1/17/11 CS 360 Lecture 4 E-NFAs, Regular Expressions,
Every DFA accepts a regular language
Every regular language is accepted by an NFA
note: homework 2

Recall: DFAs \( \leftrightarrow \) NFAs

Deterministic finite automata (DFAs) are special cases of
Nondeterministic finite automata (NFAs)
Conversely, every NFA can be converted into a DFA (possibly
with an exponential blowup in \# of states)

Set of problems solvable by a DFA ("languages")
= Set of problems solvable by an NFA

Nondeterminism (guessing) may allow for more efficient solutions, but
it does not change the class of problems that are solvable.

Today:
- Define regular expressions (probably familiar—from perl?)
- Define regular languages = \( \Sigma \) languages specified by a regular expression
- Prove Theorem: Every DFA accepts a regular language, and
  Every regular language is accepted by an NFA.
  \[ \Rightarrow 3 \text{ equivalent ways of specifying the same set of problems} \]

\[ \text{DFAs} \rightarrow \text{regular expressions} \]
\[ \text{NFAs} \leftarrow \text{regular expressions} \]

operational/computational grammatical

from two very different perspectives
E-NFAs:

Common idea in computer science: How does a computational model change as we add new features?
- Oracles (in complexity theory)
- Quantumness
- Ability to apply a Fourier Transform quickly (FFT) (in algorithms)
- Ability to break RSA/your favorite encryption scheme (in cryptography)

So far: For finite automata, adding nondeterminism does not increase computational power.

Another feature to consider: E-moves

\[ p \xrightarrow{\varepsilon} q \] transition not consuming input

Examples

\[ \begin{array}{c}
  \text{Novio} \\
  \begin{array}{c}
    \text{E} \\
    \begin{array}{c}
      0,1 \\
      0 \\
      q_0 \\
      0,1 \\
    \end{array}
  \end{array}
  \rightarrow \\
  \begin{array}{c}
    \begin{array}{c}
      0 \\
      q_0 \\
    \end{array}
  \end{array}
\end{array} \]

\[ \begin{array}{c}
  \text{Integers} \\
  \begin{array}{c}
    \text{E} \\
    \begin{array}{c}
      0,9 \\
      0,9 \\
    \end{array}
  \end{array}
  \rightarrow \\
  \begin{array}{c}
    \begin{array}{c}
      0,9 \\
      e \\
    \end{array}
  \end{array}
\end{array} \]

Pascal keywords
- IF, IN, END
- X, A, 10, ALL...

Identifiers
- a space or blank

- useful descriptive tool
- combining NFAs or DFAs
- can be converted into DFAs

Formally \( E-NFA \quad N = (Q, \Sigma, \delta, q_0, F) \)
- same as NFA except \( \delta : Q \times (\Sigma \cup \{\varepsilon}\} \rightarrow 2^Q \)
- (eg, \( \delta(q_0, \varepsilon) = \{q_1, q_2, \ldots\} \))

Language \( L(N) = \{ \text{all strings for which there is an accepting path, with } \varepsilon \text{-moves allowed} \} \)
Theorem: For any ε-NFA $N$, there exists an NFA $M$ such that $L(M) = L(N)$.

$\therefore$ Allowing $\epsilon$-moves is convenient, but not necessary — does not increase computational power.

Proof: For a formal proof, we should define $L(N)$ more carefully, by defining, as before, the extended transition function $\mathcal{L} : Q \times \Sigma^* \rightarrow 2^Q$.

Main idea: $\epsilon$-Closure

Def: If $q \in Q$, $\epsilon$-closure$(q) =$ set of all states reachable from $q$ using only $\epsilon$-moves — includes $q$ itself.

Example:

$N$: $\begin{array}{c}
\xymatrix{
q_0 
\ar[rr]^\epsilon & & q_2 
\ar[rr]^a & & q_3 
\ar[l]^\epsilon & \\
q_1 & & q_2 & & q_3 
\ar[ll]^\epsilon & & \\
q_4 & & q_5 & & q_6 
\ar[ll]^\epsilon & & 
\end{array}$

$\begin{array}{c}
q \xrightarrow{\epsilon}\text{-\text{CL}}(q) \\
q_0 \rightarrow \{q_0, q_2, q_3\} \\
q_1 \rightarrow \{q_1\} \\
q_2 \rightarrow \{q_2, q_3\} \\
q_3 \rightarrow \{q_3, q_4, q_5, q_6\} \\
q_4 \rightarrow \{q_4, q_5, q_6\} \\
q_5 \rightarrow \{q_5, q_6\} \\
q_6 \rightarrow \{q_6\} \\
\end{array}$

One step in $M$ should simulate in $N$ any number of $\epsilon$-moves followed by a step $C \xrightarrow{\sigma} C$, labeled by an element of $\Sigma$, followed by any number of $\epsilon$-moves.

$M$: $\begin{array}{c}
\xymatrix{
O 
\ar[rr]^\epsilon & & O 
\ar[rr]^a & & O 
\ar[l]^\epsilon & \\
O & & O & & O 
\ar[ll]^\epsilon & & \\
O & & O & & O 
\ar[ll]^\epsilon & & 
\end{array}$

$\iff$ Note: No more $\epsilon$-moves!

$s_M(q, a) = \epsilon\text{-CL}[s_N(\epsilon\text{-CL}(q), a)]$
Example (ε-NFA to NFA conversion):

Note: $q_0$ must be added to $F$ in $M$, since $ε \in L(N)$.
Example: 2-way finite automata
allowing a DFA to step forward or backward does not
increase the computational power.

Let \( M = (Q, \Sigma, \delta, q_0, F) \) be a 2-way DFA
with \( \delta: Q \times \Sigma \rightarrow Q \times \{ \text{forward, backwards} \} \).

Observe: If \( x \in L(M) \), then \( M \) accepting \( x \) can look
at any one position at most \( |Q| \) times.

Why? If there is a position that \( M \) looks at \( |Q|+1 \)
times, then two of those times it must have been
in the same state (pigeonhole principle)
\( \Rightarrow M \) enters an infinite loop
(repeating the steps between those two visits forever)
\( \Rightarrow M \) does not accept \( x \), \( x \notin L(M) \).

- Rough idea to convert \( M \) into an \( \varepsilon \)-NFA \( N \):

\[ \text{input string} \rightarrow x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow x_4 \rightarrow x_5 \]

\( N \) will simulate this forward/backward execution
by going forward only, by keeping track of the whole
stack of states (at most \( |Q| \) high)—getting there
via an \( \varepsilon \)-move guess—and verifying consistency between
the stacks one step at a time.
\( N \) has \( \leq (|Q|+1)^{|Q|} \) \( |Q| \) states

Converting \( N \) to a DFA will give another exponential blowup.

- Is there a more efficient conversion?

Recall: Easy, using a memory DFA, if \( M \) never backtracks by
more than \( O(1) \) steps.
Regular expressions
- algebraic notation for specifying a language \( L \subseteq \Sigma^* \)

Operators:
- \( + \) (union)
- \( \cdot \) (concatenation)
- \( \ast \) (Kleene closure)

Operands:
- all characters \( a \in \Sigma \)
- \( \emptyset \) (empty set)
- \( \epsilon \) (empty string)

Example: \((\emptyset + a b) \ast (\epsilon + c) \ast \epsilon \ast\)

Def.: For a regex \( r \), \( L(r) \) is the language it specifies, defined inductively as follows:
- For a \( a \in \Sigma \), \( L(a) = \Sigma \cdot a \)
  \( L(\emptyset) = \emptyset \)
  \( L(\epsilon) = \epsilon \cdot \)
- Union: \( L(r + s) = L(r) \cup L(s) \)
- Concatenation:
  \( L(r \cdot s) = L(r) \cdot L(s) \)
  \( \equiv \{ \text{strings } x \cdot y \mid x \in L(r), y \in L(s) \} \}
- Kleene closure:
  \( L(r^\ast) = \{ \text{strings } x_1 x_2 \cdots x_k \mid k \geq 0, x_i \in L(r) \} \)
  \( = L(\epsilon) \cup L(r) \cup L(r \cdot r) \cup L(r \cdot r \cdot r) \cup \cdots \)

(\text{in other words, for a language } L, \text{ let}
L^0 = \{ \epsilon \}, L^1 = L, L^2 = L \cdot L, \ldots, L^{k+1} = L^k \cdot L, \\
L^* = L^0 \cup L^1 \cup L^2 \cup L^3 \cup \cdots, L(r^\ast) = L(r^\ast)\}

Examples:
- \( r = \epsilon + 1 \) \( \Rightarrow \) \( L(r) = \{ \epsilon, 1 \} \)
- \( s = \epsilon + 0 + 1 \) \( \Rightarrow \) \( L(s) = \{ \epsilon, 0, 1 \} \)
- \( t = r \cdot s \) \( \Rightarrow \) \( L(t) = \{ \epsilon, 0, 1, 10, 11 \} \)
- \( r = 0 + 1 \) \( \Rightarrow \) \( L(r) = \{ 0, 1 \} \)
- \( s = r^\ast \) \( \Rightarrow \) \( L(s) = \{ \text{all bit strings (including } \epsilon \} \} \)
- \( r = 00 \) \( \Rightarrow \) \( L(r^\ast) = \{ \text{all even-length strings of } 0 \cdot \} \)
Positive closure:

\[ L^+ = L \cup L^2 \cup L^3 \cup \cdots \]
\[ L(r^+) = L(r) \cup L(r^2) \cup L(r^3) \cup \cdots \]

Notice:

- \( r^+ = r \cdot r^* \) (meaning \( L(r^+) = L(r \cdot r^*) = L(r^* \cdot r) \))
- \( L(r^*) = L(\varepsilon + r^*) \)
- \( L(\varepsilon^*) = L(\varepsilon) = \{ \varepsilon \} \) (since \( \varepsilon^2 = \varepsilon^3 = \cdots = \varepsilon \))
- \( L(\emptyset^*) = L(\emptyset) \cup L(\emptyset^1) \cup \cdots = \{ \emptyset \} \)
- \( L(a^*) = \{ \varepsilon, a, aa, aaa, \ldots \} \)

Examples:

- \( L(a^*, b^*) = L(a^*) \cdot L(b^*) \)
  - \( = \{ \varepsilon, a, b, ab, ba, aa, aaa, aab, \ldots \} \)
  - \( = \{ \text{all } x \in \{a, b\}^* \text{ where each } a \text{ comes before all } b's \} \)
- \( L(a^*, b^*) \neq L((a+b)^*) = \{a, b\}^* \) all strings in a's and b's
- \( L((a+b)^*) = L((a^*, b^*)^*) \)

- \( L((0+1)^* \cdot 1) = L((0+1)^*) \cdot L(1) \)
  - \( = \{0, 1\}^* \cdot \{1\} \)
  - \( = \{ \text{all strings ending with } 1 \} \)
- \( L((0+1)^*(01+0)) = \{ \text{all strings ending with } 01 \text{ or } 10 \} \)

- \( r = (0+10)^* (\varepsilon+1) \)
  \( L(r) = \{ \text{all } \varepsilon \text{ strings without two consecutive } 1s \} \)
  - \( \varepsilon, 00101 \in L(r) \)

- \( r_1 + r_2 \cdot r_3^* = r_1 + (r_2 \cdot (r_3^*)) \)
  - when we omit parentheses, \( \cdot \) has highest precedence,
  then \( + \), then \( \) (just like in algebra: powers, multiplication, addition)

- Associativity: \( (r_1 + r_2) \cdot r_3 = r_1 \cdot (r_2 + r_3) = r_1 + r_2 \cdot r_3 \)
- \( (r_1 + r_2) + r_3 = r_1 + (r_2 + r_3) = r_1 + r_2 + r_3 \)
- Distributivity: \( L[(r_1 + r_2) \cdot r_3] = L[r_1 \cdot (r_2 + r_3)] \)