A. Implementing an R.E. Using an NFA

- It’s fairly easy to take a regular expression and encode its search as a transition graph for an NFA. We initialize the NFA to just have one state marked start and accept simultaneously. As we go through the expression, we extend the graph and/or join it with other graphs to form larger and larger graphs.

- For an expression \( a \), build a graph with four nodes: Node 1 is the start state and has an \( \varepsilon \)-transition to node 2, which has an arc labeled “a” to node 3, which has an \( \varepsilon \)-transition to a node 4, our accept state.

- For an alternation \( R_1 \mid R_2 \), recursively generate graphs for \( R_1 \) and \( R_2 \) and add two more nodes. The accept state of our current graph has an \( \varepsilon \)-transition to the first new node, and this new node has \( \varepsilon \)-transitions to the start states of the graphs for \( R_1 \) and \( R_2 \). The second new node has \( \varepsilon \)-transitions from the accept states for \( R_1 \) and \( R_2 \) and it is the accept state of the new graph.

- For a sequence \( R_1 R_2 \), we \( \varepsilon \)-link the accept state of \( R_1 \) to the start state for \( R_2 \). The start state of \( R_1 \) is our start state and the accept state for \( R_2 \) is our accept state.

- For a Kleene * expression \( R^* \), we take the graph for \( R \) and add new start and accept nodes and add an \( \varepsilon \)-transition from the end of \( R \) to the beginning of \( R \) (this gives us the repeating part of \( R^* \)) and also an \( \varepsilon \)-transition from our start to accept states (this is the zero occurrences of \( R \) part of \( R^* \)).
• As an example, Figure 5 is a graph for \(a (b | c)^*\) that follows the algorithm in the book. (It contains even more \(\varepsilon\)-transitions than the transformations from Figures 1 – 4.)
  • There are two sorts of nondeterminism in Figure 5: Arcs labeled by \(\varepsilon\) and decisions (two arcs from the same node with the same label).
• To make it easier to see what the NFA does, Figure 6 shows the result of eliminating some redundant \(\varepsilon\)-arrows from Figure 5.
• Not only does there exist an NFA that accepts the same language as a given regular expression, there exists a regular expression for each NFA. (This is not as obvious.) In that sense, NFAs have the same expressiveness as regular expressions.

B. Converting from an NFA to an Equivalent DFA
• For every NFA there exists an equivalent DFA; we can build it by tracking all the states the NFA could be in after seeing the current input.
  • Say NFA node \(S\) goes to two different states on character \(a\). When we use backtracking search, we follow one of those arcs and backtrack to try the other arc if necessary.
  • In the DFA, we’ll instead have a single set of states that answers the question "What are all the NFA states I could have gotten to from \(S\) with an arc labeled \(a\)?"
  • Whenever execution reaches this DFA state we’ll be in one of the NFA states but not know which one.\(^1\)
• So each DFA state is the set of all current possible NFA states, and to get from one DFA state to another, we have to look at all possible NFA states we can get to from any of the current possible NFA states. This gives us a DFA state = another set of NFA states. If it’s a set we’ve already seen, it’s a new state to add to the set of DFA states.
• Note that if there are \(n\) states in the NFA then there are \(2^n\) possible DFA states (though not all of them are likely to be in the set of actual needed DFA states).

\(^1\) Or if you prefer, we can think of being in all of those states simultaneously by following all possible search paths simultaneously. For example, using threads, whenever we encounter a state with multiple outgoing arcs with the same label, fork off a new thread of control and follow both paths simultaneously.
C. Converting From an $\varepsilon$-Transition-Free NFA to an Equivalent DFA

Here’s an algorithm for the NFA-to-DFA conversion. Let’s assume we’ve removed $\varepsilon$-transitions from the NFA in a preprocessing step. (We’ll show it in the next section.)

- Initialize the set of DFA states to the one set of all initial NFA states. This state will be the initial DFA state. Mark this set “unprocessed.”
- While there exists an unprocessed $U \in$ the set of DFA states
  - For every character $a$ in the alphabet
    - Calculate $T =$ the set of target NFA states for all the NFA states in $U$ with character $a$.
      - If $T$ is not already in the set of DFA states, add it and mark it unprocessed.
      - If $T$ contains any NFA-accepting states, mark it as accepting in the DFA.
    - Create a DFA transition from $U$ to $T$ labeled $a$.
  - Mark $U$ as being processed.

D. Calculate the $\varepsilon$-Closure of a NFA

- “Calculating the $\varepsilon$-closure of an NFA” involves taking an NFA and finding an equivalent NFA that doesn’t contain $\varepsilon$-transitions.
- This is useful because an NFA with $\varepsilon$-transitions can be “in” more than one state at a time.
  - If we have a state $S_1$ connected to $S_2$ by an $\varepsilon$ arc, then we have the choice of staying in $S_1$ or going to $S_2$.
    - The choice may be important because we can have arcs from $S_1$ and $S_2$ with the same labels.
- To remove $\varepsilon$-transitions, we can do the conversion by looking at the states reachable by $\varepsilon$-paths from a given state. Here’s an algorithm:
  - Let NFA₁ have states $\Sigma_1$ and a set of state transitions $\tau_1 \subseteq \Sigma_1 \times (\Sigma \cup \{\varepsilon\}) \times \Sigma_1$. (Recall that $\Sigma$ is the alphabet of input characters.)
  - For $\Sigma_2$ (the states of our result NFA₂)
    - Initialize $\Sigma_2 = \emptyset$
    - For each $s \in \Sigma_1$
      - Find all the states reachable from $s$ via $\varepsilon$-transitions.
      - If this set is not already in $\Sigma_2$ then add it.
      - Mark the state as initial in NFA₂ if it contains any state initial in NFA₁
      - Mark the state as accepting in NFA₂ if it contains any state accepting in NFA₁
    - Now to define $\tau_2$ (the transitions for NFA₂) where $\tau_2 \subseteq \Sigma_2 \times \Sigma \times \Sigma_2$.
      - For each $S \in \Sigma_2$ and $a \in \Sigma$
        - Let $T = \emptyset$
        - For each $s \in S$, add $t$ to $T$ if $(s, a) \rightarrow t$ is a transition in $\tau_1$
        - Add $(S, a) \rightarrow T$ as a transition to $\tau_2$.
      - It’s possible to have one algorithm that combines removing $\varepsilon$-transitions and converting from the NFA-to-DFA, but it’s a little messy, so I’m omitting it.
E. Removing Stuck Configurations

- In an NFA, you can have a state $s$ that has no outgoing arc for a symbol $a$. We say that $(s, a)$ is a “stuck” configuration in that if you’re in state $s$ and see an $a$, the computation can’t proceed. In the backtracking search, you have to go back to most recent non-deterministic choice and try an alternative.

- If $(s, a)$ is stuck in the NFA, it’s possible for the DFA to be missing a transition for $s$ on $a$. We can model this “stuck” behavior as going to an *error state* (a state that never leads to an accepting state).
  - Add a single new error state $E$ and add a DFA arc labeled $a$ from $S'$ to $E$.
  - We only need one $E$, but we need an arc to $E$ for every missing arc in the DFA.
    - (i.e., one arc for every combination of $a$ and $S'$ where there’s no outgoing arc from $S$ labeled $a$.)
  - $E$ also needs circular arcs for every character in the alphabet: For each character $a$, add an $E$-to-$E$ link labeled $a$. This ensures that once we get to $E$, we never leave.

---

**Figure 5**: $a (b | c)^*$ Before Removing $\varepsilon$-Transitions

**Figure 6**: $a (b | c)^*$ After Removing Some $\varepsilon$-Transitions
F. Another Example of NFA to DFA Conversion

- Figure 7 shows a full example with the original NFA, the NFA after ε-closure, the set-of-states DFA, and the DFA plus error state.
Activity 8 Problems

1. Go back to Activity 7 and look at the NFA that accepts $L_1 \cup L_2$ where $L_1$ is the language of $a$’s and $b$’s with at least one $b$ after every and $L_2$ is its opposite, the language of all $a$’s and $b$’s with at least one $a$ after every $b$.
   a. Remove $\varepsilon$-transitions from it.
   b. Convert the $\varepsilon$-free NFA to a DFA. Add an error state if you need one.

2. Take the NFA described below and find an equivalent DFA.
   a. Identify all the error states and combine them into one error state and remove them from the NFA and all transitions in the NFA.
   b. Remove $\varepsilon$-transitions. (If we still had any error states, then if we had a state that combined an error and non-error state, we would have removed the error state.)
   c. Convert to a DFA using the set-of-states transformation. (If there are any stuck transitions) add an error state back into the automaton and fill in transitions to the error state from stuck configurations.

   Start state = A, Accept state = G

<table>
<thead>
<tr>
<th>State</th>
<th>$\varepsilon$</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>$D, F$</td>
<td>$E, G$</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>C</td>
<td>$F$</td>
<td>$G$</td>
</tr>
<tr>
<td>C</td>
<td>$H$</td>
<td>$E$</td>
<td>$E$</td>
<td>$G$</td>
</tr>
<tr>
<td>D</td>
<td>$E$</td>
<td>$E$</td>
<td></td>
<td>$D$</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>$E$</td>
<td></td>
<td>$D$</td>
</tr>
<tr>
<td>F</td>
<td></td>
<td>$G$</td>
<td>$E$</td>
<td>$B$</td>
</tr>
<tr>
<td>G</td>
<td></td>
<td>$H$</td>
<td>$D$</td>
<td>$F$</td>
</tr>
<tr>
<td>H</td>
<td>$C$</td>
<td>$D$</td>
<td>$D$</td>
<td>$C$</td>
</tr>
</tbody>
</table>

From A, there are two arrows labeled y and two labeled z
Selected Solutions to Activity 8 Problems

2. (NFA with error states.)
   a. After finding error states D and E and removing them from the automaton.
      [10/2] (Since we removed all error states and transitions to them, we have a bunch of empty
      squares instead of entries labeled Error; there’s also no row for Error.)
      
      Start state = A, Accept state = G

      \[
      \begin{array}{c|cccc}
      \text{State} & \varepsilon & x & y & z \\
      \hline
      A & B & C & F & G \\
      B & C & F & G \\
      C & H & & G \\
      F & & G & B \\
      G & H & & F \\
      H & C & & C \\
      \end{array}
      \]

   b. After removing \(\varepsilon\)-transitions. You can write anything for \(AB\) that indicates a set of two states:
      \(\{A, B\}, AB, A+B\).
      
      Start state = AB, Accept state = G

      \[
      \begin{array}{c|ccc}
      \text{State} & x & y & z \\
      \hline
      AB & CH & F & G \\
      CH & & CH, G \\
      F & G & AB \\
      G & CH & F \\
      \end{array}
      \]

      From CH, there are two arrows labeled \(z\)
c. After converting to set-of-states DFA, adding an error state, and filling in stuck configurations. [10/2: Went uniformly to string of letters notation: e.g., CGH = \{C, G, H\}.

Start States = \{AB, ABCFGH\}, Accept State = G

<table>
<thead>
<tr>
<th>State</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>AB</td>
<td>CH</td>
<td>F</td>
<td>G</td>
</tr>
<tr>
<td>CH</td>
<td>Err</td>
<td>Err</td>
<td>CGH</td>
</tr>
<tr>
<td>CGH</td>
<td>CH</td>
<td>Err</td>
<td>CFGH</td>
</tr>
<tr>
<td>CFGH</td>
<td>CGH</td>
<td>Err</td>
<td>ABCFGH</td>
</tr>
<tr>
<td>ABCFGH</td>
<td>CGH</td>
<td>F</td>
<td>ABCFGH</td>
</tr>
<tr>
<td>F</td>
<td>G</td>
<td>Err</td>
<td>AB</td>
</tr>
<tr>
<td>G</td>
<td>CH</td>
<td>Err</td>
<td>F</td>
</tr>
<tr>
<td>Err</td>
<td>Err</td>
<td>Err</td>
<td>Err</td>
</tr>
</tbody>
</table>