CS 536 Announcements for Tuesday, February 1, 2022

Course websites:

pages.cs.wisc.edu/~hasti/cs536/
www.piazza.com/wisc/spring2022/compsci536

- waitlisted folks: feel free to add yourself to Piazza

Programming Assignment 1
- test code due Friday, Feb. 4 by 11:59 pm
- other files due Tuesday, Feb. 8 by 11:59 pm

Last Time
- start scanning
- finite state machines
  - formalizing finite state machines
  - coding finite state machines
  - deterministic vs non-deterministic FSMs

Today
- non-deterministic FSMs
- equivalence of NFAs and DFAs
- regular languages
- intro regular expressions

Next Time
- regular expressions
- regular expressions → DFAs

Recall
- scanner: converts a sequence of characters to a sequence of tokens
- scanner implemented using FSMs
- FSMs can be DFA or NFA

Creating a scanner

\[
\text{scanner} = \text{token to regex} + \text{regex to NFA} + \text{NFA to DFA} + \text{DFA to code}
\]
NFAs, formally

finite state machine $M = (Q, \Sigma, \delta, q_0, F)$

$L(M) = \text{the language of FSM } M = \text{set of all strings } M \text{ accepts}$

Example:

"Running" an NFA

To check if a string is in $L(M)$ of NFA $M$, simulate set of choices it could make.

The string is in $L(M)$ iff there is at least one sequence of transitions that
  • consumes all input (without getting stuck)
  • ends in one of the final states
NFA and DFA are equivalent

Two automata $M$ and $M^*$ are equivalent iff $L(M) = L(M^*)$

**Lemmas to be proven:**

**Lemma 1:** Given a DFA $M$, one can construct an NFA $M^*$ that recognizes the same language as $M$, i.e., $L(M^*) = L(M)$

**Lemma 2:** Given an NFA $M$, one can construct a DFA $M^*$ that recognizes the same language as $M$, i.e., $L(M^*) = L(M)$

**Proving Lemma 2**

**Lemma 2:** Given an NFA $M$, one can construct a DFA $M^*$ that recognizes the same language as $M$, i.e., $L(M^*) = L(M)$

**Part 1:** Given an NFA $M$ without $\varepsilon$-transitions, one can construct a DFA $M^*$ that recognizes the same language as $M$

**Part 2:** Given an NFA $M$ with $\varepsilon$-transitions, one can construct a NFA $M^*$ without $\varepsilon$-transitions that recognizes the same language as $M$
**NFA without $\varepsilon$-transitions to DFA**

*Observation*: we can only be in finitely many subsets of states at any one time

*Idea*: to do NFA $M \rightarrow$ DFA $M^*$, use a single state in $M^*$ to simulate sets of states in $M$.

Suppose $M$ has $|Q|$ states. Then $M^*$ can have only up to $|Q|$ states.

Why?

---

**Example**
NFA without $\varepsilon$-transitions to DFA

Given NFA $M$:

**Build new DFA $M^*$**

**To build DFA:** Add an edge from state $S$ on character $c$ to state $S^*$ if $S^*$ represents the set of all states that a state in $S$ could possibly transition to on input $c$.

**$\varepsilon$-transitions**

**Example:** $x^n$, where $n$ is even or divisible by 3
Eliminating $\epsilon$-transitions

**Goal:** given NFA $M$ with $\epsilon$-transitions, construct an $\epsilon$-free NFA $M^*$ that is equivalent to $M$

**Definition:** *epsilon closure*

$$\text{eclose}(s) = \text{set of all states reachable from } s \text{ using 0 or more epsilon transitions}$$
Summary of FSMs

DFAs and NFAs are equivalent
- an NFA can be converted into a DFA, which can be implemented via the table-drive approach

$\epsilon$-transitions do not add expressiveness to NFAs
- algorithm to remove $\epsilon$-transitions

Regular Languages and Regular Expressions

Regular language
Any language recognized by an FSM is a regular language
Examples:
- single-line comments beginning with //
- hexadecimal integer literals in Java
- C/C++ identifiers
- $\{\epsilon, ab, abab, ababab, abababab, \ldots\}$

Regular expression
= a pattern that defines a regular language

regular language: set of (potentially infinite) strings
regular expression: represents a set of (potentially infinite) strings by a single pattern
Example: $\{\epsilon, ab, abab, ababab, abababab, \ldots\} \longleftrightarrow (ab)^*$

Why do we need them?
- Each token in a programming language can be defined by a regular language
- Scanner-generator input = one regular expression for each token to be recognized by the scanner
Regular expressions

Formal definition

A regular expression over an alphabet $\Sigma$ is any of the following:

- $\emptyset$ (the empty regular expression)
- $\epsilon$
- $a$ (for any $a \in \Sigma$)

Moreover, if $R_1$ and $R_2$ are regular expressions over $\Sigma$, then so are: $R_1 \mid R_2$, $R_1 \cdot R_2$, $R_1^*$

Regular expressions as an expression language

regular expression $=$ pattern describing a set of strings

operands: single characters, epsilon

operators:

- alternation ("or"): $a \mid b$
- concatenation ("followed by"): $a.b \quad ab$
- iteration ("Kleene star"): $a^*$

Conventions

- $aa$ is $a.a$
- $a+$ is $aa^*$
- letter $=$ $a|b|c|d|...|y|z|A|B|...|Z$
- digit $=$ $0|1|2|...|9$
- $\text{not}(x)$ is all characters except $x$
- parentheses for grouping and overriding precedence, e.g., $(ab)^*$

Example: single-line comments beginning with //

Example: hexadecimal integer literals in Java

- must start 0x or 0X
- followed by at least one hexadecimal digit (hexdigit)
  - hexdigit $=$ $0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f, A, B, C, D, E, F$
- optionally can add long specifier (l or L) at end