Represented by a Regular Expression

For a regular expression $R$, $L(R)$ denotes the language $R$ expresses.

1. For each $a \in \Sigma$, $L(a) = \{a\}$.
2. $L(\epsilon) = \{\epsilon\}$
3. $L(\emptyset) = \emptyset$.
4. $L(R_1 \cup R_2) = L(R_1) \cup L(R_2)$.
5. $L(R_1 \circ R_2) = L(R_1) \circ L(R_2) = \{uv \mid u \in R_1 \text{ and } v \in R_2\}$.
6. $L(R_1^*) = \{\epsilon\} \cup \{u_1 \cdots u_k \mid u_1, \ldots, u_k \in R_1\}$. 
Regular Expression Examples

• $L(a) = \{a\}$ (for a single-element regular expression, you may simply write the element)

• $L(abab \cup bc) = \{abab, bc\}$

• $L((abab \cup abc)^*) = \{\epsilon\} \cup \{w_1 \cdots w_k \mid k \geq 1 \text{ and } w_1, \cdots, w_k \in \{abab, abc\}\}.$

• $L((abab \cup abc)^* \cup c^* \cup abc(abca)^*)$

  $= \{\epsilon\} \cup \{w_1 \cdots w_k \mid k \geq 1 \text{ and } w_1, \cdots, w_k \in \{abab, abc\}\}$

  $\cup \{w \mid w \text{ is a repetition of } c's\}$

  $\cup \{abc, abcabca, abcabcaabca, \cdots\}.$
Finite Automata $\equiv$ Regular Expression

**Theorem.** The regular languages are precisely those that are expressed by regular expressions.

We show:

- Each language expressed by a regular expression can be recognized by a finite automata.
- Regular languages can be expressed as regular expressions.
Regular expressions’ languages can be recognized by FAs

By induction

Base Cases

1. \( a \) for some \( a \) in some alphabet \( \Sigma \),
2. \( \epsilon \),
3. \( \emptyset \),

Induction

4. \( (R_1 \cup R_2) \) for some regular expressions \( R_1 \) and \( R_2 \),
5. \( (R_1 \circ R_2) \) for some regular expressions \( R_1 \) and \( R_2 \), or
6. \( (R_1)^* \) for some regular expression \( R_1 \).
Proof

The Base Case

The following are all regular:

• any set consisting only of a single letter,
• the set consisting only of the empty string,
• the empty set.
The Induction Step

We have only to show the following:

If $L_1$ is accepted by an FA $N_1 = (Q_1, \Sigma, \delta_1, p_1, F_1)$ and $L_2$ is accepted by an FA $N_2 = (Q_2, \Sigma, \delta_2, p_2, F_2)$ (here $Q_1$ and $Q_2$ are disjoint; thus, clearly $F_1$ and $F_2$ are also disjoint), then there are NFA for $L_1 \cup L_2$, $L_1 \circ L_2$, and $L_1^*$. 
The new NFA has an $\epsilon$-arrow from its initial state $p_0$ to $p_1$ and to $p_2$.
Its final state set is $F_1 \cup F_2$. 

\begin{center}
\begin{tikzpicture}
    \node[state, initial] (p0) at (0, 0) {$p_0$};
    \node[state, accepting] (p1) at (1, 1) {$p_1$};
    \node[state, accepting] (p2) at (1, -1) {$p_2$};
    \path[->, bend left=30]
    (p0) edge node {$\epsilon$} (p1);
    \path[->, bend left=30]
    (p0) edge node {$\epsilon$} (p2);
\end{tikzpicture}
\end{center}
**Concatenation**

The state set of the new NFA is the union of the state sets of $N_1$ and $N_2$.

The initial state is $p_1$.

There is an $\epsilon$-arrow from each $s \in F_1$ to $p_2$.

The final state set $F_0$ is $F_2$. 

![Diagram showing states and transitions of two NFAs concatenated]
Introduce a new initial state $p_0$.
Add an $\epsilon$-arrow from $p_0$ to $p_1$.
Add an $\epsilon$-arrow from each $s \in F_1$ to $p_0$.
The final state set is \{\{p_0\}\}. 

\[ L_1^* \]
Regular languages can be expressed as regular expressions.

A general nondeterministic finite automaton (GNFA) is a kind of NFA such that:

• There is a unique start state and is a unique accept state.
• Every pair of nodes is connected by an arrow in each direction, each labeled with a regular expression. Exceptions are:
  • The start state has no incoming edges.
  • The accept state has no outgoing edges.

The language accepted by the GNFA is the union of all $L(R)$ such that $R$ is the regular expression constructed by concatenating all the regular expressions appearing on the path from the start state to the accept state in the order they appear.
An Example of GNFA

Here the dashed arrows represent arrows with ∅ as the label.

(Quite a few dotted arrows aren’t drawn here (and on the coming slides) but really should be. For example, there should be dotted arrows from the far left state to each of the 3 rightmost states, and 4 more backward dotted arrows along the upper and lower segments. GNFAs are painful to draw in full correctly!)
And our woes are not over yet. Is there some problem in this drawing? I claim that two of the edges have illegal labels. Can you spot those two? And what should those labels be replaced with (remember, labels should be regular expressions, and “a, b” is not a regular expression)? (The rest of this example still uses “a, b,” but each such use is a formal error! So please do as we say, not as we (incorrectly) do (here)!)
Proof Plan

1. Show that any NFA can be converted to an equivalent GNFA.
2. Show that any GNFA can be converted to an equivalent regular expression.
From an NFA to a GNFA

Given an NFA $N$, construct a GNFA $G$ as follows:

1. Add a special start state $q_{\text{start}}$ and connect it to the initial state of $N$ with $\epsilon$ as the label. Connect $q_{\text{start}}$ to each remaining state with $\emptyset$ as the label.
From an NFA to a GNFA

Given an NFA $N$, construct a GNFA $G$ as follows:

1. Add a special start state $q_{\text{start}}$ and connect it to the initial state of $N$ with $\epsilon$ as the label. Connect $q_{\text{start}}$ to each remaining state with $\emptyset$ as the label.

2. Add a special accept state $q_{\text{accept}}$ and connect to it from each final state of $N$ with $\epsilon$ as the label. Connect from each of the remaining state of $N$ to $q_{\text{accept}}$ with $\emptyset$ as the label.
From an NFA to a GNFA

Given an NFA $N$, construct a GNFA $G$ as follows:

1. Add a special start state $q_{start}$ and connect it to the initial state of $N$ with $\epsilon$ as the label. Connect $q_{start}$ to each remaining state with $\emptyset$ as the label.

2. Add a special accept state $q_{accept}$ and connect to it from each final state of $N$ with $\epsilon$ as the label. Connect from each of the remaining state of $N$ to $q_{accept}$ with $\emptyset$ as the label.

3. Add an arrow with $\emptyset$ as the label wherever necessary.
(Aside: What is missing here? Hint: We need to know the start state of this NFA?)
Before and After

(Can you see why $q_{\text{start}}$ does not need to have a “this is the start state” arrow?)

(Dashed-arrow self-loops labeled with $\emptyset$ are, except for gray states, there (but not drawn), here and in the rest of Ch. 1, Pt. 3; and various other dotted-arrow omissions exist as mentioned earlier.)
What Have We Done?

We have constructed our initial GNFA $G$ so that:

(I) For each word $w$ in $L(N)$, there is a path $[u_0, \ldots, u_m]$ in $G$ from $q_{\text{start}}$ to $q_{\text{accept}}$ such that
   (a) $w$ is decomposed as $w_1w_2\cdots w_m$;
   (b) for all $i$, $1 \leq i \leq m$, $w_i$ is a member of the language represented by the regular expression of the arrow $(u_{i-1}, u_i)$. 
What Have We Done?

We have constructed our initial GNFA $G$ so that:

(I) For each word $w$ in $L(N)$, there is a path $[u_0, \ldots, u_m]$ in $G$ from $q_{\text{start}}$ to $q_{\text{accept}}$ such that
   (a) $w$ is decomposed as $w_1w_2\cdots w_m$;
   (b) for all $i$, $1 \leq i \leq m$, $w_i$ is a member of the language represented by the regular expression of the arrow $(u_{i-1}, u_i)$.

(II) For each word $w$ that can be decomposed as in the above, $w$ is a member of $L(N)$.
Convert a GNFA to a Regular Expression

We will remove intermediate nodes one after the other while preserving the two properties.

Repeat the following until there is no state left other than the start state and the accept state.

- Select an arbitrary intermediate state \( s \).
- Produce from the current GNFA an equivalent GNFA by:
  - For each arrow \((p, q)\) such that \( p, q \neq s \), replace its label with the label corresponding to the paths \([p, q], [p, s, q], [p, s, s, q], [p, s, s, s, q], \ldots\). (The cases where \( p = q \) are quite important, and should not be forgotten; that is a very easy, common mistake to make; I make it often!)
  - Remove \( s \) and all edges attached to it.
The Procedure

Let $G$ be the current GNFA. Replace $G$ with a new GNFA $G'$ constructed from $G$ as follows:

- $G'$ has all states of $G$ except for $s$.
- For each pair $(p, q)$ of states, $p, q, \neq s$, label the arrow $(p, q)$ of $G'$ by $u \cup xy^*z$, where
  - $u$ is the label of $(p, q)$ in $G$,
  - $x$ is the label of $(p, s)$ in $G$,
  - $y$ is the label of $(s, s)$ in $G$,
  - $z$ is the label of $(s, q)$ in $G$,

**Side note:** since $A\emptyset = \emptyset A = \emptyset$ and $\emptyset^* = \{\epsilon\}$, exactly the right things happen even for things with $\emptyset$ labels; also, we use the just-mentioned equalities to simplify labels on the fly.

The two properties (I) and (II) are preserved.
State Elimination

becomes
State Elimination

becomes

CSC280, Chapter 1, Part 3 © 2012 Mitsunori Ogihara. Edits/updates by Lane A. Hemaspaandra, 20{13,14,17}. 26
State Elimination

becomes
State Elimination

becomes

CSC280, Chapter 1, Part 3 © 2012 Mitsunori Ogihara. Edits/updates by Lane A. Hemaspaandra, 20{13,14,17}.

28
State Elimination

becomes

\[
\text{State Elimination}
\]

\[
\text{becomes}
\]

\[
\text{becomes}
\]
State Elimination

becomes

\[ \varepsilon \text{ becomes } aa\{a,b\}^* \cup bb\{a,b\}^* \]
State Elimination

\[ \varepsilon \xrightarrow{aa\{a,b\}\ast \cup bb\{a,b\}\ast} \]

becomes

\[ aa\{a,b\}\ast \cup bb\{a,b\}\ast \]
State Elimination

\[ aa(a,b)^* \cup bb(a,b)^* \]

becomes

\[ aa(a,b)^* \cup bb(a,b)^* \]

The conversion has been completed.
The Regular Expression Is Not Necessarily Unique

Note that the regular expression is dependent on the order in which the states are eliminated and thus may not be unique.
Closure Properties of Regular Languages

Theorem. The regular languages are closed under complement, union, intersection, concatenation, and star.

Proof The closure properties under union, concatenation, and star follow from the fact that the regular languages are those that are expressible with regular expressions.

For complement, note that the complement of a language accepted by a DFA \((Q, \Sigma, \delta, p_0, F)\) is accepted by a DFA \((Q, \Sigma, \delta, p_0, Q \setminus F)\).

For intersection, use DeMorgan’s Law.
Nonregular Languages

How can we show that a language is not regular?
The Pumping Lemma

Theorem. (Pumping Lemma) Let $L$ be an arbitrary regular language. Then there exists a positive integer $p$ with the following property:

Given an arbitrary member $w$ of $L$ having length at least $p$ (i.e., $|w| \geq p$), $w$ can be divided into three parts, $w = xyz$, such that

- $|y| \geq 1$ (the middle part is nonempty),
- $|xy| \leq p$ (the first two parts together have length at most $p$), and
- for each $i \geq 0$, $xy^iz \in L$ (removing or repeating the middle part produces members of $L$).
Proof of the Pumping Lemma

Let $L$ be an arbitrary regular language. Then there is an FA, say $M$, that decides $L$. Let $p$ be the number of states of $M$.

Let $w$ be an arbitrary member of $L$ having length $n$ with $n \geq p$. (If $L$ has no strings of length greater or equal to $n$, then the theorem is clearly satisfied and we are done.)

Let $q_0, q_1, \ldots, q_n$ be the states that $M$ on input $w$. That is, for each $i$, after reading the first $i$ symbols of $w$, $M$ is at $q_i$.

Clearly, $q_0$ is the initial state of $M$. Also, because $w \in L$, $q_n$ is a final state of $M$. 


The Pigeonhole Principle

We are placing a number of pigeons in a number of holes.

If there are more pigeons than there are holes, at least one hole must host more than one pigeon.
The Pigeonhole Principle

We are placing a number of pigeons in a number of holes.

If there are more pigeons than there are holes, at least one hole should host more than one pigeon.

Consider \( q_0, \ldots, q_p \) (the first \( p + 1 \) states that \( M \) goes through on input \( w \)). By the pigeonhole principle, there exist \( c \) and \( d \), \( 0 \leq c < d \leq p \), such that \( q_c = q_d \).

Pick an arbitrary such pair \( (c, d) \).
Proof of the Pumping Lemma (cont’d)

Let \( x = w_1 \cdots w_c, y = w_{c+1} \cdots w_d, \) and \( z = w_{d+1} \cdots w_n. \) Then

- \( |y| \geq 1, \)
- \( |xy| \leq p, \)
- \( M \) transitions from \( q_0 \) to \( q_c \) on \( x, \)
- \( M \) transitions from \( q_c \) to \( q_c \) on \( y, \)
- \( M \) transitions from \( q_c \) to \( q_n \) on \( z. \)

Thus, for every \( i \geq 0, \) \( M \) transitions from \( q_0 \) to \( q_n \) on \( xy^i z, \) and so \( xy^i z \) is a member of \( L. \)
Application of the Pumping Lemma

Example 1: \( B = \{0^n1^n \mid n \geq 0\} \) is not regular.
Application of the Pumping Lemma

Example 1: $B = \{0^n1^n \mid n \geq 0\}$ is not regular.

Proof Assume that $B$ is regular. Let $p$ be a constant from the pumping lemma for $B$. Let $w = 0^p1^p$. 
Application of the Pumping Lemma

Example 1: \( B = \{0^n1^n \mid n \geq 0\} \) is not regular.

Proof Assume that \( B \) is regular. Let \( p \) be a constant from the pumping lemma for \( B \). Let \( w = 0^p1^p \).

Then \( w \) is in \( B \) so it can be divided into \( w = xyz \) such that

- \(|y| \geq 1\),
- \(|xy| \leq p\), and
- for each \( i \geq 0 \), \( xy^iz \in B \).
Application of the Pumping Lemma

Example 1: \( B = \{0^n1^n \mid n \geq 0\} \) is not regular.

Proof  Assume that \( B \) is regular. Let \( p \) be a constant from the pumping lemma for \( B \). Let \( w = 0^p1^p \).

Then \( w \) is in \( B \) so it can be divided into \( w = xyz \) such that

- \( |y| \geq 1 \),
- \( |xy| \leq p \), and
- for each \( i \geq 0 \), \( xy^iz \in B \).

Since \( |xy| \leq p \), both \( x \) and \( y \) consist solely of 0s. The word \( xy^2y \) (the \( i = 2 \) case) has more 0s than 1s, and thus, not in \( B \). However, by the pumping lemma, \( xy^2y \in B \), a contradiction. Hence, \( B \) is not regular.
Illustrating Conversation

I think "0^n1^n" is regular...
I think "0^n1^n" is regular...

You're wrong, and here's why. Assuming it's regular, how many states do you need to build an FA for it?
Illustrating Conversation

Well, it must be large, but I think it should be less than a trillion...
Illustrating Conversation

Well, it must be large, but I think it should be less than a trillion...

Let $p$ be the number and $w=0^p1^p$. The Pumping Lemma divides this into $xyz$. What is $y$?
Illustrating Conversation

Well, $y = 0^k$ for some positive $k$...
Illustrating Conversation

Well, \( y = 0^k \) for some positive \( k \)...

Then \( xz \) must be a member, but it has fewer 0s than 1s, so it can’t be. We thus have a contradiction.
Illustrating Conversation

Oh, I was so naive. I should have taken CSC280...

It's not too late.
Example 2

$C = \{ w \mid w \in \{0, 1\}^* \text{ and has an equal number of 0s and 1s } \}$ is not regular.
Example 2

\[ C = \{ w \mid w \in \{0, 1\}^* \text{ and has an equal number of 0s and 1s } \} \text{ is not regular.} \]

**Proof** Assume, to the contrary, that \( C \) is regular. Let \( p \) be the constant from the pumping lemma for \( C \).
Example 2

$C = \{ w \mid w \in \{0,1\}^* \text{ and has an equal number of 0s and 1s } \}$ is not regular.

Proof Assume, to the contrary, that $C$ is regular. Let $p$ be the constant from the pumping lemma for $C$.

Let $w = 0^p1^p$. Then $w = xyz$ such that $|xy| \leq p$, $|y| \geq 1$, and for every $i \geq 0$, $xy^iz \in C$. 
Example 2

$C = \{w \mid w \in \{0, 1\}^* \text{ and has an equal number of 0s and 1s } \}$ is not regular.

**Proof**  Assume, to the contrary, that $C$ is regular. Let $p$ be the constant from the pumping lemma for $C$.

Let $w = 0^p1^p$. Then $w = xyz$ such that $|xy| \leq p$, $|y| \geq 1$, and for every $i \geq 0$, $xy^iz \in C$.

Let $w' = xz$. Then $w' \in C$ but $w'$ has fewer 0s than 1s.
Example 3

The language $F = \{ vv \mid v \in \{0, 1\}^* \}$ is not regular ($F$ is the language of all even length strings over $\{0, 1\}$ whose first half is identical to the second half).

Proof Assume, to the contrary, that $F$ is regular. Let $p$ be a constant for which the pumping lemma holds for $F$. 
Example 3

The language $F = \{vv \mid v \in \{0, 1\}^*\}$ is not regular ($F$ is the language of all even length strings over $\{0, 1\}$ whose first half is identical to the second half).

Proof Assume, to the contrary, that $F$ is regular. Let $p$ be a constant for which the pumping lemma holds for $F$.

Let $w = 0^p1^p0^p1^p$. Then, $w$ is divided into $w = xyz$ such that $|y| \geq 1$, $|xy| \leq p$, and $(\forall i \geq 0)[xy^iz \in F]$. Here $y \in 0^+$ since $w$ begins with $0^p$. 
Example 3

The language $F = \{vv \mid v \in \{0, 1\}^*\}$ is not regular ($F$ is the language of all even length strings over $\{0, 1\}$ whose first half is identical to the second half).

Proof  Assume, to the contrary, that $F$ is regular. Let $p$ be a constant for which the pumping lemma holds for $F$.

Let $w = 0^p1^p0^p1^p$. Then, $w$ is divided into $w = xyz$ such that $|y| > 0$, $|xy| \leq p$, and $(\forall i \geq 0)[xy^iz \in F]$. Here $y \in 0^+$ since $w$ begins with $0^p$.

Pick $i = 0$, we have $0^q1^p0^p1^p \in F$, where $q < p$. This word cannot be decomposed as $uu$. This is a contradiction.
Example 4

\[ D = \{1^{n^2} \mid n \geq 0\} \text{ is not regular.} \]
Example 4

\[ D = \{1^{n^2} \mid n \geq 0\} \text{ is not regular.} \]

**Proof** Assume, to the contrary, that \( D \) is regular. Let \( p \) be a constant for which the pumping lemma holds for \( D \).
Example 4

\[ D = \{1^{n^2} \mid n \geq 0\} \text{ is not regular.} \]

**Proof**  Assume, to the contrary, that \( D \) is regular. Let \( p \) be a constant for which the pumping lemma holds for \( D \).

Let \( w = 1^{p^2} \). Then \( w = xyz \) for some \( x, y, z \) such that \( |y| \geq 1 \), \( |xy| \leq p \), and \( (\forall i \geq 0)[xy^iz \in D] \).
Example 4

\[ D = \{ 1^{n^2} \mid n \geq 0 \} \] is not regular.

**Proof** Assume, to the contrary, that \( D \) is regular. Let \( p \) be a constant for which the pumping lemma holds for \( D \).

Let \( w = 1^p^2 \). Then \( w = xyz \) for some \( x, y, z \) such that \( |y| \geq 1 \), \( |xy| \leq p \), and \((\forall i \geq 0)[xy^iz \in D]\).

Let \( l = |y| \). Then \( 1 \leq l \leq p \). By plugging in \( i = 2 \), we have \( 1^p^2 + l \in D \), but \( p^2 + l \leq p^2 + p < (p + 1)^2 \), a contradiction. \( \blacksquare \)
Example 5

\[ E = \{0^i1^j \mid i > j\} \] is not regular.
**Example 5**

\[ E = \{0^i1^j \mid i > j\} \] is not regular.

**Proof**  Assume, to the contrary, that \( E \) is regular. Let \( p \) be a constant for which the pumping lemma holds for \( E \).
Example 5

\[ E = \{0^i1^j \mid i > j\} \] is not regular.

**Proof**  Assume, to the contrary, that \( E \) is regular. Let \( p \) be a constant for which the pumping lemma holds for \( E \).

Let \( w = 0^p1^{p-1} \). Then \( w = xyz \) for some \( x, y, z \) such that \( |y| \geq 1 \), \( |xy| \leq p \), and \( (\forall i \geq 0)[xy^iz \in E] \). Here \( y \in 0^+ \) since the first \( p \) symbols of \( w \) are all 0.
Example 5

\[ E = \{0^i1^j \mid i > j\} \] is not regular.

**Proof** Assume, to the contrary, that \( E \) is regular. Let \( p \) be a constant for which the pumping lemma holds for \( E \).

Let \( w = 0^p1^{p-1} \). Then \( w = xyz \) for some \( x, y, z \) such that \( |y| \geq 1 \), \( |xy| \leq p \), and \( (\forall i \geq 0)[xy^i z \in E] \). Here \( y \in 0^+ \) since the first \( p \) symbols of \( w \) are all 0.

With \( i = 0 \), we have \( 0^q1^{p-1} \in E \), where \( q \leq p - 1 \), a contradiction.
Chapter 2, Part 1

Context-free Languages
A **context-free grammar** is a 4-tuple $G = (V, \Sigma, R, S)$. Here

1. $V$ is the set of **variables** (or **nonterminals**),
2. $\Sigma$ is the set of **terminals**,
3. $R$ is the set of **substitution rules** (or **production rules**), each of which is of the form
   
   $$A \rightarrow w,$$

   for some nonterminal $A$ and some word $w$ over $V \cup \Sigma$; and
4. $S$ is a nonterminal called the **start symbol**.
Substitution

Given a word $x \in (V \cup \Sigma)^*$ of the form $yAz$ and a rule $A \rightarrow w$, $x$ can be turned into $ywz$ by substituting the $A$ with $w$.

Note

- If $A$ does not appear in $x$, the rule has no effect on $x$.
- If there are multiple rules for substituting $A$, then you nondeterministically choose the one you apply.
- If there are multiple occurrences of $A$, then you nondeterministically choose the one and the rule is applied.
**Derivation**

We write $u \Rightarrow v$ to mean that $v$ can be produced from $u$ by applying in sequence production rules; that is, if there is a sequence $[u_0, \ldots, u_m]$ of strings over $V \cup \Sigma$ and there is a sequence $[r_1, \ldots, r_m]$ of rules $R$ such that

- $u_0 = u$ and $u_m = v$;
- for each $i$, $1 \leq i \leq m$, $u_i$ can be obtained from $u_{i-1}$ by applying $r_i$.

We say that $G$ produces $w \in \Sigma^*$ if $S \Rightarrow w$, i.e., $w$ can be obtained from $S$ by derivation.
A **parse tree** (or **derivation tree**) is a tree that depicts the process of derivation.

Since each derivation step substitutes one nonterminal, the series of substitutions can be visualized using a tree.
Example

The strings over $\Sigma = \{a, b\}$ consisting of an equal number of a’s and b’s.
Example: A Tempting but Incorrect Grammar for This!

The strings over $\Sigma = \{a, b\}$ consisting of an equal number of a’s and b’s.

$V = \{S\}$ and the derivation rules are

$S \rightarrow \epsilon | abS | baS | aSb | bSa | S'ab | S'ba.$

Are all the strings this generates good?

Does this generate all strings in the set? Can you find a counterexample?
Example: A Tempting but Incorrect Grammar for This!

The strings over $\Sigma = \{a, b\}$ consisting of an equal number of a’s and b’s.

$V = \{S\}$ and the derivation rules are

$S \rightarrow \epsilon | abS | baS | aSb | bSa | S'ab | S'ba.$

Are all the strings this generates good? Yes!

Does this generate all strings in the set? Can you find a counterexample? Here is one:

Can this grammar ever yield a string that starts and ends in a? Yes it can! Oops—but let’s not give up! So, can this grammar ever yield a string that starts and ends in aa, such as aabbbbaa? No! Why? What could the first rule be?!? None work!
Example: Two Good Grammars

The strings over $\Sigma = \{a, b\}$ consisting of an equal number of a’s and b’s.

Grammar 1 that works for this:
$V = \{S\}$ and the derivation rules are $S \rightarrow \epsilon \mid aSbS \mid bSaS$.

Grammar 2 that works for this:
$V = \{S\}$ and the derivation rules are $S \rightarrow \epsilon \mid aSb \mid bSa \mid SS$. 
Example

The strings over $\Sigma = \{a, b\}$ consisting of an equal number of a’s and b’s.

$V = \{S\}$ and the derivation rules are $S \rightarrow \epsilon | aSbS | bSaS$.

$abab$ is derived as follows (this is an example using Grammar 1):

$$
S \Rightarrow aSbS \Rightarrow abSaSbS \Rightarrow abSabbS \Rightarrow ababS \Rightarrow abab.
$$
Why Does the Grammar 2 Work?

Grammar 2: \( V = \{ S \} \) and the derivation rules are
\[
S \rightarrow \varepsilon | aSb | bSa | SS.
\]

Everything grammar 2 generates has the same number of a’s and b’s.

Consider any nonempty (even-length) string of a’s and b’s with the same number of a’s and b’s. If it is \( \varepsilon \), clearly this grammar gives it. If it is a...b or b...a, strip the inside thanks to the aSb or bSa productions, and inductively we’re done. If it is a...a (and b...b is analogous), then there must be (why?) some break-up of the string into two nonempty parts (e.g., breaking aabbabbbabbbba into aabb \cdot abbbabbbba) that each have the same number of a’s and b’s, and so thanks to the SS production inductively (twice, on those smaller strings) we’re again done.
Why Does the Grammar 1 Work?

Let $L$ be the language. For each word over $\{a, b\}$, let $d(w)$ be the number of a’s in $w$ minus the number of b’s in $w$. We observe:

- For all $w$, $w \in L$ if and only if $d(w) = 0$.
- $d(\epsilon) = 0$; $d(a) = 1$; $d(b) = -1$.
- For all $u$ and $v$, $d(uv) = d(u) + d(v)$.

So, if $w \in L$ and if $w_1 = a$, there exists some $k \geq 2$ such that $d(w_1 \cdots w_k) = 0$ and $d(w_1 \cdots w_{k-1}) = d(w_1) = 1$. This implies that $d(w_2 \cdots w_{k-1}) = d(w_{k+1} \cdots w_n) = 0$.

This gives $S \rightarrow aSbS$. By exchanging the role between $a$ and $b$, we have $S \rightarrow bSaS$. 
Ambiguity and Leftmost Derivation

It looks like there are many different ways to produce the same word \( w \in L(G) \) for a grammar \( G \).
Ambiguity and Leftmost Derivation

It looks like there are many different ways to produce the same word $w \in L(G)$ for a grammar $G$.

Of course, this is true because when multiple nonterminals appear on an intermediate word, the order in which the nonterminals are chosen for substitution doesn’t affect the word produced.
Ambiguity and Leftmost Derivation

It looks like there are many different ways to produce the same word $w \in L(G)$ for a grammar $G$.

Of course, this is true because when multiple nonterminals appear on an intermediate word, the order in which the nonterminals are chosen for substitution doesn’t affect the word produced.

But what if you force the order to be always from left to right, will there still be multiple ways to derive the target word?
Leftmost Derivation

A **leftmost derivation** is the derivation in which each production rule is applied to the leftmost nonterminal at the moment.
A **leftmost derivation** is the derivation in which each production rule is applied to the leftmost nonterminal at the moment. For $abab$ in the previous example,

$$S \Rightarrow aS' b S \Rightarrow abS' a S' b S \Rightarrow abaS' b S \Rightarrow ababS \Rightarrow abab$$

is a leftmost derivation,
Leftmost Derivation

A *leftmost derivation* is the derivation in which each production rule is applied to the leftmost nonterminal at the moment. For $abab$ in the previous example,

$$
S \Rightarrow aS'bS \Rightarrow abSaS'bS \Rightarrow abaS'bS \Rightarrow ababS \Rightarrow abab
$$

is a leftmost derivation, while

$$
S \Rightarrow aS'bS \Rightarrow aS'baS'bS \Rightarrow aS'babS \Rightarrow aS'bab \Rightarrow abab
$$

isn’t one.
Ambiguity and Leftmost Derivation

A context-free grammar is **unambiguous** if it has a unique leftmost derivation for every word it generates. Otherwise, the grammar is **ambiguous**.

There is a context-free language that is **inherently ambiguous** — every grammar that produces the language is ambiguous.
A context-free grammar $G = (V, \Sigma, R, S)$ is in **Chomsky normal form** if each rule in $R$ is of the following form:

- $S \rightarrow \epsilon$ (note that $S$ is the start symbol).
- $A \rightarrow BC$ for some $A \in V$ and $B, C \in V - \{S\}$ and
- $A \rightarrow a$ for some $a \in \Sigma$. 

**Chomsky Normal Form**

A context-free grammar $G = (V, \Sigma, R, S)$ is in **Chomsky normal form** if each rule in $R$ is of the following form:

- $S \rightarrow \epsilon$ (note that $S$ is the start symbol).
- $A \rightarrow BC$ for some $A \in V$ and $B, C \in V - \{S\}$ and
- $A \rightarrow a$ for some $a \in \Sigma$. 

Chomsky Normal Form

A context-free grammar $G = (V, \Sigma, R, S)$ is in **Chomsky normal form** if each rule in $R$ is of the following form:

- $S \rightarrow \epsilon$ (note that $S$ is the start symbol).
- $A \rightarrow BC$ for some $A \in V$ and $B, C \in V - \{S\}$ and
- $A \rightarrow a$ for some $a \in \Sigma$.

**Theorem.** Each context-free language is generated by a Chomsky normal form grammar.
Converting an Arbitrary CFG to a CNF Grammar

Let $G = (V, \Sigma, R, S)$ be an arbitrary CFG and let $L = L(G)$.

We will convert this to a CNF grammar $G' = (V', \Sigma, R', S_0)$. 
Conversion Policy

We perform the following steps in order:

1. Do away with rules of the form $A \rightarrow \epsilon$.
   This rule is permitted for only $A = S_0$ only in the case where $\epsilon \in L$.
2. Do away with rules of the form $A \rightarrow B$ where $B$ is a variable.
3. Do away with rules of the form $A \rightarrow w$ where either $w$ is having length 2 and contains a terminal or $w$ is having length $\geq 3$. 
Step 1: Rules of the form $A \rightarrow \epsilon$

We need to eliminate all rules of the form $A \rightarrow \epsilon$.

We say that a variable $A$ of $G$ is **nullable** if $A \Rightarrow \epsilon$. 
Finding All “Nullable” Variables

We find all “nullable” variables using the following greedy algorithm:

- Initialize a set $U$ as the collection of all variables $A$ such that $A \rightarrow \epsilon$ is a valid production rule.
- While there is a rule of the form $B \rightarrow A_1 \cdots A_k$ such that $B \notin U$ and $A_1, \ldots, A_k \in U$, select such a $B$ add it to $U$. 
Does $\epsilon$ Belong to $L(G)$?

After $U$ has been computed, whether $\epsilon \in L(G)$ can be tested by checking whether $S \in U$ because:

$$S \in U \iff \epsilon \in L(G).$$

If $S \in U$, we will add $S_0 \rightarrow \epsilon$ at the end.

From this point on we will assume that $\epsilon \notin L(G)$. 
Removable of $A \rightarrow \epsilon$

- Initialize $V'$ with the set $V$.
- Add to $R'$ all the rules in $R$ of the form $B \rightarrow y$ such that $y$ is nonempty, is not equal to $B$, and has no nullable variables.
- Add $S_0$ to $V'$ and rule $S_0 \rightarrow S$ to $R'$. 
For each rule of the form $B \rightarrow y$ in $R$ such that a nullable variable appears in $y$ do the following:

- Create all rules produced from $B \rightarrow y$ by selecting, independently at each position in $y$ where the letter is a nullable variable, whether to keep the variable in place or replace it with $\epsilon$.

- Add all the rules thus created (which includes the original $B \rightarrow y$) to $R'$, except for $B \rightarrow \epsilon$ and $B \rightarrow B$ if such a rule is at all created.
**Removable of** $A \rightarrow B$

While $R'$ contains a rule of the form $A \rightarrow B$ such that $B \in V$, pick such a rule $r$ and do the following:

- Remove $r$ from $R'$.
- For each rule $B \rightarrow w$ in $R'$, add $A \rightarrow w$ to $R'$ if $w \neq A$. 
Each rule in $R'$ is one of the following forms:

- $A \rightarrow b$ for some $b \in \Sigma$.
- $A \rightarrow w$ for some $w \in (V \cup \Sigma)^*$ having length $\geq 2$. 
Normalization Part 1

For each terminal $d$

- add a new variable $D$,
- add a new rule $D \rightarrow d$, and
- for each rule $A \rightarrow u$ such that $|u| \geq 2$ and $d$ appears in $u$, replace each occurrence of $d$ with a $D$. 
Normalization Part 2

For each rule $A \rightarrow w_1 \ldots w_m$ such that $m \geq 3$, do the following:

- Add a new variable $X$.
- Replace $A \rightarrow w$ by two rules: $A \rightarrow w_1X$ and $X \rightarrow w_2 \ldots w_m$. 
Example

\[ V = \{S\}, \Sigma = \{a, b\}, \text{ and } R \text{ consists of } S \rightarrow \epsilon | aSbS | bSaS \]

**Step 1** Add \( S_0 \rightarrow S | \epsilon \).

**Step 2** Eliminate \( S \rightarrow \epsilon \). The rules are

\[
S_0 \rightarrow S | \epsilon, \\
S \rightarrow ab | abS | aSbS | aSb | ba | baS | bSaS | bSa.
\]

**STEP 3** Eliminate \( S_0 \rightarrow S \) and add

\[
S_0 \rightarrow ab | abS | aSbS | aSb | ba | baS | bSaS | bSa
\]


**Example (cont’d)**

STEP 4 The rules are

\[
S_0 \rightarrow \epsilon, \quad A \rightarrow a, \quad B \rightarrow b, \\
S_0 \rightarrow AB \mid AX_1 \mid AX_2 \mid AX_3 \mid BA \mid BY_1 \mid BY_2 \mid BY_3, \\
S \rightarrow AB \mid AX_1 \mid AX_2 \mid AX_3 \mid BA \mid BY_1 \mid BY_2 \mid BY_3, \\
X_1 \rightarrow BS, \quad X_2 \rightarrow SX_4, \\
X_3 \rightarrow SB, \quad X_4 \rightarrow BS, \\
Y_1 \rightarrow AS, \quad Y_2 \rightarrow SY_4, \\
Y_3 \rightarrow SA, \quad Y_4 \rightarrow AS.
\]

Here we are using the same variables \(X_1, \ldots, X_4\) and \(Y_1, \ldots, Y_3\) for \(S_0\) and \(S\).
Pushdown Automata

The machine model for the context-free language.
A pushdown automaton is an NFA with a last-in, first-out storage device called \textit{stack}.
A pushdown automaton is an NFA with a last-in, first-out storage device called \textit{stack}.

A \textbf{pushdown automaton} is a 6-tuple \((Q, \Sigma, \Gamma, \delta, q_0, F)\) where

1. \(Q\) is a finite \textit{set of states},
2. \(\Sigma\) is a finite \textit{input alphabet},
3. \(\Gamma\) is a finite \textit{stack alphabet},
4. \(\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \rightarrow \mathcal{P}(Q \times \Gamma_\epsilon)\) is the \textit{transition},
5. \(q_0 \in Q\) is the \textit{start state}, and
6. \(F \subseteq Q\) is the \textit{set of accept states},

where \(\Sigma_\epsilon = \Sigma \cup \{\epsilon\}\) and \(\Gamma_\epsilon = \Gamma \cup \{\epsilon\}\).
Represent the contents of the stack by taking the letters from top to bottom and putting them from left to right. Here the stack has the word *tsts* (see: *t* is at the top of the stack in the picture, and so is leftmost in *tsts*).
Computation by Pushdown Automata

(A) If not the entire input has been read, it may choose to **read the next letter**.
- If either the entire input has already been read or it decides not to read the next letter, the input letter is considered to be $\varepsilon$. 

Computation by Pushdown Automata

(B) It may choose to **read the top letter** of the stack.
- If the stack is already empty, then the computation stops there without accepting.
- If it decides not to read the stack, the stack letter is considered to be $\epsilon$. 

CSC280, Chapter 2, Part 2 © 2012 Mitsunori Ogihara. Edits/updates by Lane A. Hemaspaandra, 20{13,14,17}. 6
(C) Depending on the current state, the input letter, and the stack letter, it **nondeterministically decides the next state and a letter to be placed on the stack**, with a possible option of not placing a letter, in which case the letter is considered to be \(\epsilon\).
Transition Function of a PDA

- The input: $Q \times \Sigma_\epsilon \times \Gamma_\epsilon$.
- The output: $2Q \times \Gamma_\epsilon$. 
Acceptance of Pushdown Automata

A pushdown automaton $M$ accepts an input $w$ if $M$ arrives at an accept state at some time after reading all the input letters.

A computation path of $M$ on input $w$ halts without accepting if there is no applicable next move.

If the current state is $q$, the next input letter is $a$, and the top stack letter is $b$, there are four possible courses of action:

1. a move in $\delta(q, \epsilon, \epsilon)$
2. a move in $\delta(q, \epsilon, b)$
3. a move in $\delta(q, a, \epsilon)$
4. a move in $\delta(q, a, b)$
Acceptance of Pushdown Automata

A pushdown automaton $M$ accepts an input $w$ if $M$ arrives at an accept state at some time after reading all the input letters.

A computation path of $M$ on input $w$ halts without accepting if there is no applicable next move.

If the current state is $q$, the next input letter is $a$, and the top stack letter is $b$, there are four possible courses of action:

1. a move in $\delta(q, \epsilon, \epsilon)$
2. a move in $\delta(q, \epsilon, b)$ – impossible if stack is empty
3. a move in $\delta(q, a, \epsilon)$ – impossible if no input letter is left
4. a move in $\delta(q, a, b)$ – impossible if no input letter is left or stack is empty
Acceptance by a Pushdown Automaton

Formally, $M$ accepts $w \in \Sigma^*$ if there exist

- $r_0, \ldots, r_m \in Q$ (states), $w_1, \ldots, w_m \in \Sigma_\epsilon$ (input letters),
- $a_1, \ldots, a_m \in \Gamma_\epsilon$ (stack letters read),
- $b_1, \ldots, b_m \in \Gamma_\epsilon$ (stack letters written),
- $s_1, \ldots, s_m \in \Gamma^*$ (stack word before pop),
- $t_1, \ldots, t_m \in \Gamma^*$ (stack word after pop), such that:

1. $r_0 = q_0$, $r_m \in F$ (start with $q_0$ and accept),
2. $w = w_1 \cdots w_m$ (the input decomposition),
3. $a_1 = s_1 = t_1 = \epsilon$ (start with empty stack),
4. for all $i, 1 \leq i \leq m - 1$, $s_i = a_i t_i$ and $s_{i+1} = b_i t_i$ (preservation of stack content other than top),
5. for all $i, 1 \leq i \leq m$, $(r_i, b_i) \in \delta(r_{i-1}, w_i, a_i)$ (respecting the transition function).
Diagram

Input \[ \cdots \quad w_i \quad \cdots \quad w_{i+1} \]

\[ \delta(r_{i-1}, w_i, a_i) \ni (r_i, b_i) \]

State \[ r_0 \quad \cdots \quad r_{i-1} \quad r_i \quad \cdots \quad r_m \]

Stack

\[ \begin{array}{c}
  a_i \\
  t_i \\
  s_i \\
\end{array} \quad \begin{array}{c}
  b_i \\
  t_i \\
  s_{i+1} \\
\end{array} \]
Example 1

A PDA for $L = \{0^n1^n \mid n \geq 0\}$.

For example, 000111 is a member, 0011 is a member, but 00110011 is not.
Example 1

Idea

Using stack keep a tally of the leading 0s. Compare the number against the number of trailing 1s.
Example 1

Algorithm

1. May accept immediately.
2. Place a special symbol $\$ on the stack to mark the bottom.
3. Read input symbols without popping from stack. If the symbol is a 0, put a 0 onto stack; otherwise, stop.
4. Guess the start of 1s and begin simultaneously reading input and popping from stack.
   If either the input symbol is not a 1 or the stack symbol popped is not a 0, stop.
   Any time during this, stop reading the input and then
     • verify that the top of stack is $\$ and accept.
How This Method Works

• $0^n1^n$ for some $n \geq 1$: Will accept.
• $\epsilon$: Will accept by choosing to verify $\$ \$ immediately after place it on the top.
How This Method Works

- $0^n1^n$ for some $n \geq 1$: Will accept.
- $\epsilon$: Will accept by choosing to verify $\$$ immediately after placing it on the top.
- $1y$ for some $y \in \{0, 1\}^*$: Will either pop the $\$$ on reading the first 1 or enter accept without reading any input letter.
- $0^n1^{n+k}$ for some $n, k \geq 1$: Will either pop the $\$$ on reading the first 1 or enter accept without finishing to read the input.
- $0^n1^{n-k}$ for some $n, k \geq 1$: Verification for $\$$ will fail.
Transition Function

\[ \Gamma = \{0, 1, \$\}, \ Q = \{q_1, q_2, q_3, q_4\}, \ F = \{q_1, q_4\}, \ q_1 \text{ is the initial state} \]

<table>
<thead>
<tr>
<th>Input:</th>
<th>0</th>
<th>1</th>
<th>(\epsilon)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0/$$</td>
<td>0</td>
<td>0</td>
<td>($/\epsilon)</td>
</tr>
<tr>
<td>(q_1)</td>
<td></td>
<td></td>
<td>(q_1)</td>
</tr>
<tr>
<td>(q_2)</td>
<td>((q_2, 0))</td>
<td>((q_3, \epsilon))</td>
<td>((q_2, $))</td>
</tr>
<tr>
<td>(q_3)</td>
<td>((q_3, \epsilon))</td>
<td>((q_3, \epsilon))</td>
<td>((q_4, \epsilon))</td>
</tr>
</tbody>
</table>
Example 1

Let \((q, u, v), q \in Q, u \in \Sigma^*, v \in \Gamma^*,\) denote the configuration where the state is \(q,\) the remaining input is \(u,\) and the stack word is \(v.\)

000111 is accepted by the following path:

\[
(q_1, 000111, \epsilon) \Rightarrow (q_2, 000111, \$) \Rightarrow (q_2, 00111, 0\$) \\
\Rightarrow (q_2, 0111, 00\$) \Rightarrow (q_2, 111, 000\$) \Rightarrow (q_3, 11, 00\$) \\
\Rightarrow (q_3, 1, 0\$) \Rightarrow (q_3, \epsilon, \$) \Rightarrow (q_4, \epsilon, \epsilon). 
\]
Example 2

$L = \{ u \in \{0, 1\}^* \mid u \text{ has the same number of } 0\text{s as } 1\text{s} \}.$

For example, 011100 is a member, 10010101 is a member, and 00010 is a nonmember.
Example 2

$L = \{ u \in \{0, 1\}^* \mid u \text{ has the same number of } 0\text{s as } 1\text{s } \}.$

Idea

Using stack maintain a \textit{tally of the difference between the number of } 0\text{s and the number of } 1\text{s that have been read so far}. Use a tally of 0\text{s if there have been more } 0\text{s than } 1\text{s and a tally of } 1\text{s if there have been more } 1\text{s than } 0\text{s.}

Compare the first letter of the tally and an input letter. If one is a 0 and the other is a 1, they cancel out; otherwise, increase the tally.
Example 2

\[ L = \{ u \in \{0, 1\}^* \mid u \text{ has the same number of 0s as 1s } \}. \]

\[ Q = \{ q_0, q_1, q_2, q_3, q_4 \}, \quad q_4: \text{final}, \quad q_0: \text{initial} \]

No permissible actions in \( q_4 \)

\[ \Gamma = \{0, 1, $\} \]

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
<th>0</th>
<th>1</th>
<th>$</th>
<th>\epsilon</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q_0 )</td>
<td>( \epsilon )</td>
<td></td>
<td></td>
<td></td>
<td>( (q_1, $) )</td>
</tr>
<tr>
<td>( q_1 )</td>
<td>0</td>
<td>( (q_2, 0) )</td>
<td>( (q_1, \epsilon) )</td>
<td>( (q_2, $) )</td>
<td></td>
</tr>
<tr>
<td>( q_1 )</td>
<td>1</td>
<td>( (q_1, \epsilon) )</td>
<td>( (q_3, 1) )</td>
<td>( (q_3, $) )</td>
<td></td>
</tr>
<tr>
<td>( q_2 )</td>
<td>( \epsilon )</td>
<td></td>
<td></td>
<td>( (q_4, \epsilon) )</td>
<td></td>
</tr>
<tr>
<td>( q_3 )</td>
<td>( \epsilon )</td>
<td></td>
<td></td>
<td>( (q_1, 0) )</td>
<td>( (q_1, 1) )</td>
</tr>
</tbody>
</table>

Here \{ and \} are omitted.
But wait! There is a one-character flaw in this machine *drawing* (here and also on the next slide). Can you spot it? (Hint: It has to do with the arrow from $q_1$ to $q_2$; compare the labels on that arrow to the ones on the arrow from $q_1$ to $q_3$; is there a problem regarding the topmost “/0” on that arrow to $q_2$?)
Example: \((q_0, 011100, \epsilon) \Rightarrow (q_1, 011100, \$) \Rightarrow (q_2, 11100, \$) \Rightarrow (q_1, 11100, 0\$) \Rightarrow (q_1, 1100, \$) \Rightarrow (q_3, 100, \$) \Rightarrow (q_1, 100, 1\$) \Rightarrow (q_3, 00, 1\$) \Rightarrow (q_1, 00, 11\$) \Rightarrow (q_1, 0, 1\$) \Rightarrow (q_1, \epsilon, \$) \Rightarrow (q_4, \epsilon, \epsilon)\)
Chapter 2, Part 3

Pushdown Automata and CFLs Are Equivalent
Properties of Context-Free Languages

Theorem. The context-free languages are closed under union, concatenation, and star.

Proof

Let $S_1$ and $S_2$ be the start symbols of two CFG. Let $S_0$ be the new start symbol of the new CFG we are to create. (If needed, first rename things to ensure that the nonterminals of the two CFGs are disjoint, and that $S_0$ is not in either of the nonterminal symbol sets.)

Adding $S_0 \Rightarrow S_1 \mid S_2$ works for union.

Adding $S_0 \Rightarrow S_1S_2$ works for concatenation.

Adding $S_0 \Rightarrow \epsilon \mid S_0S_1$ works for star.
CFLs Capture PDA

Theorem. Every language recognized by PDA is context-free.

Let $L$ be a language recognized by a PDA $M = (Q, \Sigma, \Gamma, \delta, p_0, F)$.

We can modify $M$ so that:

(*) $M$ has a unique accept state and, when it enters the state, the stack is empty.

(**) In a single move $M$ may not both pop and push.
Unique accept state and Empty Stack Acceptance

Modify $M$ to create an equivalent PDA $N = (Q', \Sigma, \Gamma', \delta', q'_0, \{q_f\})$. 
Unique accept state and Empty Stack Acceptance

Modify $M$ to create an equivalent PDA $N = (Q', \Sigma, \Gamma', \delta', q'_0, \{q_f\})$.

$N$ simulates $M$ after adding a new special symbol $\perp$ to the stack. If $M$ enters a accept state, $N$ may choose to empty the stack until it encounters $\perp$, when $N$ may accept.
Unique Accept State and Empty Stack Acceptance

Modify $M$ to create an equivalent PDA $N = (Q', \Sigma, \Gamma', \delta', q'_0, \{q_f\})$.

- $\Gamma' = \Gamma \cup \{\bot\}$.
- $Q'$ consists of:
  - $Q$,
  - a new initial state $q'_0$,
  - a new, unique accept state $q_f$,
  - a clean-up state $C$,
  - some additional states for achieving the “not both push and pop” requirement.
The use of $q'_0$ and $C$ as a Bottom Marker

There is just one move in state $q'_0$: $\delta'(q'_0, \epsilon, \epsilon) = \{(p_0, \bot)\}$.

The transition means: **place a $\bot$ on stack** and then proceed to the initial state of $M$. 
The Role of \( \bot \) and \( C \)

In each accept state \( p \) of \( M \), we add \((C, \epsilon)\) to \( \delta'(p, \epsilon, \epsilon) \).

The transition means: from any accept state of \( F \), you may proceed to \( C \).
The Role of $\bot$ and $C$

We have

- $\delta'(C, \epsilon, \bot) = \{(q_f, \epsilon)\}$ and
- for each $a \in \Gamma$, $\delta'(C, \epsilon, a) = \{(C, \epsilon)\}$.

These transitions allow emptying stack and then entering $q_f$. 

CSC280, Chapter 2, Part 3 © 2012 Mitsunori Ogihara. Edits/updates by Lane A. Hemaspaandra, 20\{13,14,17\}. 9
No Pop and Push at the Same Time

Suppose we have a permissible transition \((q, c)\) for \(\delta(p, a, b)\), where \(a \in \Sigma_\epsilon\) and \(b, c \in \Gamma\). Then we add a new state \(q'\) exclusively for this particular transition and replace this transition with two transitions:

- \((q', \epsilon)\) in \(\delta(p, a, b)\) and
- \((q, c)\) in \(\delta(q', \epsilon, \epsilon)\).
Construction of Grammar

We say that $M$ can transition from state $p$ to state $q$ on input $w$ while maintaining the minimum stack height if it is possible for $M$ to transition from $p$ to $q$ by processing $w$ so that

- the stack height before reading $w$ is the same as the stack height after finishing to read $w$ and then entering $q$,
- during these two events the stack height never goes below the stack level at the time $M$ starts processing $w$. 
Construction of Grammar

Construct a CFG \((V, \Sigma, P, S)\): \(V = \{A_{pq} \mid p, q \in Q\}\) and \(S = A_{q_0q_f}\), where \(q_0\) is the initial state of \(M\) and \(q_f\) is the unique accept state of \(M\).

\(A_{pq}\) is the variable corresponding to the set of all strings \(w\) that \(M\) can process and transition from \(p\) to \(q\) while maintaining the minimum stack height.
Production rules:

• For every $p \in Q$, $A_{pp} \rightarrow \epsilon$. 
Production rules:

• For every \( p \in Q \), \( A_{pp} \rightarrow \epsilon \).
• For all \( p, q, r \in Q \), \( A_{pq} \rightarrow A_{pr}A_{rq} \).
Production rules:

- For every $p \in Q$, $A_{pp} \rightarrow \epsilon$.
- For all $p, q, r \in Q$, $A_{pq} \rightarrow A_{pr}A_{rq}$.
- For all $p, q, r, s \in Q$, $b, c \in \Sigma_{\epsilon}$, and $d \in \Gamma_{\epsilon}$, if $(r, d) \in \delta(p, b, \epsilon)$ and $(q, \epsilon) \in \delta(s, c, d)$, then $A_{pq} \rightarrow bA_{rs}c$.

This means: one possibility for transition from $p$ to $q$ while maintaining the stack height is to:

- transition from $p$ to $r$ after adding $d$ on top of stack,
- transition from $r$ to $s$ while maintaining the stack height, and
- transition from $s$ to $q$ while popping the $d$. 
Theorem. Each context-free language is recognized by a PDA.

Given any CFL $L$ such that $\epsilon \not\in L$, we want to construct a PDA for $L$.

We can assume that $L$ is given by a CNF grammar $G = (V, \Sigma, R, S)$. (Warning: This is a somewhat different proof approach than Sipser’s!)

We will design a PDA that simulates a leftmost derivation with respect to $G$.

---

1Adding in $\epsilon$ is not hard—it just takes a tweaking of the construction, namely, at the very start guessing whether the input is or is not $\epsilon$, and if we guess the former, going into the accepting state, and if we guess the latter, doing the programming described below.
Simulating a Leftmost Derivation

Use symbol ⊥ to mark the bottom of stack.

After placing a string $S \perp$ (read from top to bottom) on top of the stack, repeat the following:

- Pop one symbol $X$ from stack.
- If $X = \perp$ enter a unique accept state (so if we’re through the input, then the input is accepted).
- Otherwise, nondeterministically select a rule $X \rightarrow w$.
  - If $w = a$ for some terminal $a$, read one input symbol; if the symbol is $a$, continue; otherwise, stop without accepting.
  - If $w = AB$, place $AB$ on top of the stack and continue.

This simple approach correctly accepts $L$. 
Chapter 2, Part 4

Non-context-free Languages

How do we show something is not context free?
The Pumping Lemma

Theorem. (Pumping Lemma) Let $L$ be context free. There exists a positive integer $p$ with the following property.

For every $w \in L$ of length at least $p$, there exist $u, v, x, y, z$, with $w = uvxyz$, such that the following hold:

- $|vy| \geq 1$,
- $|vxy| \leq p$, and
- for each $i \geq 0$, $uv^i xy^i z \in L$. 
The Pumping Lemma

Theorem. (Pumping Lemma) Let $L$ be context free. There exists a positive integer $p$ with the following property.

For every $w \in L$ of length at least $p$, there exist $u, v, x, y, z$, with $w = uvxyz$, such that the following hold:

- $|vy| \geq 1$,
- $|vxy| \leq p$, and
- for each $i \geq 0$, $uv^i xy^i z \in L$.

The differences between this pumping lemma and the regular-sets one.

- There are two components that are jointly inserted or deleted.
- The part $vxy$ might not be at the beginning of $w$. 

Proving the Pumping Lemma

Let $L = L(G)$ for some CNF grammar $G = (V, \Sigma, R, S)$.

If $L$ is finite (i.e., has only a finite number of members), then there is a length $k$ such that each member of $L$ has length less than $k$. We have only to choose $p$ to be $k$.

So we will assume $L$ is infinite.
Set $m = \|V\|$ and $p = 2^m$.

Let $w$ be an arbitrary member of $L$ having length at least $p$. Let $T$ be a derivation tree for $w$.

Since $G$ is a CNF grammar, for each subtree of $T$, the following properties hold:

- Each nonleaf node of $R$ is a variable.
- Each leaf of $R$ is a terminal.
- Each leaf of $R$ is a unique child of its parent.
- Except for the leaves and their parents each node of $R$ has exactly two children.
- The concatenation of the leaves of $R$ is a substring of $w$. 
A Useful Property

An ancestor–descendant pair with identical label (ADPIL, for short) in a production tree $R$ is a node pair $(r, s)$ such that

- $r$ is an ancestor of $s$ and
- the label of $r$ is identical to the label of $s$ (and thus, the label is a nonterminal).
Proof (cont’d)

Claim. If $R$ has more than $p/2 = 2^{m-1}$ leaves, then $R$ contains an ADPIL.
Claim. If \( R \) has more than \( p/2 = 2^{m-1} \) leaves, then \( R \) contains an ADPIL.

Proof

Suppose \( R \) is a subtree of \( T \) with at least \( 2^{m-1} + 1 \) leaves.

Let \( R' \) be the tree constructed form \( R \) by removing all the leaves.

Since the terminals appear only at the leaves, the claim is equivalent to saying that \( R' \) has an ADPIL.

The claim is proved by showing, by contradiction, that there is a root-to-leaf path in \( R' \) with at least \( m + 1 \) nodes,
Proof (cont’d)

Assume, by way of contradiction, that every root-to-leaf path of $R'$ has at most $m$ nodes,
Proof (cont’d)

Assume, by way of contradiction, that every root-to-leaf path of \( R' \) has at most \( m \) nodes,

Then the number of branches in any such path is at most \( m - 1 \). Since \( R' \) is a binary tree, \( R' \) has at most \( 2^{m-1} \) leaves.
Proof (cont’d)

However, the number of leaves of $R'$ is greater than $2^{m-1}$. Thus, there is a root-to-leaf path, say $\pi$, in $R'$ having length at least $m + 1$.

Then, by the pigeonhole principle, an ADPIL appears on $\pi$.

Proof of Claim
Proof of Pumping Lemma (cont’d)

Using the following algorithm to find an ADPIL \((r, s)\) farthest from the root of \(T\).

1. Set \(u\) to the root of \(T\).
2. Execute the following loop:
   - If the left child of \(u\) has an ADPIL, set \(u\) to the left child of \(u\).
   - Otherwise, if the right child of \(u\) has an ADPIL, set \(u\) to the right child of \(u\).
   - Otherwise, quit the loop.
3. Set \(r = u\) and \(s\) to the leftmost node with the same label as \(r\).
Using the following algorithm to find an ADPIL \((r, s)\) farthest from the root of \(T\).

1. Set \(u\) to the root of \(T\).
2. Execute the following loop:
   - If the left child of \(u\) has an ADPIL, set \(u\) to the left child of \(u\).
   - Otherwise, if the right child of \(u\) has an ADPIL, set \(u\) to the right child of \(u\).
   - Otherwise, quit the loop.
3. Set \(r = u\) and \(s\) to the leftmost node with the same label as \(r\).

The children of \(r\) have no ADPILs. Thus, both children have at most \(2^{m-1}\) leaves and so \(r\) has at most \(p = 2^m\) leaves.
Proof of Pumping Lemma (cont’d)

Let \( x \) be the string at the leaf-level of the subtree rooted at \( s \). Similarly, let \( vxy \) be the one for \( r \), where \( v \) and \( y \) are those to the left and to the right of \( x \), respectively.

Let \( u \) be the string produced to the left of \( r \) and \( z \) to the right of \( s \).
Proof of Pumping Lemma (cont’d)

Let $x$ be the string at the leaf-level of the subtree rooted at $s$. Similarly, let $vxy$ be the one for $r$, where $v$ and $y$ are those to the left and to the right of $x$, respectively.

Let $u$ be the string produced to the left of $r$ and $z$ to the right of $s$.

Then $|vxy| \leq p$.

Also, since $s$ is a descendant of $r$ and $G$ has no $\epsilon$-production except for $S \rightarrow \epsilon$, $x$ is a proper substring of $vxy$. Thus, $|vy| \geq 1$. 
Proof of Pumping Lemma (cont’d)

Since both \( r \) and \( s \) have the same label, they are swappable. So, for every \( i \geq 0 \), \( uv^ixy^iz \in L \).
Example 1

\[ A = \{0^n 1^n 2^n \mid n \geq 0\} \text{ is not context free.} \]
Example 1

\[ A = \{0^n1^n2^n \mid n \geq 0\} \] is not context free.

**Proof**  Assume, to the contrary, that \( A \) is context free. By Pumping Lemma there exists a constant \( p \) such that every \( w \in A \) of length \( \geq p \) can be divided into \( w = uvxyz \) such that \(|vxy| \leq p\), \(|vy| \geq 1\), and for every \( i \geq 0\), \( uv^ixy^iz \in A \).

Let \( w = 0^p1^p2^p \). Since \(|vxy| \leq p\), \( vxy \) is either in \( 0^*1^* \) or in \( 1^*2^* \). This means that \( uv^2xy^2z \) cannot have the same number of 0s, 1s, as 2s.
Hey, I think I can show $0^n1^n2^n$ isn't context free.
Hey, I think I can show $0^n1^n2^n$ isn't context free.

Wow, that's great. Tell me about it.
Illustrating Conversation

If that thing is context free, I get this magic constant $p$. 
Illustrating Conversation

If that thing is context free, I get this magic constant \( p \).

Oh, I know it. You can divide any word in that thing into three parts...
Illustrating Conversation

No, you idiot! It's FIVE parts, w=uvwxyz!
No, you idiot! It's FIVE parts, $w=uvxyz!$

Okay. Let's say $w = 0^p1^p2^p$. Then what?
$w$ can be broken down into $uvxyz$, where $vy$ is nonempty and $vxy$ has length at most $p$. 
Illustrating Conversation

$w$ can be broken down into $uvxyz$, where $vy$ is nonempty and $vxy$ has length at most $p$.

I see. Then $v$ and $y$ can touch at most two sections.
Right. So, either pumping in or pumping out, you can create a word that can't be in that thing.
Illustrating Conversation

Right. So, either pumping in or pumping out, you can create a word that can't be in that thing.

Brilliant. You learned well in your CSC280.
By the way, didn't you call me "idiot"?
Example 2

\[
B = \{a\#b\#c \mid a, b \text{ and } c \text{ are binary numbers such that } a + b = c\}
\]
is not context free.
Example 2

\[ B = \{a\#b\#c \mid a, b \text{ and } c \text{ are binary numbers such that } a + b = c \} \]
is not context free.

**Proof** Assume, to the contrary, that \( B \) is context free. Let \( p \) be the constant from Pumping Lemma for \( B \). Let \( w = 10^p\#10^p\#10^{p+1} \), where \( a = b = 2^p \) and \( c = 2^{p+1} \). Let \( uvxyz \) be a decomposition of \( w \) as in the lemma.
Example 2

\[ B = \{ a\#b\#c \mid a, b \text{ and } c \text{ are binary numbers such that } a + b = c \} \]
is not context free.

Proof Assume, to the contrary, that \( B \) is context free. Let \( p \) be the constant from Pumping Lemma for \( B \). Let \( w = 10^p \#10^p \#10^{p+1} \), where \( a = b = 2^p \) and \( c = 2^{p+1} \). Let \( uvxyz \) be a decomposition of \( w \) as in the lemma.

Since each member of \( B \) has exactly two \( \# \)'s, neither \( v \) nor \( y \) can contain a \( \# \) (as otherwise pumping with \( i = 2 \) would put us clearly out of the language due to too many \( \# \)'s). So, \( v \) must be a substring of \( a \), a substring of \( b \), or a substring of \( c \). The same holds for \( y \).
Example 2 (cont’d)

Since the equality $a + b = c$ must be maintained during pumping and $|vy| \geq 1$, $y$ must be a nonempty substring of $c$ and $v$ must be either a nonempty substring of $a$ or a nonempty substring of $b$. However, since $vxy$ has length at most $p$, it must be the case that $v$ is a nonempty substring of $b$. 
Example 2 (cont’d)

Since the equality $a + b = c$ must be maintained during pumping and $|vy| \geq 1$, $y$ must be a nonempty substring of $c$ and $v$ must be either a nonempty substring of $a$ or a nonempty substring of $b$. However, since $vxy$ has length at most $p$, it must be the case that $v$ is a nonempty substring of $b$.

If $v$ contains the first letter of $b$, then $vxy$ (due to its length) must all be a part of $b$, contradicting the above; so $v$ should consist solely of 0’s.
Example 2 (cont’d)

Since the equality $a + b = c$ must be maintained during pumping and $|vy| \geq 1$, $y$ must be a nonempty substring of $c$ and $v$ must be either a nonempty substring of $a$ or a nonempty substring of $b$. However, since $vxy$ has length at most $p$, it must be the case that $v$ is a nonempty substring of $b$.

If $v$ contains the first letter of $b$, then $vxy$ (due to its length) must all be a part of $b$, contradicting the above; so $v$ should consist solely of 0’s.

Suppose $y$ contains 1, the first letter of $c$. Then $uvz$ is the form $10^p \#10^q \#0^r$, which clearly is not a member of $B$. 
Example 2 (cont’d)

Since the equality $a + b = c$ must be maintained during pumping and $|vy| \geq 1$, $y$ must be a nonempty substring of $c$ and $v$ must be either a nonempty substring of $a$ or a nonempty substring of $b$. However, since $vxy$ has length at most $p$, it must be the case that $v$ is a nonempty substring of $b$.

If $v$ contains the first letter of $b$, then $vxy$ (due to its length) must all be a part of $b$, contradicting the above; so $v$ should consist solely of 0’s.

Suppose $y$ contains 1, the first letter of $c$. Then $uvz$ is the form $10^p \#10^q \#0^r$, which clearly is not a member of $B$.

Suppose $y$ doesn’t contain the letter 1. Then $y$ consists solely of 0s; so $uvvxyyz$ is of the form $10^p \#10^q \#10^r$ such that $q, r > p$, which isn’t a member of $B$ (consider: $2 + 8$ isn’t a power of 2).
Example 3

\[ C = \{ww \mid w \in \{0, 1\}^*\} \] is not context free.
Example 3

$C = \{ww \mid w \in \{0, 1\}^*\}$ is not context free.

Proof Assume $C$ is context free. Let $p$ the constant from the pumping lemma for $C$.

Let $w = 0^p1^p0^p1^p$. Then $w$ in $C$. 
Example 3

$C = \{ww \mid w \in \{0, 1\}^*\}$ is not context free.

**Proof** Assume $C$ is context free. Let $p$ the constant from the pumping lemma for $C$.

Let $w = 0^p1^p0^p1^p$. Then $w$ in $C$.

Let $w = uvxyz$ be a decomposition of $w$ such that $|vy| > 0$, $|vxy| \leq p$, and for every $i \geq 0$, $uv^ixy^iz \in C$. 
Example 3

\( C = \{ww \mid w \in \{0, 1\}^*\} \) is not context free.

**Proof**  Assume \( C \) is context free. Let \( p \) the constant from the pumping lemma for \( C \).

Let \( w = 0^p1^p0^p1^p \). Then \( w \in C \).

Let \( w = uvxyz \) be a decomposition of \( w \) such that \(|vy| > 0\), \(|vxy| \leq p\), and for every \( i \geq 0 \), \( uv^i xy^i z \in C \).

If \( v \) contains a symbol from the first \( 0^p \) then \( y \) cannot contain one from the second \( 0^p \), so pumping doesn’t work.
Example 3

$C = \{ww \mid w \in \{0, 1\}^*\}$ is not context free.

**Proof** Assume $C$ is context free. Let $p$ the constant from the pumping lemma for $C$.

Let $w = 0^p1^p0^p1^p$. Then $w$ in $C$.

Let $w = uvxyz$ be a decomposition of $w$ such that $|vy| > 0$, $|vxy| \leq p$, and for every $i \geq 0$, $uv^ixy^iz \in C$.

If $v$ contains a symbol from the first $0^p$ then $y$ cannot contain one from the second $0^p$, so pumping doesn’t work.

If $v$ contains only symbols from the first $1^p$ then $y$ cannot contain one from the second $1^p$, so pumping doesn’t work.
Example 3

$C = \{ww \mid w \in \{0, 1\}^*\}$ is not context free.

Proof Assume $C$ is context free. Let $p$ the constant from the pumping lemma for $C$.

Let $w = 0^p1^p0^p1^p$. Then $w$ in $C$.

Let $w = uvxyz$ be a decomposition of $w$ such that $|vy| > 0$, $|vxy| \leq p$, and for every $i \geq 0$, $uv^i xy^i z \in C$.

If $v$ contains a symbol from the first $0^p$ then $y$ cannot contain one from the second $0^p$, so pumping doesn’t work.

If $v$ contains only symbols from the first $1^p$ then $y$ cannot contain one from the second $1^p$, so pumping doesn’t work.

If $v$ contains only symbols from the second $0^p1^p$ then pumping does not work.
Corollary. The class of context-free languages is not closed under intersection.

Proof Let $L_1 = \{0^i1^j2^k \mid i = j\}$ and $L_2 = \{0^i1^j2^k \mid j = k\}$. Then $L_1$ and $L_2$ are both context free. If the class were closed under intersection then $L_1 \cap L_2 = \{0^n1^n2^n \mid n \geq 0\}$ would be context free.

Corollary. The class of context-free languages is not closed under complement.

Proof We know that the class is closed under union. If the class were closed under complement, then by DeMorgan’s Law, it would be closed under intersection.
Chapter 3, Part 1

Computability Theory
Turing Machines

A Turing machine is a step-wise computing device consisting of:

1. an infinitely long tape divided into tape squares (or tape cells),
2. a head for scanning the tape, and
3. a finite control that maintains the current state.
Turing Machines

A Turing machine is a step-wise computing device consisting of:

1. an infinitely long tape divided into tape squares (or tape cells),
2. a head for scanning the tape, and
3. a finite control that maintains the current state.
Computation by a Turing Machine

In each computational step, a Turing machine:

• reads the symbol written at the current head location, and then,

• depending on the symbol and the current state
  • determines its next state,

• write a symbol at the current head location, and

• moves the head to either the next or the prior tape square.
A **Turing machine** is a 7-tuple \((Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})\), where \(Q, \Sigma, \Gamma\) are finite sets,

1. \(Q\) is a set of states,
2. \(\Sigma\) is the input alphabet,
3. \(\Gamma\) is the tape alphabet such that \(\Sigma \subseteq \Gamma\). There is a special symbol \(\sqcup\) in \(\Gamma - \Sigma\).
4. \(\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}\) is the transition function,
5. \(q_0 \in Q\) is called the **initial state**,
6. \(q_{\text{accept}} \in Q\) is called the **accept state**, and
7. \(q_{\text{reject}} \in Q\) is called the **reject state**, \(q_{\text{accept}} \neq q_{\text{reject}}\).
Initial Condition of a Turing Machine

At the beginning

1. the tape contains its input in the leftmost squares,
2. the rest of the tape is filled with $\square$,
3. the head is at the leftmost cell, and
4. the state is $q_0$. 

\[
\begin{array}{c}
\text{left end of tape}\\
\downarrow\\
\begin{array}{ccccccc}
\text{tape} \\
\text{head} \\
\text{finite control} \\
\end{array}
\end{array}
\]
Termination of a Turing Machine

The Turing machine halts when it enters either $q_{\text{accept}}$ or $q_{\text{reject}}$. We say that the machine accepts when the former occurs and rejects when the latter occurs.
Computation of a Turing Machine in One Step (Again)

In each computational step, a Turing machine:

- reads the symbol written at the current head location, and then,
- depending on the symbol and the current state
  - determines its next state,
  - write a symbol at the current head location, and
  - moves the head to either the next or the prior tape square.
Computation of a Turing Machine in One Step (Again)

In each computational step, a Turing machine:

- reads the symbol written at the current head location, and then,
- depending on the symbol and the current state
  - determines its next state,
  - write a symbol at the current head location, and
  - moves the head to either the next or the prior tape square.

However, if the instruction is to move the head to the left and the current head position is the leftmost square, then the head stays at the leftmost square.
Example

\[ \text{becomes} \]

\[ \text{becomes} \]
The Languages of Turing Machines

A Turing machine $M$ accepts (rejects) a string $x$ if it enters the accepting (rejecting) state on input $x$.

A Turing machine $M$ recognizes a language $L$, if for every input $x$, $M$ on $x$ accepts if $x \in L$ and does not accept if $x \notin L$.

A language is Turing-recognizable (or recursively enumerable) if there is a Turing machine that recognizes it.
A Turing machine $M$ **decides** a language $L$, if $M$ recognizes $L$ and halts on all inputs.

In other words, a Turing machine $M$ **decides** a language $L$ if, for all $x$, $M$ on $x$ accepts if $x \in L$ and rejects if $x \notin L$.

A language is **Turing-decidable** (or simply **decidable**, or **recursive**) if there is a Turing machine that decides it.
**Example 1:** \( L = \{w#w \mid w \in \{0, 1\}^*\} \)

\( L \) is not context free.

To decide whether an input \( z \in \{0, 1, \#\}^* \) is in \( L \) our Turing machine \( M \) operates as follows:

- \( M \) attempts to match letter by letter the word to the left of \( \# \) (call the word \( u \)) and the word to the right of \( \# \) (call the word \( v \)).
- When a letter in \( u \) matches a letter in \( v \) at the corresponding position, both letters are erased using a special letter \( x \).
- When all the letters of \( u \) and \( v \) have been matched, \( M \) accepts.
Components of $M$

As a very informal, loose intuition, $M$ inside itself has the following components:

- an automaton for $x^* \# x^*$;
- an automaton for $x^* 0\{0,1\}^* \# x^* 0\{0,1\}^*$; and
- an automaton for $x^* 1\{0,1\}^* \# x^* 1\{0,1\}^*$;

Or, to be more detailed and to be correct (always a nice thing) in phrasing this, $M$ will have the programming that is necessary to correctly handle *everything* it might face, including these three situations, which are central ones underpinning the language we are trying to handle (namely, to decide).

The last two automata will replace the letter immediately after each run of $x$ with an $x$ if the two letters match.
Method for Matching

The tape contents are always of the form \( x^* \{0, 1\}^* \# x^* \{0, 1\}^* \), where the number of \( x \)'s on the left is equal to the number of \( x \)'s on the right.
Action in the Very First Step

• If the symbol being scanned is $\sqcup$, then immediately reject $z$ because $z = \epsilon$.
• Otherwise, enter the loop.
Main Loop

• **(Case 1)** If the letter at the current head position is a #, then:
  • scan to the right until a letter other than x is found, and then
  • accept if that non-x is ⊔; reject otherwise.
Method for Matching

• (Case 2) If the letter at the current head location is a 0 or a 1, then:
  • memorize the letter, say \( c \), and write an \( x \),
  • scan to the right until a \# is found,
  • scan to the right until a non-\( x \) is found, and then
  • if that non-\( x \) is not a \( c \), then reject,
  • replace the non-\( x \) with an \( x \),
  • scan to the left until a \# is found,
  • scan to the left until an \( x \) is found,
  • move to the right thereby locating the head to the very first non-\( x \).
Example \( \{ w\#w \mid w \in \{0, 1\}^* \} \)
The Main Loop

First non-X
The Main Loop

0 memorized
The Main Loop

0 memorized
The Main Loop

0 memorized
The Main Loop

0 memorized
and
# found
The Main Loop

0 memorized and # found
The Main Loop

```
X  X  1  0  #  X  0  1  0
```

0 memorized
# found
and
0 found
The Main Loop

\[
\begin{array}{cccccccc}
X & X & 1 & 0 & \# & X & X & 1 & 0 \\
\end{array}
\]

matched
The Main Loop

```
X X 1 0 # X X 1 0
```

matched
# found
The Main Loop

X X 1 0 # X X 1 0

matched
# found
The Main Loop

matched

# found
The Main Loop

matched
# found
X found
The Main Loop

X X 1 0 # X X 1 0

matched
# found
X found
Start of new round
Example 2: \( \{ w\#w\#w \mid w \in \{0,1\}^* \} \)

- Instead of \( x \), use special symbols \( x_0 \) and \( x_1 \) used for crossing out a 0 and a 1, respectively.
- Execute the algorithm for the previous example with these substitutions and treating the second \( \# \) as \( \sqcup \).
- The algorithm halts in the accept state if and only if the input is in the form \( w\#w\#y \) for some \( w, y \in \{0,1\}^* \).
- If accept, move on to the next stage.
Example 2: \( \{ w\#w\#w \mid w \in \{0, 1\}^* \} \)

- Move the head to the symbol immediately to the right of the first \#.
- Execute the algorithm for the previous example by treating \( x_0 \) as 0 and \( x_1 \) as 1.
- Accept if and only if this simulation accepts.
**Configurations**

A **configuration** of a Turing machine is the setting of its tape, head, and state.

If the tape contents are $a_1, \ldots, a_m, \sqcup, \ldots$, the head is located on the $k$th square, and the state is $q$, then write

$$a_1 \ldots a_{k-1}qa_k \ldots a_m$$

to represent the configuration as a word. Note that $Q$ is treated as an alphabet. Of the infinite stretch of $\sqcup$’s that form a right suffix of the tape, we leave out of the configuration any that happen to fall to the right of the head position (and optionally if one of that infinite right-suffix of blanks is under the tape head it can also be omitted, since $uq_i$ should and will be treated as being in fact $uq_i\sqcup$). Thus a configuration is a word in $\Gamma^*Q\Gamma^*$. 
Examples

The configuration is \( abq_1abc \).

The configuration is \( aq_2bbbc \).
Configurations (cont’d)

The action of a TM can be viewed as rewriting of the configuration.

A configuration $C_1$ **yields** $C_2$ if the Turing machine can go from $C_1$ to $C_2$ in a single step. (Recall that $\hat{u}q_i$ is treated as equivalent to $\hat{u}q_i \sqcup$; for this case, the rules below regard the latter.)

- $uaq_i bv$ yields $uq_j acv$ if $\delta(q_i, b) = (q_j, c, L)$.
- $q_i bv$ yields $q_j cv$ if $\delta(q_i, b) = (q_j, c, L)$ (the set cannot move to the left).
- $uq_i bv$ yields $ucq_j v'$ if $\delta(q_i, b) = (q_j, c, R)$.

Here $v' = v$ if $v \neq \epsilon$ and $v' = \sqcup$ if $v = \epsilon$. 

Special Configurations

An accepting configuration (A rejecting configuration) is one in which the state is $q_{\text{accept}} (q_{\text{reject}})$.

Both accepting configuration and rejecting configuration are halting configurations.
Chapter 3, Part 2

Variants of Turing Machines
Multitape TMs

A multitape Turing machine is a Turing machine with additional tapes with each tape is accessible individually, with the input on the first tape, and with the others blank at the beginning.

For a $k$-tape Turing machine, the transition $\delta$ is a mapping from $Q \times \Gamma^k$ to $Q \times \Gamma^k \times \{L, R\}^k$. 
**Nondeterministic TMs**

A **nondeterministic Turing machine** is one in which the transition is mapping to the power set of $Q \times \Gamma \times \{L, R\}$.

A nondeterministic Turing machine **accepts** an input if it enters an accepting state **for some computation path**.
Equivalence Between Single-tape TMs and Multitape TMs

Theorem. Every multitape Turing machine has an equivalent single-tape Turing machine.
Theorem. Every multitape Turing machine has an equivalent single-tape Turing machine.

Proof From a $k$-tape TM $M$ build a single-tape simulator $S$. 
Equivalence Between Single-tape TMs and Multitape TMs

**Theorem.** Every multitape Turing machine has an equivalent single-tape Turing machine.

**Proof** From a $k$-tape TM $M$ build a single-tape simulator $S$.

The main idea is to use the tape available to represent

- the contents of the tape squares that the head has ever visited for each tape, including the entire squares that initially hold the input,
- and the current head position for each tape.

Create such a representation for each tape and connect them with a delimiter in between, at the beginning, and at the end.
Tape Encoding

For each \( a \in \Gamma \), let \( \tilde{a} \) be a new symbol to signify that a head is located on the symbol.

- The input tape \( w_1 \cdots w_n \sqsubset \cdots \) with the head scanning the first symbol (this occurs at the beginning)

\[
\tilde{w}_1w_2\cdots w_n.
\]
Tape Encoding

• If a tape holds $a_1 \cdots a_s \sqcup \cdots$ and the farthest position the head has traveled is $t > r$.
  • If the head position is $r < s$, then its representation is:
    \[
    a_1 \cdots a_{r-1} \widehat{r} s a_r \cdots a_s \sqcup \cdots \sqcup t - s.
    \]

• If the head position is $r > s$, then its representation is:
    \[
    a_1 \cdots a_s \sqcup \cdots \sqcup \widehat{r - s - 1} \sqcup \cdots \sqcup t - r.
    \]

The encoding never decreases in length.
Delimiter

Use a new symbol \# as a delimiter.

On input $w = w_1 \cdots w_n$, the initial form of encoding is:

$$\#\bar{w}_1 w_2 \cdots w_n \#\bar{\bar{\#}} \# \bar{\bar{\#}} \cdots \bar{\bar{\#}}$$
$S$’s Action

Memorize $M$’s state using a state.

1. Construct the initial form.

2. Repeat the following:
   (a) If $M$ has accepted or rejected, accept or reject accordingly.

   (b) Otherwise, scan the tape in a direction and record, using the state, the symbols being scanned by the heads of $M$.

   (c) **Determine the next move** of $M$.

   (d) **Modify the encoding accordingly**. Insert symbols if necessary.

   (e) **Change the state accordingly**.
Identification of Symbols Scanned

In the case when the tape is scanned from right to left, use the following states.

1. \((p_{\text{scan}}, q, a_1, \ldots, a_k), a_1, \ldots, a_k \in \Gamma\): This means that the current state is \(q\) and the symbols being scanned are \(a_1, \ldots, a_k\).

2. \((p_{\text{scan}}, q, ?, \ldots, ?, a_{r+1}, \ldots, a_k), a_{r+1}, \ldots, a_k \in \Gamma\): This means that for the first \(r\) tapes the symbols being scanned are yet to be identified but for the others the symbols have been identified to be \(a_{r+1}, \ldots, a_k\).
Identification of Symbols Scanned

In the case when the tape is scanned from right to left, use the following states.

1. \((p_{\text{scan}}, q, a_1, \ldots, a_k), a_1, \ldots, a_k \in \Gamma\): This means that the current state is \(q\) and the symbols being scanned are \((p_{\text{scan}}, q, ?, \ldots, ?, a_{r+1}, \ldots, a_k), a_{r+1}, \ldots, a_k \in \Gamma\): This means that for the first \(r\) tapes the symbols being scanned are yet to be identified but for the others the symbols have been identified to be \(a_{r+1}, \ldots, a_k\).

Start scanning from the end in state \((p_{\text{scan}}, q, ?, \ldots, ?)\).

Each time a symbol of the form \(\hat{X}\) is encountered, replace the rightmost \(?\) with that \(X\).

When all \(?\)'s are gone, \(S\) knows the action of \(M\).

If forward motion is used, the symbols are from left to right.
Tape Modification

This step consists of

- rewriting the symbol on the current head position and
- rewriting the symbols around the current head position to move the head to the left or to the right.
Tape Modification Rules

- If the current contents are \( \cdots a\tilde{b}\cdots \), \( a \neq \# \), \( b \) is to be replaced by \( b' \), and the head moves to the left, then replace the two symbols by \( \tilde{a}b' \).

- If the current contents are \( \cdots \#\tilde{b}\cdots \), \( b \) is to be replaced by \( b' \), and the head moves to the left, then replace the two symbols by \( \#\tilde{b}' \).

- If the current contents are \( \cdots \tilde{b}a\cdots \), \( a \neq \# \), \( b \) is to be replaced by \( b' \), and the head moves to the right, then replace the two symbols by \( b'\tilde{a} \).
Tape Modification Rules (cont’d)

• If the current contents are \( \cdots \tilde{b}\# \cdots \), \( b \) is to be replaced by \( b' \), and the head moves to the right, then replace the \( \tilde{b}\# \) by \( b'\tilde{b}\# \).

This triggers insertion:

• Use a state to remember the symbol to be inserted.

• Start by memorizing the very first insertion, \( \tilde{b}\# \), and then move to the right.

• While scanning to the right, swap the symbol to be inserted and the symbol stored in the tape cell.

• Keep scanning until the \( \sqsubset \) after the very last symbol of the encoding.
Insertion

Change Y to Y'D
Insertion

Must Insert D Here
Insertion

Must Insert Z
Here
Insertion

# X Y' D Z # V A B C # □ □ □ □ □

Must Insert U Here
Insertion

Must Insert #
Here
Insertion

Must Insert V Here
Insertion

\[ \# X Y' D Z U \# V B C \# \square \square \square \square \square \]

Must Insert A
Here
Insertion

# X Y' D Z U # V A C # □ □ □ □ □ □

Must Insert B Here
Insertion

Must Insert C Here
Insertion

Must Insert # Here
**Insertion**

```
# X Y' D Z U # V A B C # □ □ □ □
```

Done
Equivalence Between NTMs and TMs
Theorem. Every nondeterministic Turing machine has an equivalent deterministic Turing machine.
Equivalence Between NTMs and TMs

Theorem. Every nondeterministic Turing machine has an equivalent deterministic Turing machine.

Proof We may assume that $N$ is a single-tape machine — we can use the same proof as before.
Equivalence Between NTMs and TMs

Theorem. Every nondeterministic Turing machine has an equivalent deterministic Turing machine.

Proof We may assume that $N$ is a single-tape machine — we can use the same proof as before.

We will construct a three-tape simulator $D$ of $N$. 
Construction

Let $C$ be a constant such that each transition has at most $C$ possible values. Let $\Theta = \{a_1, a_2, \ldots, a_C\}$.

Use a word $p \in \Theta^*$ to encode a nondeterministic path, where for all $i \geq 1$ and $j$, $1 \leq j \leq b$, if the $i$-th symbol of $p$ is $a_j$, then it specifies at step $i$, $M$ must choose the $j$-th possibility from all possible moves available at that point (if such one exists).

The word $p$ over $\Theta$ is a valid computation path of $N$ on input $w$ if $N$ on $w$ halts according to the choices written on $p$. 
Three-tape Simulation

Use Tape 1 to store the input, Tape 2 to simulate the tape of $N$, and Tape 3 to keep an encoding of a computation path.

Define the lexicographic order of paths:

$\mathbf{u}_1 \cdots \mathbf{u}_s < \mathbf{v}_1 \cdots \mathbf{v}_t \in \Theta^*$ if and only if either

- $s < t$ or
- $s = t$ and there exists some $k$, $1 \leq k \leq s$, such that $\mathbf{u}_1 = \mathbf{v}_1, \ldots, \mathbf{u}_{k-1} = \mathbf{v}_{k-1}$, and $\mathbf{u}_k < \mathbf{v}_k$.

Here $\mathbf{u}_k < \mathbf{v}_k$ is evaluated according to a fixed ordering of letters in $\Theta$. 
An Algorithm for $N$

On input $w$, write the word $\#a_1$ on Tape 3, then repeat:

1. **Copy the input onto Tape 2.**
2. **Try to simulate $N$ on $w$ using the word in Tape 3 as the path.** If successful and if $N$ has accepted, then accept and halt.
3. **Modify the path to the next smallest path by incrementing it.**
4. **Erase Tape 2.**
Some Additional Results

Corollary. A language is Turing-recognizable if and only if it is recognized by a multitape TM.

Corollary. A language is Turing-recognizable if and only if it is recognized by an NTM.
Theorem. A language is Turing-recognizable if and only if it is recognized by a “two stack” machine (which is a machine that is TM-like except the input is read-only on its own tape, and instead of a tape the machine controls two stacks).

Can you see how to prove this?
Weaker Models: How Weak Can We Go?

How weak can we push things? Well, for example, we have:

**Theorem.** A language is Turing-recognizable if and only if it is recognized (in Sipser’s model for how the left end of the tape is handled) by a “three pebble machine” (the machine has three identical pebbles, that start on the left end of a Sipser-like one-way-infinite work tape and it can read whether on its current cell there are zero or one or two or three pebbles and it can lift and move a pebble one cell at a time, and the input is on a separate read-only input tape).

Proving this is way hard.
Weaker Models: How Weak Can We Go?

Theorem. A language is Turing-recognizable if and only if it is recognized (in Sipser’s model for how the left end of the tape is handled) by a “three pebble machine.”

As mentioned last slide, proving this is way hard. We have to see that three pebbles can in the Sipser model simulate “two nonnegative integer counters.” And that two counters can simulate four counters. And that four counters can simulate an arbitrary TM. Each of those steps takes serious proving, and we won’t cover them, except you probably can see right now how to simulate two counters with three pebbles in the Sipser model. (I mention in passing that if the only actions we need to perform on our counters are increment, test-for-zero, and proper-decrement, we can even do that in the Sipser-type model with two pebbles, being very careful about the left end of the tape. Do you see how?)
Enumerators

An enumerator of a language $A$ is a TM with a special output tape such that the machines write on the output tape all the members of $A$ with a special symbol $\#$ as a delimiter (put at the end of each string). Note: The output-tape head can only either write-a-character-and-move-right or stay-in-place.

What we mean by this is that it must have the property that each string it prints is in the set, and for each string in the set it holds that that string will eventually be printed. For example, an enumerator for the set $0^*$ might, after a certain number of steps, have this on its output tape (and it would eventually have to catch the ones currently skipped):

$$0##000000##0##0##0##0##0##0##0##000###000#$$

(An enumerator for $\{\epsilon\}$ might print for example $\#$ or $\#\#$. An enumerator for $\emptyset$ ideally should just leave its output tape blank.)
Enumerators

An **enumerator** of a language $A$ is a TM with a special **output tape** such that the machines write on the output tape all the members of $A$ with a special symbol $\#$ as a delimiter.

**Theorem.** A language is Turing-recognizable if and only if it has an enumerator.
Enumerators

An enumerator of a language $A$ is a TM with a special output tape such that the machines write on the output tape all the members of $A$ with a special symbol $\#$ as a delimiter.

**Theorem.** A language is Turing-recognizable if and only if it has an enumerator.

**Proof**  The “if” part: Simulate the enumerator, and accept when the input word is produced by the enumerator.

The “only if” part: Simulate a recognizer $M$. For $i = 1, 2, \ldots$, for each $w$ coming at or before position $i$ within the shortlex (aka string; CS theory people other than Sipser typically call this lexicographical order but, sigh, he saves that term for the dictionary order) order on $\Sigma^*$, simulate $M$ on $w$ for $i$ steps and output $w$ if $M$ accepts $w$ within $i$ steps.
Enumerators

An enumerator of a language $A$ is a TM with a special output tape such that the machines write on the output tape all the members of $A$ with a special symbol $\#$ as a delimiter.

Theorem. A language is Turing-decidable if and only if it has an enumerator that enumerates the strings in increasing order (within the shortlex ordering).

Can you see the proof of each direction? Be careful—there is a shocking twist in proving the “if” direction. (Hint: Are there two quite different cases there?)
Enumerators Applied

An enumerators of a language \( A \) is a TM with a special output tape such that the machines write on the output tape all the members of \( A \) with a special symbol \( \# \) as a delimiter.

**Theorem.** A language \( L \) is Turing-decidable if and only if both \( L \) and \( \overline{L} \) are Turing-recognizable.

Can you see the proof of each direction?

Note: The complexity analogue of this result would be \( \text{NP} \cap \text{coNP} = \text{P} \)—but that complexity analogue most definitely is an open issue!
Description of Objects

We assume that there is a systematic way of describing computing devices as well as their inputs. For example, a Turing machine $M$ can be described by putting down in symbols states, symbols, and transition. We fix such an encoding system. We will use $\langle M \rangle$ to represent the encoding of $M$. 
Description of Multiple Objects

To encode multiple objects in a sequence, we simply concatenate the encodings of the objects in order with a special delimiter in between.
Description of Multiple Objects

To encode multiple objects in a sequence, we simply concatenate the encodings of the objects in order with a special delimiter in between.

If you’re a student, ignore the rest of this page.¹

¹ Note just to myself: If I am going to cover the recursion theorem and use the Chapter-6-slides’ rather than Sipser’s version of its proof, then one has to add in this unnatural special requirement as to the interpretation of the coding: For Turing machines \( M \) and \( N \), \( \langle M \rangle \langle N \rangle \) (yes, that really is a concatenation with no special delimiter, sigh) is a representation of a Turing machine that executes \( M \)’s program first and if \( M \) accepts immediately jumps into \( N \)’s program.
A Fun, Hard Challenge

Can you write a Turing machine that on each input \( i = 0, 1, 2, 3, \ldots \) outputs a Turing machine encoding, call it \( \langle \sigma(i) \rangle \), and halts, and those outputs are such that we have that the set \( \{L(\sigma(0)), L(\sigma(1)), L(\sigma(2)), \ldots \} \) is the set of all decidable languages over the alphabet \( \{0, 1\} \)? Many people who know a lot might think it is impossible to do this. But it can be done (admittedly in an extremely evil, sneaky, unsatisfying way)!

To think about this, you might want to think about how to do it in C++ or Java!

Super-challenge: Can you remove the extremely evil, sneaky, unsatisfying trick? Or can you prove that it cannot possibly be removed!? (Hint: The latter holds, using a technique that will be central to us in Ch. 4, Pt. 2.)
Another (Your Lucky Day!) Fun, Hard Challenge

Can you write a Turing machine that on each input $i = 0, 1, 2, 3, \ldots$ outputs a Turing machine encoding, call it $\langle \sigma(i) \rangle$, and halts, such that both of the following hold: (1) in the list of languages $L(\sigma(0))$, $L(\sigma(1))$, $L(\sigma(2))$, $\ldots$, no three languages in a row in that list are all the same, and (2) the set $\{L(\sigma(0)), L(\sigma(1)), L(\sigma(2)), \ldots\}$ is the set of all Turing-recognizable languages over the alphabet $\{0, 1\}$? What about if we change clause (1) to say no two languages in a row? (Hint: The answer is yes even for that, even though the “recursion theorem” would seem to prove that the answer is no—amaze your professors with this, as a CSC280 student once amazed me!)
Chapter 4, Part 1

Decidability
Decidable Problems About Regular Languages

The Acceptance Problem for DFA

Define $A_{DFA}$ to be:

$$\{ \langle B, w \rangle \mid B \text{ is a DFA that accepts input string } w \}.$$ 

**Theorem.** $A_{DFA}$ is decidable.

**Proof** A Turing machine can, given an input $x$, try to decode $x$ into a DFA $B$ and a string $w$. If the decoding is successful then it can test whether $B$ accepts $w$ by **simulating $B$ on $w$.**
How This Can Be Done

• After checking the legitimacy of encoding, our Turing machine writes on its second tape the input $w$ (as an encoded form).

• Our machine starts simulating $M$, using the second tape as the tape of $M$ by looking up information about $M$'s action in the first tape and using a tape symbol encoding scheme consistent with the input $x$.

When $M$ terminates, our machine terminates accordingly.
The Acceptance Problem for NFA

Define $A_{\text{NFA}}$ to be:

\[ \{ \langle B, w \rangle \mid B \text{ is an NFA that accepts input string } w \} \].

**Theorem.** $A_{\text{NFA}}$ is decidable.

**Proof**  Given an input $x$, try to decode $x$ into an NFA $B$ and a string $w$. If “successful” then:

1. Convert $B$ to a DFA $C$.
2. Run the machine for $A_{\text{DFA}}$ on $\langle C, w \rangle$. If the machine accepts, then accept; otherwise reject.
The Acceptance Problem for Regular Exp.

Define $A_{\text{REX}}$ to be:

$$\{\langle R, w \rangle \mid R \text{ is a regular expression that produces } w \}.$$  

**Theorem.** $A_{\text{REX}}$ is decidable.

**Proof**  Given an input $x$, try to decode $x$ into a regular expression $R$ and a string $w$. If “successful” then:

1. Convert $R$ to a DFA $C$.
2. Run the machine for $A_{\text{DFA}}$ on $\langle C, w \rangle$. If the machine accepts, then accept; otherwise reject.
The Emptiness Problem for DFA

Define $E_{\text{DFA}} = \{ \langle A \rangle \mid A \text{ is a DFA that accepts no string} \}$. 

**Theorem.** $E_{\text{DFA}}$ is decidable.

**Proof**  Given an input $x$, try to decode a DFA $A$ out of $x$. If “successful” then:

1. **Mark the start state** of $A$.
2. **Repeat until no new states are marked:**
   - Mark any unmarked state that has a transition from a marked state
3. Accept if **no final state is marked**; reject otherwise.
The Equivalence Problem for DFA

Define \( EQ_{\text{DFA}} \) to be:
\[
\{ \langle A, B \rangle \mid A \text{ and } B \text{ are DFA and accept the same language} \}\.
\]

**Theorem.** \( EQ_{\text{DFA}} \) is decidable.

**Proof** Given a string \( x \), try to decode \( x \) into a pair of DFAs \( A \) and \( B \). If “successful” then construct a DFA \( C \) that accepts the symmetric difference of \( L(A) \) and \( L(B) \),
\[
(L(A) \cap \overline{L(B)}) \cup (L(A) \cap \overline{L(B)}),
\]
and test the emptiness of \( L(C) \).
The Acceptance Problem for CFG

Define $A_{\text{CFG}} = \{ \langle G, w \rangle \mid G \text{ is a CFG that generates } w \}$.

**Theorem.** $A_{\text{CFG}}$ is decidable.

**Proof**  Given an input $x$, try to decode $x$ into a CFG $G$ and a string $w$. If “successful” then:

2. List all derivations with $2n - 1$ steps, where $n = |w|$.
3. If any of the listed derivations generate $w$, then accept; otherwise, reject.
The Acceptance Problem for CFG

Define $A_{\text{CFG}} = \{ \langle G, w \rangle \mid G \text{ is a CFG that generates } w \}$.

**Theorem.** $A_{\text{CFG}}$ is decidable.

**Proof** Given an input $x$, try to decode $x$ into a CFG $G$ and a string $w$. If “successful” then:


2. List all derivations with $2n - 1$ steps, where $n = |w|$.

3. If any of the listed derivations generate $w$, then accept; otherwise, reject.

Warning: The above proof is so terse that it contains an error. Can you see that and fix that? Hint: Think “boundary.” (See the next slide for the answer.)
**The Acceptance Problem for CFG**

Define $A_{CFG} = \{ \langle G, w \rangle \mid G \text{ is a CFG that generates } w \}$.

**Theorem.** $A_{CFG}$ is decidable.

**Proof**  Given an input $x$, try to decode $x$ into a CFG $G$ and a string $w$. If “successful” then:

2. List all derivations with $2n - 1$ steps, where $n = |w|$.
3. If any of the listed derivations generate $w$, then accept; otherwise, reject.

Warning: The above proof is so terse that it contains an error. Can you see that and fix that? Hint: Think “boundary.”

Answer: The proof above mishandles the empty string. For that, we need to check all length 1 derivations, not all length -1 derivations. (Don’t worry; the proof in SIP is fine!)
The Emptiness Problem for CFG

Define \( E_{\text{CFG}} = \{ \langle G \rangle \mid G \text{ is a CFG such that } L(G) = \emptyset \} \).

**Theorem.** \( E_{\text{CFG}} \) is decidable.

**Proof** Given \( x \), first try to decode a grammar \( G \) out of it. If “pass” then test the ability of generating terminal strings:

1. **Mark all the terminals.**
2. Repeat the following until no new symbols are marked:
   - Mark any variables \( A \) such that \( G \) has a production \( A \rightarrow w \) such that all symbols in \( w \) are *already* marked.
3. **Accept** if the start symbol is marked; **reject** otherwise.
Context-Free Languages are Decidable

**Theorem.** *Every context-free language is decidable.*

Simulation of a PDA may not halt.

**Proof** Use the machine $M$ for $A_{\text{CFG}}$. Let $G$ be a fixed CFG. The machine for $L(G)$, on input $w$,

1. run $\langle G, w \rangle$ on $M$, and
2. accepts if $M$ accepts and rejects otherwise.
Chapter 4, Part 2

The Halting Problem
The Halting Problem

Define $A_{TM} = \{ \langle M, w \rangle \mid M \text{ is a Turing machine and accepts } w \}$.

**Theorem.** $A_{TM}$ is not decidable.
The Halting Problem

Define $A_{TM} = \{ \langle M, w \rangle \mid M \text{ is a Turing machine and accepts } w \}$.

**Theorem.** $A_{TM}$ is not decidable.

From this theorem we obtain:

**Corollary.** $\overline{A_{TM}}$ is not Turing-recognizable, and thus, not decidable.
The Halting Problem

Define $A_{TM} = \{\langle M, w \rangle \mid M$ is a Turing machine and accepts $w \}$.

**Theorem.** $A_{TM}$ is not decidable.

From this theorem we obtain:

**Corollary.** $\overline{A_{TM}}$ is not Turing-recognizable, and thus, not decidable.

For this corollary we need the following fact.

**Fact.** A language $L$ is decidable if and only if both $L$ and $\overline{L}$ are Turing-recognizable.

**Proof of Corollary** $A_{TM}$ is clearly Turing-recognizable. If $A_{TM}$ were also Turing-recognizable, then by the Fact $A_{TM}$ would be decidable; but the theorem says that $A_{TM}$ is not decidable.
Proof of Fact  (Note: We also proved this in the Chap. 3, Part 2 slides)

**Proof of Fact**  \(\Rightarrow\) Let \(L\) be decidable and let \(M\) be a Turing machine that decides \(L\). By swapping \(q_{\text{accept}}\) and \(q_{\text{reject}}\) of \(M\) we get a Turing machine \(M'\) that decides \(\overline{L}\). So both \(L\) and \(\overline{L}\) are Turing-decidable, and thus, Turing-recognizable.
Proof of Fact (cont’d)

[⇐] Let $L$ and $\overline{L}$ be recognized by TMs $M_1$ and $M_2$, respectively. Define a two-tape machine $M$ that, on input $x$, does the following:

1. $M$ copies $x$ onto Tape 2.
2. $M$ repeats the following until either $M_1$ or $M_2$ accepts:
   - $M$ simulates one (more) step of $M_1$ on Tape 1 then one (more) step of $M_2$ on Tape 2.
3. $M$ accepts $x$ if either $M_1$ accepts $x$ or $M_2$ rejects $x$; $M$ rejects $x$ if either $M_2$ accepts $x$ or $M_1$ rejects $x$.

Then $M$ decides $L$ because for every $x$, at least one of the two machines halts on input $x$. □

Fact
Diagonalization

(For the rest of these Chapter 4, Part 2 slides, we’ll write \( \mathcal{N} \) but will actually mean \( \mathcal{N}^+ \), i.e., \{1, 2, 3, \ldots \}.)

A set \( S \) is **countable** if either it is finite or it has the same size as \( \mathcal{N} \); i.e., there is a **one-to-one, onto correspondence** between \( S \) and \( \mathcal{N} \) (i.e., there is a **bijection** from \( S \) to \( \mathcal{N} \)).
Simple Facts About the Countable

Let $\mathbb{Q}$ be the set of all positive rational numbers and let $\mathbb{R}$ be the set of all positive real numbers.

Fact. $\mathbb{Q}$ is countable while $\mathbb{R}$ is not.
Proving the Fact

Proof  Each member of $Q$ is expressed as a fraction $\frac{m}{n}$ such that $m, n \in \mathcal{N}$ and $\gcd(m, n) = 1$.

So we have only to come up with a bijection from $\mathcal{N}$ to the set $\left\{ \frac{m}{n} \mid m, n \geq 1 \land \gcd(m, n) = 1 \right\}$. 
**$\mathbb{Q}$ is countable**

We will visit all the grid points in the first quadrant of the $xy$-plane.

For $p = 2, 3, 4, \ldots$, visit the points $(x, y)$ on the line $x + y = p$

$$(1, p - 1), (2, p - 2), \ldots, (p - 1, 1)$$

and collect only those points at which $x$ and $y$ are relatively prime to each other and both are at least one.
The numbers show the visiting order (not the bijection, e.g., the number 5 is (2, 2) and since 2 and 2 are not relatively prime this point is skipped in the bijection; so in the bijection, 1 maps to (1, 1) and 5 maps to (3, 1), which is the point numbered 6 above).
**R is not countable**

Assume, by way of contradiction, that \( \mathbb{R} \) is countable. Then the real numbers can be enumerated as \( r_1, r_2, \ldots \) (where no \( r_i \) ends with an infinite string of adjacent 9’s).

Define \( x \) to be the number between 0 and 1 defined as follows:

(*) For each \( i \in \mathbb{N} \), the \( i \)th digit of \( x \) after the decimal point is that of \( r_i \) plus 1 (modulo 10).

For example, if \( r_1 = 3.\overline{14159} \), \( r_2 = 2.\overline{23606} \), \( r_3 = 1.\overline{73205} \), \ldots, then \( x = .243 \ldots \),
\[ \mathcal{R} \text{ is not countable} \]

Assume, by way of contradiction, that \( \mathcal{R} \) is countable. Then the real numbers can be enumerated as \( r_1, r_2, \ldots \) (where no \( r_i \) ends with an infinite string of adjacent 9’s).

Define \( x \) to be the number between 0 and 1 defined as follows:

\[ (*) \text{ For each } i \in \mathbb{N}, \text{ the } i\text{th digit of } x \text{ after the decimal point is that of } r_i \text{ plus } 1 \text{ (modulo } 10). \]

For example, if \( r_1 = 3.14159, r_2 = 2.23606, r_3 = 1.73205, \ldots, \) then \( x = .243 \ldots, \)

This \( x \) is real. By assumption there must exist a \( k \) such that \( r_k \) is \( x \). However, by definition, the \( k \)-th digit of \( r_k \) is different from that of \( x \), a contradiction. (WARNING: That claim isn’t quite true, as we’ll discuss on the next slide! Do you see what the potential problem is here?) Thus, \( \mathcal{R} \) is not countable.
R is not countable

Assume, by way of contradiction, that R is countable. Then the real numbers can be enumerated as \( r_1, r_2, \ldots \) (where no \( r_i \) ends with an infinite string of adjacent 9’s).

Define \( x \) to be the number between 0 and 1 defined as follows:

(*) For each \( i \in \mathbb{N} \), the \( i \)th digit of \( x \) after the decimal point is that of \( r_i \) plus 1 (mod 10).

For example, if \( r_1 = 3.14159 \), \( r_2 = 2.23606 \), \( r_3 = 1.73205 \), . . . , then \( x = .243 \ldots \).

This \( x \) is real. By assumption there must exist a \( k \) such that \( r_k \) is \( x \). However, by definition, the \( k \)-th digit of \( r_k \) is different from that of \( x \), a contradiction.

Hold on! What if \( x \) ends with an infinite string of adjacent 9’s? Then it MIGHT equal some \( r_k \) even though it differs from that \( r_k \) in the \( k \)th digit. Here is a clever patch (suggested by a previous CSC280 student, Doug Miller): Infinitely many \( r_i \)’s have no 8’s anywhere, and so \( x \) cannot end with an infinite string of 9’s! Here is an alternate patch (suggested by a previous CSC280 student, Chris Frederickson): Alter line (*) above to now change 0 to 1 and “nonzero” to 0; this ensures \( x \) won’t have any 9’s! (NOTE: The proof found in SIP is clean; you don’t have to worry about that proof.)

Thus, \( R \) is not countable.
An Immediate Application of Diagonalization

Corollary. There is a language that is not Turing-recognizable.
Corollary. There is a language that is not Turing-recognizable.

Proof Consider all Turing machines whose input alphabet is \{0\}. Since each Turing machine can be encoded as a word of finite length, this set of Turing machines is countable.

Let $M_1, M_2, \ldots$ be the enumeration of all Turing machines in this set.
An Immediate Application of Diagonalization

Corollary. There is a language that is not Turing-recognizable.

Proof Consider all Turing machines whose input alphabet is \{0\}.

Since each Turing machine can be encoded as a word of finite length, this set of Turing machines is countable.

Let $M_1, M_2, \ldots$ be the enumeration of all Turing machines in this set.

Define $L = \{0^i \mid M_i \text{ on input } 0^i \text{ does not accept} \}$. 
Define $L = \{0^i \mid M_i \text{ on input } 0^i \text{ does not accept} \}$.

There is no machine $M_k$ that recognizes $L$. Why?

If there were such a $k$, then we have by definition of $L$

$$0^k \in L \Leftrightarrow M_k \text{ does not accept } 0^k.$$  

The latter condition is equivalent to $0^k \not\in L(M_k)$. But by the definition of $k$, $L(M_k) = L$ (!!), so $0^k \not\in L(M_k)$ is equivalent to $0^k \not\in L$. Thus, we have

$$0^k \in L \Leftrightarrow 0^k \not\in L,$$

a contradiction.
Proof of Theorem ($A_{TM}$ is not decidable)

Assume that $A_{TM}$ is decidable. Let $T$ be a Turing machine that decides $A_{TM}$. Define $D$ to be a machine that, on input $w$, does this:

1. Check whether $w$ is a legal encoding of some Turing machine, say $M$. If not, immediately reject $w$.
2. Simulate $T$ on $\langle M, \langle M \rangle \rangle$.
3. If $T$ accepts, then reject; otherwise, accept.

Since $T$ decides $A_{TM}$ by assumption, $T$ always halts; thus by the above program, so does $D$. For every Turing machine $M$,

\[ D \text{ accepts } \langle M \rangle \iff M \text{ does not accept } \langle M \rangle \]

With $M = D$, we have

\[ D \text{ accepts } \langle D \rangle \iff D \text{ does not accept } \langle D \rangle. \]

This is a contradiction.
Chapter 5, Part 1

Reducibility
The Halting Problem

Based on undecidability of one language, \( A \), undecidability of another language, \( B \), can sometimes be shown.
The Halting Problem

Based on undecidability of one language, $A$, undecidability of another language, $B$, can sometimes be shown.

We will use the concept of a reduction for this purpose.

This involves showing that one could build a Turing machine that decides $A$ assuming that there is a Turing machine for deciding $B$. 
Assumption About Coding

The set of all possible inputs to the machine we will build as well as that to the machine we assume to exist (that is, $\Sigma^*$ of the input alphabet $\Sigma$) may contain strings not encoding any meaningful objects.

For completeness our reduction has to handle such strings, but since the way we handle them is very simple (either accept all such strings or reject all such strings depending on how $A$ or $B$ is defined), we will simply ignore such strings.
The Halting Problem

\[ \text{HALT}_{\text{TM}} = \{ \langle M, w \rangle \mid M \text{ is a TM and halts on input } w \} \].

**Theorem.** \( \text{HALT}_{\text{TM}} \) is undecidable.
The Halting Problem

\[ \text{HALT}_{TM} = \{ \langle M, w \rangle \mid M \text{ is a TM and halts on input } w \}. \]

**Theorem.** \( \text{HALT}_{TM} \) is undecidable.

**Proof** (Proof 1: Contradiction-based) Assume that there is a Turing machine \( R \) that decides \( \text{HALT}_{TM} \).

We then would be able to construct a Turing machine \( S \) that decides

\[ A_{TM} = \{ \langle M, w \rangle \mid M \text{ is a Turing machine and accepts } w \}, \]

which is, however, known to be undecidable.
Logic Behind the Construction

- To handle $A_{TM}$, we want to know whether a Turing machine $M$ accepts on input $w$.
- A natural direct approach to finding an answer to that will be to simulate $M$ on $w$, but the simulation may not stop.
  1. $M$ accepts $w \rightarrow \text{“yes” … ACCEPT}$
  2. $M$ rejects $w \rightarrow \text{“no” … REJECT}$
  3. $M$ on $w$ never stops $\rightarrow \text{“no” … PROBLEM}$
- If there is a TM machine that tells whether on a given input we will fall into that horrible, evil Case 3, we can use that TM to avoid the problem.
Our Turing Machine $S$ for $A_{TM}$

Let the input $x = \langle M, w \rangle$.

1. Simulate $R$ on $x$.
   
   **Note:** $R$ decides $HALT_{TM}$ and so $R$ on $x$ halts.

2. If $R$ rejects $x$, reject $x$.
   
   **Note:** This corresponds to Case 3 - the problematic case.

3. If $R$ accepts $x$, simulate $M$ on $w$, and accept if and only if $M$ accepts.
   
   **Note:** Here we are distinguishing between Case 1 and Case 2.

This machine would correctly decide $A_{TM}$. So a contradiction has been reached to our earlier result that $A_{TM}$ is not decidable.
Alternative Approach  (Proof 2: Slightly More Reduction-like in Flavor, though Not Really)

- $R$ is a Turing machine that purportedly decides the halting problem.
- Given a machine $M$ we can modify its code to create a new machine $M'$ such that $M'$ enters a nonaccepting infinite loop instead of rejecting. Then we have:
  1. $M$ accepts $w$ ... $M'$ on $w$ accepts.
  2. $M$ rejects $w$ ... $M'$ on $w$ does not halt.
  3. $M$ on $w$ never stops ... $M'$ on $w$ does not halt.
- Then $M$ on $w$ accepts if and only if $M'$ on $w$ halts.
Alternative Approach (Proof 2: Slightly More Reduction-like in Flavor, though Not Really)

1. From $M$ construct a new Turing $M'$ that simulates $M$ and instead of entering $q_{\text{reject}}$, $M'$ enters an infinite loop.
2. Simulate $R$ on $\langle M', w \rangle$.
3. Accept if $R$ accepts and reject otherwise.
A Convention About High-Level Descriptions—Part I

In constructions, we sometimes internally build a TM and speak about it. In doing so, we often describe it in high-level pseudocode. This raises (as a side effect of Sipser’s sort of kinky approach to Turing machine behavior) a question as to what the action is of the actual underlying TM being described, if the high-level code simply reaches its very, very end without having itself, or within a machine it was simulating, halted by rejecting or accepting. (Should we say the underlying TM loops since it neither accepted not rejected, even though the pseudocode seems itself to halt?? Yuck!)

Sipser is silent on this because he by hand just avoids having it happen in any of his constructions. (And that is a good thing, but we can’t make it a “syntactic” validity requirement, because doing so would make validity testing itself become undecidable!)
A Convention About High-Level Descriptions—Part II

Our convention will simply be to view hitting the very end of your code in a high-level pseudocode description to be, when translated into action TM behavior, (halting and) rejecting. You can view the program’s final “}” as, by this convention, being a “Reject;}”.

So for example, if we write in a proof/construction:

Let $M'$ be the TM that on arbitrary input $y$ does:
{If $y = \epsilon$ then accept;}

Then by our convention we’ll take that pseudocode as describing a TM $M'$ that on input $\epsilon$ accepts (thus surely halts), and that on all other inputs rejects (and thus surely halts).

But nonetheless it is most elegant and clear to write only pseudocode that will simply never smash into its final “}”.
The Emptiness Problem

Define $E_{TM} = \{ \langle M \rangle \mid M$ is a TM and $L(M) = \emptyset \}$. 

**Theorem.** $E_{TM}$ is undecidable.
The Emptiness Problem

Define $E_{TM} = \{ \langle M \rangle \mid M \text{ is a TM and } L(M) = \emptyset \}$.

**Theorem.** $E_{TM}$ is undecidable.

**Proof** Assume there is a TM $R$ that decides $E_{TM}$. We’ll construct a TM $S$ that decides $A_{TM}$. 
Algorithm of $S$ for $A_{TM}$

1. Input $x$ is $\langle M, w \rangle$ for some $M$ and $w$.
   **We want to know whether $M$ on $w$ accepts.**

2. Construct a Turing machine $M_1$:
   $M_1$ erases its input $y$, reproduces $w$ on the input tape, and then enters an (immersive) simulation of $M$. That is, $M_1$ ultimately behaves as $M$ on input $w$, regardless of the input to $M_1$. We have:

   $$L(M_1) = \begin{cases} 
   \Sigma^* & \text{if } M \text{ accepts } w \\
   \emptyset & \text{if } M \text{ does not accept } w.
   \end{cases}$$

   Also, we have $\langle M_1 \rangle \in E_{TM}$ if and only if $L(M_1) = \emptyset$. $R$ purportedly decides $E_{TM}$.

3. Simulate $R$ on $\langle M_1 \rangle$. Accept if $R$ rejects and reject otherwise.
Testing Whether a TM Accepts a Regular Language

\[ \text{REGULAR}_{\text{TM}} = \{ \langle M \rangle \mid M \text{ is a TM and } L(M) \text{ is regular} \}. \]

**Theorem.** \( \text{REGULAR}_{\text{TM}} \) is undecidable.
Testing Whether a TM Accepts a Regular Language

\[ REGULAR_{TM} = \{ \langle M \rangle \mid M \text{ is a TM and } L(M) \text{ is regular} \} \].

**Theorem.** \( REGULAR_{TM} \) is undecidable.

**Proof**  Assume there is a TM \( R \) that decides \( REGULAR_{TM} \). We’ll construct a TM \( S \) that decides \( A_{TM} \).
Testing Whether a TM Accepts a Regular Language

Input \( x = \langle M, w \rangle \).

1. Let \( \Sigma \) be the input alphabet of \( M \). If \( \Sigma \) has only one symbol add another symbol.

2. Choose two symbols, say \( a \) and \( b \), from \( \Sigma \).

3. Construct a machine \( M_1 \) that on input \( y \) behaves as follow:
   (a) If \( y = a^n b^n \) for some \( n \geq 0 \), accept.
   (b) Otherwise, erase \( y \), reproduce \( w \), simulate \( M \) on \( w \).

We have:

\[
L(M_1) = \begin{cases} 
  \Sigma^* & \text{if } M \text{ accepts } w \\
  \{a^n b^n \mid n \geq 0\} & \text{if } M \text{ does not accept } w
\end{cases}
\]

Also, \( \Sigma^* \) is regular and \( \{a^n b^n \mid n \geq 0\} \) is nonregular.

4. Simulate \( R \) on \( \langle M_1 \rangle \). Accept if and only if \( R \) accepts.
Testing Equivalence Between TMs

Define $E_{\text{TM}} = \{⟨M_1, M_2⟩ \mid \text{both } M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2)\}$.

**Theorem.** $E_{\text{TM}}$ is undecidable.
Testing Equivalence Between TMs

Define $E_{\text{TM}} = \{ \langle M_1, M_2 \rangle \mid \text{both } M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2) \}$. 

**Theorem.** $E_{\text{TM}}$ is undecidable.

**Proof** Assume there is a TM $R$ that decides $E_{\text{TM}}$. We’ll construct a TM $S$ that decides $A_{\text{TM}}$.

Build $M_1$ just as in the previous proof. In addition, construct a machine that accepts $\Sigma^*$; call that machine $M_2$. Then we have

$$L(M_1) = L(M_2) \text{ if and only if } M \text{ accepts } w.$$ 

We simulate $R$ on $\langle M_1, M_2 \rangle$. Accept if $R$ accepts and reject otherwise.
Mapping Reducibility

(This generally corresponds to Chapter 5.3 of SIP.)
Computable Functions

A function \( f : \Sigma^* \rightarrow \Sigma^* \) is **computable** if there exists a Turing machine \( M \) such that for every \( x \in \Sigma^* \), \( M \) on \( x \) halts with just \( f(x) \) on its tape.

**Example:** Let \( \Sigma \) be a fixed alphabet. Define \( f : \Sigma^* \rightarrow \Sigma^* \) as follows:

- If \( w = \langle M \rangle \) for some Turing machine, then \( f(w) = \langle M' \rangle \) where \( M' \) is \( M \) with \( q_{\text{accept}} \) and \( q_{\text{reject}} \) swapped.
- Otherwise, \( f(w) = w \).

Then \( f \) is computable.
Mapping Reductions, a.k.a. Many-One Reductions

A language $A \subseteq \Sigma^*$ is mapping reducible (or, equivalently, is (recursive) many-one reducible) to $B \subseteq \Sigma^*$ (often written as $A \leq_m B$) if there exists a computable function $f : \Sigma^* \to \Sigma^*$ such that for every $x \in \Sigma^*$, $x \in A$ if and only if $f(x) \in B$.

Namely, the function $f$ maps members of $A$ to members of $B$ and nonmembers of $A$ to nonmembers of $B$. The function $f$ is said to be a reduction from $A$ to $B$.

Can you draw a clear picture of what is up here?

A particularly elegant way to do undecidability proofs is to note that if $A \leq_m B$ and $A$ is not decidable, then $B$ is not decidable, and then to apply that. We then prove a set is undecidable by $\leq_m$-reducing to it from some already-known-undecidable set!
Properties About Mapping Reducibility

**Theorem.** If $A \leq_m B$ and $B$ is decidable then $A$ is decidable.

**Proof** Let $A \leq_m B$ be witnessed by a Turing machine $R$ that computes a mapping reduction $f$ from $A$ to $B$.

Suppose $B$ is decided by a Turing machine $M$. Construct a new Turing machine $N$:

1. On input $x$, simulate $R$ on $x$ to compute $f(x)$.
2. **Simulate $M$ on $f(x)$**. Accept if $M$ accepts and reject if $M$ rejects.

Then $N$ decides $A$. 

---

CSC280, Chapter 5, Parts 2/3 © 2012 Mitsunori Ogihara. Edits/updates by Lane A. Hemaspaandra, 20{13,14,17}.
Corollary. If $A \leq_m B$ and $A$ is undecidable then $B$ is undecidable.

Theorem. If $A \leq_m B$ and $B$ is Turing-recognizable then $A$ is Turing-recognizable.

Corollary. If $A \leq_m B$ and $A$ is not Turing-recognizable then $B$ is not Turing-recognizable.

Can you see why each of these holds?
Recall that $EQ_{TM} = \{\langle M_1, M_2 \rangle \mid M_1 \text{ and } M_2 \text{ are Turing machines and } L(M_1) = L(M_2)\}$.

**Theorem.** $EQ_{TM}$ is neither Turing-recognizable nor co-Turing-recognizable.
Proof

Show that $A_{TM}$ is mapping reducible to $EQ_{TM}$ as well as to $\overline{EQ_{TM}}$. Let $s \in EQ_{TM}$ and $t \in \overline{EQ_{TM}}$ be fixed.

Reduction to $EQ_{TM}$

- If $x$ is of the form $\langle M, w \rangle$, then $f(x) = \langle M_1, M_2 \rangle$, where
  - $M_1$ accepts every input; and
  - $M_2$ first simulates $M$ on $w$ and accepts its own input if $M$ accepts.
- Otherwise, $f(x) = t$.

$f$ is computable, and for every $x$, $x \in A_{TM}$ if and only if $f(x) \in EQ_{TM}$. 
Proof (cont’d)

Reduction to $\overline{EQ_{TM}}$

- If $x$ is of the form $\langle M, w \rangle$, then $g(x) = \langle M_1, M_2 \rangle$, where
  - $M_1$ rejects every input; and
  - $M_2$ first simulates $M$ on $w$ and accepts its own input if $M$ accepts.

- Otherwise, $g(x) = s$.

$g$ is computable and for every $x$, $x \in A_{TM}$ if and only if $g(x) \notin \overline{EQ_{TM}}$.

Thus, $A_{TM} \leq_m \overline{EQ_{TM}}$ and $A_{TM} \leq_m \overline{EQ_{TM}}$, and so by the second corollary from a few slides ago, the theorem is proven.
Chapter “P and NP”

P and NP

For our quick coverage of P and NP, we’ll use (a subset of) my handwritten slides used in the UR graduate continuation of this course!

And then if time permits we’ll have some lecture(s) on P and NP-completeness in use, perhaps taking as our focus area the complexity of protecting (and attacking) elections.