CS 536 Announcements for Wednesday, January 31, 2024

Course websites:

`pages.cs.wisc.edu/~hasti/cs536`
`www.piazza.com/wisc/spring2024/compsci536`

Programming Assignment 1

- test code due Sunday, Feb. 4 by 11:59 pm
- other files due Thursday, 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
- regular expressions

Next Time

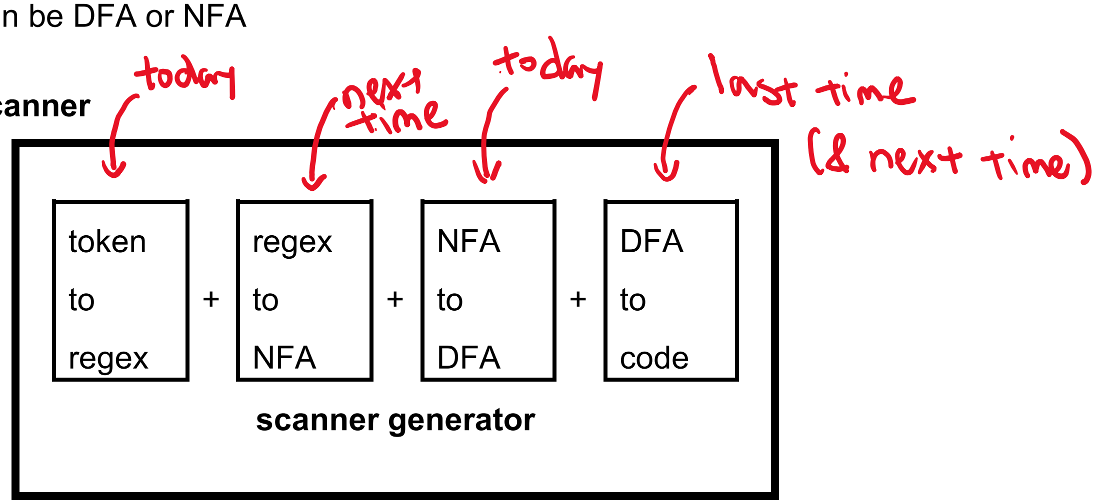
- regular expressions → DFAs
- language recognition → tokenizers
- scanner generators
- JLex

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

scanner =



NFAs, formally

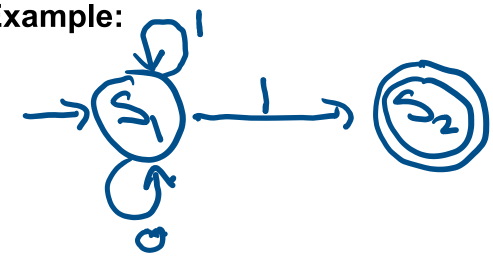
$\mathcal{P}(S)$ = power set of S = set of all subsets of S

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

- Q: finite set of states
- Σ: alphabet (symbol-characters)
- δ: transition function : $\delta : Q \times \Sigma \to \underbrace{\mathcal{P}(Q)}_{\text{↳ } Q \text{ if DFA}}$
- q: start state, $q \in Q$
- F: final states, $F \subseteq Q$

L(*M*) = the language of FSM *M* = set of all strings *M* accepts

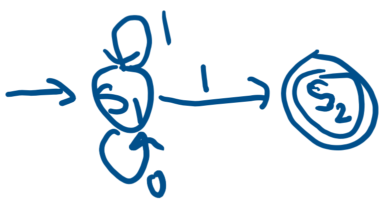
Example:



	0	1
S_1	$\{S_1\}$	$\{S_1, S_2\}$
S_2	$\{\}$	$\{\}$

"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) and
- 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 ε-transitions, one can construct a DFA *M** that recognizes the same language as *M*

Part 2: Given an NFA *M* with ε-transitions, one can construct a NFA *M** without ε-transitions that recognizes the same language as *M*

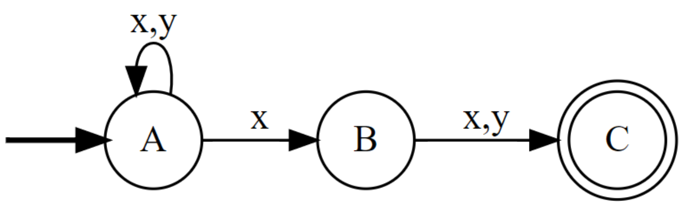
NFA w/ε $\xrightarrow{\text{Part 2}}$ NFA w/o ε $\xrightarrow{\text{Part 1}}$ DFA

NFA without ε-transitions to DFA

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

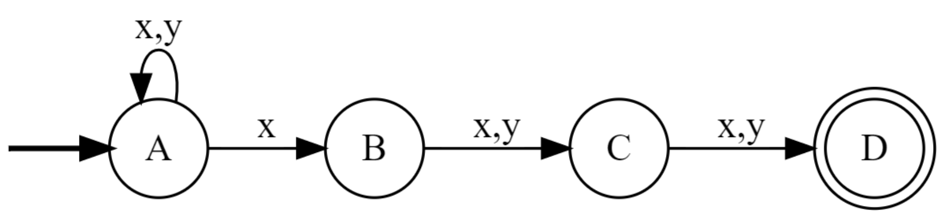
Idea: to do NFA M → DFA M^*, use a single state in M^* to simulate sets of states in M

Suppose M has $|Q|$ states. Then M^* can have only up to $2^{|Q|}$ states.
Why?



A	B	C	
0	0	0	$= \{\}$
0	0	1	$= \{C\}$
0	1	0	$= \{B\}$
0	1	1	$= \{B, C\}$
1	0	0	$= \{A\}$
1	0	1	$= \{A, C\}$
1	1	0	$= \{A, B\}$
1	1	1	$= \{A, B, C\}$

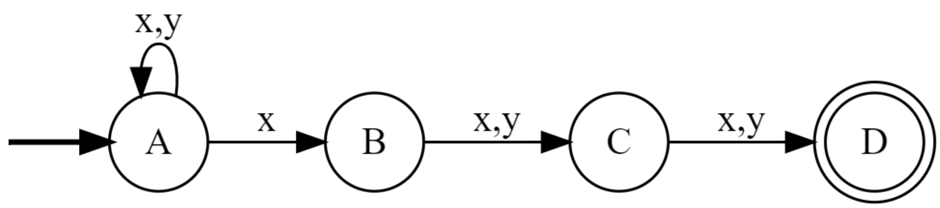
Example



	x	y
A	$\{A, B\}$	$\{A\}$
B	$\{C\}$	$\{C\}$
C	$\{D\}$	$\{D\}$
D	$\{\}$	$\{\}$

NFA without ε-transitions to DFA

Given NFA *M*:

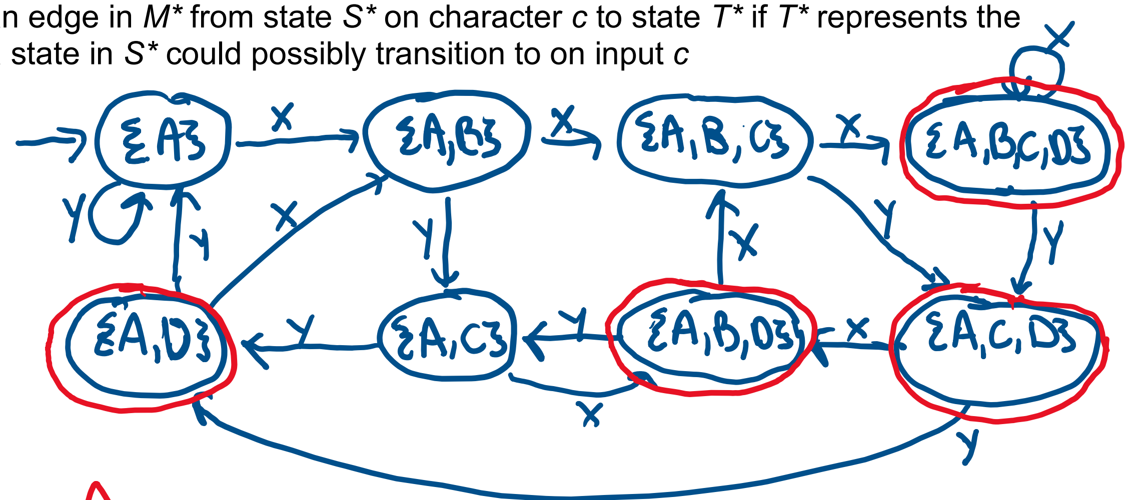


Build new DFA *M** where $Q^* \subseteq \mathcal{P}(Q)$ $|Q^*| \le 2^{|Q|}$

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

	x	y
A	{A,B}	{A}
B	{C}	{C}
C	{D}	{D}
D	{ }	{ }

final state in M

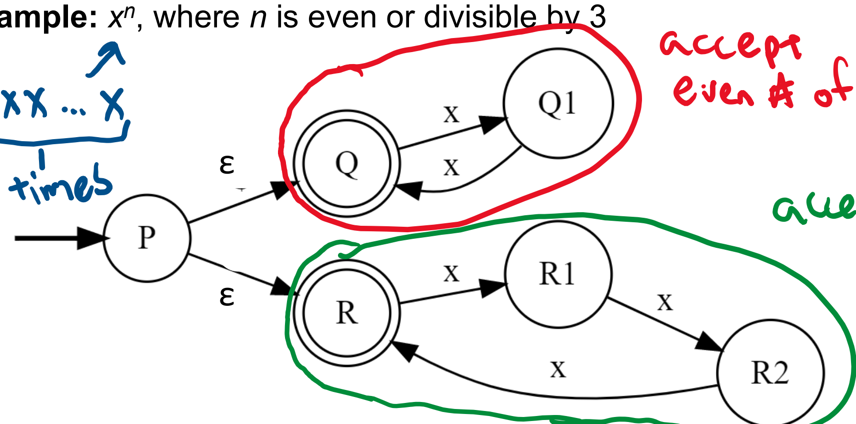


Any state in M* whose subset contains a final state of M is a final state in M*

ε-transitions

Example: x^n, where n is even or divisible by 3

$\underbrace{xxx \ldots x}_{n \text{ times}}$



accept even # of x's

accepts # of x's divisible by 3

useful for taking union of 2 FSMs

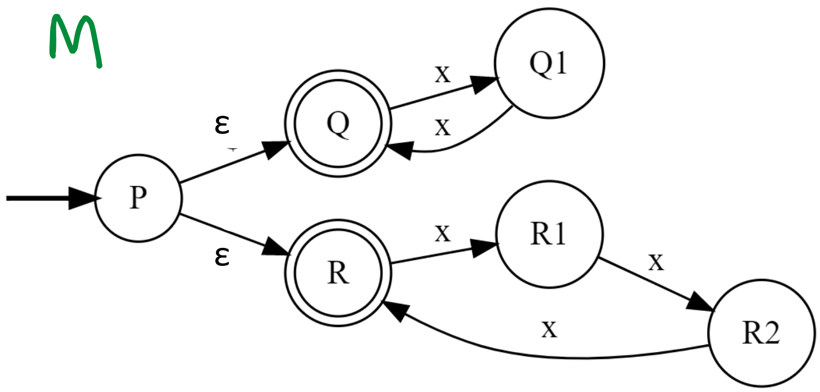
Eliminating *ε*-transitions

Goal: given NFA *M* with *ε*-transitions, construct an *ε*-free NFA *M** that is equivalent to *M*

Definition: *epsilon closure*

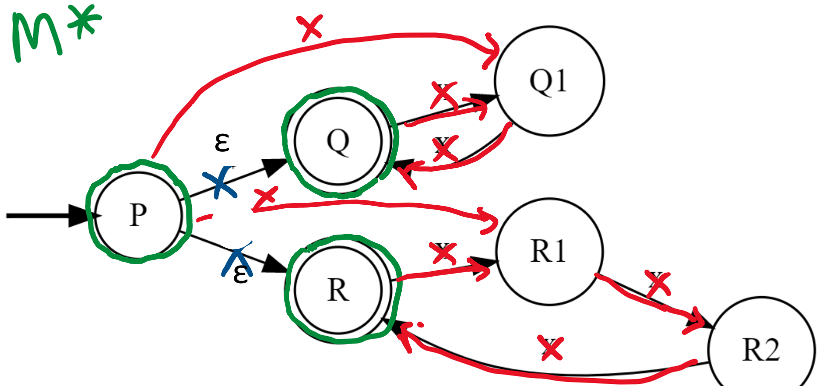
eclose(*S*) = set of all states reachable from *S* using 0 or more epsilon transitions

M



	eclose
P	{P, Q, R}
Q	{Q}
R	{R}
Q1	{Q1}
R1	{R1}
R2	{R2}

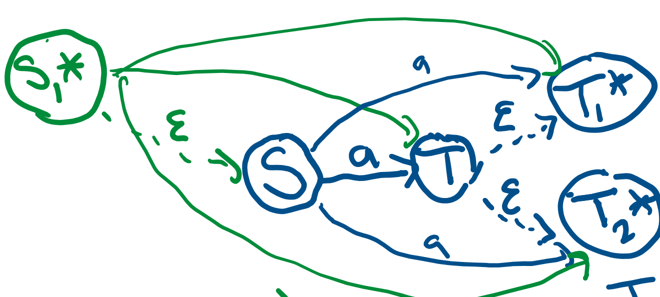
M*



1) Make S an accepting state in M* iff eclose(S) contains an accepting state of M

2) For each edge $S \xrightarrow{a} T$ in M (not ε) add edge to M*
$S^* \xrightarrow{a} T^*$
for each S^* s.t. $S \in$ eclose(S^*)
& for each T^* s.t. $T^* \in$ eclose(T)

3) Delete all edges labeled with ε



$T_1^* \in$ eclose(T)

$S \in$ eclose(S_1^*)

Summary of FSMs

DFAs and NFAs are equivalent

- an NFA can be converted into a DFA, which can be implemented via the table-driven 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, …} ← try creating FSM

Regular expression (regex)

= a pattern that defines a regular language

regular language: (potentially infinite) set of strings

regular expression: represents a (potentially infinite) set of strings by a single pattern

Example: {$\boldsymbol{\varepsilon}$, 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

→ regex's are inputs to scanner generator

Formal definition

A **regular expression** over an alphabet Σ is any of the following:

- ∅ (the empty regular expression)
- ε
- *a* (for any $a \in \Sigma$)

Moreover, if R_1 and R_2 are regular expressions over Σ, 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 ε

operators:

Precedence
low → high

alternation ("or"): a | b   matches a, matches b

concatenation ("followed by"): a.b   ab   matches ab

catenation

iteration ("Kleene star"): a*   matches 0 or more a's

Kleene closure, closure

↳ ε, a, aa, aaa, ...

Conventions

aa	is a.a
a+	is aa*
L letter	is a|b|c|d|…|y|z|A|B|…|Z
D digit	is 0|1|2|…|9
not(x)	is all characters except x

parentheses for grouping and overriding precedence, e.g., (ab)*

ab* ≡ a(b*)

Example: single-line comments beginning with //

// not('\n')* '\n'

newline

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

0(x|X) hexdigit$^+$ (ε|l|L)

Example: C/C++ identifiers (with one added restriction)

- sequence of letters/digits/underscores
- cannot begin with a digit
- cannot end with an underscore