Pushdown Automata

- Pushdown Automata
  - Stacks and recursiveness
  - Formal Definition

Recursive Algorithms and Stacks

- General Principle in Computer Science: Any recursive algorithm can be turned into a non-recursive one using a stack and a while-loop which exits only when stack is empty.

EG: JVM keeps stack of activation records generated at each method call. Consider:

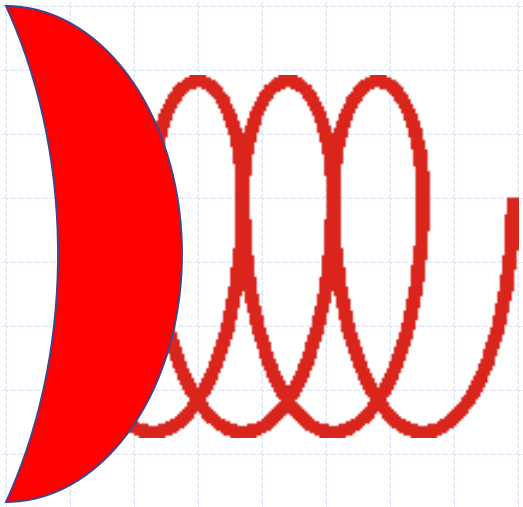
```
long factorial(int n){
  if (n<=0) return 1;
  return n*factorial(n-1);
}
```

What does the JVM do for `factorial(5)`?

Recursive Algorithms and Stacks

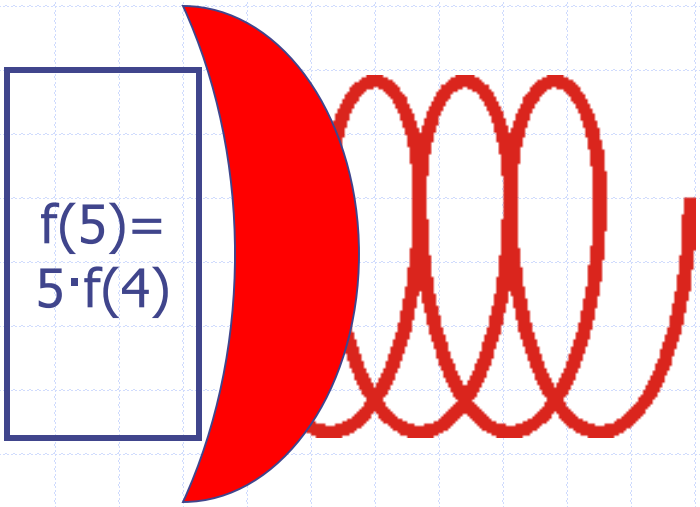
```
long factorial(int n){
   if (n<=0) return 1;
   return n*factorial(n-1);
}
```

Compute 5!



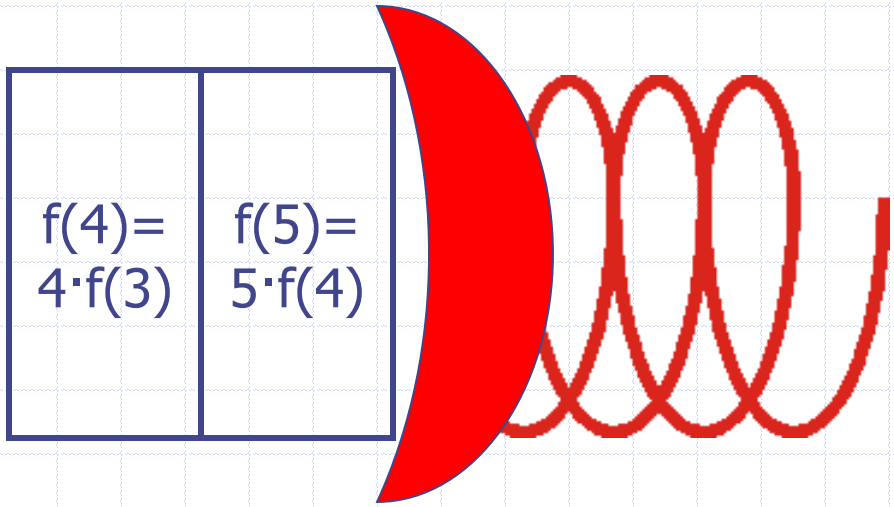
Recursive Algorithms and Stacks

```
long factorial(int n){
   if (n<=0) return 1;
   return n*factorial(n-1);
}
```



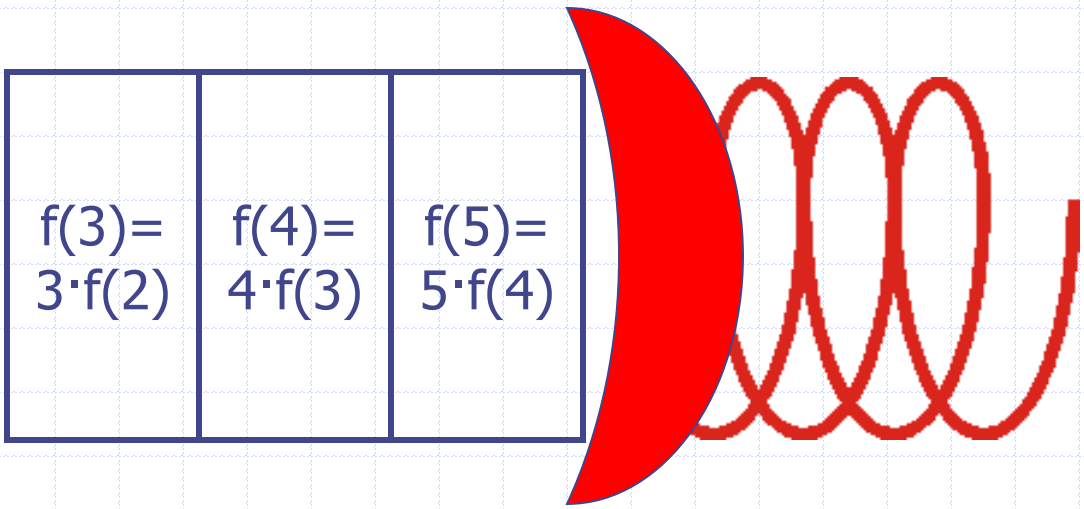
Recursive Algorithms and Stacks

```
long factorial(int n){
   if (n<=0) return 1;
   return n*factorial(n-1);
}
```



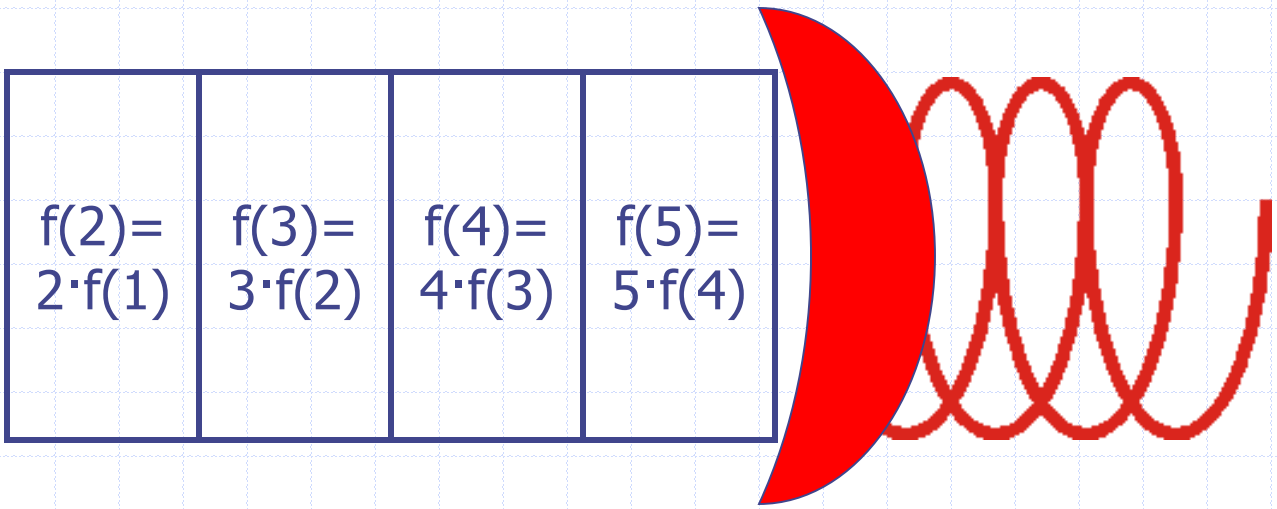
Recursive Algorithms and Stacks

```
long factorial(int n){
   if (n<=0) return 1;
   return n*factorial(n-1);
}
```



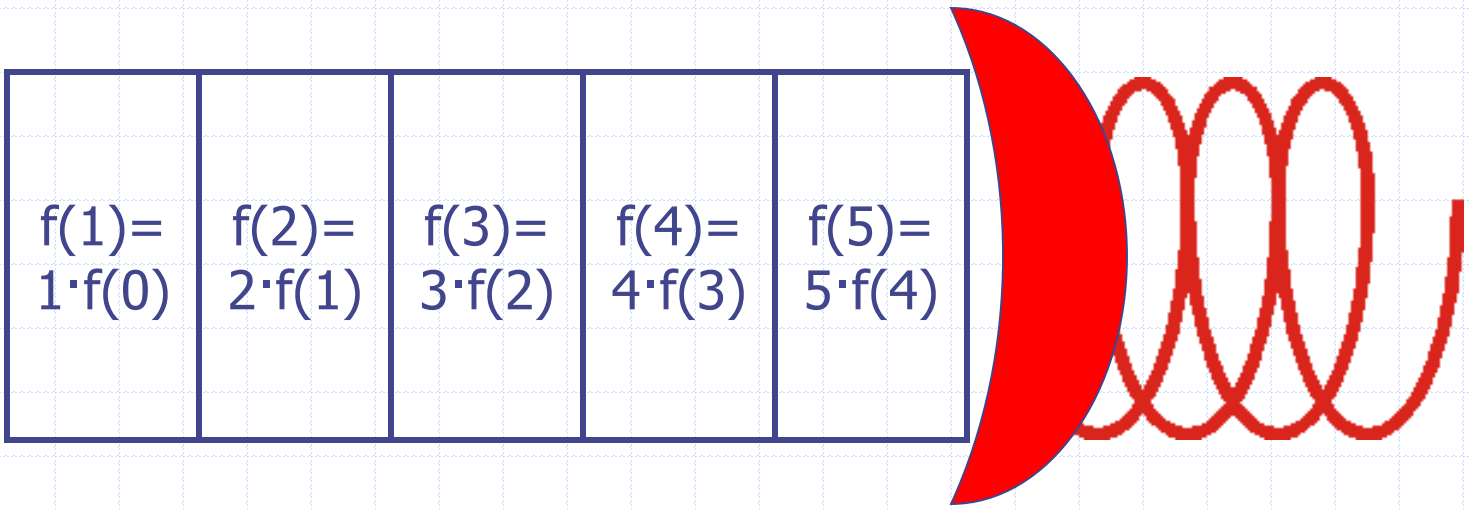
Recursive Algorithms and Stacks

```
long factorial(int n){
   if (n<=0) return 1;
   return n*factorial(n-1);
}
```



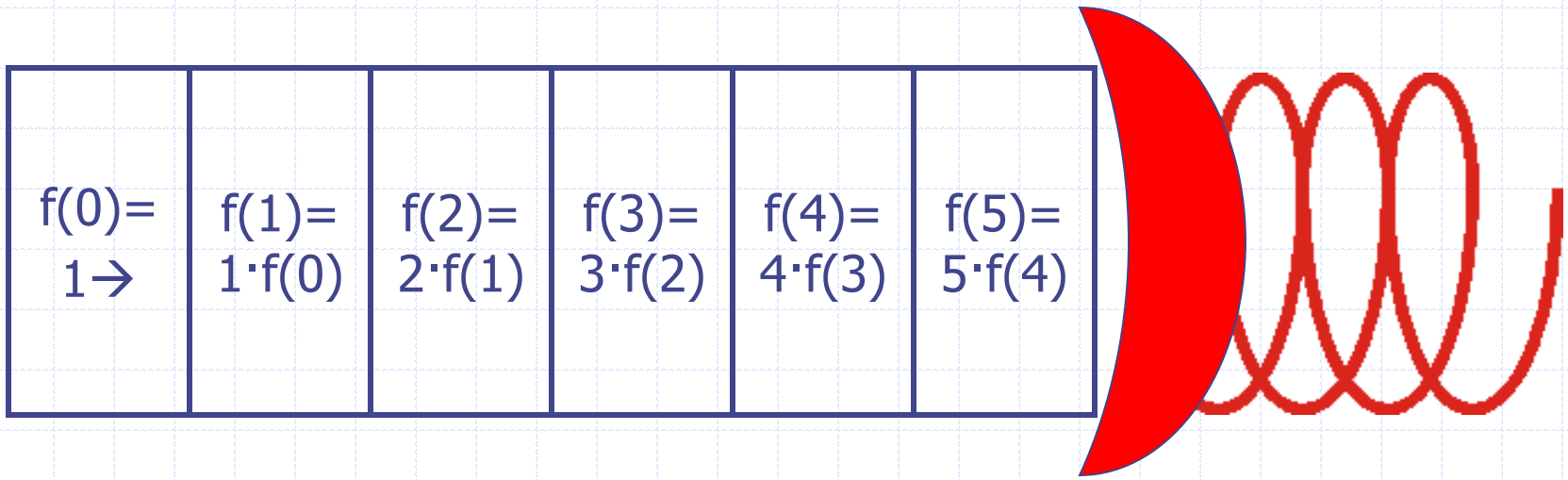
Recursive Algorithms and Stacks

```
long factorial(int n){
   if (n<=0) return 1;
   return n*factorial(n-1);
}
```



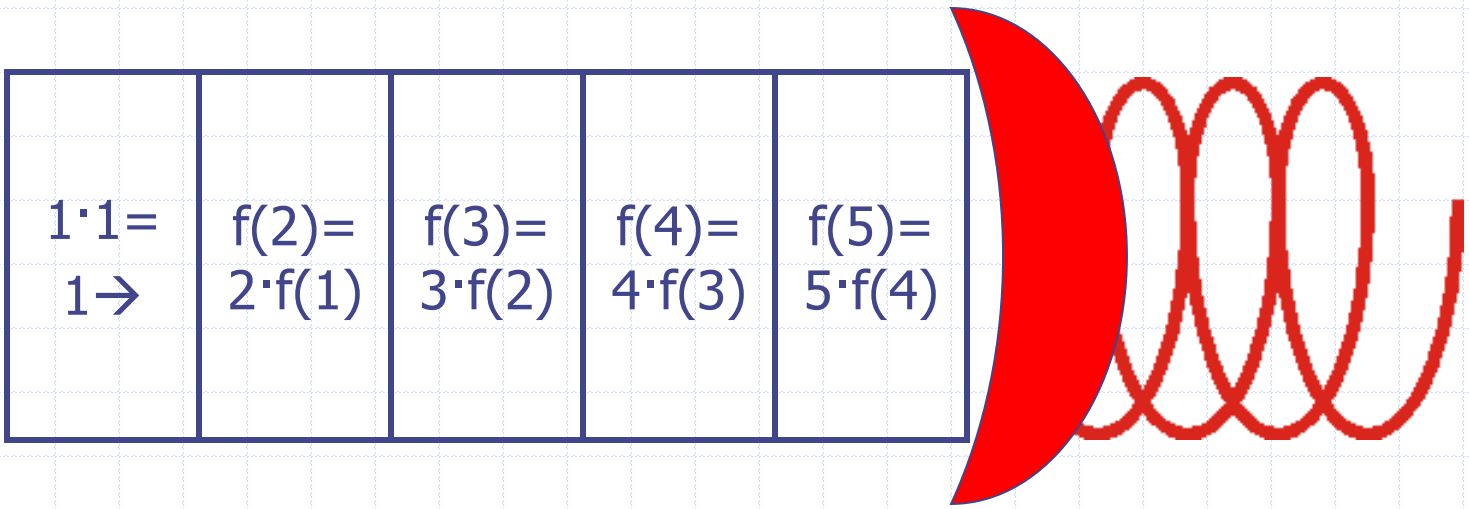
Recursive Algorithms and Stacks

```
long factorial(int n){
   if (n<=0) return 1;
   return n*factorial(n-1);
}
```



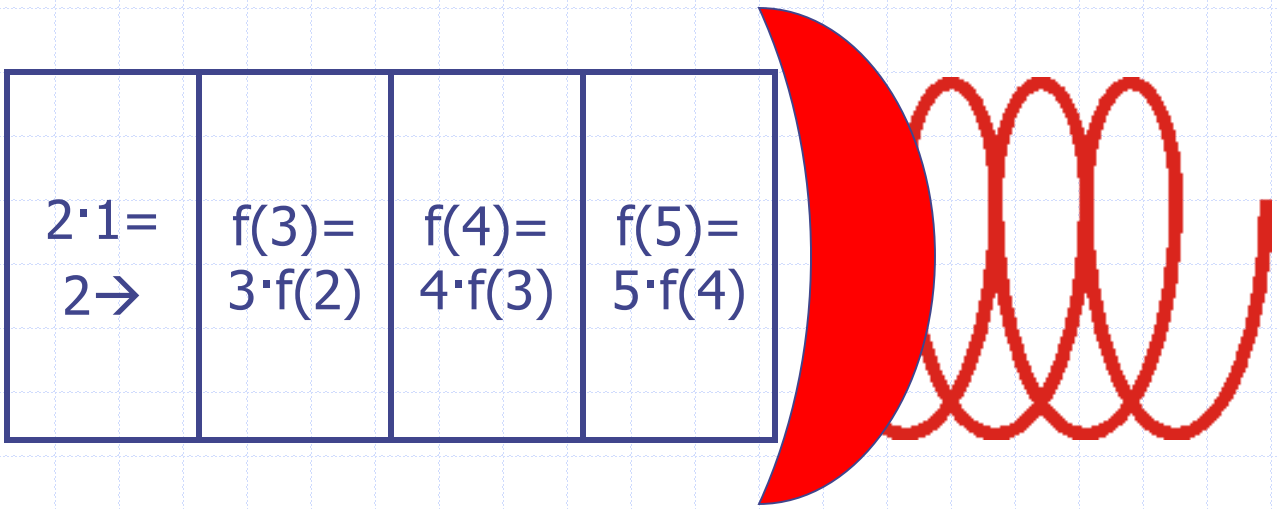
Recursive Algorithms and Stacks

```
long factorial(int n){
   if (n<=0) return 1;
   return n*factorial(n-1);
}
```



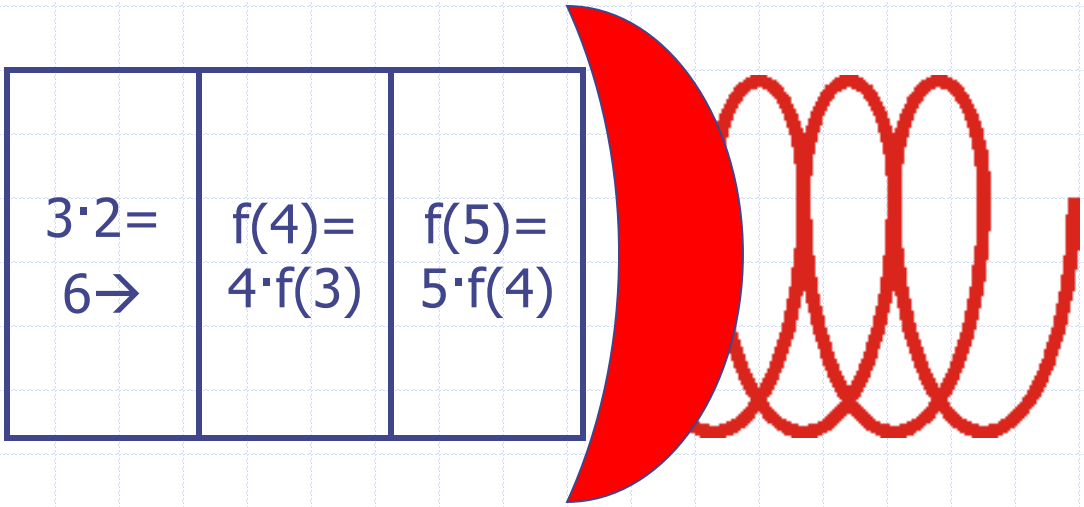
Recursive Algorithms and Stacks

```
long factorial(int n){
   if (n<=0) return 1;
   return n*factorial(n-1);
}
```



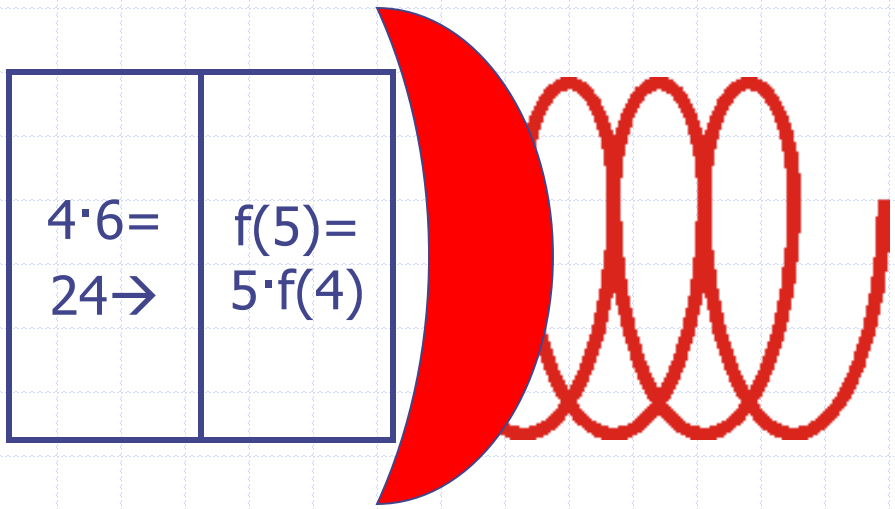
Recursive Algorithms and Stacks

```
long factorial(int n){
   if (n<=0) return 1;
   return n*factorial(n-1);
}
```



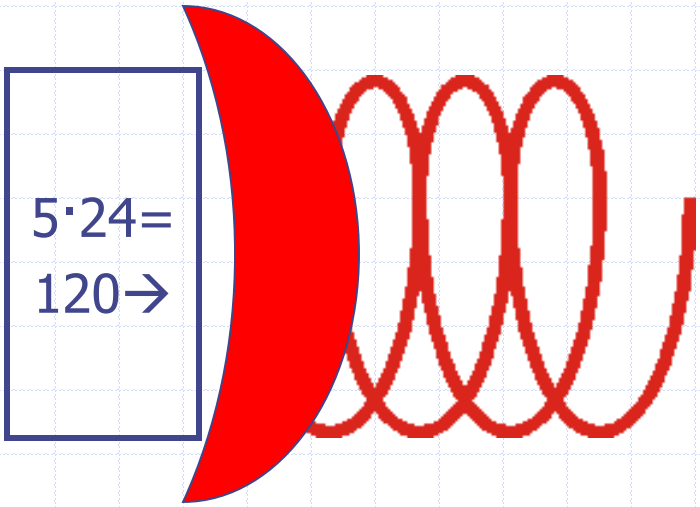
Recursive Algorithms and Stacks

```
long factorial(int n){
   if (n<=0) return 1;
   return n*factorial(n-1);
}
```



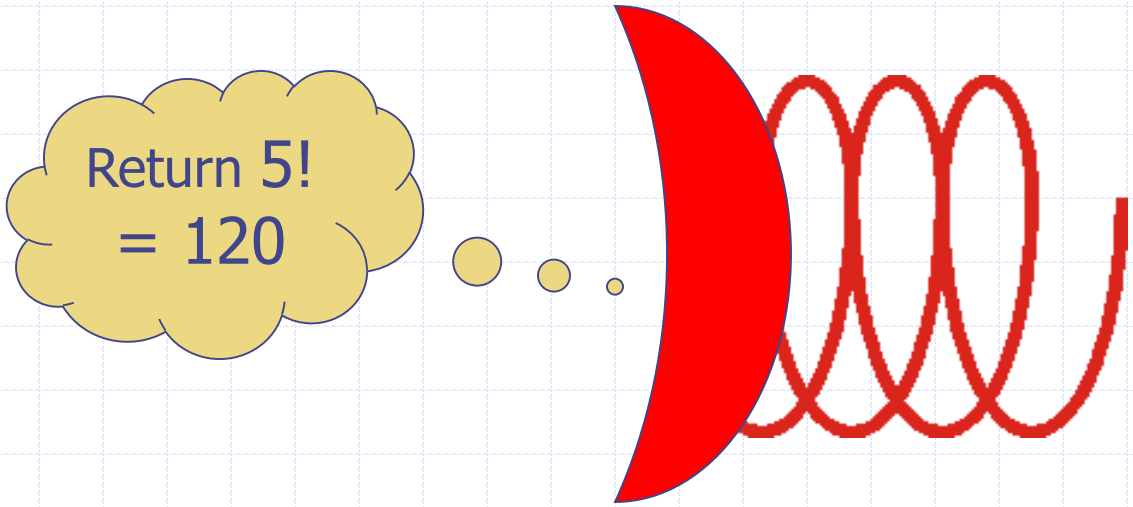
Recursive Algorithms and Stacks

```
long factorial(int n){
   if (n<=0) return 1;
   return n*factorial(n-1);
}
```



Recursive Algorithms and Stacks

```
long factorial(int n){
   if (n<=0) return 1;
   return n*factorial(n-1);
}
```



From CFG's to Stack Machines

CFG's naturally define recursive procedure:

```
boolean derives(strings x, y)
1. if (x==y) return true
2. for(all u⇒y)
       if derives(x,u) return true
3. return false //no successful branch
```

EG: $S \rightarrow \# \mid aSa \mid bSb$

From CFG's to Stack Machines

By general principles, can carry out *any* recursive computation on a stack. Can do it on a restricted version of an activation record stack, called a "**P**ush**d**own (Stack) **A**utomaton" or **PDA** for short.

Q: What is the language generated by

$S \rightarrow \# \mid aSa \mid bSb$?

From CFG's to Stack Machines

A: Palindromes in {*a,b,#*}* containing exactly one #-symbol.

Q: Using a stack, how can we recognize such strings?

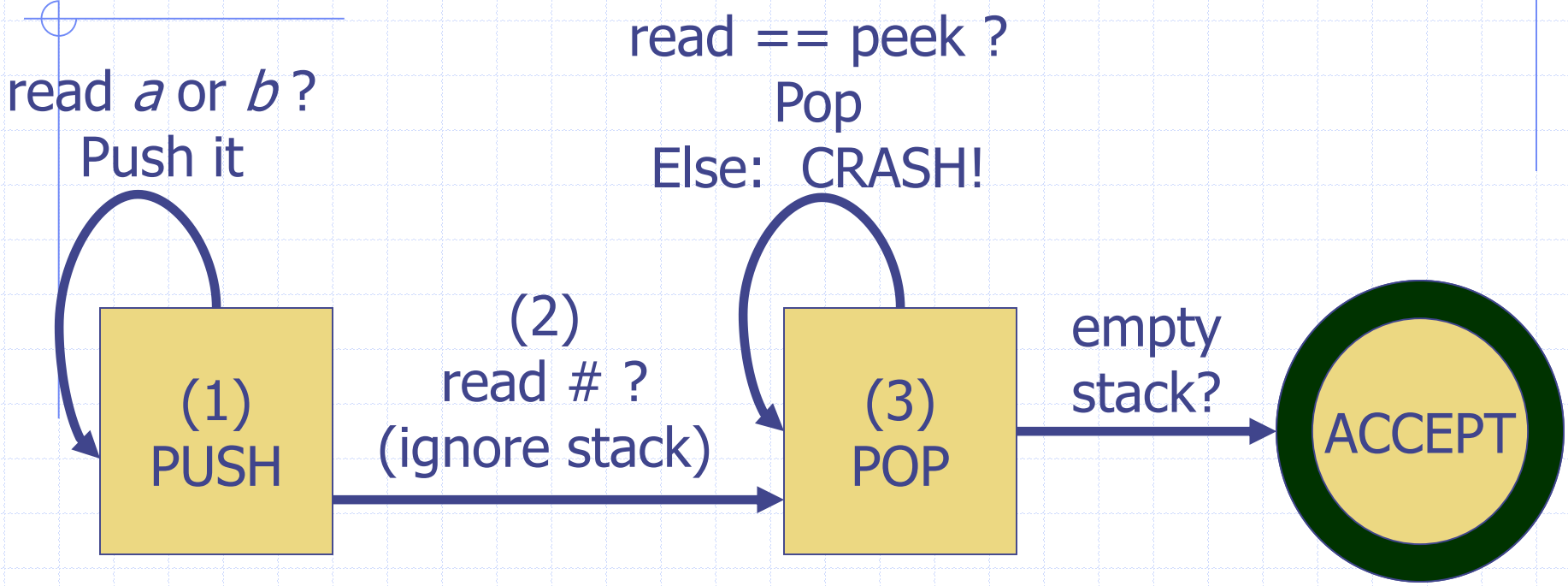
From CFG's to Stack Machines

A: Use a three phase process:

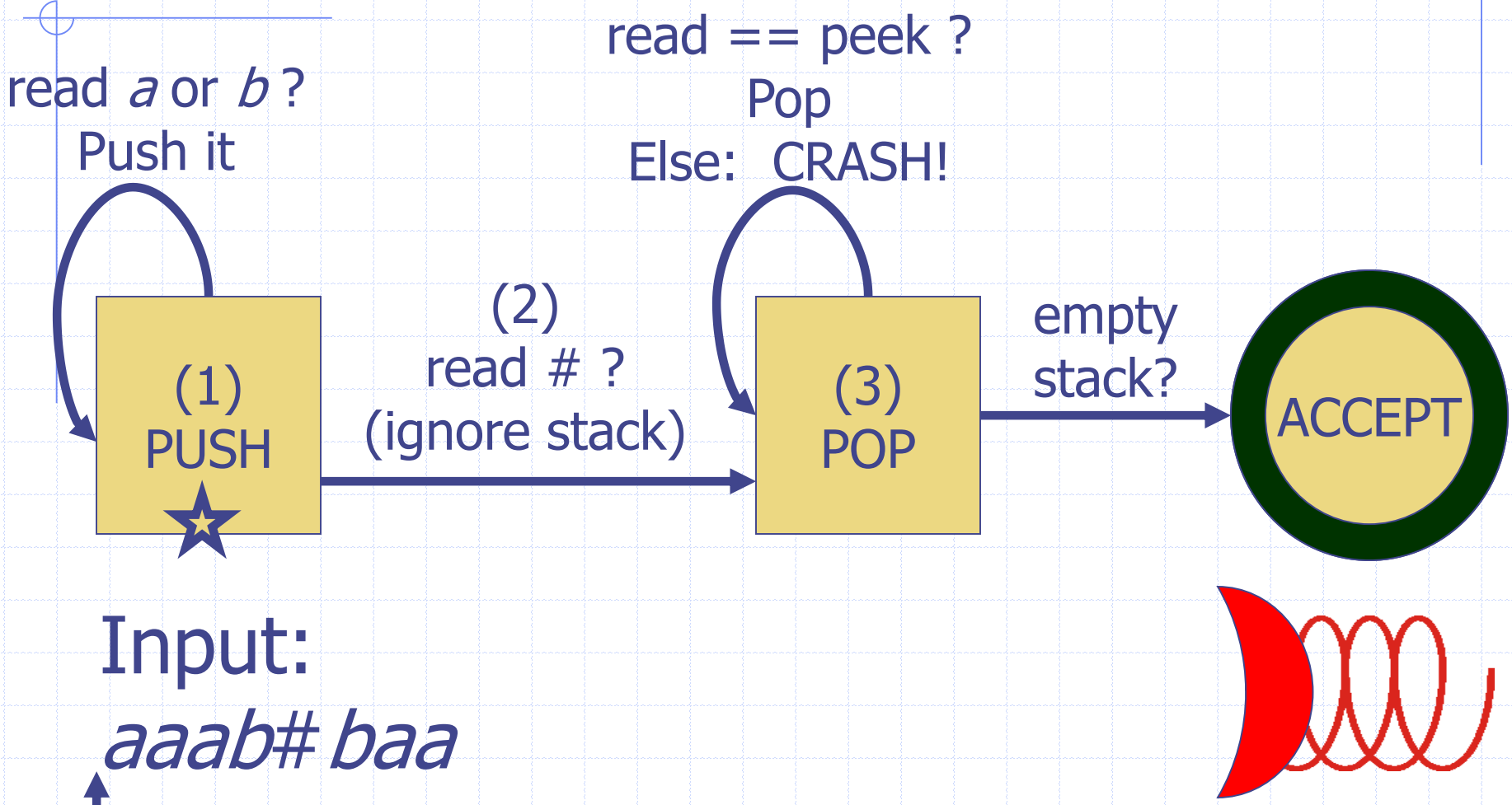
1. **Push mode:** Before reading "#", push everything on the stack.
2. Reading "#" switches modes.
3. **Pop mode:** Read remaining symbols making sure that each new read symbol is identical to symbol popped from stack.

Accept if able to empty stack completely. Otherwise reject, and reject if could not pop somewhere.

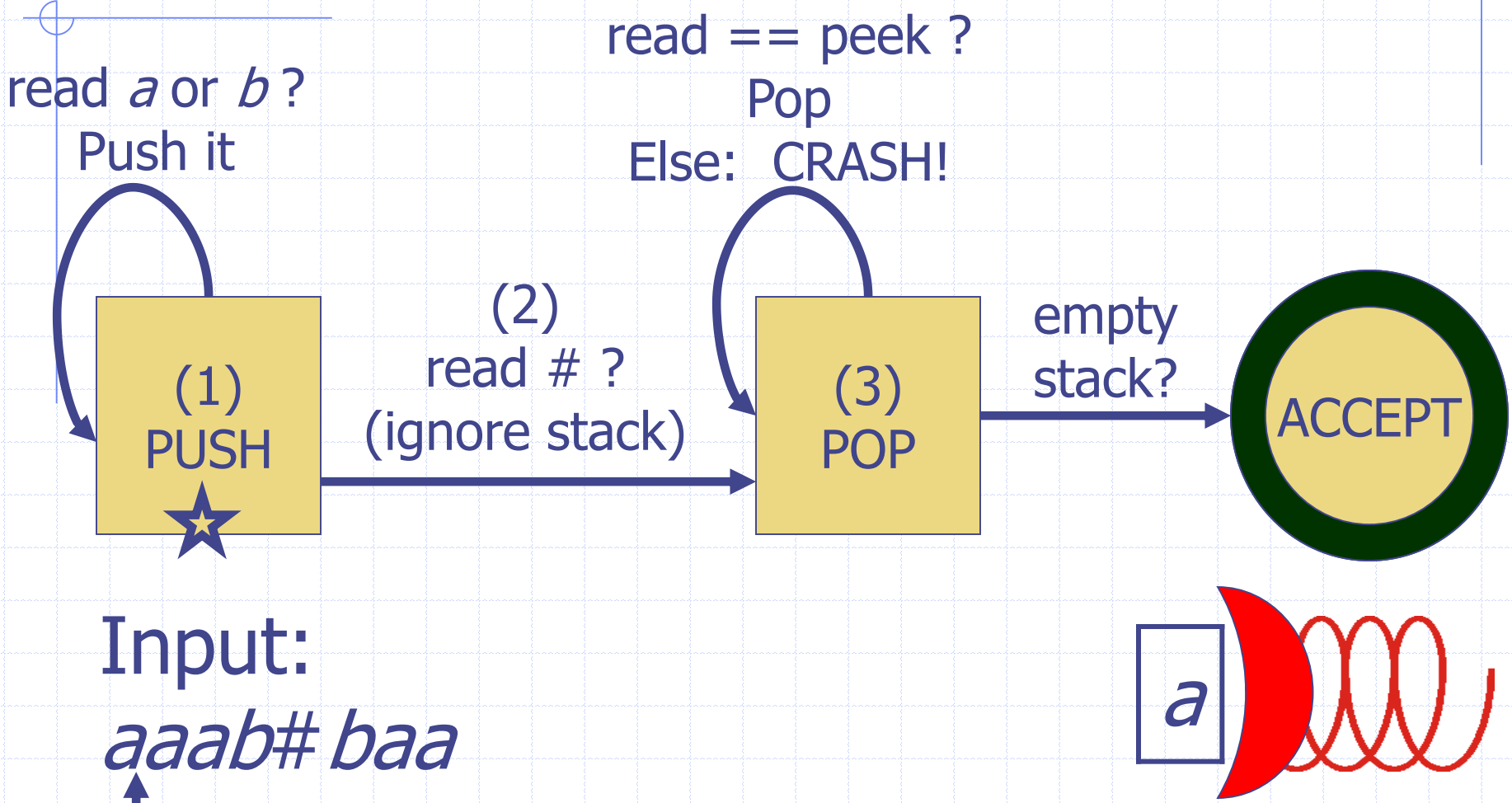
From CFG's to Stack Machines



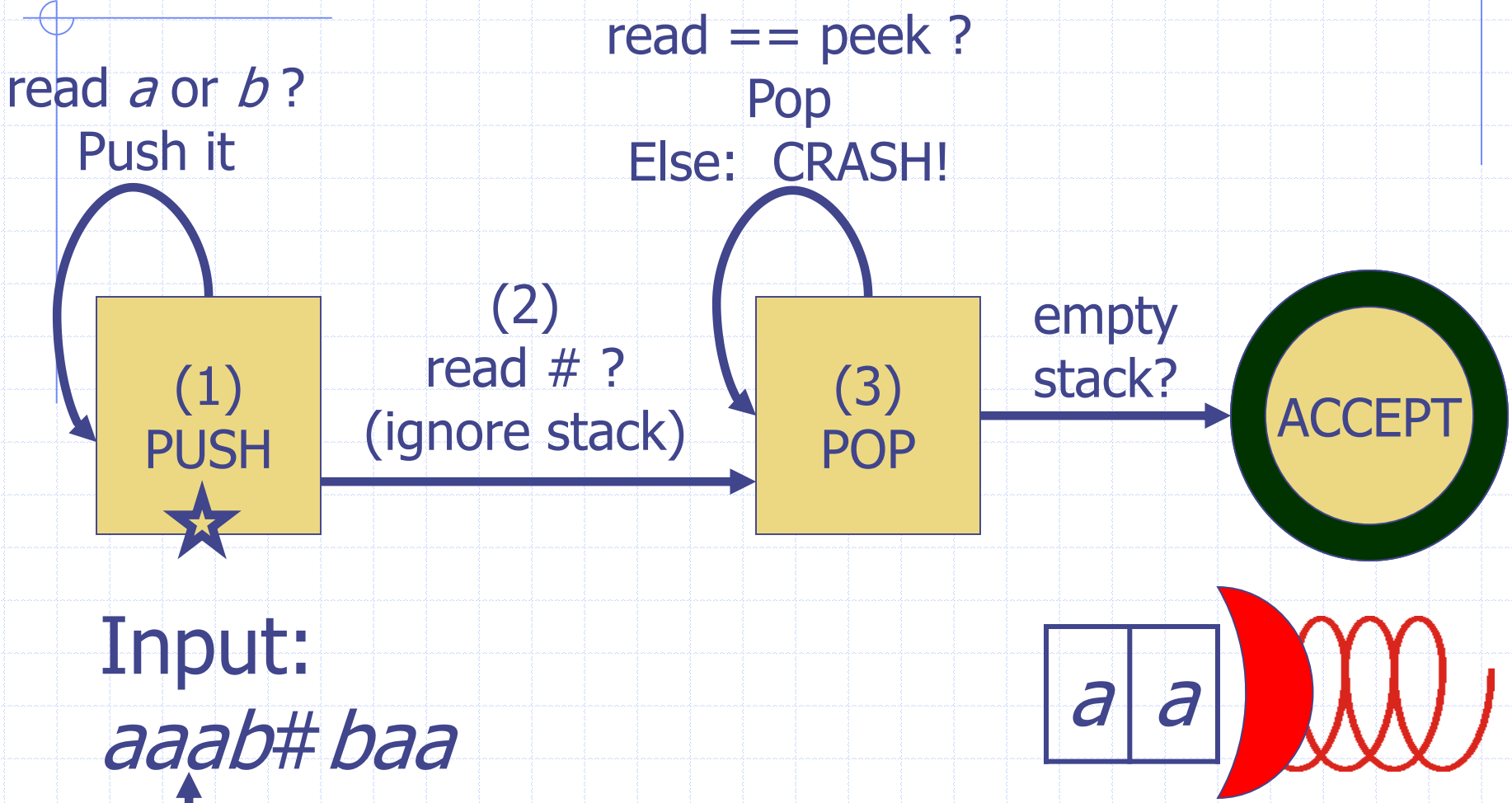
From CFG's to Stack Machines



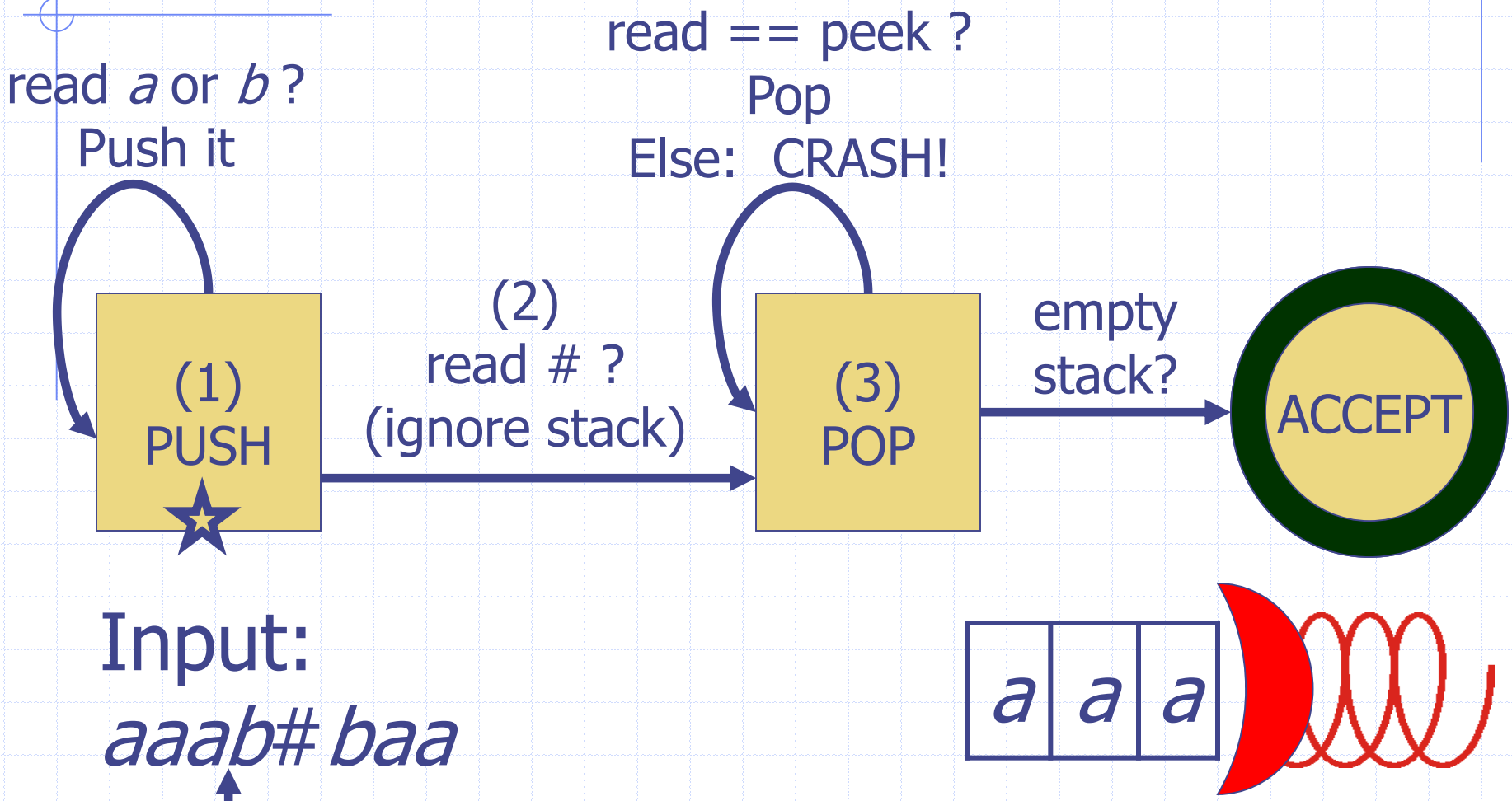
From CFG's to Stack Machines



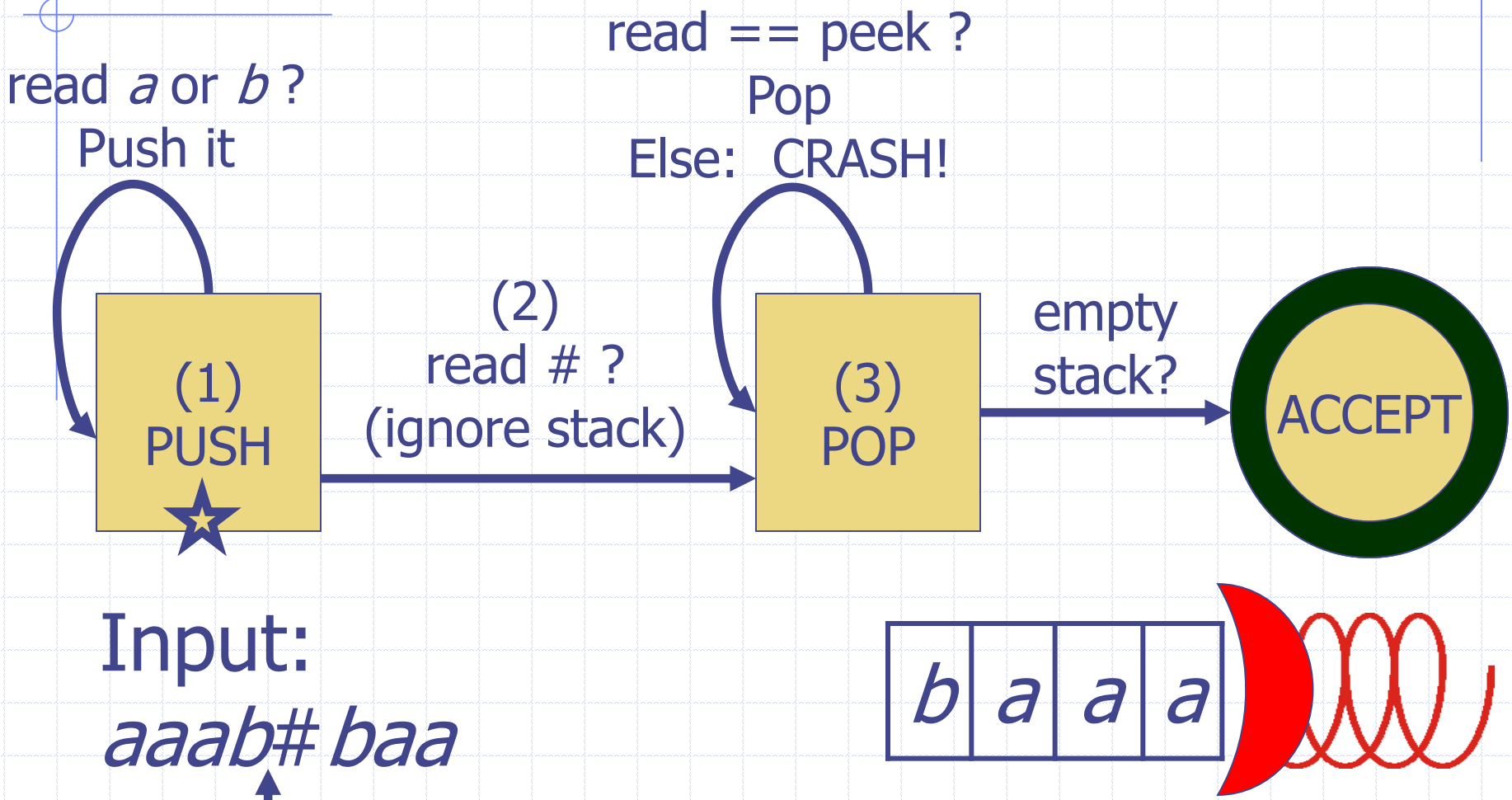
From CFG's to Stack Machines



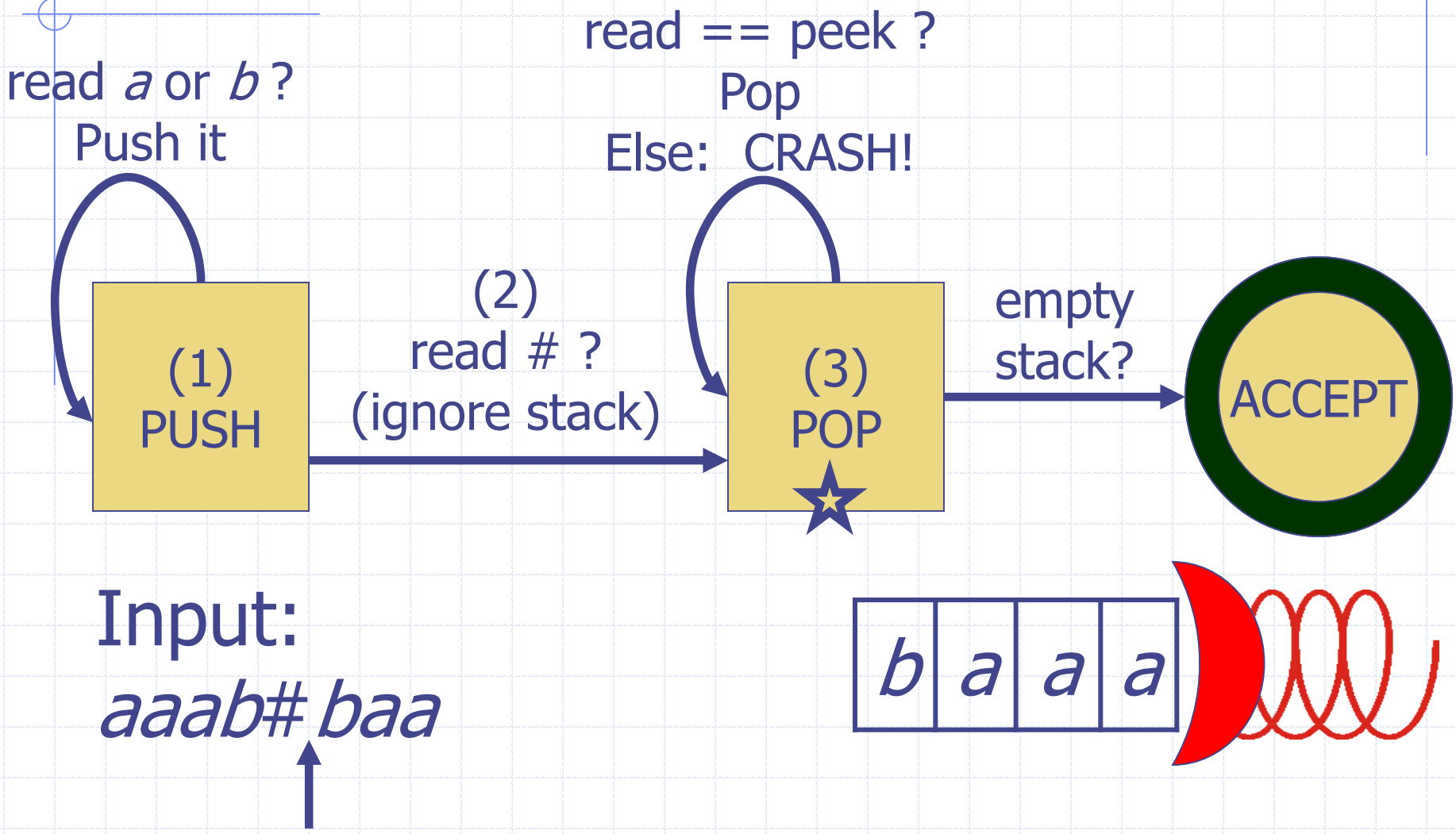
From CFG's to Stack Machines



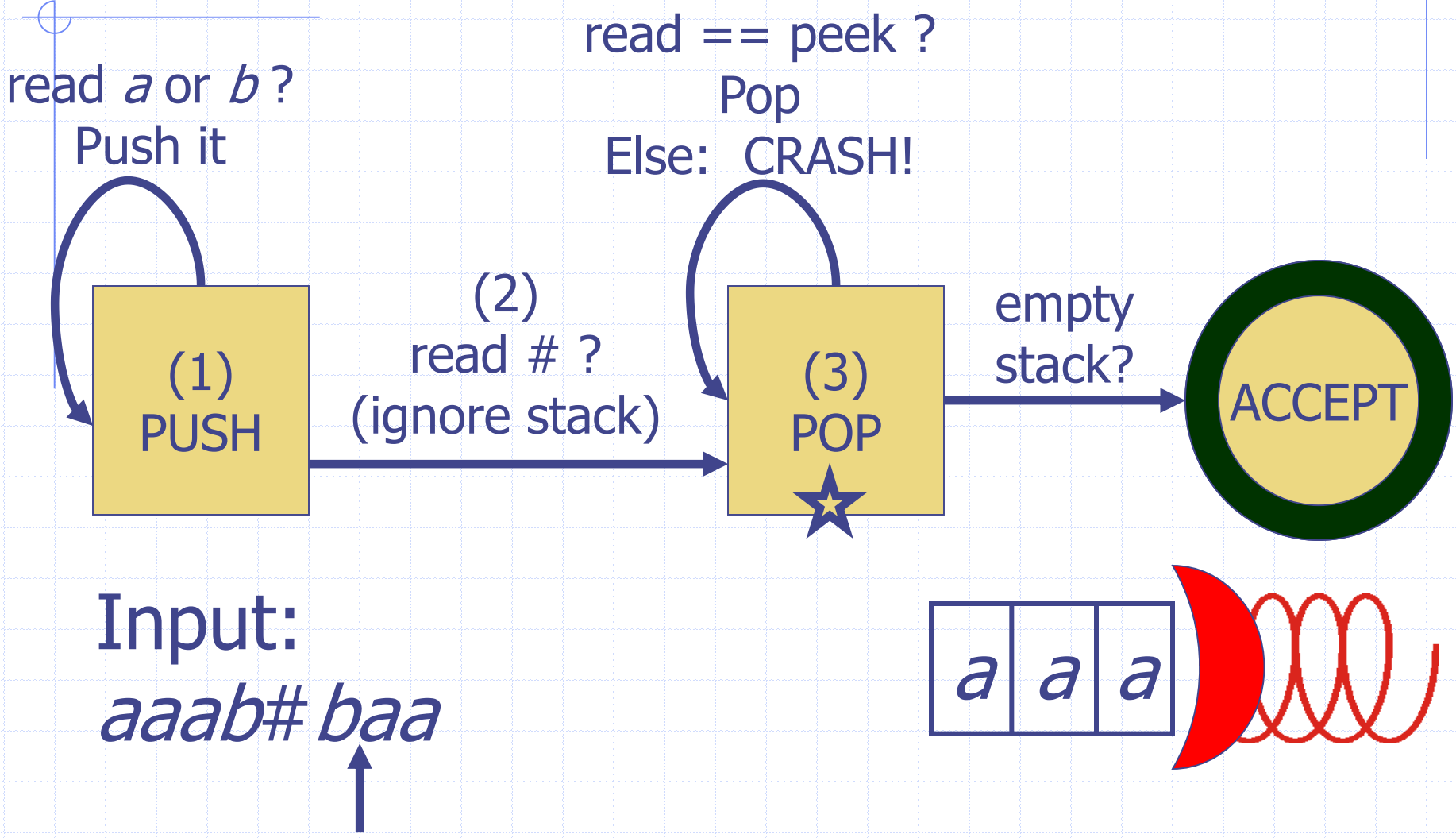
From CFG's to Stack Machines



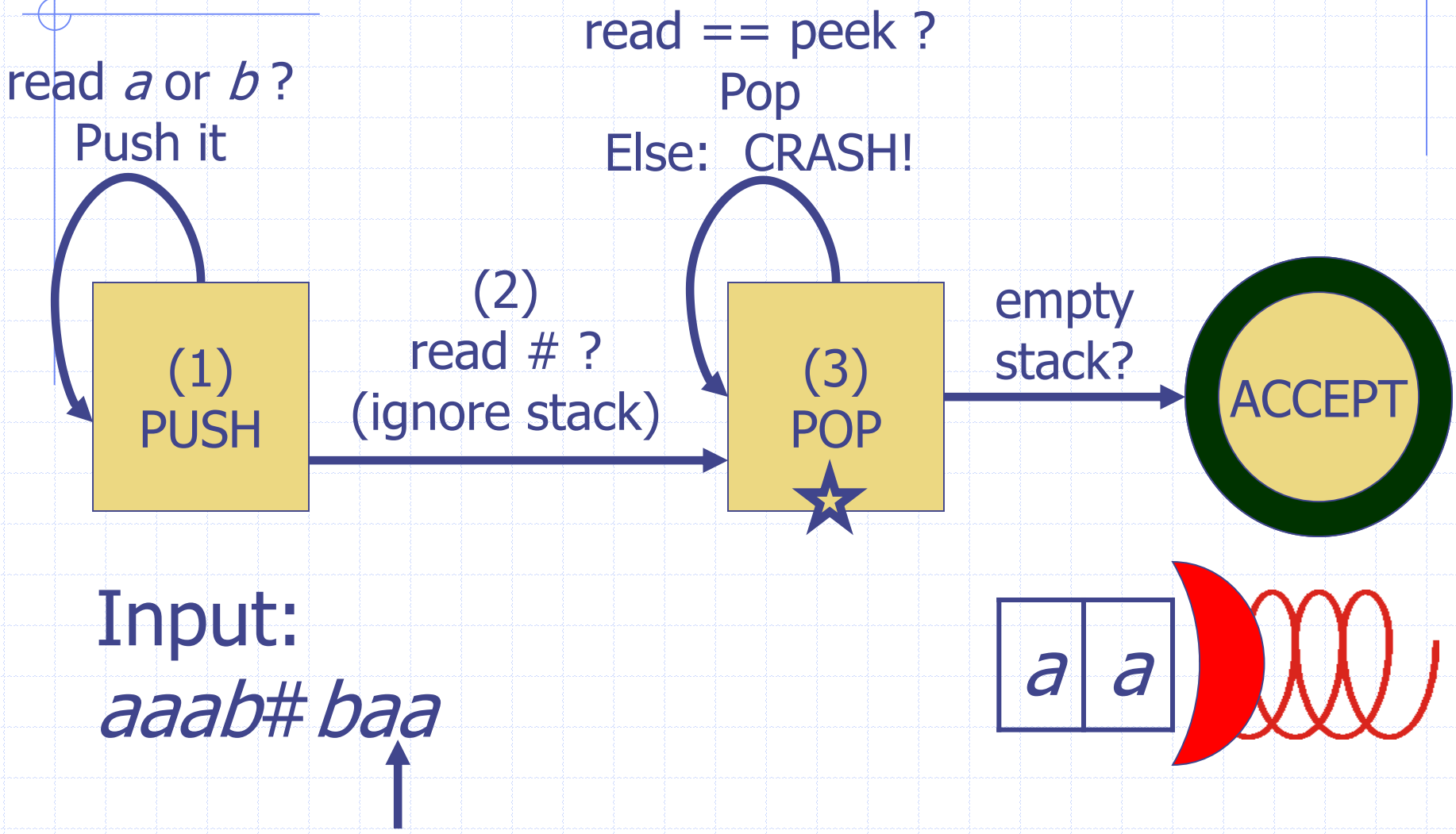
From CFG's to Stack Machines



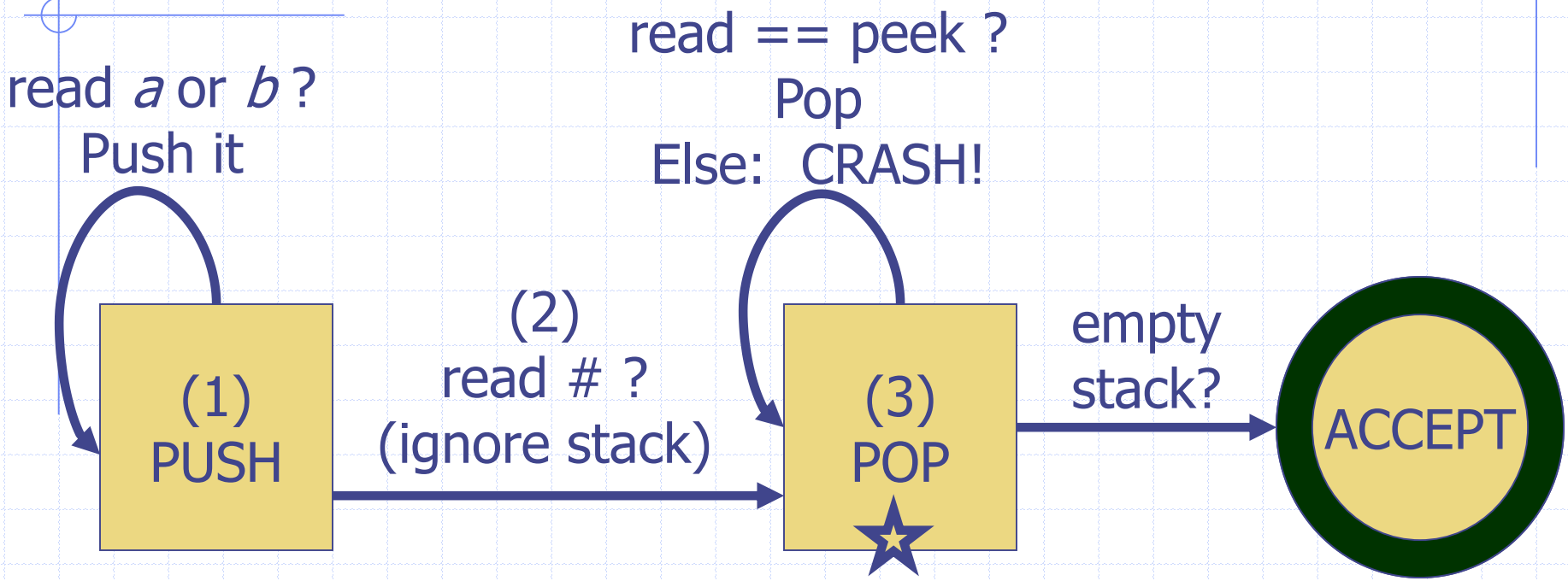
From CFG's to Stack Machines



From CFG's to Stack Machines

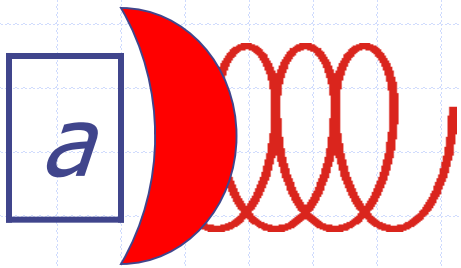


From CFG's to Stack Machines



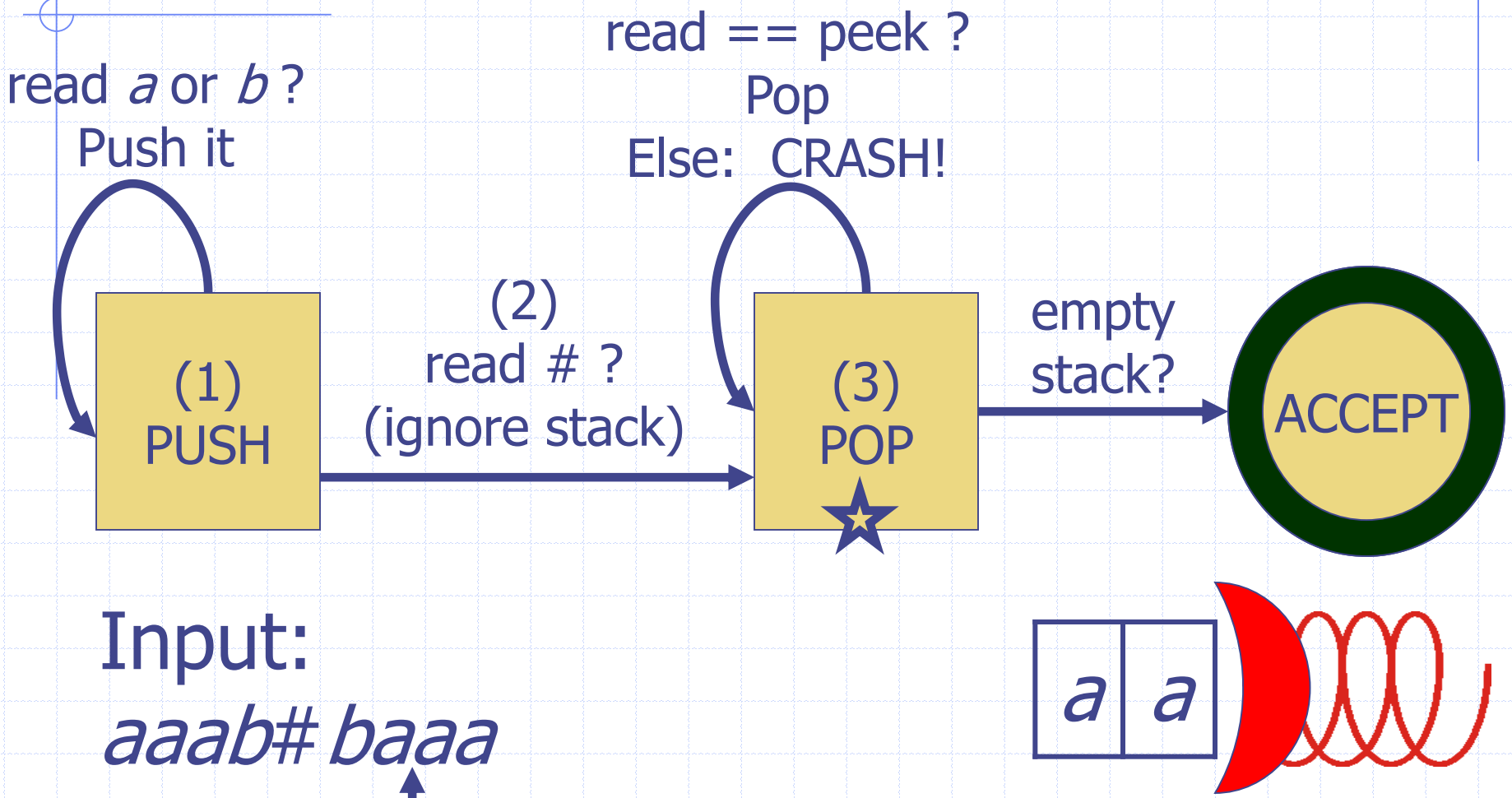
Input:

aaab# baa

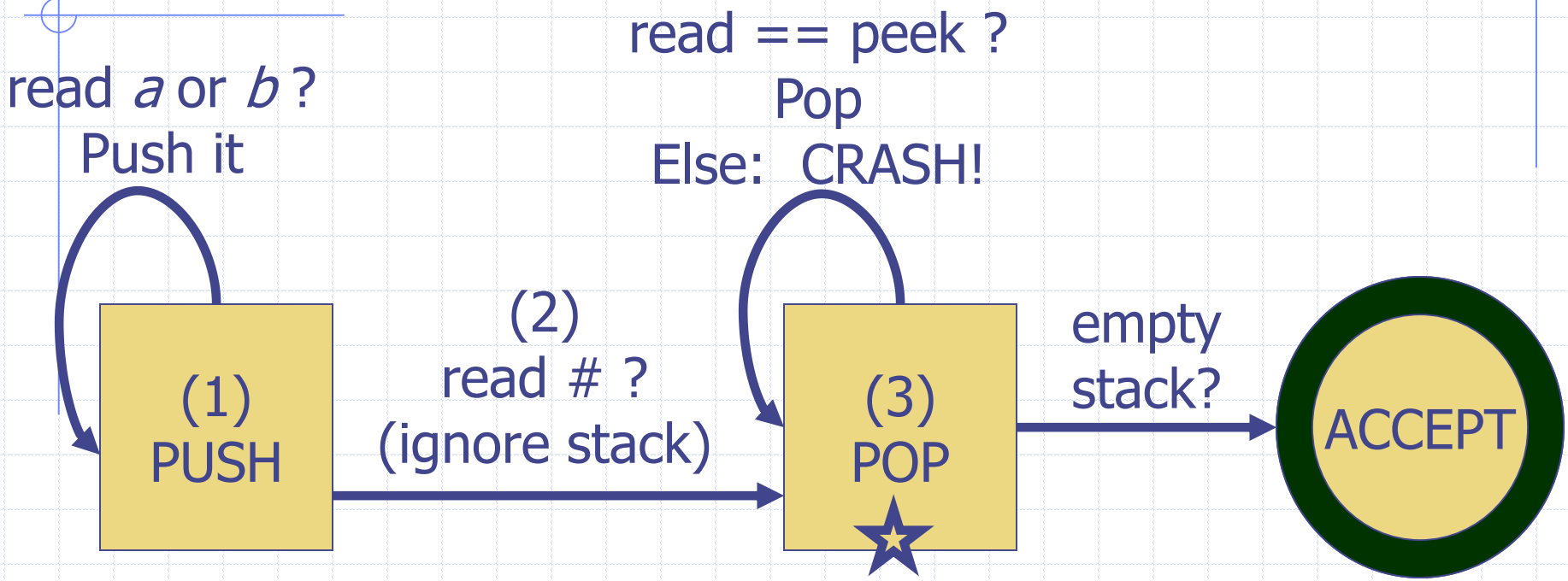


REJECT (nonempty stack)

From CFG's to Stack Machines

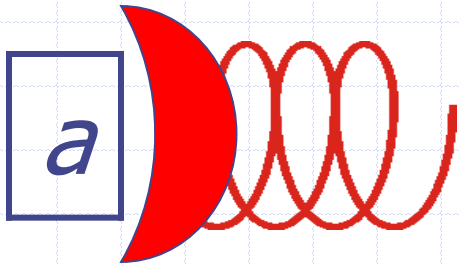


From CFG's to Stack Machines

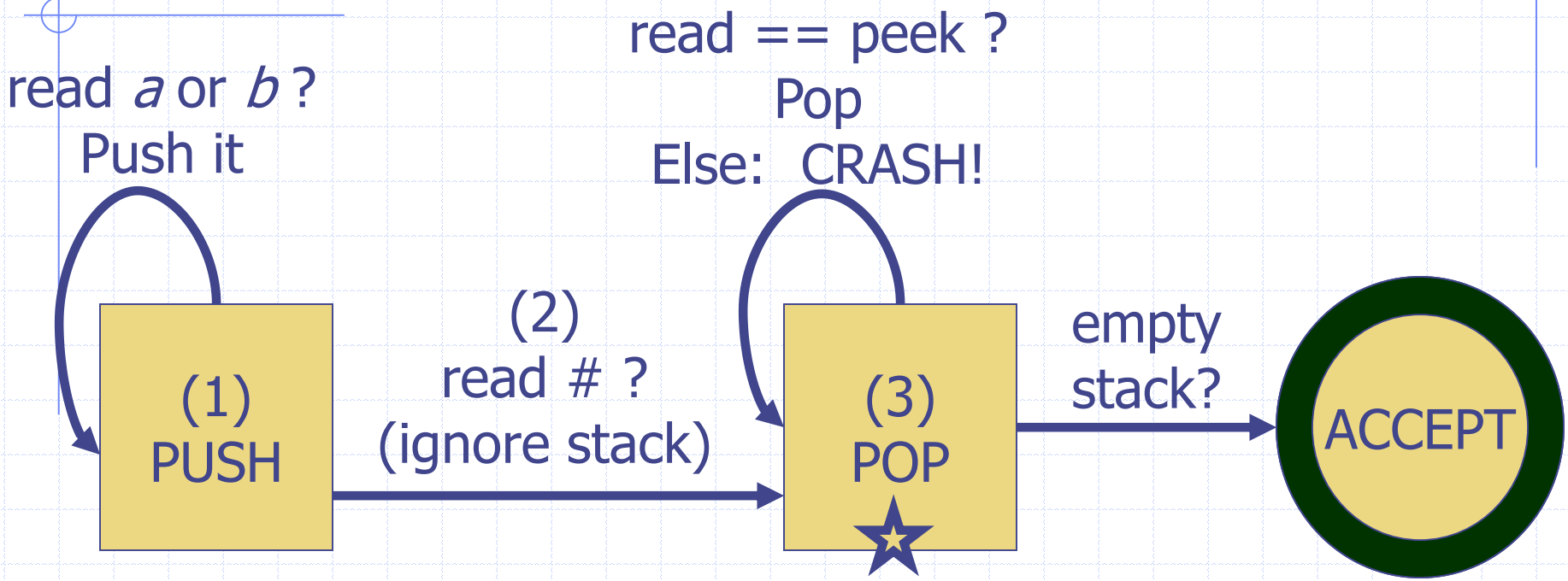


Input:

aaab# baaa



From CFG's to Stack Machines

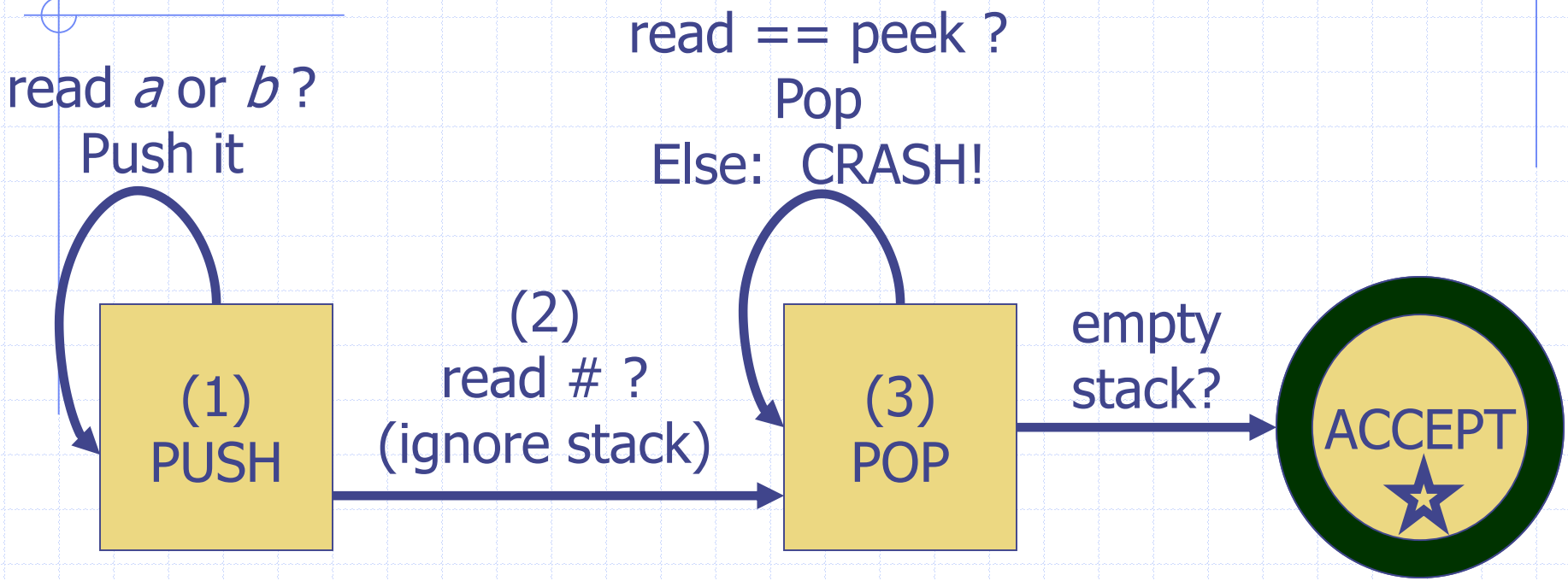


Input:

aaab# baaa

Pause input

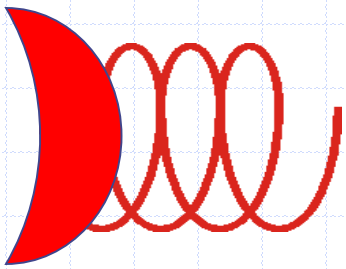
From CFG's to Stack Machines



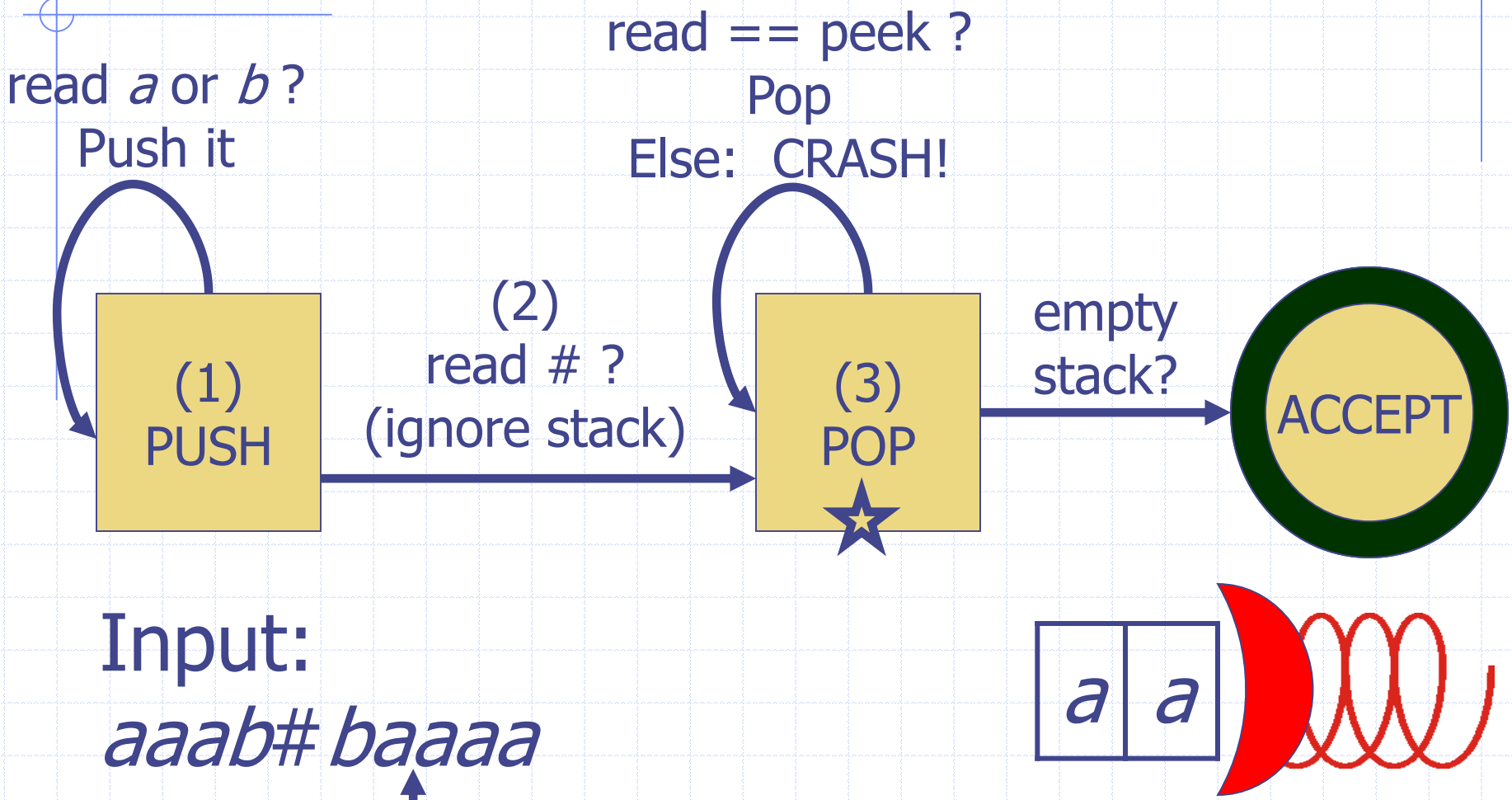
Input:

aaab# baaa

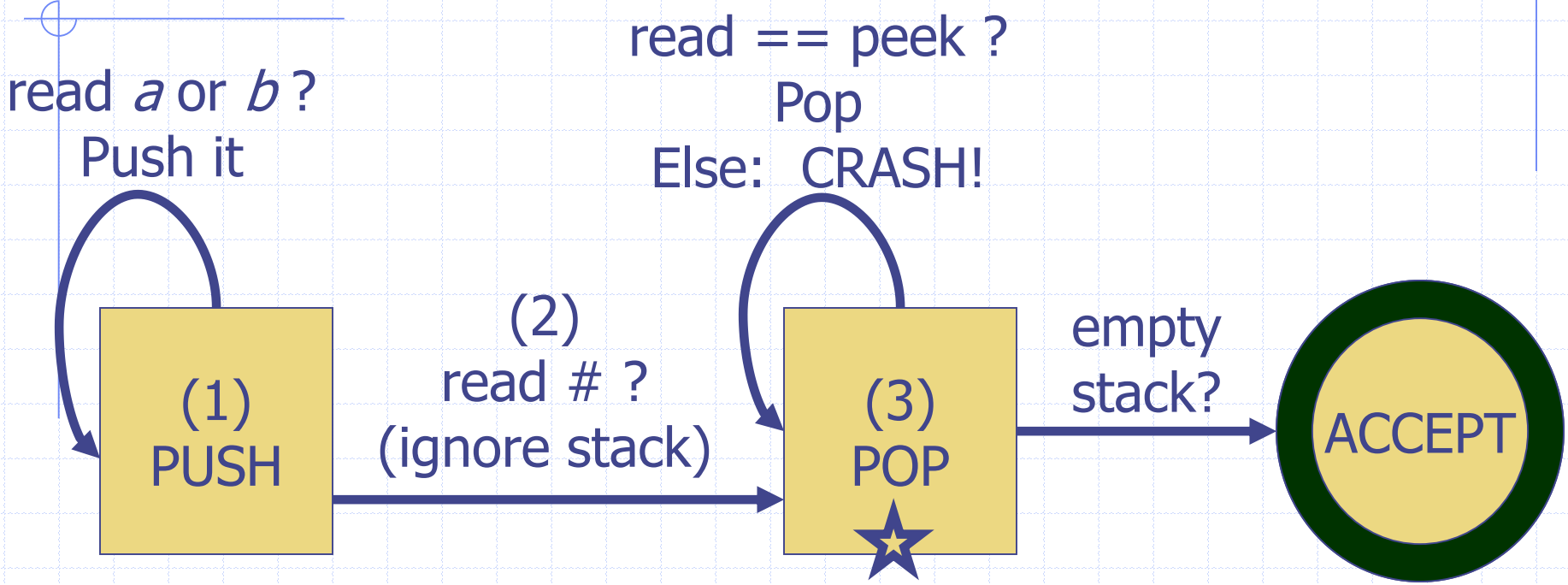
↑ *ACCEPT*



From CFG's to Stack Machines

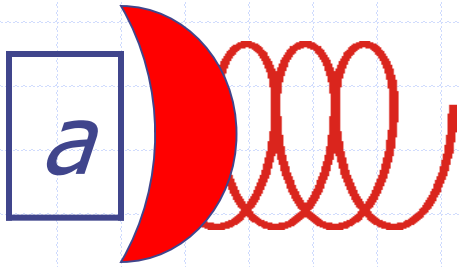


From CFG's to Stack Machines

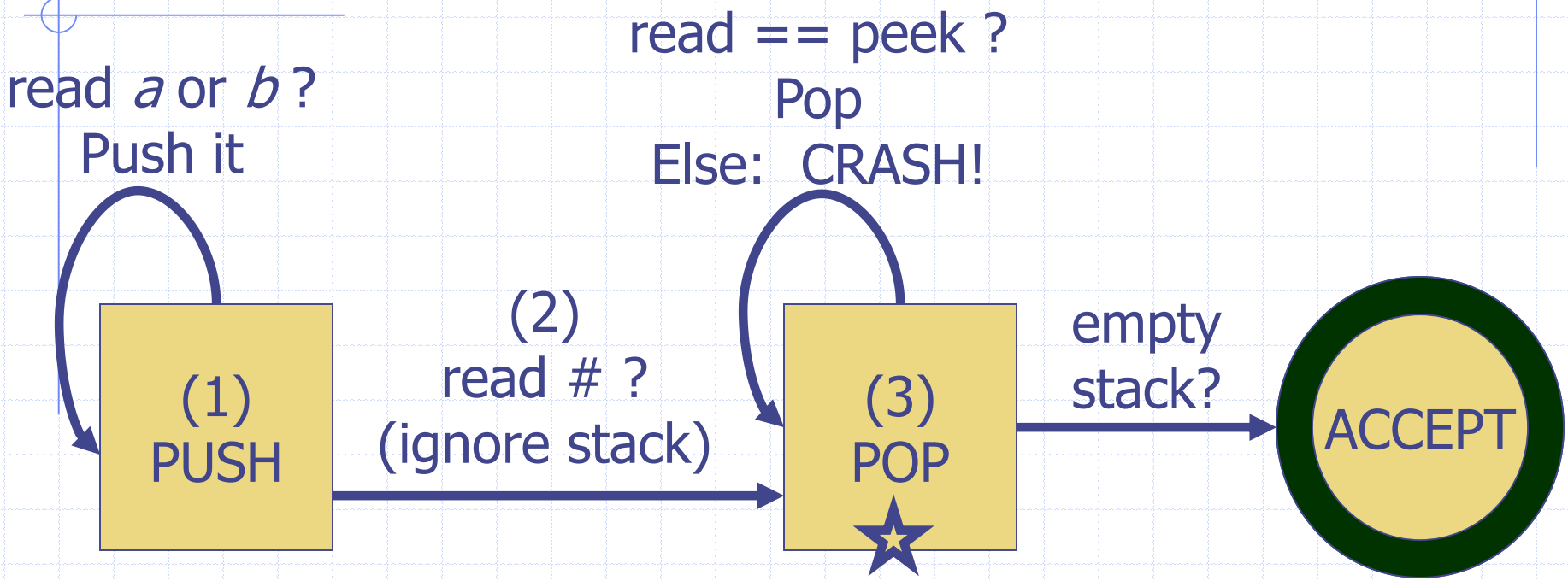


Input:

aaab# baaaa



From CFG's to Stack Machines

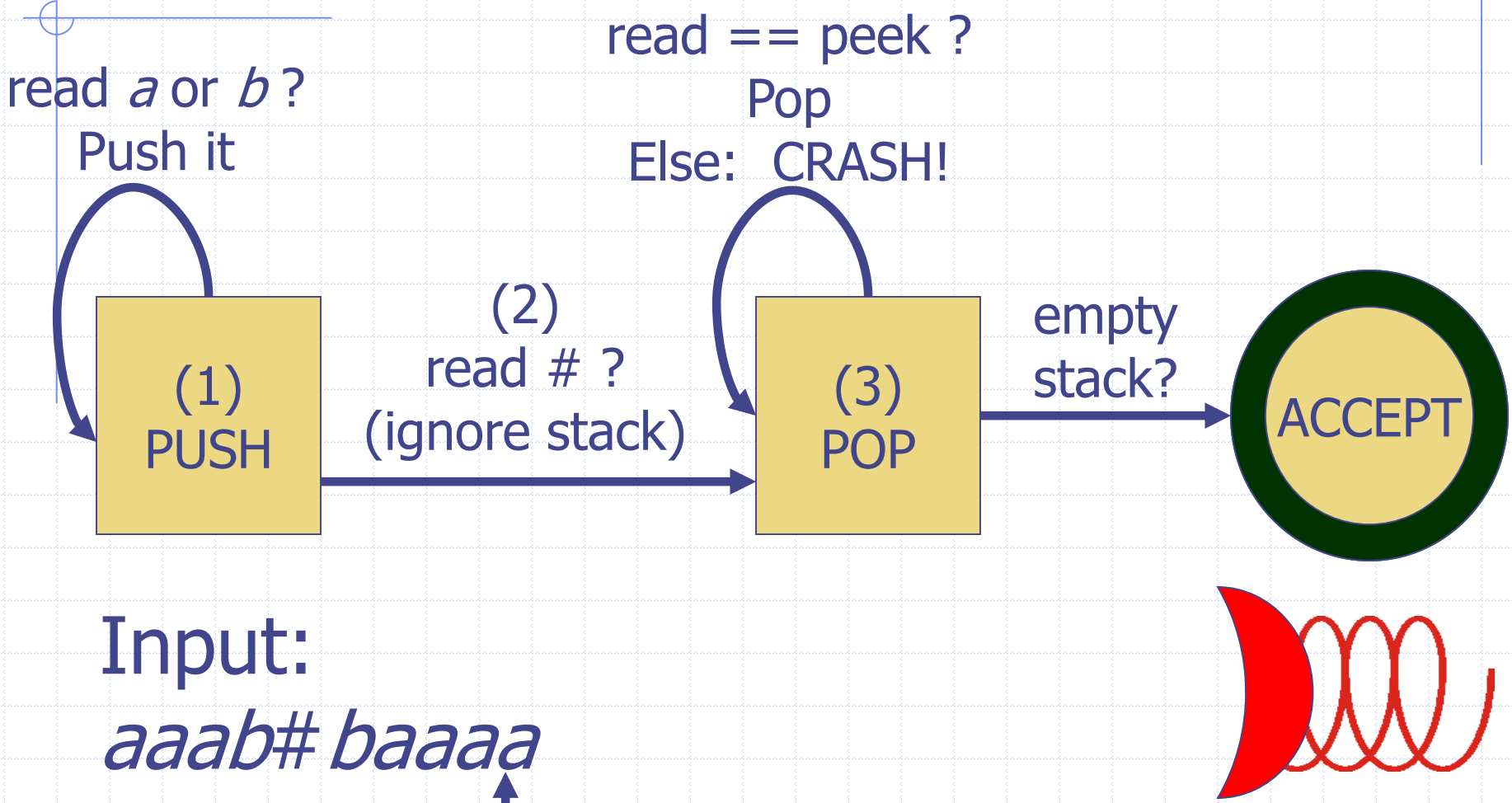


Input:

aaab# baaaa

Pause input

From CFG's to Stack Machines



Input:

aaab# baaaa

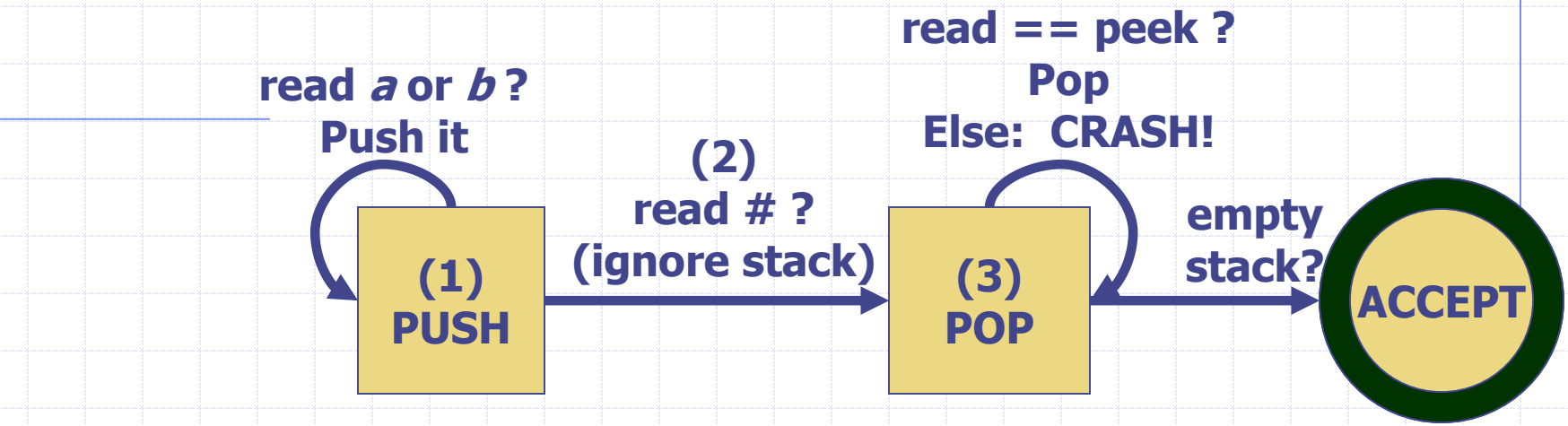
CRASH

PDA's Sipser Model

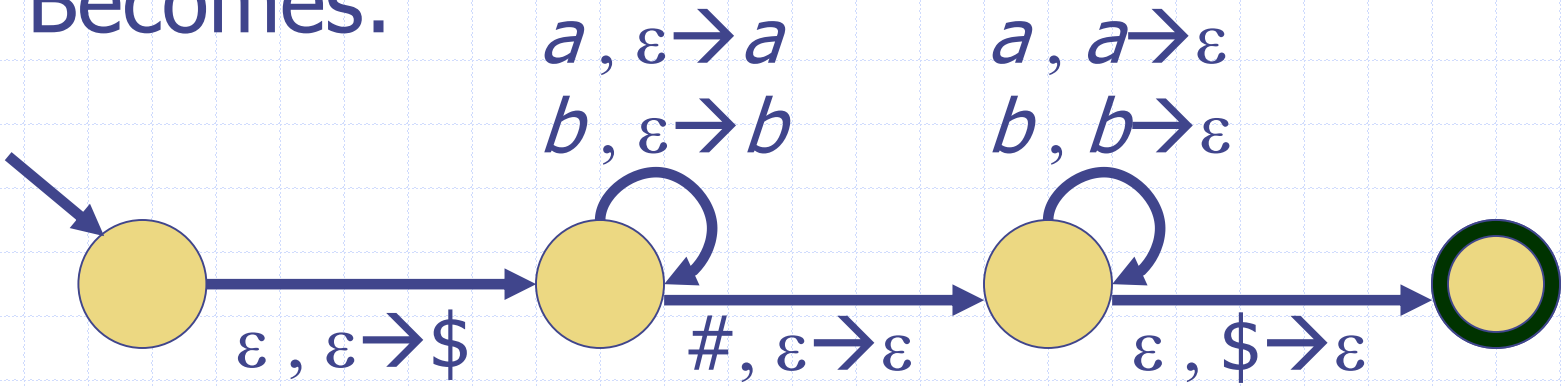
To aid analysis, theoretical stack machines restrict the allowable operations. Each text-book author has his own version. Sipser's machines are especially simple:

- Push/Pop rolled into a single operation: *replace top stack symbol*
- No intrinsic way to test for empty stack
- Epsilon's used to increase functionality, result in default *nondeterministic* machines.

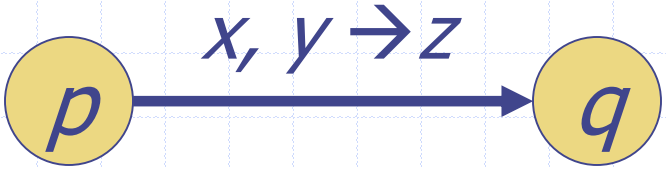
Sipser's Version



Becomes:



Sipser's Version

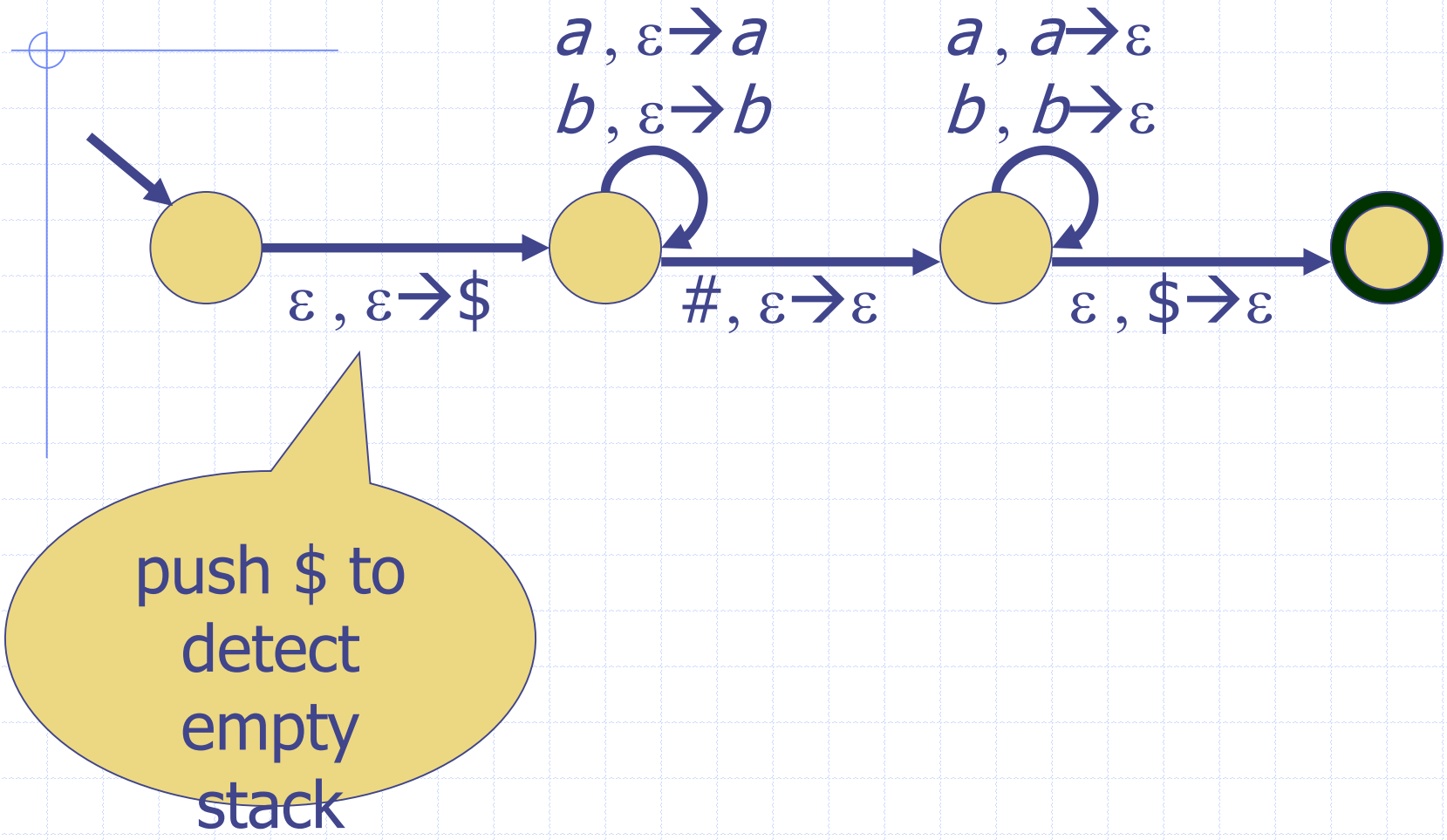


Meaning of labeling convention:

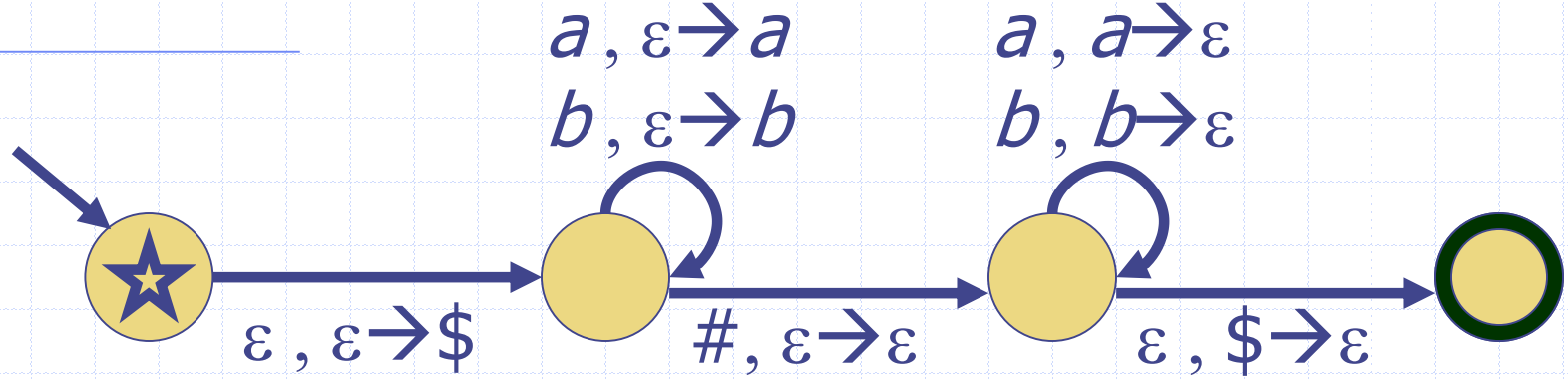
If at p **and** next input x **and** top stack y, then go to q **and** replace y by z on stack.

- $x = \varepsilon$: ignore input, don't read
- $y = \varepsilon$: ignore top of stack and push z
- $z = \varepsilon$: pop y

Sipser's Version

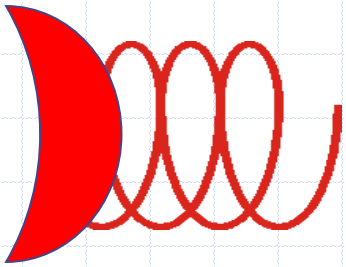


Sipser's Version

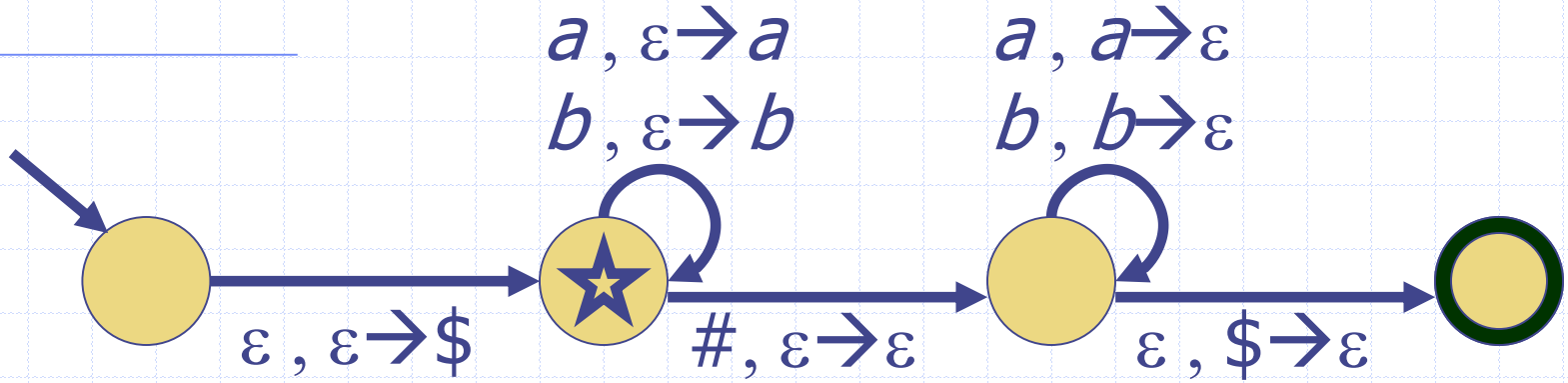


Input:

aaab# baaa

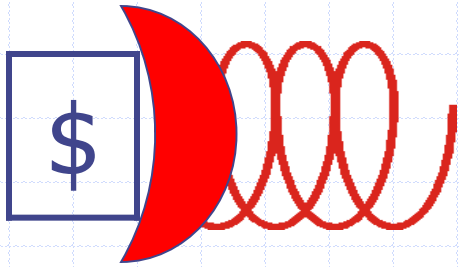


Sipser's Version

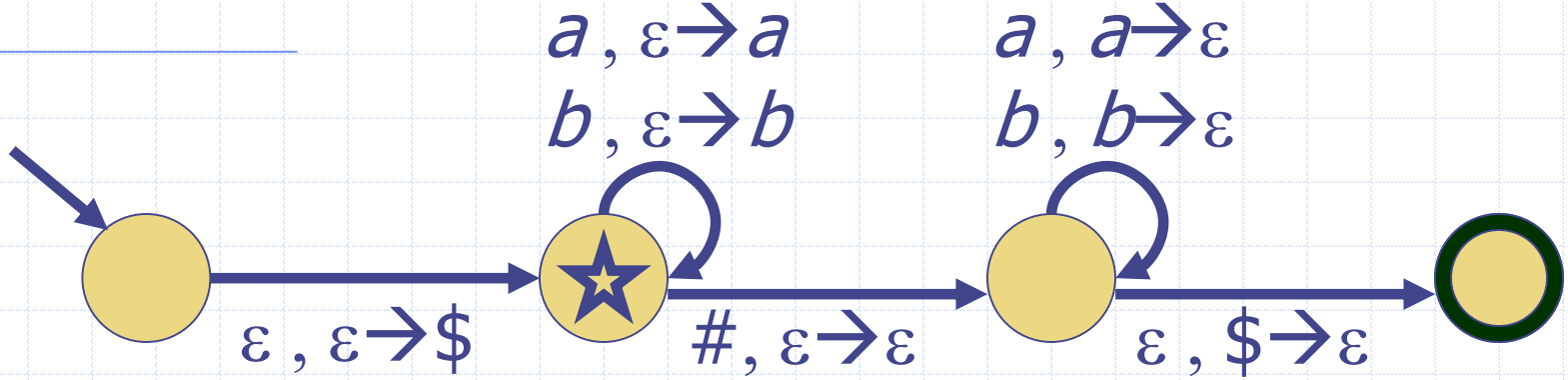


Input:

aaab# baaa

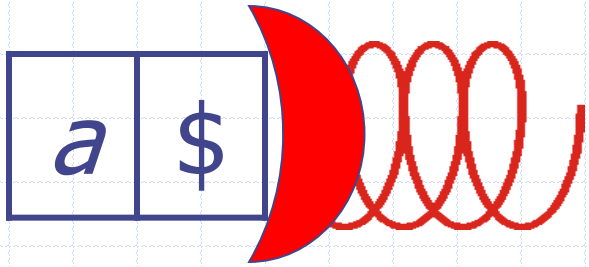


Sipser's Version

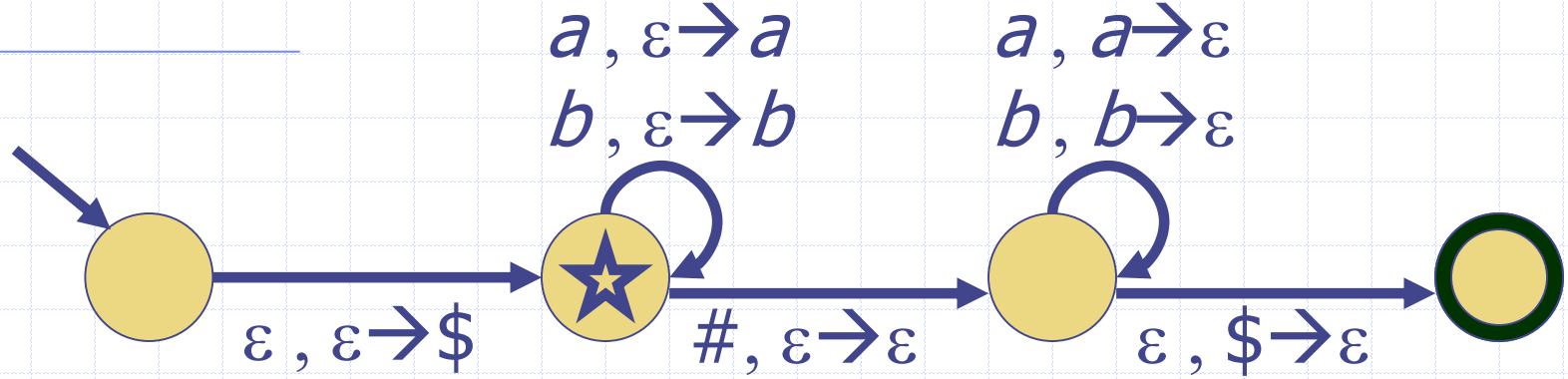


Input:

aaab# baaa

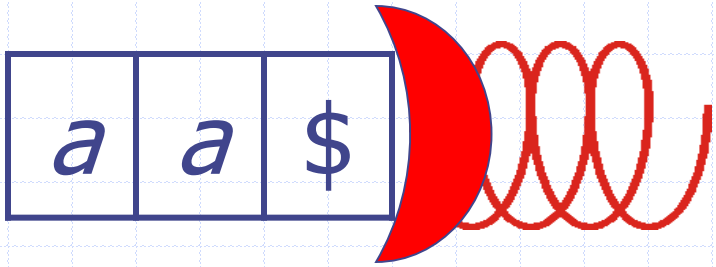


Sipser's Version

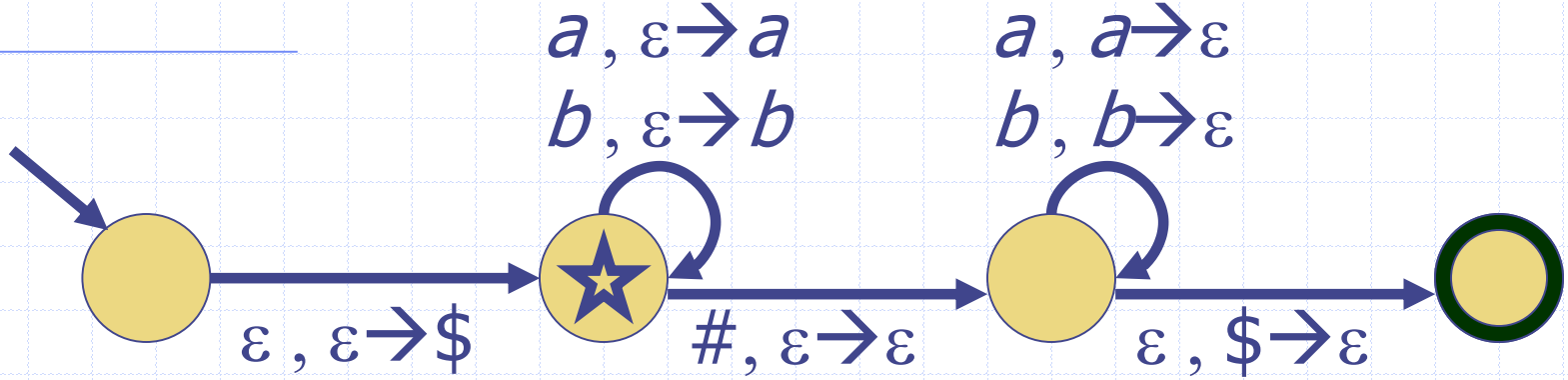


Input:

aaab# baaa

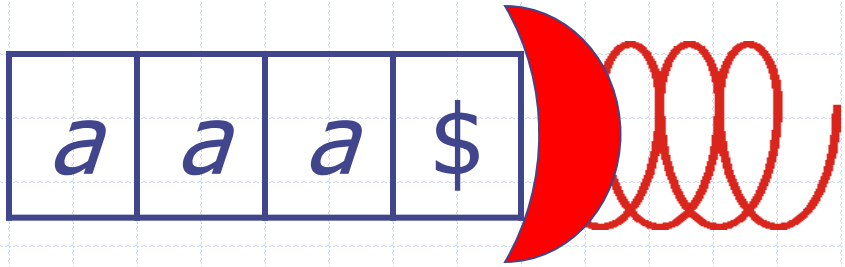


Sipser's Version

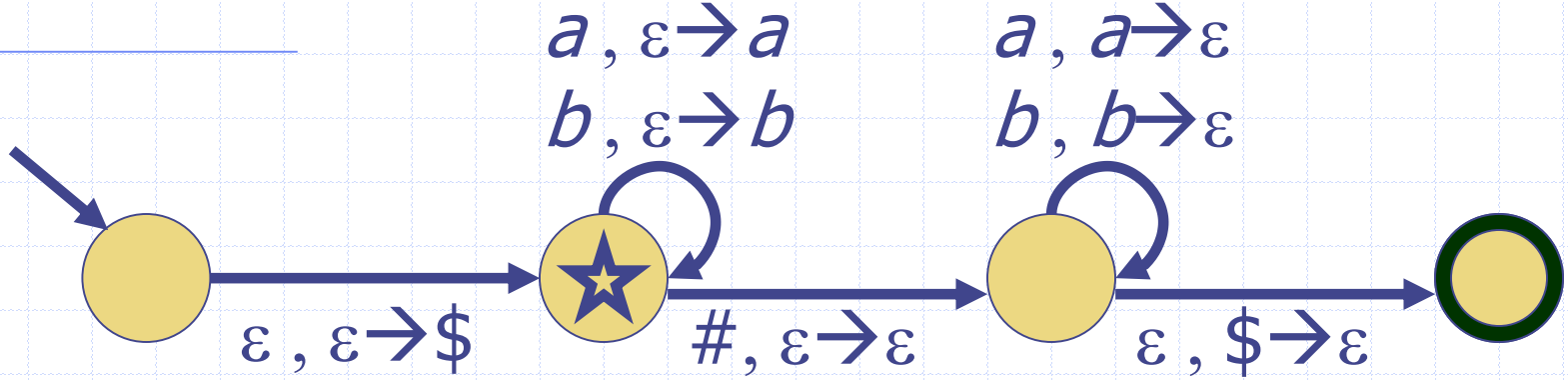


Input:

aaab# baaa

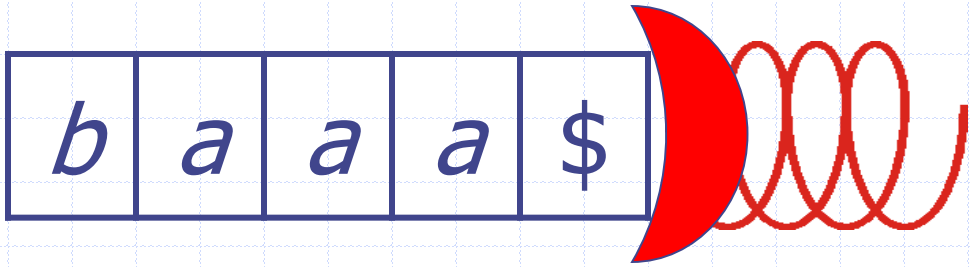


Sipser's Version

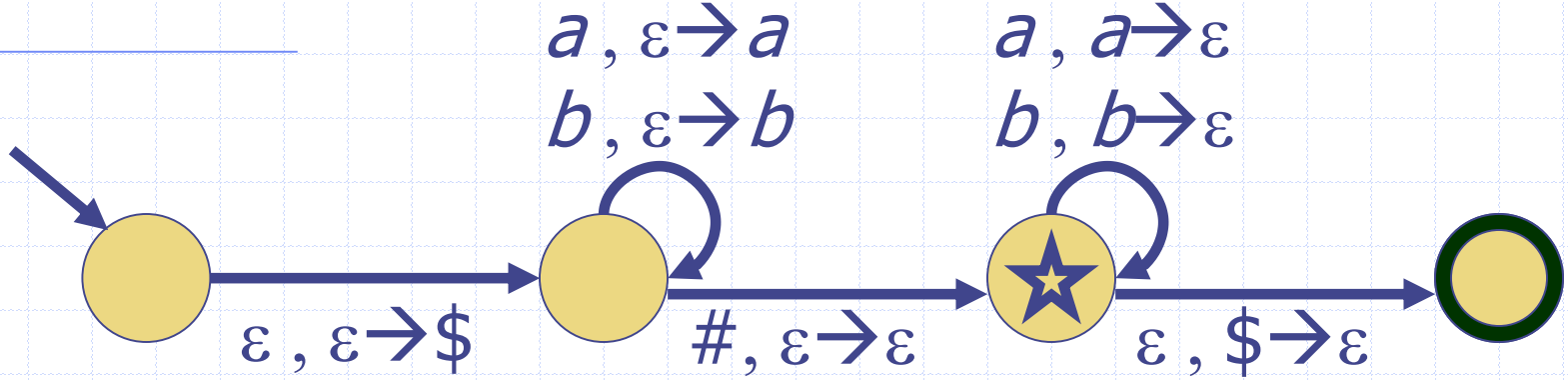


Input:

aaab# baaa

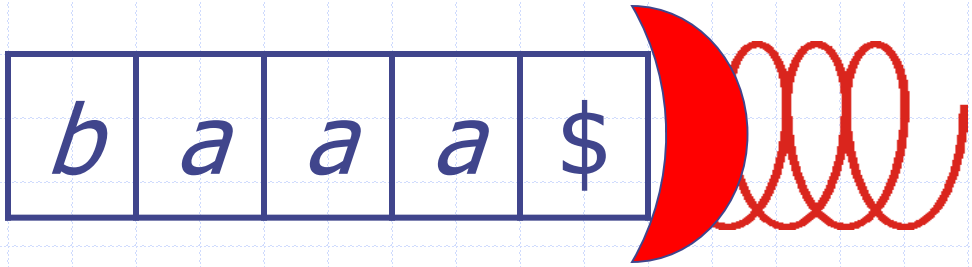


Sipser's Version

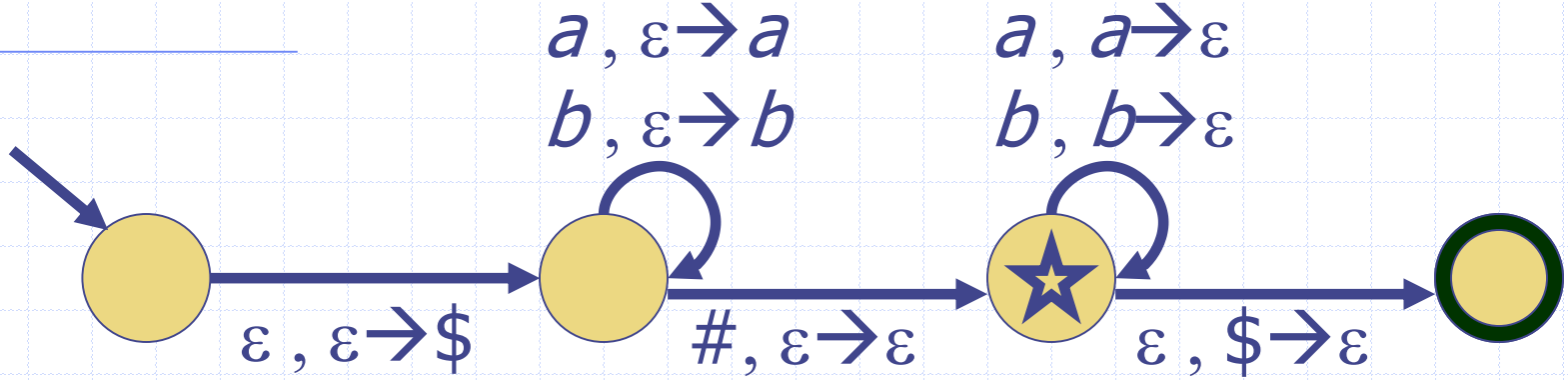


Input:

aaab# baaa

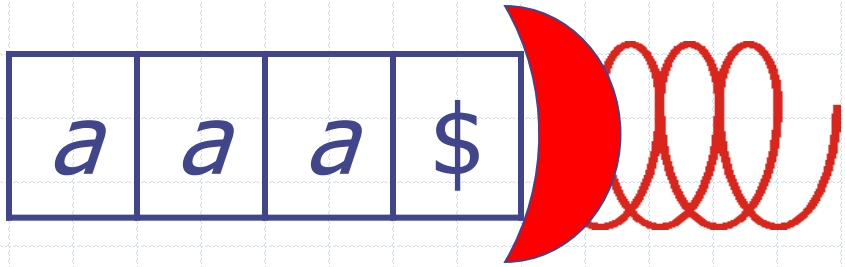


Sipser's Version

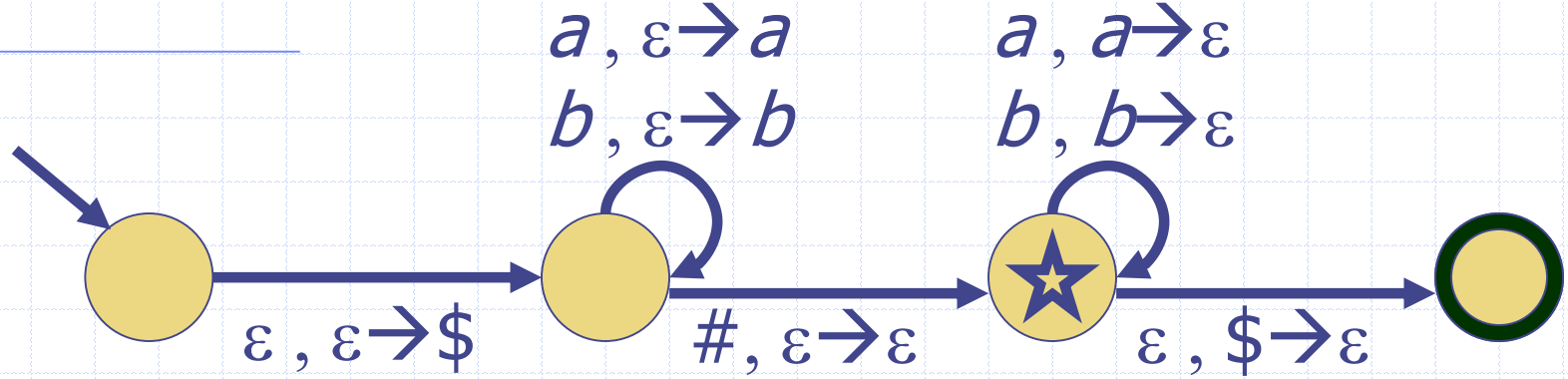


Input:

aaab# baaa

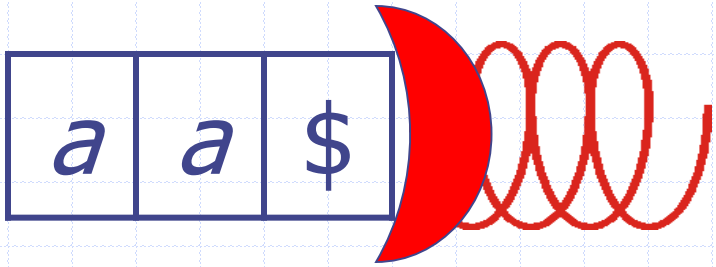


Sipser's Version

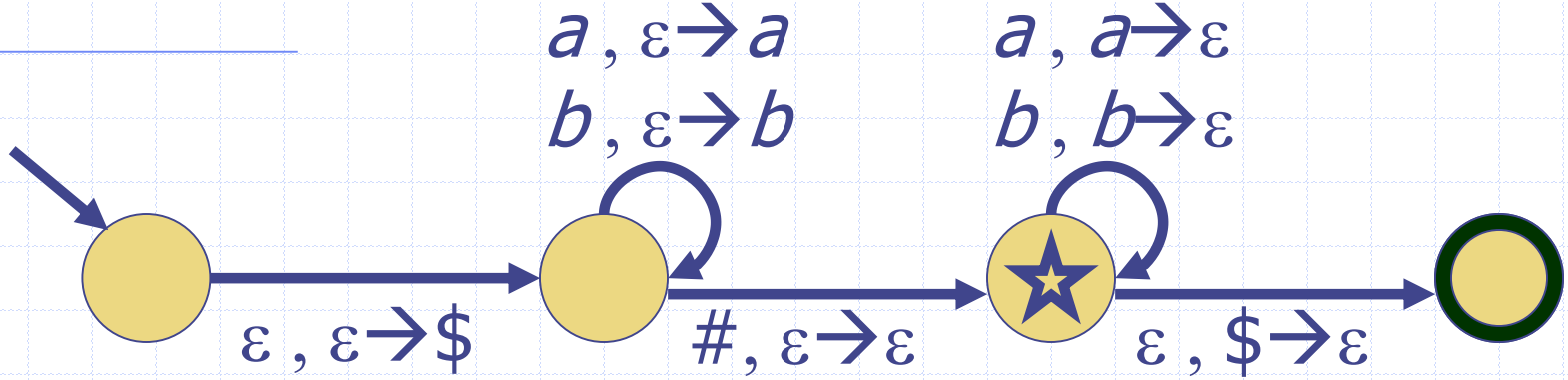


Input:

aaab# baaa

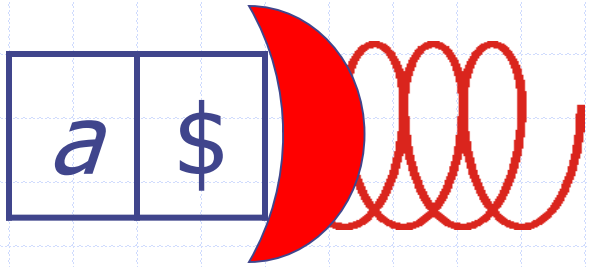


Sipser's Version

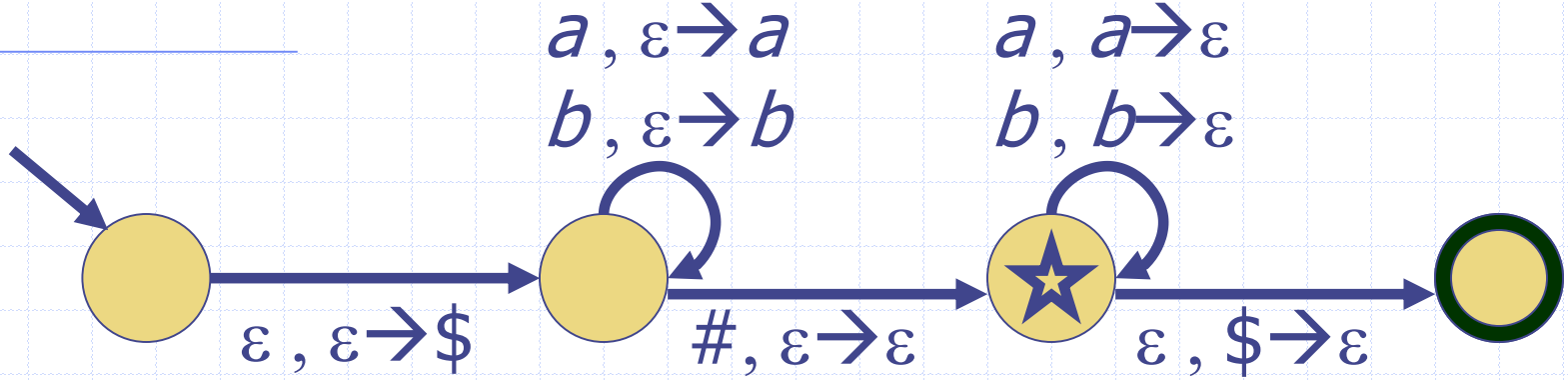


Input:

aaab# baaa

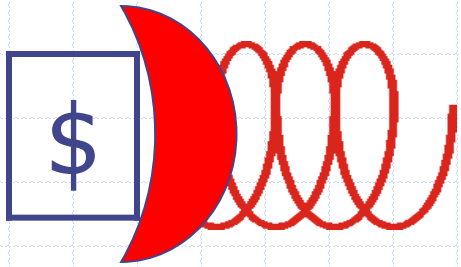


Sipser's Version

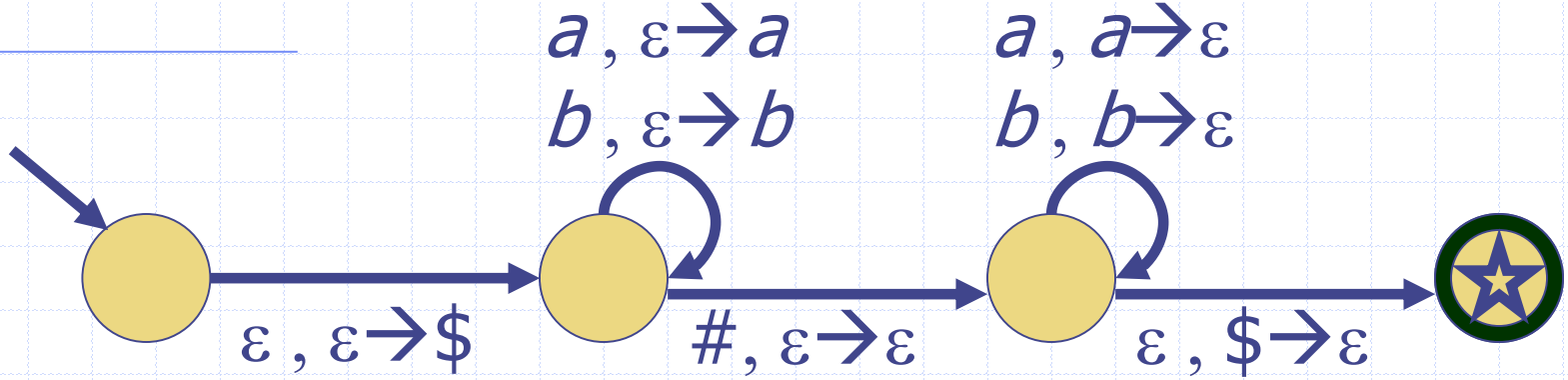


Input:

aaab# baaa



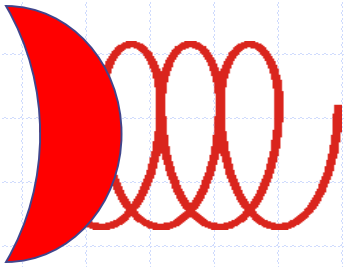
Sipser's Version



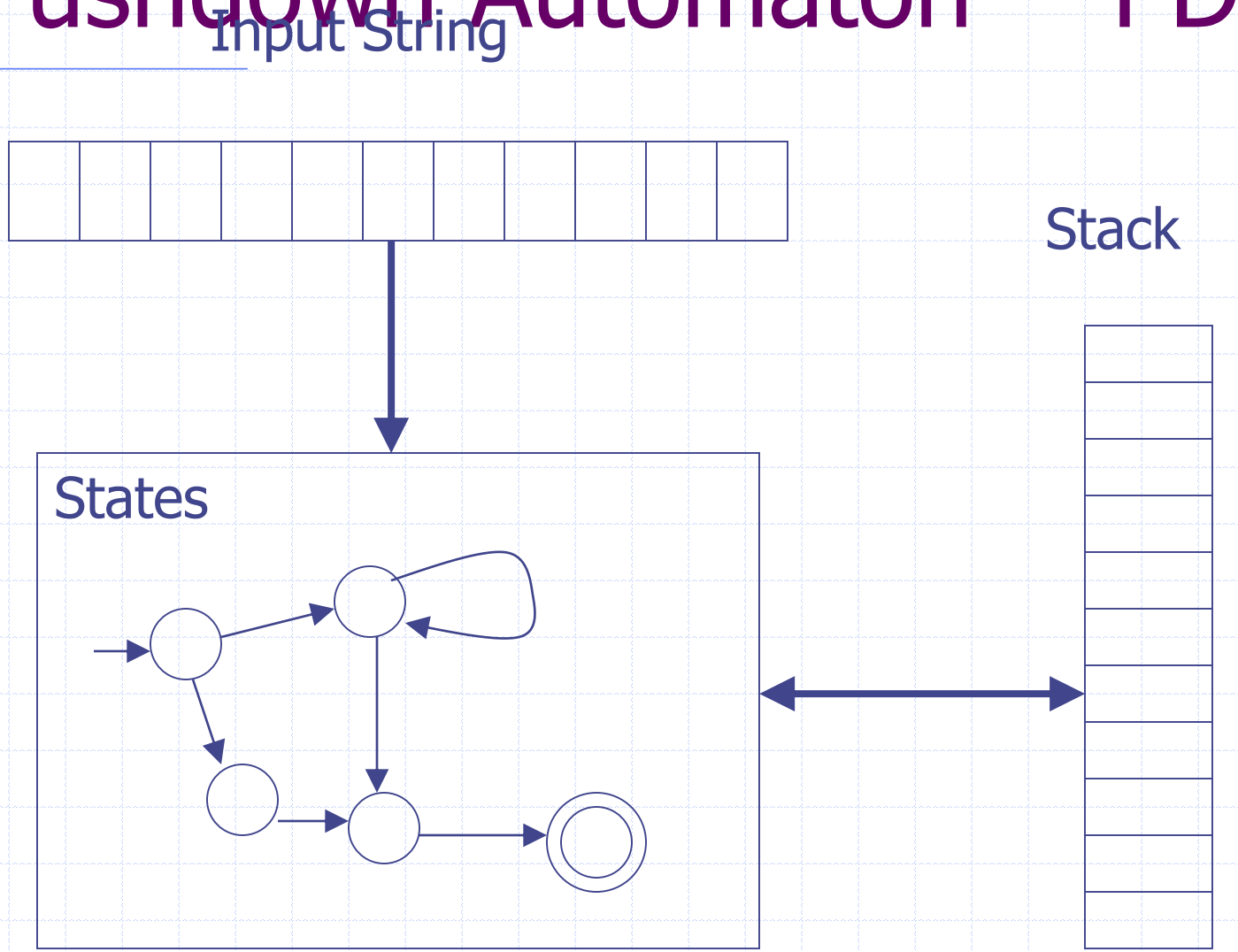
Input:

aaab# baaa

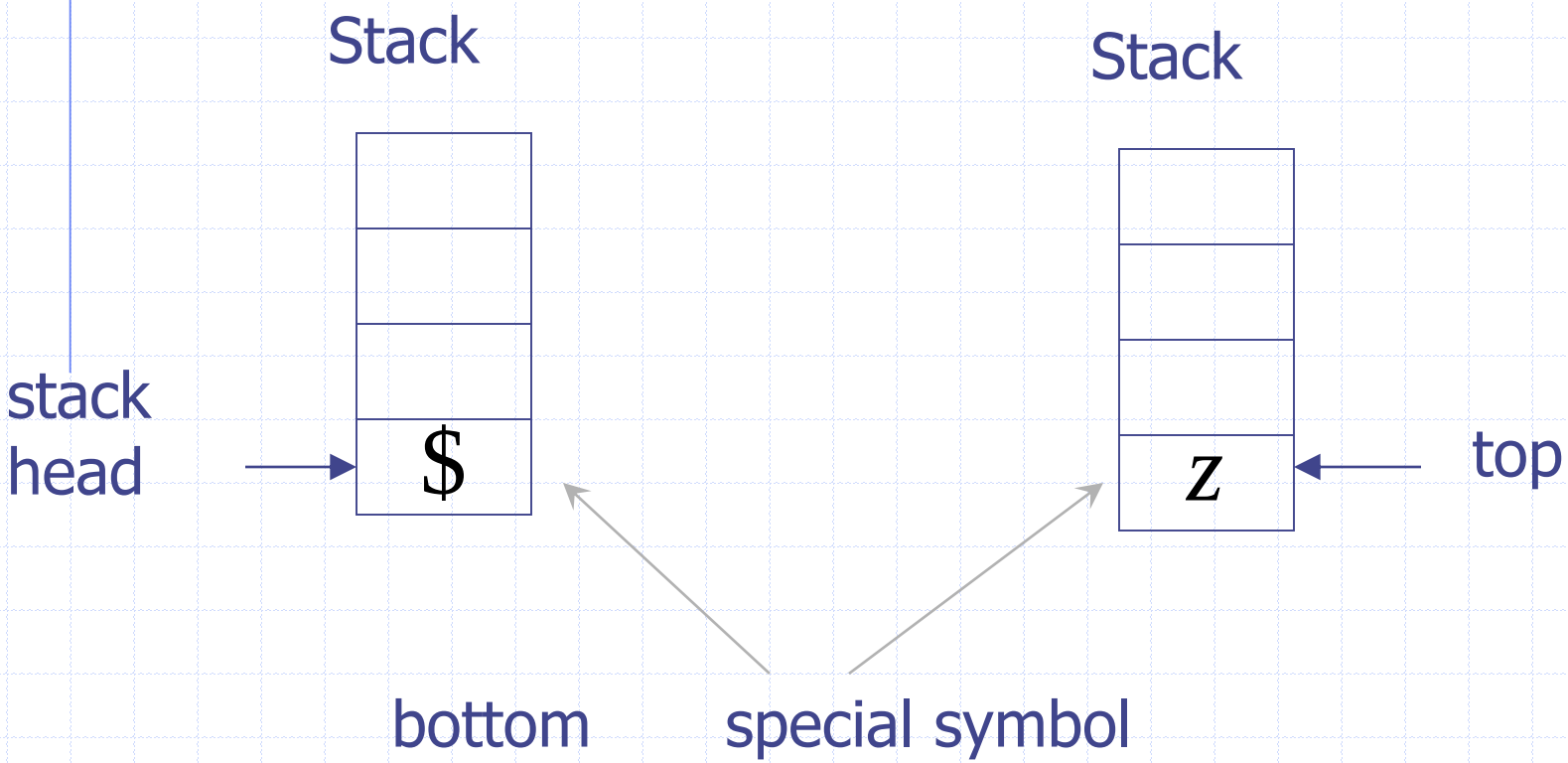
ACCEPT!



Pushdown Automaton -- PDA



Initial Stack Symbol



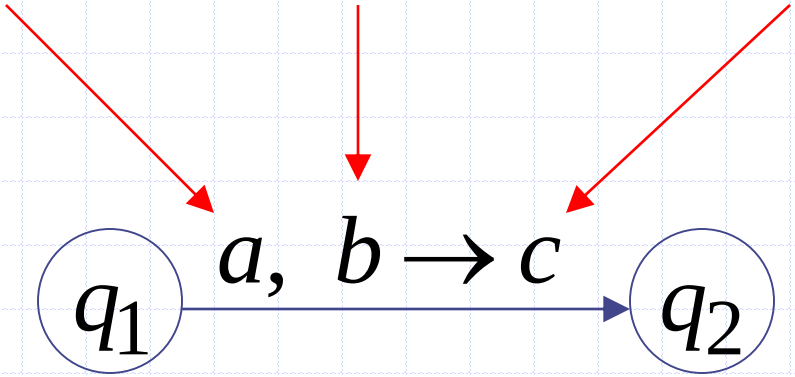
Appears at time 0

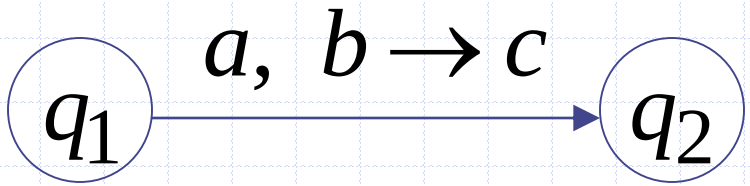
The States

Input symbol

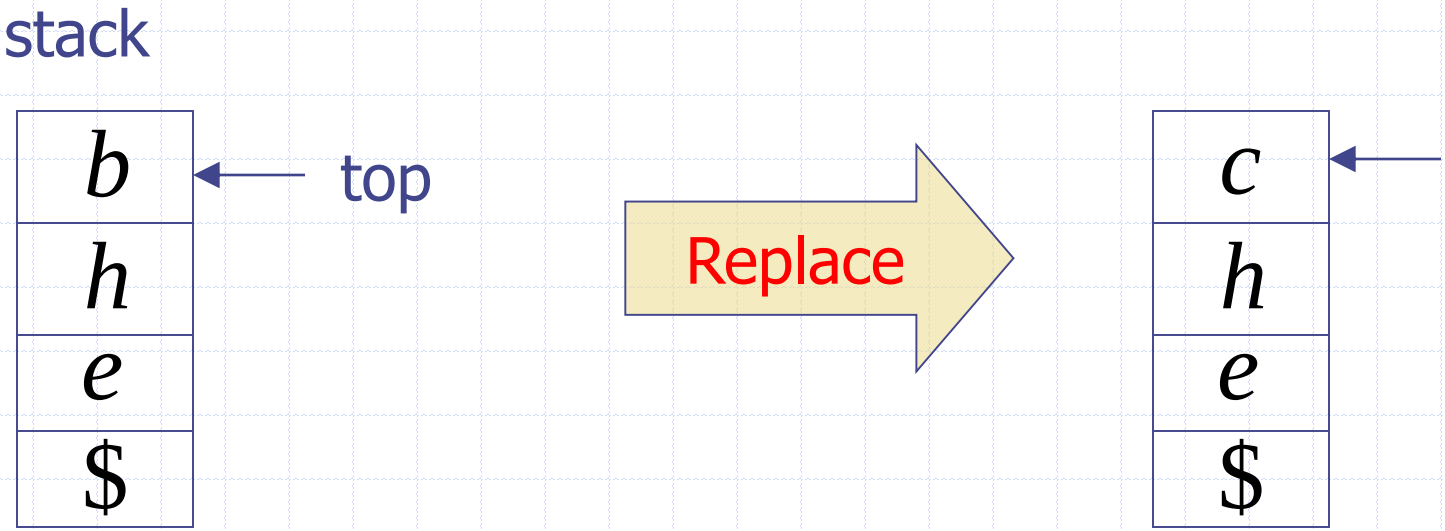
Pop symbol

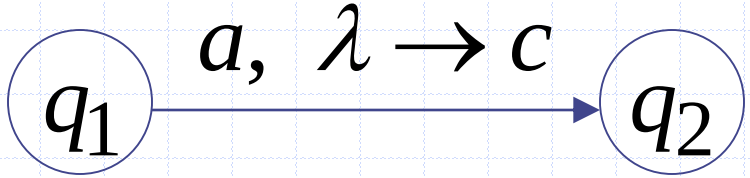
Push symbol



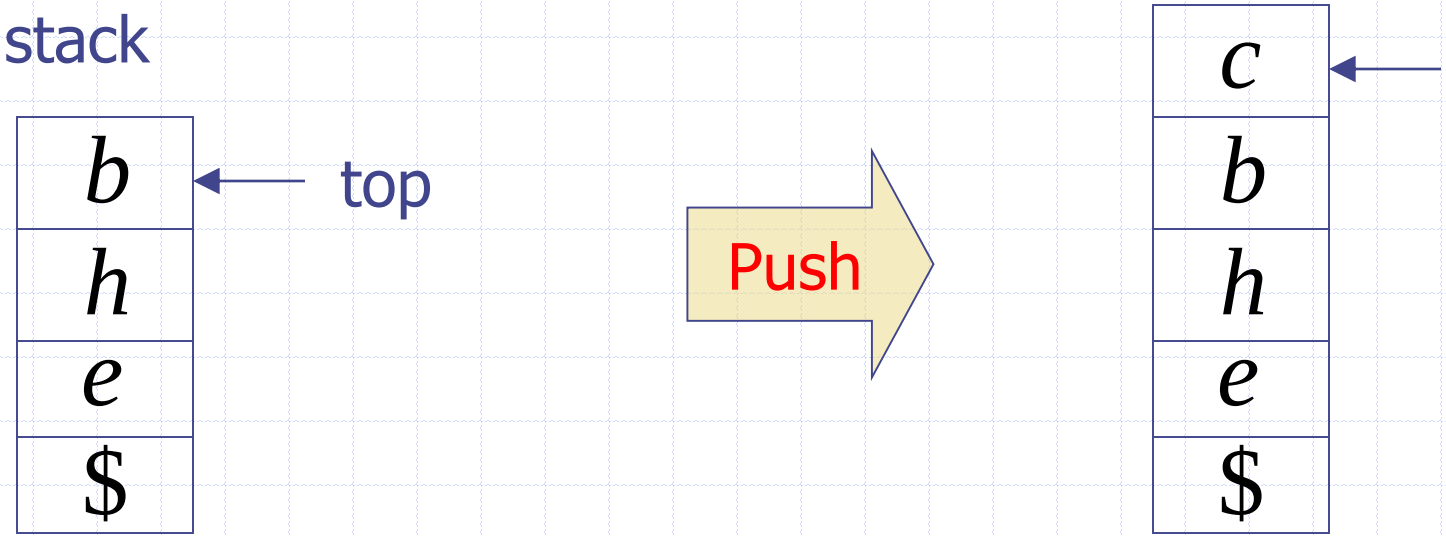

$a,\ b \to c$
q_1
q_2

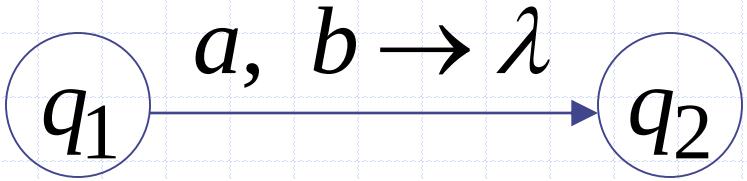

input
...
a
...
...
a
...

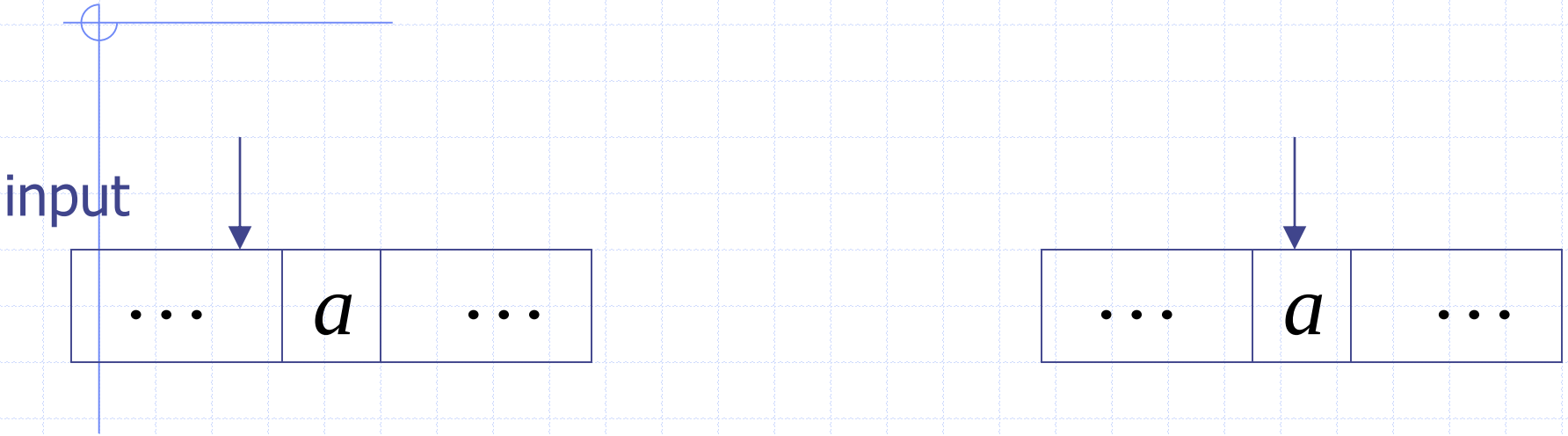

stack
b
top
h
e
$
Replace
c
h
e
$

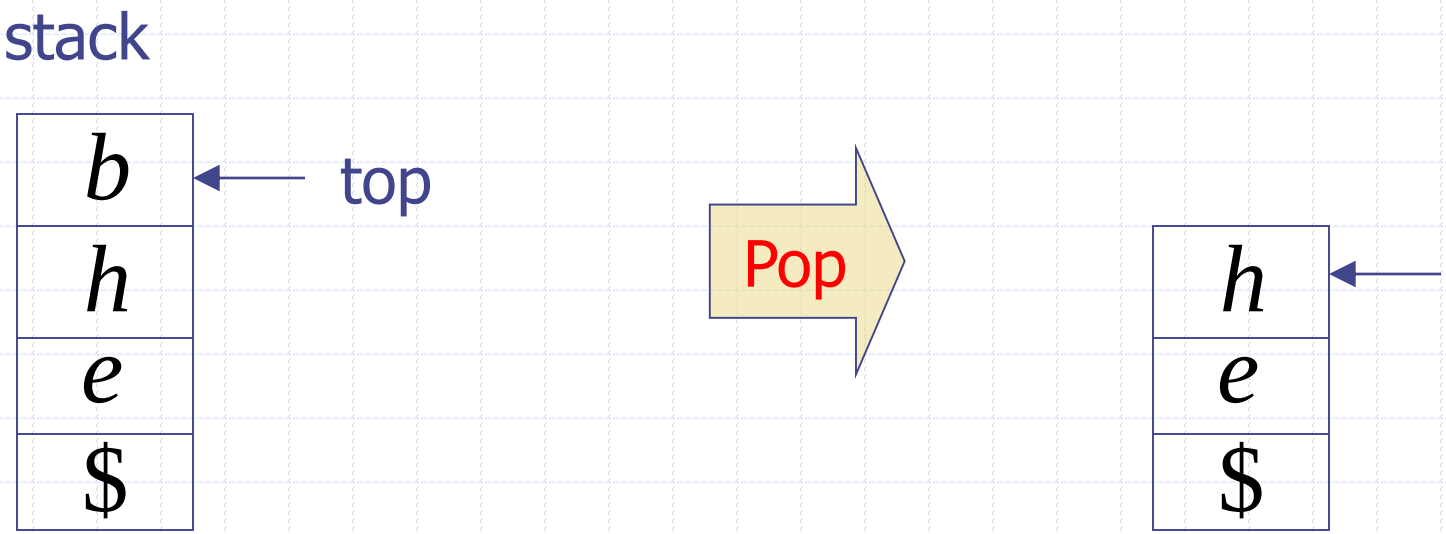

$a,\ \lambda \rightarrow c$
q_1
q_2

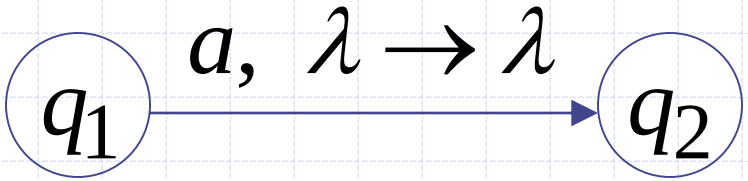

input
$\ldots$
a
$\ldots$
$\ldots$
a
$\ldots$


stack
b
top
h
e
$\$$
Push
c
b
h
e
$\$$

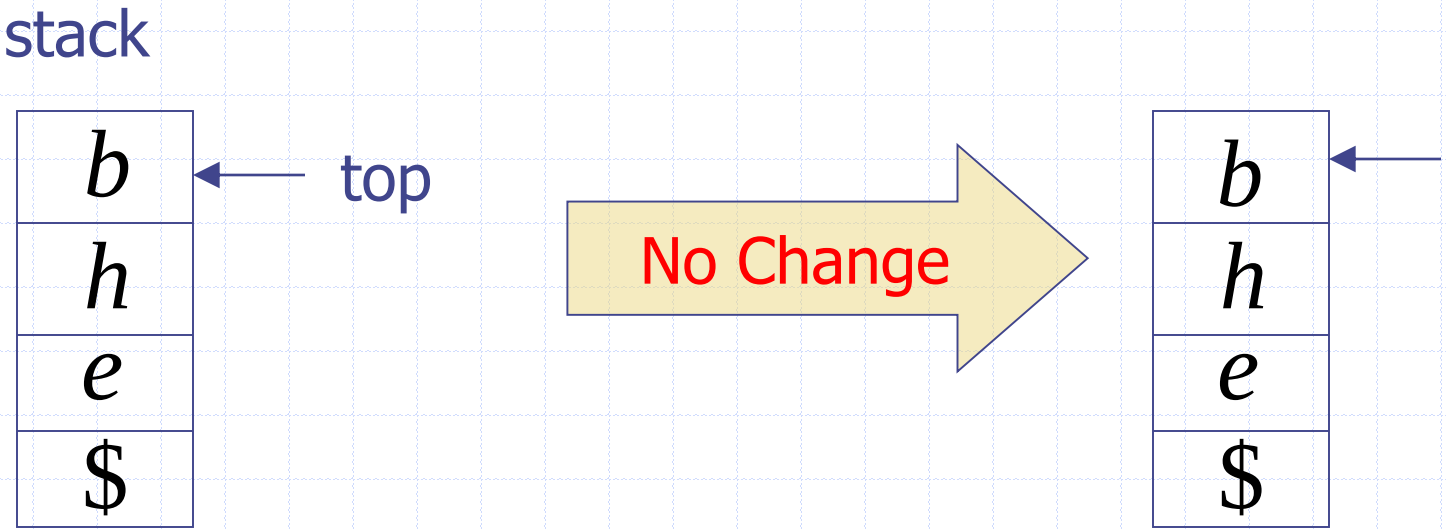

$a,\ b \to \lambda$
q_1
q_2


input
$\ldots$
a
$\ldots$
$\ldots$
a
$\ldots$

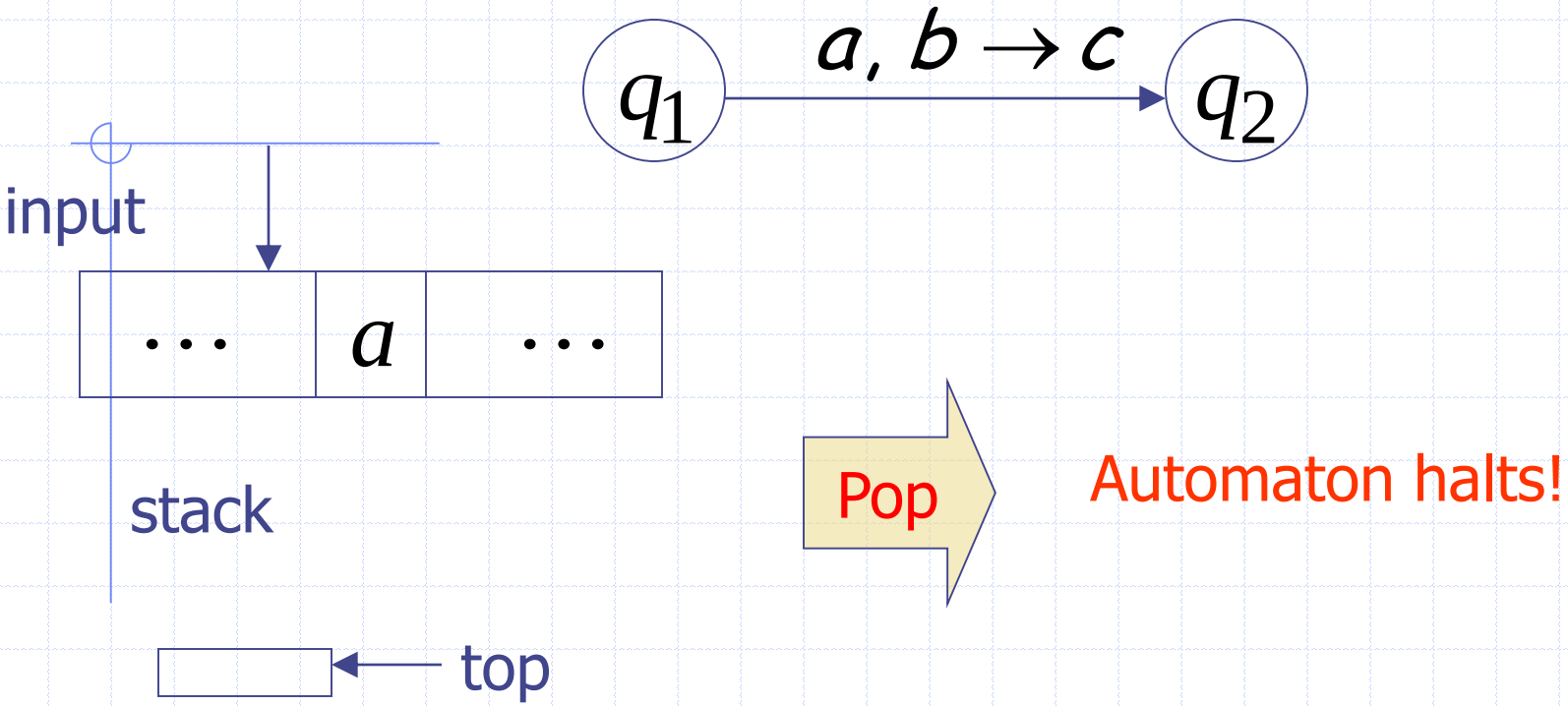

stack
b
top
h
e
$\$$
Pop
h
e
$\$$


$a,\ \lambda \rightarrow \lambda$
q_1
q_2


input
$\ldots$
a
$\ldots$
$\ldots$
a
$\ldots$


stack
b
top
h
e
$\$$
No Change
b
h
e
$\$$

Pop from Empty Stack

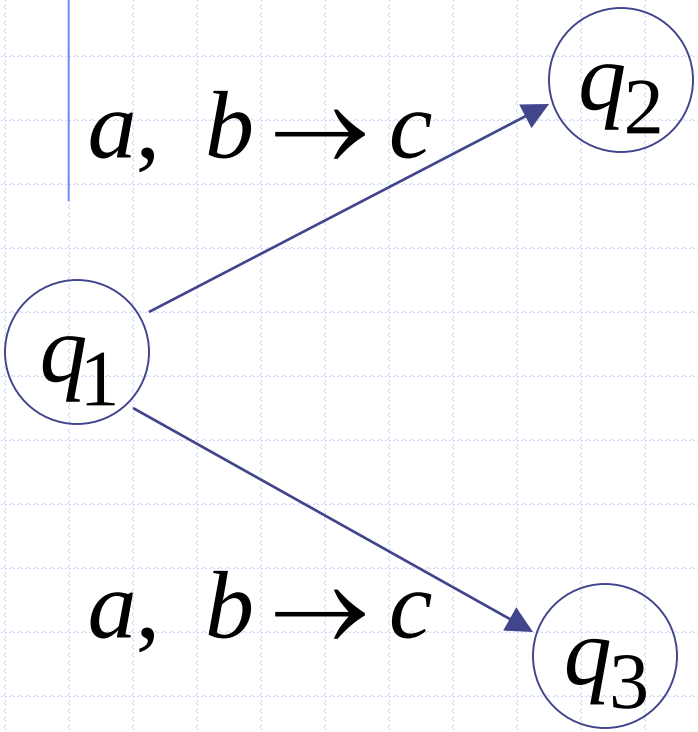


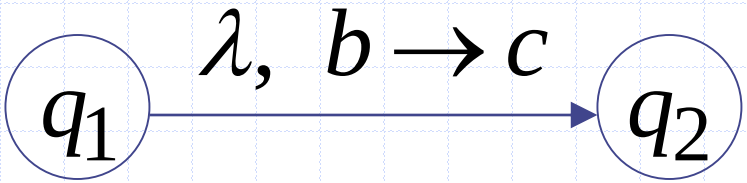
If the automaton attempts to pop from empty stack then it halts and rejects input

Non-Determinism

PDAs are non-deterministic

Allowed non-deterministic transitions



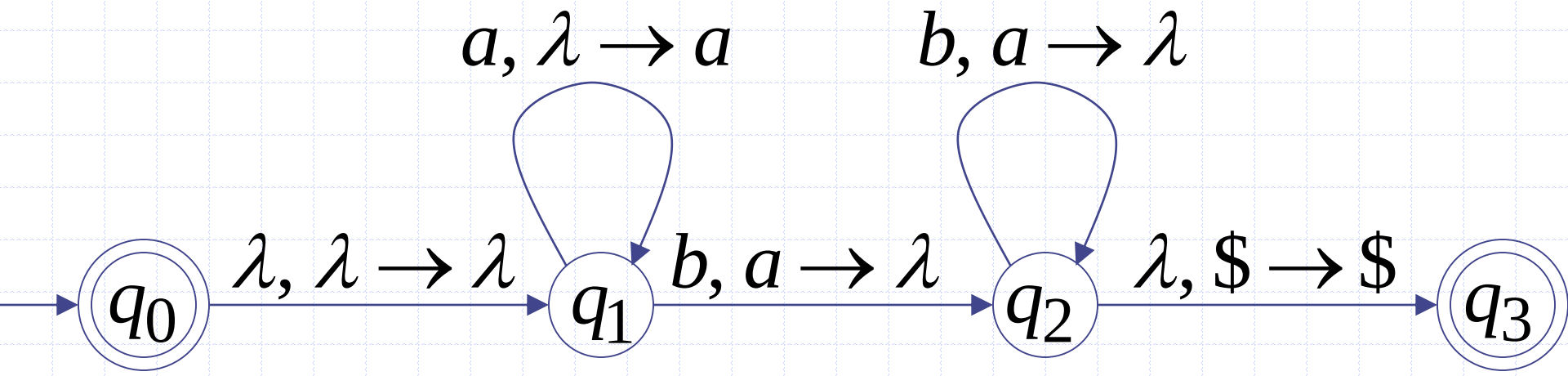


λ – transition

Example PDA

PDA M

$$L(M) = \{a^n b^n : n \geq 0\}$$



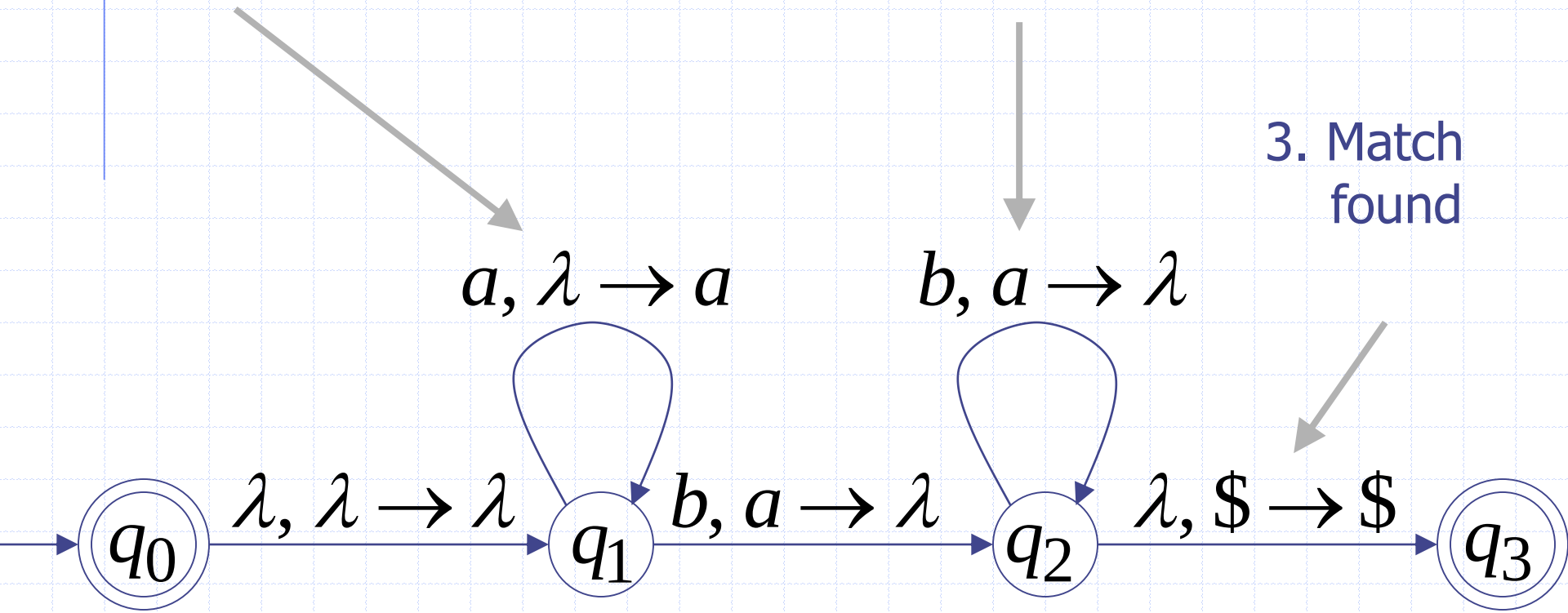
$$L(M) = \{a^n b^n : n \geq 0\}$$

Basic Idea:

1. Push the a's on the stack

2. Match the b's on input with a's on stack

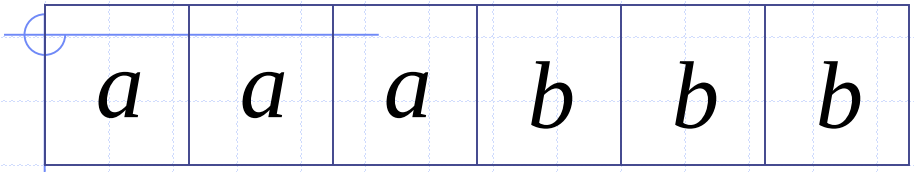
3. Match found

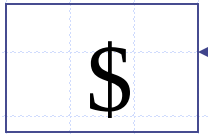


Execution Example:

Time 0

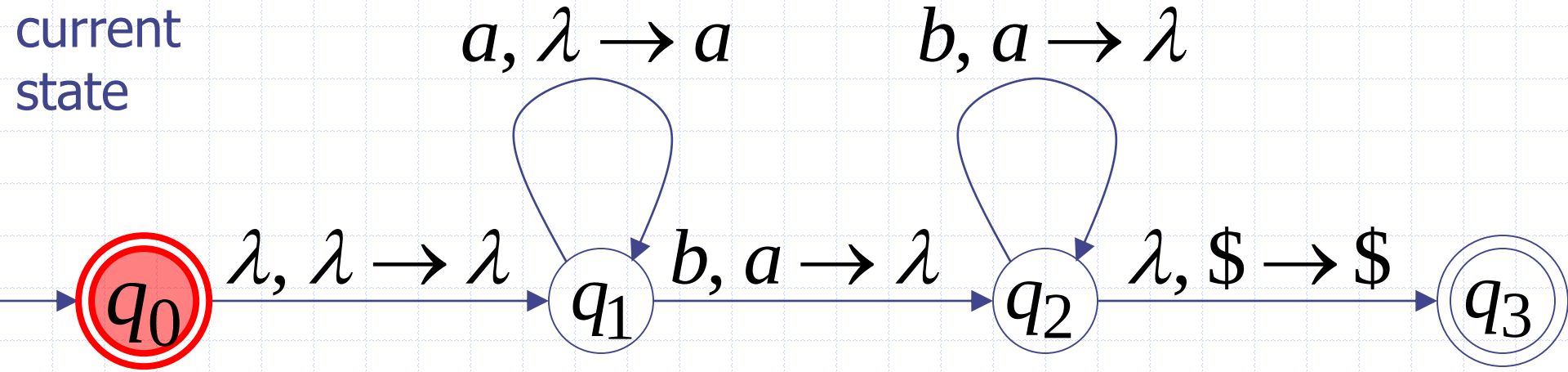
Input



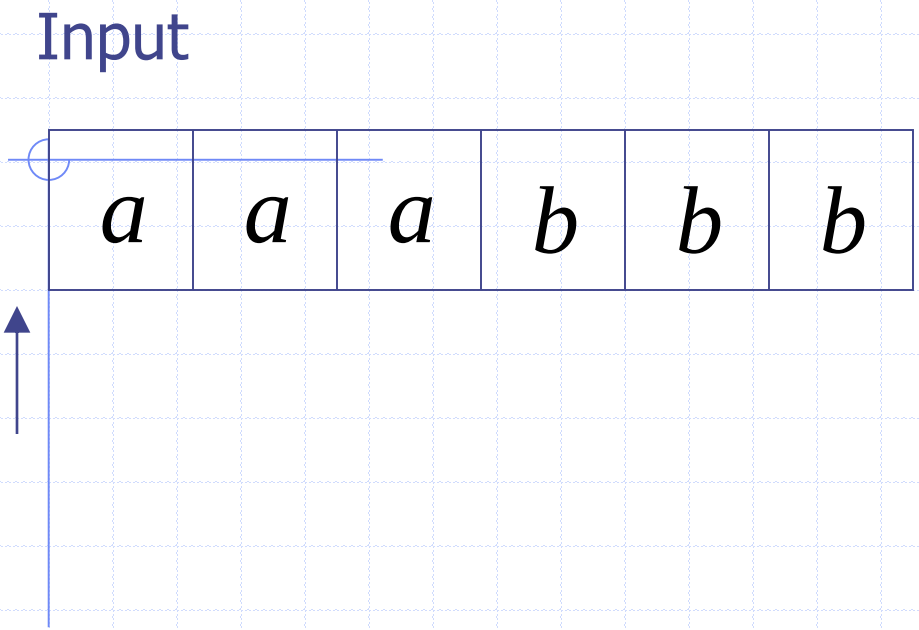


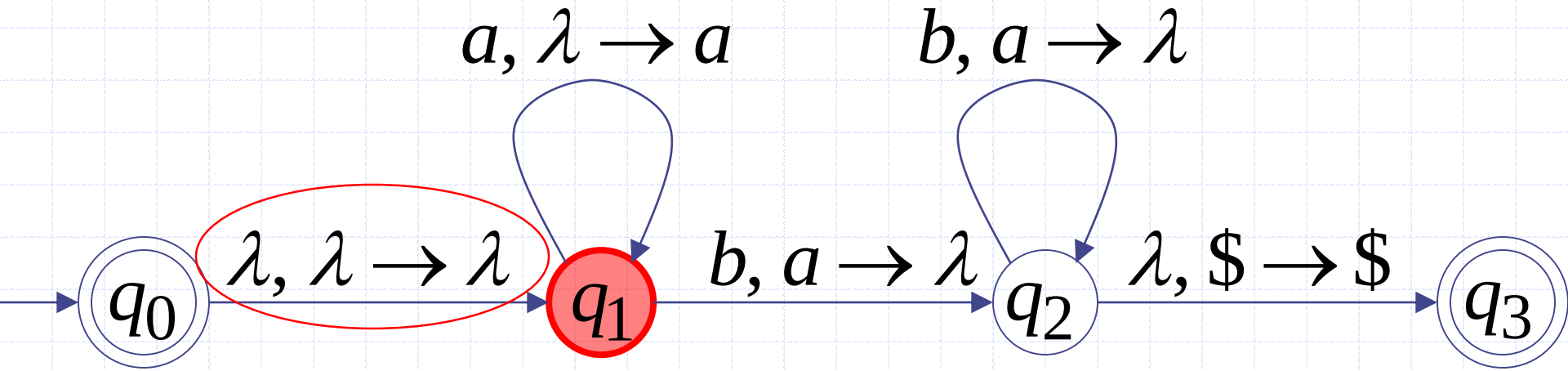
Stack

current state



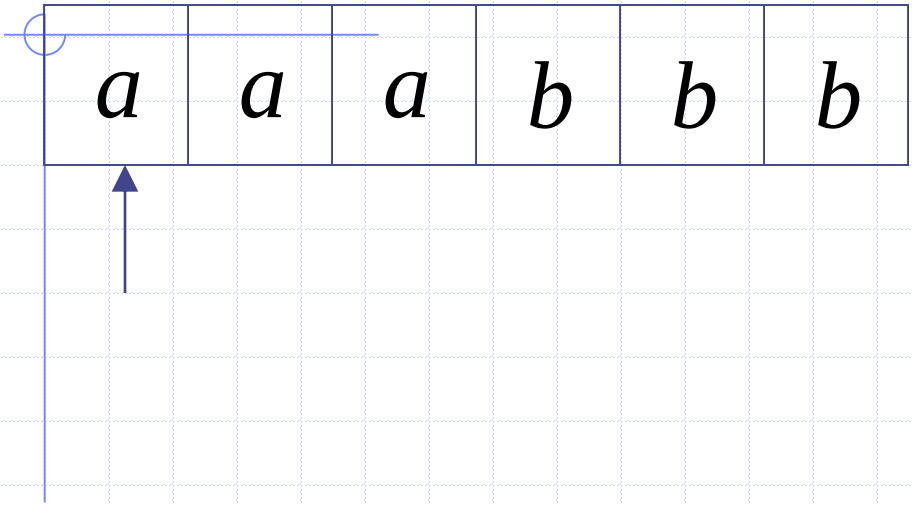
Time 1

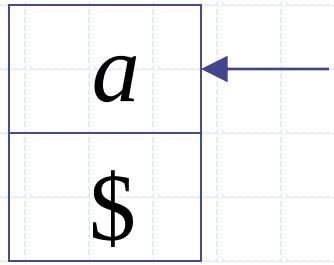




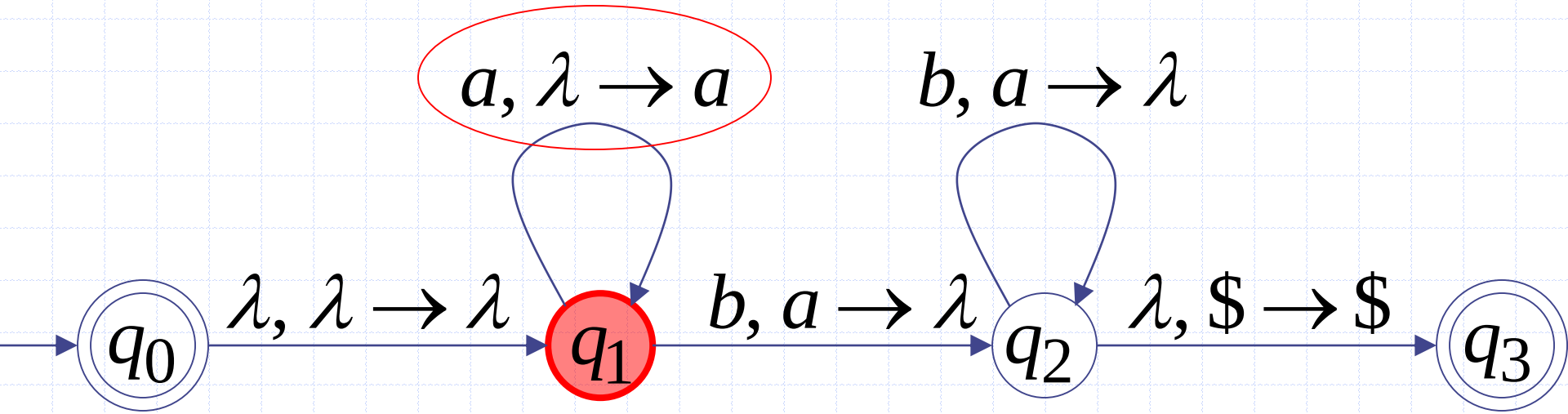
Time 2

Input



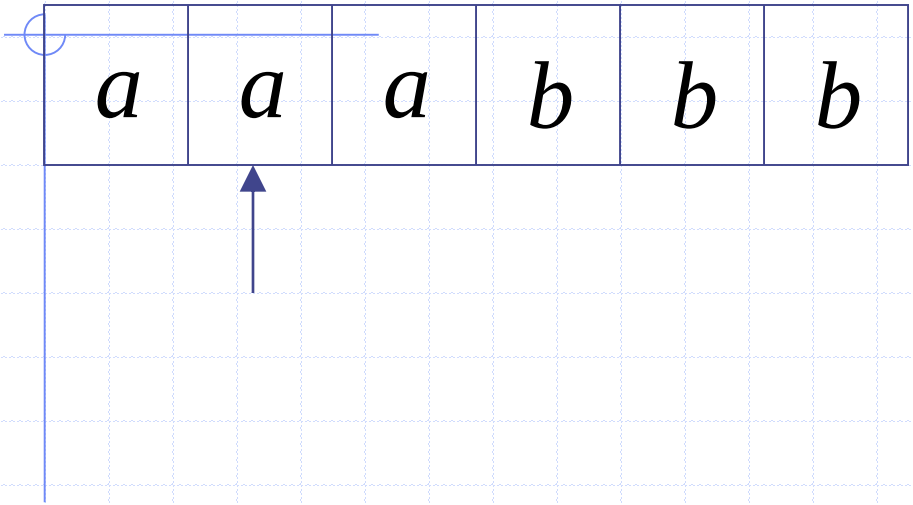


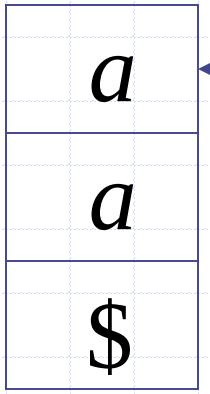
Stack



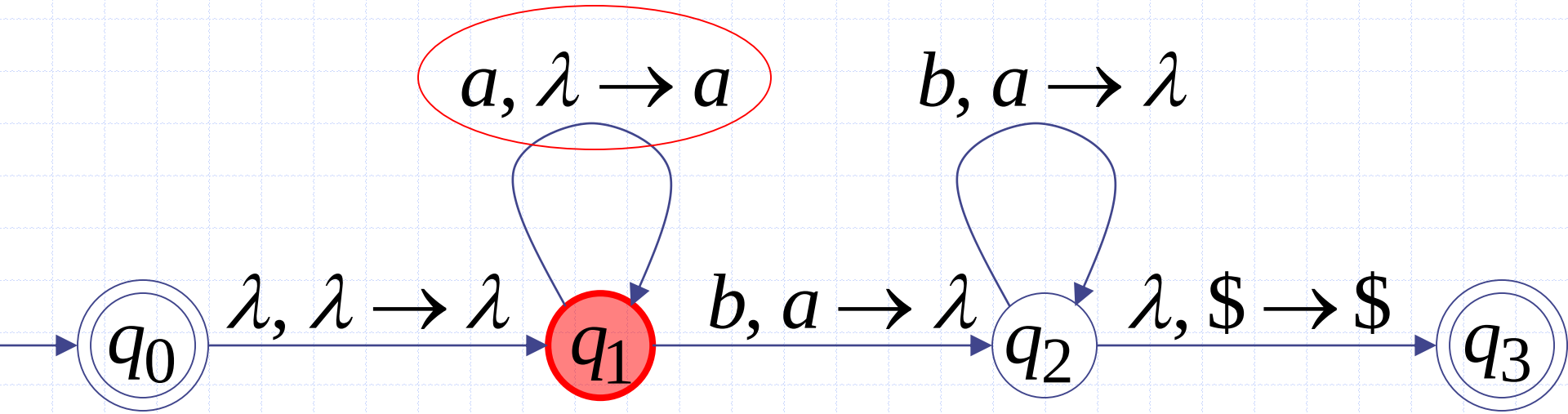
Time 3

Input



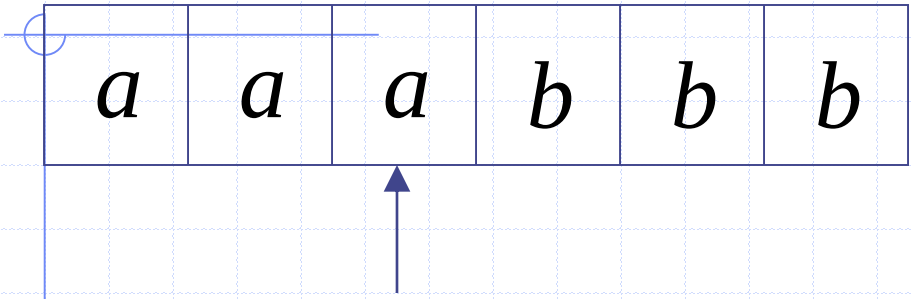


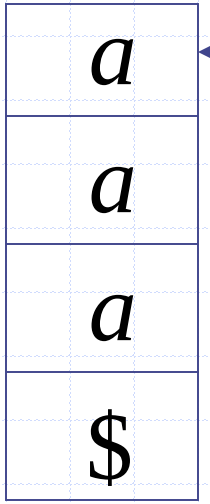
Stack



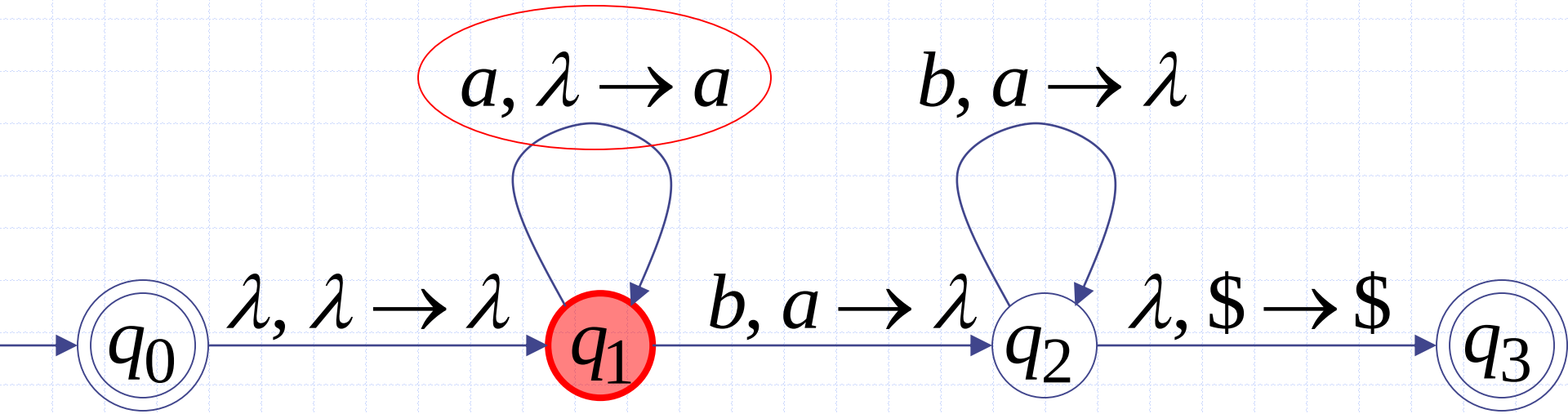
Time 4

Input



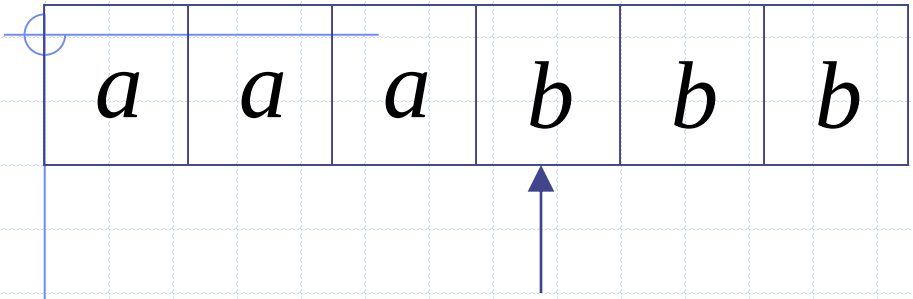


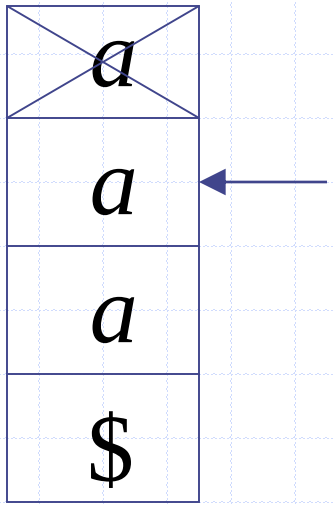
Stack



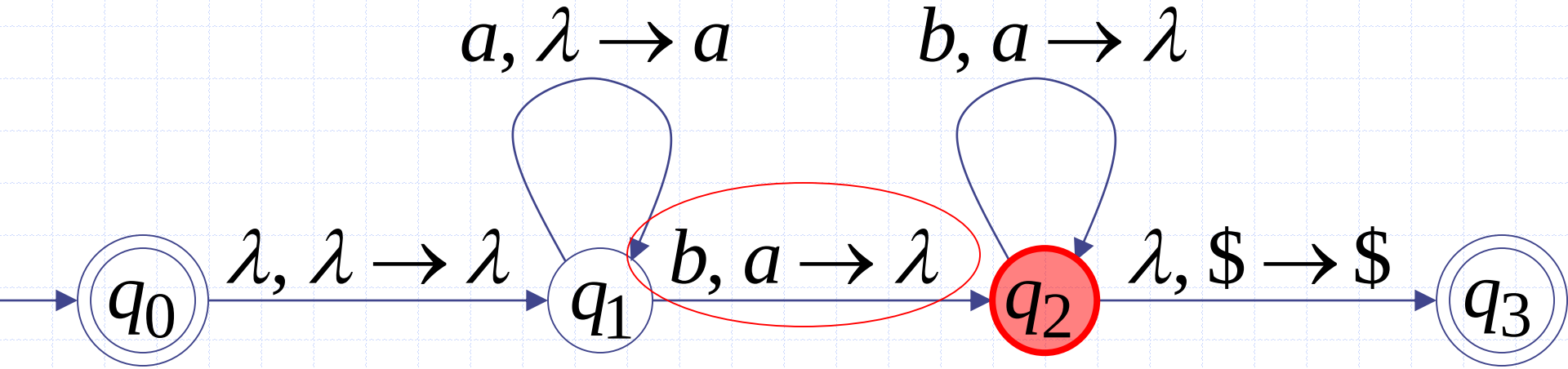
Time 5

Input



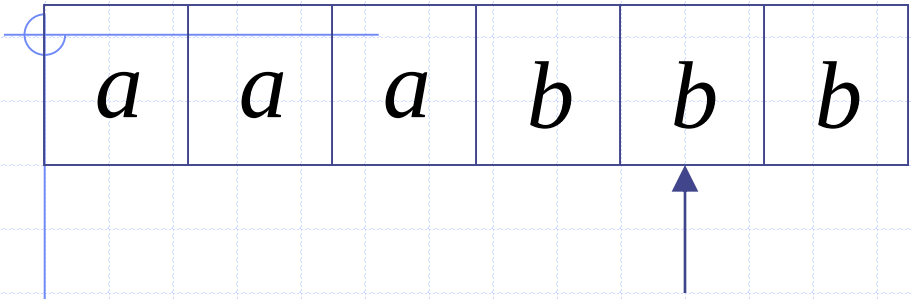


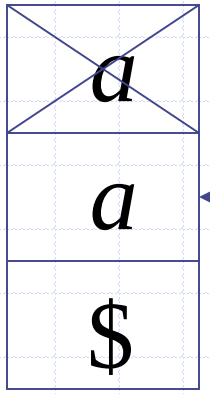
Stack



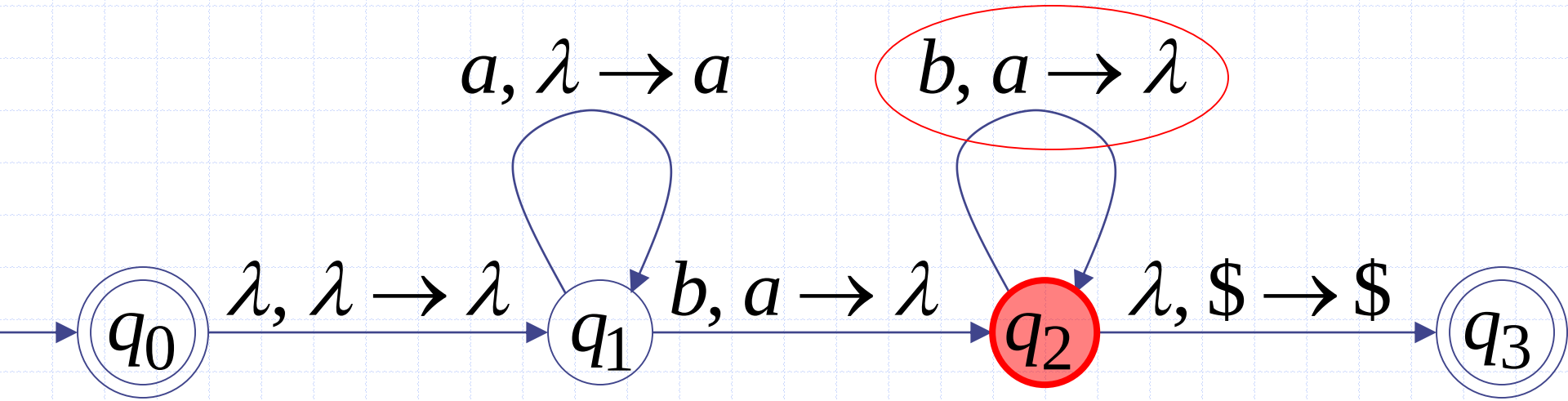
Time 6

Input



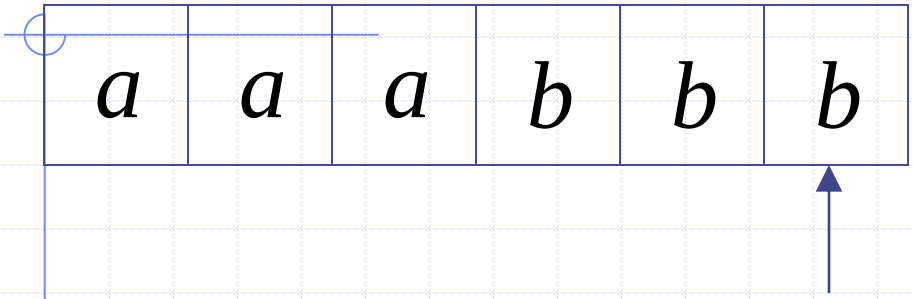


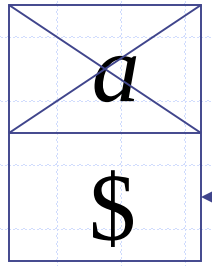
Stack



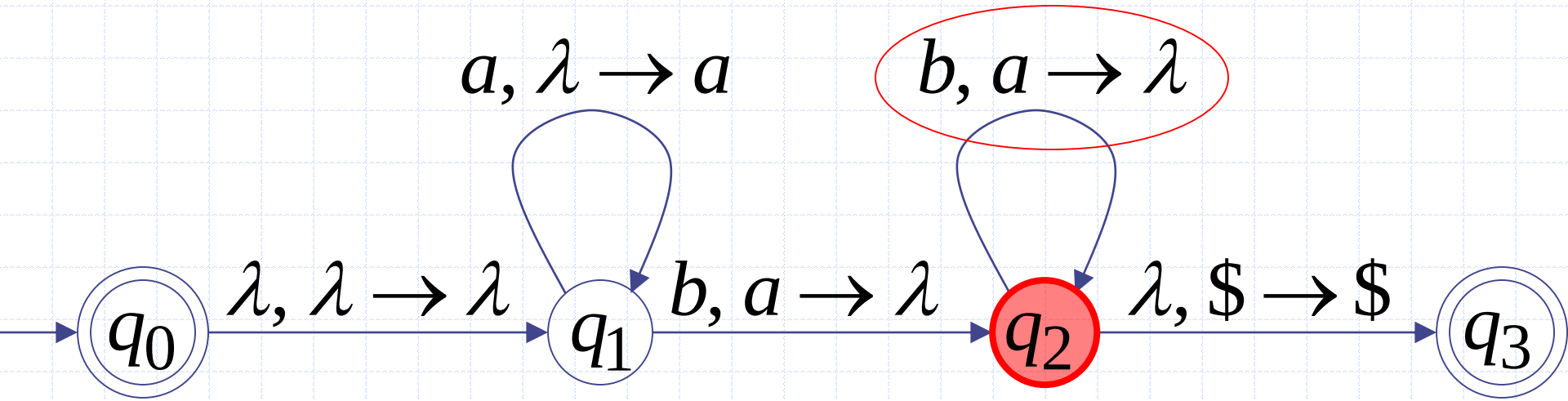
Time 7

Input



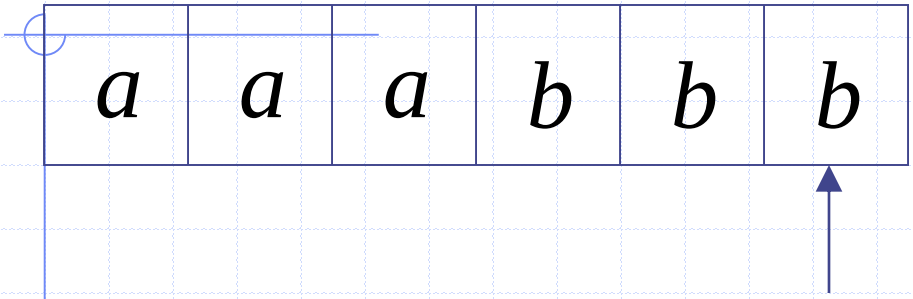


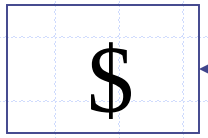
Stack



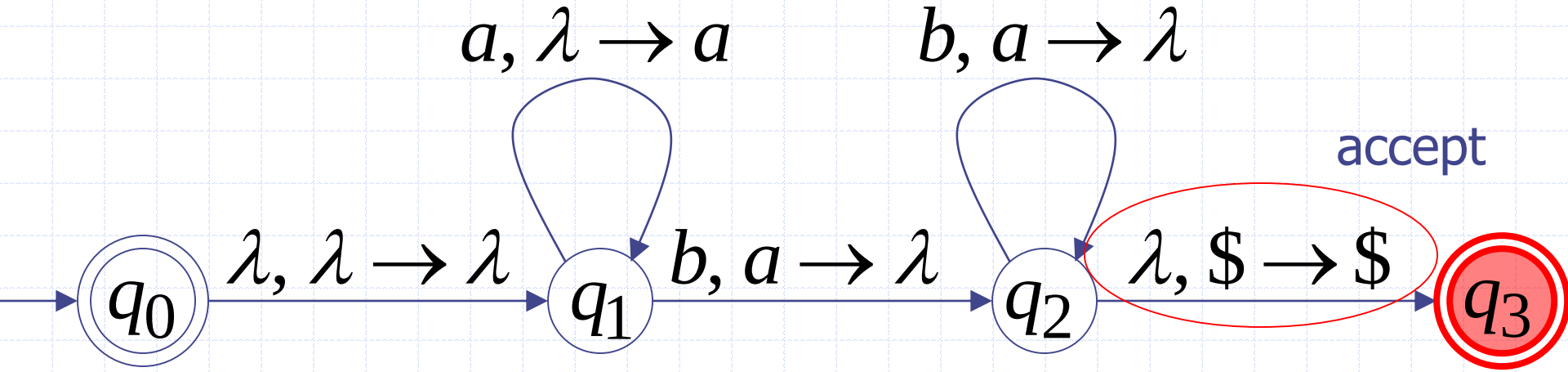
Time 8

Input





Stack



A string is accepted if there is a computation such that:

All the input is consumed

AND

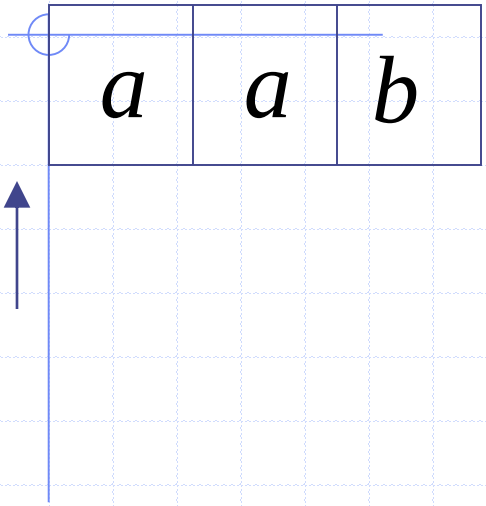
The last state is an accepting state

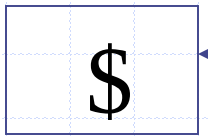
we do not care about the stack contents at the end of the accepting computation

Rejection Example:

Time 0

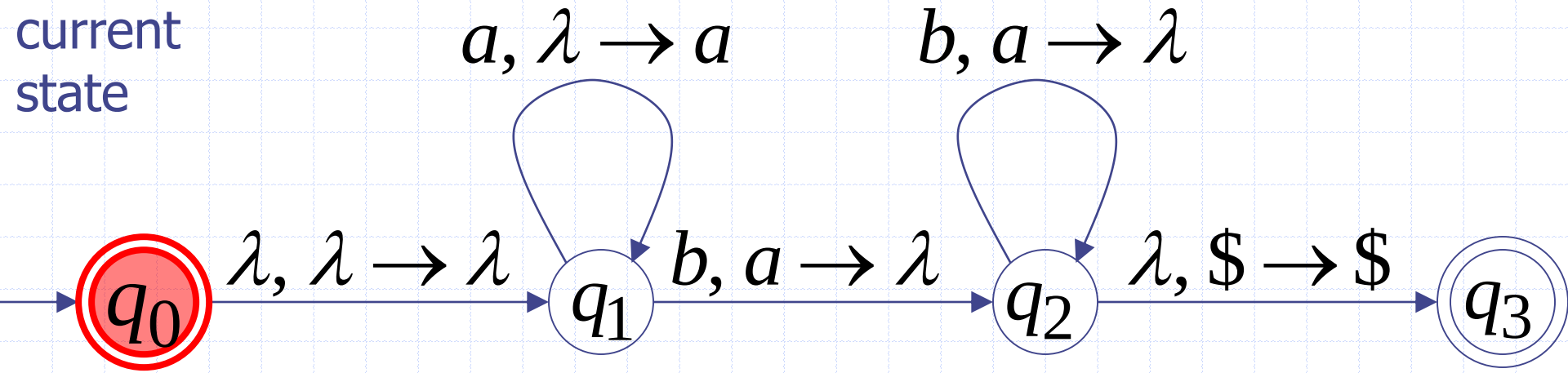
Input





Stack

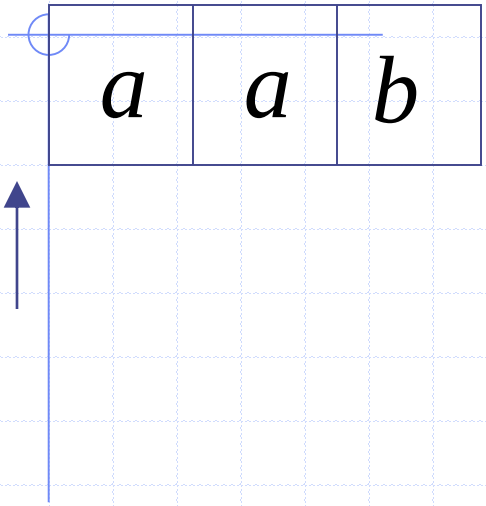
current state

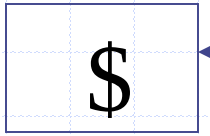


Rejection Example:

Time 1

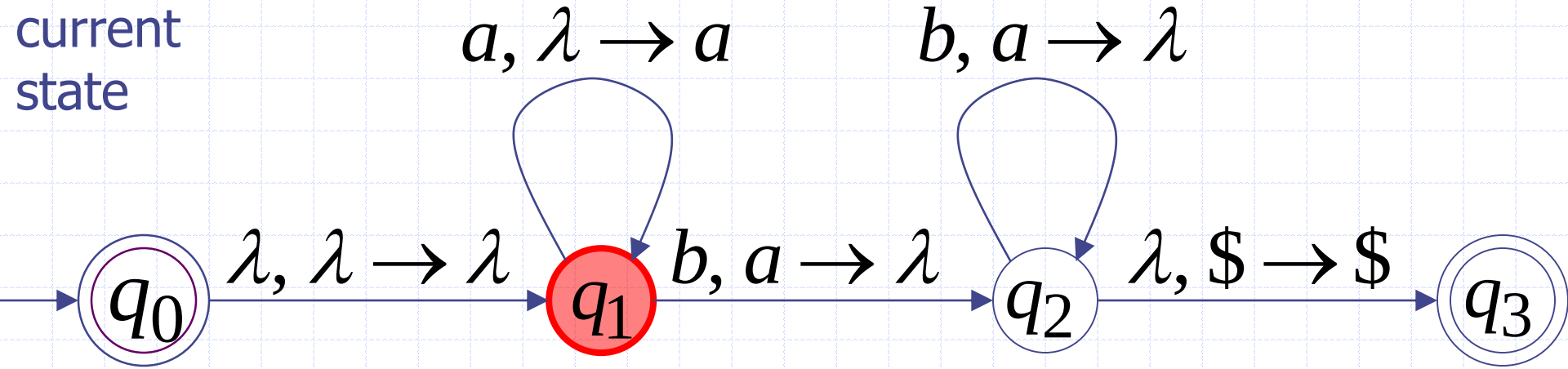
Input





Stack

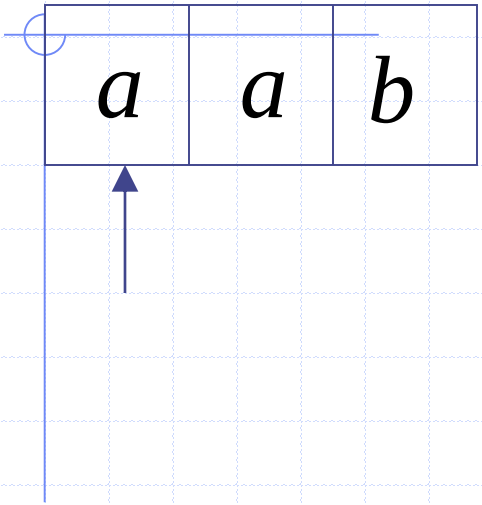
current state

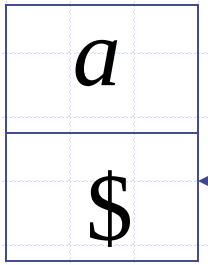


Rejection Example:

Time 2

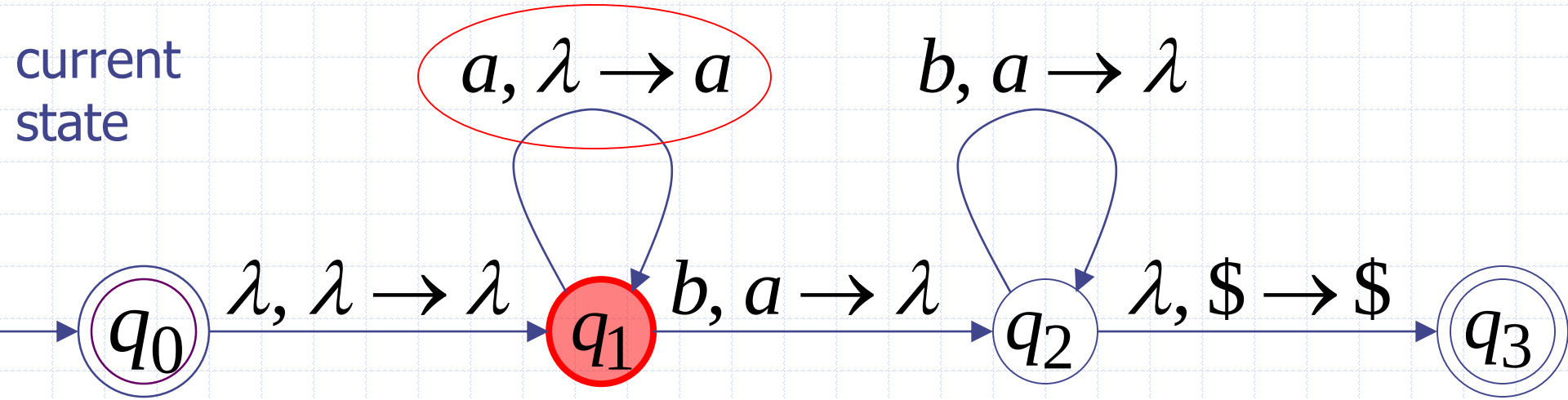
Input





Stack

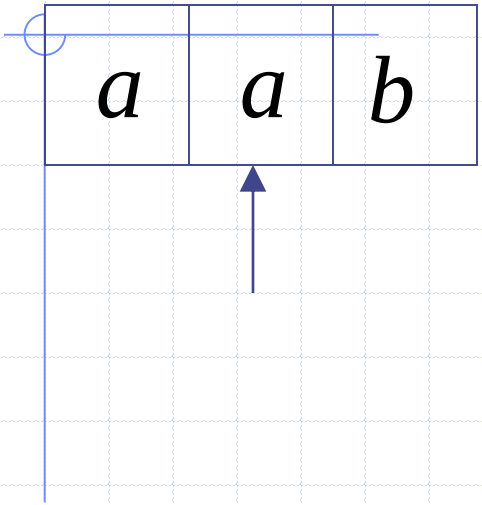
current state

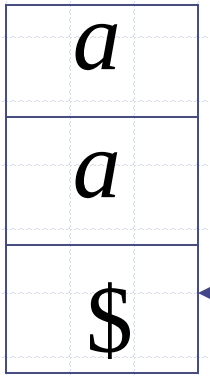


Rejection Example:

Time 3

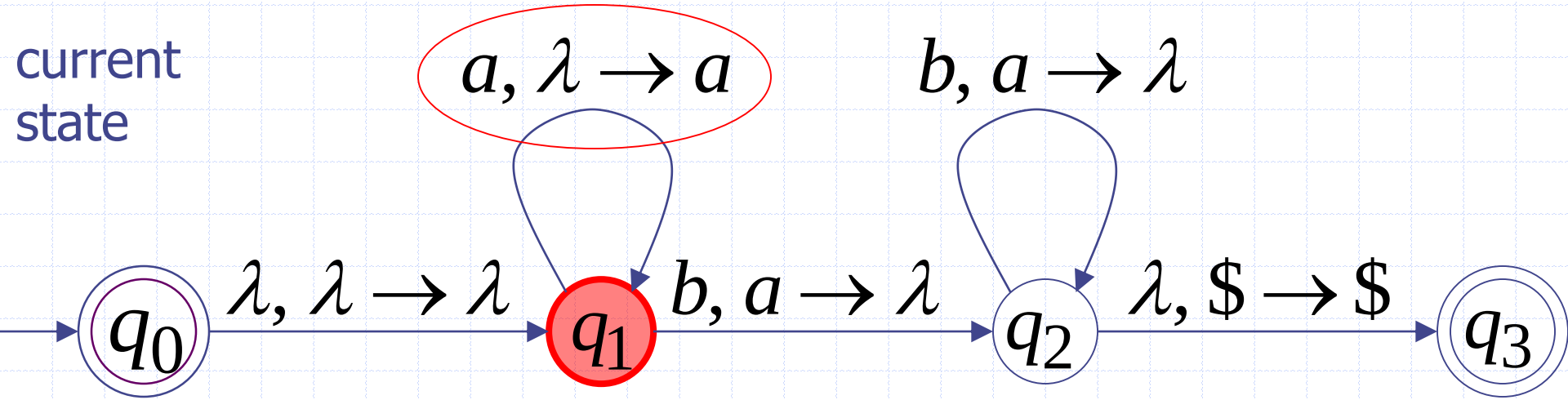
Input





Stack

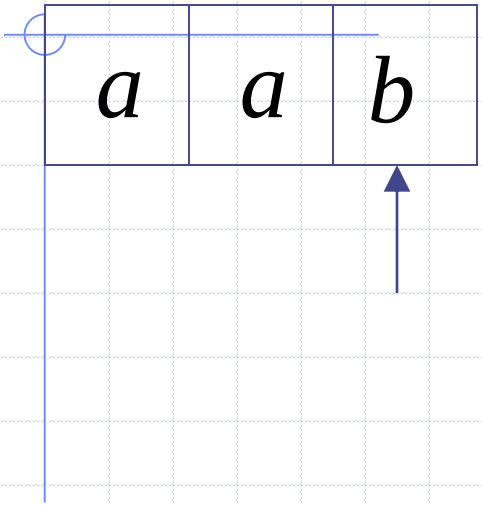
current state

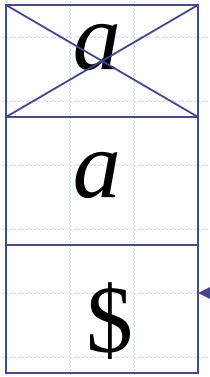


Rejection Example:

Time 4

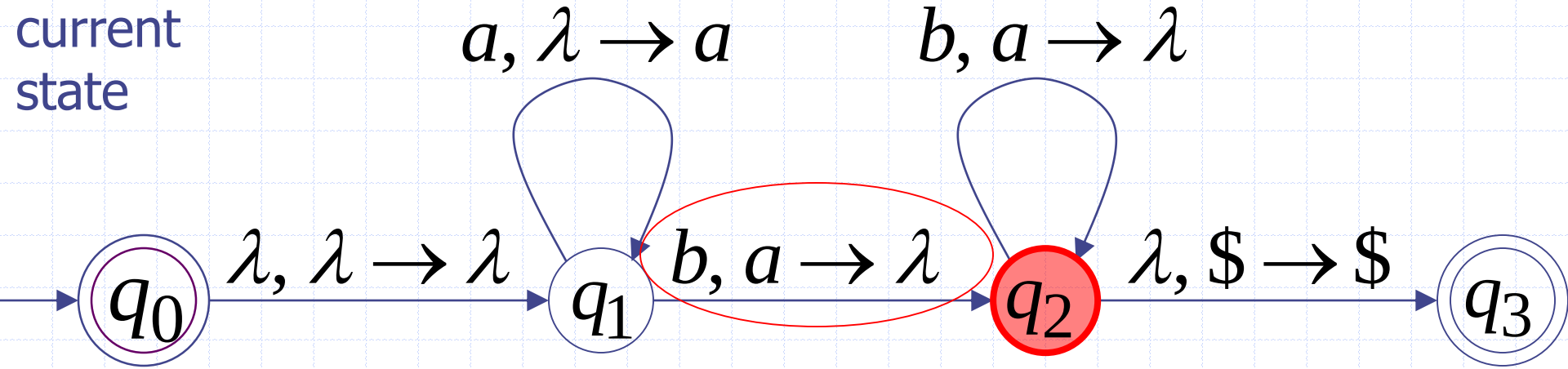
Input





Stack

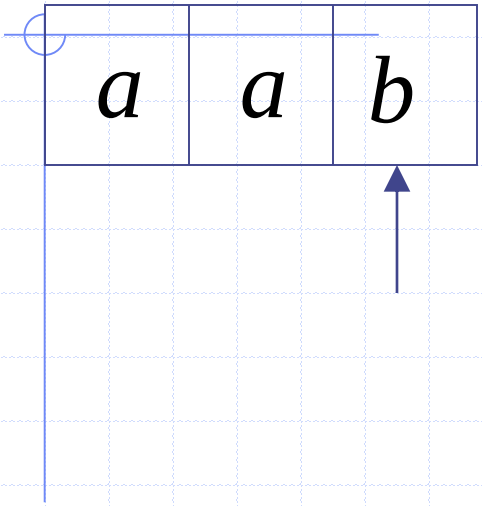
current state

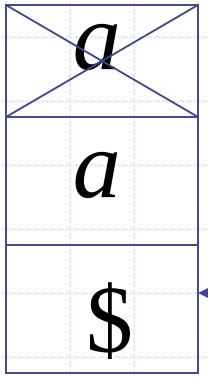


Rejection Example:

Time 4

Input

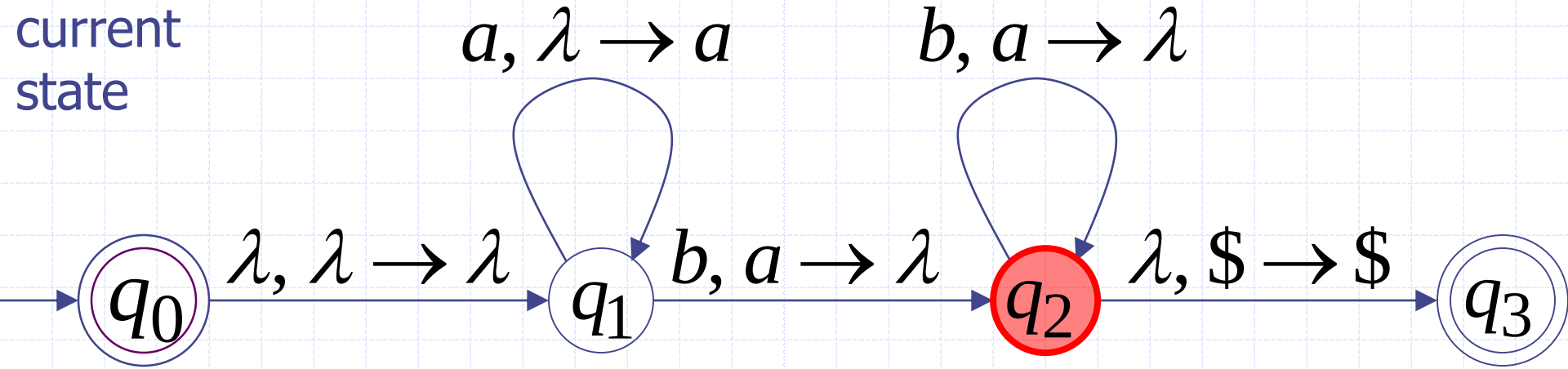




Stack

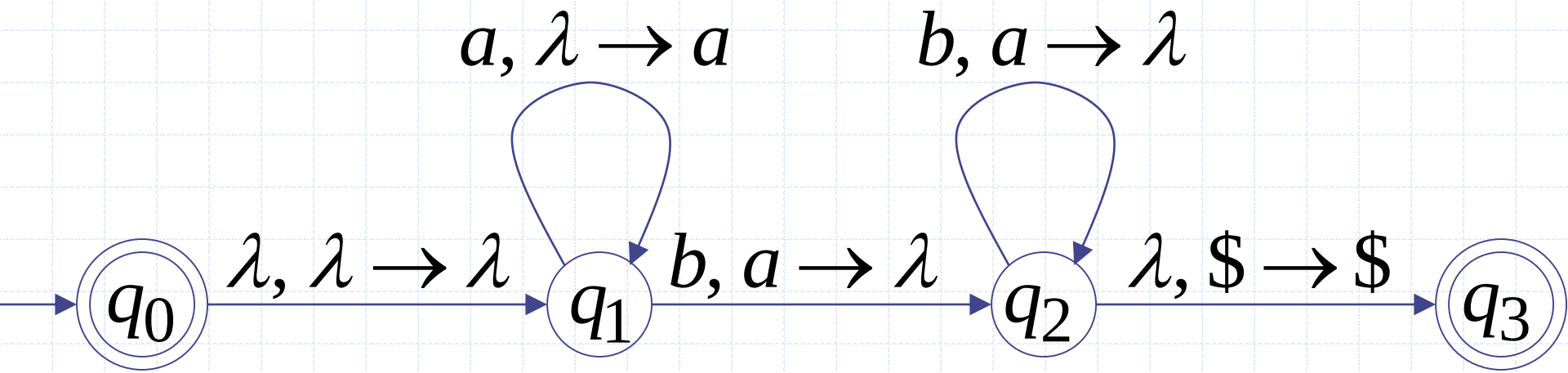
reject

current state



There is no accepting computation for aab

The string aab is rejected by the PDA



Another PDA example

PDA M

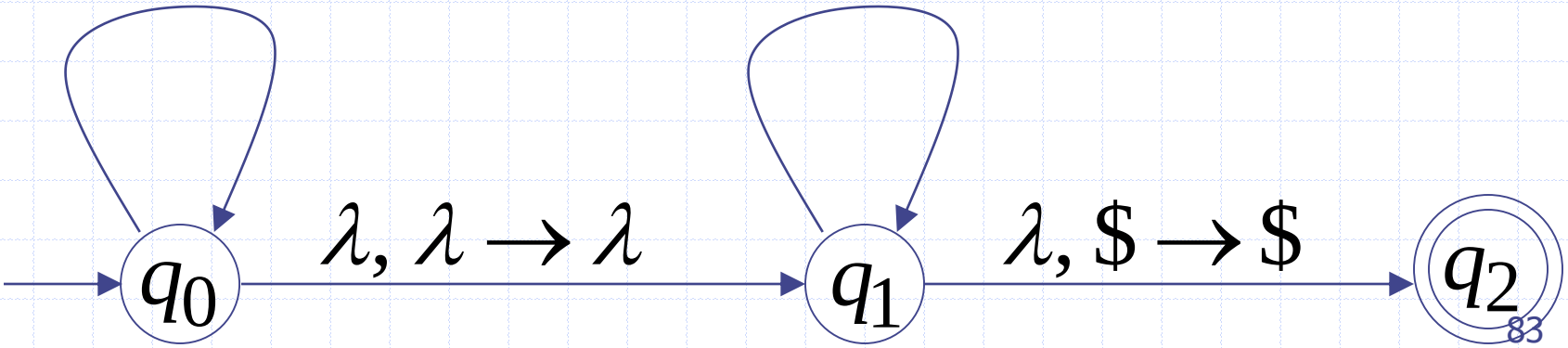
$$L(M) = \{vv^R : v \in \{a,b\}^*\}$$

$a, \lambda \to a$

$b, \lambda \to b$

$a, a \to \lambda$

$b, b \to \lambda$



Basic Idea:

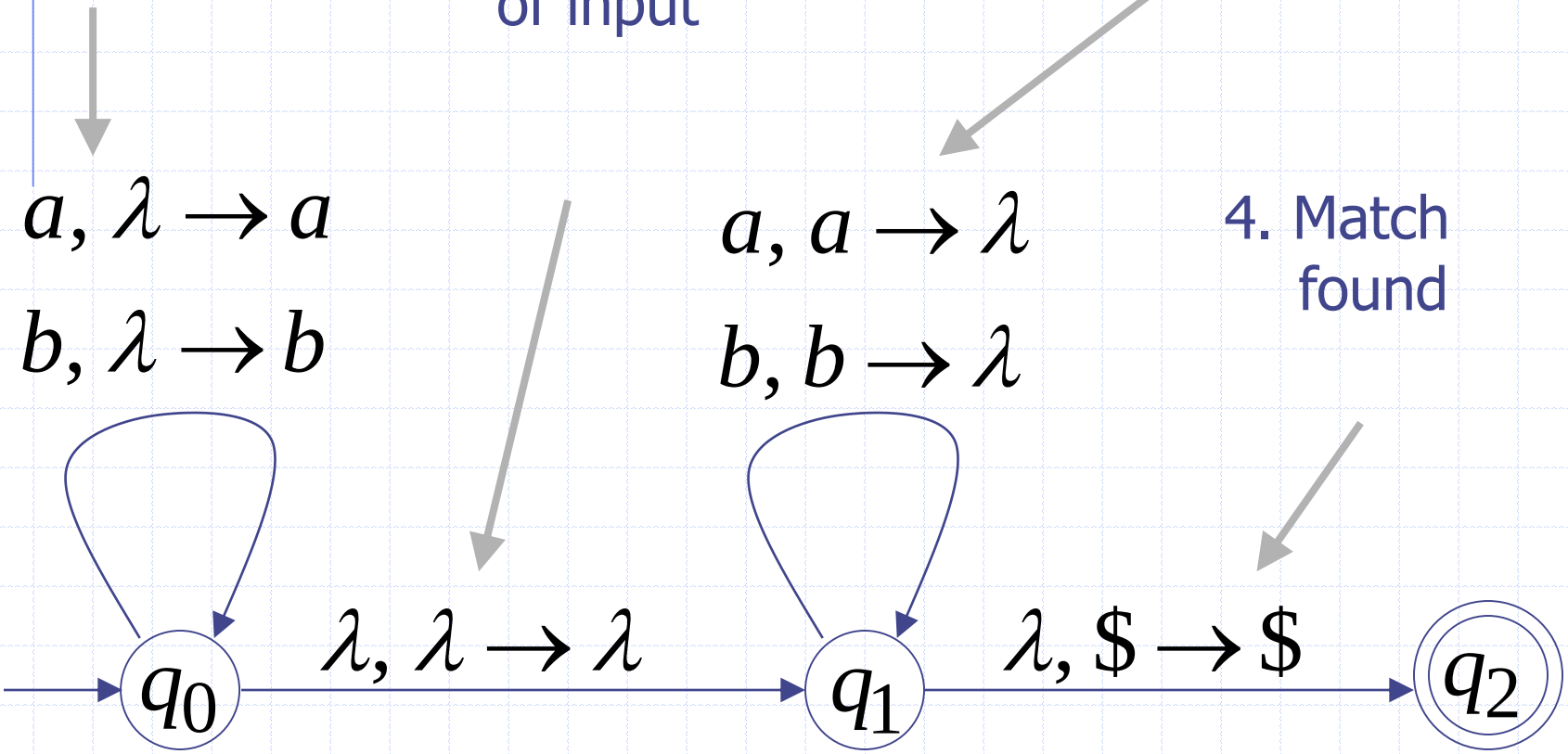
$$L(M) = \{vv^R : v \in \{a,b\}^*\}$$

1. Push v on stack

2. Guess middle of input

3. Match v^R on input with v on stack

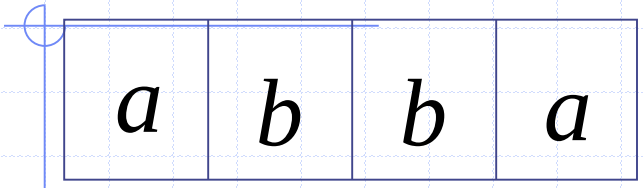
4. Match found

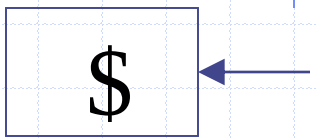


Execution Example:

Time 0

Input





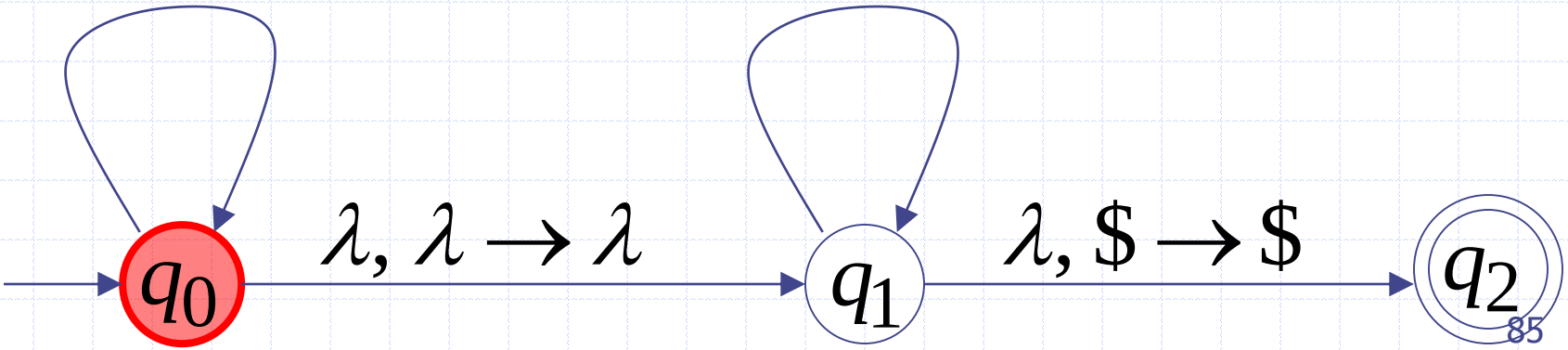
Stack

$a, \lambda \to a$

$b, \lambda \to b$

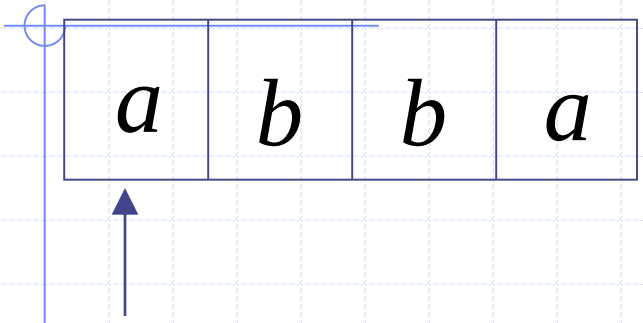
$a, a \to \lambda$

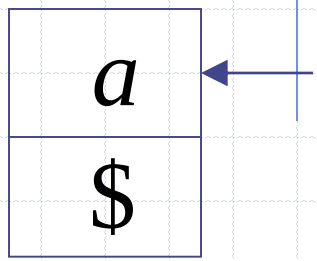
$b, b \to \lambda$



Time 1

Input





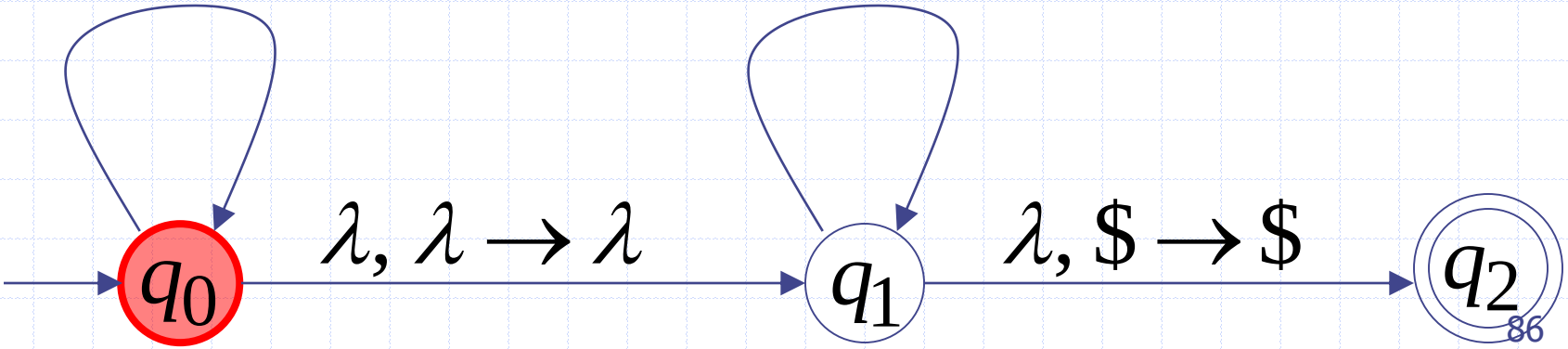
Stack

$a, \lambda \rightarrow a$

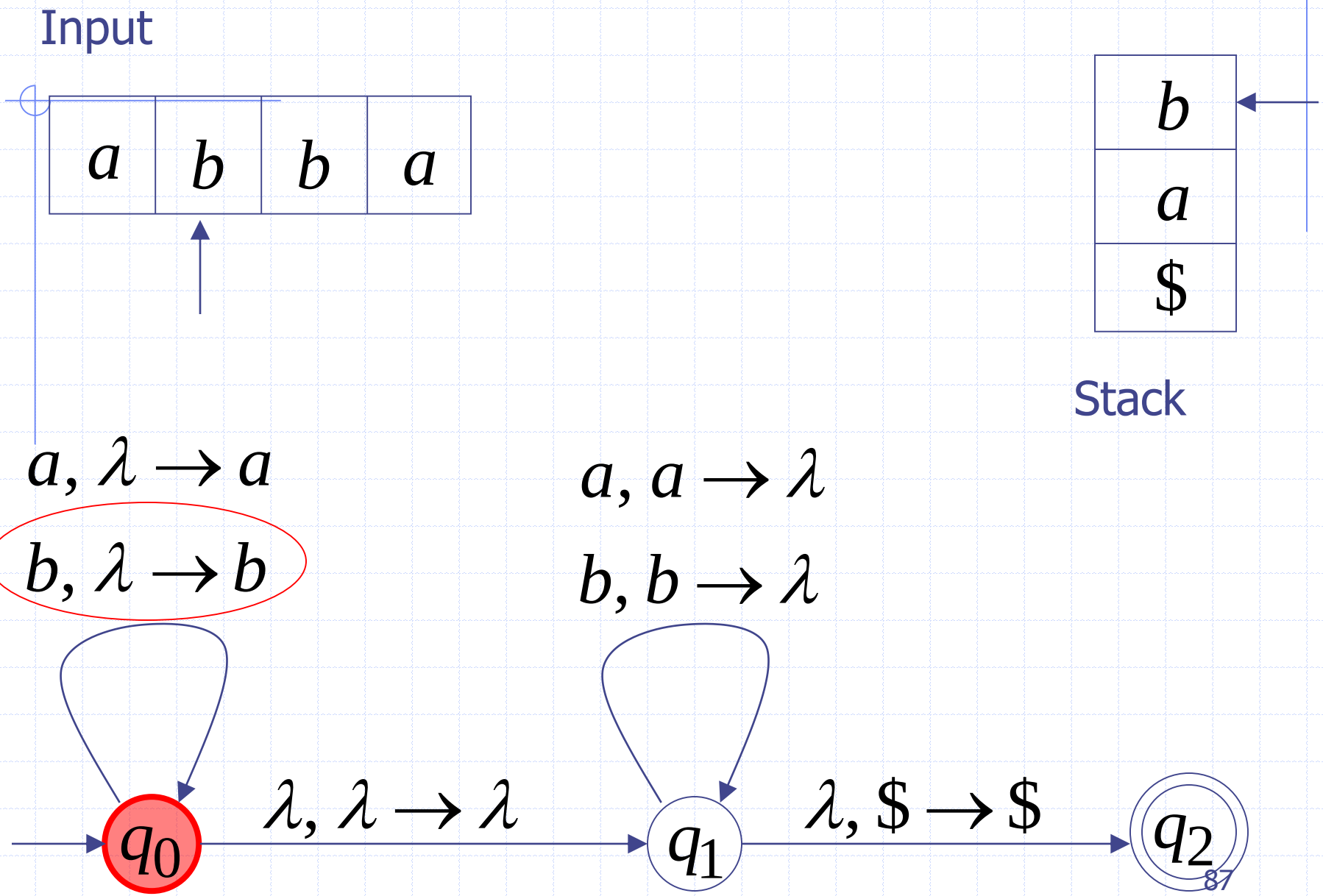
$b, \lambda \rightarrow b$

$a, a \rightarrow \lambda$

$b, b \rightarrow \lambda$

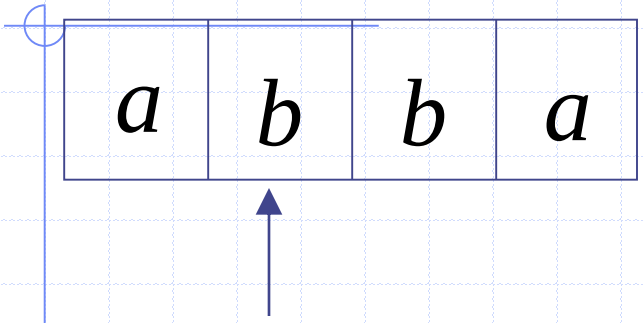


Time 2

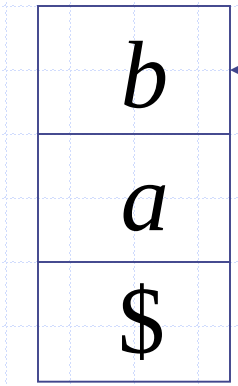


Time 3

Input



Guess the middle of string



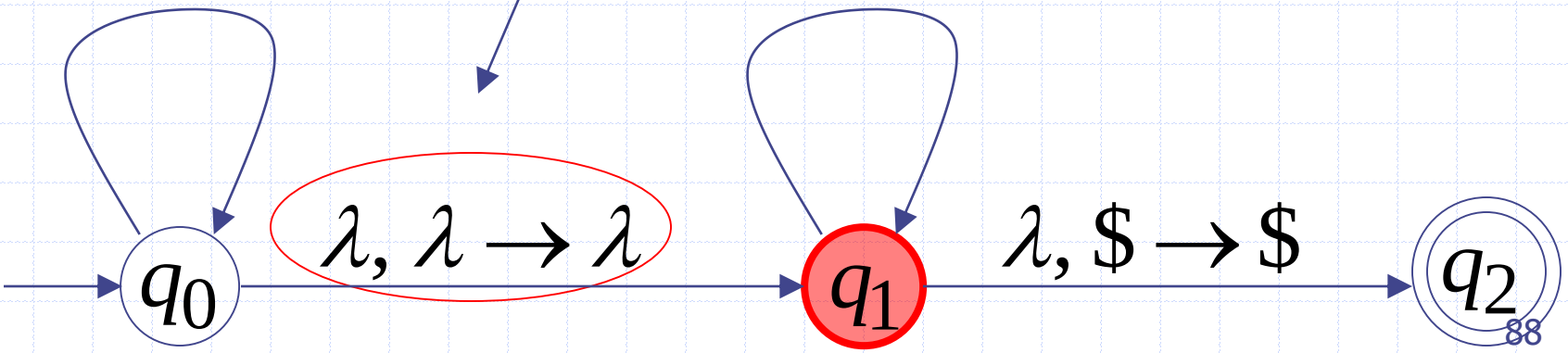
Stack

$a, \lambda \to a$

$b, \lambda \to b$

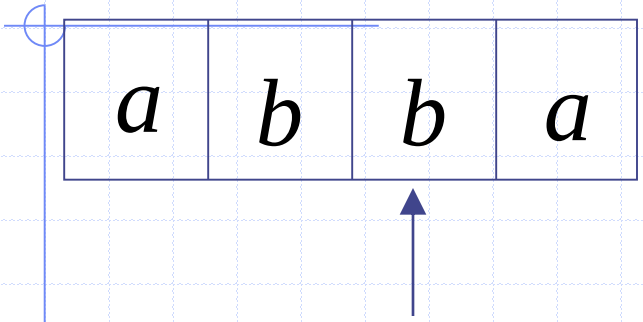
$a, a \to \lambda$

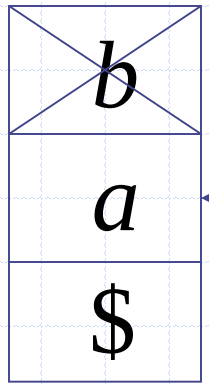
$b, b \to \lambda$



Time 4

Input





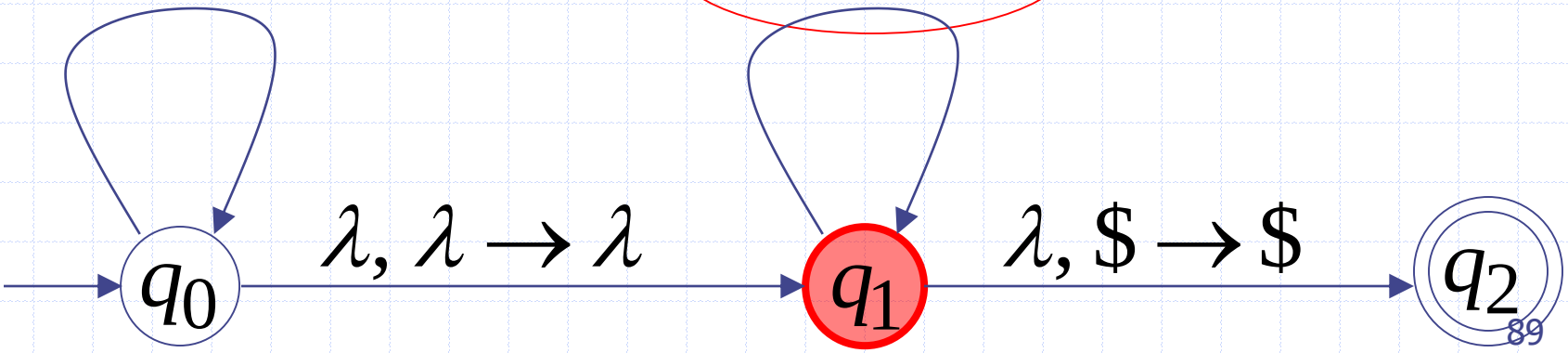
Stack

$a, \lambda \rightarrow a$

$b, \lambda \rightarrow b$

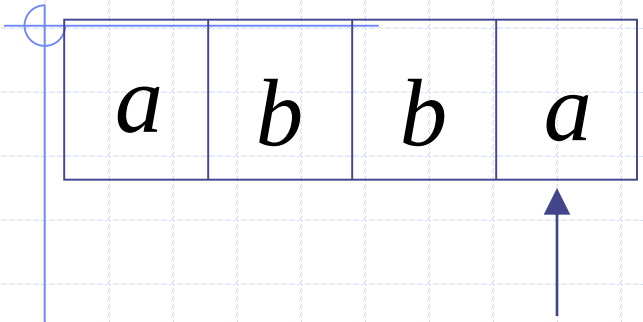
$a, a \rightarrow \lambda$

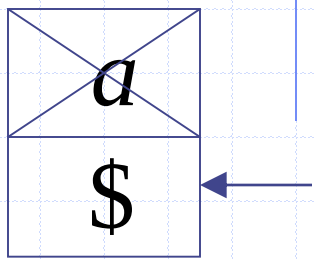
$b, b \rightarrow \lambda$



Time 5

Input





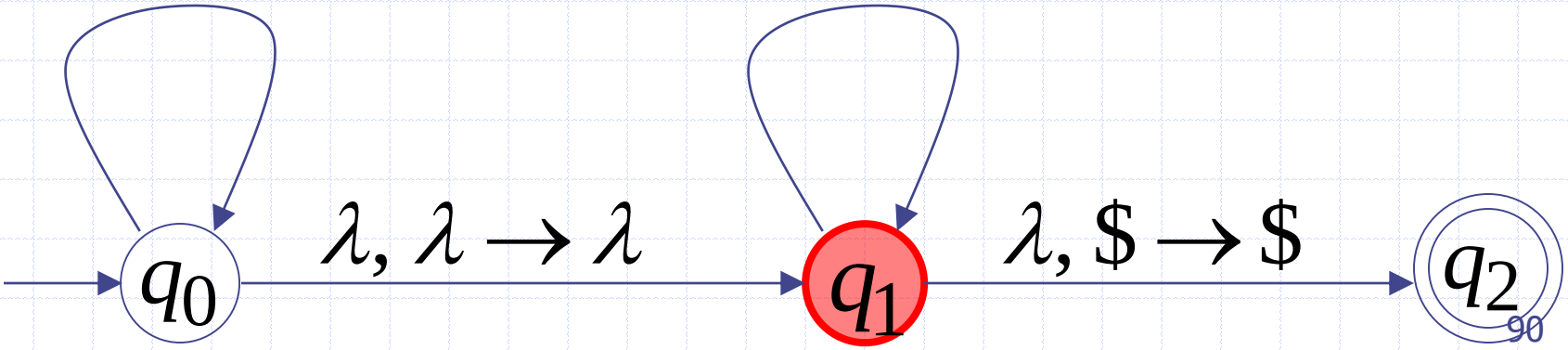
Stack

$a, \lambda \rightarrow a$

$b, \lambda \rightarrow b$

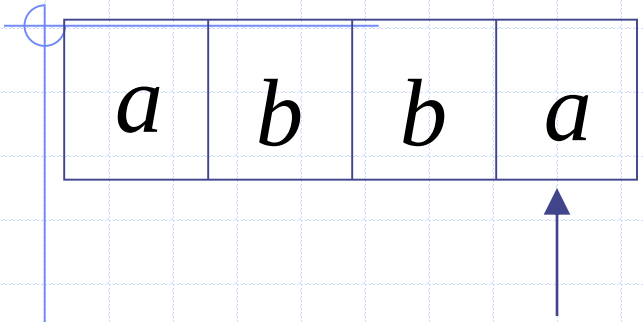
$a, a \rightarrow \lambda$

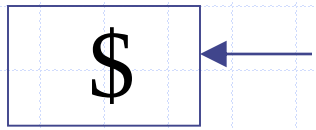
$b, b \rightarrow \lambda$



Time 6

Input





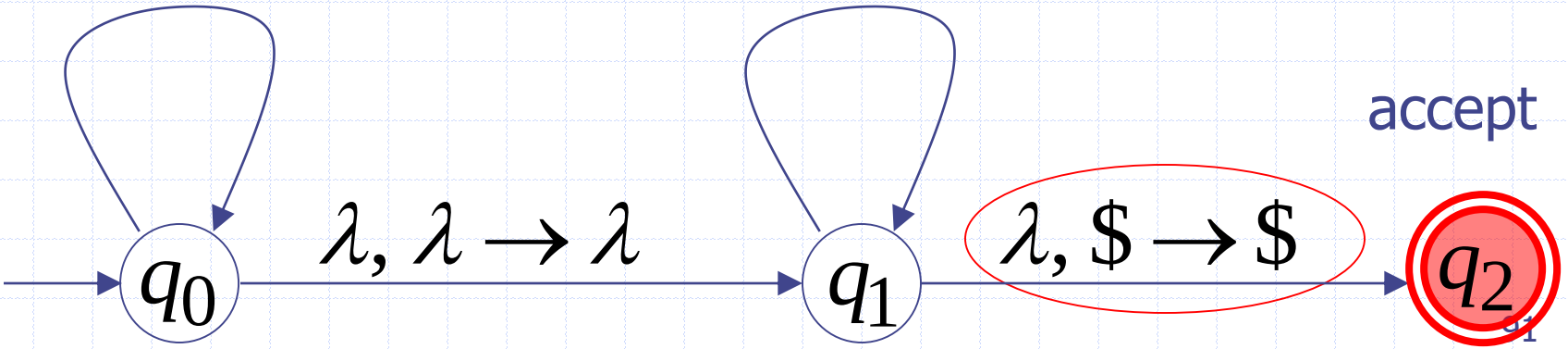
Stack

$a, \lambda \rightarrow a$

$b, \lambda \rightarrow b$

$a, a \rightarrow \lambda$

$b, b \rightarrow \lambda$

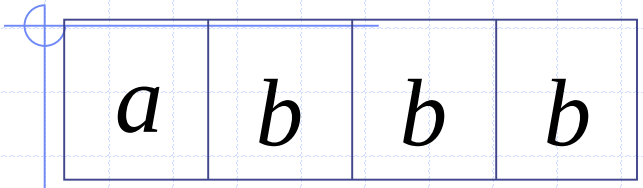


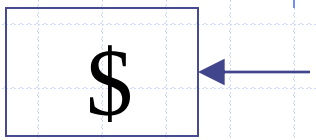
accept

Rejection Example:

Time 0

Input





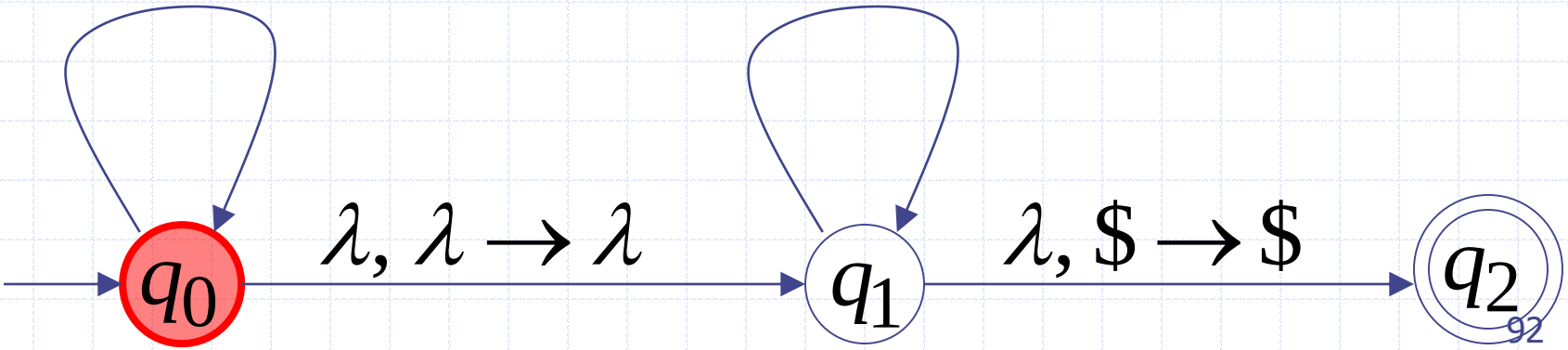
Stack

$a, \lambda \to a$

$b, \lambda \to b$

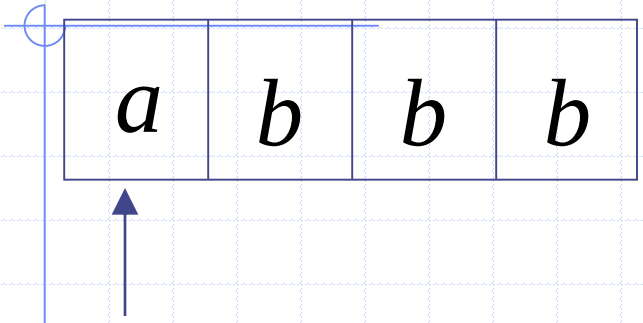
$a, a \to \lambda$

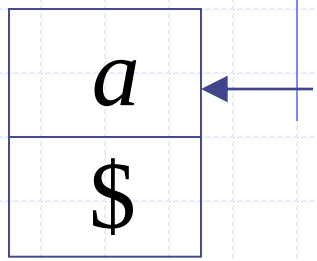
$b, b \to \lambda$



Time 1

Input





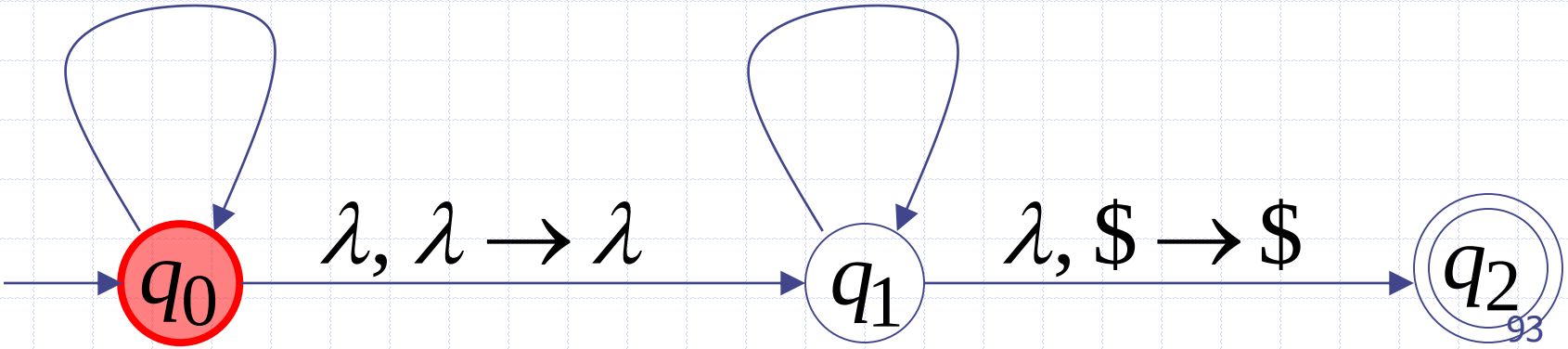
Stack

$a, \lambda \rightarrow a$

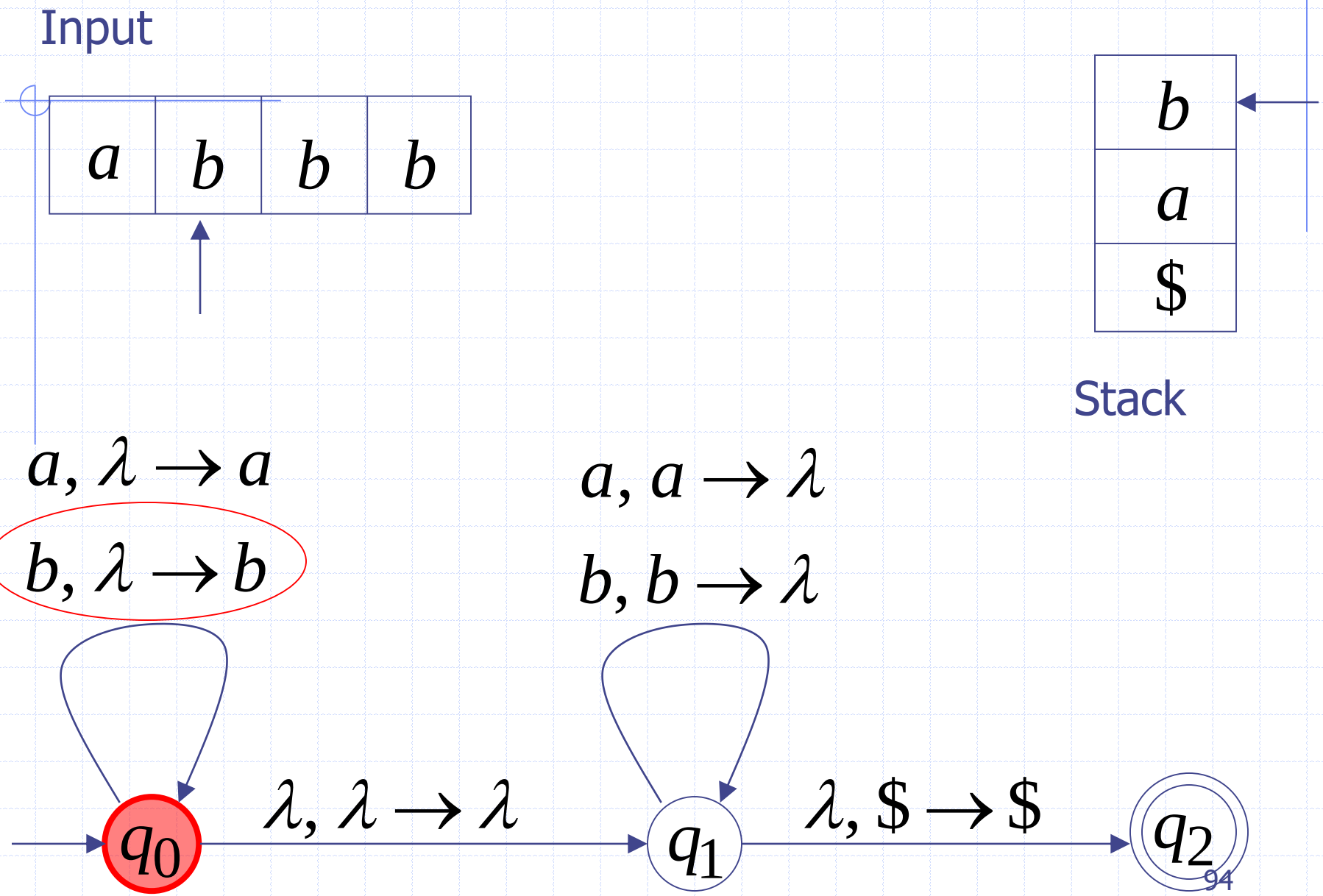
$b, \lambda \rightarrow b$

$a, a \rightarrow \lambda$

$b, b \rightarrow \lambda$

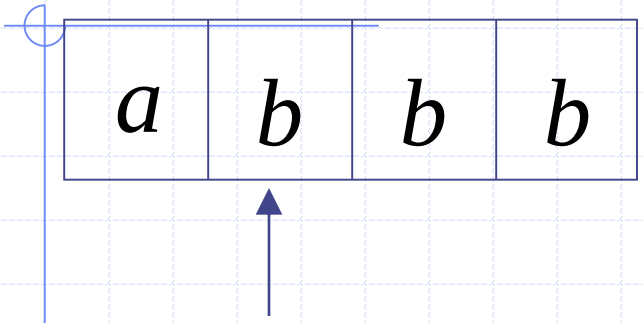


Time 2

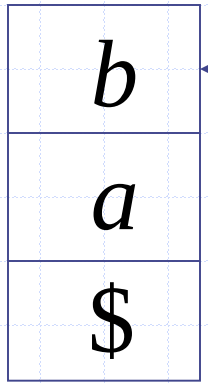


Time 3

Input



Guess the middle of string



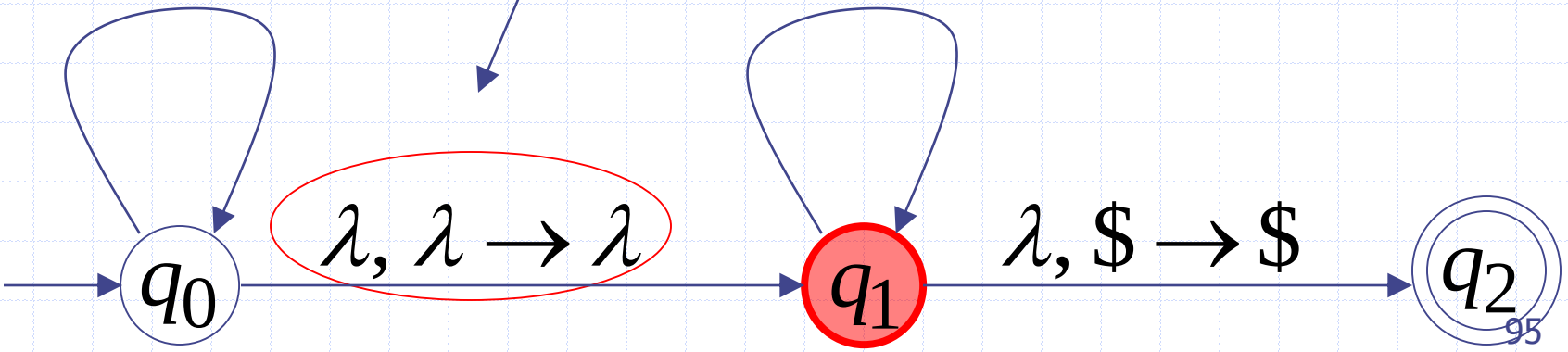
Stack

$a, \lambda \to a$

$b, \lambda \to b$

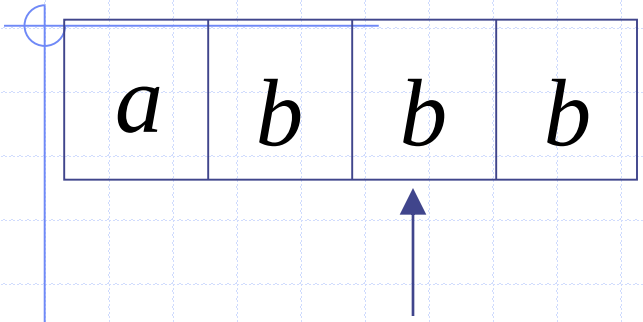
$a, a \to \lambda$

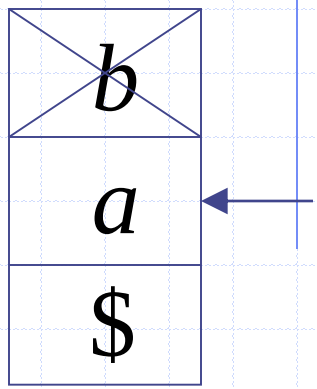
$b, b \to \lambda$



Time 4

Input





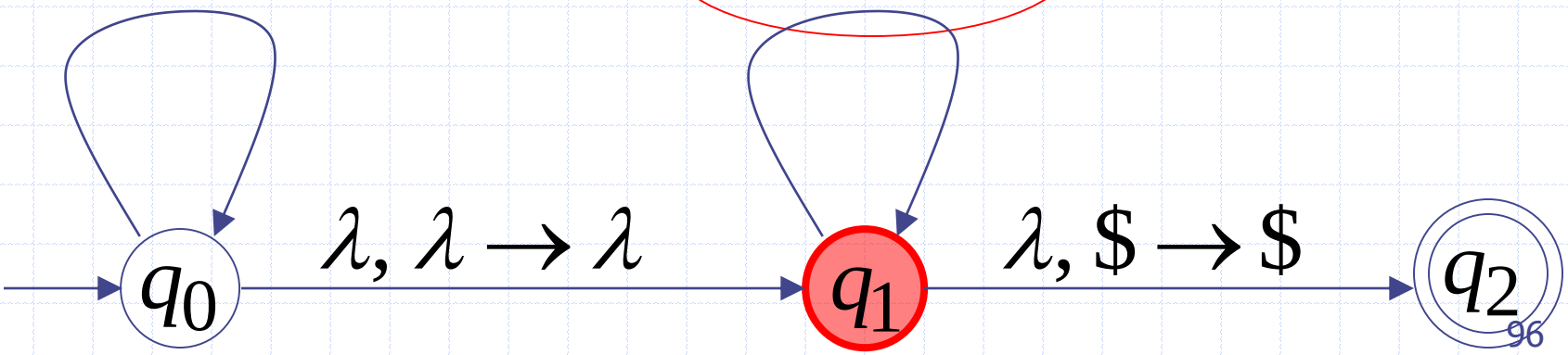
Stack

$a, \lambda \to a$

$b, \lambda \to b$

$a, a \to \lambda$

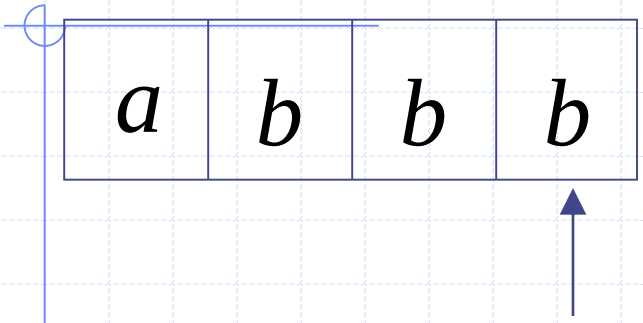
$b, b \to \lambda$



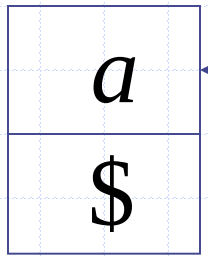
Time 5

There is no possible transition.

Input



Input is not consumed



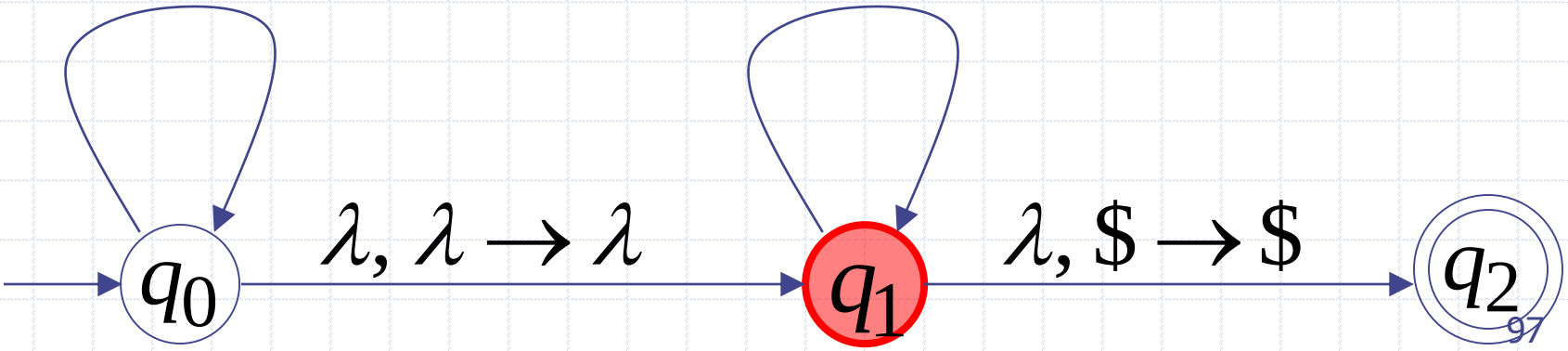
Stack

$a, \lambda \rightarrow a$

$b, \lambda \rightarrow b$

$a, a \rightarrow \lambda$

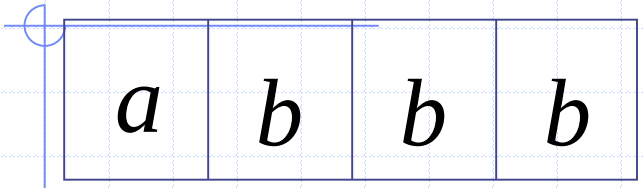
$b, b \rightarrow \lambda$

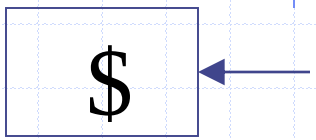


Another computation on same string:

Input

Time 0





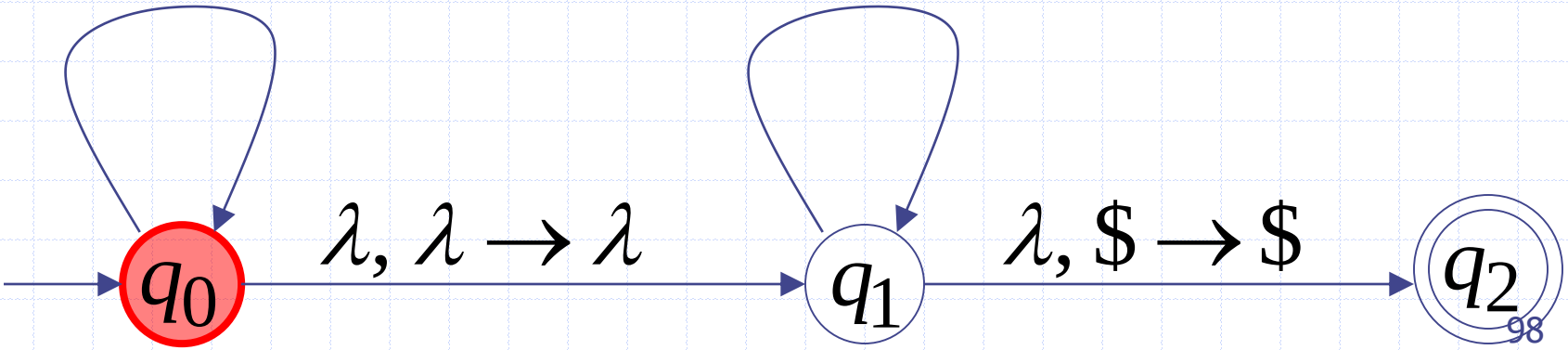
Stack

$a, \lambda \to a$

$b, \lambda \to b$

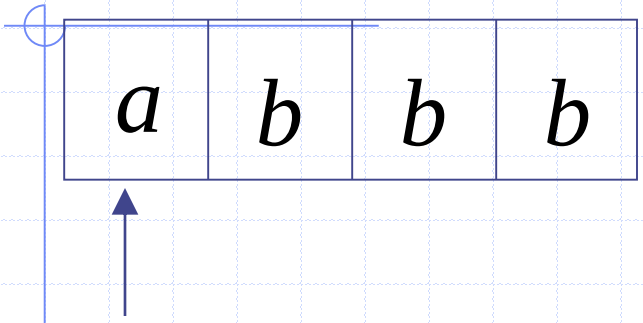
$a, a \to \lambda$

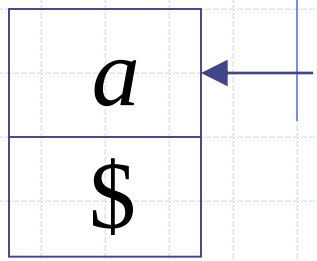
$b, b \to \lambda$



Time 1

Input





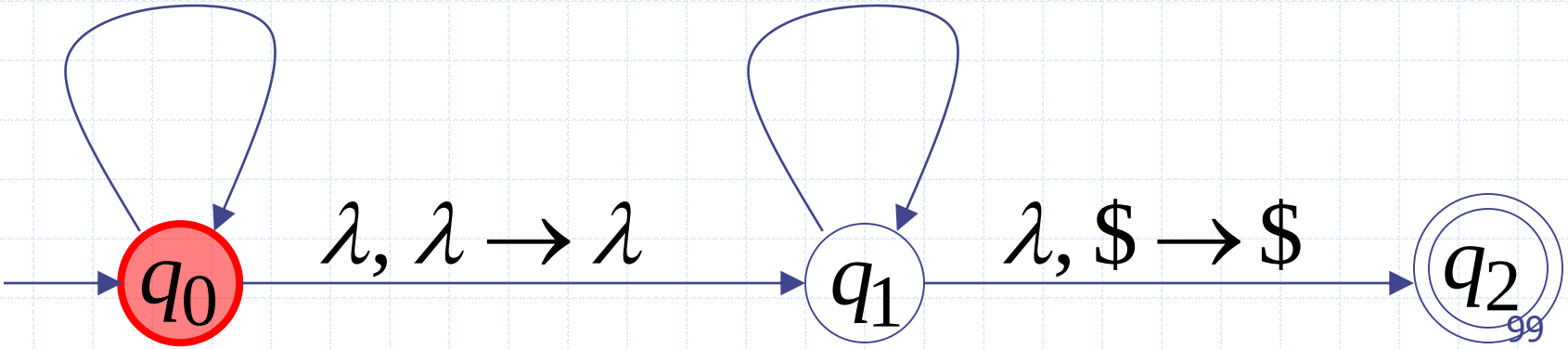
Stack

$a, \lambda \rightarrow a$

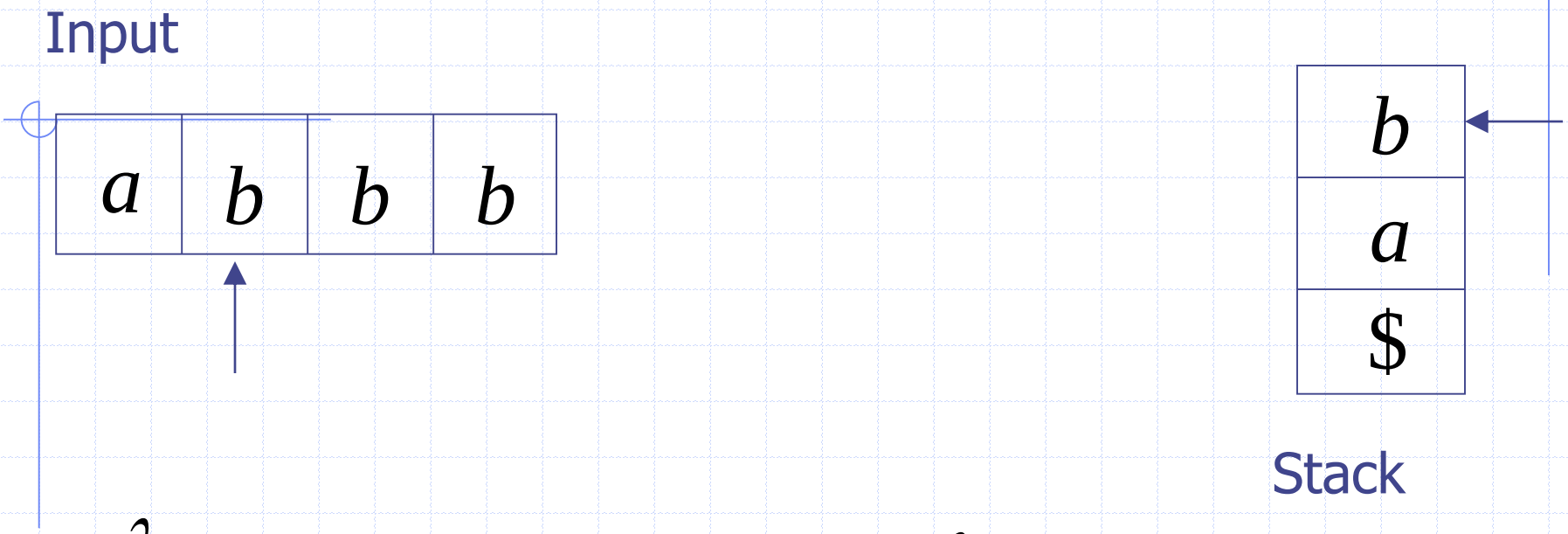
$b, \lambda \rightarrow b$

$a, a \rightarrow \lambda$

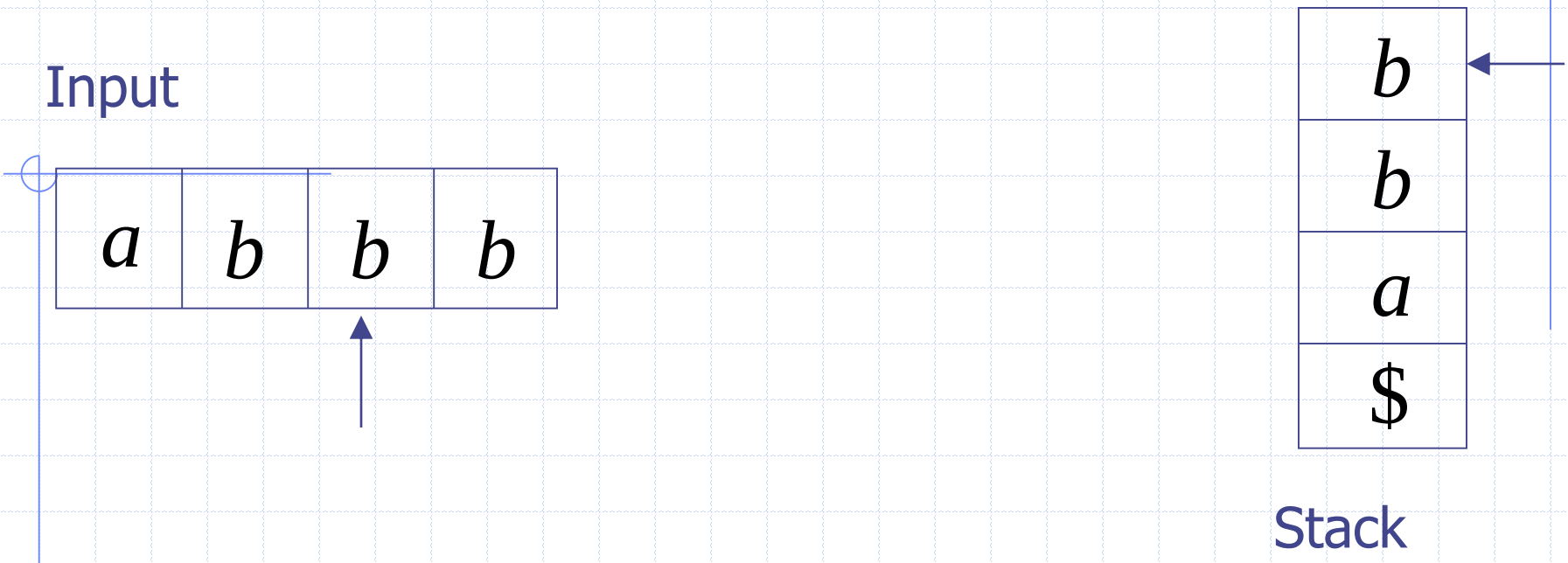
$b, b \rightarrow \lambda$



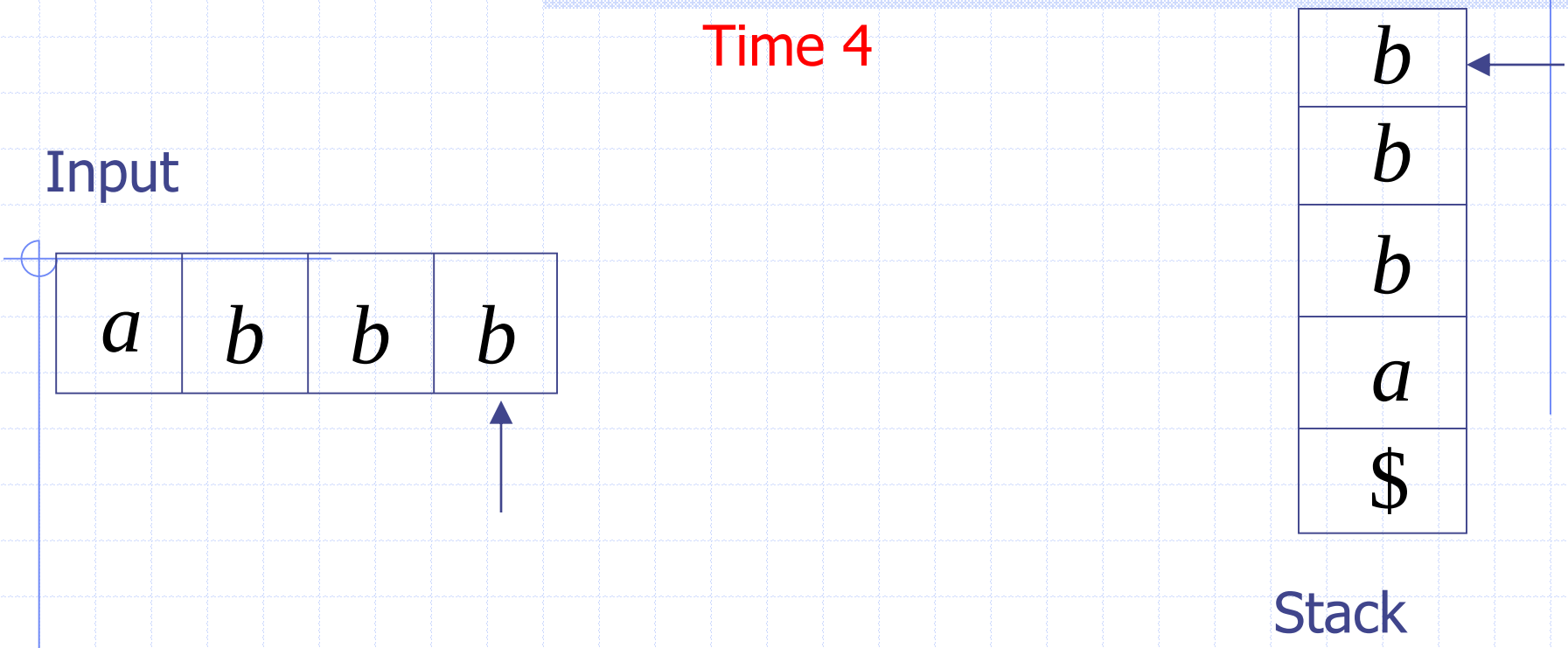
Time 2



Time 3

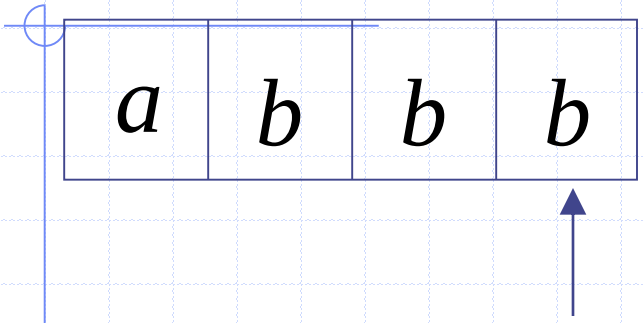


Time 4

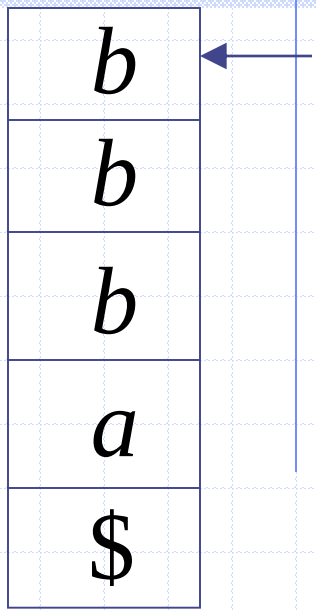

Input
a
b
b
b
b
b
b
a
$
Stack
$a, \lambda \rightarrow a$
$b, \lambda \rightarrow b$
$a, a \rightarrow \lambda$
$b, b \rightarrow \lambda$
q_0
$\lambda, \lambda \rightarrow \lambda$
q_1
$\lambda, \$ \rightarrow \$$
q_2

Time 5

Input



No accept state is reached



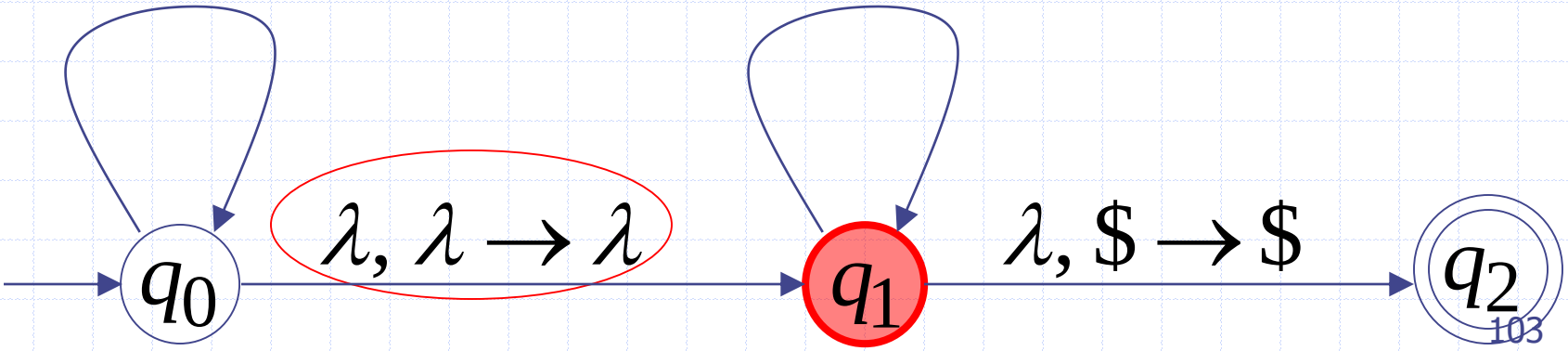
Stack

$a, \lambda \rightarrow a$

$b, \lambda \rightarrow b$

$a, a \rightarrow \lambda$

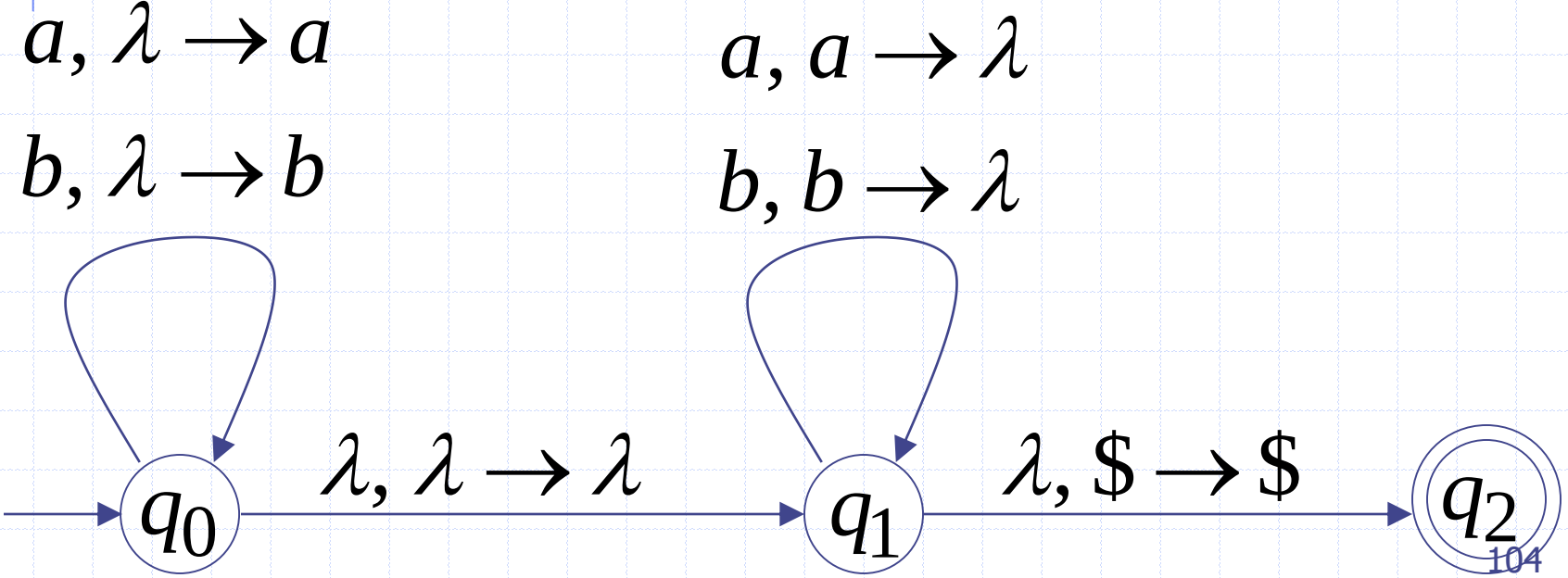
$b, b \rightarrow \lambda$



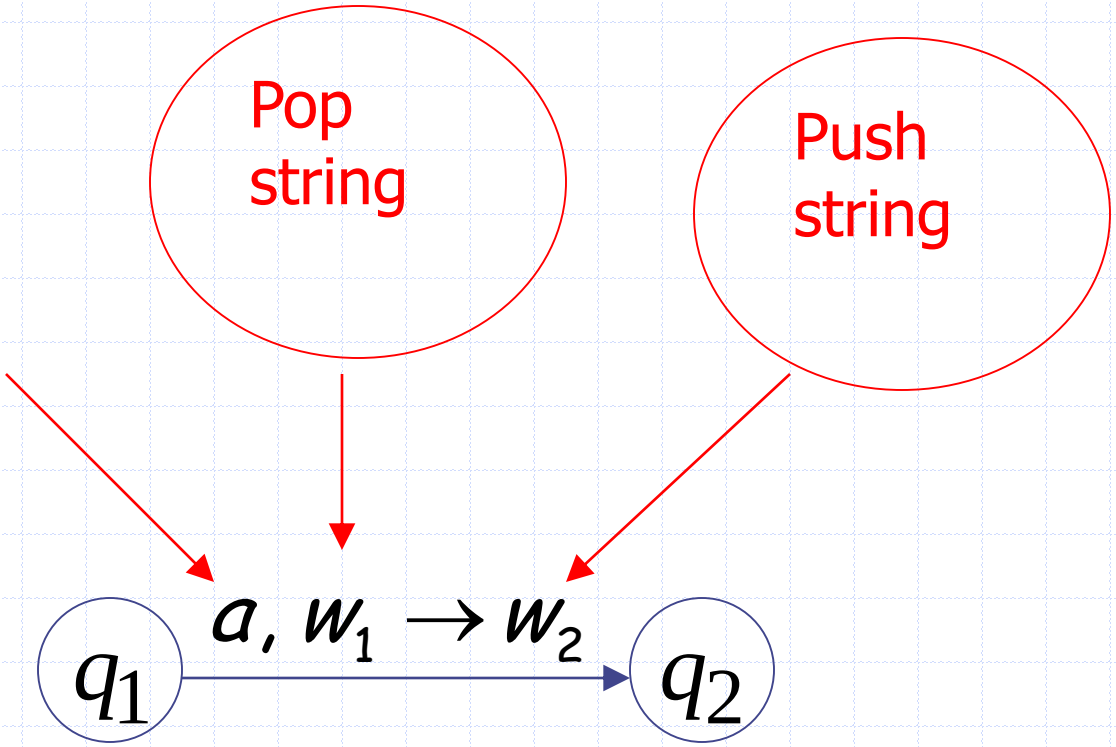
There is no computation that accepts string

$abbb$

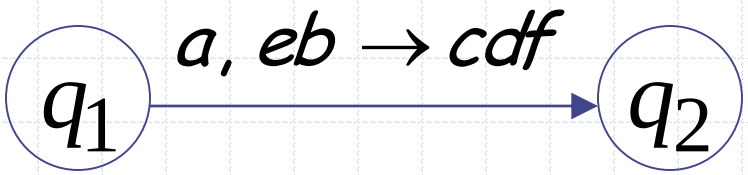
$$abbb \notin L(M)$$

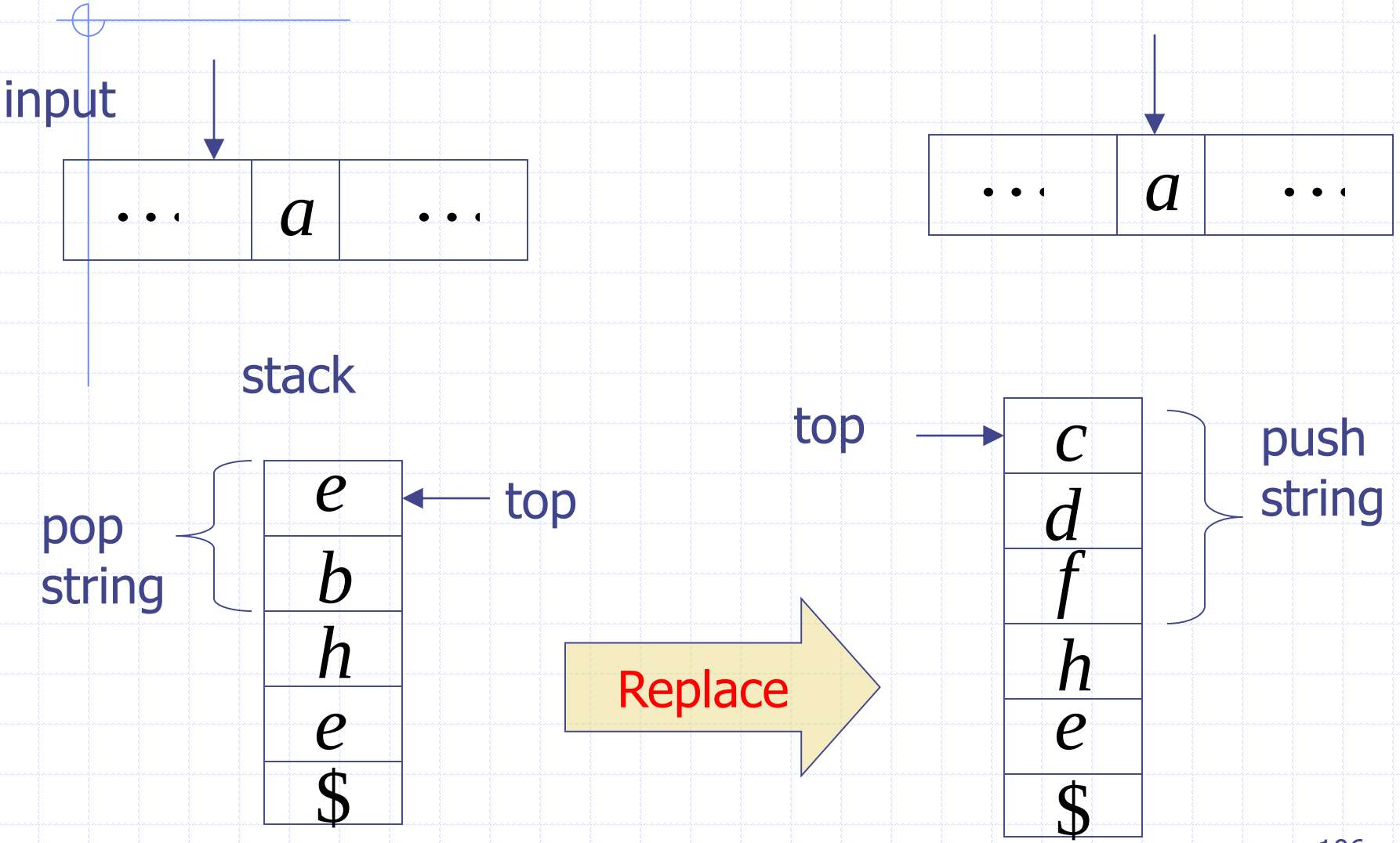


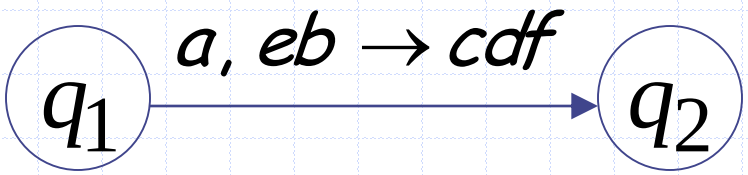
Pushing & Popping Strings



Example:

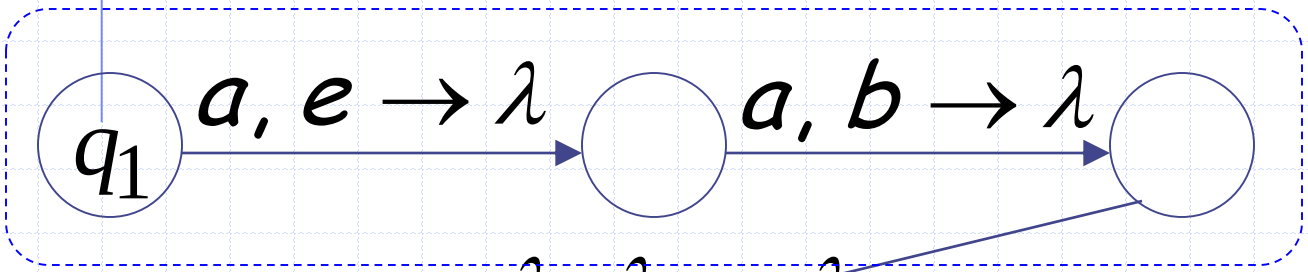

$a, eb \rightarrow cdf$
q_1
q_2


input
$\cdots$ a $\cdots$
$\cdots$ a $\cdots$
stack
top
e
b
h
e
$\$$
pop string
Replace
top
c
d
f
h
e
$\$$
push string



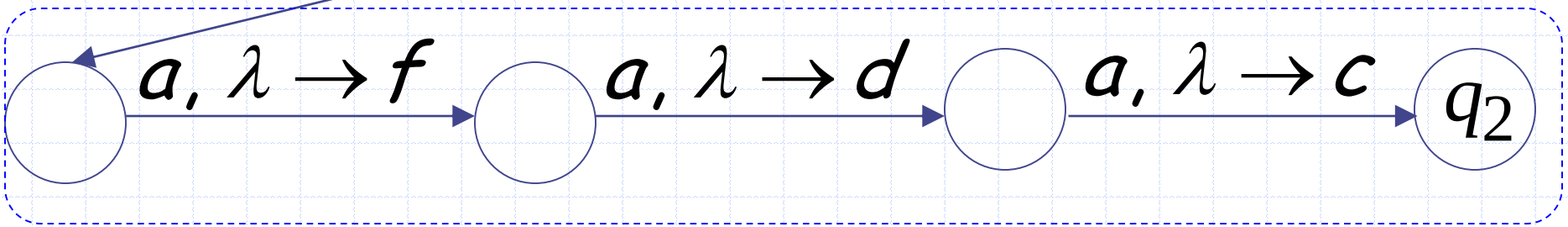
Equivalent transitions

pop



$\lambda, \lambda \rightarrow \lambda$

push



Another PDA example

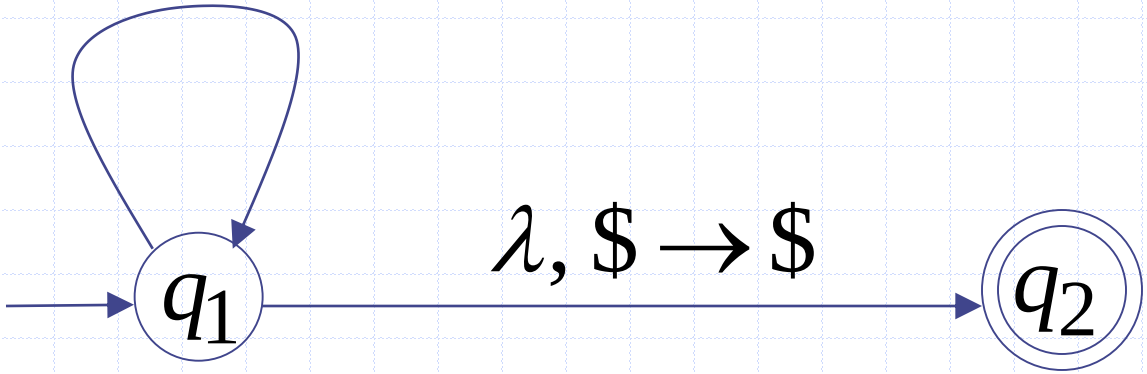
$$L(M) = \{w \in \{a,b\}^* : \quad n_a(w) = n_b(w)\}$$

PDA M

$a, \$ \to 0\$$ $\quad b, \$ \to 1\$$

$a, 0 \to 00$ $\quad b, 1 \to 11$

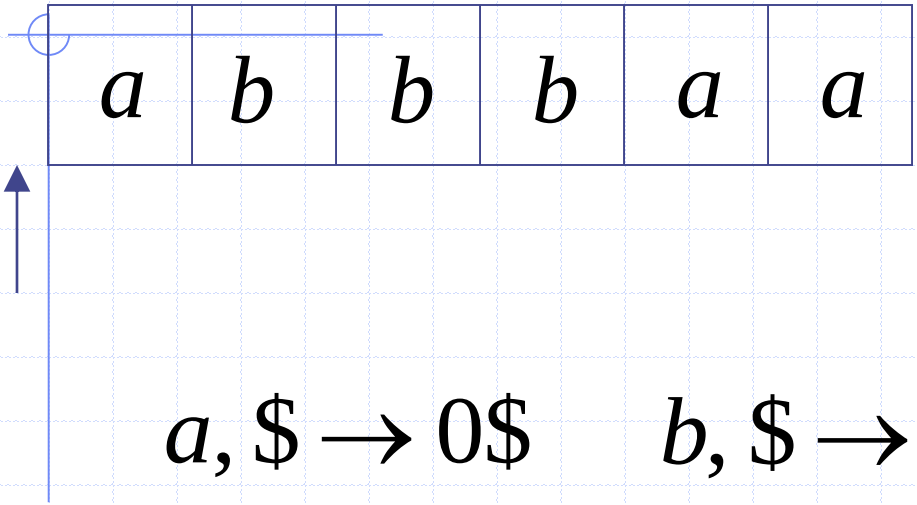
$a, 1 \to \lambda$ $\quad b, 0 \to \lambda$

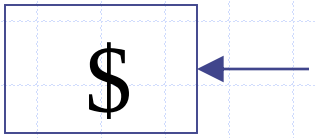


Execution Example:

Time 0

Input





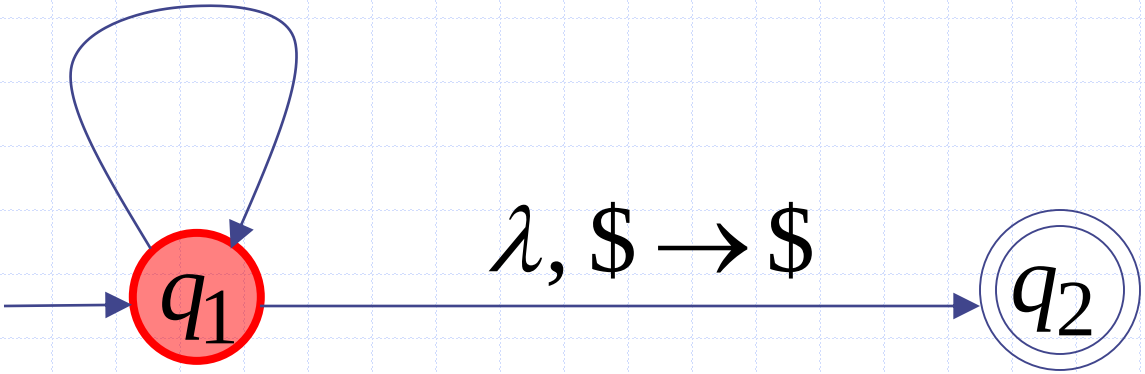
Stack

$a, \$ \rightarrow 0\$$ $b, \$ \rightarrow 1\$$

$a, 0 \rightarrow 00$ $b, 1 \rightarrow 11$

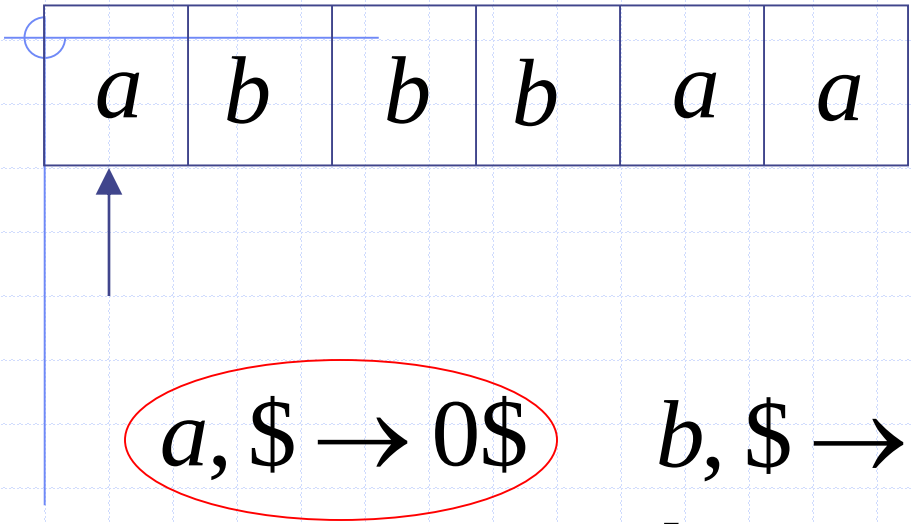
$a, 1 \rightarrow \lambda$ $b, 0 \rightarrow \lambda$

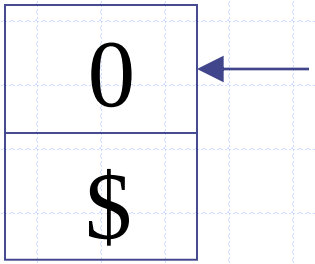
current state



Time 1

Input



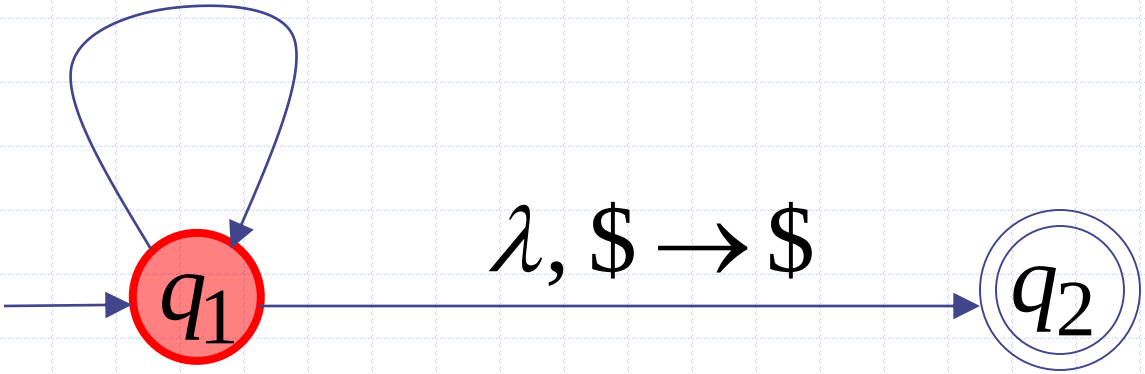


Stack

$a, \$ \rightarrow 0\$$ $b, \$ \rightarrow 1\$$

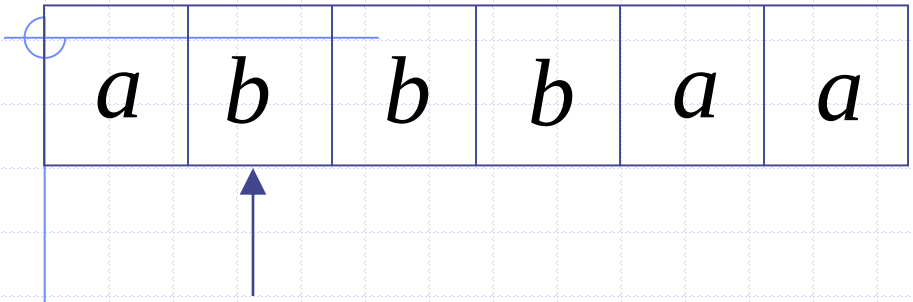
$a, 0 \rightarrow 00$ $b, 1 \rightarrow 11$

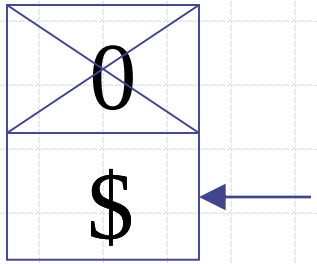
$a, 1 \rightarrow \lambda$ $b, 0 \rightarrow \lambda$



Time 3

Input



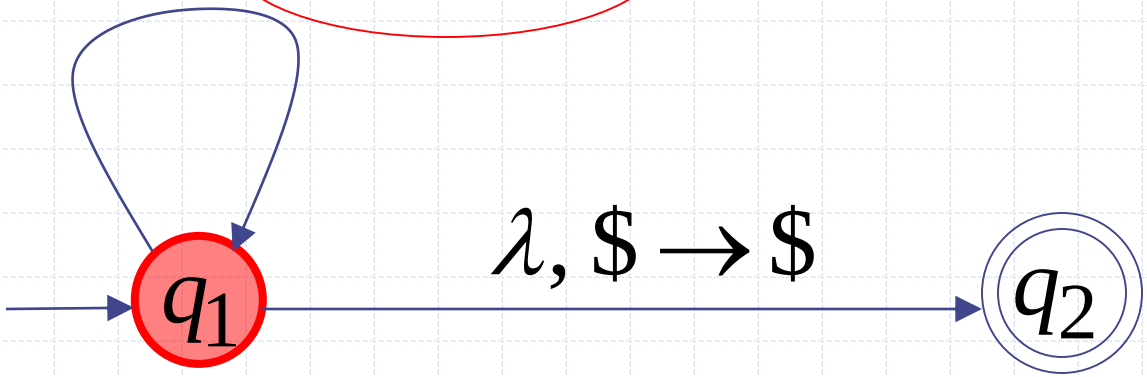


Stack

$a, \$ \rightarrow 0\$$ $b, \$ \rightarrow 1\$$

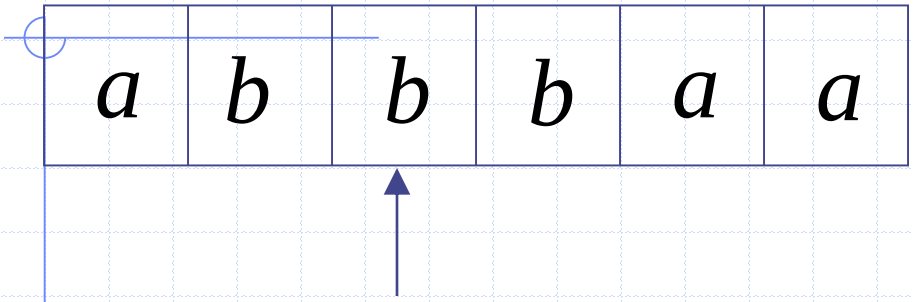
$a, 0 \rightarrow 00$ $b, 1 \rightarrow 11$

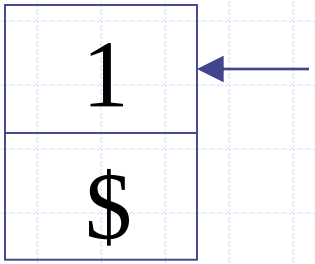
$a, 1 \rightarrow \lambda$ $b, 0 \rightarrow \lambda$



Time 4

Input



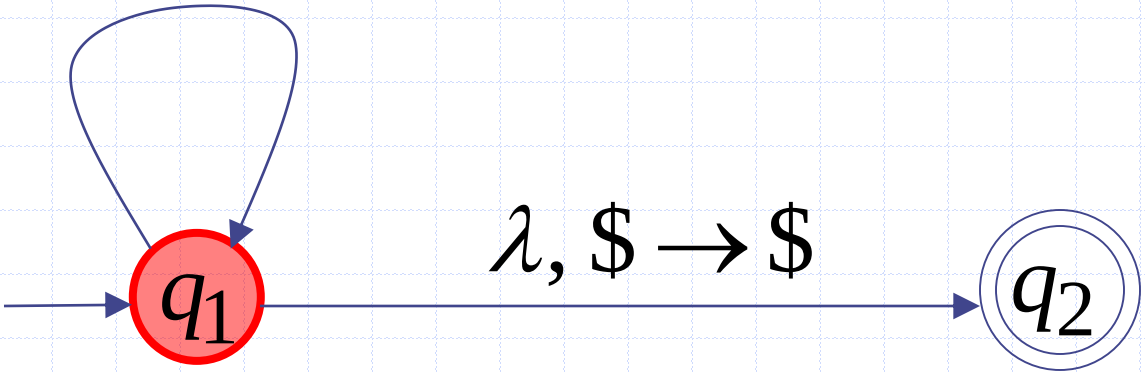


Stack

$a, \$ \rightarrow 0\$$ $b, \$ \rightarrow 1\$$

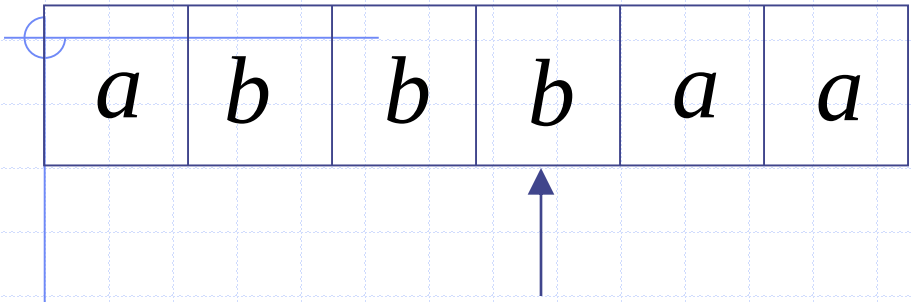
$a, 0 \rightarrow 00$ $b, 1 \rightarrow 11$

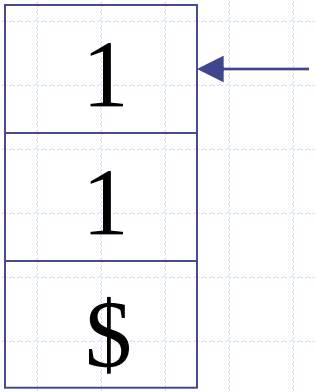
$a, 1 \rightarrow \lambda$ $b, 0 \rightarrow \lambda$



Time 5

Input



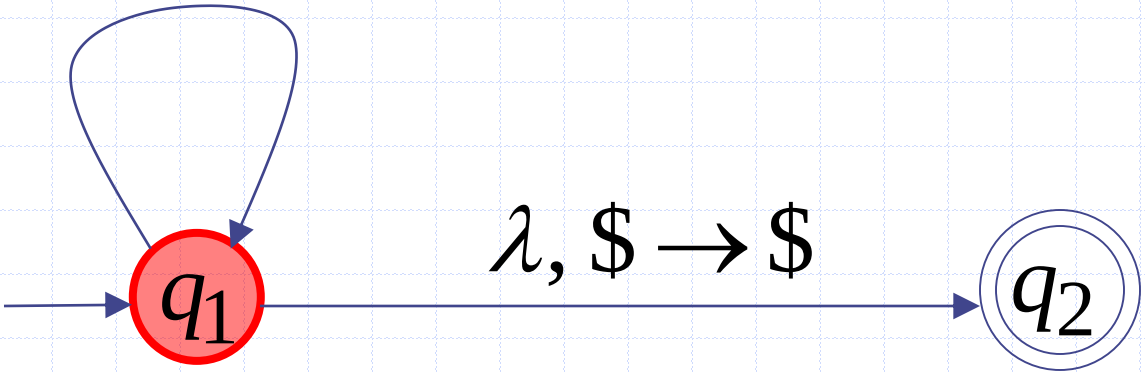


Stack

$a, \$ \rightarrow 0\$$ $b, \$ \rightarrow 1\$$

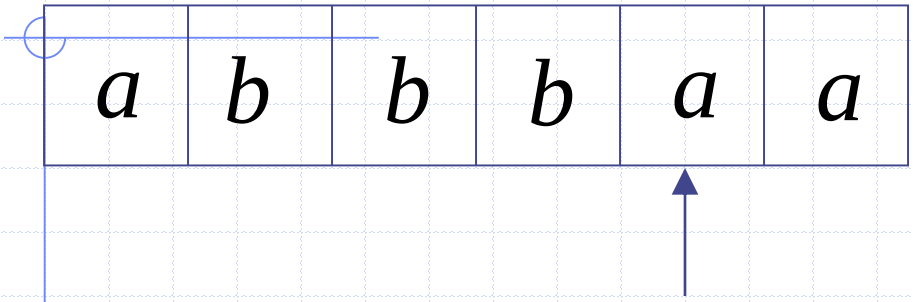
$a, 0 \rightarrow 00$ $b, 1 \rightarrow 11$

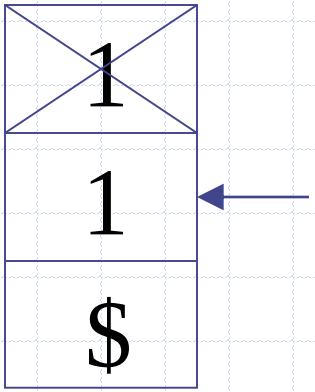
$a, 1 \rightarrow \lambda$ $b, 0 \rightarrow \lambda$



Time 6

Input



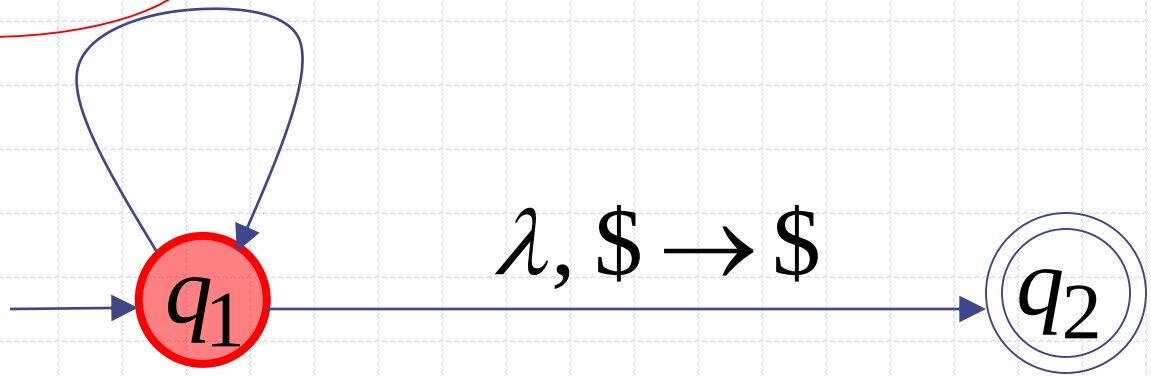


Stack

$a, \$ \rightarrow 0\$$ $b, \$ \rightarrow 1\$$

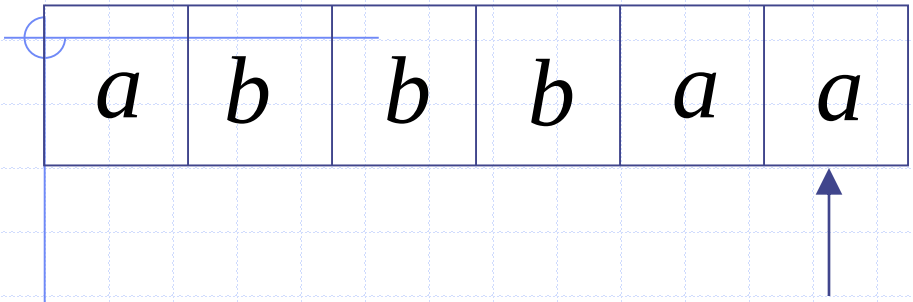
$a, 0 \rightarrow 00$ $b, 1 \rightarrow 11$

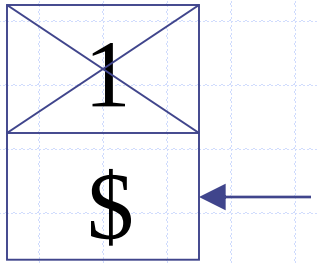
$a, 1 \rightarrow \lambda$ $b, 0 \rightarrow \lambda$



Time 7

Input



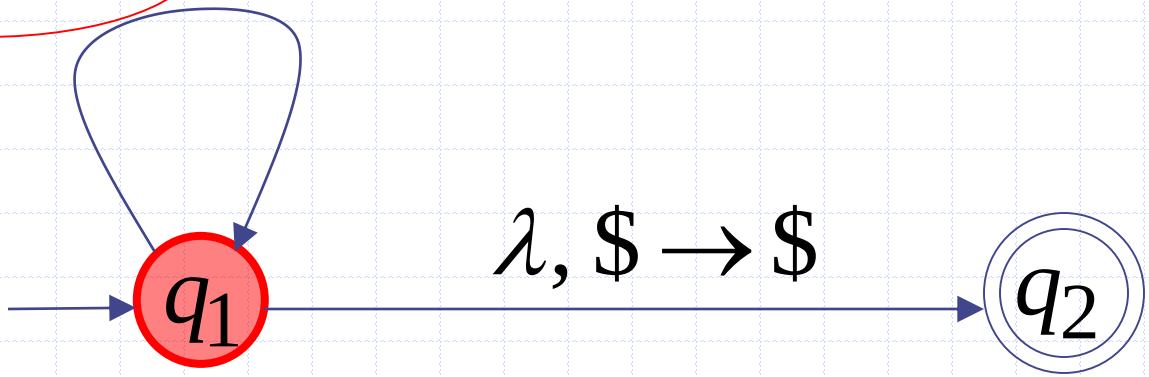


Stack

$a, \$ \rightarrow 0\$$ $b, \$ \rightarrow 1\$$

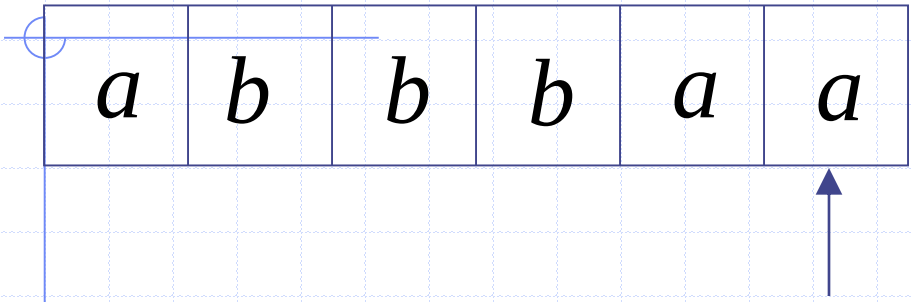
$a, 0 \rightarrow 00$ $b, 1 \rightarrow 11$

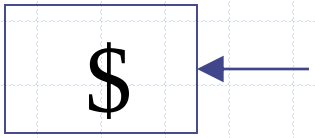
$a, 1 \rightarrow \lambda$ $b, 0 \rightarrow \lambda$



Time 8

Input



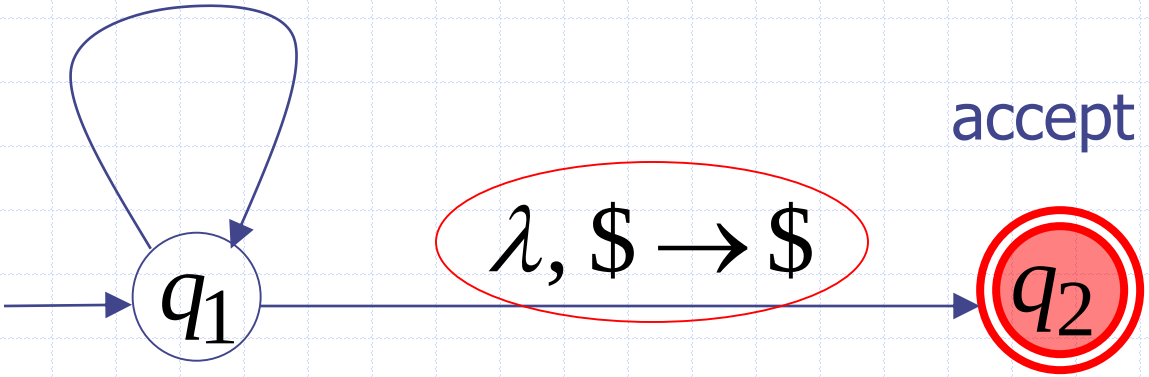


Stack

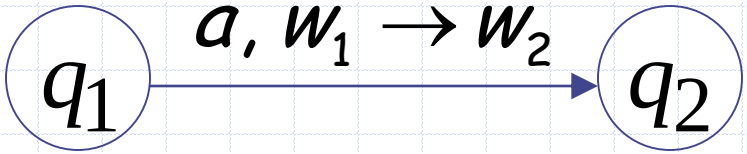
$a, \$ \rightarrow 0\$$ $b, \$ \rightarrow 1\$$

$a, 0 \rightarrow 00$ $b, 1 \rightarrow 11$

$a, 1 \rightarrow \lambda$ $b, 0 \rightarrow \lambda$

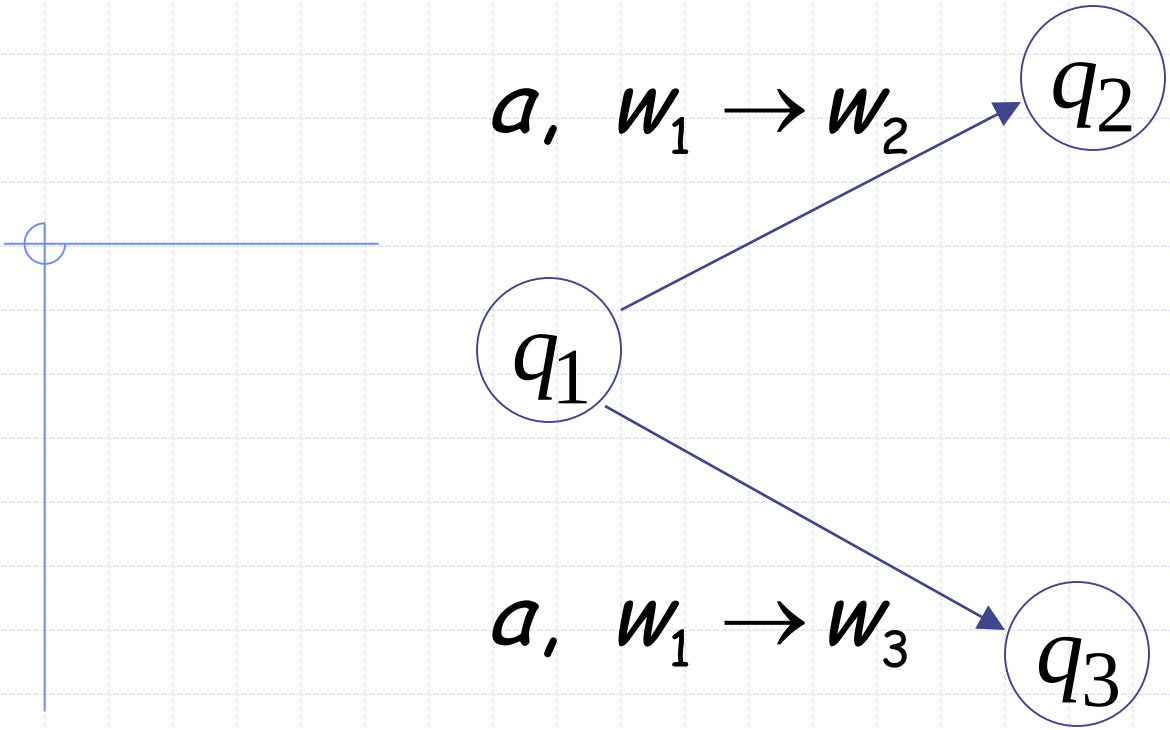


Formalities for PDAs



Transition function:

$$\delta(q_1, a, w_1) = \{(q_2, w_2)\}$$



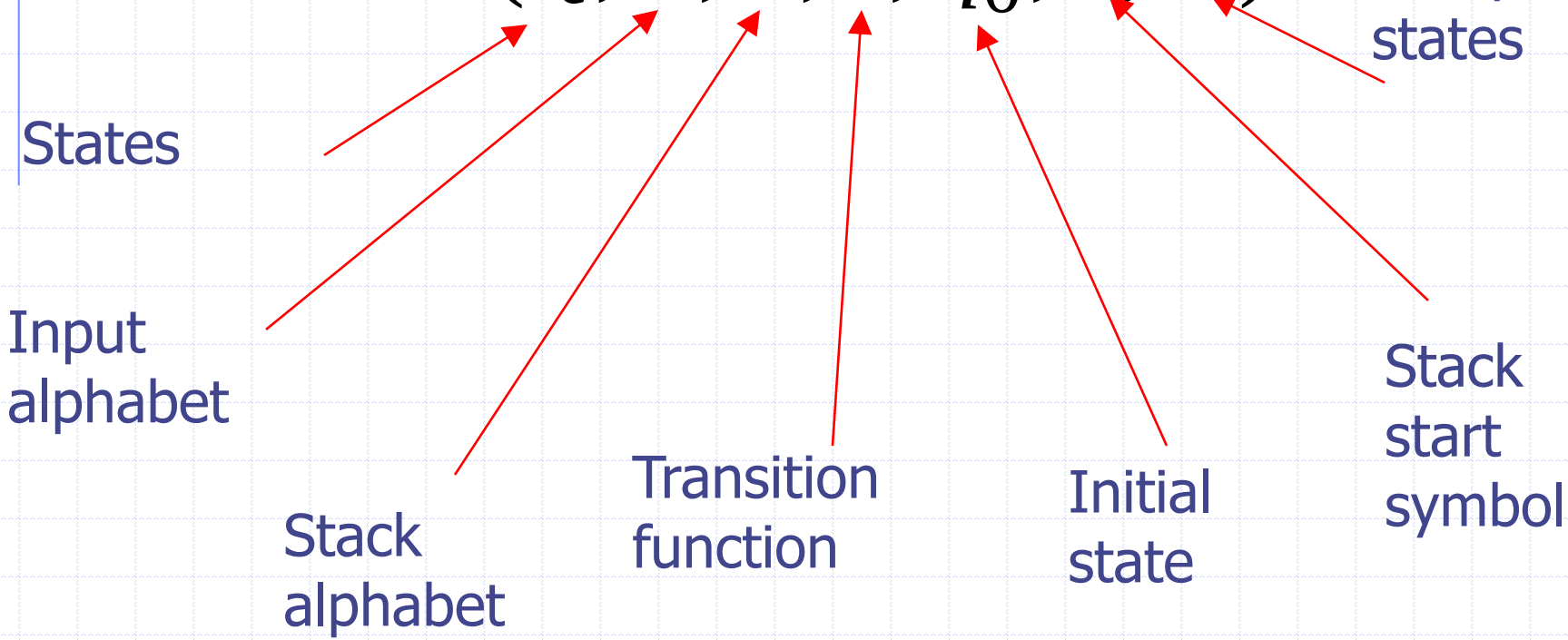
Transition function:

$$\delta(q_1, a, w_1) = \{(q_2, w_2),\ (q_3, w_3)\}$$

Formal Definition

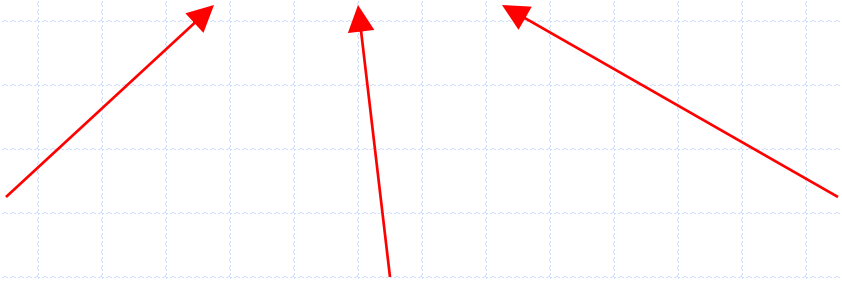
Pushdown Automaton (PDA)

$$M = (Q, \Sigma, \Gamma, \delta, q_0, z, F)$$



Instantaneous Description

$$(q,\ u,\ s)$$



Current state

Remaining input

Current stack contents

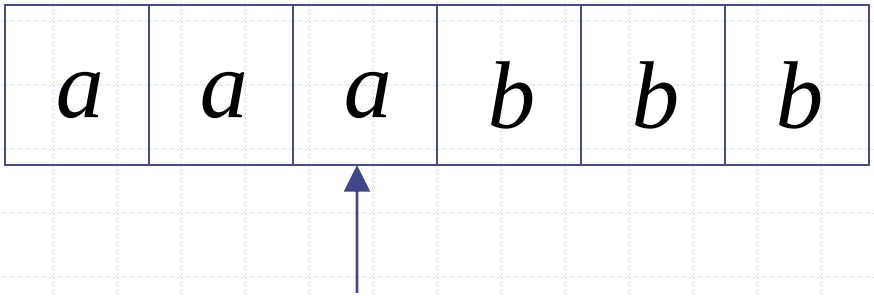
Example:

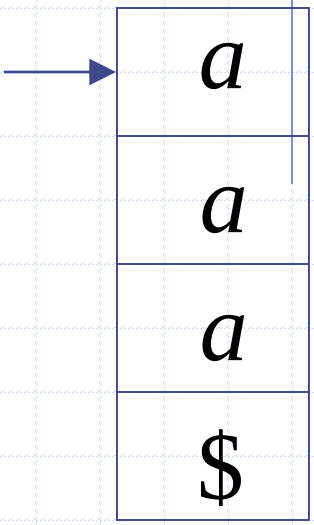
Instantaneous Description

$$(q_1, bbb, aaa\$)$$

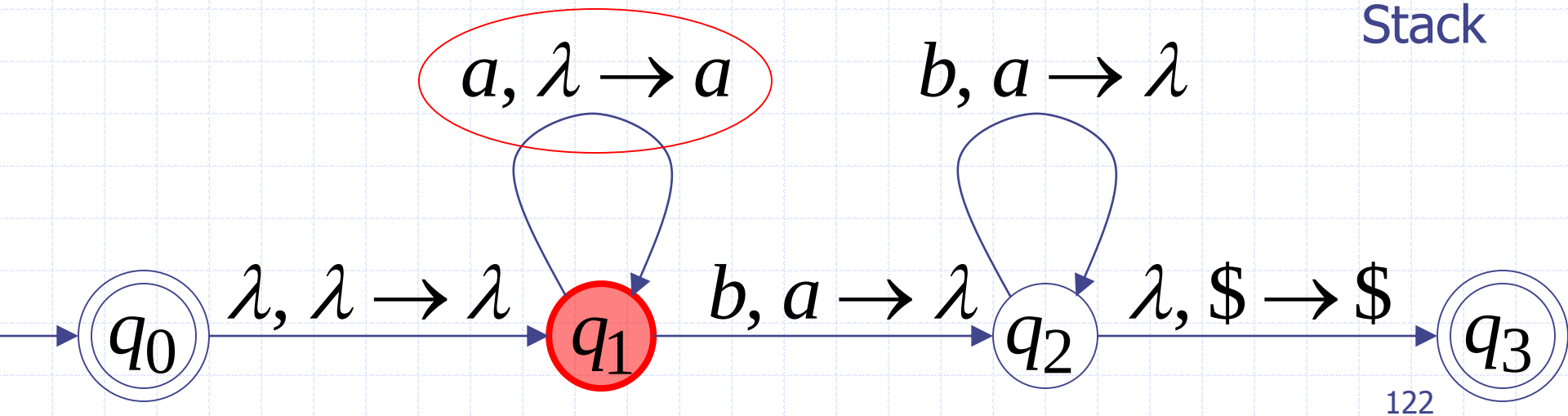
Time 4:

Input





Stack



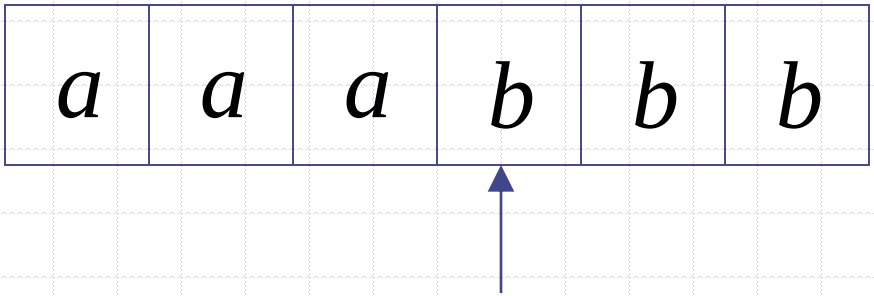
Example:

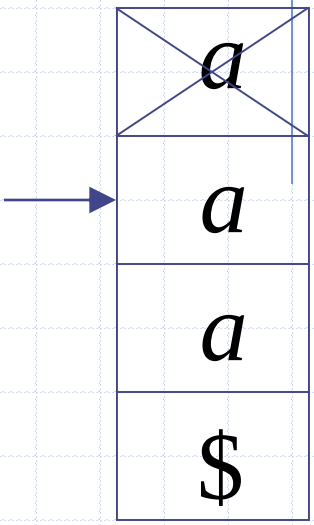
Instantaneous Description

$$(q_2, bb, aa\$)$$

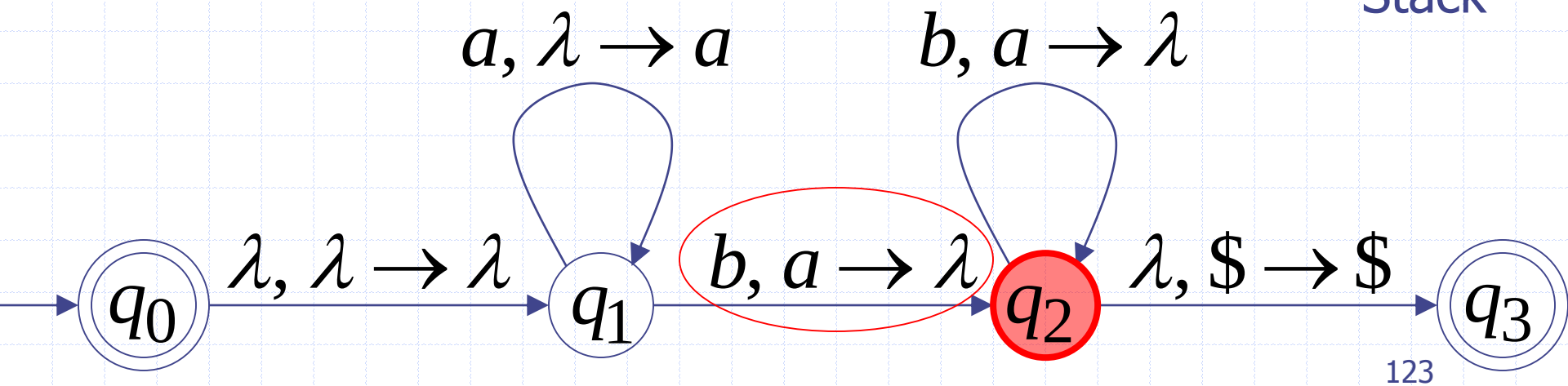
Time 5:

Input





Stack



We write:

$$(q_1, bbb, aaa\$) \succ (q_2, bb, aa\$)$$

Time 4

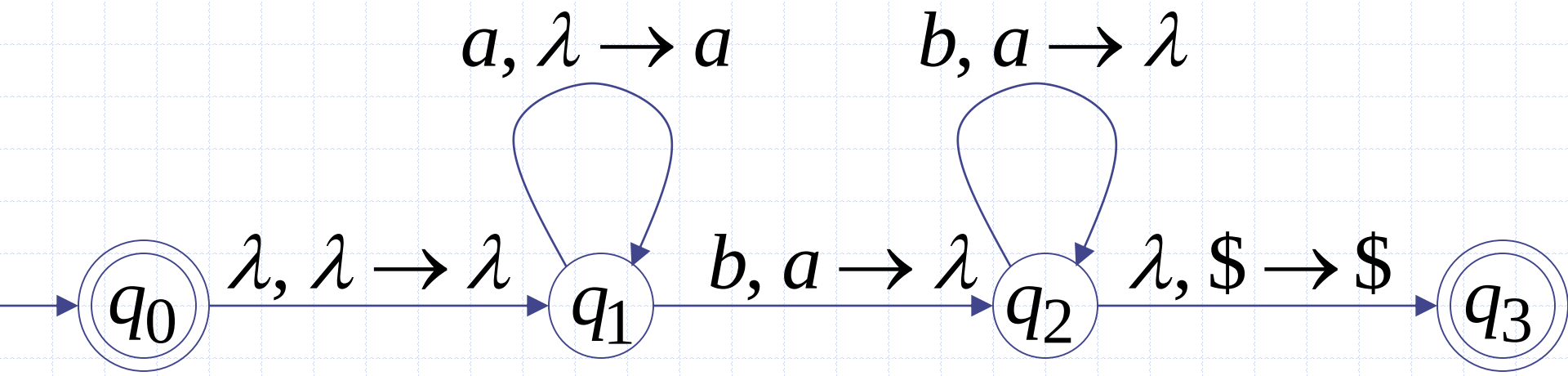
Time 5

A computation:

$$(q_0, aaabbb, \$) \succ (q_1, aaabbb, \$) \succ$$

$$(q_1, aabbb, a\$) \succ (q_1, abbb, aa\$) \succ (q_1, bbb, aaa\$) \succ$$

$$(q_2, bb, aa\$) \succ (q_2, b, a\$) \succ (q_2, \lambda, \$) \succ (q_3, \lambda, \$)$$



$$(q_0, aaabbb, \$) \succ (q_1, aaabbb, \$) \succ$$

$$(q_1, aabbb, a\$) \succ (q_1, abbb, aa\$) \succ (q_1, bbb, aaa\$) \succ$$

$$(q_2, bb, aa\$) \succ (q_2, b, a\$) \succ (q_2, \lambda, \$) \succ (q_3, \lambda, \$)$$

For convenience we write:

$$(q_0, aaabbb, \$) \overset{*}{\succ} (q_3, \lambda, \$)$$

Language of PDA

Language $L(M)$ accepted by PDA : M

$$L(M) = \{w : \ (q_0, w, z) \overset{*}{\succ} (q_f, \lambda, s)\}$$

Initial state

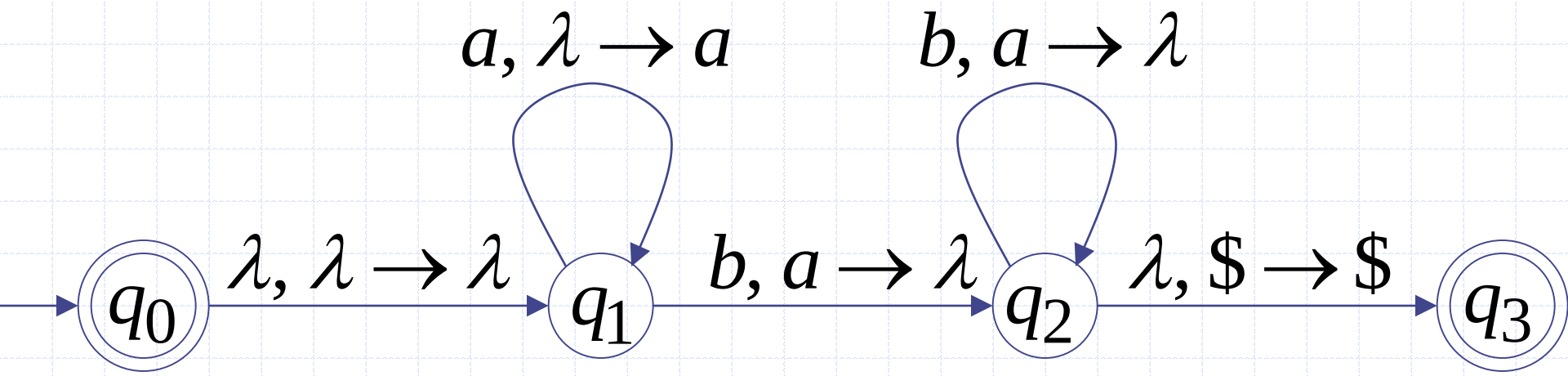
Accept state

Example:

$$(q_0, aaabbb, \$) \overset{*}{\succ} (q_3, \lambda, \$)$$

$$aaabbb \in L(M)$$

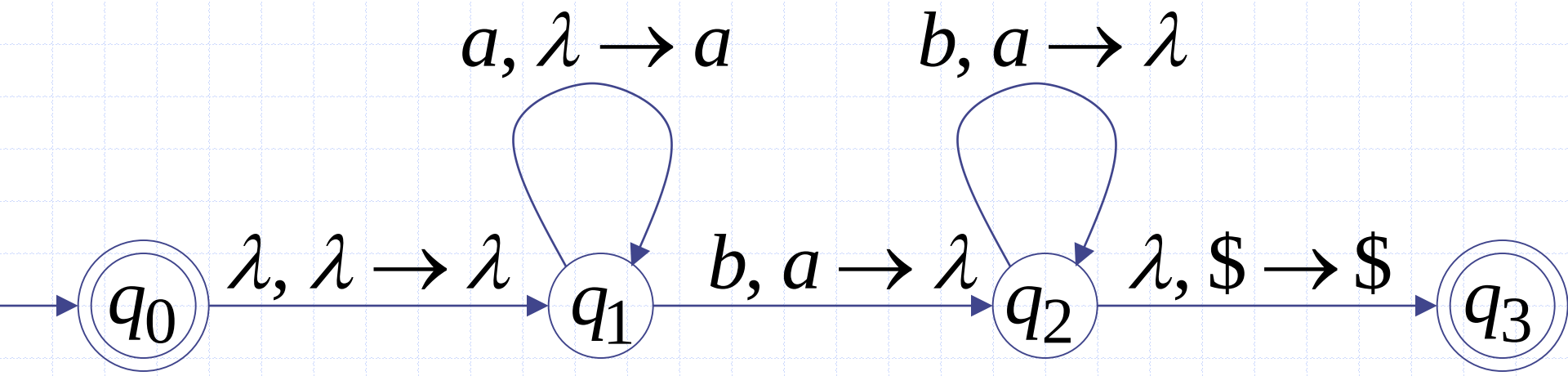
PDA M:



$$(q_0, a^n b^n, \$) \overset{*}{\succ} (q_3, \lambda, \$)$$

$$a^n b^n \in L(M)$$

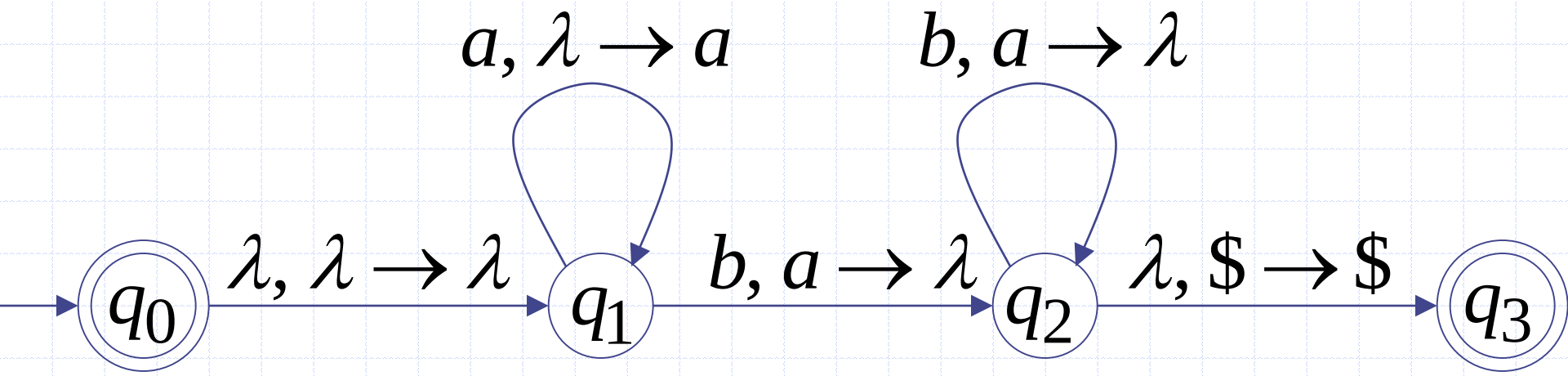
PDA M :



Therefore:

$$L(M) = \{a^n b^n : n \geq 0\}$$

PDA M :



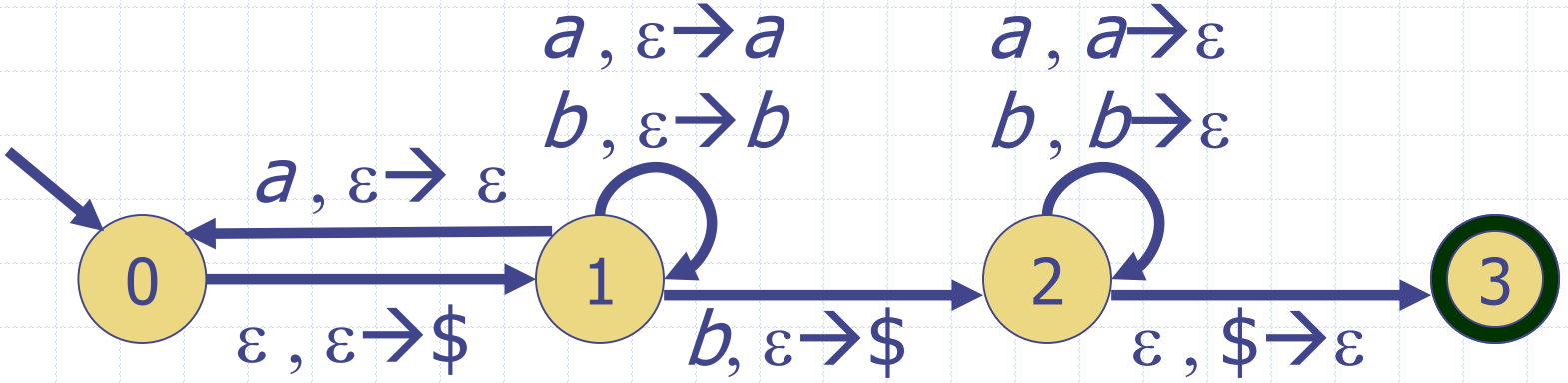
PDA
Formal Definition

DEF: A ***pushdown automaton*** (PDA) is a 6-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$. Q, Σ, and q_0, are the same as for an FA. Γ is the ***tape alphabet***. δ is as follows:

$$\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to P(Q \times \Gamma_\varepsilon)$$

So given a state p, an input letter x and a tape letter y, $\delta(p,x,y)$ gives all (q,z) where q is a target state and z a stack replacement for y.

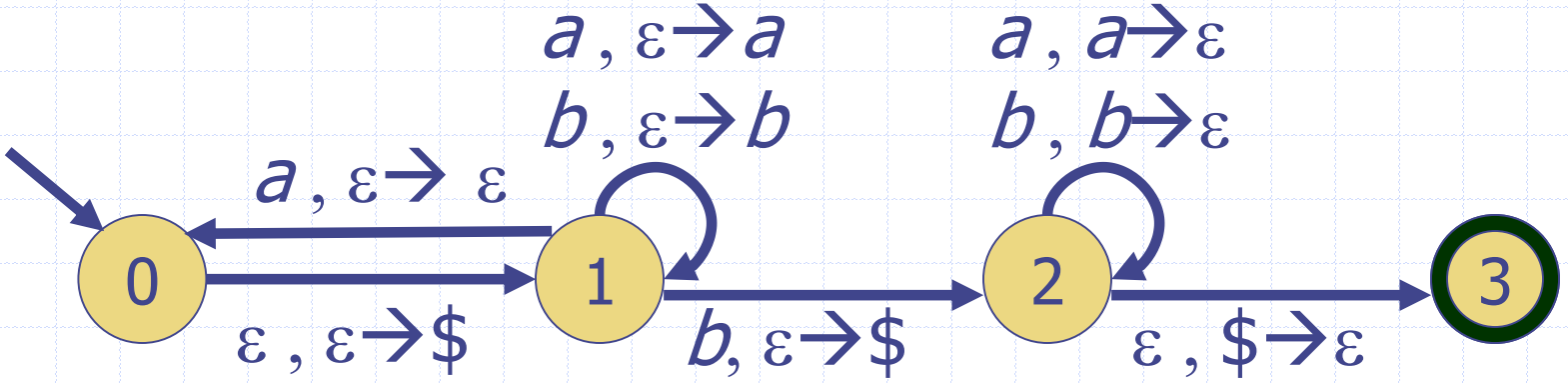
PDA
Formal Definition



Q: What is $\delta(p,x,y)$ in each case?

1. $\delta(0,a,b)$
2. $\delta(0,\varepsilon,\varepsilon)$
3. $\delta(1,a,\varepsilon)$
4. $\delta(3,\varepsilon,\varepsilon)$

PDA
Formal Definition



A:

1. $\delta(0,a,b) = \varnothing$
2. $\delta(0,\varepsilon,\varepsilon) = \{(1,\$)\}$
3. $\delta(1,a,\varepsilon) = \{(0,\varepsilon),(1,a)\}$
4. $\delta(3,\varepsilon,\varepsilon) = \varnothing$