CSCI 340: Computational Models

# Kleene's Theorem

Chapter 7

Department of Computer Science

# Unification

In 1954, Kleene presented (and proved) a theorem which (in our version) states that if a language can be defined by any one of the three ways, then it can be defined by the other two.

All three of these *methods* of defining languages are *equivalent*.

## Theorem

*Any language that can be defined by:*

- *regular expression, or*
- *finite automaton, or*
- *transition graph*

*can be defined by all three methods.*

# How do we prove this theorem?

- This theorem is the most important and fundamental result in the theory of finite automata
- We need to carefully prove that it is correct
- We will do so by introducing **four algorithms** that enable us to construct the corresponding machines and expressions
- The general logic of this proof is as follows:
  1. Show that the set of all FAs can be defined by a set of TGs
  2. Show that the set of all TGs can be defined by a set of REs
  3. Show that the set of all REs can be defined by a set of FAs

  Mathematically:

$$[FA \subset TG \subset RE \subset FA] \equiv [FA = TG = RE]$$

# Formal Proof of Kleene's Theorem

## Proof.

The three sections of our proof will be:

1. Every language that can be defined by a finite automaton can also be defined by a transition graph
2. Every language that can be defined by a transition graph can also be defined by a regular expression
3. Every language that can be defined by a regular expression can also be defined by a finite automaton

When we have proven these three parts, we have finished our theorem □

## Proof of Part 1: $FA \subset TG$.

Every finite automaton *is* itself already a transition graph. Therefore, any language that has been defined by a finite automaton has already been defined by a transition graph. □

# Proof of Part 2: $TG \subset RE$

- We will prove part 2 by *constructive algorithm*
- Present a procedure that takes a TG and yields an RE which defines the same language
  - It must work for every conceivable TG
  - It must guarantee to finish its job in a finite time
  - It does not have to be a "good" algorithm – it just has to work

## How We Wish It Could Work

Look at the machine, figure out its language, and write down an equivalent regular expression

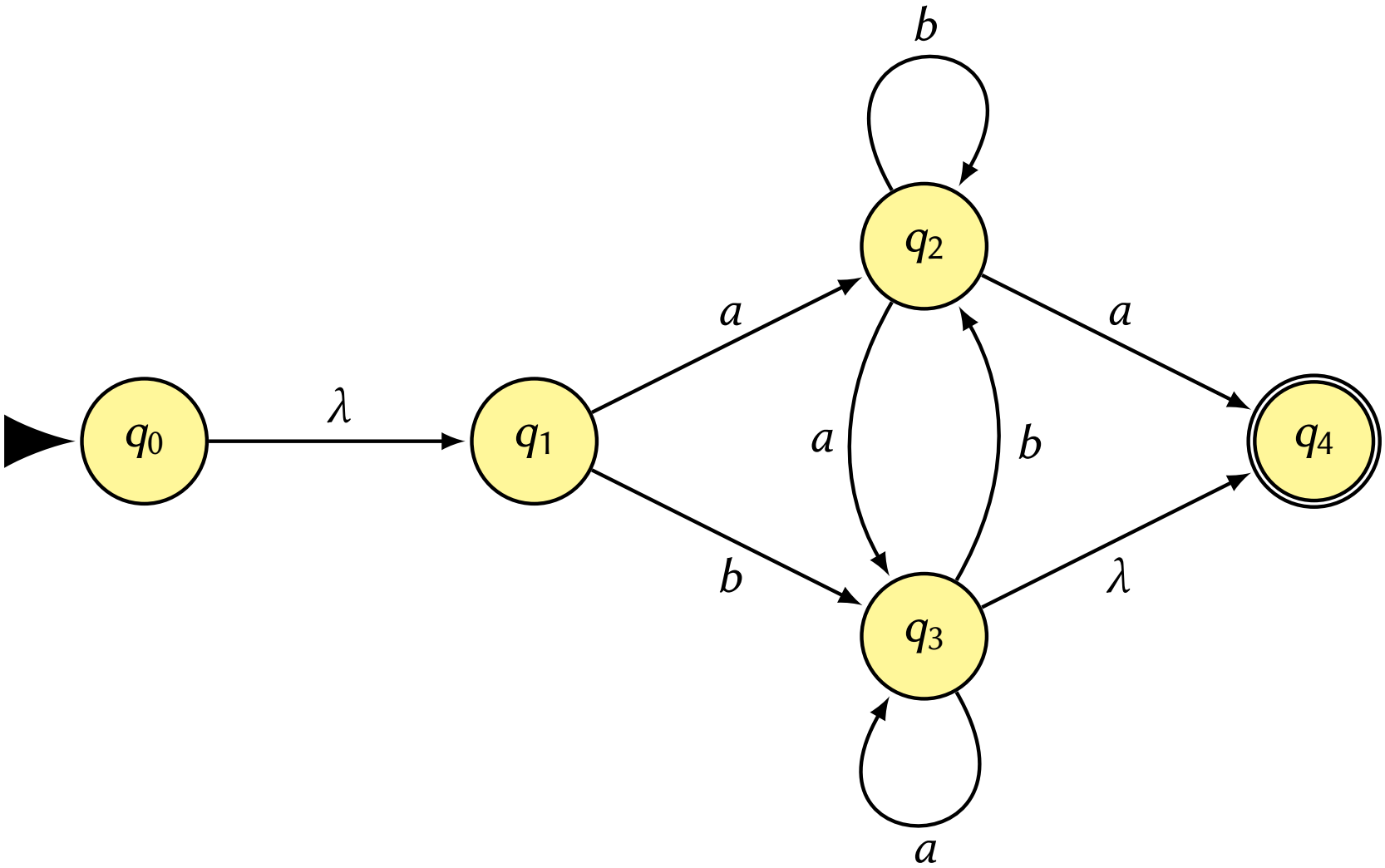
- people are not as reliably creative as they are reliable drones
- we don't want to wait for DaVinci to be in the suitable mood
- all cleverness should be incorporated into the algorithm

# Proof of Part 2: $TG \subset RE$

## Algorithm

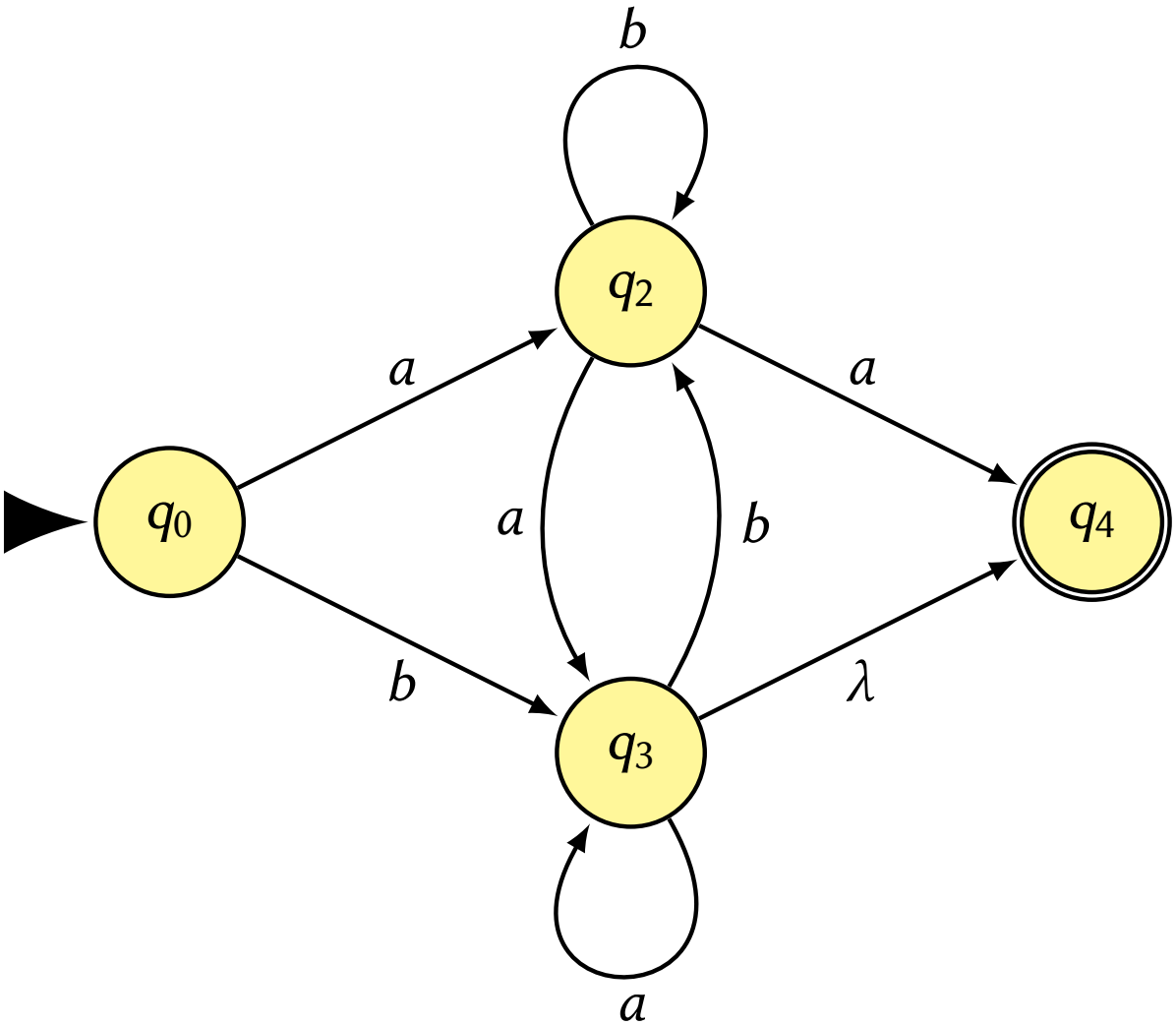
1. Create a unique, un-enterable final state and a unique, un-leaveable initial state
2. One-by-one, in *any* order, bypass and eliminate all the non-initial or final states in the TG.
   - A state is **bypassed** by connecting each incoming edge with each outgoing edge.
   - The label of each resultant edge is the **concatenation** of the label on the incoming edge with the label on the loop edge (if there is one) and the label on the outgoing edge
3. When two states are joined by more than one edge going in the same direction, **unify** them by adding their labels
4. Finally, when all that remains is one edge from "initial" to "final", the label on that edge is a regular expression that generates the same language as what was recognized by the original machine

# Example for Part 2: $TG \subset RE$



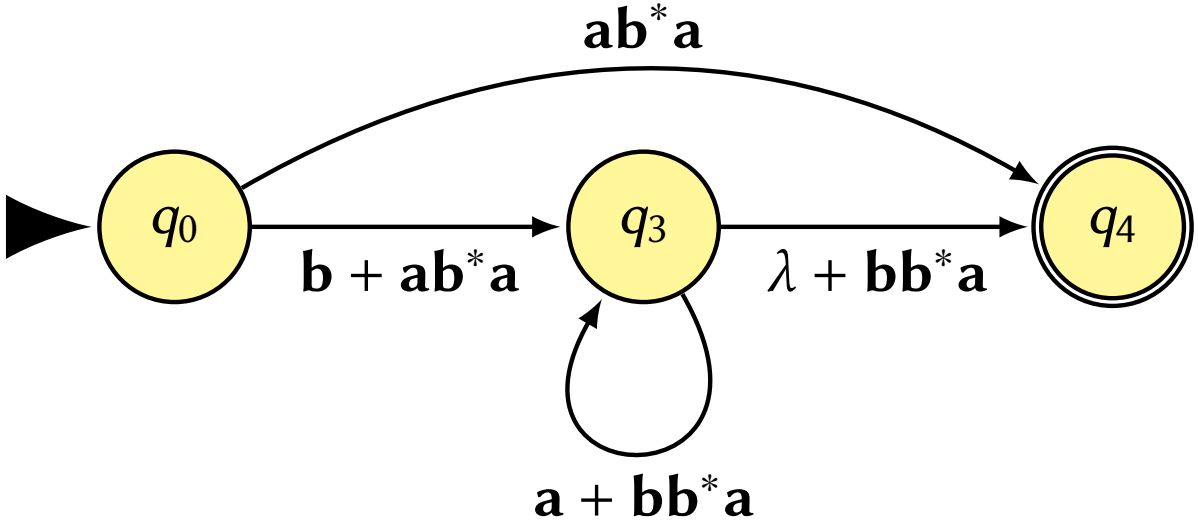

Eliminating states in the order: $q_1, q_2, q_3$

# Example for Part 2: $TG \subset RE$



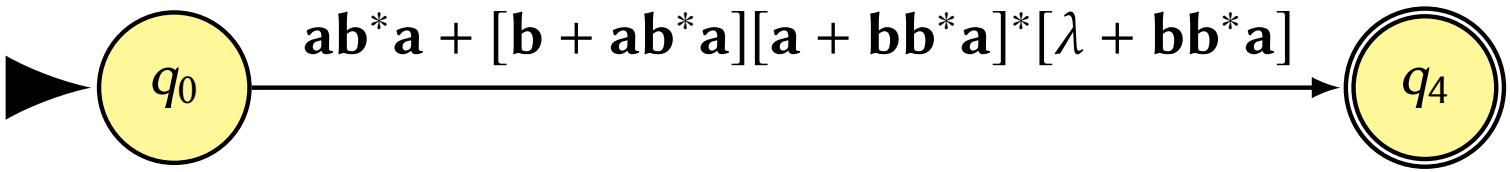

After eliminating $q_1$

# Example for Part 2: $TG \subset RE$




After eliminating $q_2$

# Example for Part 2: $TG \subset RE$

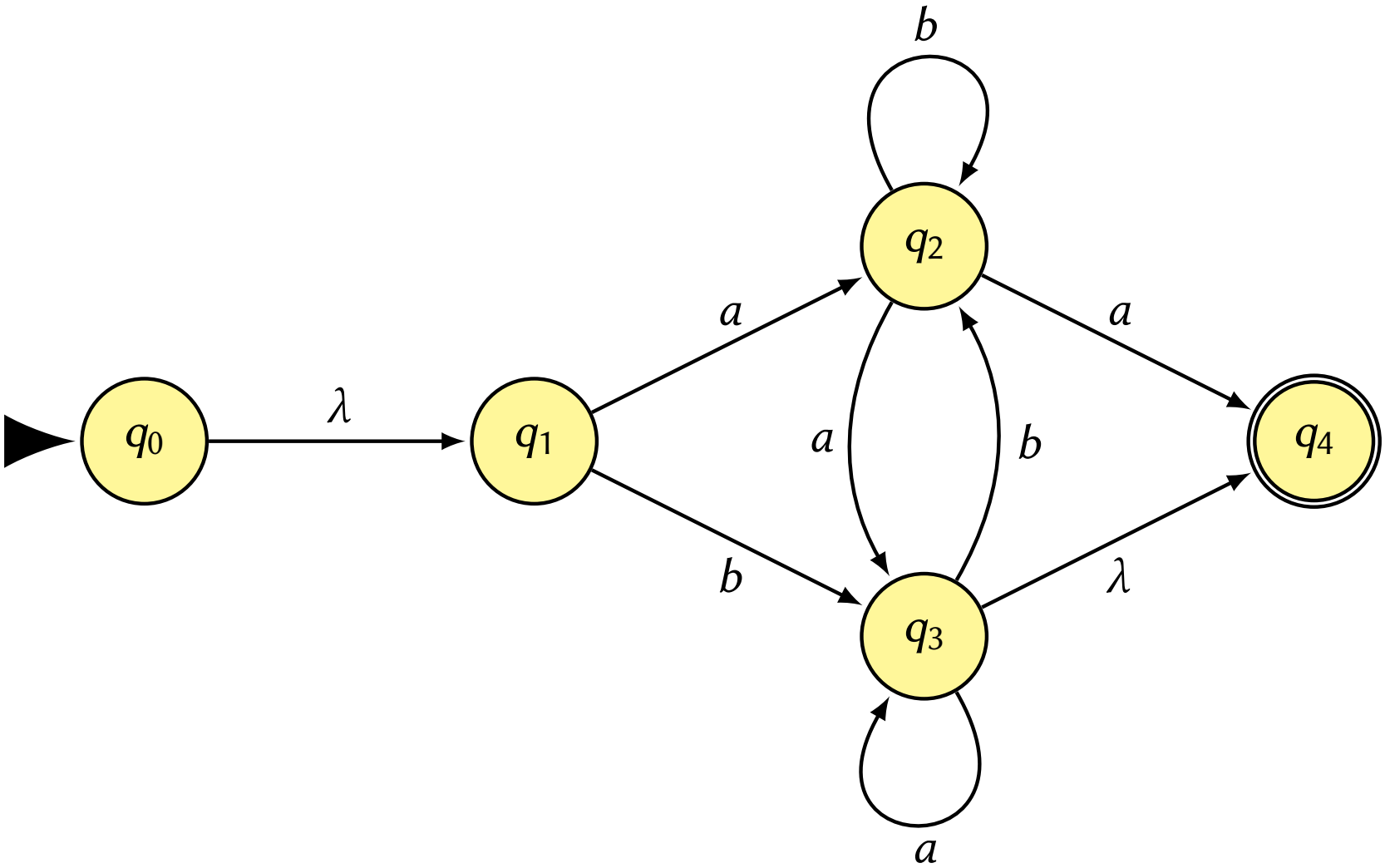



After eliminating $q_3$

Yielding:

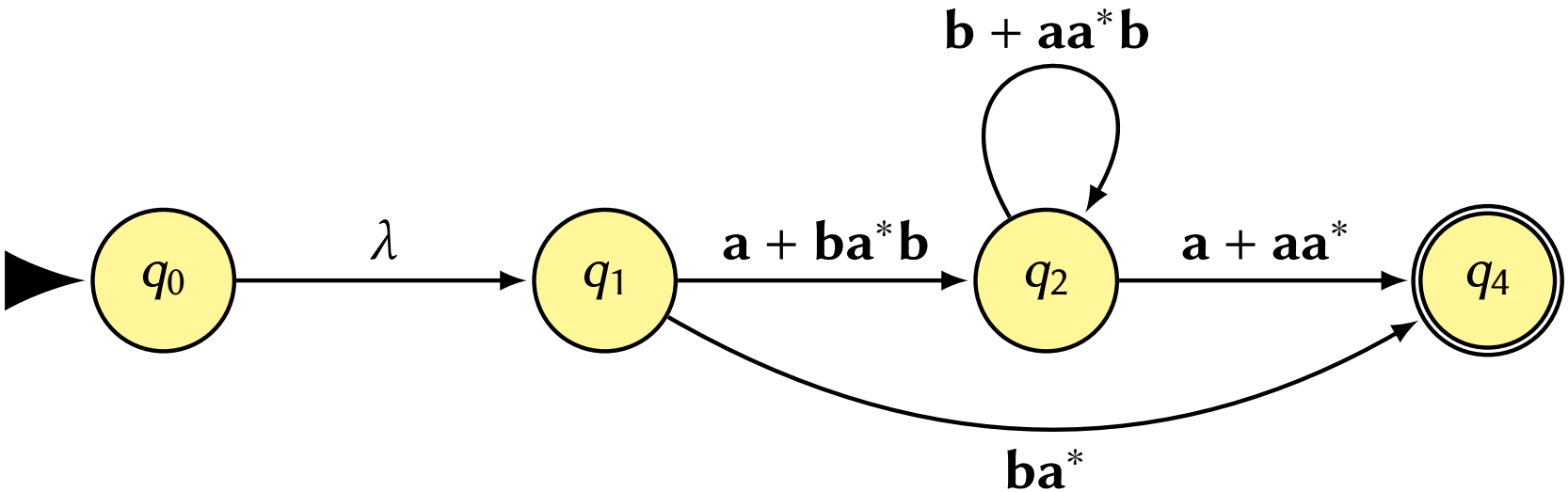
$$\mathbf{ab^*a} + [\mathbf{b} + \mathbf{ab^*a}][\mathbf{a} + \mathbf{bb^*a}]^*[\lambda + \mathbf{bb^*a}]$$

# Example for Part 2: $TG \subset RE$



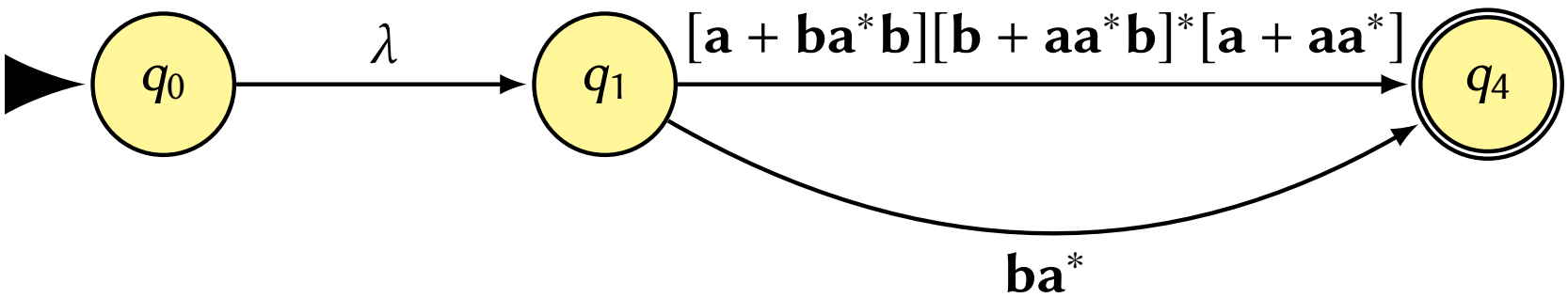

Eliminating states in the order: $q_3$, $q_2$, $q_1$

# Example for Part 2: $TG \subset RE$



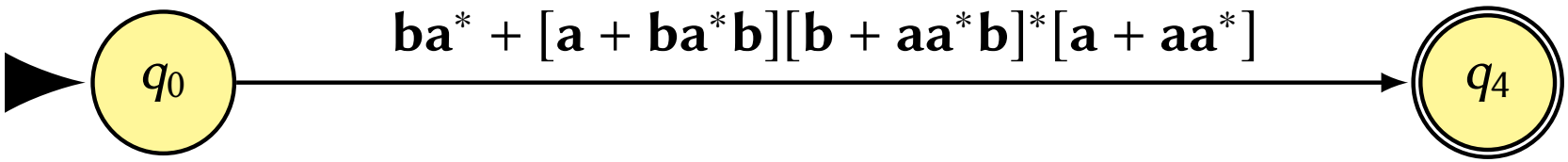

After eliminating $q_3$

# Example for Part 2: $TG \subset RE$




After eliminating $q_2$

# Example for Part 2: $TG \subset RE$




After eliminating $q_1$

Yielding:

$$\mathbf{ba^* + [a + ba^*b][b + aa^*b]^*[a + aa^*]}$$

# Proof of Part 3: $RE \subset FA$

## Proof.

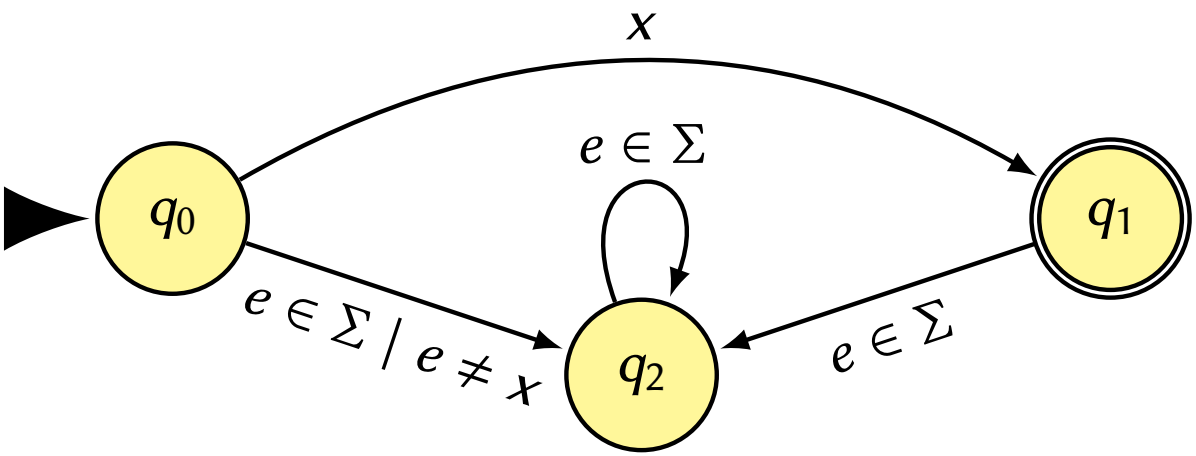
1. There is an FA that accepts any particular letter of the alphabet. There is an FA that accepts only the word $\lambda$.
2. If there is an FA called $FA_1$ that accepts the language defined by the regular expression $\mathbf{r_1}$, and there is an FA called $FA_2$ that accepts the language defined by the regular expression $\mathbf{r_2}$, then there is an FA (called $FA_3$) that accepts the language defined by $(\mathbf{r_1} + \mathbf{r_2})$, the *sum language*.
3. If there is an $FA_1$ that accepts the language defined by regular expression $\mathbf{r_1}$ and an $FA_2$ that accepts the language defined by regular expression $\mathbf{r_2}$, then there is an $FA_3$ that accepts the language defined by the concatenation $\mathbf{r_1 r_2}$, the *product language*.
4. If $\mathbf{r}$ is a regular expression and $FA_1$ accepts language $(\mathbf{r})$, then there exists $FA_2$ where it accepts the language $(\mathbf{r}^*)$ [Kleene Star].

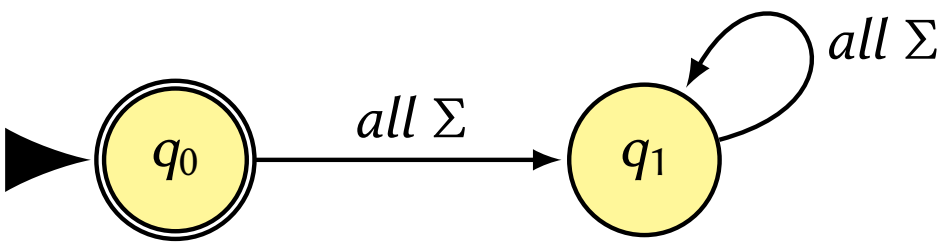
□

# Proof of Part 3 Rule 1

## Rule 1

There is an FA that accepts any particular letter of the alphabet.
There is an FA that accepts only the word $\lambda$.



# Proof of Part 3 Rule 2

## Rule 2

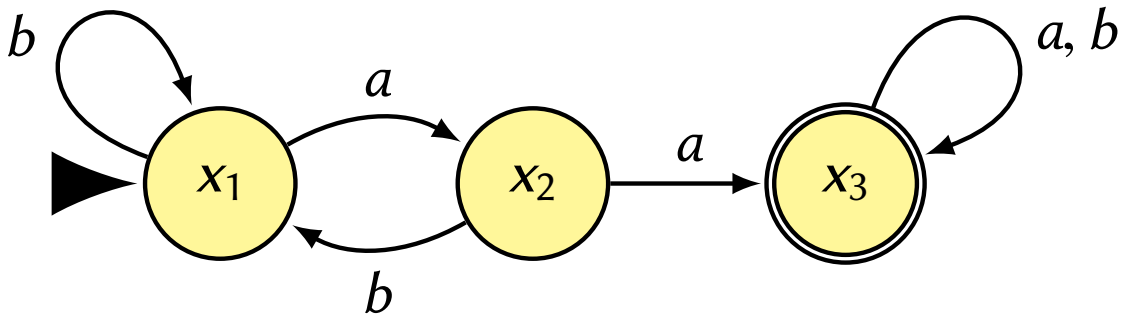
If $FA_1$ accepts language($\mathbf{r_1}$) and $FA_2$ accepts language($\mathbf{r_2}$)
then an $FA_3$ exists and accepts language($\mathbf{r_1} + \mathbf{r_2}$)

The general description is as follows:

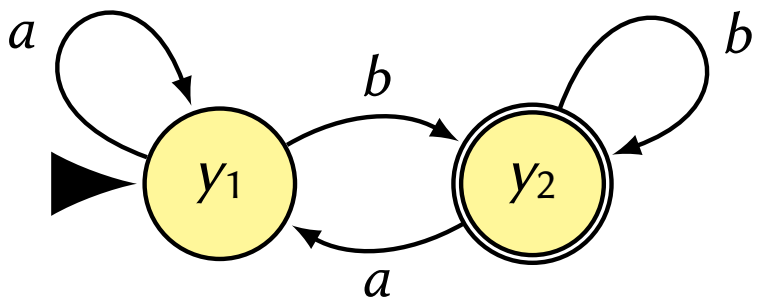
- $FA_1$ has states $x_1, x_2, x_3, \ldots$
- $FA_2$ has states $y_1, y_2, y_3, \ldots$
- We will construct $FA_3$ with states $z_1, z_2, z_3, \ldots$
  - each $z_k$ is of the form $x_i$ *or* $y_j$
  - $x_{start}$ *or* $y_{start}$ is the start state of $FA_3$
  - $z_k$ ($= x_i$ *or* $y_j$) is a final state IFF $x_i$ is final or $y_j$ is final

# Example for Part 3 Rule 2

$FA_1$




$FA_2$




*Remainder of Exercise on Chalkboard (show* $(L_1 + L_2)$*)*

# Proof of Part 3 Rule 3

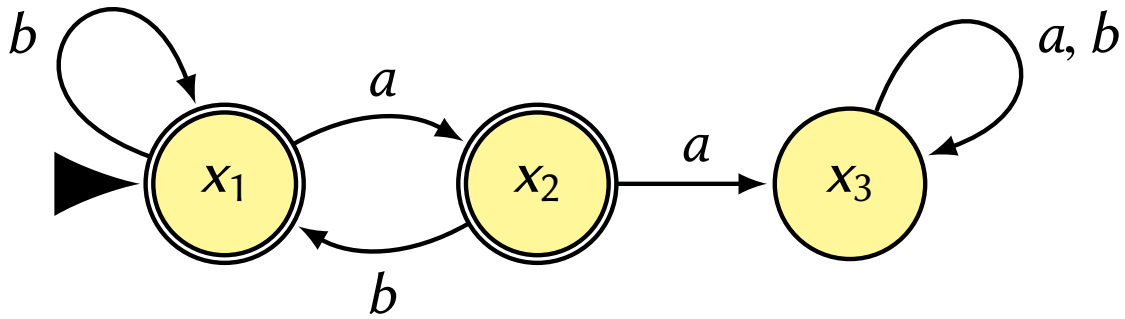
## Rule 3

If $FA_1$ accepts language($\mathbf{r_1}$) and $FA_2$ accepts language($\mathbf{r_2}$),
then an $FA_3$ exists and accepts language($\mathbf{r_1 r_2}$)

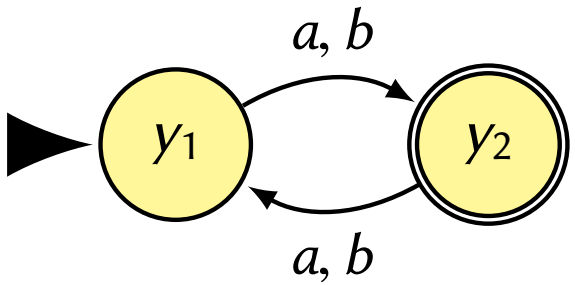
- Make a $z$-state for every non-final $x$-state in $FA_1$, reached before ever hitting a final state on $FA_1$
- For each final state in $FA_1$, we establish a $z$-state that expresses either (1) we are continuing on $FA_1$ or beginning on $FA_2$.
- Initial states are the initial states from $FA_1$
- Final states are the $z$-states that represent the disjunction of any final state from $FA_2$

# Example of Part 3 Rule 3

$FA_1$




$FA_2$




*Remainder of Exercise on Chalkboard (show $L_1 L_2$)*

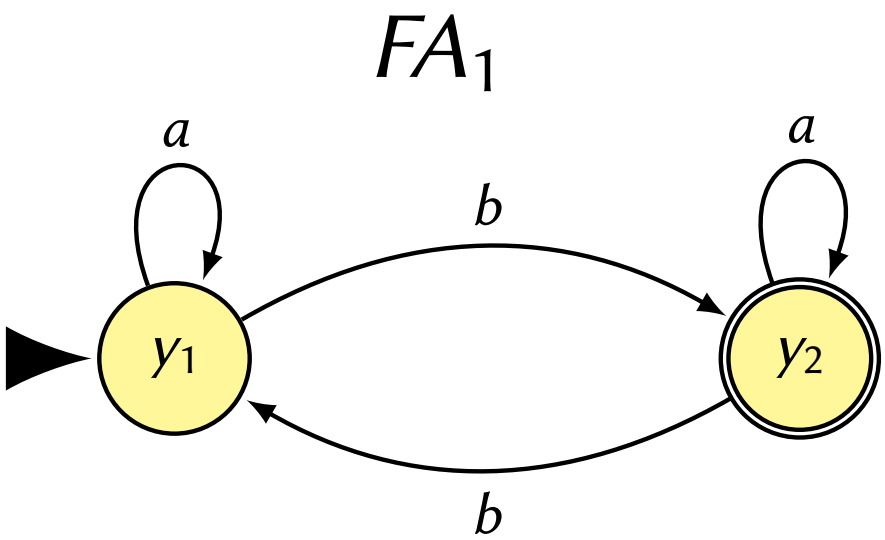
# Proof of Part 3 Rule 4

## Rule 4

If **r** is a regular expression and $FA_1$ accepts language (**r**), then there exists $FA_2$ where it accepts language (**r**$^*$) (a.k.a. the Kleene Star)

- Create a state for every subset of $x$'s. Cancel any subset that contains a final $x$-state, but does not contain the start state.
- For all remaining non-empty states, draw an $a$-edge and $b$-edge to the collection of $x$-states reachable in the original FA from the component $x$'s by $a$- and $b$-edges, respectively.
- Call the null subset a initial-and-final state and connect it to whatever states the original start state is connect to by $a$- and $b$-edges (even the start state)
- Finally, mark states as final if the contain an $x$-component that is a final state of the original FA

# Example of Part 3 Rule 4




*Remainder of Exercise on Chalkboard (show $L_1^*$)*

# Part 3 Recap

1. There is an FA that accepts any letter or $\lambda$
2. If $FA_1$ accepts $\mathbf{r_1}$ and $FA_2$ accepts $\mathbf{r_2}$, then there exists $FA_3$ accepting $\mathbf{r_1 r_2}$
3. If $FA_1$ accepts $\mathbf{r_1}$ and $FA_2$ accepts $\mathbf{r_2}$, then there exists $FA_3$ accepting $\mathbf{r_1 + r_2}$
4. If $FA_1$ accepts $\mathbf{r_1}$ then there exists $FA_2$ accepting $\mathbf{r_1}^*$

We can construct any regular expression through reapplications of the following four rules above.

Therefore, any RE can be converted into an FA.

# Nondeterministic Finite Automata

- Finite Automata introduced up to this point have all been **deterministic**
- Transition Graphs are **non-deterministic** by default but can also be much more complicated.
- Maybe there exists a happy medium between the two?

# Nondeterministic Finite Automata

- Finite Automata introduced up to this point have all been **deterministic**
- Transition Graphs are **non-deterministic** by default but can also be much more complicated.
- Maybe there exists a happy medium between the two?

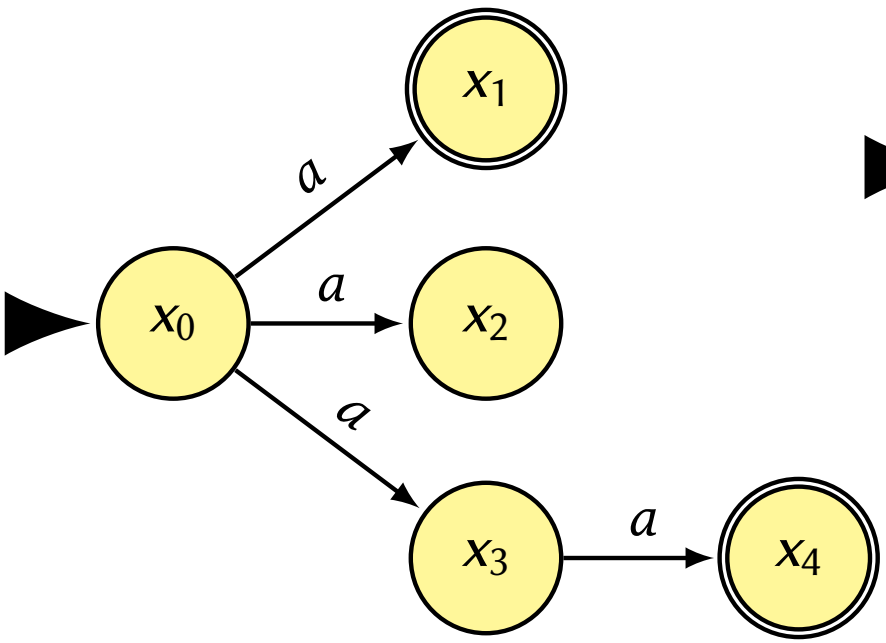
## NFA

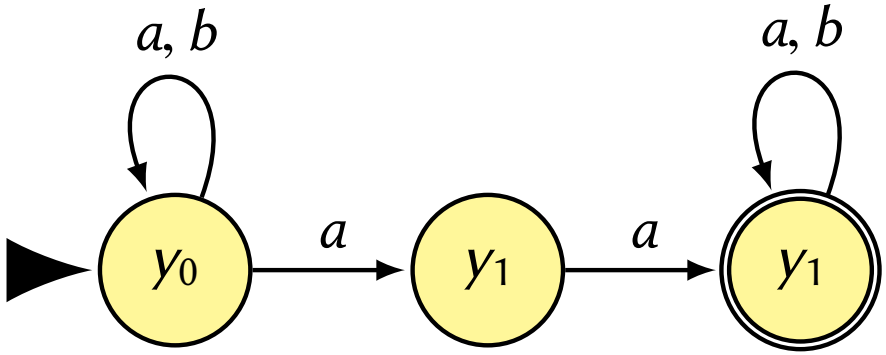
is a type of Transition Graph

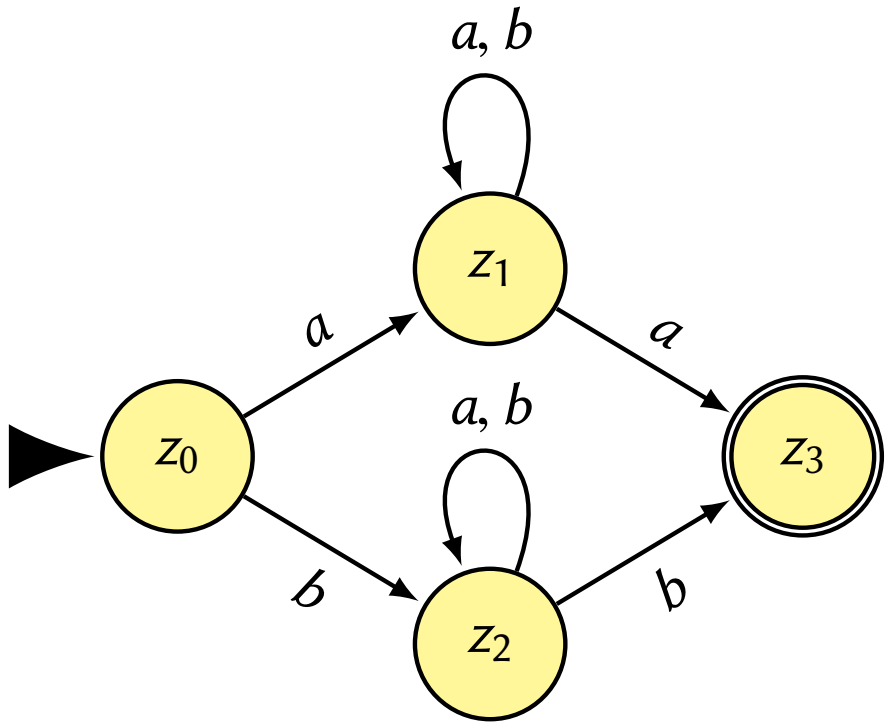
- which has a unique start state
- where each edge has a single alphabet letter
- and where many $a$- and $b$-edges could come out of each state

Invented by Rabin and Scott in 1959

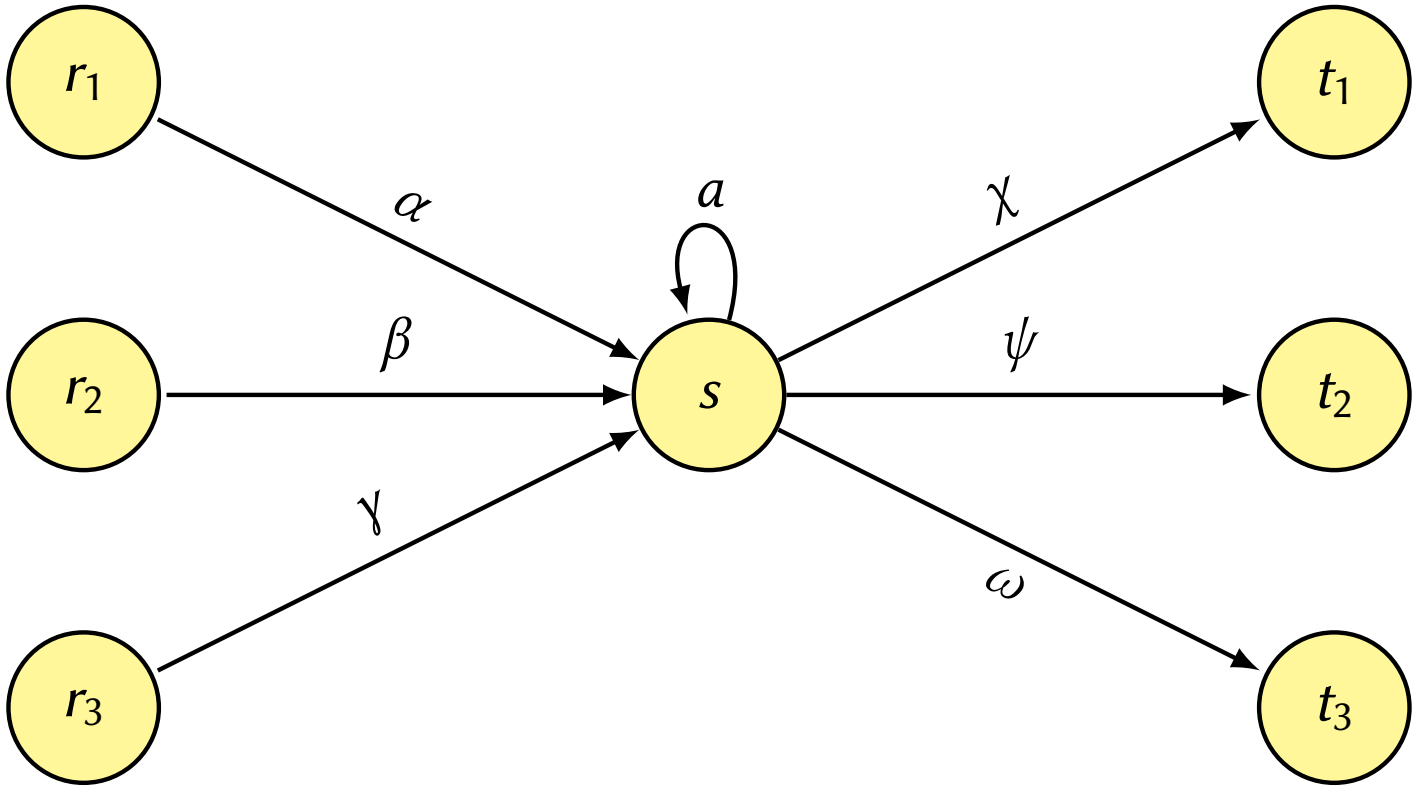
# Examples of NFAs



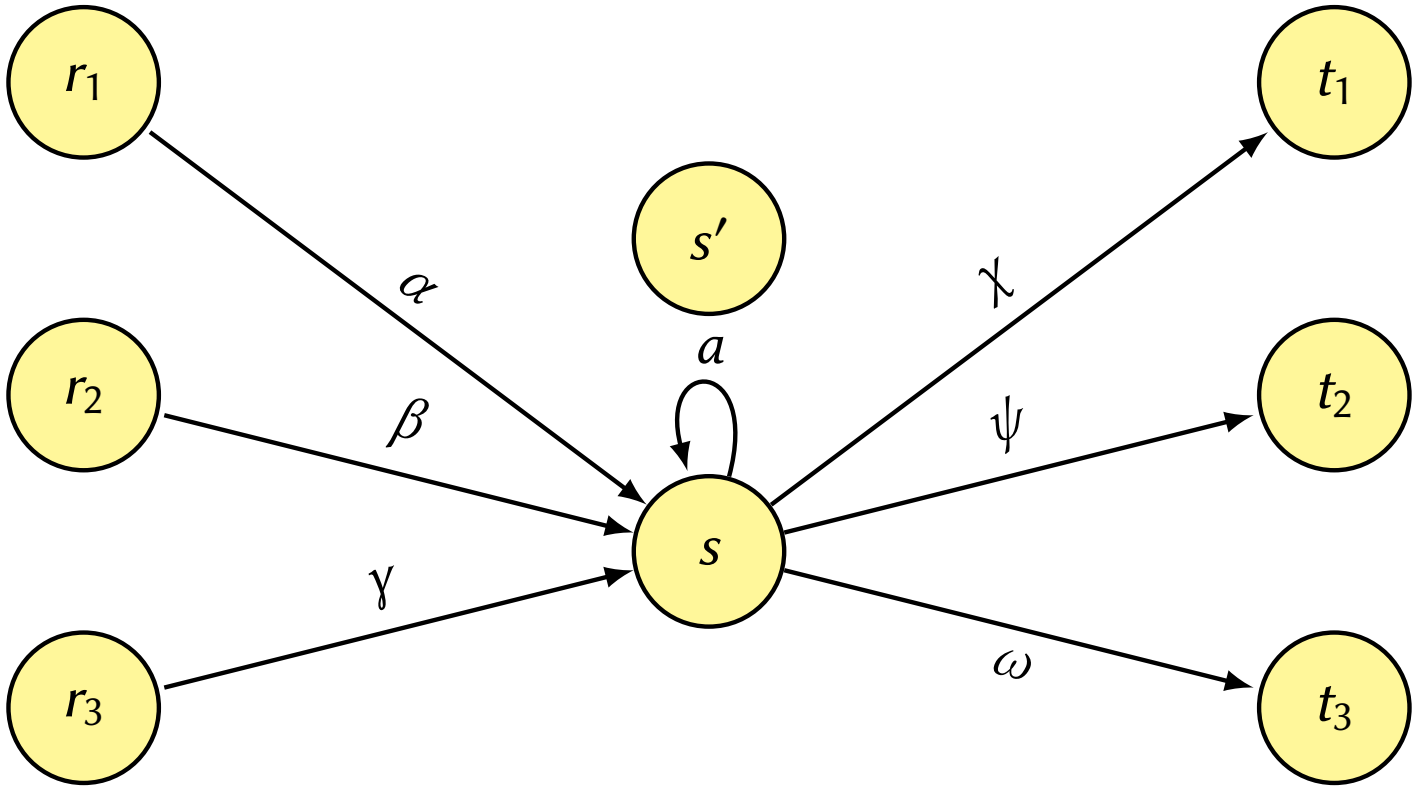



# Eliminate all Loop States



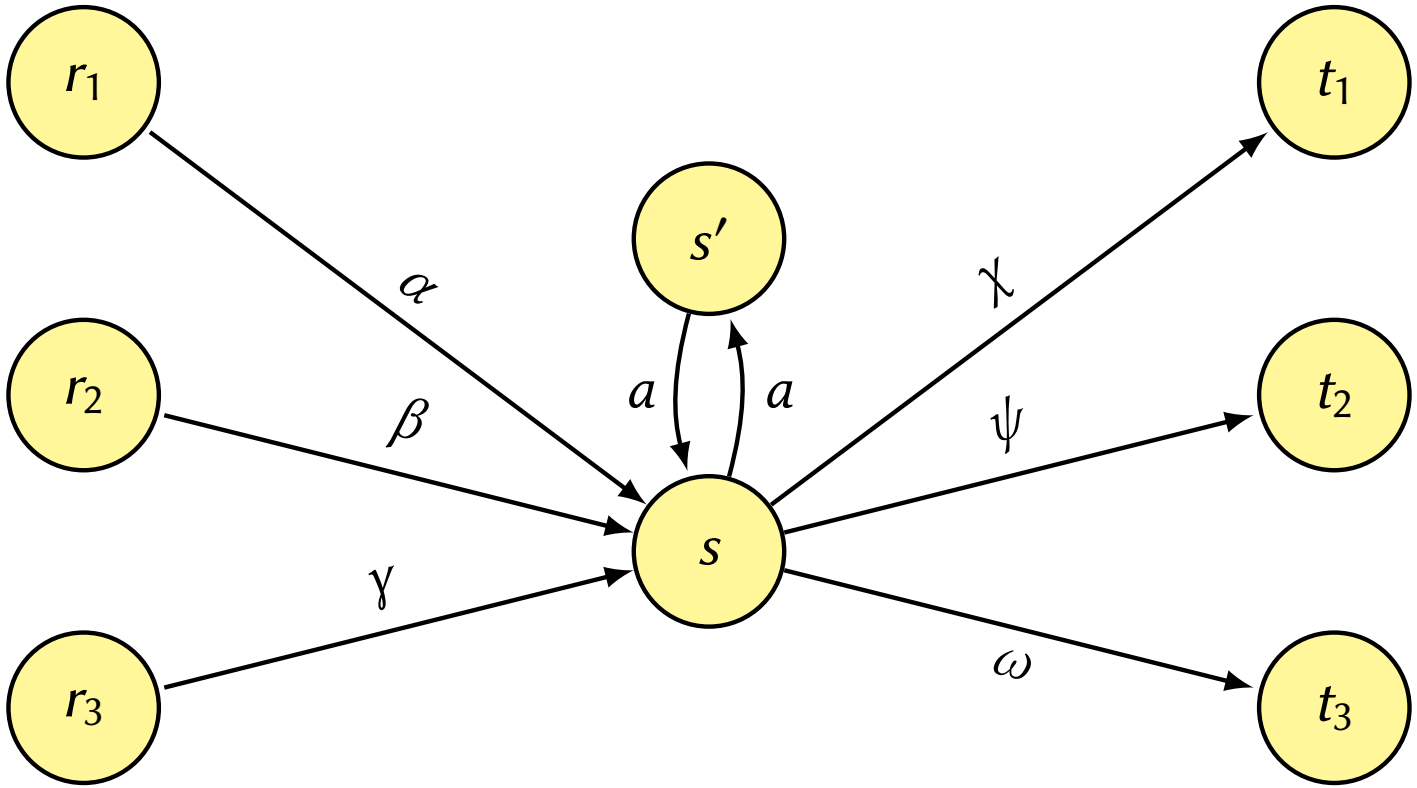

*Maybe it would be cool if we could remove all loops?*

# Eliminate all Loop States



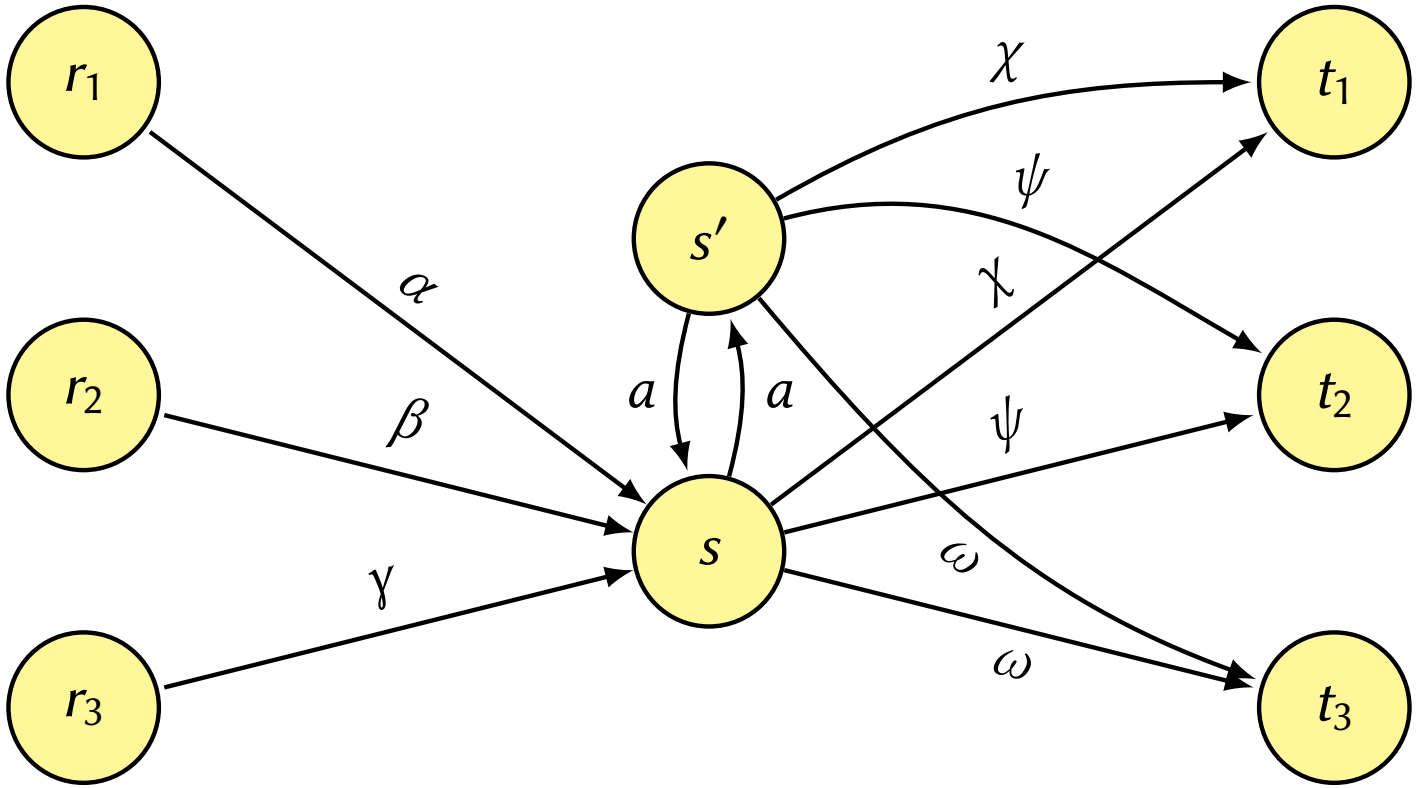

*duplicate the state*

# Eliminate all Loop States




*remove loop and add transitions*

# Eliminate all Loop States




*copy all outgoing transitions to new state*

# Example

All strings with a triple *a* followed by a triple *b*.

Should we accept *bbbaaabbb* ?

# Theorems and Proofs of NFAs

## Theorem

*For every NFA, there is some FA (DFA) that accepts exactly the same language*

## Proof.

1. Using Part 2 from before, convert the NFA into an RE
2. Using the four rules from Part 3 from before, construct an FA that accepts the RE generated in (1)
3. Because we proved each part of the proof prior, we are done

□

# Theorems and Proofs of NFAs

## Theorem

*For every NFA, there is some FA (DFA) that accepts exactly the same language*

## Proof.

1. Using Part 2 from before, convert the NFA into an RE
2. Using the four rules from Part 3 from before, construct an FA that accepts the RE generated in (1)
3. Because we proved each part of the proof prior, we are done

□

Wait... isn't that crazy complicated and borderline cheating?

# NFA to DFA Algorithm

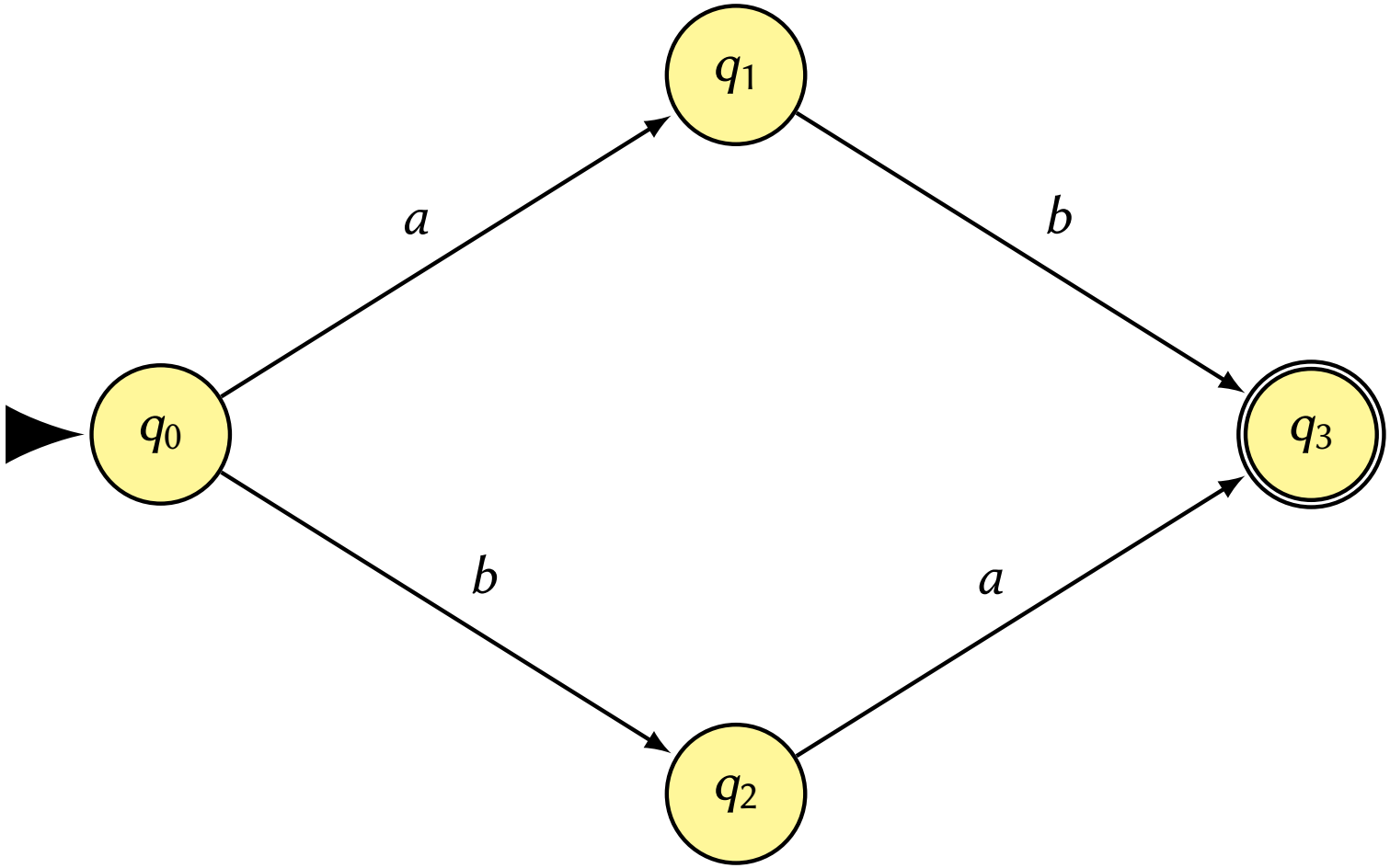
**The Big Picture:** Remember "Rule 4" from Part 3 which tells us to represent new states as a choice of $x$-states

All states in the (D)FA we will produce will also be the collections of states from the original NFA.

Every time we encounter a transition, we must "follow" that transition for each state in where we were

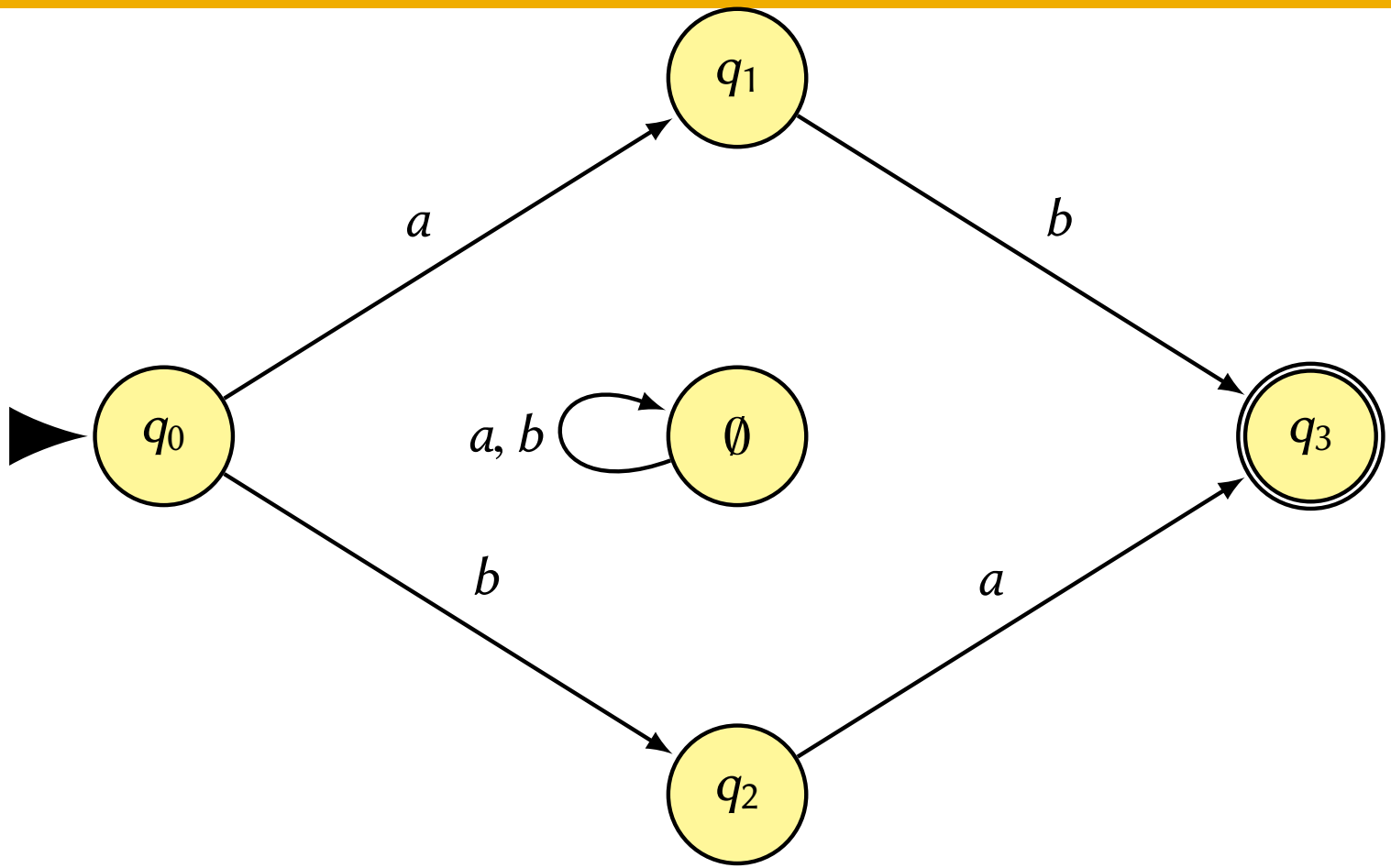
Every time we try to take an $a$- or $b$-transition which does not *collectively* exist across all NFA states in our current collection, we go to a common **hell state** – of which there is never any escape

# NFA to DFA – Trap State



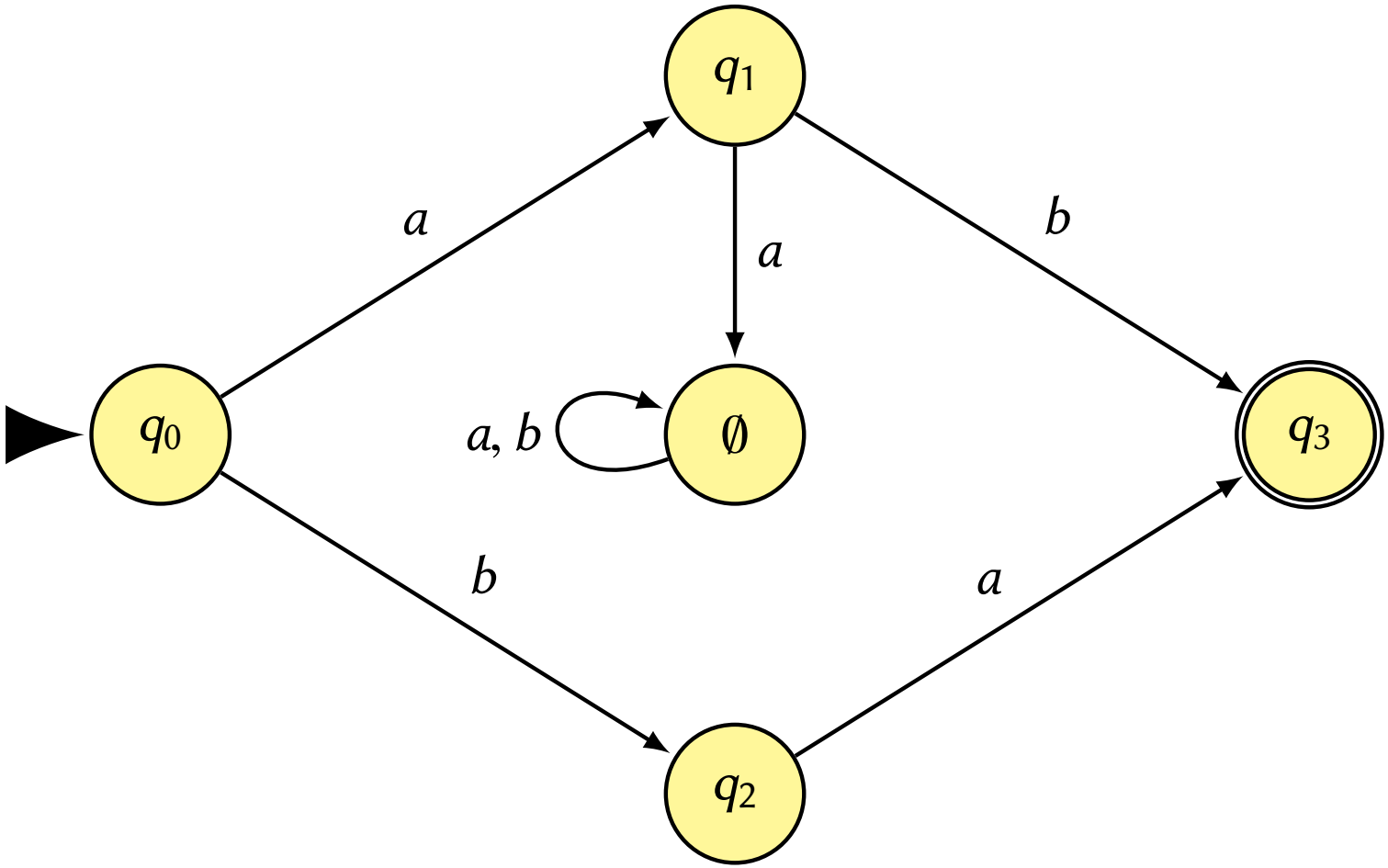

At $q0$ we have exactly one $a$-transition and one $b$-transition. Next!

# NFA to DFA – Trap State

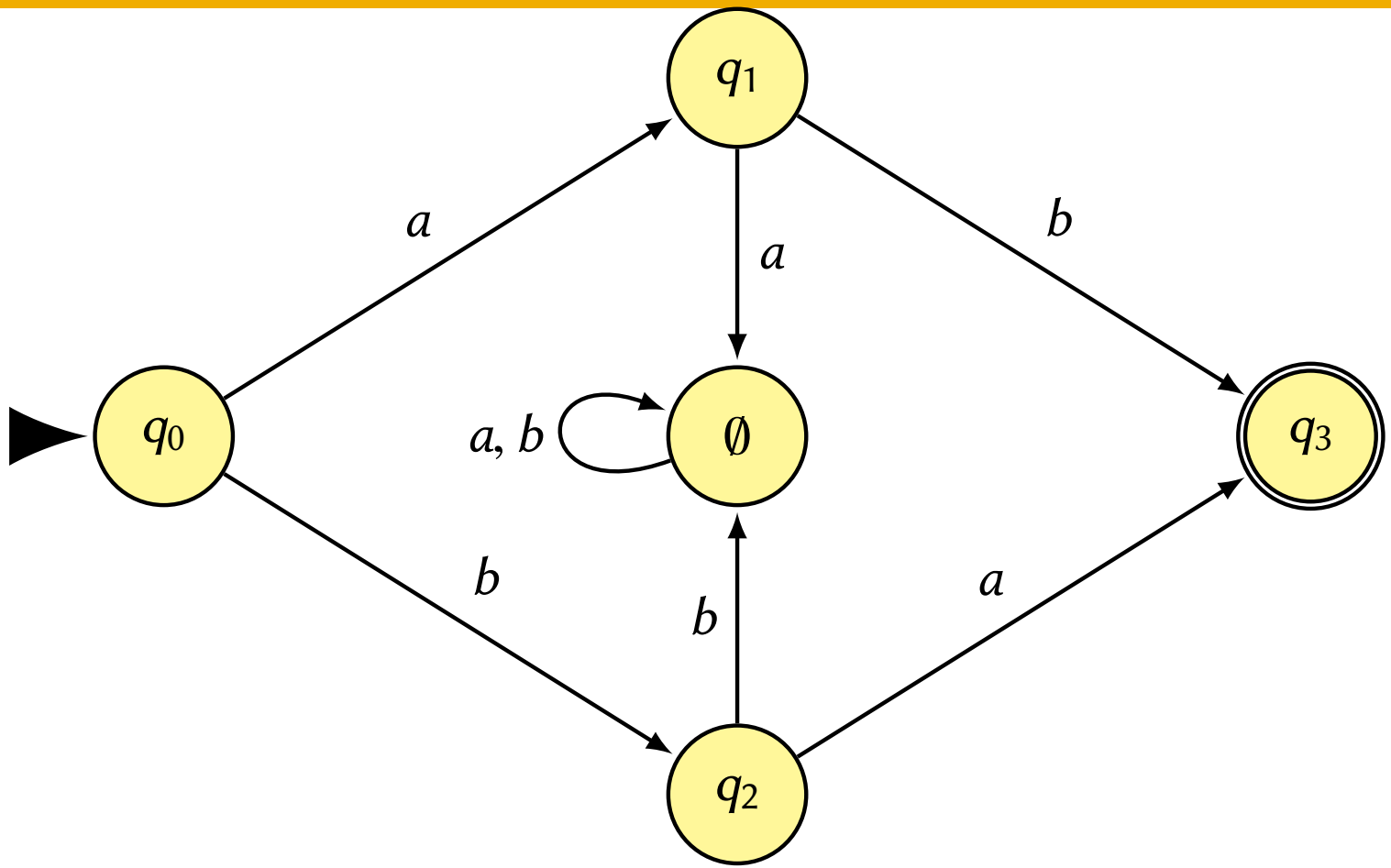



At $q1$ we have one $b$-transition BUT no $a$-transition.
We must (1) create a trap state and (2) make an $a$-transition to it

# NFA to DFA – Trap State

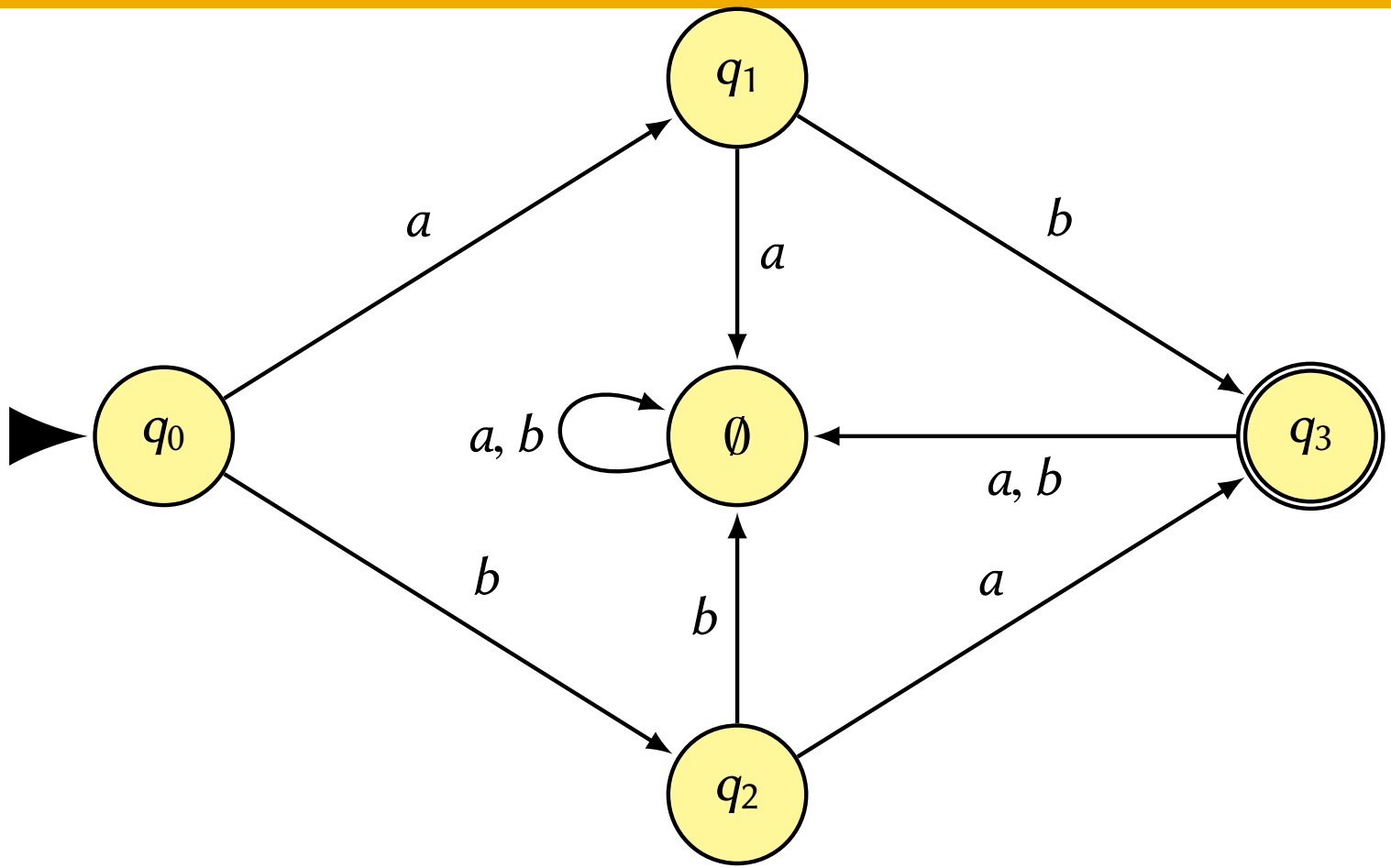


# NFA to DFA – Trap State




At $q2$ we have one $a$-transition BUT no $b$-transition.
We must make an $b$-transition from $q2$ to the trap state

# NFA to DFA – Trap State

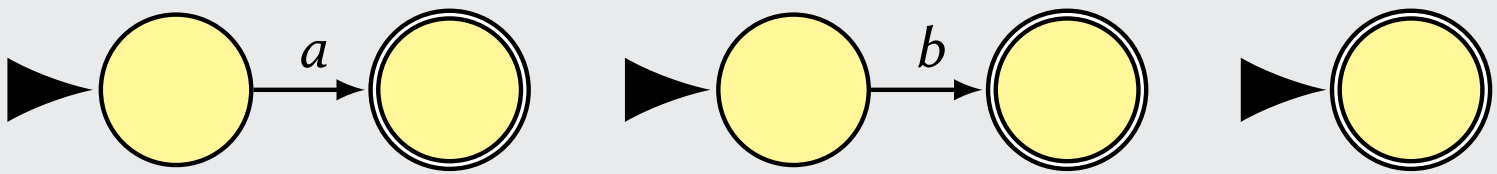



At $q3$ we have no transitions, so we must create a transition for $a$ and $b$ to the trap state

# Adding NFAs to Kleene's Theorem

**Proof.**

1. The following NFAs can accept $\{a\}$, $\{b\}$ and $\{\lambda\}$ respectively




2. Because of our theorem *"For every NFA there is some FA that accepts exactly the same language"*, there is an equivalent TG and RE for a given NFA

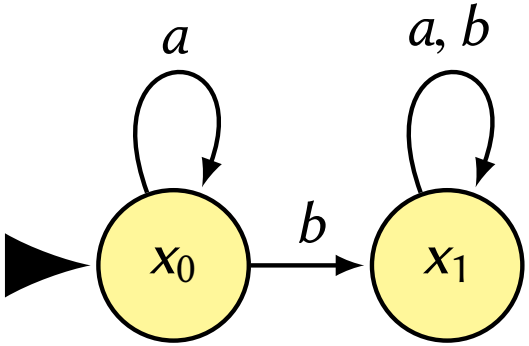
□

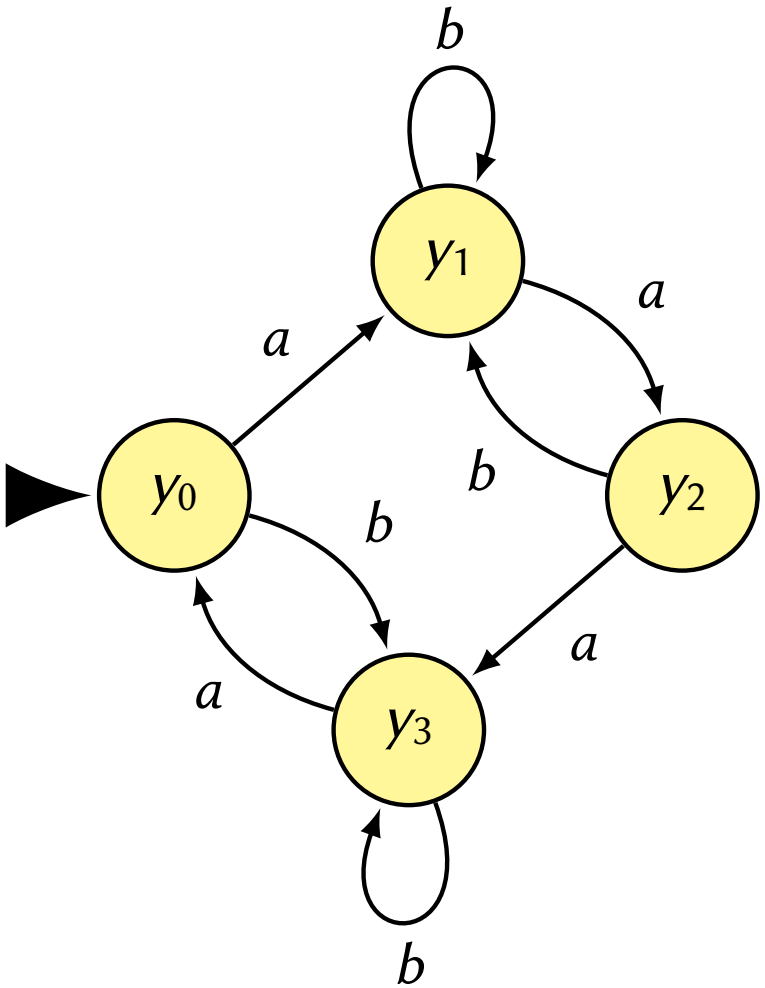
# Simplifying the creation of $FA_1 + FA_2$

## Algorithm

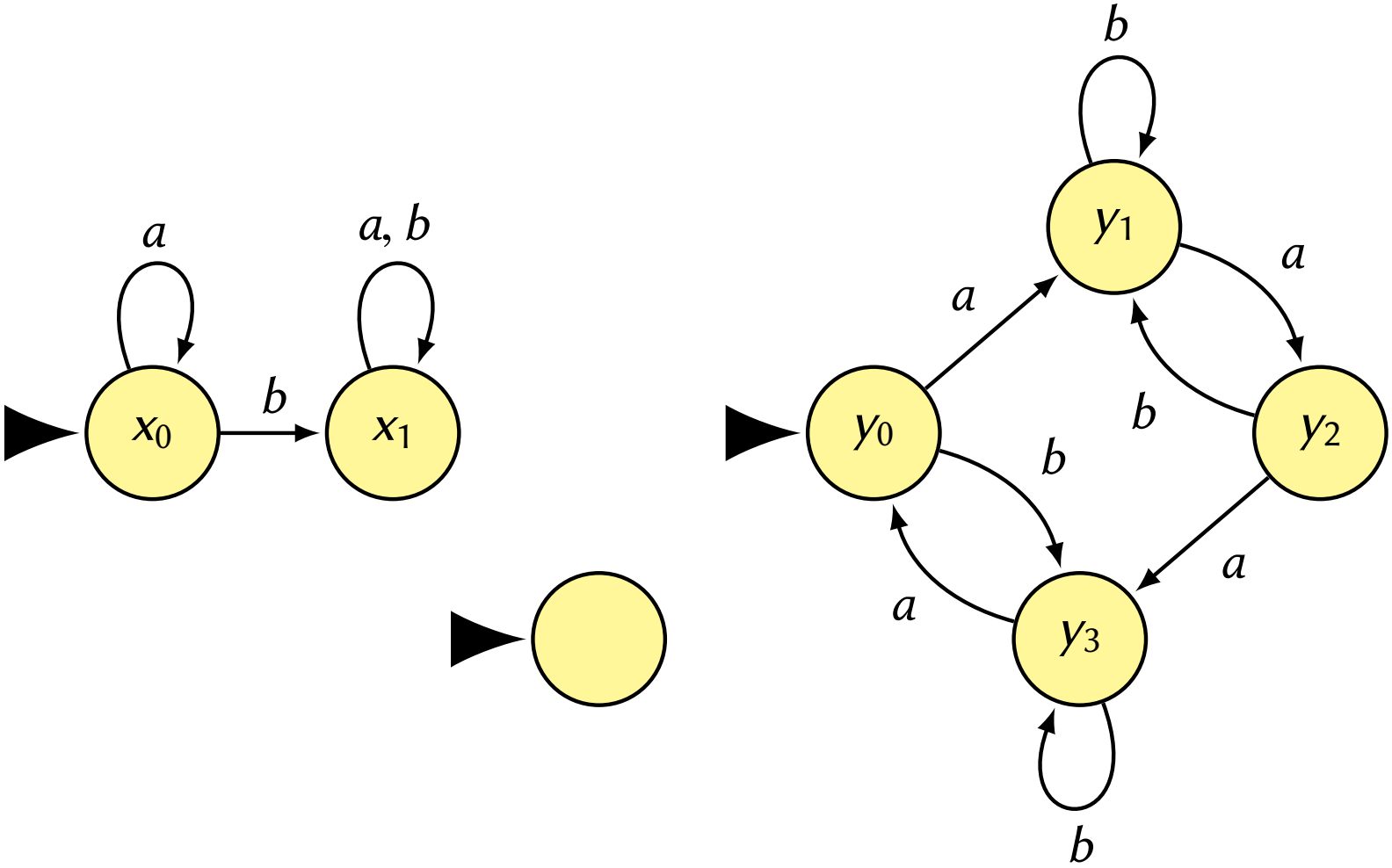
1. Introduce a **new and unique** state state with two outgoing $a$-edges and two outgoing $b$-edges
2. Connect each of the edges to the states that follow the start of both $FA_1$ and $FA_2$.
3. Remove the "start" markers for the states which had originally started $FA_1$ and $FA_2$
4. Using the algorithm provided earlier, convert the NFA into an FA

# Example



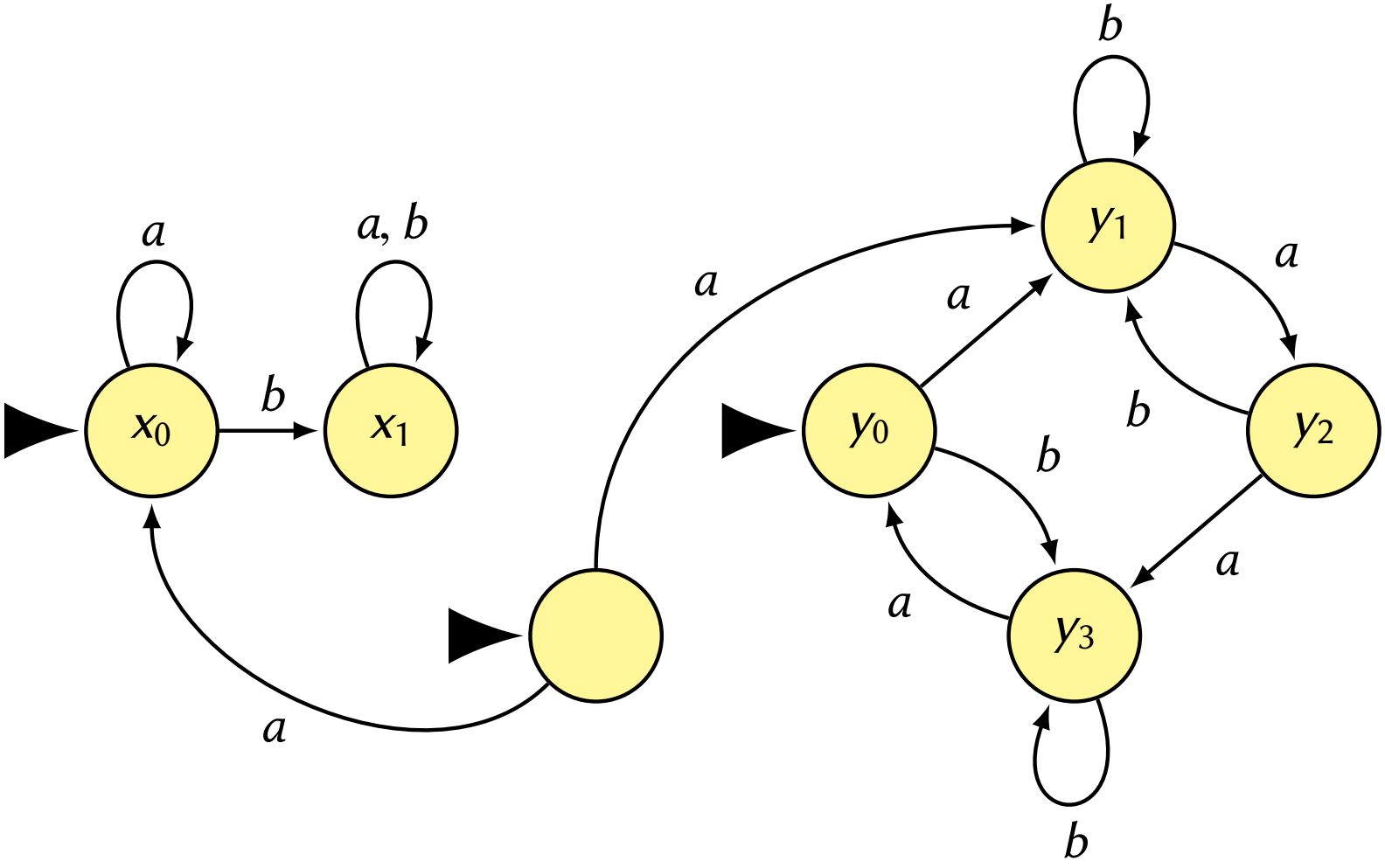



# Example



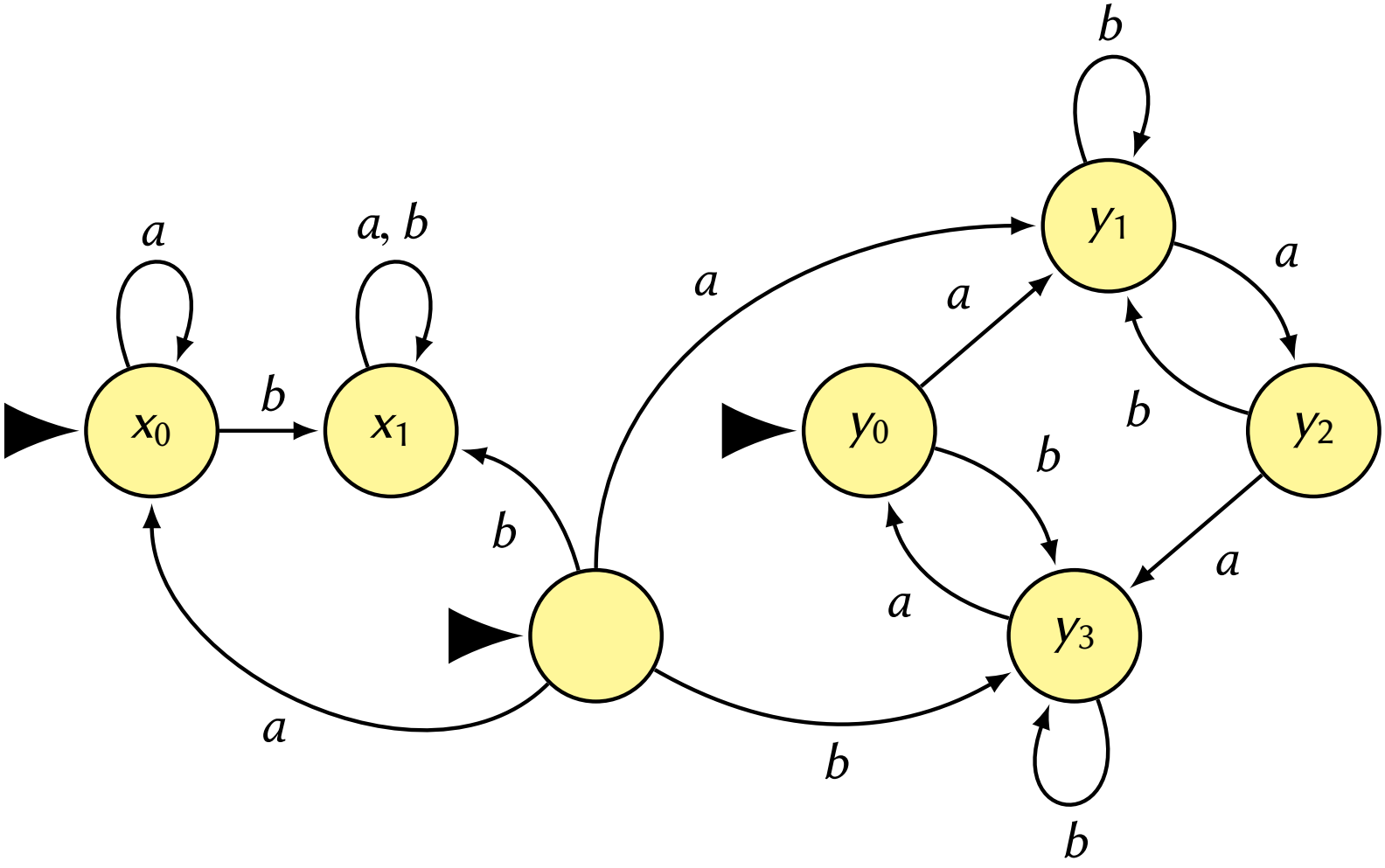

*Introduce a new and unique start state*

# Example



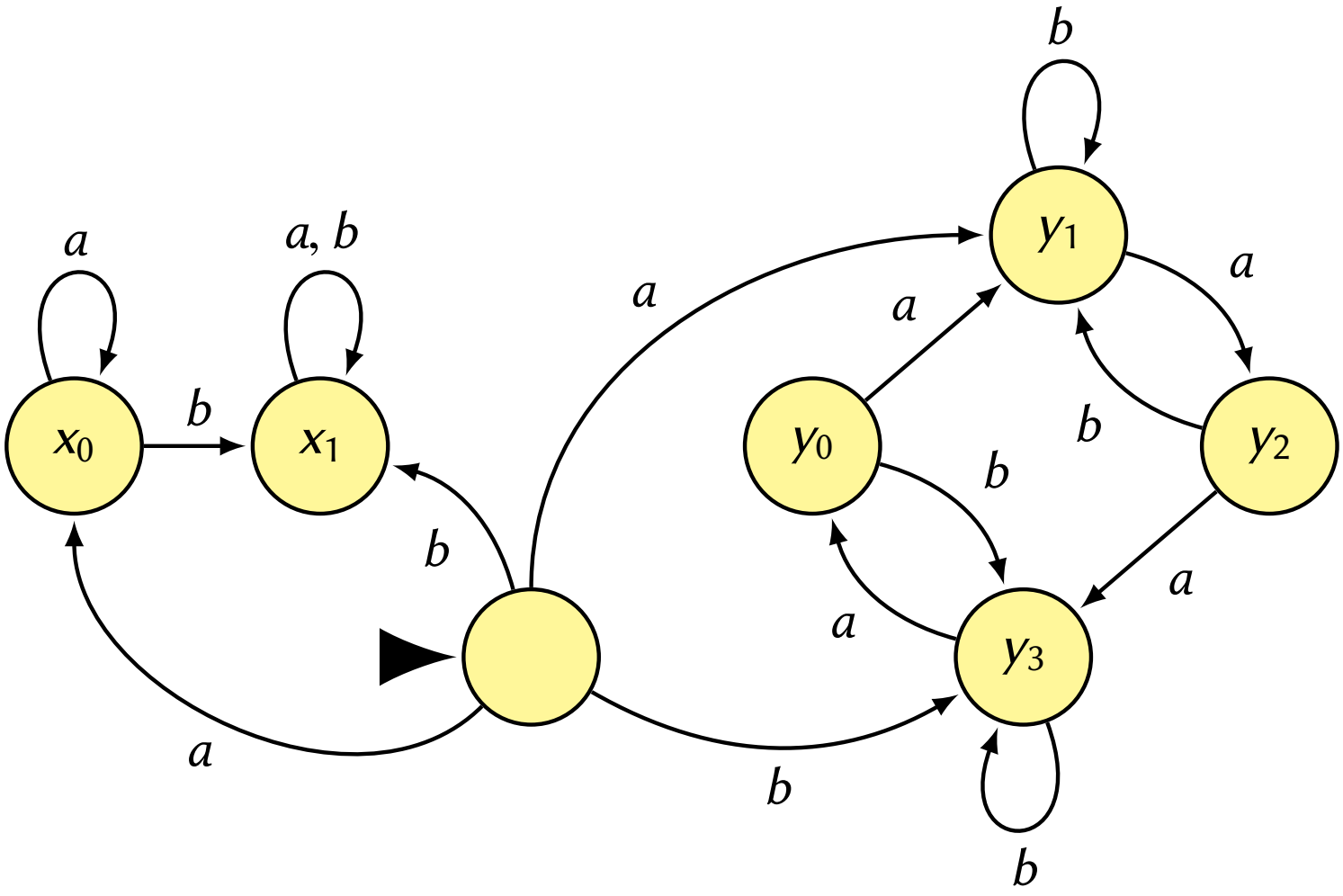

*Add outgoing a-transitions to the* **follow** *states of each FA*

# Example




*Add outgoing b-transitions to the* **follow** *states of each FA*

# Example




*Remove the two "start"s which had originally started the two FAs*