CS 536 Announcements for Wednesday, February 1, 2023

Course websites:
   - pages.cs.wisc.edu/~hasti/cs536
   - www.piazza.com/wisc/spring2023/compsci536

Programming Assignment 1
- test code due Friday, Feb. 3 by 11:59 pm
- other files due Tuesday, Feb. 7 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
- regular expressions

Next Time
- regular expressions $\rightarrow$ DFAs
- language recognition $\rightarrow$ tokenizers
- scanner generators
- JLex

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

$L(M) =$ the language of FSM $M =$ set of all strings $M$ 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 ε-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 $2^{|Q|}$ states.

Why?

---

**Example**

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

Given NFA $M$:

![Diagram of NFA](image)

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

![Diagram of $\varepsilon$-transitions](image)
Eliminating $\varepsilon$-transitions

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

Definition: epsilon closure

eclose($s$) = set of all states reachable from $s$ 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

ε-transitions do not add expressiveness to NFAs
• algorithm to remove ε-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
• {ε, ab, abab, ababab, abababab, …}

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: {ε, ab, abab, ababab, abababab, …} ⟷ (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

Formal definition
A regular expression over an alphabet Σ is any of the following:
• ∅ (the empty regular expression)
• ε
• a (for any a ∈ Σ)

Moreover, if R₁ and R₂ are regular expressions over Σ, then so are: R₁ | R₂ , R₁ · R₂ , R₁*
Regular expressions (as an expression language)

regular expression = pattern describing a set of strings

operands: single characters, epsilon

operators:

alternation ("or"):
  a | b

catenation ("followed by"):
  a.b  ab

iteration ("Kleene star"):
  a*

Conventions

aa  is a.a
a+  is aa*

letter  is a|b|c|d|…|y|z|A|B|…|Z

digit  is 0|1|2|…|9

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

Example: C/C++ identifiers (with one added restriction)
  • sequence of letters/digits/underscores
  • cannot begin with a digit
  • cannot end with an underscore