CSE 311: Foundations of Computing

Topic 10: Finite State Machines
The set of binary strings with a 1 in the 3rd position from the start.
The set of binary strings with a 1 in the 3\textsuperscript{rd} position from the end
Adding Output to Finite State Machines

• So far we have considered finite state machines that just accept/reject strings
  – called “Deterministic Finite Automata” or DFAs

• Now we consider finite state machines with output
  – These are the kinds used as controllers
Enter 15 cents in dimes or nickels
Press S or B for a candy bar
Adding additional “unexpected” transitions to cover all symbols for each state
Recall: Finite State Machines

- States
- Transitions on input symbols
- Start state and final states
- The “language recognized” by the machine is the set of strings that reach a final state from the start state.

<table>
<thead>
<tr>
<th>Old State</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>s₀</td>
<td>s₀</td>
<td>s₁</td>
</tr>
<tr>
<td>s₁</td>
<td>s₀</td>
<td>s₂</td>
</tr>
<tr>
<td>s₂</td>
<td>s₀</td>
<td>s₃</td>
</tr>
<tr>
<td>s₃</td>
<td>s₃</td>
<td>s₃</td>
</tr>
</tbody>
</table>
Recall: Finite State Machines

• Each machine designed for strings over some fixed alphabet $\Sigma$.

• Must have a transition defined from each state for every symbol in $\Sigma$.

<table>
<thead>
<tr>
<th>Old State</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_0$</td>
<td>$s_0$</td>
<td>$s_1$</td>
</tr>
<tr>
<td>$s_1$</td>
<td>$s_0$</td>
<td>$s_2$</td>
</tr>
<tr>
<td>$s_2$</td>
<td>$s_0$</td>
<td>$s_3$</td>
</tr>
<tr>
<td>$s_3$</td>
<td>$s_3$</td>
<td>$s_3$</td>
</tr>
</tbody>
</table>
State Minimization

- Many FSMs (DFAs) for the same problem
- Take a given FSM and try to reduce its state set by combining states
  - Algorithm will always produce the unique minimal equivalent machine (up to renaming of states) but we won’t prove this
State Minimization Algorithm

• Put states into groups

• Try to find groups that can be collapsed into one state
  – states can keep track of information that isn’t necessary to determine whether to accept or reject

• Group states together until we can prove that collapsing them can change the accept/reject result
  – find a specific string $x$ such that:
    starting from state A, following edges according to $x$ ends in accept
    starting from state B, following edges according to $x$ ends in reject
  – (algorithm below could be modified to show these strings)
State Minimization Algorithm

1. Put states into groups based on their outputs (whether they accept or reject)
State Minimization Algorithm

1. Put states into groups based on their outputs (whether they accept or reject)

2. Repeat the following until no change happens
   a. If there is a symbol $s$ so that not all states in a group $G$ agree on which group $s$ leads to, split $G$ into smaller groups based on which group the states go to on $s$

3. Finally, convert groups to states
State Minimization Example

Put states into groups based on their outputs (or whether they accept or reject)

<table>
<thead>
<tr>
<th>present state</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>S0</td>
<td>S1</td>
<td>S2</td>
<td>S3</td>
<td>1</td>
</tr>
<tr>
<td>S1</td>
<td>S0</td>
<td>S3</td>
<td>S1</td>
<td>S5</td>
<td>0</td>
</tr>
<tr>
<td>S2</td>
<td>S1</td>
<td>S3</td>
<td>S2</td>
<td>S4</td>
<td>1</td>
</tr>
<tr>
<td>S3</td>
<td>S1</td>
<td>S0</td>
<td>S4</td>
<td>S5</td>
<td>0</td>
</tr>
<tr>
<td>S4</td>
<td>S0</td>
<td>S1</td>
<td>S2</td>
<td>S5</td>
<td>1</td>
</tr>
<tr>
<td>S5</td>
<td>S1</td>
<td>S4</td>
<td>S0</td>
<td>S5</td>
<td>0</td>
</tr>
</tbody>
</table>
State Minimization Example

Put states into groups based on their outputs (or whether they accept or reject)
State Minimization Example

<table>
<thead>
<tr>
<th>Present state</th>
<th>Next state</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>S0</td>
<td>1</td>
</tr>
<tr>
<td>S1</td>
<td>S0 S3 S1 S5</td>
<td>0</td>
</tr>
<tr>
<td>S2</td>
<td>S1 S3 S2 S4</td>
<td>1</td>
</tr>
<tr>
<td>S3</td>
<td>S1 S0 S4 S5</td>
<td>0</td>
</tr>
<tr>
<td>S4</td>
<td>S0 S1 S2 S5</td>
<td>1</td>
</tr>
<tr>
<td>S5</td>
<td>S1 S4 S0 S5</td>
<td>0</td>
</tr>
</tbody>
</table>

Put states into groups based on their outputs (or whether they accept or reject)

If there is a symbol $s$ so that not all states in a group $G$ agree on which group $s$ leads to, split $G$ based on which group the states go to on $s$
Put states into groups based on their outputs (or whether they accept or reject).

If there is a symbol $s$ so that not all states in a group $G$ agree on which group $s$ leads to, split $G$ based on which group the states go to on $s$. 

---

*State Minimization Example*

<table>
<thead>
<tr>
<th>present state</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>S0</td>
<td>S1</td>
<td>S2</td>
<td>S3</td>
<td>1</td>
</tr>
<tr>
<td>S1</td>
<td>S0</td>
<td>S3</td>
<td>S1</td>
<td>S5</td>
<td>0</td>
</tr>
<tr>
<td>S2</td>
<td>S1</td>
<td>S3</td>
<td>S2</td>
<td>S4</td>
<td>1</td>
</tr>
<tr>
<td>S3</td>
<td>S1</td>
<td>S0</td>
<td>S4</td>
<td>S5</td>
<td>0</td>
</tr>
<tr>
<td>S4</td>
<td>S0</td>
<td>S1</td>
<td>S2</td>
<td>S5</td>
<td>1</td>
</tr>
<tr>
<td>S5</td>
<td>S1</td>
<td>S4</td>
<td>S0</td>
<td>S5</td>
<td>0</td>
</tr>
</tbody>
</table>

State transition table
State Minimization Example

State transition table

<table>
<thead>
<tr>
<th>present state</th>
<th>next state</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>S0 S1 S2 S3</td>
<td>1</td>
</tr>
<tr>
<td>S1</td>
<td>S0 S3 S1 S5</td>
<td>0</td>
</tr>
<tr>
<td>S2</td>
<td>S1 S3 S2 S4</td>
<td>1</td>
</tr>
<tr>
<td>S3</td>
<td>S1 S0 S4 S5</td>
<td>0</td>
</tr>
<tr>
<td>S4</td>
<td>S0 S1 S2 S5</td>
<td>1</td>
</tr>
<tr>
<td>S5</td>
<td>S1 S4 S0 S5</td>
<td>0</td>
</tr>
</tbody>
</table>

Put states into groups based on their outputs (or whether they accept or reject)

If there is a symbol $s$ so that not all states in a group $G$ agree on which group $s$ leads to, split $G$ based on which group the states go to on $s$
State Minimization Example

Put states into groups based on their outputs (or whether they accept or reject)

If there is a symbol $s$ so that not all states in a group $G$ agree on which group $s$ leads to, split $G$ based on which group the states go to on $s$
State Minimization Example

Put states into groups based on their outputs (or whether they accept or reject)

If there is a symbol $s$ so that not all states in a group $G$ agree on which group $s$ leads to, split $G$ based on which group the states go to on $s$
State Minimization Example

State transition table

<table>
<thead>
<tr>
<th>Present State</th>
<th>Next State</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>S0 S1 S2 S3</td>
<td>1</td>
</tr>
<tr>
<td>S1</td>
<td>S0 S3 S1 S5</td>
<td>0</td>
</tr>
<tr>
<td>S2</td>
<td>S1 S3 S2 S4</td>
<td>1</td>
</tr>
<tr>
<td>S3</td>
<td>S1 S0 S4 S5</td>
<td>0</td>
</tr>
<tr>
<td>S4</td>
<td>S0 S1 S2 S5</td>
<td>1</td>
</tr>
<tr>
<td>S5</td>
<td>S1 S4 S0 S5</td>
<td>0</td>
</tr>
</tbody>
</table>

Finally convert groups to states:

Can combine states S0-S4 and S3-S5.

In table replace all S4 with S0 and all S5 with S3
Minimized Machine

**State Transition Table**

<table>
<thead>
<tr>
<th>Present State</th>
<th>Next State</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>S0 S1 S2 S3</td>
<td>1</td>
</tr>
<tr>
<td>S1</td>
<td>S0 S3 S1 S3</td>
<td>0</td>
</tr>
<tr>
<td>S2</td>
<td>S1 S3 S2 S0</td>
<td>1</td>
</tr>
<tr>
<td>S3</td>
<td>S1 S0 S0 S3</td>
<td>0</td>
</tr>
</tbody>
</table>

**State Transition Diagram**
The set of all binary strings with \# of 1's \equiv \# of 0's (mod 2).
A Simpler Minimization Example

Split states into accept/reject groups

Every symbol causes the DFA to go from one group to the other so neither group needs to be split
Minimized DFA

- The set of all binary strings with # of 1’s ≡ # of 0’s (mod 2).
- = The set of all binary strings with even length.
The Characters

\[
\begin{align*}
\text{REs} & \subseteq \text{CFGs} \\
\text{DFAs} & \subseteq \text{NFAs}
\end{align*}
\]
Nondeterministic Finite Automata (NFA)

- Graph with start state, final states, edges labeled by symbols (like DFA) but
  - Not required to have exactly 1 edge out of each state labeled by each symbol—can have 0 or >1
  - Also can have edges labeled by empty string $\varepsilon$

- **Definition:** $x$ is in the language recognized by an NFA if and only if some valid execution of the machine gets to an accept state
What language does this NFA accept?
What language does this NFA accept?

\[ 10(10)^* \cup 111(0 \cup 1)^* \]
NFA $\varepsilon$-moves

Diagram:

- States: $s_0$, $s_1$, $t_0$, $t_1$, $t_2$, and $q$.
- Transitions:
  - $s_0$ to $s_1$: $2$
  - $s_0$ to $s_0$: $\varepsilon$
  - $s_1$ to $s_1$: $0,1$
  - $t_0$ to $t_0$: $\varepsilon$
  - $t_0$ to $t_1$: $0$
  - $t_0$ to $t_2$: $1$
  - $t_1$ to $t_0$: $1$
  - $t_1$ to $t_1$: $\varepsilon$
  - $t_1$ to $t_2$: $1$
  - $t_2$ to $t_0$: $2$
  - $t_2$ to $t_2$: $2$
  - $t_2$ to $t_1$: $1$
  - $q$ to $s_0$: $0,1$
NFA \( \varepsilon \)-moves

Strings over \{0,1,2\} w/ even # of 2’s OR sum to 0 mod 3
NFA for set of binary strings with a 1 in the 3\textsuperscript{rd} position from the end
NFA for set of binary strings with a 1 in the 3\textsuperscript{rd} position from the end
Compare with the smallest DFA

[Diagram of a DFA with states s3, s2, s1, s0 and transitions labeled with 0, 1, and 0,1]
Three ways of thinking about NFAs

• Perfect guesser: The NFA has input $x$ and whenever there is a choice of what to do it magically guesses a good one (if one exists)

• Parallel exploration: The NFA computation runs all possible computations on $x$ step-by-step at the same time in parallel

• Outside observer: Is there a path labeled by $x$ from the start state to some accepting state?
Compare with the smallest DFA
Parallel Exploration view of an NFA

Input string 0101100
Three ways of thinking about NFAs

• Perfect guesser: The NFA has input $x$ and whenever there is a choice of what to do it magically guesses a good one (if one exists)

• Parallel exploration: The NFA computation runs all possible computations on $x$ step-by-step at the same time in parallel

• Outside observer: Is there a path labeled by $x$ from the start state to some accepting state?
Path Labels

Def: The label of path $v_0, v_1, ..., v_n$ is the concatenation of the labels of the edges $(v_0, v_1), (v_1, v_2), ..., (v_{n-1}, v_n)$

Example: The label of path $s_0, s_1, s_2, s_0, s_0$ is 1100
Deterministic Finite Automata (DFA)

- **Theorem:** \( x \) is in the language recognized by an DFA if and only if \( x \) labels a path from the start state to some final state.

- Path \( v_0, v_1, \ldots, v_n \) with \( v_0 = s_0 \) and label \( x \) describes a correct simulation of the DFA on input \( x \)
  - i-th step must match the i-th character of \( x \)
Nondeterministic Finite Automata (NFA)

- Graph with start state, final states, edges labeled by symbols (like DFA) but
  - Not required to have exactly 1 edge out of each state labeled by each symbol—can have 0 or >1
  - Can also have edges labeled by empty string ε

- **Theorem:** \( x \) is in the language recognized by an NFA if and only if \( x \) labels some path from the start state to an accepting state

![NFA Diagram]

\[s_0 \xrightarrow{1} s_1 \xrightarrow{1} s_2 \xrightarrow{1} s_3\]
Summary of NFAs

• Generalization of DFAs
  – drop two restrictions of DFAs
  – every DFA is an NFA

• Seem to be more powerful
  – designing is easier than with DFAs

• Seem related to regular expressions
The story so far...

REs \subseteq CFGs

DFAs \subseteq NFAs
Theorem: For any set of strings (language) $A$ described by a regular expression, there is an NFA that recognizes $A$.

Proof idea: Structural induction based on the recursive definition of regular expressions...
Regular Expressions over $\Sigma$

• **Basis:**
  – $\varepsilon$ is a regular expression
  – $a$ is a regular expression for any $a \in \Sigma$

• **Recursive step:**
  – If $A$ and $B$ are regular expressions, then so are:
    
    $A \cup B$
    
    $AB$
    
    $A^*$
Base Case

• Case $\varepsilon$:  

• Case $a$: 
Base Case

- Case ε:

- Case a:
Base Case

- Case $\varepsilon$:

- Case $a$: 
Regular Expressions over $\Sigma$

• **Basis:**
  - $\varepsilon$ is a regular expression
  - $a$ is a regular expression for any $a \in \Sigma$

• **Recursive step:**
  - If $A$ and $B$ are regular expressions, then so are:
    - $A \cup B$
    - $AB$
    - $A^*$
Inductive Hypothesis

• Suppose that for some regular expressions $A$ and $B$ there exist NFAs $N_A$ and $N_B$ such that $N_A$ recognizes the language given by $A$ and $N_B$ recognizes the language given by $B$
Inductive Step

Case $A \cup B$: 

\[ N_A \]

\[ N_B \]
Inductive Step

Case $A \cup B$:
Inductive Step

Case AB:
Inductive Step

Case AB:
Inductive Step

Case A*
Inductive Step

Case $A^*$
Build an NFA for \((01 \cup 1)^*0\)
Solution

\[(01 \cup 1)^*0\]
The story so far...

REs ⊆ CFGs

DFAs ⊆ NFAs
NFAs and DFAs

Every DFA is an NFA
  – DFAs have requirements that NFAs don’t have

Can NFAs recognize more languages?
NFAs and DFAs

Every DFA is an NFA
  – DFAs have requirements that NFAs don’t have

Can NFAs recognize more languages?  No!

Theorem: For every NFA there is a DFA that recognizes exactly the same language
Three ways of thinking about NFAs

- Perfect guesser: The NFA has input $x$ and whenever there is a choice of what to do it magically guesses a good one (if one exists)

- Parallel exploration: The NFA computation runs all possible computations on $x$ step-by-step at the same time in parallel

- Outside observer: Is there a path labeled by $x$ from the start state to some final state?
Parallel Exploration view of an NFA

Input string 0101100

---

Diagram of an NFA with states s₃, s₂, s₁, and s₀. The graph shows transitions on inputs 0, 1, and the initial state is s₃.
Conversion of NFAs to a DFAs

• Construction Idea:
  – The DFA keeps track of ALL states reachable in the NFA along a path labeled by the input so far (Note: not all *paths*; all *last states* on those paths.)
  – There will be one state in the DFA for each *subset* of states of the NFA that can be reached by some string
Conversion of NFAs to a DFAs

New start state for DFA

– The set of all states reachable from the start state of the NFA using only edges labeled $\varepsilon$
Conversion of NFAs to a DFAs

For each state of the DFA corresponding to a set $S$ of states of the NFA and each symbol $s$

- Add an edge labeled $s$ to state corresponding to $T$, the set of states of the NFA reached by
  - starting from some state in $S$, then
  - following one edge labeled by $s$, and
  - then following some number of edges labeled by $\varepsilon$
- $T$ will be $\emptyset$ if no edges from $S$ labeled $s$ exist
Conversion of NFAs to a DFAs

Final states for the DFA

– All states whose set contain some final state of the NFA

NFA

DFA

States:
- a
- b
- c
- e

Final States: a, b, c, e
Example: NFA to DFA
Example: NFA to DFA

NFA

DFA
Example: NFA to DFA

NFA

DFA
Example: NFA to DFA
Example: NFA to DFA
Example: NFA to DFA
Example: NFA to DFA
Example: NFA to DFA
The story so far...

\[
\text{REs} \subseteq \text{CFGs} \quad \text{DFAs} = \text{NFAs}
\]
Regular expressions $\subseteq$ NFAs $\equiv$ DFAs

We have shown how to build an optimal DFA for every regular expression

- Build NFA
- Convert NFA to DFA using subset construction
- Minimize resulting DFA

Thus, we could now implement a RegExp library

- most RegExp libraries actually simulate the NFA
- (even better: one can combine the two approaches: apply DFA minimization lazily while simulating the NFA)
The story so far...

Is this \( \subseteq \) really “=” or “\( \subseteq \)”?
Regular expressions ≡ NFAs ≡ DFAs

**Theorem:** For any NFA, there is a regular expression that accepts the same language

**Corollary:** A language is recognized by a DFA (or NFA) if and only if it has a regular expression

You need to know these facts

- the construction for the Theorem is included in the slides after this, but you will not be tested on it
Languages represented by DFA, NFAs, or regular expressions are called Regular Languages.
Recall: Algorithms for Regular Languages

We have seen algorithms for

• RE to NFA
• NFA to DFA
• DFA/NFA to RE (not tested)
• DFA minimization

Practice three of these in HW.
(May also be on the final.)
Example Corollary of These Results

**Corollary**: If $A$ is the language of a regular expression, then $\overline{A}$ is the language of a regular expression$^*$. 

(This is the complement with respect to the universe of all strings over the alphabet, i.e., $\overline{A} = \Sigma^* \setminus A$.)
The story so far...

Now: Is this $\subseteq$ really "=" or "$\nsubseteq$"?
What languages have DFAs? CFGs?

All of them?
Languages and Representations!

All

Context-Free

Regular

\(0^*\)

DFA

NFA

Regex

Finite

\{001, 10, 12\}
Languages and Representations!

Reminder: All finite languages are regular.
Construct a DFA for each string in the language.

Then, put them together using the union construction.
Languages and Machines!

All

Context-Free

Regular

Finite

\{001, 10, 12\}

DFA

NFA

Regex

Warmup 2: Surprising example here
An Interesting Infinite Regular Language

L = \{x \in \{0, 1\}^*: x \text{ has an equal number of substrings 01 and 10}\}.

L is infinite.

0, 00, 000, ...

L is regular. How could this be?

That seems to require comparing counts...

- easy for a CFG
- but seems hard for DFAs!
An Interesting Infinite Regular Language

L = \{x \in \{0, 1\}^* : x \text{ has an equal number of substrings 01 and 10}\}.

L is infinite.

0, 00, 000, ...

L is regular. How could this be? It is just the set of binary strings that are empty or begin and end with the same character!
Languages and Representations!

Main Event: Prove there is a context-free language that isn’t regular.

???

Context-Free

Regular

\(0^*\)

Finite

\{001, 10, 12\}

DFA

NFA

Regex
Tangent: How to prove a DFA minimal?

- Show there is no smaller DFA...

- Find a set of strings that *must* be distinguished
  - Such a set is a lower bound on the DFA size
Recall: Binary strings with a 1 in the 3rd position from the start

Distinguishing set:

\{\varepsilon, 0, 00, 000, 001\}
The language of “Binary Palindromes” is Context-Free

\[ S \rightarrow \varepsilon \mid 0 \mid 1 \mid 0S0 \mid 1S1 \]
Is the language of “Binary Palindromes” Regular?

Intuition (NOT A PROOF!):

Q: What would a DFA need to keep track of to decide?
A: It would need to keep track of the “first part” of the input in order to check the second part against it
...but there are an infinite # of possible first parts and we only have finitely many states.

Proof idea: any machine that does not remember the entire first half will be wrong for some inputs
Useful Lemmas about DFAs

**Lemma 1**: If DFA $M$ takes $x, y \in \Sigma^*$ to the same state, then for every $z \in \Sigma^*$, $M$ accepts $x \cdot z$ iff it accepts $y \cdot z$.

$M$ can’t remember that the input was $x$, not $y$. 

$x \cdot z = x_1 x_2 \ldots x_n z_1 z_2 \ldots z_k$

$y \cdot z = y_1 y_2 \ldots y_m z_1 z_2 \ldots z_k$
Useful Lemmas about DFAs

Lemma 2: If DFA $M$ has $n$ states and a set $S$ contains more than $n$ strings, then $M$ takes at least two strings from $S$ to the same state.

$M$ can’t take $n+1$ or more strings to different states because it doesn’t have $n+1$ different states.
So, some pair of strings must go to the same state.
B = \{\text{binary palindromes}\} \text{ can’t be recognized by any DFA}

Suppose for contradiction that some DFA, M, recognizes B. We will show M accepts or rejects a string it shouldn’t.

Consider \( S = \{1, 01, 001, 0001, 00001, \ldots\} = \{0^n1 : n \geq 0\} \).
$B = \{\text{binary palindromes}\}$ can’t be recognized by any DFA

Suppose for contradiction that some DFA, $M$, accepts $B$. We will show $M$ accepts or rejects a string it shouldn’t.

Consider $S = \{1, 01, 001, 0001, 00001, \ldots\} = \{0^n1 : n \geq 0\}$.

Since there are finitely many states in $M$ and infinitely many strings in $S$, by Lemma 2, there exist strings $0^a1 \in S$ and $0^b1 \in S$ with $a \neq b$ that end in the same state of $M$.

SUPER IMPORTANT POINT: You do not get to choose what $a$ and $b$ are. Remember, we’ve just proven they exist...we must take the ones we’re given!
\[ B = \{\text{binary palindromes}\} \text{ can’t be recognized by any DFA} \]

Suppose for contradiction that some DFA, \( M \), accepts \( B \). We will show \( M \) accepts or rejects a string it shouldn’t.

Consider \( S = \{1, 01, 001, 0001, 00001, \ldots\} = \{0^n1 : n \geq 0\} \).

Since there are finitely many states in \( M \) and infinitely many strings in \( S \), by Lemma 2, there exist strings \( 0^a1 \in S \) and \( 0^b1 \in S \) with \( a \neq b \) that end in the same state of \( M \).

Now, consider appending \( 0^a \) to both strings.
B = \{\text{binary palindromes}\} can't be recognized by any DFA

Suppose for contradiction that some DFA, \( M \), accepts B. We will show \( M \) accepts or rejects a string it shouldn't.

Consider \( S = \{1, 01, 001, 0001, 00001, \ldots\} = \{0^n1 : n \geq 0\} \).

Since there are finitely many states in \( M \) and infinitely many strings in \( S \), by Lemma 2, there exist strings \( 0^a1 \in S \) and \( 0^b1 \in S \) with \( a \neq b \) that end in the same state of \( M \).

Now, consider appending \( 0^a \) to both strings.

\[
\begin{array}{c}
0^a1 & \rightarrow & 0^a \\
0^b1 & \rightarrow & q \\
\end{array}
\]

Since \( 0^a1 \) and \( 0^b1 \) end in the same state, \( 0^a10^a \) and \( 0^b10^a \) also end in the same state, call it \( q \). But then \( M \) makes a mistake: \( q \) needs to be an accept state since \( 0^a10^a \in B \), but \( M \) would accept \( 0^b10^a \notin B \), which is an error.
B = \{binary palindromes\} can’t be recognized by any DFA

Suppose for contradiction that some DFA, M, accepts B. We will show M accepts or rejects a string it shouldn’t.

Consider \( S = \{1, 01, 001, 0001, 00001, \ldots\} = \{0^n1 : n \geq 0\} \).

Since there are finitely many states in M and infinitely many strings in S, by Lemma 2, there exist strings \( 0^a1 \in S \) and \( 0^b1 \in S \) with \( a \neq b \) that end in the same state of M.

Now, consider appending \( 0^a \) to both strings.

Since \( 0^a1 \) and \( 0^b1 \) end in the same state, \( 0^a10^a \) and \( 0^b10^a \) also end in the same state, call it \( q \). But then M makes a mistake: \( q \) needs to be an accept state since \( 0^a10^a \in B \), but M would accept \( 0^b10^a \notin B \), which is an error.

This proves that M does not recognize B, contradicting our assumption that it does. Thus, no DFA recognizes B.
Showing that a Language $L$ is not regular

1. “Suppose for contradiction that some DFA $M$ recognizes $L$.”
2. Consider an INFINITE set $S$ of prefixes (which we intend to complete later).
3. “Since $S$ is infinite and $M$ has finitely many states, there must be two strings $s_a$ and $s_b$ in $S$ for $s_a \neq s_b$ that end up at the same state of $M$.”
4. Consider appending the (correct) completion $t$ to each of the two strings.
5. “Since $s_a$ and $s_b$ both end up at the same state of $M$, and we appended the same string $t$, both $s_a t$ and $s_b t$ end at the same state $q$ of $M$. Since $s_a t \in L$ and $s_b t \notin L$, $M$ does not recognize $L$.”
6. “Thus, no DFA recognizes $L$.”
Showing that a Language $L$ is not regular

The choice of $S$ is the creative part of the proof

You must find an infinite set $S$ with the property that no two strings can be taken to the same state

   – i.e., for every pair of strings $S$ there is an “accept” completion that the two strings DO NOT SHARE
Prove $A = \{0^n1^n : n \geq 0\}$ is not regular

Suppose for contradiction that some DFA, $M$, recognizes $A$.

Let $S =$
Prove $A = \{0^n1^n : n \geq 0\}$ is not regular

Suppose for contradiction that some DFA, $M$, recognizes $A$.

Let $S = \{0^n : n \geq 0\}$. Since $S$ is infinite and $M$ has finitely many states, there must be two strings, $0^a$ and $0^b$ for some $a \neq b$ that end in the same state in $M$. 
Prove $A = \{0^n1^n : n \geq 0\}$ is not regular

Suppose for contradiction that some DFA, $M$, recognizes $A$.

Let $S = \{0^n : n \geq 0\}$. Since $S$ is infinite and $M$ has finitely many states, there must be two strings, $0^a$ and $0^b$ for some $a \neq b$ that end in the same state in $M$.

Consider appending $1^a$ to both strings.
Prove $A = \{0^n1^n : n \geq 0\}$ is not regular

Suppose for contradiction that some DFA, $M$, recognizes $A$.

Let $S = \{0^n : n \geq 0\}$. Since $S$ is infinite and $M$ has finitely many states, there must be two strings, $0^a$ and $0^b$ for some $a \neq b$ that end in the same state in $M$.

Consider appending $1^a$ to both strings.

Note that $0^a1^a \in A$, but $0^b1^a \notin A$ since $a \neq b$. But they both end up in the same state of $M$, call it $q$. Since $0^a1^a \in A$, state $q$ must be an accept state but then $M$ would incorrectly accept $0^b1^a \notin A$ so $M$ does not recognize $A$.

Thus, no DFA recognizes $A$. 
Prove $P = \{\text{balanced parentheses}\}$ is not regular

Suppose for contradiction that some DFA, $M$, accepts $P$.

Let $S =$
Prove $P = \{\text{balanced parentheses}\}$ is not regular

Suppose for contradiction that some DFA, $M$, recognizes $P$.

Let $S = \{ (n : n \geq 0) \}$. Since $S$ is infinite and $M$ has finitely many states, there must be two strings, $(^a$ and $(^b$ for some $a \neq b$ that end in the same state in $M$. 
Prove \( P = \{ \text{balanced parentheses} \} \) is not regular

Suppose for contradiction that some DFA, \( M \), recognizes \( P \).

Let \( S = \{ (n : n \geq 0) \} \). Since \( S \) is infinite and \( M \) has finitely many states, there must be two strings, \( (a) \) and \( (b) \) for some \( a \neq b \) that end in the same state in \( M \).

Consider appending \( )^a \) to both strings.
Prove \( P = \{\text{balanced parentheses}\} \) is not regular

Suppose for contradiction that some DFA, \( M \), recognizes \( P \).

Let \( S = \{ (n : n \geq 0) \}. \) Since \( S \) is infinite and \( M \) has finitely many states, there must be two strings, \( (a) \) and \( (b) \) for some \( a \neq b \) that end in the same state in \( M \).

Consider appending \( )^a \) to both strings.

Note that \( (a)^a \in P \), but \( (b)^a \notin P \) since \( a \neq b \). But they both end up in the same state of \( M \), call it \( q \). Since \( (a)^a \in P \), state \( q \) must be an accept state but then \( M \) would incorrectly accept \( (b)^a \notin P \) so \( M \) does not recognize \( P \).

Thus, no DFA recognizes \( P \).
Showing that a Language $L$ is not regular

1. “Suppose for contradiction that some DFA $M$ recognizes $L$.”
2. Consider an INFINITE set $S$ of prefixes (which we intend to complete later). It is imperative that for every pair of strings in our set there is an “accept” completion that the two strings DO NOT SHARE.
3. “Since $S$ is infinite and $M$ has finitely many states, there must be two strings $s_a$ and $s_b$ in $S$ for $s_a \neq s_b$ that end up at the same state of $M$.”
4. Consider appending the (correct) completion $t$ to each of the two strings.
5. “Since $s_a$ and $s_b$ both end up at the same state of $M$, and we appended the same string $t$, both $s_a t$ and $s_b t$ end at the same state $q$ of $M$. Since $s_a t \in L$ and $s_b t \not\in L$, $M$ does not recognize $L$.”
6. “Thus, no DFA recognizes $L$.”
Fact: This method is optimal

- Suppose that for a language $L$, the set $S$ is a largest set of prefixes with the property that, for every pair $s_a \neq s_b \in S$, there is some string $t$ such that one of $s_a t$, $s_b t$ is in $L$ but the other isn’t.
- If $S$ is infinite, then $L$ is not regular
- If $S$ is finite, then the minimal DFA for $L$ has precisely $|S|$ states, one reached by each member of $S$. 
Fact: This method is optimal

• Suppose that for a language $L$, the set $S$ is a largest set of prefixes with the property that, for every pair $s_a \neq s_b \in S$, there is some string $t$ such that one of $s_a t, s_b t$ is in $L$ but the other isn’t.

• If $S$ is infinite, then $L$ is not regular

• If $S$ is finite, then the minimal DFA for $L$ has precisely $|S|$ states, one reached by each member of $S$.

Corollary: Our minimization algorithm was correct.

– we separated exactly those states for which some $t$ would make one accept and another not accept
Important Notes

• It is not necessary for our strings $xz$ with $x \in L$ to allow any string in the language
  – we only need to find a small “core” set of strings that must be distinguished by the machine

• It is not true that, if $L$ is irregular and $L \subseteq U$, then $U$ is irregular!
  – we always have $L \subseteq \Sigma^*$ and $\Sigma^*$ is regular!
  – our argument needs different answers: $xz \in L \iff yz \in L$
    for $\Sigma^*$, both strings are always in the language

Do not claim in your proof that, because $L \subseteq U$, $U$ is also irregular
New Machinery: Generalized NFAs

• Like NFAs but allow
  – parallel edges (between the same pair of states)
  – regular expressions as edge labels
    NFAs already have edges labeled $\varepsilon$ or $a$

• Machine can follow an edge labeled by A by reading a string of input characters in the language of A
  – (if A is $a$ or $\varepsilon$, this matches the original definition, but we now allow REs built with recursive steps.)
New Machinery: Generalized NFAs

• Like NFAs but allow
  – parallel edges
  – regular expressions as edge labels
    NFAs already have edges labeled $\varepsilon$ or $a$

• The label of a path is now the concatenation of the regular expressions on those edges, making it a regular expression

• Def: A string $x$ is accepted by a generalized NFA iff there is a path from start to final state labeled by a regular expression whose language contains $x$
Construction Idea

Add new start state and final state

Then delete the original states one by one, adding edges to keep the same language, until the graph looks like:
Starting from an NFA

Then delete the original states one by one, adding edges to keep the same language, until the graph looks like:

Final graph has only one path to the accepting state, which is labeled by $A$, so it accepts iff $x$ is in the language of $A$

Thus, $A$ is a regular expression with the same language as the original NFA.
Only two simplification rules

- **Rule 1:** For any two states \( q_1 \) and \( q_2 \) with parallel edges (possibly \( q_1 = q_2 \)), replace

\[
\begin{array}{c}
q_1 \\
\text{A} \\
q_2 \\
\text{B} \\
q_1
\end{array}
\]

by

\[
\begin{array}{c}
q_1 \\
\text{AUB} \\
q_2
\end{array}
\]

If the machine would have used the edge labeled A by consuming an input \( x \) in the language of A, it can instead use the edge labeled \( \text{AUB} \).

Furthermore, this new edge does not allow transitions for any strings other than those that matched A or B.
Only two simplification rules

- **Rule 2**: Eliminate non-start/accepting state $q_3$ by creating direct edges that skip $q_3$

  Any path from $q_1$ to $q_2$ would have to match $AB^nC$ for some $n$ (the number of times the self loop was used), so the machine can use the new edge instead. New edge *only* allows strings that were allowed before.
While the box contains some state $s$:
for all states $r, t$ with $(r, s)$ and $(s, t)$ in $E$:
    create a direct edge $(r, t)$ by Rule 2
delete $s$ (no longer needed)
merge all parallel edges by Rule 1

Add new start state and final state
While the box contains some state $s$: for all states $r$, $t$ with $(r, s)$ and $(s, t)$ in $E$: create a direct edge $(r, t)$ by Rule 2 delete $s$ (no longer needed) merge all parallel edges by Rule 1

When the loop exits, the graph looks like this:

A is a regular expression with the same language as the original NFA.
Converting an NFA to a regular expression

Consider the DFA for the mod 3 sum
- Accept strings from \{0,1,2\}^* where the digits mod 3 sum of the digits is 0
Splicing out a state $t_1$

Create direct edges between neighbors of $t_1$ (so that we can delete it afterward)
Splicing out a state $t_1$

Regular expressions to add to edges

$t_0 \rightarrow t_1 \rightarrow t_0 : 10^*2$
$t_0 \rightarrow t_1 \rightarrow t_2 : 10^*1$
$t_2 \rightarrow t_1 \rightarrow t_0 : 20^*2$
$t_2 \rightarrow t_1 \rightarrow t_2 : 20^*1$
Splicing out a state $t_1$

Delete $t_1$ now that it is redundant

$t_0 \to t_1 \to t_0 : 10*2$
$t_0 \to t_1 \to t_2 : 10*1$
$t_2 \to t_1 \to t_0 : 20*2$
$t_2 \to t_1 \to t_2 : 20*1$
Splicing out a state $t_1$

Create direct edges between neighbors of $t_2$ (so that we can delete it afterward)
Splicing out a state $t_1$

Regular expressions to add to edges

$R_1$: $0 \cup 10^*2$
$R_2$: $2 \cup 10^*1$
$R_3$: $1 \cup 20^*2$
$R_4$: $0 \cup 20^*1$
Splicing out state \( t_2 \) (and then \( t_0 \))

Delete \( t_2 \) now that it is redundant

\[
\begin{align*}
R_1 &= 0 \cup 10*2 \\
R_2 &= 2 \cup 10*1 \\
R_3 &= 1 \cup 20*2 \\
R_4 &= 0 \cup 20*1 \\
R_5 &= R_1 \cup R_2 R_4 * R_3
\end{align*}
\]
Splicing out state $t_2$ (and then $t_0$)

Create direct ($s, f$) edge so we can delete $t_0$

$R_1$: $0 \cup 10*2$
$R_2$: $2 \cup 10*1$
$R_3$: $1 \cup 20*2$
$R_4$: $0 \cup 20*1$
$R_5$: $R_1 \cup R_2 R_4 R_3$
Splicing out state $t_2$ (and then $t_0$)

Regular expressions to add to edges

$R_1$: $0 \cup 10^*2$
$R_2$: $2 \cup 10^*1$
$R_3$: $1 \cup 20^*2$
$R_4$: $0 \cup 20^*1$
$R_5$: $R_1 \cup R_2 R_4^* R_3$

$t_0 \rightarrow t_1 \rightarrow t_0$: $R_5^*$
Splicing out state $t_2$ (and then $t_0$)

Delete $t_0$ now that it is redundant

\[
\begin{align*}
R_1 &: 0 \cup 10*2 \\
R_2 &: 2 \cup 10*1 \\
R_3 &: 1 \cup 20*2 \\
R_4 &: 0 \cup 20*1 \\
R_5 &: R_1 \cup R_2R_4*R_3 \\
R_6 &: R_5*
\end{align*}
\]
Splicing out state $t_2$ (and then $t_0$)

Regular expressions to add to edges

- $R_1$: $0 \cup 10^2$
- $R_2$: $2 \cup 10^1$
- $R_3$: $1 \cup 20^2$
- $R_4$: $0 \cup 20^1$
- $R_5$: $R_1 \cup R_2 R_4 R_3$
- $R_6$: $R_5^*$

Final regular expression: $R_6 = (0 \cup 10^2 \cup (2 \cup 10^1)(0 \cup 20^1)^*(1 \cup 20^2))^*$
Application of FSMs: Pattern matching

• Given
  – a string $s$ of $n$ characters
  – a pattern $p$ of $m$ characters
  – usually $m \ll n$

• Find
  – all occurrences of the pattern $p$ in the string $s$

• Obvious algorithm:
  – try to see if $p$ matches at each of the positions in $s$
    stop at a failed match and try matching at the next position: $O(mn)$ running time.
Application of FSMs: Pattern Matching

• With DFAs can do this in $O(m + n)$ time.

• See Extra Credit problem on HW8 for some ideas of how to get to $O(m^2 + n)$. 
Last time: NFA to DFA
In general the DFA might need a state for every subset of states of the NFA:
- Power set of the set of states of the NFA
- $n$-state NFA yields DFA with at most $2^n$ states
- We saw an example where roughly $2^n$ is necessary
  “Is the $n^{th}$ char from the end a 1?”

The famous “P=NP?” question asks whether a similar blow-up is always necessary to get rid of nondeterminism for polynomial-time algorithms.