Nondeterministic Finite Automata

CS 536

Previous Lecture

Scanner: converts a sequence of characters to a sequence of tokens

Scanner and parser relationship

Scanner implemented using FSMs

FSM: DFA or NFA

This Lecture

NFAs from a formal perspective

Theorem: NFAs and DFAs are equivalent

Regular languages and Regular expressions

NFAs, formally

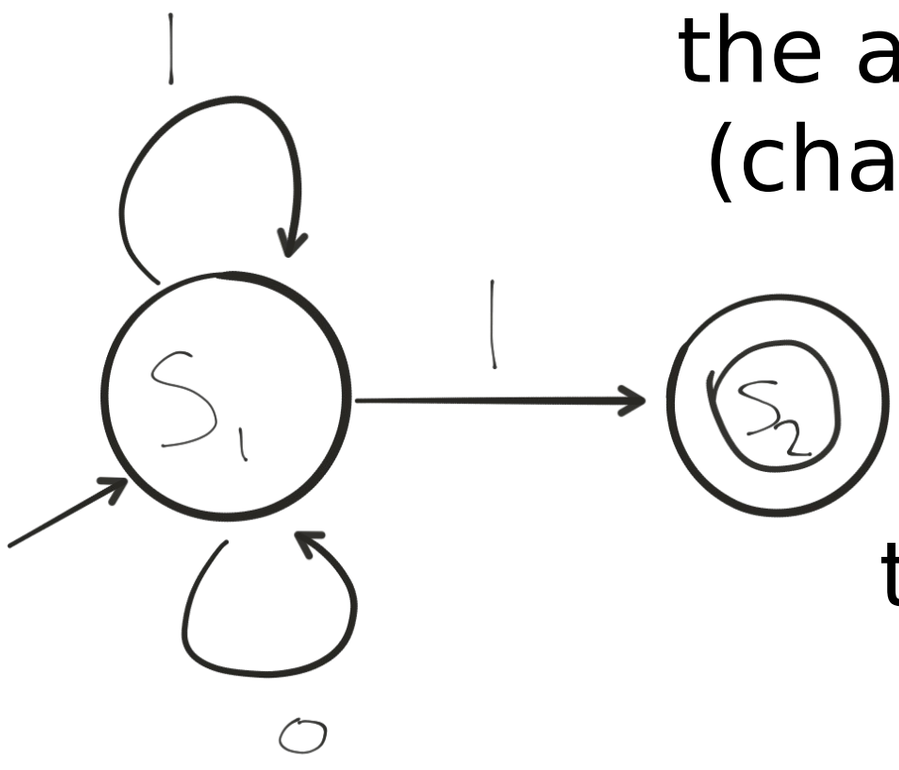
$$M \equiv (Q, \Sigma, \delta, q, F)$$

finite set of states

the alphabet (characters)

start state $q \in Q$

final states $F \subseteq Q$



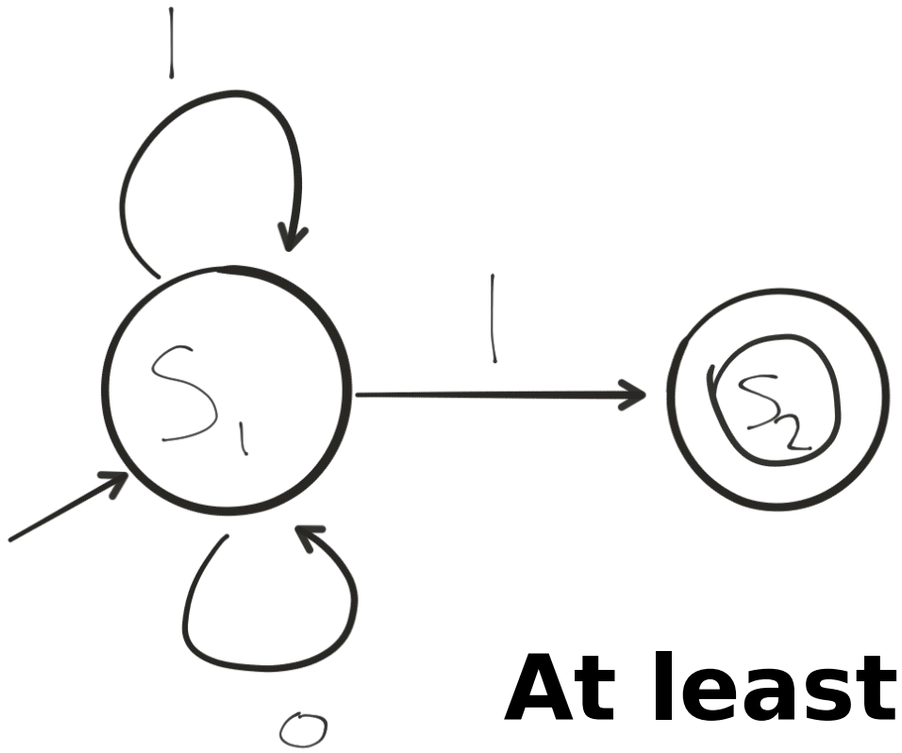
transition function

$$\delta : Q \times \Sigma \rightarrow 2^Q$$

	0	1
s1	{s1}	{s1, s2}
s2	{}	{}

NFA

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



1	1	1	
s1	s2	st	st
s1	s1	s2	st
s1	s1	s1	**s2**
s1	s1	s1	s1

At least one sequence of transitions tha

Consumes all input (without getting s

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

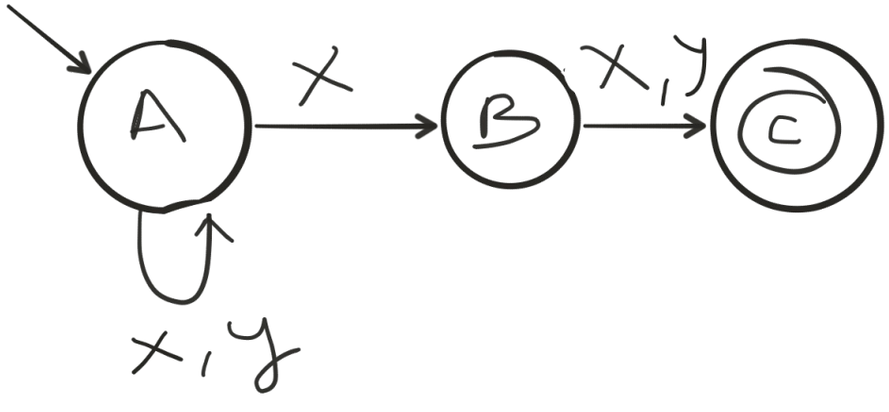
Lemma 2: Given an NFA M, one can construct a DFA M’ that recognizes the same language as M, i.e., L(M’) = L(M)

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

$2^{|Q|}$ possible combinations of states

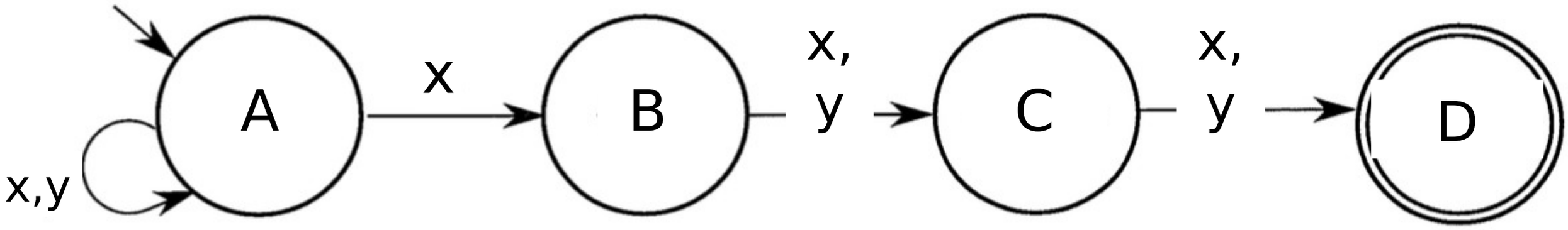
Why?

Why 2^|Q| states?



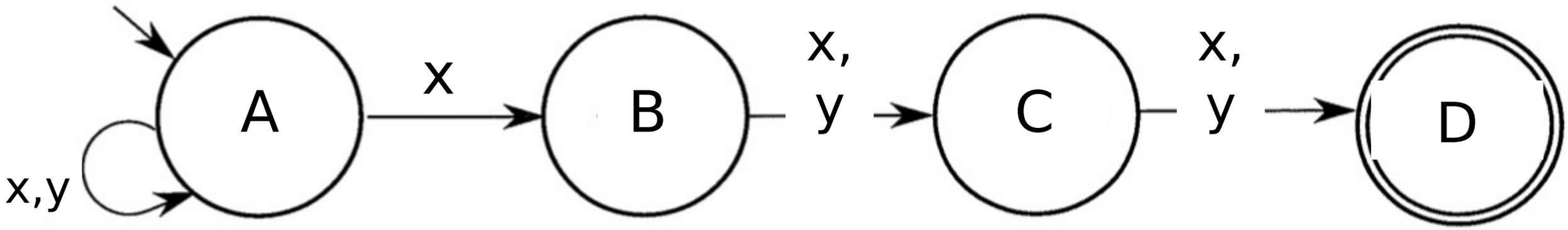
Build DFA that tracks set of states the NFA is in!

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}

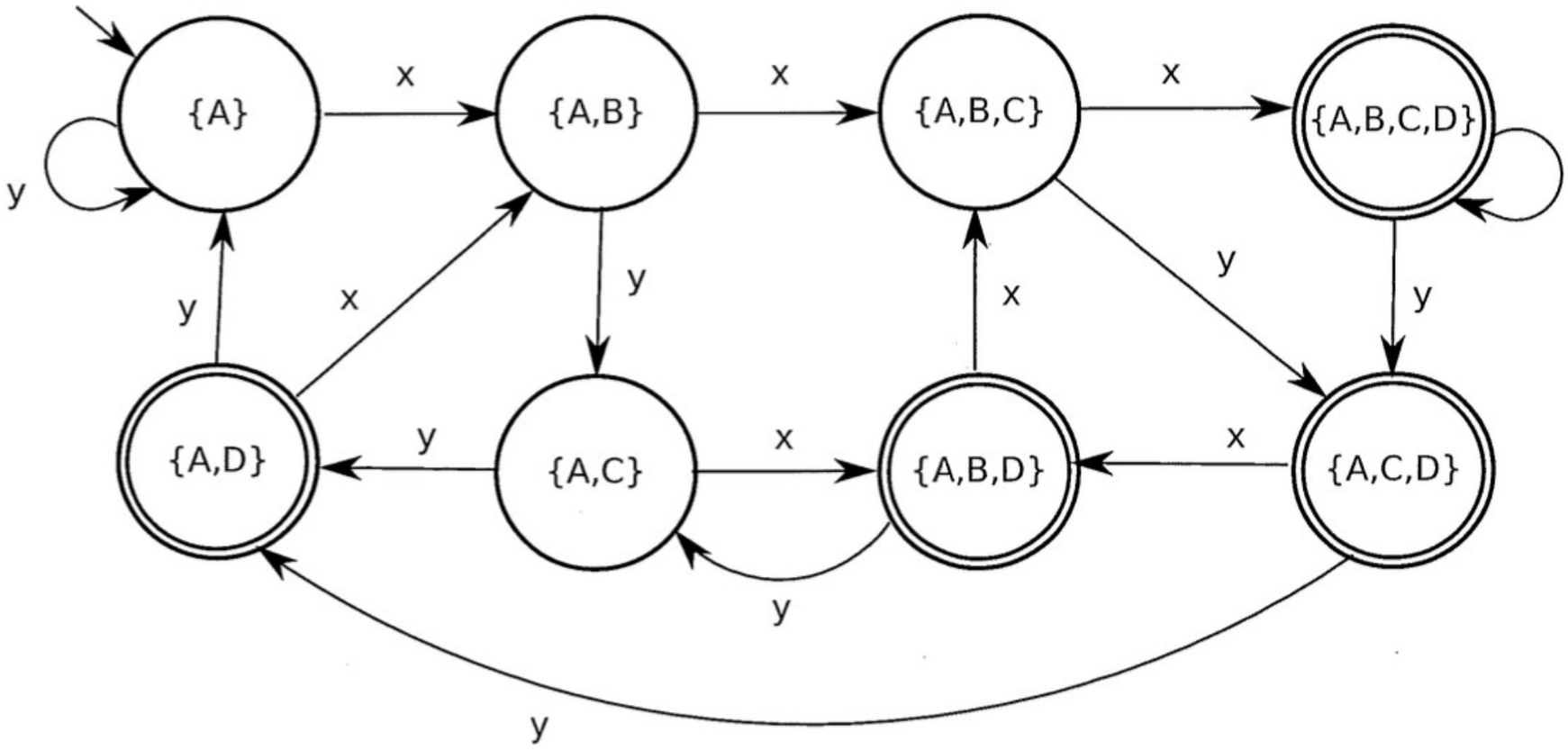


Defn: let succ(s,c) be the set of choices the NFA could make in state s with character c

succ(A,x) = {A,B}
succ(A,y) = {A}
succ(B,x) = {C}
succ(B,y) = {C}
succ(C,x) = {D}
succ(C,y) = {D}



Build new DFA M’ where Q’ = 2^Q

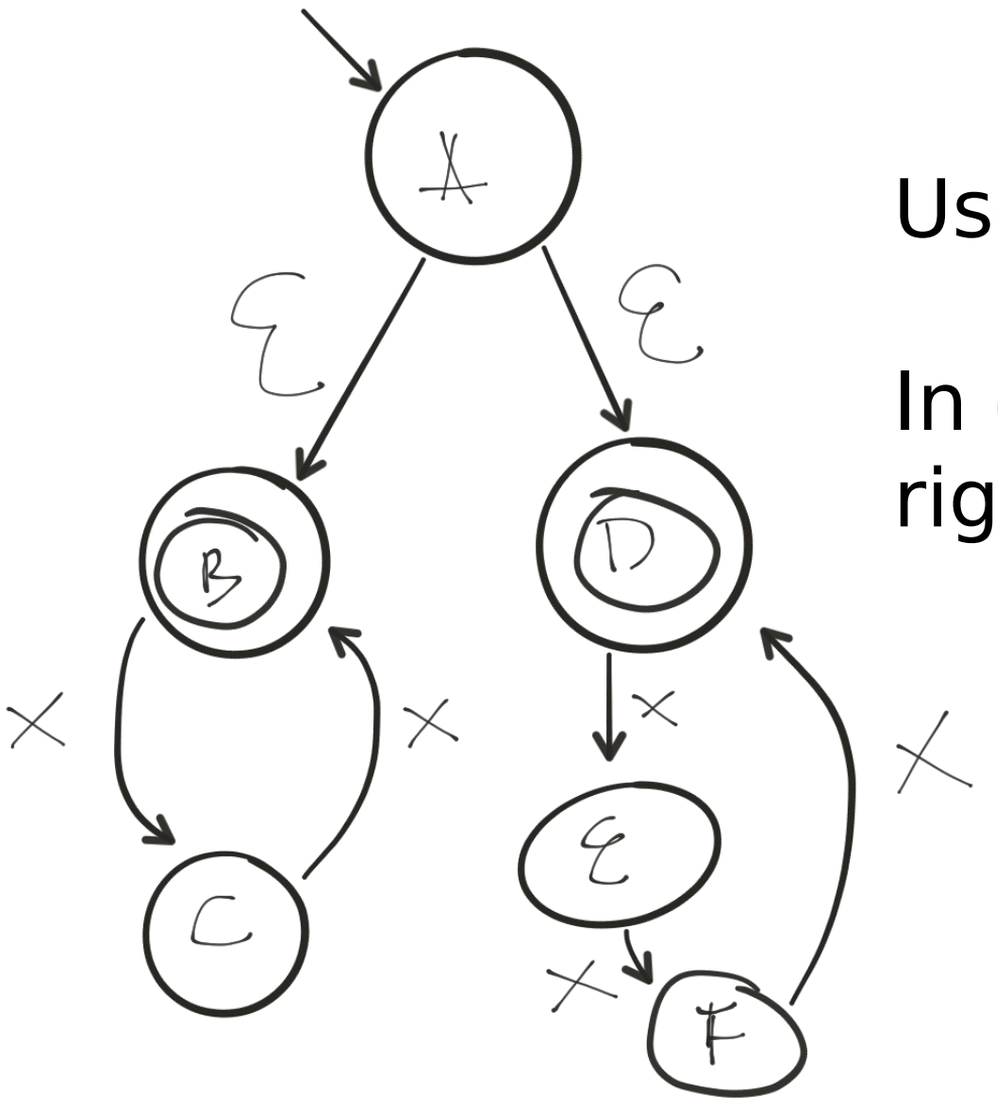


ucc(A,x) = {A,B}
ucc(A,y) = {A}
ucc(B,x) = {C}
ucc(B,y) = {C}
ucc(C,x) = {D}
ucc(C,y) = {D}

To build DFA: Add an edge from state S on character c to state S’ if S’ represents the union of states that all states in S could possibly transition to on input c

ε-transitions

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



Useful for taking union of two FSMs

In example, left side accepts even n,
right side accepts n divisible by 3

	x	x
AB	C	B
AD	E	F
A		

Eliminating ε-transitions

We want to construct ε-free FSM M’ that is equivalent to M

Definition:

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

zero or more epsilon transitions

M’ components

s is an accepting state of M’ iff eclose(s) contains an accepting state

s —c—> t is a transition in M’ iff
q —c—> t for some q in eclose(s)

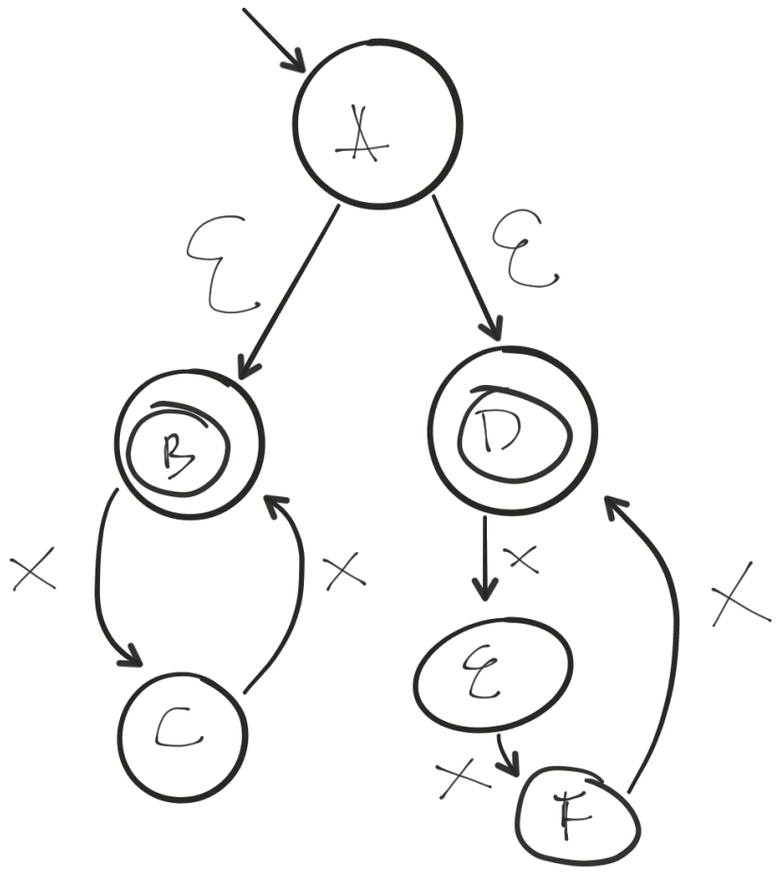
Eliminating ε-transitions

We want to construct ε-free NFA M’ that is equivalent to M

Definition: Epsilon Closure

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

zero or more epsilon transitions

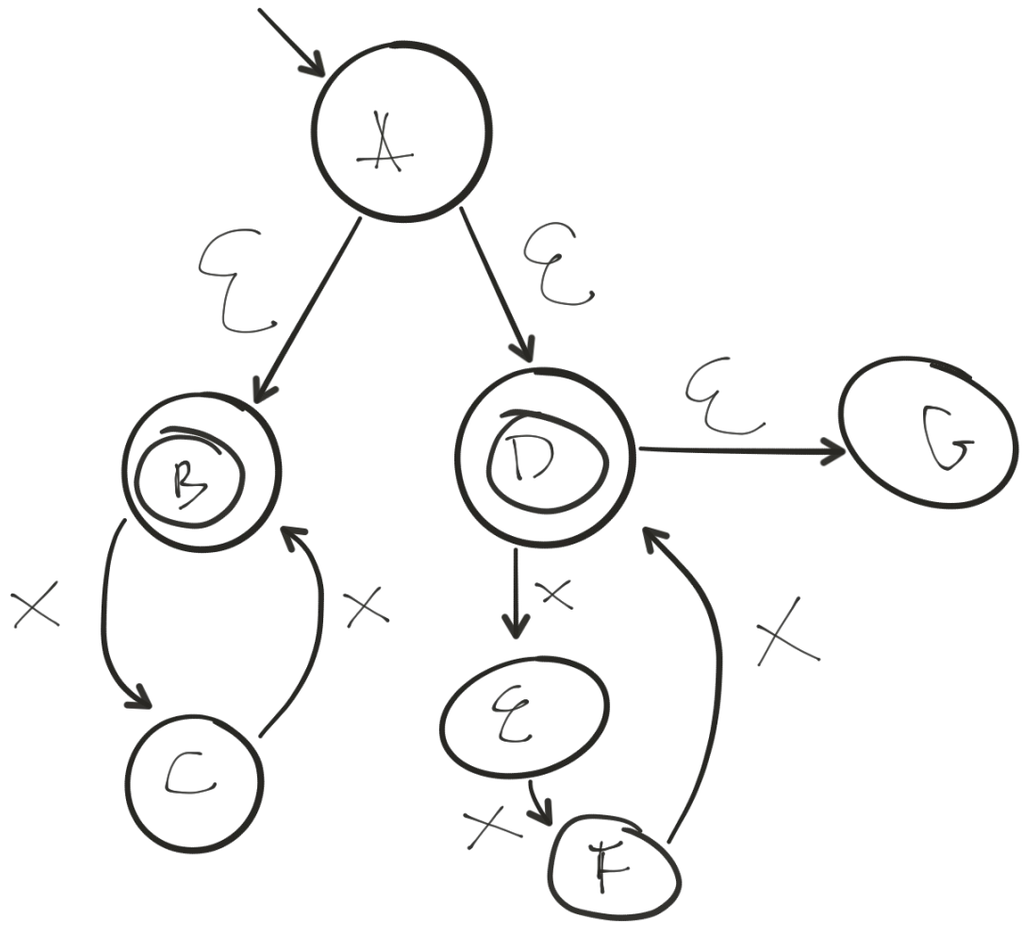


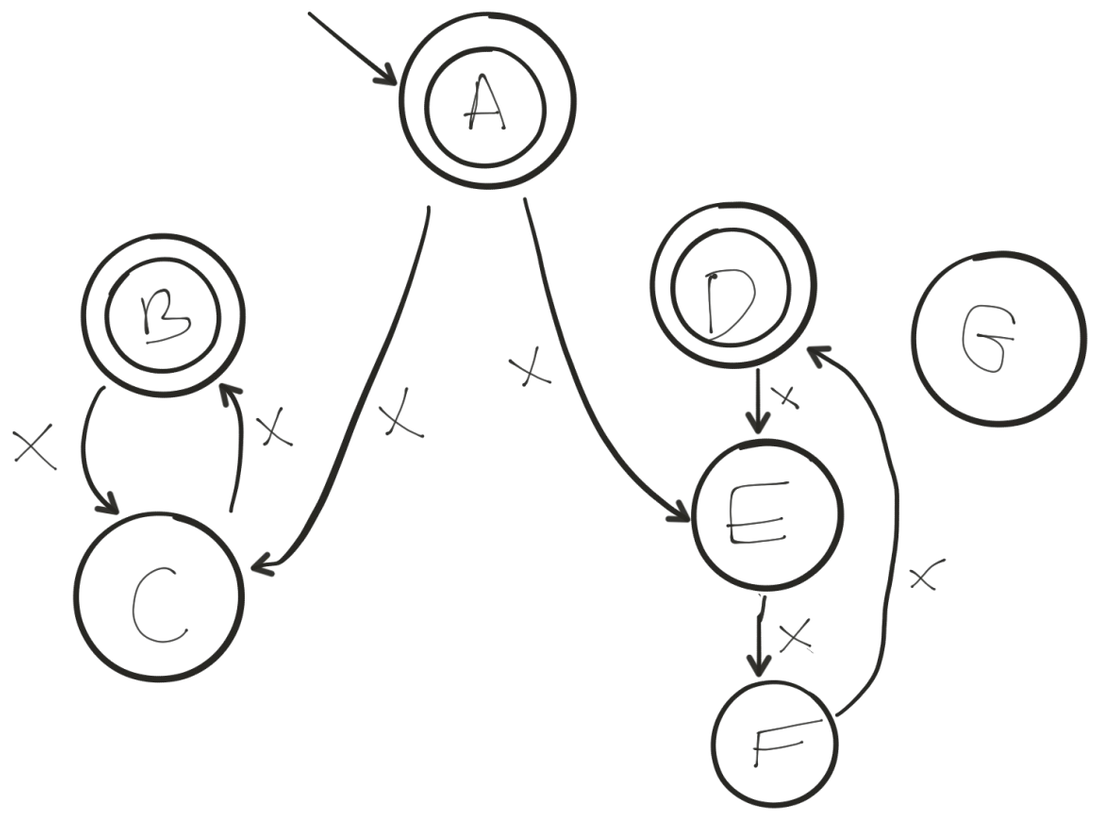
	eclose
A	{A, B, D}
B	{B}
C	{C}
D	{D}
E	{E}
F	{F}

Def: eclose(s) = set of all states reachable from s in zero or more epsilon transitions

s is an accepting state of M’ iff eclose(s) contains an accepting state

s —c—> t is a transition in M’ iff
q —c—> t for some q in eclose(s)





Recap

NFAs and DFAs are equally powerful

any language definable as an NFA is definable as a DFA

ε-transitions do not add expressiveness to NFAs

we showed a simple algorithm to remove epsilons

Regular Languages and Regular Expressions

Regular Language

Any language recognized by an FSM is a regular language

Examples:

- Single-line comments beginning with //
- Integer literals
- {ε, ab, abab, ababab, ababababab, ... }
- C/C++ identifiers

Regular expressions

Pattern describing a language

operands: single characters, epsilon

operators: from low to high precedence

alternation “or”: a | b

catenation: a.b, ab, a^3 (which is aaa)

iteration: a* (0 or more a’s) aka Kleene star

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 scanner

Regular expressions are inputs to a scanner generator

Regexp, cont’d

Conventions:

a+ is aa*

letter is a|b|c|d|...|y|z|A|B|...|Z

digit is 0|1|2|...|9

not(x) all characters except x

. is any character

parentheses for grouping, e.g., (ab)*

ε, *ab, abab, abababab*

Regexp, example

Hex strings

start with 0x or 0X

followed by one or more hexadecimal digits

optionally end with l or L

0(x|X)hexdigit+(L|l|ε)

where hexdigit = digit|a|b|c|d|e|f|A|...|F

Regexp, example

Single-line comments in Java/C/C++

// this is a comment

//(not(‘\n’))*(’\n’|epsilon)

Regexp, example

C/C++ identifiers: sequence of letters/digits/ underscores; cannot begin with a digit; cannot end with an underscore

Example: a, _bbb7, cs_536

Regular expression

letter | (letter|_)(letter|digit|_)*(letter|digit)

Recap

Regular Languages

Languages recognized/defined by FSMs

Regular Expressions

Single-pattern representations of regular languages

Used for defining tokens in a scanner generator

Creating a Scanner