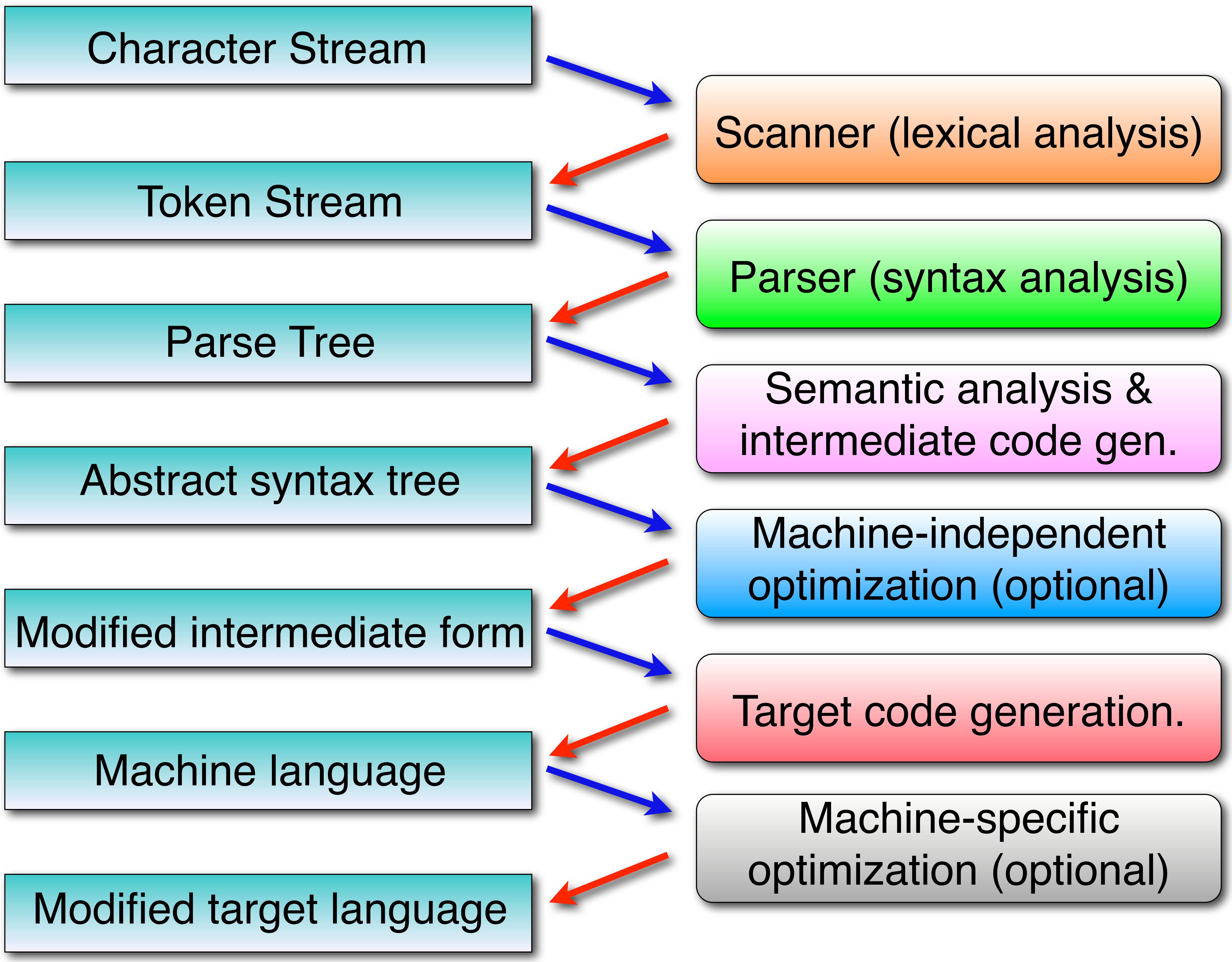
Lexical Analysis

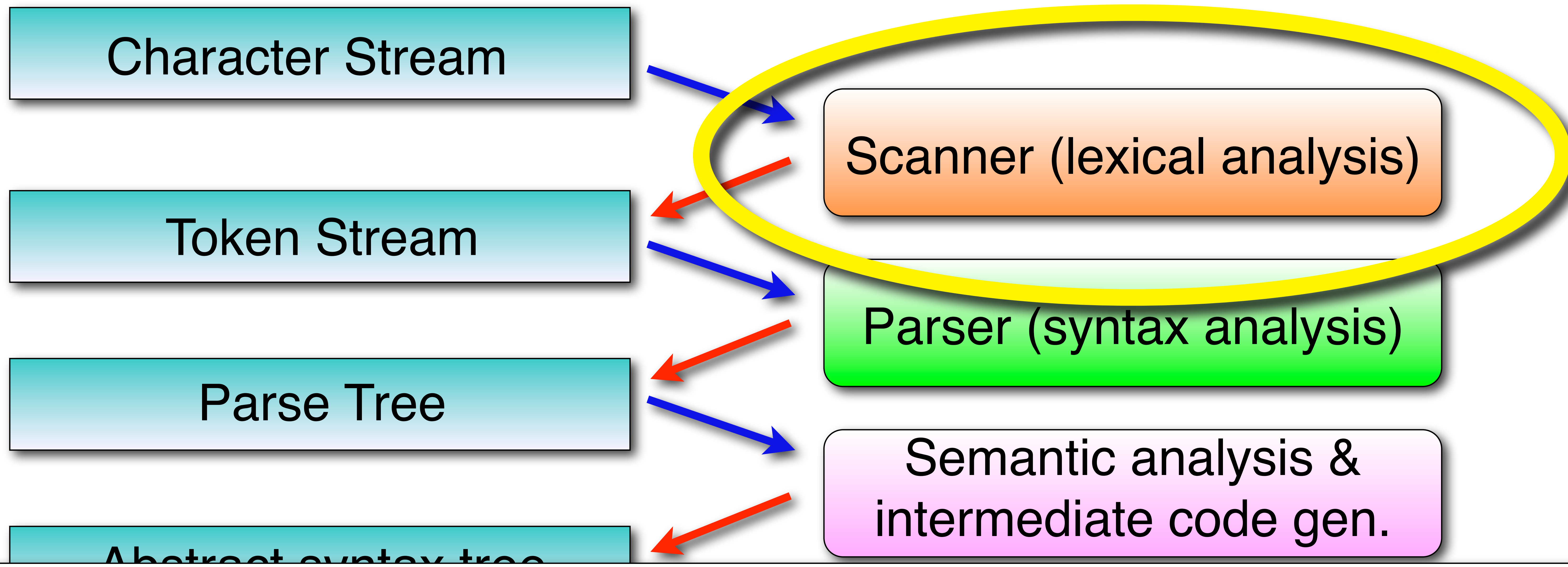
COMP 524: Programming Language Concepts
Björn B. Brandenburg

The University of North Carolina at Chapel Hill

The Big Picture



The Big Picture

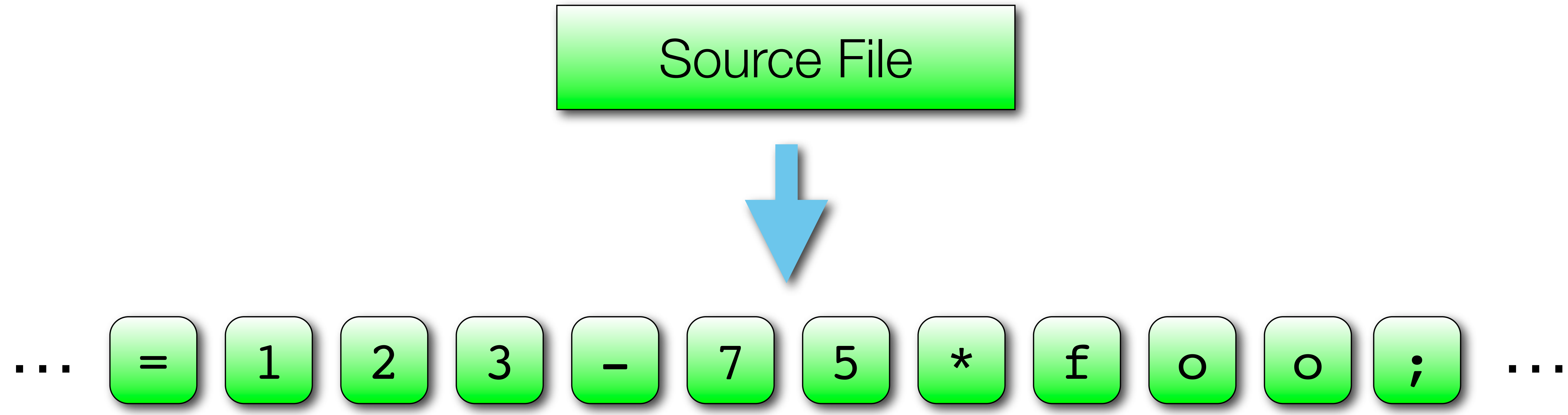


Lexical analysis:
grouping consecutive characters that “belong together.”

Turn the stream of individual characters into a **stream of tokens** that have individual meaning.

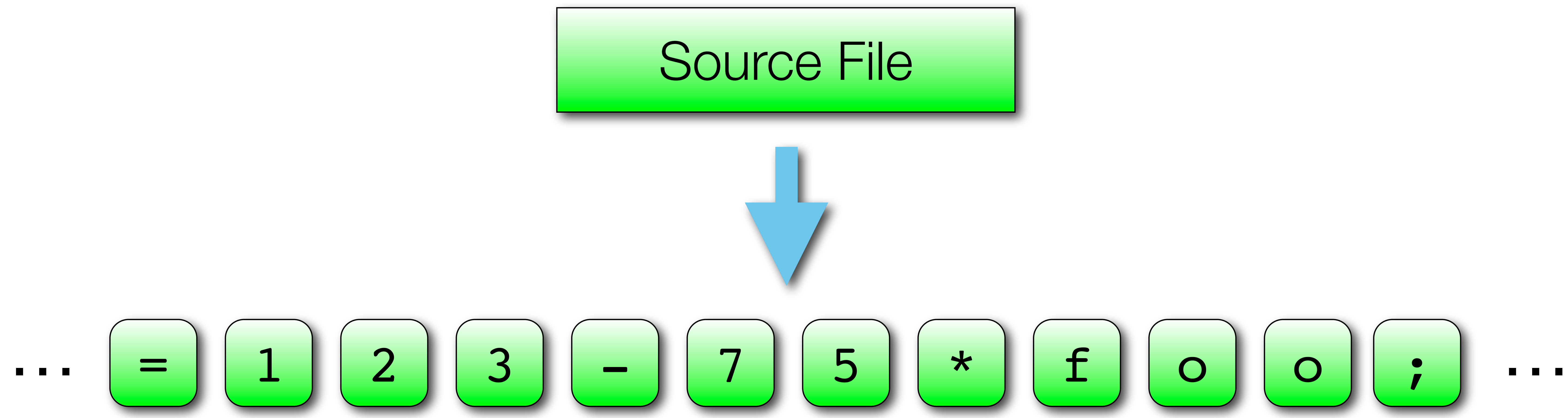
Source Program

The compiler reads the program from a file.
➡ Input as a character stream.



Compilation requires analysis of program structure.
➡ Identify subroutines, classes, methods, etc.
➡ Thus, first step is to find **units of meaning**.

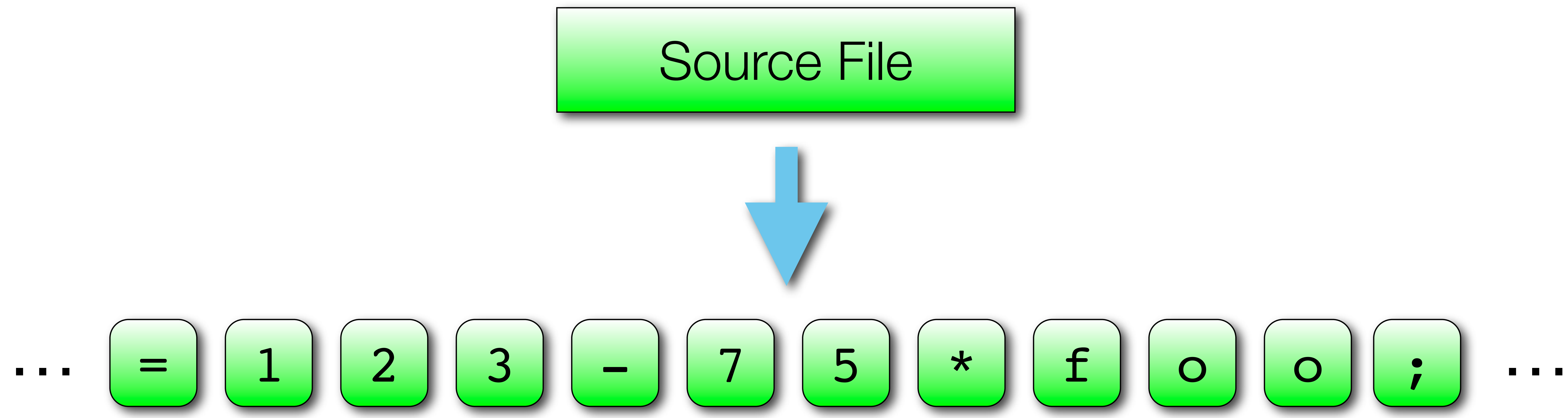
Tokens



Not every character has an individual meaning.

- ➡ In Java, a '+' can have two interpretations:
  - ‣ A single '+' means **addition**.
  - ‣ A '+' '+' sequence means **increment**.
- ➡ A sequence of characters that has an **atomic** meaning is called a **token**.
- ➡ Compiler must identify all input tokens.

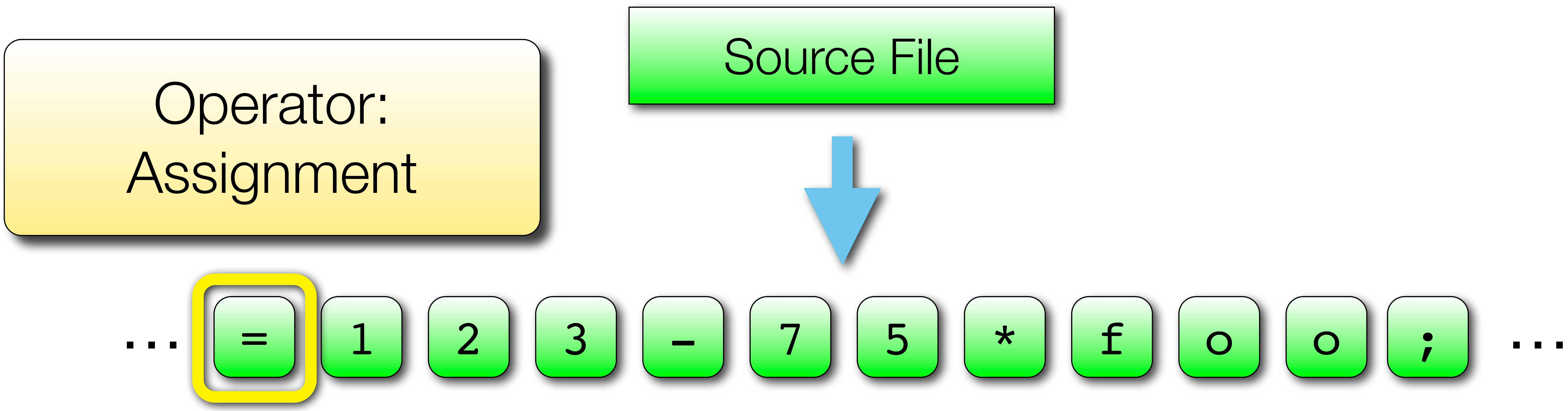
Tokens



Human Analogy:
To understand the meaning of an English sentence, we do not look at individual characters. Rather, we look at individual **words**.

Human word = Program token

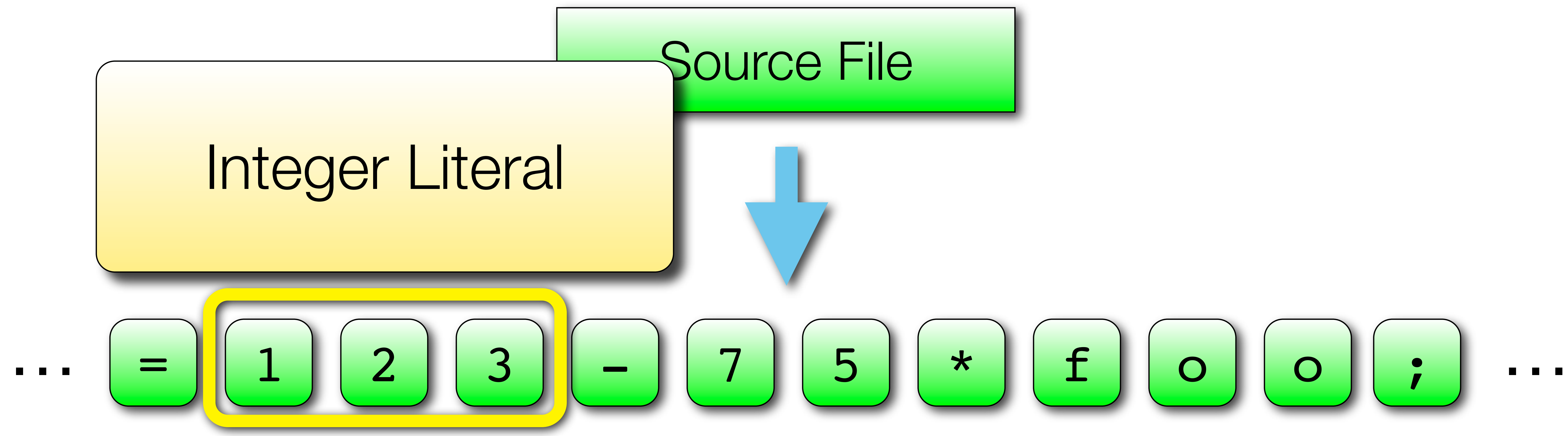
Tokens



Not every character has an individual meaning.

- In Java, a ‘+’ can have two interpretations:
  - A single ‘+’ means **addition**.
  - A ‘+’ ‘+’ followed by another means **increment**.
- A sequence of characters that has an **atomic** meaning is called a **token**.
- Compiler must identify all input tokens.

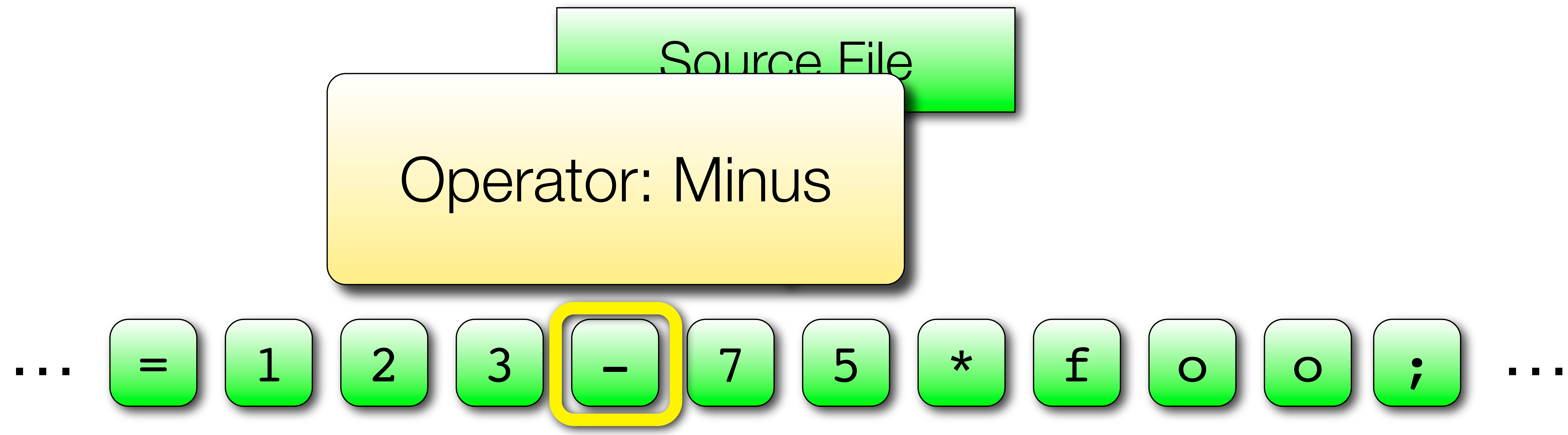
Tokens



Not every character has an individual meaning.

- ➡ In Java, a ‘+’ can have two interpretations:
  - ‣ A single ‘+’ means **addition**.
  - ‣ A ‘+’ ‘+’ followed by another means **increment**.
- ➡ A sequence of characters that has an **atomic** meaning is called a **token**.
- ➡ Compiler must identify all input tokens.

Tokens



Not every character has an individual meaning.

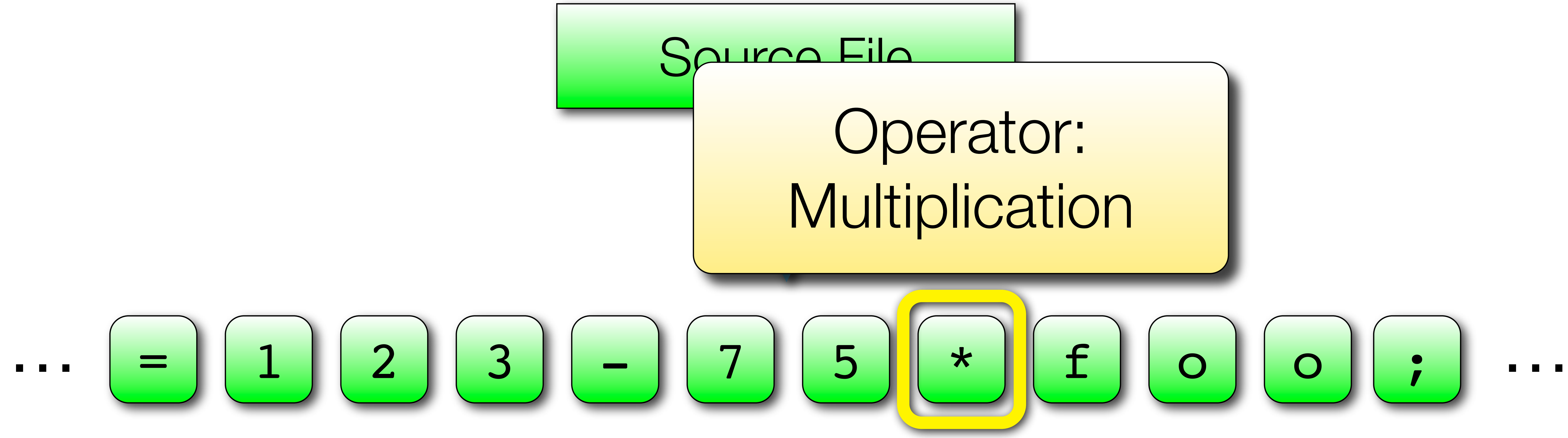
- ➡ In Java, a ‘+’ can have two interpretations:
  - A single ‘+’ means **addition**.
  - A ‘+’ ‘+’ followed by another means **increment**.
- ➡ A sequence of characters that has an **atomic** meaning is called a **token**.
- ➡ Compiler must identify all input tokens.

Tokens

Not every character has an individual meaning.

- ➡ In Java, a '+' can have two interpretations:
  - A single '+' means **addition**.
  - A '+' '+' followed by another means **increment**.
- ➡ A sequence of characters that has an **atomic** meaning is called a **token**.
- ➡ Compiler must identify all input tokens.

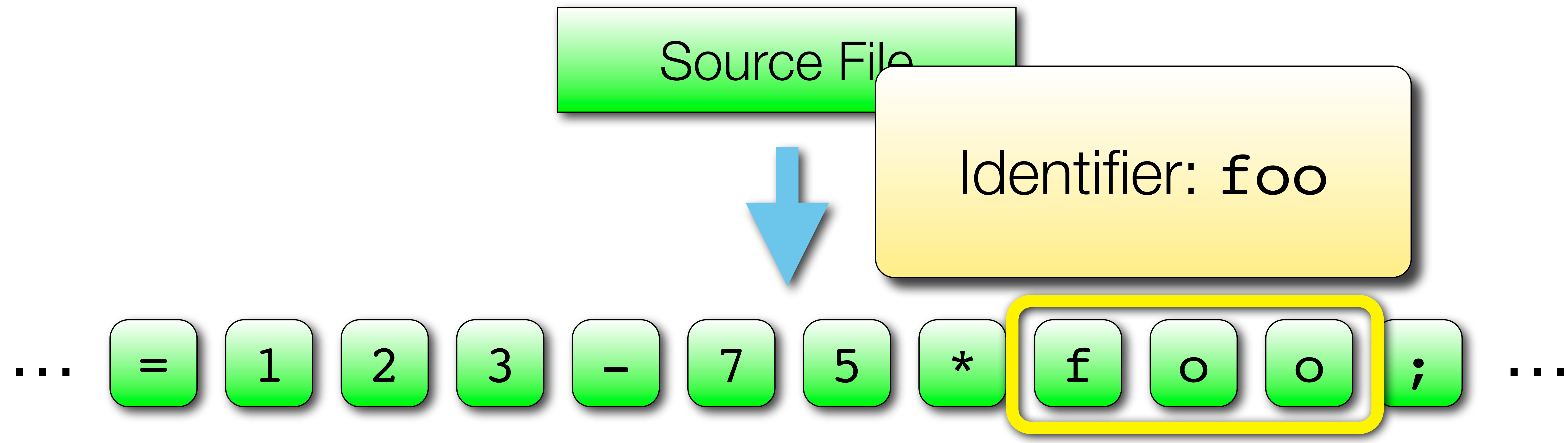
Tokens



Not every character has an individual meaning.

- ➡ In Java, a ‘+’ can have two interpretations:
  - A single ‘+’ means **addition**.
  - A ‘+’ ‘+’ followed by another means **increment**.
- ➡ A sequence of characters that has an **atomic** meaning is called a **token**.
- ➡ Compiler must identify all input tokens.

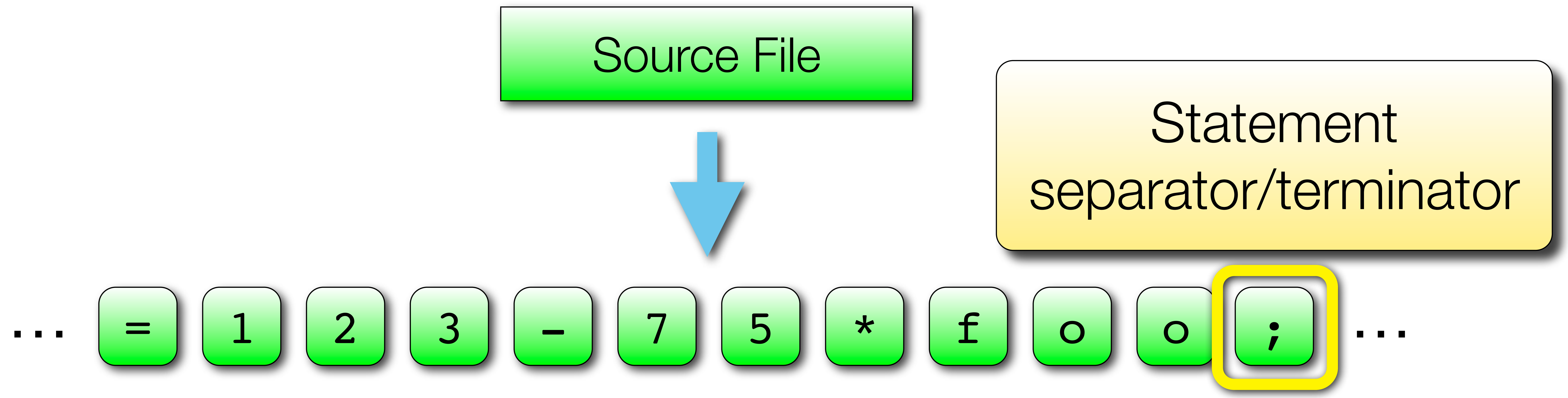
Tokens



Not every character has an individual meaning.

- ➡ In Java, a '+' can have two interpretations:
  - A single '+' means **addition**.
  - A '+' '+' followed by another means **increment**.
- ➡ A sequence of characters that has an **atomic** meaning is called a **token**.
- ➡ Compiler must identify all input tokens.

Tokens



Not every character has an individual meaning.

- In Java, a ‘+’ can have two interpretations:
  - A single ‘+’ means **addition**.
  - A ‘+’ ‘+’ followed by another means **increment**.
- A sequence of characters that has an **atomic** meaning is called a **token**.
- Compiler must identify all input tokens.

Lexical vs. Syntactical Analysis

*Why have a **separate** lexical analysis phase?*

- In theory, **token discovery** (lexical analysis) **could be done** as part of the **structure discovery** (syntactical analysis, parsing).
- However, this is **unpractical**.
- It is much easier (and much more efficient) to **express the syntax rules in terms of tokens**.
- Thus, lexical analysis is made a separate step because **it greatly simplifies** the subsequently performed syntactical analysis.

Example: Java Language Specification

Lexical Structure

The following 37 tokens are the *operators*, formed from ASCII characters:

Operator: one of

```
=    >    <    !    ~    ?    :
==   <=   >=   !=   &&   ||   ++   --
+    -    *    /    &    |    ^    %    <<    >>    >>>
+=   -=   *=   /=   &=   |=   ^=   %=   <<=   >>=   >>>=
```

Syntactical Structure

UnaryExpression:
  PreIncrementExpression
  PreDecrementExpression
  + *UnaryExpression*
  - *UnaryExpression*
  UnaryExpressionNotPlusMinus

PreIncrementExpression:
  ++ *UnaryExpression*

UnaryExpressionNotPlusMinus:
  PostfixExpression
  ~ *UnaryExpression*
  ! *UnaryExpression*
  CastExpression

Example: Jav

Token Specification:
These strings mean something, but knowledge of the exact meaning is not required to identify them.

Lexical Structure

The following 37 tokens are the *operators*, formed from ASCII characters:

Operator: one of

```
=    >    <    !    ~    ?    :
==   <=   >=   !=   &&   ||   ++   --
+    -    *    /    &    |    ^    %    <<    >>    >>>
+=   -=   *=   /=   &=   |=   ^=   %=   <<=   >>=   >>>=
```

Syntactical Structure

UnaryExpression:
- *PreIncrementExpression*
- *PreDecrementExpression*
- \+ *UnaryExpression*
- \- *UnaryExpression*
- *UnaryExpressionNotPlusMinus*

PreIncrementExpression:
- ++ *UnaryExpression*

UnaryExpressionNotPlusMinus:
- *PostfixExpression*
- ~ *UnaryExpression*
- ! *UnaryExpression*
- *CastExpression*

Example: Jav

Token Specification: These strings mean something, but knowledge of the exact meaning is not required to identify them.

Lexical Structure

The following 37 tokens are the *operators*, formed from ASCII characters:

Operator: one of

```
=    >    <    !    ~    ?    :
==   <=   >=   !=   &&   ||   ++   --
+    -    *    /    &    |    ^    %    <<    >>    >>>
+=   -=   *=   /=   &=   |=   ^=   %=   <<=   >>=   >>>=
```

Syntactical Structure

Meaning is given by where they can occur in the program (**grammar**) and and language semantics.

UnaryExpression:
  PreIncrementExpression
  PreDecrementExpression
  \+ *UnaryExpression*
  \- *UnaryExpression*
  UnaryExpressionNotPlusMinus

! *UnaryExpression*
CastExpression

PreIncrementExpression:
  ++ *UnaryExpression*

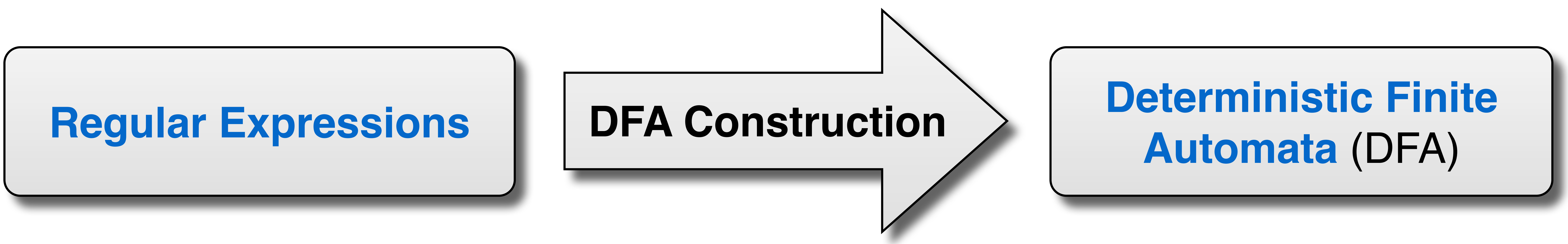
Lexical Analysis

The need to identify tokens raises two questions.

- ➡ How can we **specify the tokens** of a language?
- ➡ How can we **recognize tokens** in a character stream?

Token Specification

Token Recognition



Language Design and Specification

(several steps)

Language Implementation

Grammars and Languages

A regular expression is a kind of grammar.

- ➡ A grammar describes the **structure of strings**.
- ➡ A string that "matches" a grammar ***G***'s structure is said to be in the **language** ***L***(***G***) (which is a set).

A grammar is a set of productions:

- ➡ Rules to obtain (produce) a string that is in ***L***(***G***) via **repeated substitutions**.
- ➡ There are many grammar **classes** (see COMP 455).
- ➡ Two are commonly used to describe programming languages: **regular grammars** for tokens and **context-free grammars** for syntax.

Grammar 101

digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

non_zero_digit → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

natural_number → *non_zero_digit digit**

non_neg_number → (0 | *natural_number*) ((. *digit* non_zero_digit*) | ε)

Grammar 101: Productions

digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

non_zero_digit → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

natural_number → *non_zero_digit digit**

non … | ε)

"*A* → B" is called a **production**.

Grammar 101: Non-Terminals

digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

non_zero_digit → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

natural_number → *non_zero_digit digit**

non … | ε)

The “name” on the left is called a **non-terminal symbol**.

Grammar 101: Terminals

digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

non_zero_digit → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

natural_number → *non_zero_digit digit**

The symbols on the right are either **terminal** or **non-terminal** symbols. A terminal symbol is just a character.

Grammar 101: Definition

digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

non_zero_digit → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

na … *t**

non_neg_number … *n_zero_digit*) | ε)

“→” means “**is a**”
or “**replace with**”

Grammar 101: Choice

digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

non_zero_digit → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

na… …*t**

non_neg_number … *n_zero_digit*) | ε)

"|" denotes **or**

Grammar 101: Example

digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

non_zero_digit → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

natural_number → *non_zero_digit digit**

non … ε)

Thus, the first production means:

A *digit* is a “0” or ‘1’ or ‘2’ or … or ’9’.

Grammar 101: Optional Repetition

"*" denotes **zero or more** **of a symbol**.

digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

non_zero_digit → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

natural_number → *non_zero_digit digit**

non_neg_number → (0 | *natural_number*) ((. *digit* non_zero_digit*) | ε)

Grammar 101: Sequence

Two symbols next to each other means "**followed by**."

non_zero_digit → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

natural_number → *non_zero_digit digit**

non_neg_number → (0 | *natural_number*) ((. *digit* non_zero_digit*) | ε)

Grammar 101: Example

Thus, this means:

A *natural number* is a *non-zero digit* followed by zero or more *digits*.

non_zero_digit → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

natural_number → *non_zero_digit digit**

non_neg_number → (0 | *natural_number*) ((. *digit* non_zero_digit*) | ε)

Grammar 101: Epsilon

digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

"**ε**" is special terminal that means **empty**.
It corresponds to the empty string.

natural_number → *non_zero_digit digit**

non_neg_number → (0 | *natural_number*) ((. *digit* non_zero_digit*) | ε)

Grammar 101: Example

So, what does this mean?

digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

non_zero_digit → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

natural_number → *non_zero_digit digit**

non_neg_number → (0 | *natural_number*) ((. *digit** *non_zero_digit*) | ε)

Grammar 101: Example

A ***non-negative number*** **is a '0' or a *natural number*, followed by either nothing or a '.', followed by zero or more *digits*, followed by** (exactly one) ***digit*.**

natural_number → *non_zero_digit digit**

non_neg_number → (0 | *natural_number*) ((. *digit* non_zero_digit*) | ε)

Regular Expression Rules

Base case: a regular expression (RE) is either

- a **character** (e.g., ‘0’, ‘1’, ...), or
- the **empty string** (i.e., ‘ε’).

A compound RE is constructed by

- **concatenation**: two REs next to each other (e.g., “*non_negative_digit digit*”),
- **alternation**: two REs separated by “**|**” next to each other (e.g., “*non_negative_digit* | *digit*”),
- **optional repetition**: a RE followed by “*” (the **Kleene star**) to denote zero or more occurrences, and
- **parentheses** (in order to avoid ambiguity).

Regular Expression Rules

Base case: a regular expression (RE) is either

- a **character** (e.g., ‘0’, ‘1’, ...), or
- the **empty string** (i.e., ‘ε’).

A co

A RE is **NEVER** defined in terms of itself!
Thus, REs **cannot define recursive statements**.

- **co**
  “n
- **alternation:** two REs separated by “|” next to each other (e.g., “*non_negative_digit* | *digit*”),
- **optional repetition**: a RE followed by “*” (the **Kleene star**) to denote zero or more occurrences, and
- **parentheses** (in order to avoid ambiguity).

Example

Let's create a regular expression corresponding to the "**City, State ZIP-code**" line in mailing addresses.

E.g.: **Chapel Hill, NC 27599-3175**
Beverly Hills, CA 90210

Example

Let’s create a regular expression corresponding to the “**City, State ZIP-code**” line in mailing addresses.

E.g.: **Chapel Hill, NC 27599-3175**
Beverly Hills, CA 90210

city_line → *city* ‘, ‘ *state_abbrev* ‘ ‘ *zip_code*
city → *letter* (*letter* | ‘ ‘ *letter*)*
state_abbrev → ‘AL’ | ‘AK’ | ‘AS’ | ‘AZ’ | … | ‘WY’
zip_code → *digit digit digit digit digit* (*extra* | ε)
extra → ‘-’ *digit digit digit digit*
digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
letter → A | B | C | … | ö | …

Regular Sets and Finite Automata

If a grammar ***G*** is a regular expression, then the language ***L(G)*** is called a **regular set**.

Fundamental equivalence:

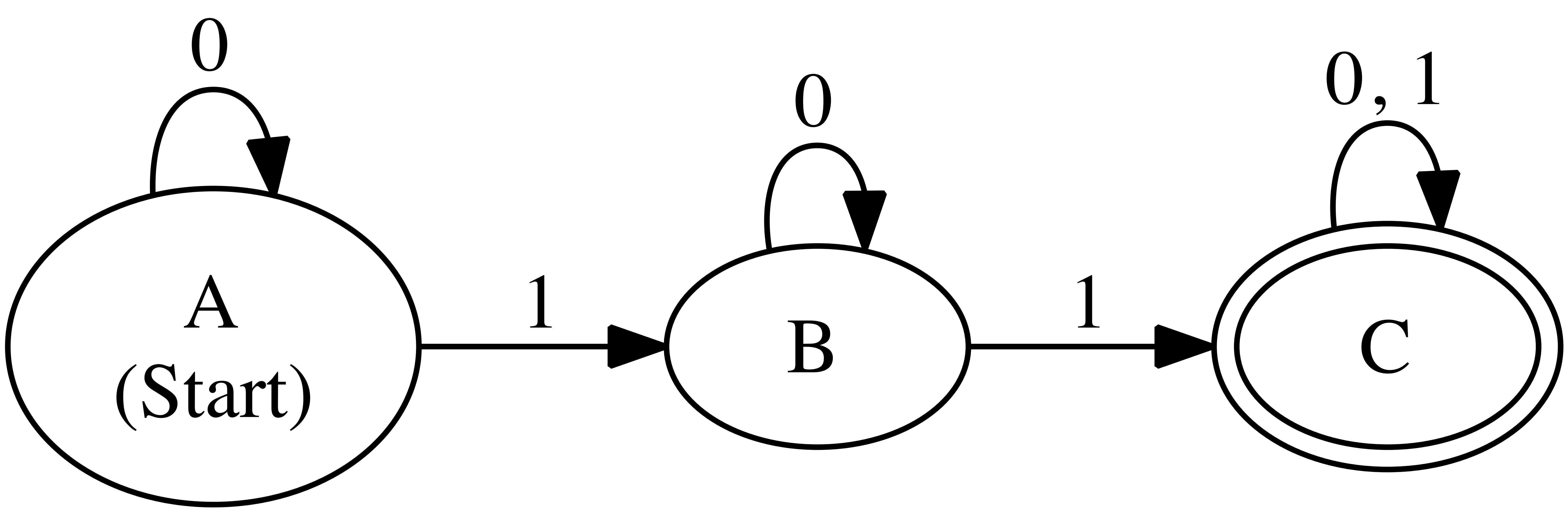
For every **regular set *L(G)***, there exists a **deterministic finite automaton** (DFA) that **accepts** a string ***S*** if and only if ***S***∈***L(G)***.

(See COMP 455 for proof.)

DFA 101

Deterministic finite automaton:

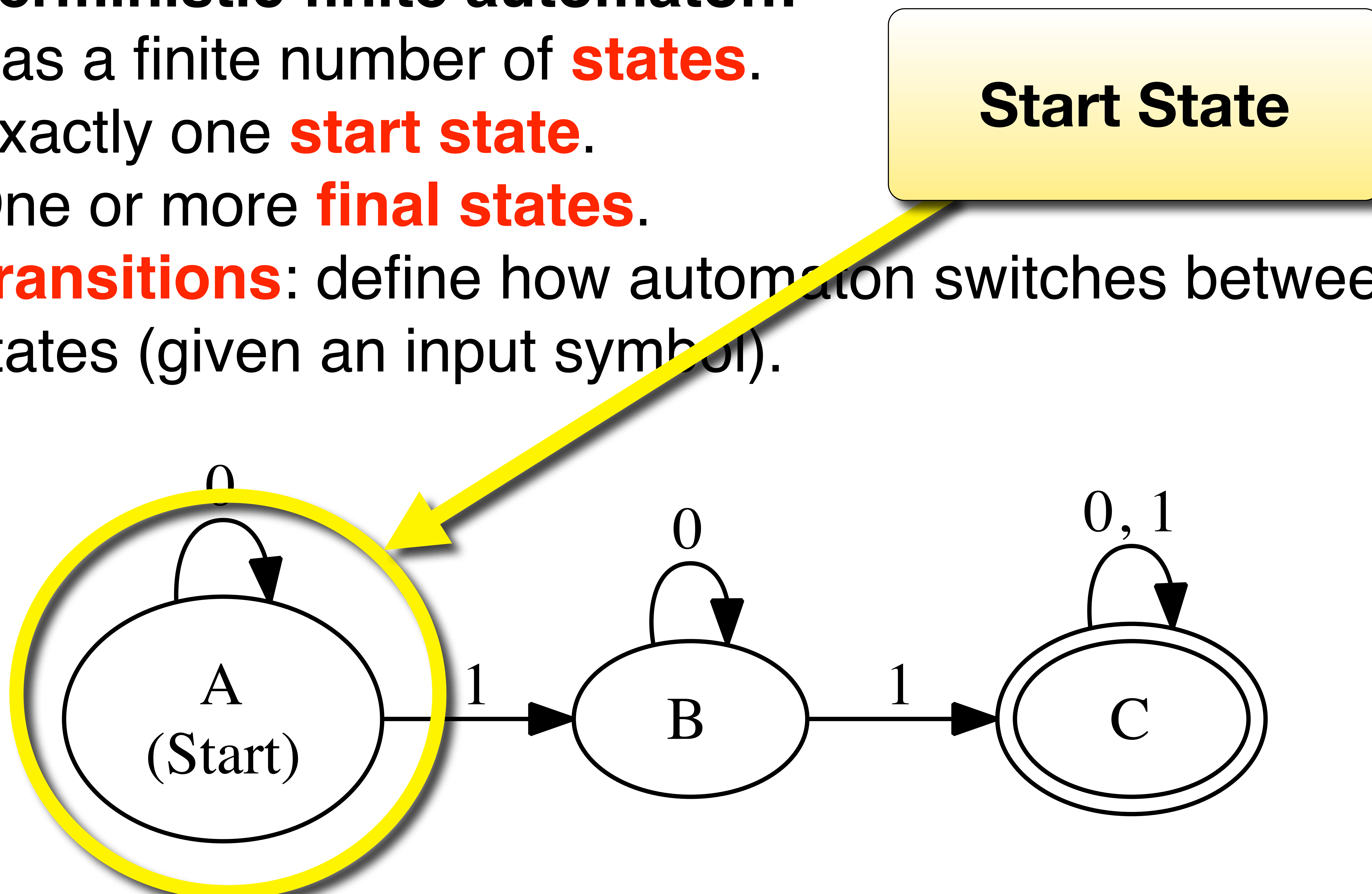
- Has a finite number of **states**.
- Exactly one **start state**.
- One or more **final states**.
- **Transitions**: define how automaton switches between states (given an input symbol).



DFA 101

Deterministic finite automaton:

- Has a finite number of **states**.
- Exactly one **start state**.
- One or more **final states**.
- **Transitions**: define how automaton switches between states (given an input symbol).

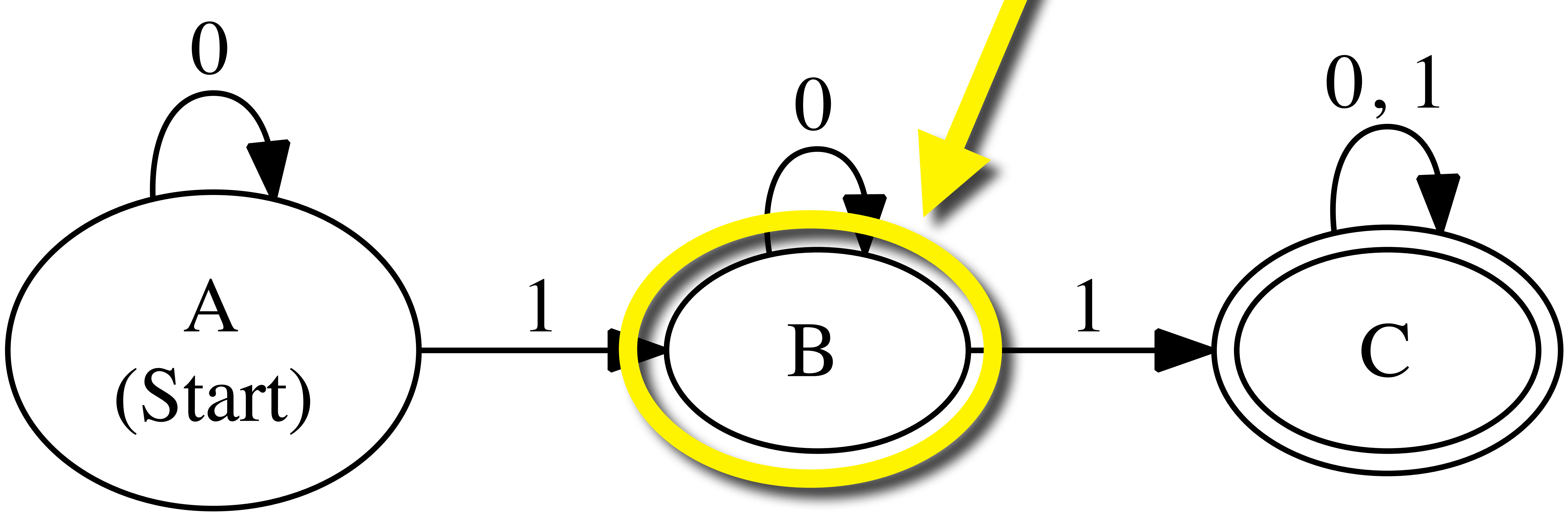


DFA 101

Deterministic finite automaton:

- ➡ Has a finite number of **states**.
- ➡ Exactly one **start state**.
- ➡ One or more **final states**.
- ➡ **Transitions**: define how automat
states (given an input symbol).

Intermediate State
(neither start nor final)

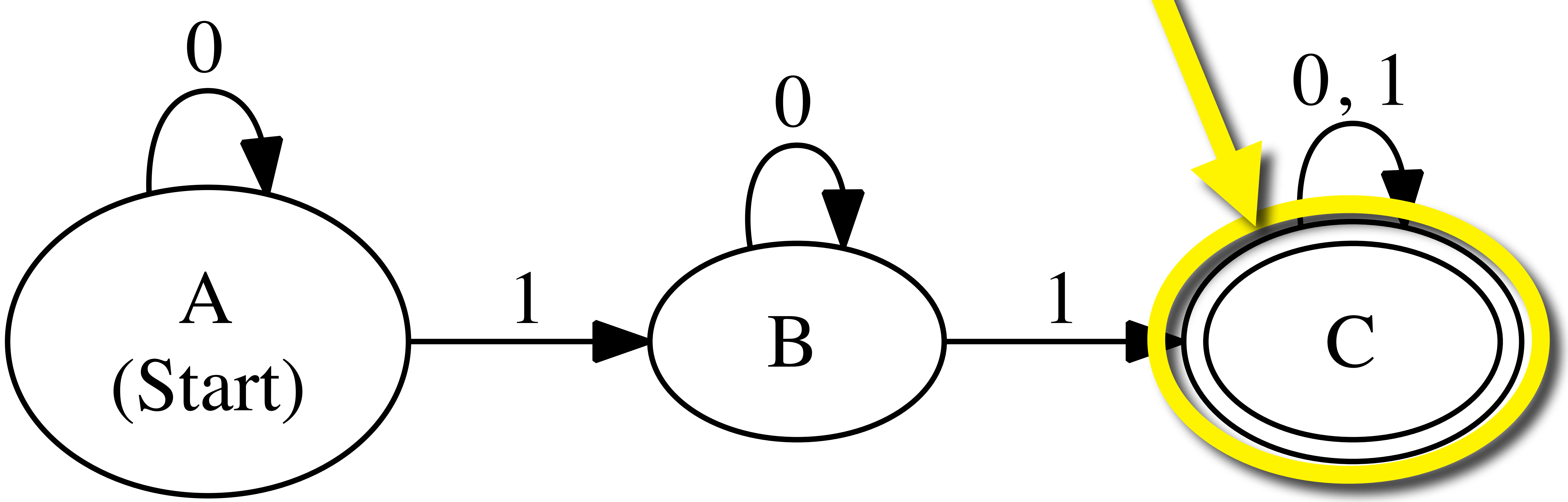


DFA 101

Deterministic finite automaton:

- ➡ Has a finite number of **states**
- ➡ Exactly one **start state**.
- ➡ One or more **final states**.
- ➡ **Transitions**: define how aut
  states (given an input symbol).

Final State
(indicated by double border)



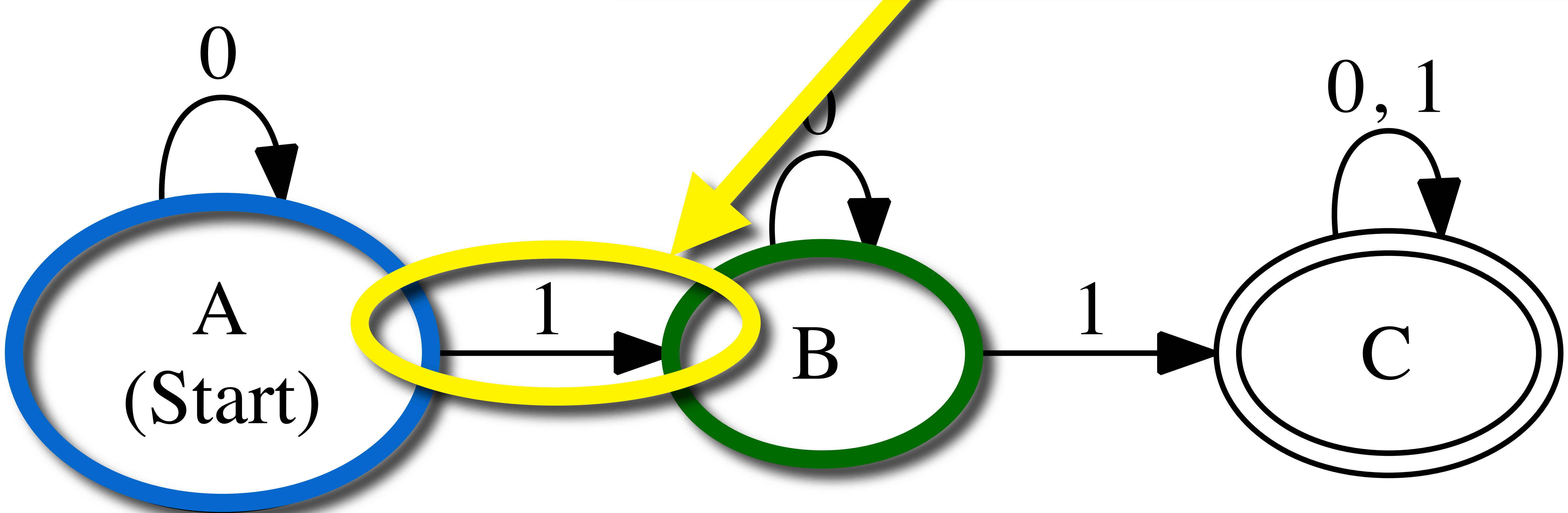
DFA 101

Deterministic finite automaton:

- ➡ Has a finite number of **states**
- ➡ Exactly one **start sta**
- ➡ One or more **final st**
- ➡ **Transitions**: define l
  states (given an inpu

Transition

Given an input of ‘**1**’, if DFA is in **state A**, then transition to **state B** (and consume the input).

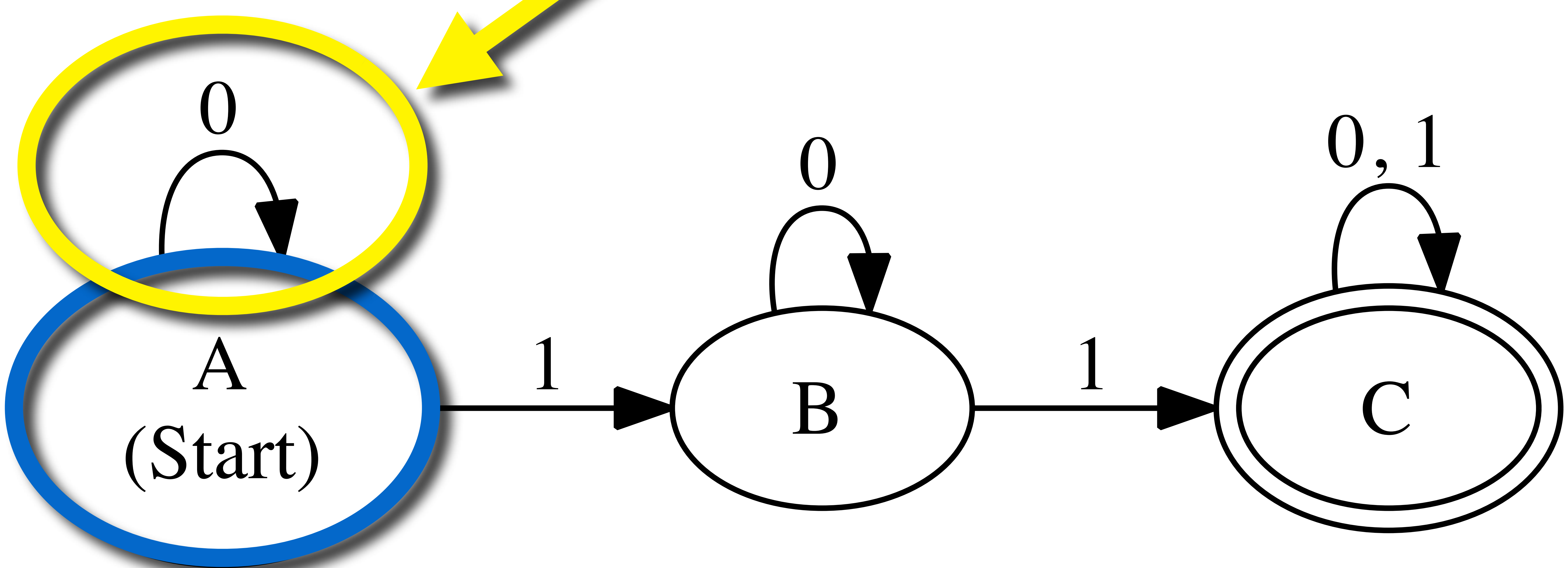


Self Transition

Given an input of ‘**0**’, if DFA is in **state A**, then **stay** in **state A** (and consume the input).

Deterministic

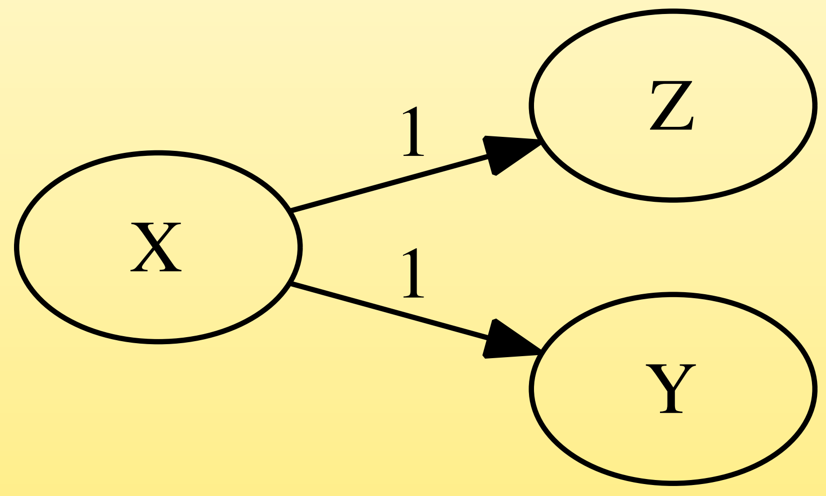
- Has a finite r
- Exactly one
- One or more **final states**.
- **Transitions**: define how automaton switches between states (given an input symbol).

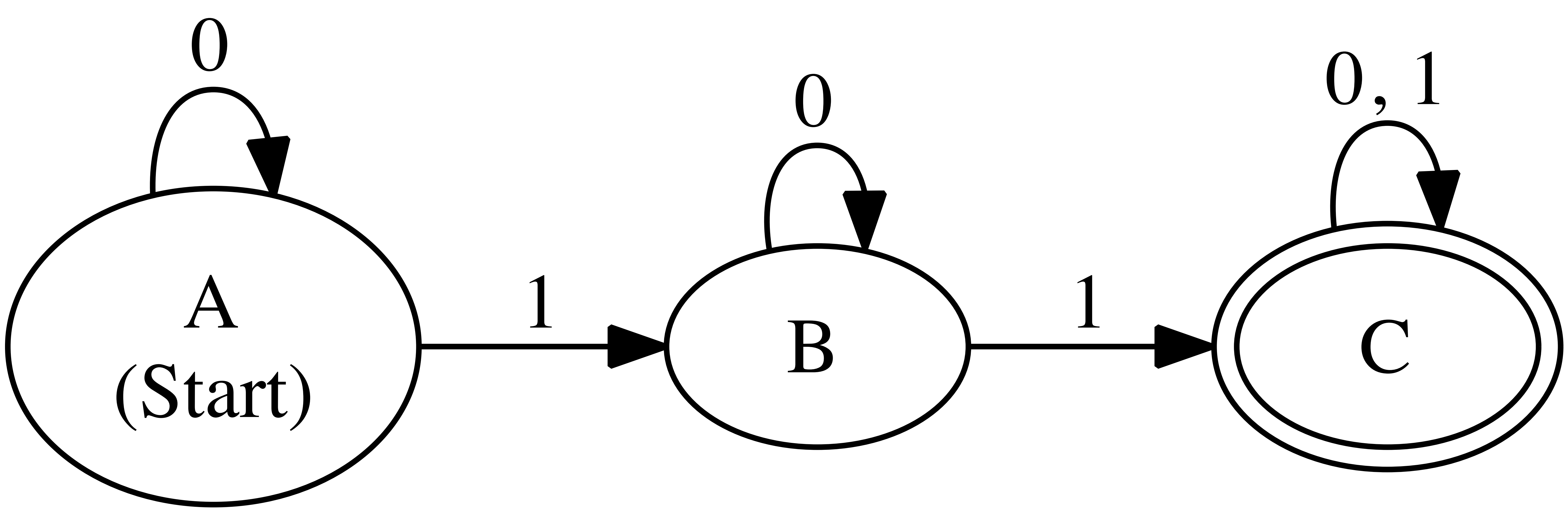


DFA 101

Transitions must be unambiguous:
For **each state and each input**, there exist **only one** transition. This is what makes the DFA ***deterministic***.

Not a legal DFA! ⟶





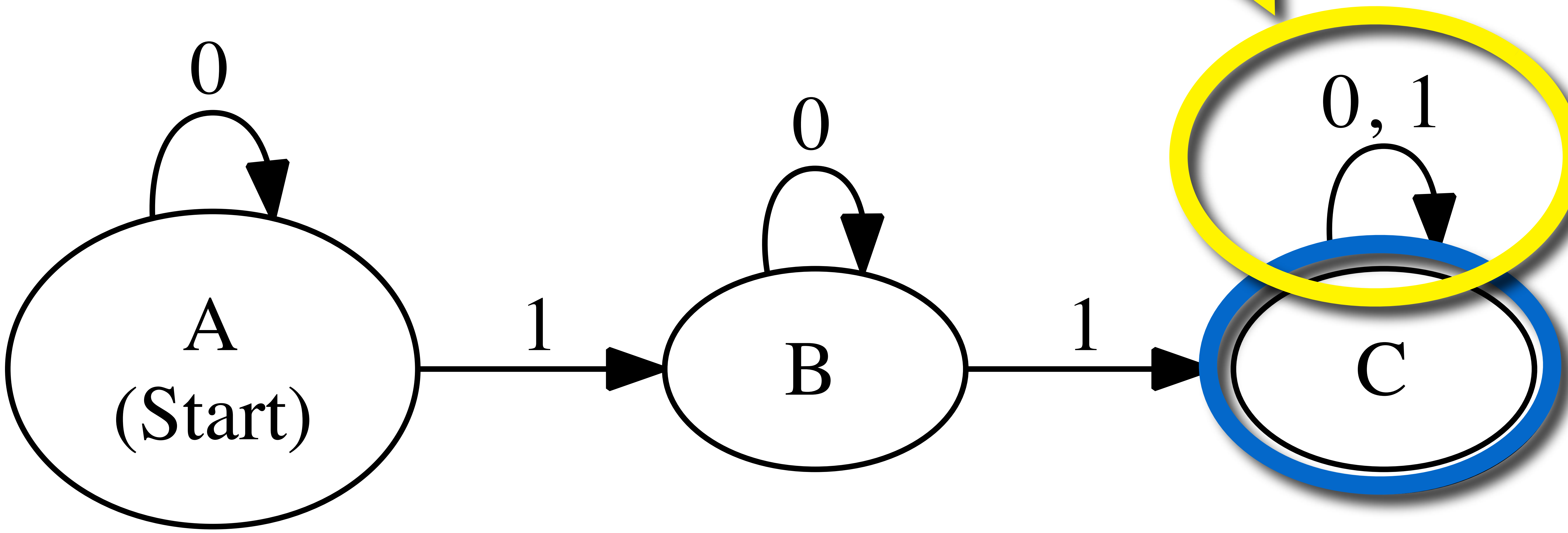
DFA 101

Determinis

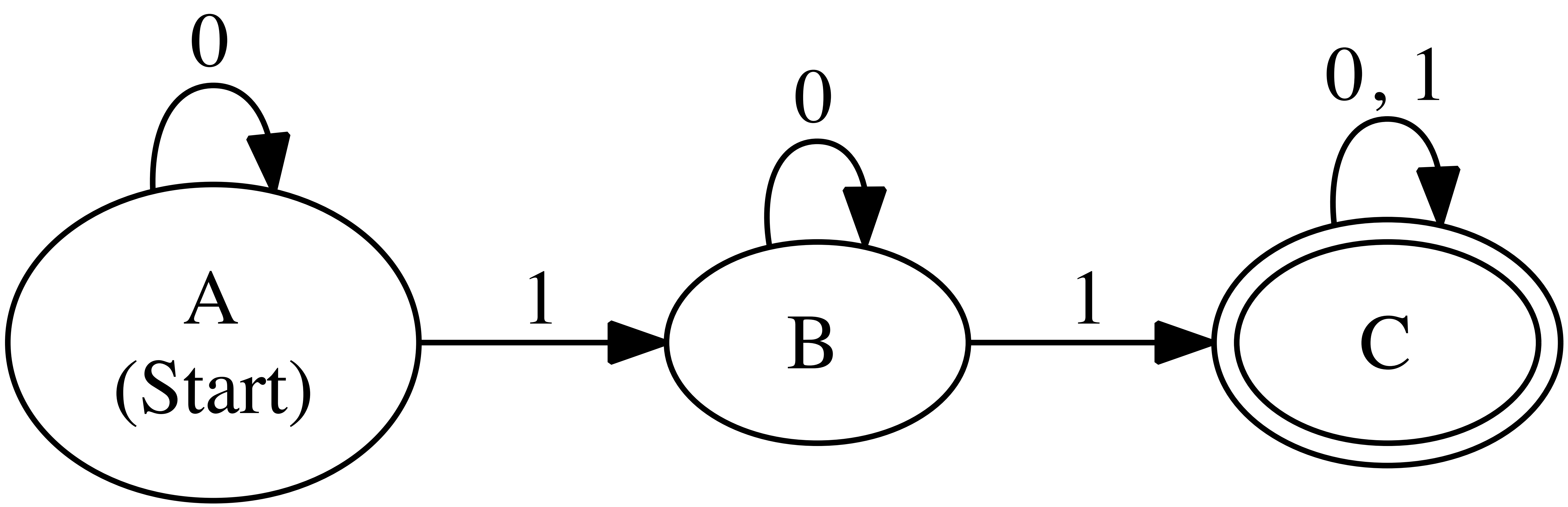
- ➡ Has a finit
- ➡ Exactly or
- ➡ One or m
- ➡ **Transitions**: define how automaton switches between states (given an input symbol).

Multiple Transitions

Given an input of **either ‘0’ or ‘1’**, if DFA is in **state C**, then stay in **state C** (and consume the input).



DFA String Processing



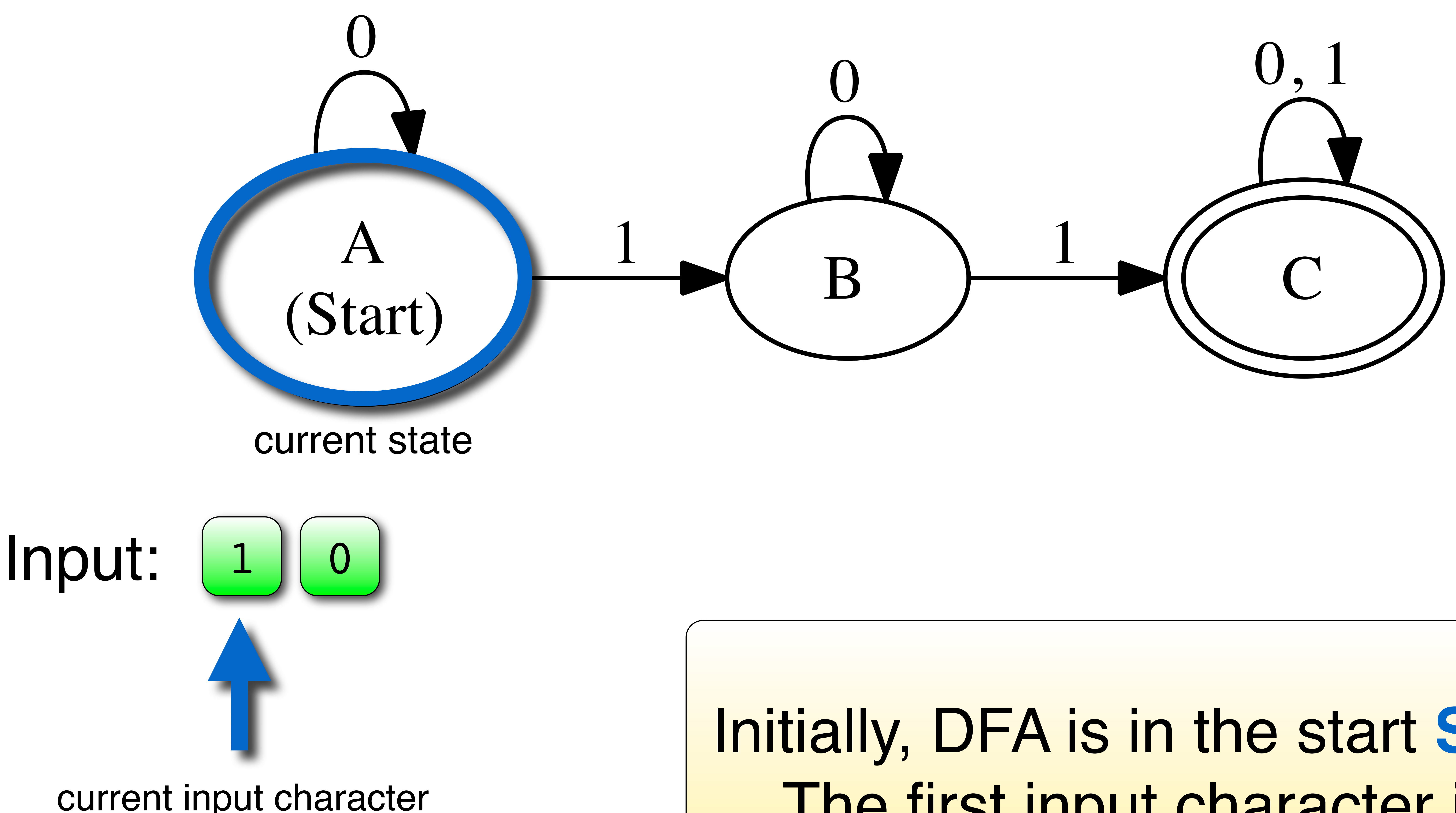
String processing.

- Initially in start state.
- Sequentially make transitions **each character** in input string.

A DFA either accepts or rejects a string.

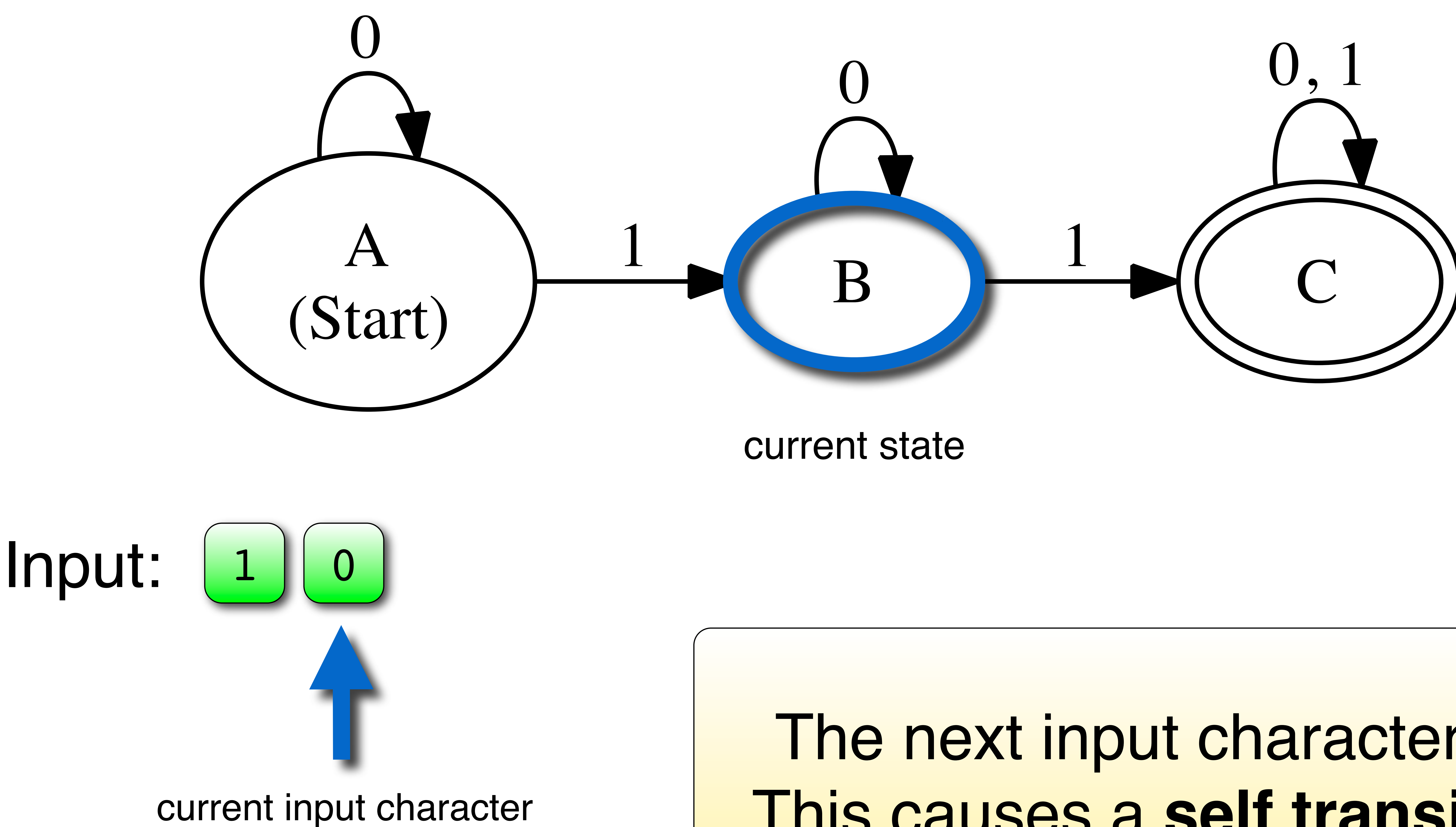
- **Reject** if a character is encountered for which no transition is defined in the current state.
- **Reject** if **end of input** is reached and DFA is not in a final state.
- **Accept** if **end of input** is reached and DFA is in final state.

DFA Example



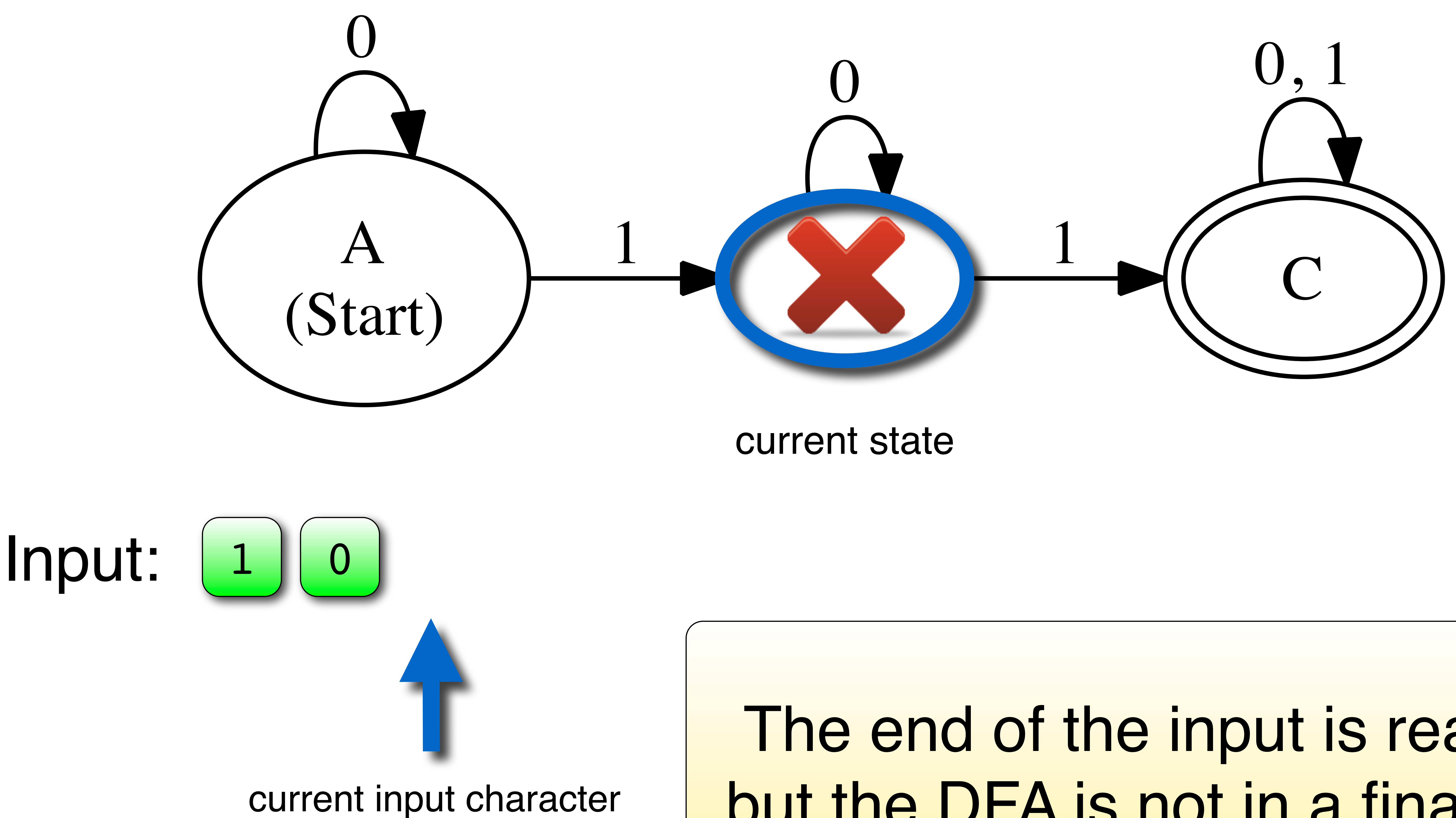
Initially, DFA is in the start **State A**.
The first input character is ‘1’.
This causes a transition to **State B**.

DFA Example



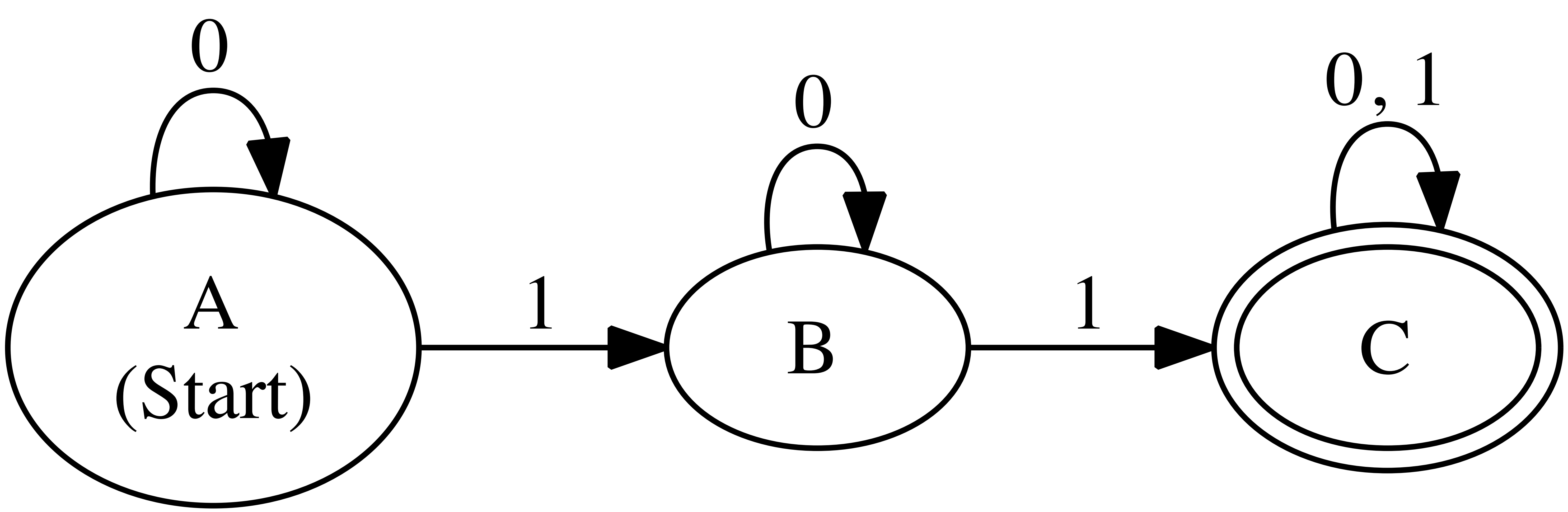
The next input character is ‘0’. This causes a **self transition** in **State B**.

DFA Example



The end of the input is reached, but the DFA is not in a final state: **the string ’10’ is rejected**!

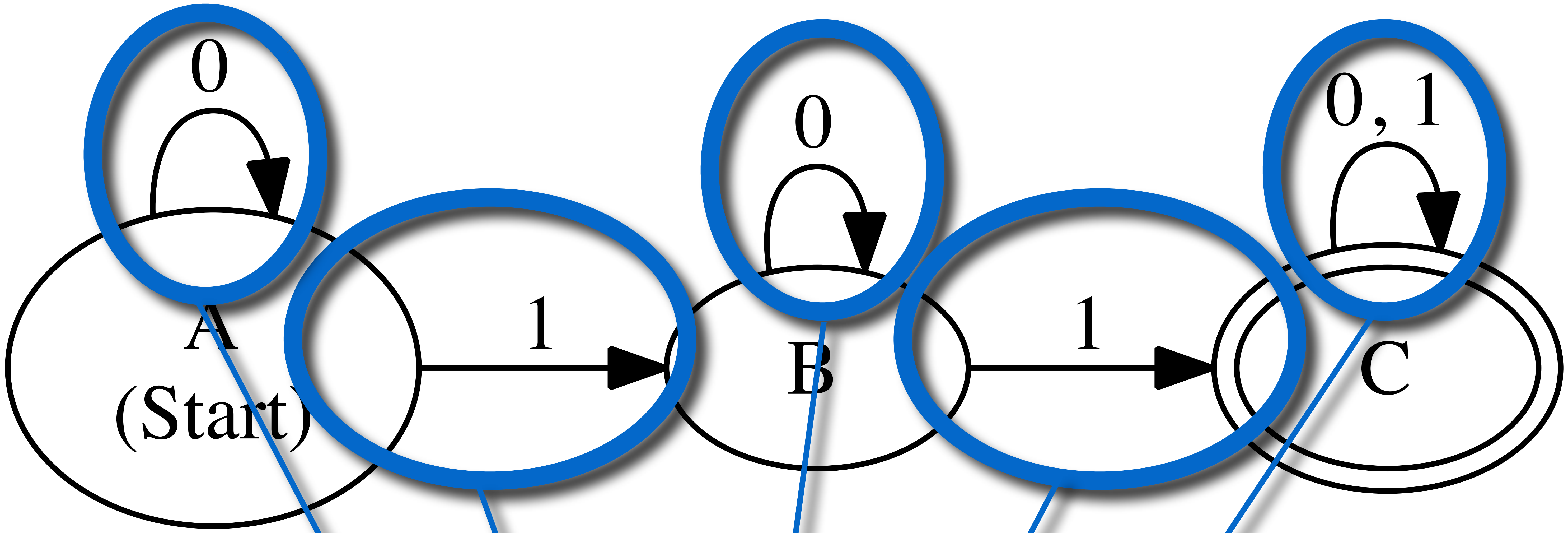
DFA-Equivalent Regular Expression



What's the RE such that the RE's language is **exactly** the set of strings that is **accepted** by this DFA?

0*10*1(1|0)*

DFA-Equivalent Regular Expression



What's the RE such that the RE's language is **exactly** the set of strings that is **accepted** by this DFA?

0*10*1(1|0)*

Recognizing Tokens with a DFA

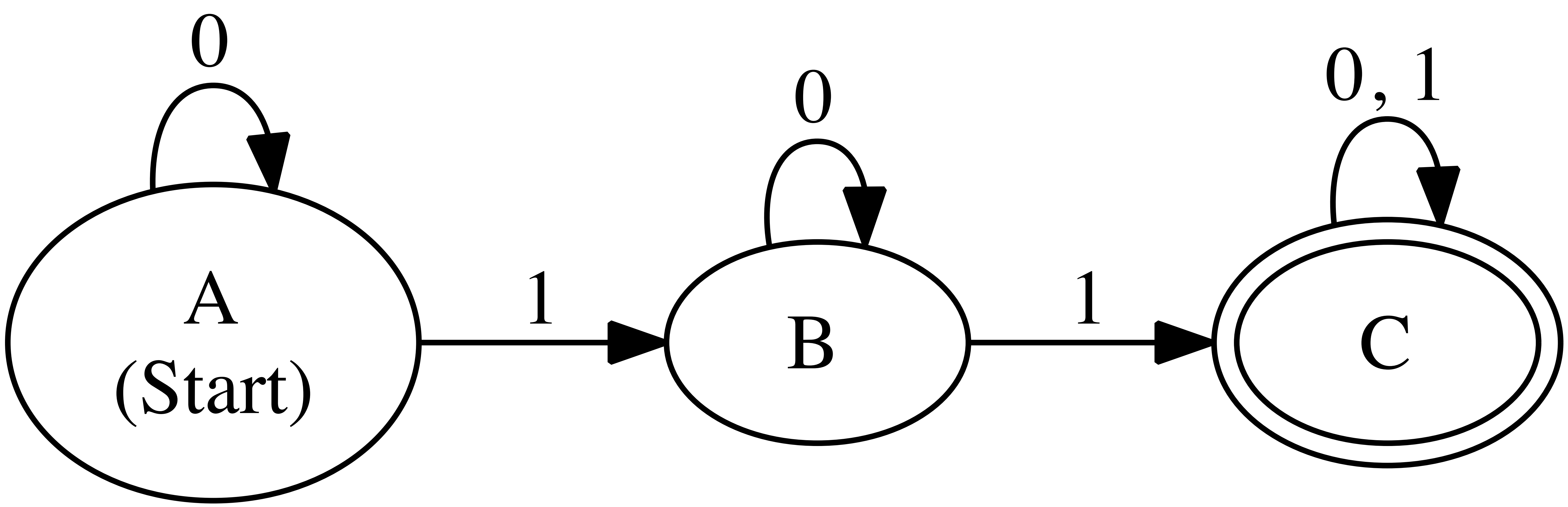


Table-driven implementation.

➡ DFA's can be represented as a 2-dimensional table.

Current State	On '0'	On '1'	Note
A	transition to A	transition to B	start
B	transition to B	transition to C	—
C	transition to C	transition to C	final

Recognizing Tokens with a DFA

```
currentState = start state;
while end of input not yet reached: {
  c = get next input character;
  if transitionTable[currentState][c] ≠ null:
    currentState = transitionTable[currentState][c]
  else:
    reject input
}
if currentState is final:
  accept input
else:
  reject input
```

Current State	On '0'	On '1'	Note
A	transition to A	transition to B	start
B	transition to B	transition to C	—
C	transition to C	transition to C	final

Recognizing Tokens with a DFA

```
currentState = start state;
while end of input not yet reached: {
  c = get next input character;
  if transitionTable[currentState][c] ≠ null:
    currentState = transitionTable[currentState][c]
  else:
    reject input
}
if currentState is final:
  accept input
else:
  reject input
```

C

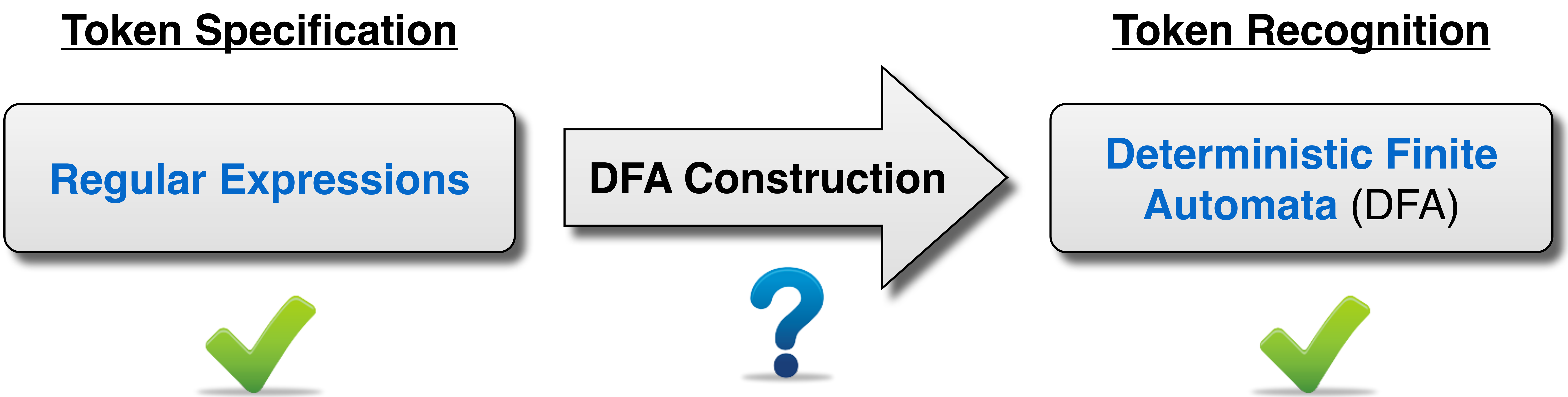
This accepts **exactly one** token in the input.
A real lexer must detect **multiple successive** tokens.

This can be achieved by **resetting** to the start state.
But what happens if the **suffix** of one token is the **prefix** of another?
(See Chapter 2 for a solution.)

Lexical Analysis

The need to identify tokens raises two questions.

- ➡ How can we **specify the tokens** of a language?
  - With regular expressions.
- ➡ How can we **recognize tokens** in a character stream?
  - With DFAs.



No single-step algorithm:
We first need to construct a Non-Deterministic Finite Automaton…

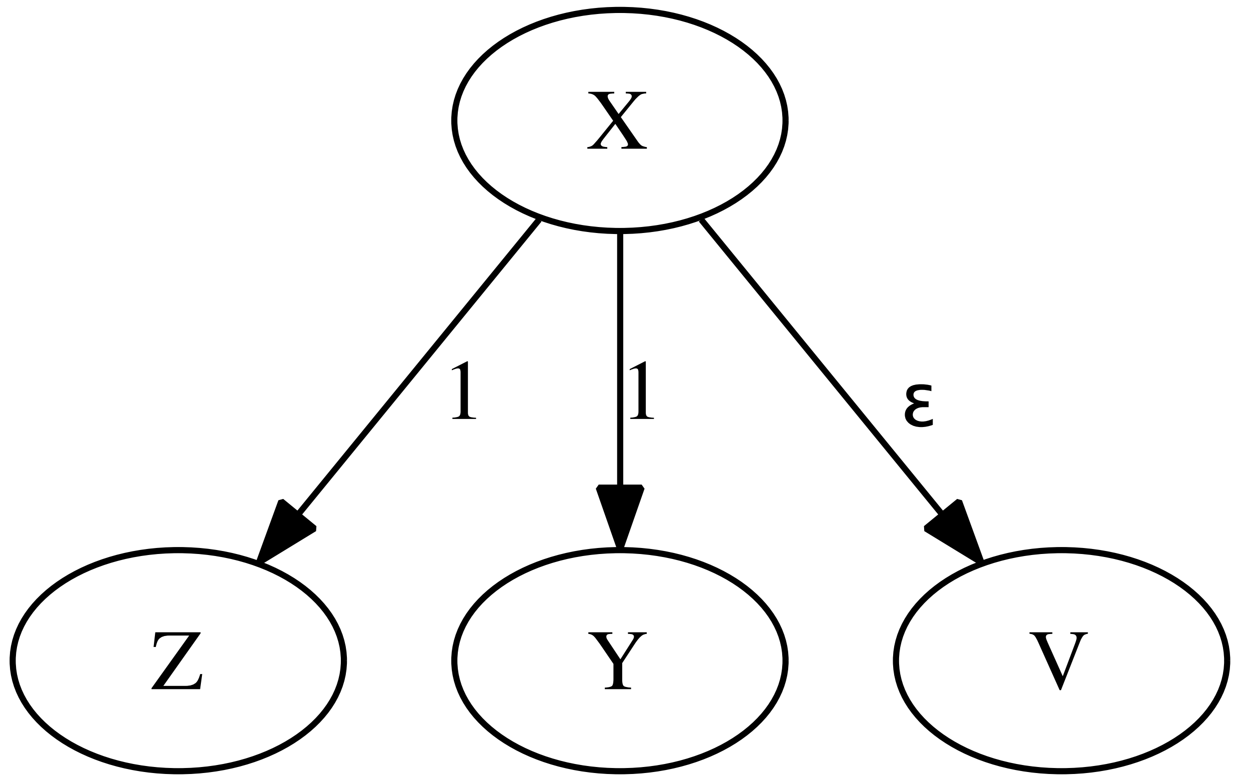
Non-Deterministic Finite Automaton (NFA)

Like a DFA, but less restrictive:

- Transitions do **not** have to be **unique**: each state may have **multiple ambiguous transitions** for the same input symbol. (Hence, it can be ***non-deterministic***.)
- **Epsilon transitions** do not consume any input. (They correspond to the empty string.)
- Note that every DFA is also a NFA.

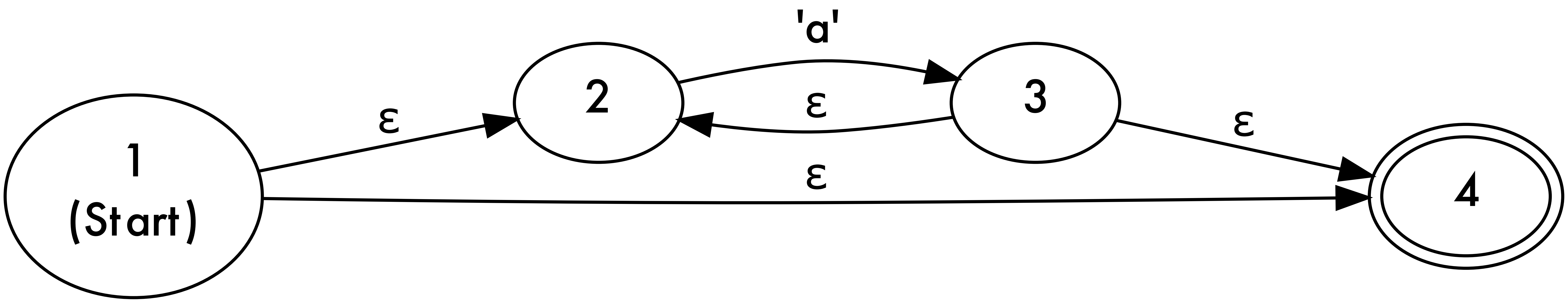
Acceptance rule:

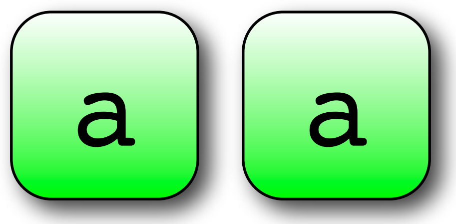
- **Accepts** an input string if **there exists** a series of transitions such that the NFA is in a final state when the end of input is reached.
- Inherent parallelism: all possible paths are **explored simultaneously**.



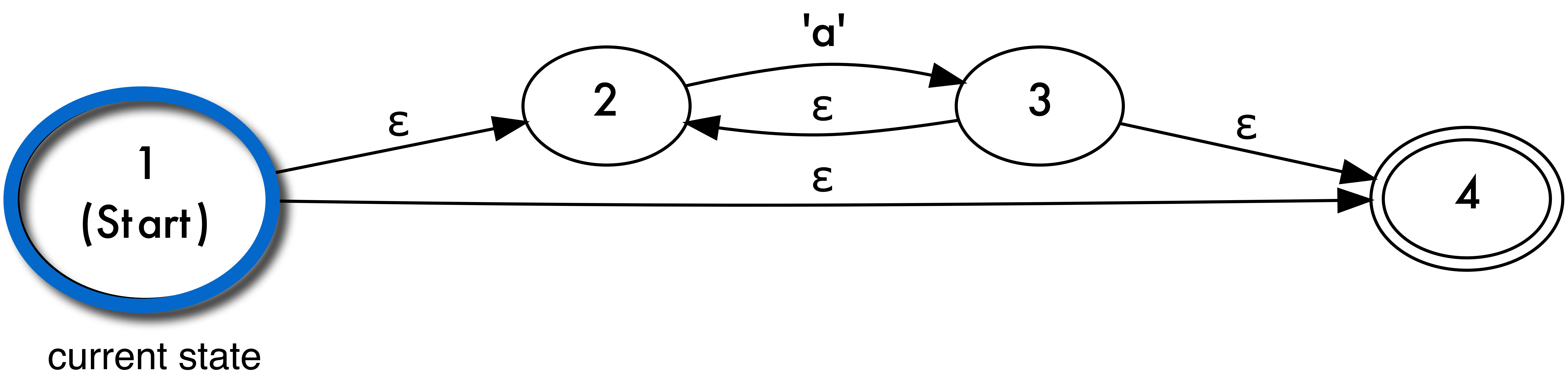
A legal NFA fragment.

NFA Example



Input: 

NFA Example

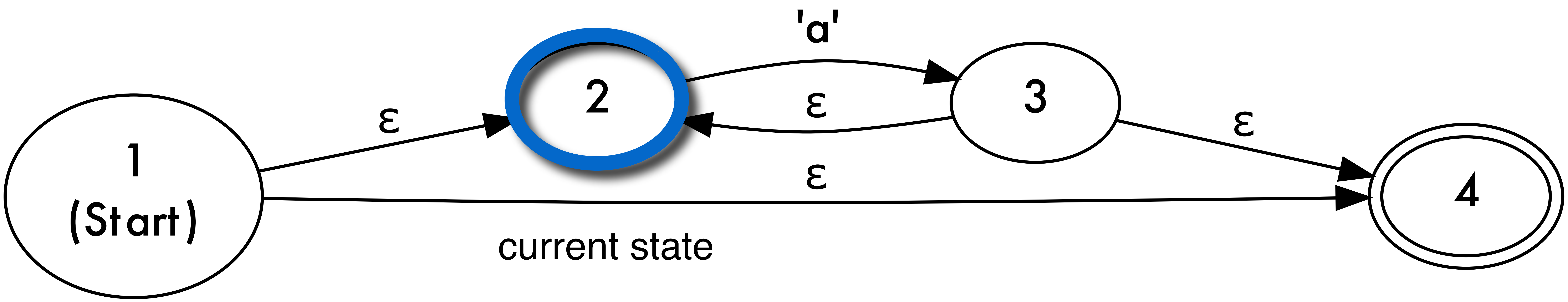


Input: a a

current input character

Epsilon transition:
Can transition from **State 1** to **State 2** without consuming any input.

NFA Example

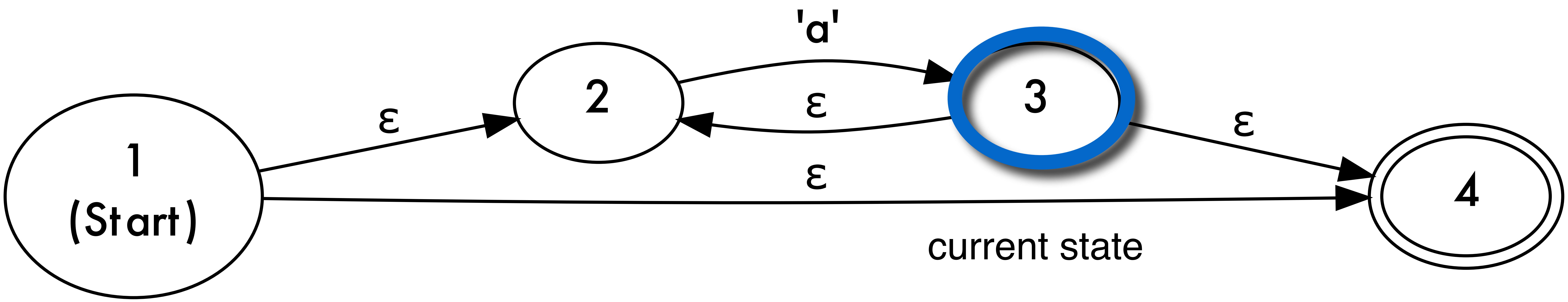


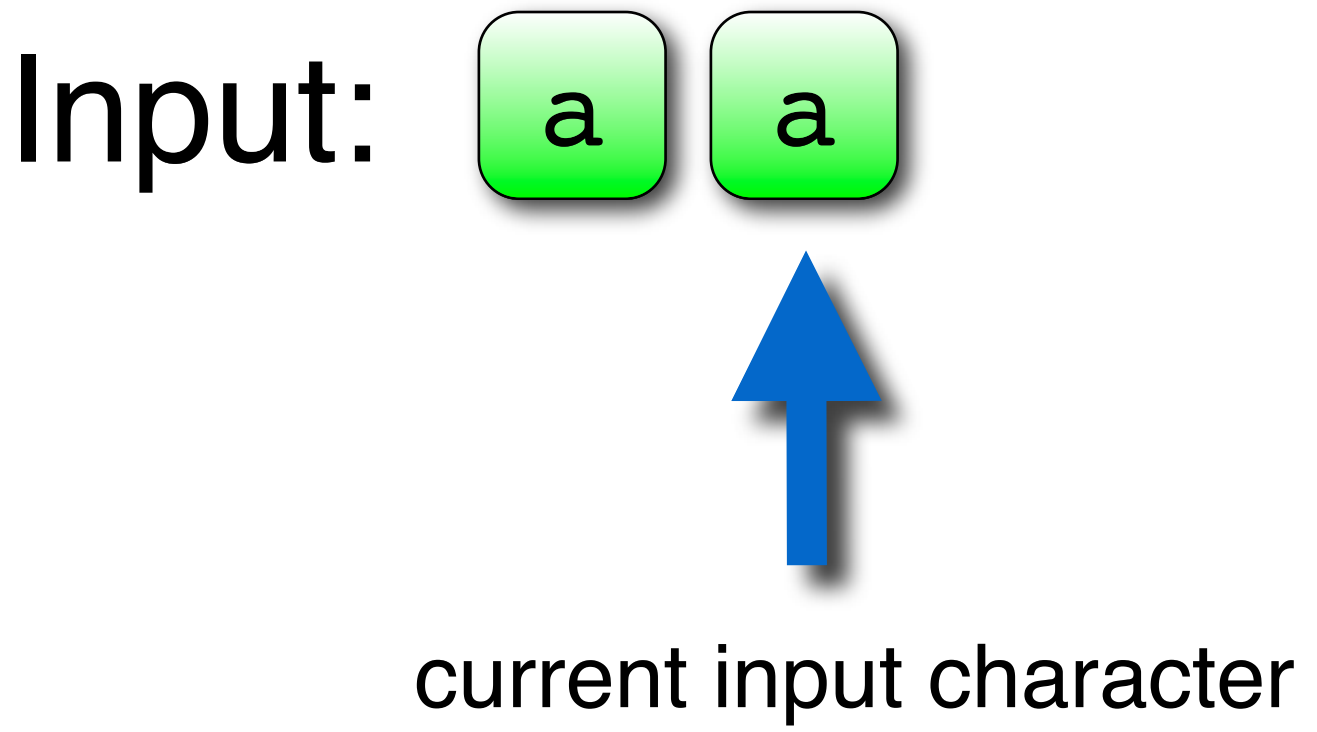
Input: a a

current input character

Regular transition:
Can transition from **State 2** to **State 3**, which consumes the first '**a**'.

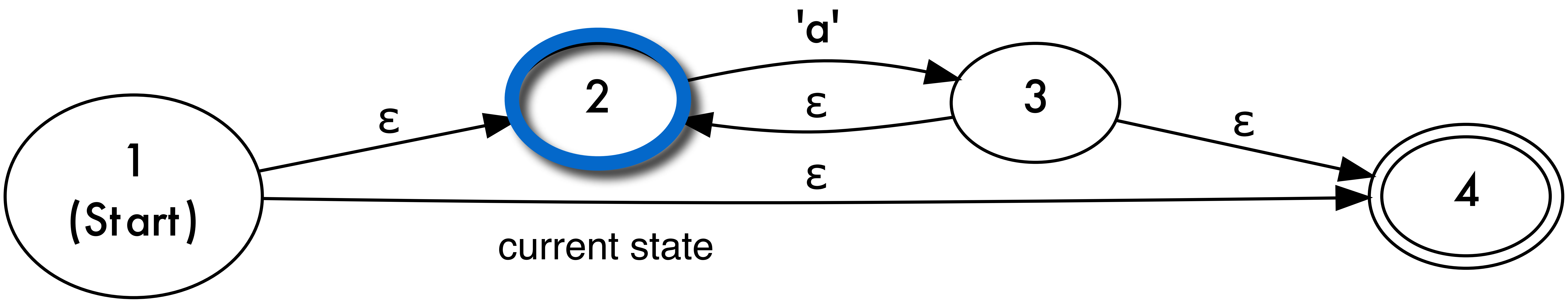
NFA Example





Epsilon transition:
Can transition from **State 3** to **State 2** without consuming any input.

NFA Example

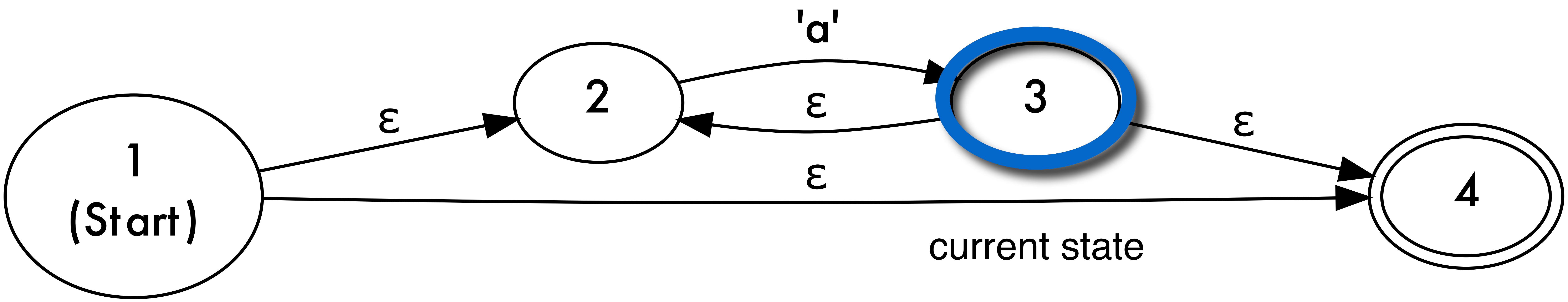


Input: a a

current input character

Regular transition:
Can transition from **State 2** to **State 3**, which consumes the second '**a**'.

NFA Example



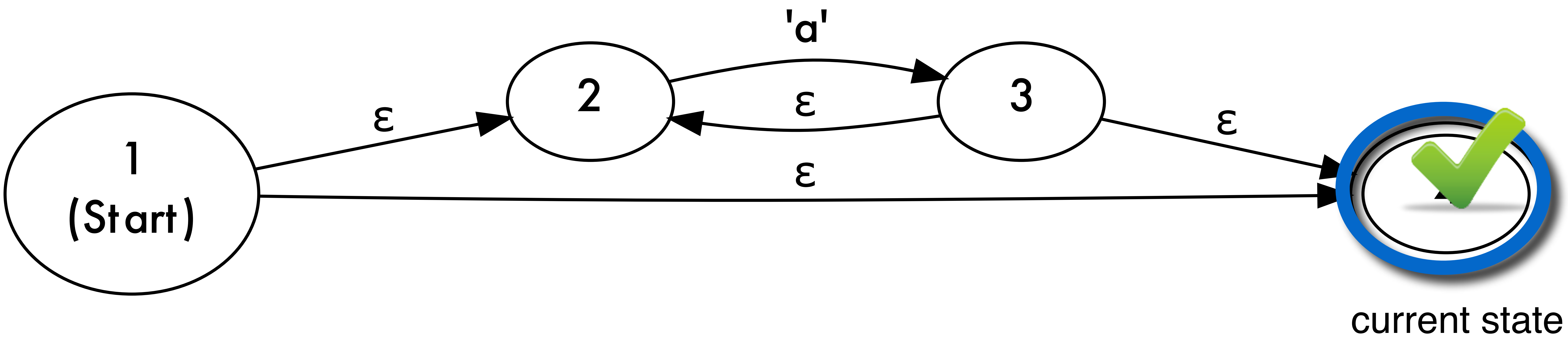
Input: a a

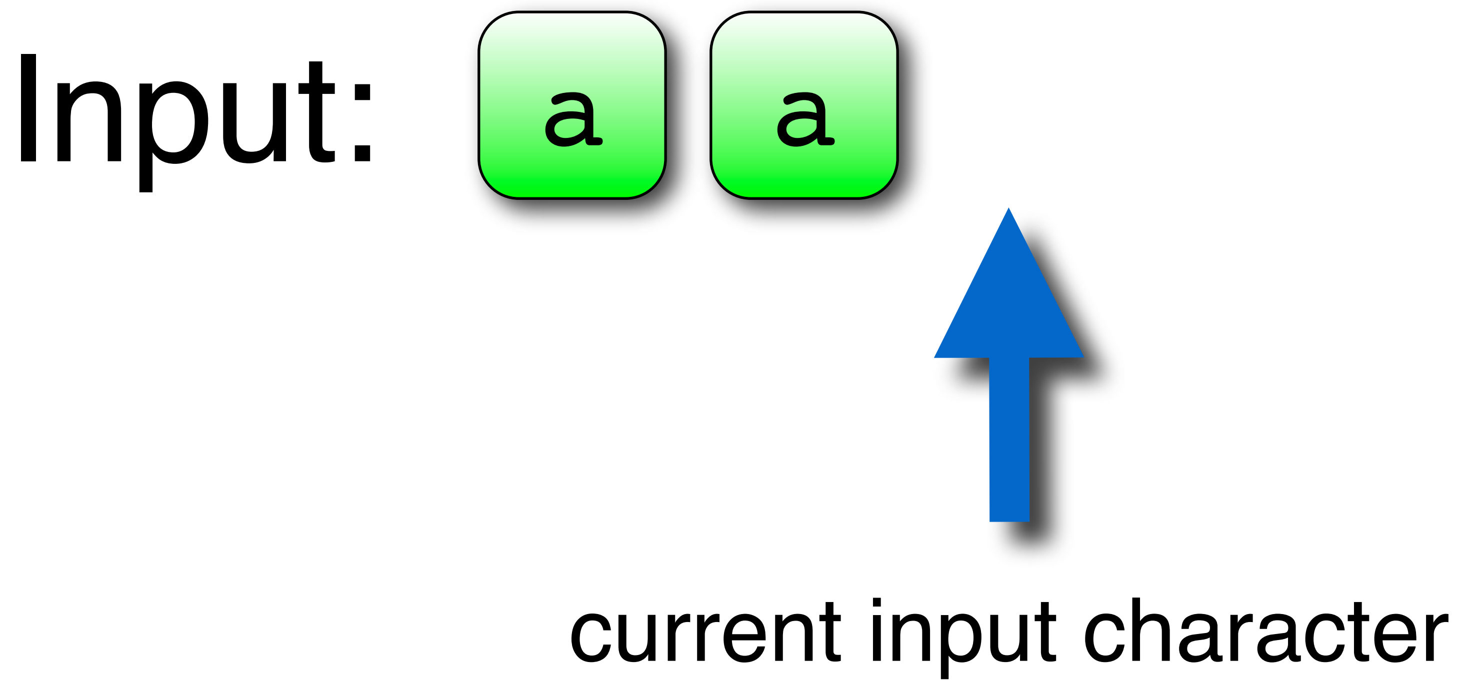
current input character

Epsilon transition from **State 3** to **4**:

End of input reached, but the NFA can still carry out epsilon transitions.

NFA Example





Input Accepted:

There exists a sequence of transitions such that the NFA is in a **final state** at the **end of input**.

Equivalent DFA Construction

Constructing a DFA corresponding to a RE.

➡ In theory, this requires **two steps**.

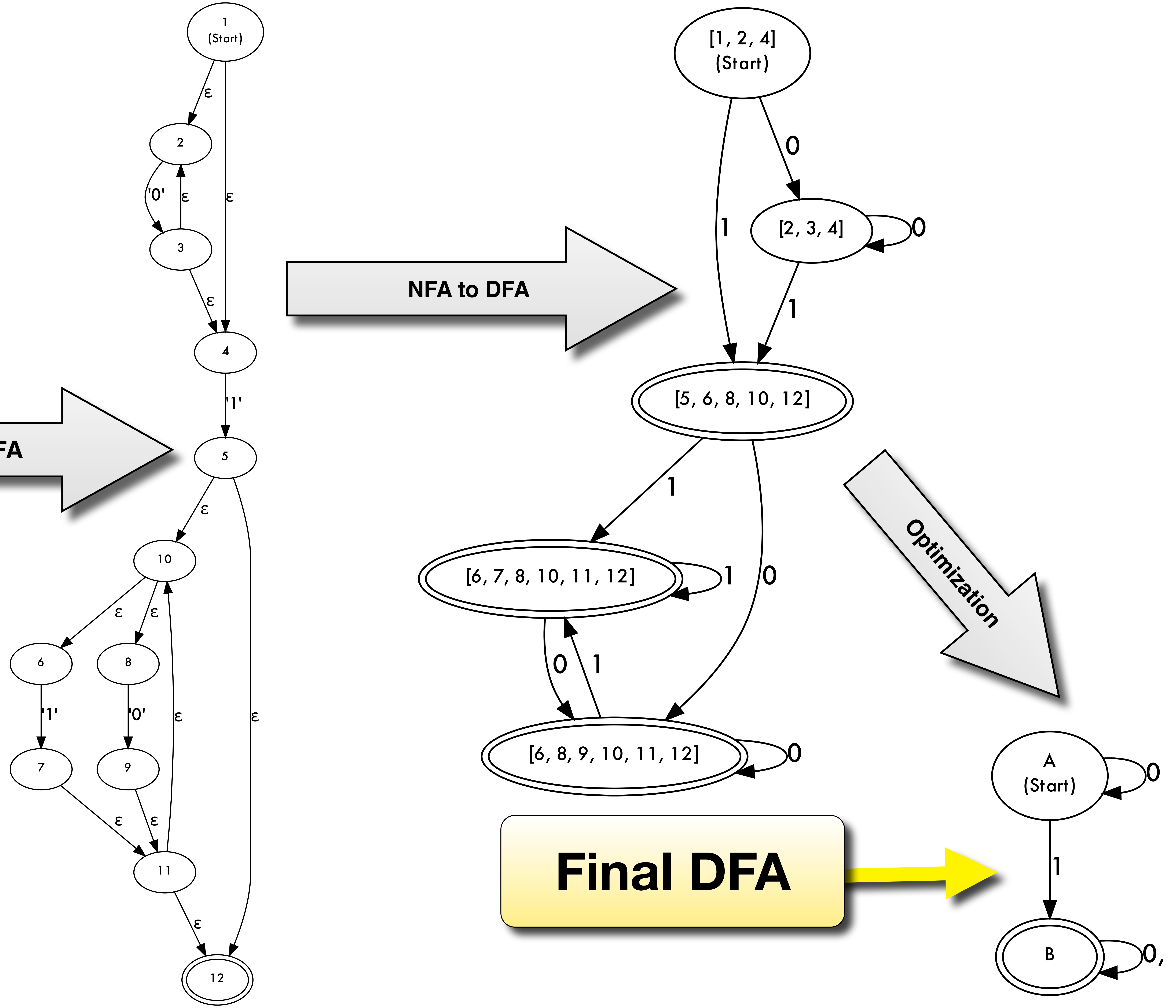
- From a **RE** to an equivalent **NFA**.
- From the **NFA** to an equivalent **DFA**.

To be practical, we require a third optimization step.

➡ Large **DFA** to **minimal DFA**.

Example

0*1(1|0)*



Step 1: RE → NFA

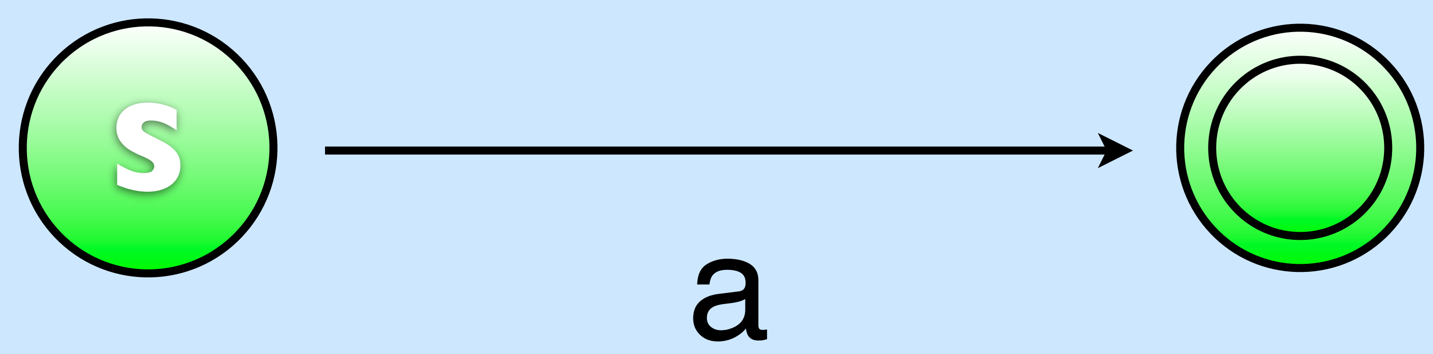
Every RE can be converted to a NFA by repeatedly applying four simple rules.

- **Base case**: a single character.
- **Concatenation**: joining two REs in sequence.
- **Alternation**: joining two REs in parallel.
- **Kleene Closure**: repeating a RE.

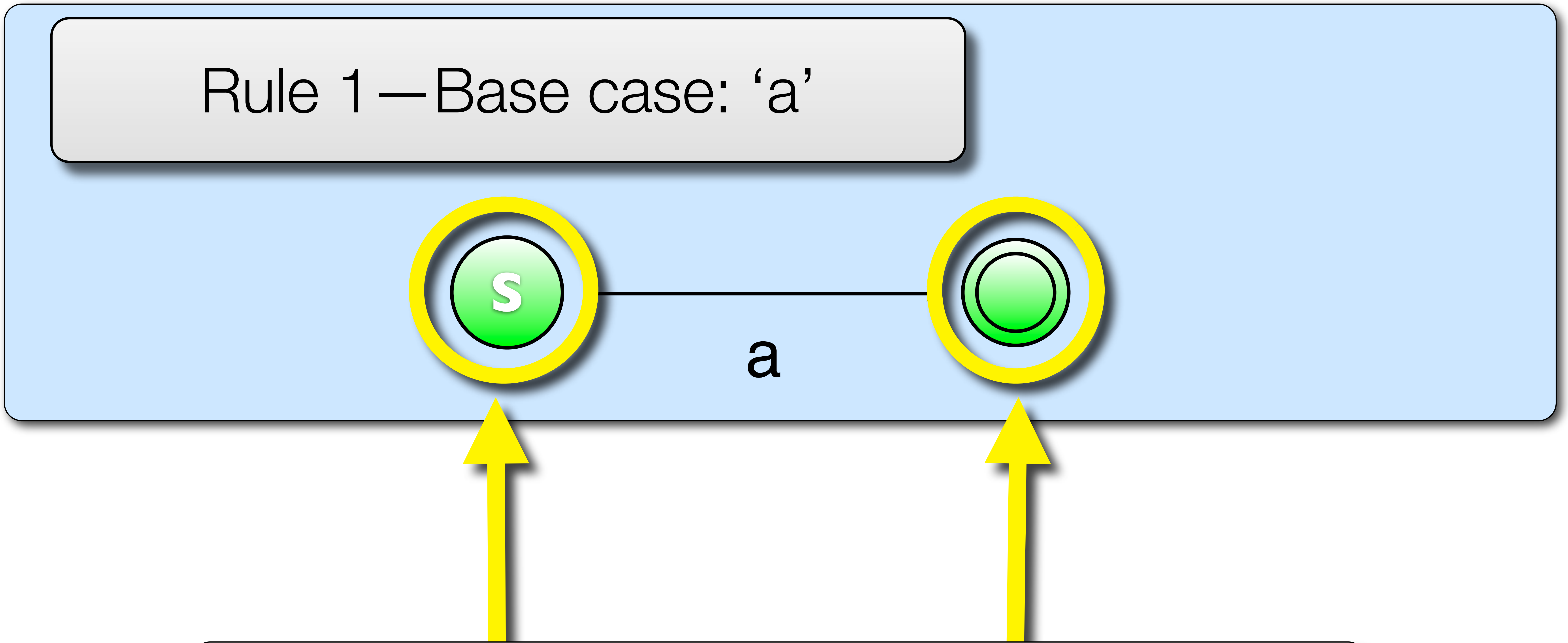
(recall the definition of a RE)

The Four NFA Construction Rules

Rule 1—Base case: ‘a’

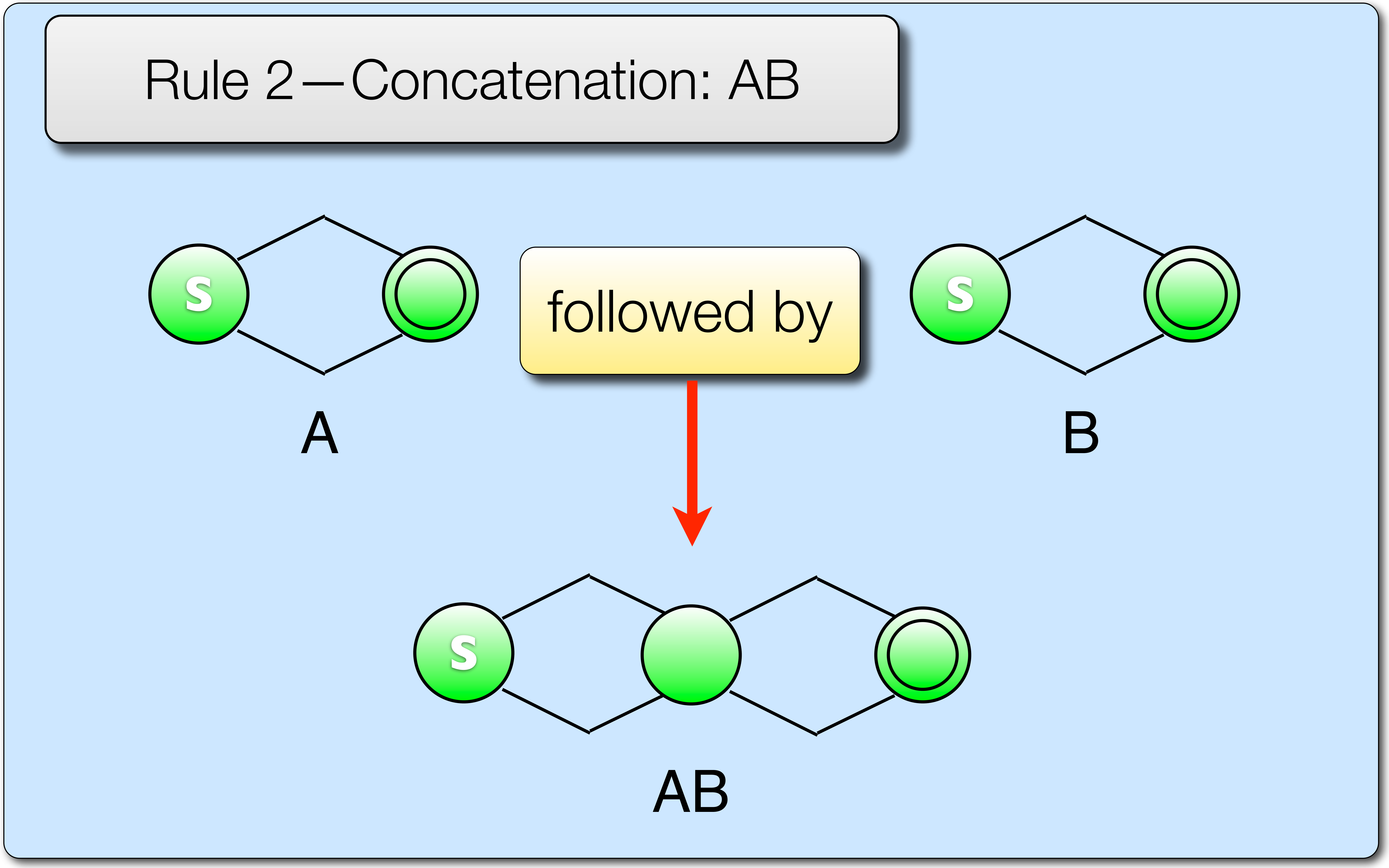


The Four NFA Construction Rules



Simple two-state NFA (even DFA, too).

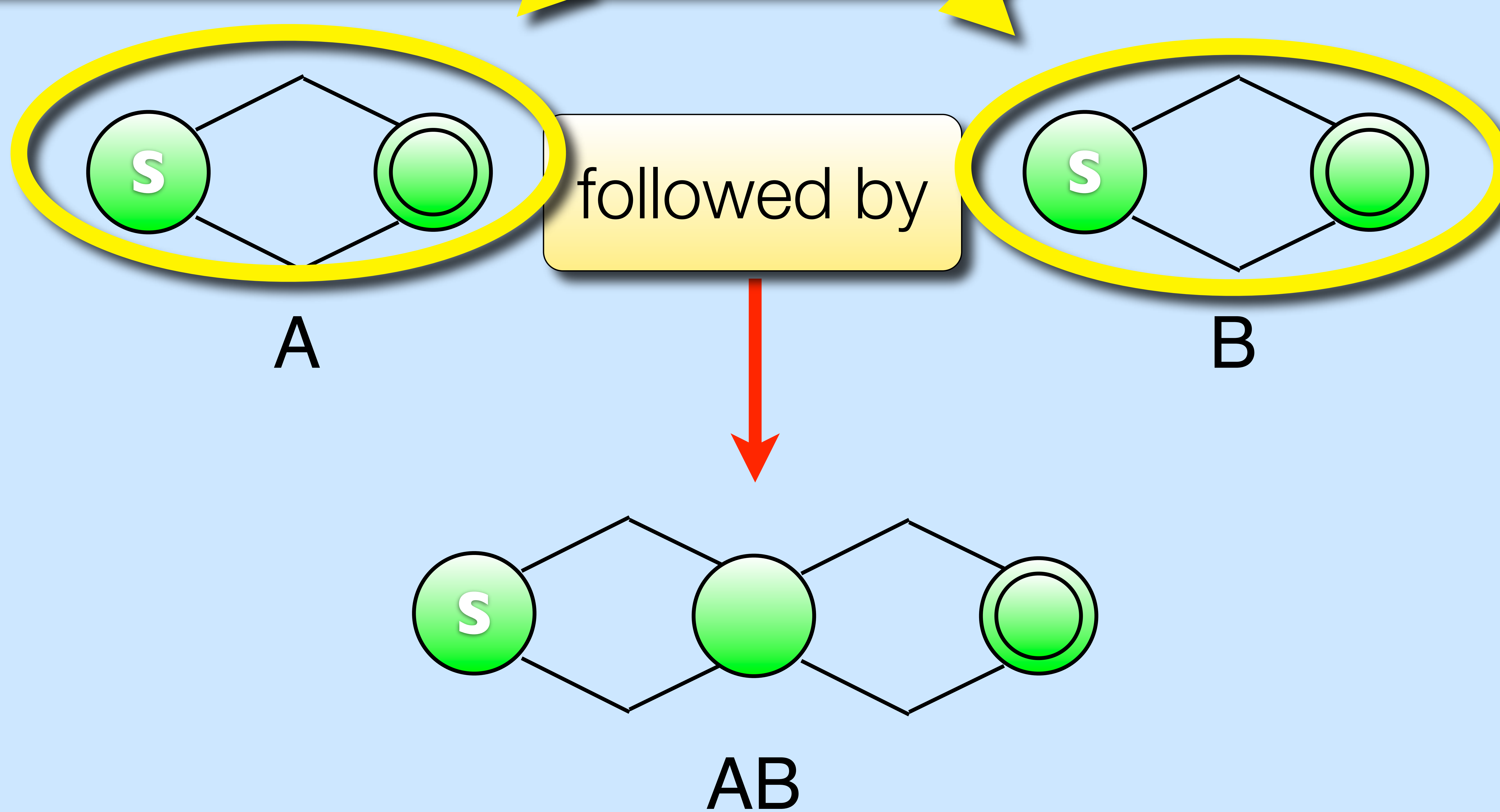
The Four NFA Construction Rules



The Fo… Rules

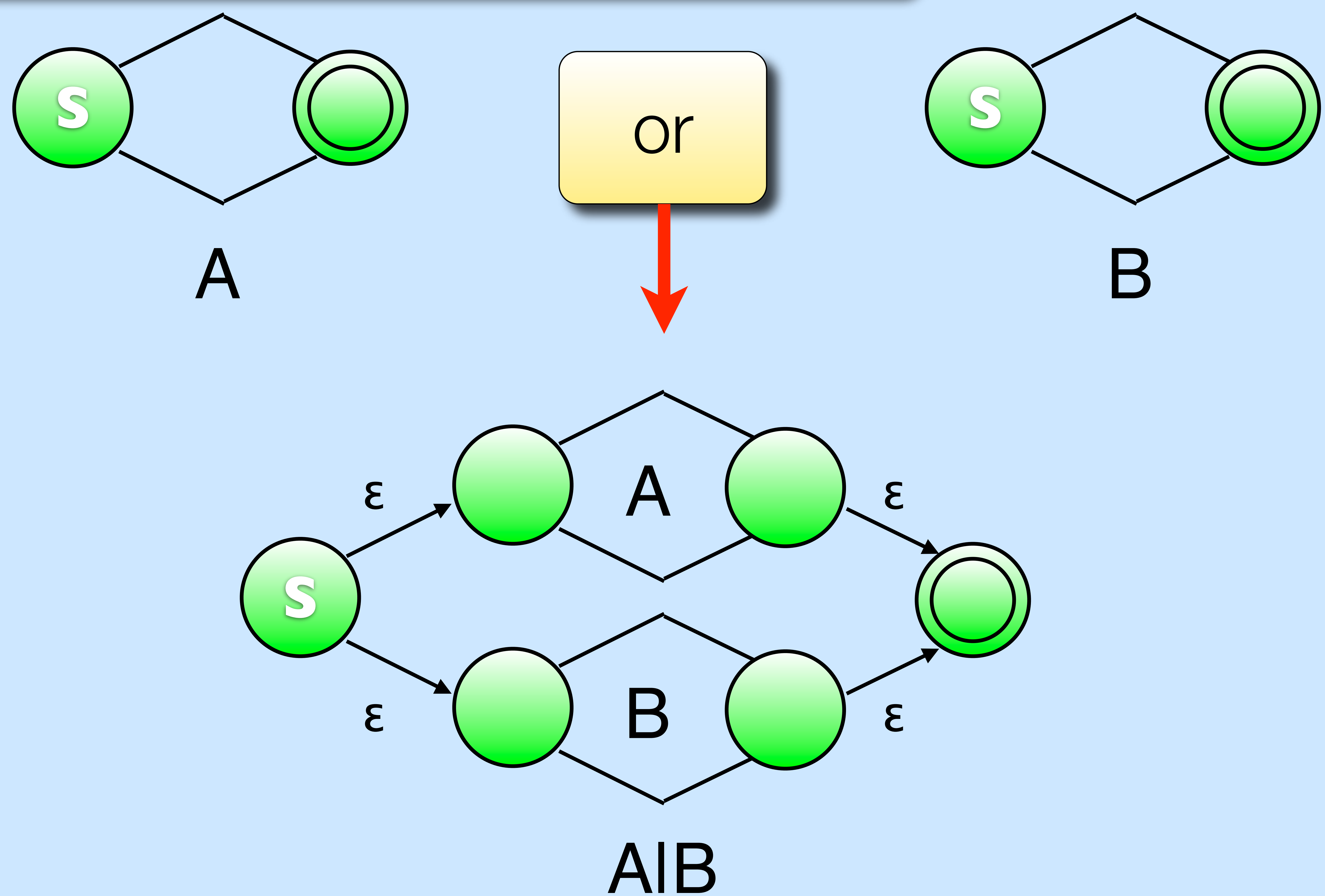
Not just two states, but any NFA with a **single final state**.

Rule 2—Concatenation: AB



The Four NFA Construction Rules

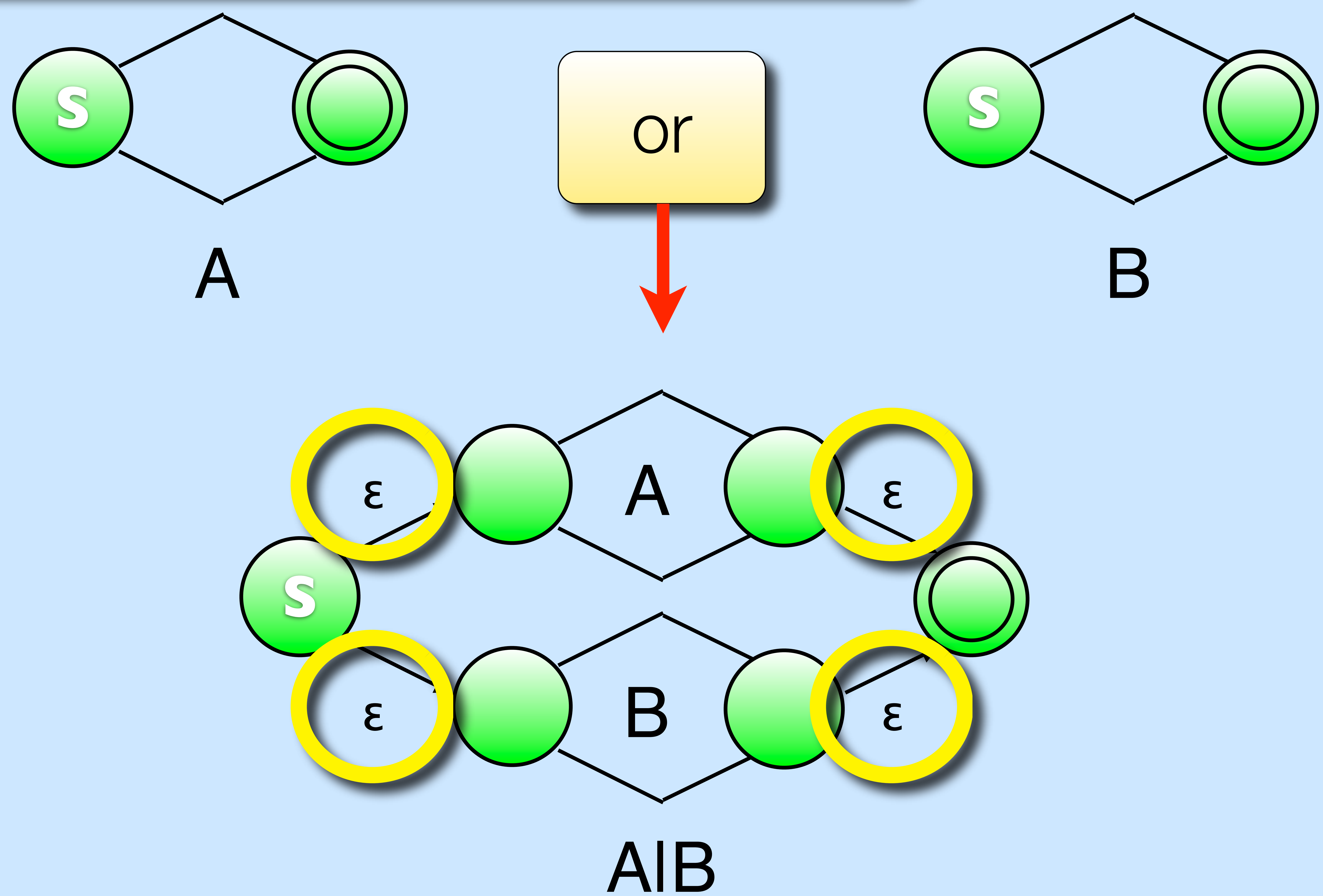
Rule 3--Alternation: “A|B”



The Four NFA Construction Rules

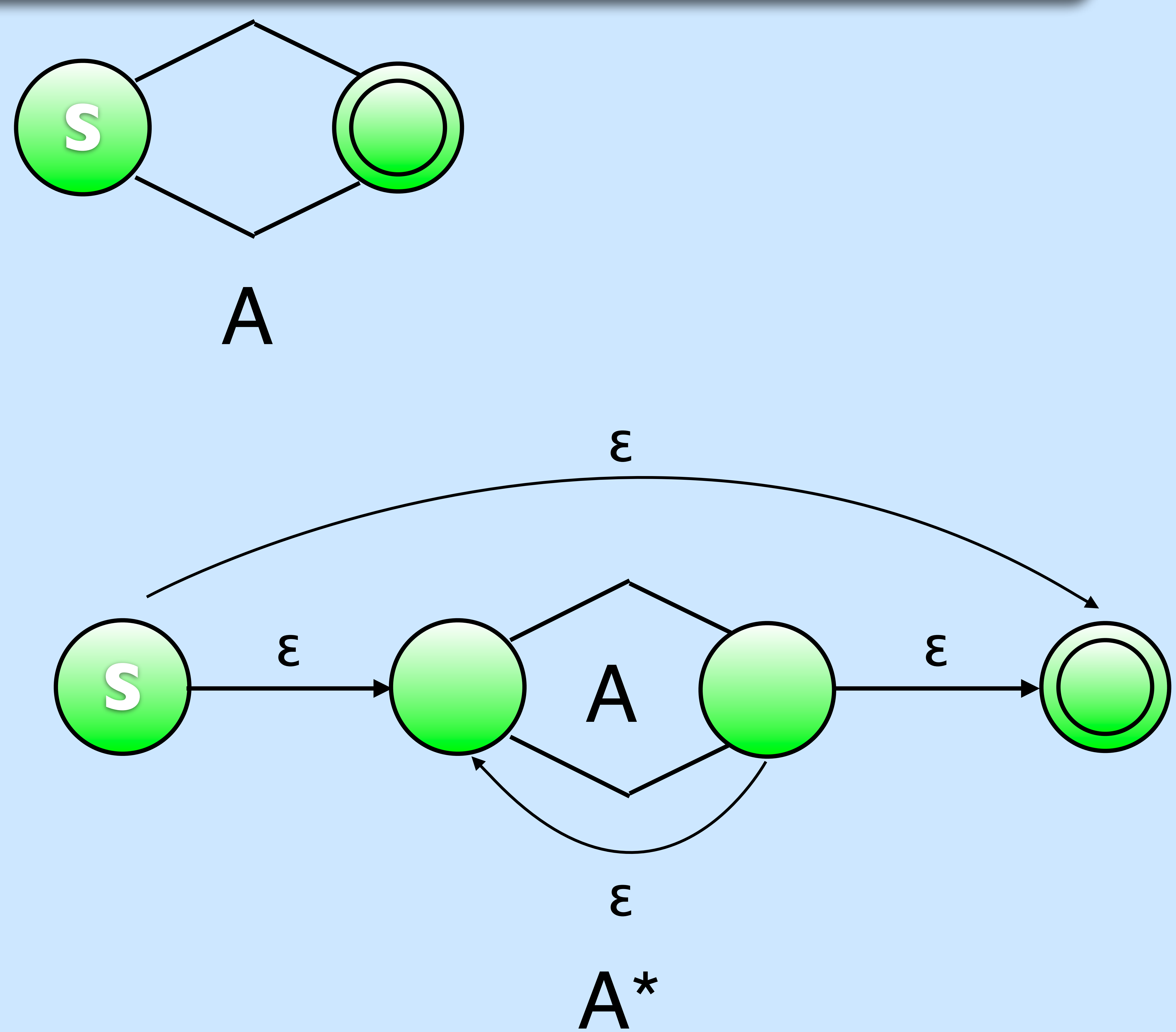
Notice the epsilon transitions.

Rule 3--Alternation: "A|B"



The Four NFA Construction Rules

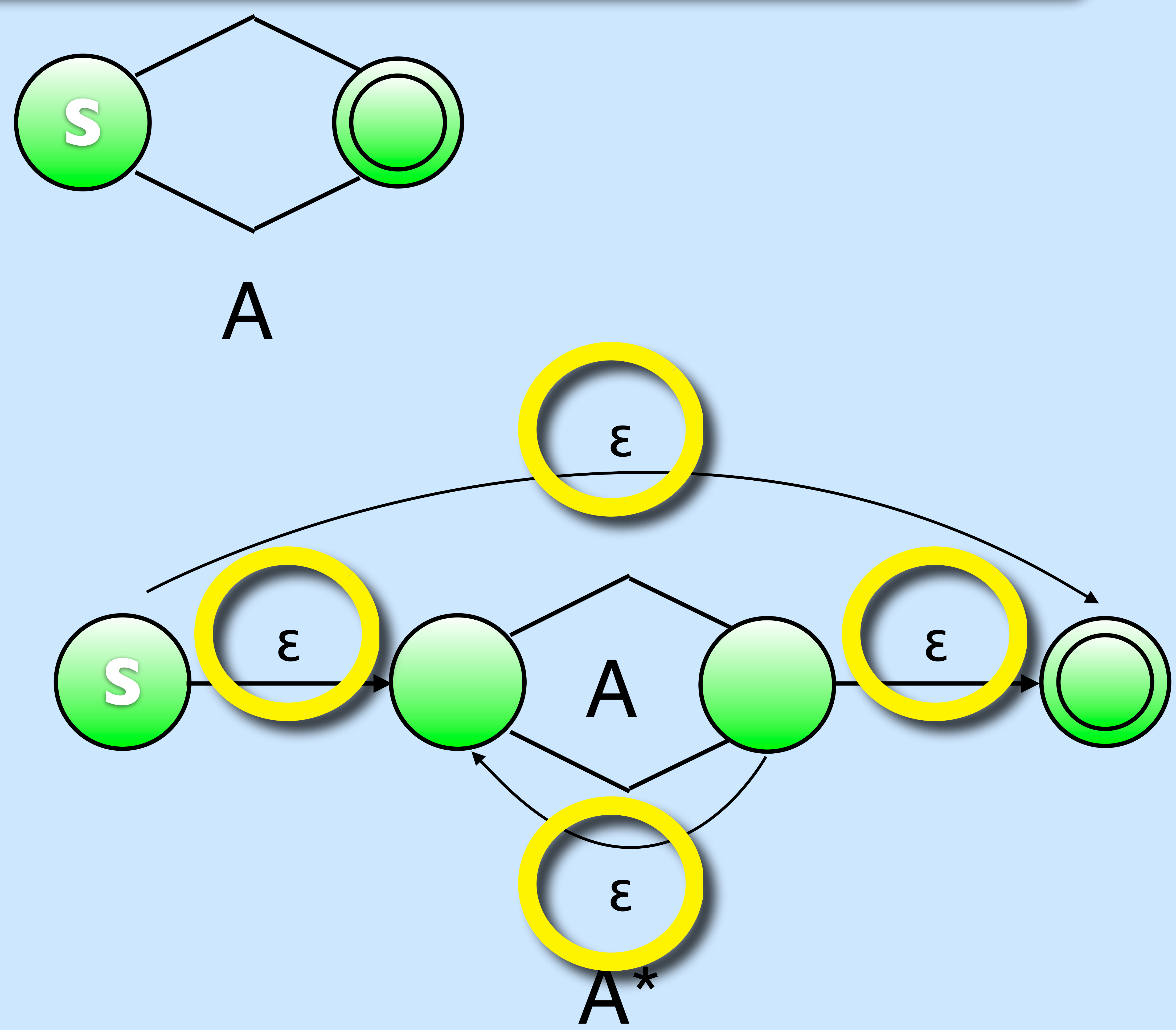
Rule 4—Kleene Closure: “A*”



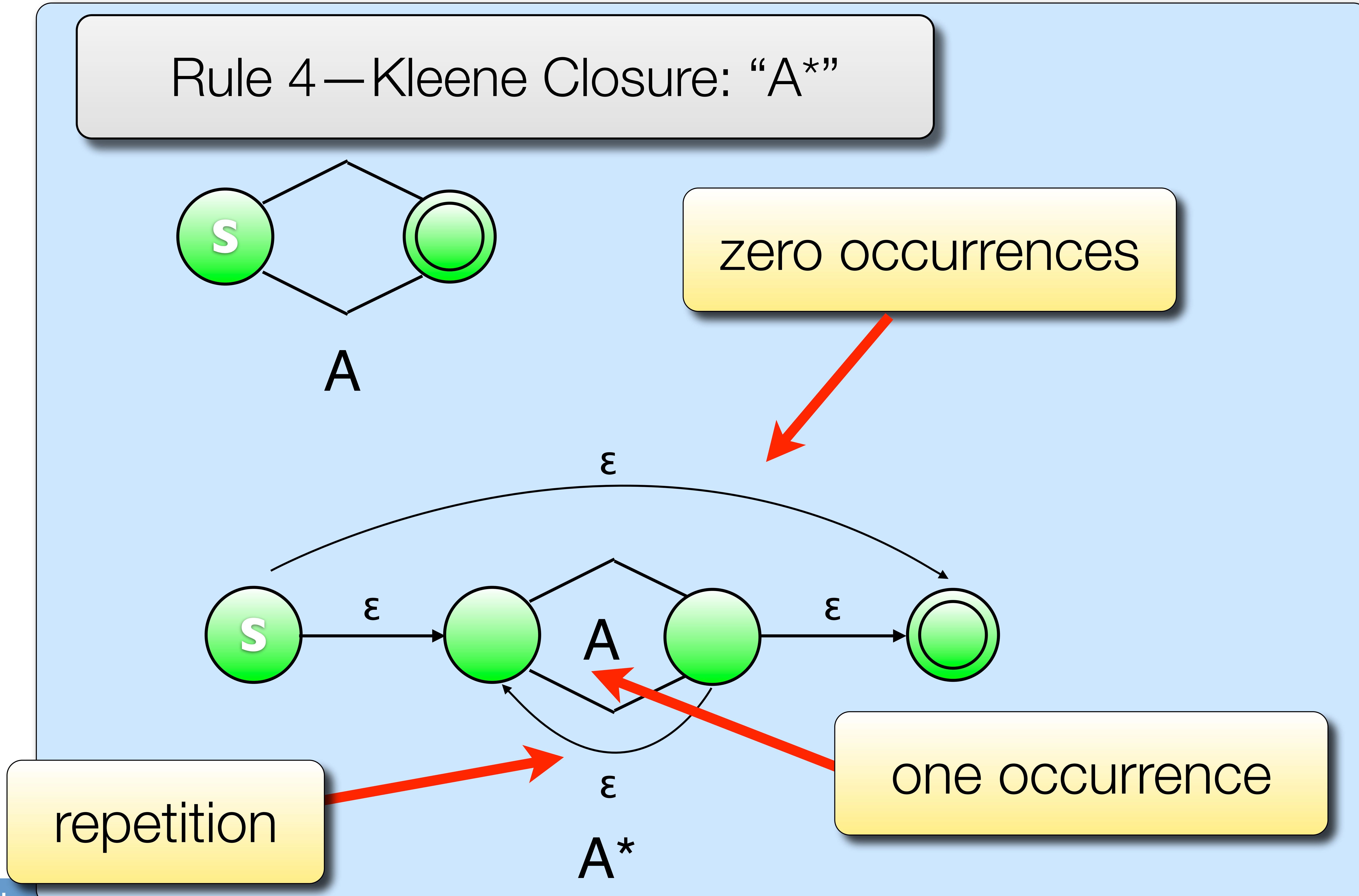
The Four NFA Construction Rules

Notice the epsilon transitions.

Rule 4—Kleene Closure: "A*"



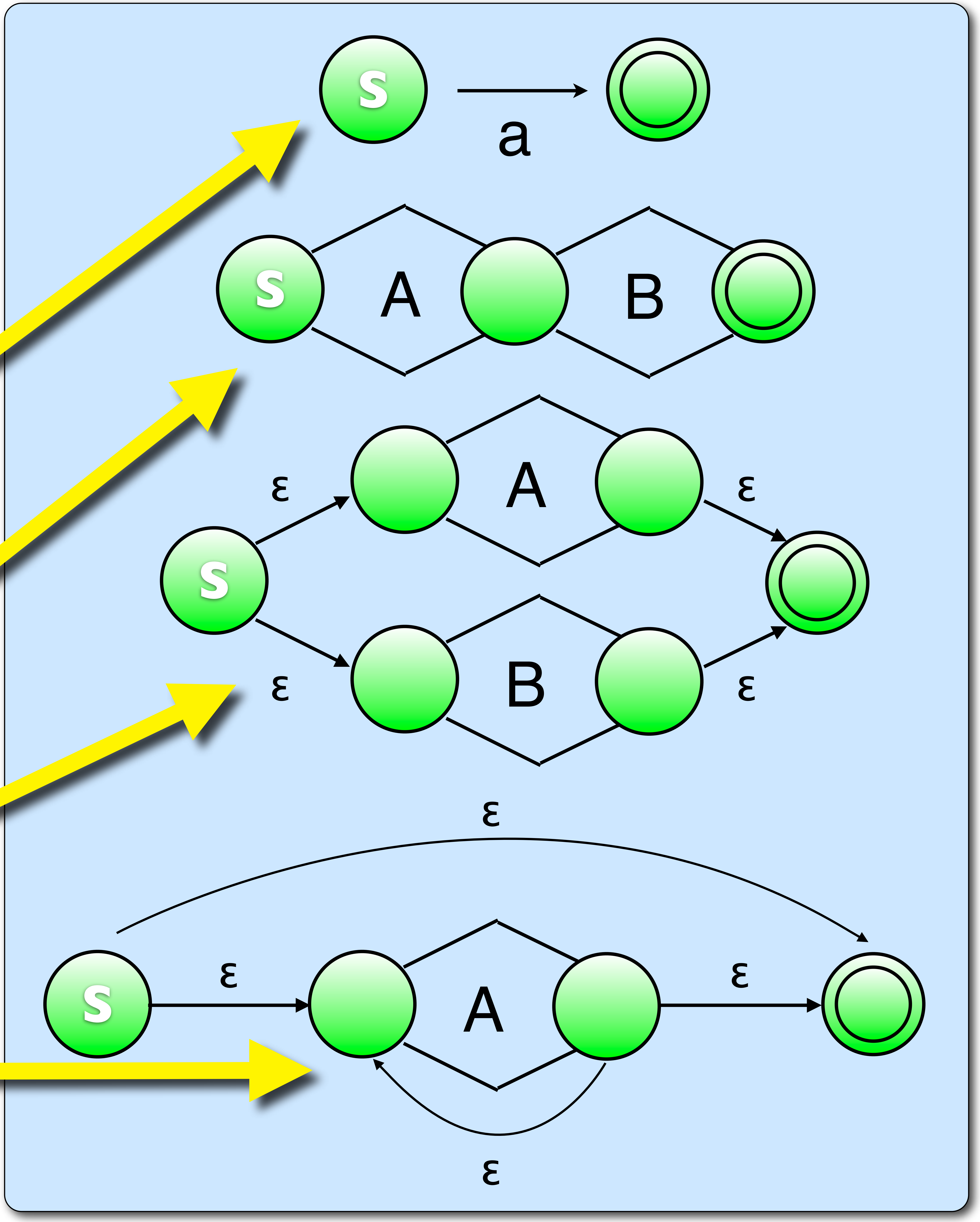
The Four NFA Construction Rules



Overview

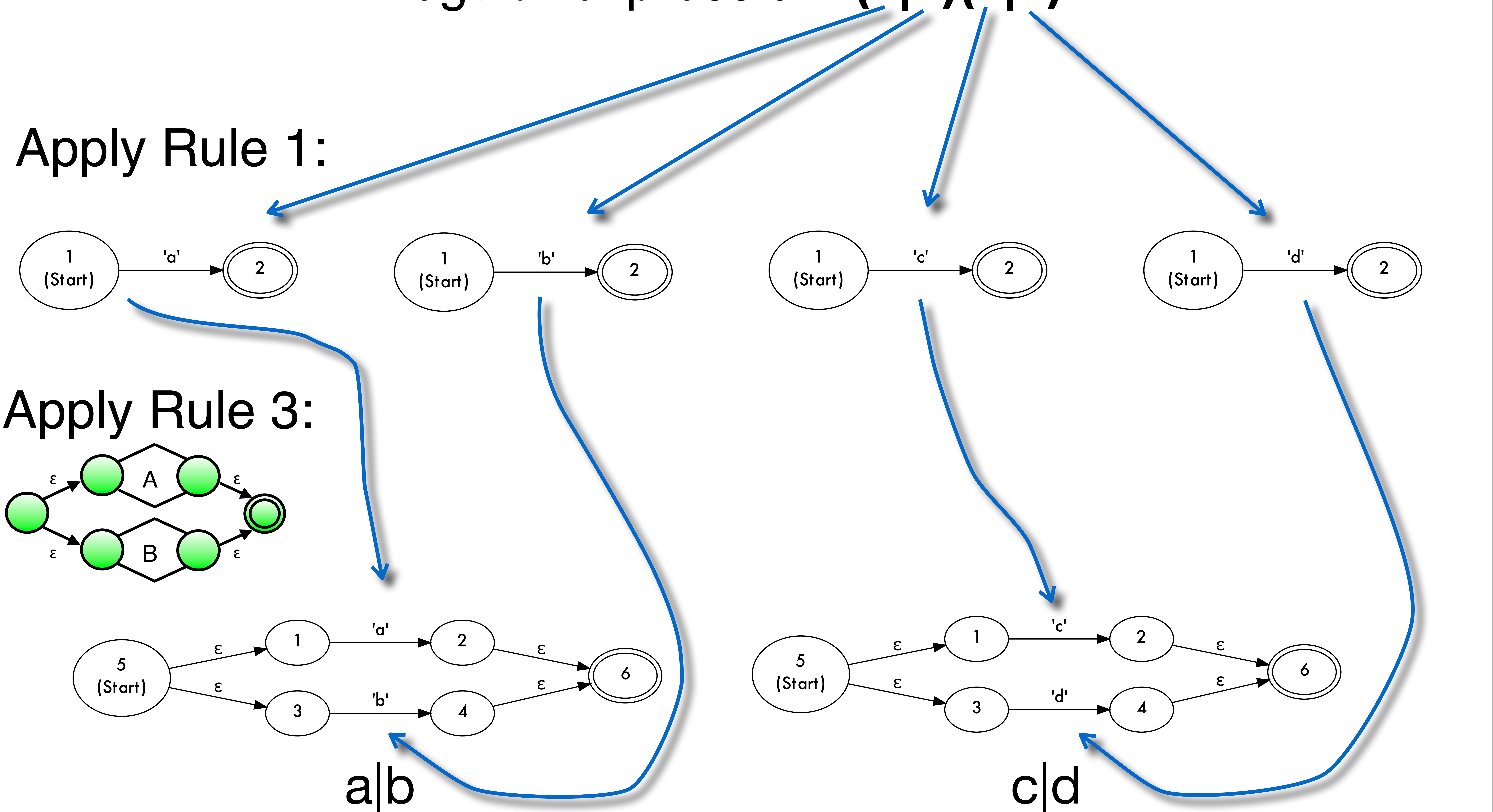
Four rules:

- **Create** two-state NFAs for individual symbols, e.g., **‘a’**.
- **Append** consecutive NFAs, e.g., **AB**.
- **Alternate** choices in parallel, e.g., **A|B**.
- Repeat **Kleene Star**, e.g., **A***.



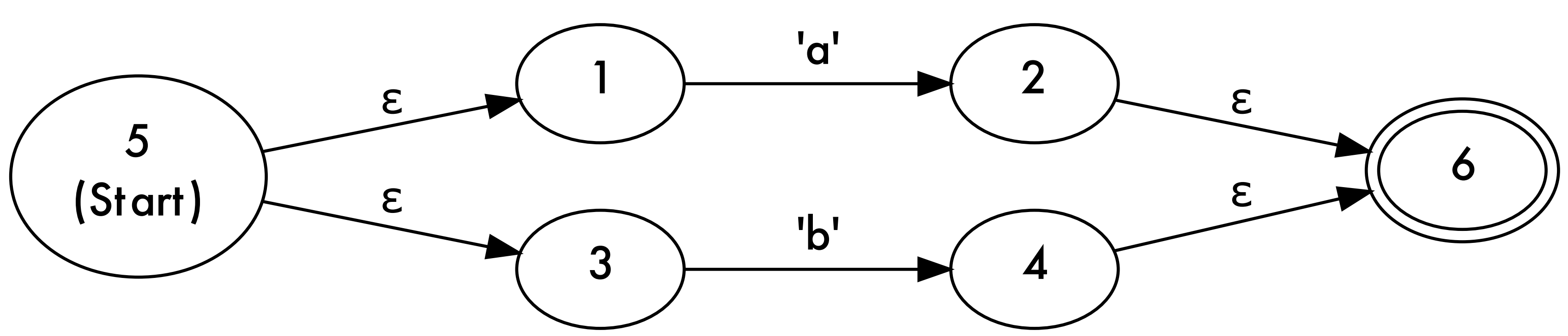
NFA Construction Example

Regular expression: **(a|b)(c|d)e***

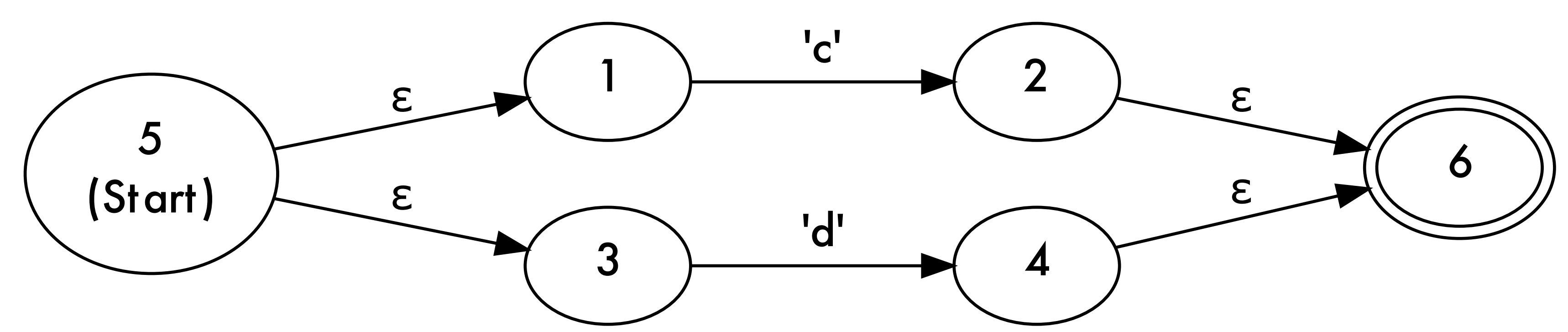


NFA Construction Example

Regular expression: **(a|b)(c|d)e***

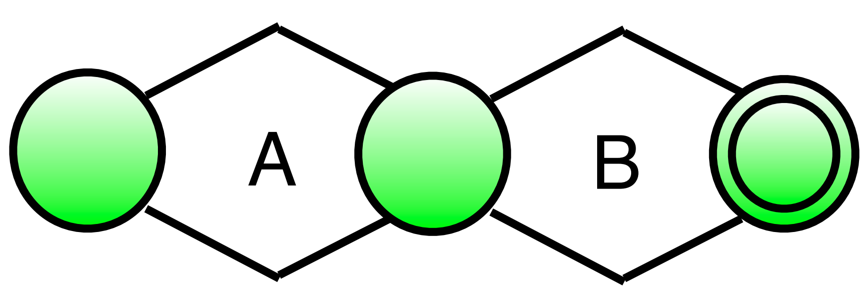


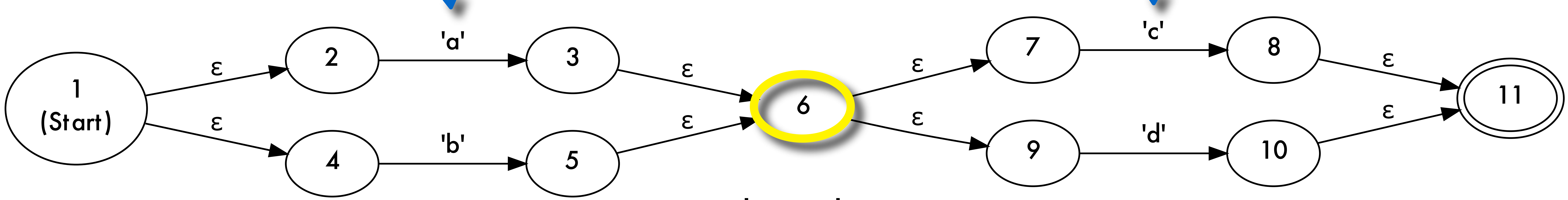
a|b



c|d

Apply Rule 2:

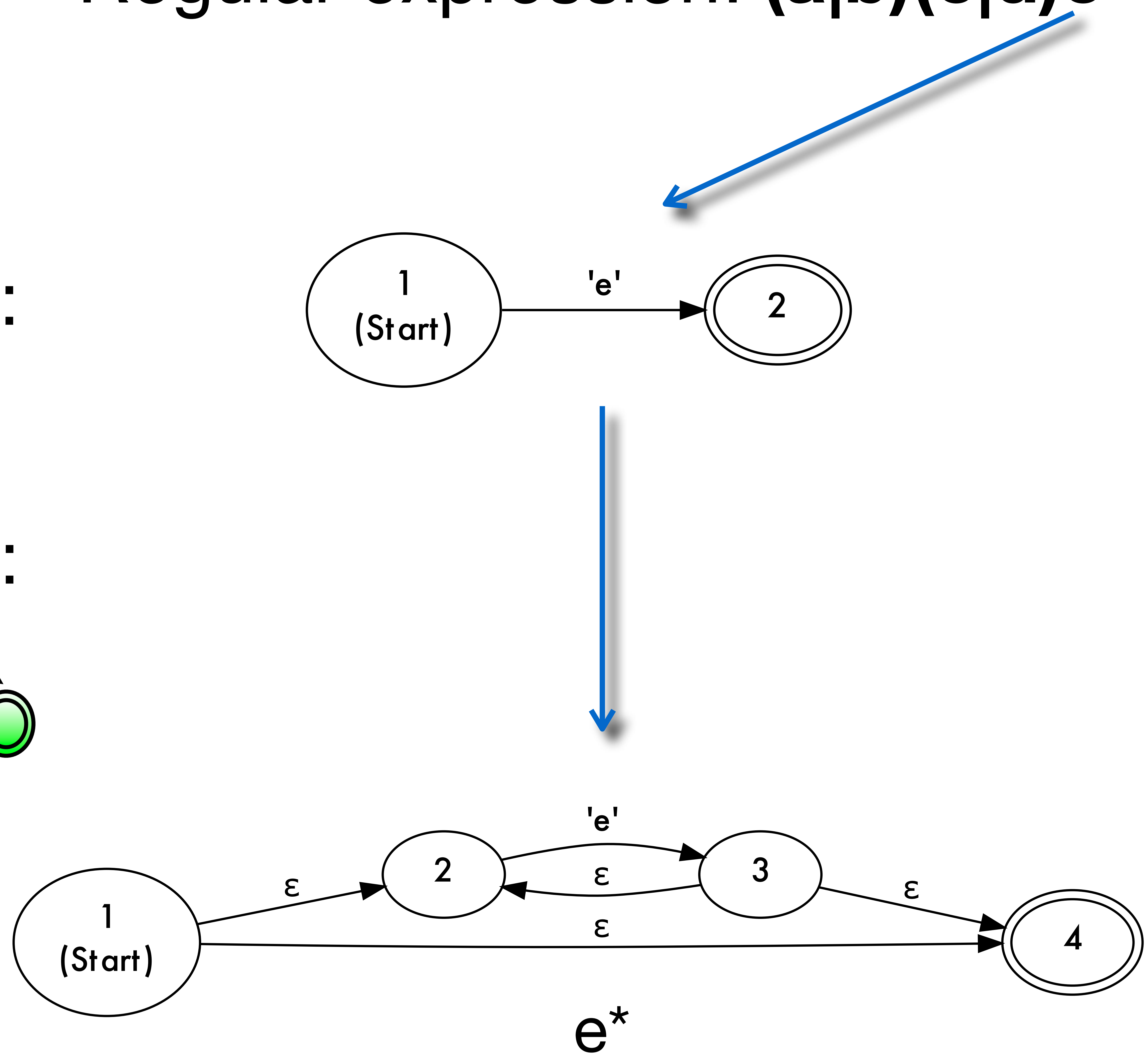




(a|b)(c|d)

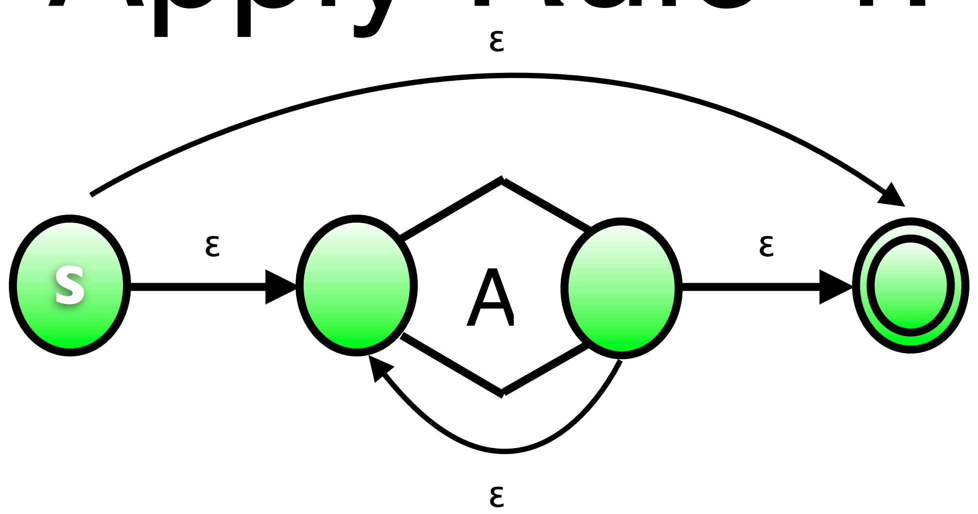
NFA Construction Example

Regular expression: **(a|b)(c|d)e***



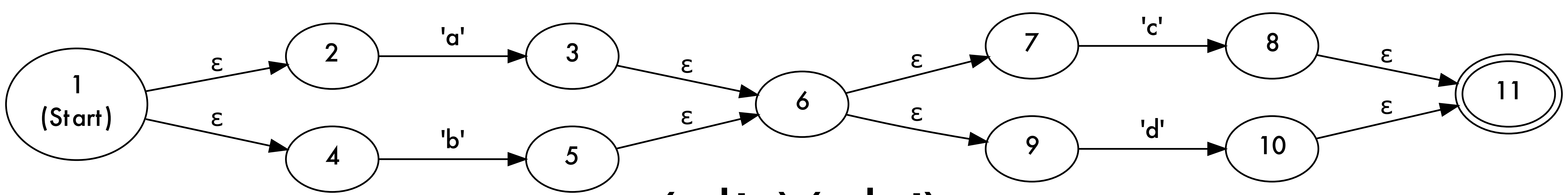
Apply Rule 1:

Apply Rule 4:

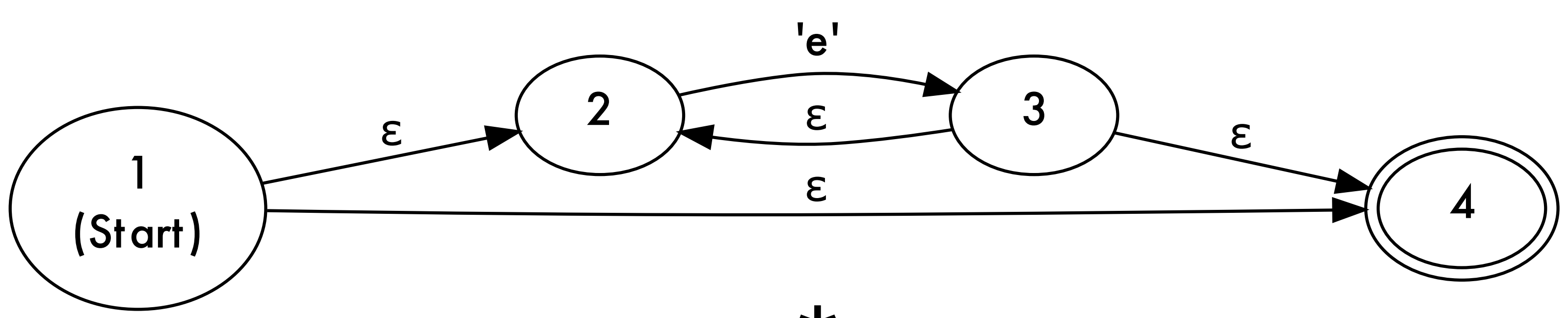


NFA Construction Example

Regular expression: **(a|b)(c|d)e***

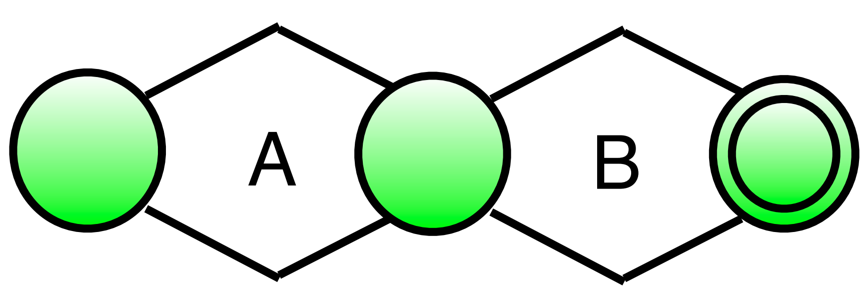


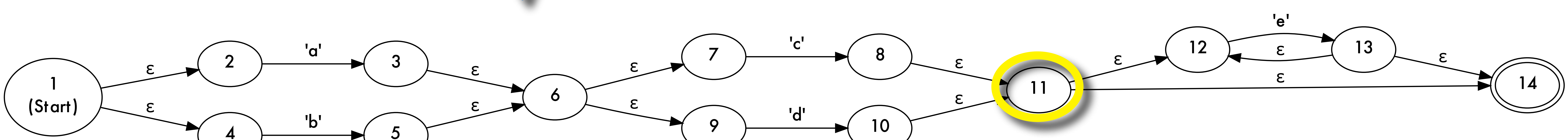
(a|b)(c|d)



e*

Apply Rule 2:





(a|b)(c|d)e*

Step 2: NFA → DFA

Simulating NFA requires exploration of all paths.

- Either in **parallel** (memory consumption!).
- Or with **backtracking** (large trees!).
- Both are **impractical**.

Instead, we derive a DFA that encodes all possible paths.

- Instead of doing a **specific parallel search** each time that we simulate the NFA, we do it **only once in general**.

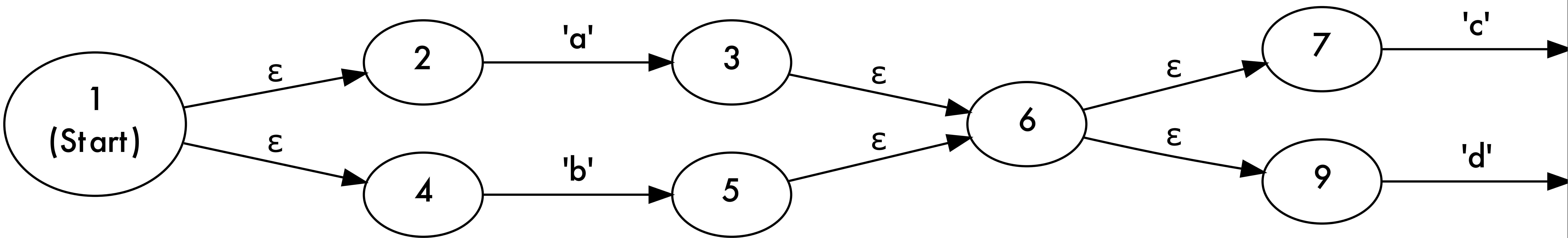
Key idea: for each input character, find sets of NFA states that can be reached.

- These are the states that a parallel search would explore.
- Create a DFA state + transitions for each such set.
- **Final states**: a DFA state is a final state if its corresponding set of NFA states **contains at least one final NFA state**.

```
NFA-to-DFA-CONVERSION:
    todo: stack of sets of NFA states.

    push {NFA start state and all epsilon-reachable states} onto todo

    while (todo is not empty):
      curNFA: set of NFA states
      curDFA: a DFA state

      curNFA = todo.pop
      mark curNFA as done

      curDFA = find or create DFA state corresponding to curNFA

      reachableNFA: set of NFA states
      reachableDFA: a DFA state

      for each symbol x for which at least one state in curNFA has a transition:
        reachableNFA = find each state that is reachable from a state in curNFA
                       via one x transition and any number of epsilon transitions

        if (reachableNFA is not empty and not done):
          push reachableNFA onto todo
        reachableDFA = find or create DFA state corresponding to reachableNFA
        add transition on x from curDFA to reachableDFA
      end for
    end while
```

DFA Conversion Example

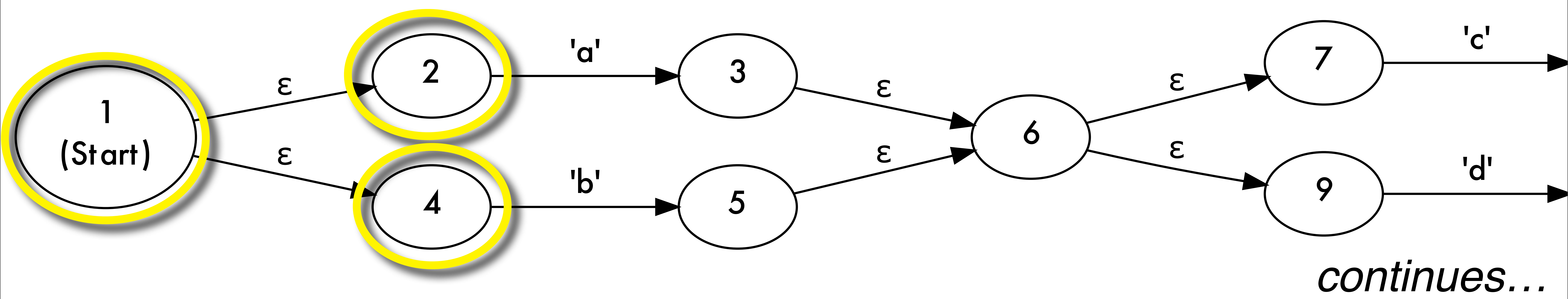
Regular expression: **(a|b)(c|d)e***



continues…

DFA Conversion Example

Regular expression: **(a|b)(c|d)e***

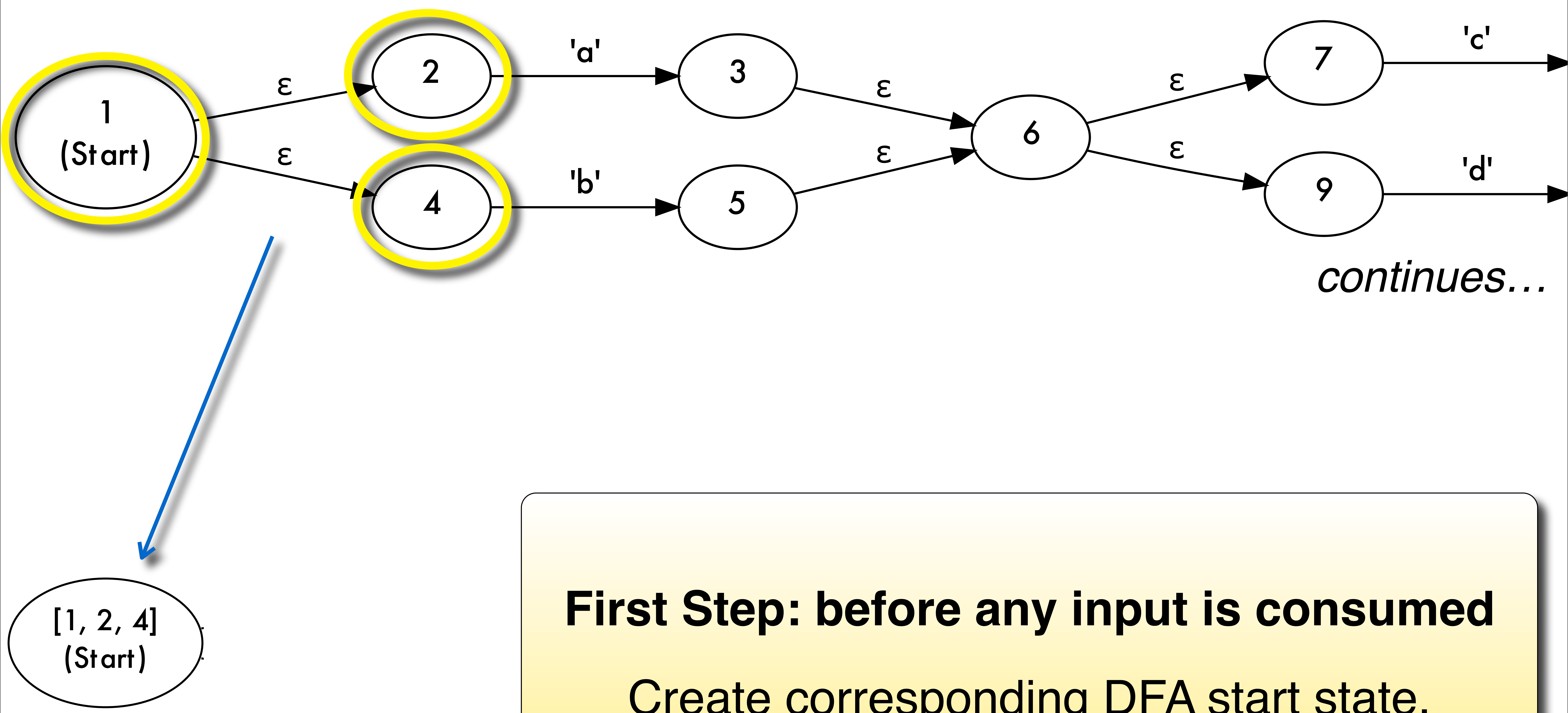


First Step: before any input is consumed

Find all states that are **reachable** from the start state **via epsilon transitions**.

DFA Conversion Example

Regular expression: **(a|b)(c|d)e***

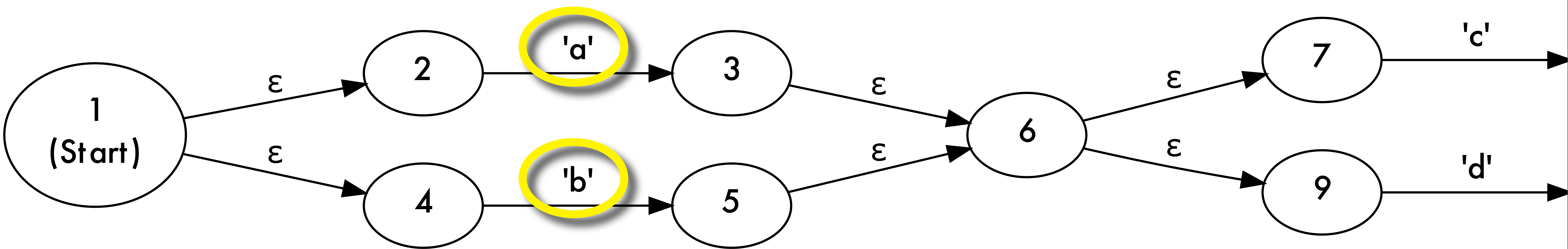


First Step: before any input is consumed

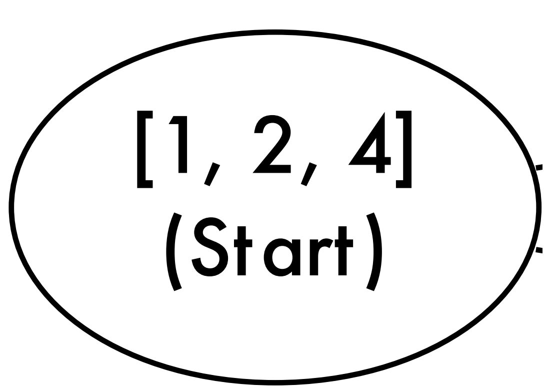
Create corresponding DFA start state.

DFA Conversion Example

Regular expression: **(a|b)(c|d)e***



continues…

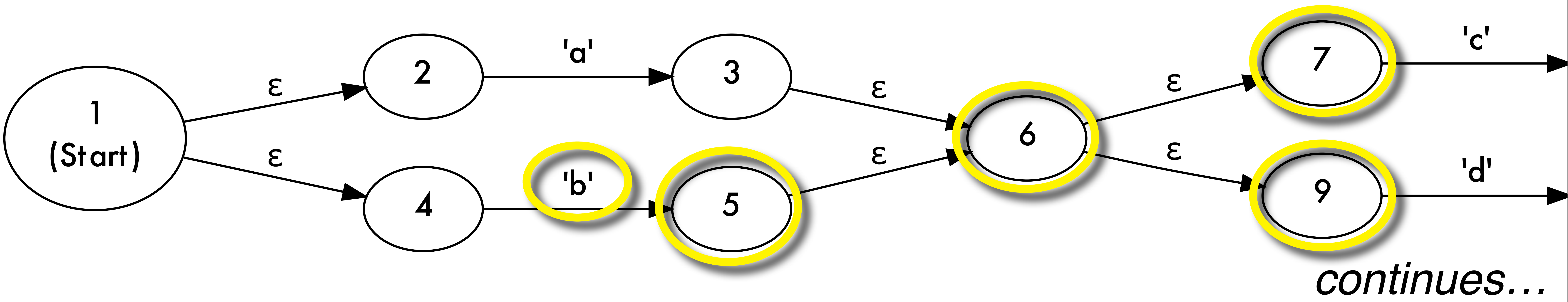


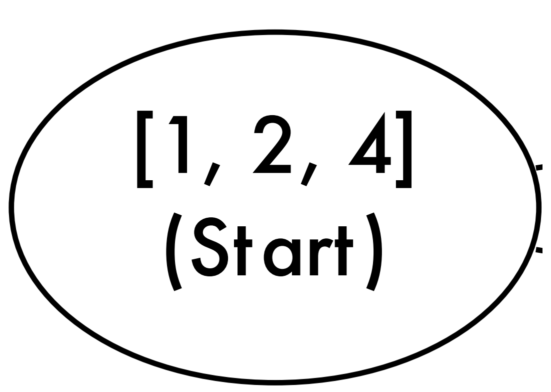
Next: find all input characters for which transitions in start set exist.

'a' and 'b' in this case.

DFA Conversion Example

Regular expression: **(a|b)(c|d)e***



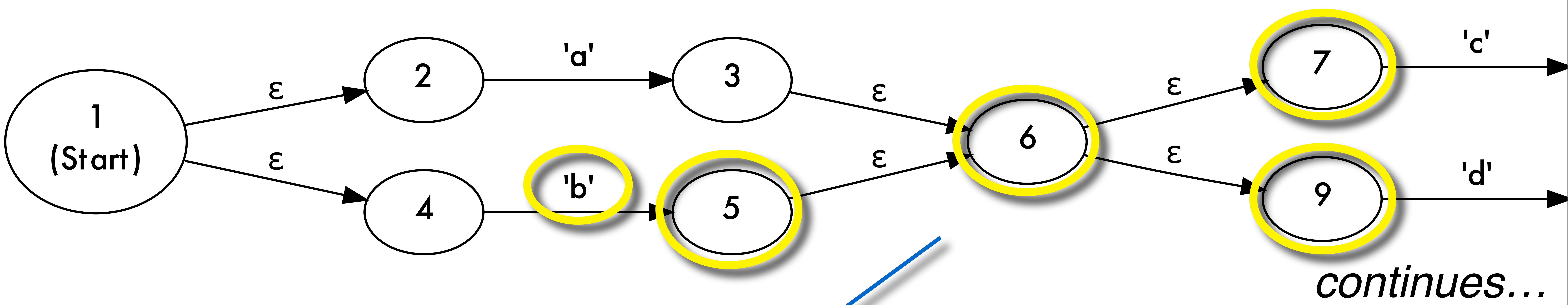


For each such input character, determine the set of reachable states (including epsilon transitions).

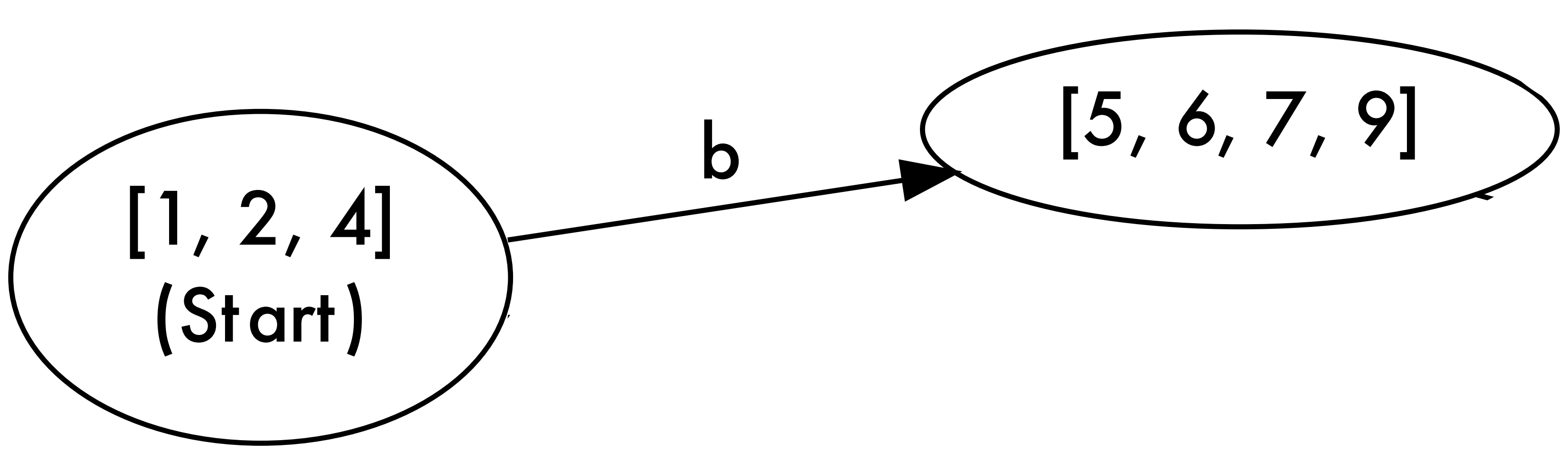
On an ‘b’, NFA can reach states 5,6,7, and 9.

DFA Conversion Example

Regular expression: **(a|b)(c|d)e***

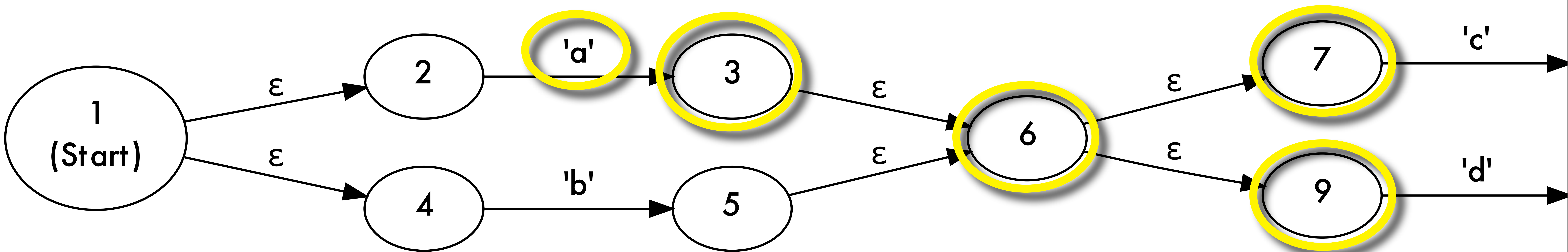


Create DFA states for each **distinct** reachable set of states.

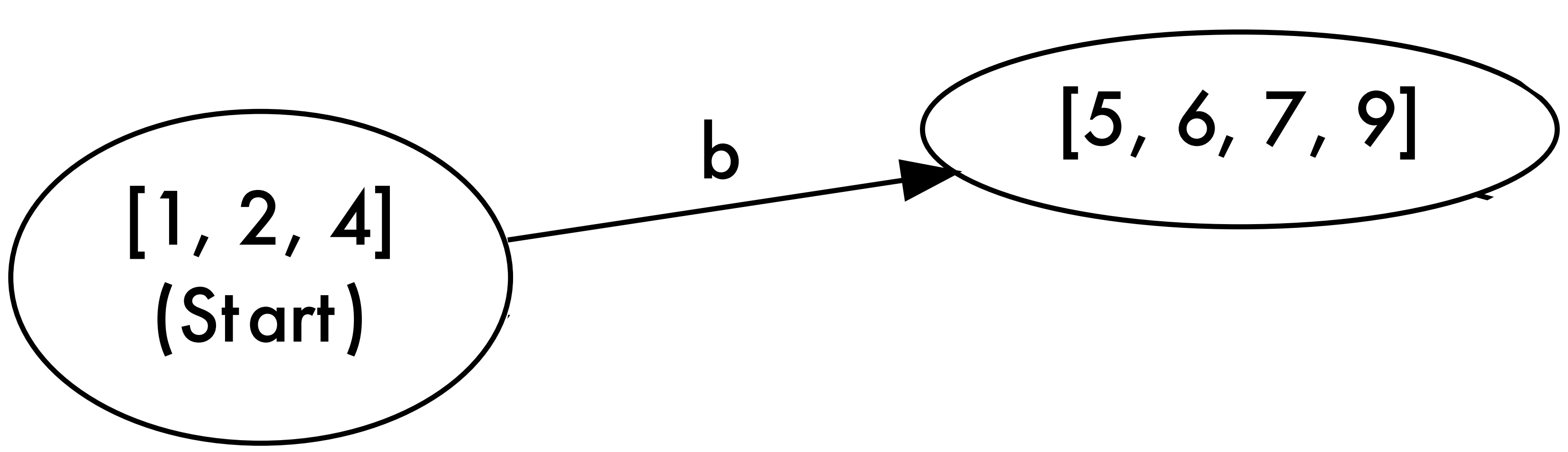


DFA Conversion Example

Regular expression: **(a|b)(c|d)e***



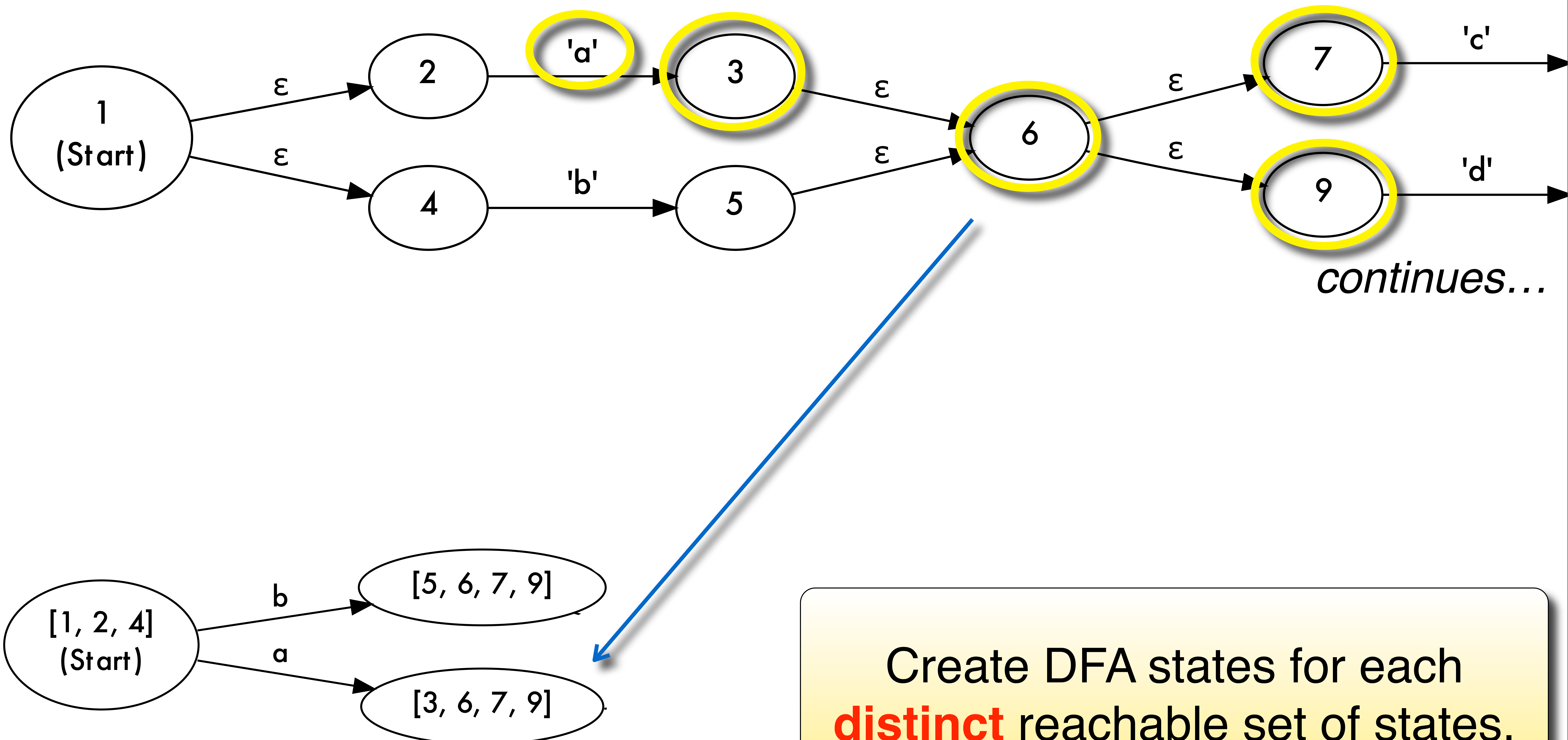
continues…



On an 'a', NFA can reach states 3,6,7, and 9.

DFA Conversion Example

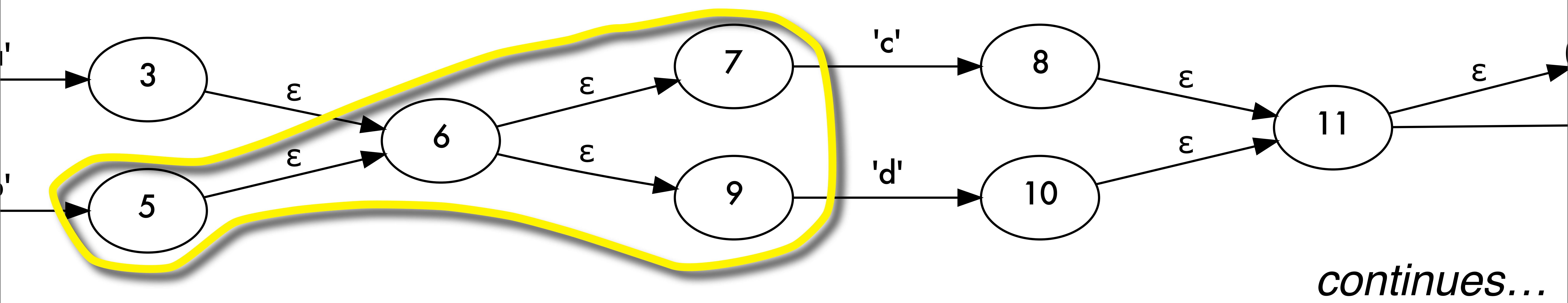
Regular expression: **(a|b)(c|d)e***



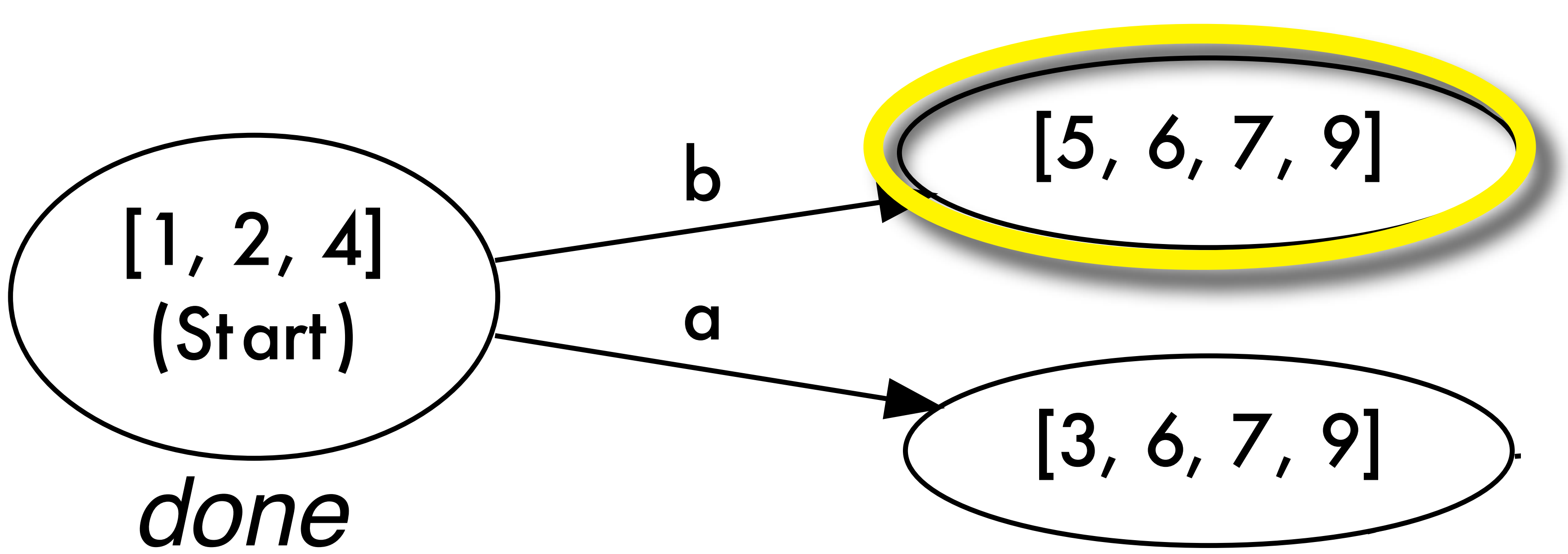
Create DFA states for each **distinct** reachable set of states.

DFA Conversion Example

Regular expression: **(a|b)(c|d)e***

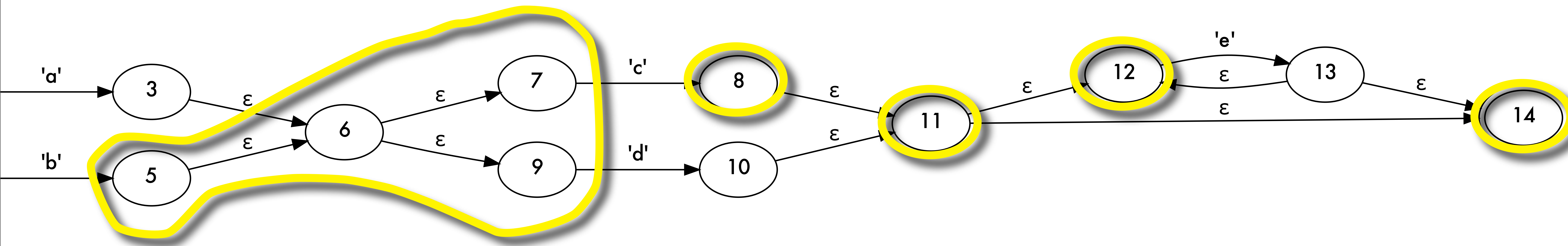


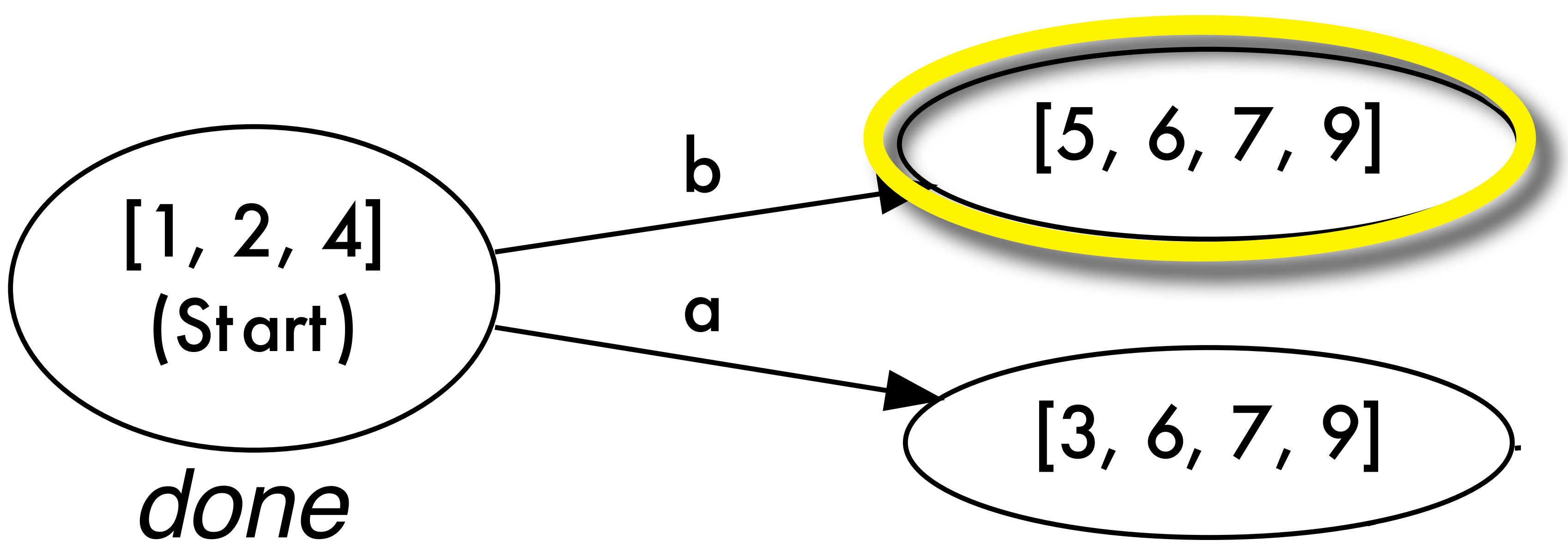
Repeat process for each newly-discovered set of states.



DFA Conversion Example

Regular expression: **(a|b)(c|d)e***

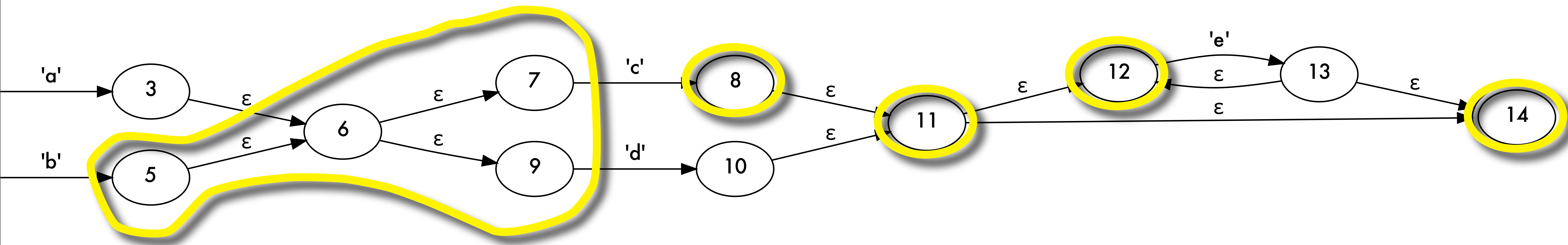




Reachable states:
on a 'c': 8,11,12,14

DFA Conversion Example

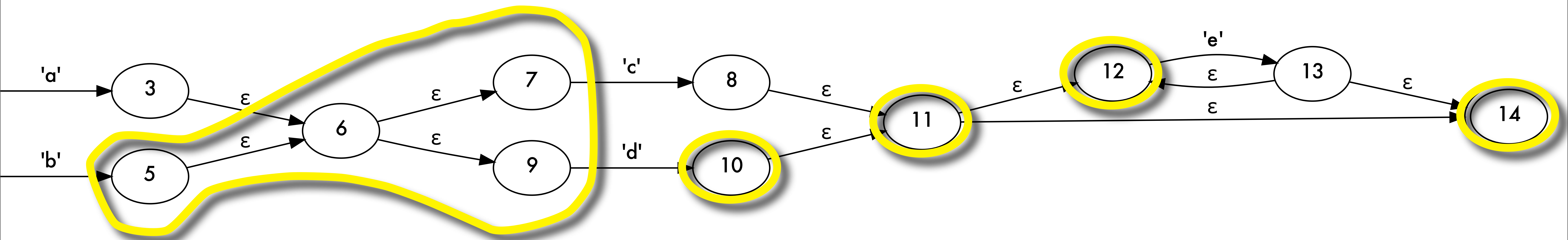
Regular expression: **(a|b)(c|d)e***



Create state and transitions for the set of reachable states.

DFA Conversion Example

Regular expression: **(a|b)(c|d)e***

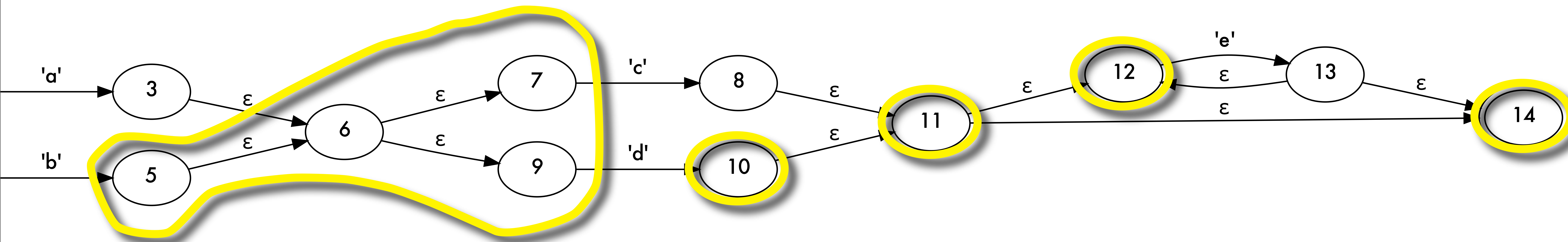


Reachable states:
on a ‘d’: 10,11,12,14

done

DFA Conversion Example

Regular expression: **(a|b)(c|d)e***

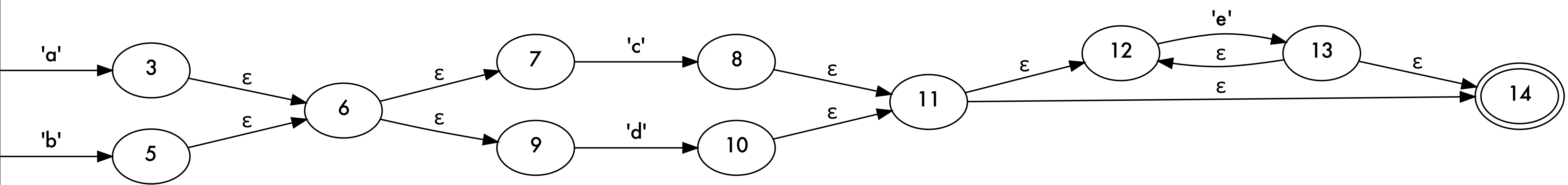


Reachable states:
on a ‘d’: 10,11,12,14

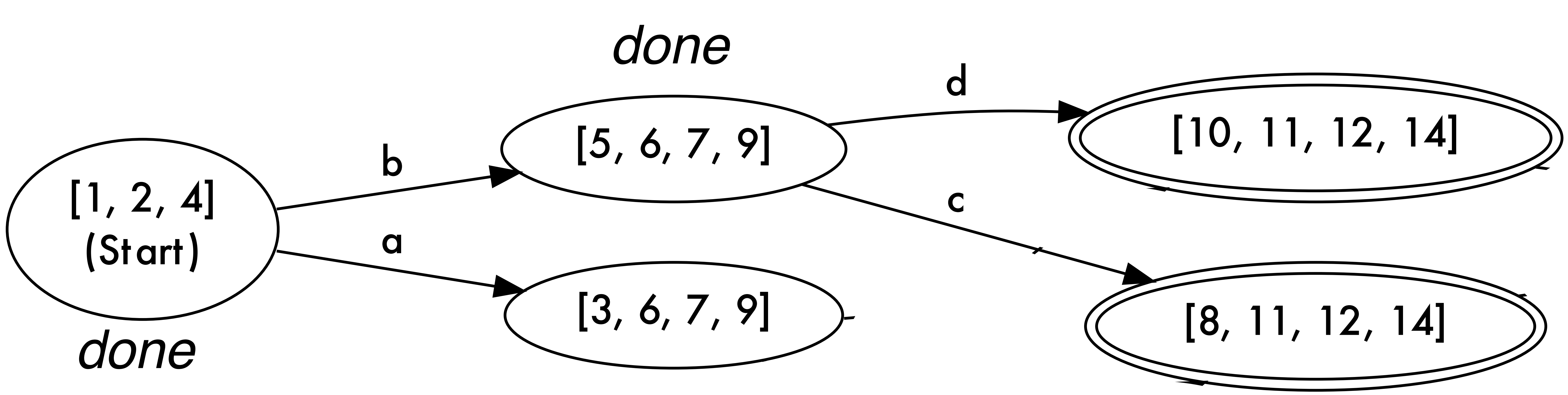
Create state and transitions for the set of reachable states.

DFA Conversion Example

Regular expression: **(a|b)(c|d)e***

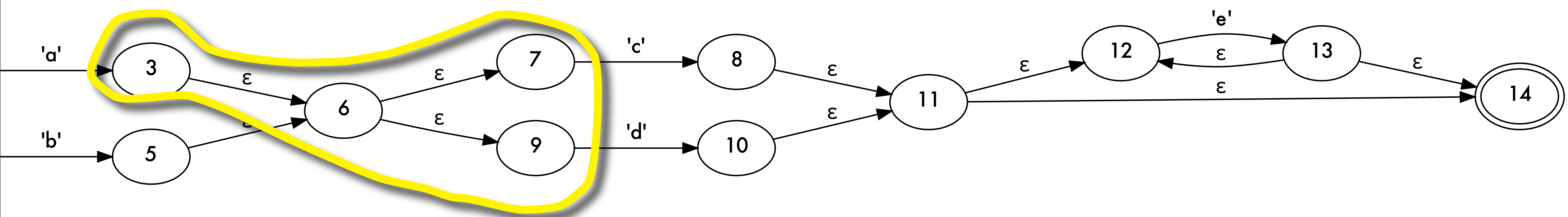


Note: both new DFA states are **final states** because their corresponding sets include NFA state 14, which is a final state.

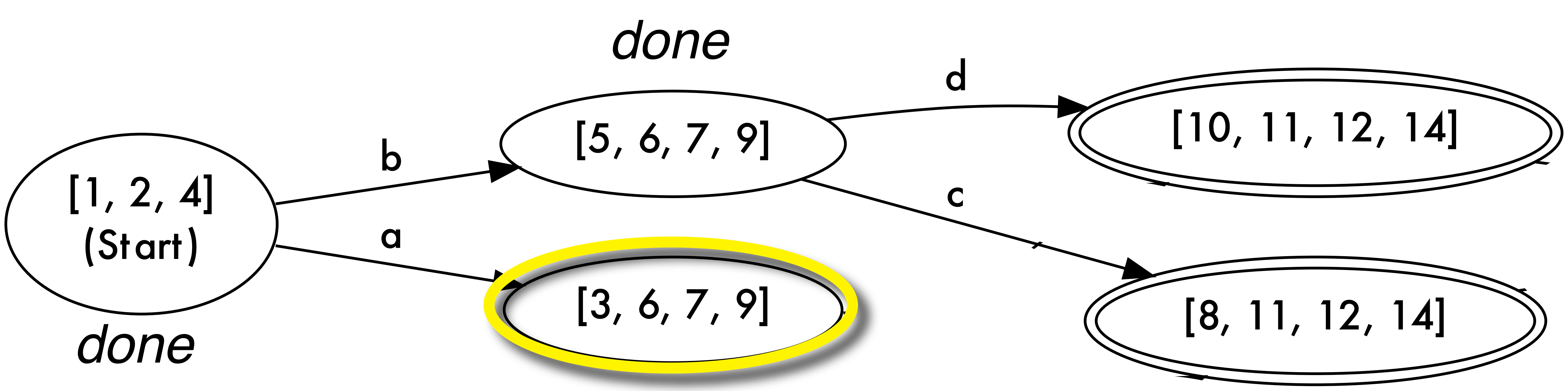


DFA Conversion Example

Regular expression: **(a|b)(c|d)e***

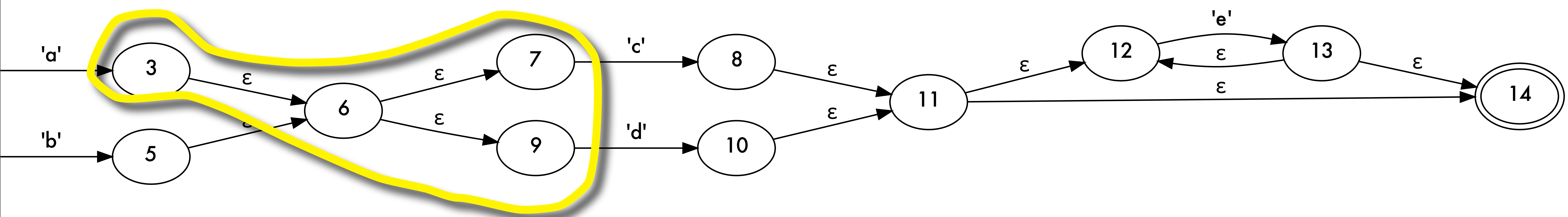


Repeat process for State [3, 6, 7, 9].



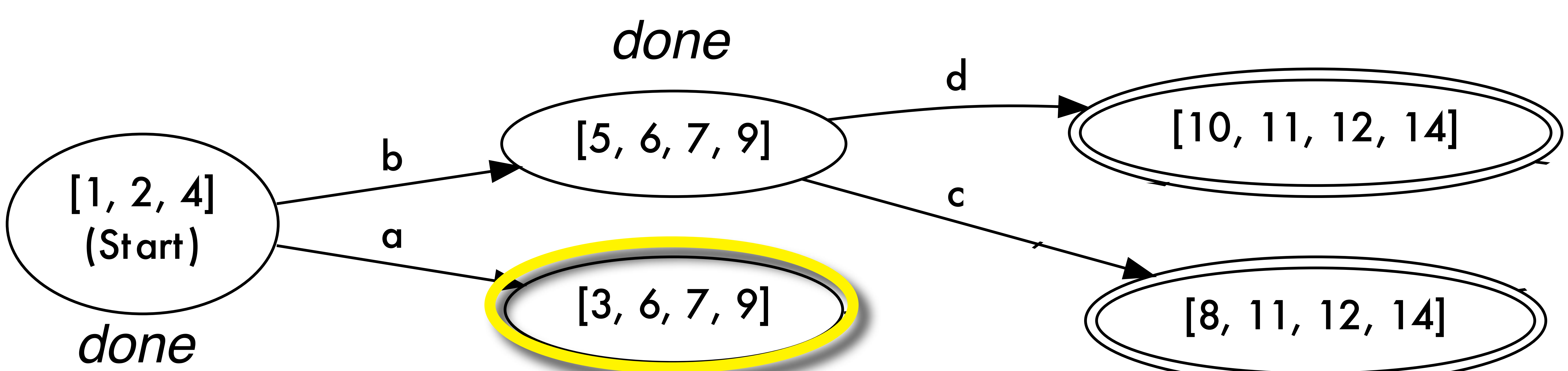
DFA Conversion Example

Regular expression: **(a|b)(c|d)e***



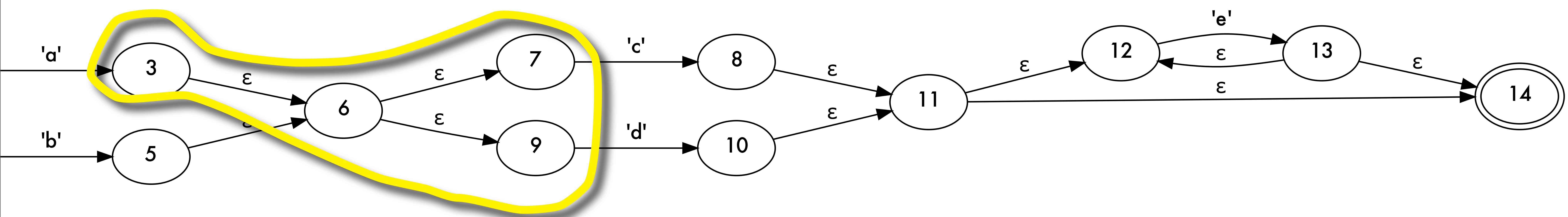
Reachable states:
on a ‘d’: 10,11,12,14

Reachable states:
on a ‘c’: 8,11,12,14

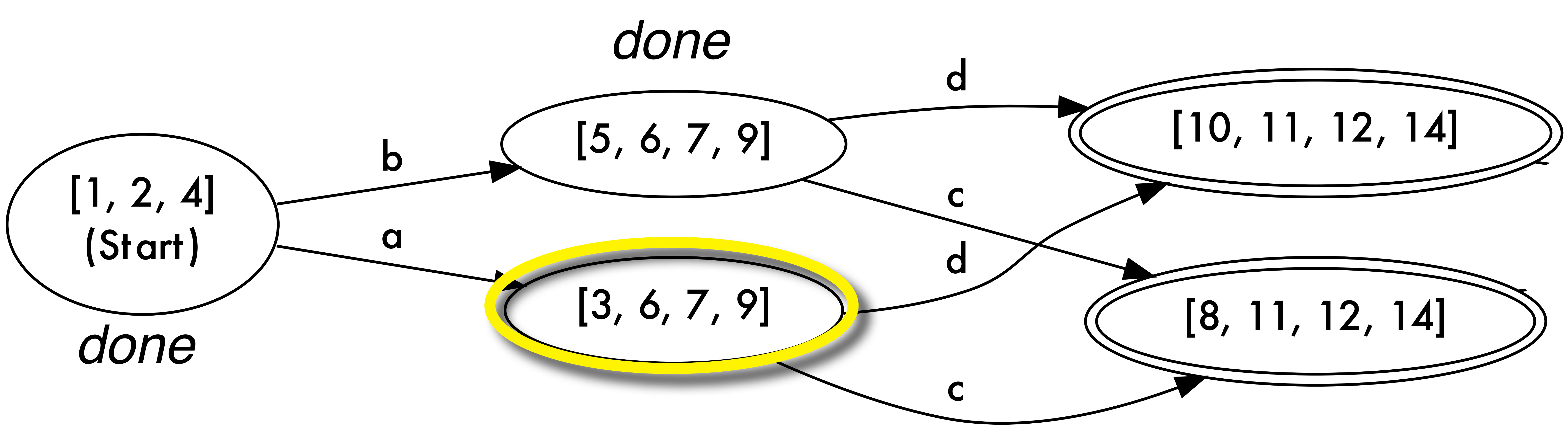


DFA Conversion Example

Regular expression: **(a|b)(c|d)e***

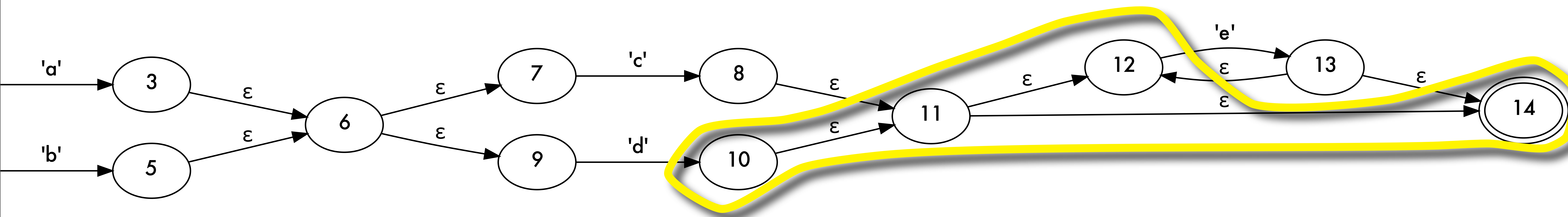


There **already exist** DFA states corresponding to those sets!
Just **add transitions** to these states.



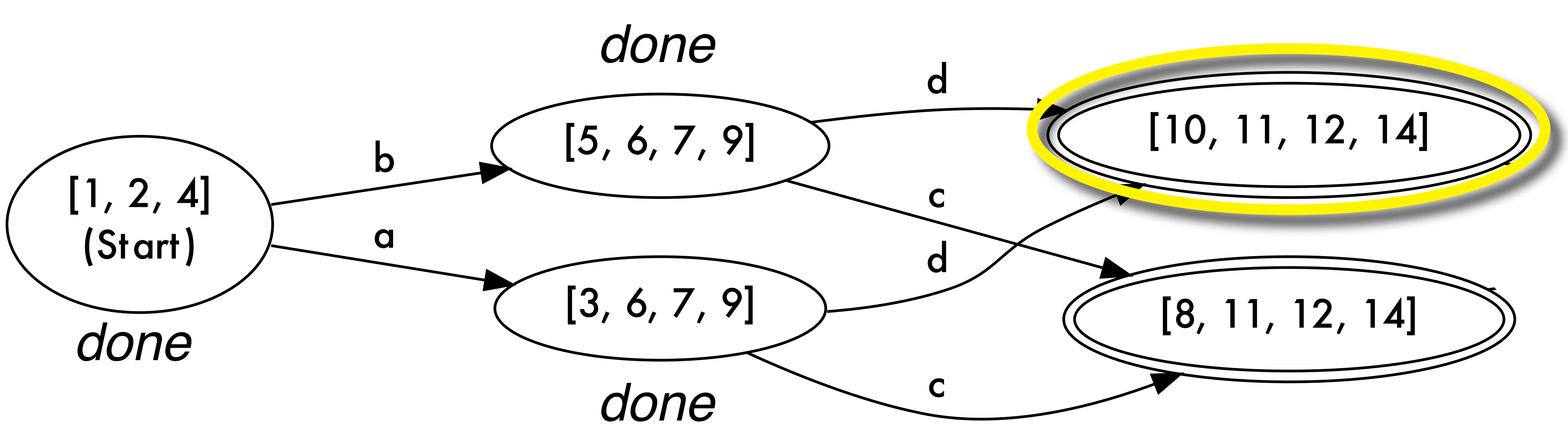
DFA Conversion Example

Regular expression: **(a|b)(c|d)e***



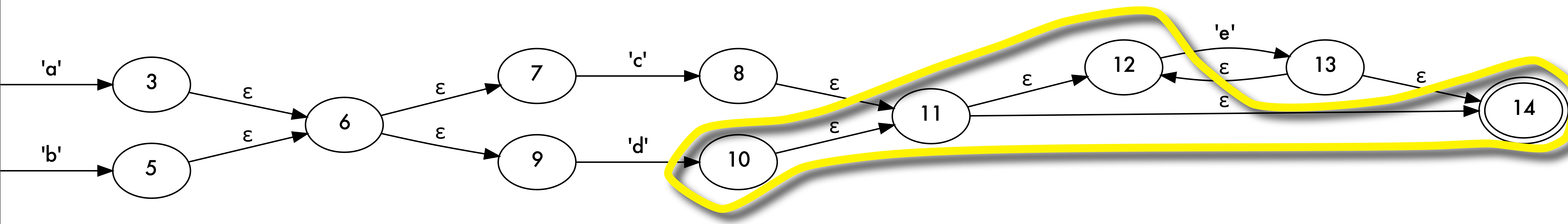
Repeat process for State [10, 11, 12, 14].

Reachable states: on an ‘e’: 12, 13, 14

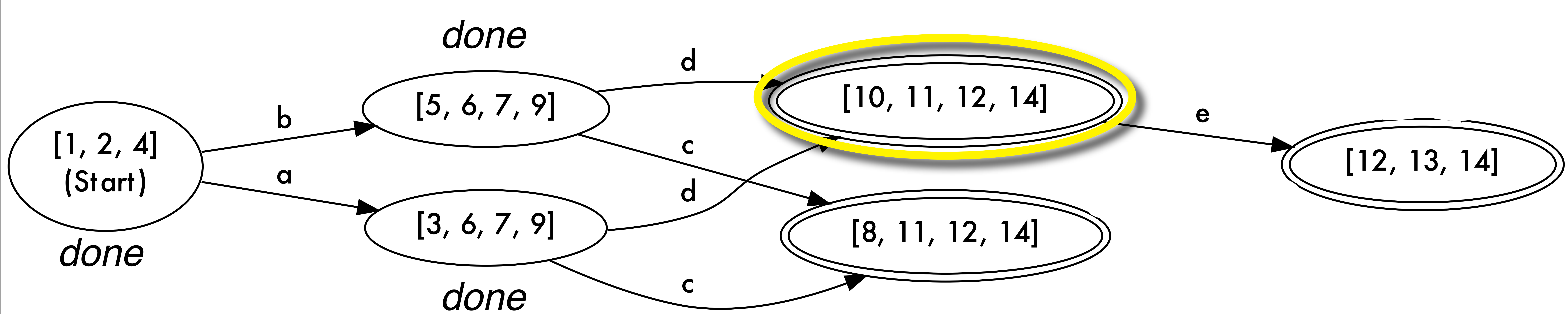


DFA Conversion Example

Regular expression: **(a|b)(c|d)e***

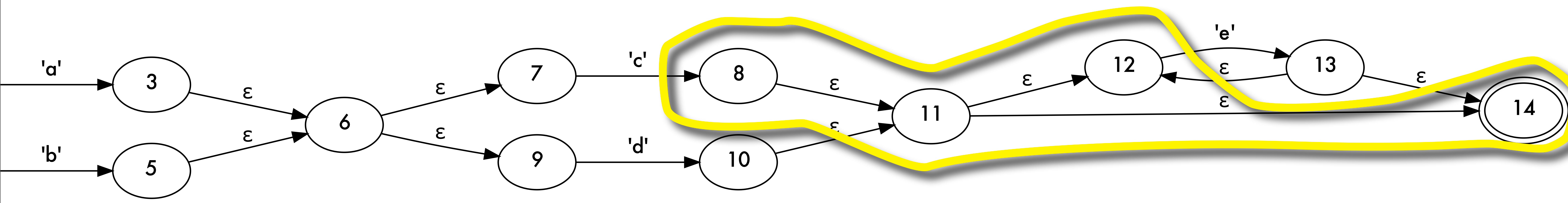


Create state and transitions for the set of reachable states.



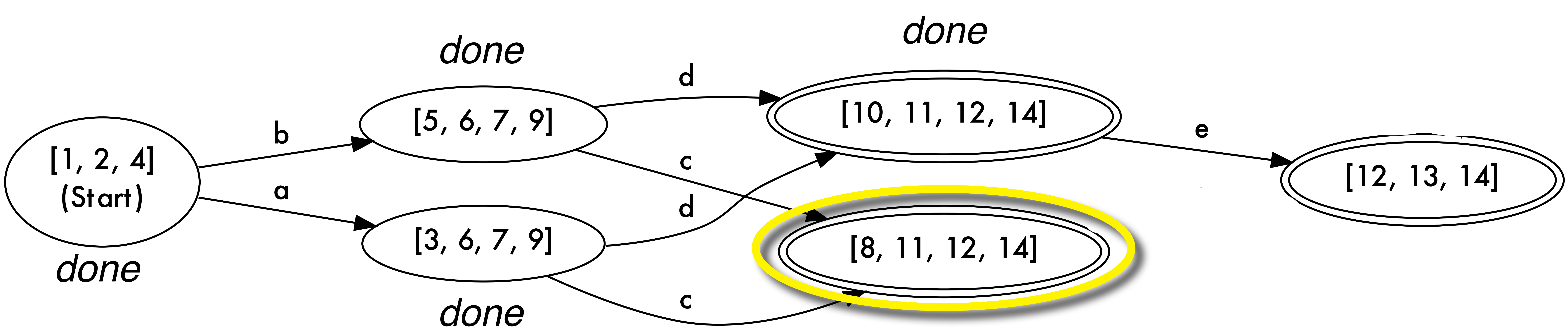
DFA Conversion Example

Regular expression: **(a|b)(c|d)e***



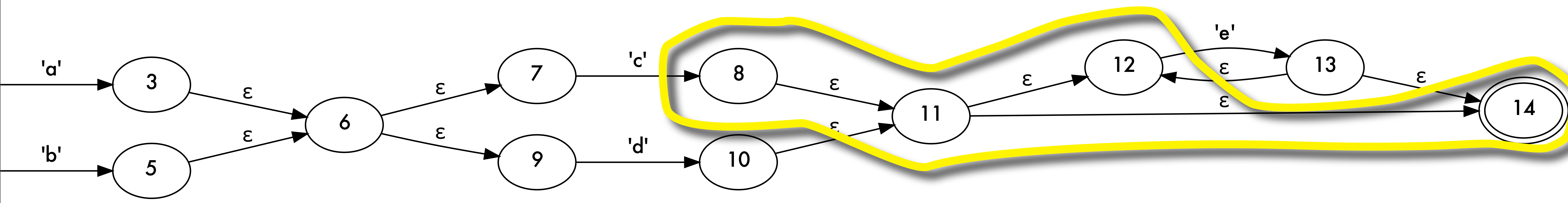
Repeat process for State [8, 11, 12, 14].

Reachable states: on an ‘e’: 12, 13, 14

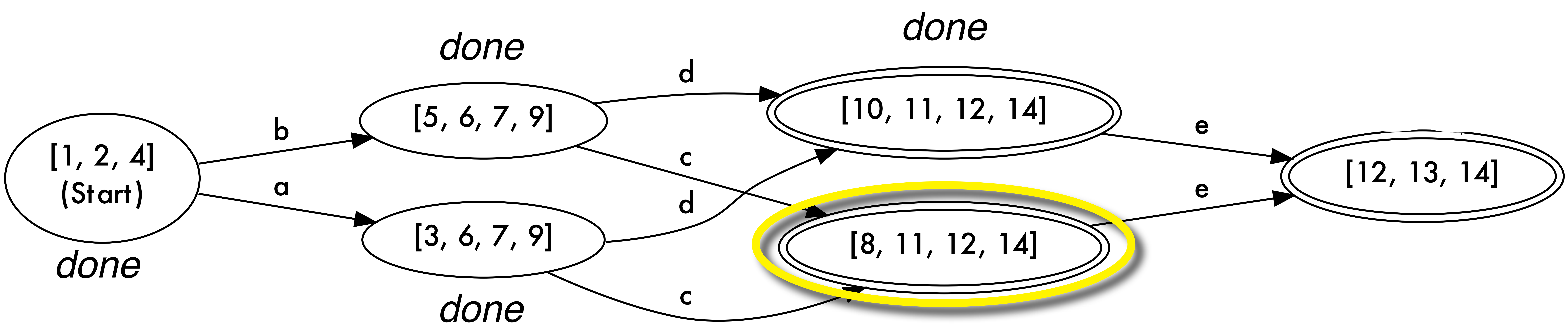


DFA Conversion Example

Regular expression: **(a|b)(c|d)e***

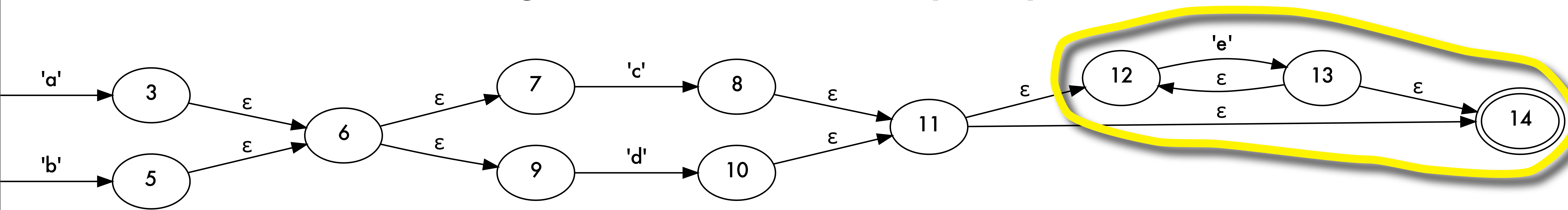


State already exists.
Just create transition.



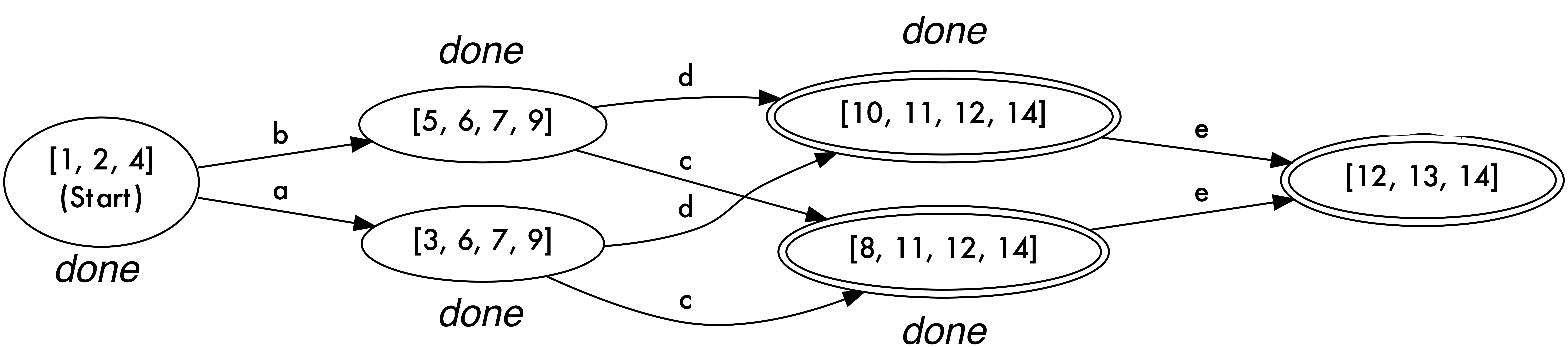
DFA Conversion Example

Regular expression: **(a|b)(c|d)e***



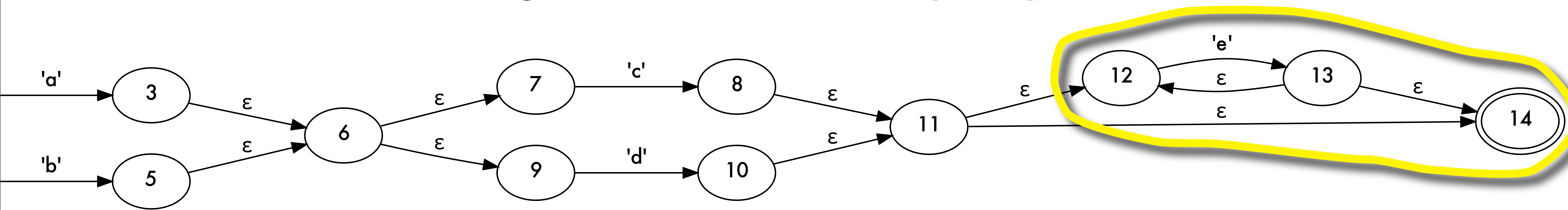
Repeat process for State [12, 13, 14].

Reachable states:
on an ‘e’: 12, 13, 14 (itself!)

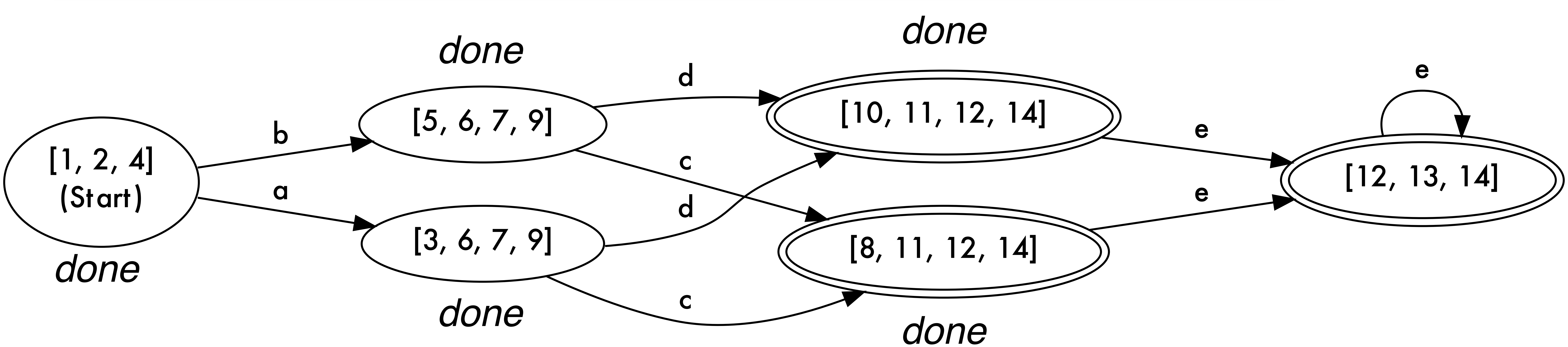


DFA Conversion Example

Regular expression: **(a|b)(c|d)e***

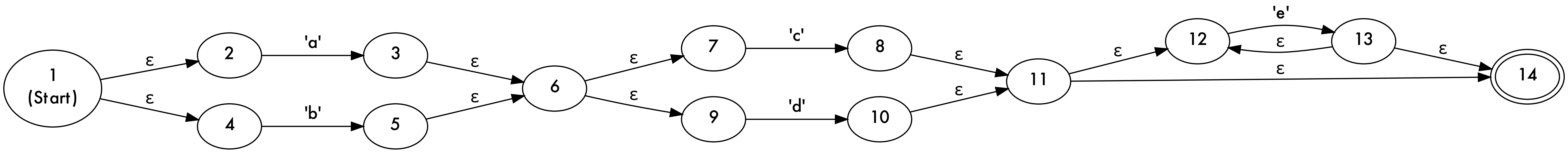


There is no "escape" from the set of states [12, 13, 14] on an 'e'. Thus, **create a self-loop**.

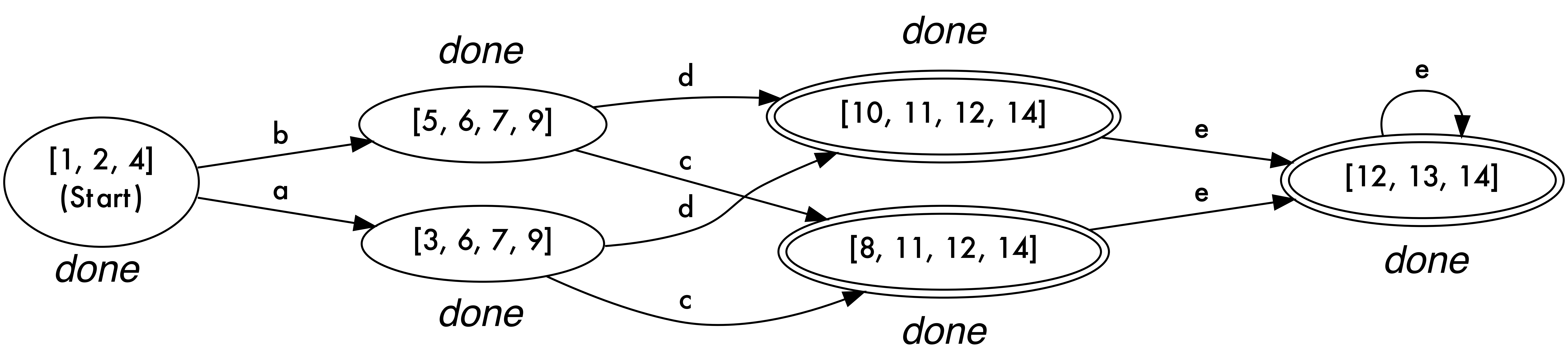


DFA Conversion Example

Regular expression: **(a|b)(c|d)e***

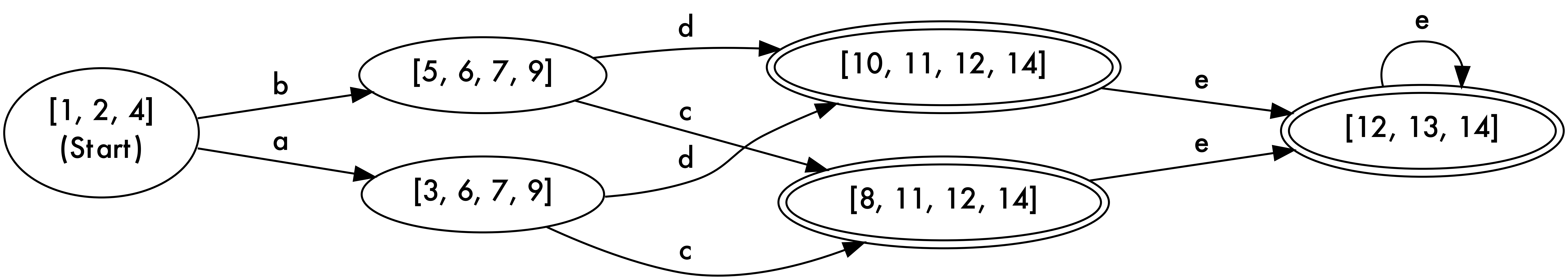


The result: an **equivalent** DFA!



NFA → DFA Conversion

- **Any NFA can be converted** into an equivalent DFA using this method.
- However, the number of states can increase **exponentially**.
- With **careful syntax design**, this problem can be avoided in practice.
- **Limitation**: resulting DFA is not necessarily optimal.

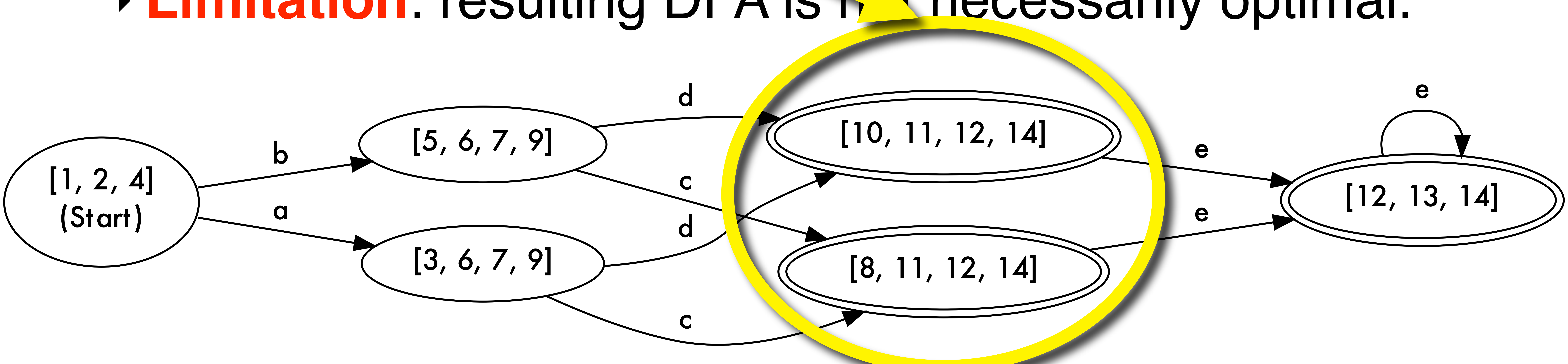


NFA → DFA Conversion

- **Any NFA can be converted** into an equivalent DFA usi
- How exp
- With avoided in practice.
- **Limitation**: resulting DFA is not necessarily optimal.

These two states are equivalent: for each input element, they both lead to the same state.

Thus, having two states is **unnecessary**.



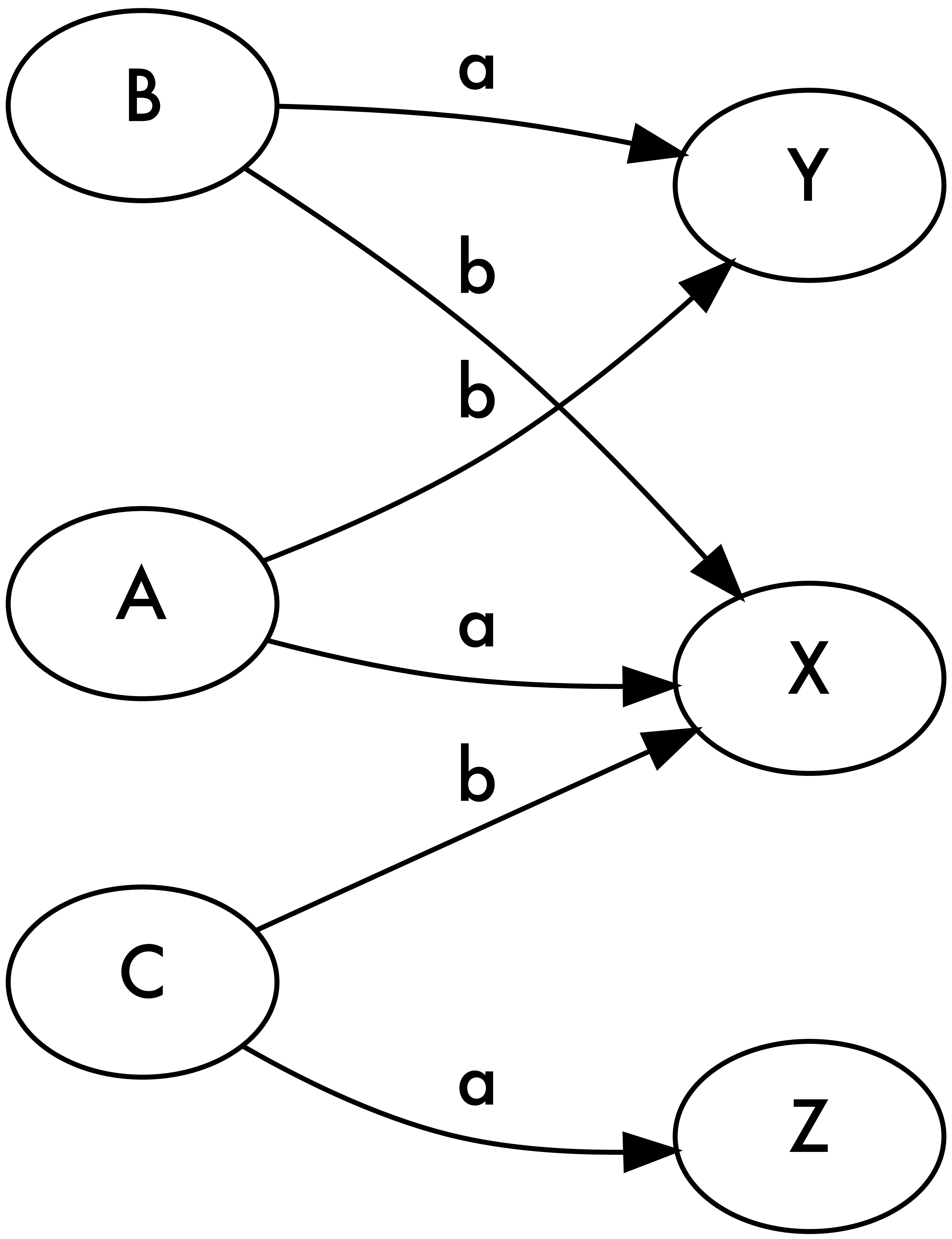
Step 3: DFA Minimization

Goal: obtain minimal DFA.

- For each RE, the minimal DFA is unique (ignoring simple renaming).
- DFA minimization: merge states that are equivalent.

Key idea: it's easier to split.

- Start with two partitions: final and non-final states.
- Repeatedly split partitions until all partitions contain only equivalent states.
- Two states ***S1***, ***S2*** are equivalent if all their transitions "**agree**," i.e., if there exists an input symbol ***x*** such that the DFA transitions (on input ***x***) to a state in partition ***P1*** if in ***S1*** and to state in partition ***P2*** if in ***S2*** and ***P1≠P2***, then ***S1*** and ***S2*** are **not equivalent**.



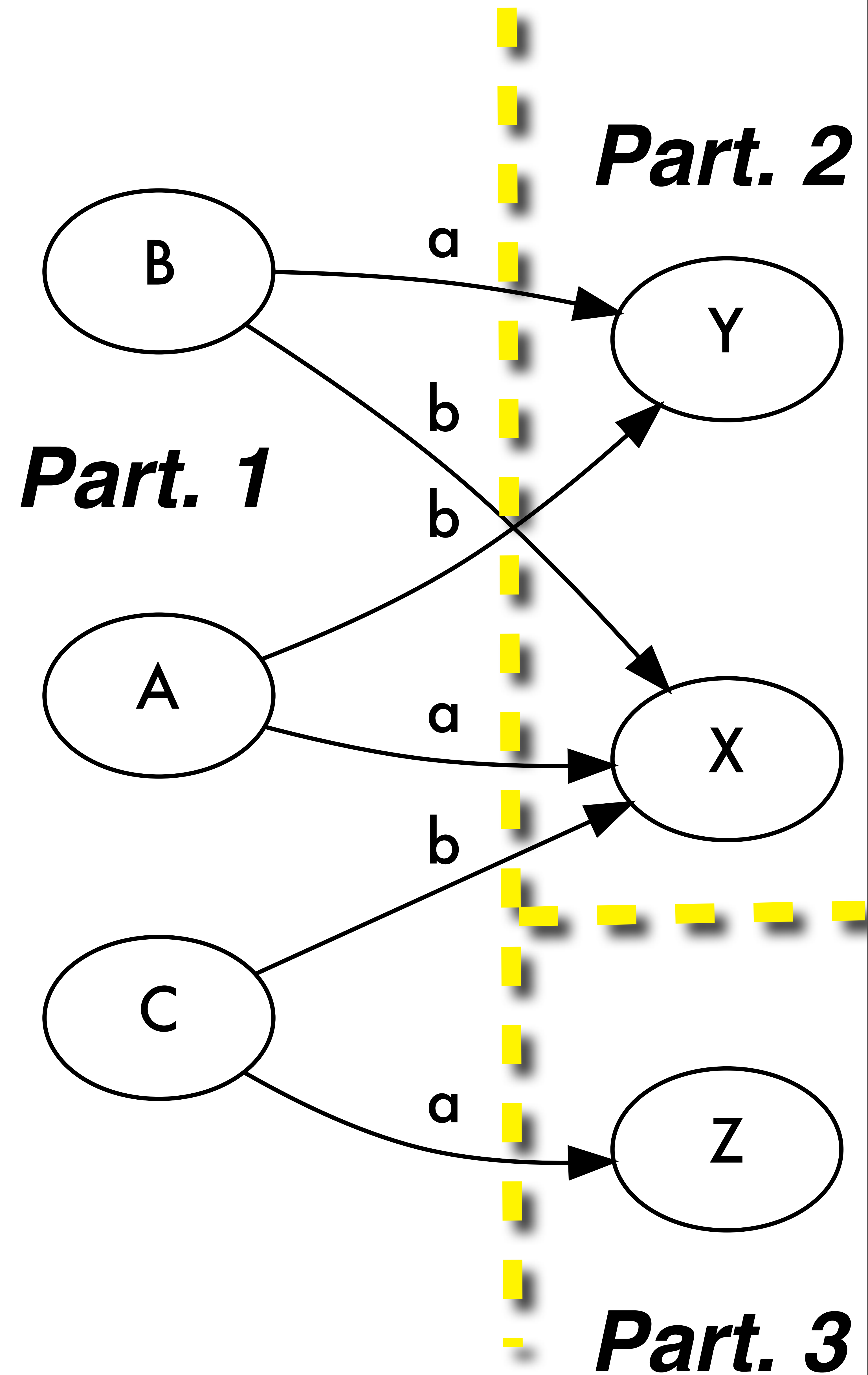
Step 3: DFA Minimization

Goal: obtain minimal DFA.

- For each RE, the minimal DFA is unique (ignoring simple renaming).
- DFA minimization: merge states that are equivalent.

Key idea: it's easier to split.

- Start with two partitions: final and non-final states.
- Repeatedly split partitions until all partitions contain only equivalent states.
- Two states ***S1***, ***S2*** are equivalent if all their transitions "**agree**," i.e., if there exists an input symbol ***x*** such that the DFA transitions (on input ***x***) to a state in partition ***P1*** if in ***S1*** and to state in partition ***P2*** if in ***S2*** and ***P1≠P2***, then ***S1*** and ***S2*** are **not equivalent**.



Step 3: DFA Minimization

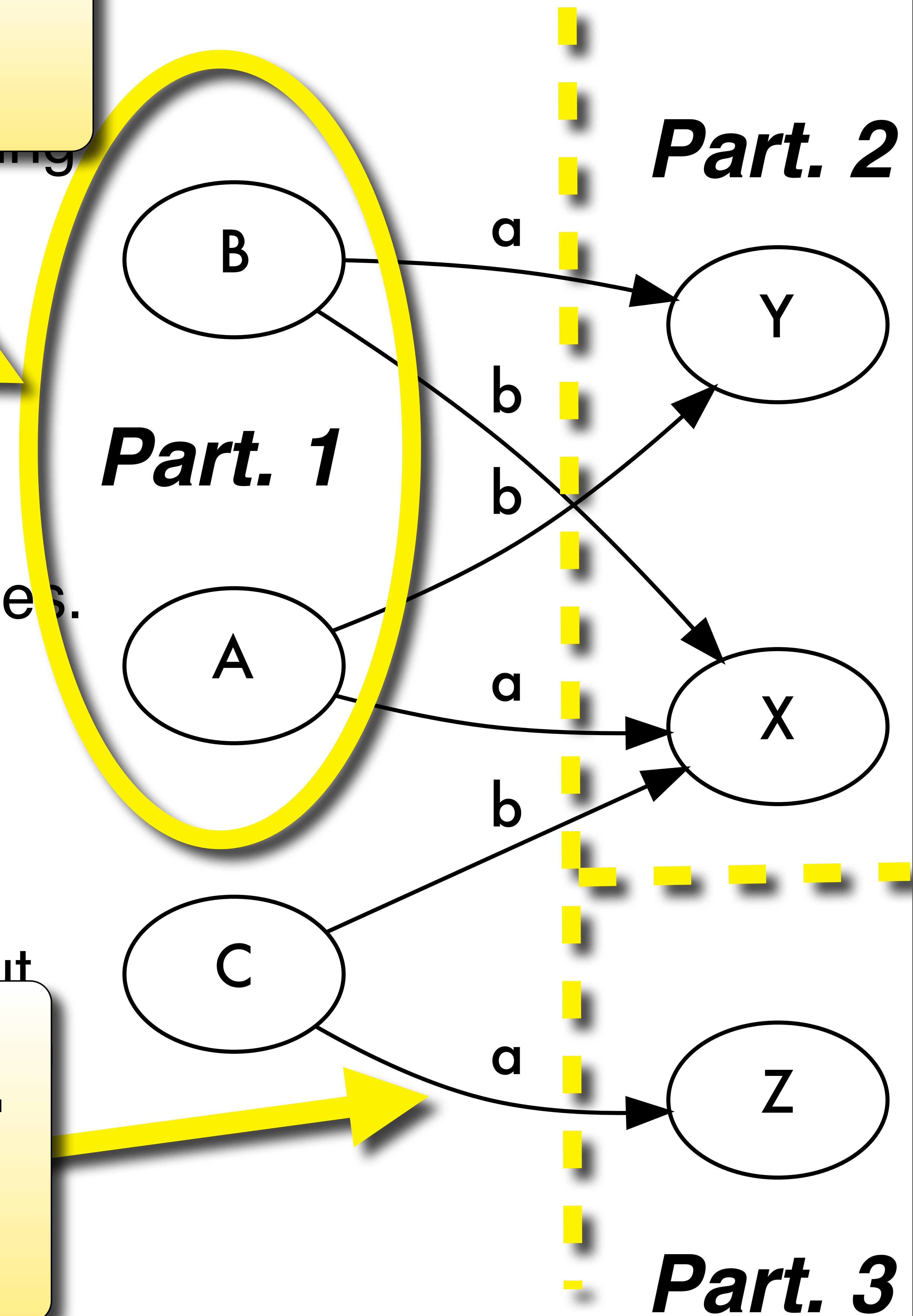
A and B are equivalent.

Goal: obtain
- ➡ For each RE, the minimal DFA is unique (ignoring simple renaming).
- ➡ DFA minimization: merge states that are equivalent.

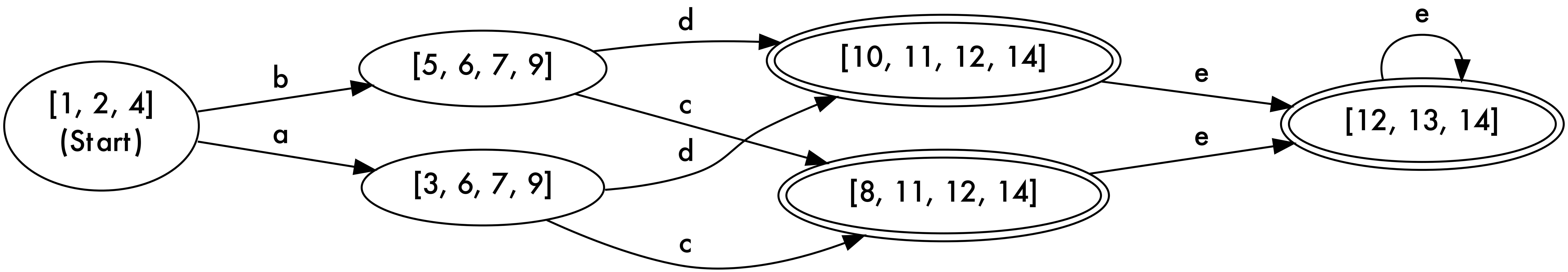
Key idea: it's easier to split.
- ➡ Start with two partitions: final and non-final states.
- ➡ Repeatedly split partitions until all partitions contain only equivalent states.
- ➡ Two states ***S1***, ***S2*** are equivalent if all their transitions "**agree**," i.e., if there exists an input symbol ***x*** such that the DFA transitions (on input

C is not equivalent to either A or B.

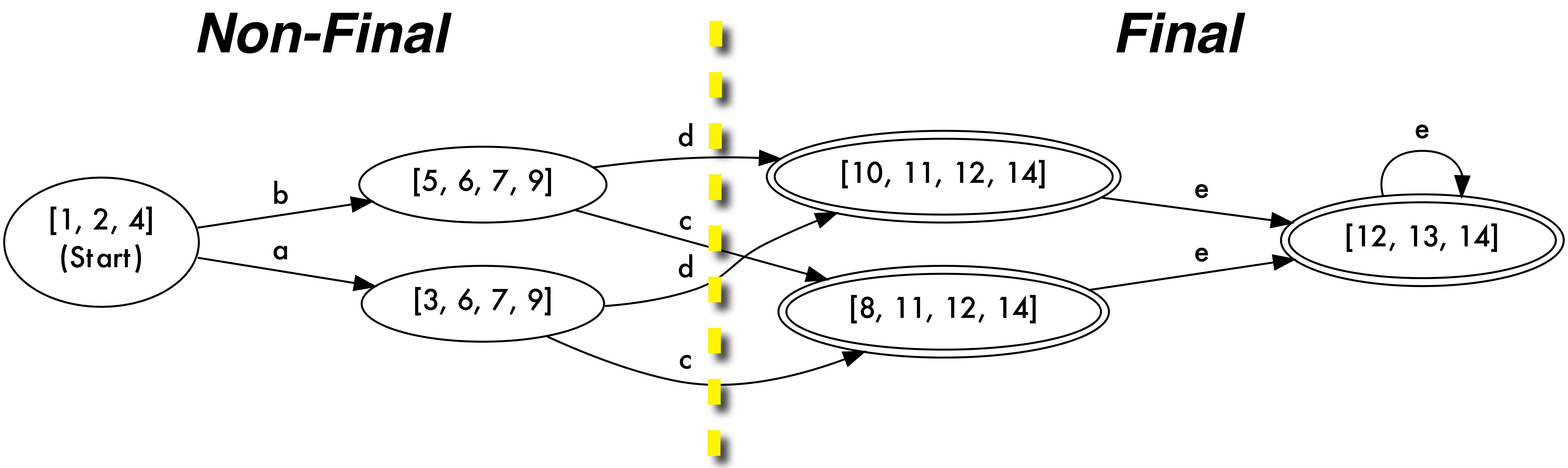
Because it has a transition into Part.3.



DFA Minimization Example

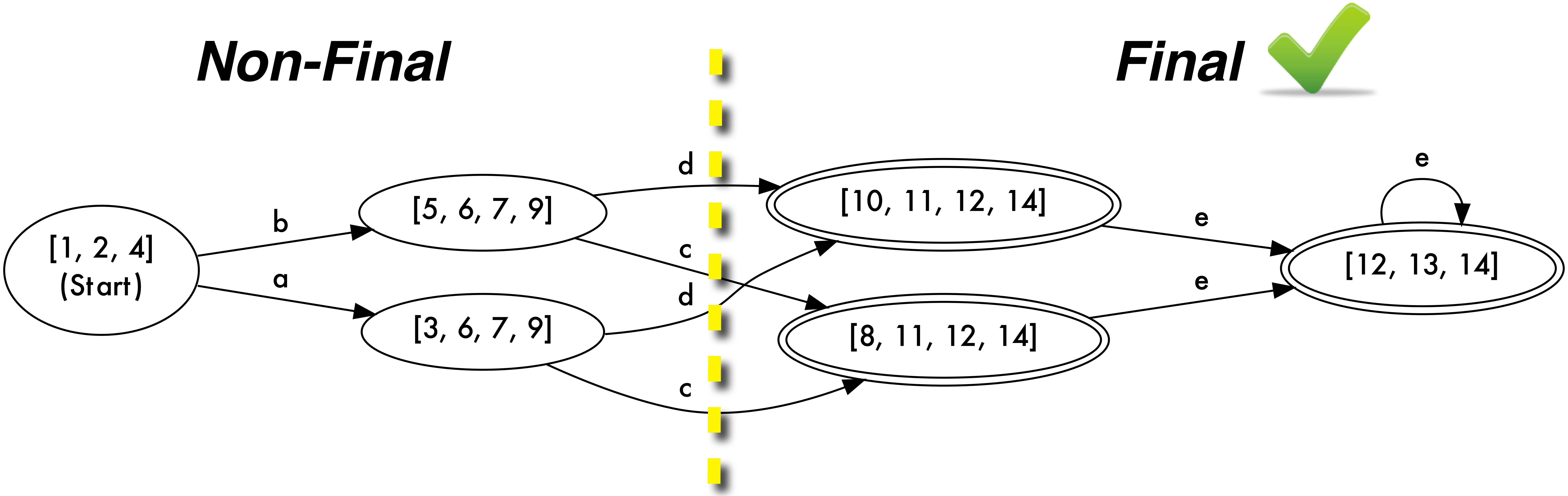


DFA Minimization Example



Partition final and non-final states.

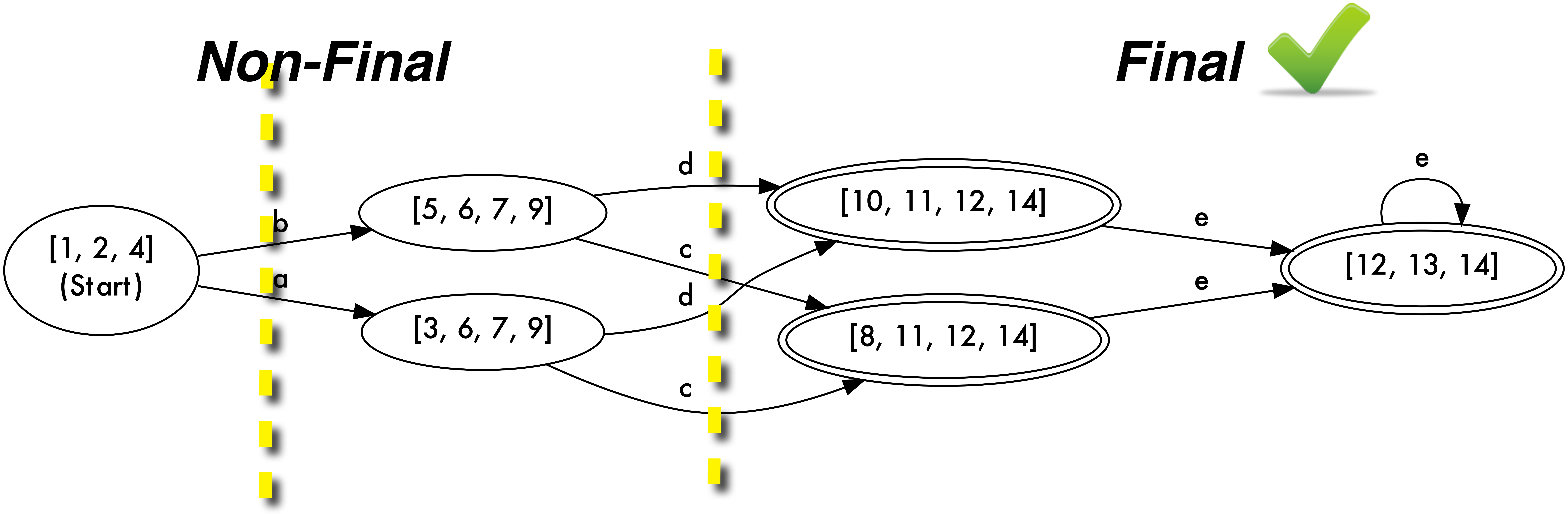
DFA Minimization Example



Examine final states.

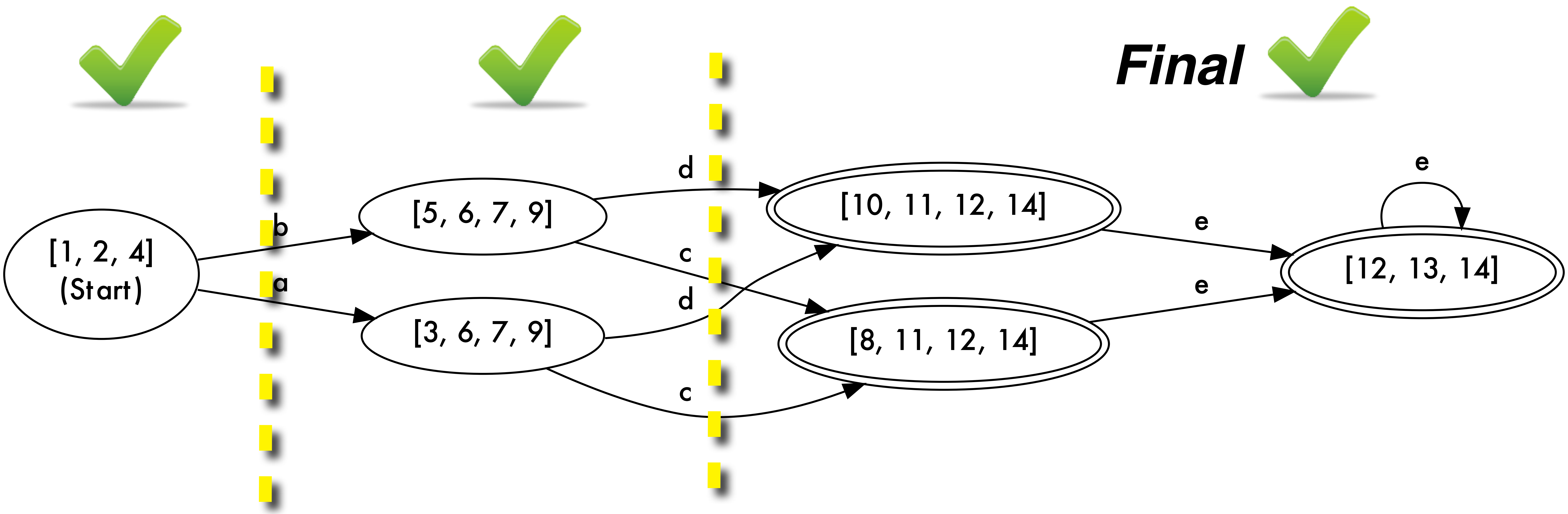
All final states are equivalent!

DFA Minimization Example



[1,2,4] is not equivalent to any other state:
it is the only state with a transition to the non-final partition.

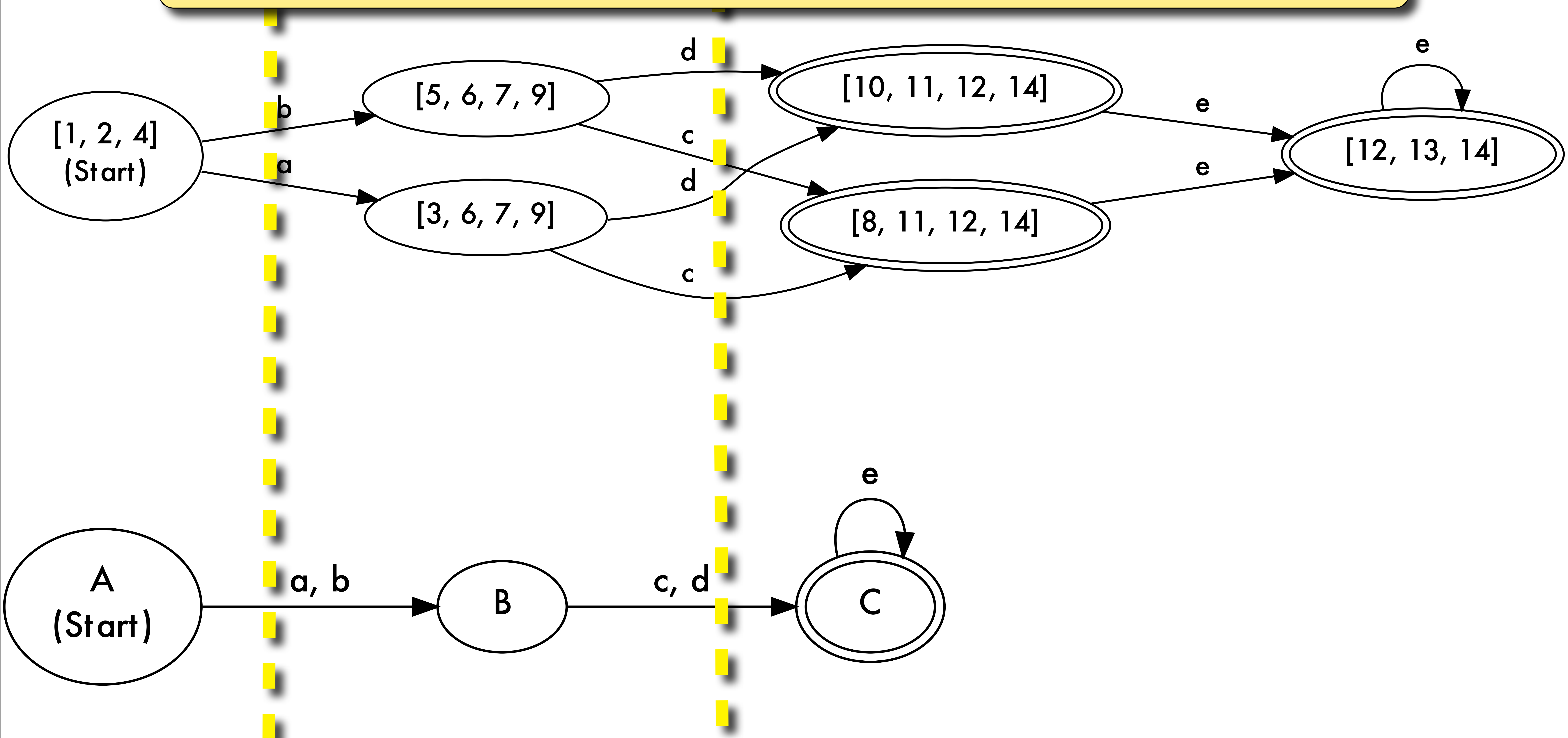
DFA Minimization Example



[5,6,7,9] and [3,6,7,9] are equivalent.
Thus, we are done.

DFA Minimization Example

Create one state for each partition.
We have obtained a minimal DFA for **(a|b)(c|d)e***.



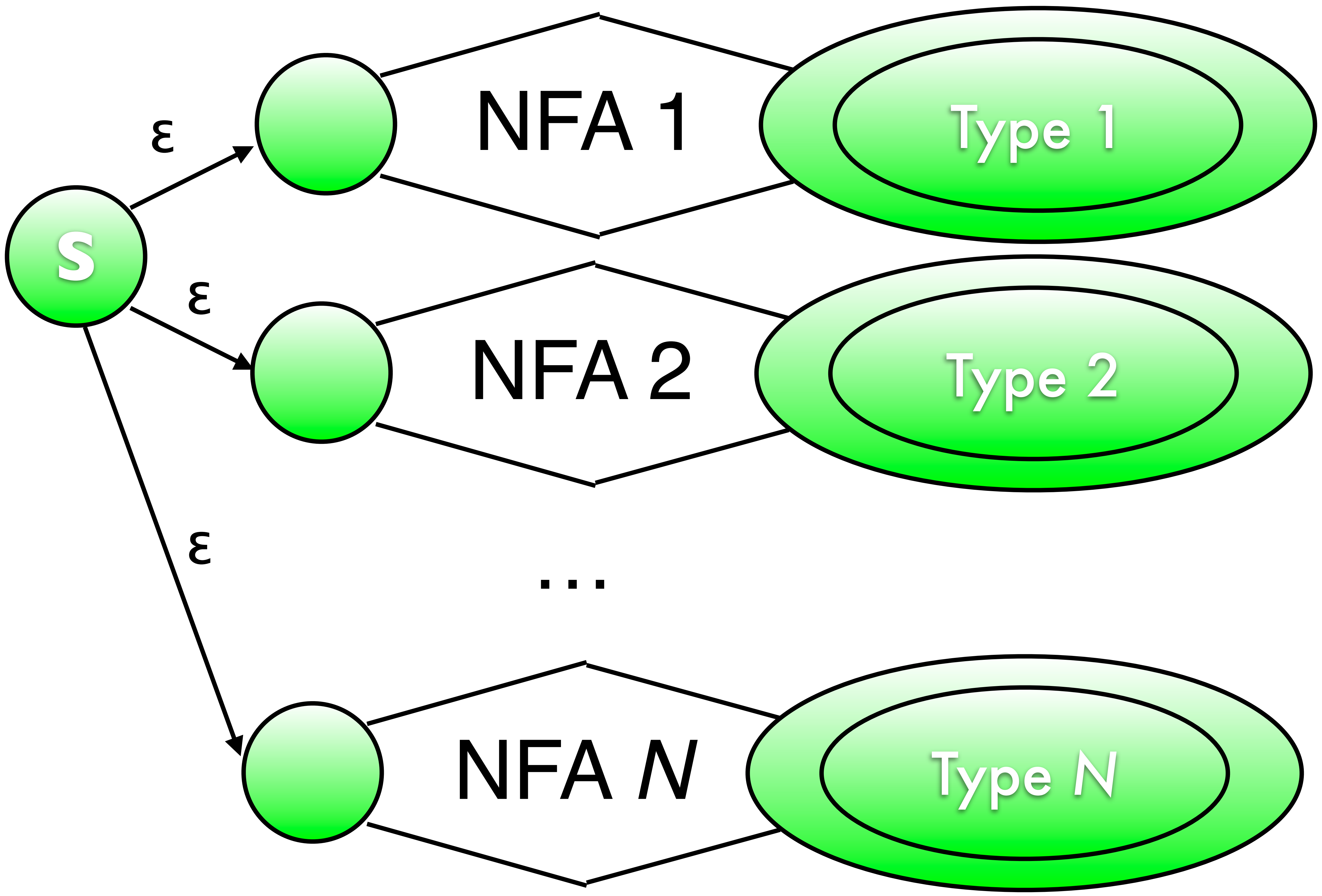
Recognizing Multiple Tokens

- Construction up to this point can only recognize a single token type.
  - Results in **Accept** or **Reject**, but does not yield **which** token was seen.
- Real lexical analysis must discern between **multiple token types**.
- Solution: **annotate final states** with token type.

Multi Token Construction

To build DFA for *N* tokens:

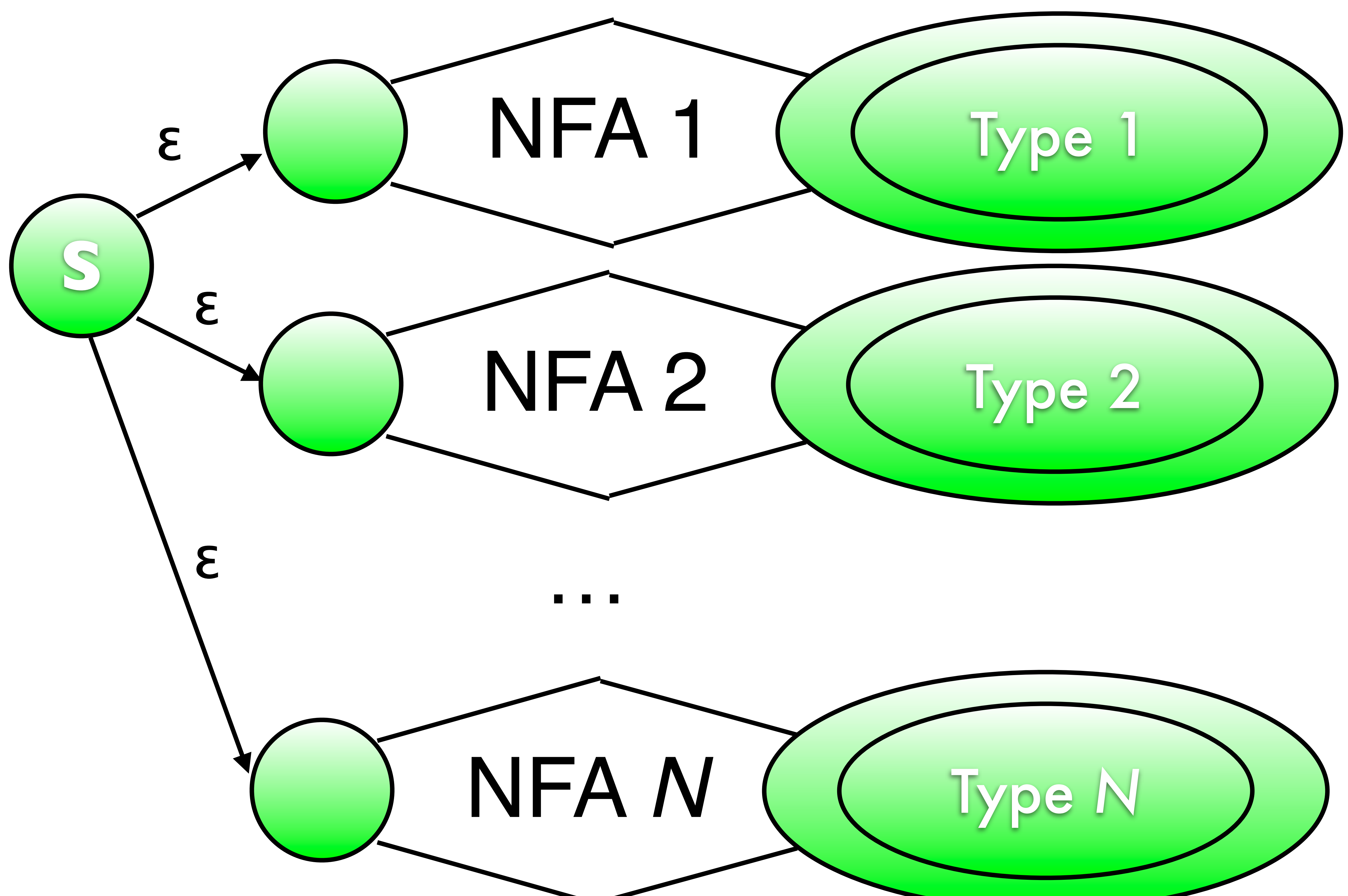
- ➡ Create a **NFA for each token type** RE as before.
- ➡ Join all token NFAs as shown below:



Multi Token Construction

To bu
➡ Cre ...ore.
➡ Joi

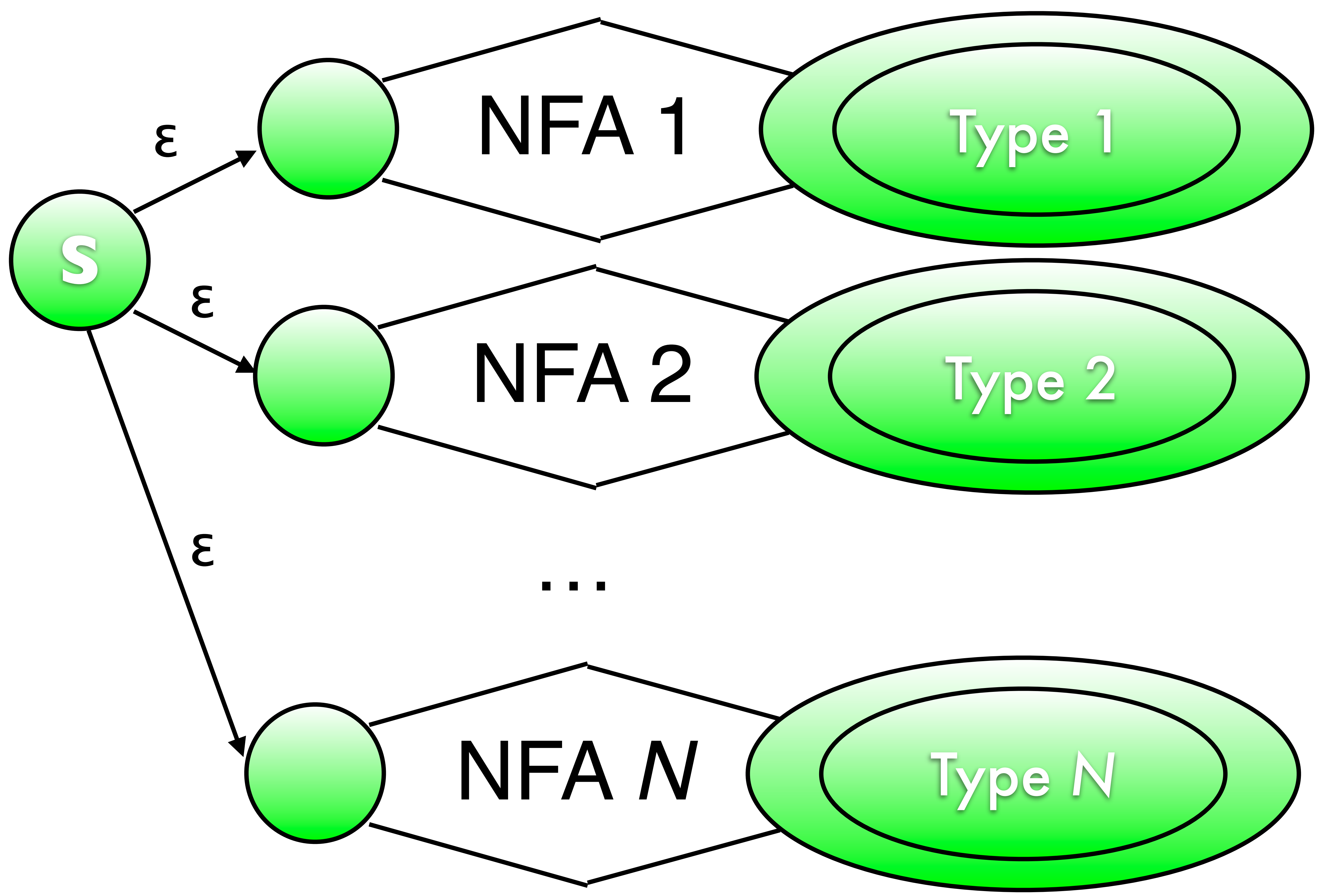
This is similar to **NFA construction rule 3**.
Key difference: we **keep all final states**.



Multi Token Construction

To bu
- ➡ Cre ... ore.
- ➡ Joi

This is similar to **NFA construction rule 3**.
Key difference: we **keep all final states**.



Token Precedence

Consider the following regular grammar.

➡ Create DFA to recognize **identifiers** and **keywords**.

identifier	→	*letter* (*letter* | *digit* | _)*
keyword	→	if | else | while
digit	→	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
letter	→	a | b | c | … | z

Can you spot a problem?

Token Precedence

Consider the following regular grammar.

➡ Create DFA to recognize **identifiers** and **keywords**.

identifier	→	*letter* (*letter* | *digit* | _)*
keyword	→	if | else | while
digit	→	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
letter	→	a | b | c | … | z

All keywords are also identifiers!
The grammar is ambiguous.
Example: for string 'while', there are two accepting states in the final NFA with **different labels**.

Token Precedence

Consider the following regular grammar.

- ➡ Create DFA to recognize **identifiers** and **keywords**.

identifier	→ *letter* (*letter* | *digit* | _)*
keyword	→ if | else | while
digit	→ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
letter	→ a | b | c | … | z

Solution

- ➡ Assign **precedence values** to tokens (and labels).
- ➡ In case of **ambiguity**, prefer final state with highest precedence value.

Token Precedence

Consider the following regular grammar.

➡ Create DFA to recognize **identifiers** and **keywords**.

> **Note**: during DFA optimization, two final states are **not equivalent** if they are labeled with **different token types**.

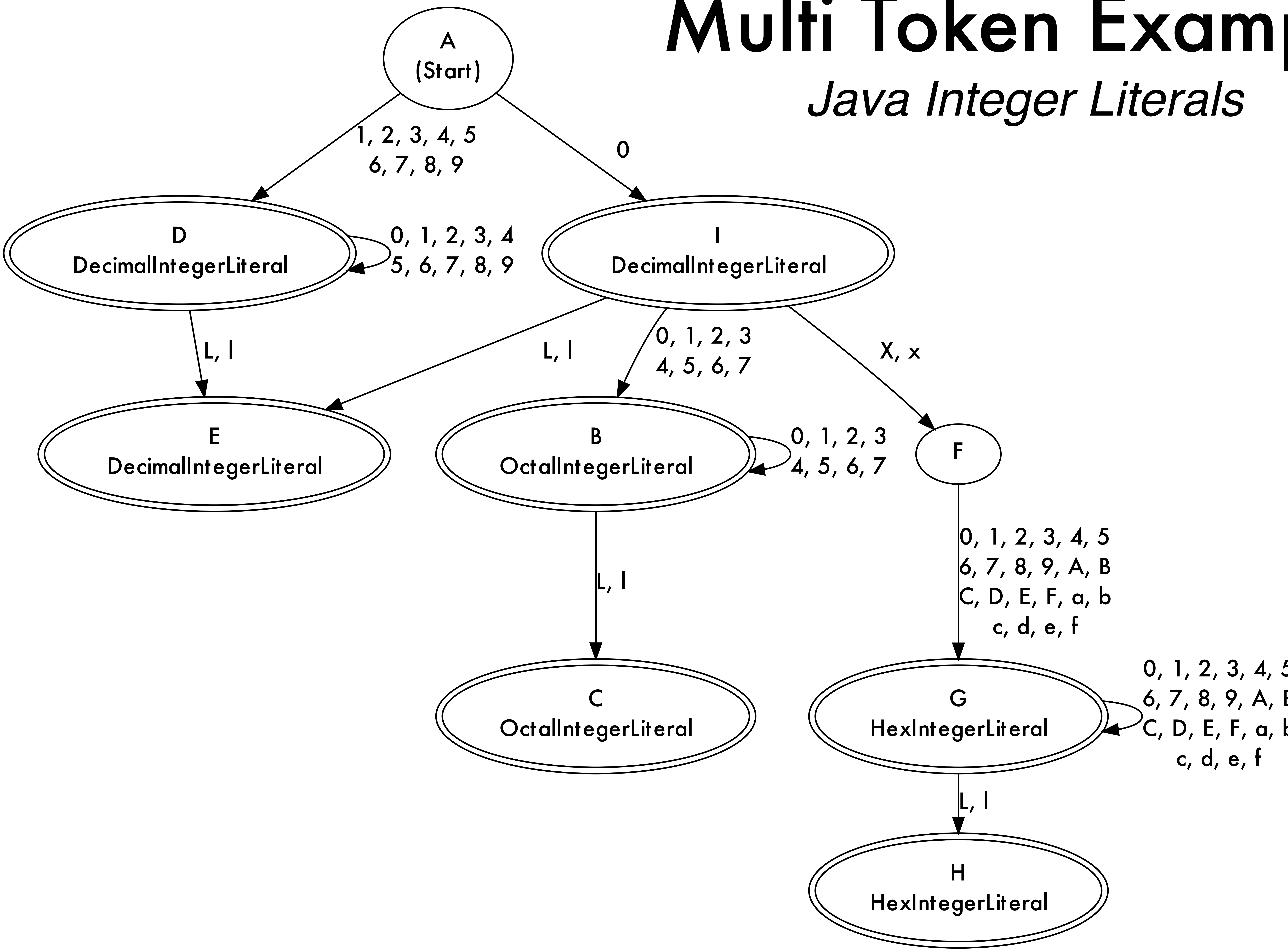
Solution

➡ Assign **precedence values** to tokens (and labels).

➡ In case of **ambiguity**, prefer final state with highest precedence value.

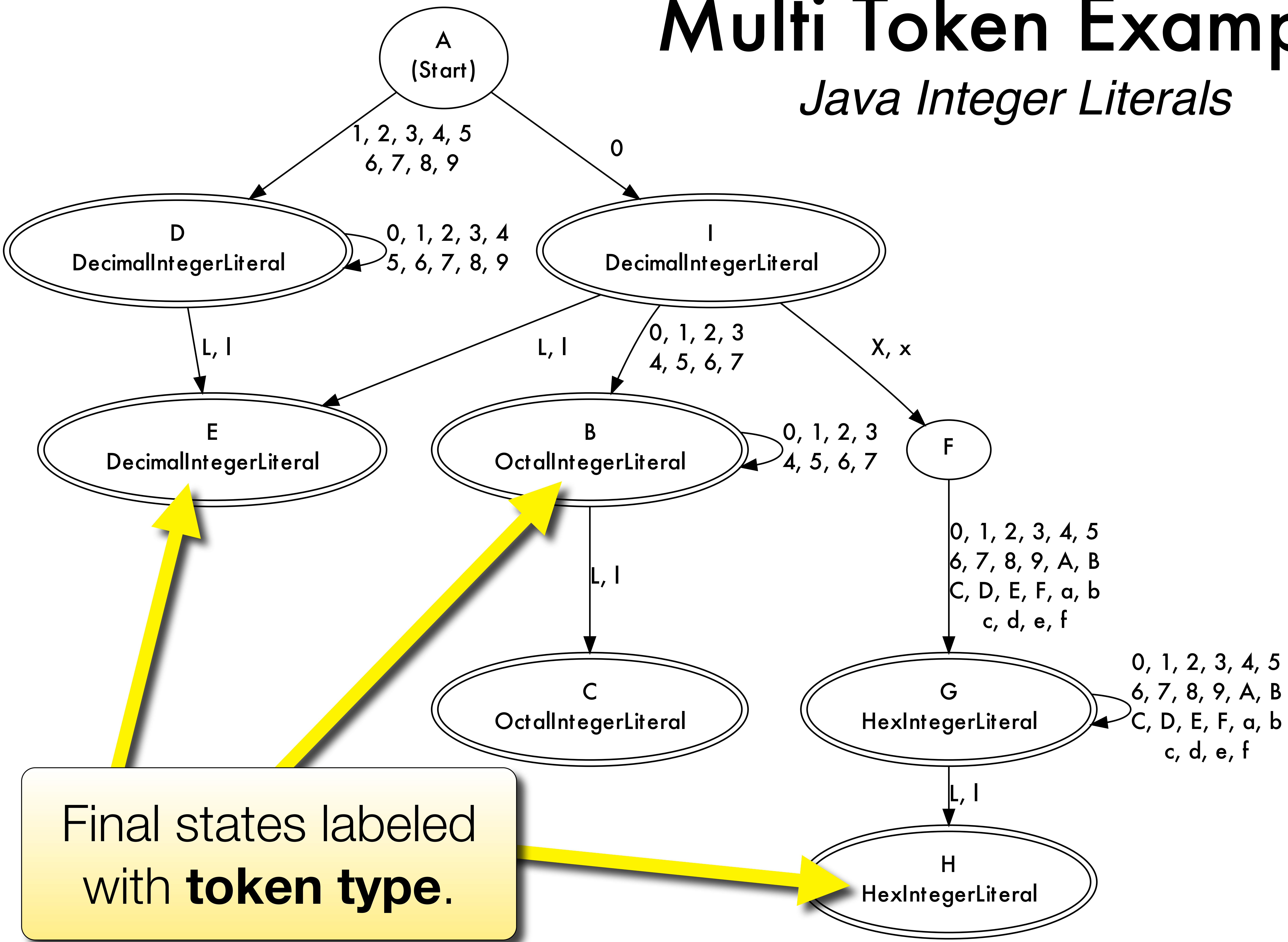
Multi Token Example

Java Integer Literals



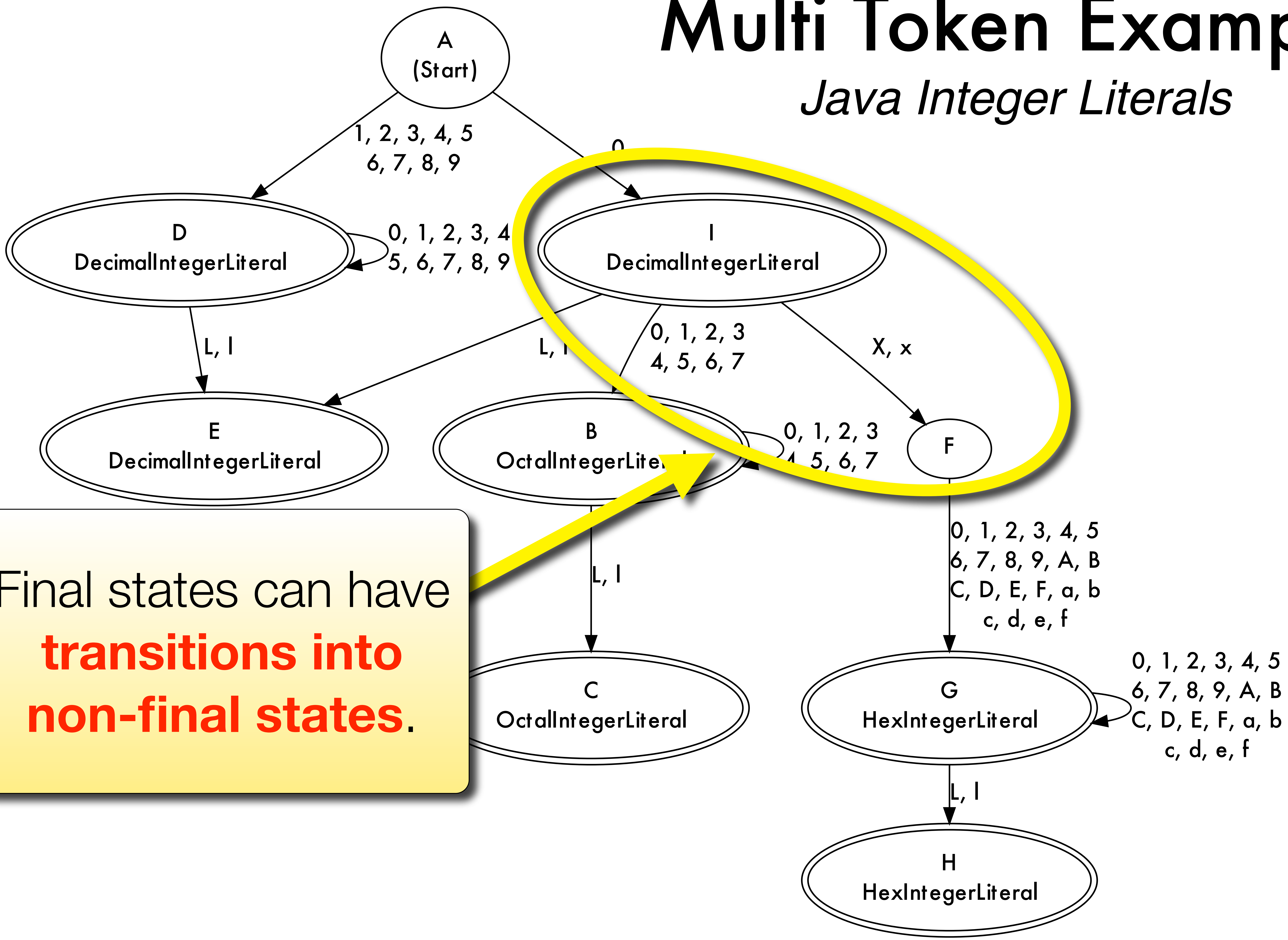
Multi Token Example

Java Integer Literals



Multi Token Example

Java Integer Literals



Final states can have **transitions into non-final states.**

Extended Regular Expressions

some commonly used abbreviations

+ n ? [] [^]

+ Kleene Plus

name → *letter*+

is the same as

name → *letter letter**

Extended Regular Expressions

some commonly used abbreviations

+ n ? [] [^]

n times

name → *letter*^{3}

is the same as

name → *letter letter letter*

Extended Regular Expressions

some commonly used abbreviations

\+ n ? [] [^]

? optionally

$ZIP \rightarrow \ digit^5 \ (\text{-}digit^4)?$

is the same as

$ZIP \rightarrow \ digit^5 \ (\ \varepsilon \ | \ \text{-}digit^4 \ )$

Extended Regular Expressions

some commonly used abbreviations

+ n ? [] [^]

[] one off

digit → [123456789]

is the same as

digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Extended Regular Expressions

some commonly used abbreviations

\+ n ? [] [^]

[^] not one off

notADigit → [^123456789]

is the same as

notADigit → A | B | C ...

Extended Regular Expressions

nly used abbreviations

? [] [^]

Every character except those listed between **[^** and **]**.

[^] not one off

notADigit → [^123456789]

is the same as

notADigit → A | B | C ...

Limitations of REs

Suppose we wanted to remove extraneous, balanced '(' ')' pairs around identifiers.

- Example: report `(sum)`, `((sum))` and `(((sum)))` simply as *Identifier*.
- But not: `((sum)`

One might try:

identifier → $(^n$ *letter+* $)^m$ **such that** $n = m$

This **cannot** be expressed with regular expressions!
Requires a **recursive grammar**: let the parser do it.