CMPU 240 · Theory of Computation

# Pushdown Automata

24 March 2026

VASSAR COLLEGE A·D·1861

## Assignment 5

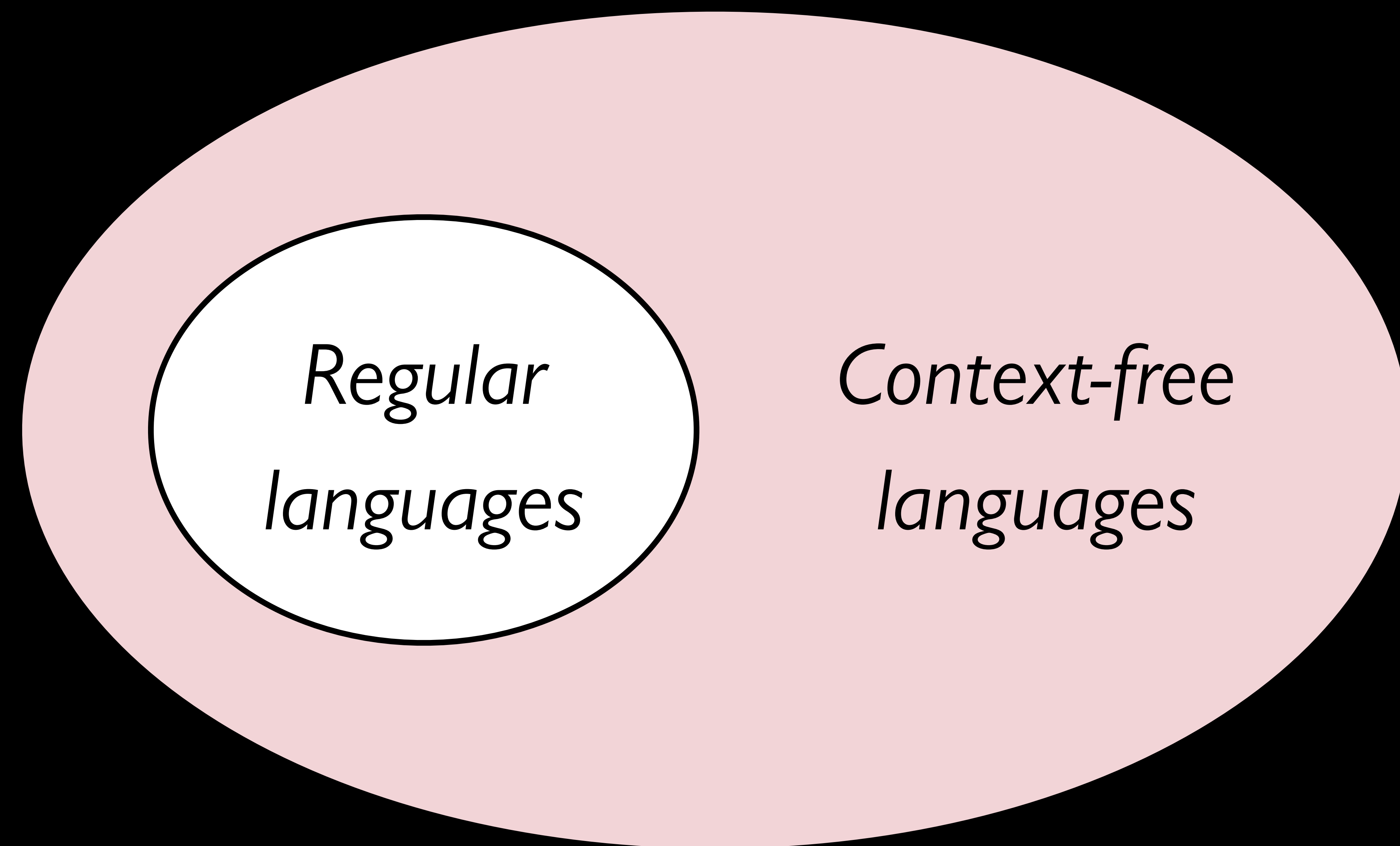
- Due today
- Corrections due Thursday

## Exam 1

- Review the example solutions

# Where are we?

Our study of the *regular languages* gives us an exact characterization of problems that can be solved by real (finite) computers.



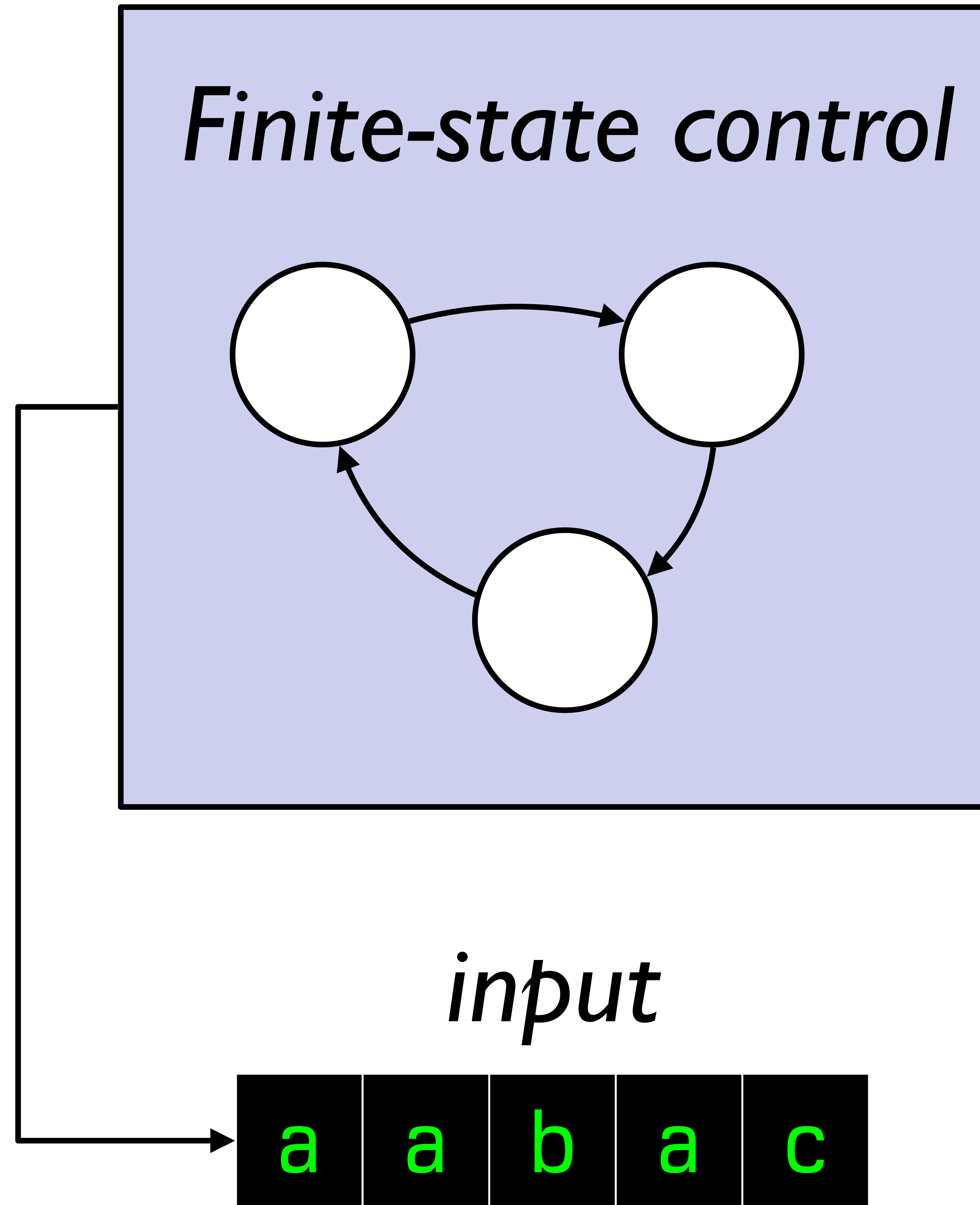

All languages

We can describe context-free languages – including nonregular languages like $\{a^n b^n \mid n \in \mathbb{N}_0\}$ – using context-free grammars.

Grammars are language *generators*.

It’s not immediately clear how they might be used to *recognize* a language – though we’ll return to that question!

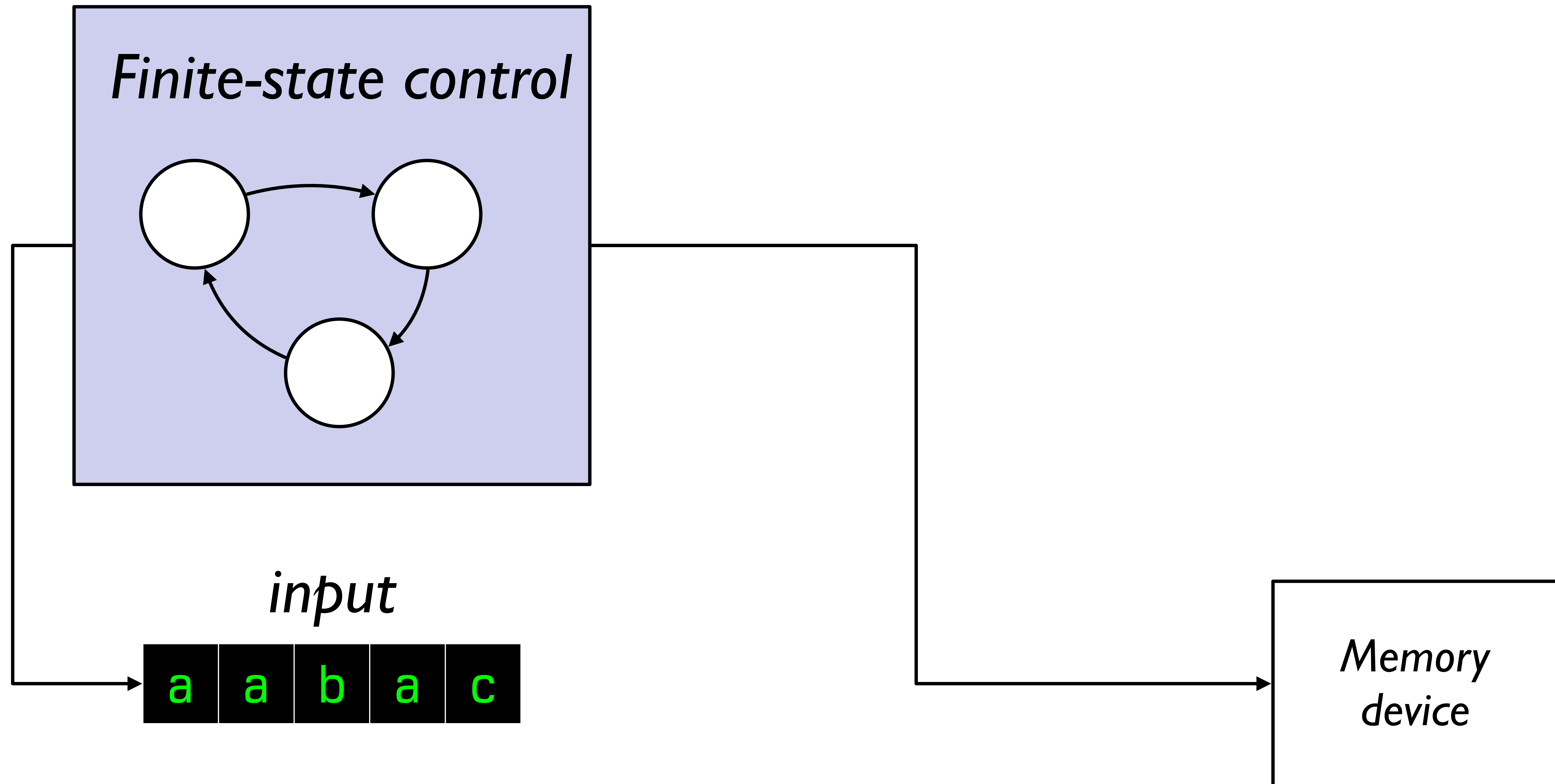
# Pushdown automata


Finite-state control
input
a
a
b
a
c

We may need ***unbounded memory*** to recognize context-free languages.

For example, $\{\mathtt{0}^n\mathtt{1}^n \mid n \in \mathbb{N}_0\}$ requires unbounded counting

# Schematic of a finite automaton



Now the finite automaton can base its transition on both

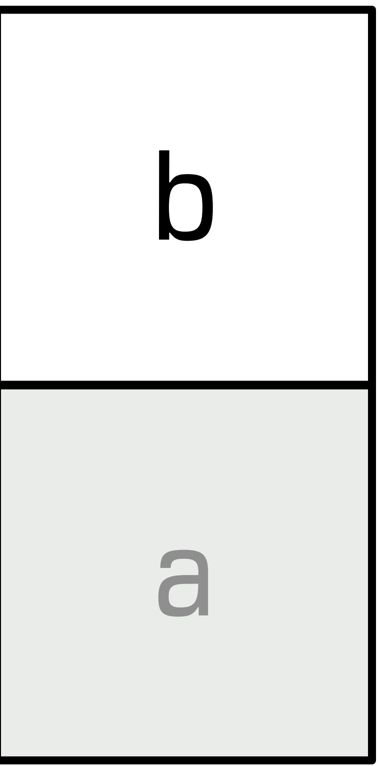
the current symbol being read and

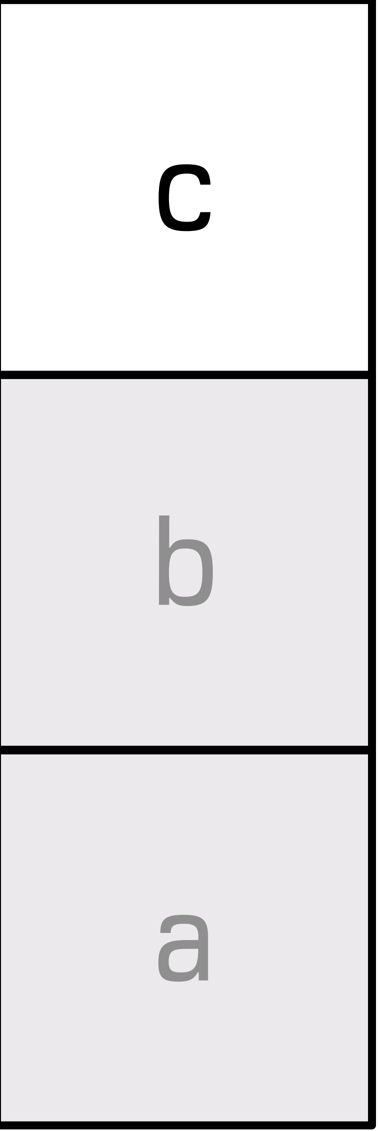
values stored in memory.

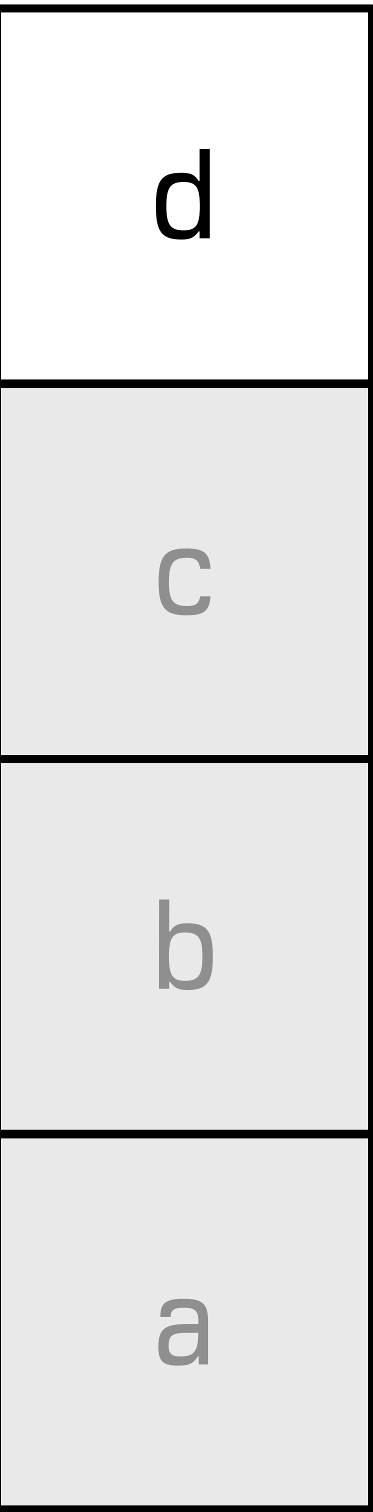
The finite automaton can issue commands to read or write from this memory whenever it makes a transition.

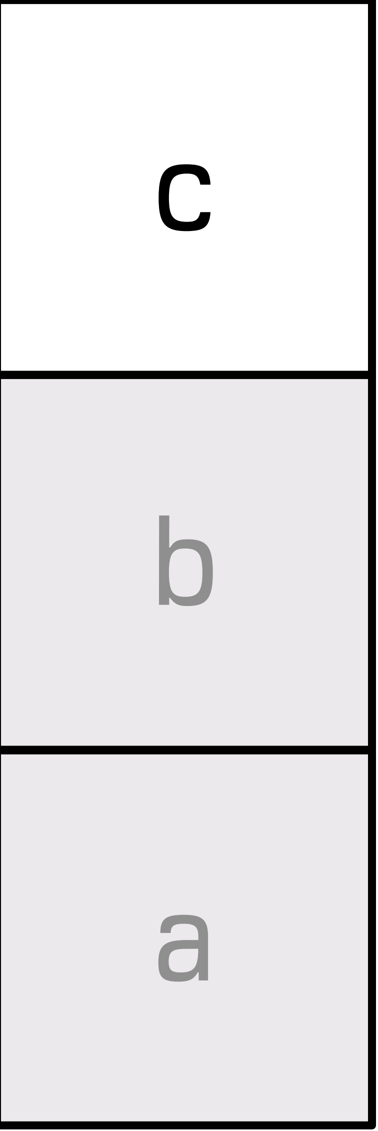
There are many types of memory we might give to an automaton, but one of the simplest is a *stack*.

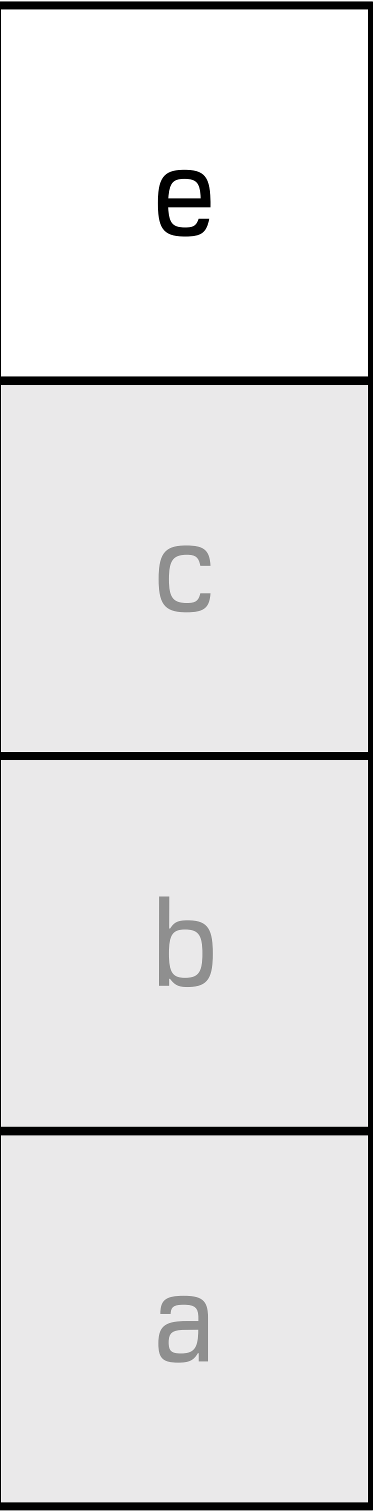
a

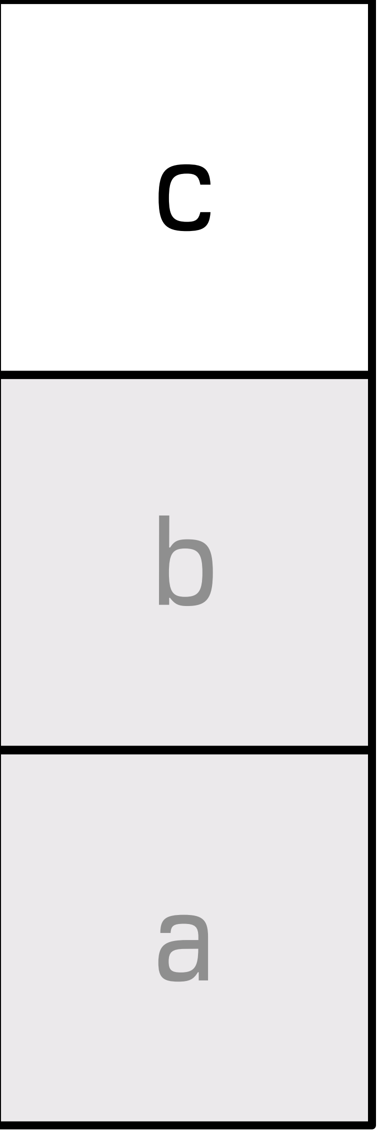

b
a

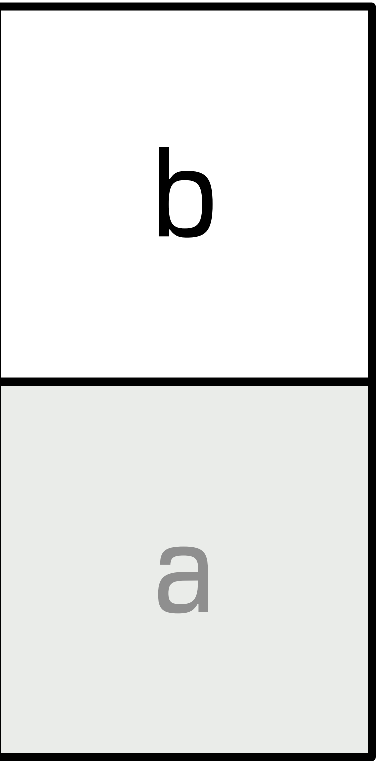

c
b
a


d
c
b
a


c
b
a


e
c
b
a


c
b
a


b
a

Only the top of the stack is visible at any point in time.

New symbols can be *pushed* onto the stack, rendering the previous top symbol inaccessible.

The top symbol of the stack may be *popped*, exposing the symbol below it.

A *pushdown automaton* (PDA) is a finite automaton equipped with stack-based memory.

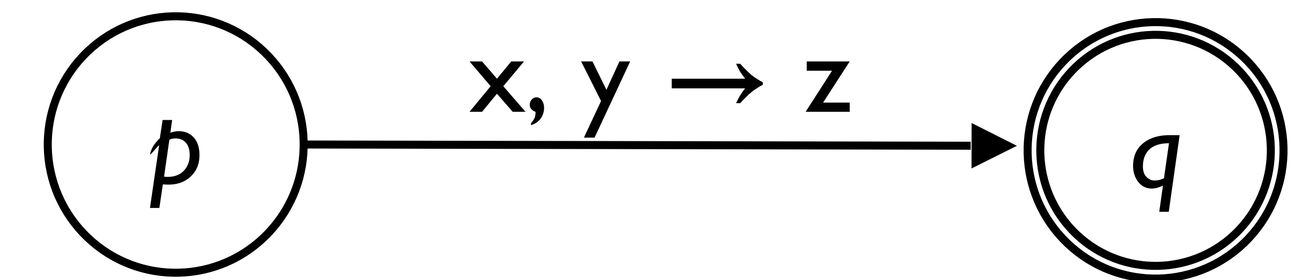
# Notation

At state $p$, if you can

read the symbol $x$ from the input and
pop the symbol $y$ from the stack,

then you can

enter state $q$ and
push the symbol $z$ onto the stack.



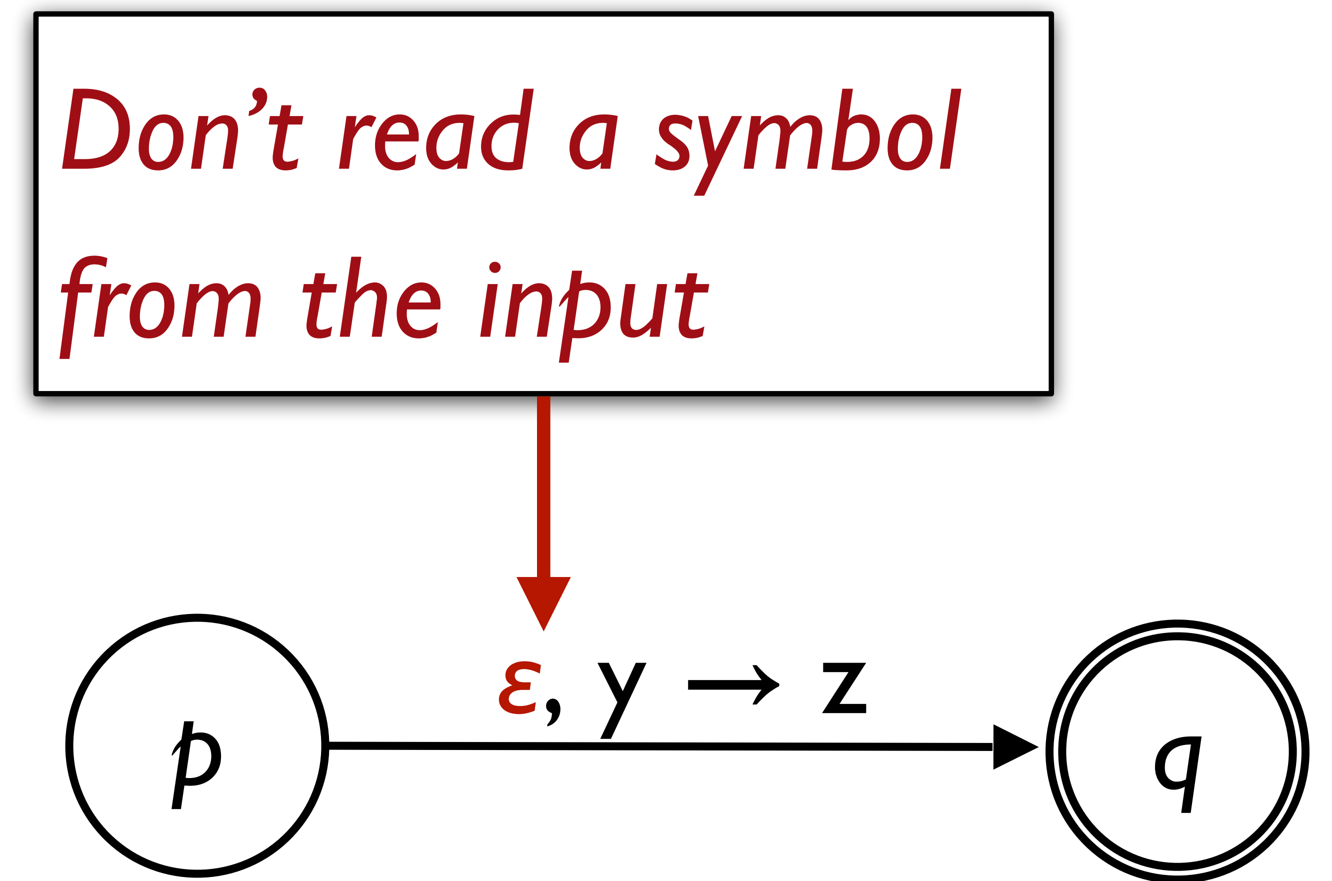
# Notation

At state $p$, if you can

pop the symbol $y$ from the stack,

then you can

enter state $q$ and
push the symbol $z$ onto the stack.



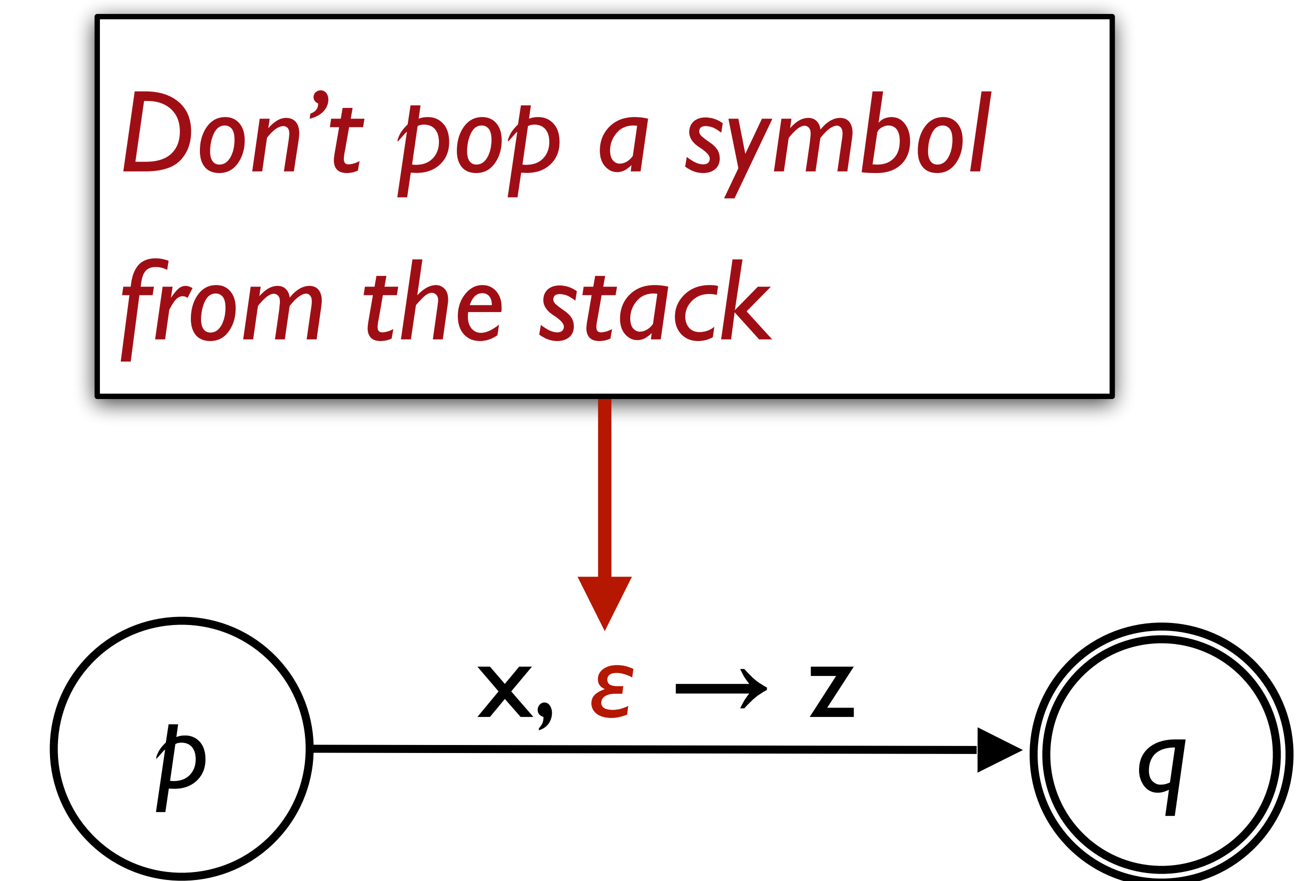
# Notation

At state $p$, if you can

read the symbol $x$ from the input,

then you can

enter state $q$ and
push the symbol $z$ onto the stack.



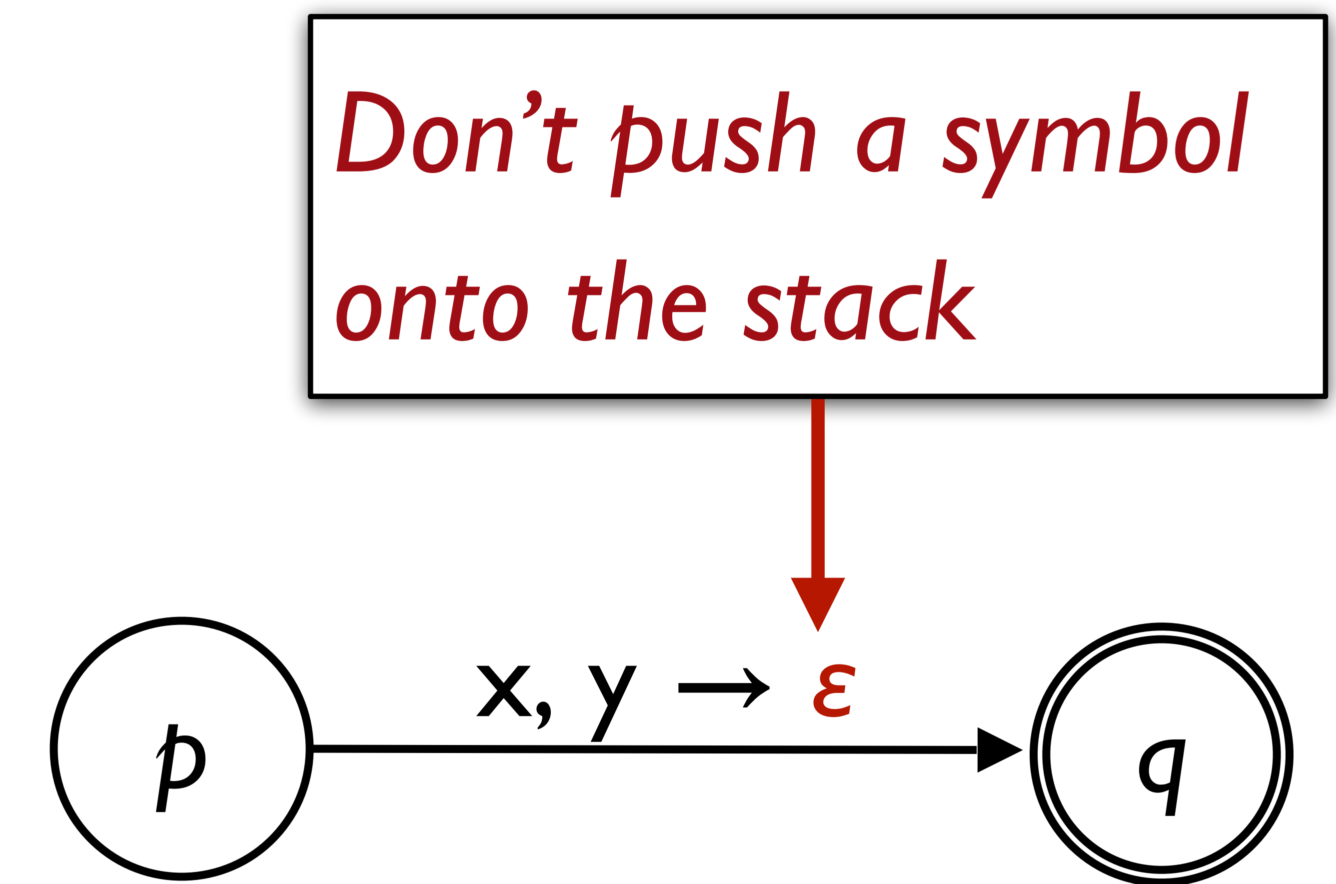
# Notation

At state $p$, if you can

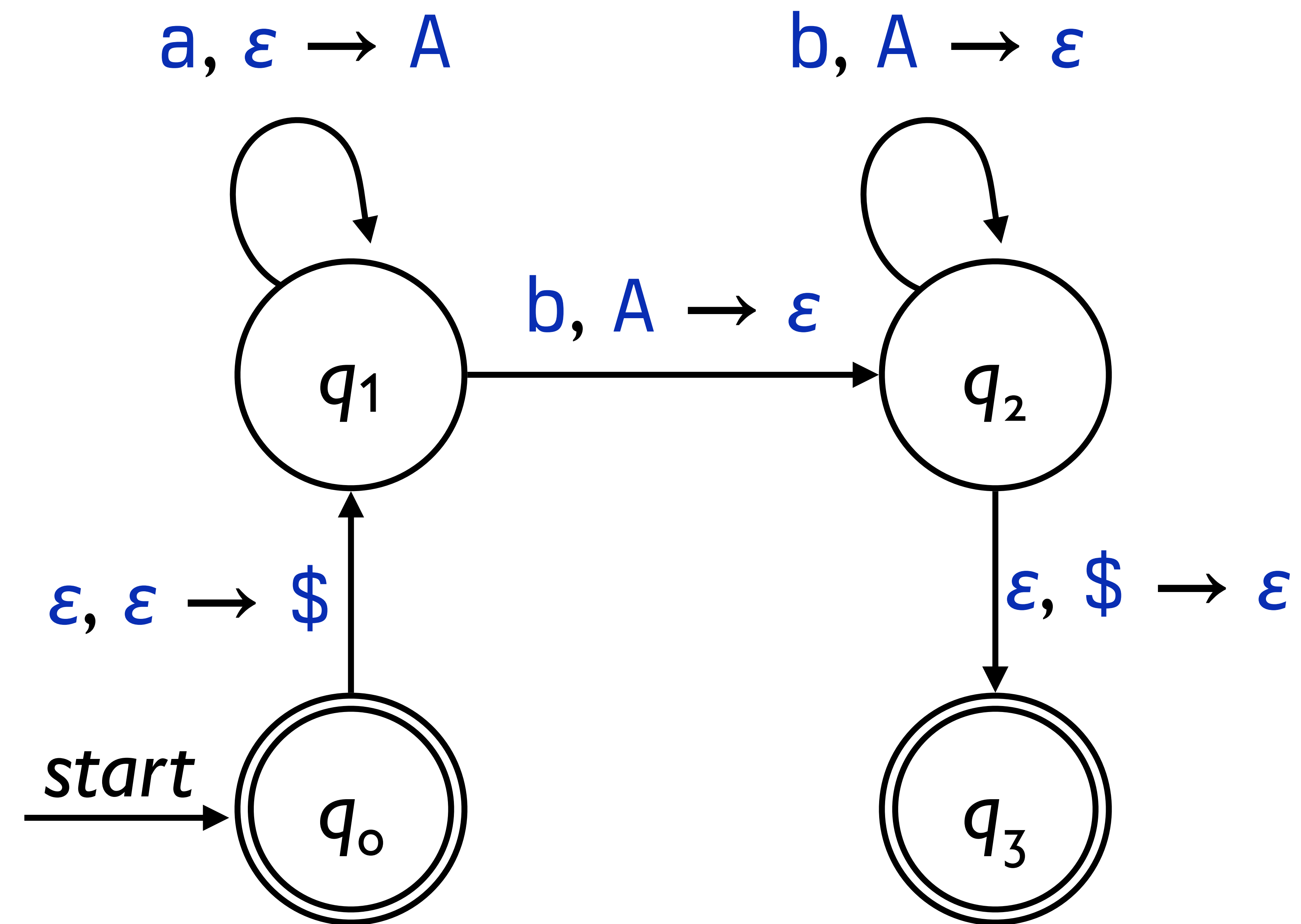
read the symbol $x$ from the input and
pop the symbol $y$ from the stack,

then you can

enter state $q$.



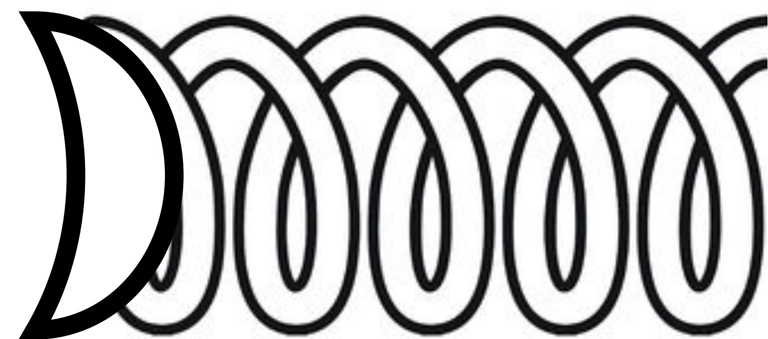
# Example: $\mathbf{a}^n\mathbf{b}^n$



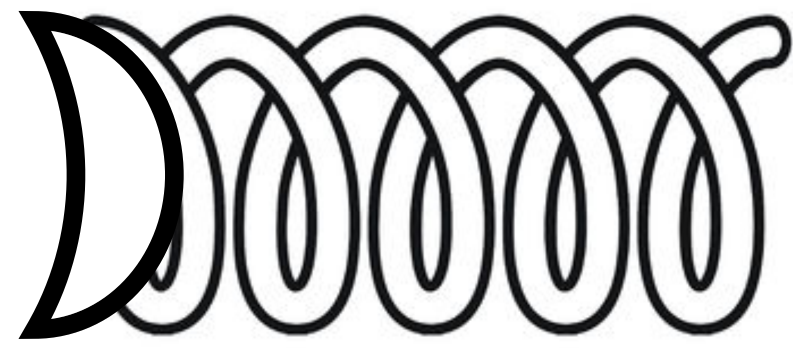

*Input*

a a a b b b

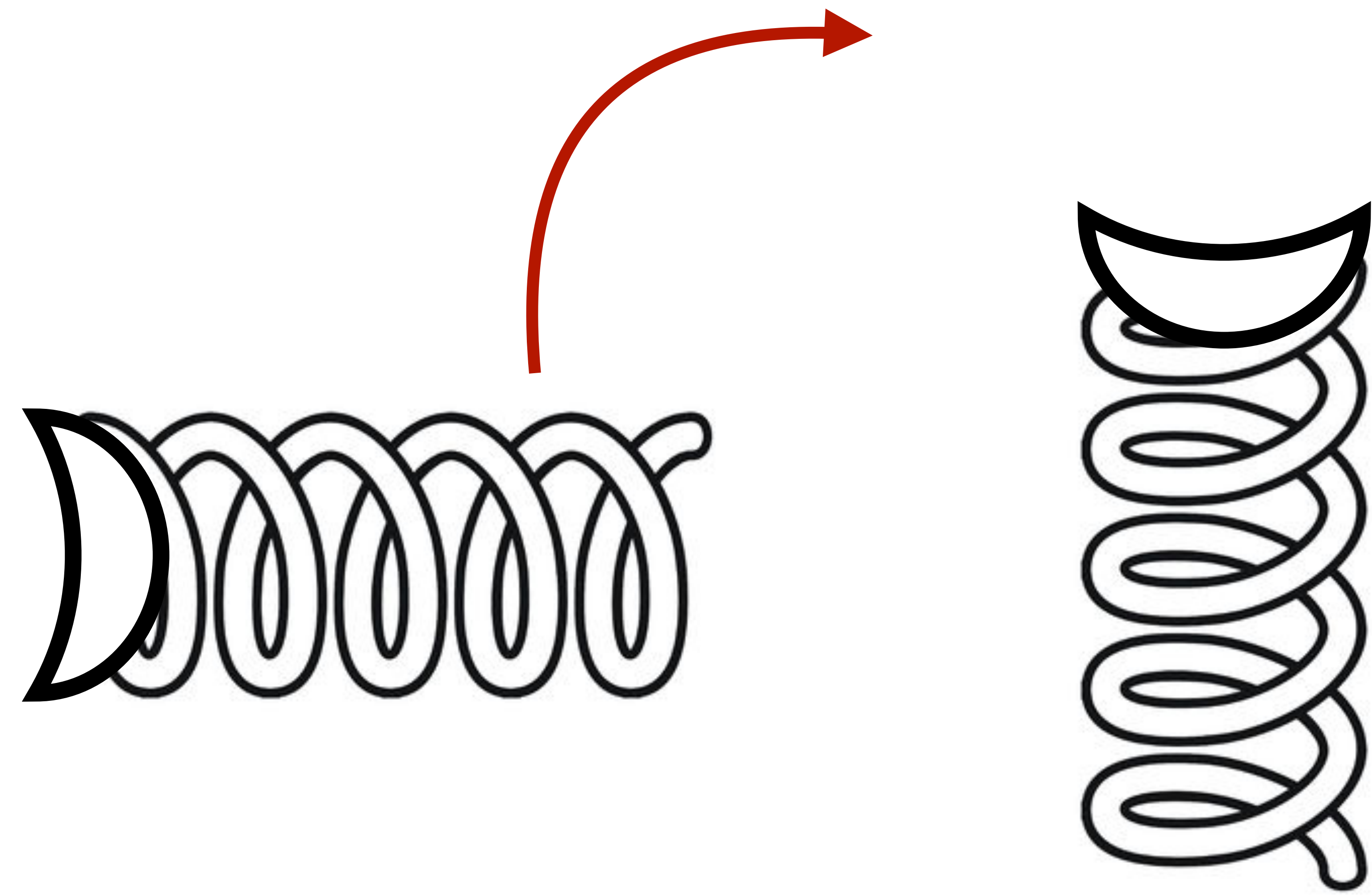
*Stack*



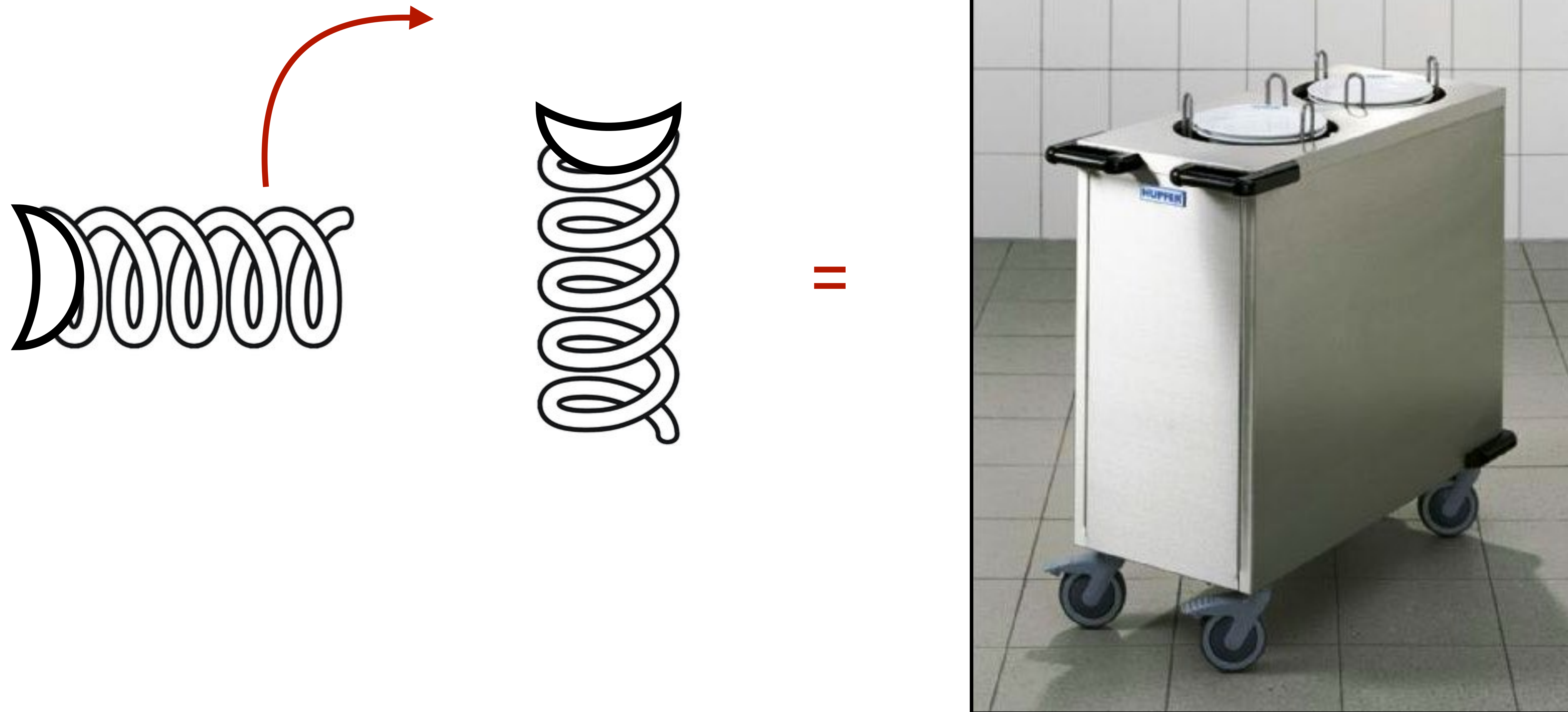
*What's with the weird spring diagram?*



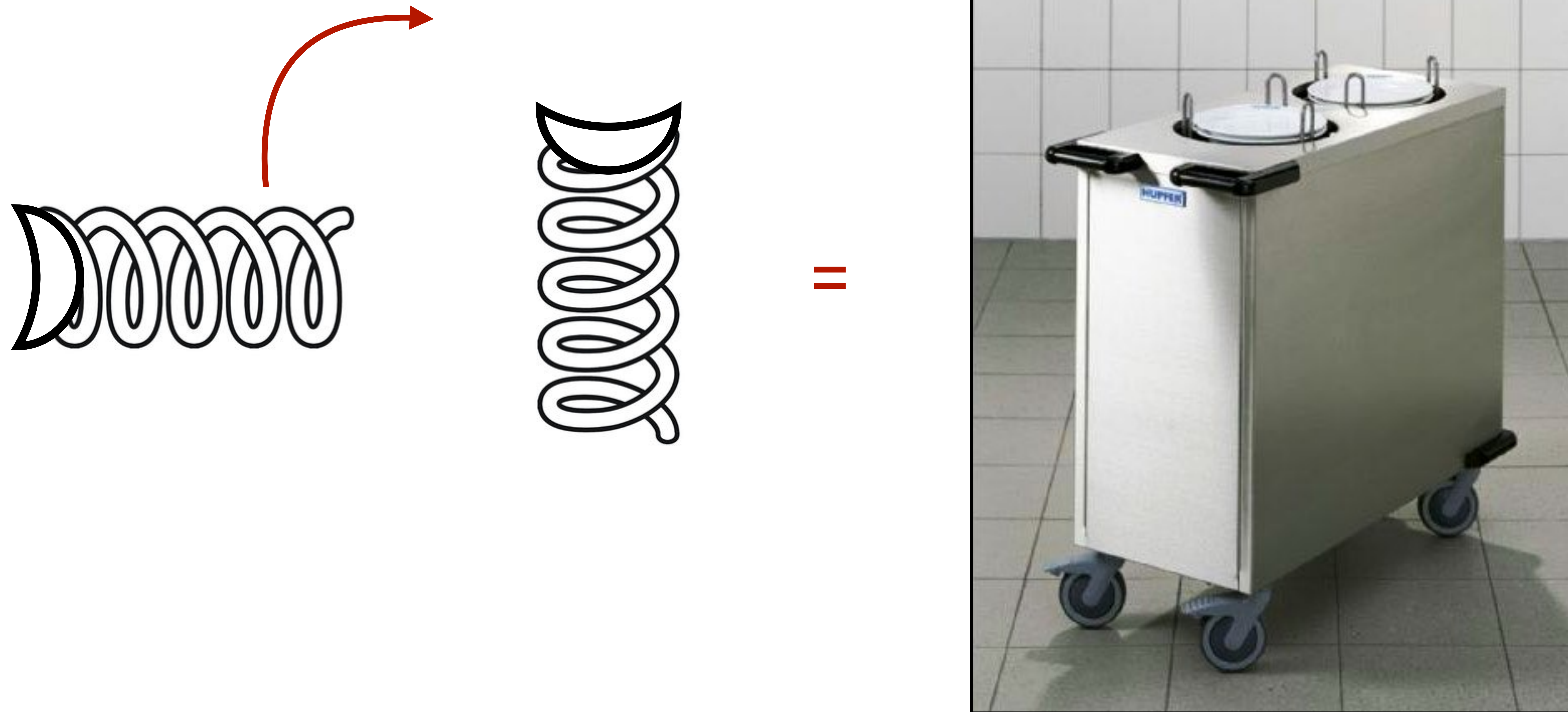
*What's with the weird spring diagram?*



*What's with the weird spring diagram?*

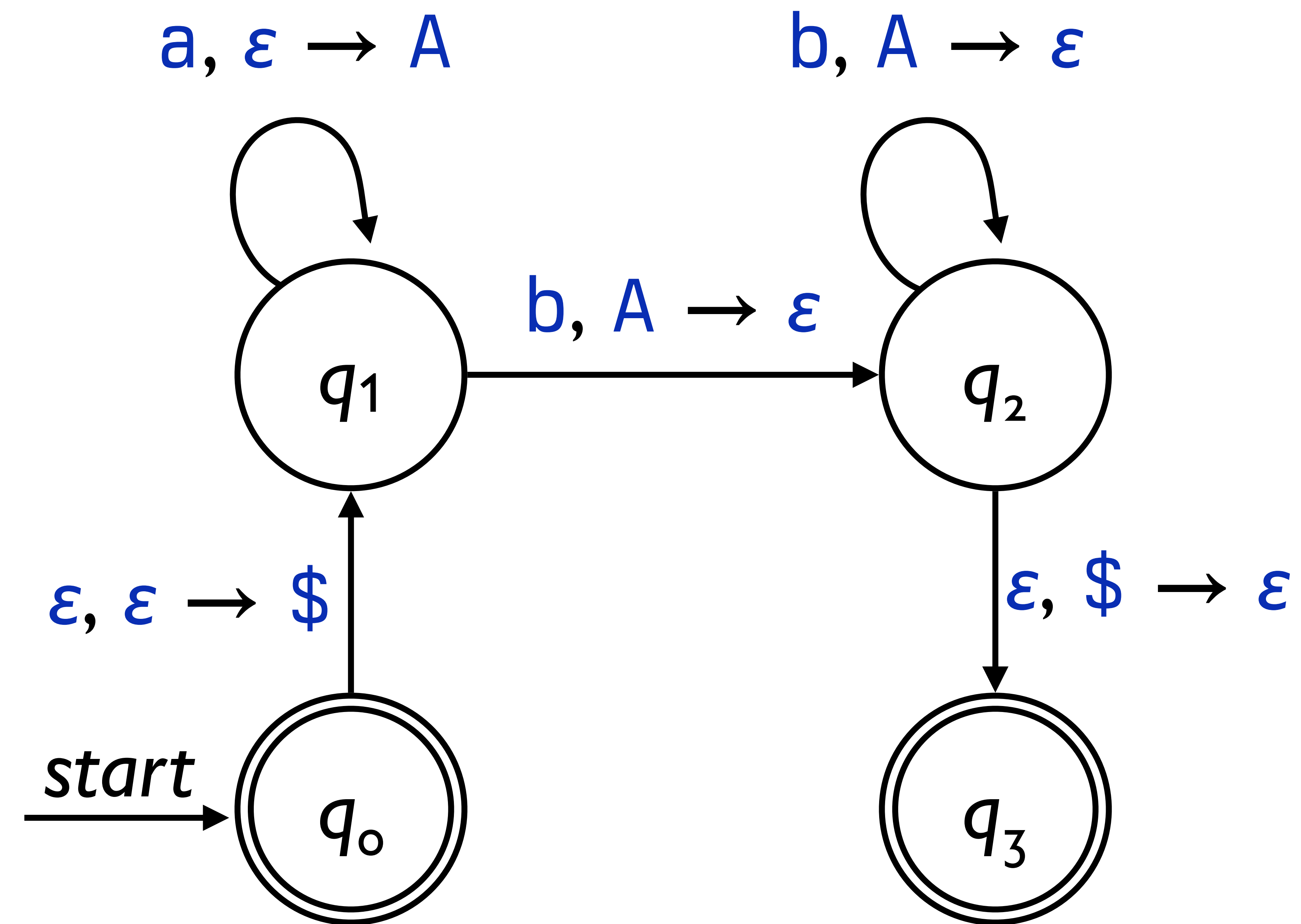


*What's with the weird spring diagram?*




*The spring supporting a stack of plates, shown on its side*

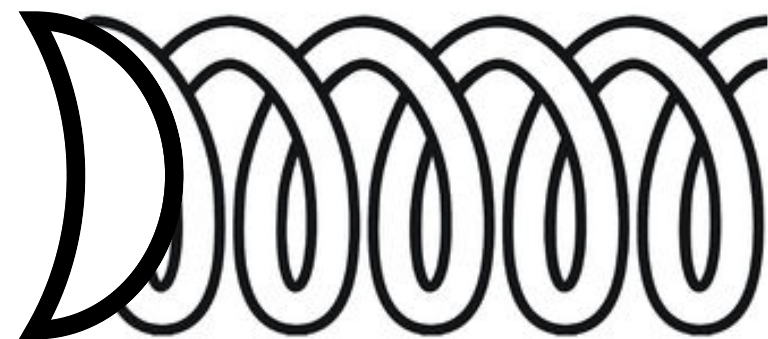
# Example: $\mathbf{a}^n\mathbf{b}^n$



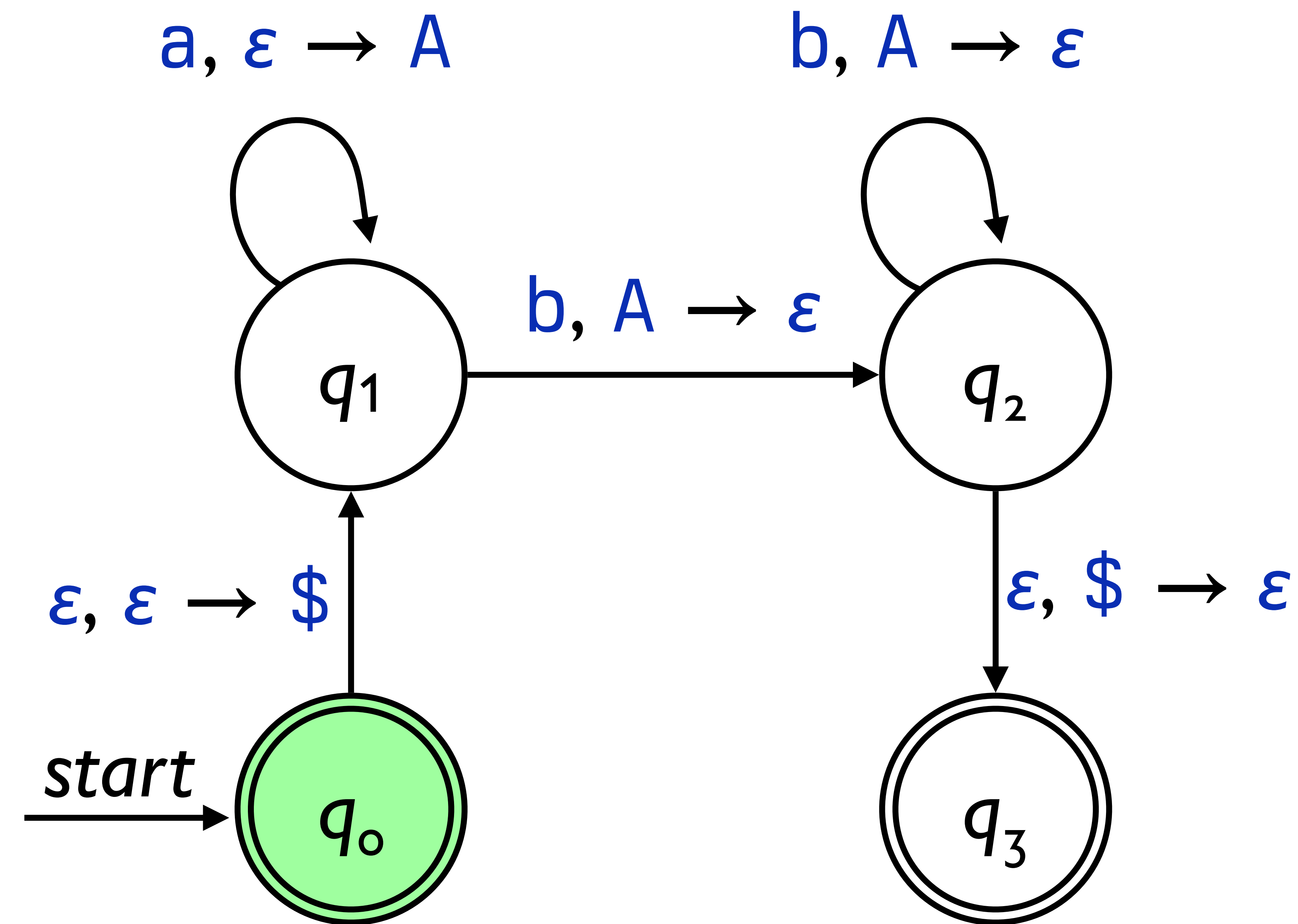

*Input*

a a a b b b

*Stack*

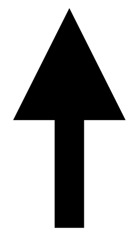


# Example: $\mathbf{a}^n\mathbf{b}^n$

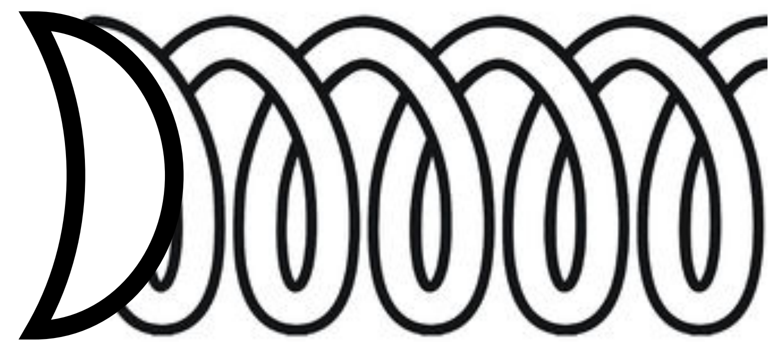



*Input*

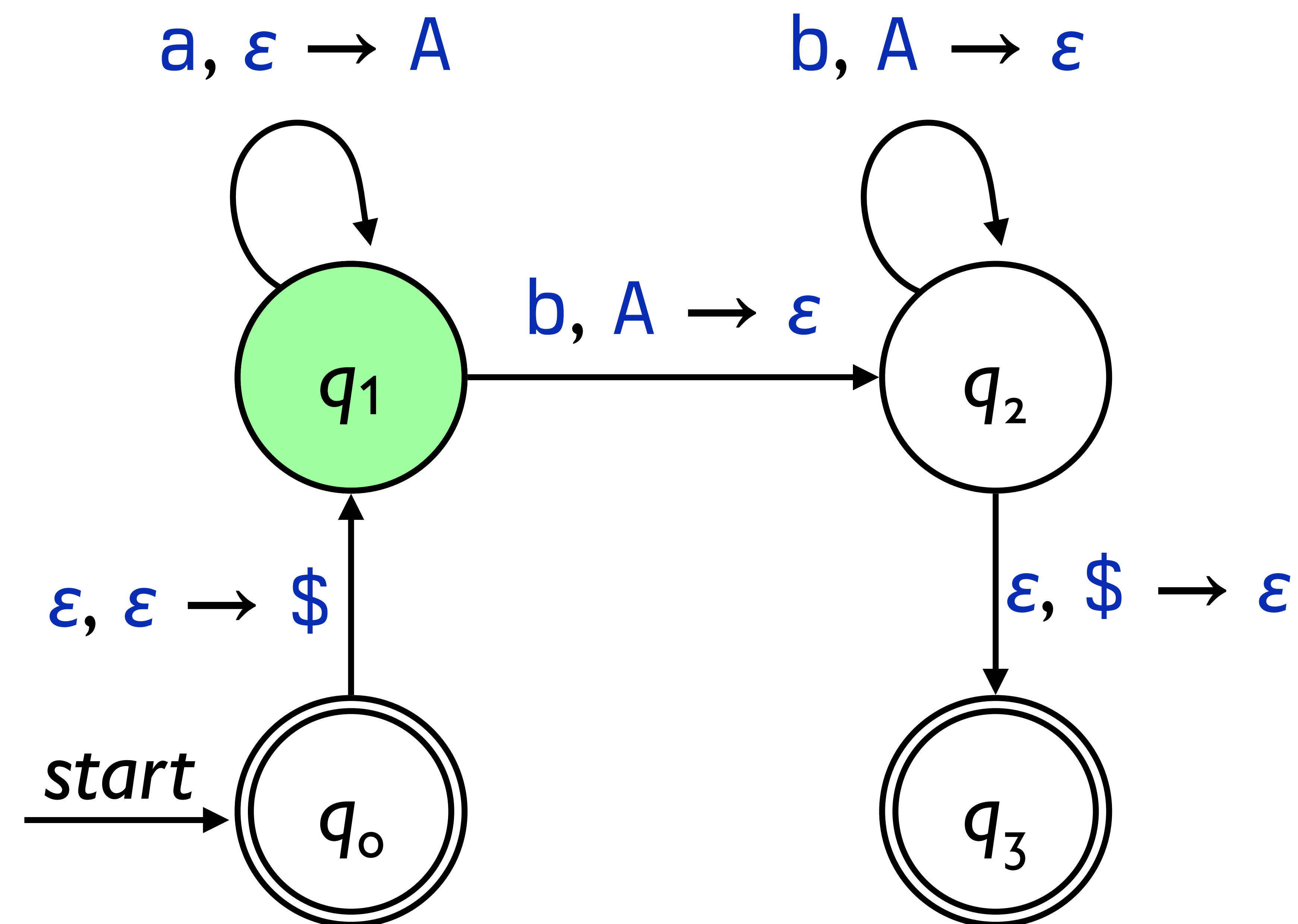
a a a b b b



*Stack*



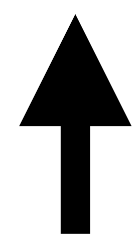
# Example: $a^n b^n$



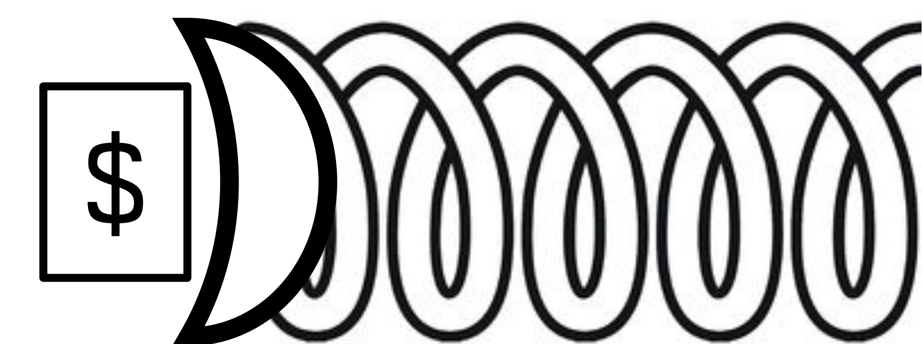

$ is our special "bottom of stack" symbol

*Input*

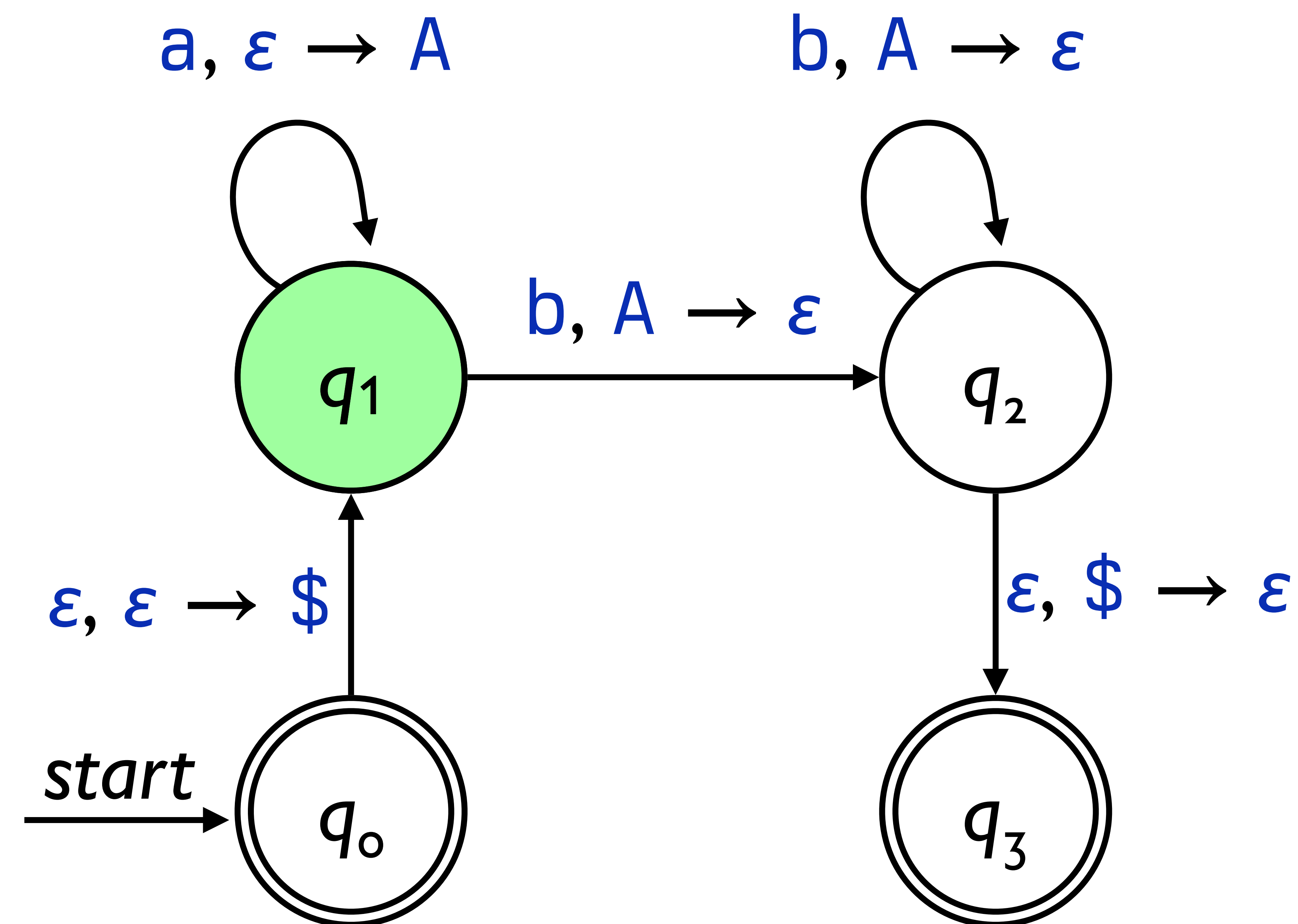
a a a b b b

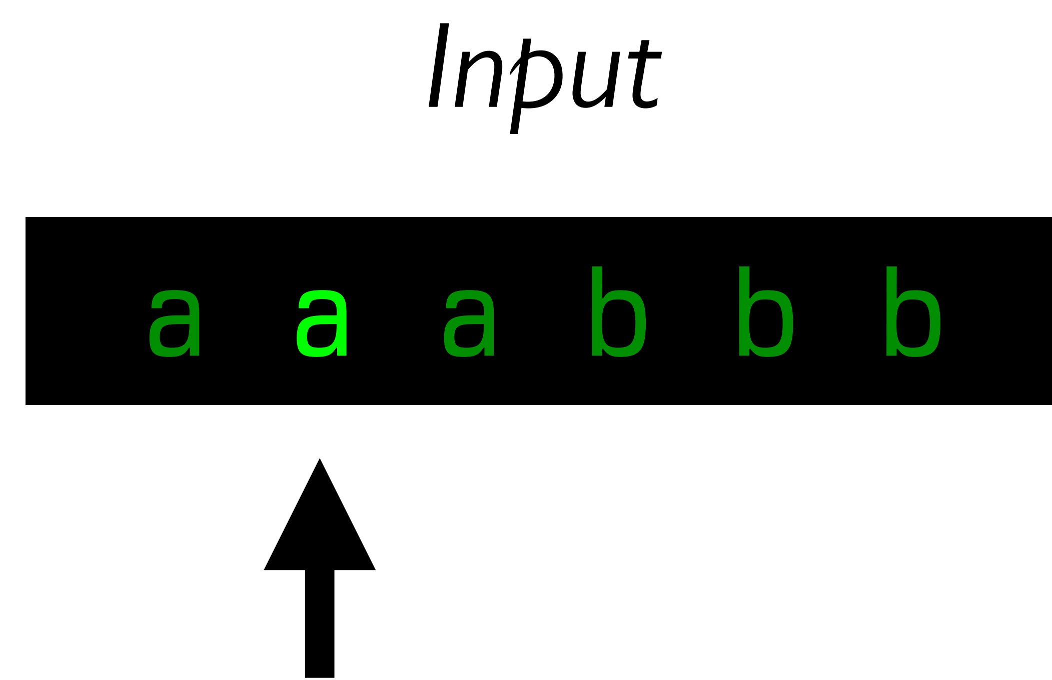


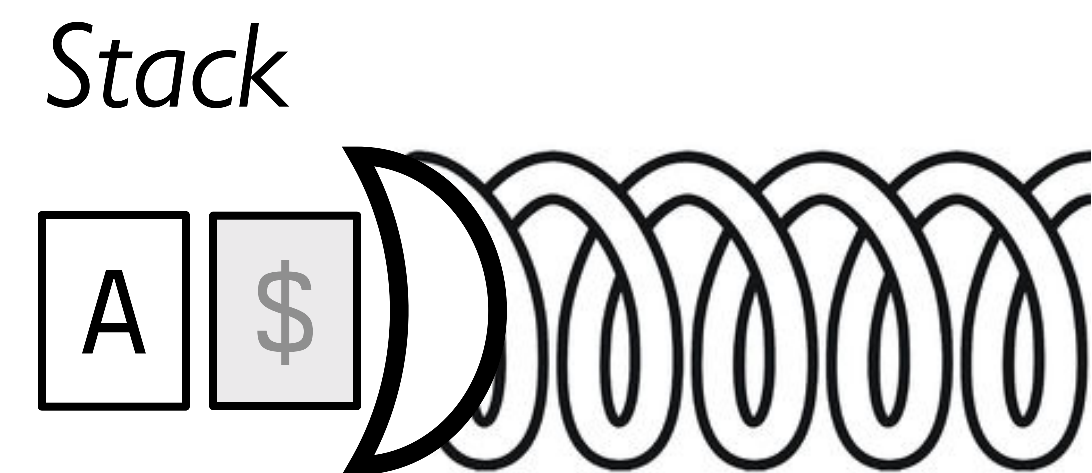
*Stack*



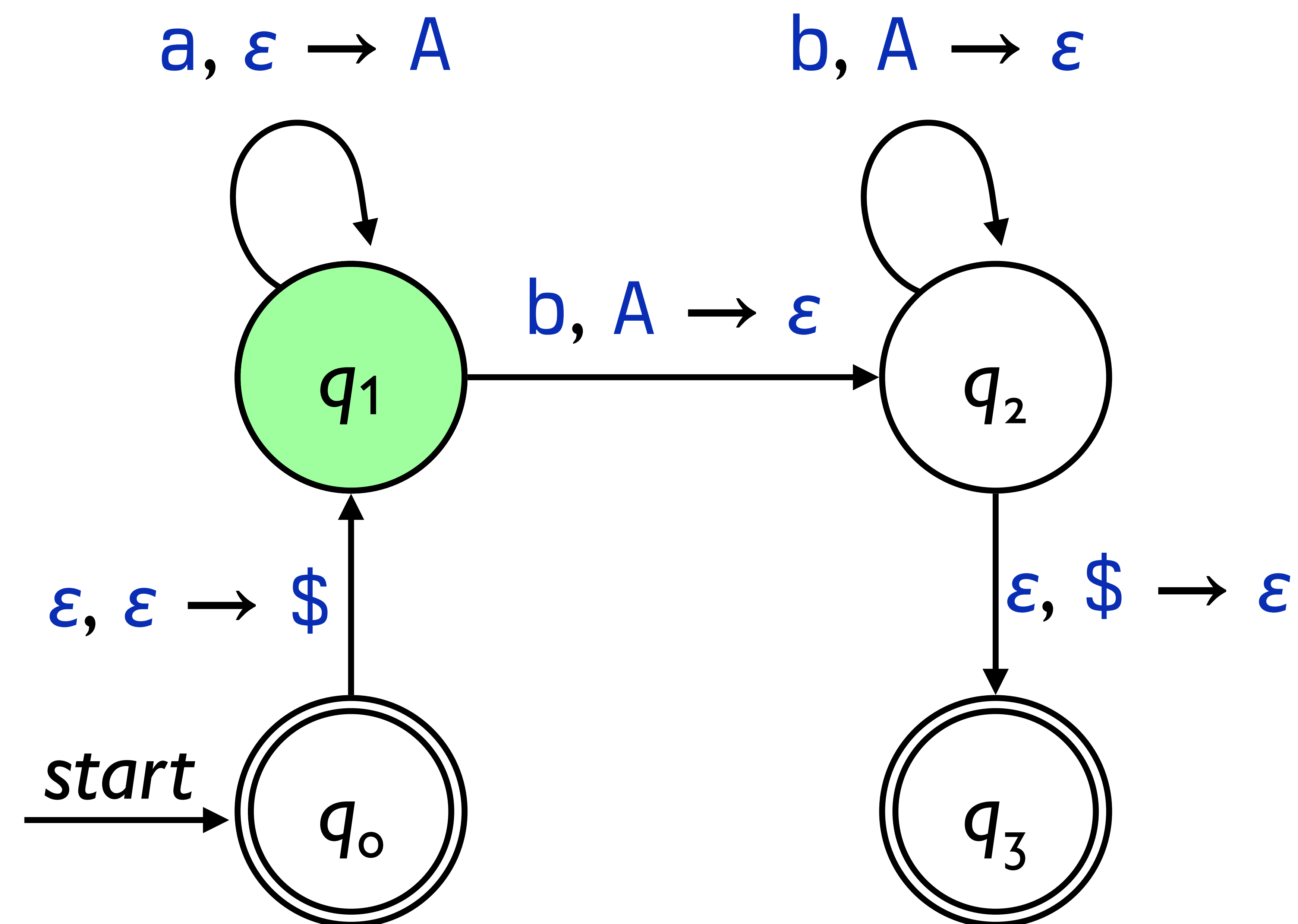
# Example: $\mathbf{a}^n\mathbf{b}^n$



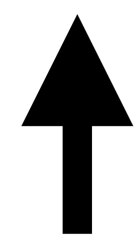



# Example: $\mathbf{a}^n\mathbf{b}^n$

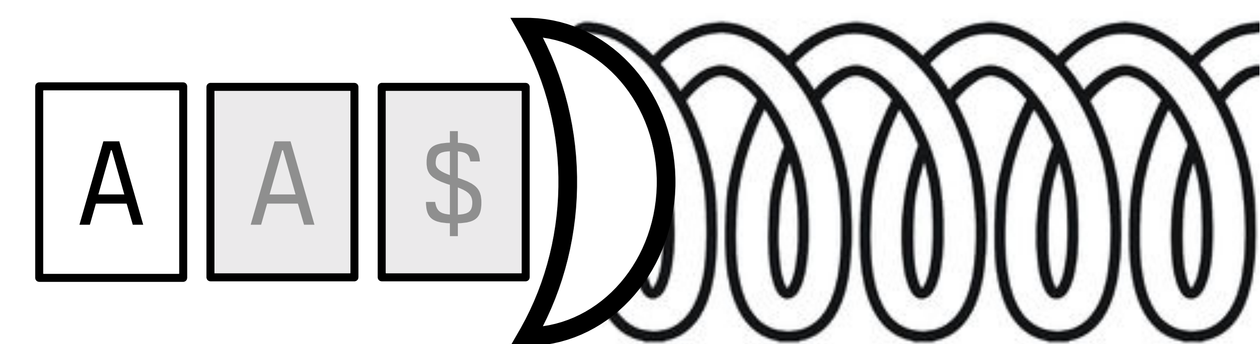



*Input*

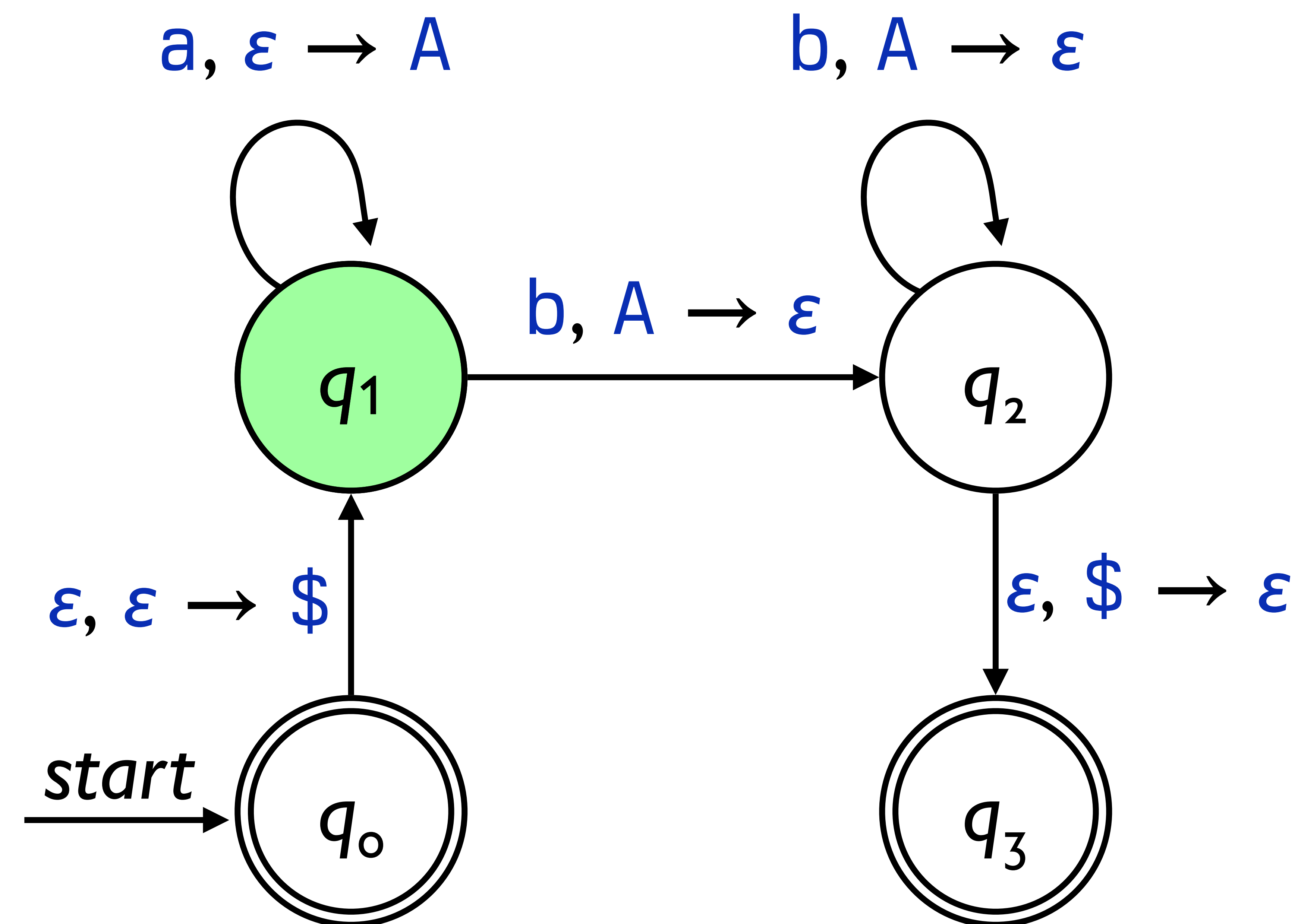
a a a b b b



*Stack*

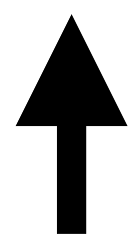


# Example: $\mathbf{a}^n\mathbf{b}^n$

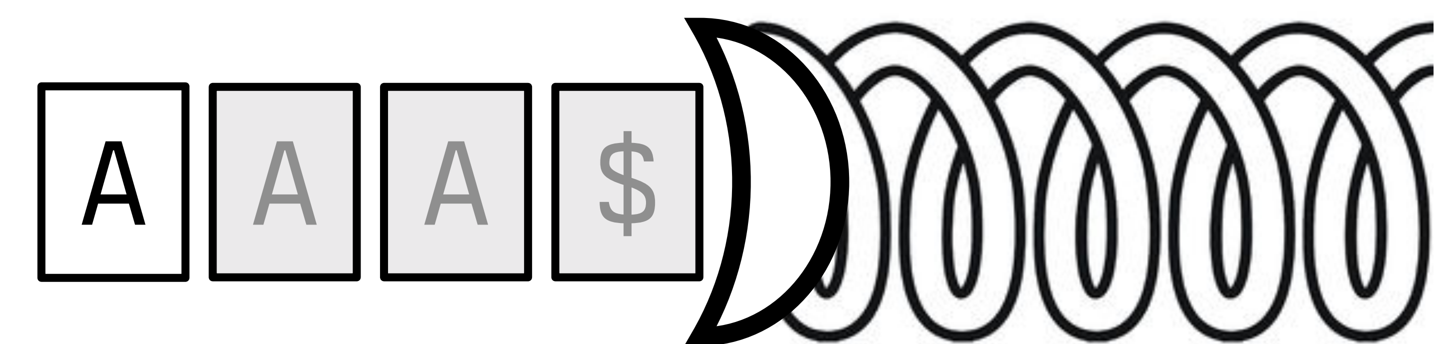



*Input*

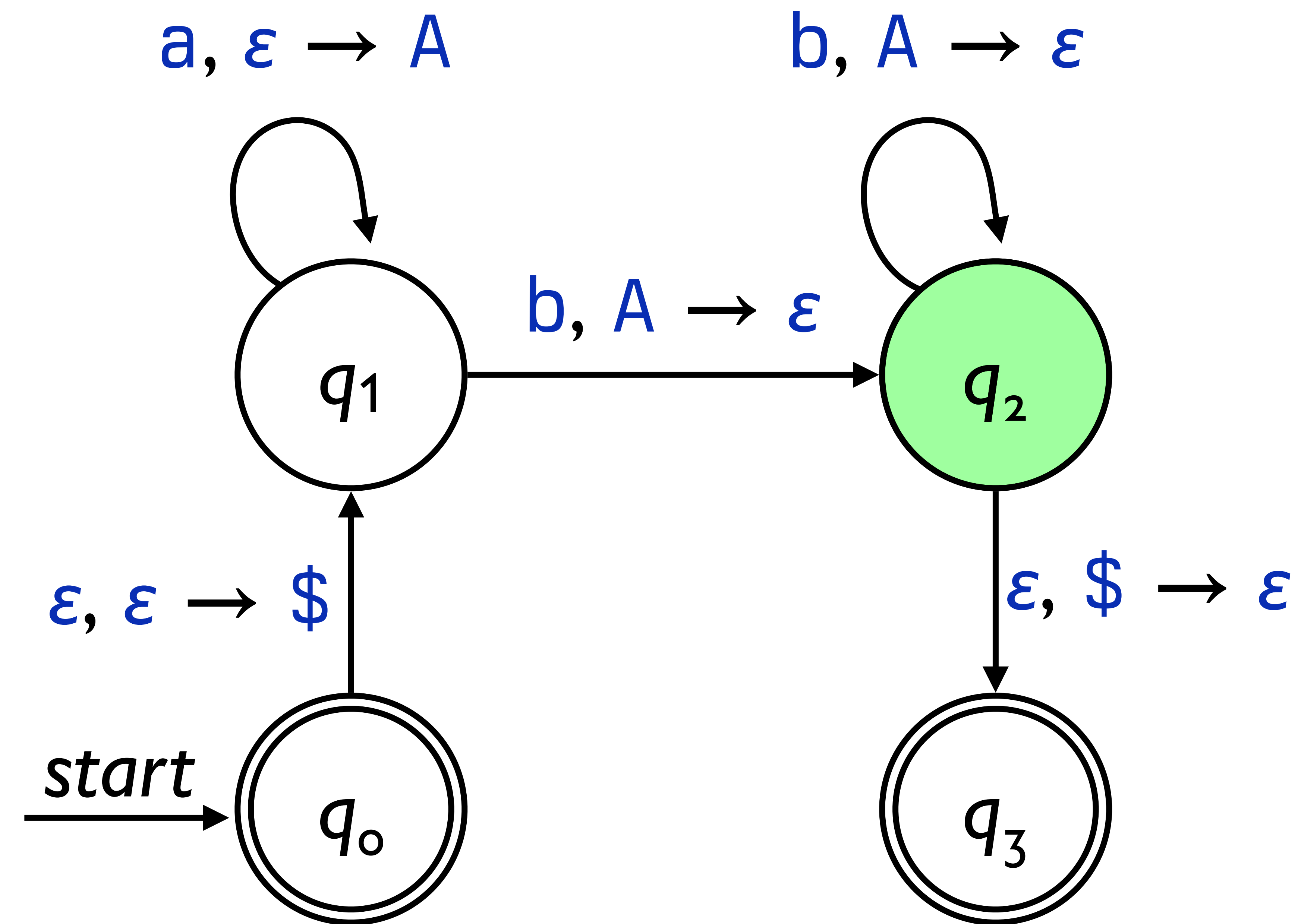
a a a b b b



*Stack*

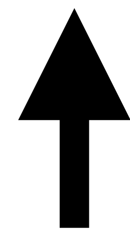


# Example: $\mathbf{a}^n\mathbf{b}^n$

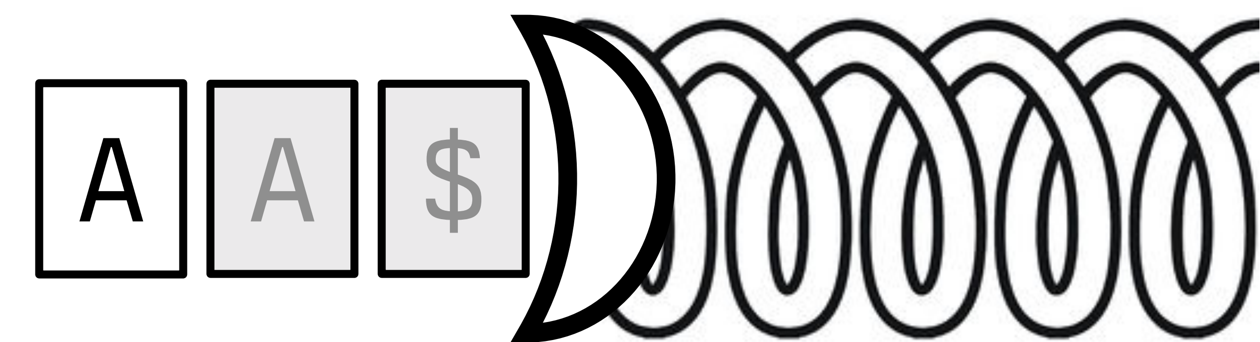



*Input*

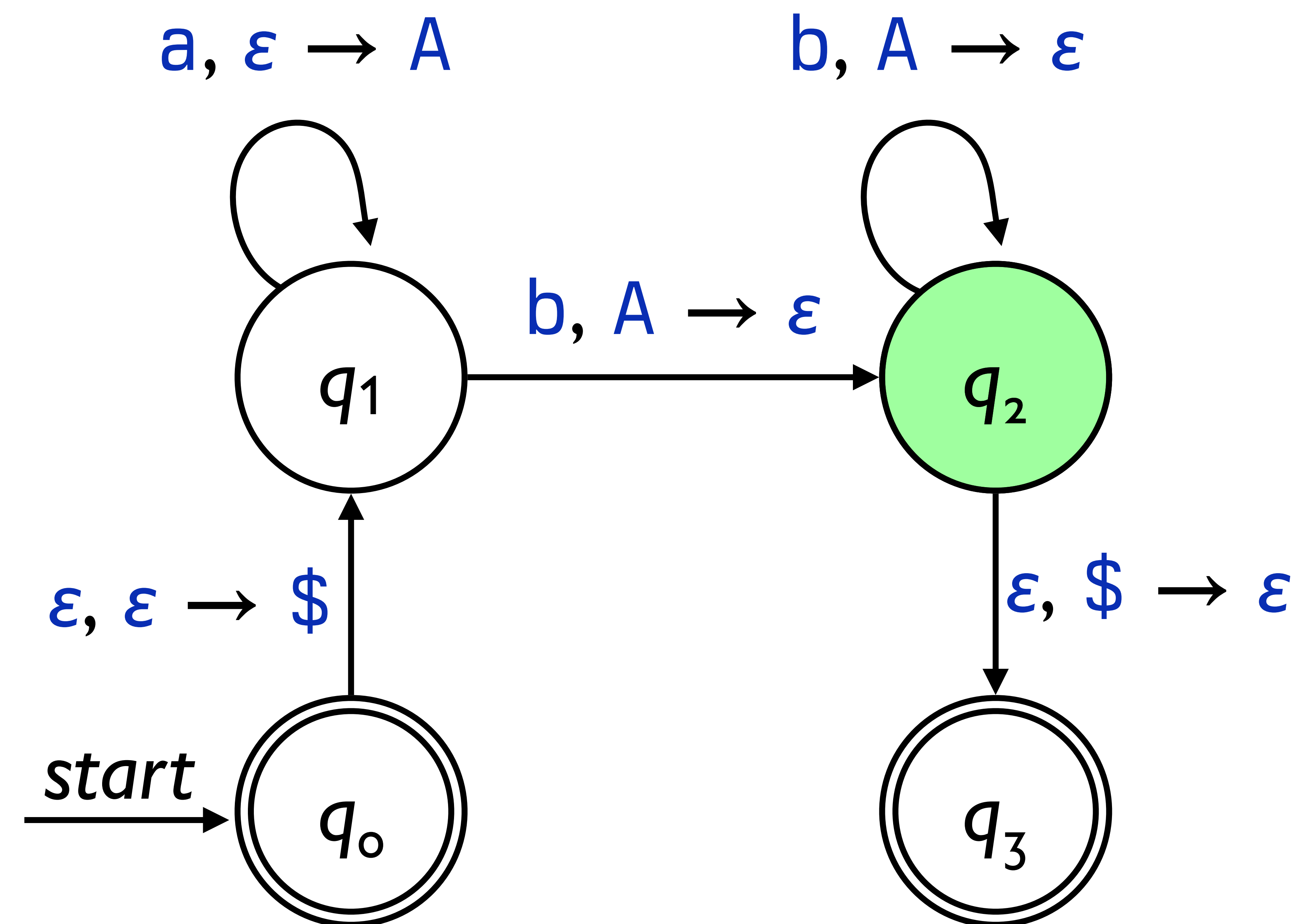
a a a b b b



*Stack*

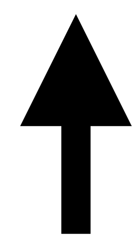


# Example: $\mathbf{a}^n\mathbf{b}^n$

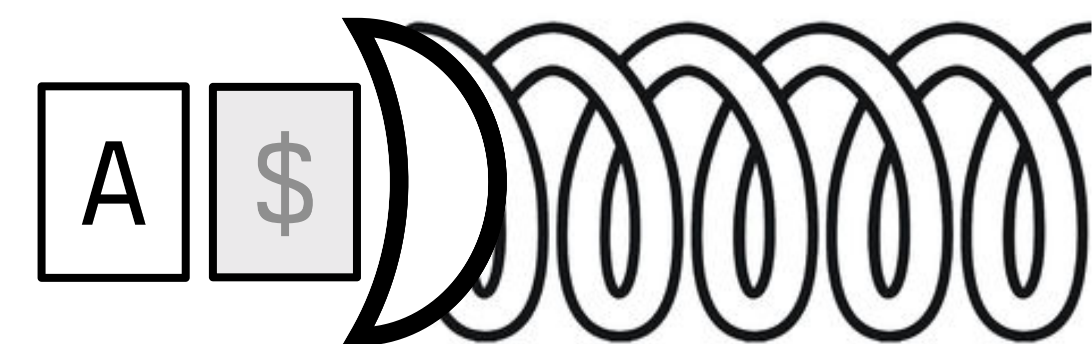



*Input*

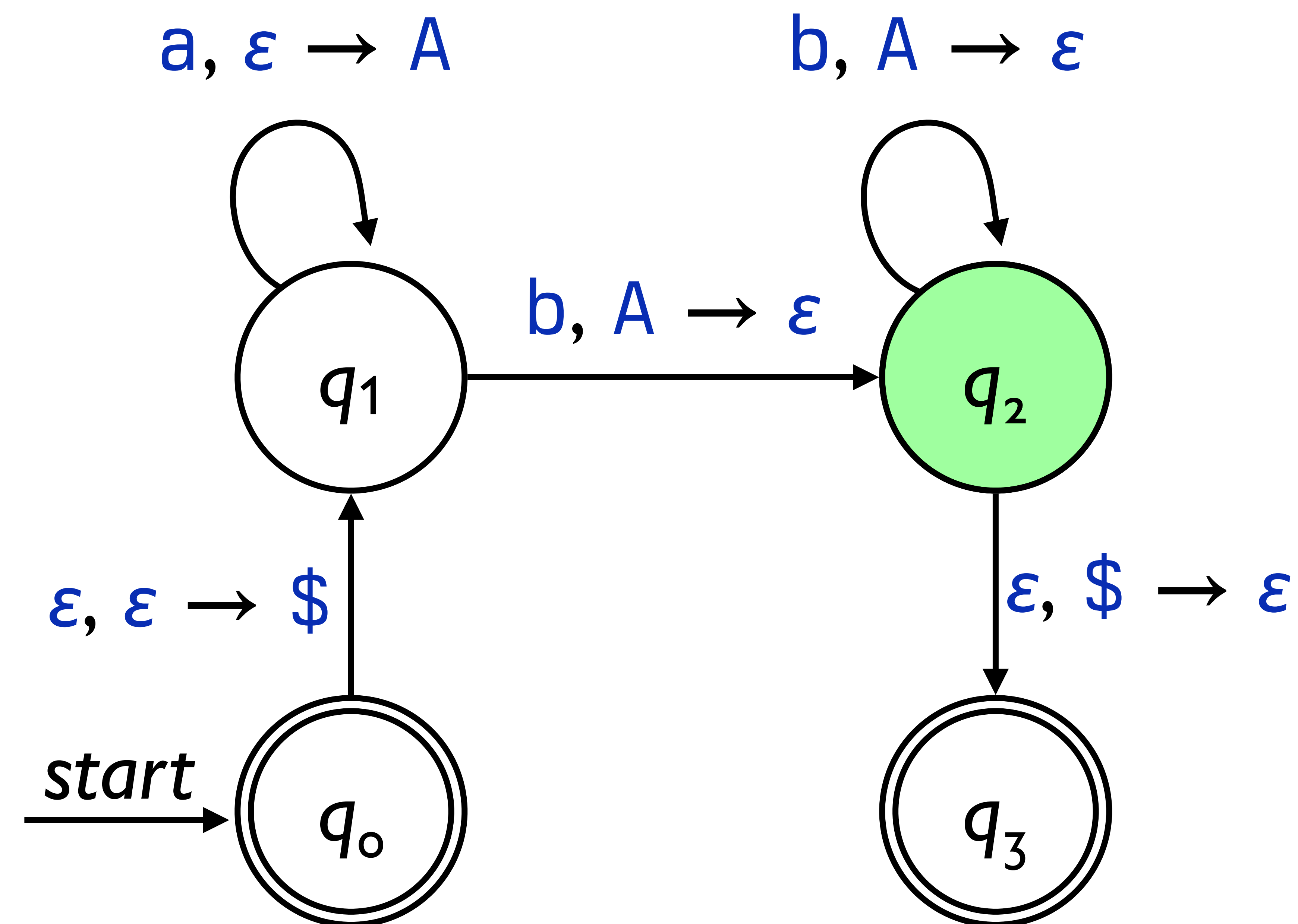
a a a b b b



*Stack*

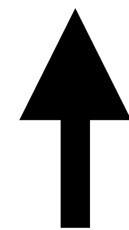


# Example: $\mathbf{a}^n\mathbf{b}^n$

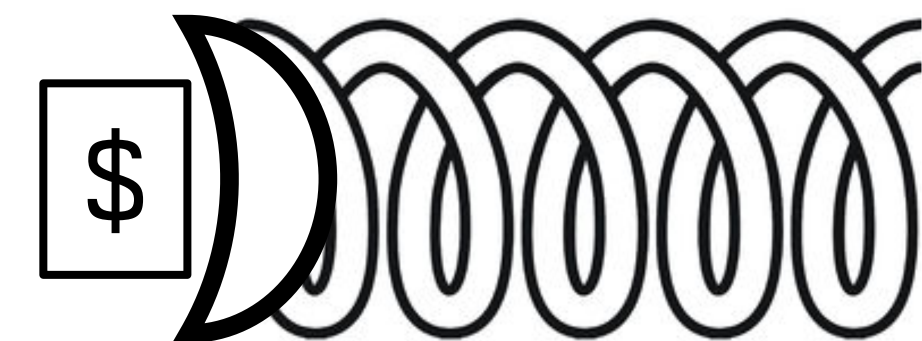



*Input*

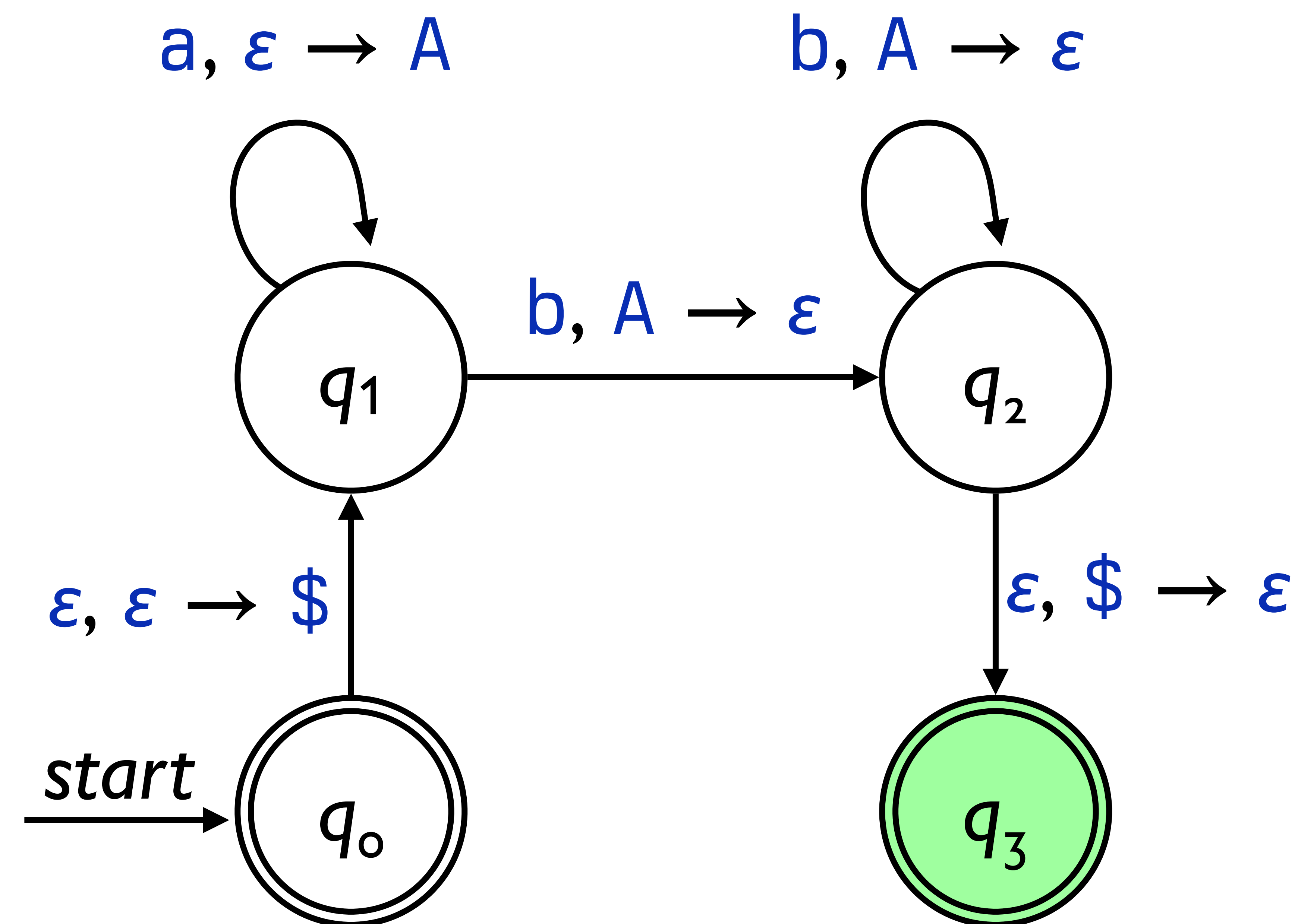
a a a b b b



*Stack*

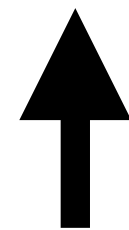


# Example: $\mathbf{a}^n\mathbf{b}^n$

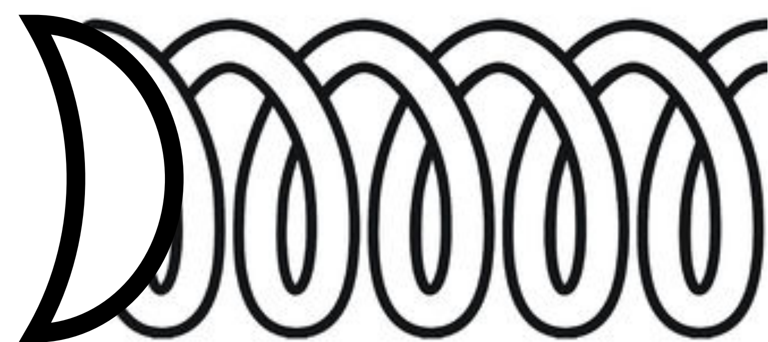



*Input*

a a a b b b



*Stack*



Formally, a *pushdown automaton* is a nondeterministic machine defined by the six-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$.

Formally, a *pushdown automaton* is a nondeterministic machine defined by the six-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$.

$Q$ is a finite set of states,

Formally, a *pushdown automaton* is a nondeterministic machine defined by the six-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$.

$Q$ is a finite set of states,

$\Sigma$ is the alphabet for input strings,

Formally, a *pushdown automaton* is a nondeterministic machine defined by the six-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$.

$Q$ is a finite set of states,

$\Sigma$ is the alphabet for input strings,

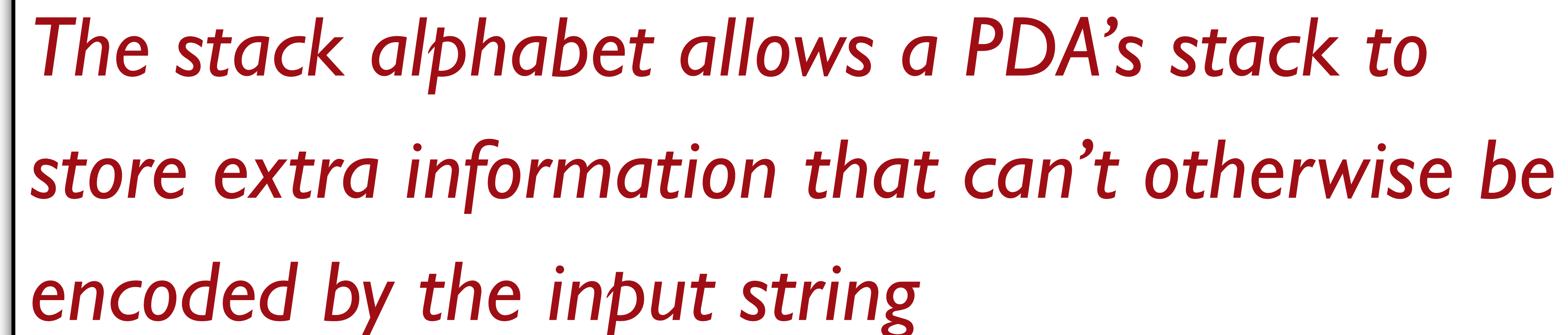
$\Gamma$ is the *stack alphabet* of symbols that can be pushed on the stack

Formally, a *pushdown automaton* is a nondeterministic machine defined by the six-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$.

$Q$ is a finite set of states,

$\Sigma$ is the alphabet for input strings,

$\Gamma$ is the *stack alphabet* of symbols that can be pushed on the stack



Formally, a *pushdown automaton* is a nondeterministic machine defined by the six-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$.

$Q$ is a finite set of states,

$\Sigma$ is the alphabet for input strings,

$\Gamma$ is the *stack alphabet* of symbols that can be pushed on the stack,

$\delta: Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \wp(Q \times \Gamma_\varepsilon)$ is the *transition function*

Formally, a *pushdown automaton* is a nondeterministic machine defined by the six-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$.

$Q$ is a finite set of states,

$\Sigma$ is the alphabet for input strings,

$\Gamma$ is the *stack alphabet* of symbols that can be pushed on the stack,

$\delta: Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \rightarrow \wp(Q \times \Gamma_\varepsilon)$ is the *transition function*

*Each transition is based on a combination of the current state, the current input symbol, and the current stack symbol.*

Formally, a *pushdown automaton* is a nondeterministic machine defined by the six-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$.

$Q$ is a finite set of states,

$\Sigma$ is the alphabet for input strings,

$\Gamma$ is the *stack alphabet* of symbols that can be pushed on the stack,

$\delta$: $Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \wp(Q \times \Gamma_\varepsilon)$ is the *transition function*

*Each transition is based on a combination of the current state, the current input symbol, and the current stack symbol.*

*The function maps to a set of state–string pairs, and the string is pushed atop the stack.*

Formally, a *pushdown automaton* is a nondeterministic machine defined by the six-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$.

$Q$ is a finite set of states,

$\Sigma$ is the alphabet for input strings,

$\Gamma$ is the *stack alphabet* of symbols that can be pushed on the stack,

$\delta: Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \wp(Q \times \Gamma_\varepsilon)$ is the *transition function*,

$q_0 \in Q$ is the *start state*, and

Formally, a *pushdown automaton* is a nondeterministic machine defined by the six-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$.

$Q$ is a finite set of states,

$\Sigma$ is the alphabet for input strings,

$\Gamma$ is the *stack alphabet* of symbols that can be pushed on the stack,

$\delta: Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \rightarrow \wp(Q \times \Gamma_\varepsilon)$ is the *transition function*,

$q_0 \in Q$ is the *start state*, and

$F \subseteq Q$ is the set of *accept states*.

Formally, a *pushdown automaton* is a nondeterministic machine defined by the six-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$.

$Q$ is a finite set of states,

$\Sigma$ is the alphabet for input strings,

$\Gamma$ is the *stack alphabet* of symbols that can be pushed on the stack,

$\delta: Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \wp(Q \times \Gamma_\varepsilon)$ is the *transition function*,

$q_0 \in Q$ is the *start state*, and

$F \subseteq Q$ is the set of *accept states*.

The automaton accepts if it ends in an accept state with no input remaining.

# Beware!

While finite automata are highly standardized, there are many different – but equivalent – definitions of PDAs.

The one we'll use matches the textbook. Avoid other materials – videos, textbooks, etc. – or you may get confused.

# Example: Palindromes

Recall: A *palindrome* is a string that is the same forwards and backwards.

Let $\Sigma = \{a, b, \#\}$ and consider the language

$$L = \{w\#w^R \mid w \in \{a, b\}^*\}$$

E.g.,

#

a#a

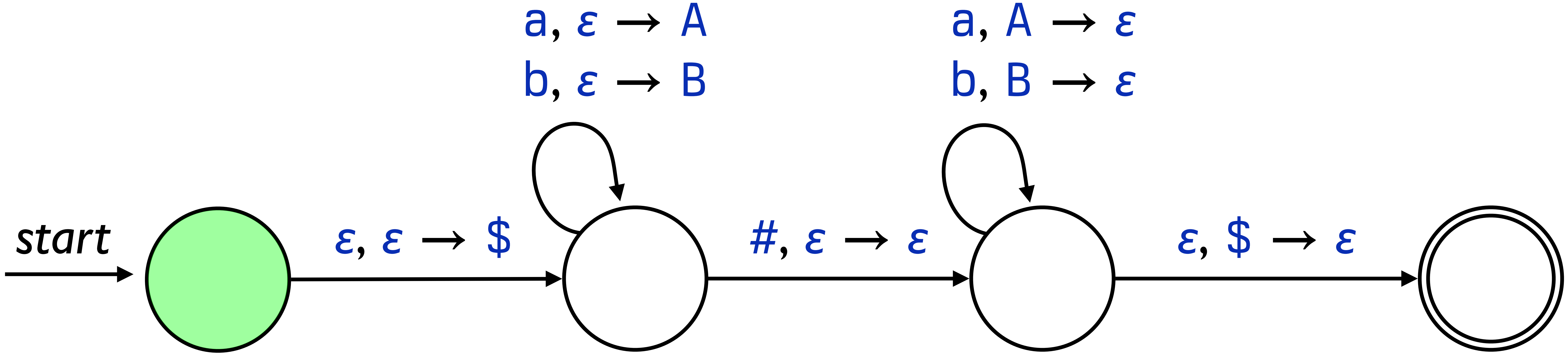
b#b

ab#ba

bb#bb

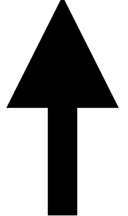
…

# Example: Palindromes

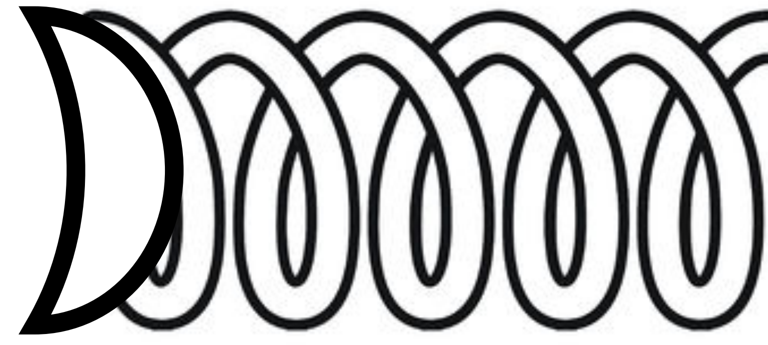



*Input*

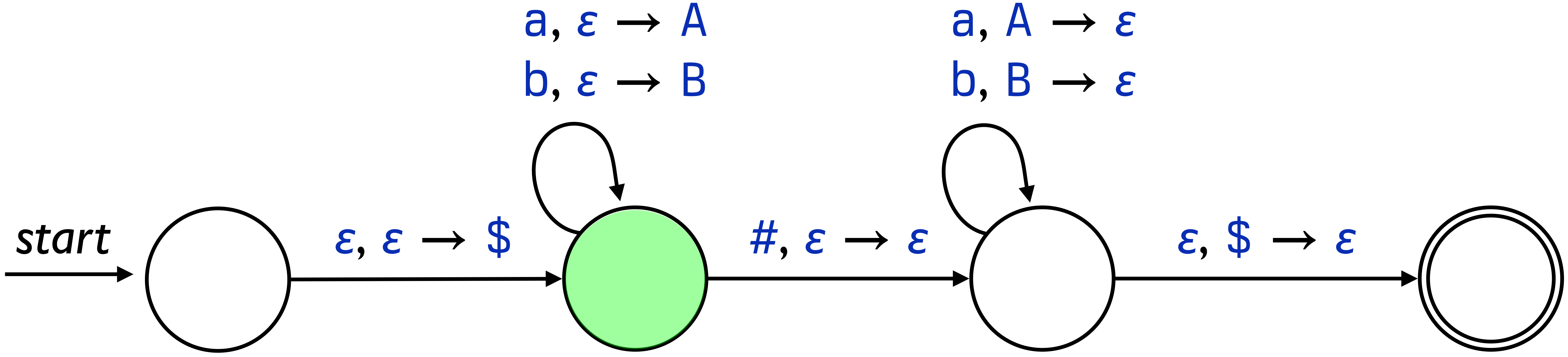
a a a b # b a a a



*Stack*

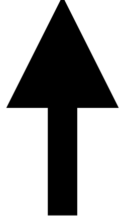


# Example: Palindromes

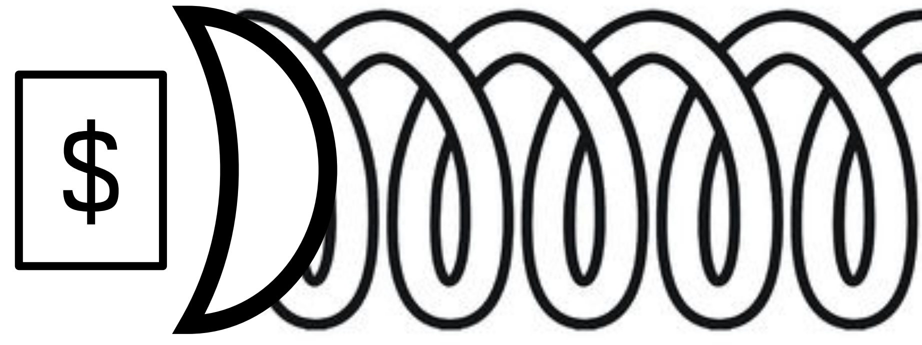



*Input*

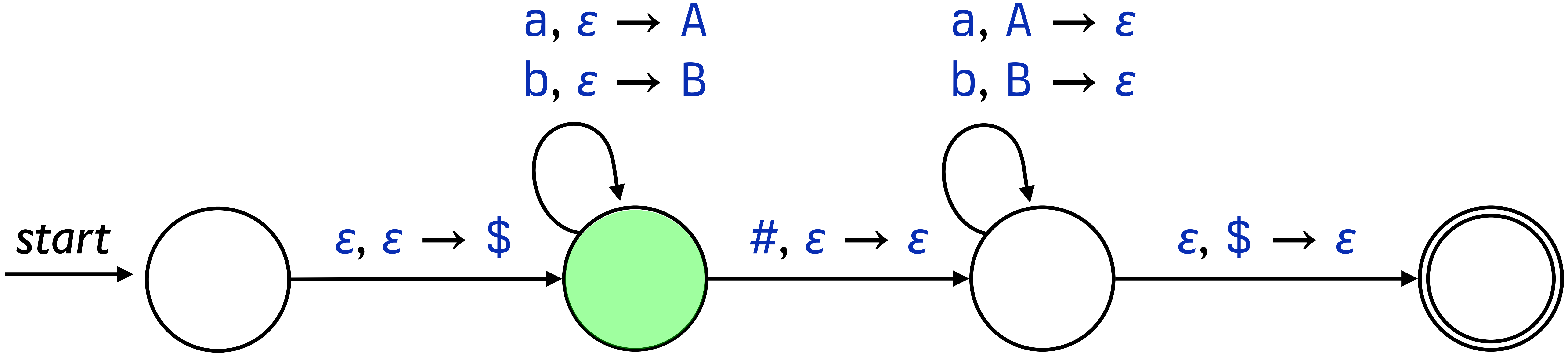
a a a b # b a a a



*Stack*

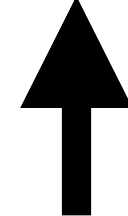


# Example: Palindromes

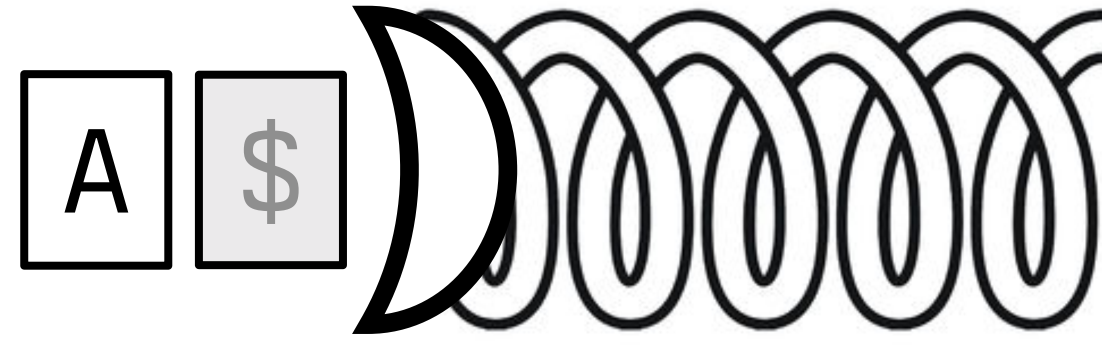



*Input*

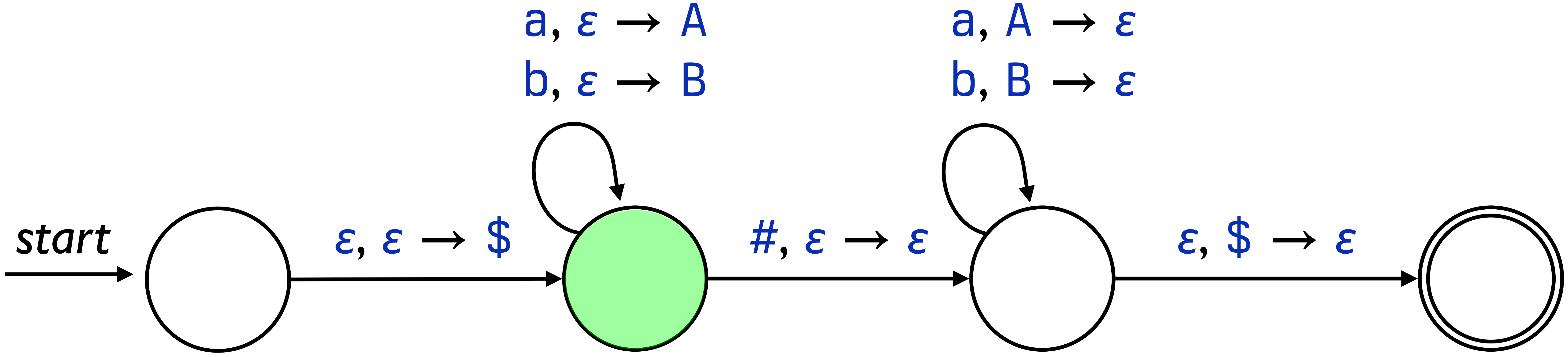
a a a b # b a a a



*Stack*

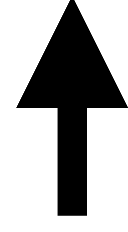


# Example: Palindromes

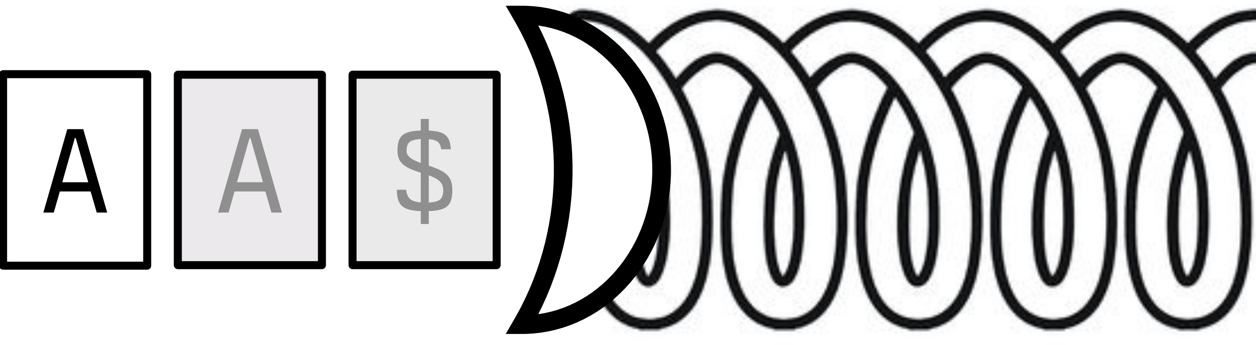



*Input*

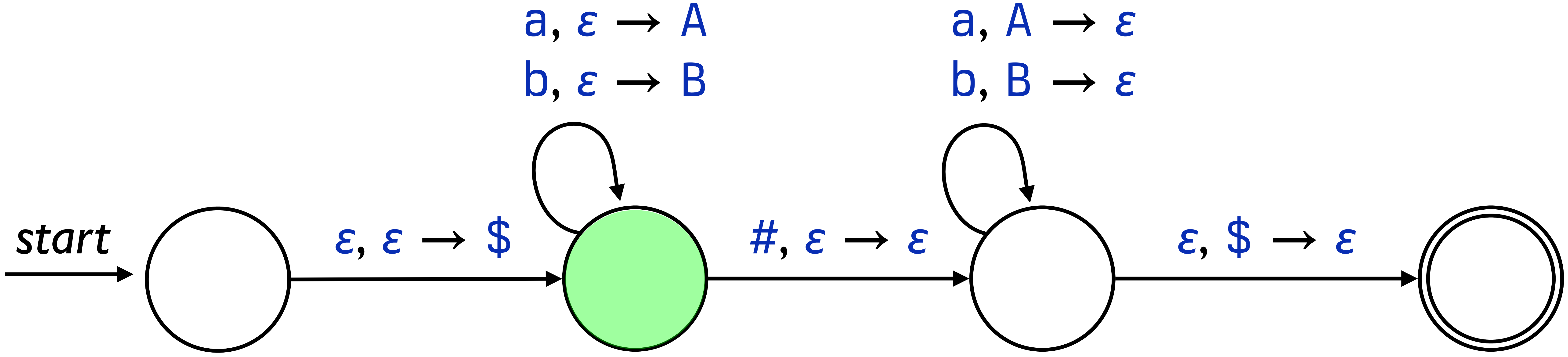
a a a b # b a a a



*Stack*

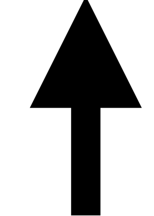


# Example: Palindromes

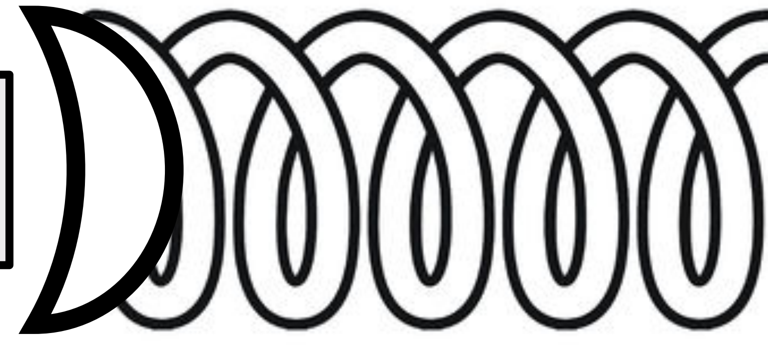



*Input*

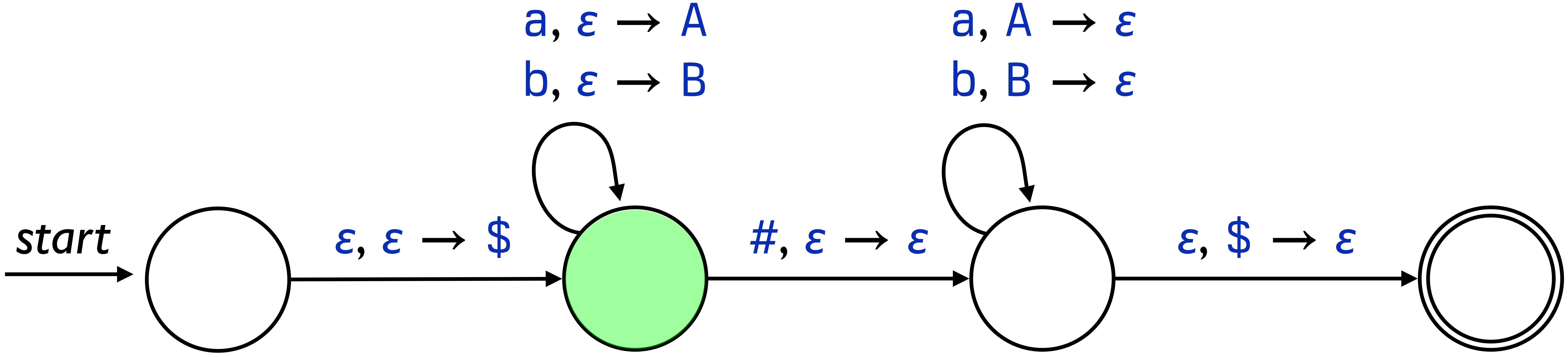
a a a b # b a a a



*Stack*

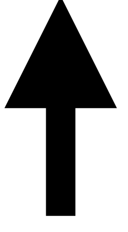


# Example: Palindromes

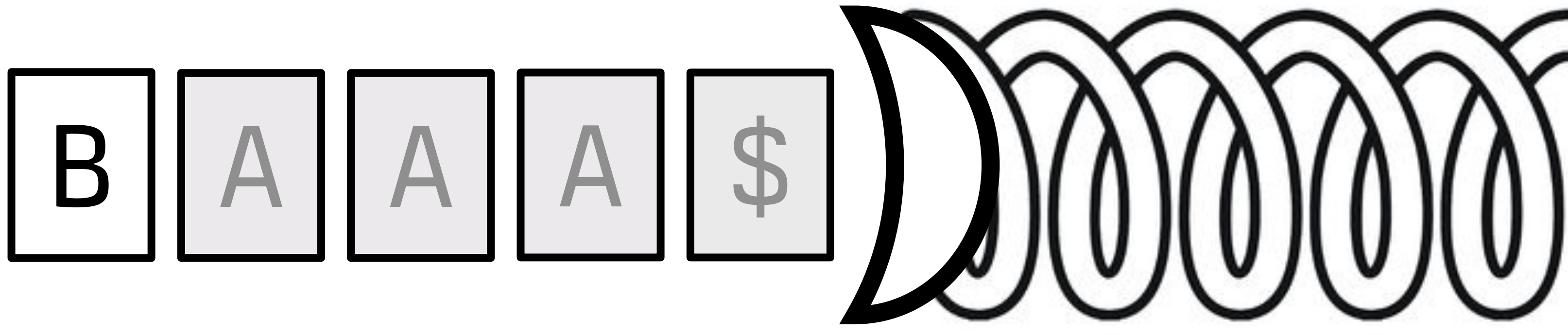



*Input*

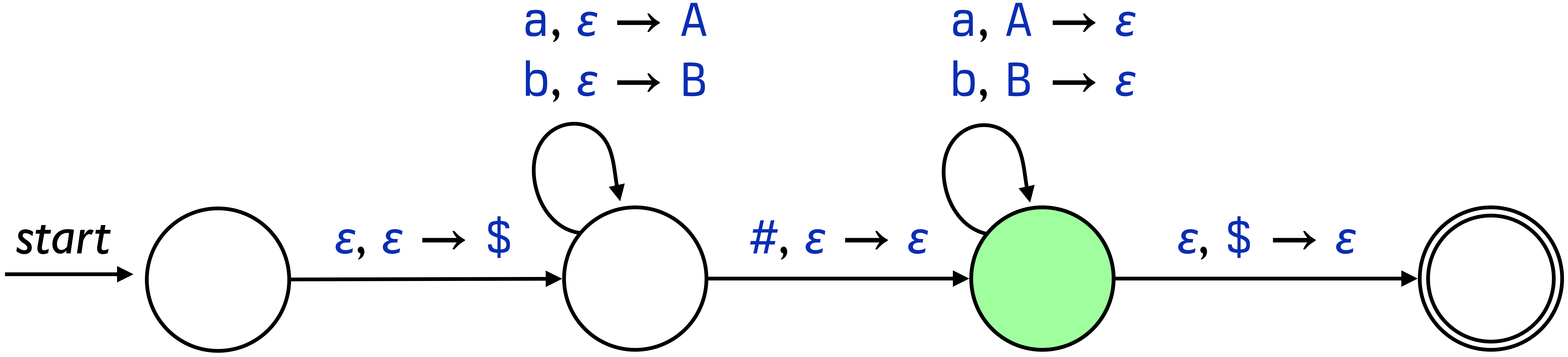
a a a b # b a a a



*Stack*

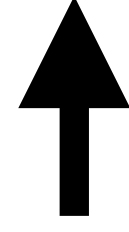


# Example: Palindromes

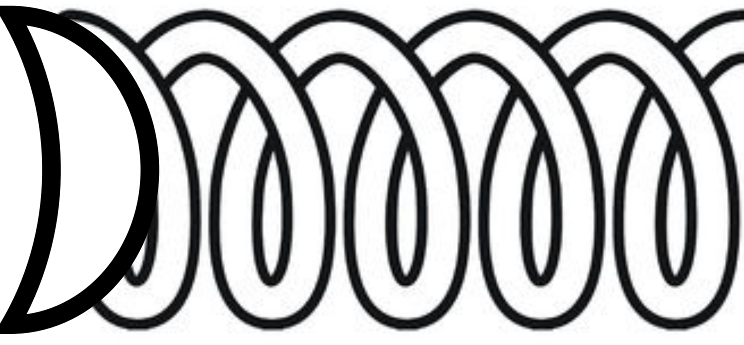



*Input*

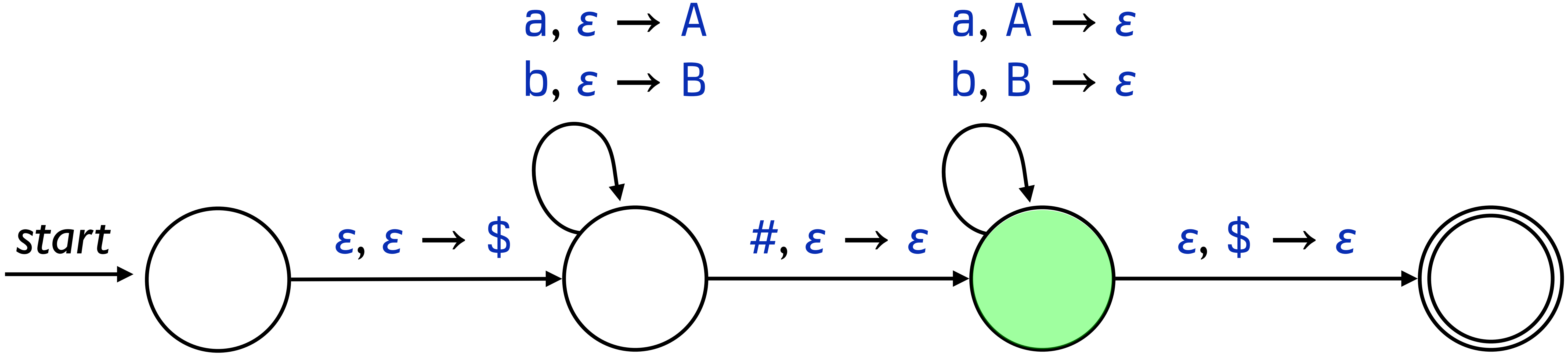
a a a b # b a a a



*Stack*

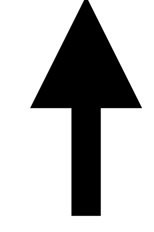


# Example: Palindromes

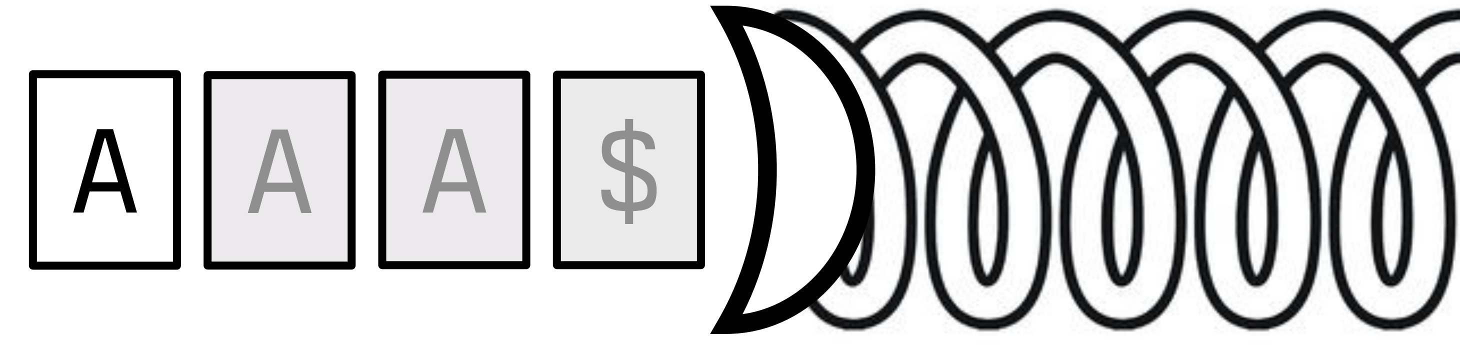



*Input*

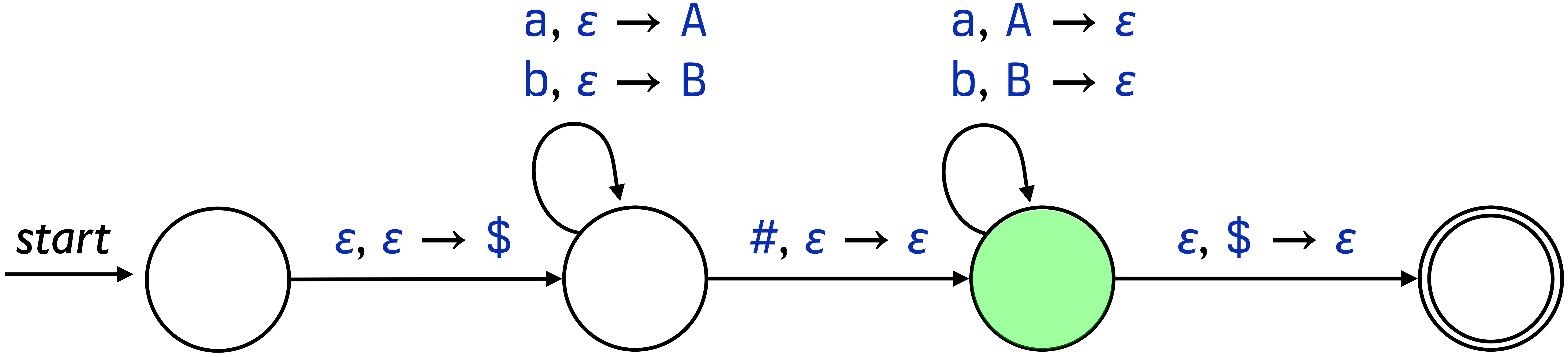
a a a b # b a a a



*Stack*

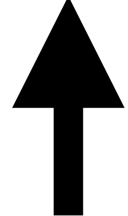


# Example: Palindromes

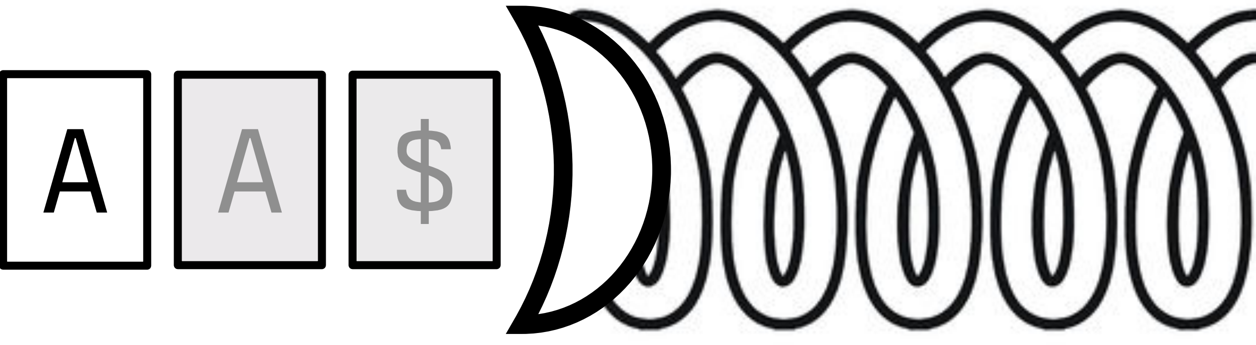



*Input*

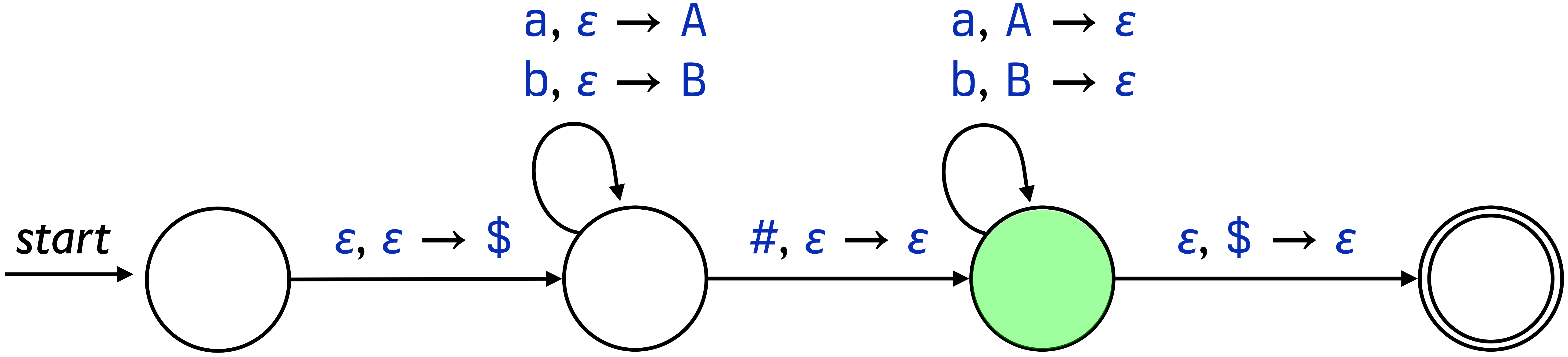
a a a b # b a a a



*Stack*

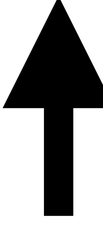


# Example: Palindromes

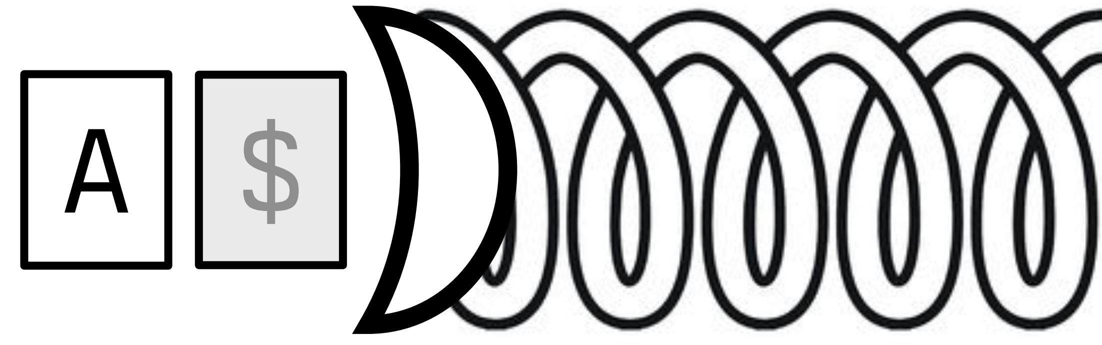



*Input*

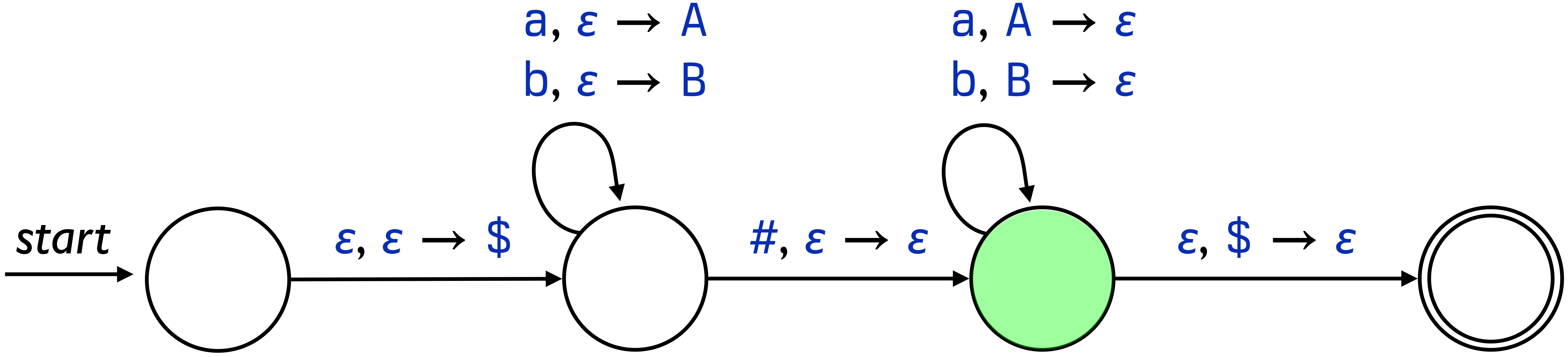
a a a b # b a a a



*Stack*

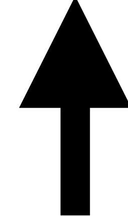


# Example: Palindromes

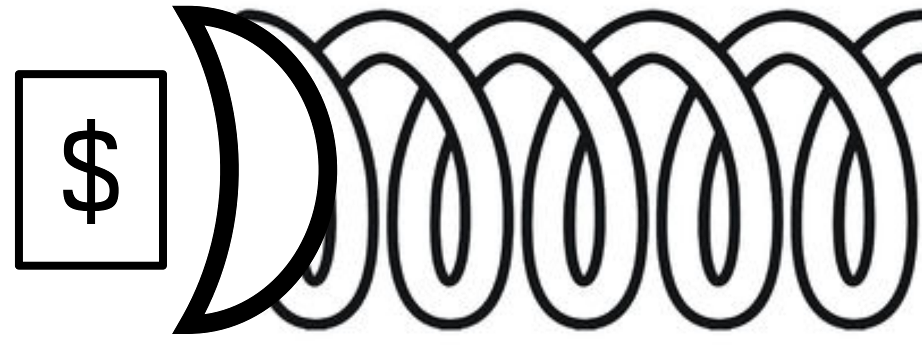



*Input*

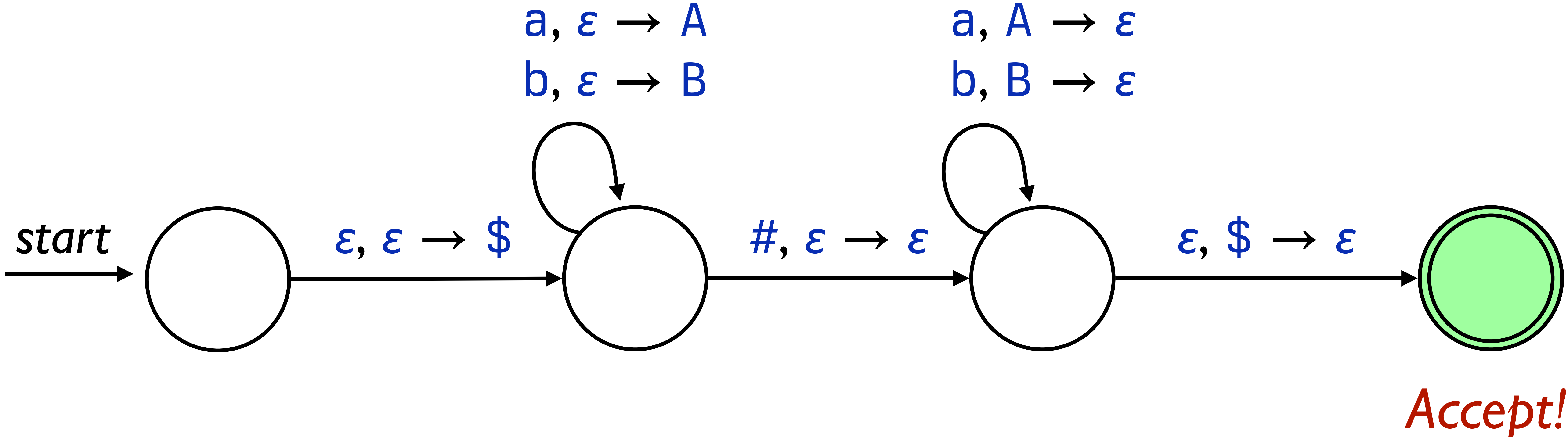
a a a b # b a a a



*Stack*

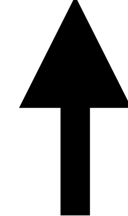


# Example: Palindromes

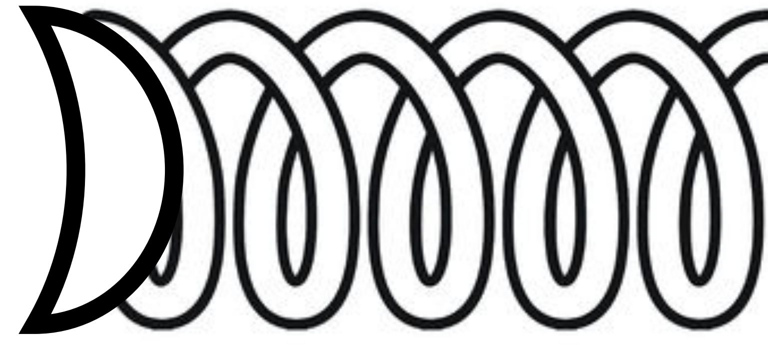



*Input*

a a a b # b a a a



*Stack*



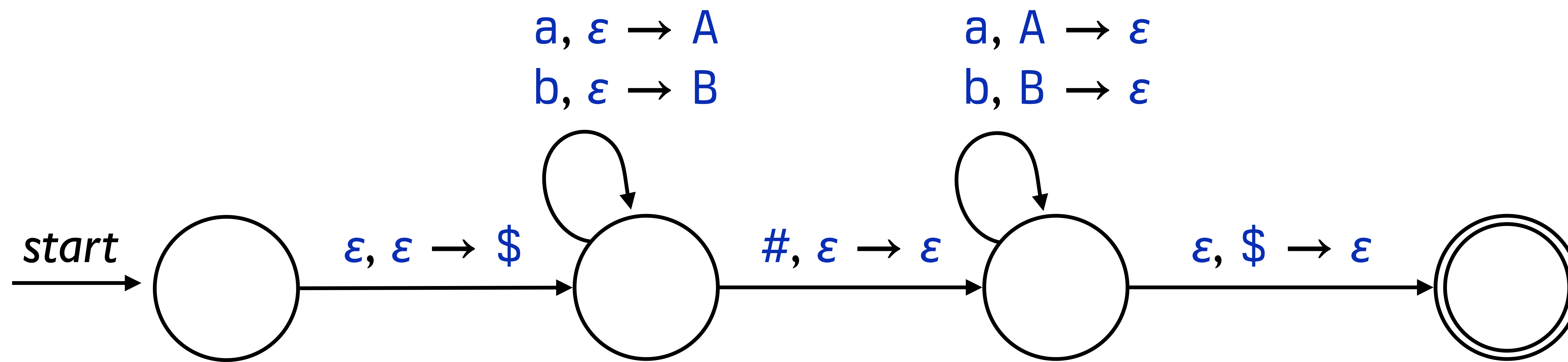
What about building a PDA to recognize the language of palindromes without a special dividing character like #?

Let $\Sigma$ = {a, b} and consider the language

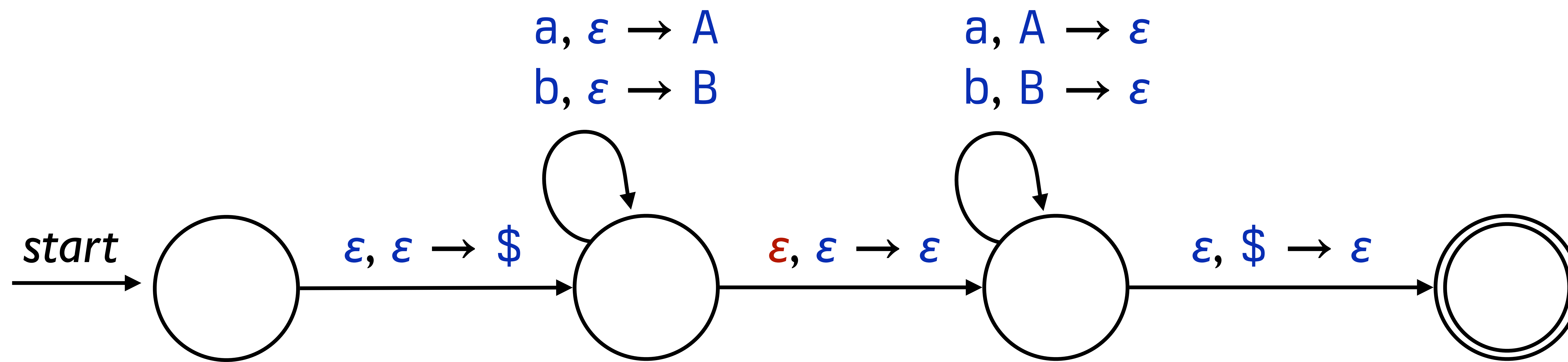
$$PALINDROME = \{w \in \Sigma^* \mid w \text{ is a palindrome}\}.$$

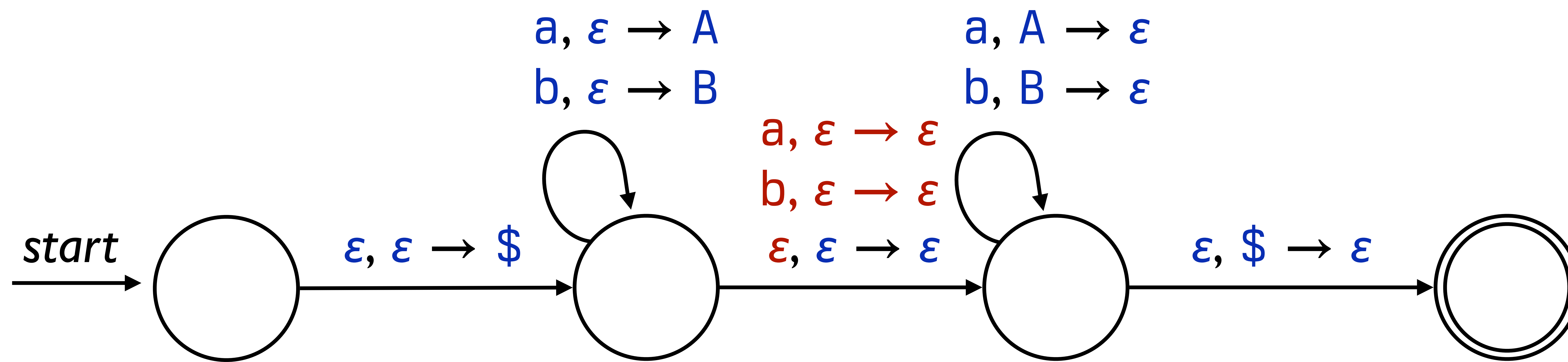
How would we build a PDA for *PALINDROME*?

Nondeterminism to the rescue!

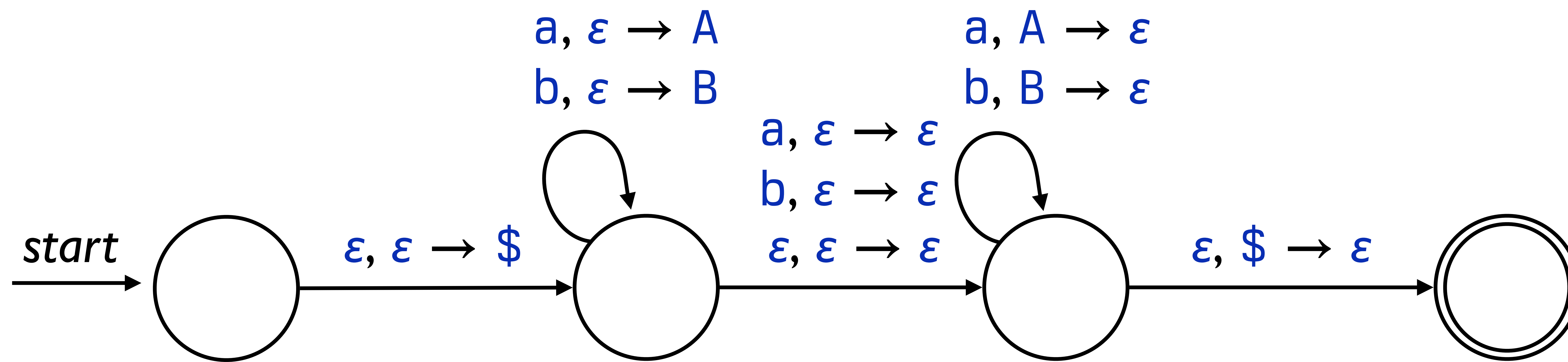


a, ε → A
b, ε → B
a, A → ε
b, B → ε
start
ε, ε → $
#, ε → ε
ε, $ → ε


a, ε → A
b, ε → B
a, A → ε
b, B → ε
start
ε, ε → $
ε, ε → ε
ε, $ → ε



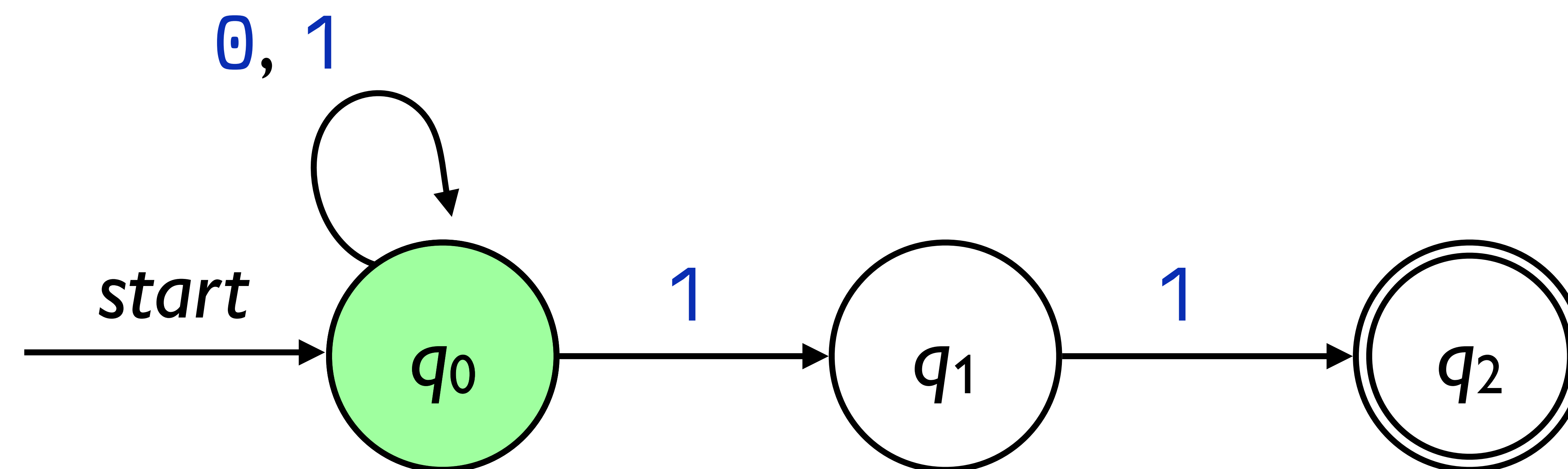
start
ε, ε → $
a, ε → A
b, ε → B
a, ε → ε
b, ε → ε
ε, ε → ε
a, A → ε
b, B → ε
ε, $ → ε


start
ε, ε → $
a, ε → A
b, ε → B
a, ε → ε
b, ε → ε
ε, ε → ε
a, A → ε
b, B → ε
ε, $ → ε

# A note on nondeterminism

In an NFA, we could interpret nondeterminism as being in multiple states simultaneously.

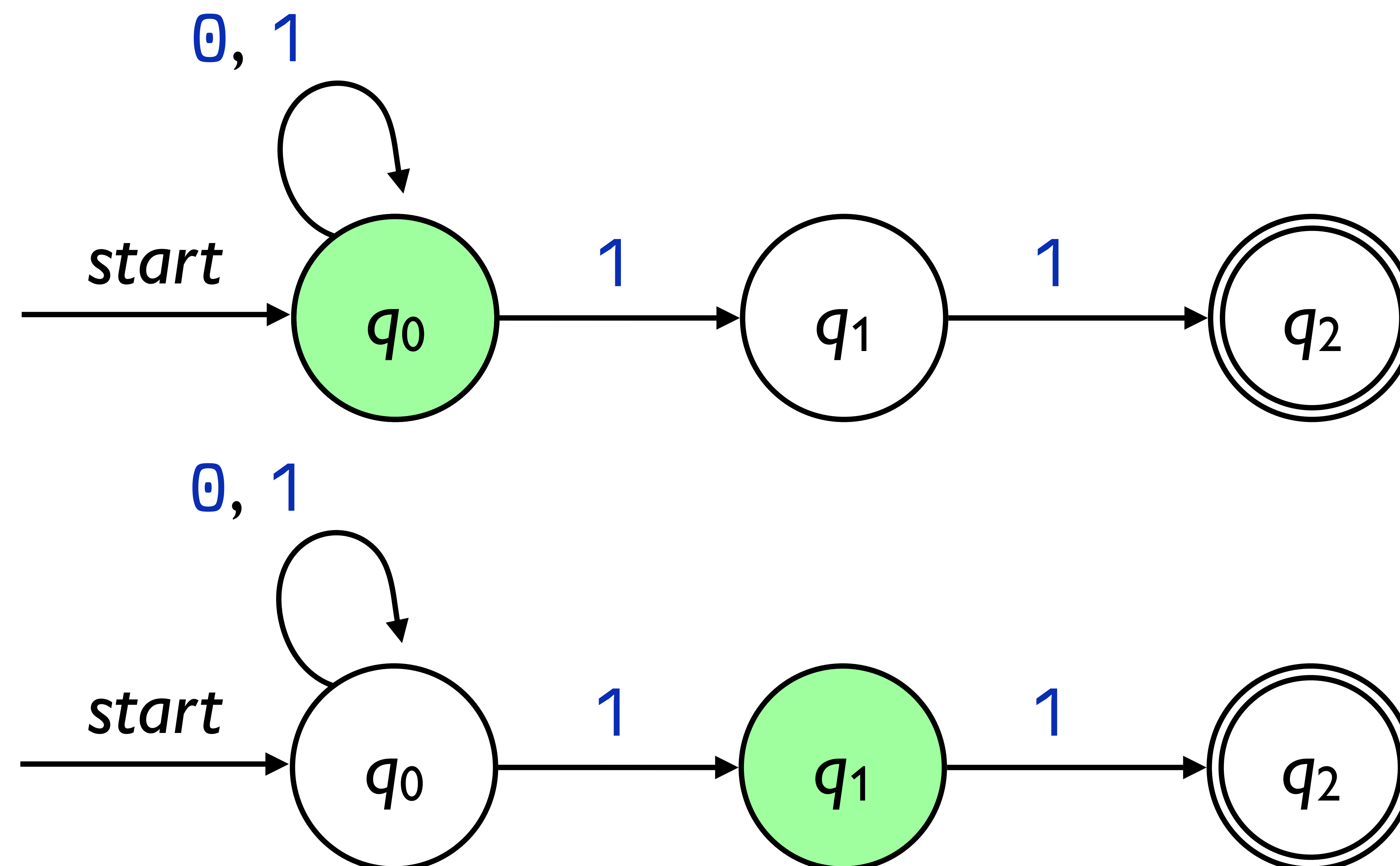
This is only possible because NFAs have no extra storage.



# A note on nondeterminism

In an NFA, we could interpret nondeterminism as being in multiple states simultaneously.

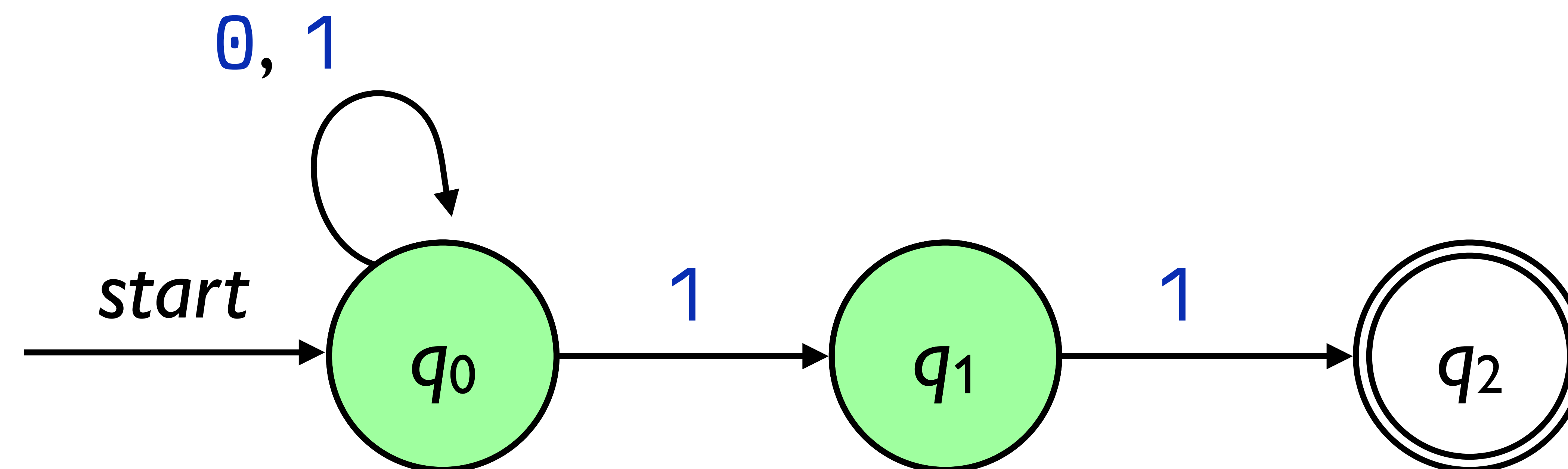
This is only possible because NFAs have no extra storage.



# A note on nondeterminism

In an NFA, we could interpret nondeterminism as being in multiple states simultaneously.

This is only possible because NFAs have no extra storage.



# A note on nondeterminism

In a PDA, if there are multiple nondeterministic choices, you *cannot* treat the machine as being in multiple states at once.

Each state might have its own stack associated with it.

Instead, there are multiple parallel copies of the machine running at once, each of which has its own stack.

# Exercise

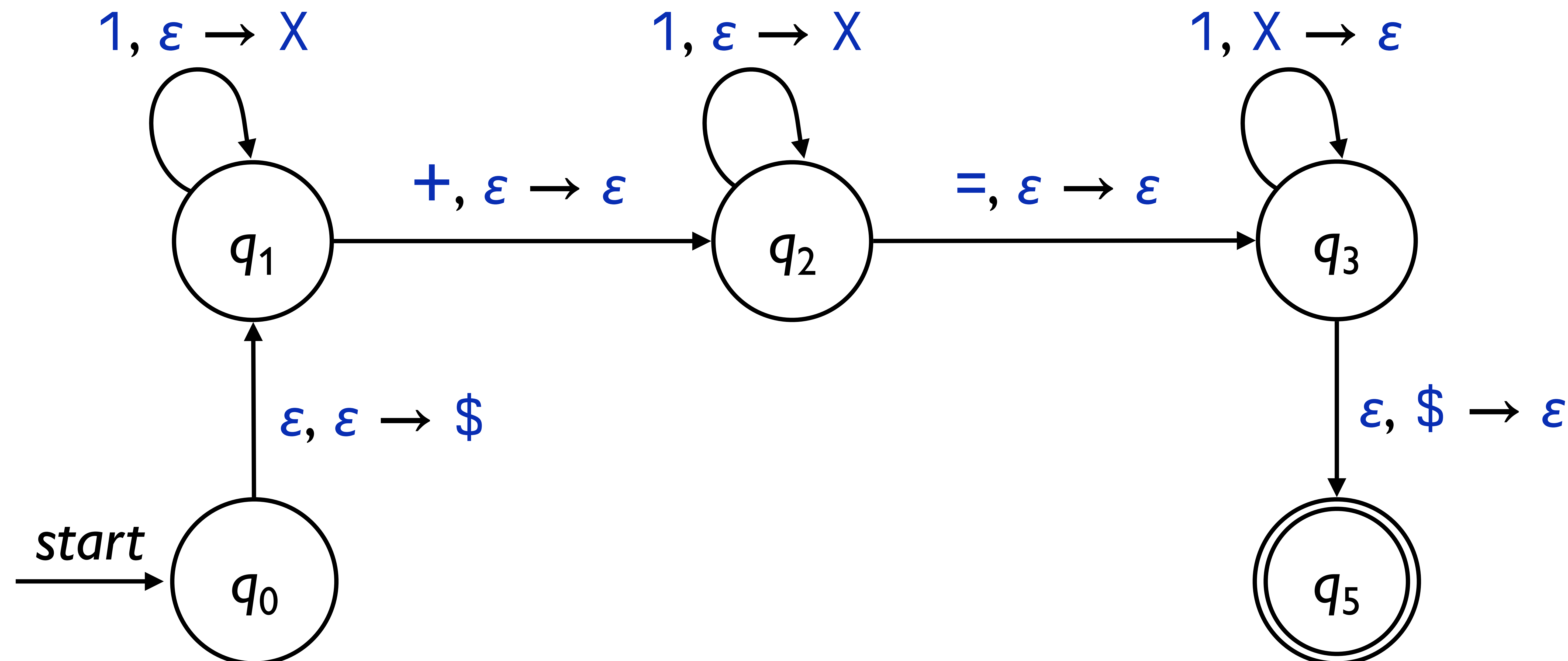
Design a PDA to recognize the language

$$ADD = \{1^m+1^n=1^{m+n} \mid m, n \in \mathbb{N}_0\}.$$

# Exercise

Design a PDA to recognize the language

$$ADD = \{1^m+1^n=1^{m+n} \mid m, n \in \mathbb{N}_0\}.$$



# A PDA for arithmetic

Let $\Sigma$ = {`int`, `+`, `×`, `(`, `)`}.

Consider the language

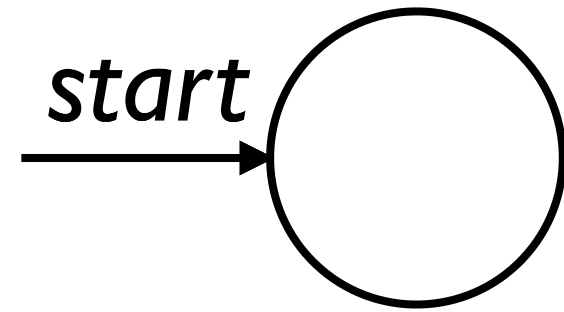
$$ARITH = \{w \in \Sigma^* \mid w \text{ is a legal arithmetic expression}\}.$$

For example,

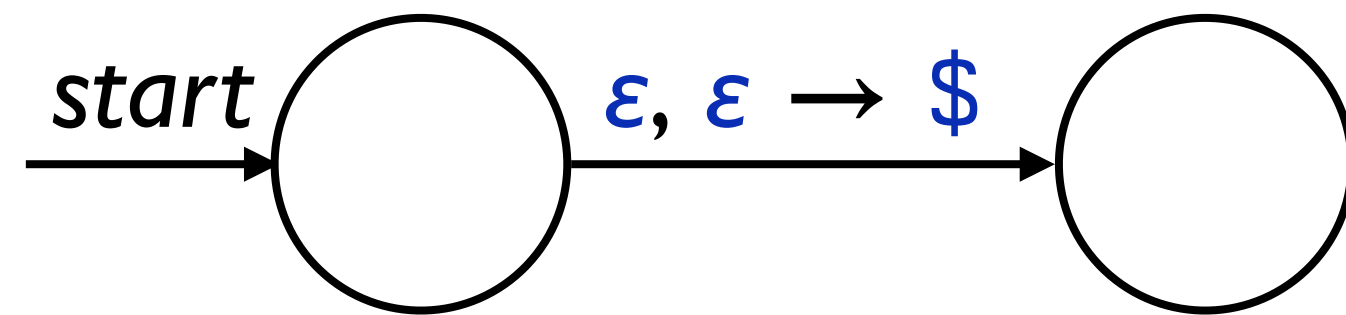
```
int + int × int
((int + int) × (int + int)) + (int)
```

Can we build a PDA for *ARITH*?

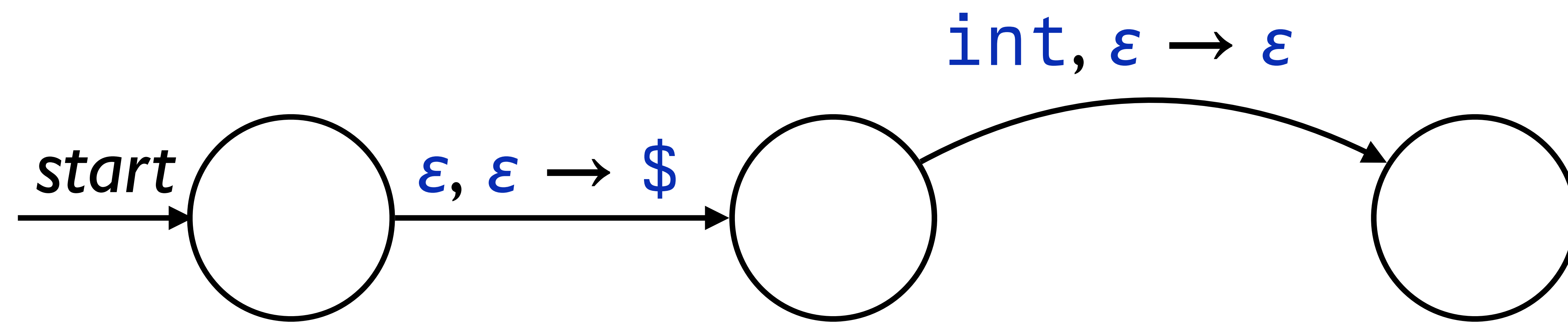
# A PDA for arithmetic



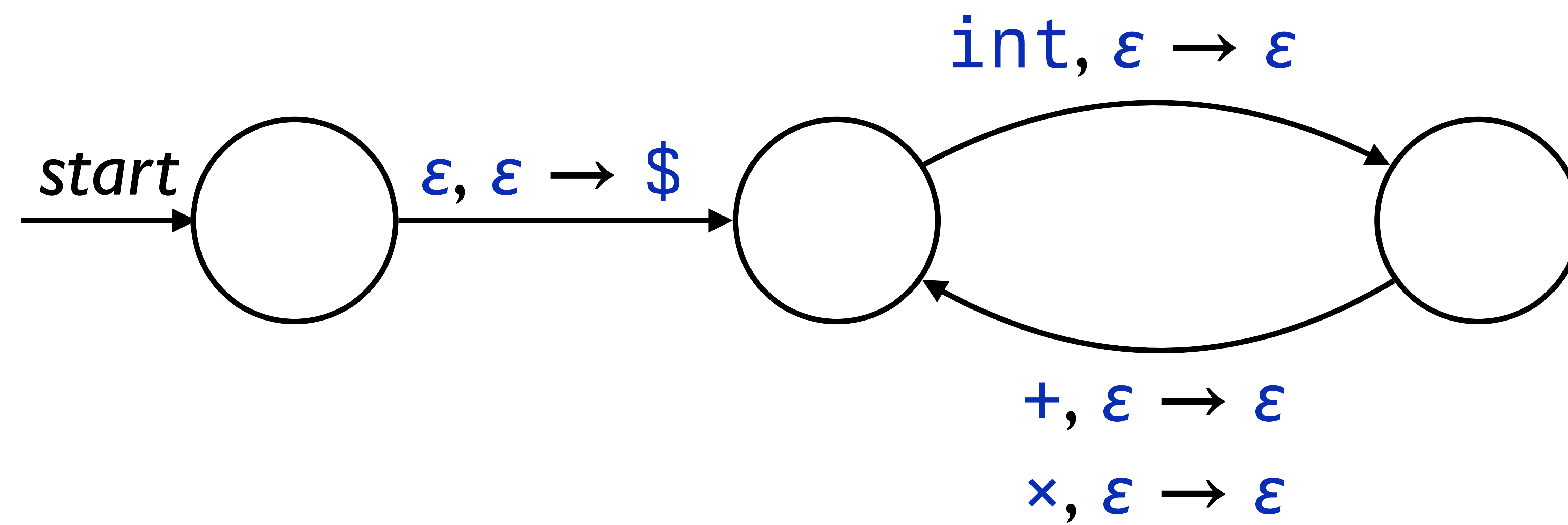
# A PDA for arithmetic



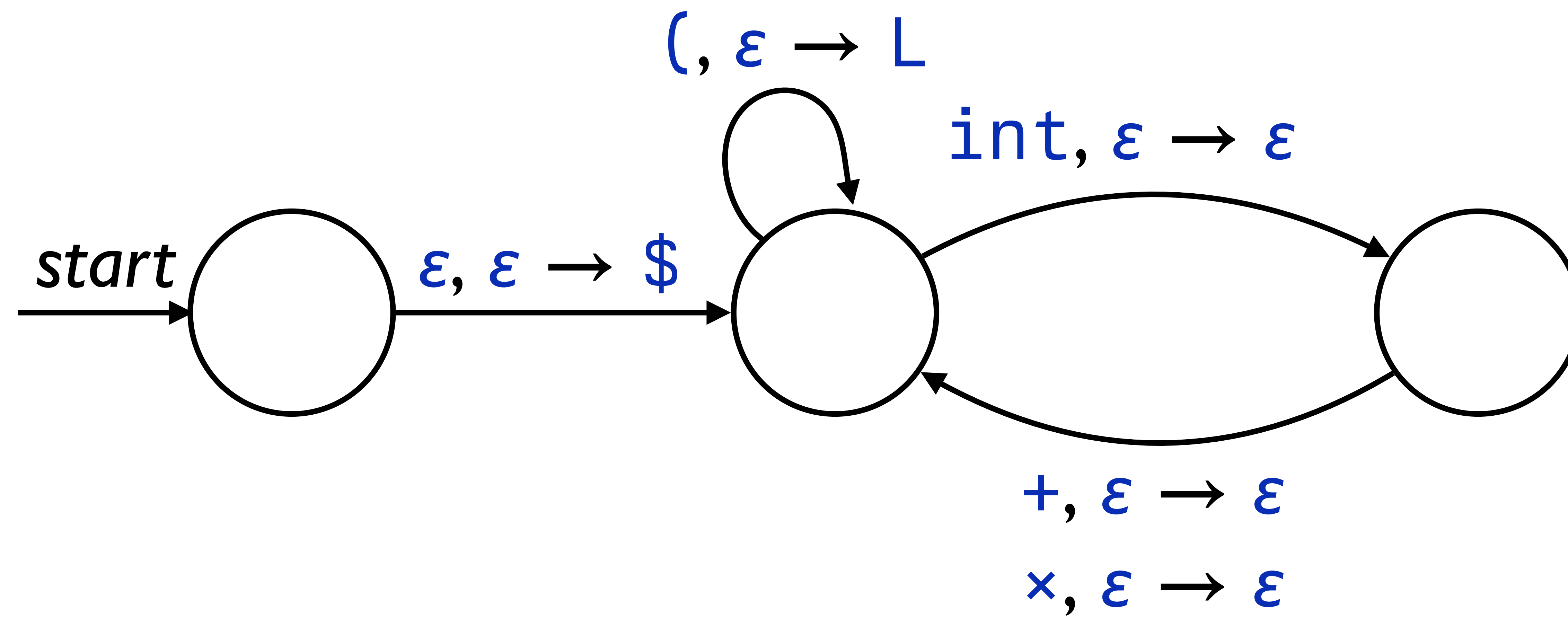
# A PDA for arithmetic



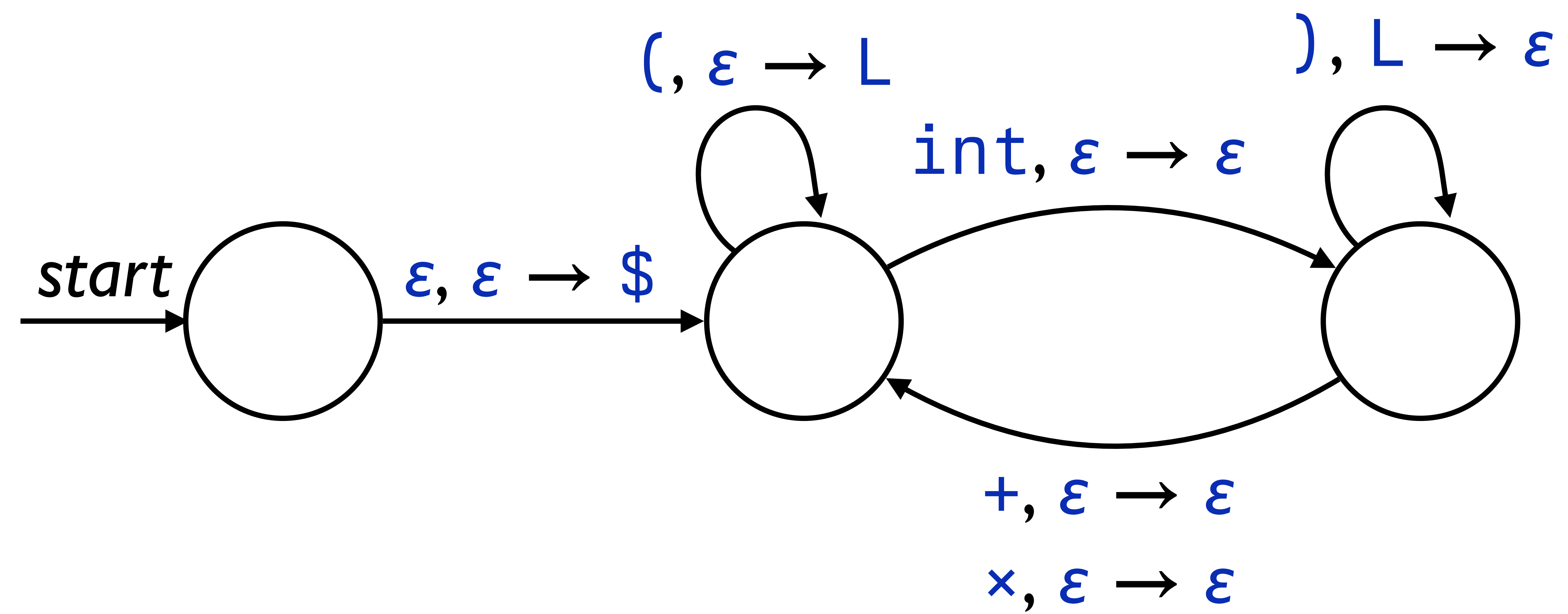
# A PDA for arithmetic



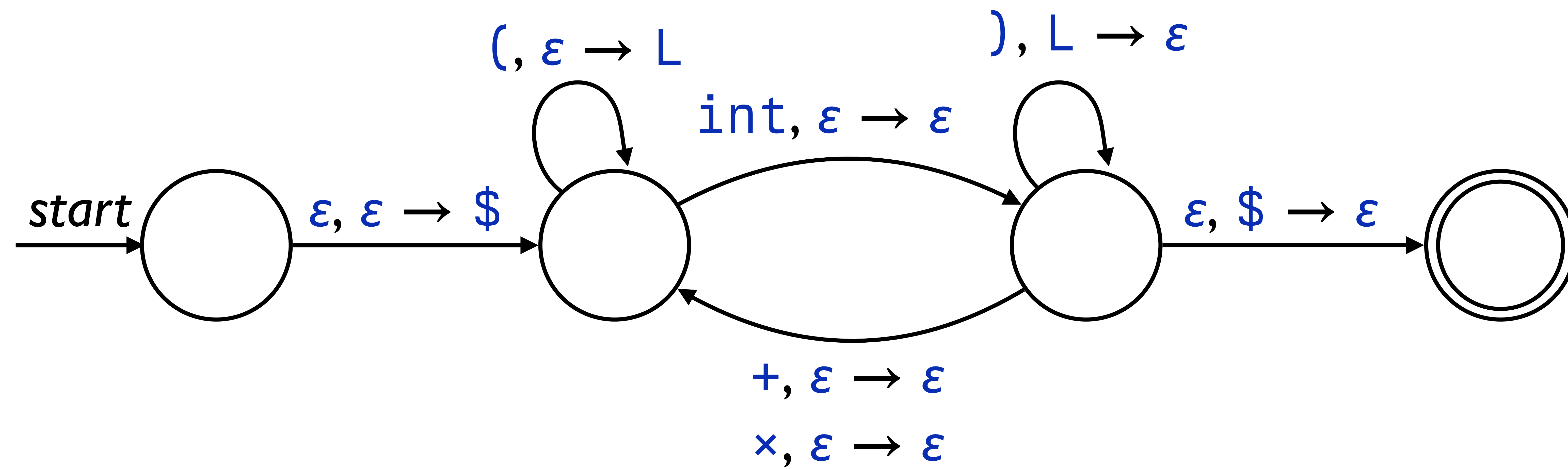
# A PDA for arithmetic



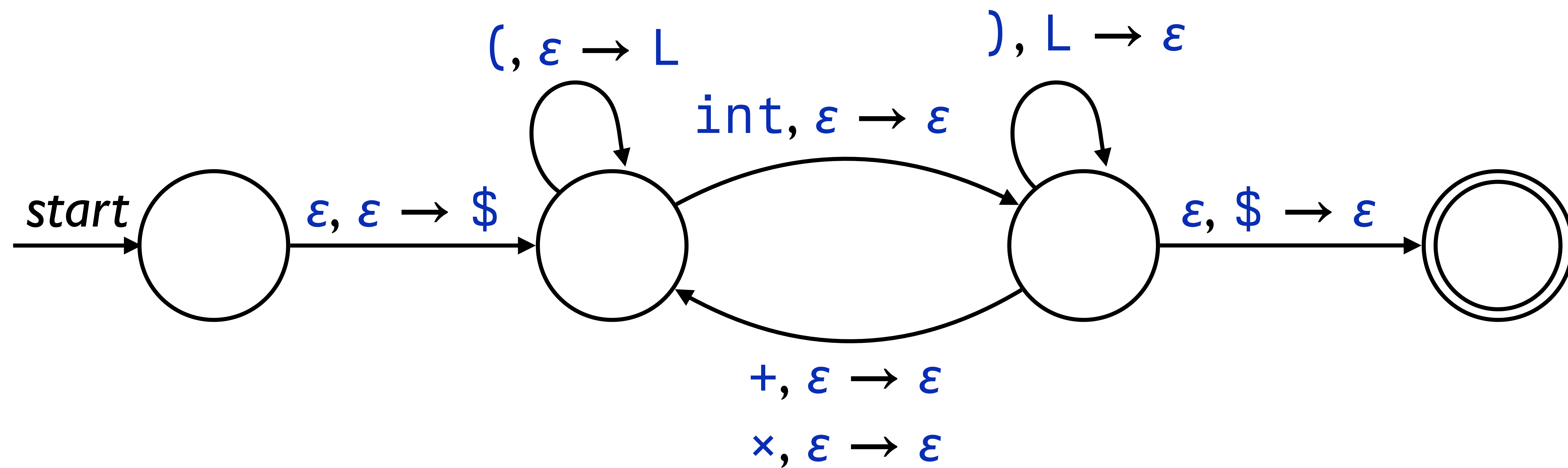
# A PDA for arithmetic



# A PDA for arithmetic



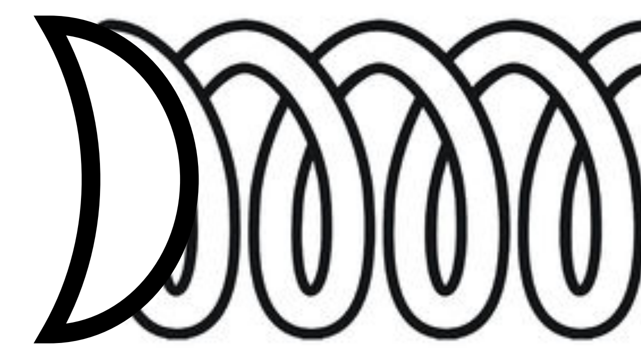
# A PDA for arithmetic



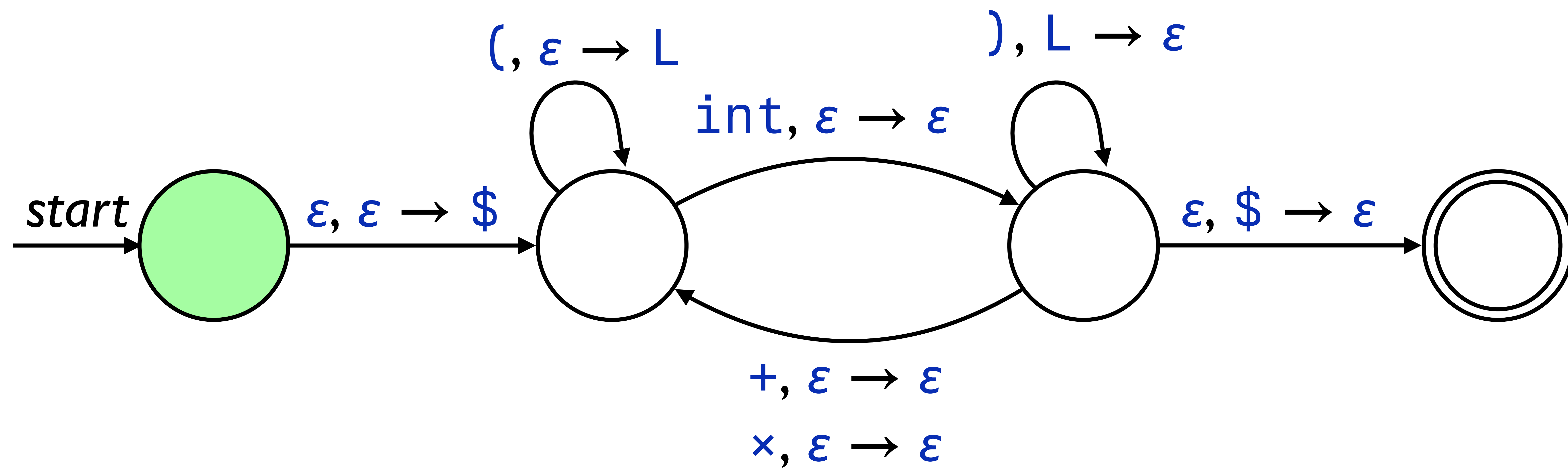

*Input*

*Stack*

```
int + int × int
```



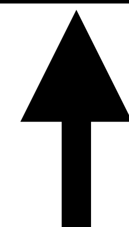
# A PDA for arithmetic

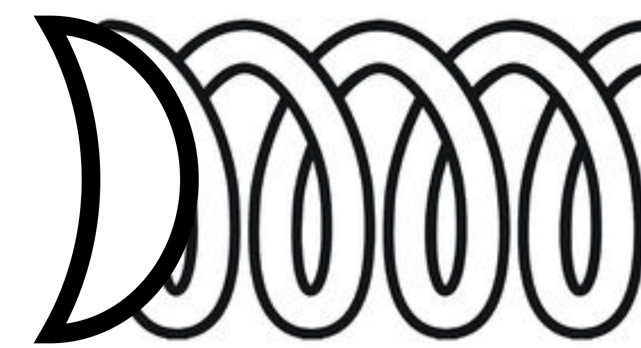



*Input*

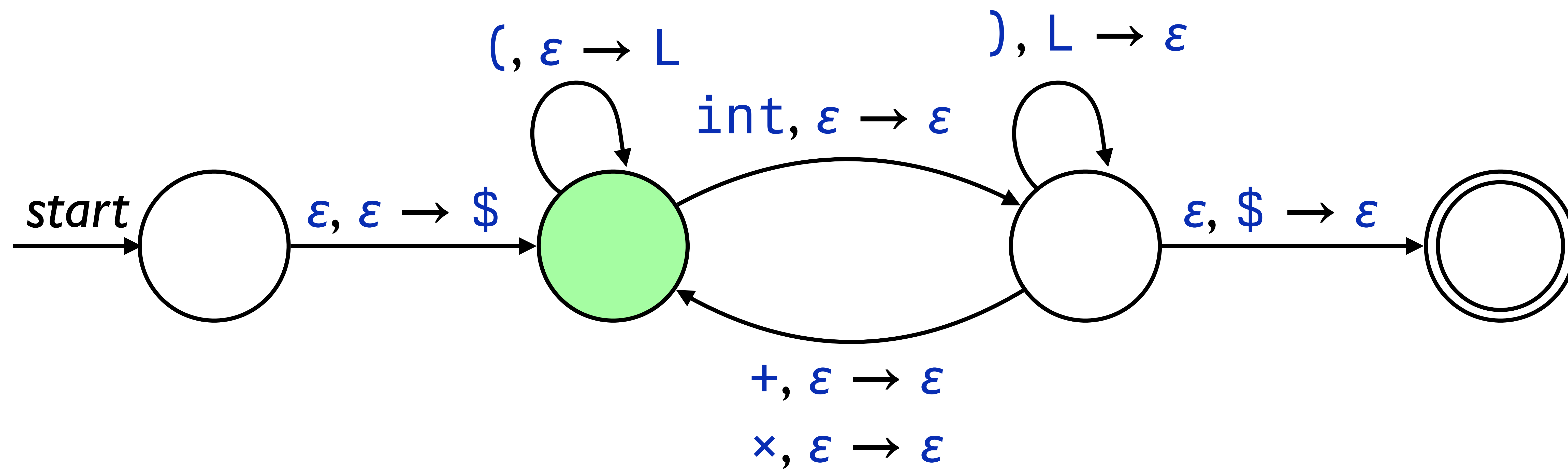
*Stack*

int + int × int





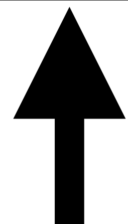
# A PDA for arithmetic

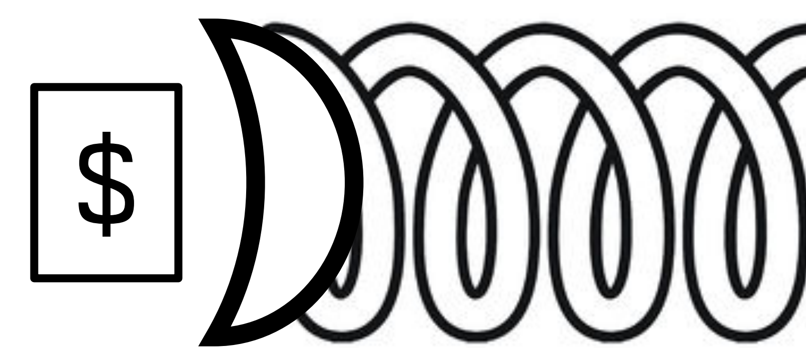



*Input*

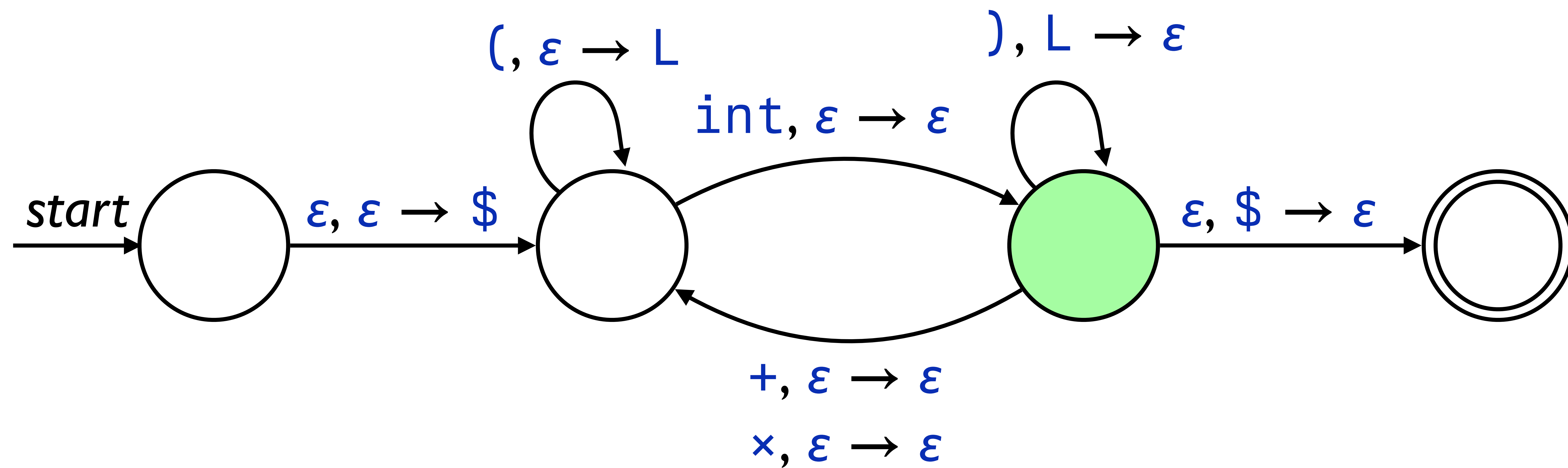
*Stack*

int + int × int



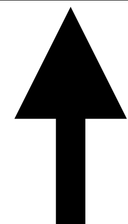


# A PDA for arithmetic

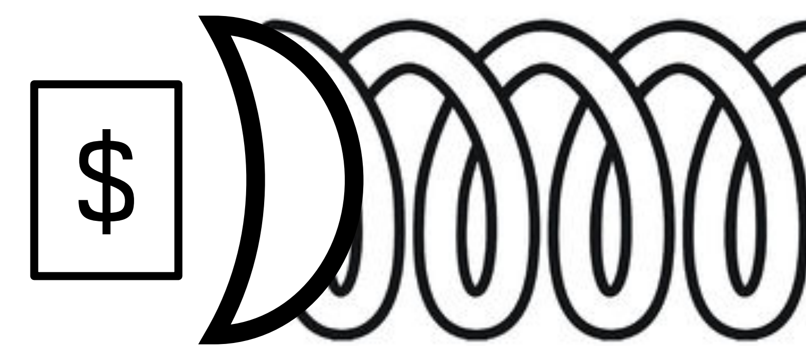



*Input*

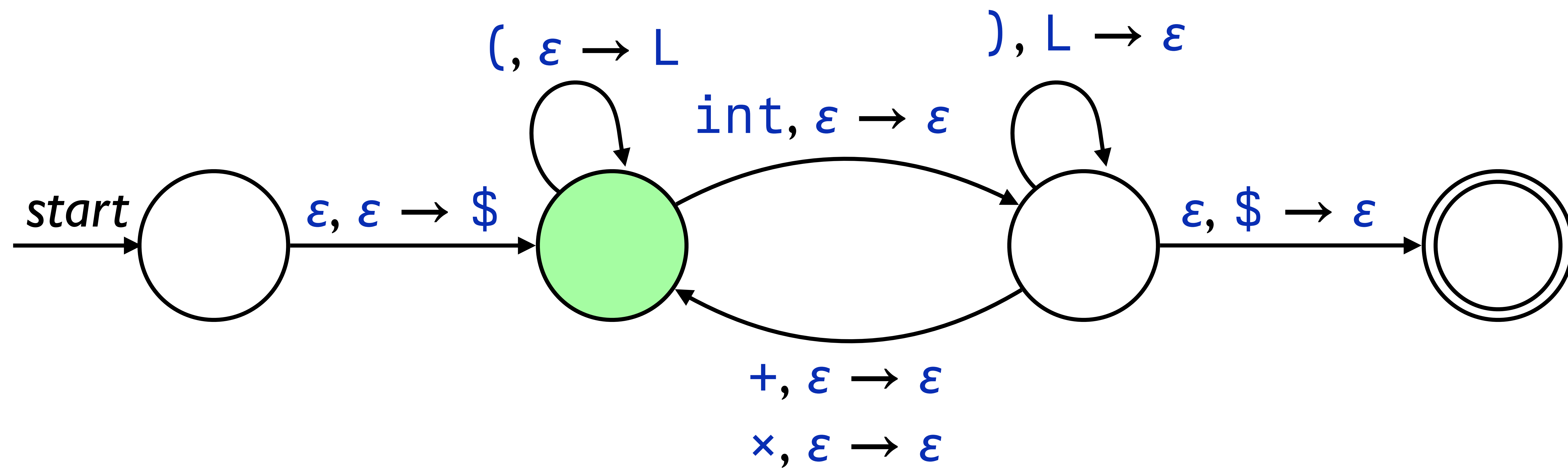
int + int × int



*Stack*

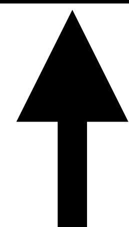


# A PDA for arithmetic

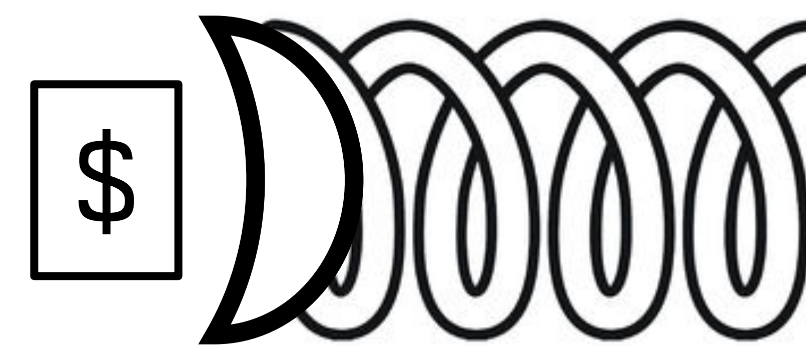



*Input*

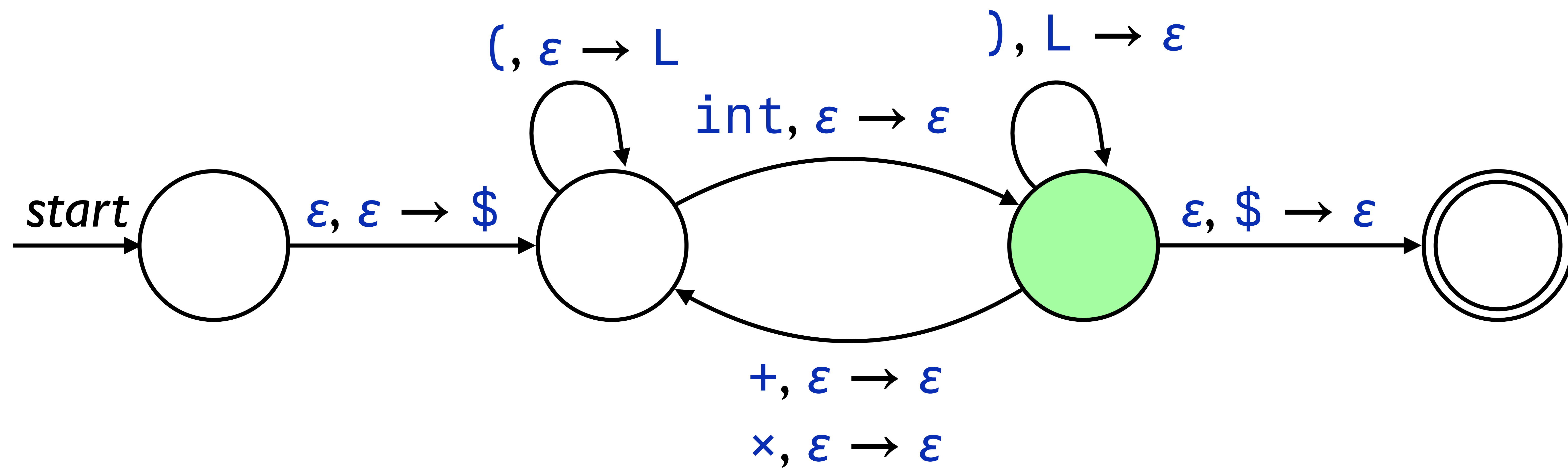
`int + int × int`



*Stack*

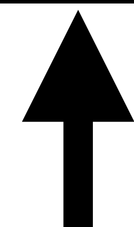


# A PDA for arithmetic

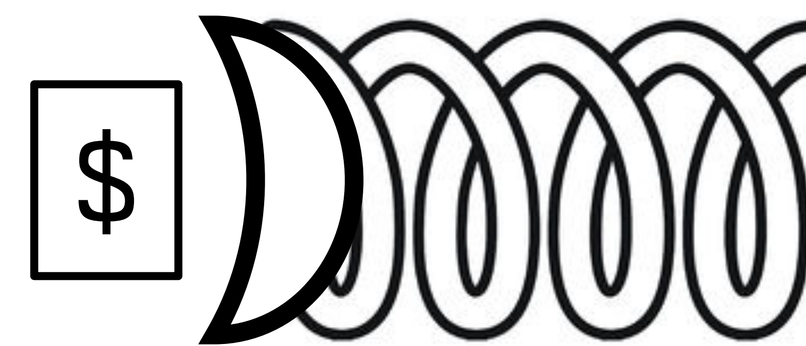



*Input*

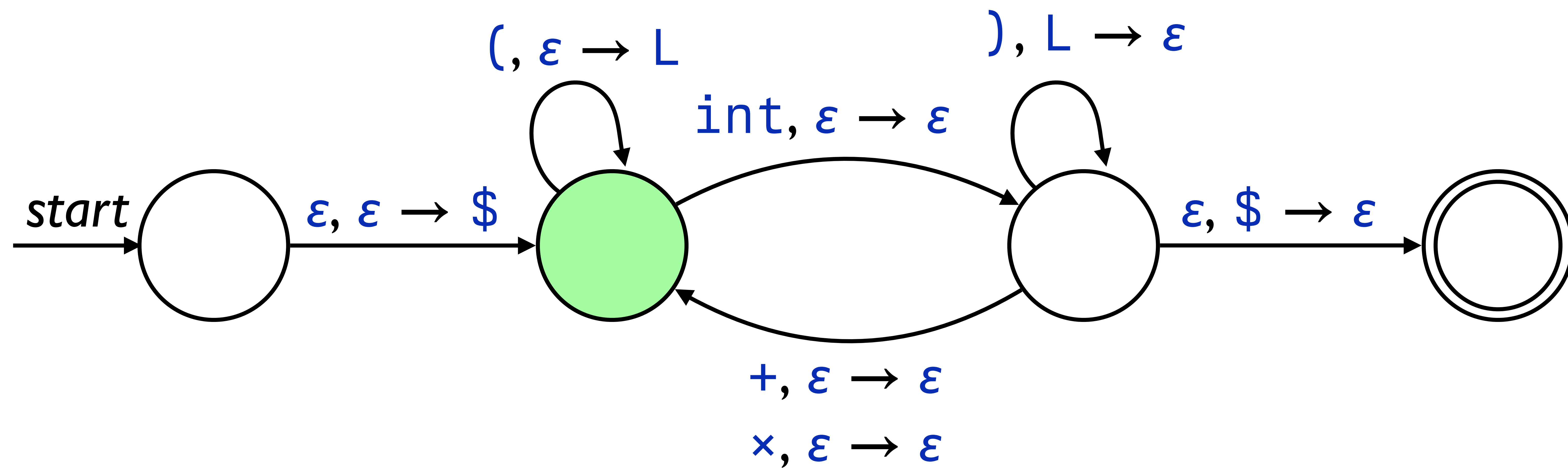
int + int × int



*Stack*

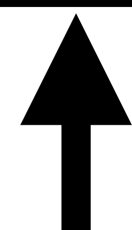


# A PDA for arithmetic

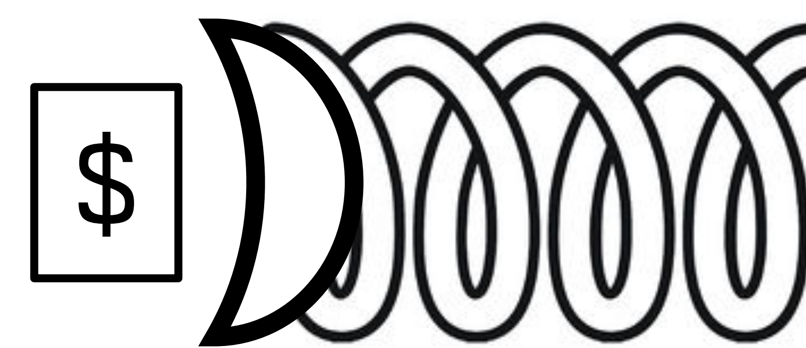



*Input*

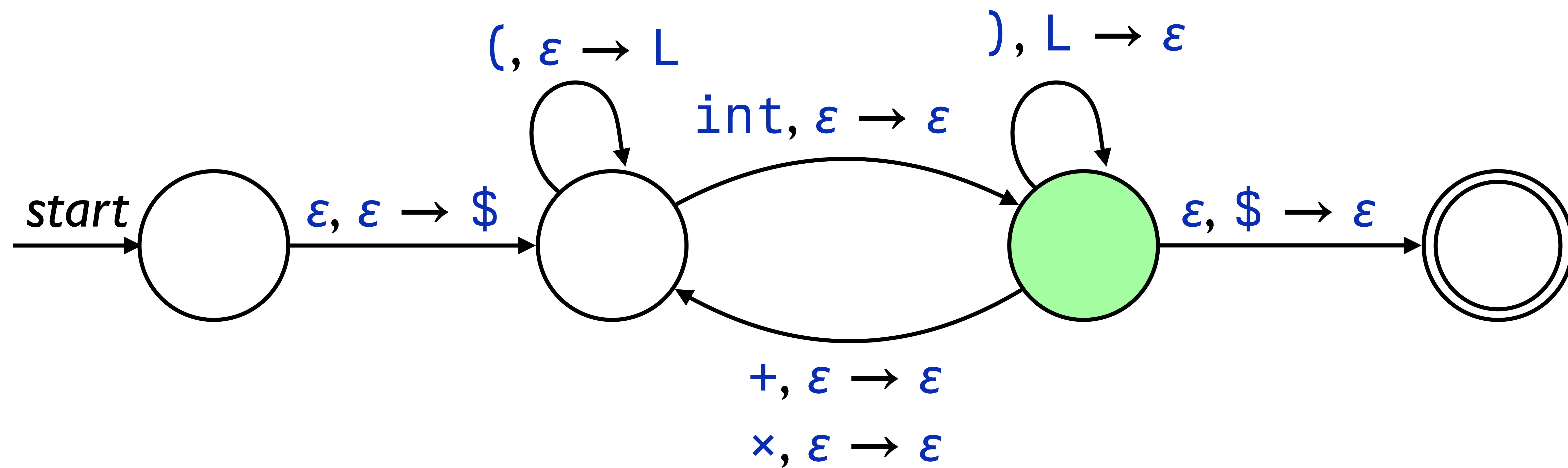
int + int × int



*Stack*

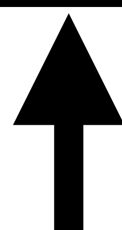


# A PDA for arithmetic

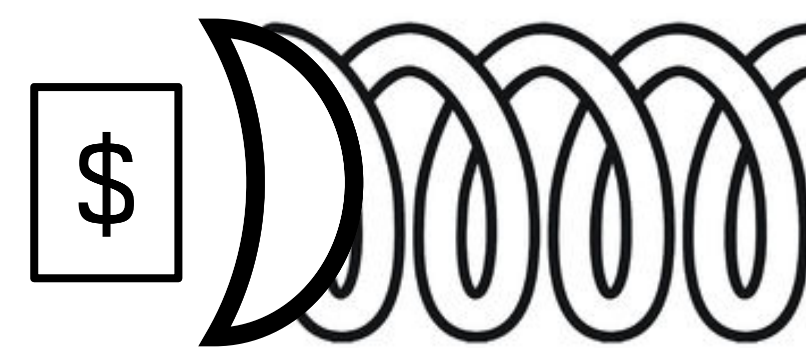



*Input*

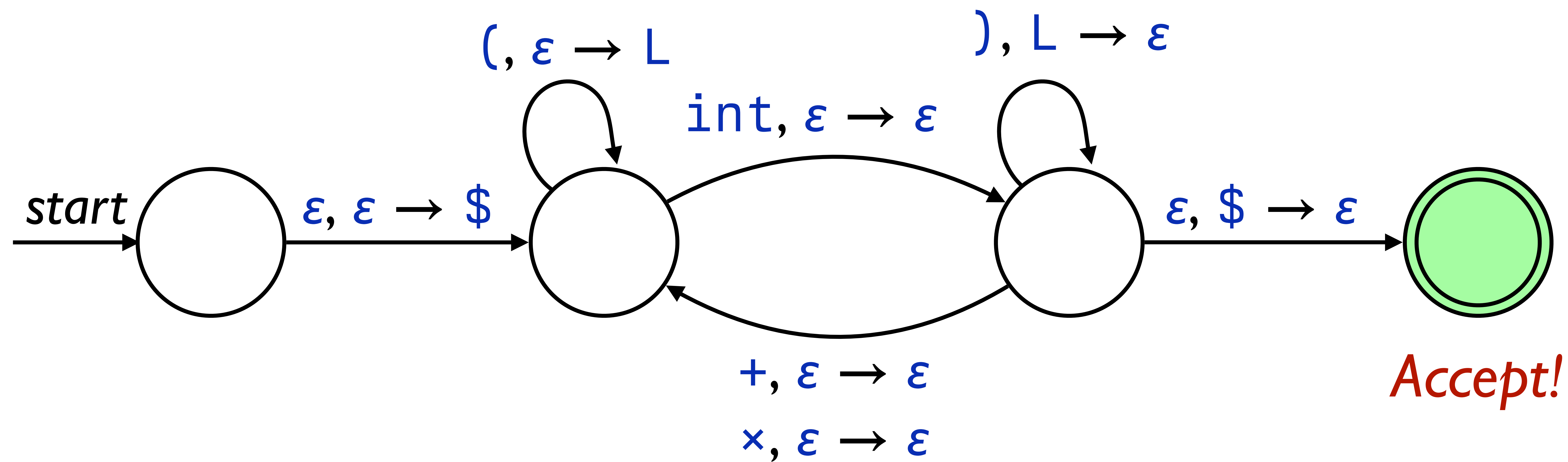
int + int × int



*Stack*



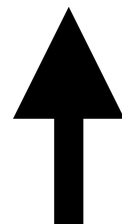
# A PDA for arithmetic

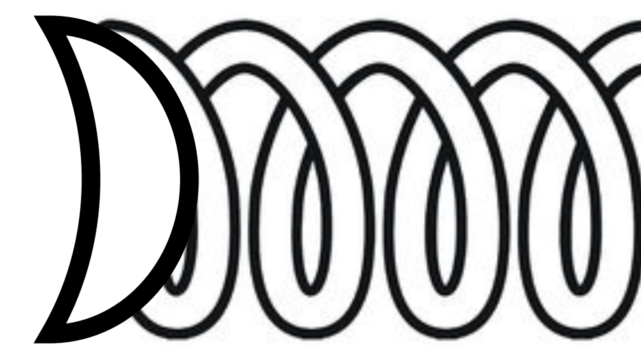



*Input*

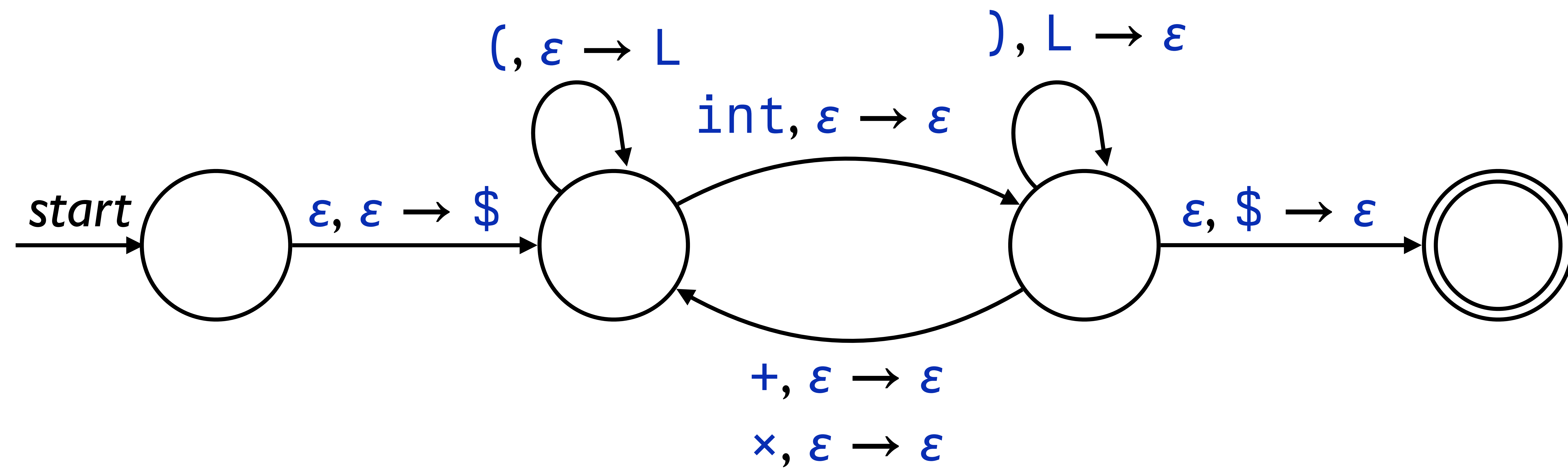
*Stack*

`int + int × int`



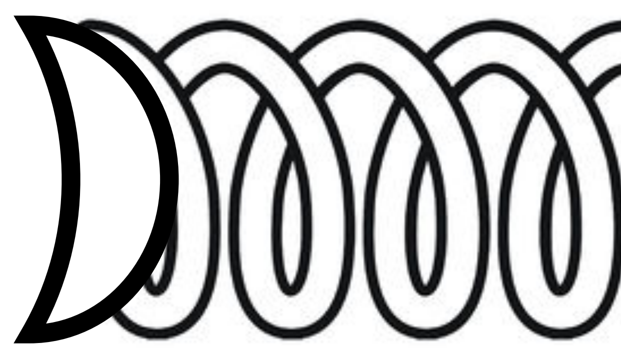


# A PDA for arithmetic

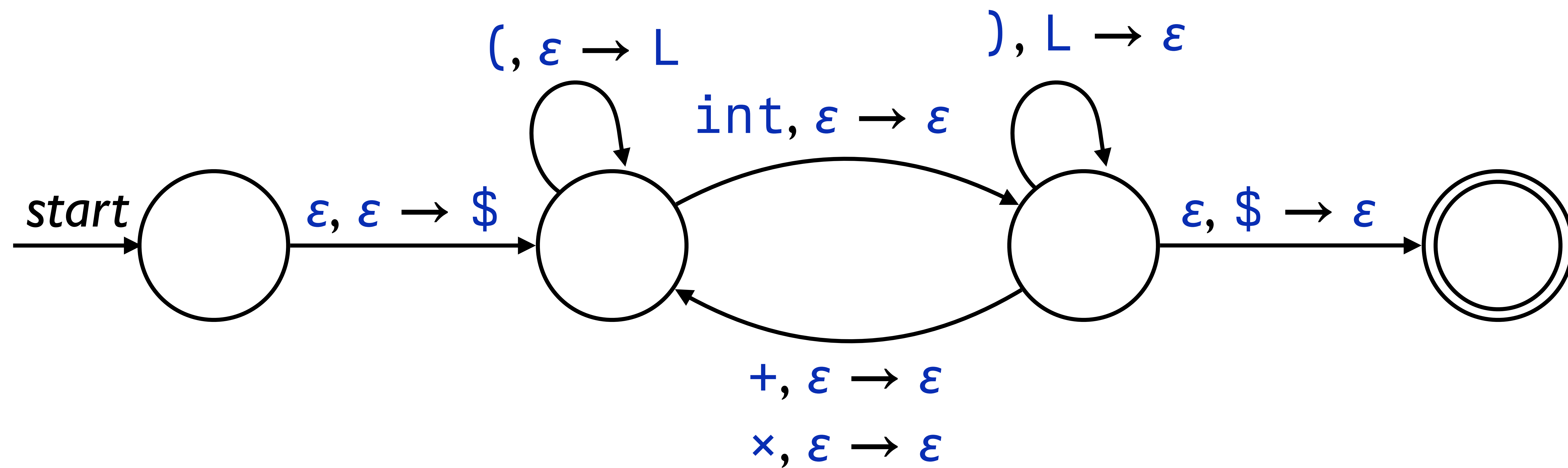



*Input*

*Stack*



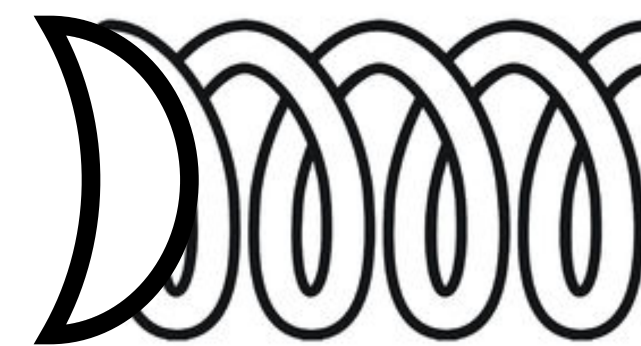
# A PDA for arithmetic



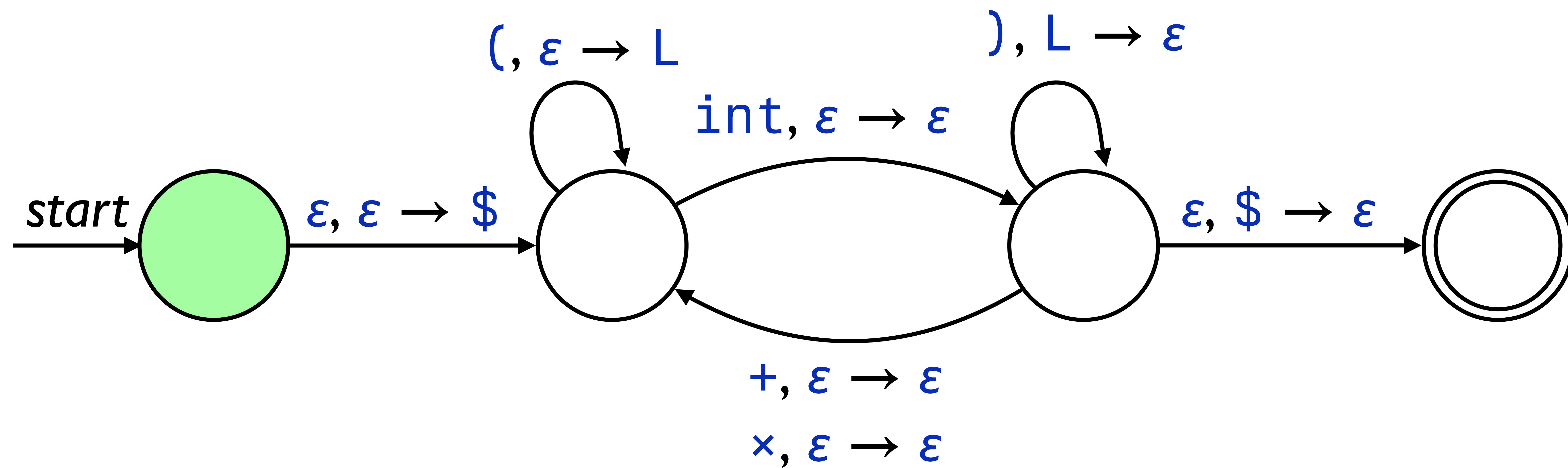

*Input*

*Stack*

`int + ( ( int × int ) + int )`



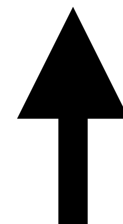
# A PDA for arithmetic

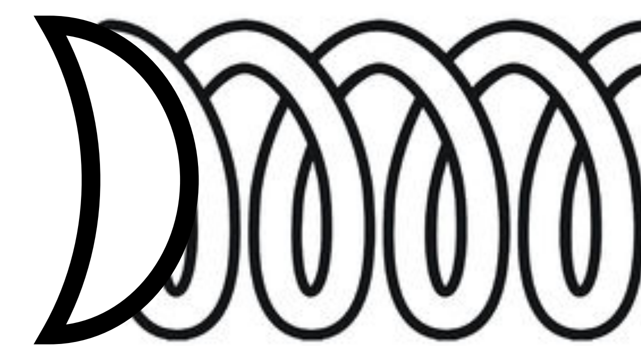



*Input*

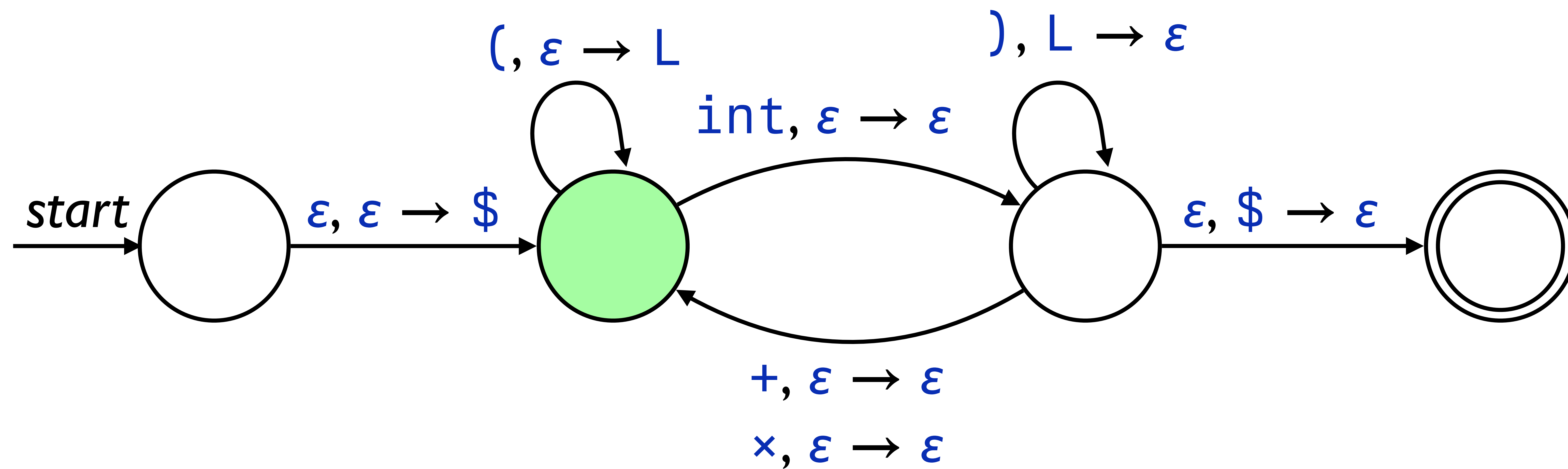
*Stack*

int + ( ( int × int ) + int )





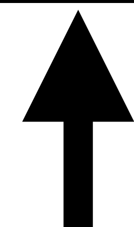
# A PDA for arithmetic

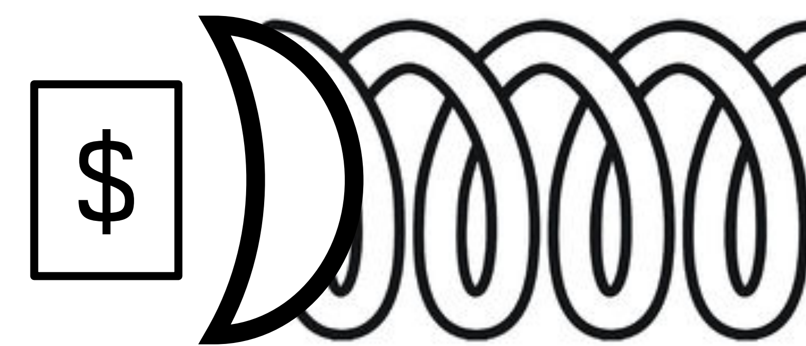



*Input*

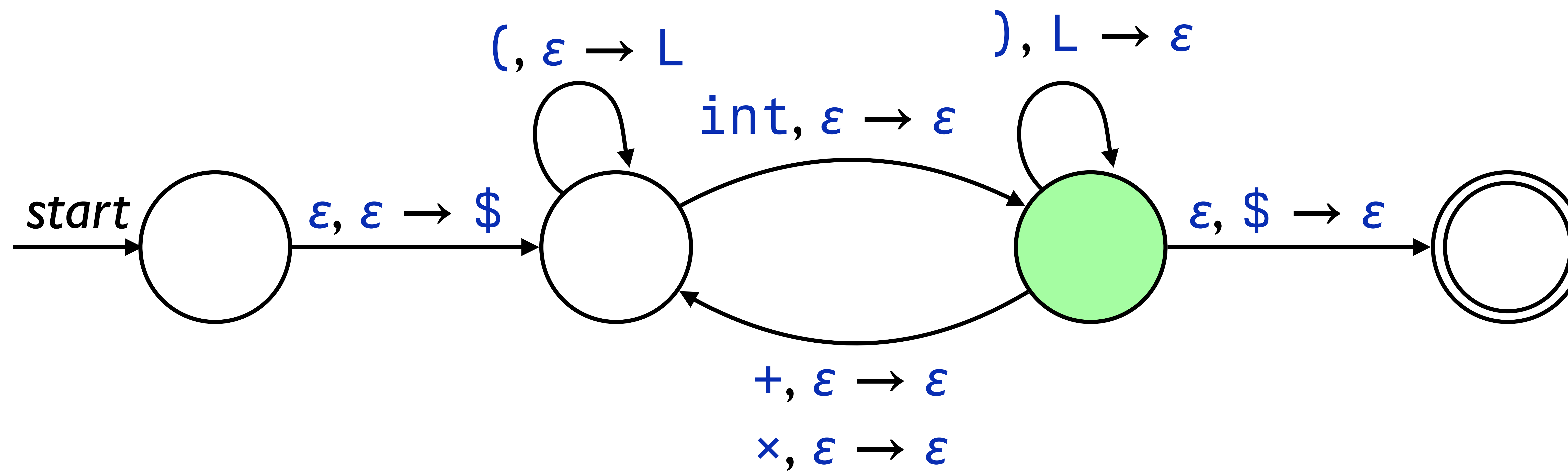
*Stack*

int + ( ( int × int ) + int )



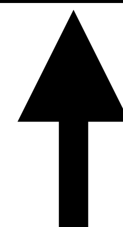


# A PDA for arithmetic

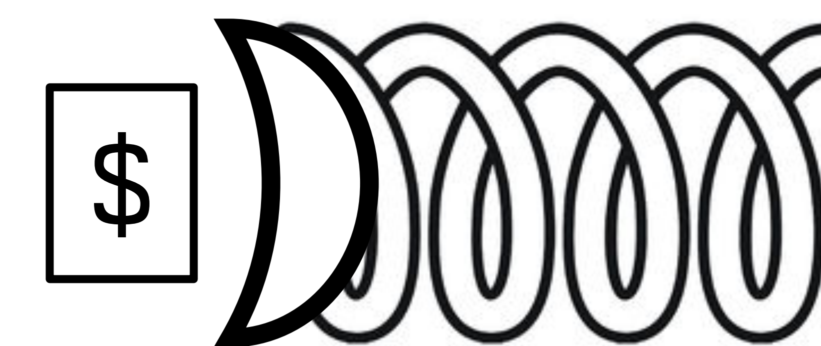



*Input*

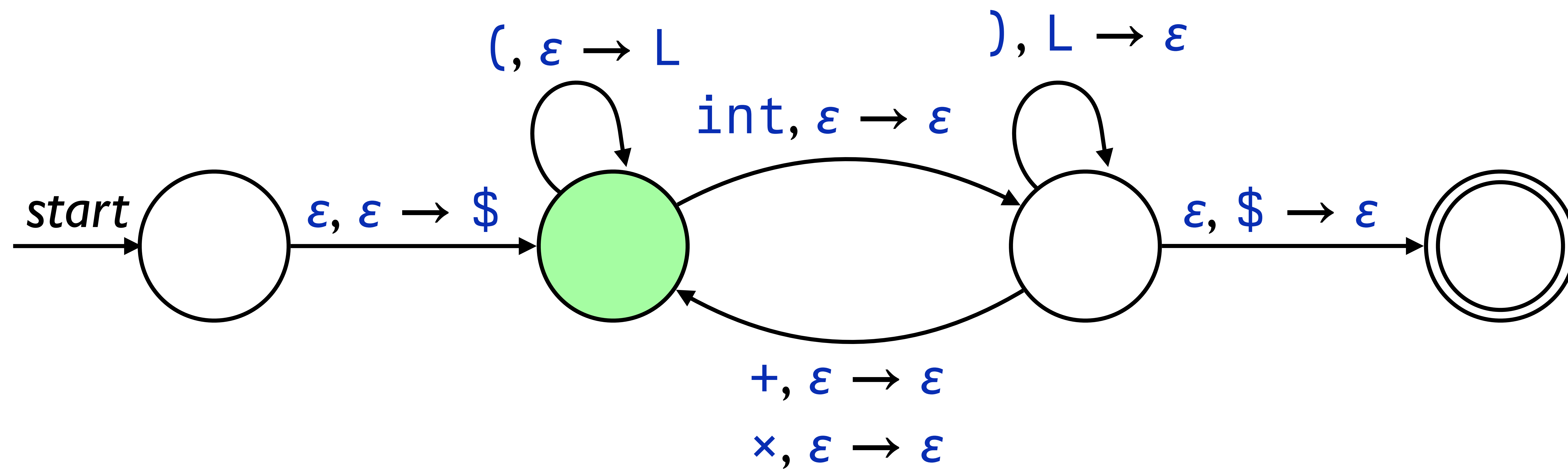
int + ( ( int × int ) + int )



*Stack*

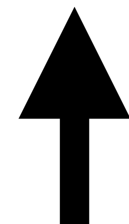


# A PDA for arithmetic

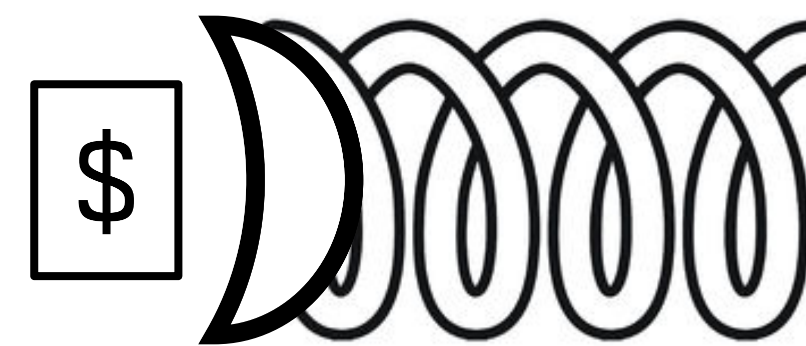



*Input*

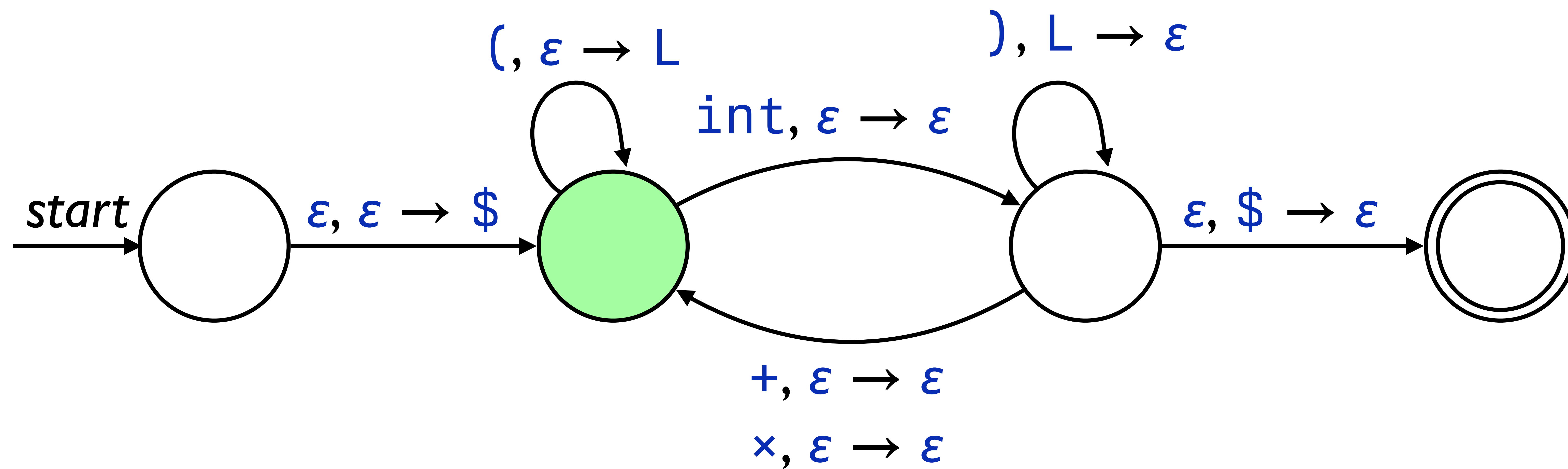
int + ( ( int × int ) + int )



*Stack*



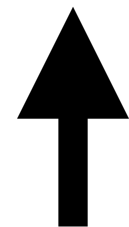
# A PDA for arithmetic

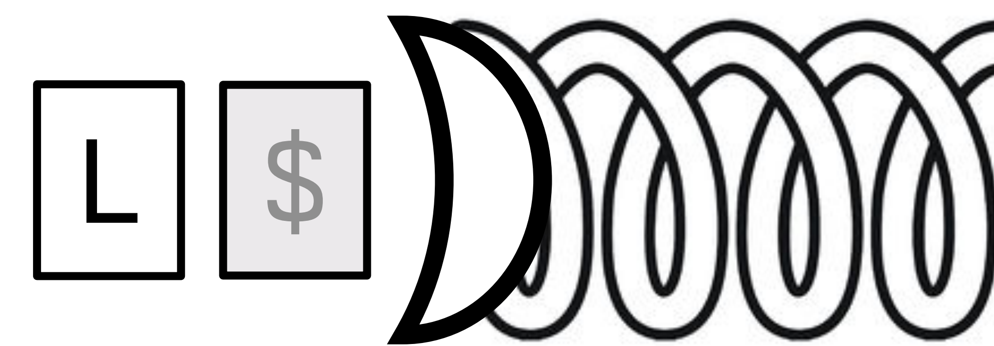



*Input*

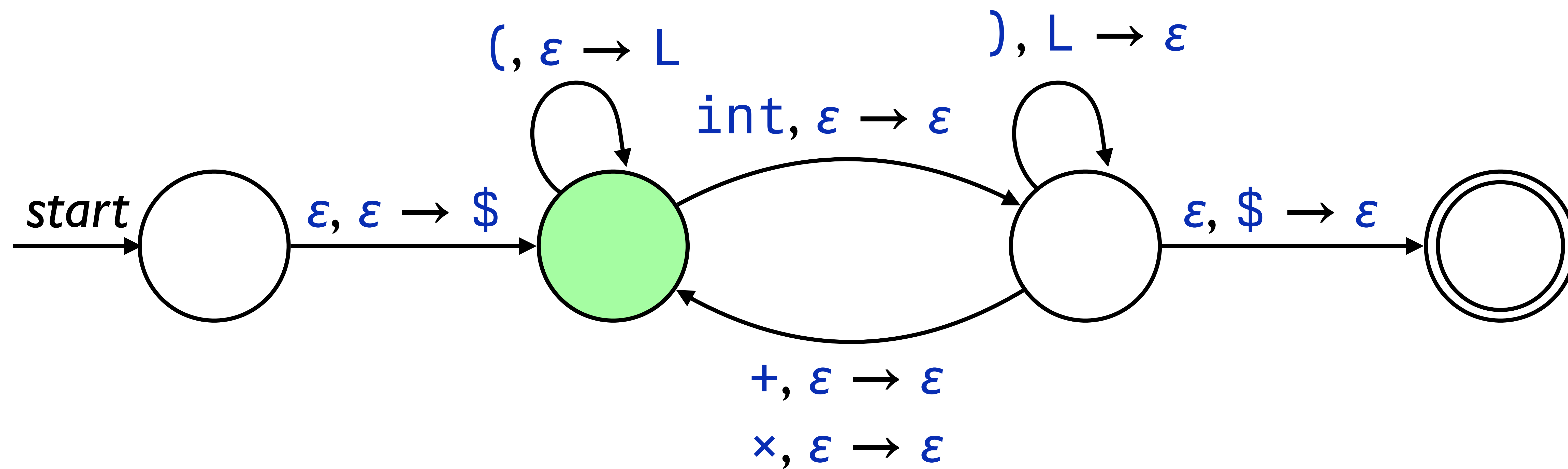
*Stack*

int + ( ( int × int ) + int )



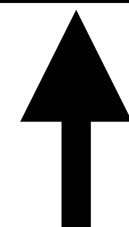


# A PDA for arithmetic

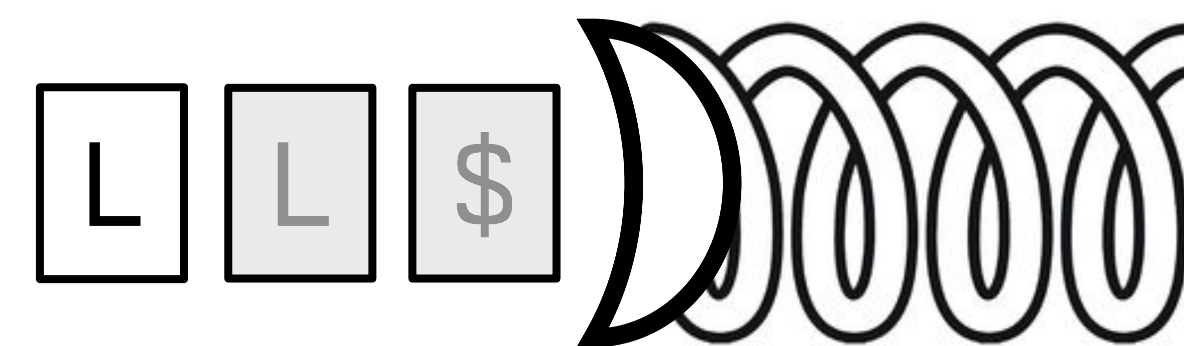



*Input*

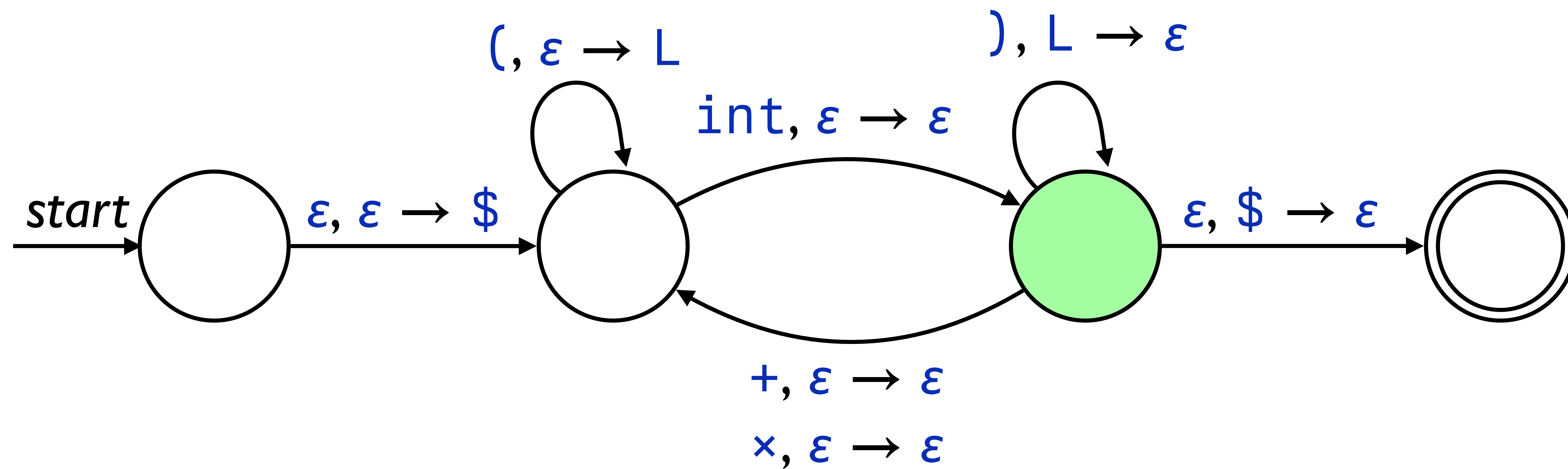
int + ( ( int × int ) + int )



*Stack*

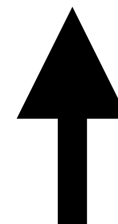


# A PDA for arithmetic

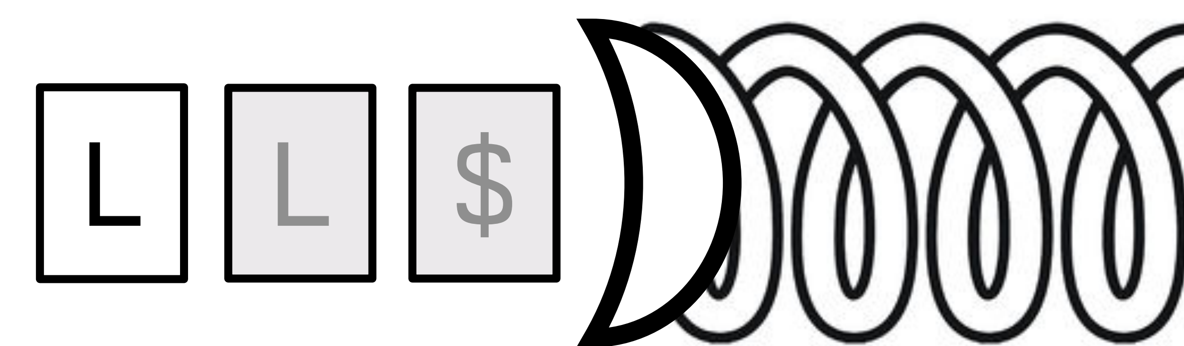



*Input*

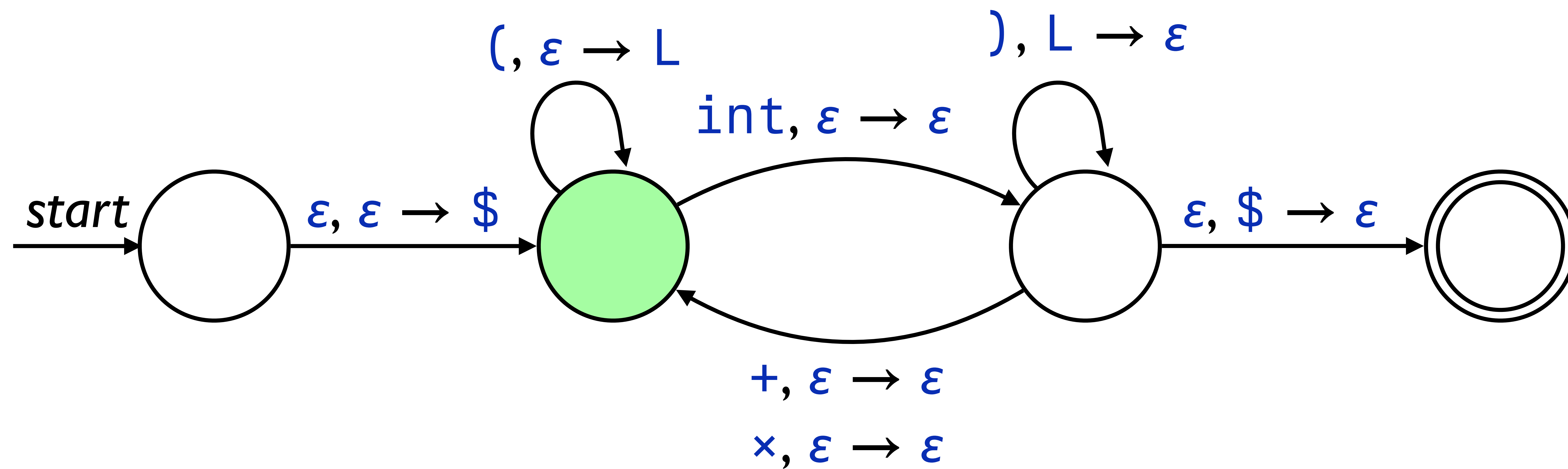
int + ( ( int × int ) + int )



*Stack*

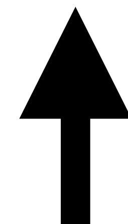


# A PDA for arithmetic

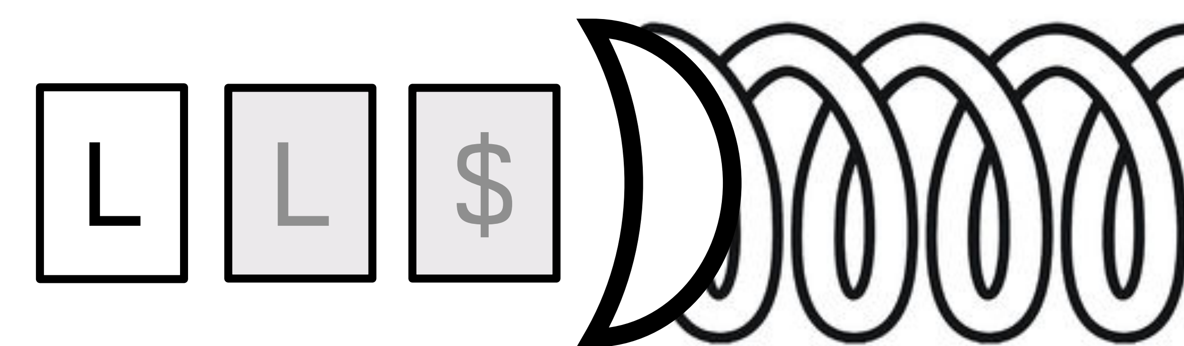



*Input*

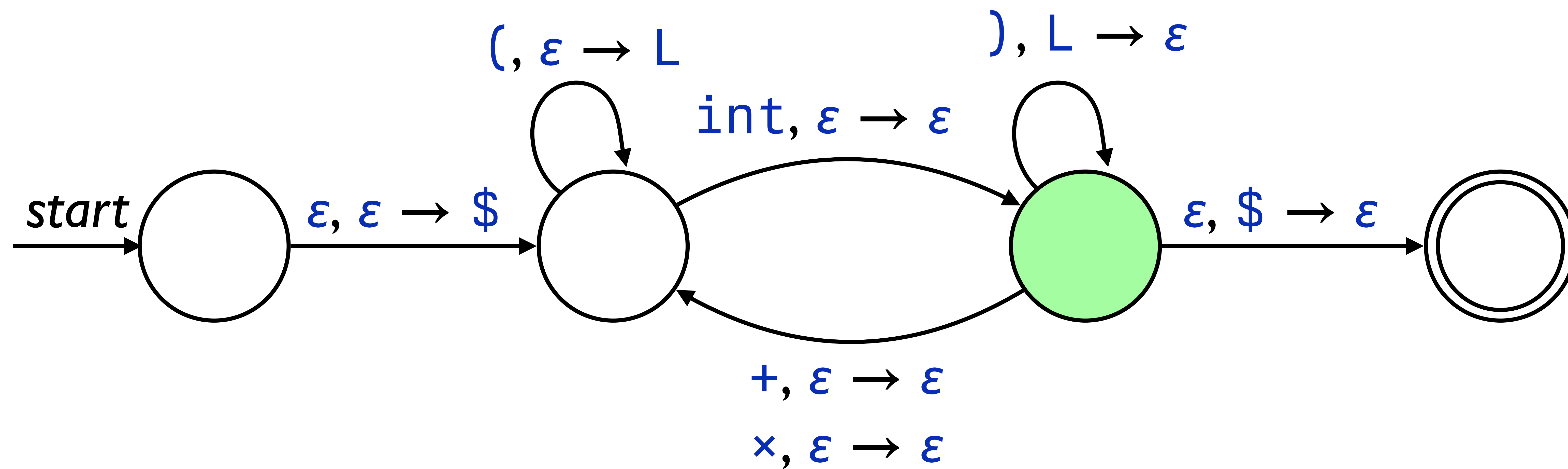
`int + ( ( int × int ) + int )`



*Stack*



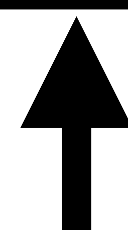
# A PDA for arithmetic

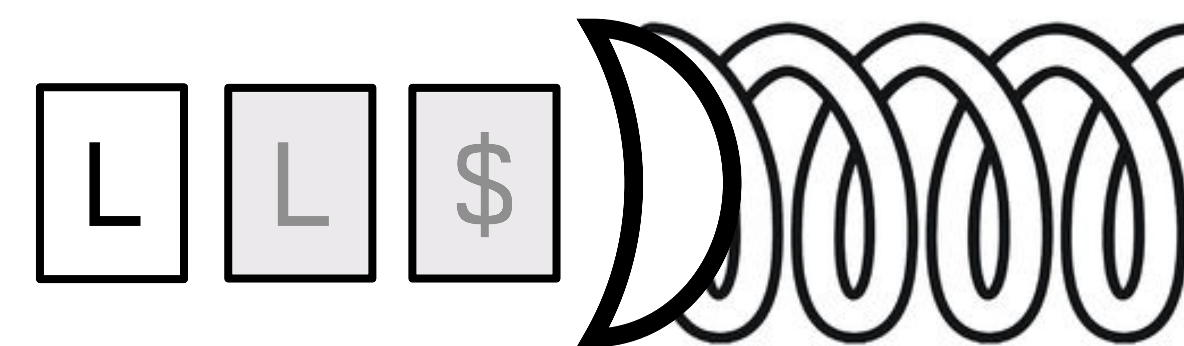



*Input*

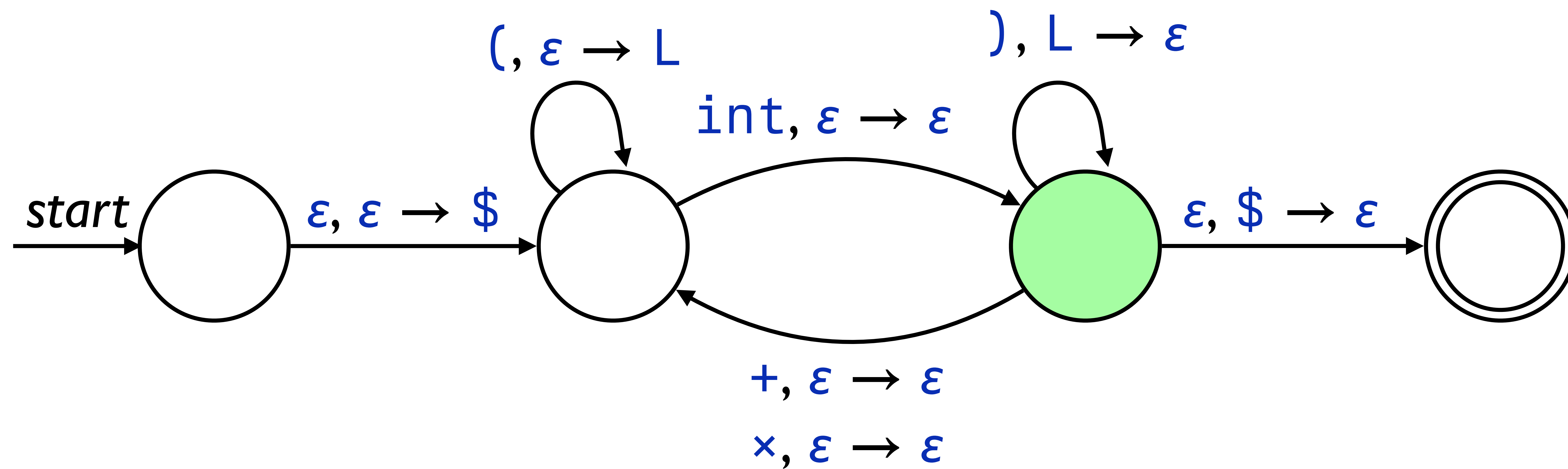
*Stack*

int + ( ( int × int ) + int )



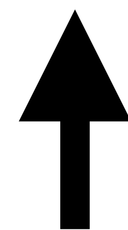


# A PDA for arithmetic

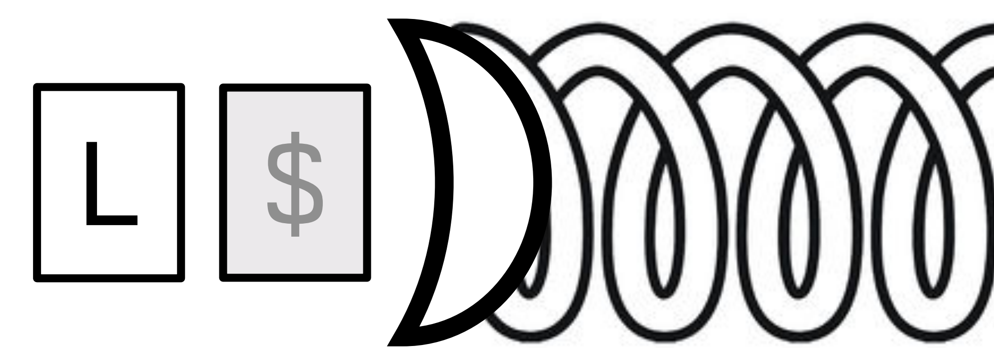



*Input*

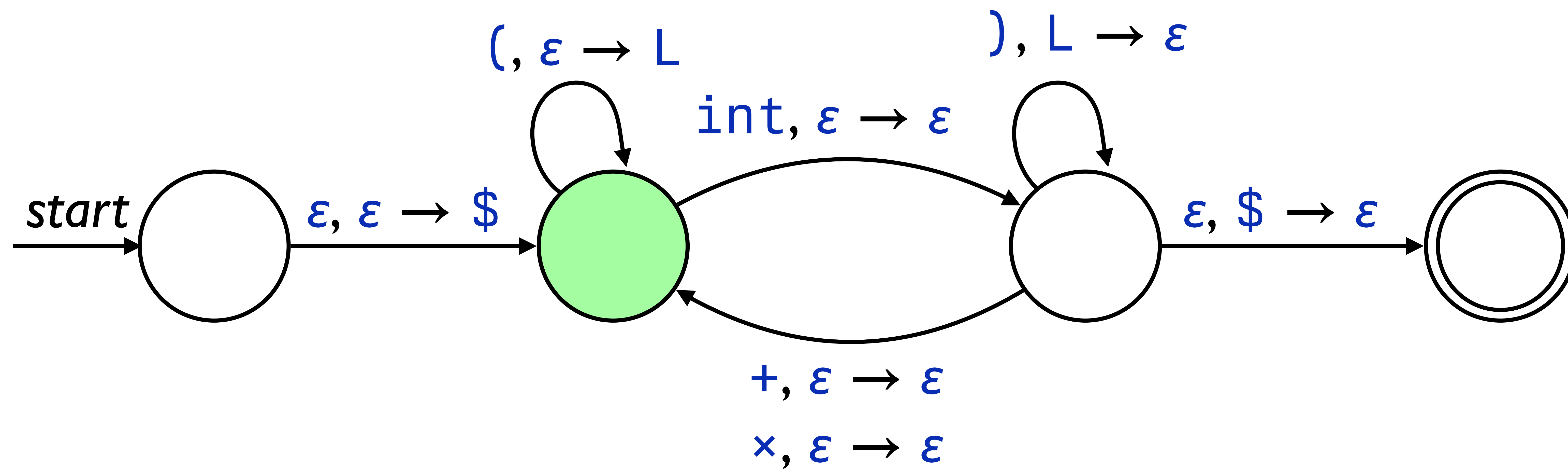
int + ( ( int × int ) + int )



*Stack*

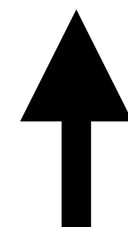


# A PDA for arithmetic

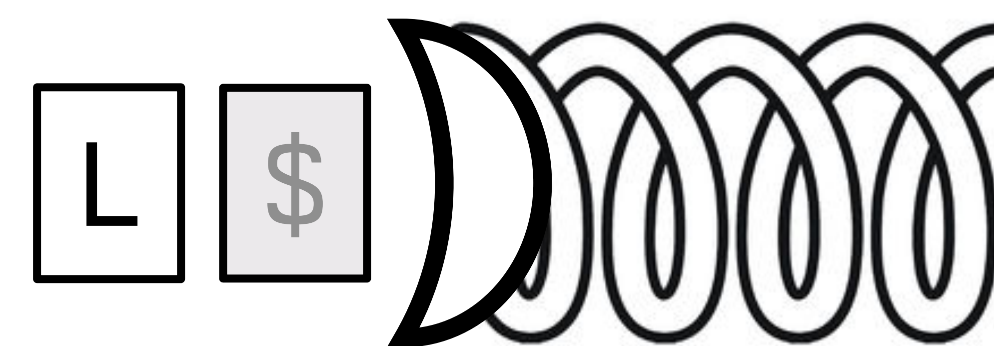



*Input*

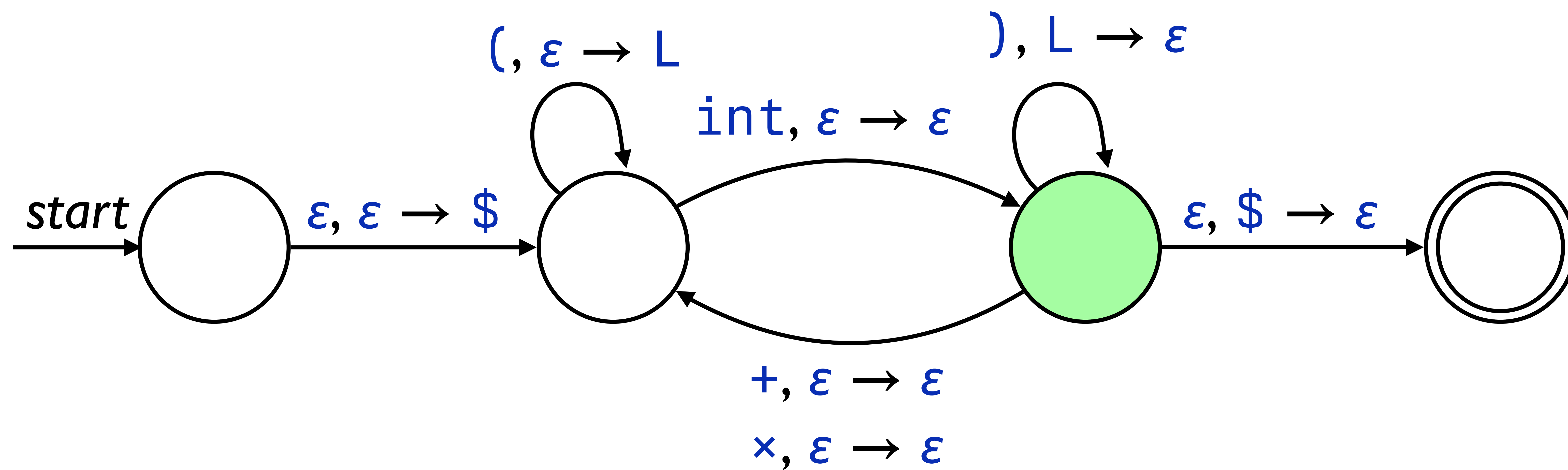
int + ( ( int × int ) + int )



*Stack*

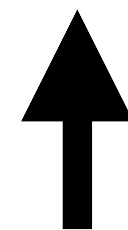


# A PDA for arithmetic

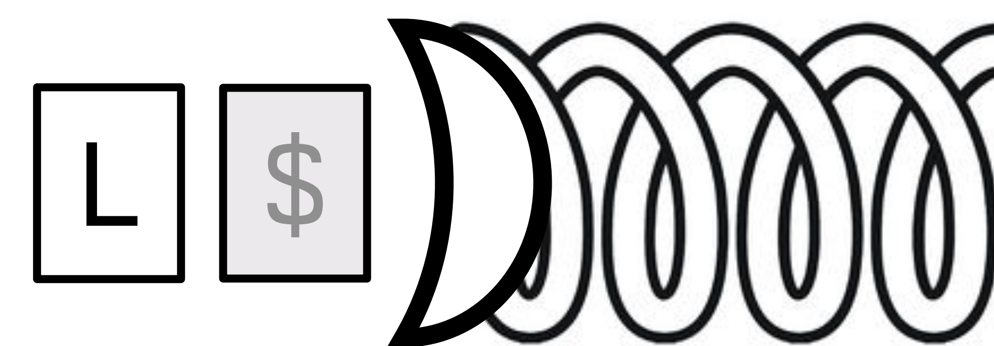



*Input*

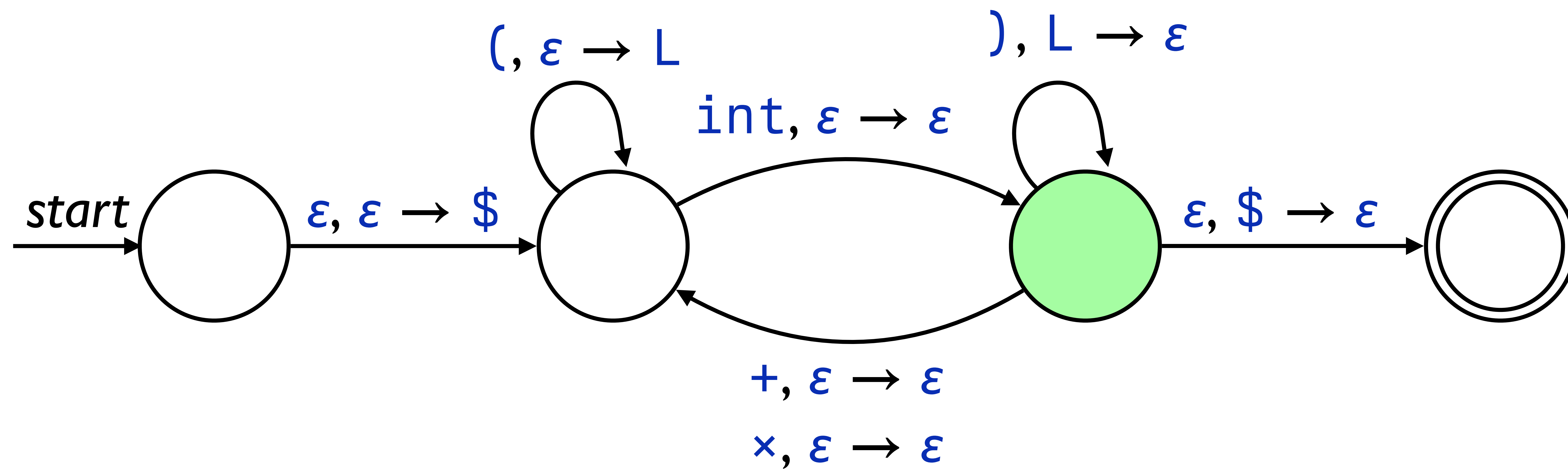
int + ( ( int × int ) + int )



*Stack*



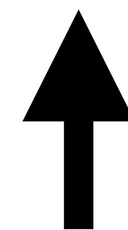
# A PDA for arithmetic

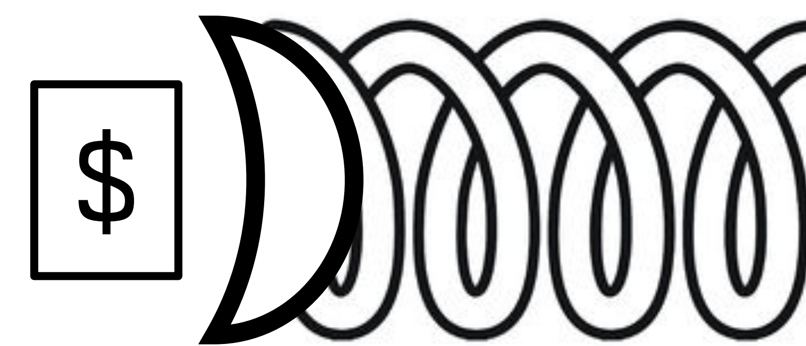



*Input*

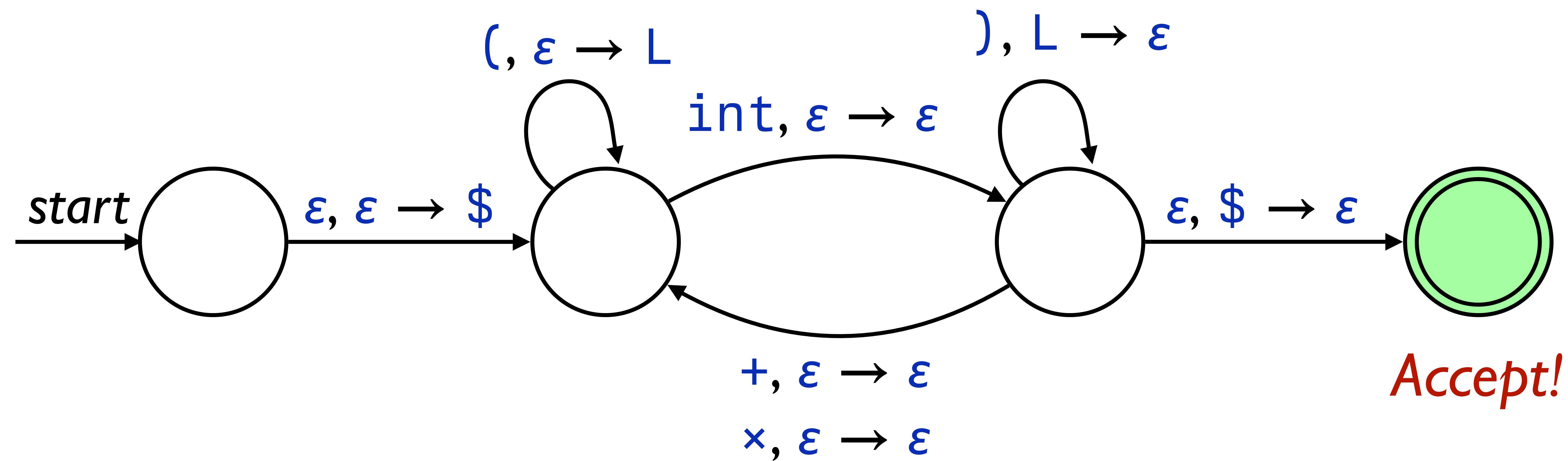
*Stack*

int + ( ( int × int ) + int )





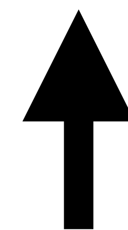
# A PDA for arithmetic

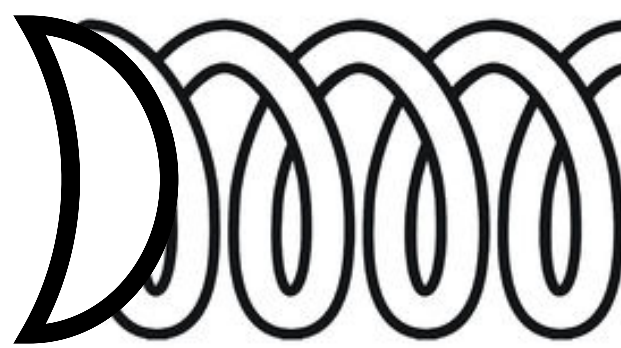



*Input*

*Stack*

`int + ( ( int × int ) + int )`





# Shorthand for pushing multiple symbols to the stack

$(r, \text{xyz}) \in \delta(q, \text{a}, \text{s})$

$q$ is the current state

a is the next input symbol

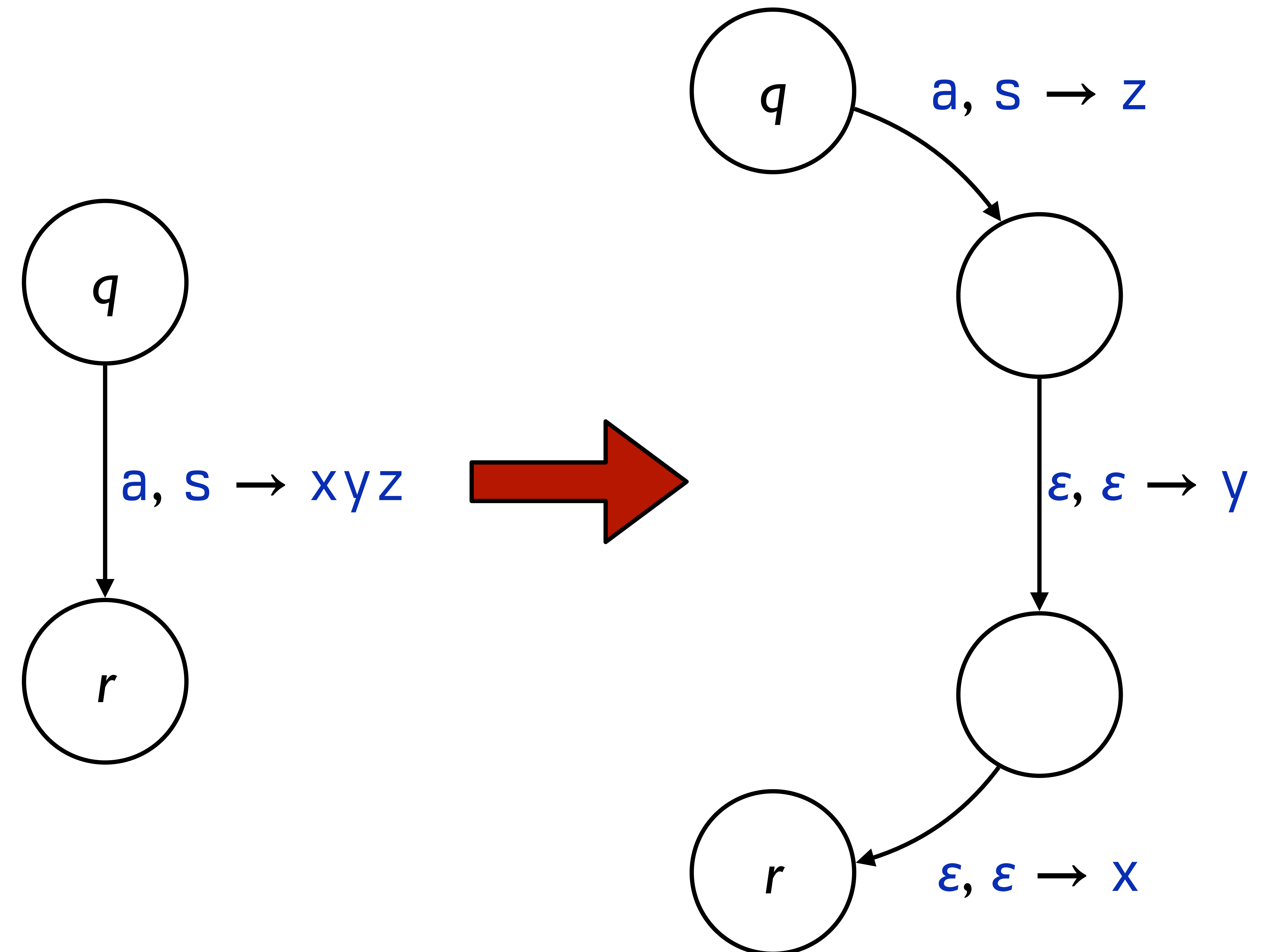
s is on the top of the stack

## Do the following:

Read a

Pop s

Push xyz



# Acknowledgments

This lecture incorporates material from:

- Nancy Ide, Vassar College
- Keith Schwarz, Stanford University
- Michael Sipser, *Introduction to the Theory of Computation*
- Jeffrey Ullman, Stanford University
- Jennifer Walter, Vassar College