Pushdown Automata

Friday Four Square!
Today at 4:15PM, Outside Gates

Announcements

- Problem Set 5 due right now
  - Or Monday at 2:15PM with a late day.
- Problem Set 6 out, due next **Friday, November 9**.
  - Covers context-free languages, CFGs, and PDAs.
- Midterm and Problem Set 4 should be graded by Monday.

Generation vs. Recognition

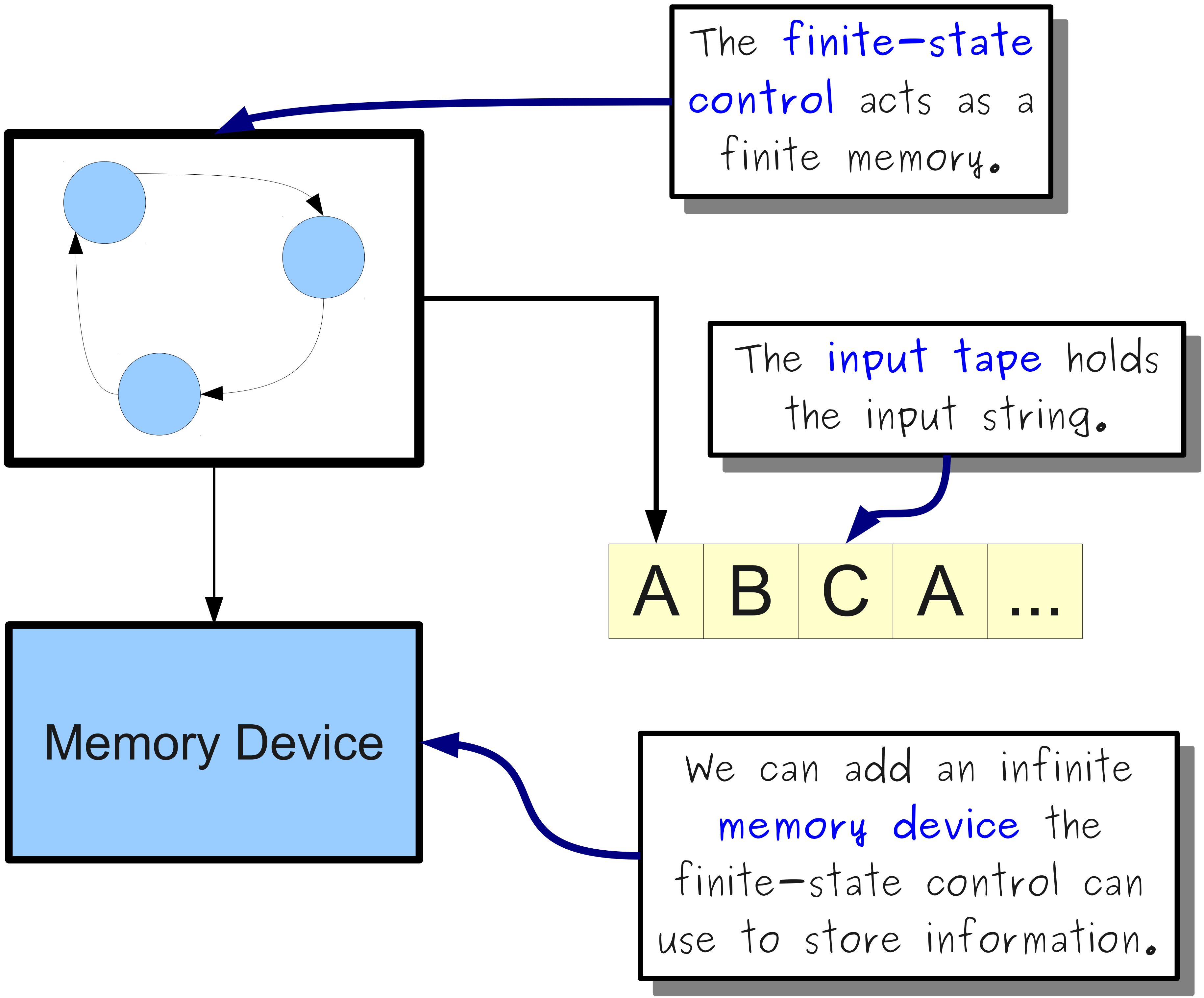
- We saw two approaches to describe regular languages:
  - Build **automata** that accept precisely the strings in the language.
  - Design **regular expressions** that describe precisely the strings in the language.
- Regular expressions **generate** all of the strings in the language.
  - Useful for listing off all strings in the language.
- Finite automata **recognize** all of the strings in the language.
  - Useful for detecting whether a specific string is in the language.

Context-Free Languages

- Yesterday, we saw the **context-free languages**, which are those that can be generated by **context-free grammars**.
- Is there some way to build an automaton that can **recognize** the context-free languages?

The Problem

- Finite automata accept precisely the regular languages.
- We may need unbounded memory to recognize context-free languages.
    - e.g. $\{ \mathtt{0}^n\mathtt{1}^n \mid n \in \mathbb{N} \}$ requires unbounded counting.
- How do we build an automaton with finitely many states but unbounded memory?


The finite-state control acts as a finite memory.
The input tape holds the input string.
A
B
C
A
...
Memory Device
We can add an infinite memory device the finite-state control can use to store information.

Adding Memory to Automata

- We can augment a finite automaton by adding in a **memory device** for the automaton to store extra information.
- The finite automaton now can base its transition on both the current symbol being read and values stored in memory.
- The finite automaton can issue commands to the memory device whenever it makes a transition.
  - e.g. add new data, change existing data, etc.

Stack-Based Memory

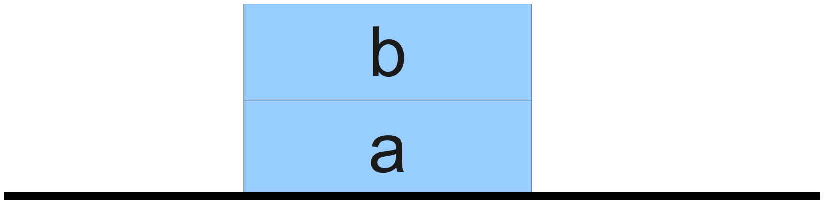
- There are **many** types of memory that we might give to an automaton.
  - We'll see at least two this quarter.
- One of the simplest types of memory is a **stack**.

Stack-Based Memory

- There are **many** types of memory that we might give to an automaton.
  - We'll see at least two this quarter.
- One of the simplest types of memory is a **stack**.

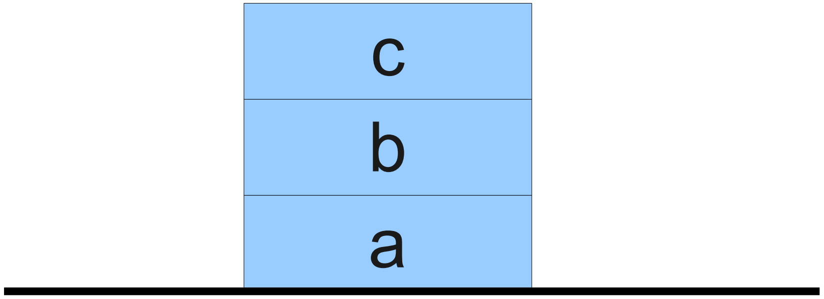
Stack-Based Memory

- There are **many** types of memory that we might give to an automaton.
  - We'll see at least two this quarter.
- One of the simplest types of memory is a **stack**.



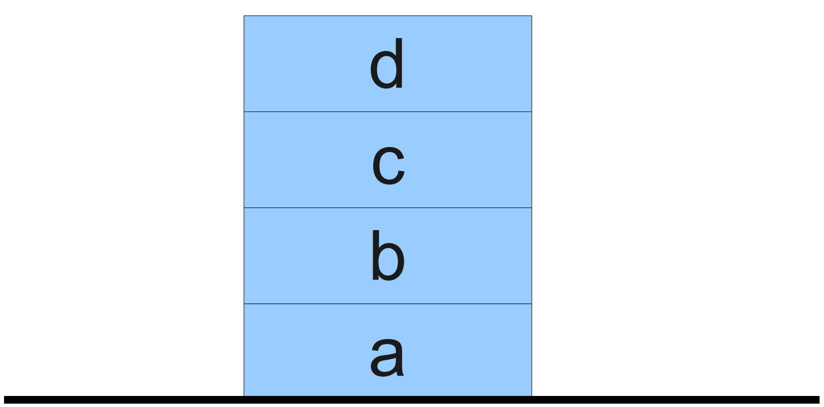
Stack-Based Memory

- There are **many** types of memory that we might give to an automaton.
  - We'll see at least two this quarter.
- One of the simplest types of memory is a **stack**.



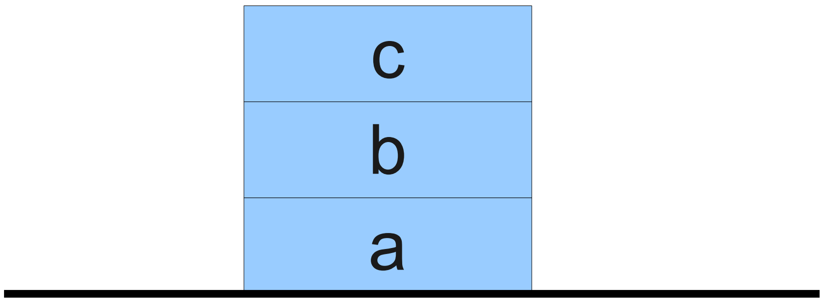
Stack-Based Memory

- There are **many** types of memory that we might give to an automaton.
  - We'll see at least two this quarter.
- One of the simplest types of memory is a **stack**.



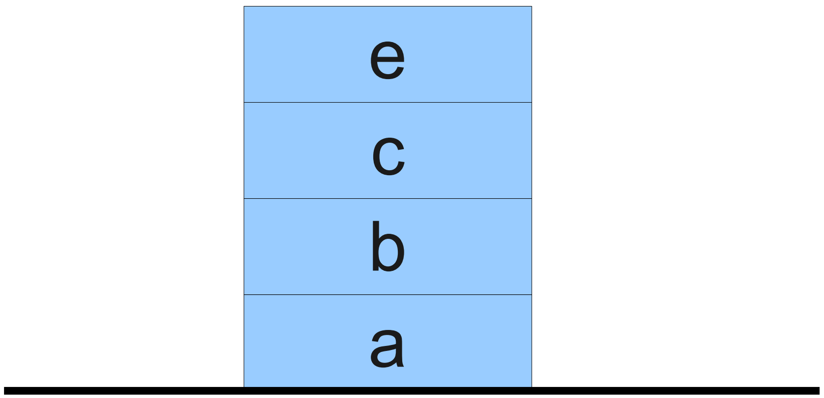
Stack-Based Memory

- There are **many** types of memory that we might give to an automaton.
  - We'll see at least two this quarter.
- One of the simplest types of memory is a **stack**.



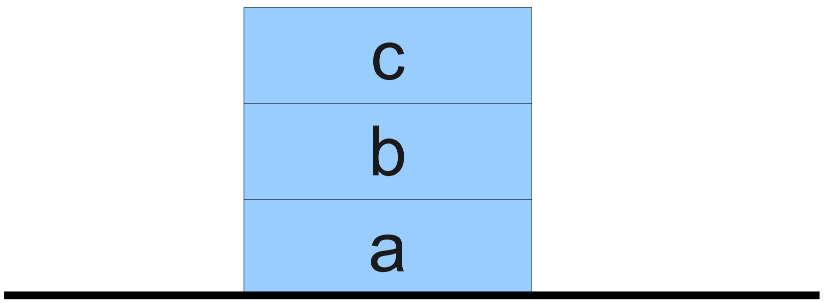
Stack-Based Memory

- There are **many** types of memory that we might give to an automaton.
  - We'll see at least two this quarter.
- One of the simplest types of memory is a **stack**.



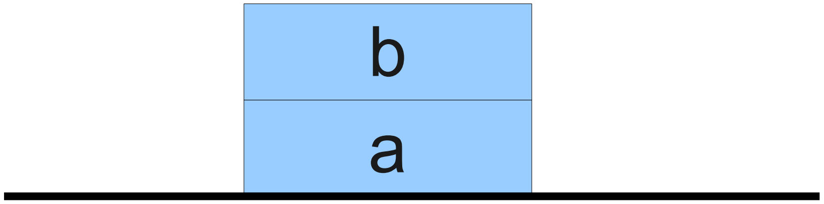
Stack-Based Memory

- There are **many** types of memory that we might give to an automaton.
  - We'll see at least two this quarter.
- One of the simplest types of memory is a **stack**.



Stack-Based Memory

- There are **many** types of memory that we might give to an automaton.
  - We'll see at least two this quarter.
- One of the simplest types of memory is a **stack**.



Stack-Based Memory

- Only the top of the stack is visible at any point in time.
- New symbols may be **pushed** onto the stack, which cover up the old stack top.
- The top symbol of the stack may be **popped**, exposing the symbol below it.

Pushdown Automata

- A **pushdown automaton** (PDA) is a finite automaton equipped with a stack-based memory.
- Each transition
  - is based on the current input symbol and the top of the stack,
  - optionally pops the top of the stack, and
  - optionally pushes new symbols onto the stack.
- Initially, the stack holds a special symbol $\mathbf{Z}_0$ that indicates the bottom of the stack.

Our First PDA

- Consider the language

$$L = \{ w \in \Sigma^* \mid w \text{ is a string of balanced parentheses} \}$$

over $\Sigma = \{$ (,) $\}$

- We can exploit the stack to our advantage:
  - Whenever we see a (, push it onto the stack.
  - Whenever we see a), pop the corresponding (from the stack (or fail if not matched)
  - When input is consumed, if the stack is empty, accept.

Our First PDA

- Consider the language

  $$L = \{ w \in \Sigma^* \mid w \text{ is a string of balanced parentheses} \}$$

  over $\Sigma = \{ (,) \}$

- We can exploit the stack to our advantage:
  - Whenever we see a (, push it onto the stack.
  - Whenever we see a), pop the corresponding (from the stack (or fail if not matched)
  - When input is consumed, if the stack is empty, accept.

Our First PDA

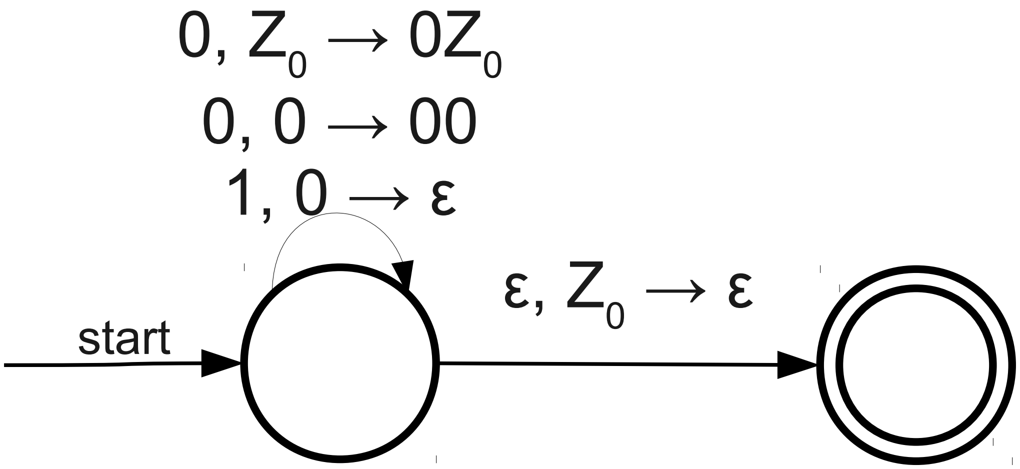
- Consider the language

  $L = \{ w \in \Sigma^* \mid w$ is a string of balanced digits $\}$

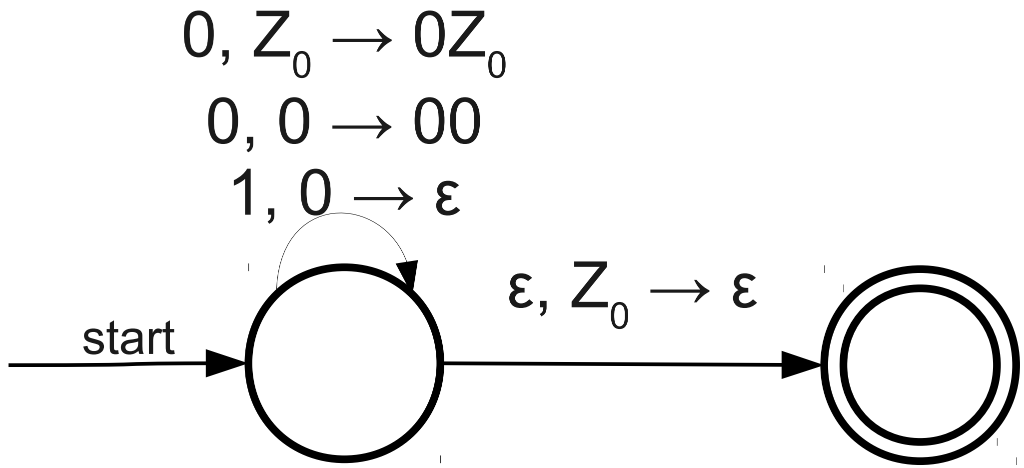
  over $\Sigma = \{$ `0`, `1` $\}$

- We can exploit the stack to our advantage:
  - Whenever we see a `0`, push it onto the stack.
  - Whenever we see a `1`, pop the corresponding `0` from the stack (or fail if not matched)
  - When input is consumed, if the stack is empty, accept.

A Simple Pushdown Automaton

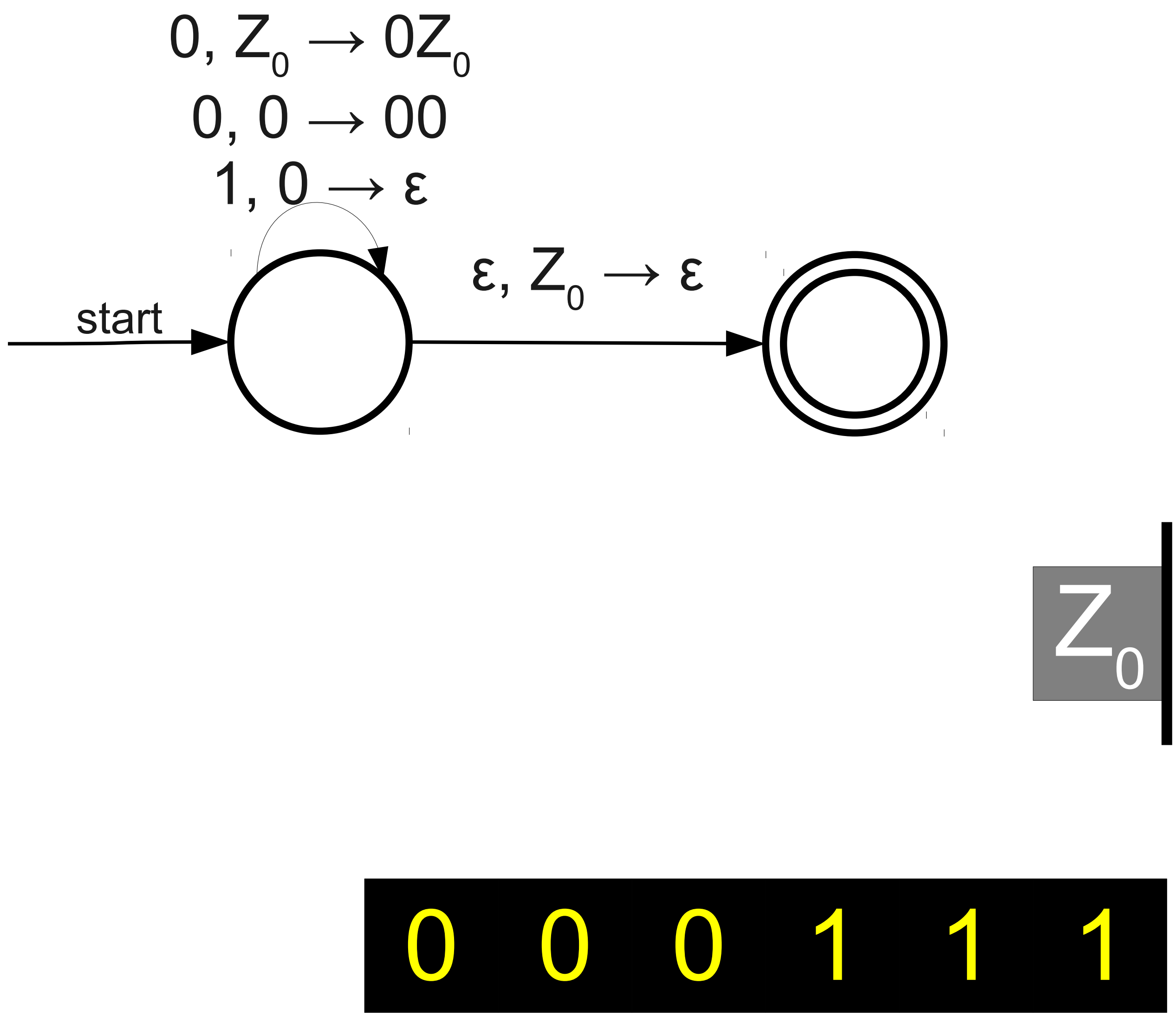


A Simple Pushdown Automaton

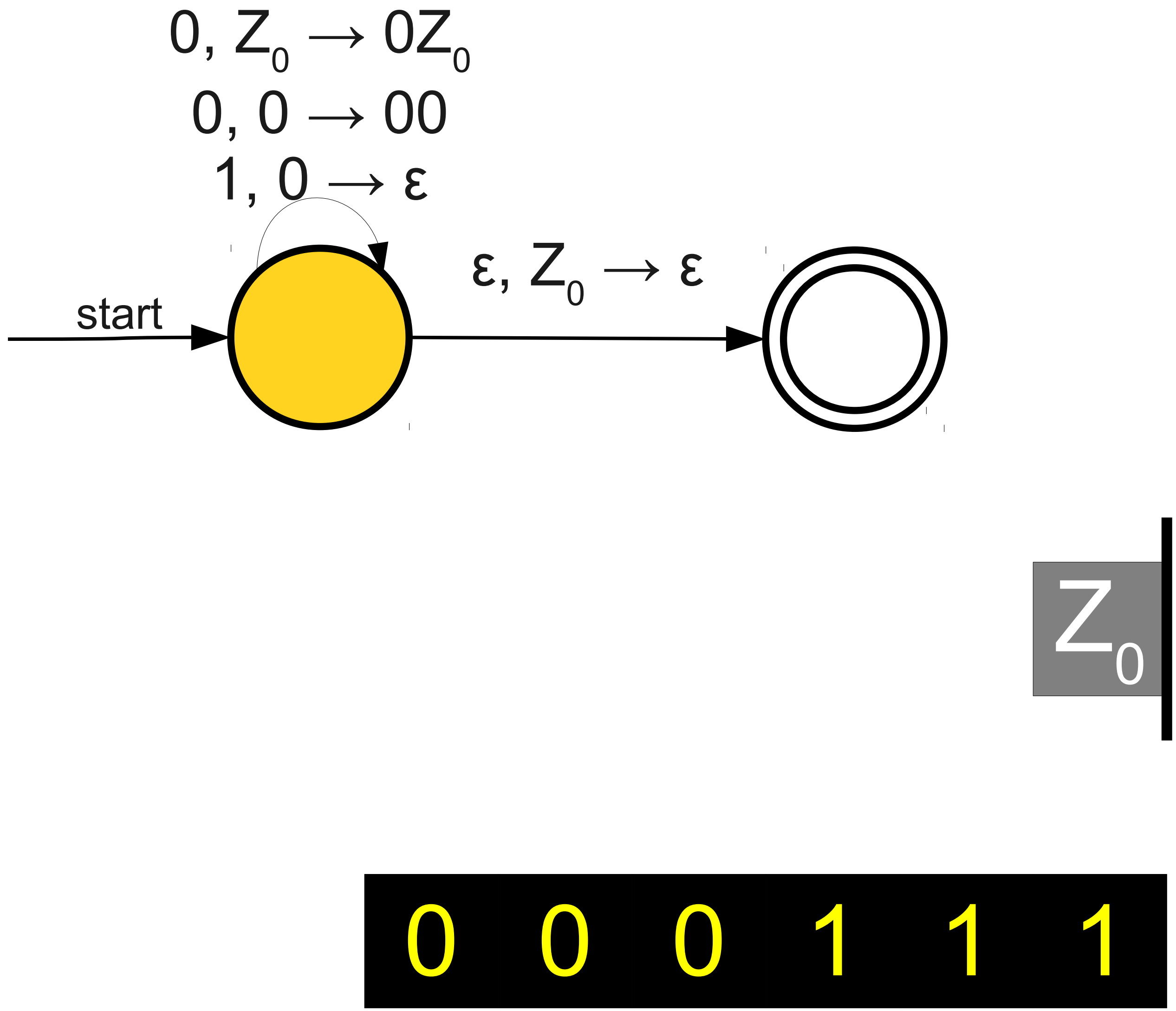


0 0 0 1 1 1

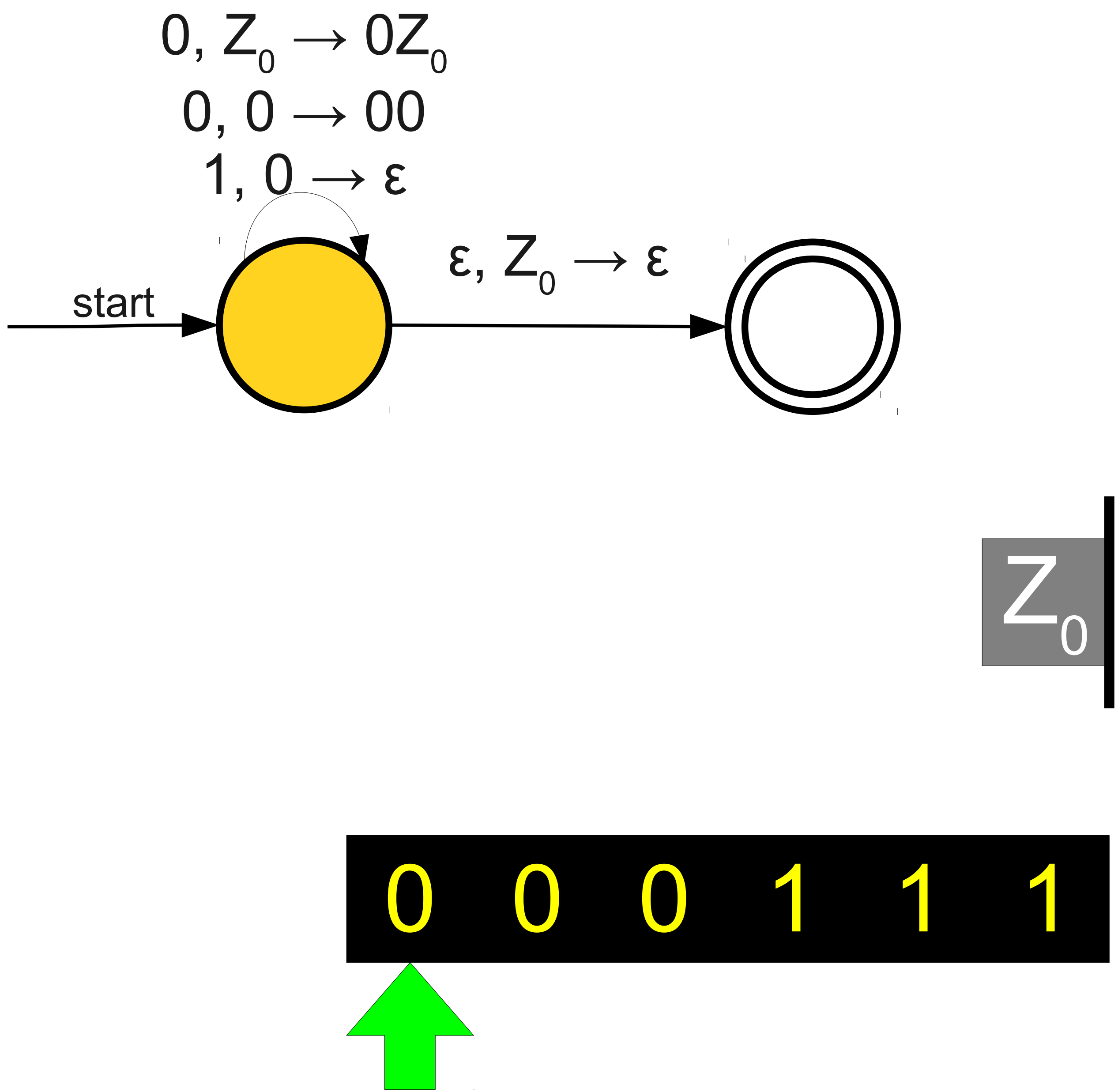
A Simple Pushdown Automaton



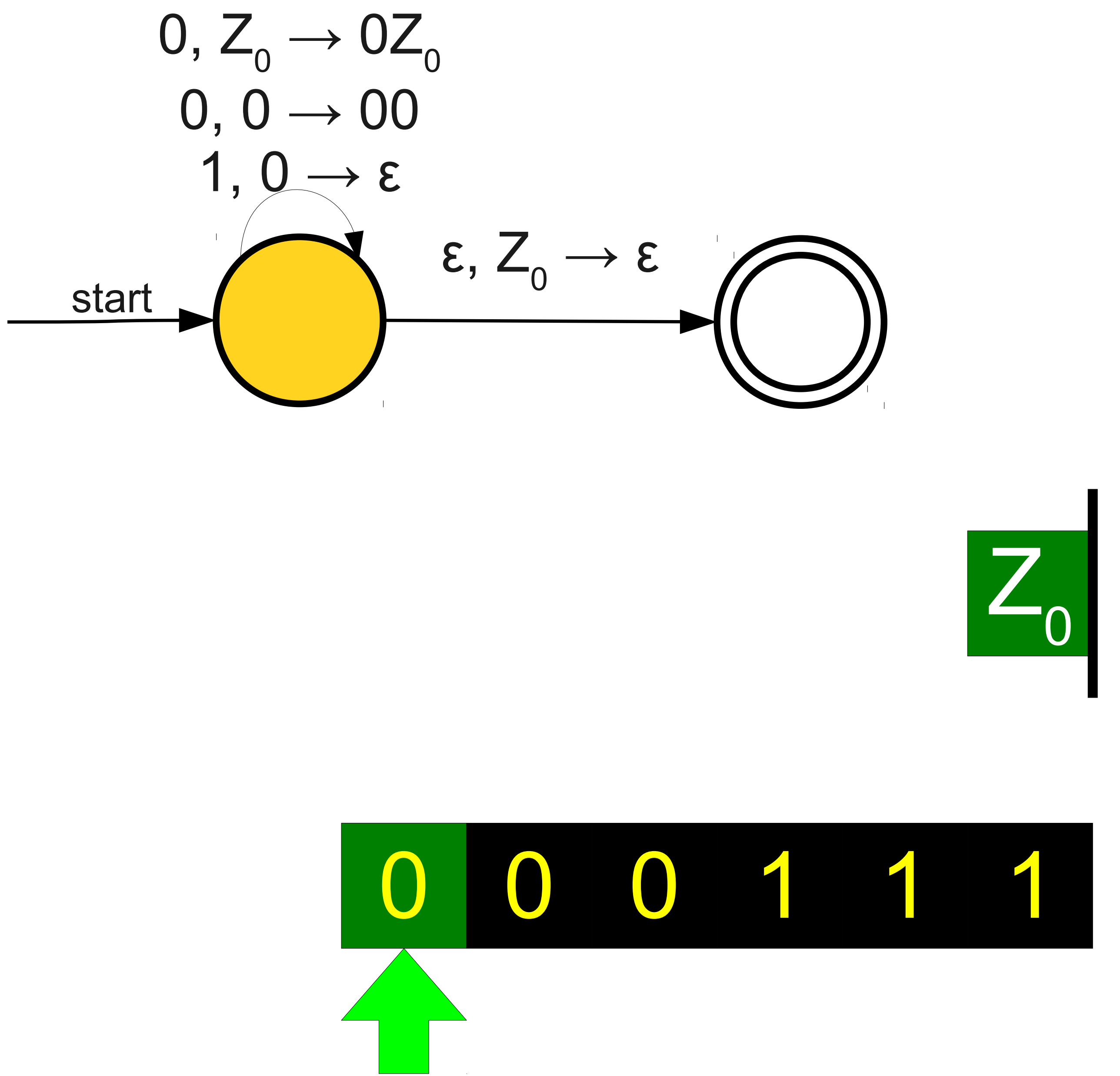
A Simple Pushdown Automaton



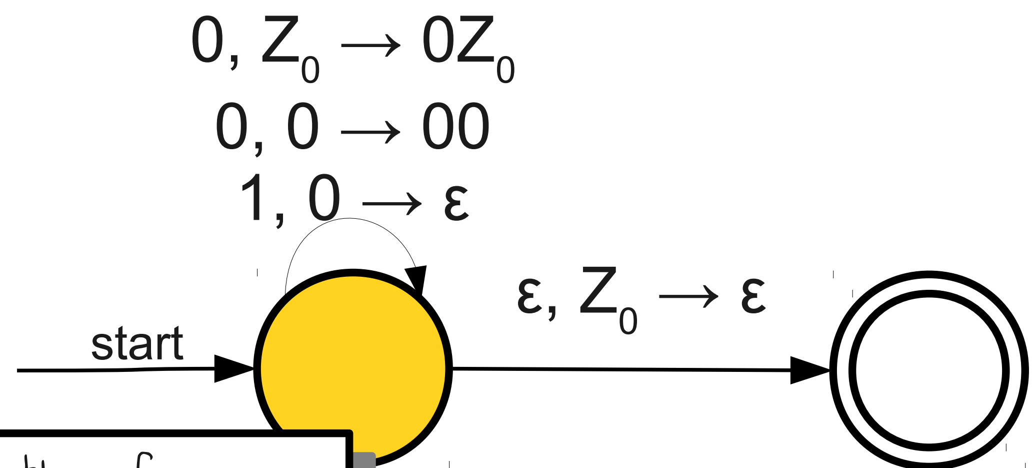
A Simple Pushdown Automaton



A Simple Pushdown Automaton



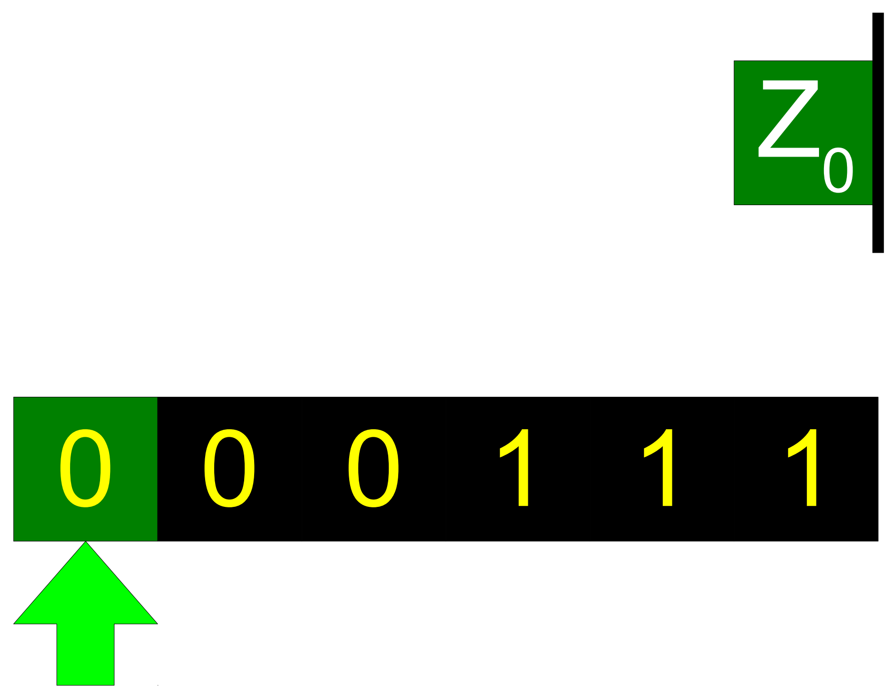
A Simple Pushdown Automaton



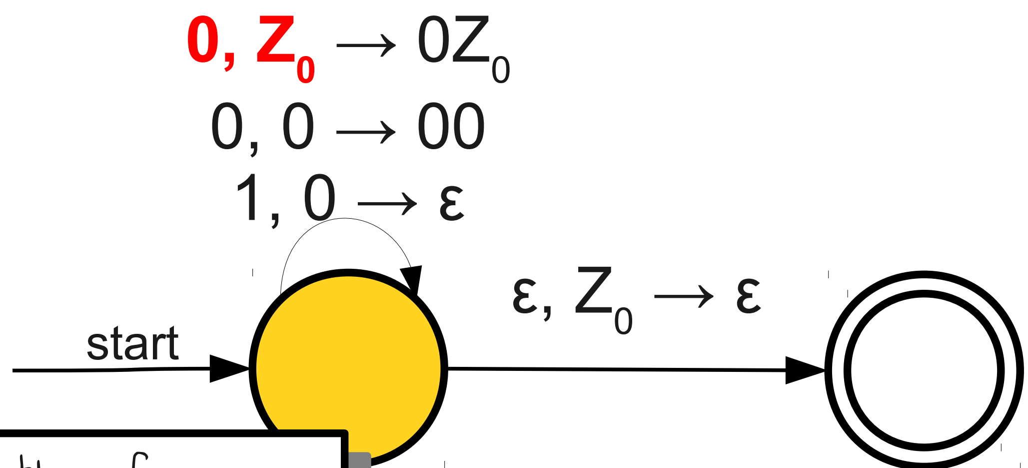
A transition of the form

$$\mathbf{a, b \rightarrow z}$$

Means "If the current input symbol is a and the current stack symbol is b, then follow this transition, pop b, and push the string z.



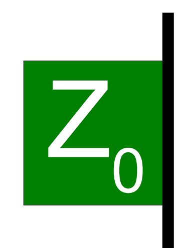
A Simple Pushdown Automaton



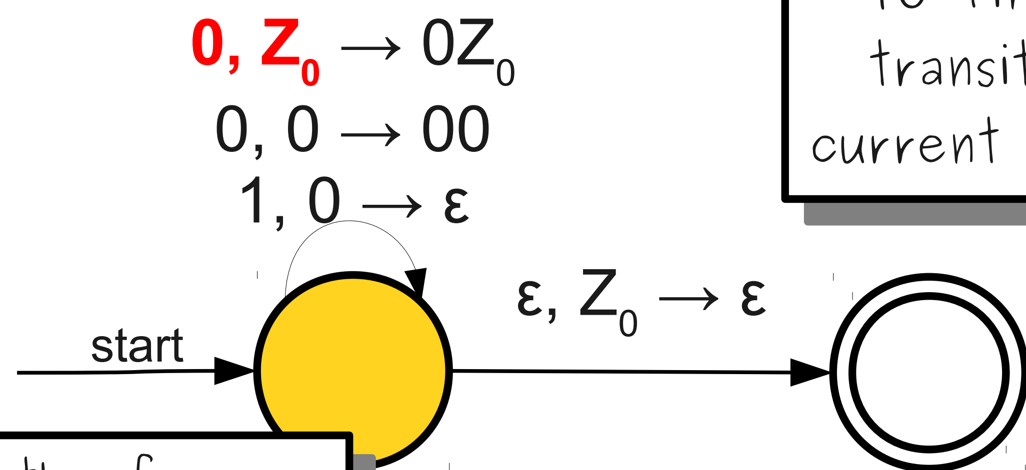
A transition of the form

a, b → z

Means "If the current input symbol is a and the current stack symbol is b, then follow this transition, pop b, and push the string z.



A Simple Pushdown Automaton

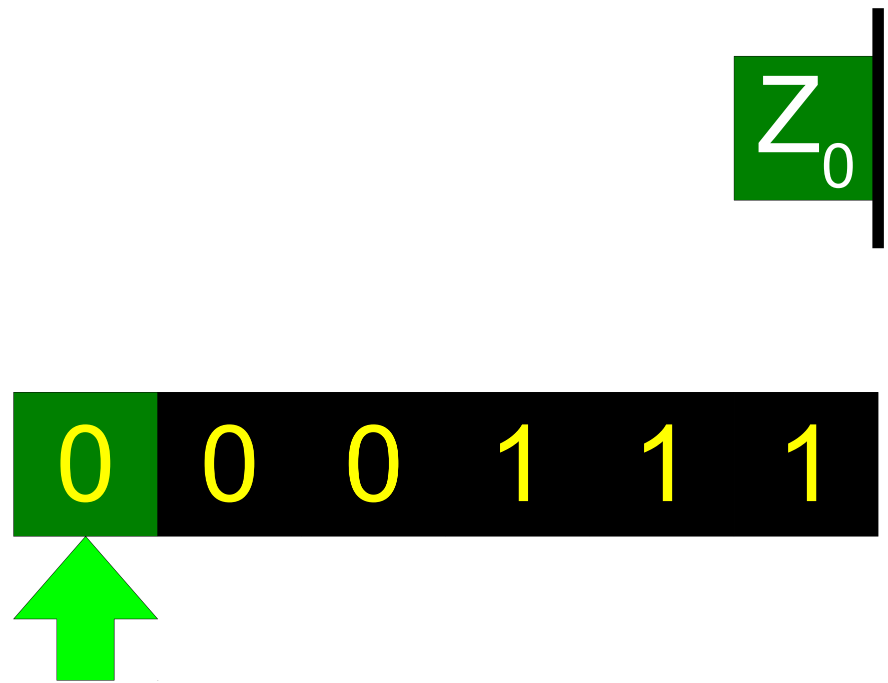


To find an applicable transition, match the current input/stack pair.

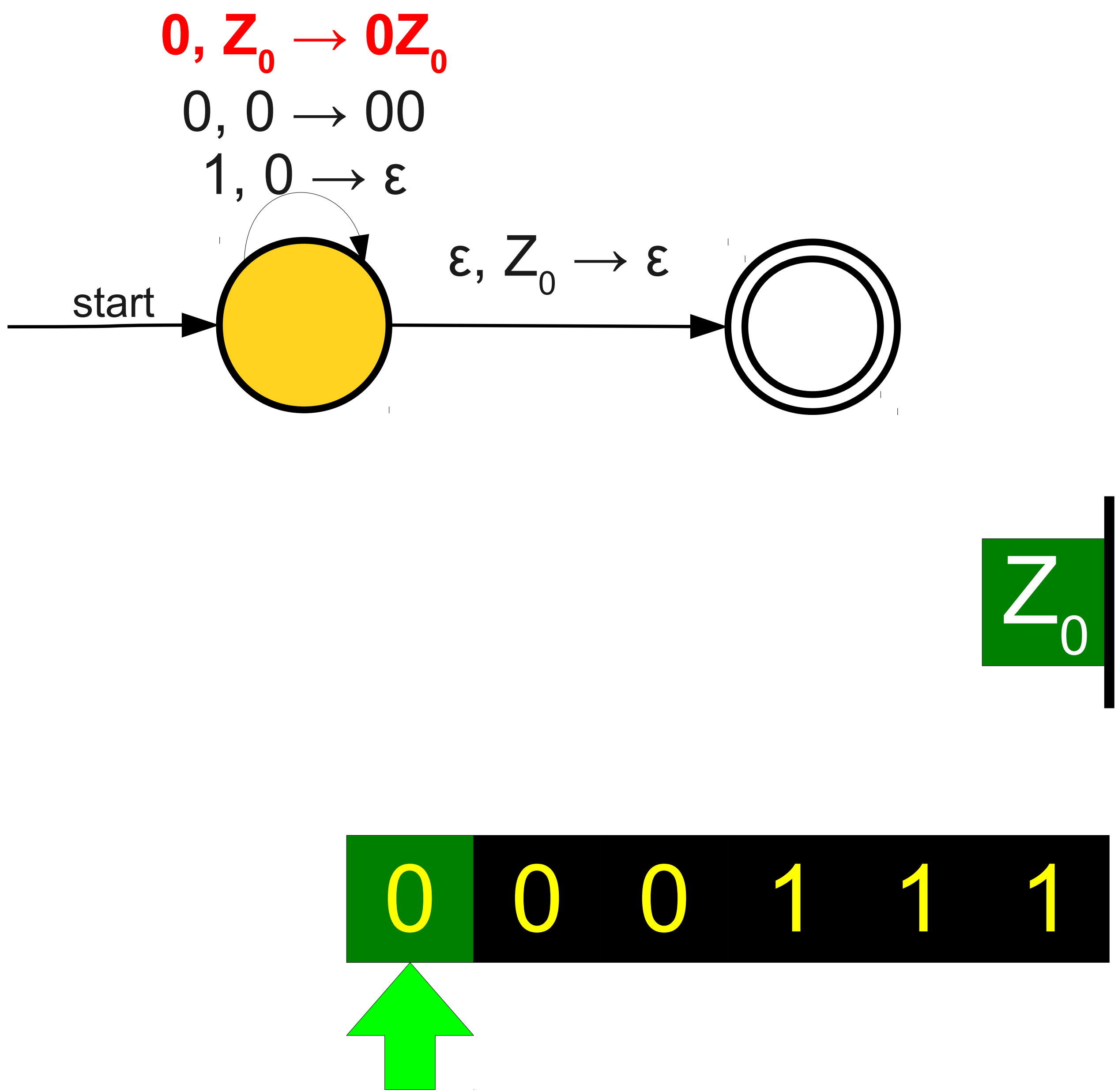
A transition of the form

a, b → z

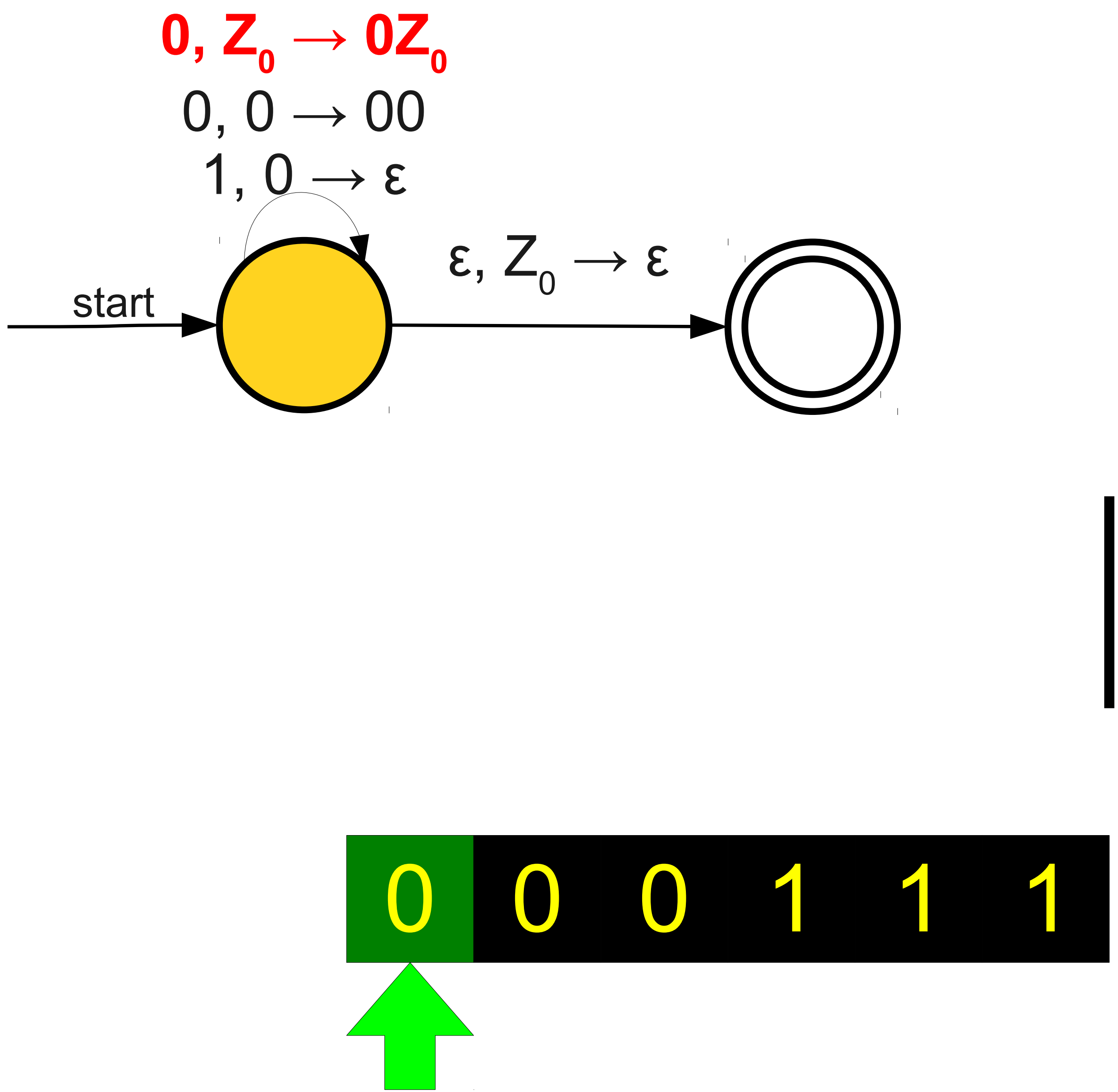
Means "If the current input symbol is a and the current stack symbol is b, then follow this transition, pop b, and push the string z.



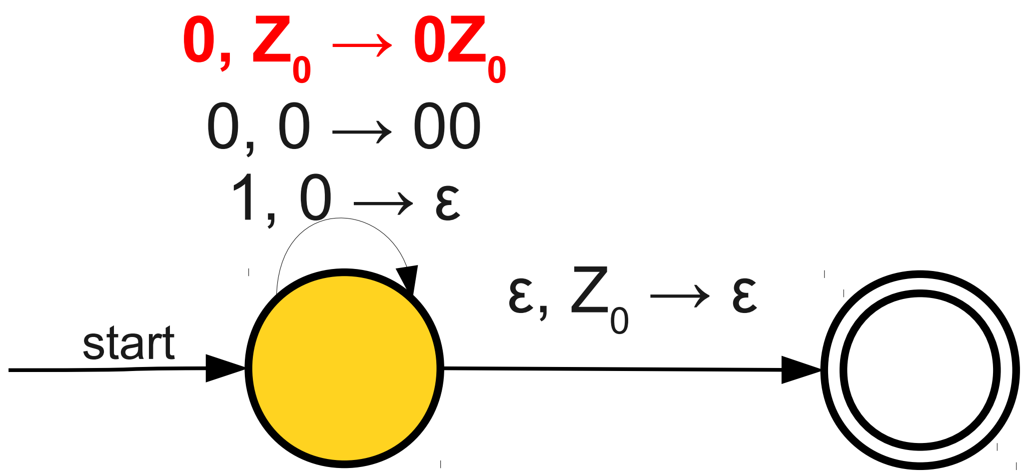
A Simple Pushdown Automaton



A Simple Pushdown Automaton

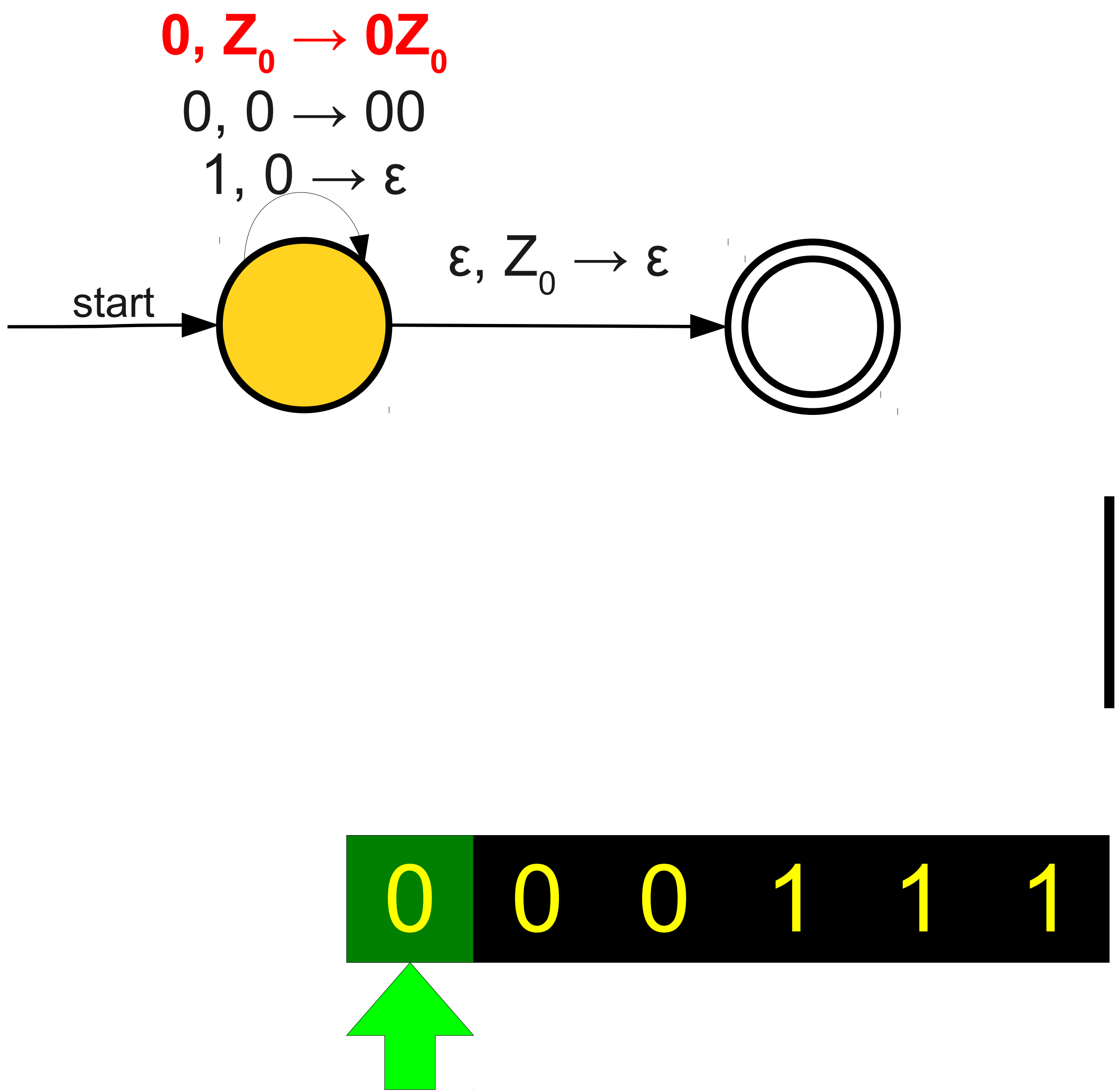


A Simple Pushdown Automaton

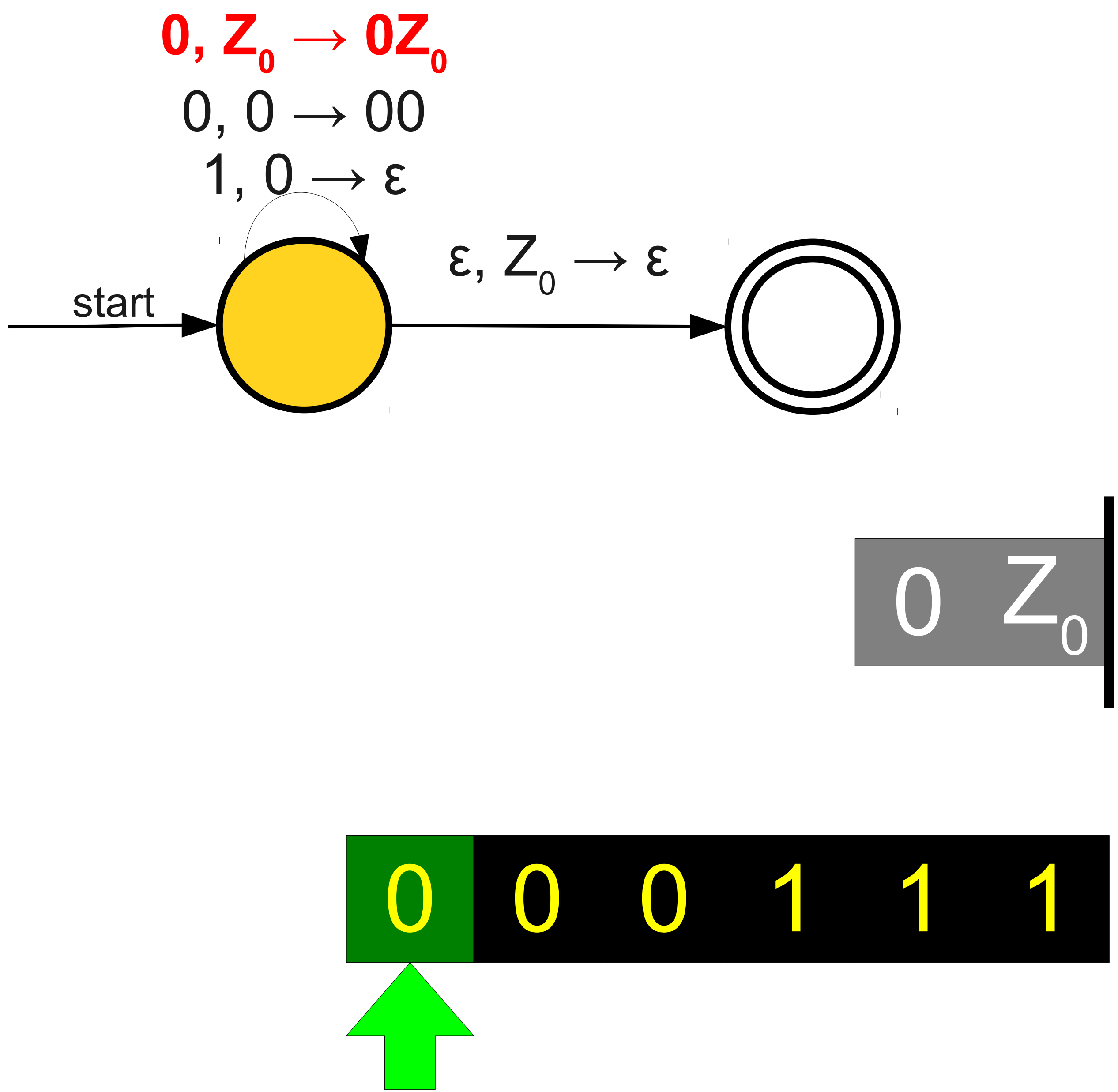


If a transition reads the top symbol of the stack, it always pops that symbol (though it might replace it)

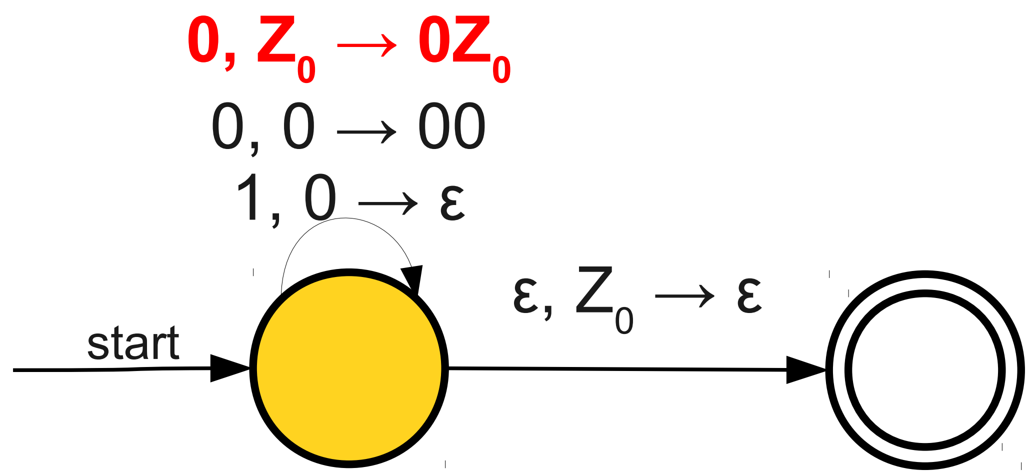
A Simple Pushdown Automaton



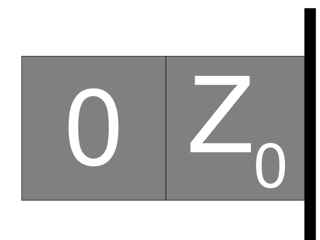
A Simple Pushdown Automaton



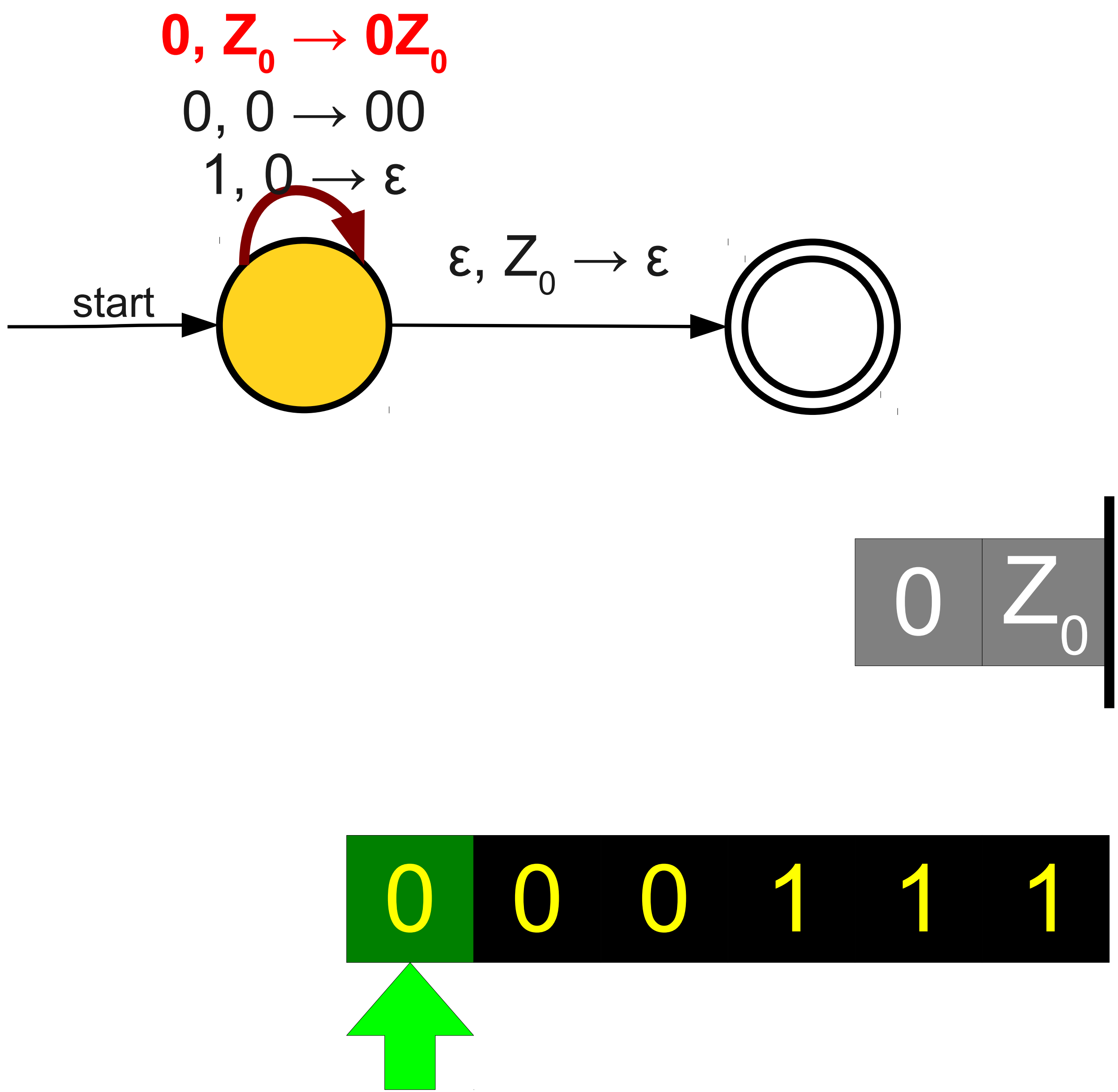
A Simple Pushdown Automaton



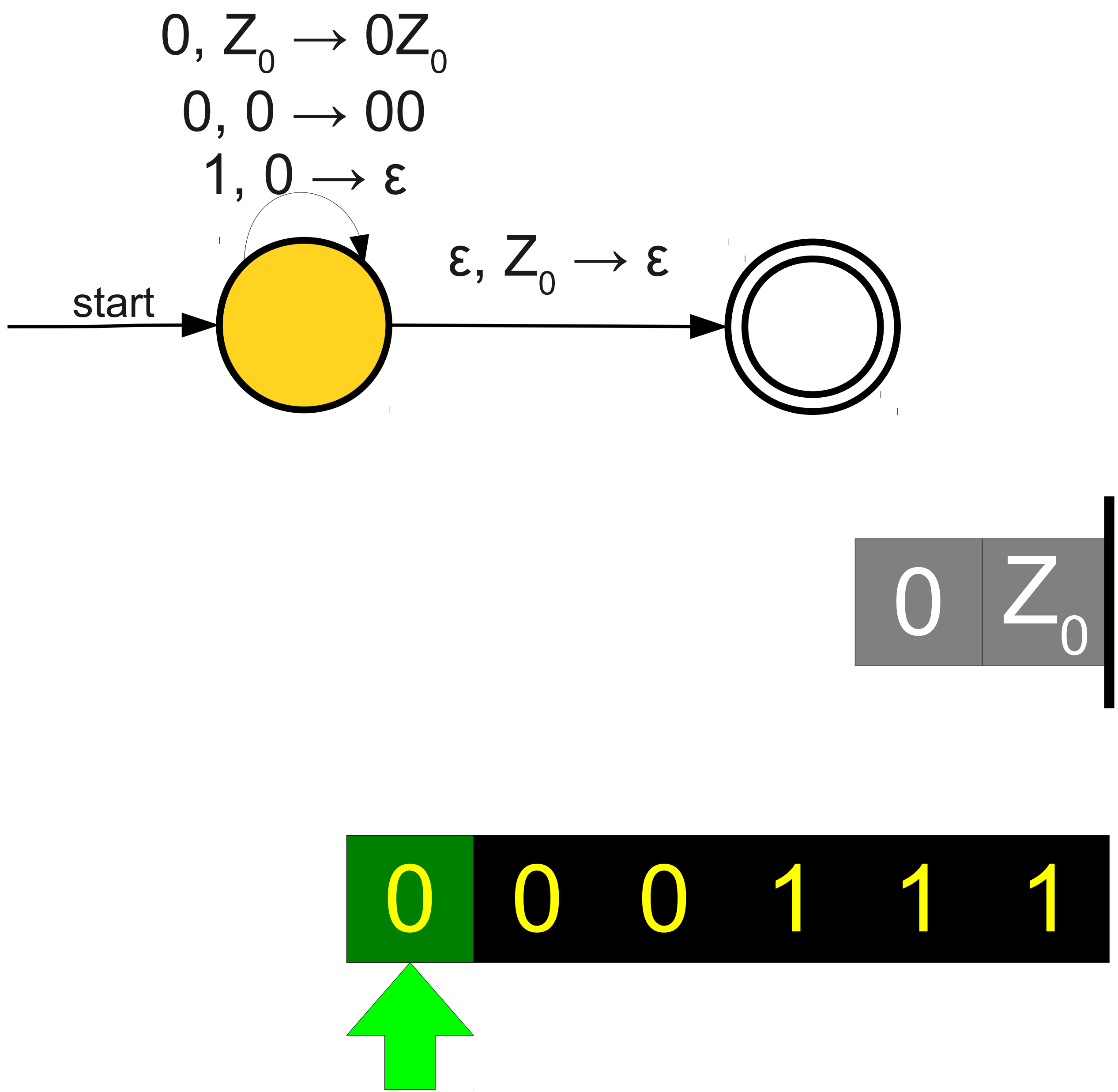
Each transition then pushes some (possibly empty) string back onto the stack. Notice that the leftmost symbol is pushed onto the top.



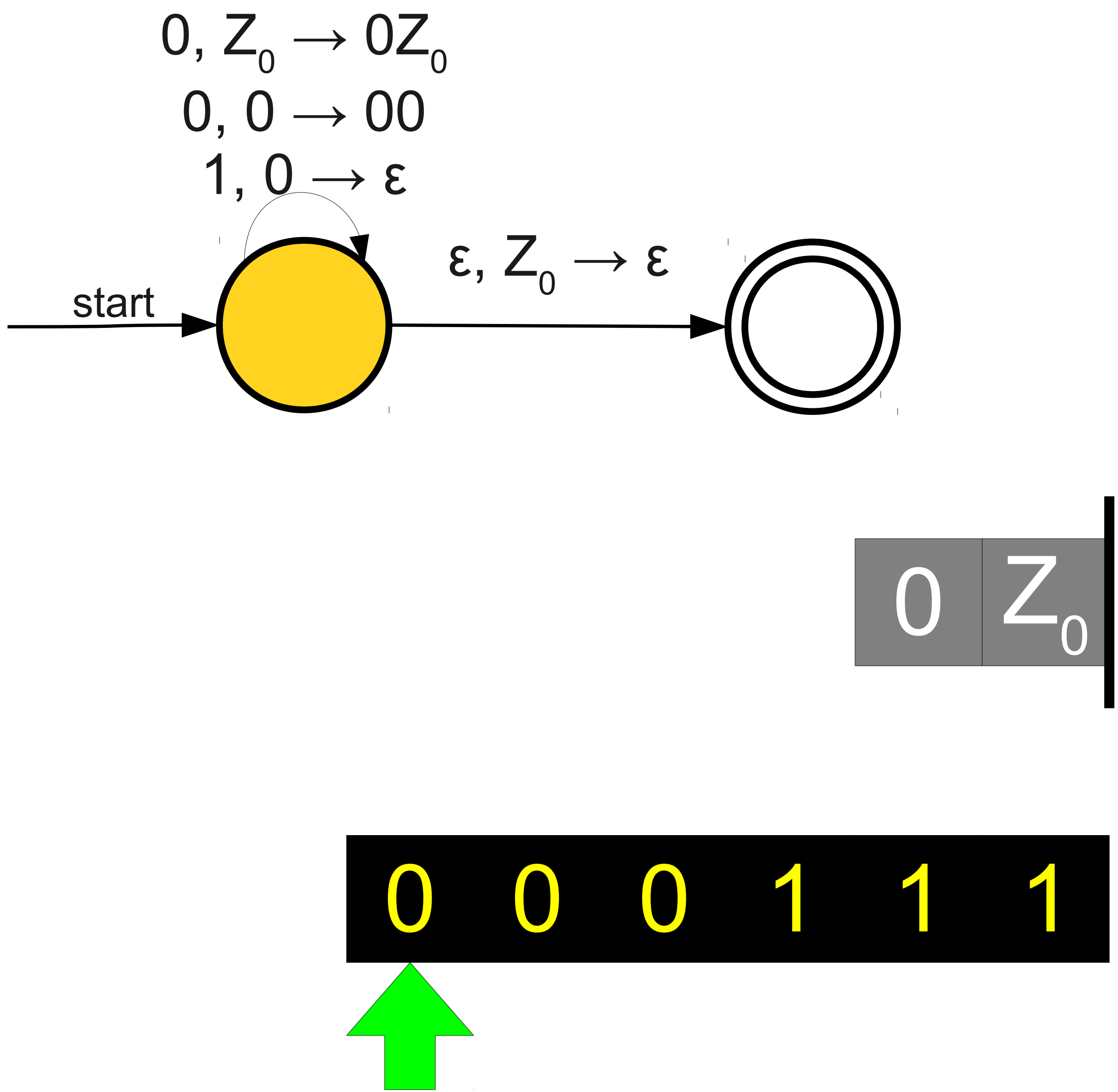
A Simple Pushdown Automaton



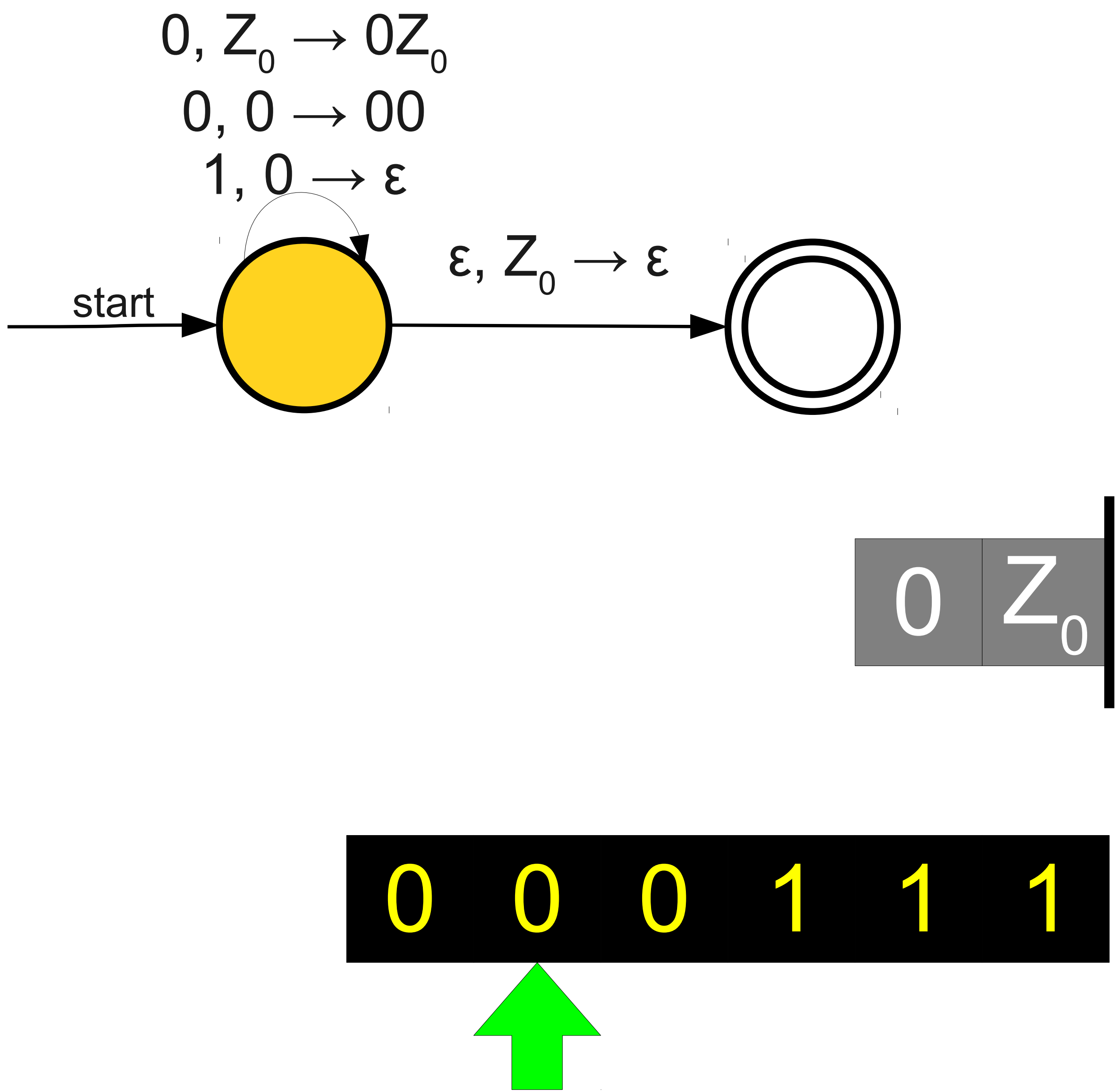
A Simple Pushdown Automaton



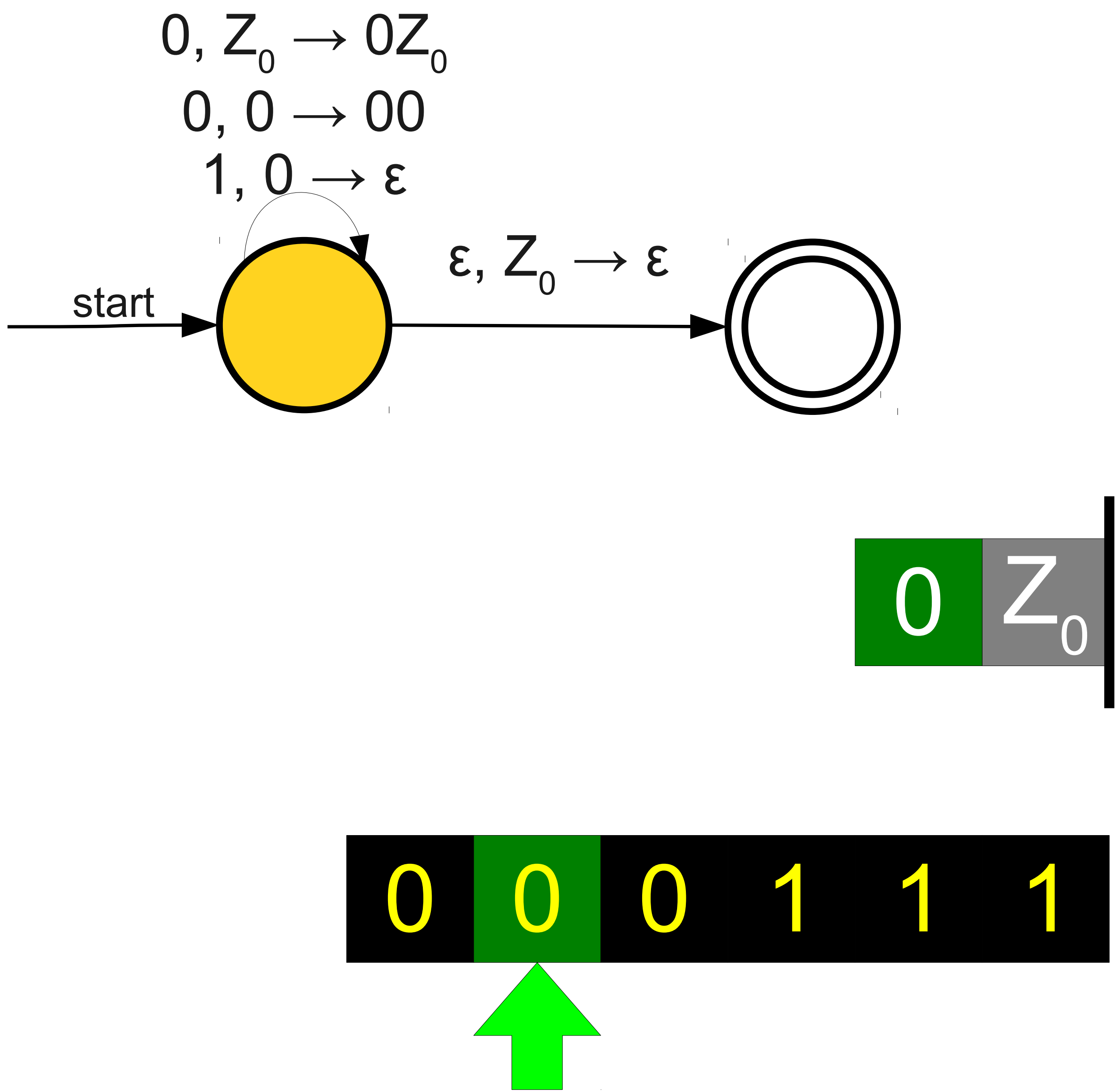
A Simple Pushdown Automaton



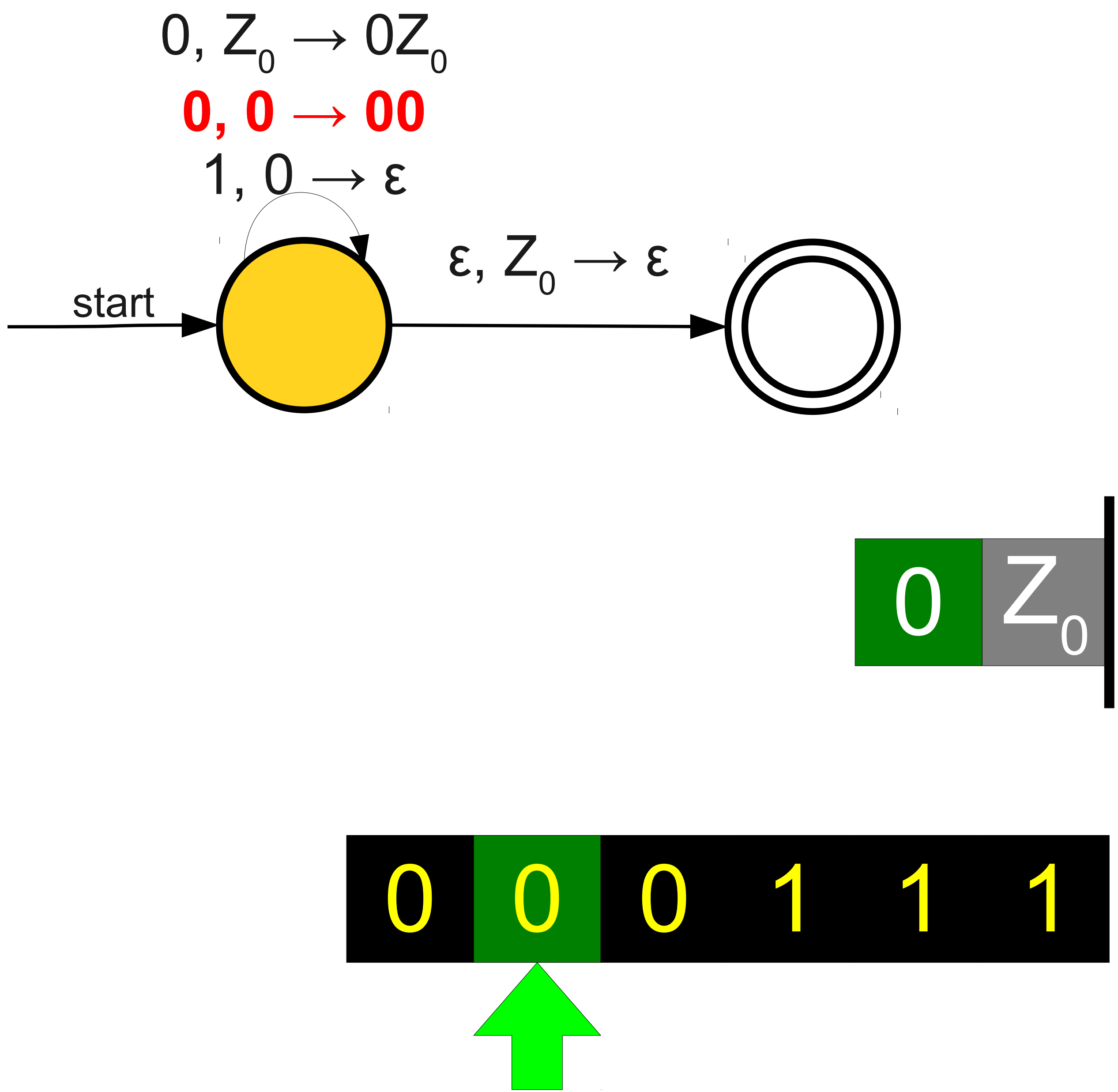
A Simple Pushdown Automaton



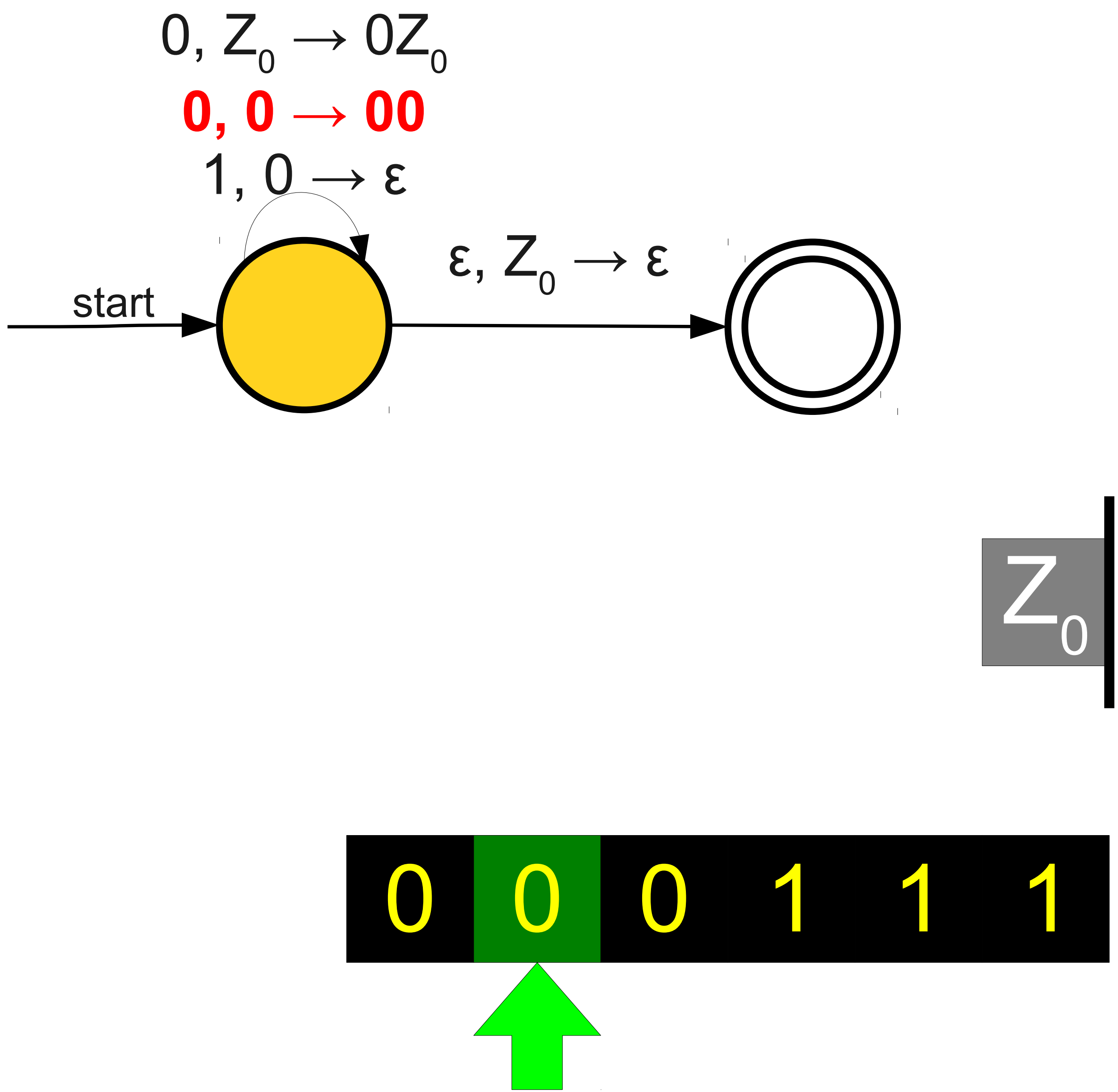
A Simple Pushdown Automaton



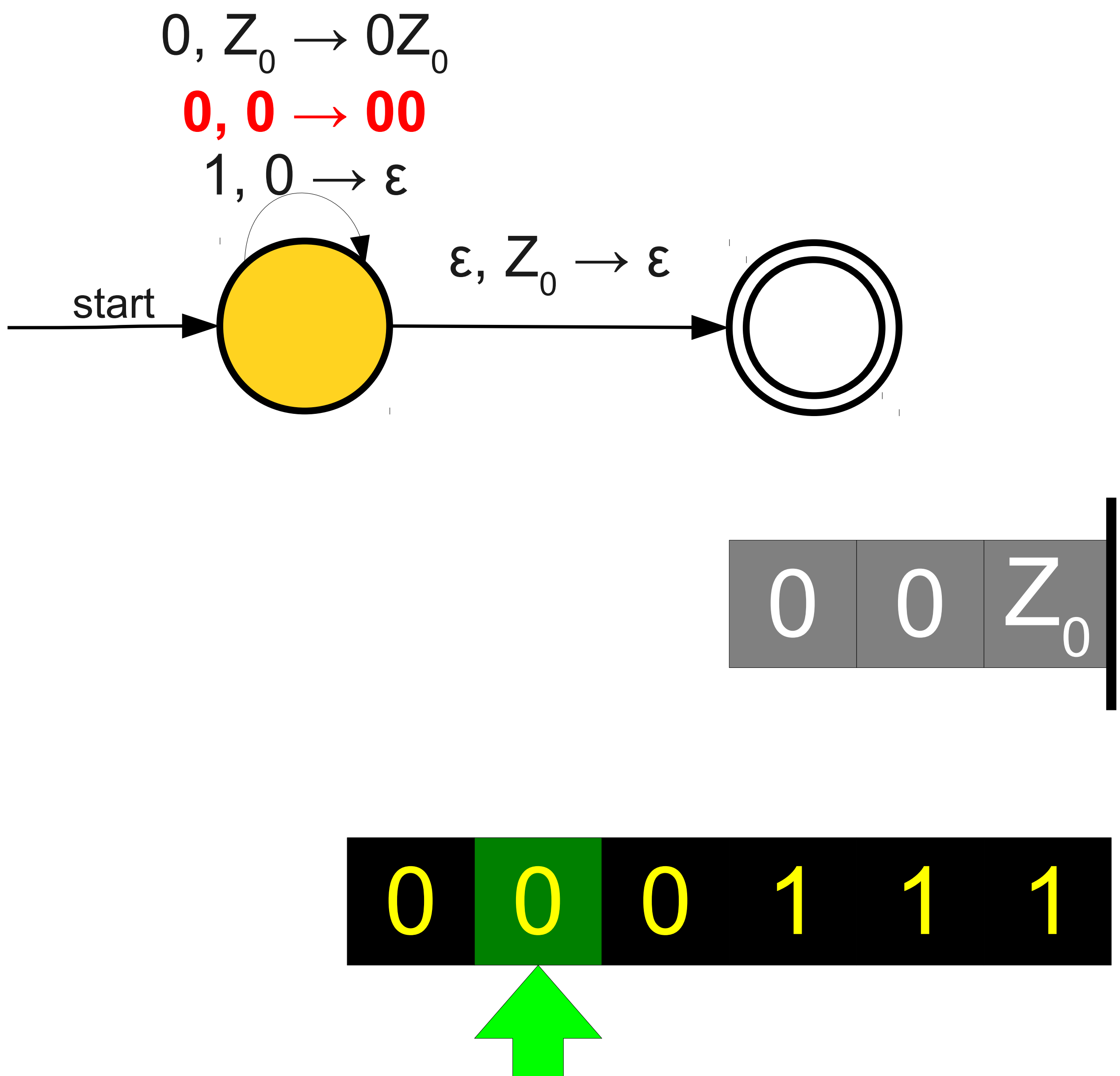
A Simple Pushdown Automaton



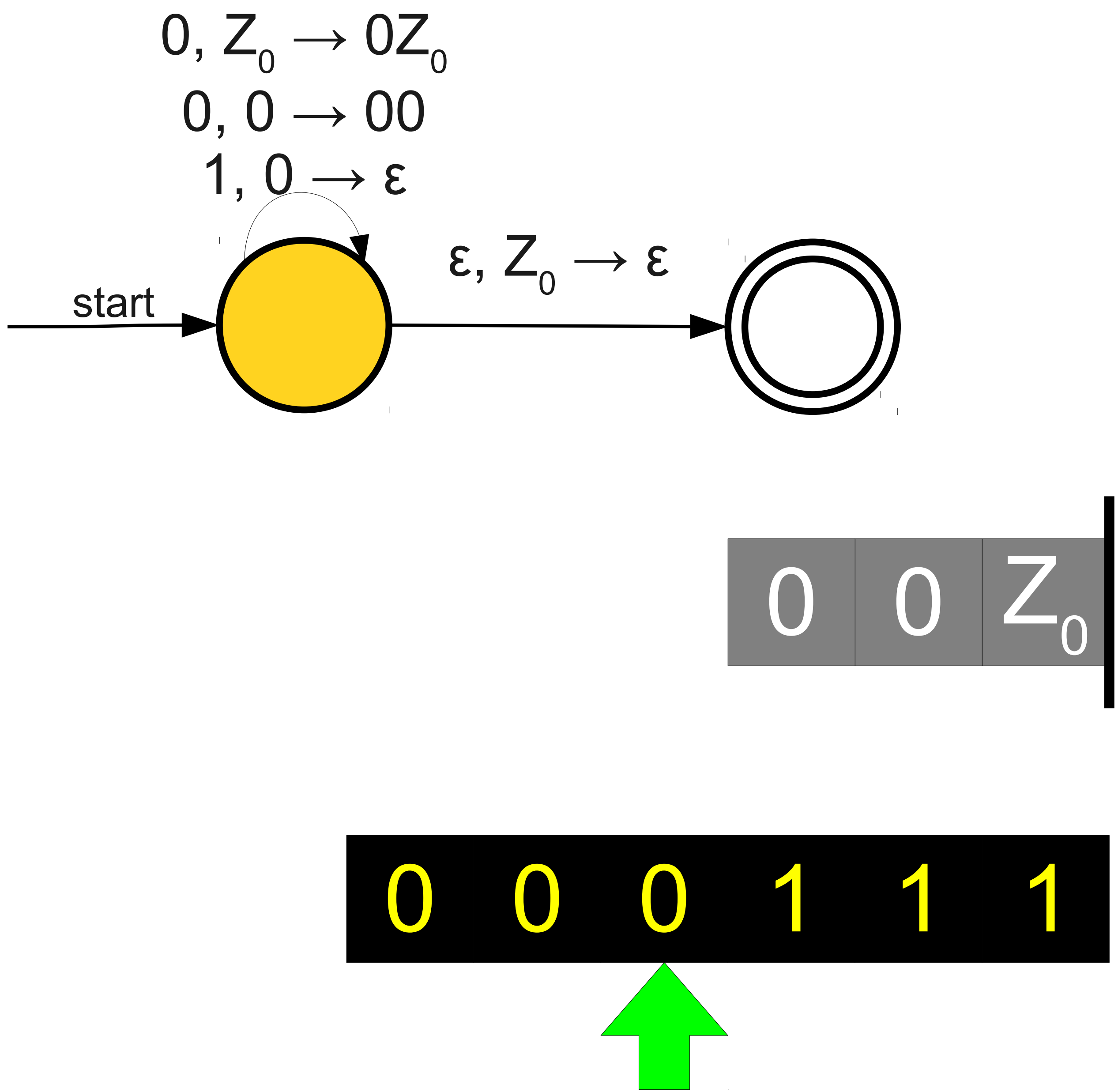
A Simple Pushdown Automaton



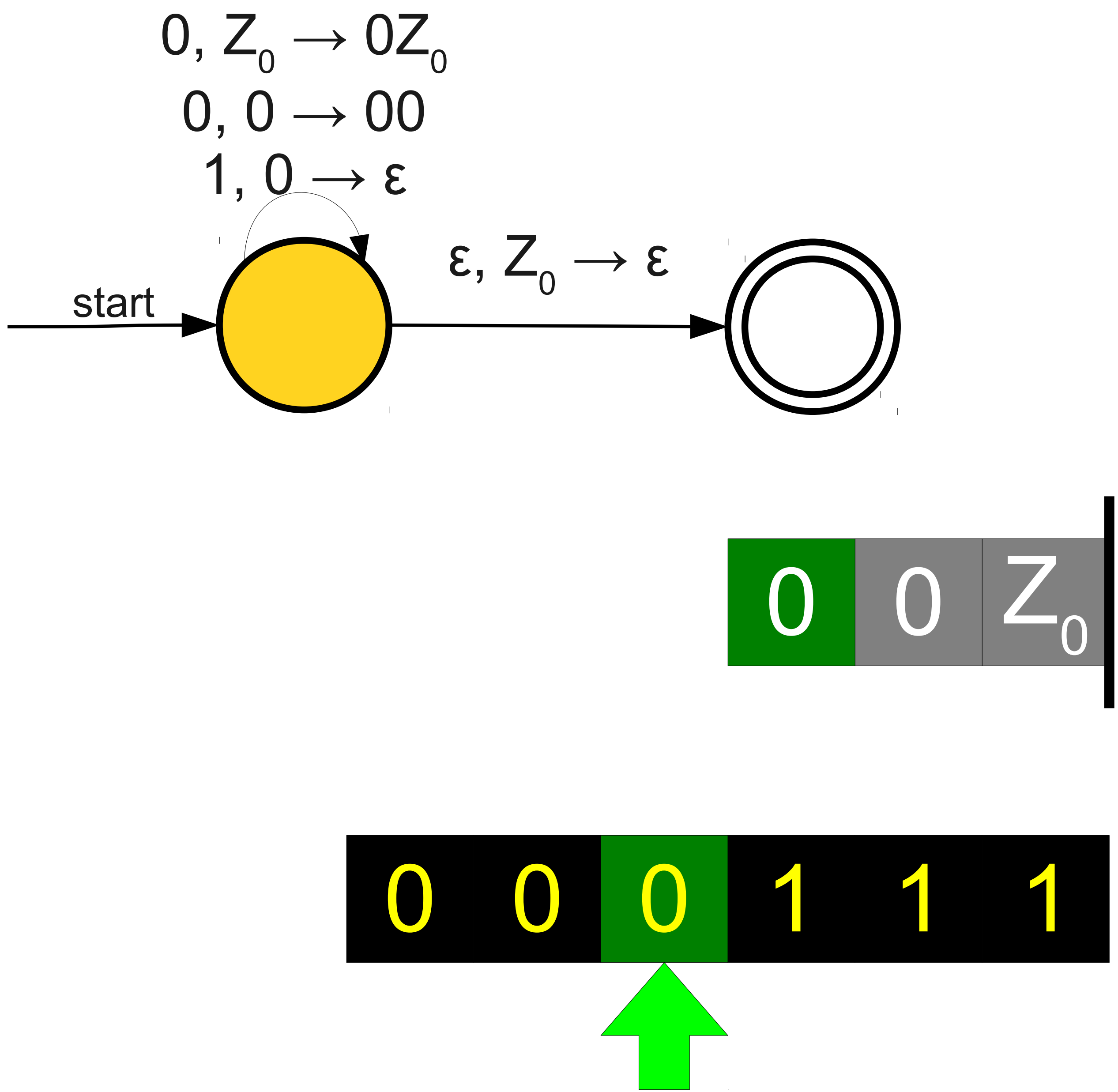
A Simple Pushdown Automaton



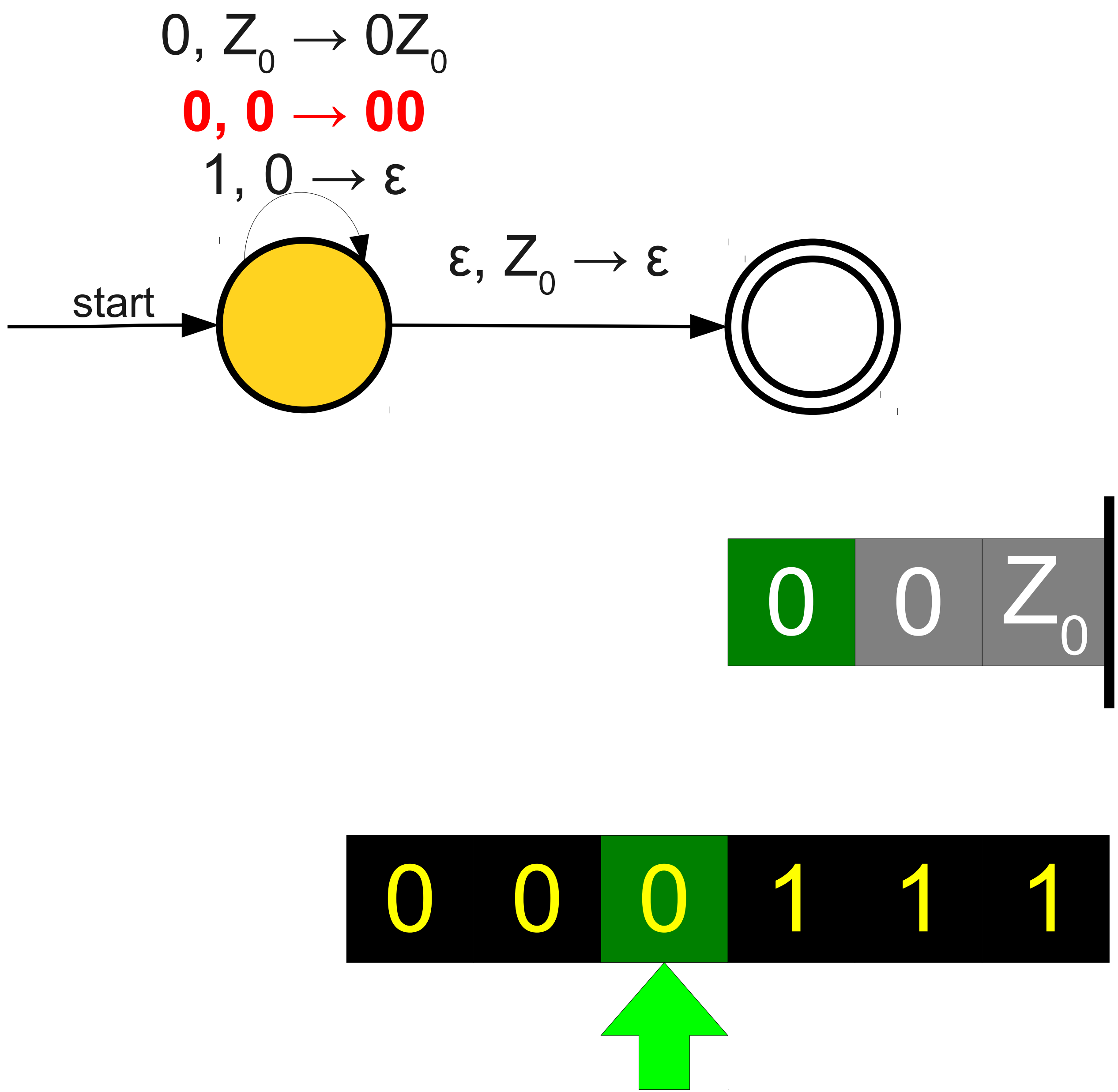
A Simple Pushdown Automaton



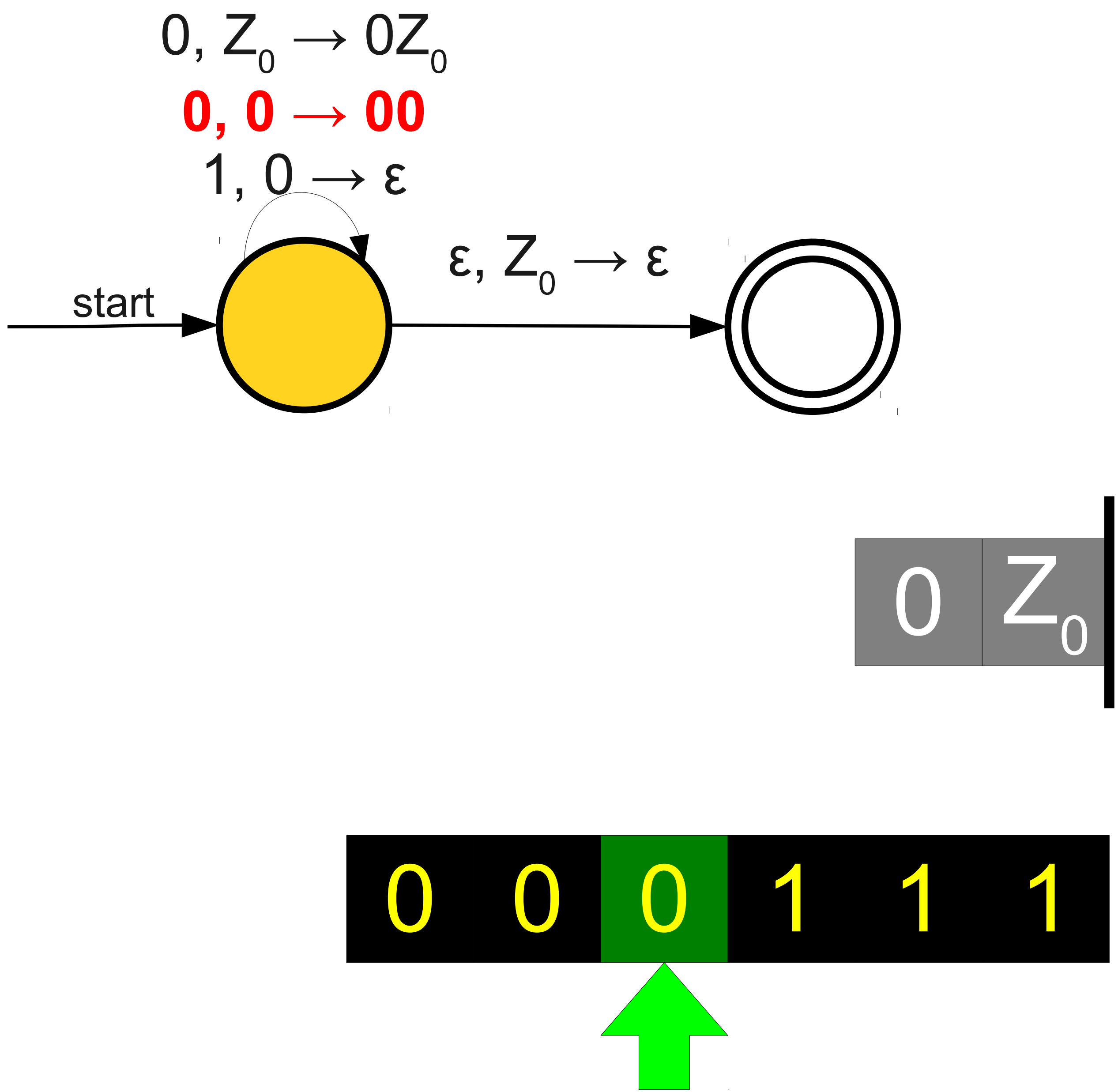
A Simple Pushdown Automaton



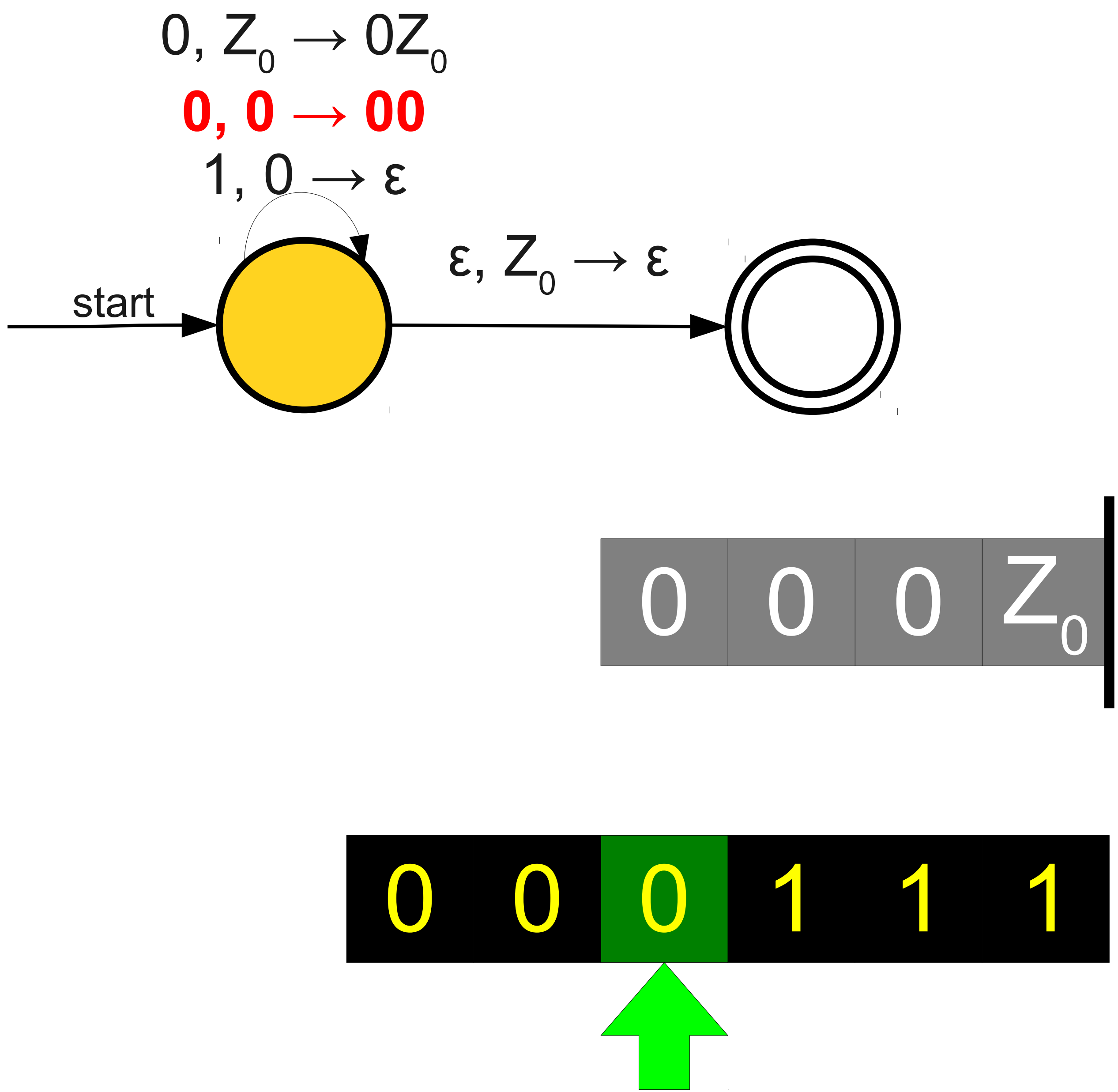
A Simple Pushdown Automaton



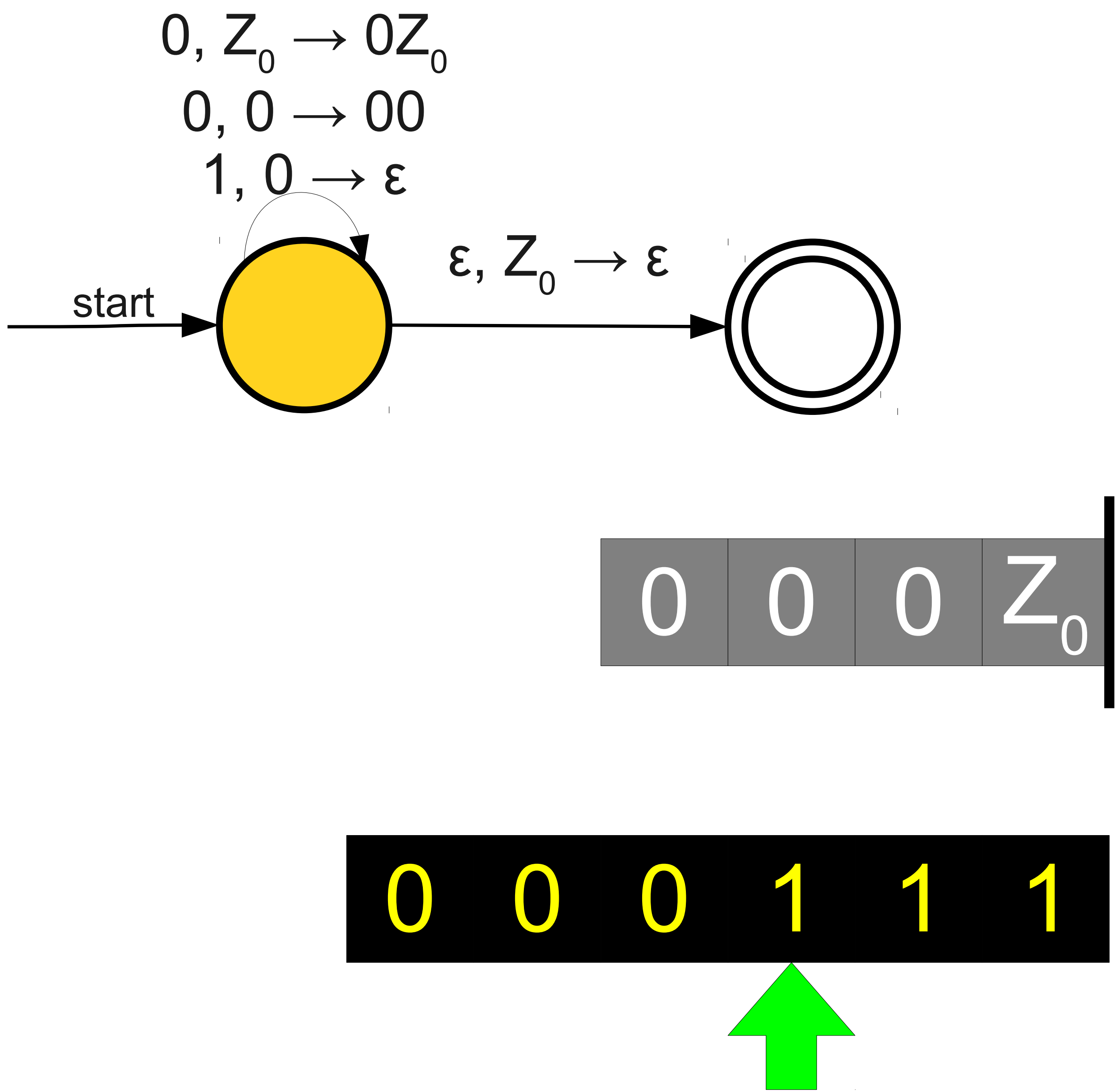
A Simple Pushdown Automaton



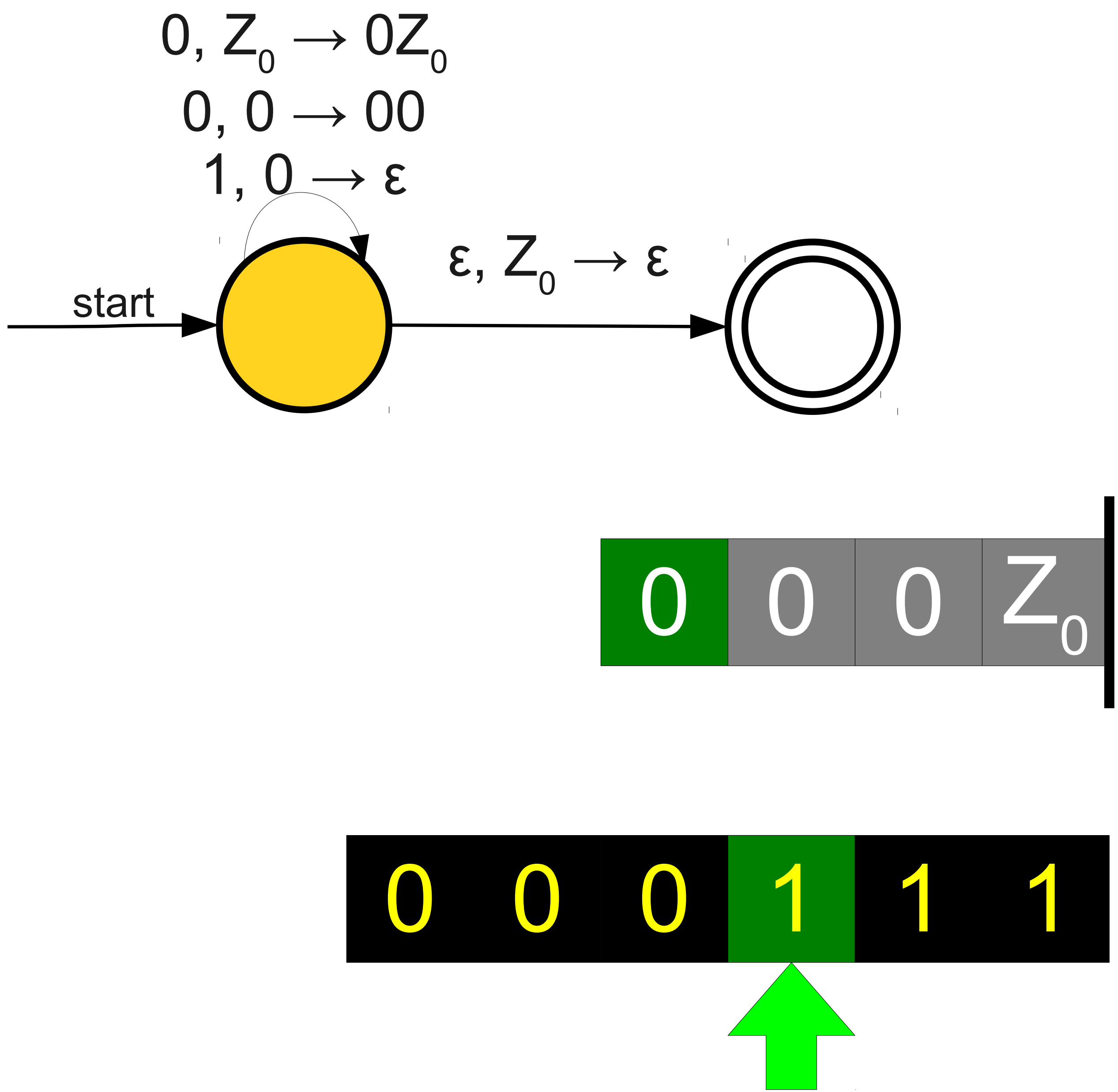
A Simple Pushdown Automaton



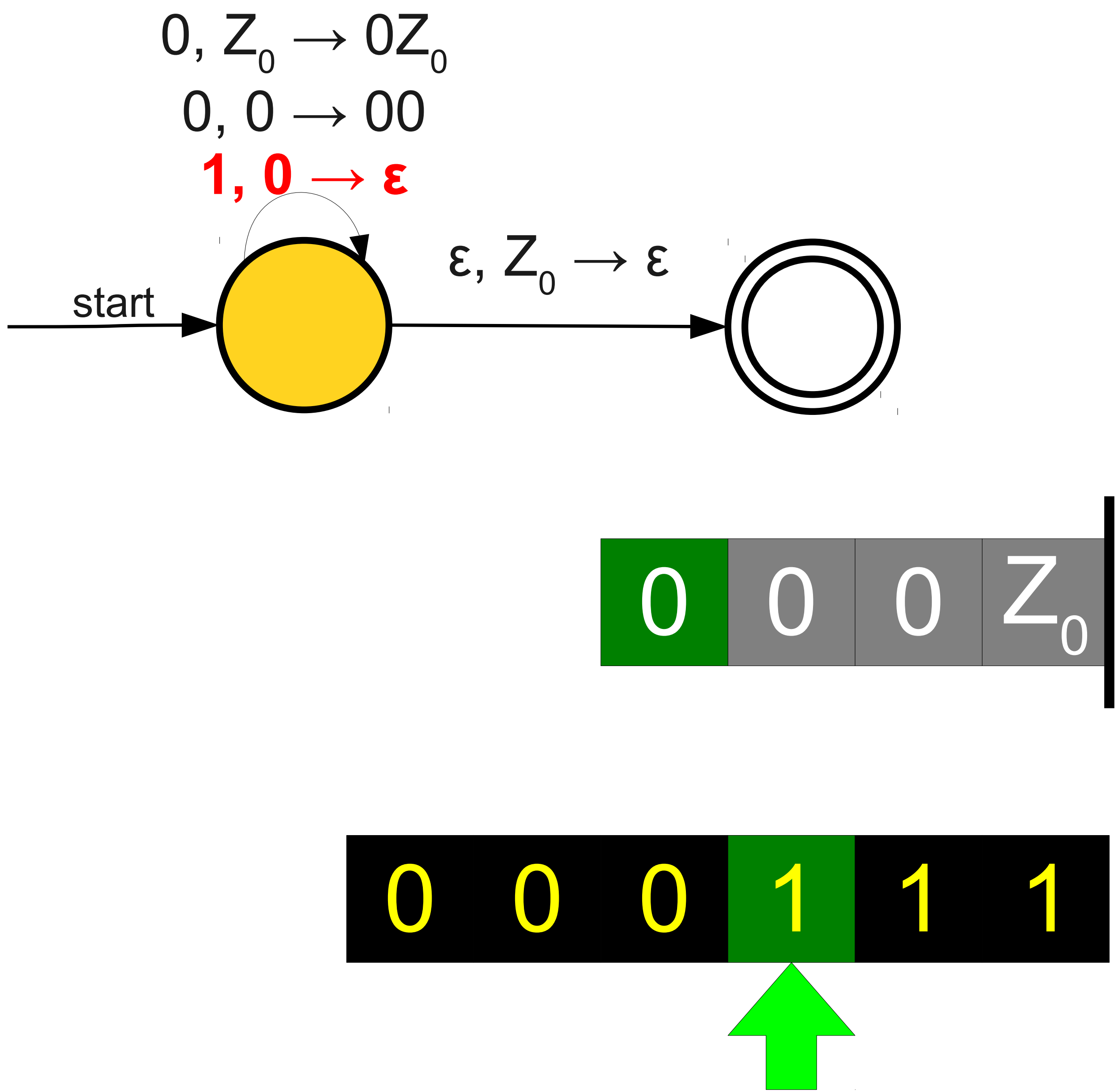
A Simple Pushdown Automaton



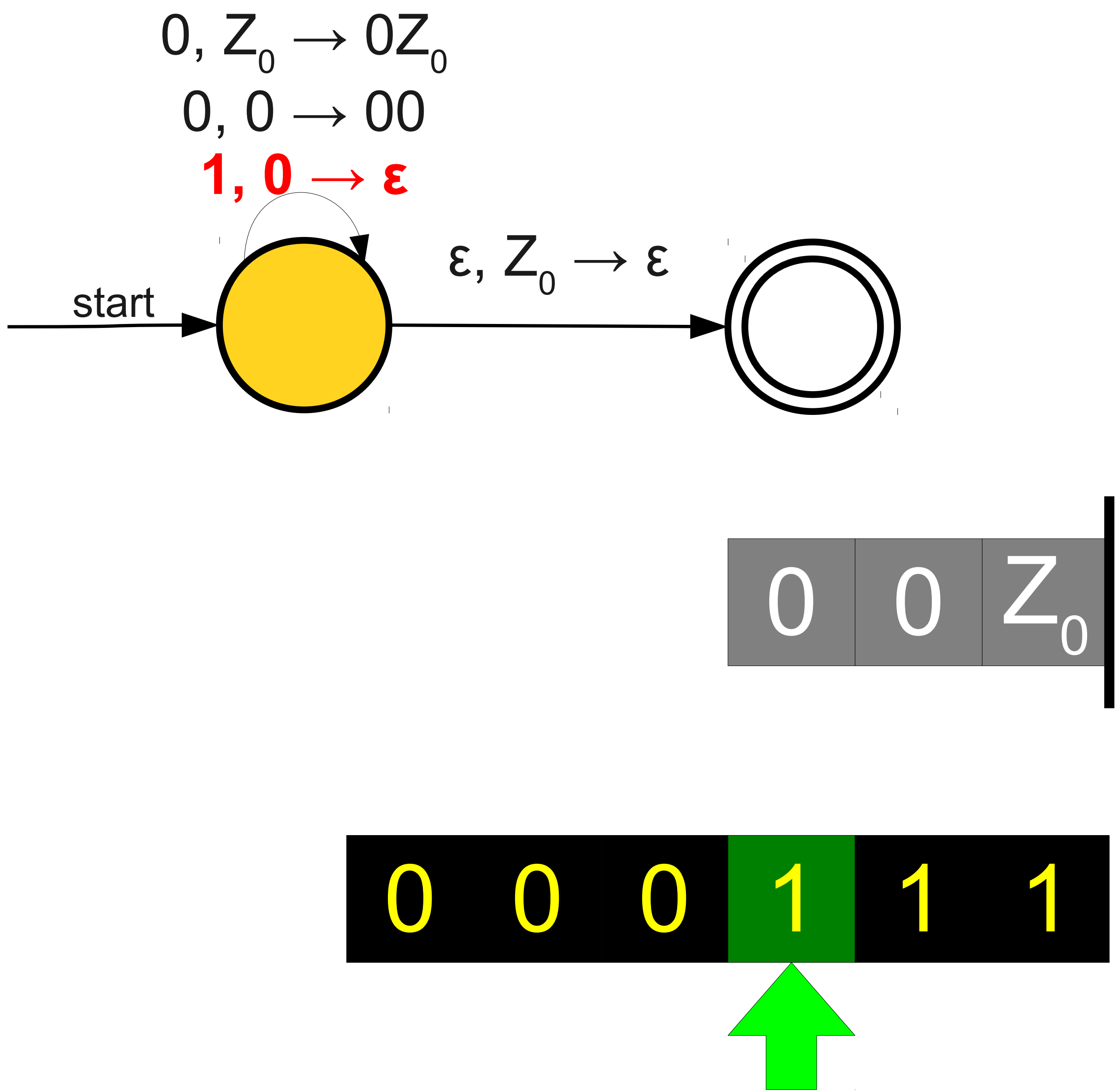
A Simple Pushdown Automaton



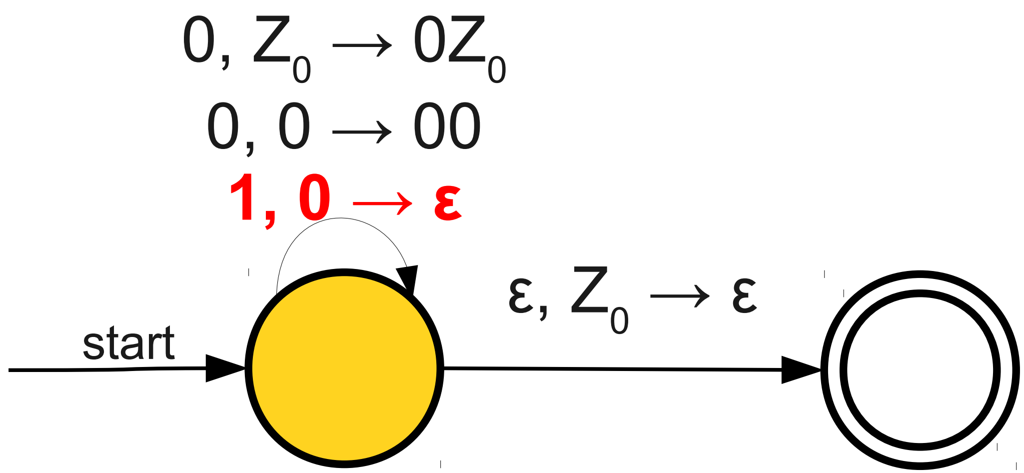
A Simple Pushdown Automaton



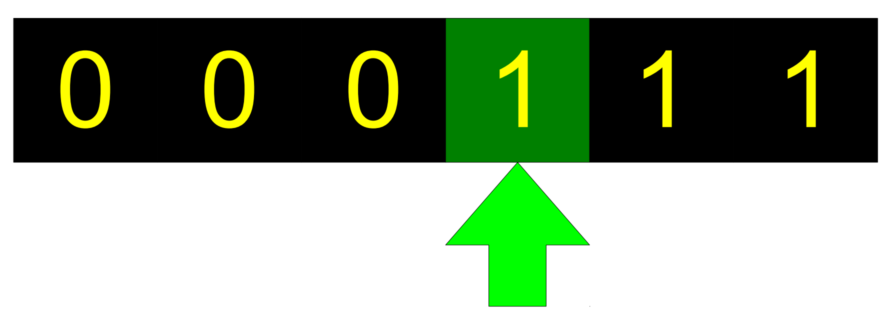
A Simple Pushdown Automaton



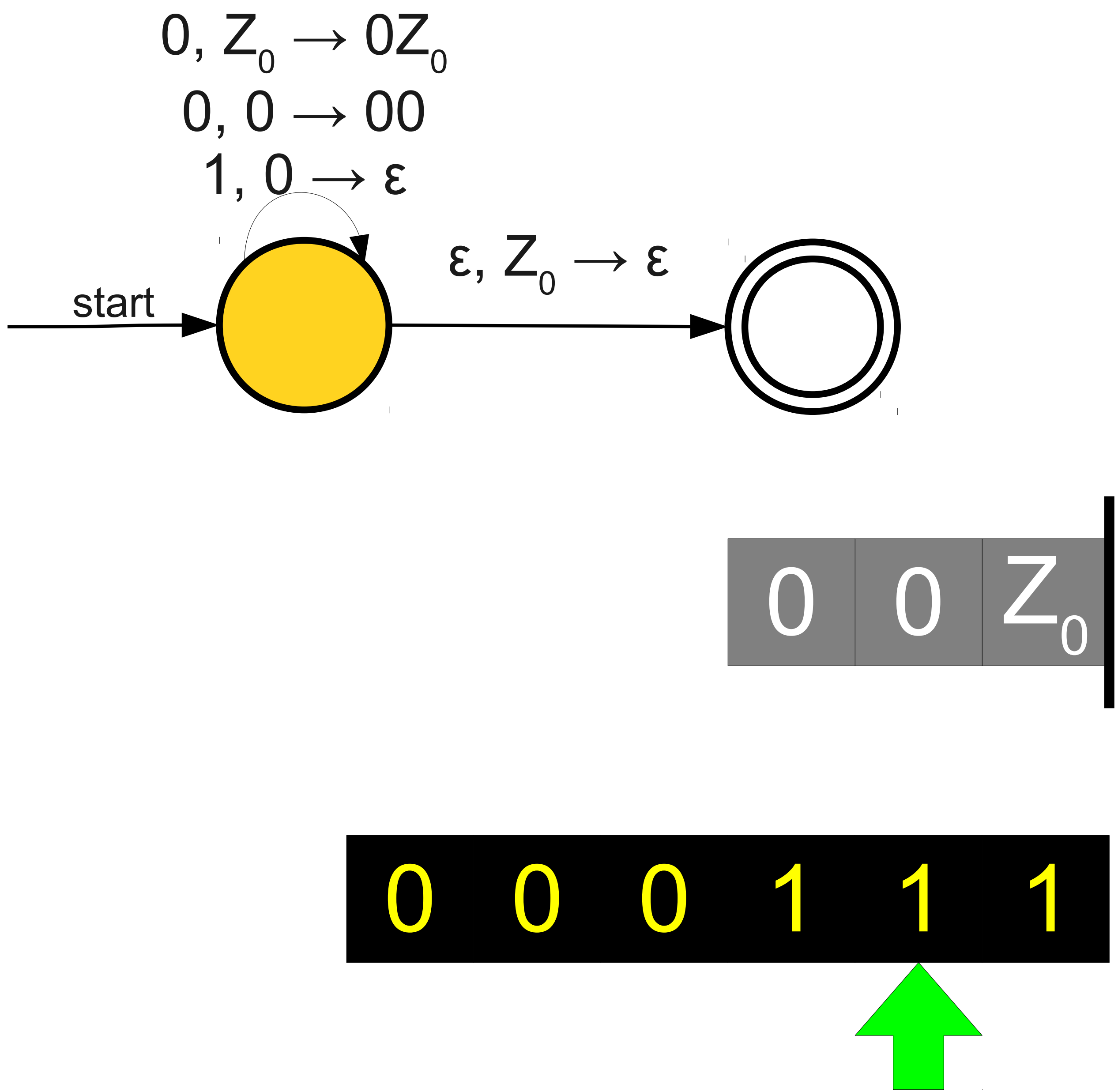
A Simple Pushdown Automaton



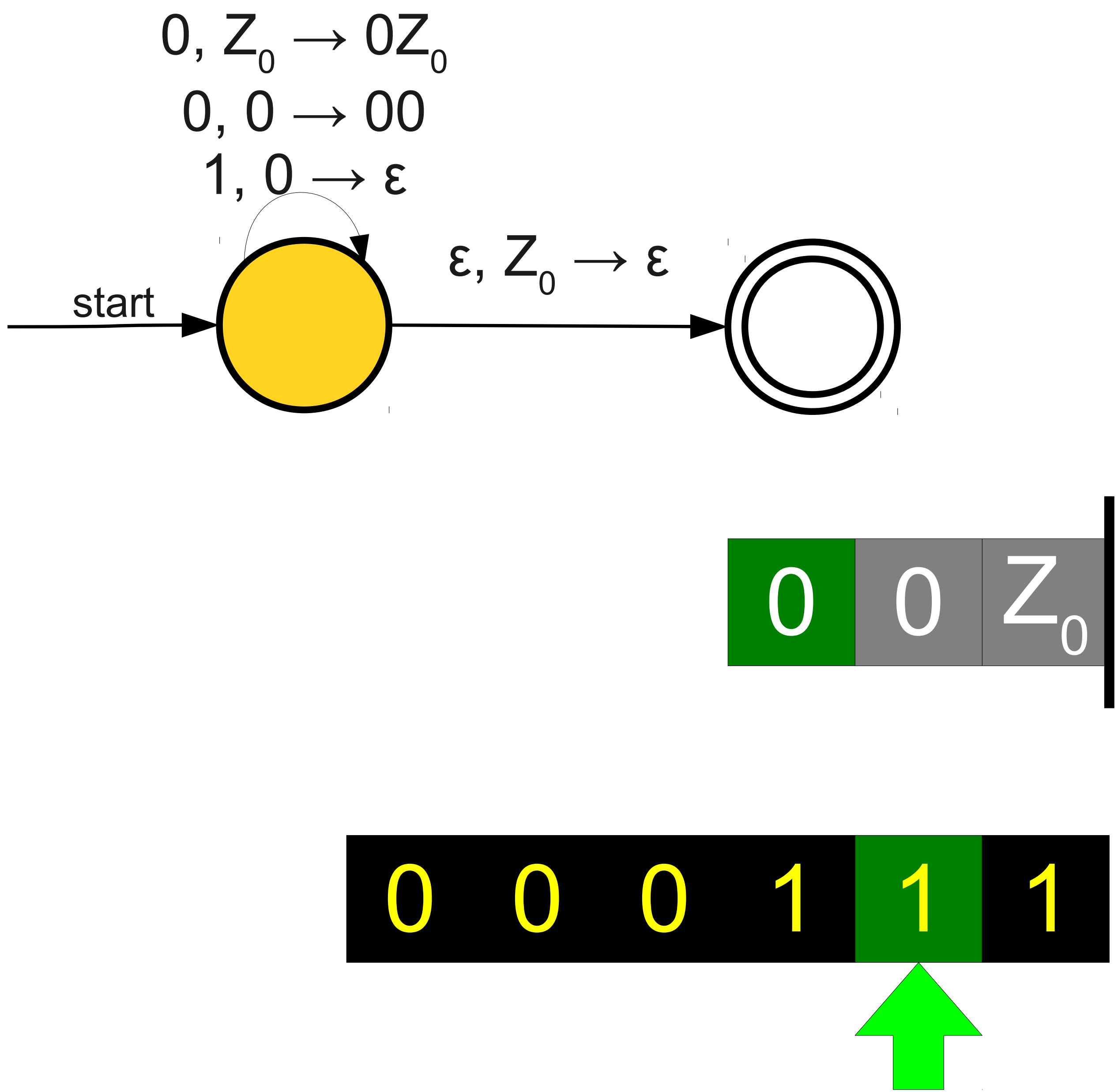
We now push the string ε onto the stack, which adds no new characters. This essentially means "pop the stack."



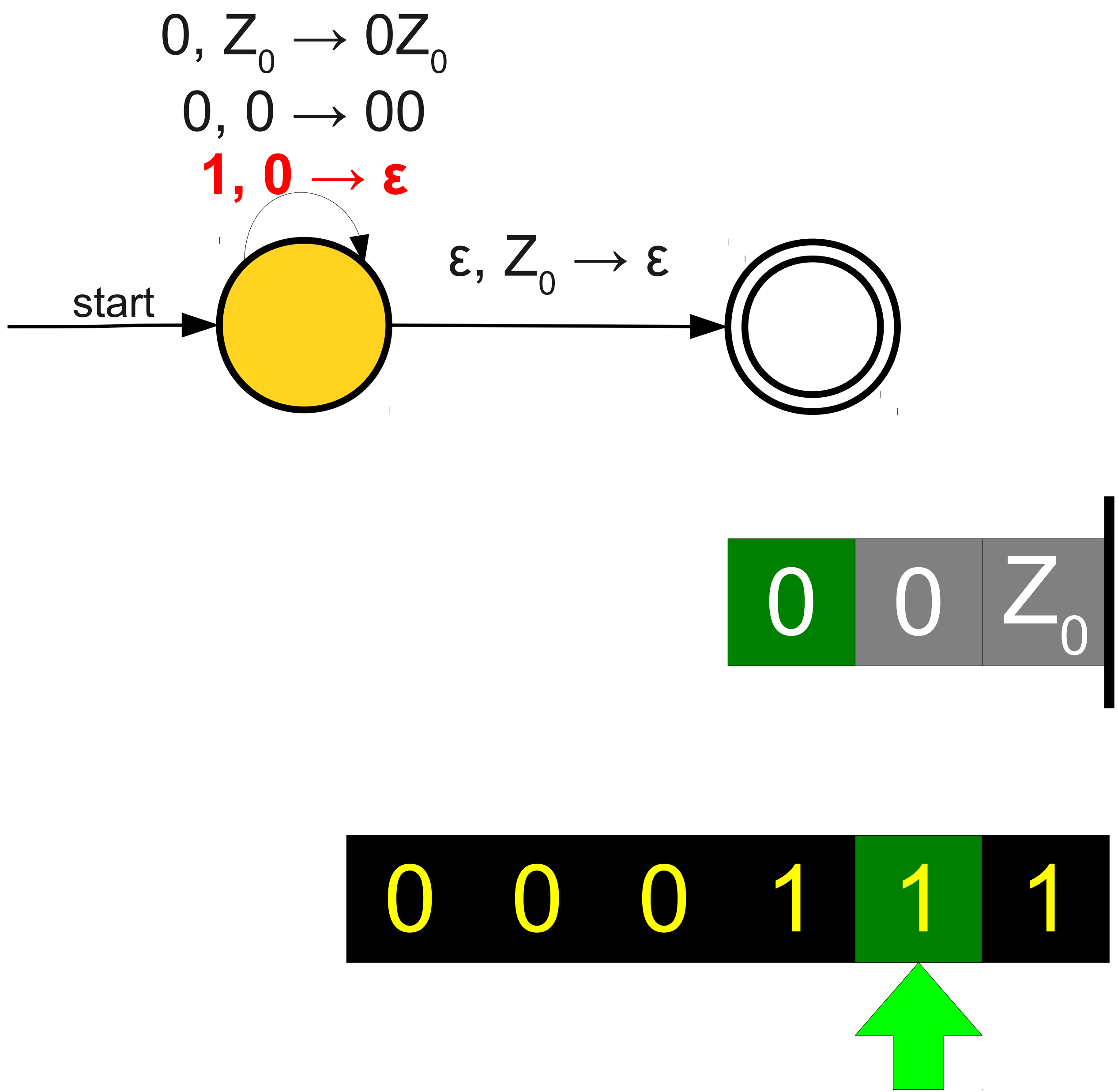
A Simple Pushdown Automaton



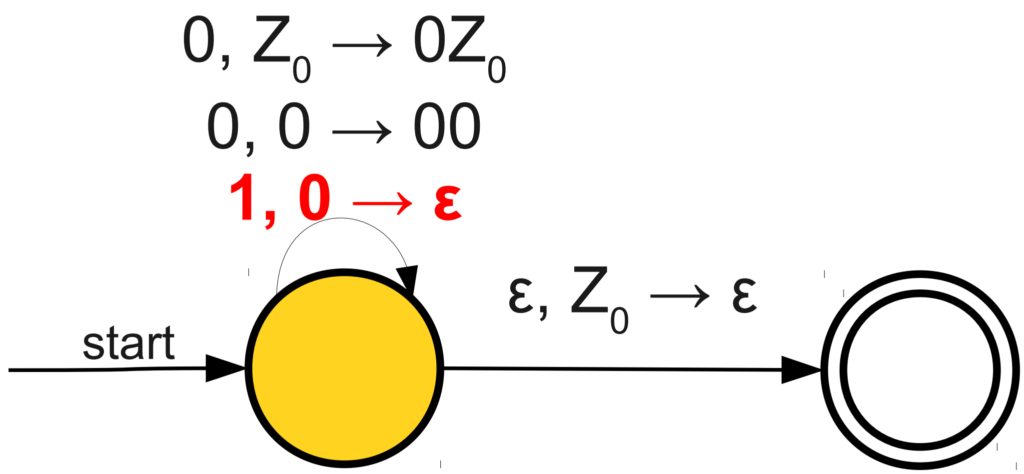
A Simple Pushdown Automaton

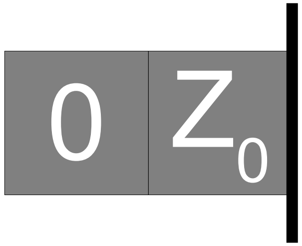


A Simple Pushdown Automaton

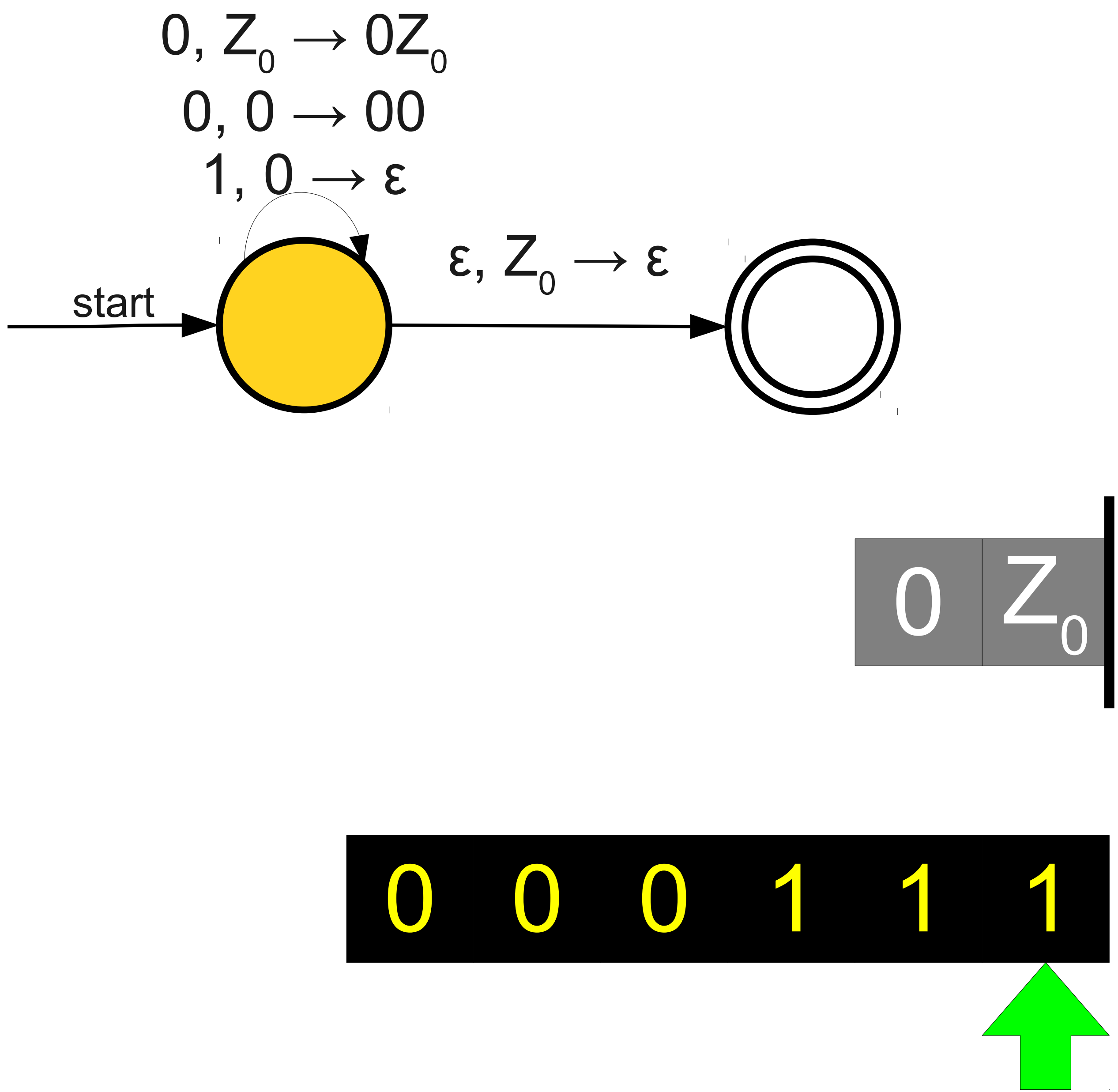


A Simple Pushdown Automaton

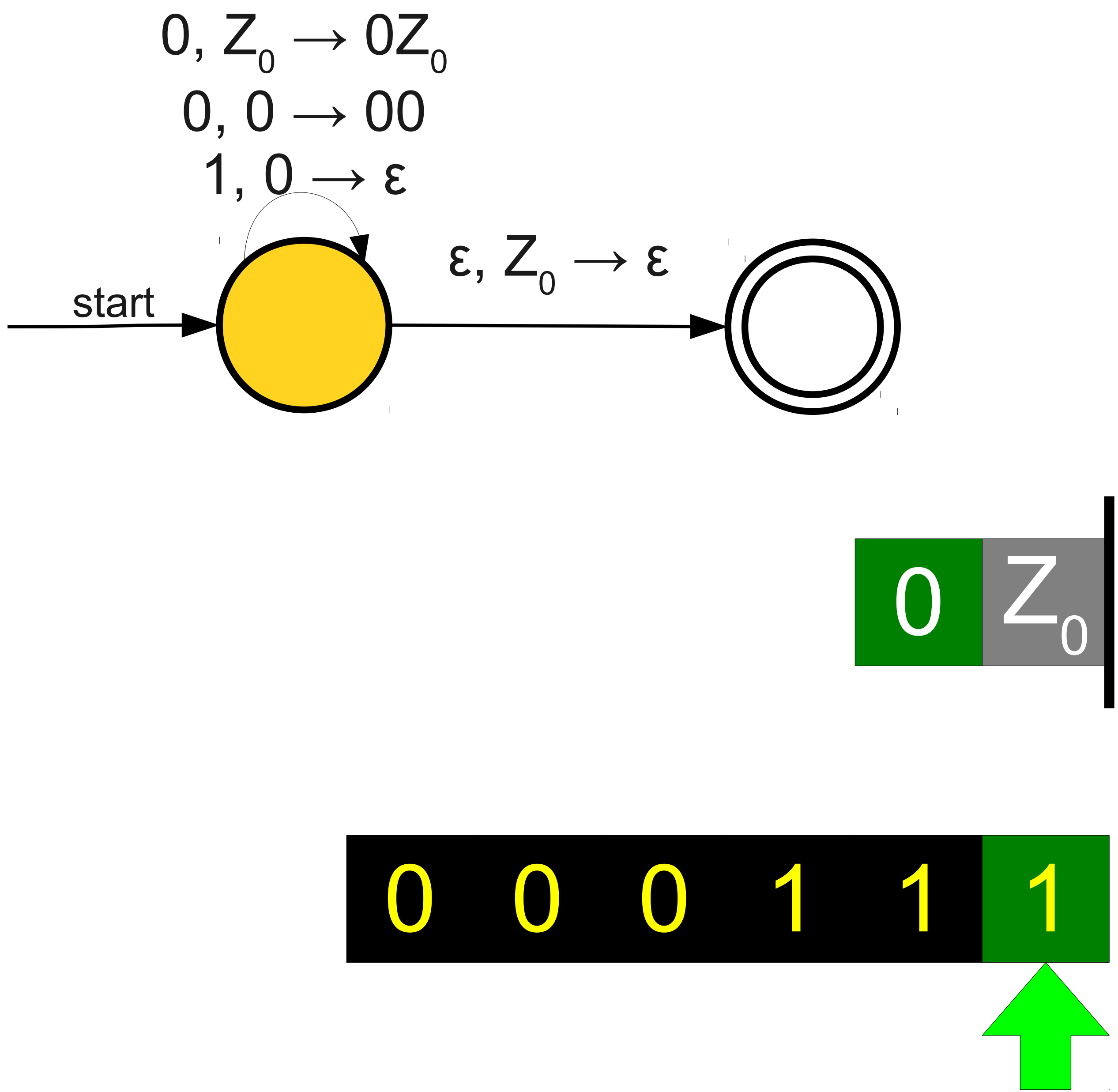




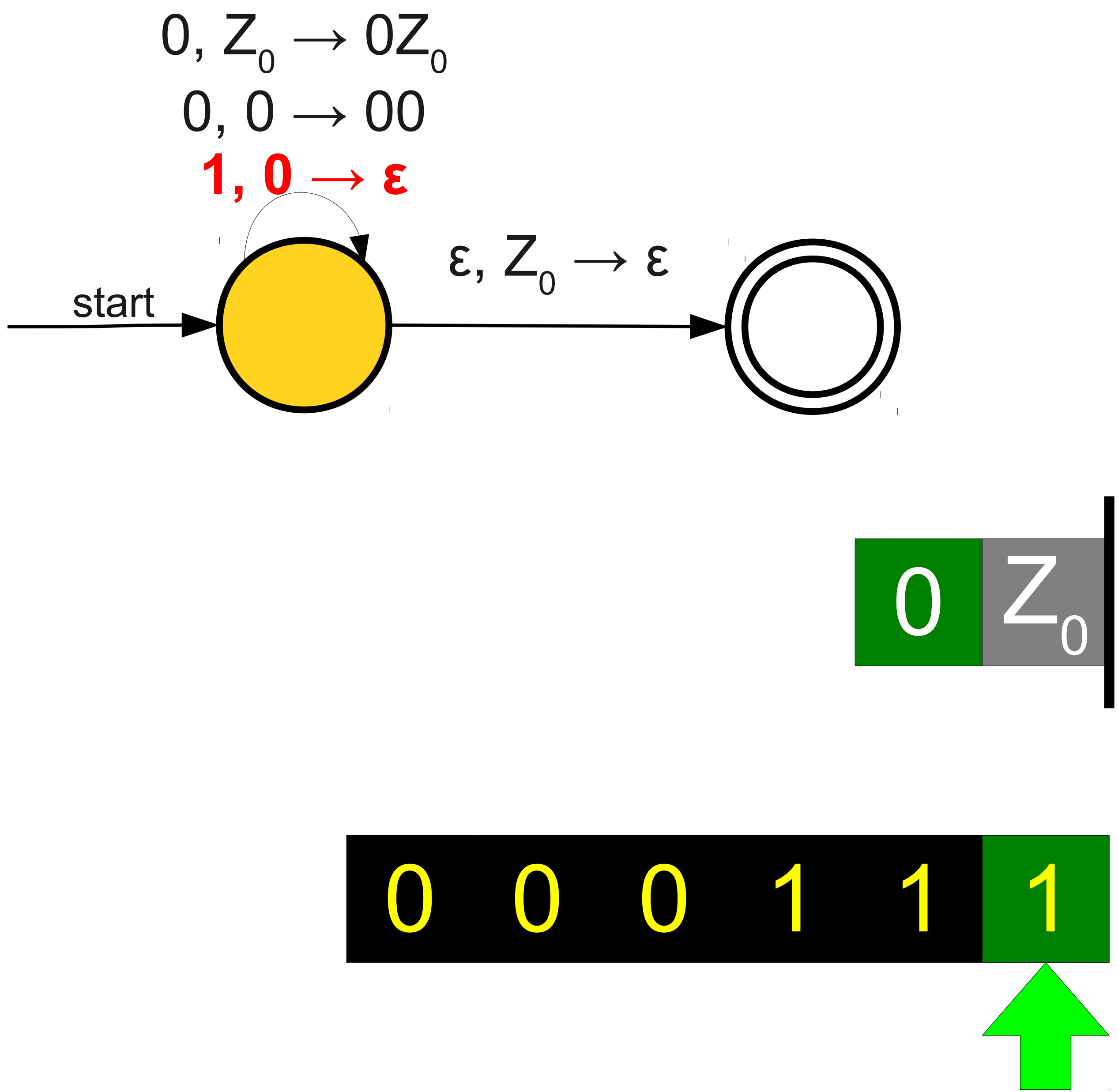
A Simple Pushdown Automaton



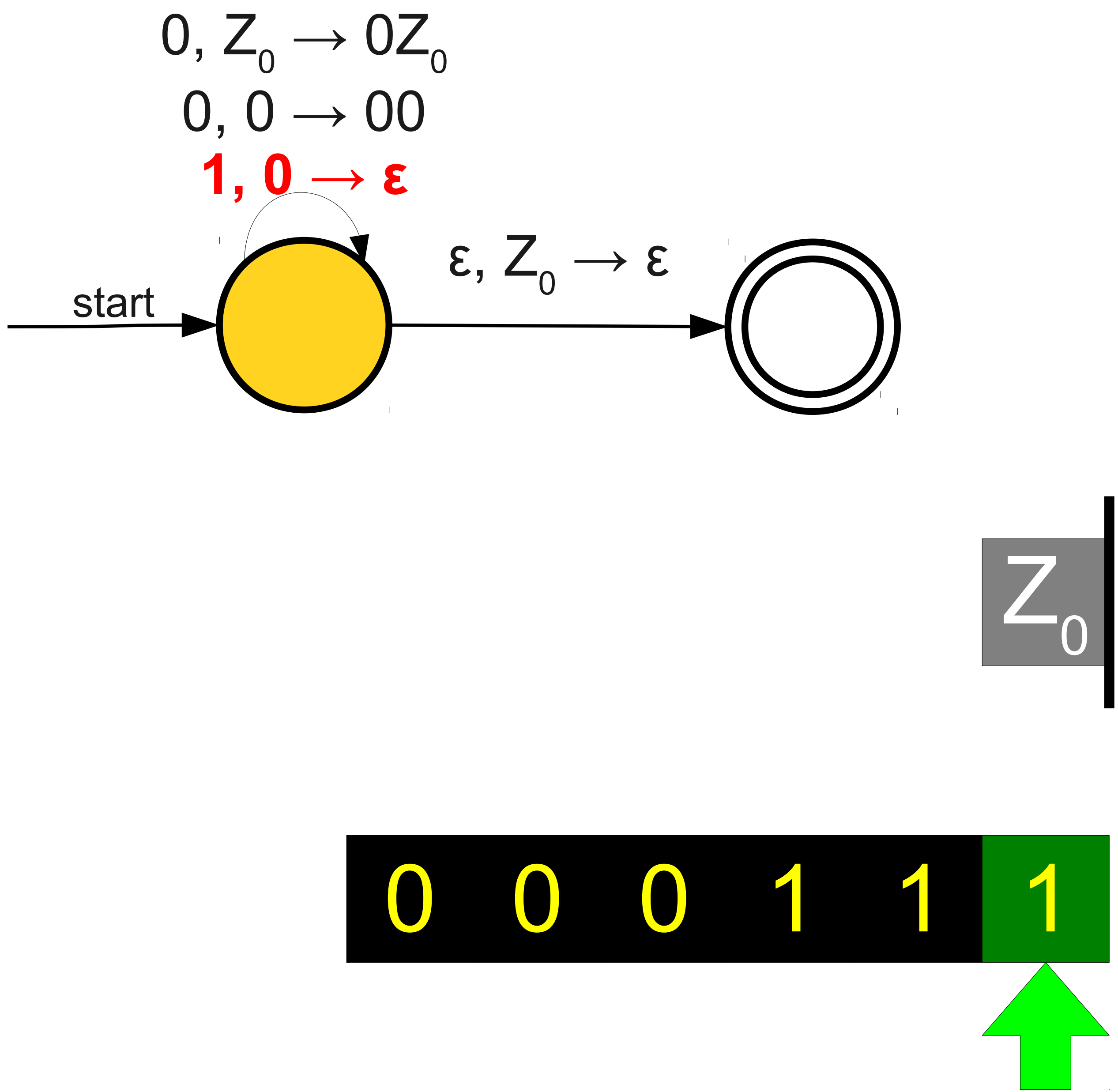
A Simple Pushdown Automaton



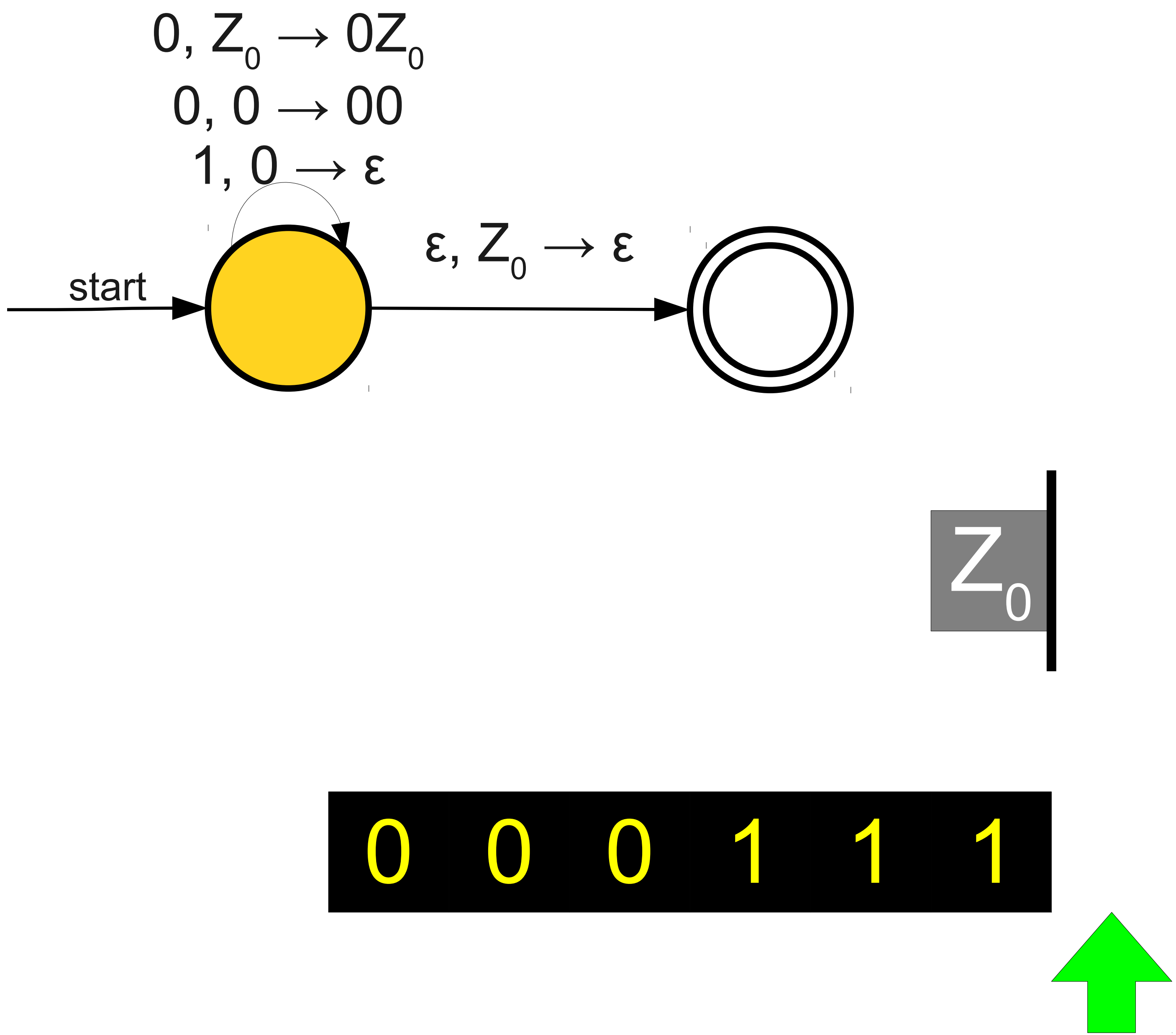
A Simple Pushdown Automaton



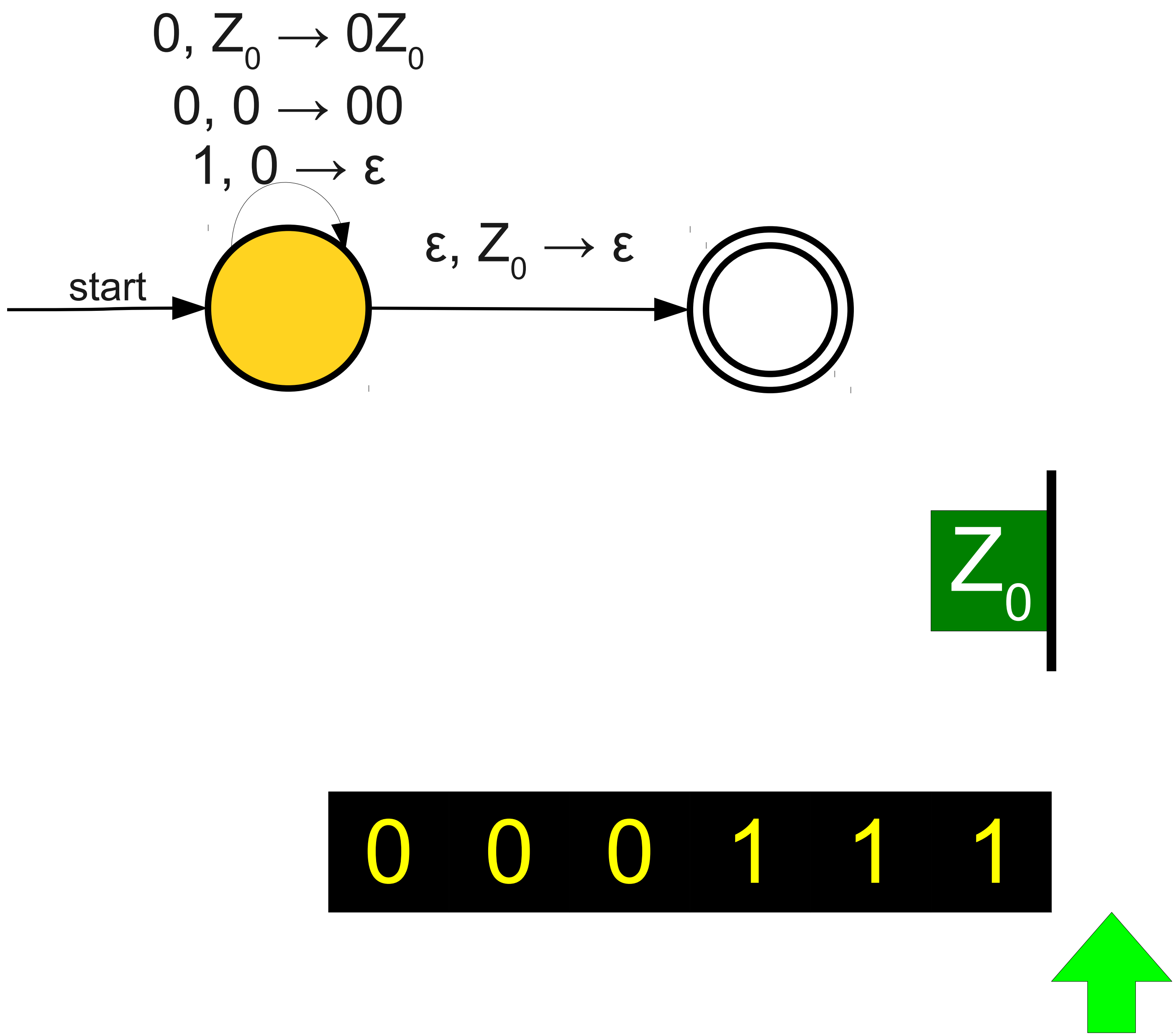
A Simple Pushdown Automaton



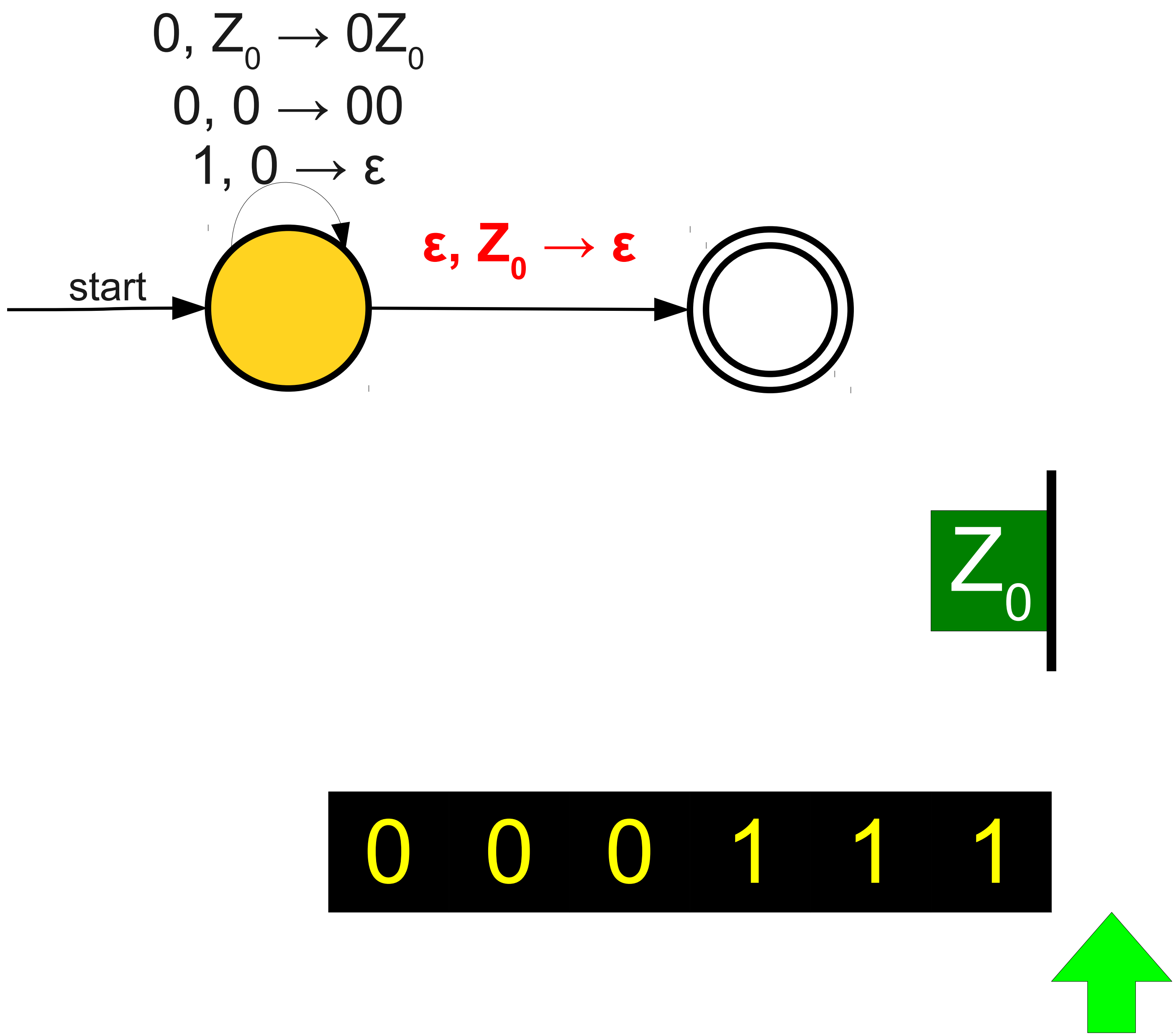
A Simple Pushdown Automaton



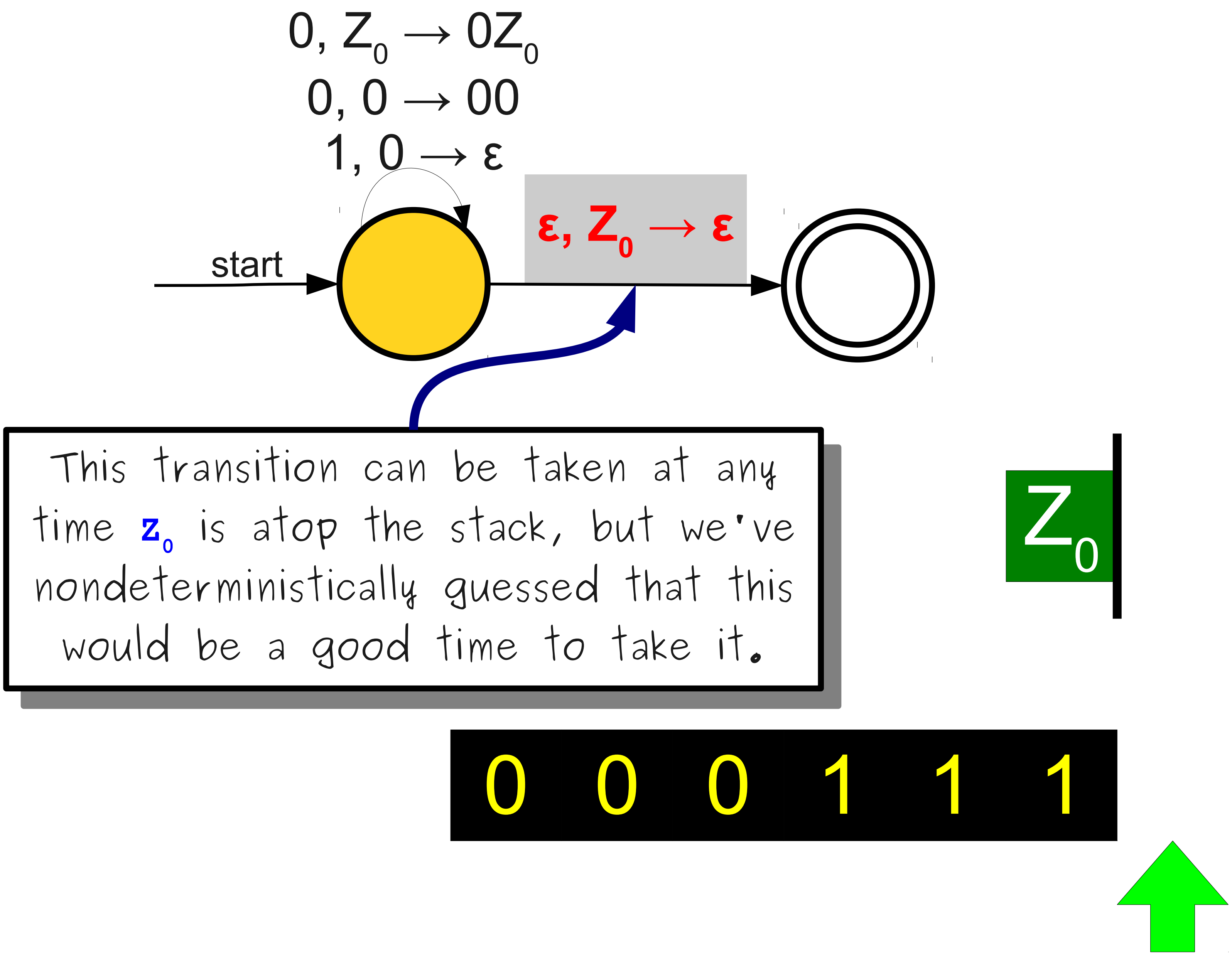
A Simple Pushdown Automaton



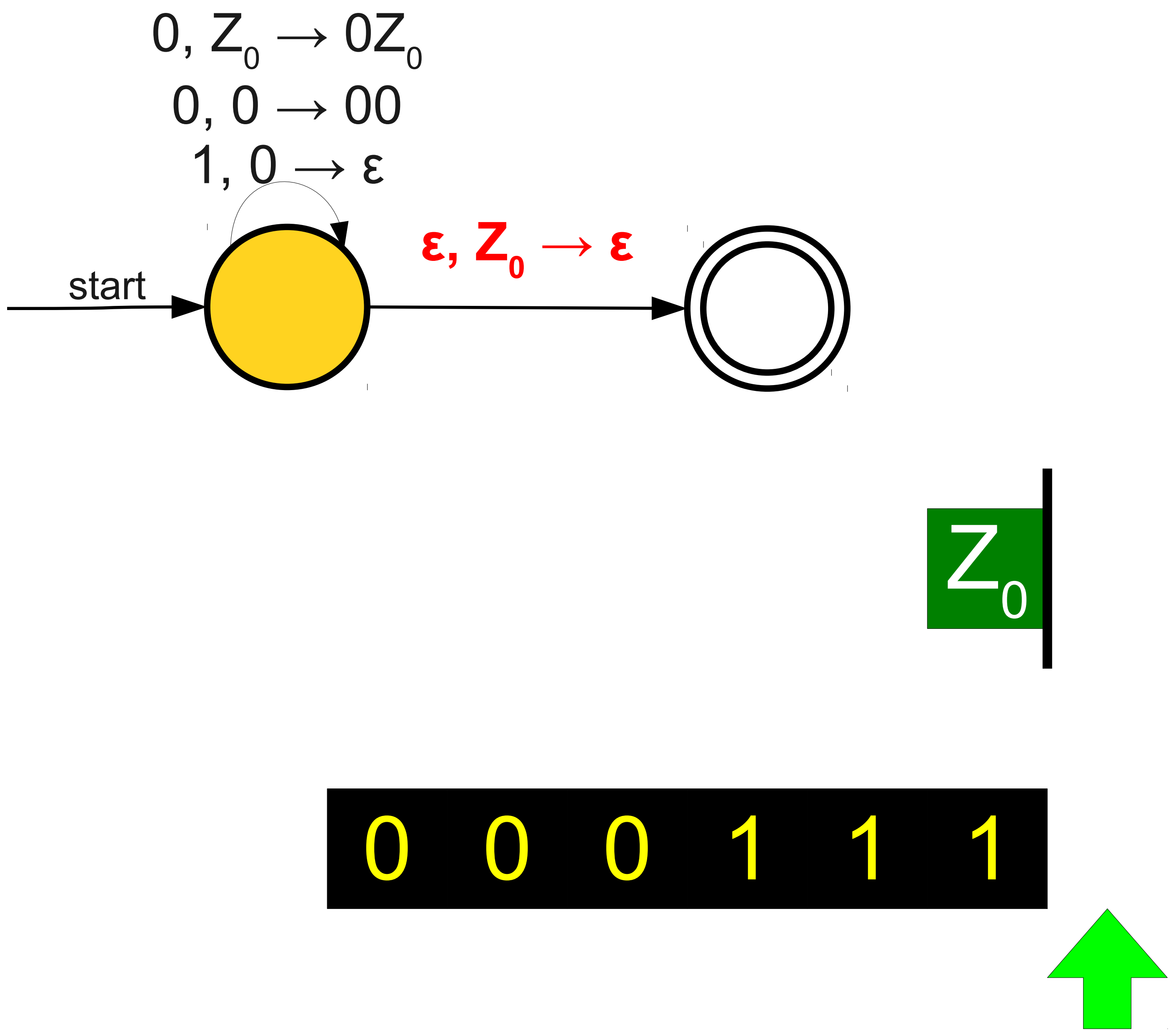
A Simple Pushdown Automaton



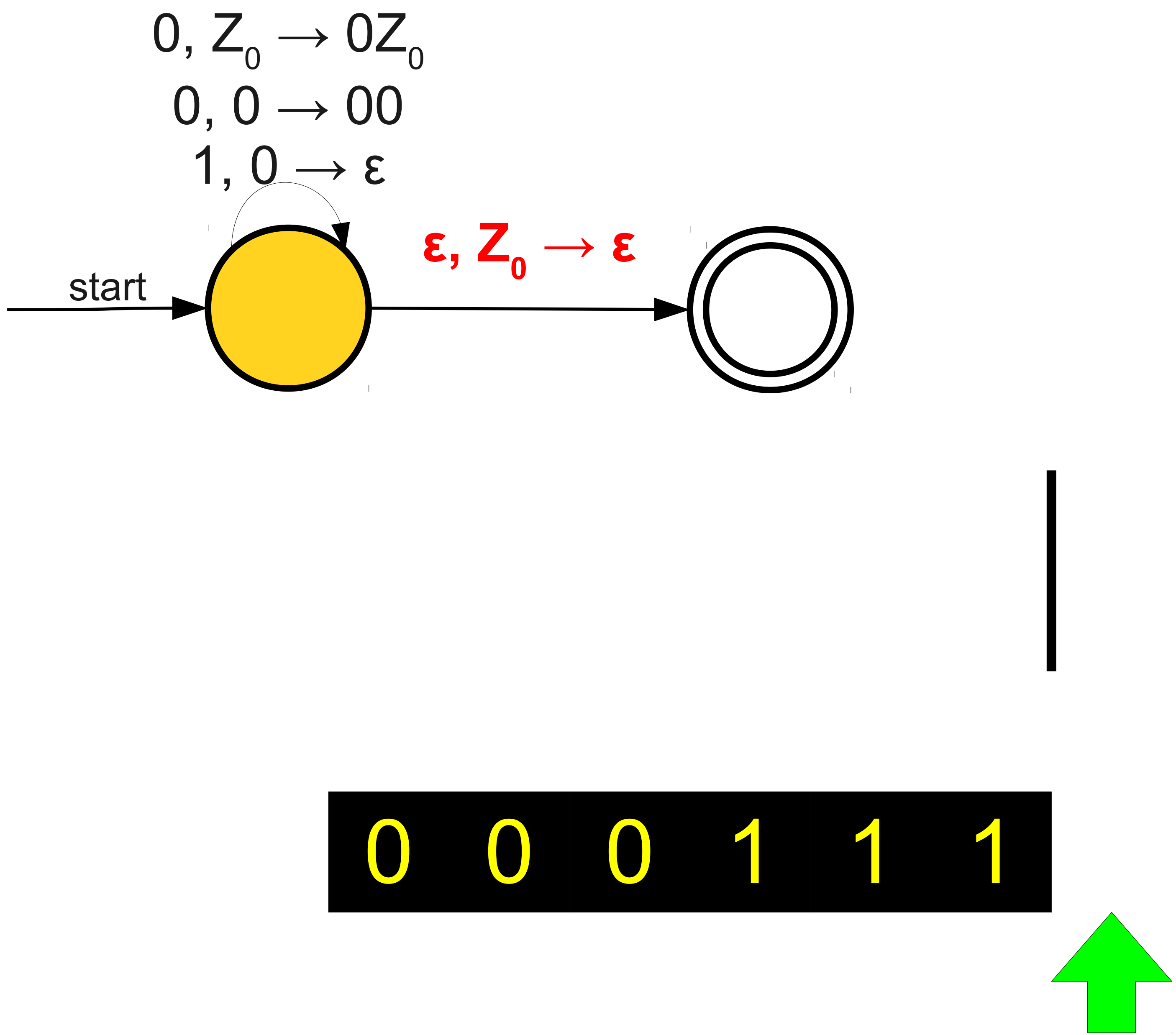
A Simple Pushdown Automaton



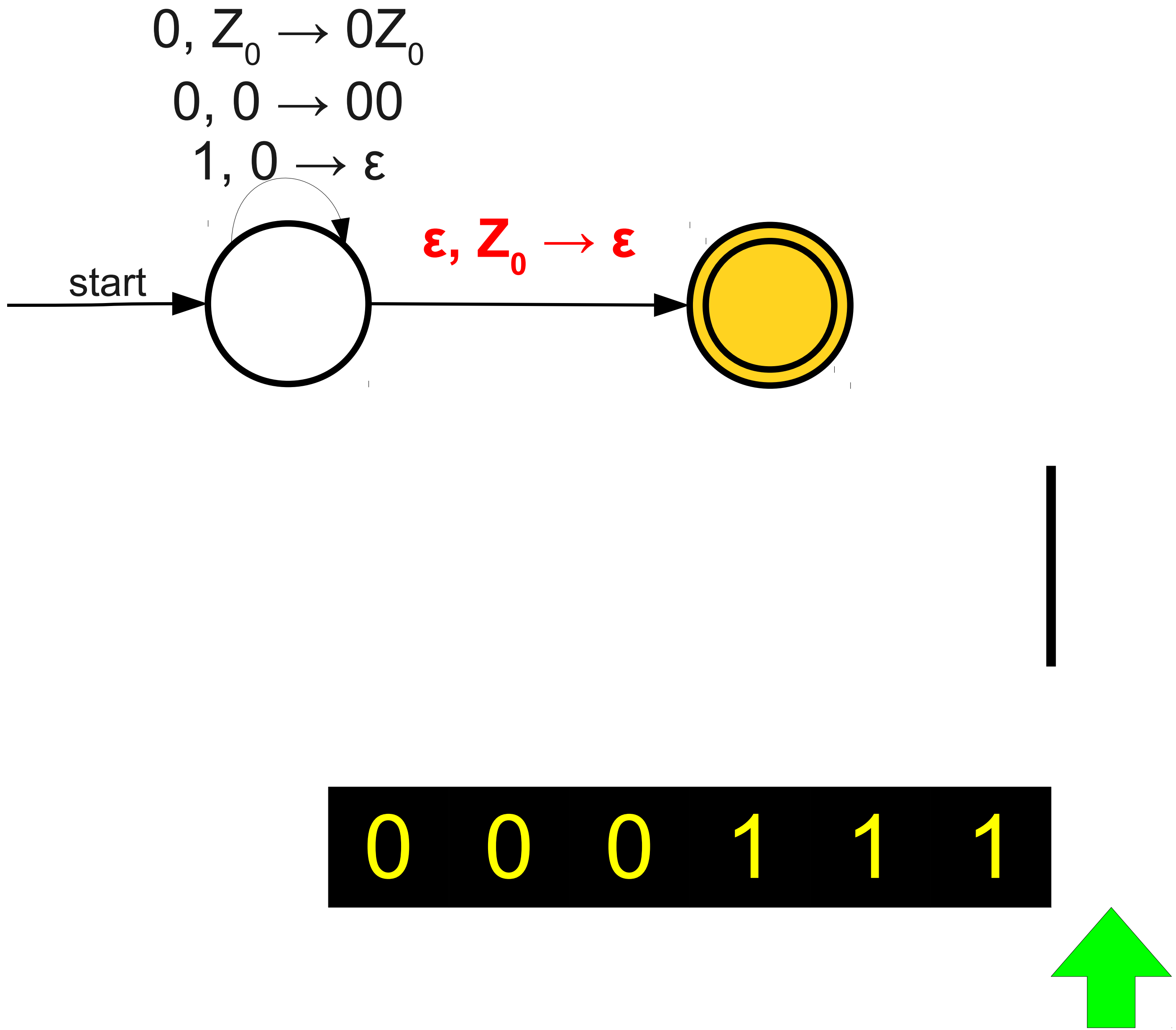
A Simple Pushdown Automaton



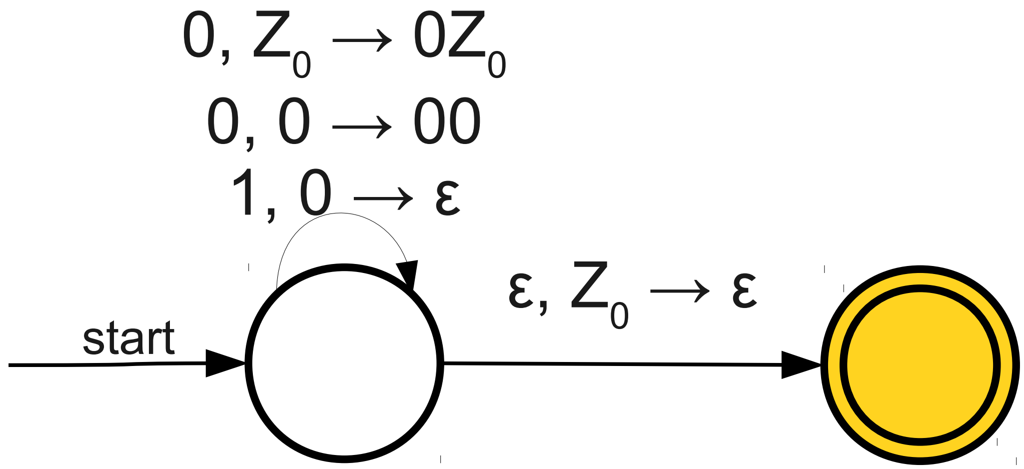
A Simple Pushdown Automaton



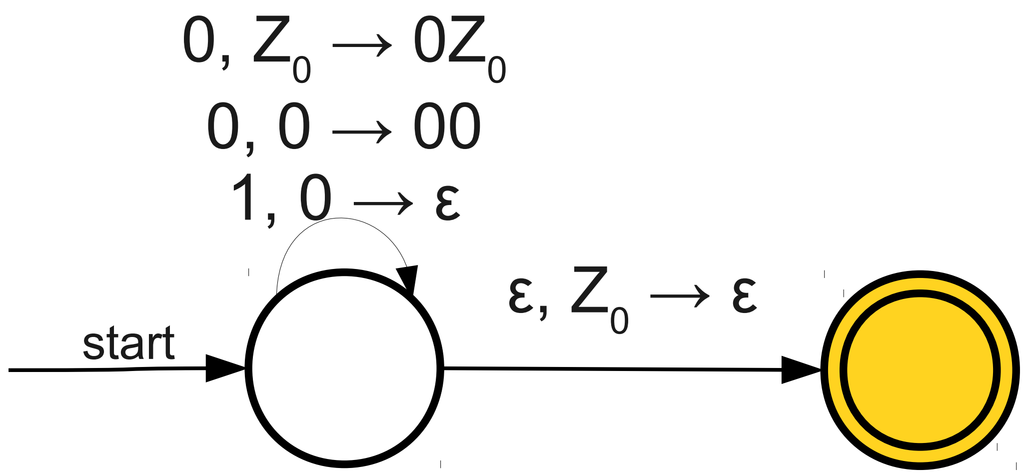
A Simple Pushdown Automaton



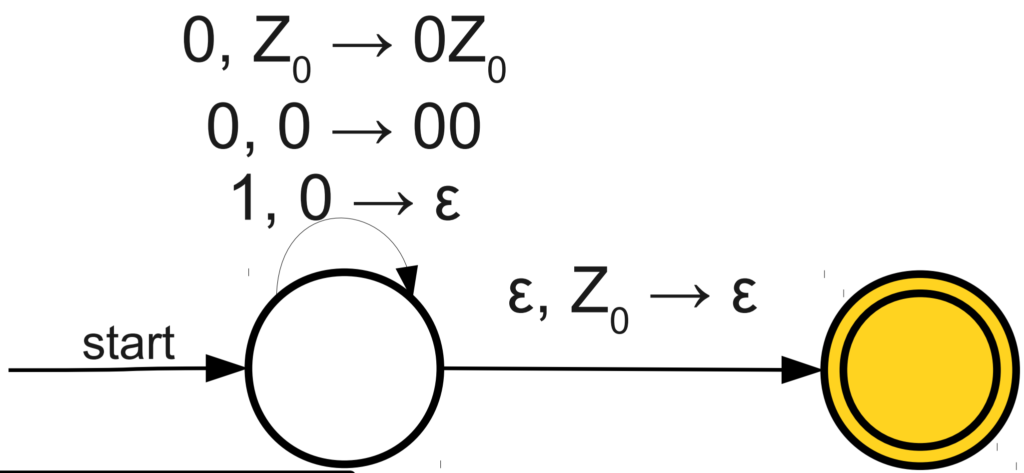
A Simple Pushdown Automaton



A Simple Pushdown Automaton

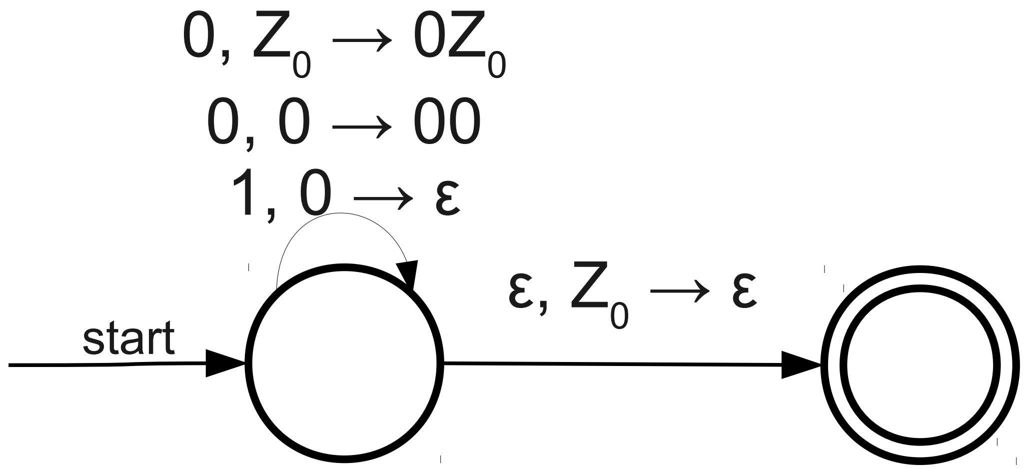


A Simple Pushdown Automaton

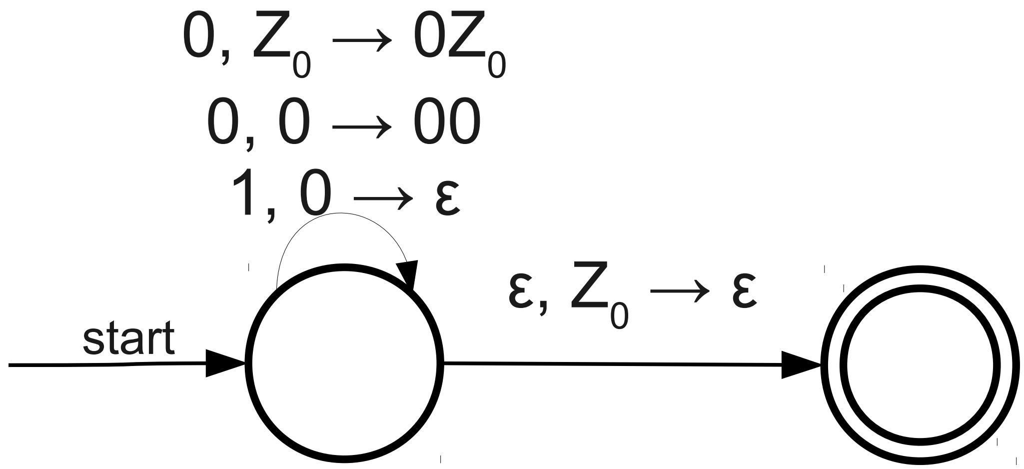


0 0 0 1 1 1

A Simple Pushdown Automaton

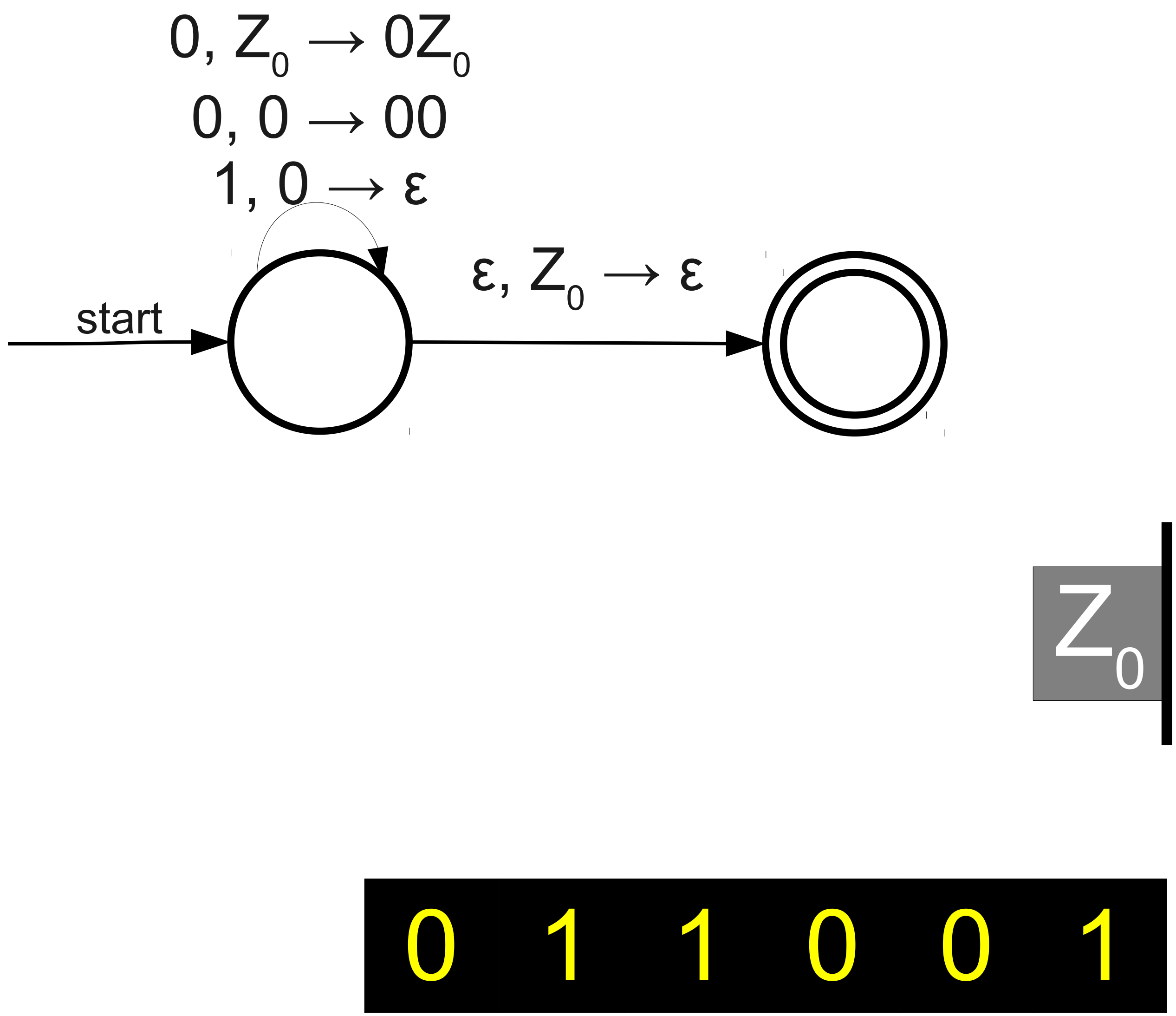


A Simple Pushdown Automaton

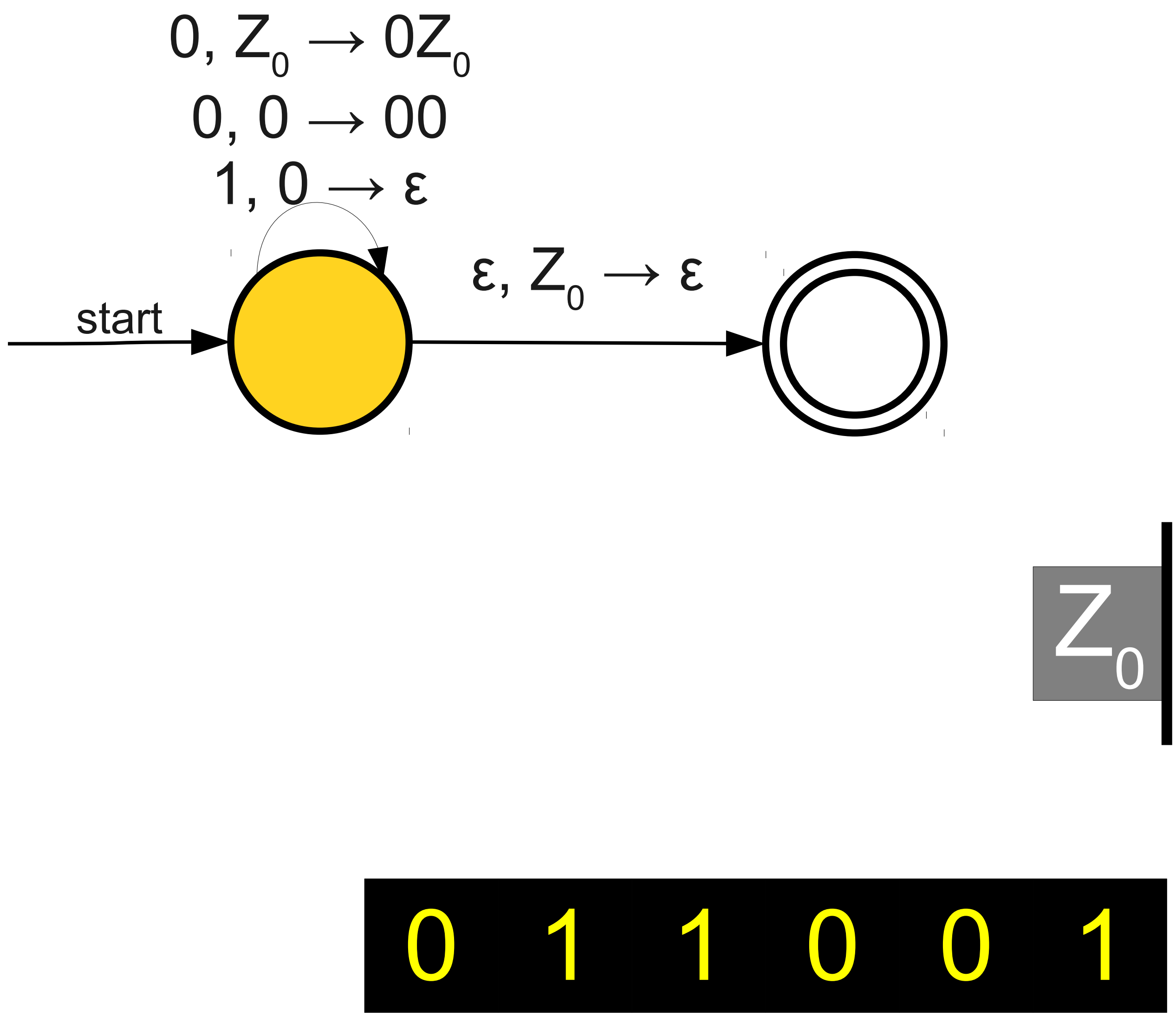


0 1 1 0 0 1

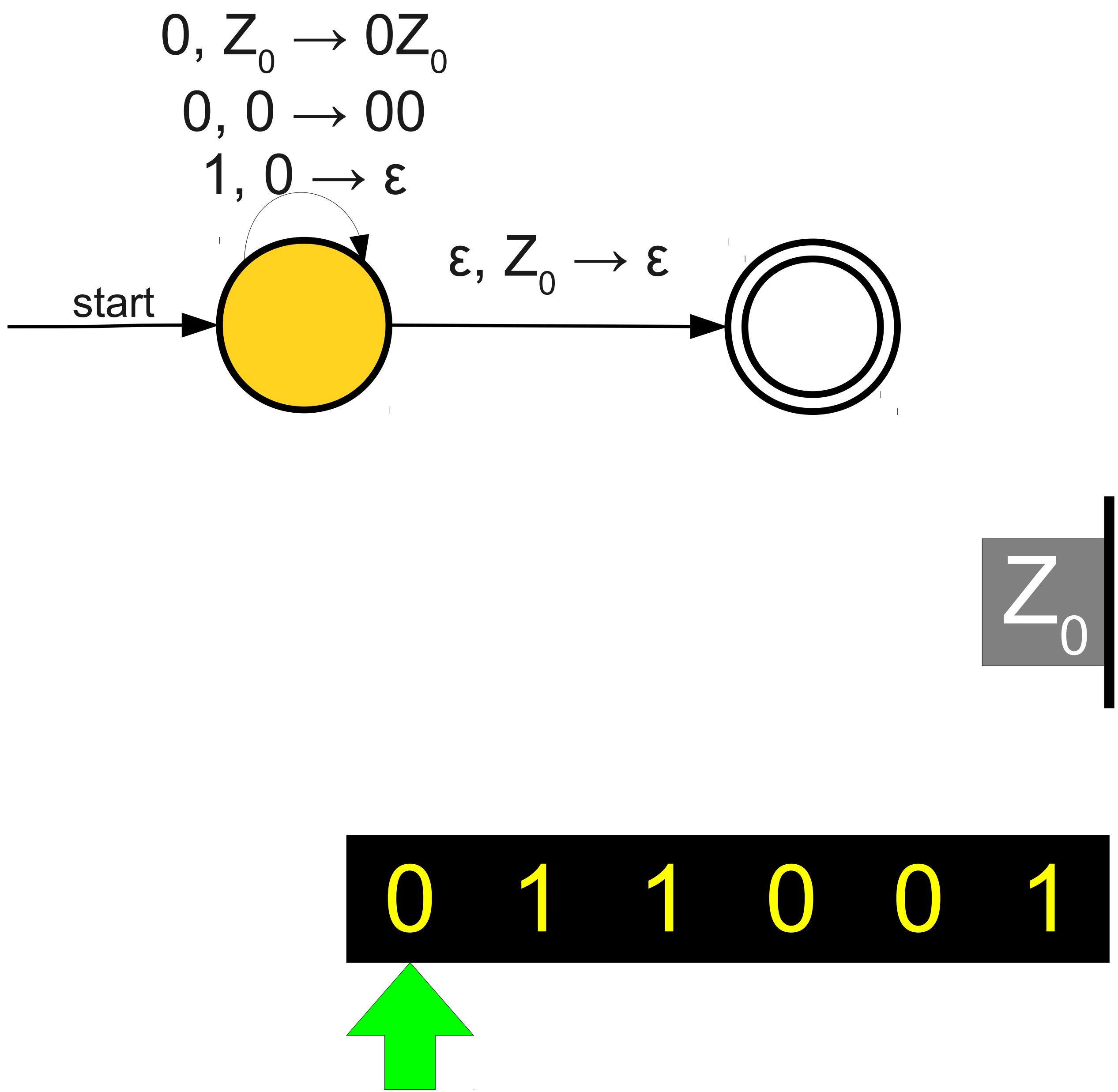
A Simple Pushdown Automaton



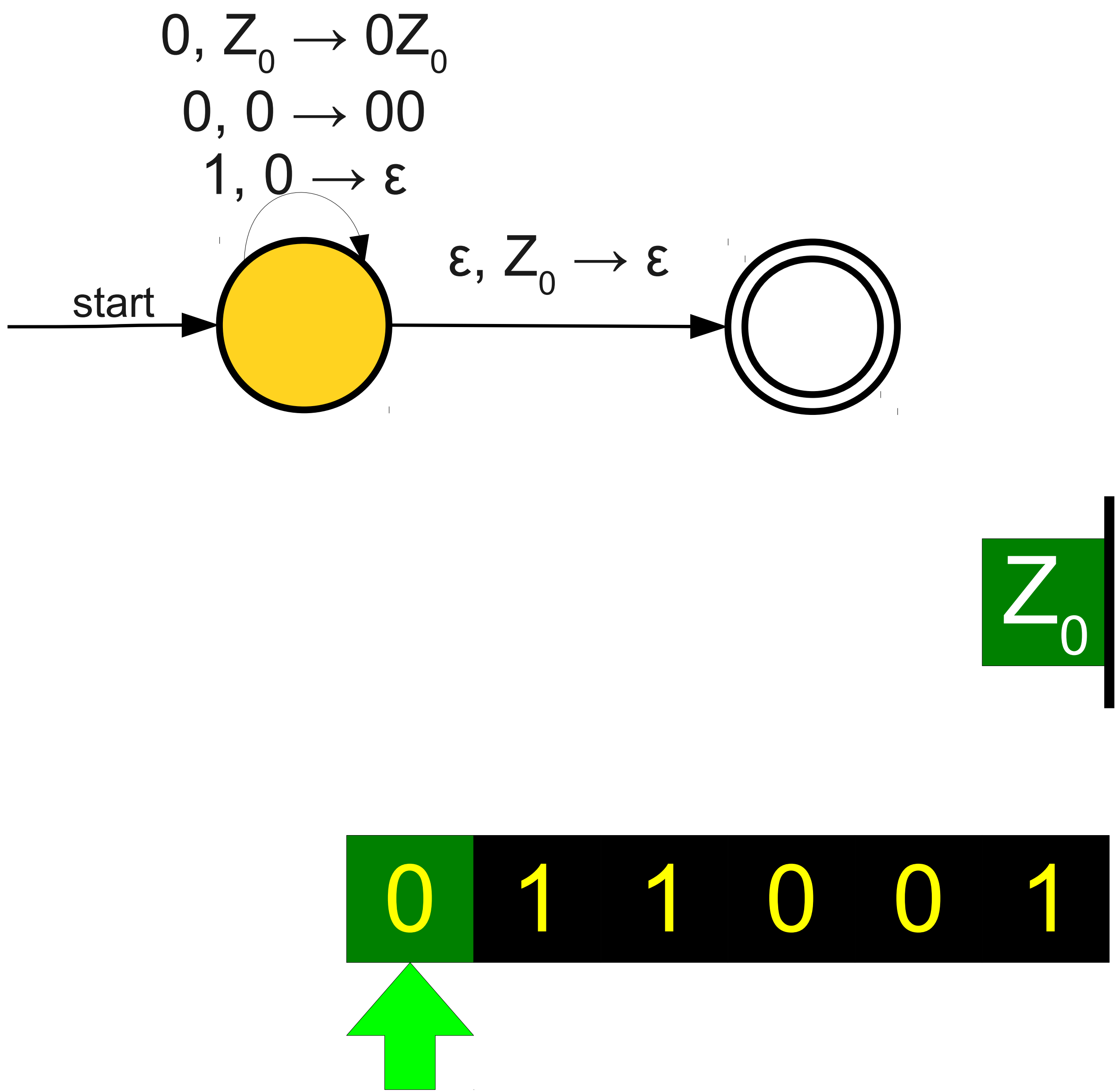
A Simple Pushdown Automaton



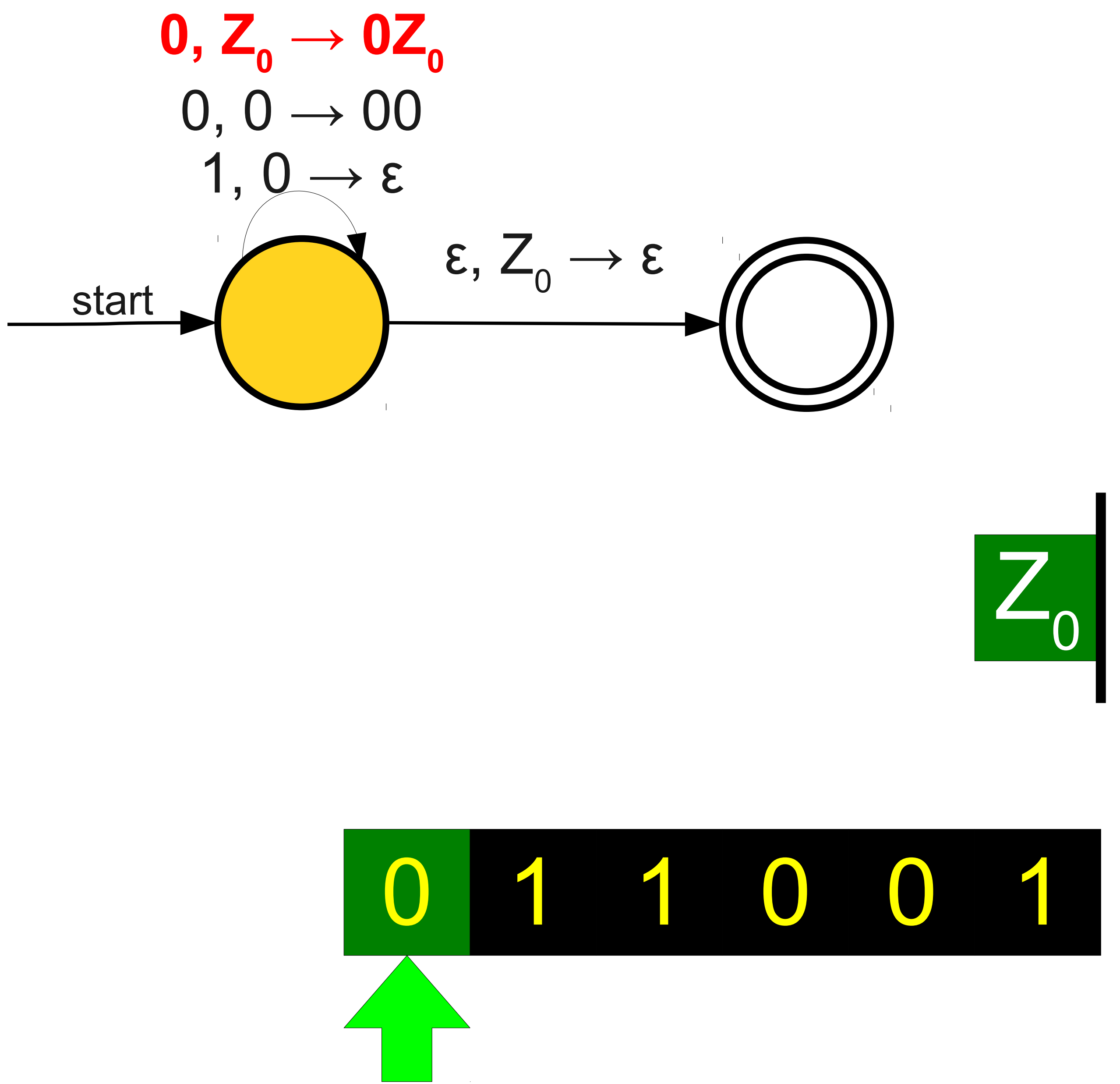
A Simple Pushdown Automaton



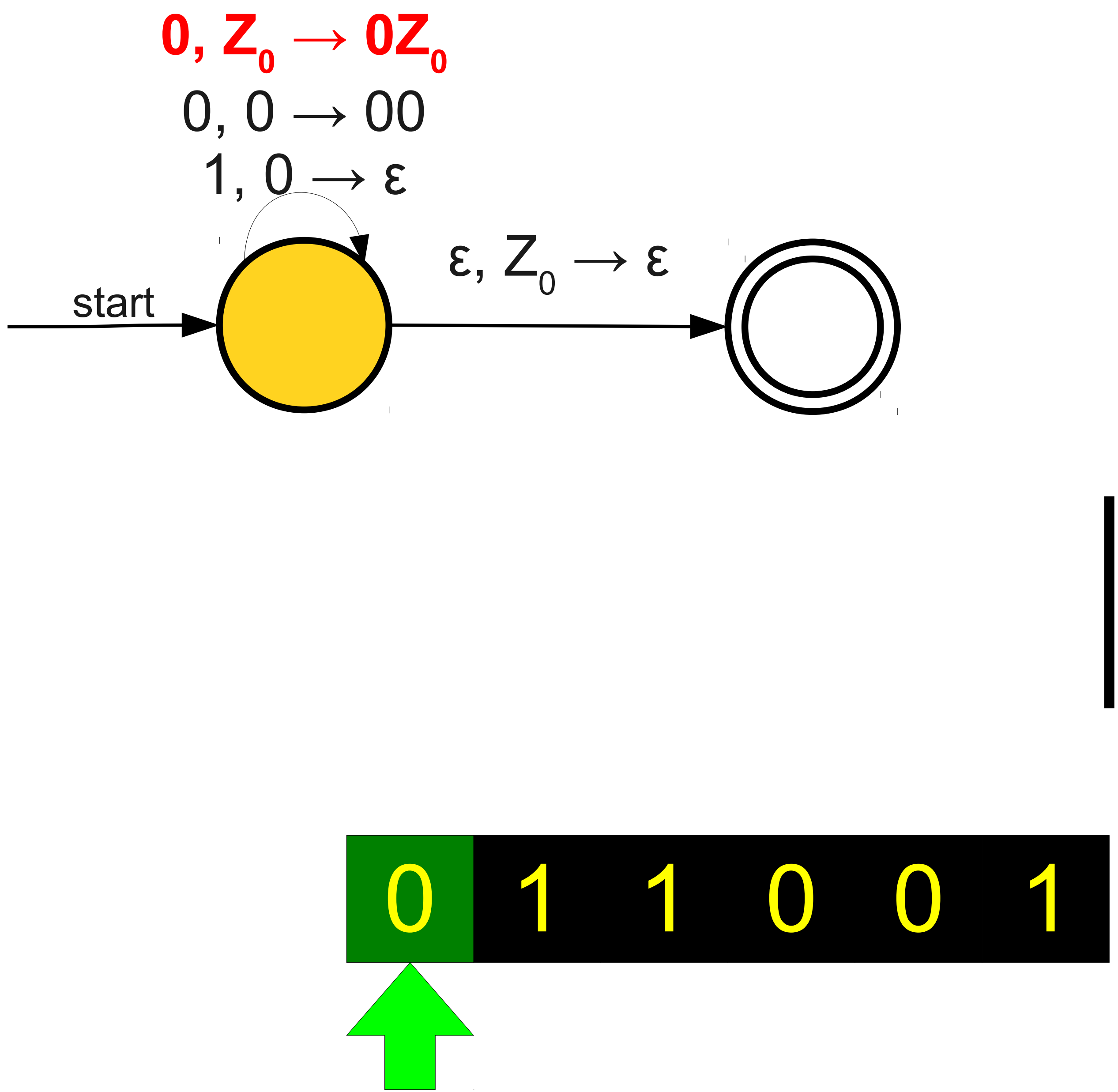
A Simple Pushdown Automaton



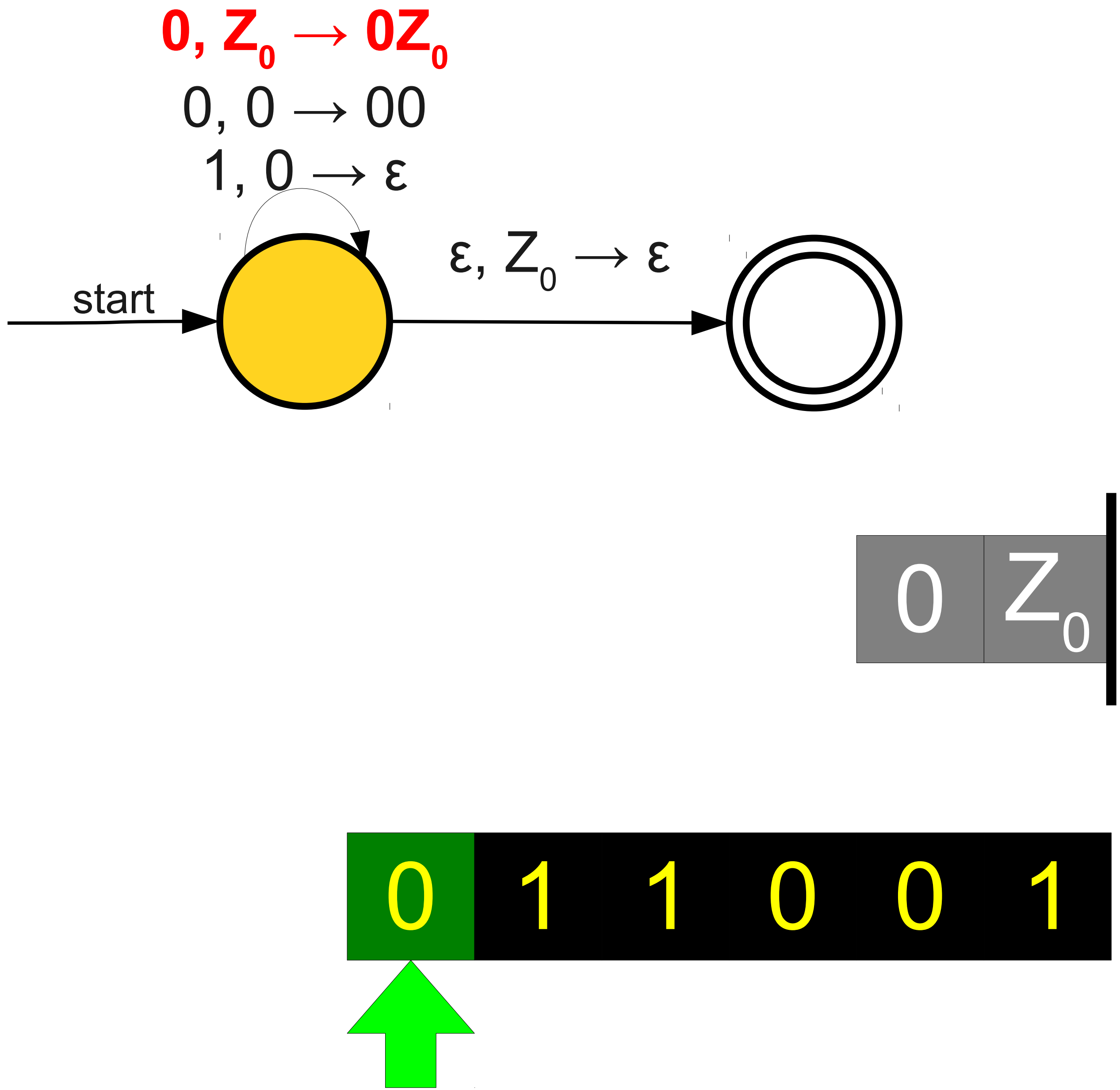
A Simple Pushdown Automaton



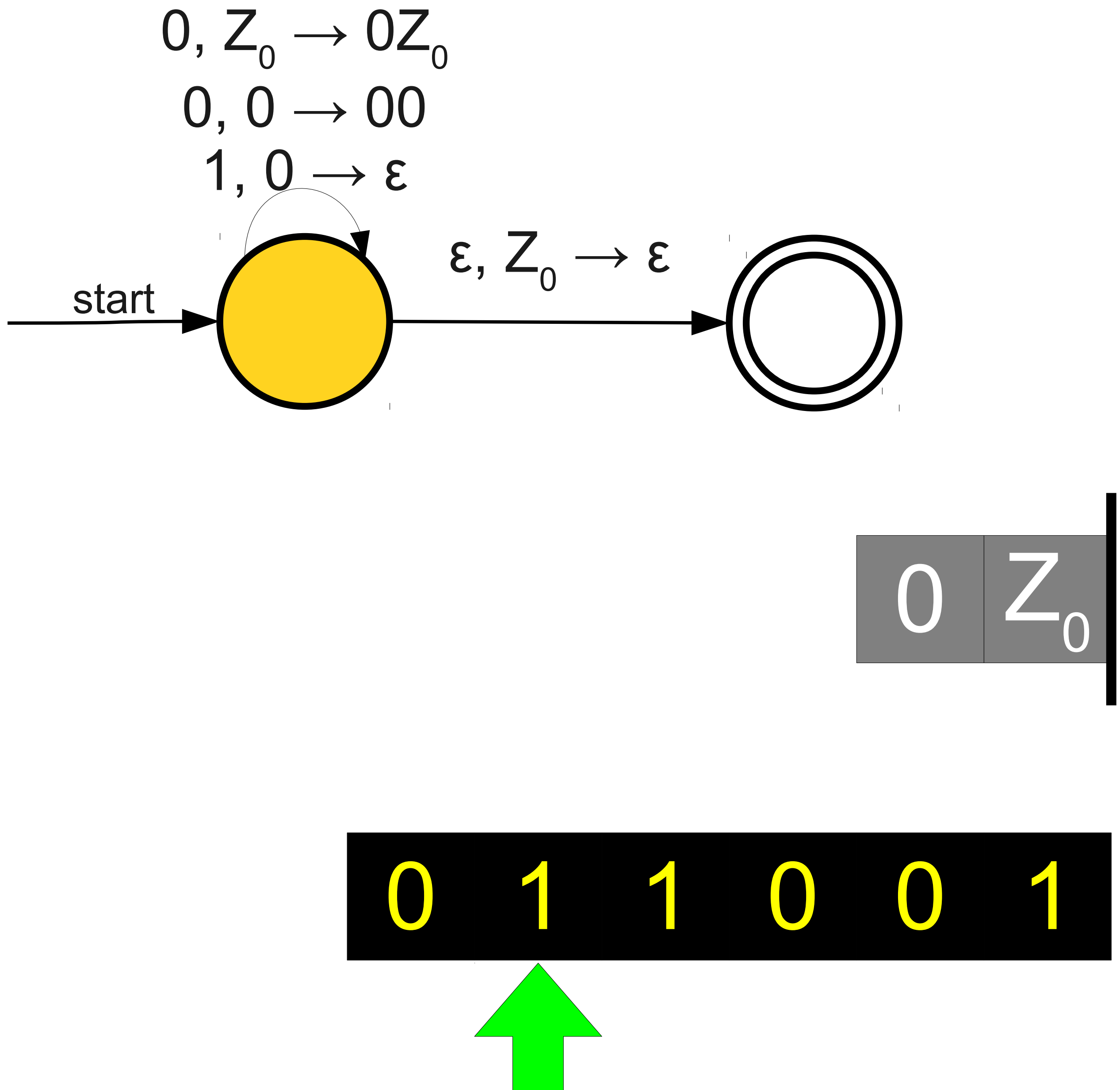
A Simple Pushdown Automaton



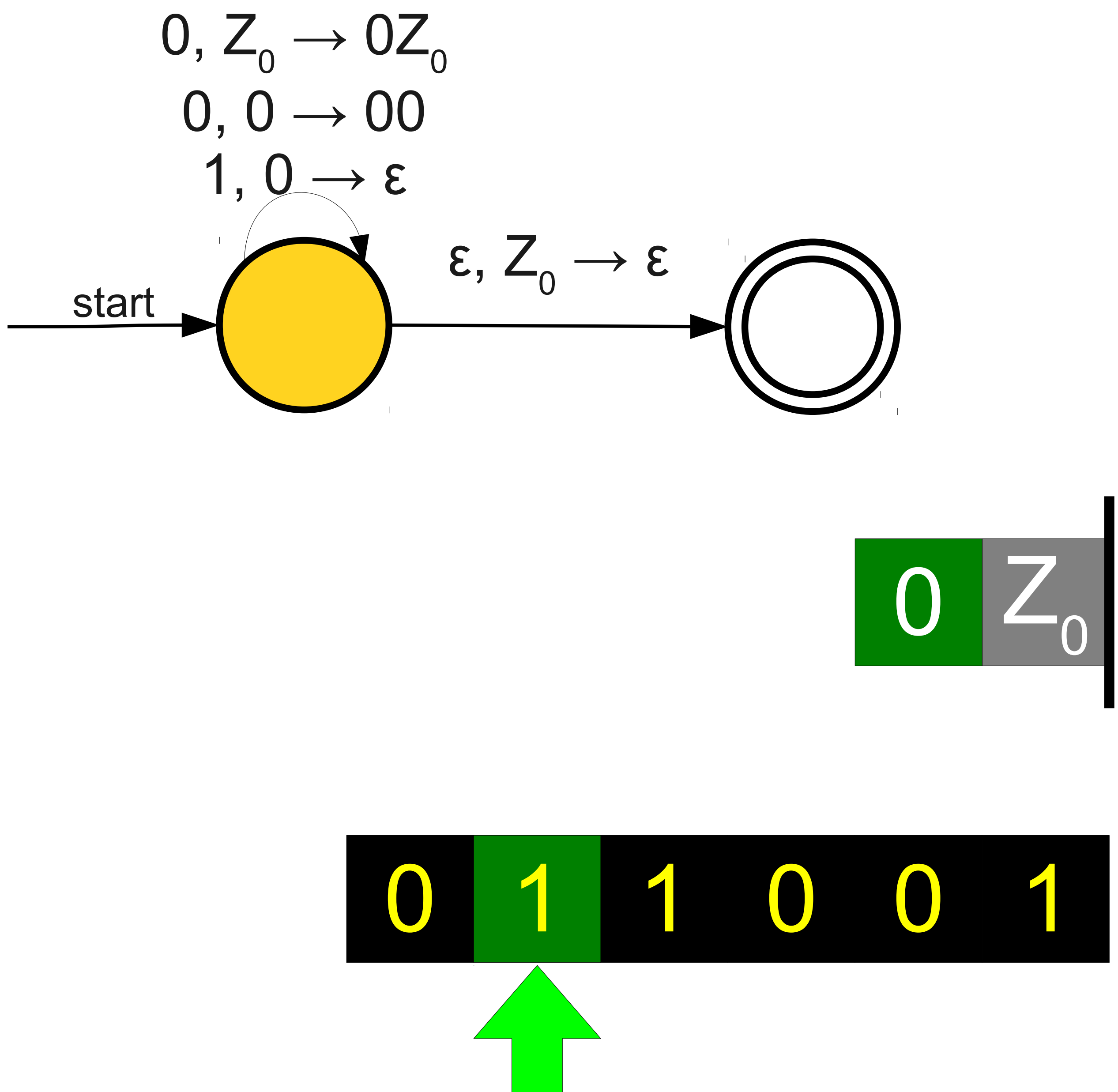
A Simple Pushdown Automaton



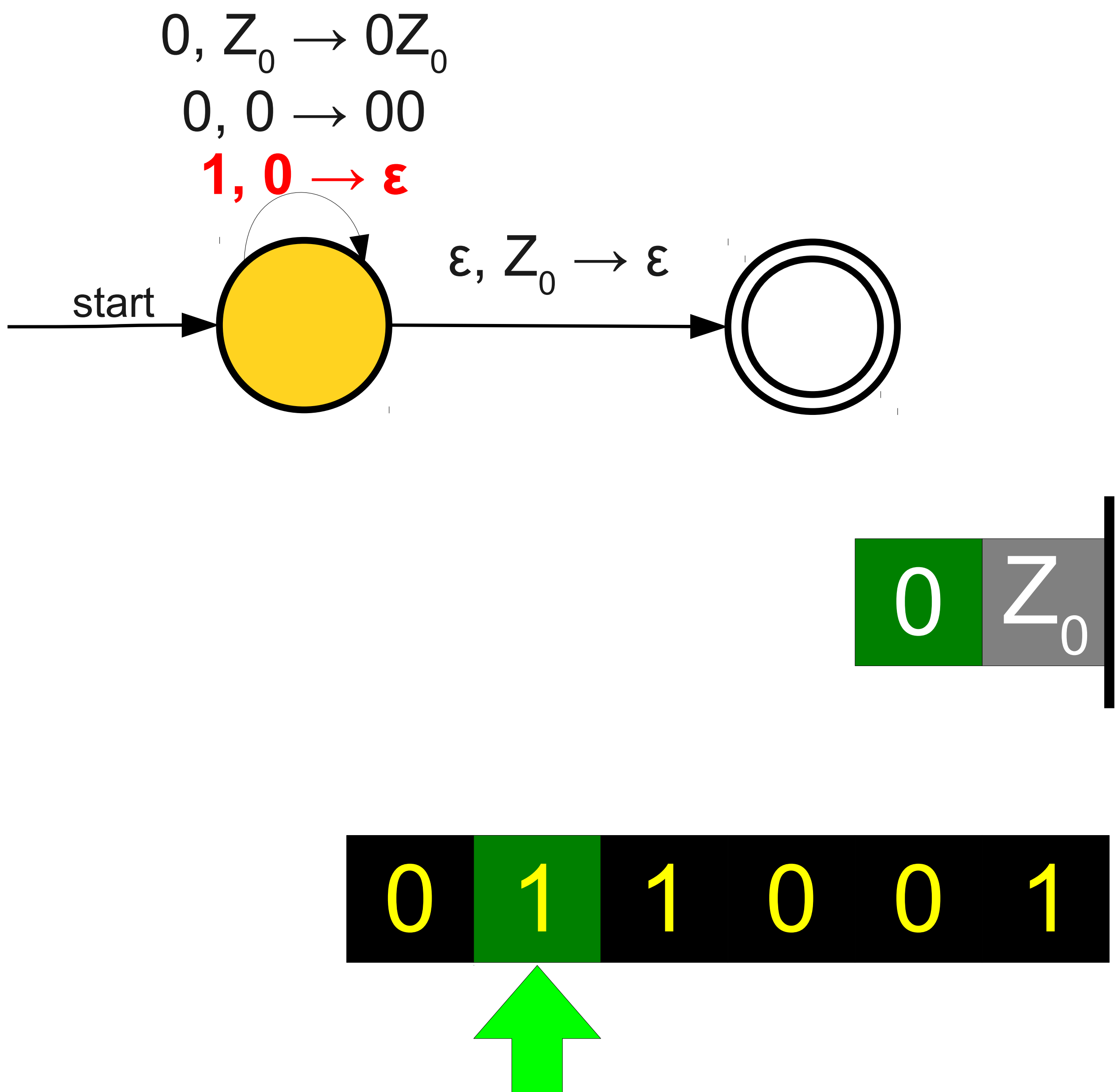
A Simple Pushdown Automaton



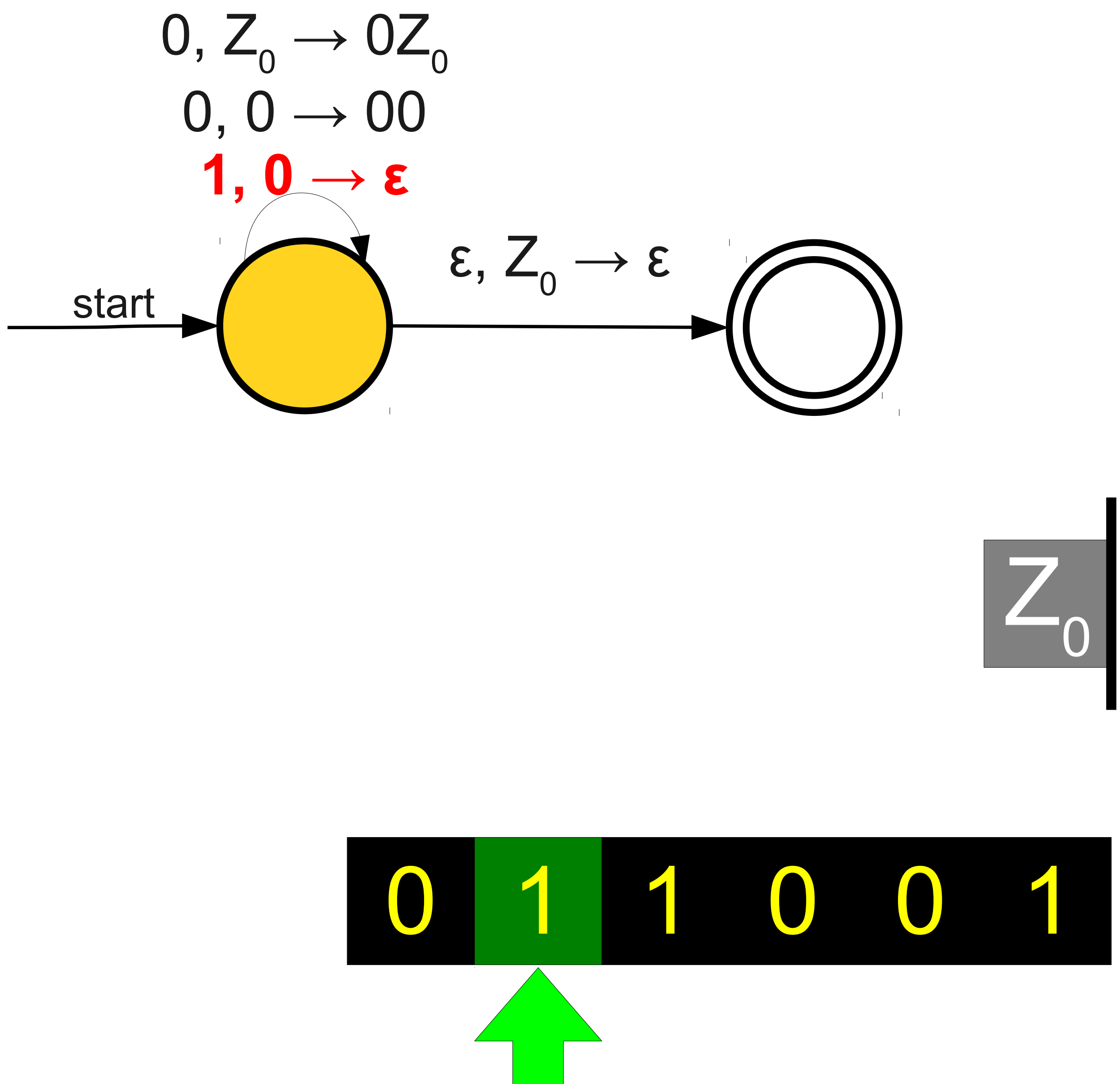
A Simple Pushdown Automaton



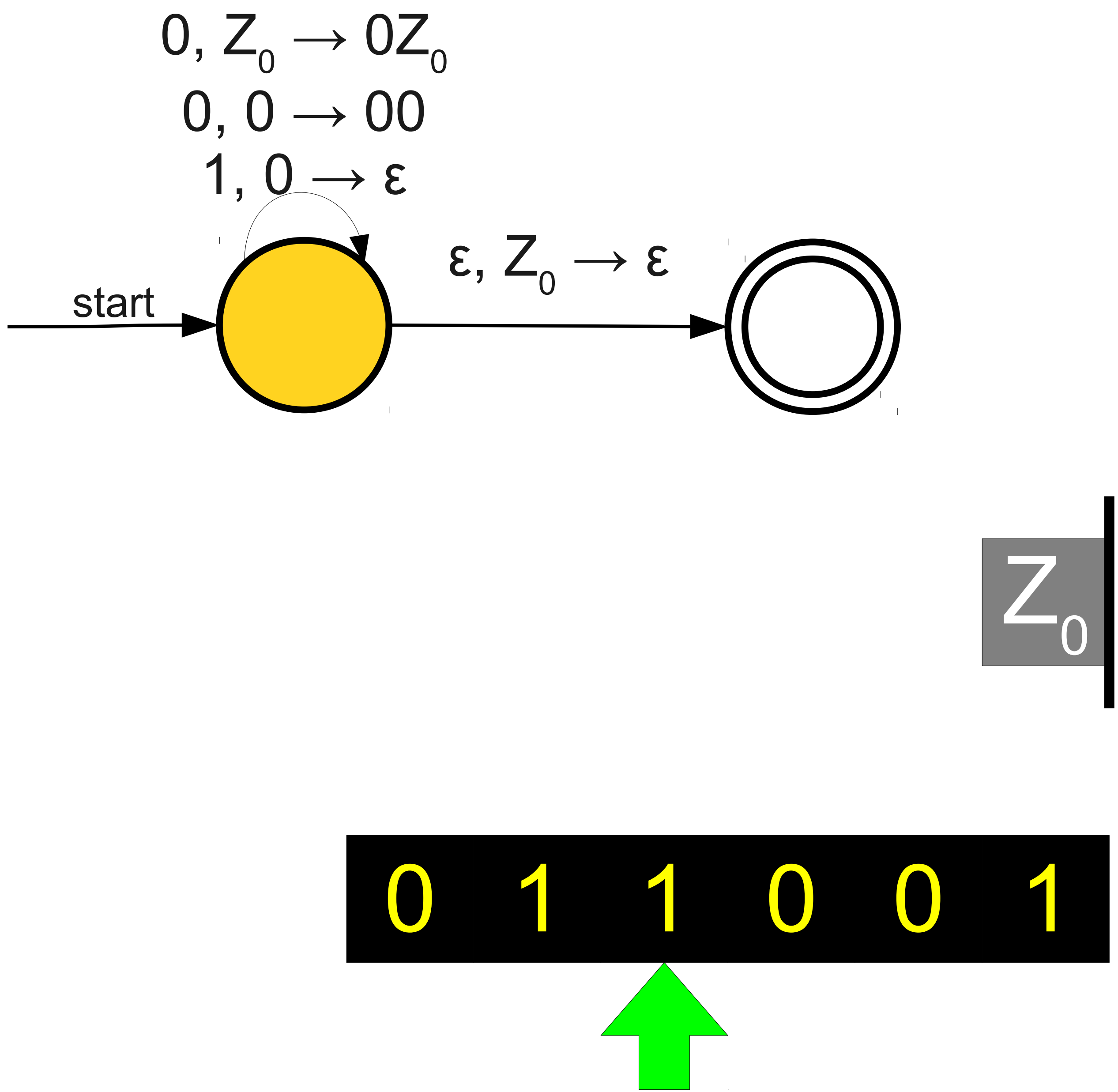
A Simple Pushdown Automaton



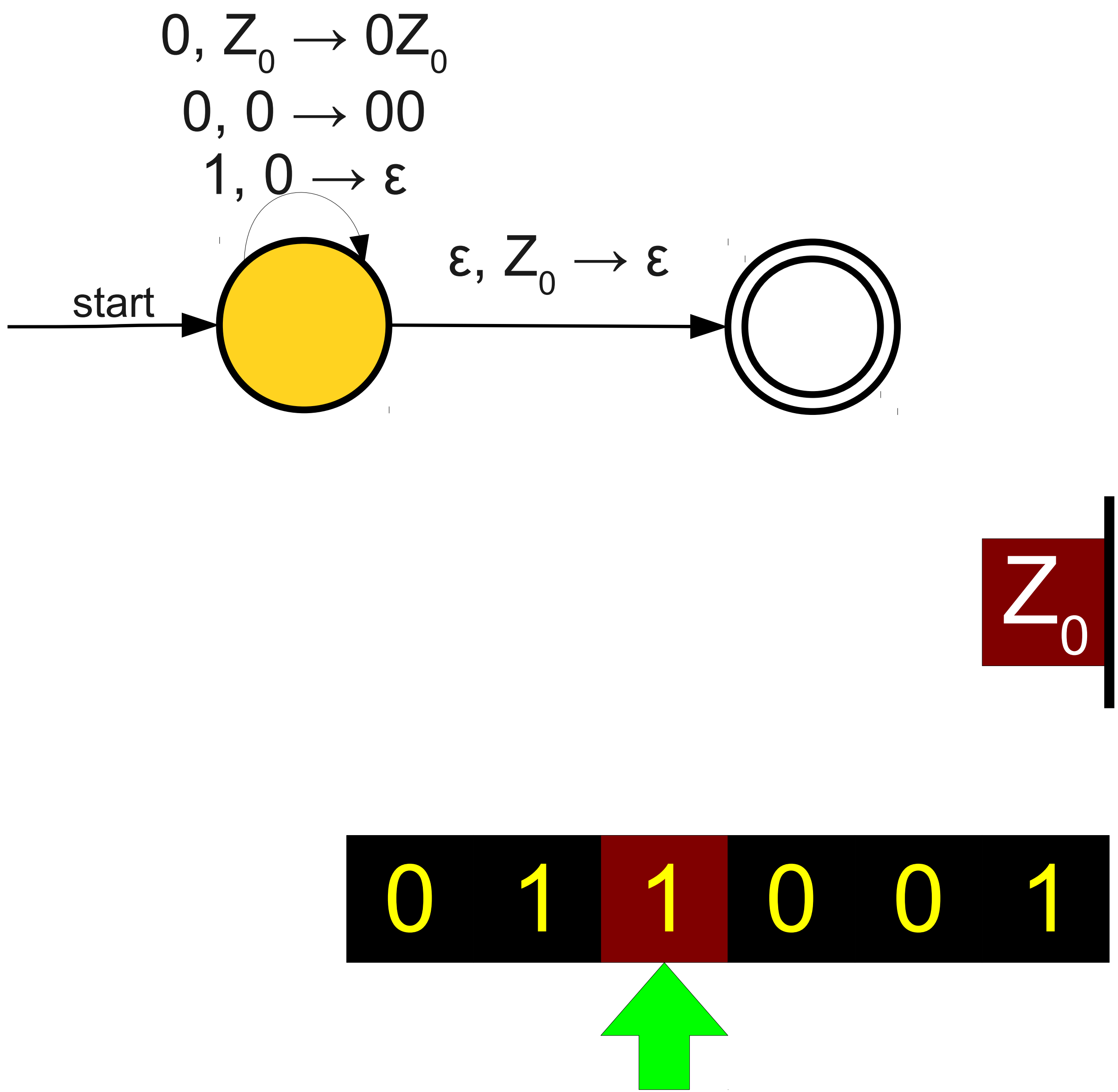
A Simple Pushdown Automaton



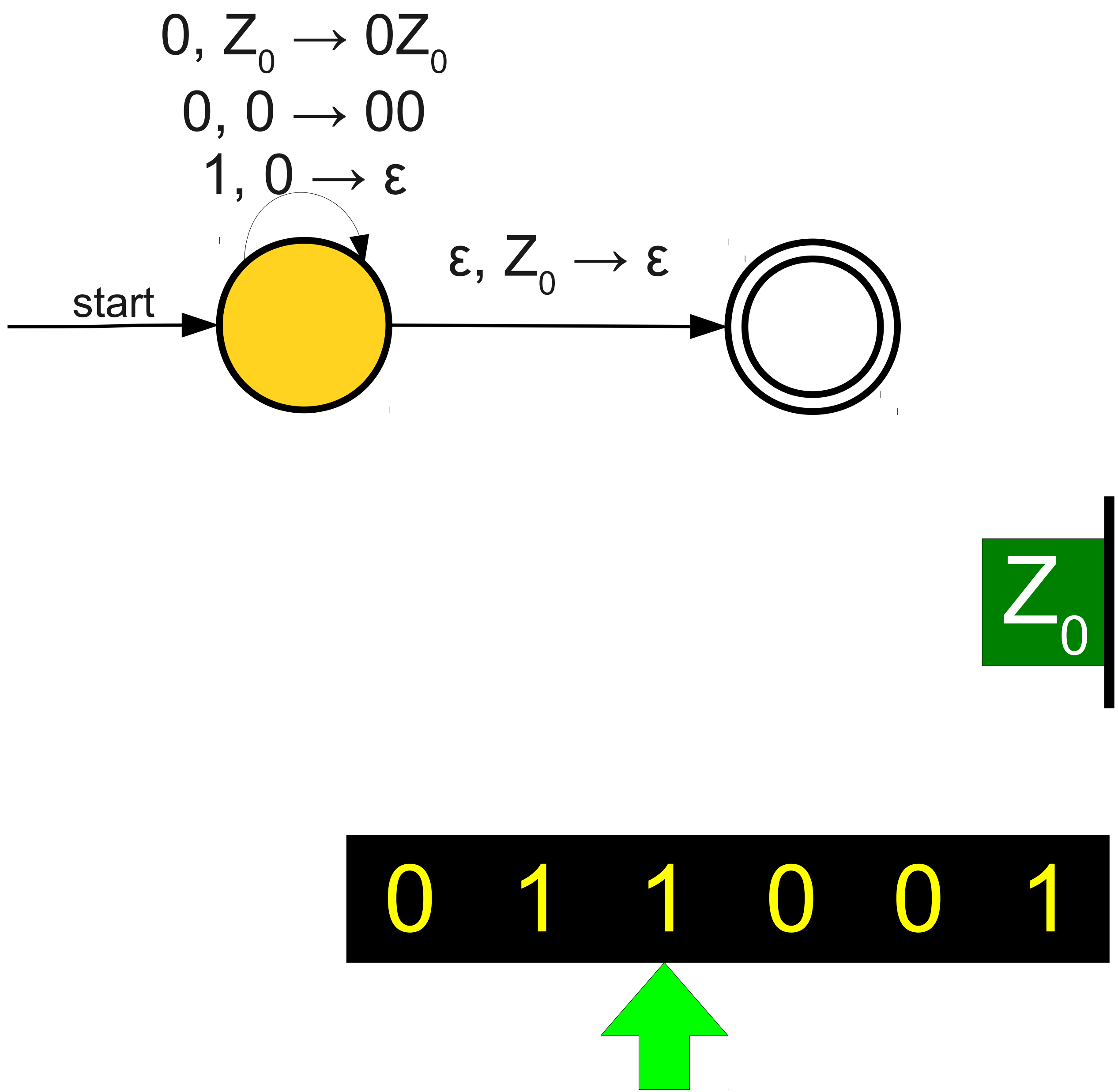
A Simple Pushdown Automaton



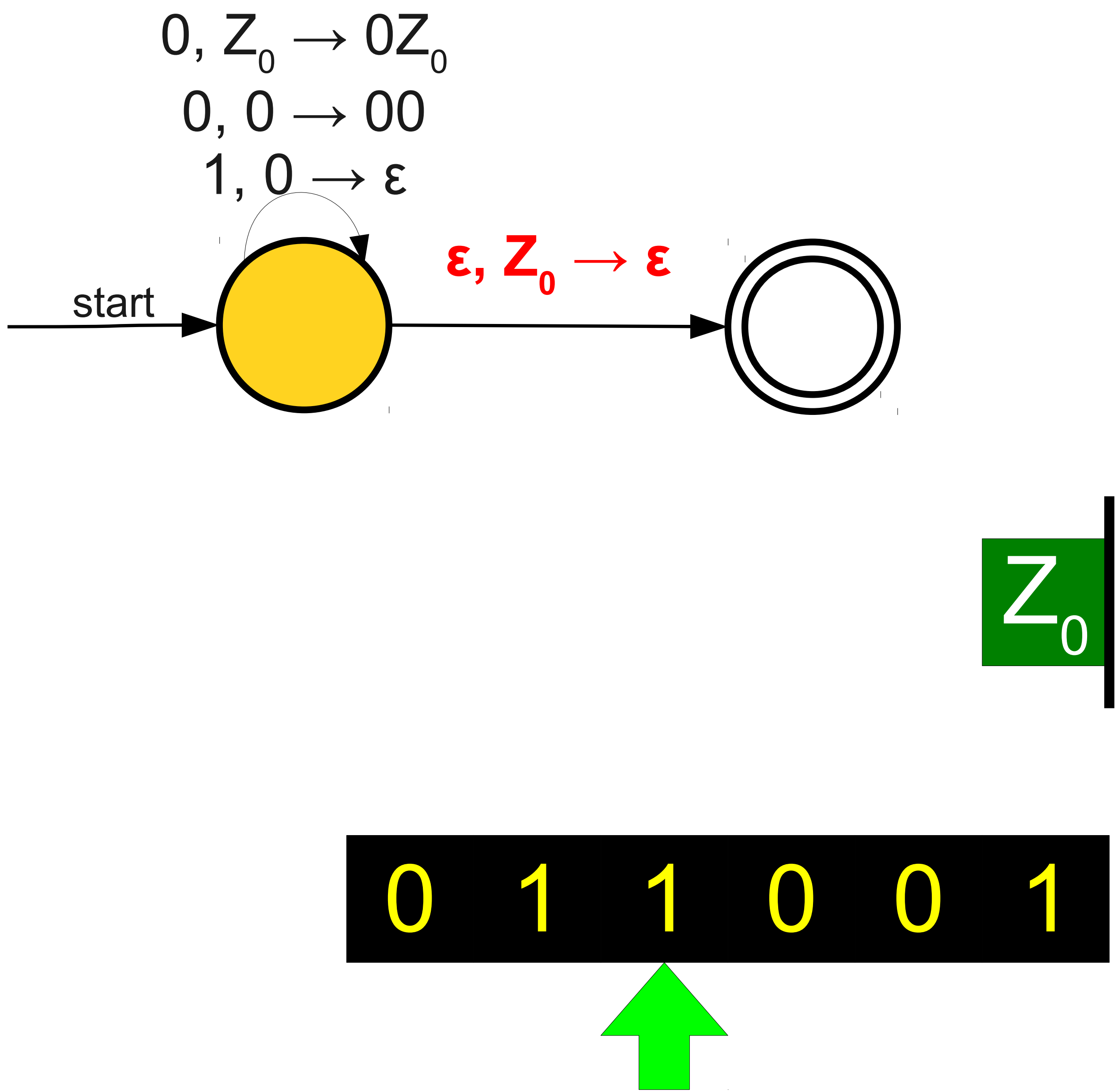
A Simple Pushdown Automaton



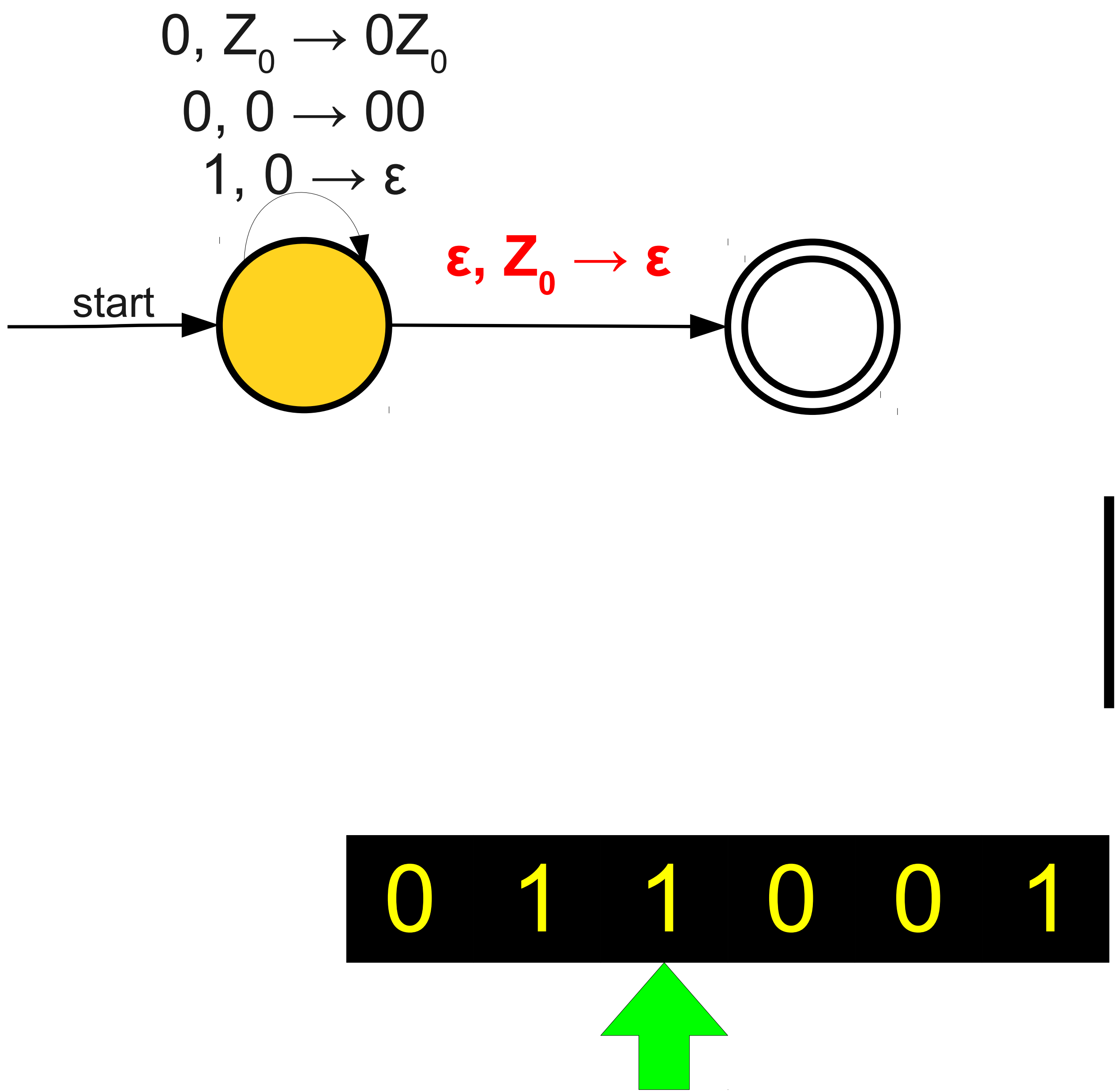
A Simple Pushdown Automaton



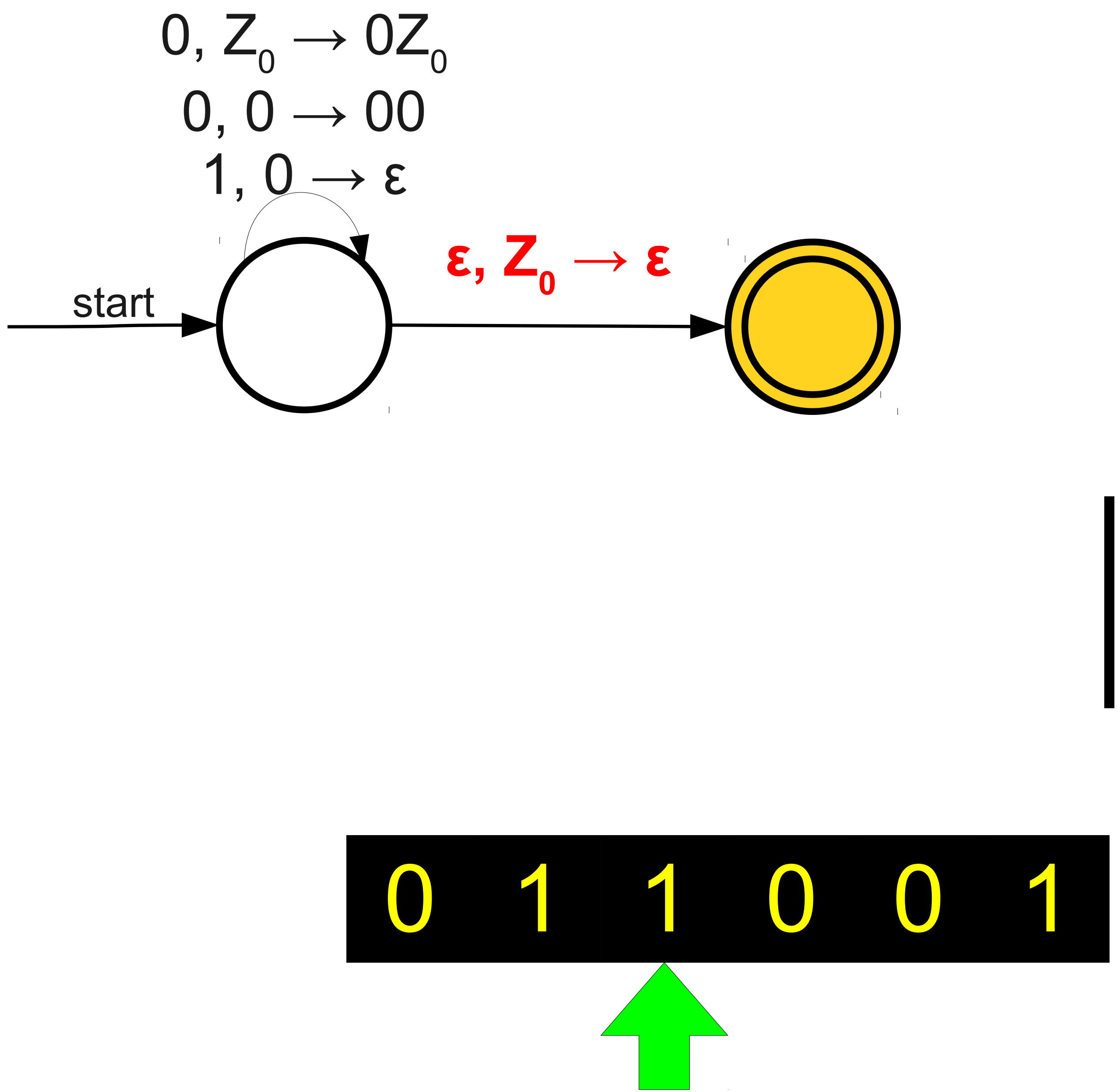
A Simple Pushdown Automaton



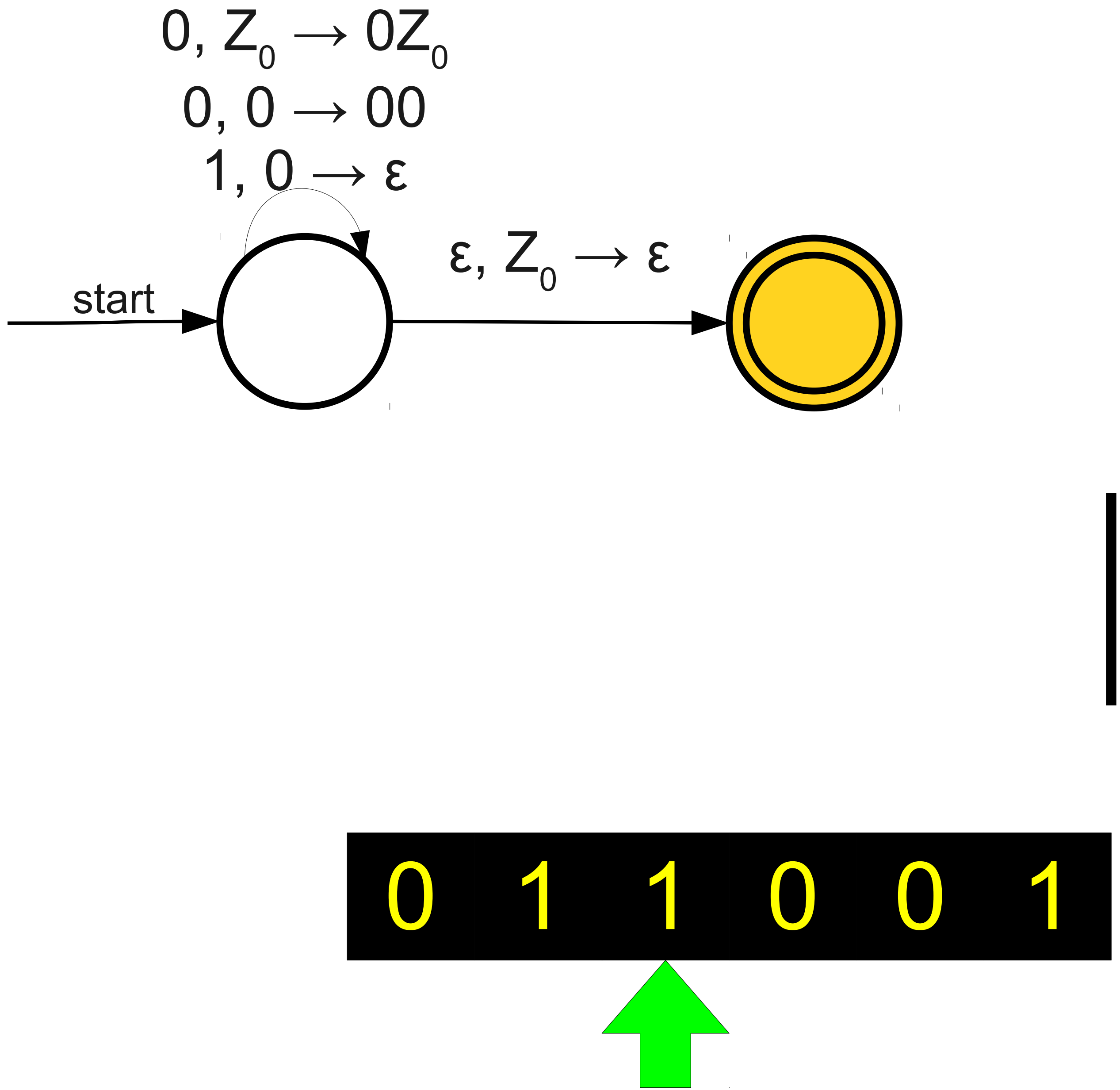
A Simple Pushdown Automaton



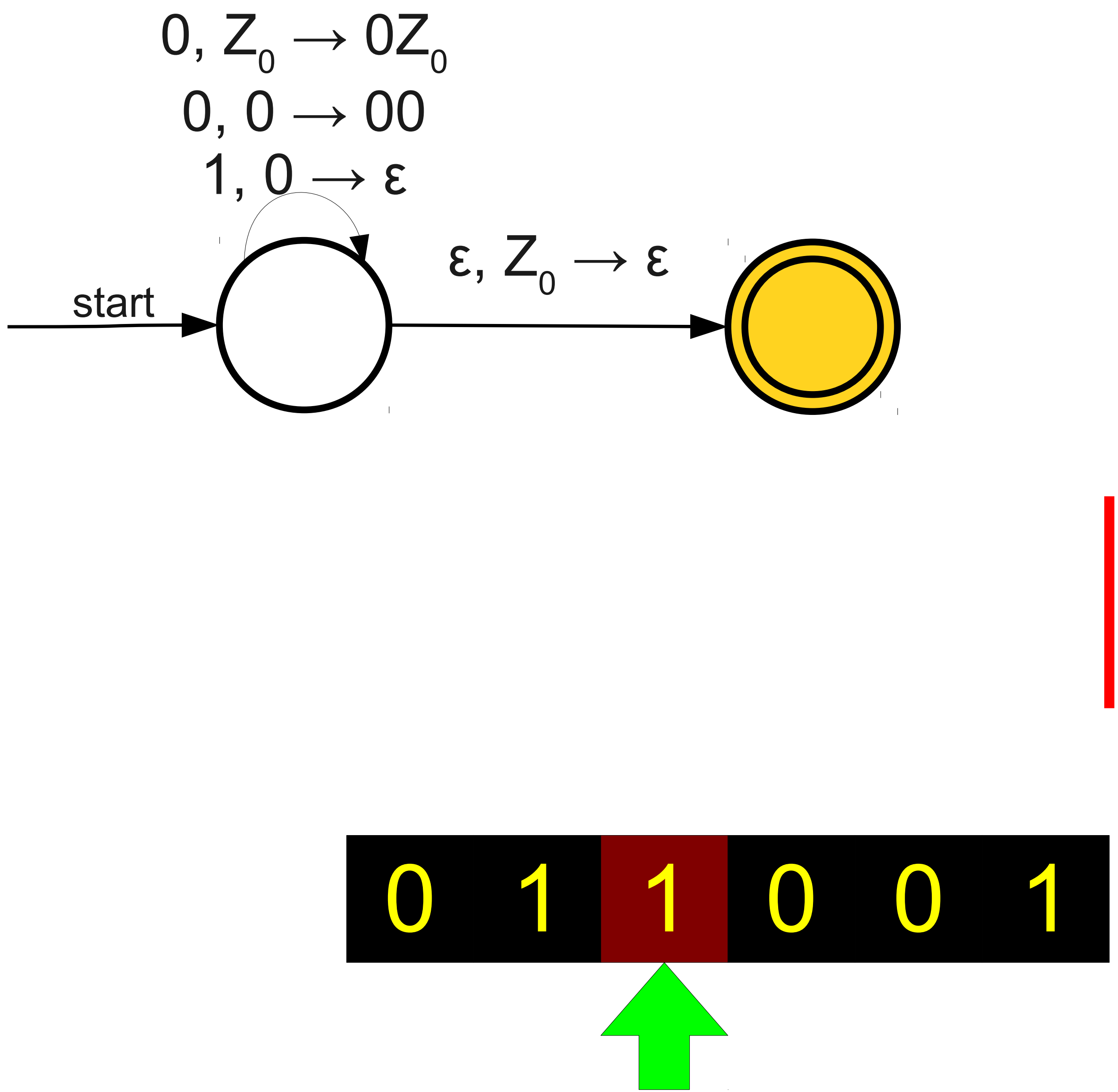
A Simple Pushdown Automaton



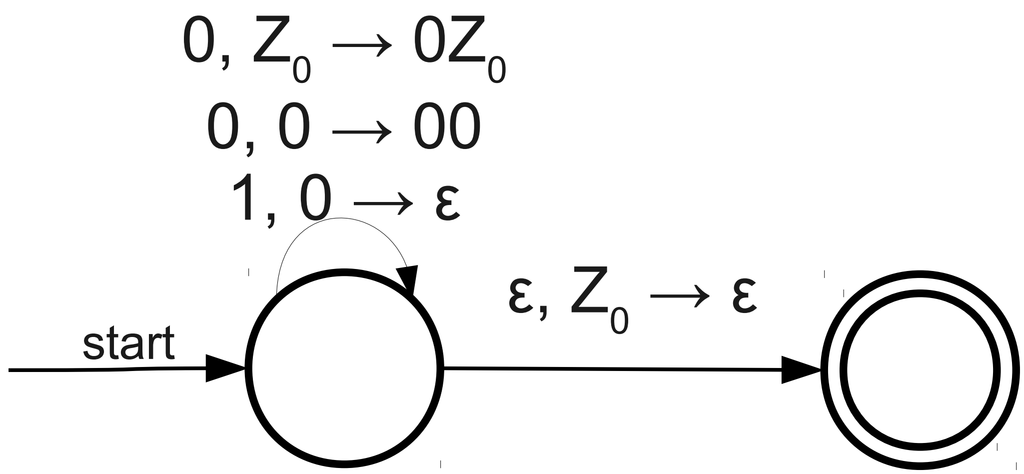
A Simple Pushdown Automaton

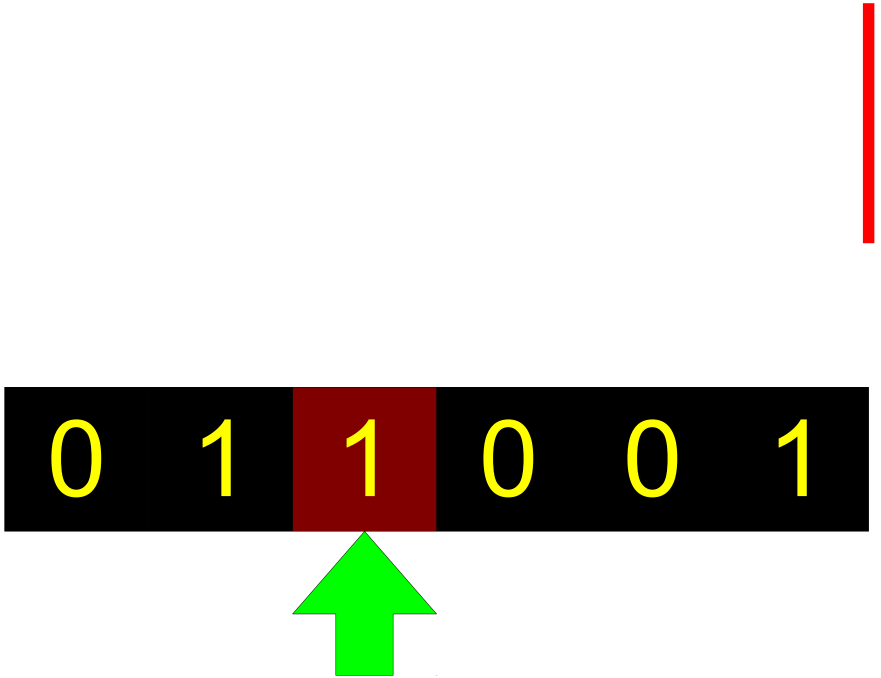


A Simple Pushdown Automaton

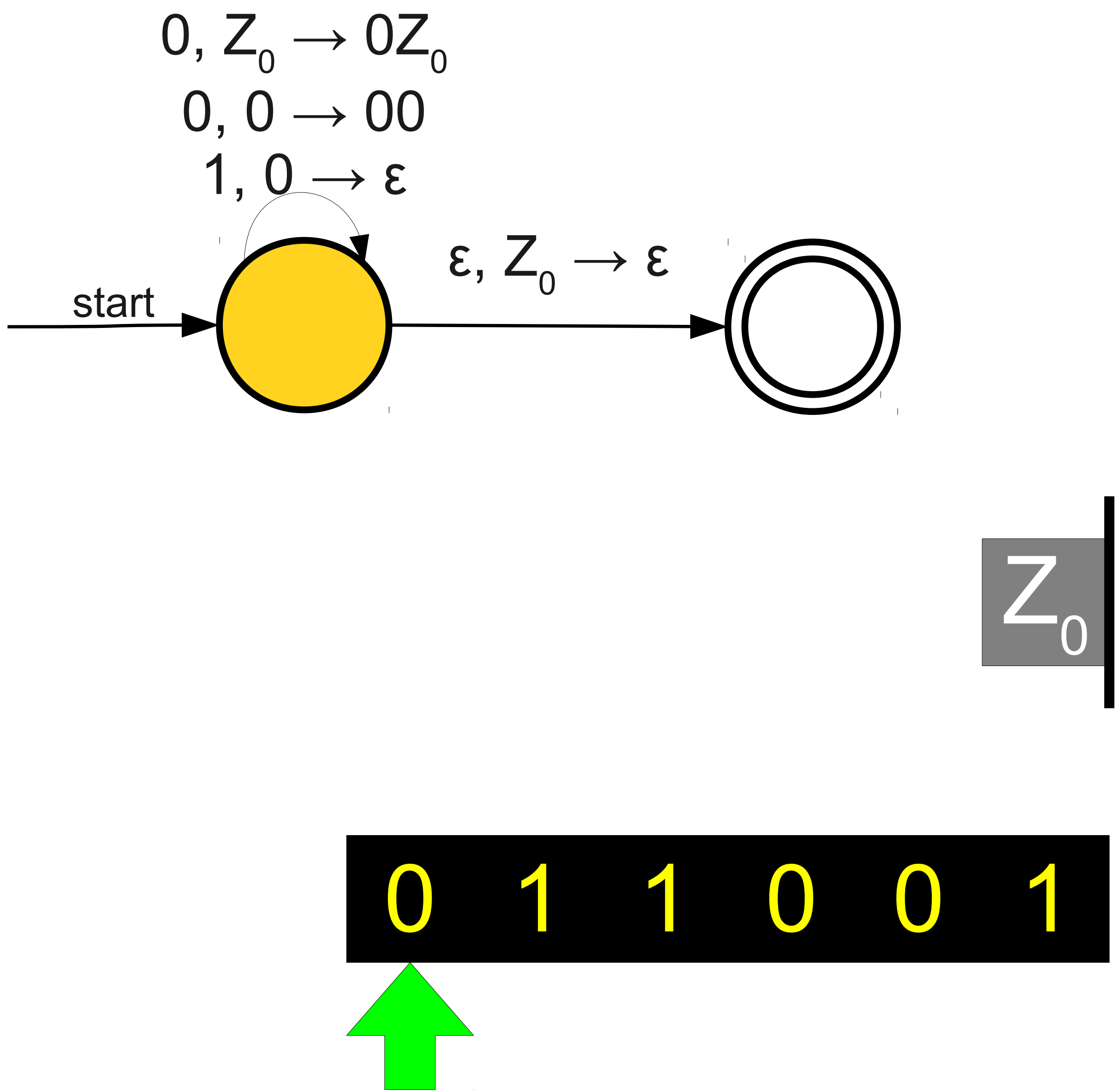


A Simple Pushdown Automaton

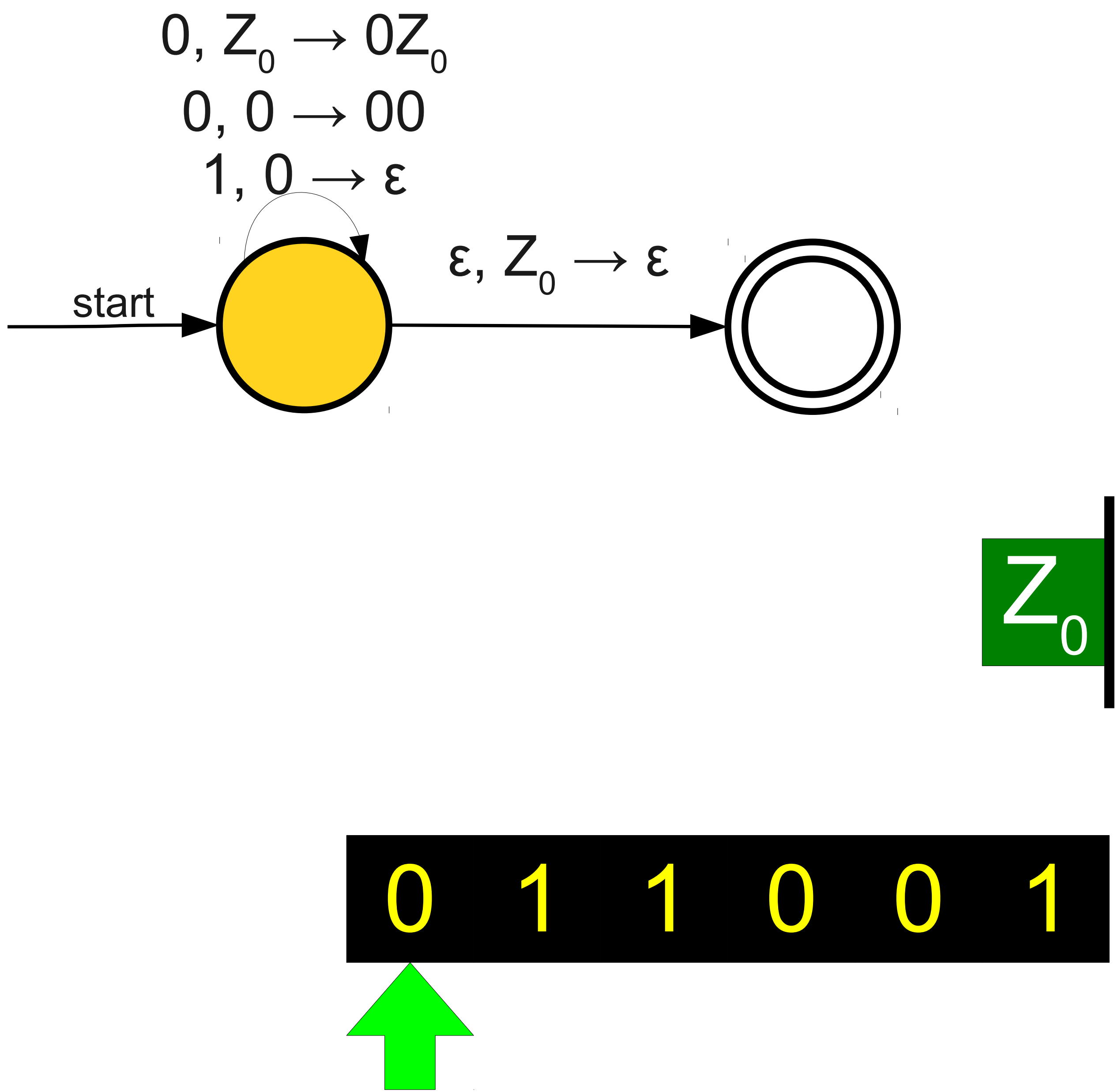




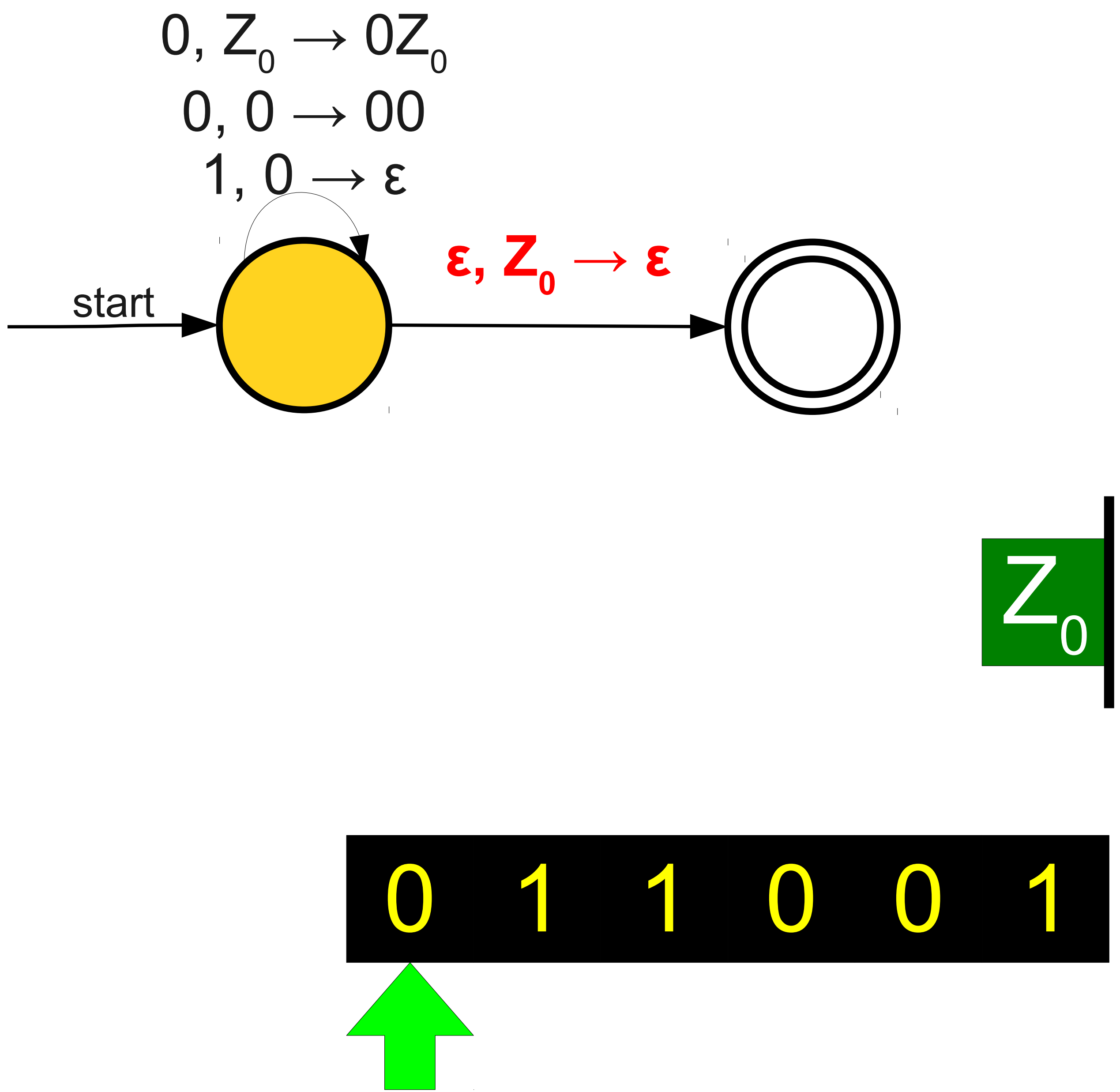
A Simple Pushdown Automaton



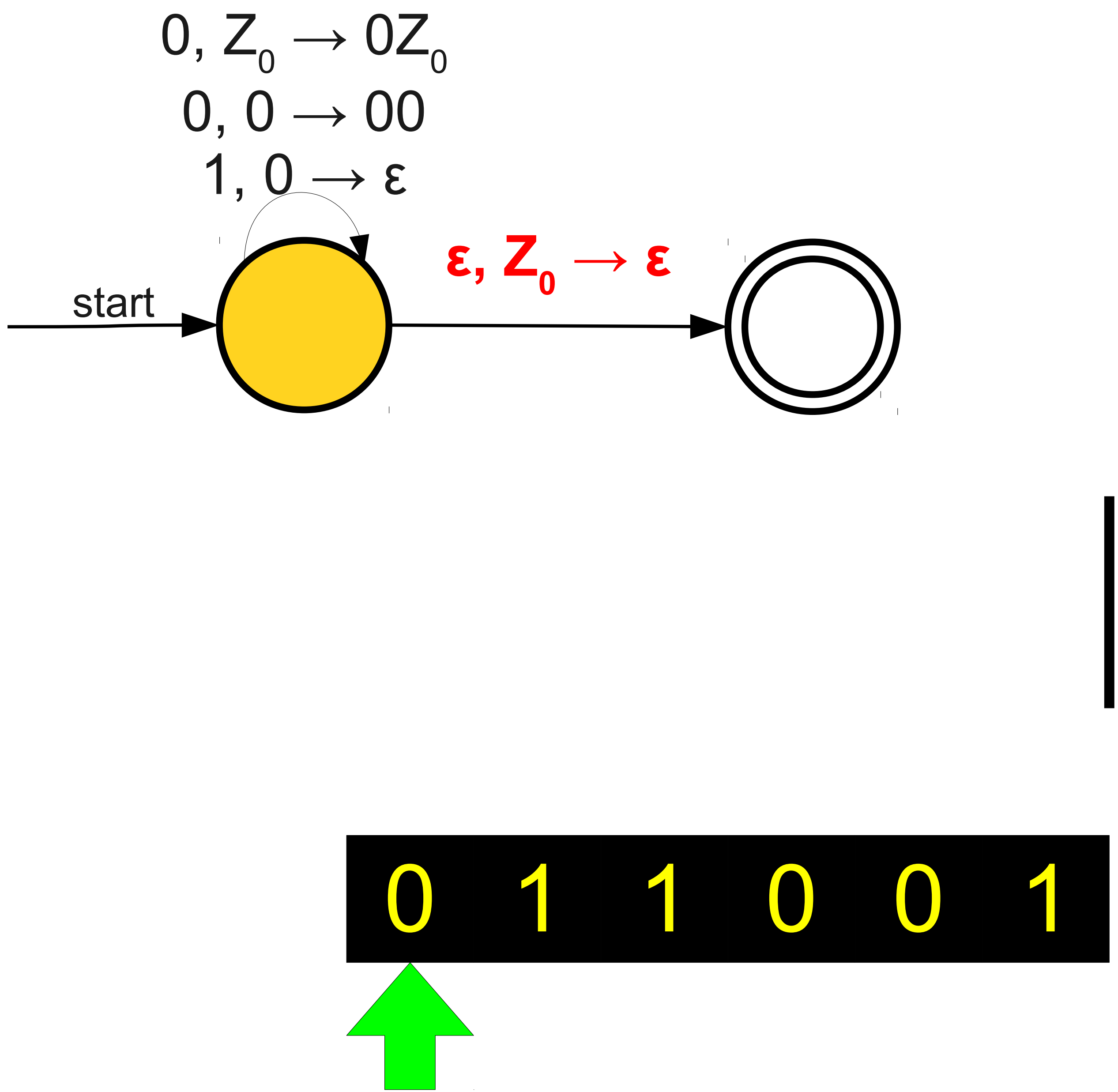
A Simple Pushdown Automaton



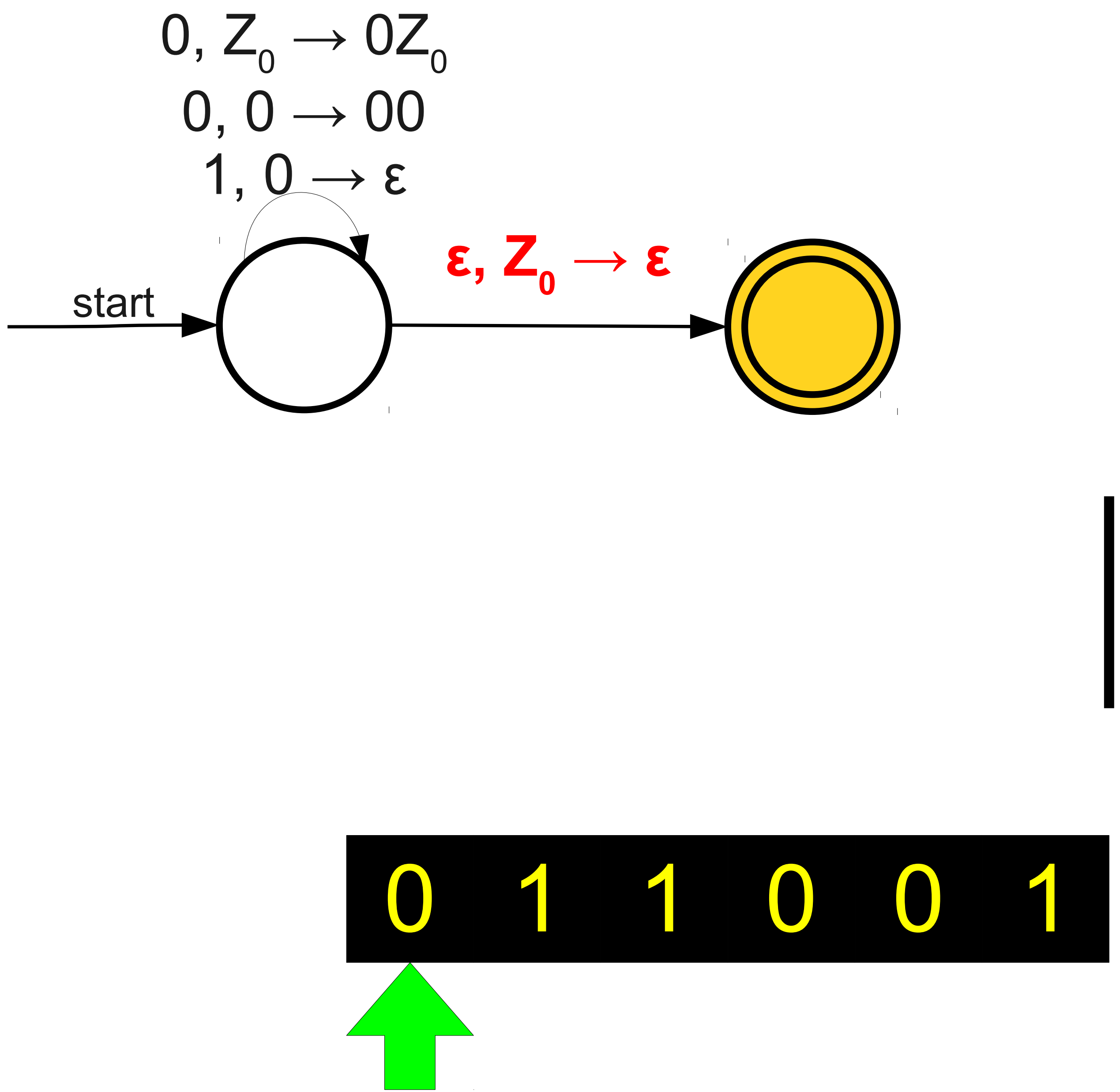
A Simple Pushdown Automaton



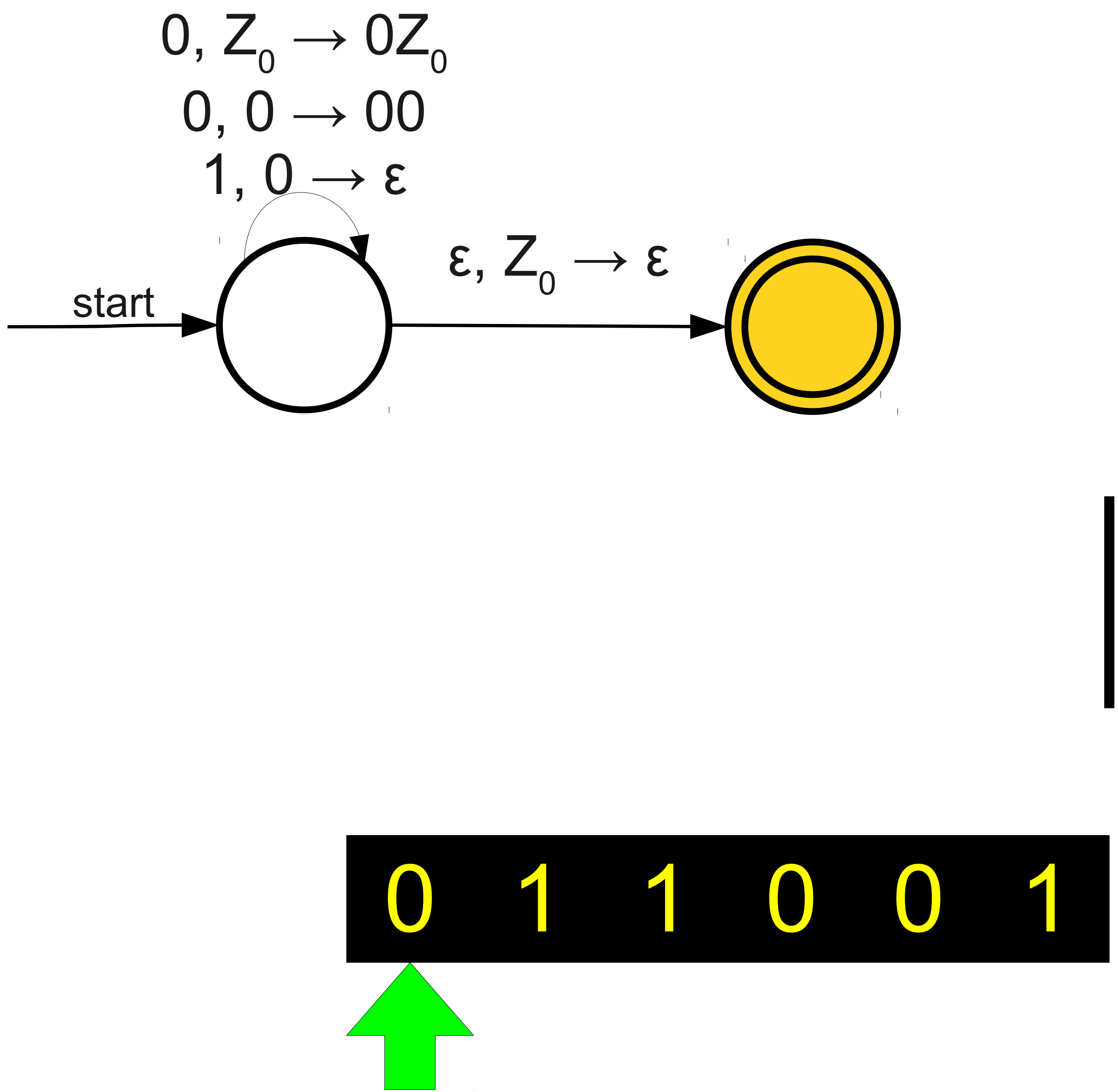
A Simple Pushdown Automaton



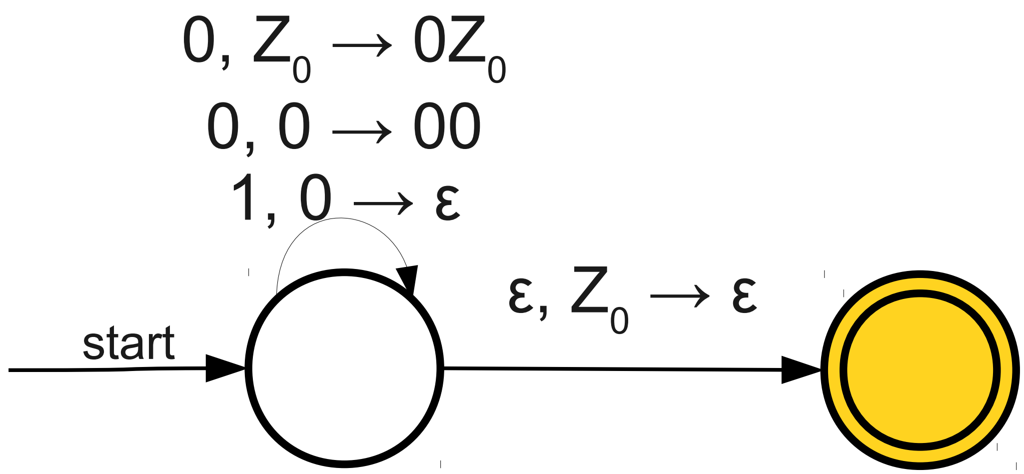
A Simple Pushdown Automaton

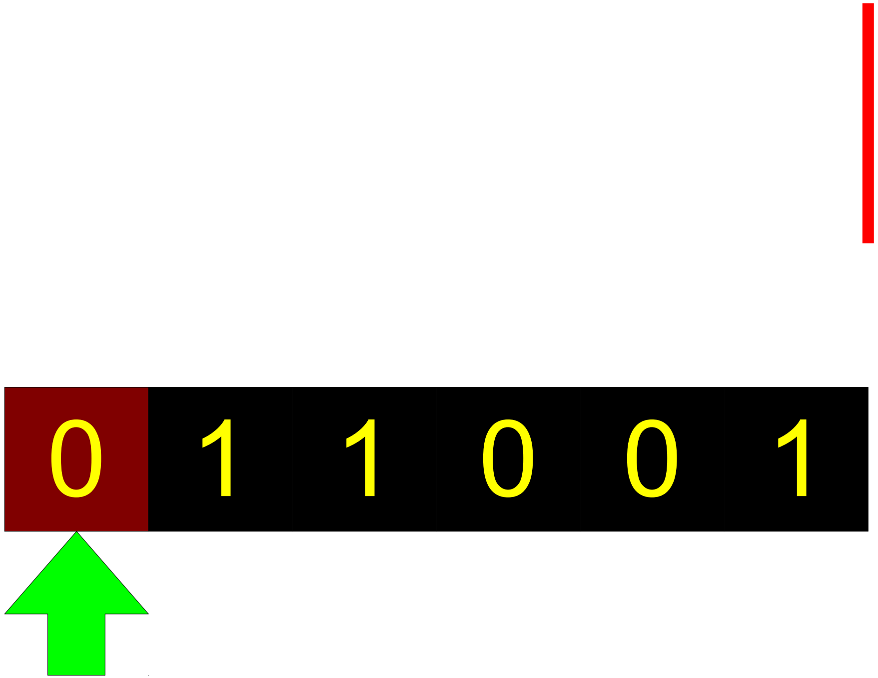


A Simple Pushdown Automaton

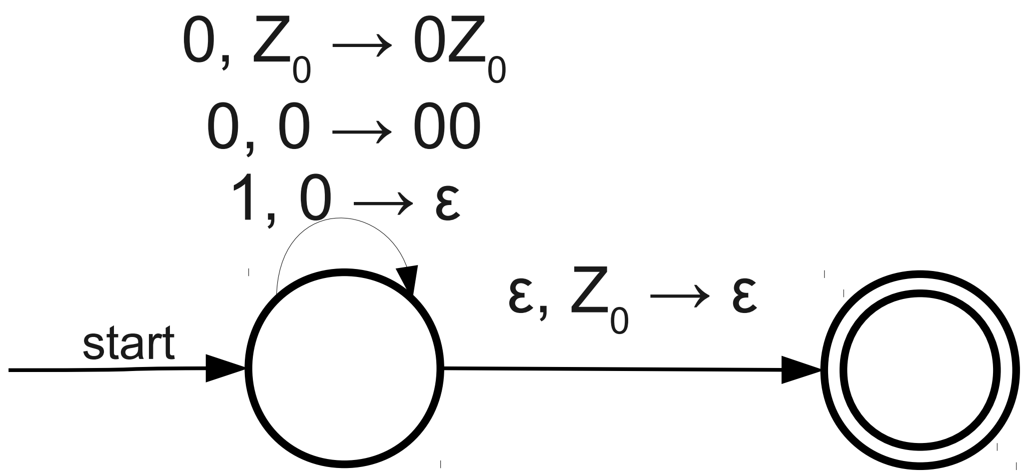


A Simple Pushdown Automaton

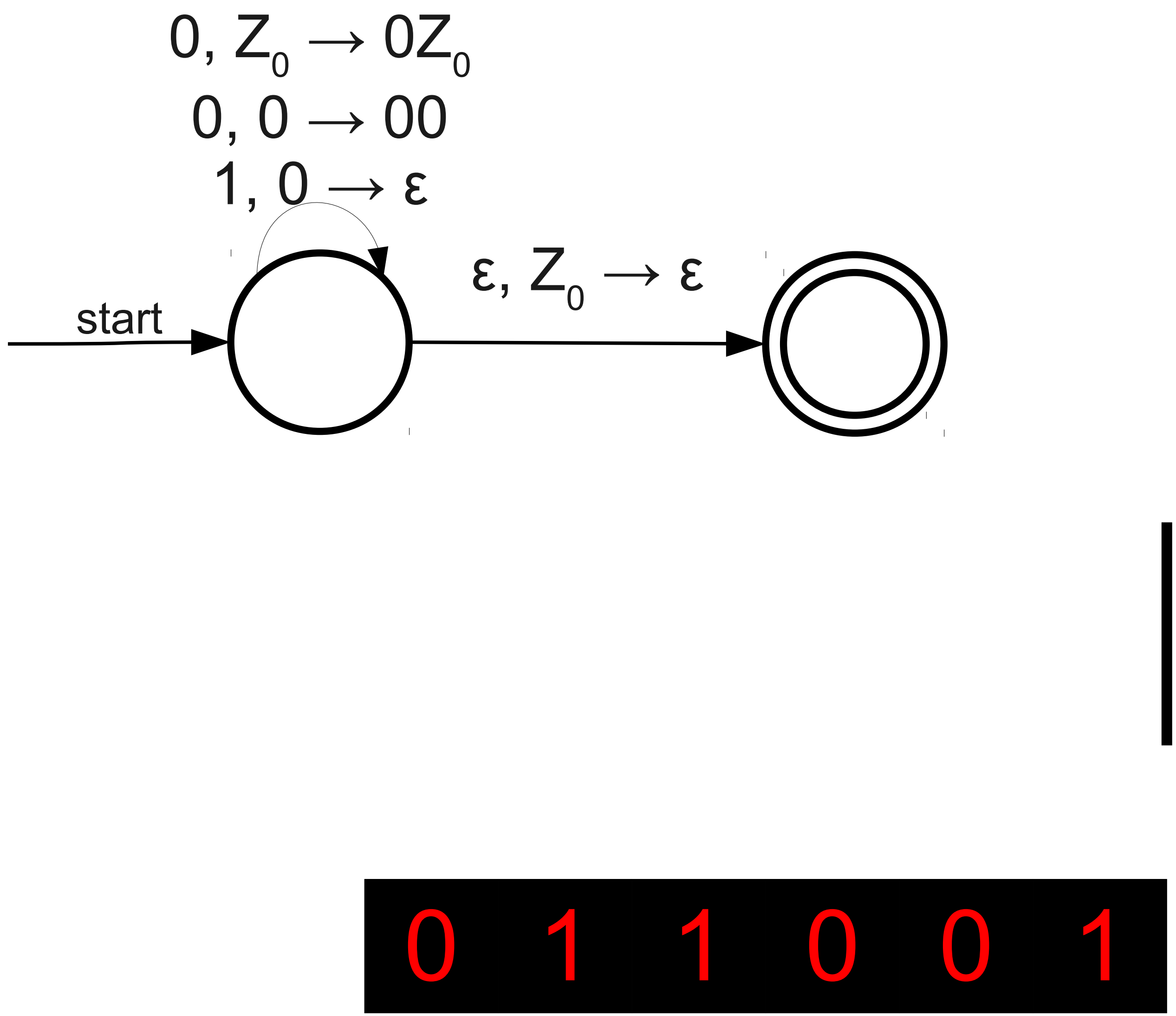




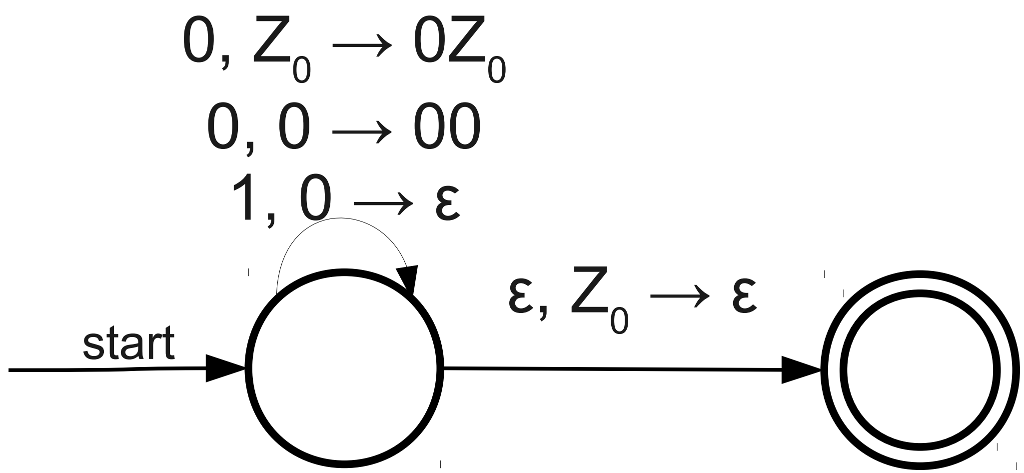
A Simple Pushdown Automaton

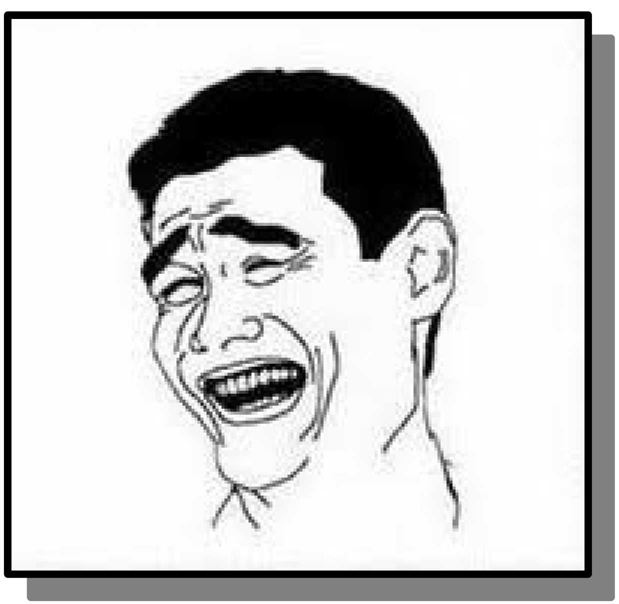


A Simple Pushdown Automaton



A Simple Pushdown Automaton





0 1 1 0 0 1

Pushdown Automata

- Formally, a **pushdown automaton** is a nondeterministic machine defined by the 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$

Pushdown Automata

- Formally, a **pushdown automaton** is a nondeterministic machine defined by the 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, where
  - Q is a finite set of states

Pushdown Automata

- Formally, a **pushdown automaton** is a nondeterministic machine defined by the 7-tuple (Q, Σ, Γ, δ, q_0, Z_0, F), where
  - Q is a finite set of states,
  - Σ is an alphabet

Pushdown Automata

- Formally, a **pushdown automaton** is a nondeterministic machine defined by the 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, where
  - Q is a finite set of states,
  - Σ is an alphabet,
  - Γ is the **stack alphabet** of symbols that can be pushed on the stack

Pushdown Automata

- Formally, a **pushdown automaton** is a nondeterministic machine defined by the 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, where
  - Q is a finite set of states,
  - Σ is an alphabet,
  - Γ is the **stack alphabet** of symbols that can be pushed on the stack

The **stack alphabet** allows PDAs' stacks to store extra information that can't otherwise be encoded by the input string.

Pushdown Automata

- Formally, a **pushdown automaton** is a nondeterministic machine defined by the 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, where
  - Q is a finite set of states,
  - Σ is an alphabet,
  - Γ is the **stack alphabet** of symbols that can be pushed on the stack,
  - $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \wp(Q \times \Gamma^*)$ is the **transition function**, where no tuple is mapped to an infinite set

Pushdown Automata

- Formally, a **pushdown automaton** is a nondeterministic machine defined by the 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, where
  - Q is a finite set of states,
  - Σ is an alphabet,
  - Γ is the **stack alphabet** of symbols that can be pushed on the stack,
  - $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \wp(Q \times \Gamma^*)$ is the **transition function**, where no tuple is mapped to an infinite set

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

Pushdown Automata

- Formally, a **pushdown automaton** is a nondeterministic machine defined by the 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, where
  - Q is a finite set of states,
  - Σ is an alphabet,
  - Γ is the **stack alphabet** of symbols that can be pushed on the stack,
  - $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \wp(Q \times \Gamma^*)$ is the **transition function**, where no tuple is mapped to an infinite set

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.

Pushdown Automata

- Formally, a **pushdown automaton** is a nondeterministic machine defined by the 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, where
  - Q is a finite set of states,
  - Σ is an alphabet,
  - Γ is the **stack alphabet** of symbols that can be pushed on the stack,
  - $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \wp(Q \times \Gamma^*)$ is the **transition function**, where no tuple is mapped to an infinite set

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.

We only allow a finite set of choices to be made at each point.

Pushdown Automata

- Formally, a **pushdown automaton** is a nondeterministic machine defined by the 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, where
  - Q is a finite set of states,
  - Σ is an alphabet,
  - Γ is the **stack alphabet** of symbols that can be pushed on the stack,
  - $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \wp(Q \times \Gamma^*)$ is the **transition function**, where no tuple is mapped to an infinite set,
  - $q_0 \in Q$ is the **start state**

Pushdown Automata

- Formally, a **pushdown automaton** is a nondeterministic machine defined by the 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, where
  - Q is a finite set of states,
  - Σ is an alphabet,
  - Γ is the **stack alphabet** of symbols that can be pushed on the stack,
  - $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \wp(Q \times \Gamma^*)$ is the **transition function**, where no tuple is mapped to an infinite set,
  - $q_0 \in Q$ is the **start state**,
  - $Z_0 \in \Gamma$ is the **stack start symbol**

Pushdown Automata

- Formally, a **pushdown automaton** is a nondeterministic machine defined by the 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, where
  - Q is a finite set of states,
  - Σ is an alphabet,
  - Γ is the **stack alphabet** of symbols that can be pushed on the stack,
  - $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \wp(Q \times \Gamma^*)$ is the **transition function**, where no tuple is mapped to an infinite set,
  - $q_0 \in Q$ is the **start state**,
  - $Z_0 \in \Gamma$ is the **stack start symbol**

This ensures that there is a symbol on the stack that we can use to detect whether the stack has nothing else on it.

Pushdown Automata

- Formally, a **pushdown automaton** is a nondeterministic machine defined by the 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, where
  - Q is a finite set of states,
  - Σ is an alphabet,
  - Γ is the **stack alphabet** of symbols that can be pushed on the stack,
  - $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \wp(Q \times \Gamma^*)$ is the **transition function**, where no tuple is mapped to an infinite set,
  - $q_0 \in Q$ is the **start state**,
  - $Z_0 \in \Gamma$ is the **stack start symbol**, and
  - $F \subseteq Q$ is the set of **accepting states**.

Pushdown Automata

- Formally, a **pushdown automaton** is a nondeterministic machine defined by the 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, where
  - Q is a finite set of states,
  - Σ is an alphabet,
  - Γ is the **stack alphabet** of symbols that can be pushed on the stack,
  - $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \wp(Q \times \Gamma^*)$ is the **transition function**, where no tuple is mapped to an infinite set,
  - $q_0 \in Q$ is the **start state**,
  - $Z_0 \in \Gamma$ is the **stack start symbol**, and
  - $F \subseteq Q$ is the set of **accepting states**.
- The automaton accepts if it ends in an accepting state with no input remaining.

The Language of a PDA

- The **language of a PDA** is the set of strings that the PDA accepts:

$$\mathscr{L}(P) = \{ w \in \Sigma^* \mid P \text{ accepts } w \}$$

- If P is a PDA where $\mathscr{L}(P) = L$, we say that P **recognizes** L.

A Note on Terminology

- Finite automata are highly standardized.
- There are many equivalent but different definitions of PDAs.
- The one we will use is a slight variant on the one described in Sipser.
  - Sipser does not have a start stack symbol.
  - Sipser does not allow transitions to push multiple symbols onto the stack.
- Feel free to use either this version or Sipser's; the two are equivalent to one another.

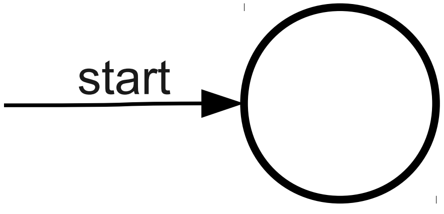
A PDA for Palindromes

- A **palindrome** is a string that is the same forwards and backwards.
- Let Σ = {`0`, `1`} and consider the language

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

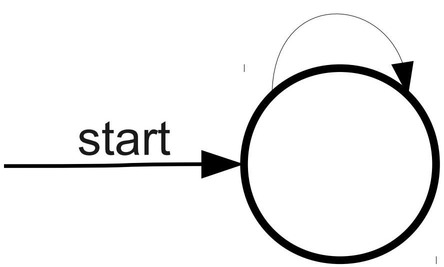
- How would we build a PDA for *PALINDROME*?
- ***Idea***: Push the first half of the symbols on to the stack, then verify that the second half of the symbols match.
- ***Nondeterministically*** guess when we've read half of the symbols.
- This handles even-length strings; we'll see a cute trick to handle odd-length strings in a minute.

A PDA for Palindromes



A PDA for Palindromes

$0, Z_0 \rightarrow 0Z_0$

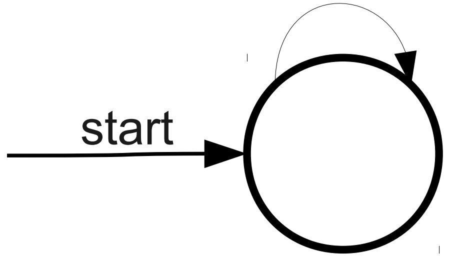


A PDA for Palindromes

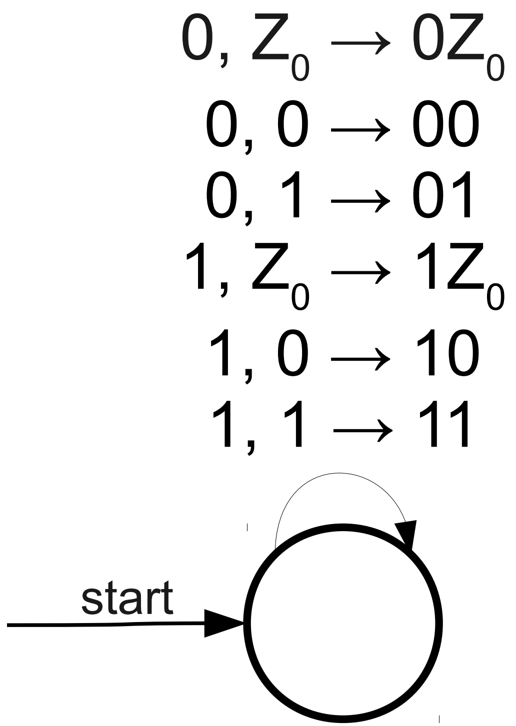
$0, Z_0 \to 0Z_0$

$0, 0 \to 00$

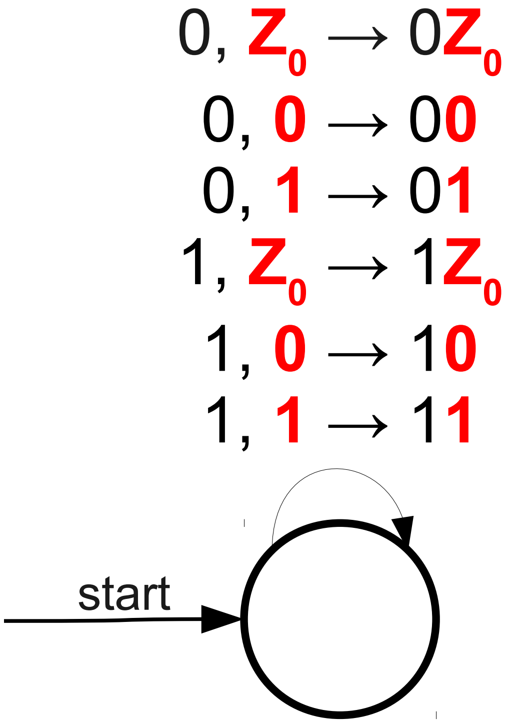
$0, 1 \to 01$



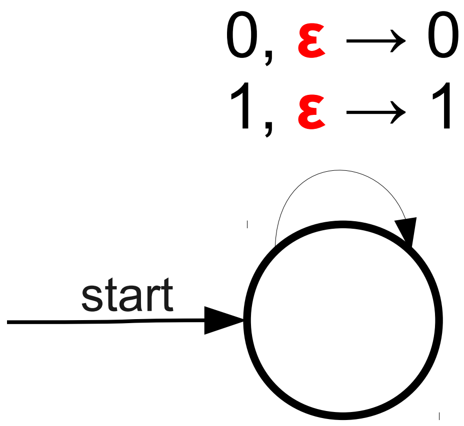
A PDA for Palindromes



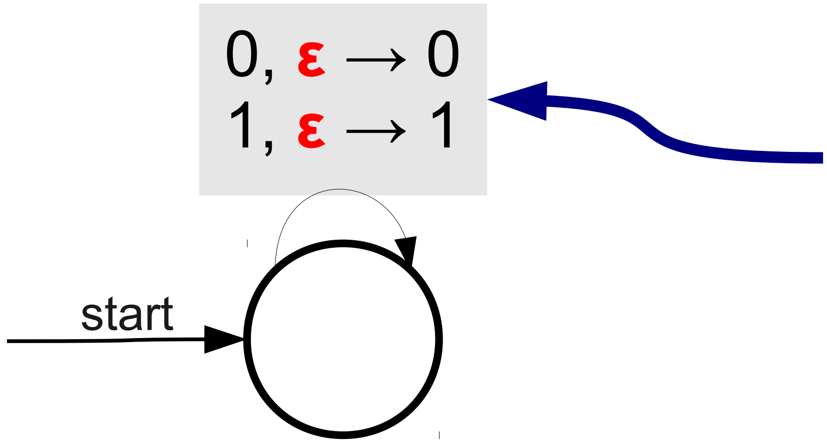
A PDA for Palindromes



A PDA for Palindromes

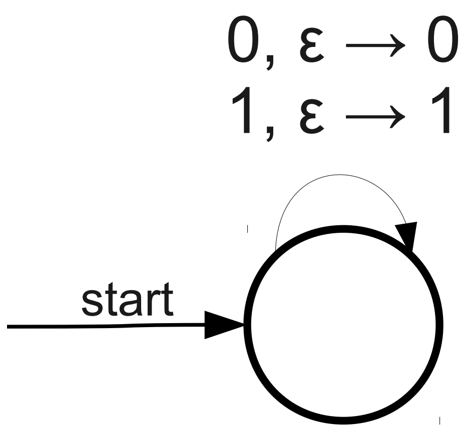


A PDA for Palindromes

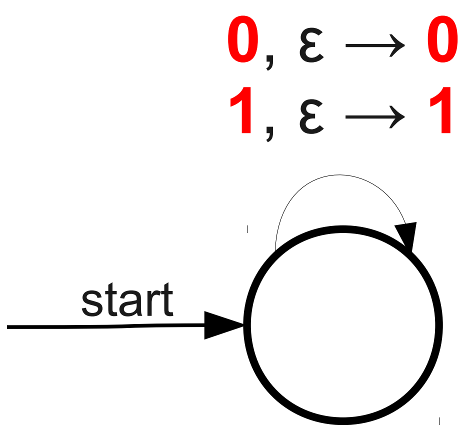


This transition indicates that the transition does not pop anything from the stack. It just pushes on a new symbol instead.

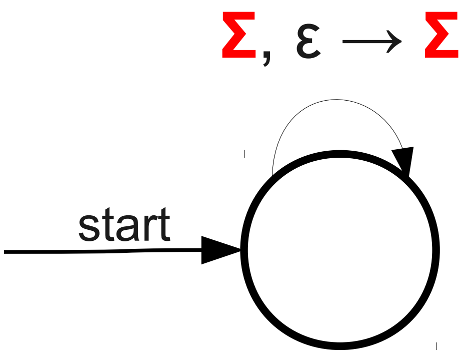
A PDA for Palindromes



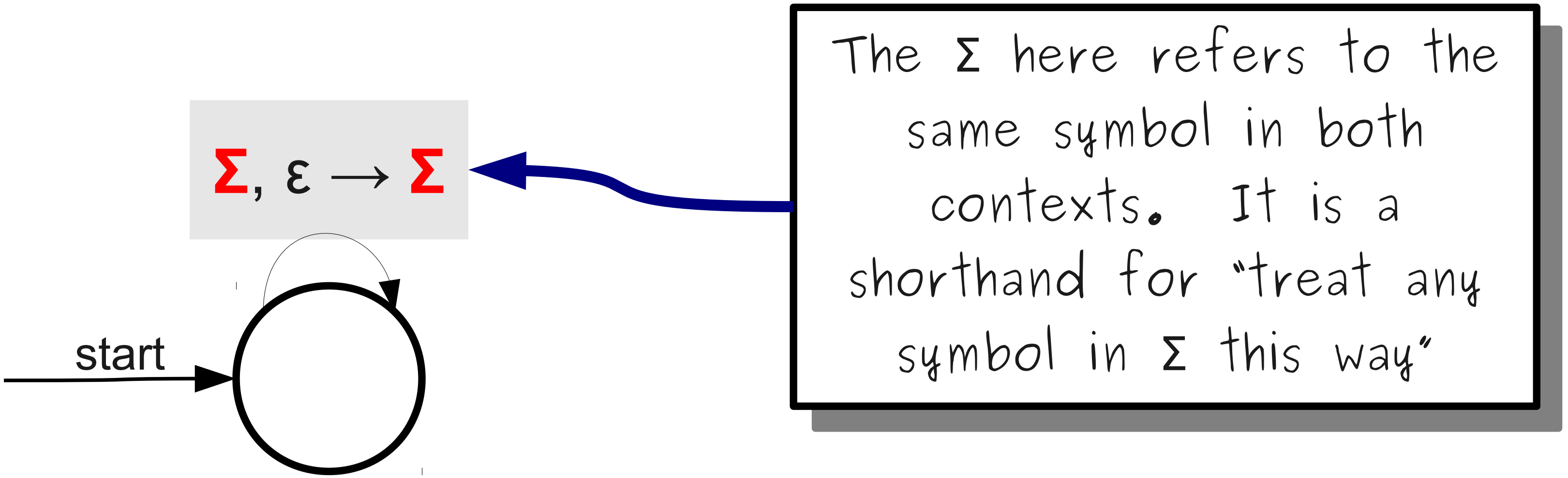
A PDA for Palindromes



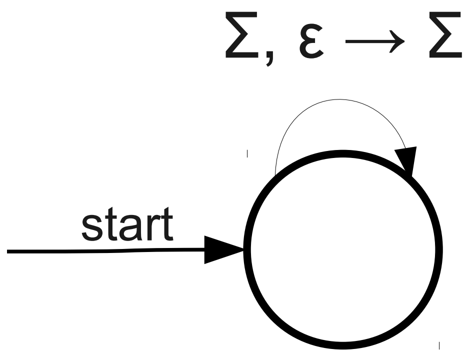
A PDA for Palindromes



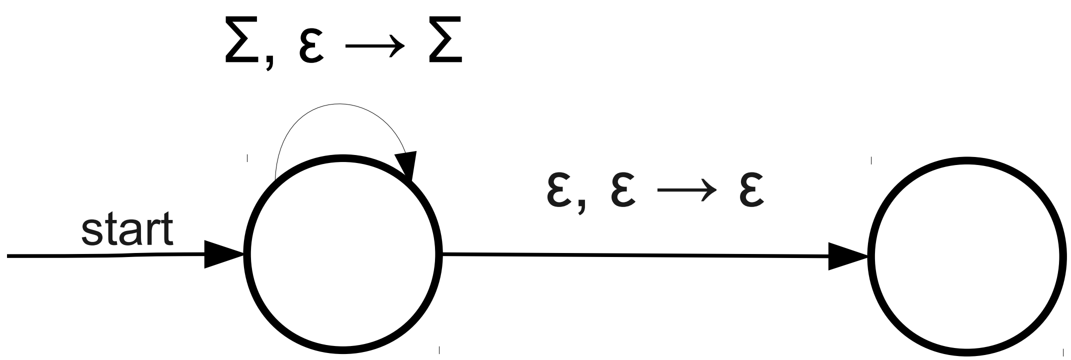
A PDA for Palindromes



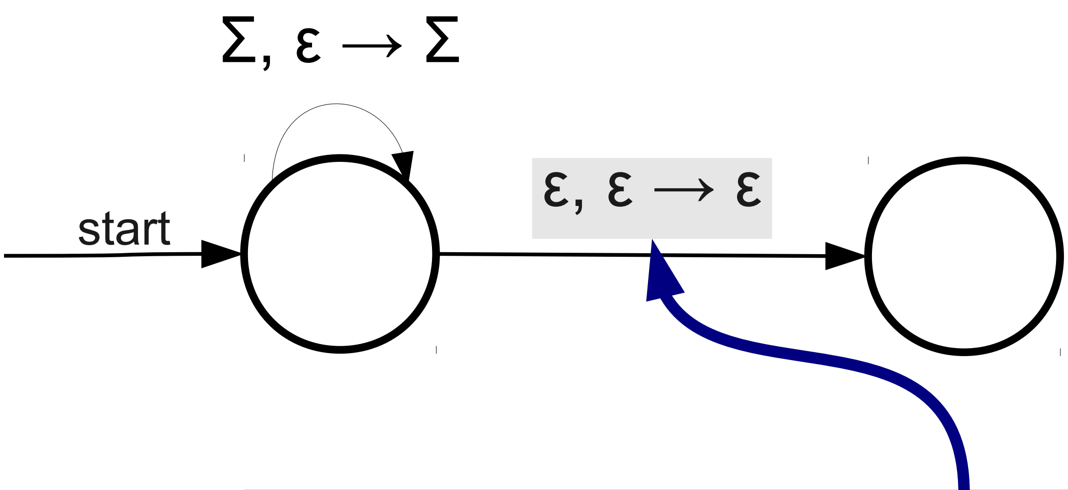
A PDA for Palindromes



A PDA for Palindromes

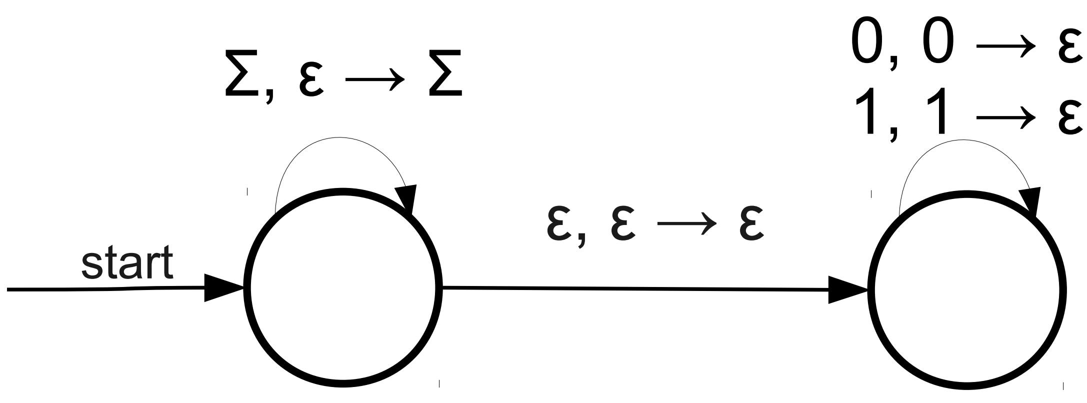


A PDA for Palindromes

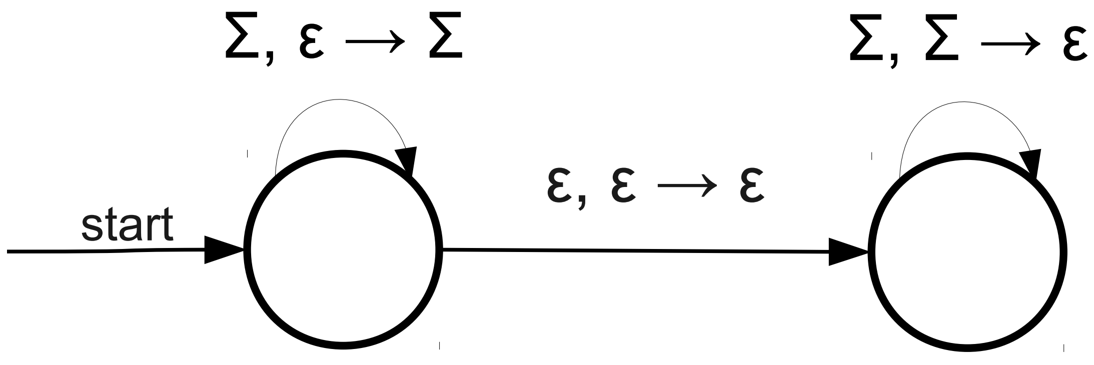


This transition means "don't consume any input, don't change the top of the stack, and don't add anything to a stack. It's the equivalent of an ε-transition in an NFA.

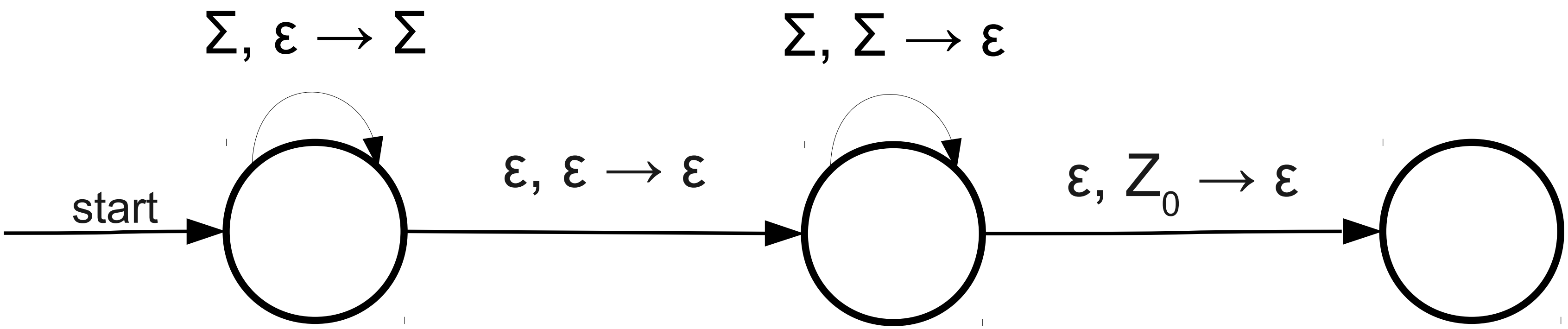
A PDA for Palindromes



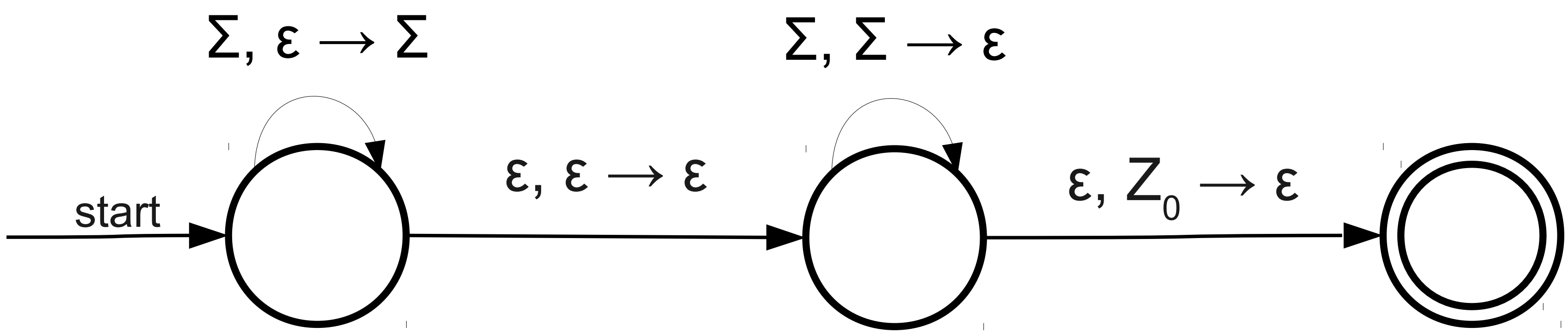
A PDA for Palindromes



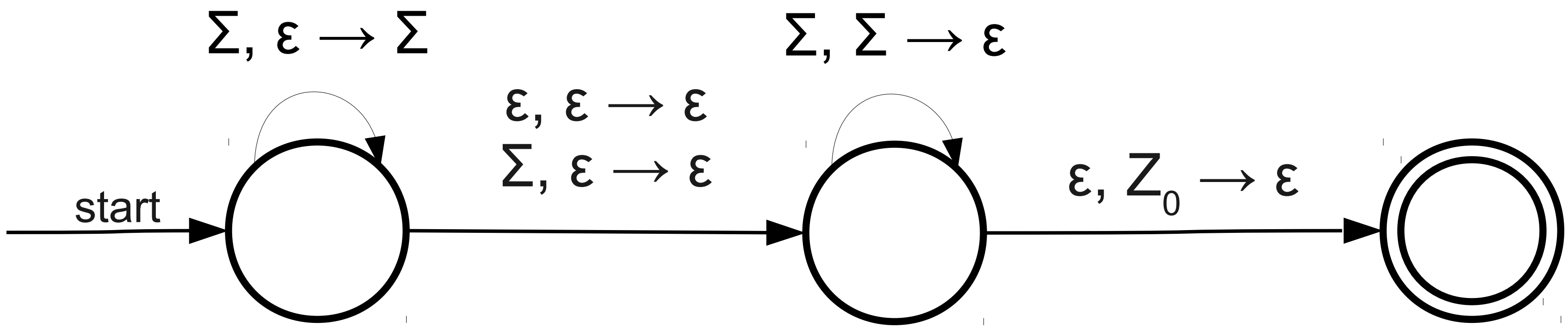
A PDA for Palindromes



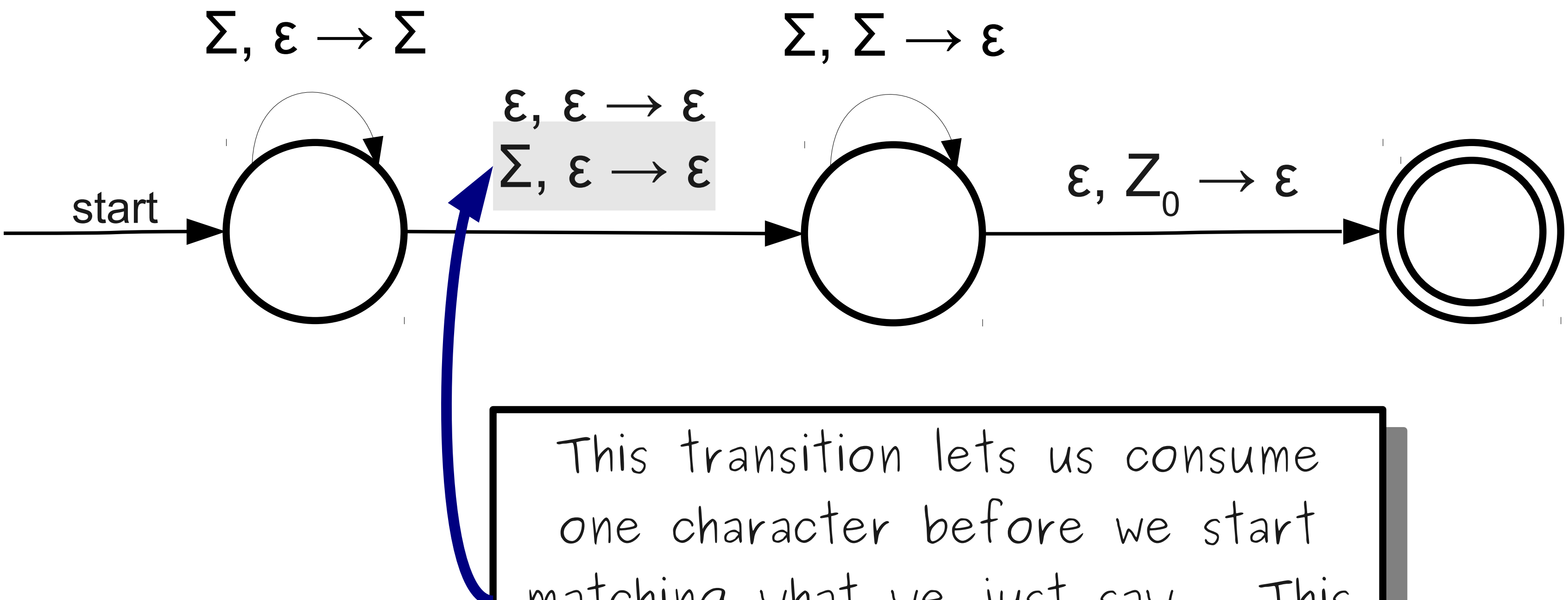
A PDA for Palindromes



A PDA for Palindromes

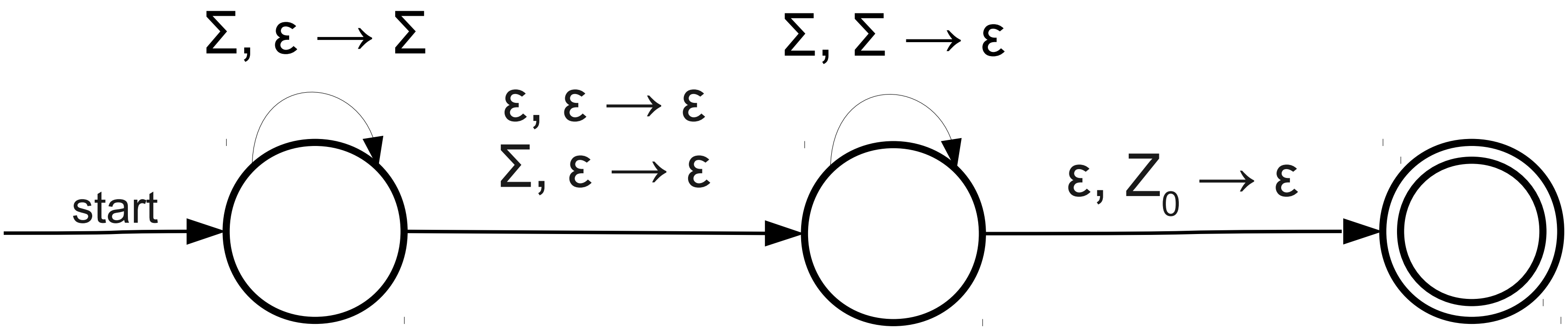


A PDA for Palindromes



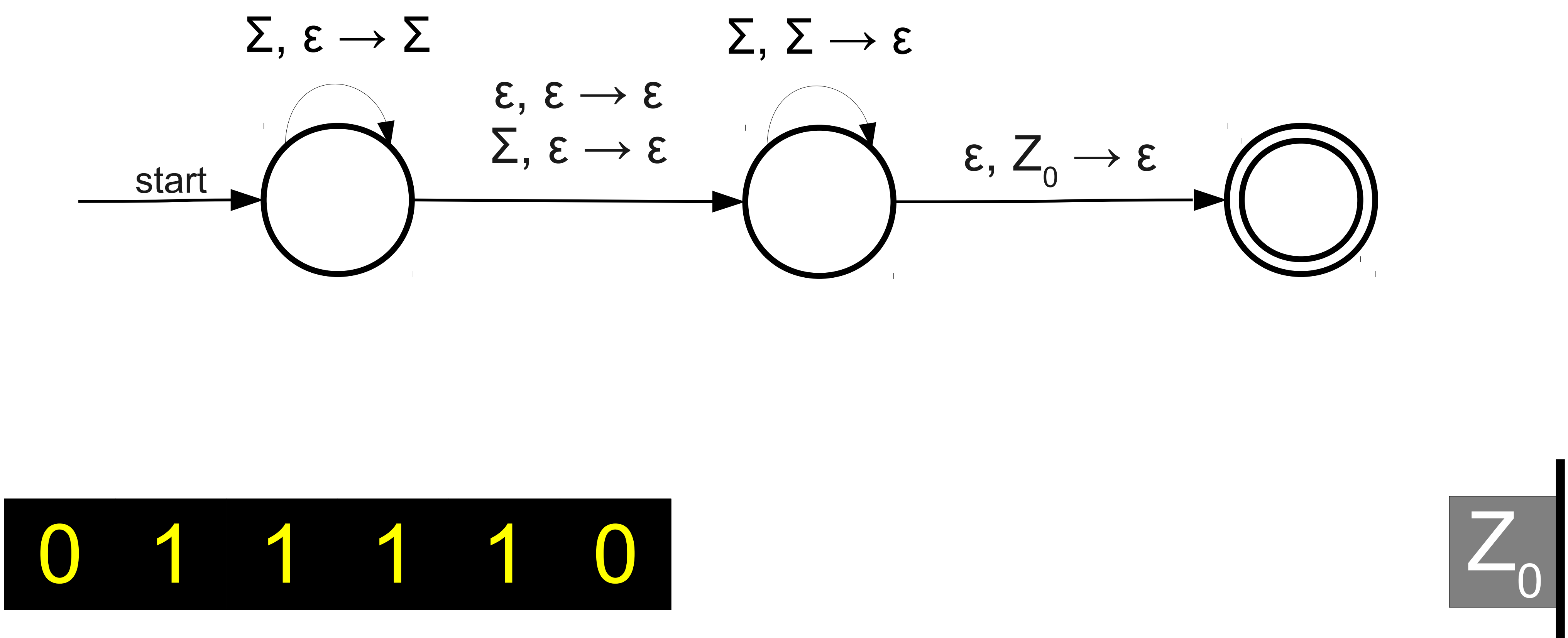
This transition lets us consume one character before we start matching what we just saw. This lets us match odd-length palindromes

A PDA for Palindromes

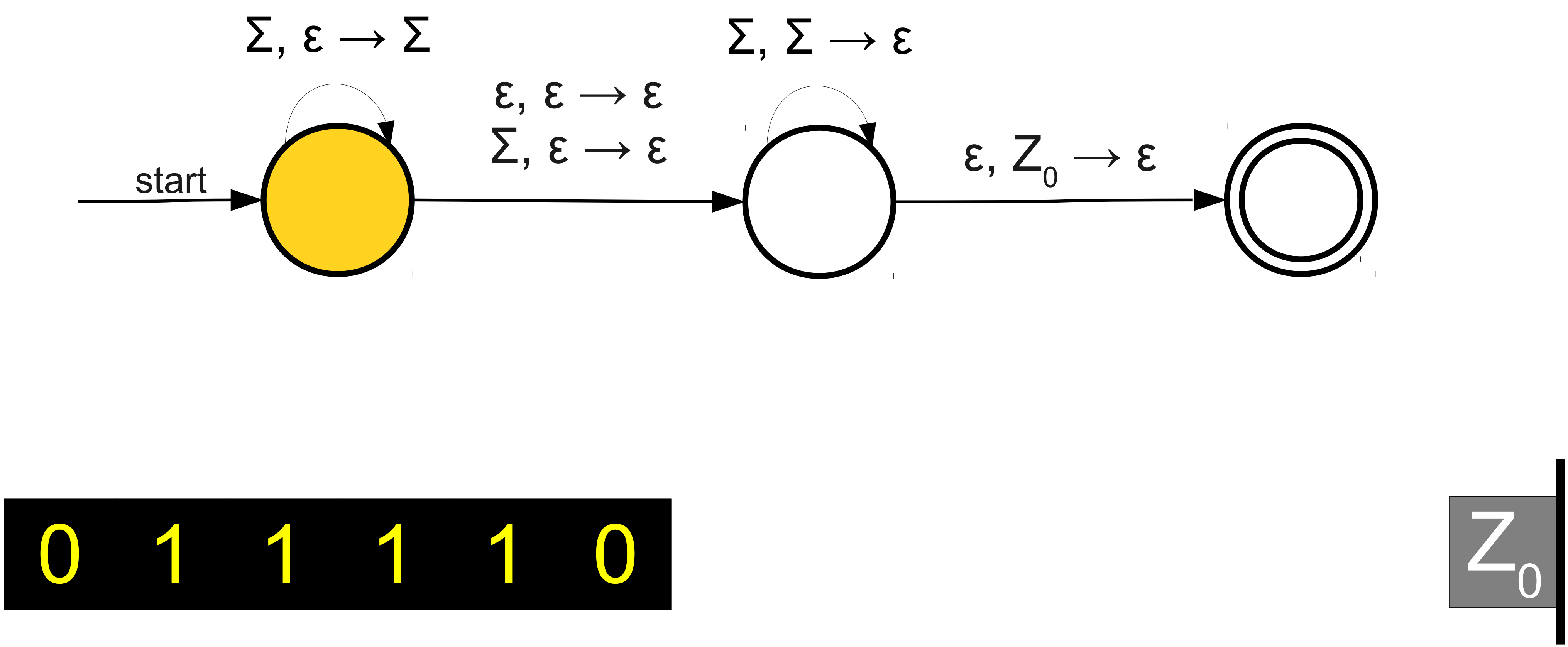


0 1 1 1 1 0

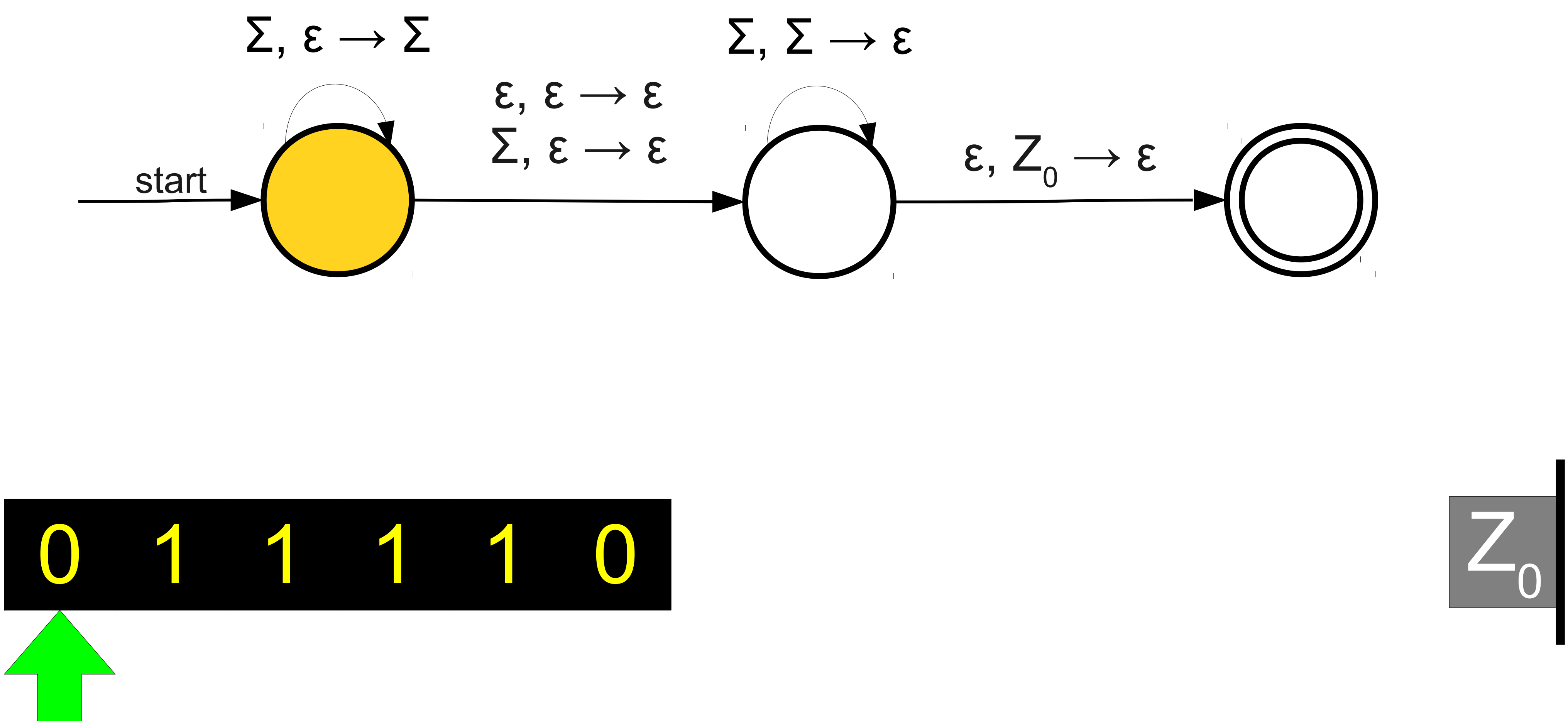
A PDA for Palindromes



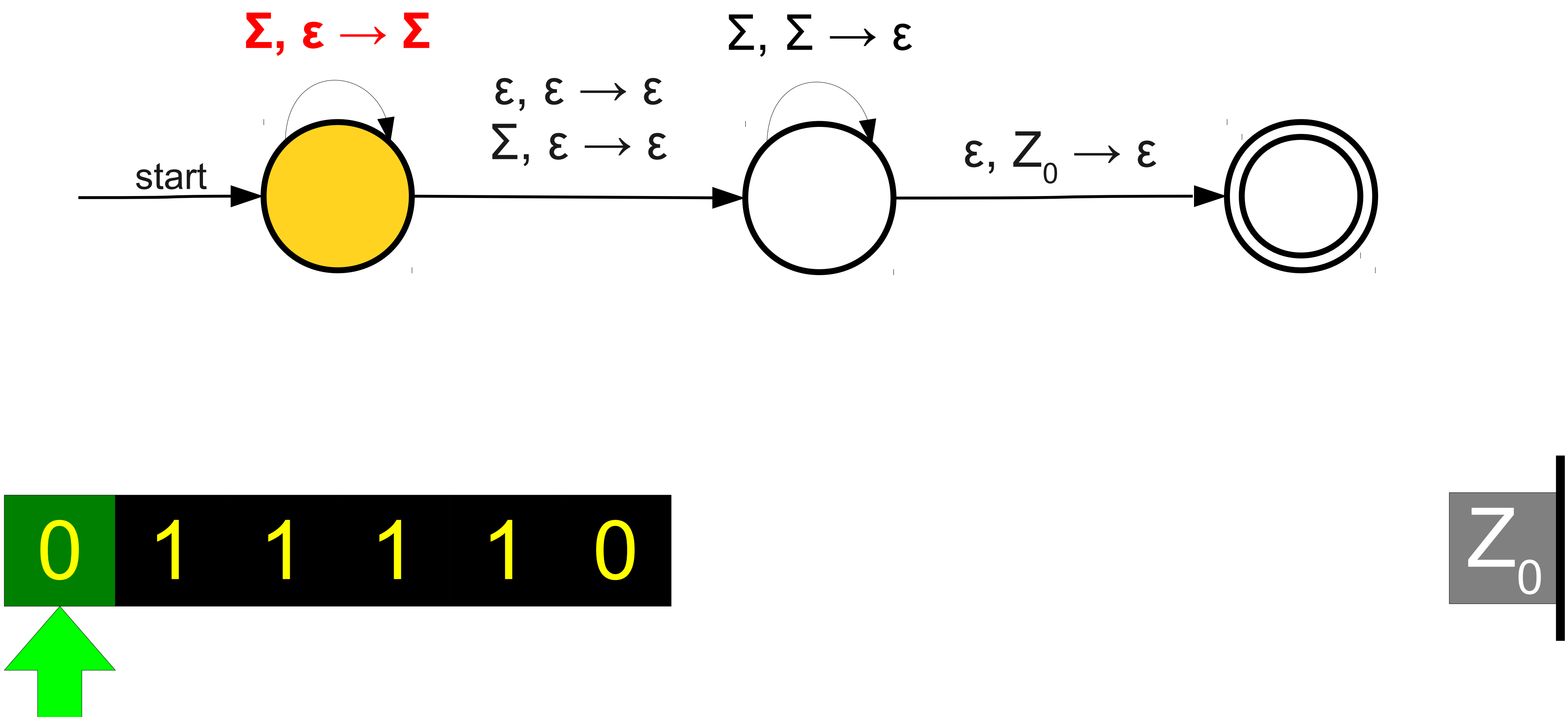
A PDA for Palindromes



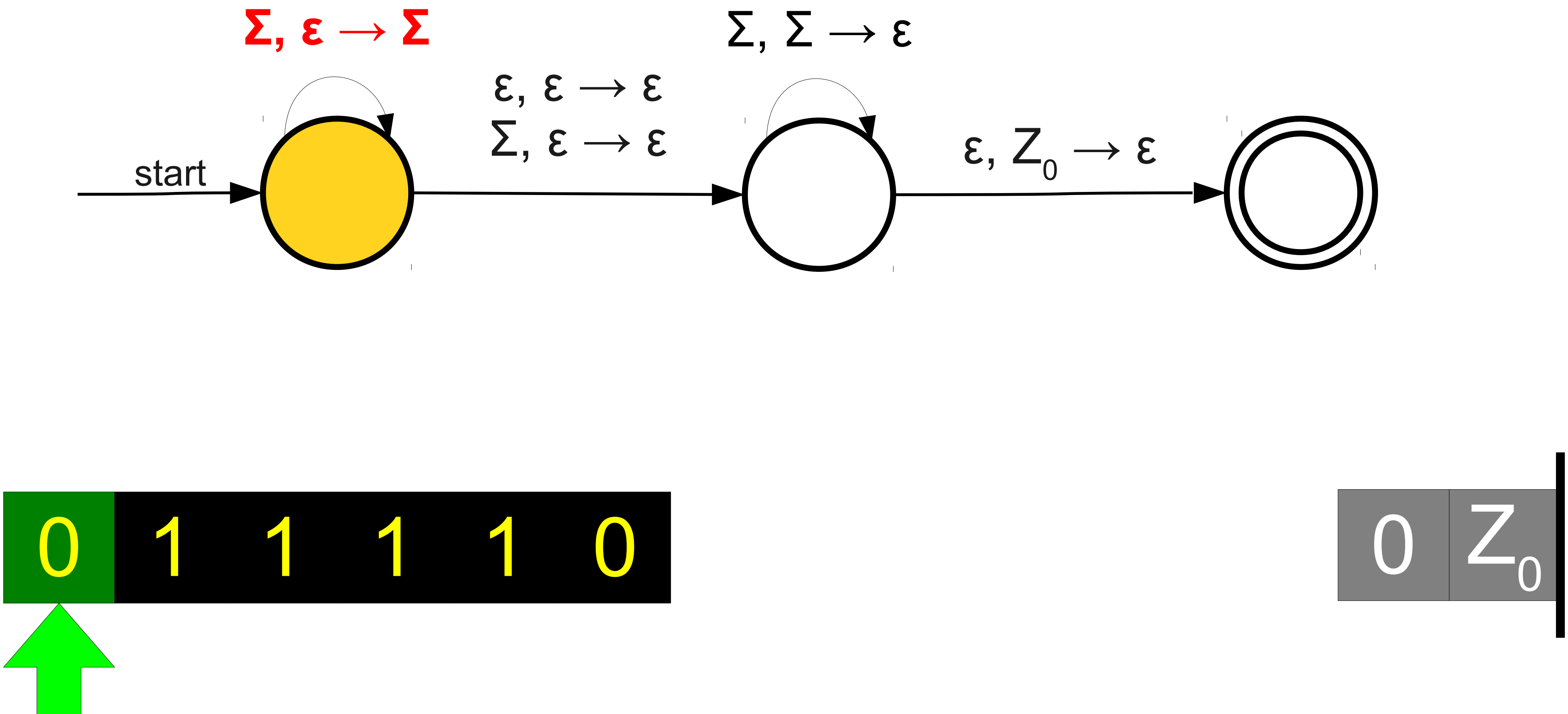
A PDA for Palindromes



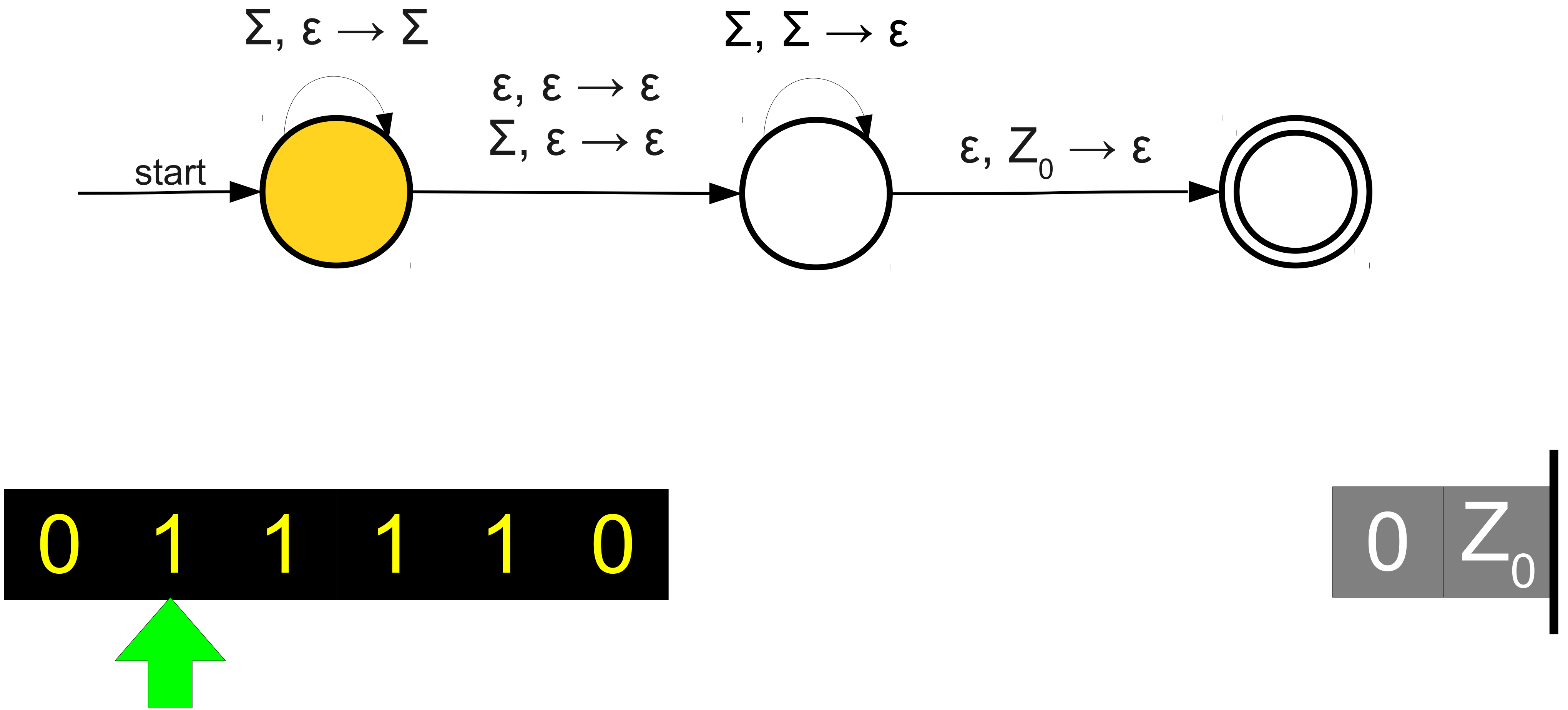
A PDA for Palindromes



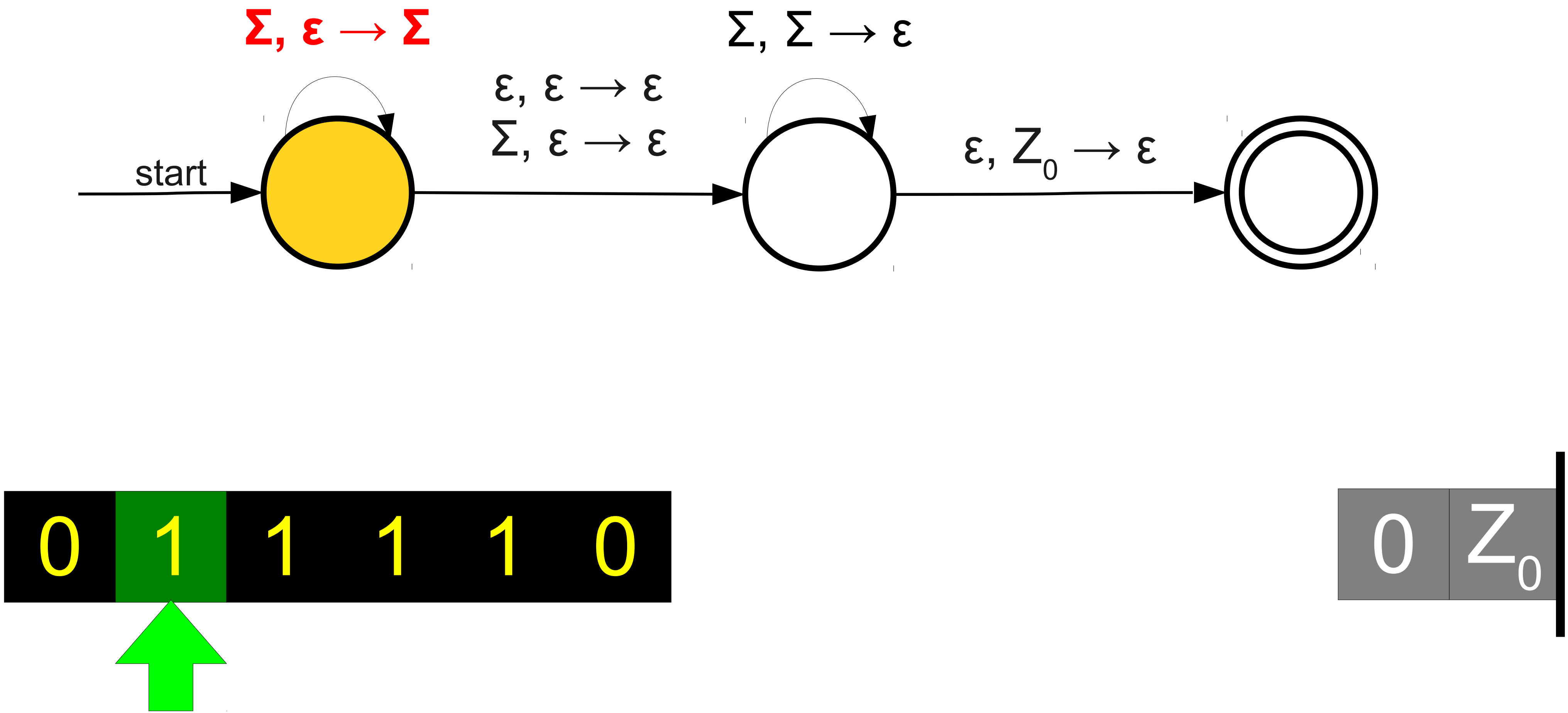
A PDA for Palindromes



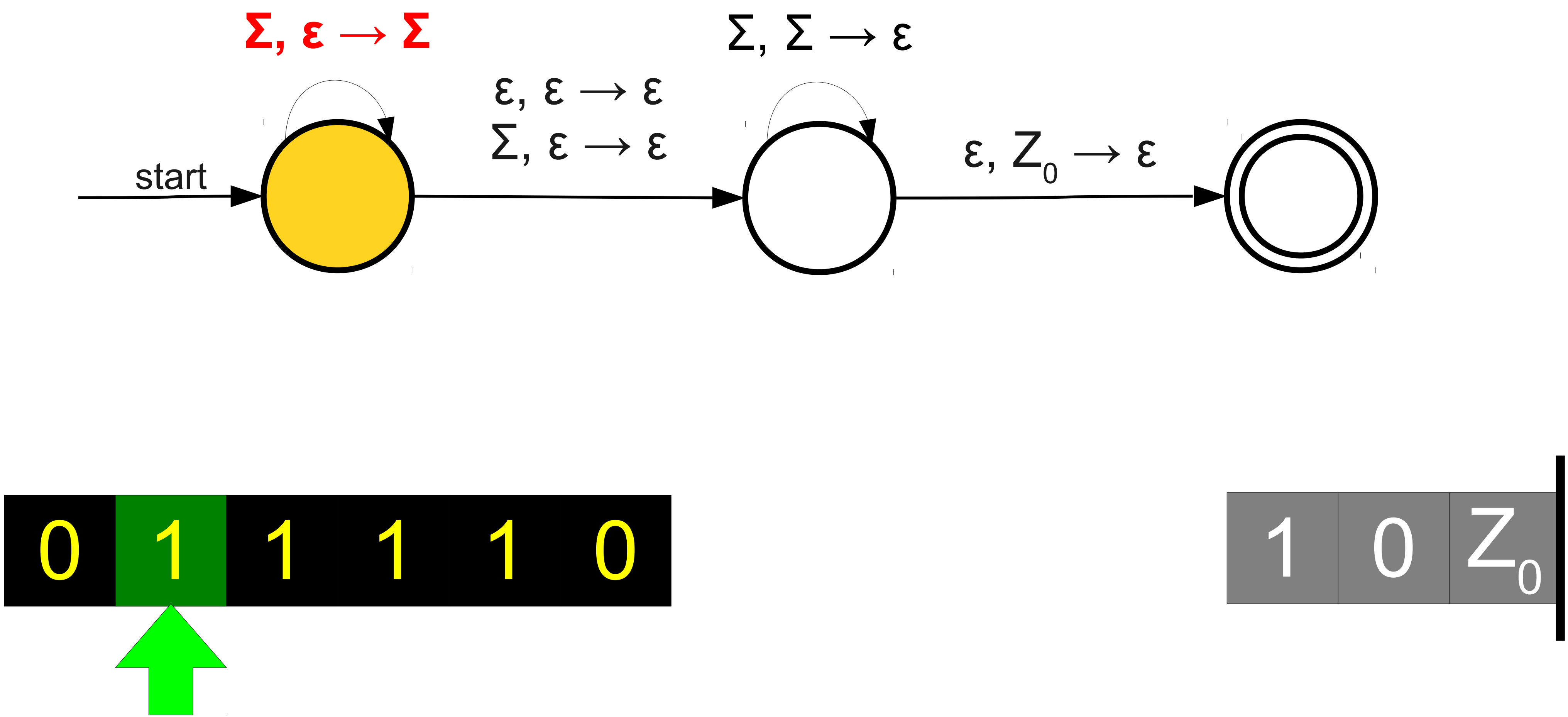
A PDA for Palindromes



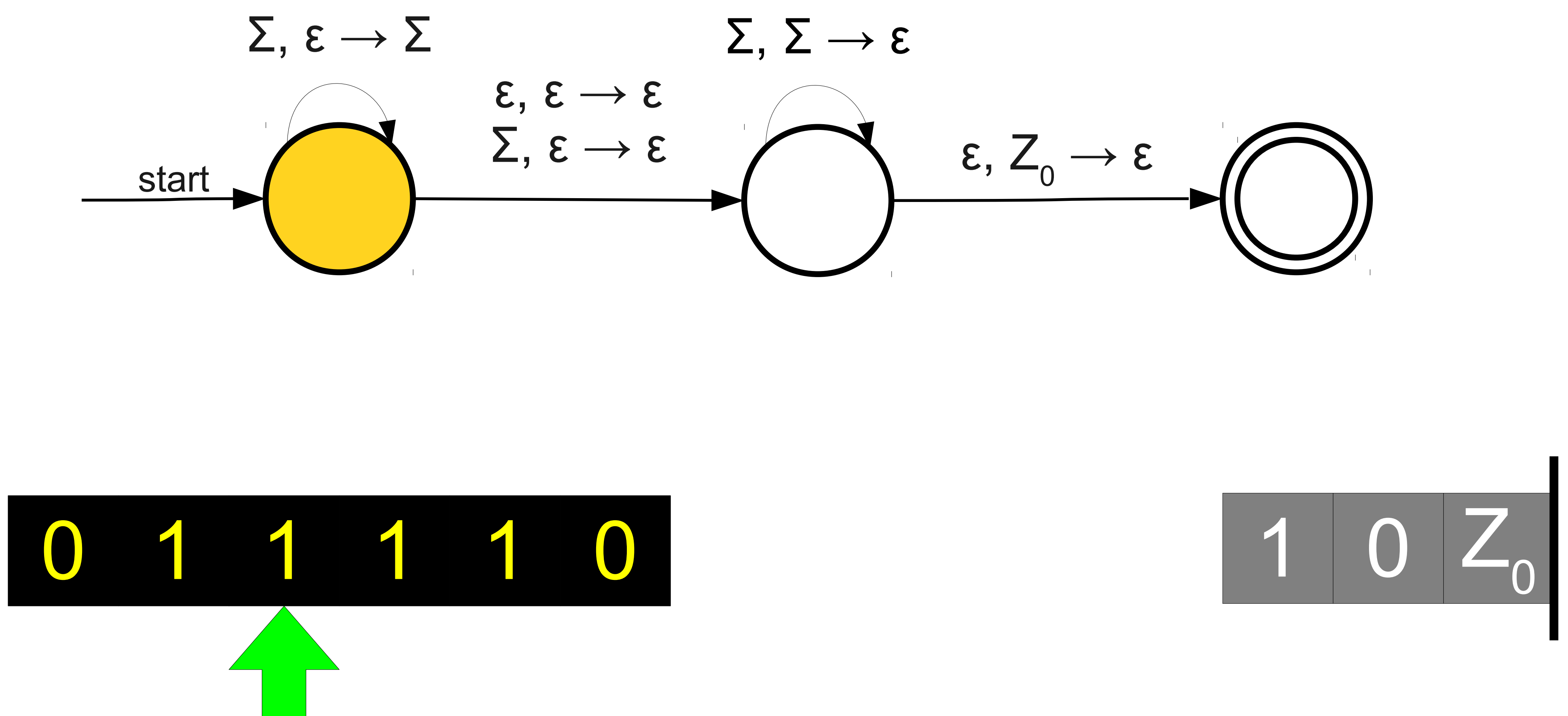
A PDA for Palindromes



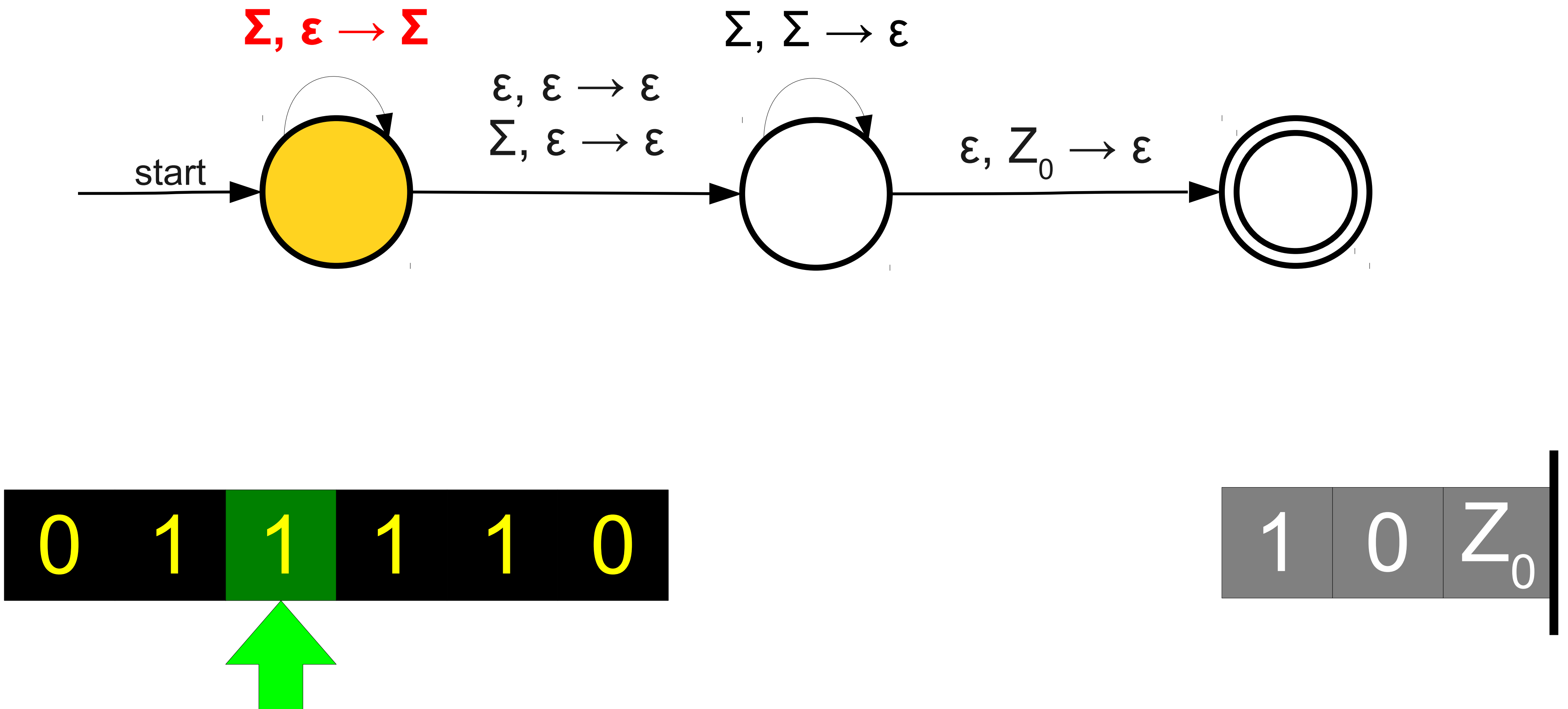
A PDA for Palindromes



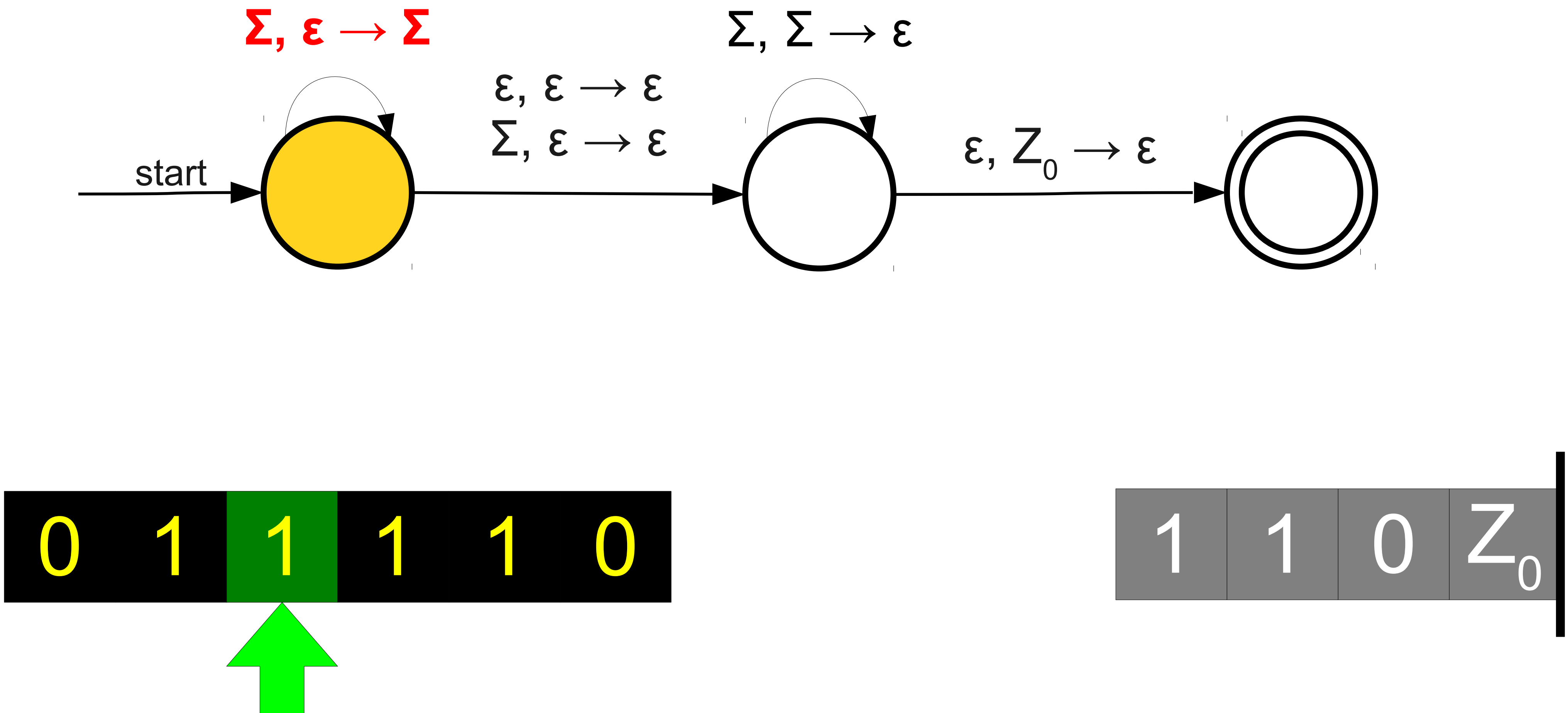
A PDA for Palindromes



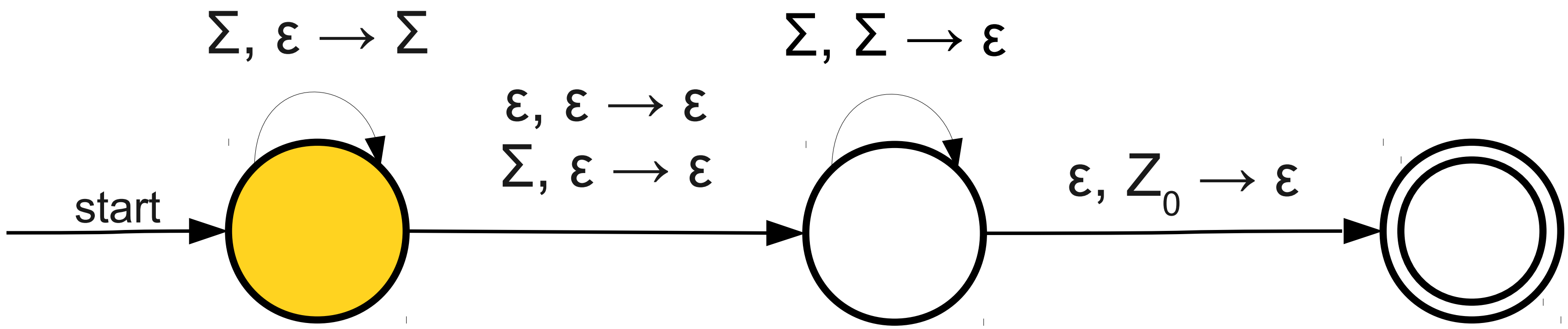
A PDA for Palindromes



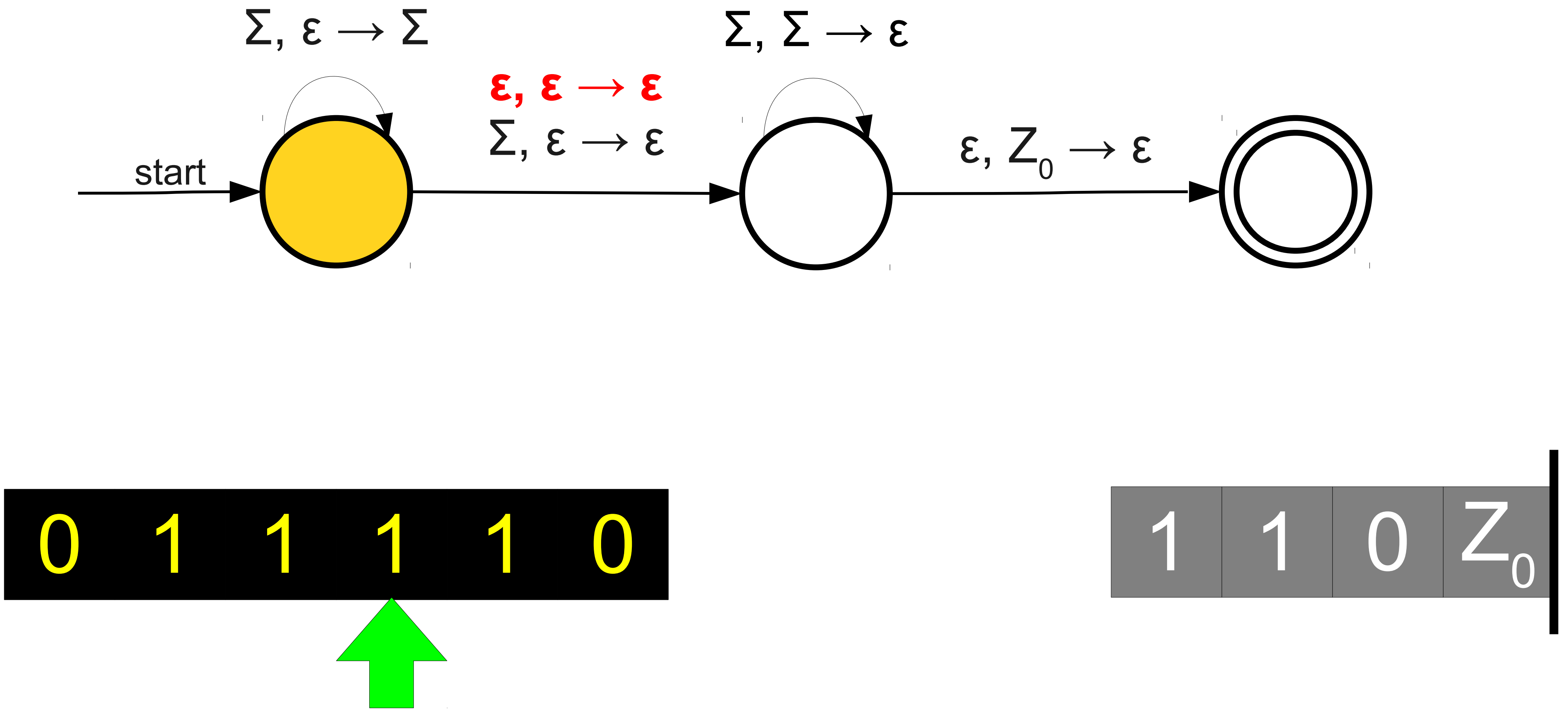
A PDA for Palindromes



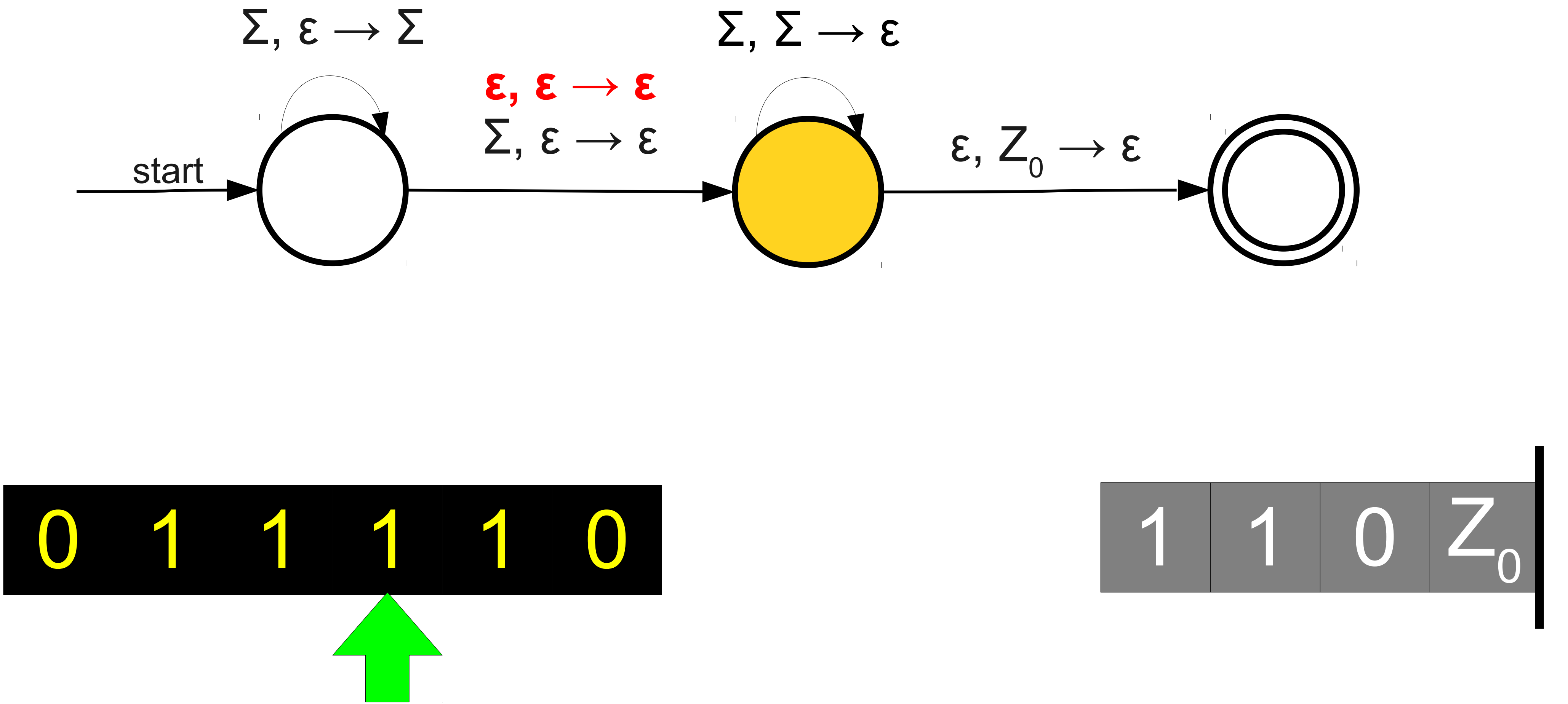
A PDA for Palindromes



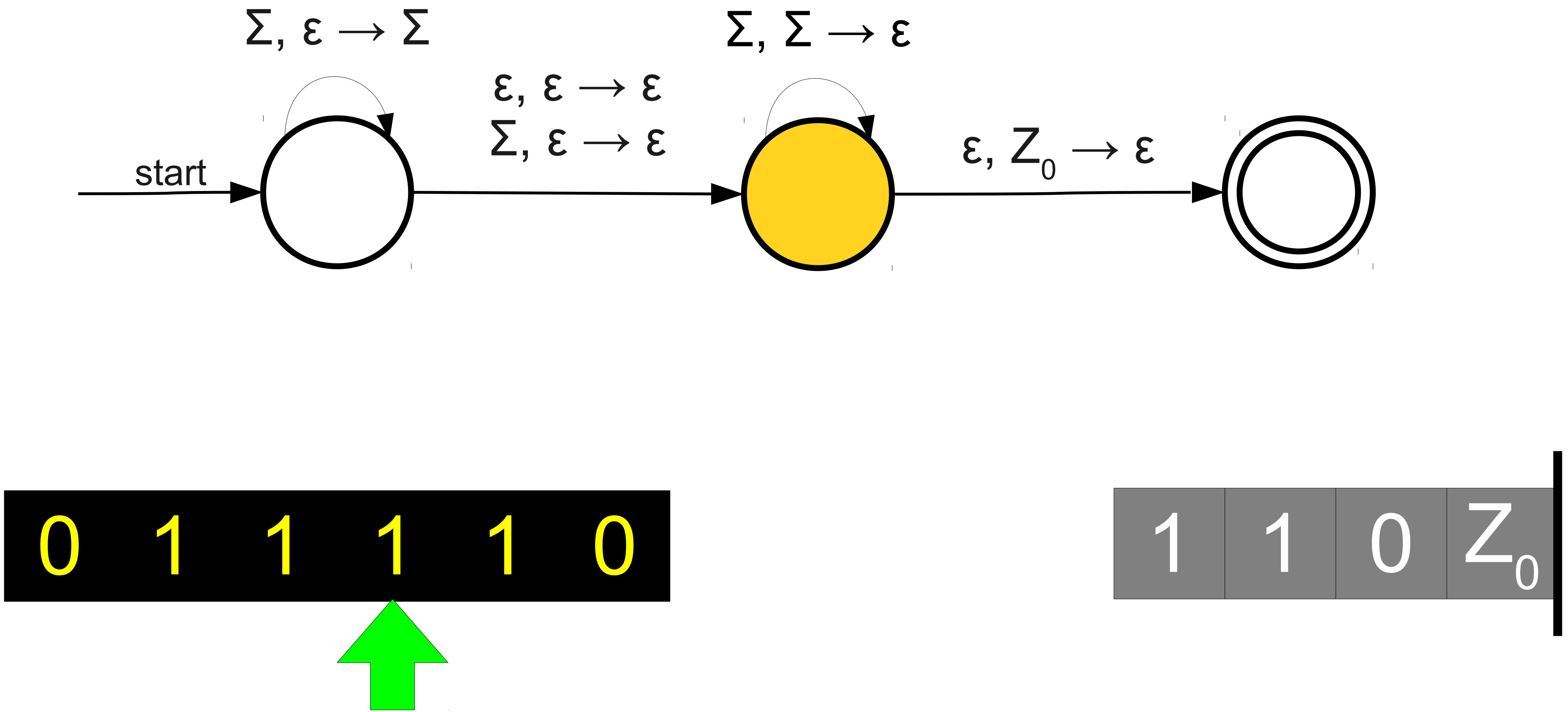
A PDA for Palindromes



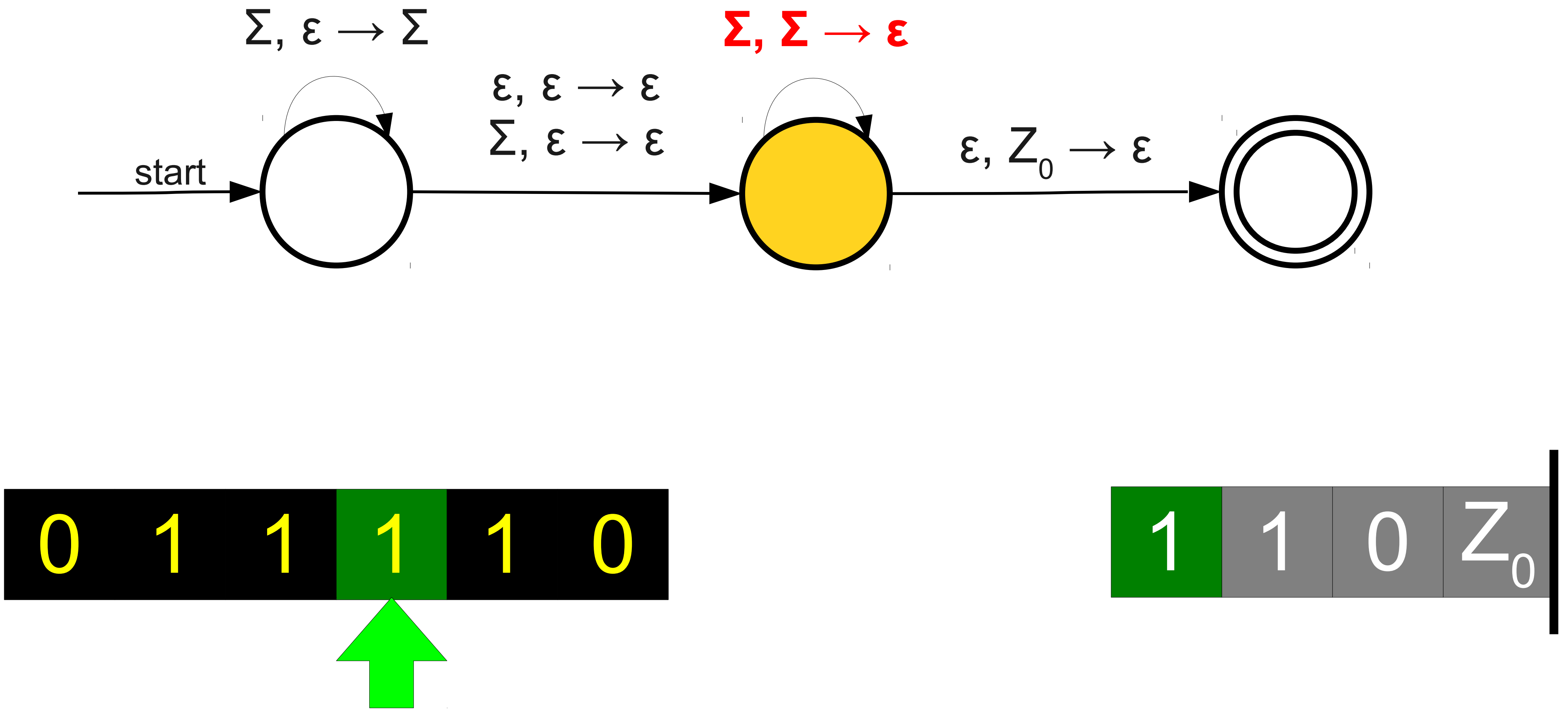
A PDA for Palindromes



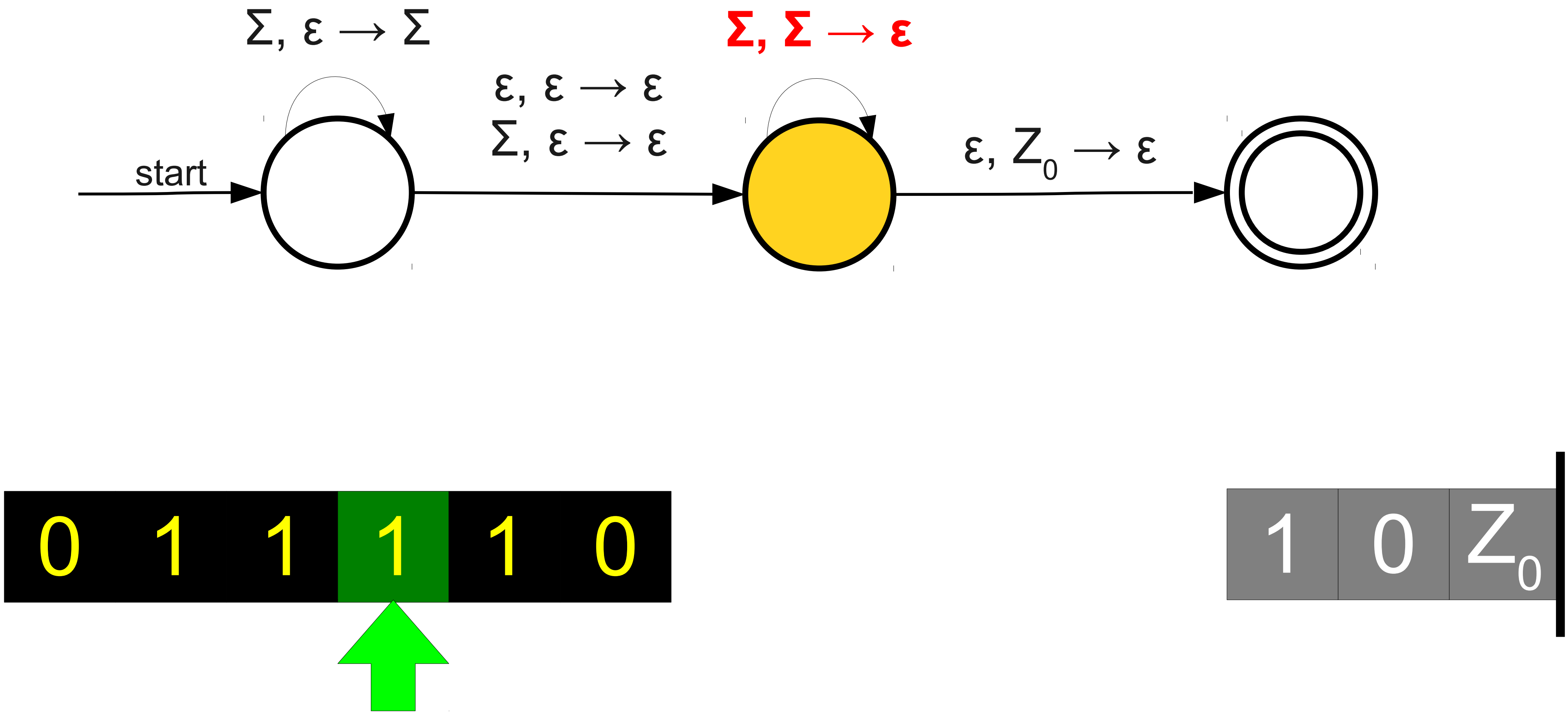
A PDA for Palindromes



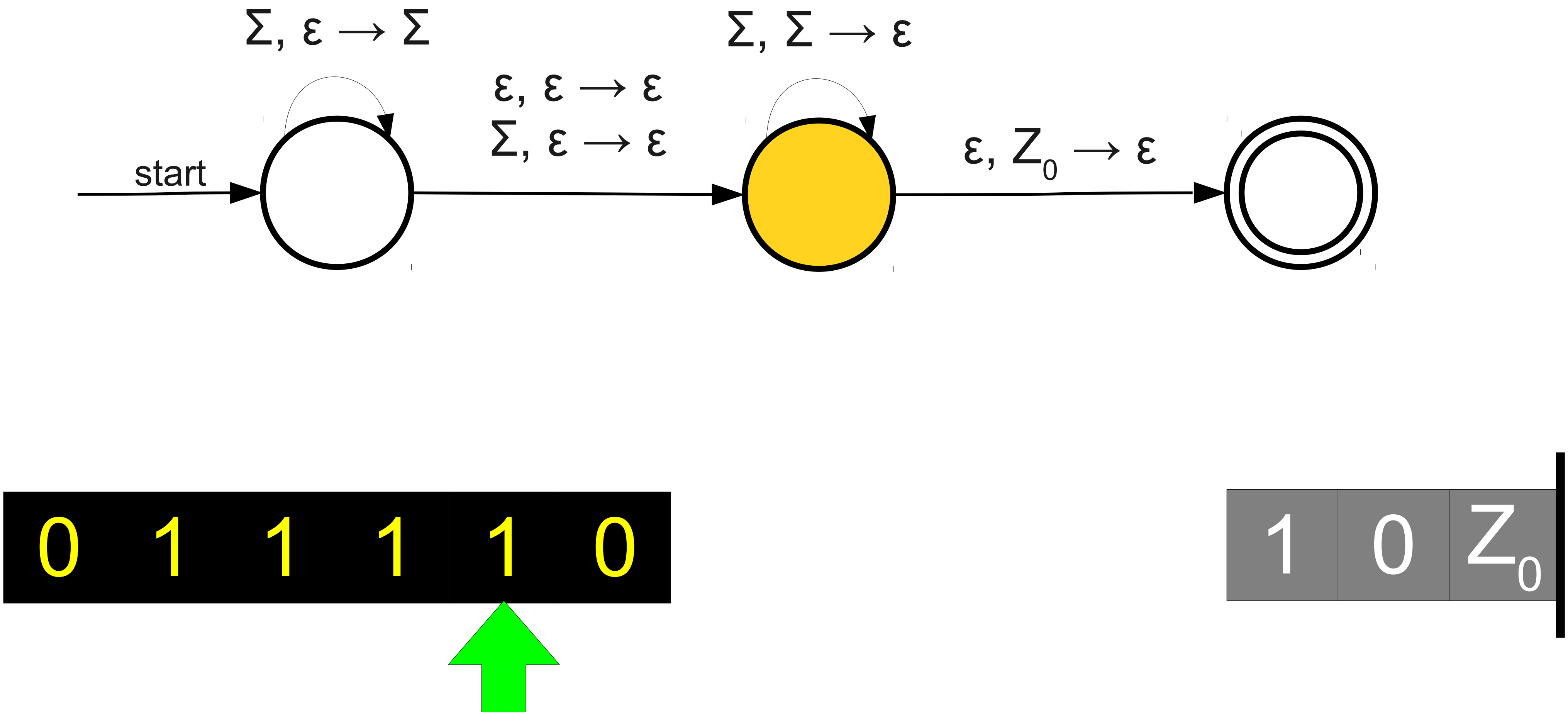
A PDA for Palindromes



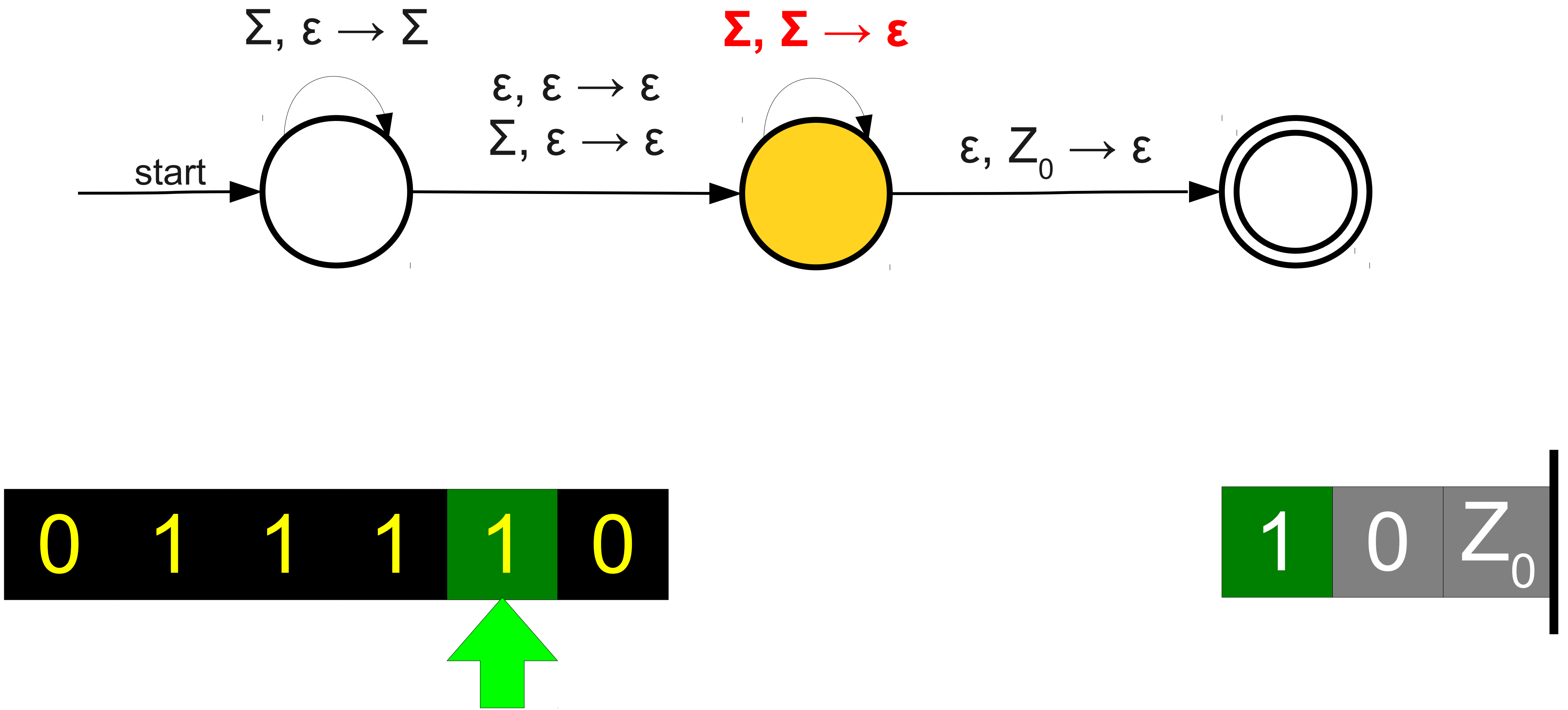
A PDA for Palindromes



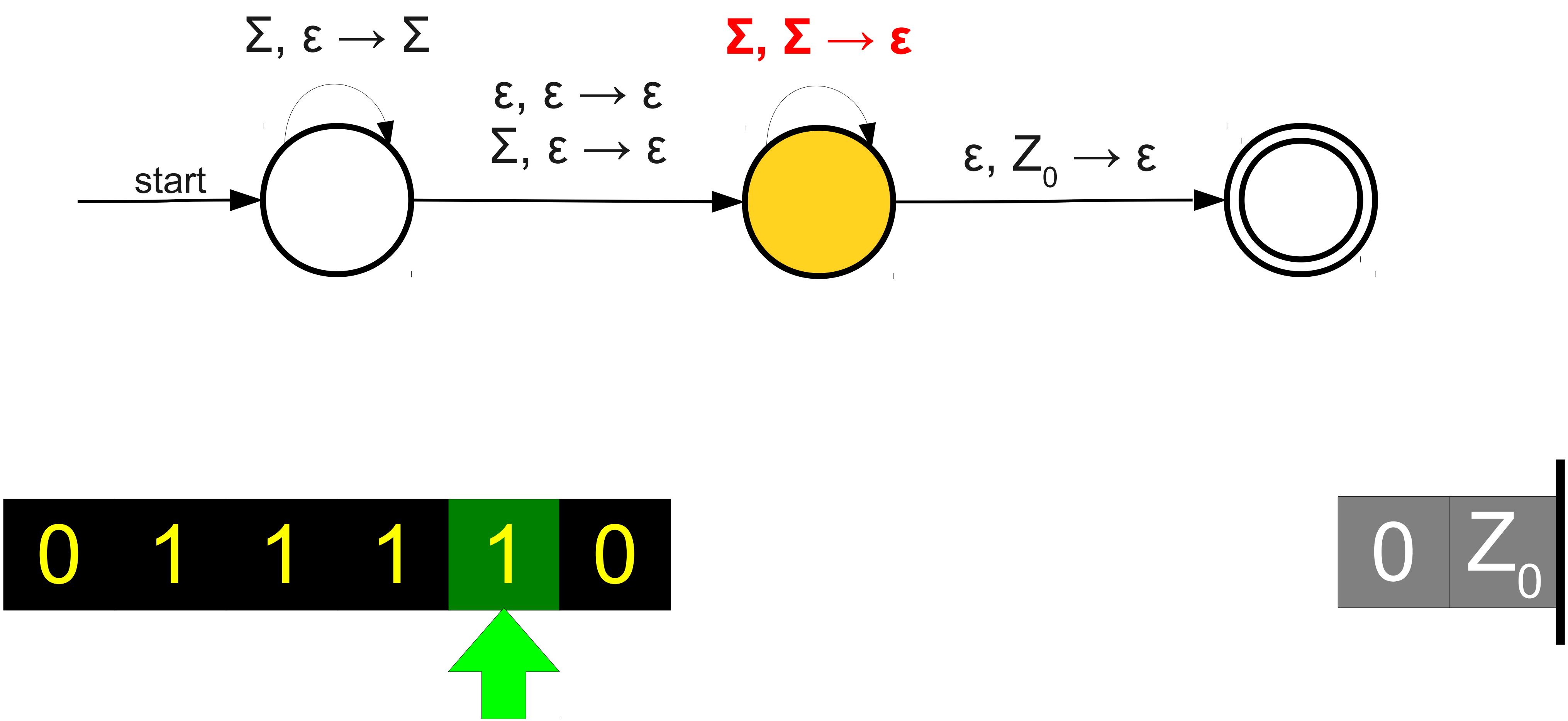
A PDA for Palindromes



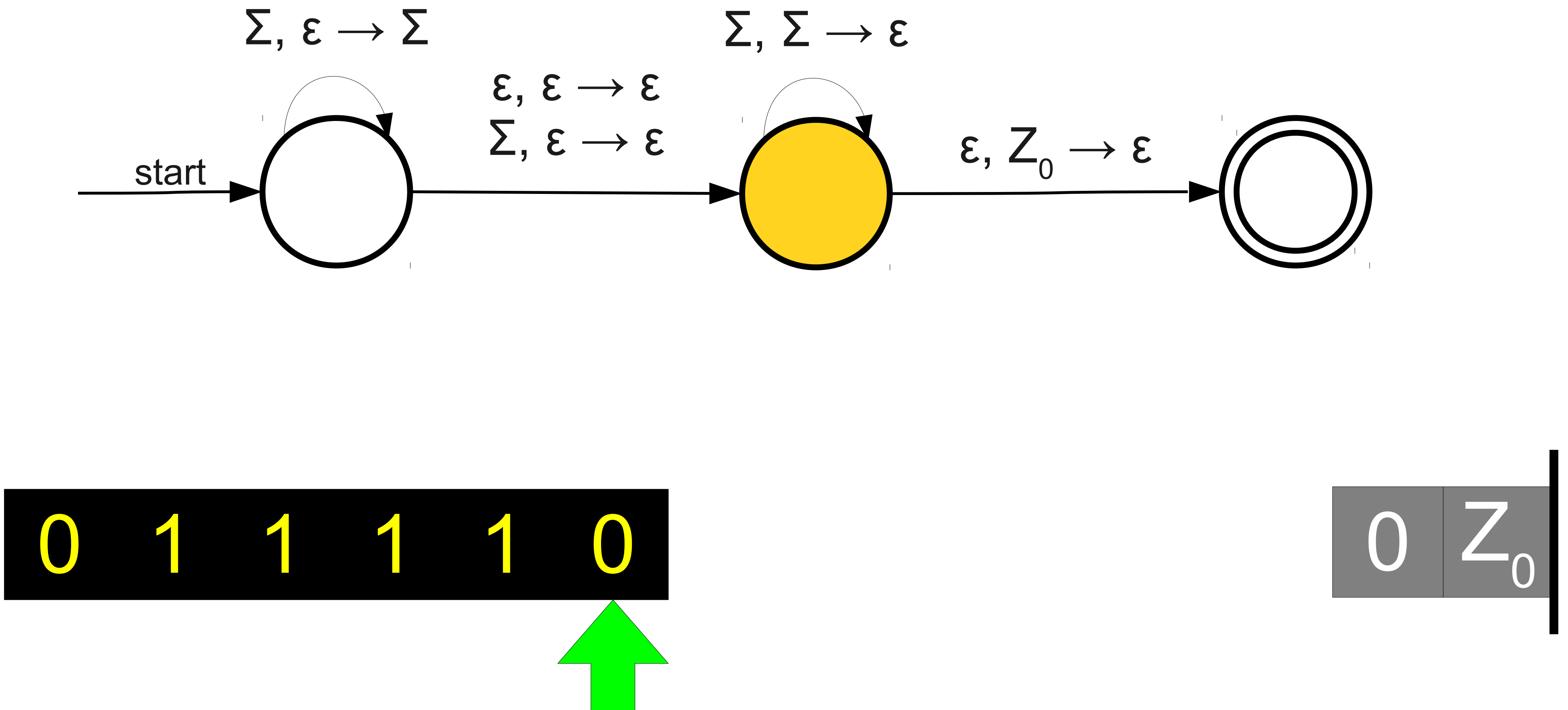
A PDA for Palindromes



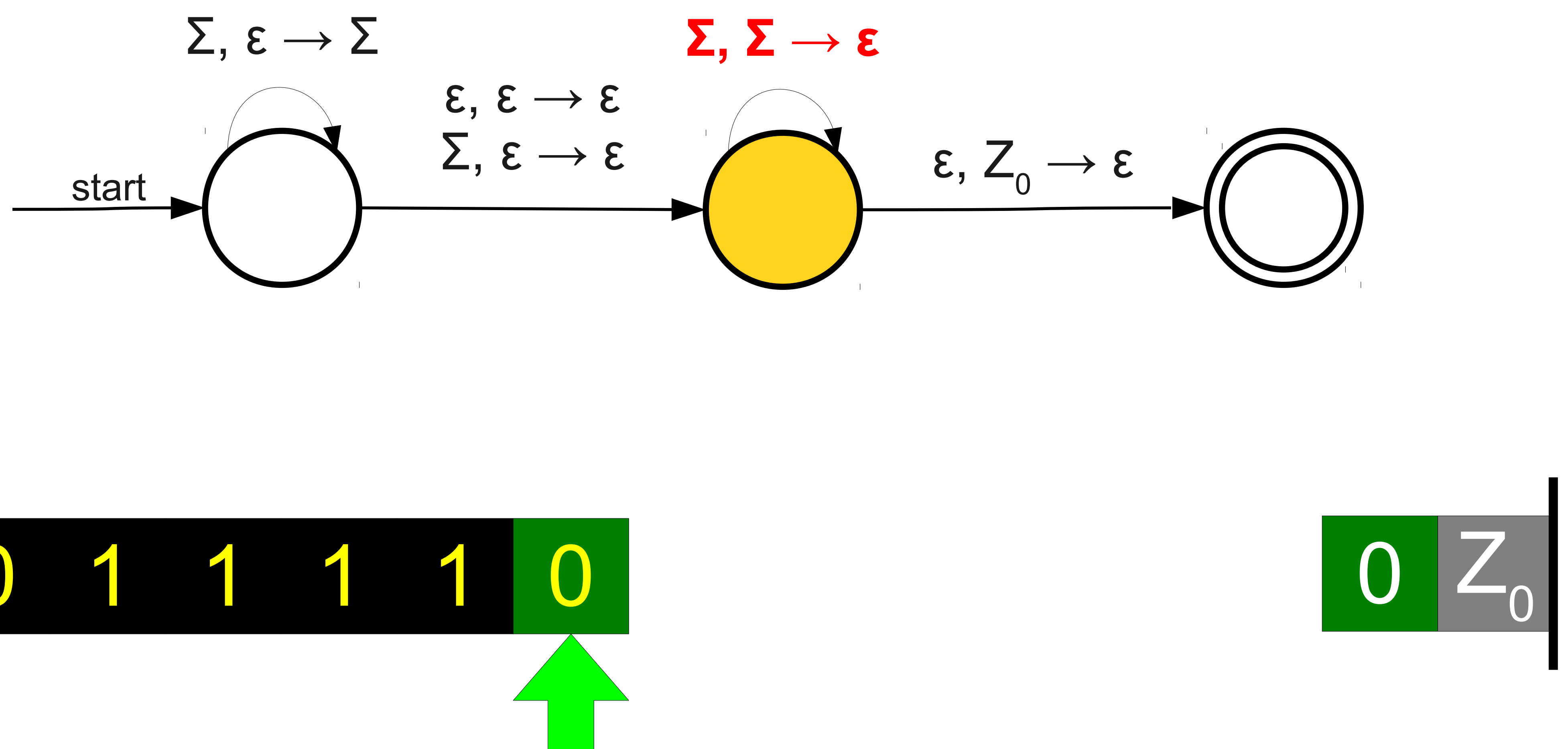
A PDA for Palindromes



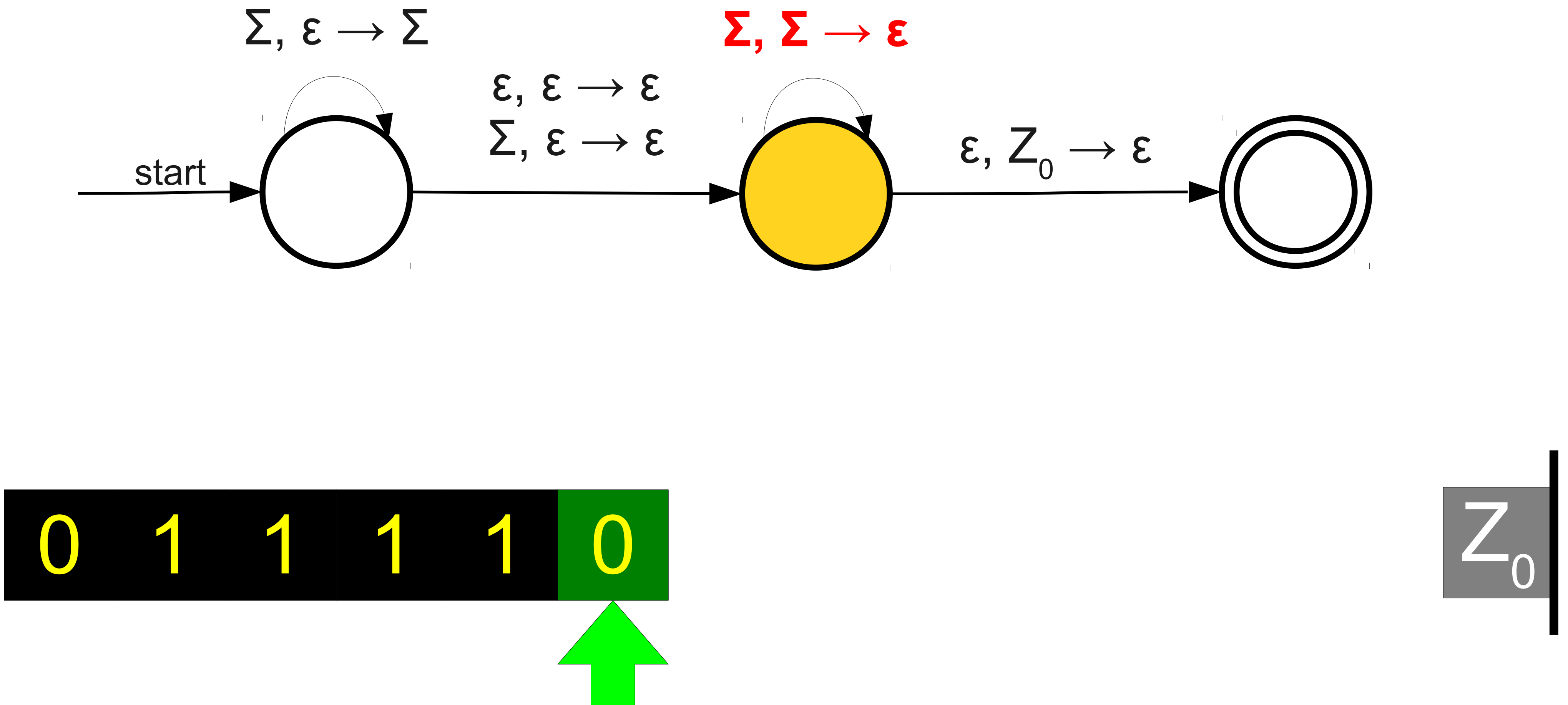
A PDA for Palindromes



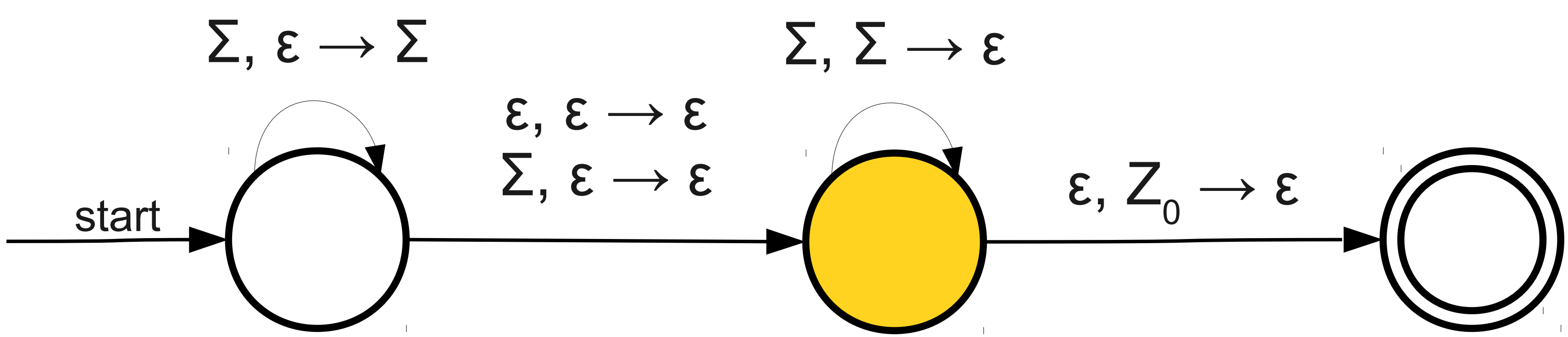
A PDA for Palindromes

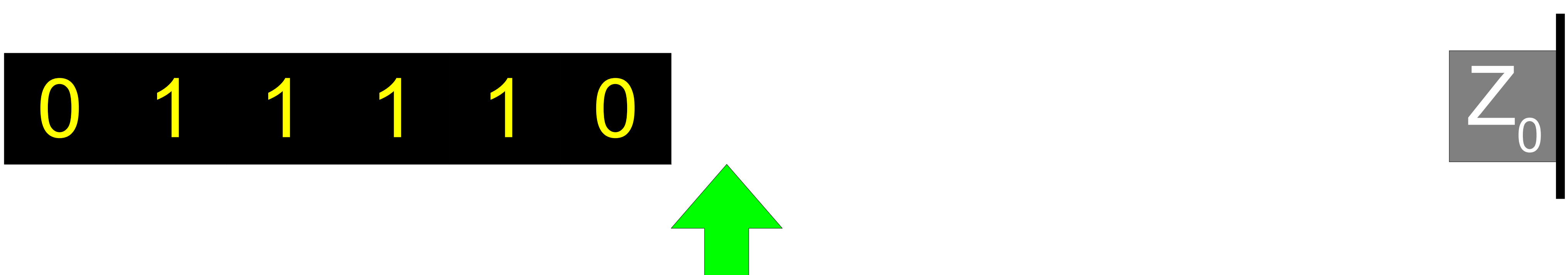


A PDA for Palindromes

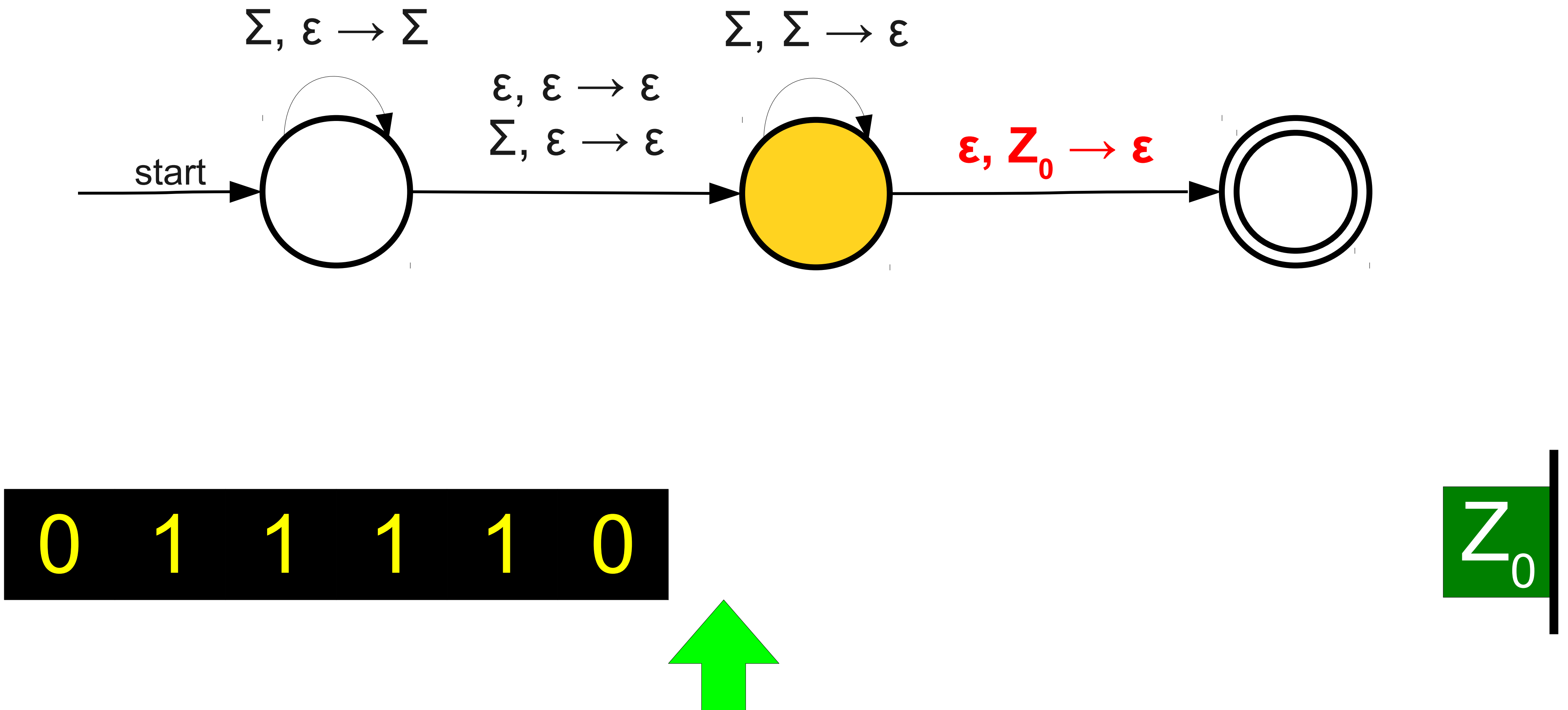


A PDA for Palindromes

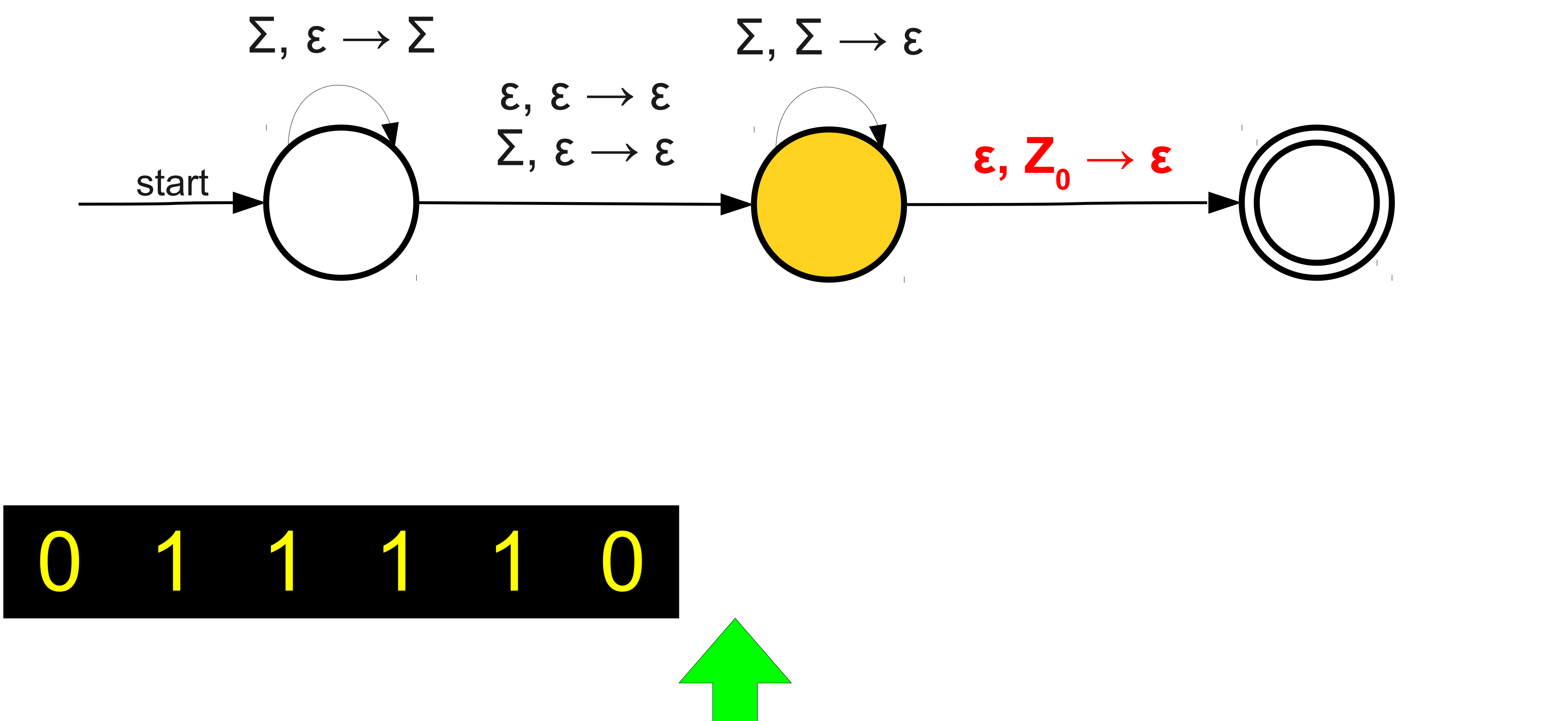




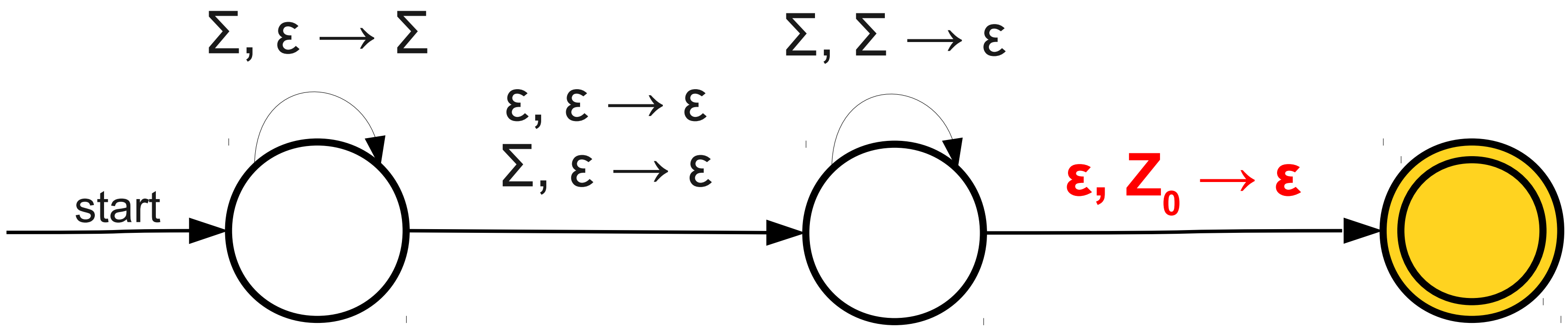
A PDA for Palindromes

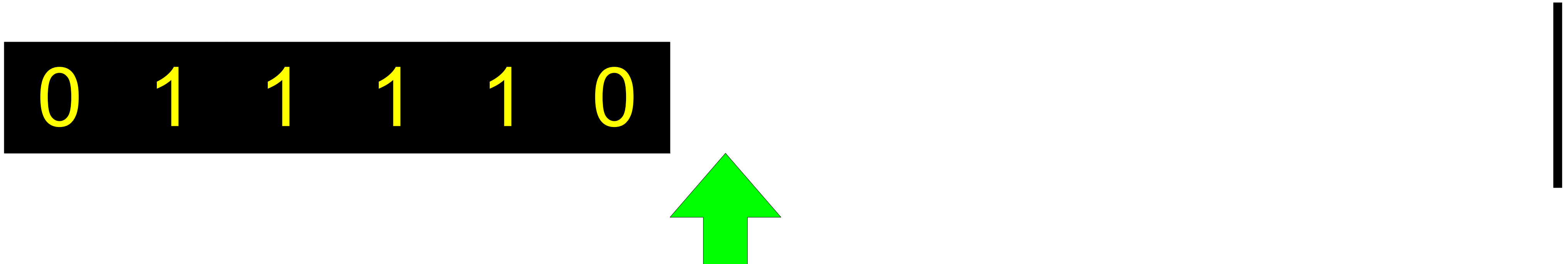


A PDA for Palindromes

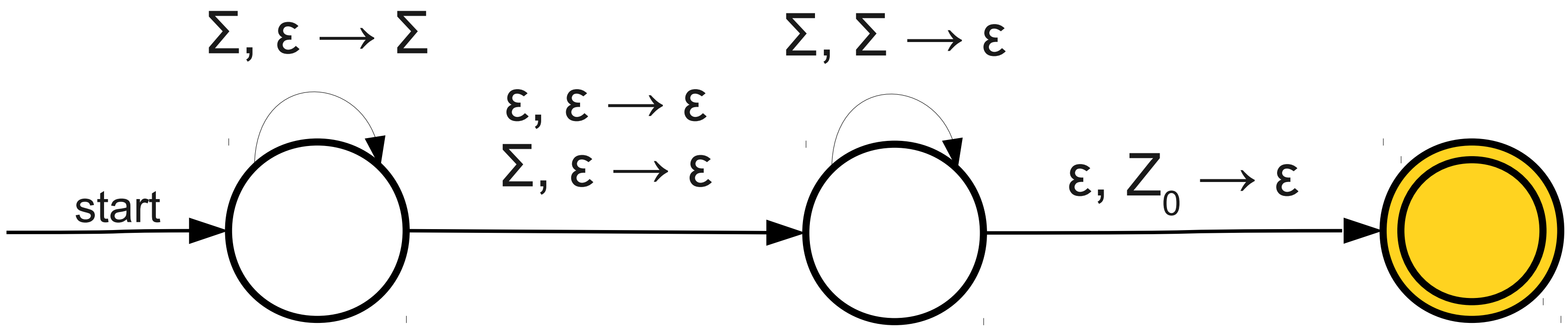


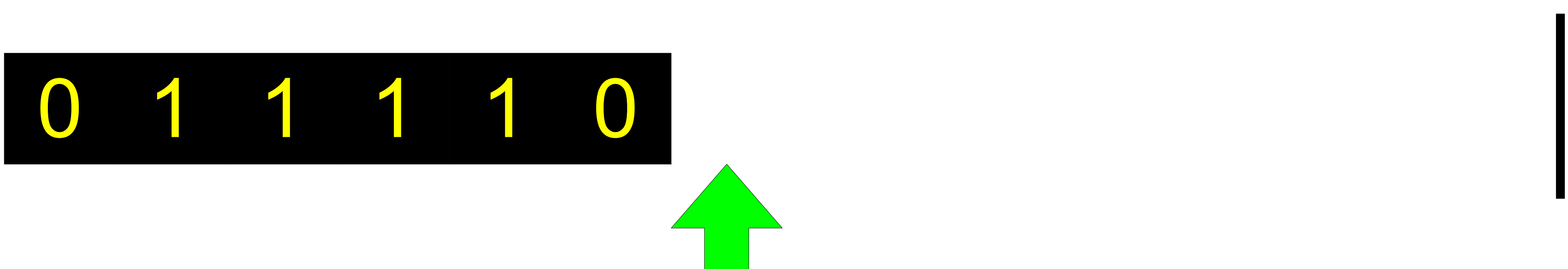
A PDA for Palindromes



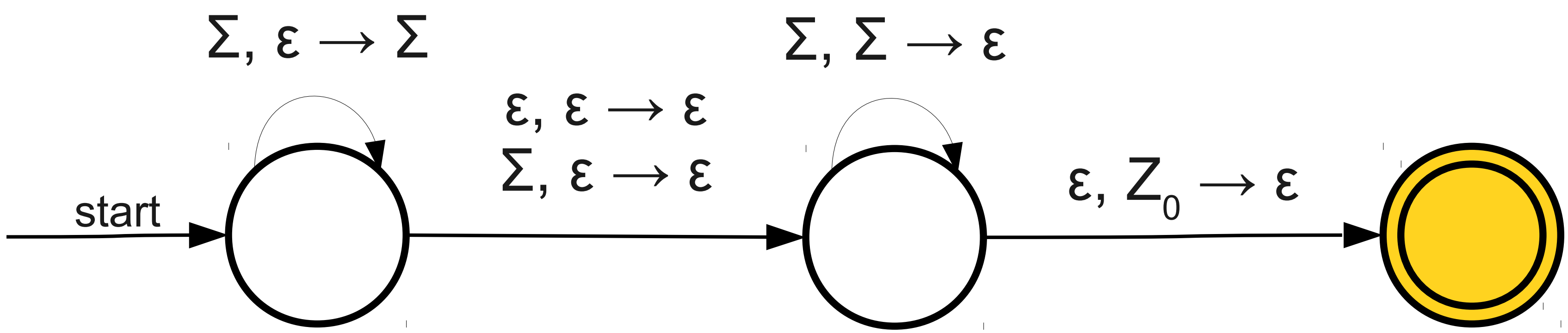


A PDA for Palindromes



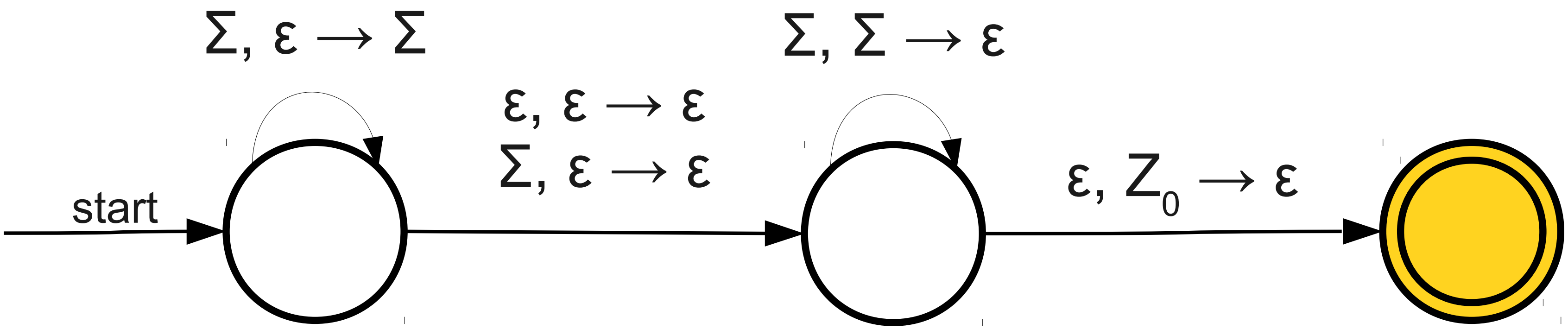


A PDA for Palindromes

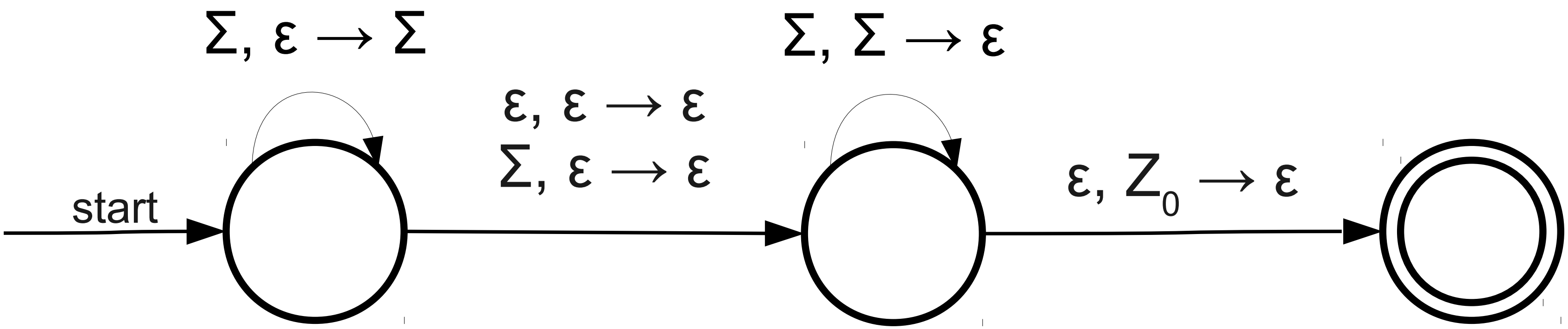


0 1 1 1 1 0

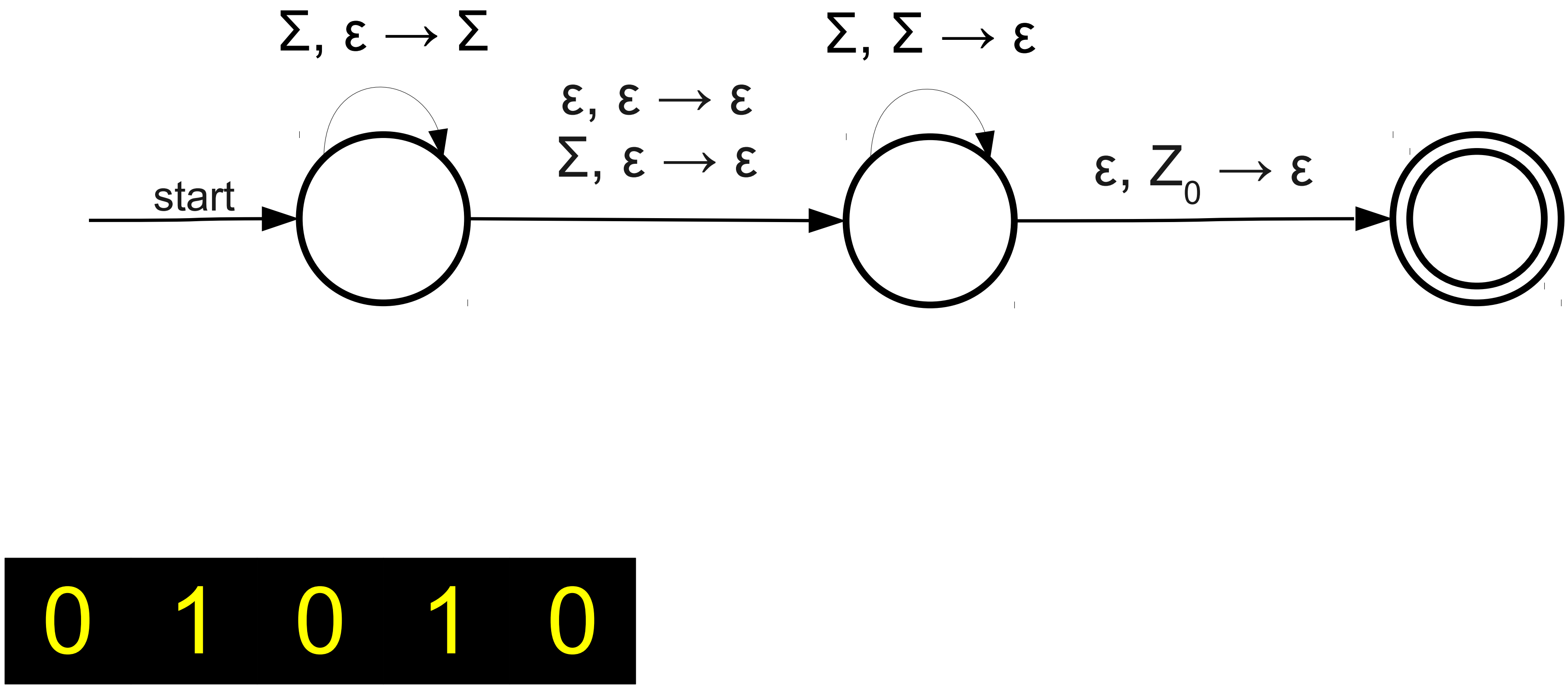
A PDA for Palindromes



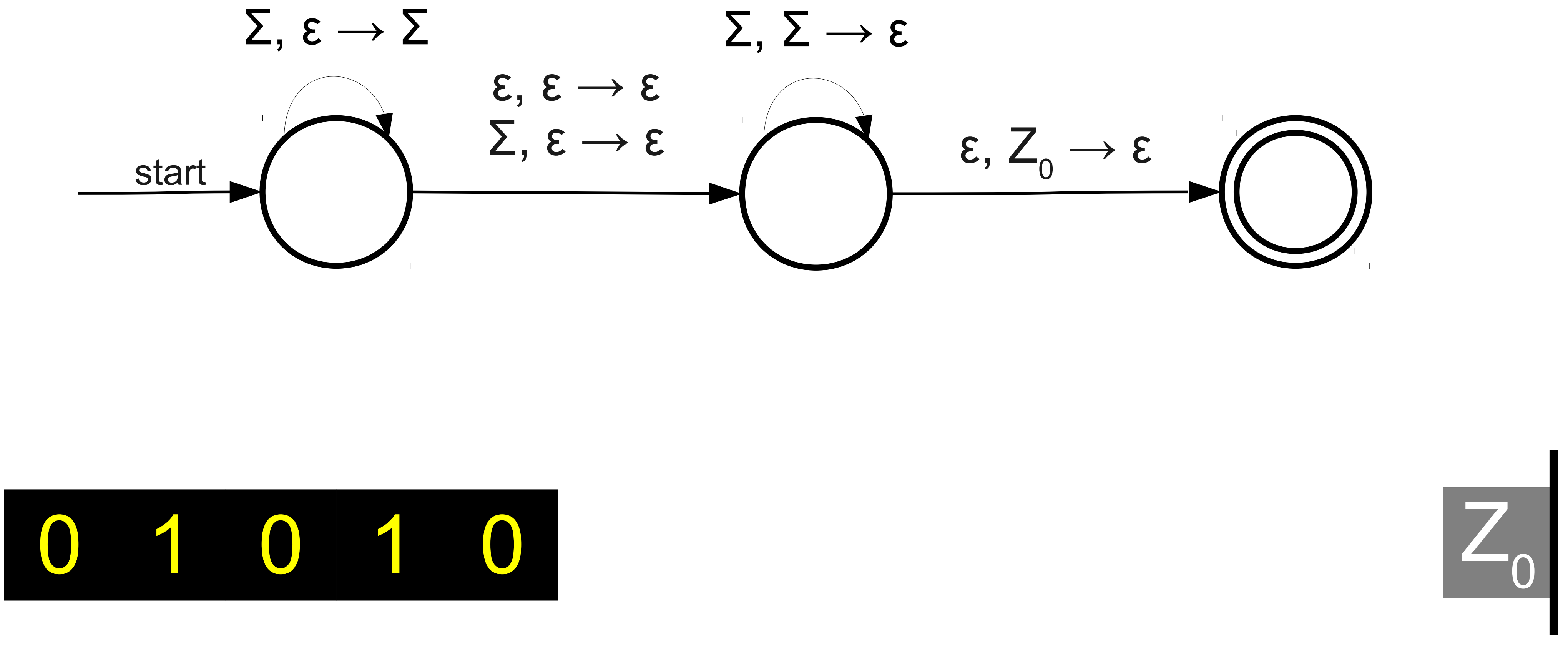
A PDA for Palindromes



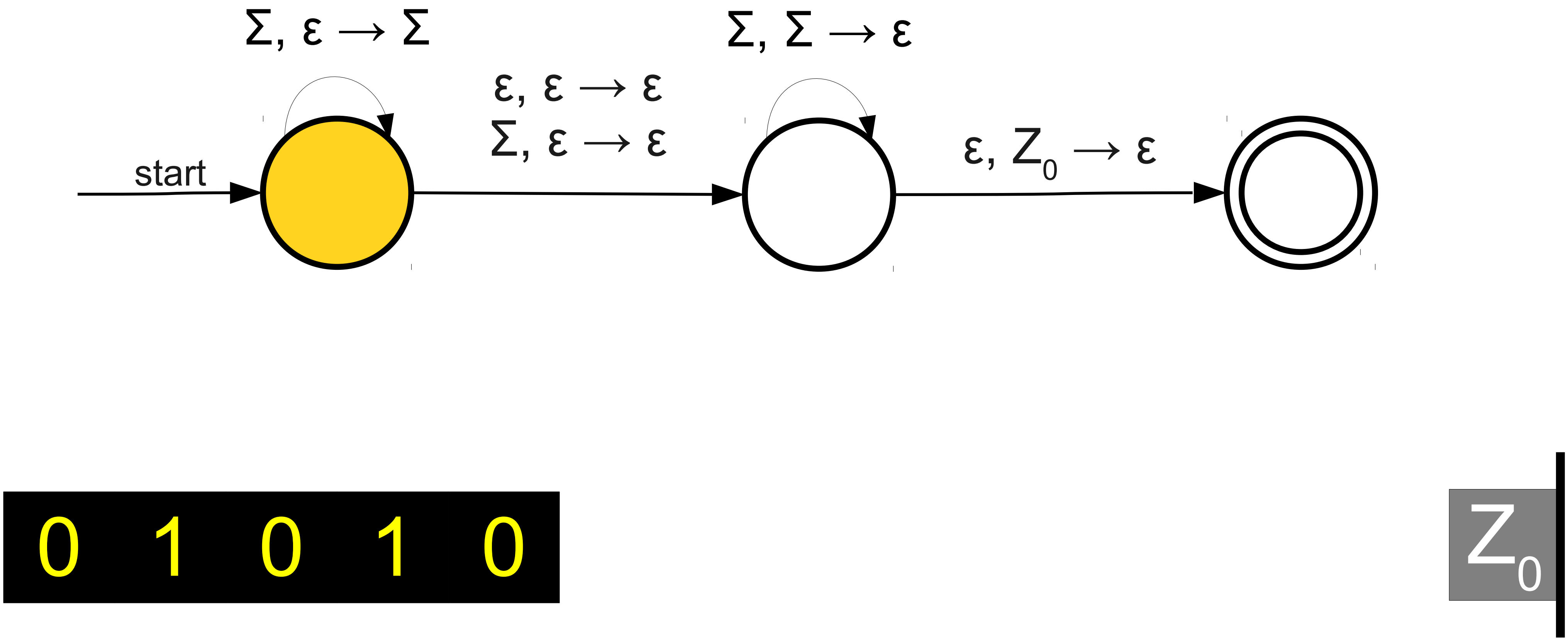
A PDA for Palindromes



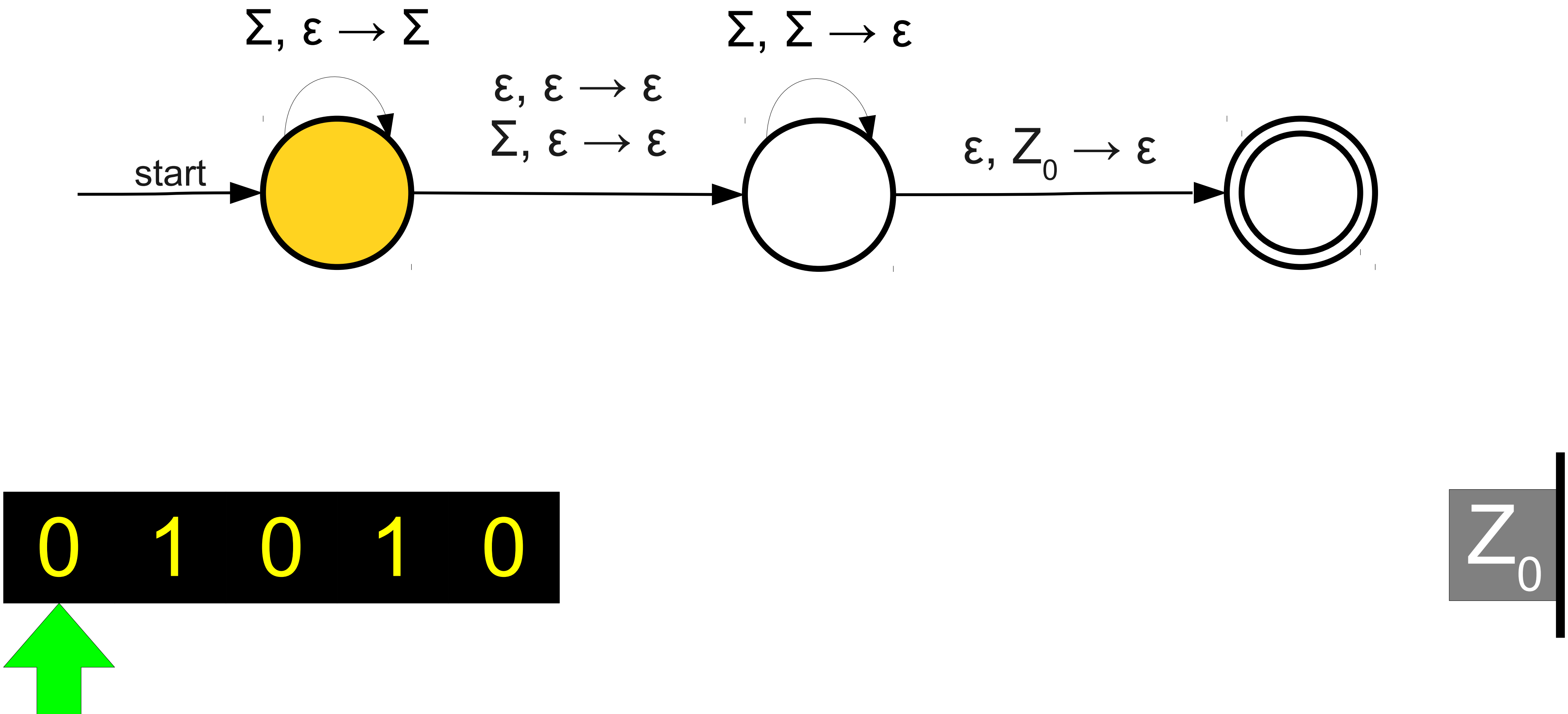
A PDA for Palindromes



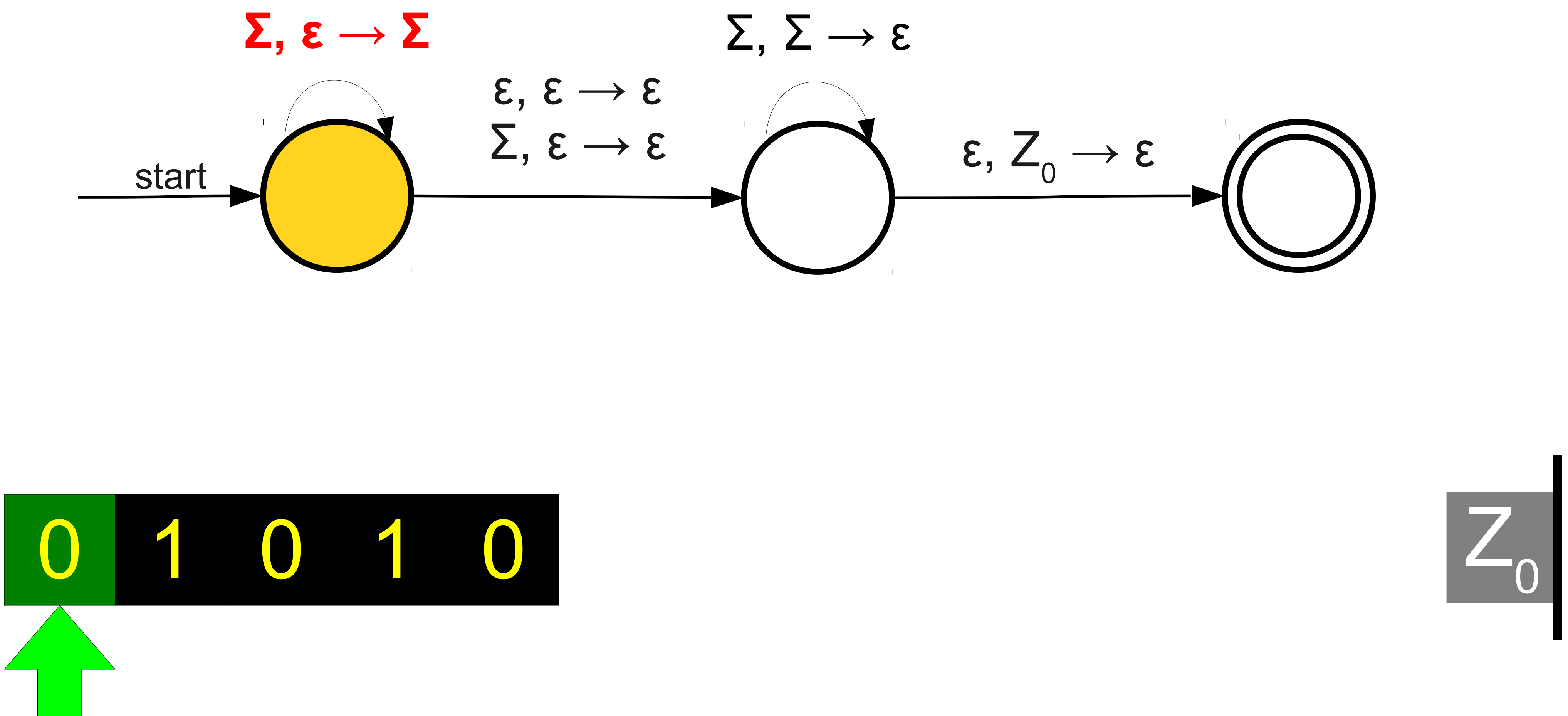
A PDA for Palindromes



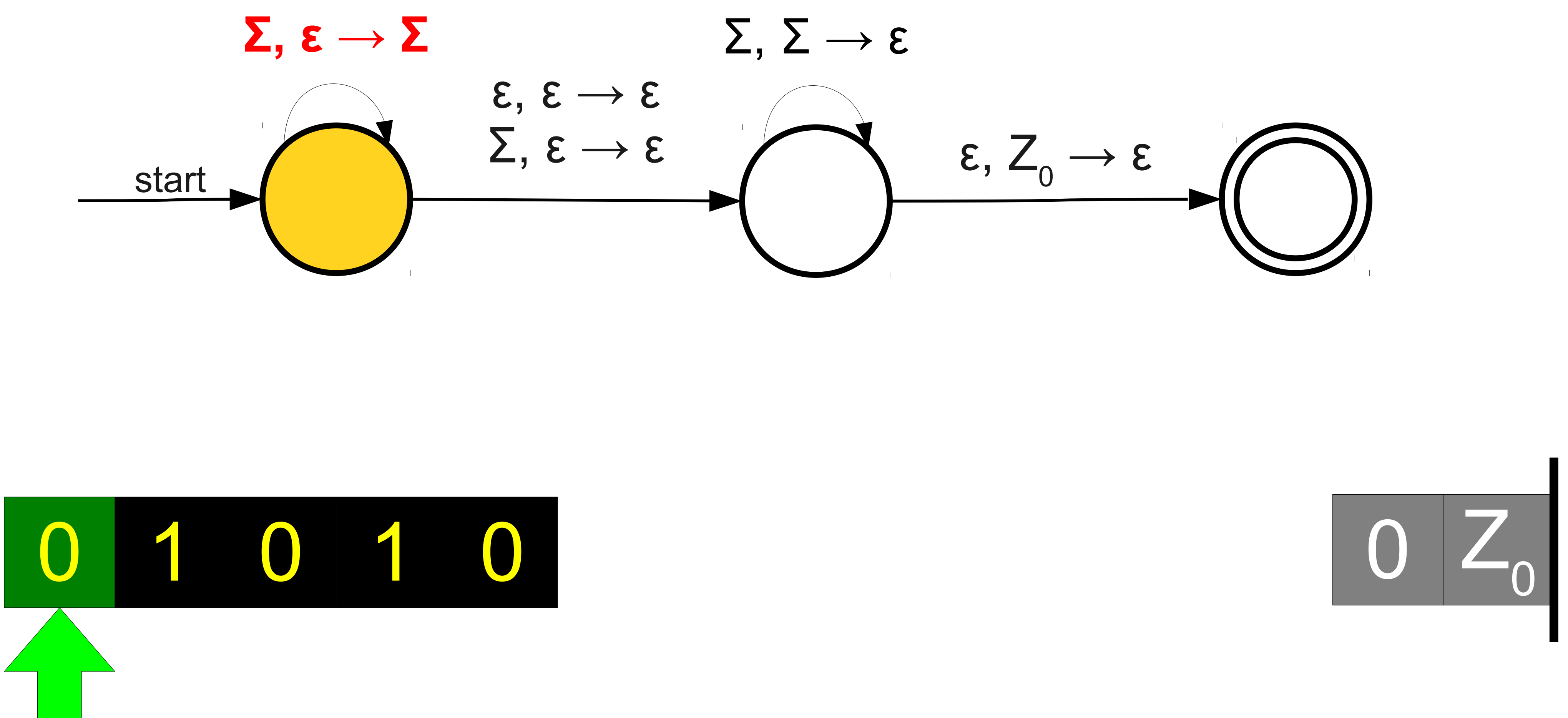
A PDA for Palindromes



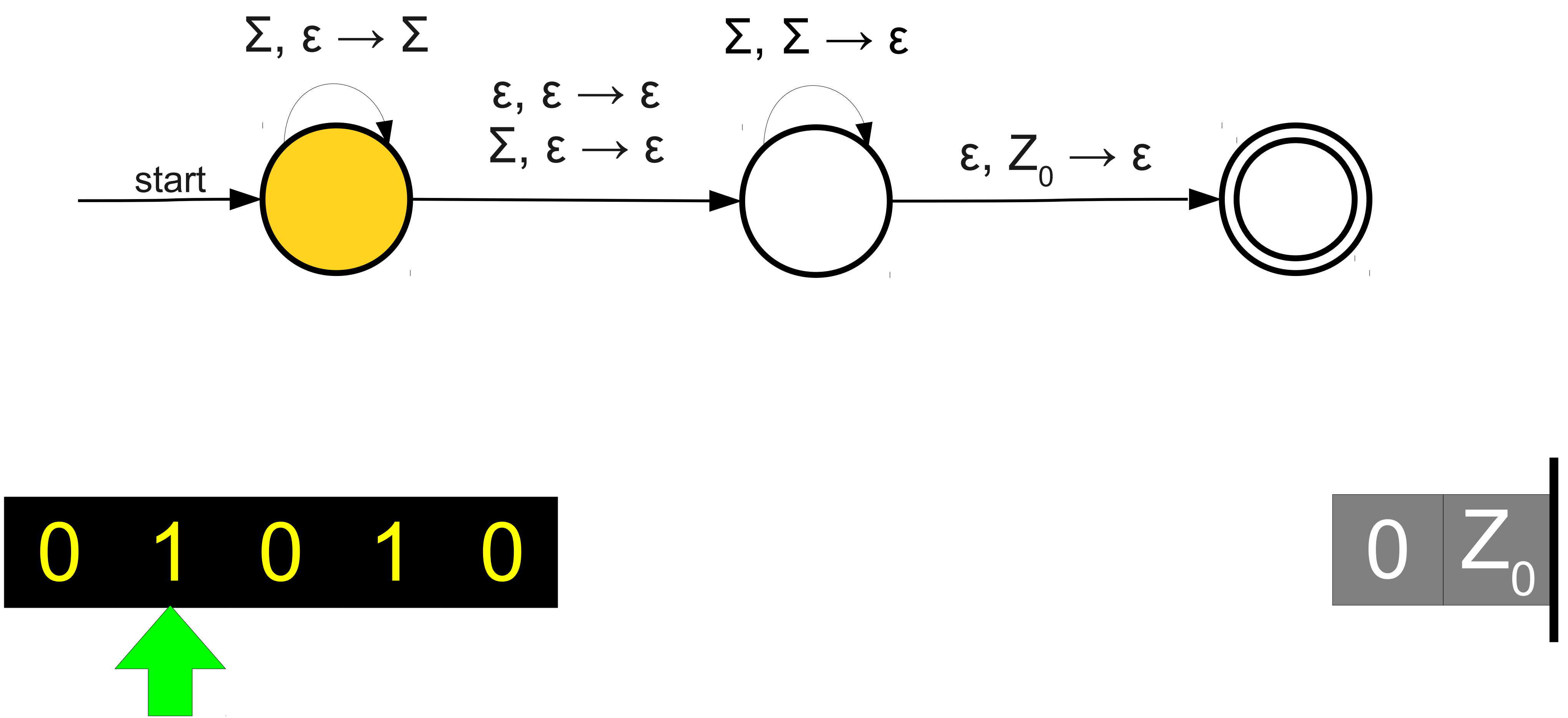
A PDA for Palindromes



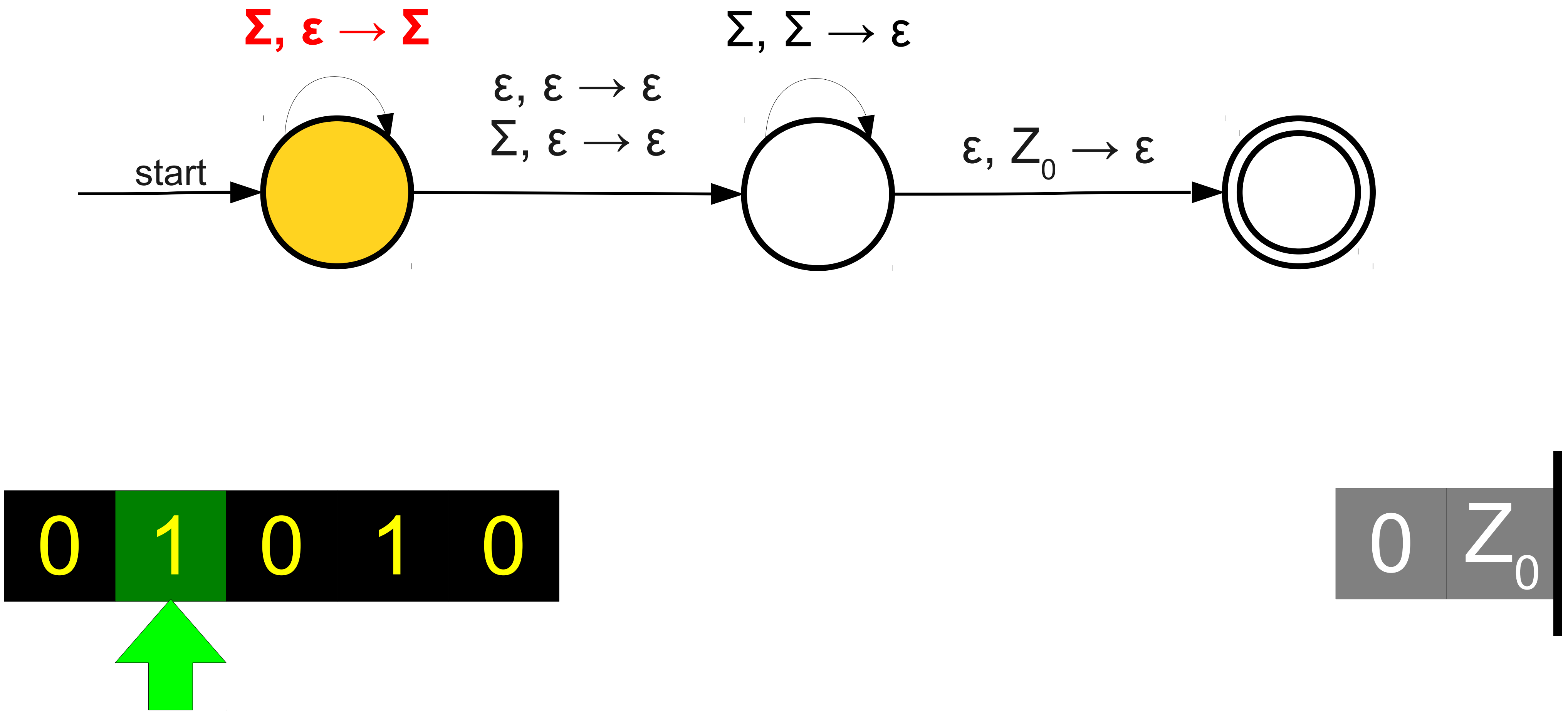
A PDA for Palindromes



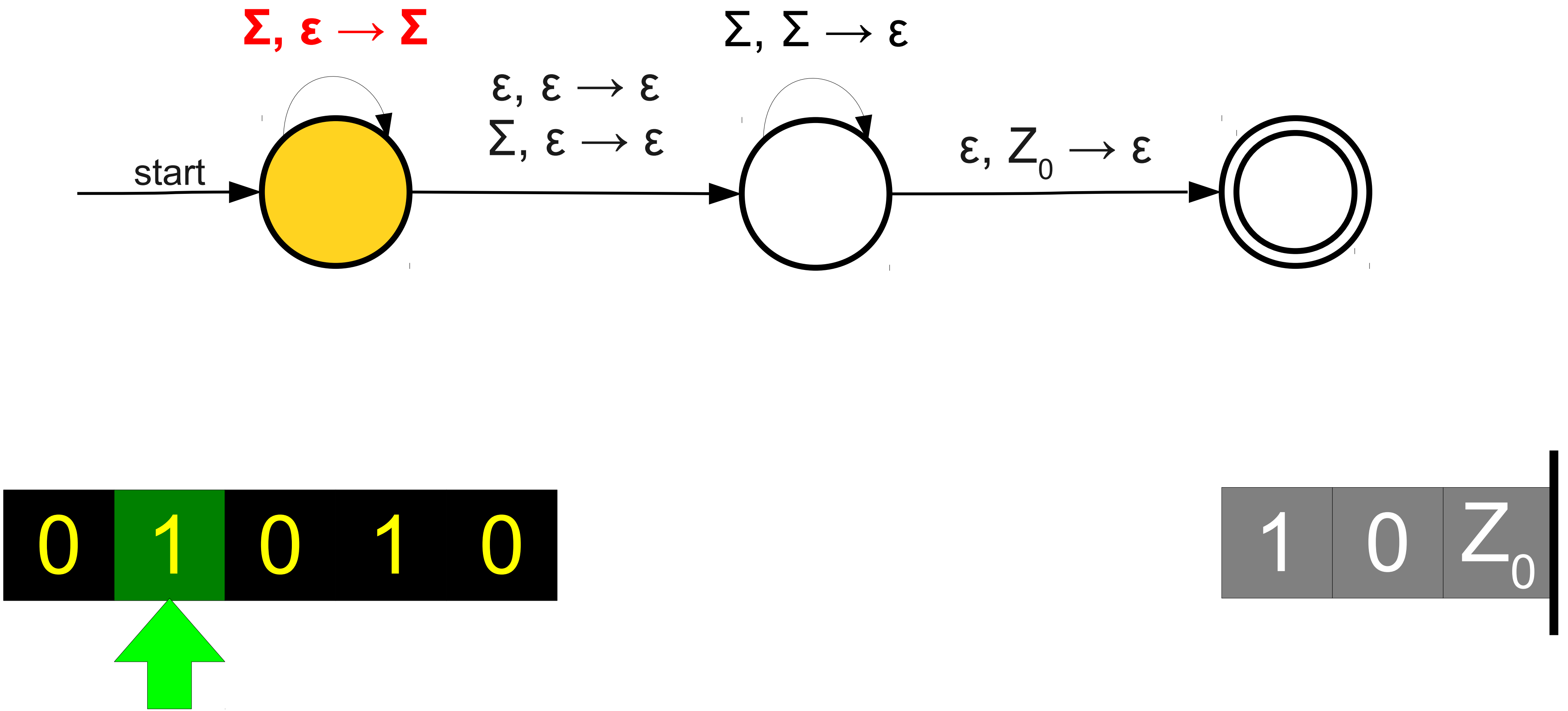
A PDA for Palindromes



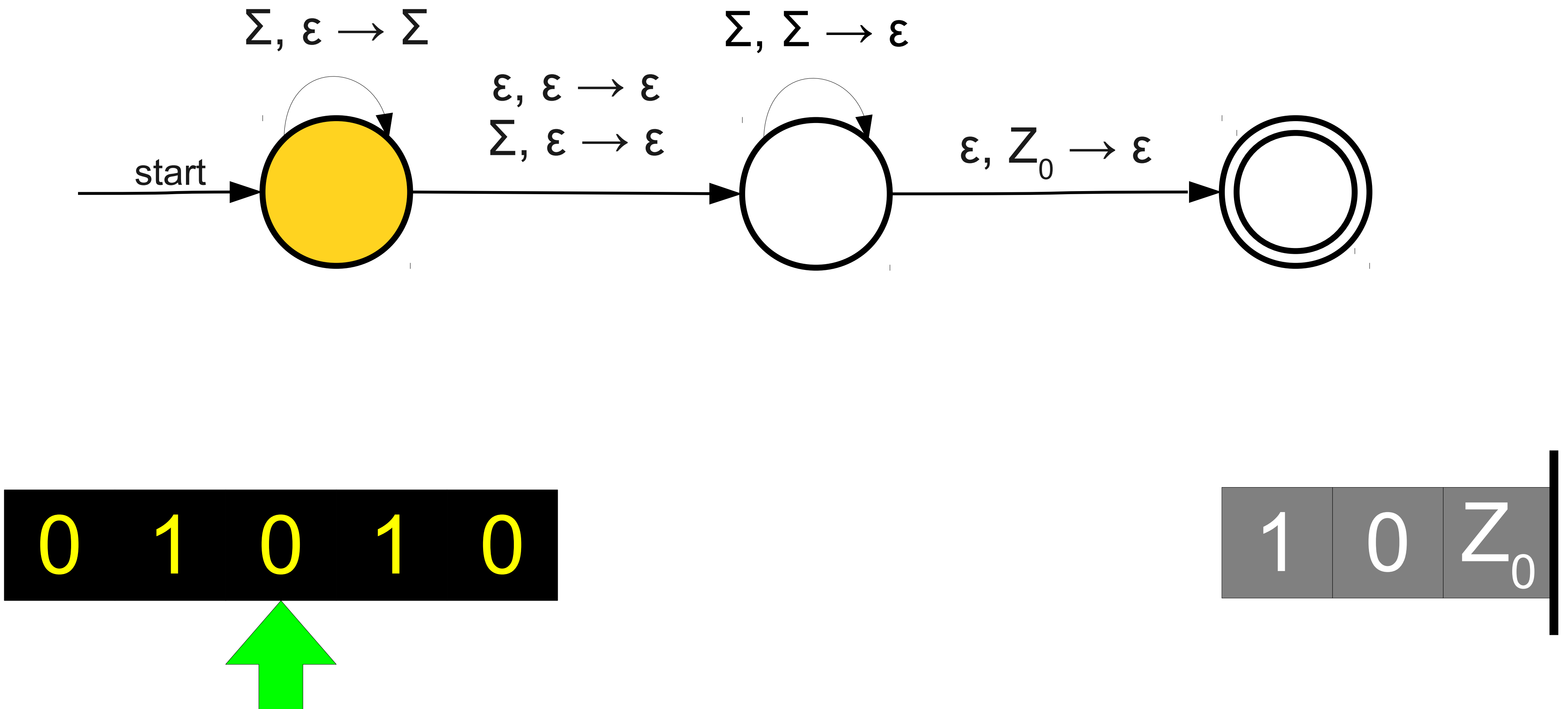
A PDA for Palindromes



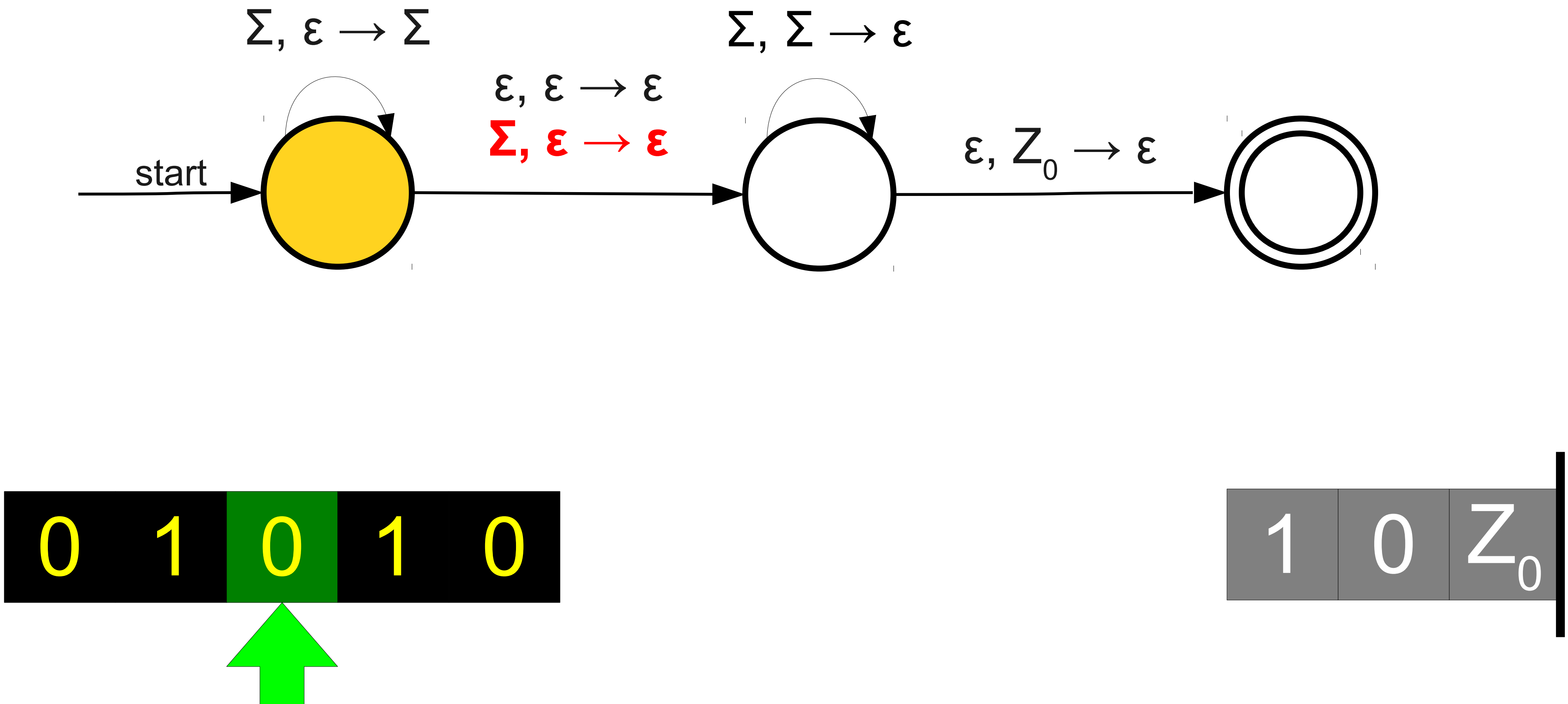
A PDA for Palindromes



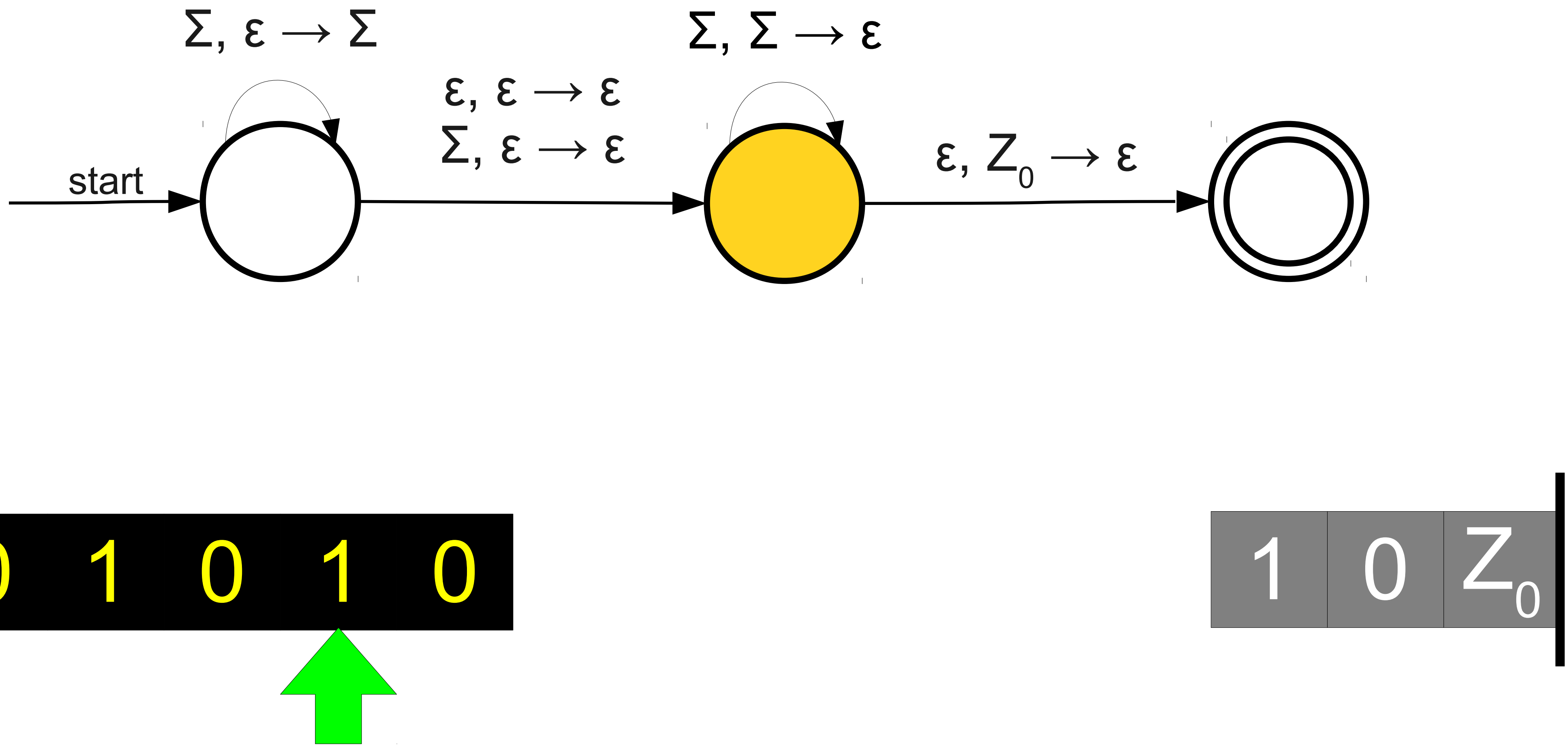
A PDA for Palindromes



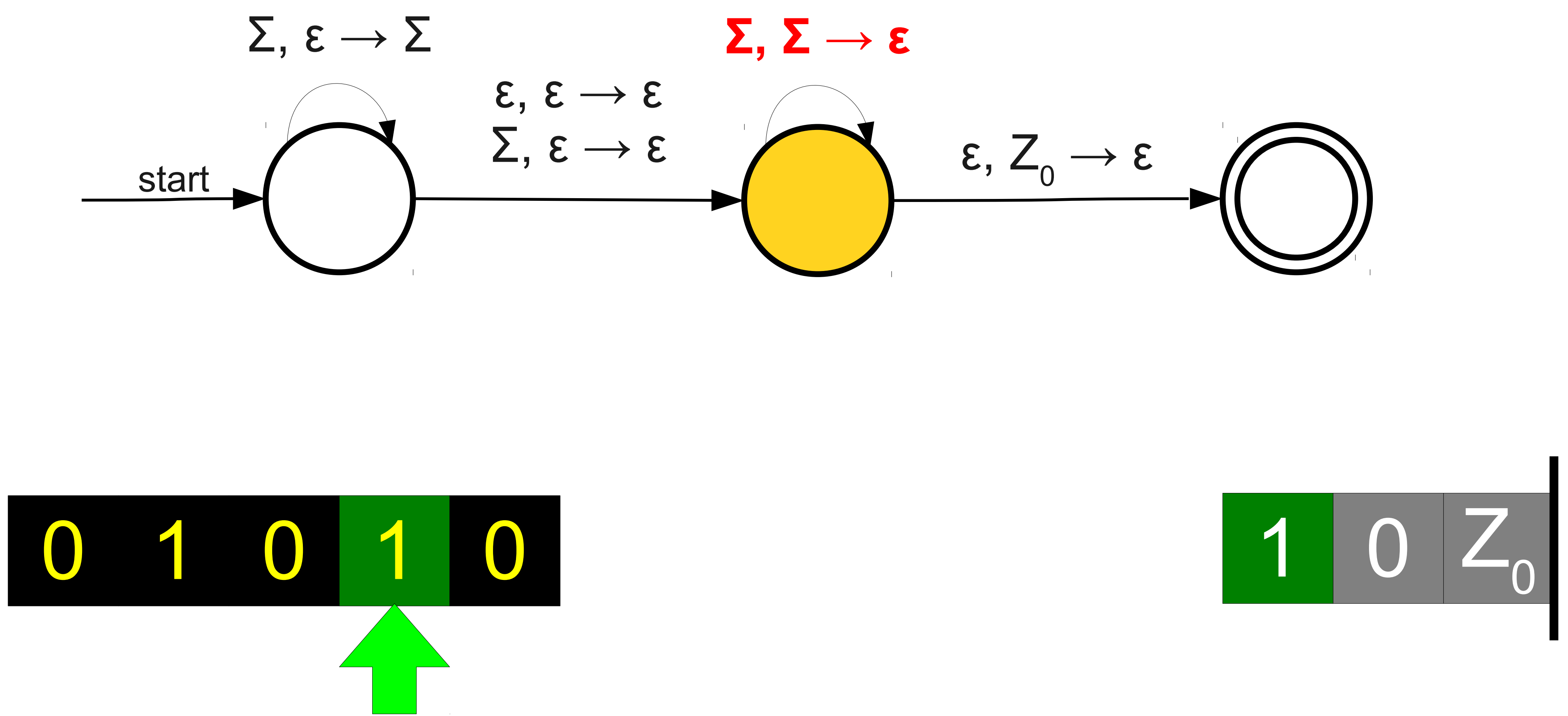
A PDA for Palindromes



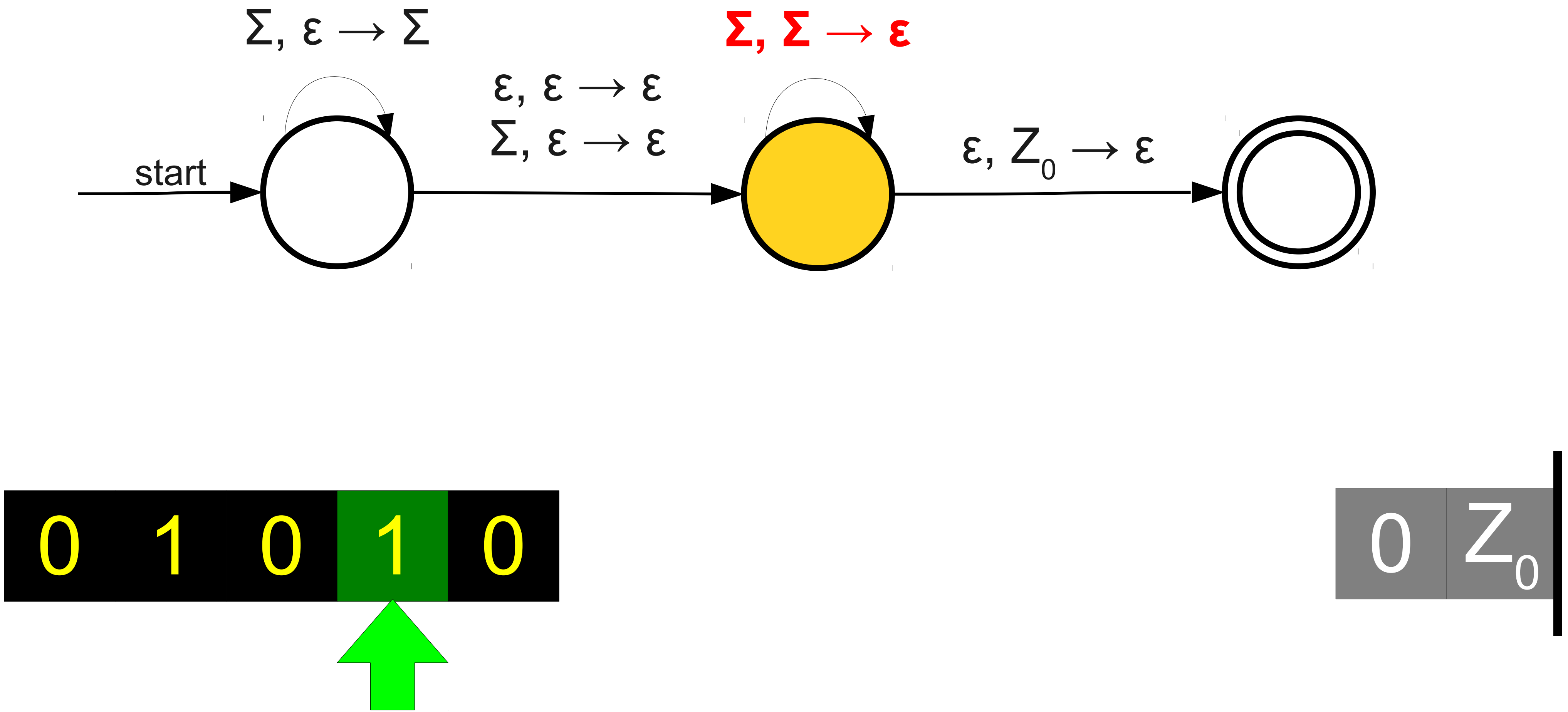
A PDA for Palindromes



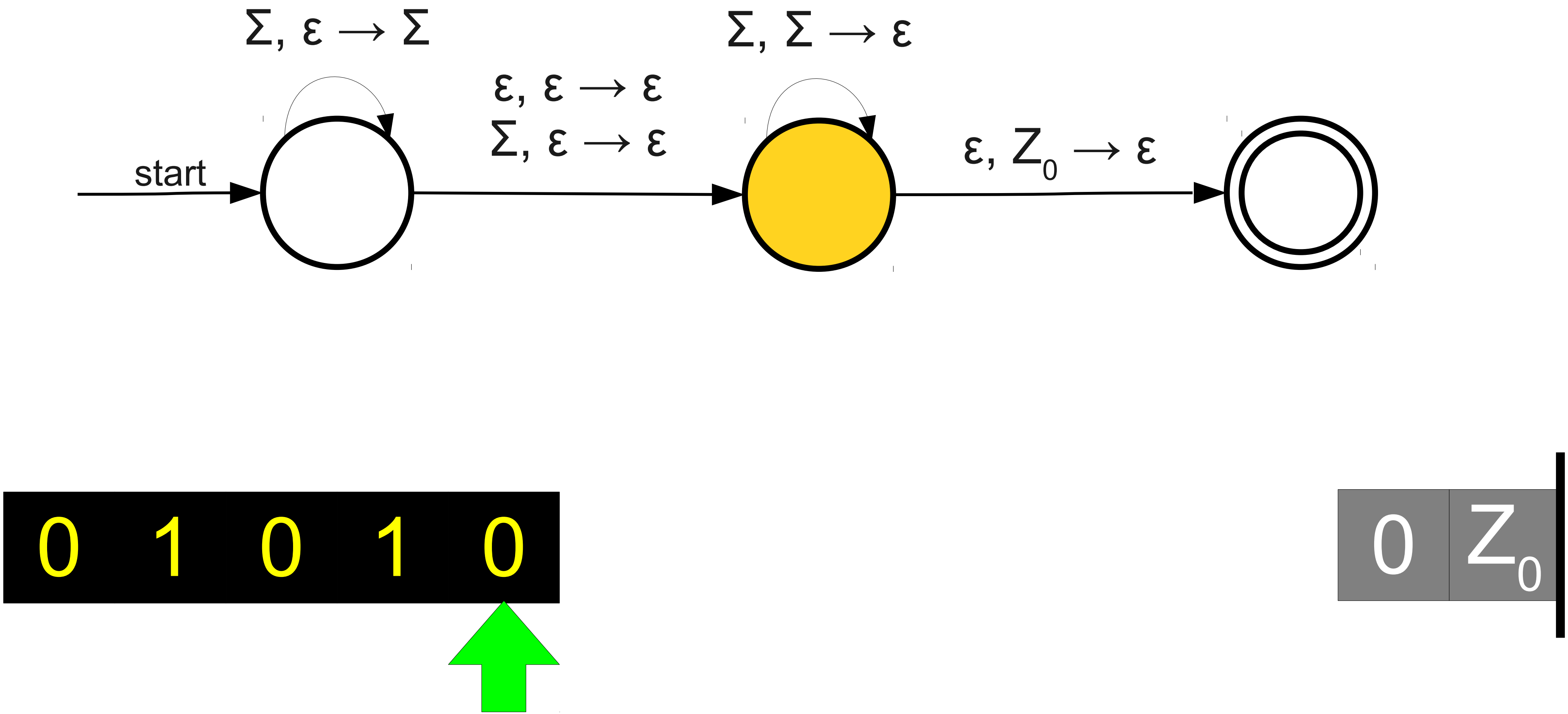
A PDA for Palindromes



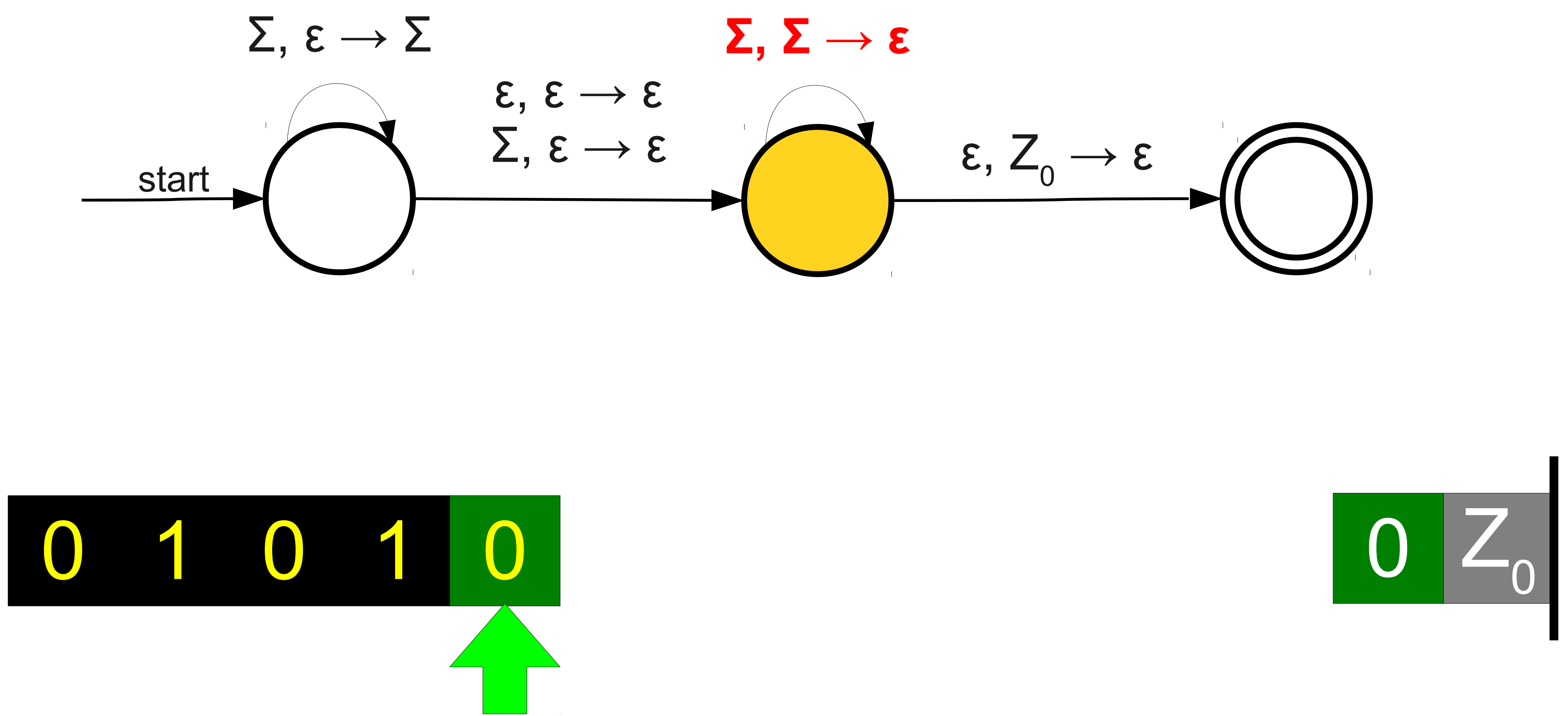
A PDA for Palindromes



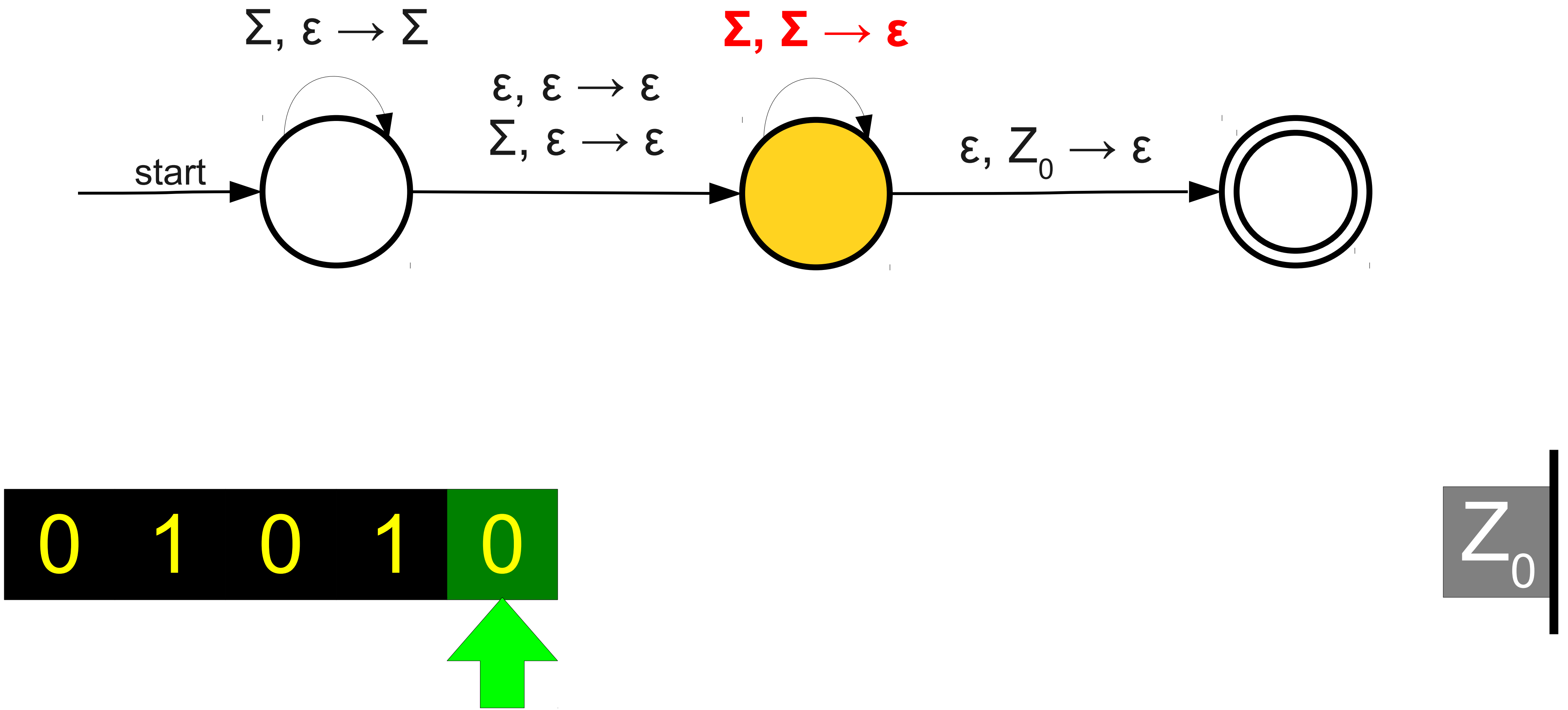
A PDA for Palindromes



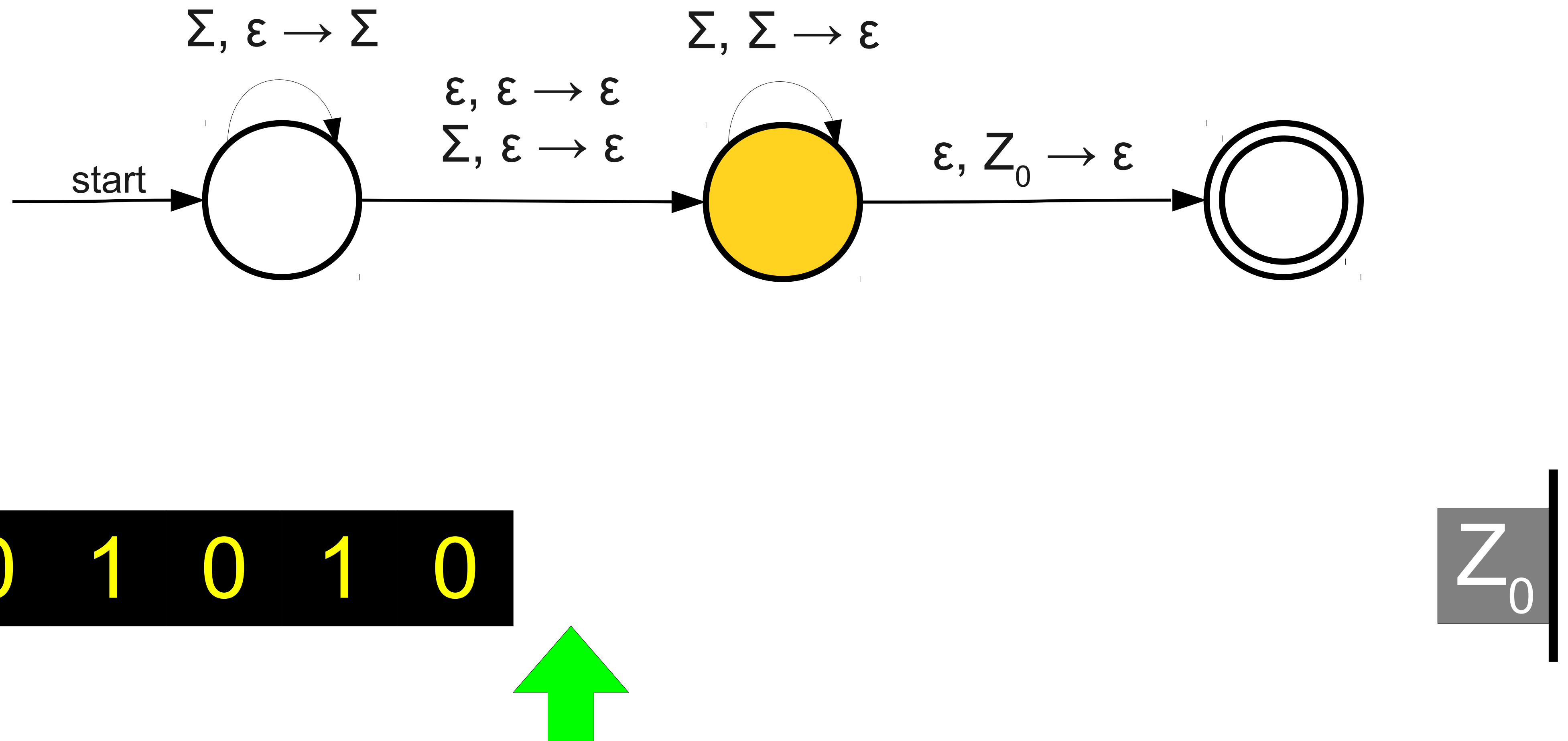
A PDA for Palindromes



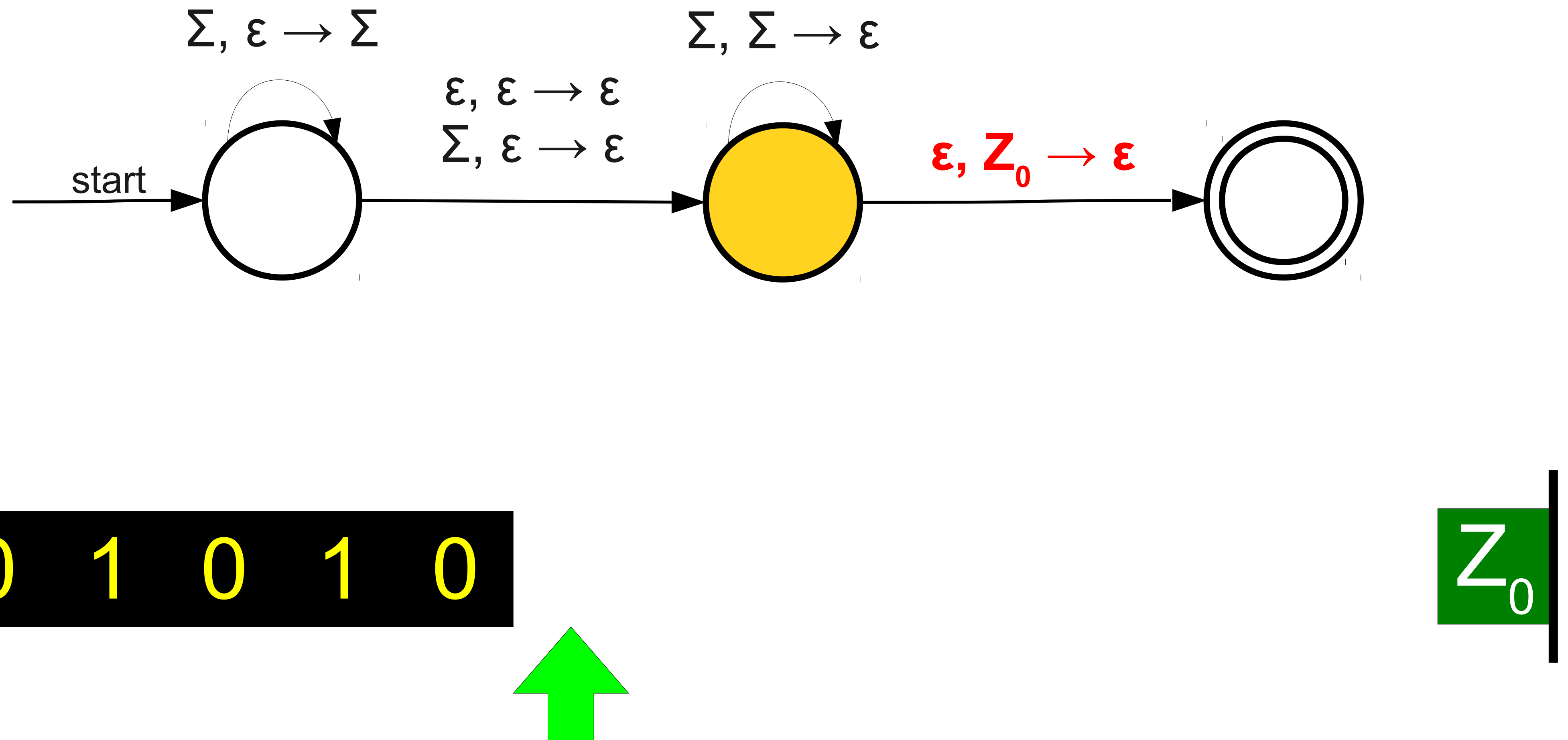
A PDA for Palindromes



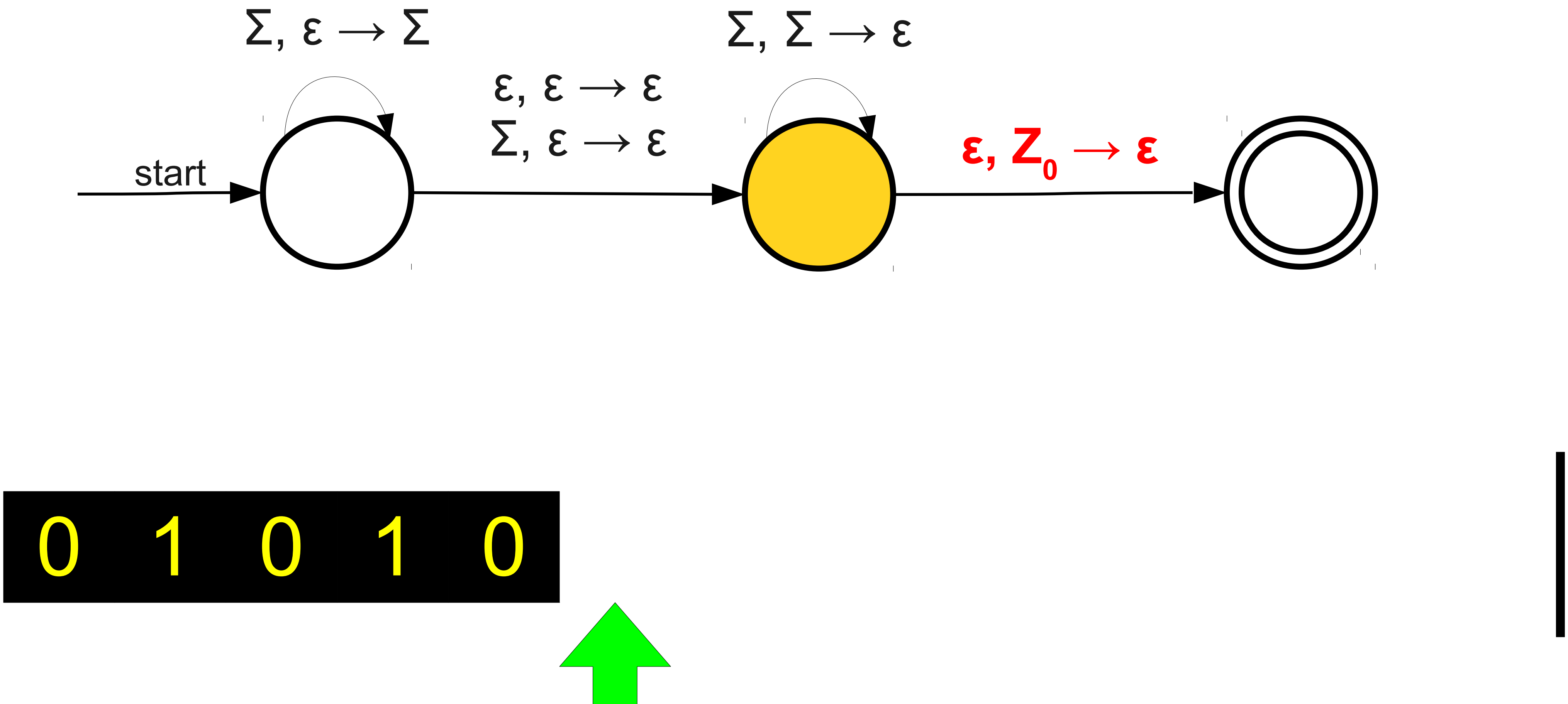
A PDA for Palindromes



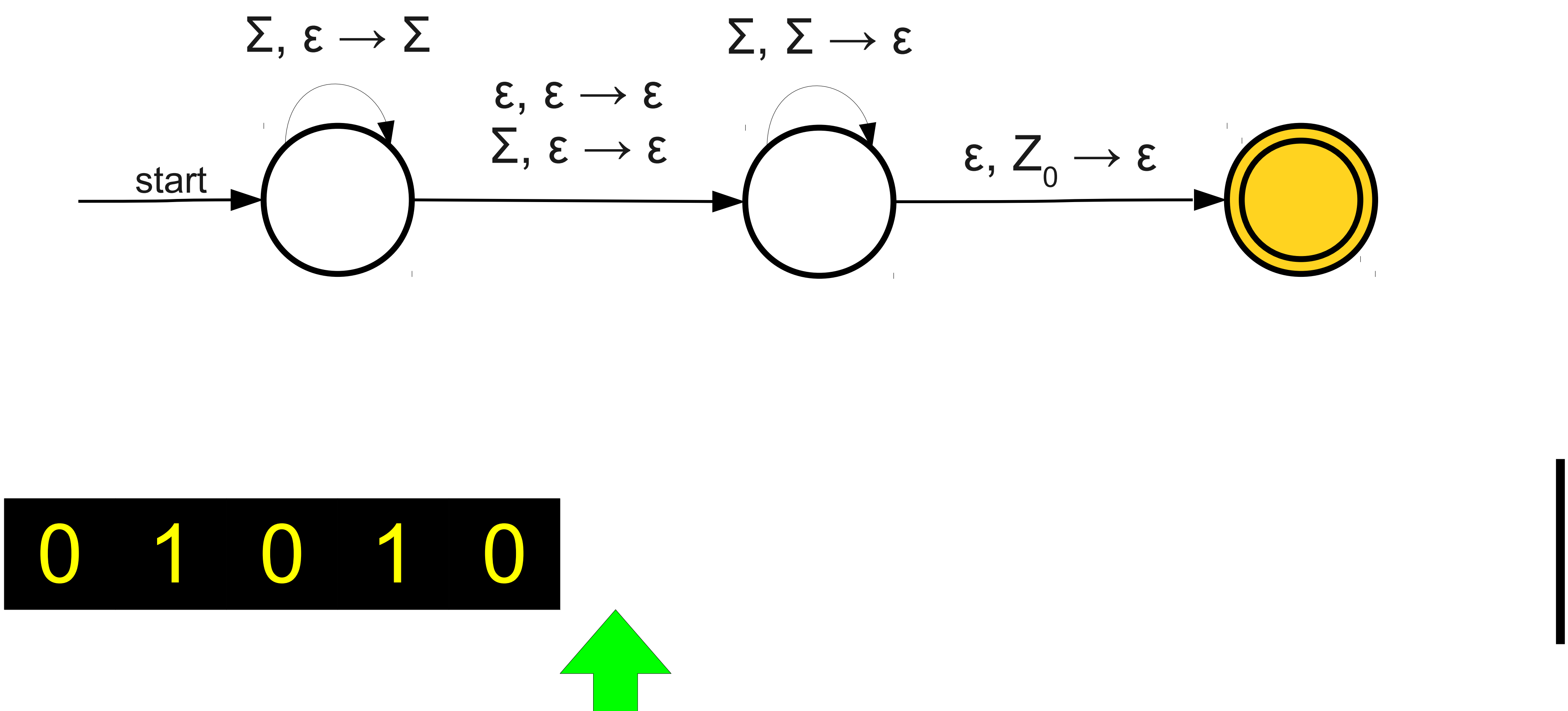
A PDA for Palindromes



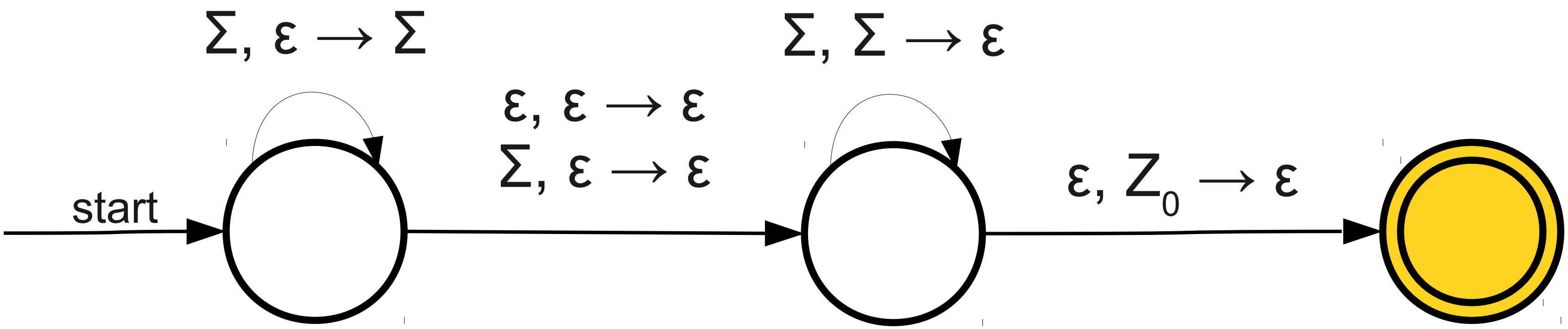
A PDA for Palindromes



A PDA for Palindromes

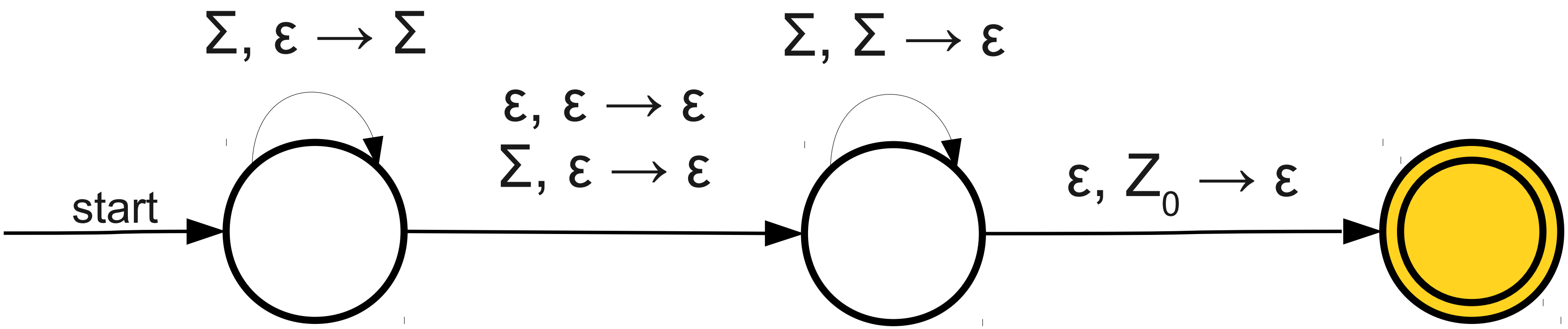


A PDA for Palindromes



0 1 0 1 0

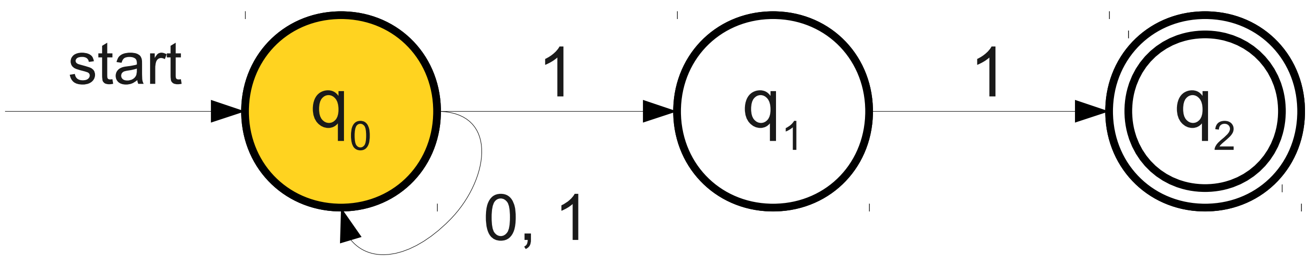
A PDA for Palindromes



0 1 0 1 0

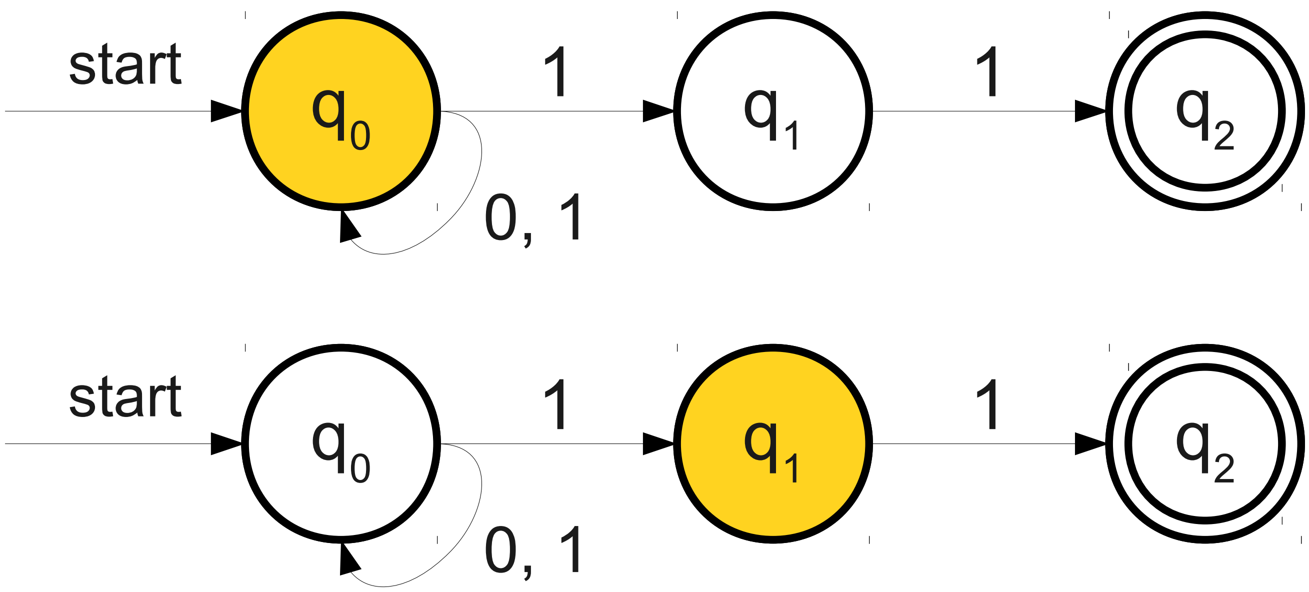
A Note on Nondeterminism

- In an NFA, we could interpret nondeterminism as being in multiple states simultaneously.
- This is only possible because NFAs have no extra storage.



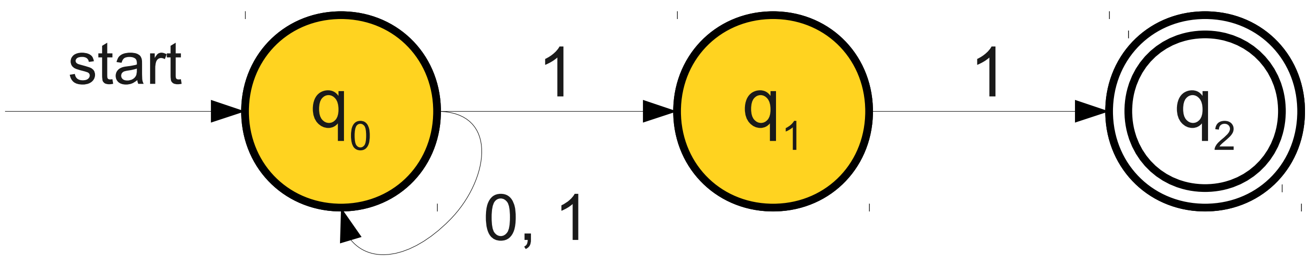
A Note on Nondeterminism

- 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.

A PDA for Arithmetic

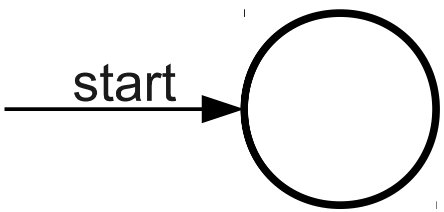
- Let Σ = { `int`, `+`, `*`, `(`, `)` } and consider the language

  ARITH = { *w* ∈ Σ* | *w* is a legal arithmetic expression }

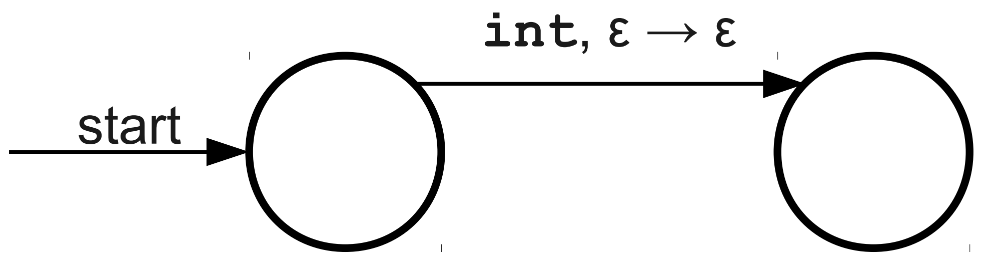
- Examples:
  - `int + int * int`
  - `((int + int) * (int + int)) + (int)`
- Can we build a PDA for *ARITH*?

A PDA for Arithmetic

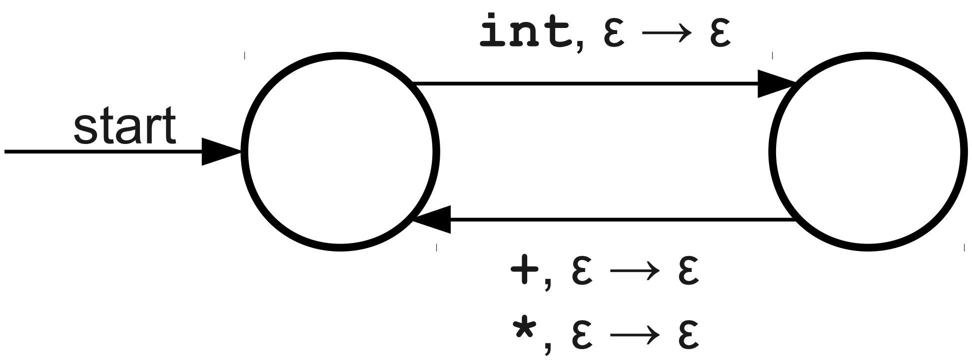
A PDA for Arithmetic



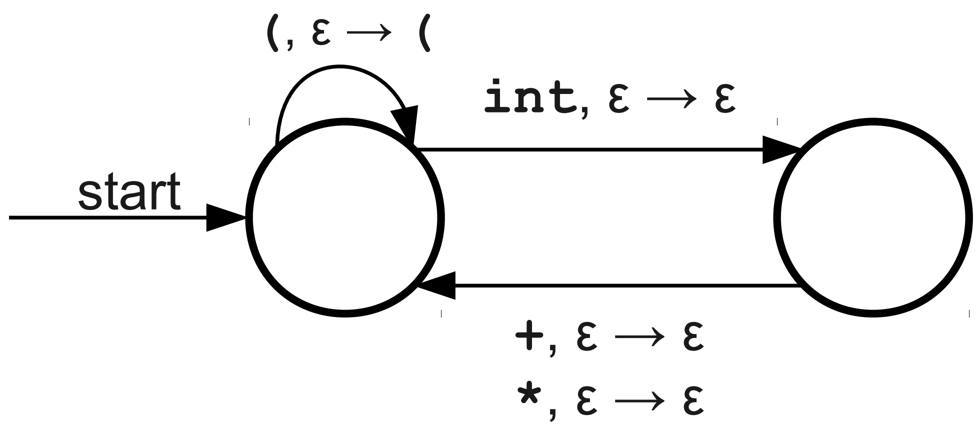
A PDA for Arithmetic



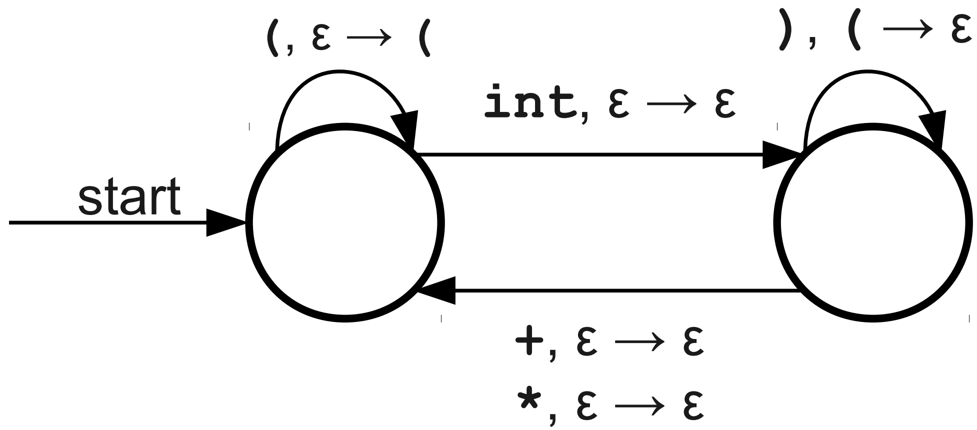
A PDA for Arithmetic



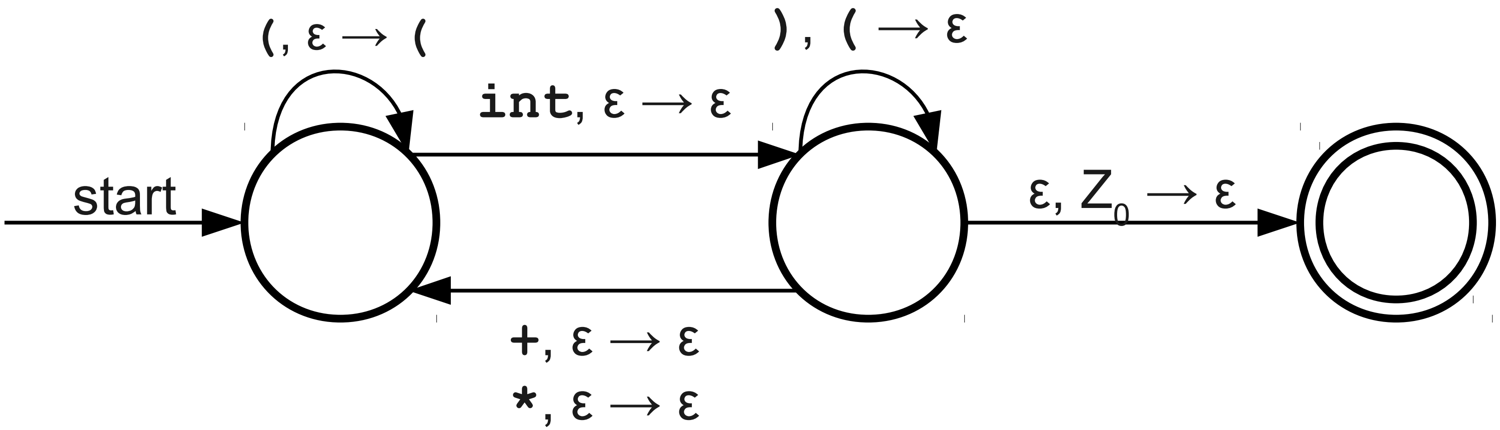
A PDA for Arithmetic



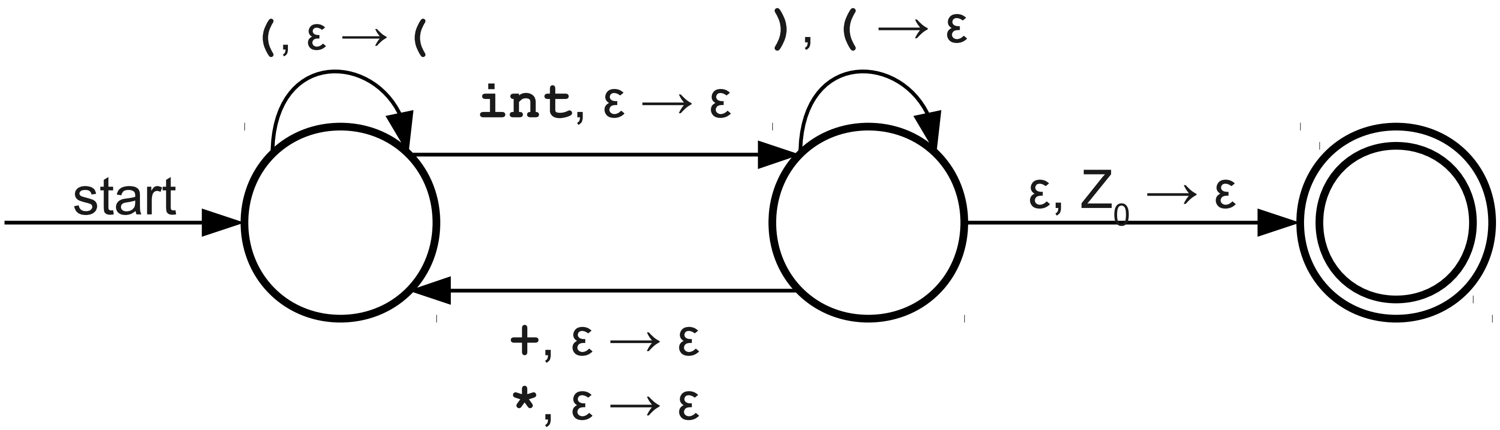
A PDA for Arithmetic



A PDA for Arithmetic

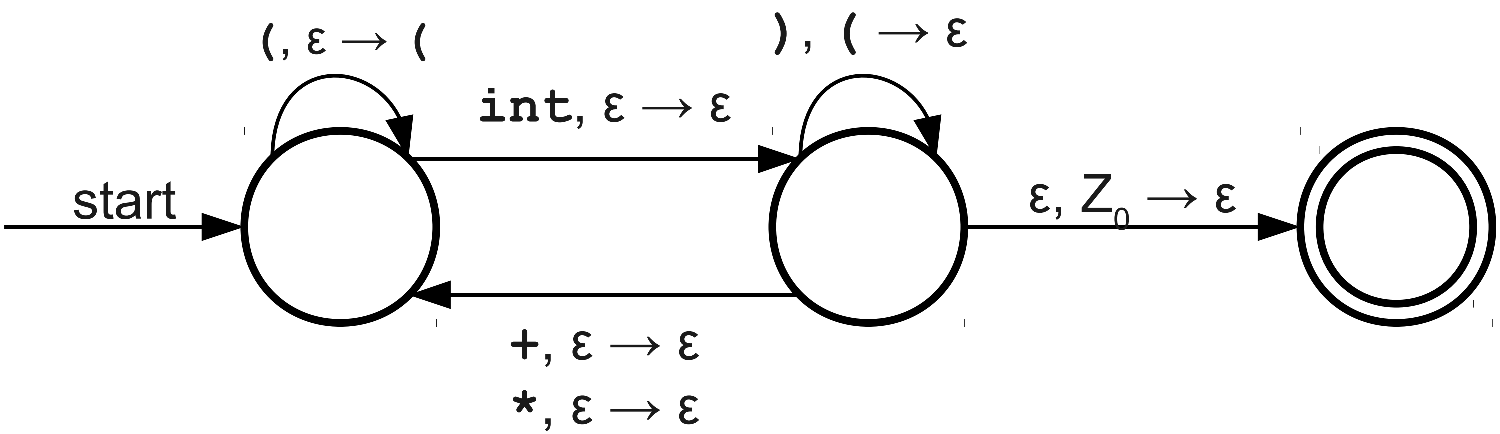


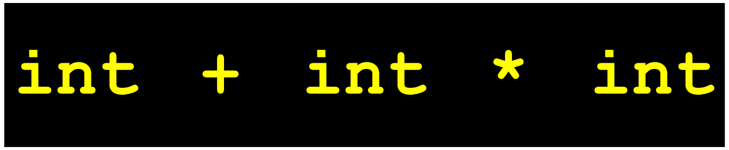
A PDA for Arithmetic

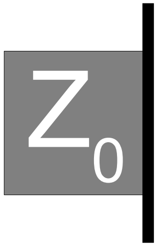



```
int + int * int
```

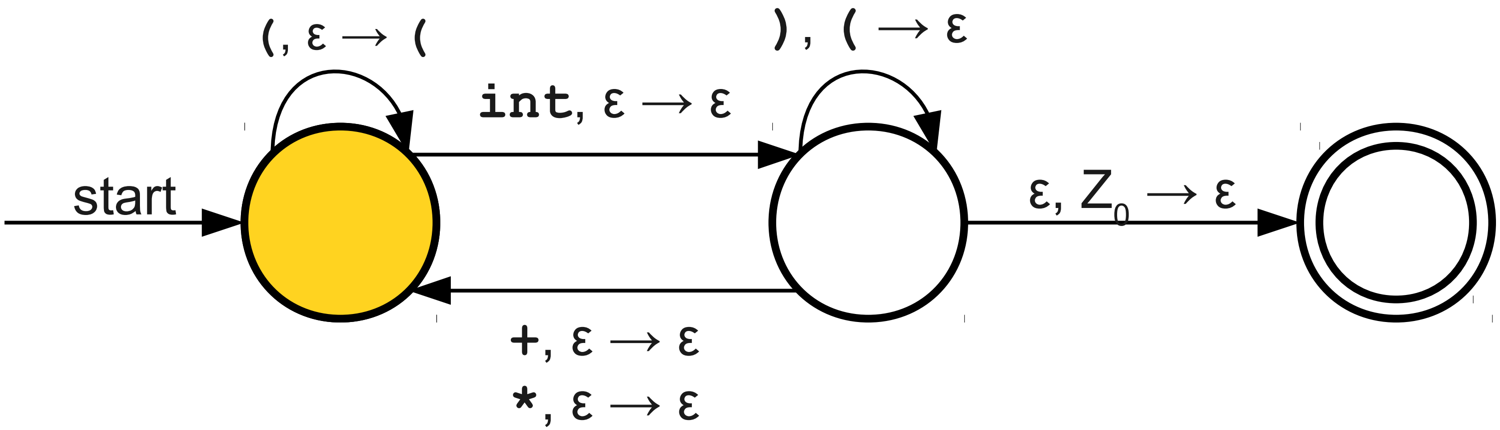
A PDA for Arithmetic







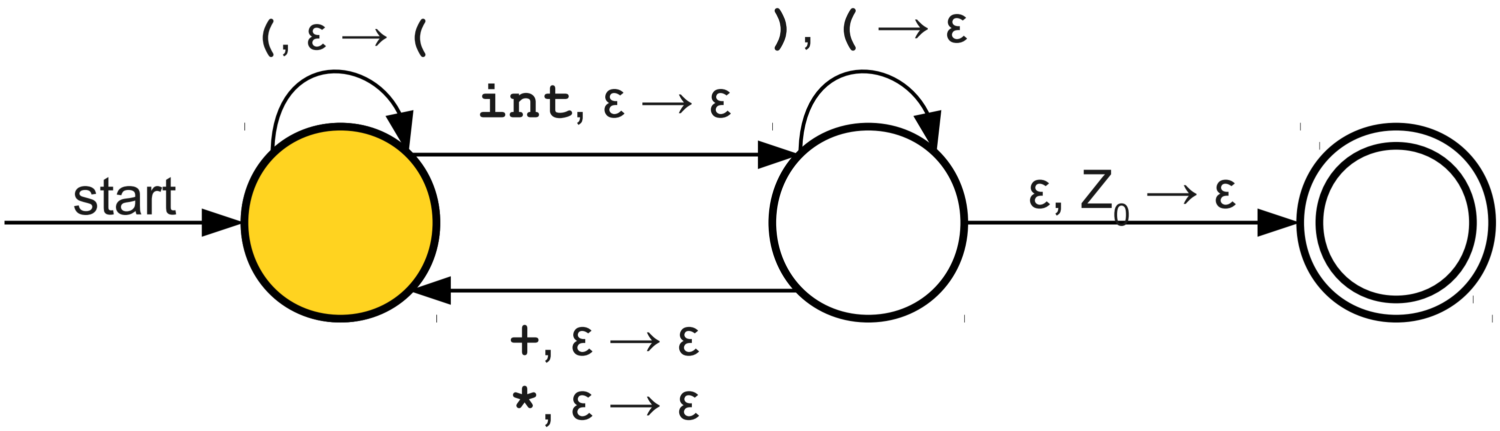
A PDA for Arithmetic

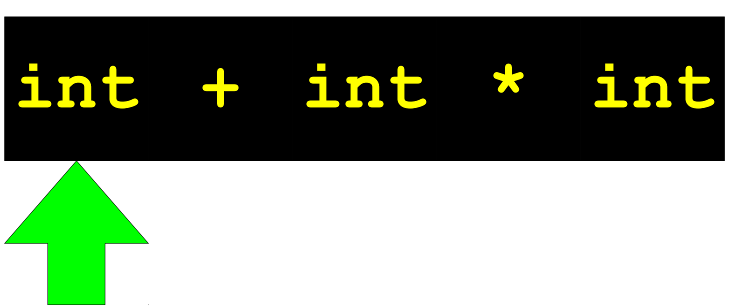


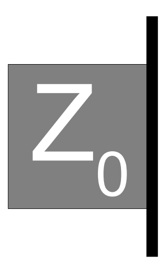
`int + int * int`

Z_0

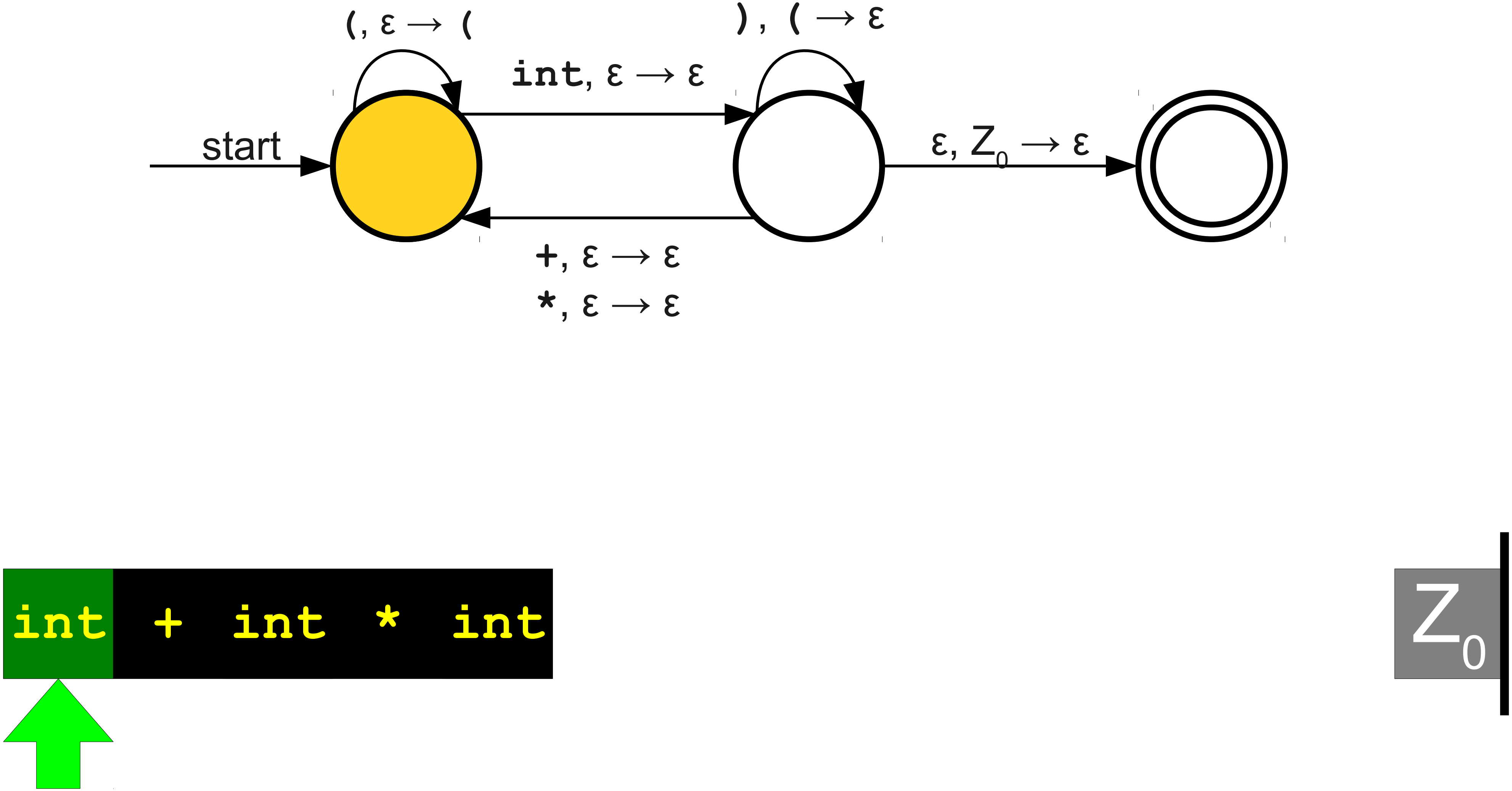
A PDA for Arithmetic



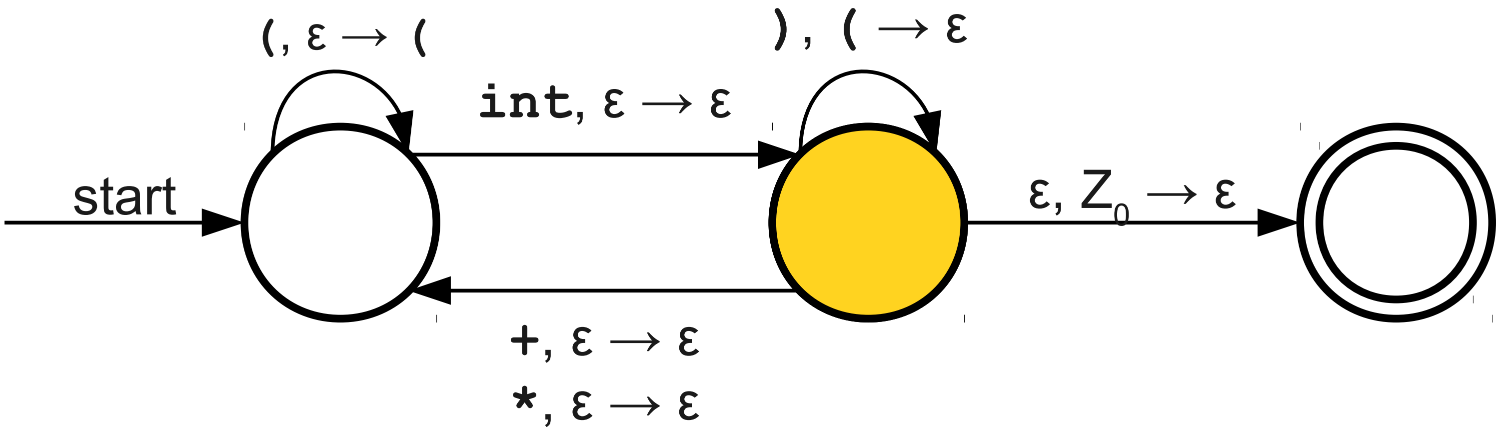


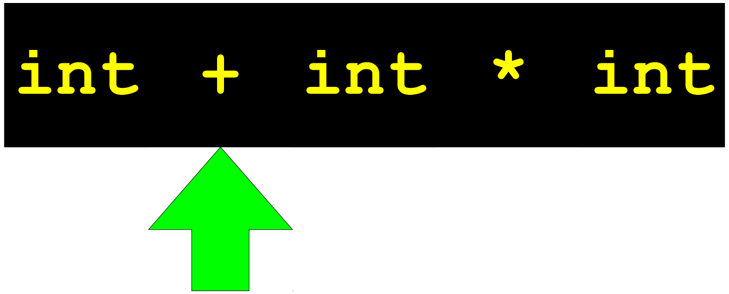


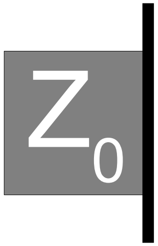
A PDA for Arithmetic



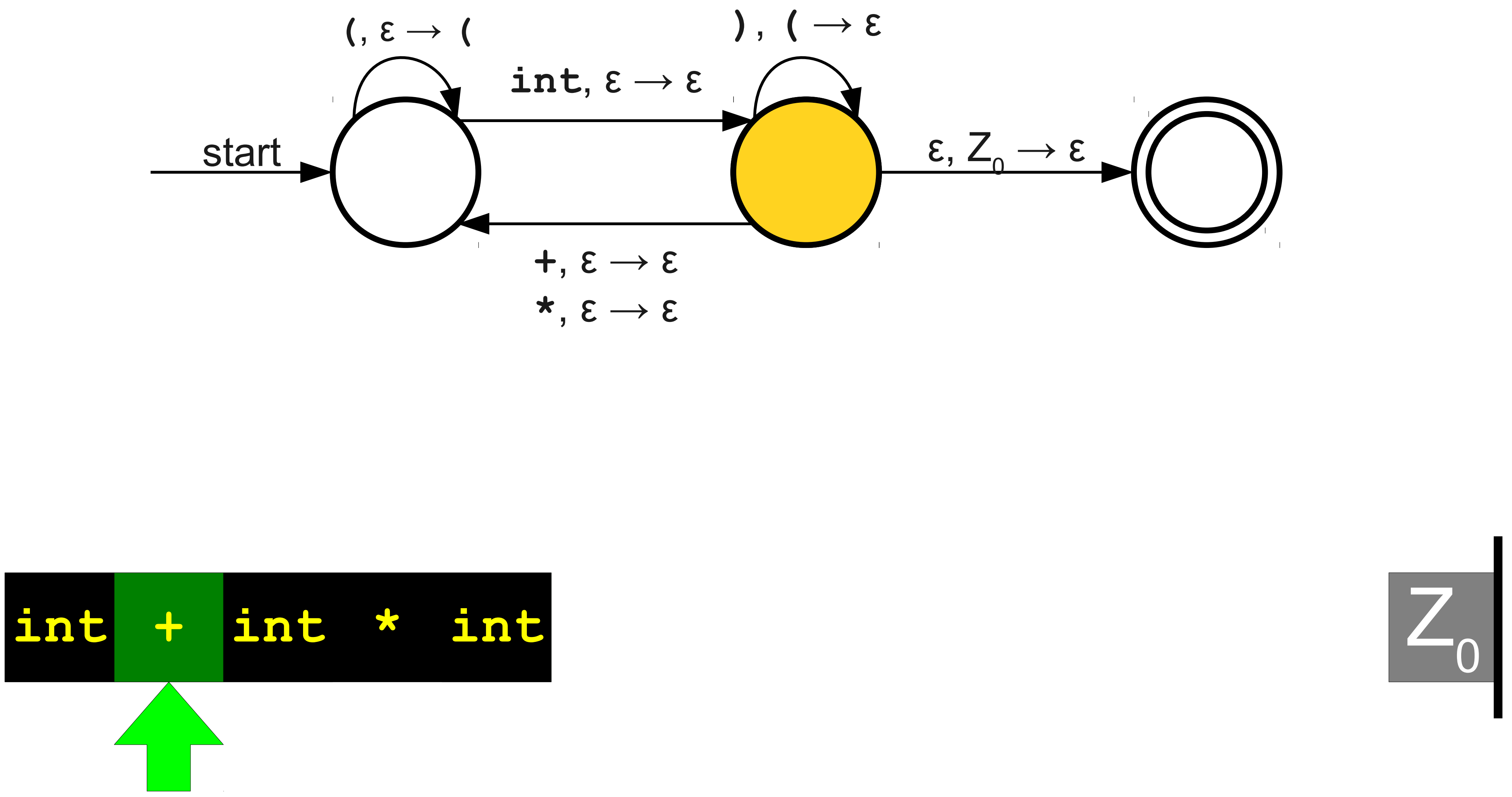
A PDA for Arithmetic



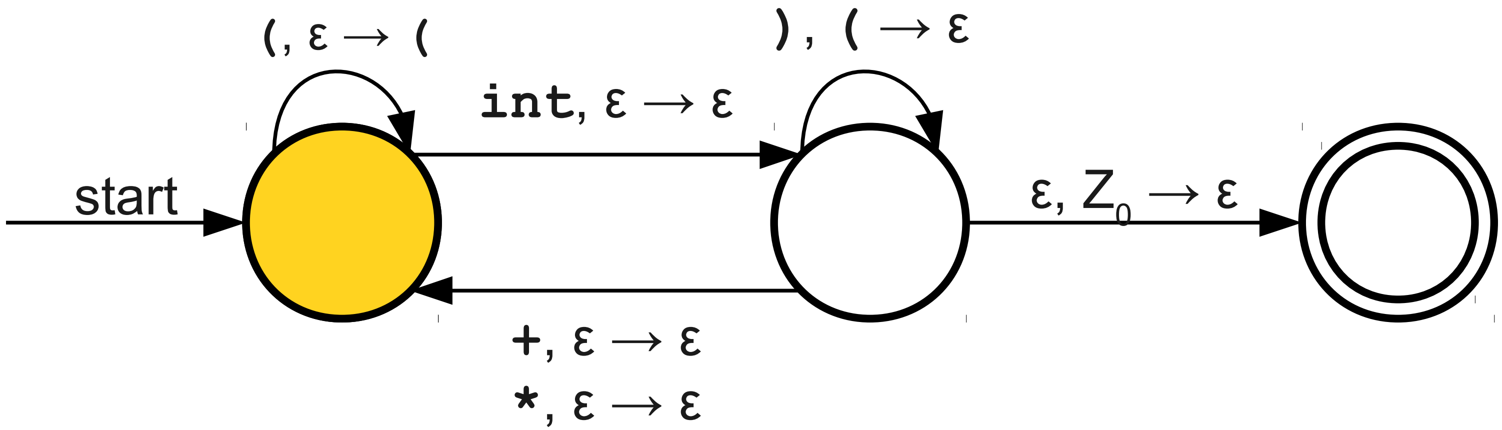




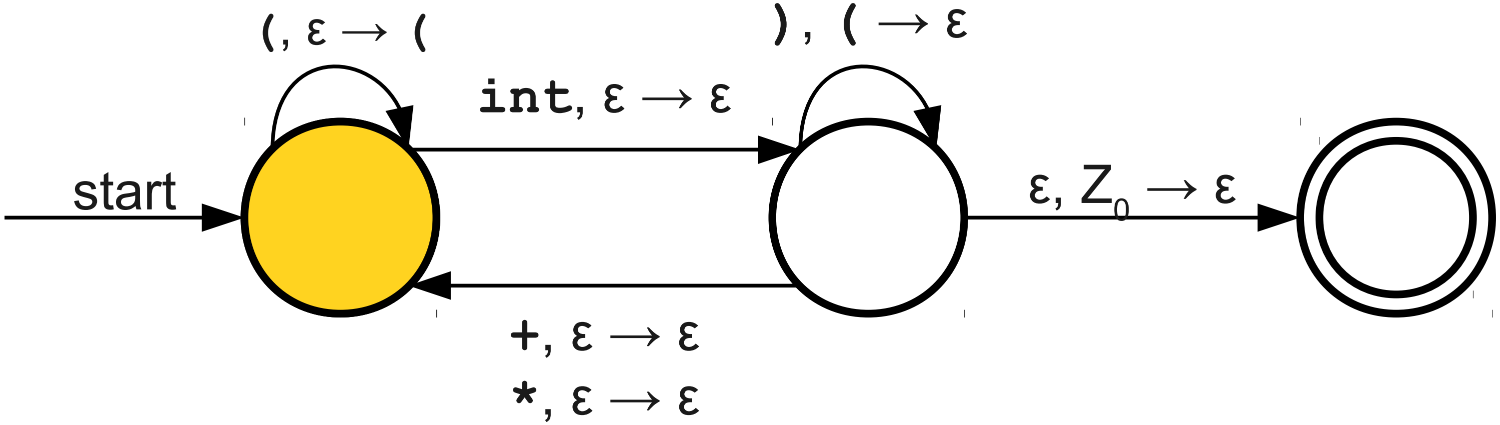
A PDA for Arithmetic

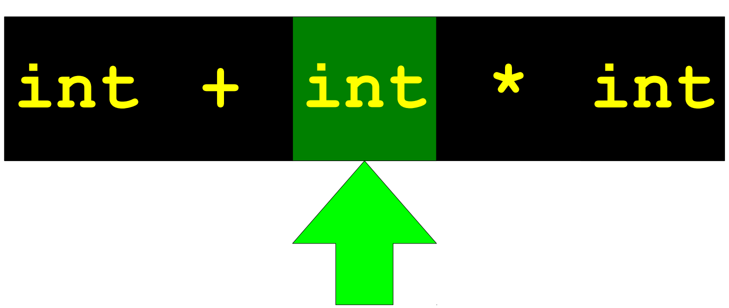


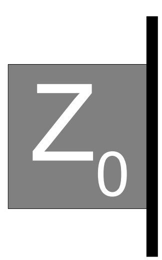
A PDA for Arithmetic



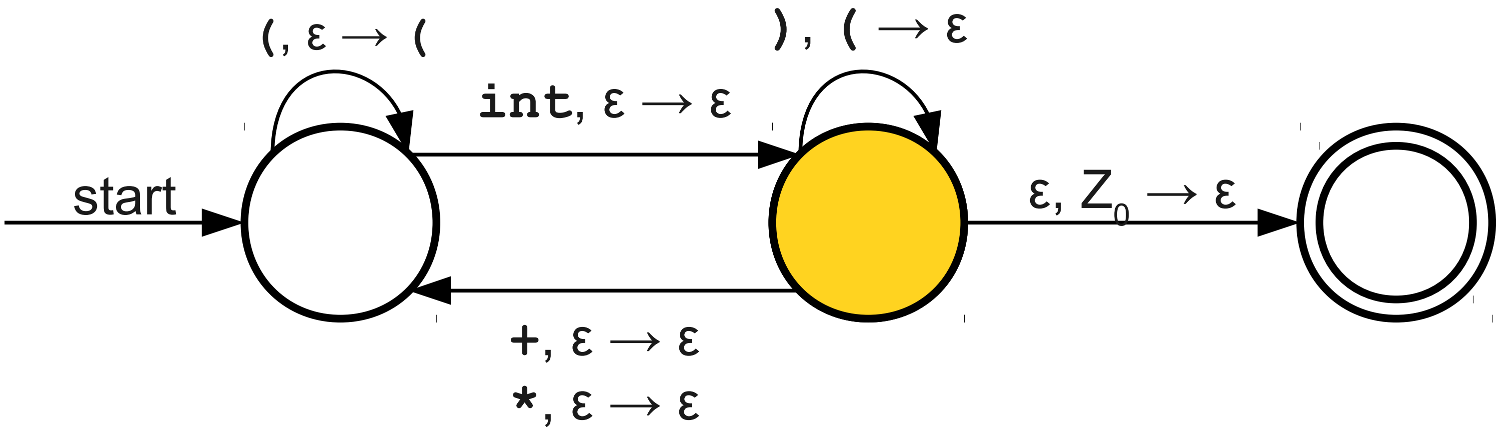
A PDA for Arithmetic



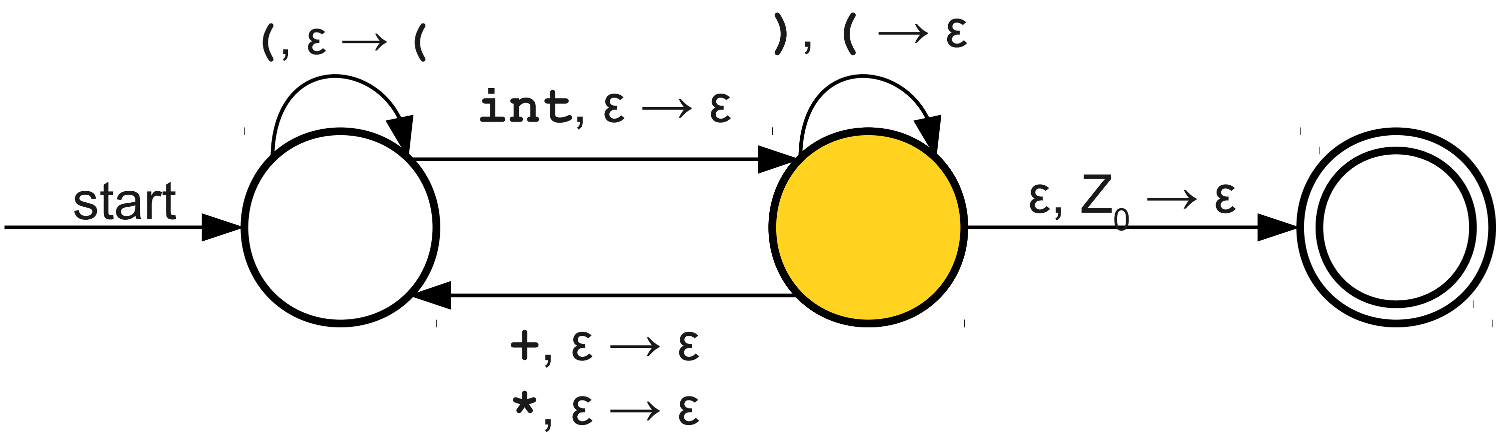


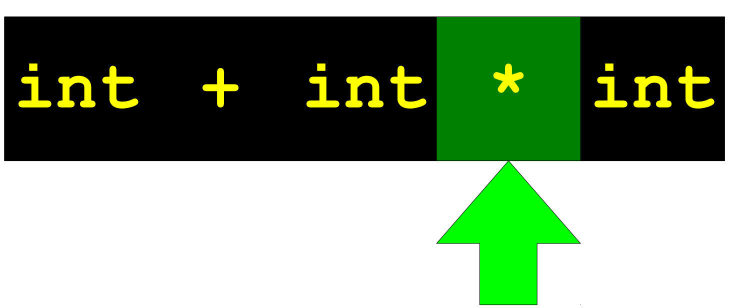


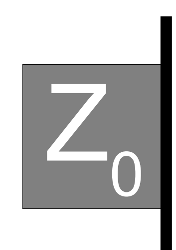
A PDA for Arithmetic



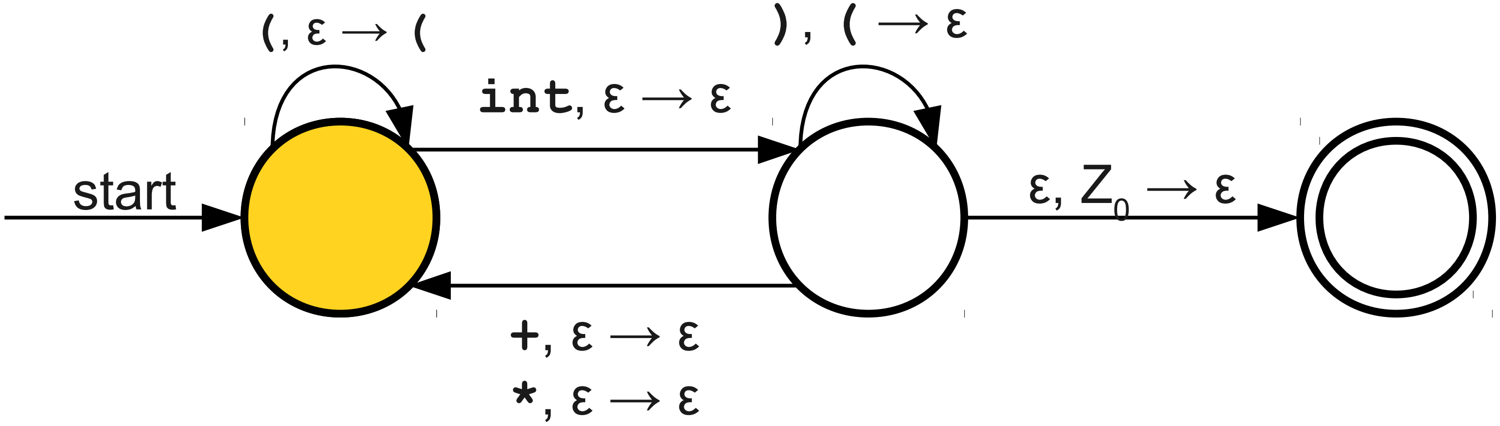
A PDA for Arithmetic



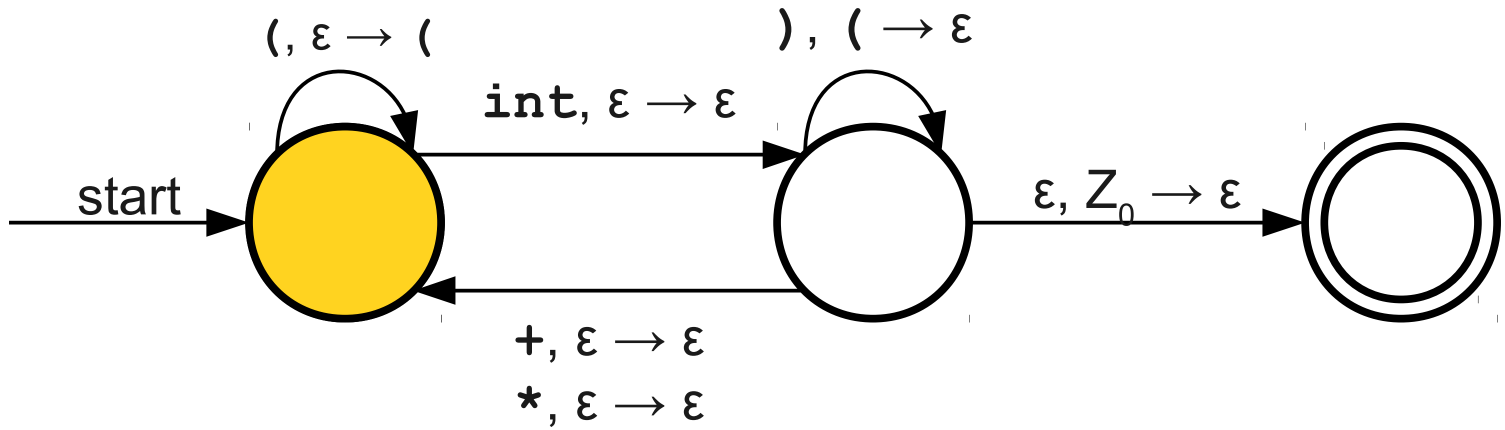


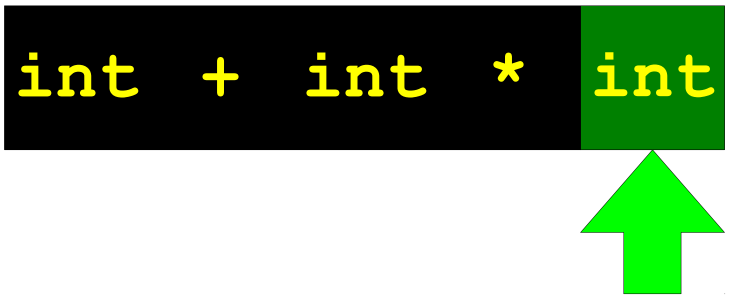


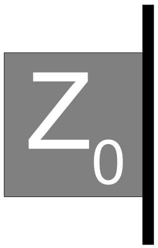
A PDA for Arithmetic



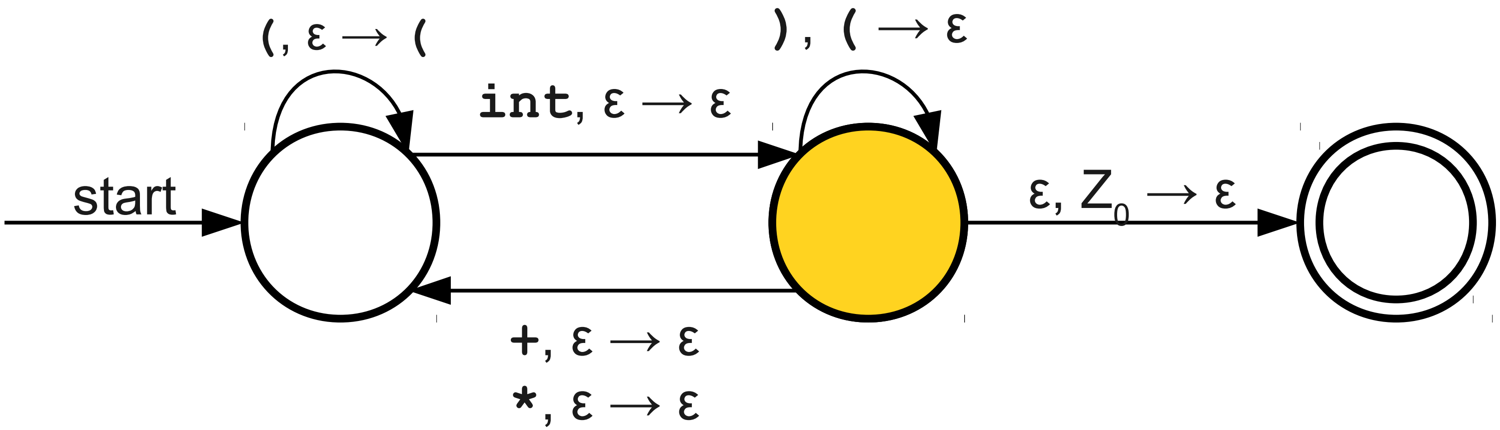
A PDA for Arithmetic

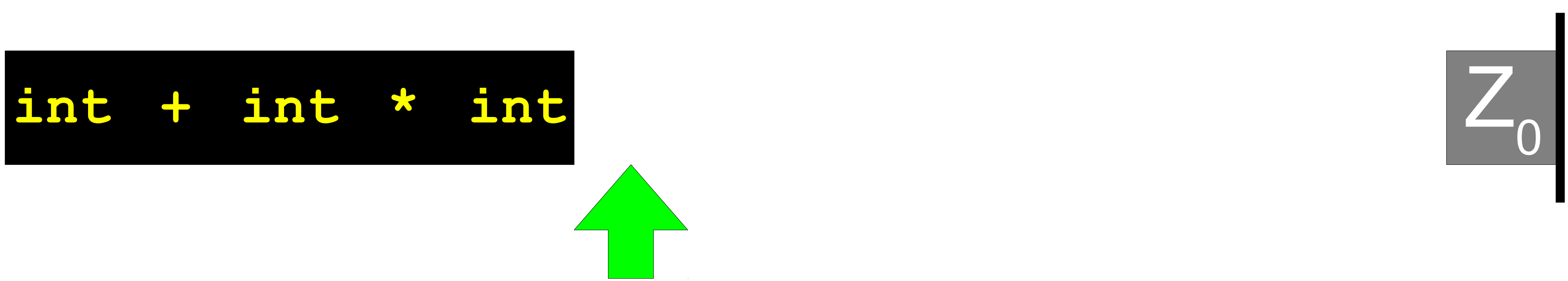




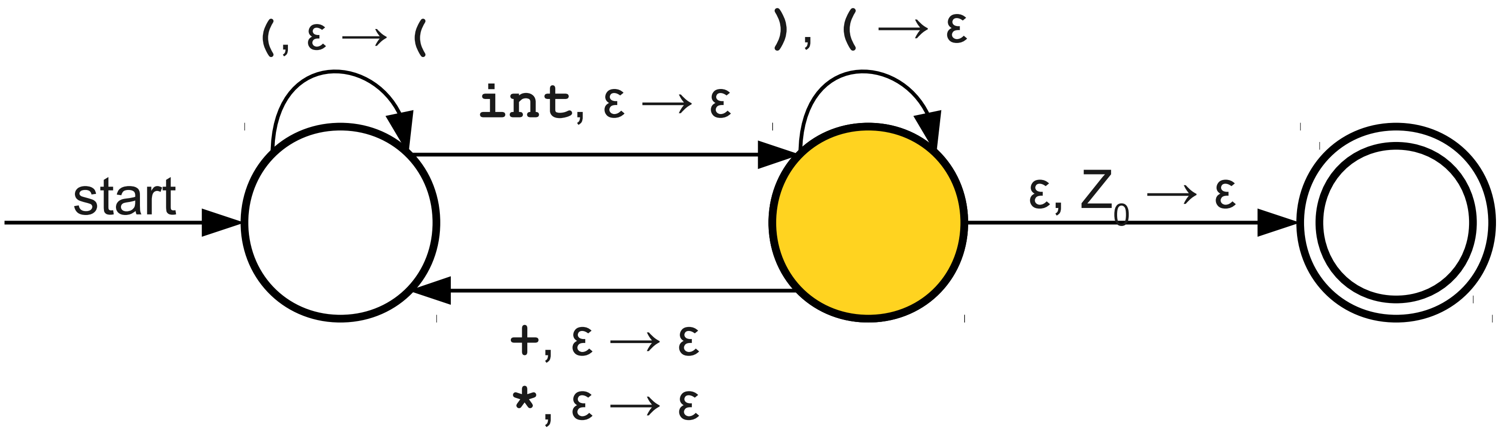


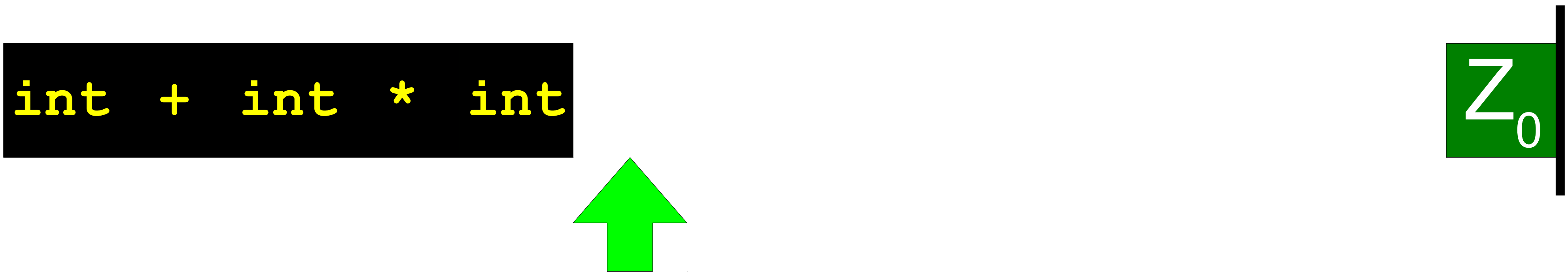
A PDA for Arithmetic



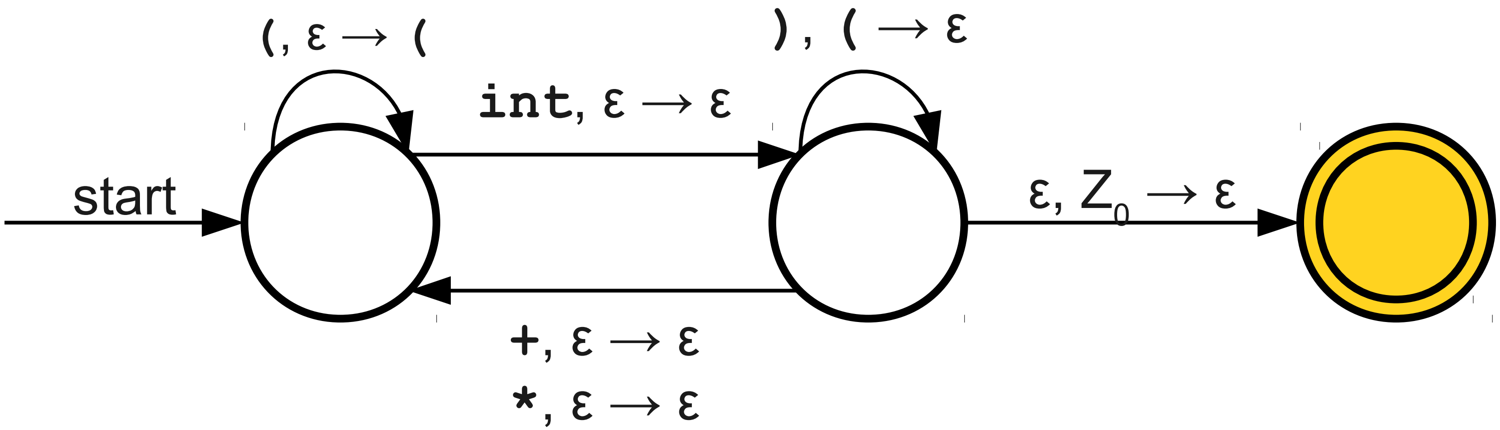


A PDA for Arithmetic



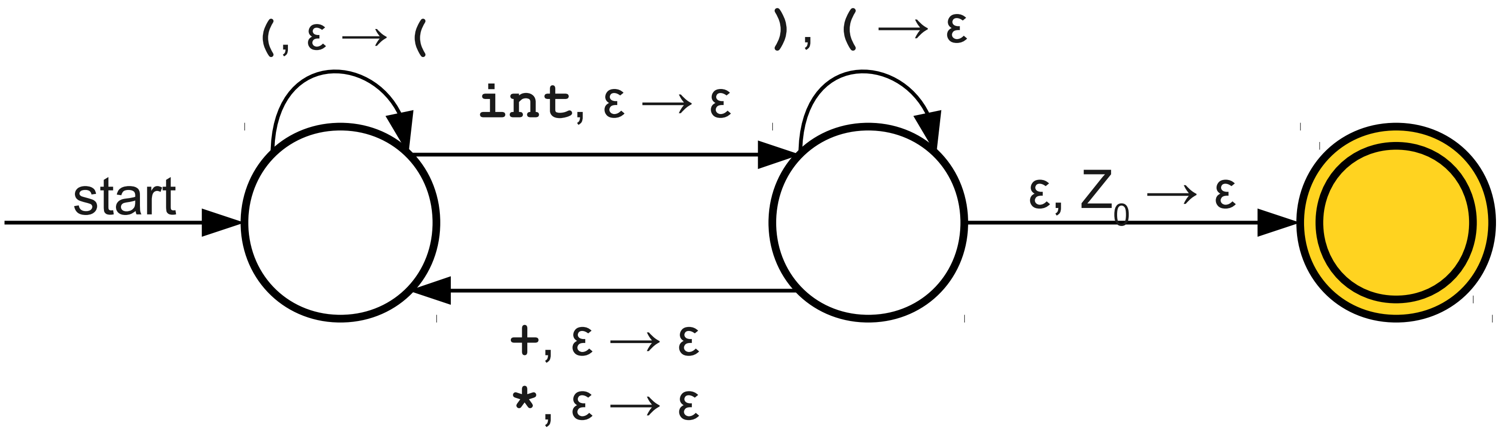


A PDA for Arithmetic



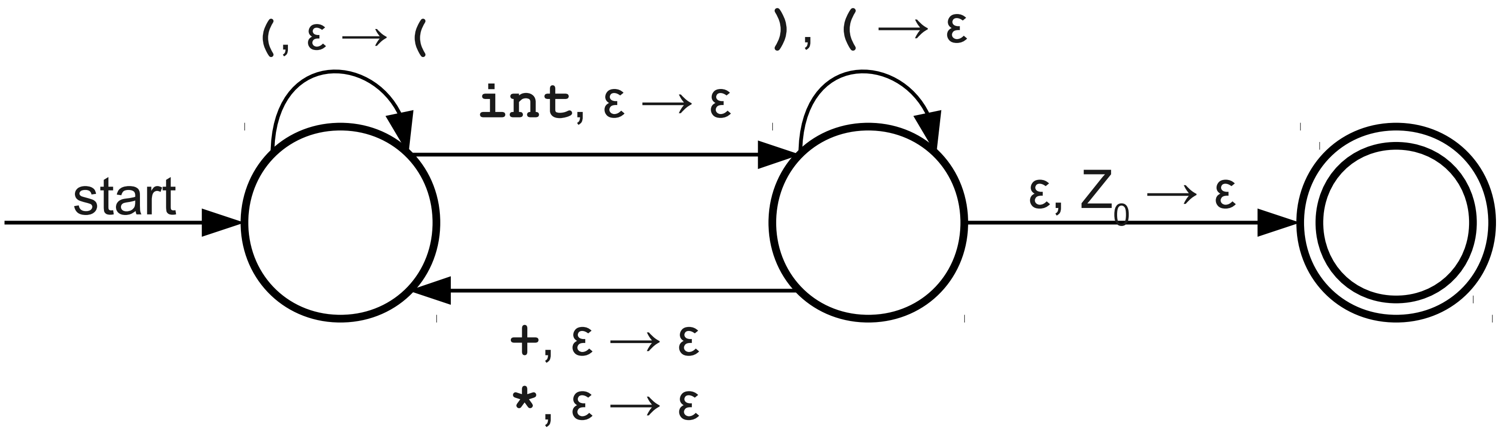
int + int * int

A PDA for Arithmetic

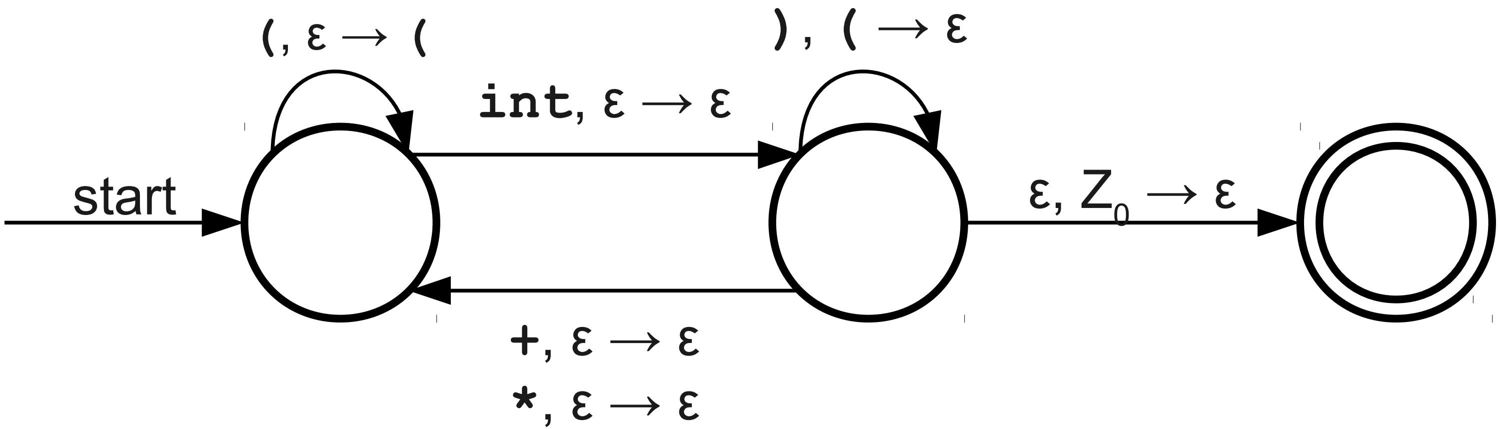


`int + int * int`

A PDA for Arithmetic

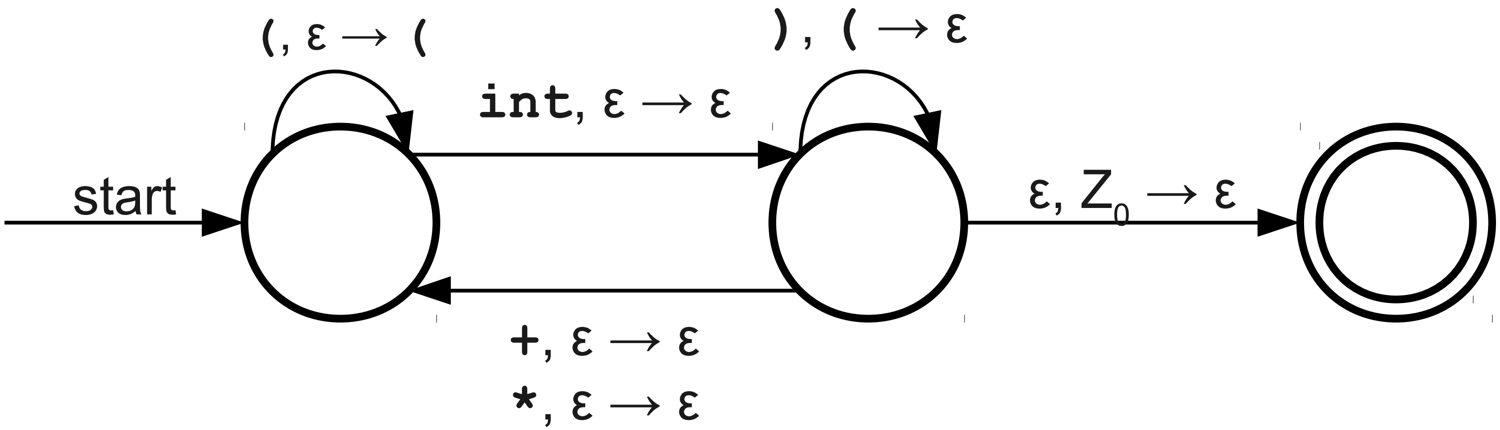


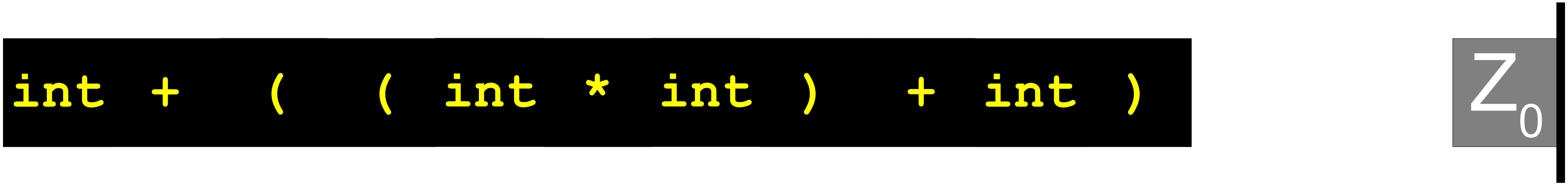
A PDA for Arithmetic



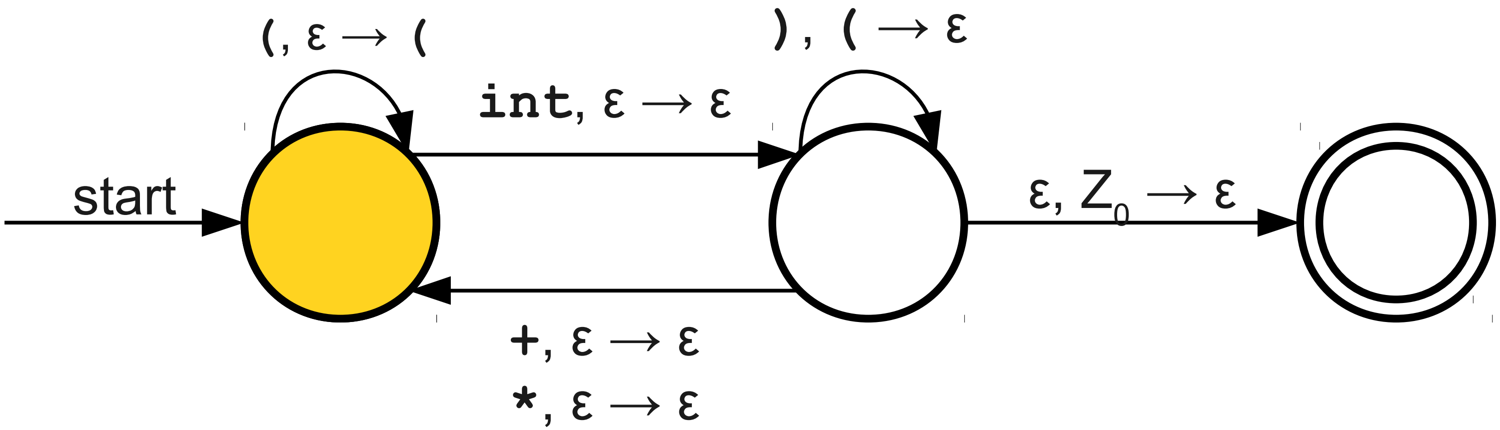
int + ((int * int) + int)

A PDA for Arithmetic

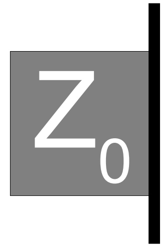




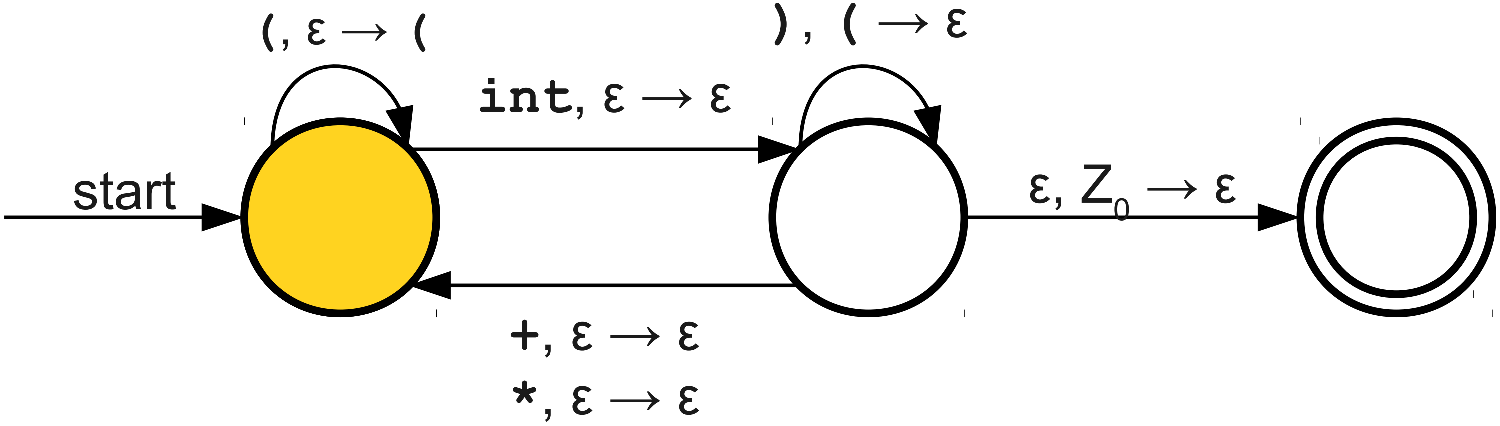
A PDA for Arithmetic

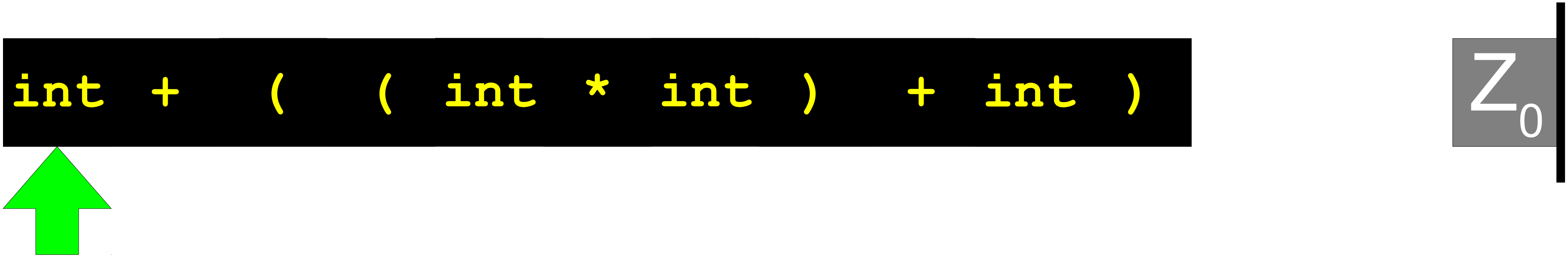


int + ((int * int) + int)

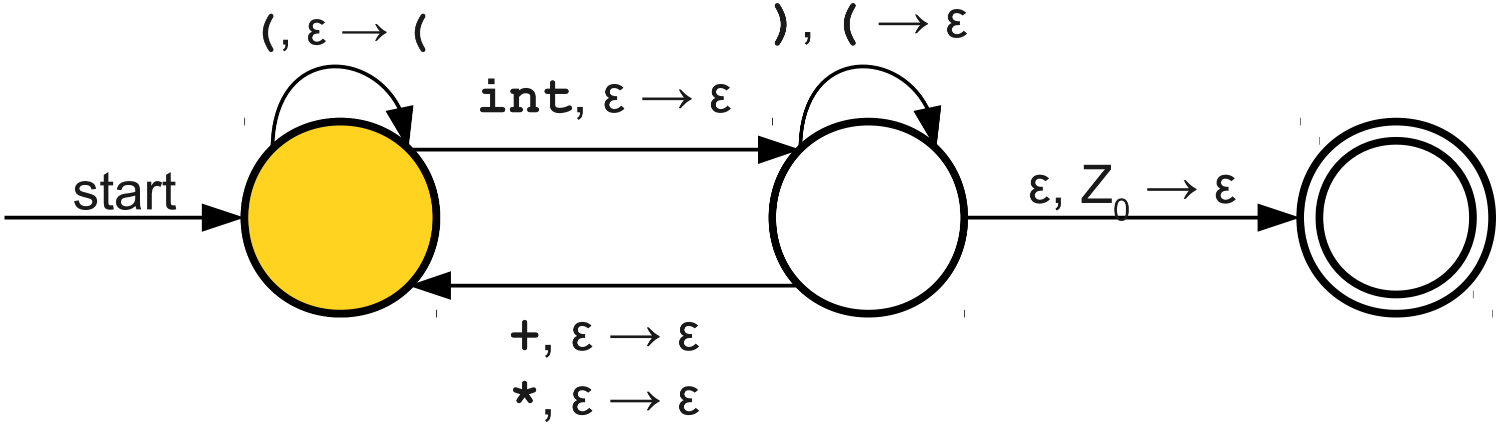


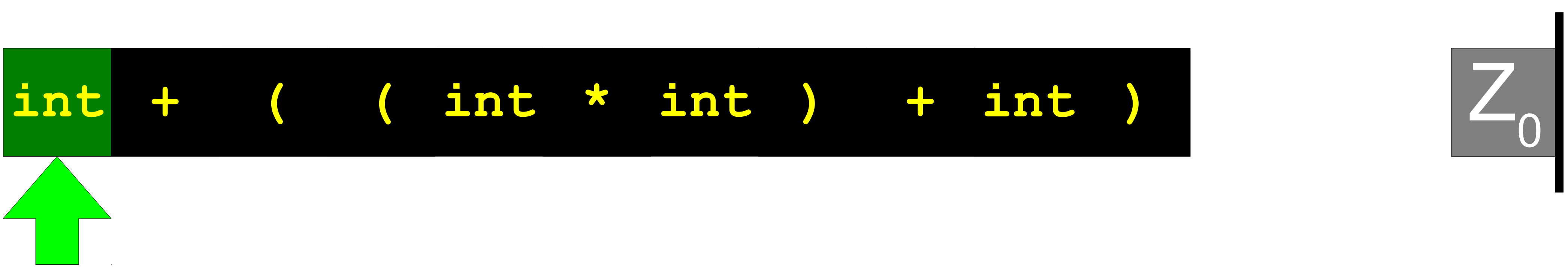
A PDA for Arithmetic



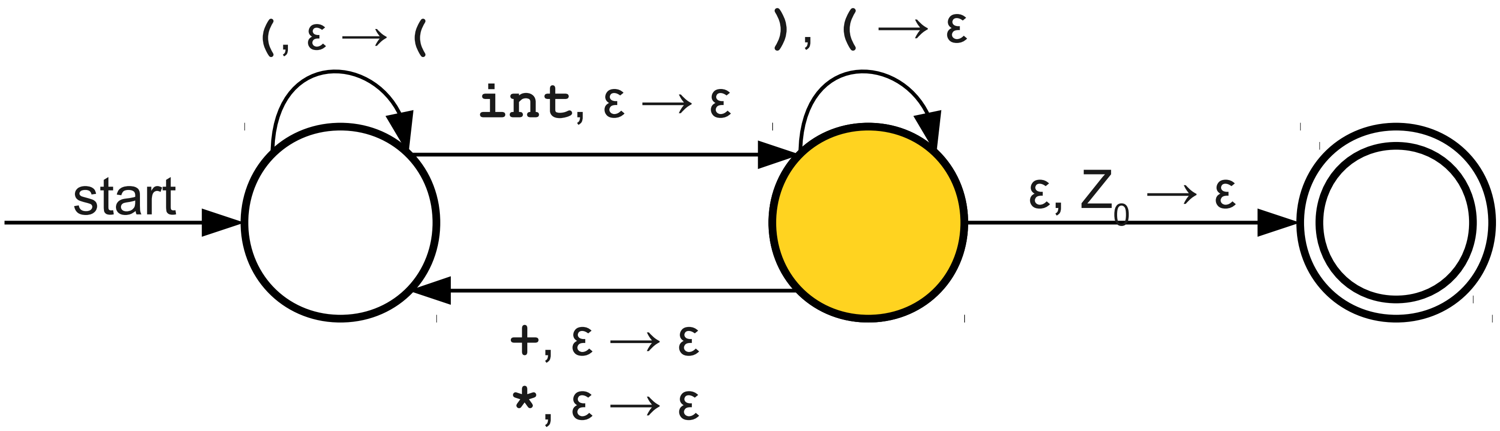


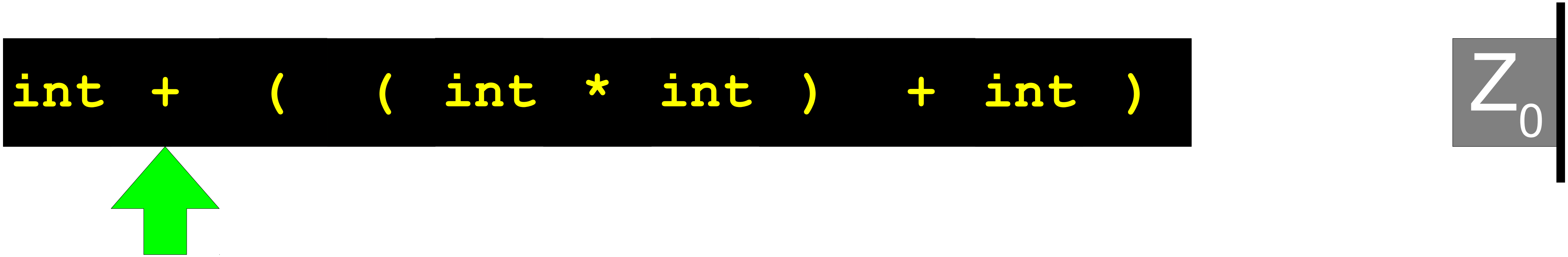
A PDA for Arithmetic



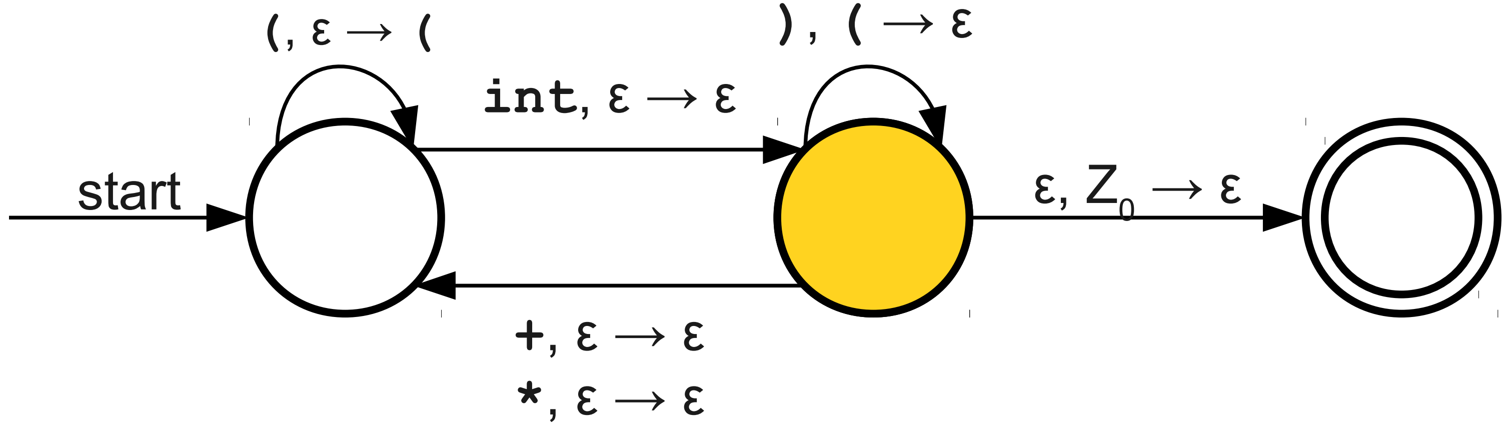


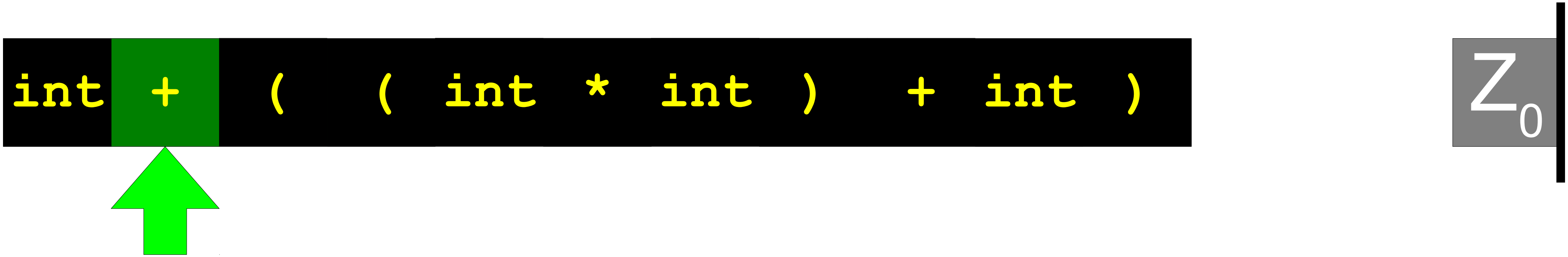
A PDA for Arithmetic



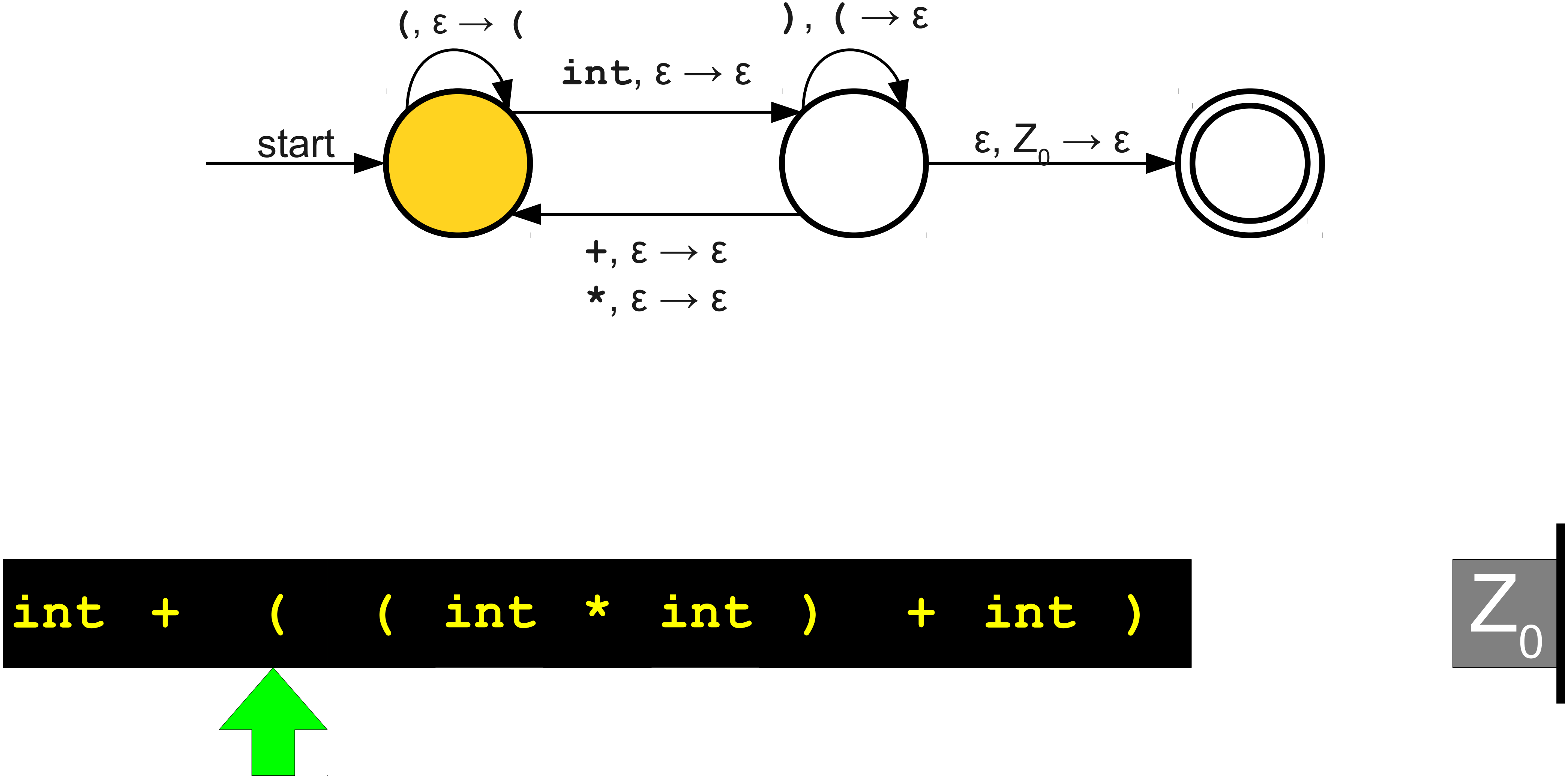


A PDA for Arithmetic

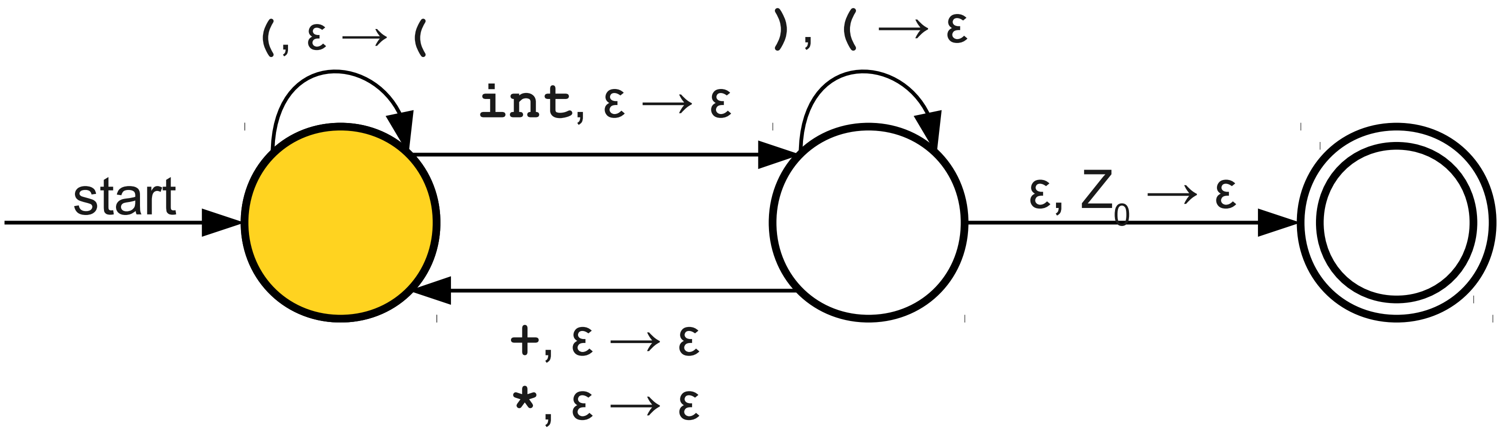


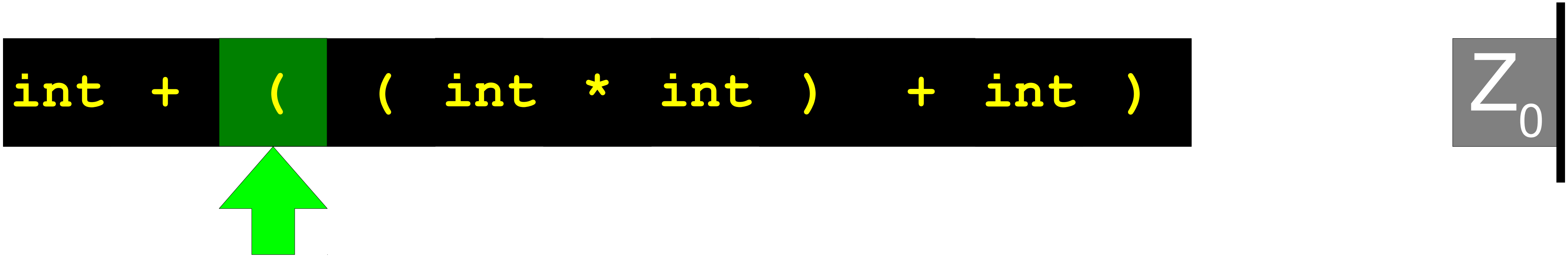


A PDA for Arithmetic

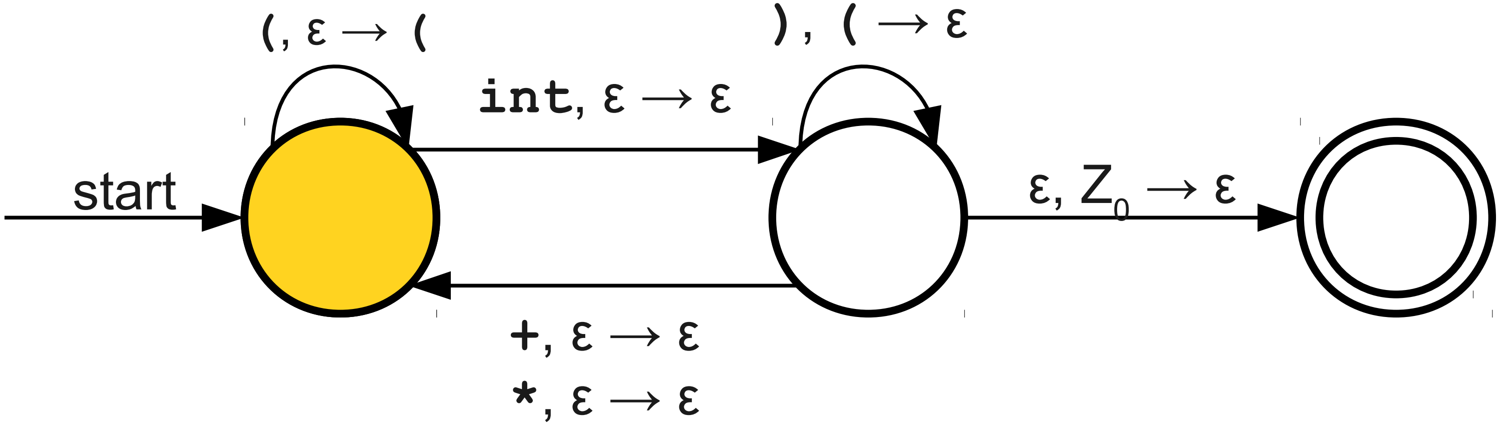


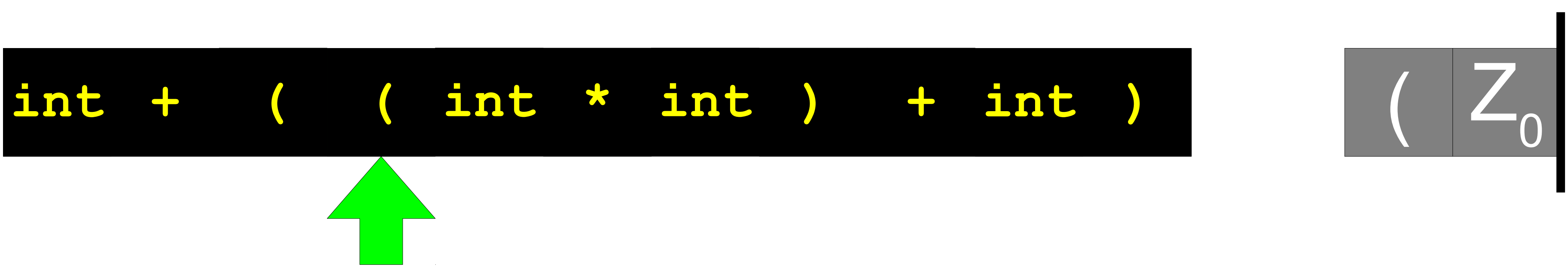
A PDA for Arithmetic



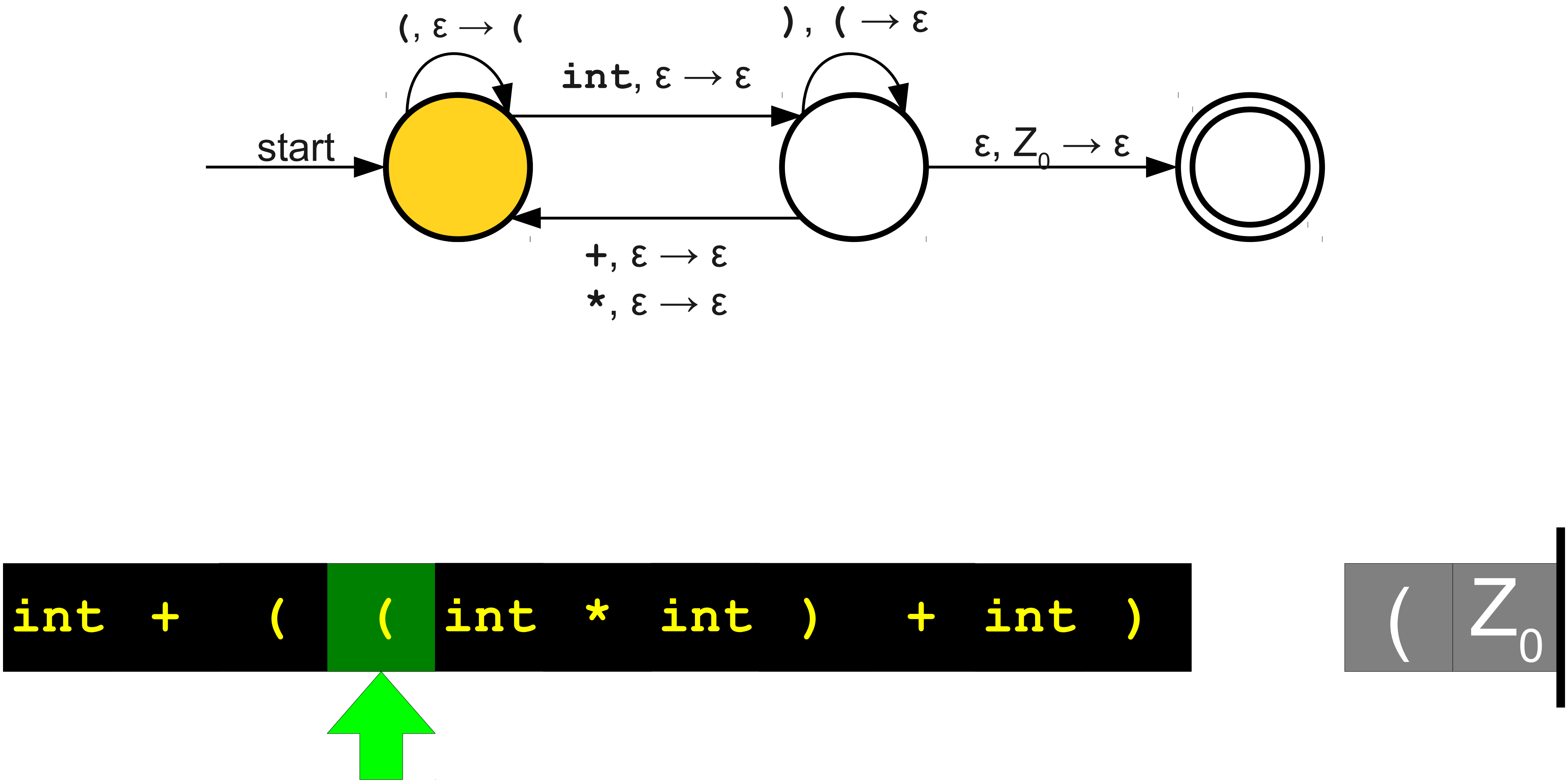


A PDA for Arithmetic

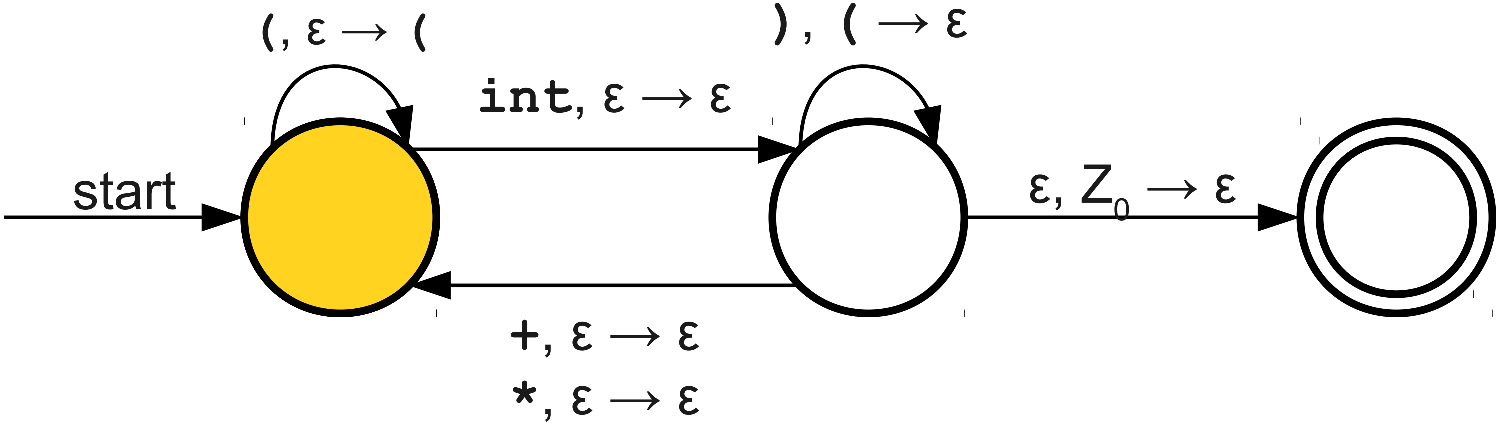


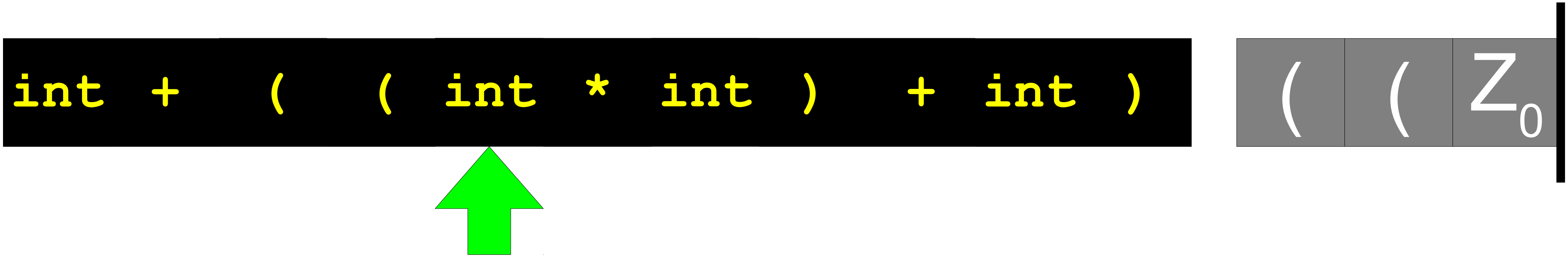


A PDA for Arithmetic

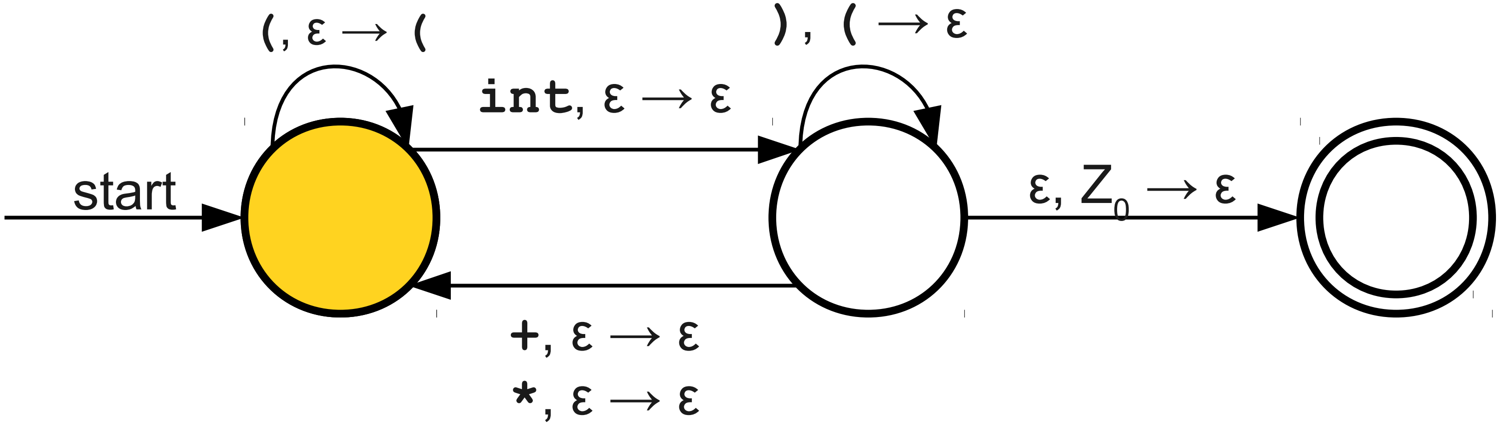


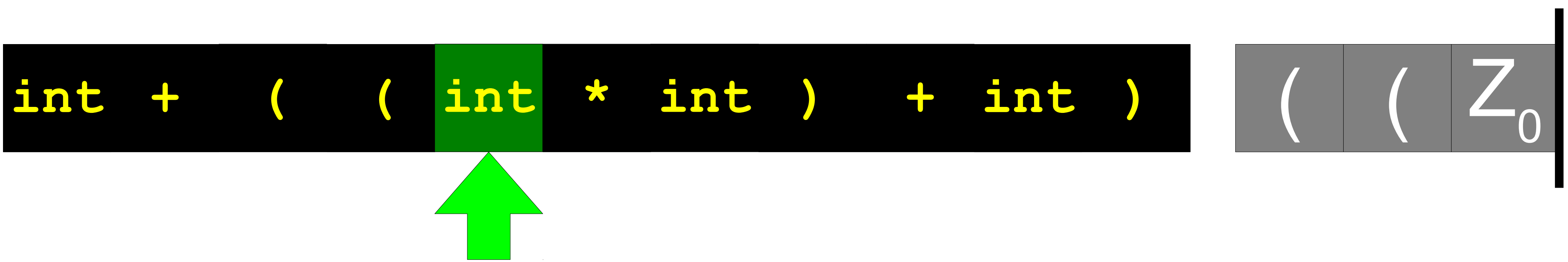
A PDA for Arithmetic



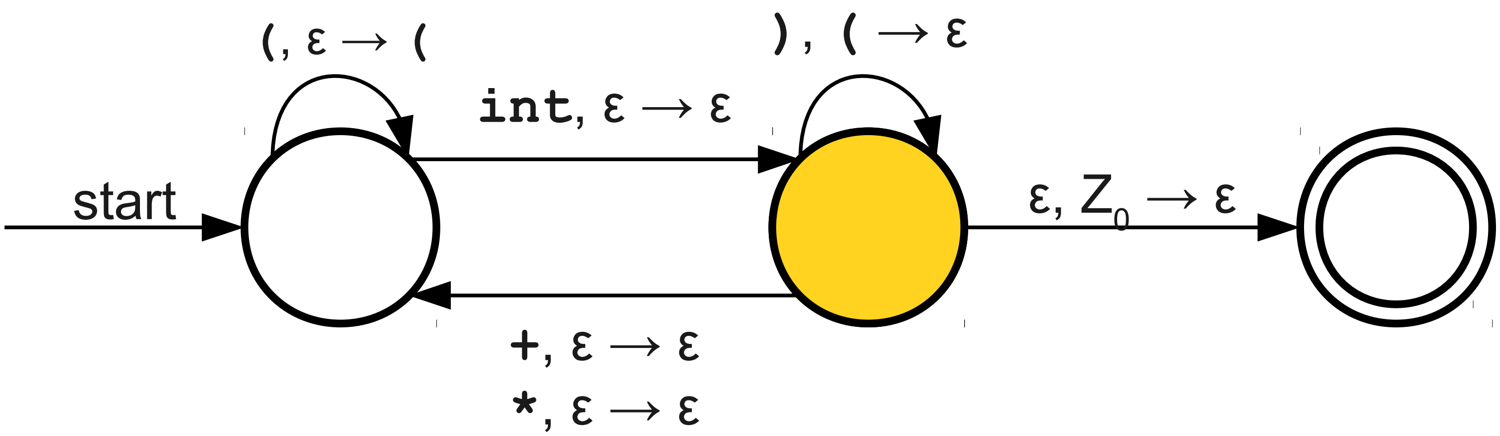


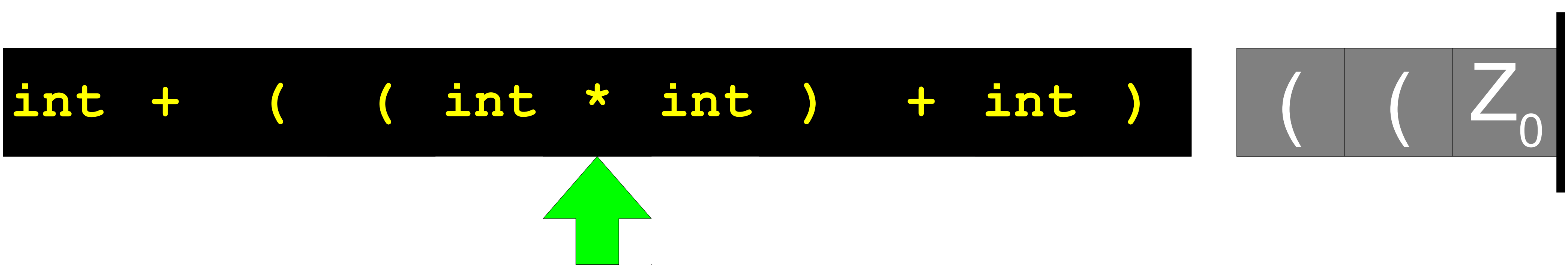
A PDA for Arithmetic



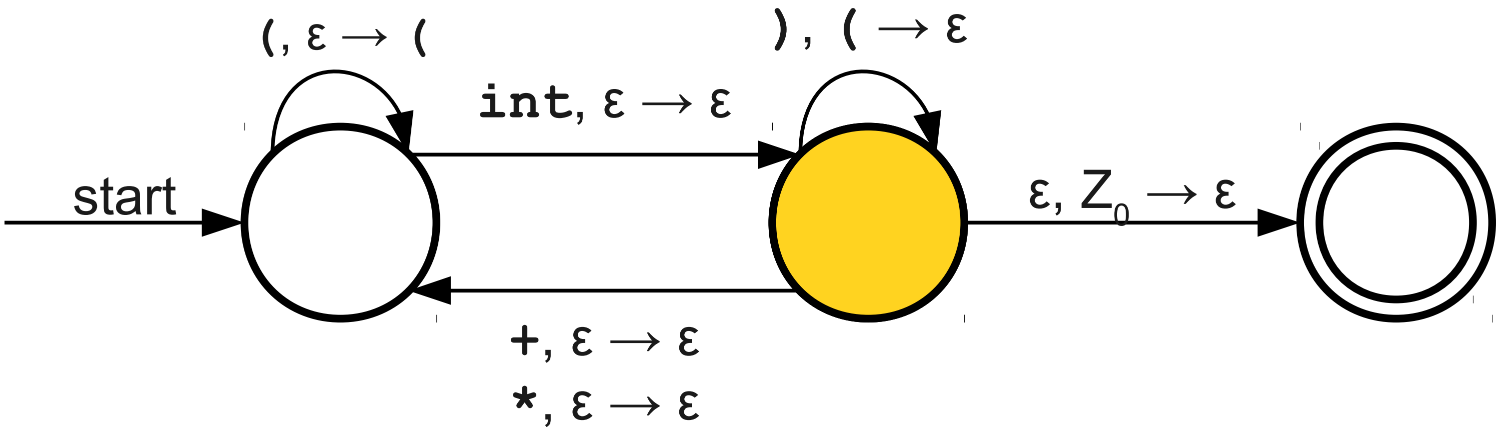


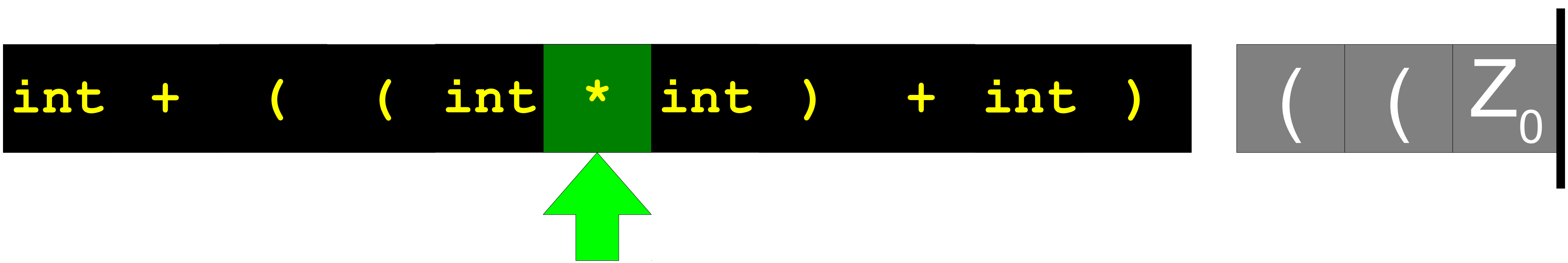
A PDA for Arithmetic



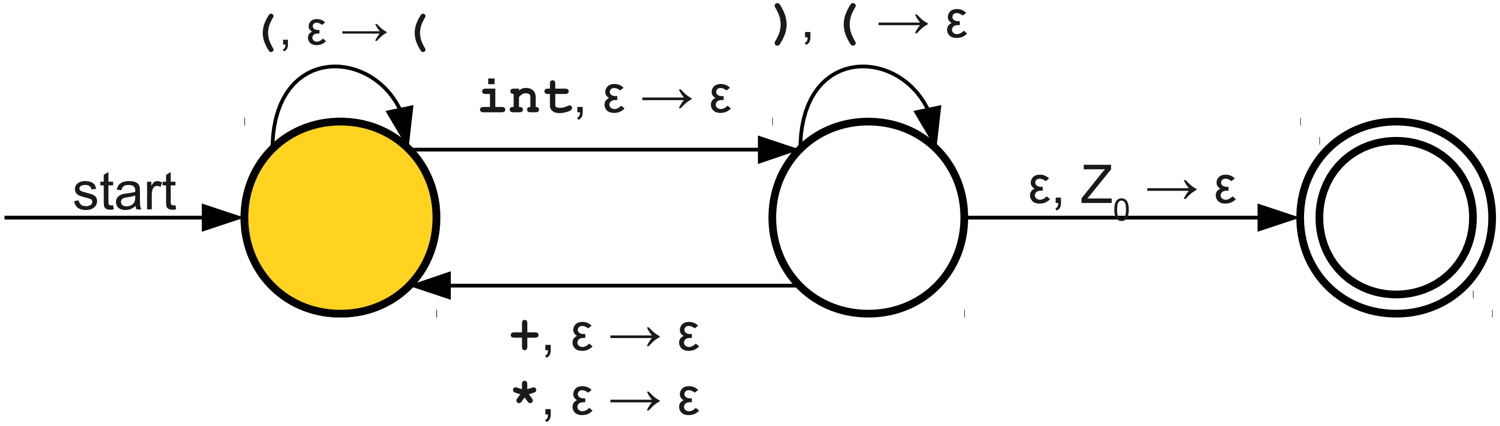


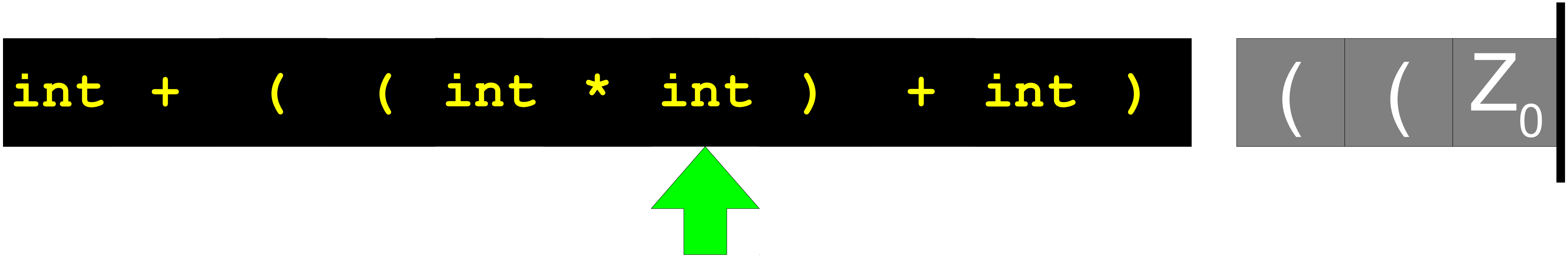
A PDA for Arithmetic



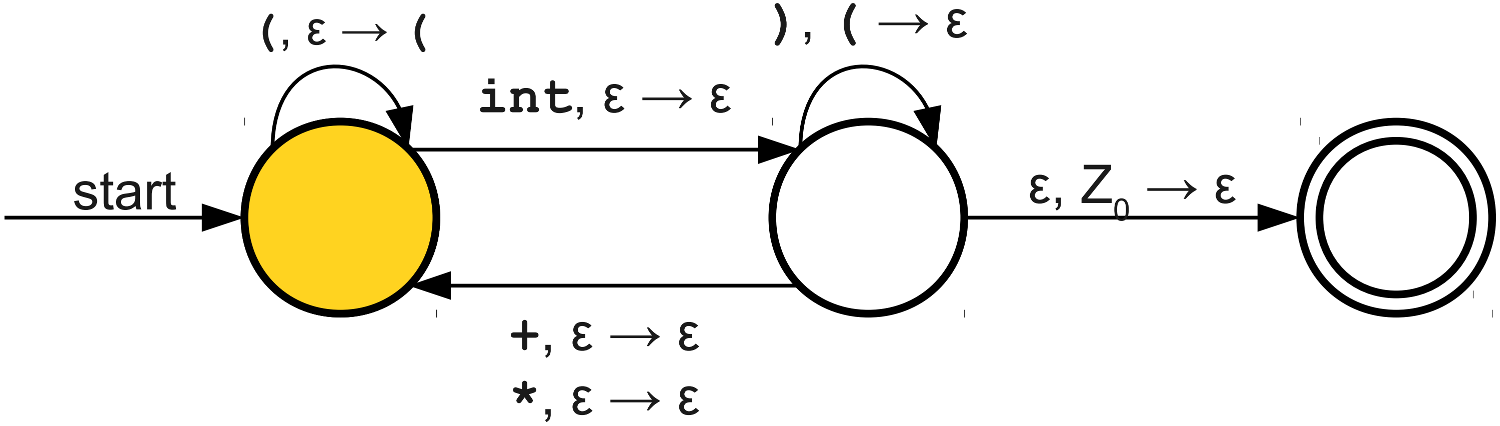


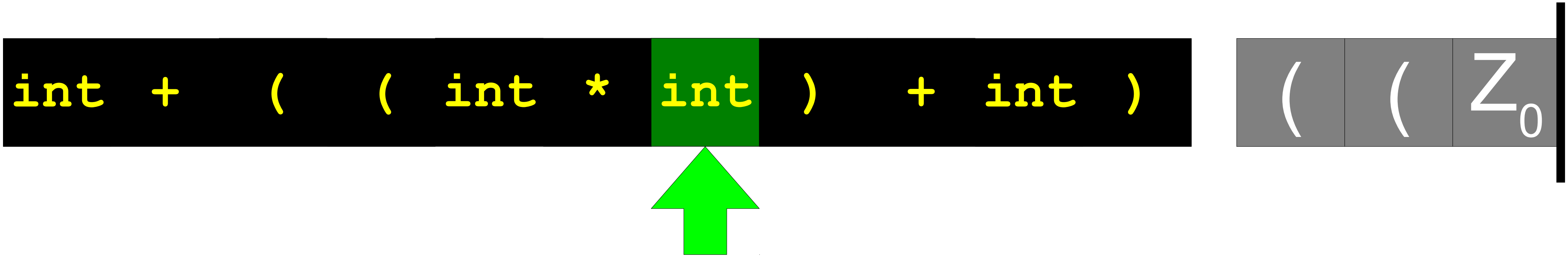
A PDA for Arithmetic



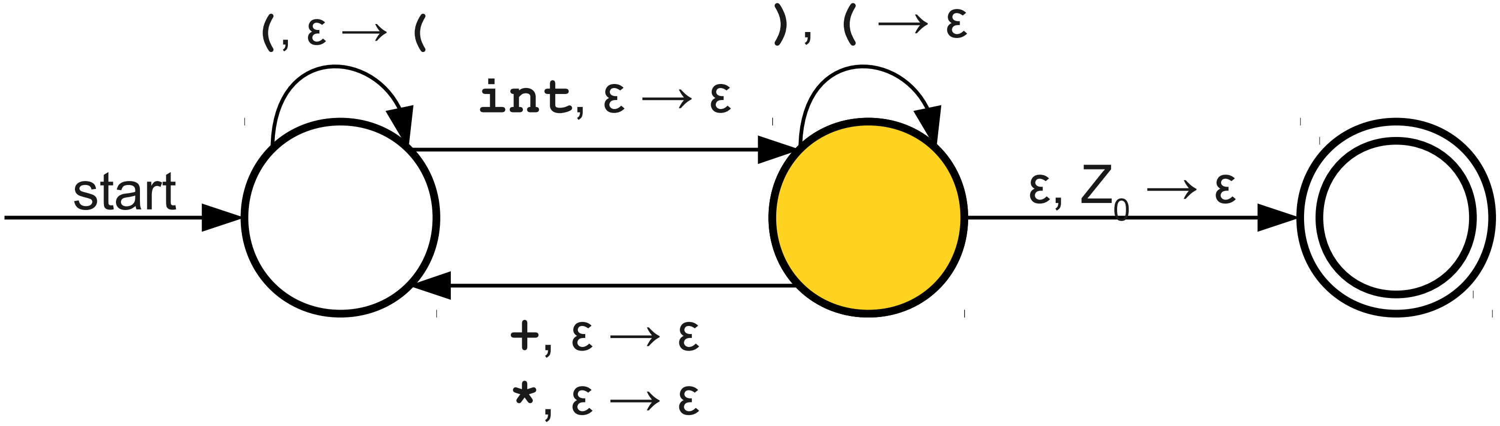


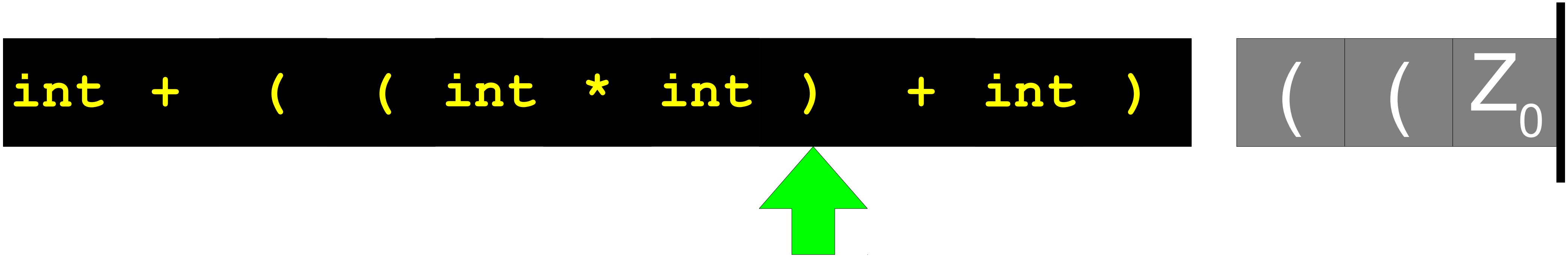
A PDA for Arithmetic



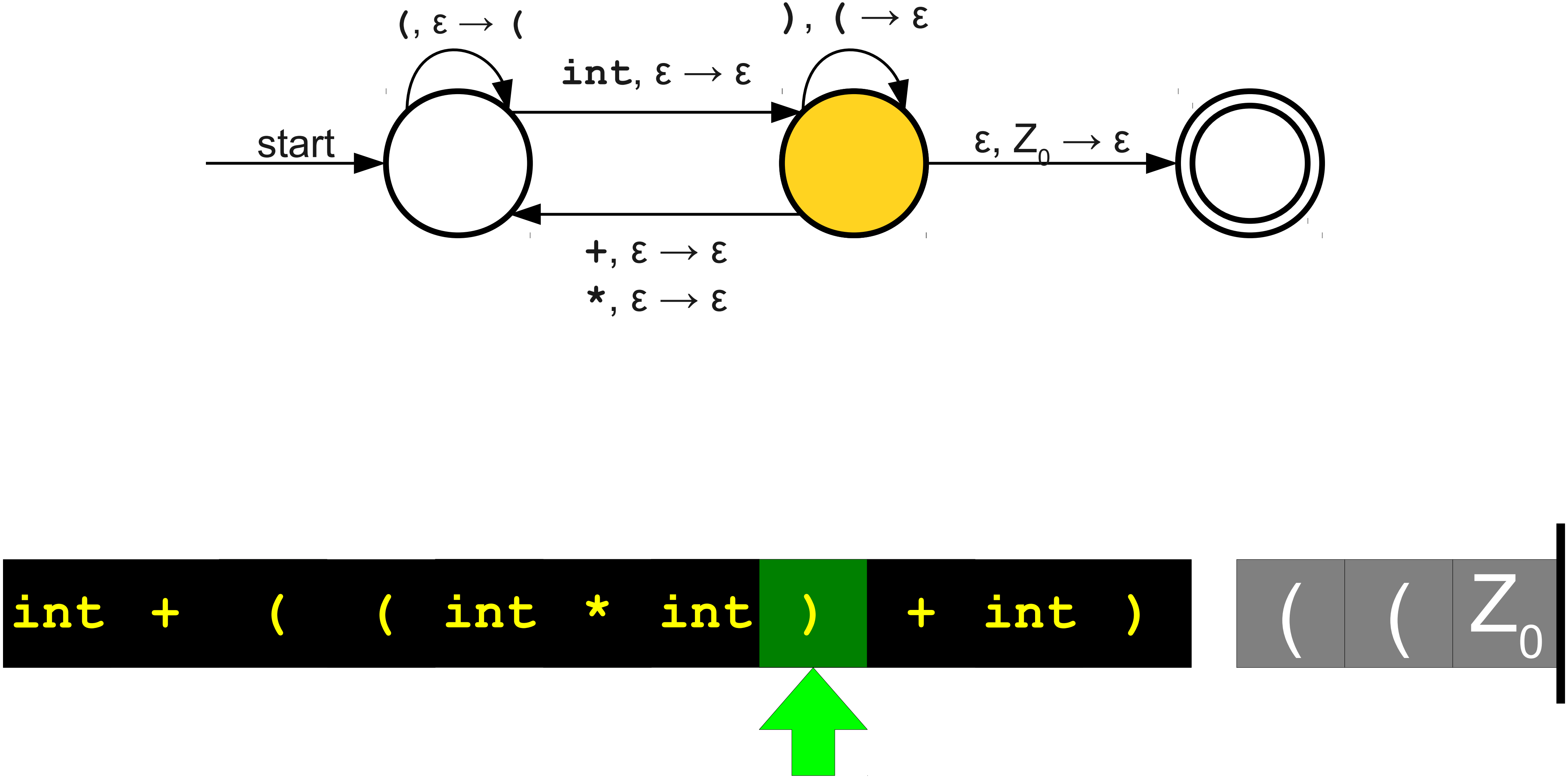


A PDA for Arithmetic

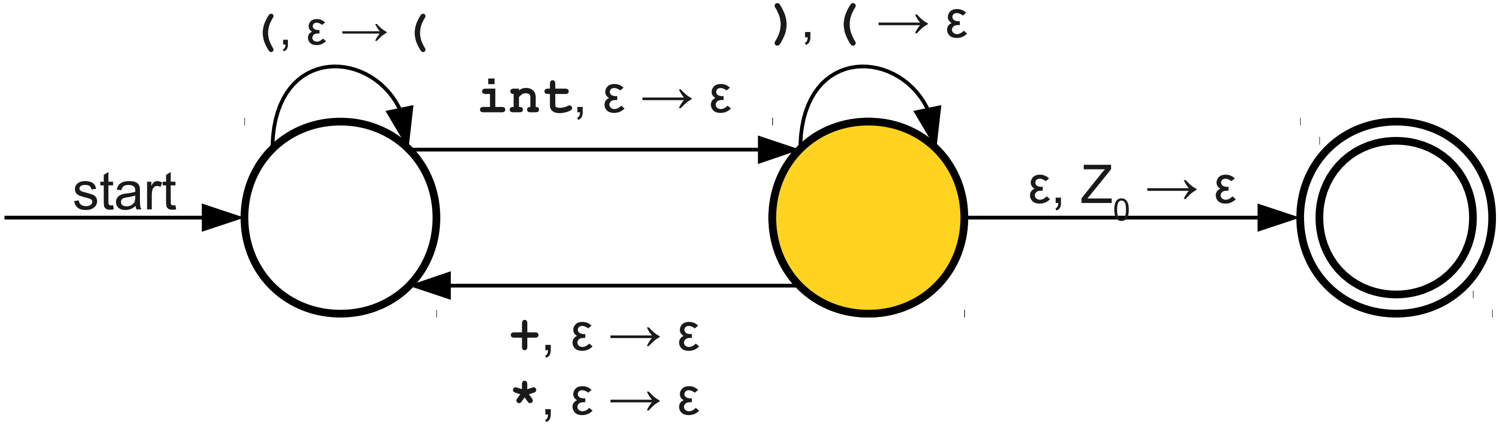


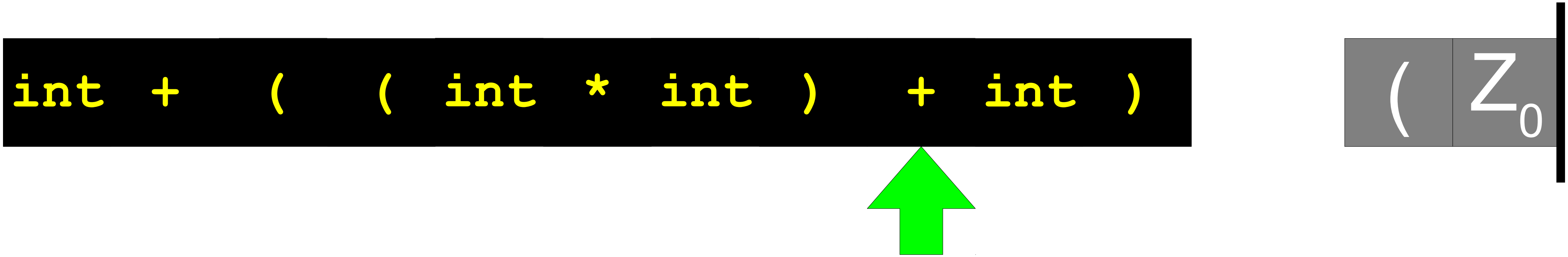


A PDA for Arithmetic

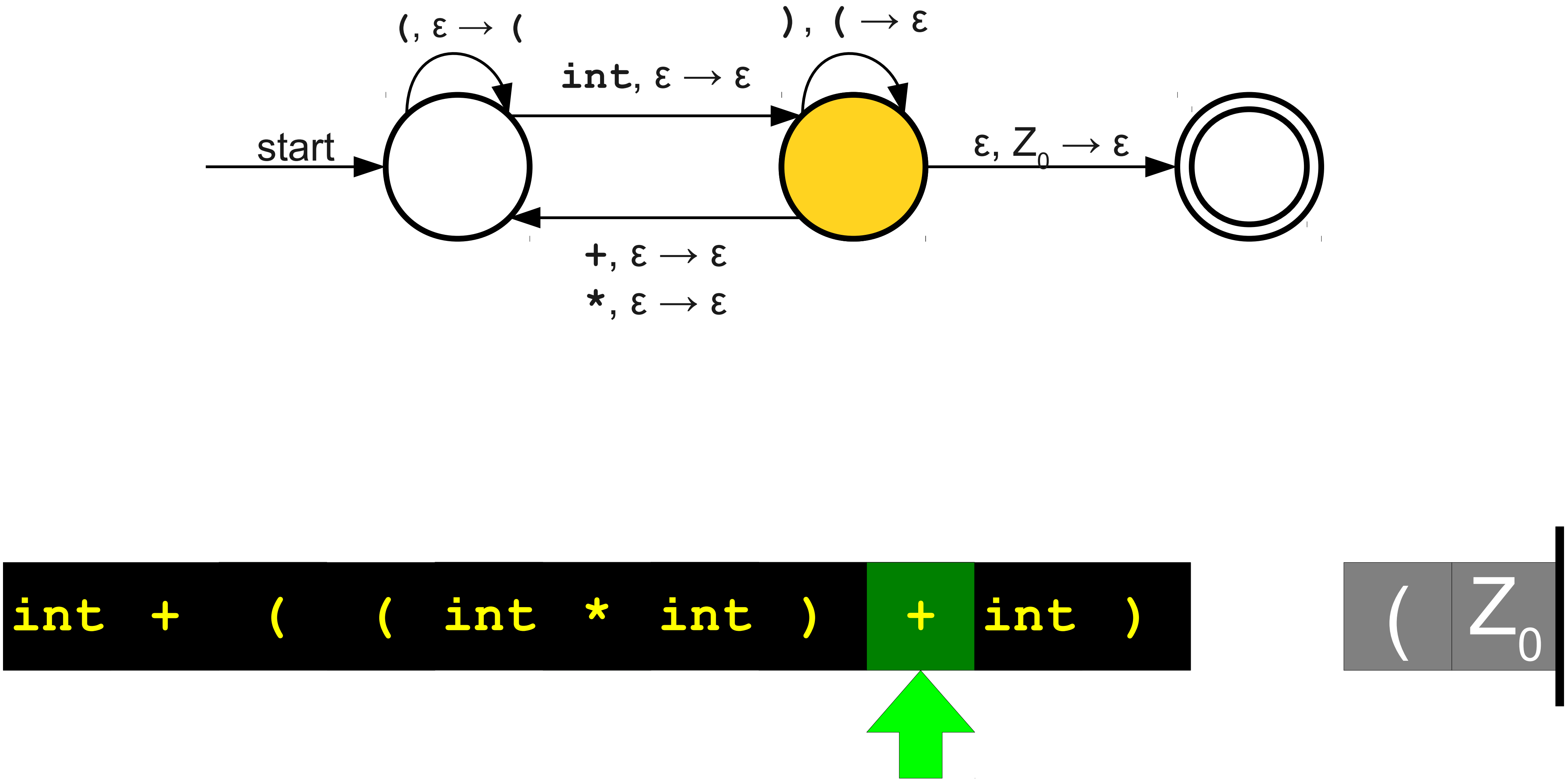


A PDA for Arithmetic

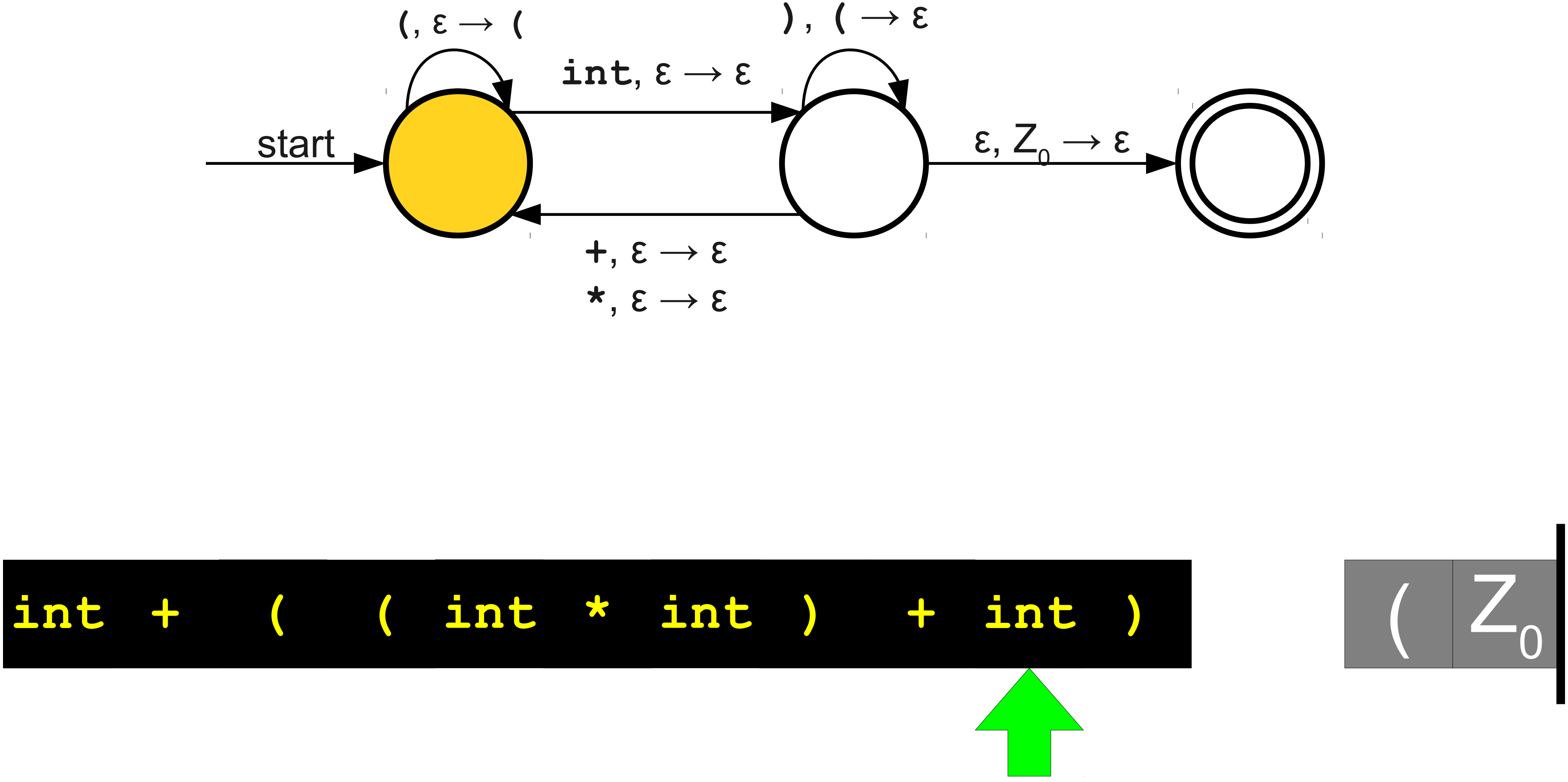




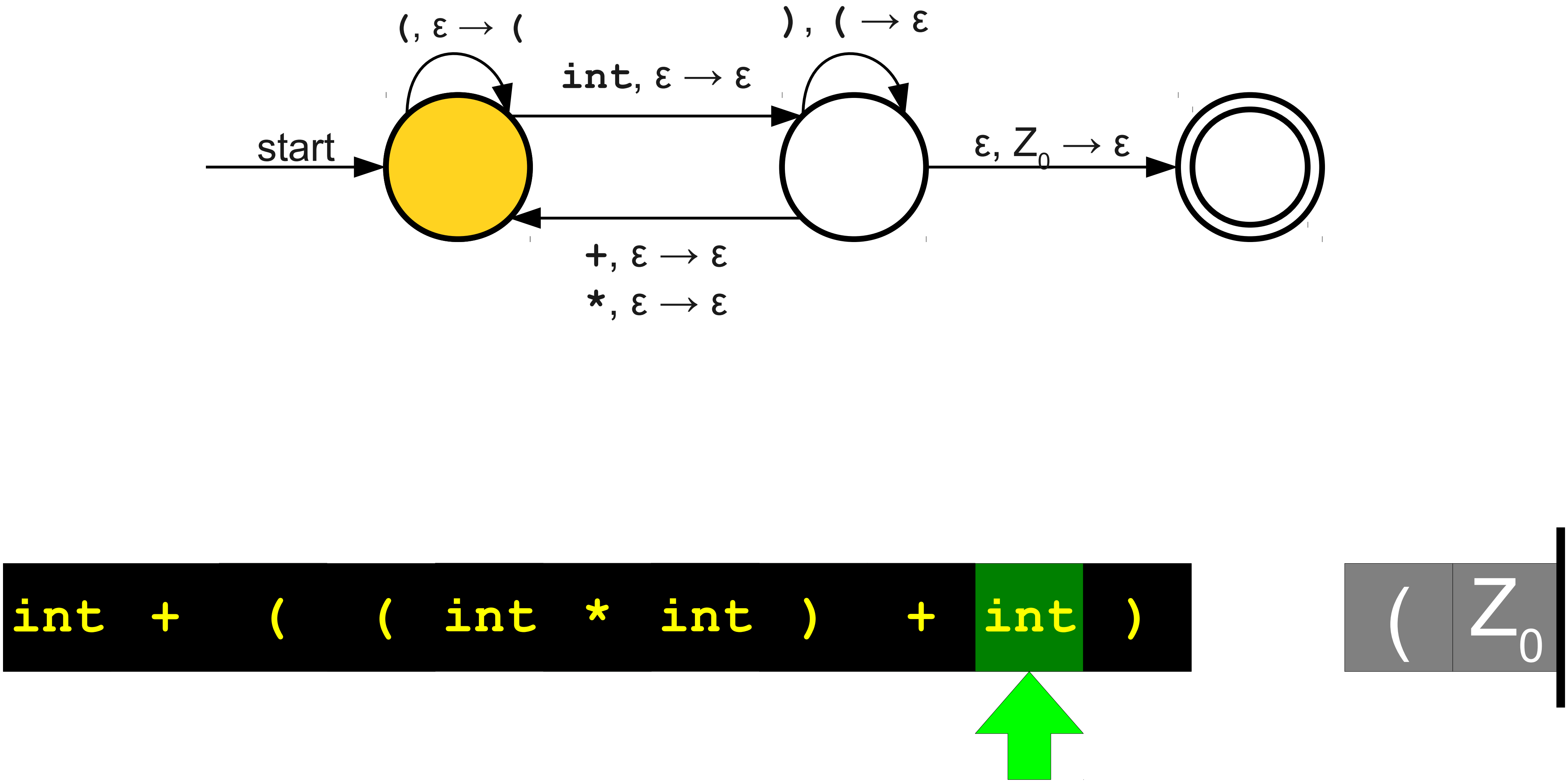
A PDA for Arithmetic



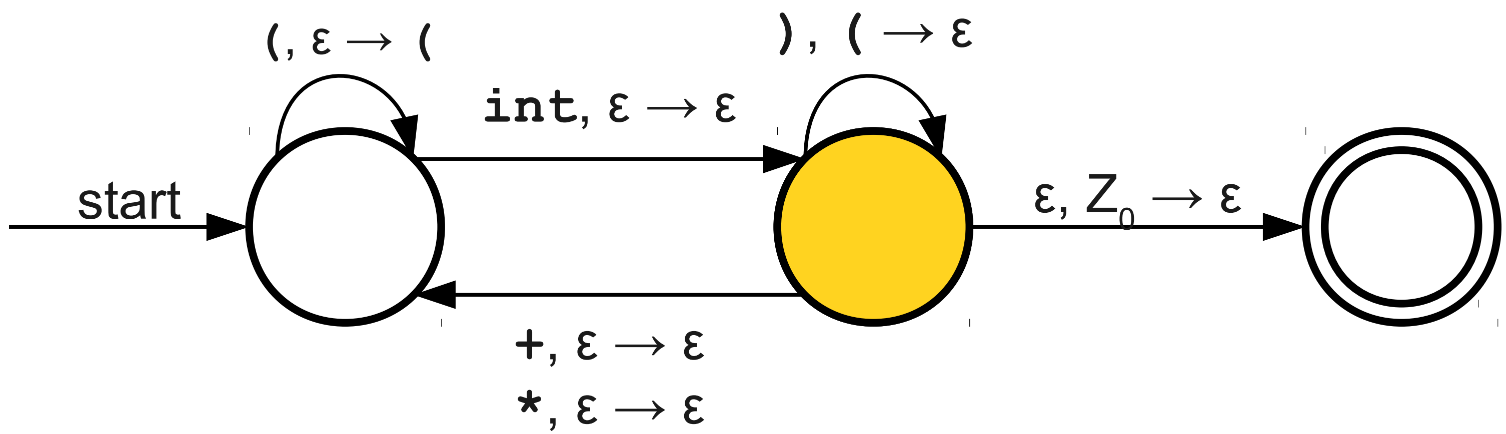
A PDA for Arithmetic

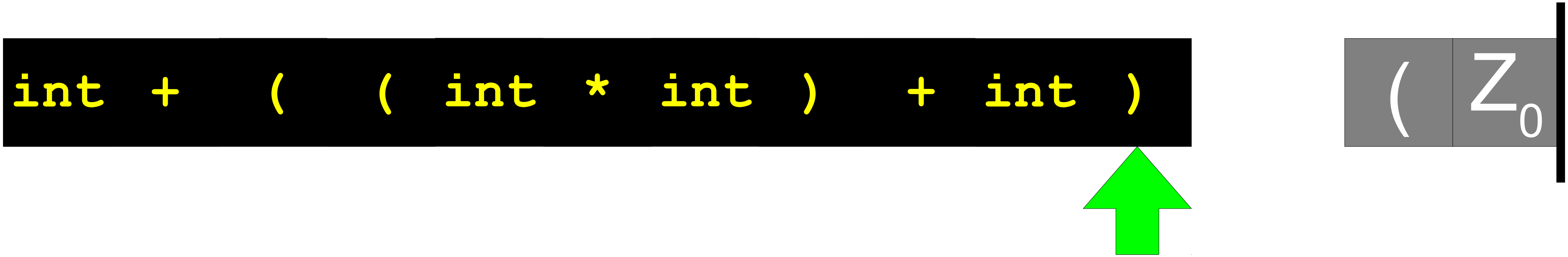


A PDA for Arithmetic

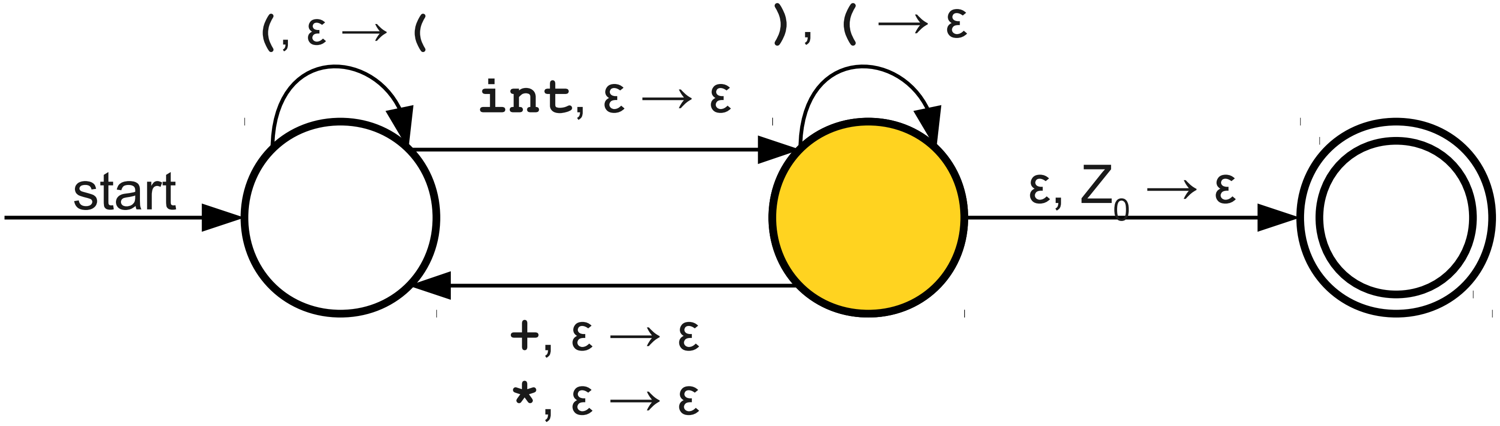


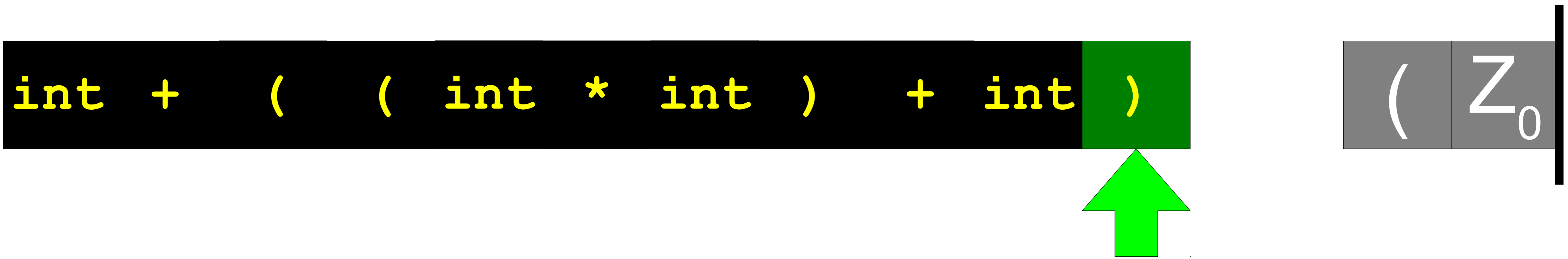
A PDA for Arithmetic



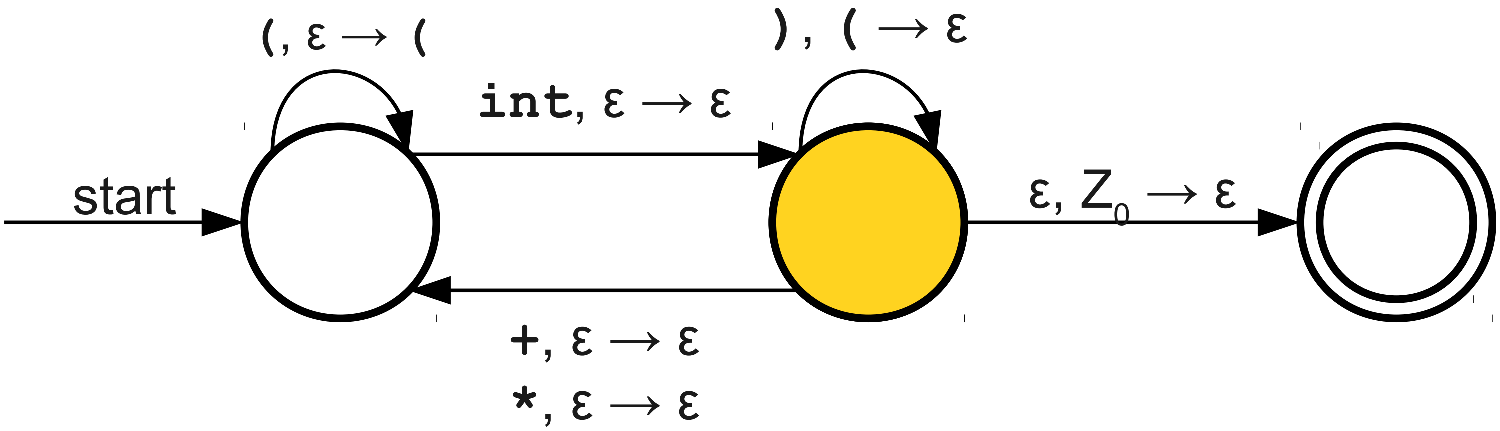


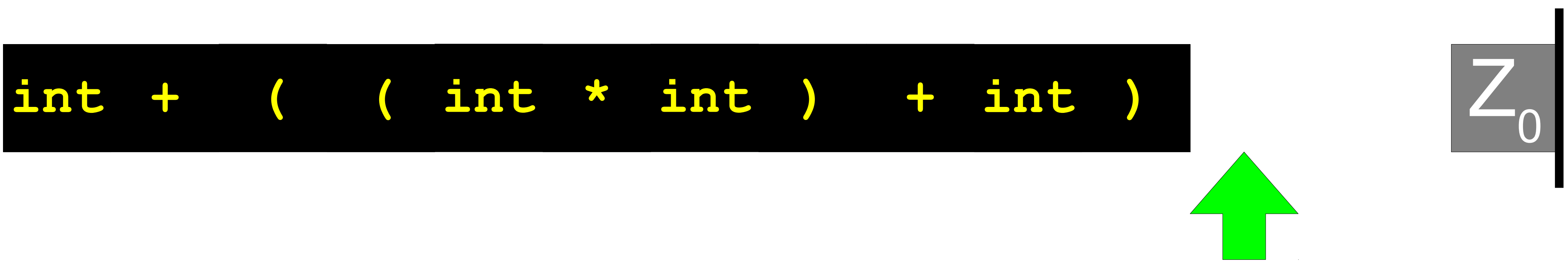
A PDA for Arithmetic



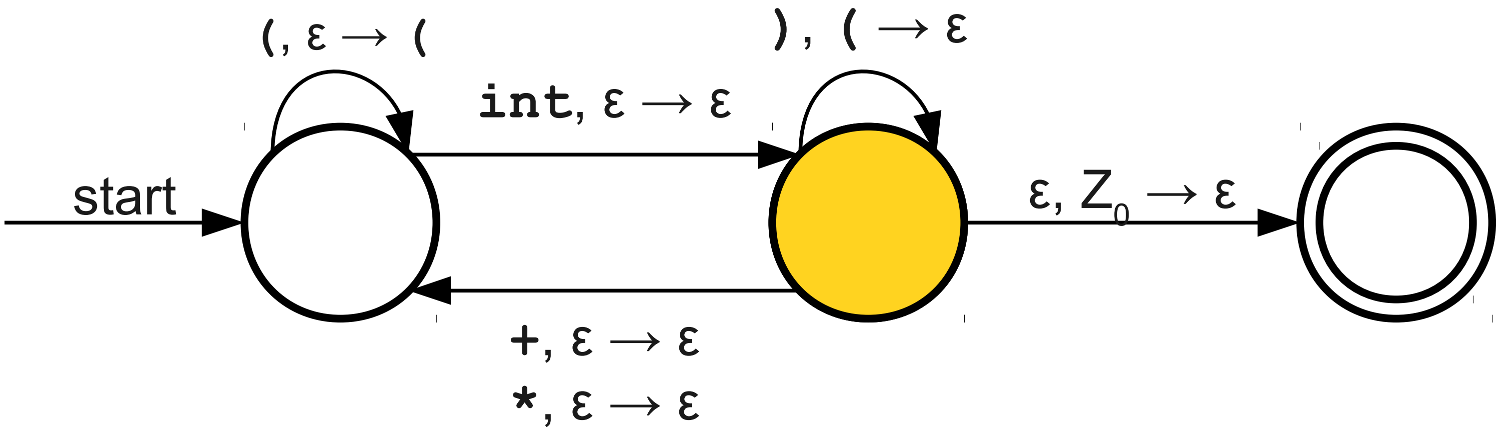


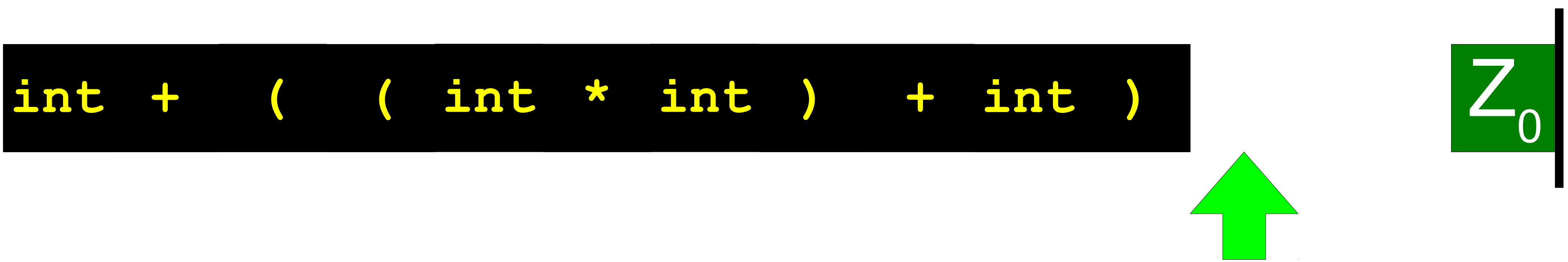
A PDA for Arithmetic



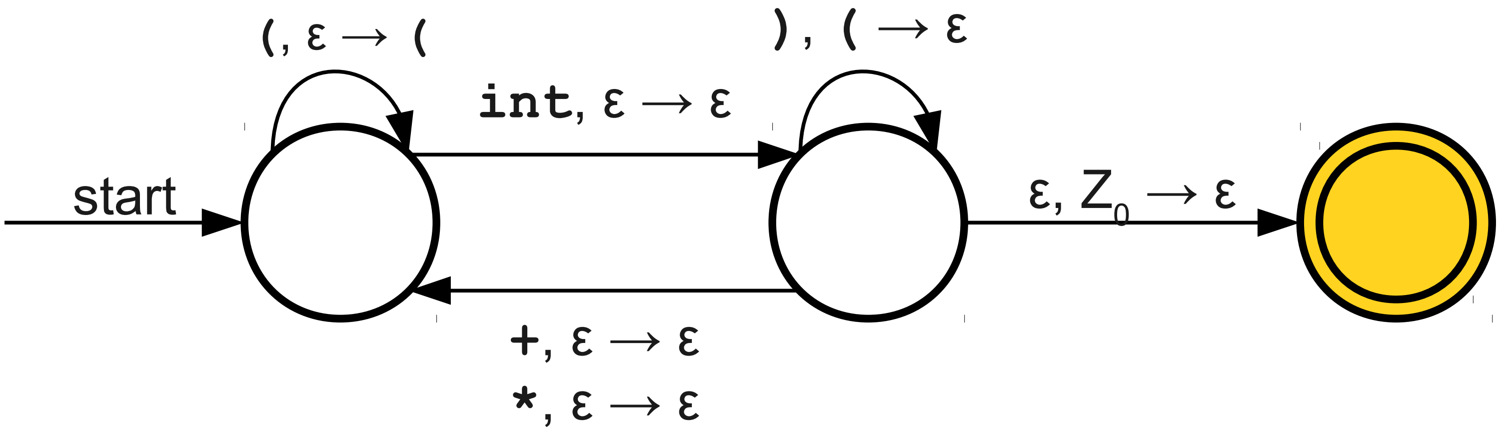


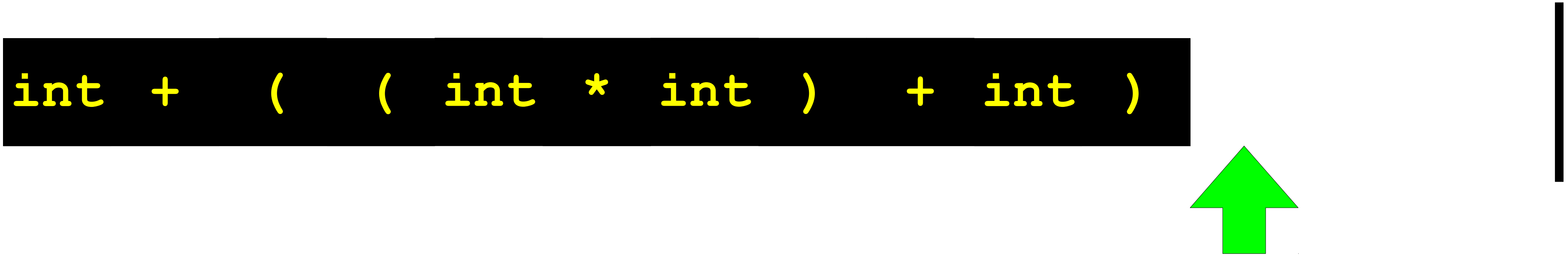
A PDA for Arithmetic



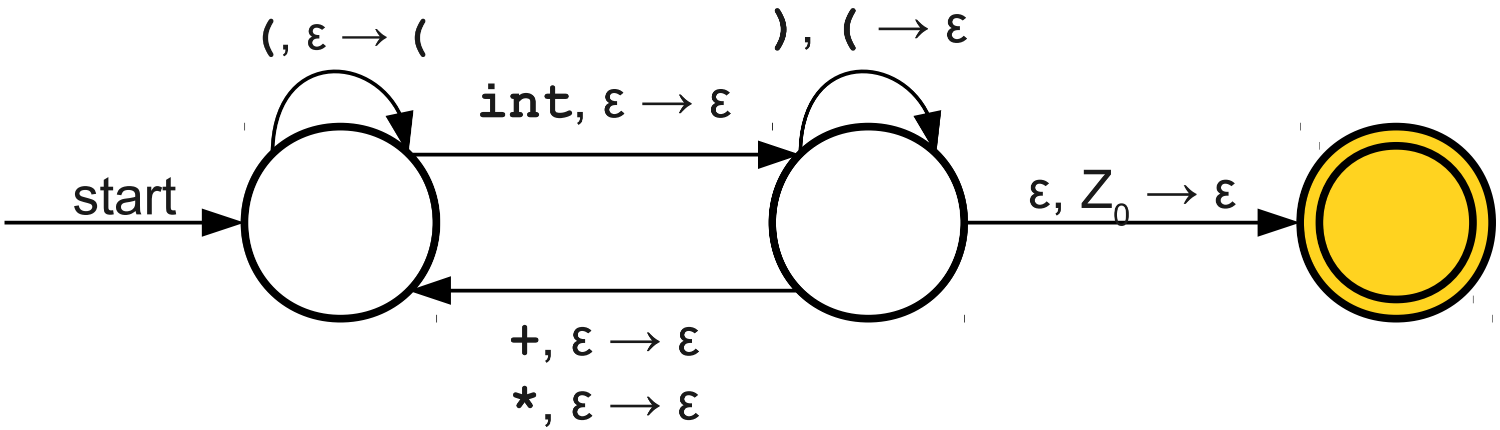


A PDA for Arithmetic

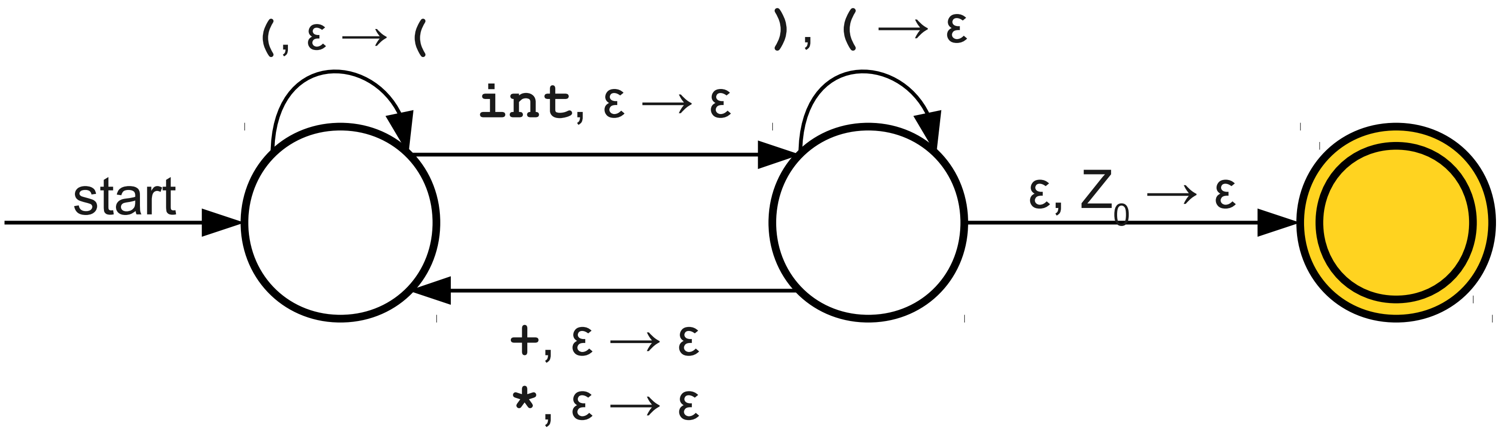




A PDA for Arithmetic



A PDA for Arithmetic



`int + ( ( int * int ) + int )`

Why PDAs Matter

- Recall: A language is context-free iff there is some CFG that generates it.
- **Important, non-obvious theorem:** A language is context-free iff there is some PDA that recognizes it.
- Need to prove two directions:
  - If L is context-free, then there is a PDA for it.
  - If there is a PDA for L, then L is context-free.
- Part (1) is absolutely beautiful and we'll see it in a second.
- Part (2) is brilliant, but a bit too involved for lecture (you should read this in Sipser).

From CFGs to PDAs

- ***Theorem:*** If G is a CFG for a language L, then there exists a PDA for L as well.
- **Idea:** Build a PDA that simulates expanding out the CFG from the start symbol to some particular string.
- Stack holds the part of the string we haven't matched yet.

From CFGs to PDAs

- Example: Let $\Sigma = \{ \mathtt{1}, \mathtt{\geq} \}$ and let $GE = \{ \mathtt{1}^m \mathtt{\geq} \mathtt{1}^n \mid m, n \in \mathbb{N} \wedge m \geq n \}$
  - $\mathtt{111\geq11} \in GE$
  - $\mathtt{11\geq11} \in GE$
  - $\mathtt{1111\geq11} \in GE$
  - $\mathtt{\geq} \in GE$
- One CFG for *GE* is the following:

$$\mathbf{S} \rightarrow \mathtt{1}\mathbf{S}\mathtt{1} \mid \mathtt{1}\mathbf{S} \mid \mathtt{\geq}$$

- How would we build a PDA for *GE*?

From CFGs to PDAs

$S \to 1S1$

$S \to 1S$

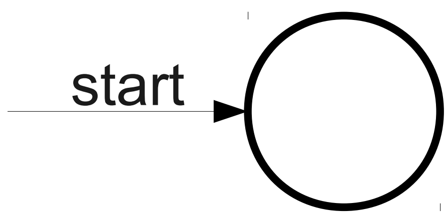
$S \to \geq$

From CFGs to PDAs

$S \rightarrow 1S1$

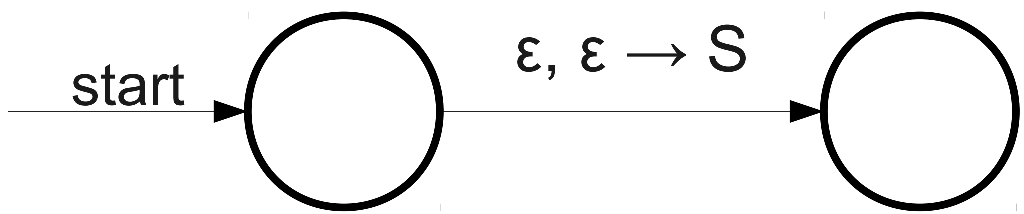
$S \rightarrow 1S$

$S \rightarrow \geq$



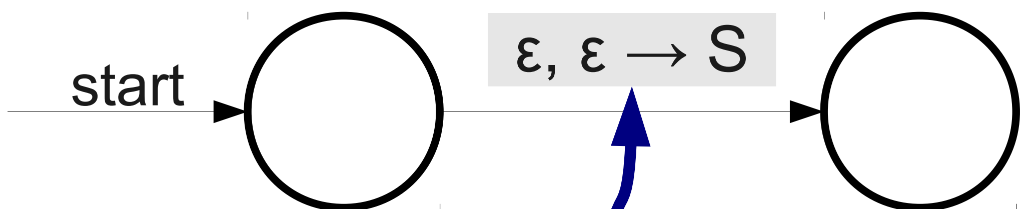
From CFGs to PDAs

S → 1S1
S → 1S
S → ≥



From CFGs to PDAs

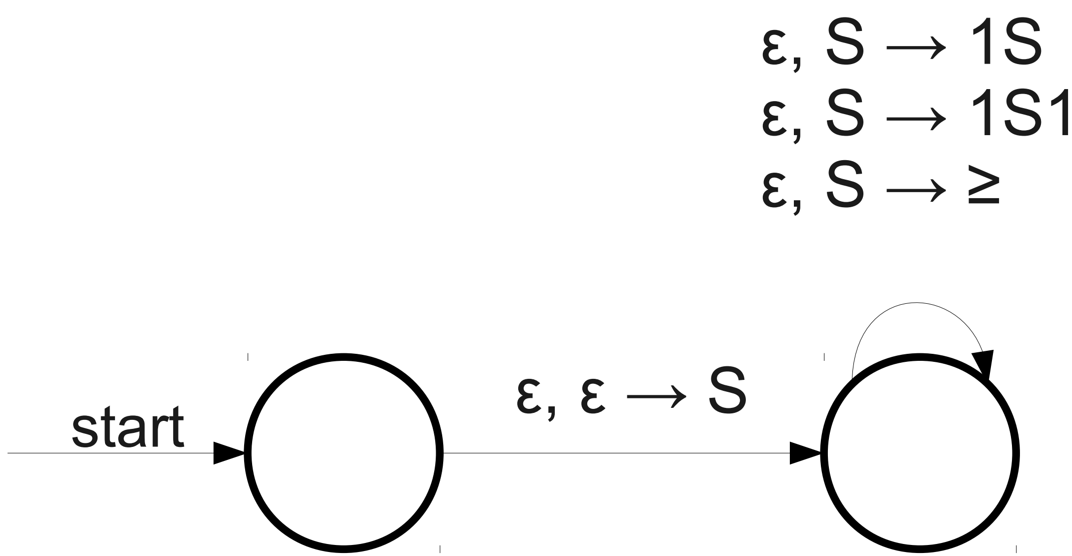
$S \rightarrow 1S1$
$S \rightarrow 1S$
$S \rightarrow$ ≥



We begin by putting the start symbol of the grammar onto the stack so that we can begin applying productions.

From CFGs to PDAs

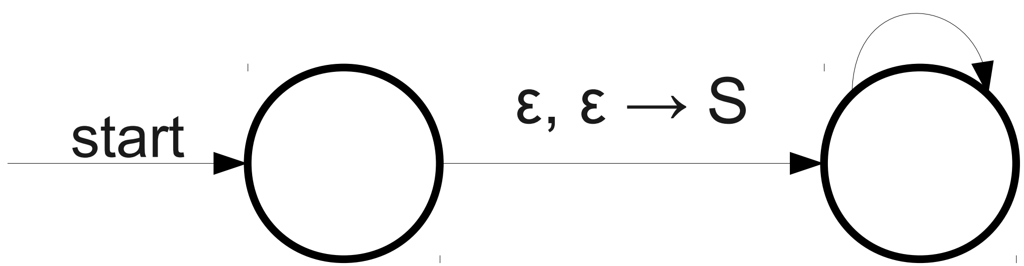
S → 1S1
S → 1S
S → ≥



From CFGs to PDAs

S → **1S1**
S → **1S**
S → **≥**

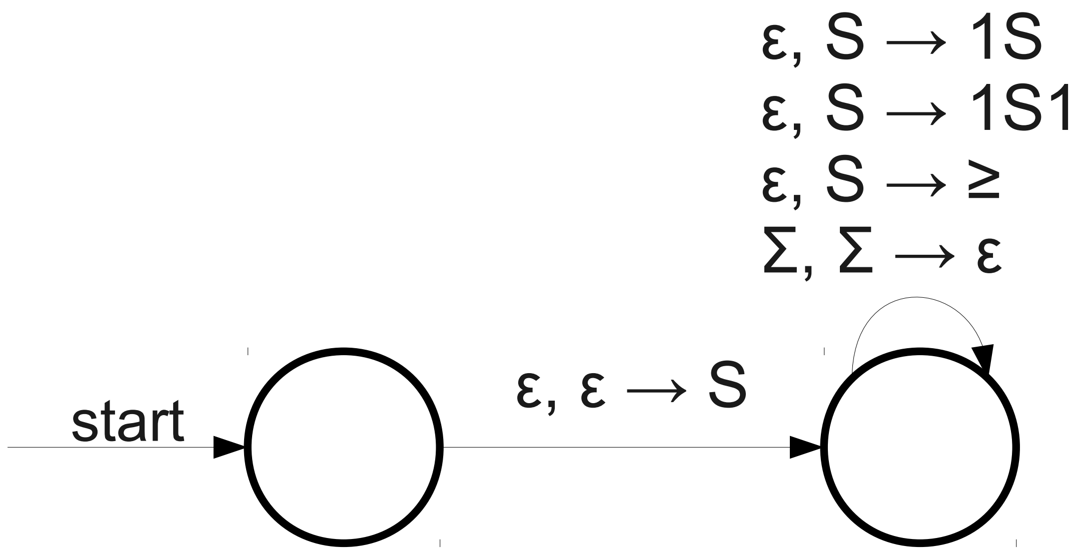
ε, S → 1S
ε, S → 1S1
ε, S → ≥



These transitions allow us to nondeterministically guess which production to use when the top of the stack is a nonterminal.

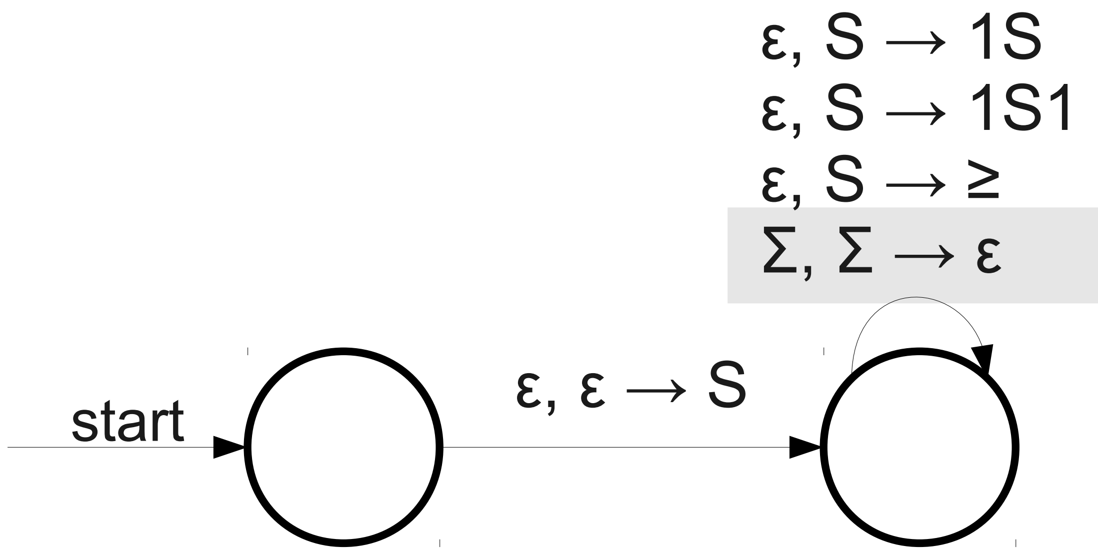
From CFGs to PDAs

S → 1S1
S → 1S
S → ≥



From CFGs to PDAs

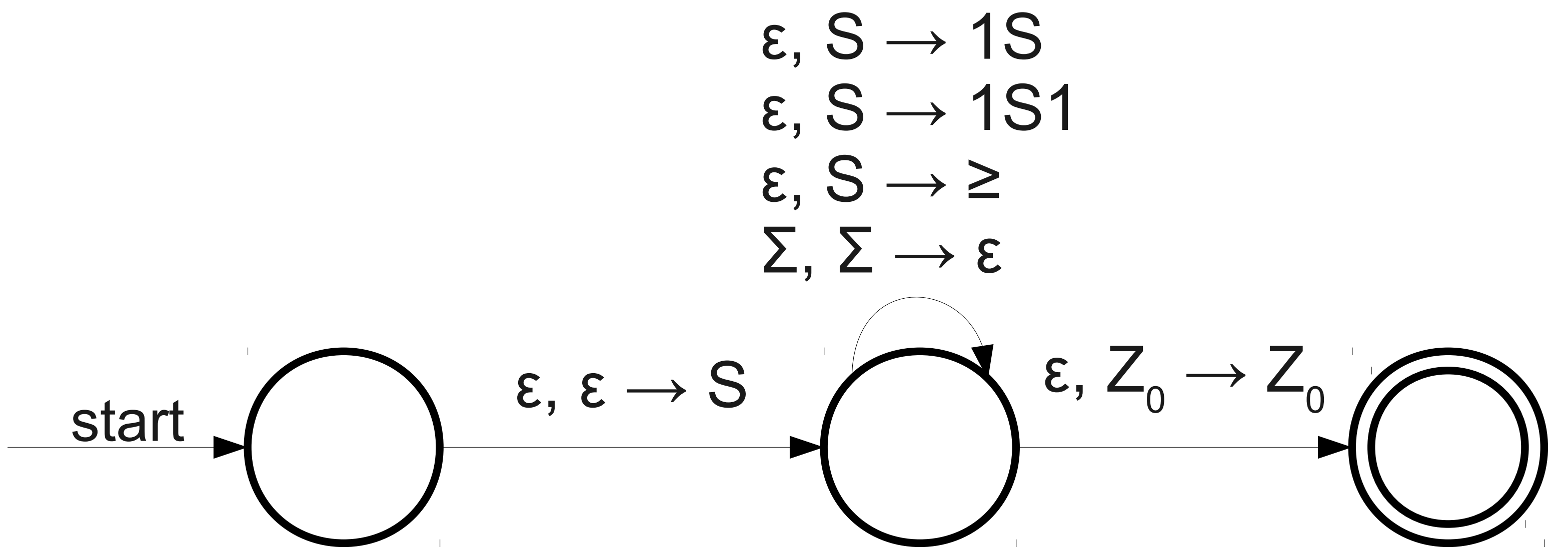
S → 1S1
S → 1S
S → ≥



Once we have guessed the right production, this rule lets us match the next character from the input with the next terminal we produced.

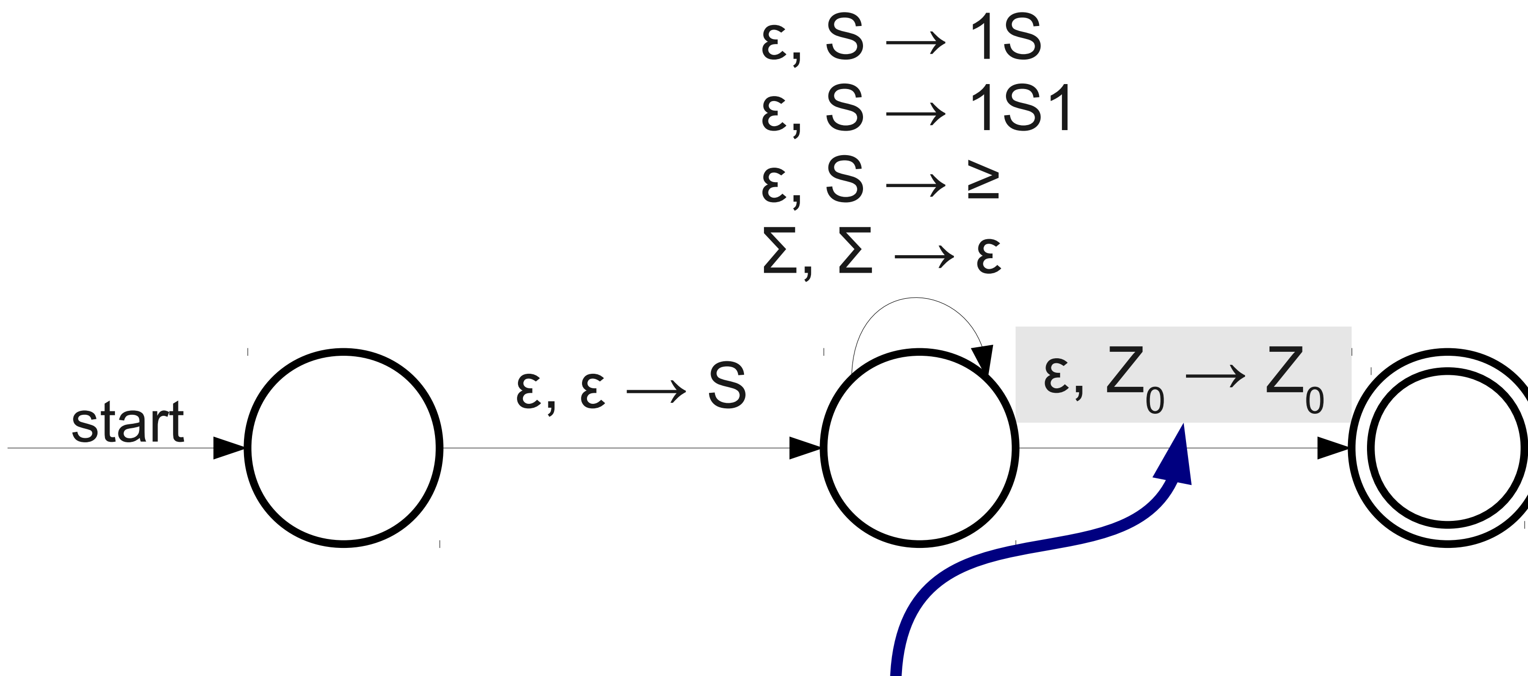
From CFGs to PDAs

$S \rightarrow 1S1$
$S \rightarrow 1S$
$S \rightarrow$ ≥



From CFGs to PDAs

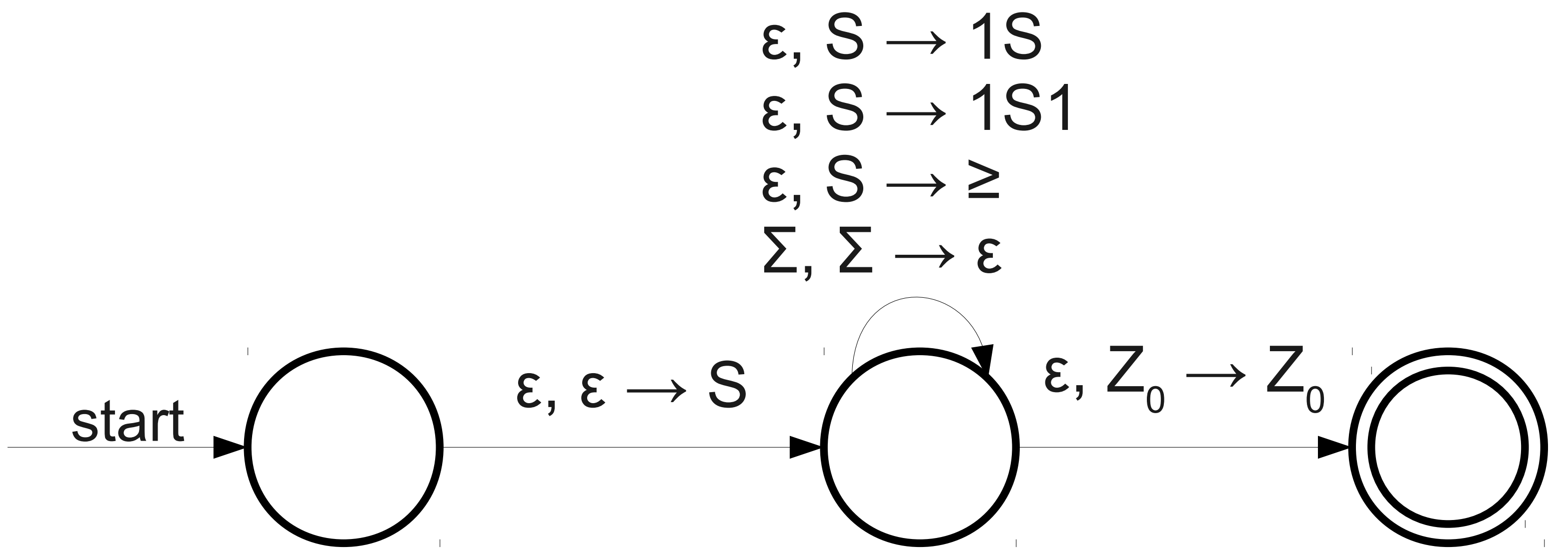
S → **1S1**
S → **1S**
S → **≥**



Once we have fully expanded out all nonterminals and matched all the terminals on the stack, we can transition into the accepting state.

From CFGs to PDAs

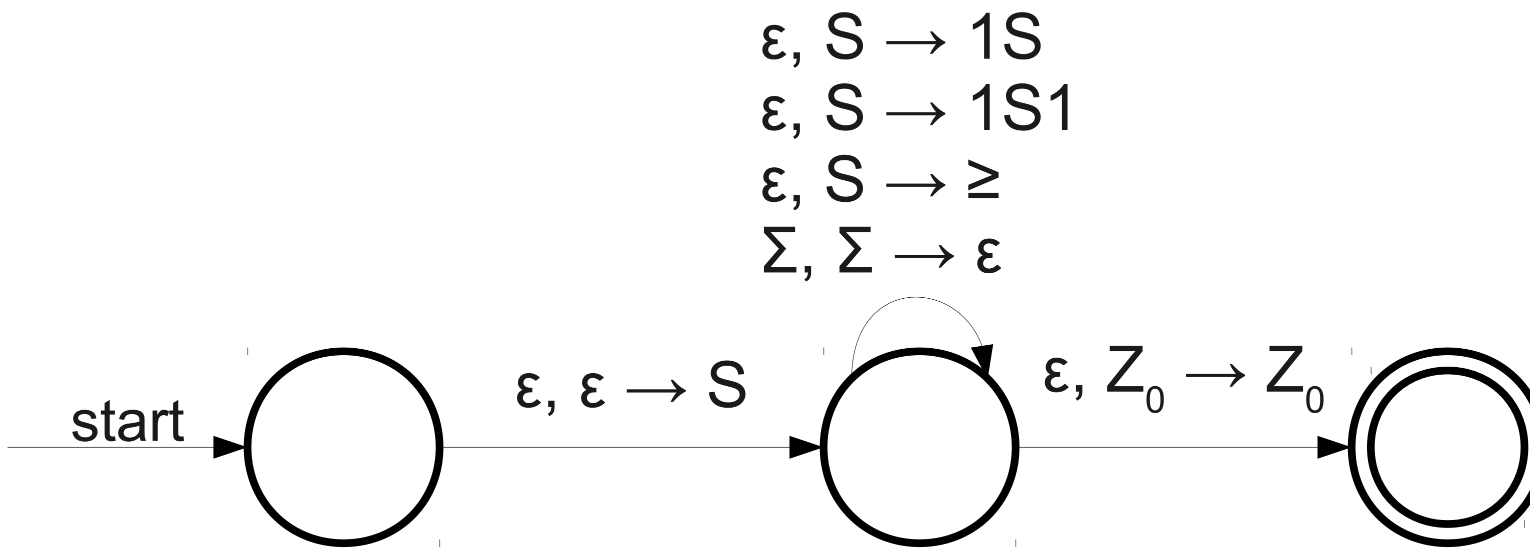
S → **1S1**
S → **1S**
S → **≥**

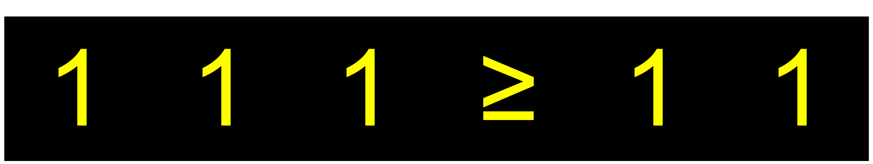


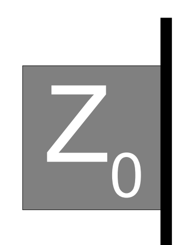
1 1 1 ≥ 1 1

From CFGs to PDAs

S → **1S1**
S → **1S**
S → **≥**

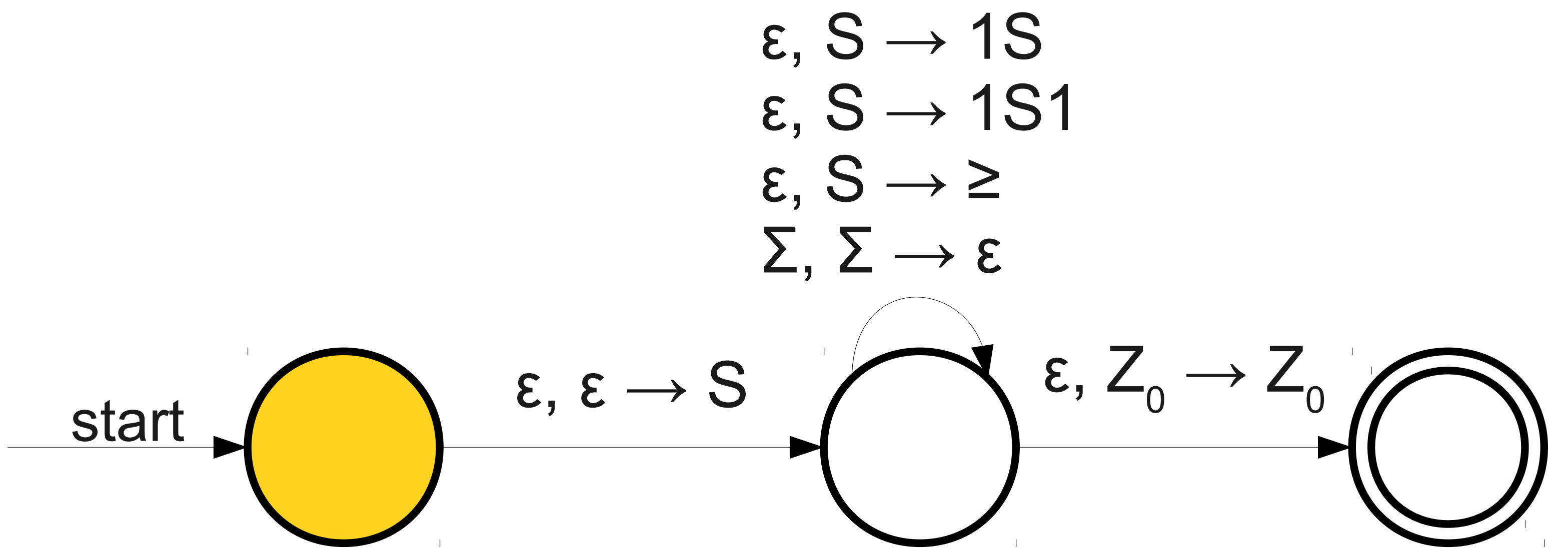




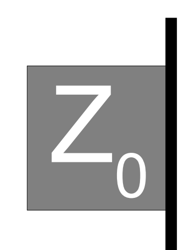


From CFGs to PDAs

$S \rightarrow 1S1$
$S \rightarrow 1S$
$S \rightarrow$ ≥

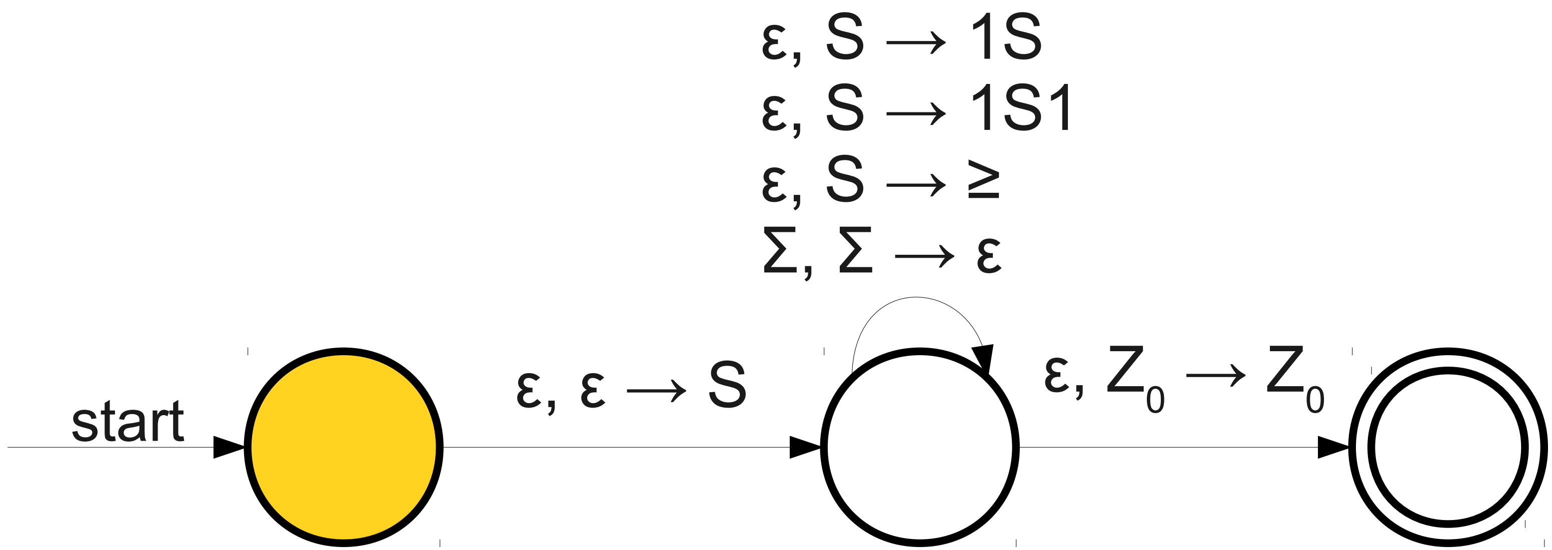


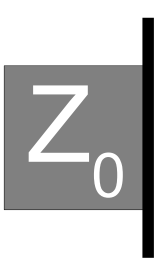
1 1 1 ≥ 1 1



From CFGs to PDAs

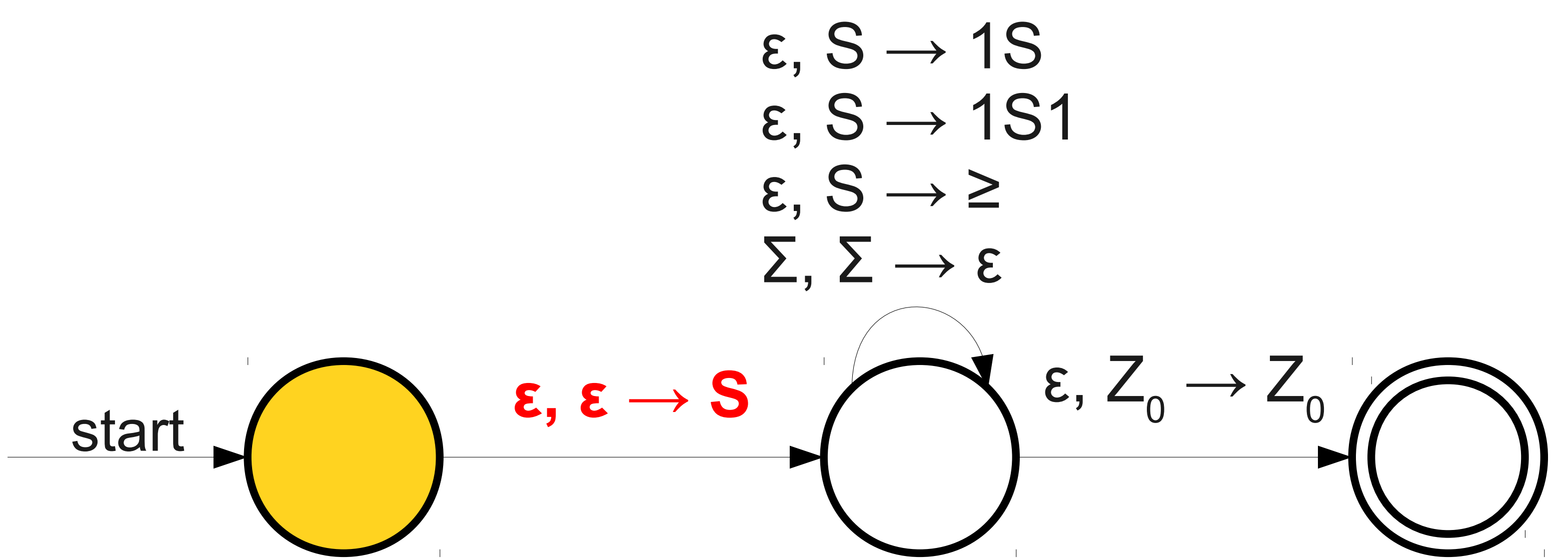
$S \rightarrow 1S1$
$S \rightarrow 1S$
$S \rightarrow ≥$





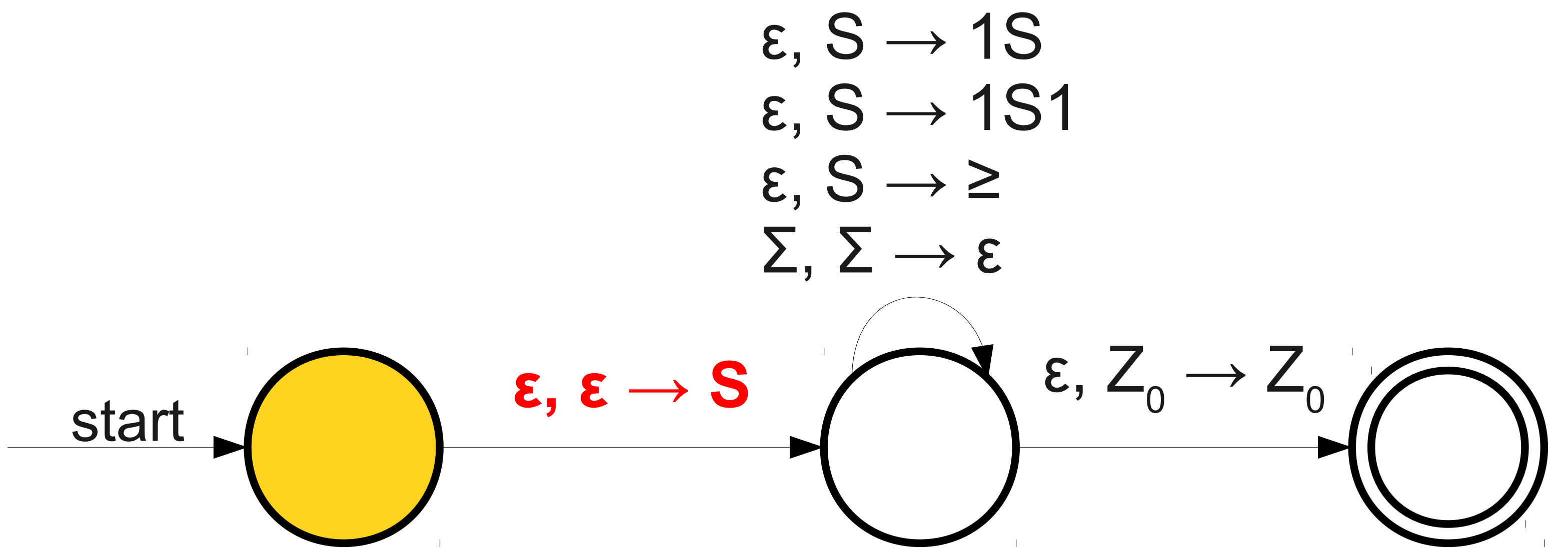
From CFGs to PDAs

S → 1S1
S → 1S
S → ≥



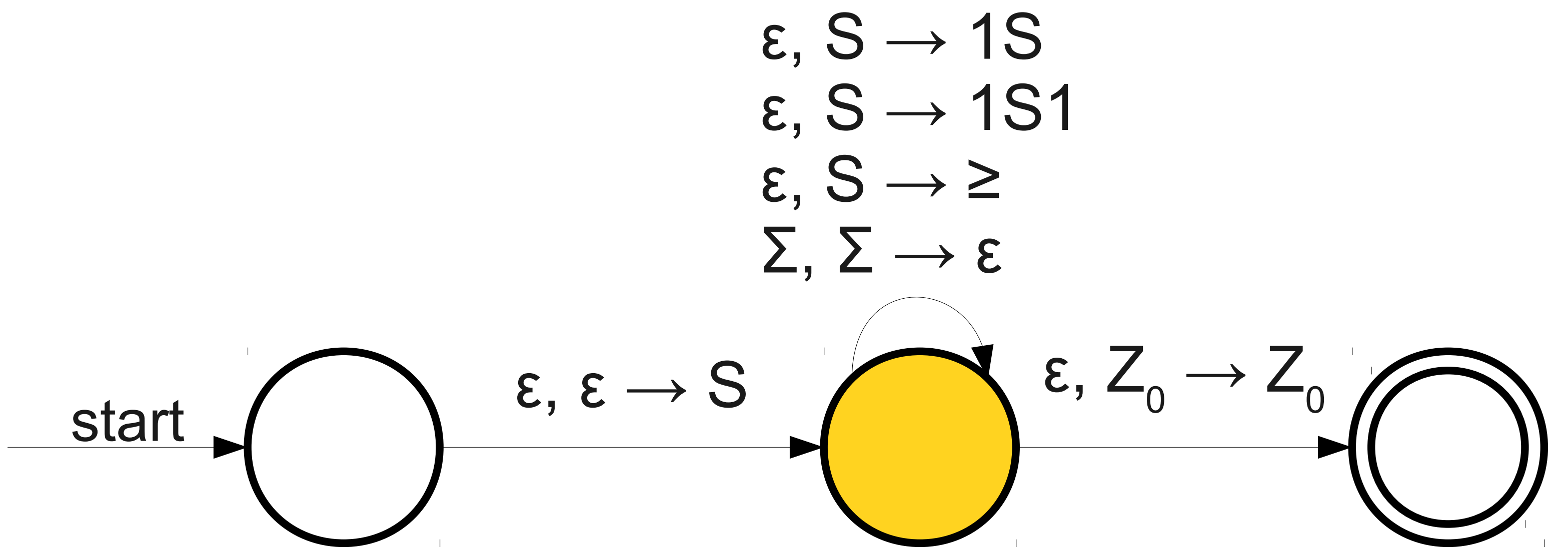
From CFGs to PDAs

S → 1S1
S → 1S
S → ≥



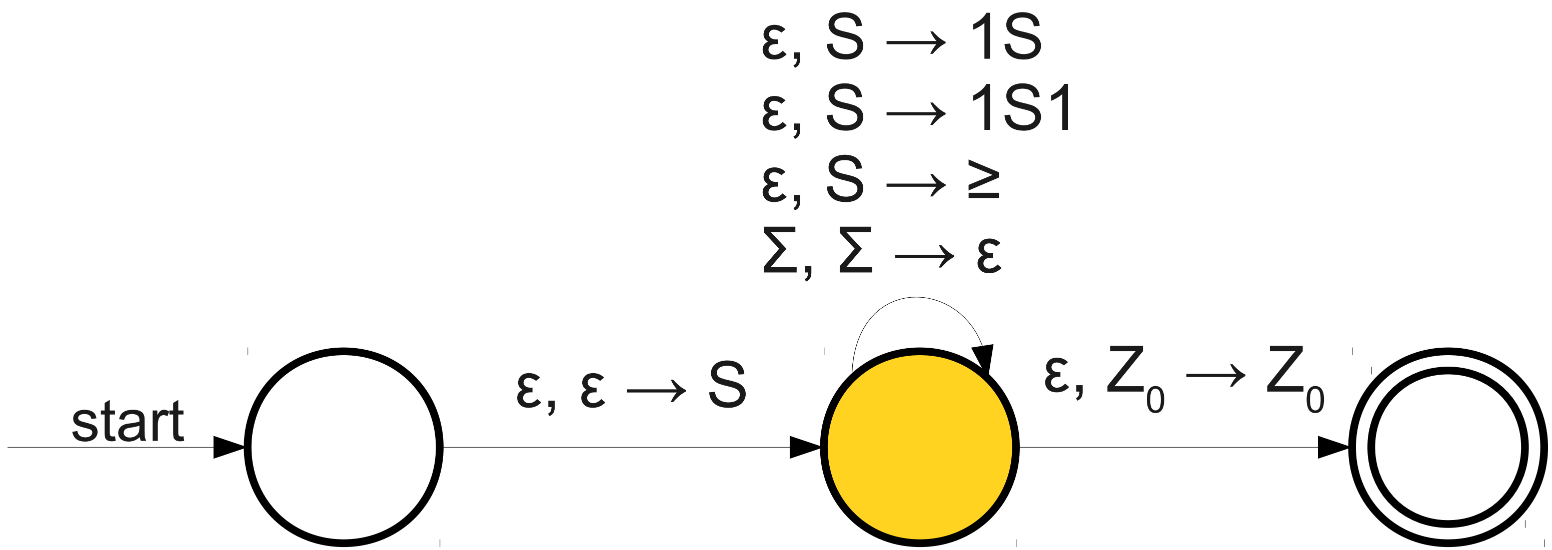
From CFGs to PDAs

S → 1S1
S → 1S
S → ≥



From CFGs to PDAs

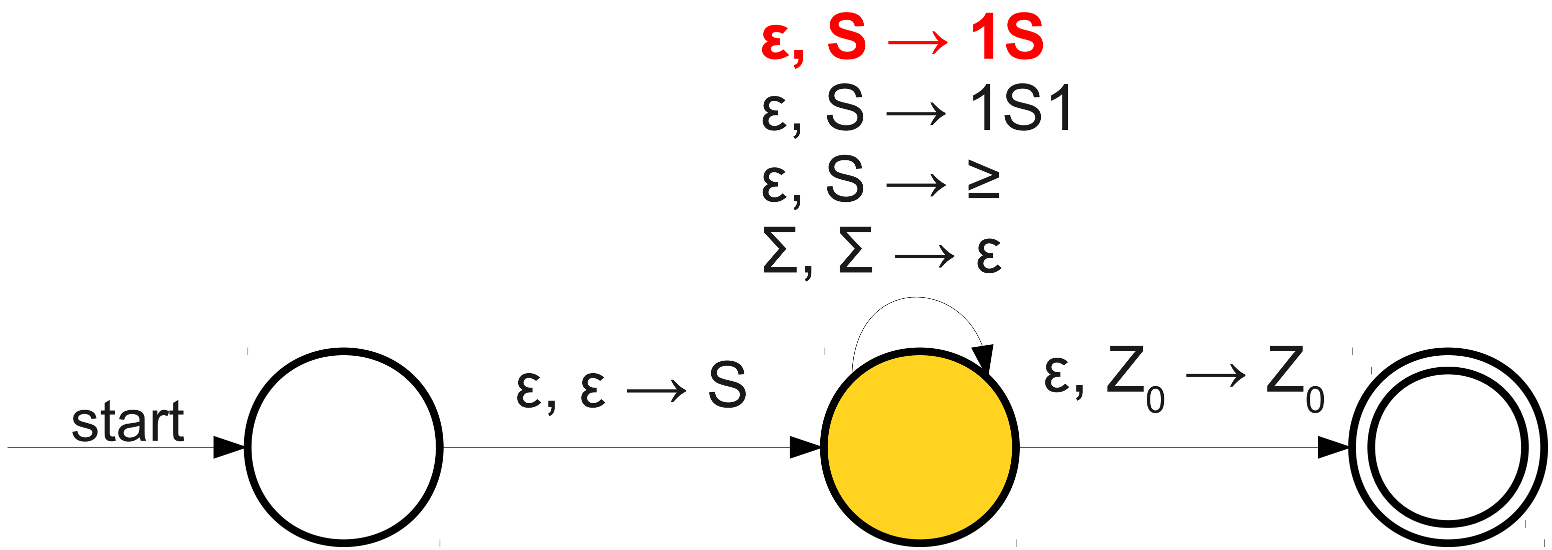
$S \rightarrow 1S1$
$S \rightarrow 1S$
$S \rightarrow ≥$



Now that the stack top is a nonterminal, we guess which production to use.

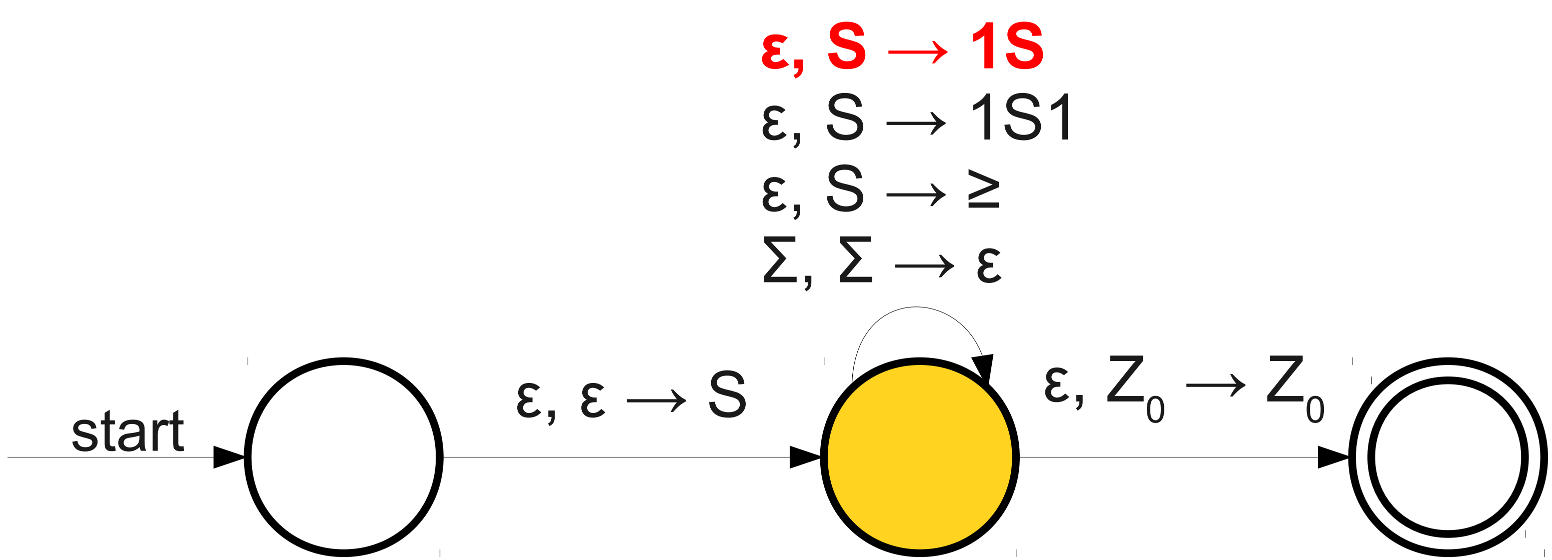
From CFGs to PDAs

S → 1S1
S → 1S
S → ≥



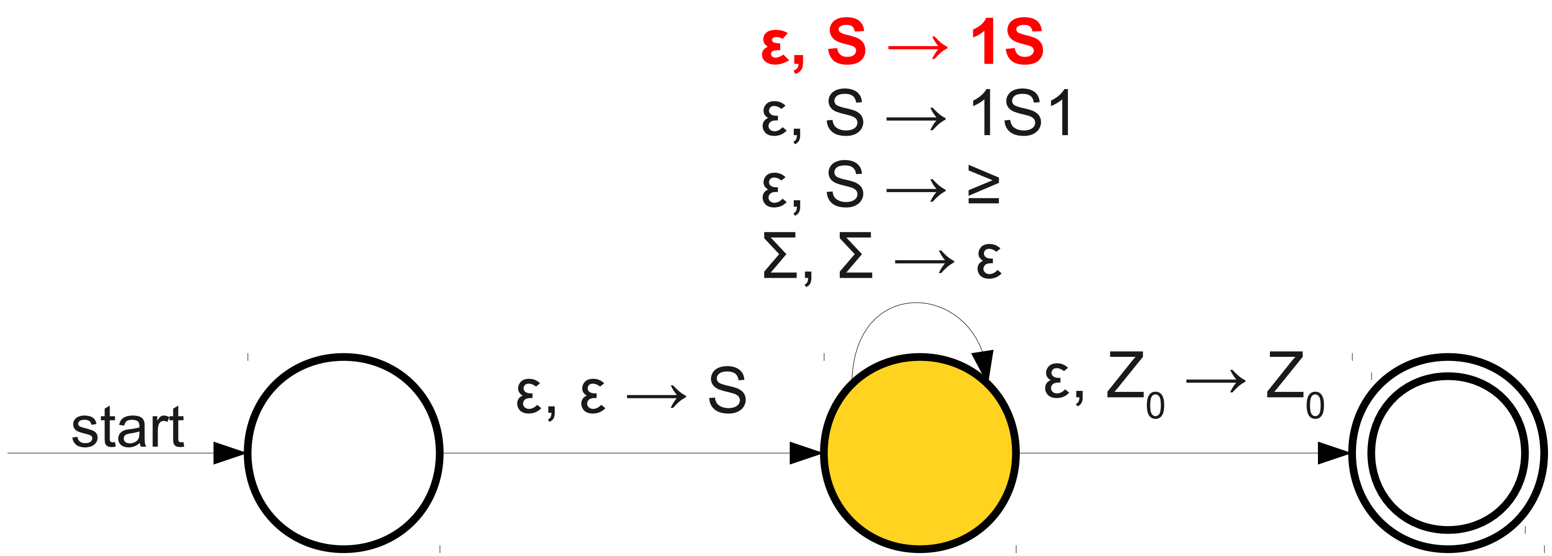
From CFGs to PDAs

S → 1S1
S → 1S
S → ≥



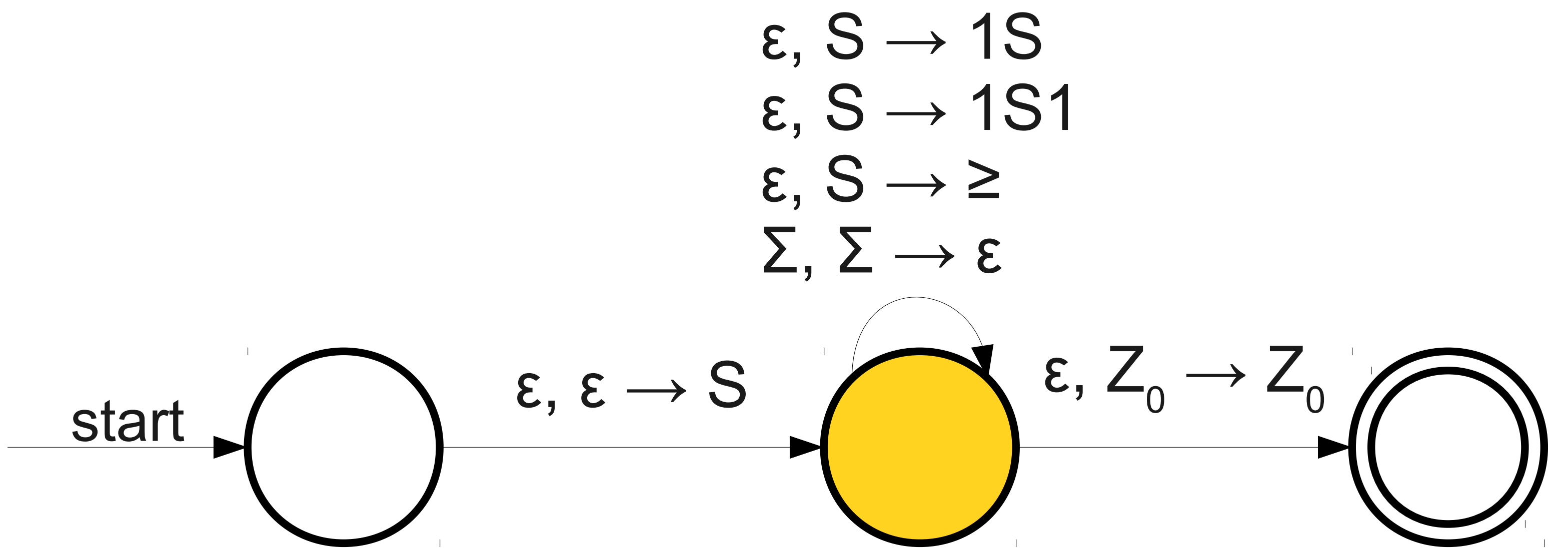
From CFGs to PDAs

S → 1S1
S → 1S
S → ≥



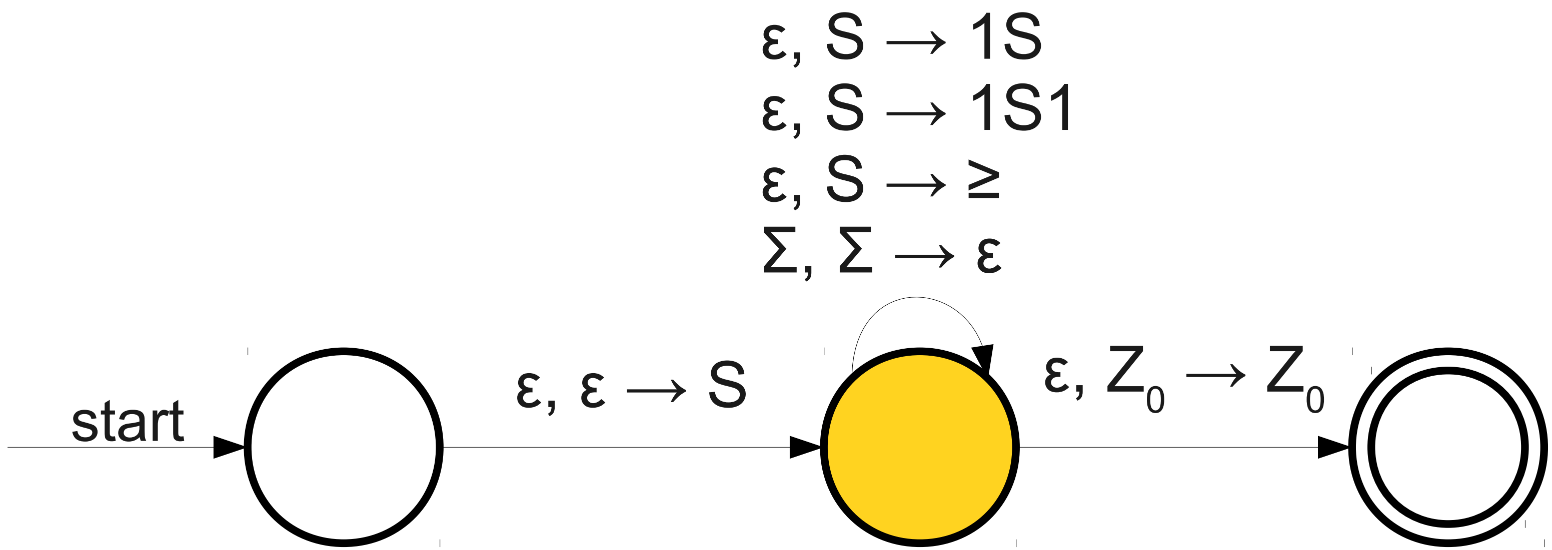
From CFGs to PDAs

S → 1S1
S → 1S
S → ≥



From CFGs to PDAs

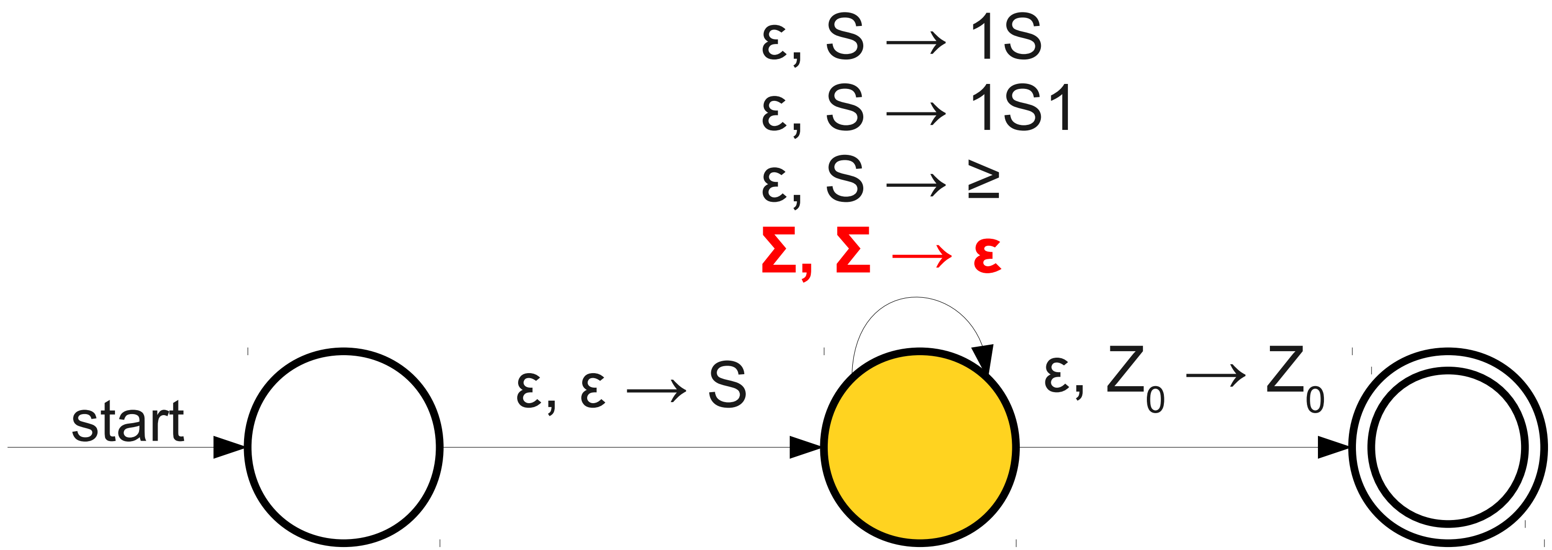
S → 1S1
S → 1S
S → ≥



Since the top of the stack is a terminal, we can match it with the next input symbol.

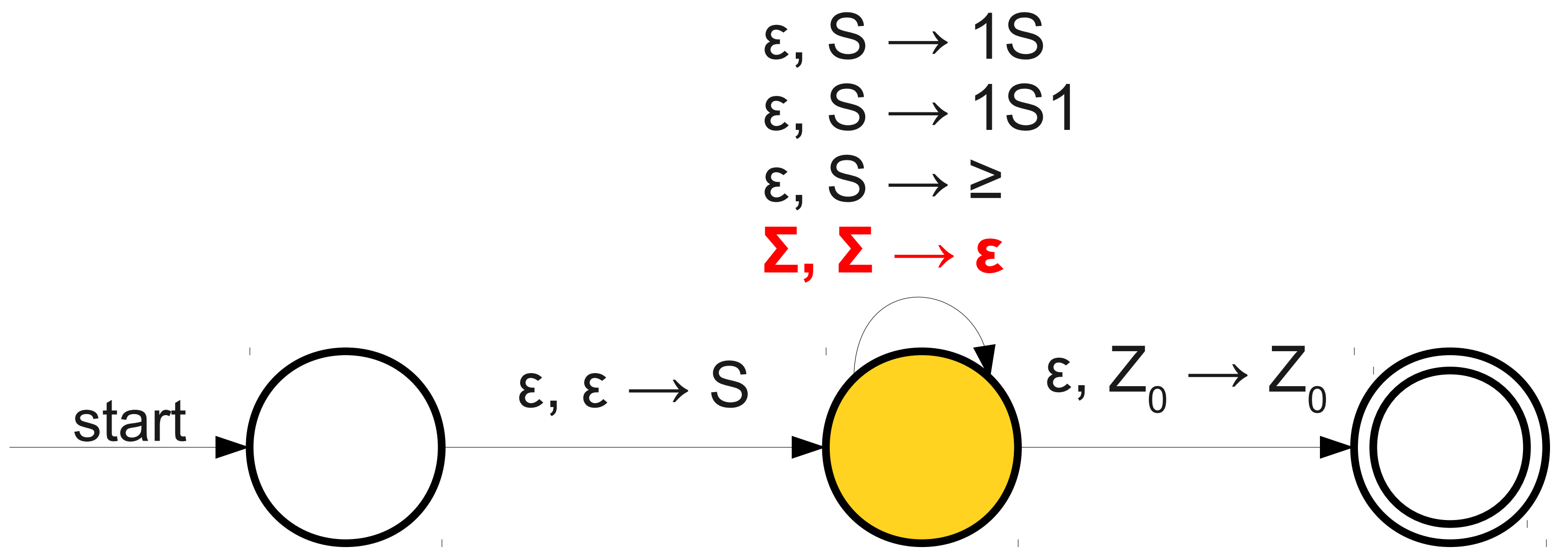
From CFGs to PDAs

S → 1S1
S → 1S
S → ≥



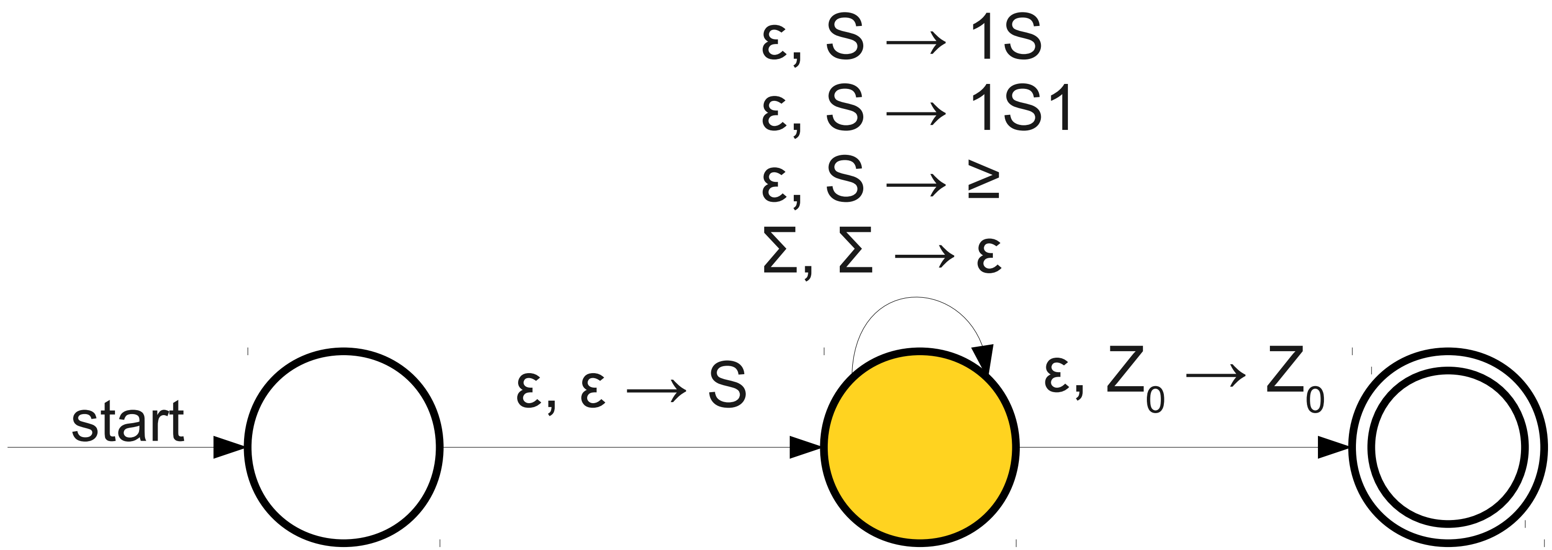
From CFGs to PDAs

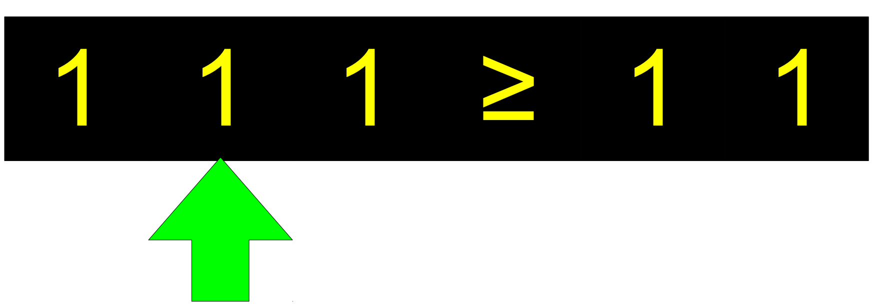
S → 1S1
S → 1S
S → ≥



From CFGs to PDAs

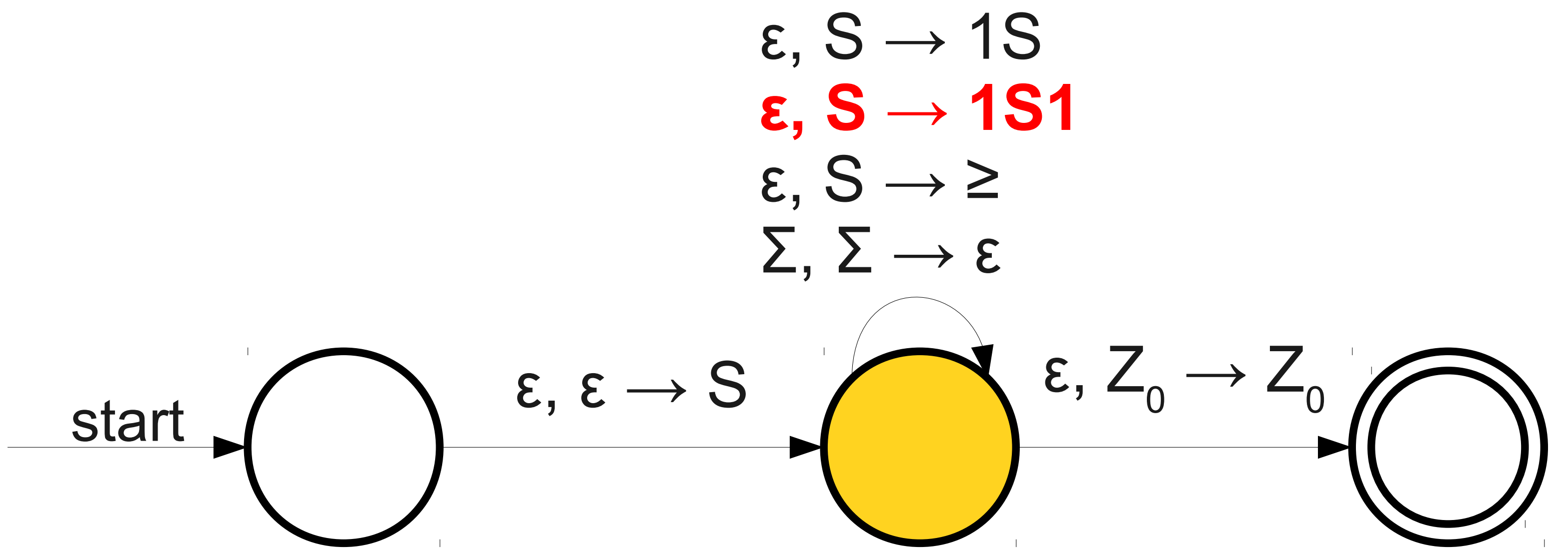
S → 1S1
S → 1S
S → ≥

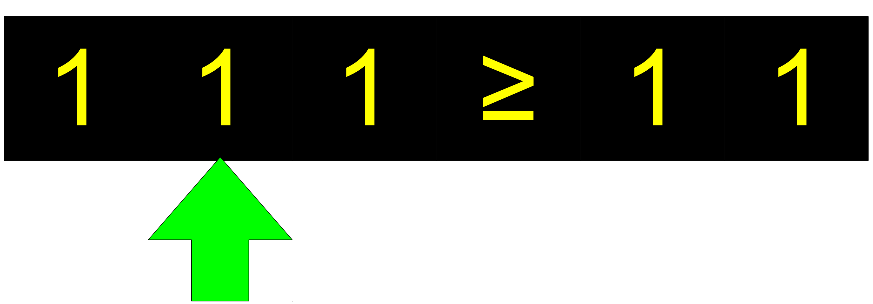




From CFGs to PDAs

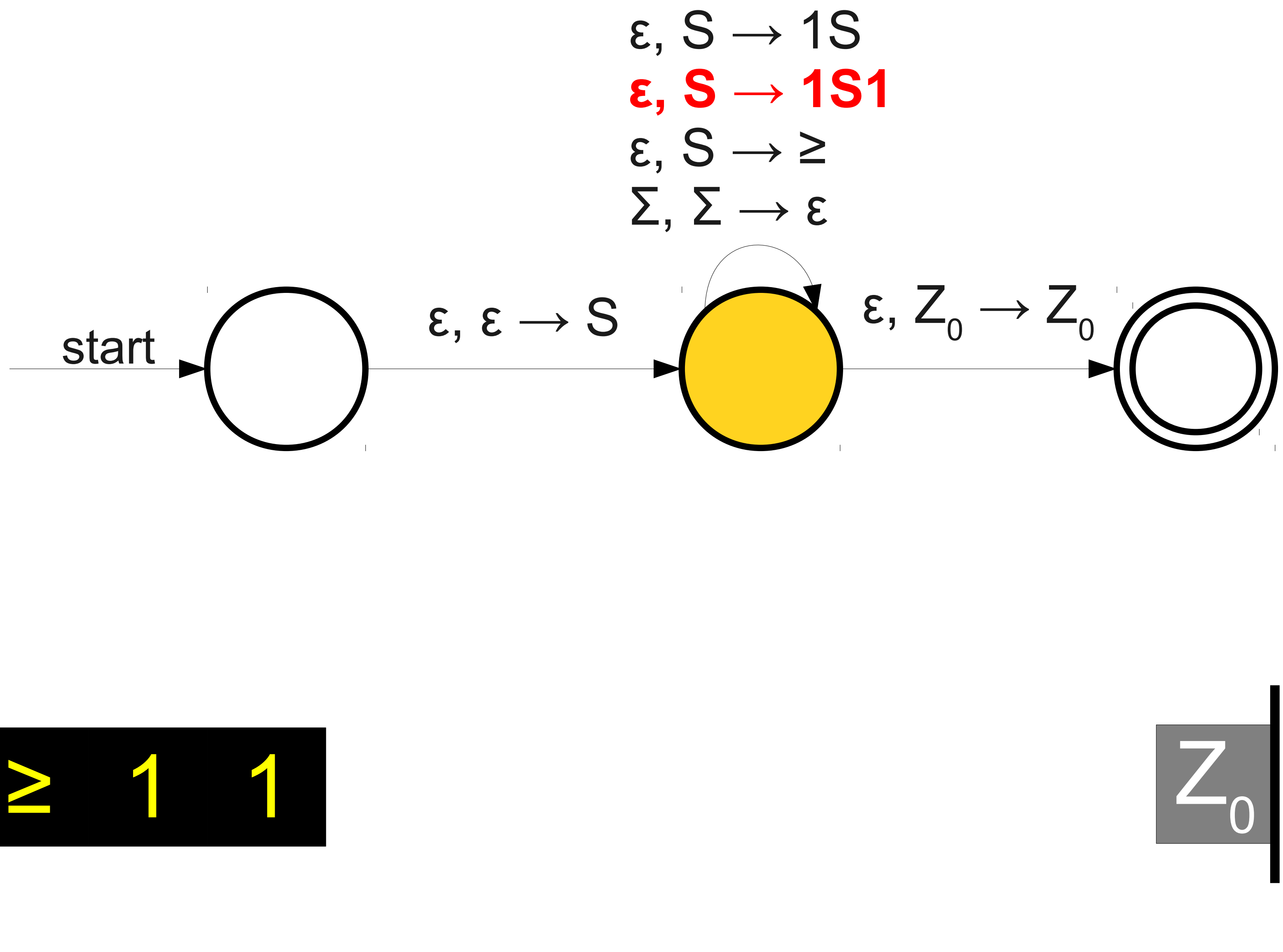
S → **1S1**
S → **1S**
S → **≥**





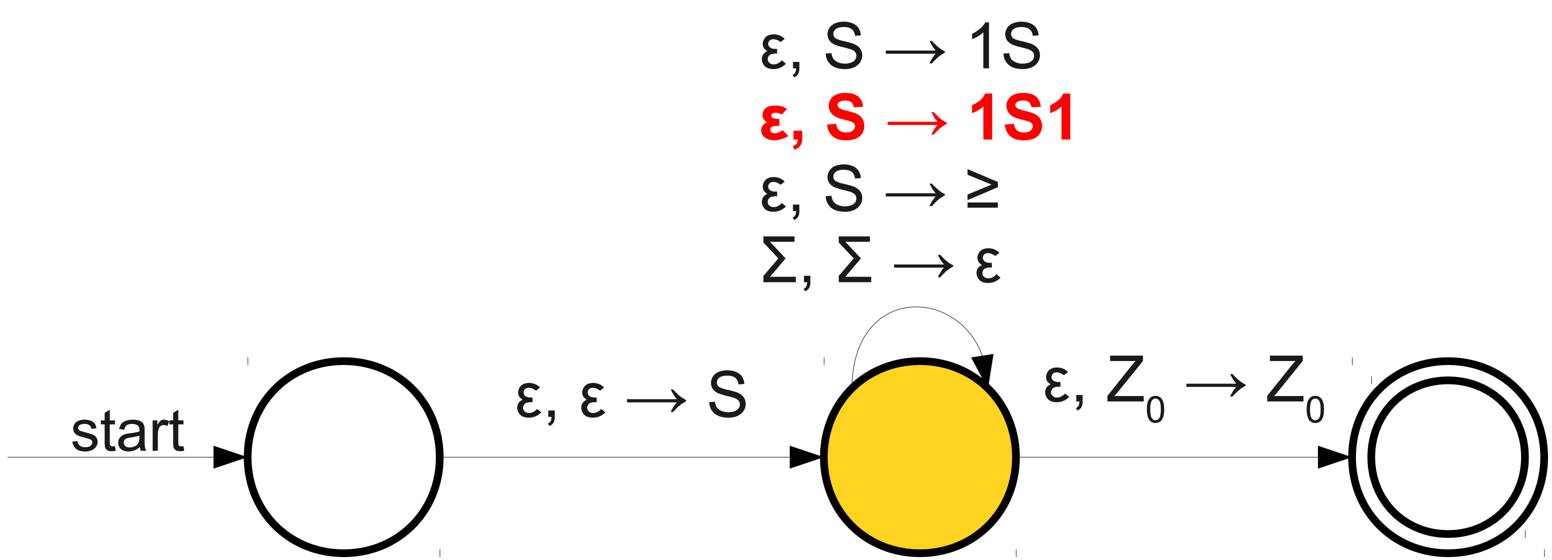
From CFGs to PDAs

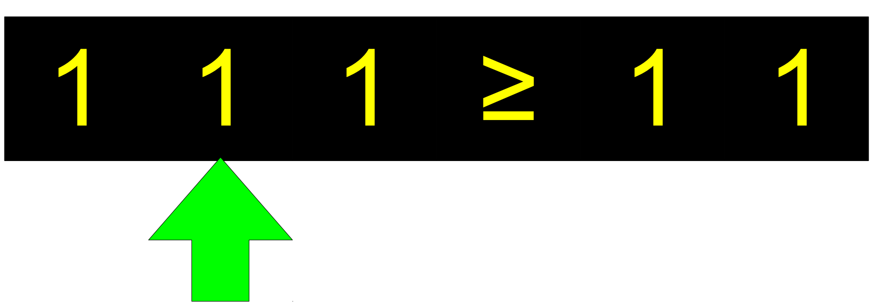
S → 1S1
S → 1S
S → ≥



From CFGs to PDAs

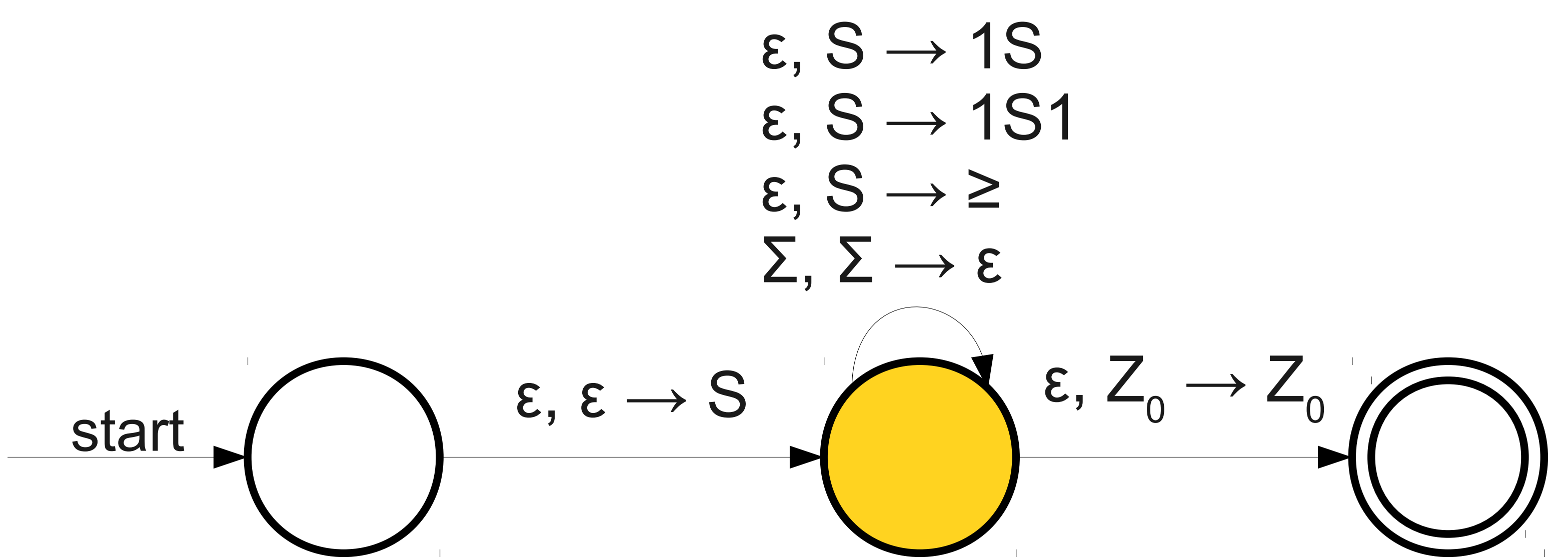
S → 1S1
S → 1S
S → ≥





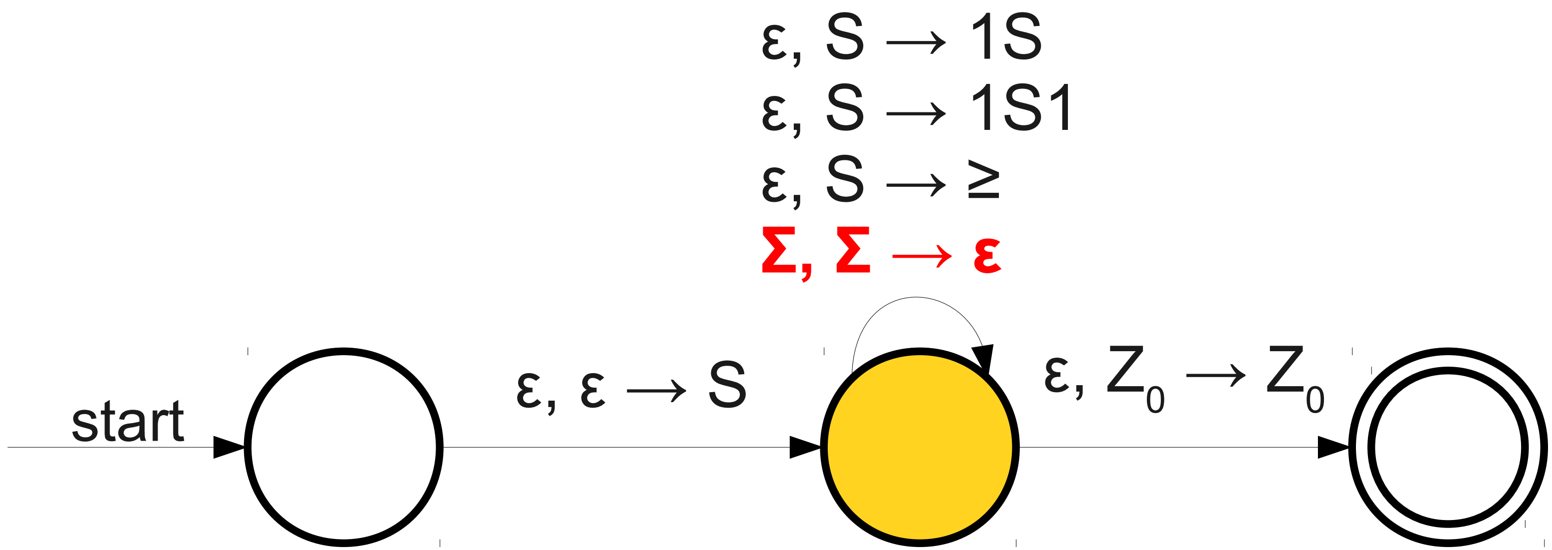
From CFGs to PDAs

S → 1S1
S → 1S
S → ≥



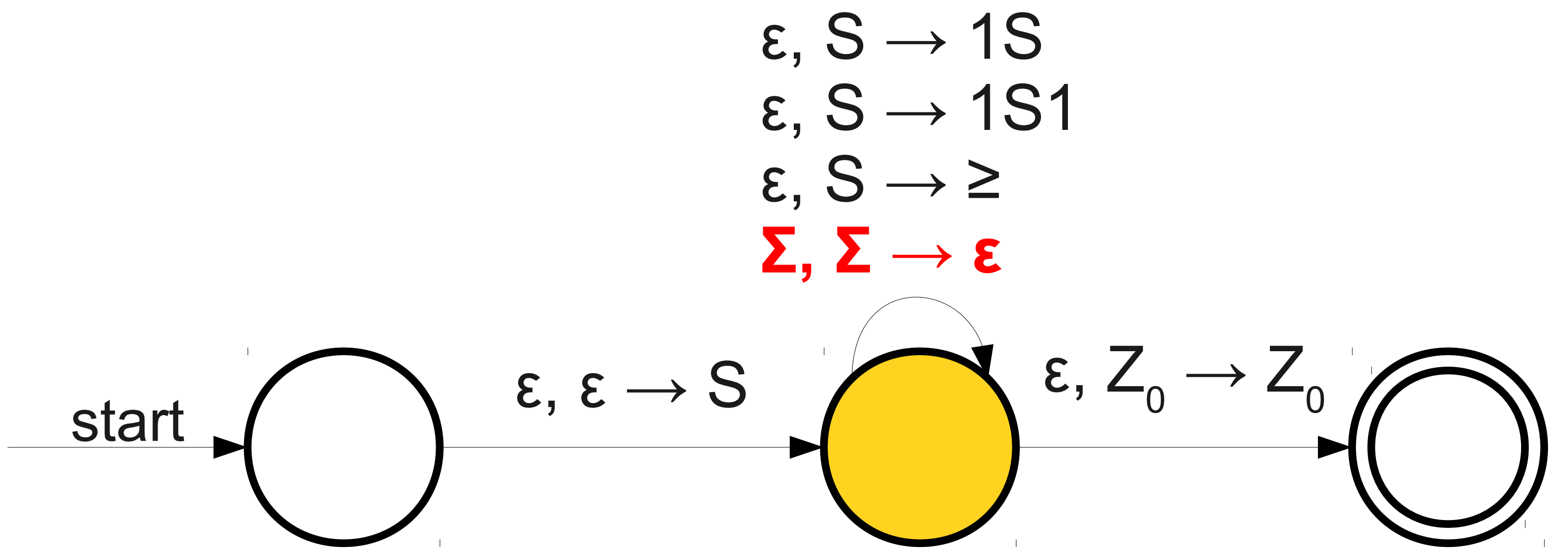
From CFGs to PDAs

S → 1S1
S → 1S
S → ≥



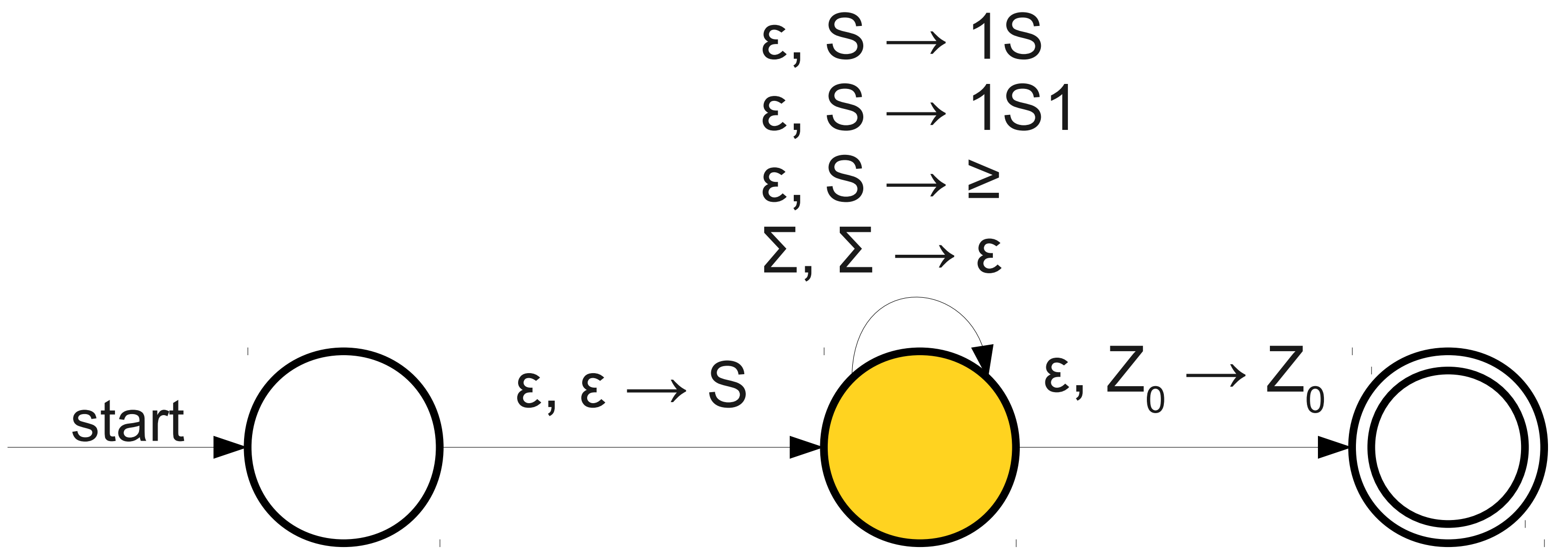
From CFGs to PDAs

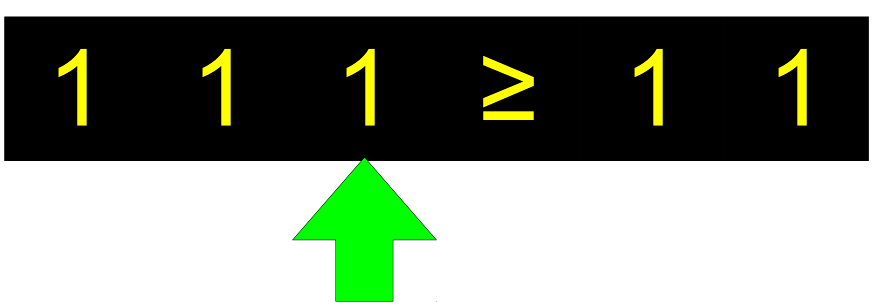
S → 1S1
S → 1S
S → ≥



From CFGs to PDAs

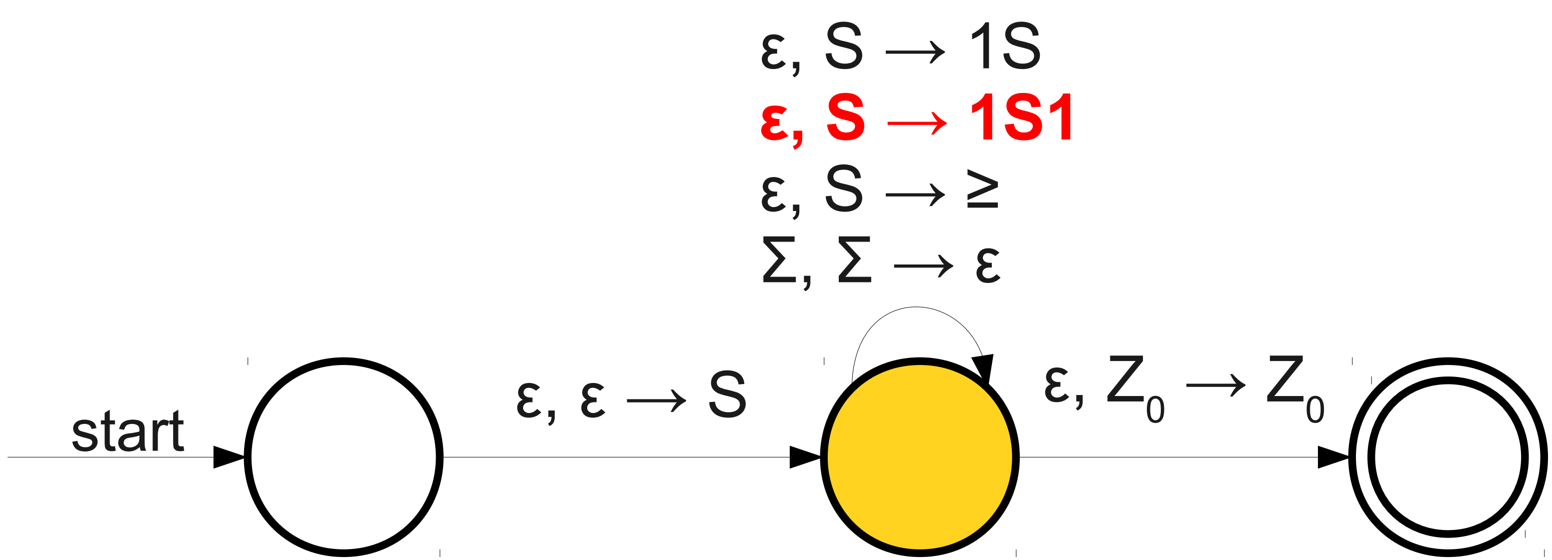
S → **1S1**
S → **1S**
S → **≥**

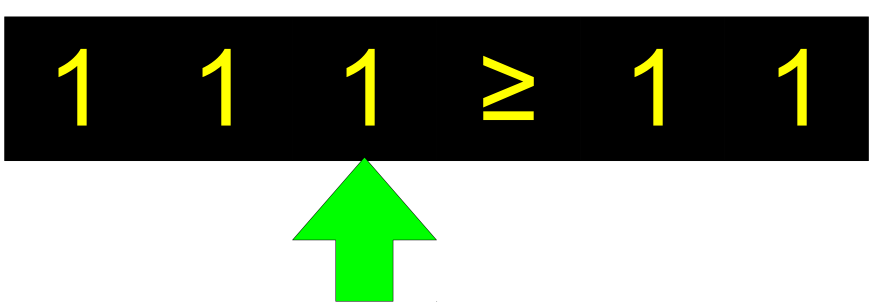




From CFGs to PDAs

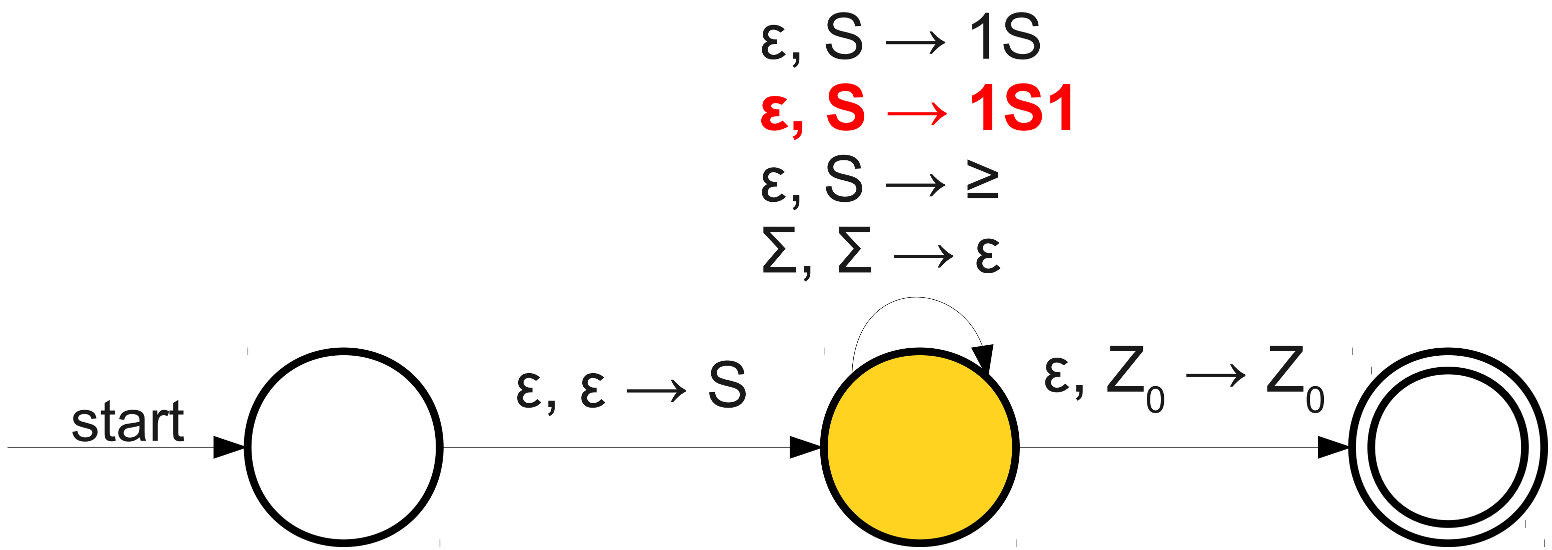
S → 1S1
S → 1S
S → ≥





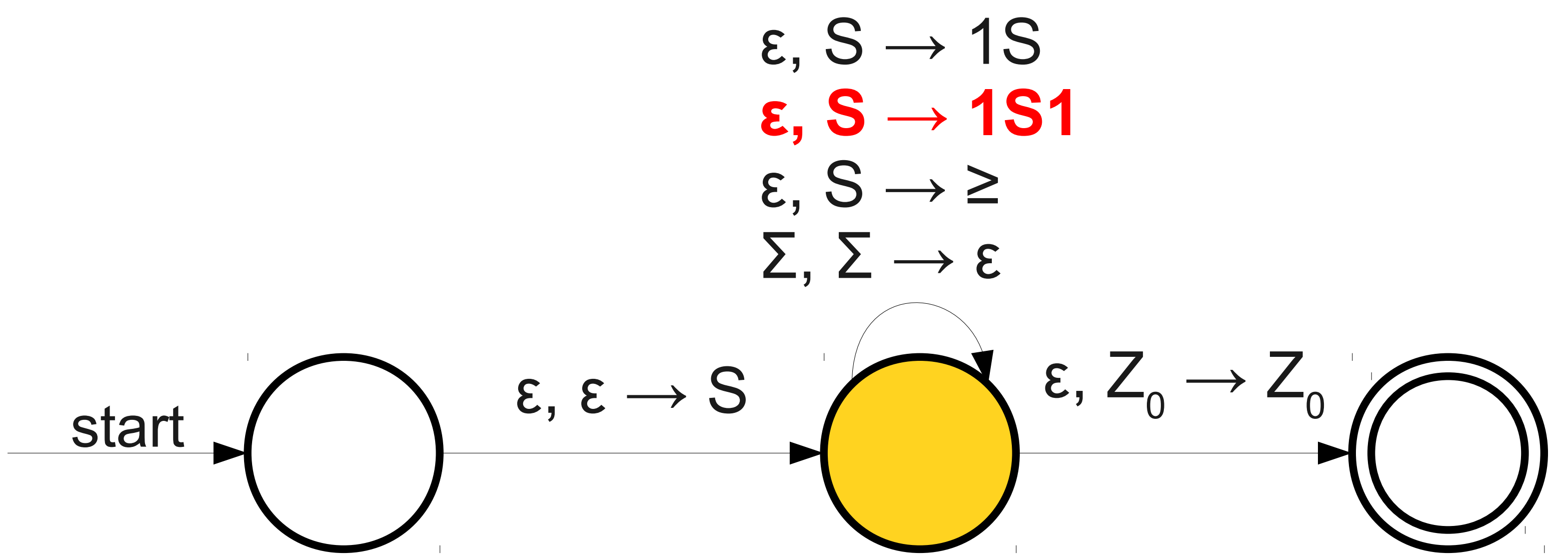
From CFGs to PDAs

$S \rightarrow 1S1$
$S \rightarrow 1S$
$S \rightarrow$ ≥



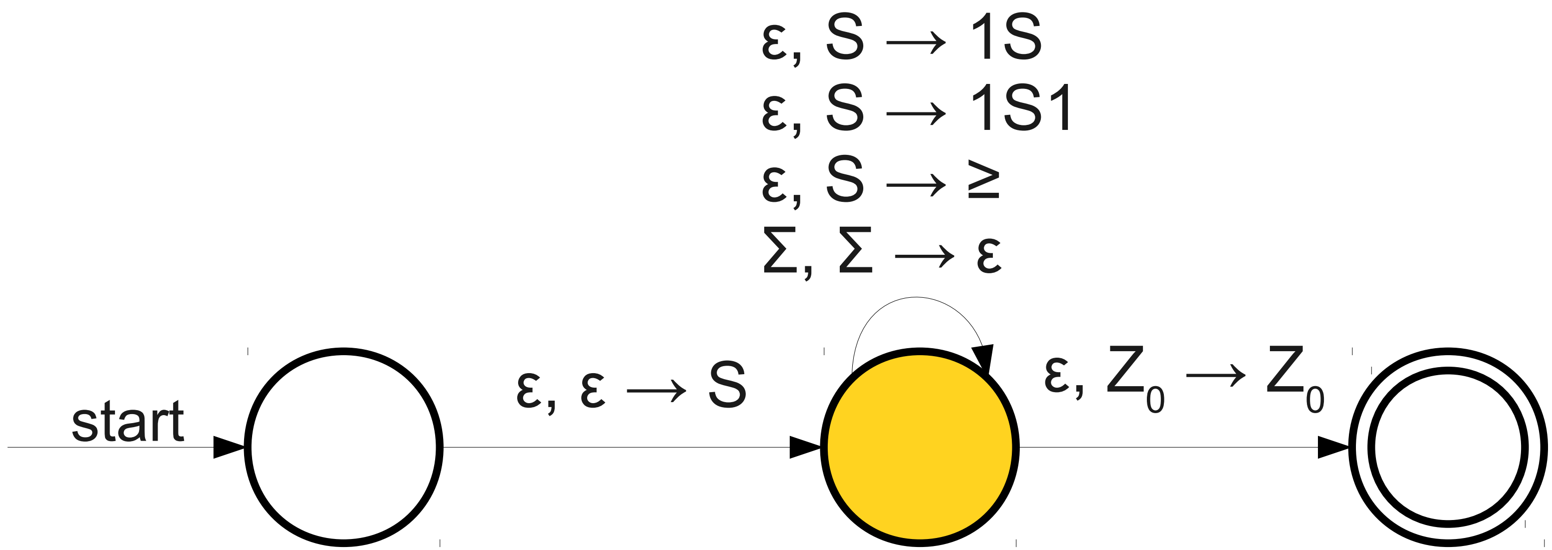
From CFGs to PDAs

S → **1S1**
S → **1S**
S → **≥**



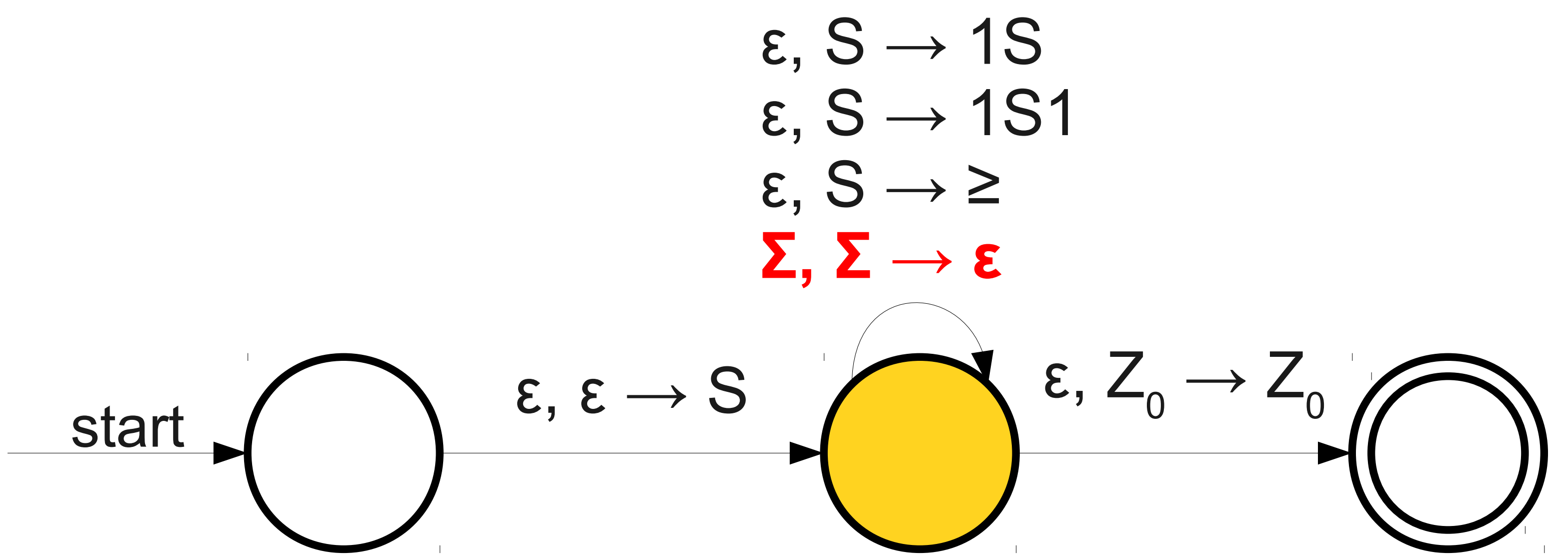
From CFGs to PDAs

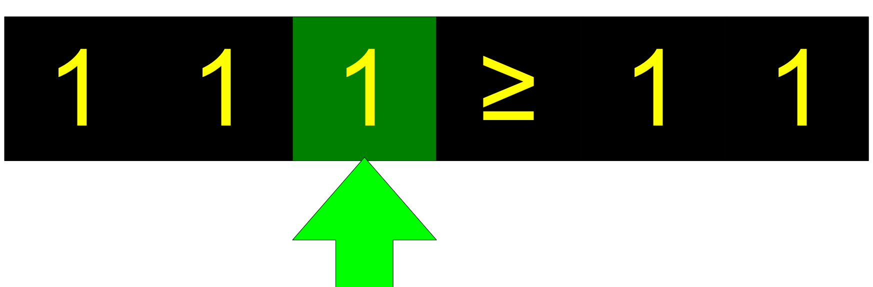
S → 1S1
S → 1S
S → ≥



From CFGs to PDAs

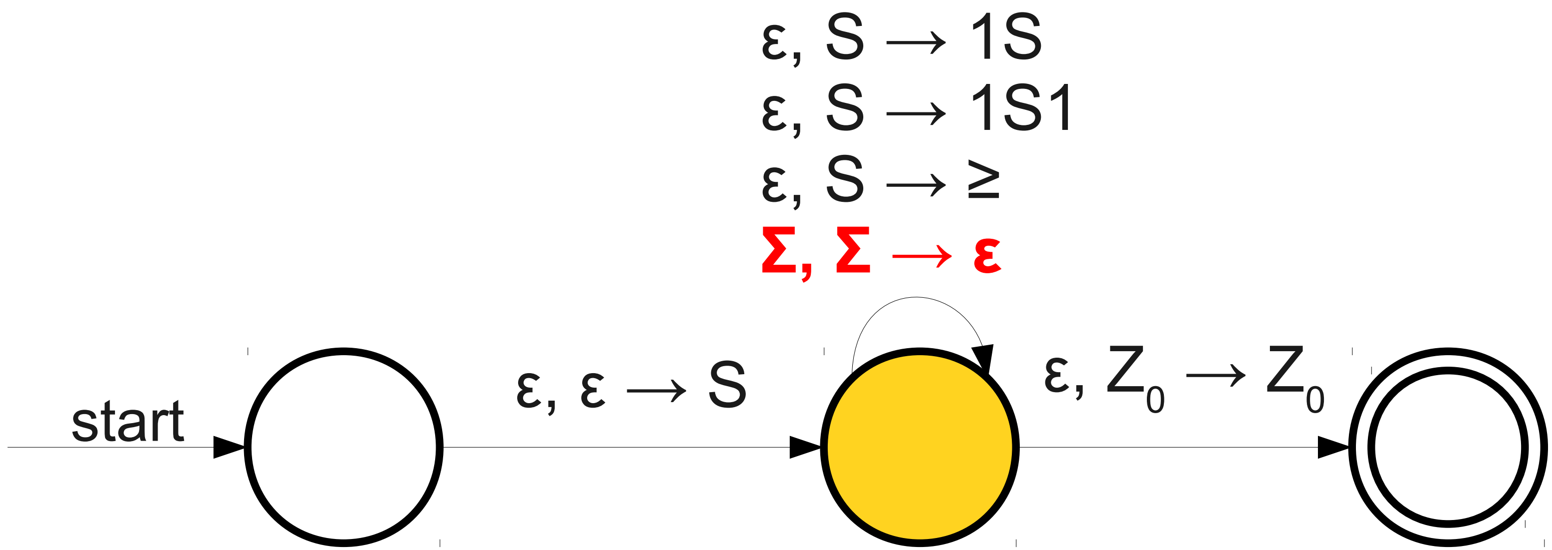
S → **1S1**
S → **1S**
S → **≥**





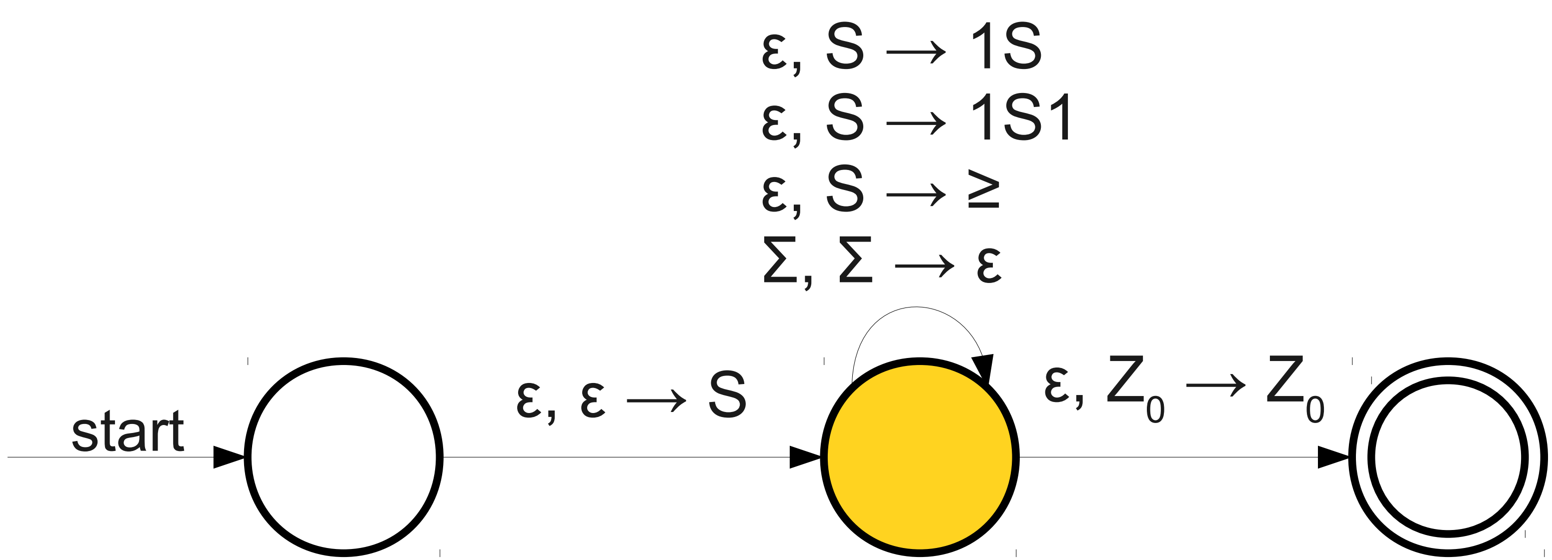
From CFGs to PDAs

$S \to 1S1$
$S \to 1S$
$S \to$ ≥



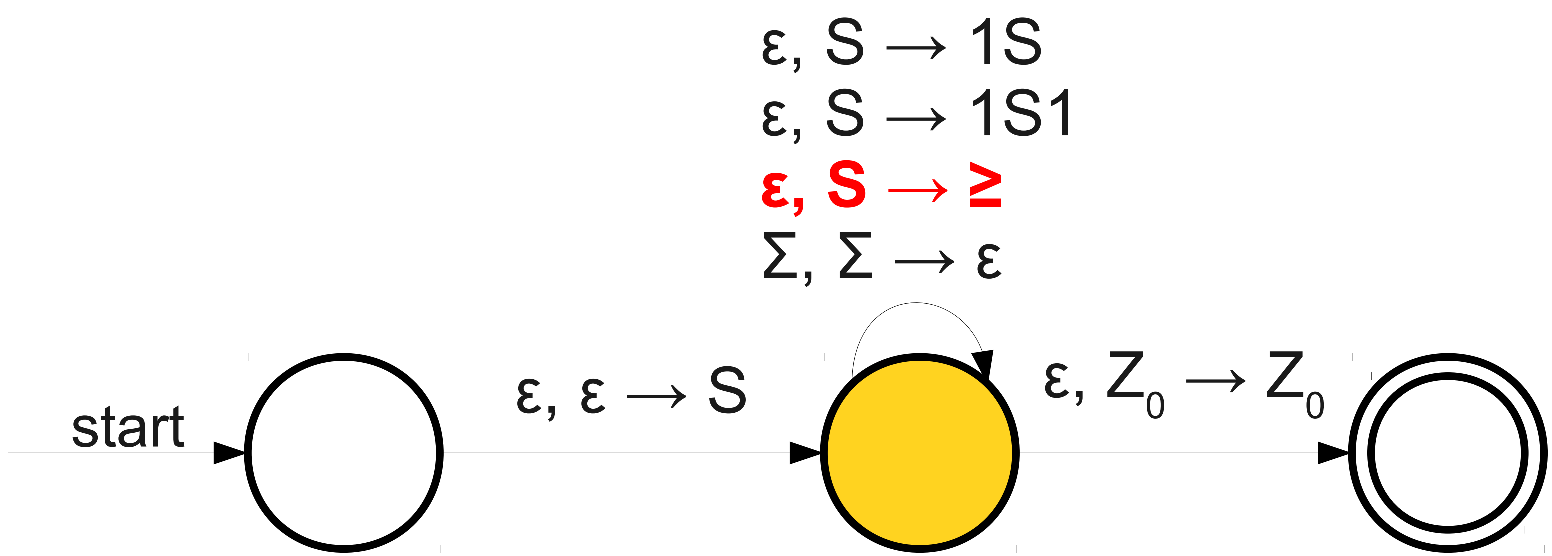
From CFGs to PDAs

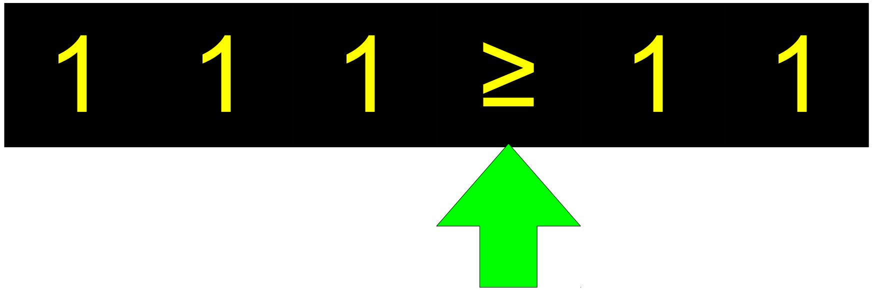
S → **1S1**
S → **1S**
S → **≥**

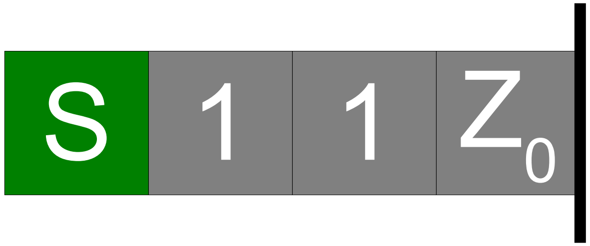


From CFGs to PDAs

S → 1S1
S → 1S
S → ≥

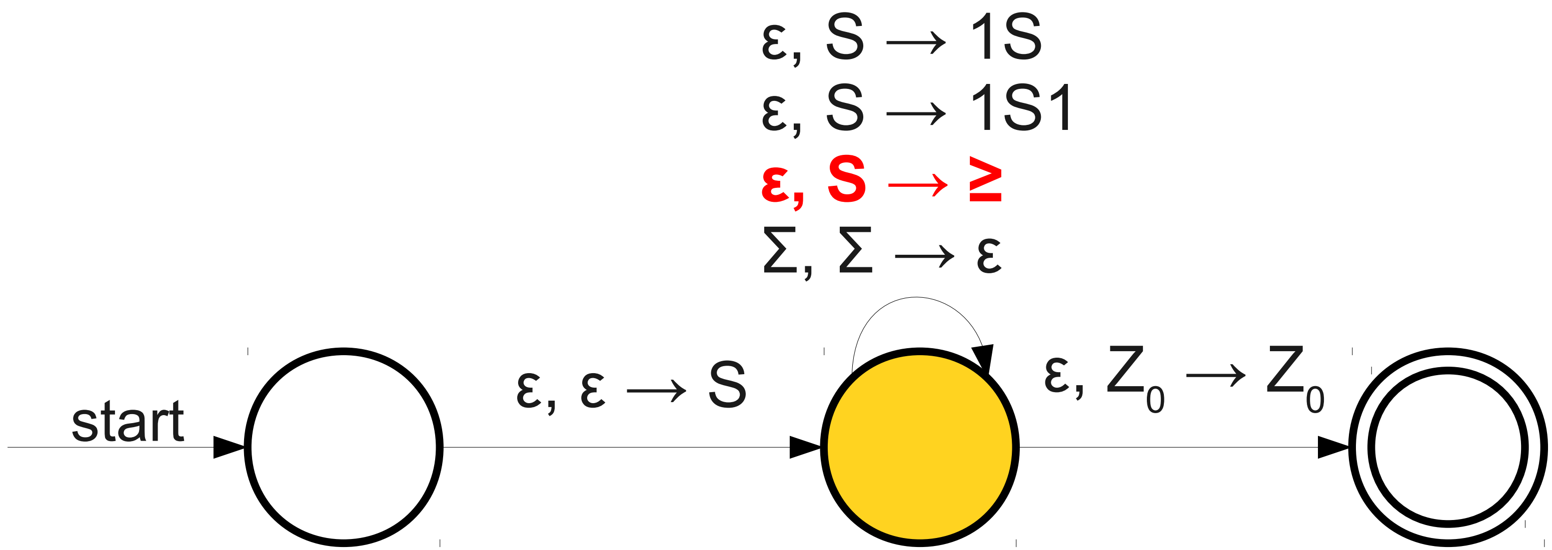


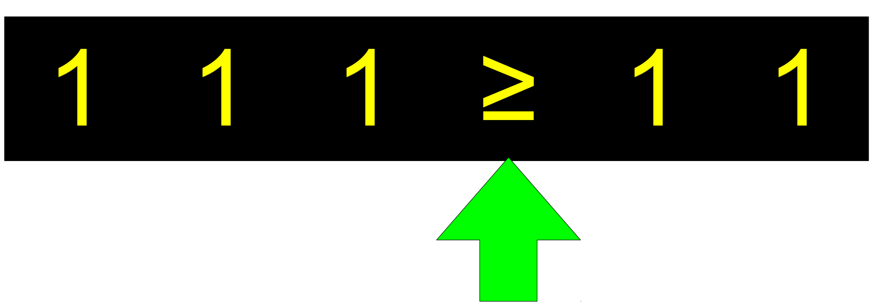




From CFGs to PDAs

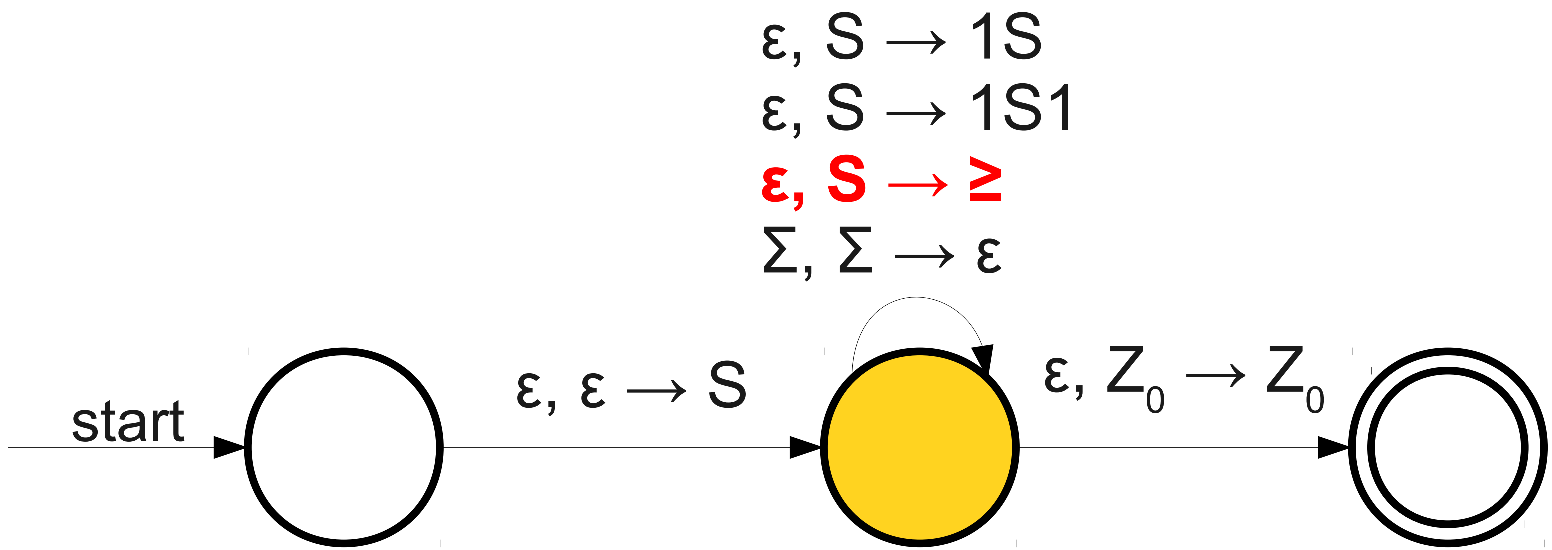
S → 1S1
S → 1S
S → ≥

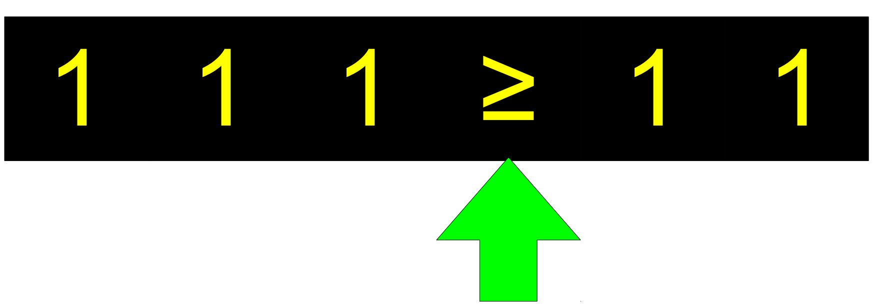




From CFGs to PDAs

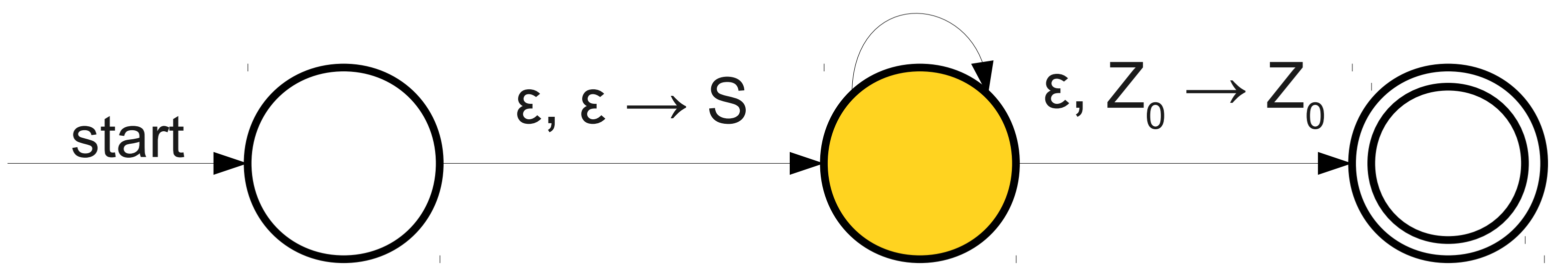
$S \rightarrow 1S1$
$S \rightarrow 1S$
$S \rightarrow$ ≥

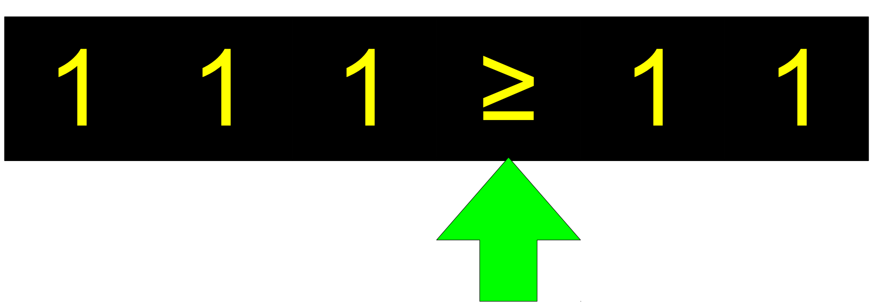




From CFGs to PDAs

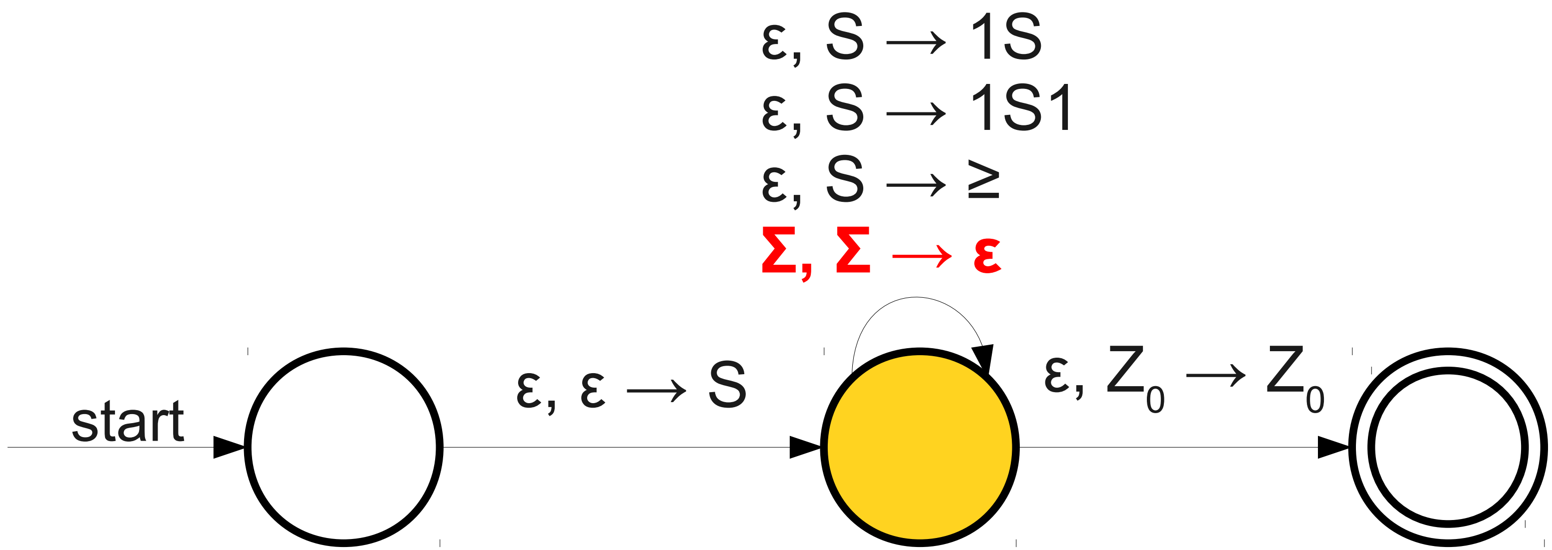
S → 1S1
S → 1S
S → ≥

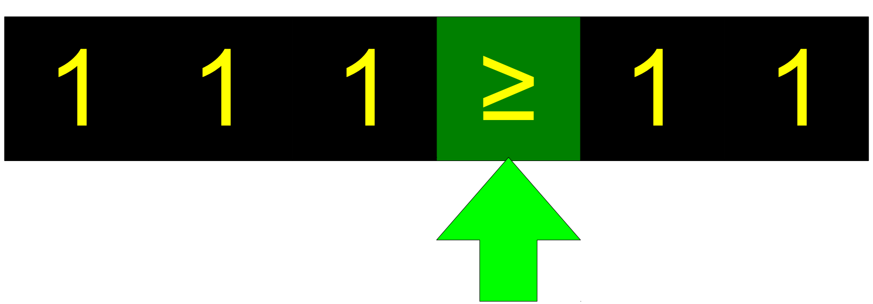




From CFGs to PDAs

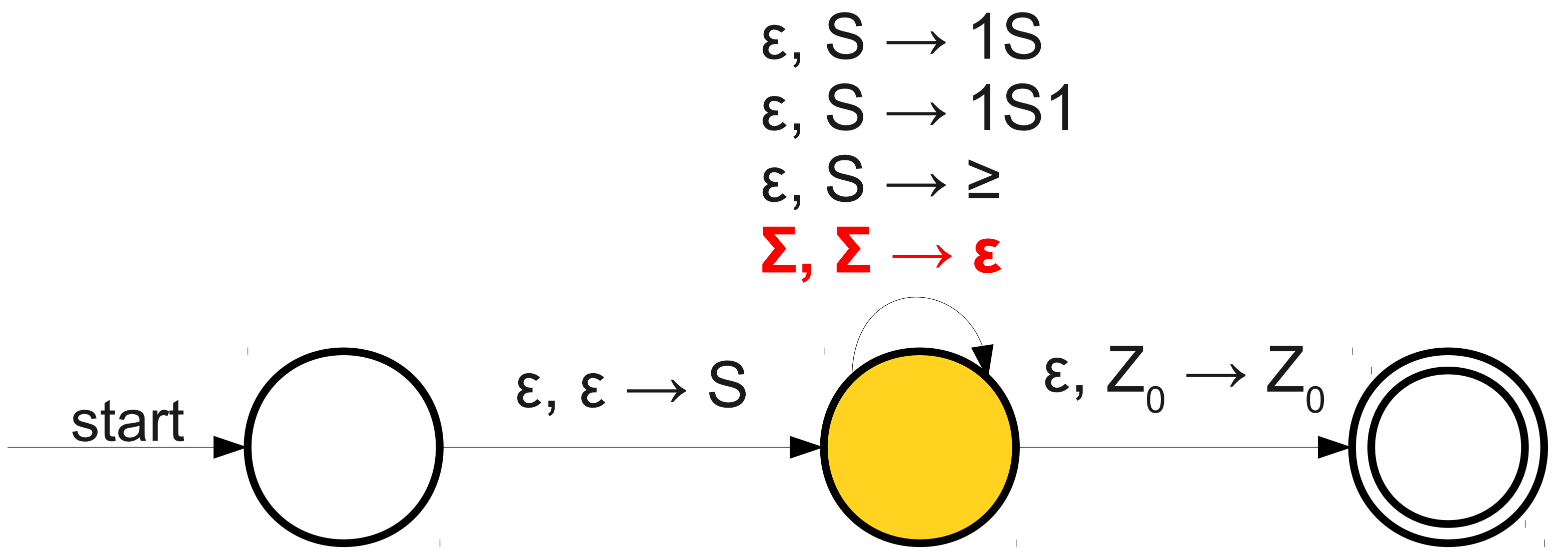
S → 1S1
S → 1S
S → ≥

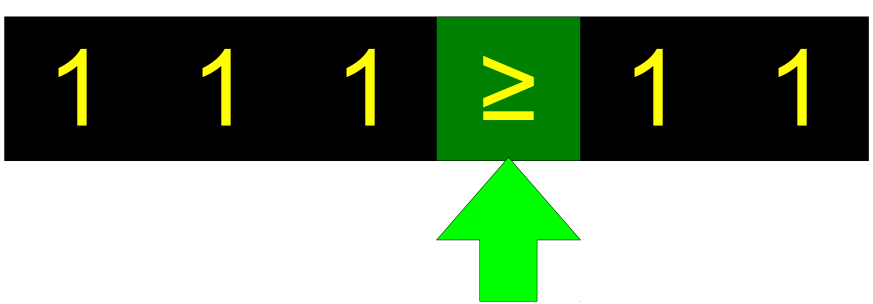




From CFGs to PDAs

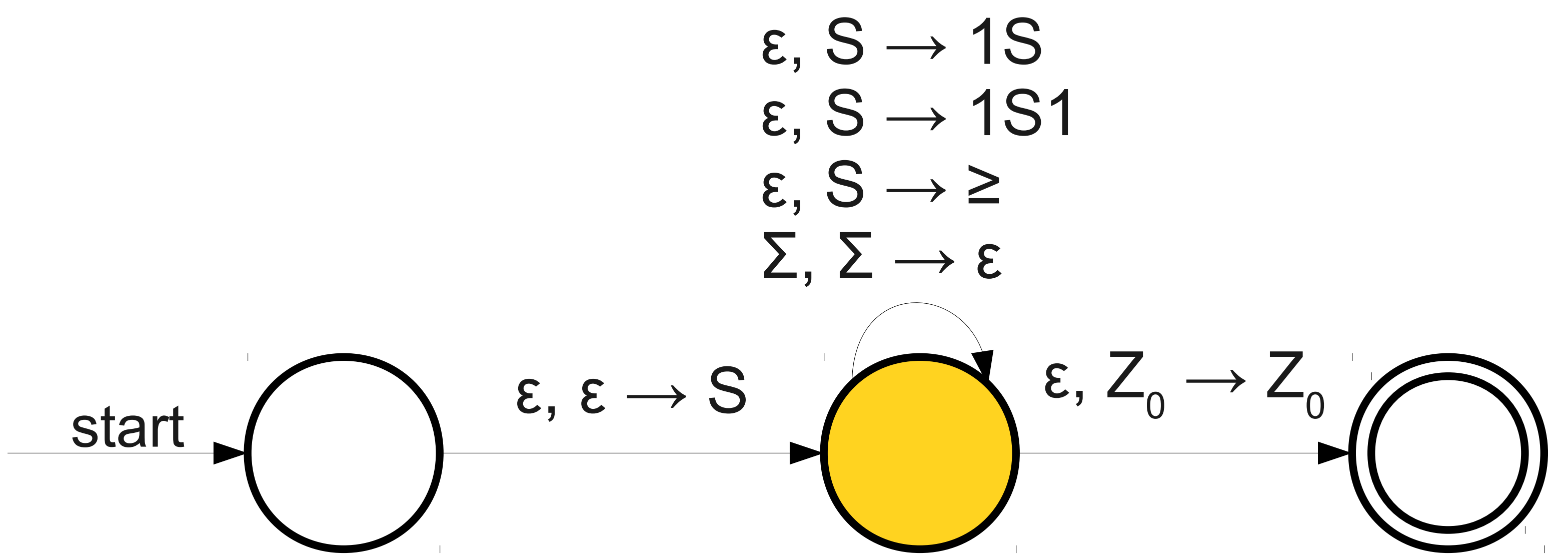
S → 1S1
S → 1S
S → ≥

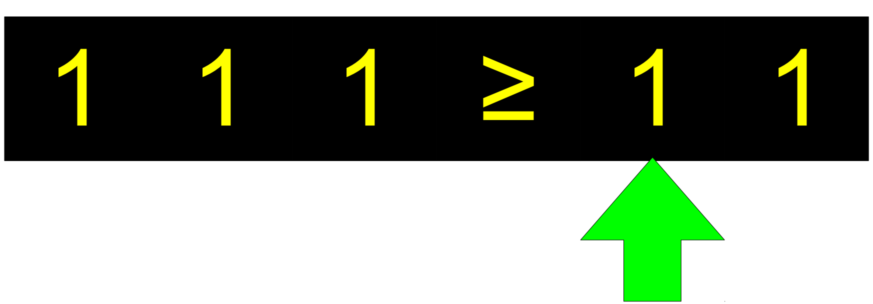




From CFGs to PDAs

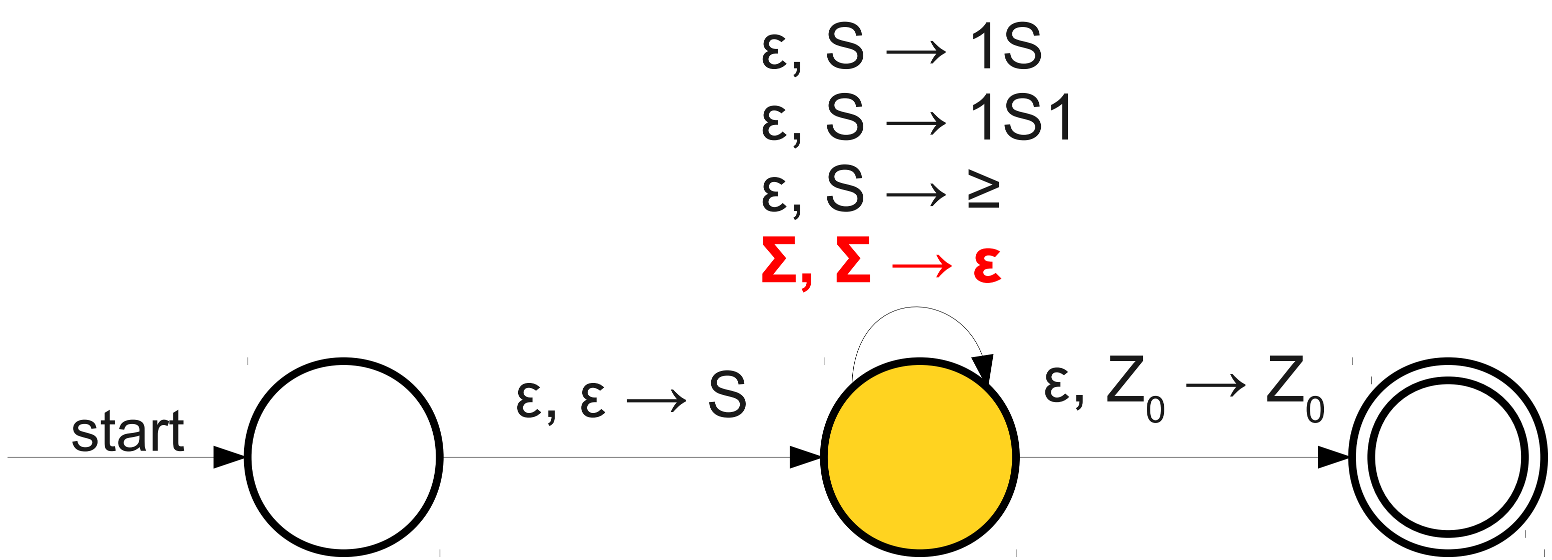
S → 1S1
S → 1S
S → ≥

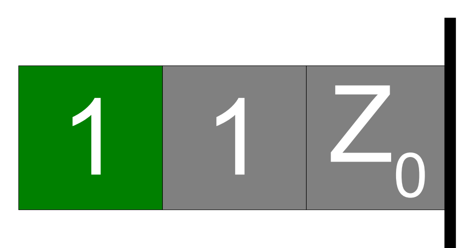




From CFGs to PDAs

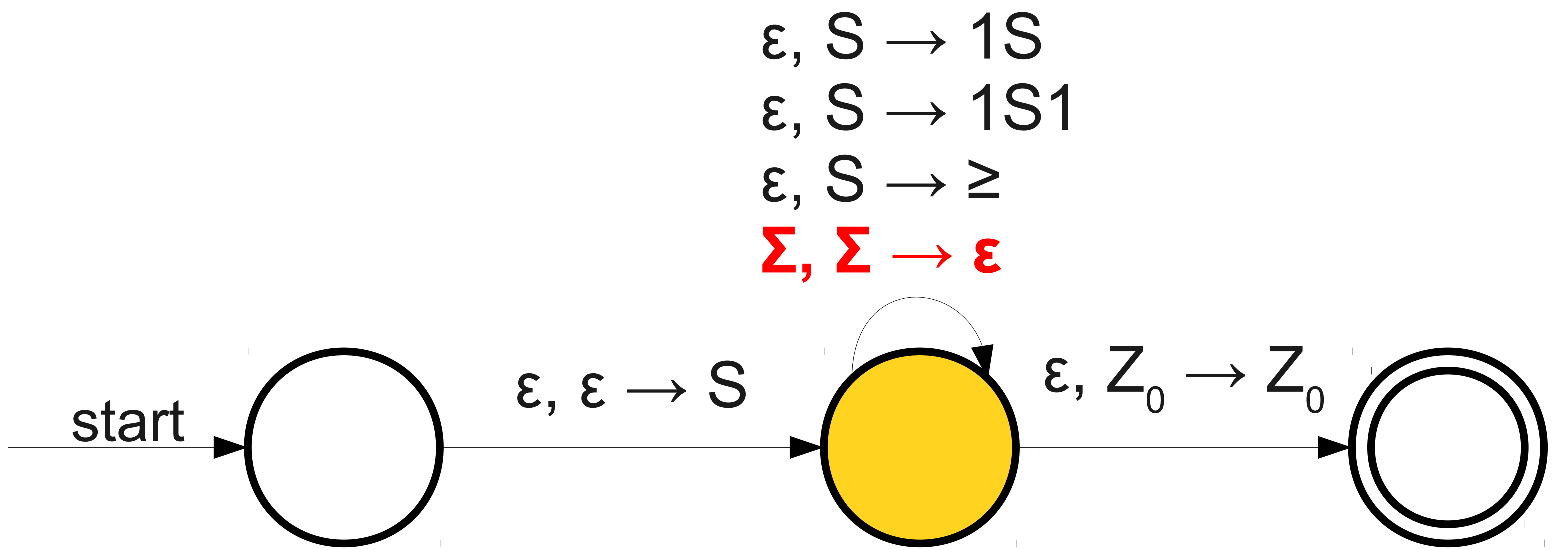
S → 1S1
S → 1S
S → ≥





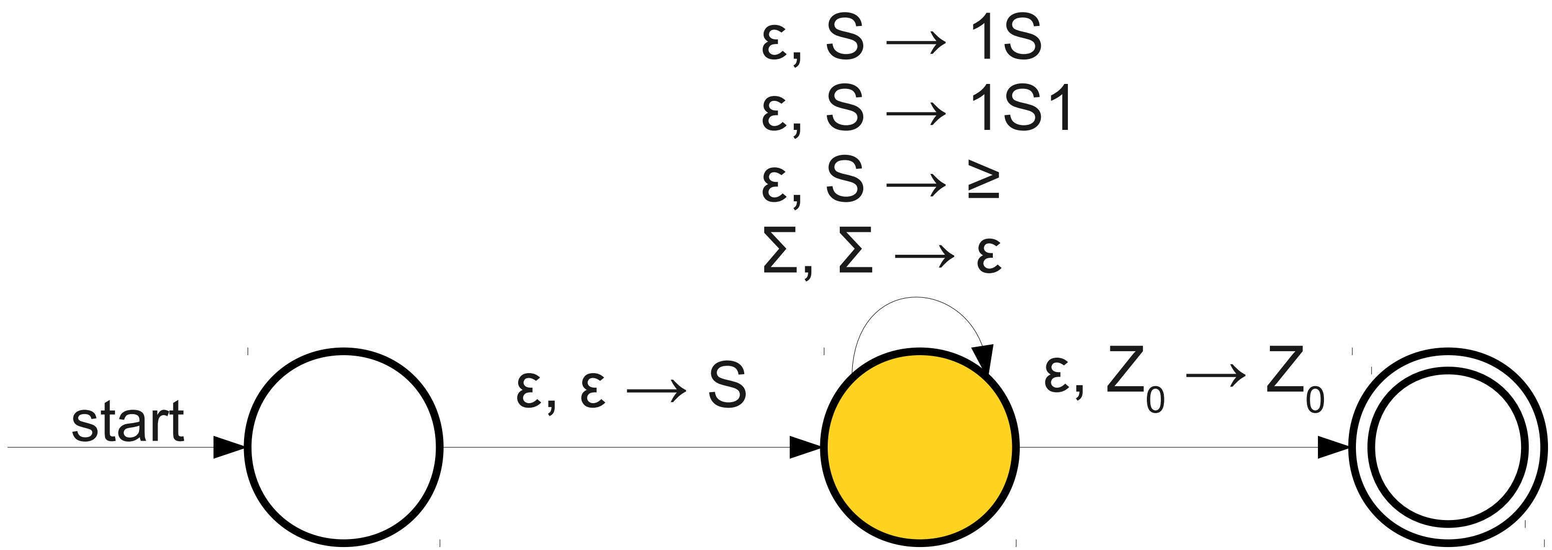
From CFGs to PDAs

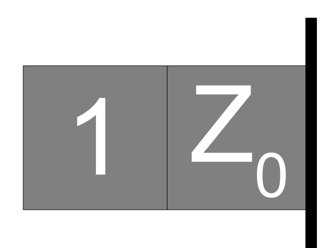
S → 1S1
S → 1S
S → ≥



From CFGs to PDAs

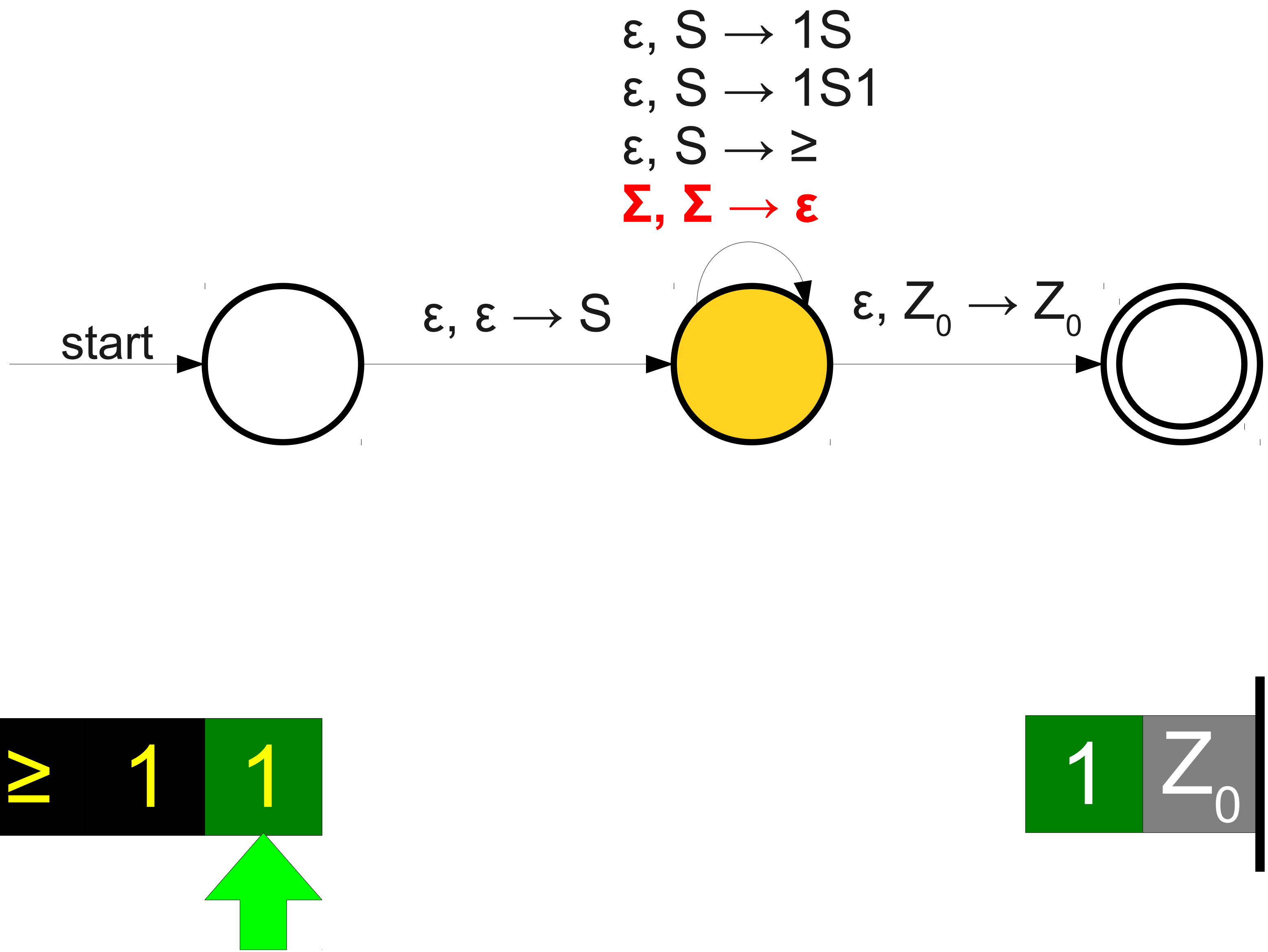
S → 1S1
S → 1S
S → ≥





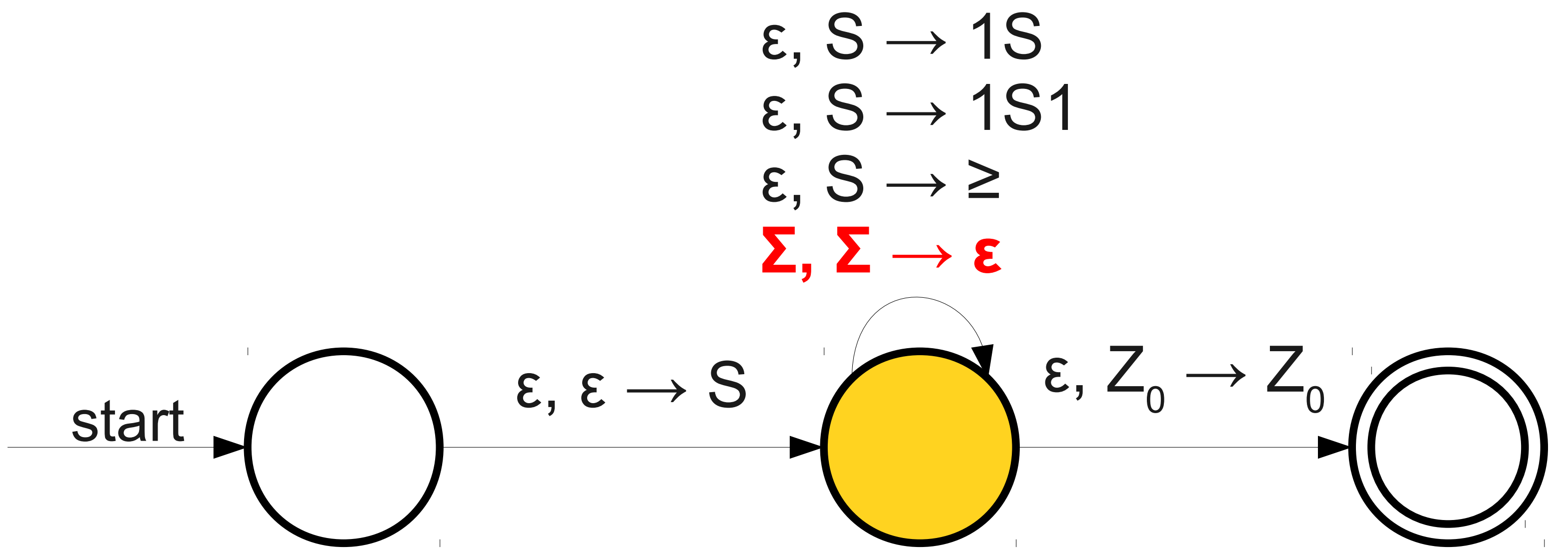
From CFGs to PDAs

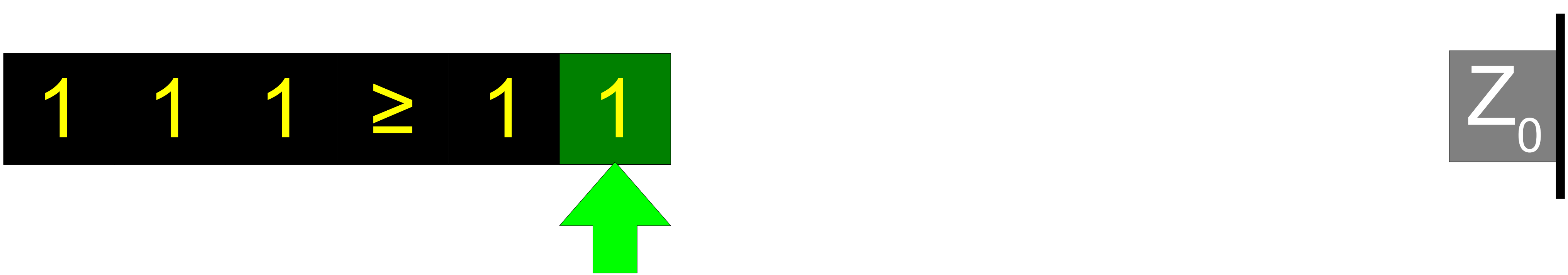
S → 1S1
S → 1S
S → ≥



From CFGs to PDAs

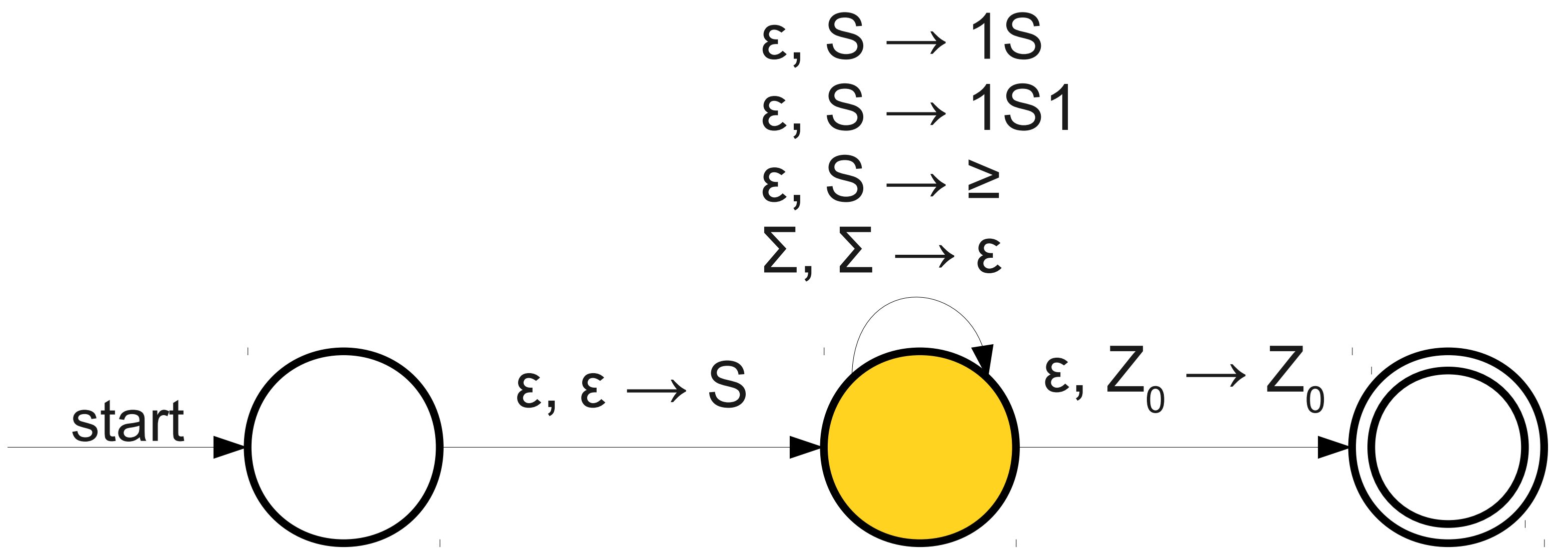
S → 1S1
S → 1S
S → ≥

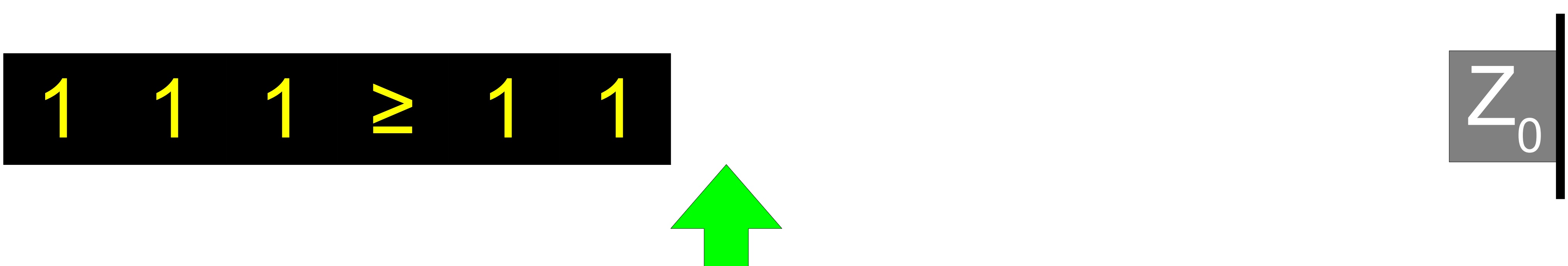




From CFGs to PDAs

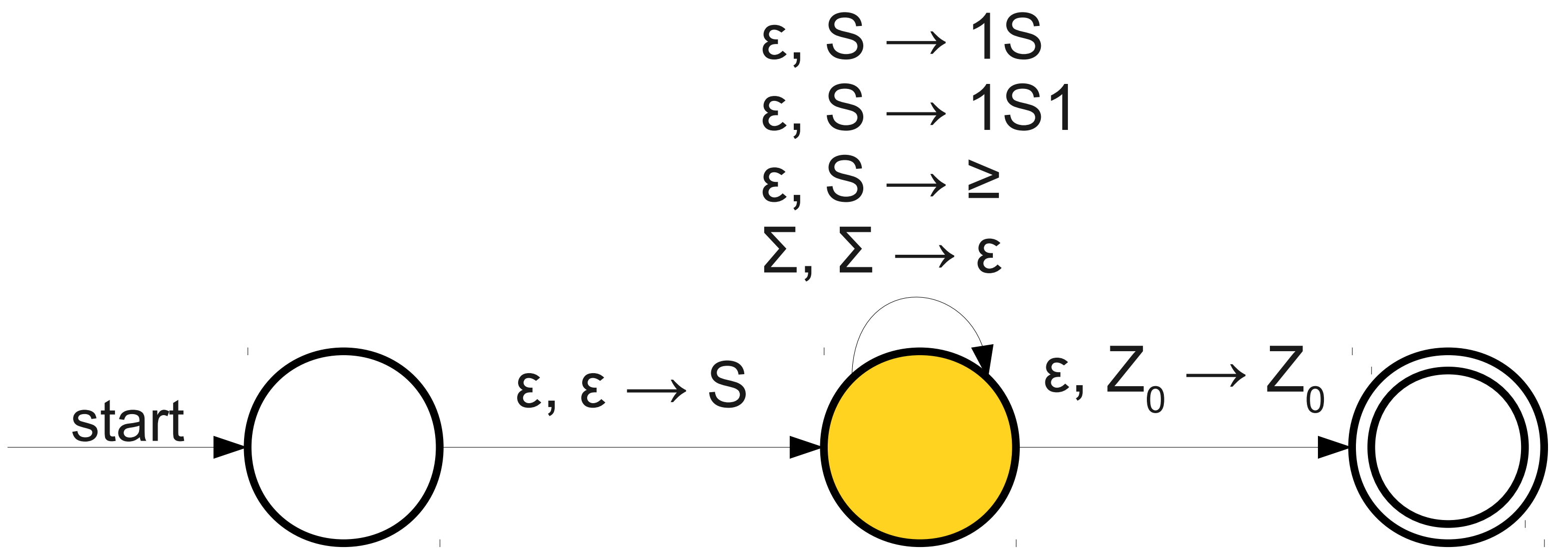
S → 1S1
S → 1S
S → ≥



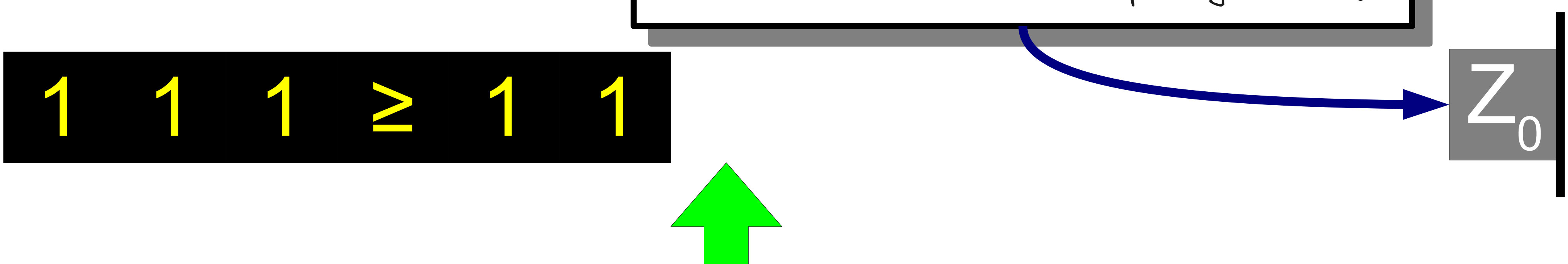


From CFGs to PDAs

$S \rightarrow 1S1$
$S \rightarrow 1S$
$S \rightarrow \geq$

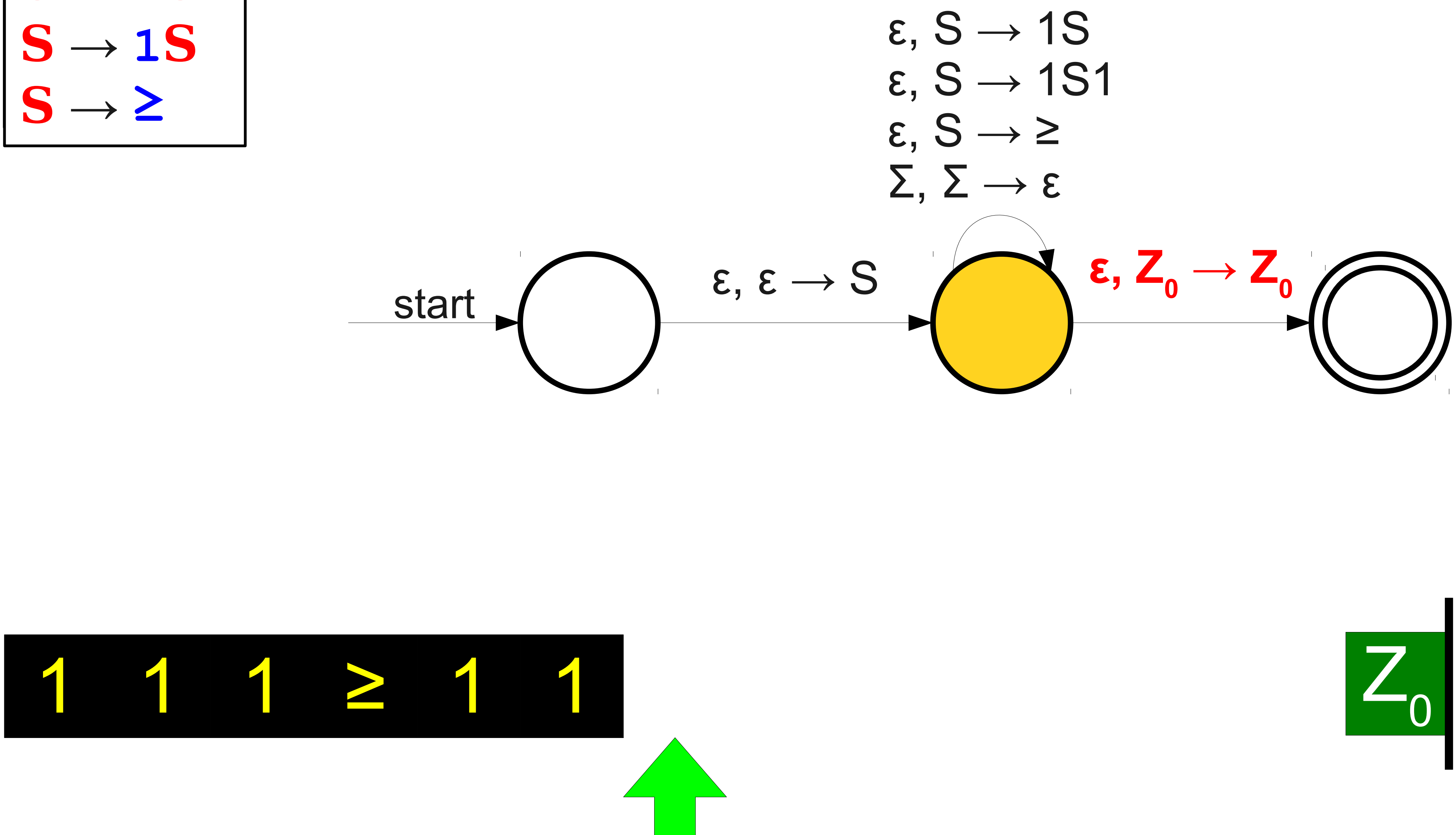


At this point we've completely matched the string, so it's time to transition to the accepting state.



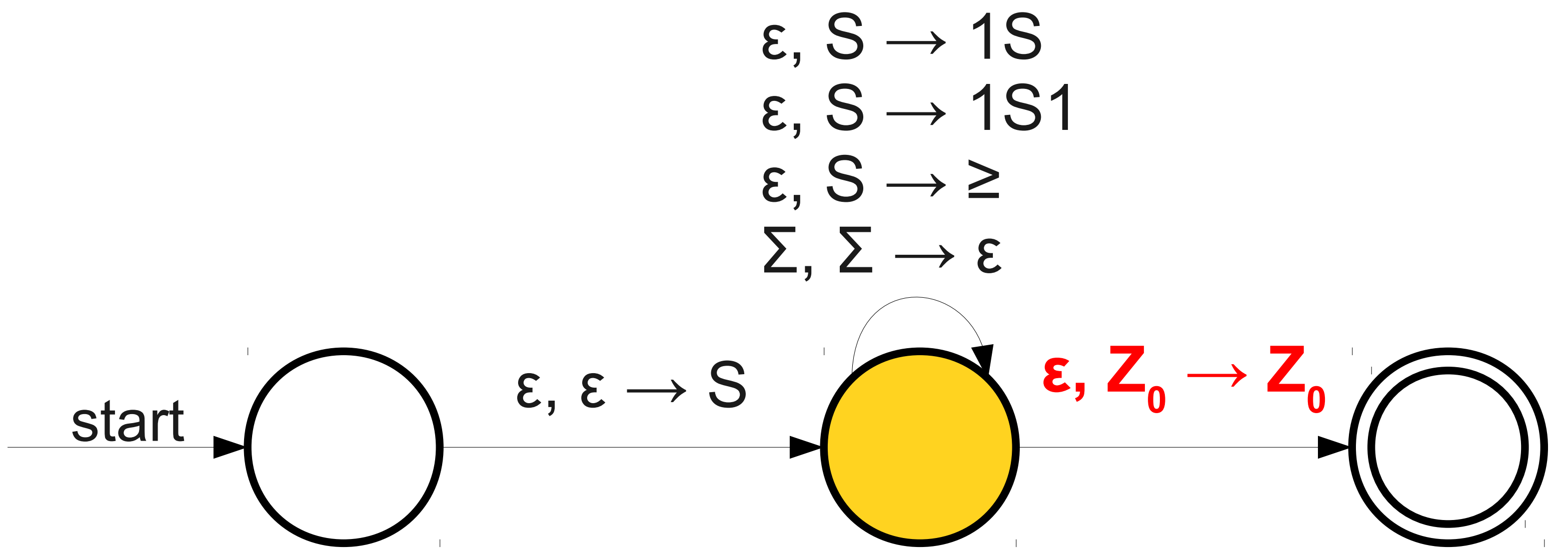
From CFGs to PDAs

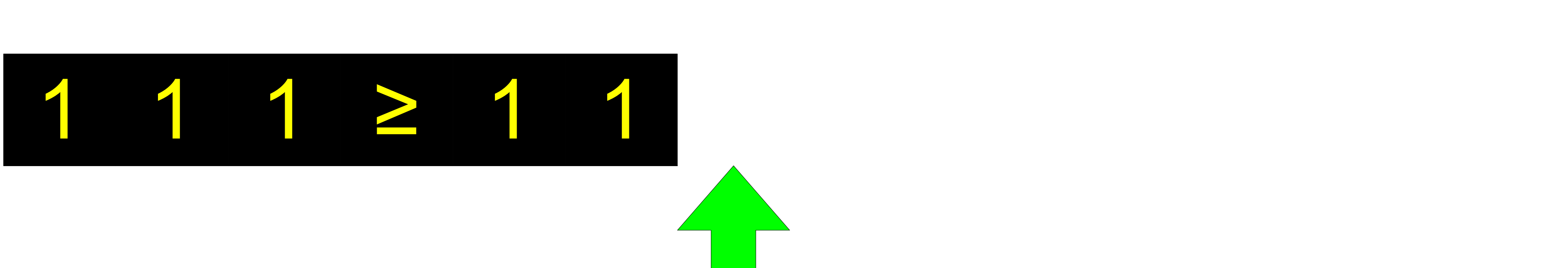
$S \rightarrow 1S1$
$S \rightarrow 1S$
$S \rightarrow ≥$



From CFGs to PDAs

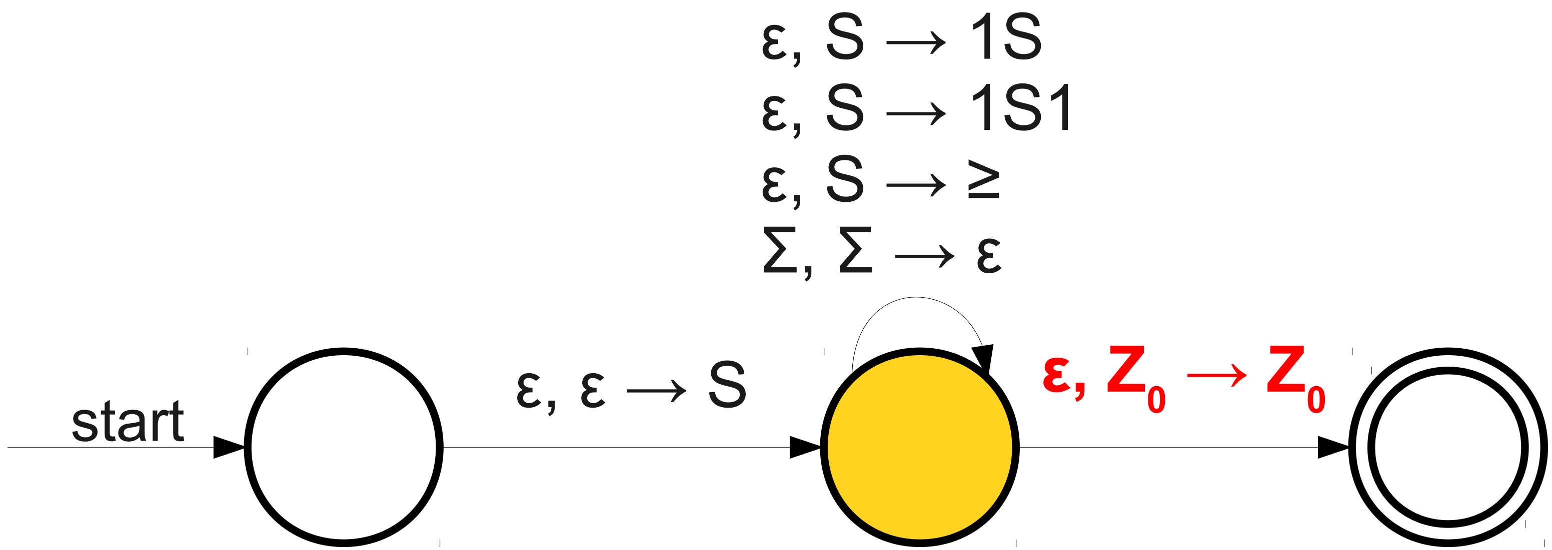
S → 1S1
S → 1S
S → ≥

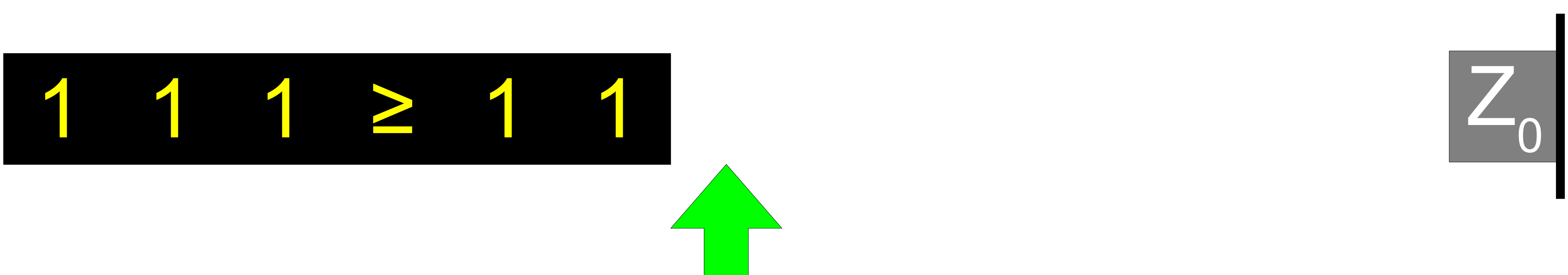




From CFGs to PDAs

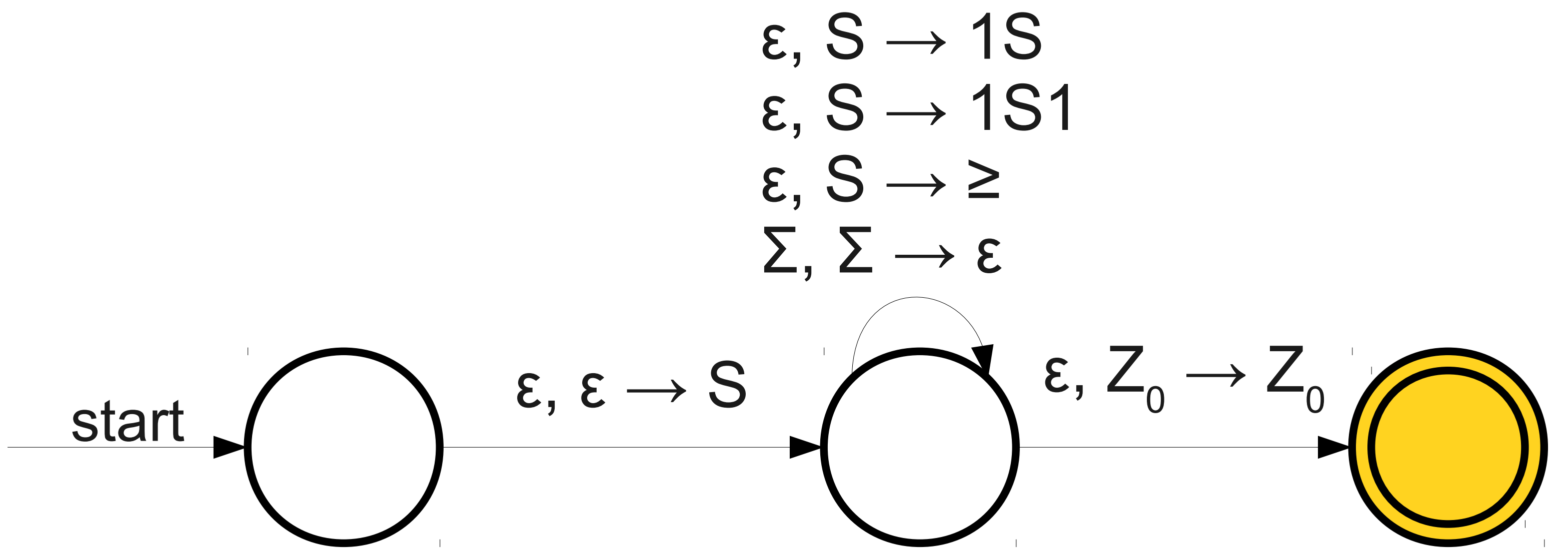
$S \to 1S1$
$S \to 1S$
$S \to \geq$

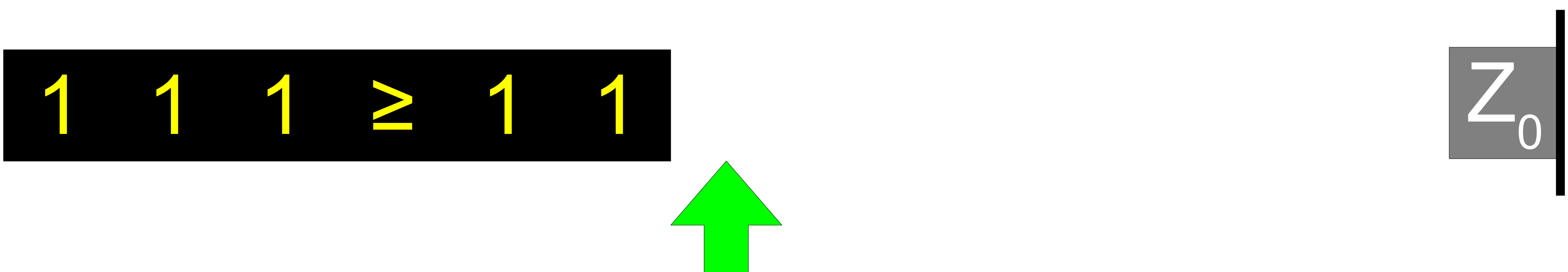




From CFGs to PDAs

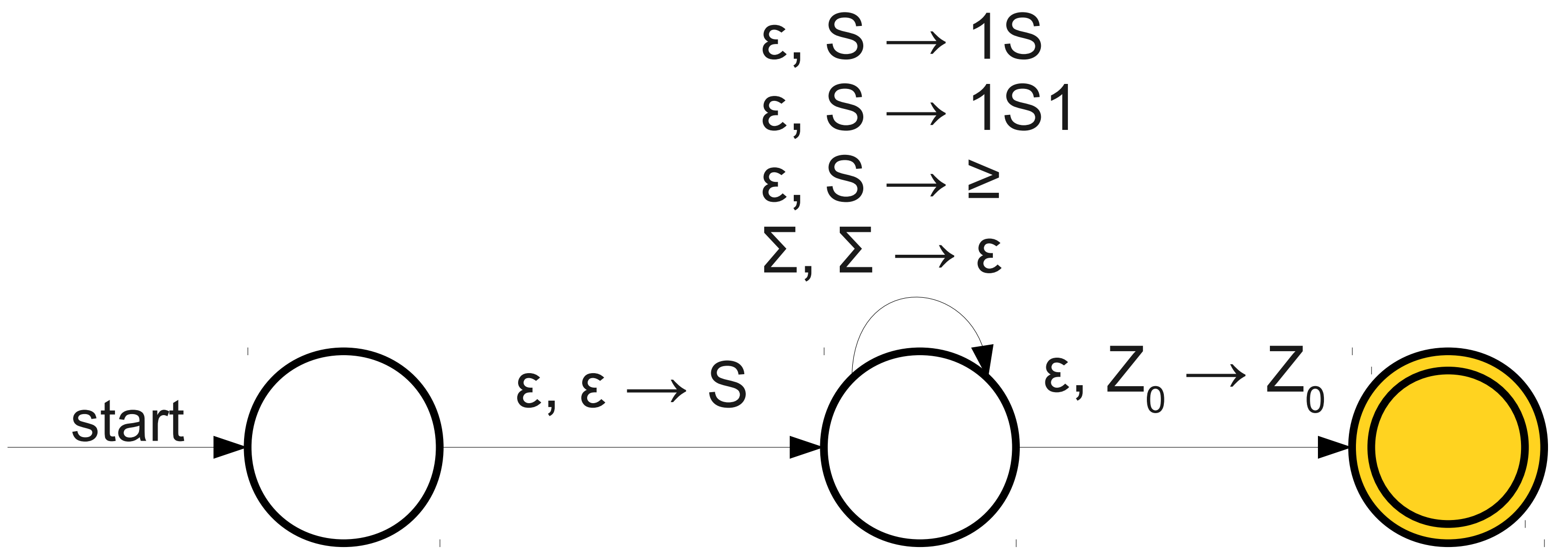
S → 1S1
S → 1S
S → ≥





From CFGs to PDAs

S → 1S1
S → 1S
S → ≥

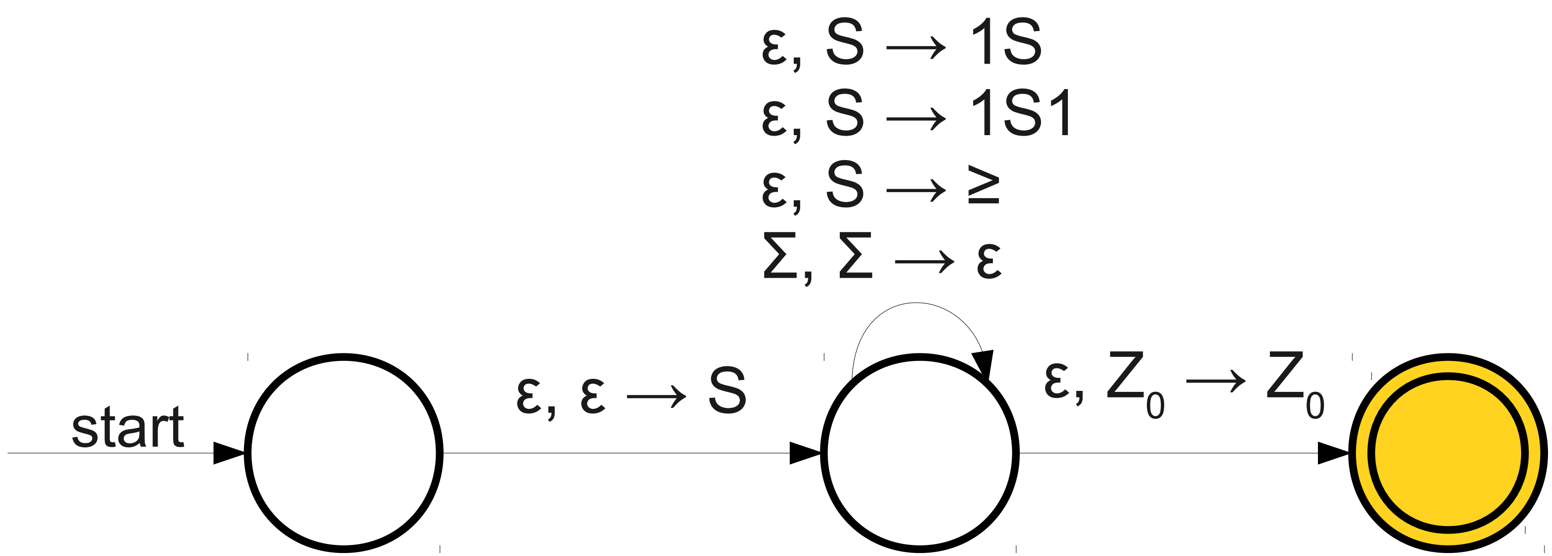


1 1 1 ≥ 1 1

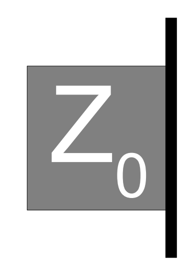
Z_0

From CFGs to PDAs

$S \rightarrow 1S1$
$S \rightarrow 1S$
$S \rightarrow \geq$



1 1 1 ≥ 1 1



From CFGs to PDAs

- Make three states: **start**, **parsing**, and **accepting**.
- There is a transition ε, ε → **S** from **start** to **parsing**.
  - Corresponds to starting off with the start symbol S.
- There is a transition ε, **A** → ***ω*** from **parsing** to itself for each production **A** → ***ω***.
  - Corresponds to predicting which production to use.
- There is a transition Σ, Σ → ε from **parsing** to itself.
  - Corresponds to matching a character of the input.
- There is a transition ε, Z_0 → Z_0 from **parsing** to **accepting**.
  - Corresponds to completely matching the input.

From CFGs to PDAs

- The PDA constructed this way is called a **predict/match parser**.
- Each step either **predicts** which production to use or **matches** some symbol of the input.

From PDAs to CFGs

- The other direction of the proof (converting a PDA to a CFG) is much harder.
- Intuitively, create a CFG representing paths between states in the PDA.
- Lots of tricky details, but a marvelous proof.
  - It's just too large to fit into the margins of this slide.
- Read Sipser for more details.

Regular and Context-Free Languages

Theorem: Any regular language is context-free.

Regular and Context-Free Languages

Theorem: Any regular language is context-free.

Proof Sketch: Let L be any regular language and consider a DFA D for L.

Regular and Context-Free Languages

Theorem: Any regular language is context-free.

Proof Sketch: Let *L* be any regular language and consider a DFA *D* for *L*. Then we can convert *D* into a PDA for *L* by converting any transition on a symbol **a** into a transition **a**, ε → ε that ignores the stack. This new PDA accepts *L*, so *L* is context-free.

Regular and Context-Free Languages

Theorem: Any regular language is context-free.

Proof Sketch: Let *L* be any regular language and consider a DFA *D* for *L*. Then we can convert *D* into a PDA for *L* by converting any transition on a symbol **`a`** into a transition **`a`**, ε → ε that ignores the stack. This new PDA accepts *L*, so *L* is context-free. ■-*ish*

Refining the Context-Free Languages

NPDAs and DPDAs

- With finite automata, we considered both deterministic (DFAs) and nondeterministic (NFAs) automata.
- So far, we've only seen nondeterministic PDAs (or **NPDAs**).
- What about deterministic PDAs (**DPDAs**)?

DPDAs

- A **deterministic pushdown automaton** is a PDA with the extra property that

  For each state in the PDA, and for any combination of a current input symbol and a current stack symbol, there is **at most** one transition defined.

- In other words, there is ***at most*** one legal sequence of transitions that can be followed for any input.
- This does ***not*** preclude ε-transitions, as long as there is never a conflict between following the ε-transition or some other transition.
- However, there can be ***at most*** one ε-transition that could be followed at any one time.
- This does ***not*** preclude the automaton "dying" from having no transitions defined; DPDAs can have undefined transitions.

DPDAs

A **deterministic pushdown automaton** is a PDA with the extra property that

For each state in the PDA, and for any combination of a current input symbol and a current stack symbol, there is **at most** one transition defined.

In other words, there is ***at most*** one legal sequence of transitions that can be followed for any input.

This does ***not*** preclude ε-transitions, as long as there is never a conflict between following the ε-transition or some other transition.

However, there can be ***at most*** one ε-transition that could be followed at any one time.

- This does ***not*** preclude the automaton "dying" from having no transitions defined; DPDAs can have undefined transitions.

DPDAs

A **deterministic pushdown automaton** is a PDA with the extra property that

For each state in the PDA, and for any combination of a current input symbol and a current stack symbol, ther

In other words, … of transitions that …

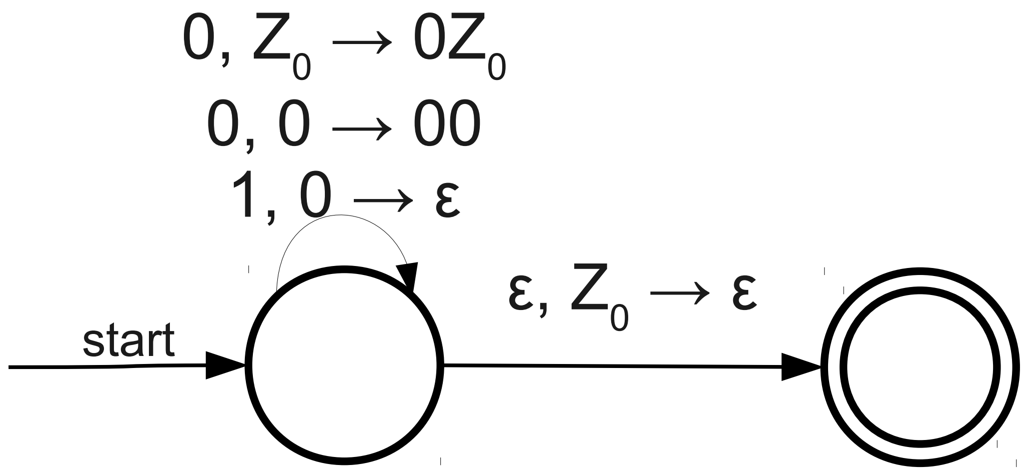
This does ***not*** pr… e is never a conflict between … ther transition.

> Sipser's definition of DPDAs does not allow the machine to "die" in some configuration.
> For CS103, we'll allow transitions to be missing.

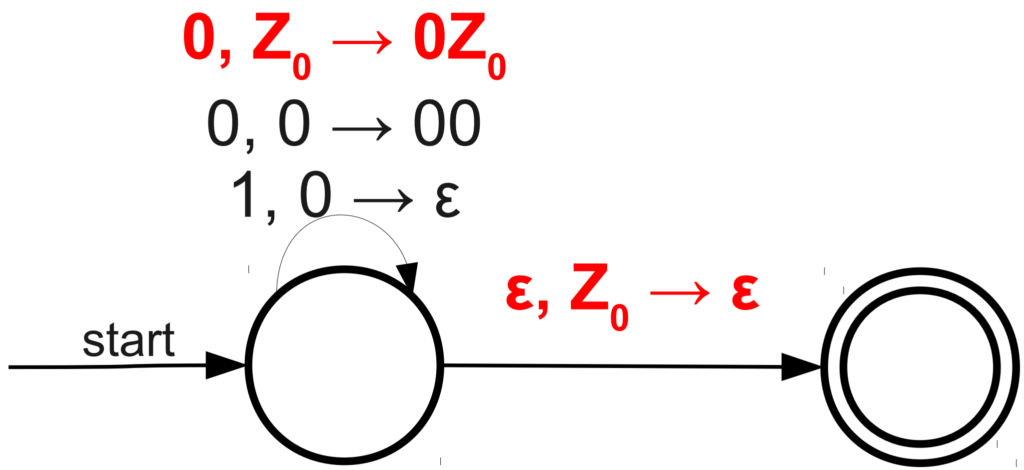
However, there can be ***at most*** one ε-transition that could be followed at any one time.

- This does ***not*** preclude the automaton "dying" from having no transitions defined; DPDAs can have undefined transitions.

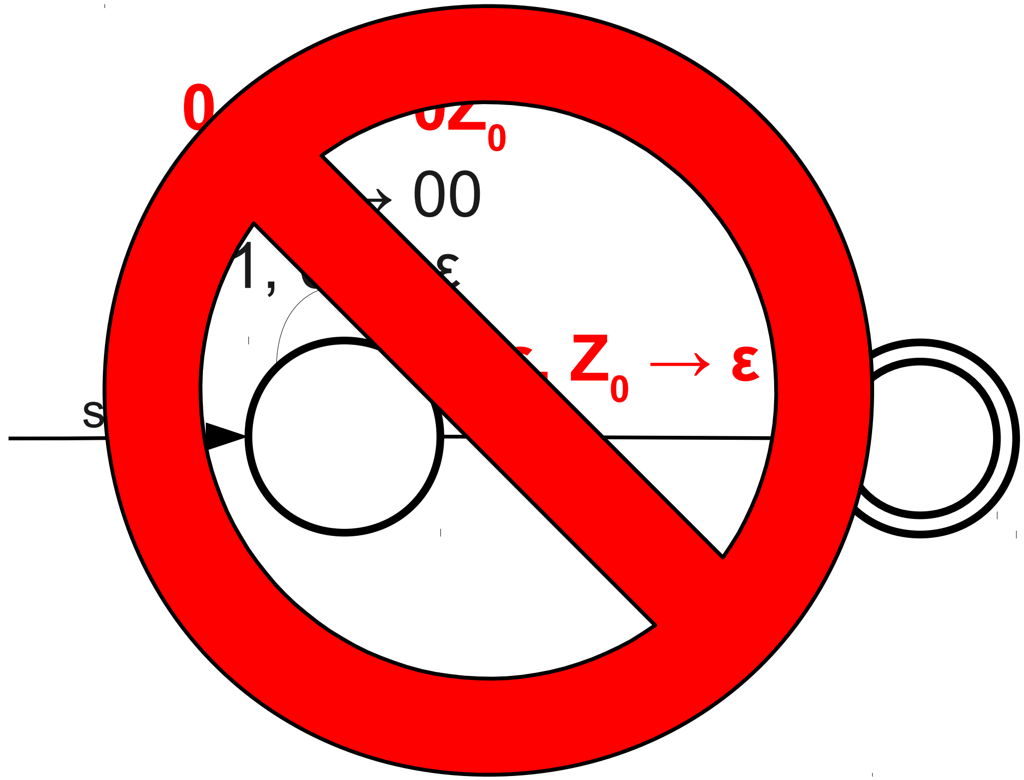
Is this a DPDA?



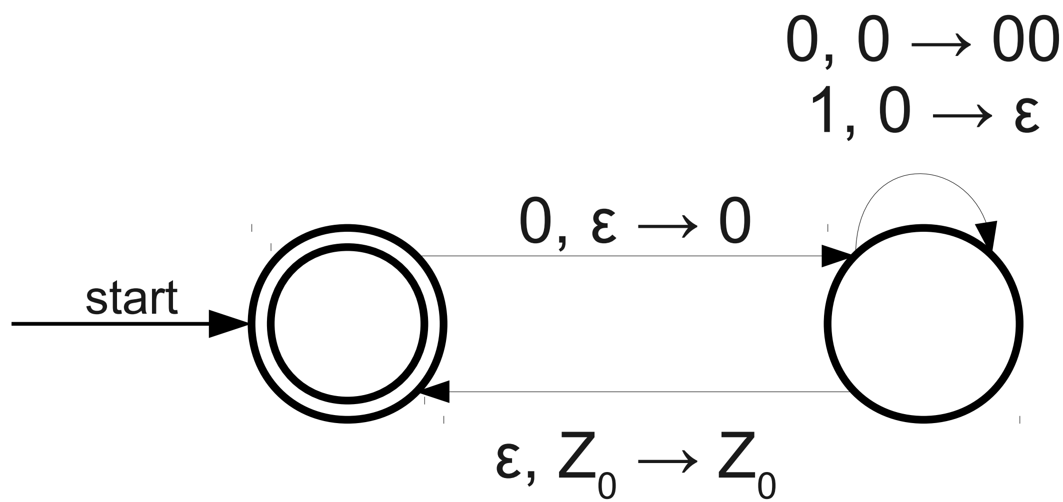
Is this a DPDA?



Is this a DPDA?

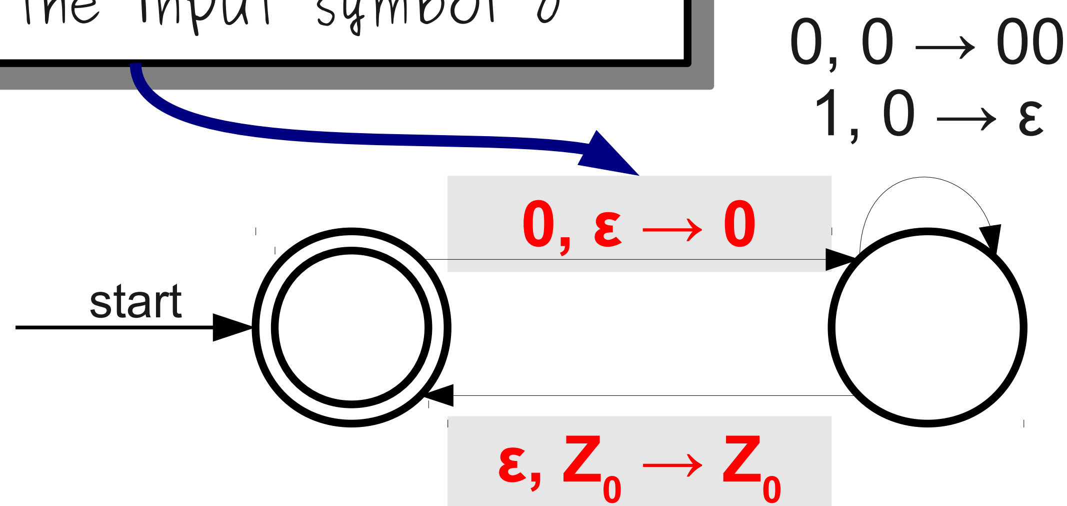


Is this a DPDA?



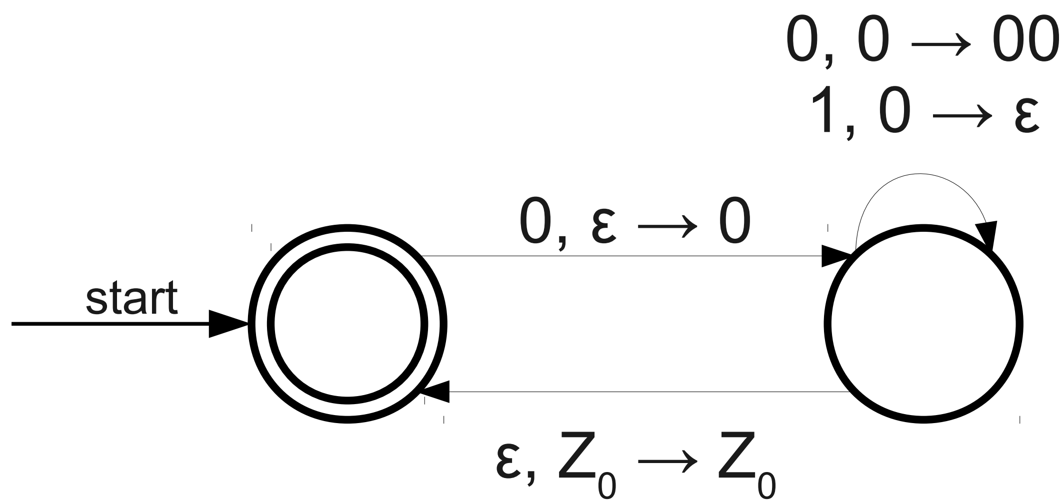
Is this a DPDA?

This ε–transition is allowable because no other transitions in this state use the input symbol 0

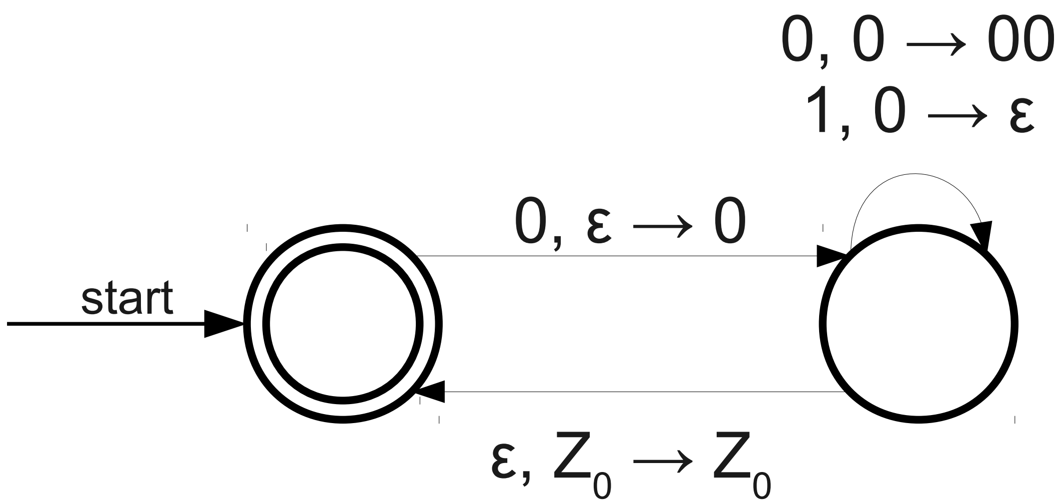


This ε–transition is allowable because no other transitions in this state use the stack symbol Z_0.

Is this a DPDA?

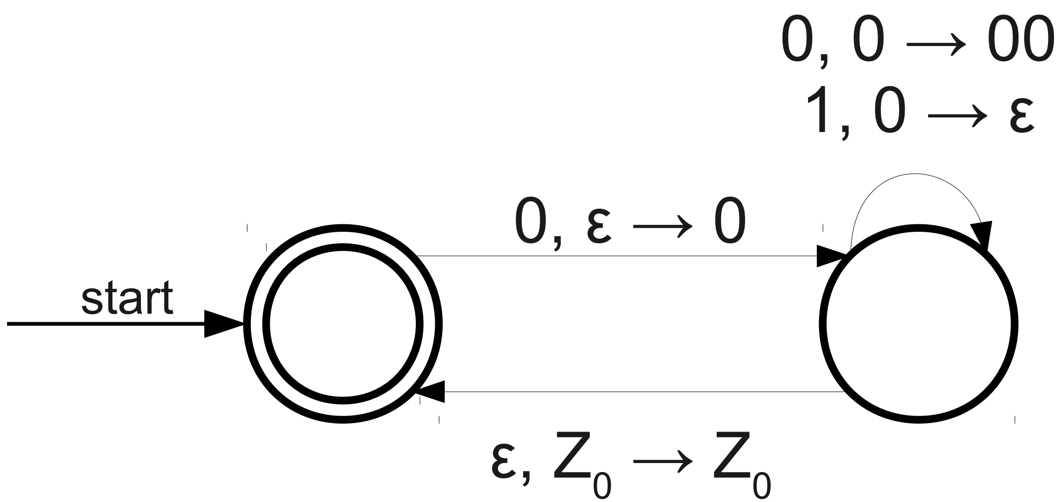


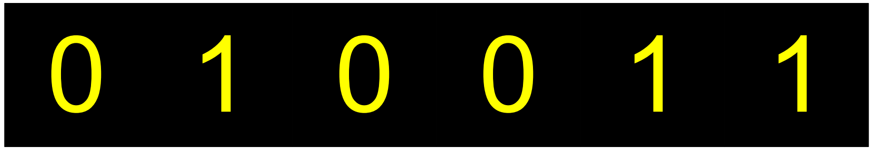
Is this a DPDA?

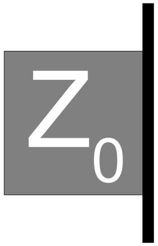


0 1 0 0 1 1

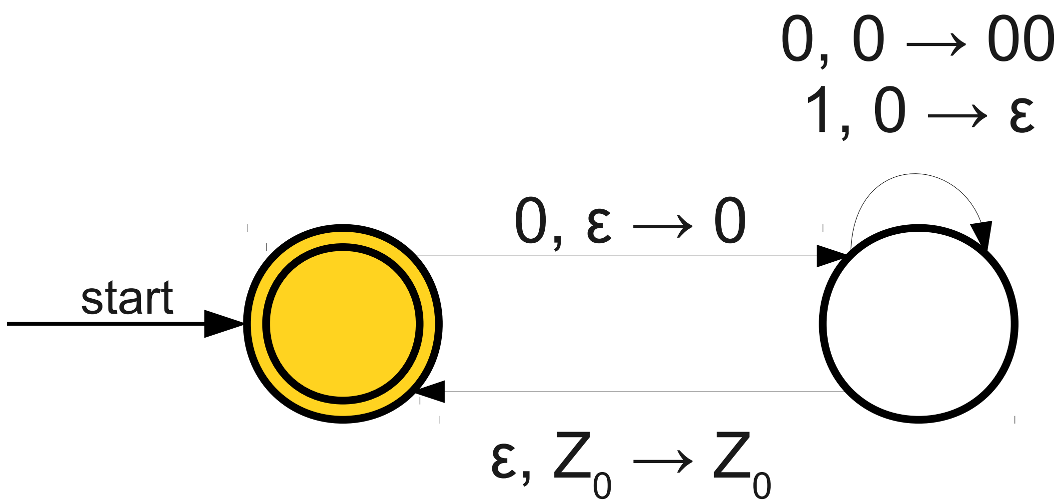
Is this a DPDA?

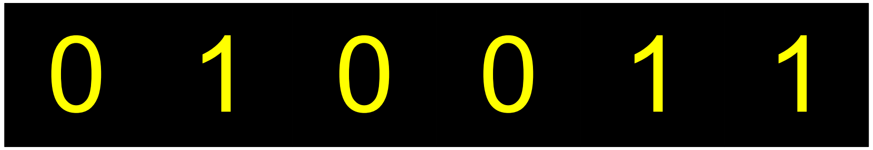


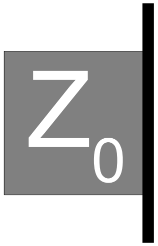




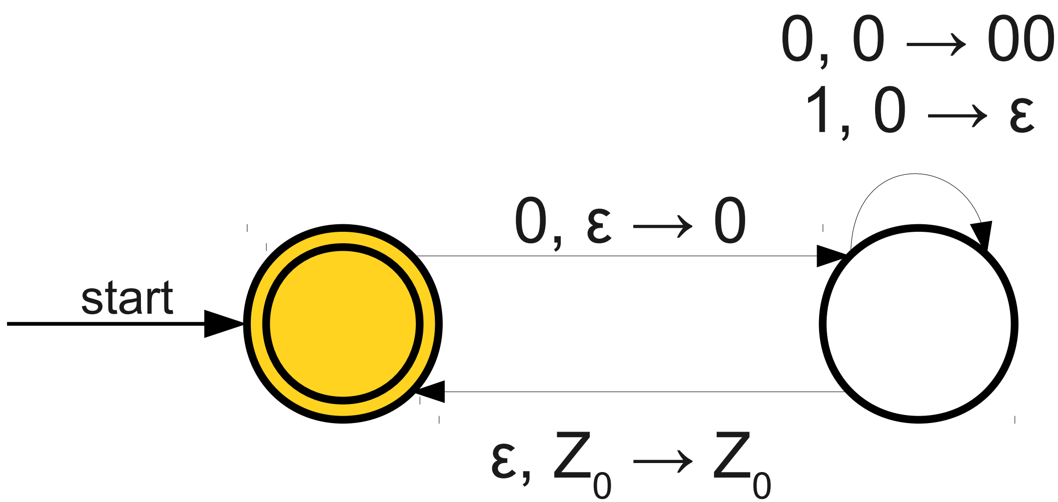
Is this a DPDA?



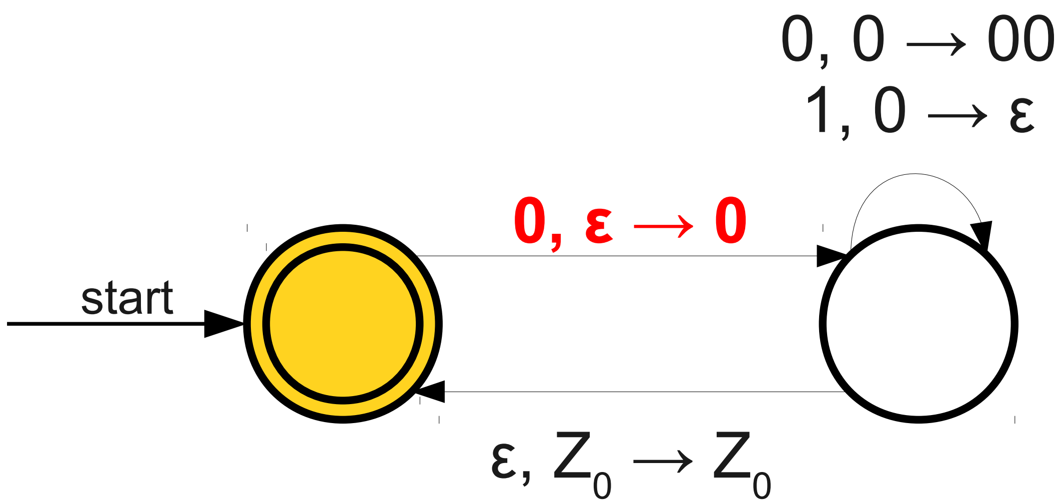


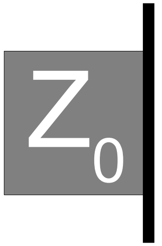


Is this a DPDA?

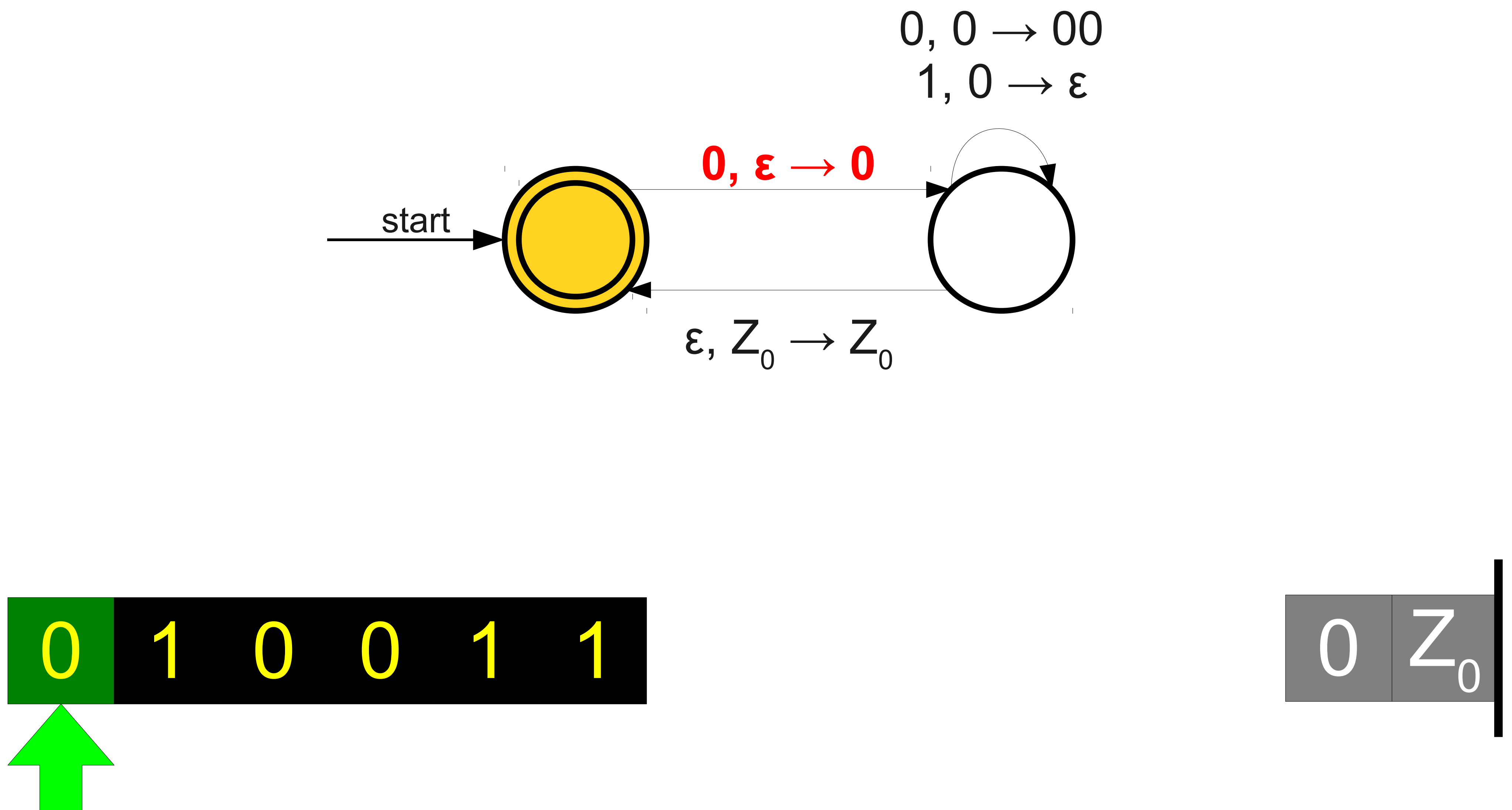


Is this a DPDA?

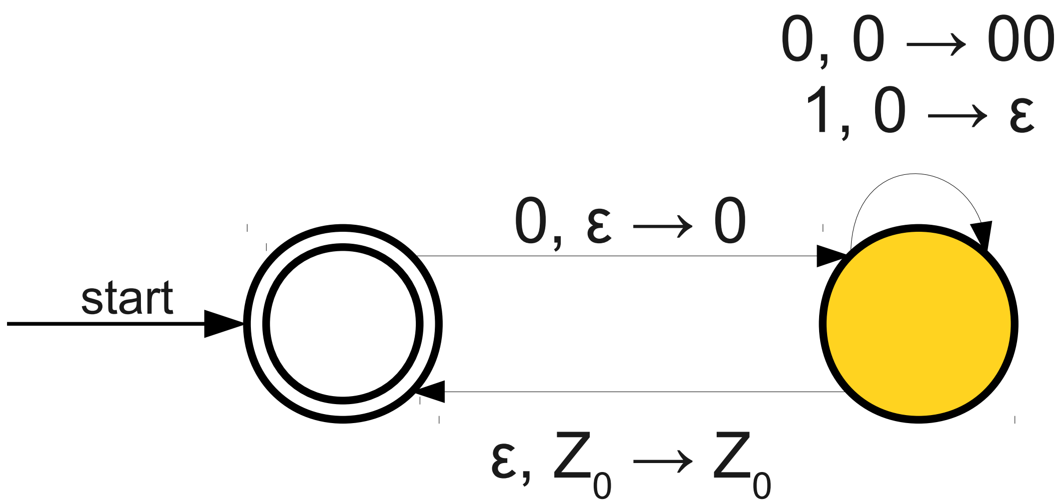


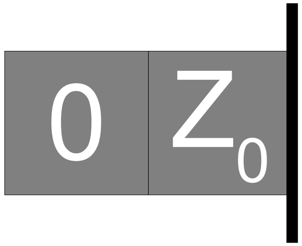


Is this a DPDA?

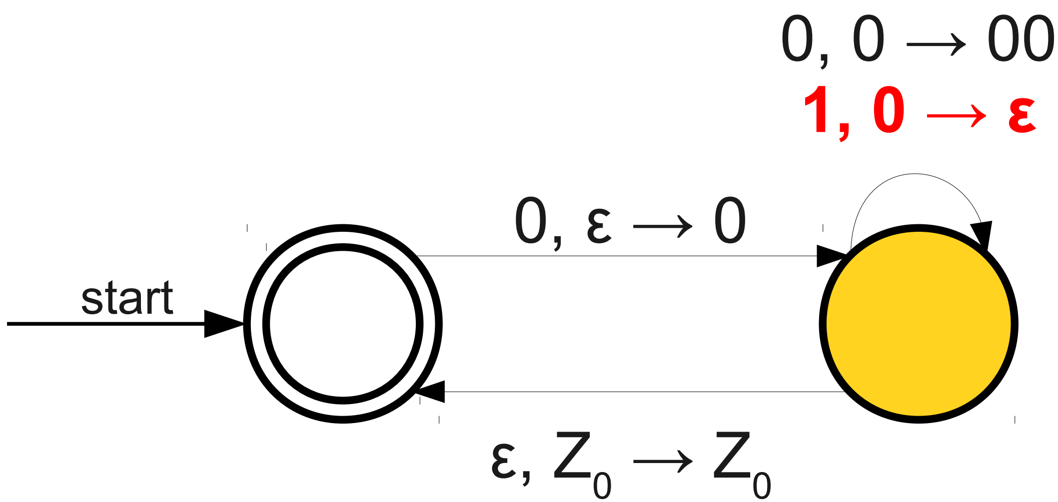


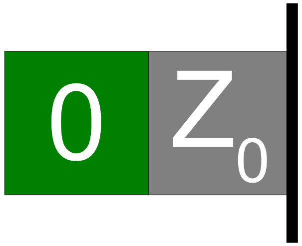
Is this a DPDA?



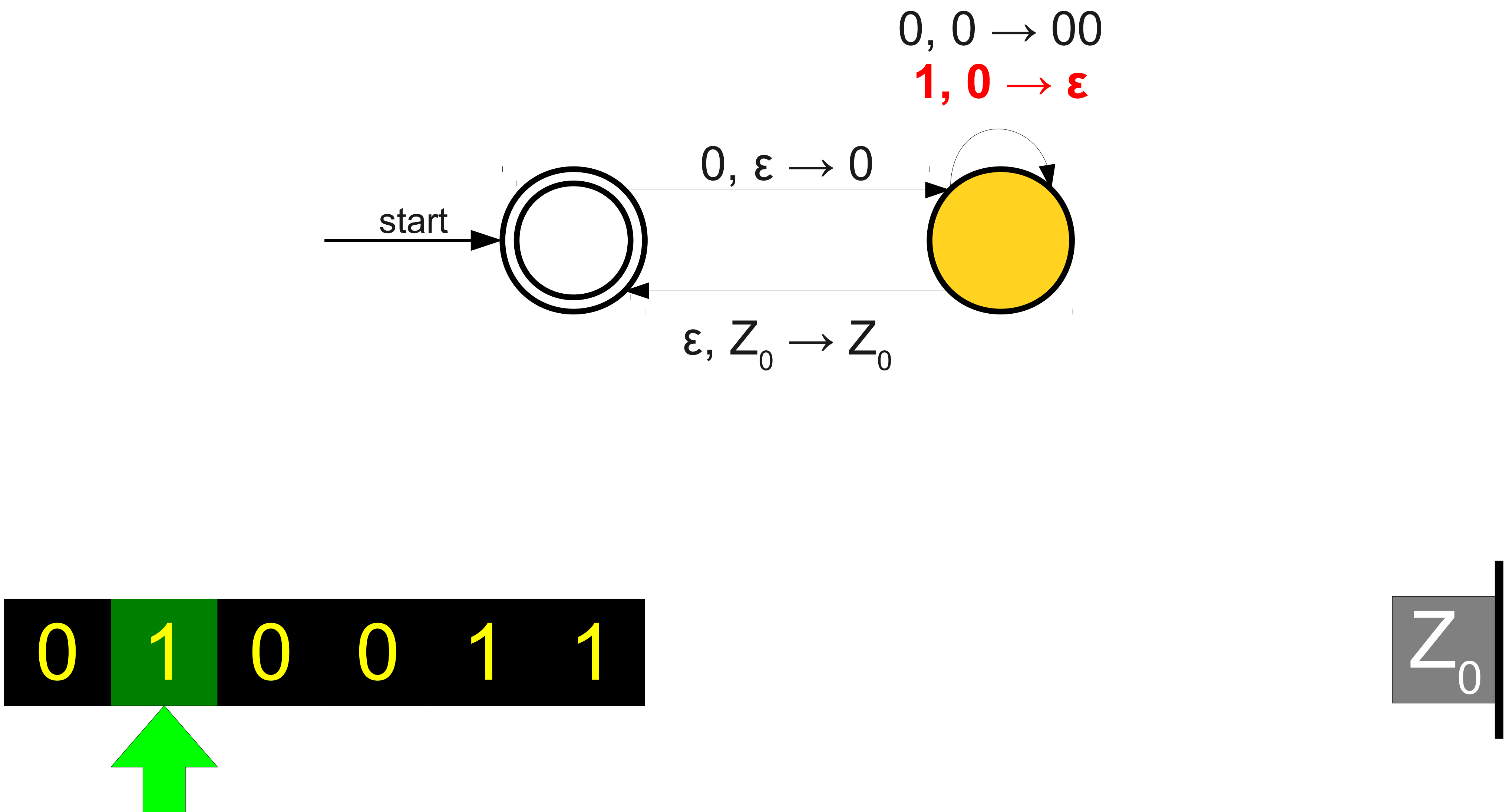


Is this a DPDA?

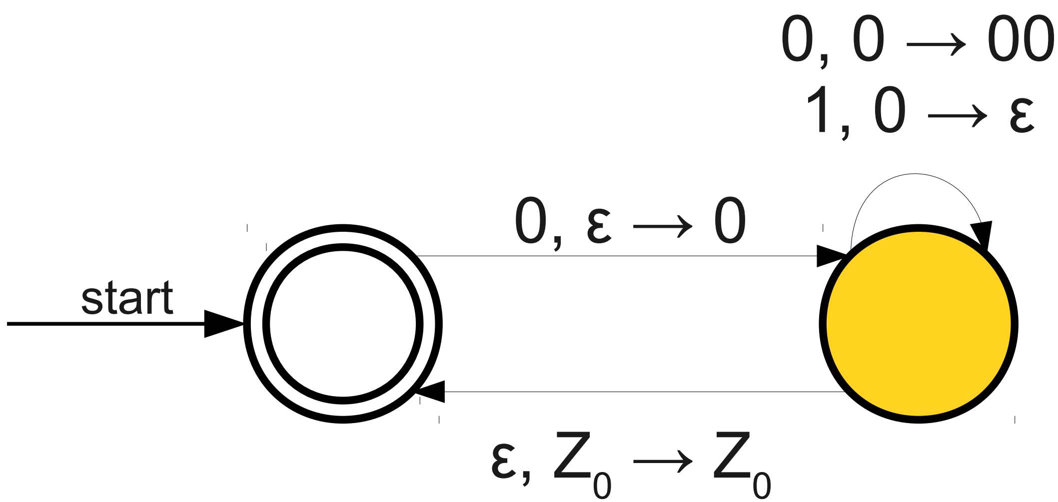


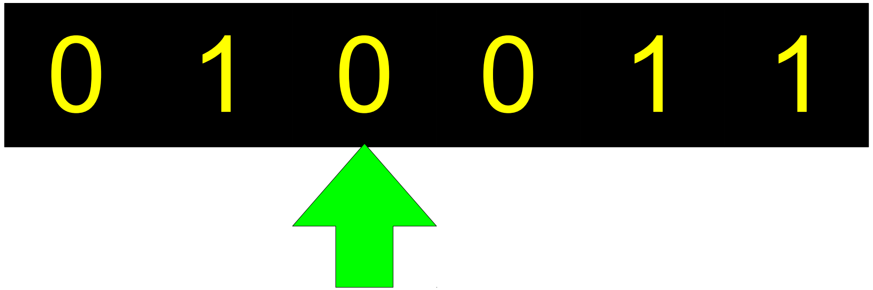


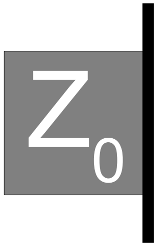
Is this a DPDA?



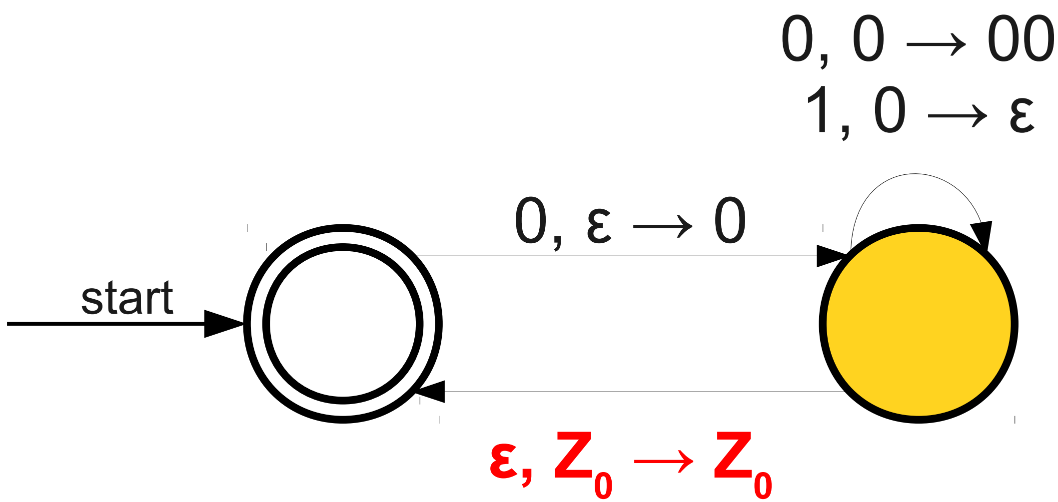
Is this a DPDA?



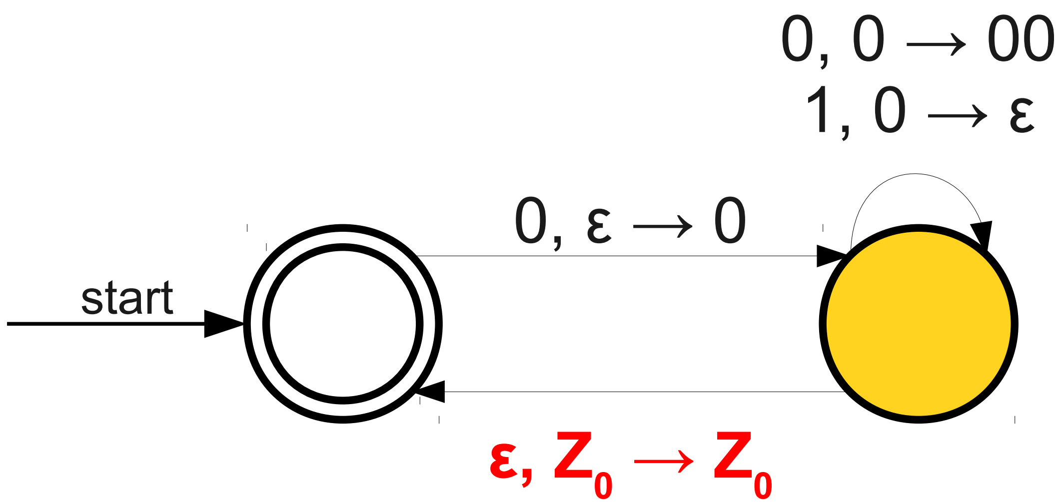




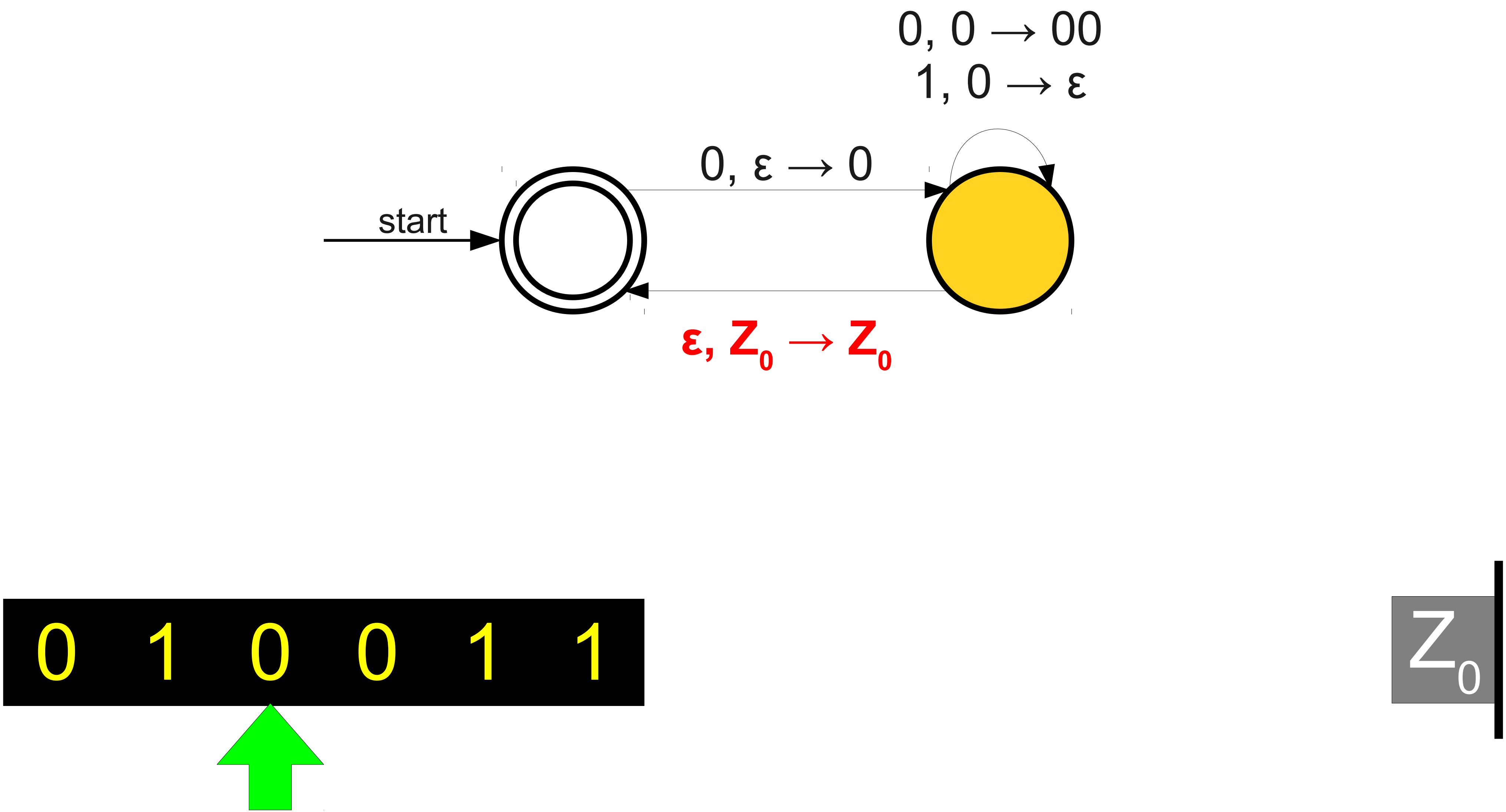
Is this a DPDA?



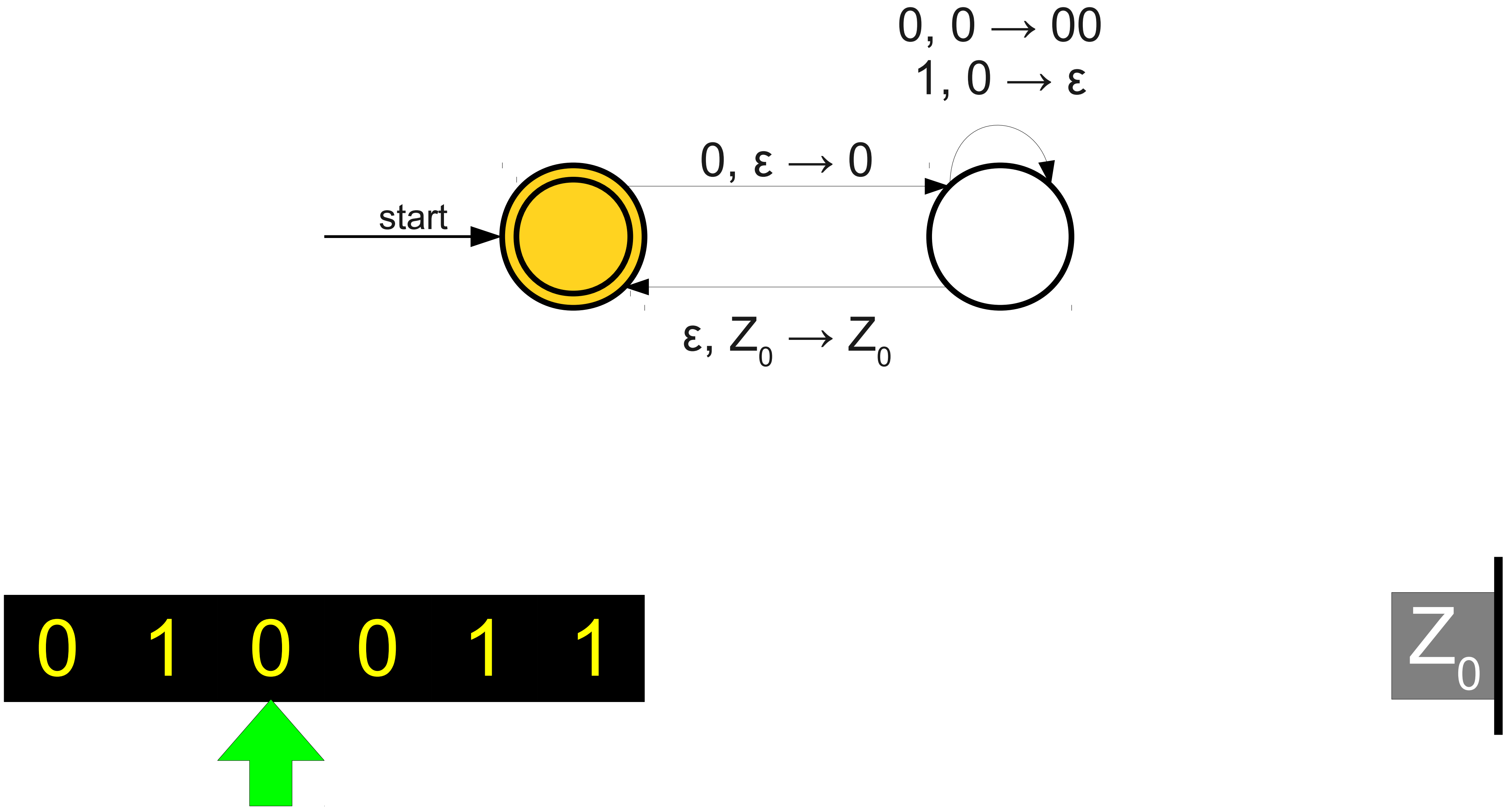
Is this a DPDA?



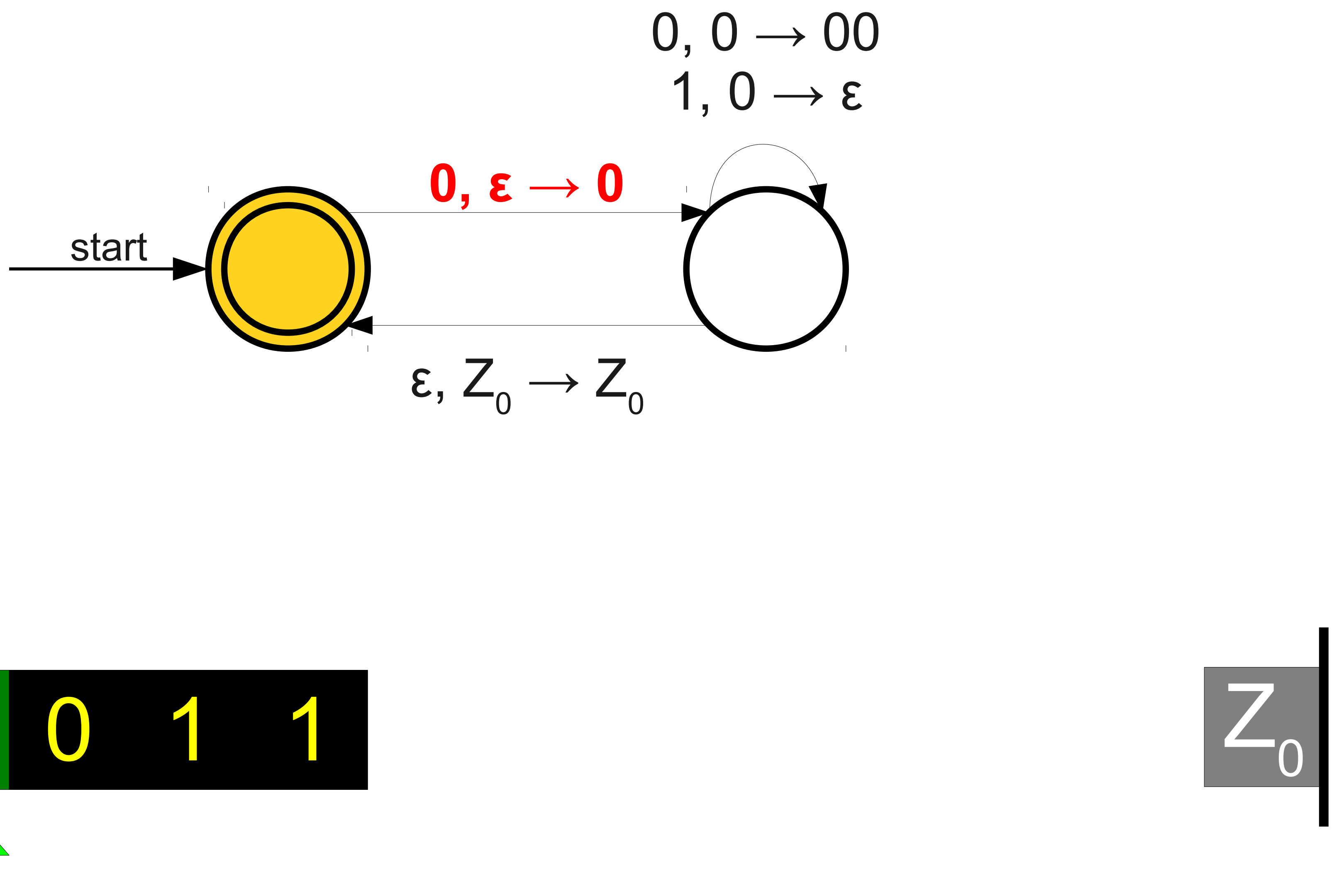
Is this a DPDA?



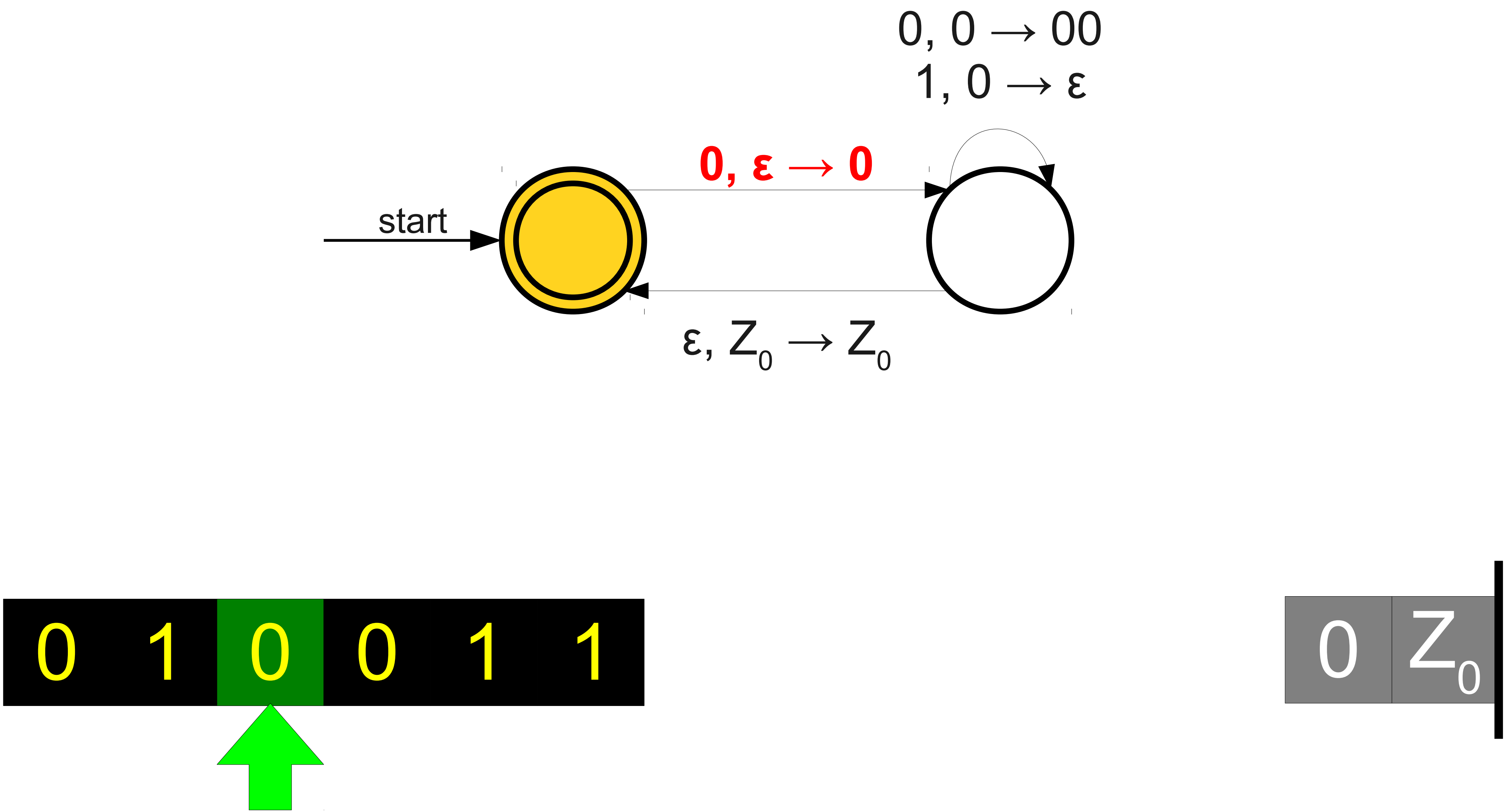
Is this a DPDA?



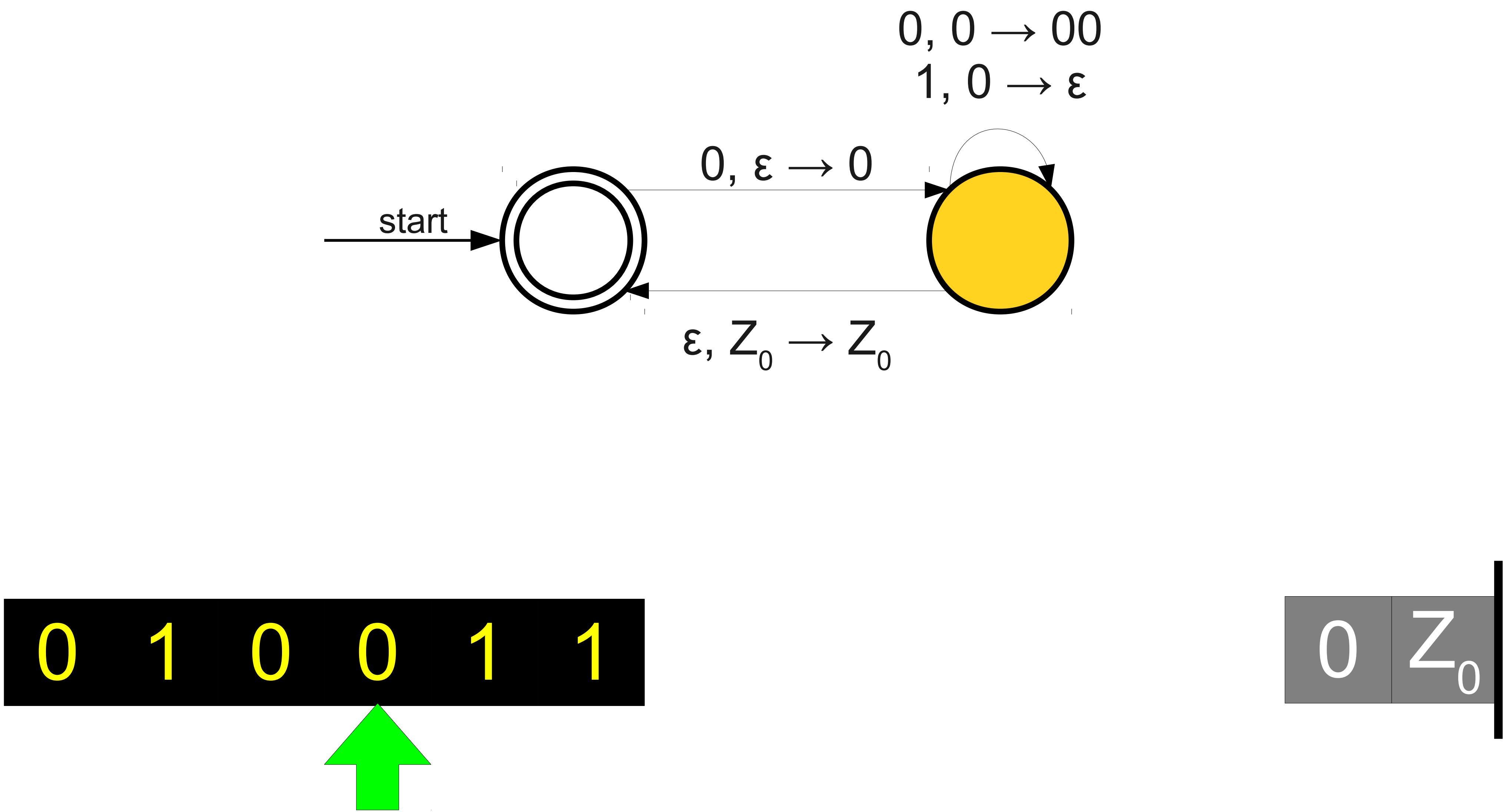
Is this a DPDA?



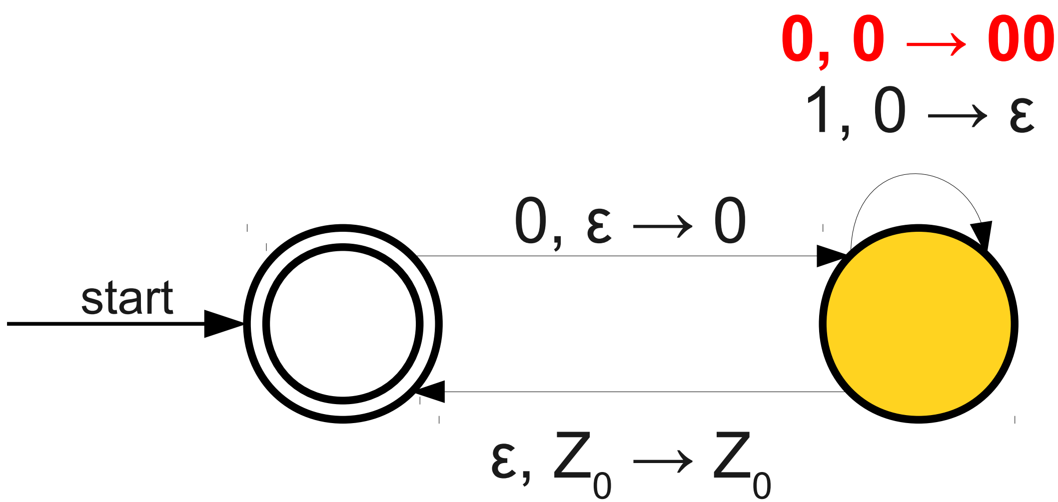
Is this a DPDA?

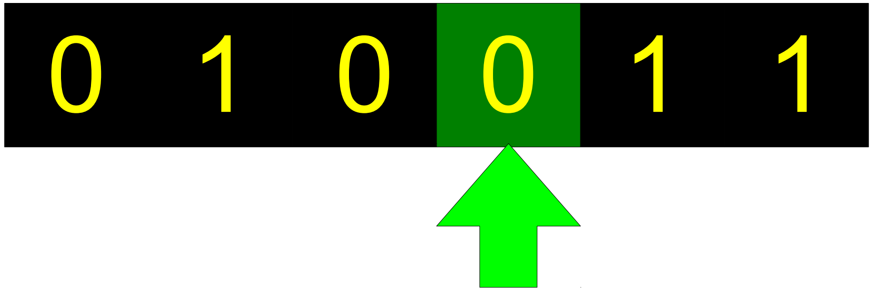


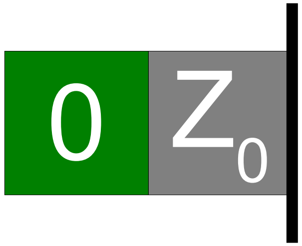
Is this a DPDA?



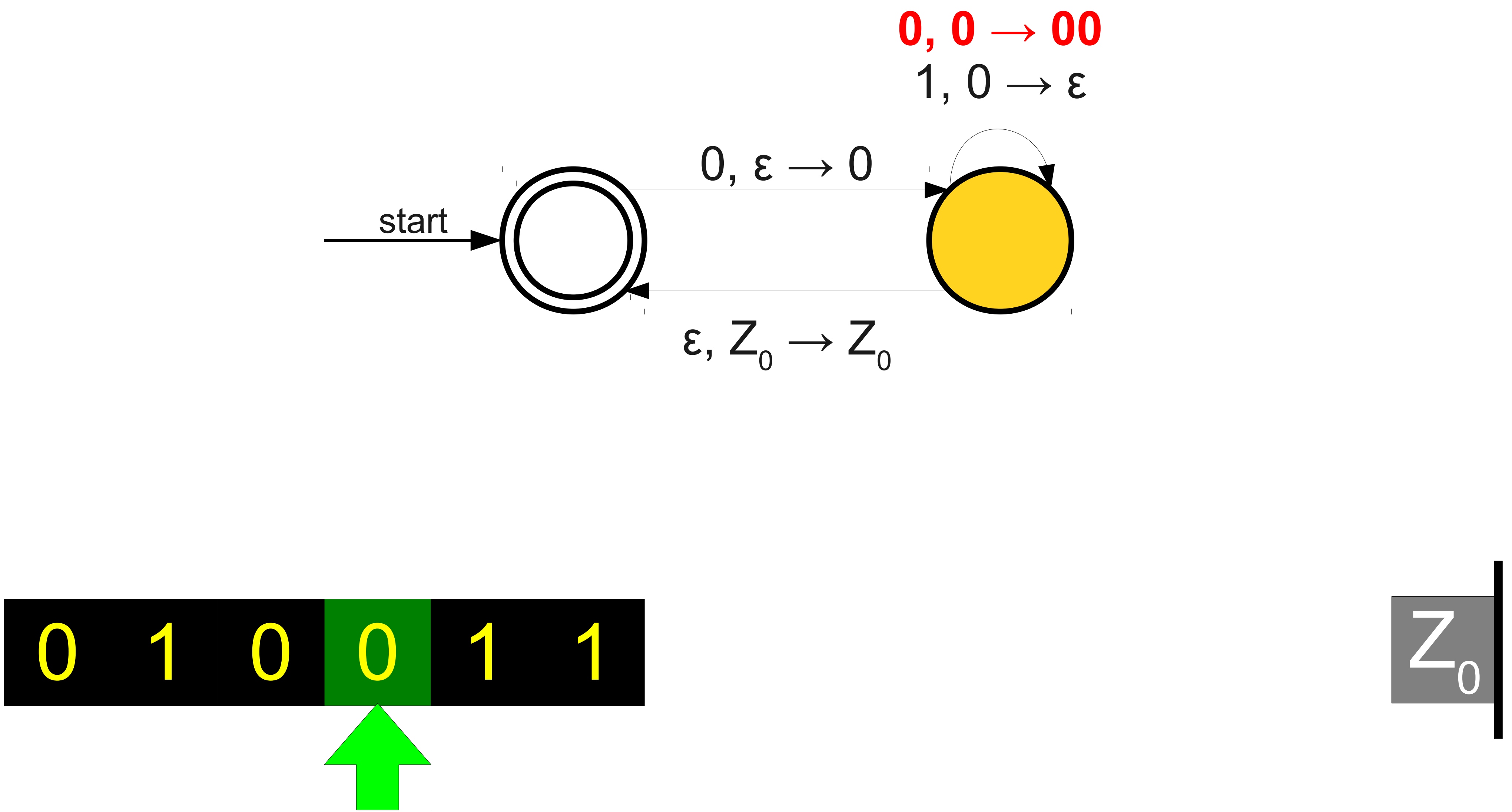
Is this a DPDA?



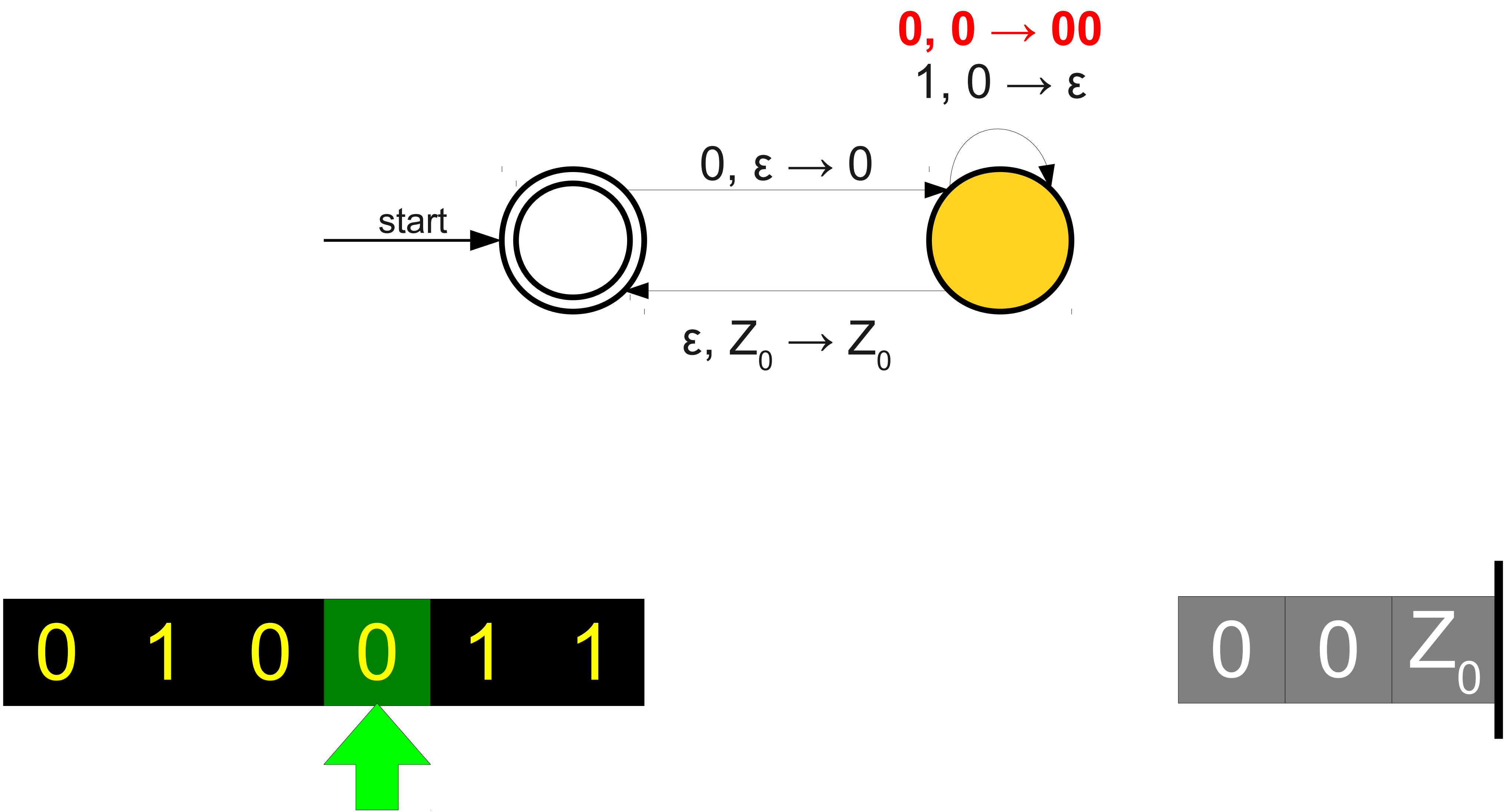




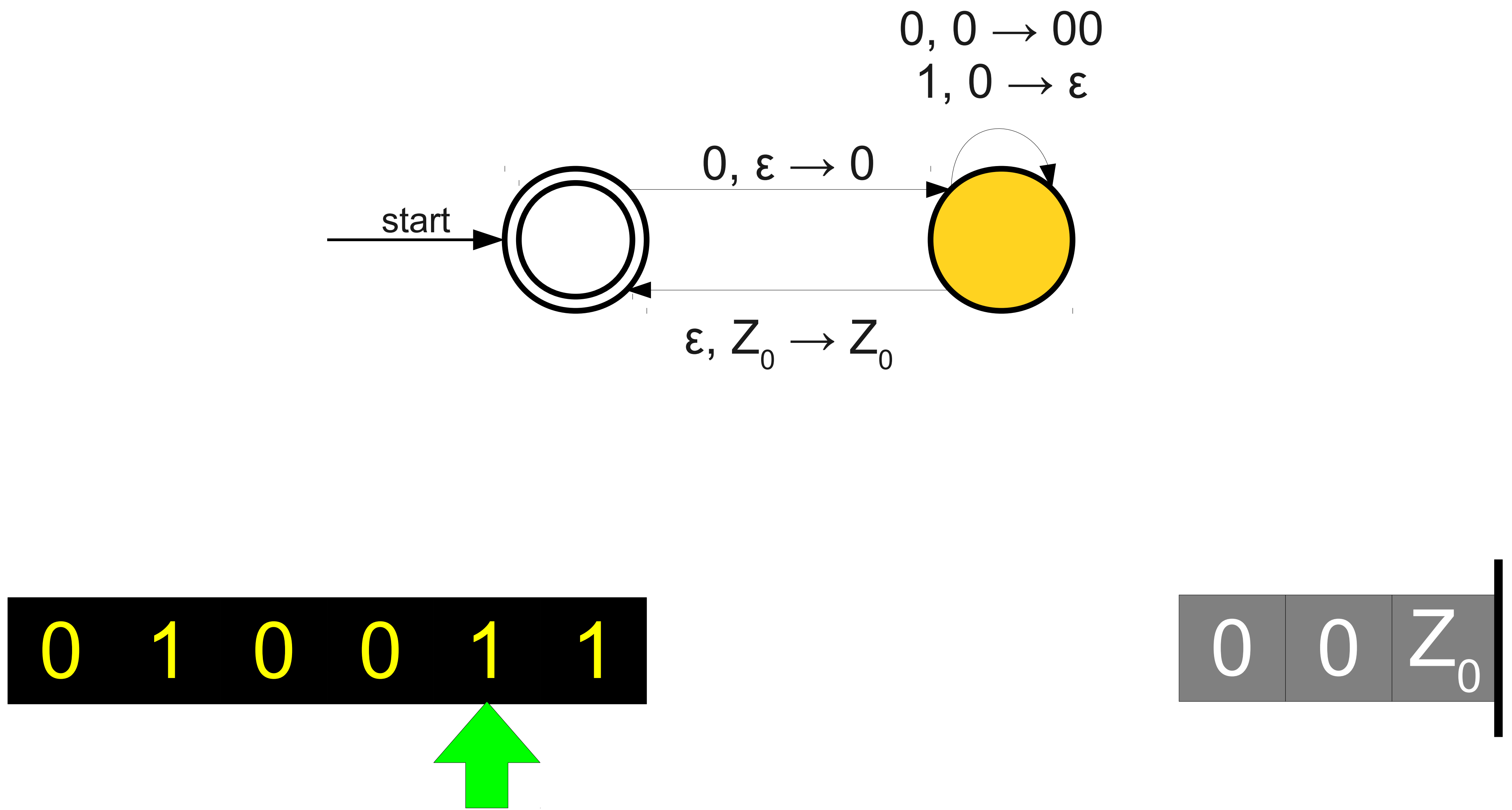
Is this a DPDA?



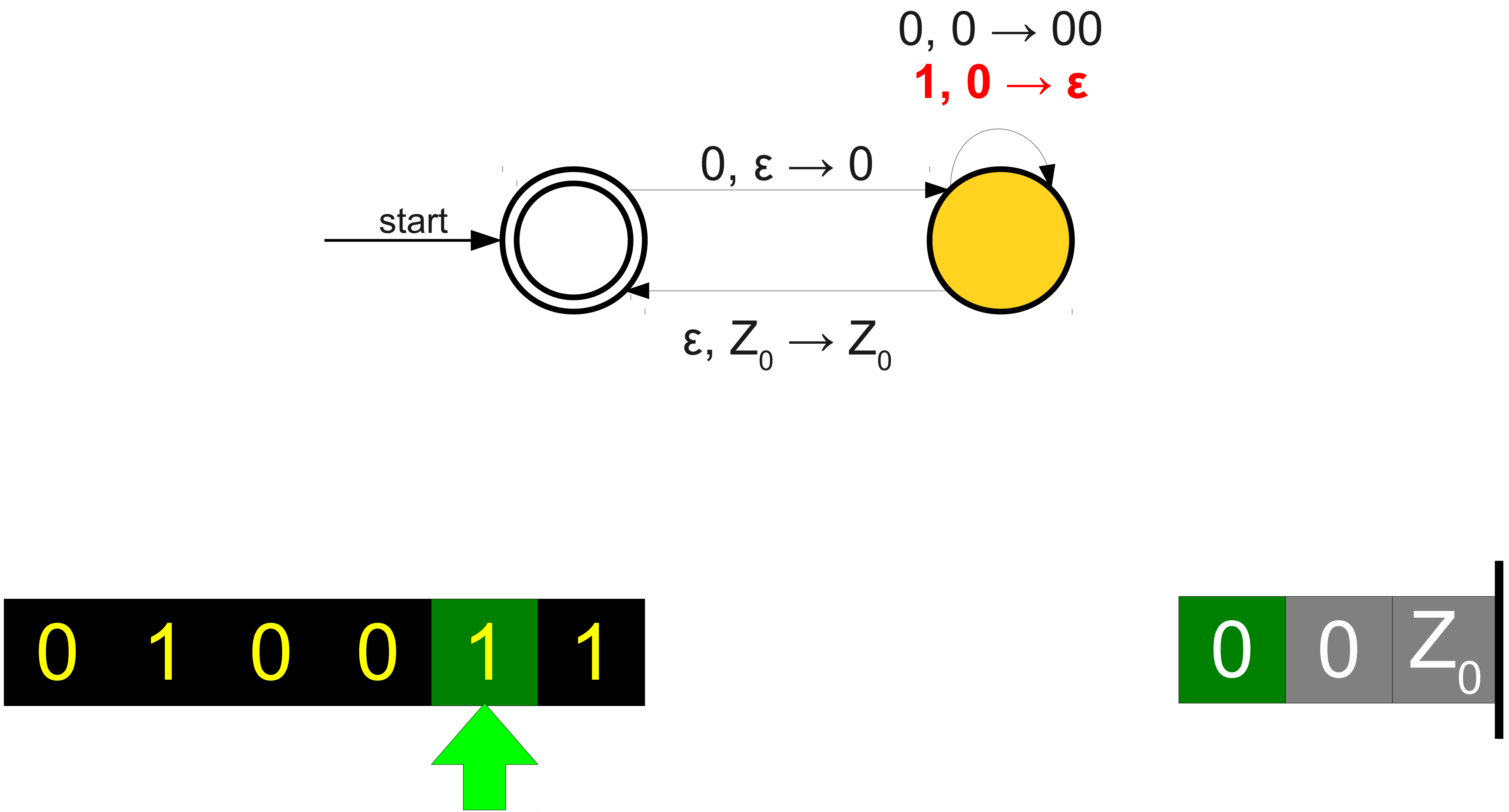
Is this a DPDA?



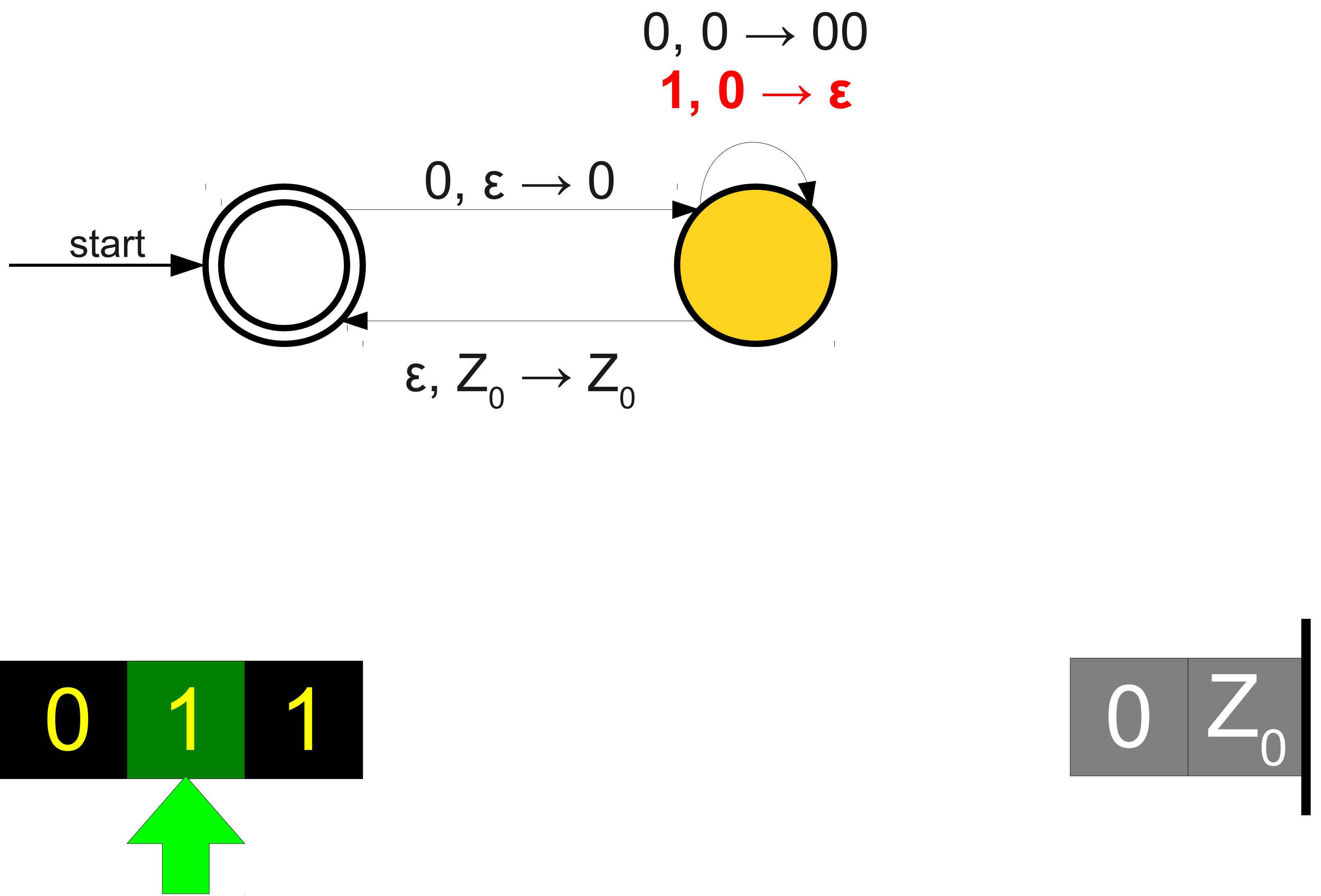
Is this a DPDA?



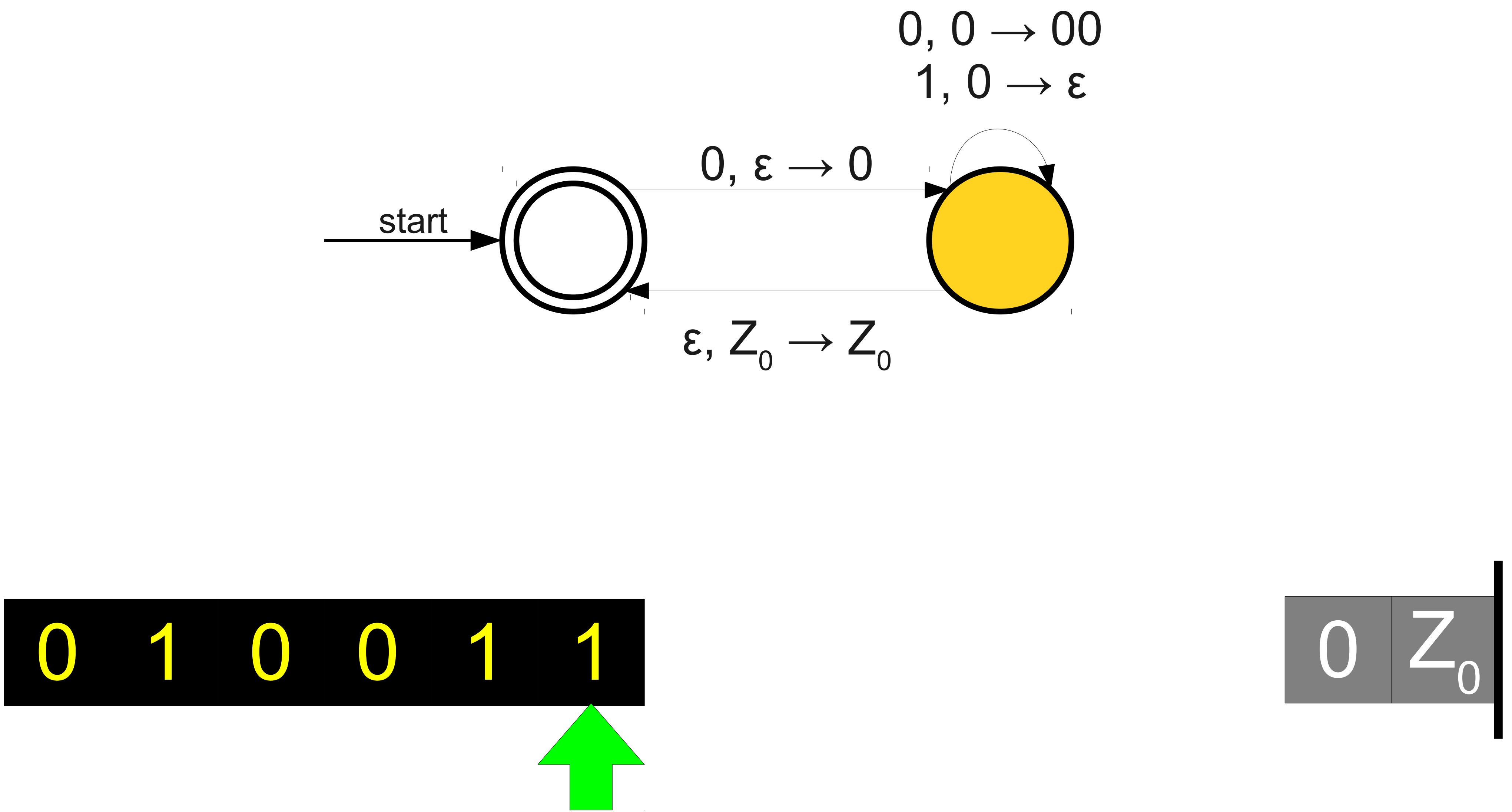
Is this a DPDA?



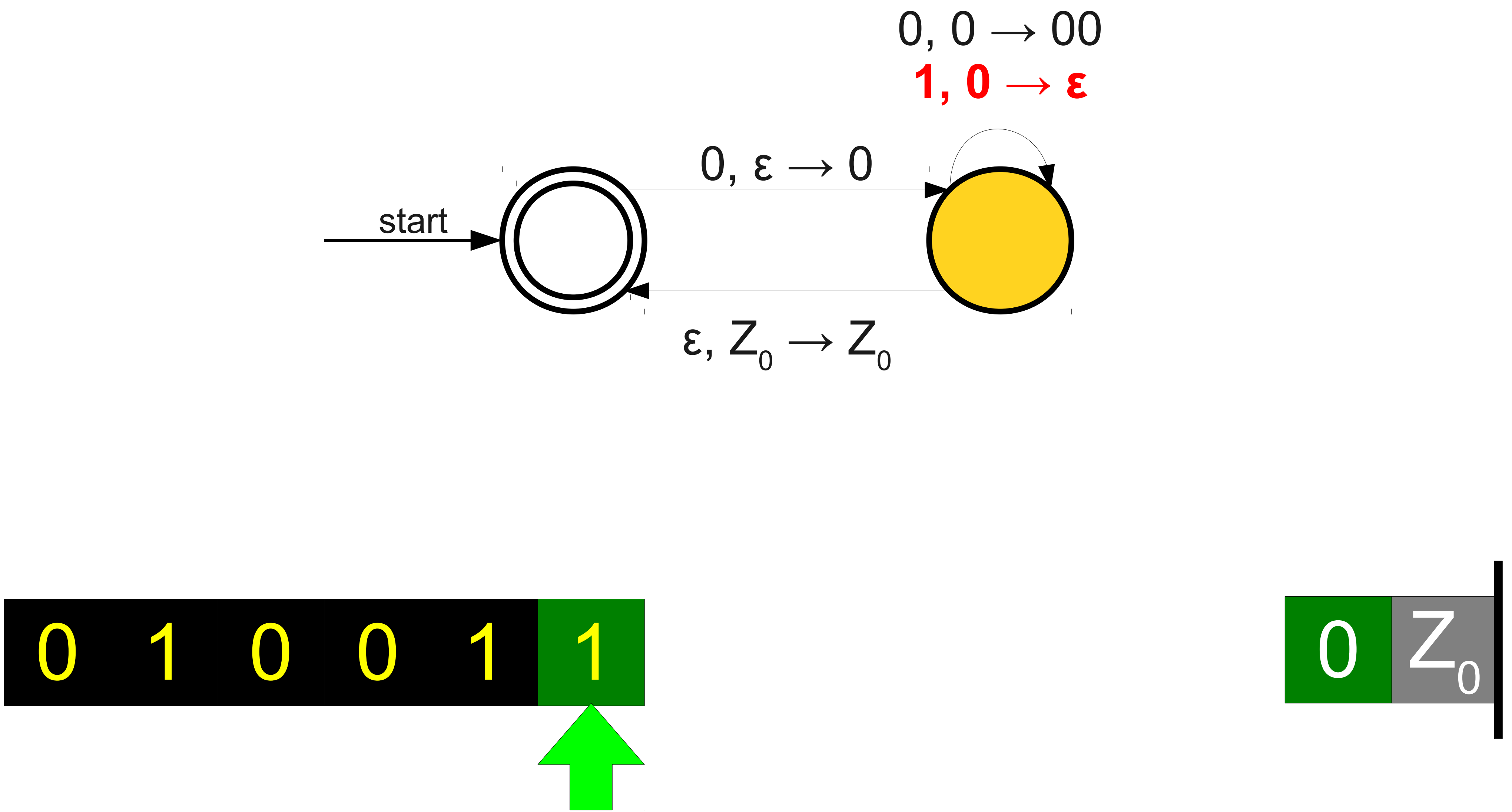
Is this a DPDA?



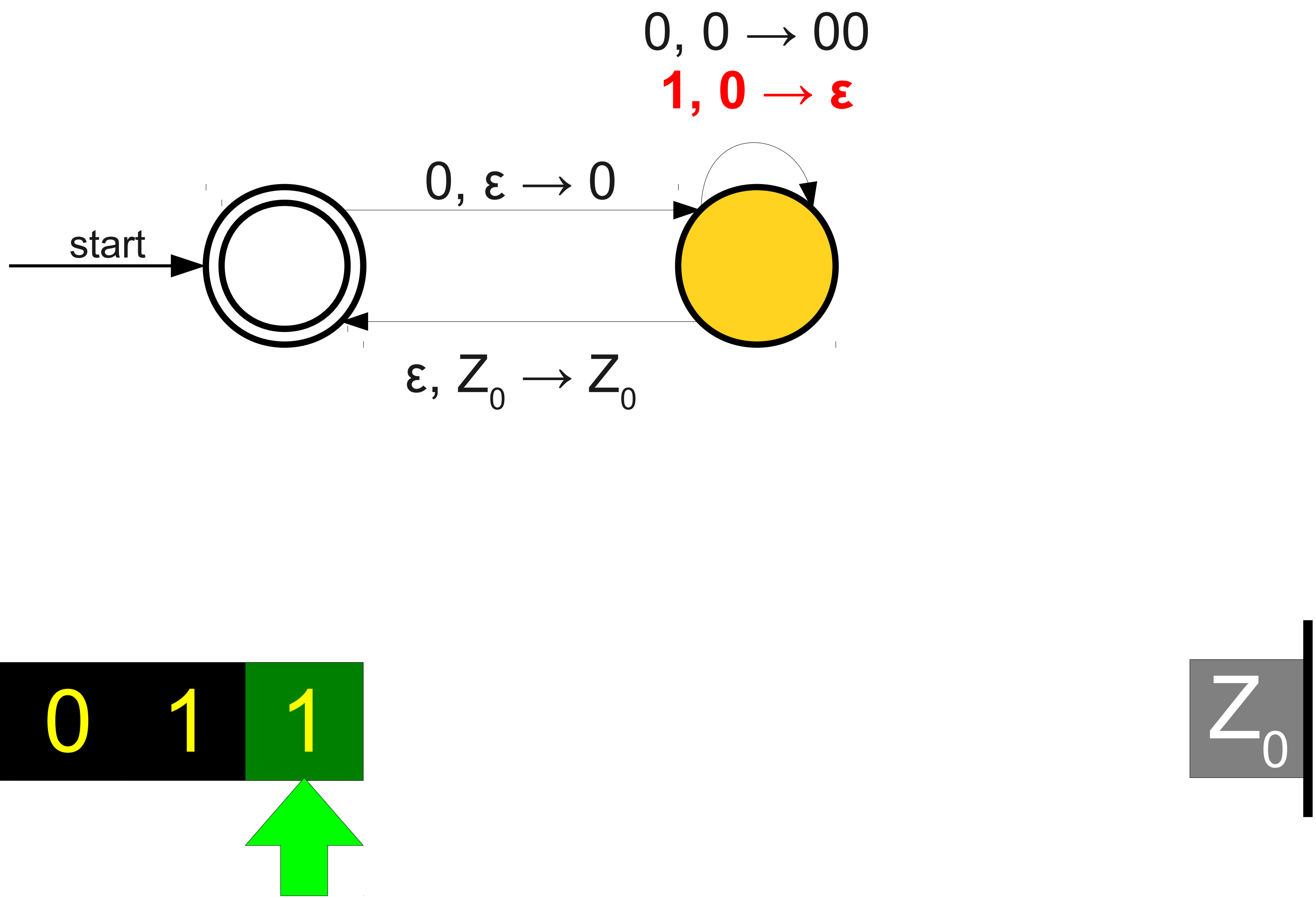
Is this a DPDA?



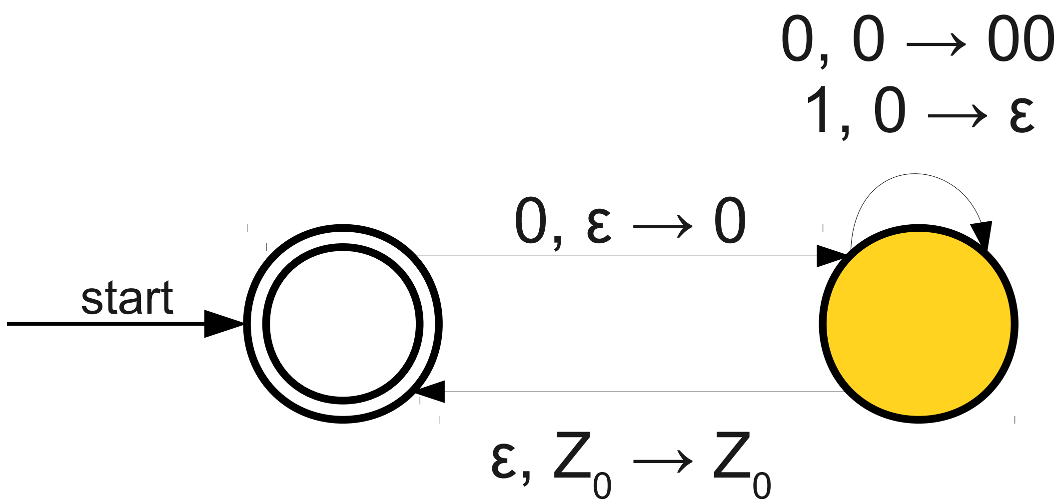
Is this a DPDA?

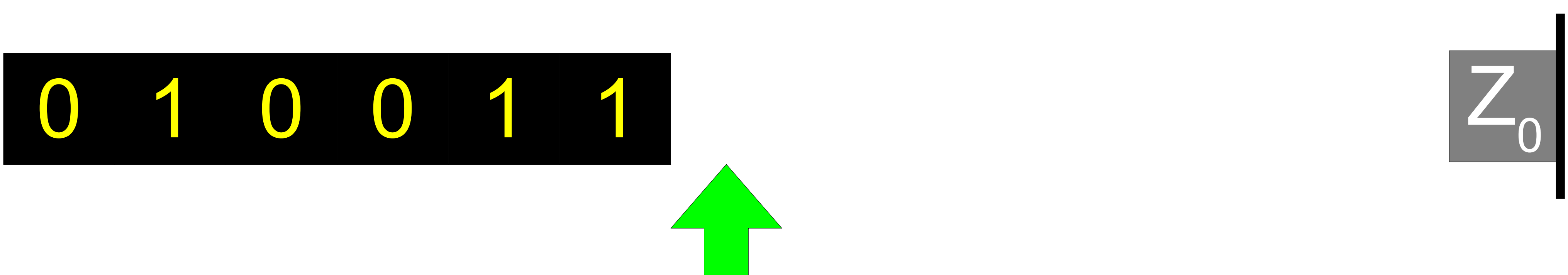


Is this a DPDA?

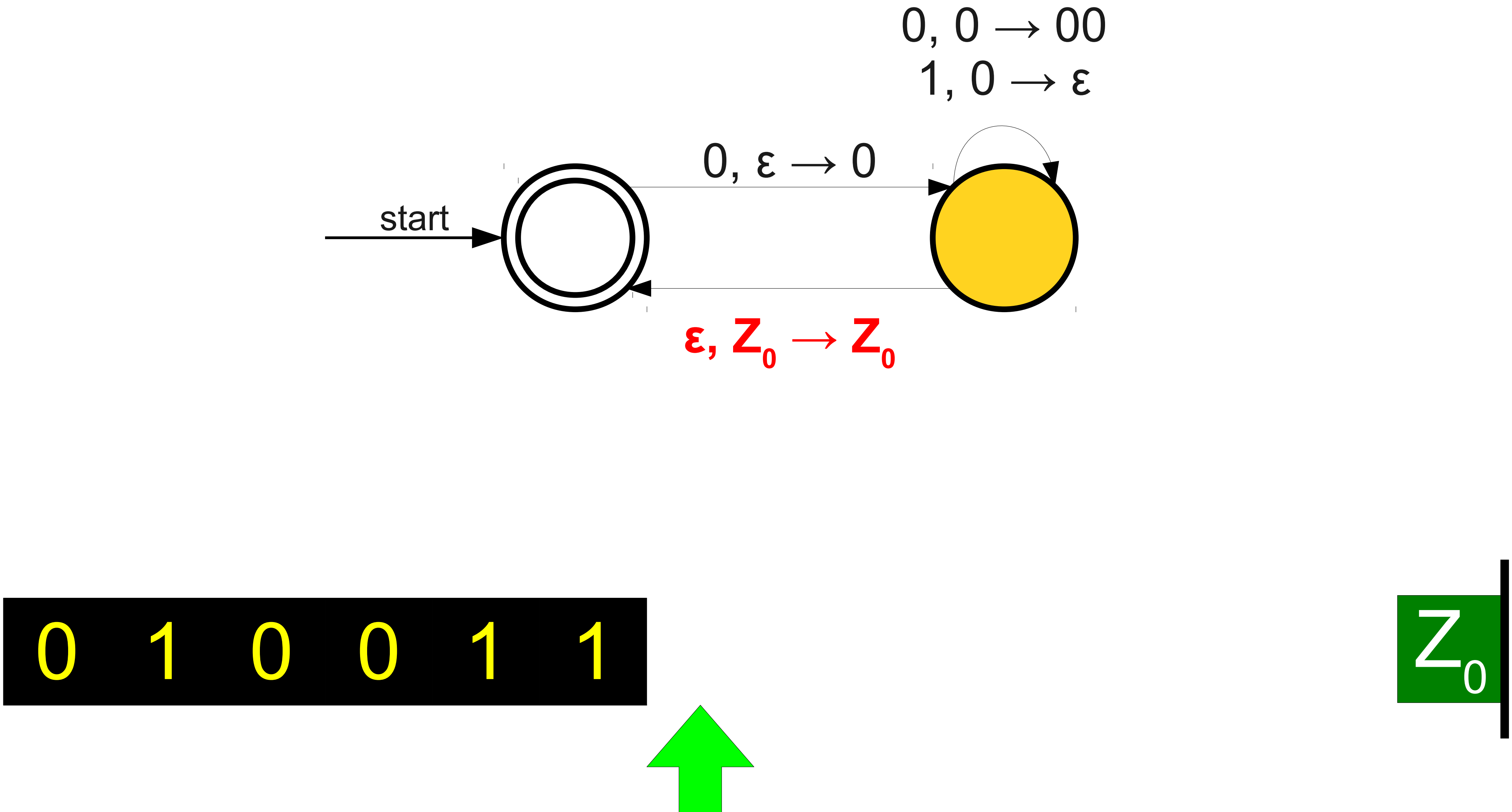


Is this a DPDA?

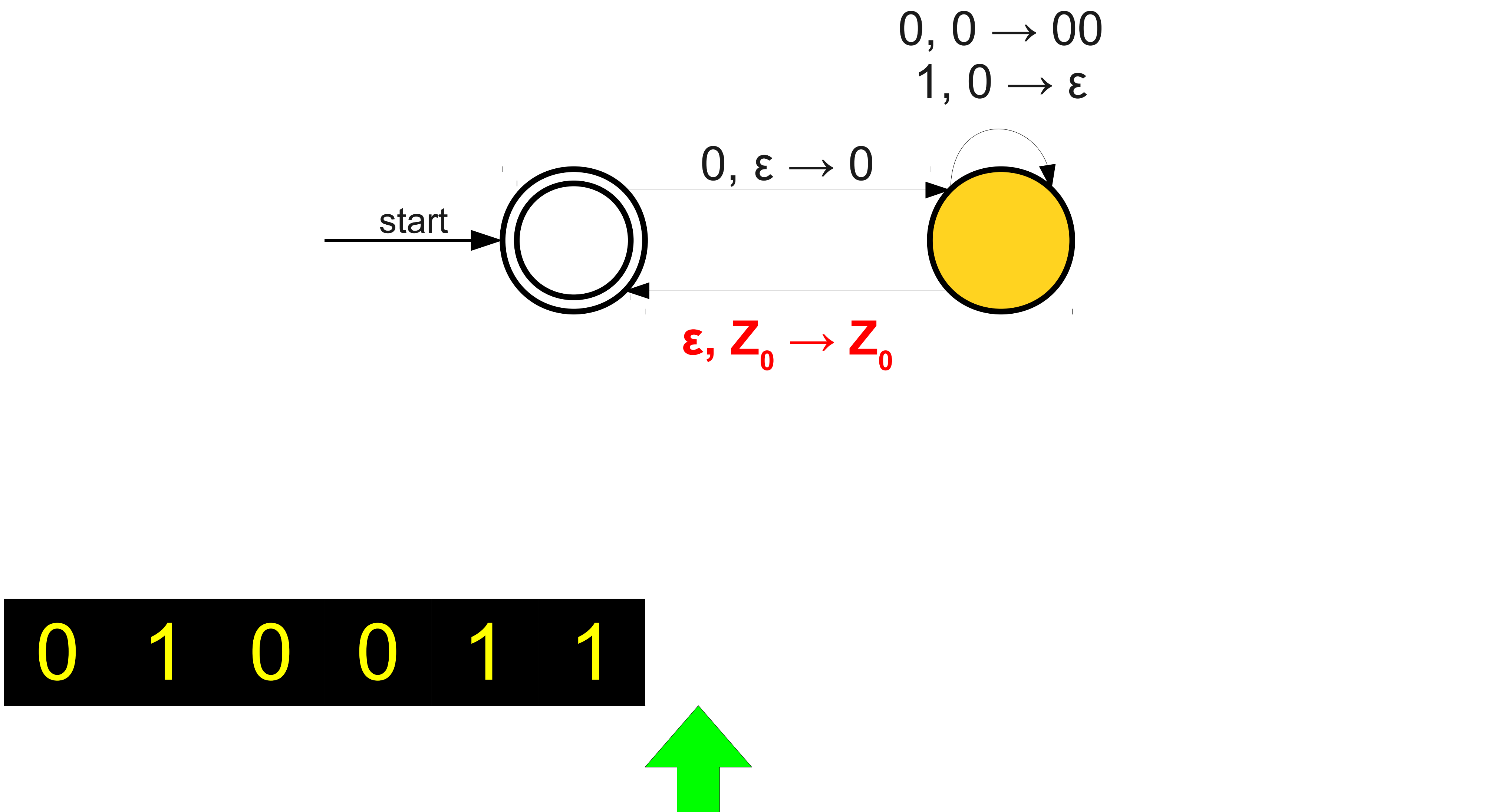




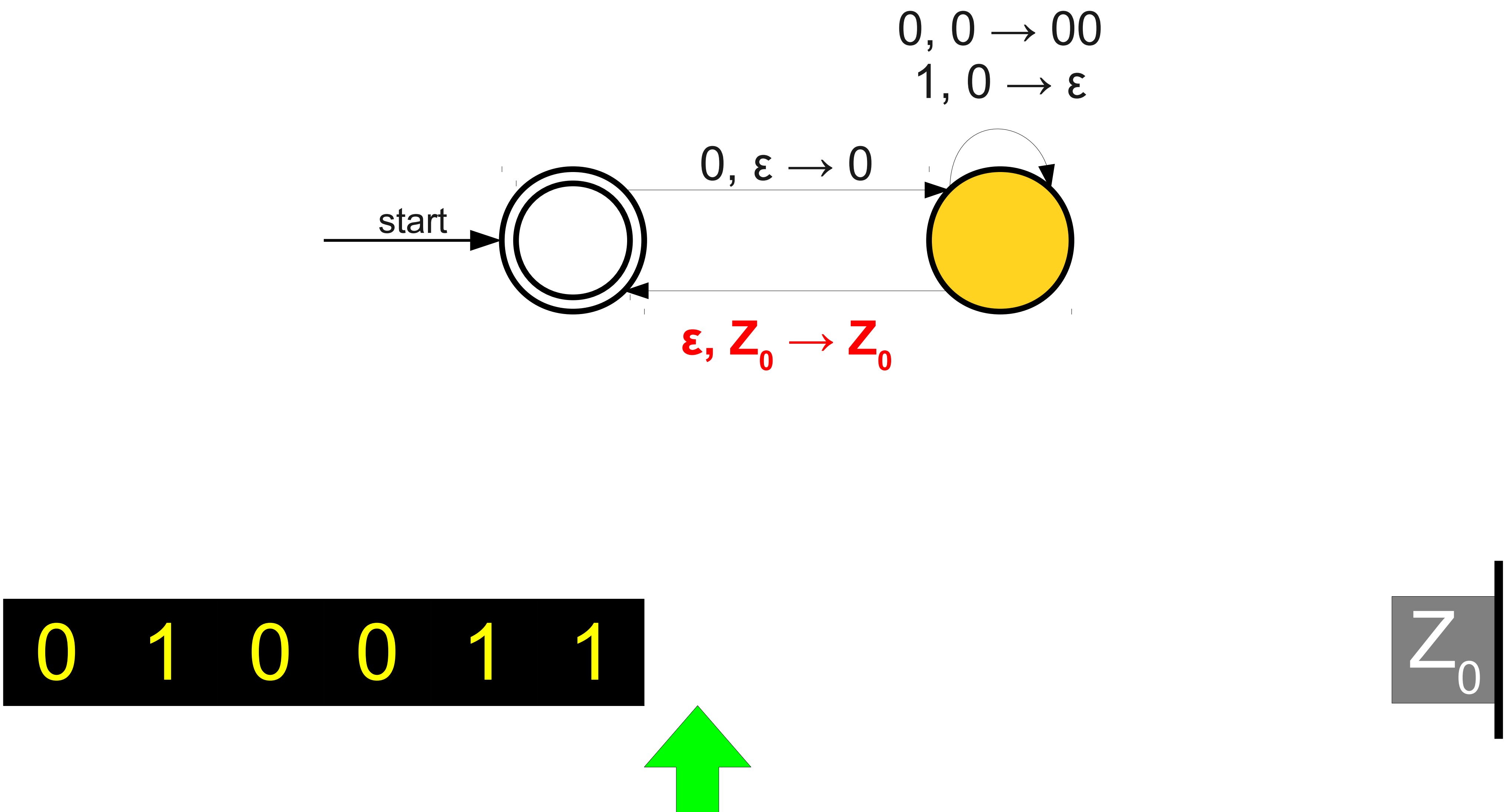
Is this a DPDA?



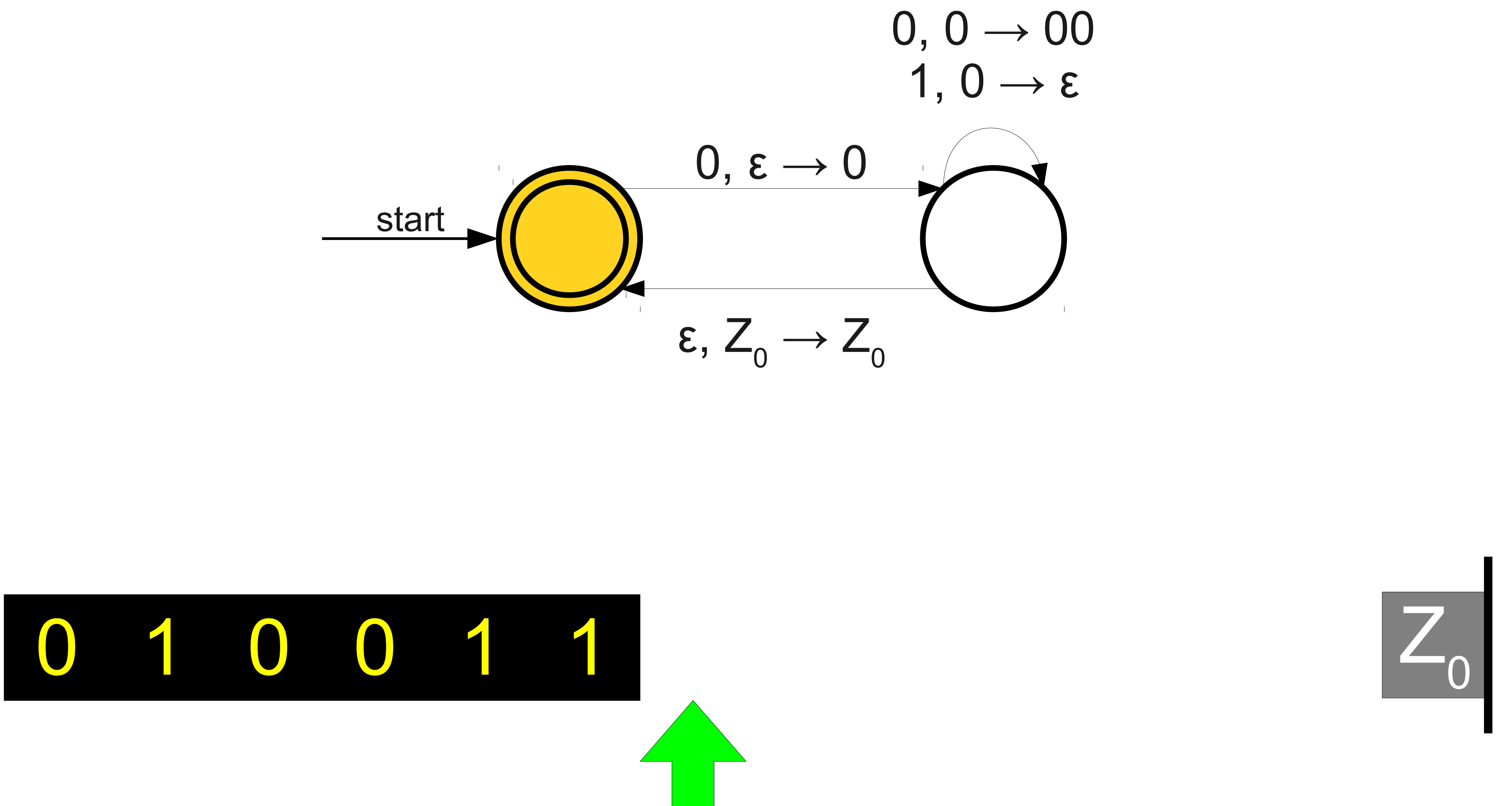
Is this a DPDA?



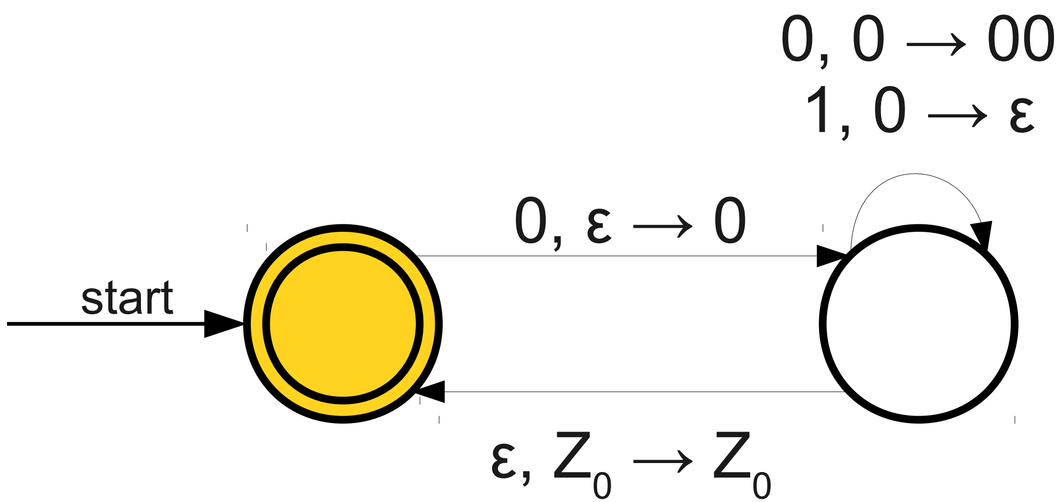
Is this a DPDA?



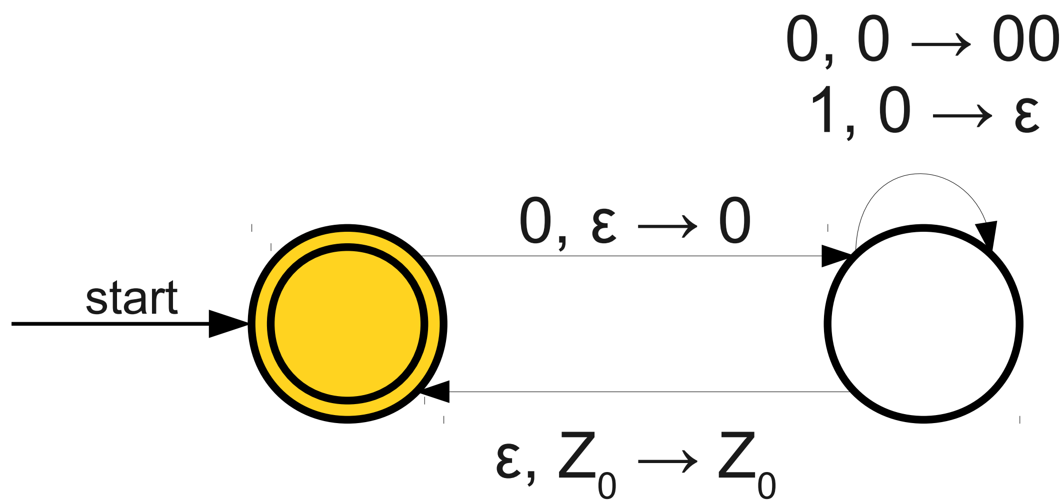
Is this a DPDA?



Is this a DPDA?



Is this a DPDA?



0 1 0 0 1 1

Z_0

Why DPDAs Matter

- Because DPDAs are deterministic, they can be simulated efficiently:
  - Keep track of the top of the stack.
  - Store an **action/goto table** that says what operations to perform on the stack and what state to enter on each input/stack pair.
  - Loop over the input, processing input/stack pairs until the automaton rejects or ends in an accepting state with all input consumed.
- If we can find a DPDA for a CFL, then we can recognize strings in that language efficiently.

If we can find a DPDA for a CFL, then we can recognize strings in that language efficiently.

Can we guarantee that we can always find a DPDA for a CFL?

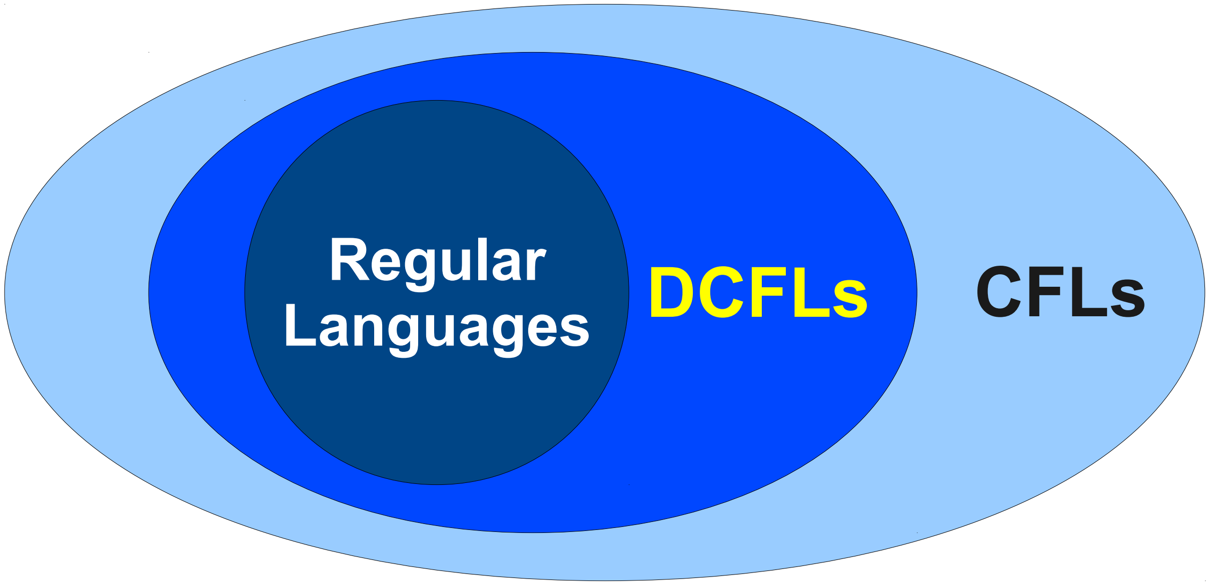
The Power of Nondeterminism

- When dealing with finite automata, there is no difference in the power of NFAs and DFAs.
- However, when dealing with PDAs, there are CFLs that can be recognized by NPDAs that **cannot** be recognized by DPDAs.
- Simple example: The language of palindromes.
  - How do you know when you've read half the string?
- NPDAs are **more powerful** than DPDAs.

Deterministic CFLs

- A context-free language L is called a **deterministic context-free language** (DCFL) if there is some DPDA that recognizes L.
- Not all CFLs are DCFLs, though many important ones are.
  - Balanced parentheses, most programming languages, etc.

Why are all regular languages DCFLs?



Summary

- Automata can be augmented with a memory storage to increase their power.
- PDAs are finite automata equipped with a stack.
- PDAs accept precisely the context-free languages:
  - Any CFG can be converted to a PDA.
  - Any PDA can be converted to a CFG.
- Deterministic PDAs are strictly weaker than nondeterministic PDAs.

Next Time

- **The Limits of CFLs**
  - A New Pumping Lemma
  - Non-Closure Properties of CFLs
- **Turing Machines**
  - An extremely powerful computing device…
  - …that is almost impossible to program.