

Announcements

Contents

1. NFA: **math notations**
2. DFAs and NFAs are equivalent (2 lemmas).
3. Build an equivalent DFA from an NFA.
4. Epsilon-transition and eliminating epsilon-transition.
5. Regular expression.

- H1 posted. Due Next Tuesday

Nondeterministic Finite Automata

CS 536

Previous Lecture

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

Scanner and parser: master-slave 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

For DFA, we have a string $x_1x_2..x_n$ in $L(M)$ if:

$$\delta(...\delta(\delta(q, x_1), x_2)..., x_n) \in F$$

For NFA, we just extend the single state to a set of states:

$$|\delta(...\delta(\delta(q', x_1), x_2)..., x_n) \cap F| > 0$$

q' and the output of each δ is a set of states.

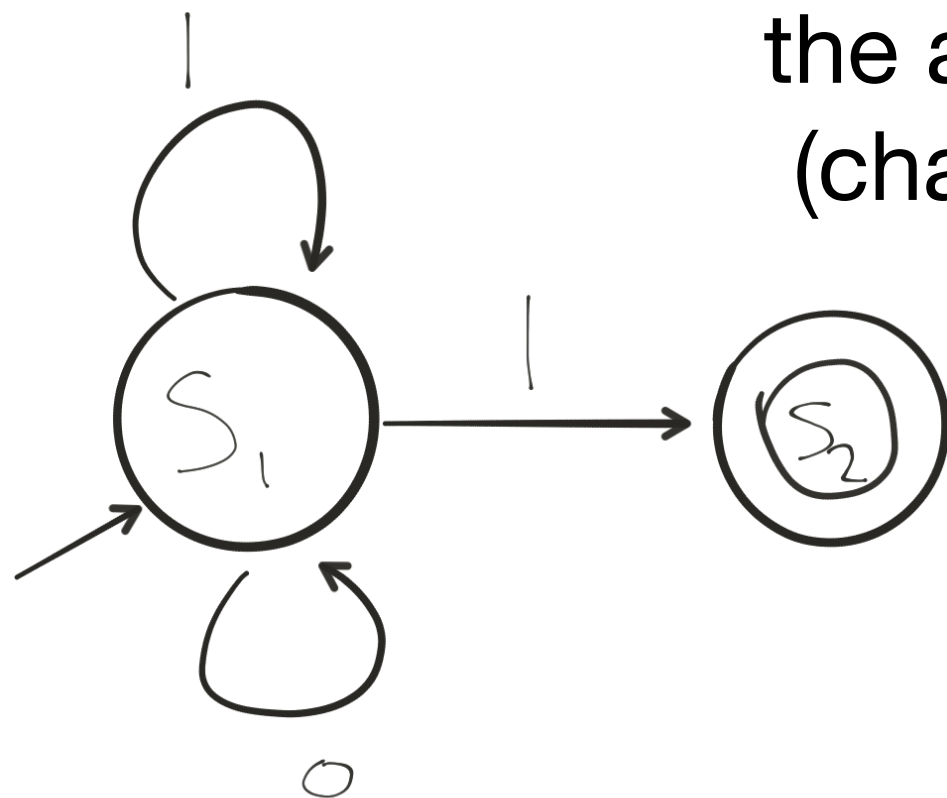
$$M \equiv (Q, \Sigma, \delta, q, F)$$

finite set of states

the alphabet
(characters)

final states
 $F \subseteq Q$

start state
 $q \in Q$



δ is a relationship where:

(An element of Q) \times (An element of Σ) \rightarrow (An element of the power set of Q)

transition function

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

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

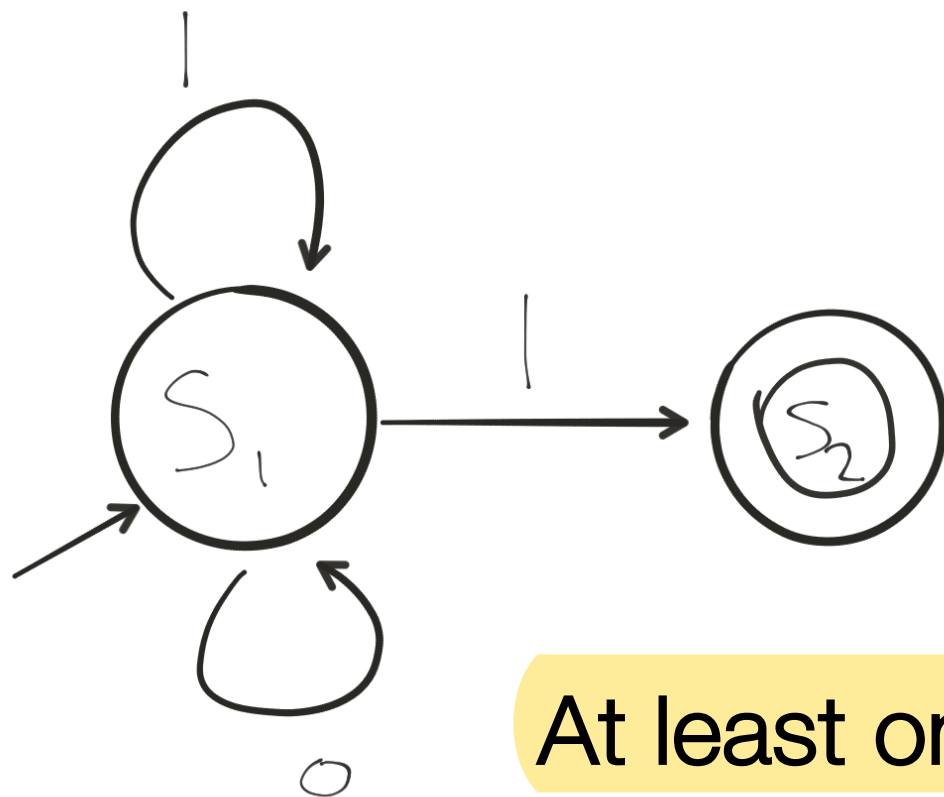
For DFA, $\delta : Q \times \Sigma \rightarrow Q$. Each entry in the transition table is a single state (or stuck);

For NFA, since multiple edge with the same label is allowed, $\delta : Q \times \Sigma \rightarrow \{a \text{ set of states from } \Sigma\}$. Each entry in the entry table is a set (or stuck).

NFA

Conceptually, you can think of the machine making different choices by choosing different paths in a **parallel** manner.

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 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 1 is trivial as each DFA can be considered as a NFA.

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?

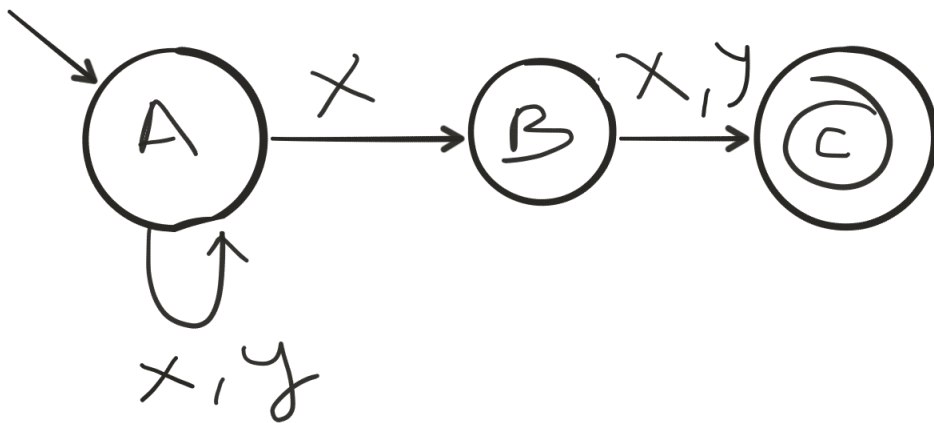
The path in a NFA can be infinitely long, thus it is infeasible to build a DFA that completely follows all path that the NFA takes.
However, the total number of **set of states** the NFA can be in is finite!

In our previous representation of DFAs, each circle in the FSM represents a single state. However, for NFA, if we think of the FSM as making different choices **in parallel**, the NFA can be in different states at the same time!

For example, in the following NFA, the machine would be in $\{S1, S2\}$ after reading the first 'a'.

```
S -- a --> S1
|
+- a --> S2
```

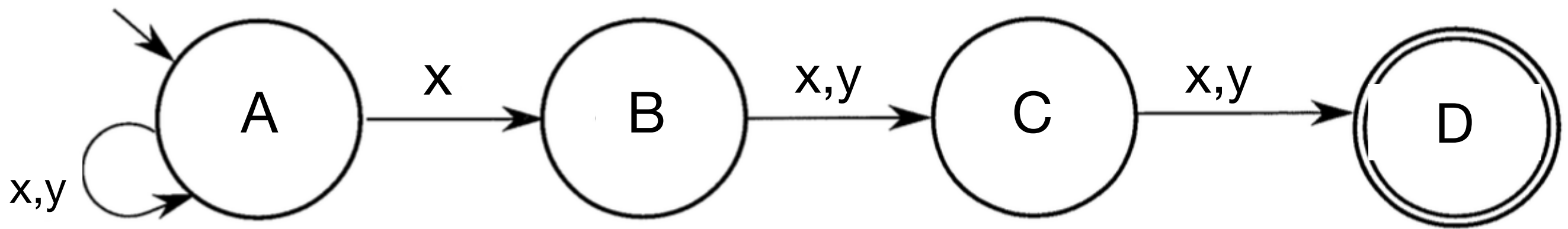

Why $2^{|Q|}$ states?



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

So we are guaranteed that the DFA is of finite size.

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}



You can consider succ as δ .

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

$$\text{succ}(A,x) = \{A,B\}$$

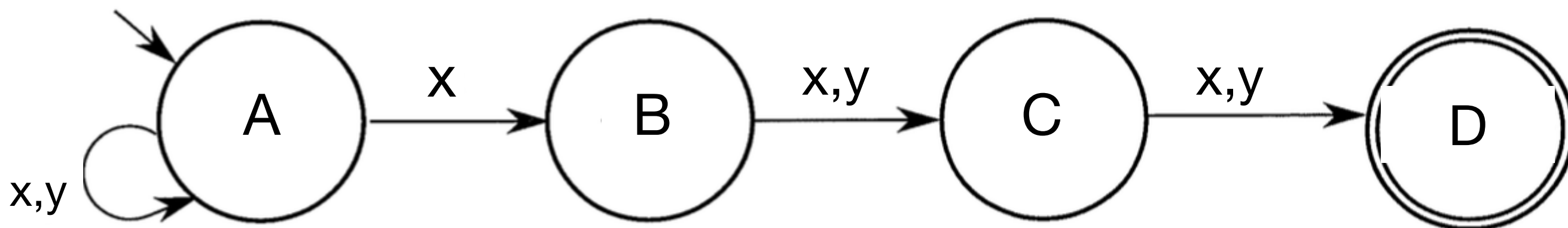
$$\text{succ}(A,y) = \{A\}$$

$$\text{succ}(B,x) = \{C\}$$

$$\text{succ}(B,y) = \{C\}$$

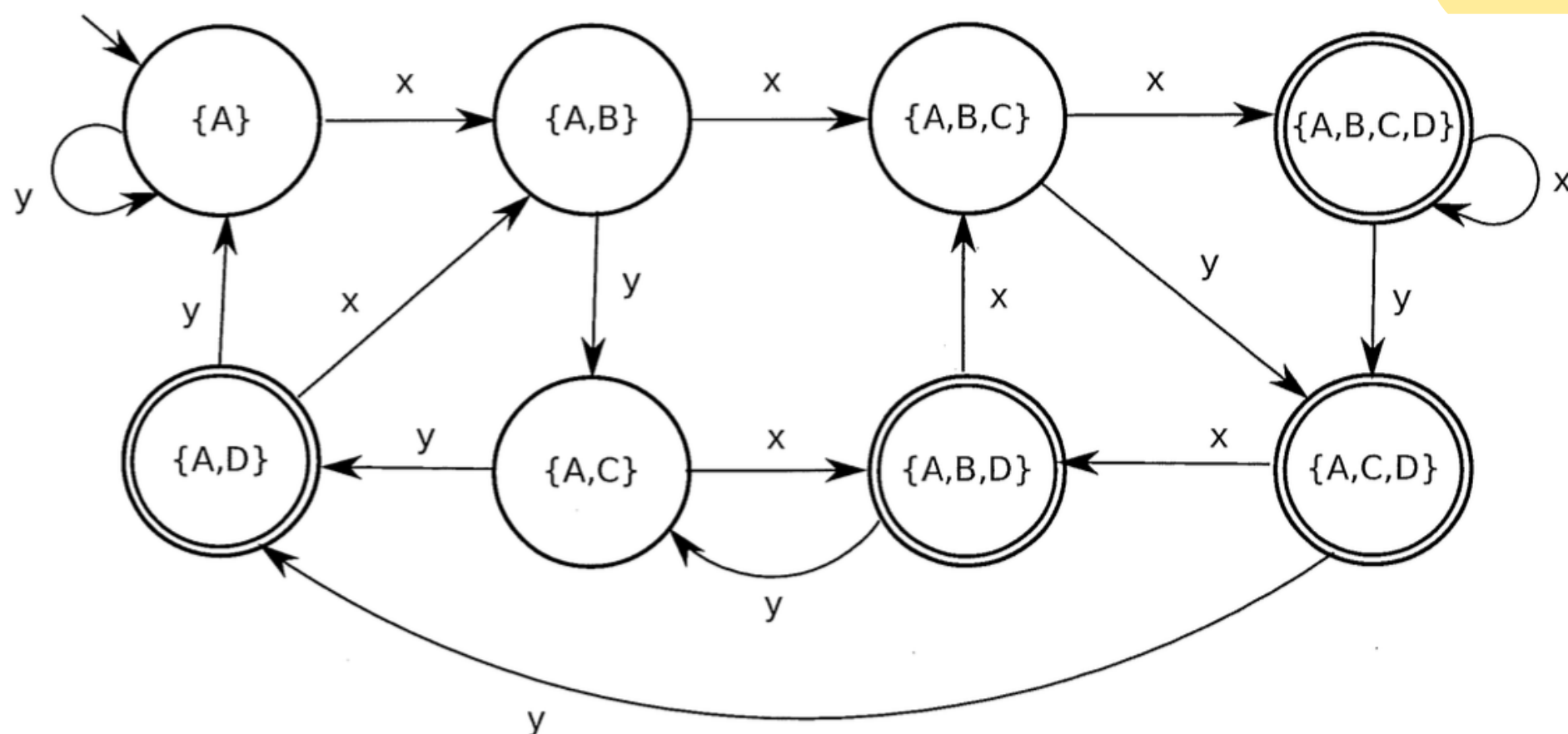
$$\text{succ}(C,x) = \{D\}$$

$$\text{succ}(C,y) = \{D\}$$



Build new DFA M' where $Q' = 2^Q$

Some of the elements in the powerset might not be present in M' .

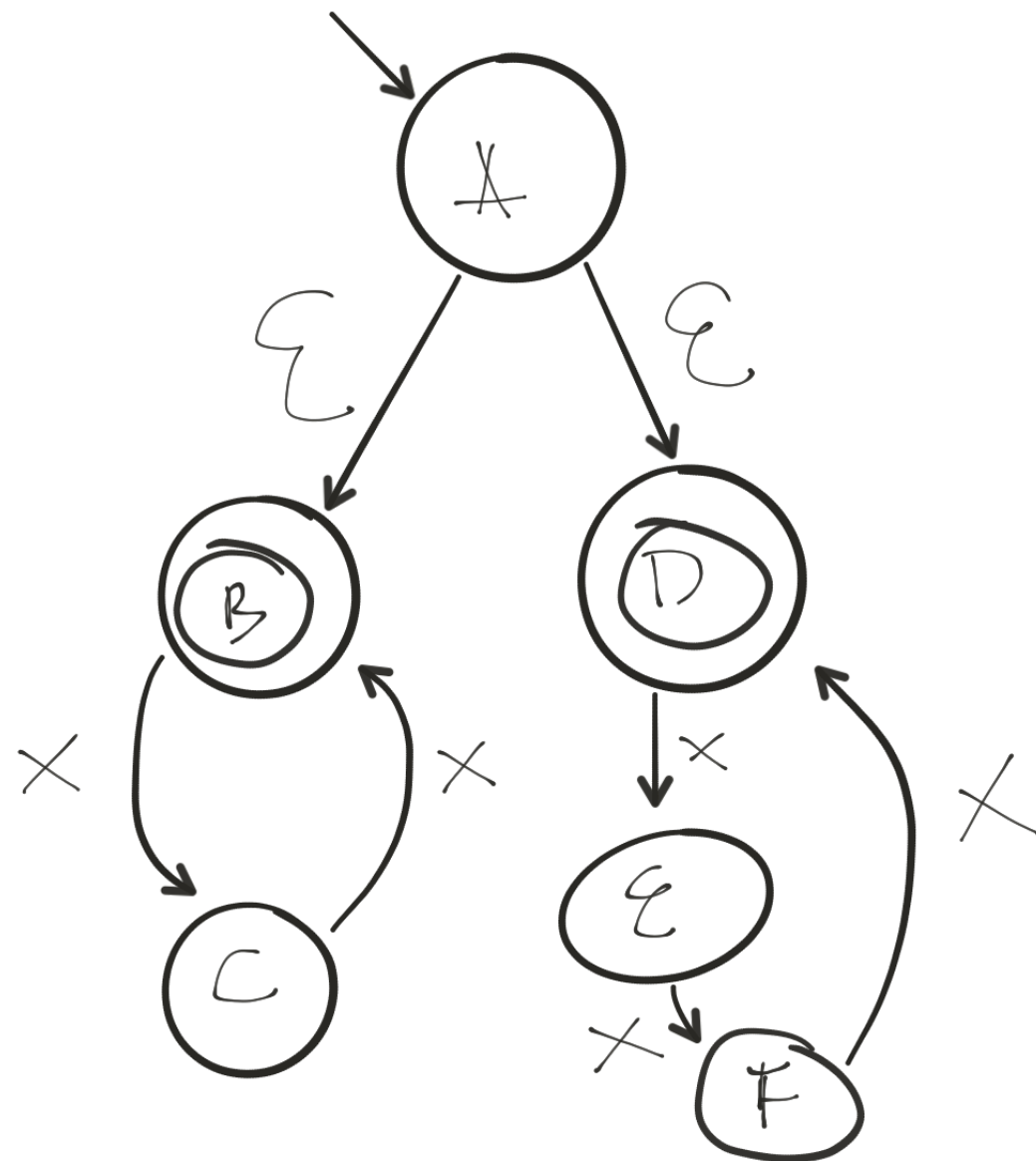


$\text{succ}(A, x) = \{A, B\}$
 $\text{succ}(A, y) = \{A\}$
 $\text{succ}(B, x) = \{C\}$
 $\text{succ}(B, y) = \{C\}$
 $\text{succ}(C, x) = \{D\}$
 $\text{succ}(C, y) = \{D\}$

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



The "OR" operator.

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:

$\text{eclose}(s)$ = set of all states reachable from s in
zero or more epsilon transitions

including the state
itself.

M' components

s is an accepting state of M' iff $\text{eclose}(s)$ contains an accepting state

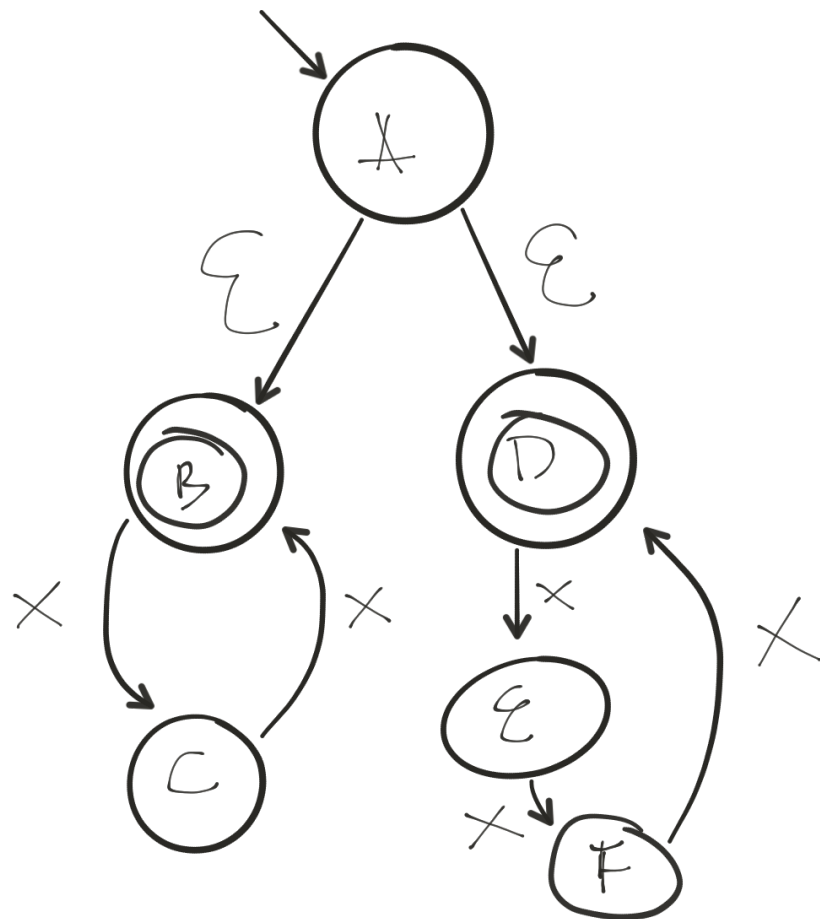
$s \xrightarrow{c} t$ is a transition in M' iff
 $q \xrightarrow{c} t$ for some q in $\text{eclose}(s)$

Eliminating ε -transitions

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

Definition: Epsilon Closure

$\text{eclose}(s)$ = 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}

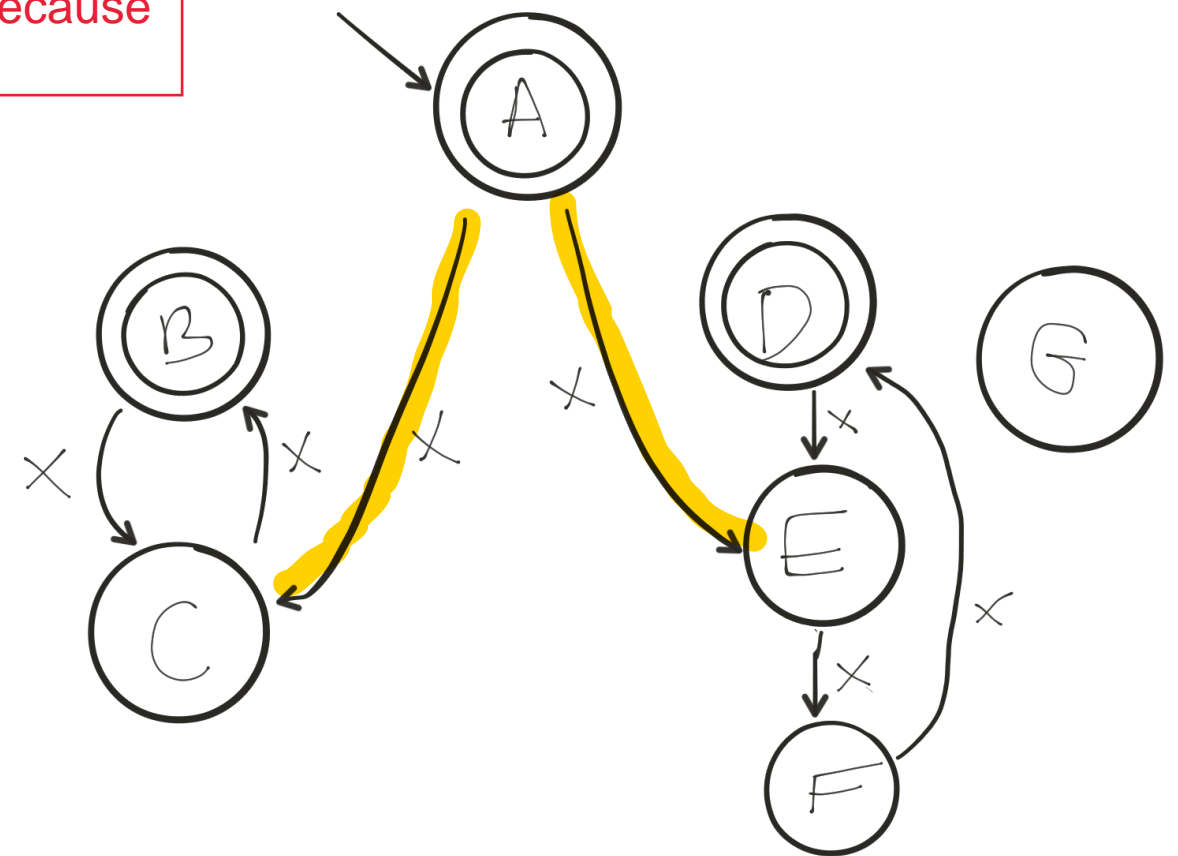
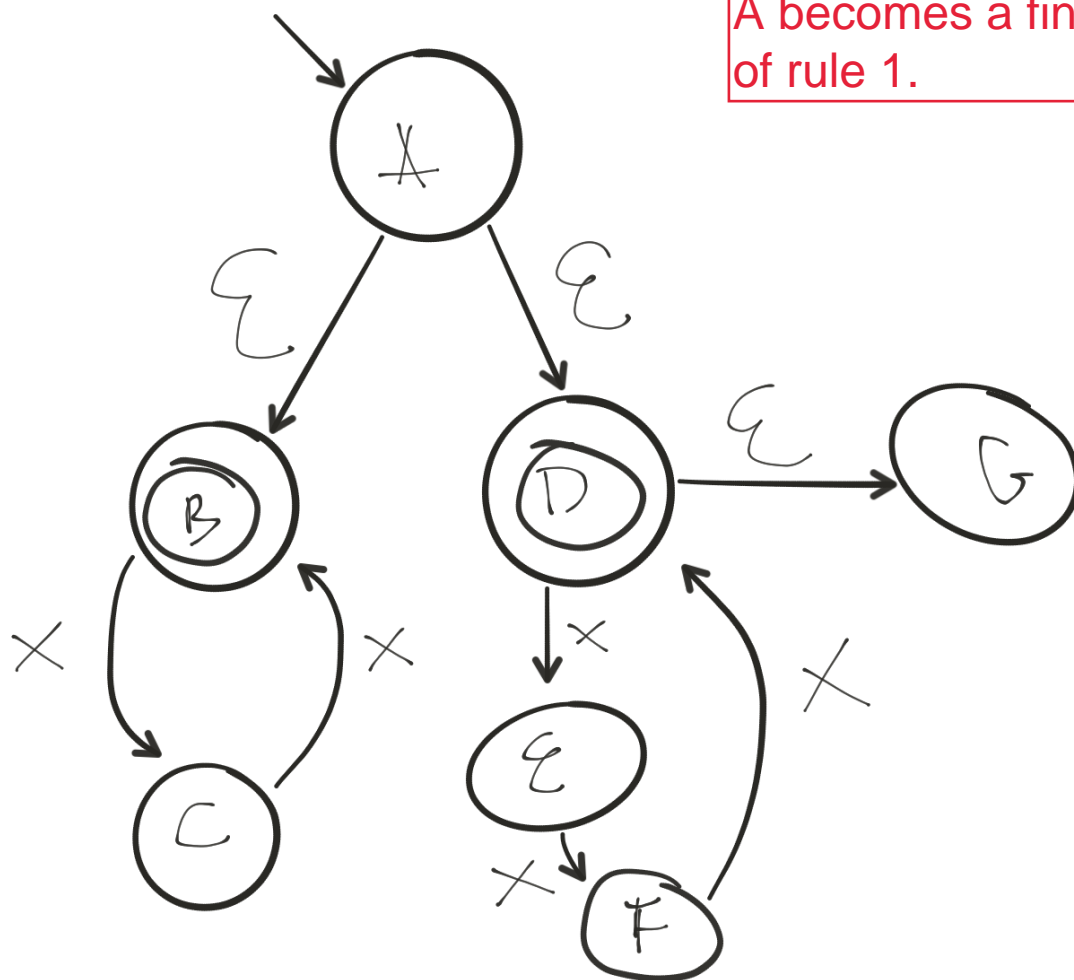
Def: $\text{eclose}(s)$ = set of all states reachable from s in zero or more epsilon transitions

Rule 1 s is an accepting state of M' iff $\text{eclose}(s)$ contains an accepting state

Rule 2 $s \xrightarrow{c} t$ is a transition in M' iff
 $q \xrightarrow{c} t$ for some q in $\text{eclose}(s)$

G is connected in the old machine but not in the new machine. The reason is that in the new epsilon-free machine, we only have transitions when reading **real** characters, not epsilons.

The two highlighted transitions exist due to rule 2;
 A becomes a final-state because of rule 1.



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

The language can be described by a DFA or NFA.

Any language recognized by an FSM is a regular language

Examples:

- Single-line comments beginning with //
- Integer literals
- $\{\epsilon, ab, abab, ababab, abababab, \dots\}$
- C/C++ identifiers

Regular expressions

Pattern describing a language

operands: single characters, epsilon

operators: from low to high precedence

alternation “or”: $a \mid b$

The dot is
usually omitted
by convention.

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

Regex, 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*, *ababab*

Regexp, example

Hex strings

start with 0x or 0X

followed by one or more hexadecimal digits

optionally end with l or L

$0(x|X)\text{hexdigit}^+(L||\epsilon)$

where $\text{hexdigit} = \text{digit}|a|b|c|d|e|f|A|\dots|F$

Regex, example

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

```
// this is a comment
```

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

not('\n') to ensure that we
only match a **single-line**.

Regex, 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)

Edge case: single-char
identifier.

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

