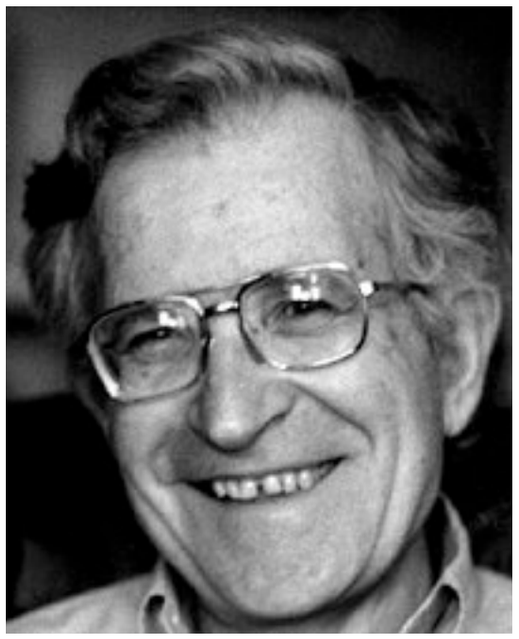
CS311 Computational Structures

Context-free Languages: Grammars and Automata

Lecture 8

Andrew Black
Andrew Tolmach

Chomsky hierarchy



In 1957, Noam Chomsky published *Syntactic Structures*, an landmark book that defined the so-called Chomsky hierarchy of languages

original name	language generated	productions:
Type-3 Grammars	Regular	$A \to \alpha$ and $A \to \alpha B$
Type-2 Grammars $\cap$	Contex-free	$A \to \gamma$
Type-1 Grammars $\cap$	Context-sensitive	$\alpha A \beta \to \alpha \gamma \beta$
Type-0 Grammars $\cap$	Recursively-enumerable	no restriction

A, B: variables, a, b terminals, α, β sequences of terminals and variables

Regular languages

- Closed under $\cup \cap * \cdot$ and $\overline{}$
- Recognizable by finite-state automata
- Denoted by Regular Expressions
- Generated by Regular Grammars

Context-free Grammars

Context-free Grammars

- More general productions than regular grammars

  S → w where w is any string of terminals and non-terminals

Context-free Grammars

- More general productions than regular grammars

  S → w where w is any string of terminals and non-terminals

- What languages do these grammars generate?

  S →(A)
  A → ε | aA | ASA

Context-free Grammars

- More general productions than regular grammars

  S → w where w is any string of terminals and non-terminals

- What languages do these grammars generate?

  S →(A)
  A → ε | aA | ASA

  S → ε | aSb

Context-free languages more general than regular languages

Context-free languages more general than regular languages

- $\{a^n b^n \mid n \geq 0\}$ is not regular

Context-free languages more general than regular languages

- $\{a^n b^n \mid n \geq 0\}$ is not regular
  - but it *is* context-free

Context-free languages more general than regular languages

- $\{a^n b^n \mid n \geq 0\}$ is not regular
  - but it *is* context-free
- Why are they called “context-free”?

Context-free languages more general than regular languages

- $\{a^n b^n \mid n \geq 0\}$ is not regular
  - but it *is* context-free
- Why are they called “context-free”?
  - Context-sensitive grammars allow more than one symbol on the lhs of productions

Context-free languages more general than regular languages

- $\{a^n b^n \mid n \geq 0\}$ is not regular
  - but it *is* context-free
- Why are they called “context-free”?
  - Context-sensitive grammars allow more than one symbol on the lhs of productions
    - xAy → x(S)y can only be applied to the non-terminal A when it is in the *context* of x and y

Context-free grammars are widely used for programming languages

- From the definition of Algol-60:

procedure_identifier::= **identifier**.
actual_parameter::= **string_literal** | **expression** | **array_identifier** | **switch_identifier** | **procedure_identifier**.
letter_string::= **letter** | **letter_string letter**.
parameter_delimiter::= "," | ")" **letter_string** ":" "(".
actual_parameter_list::= **actual_parameter** | **actual_parameter_list parameter_delimiter actual_parameter**.
actual_parameter_part::= **empty** | "(" **actual_parameter_list**} ")".
function_designator::= **procedure_identifier actual_parameter_part**.

- We say: “most programming languages are context-free”
  - This isn’t strictly true
  - … but we pretend that it is!

Example

adding_operator::= "+" | "−" .
multiplying_operator::= "×" | "/" | "÷" .
primary::= **unsigned_number** | **variable** | **function_designator** | "(" **arithmetic_expression** ")".
factor::= **primary** | **factor** | **factor power primary**.
term::= **factor** | **term multiplying_operator factor**.
simple_arithmetic_expression::= **term** | **adding_operator term** |
    simple_arithmetic_expression adding_operator term.
if_clause::= **if Boolean_expression then**.
arithmetic_expression::= **simple_arithmetic_expression** |
    if_clause simple_arithmetic_expression else arithmetic_expression.

if a < 0 then U+V else if a * b < 17 then U/V else if k <> y then V/U else 0

Example derivation in a Grammar

- Grammar: start symbol is *A*

  $A \rightarrow aAa$
  $A \rightarrow B$
  $B \rightarrow bB$
  $B \rightarrow \varepsilon$

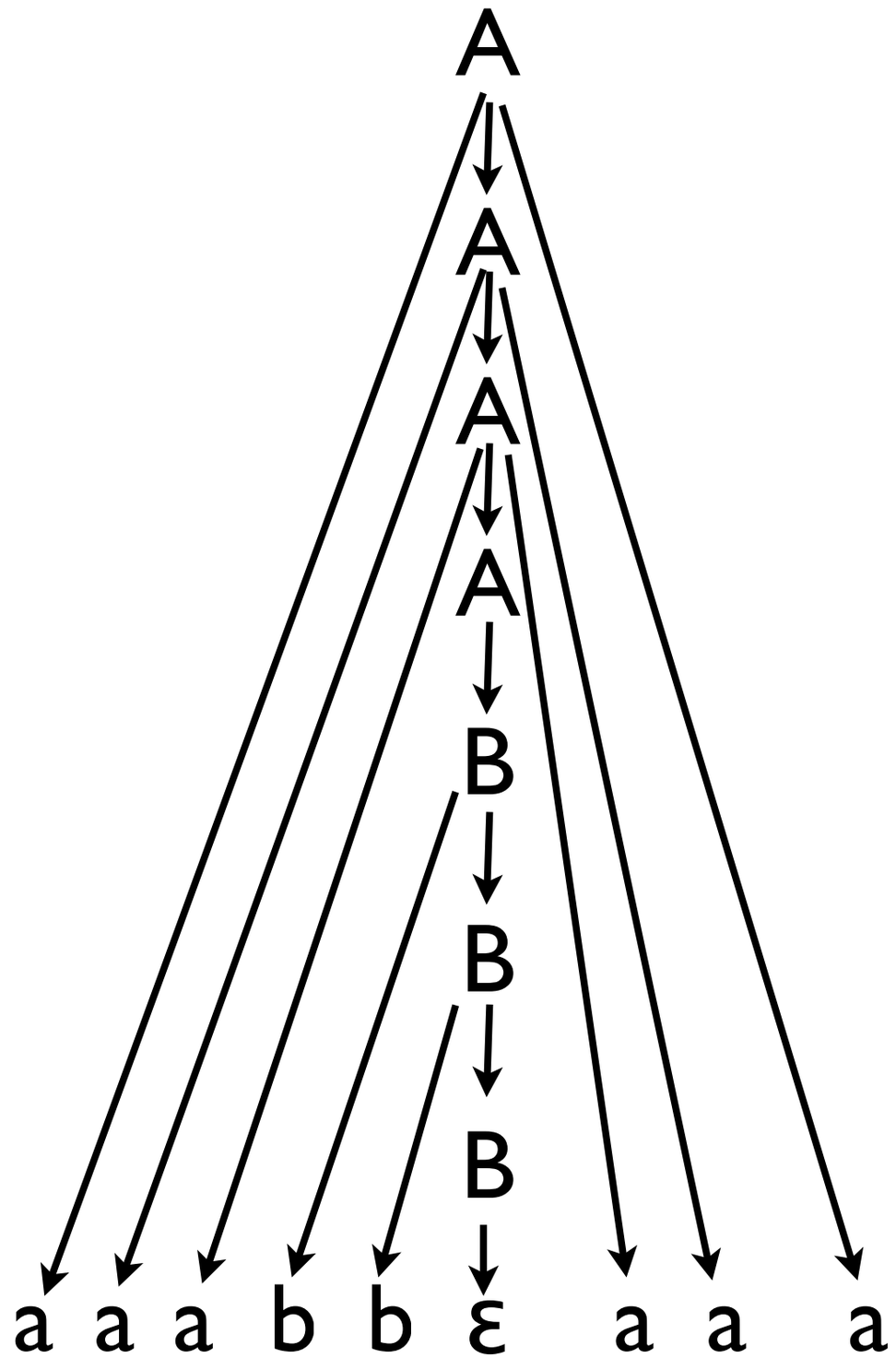
- Sample Derivation:

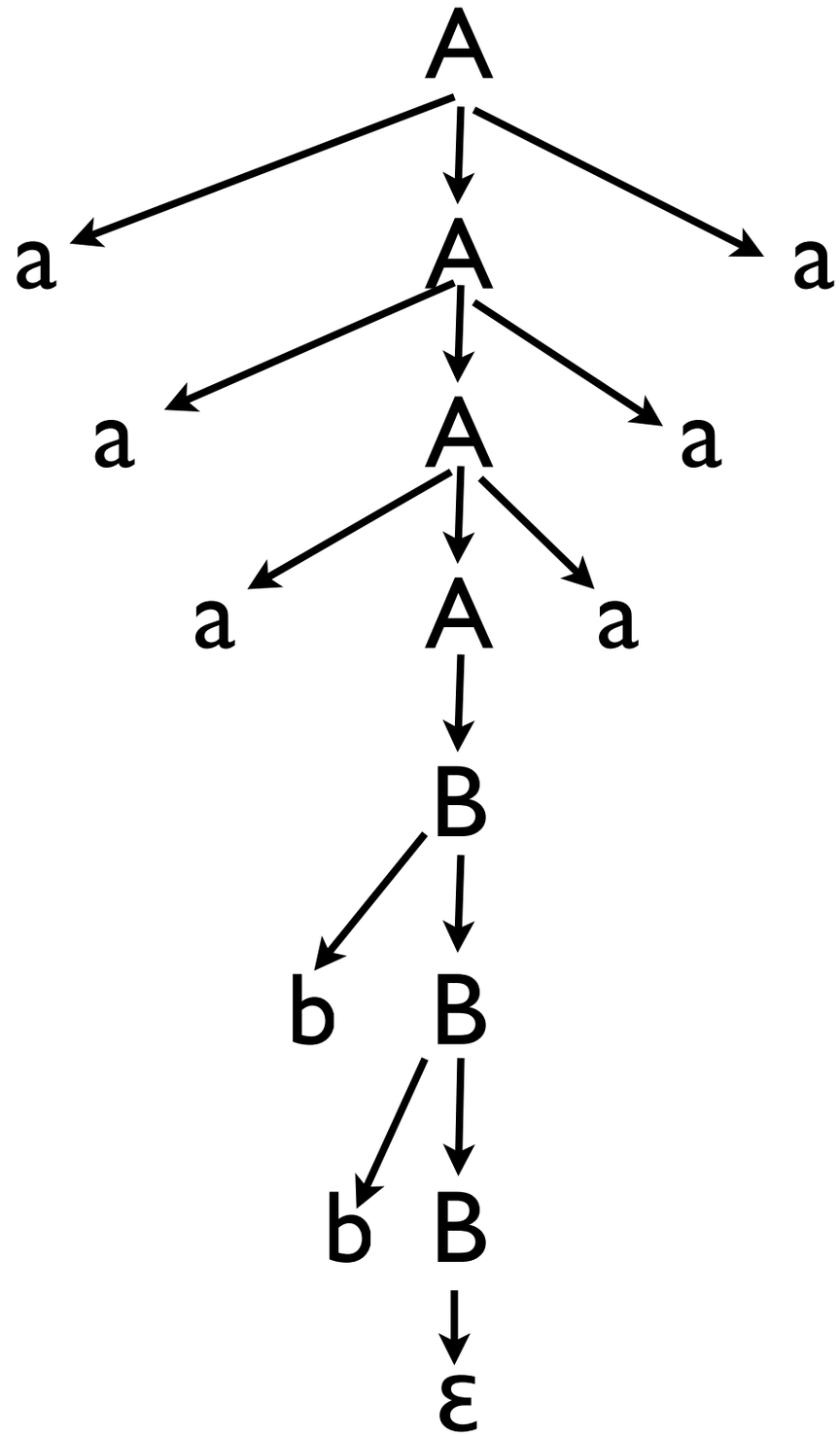
  $\underline{A} \Rightarrow a\underline{A}a \Rightarrow aa\underline{A}aa \Rightarrow aaa\underline{A}aaa \Rightarrow aaa\underline{B}aaa$

  $\Rightarrow aaab\underline{B}aaa \Rightarrow aaabb\underline{B}aaa \Rightarrow aaabbaaa$

- Language?

Derivations in Tree Form





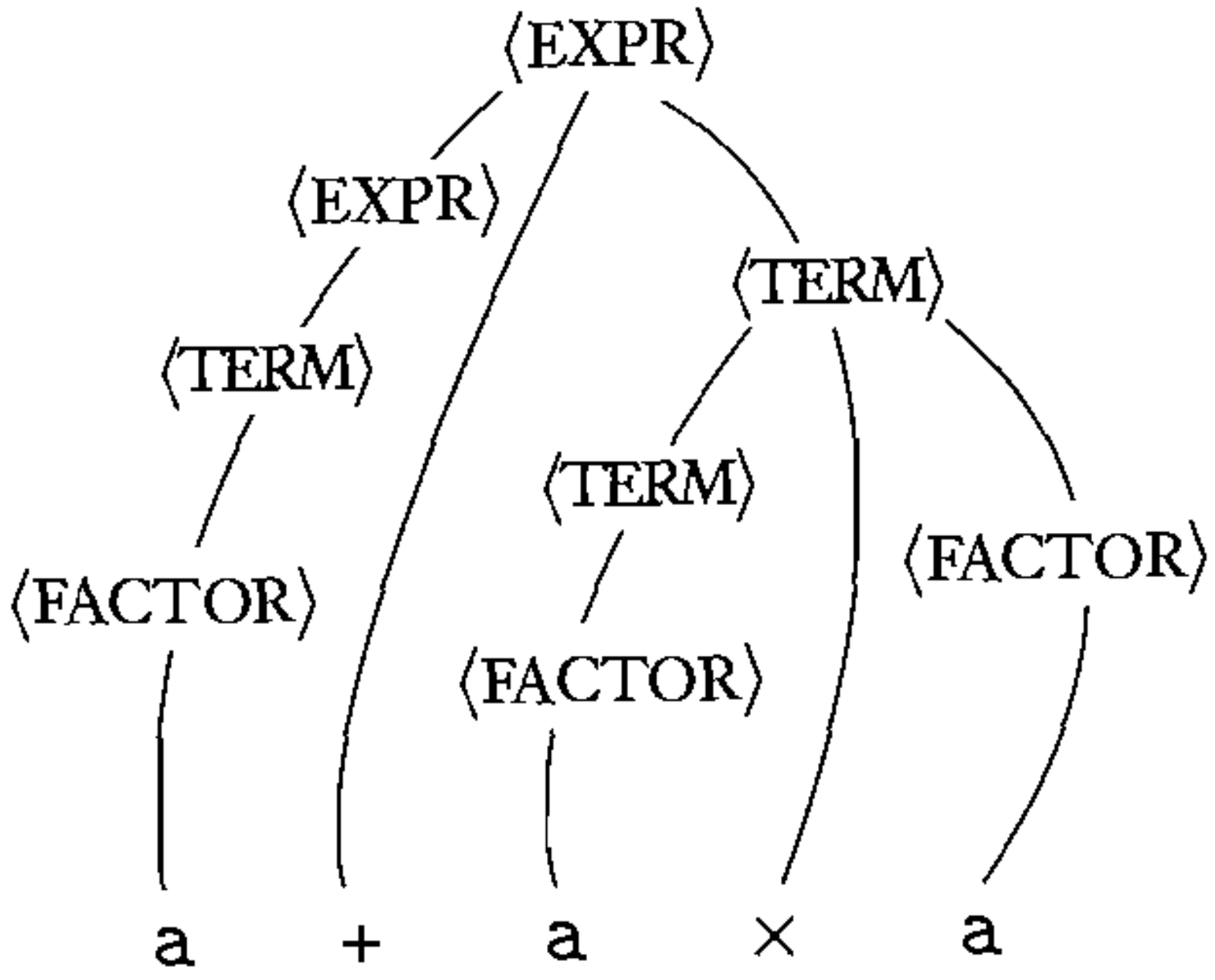
Arithmetic expressions in a programming language

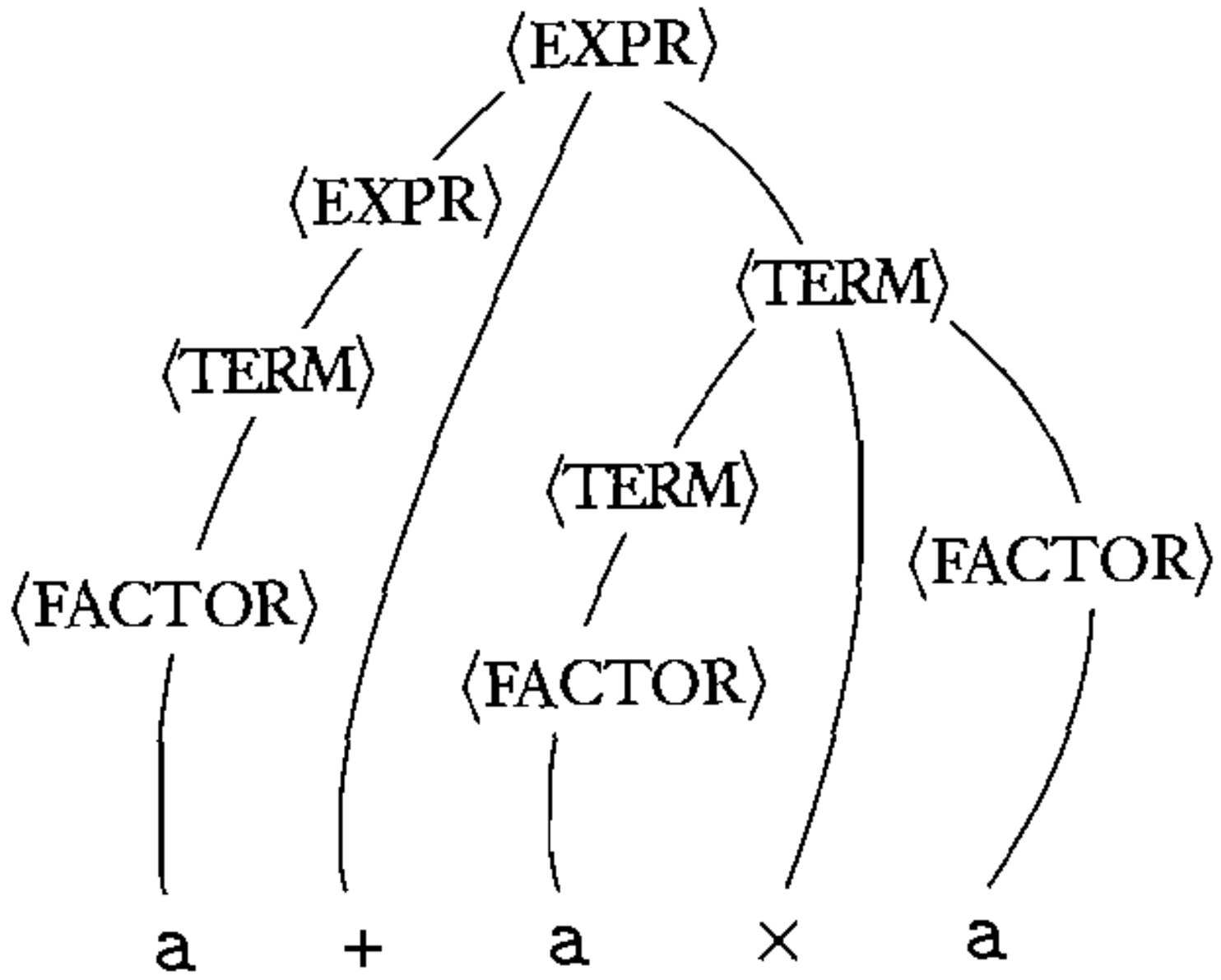
Consider grammar $G_4 = (V, \Sigma, R, \langle \text{EXPR} \rangle)$.

V is $\{\langle \text{EXPR} \rangle, \langle \text{TERM} \rangle, \langle \text{FACTOR} \rangle\}$ and Σ is $\{\texttt{a}, \texttt{+}, \texttt{x}, \texttt{(}, \texttt{)}\}$. The rules are

$$\langle \text{EXPR} \rangle \rightarrow \langle \text{EXPR} \rangle \texttt{+} \langle \text{TERM} \rangle \mid \langle \text{TERM} \rangle$$
$$\langle \text{TERM} \rangle \rightarrow \langle \text{TERM} \rangle \texttt{x} \langle \text{FACTOR} \rangle \mid \langle \text{FACTOR} \rangle$$
$$\langle \text{FACTOR} \rangle \rightarrow \texttt{(} \langle \text{EXPR} \rangle \texttt{)} \mid \texttt{a}$$

- Derive: a + a × a


⟨EXPR⟩
⟨EXPR⟩
⟨TERM⟩
⟨TERM⟩
⟨TERM⟩
⟨FACTOR⟩
⟨FACTOR⟩
⟨FACTOR⟩
a
+
a
×
a



Notice how the grammar gives the meaning a + (a×a)

Grammars in real computing

- CFG's are universally used to describe the syntax of programming languages
  - Perfectly suited to describing **recursive** syntax of expressions and statements
  - Tools like compilers must **parse** programs; parsers can be generated automatically from CFG's
  - Real languages usually have a few non-CF bits
- CFG's are also used in XML DTD's

Formal definition of CFG

- A Context-free grammar is a 4-tuple (V, Σ, R, S) where

1. V is a finite set called the **variables** (non-terminals)
2. Σ is a finite set (disjoint from V) called the **terminals**,
3. R is a finite set of **rules**, where each rule maps a variable to a string $s \in (V \cup \Sigma)^*$
4. $S \in V$ is the start symbol

Definition of Derivation

- Let u, v and w be strings in $(V \cup \Sigma)^*$, and let $A \rightarrow w$ be a rule in R,
- then $uAv \Rightarrow uwv$ (read: uAv **yields** uwv)
- We say that $u \overset{*}{\Rightarrow} v$ (read: u **derives** v) if

  u = v or there is a sequence $u_1, u_2, \ldots u_k$, $k \geq 0$, s.t. $u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow u_k \Rightarrow v$

- The **language** of the grammar is

  $\{w \in \Sigma^* \mid S \overset{*}{\Rightarrow} w\}$

Derivations ⇔ Parse Trees

- Each derivation can be viewed as a **parse tree** with variables at the internal nodes and terminals at the leaves
  - Start symbol is at the root
  - Each yield step $uAv \Rightarrow uwv$ where $w=w_1w_2...w_n$ corresponds to a node labeled A with children $w_1,w_2,...,w_n$.
  - The final result in Σ^* can be seen by reading the leaves left-to-right

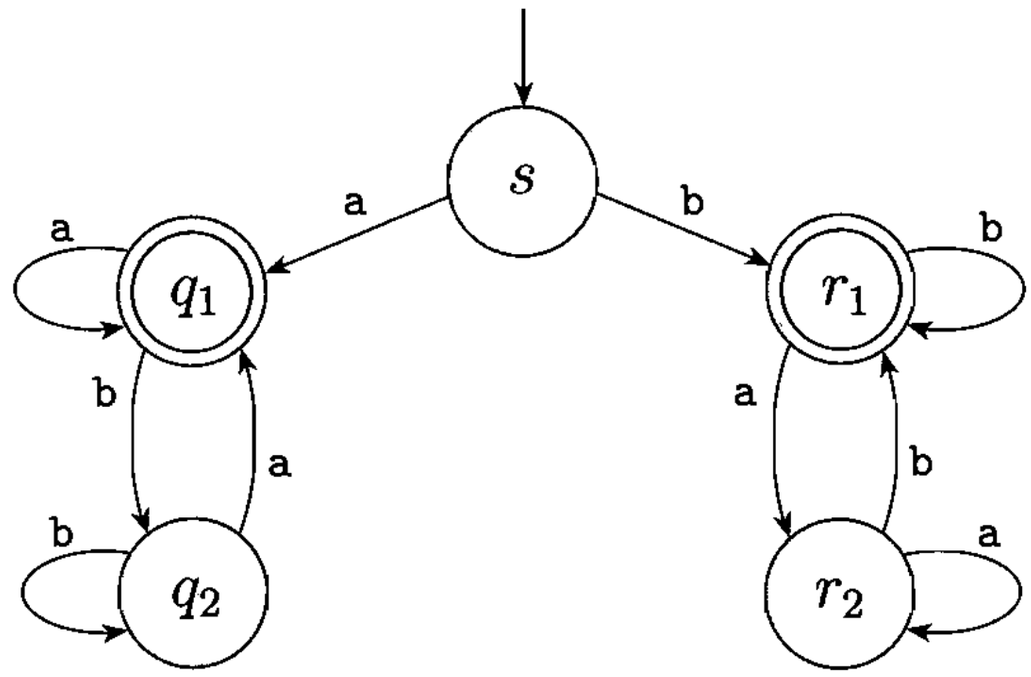
Simple CFG examples

- Find grammars for:
  - L = {w ∈ {a,b}* l w begins and ends with the same symbol}
  - L = {w ∈ {a,b}* l w contains an odd number of a's}
  - $\mathscr{L}\,[(\varepsilon + 1)(01)^*(\varepsilon + 0)]$
- Draw example derivations

All regular languages have context free grammars

- Proof:
  - Regular language is accepted by an NFA.
  - We can generate a regular grammar from the NFA (Lecture 6, Hein Alg. 11.11)
  - Any regular grammar is also a CFG. (Immediate from the definition of the grammars).

Example



- $S \rightarrow aQ_1$ $S \rightarrow bR_1$
- $Q_1 \rightarrow aQ_1$ $Q_1 \rightarrow bQ_2$
- $Q_2 \rightarrow aQ_1$ $Q_2 \rightarrow bQ_2$
- $R_1 \rightarrow aR_2$ $R_1 \rightarrow bR_1$
- $R_2 \rightarrow aR_2$ $R_2 \rightarrow bR_1$
- $Q_1 \rightarrow \varepsilon$ $R_1 \rightarrow \varepsilon$
  - Resulting grammar may be quite different from one we designed by hand.

Some CFG's generate non-regular languages

- Find grammars for the following languages
  - $L = \{a^n b^m a^n \mid a,b \geq 0\}$
  - $L = \{w \in \{a,b\}^* \mid w$ contains equal numbers of a's and b's$\}$
  - $L = \{ww^R \mid w \in \{a,b\}^*\}$
- Draw example derivations

Ambiguity

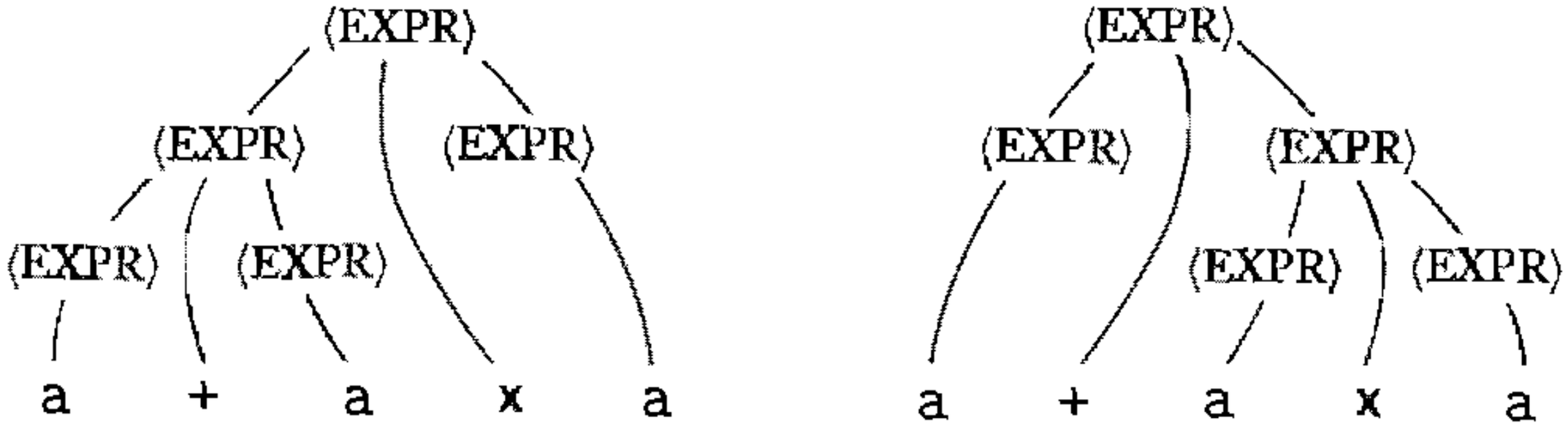
- A grammar in which the same string can be given more than one parse *tree* is **ambiguous**.

- Example: another grammar for arithmetic expressions

‹EXPR› → ‹EXPR› + ‹EXPR› |
‹EXPR› × ‹EXPR› |
(‹EXPR›)
| a

- Derive: a + a × a

- This grammar is ambiguous: there are two *different* parse trees for a + a × a



- Ambiguity is a bad thing if we're interested in the structure of the parse
  - Ambiguity doesn't matter if we're interested only in *defining* a language.

Leftmost Derivations

- In general, in any step of a derivation, there might be several variables that can be reduced by rules of the grammar.
- In a leftmost derivation, we choose to always reduce the leftmost variable.

- Example: given grammar $S \rightarrow aSb \mid SS \mid \varepsilon$
  - A left-most derivation:

    $\underline{S} \Rightarrow a\underline{S}b \Rightarrow a\underline{S}Sb \Rightarrow aa\underline{S}bSb \Rightarrow aab\underline{S}b \Rightarrow aabb$

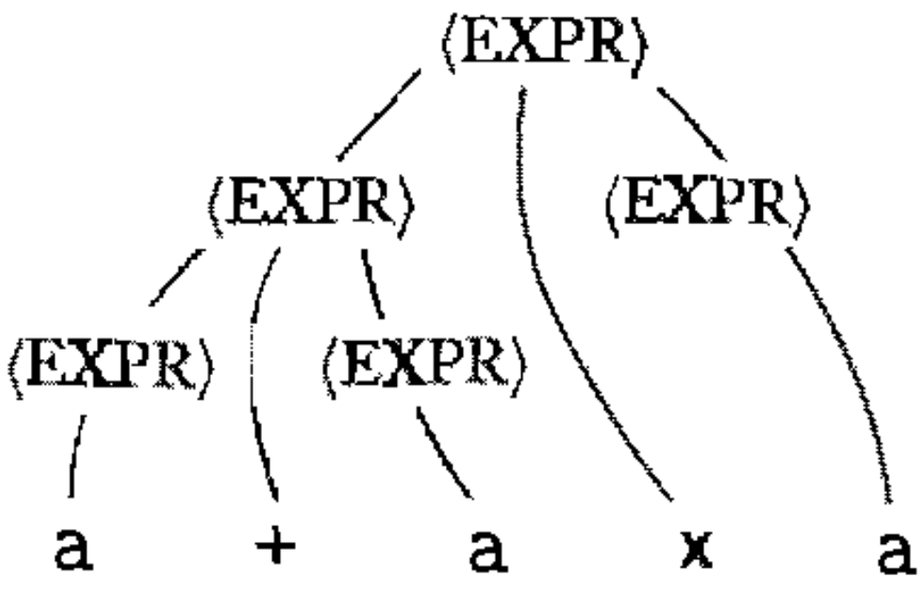
  - A non-left-most derivation:

    $\underline{S} \Rightarrow a\underline{S}b \Rightarrow aS\underline{S}b \Rightarrow a\underline{S}b \Rightarrow aa\underline{S}bb \Rightarrow aabb$

Ambiguity *via* left-most derivations

- Every parse tree corresponds to a unique left-most derivation
- So if a grammar has more than one left-most derivation for some string, the grammar is ambiguous
  - Note: merely having two derivations (not necessarily left-most) for one string is **not** enough to show ambiguity

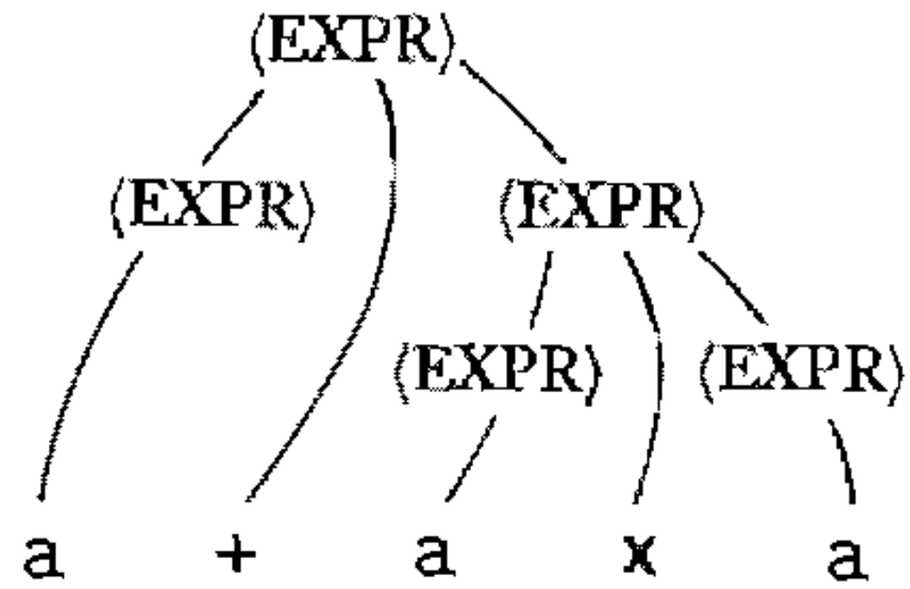
Ambiguity



$\underline{E} \Rightarrow \underline{E} \times E \Rightarrow \underline{E} + E \times E$

$\Rightarrow a + \underline{E} \times E \Rightarrow a + a \times \underline{E}$

$\Rightarrow a + a \times a$



$\underline{E} \Rightarrow \underline{E} + E \Rightarrow a + \underline{E} \Rightarrow$

$a + \underline{E} \times E \Rightarrow a + a \times \underline{E} \Rightarrow$

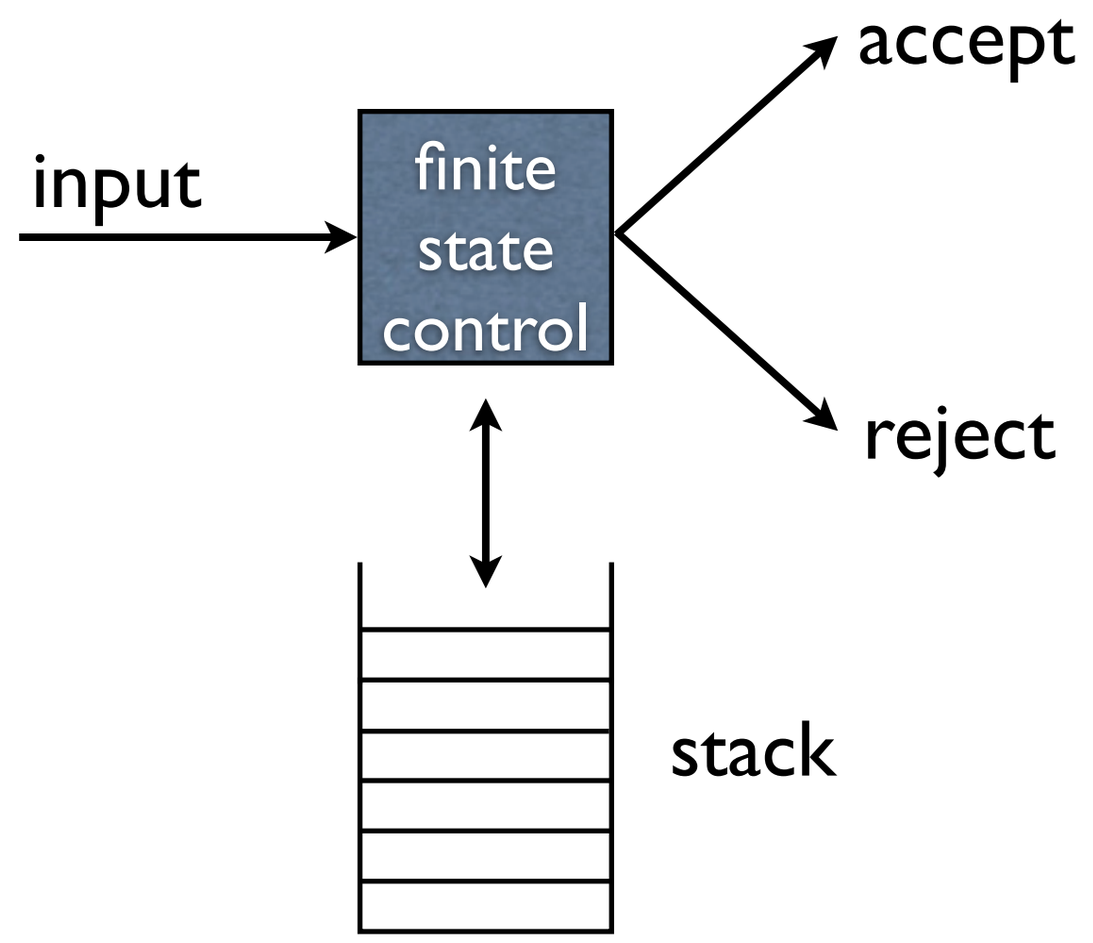
$a + a \times a$

Context-free languages

- Closed under $\cup$, $*$ and $\cdot$, and under $\cap$ with a regular language
  - How do we prove these properties?
- *Not* closed under intersection, complement or difference
- Recognizable by pushdown automata
  - A pushdown automaton is a generalization of a finite-state automaton

Pushdown Automata

- Why can't a FSA recognize $a^n b^n$?
  - "storage" is finite
- How can we fix the problem?
  - add unbounded storage
- What's the simplest kind of unbounded storage
  - a pushdown stack



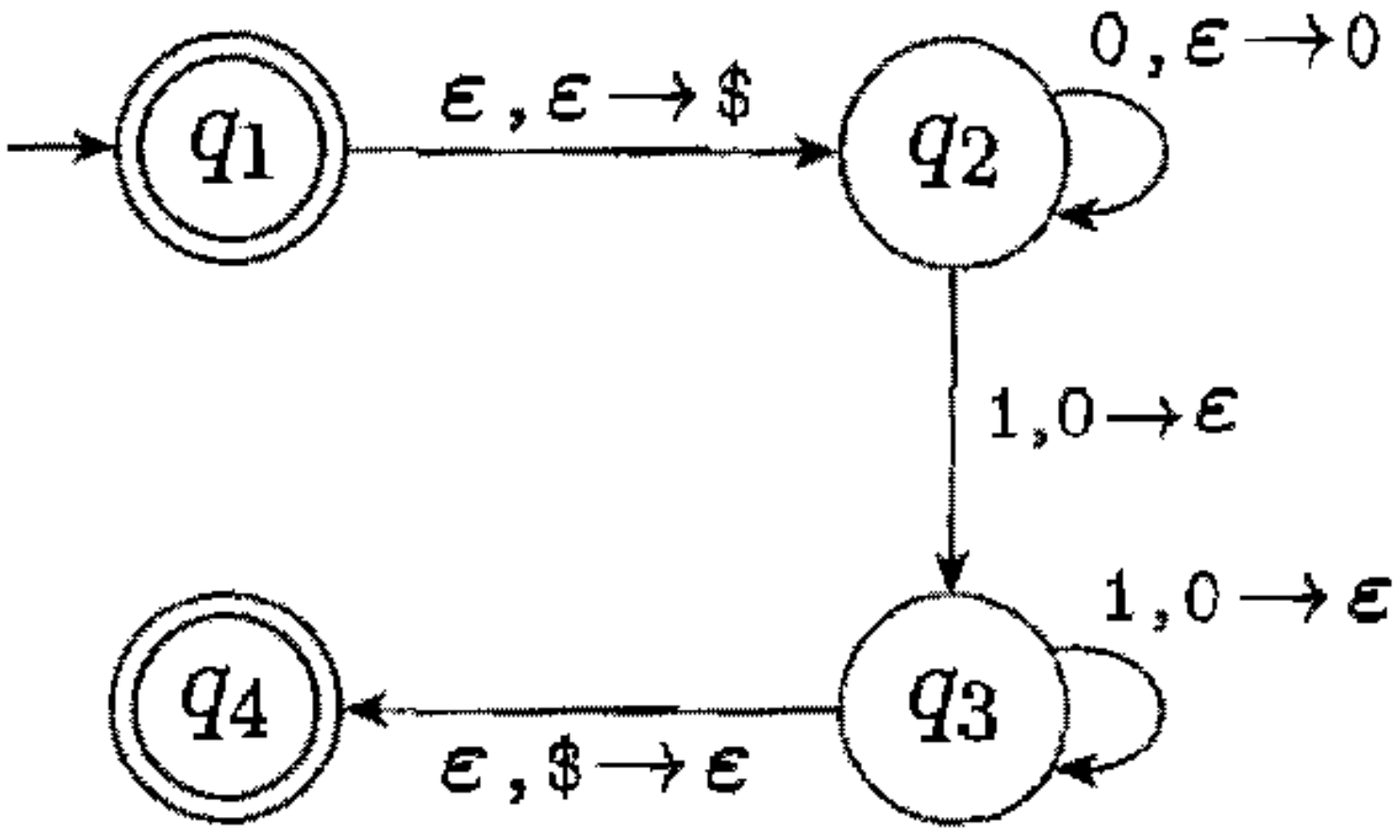
History

- PDAs independently invented by Oettinger [1961] and Schutzenberger [1963]
- Equivalence between PDAs and CFG known to Chomsky in 1961; first published by Evey [1963].

Executing a PDA

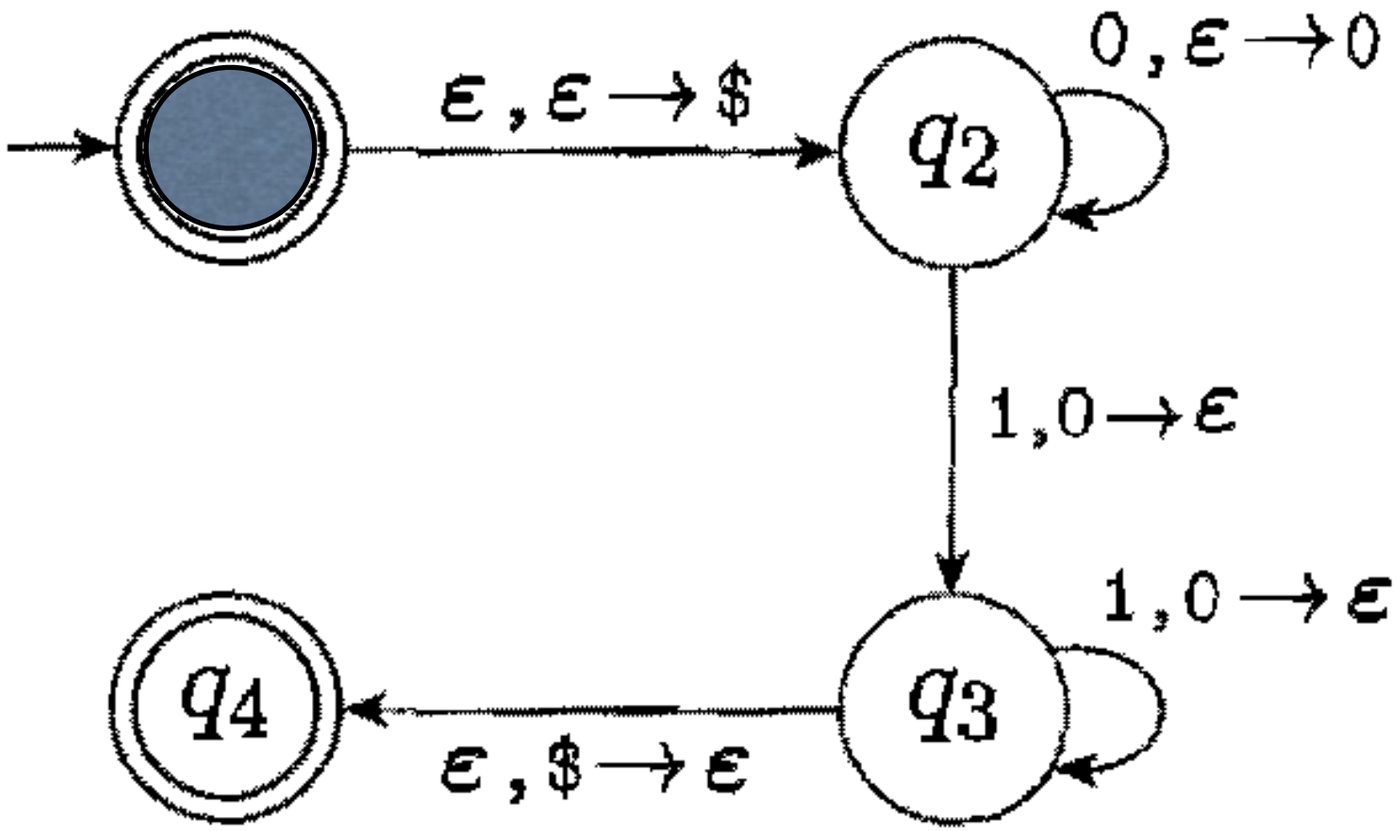
- The behavior of a PDA at any point depends on ⟨state,stack,unread input⟩
- PDA begins in start state with stated symbol on the stack
- On each step it optionally reads an input character, pops a stack symbol, and non-deterministically chooses a new state and optionally pushes one or more stack symbols
- PDA accepts if it is in a final state and there is no more input.

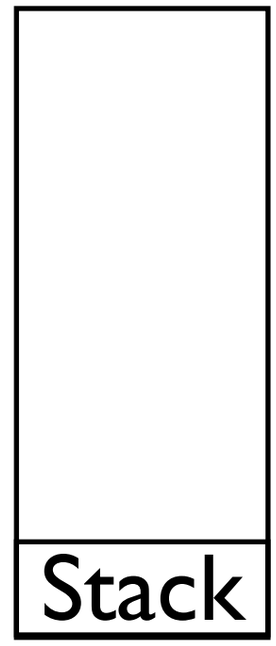
Example PDA


q_1
$\varepsilon, \varepsilon \to \$$
q_2
$0, \varepsilon \to 0$
$1, 0 \to \varepsilon$
q_3
$1, 0 \to \varepsilon$
q_4
$\varepsilon, \$ \to \varepsilon$

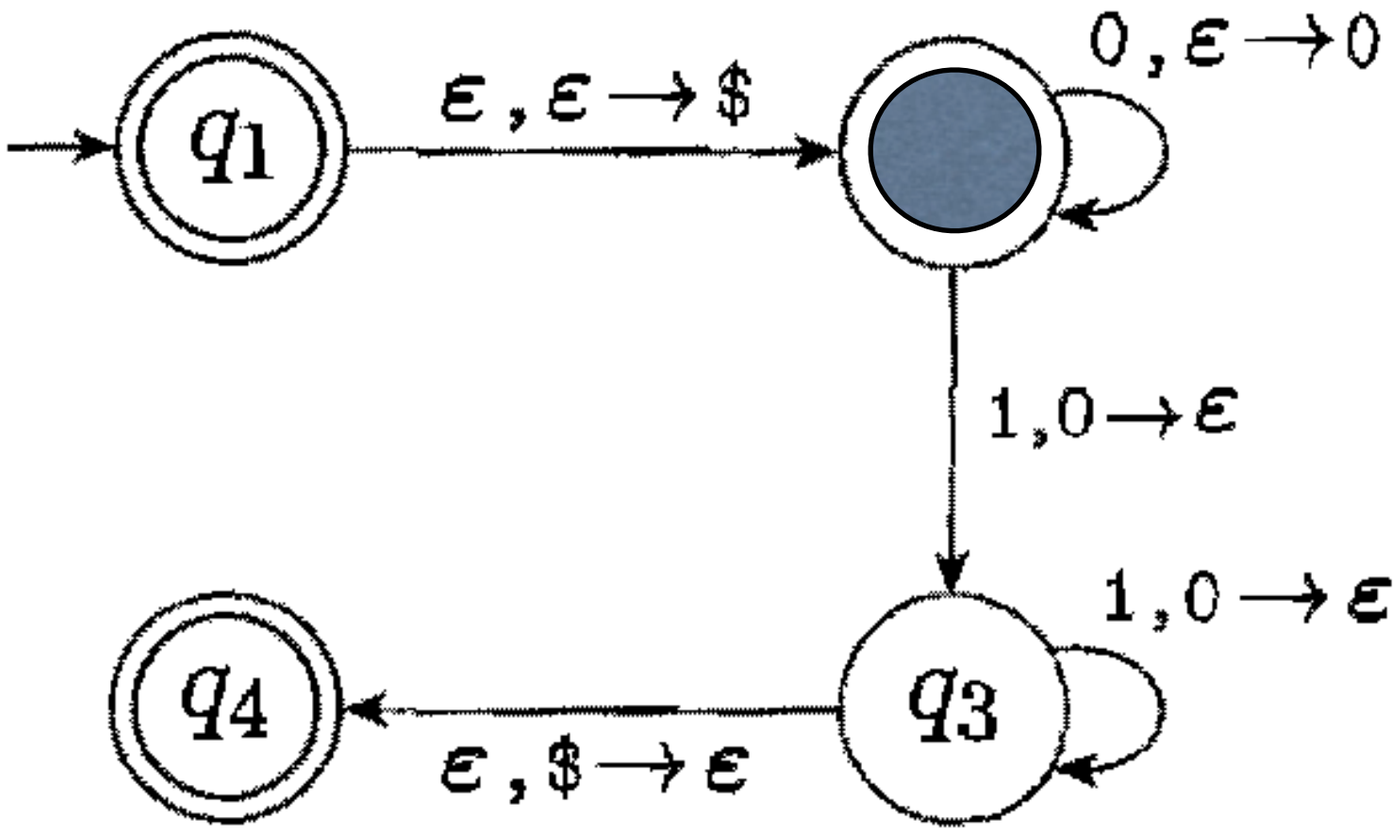
Example Execution: 0011

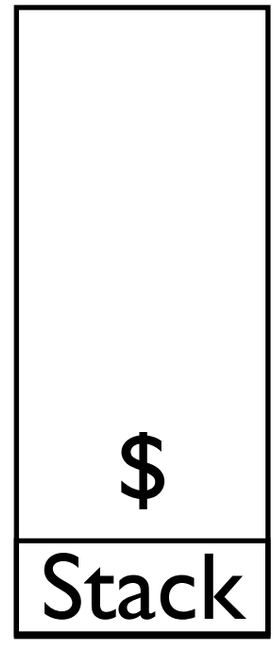
Begin in initial state
with empty stack



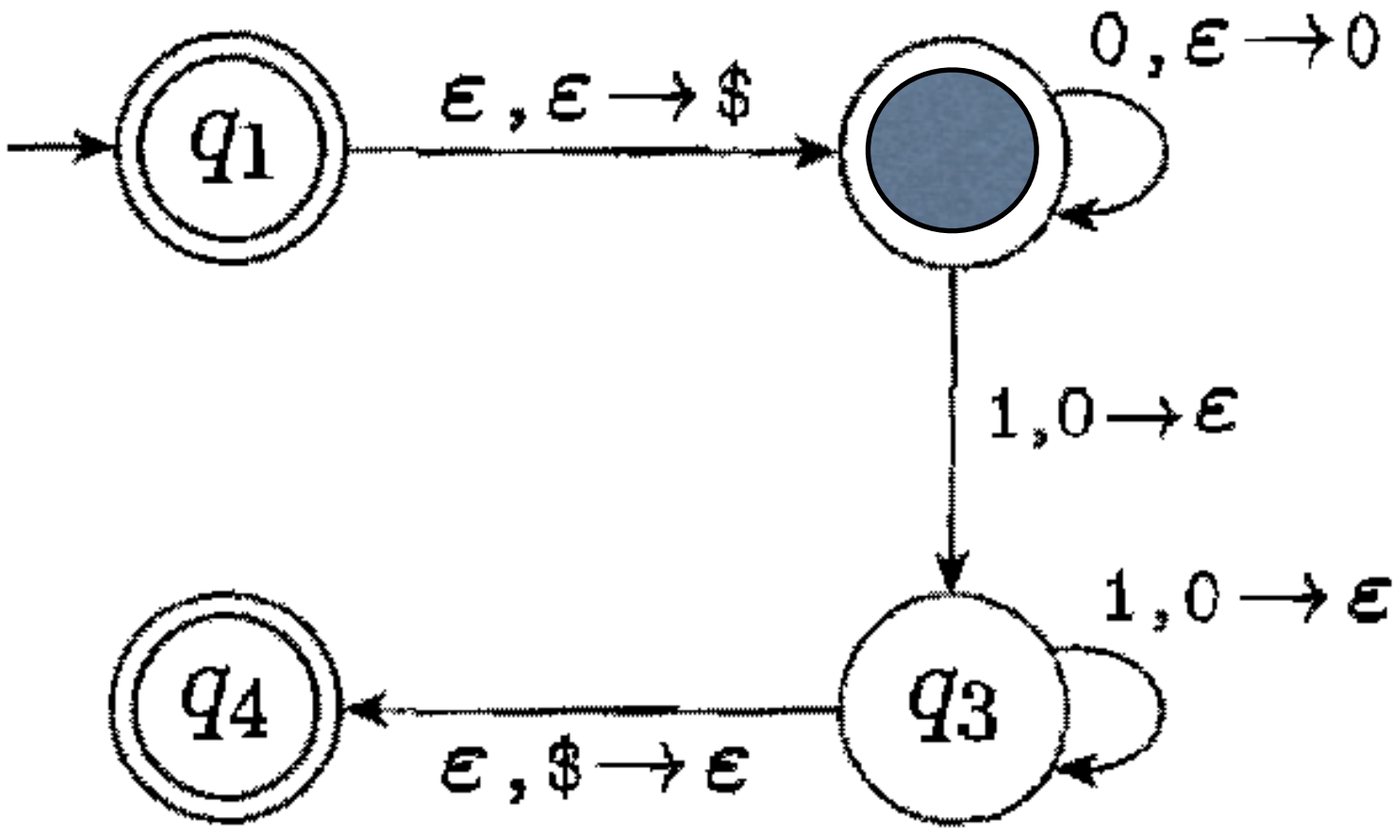


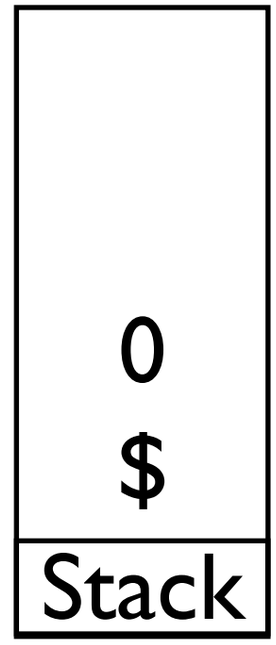
Example Execution: 0011



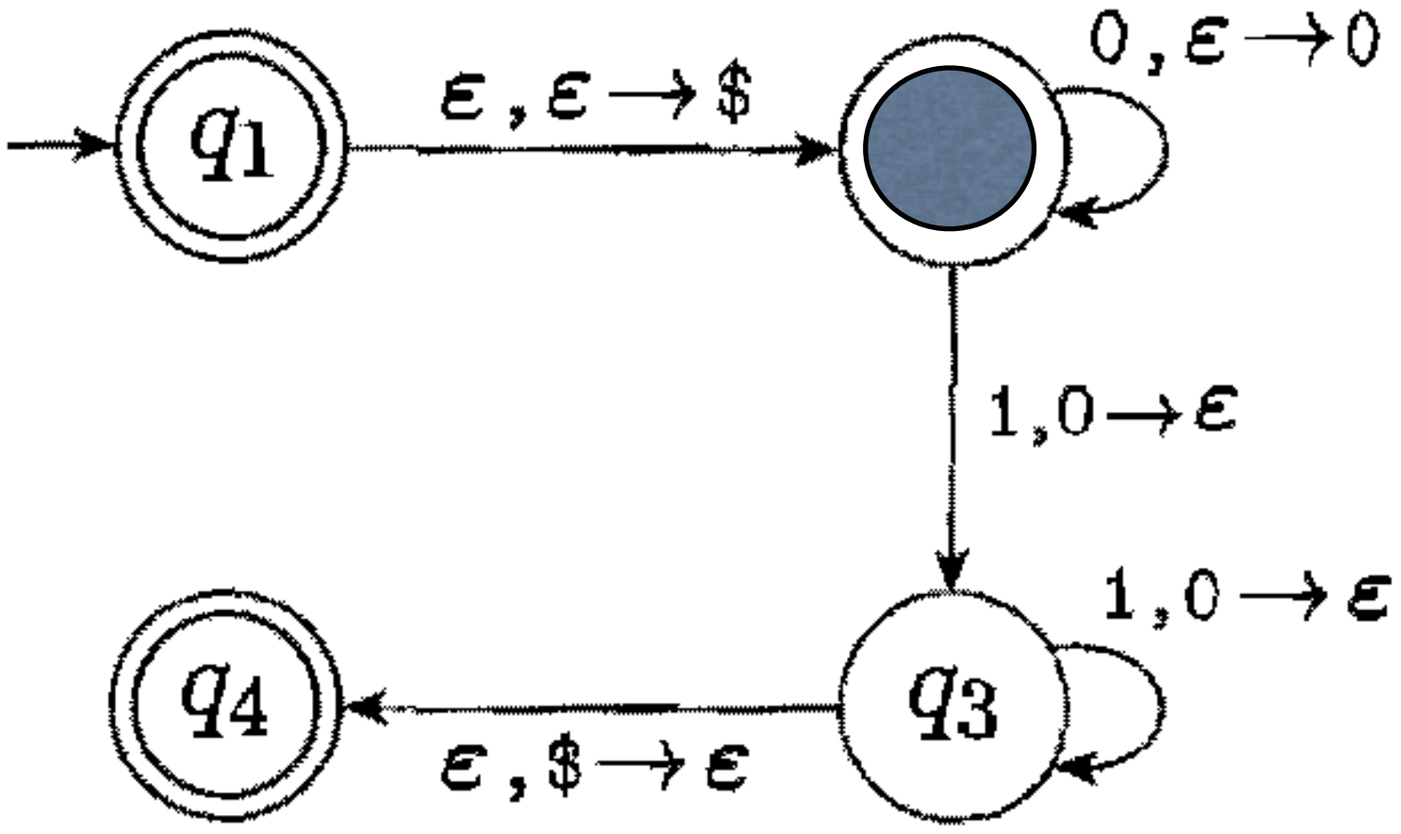


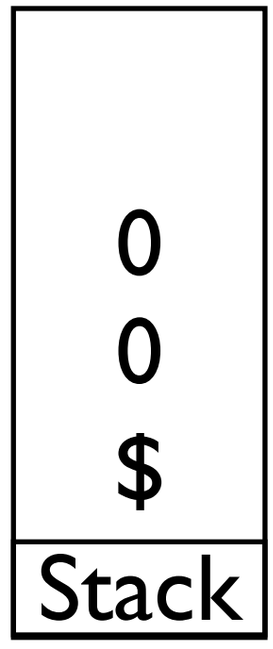
Example Execution: 0011



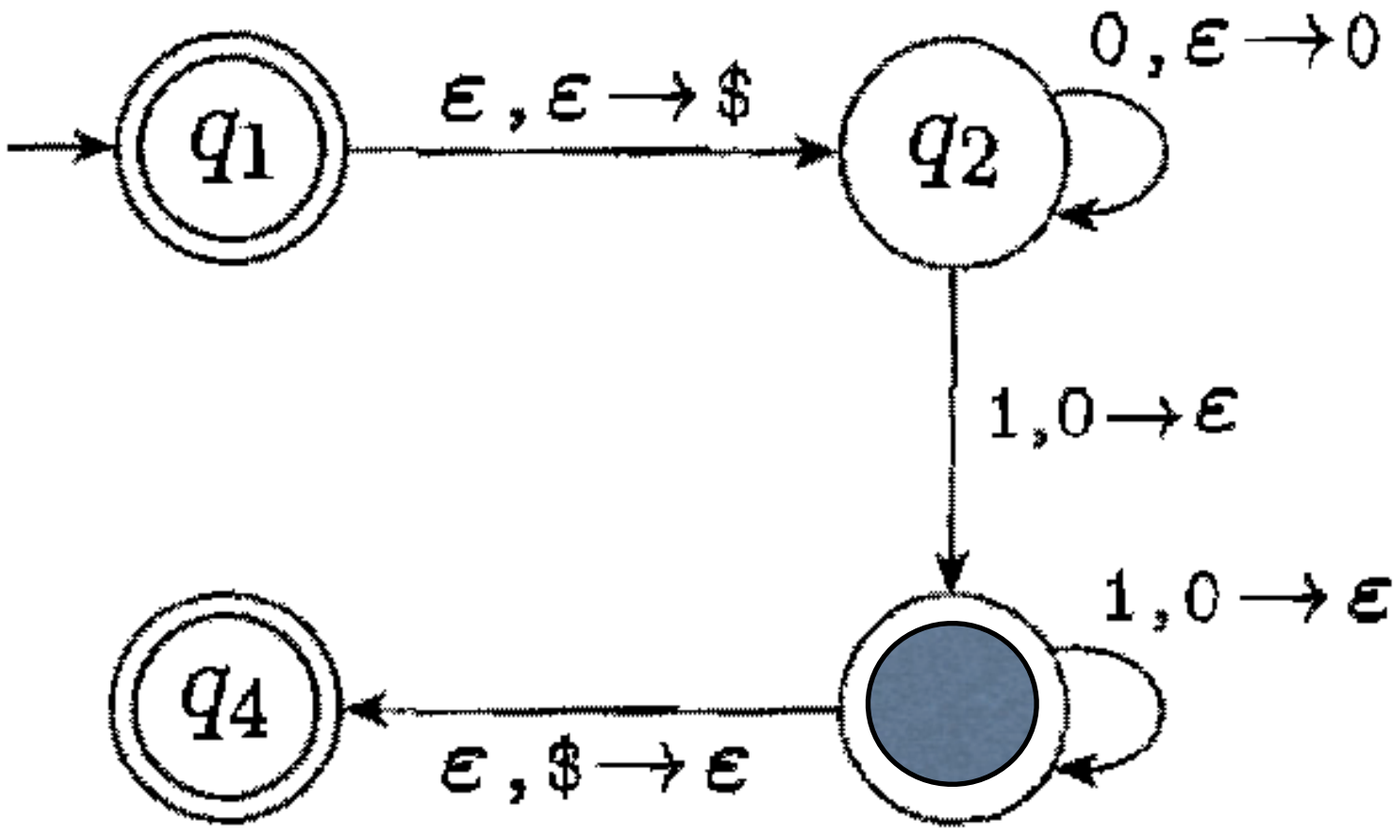


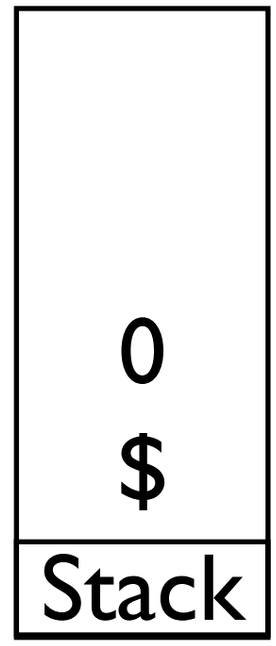
Example Execution: 0011



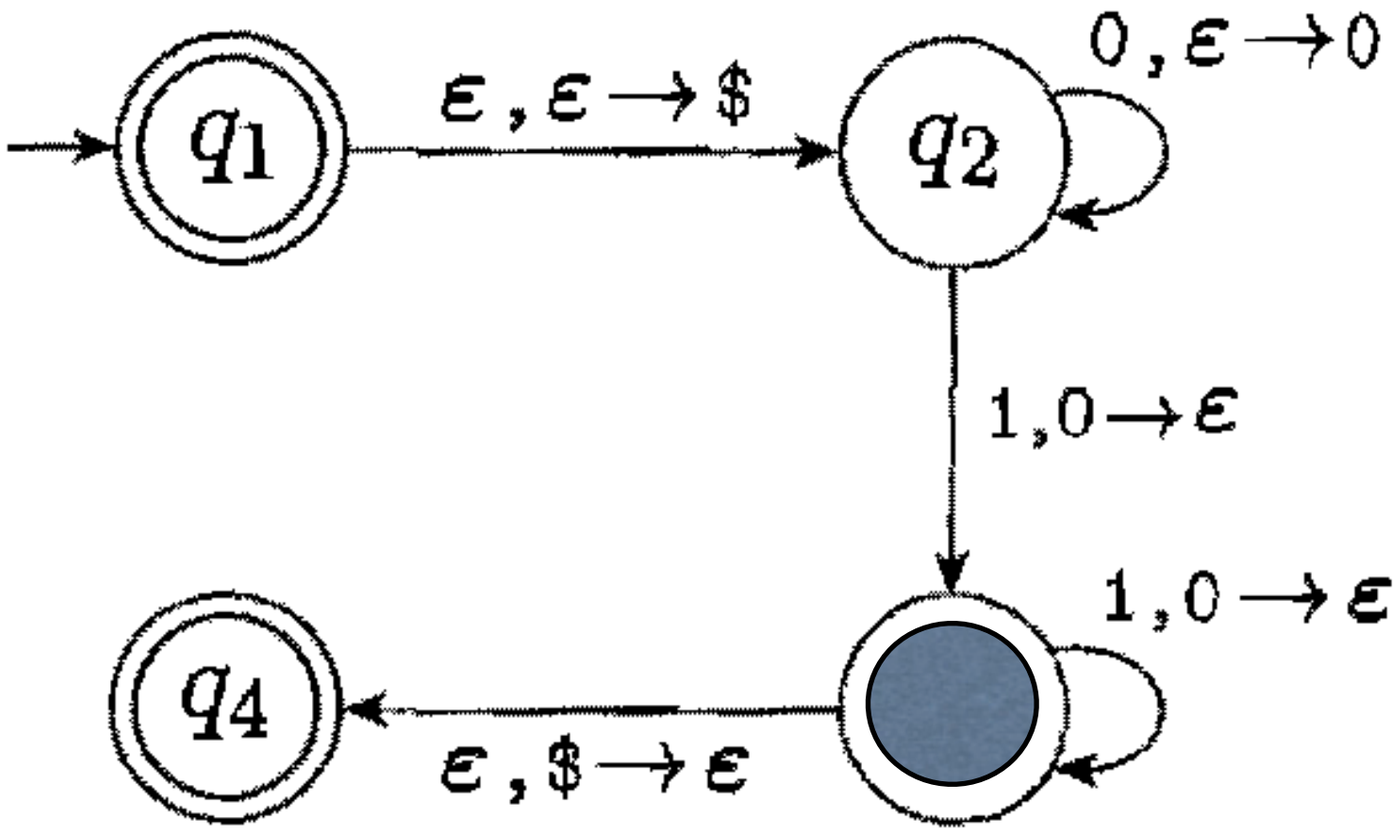


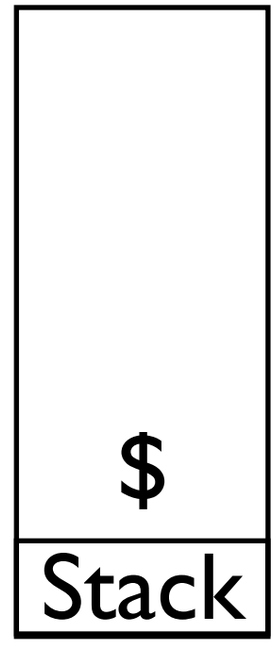
Example Execution: 0011



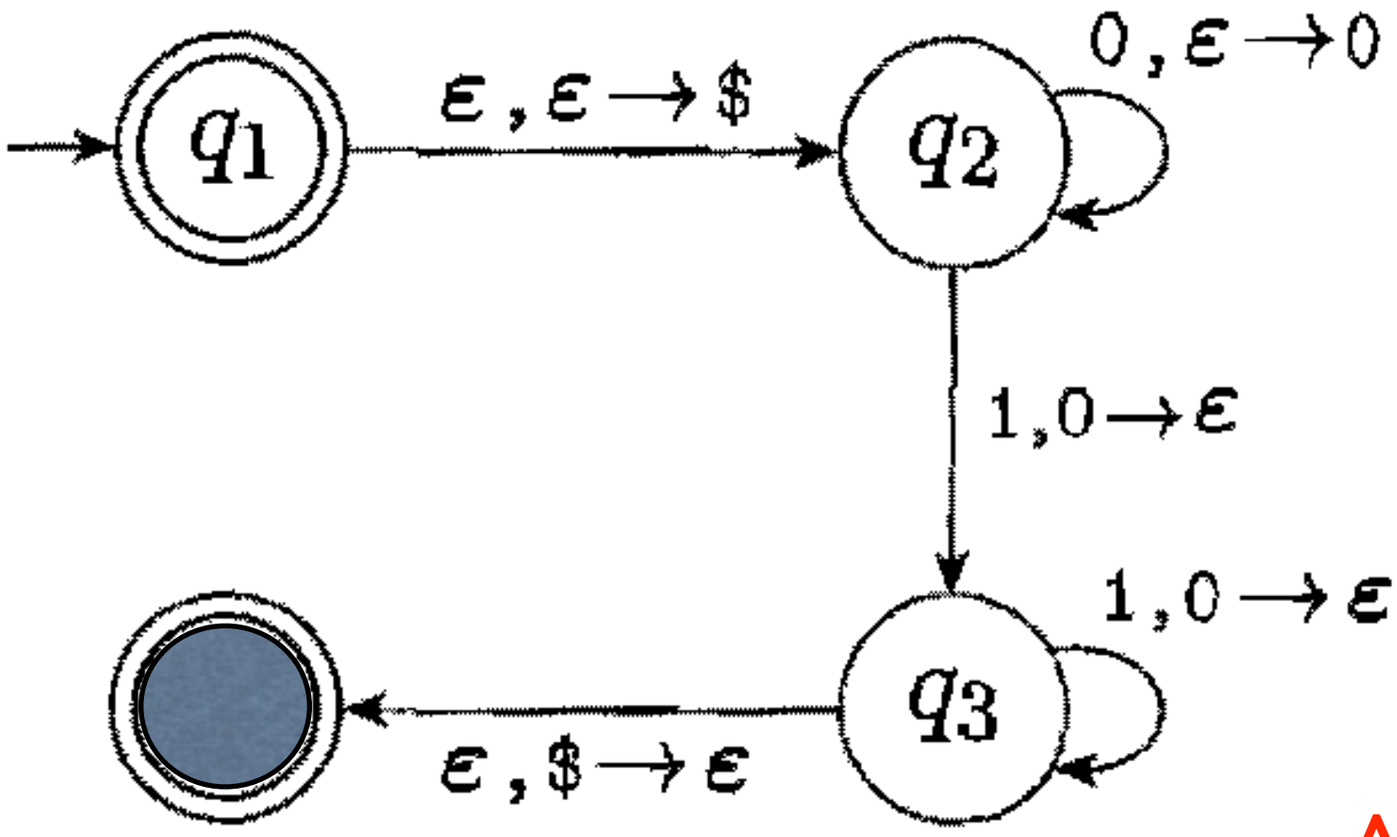


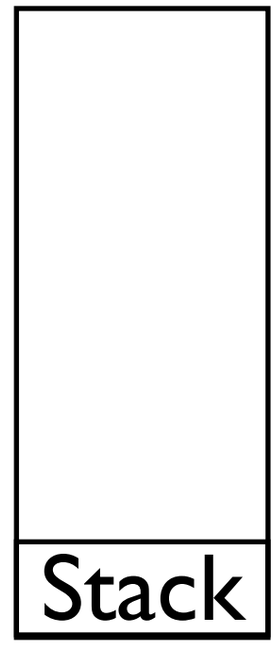
Example Execution: 0011





Example Execution: 0011





Accept!

PDA's can keep count!

- This PDA can recognize $\{0^n1^n \mid n \geq 0\}$ by
  - First, pushing a symbol on the stack for each 0
  - Then, popping a symbol off the stack for each 1
  - Accepting iff the stack is empty when the end of input is reached (and not before)
- The size of the stack is unbounded.
  - That is, no matter how big the stack grows, it is always possible to push another symbol on it.
  - So PDA's can use the stack to count arbitrarily high

Pushdown Automata (PDA)

- A pushdown automaton M is defined as a 7-tuple: $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, where:
  - Q is a set of states, $q_0 \in Q$ is the start state
  - Σ is the input alphabet,
  - Γ is the stack alphabet, $Z_0 \in \Gamma$ is the initial stack symbol
  - $\delta : (Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon) \rightarrow \mathcal{P}\{Q \times \Gamma^*\}$ is the transition function
  - $F \subseteq Q$ is a set of final states, and
  - $X_\varepsilon = X \cup \{\varepsilon\}$, the set X augmented with ε

Executing a PDA

- The *configuration* (or PDA + Hopcroft's *instantaneous description*, ID) of a PDA consists of
  1. The PDA,
  2. The current state of the PDA,
  3. the remaining input, and
  4. The contents of its stack.

Transitions

- We defined $\delta : (Q \times A_\varepsilon \times \Gamma_\varepsilon) \rightarrow \mathcal{P}\{Q \times \Gamma^*\}$
  - The transitions $\delta(q, a, \gamma)$ are applicable iff
    - q is the current state,
    - $a = \varepsilon$, or the next character on the input tape is a , and
    - $\gamma = \varepsilon$, or the top of the stack is γ
  - If you select a transition (q', ω) , then
    - The new state is q'
    - if $\gamma \neq \varepsilon$, γ is popped off of the stack, and
    - the (possibly empty) sequence of symbols ω is pushed onto the stack

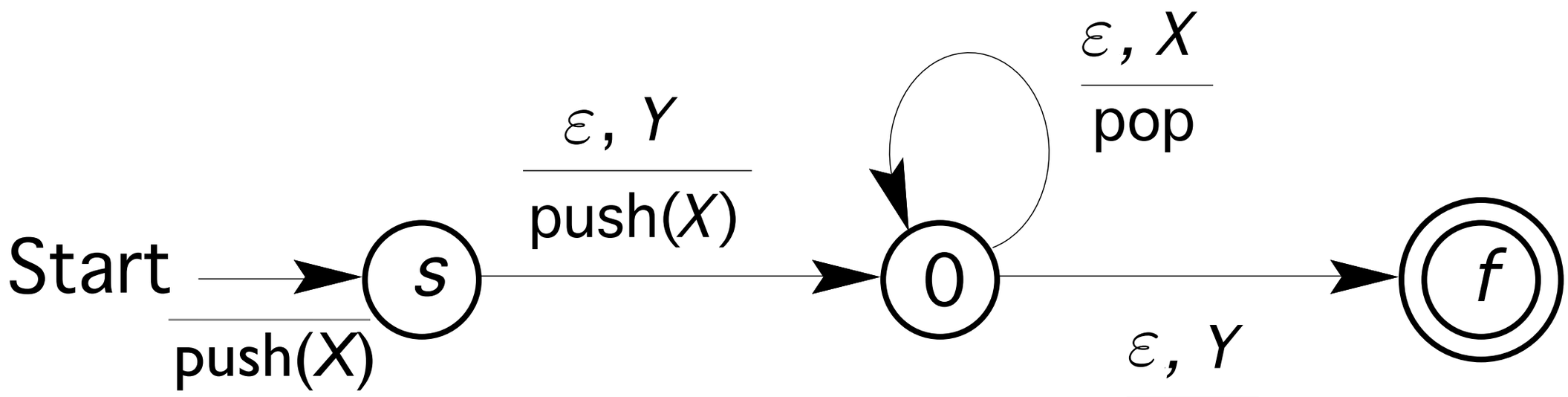
PDAs are non-deterministic

For two reasons:

- The transition function δ answers a *set*
- We have the choice of taking an ε–transition, or reading an input symbol

Trivial Example

Hein Example 12.4



- $\delta(s, \varepsilon, Y) = \{(0, XY)\}$
- $\delta(0, \varepsilon, X) = \{(0, \varepsilon)\}$
- $\delta(0, \varepsilon, Y) = \{(f, Y)\}$

Accepting a String

- There are two ways in which a PDA can *accept* a string:
  1. final state is in the set F (implies F non-empty)
  2. if F is empty (there are no final states), then the PDA accepts when the stack is empty
- The previous example accepts by …

Acceptance by Final State

A run of PDA $M = (Q, A, \Gamma, \delta, q_0, \gamma_0, F)$ is a sequence

$$(q_0, \gamma_0) \stackrel{a_0}{\to} (q_1, s_1) \stackrel{a_1}{\to} \cdots \stackrel{a_{n-1}}{\to} (q_n, s_n)$$

with $q_0, \ldots, q_n \in Q$, $s_1, \ldots, s_n \in \Gamma^*$, and $a_0, \ldots, a_{n-1} \in A$ such that:

for all $i \in [0 \,..\, n-1]$. $(q_{i+1}, \gamma_{i+1}) \in \delta(q_i, a_i, \gamma_i)$ and

$s_i = \gamma_i t_i$ and $s_{i+1} = \gamma_{i+1} t_i$ for some $t_i \in \Gamma^*$, and

$w = a_0 a_1 a_2 \ldots a_{n-1}$ is the input.

The run accepts w if $q_n \in F$.

The language of M, $L(M)$ is given by

$$L(M) = \{w \in A^* | \; w \text{ is accepted by some run of } M\}$$

Acceptance by Empty Stack

- There is an alternative definition of acceptance:
  - If the set of final states is empty, then
  - the PDA accepts an input string if it can read all of the characters of the string and end up in a state in which the stack is empty.

Acceptance by Empty Stack

A run of PDA $M = (Q, A, \Gamma, \delta, q_0, \gamma_0, \emptyset)$ is a sequence

$$(q_0, \gamma_0) \stackrel{a_0}{\to} (q_1, s_1) \stackrel{a_1}{\to} \cdots \stackrel{a_{n-1}}{\to} (q_n, s_n)$$

with $q_0, \ldots, q_n \in Q$, $s_1, \ldots, s_n \in \Gamma^*$, and $a_0, \ldots, a_{n-1} \in A$ such that:

for all $i \in [0 \,..\, n-1]$. $(q_{i+1}, \gamma_{i+1}) \in \delta(q_i, a_i, \gamma_i)$ and

$s_i = \gamma_i t_i$ and $s_{i+1} = \gamma_{i+1} t_i$ for some $t_i \in \Gamma^*$, and

$w = a_0 a_1 a_2 \ldots a_{n-1}$ is the input.

The run accepts w if $s_n = \varepsilon$.

The language of M, $L(M)$ is given by

$$L(M) = \{w \in A^* | \; w \text{ is accepted by some run of } M\}$$

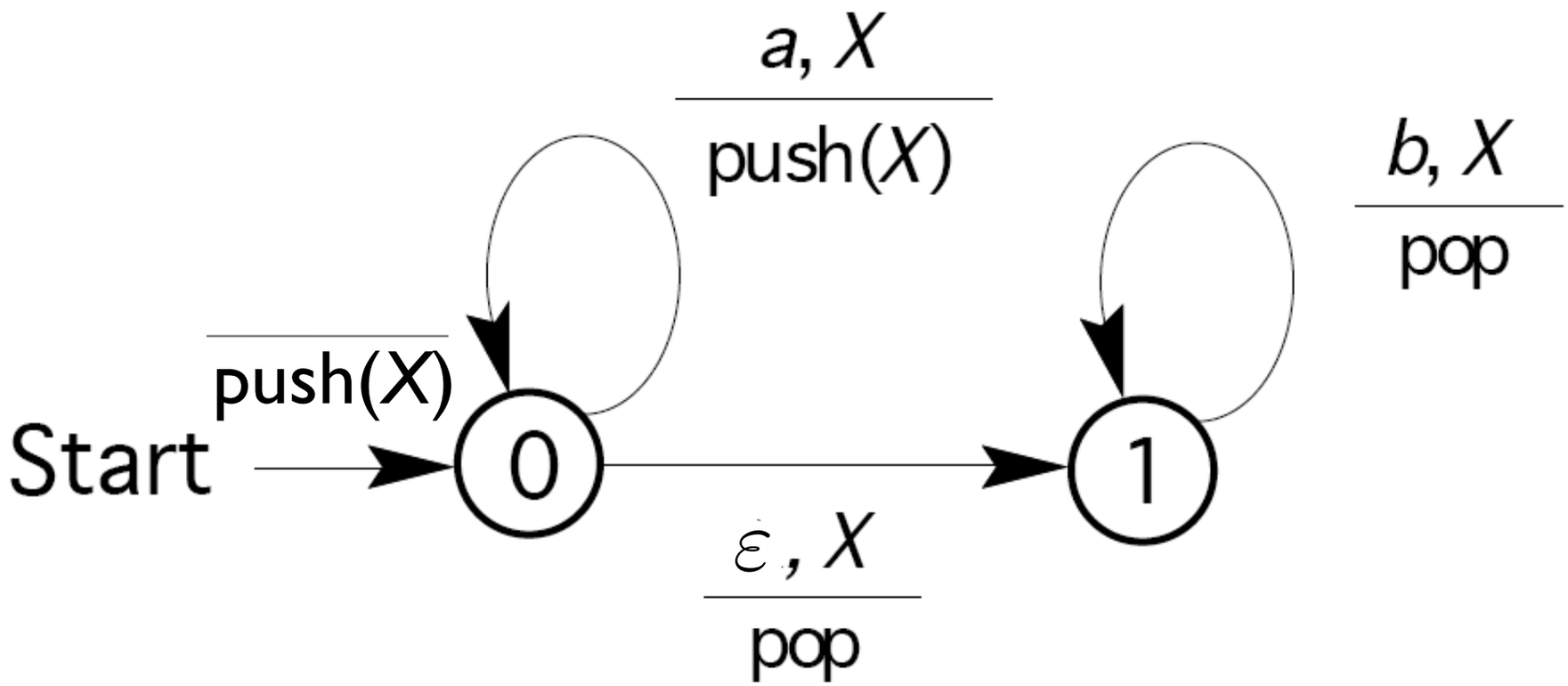
It doesn't matter!

- PDAs that accept by empty stack and PDAs that accept by final state have *equivalent power*

- What does this mean?
  - That given either one, we can build the other
  - Consequently, the class of languages that they accept are the same

Example

(Hein 12.2)

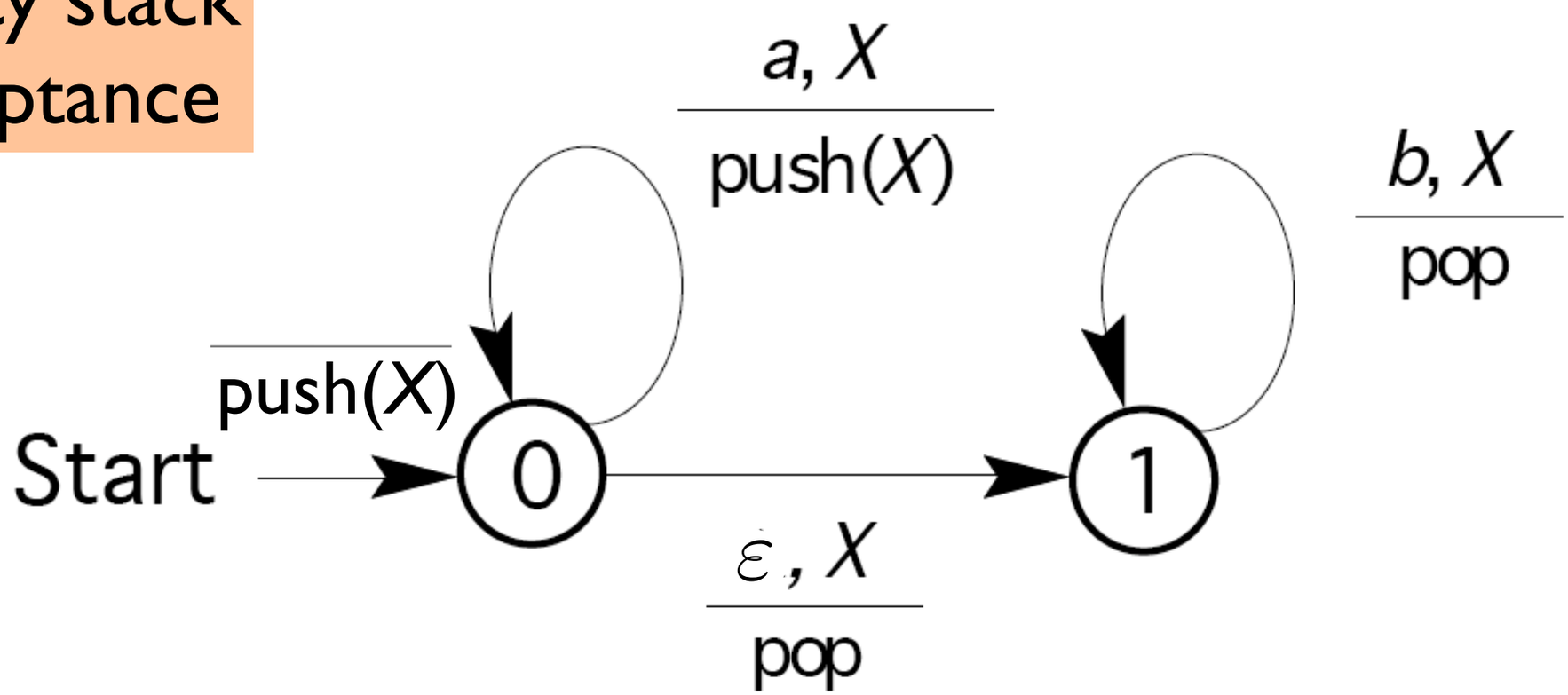


$\frac{a, X}{push(X)}$ means that (in state 0), with input a and with X on the top of the stack, we can transition into state 0, and put an additional X onto the stack.

Example

(Hein 12.2)

Empty stack acceptance

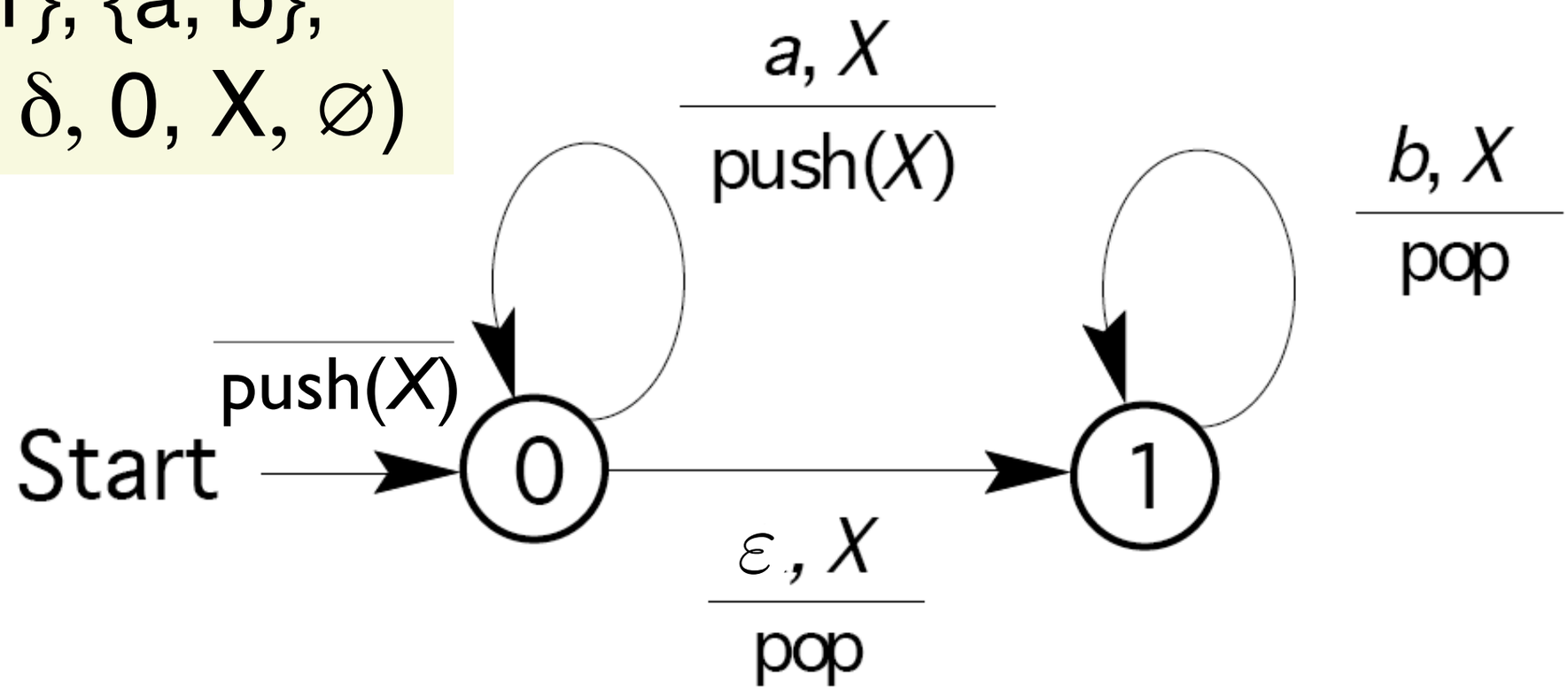


$\frac{a, X}{push(X)}$ means that (in state 0), with input a and with X on the top of the stack, we can transition into state 0, and put an additional X onto the stack.

Example

(Hein 12.2)

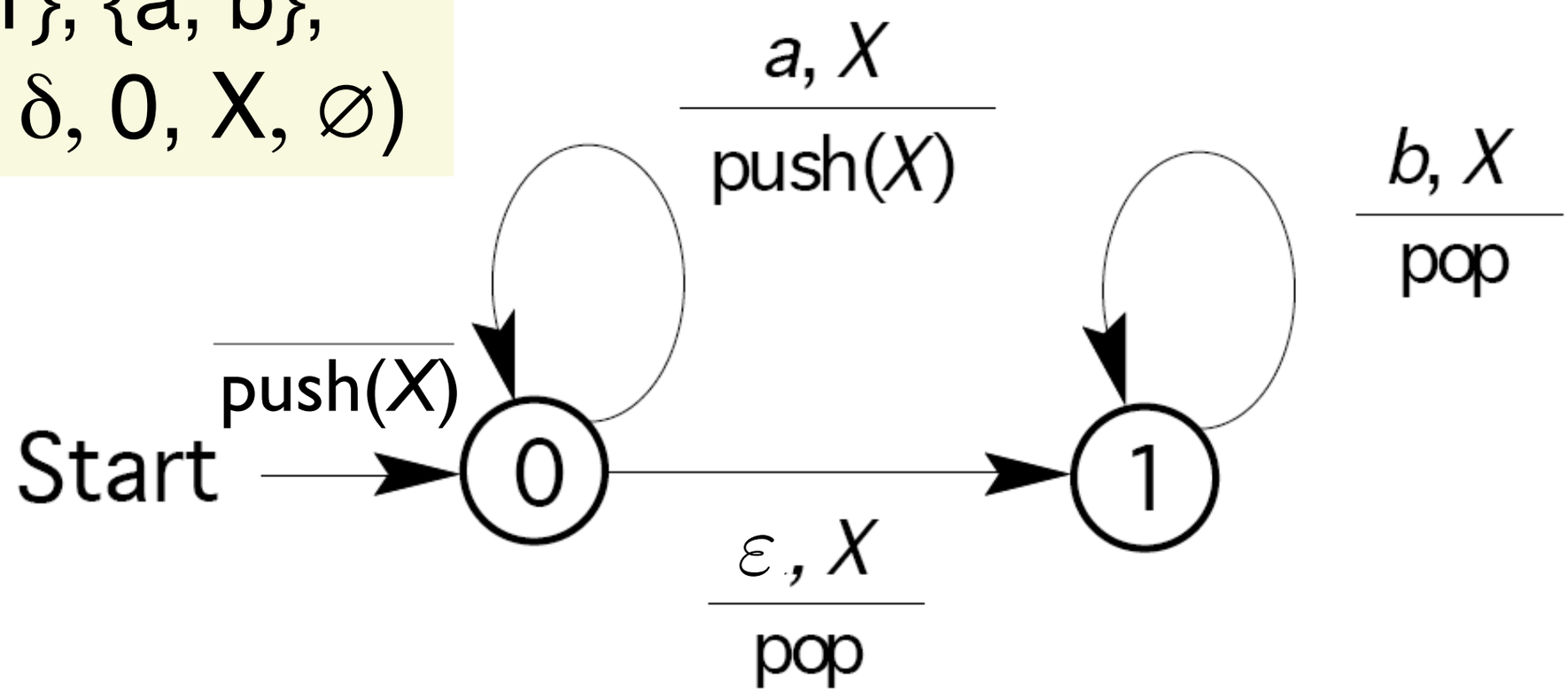
M = ({0,1}, {a, b},
{X}, δ, 0, X, ∅)



Example

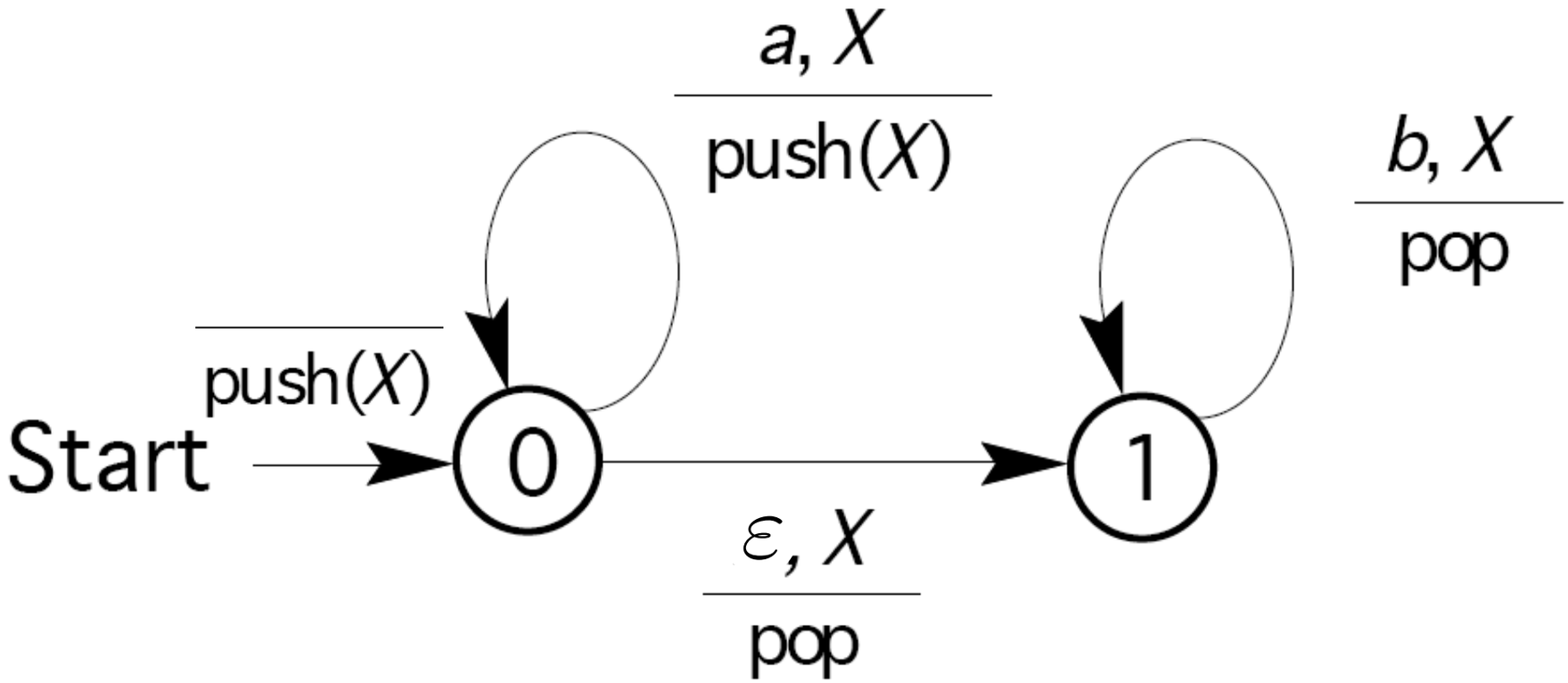
(Hein 12.2)

M = ({0,1}, {a, b},
{X}, δ, 0, X, ∅)

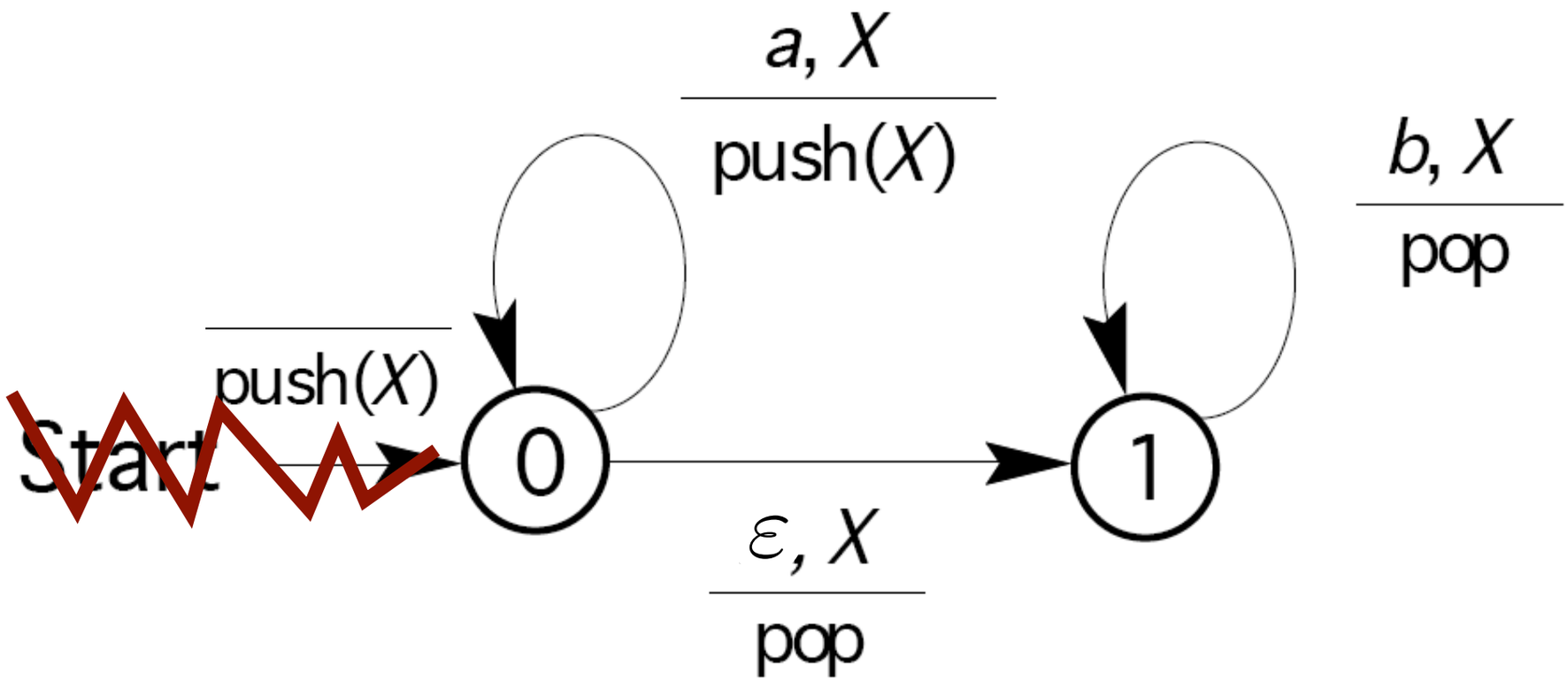


- δ(0, a, X) = {(0, XX)}
- δ(0, ε, X) = {(1, ε)}
- δ(1, b, X) = {(1, ε)}

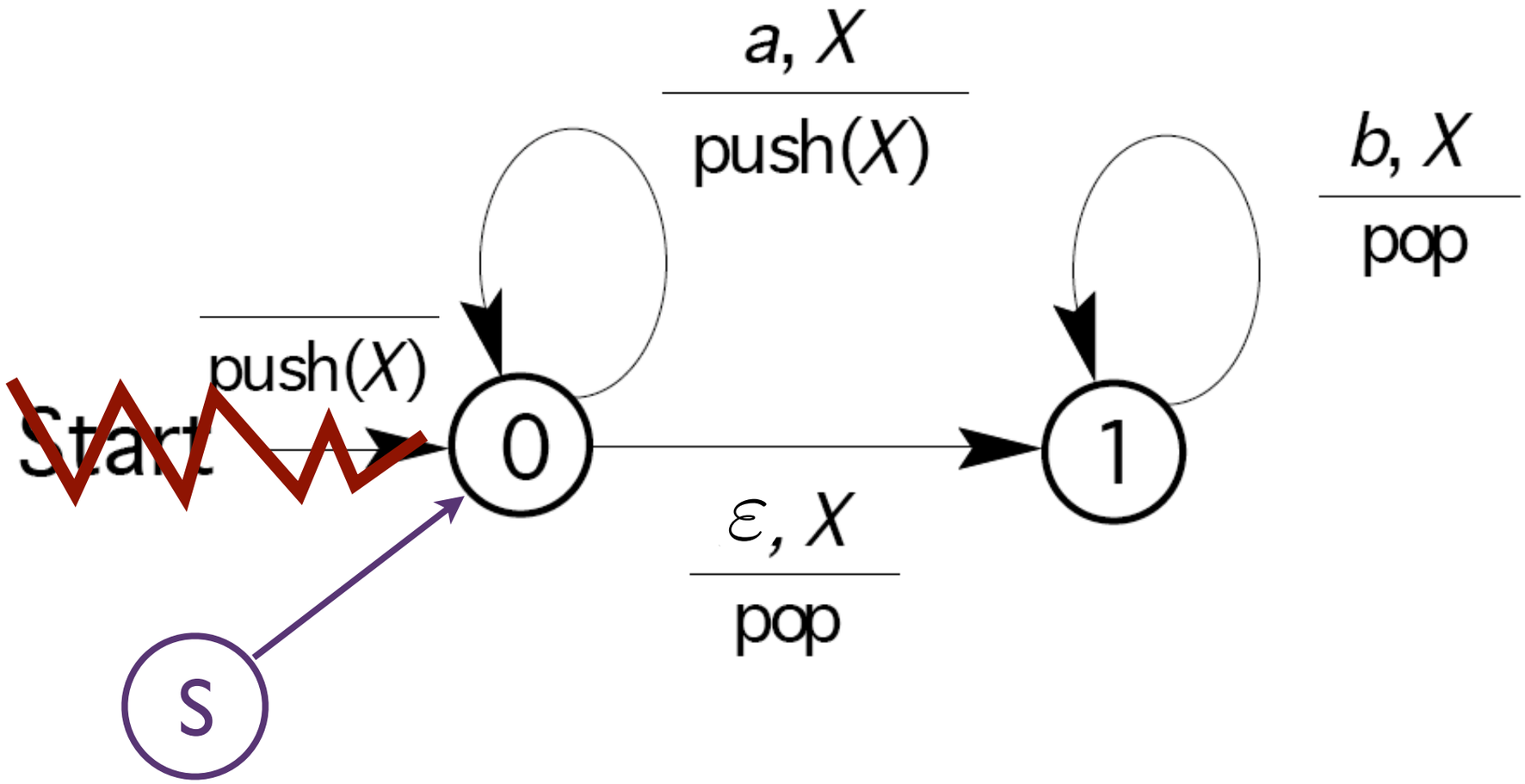
Transformation to a Final State PDA



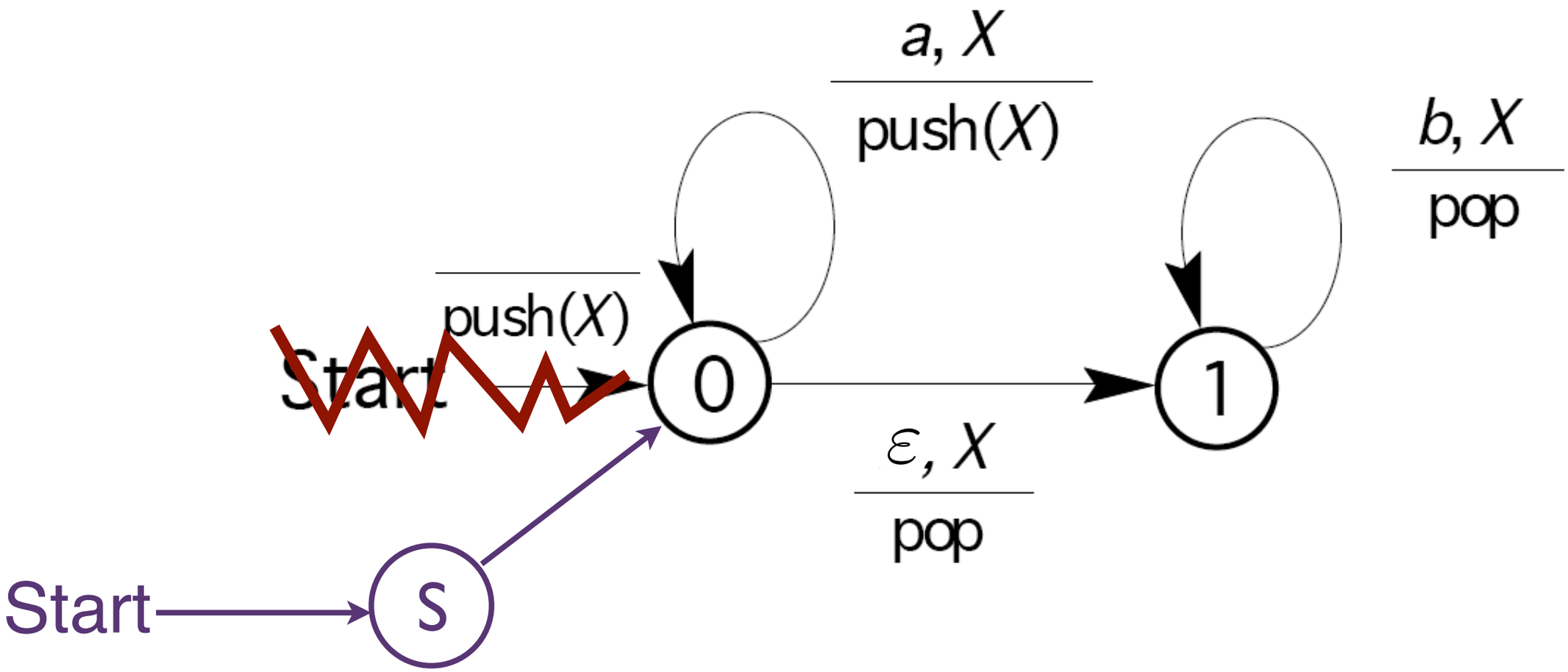
Transformation to a Final State PDA



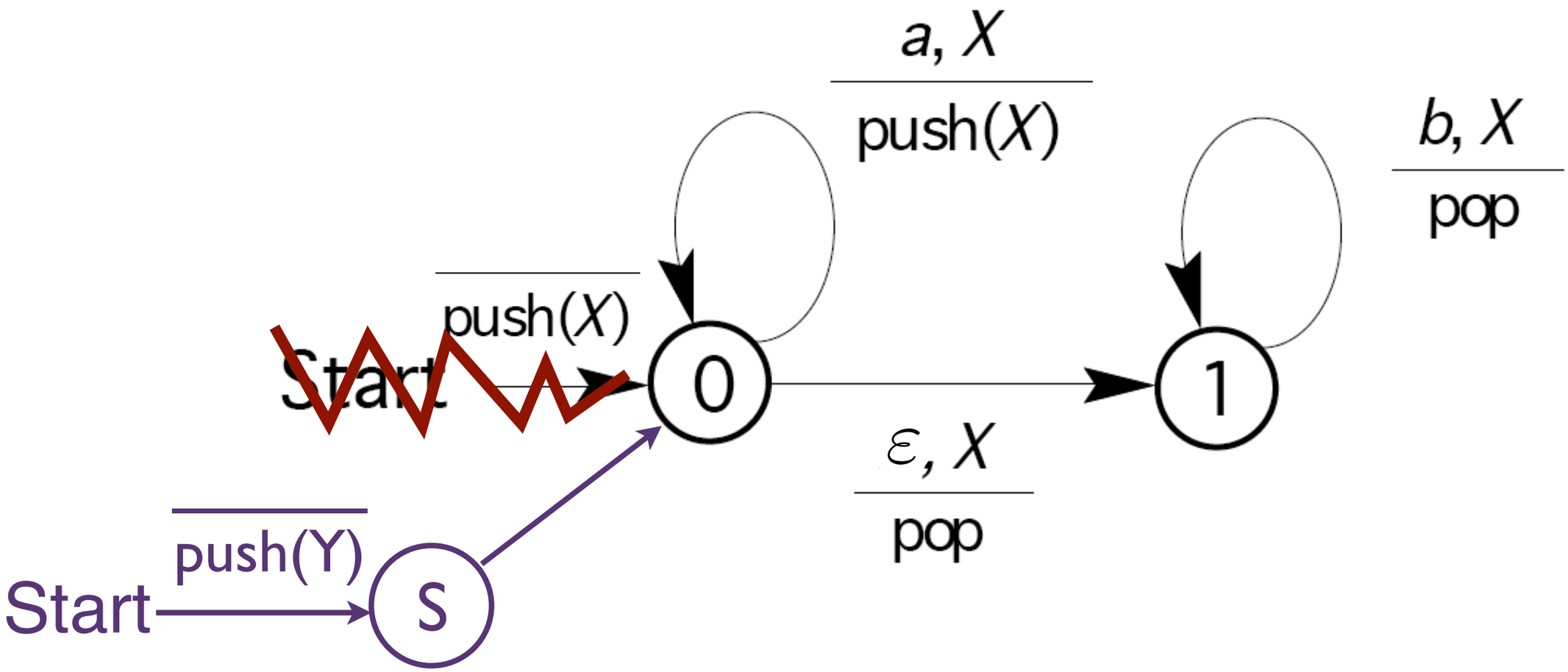
Transformation to a Final State PDA



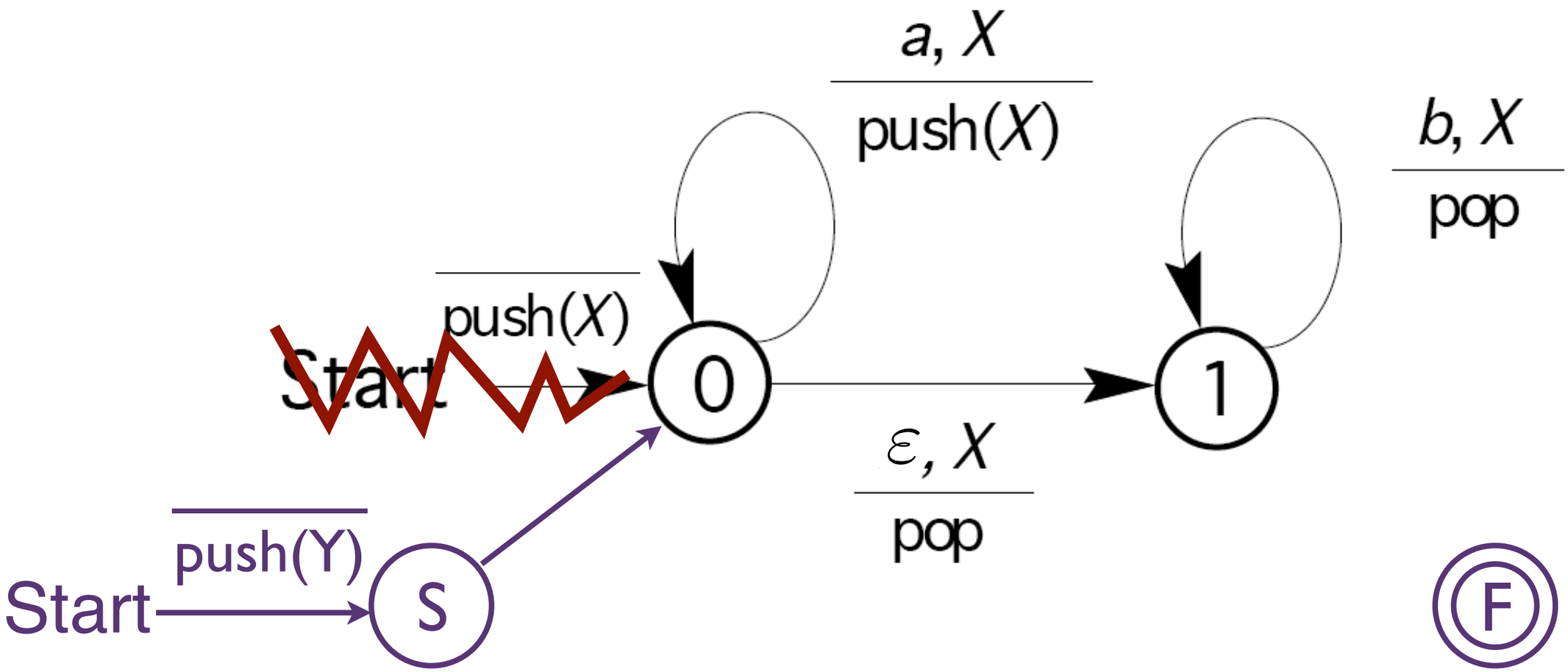
Transformation to a Final State PDA



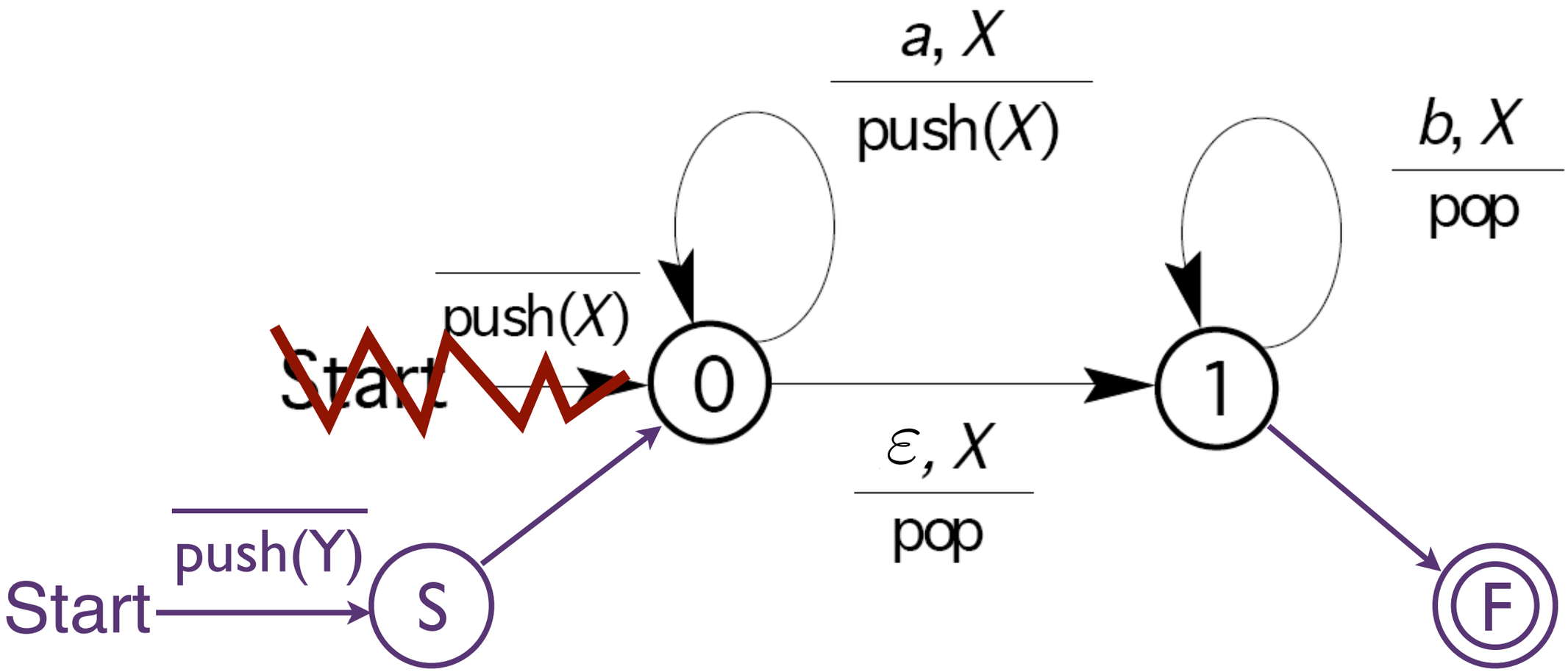
Transformation to a Final State PDA



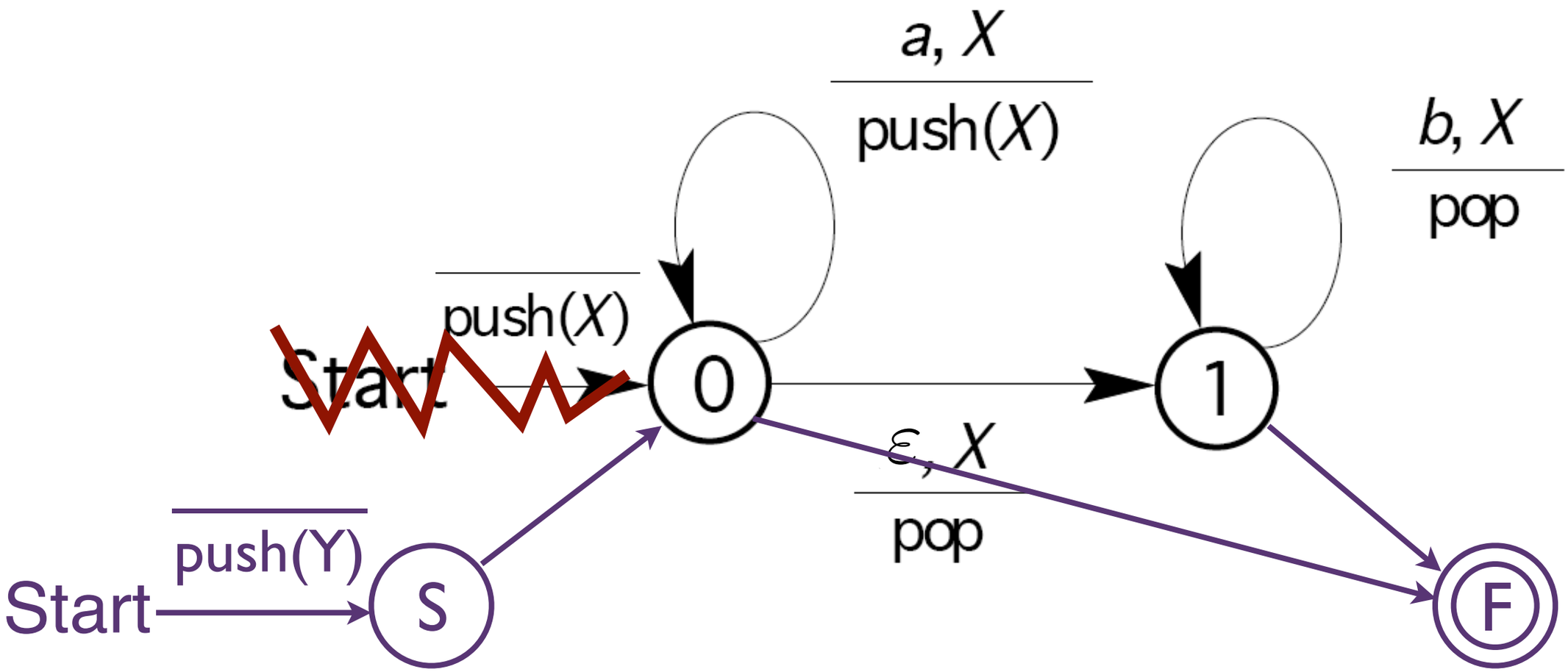
Transformation to a Final State PDA



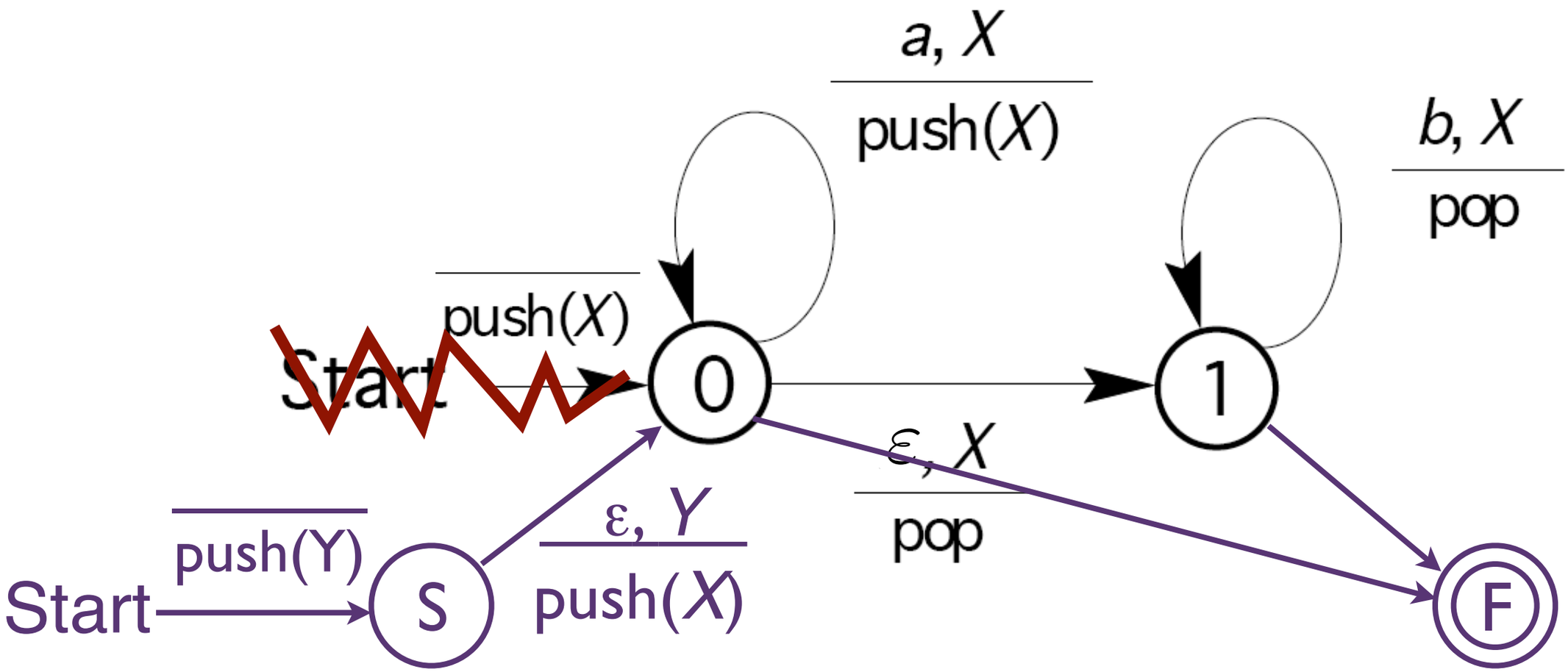
Transformation to a Final State PDA



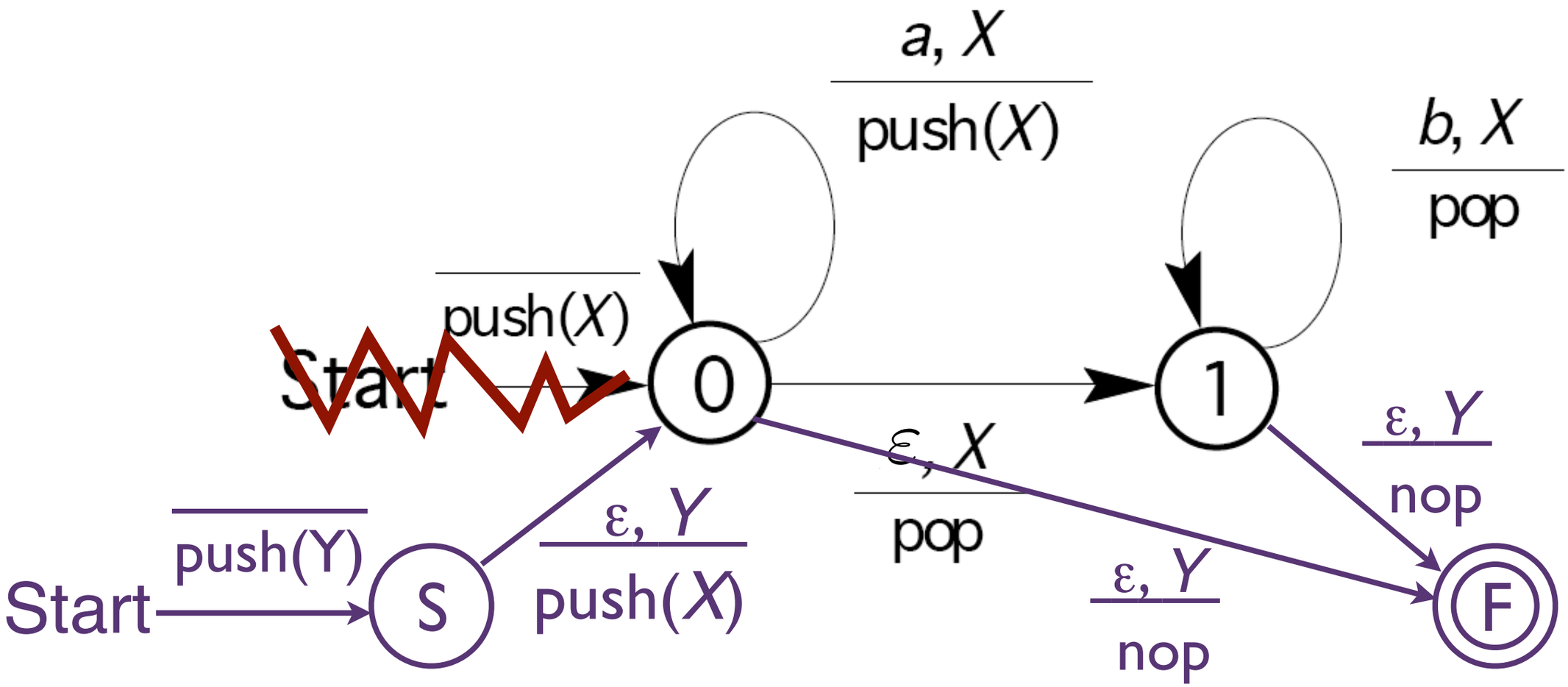
Transformation to a Final State PDA



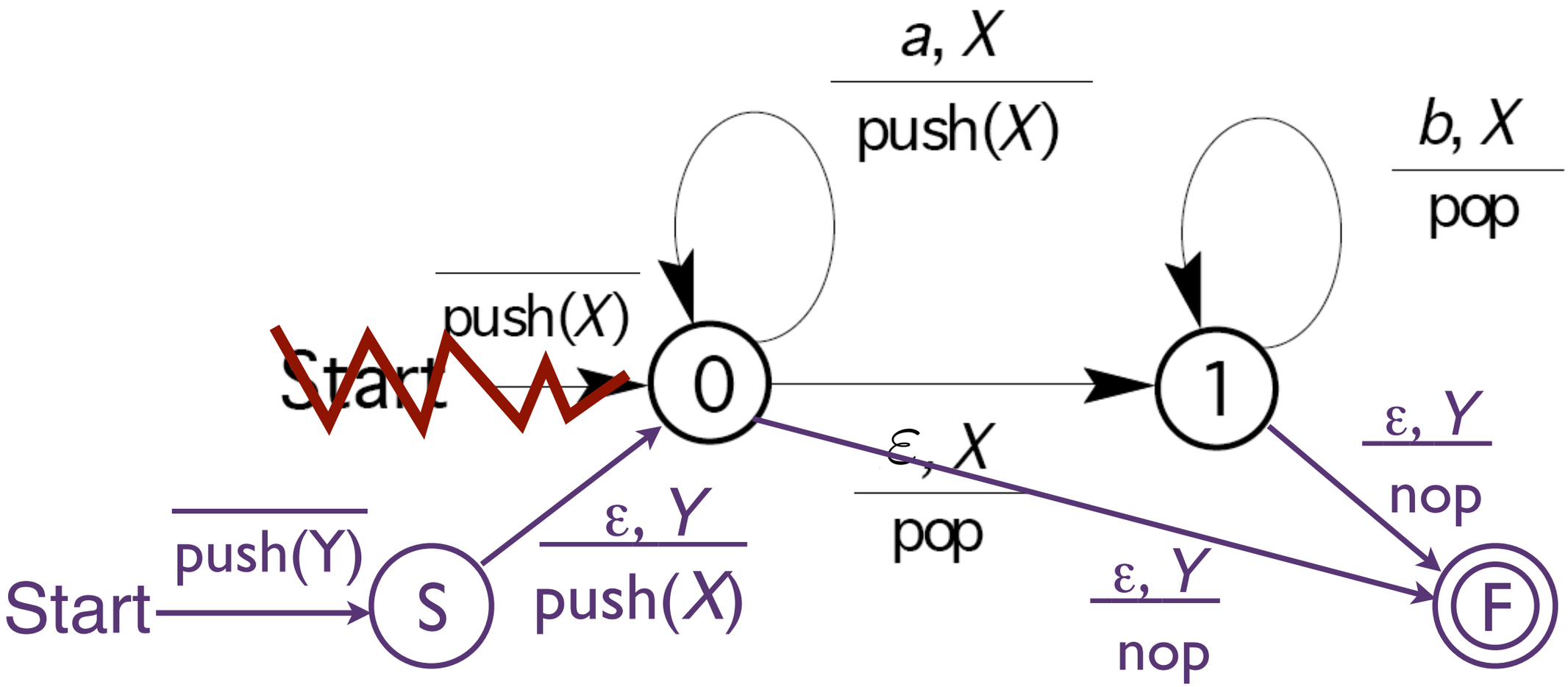
Transformation to a Final State PDA



Transformation to a Final State PDA

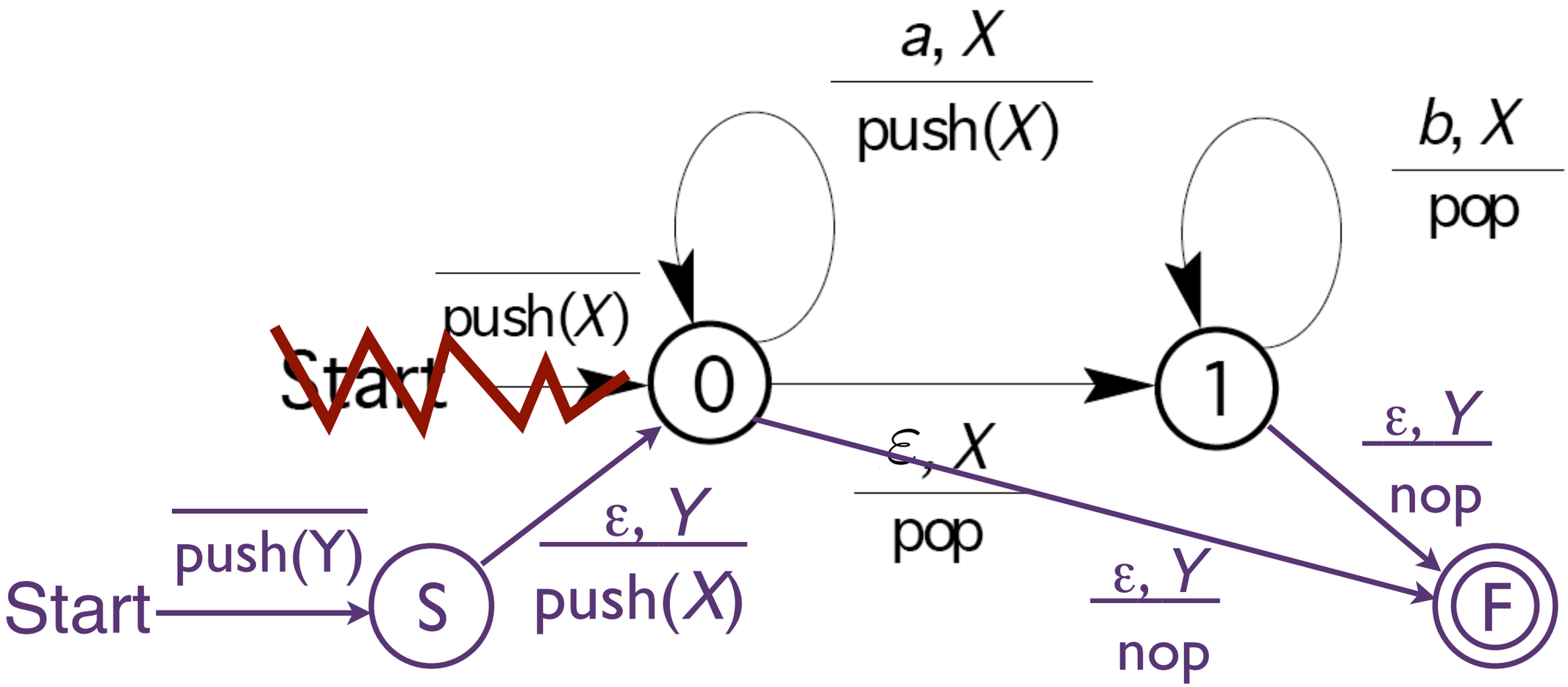


Transformation to a Final State PDA



- N = ({0,1, S, F}, {a, b}, {X,Y}, δ′, S, Y, {F})

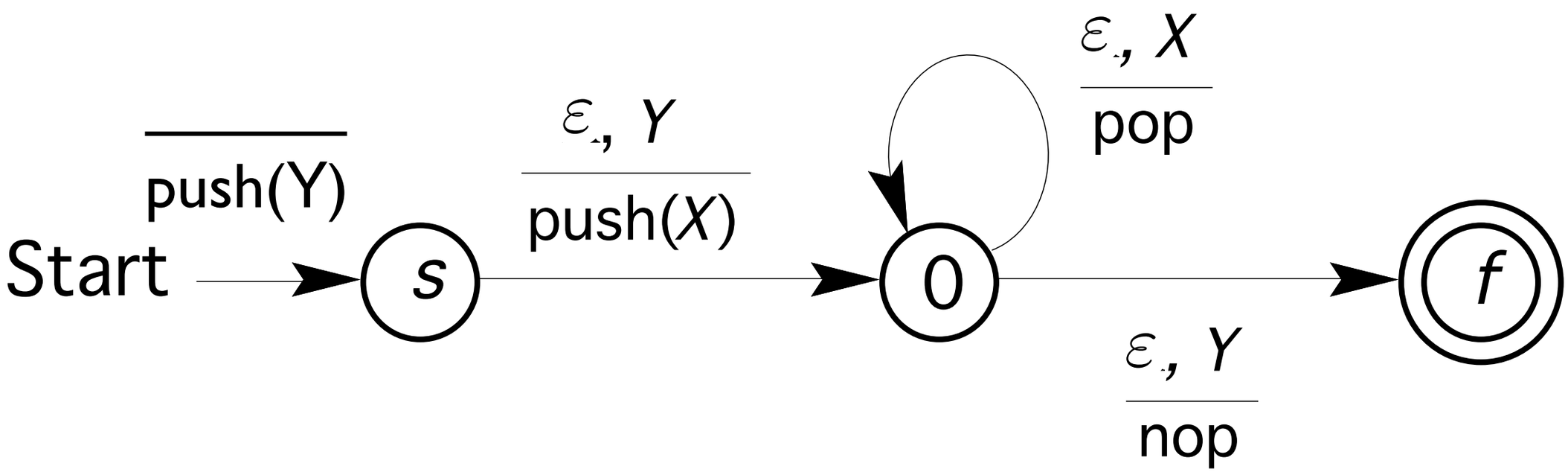
Transformation to a Final State PDA



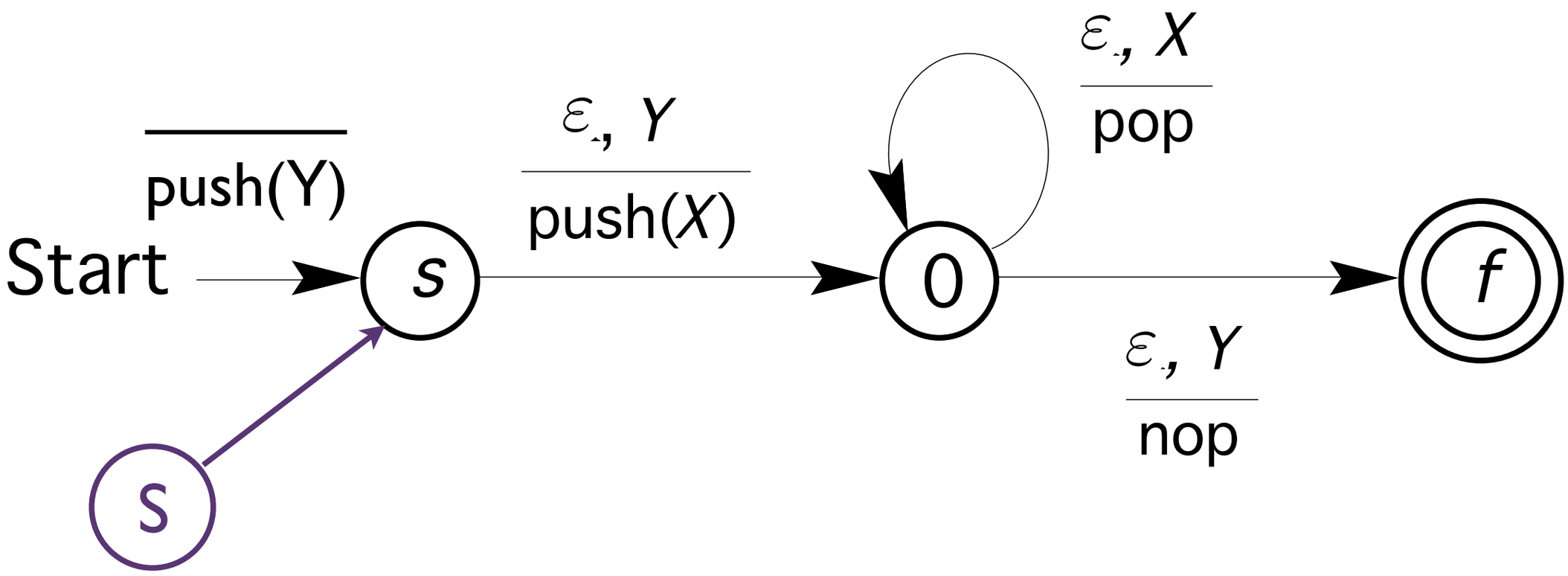
- N = ({0,1, S, F}, {a, b}, {X,Y}, δ′, S, Y, {F})
- δ′ = δ + δ′(S, ε, Y) = {(0, XY)} +

  δ′(0, ε, Y) = {(F, ε)} + δ′(1, ε, Y) = {(F, ε)}

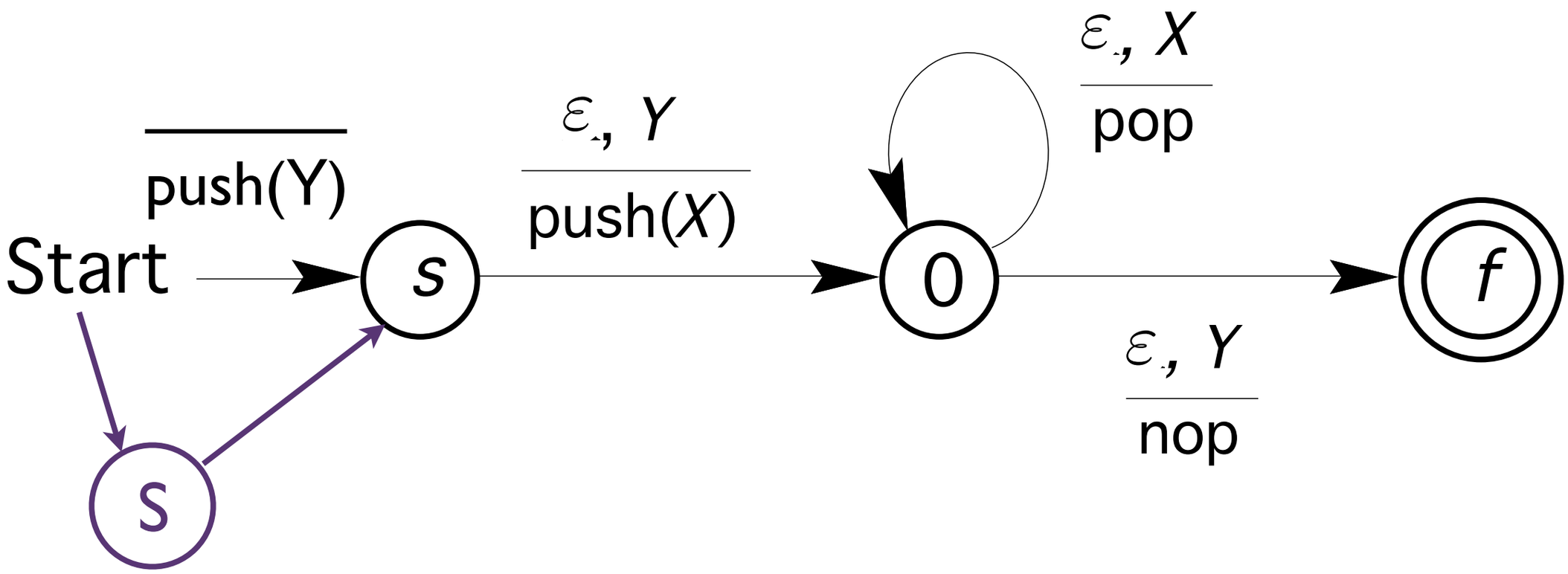
Transformation to an EmptyStack PDA



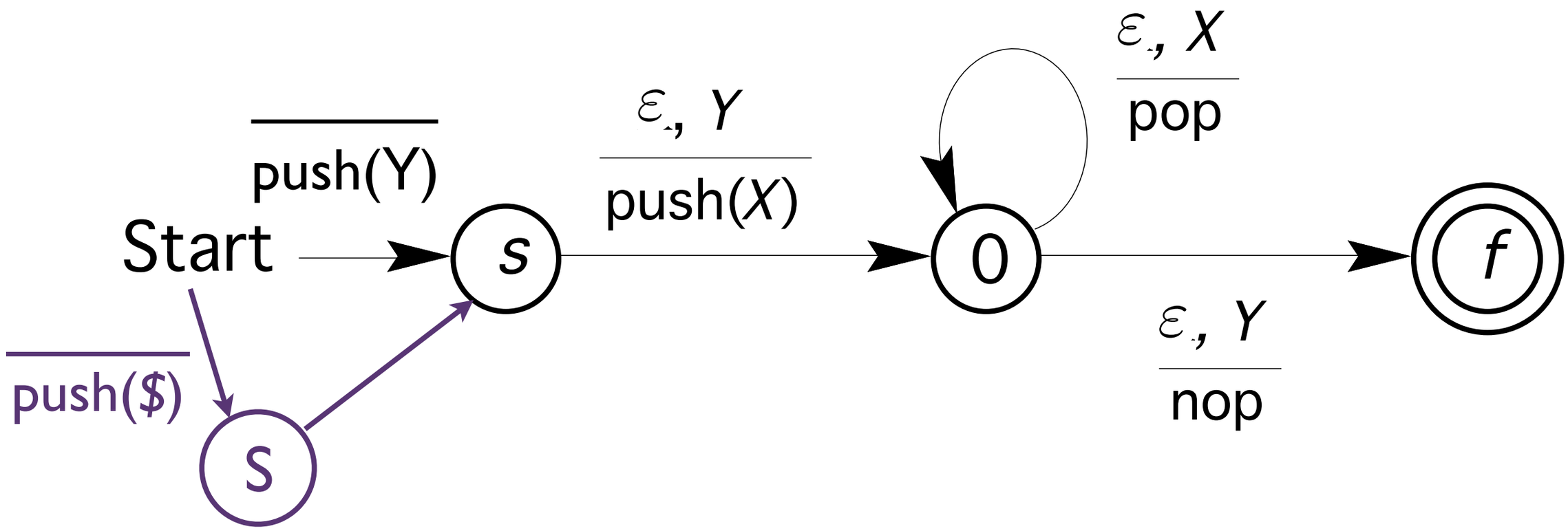
Transformation to an EmptyStack PDA



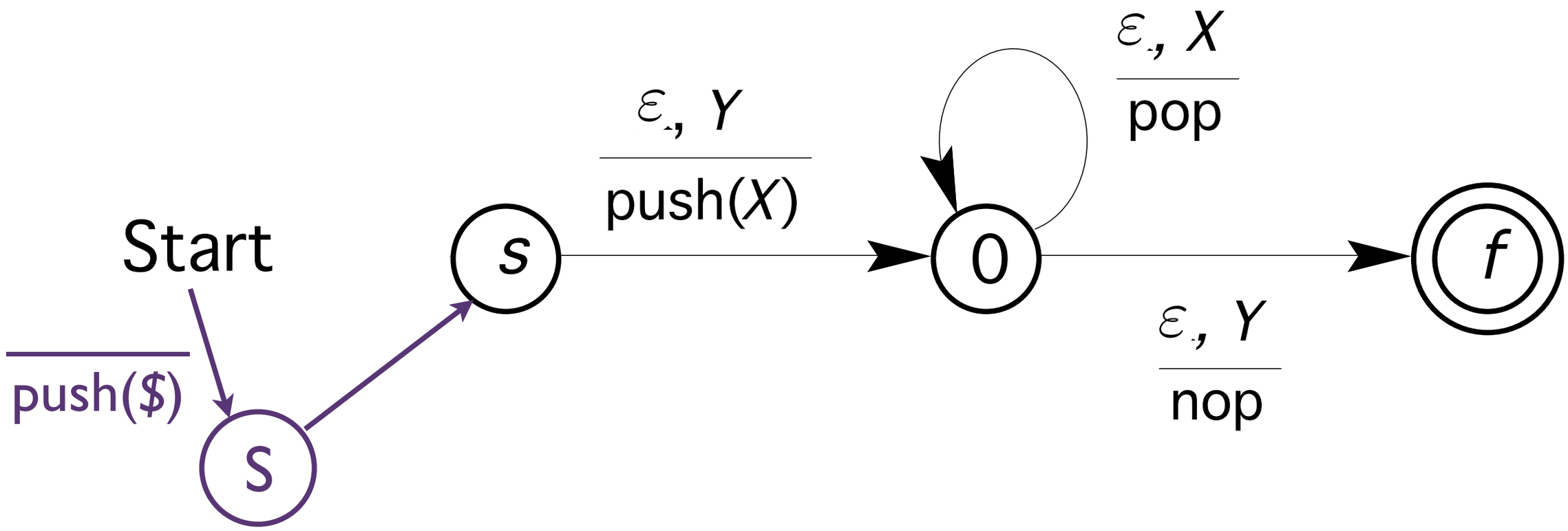
Transformation to an EmptyStack PDA



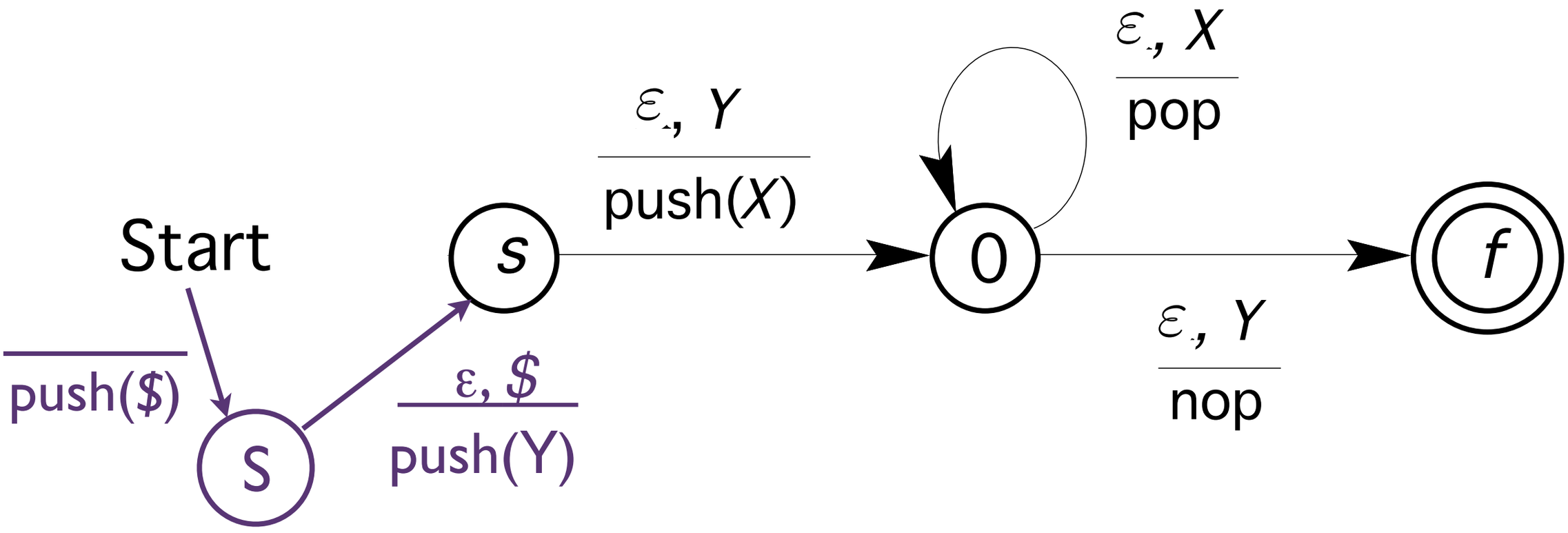
Transformation to an EmptyStack PDA



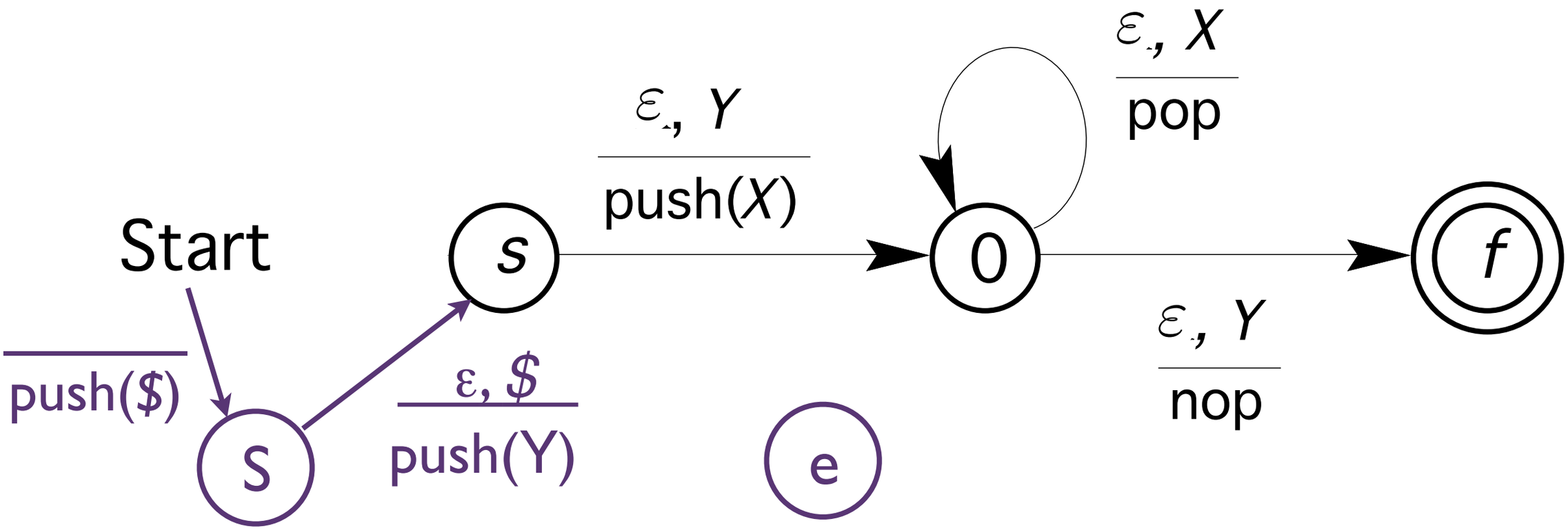
Transformation to an EmptyStack PDA



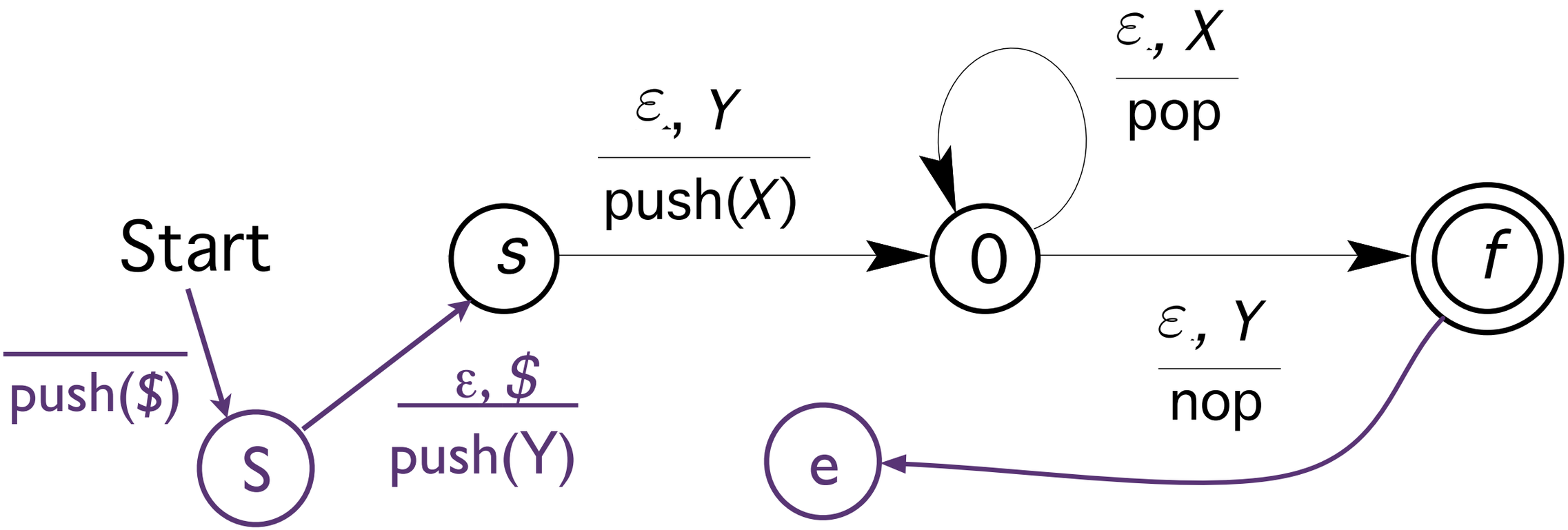
Transformation to an EmptyStack PDA



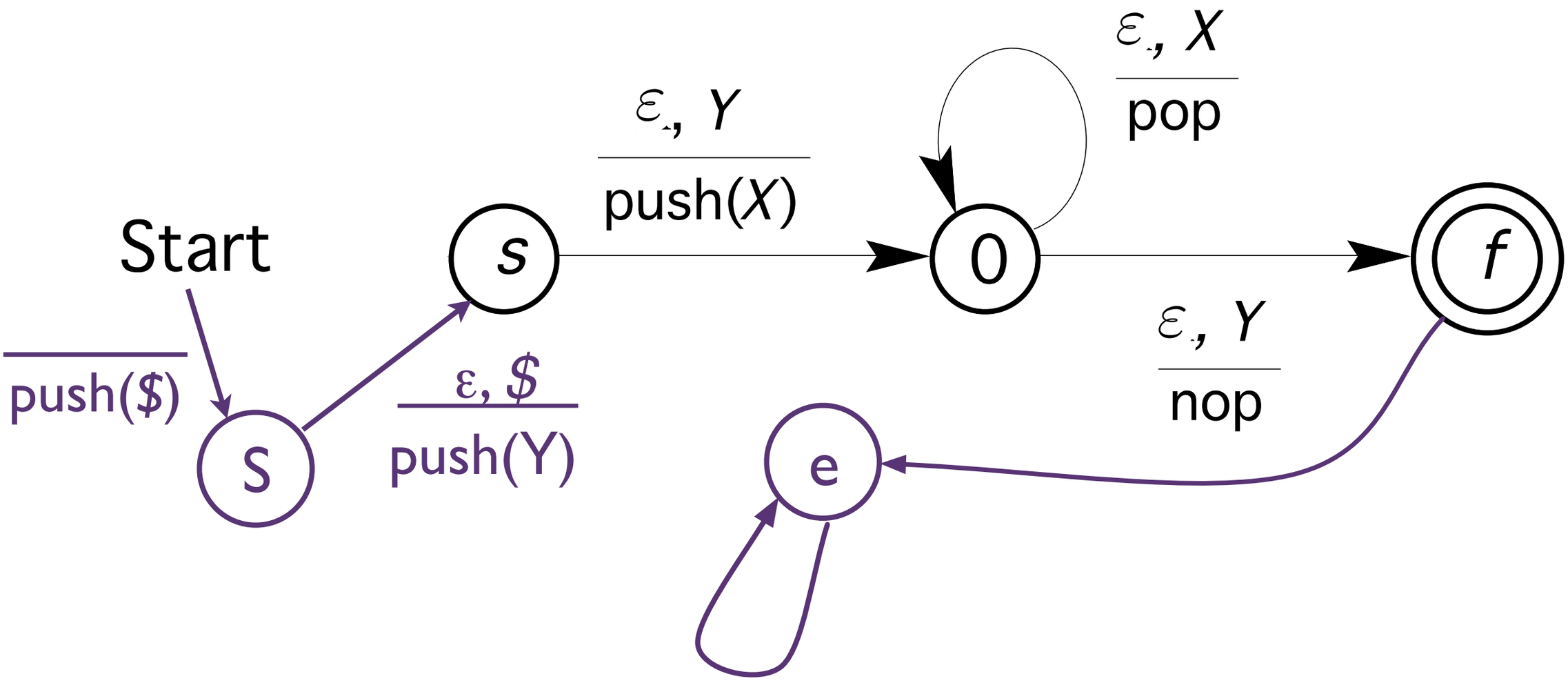
Transformation to an EmptyStack PDA



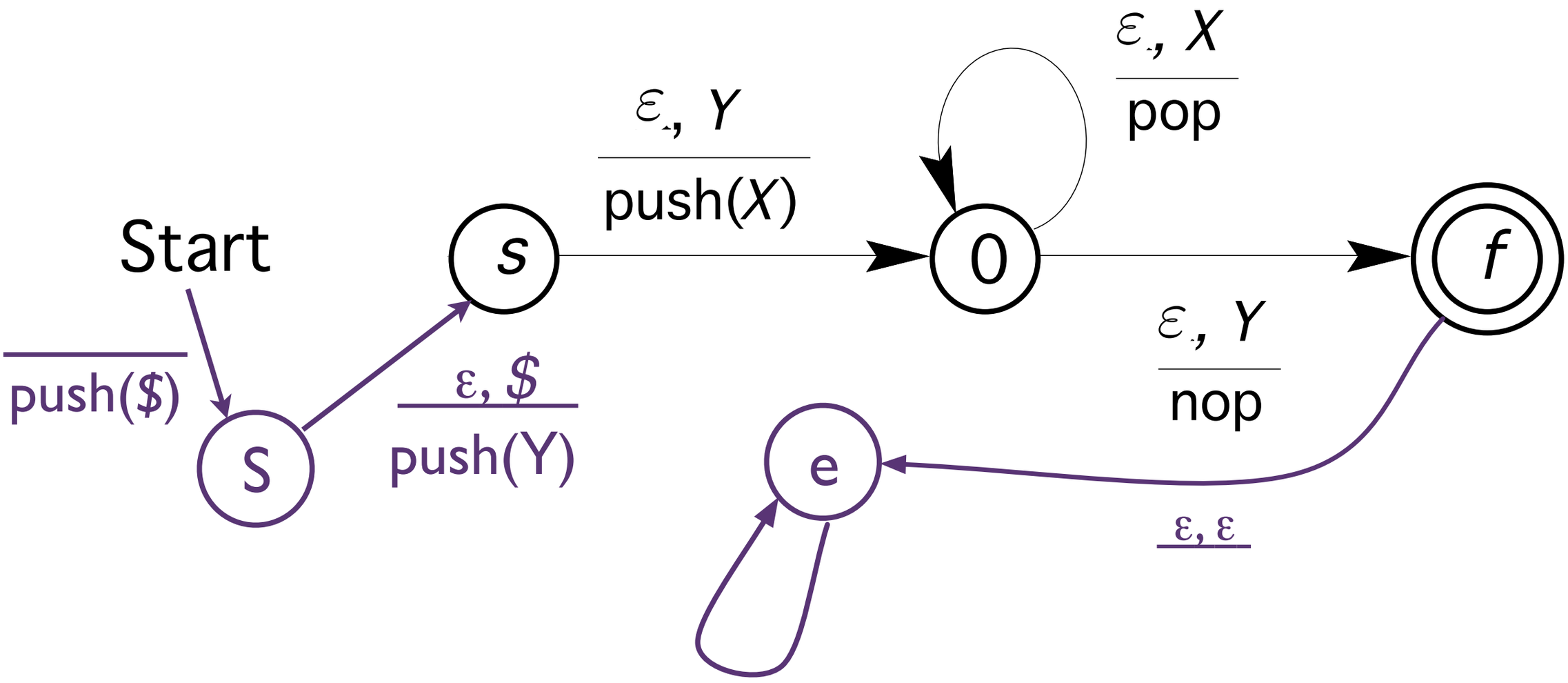
Transformation to an EmptyStack PDA



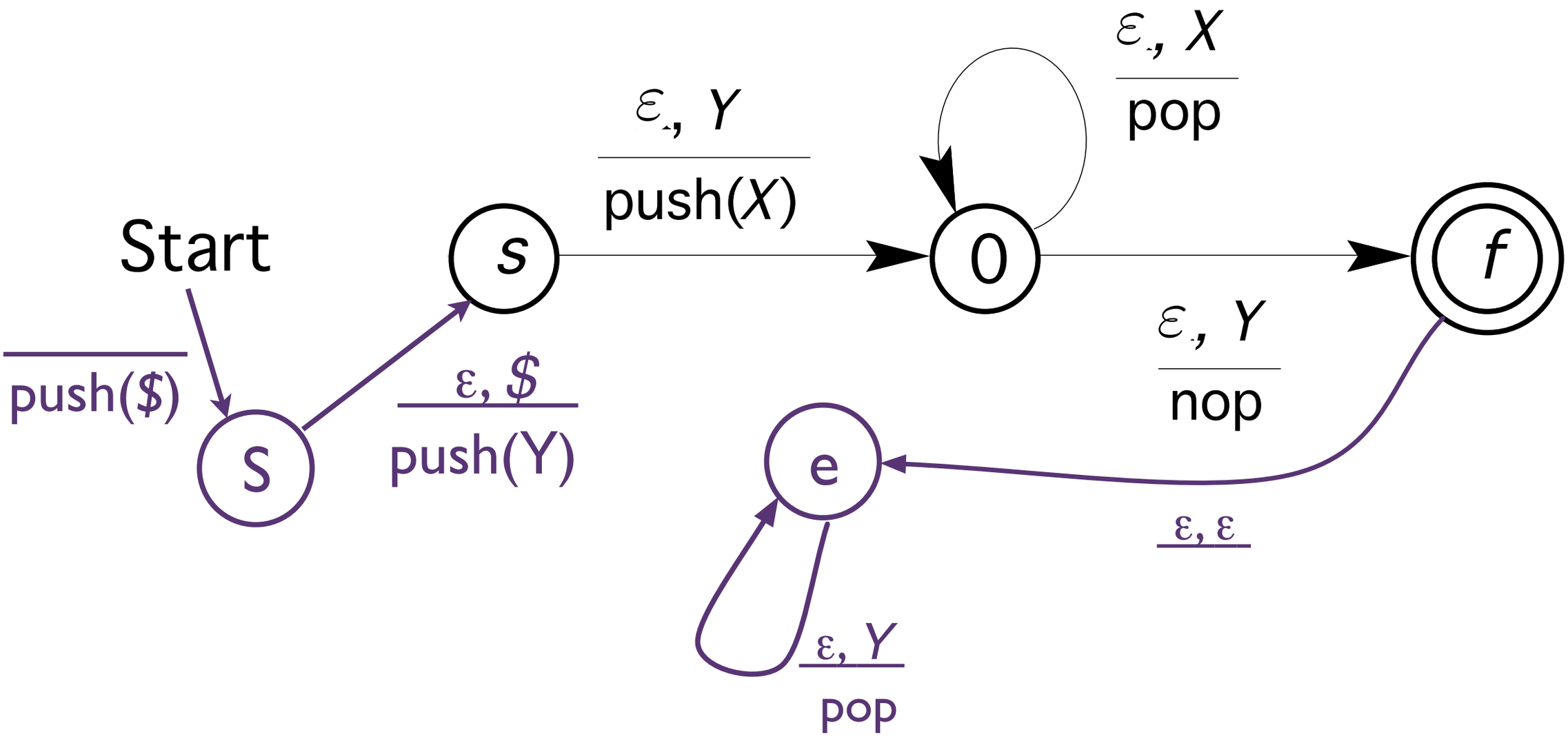
Transformation to an EmptyStack PDA



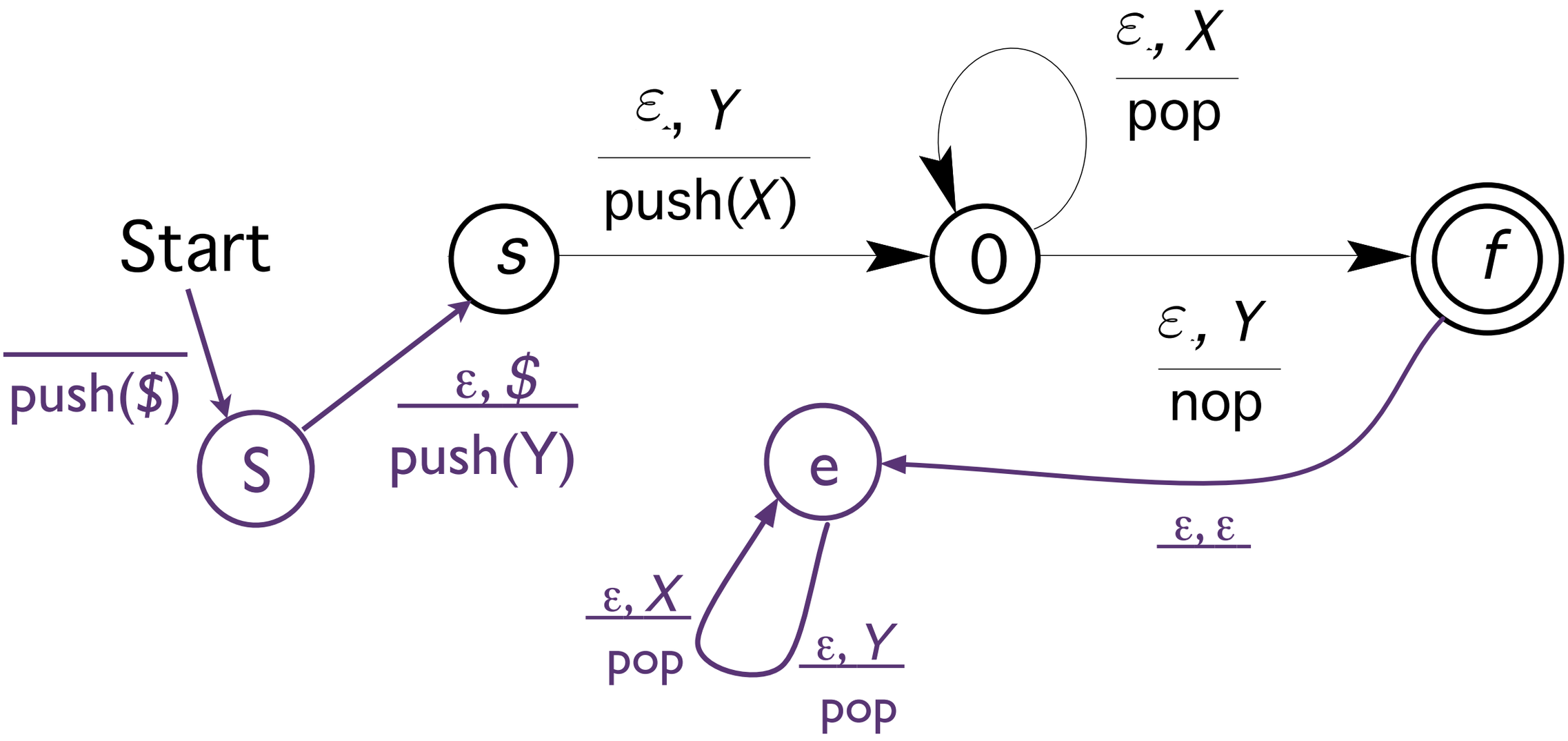
Transformation to an EmptyStack PDA



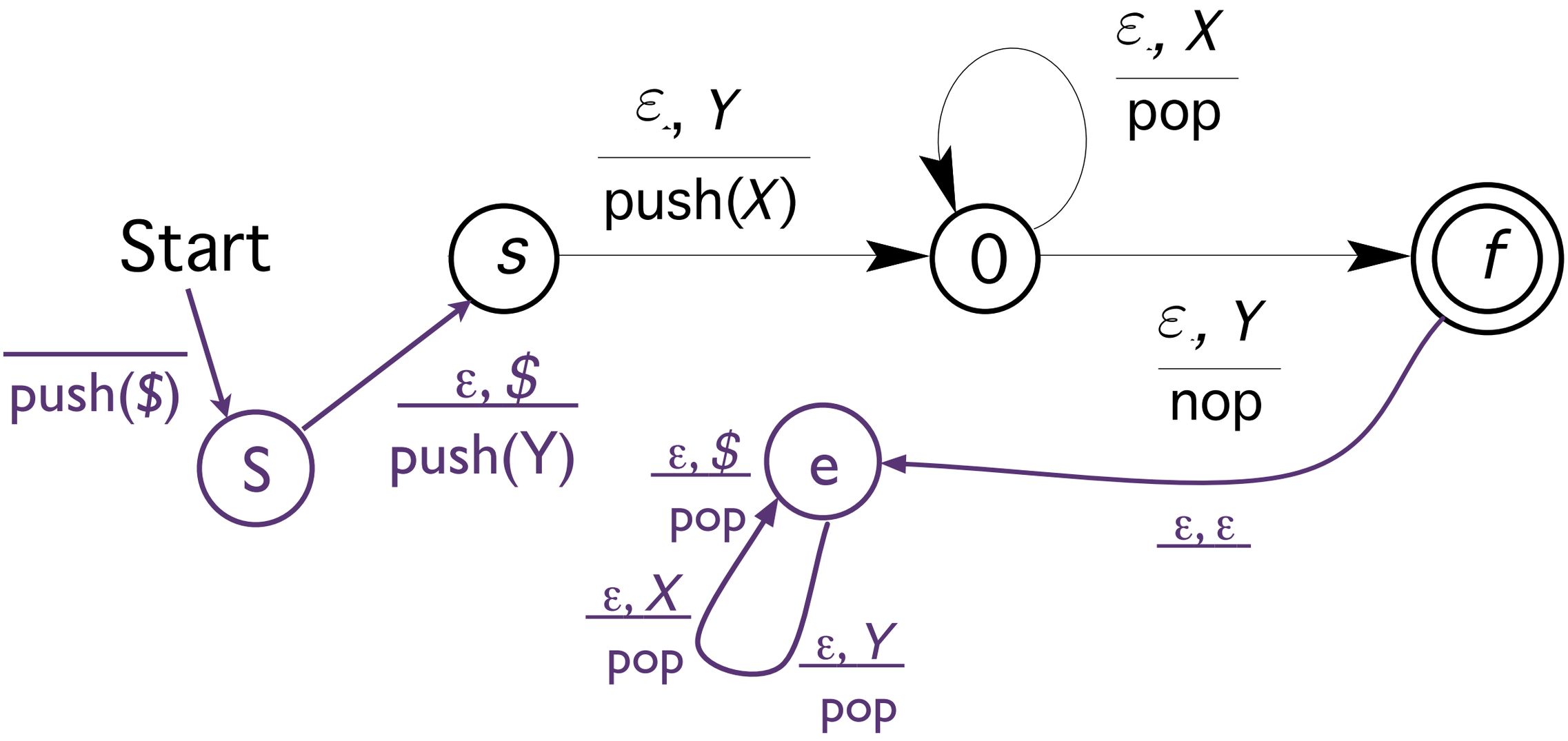
Transformation to an EmptyStack PDA



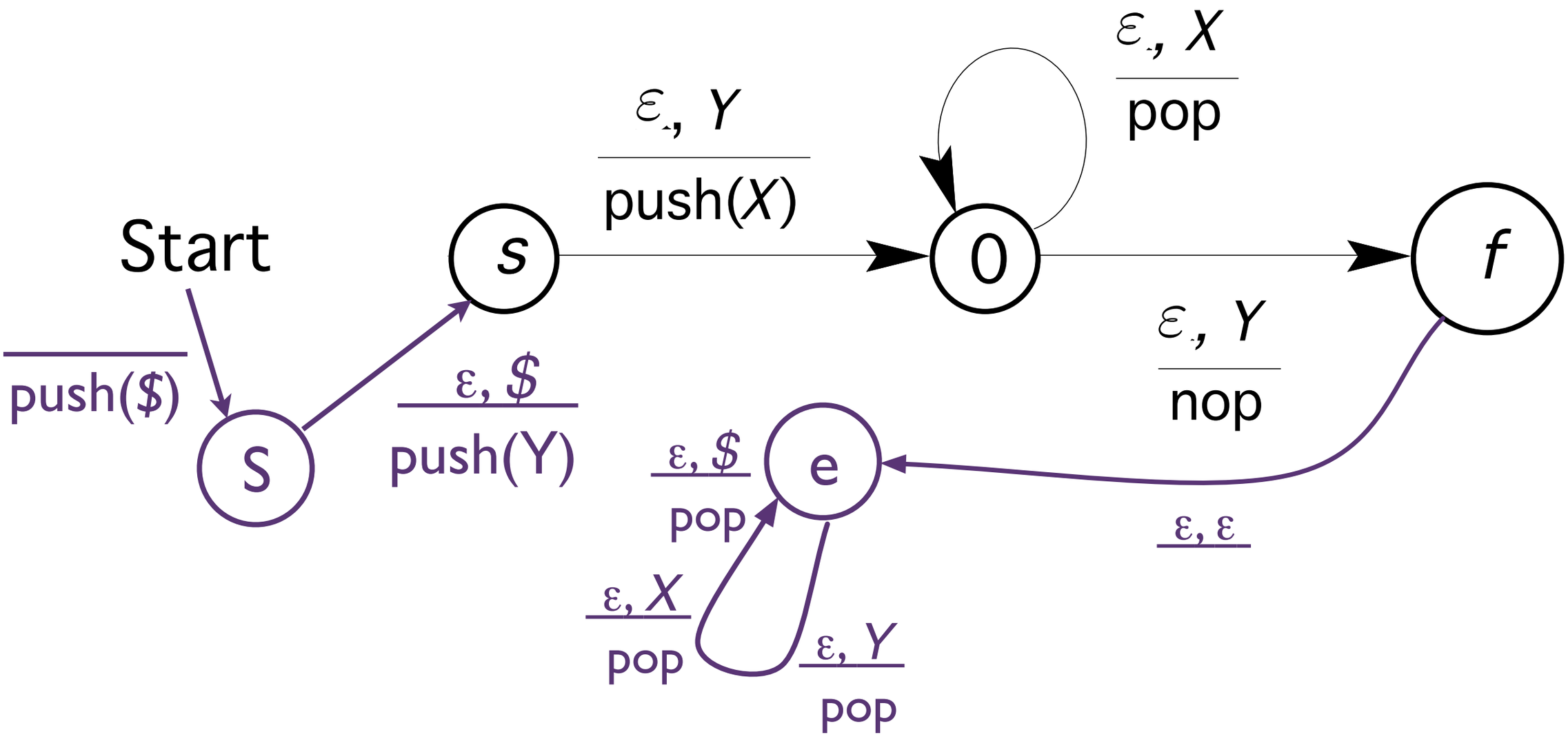
Transformation to an EmptyStack PDA



Transformation to an EmptyStack PDA



Transformation to an EmptyStack PDA



Other Variations

- Hein says: “the execution of a PDA always begins with one symbol on the stack”
- Other authors say “the stack is initially empty”
- Does it matter?

Other Variations

- Hein says: “the execution of a PDA always begins with one symbol on the stack”
- Other authors say “the stack is initially empty”
- Does it matter?

- Can you always convert an initially empty PDA to one with an initial symbol on the stack?

Other Variations

- Hein says: “the execution of a PDA always begins with one symbol on the stack”
- Other authors say “the stack is initially empty”
- Does it matter?

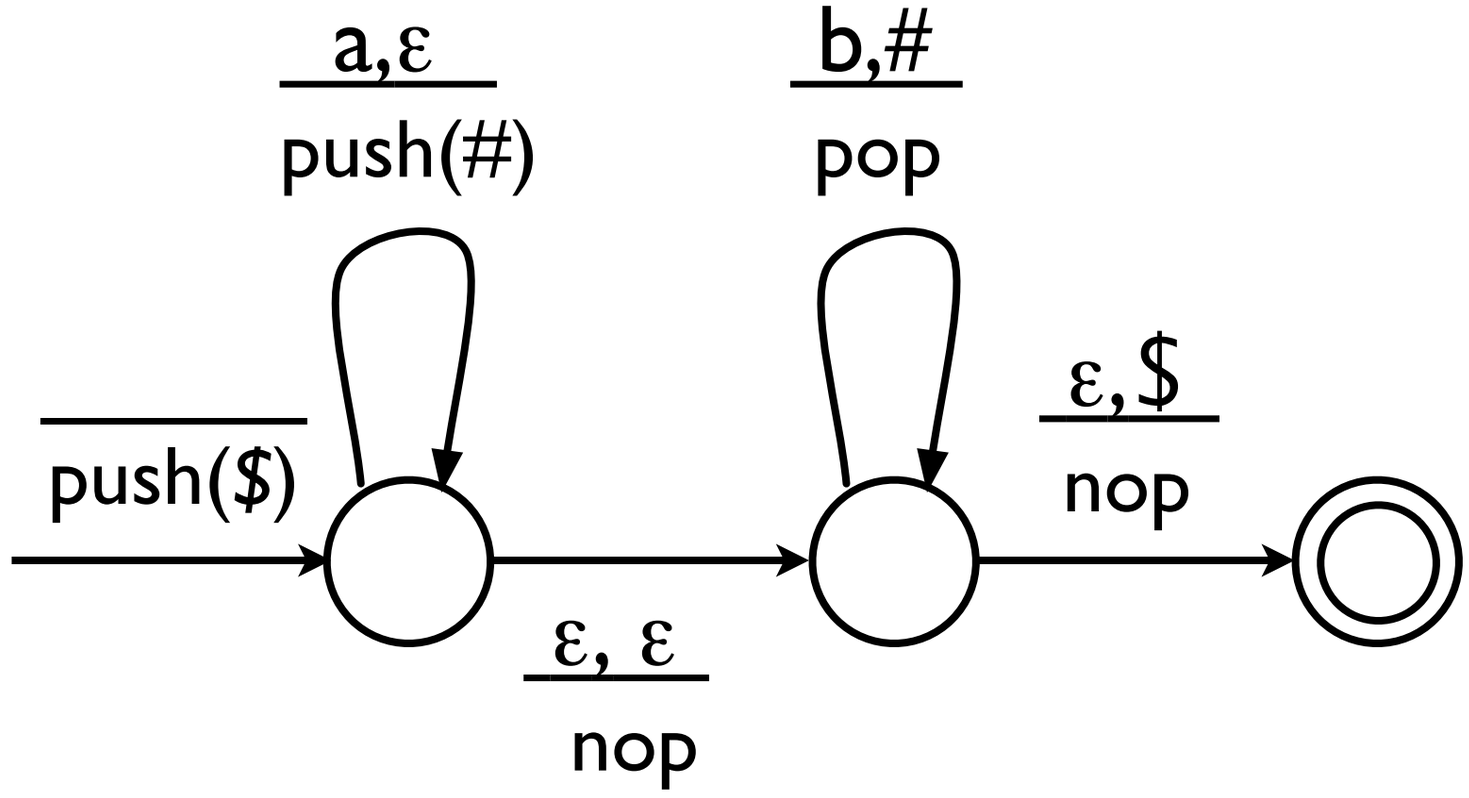
- Can you always convert an initially empty PDA to one with an initial symbol on the stack?
- Can you always convert a PDA with an initial symbol to one whose stack is initially empty?

Stack Operations

- Hopcroft *et al.* says:
  - on each transition, to look at the top of the stack you must pop it, but
  - you can push on as many stack symbols as you like, including the one that you just popped.
- Other authors say
  - you can push *zero or one* symbols onto the stack
- Hein says:
  - on each transition, you can **push**, **pop** or **nop** the stack, but you can't do more than one thing
- Does it matter?

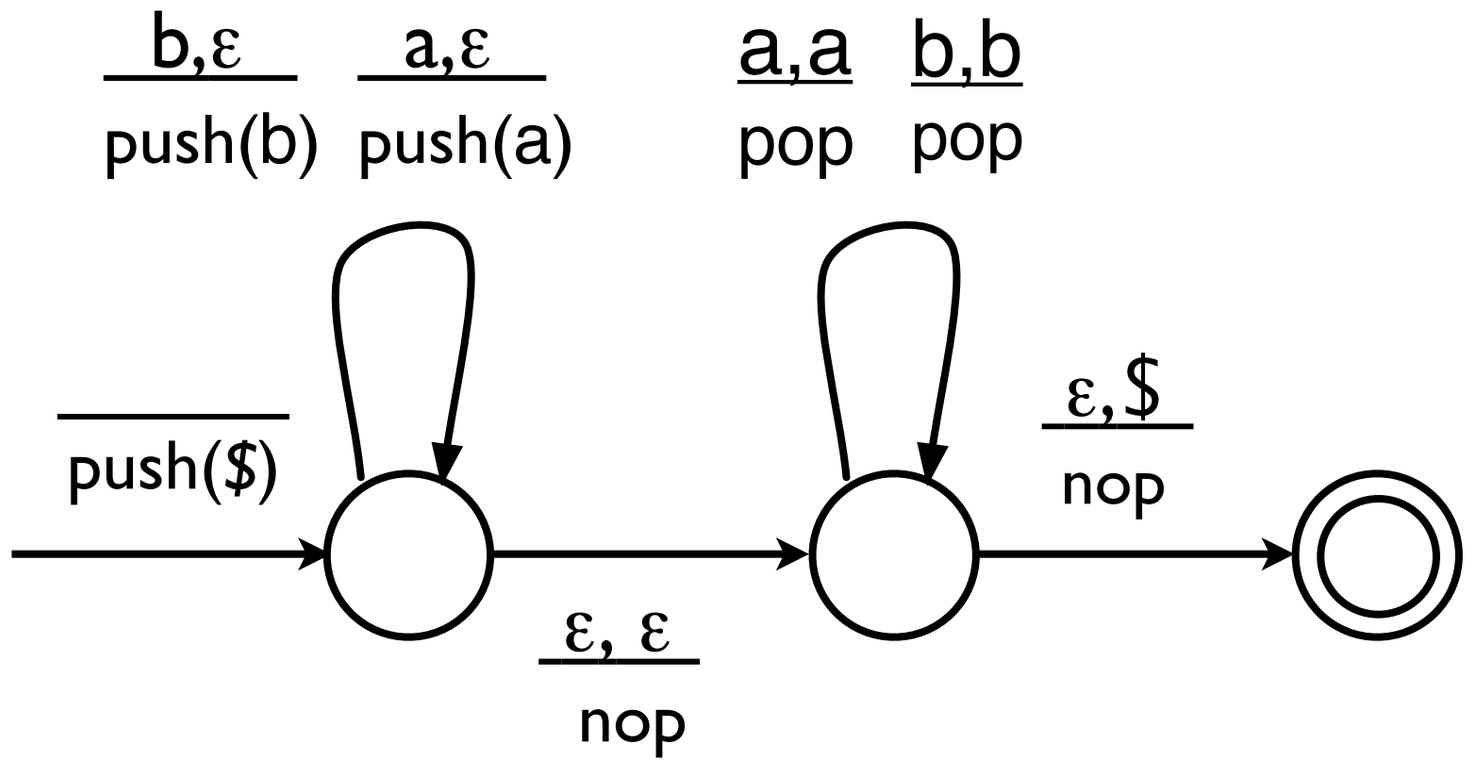
More Examples

$L_1 = \{a^n b^n : n \geq 0\}$



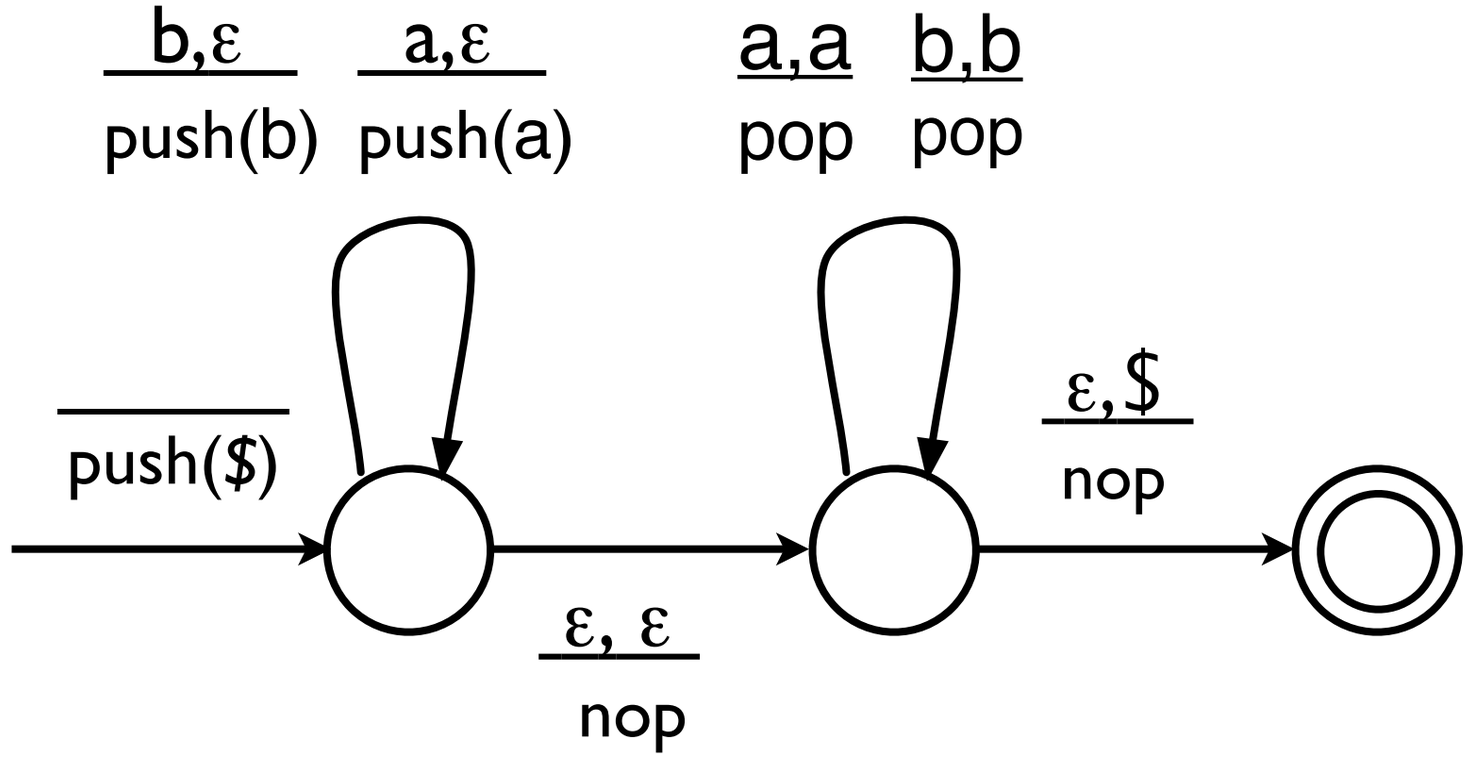
Palindromes of even length

$L_2 = \{ww^R \mid w \in \{a, b\}^*\}$



Palindromes of even length

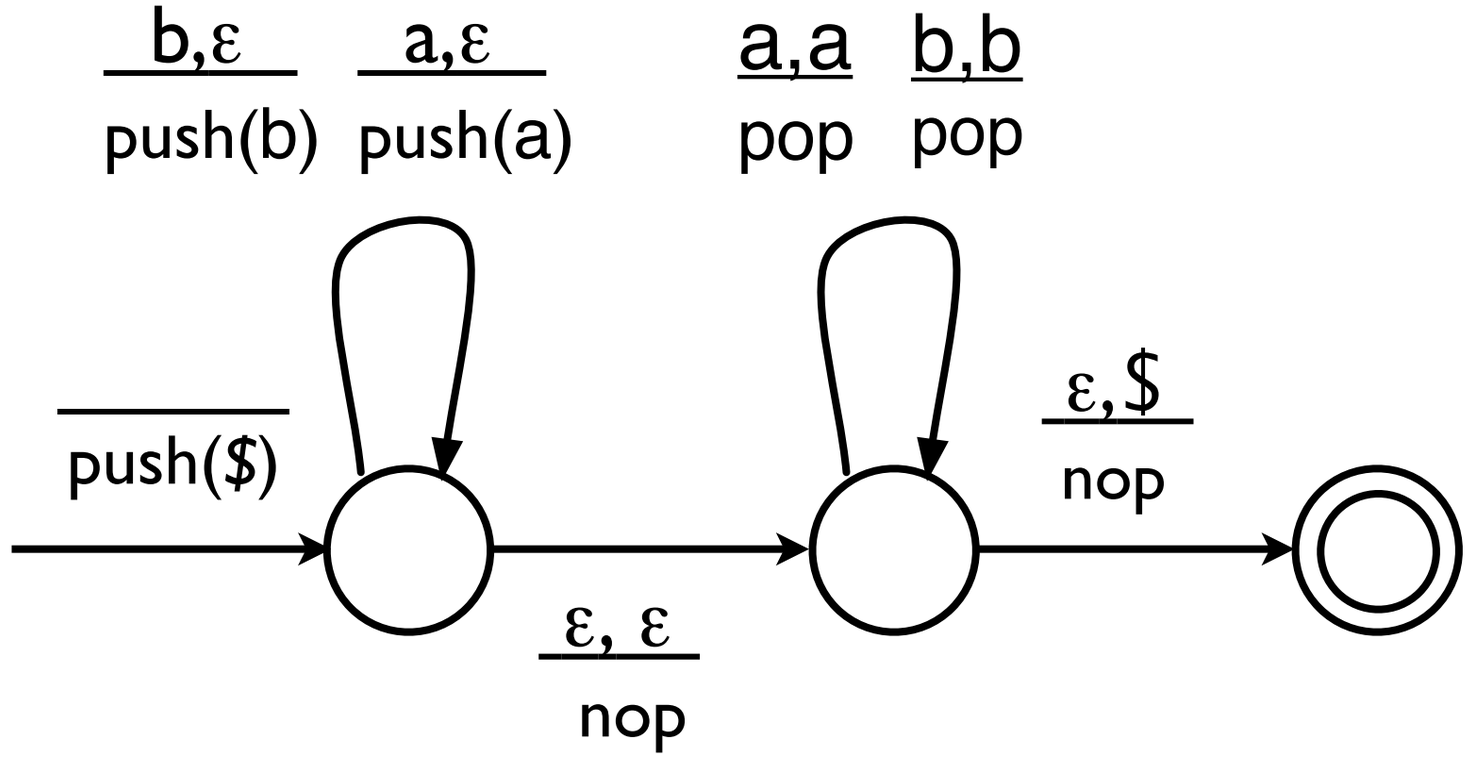
$L_2 = \{ww^R \mid w \in \{a, b\}^*\}$



- This machine is non-deterministic. (Why?)

Palindromes of even length

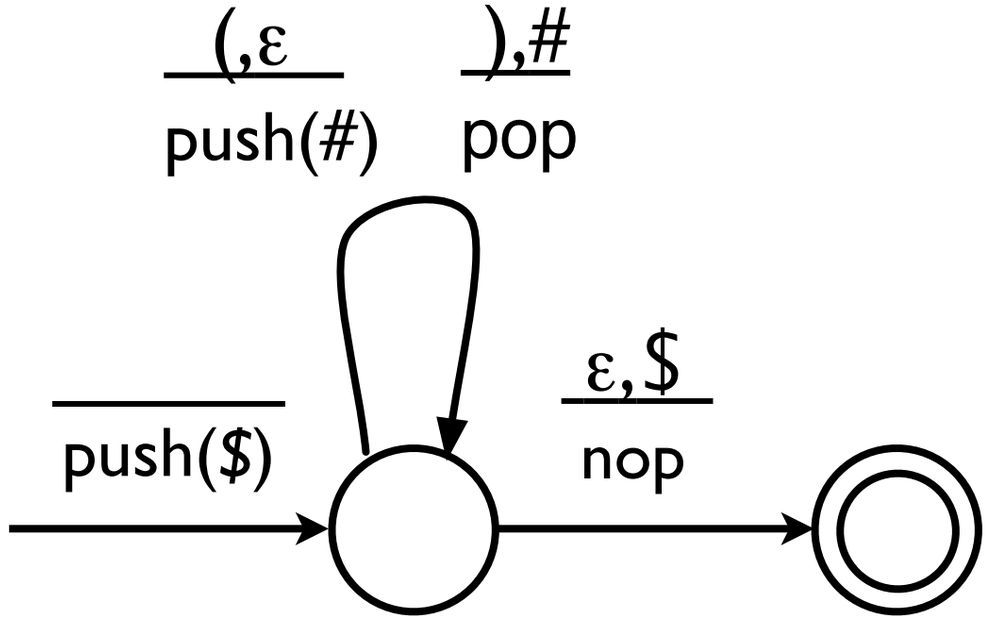
$L_2 = \{ww^R \mid w \in \{a, b\}^*\}$



- This machine is non-deterministic. (Why?)
- Can you design a *deterministic* PDA for L_2?

Balanced Parenthesis

$L_3 = \{w \in \{(,)\}^* : w \text{ is a balanced string of parenthesis}\}$



Equal numbers of *a*s and *b*s