Regular Expressions

Regular expressions (REs) describe regular languages

Regular Expression (over alphabet $\Sigma$, alphabet could be the ASCII character set for example)

- $\varepsilon$ (empty string) is a RE denoting the set $\{\varepsilon\}$
- If $a$ is in $\Sigma$, then $a$ is a RE denoting $\{a\}$
- If $x$ and $y$ are REs denoting languages $L(x)$ and $L(y)$ then
  - $x$ is an RE denoting $L(x)$
  - $x \mid y$ is an RE denoting $L(x) \cup L(y)$
  - $xy$ is an RE denoting $L(x)L(y)$
  - $x^*$ is an RE denoting $L(x)^*$

**Precedence is:** closure first, then concatenation, then alternation

*All left-associative*

$x \mid y^* z$ is equivalent to $x \mid ((y^*) z)$

Extensions to Regular Expressions

- $x^{++} = xx^*$ denotes $L(x)^+$
- $x? = x \mid \varepsilon$ denotes $L(x) \cup \{\varepsilon\}$
- $[abc] = a \mid b \mid c$ matches one character in the square bracket
- $a-z = a \mid b \mid c \mid \ldots \mid z$ range
- $[0-9a-z] = 0 \mid 1 \mid 2 \mid \ldots \mid 9 \mid a \mid b \mid c \mid \ldots \mid z$
- $[^abc]$ $^\text{^ means negation}$ matches any character except a, b or c
- $.$ = $[^\text{\n}]$ dot matches any character except the newline
- $\backslash.$ = $[^\text{\n}]$ \n means newline so dot is equivalent to $[^\text{\n}]$
- "[" matches left square bracket, meta-characters in double quotes become plain characters
- $\backslash[\text{\} matches left square bracket, meta-character after backslash becomes plain character
DFA simulation, recognizing longest matching prefix

\[
\begin{align*}
\text{state} &= s_0; \\
\text{char} &= \text{get_next_char();} \\
\text{while (char} \neq \text{EOF) \{} \\
& \quad \text{state} = \delta(\text{state, char);} \\
& \quad \text{char} = \text{get_next_char();} \\
& \quad \text{if (state} \in \text{Final) \{} \\
& \quad \quad \text{report acceptance;} \\
& \quad \quad \text{else} \\
& \quad \quad \text{report failure;} \\
\end{align*}
\]

NFA Simulation

- \(\text{move}(q, a)\) is set of states reachable by “a” from states in \(q\)
- \(\varepsilon\)-closure\((q)\) is set of states reachable by \(\varepsilon\) from states in \(q\)

\[
\begin{align*}
\text{states} &= \varepsilon\text{-closure}\(\{s_0\}\); \\
\text{char} &= \text{get_next_char();} \\
\text{while (char} \neq \text{EOF) \{} \\
& \quad \text{states} = \varepsilon\text{-closure(\text{move(states, char);)}} \\
& \quad \text{char} = \text{get_next_char();} \\
& \text{if (states} \cap \text{Final is not empty) \{} \\
& \quad \text{report acceptance;} \\
& \quad \text{else} \\
& \quad \text{report failure;} \\
\end{align*}
\]
**RE → NFA using Thompson’s Construction**

**Key idea**
- We have a pattern for constructing an NFA for each symbol & each operator
- We apply these patterns respecting the precedence order
- Patterns for operators combine the NFAs from previous steps with \( \varepsilon \) moves

---

**NFA → DFA with Subset Construction**

- States of the DFA correspond to sets of states of the NFA
  - DFA is keeping track of all the states the NFA can reach after reading an input string
- Start state of the DFA is derived from the start state of the NFA
  - Take \( \varepsilon \)-closure of the start state of the NFA and make it the start state of the DFA
- Compute \( \text{Move}(q, a) \) for each new state \( q \) of the DFA and each \( a \in \Sigma \), take the \( \varepsilon \)-closure of the result and make it a state of the DFA
- Iterative algorithm halts when there are no new states generated
**DFA Minimization**

**Important Result:** Every regular language is recognized by a minimum-state DFA that is unique up to state names.

**Main ideas in DFA minimization**
- Discover sets of equivalent states
- Represent each such set with just one state

Two states are equivalent if and only if:
- They are both accepting or rejecting and
- \( \forall \alpha \in \Sigma, \) transitions on \( \alpha \) lead to equivalent states

A partition \( P \) of \( S \)
- Each \( s \in S \) is in exactly one set \( p_i \in P \)
- The algorithm iteratively partitions the DFA’s states

**DFA Minimization**

**Details of the algorithm**
- Group states into maximal size sets, *optimistically*
  - Assume that two states are equivalent until they are proven to be un-equivalent
- Iteratively subdivide those sets, look for states that can be distinguished from each other
- States that remain grouped together are equivalent

Initial partition, has two sets:
- \{final states\} and \{all the states which are not final states\}

Splitting a set
- Split a set \( s \) into smaller groups if states in \( s \) have transitions to different sets for the same input