A Few Questions about Regular Expressions

- What does a regular expression represent?
  - Just a set of strings
- What are the basic components of r.e.’s?
  - E.g., we saw that $e+$ is the same as $ee^*$
- How are r.e.’s implemented?
  - We’ll see how to turn a r.e. into a program
- Can r.e.’s represent all possible languages?
  - The answer turns out to be no!
  - The languages represented by regular expressions are called, appropriately, the regular languages

Some Definitions

- An alphabet is a finite set of symbols
  - Usually denoted $\Sigma$
- A string is a finite sequence of symbols from $\Sigma$
  - $\varepsilon$ is the empty string (“” in Ruby)
  - $|s|$ is the length of string $s$
    - $|\text{Hello}| = 5, |\varepsilon| = 0$
  - Note: $\emptyset$ is the empty set (with 0 elements); $\emptyset \neq \{ \varepsilon \}$
- Concatenation is indicated by juxtaposition
  - If $s_1 = \text{super}$ and $s_2 = \text{hero}$, then $s_1 s_2 = \text{superhero}$
  - Sometimes also written $s_1 ; s_2$
  - For any string $s$, we have $s \varepsilon = \varepsilon s = s$

Languages

- A language is a set of strings over an alphabet
- Example: The set of all valid Ruby programs
  - Is there a Ruby regular expression for this language?
- Example: The set of strings of length 0 over the alphabet $\Sigma = \{a, b, c\}$
  - $\{ s \mid s \in \Sigma^* \text{ and } |s| = 0 \} = \{ \varepsilon \} \neq \emptyset$

Languages (cont’d)

- Example: The set of phone numbers over the alphabet $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 9, (, )\}$
  - Give an example element of this language
  - Are all strings over the alphabet in the language?
  - Is there a Ruby regular expression for this language?
    - Is the Ruby regular expression over the same alphabet?
  - Often written $\Sigma^*$
Operations on Languages

• Let $\Sigma$ be an alphabet and let $L_1$, $L_2$ be languages over $\Sigma$

• Concatenation $L_1 L_2$ is defined as
  
  $L_1 L_2 = \{ xy \mid x \in L_1 \text{ and } y \in L_2 \}$

  Example: $L_1 = \{ \text{"hi", "bye"} \}$, $L_2 = \{ \text{"1", "2"} \}$
  
  $L_1 L_2 = \{ \text{"hi1", "hi2", "bye1", "bye2"} \}$

• Union is defined as
  
  $L_1 \cup L_2 = \{ x \mid x \in L_1 \text{ or } x \in L_2 \}$

  Example: $L_1 = \{ \text{"hi", "bye"} \}$, $L_2 = \{ \text{"1", "2"} \}$
  
  $L_1 \cup L_2 = \{ \text{"hi", "bye", "1", "2"} \}$

Examples of $L^n$

• Let $L = \{ a, b, c \}$

• Then
  
  $L^0 = \{ \}$
  $L^1 = \{ a, b, c \}$
  $L^2 = \{ aa, ab, ac, ba, bb, bc, ca, cb, cc \}$

Definition of Regexps

• Given an alphabet $\Sigma$, the regular expressions over $\Sigma$ are defined inductively as

<table>
<thead>
<tr>
<th>regular expression</th>
<th>denotes language</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$\epsilon$</td>
<td>${ } \epsilon$</td>
</tr>
<tr>
<td>each element $a \in \Sigma$</td>
<td>${ }$</td>
</tr>
</tbody>
</table>

Operations on Languages (cont’d)

• Define $L^n$ inductively as
  
  $L^0 = \{ \}$
  $L^n = L^{n-1} L$ for $n > 0$

  In other words,
  
  $L^1 = L \{ \} = L$
  $L^2 = LL = LL$
  $L^3 = LL^2 = LLL$
  $\ldots$

• Kleene closure is defined as
  
  $L^* = \bigcup_{i \geq 0} L^i$

  In other words...

  $L^*$ is the language (set of all strings) formed by concatenating together zero or more strings from $L$

Definition of Regexps (cont’d)

• Let $A$ and $B$ be regular expressions denoting languages $L_A$ and $L_B$, respectively

<table>
<thead>
<tr>
<th>regular expression</th>
<th>denotes language</th>
</tr>
</thead>
<tbody>
<tr>
<td>$AB$</td>
<td>$L_A L_B$</td>
</tr>
<tr>
<td>$(A</td>
<td>B)$</td>
</tr>
<tr>
<td>$A^*$</td>
<td>$L_A^*$</td>
</tr>
</tbody>
</table>

• There are no other regular expressions for $\Sigma$

• We use '('s as needed for grouping
The Language Denoted by an r.e.

- For a regular expression $e$, we will write $[[e]]$ to mean the language denoted by $e$
  - $[[a]] = \{a\}$
  - $[[a|b]] = \{a, b\}$
- If $s = [[re]]$, we say that $re$ accepts, describes, or recognizes $s$.

Which Strings Does $a^*b^*c^*$ Recognize?

<table>
<thead>
<tr>
<th>String</th>
<th>Acceptance</th>
</tr>
</thead>
<tbody>
<tr>
<td>aabbbc</td>
<td>Yes</td>
</tr>
<tr>
<td>abb</td>
<td>Yes</td>
</tr>
<tr>
<td>ac</td>
<td>Yes</td>
</tr>
<tr>
<td>$\varepsilon$</td>
<td>Yes</td>
</tr>
<tr>
<td>aacb</td>
<td>No</td>
</tr>
<tr>
<td>abcd</td>
<td>No</td>
</tr>
</tbody>
</table>

Example 1

- All strings over $\Sigma = \{a, b, c\}$ such that all the $a$'s are first, the $b$'s are next, and the $c$'s last
  - Example: $aabbbc$ but not $abc$ $b$
- Regexp: $a^*b^*c^*$
  - This is a valid regexp because...
  - $a$ is a regexp $[[a]] = \{a\}$
  - $a^*$ is a regexp $[[a^*]] = \{\varepsilon, a, aa, ...\}$
  - Similarly for $b^*$ and $c^*$
  - So $a^*b^*c^*$ is a regular expression

Example 2

- All strings over $\Sigma = \{a, b, c\}$
- Regexp: $(a|b|c)^*$
- Other regular expressions for the same language?
  - $(c|b|a)^*$
  - $(a^*b^*c^*)^*$
  - $(a|b|c)^*(abc)$
  - etc.

Example 3

- All whole numbers containing the substring $330$
- Regular expression: $(0|1)...|9)^*330(0|1)...|9)^*$
- What if we want to get rid of leading $0$'s?
  - $(1...|9)(0|1)...|9)*330(0|1)...|9)^*$
  - $330(0|1)...|9)^*$
- Any other solutions?
  - What about all whole numbers not containing the substring $330$?
  - Is it recognized by a regexp?

Example 4

- What language does $(10|0)^*(10|1)^*$ denote?
  - $(10|0)^*$
    - $0$ may appear anywhere
    - $1$ must always be followed by $0$
  - $(10|1)^*$
    - $1$ may appear anywhere
    - $0$ must always be preceded by $1$
- Put together, all strings of $0$'s and $1$'s where every pair of adjacent $0$'s precedes any pair of adjacent $1$'s
What Strings are in \((10|0)^*(10|1)^*\)?

\[
\begin{align*}
00101000 & \quad 11011101 \\
& \text{First part in } \left[\left[(10|0)^*\right]\right] \\
& \text{Second part in } \left[\left[(10|1)^*\right]\right] \\
& \text{Notice that } 0010 \text{ also in } \left[\left[(10|0)^*\right]\right] \\
& \text{But remainder of string is not in } \left[\left[(10|1)^*\right]\right]
\end{align*}
\]

0010101
Yes
101
Yes
011001
No

Example 5

- What language does this regular expression recognize?
  \(\left( (1|0)(0|1)[9] \right) \cup (2(0|123)) \cup (0|1)[5](0|1)[9]\)

- All valid times written in 24-hour format:
  - 10:17
  - 23:59
  - 0:45
  - 8:30

Two More Examples

- \((000|001)^*\)
  - Any string of 0’s and 1’s with no single 0’s

- \((0000000)^*\)
  - Strings with an even number of 0’s
  - Notice that some strings can be accepted more than one way
    - 000000 = 00 00 00 = 00 00 00 = 00 00 00

Regular Languages

- The languages that can be described using regular expressions are the \textit{regular languages} or \textit{regular sets}
- Not all languages are regular
  - Examples (without proof):
    - The set of palindromes over \(\Sigma\)
    - \(\{a^n b^n \mid n > 0\}\) \(a^n = \text{sequence of } n \text{ a's}\)

Almost all programming languages are not regular
- But aspects of them sometimes are (e.g., identifiers)
- Regular expressions are commonly used in parsing tools

Ruby Regular Expressions

- Almost all of the features we’ve seen for Ruby re.’s can be reduced to this formal definition
  - /Ruby/ – concatenation of single-character re.’s
  - /Ruby(Ruby)/ – union
  - /Ruby\/* – Kleene closure
  - /Ruby+s/ – same as (Ruby)(Ruby)\*
  - /Ruby/? – same as \(\epsilon\text{?}(\text{Ruby})\) (i.e. 1\(\epsilon\))
  - /[a-z]/ – same as \(\{a|b|c|…|z\}\)
  - /[^0-9]/ – same as \(\{a|b|c|…\} \text{ for } a,b,c,… \in \Sigma - \{0,9\}\)
  - \^, $ – correspond to extra characters in alphabet

Implementing Regular Expressions

- We can implement regular expressions by turning them into a \textit{finite automaton}
  - A “machine” for recognizing a regular language
Example

- Machine starts in \textit{start} or \textit{initial} state
- Repeat until the end of the string is reached:
  - Scan the next symbol \(s\) of the string
  - Take transition edge labeled with \(s\)
- The string is \textit{accepted} if the automaton is in a \textit{final} or \textit{accepting} state when the end of the string is reached

Example

- \(001011\) \hspace{1cm} \text{accepted}

What Language is This?

- All strings over \((0, 1)\) that end in 1
- What is a regular expression for this language?\((01)^*1\)

Formal Definition

- A \textit{deterministic finite automaton (DFA)} is a 5-tuple \((\Sigma, Q, q_0, F, \delta)\) where
  - \(\Sigma\) is an alphabet
  - \(Q\) is a nonempty set of states
  - \(q_0 \in Q\) is the start state
  - \(F \subseteq Q\) is the set of final states
  - \(\delta: Q \times \Sigma \rightarrow Q\) specifies the DFA's transitions
  - What's this definition saying that \(\delta\) is?

More on DFAs

- An FSA can have more than one final state:
- A string is accepted as long as there is at least one path to a final state
Our Example, Formally

- $\Sigma = \{0, 1\}$
- $Q = \{S_0, S_1\}$
- $q_0 = S_0$
- $F = \{S_1\}$
- $\delta$

<table>
<thead>
<tr>
<th>$S_0$</th>
<th>$S_0$</th>
<th>$S_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>S_1</td>
<td>S_0</td>
<td>S_1</td>
</tr>
</tbody>
</table>

Another Example

<table>
<thead>
<tr>
<th>string</th>
<th>state at end</th>
<th>accepts</th>
</tr>
</thead>
<tbody>
<tr>
<td>aabcc</td>
<td>S_2</td>
<td>Y</td>
</tr>
<tr>
<td>acc</td>
<td>S_2</td>
<td>Y</td>
</tr>
<tr>
<td>bbc</td>
<td>S_2</td>
<td>Y</td>
</tr>
<tr>
<td>aabbb</td>
<td>S_1</td>
<td>Y</td>
</tr>
<tr>
<td>aa</td>
<td>S_0</td>
<td>Y</td>
</tr>
<tr>
<td>$\epsilon$</td>
<td>S_0</td>
<td>Y</td>
</tr>
<tr>
<td>acba</td>
<td>S_3</td>
<td>N</td>
</tr>
</tbody>
</table>

(a,b,c notation shorthand for three self loops)

Another Example (cont’d)

What language does this DFA accept? $a^*b^*c^*$

S_3 is a dead state – a nonfinal state with no transition to another state

Shorthand Notation

• If a transition is omitted, assume it goes to a dead state that is not shown

What Lang. Does This DFA Accept?

$a^*b^*c^*$ again, so DFAs are not unique

Non-deterministic Finite Automata (NFA)

• An NFA is a 5-tuple ($\Sigma$, $Q$, $q_0$, $F$, $\delta$) where
  - $\Sigma$ is an alphabet
  - $Q$ is a nonempty set of states
  - $q_0 \in Q$ is the start state
  - $F \subseteq Q$ is the set of final states
  - $\delta \subseteq Q \times(\Sigma\cup\{\epsilon\}) \times Q$ specifies the NFA's transitions
    - Transitions on $\epsilon$ are allowed – can optionally take these transitions without consuming any input
    - Can have more than one transition for a given state and symbol
  - An NFA accepts $s$ if there is at least one path from its start to final state on $s$
Example DFA

- $S_0$ = "Haven’t seen anything yet"
- $S_1$ = "Last symbol seen was an a"
- $S_2$ = "Last two symbols seen were ab"
- $S_3$ = "Last three symbols seen were abb"

- Language?
- $(a|b)^*abb$  

NFA for $(a|b)^*abb$

- $ba$
  - Has paths to either $S_0$ or $S_1$
  - Neither is final, so rejected
- $babaabb$
  - Has paths to different states
  - One leads to $S_3$, so accepted

Another example DFA

- Language?
- $(ab|aba)^*$

NFA for $(ab|aba)^*$

- $aba$
  - Has paths to states $S_0$, $S_1$
- $ababa$
  - Has paths to $S_0$, $S_1$
  - Need to use $\epsilon$-transition

Relating R.E.’s to DFAs and NFAs

- Regular expressions, NFAs, and DFAs accept the same languages!

Reducing Regular Expressions to NFAs

- Goal: Given regular expression $e$, construct NFA $<e> = (\Sigma, Q, q_0, F, \delta)$
  - Remember r.e. defined recursively from primitive r.e. languages
  - Invariant: $|F| = 1$ in our NFAs
- Base case: $a$
  
  $<a> = ((a), \{S_0, S_1\}, S_0, \{S_1\}, \{(S_0, a, S_1)\})$
Reduction (cont’d)

- Base case: $\varepsilon$

$$<\varepsilon> = (\varepsilon, \{S0\}, S0, \{S0\}, \varnothing)$$

- Base case: $\varnothing$

$$<\varnothing> = (\varnothing, \{S0, S1\}, S0, \{S1\}, \varnothing)$$

Reduction (cont’d)

- Induction: $A B$

$$<A>$$
$$<B>$$

- $<A> = (\Sigma_A, Q_A, q_A, \delta_A, f_A)$
- $<B> = (\Sigma_B, Q_B, q_B, \delta_B)$
- $<AB> = (\Sigma_A \cup \Sigma_B, Q_A \cup Q_B, q_0, \delta_A \cup \delta_B \cup \{(f_A, \varepsilon, q_A), (f_B, \varepsilon, S1)\})$

Reduction (cont’d)

- Induction: (A|B)

- $<A*$

- $<A*> = (\Sigma_A, Q_A, q_A, \delta_A)$
- $<B*> = (\Sigma_B, Q_B, q_B, \delta_B)$
- $<A|B*> = (\Sigma_A \cup \Sigma_B, Q_A \cup Q_B \cup \{(S0, \varepsilon, q_A), (S1, \varepsilon, q_B)\}, \delta_A \cup \delta_B \cup \{(f_A, \varepsilon, S1), (f_B, \varepsilon, S1)\})$
Reduction (cont’d)

- Induction: $A^*$

$<A> = (\Sigma, Q_A, q_0, \delta_A)$

$<A^*> = (\Sigma, Q_A \cup \{S_0, S_1\}, S_0, \{S_1\},\delta_A \cup \{(f_0, \epsilon, S_1), (S_0, \epsilon, q_1), (S_0, \epsilon, S_1), (S_1, \epsilon, S_0)\})$

Relating R.E.’s to DFAs and NFAs

Equivalence of DFAs and NFAs

- Let subsets of states be states in DFA
- Keep track of which subset you can be in

- Any string from (A) to either (D) or (CD) represents a path from A to D in the original NFA.

How NFA Works

- When NFA processes a string
  - NFA may be in several possible states
    - Multiple transitions with same label
    - $\epsilon$-transitions
- Example
  - After processing “a”
    - NFA may be in states $S_1$, $S_2$, $S_3$

Reducing NFA to DFA

- NFA may be reduced to DFA
  - By explicitly tracking the set of NFA states
- Intuition
  - Build DFA where
    - Each DFA state represents a set of NFA states
- Example

Reduction Complexity

- Given a regular expression $A$ of size $n$...
  - Size = # of symbols + # of operations
- How many states does $<A>$ have?
  - $O(n)$
  - That’s pretty good!
- NFA to DFA reduction
  - Intuition: Build DFA where each DFA state represents a set of NFA states
  - Given NFA with $n$ states, DFA may have $2^n$ states
  - Not so good, since DFAs are what we can implement easily

Can transform

Can transform

Can transform

can transform
Reducing NFA to DFA (cont.)

- Reduction applied using the subset algorithm
  - DFA state is a subset of set of all NFA states

- Algorithm
  - Input
    - NFA (Σ, Q, q₀, Fₙ, δ)
  - Output
    - DFA (Σ, R, r₀, Fₕ, δ)
  - Using
    - ε-closure(p)
    - move(p, a)

ε-transitions and ε-closure

- We say p ~ₚ q
  - If it is possible to go from state p to state q by taking only ε-transitions
  - If ∃ p, p₁, p₂, ..., pₙ, q ∈ Q such that
    - (p, ε, p₁) ∈ δ, (p₁, ε, p₂) ∈ δ, ..., (pₙ, ε, q) ∈ δ

- ε-closure(p)
  - Set of states reachable from p using ε-transitions alone
  - Set of states q such that p ~ₚ q
  - ε-closure(p) = {q | p ~ₚ q }
  - Note
    - ε-closure(p) always includes p
    - ε-closure( ) may be applied to set of states (take union)

ε-closure: Example 1

- Following NFA contains
  - S₁ ~ₚ S₂
  - S₂ ~ₚ S₃
  - S₁ ~ₚ S₃

- ε-closures
  - ε-closure(S₁) = { S₁, S₂, S₃ }
  - ε-closure(S₂) = { S₂, S₃ }
  - ε-closure(S₃) = { S₃ }
  - ε-closure( { S₁, S₂ } ) = { S₁, S₂, S₃ } ∪ { S₂, S₃ }

ε-closure: Example 2

- Following NFA contains
  - S₁ ~ₚ S₃
  - S₃ ~ₚ S₂
  - S₁ ~ₚ S₂

- ε-closures
  - ε-closure(S₁) = { S₁, S₂, S₃ }
  - ε-closure(S₂) = { S₂ }
  - ε-closure(S₃) = { S₂, S₃ }
  - ε-closure( { S₂, S₃ } ) = { S₂ } ∪ { S₂, S₃ }

ε-closure: Practice

- Find ε-closures for following NFA

- Find ε-closures for the NFA you construct for
  - The regular expression (0|1)*111(0*|1)

Calculating move(p,a)

- move(p, a)
  - Set of states reachable from p using exactly one transition on a
    - Set of states q such that (p, a, q) ∈ δ
    - move(p, a) = {q | (p, a, q) ∈ δ }
  - Note move(p, a) may be empty ∅
    - If no transition from p with label a
move(a,p) : Example 1

- Following NFA
  - $\Sigma = \{a, b\}$

- Move
  - $\text{move}(S1, a) = \{S2, S3\}$
  - $\text{move}(S1, b) = \emptyset$
  - $\text{move}(S2, a) = \emptyset$
  - $\text{move}(S2, b) = \{S3\}$
  - $\text{move}(S3, a) = \emptyset$
  - $\text{move}(S3, b) = \emptyset$

NFA $\rightarrow$ DFA Reduction Algorithm

- Input NFA $(\Sigma, Q, q_0, F_n, \delta)$, Output DFA $(\Sigma, R, r_0, F_d, \delta)$
- Algorithm
  - Let $r_0 = \varepsilon$-closure$(q_0)$, add it to $R$ // DFA start state
  - While // an unmarked state $r \in R$
    - Mark $r$ // each state visited once
    - For each $a \in \Sigma$
      - Let $S = \{ q \mid q \in r \& \text{move}(q,a) = s \}$
        - if states reached via $a$
        - Let $e = \varepsilon$-closure$(S)$ // states reached via $\varepsilon$
        - if $e \in R$
          - Let $R = e \cup R$ // add $e$ to $R$ (unmarked)
        - Let $\delta = \delta \cup (r, a, e)$ // add transition $r \rightarrow e$
      - Let $F_d = \{ r \mid \exists s \in r \text{ with } s \in F_n \}$ // final if include state in $F_n$

move(a,p) : Example 2

- Following NFA
  - $\Sigma = \{a, b\}$

- Move
  - $\text{move}(S1, a) = \emptyset$
  - $\text{move}(S1, b) = \{S2\}$
  - $\text{move}(S2, a) = \{S3\}$
  - $\text{move}(S2, b) = \emptyset$
  - $\text{move}(S3, a) = \emptyset$
  - $\text{move}(S3, b) = \emptyset$

NFA $\rightarrow$ DFA Example 1

- Start = $\varepsilon$-closure$(S1) = \{S1,S3\}$
- $r_0 = \emptyset$
- $r = \{S1,S3\}$
- Move$((S1,S3), a) = \{S2\}$
  - $e = \varepsilon$-closure$(S2) = \{S2\}$
  - $R = R \cup \{S2\} = \{S1,S3, S2\}$
  - $\delta = \delta \cup (\{S1,S3\}, a, \{S2\})$
- Move$((S1,S3), b) = \emptyset$

NFA $\rightarrow$ DFA Example 1 (cont.)

- $R = \{S1,S3\}$
  - $r = \{S3\}$
  - $\text{move}(S3, a) = \emptyset$
  - $\text{move}(S3, b) = \emptyset$
  - $F_d = \{S1,S3\}$
- Since $S3 \in F_d$
- Done!
NFA → DFA Example 2

- NFA

- DFA

\[ R = \{ [A], [B, D], [C, D] \} \]

Equivalence of DFAs and NFAs

- Any string from (A) to either (D) or (CD)
  - Represents a path from A to D in the original NFA

Minimizing DFA

- Result from CS theory
  - Every regular language is recognizable by a minimum-state DFA that is unique up to state names
- In other words
  - For every DFA, there is a unique DFA with minimum number of states that accepts the same language
  - Two minimum-state DFAs have same underlying shape

Minimizing DFA: Hopcroft Reduction

- Intuition
  - Look for states that can be distinguish from each other
    - End up in different accept / non-accept state with identical input
- Algorithm
  - Construct initial partition
    - Accepting & non-accepting states
  - Iteratively refine partitions (until partitions remain fixed)
    - Split a partition if members in partition have transitions to different partitions for same input
      - Two states \( x, y \) belong in same partition if and only if for all symbols in \( \Sigma \), they transition to the same partition
    - Update transitions & remove dead states
Splitting Partitions

- No need to split partition \( \{S,T,U,V\} \)
  - All transitions on \( a \) lead to identical partition \( P_2 \)
  - Even though transitions on \( a \) lead to different states

Splitting Partitions (cont.)

- Need to split partition \( \{S,T,U\} \) into \( \{S,T\}, \{U\} \)
  - Transitions on \( a \) from \( S,T \) lead to partition \( P_2 \)
  - Transition on \( a \) from \( R \) lead to partition \( P_3 \)

Resplitting Partitions

- Need to reexamine partitions after splits
  - Initially no need to split partition \( \{S,T,U\} \)
  - After splitting partition \( \{X,Y\} \) into \( \{X\}, \{Y\} \)
  - Need to split partition \( \{S,T,U\} \) into \( \{S,T\}, \{U\} \)

Minimizing DFA: Example 1

- DFA
  - Initial partitions
    - Accept \( \{ R \} \) → \( P_1 \)
    - Reject \( \{ S, T \} \) → \( P_2 \)
  - Split partition? → Not required, minimization done
    - \( \text{move}(S,a) = T \rightarrow P_2 \)
    - \( \text{move}(S,b) = R \rightarrow P_1 \)
    - \( \text{move}(T,a) = T \rightarrow P_2 \)
    - \( \text{move}(T,b) = R \rightarrow P_1 \)

Minimizing DFA: Example 2

- DFA
  - Initial partitions
    - Accept \( \{ R \} \) → \( P_1 \)
    - Reject \( \{ S, T \} \) → \( P_2 \)
  - Split partition? → Not required, minimization done
    - \( \text{move}(S,a) = T \rightarrow P_2 \)
    - \( \text{move}(S,b) = R \rightarrow P_1 \)
    - \( \text{move}(T,a) = S \rightarrow P_2 \)
    - \( \text{move}(T,b) = R \rightarrow P_1 \)

Minimizing DFA: Example 3

- DFA
  - Initial partitions
    - Accept \( \{ R \} \) → \( P_1 \)
    - Reject \( \{ S, T \} \) → \( P_2 \)
  - Split partition? → Yes, different partitions for \( B \)
    - \( \text{move}(S,a) = T \rightarrow P_2 \)
    - \( \text{move}(S,b) = T \rightarrow P_2 \)
    - \( \text{move}(T,a) = T \rightarrow P_2 \)
    - \( \text{move}(T,b) = R \rightarrow P_1 \)
    - DFA already minimal
**Complement of DFA**

- Given a DFA accepting language $L$
  - How can we create a DFA accepting its complement?
  - Example DFA

  ![Example DFA Diagram](image)

**Complement of DFA (cont.)**

- Algorithm
  - Add explicit transitions to a dead state
  - Change every accepting state to a non-accepting state & every non-accepting state to an accepting state
- Note this only works with DFAs
  - Why not with NFAs?

**Practice**

Make the DFA which accepts the complement of the language accepted by the DFA below.

![Practice DFA Diagram](image)

**Relating REs to DFAs and NFAs**

- Why do we want to convert between these?
  - Can make it easier to express ideas
  - Can be easier to implement

**Implementing DFAs**

It's easy to build a program which mimics a DFA

![Implementing DFAs](image)

**Implementing DFAs (Alternative)**

Alternatively, use generic table-driven DFA

```java
Given components $(E, Q, q_0, F)$ of a DFA:
let q = q_0
while (there exists another symbol s of the input string)
    s = inputSymbol()
    q = E(q, s)
if q $\in F$
    accept
else
    reject
```

- q is just an integer
- Represent $E$ using arrays or hash tables
- Represent $F$ as a set
Relating R.E.'s to DFAs and NFAs

- Regular expressions, NFAs, and DFAs accept the same languages!

Reducing DFAs to REs

- General idea
  - Remove states one by one, labeling transitions with regular expressions
  - When two states are left (start and final), the transition label is the regular expression for the DFA

Run Time of Algorithm

- Given a string $s$, how long does algorithm take to decide whether $s$ is accepted?
  - Assume we can compute $\delta(q_0, c)$ in constant time
  - Then the time per string $s$ to determine acceptance is $O(|s|)$
  - Can't get much faster!

- But recall that constructing the DFA from the regular expression $A$ may take $O(2^{|A|})$ time
  - But this is usually not the case in practice

- So there's the initial overhead, but then accepting strings is fast

Regular Expressions in Practice

- Regular expressions are typically “compiled” into tables for the generic algorithm
  - Can think of this as a simple byte code interpreter
  - But really just a representation of $(\Sigma, Q_0, q_0, (f_1), \delta_1)$, the components of the DFA produced from the r.e.

- Regular expression implementations often have extra constructs that are non-regular
  - I.e., can accept more than the regular languages
  - Can be useful in certain cases

- Disadvantages: nonstandard, plus can have higher complexity