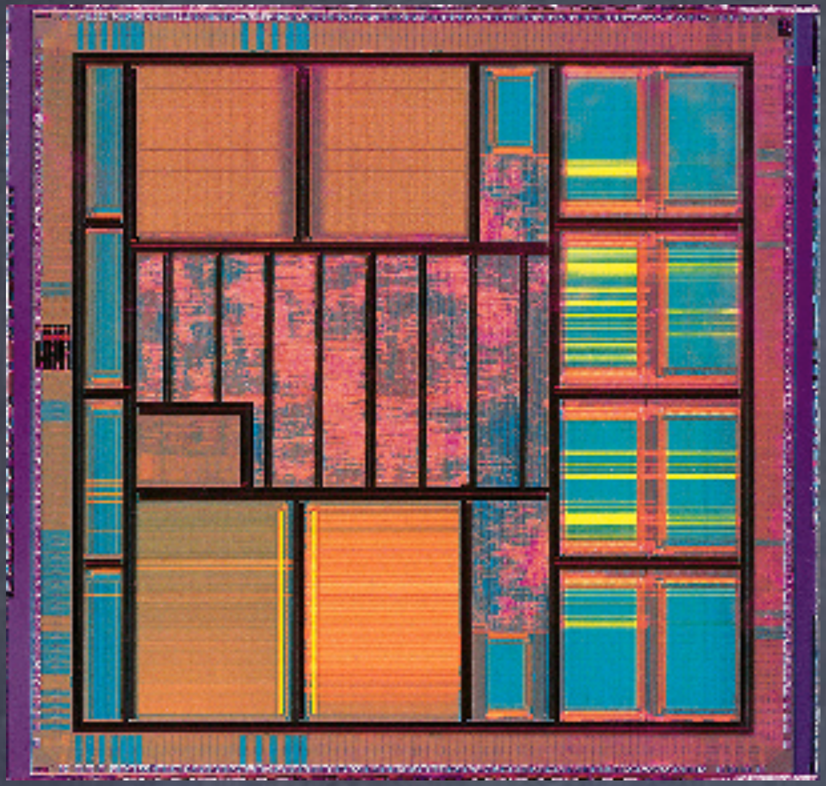
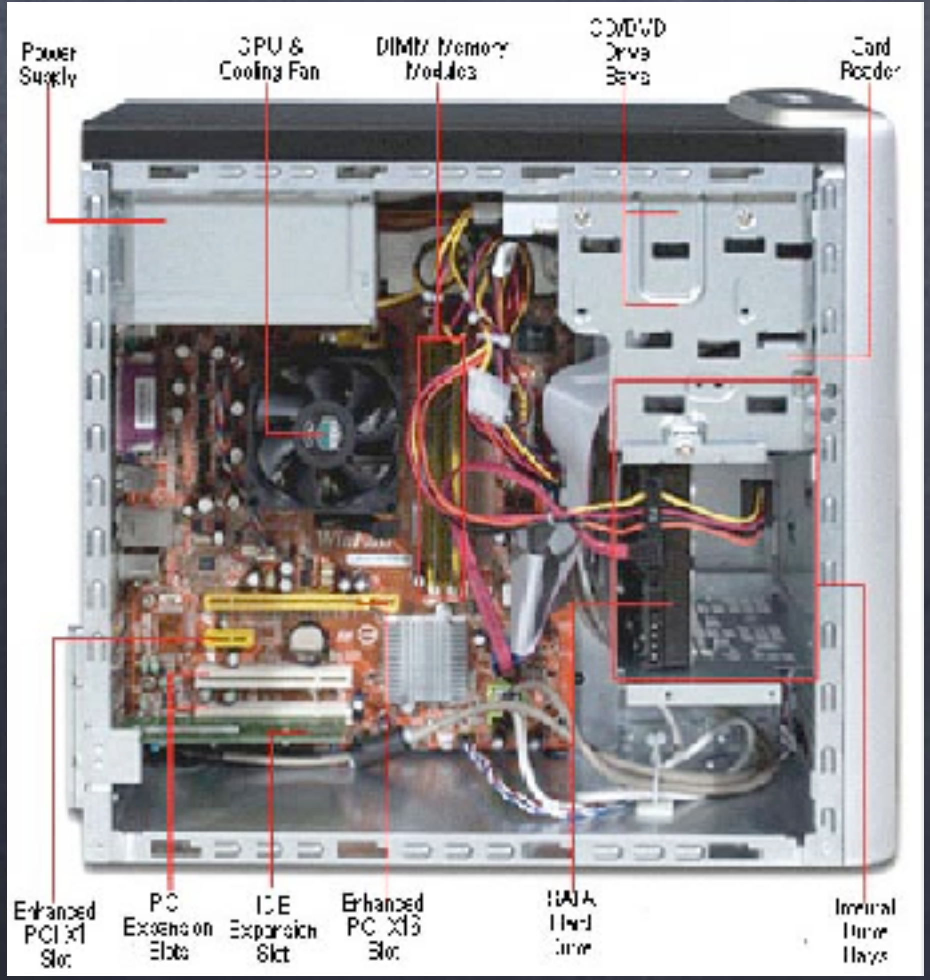
CSC173
Computation and Formal Systems

Lecture 1.1

Announcements

- Watch BlackBoard for announcements re: Study Sessions THIS WEEK
    - Focus: C programming setup and help
- Project 1 will be posted after class

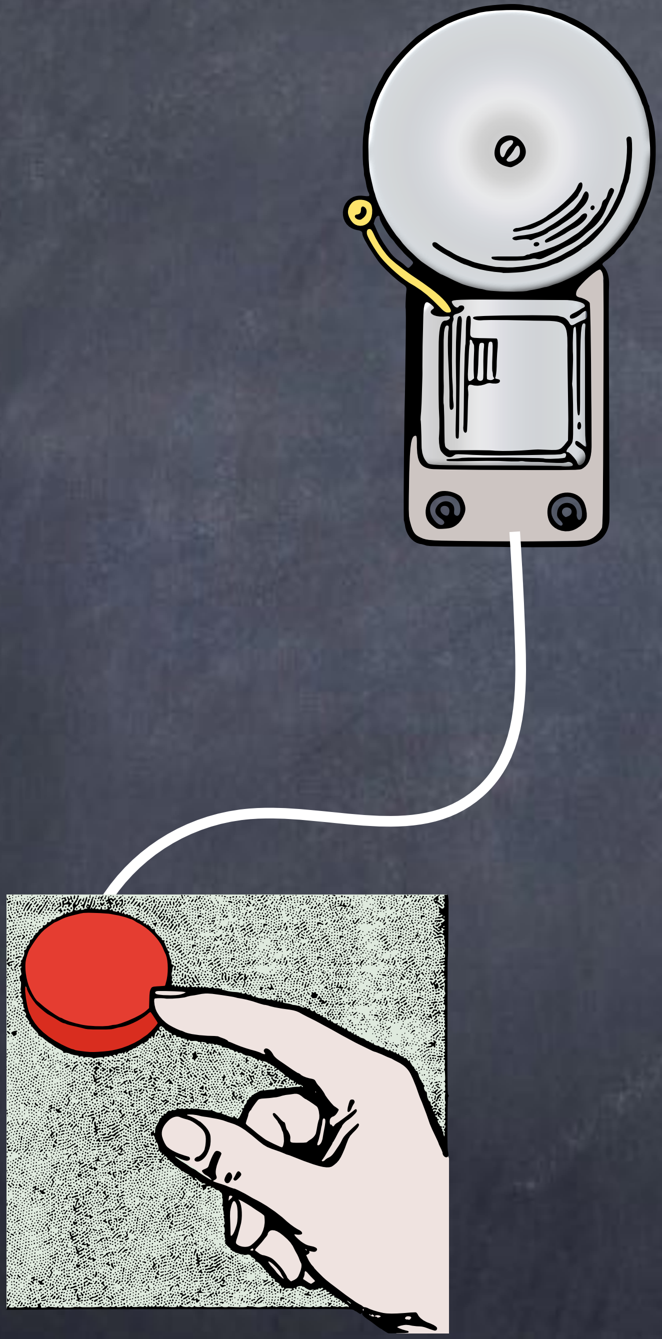


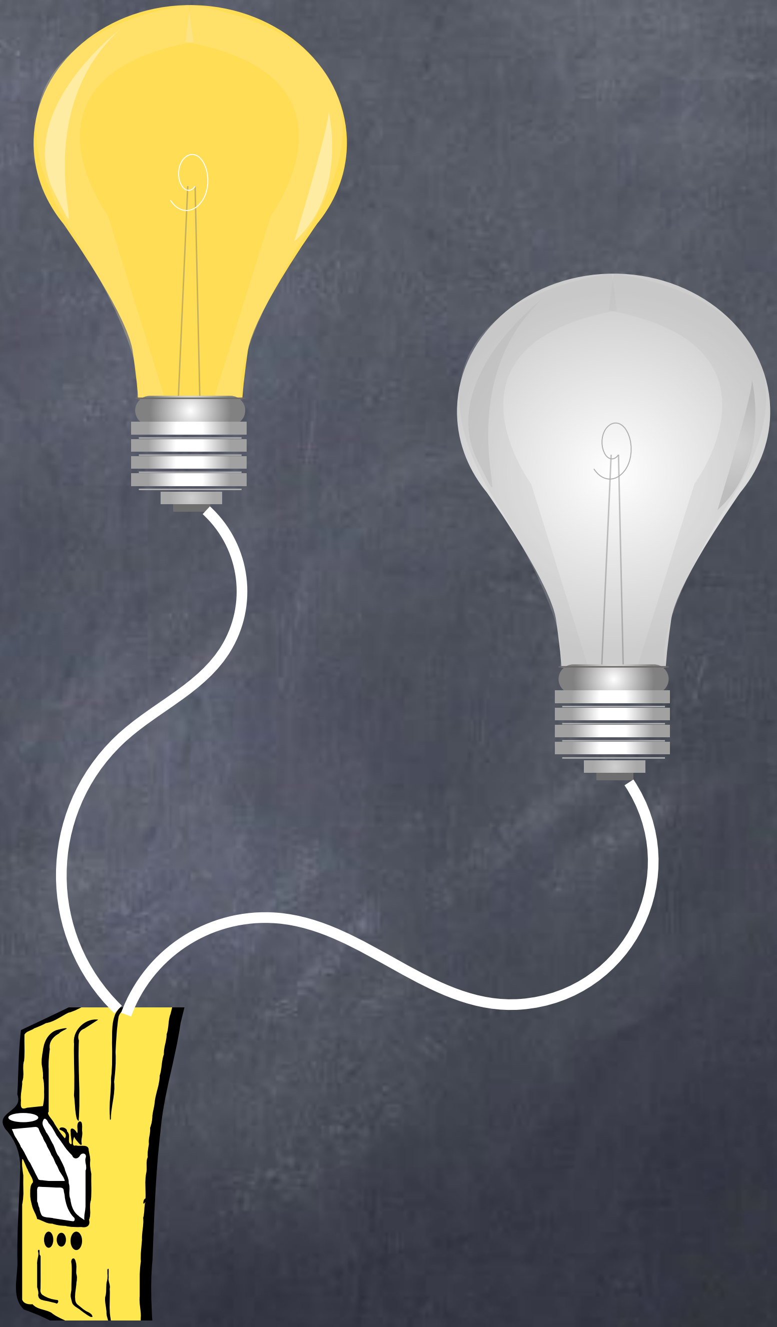

Power Supply
CPU & Cooling Fan
DIMM Memory Modules
CD/DVD Drive Bays
Card Reader
Enhanced PCI X1 Slot
PCI Expansion Slots
IDE Expansion Slot
Enhanced PCI X16 Slot
SATA Hard Drive
Internal Drive Bays

Abstraction

We “abstract” away the details whose effect on the solution to a problem is minimal or nonexistent, thereby creating a model that lets us deal with the essence of the problem.

– Aho & Ullman,
Foundations of Computer Science





openclipart.org: j4p4n, Arvin61r58, liftarn, afaulconbridge

Patterns

pattern |ˈpadərn|

noun

- an arrangement or sequence regularly found in comparable objects or events

Patterns

- Text patterns, for search & replace
- Image patterns (face recognition, text recognition, etc.)
- Patterns of activity
- Patterns of elements of a programming language a.k.a. programs

Patterns of Symbols

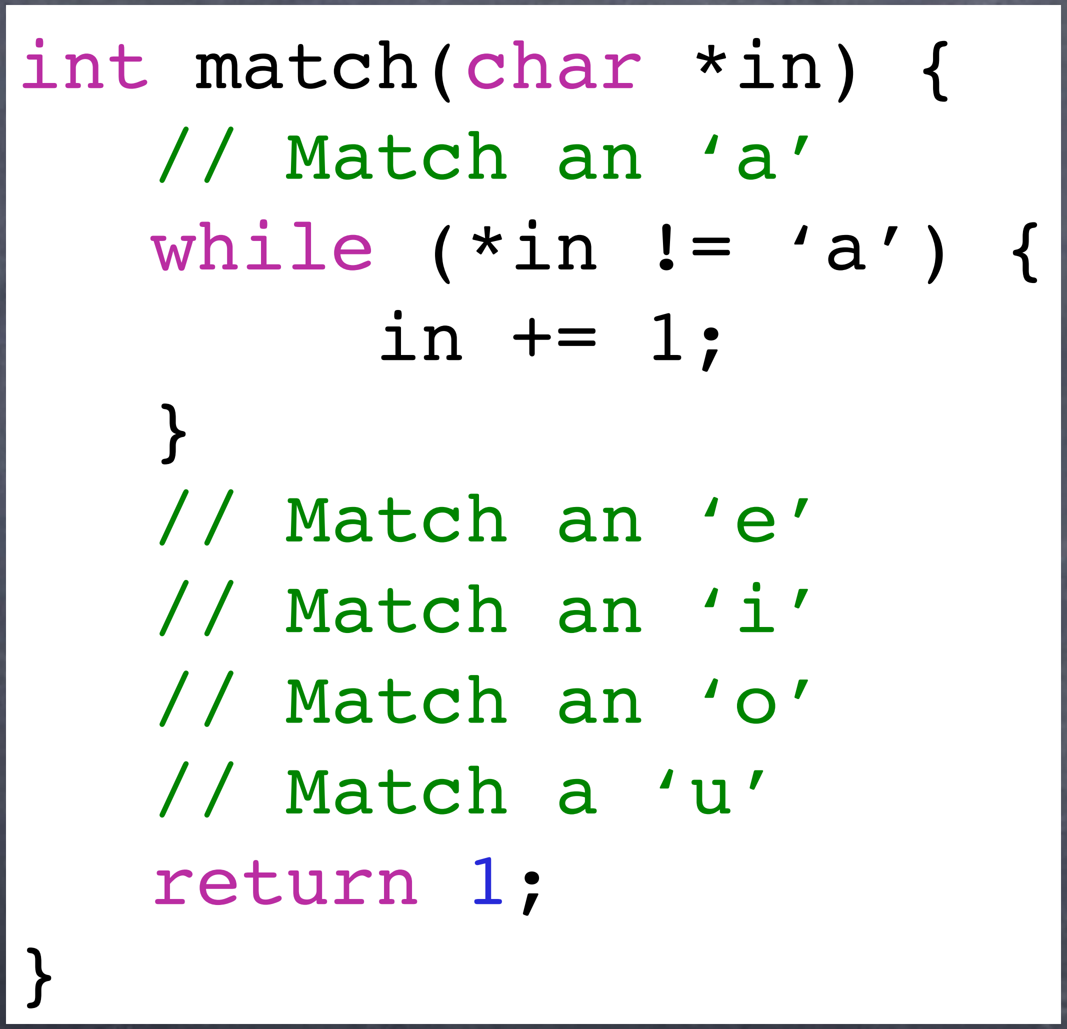
Patterns and Programs

Patterns and Programs

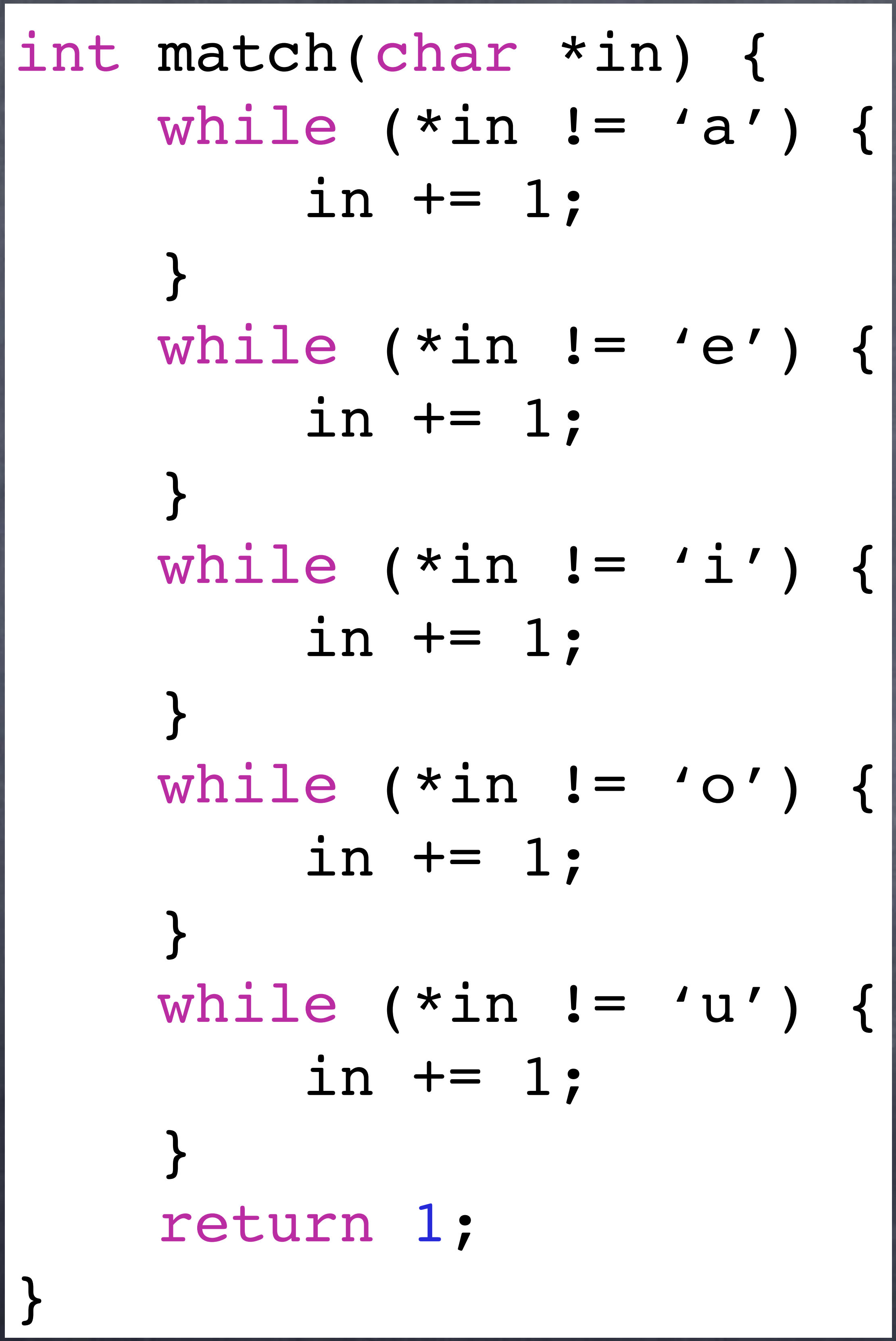
… a … e … i … o … u …

- Match an “a”
- Then match an “e”
- Then match an “i”
- Then match an “o”
- Then match a “u”
- Success!

```
int match(char *in) {
    // Match an ‘a’
    // Match an ‘e’
    // Match an ‘i’
    // Match an ‘o’
    // Match a ‘u’
    return 1;
}
```



```
int match(char *in) {
   // Match an ‘a’
   while (*in != ‘a’) {
        in += 1;
   }
   // Match an ‘e’
   // Match an ‘i’
   // Match an ‘o’
   // Match a ‘u’
   return 1;
}
```

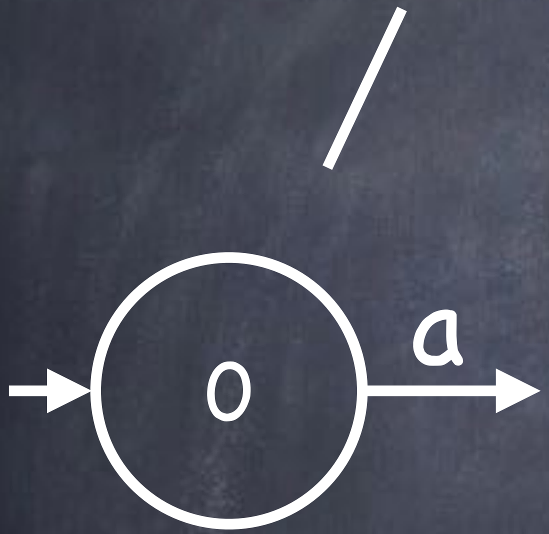


```
int match(char *in) {
    while (*in != ‘a’) {
        in += 1;
    }
    while (*in != ‘e’) {
        in += 1;
    }
    while (*in != ‘i’) {
        in += 1;
    }
    while (*in != ‘o’) {
        in += 1;
    }
    while (*in != ‘u’) {
        in += 1;
    }
    return 1;
}
```

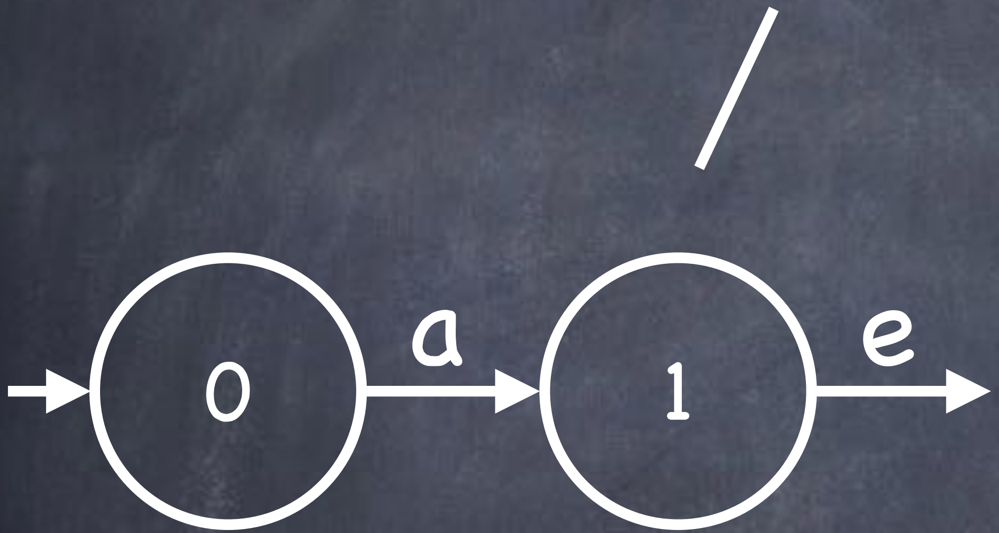
Finite State System

- Can be in any one of a finite number of internal configurations or states
- The state of the system summarizes any information needed to determine the subsequent behavior of the system

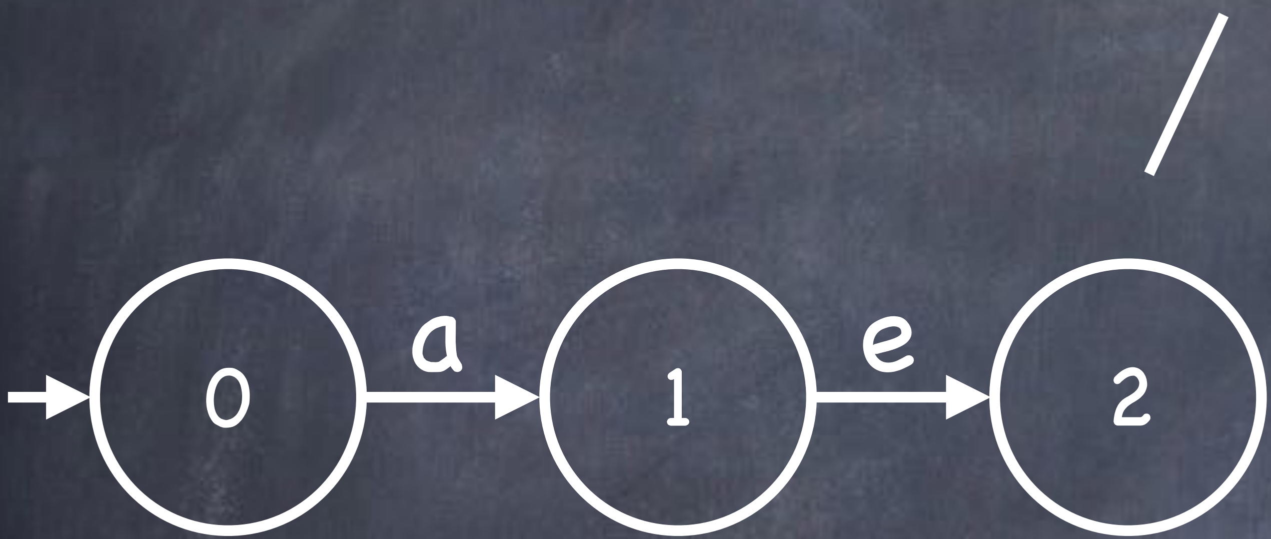
I haven’t seen an “a”



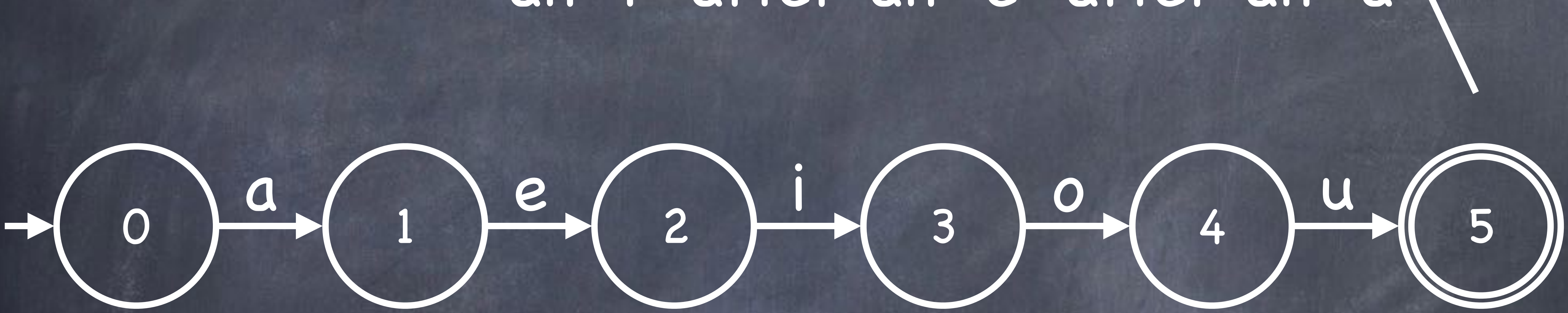
I've seen an "a"

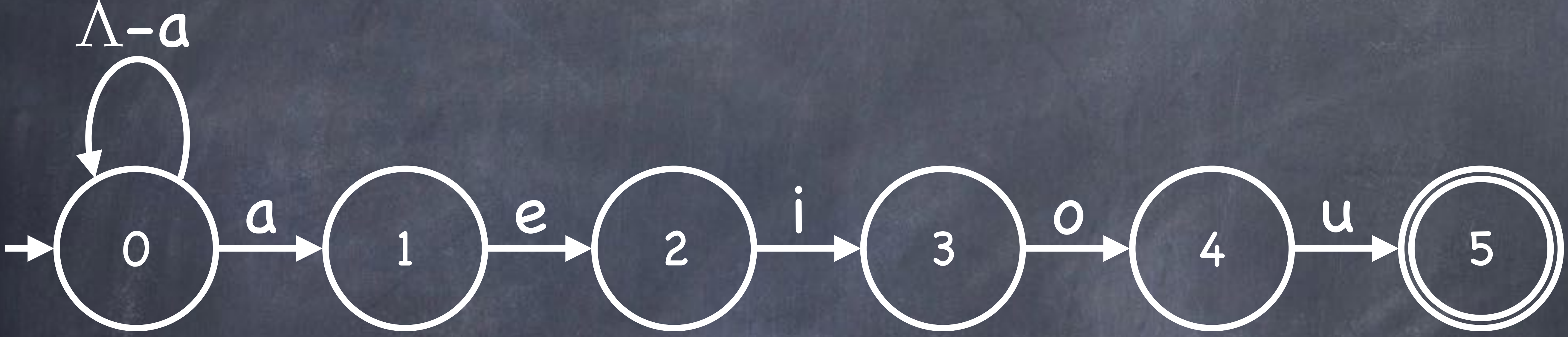


I’ve seen an “e” after an “a”

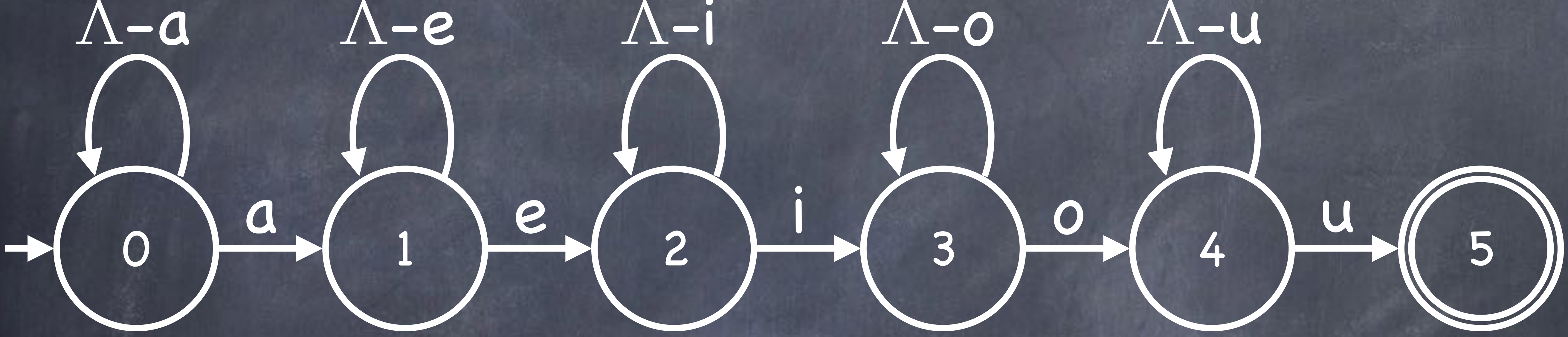


I’ve seen a “u” after an “o” after an “i” after an “e” after an “a”





Λ: Set of all upper- and lowercase characters


Λ–a
Λ–e
Λ–i
Λ–o
Λ–u
0
a
1
e
2
i
3
o
4
u
5

Automaton

- a.k.a. Finite State Automaton (FSA), Finite Automaton (FA)

Automaton

automaton |ôˈtämədənôˈtäməˌtän|

noun (pl. **automata** |-tə| or **automatons**)

- a machine that performs a function according to a predetermined set of coded instructions, especially one capable of a range of programmed responses to different circumstances.

ORIGIN

early 17th cent.: via Latin from Greek, neuter of *automatos* ‘**acting of itself**,’ from autos ‘**self**.’

Automaton

- a.k.a. Finite State Automaton (FSA), Finite Automaton (FA)
- Starts in initial state
- Reads input sequence one symbol (character) at a time
- Transitions to new (or same) state based on input symbol

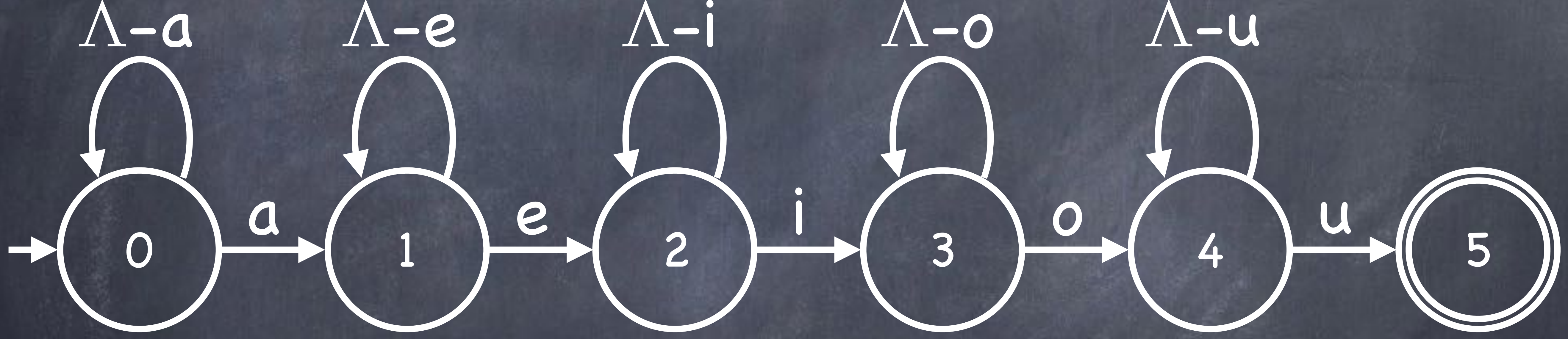
Automaton

- If in “success” (accepting) state:
    - Then accepts input read so far
- At end of input:
    - Accepts entire input if in accepting state, else rejects input

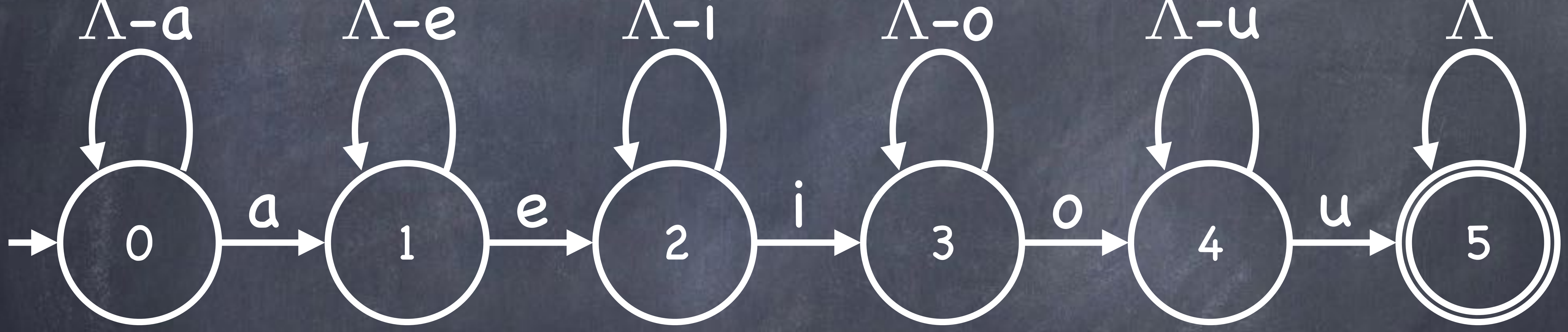
Automaton

- If in “success” (accepting) state:
  - Then accepts input read so far
- At end of input:
  - Accepts entire input if in accepting state, else rejects input
- If no transition:
  - Halt and reject

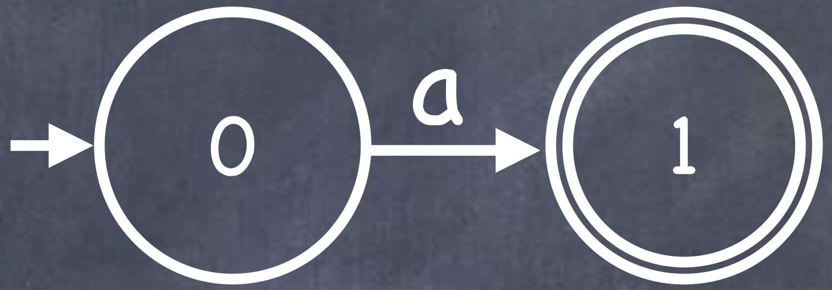
Automata and Their Programs



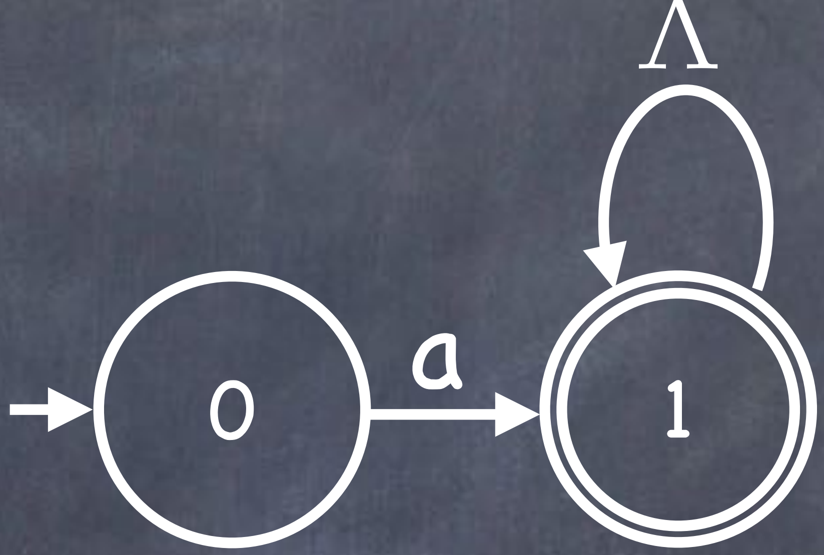
Automata and Their Programs



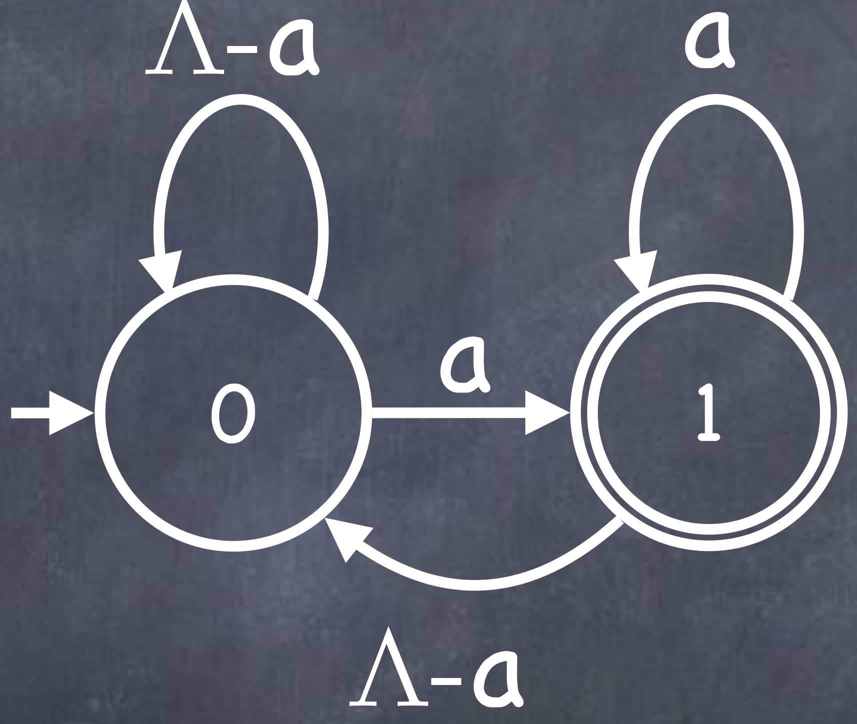
Exactly “a”



Starts with “a”



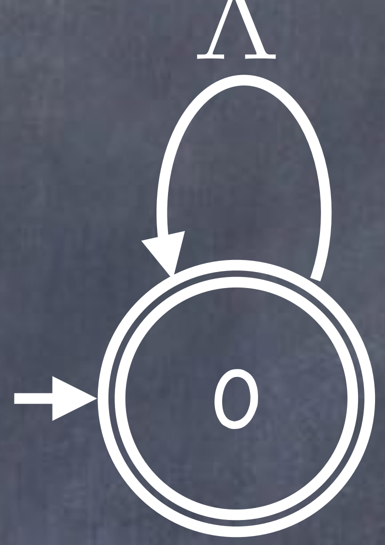
Ends with “a”



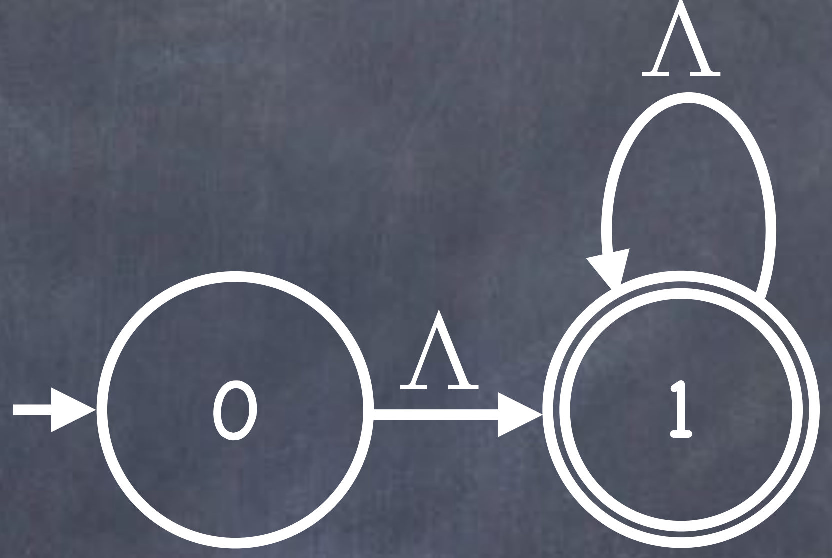
The Empty String ε

```
String s = "";

char *s = "";
```

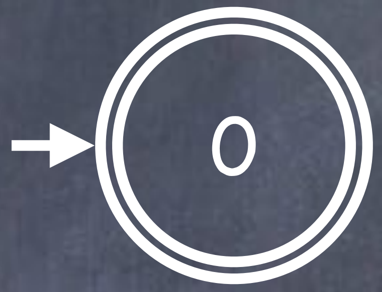
Any string (including ε)

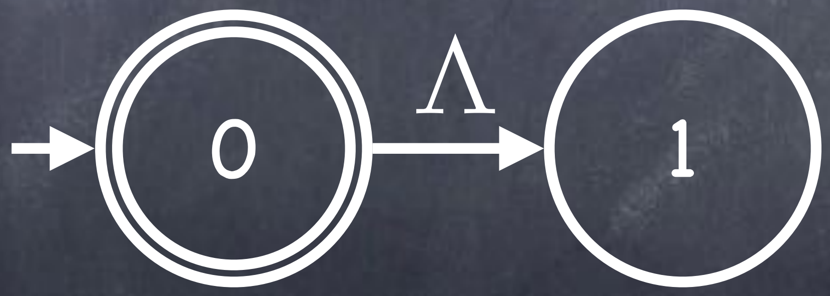


Any non-empty string

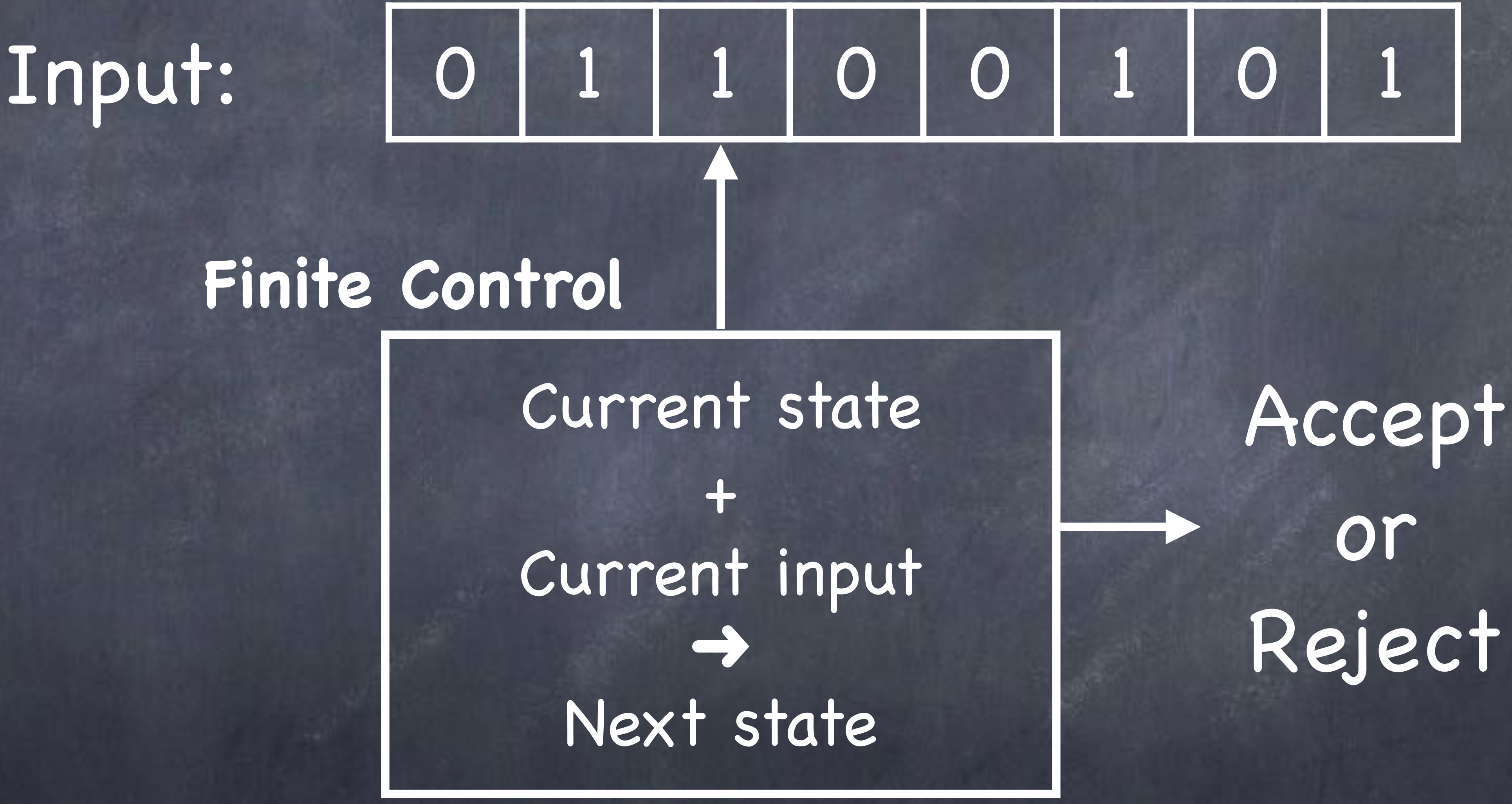


Empty string





Finite Automaton



Input:

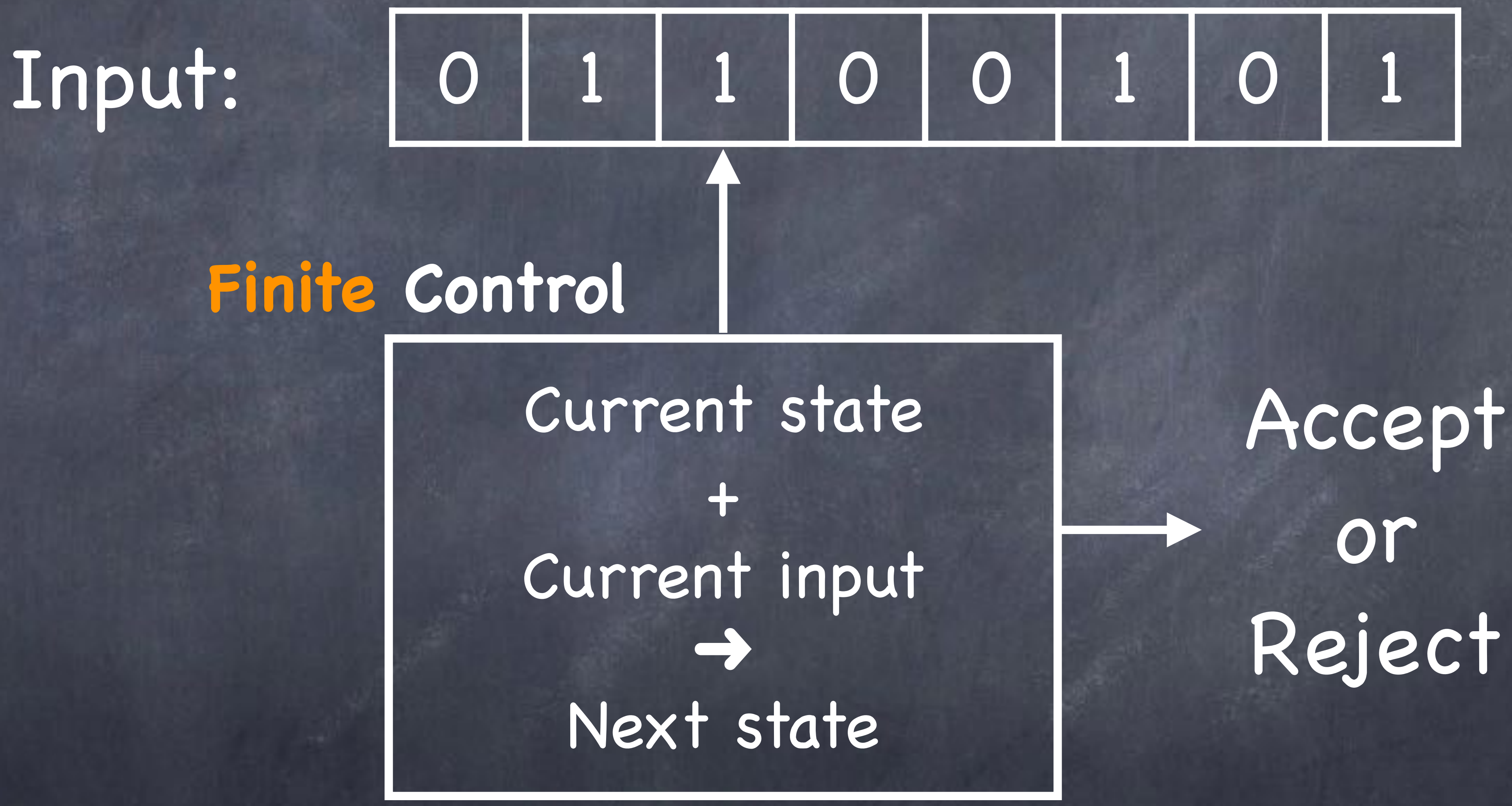
0	1	1	0	0	1	0	1

Finite Control

Current state
+
Current input
➜
Next state

Accept
or
Reject

Finite Automaton



Input:

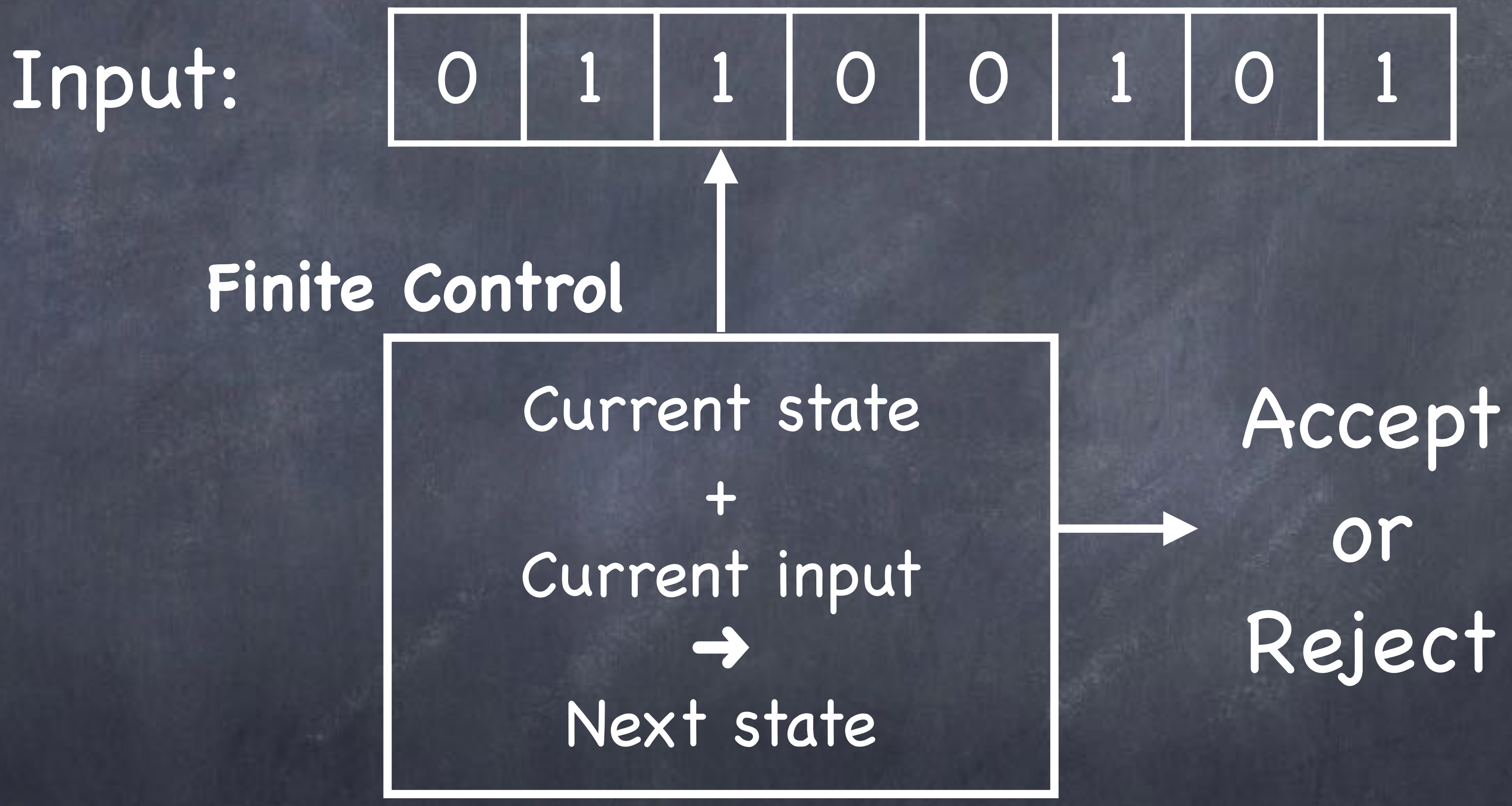
0	1	1	0	0	1	0	1

Finite Control

Current state
+
Current input
→
Next state

Accept
or
Reject

Finite Automaton



Input:

0	1	1	0	0	1	0	1

Finite Control

Current state
+
Current input
→
Next state

Accept
or
Reject

FA Programming?

Finite Control

Given current state and input symbol:

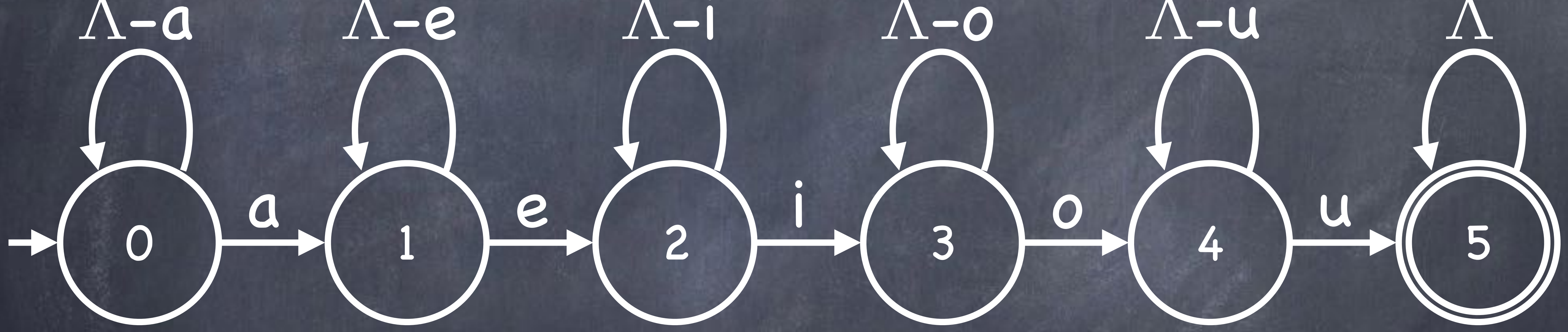
Return next state

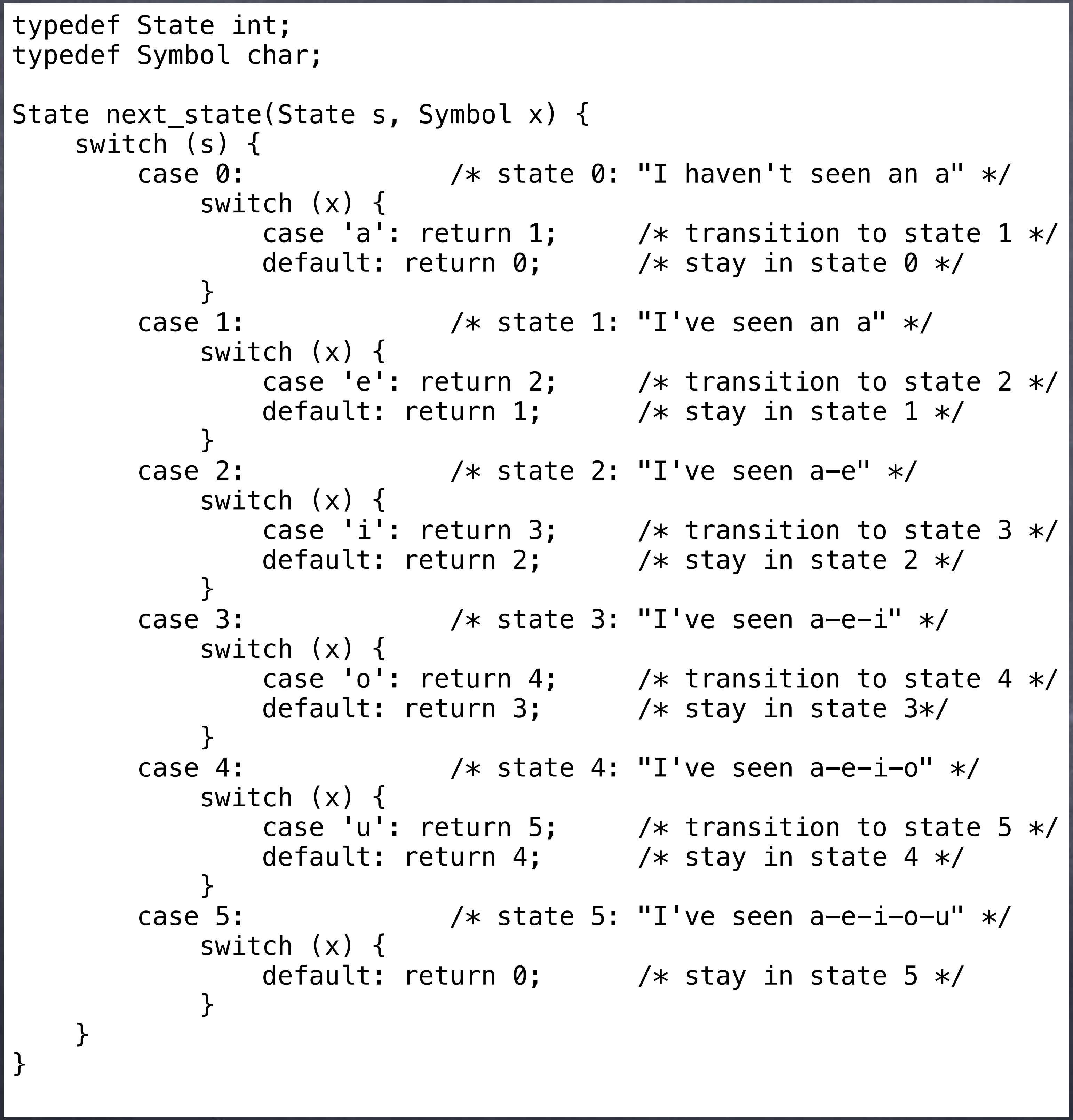
Finite Control

Given current state and input symbol:

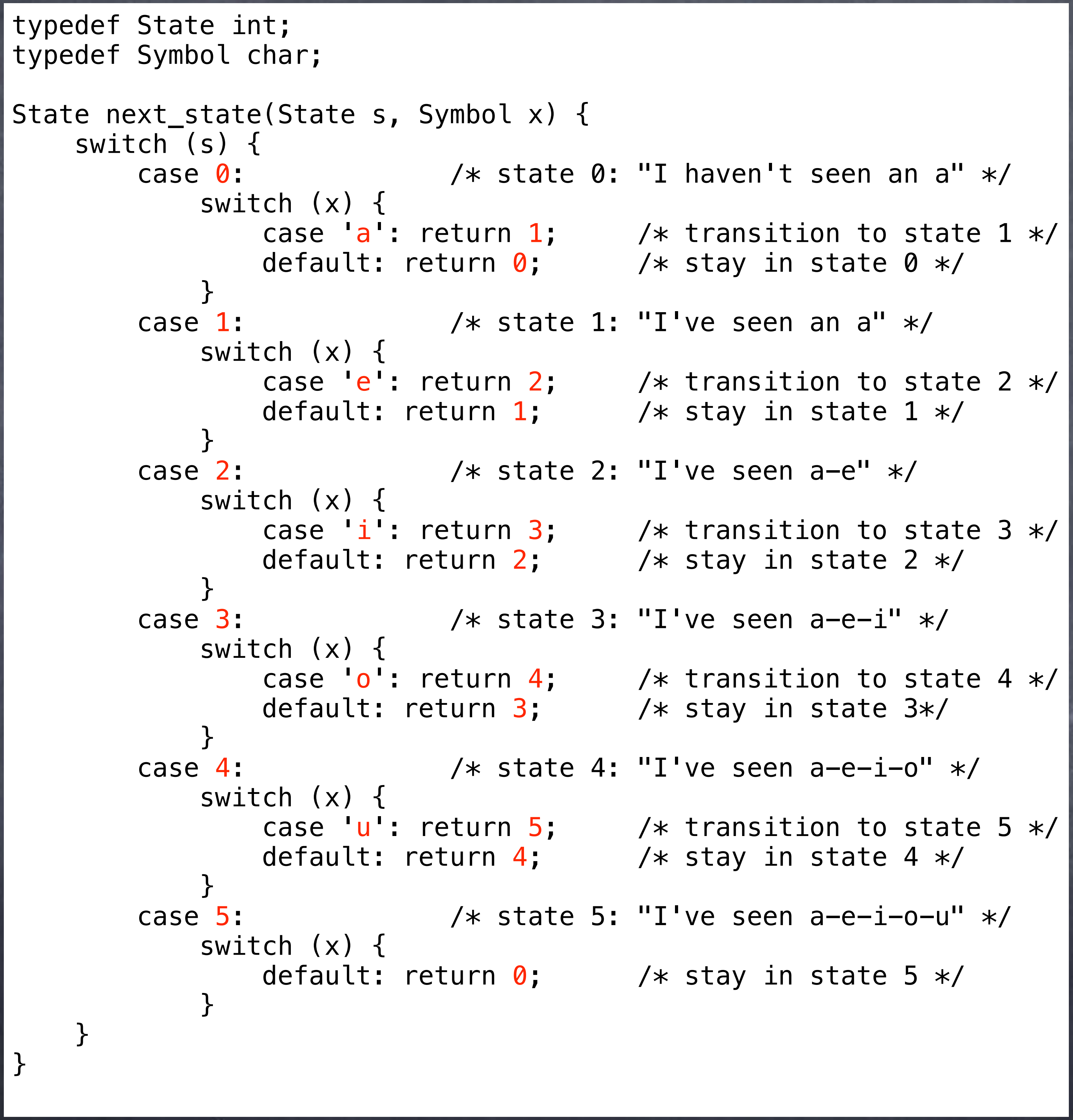
Return next state

```
State next_state(State s, Symbol x)
```

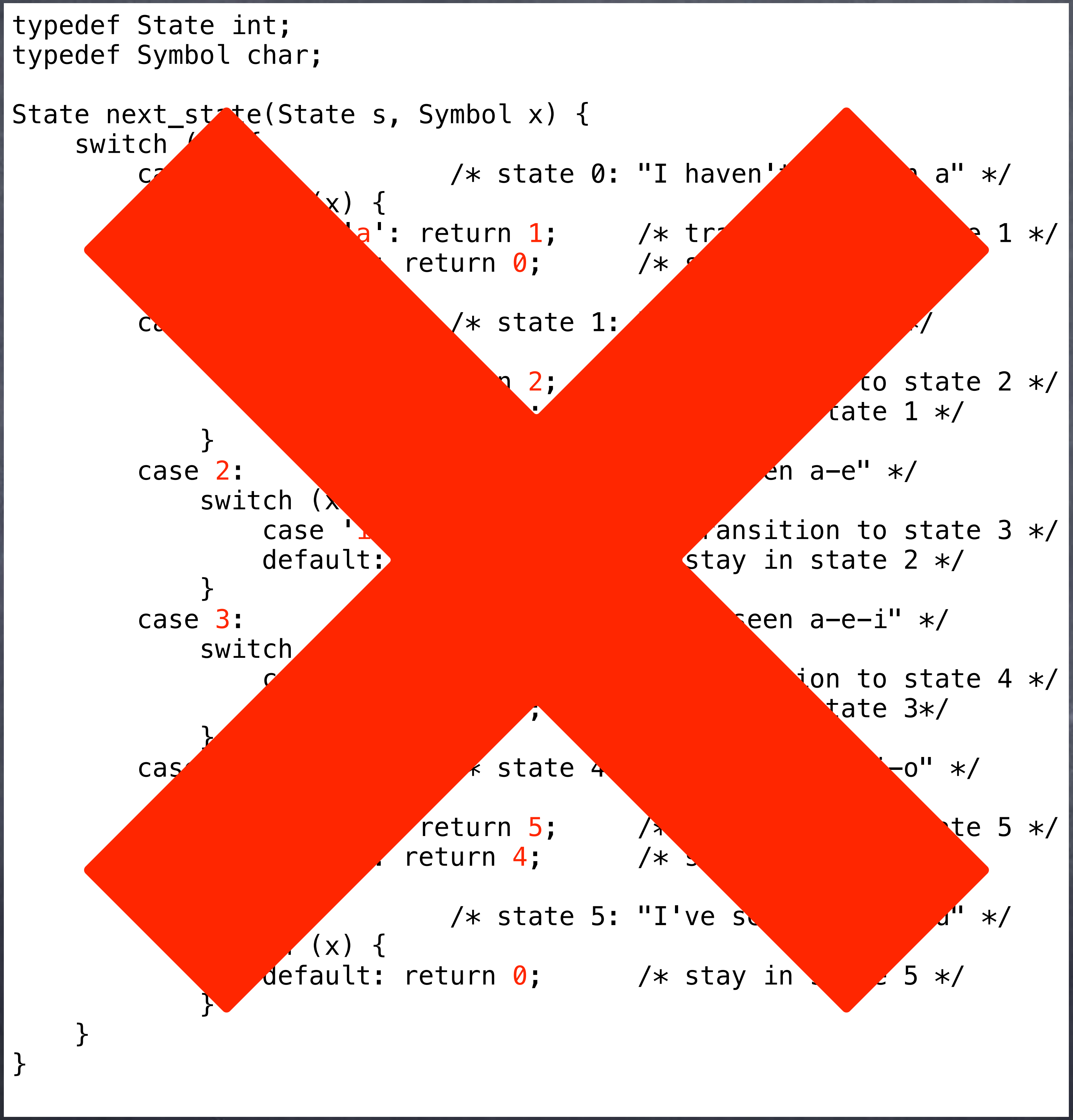

Λ–a
Λ–e
Λ–i
Λ–o
Λ–u
Λ
0
a
1
e
2
i
3
o
4
u
5



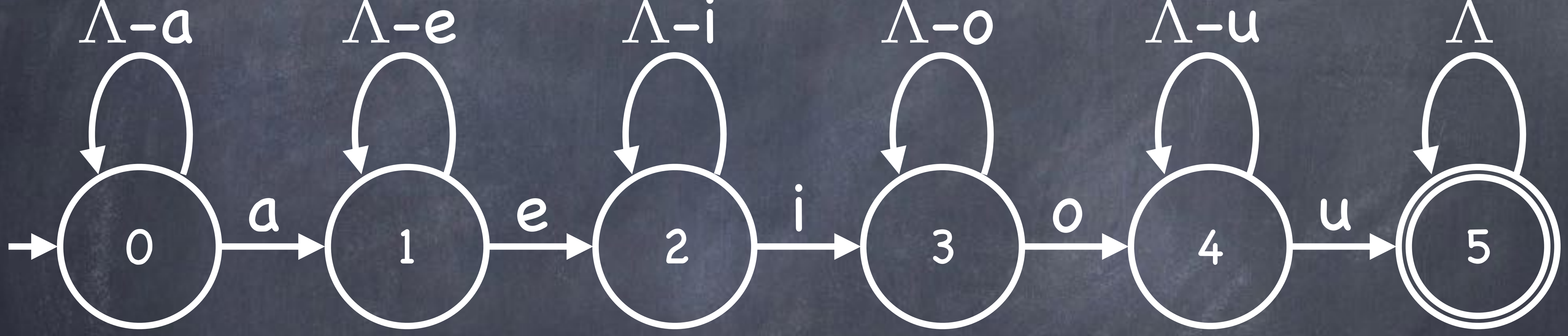
```
typedef State int;
typedef Symbol char;

State next_state(State s, Symbol x) {
    switch (s) {
        case 0:             /* state 0: "I haven't seen an a" */
            switch (x) {
                case 'a': return 1;     /* transition to state 1 */
                default: return 0;      /* stay in state 0 */
            }
        case 1:             /* state 1: "I've seen an a" */
            switch (x) {
                case 'e': return 2;     /* transition to state 2 */
                default: return 1;      /* stay in state 1 */
            }
        case 2:             /* state 2: "I've seen a-e" */
            switch (x) {
                case 'i': return 3;     /* transition to state 3 */
                default: return 2;      /* stay in state 2 */
            }
        case 3:             /* state 3: "I've seen a-e-i" */
            switch (x) {
                case 'o': return 4;     /* transition to state 4 */
                default: return 3;      /* stay in state 3*/
            }
        case 4:             /* state 4: "I've seen a-e-i-o" */
            switch (x) {
                case 'u': return 5;     /* transition to state 5 */
                default: return 4;      /* stay in state 4 */
            }
        case 5:             /* state 5: "I've seen a-e-i-o-u" */
            switch (x) {
                default: return 0;      /* stay in state 5 */
            }
    }
}
```



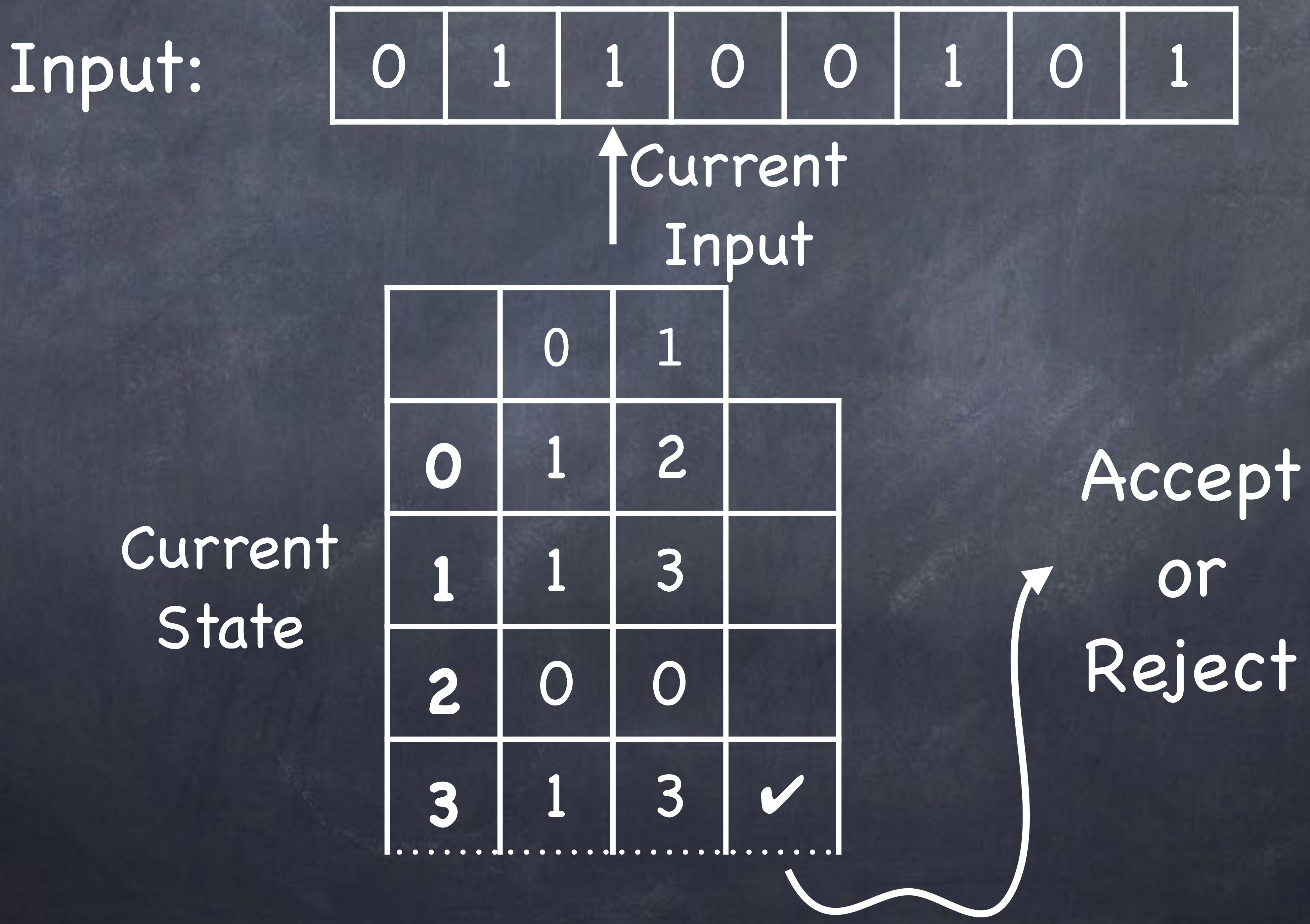
```
typedef State int;
typedef Symbol char;

State next_state(State s, Symbol x) {
    switch (s) {
        case 0:             /* state 0: "I haven't seen an a" */
            switch (x) {
                case 'a': return 1;     /* transition to state 1 */
                default: return 0;      /* stay in state 0 */
            }
        case 1:             /* state 1: "I've seen an a" */
            switch (x) {
                case 'e': return 2;     /* transition to state 2 */
                default: return 1;      /* stay in state 1 */
            }
        case 2:             /* state 2: "I've seen a-e" */
            switch (x) {
                case 'i': return 3;     /* transition to state 3 */
                default: return 2;      /* stay in state 2 */
            }
        case 3:             /* state 3: "I've seen a-e-i" */
            switch (x) {
                case 'o': return 4;     /* transition to state 4 */
                default: return 3;      /* stay in state 3*/
            }
        case 4:             /* state 4: "I've seen a-e-i-o" */
            switch (x) {
                case 'u': return 5;     /* transition to state 5 */
                default: return 4;      /* stay in state 4 */
            }
        case 5:             /* state 5: "I've seen a-e-i-o-u" */
            switch (x) {
                default: return 0;      /* stay in state 5 */
            }
    }
}
```



```
typedef State int;
typedef Symbol char;

State next_st te(State s, Symbol x) {
    switch (
        ca                  /* state 0: "I haven'             a" */
                    x) {
                     a': return 1;     /* tr                  e 1 */
                         return 0;     /* s

        ca                  /* state 1:                    */

                               n 2;                  to state 2 */
                                                    tate 1 */
            }
        case 2:                                     en a-e" */
            switch (x
                case '                           ansition to state 3 */
                default:                       stay in state 2 */
            }
        case 3:                                     een a-e-i" */
            switch
                c                                     ion to state 4 */
                                 ;                  tate 3*/
            }
        cas                  * state 4                      -o" */
                          return 5;     /*                    te 5 */
                         return 4;     /* s

                            /* state 5: "I've s              u" */
                   (x) {
                default: return 0;     /* stay in        e 5 */
            }
    }
}
```


Λ-a
Λ-e
Λ-i
Λ-o
Λ-u
Λ
0
a
1
e
2
i
3
o
4
u
5

Transition Table



Input:

0	1	1	0	0	1	0	1

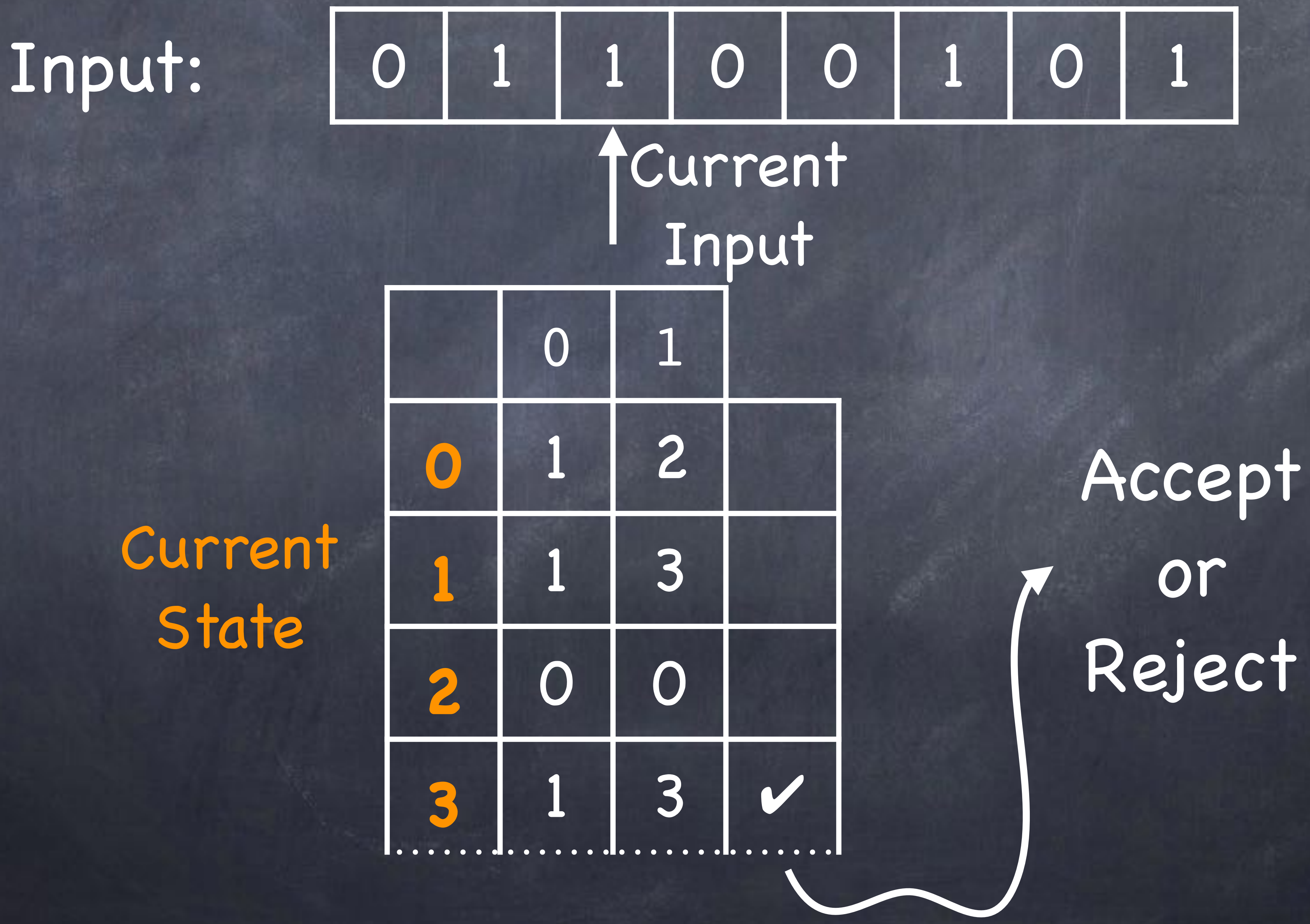
Current Input

Current State

	0	1	
0	1	2	
1	1	3	
2	0	0	
3	1	3	✔

Accept or Reject

Transition Table



Input:							
0	1	1	0	0	1	0	1

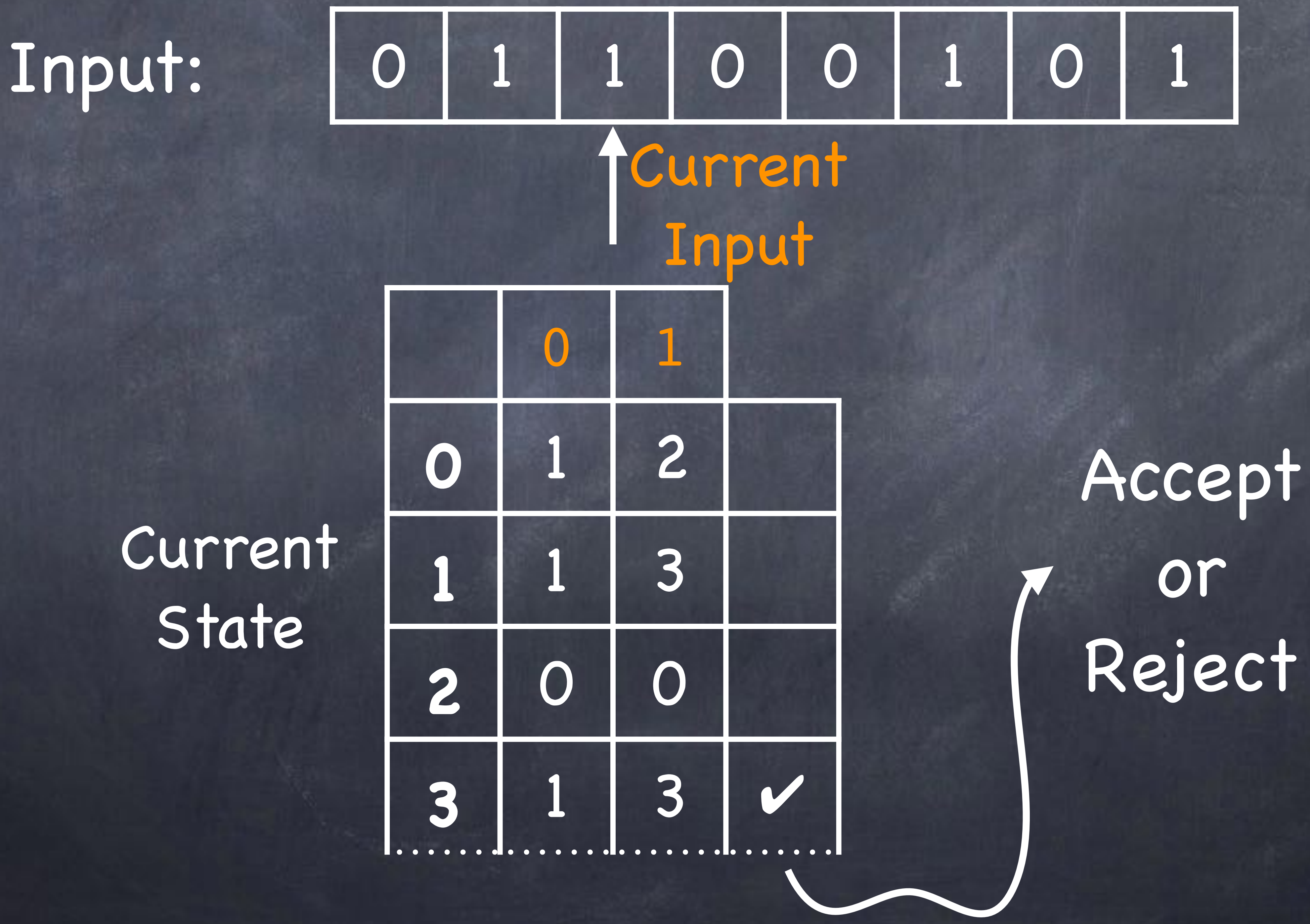
Current Input

Current State

	0	1	
0	1	2	
1	1	3	
2	0	0	
3	1	3	✓

Accept or Reject

Transition Table



Input:

0	1	1	0	0	1	0	1

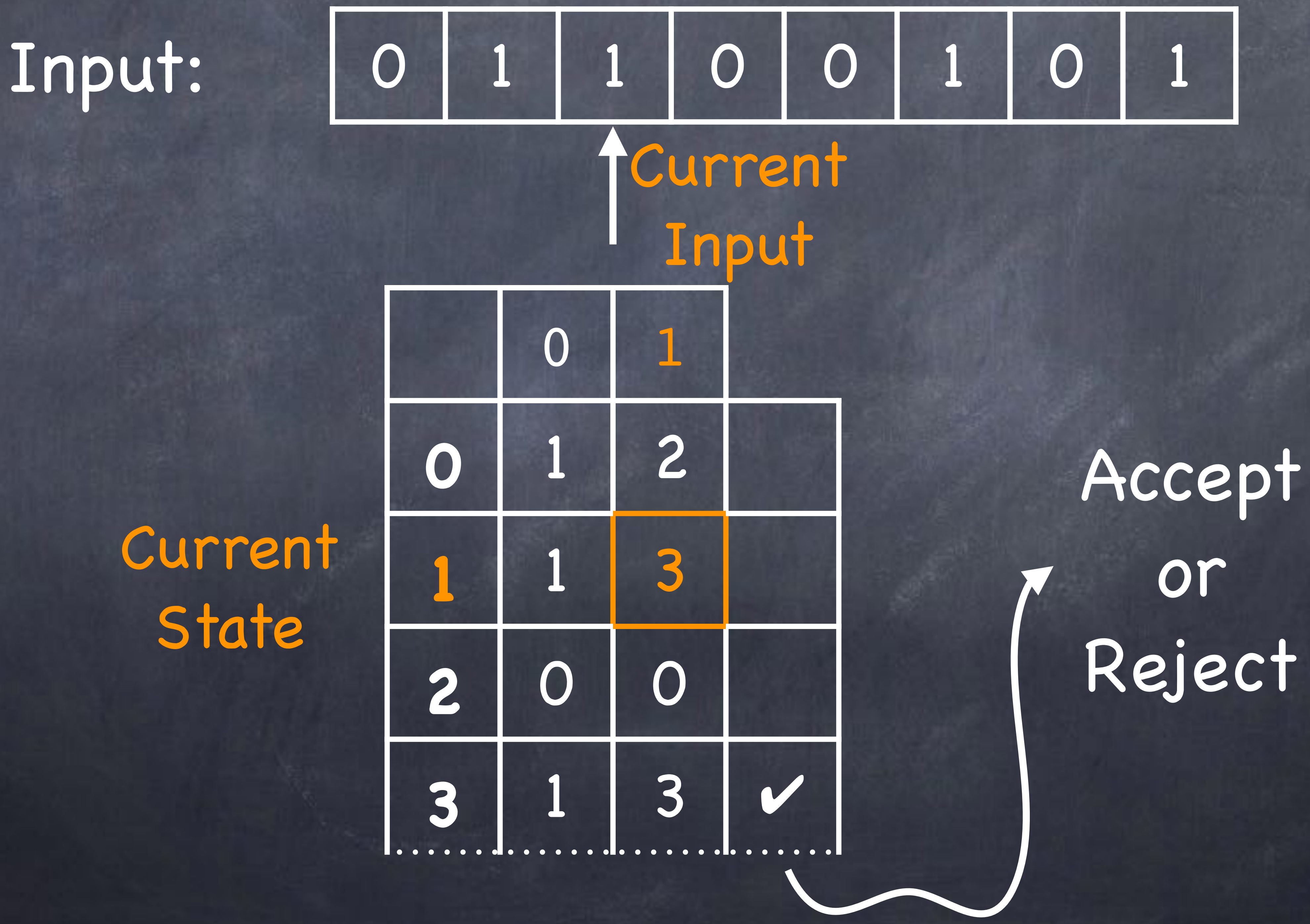
Current Input

Current State

	0	1	
0	1	2	
1	1	3	
2	0	0	
3	1	3	✓

Accept or Reject

Transition Table



Input:

0	1	1	0	0	1	0	1

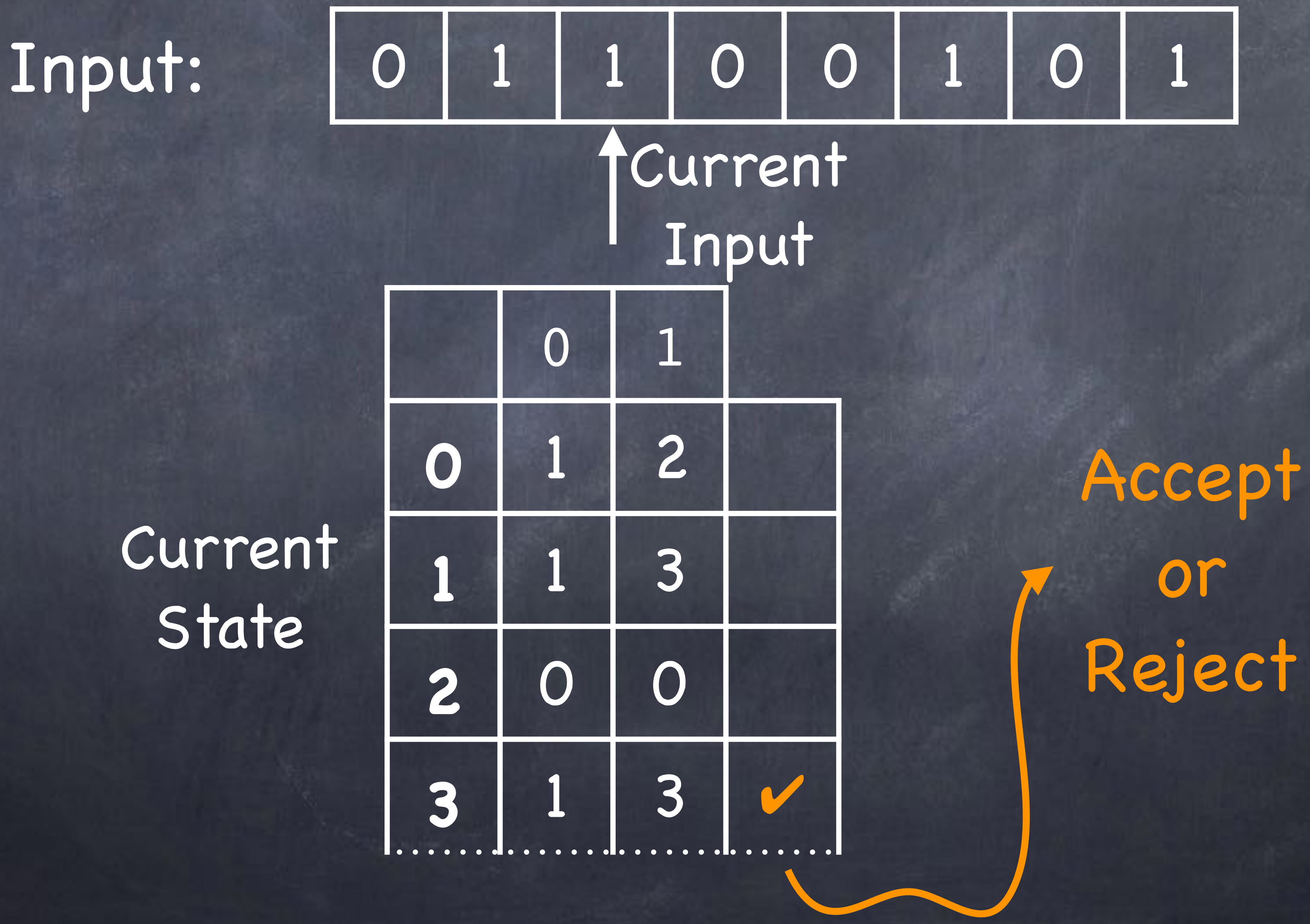
Current Input

Current State

	0	1	
0	1	2	
1	1	3	
2	0	0	
3	1	3	✓

Accept or Reject

Transition Table



Input:

0	1	1	0	0	1	0	1

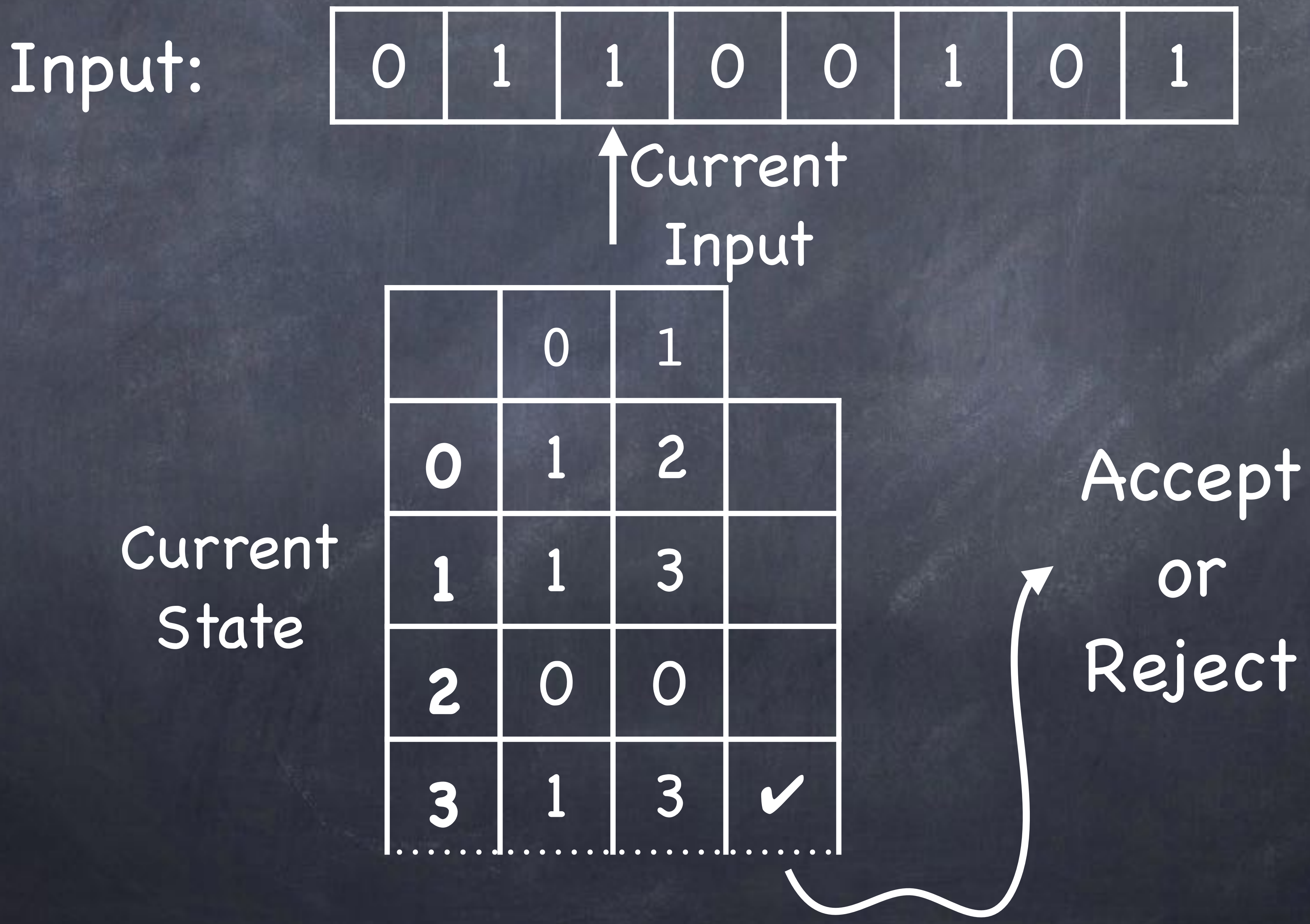
Current Input

Current State

	0	1	
0	1	2	
1	1	3	
2	0	0	
3	1	3	✓

Accept or Reject

Transition Table



Input:

0	1	1	0	0	1	0	1

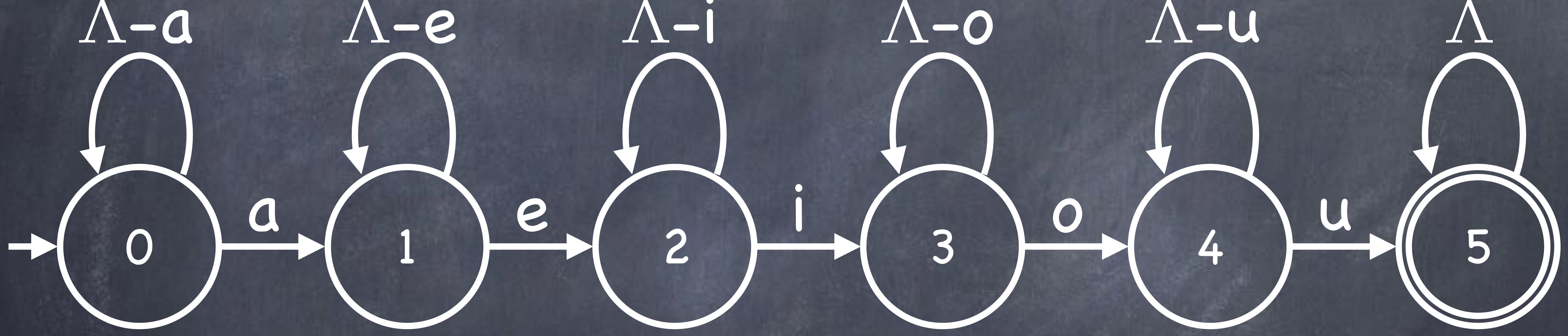
Current Input

Current State

	0	1	
0	1	2	
1	1	3	
2	0	0	
3	1	3	✓

Accept or Reject

Finite Automaton



Finite Automaton

$$M = \langle S,\ \Sigma,\ \delta,\ s_0,\ F \rangle$$

- S: finite set of states
- Σ: finite input alphabet
- δ: transition function $S \times \Sigma \rightarrow S$
- $s_0 \in S$: initial state
- $F \subseteq S$: set of accepting (final) states

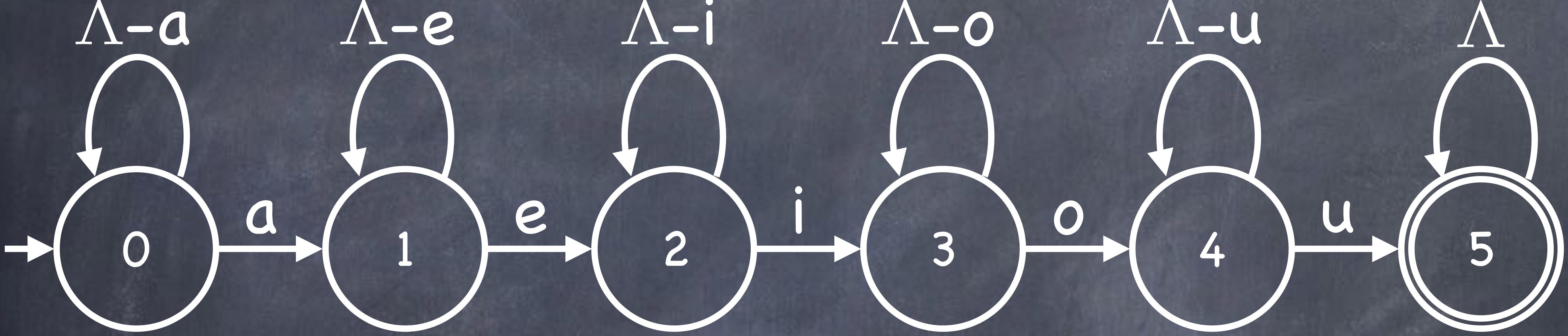
Language of an Automaton

- The language accepted by an automaton M, $L(M)$, is the set of input strings that it accepts
    - That is, the set of strings for which it ends up in an accepting state after processing the entire string

Paths and Labels

- Automaton M on input $a_1\ a_2\ \ldots\ a_k$
- Path (state sequence) $s_0\ s_1\ s_2\ \ldots\ s_k$

Paths and Labels



Input: adept

Input:		a		d		e		p		t	
State:	0		1		1		2		2		2

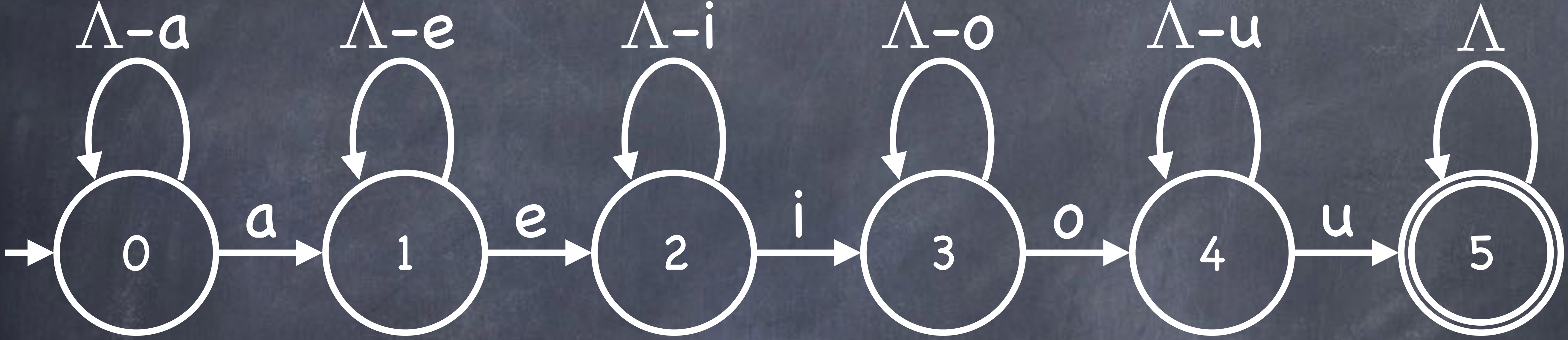
Paths and Labels

- Automaton M on input a_1 a_2 ... a_k
- Path (state sequence) s_0 s_1 s_2 ... s_k
- a_1 a_2 ... a_k is a label of the path

Paths and Labels

- Automaton M on input $a_1\ a_2\ ...\ a_k$
- Path (state sequence) $s_0\ s_1\ s_2\ ...\ s_k$
- $a_1\ a_2\ ...\ a_k$ is a label of the path
  - It may have other labels

Paths and Labels



Input: ahead

Input:		a		h		e		a		d	
State:	0		1		1		2		2		2

Languages

- Language accepted by M is the set of labels of paths to accepting states
  - The set of inputs that it accepts

Transition Function

δ: transition function $S \times \Sigma \to S$

Transition Function

δ: transition function $S \times \Sigma \rightarrow S$

$\delta'(s, \varepsilon) = s$

$\delta'(s, wa) = \delta(\delta'(s, w), a)$

Transition Function

δ: transition function $S \times \Sigma \rightarrow S$

$$\delta'(s, \varepsilon) = s$$

$$\delta'(s, wa) = \delta(\delta'(s, w), a)$$

Transition Function

δ: transition function $S \times \Sigma \rightarrow S$

$\delta'(s, \varepsilon) = s$

$\delta'(s, wa) = \delta(\delta'(s, w), a)$

Languages

- Language accepted by M is the set of labels of paths to accepting states
  - The set of inputs that it accepts
- $M = \langle S, \Sigma, \delta, s_0, F \rangle$

  $\delta'(s, \varepsilon) = s$

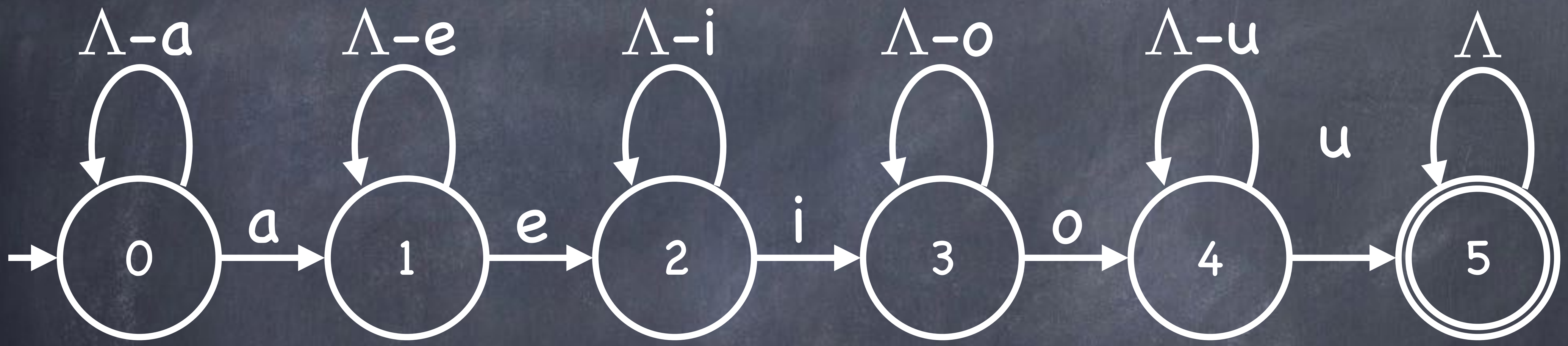
  $\delta'(s, wa) = \delta(\delta'(s, w), a)$

  $L(M) = \{ x \mid \delta'(s_0, x) \in F \}$

Finite Automata (So Far)

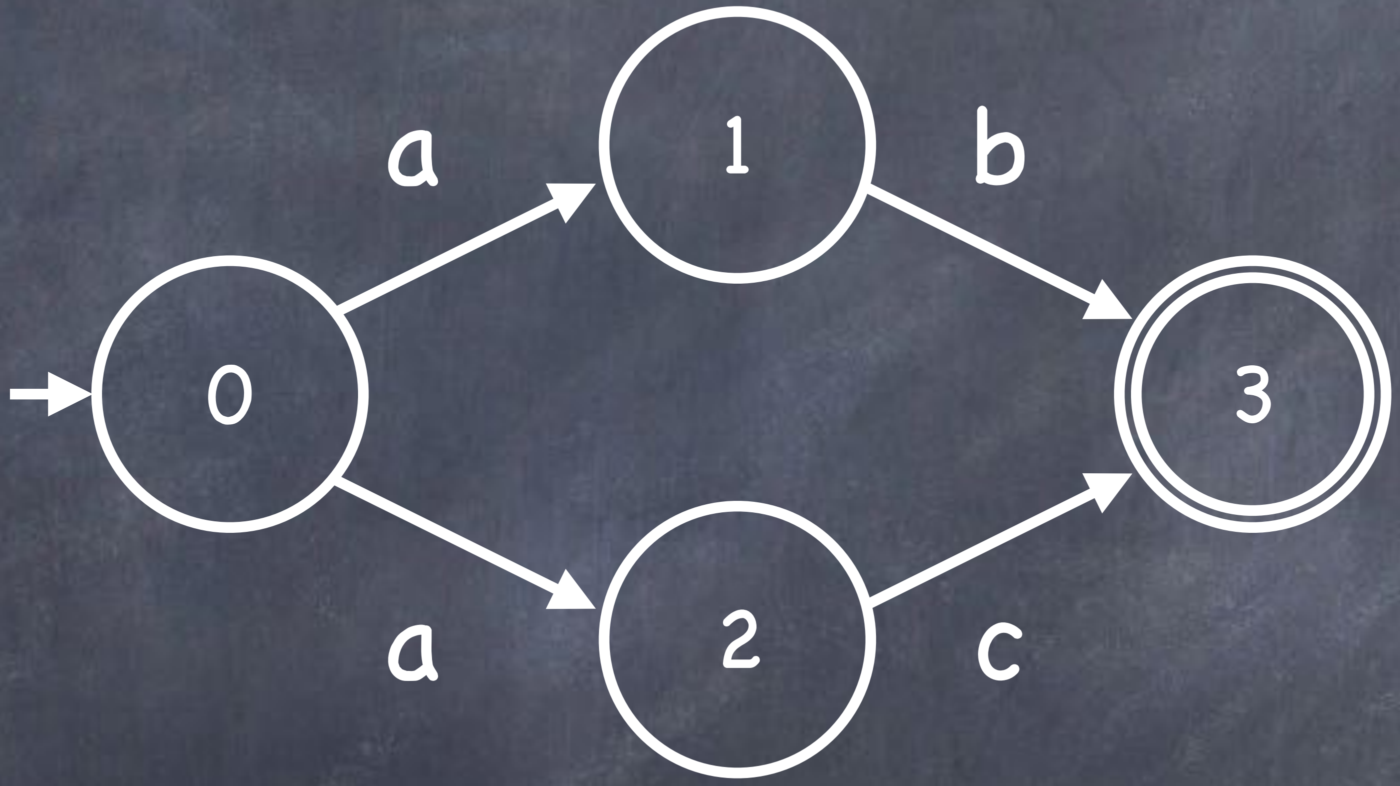
- Finite automata are a graph-based way of specifying patterns
- Also a model of a simple form of computation (computing device)
- Also describe languages: the set of inputs on which the automaton accepts

Deterministic Automata (DFA)

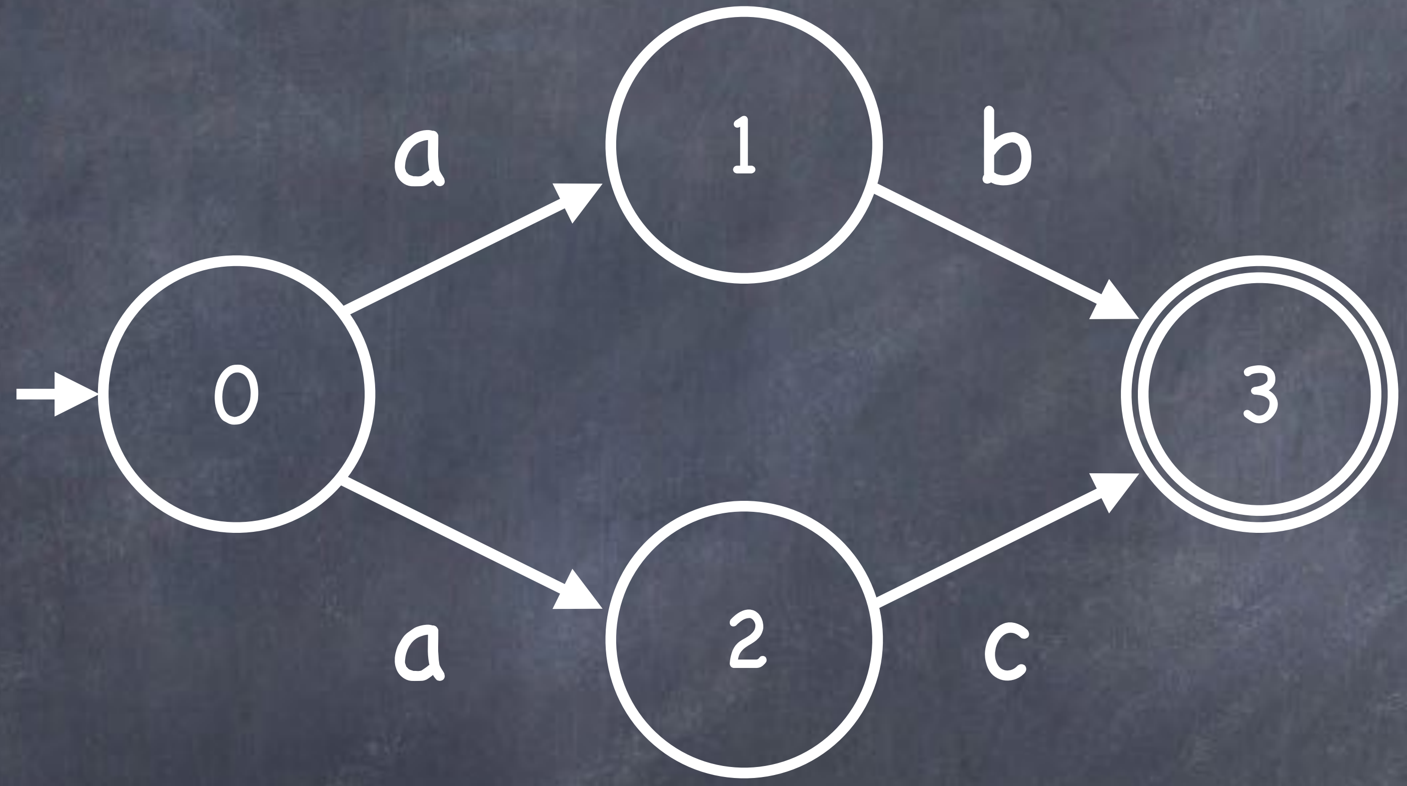


For each state s and input symbol x:

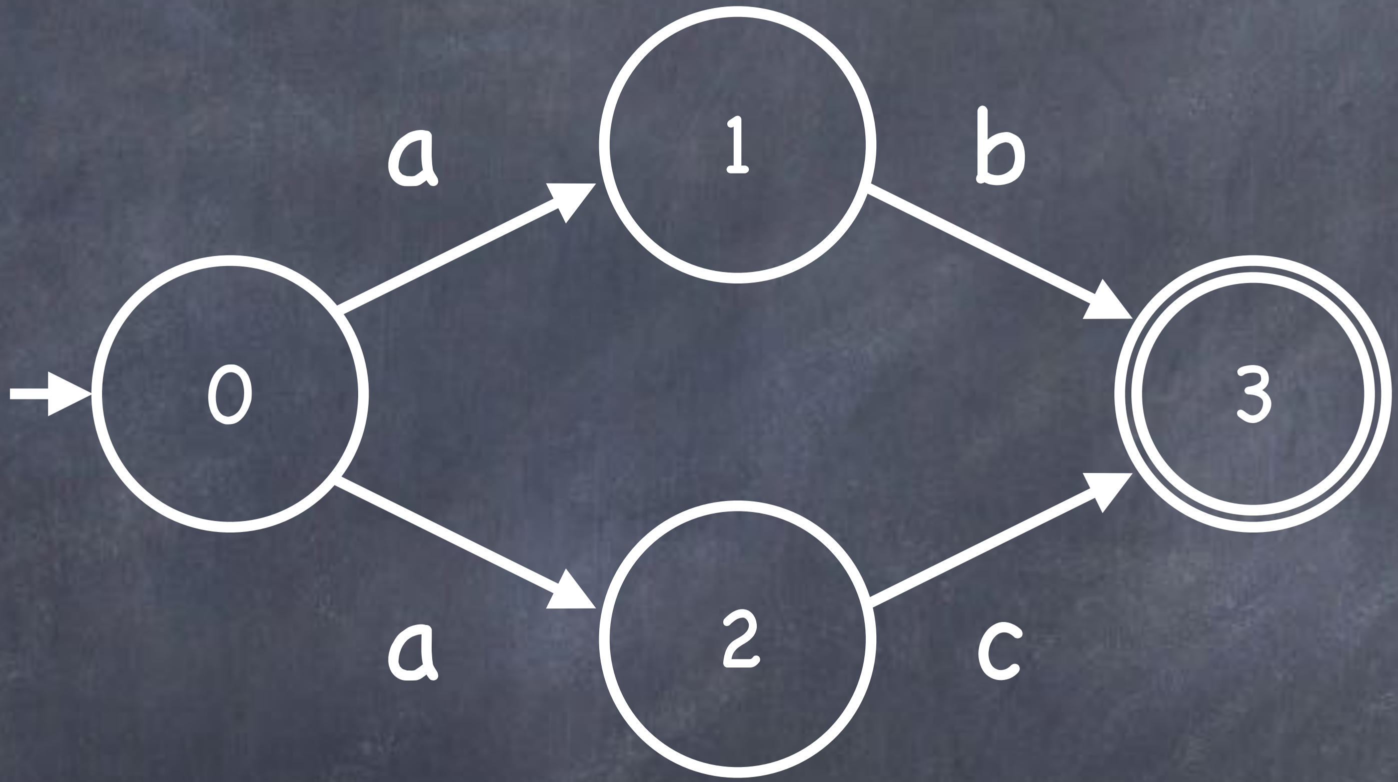
≤ 1 transition from s on x

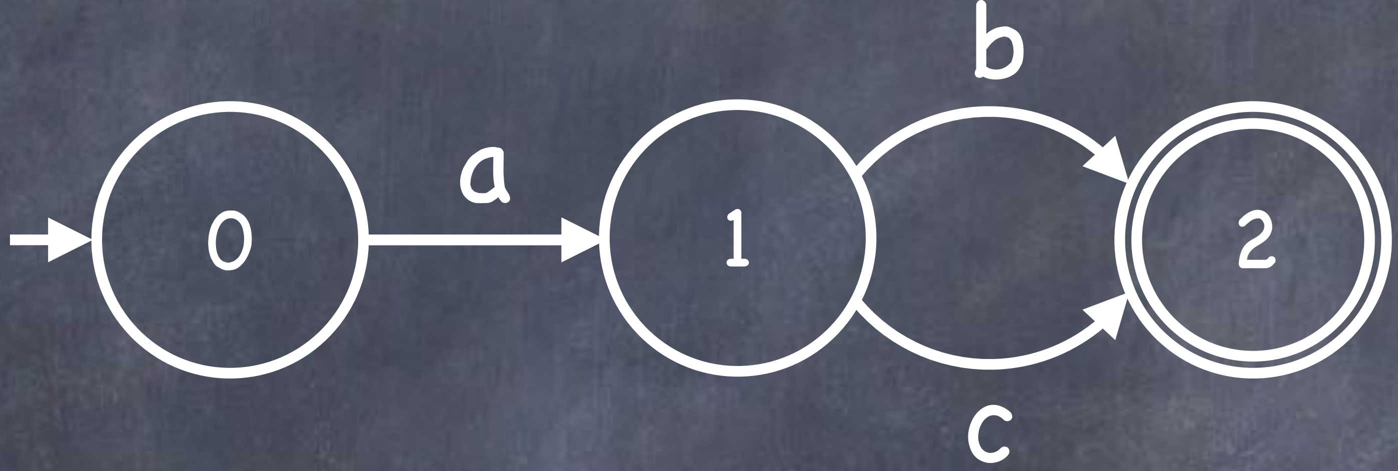

a
1
b
0
3
a
2
c

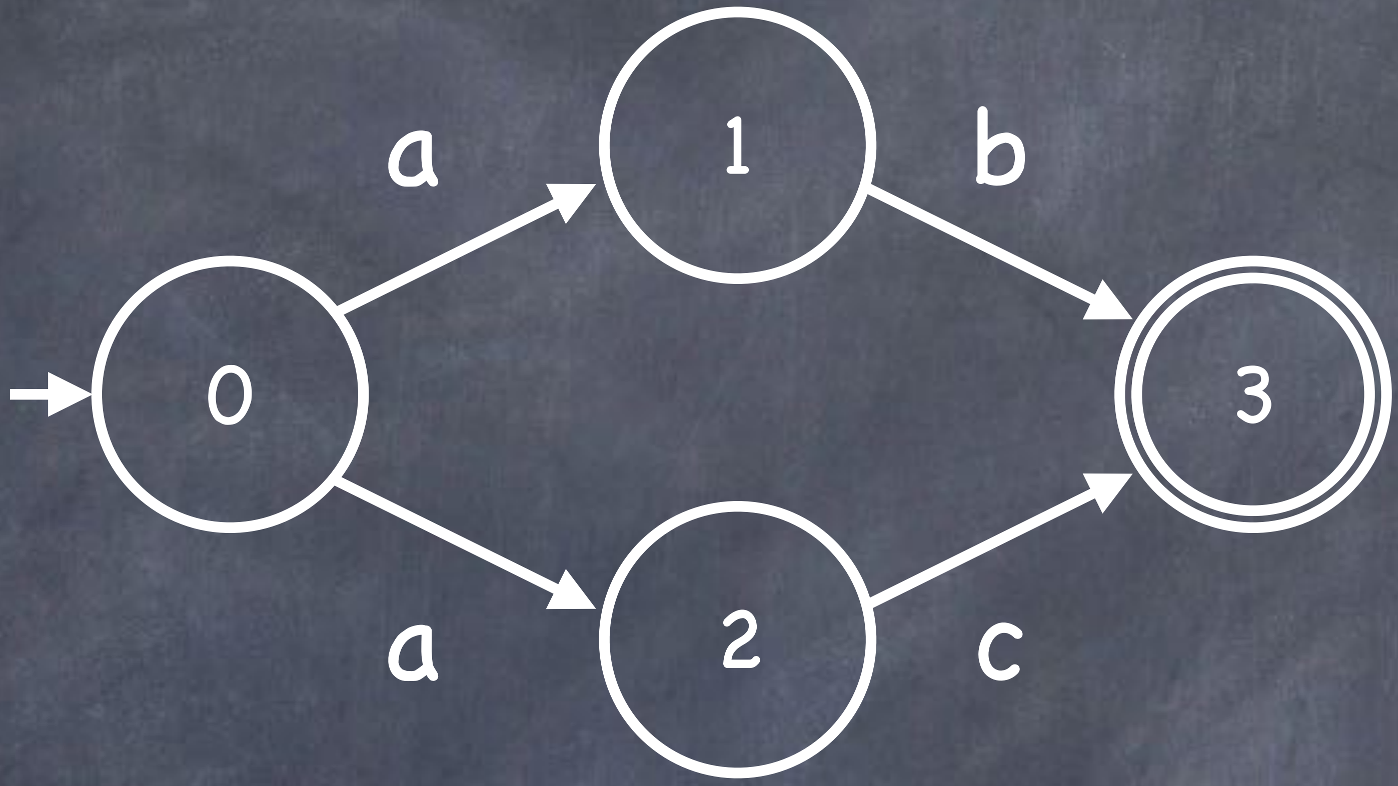
Nondeterministic Automata (NFA)

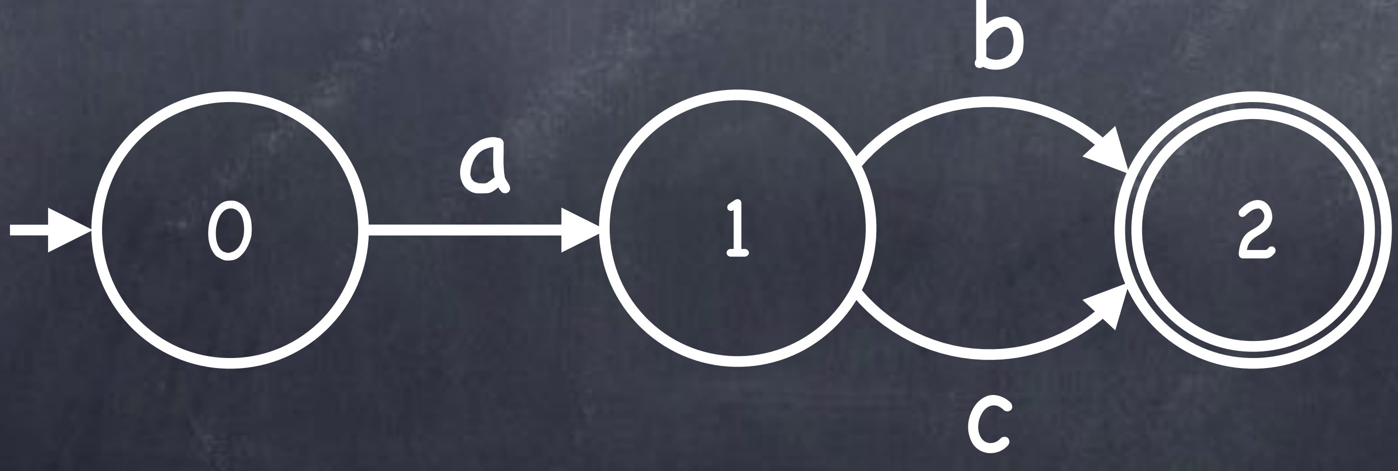


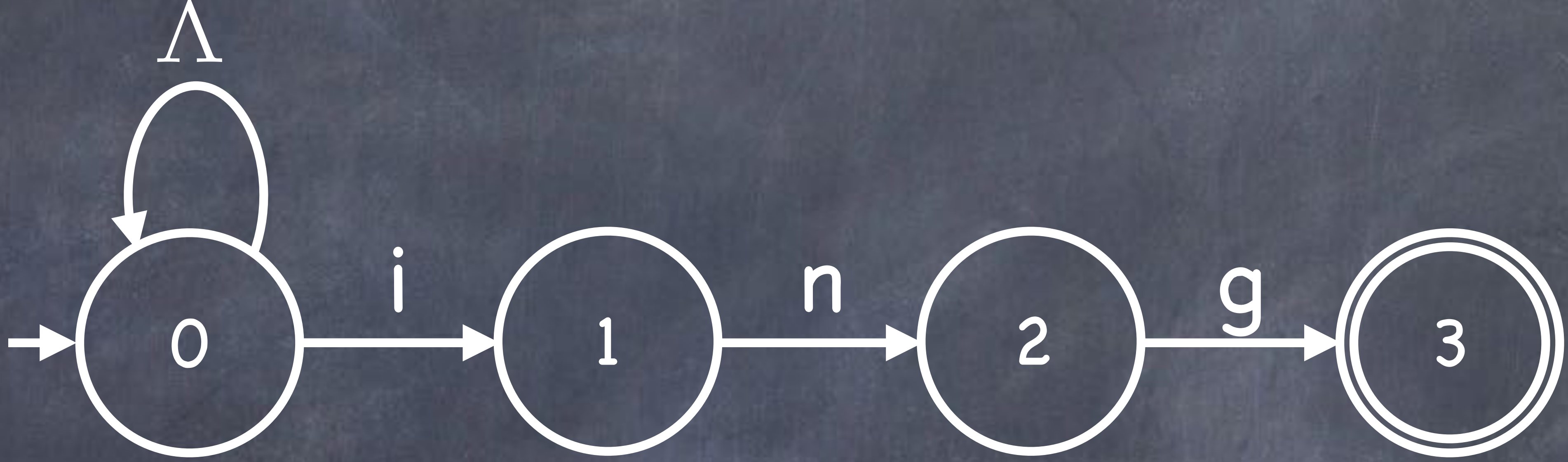
May have two or more transitions from a state on the same input symbol

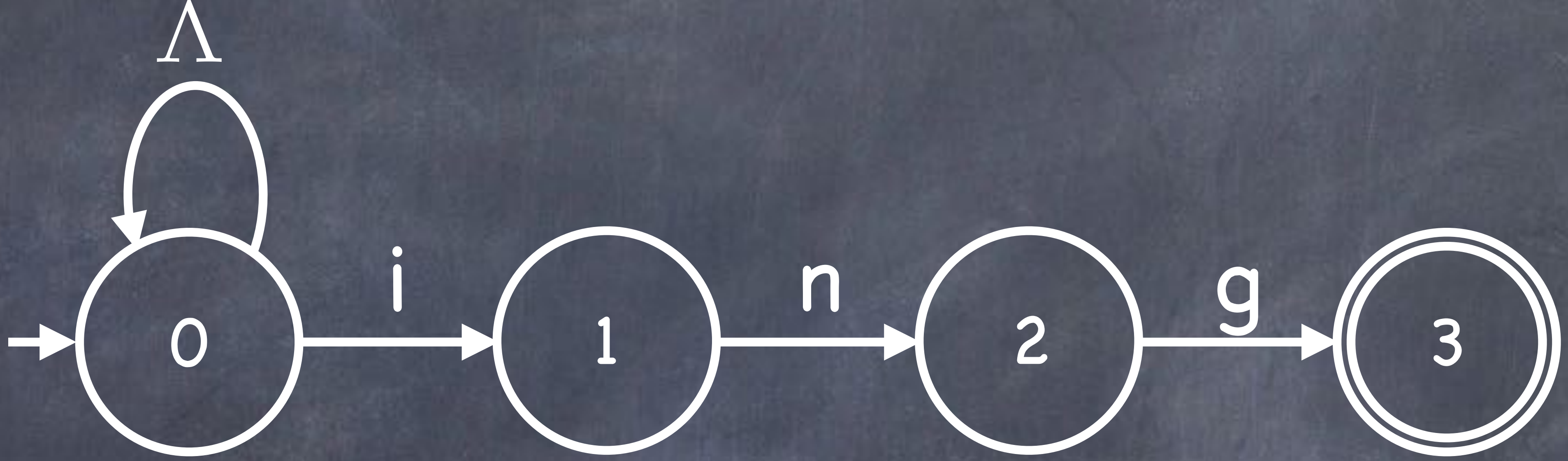

a
1
b
0
3
a
2
c


b
a
0
1
2
c

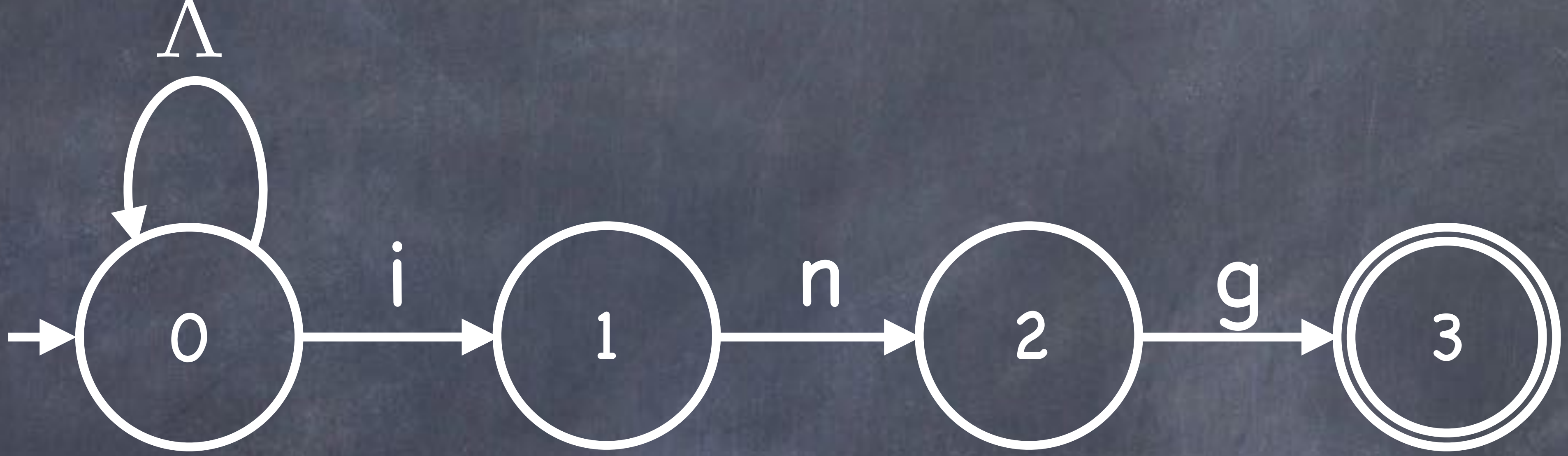

a
1
b
0
3
a
2
c


b
0
a
1
2
c

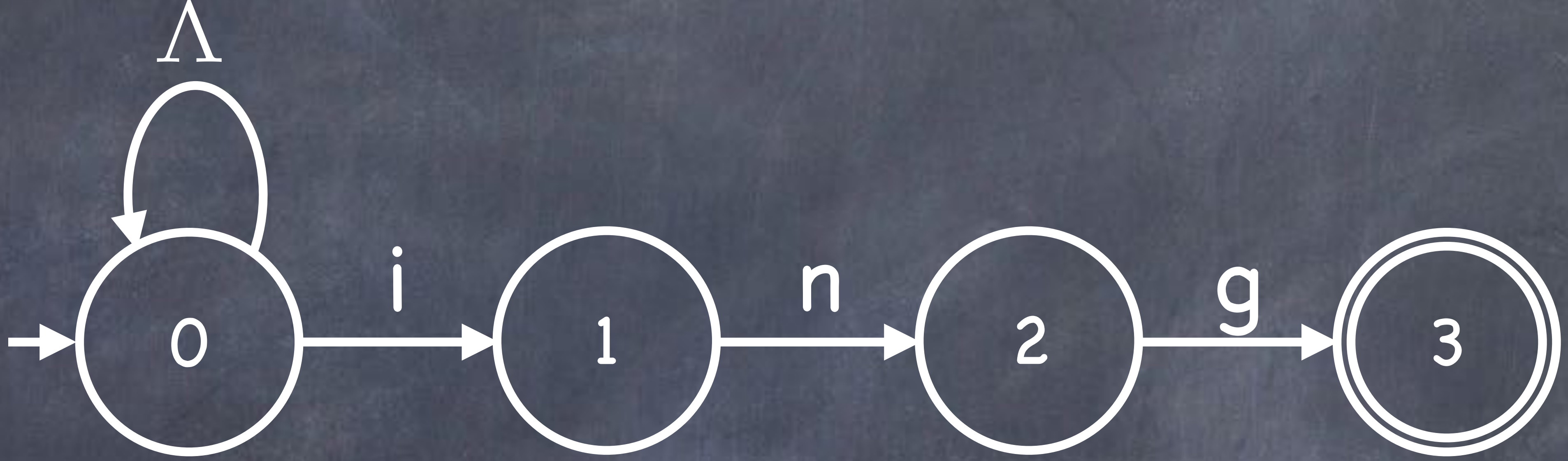

Λ
0
i
1
n
2
g
3



Input: “being”



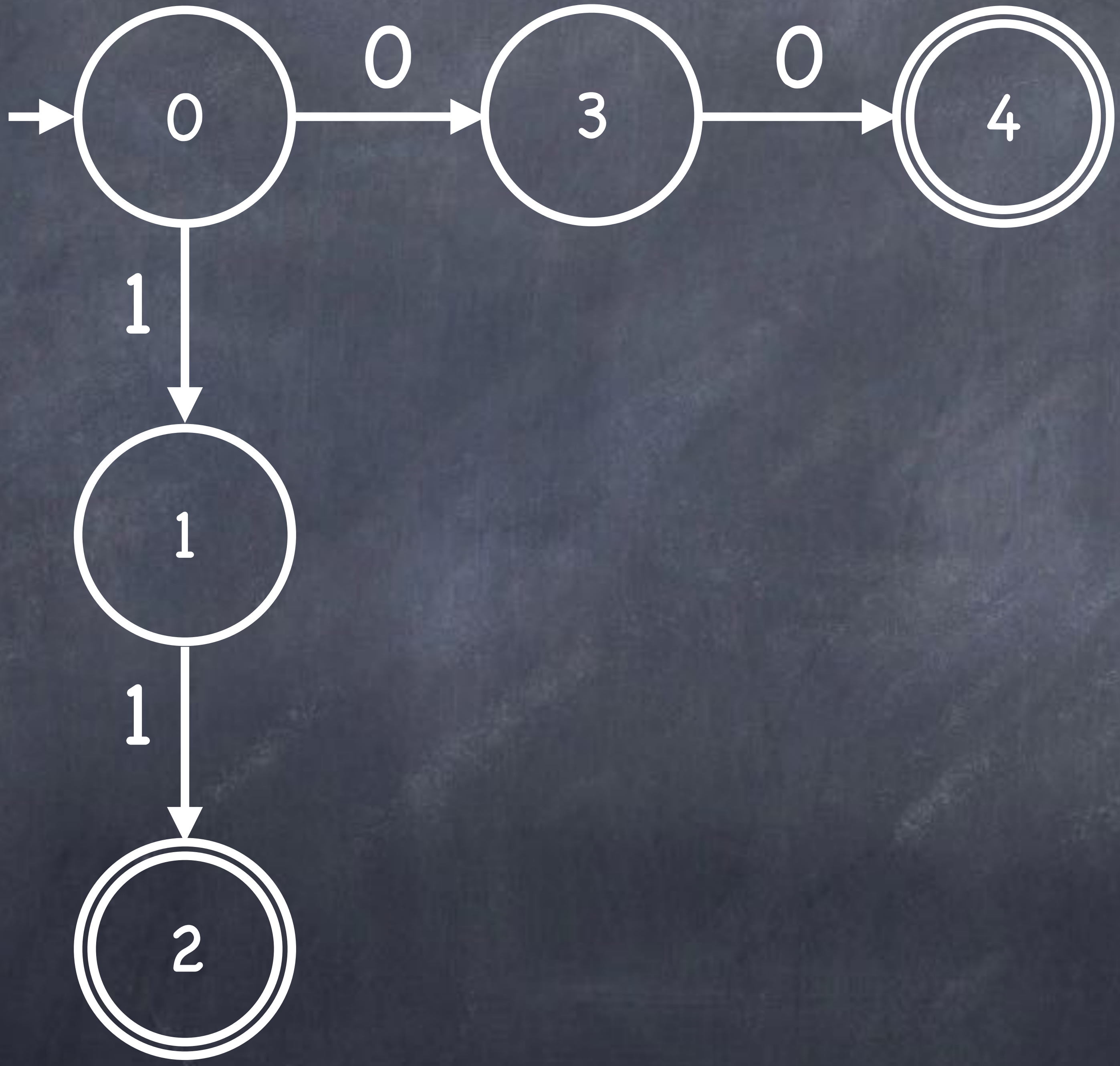
Input: “hide”

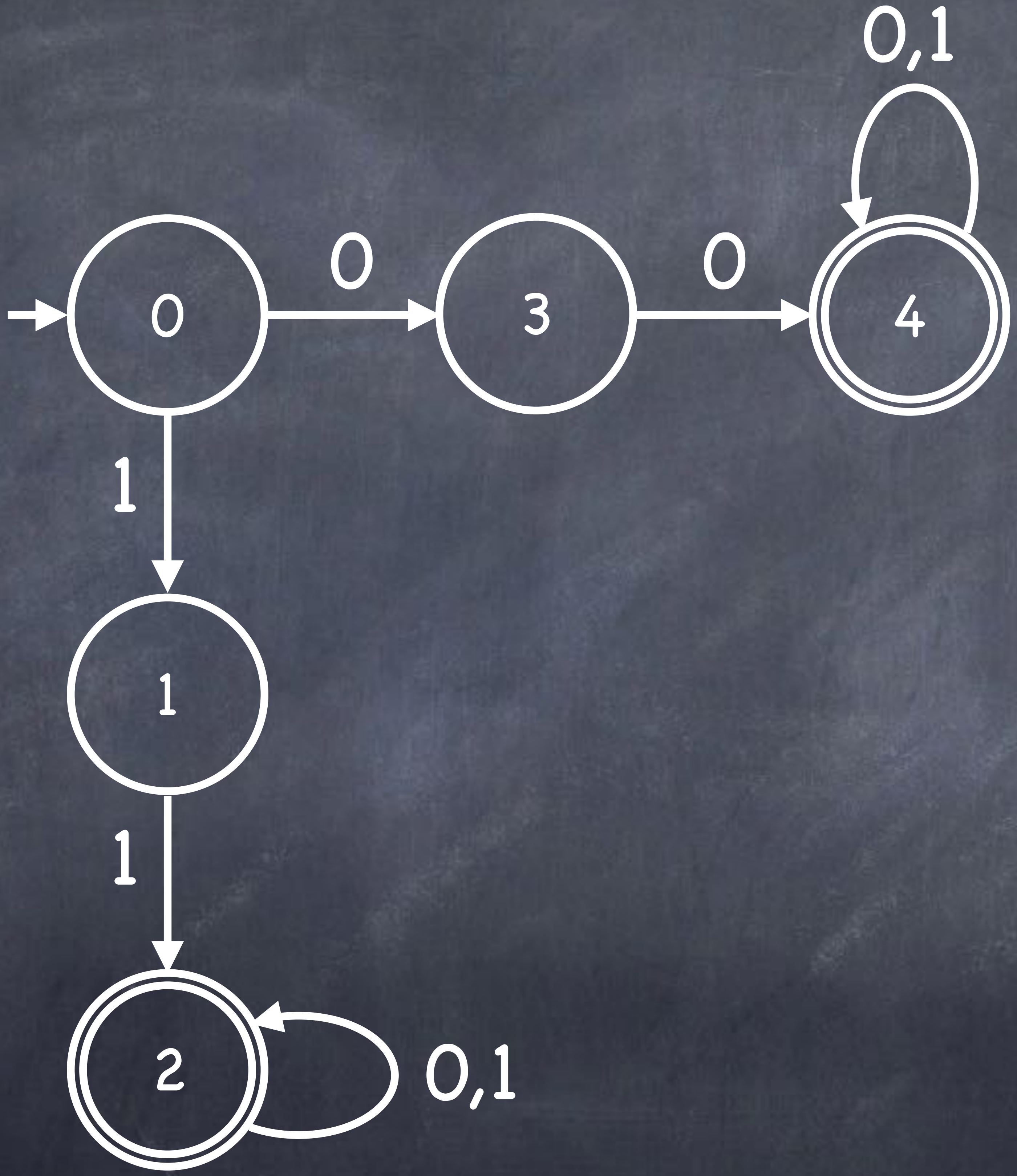


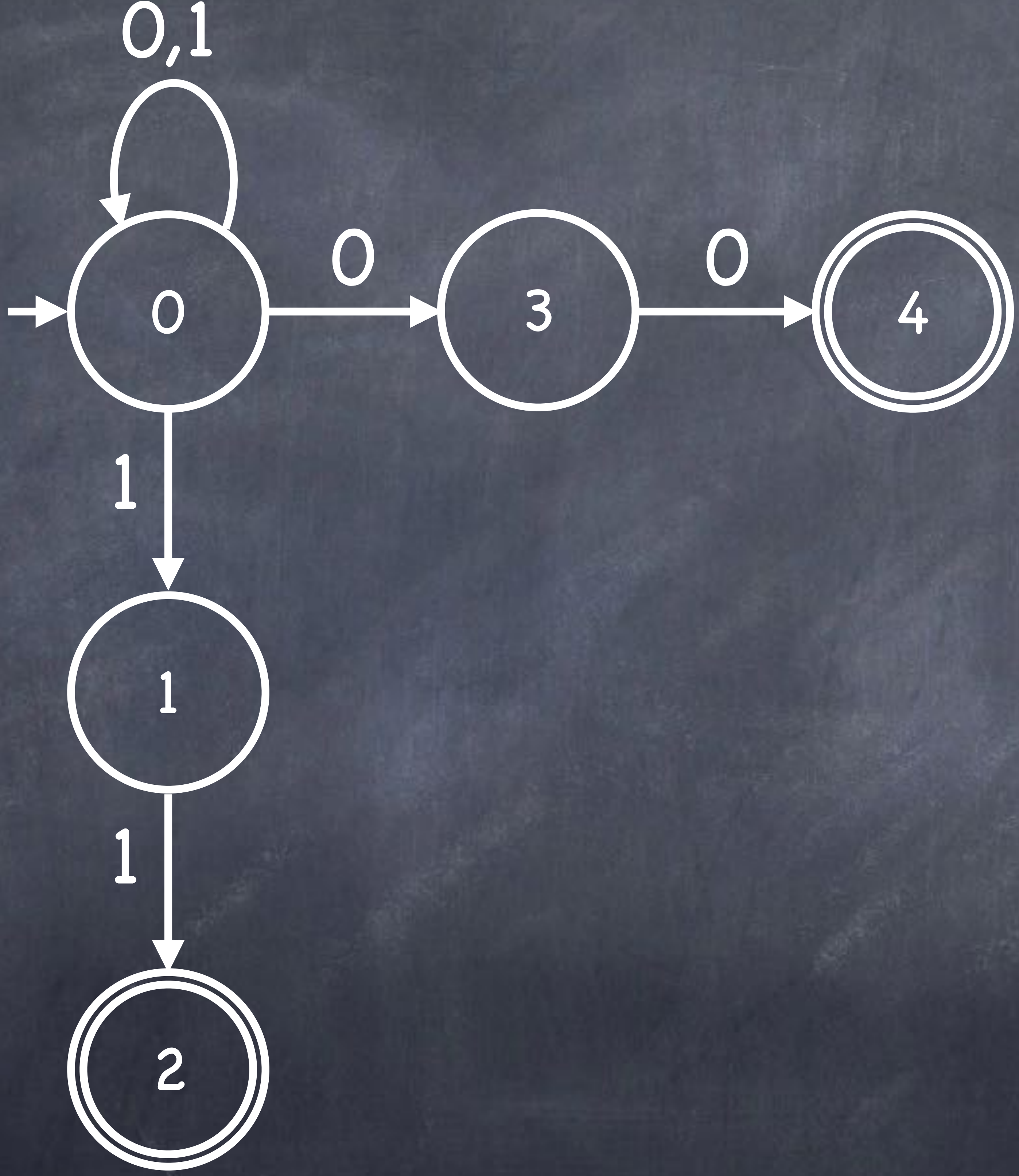
Input: “hiding”

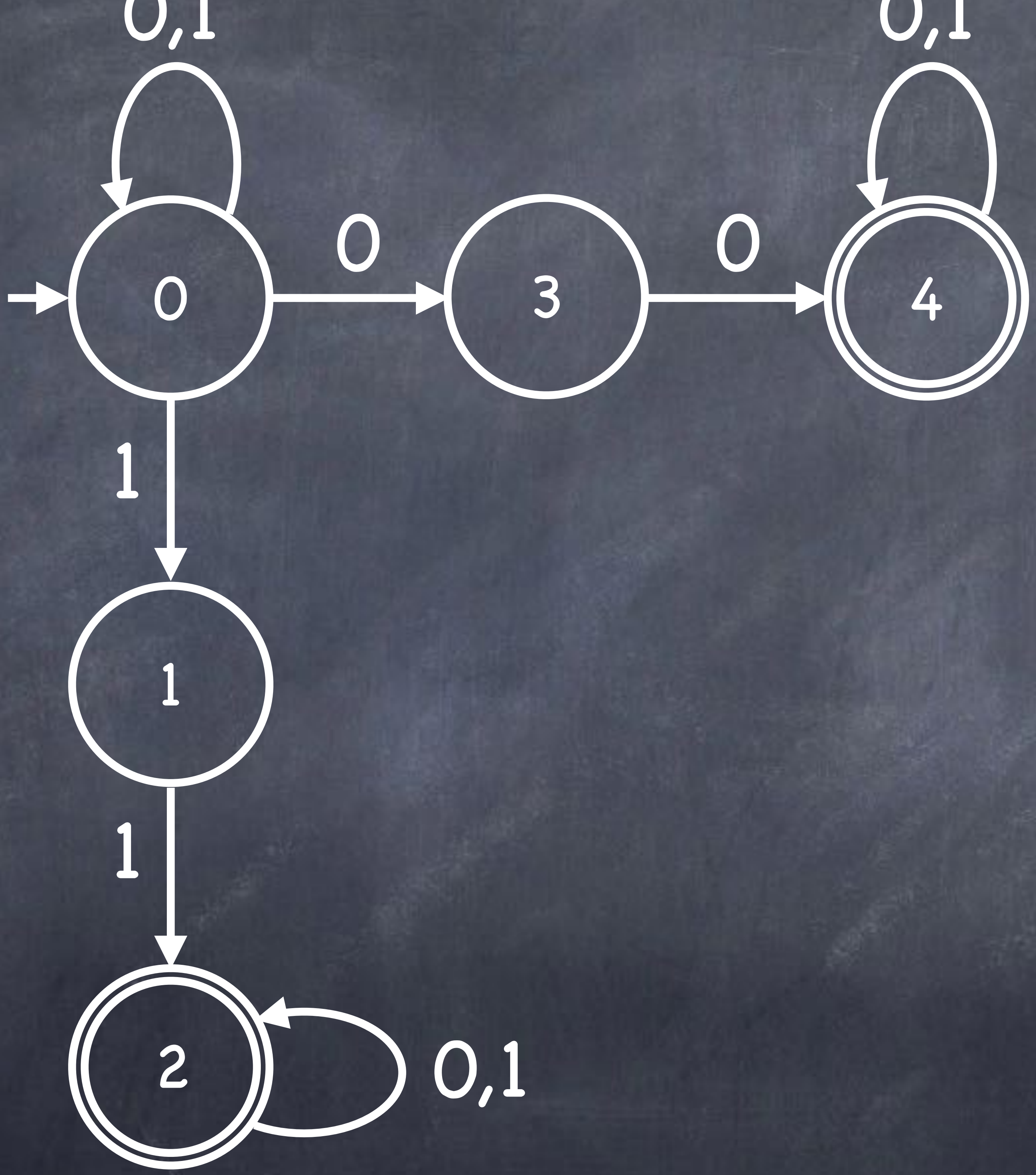
Acceptance in NFAs

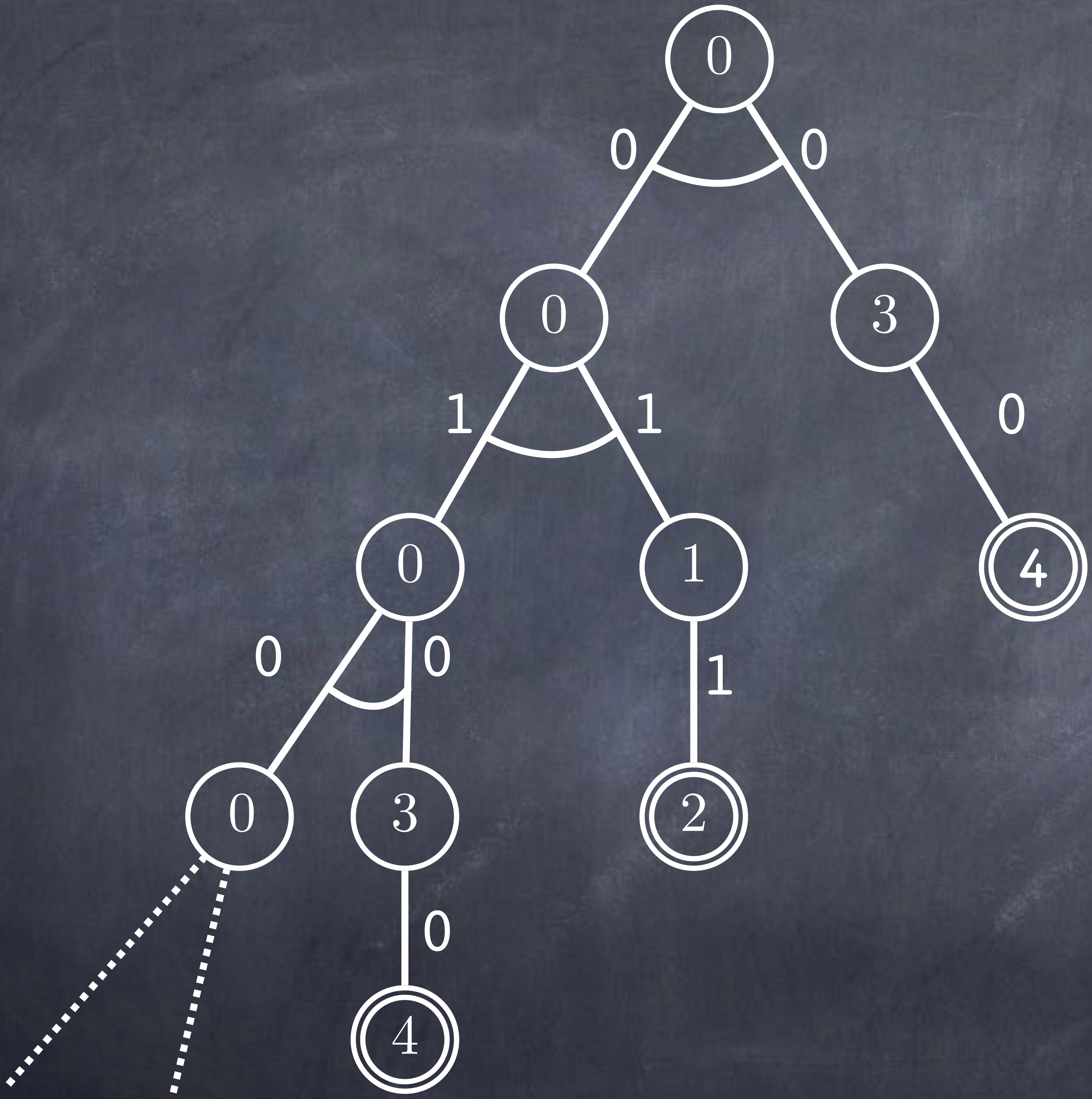
- Input sequence a_1 a_2 … a_k can label many paths
- An NFA accepts if any path accepts (ends in an accepting state)
- Nondeterminism allows an automaton to “guess” which transition to take

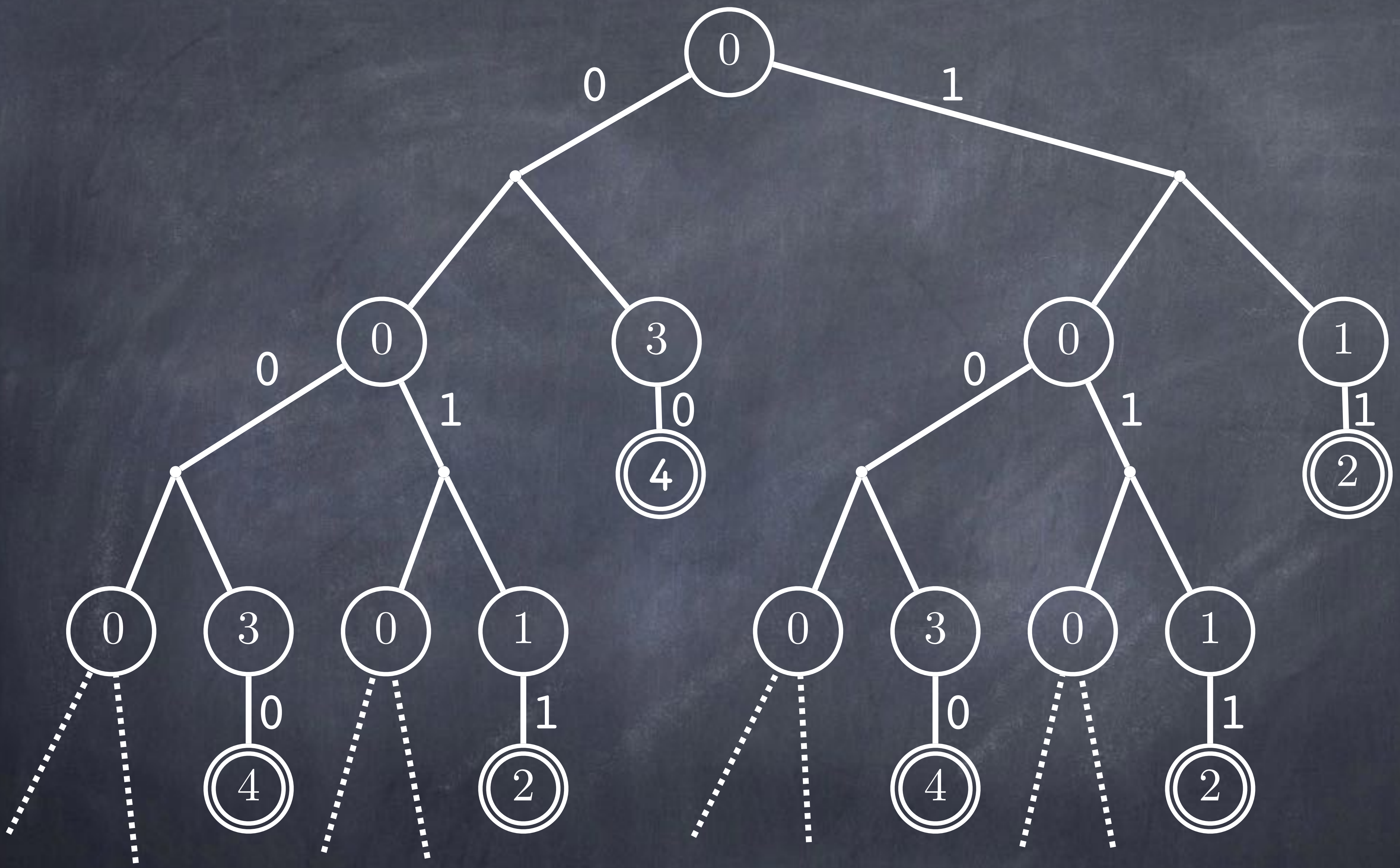

0
0
0
3
0
4
1
1
1
2


0,1
0
0
0
3
4
1
1
1
2
0,1


0,1
0
0
0
3
4
1
1
1
2


0,1
0,1
0
0
0
3
4
1
1
1
2
0,1

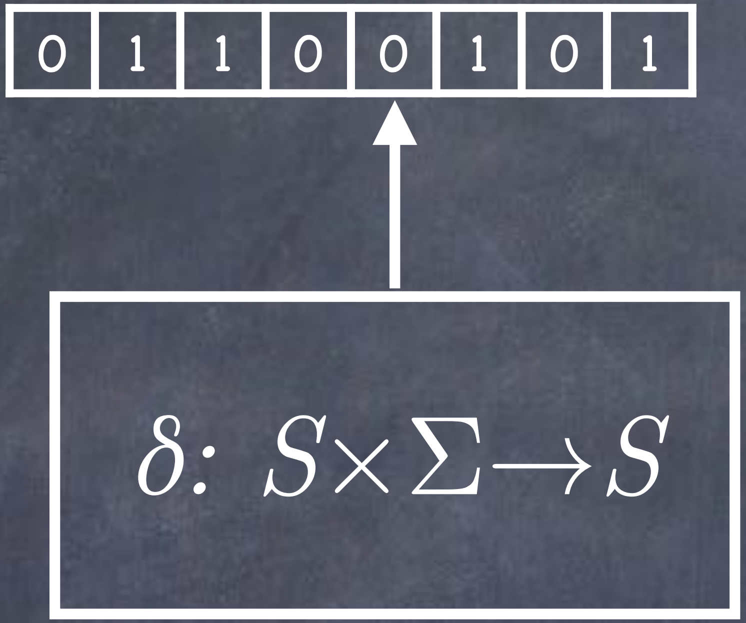

0
0
0
0
3
1
1
0
0
1
4
0
0
1
0
3
2
0
4
4


0
0
1
0
3
0
1
0
1
0
1
4
2
0
3
0
1
0
3
0
1
0
1
4
2
4
2

Acceptance in NFAs

- Input sequence a_1 a_2 ... a_k can label many paths
- An NFA accepts if any path accepts (ends in an accepting state)
- Nondeterminism allows an automaton to “guess” which transition to take

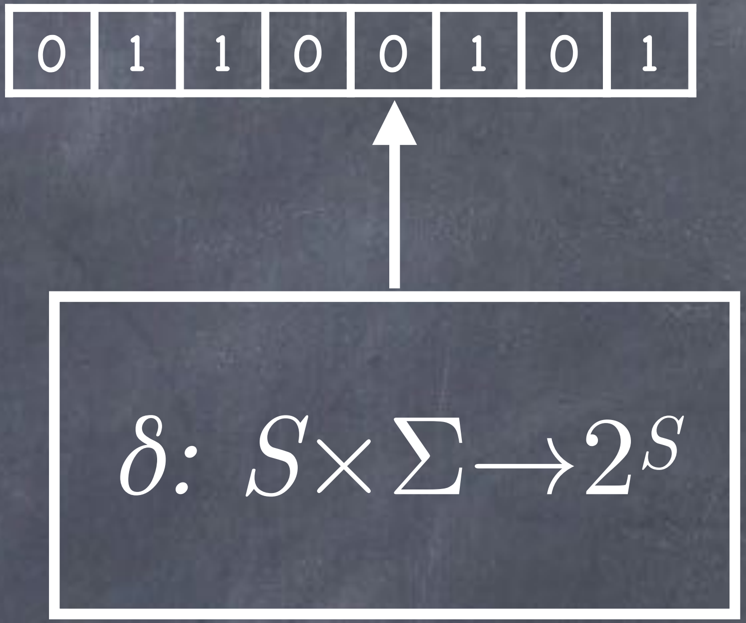
DFA



0	1	1	0	0	1	0	1

$\delta\colon\ S\times\Sigma\rightarrow S$

NFA



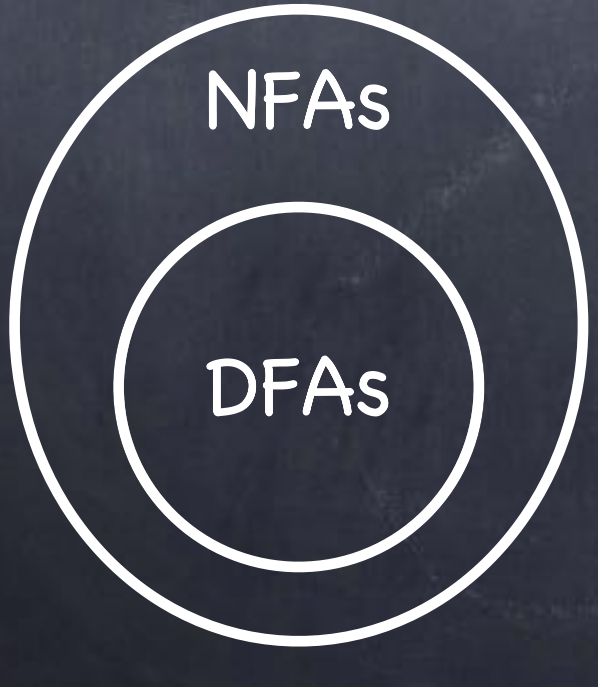
0	1	1	0	0	1	0	1

$\delta\colon\ S\times\Sigma\rightarrow 2^S$

DFAs and NFAs

- All DFAs are NFAs
- Not all NFAs are DFAs

$$DFAs \subset NFAs$$

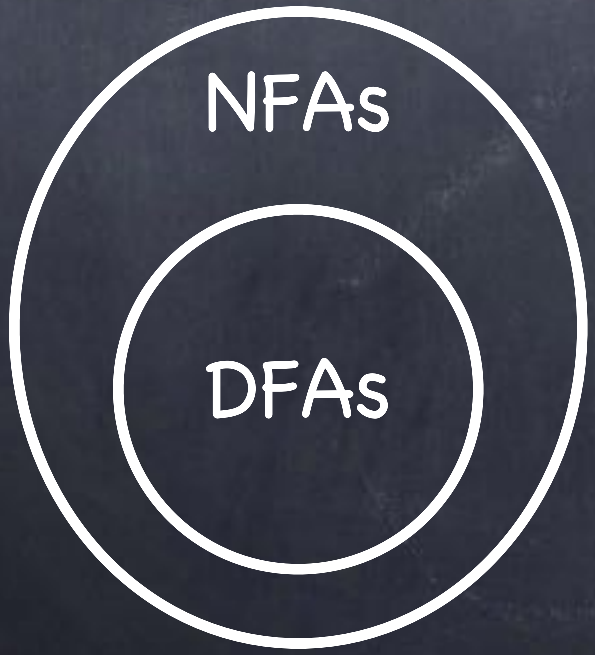


DFAs and NFAs

- All DFAs are NFAs
- Not all NFAs are DFAs

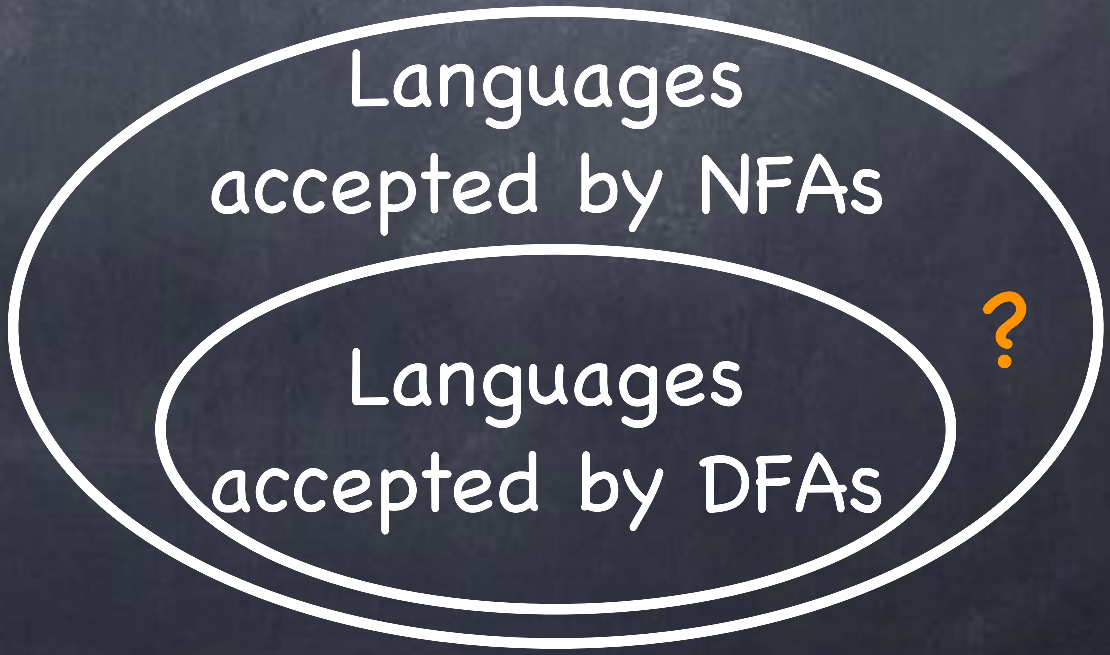
$DFAs \subset NFAs$

$L(DFAs) \overset{?}{\subset} L(NFAs)$



NFAs

DFAs



Languages accepted by NFAs

Languages accepted by DFAs

?

For Next Class

- Homework 1.1
- FOCS 10.4