Finite Automata

CS 4100
Gordon Stewart
Ohio University

A Language of Regexps

Regexp Syntax

```
r ::= empty | ϵ | c          (*c ∈ Σ*)
    | r1 . r2                (*concatenation*)
    | r*                     (*Kleene star*)
    | (r1 || r2)             (*r1 or r2*)
    | (r1 && r2)             (*r1 and r2*)
    | ¬r                     (*not r*)
```

Regexp Semantics

$$
\begin{aligned}
L(empty) &= \emptyset \\
L(\epsilon) &= \{ \text{“”} \} \\
L(c) &= \{ \text{“}c\text{”} \} \\
L(r1\,.\,r2) &= \{ s1 \wedge s2 \mid s1 \in L(r1) \textit{ and } s2 \in L(r2) \} \\
L(r^*) &= \{ \text{“”} \} \cup L(r\,.\,r^*) \\
L(r1 \,||\, r2) &= L(r1) \cup L(r2) \\
L(r1 \,\&\&\, r2) &= L(r1) \cap L(r2) \\
L(\neg r) &= \Sigma^* - L(r)
\end{aligned}
$$

A Naïve Implementation

Regexp Semantics

$$L(empty) = \emptyset$$
$$L(\epsilon) = \{ \text{“”} \}$$
$$L(c) = \{ \text{“}c\text{”} \}$$
$$\boxed{L(r1 \,.\, r2) = \{ s1 \wedge s2 \mid s1 \in L(r1) \; and \; s2 \in L(r2) \}}$$
$$L(r^*) = \{ \text{“”} \} \cup L(r \,.\, r^*)$$

Problem: Is string s in the language of r1 . r2?

Naïve Approach:

- Consider all strings s’ formed by s1 ^ s2, where
  - s1 drawn from L(r1)
  - s2 drawn from L(r2)
- In general, there are O(size(L(r1)) * size(L(r2))) such strings.
- Check whether s = s’.

A Slightly Better Implementation

Regexp Semantics

$$
\begin{aligned}
L(empty) &= \emptyset \\
L(\epsilon) &= \{\text{“”}\} \\
L(c) &= \{\text{“}c\text{”}\} \\
\boxed{L(r1\,.\,r2)} &\boxed{= \{\, s1 \wedge s2 \mid s1 \in L(r1) \text{ and } s2 \in L(r2) \,\}} \\
L(r^*) &= \{\text{“”}\} \cup L(r\,.\,r^*)
\end{aligned}
$$

Problem: Is string s in the language of r1 . r2?

Slightly Better Approach:

- Consider all *splittings* of s = s1 ^ s2. E.g.,
  - If s = “hello”, then
    splittings = {(“”, “hello”), (“h”,“ello”), (“he”,“llo”), …}.
- String s is in L(r1 . r2) iff there exists a splitting (s1, s2) such that
  - s1 is in L(r1) and
  - s2 is in L(r2).
- How many splittings of a string of length ***n***?

A Slightly Better Implementation

Regexp Semantics

$$
\begin{aligned}
L(empty) &= \emptyset \\
L(\epsilon) &= \{\text{“”}\} \\
L(c) &= \{\text{“}c\text{”}\} \\
L(r1\,.\,r2) &= \{\, s1 \wedge s2 \mid s1 \in L(r1)\ and\ s2 \in L(r2) \,\} \\
L(r^*) &= \{\text{“”}\} \cup L(r\,.\,r^*)
\end{aligned}
$$

Problem: Is string s in the language of r1 . r2?

A Better Slightly Better Approach:

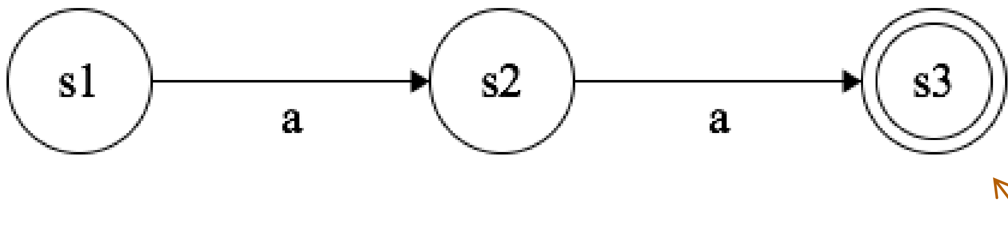
- Brzozowksi derivatives!

FINITE AUTOMATA

Finite Automata

- A finite automaton ***A*** consists of
  - A finite set of *states* S
  - A set of *labels* (drawn from some alphabet Σ)
  - A finite set of *labeled transitions* between states
- When specifying a finite automaton, we also often declare
  - A set of *initial states* INITIAL
  - A set of *final states* FINAL
- FINAL states indicate *acceptance* of a particular string

A Simple Example

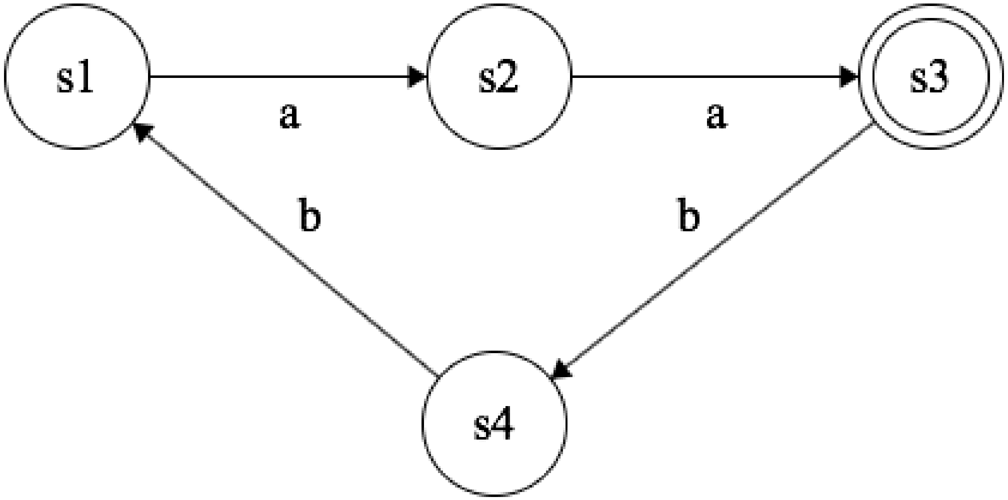


Double circle indicates **FINAL** state

- States S = {s1, s2, s3}
- Labels = { a }
- Transitions = { (s1, a, s2), (s2, a, s3) }
- INITIAL states = { s1 }
- FINAL states = { s3 }

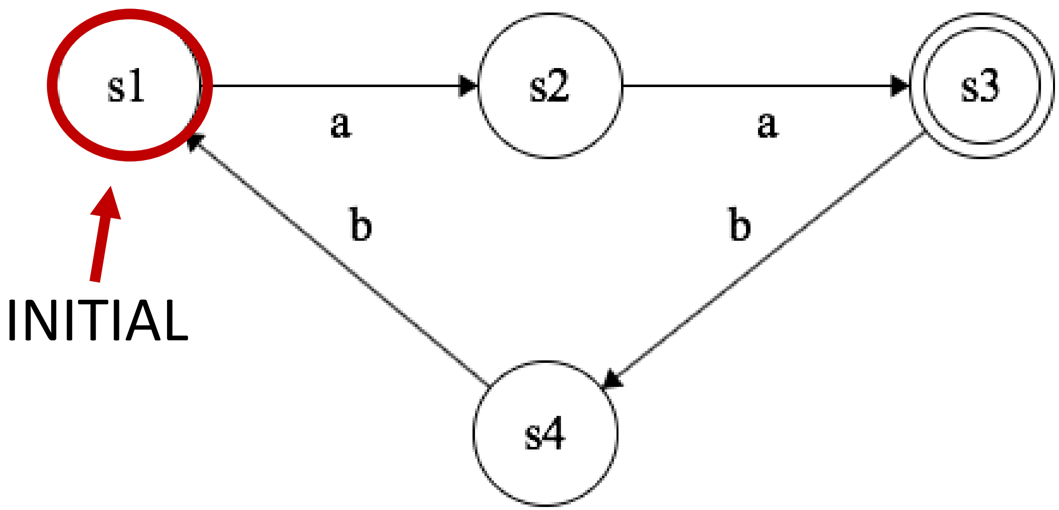
Which strings does this automaton accept?

A Second Example



- States S = {s1, s2, s3,s4,s5}
- Labels = { a, b }
- Transitions = { (s1, a, s2), (s2, a, s3), (s3, b, s4), (s4, b, s1) }
- INITIAL = { s1 }
- FINAL = { s3 }

Does an Automaton Match a String?



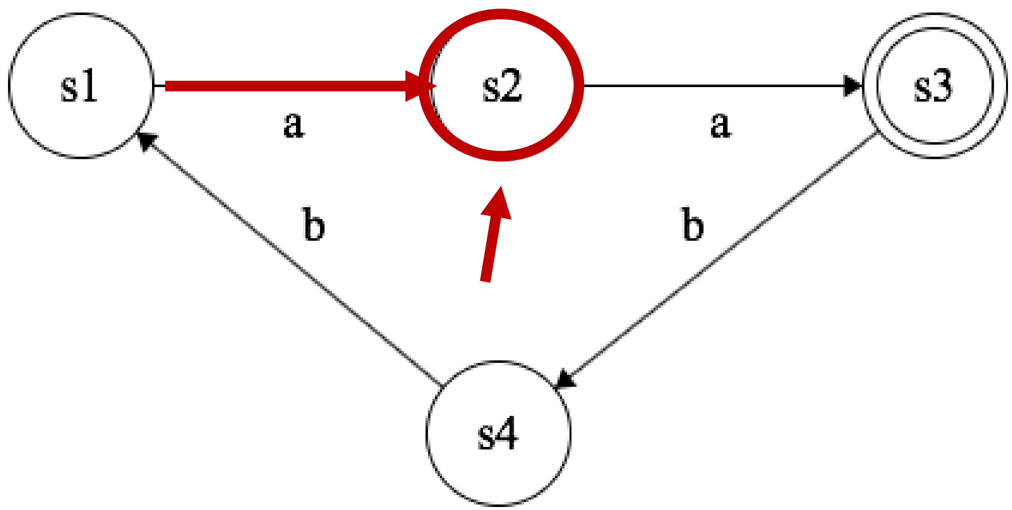
"aabbaa"

Position 0

Algorithm:

1. Are we in FINAL state?
2. If so and no chars left, **accept**.
3. Otherwise, follow a transition (advancing one char) and go to 1.
4. If no transition possible, **reject**.

Does an Automaton Match a String?



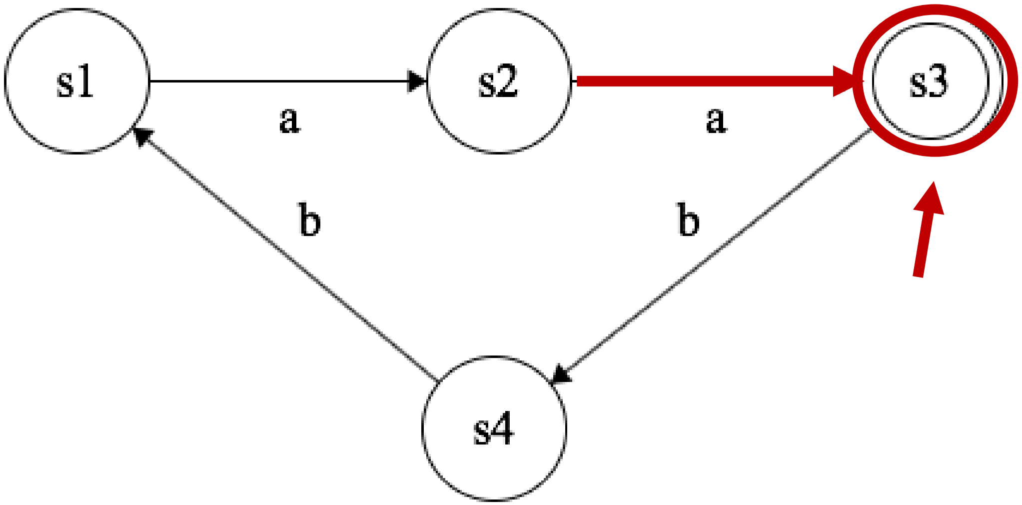
“aabbaa”

Position 1

Algorithm:

1. Are we in FINAL state?
2. If so and no chars left, **accept**.
3. Otherwise, follow a transition (advancing one char) and go to 1.
4. If no transition possible, **reject**.

Does an Automaton Match a String?

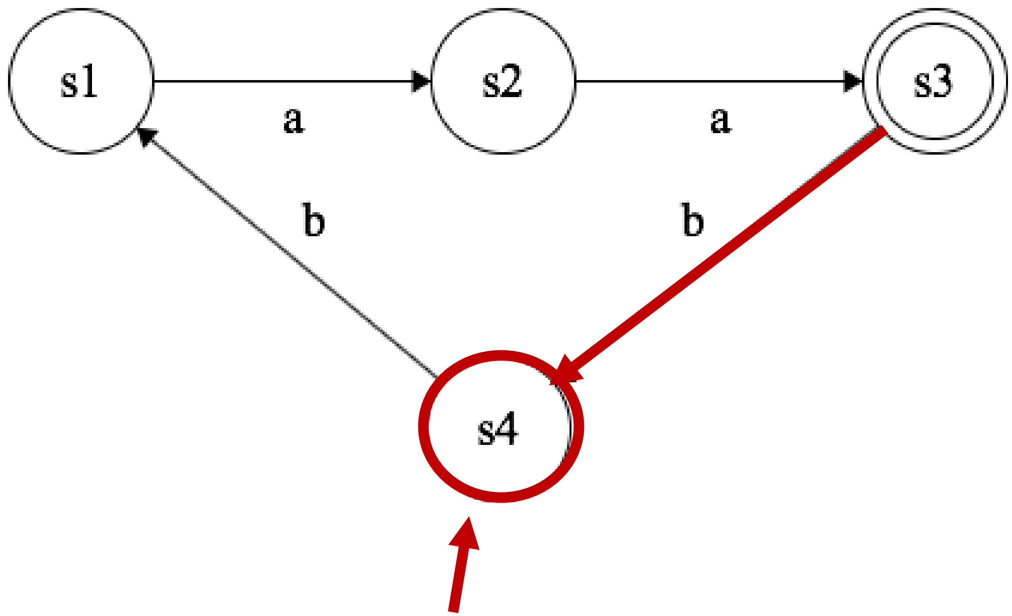


“aabbaa”

Algorithm:

1. Are we in FINAL state?
2. If so and no chars left, **accept**.
3. Otherwise, follow a transition (advancing one char) and go to 1.
4. If no transition possible, **reject**.

Does an Automaton Match a String?

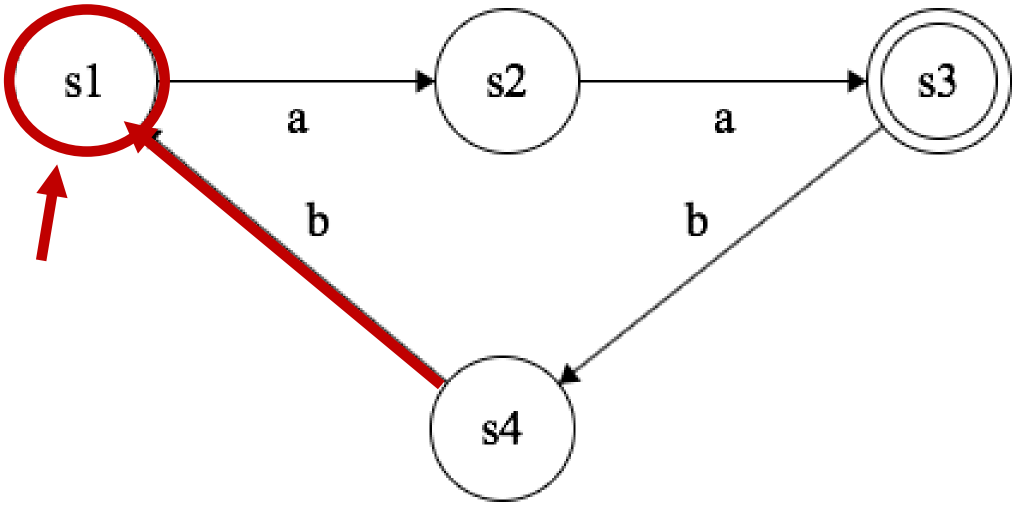


“aabbaa”

Algorithm:

1. Are we in FINAL state?
2. If so and no chars left, **accept**.
3. Otherwise, follow a transition (advancing one char) and go to 1.
4. If no transition possible, **reject**.

Does an Automaton Match a String?

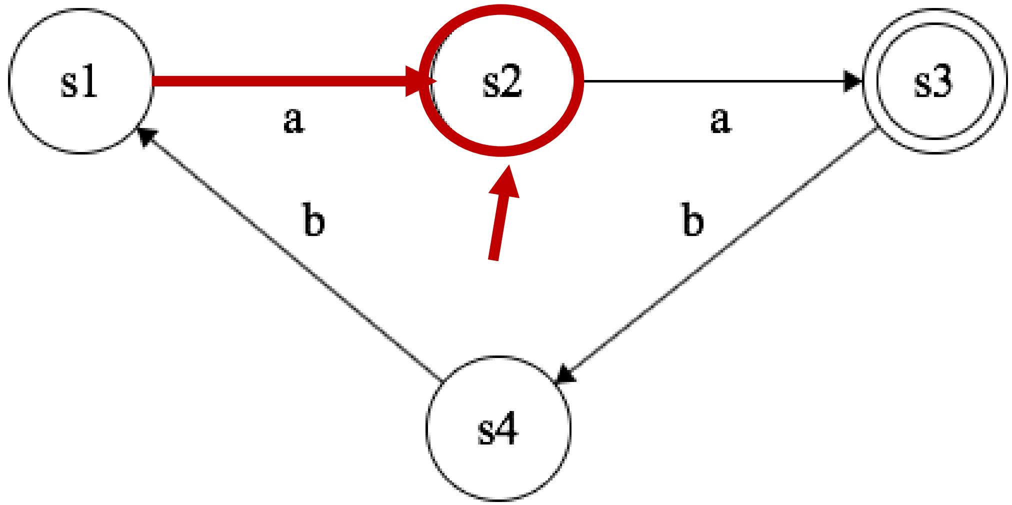


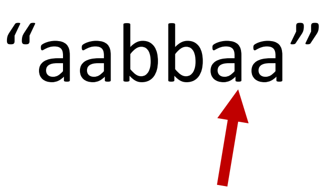
“aabbaa”

Algorithm:

1. Are we in FINAL state?
2. If so and no chars left, **accept**.
3. Otherwise, follow a transition (advancing one char) and go to 1.
4. If no transition possible, **reject**.

Does an Automaton Match a String?

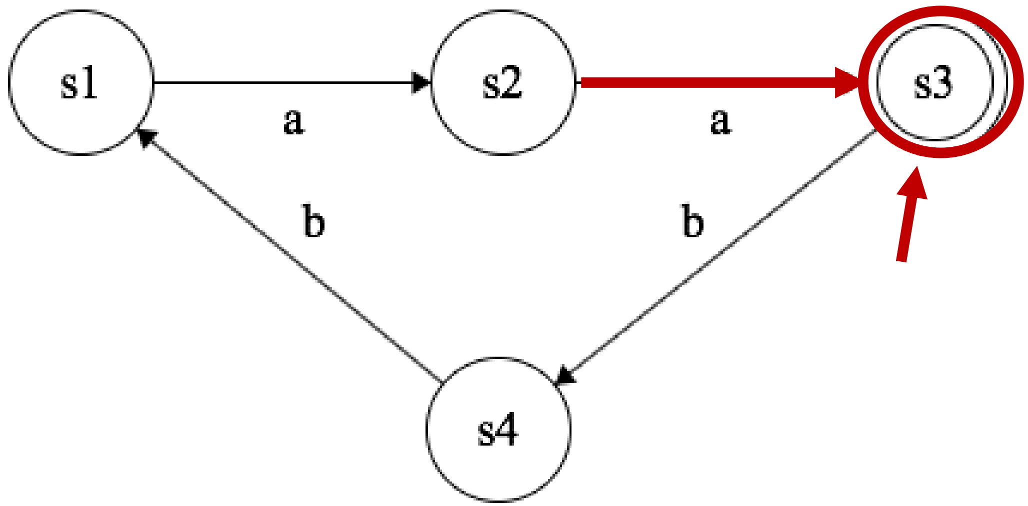




Algorithm:

1. Are we in FINAL state?
2. If so and no chars left, **accept**.
3. Otherwise, follow a transition (advancing one char) and go to 1.
4. If no transition possible, **reject**.

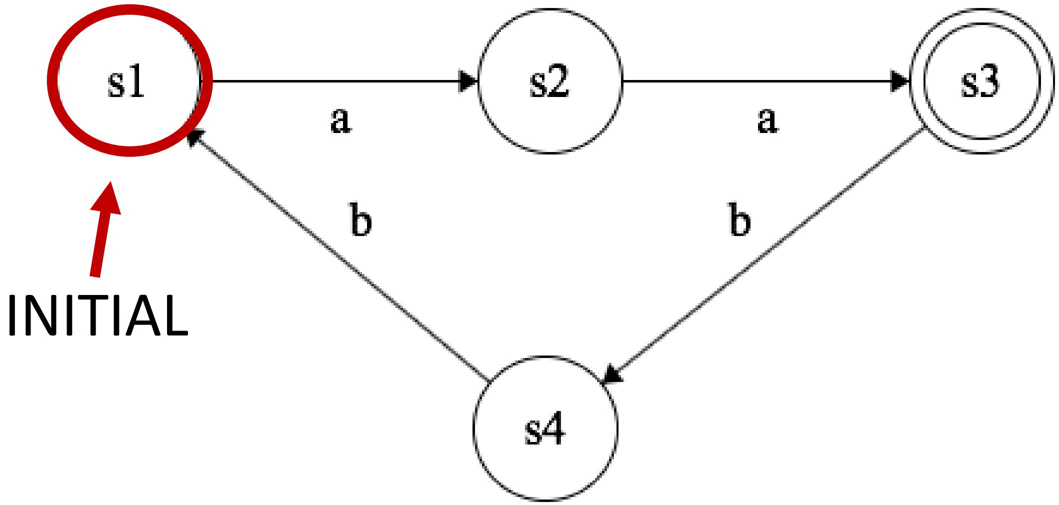
Does an Automaton Match a String?



"aabbaa"

Algorithm:
1. Are we in FINAL state?
2. If so and no chars left, **accept.**
3. Otherwise, follow a transition (advancing one char) and go to 1.
4. If no transition possible, **reject**.

Does an Automaton Match a String?



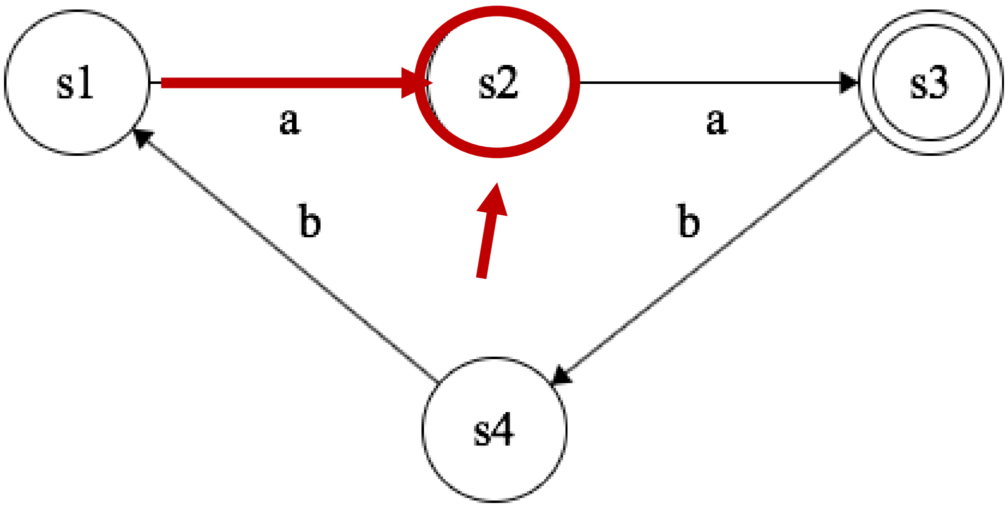
“aabb”

Position 0

Algorithm:

1. Are we in FINAL state?
2. If so and no chars left, **accept**.
3. Otherwise, follow a transition (advancing one char) and go to 1.
4. If no transition possible, **reject**.

Does an Automaton Match a String?



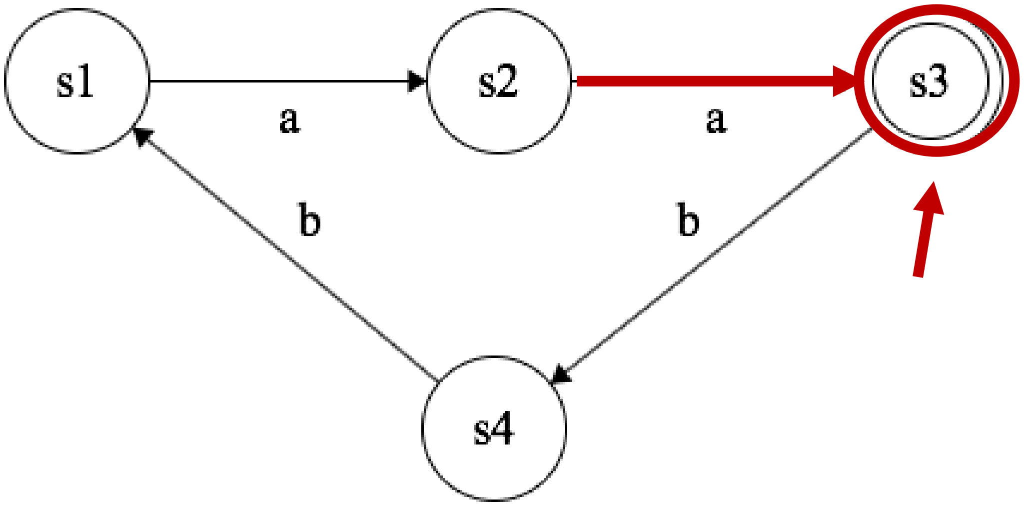
“aabb”

Position 1

Algorithm:

1. Are we in FINAL state?
2. If so and no chars left, **accept**.
3. Otherwise, follow a transition (advancing one char) and go to 1.
4. If no transition possible, **reject**.

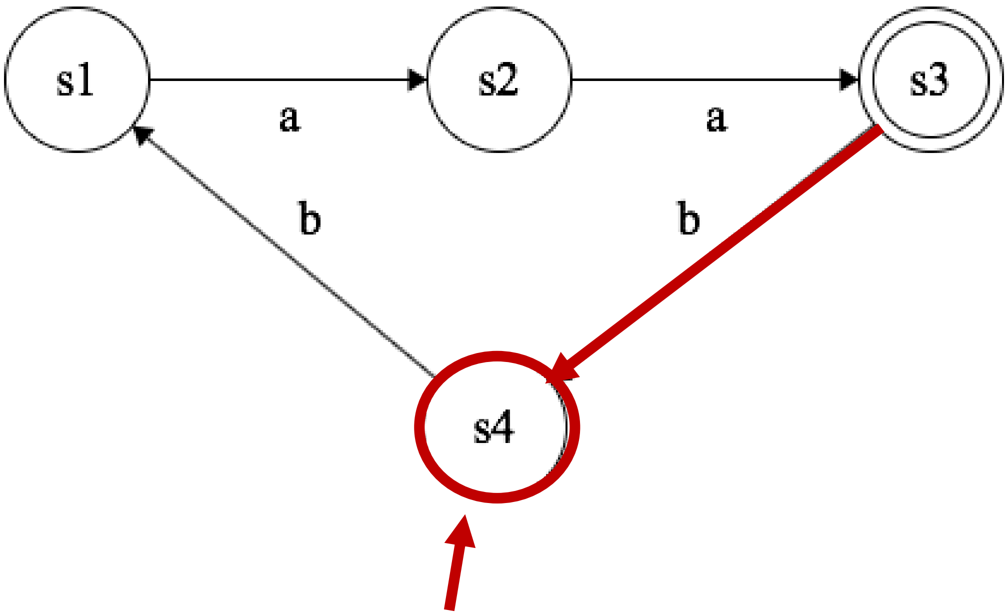
Does an Automaton Match a String?



“aabb”

Algorithm:
1. Are we in FINAL state?
2. If so and no chars left, **accept**.
3. Otherwise, follow a transition (advancing one char) and go to 1.
4. If no transition possible, **reject**.

Does an Automaton Match a String?

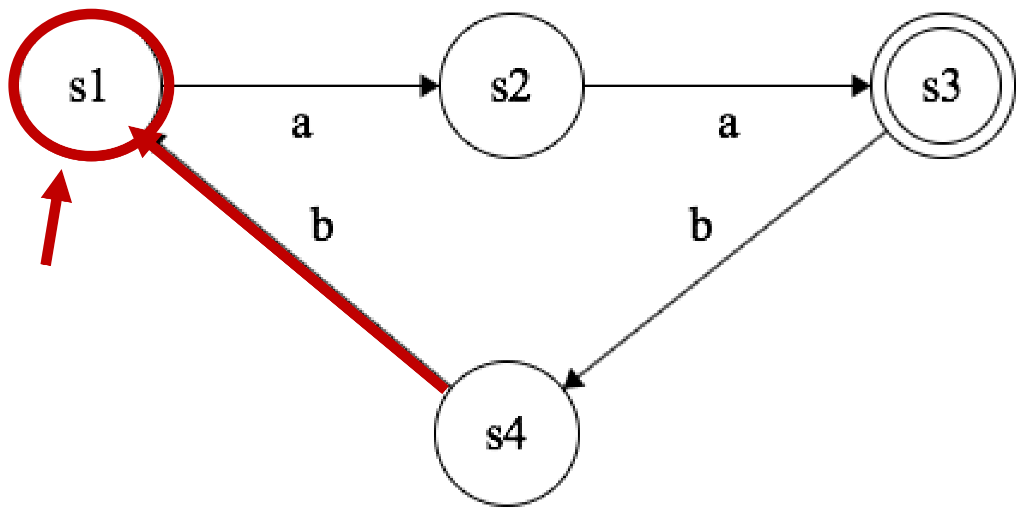


“aabb”

Algorithm:

1. Are we in FINAL state?
2. If so and no chars left, **accept**.
3. Otherwise, follow a transition (advancing one char) and go to 1.
4. If no transition possible, **reject**.

Does an Automaton Match a String?

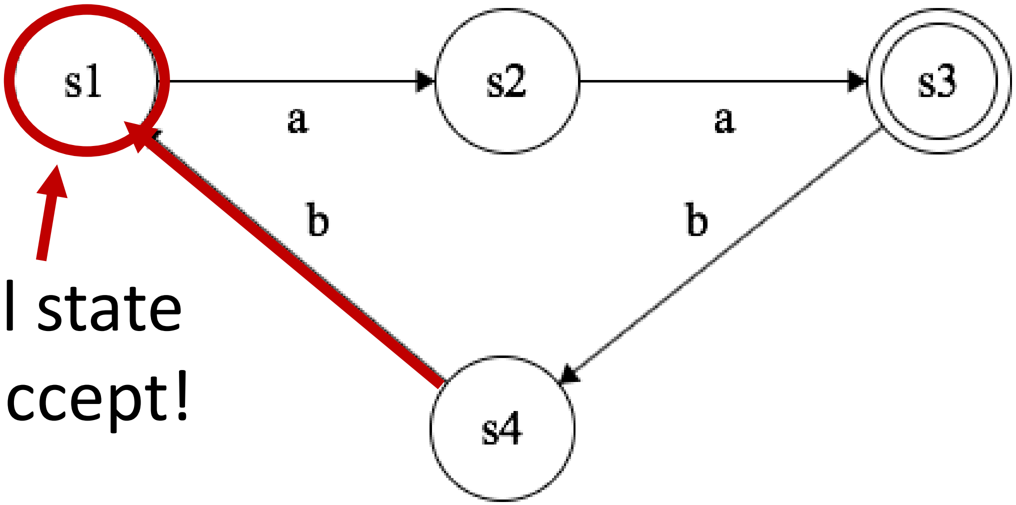


“aabb”

Algorithm:

1. Are we in FINAL state?
2. If so and no chars left, **accept**.
3. Otherwise, follow a transition (advancing one char) and go to 1.
4. If no transition possible, **reject**.

Does an Automaton Match a String?



Not a final state so don't accept!

"aabb"

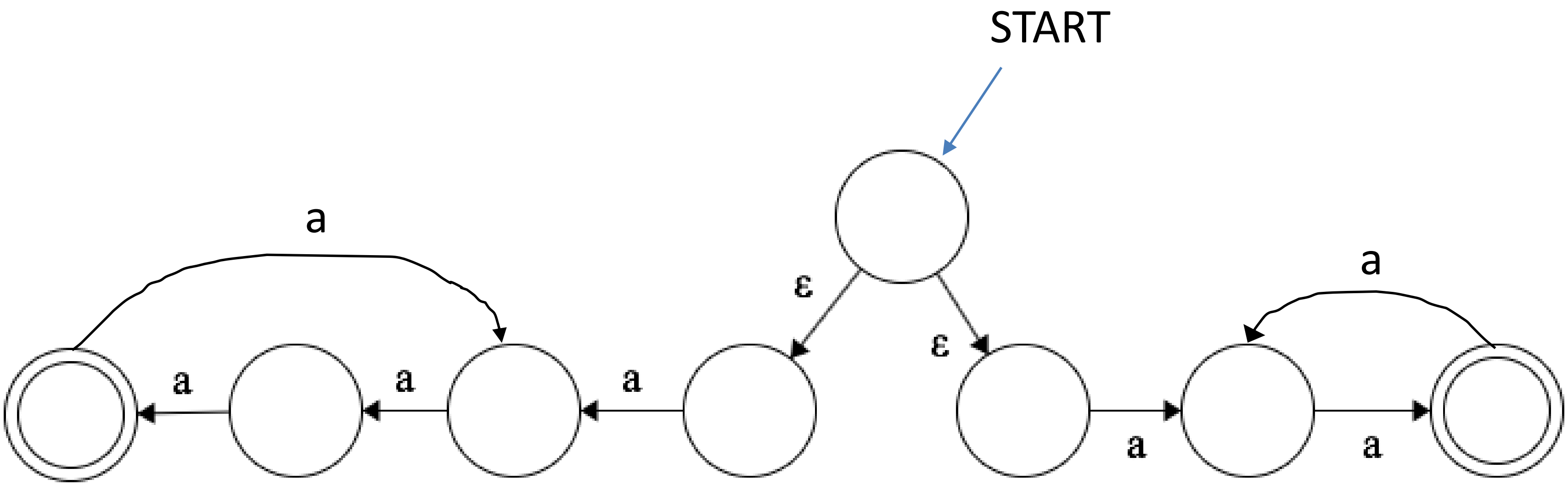
No more input…

Algorithm:

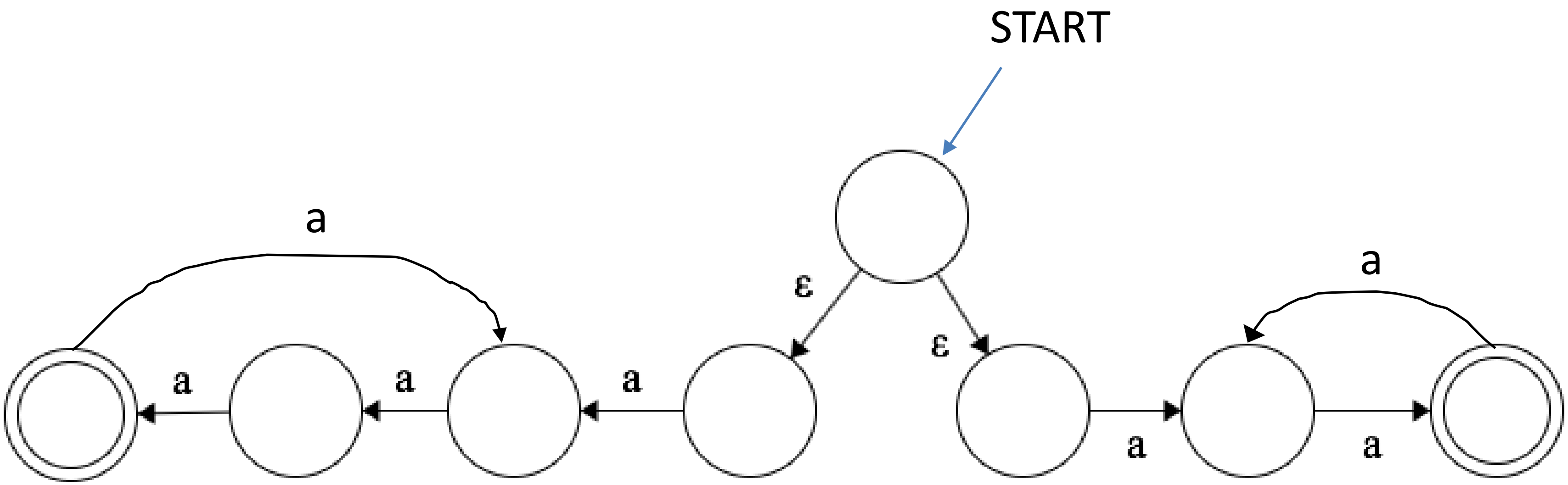
1. Are we in FINAL state?
2. If so and no chars left, **accept**.
3. Otherwise, follow a transition (advancing one char) and go to 1.
4. If no transition possible, **reject**.

NFAS: NONDETERMINISTIC FINITE AUTOMATA

A (Nondeterministic) Example



A (Nondeterministic) Example



Multiples of length 2

"aa"

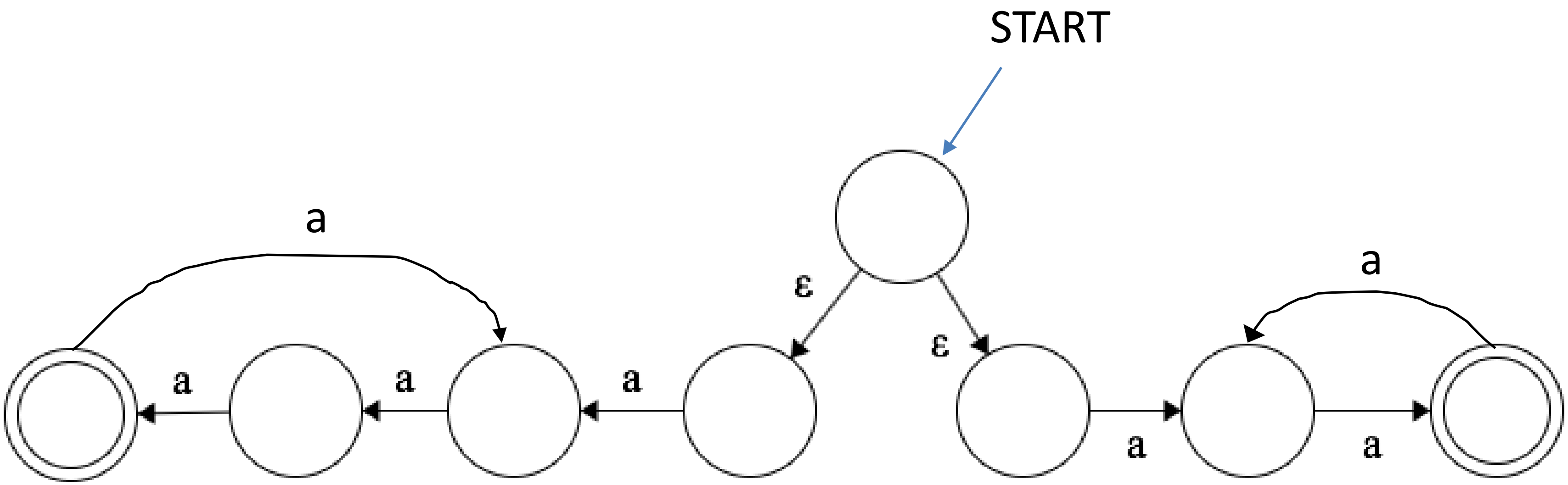
"aaaa"

"aaaaaa"

"aaaaaaaa"

…

A (Nondeterministic) Example



Multiples of length 3

“aaa”

“aaaaaa”

“aaaaaaaaa”

“aaaaaaaaaaaa”

…

Multiples of length 2

“aa”

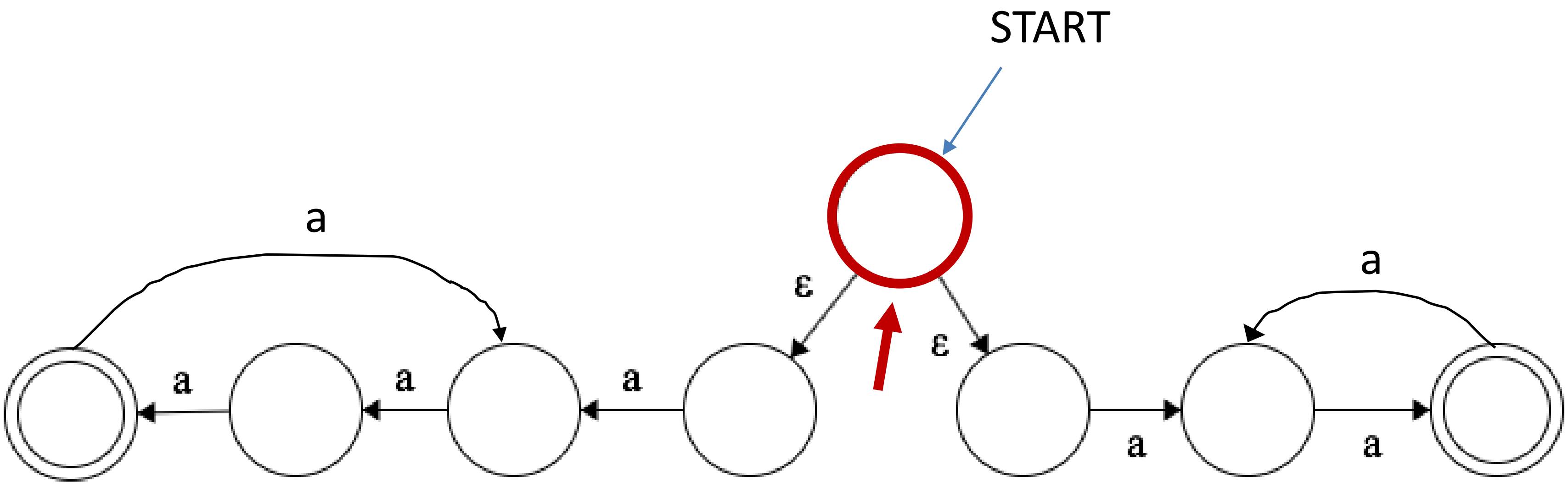
“aaaa”

“aaaaaa”

“aaaaaaaa”

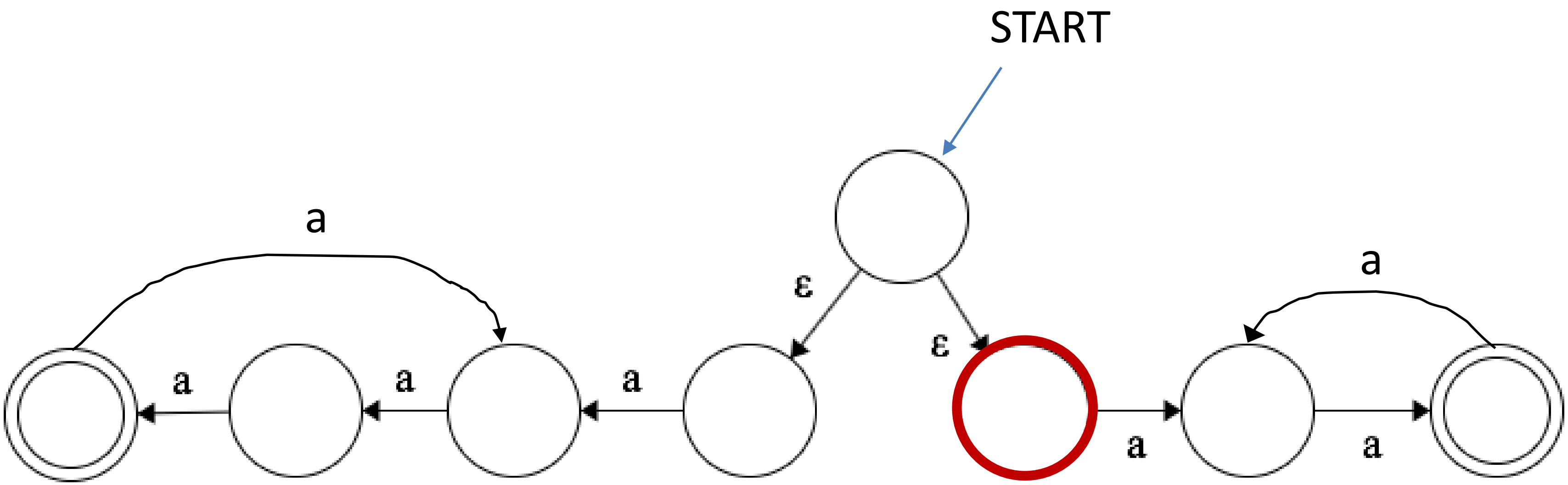
…

A (Nondeterministic) Example



“aaaa”

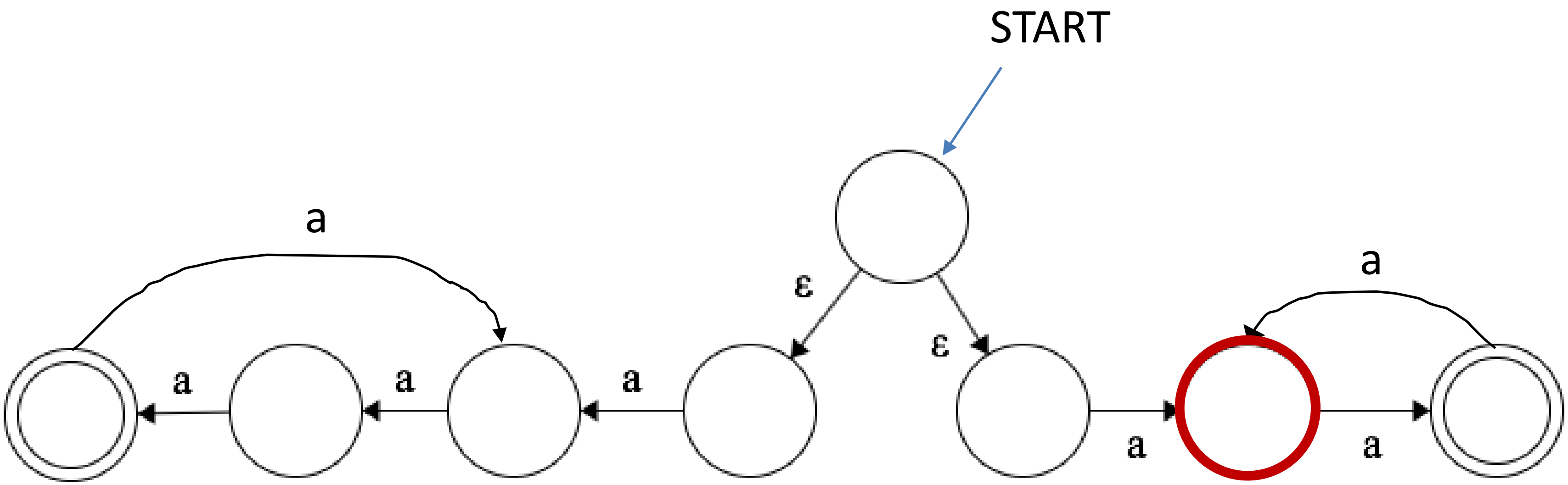
A (Nondeterministic) Example



Nondeterministically "guess" which path to take…must guess correctly! (automaton accepts if ***any*** path leads to accept state)

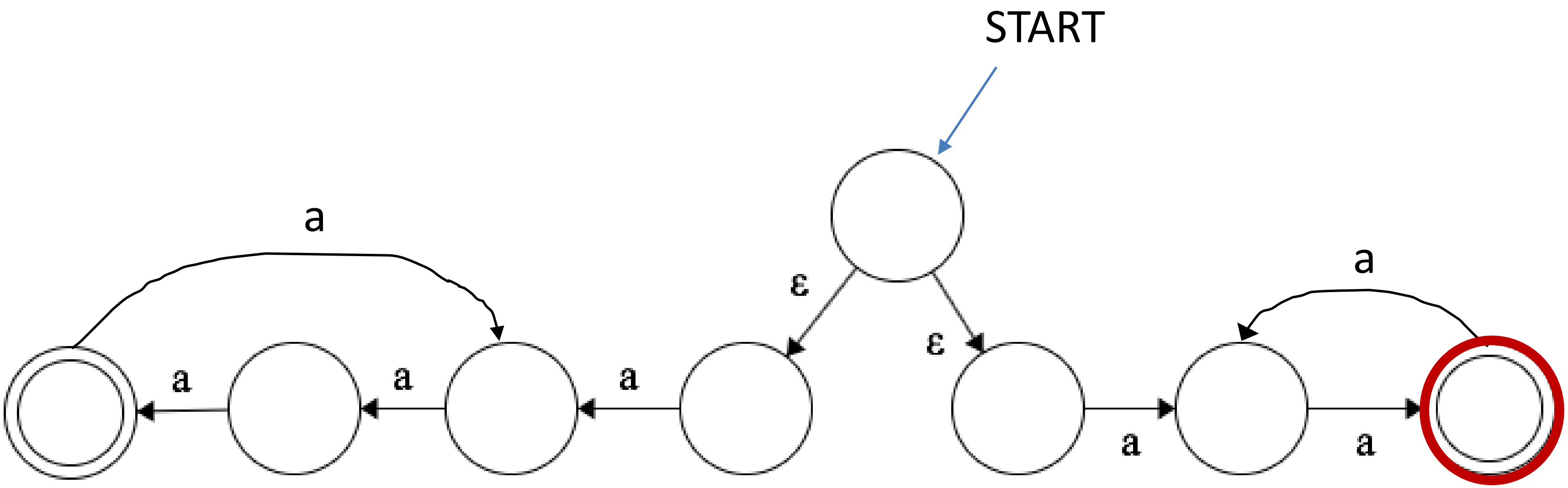
"aaaa"

A (Nondeterministic) Example



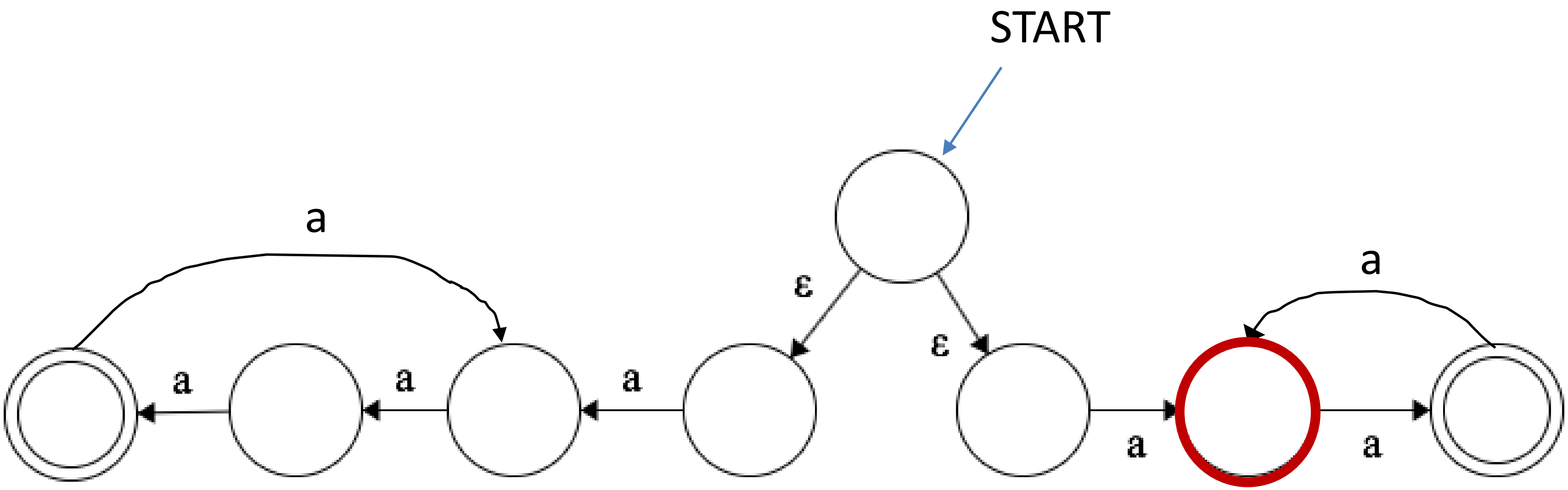
“aaaa”

A (Nondeterministic) Example



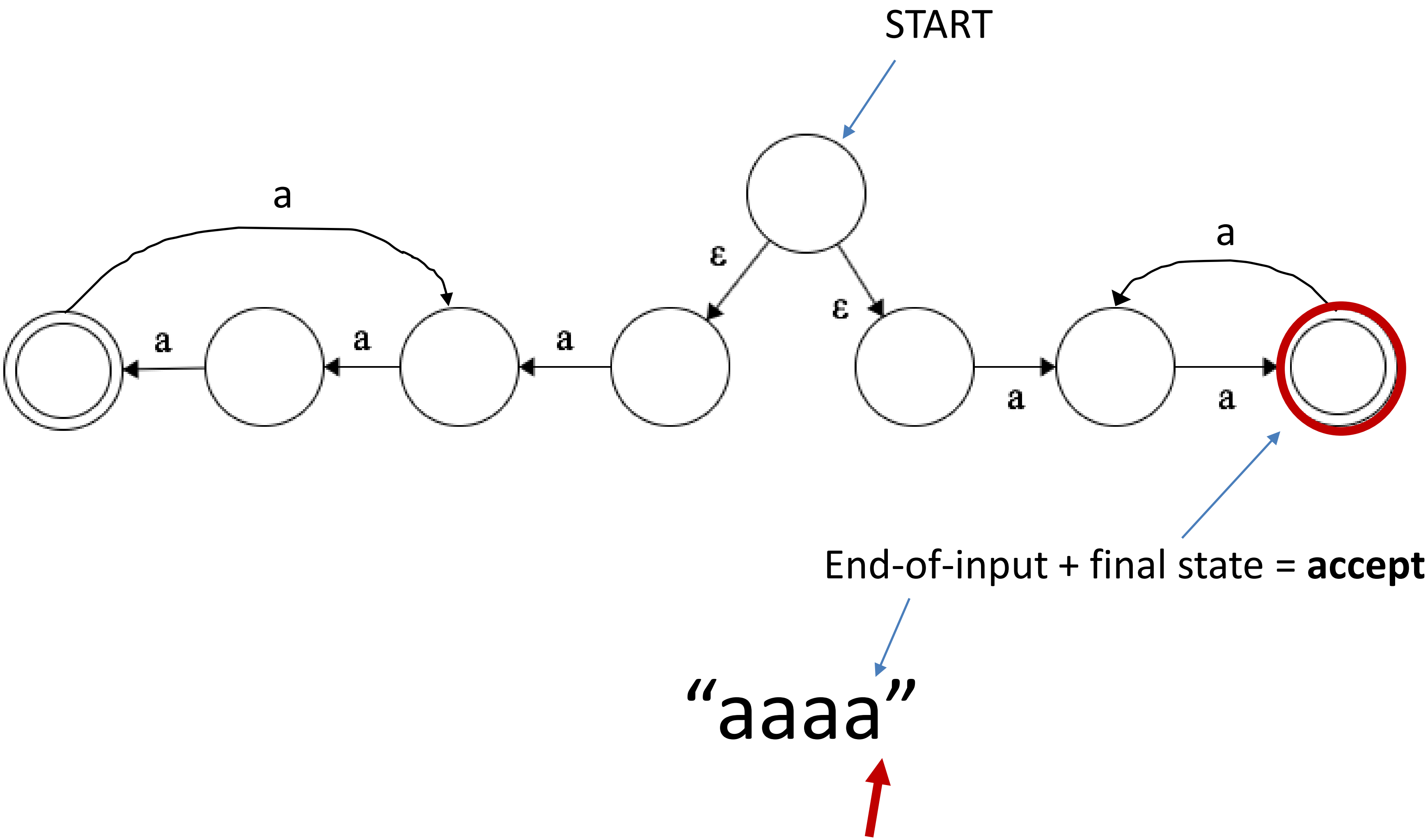
“aaaa”

A (Nondeterministic) Example



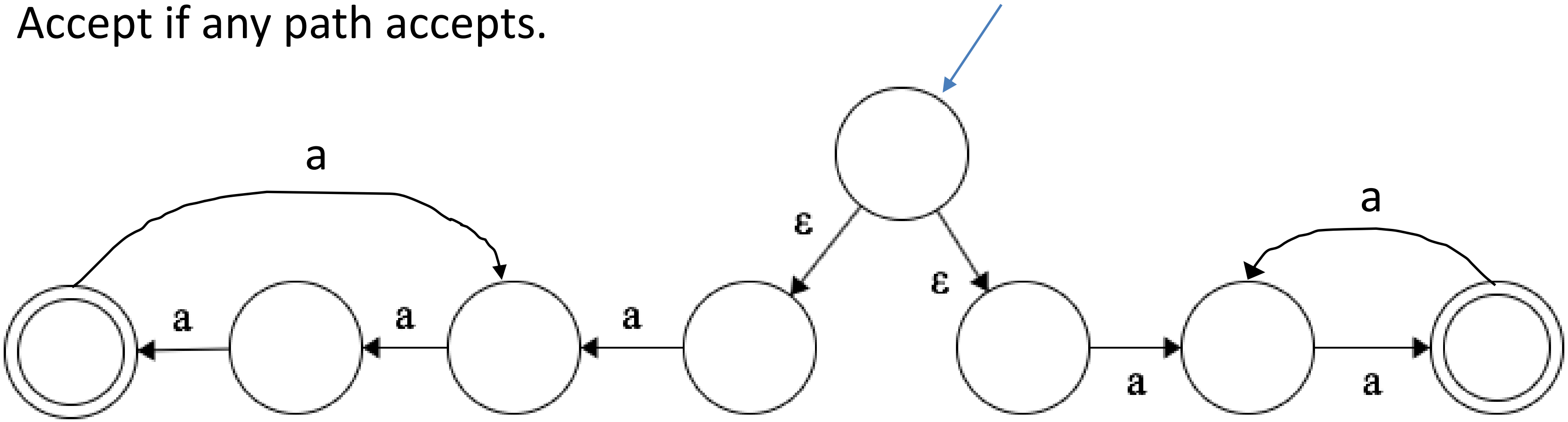
"aaaa"

A (Nondeterministic) Example



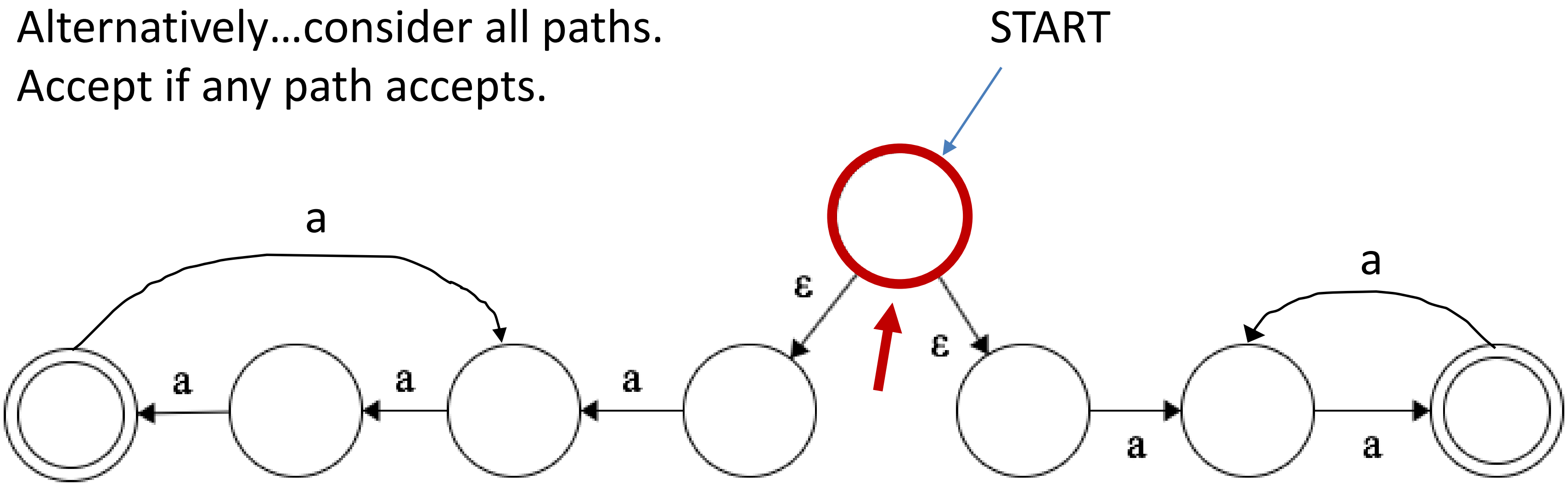
A (Nondeterministic) Example

Alternatively…consider all paths.
Accept if any path accepts.



A (Nondeterministic) Example

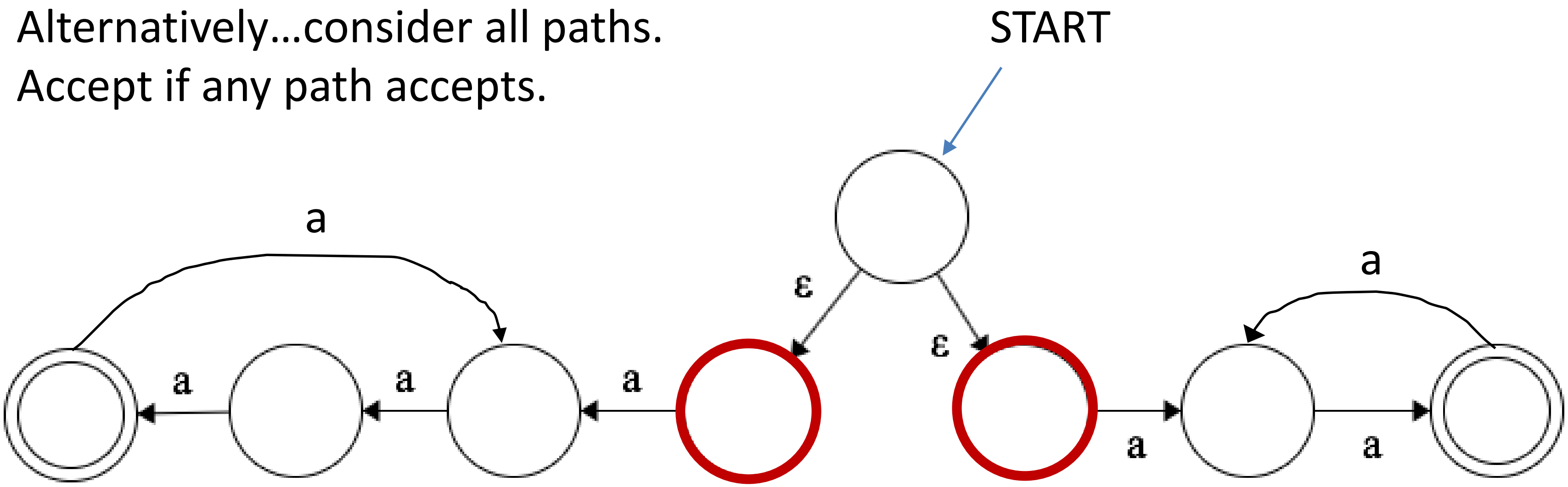
Alternatively…consider all paths.
Accept if any path accepts.



“aaaa”

A (Nondeterministic) Example

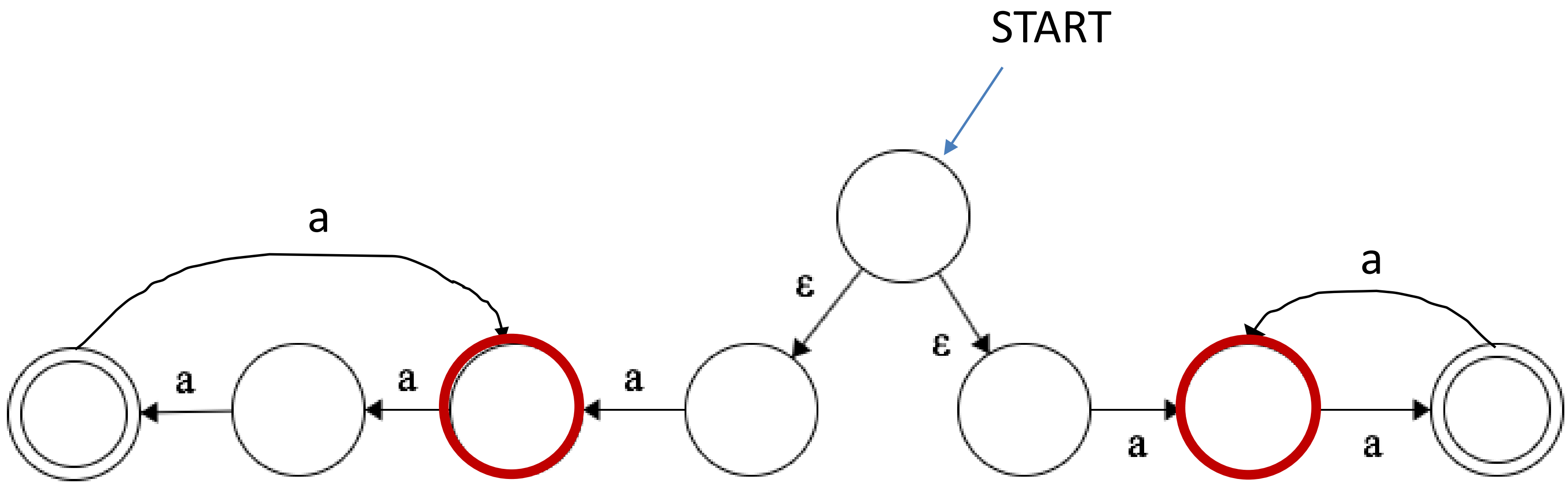
Alternatively…consider all paths.
Accept if any path accepts.



At nondeterministic transitions (epsilon or otherwise), explore all possible paths…

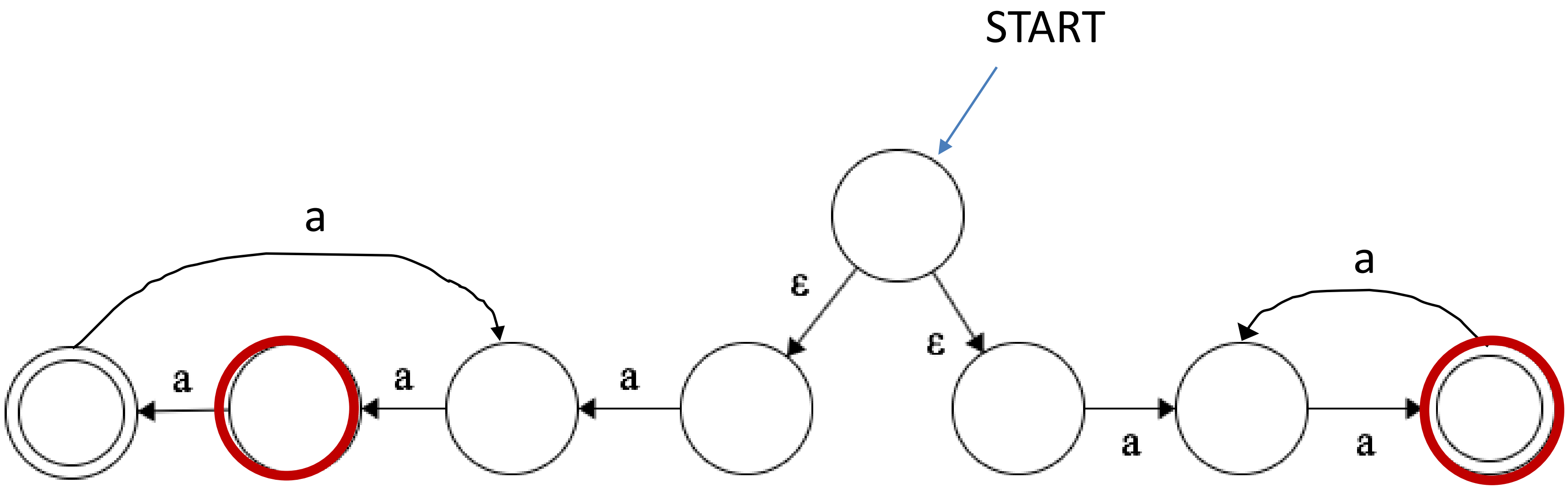
“aaaa”

A (Nondeterministic) Example



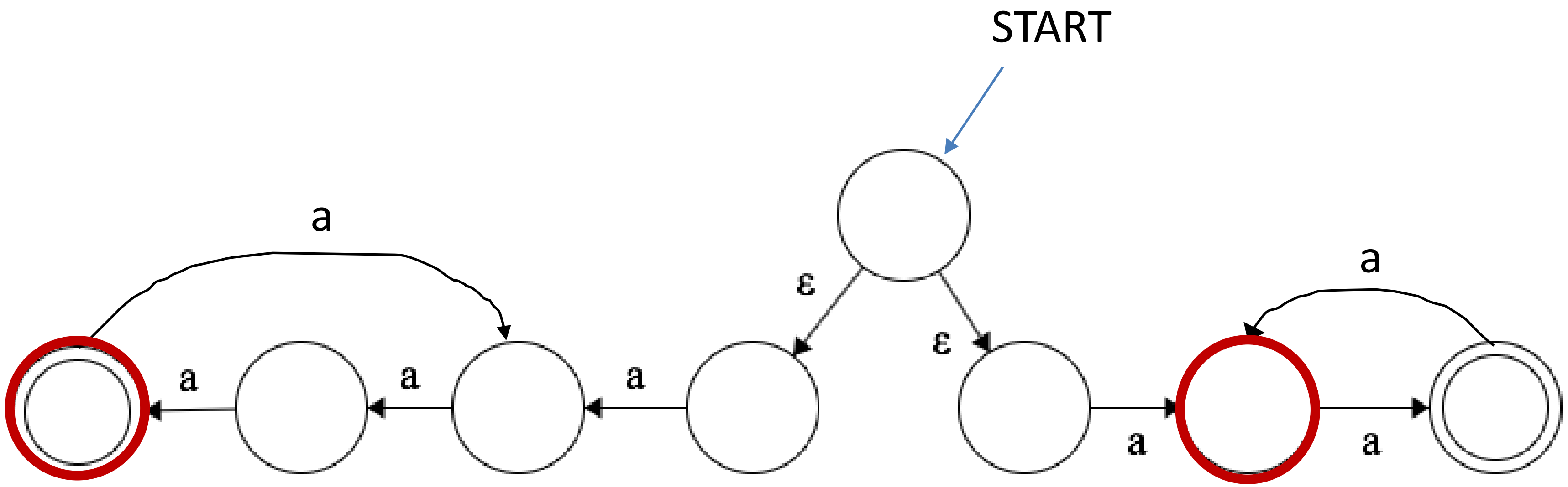
“aaaa”

A (Nondeterministic) Example



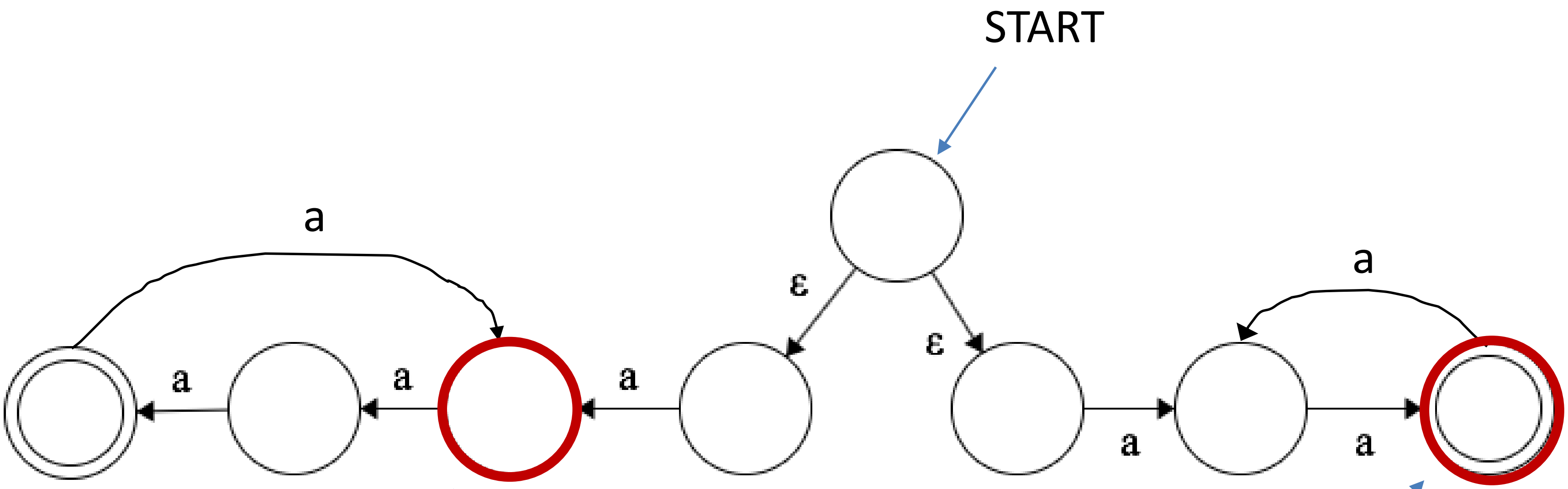
“aaaa”

A (Nondeterministic) Example



“aaaa”

A (Nondeterministic) Example



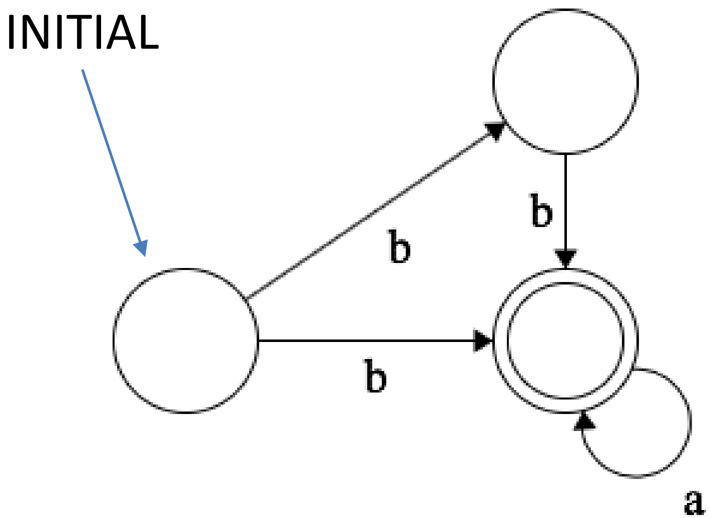
Not a final state but it doesn't matter…**accept** if at least one path accepts.

End-of-input + final state = **accept**

"aaaa"

NONDETERMINISM WITHOUT EPSILON

Nondeterminism without Epsilon



Which strings does this NFA accept?

Path Lower

“b”
“ba”
”baa”
”baa…”

Path Upper

“bb”
“bba”
“bbaa”
“bbaa…”

Summary: DFAs vs. NFAs

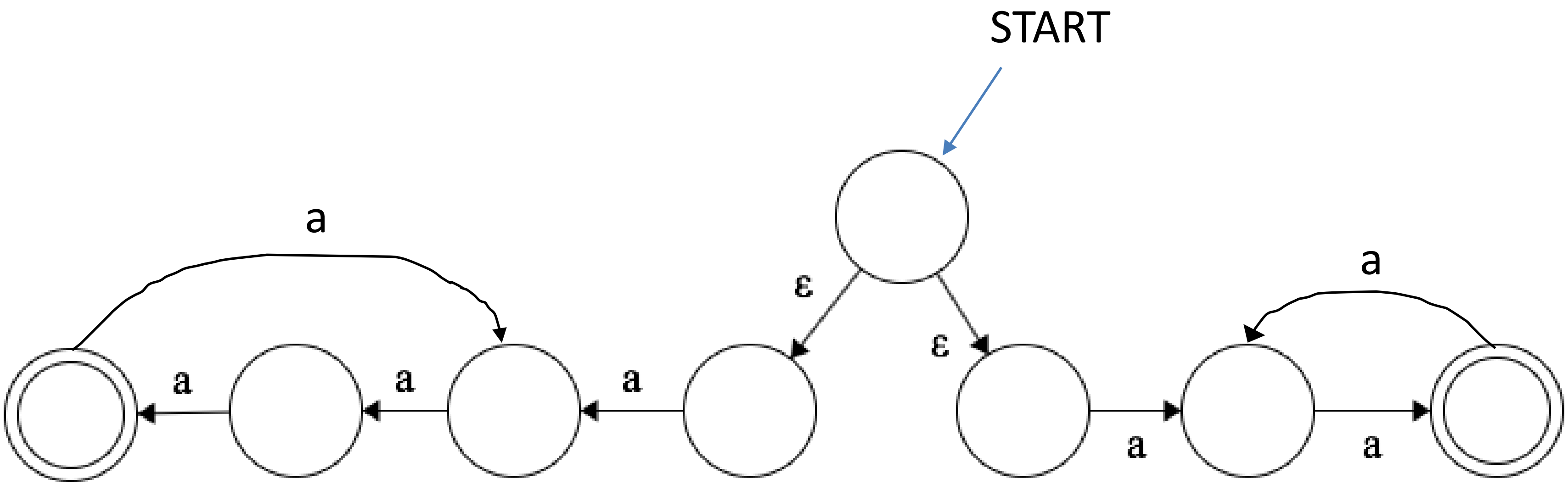
- DFA: A ***deterministic*** finite automaton is characterized by
  - ***Deterministic transition relation:*** For each state and input character, at most one next state
  - Exactly one ***initial state***
  - Typically not drawn with epsilon transitions

- NFA: A ***nondeterministic*** finite automaton is characterized by
  - A (potentially) ***nondeterministic transition relation***: For each state and input character, possibly more than one next state
  - Possibly many ***initial states***
  - Typically drawn using epsilon transitions, though an automaton may be nondeterministic even without epsilon

- Every DFA is trivially also an NFA.

NFA->DFA:
THE POWERSET ALGORITHM

Reducing NFAs to DFAs

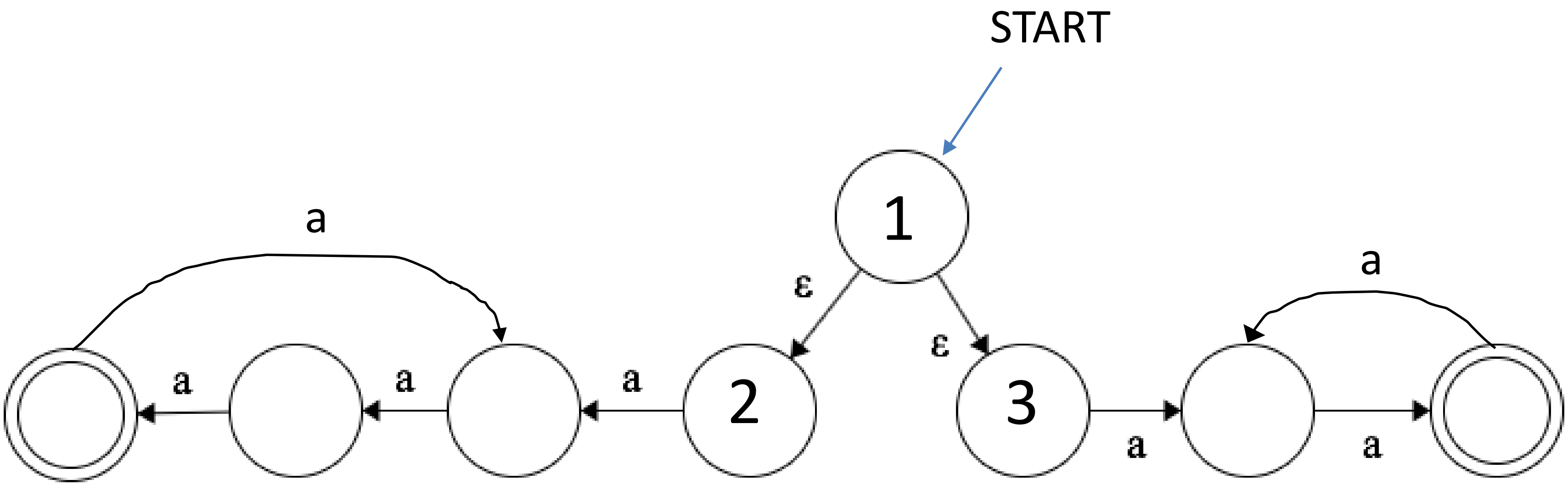


It might seem like NFAs are strictly more powerful than DFAs – they can guess!

In fact, they recognize exactly the same set of languages.

Skeptical?

Epsilon Closure



The "epsilon closure" of a state ***s*** is the set of states (including ***s***) reachable from ***s*** by following ϵ transitions (i.e., without reading any input).

E(s) = {s} ∪ $any\ states\ \epsilon - reachable\ from\ \mathbf{s}$

Example:

E(1) = {1, 2, 3}

NFA->DFA

Goal: Given an NFA ***N***, construct an equivalent DFA ***D***.

Equivalent? Recognizes exactly the same set of strings

Algorithm:

1. Let S = the states of ***N***. Draw the states S' of ***D*** to correspond to each possible subset of S.
   – For example, if S = {1, 2}, then S' = {∅, {1}, {2}, {1,2}}

2. Let INITIAL = the start states of ***N***. Mark as the *start state* of ***D*** the state in S' given by E(INITIAL).

3. Let FINAL = the accept states of ***N***. Mark as *accept states* of ***D*** ***any subset*** of states that contains a FINAL state.

NFA->DFA

Goal: Given an NFA ***N***, construct an equivalent DFA ***D***.

Equivalent? Recognizes exactly the same set of strings

Algorithm Continued:

4. Draw the transition arrows of ***D*** as follows:

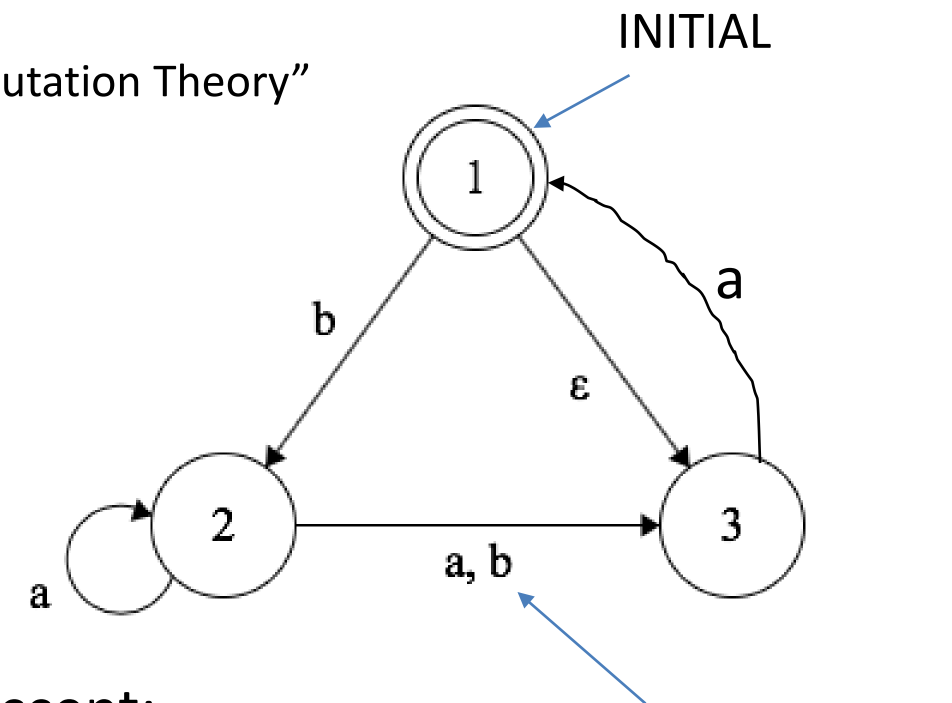
   For every state ***s*** in ***D*** and input character ***c***, draw an arrow from ***s*** to the state ***s'*** in D that includes every possible state reachable in ***N*** from any state in ***s*** by reading input character ***c***, **then** following $\epsilon - transitions.$

Recall that the states ***s*** of ***D*** correspond to ***sets of states*** of ***N.***

This rule is quite confusing…
Let's consider an example.

NFA->DFA Example

Example 1.42 in
Sipser's "Intro. to Computation Theory"



Read either a or b

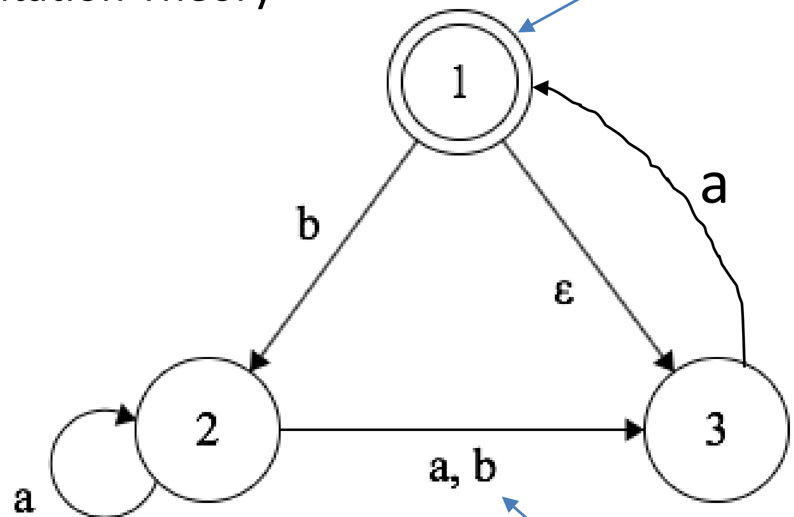
Does this NFA accept:

- "a"?
- "bb"?
- "baa"?
- "bba"?
- "baaaaaaaab"?
- "baaaaaaaaba"?

NFA->DFA Example

Example 1.42 in
Sipser's "Intro. to Computation Theory"

INITIAL

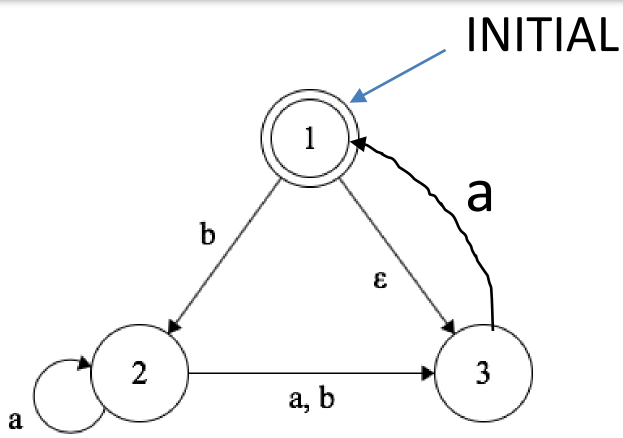


Read either a or b

Does this NFA accept:

- "a"? ***YES***
- "bb"? ***NO***
- "baa"? ***YES***
- "bba"? ***YES***
- "baaaaaaaab"? ***NO***
- "baaaaaaaaba"? ***YES***

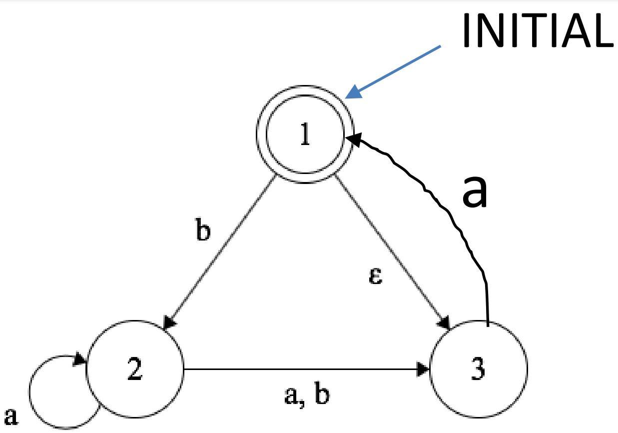
NFA->DFA Example



Algorithm:

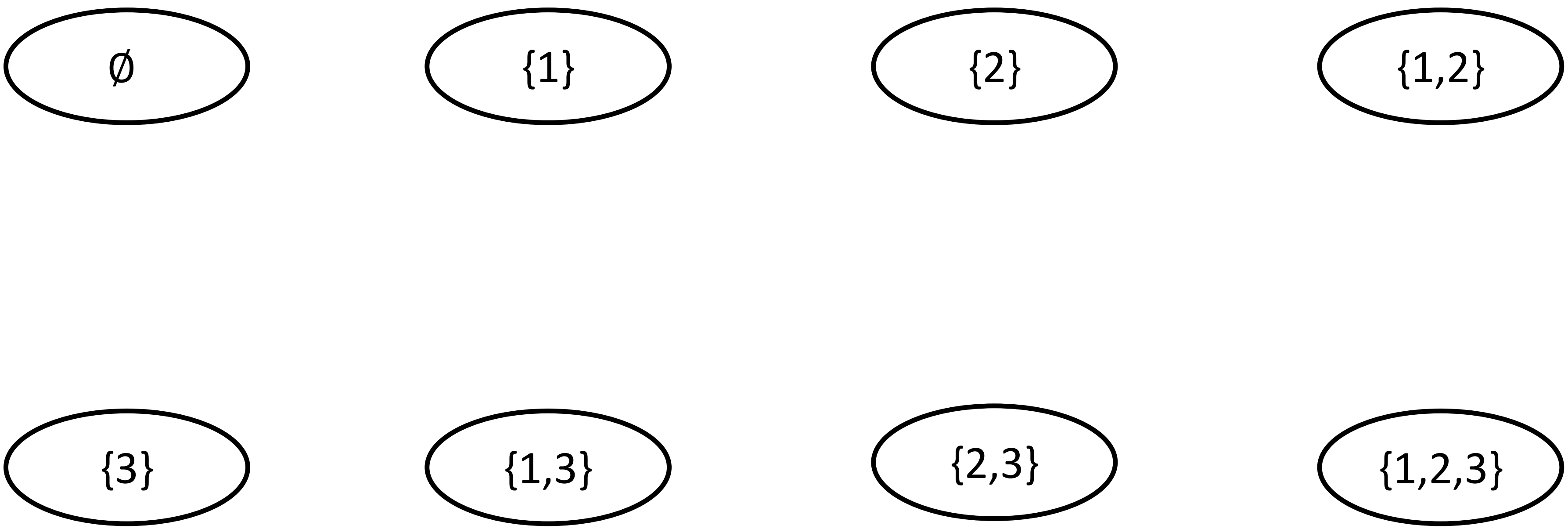
1. Let S = the states of ***N***. Draw the states S' of ***D*** to correspond to each possible subset of S.

NFA->DFA Example

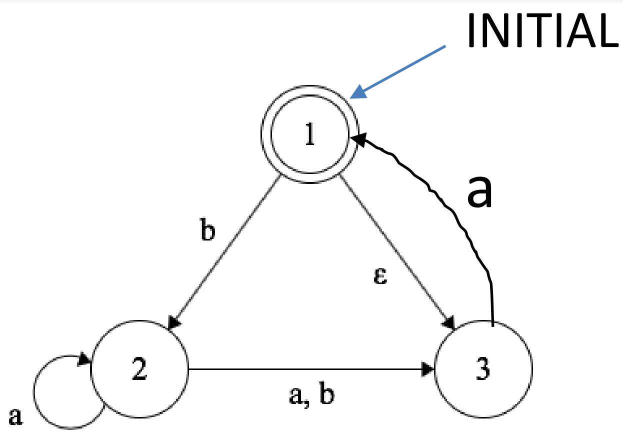


Algorithm:

1. Let S = the states of ***N***. Draw the states S' of ***D*** to correspond to each possible subset of S.

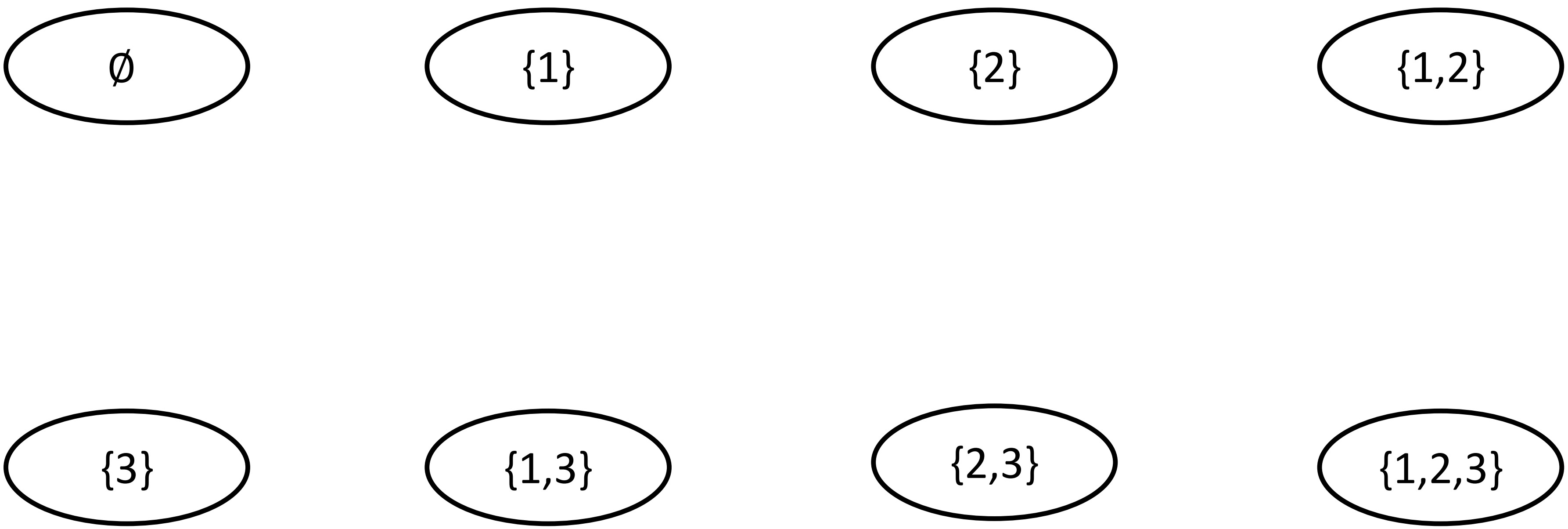


NFA->DFA Example

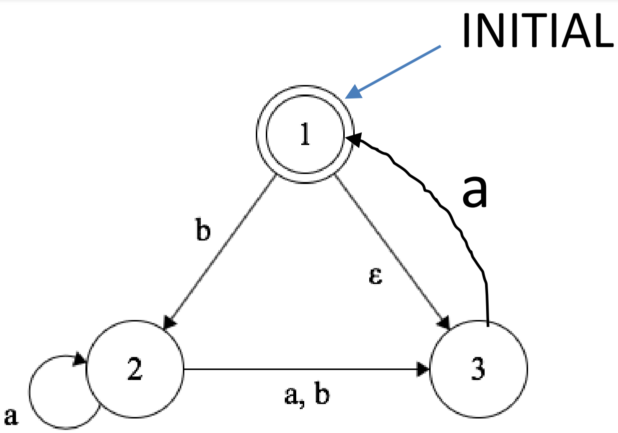


Algorithm:

1. …
2. Let INITIAL = the start states of ***N***. Mark as the *start state* of ***D*** the state in S' given by E(INITIAL).

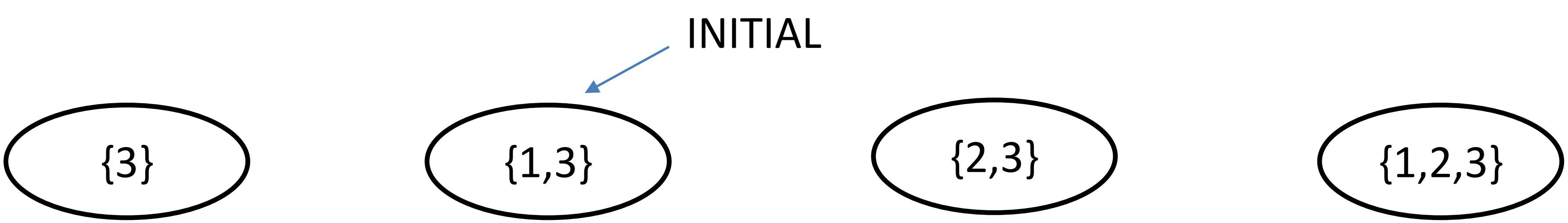


NFA->DFA Example

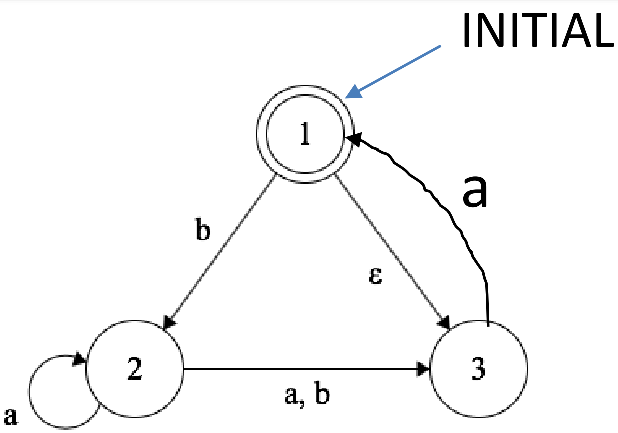


Algorithm:

1. …
2. Let INITIAL = the start states of ***N***. Mark as the *start state* of ***D*** the state in S’ given by E(INITIAL).

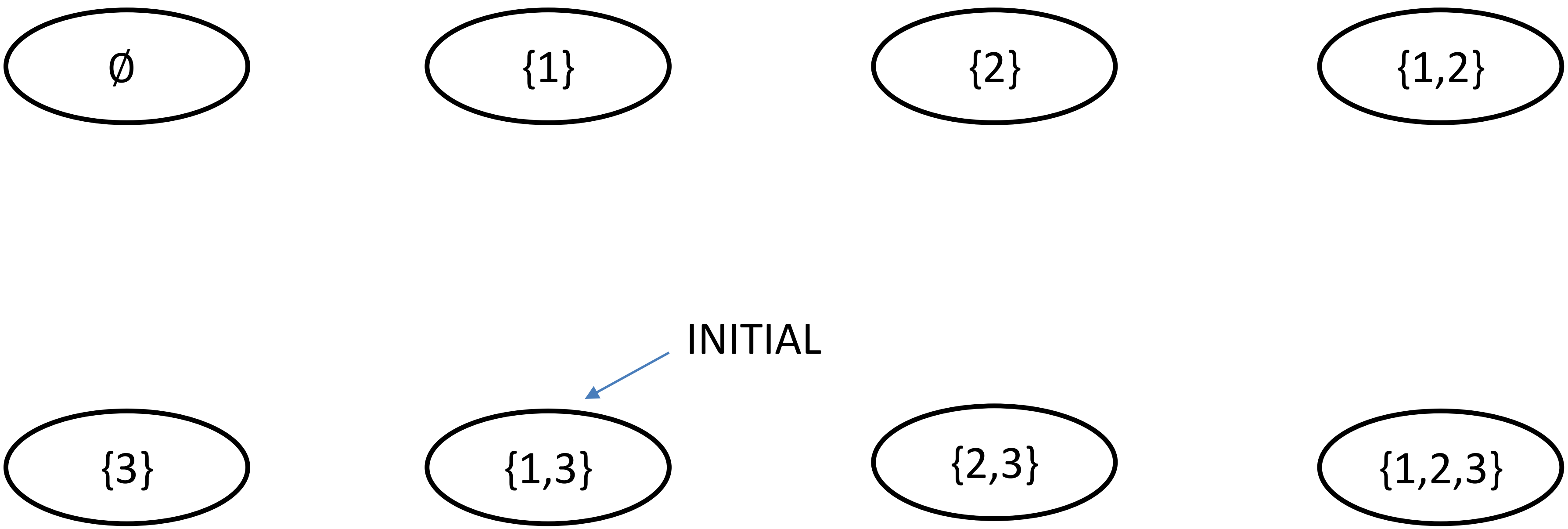


NFA->DFA Example

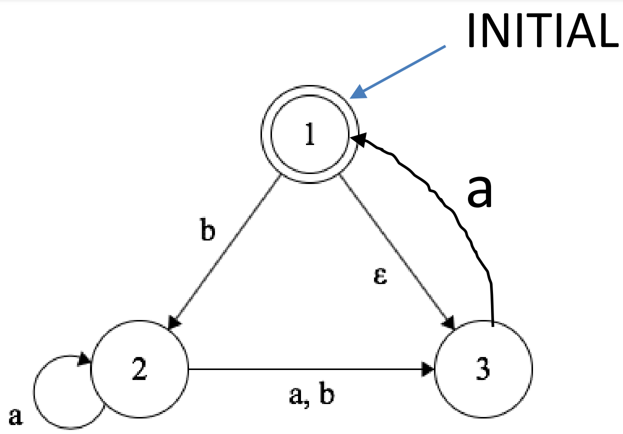


Algorithm:

2. …
3. Let FINAL = the accept states of ***N***. Mark as *accept states* of ***D*** ***any subset*** of states that contains a FINAL state.

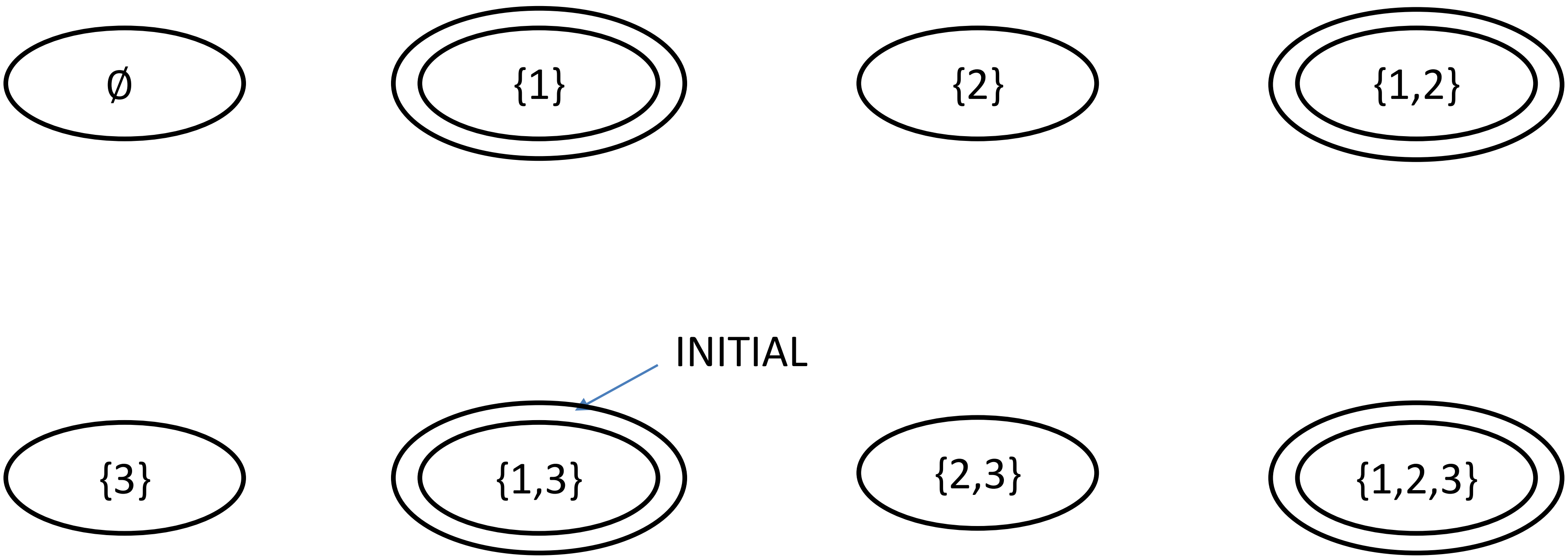


NFA->DFA Example

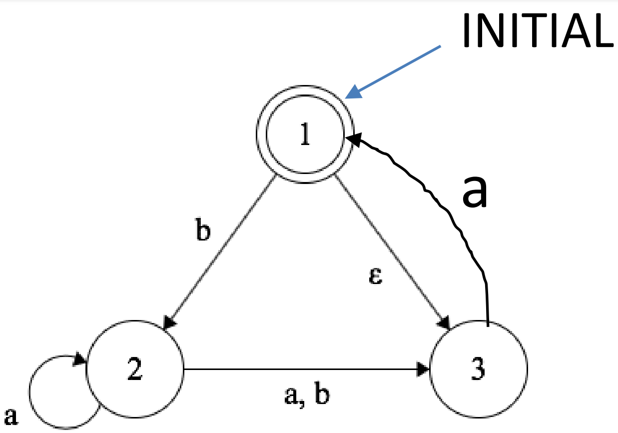


Algorithm:

2. …
3. Let FINAL = the accept states of ***N***. Mark as *accept states* of ***D*** ***any subset*** of states that contains a FINAL state.



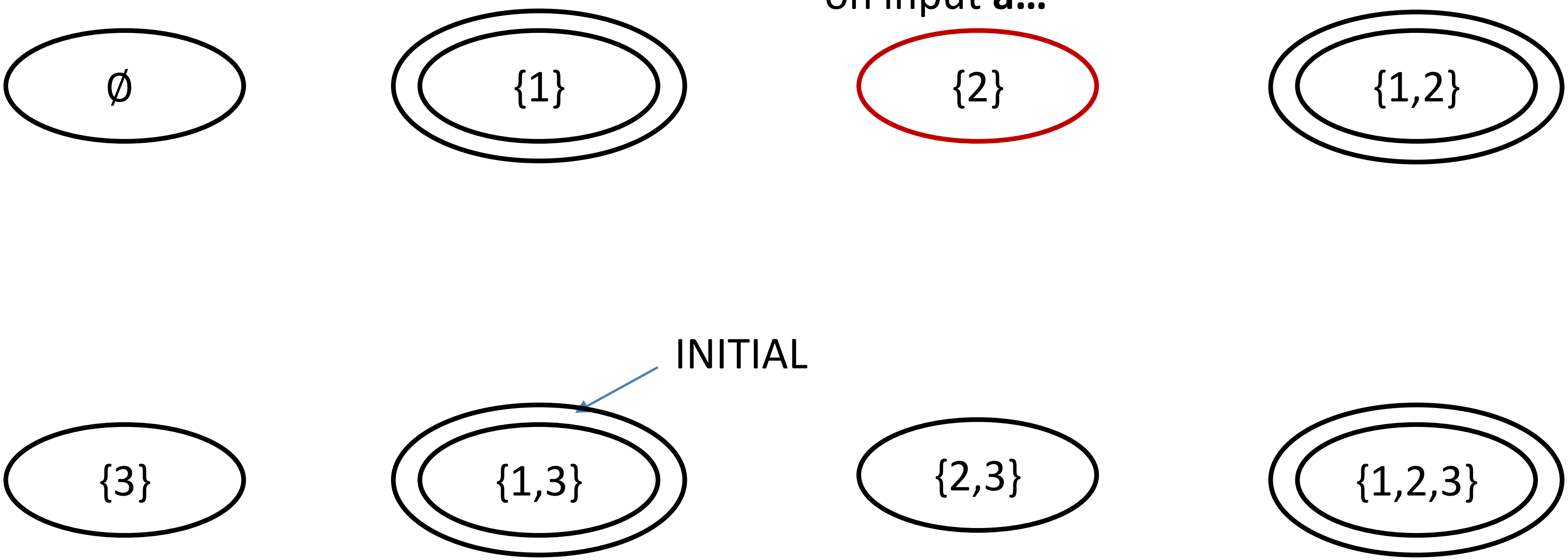
NFA->DFA Example



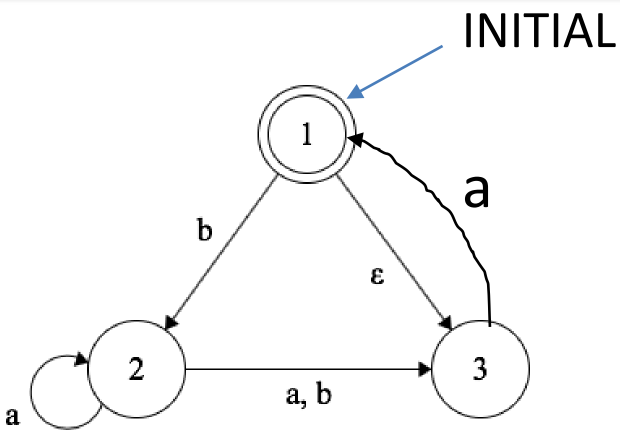
Algorithm:

4. For every state ***s*** in ***D*** and input character ***c***, draw an arrow from ***s*** to the state ***s'*** in D that includes every possible state reachable in ***N*** from any state in ***s*** by reading input character ***c***, **then** following $\epsilon - transitions$.

Consider state {2}
on input **a…**



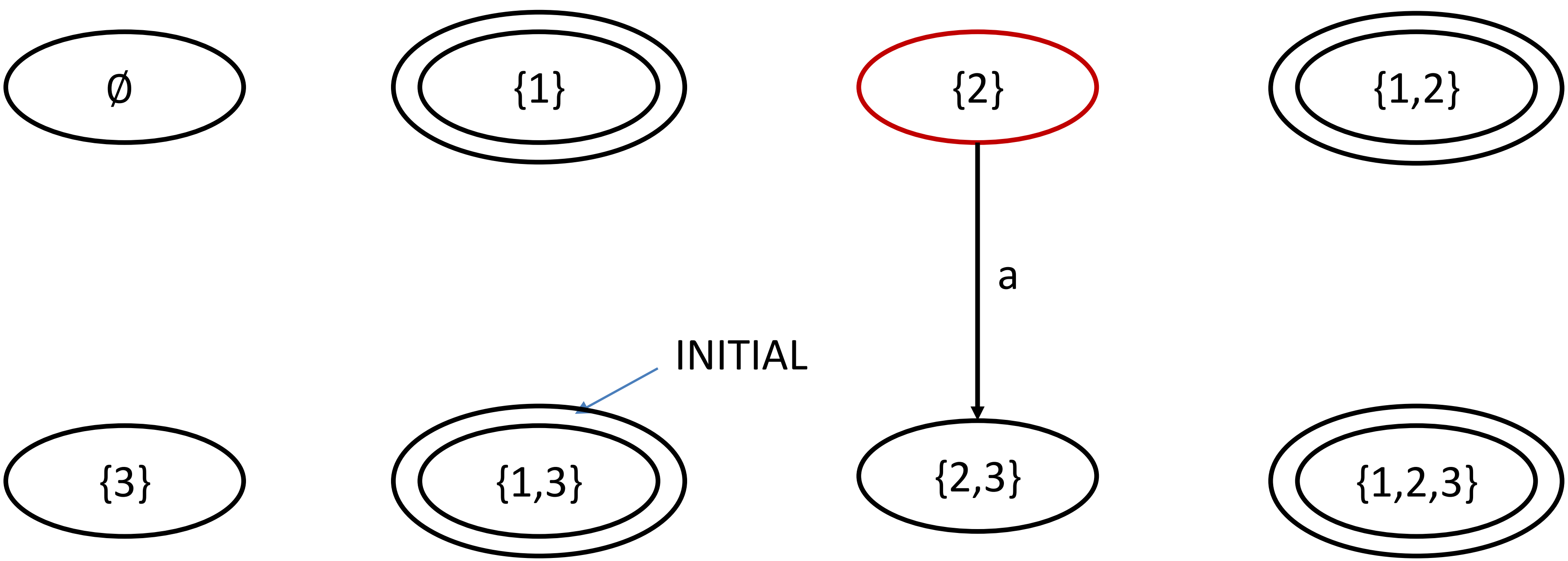
NFA->DFA Example



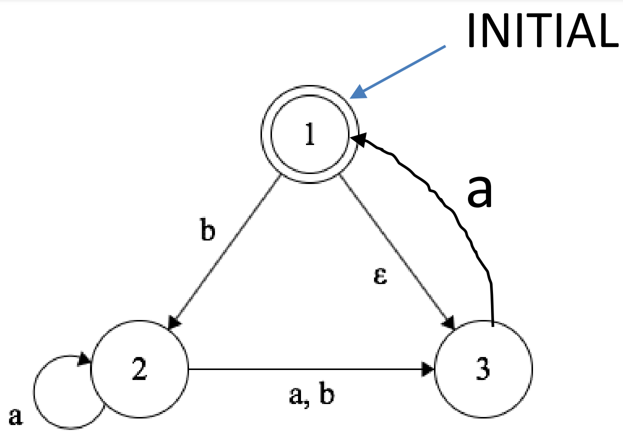
Algorithm:

4. For every state ***s*** in ***D*** and input character ***c***, draw an arrow from ***s*** to the state ***s'*** in D that includes every possible state reachable in ***N*** from any state in ***s*** by reading input character ***c***, **then** following $\epsilon - transitions$.

Now on input **b…**

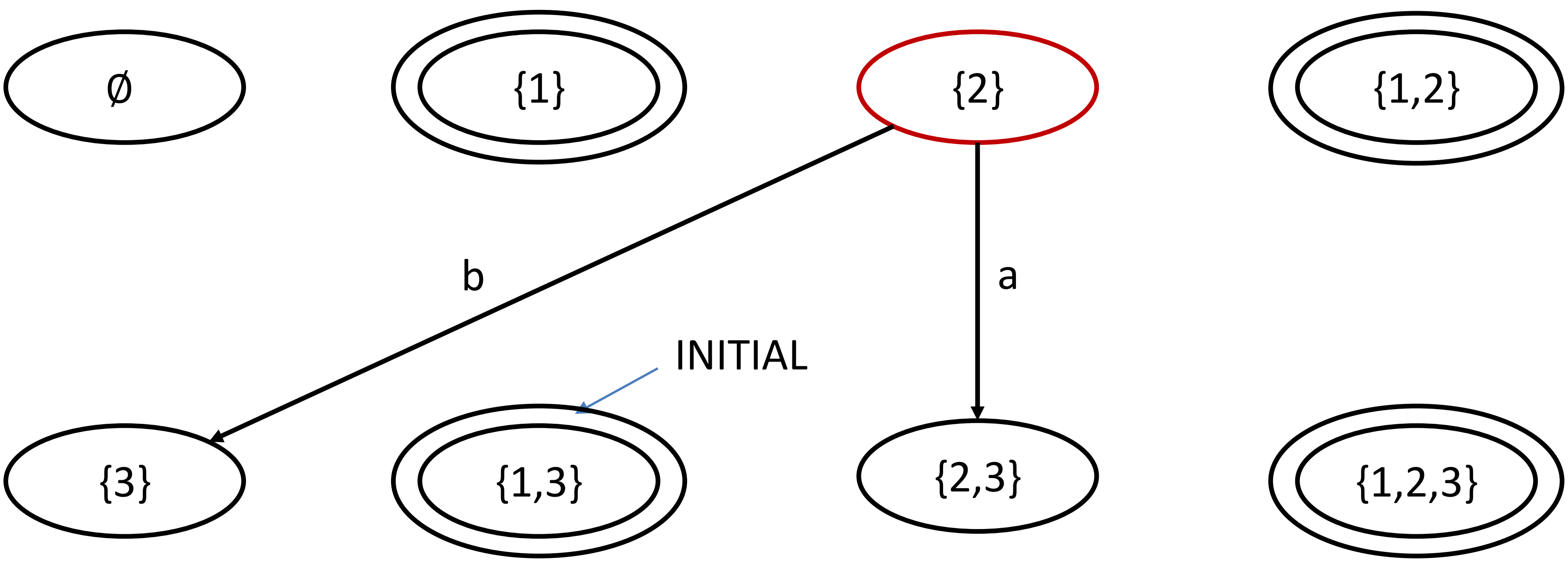


NFA->DFA Example

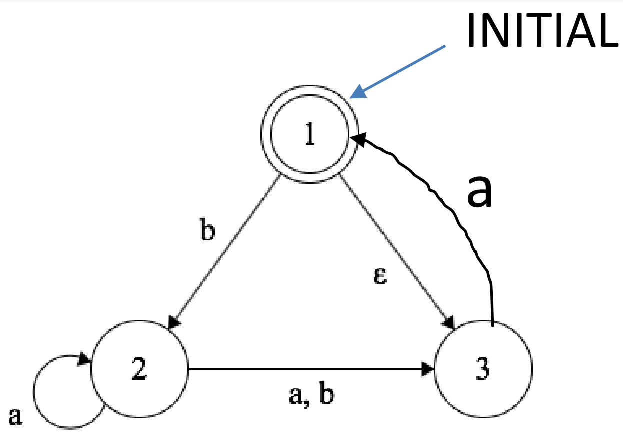


Algorithm:

4. For every state ***s*** in ***D*** and input character ***c***, draw an arrow from ***s*** to the state ***s'*** in D that includes every possible state reachable in ***N*** from any state in ***s*** by reading input character ***c***, **then** following $\epsilon - transitions$.

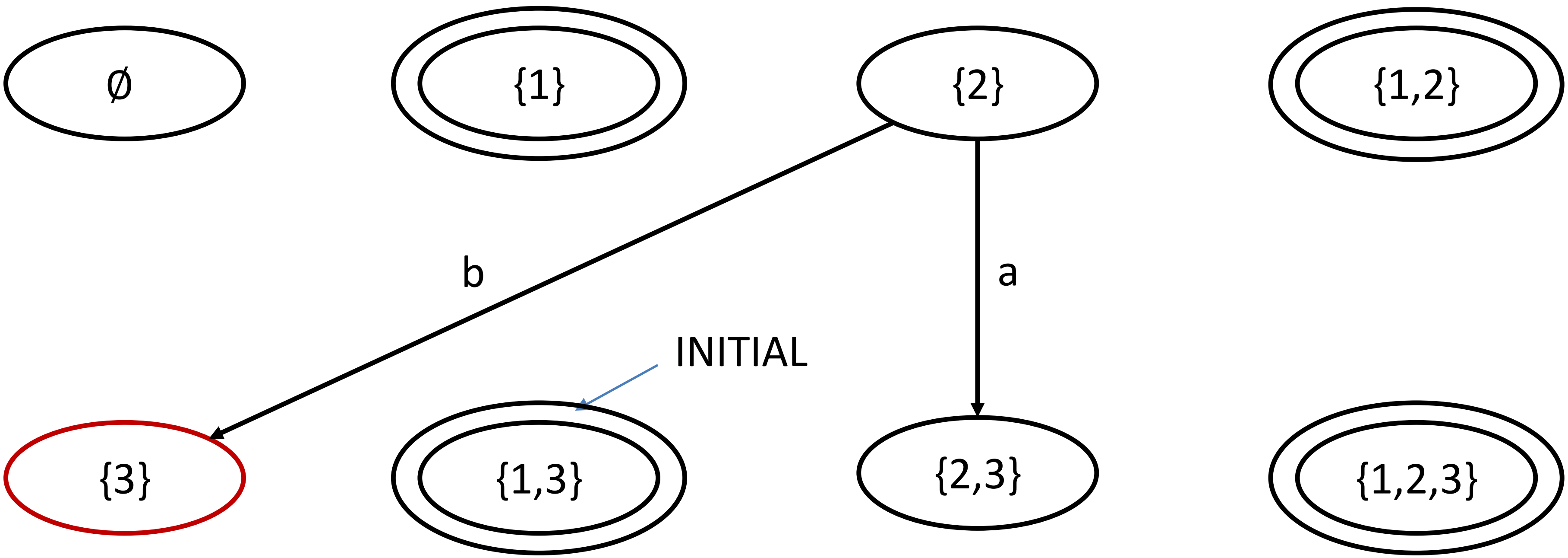


NFA->DFA Example

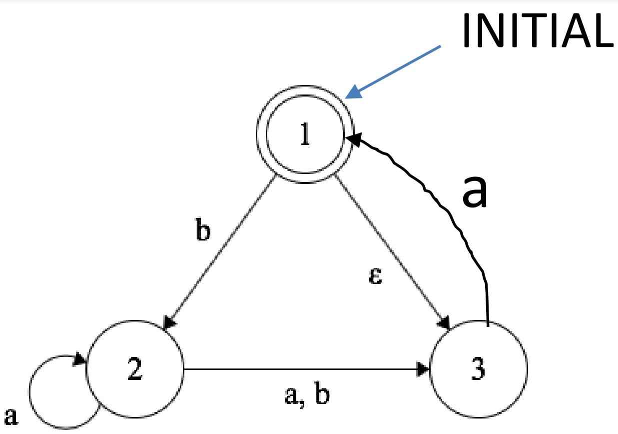


Algorithm:

4. For every state ***s*** in ***D*** and input character ***c***, draw an arrow from ***s*** to the state ***s'*** in D that includes every possible state reachable in ***N*** from any state in ***s*** by reading input character ***c***, **then** following $\epsilon - transitions$.

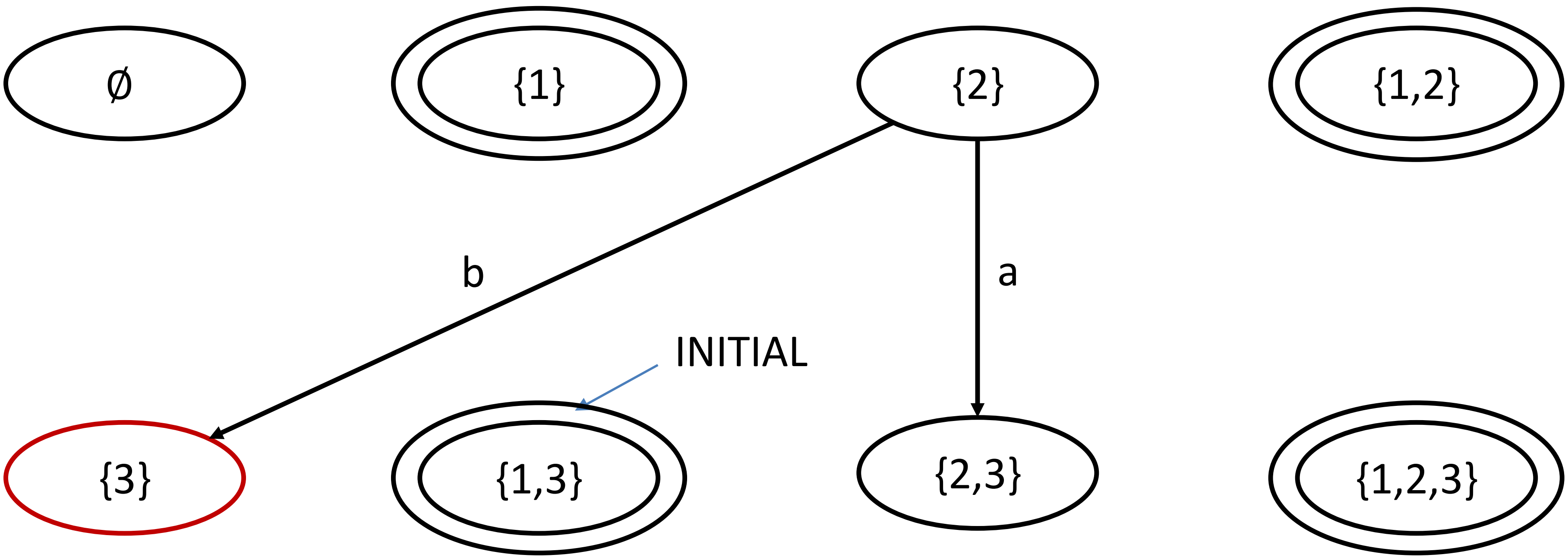


NFA->DFA Example

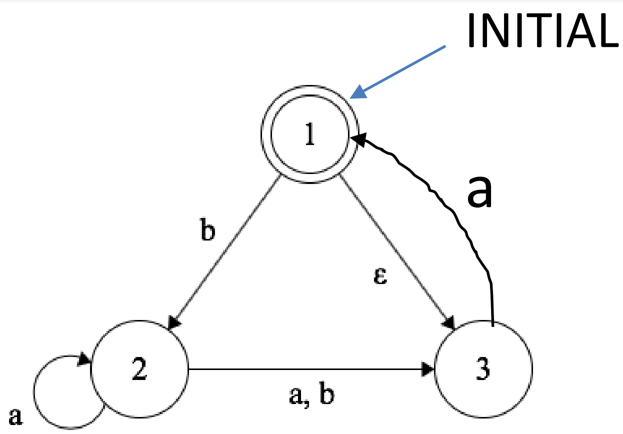


Transition table for NFA to the left:

State\Char	a	b
1		
2	{2,3}	{3}
3		

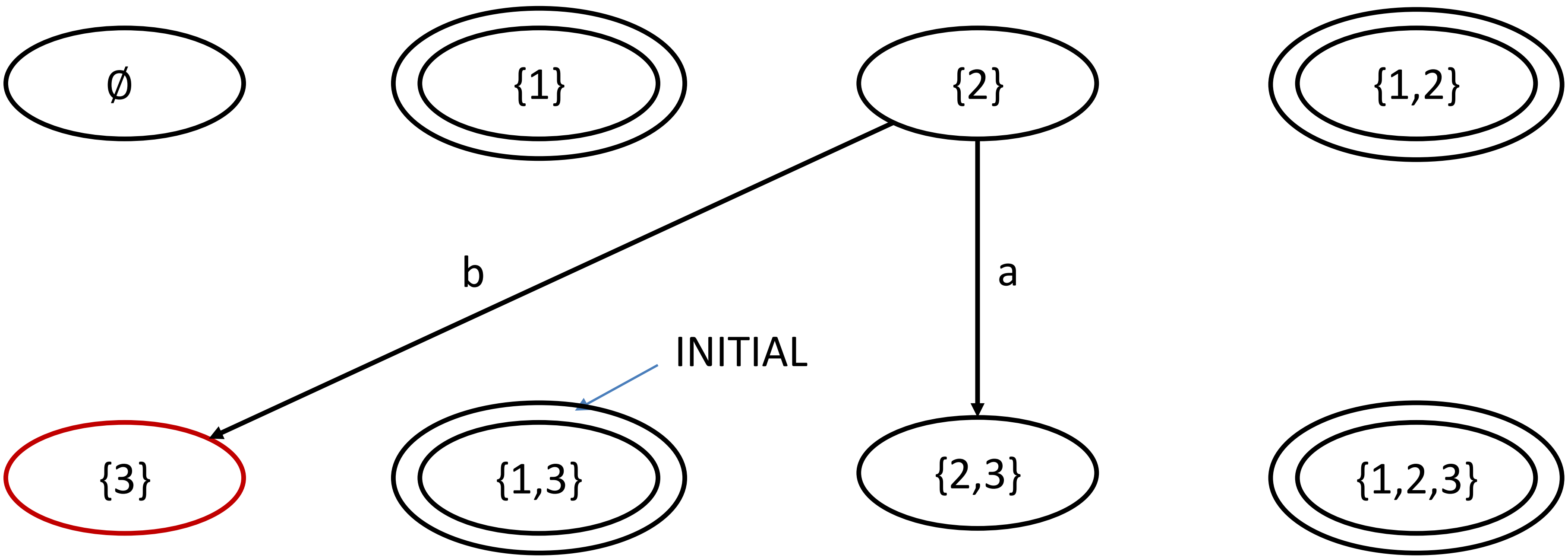


NFA->DFA Example

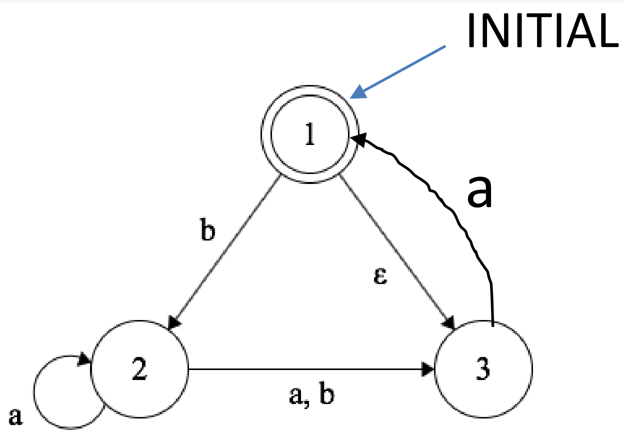


Transition table for NFA to the left:

State\Char	a	b
1		
2	{2,3}	{3}
3	{1,3}	

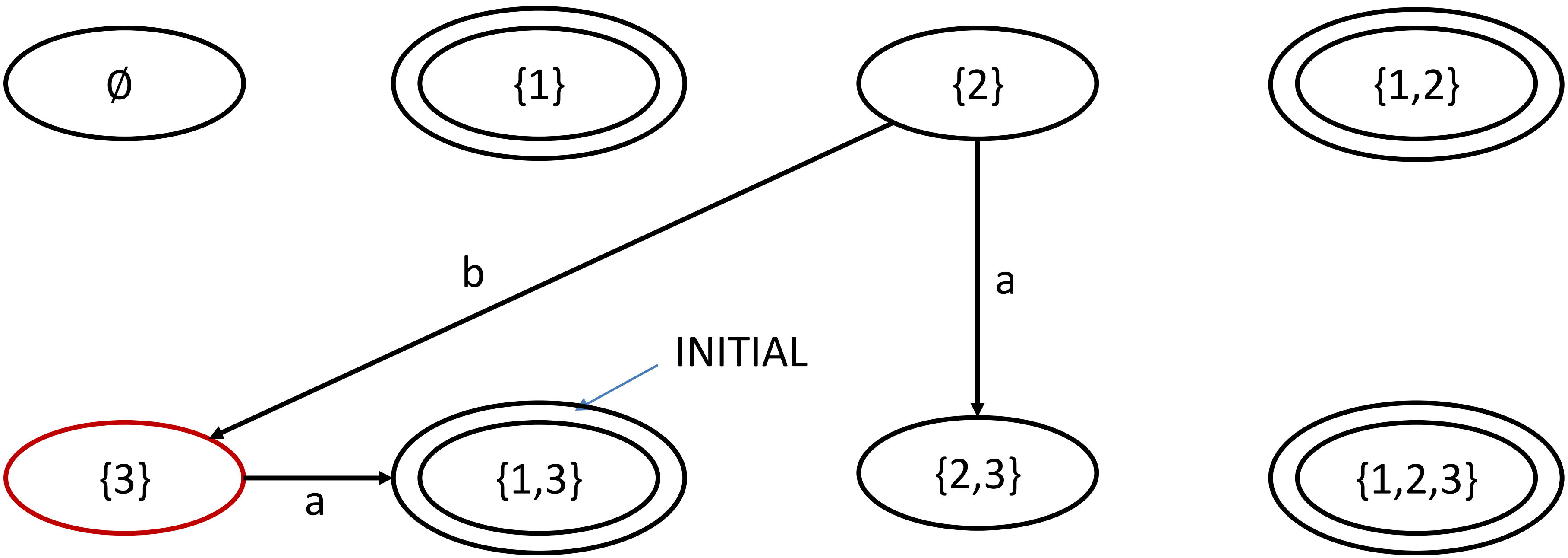


NFA->DFA Example

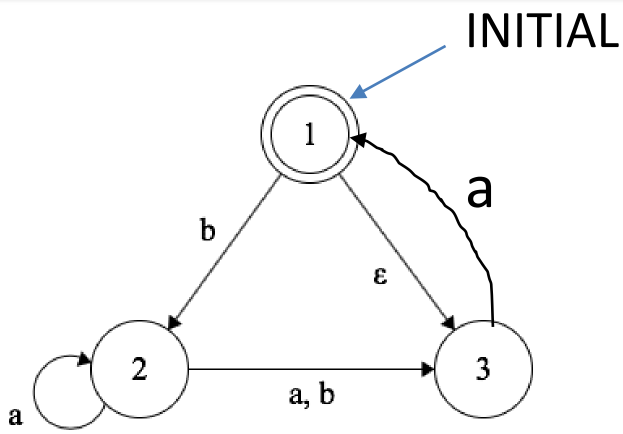


Transition table for NFA to the left:

State\Char	a	b
1		
2	{2,3}	{3}
3	{1,3}	



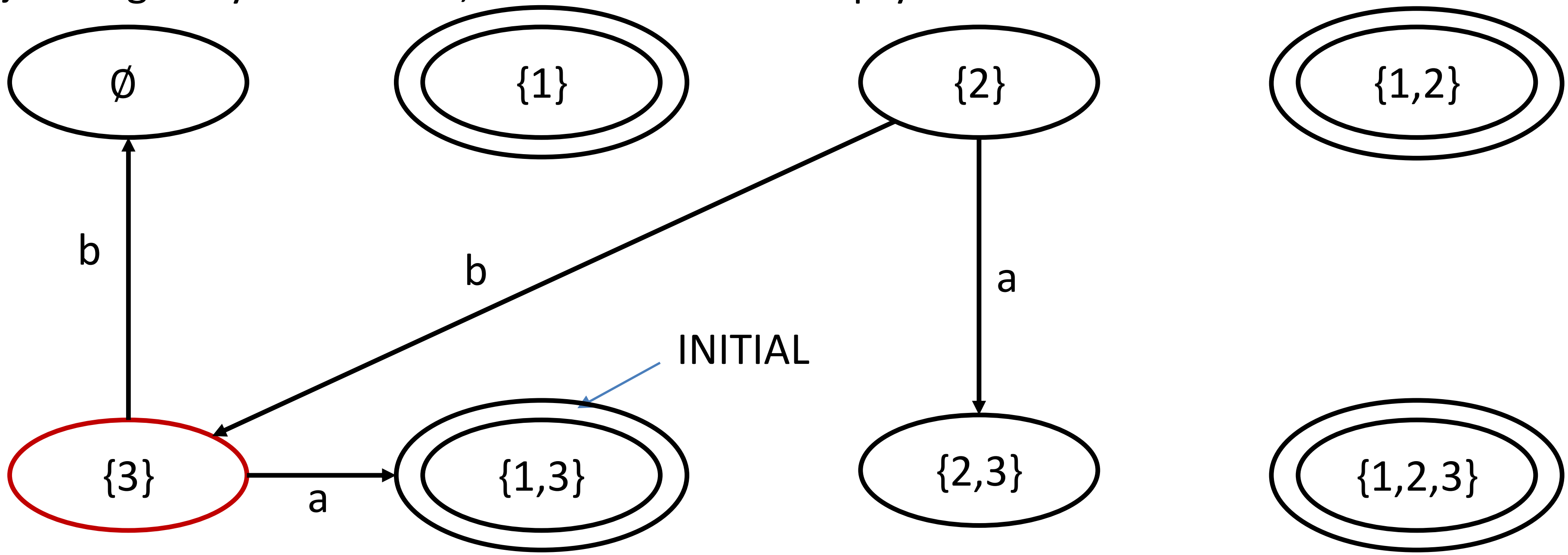
NFA->DFA Example



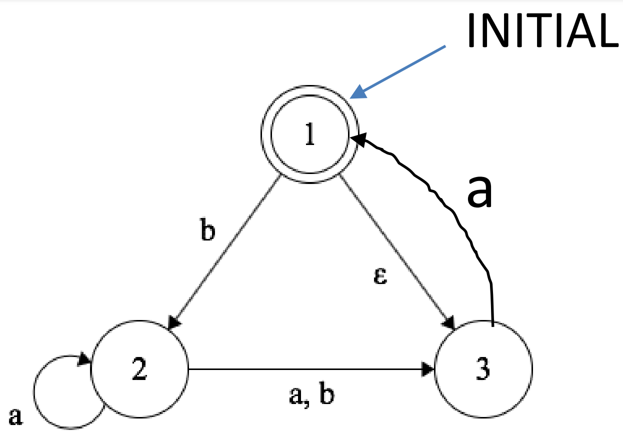
Transition table for NFA to the left:

State\Char	a	b
1		
2	{2,3}	{3}
3	{1,3}	{}

{3} can't go anywhere on **b**, so send it to the empty state Ø!

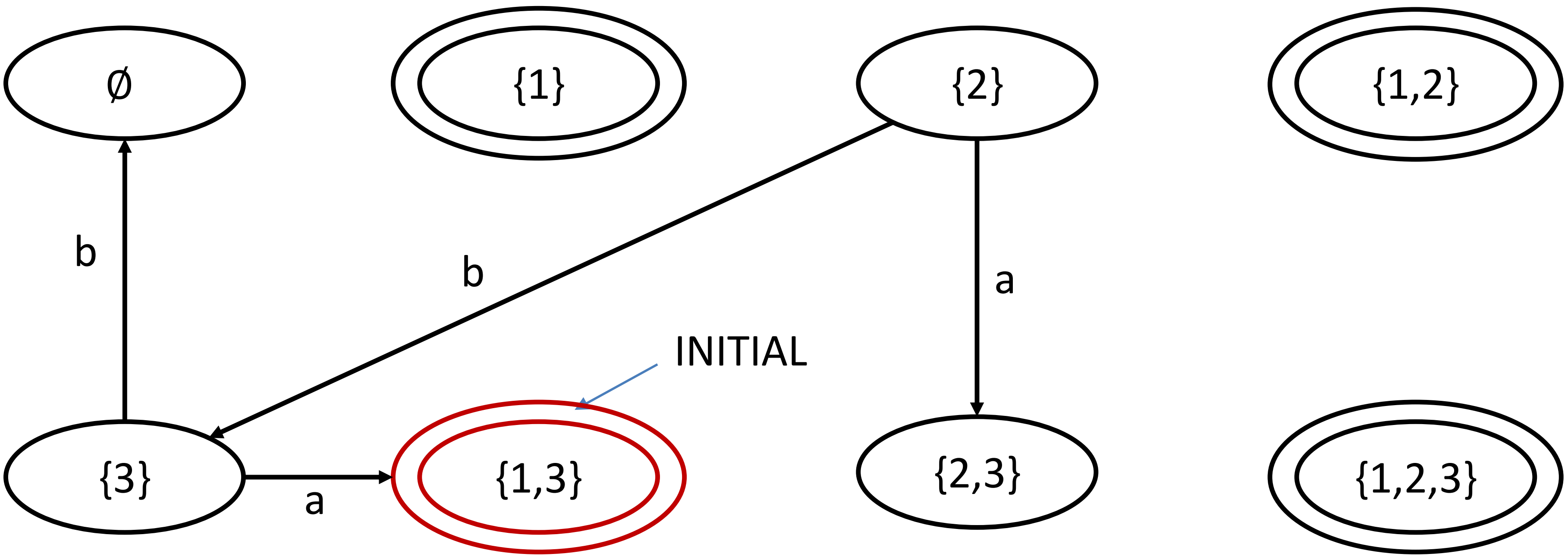


NFA->DFA Example

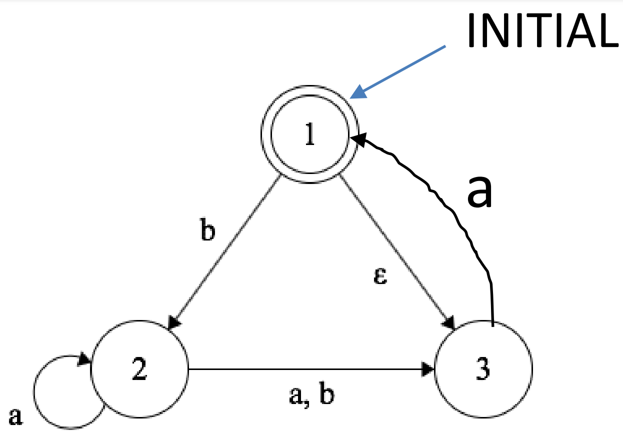


Transition table for NFA to the left:

State\Char	a	b
1	{}	{2}
2	{2,3}	{3}
3	{1,3}	{}

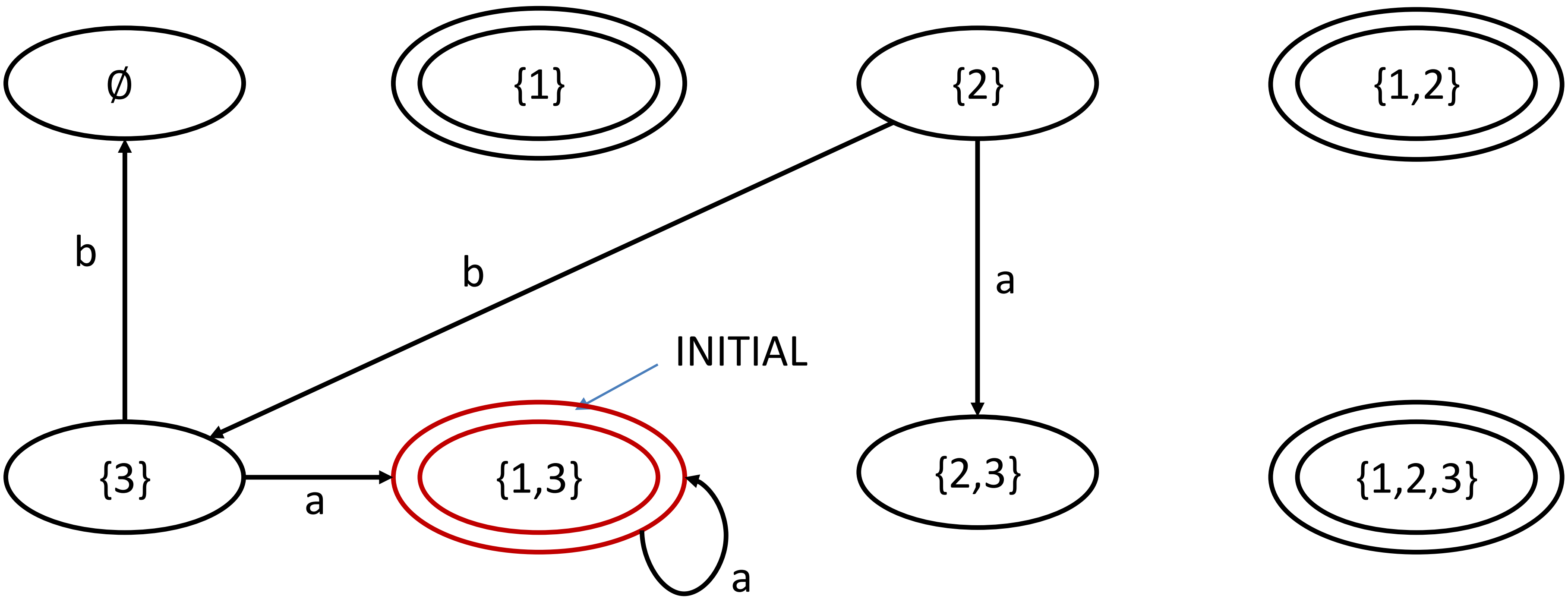


NFA->DFA Example

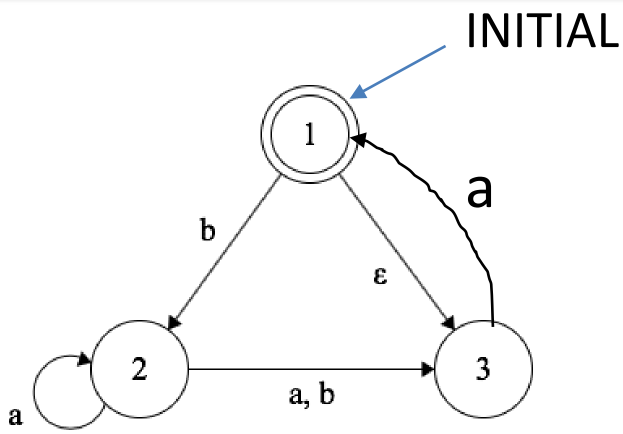


Transition table for NFA to the left:

State\Char	a	b
1	{}	{2}
2	{2,3}	{3}
3	{1,3}	{}

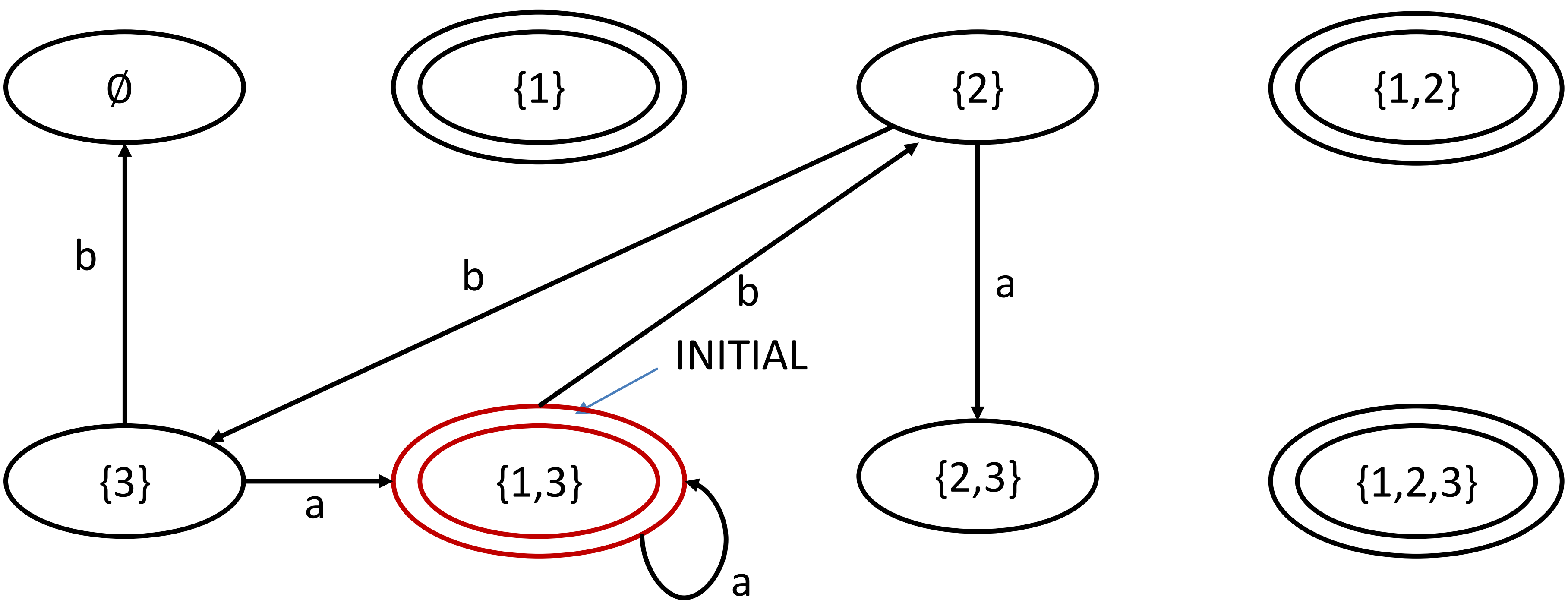


NFA->DFA Example

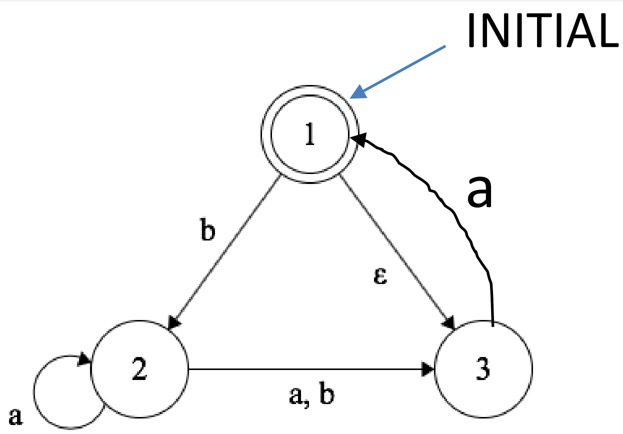


Transition table for NFA to the left:

State\Char	a	b
1	{}	{2}
2	{2,3}	{3}
3	{1,3}	{}

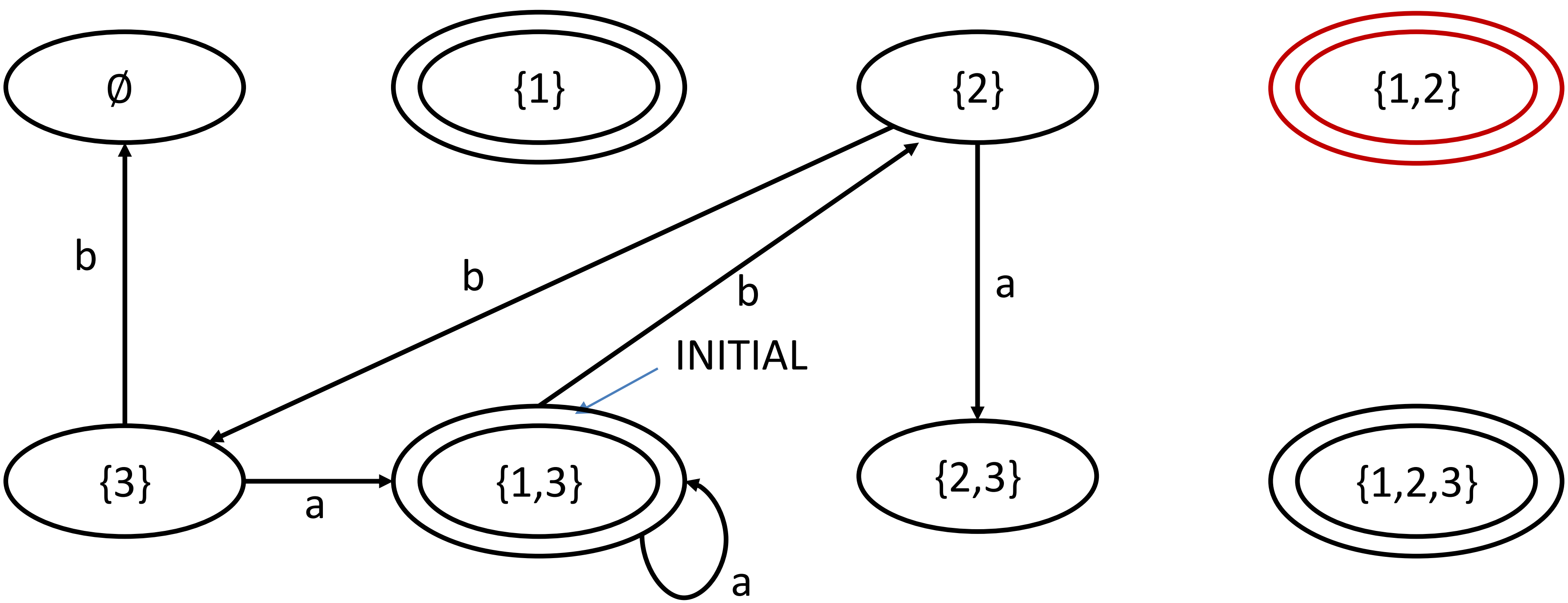


NFA->DFA Example

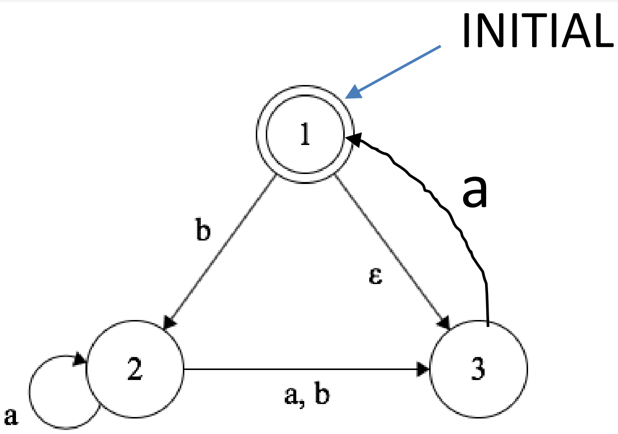


Transition table for NFA to the left:

State\Char	a	b
1	{}	{2}
2	{2,3}	{3}
3	{1,3}	{}

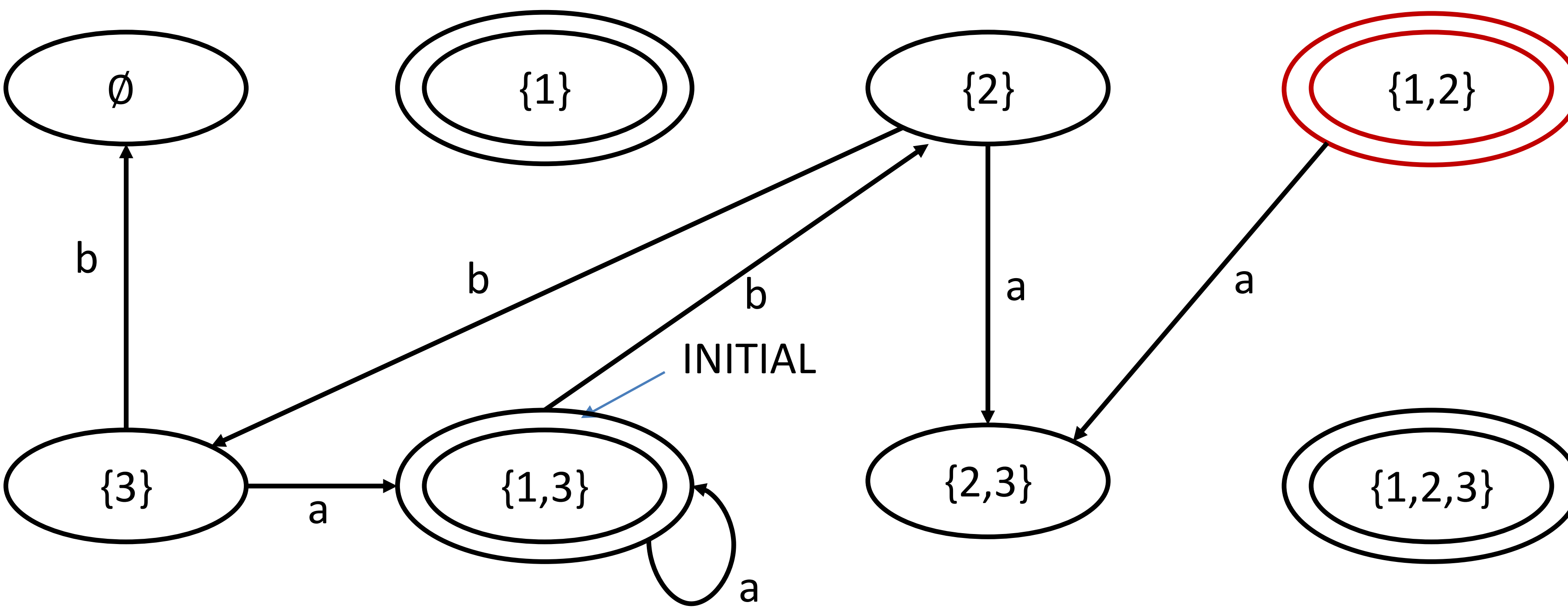


NFA->DFA Example

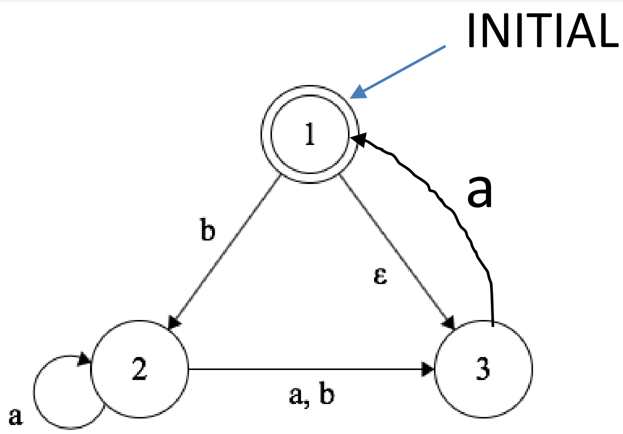


Transition table for NFA to the left:

State\Char	a	b
1	{}	{2}
2	{2,3}	{3}
3	{1,3}	{}

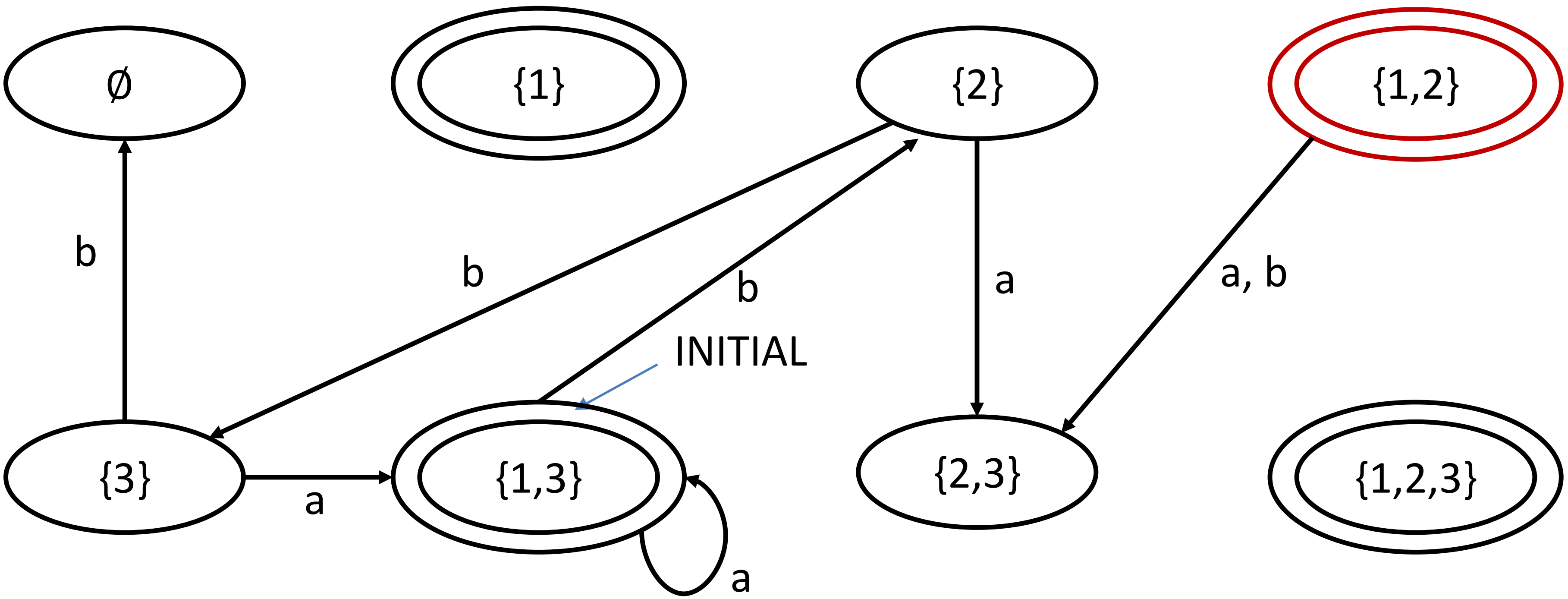


NFA->DFA Example

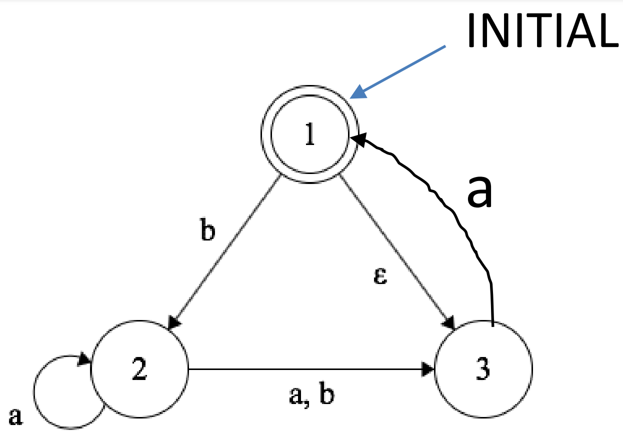


Transition table for NFA to the left:

State\Char	a	b
1	{}	{2}
2	{2,3}	{3}
3	{1,3}	{}

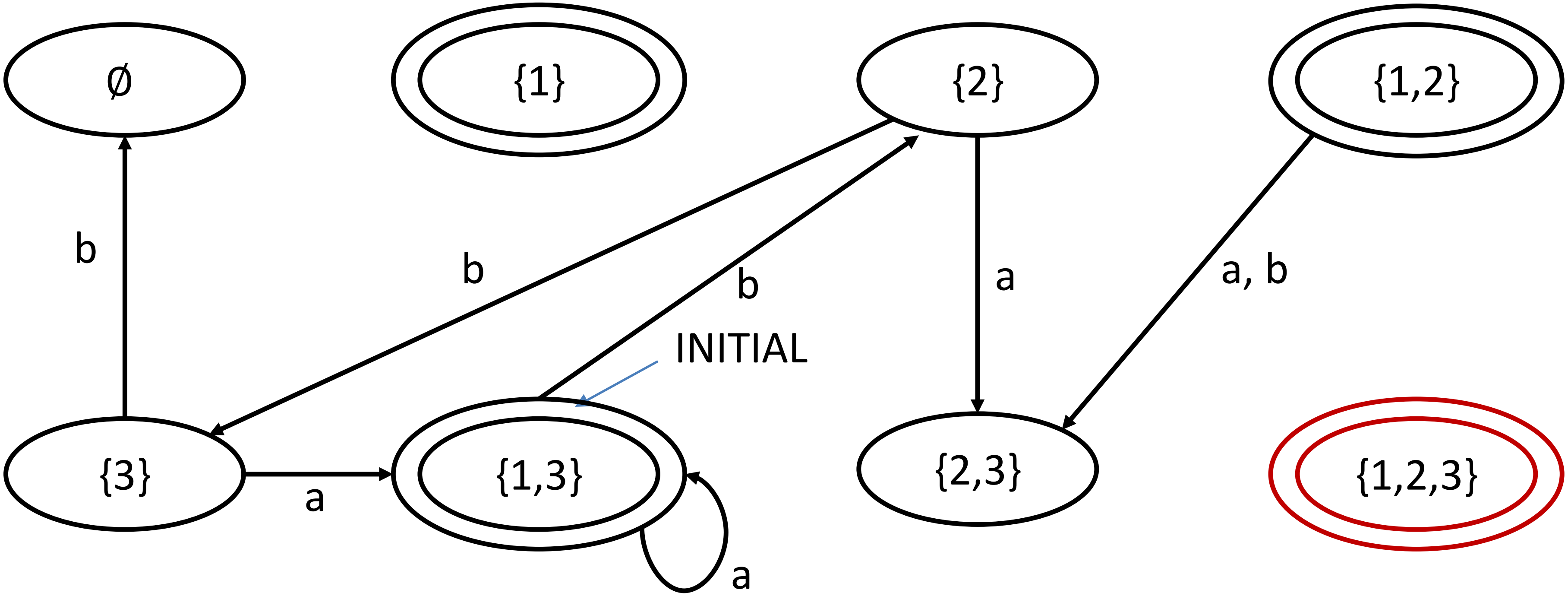


NFA->DFA Example

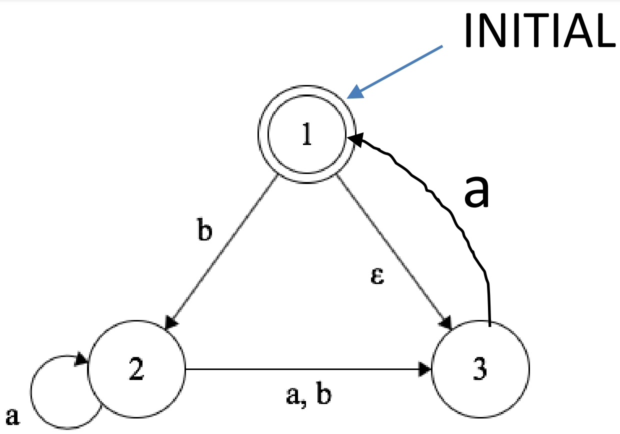


Transition table for NFA to the left:

State\Char	a	b
1	{}	{2}
2	{2,3}	{3}
3	{1,3}	{}

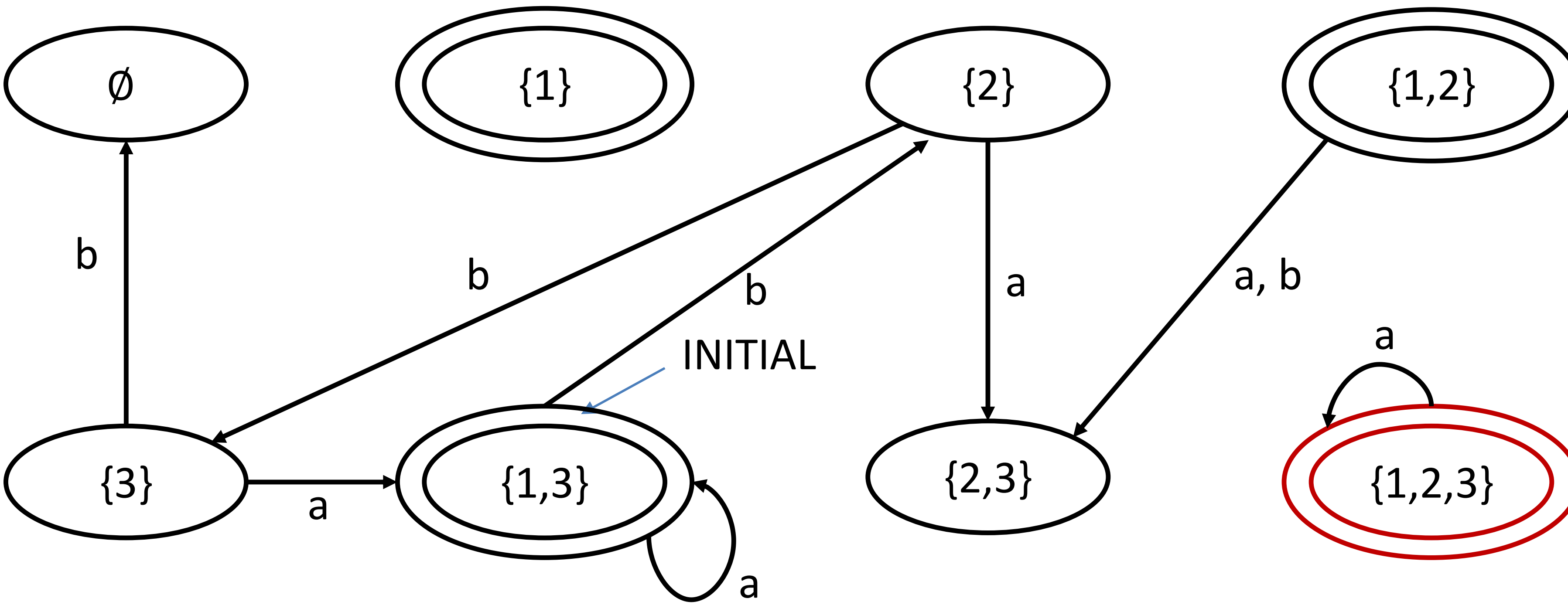


NFA->DFA Example

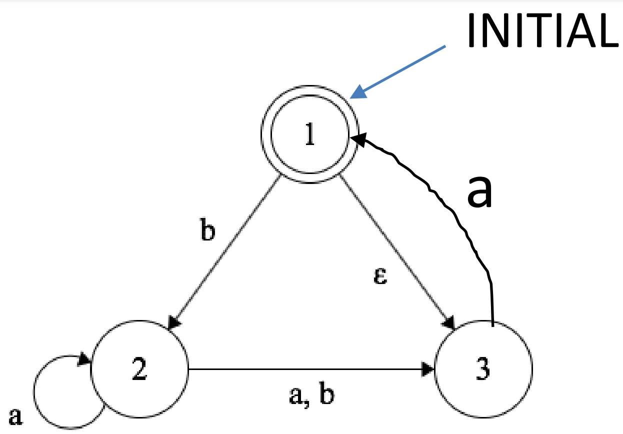


Transition table for NFA to the left:

State\Char	a	b
1	{}	{2}
2	{2,3}	{3}
3	{1,3}	{}

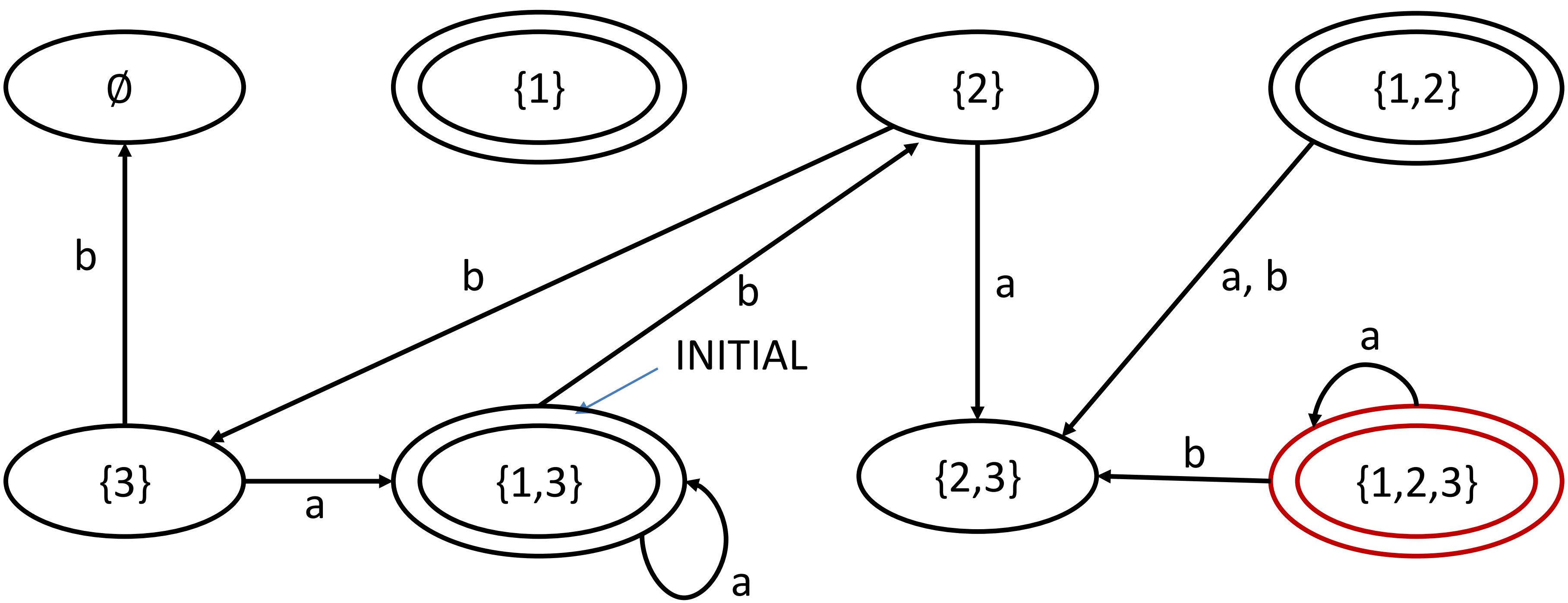


NFA->DFA Example

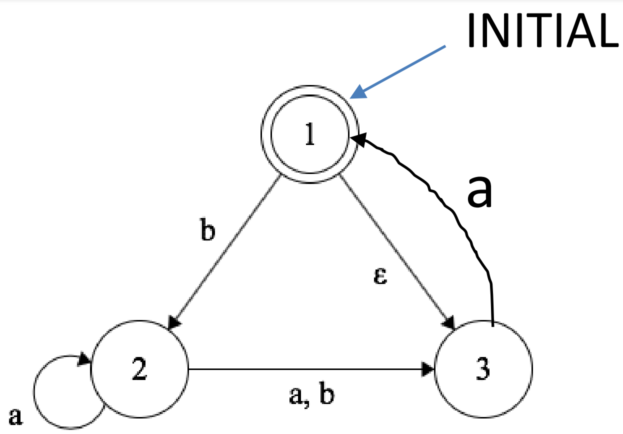


Transition table for NFA to the left:

State\Char	a	b
1	{}	{2}
2	{2,3}	{3}
3	{1,3}	{}

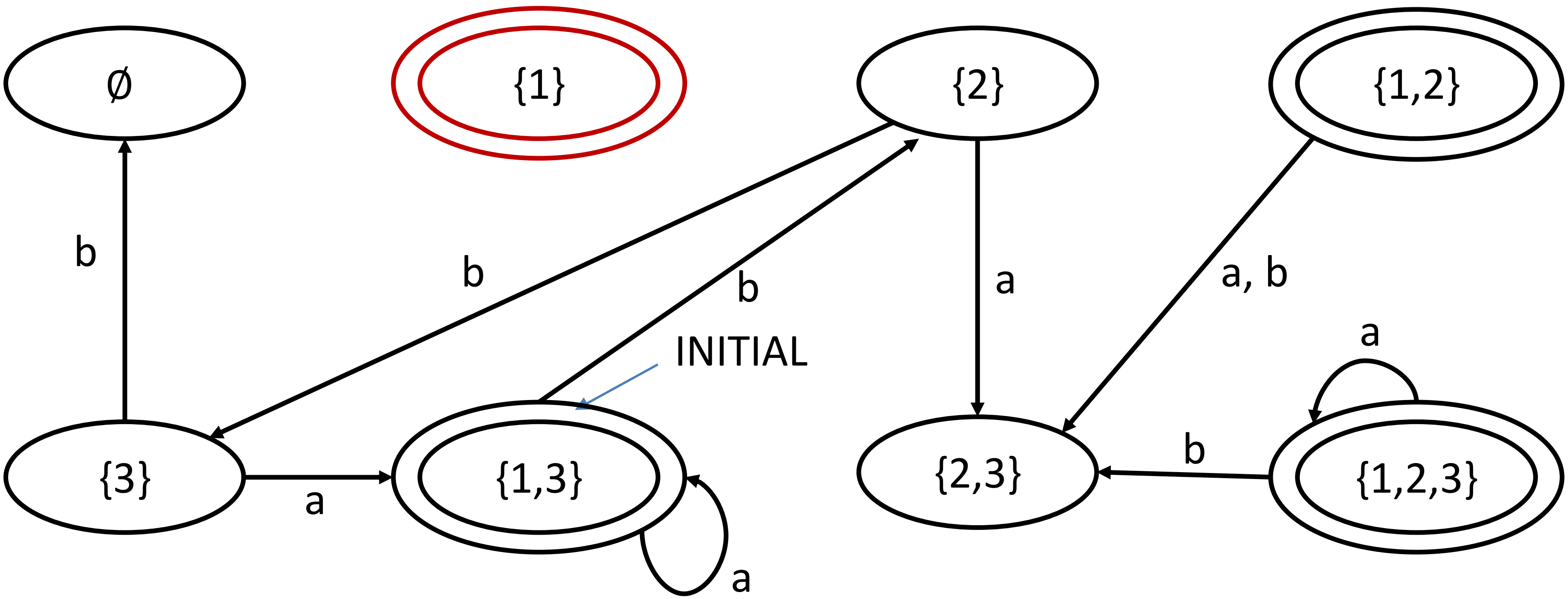


NFA->DFA Example

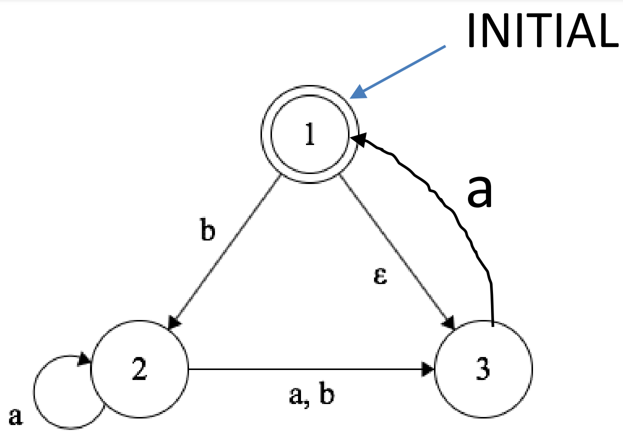


Transition table for NFA to the left:

State\Char	a	b
1	{}	{2}
2	{2,3}	{3}
3	{1,3}	{}

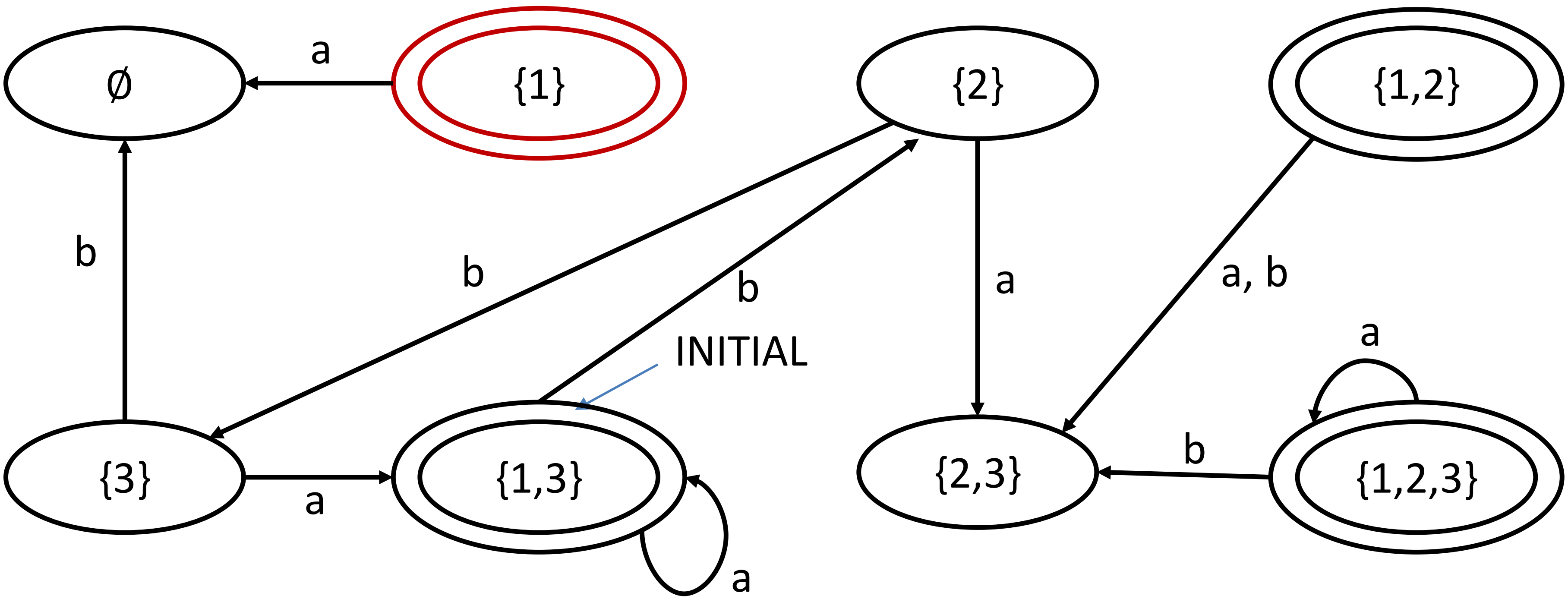


NFA->DFA Example

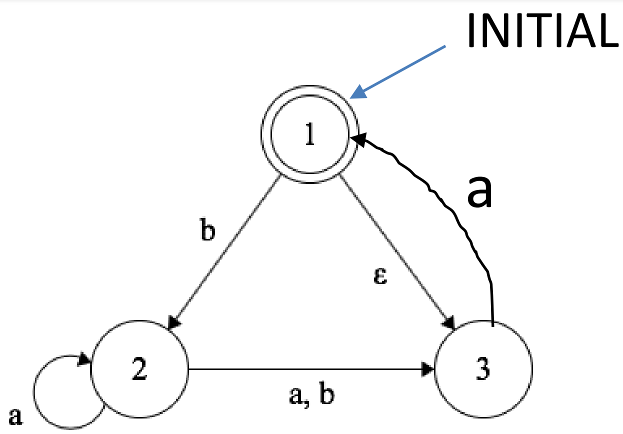


Transition table for NFA to the left:

State\Char	a	b
1	{}	{2}
2	{2,3}	{3}
3	{1,3}	{}

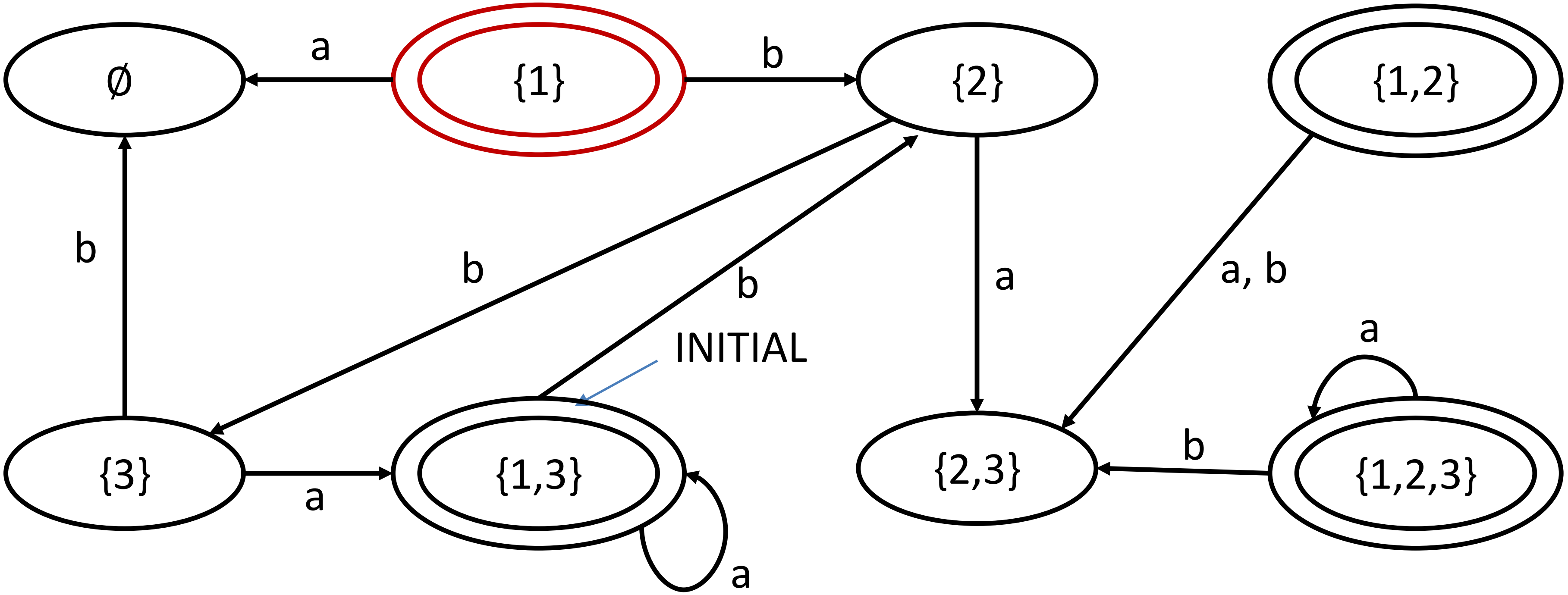


NFA->DFA Example

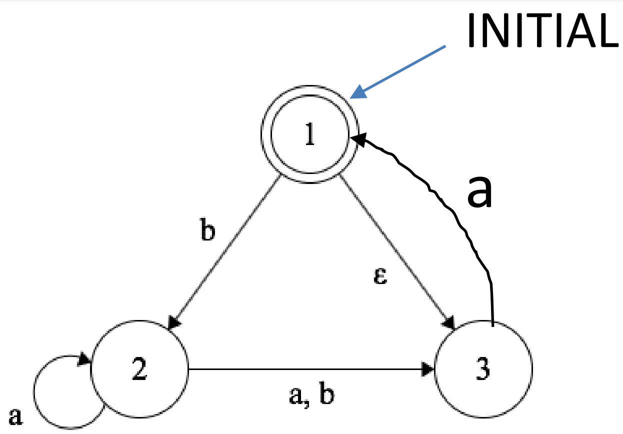


Transition table for NFA to the left:

State\Char	a	b
1	{}	{2}
2	{2,3}	{3}
3	{1,3}	{}

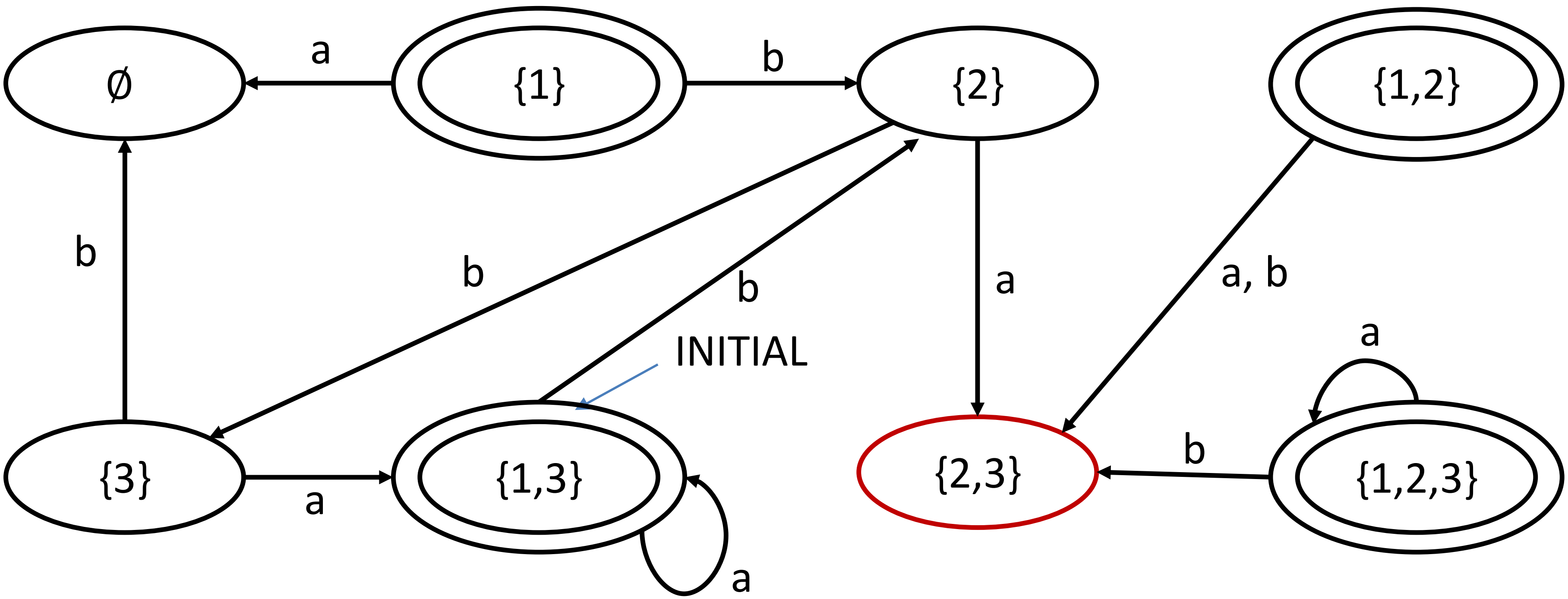


NFA->DFA Example

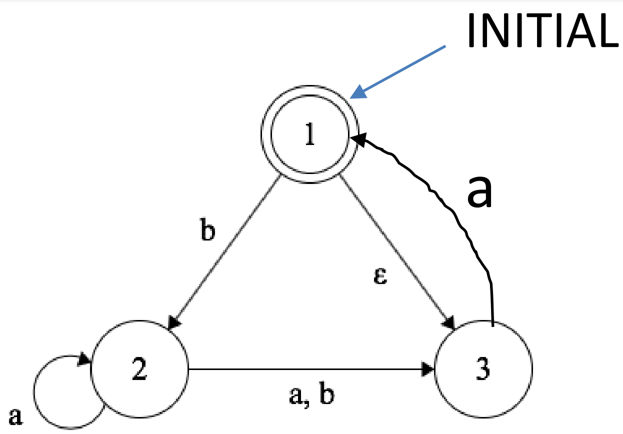


Transition table for NFA to the left:

State\Char	a	b
1	{}	{2}
2	{2,3}	{3}
3	{1,3}	{}

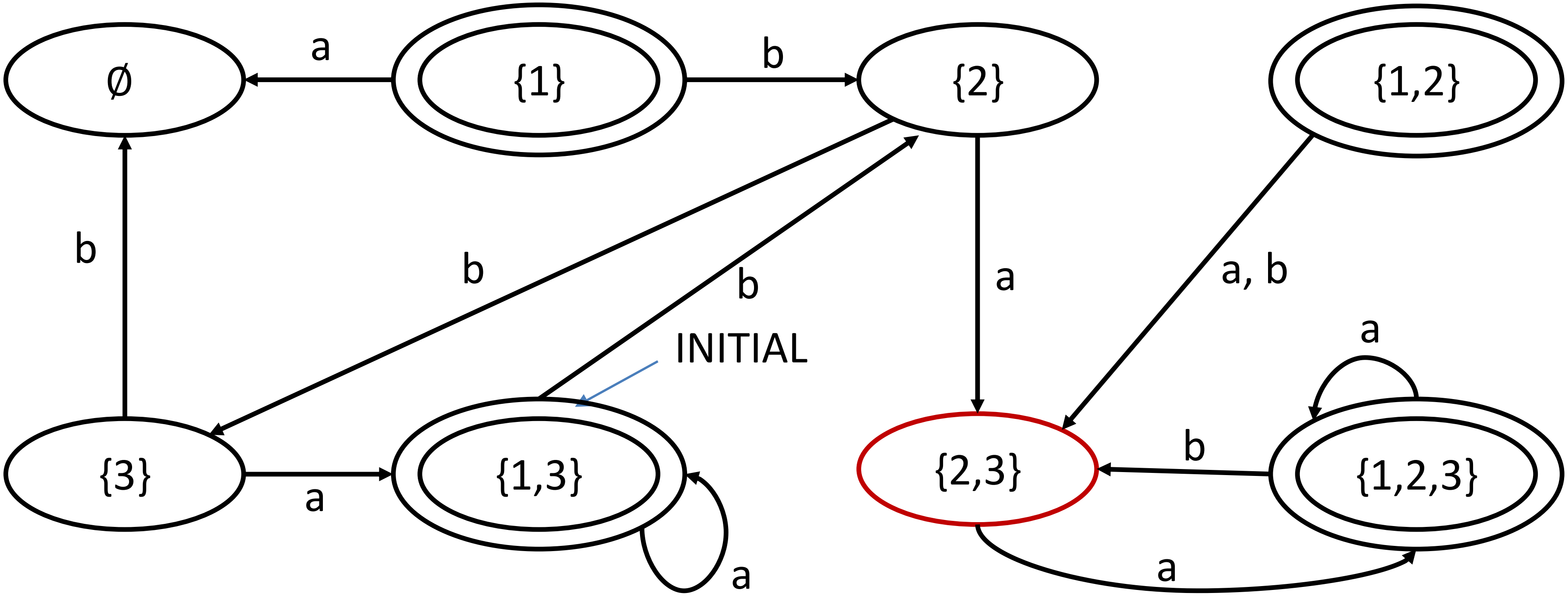


NFA->DFA Example

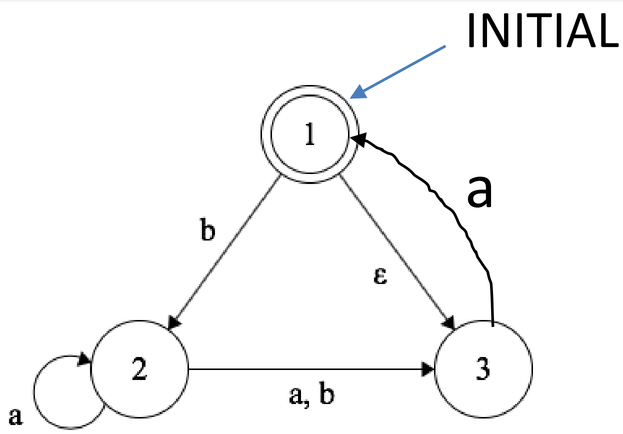


Transition table for NFA to the left:

State\Char	a	b
1	{}	{2}
2	{2,3}	{3}
3	{1,3}	{}

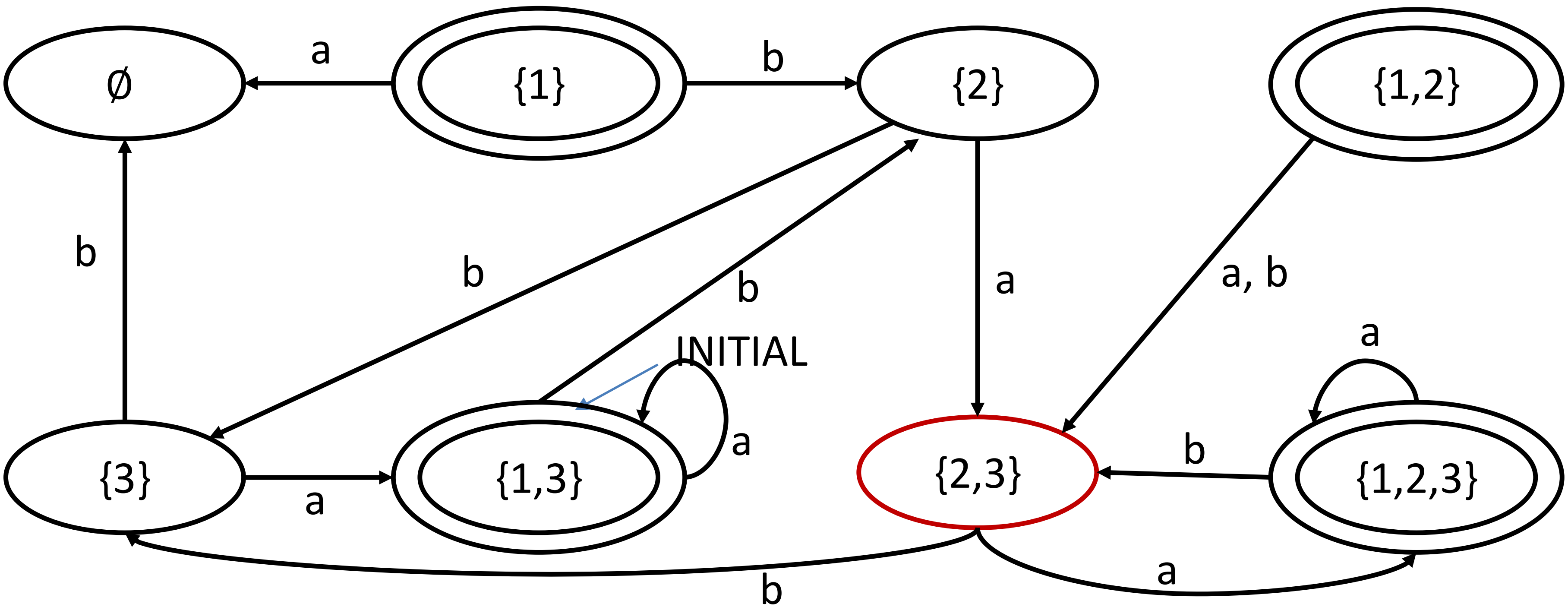


NFA->DFA Example

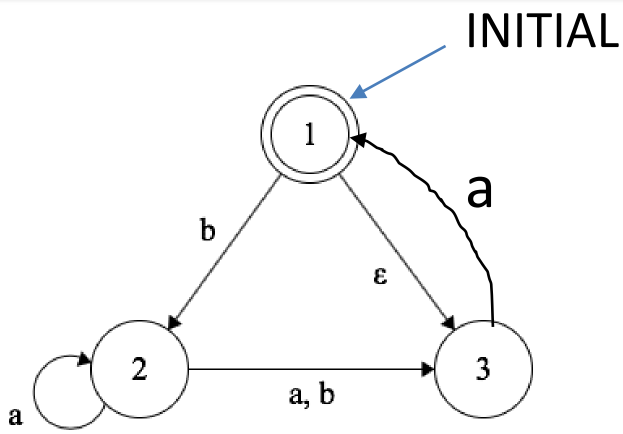


Transition table for NFA to the left:

State\Char	a	b
1	{}	{2}
2	{2,3}	{3}
3	{1,3}	{}



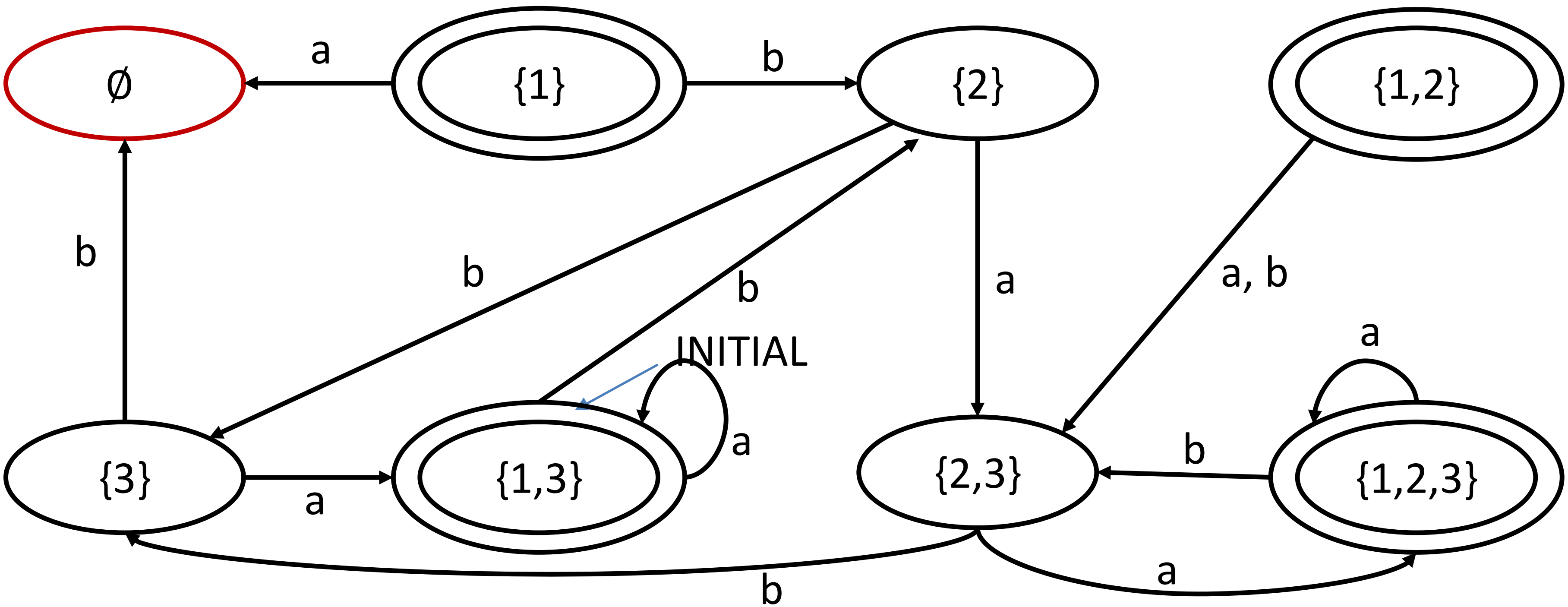
NFA->DFA Example



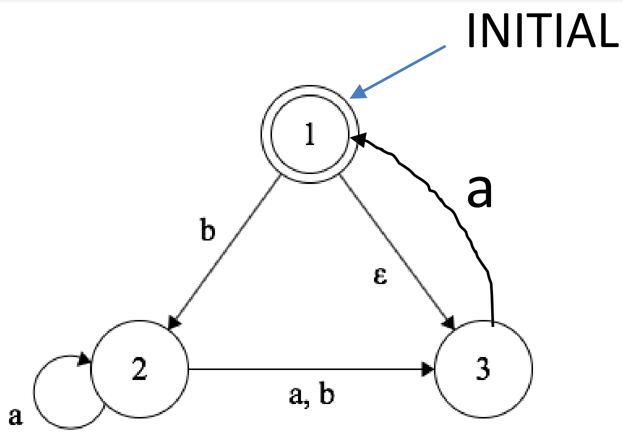
Transition table for NFA to the left:

State\Char	a	b
1	{}	{2}
2	{2,3}	{3}
3	{1,3}	{}

What about transitions from Ø?

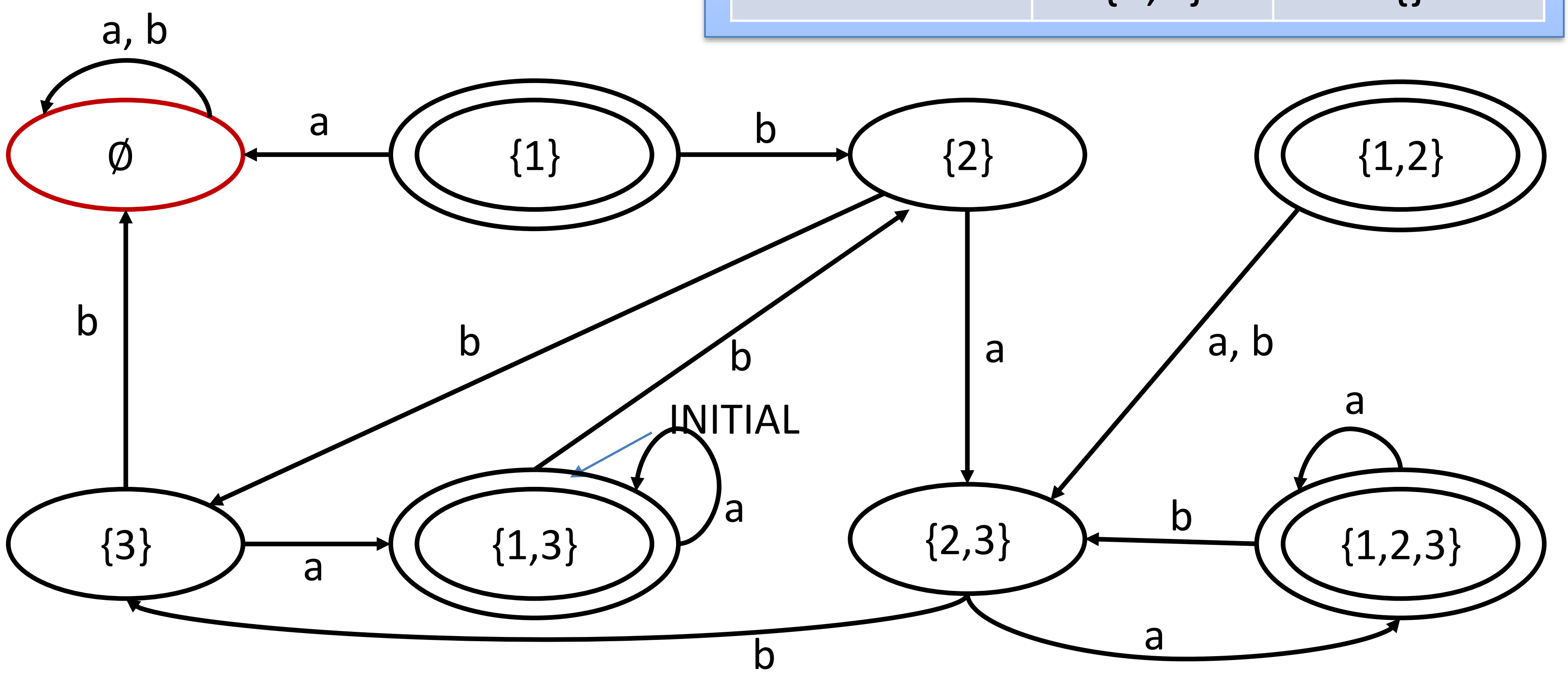


NFA->DFA Example

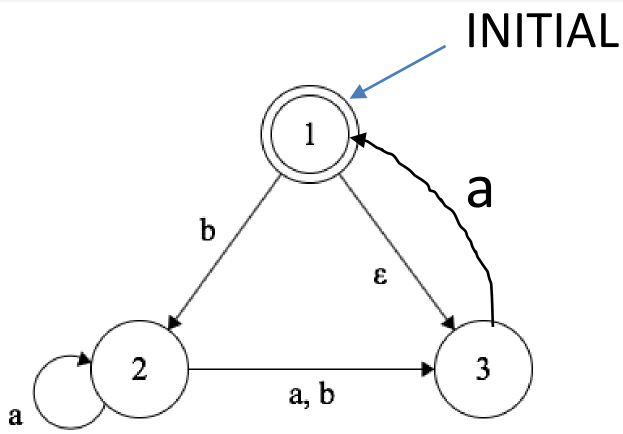


Transition table for NFA to the left:

State\Char	a	b
1	{}	{2}
2	{2,3}	{3}
3	{1,3}	{}



NFA->DFA Example



Does this **DFA** accept:

"a"? ***YES***

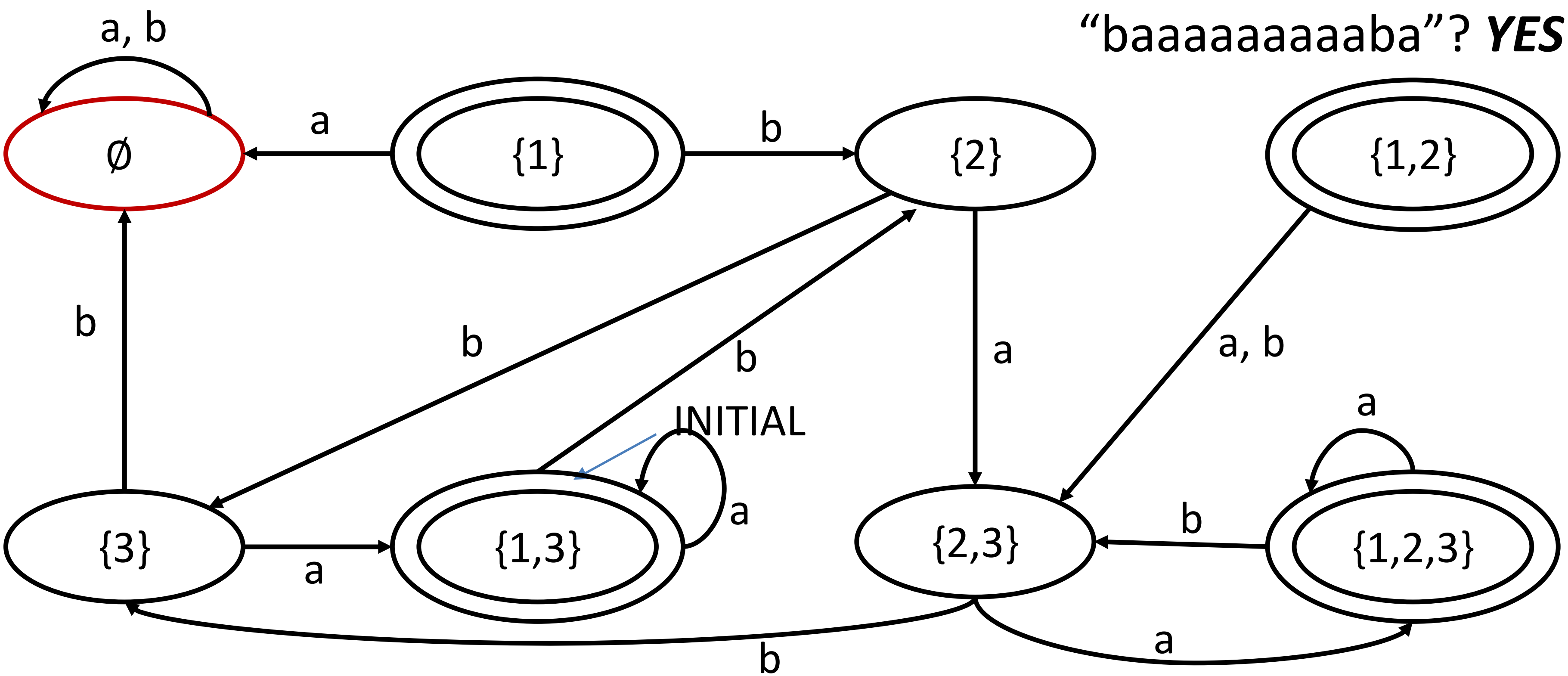
"bb"? ***NO***

"baa"? ***YES***

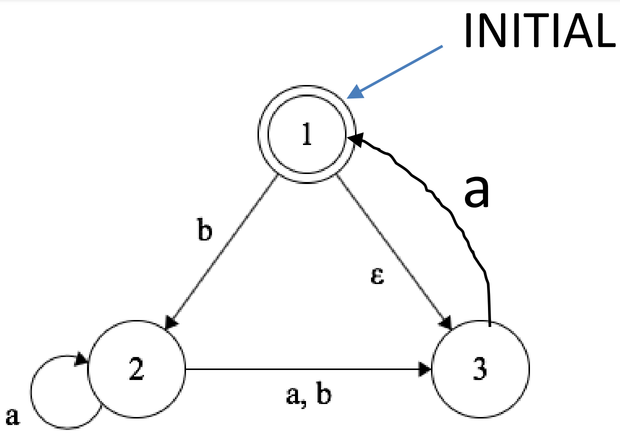
"bba"? ***YES***

"baaaaaaaaab"? ***NO***

"baaaaaaaaaba"? ***YES***



Removing Unreachable States



Does this **DFA** accept:

"a"? ***YES***

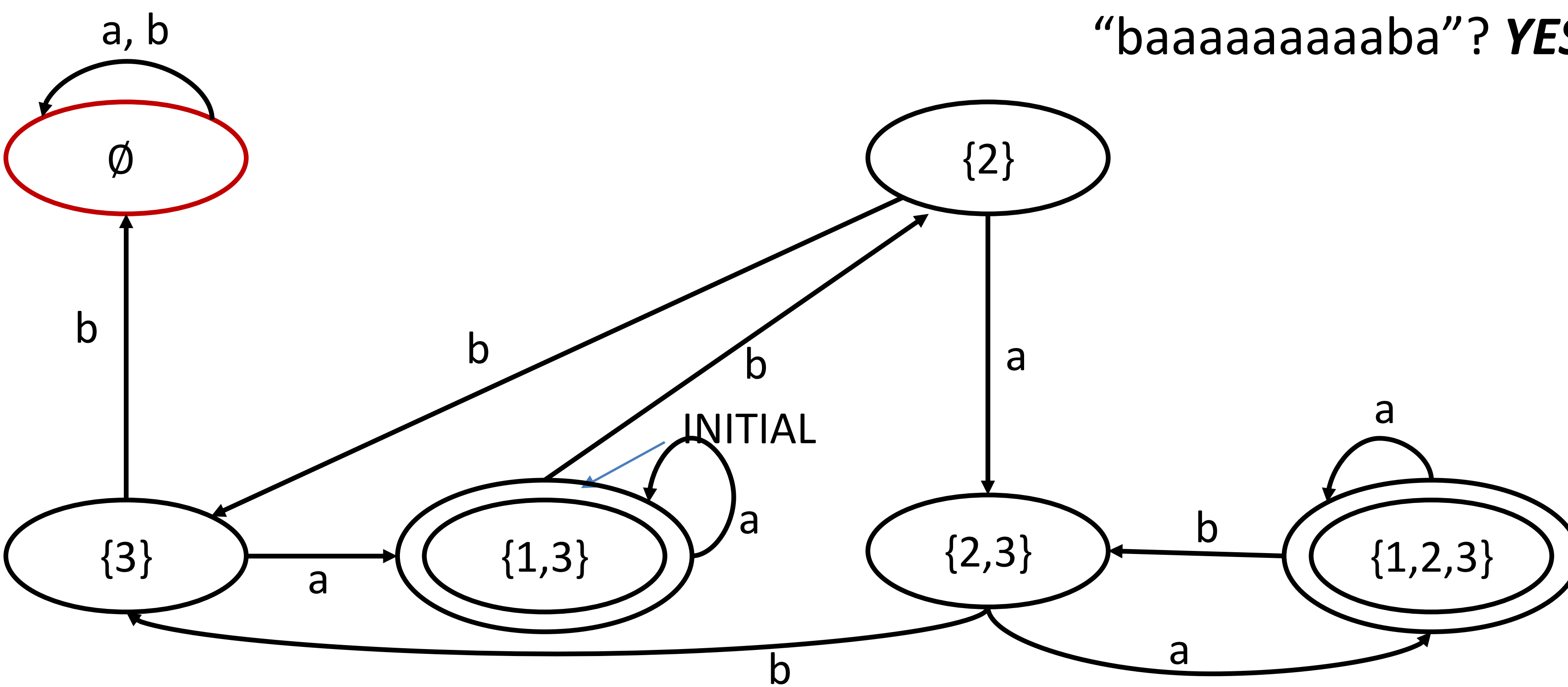
"bb"? ***NO***

"baa"? ***YES***

"bba"? ***YES***

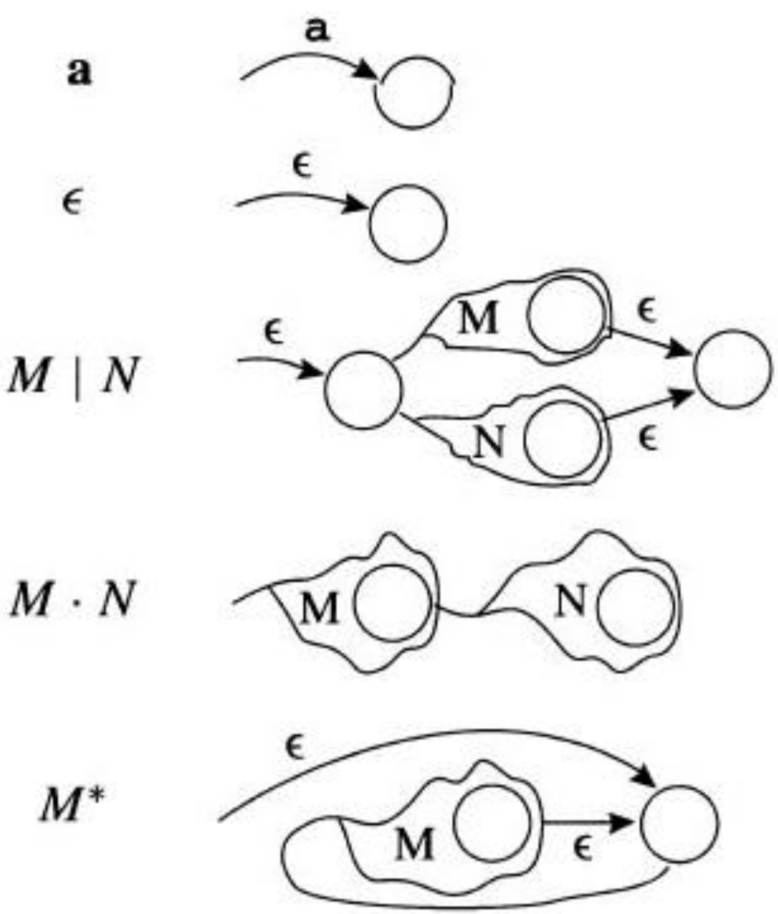
"baaaaaaaaab"? ***NO***

"baaaaaaaaaba"? ***YES***

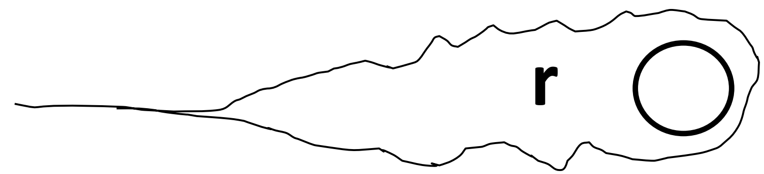


RE->NFA

Converting REs to NFAs



[Appel 2.4]

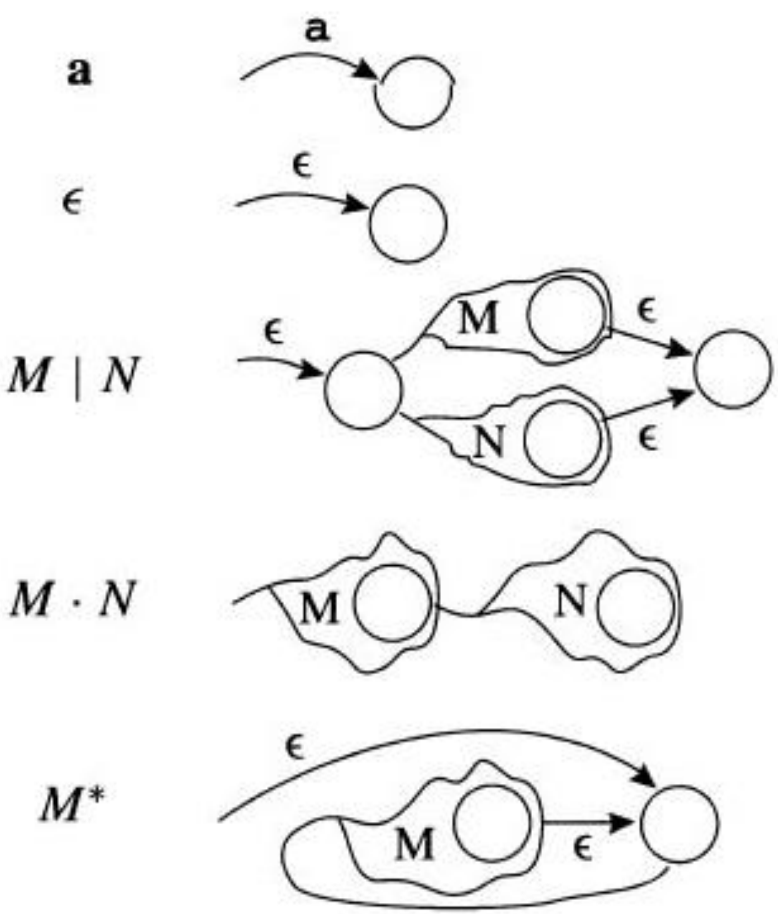


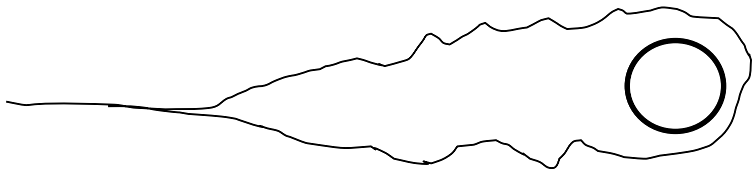
= Recursively apply rules to the left to regular expression **r**, connecting heads and tails as you go

Every RE is encodable as an equivalent NFA (and thus also an equivalent DFA)

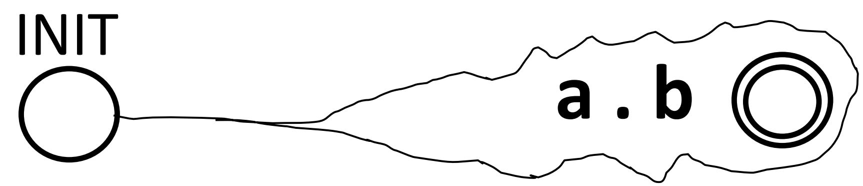
Reminder: ***equivalent =*** accepts exactly the same set of strings

Example 1



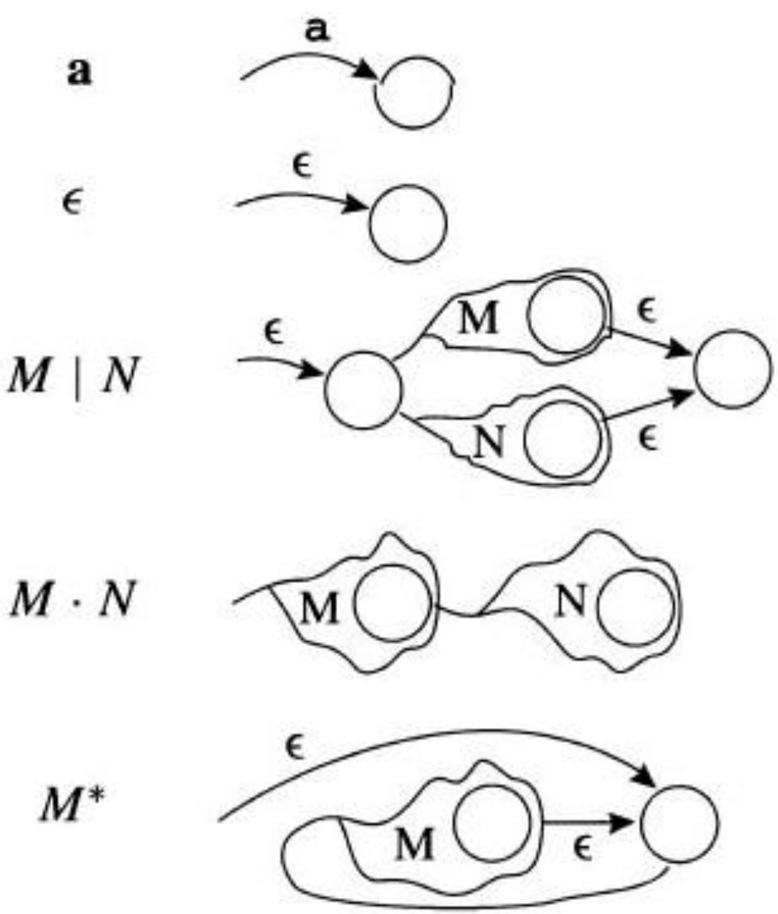


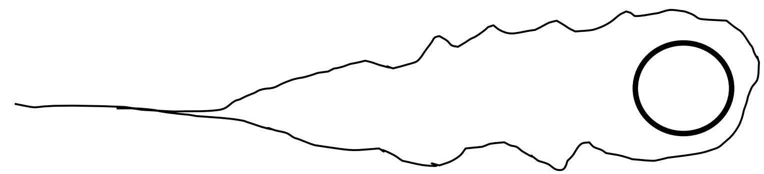
= Recursively apply rules to the left to regular expression **r**, connecting heads and tails as you go



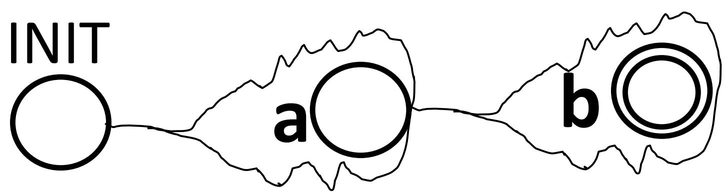
[Appel 2.4]

Example 1



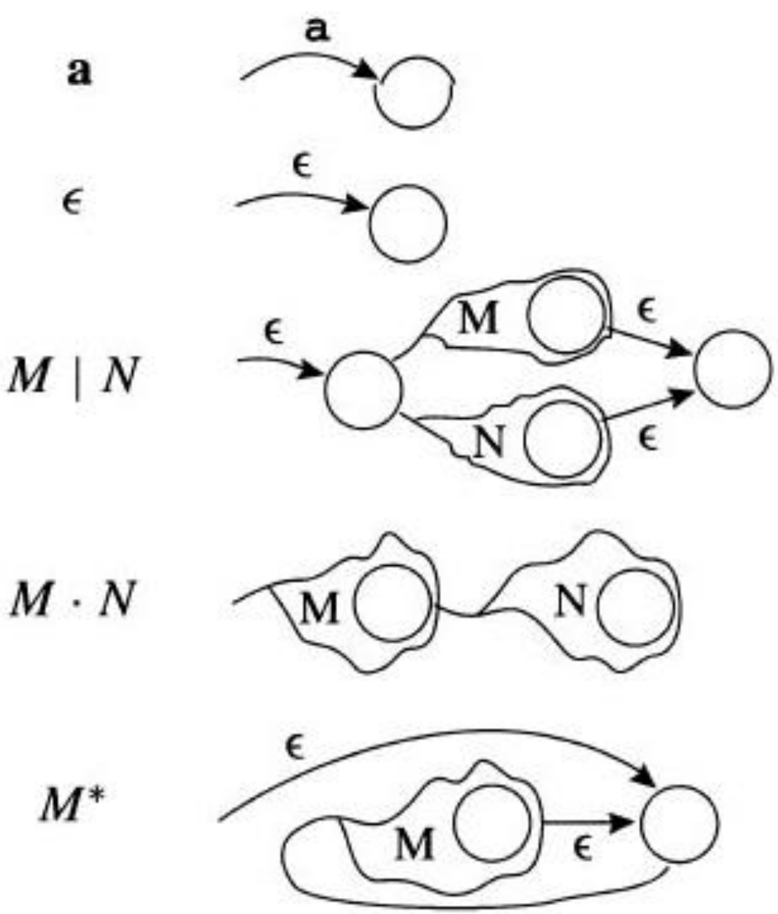


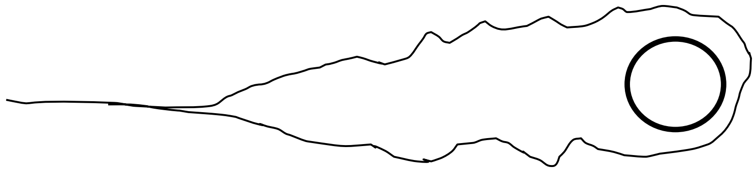
= Recursively apply rules to the left to regular expression **r**, connecting heads and tails as you go



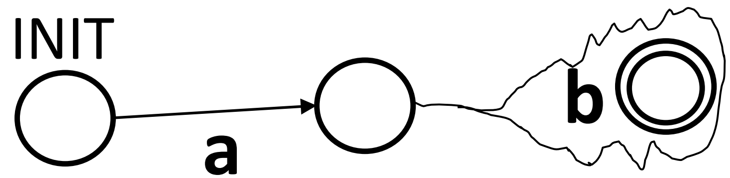
[Appel 2.4]

Example 1



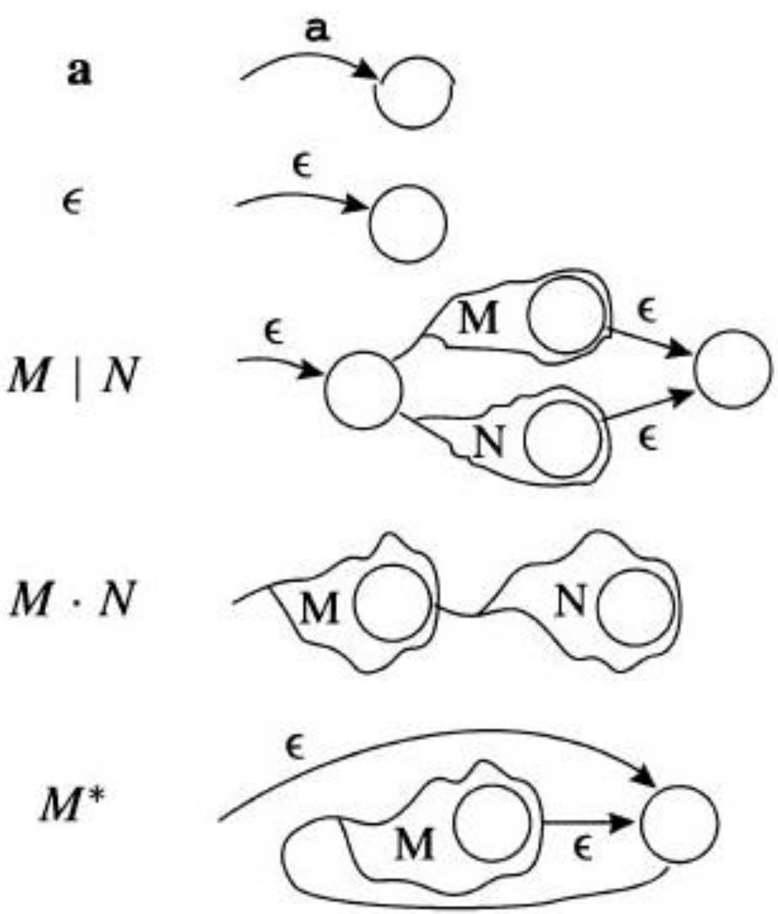


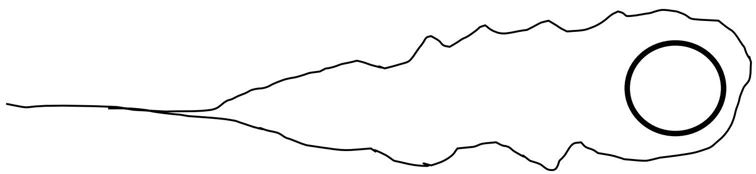
= Recursively apply rules to the left to regular expression **r**, connecting heads and tails as you go



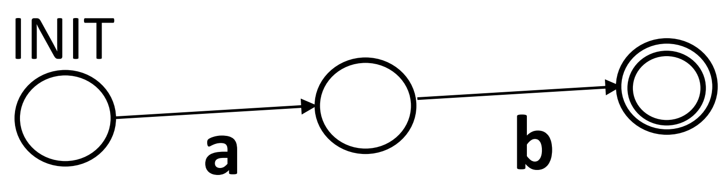
[Appel 2.4]

Example 1





= Recursively apply rules to the left to regular expression **r**, connecting heads and tails as you go

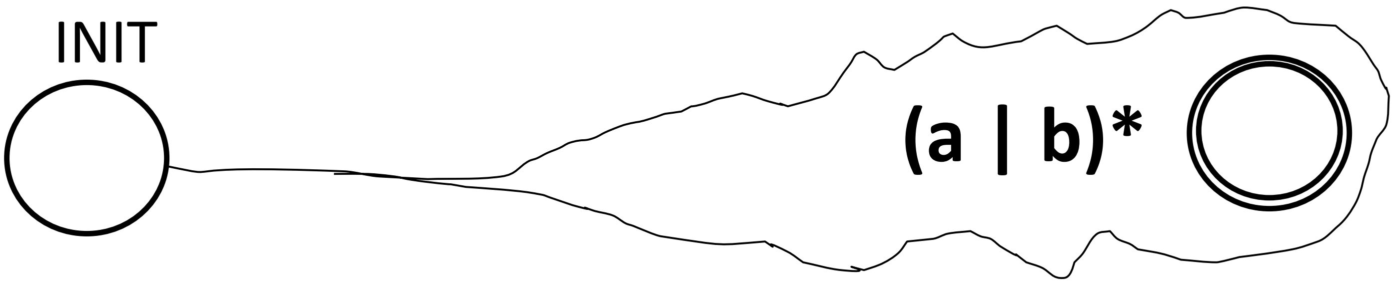


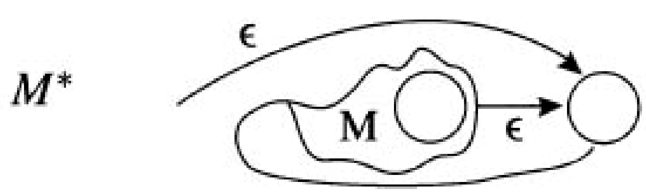
[Appel 2.4]

Example 2

Convert the following RE to an equivalent NFA:

(a | b)*

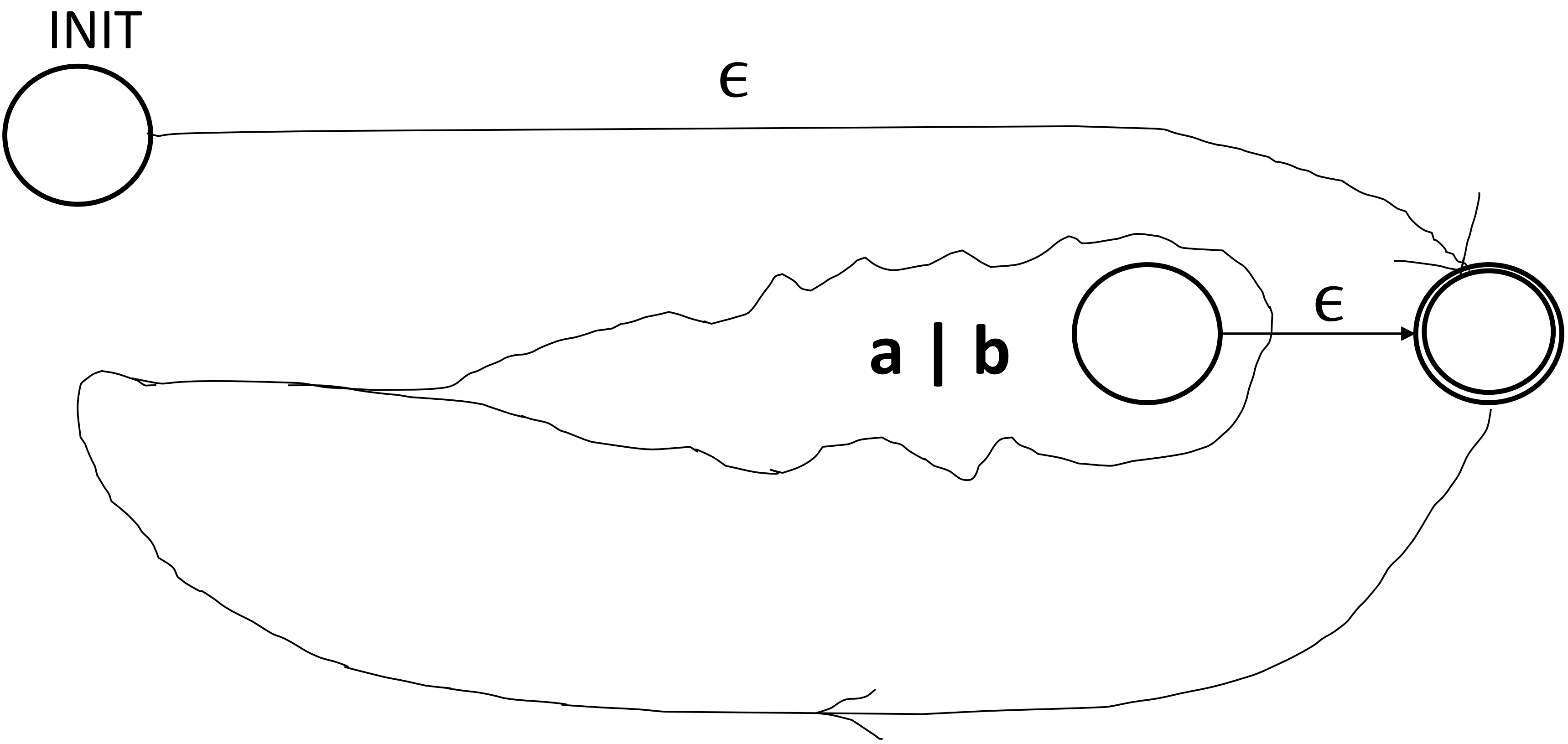




Example 2

Convert the following RE to an equivalent NFA:

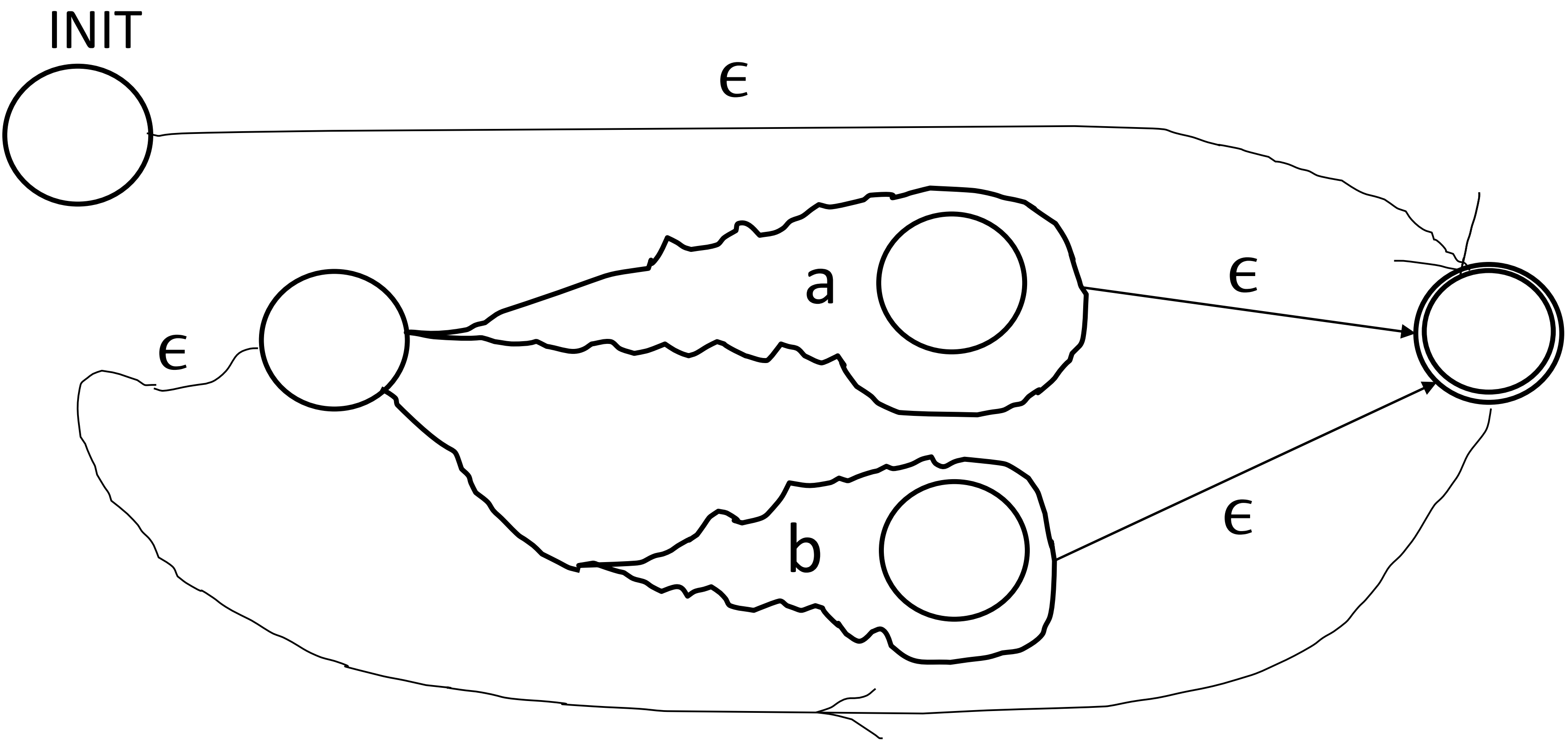
(a | b)*



Example 2

Convert the following RE to an equivalent NFA:

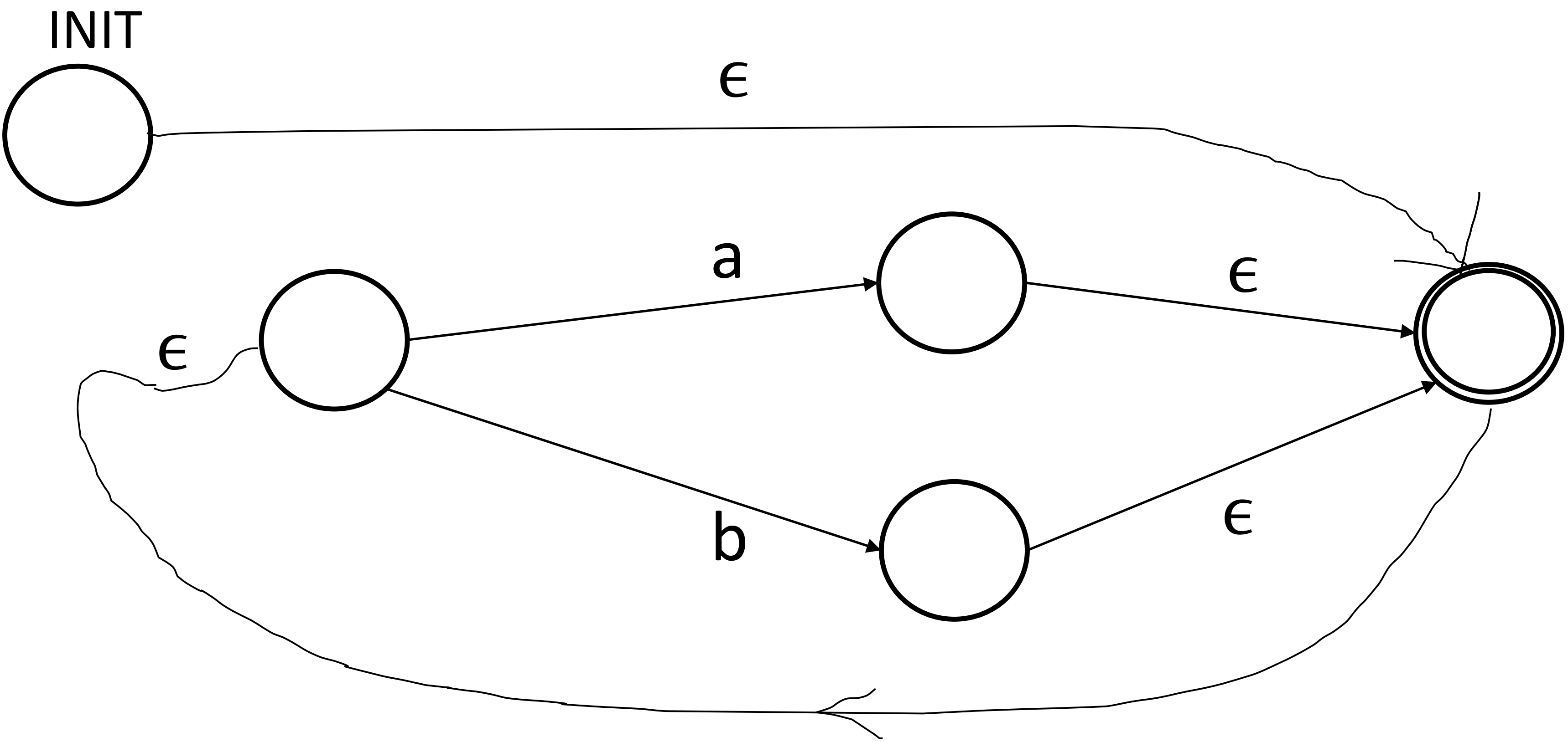
(a | b)*



Example 2

Convert the following RE to an equivalent NFA:

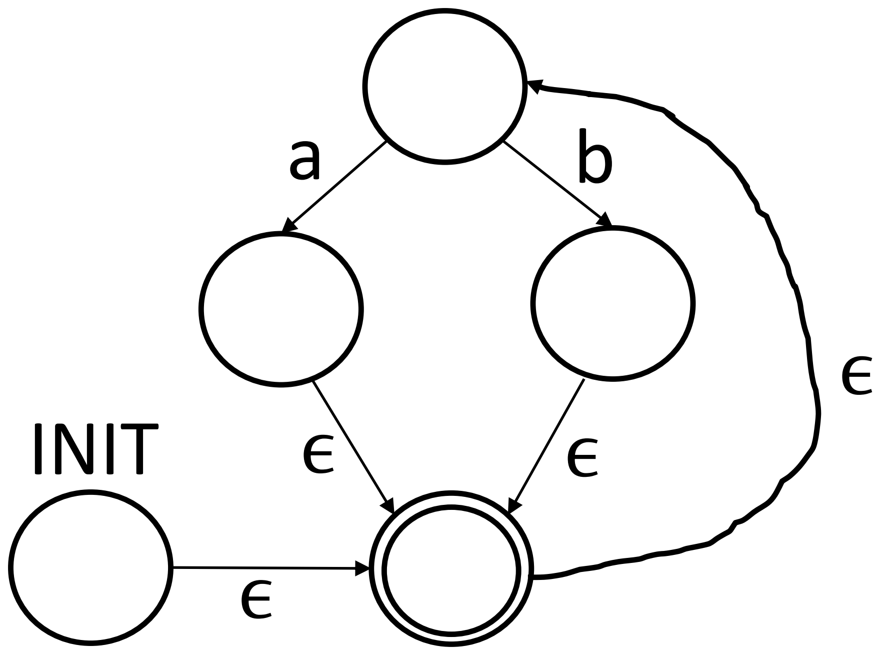
(a | b)*



A More Conventional Drawing of the Same NFA

Convert the following RE to an equivalent NFA:

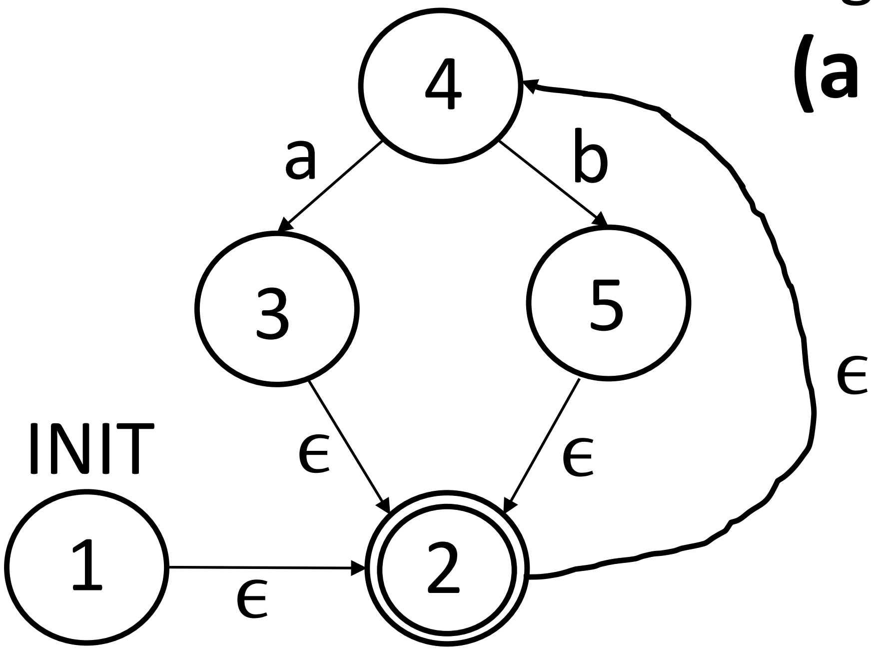
(a | b)*



RE->NFA->DFA

Convert the following RE to an equivalent NFA:

(a | b)*

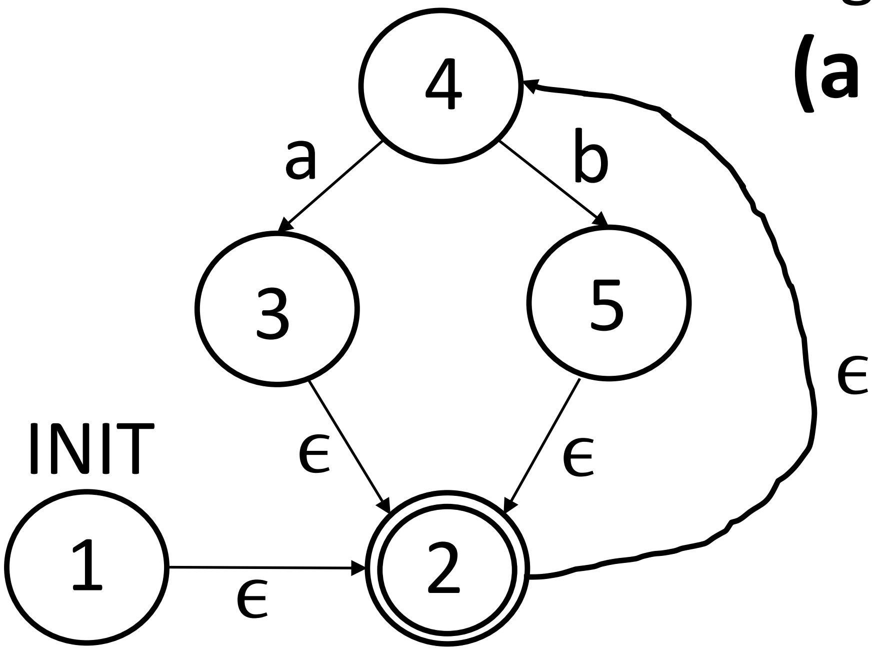


1. Label states

RE->NFA->DFA

Convert the following RE to an equivalent NFA:

(a | b)*



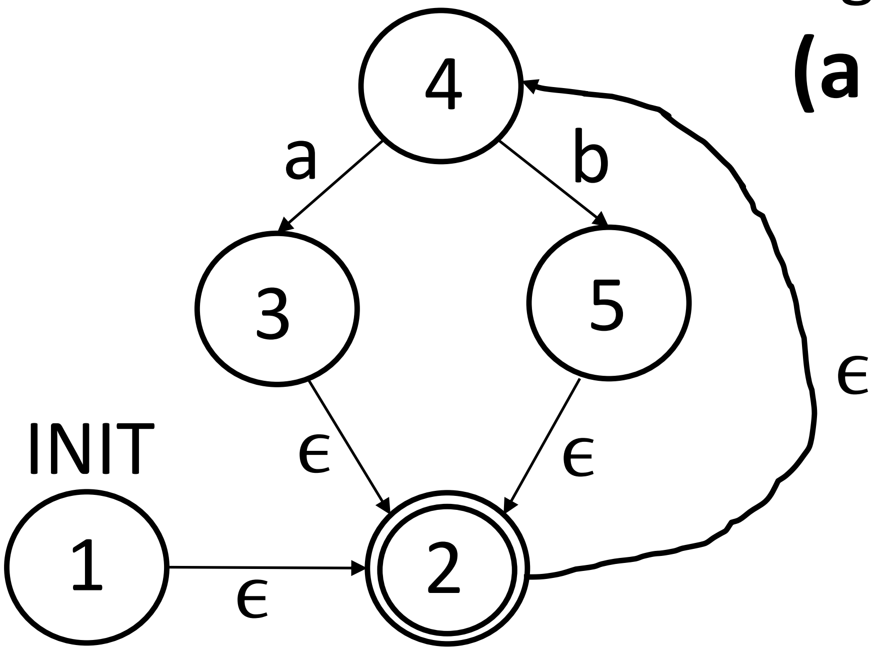
1. Label states
2. Construct transition table

State/Char	a	b
1	{}	{}
2	{}	{}
3	{}	{}
4	{2,3,4}	{2,4,5}
5	{}	{}

RE->NFA->DFA

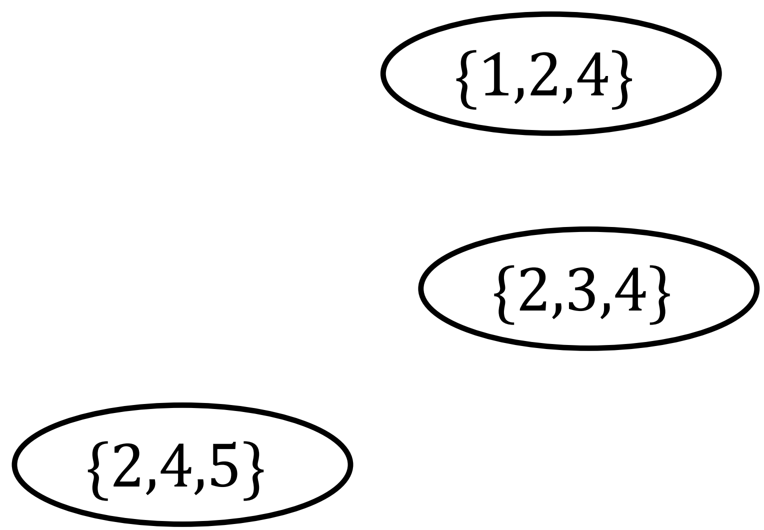
Convert the following RE to an equivalent NFA:

(a | b)*



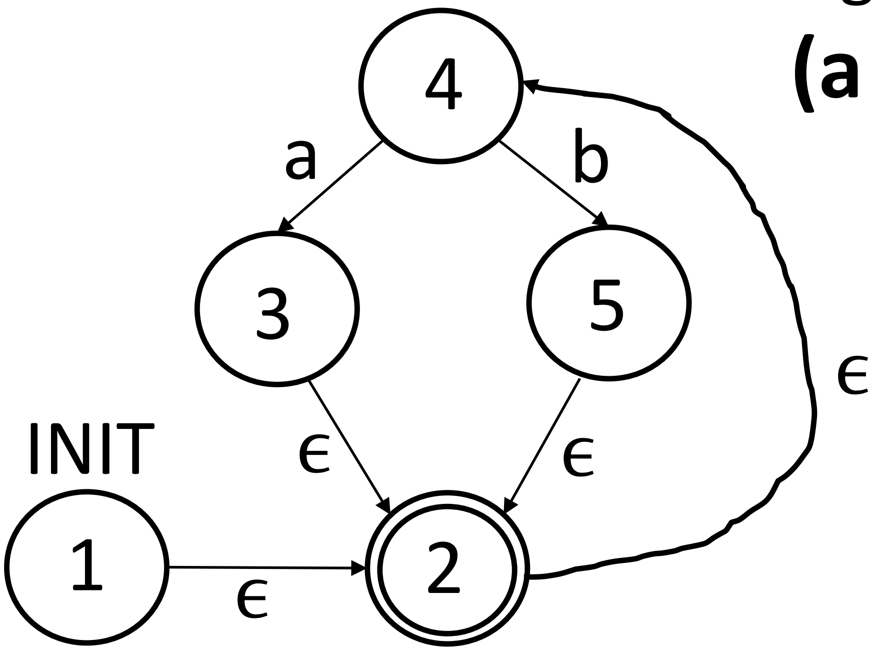
1. Label states
2. Construct transition table
3. DFA states

State/Char	a	b
1	{}	{}
2	{}	{}
3	{}	{}
4	{2,3,4}	{2,4,5}
5	{}	{}



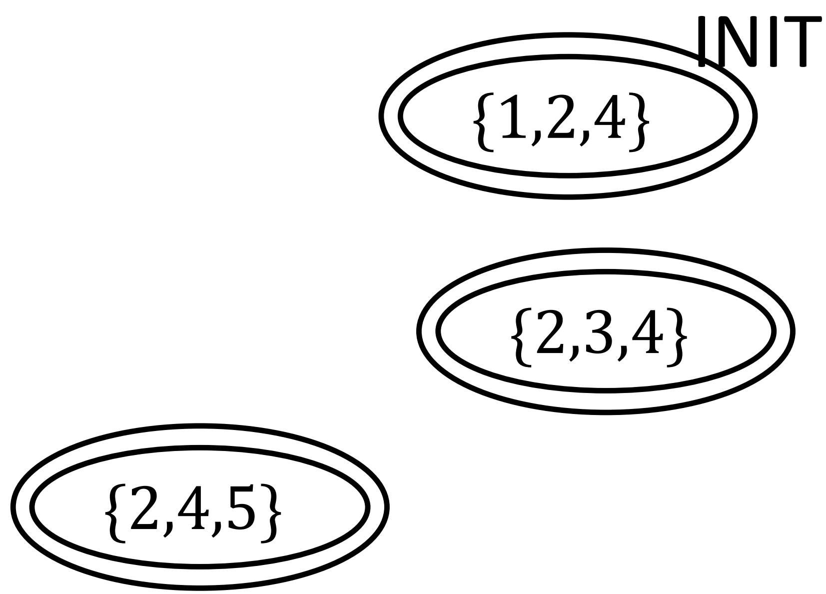
Convert the following RE to an equivalent NFA:

(a | b)*



1. Label states
2. Construct transition table
3. DFA states
4. Init and final

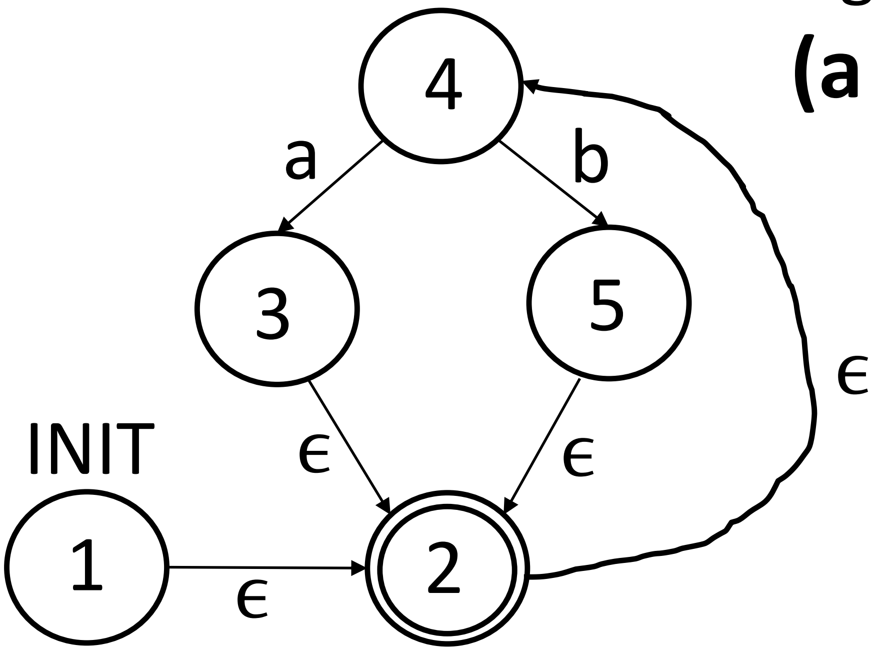
State/Char	a	b
1	{}	{}
2	{}	{}
3	{}	{}
4	{2,3,4}	{2,4,5}
5	{}	{}



RE->NFA->DFA

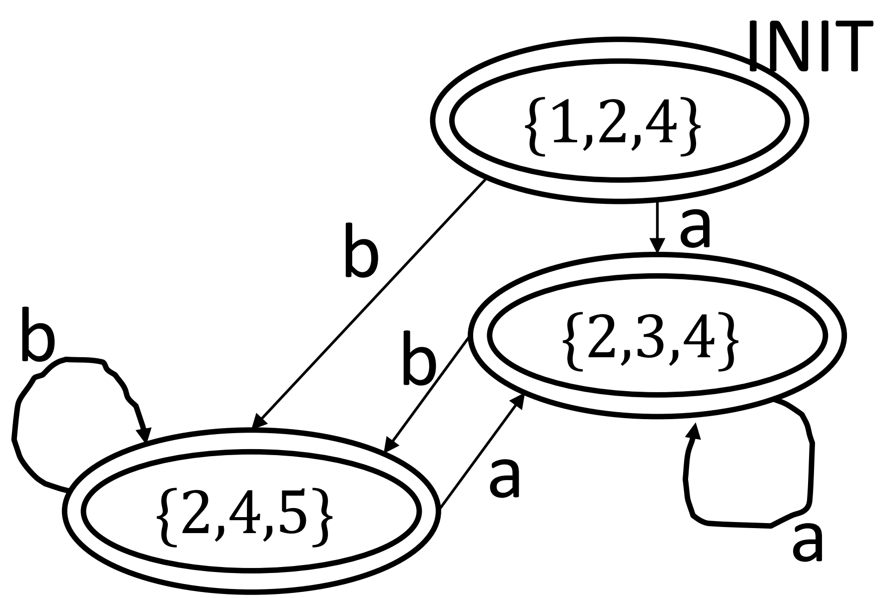
Convert the following RE to an equivalent NFA:

(a | b)*



1. Label states
2. Construct transition table
3. DFA states
4. Init and final
5. Mark transitions

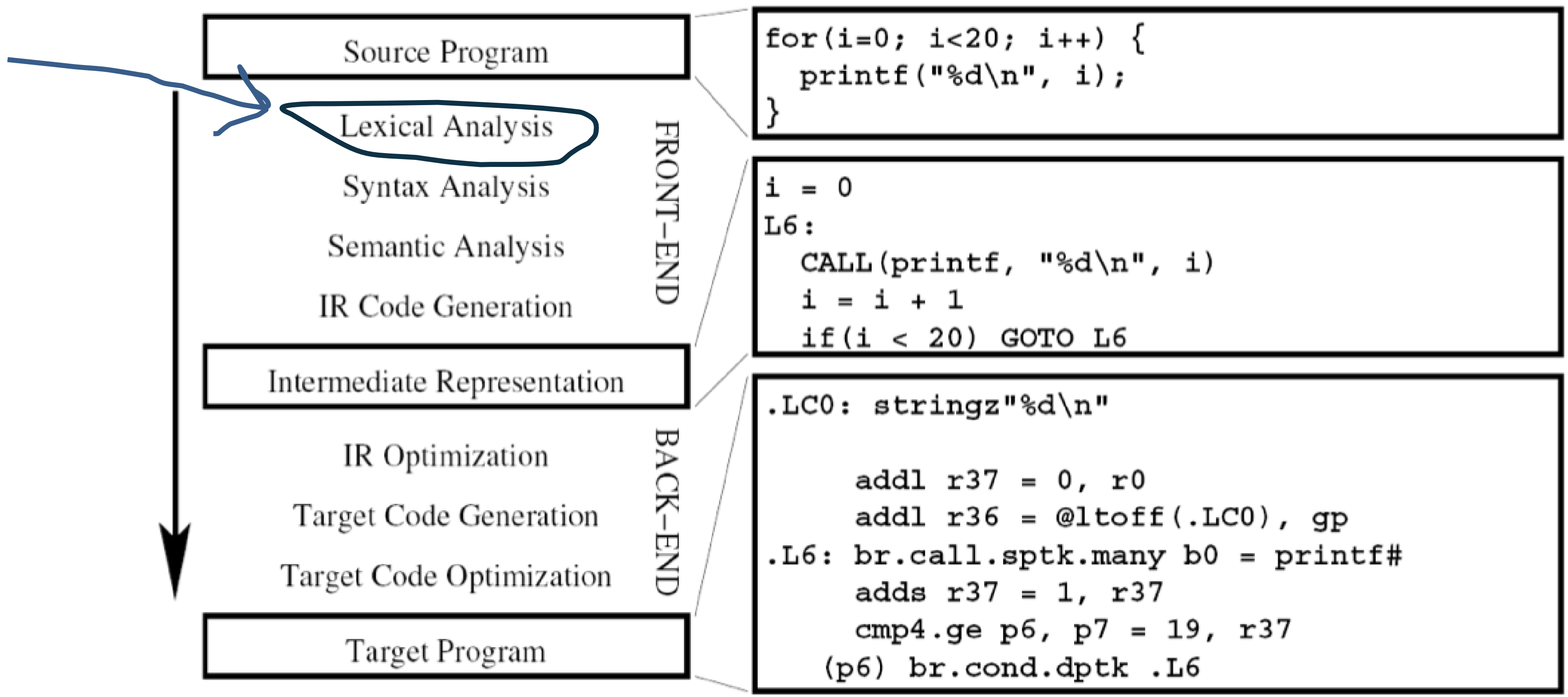
State/Char	a	b
1	{}	{}
2	{}	{}
3	{}	{}
4	{2,3,4}	{2,4,5}
5	{}	{}



OCAMLLEX

Demo: ocamllex

- The RE->NFA->DFA reduction is mechanical and tedious
- Fortunately, people have written computer programs (e.g., ocamllex) to automate this process for us!
- Following a long line of similar tools for other languages (e.g., lex for C)



The Grumpy Online Visual Debugger

- Lexing Grumpy (lexer.mll)
- https://bagnalla.github.io/webgrumpy/

EXTRAS: DFA MINIMIZATION

Minimal DFAs

- A DFA ***D*** is ***minimal*** if there is no equivalent DFA ***D'*** with strictly fewer states
- Equivalent? Accepts the same language, or set of strings
- The minimal DFA for a given language is ***unique*** (up to renamings of states)

- How to minimize?
  - [Moore '56]
  - [Brzozowski '63]
  - …

The Brzozowski Minimization Procedure

INPUT DFA ***D***

OUTPUT DFA ***D'*** minimizing ***D***

Algorithm:

1. Convert ***D*** to NFA ***R(D)***, the "reversal" of ***D***

   Reversal ***R(D)*** = ***D*** with all arrows reversed, INIT=FINAL, FINAL=INIT

2. Convert ***R(D)*** to $\boldsymbol{D_R}$ using powerset algorithm of previous slides

3. Convert $\boldsymbol{D_R}$ to NFA $\boldsymbol{R(D_R)}$

4. Convert $\boldsymbol{R(D_R)}$ to $\boldsymbol{(D_R)_R = D'}$