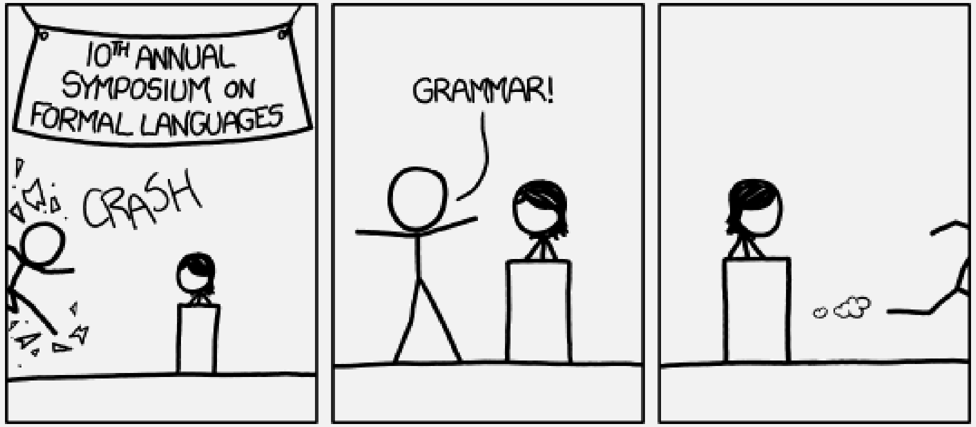
UMB CS 420

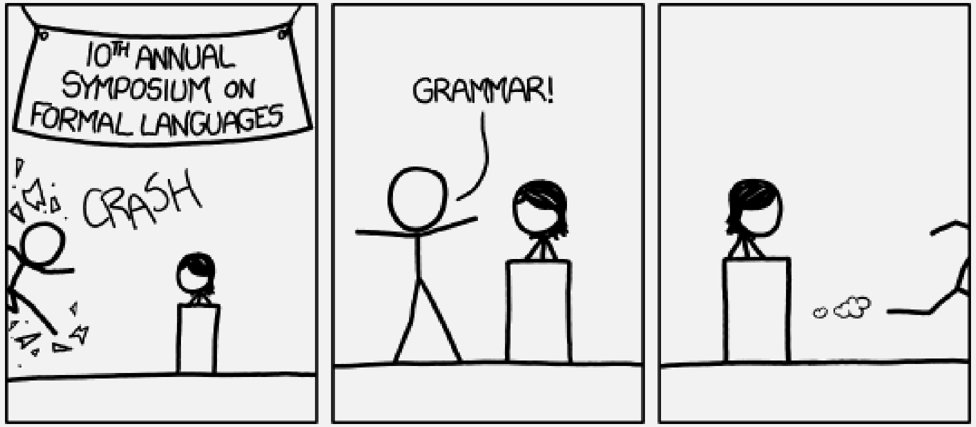
Pushdown Automata (PDAs)

Wednesday, March 20, 2024



Announcements

- HW 5 out
  - Due Mon 3/25 12pm noon



Last Time:

Context-Free Grammar (CFG)

Grammar $G_1 = (V, \Sigma, R, S)$

terminals

Top variable is:
Start variable

Variables
(a.k.a., **non-terminals**)

$$\begin{aligned} A &\rightarrow 0A1 \\ A &\rightarrow B \\ B &\rightarrow \# \end{aligned}$$

R is this <u>set</u> of rules (var-string pairs):

Substitution rules
(a.k.a., **productions**)

terminals (analogous to DFA's alphabet)

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

1. V is a finite set called the ***variables***,
2. Σ is a finite set, disjoint from V, called the ***terminals***,
3. R is a finite set of ***rules***, with each rule being a variable and a string of variables and terminals, and
4. $S \in V$ is the start variable.

$V = \{A, B\}$,

$\Sigma = \{0, 1, \#\}$,

$S = A$,

Last Time:

Generating Strings with a CFG

Grammar $G_1 = (V, \Sigma, R, S)$

$$A \rightarrow 0A1$$
$$A \rightarrow B$$
$$B \rightarrow \#$$

Strings in CFG's language
= all possible **generated** / **derived** strings

$$L(G_1) \text{ is } \{0^n\#1^n \mid n \geq 0\}$$

A CFG **generates** a string, by repeatedly applying substitution rules:

Example:

$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000B111 \Rightarrow 000\#111$$

This sequence of steps is called a **derivation**

Last Time:

Derivations: Formally

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

1. V is a finite set called the ***variables***,
2. Σ is a finite set, disjoint from V, called the ***terminals***,
3. R is a finite set of ***rules***, with each rule being a variable and a string of variables and terminals, and
4. $S \in V$ is the start variable.

Let $G = (V, \Sigma, R, S)$

Single-step

$$\alpha A \beta \underset{G}{\Rightarrow} \alpha \gamma \beta$$

Where:

$\alpha, \beta \in (V \cup \Sigma)^*$ ← sequence of terminals or variables

$A \in V$ ← Variable

$A \rightarrow \gamma \in R$ ← Rule

Last Time:

Derivations: Formally

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

1. V is a finite set called the ***variables***,
2. Σ is a finite set, disjoint from V, called the ***terminals***,
3. R is a finite set of ***rules***, with each rule being a variable and a string of variables and terminals, and
4. $S \in V$ is the start variable.

Let $G = (V, \Sigma, R, S)$

Single-step

$$\alpha A \beta \underset{G}{\Rightarrow} \alpha \gamma \beta$$

Where:

$\alpha, \beta \in (V \cup \Sigma)^*$ ← sequence of terminals or variables

$A \in V$ ← Variable

$A \to \gamma \in R$ ← Rule

Multi-step (recursively defined)

Base case: $\alpha \overset{*}{\underset{G}{\Rightarrow}} \alpha$ (0 steps)

Recursive case: (> 0 steps)

$$\alpha \overset{*}{\underset{G}{\Rightarrow}} \gamma$$

Where: $\alpha \underset{G}{\Rightarrow} \beta$ and $\beta \overset{*}{\underset{G}{\Rightarrow}} \gamma$

Single step

(smaller) Recursive "call"

Last Time:

Formal Definition of a CFL

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

1. V is a finite set called the ***variables***,
2. Σ is a finite set, disjoint from V, called the ***terminals***,
3. R is a finite set of ***rules***, with each rule being a variable and a string of variables and terminals, and
4. $S \in V$ is the start variable.

$$G = (V, \Sigma, R, S)$$

"... all possible sequences of terminal symbols (i.e., **strings**) ..."

"... that can be **generated** with rules of grammar G"

"the **language** of a grammar G is ..."

$$L(G) = \left\{ w \in \Sigma^* \mid S \overset{*}{\underset{G}{\Rightarrow}} w \right\}$$

If a **CFG** generates all strings in a **language** L, then L is a **context-free language** (CFL)

Last Time:

Designing Grammars : Basics

1. Think about what you want to “link” together

- E.g., $0^n 1^n$
  - $A \rightarrow 0A1$
  - \# 0s and # 1s are “linked”

- E.g., XML
  - $\text{ELEMENT} \rightarrow \text{<TAG>CONTENT</TAG>}$
  - **Start** and **end tags** are “linked”

2. **Start with small grammars** and **then combine**
   - just like with FSMs, and programming!

Example: Creating CFG

alphabet Σ is $\{0,1\}$

$$\{w \mid w \text{ starts and ends with the same symbol}\}$$

1) come up with examples: In the language: **010, 101, 11011** **1, 0 ?** ☑

Not in the language: **10, 01, 110** ε ? ☒

2) Create CFG:

Needed Rules:

$S \rightarrow 0M0 \mid 1M1 \mid 0 \mid 1$

$M \rightarrow MT \mid \varepsilon$

$T \rightarrow 0 \mid 1$

"**start/end symbol** are "**linked**" (ie, **same**); **middle** can be **anything**"

"**middle**: **all possible terminals**, repeated (ie, **all possible strings**)"

"**all possible terminals**"

3) Check CFG: **generates examples** in the language; **does not generate examples** not in language

Regular Language vs CFL Comparison

Regular Languages	Context-Free Languages (CFLs)
Regular Expression	Context-Free Grammar (CFG)
describes a Regular Lang	describes a CFL

Regular Language vs CFL Comparison

Regular Languages	**Context-Free Languages (CFLs)**
Regular Expression	Context-Free Grammar (CFG)
describes a Regular Lang	describes a CFL
Finite State Automaton (FSM)	???
recognizes a Regular Lang	recognizes a CFL

Regular Language vs CFL Comparison

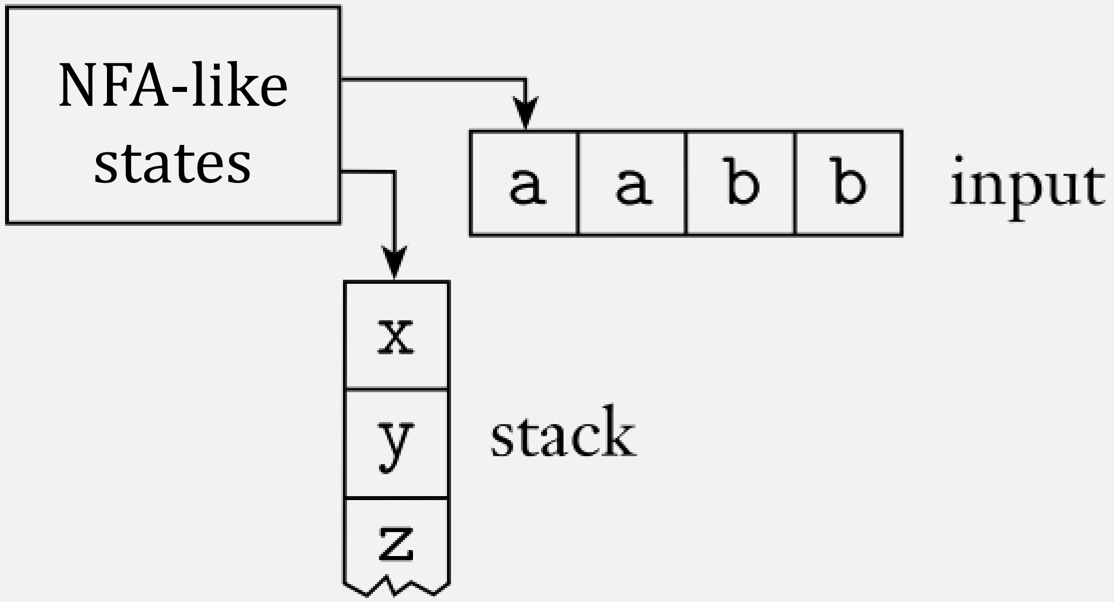
	Regular Languages	**Context-Free Languages (CFLs)**	
thm	Regular Expression describes a **Regular Lang**	Context-Free Grammar (CFG) describes a **CFL**	def
def	Finite State Automaton (FSM) recognizes a **Regular Lang**	**Push-down Automata** (PDA) recognizes a **CFL**	thm

Regular Language vs CFL Comparison

Regular Languages	Context-Free Languages (CFLs)
thm Regular Expression describes a Regular Lang	Context-Free Grammar (CFG) describes a CFL def
def Finite State Automaton (FSM) recognizes a Regular Lang	**Push-down Automata** (PDA) recognizes a CFL thm
Proved:	Proved:
Regular Lang ⇔ Regular Expr	CFL ⇔ PDA

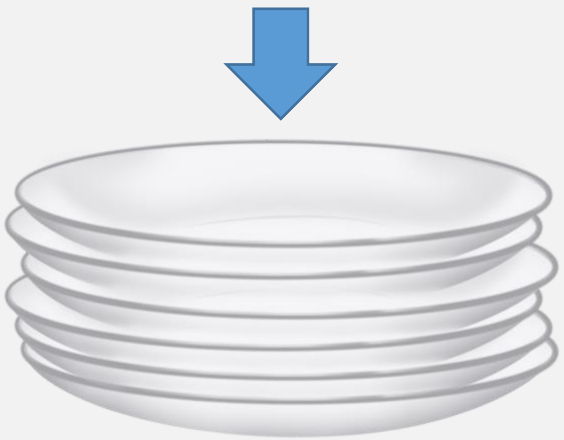
Pushdown Automata (PDA)

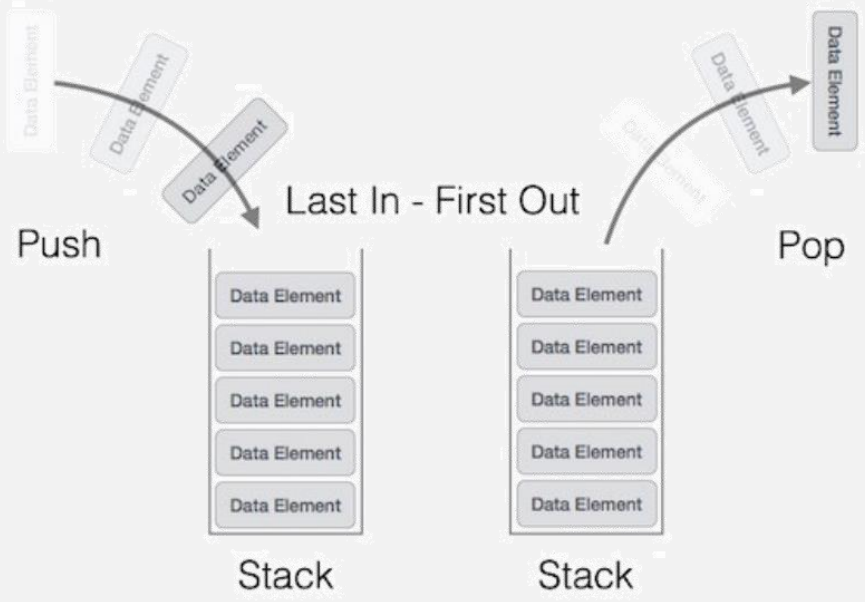
PDA = NFA + a <u>stack</u>



What is a Stack?

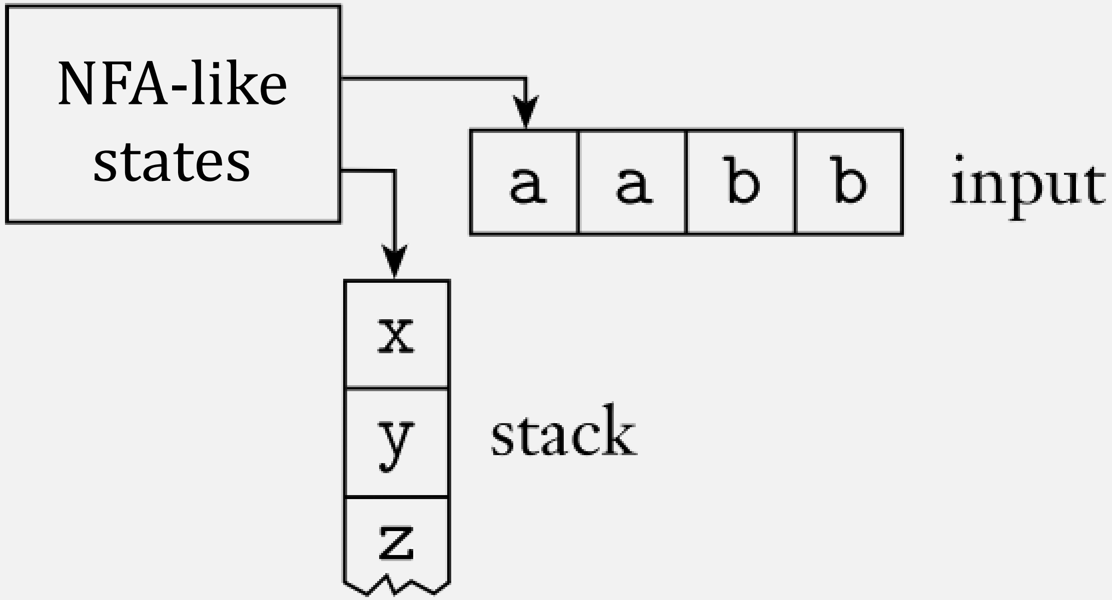
- A **restricted** kind of (**infinite!**) **memory**
- **Access** to **top element only**
- 2 Operations only: **push, pop**





Pushdown Automata (PDA)

- **PDA** = NFA + a **stack**
  - Infinite memory
  - read/write top location only
    - Push/pop

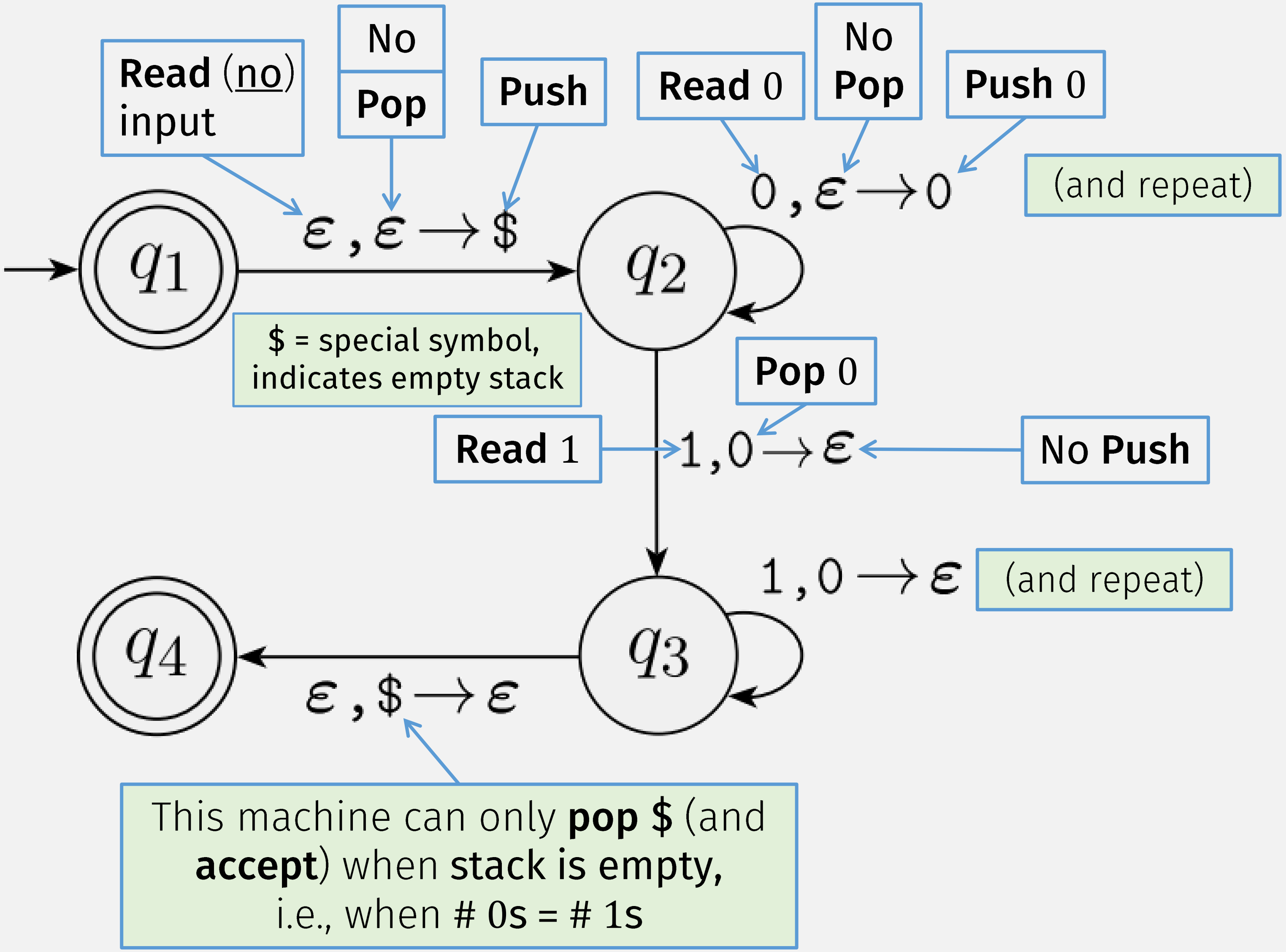


$\{0^n 1^n | \, n \geq 0\}$

An Example PDA

A PDA transition has 3 parts:

- **Read**
- **Pop**
- **Push**



Formal Definition of PDA

A ***pushdown automaton*** is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where Q, Σ, Γ, and F are all finite sets, and

1. Q is the set of states,

2. Σ is the input alphabet,

3. Γ is the stack alphabet,

Stack alphabet has special stack symbols, e.g., $

4. $\delta \colon Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$ is the transition function,

Input

Pop

Push

5. $q_0 \in Q$ is the start state, and

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

Non-deterministic!
Result of a step is **set** of (STATE, STACK CHAR) pairs

PDA Formal Definition Example

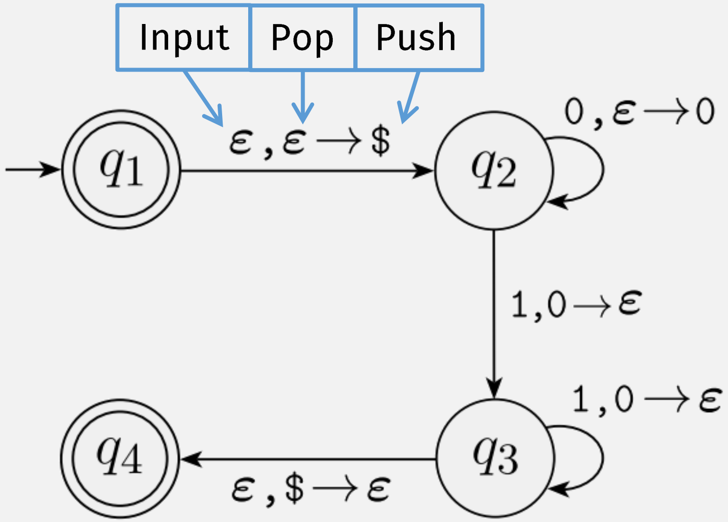
$Q = \{q_1, q_2, q_3, q_4\}$,

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

$\Gamma = \{0, \$\}$,

Stack alphabet has special stack symbol $

$F = \{q_1, q_4\}$,



A ***pushdown automaton*** is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where Q, Σ, Γ, and F are all finite sets, and

1. Q is the set of states,
2. Σ is the input alphabet,
3. Γ is the stack alphabet,
4. $\delta \colon Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$ is the transition function,
5. $q_0 \in Q$ is the start state, and
6. $F \subseteq Q$ is the set of accept states.

Input
Pop
Push

$Q = \{q_1, q_2, q_3, q_4\}$,

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

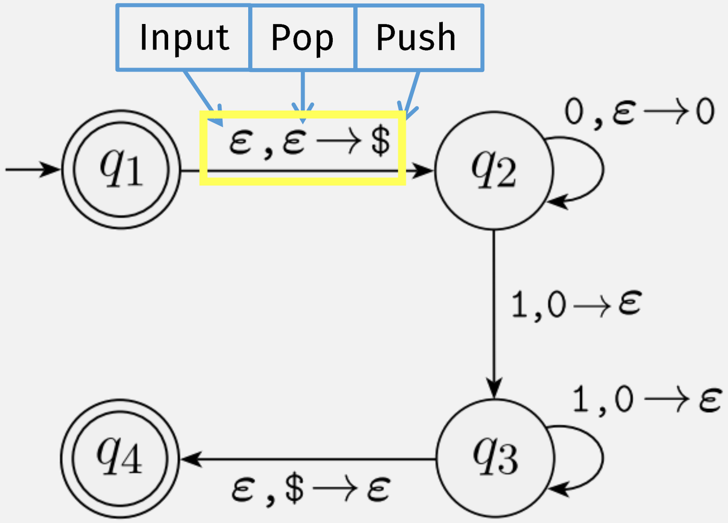
$\Gamma = \{0, \$\}$,

$F = \{q_1, q_4\}$, and

δ is given by the following table, wherein blank entries signify $\emptyset$.

Input:	0			1			ε		
Stack:	0	\$	ε	0	\$	ε	0	\$	ε
q_1									$\{(q_2, \$)\}$ **5**
q_2			$\{(q_2, 0)\}$ **1**	$\{(q_3, \varepsilon)\}$ **2**					
q_3				$\{(q_3, \varepsilon)\}$ **3**				$\{(q_4, \varepsilon)\}$ **4**	
q_4									

Input — Pop — Push



A ***pushdown automaton*** is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where Q, Σ, Γ, and F are all finite sets, and

1. Q is the set of states,
2. Σ is the input alphabet,
3. Γ is the stack alphabet,
4. $\delta \colon Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$ is the transition function,
5. $q_0 \in Q$ is the start state, and
6. $F \subseteq Q$ is the set of accept states.

Input — Pop — Push

$Q = \{q_1, q_2, q_3, q_4\}$,

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

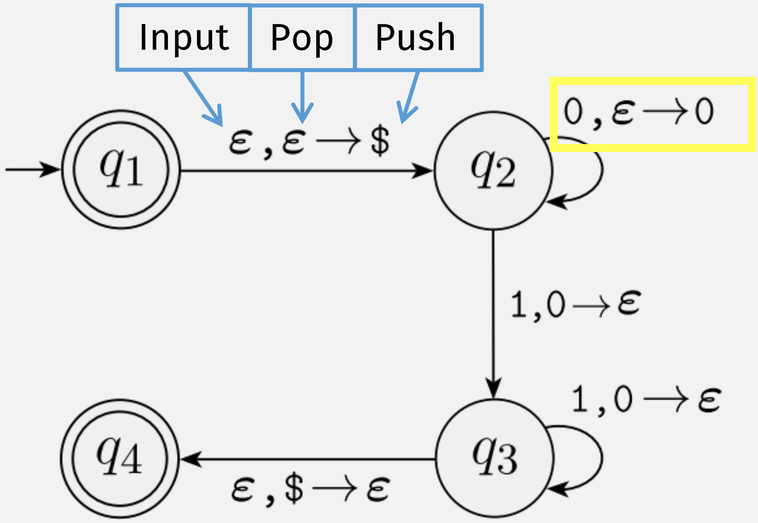
$\Gamma = \{0, \$\}$,

$F = \{q_1, q_4\}$, and

δ is given by the following table, wherein blank entries signify $\emptyset$.

Input:	0			1			ε		
Stack:	0	\$	ε	0	\$	ε	0	\$	ε
q_1									$\{(q_2, \$)\}$
q_2			$\{(q_2, 0)\}$ 1	$\{(q_3, \varepsilon)\}$ 2				4	5
q_3				$\{(q_3, \varepsilon)\}$ 3				$\{(q_4, \varepsilon)\}$	
q_4									

Input
Pop
Push



A ***pushdown automaton*** is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where Q, Σ, Γ, and F are all finite sets, and

1. Q is the set of states,
2. Σ is the input alphabet,
3. Γ is the stack alphabet,
4. $\delta \colon Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$ is the transition function,
5. $q_0 \in Q$ is the start state, and
6. $F \subseteq Q$ is the set of accept states.

Input
Pop
Push

$Q = \{q_1, q_2, q_3, q_4\}$,

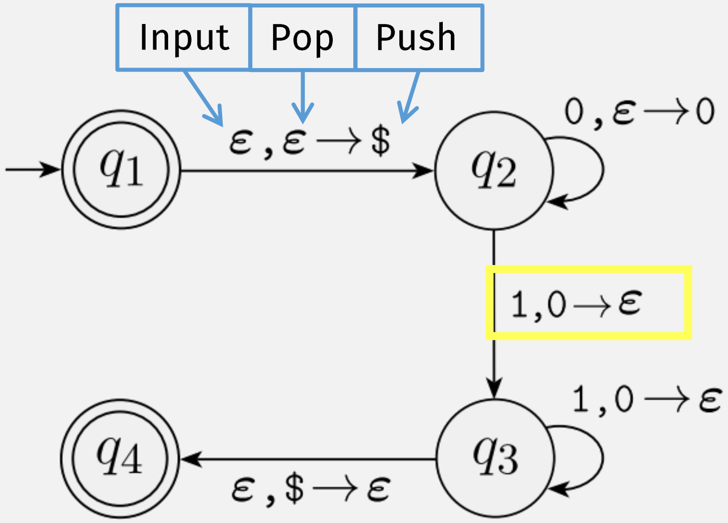
$\Sigma = \{0,1\}$,

$\Gamma = \{0, \$\}$,

$F = \{q_1, q_4\}$, and

δ is given by the following table, wherein blank entries signify $\emptyset$.

Input:	0			1			ε		
Stack:	0	\$	ε	0	\$	ε	0	\$	ε
q_1									$\{(q_2, \$)\}$
q_2			$\{(q_2, 0)\}$	$\{(q_3, \varepsilon)\}$					
q_3				$\{(q_3, \varepsilon)\}$				$\{(q_4, \varepsilon)\}$	
q_4									



A ***pushdown automaton*** is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where Q, Σ, Γ, and F are all finite sets, and

1. Q is the set of states,
2. Σ is the input alphabet,
3. Γ is the stack alphabet,
4. $\delta\colon Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$ is the transition function,
5. $q_0 \in Q$ is the start state, and
6. $F \subseteq Q$ is the set of accept states.

Input Pop Push

$Q = \{q_1, q_2, q_3, q_4\}$,

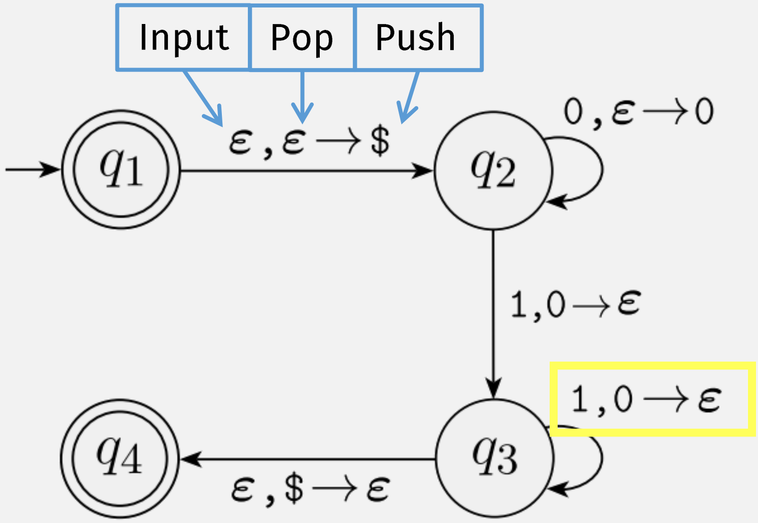
$\Sigma = \{0,1\}$,

$\Gamma = \{0, \$\}$,

$F = \{q_1, q_4\}$, and

δ is given by the following table, wherein blank entries signify $\emptyset$.

Input:	0			1			ε		
Stack:	0	\$	ε	0	\$	ε	0	\$	ε
q_1									$\{(q_2, \$)\}$
q_2			$\{(q_2, 0)\}$	$\{(q_3, \varepsilon)\}$					
q_3				$\{(q_3, \varepsilon)\}$				$\{(q_4, \varepsilon)\}$	
q_4									



A ***pushdown automaton*** is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where Q, Σ, Γ, and F are all finite sets, and

1. Q is the set of states,
2. Σ is the input alphabet,
3. Γ is the stack alphabet,
4. $\delta \colon Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$ is the transition function,
5. $q_0 \in Q$ is the start state, and
6. $F \subseteq Q$ is the set of accept states.

$Q = \{q_1, q_2, q_3, q_4\}$,

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

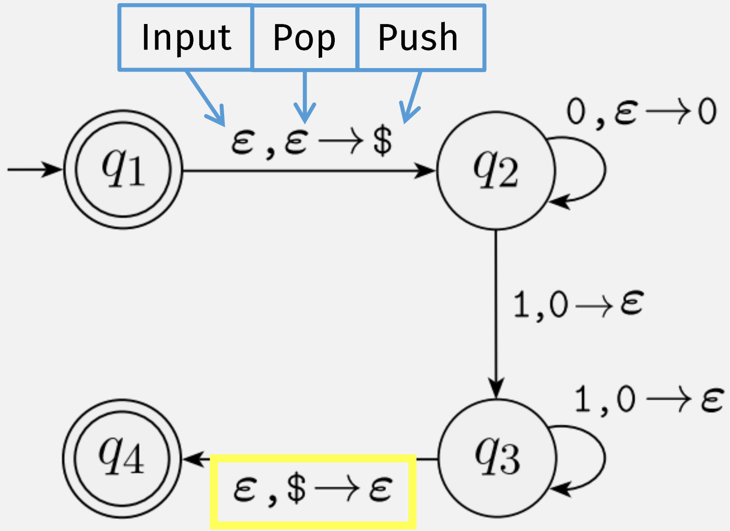
$\Gamma = \{0, \$\}$,

$F = \{q_1, q_4\}$, and

δ is given by the following table, wherein blank entries signify $\emptyset$.

Input:	0			1			ε		
Stack:	0	\$	ε	0	\$	ε	0	\$	ε
q_1									$\{(q_2, \$)\}$ 5
q_2			$\{(q_2, 0)\}$ 1	$\{(q_3, \varepsilon)\}$ 2					
q_3				$\{(q_3, \varepsilon)\}$ 3				$\{(q_4, \varepsilon)\}$ 4	
q_4									

Input
Pop
Push



A ***pushdown automaton*** is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where Q, Σ, Γ, and F are all finite sets, and

1. Q is the set of states,
2. Σ is the input alphabet,
3. Γ is the stack alphabet,
4. $\delta \colon Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$ is the transition function,
5. $q_0 \in Q$ is the start state, and
6. $F \subseteq Q$ is the set of accept states.

Input
Pop
Push

In-class exercise: Fill in the blanks

$Q =$

$\Sigma =$

$\Gamma =$

$F =$

δ is given by the following table, wherein blank entries signify $\emptyset$.

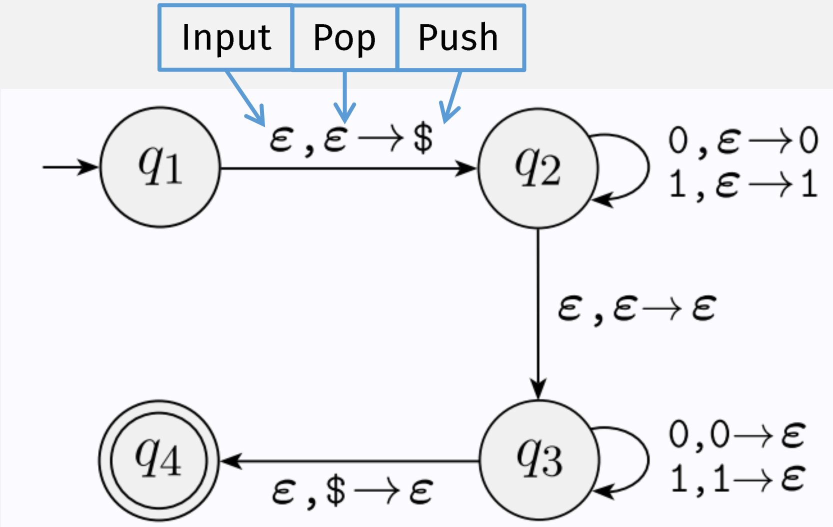
Input:	0			1			ε		
Stack:	0	$	ε	0	$	ε	0	$	ε

Input

Pop

State/ Push

PDA M_3 recognizing the language $\{ww^{\mathcal{R}} | w \in \{0,1\}^*\}$



A ***pushdown automaton*** is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where Q, Σ, Γ, and F are all finite sets, and

1. Q is the set of states,
2. Σ is the input alphabet,
3. Γ is the stack alphabet,
4. $\delta \colon Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$ is the transition function,
5. $q_0 \in Q$ is the start state, and
6. $F \subseteq Q$ is the set of accept states.

Input

Pop

Push

In-class exercise:
Fill in the blanks

$Q = \{q_1, q_2, q_3, q_4\}$,

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

$\Gamma = \{0,1,\$\}$,

$F = \{q_4\}$

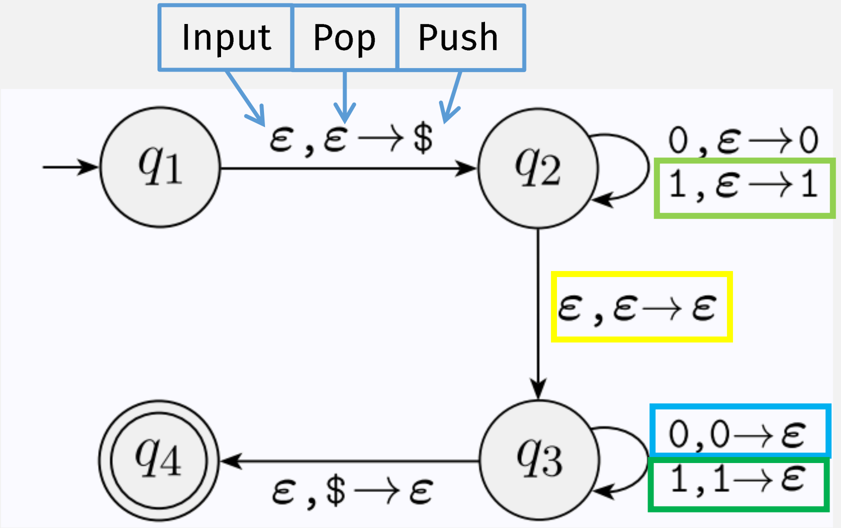
δ is given by the following table, wherein blank entries signify $\emptyset$.

Input:	0			1				ε		
Stack:	0	\$	ε	0	1	\$	ε	0	\$	ε
q_1										$\{(q_2,\$)\}$
q_2			$\{(q_2,0)\}$				$\{(q_2,1)\}$			$\{(q_3,\varepsilon)\}$
q_3	$\{(q_3,\varepsilon)\}$				$\{(q_3,\varepsilon)\}$				$\{(q_4,\varepsilon)\}$	
q_4										

Input

Pop

State/ Push



PDA M_3 recognizing the language $\{ww^{\mathcal{R}} \mid w \in \{0,1\}^*\}$

Flashback

DFA Computation Rules

Informally	*Formally (i.e., mathematically)*

Given

- A **DFA** (~ a "Program")
- and **Input = string of chars**, e.g. "1101"

- $M = (Q, \Sigma, \delta, q_0, F)$
- $w = w_1 w_2 \cdots w_n$

A **DFA** <u>**computation**</u> (~ "Program run"):

- <u>Start</u> in *start state*

A **DFA computation** is a <u>sequence of states:</u>

- <u>Repeat:</u>
  - <u>Read</u> 1 char from **Input**, and
  - <u>Change state</u> according to *transition rules*

- specified by $\hat{\delta}(q_0, w)$ where:

<u>Result</u> of **computation:**

- <u>Accept</u> if last state is *Accept state*
- <u>Reject</u> otherwise

- *M* **accepts** *w* if $\hat{\delta}(q_0, w) \in F$
- *M* **rejects** otherwise

Flashback

$\delta : Q \times \Sigma \longrightarrow Q$ is the ***transition function***

DFA Multi-step Transition Function

$$\hat{\delta} : Q \times \Sigma^* \to Q$$

- <u>Domain</u> (inputs):
  - state $q \in Q$
  - string $w = w_1 w_2 \cdots w_n$ where $w_i \in \Sigma$
- <u>Range</u> (output):
  - state $q \in Q$

A DFA **computation** is a <u>sequence of states:</u>

(Defined recursively)

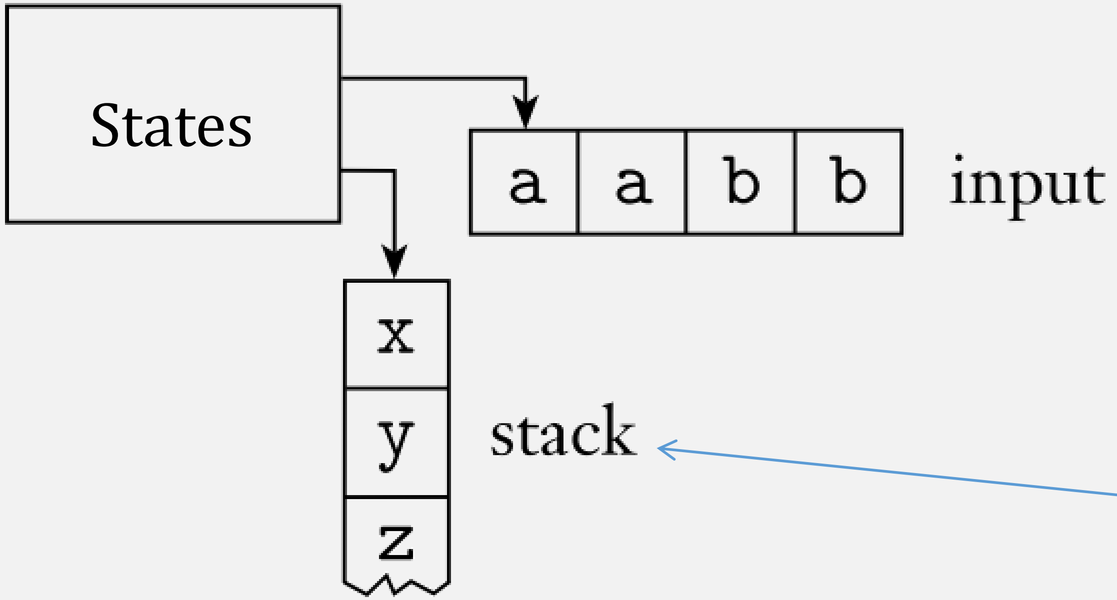
Base case $\hat{\delta}(q, \varepsilon) = q$

Recursive Case $\hat{\delta}(q, w' w_n) = \delta(\hat{\delta}(q, w'), w_n)$

where $w' = w_1 \cdots w_{n-1}$

PDA Computation?

- **PDA** = NFA + a stack
  - Infinite memory
  - Push/pop top location only



A DFA **computation** is a <u>sequence of states</u> …

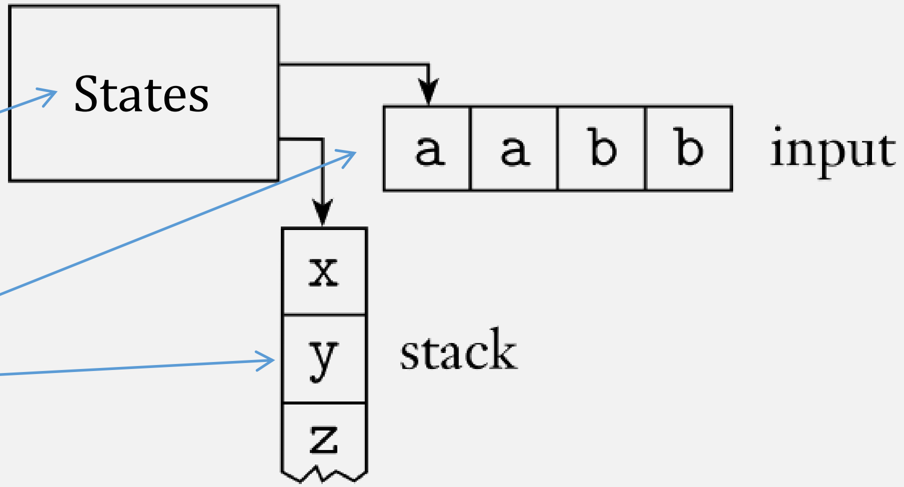
A PDA **computation** is a <u>not</u> just a <u>sequence of states</u> …

… because the **stack contents can change too!**

PDA Configurations (IDs)

- A **configuration** (or **ID**) is a "snapshot" of a PDA's computation

- 3 components (q, w, γ) :
  - q = the current state
  - w = the remaining input string
  - γ = the stack contents



A **sequence of configurations** represents a **PDA computation**

PDA Computation, Formally

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

Single-step

Before / After **configurations**

$$(q_1, aw, X\beta) \vdash (q_2, w, \alpha\beta)$$

Read Input | Pop | Less 1 char | Push

$$\text{if } \delta(q_1, a, X) \text{ contains } (q_2, \alpha)$$

$$q_1, q_2 \in Q$$

$$a \in \Sigma \quad w \in \Sigma^*$$

$$X \in \Gamma \quad \beta, \alpha \in \Gamma^*$$

A **configuration** (q, w, γ) has three components

- q = the current state
- w = the remaining input string
- γ = the stack contents

Multi-step

- Base Case — 0 steps

$$I \overset{*}{\vdash} I \text{ for any ID } I$$

- Recursive Case — > 0 steps

$$I \overset{*}{\vdash} J \text{ if there exists some ID } K$$

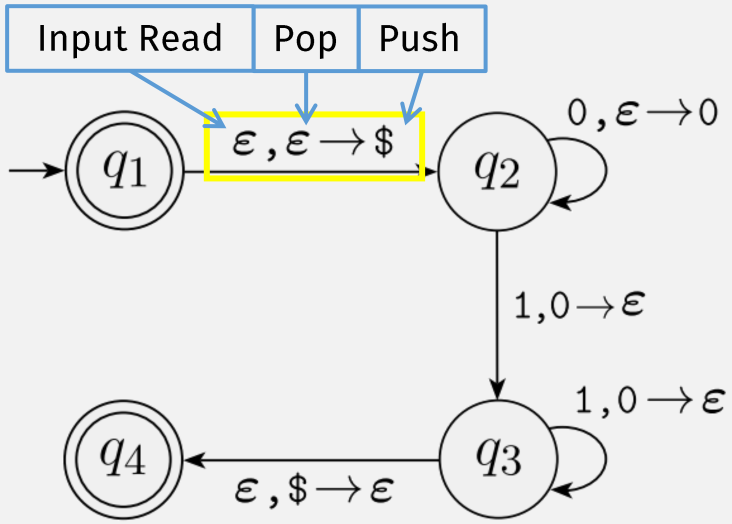
$$\text{such that } I \vdash K \text{ and } K \overset{*}{\vdash} J$$

Single step | Recursive "call"

This specifies the **sequence of configurations** for a **PDA** computation

PDA Running Input String Example

$$(q_1, 0011, \varepsilon)$$



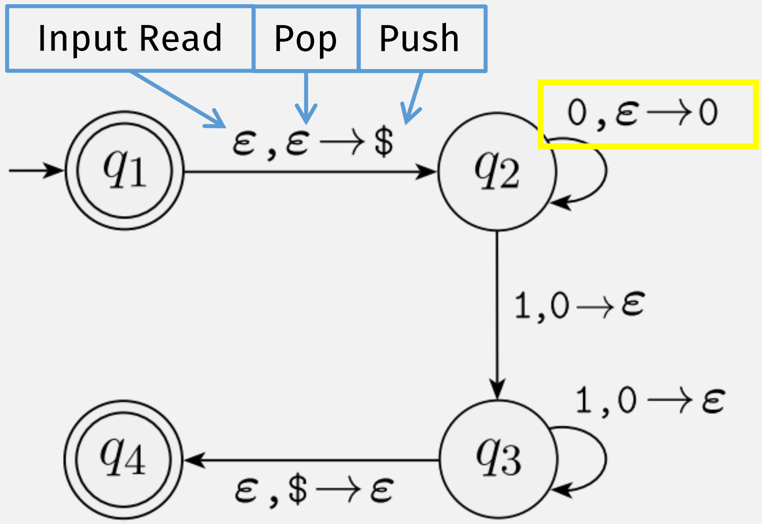
PDA Running Input String Example

State | Remaining Input | Stack

$$(q_1, 0011, \varepsilon) \vdash (q_2, 0011, \$)$$

$$\vdash (q_2, 011, 0\$)$$

Read 0, push 0



PDA Running Input String Example

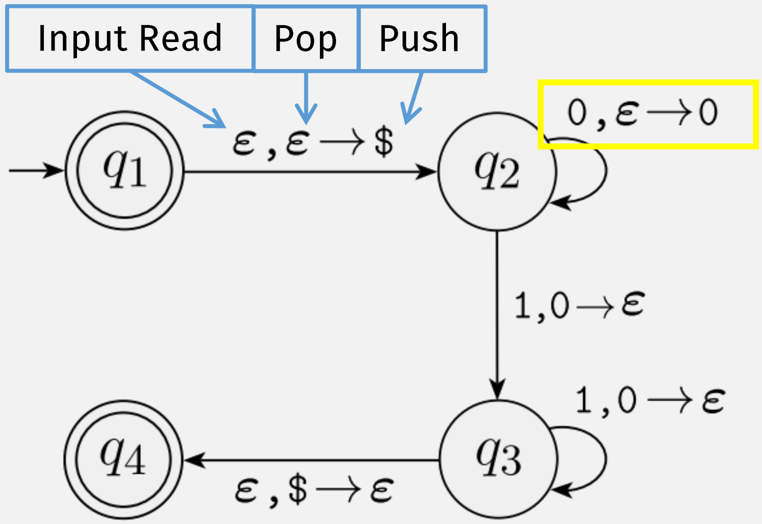
State | Remaining Input | Stack

$$(q_1, 0011, \varepsilon) \vdash (q_2, 0011, \$)$$

$$\vdash (q_2, 011, 0\$)$$

$$\vdash (q_2, 11, 00\$)$$

Read 0, push 0



PDA Running Input String Example

State | Remaining Input | Stack

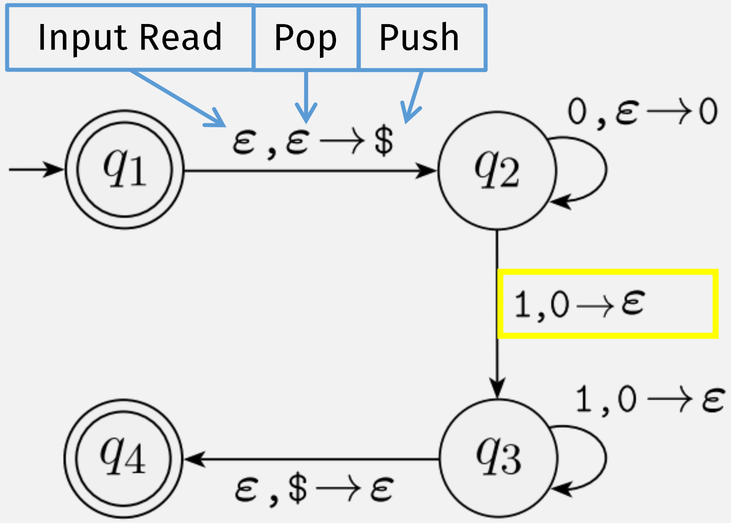
$$(q_1, 0011, \varepsilon) \vdash (q_2, 0011, \$)$$

$$\vdash (q_2, 011, 0\$)$$

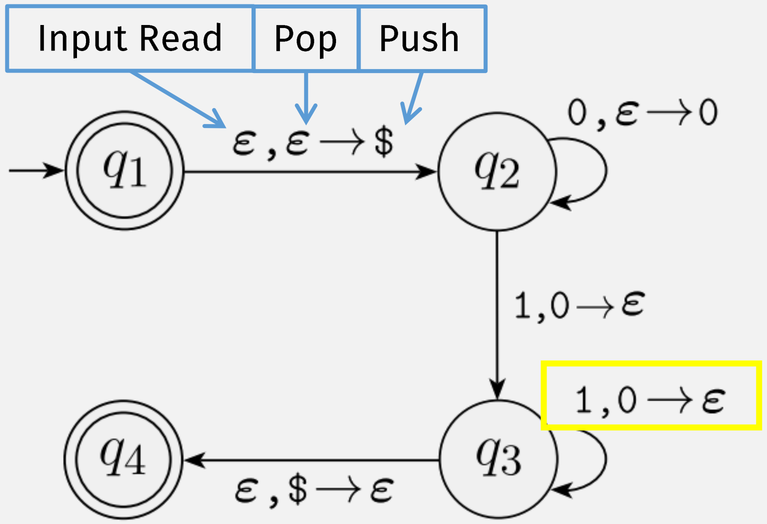
$$\vdash (q_2, 11, 00\$)$$

$$\vdash (q_3, 1, 0\$)$$

Read 1, pop 0



PDA Running Input String Example

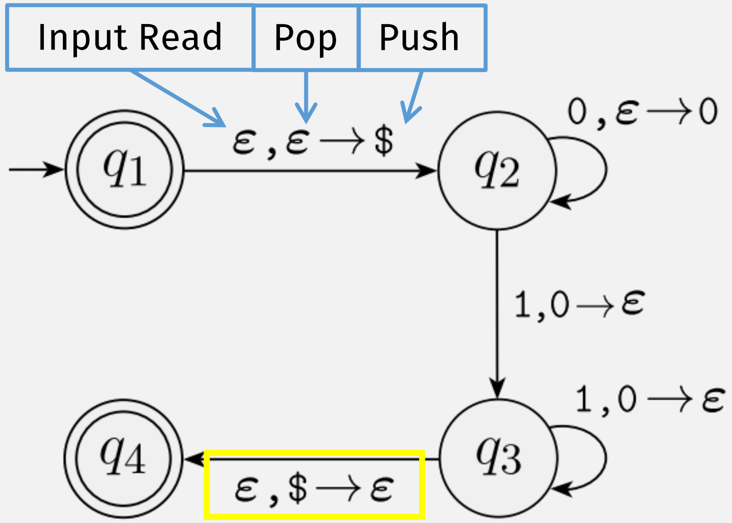


State | Remaining Input | Stack

$$(q_1, 0011, \varepsilon) \vdash (q_2, 0011, \$)$$
$$\vdash (q_2, 011, 0\$)$$
$$\vdash (q_2, 11, 00\$)$$
$$\vdash (q_3, 1, 0\$)$$
$$\vdash (q_3, \varepsilon, \$)$$

Read 1, pop 0

PDA Running Input String Example



State | Remaining Input | Stack

$$
\begin{aligned}
(q_1, 0011, \varepsilon) &\vdash (q_2, 0011, \$) \\
&\vdash (q_2, 011, 0\$) \\
&\vdash (q_2, 11, 00\$) \\
&\vdash (q_3, 1, 0\$) \\
&\vdash (q_3, \varepsilon, \$) \\
&\vdash (q_4, \varepsilon, \varepsilon)
\end{aligned}
$$

pop empty stack symbol

Flashback: Computation and Languages

- The **language** of a **machine** is the **set of all strings that it accepts**
- E.g., A **DFA** M **accepts** w **if** $\hat{\delta}(q_0, w) \in F$
- **Language** of $M = L(M) = \{ w \mid M$ **accepts** $w \}$

Language of a PDA

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

Start in start state

Stack initially empty

Computation ends when input is completely read

$$L(P) = \{w \mid (q_0, w, \varepsilon) \vdash^* (q, \varepsilon, \alpha)\} \text{ where } q \in F$$

Machine accepts if final state is accept state

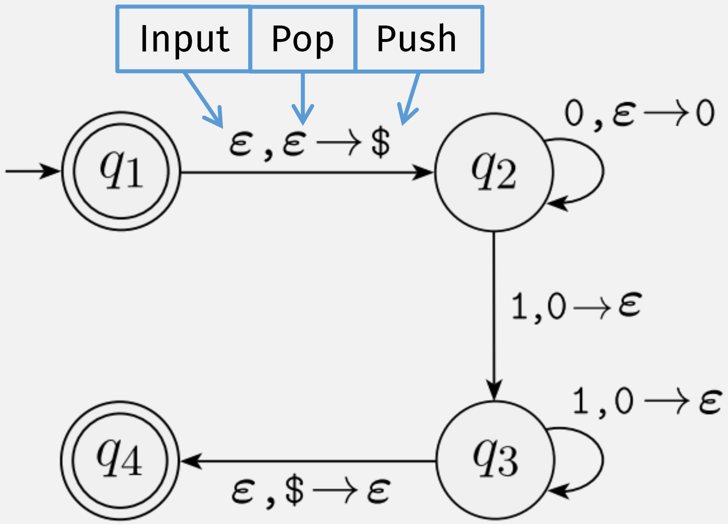
A **configuration** (q, w, γ) has three components

q = the current state

w = the remaining input string

γ = the stack contents

PDAs and CFLs?



- **PDA** = NFA + a stack
  - Infinite memory
  - Push/pop top location only
- Want to prove: PDAs represent CFLs!
- We know: a CFL, by definition, is a language that is generated by a CFG
- Need to show: PDA ⇔ CFG
- Then, to prove that a language is a CFL, we can either:
  - Create a CFG, or
  - Create a PDA

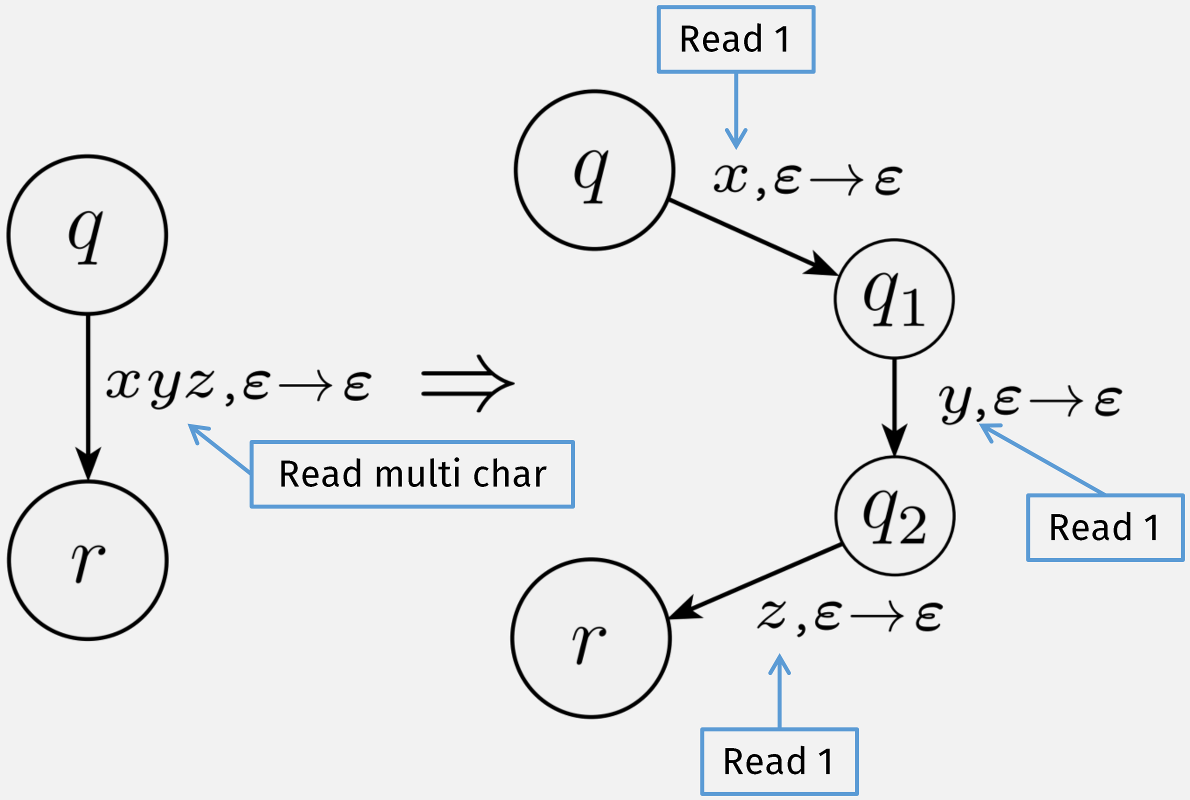
A lang is a CFL **iff** some PDA recognizes it

⇒ If **a language is a** CFL, then **a PDA recognizes it**

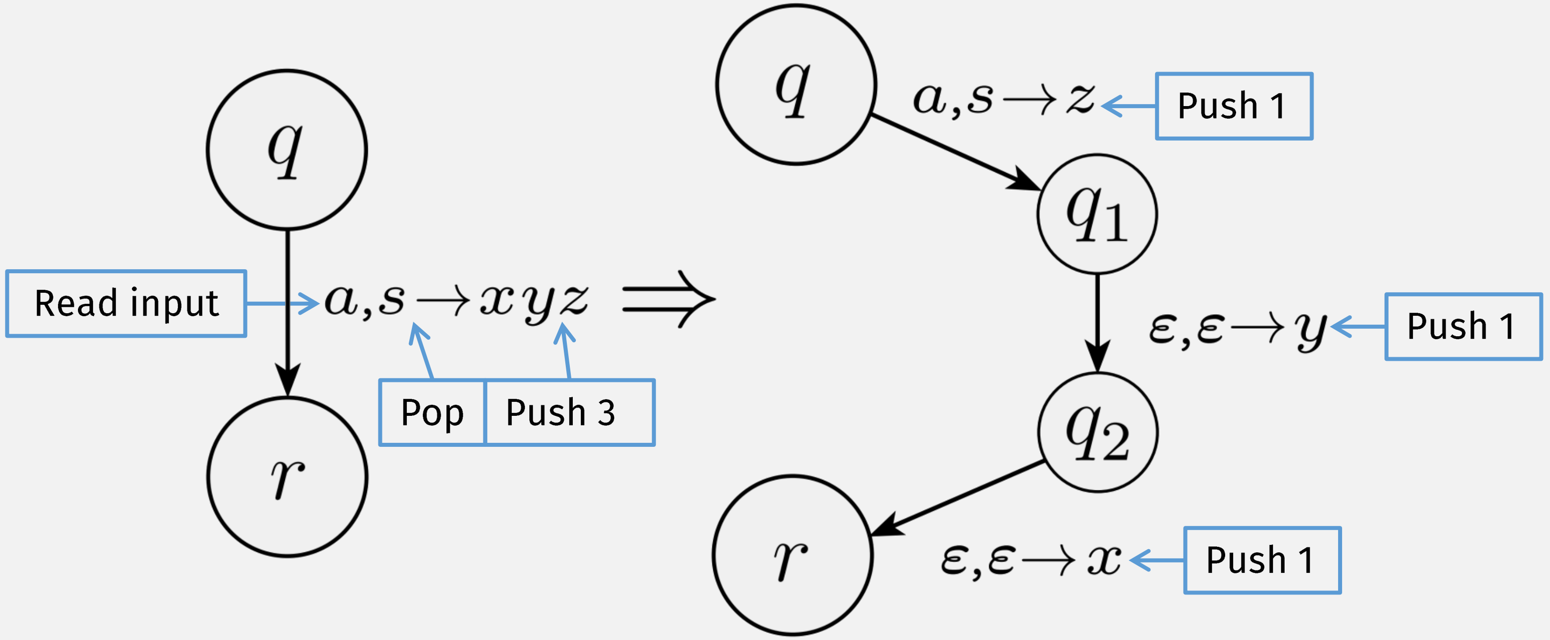
- We know: **A CFL has a CFG describing it** (definition of CFL)
- To prove this part: **show the CFG has an equivalent PDA**

⇐ If **a PDA recognizes a language,** then **it's a CFL**

Shorthand: Multi-Symbol Read Transition



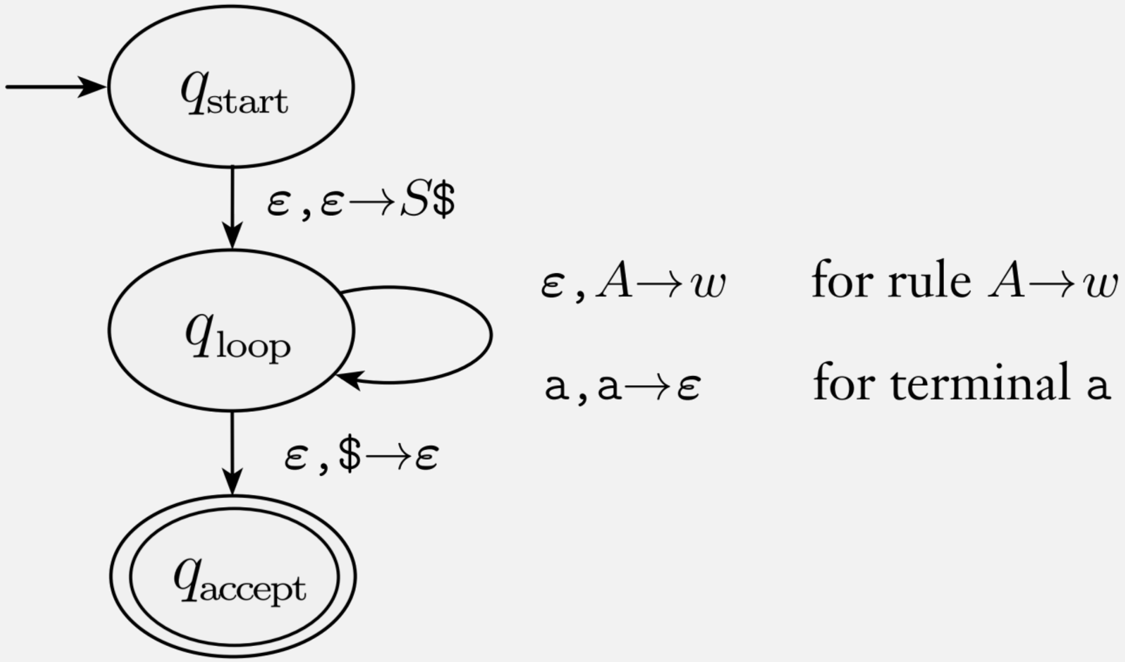
Shorthand: Multi-Stack Push Transition



Note the reverse order of pushes

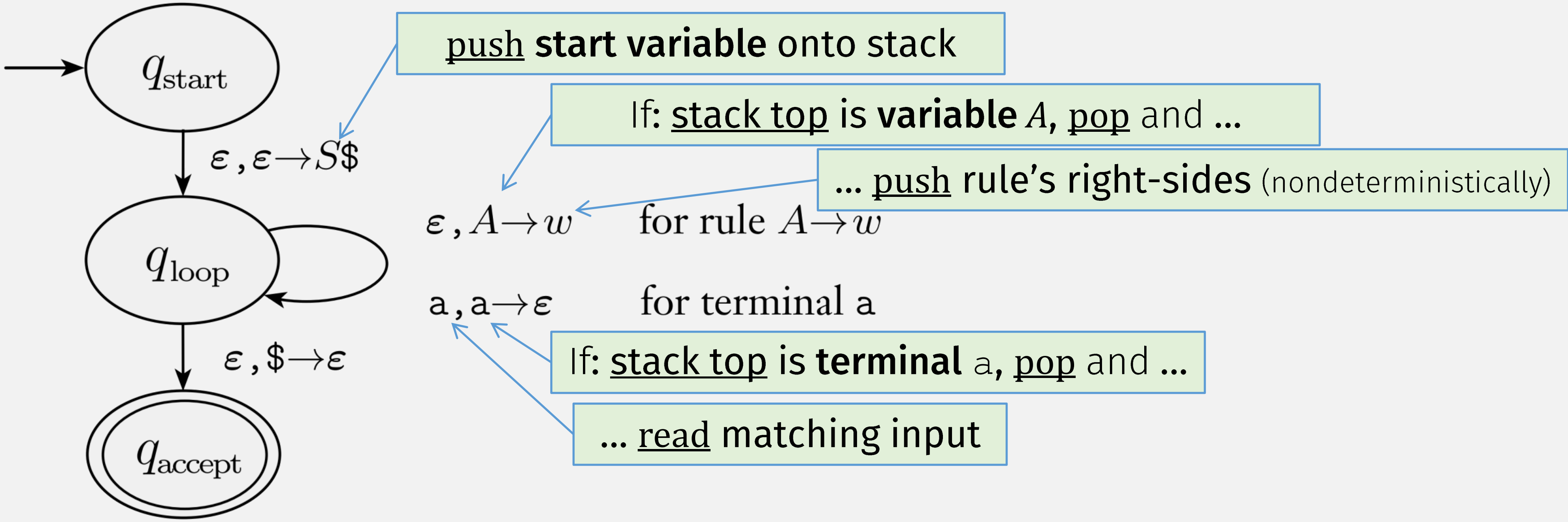
CFG→PDA (sketch)

- Construct PDA from CFG such that:
  - PDA accepts input only if CFG generates it
- PDA:
  - simulates generating a string with CFG rules
  - by (nondeterministically) trying all rules to find the right ones



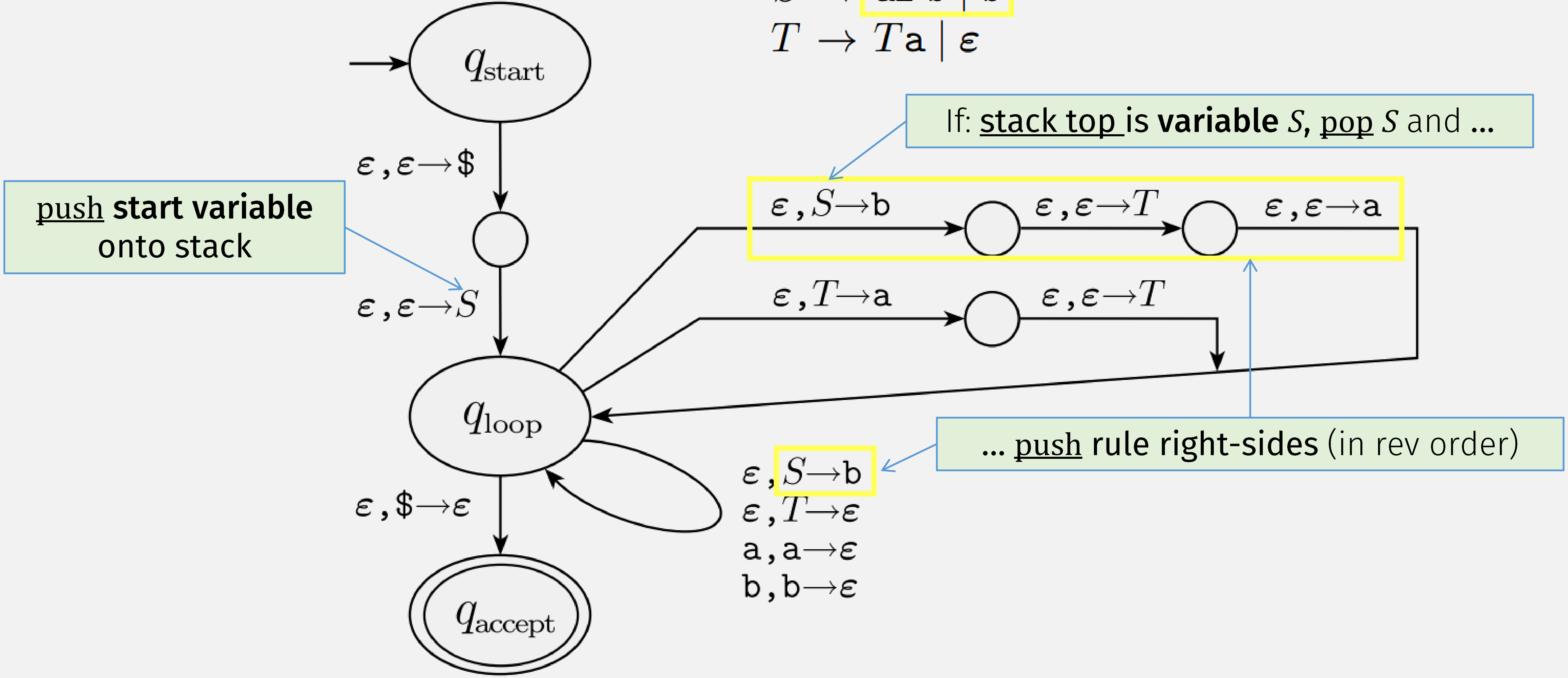
CFG→PDA (sketch)

- **Construct PDA from CFG** such that:
  - PDA accepts input only if CFG generates it
- PDA:
  - simulates generating a string with CFG rules
  - by (nondeterministically) **trying all rules to find the right ones**



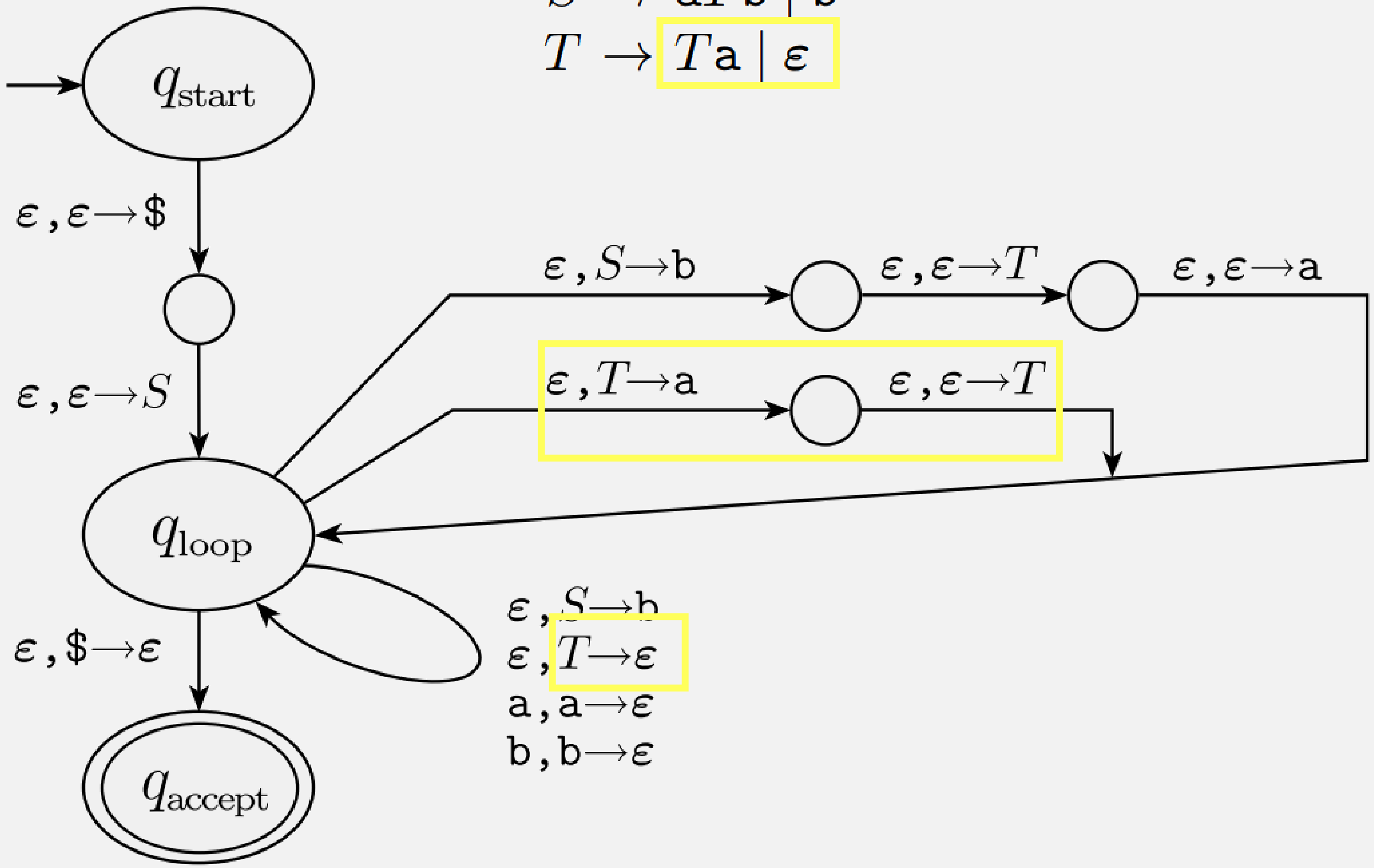
Example **CFG→PDA**

$$S \rightarrow \mathtt{a}T\mathtt{b} \mid \mathtt{b}$$
$$T \rightarrow T\mathtt{a} \mid \varepsilon$$



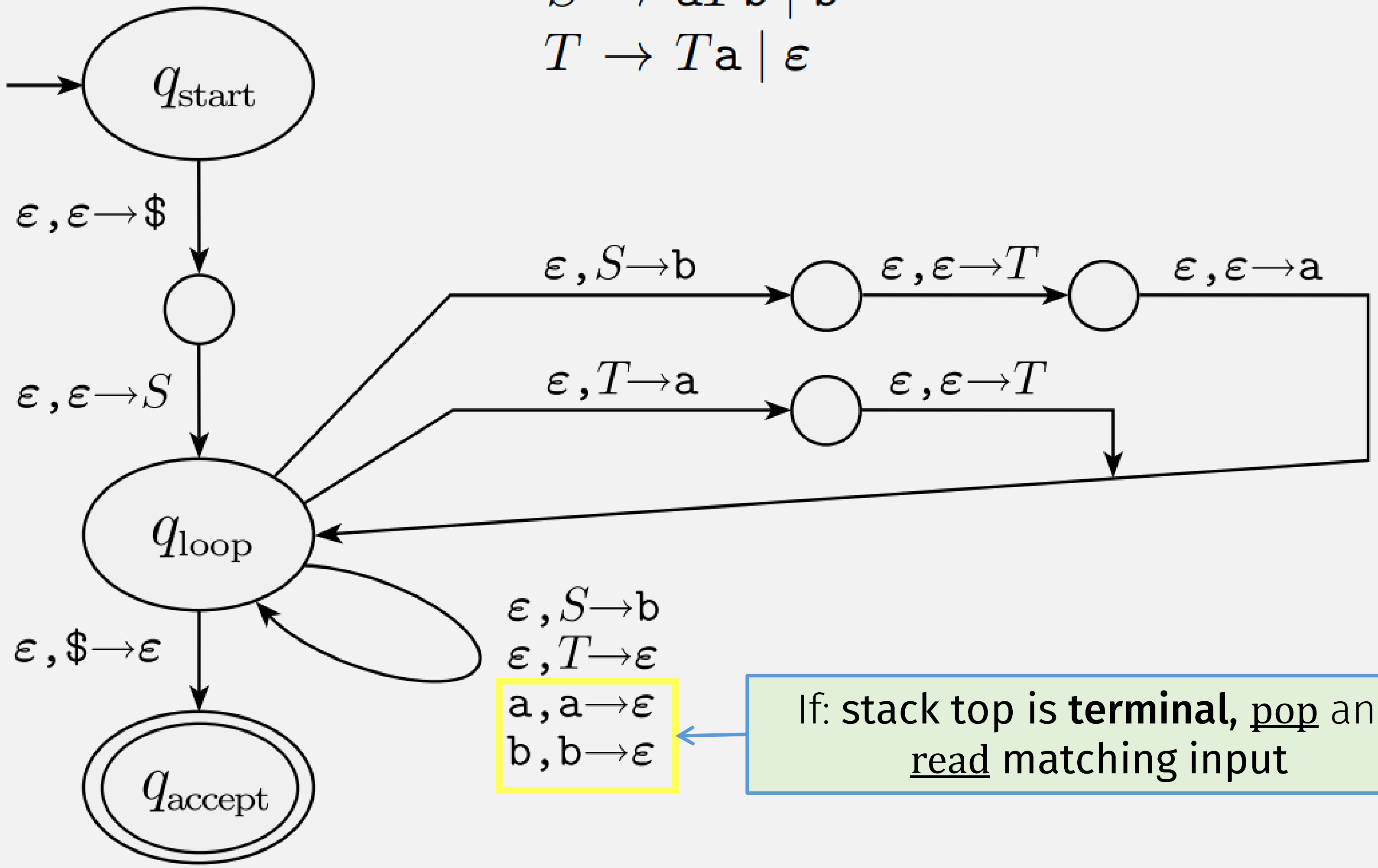
Example **CFG→PDA**

$$S \rightarrow \mathtt{a}T\mathtt{b} \mid \mathtt{b}$$
$$T \rightarrow T\mathtt{a} \mid \varepsilon$$



Example **CFG→PDA**

$$S \to \mathtt{a}T\mathtt{b} \mid \mathtt{b}$$
$$T \to T\mathtt{a} \mid \varepsilon$$



If: stack top is **terminal**, pop and read matching input

Example **CFG→PDA**

Example Derivation using CFG:

$S \Rightarrow \mathtt{a}T\mathtt{b}$ (using rule $S \rightarrow \mathtt{a}T\mathtt{b}$)

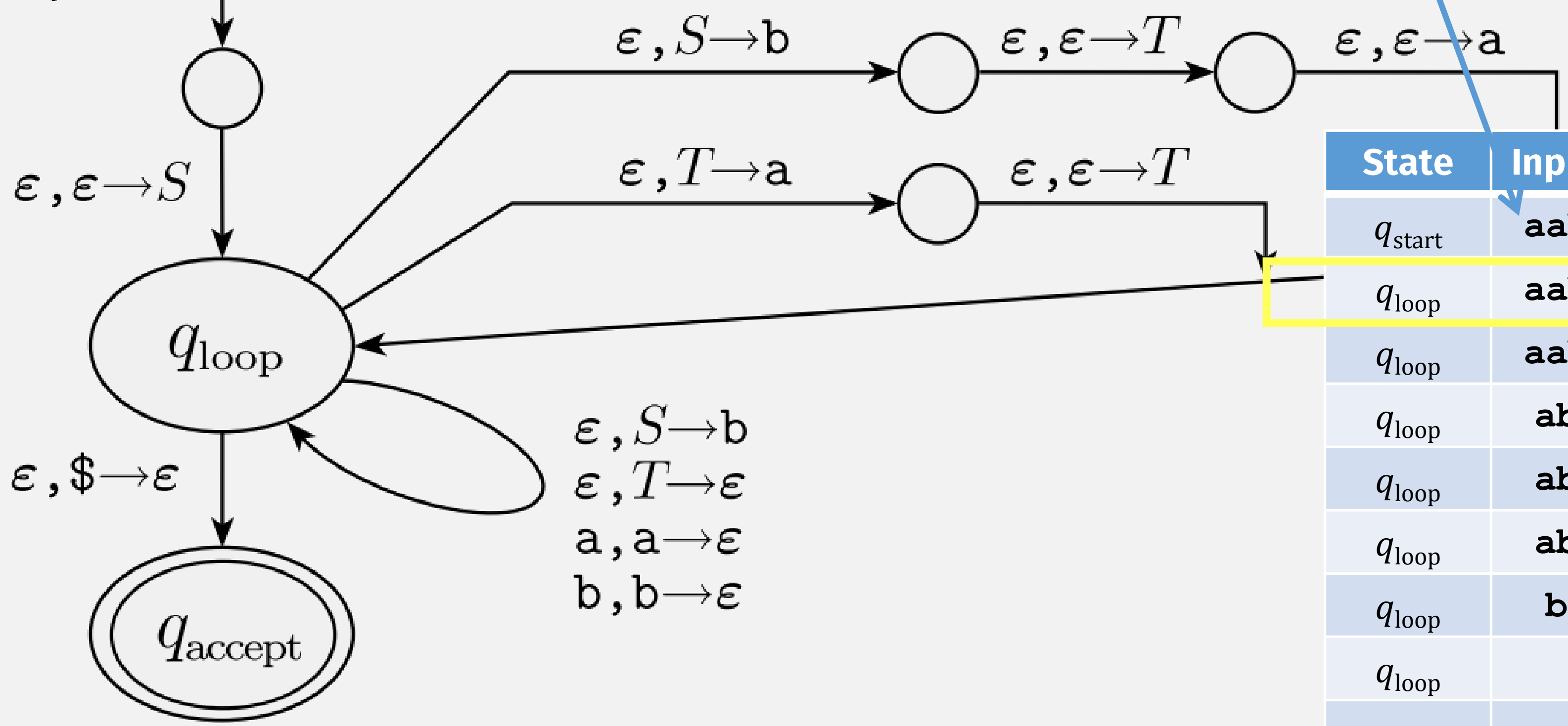
$\Rightarrow \mathtt{a}T\mathtt{ab}$ (using rule $T \rightarrow T\mathtt{a}$)

$\Rightarrow \mathtt{aab}$ (using rule $T \rightarrow \varepsilon$)

$S \rightarrow \mathtt{a}T\mathtt{b} \mid \mathtt{b}$

$T \rightarrow T\mathtt{a} \mid \varepsilon$

Machine is doing reverse of grammar:
- start with the string,
- Find rules that generate string



PDA Example

State	Input	Stack	Equiv Rule
q_{start}	aab		
q_{loop}	aab	$S\$$	
q_{loop}	aab	$\mathtt{a}T\mathtt{b}\$$	$S \rightarrow \mathtt{a}T\mathtt{b}$
q_{loop}	ab	$T\mathtt{b}\$$	
q_{loop}	ab	$T\mathtt{ab}\$$	$T \rightarrow T\mathtt{a}$
q_{loop}	ab	ab$	$T \rightarrow \varepsilon$
q_{loop}	b	b$	
q_{loop}		$	
q_{accept}			

Example **CFG→PDA**

Example Derivation using CFG:

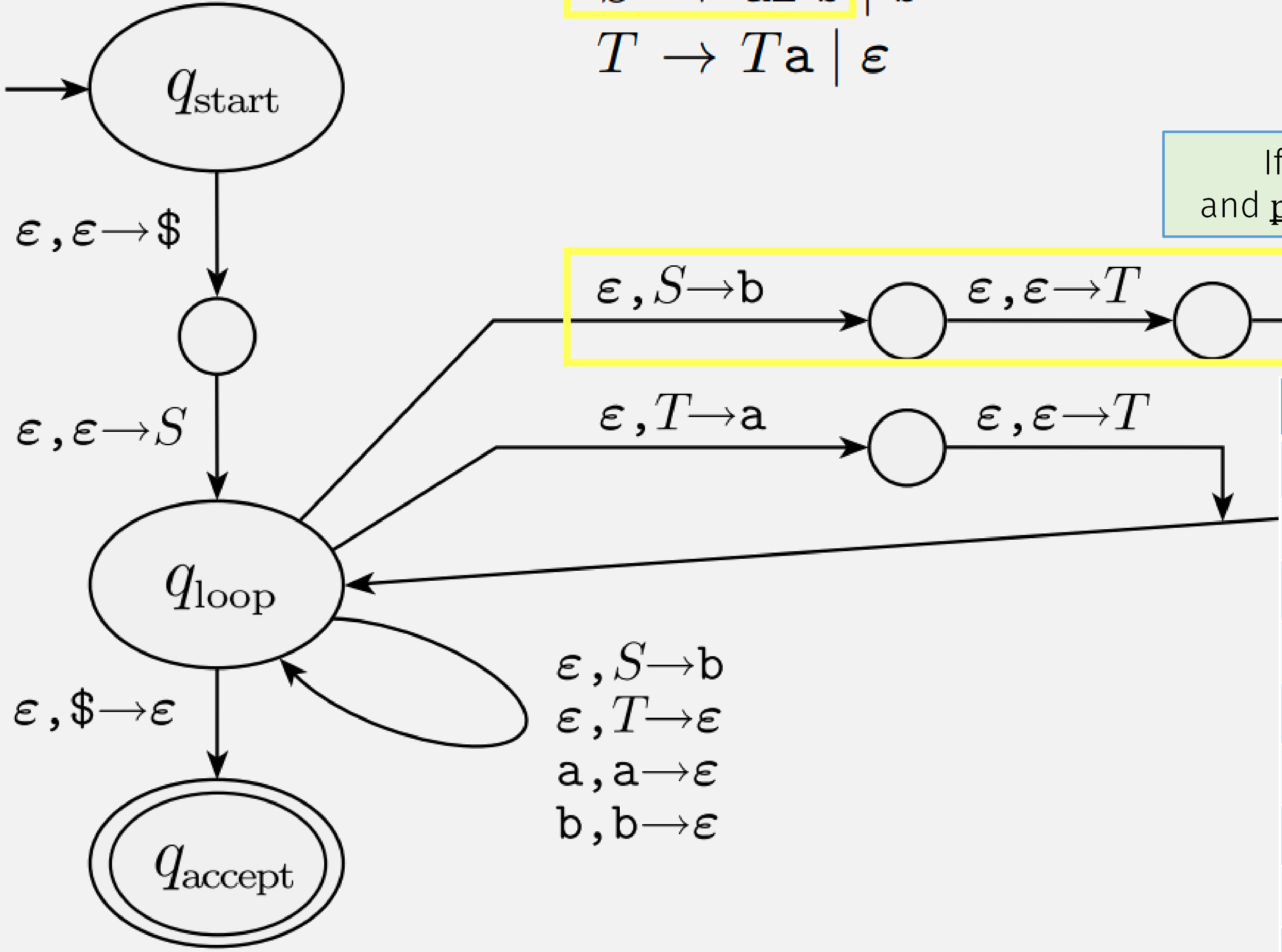
$S \Rightarrow \mathbf{a}T\mathbf{b}$ (using rule $S \to \mathbf{a}T\mathbf{b}$)

$\Rightarrow \mathbf{a}T\mathbf{ab}$ (using rule $T \to T\mathbf{a}$)

$\Rightarrow \mathbf{aab}$ (using rule $T \to \varepsilon$)

$$S \to \mathtt{a}T\mathtt{b} \mid \mathtt{b}$$
$$T \to T\mathtt{a} \mid \varepsilon$$

If: stack top is **variable** S, pop S and push **rule right-sides** (in rev order)



PDA Example

State	Input	Stack	Equiv Rule
q_{start}	**aab**		
q_{loop}	**aab**	S\$	
q_{loop}	**aab**	**a**T**b**\$	$S \to \mathbf{a}T\mathbf{b}$
q_{loop}	**ab**	T**b**\$	
q_{loop}	**ab**	T**ab**\$	$T \to T\mathbf{a}$
q_{loop}	**ab**	**ab**\$	$T \to \varepsilon$
q_{loop}	**b**	**b**\$	
q_{loop}		\$	
q_{accept}			

Example **CFG→PDA**

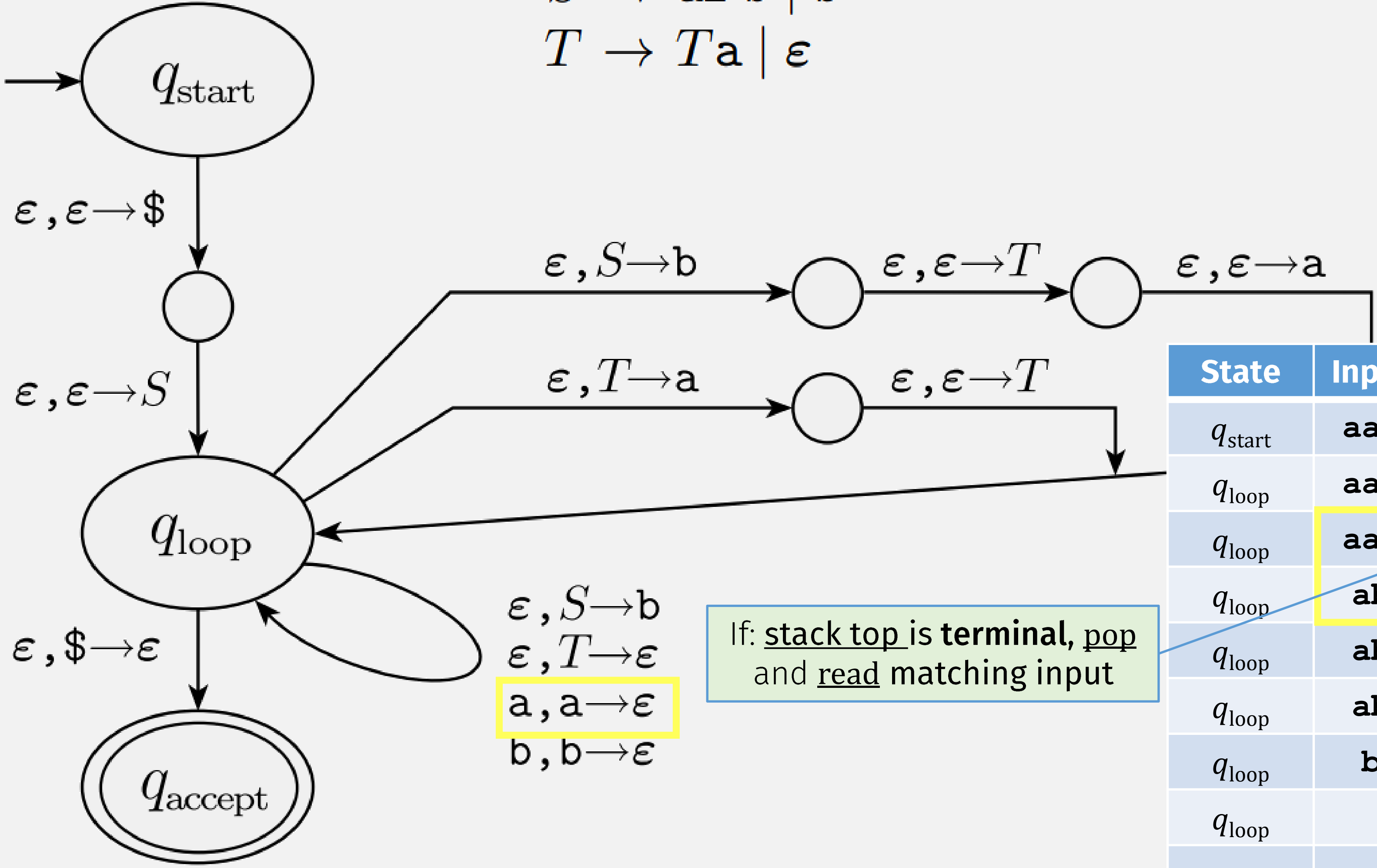
Example Derivation using CFG:

$S \Rightarrow \mathtt{a}T\mathtt{b}$ (using rule $S \rightarrow \mathtt{a}T\mathtt{b}$)

$\Rightarrow \mathtt{a}T\mathtt{ab}$ (using rule $T \rightarrow T\mathtt{a}$)

$\Rightarrow \mathtt{aab}$ (using rule $T \rightarrow \varepsilon$)

$$S \rightarrow \mathtt{a}T\mathtt{b} \mid \mathtt{b}$$
$$T \rightarrow T\mathtt{a} \mid \varepsilon$$



If: stack top is **terminal**, pop and read matching input

PDA Example

State	Input	Stack	Equiv Rule
q_{start}	aab		
q_{loop}	aab	$S\$$	
q_{loop}	aab	$\mathtt{a}T\mathtt{b}\$$	$S \rightarrow \mathtt{a}T\mathtt{b}$
q_{loop}	ab	$T\mathtt{b}\$$	
q_{loop}	ab	$T\mathtt{ab}\$$	$T \rightarrow T\mathtt{a}$
q_{loop}	ab	$\mathtt{ab}\$$	$T \rightarrow \varepsilon$
q_{loop}	b	$\mathtt{b}\$$	
q_{loop}		$\$$	
q_{accept}			

Example CFG→PDA

Example Derivation using CFG:

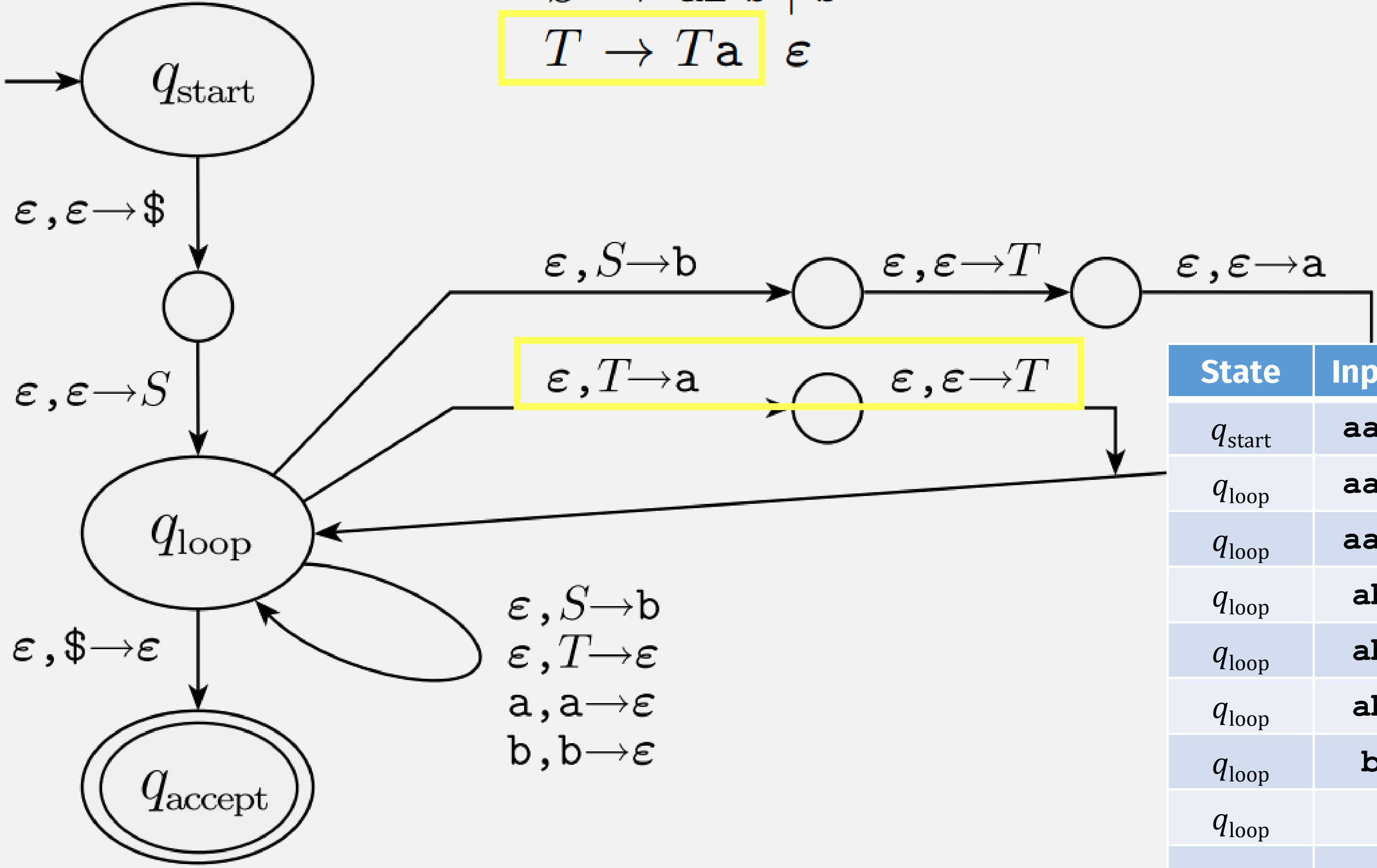
$S \Rightarrow$ **a**T**b** (using rule $S \rightarrow$ **a**T**b**)

$\Rightarrow$ **a**T**ab** (using rule $T \rightarrow T$**a**)

$\Rightarrow$ **aab** (using rule $T \rightarrow \varepsilon$)

$S \rightarrow \mathtt{a}T\mathtt{b} \mid \mathtt{b}$

$T \rightarrow T\mathtt{a} \mid \varepsilon$



PDA Example

State	Input	Stack	Equiv Rule
q_{start}	aab		
q_{loop}	aab	$S$$	
q_{loop}	aab	aTb$	$S \rightarrow$ aTb
q_{loop}	ab	Tb$	
q_{loop}	ab	Tab$	$T \rightarrow T$a
q_{loop}	ab	ab$	$T \rightarrow \varepsilon$
q_{loop}	b	b$	
q_{loop}		$	
q_{accept}			

A lang is a CFL **iff** some PDA recognizes it

☑ ⇒ If **a language is a CFL**, then **a PDA recognizes it**

- Convert CFG→PDA

⇐ If **a PDA recognizes a language**, then **it's a CFL**

- <u>To prove this part:</u> show PDA has an equivalent CFG

PDA→CFG: Prelims

Before converting PDA to CFG, modify it so :

1. It has a single accept state, q_{accept}.
2. It empties its stack before accepting.
3. Each transition either pushes a symbol onto the stack (a *push* move) or pops one off the stack (a *pop* move), but it does not do both at the same time.

Important:
This doesn't change the language recognized by the PDA

PDA P -> CFG G: Variables

$P = (Q, \Sigma, \Gamma, \delta, q_0, \{q_{\text{accept}}\})$ variables of G are $\{A_{pq} | p, q \in Q\}$

- Want: if P goes from state p to q reading input x, then some A_{pq} generates x

- So: For every pair of states p, q in P, add variable A_{pq} to G

- Then: connect the variables together by,
  - Add rules: $A_{pq} \rightarrow A_{pr}A_{rq}$, for each state r
  - These rules allow grammar to simulate every possible transition
  - (We haven't added input read/generated terminals yet)

The Key IDEA

- To add terminals: pair up stack pushes and pops (essence of a CFL)

PDA P -> CFG G: Generating Strings

$P = (Q, \Sigma, \Gamma, \delta, q_0, \{q_{\text{accept}}\})$

variables of G are $\{A_{pq} \mid p, q \in Q\}$

- The key: **pair up stack pushes and pops** (essence of a CFL)

if $\delta(p, a, \varepsilon)$ contains (r, u) and $\delta(s, b, u)$ contains (q, ε),

put the rule $A_{pq} \to aA_{rs}b$ in G

PDA P -> CFG G : Generating Strings

$P = (Q, \Sigma, \Gamma, \delta, q_0, \{q_{\text{accept}}\})$

variables of G are $\{A_{pq} | \, p, q \in Q\}$

- <u>The key</u>: **pair up stack pushes and pops** (essence of a CFL)

if $\delta(p, a, \varepsilon)$ contains (r, u) and $\delta(s, b, u)$ contains (q, ε),

put the rule $A_{pq} \to aA_{rs}b$ in G

PDA P -> CFG G: Generating Strings

$P = (Q, \Sigma, \Gamma, \delta, q_0, \{q_{\text{accept}}\})$

variables of G are $\{A_{pq} \mid p, q \in Q\}$

- The key: **pair up stack pushes and pops** (essence of a CFL)

if $\delta(p, a, \varepsilon)$ contains (r, u) and $\delta(s, b, u)$ contains (q, ε),

put the rule $A_{pq} \rightarrow aA_{rs}b$ in G

A language is a CFL ⇔ A PDA recognizes it

☑⇒ If **a language is a CFL,** then **a PDA recognizes it**
- Convert **CFG→PDA**

☑⇐ If **a PDA recognizes a language,** then **it's a CFL**
- Convert **PDA→CFG**

∎

Submit in-class work 3/20

On Gradescope