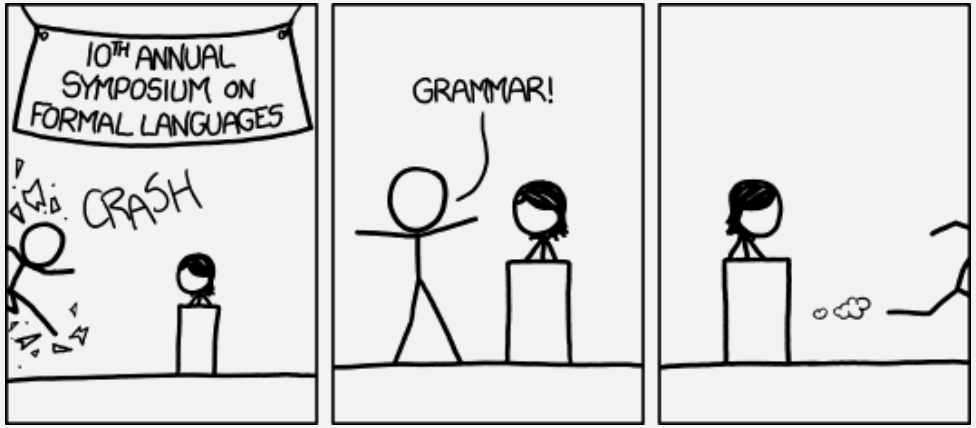
CS 420

Pushdown Automata (PDAs)

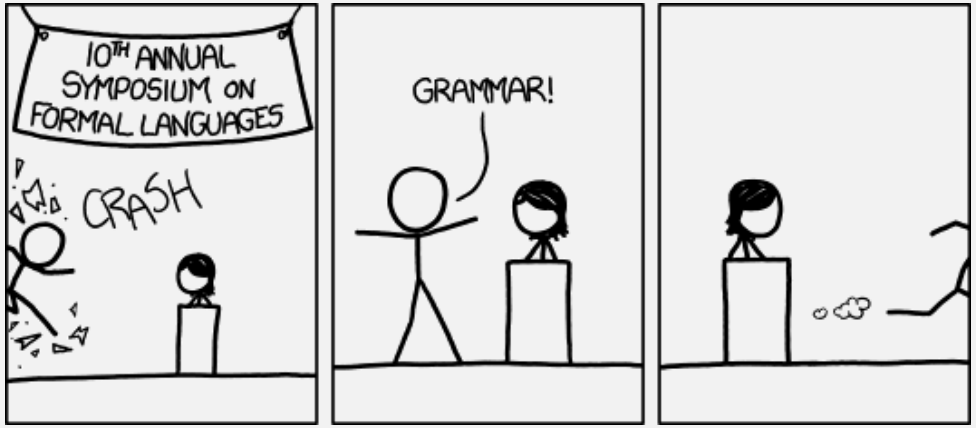
Tuesday October 27, 2026

UMass Boston Computer Science



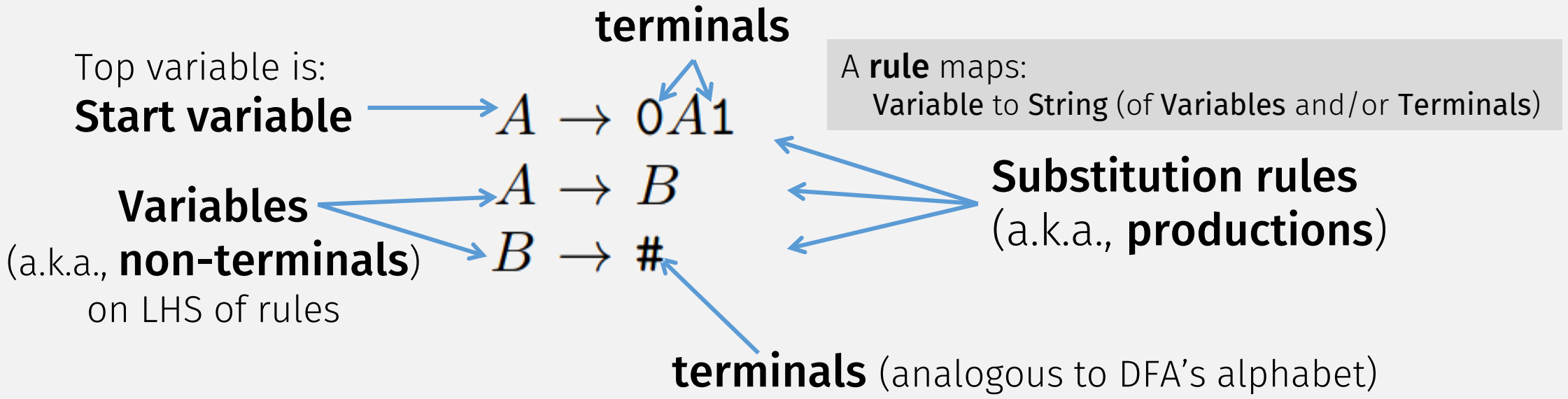
Announcements

- HW 6
  - Due: Thur 10/29 12:30pm



Last Time:

A Context-Free Grammar (CFG)



Last Time:

Context-Free Grammar (CFG): Formally

let G_1 = **mk-CFG** (V, Σ, R, S) …

R is this set of rules: pair of Var, String (of vars+terminals)

Top variable is:
Start variable → $A \to 0A1$

Variables (a.k.a., **non-terminals**) → $A \to B$, $B \to \#$

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

terminals (analogous to DFA's alphabet)

A ***context-free grammar*** is a mk-CFG (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.

where:

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

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

$S = A$,

Last Time:

Definition:
A **CFG** describes a
context-free language!

DefCFL

(some defined CFG)

IF G IS-A **Context-free** Grammar
AND
G **describes** L (some defined language)
THEN L IS-A **Context-free** Language

(abbrev)

DefCFL

IF G IS-A **CFG**
AND
G **describes** L
THEN L IS-A **CFL**

Generating Strings with a CFG

Called "context free" because: we replace vars with RHS of rule, without looking at surrounding chars, i.e., context

Definition:
A **CFG** describes a **context-free language!**

but what strings are in the language?

1st rule → $A \to 0A1$

2nd rule → $A \to B$

Last rule → $B \to \#$

"Applying a rule"
= replace LHS variable with RHS sequence

At each step, *arbitrarily* choose any variable to replace, and any rule to apply

Stop when: string is all terminals

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

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

Start with: Start variable

Apply 1st rule

1st rule again

1st rule again

Apply 2nd rule

Apply last rule

Generating Strings with a CFG

let $G_1 =$

$$A \to 0A1$$
$$A \to B$$
$$B \to \#$$

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

$$\textsc{LangOf}(G_1) = \{0^n\#1^n \mid n \geq 0\}$$

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**

Derivations: Formally

A ***context-free grammar*** is a `mk-CFG` (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 = **`mk-CFG`** (V, Σ, R, S) ...

Single-step

"context" stays same

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

Where:

sequence of: variables or terminals

$\gamma, \alpha, \beta \in (V \cup \Sigma)^*$

$A \in V$ ← Variable

$A \to \gamma \in R$ ← Rule

Derivations: Formally

A ***context-free grammar*** is a `mk-CFG` (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 = **`mk-CFG`** (V, Σ, R, S) ...

Single-step

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

Where:

$$\alpha, \beta \in (V \cup \Sigma)^*$$

$$A \in V \leftarrow \text{Variable}$$

$$A \to \gamma \in R \leftarrow \text{Rule}$$

Multi-step (recursively defined, on # steps)

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

Recursive case: (1 or more steps)

Single step

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

(smaller) Recursive "call"

- Then: $\alpha \overset{*}{\underset{G}{\Rightarrow}} \gamma$

Lang of a CFG, Formally

A ***context-free grammar*** is a `mk-CFG` (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 = **`mk-CFG`** (V, Σ, R, S) ...

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

"all possible sequences of terminal symbols ..."

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

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

Designing Grammars : Basics

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

Last Time:

Non-Regular Languages

Example:

An arbitrary count

$$L = \{ \mathtt{0}^n\mathtt{1}^n \mid n \geq 0 \}$$

- A DFA recognizing L would **require infinite states!** (impossible)
  - States representing: **zero `0`s seen, one `0` seen, two `0`s, …**
- **This language is the same as many PLs,** e.g., **HTML!**
  - To better see this replace:
    - "`0`" with "`<tag>`" or "`(`"
    - "`1`" with "`</tag>`" or "`)`"
- The Problem: **remembering <u>nestedness</u>**
  - **Need to count arbitrary nesting depths**
    - E.g., `{ { { … } } }`
  - Thus: **most programming language syntax** is **<u>not regular</u>!**

A regular lang can't "link" together open/close

Designing Grammars : Basics

1. **Think about what you want to "link" together**

- E.g., 0^n1^n
  - $A \rightarrow 0A1$
  - # 0s and # 1s are **"linked"**

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

2. **Start** with **small grammars** (computation) and **then combine**
   - just like with **DFA**s, **NFA**s, and **programming**!

In-class exercise: Creating CFG

alphabet Σ is $\{0,1\}$

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

1) come up with examples: In the language: **010, 101, 110001** **1, 0 ?** ☑

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

2) Create CFG:

Needed Rules:

$S \rightarrow 0M0 \mid 1M1 \mid 0 \mid 1$ "start/end symbol are "linked" (ie, same); middle can be anything"

$M \rightarrow MT \mid \varepsilon$ "middle: all possible terminals, repeated (ie, all possible strings)"

$T \rightarrow 0 \mid 1$ "all possible terminals"

3) Check CFG: generates examples in the language; doesn't generate examples not in language

Examples Table! (justifies that the "CFG describes L")

Designing Grammars: Building Up

- **Start** with **small grammars** and then **combine** (just like **programming**)
  - To create a **grammar for the language** $\{0^n 1^n | n \geq 0\} \cup \{1^n 0^n | n \geq 0\}$
  - First create **grammar for lang** $\{0^n 1^n | n \geq 0\}$:

$$S_1 \rightarrow 0S_1 1 \mid \varepsilon$$

  - Then create **grammar for lang** $\{1^n 0^n | n \geq 0\}$:

$$S_2 \rightarrow 1S_2 0 \mid \varepsilon$$

  - Then **combine:**

$$S \rightarrow S_1 \mid S_2$$

$$S_1 \rightarrow 0S_1 1 \mid \varepsilon$$

$$S_2 \rightarrow 1S_2 0 \mid \varepsilon$$

New start variable and **rule combines two smaller grammars**

"|" = "or" = union (combines 2 rules with same left side)

Last Time:

(Closed) Operations for CFLs?

- **Start** with **small grammars** and then **combine** (just like **programming**)
- "Or": $S \to S_1 \mid S_2$
- "Concatenate": $S \to S_1 S_2$
- "Star" (repetition): $S' \to S' S_1 \mid \varepsilon$

Status check:
Could you write out the precise **Statement to Prove** and the **full proof?**

"The set of CFLs are closed under …"

"IF L_1 IS-A CFL AND L_2 IS-A CFL THEN … IS-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**

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

thm

def

def

thm

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)	**Push-down Automata** (PDA)
recognizes a **Regular Lang**	recognizes a **CFL**
Proved:	Must Prove:
Regular Lang ⇔ Regular Expr ☑	CFL ⇔ PDA ???

thm

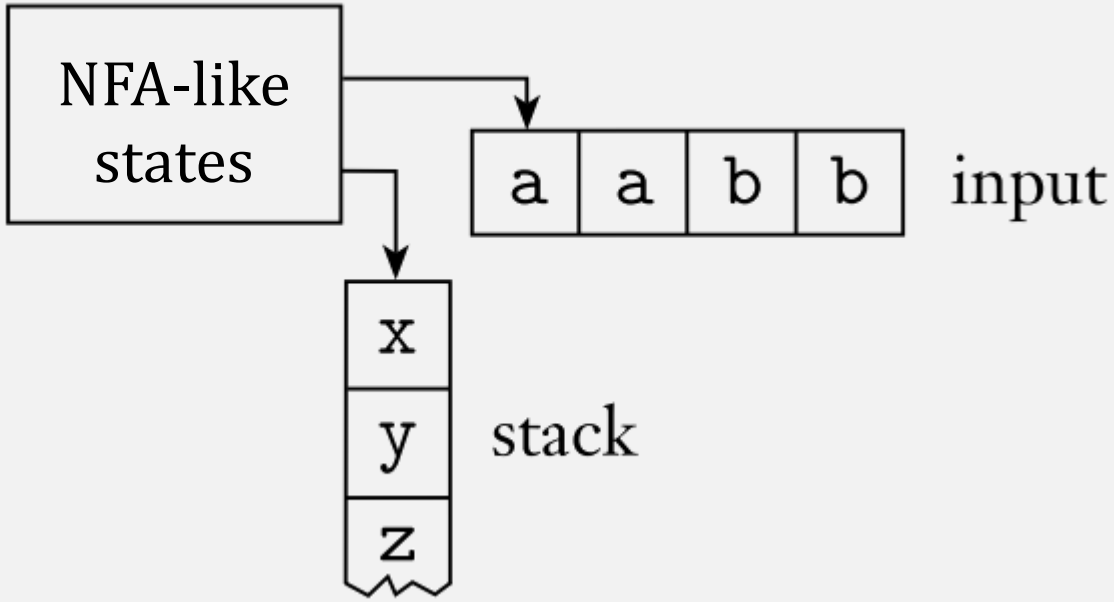
def

def

thm

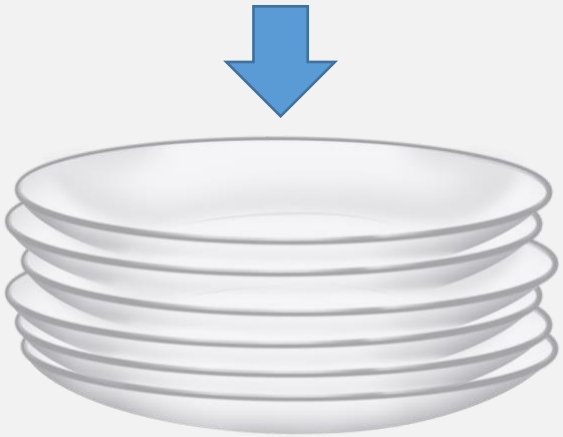
Pushdown Automata (PDA)

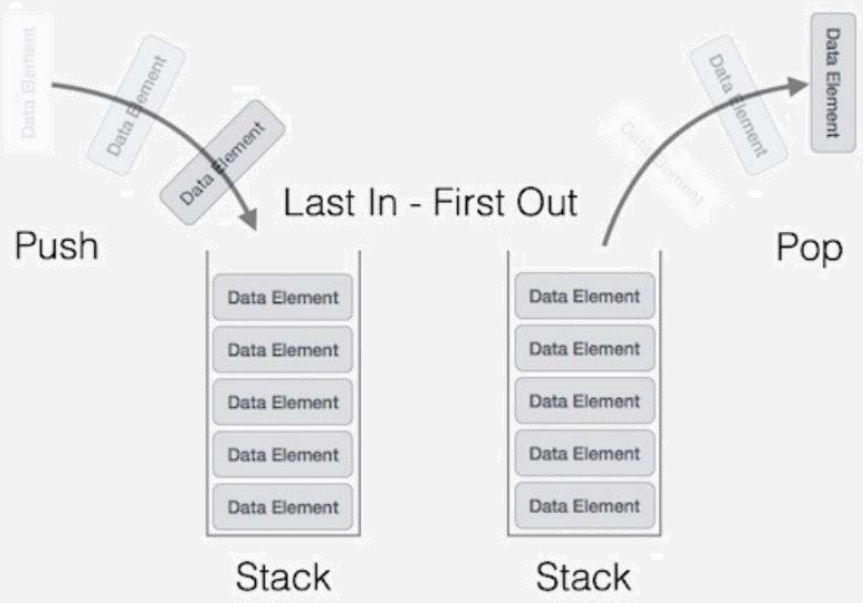
PDA = NFA + a stack



What is a Stack?

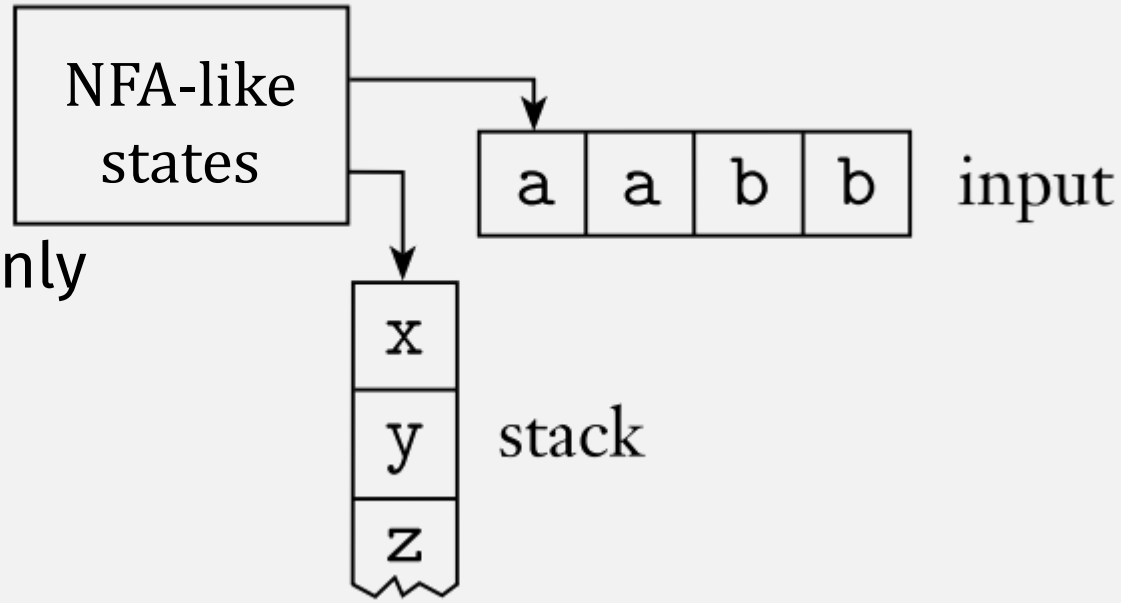
- A **restricted** kind of (**infinite!**) **memory**
- **Access** to **top element only**
- 2 Operations only: **push, pop**





Pushdown Automata (PDA)

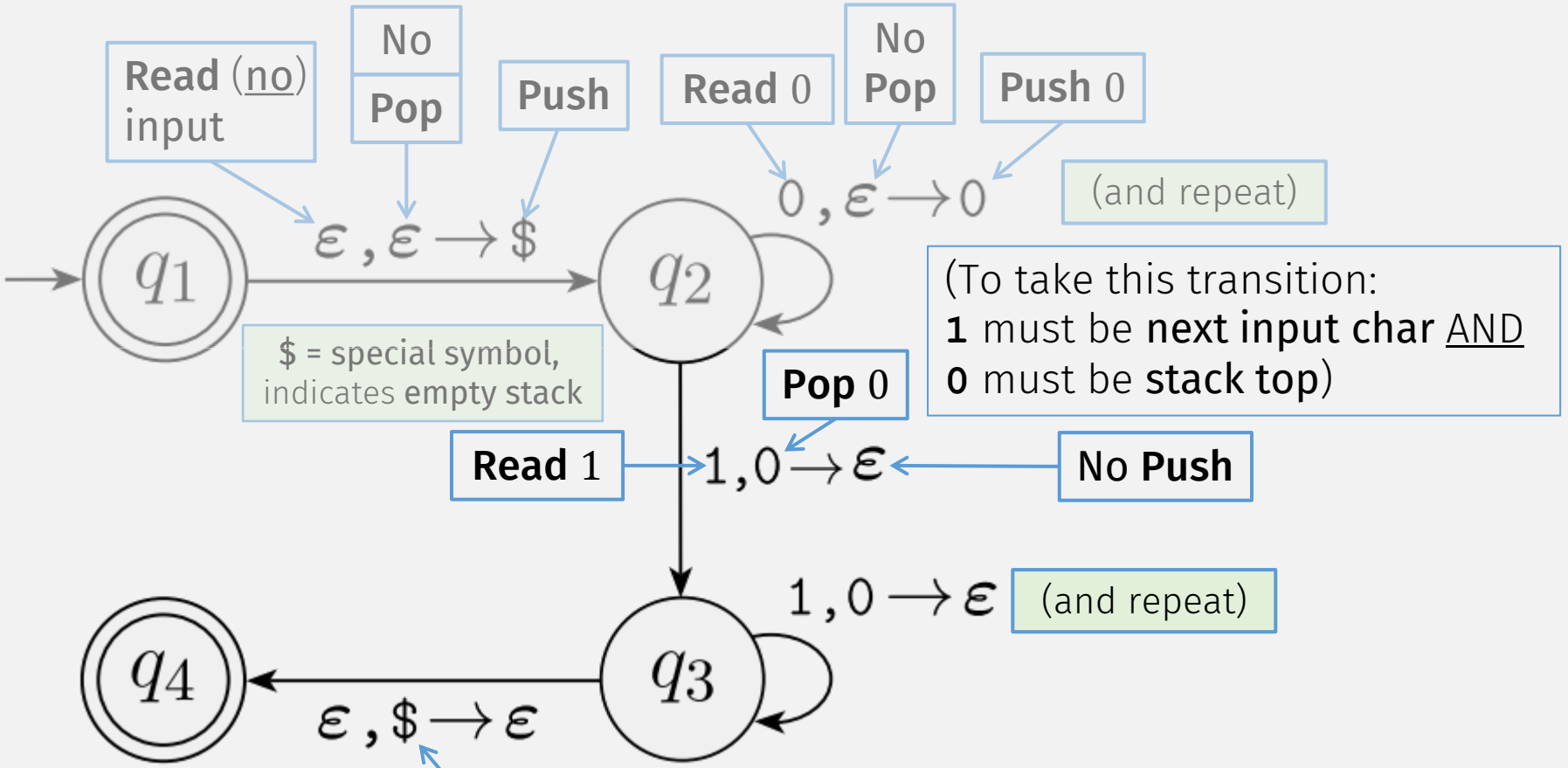
- **PDA = NFA + a stack**
  - Infinite memory!
  - But … read/write top location only
    - Push/pop



$$\{0^n 1^n \mid n \geq 0\}$$

An Example PDA

A **PDA transition** has <u>3 parts:</u>
- **Read** (input)
- **Pop** (stack)
- **Push** (stack)



Formal Definition of PDA

A ***pushdown automaton*** is a `mk-PDA` $(Q, \Sigma, \Gamma, \delta, q_0, F)$, 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. $\delta: Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$ is the transition function,
5. $q_0 \in Q$ is the start state, and
6. $F \subseteq Q$ is the set of accept states.

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

Input

Pop

Push

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

let M_1 =
mk-PDA (Q, Σ, Γ, δ, q_1, F)

where …

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

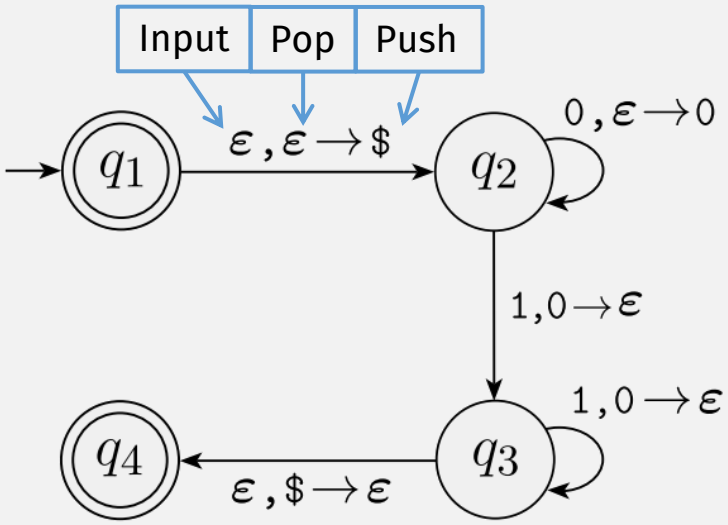
$$\Sigma = \{0, 1\}$$

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

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

PDA Formal Definition Example

Stack alphabet has special stack symbol $



A ***pushdown automaton*** is a `mk-PDA` $(Q, \Sigma, \Gamma, \delta, q_0, F)$, 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. $\delta: Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$ 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

let M_1 =
mk-PDA (Q, Σ, Γ, δ, q_1, F)
where ...

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

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

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

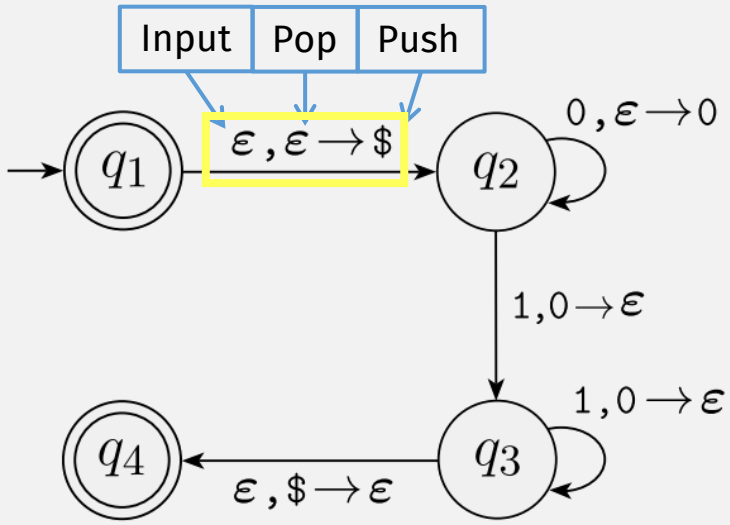
$$F = \{q_1, q_4\}$$

$$\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon) \qquad \text{blank entries signify } \emptyset$$

Let's play a game:
"Match transition to a number!"

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			4	
q_3					$\{(q_3, \varepsilon)\}$ 3			$\{(q_4, \varepsilon)\}$	
q_4									

Input
Pop
Push



A ***pushdown automaton*** is a **mk-PDA** $(Q, \Sigma, \Gamma, \delta, q_0, F)$, 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. $\delta: Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$ 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

let M_1 =
mk-PDA (Q, Σ, Γ, δ, q_1, F)
where ...

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

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

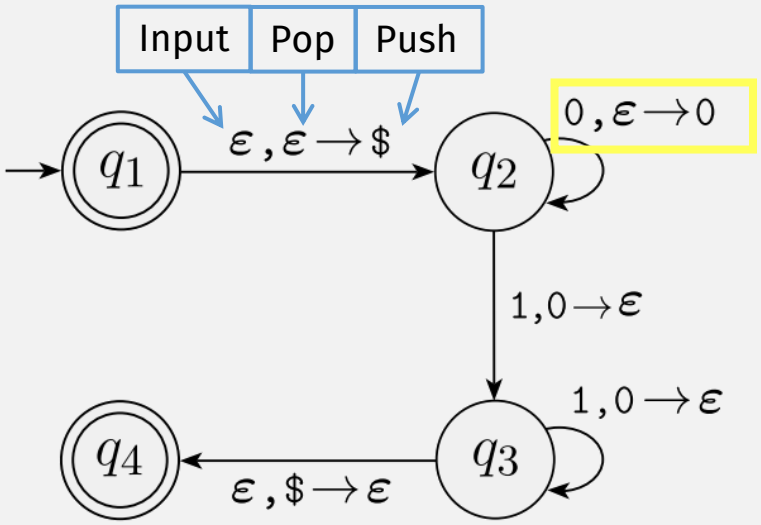
$$\Gamma = \{0, \$\}$$

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

$$\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon) \qquad \text{blank entries signify } \emptyset$$

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			$\{(q_4, \varepsilon)\}$ 4		
q_3				$\{(q_3, \varepsilon)\}$ 3					
q_4									

Input
Pop
Push



A ***pushdown automaton*** is a `mk-PDA` $(Q, \Sigma, \Gamma, \delta, q_0, F)$, 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. $\delta: Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$ 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

let M_1 =
mk-PDA (Q, Σ, Γ, δ, q_1, F)
where ...

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

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

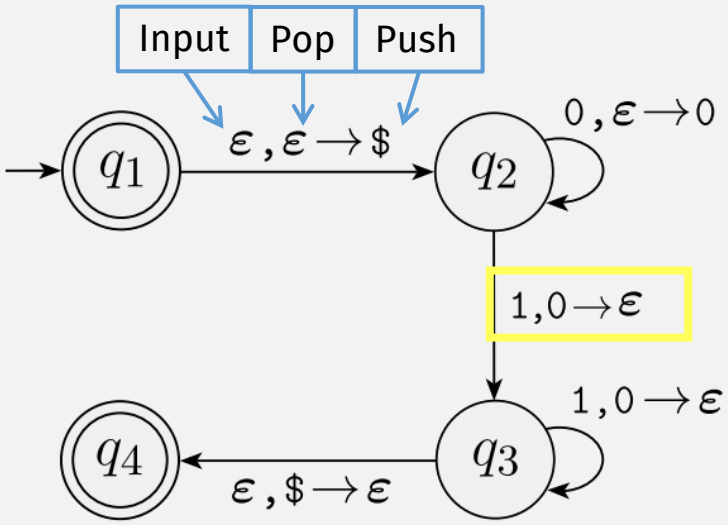
$$\Gamma = \{0, \$\}$$

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

$$\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon) \qquad \text{blank entries signify } \emptyset$$

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			4		
q_3				$\{(q_3, \varepsilon)\}$ 3			$\{(q_4, \varepsilon)\}$		
q_4									

Input
Pop
Push



A ***pushdown automaton*** is a mk-PDA $(Q, \Sigma, \Gamma, \delta, q_0, F)$, 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. $\delta: Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$ 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

let M_1 =
mk-PDA (Q, Σ, Γ, δ, q_1, F)
where ...

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

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

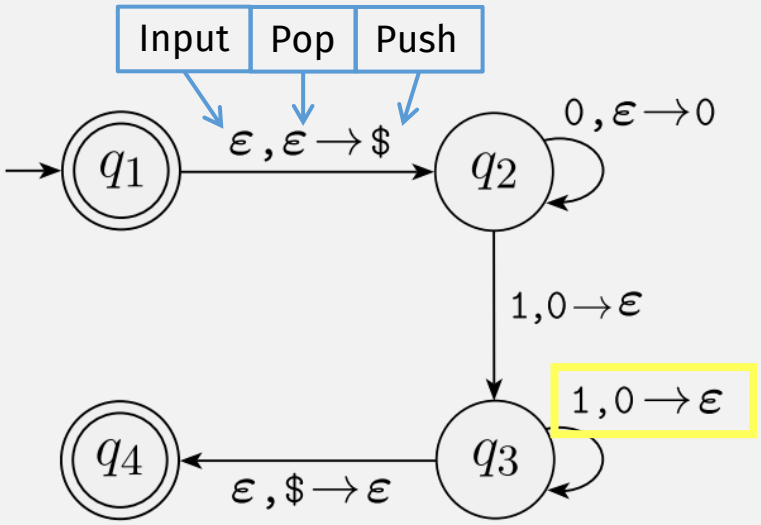
$$\Gamma = \{0, \$\}$$

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

$$\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon) \qquad \text{blank entries signify } \emptyset$$

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			4		
q_3				$\{(q_3, \varepsilon)\}$ 3			$\{(q_4, \varepsilon)\}$		
q_4									

Input
Pop
Push



A ***pushdown automaton*** is a mk-PDA $(Q, \Sigma, \Gamma, \delta, q_0, F)$, 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. $\delta: Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$ 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

let M_1 =
mk-PDA (Q, Σ, Γ, δ, q_1, F)
where ...

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

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

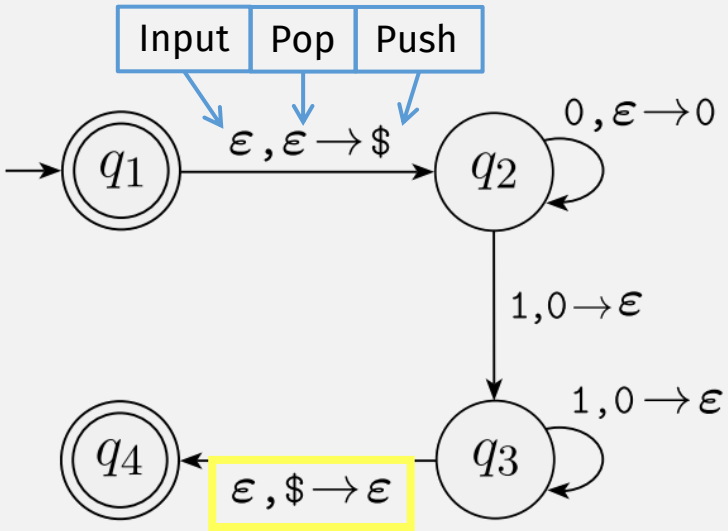
$$\Gamma = \{0, \$\}$$

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

$$\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon) \qquad \text{blank entries signify } \emptyset$$

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		4 $\{(q_4, \varepsilon)\}$		
q_3					$\{(q_3, \varepsilon)\}$ 3				
q_4									

Input
Pop
Push



A ***pushdown automaton*** is a mk-PDA $(Q, \Sigma, \Gamma, \delta, q_0, F)$, 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. $\delta: Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$ 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

let M_3 =
mk-PDA (Q, Σ, Γ, δ, q_1, F)
where ...

$Q =$

$\Sigma =$

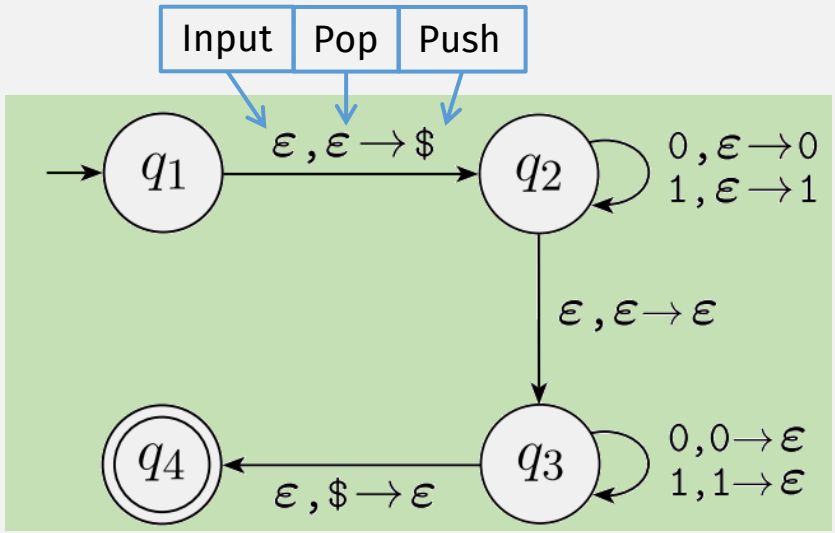
$\Gamma =$

$F =$

In-class exercise:
Fill in the blanks

$\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon)$ blank entries signify $\emptyset$

Input:	0	1	ε	← Input
Stack:	???	???	???	← Pop
?	PDA M_3 recognizing the language $\{ww^{\mathcal{R}} \mid w \in \{0,1\}^*\}$			← State/ Push
?				



A ***pushdown automaton*** is a `mk-PDA` $(Q, \Sigma, \Gamma, \delta, q_0, F)$, 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. $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$ 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

let M_3 =
mk-PDA (Q, Σ, Γ, δ, q_1, F)
where ...

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

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

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

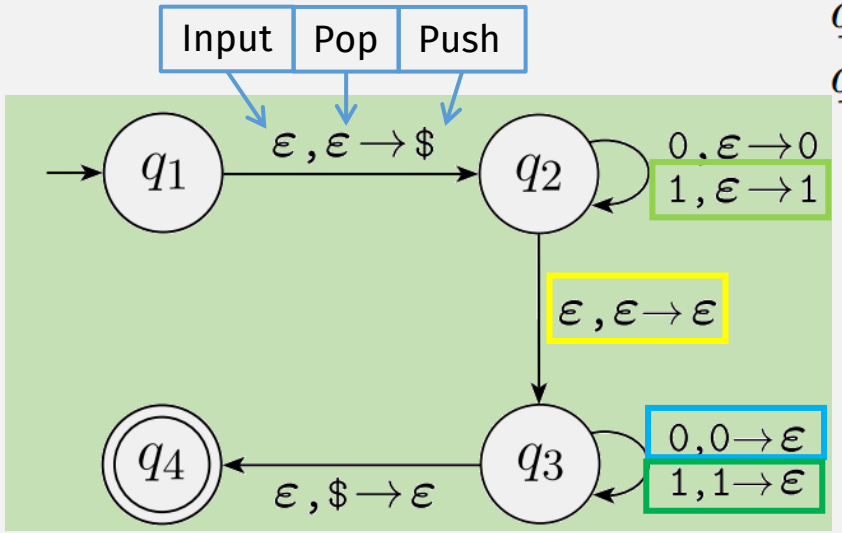
$$F = \{q_4\}$$

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

blank entries signify $\emptyset$

In-class exercise: Fill in the blanks

Input:	0			1				ε			← Input
Stack:	0	\$	ε	0	1	\$	ε	0	\$	ε	← Pop
q_1										$\{(q_2, \$)\}$	
q_2		$\{(q_2, 0)\}$				$\{(q_2, 1)\}$				$\{(q_3, \varepsilon)\}$	← State/ Push
q_3	$\{(q_3, \varepsilon)\}$			$\{(q_3, \varepsilon)\}$					$\{(q_4, \varepsilon)\}$		
q_4											



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

Flashback

DFA Computation Rules

Informally

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

A **DFA computation** (~ "Program run"):
- **Start** in ***start state***

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

Result of **computation:**
- **Accept** if **last state** is ***Accept state***
- **Reject** otherwise

Formally (i.e., mathematically)

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

A **DFA computation** is a sequence of states:

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

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

Flashback

DFA Computation Rules

Informally

Given

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

A **DFA** computation (~ "Program run"):

- Start in *start state*

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

Result of **computation:**

- Accept if **last state** is *Accept state*
- Reject otherwise

Formally (i.e., mathematically)

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

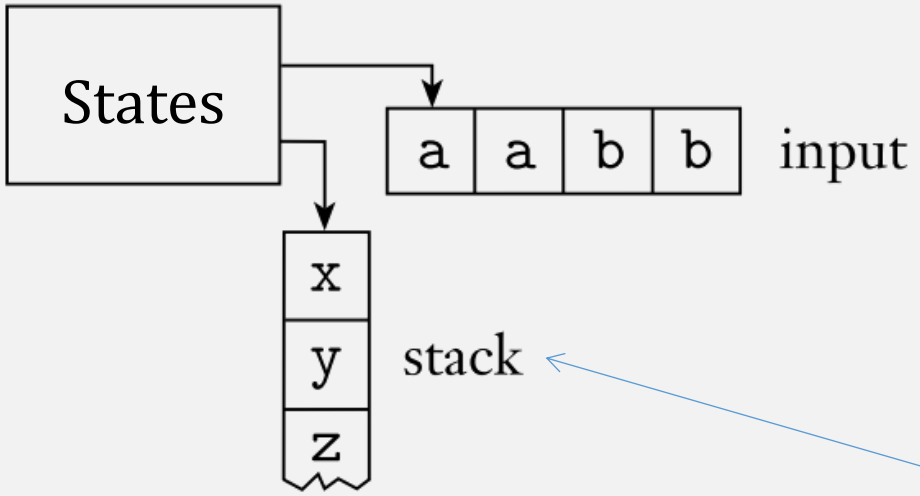
A **DFA computation** is a sequence of states:

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

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

PDA Computation?

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



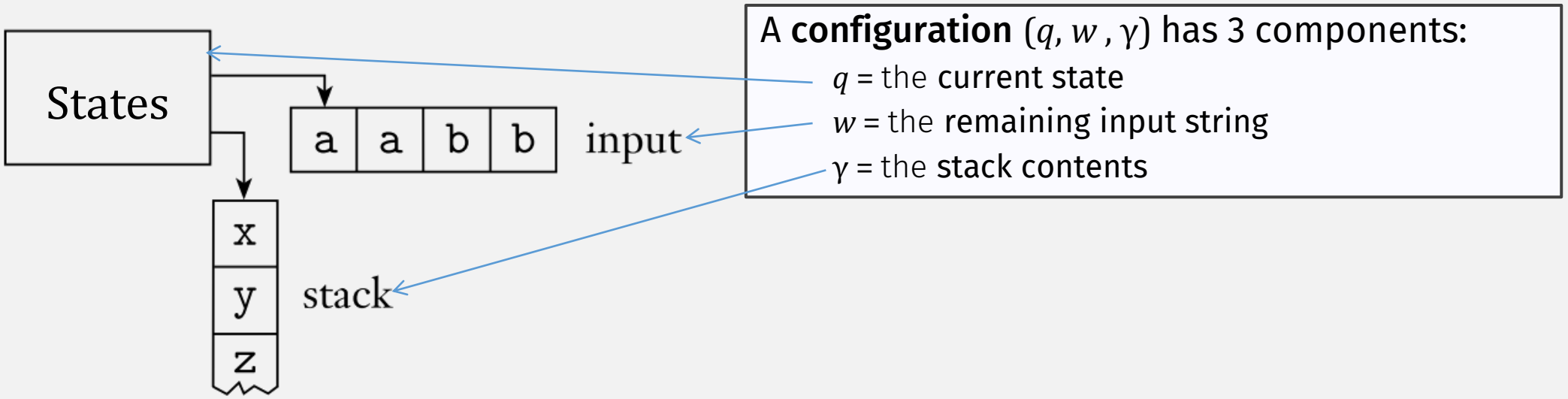
A DFA **computation** is a sequence of states …

A PDA **computation** is not just a sequence of states …

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

PDA Configurations (IDs)

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



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

PDA Computation, Formally

(one path in computation tree)

let P = **mk-PDA** (Q, Σ, Γ, δ, q_0, F)
where …

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**

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

- Recursive Case — **1 or more steps**

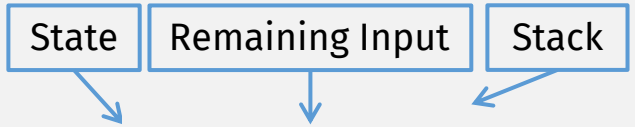
$$I \vdash^* J \text{ if there exists some ID } K$$

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

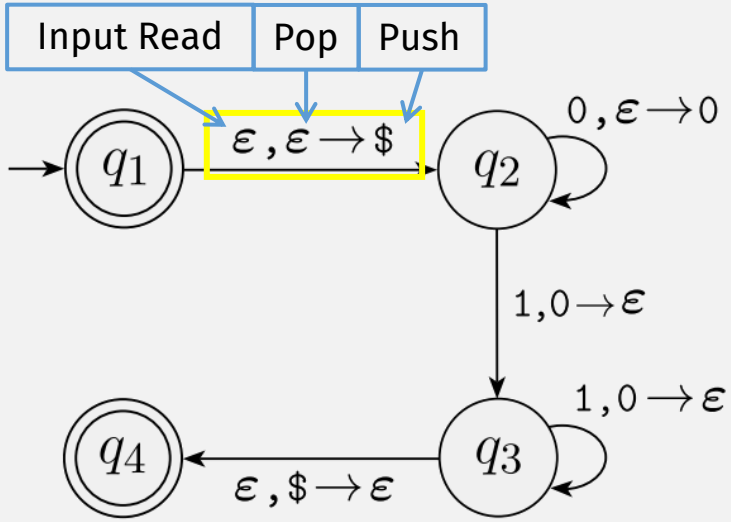
Single step | Recursive "call"

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

PDA Running Input String Example



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



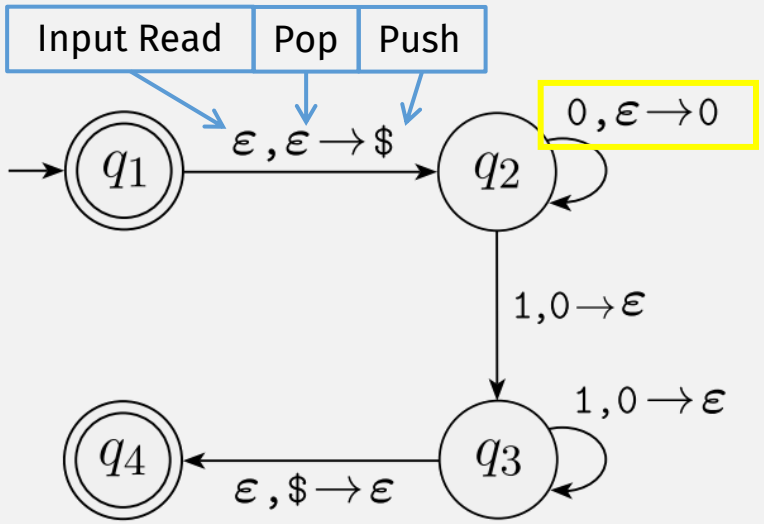
PDA Running Input String Example

State | Remaining Input | Stack

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

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

Read 0, push 0

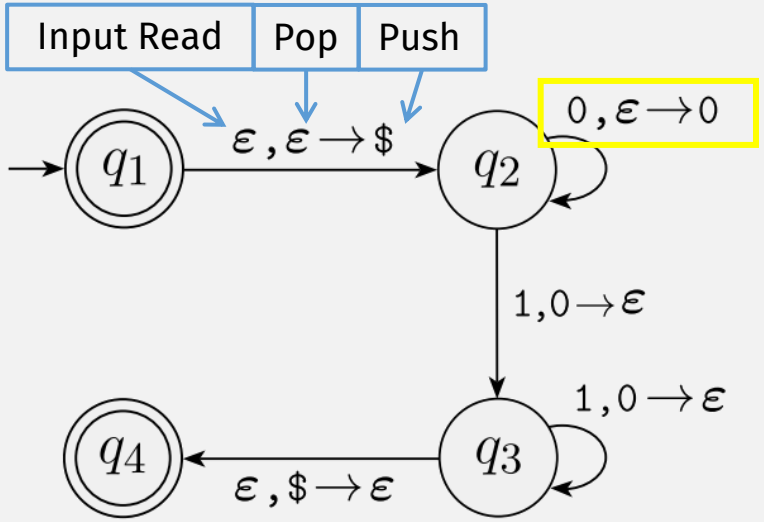


PDA Running Input String Example

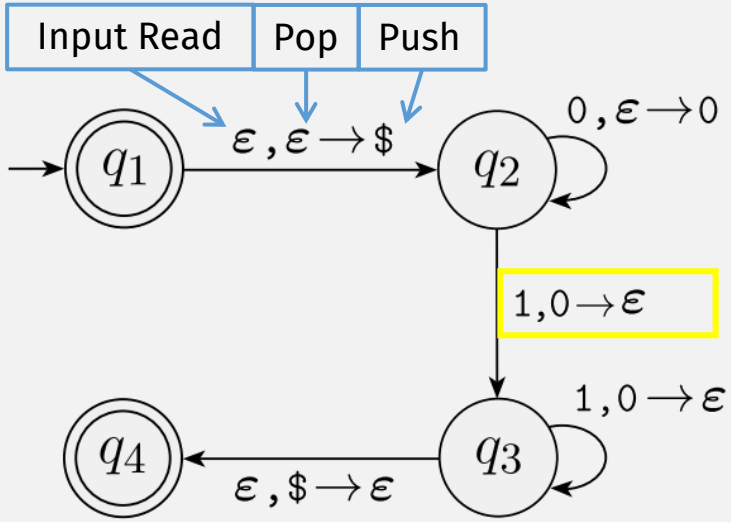
State | Remaining Input | Stack

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

Read 0, push 0



PDA Running Input String Example

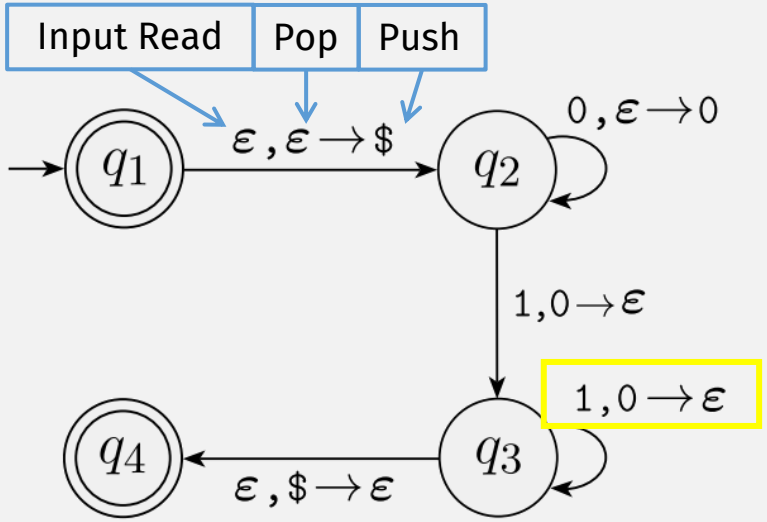


State | Remaining Input | Stack

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

Read 1, pop 0

PDA Running Input String Example

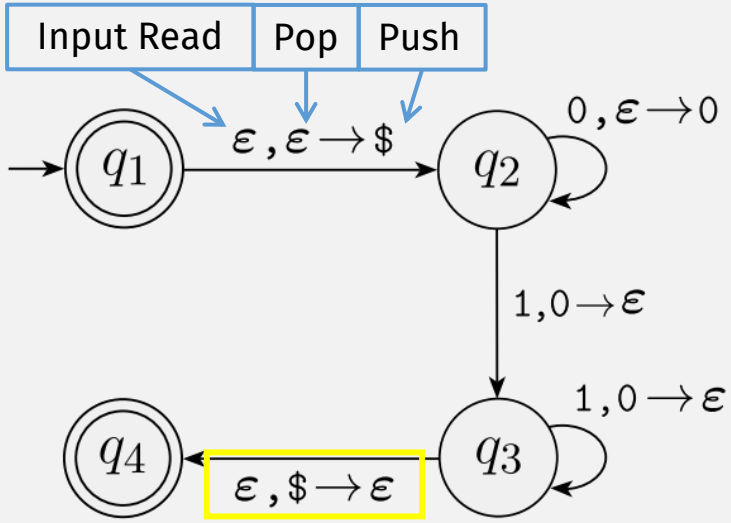


State | Remaining Input | Stack

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

Read 1, pop 0

PDA Running Input String Example



State | Remaining Input | Stack

$$(q_1, 0011, \varepsilon) \vdash (q_2, 0011, \$)$$
$$\vdash (q_2, 011, 0\$)$$
$$\vdash (q_2, 11, 00\$)$$
$$\vdash (q_3, 1, 0\$)$$
$$\vdash (q_3, \varepsilon, \$)$$
$$\vdash (q_4, \varepsilon, \varepsilon)$$

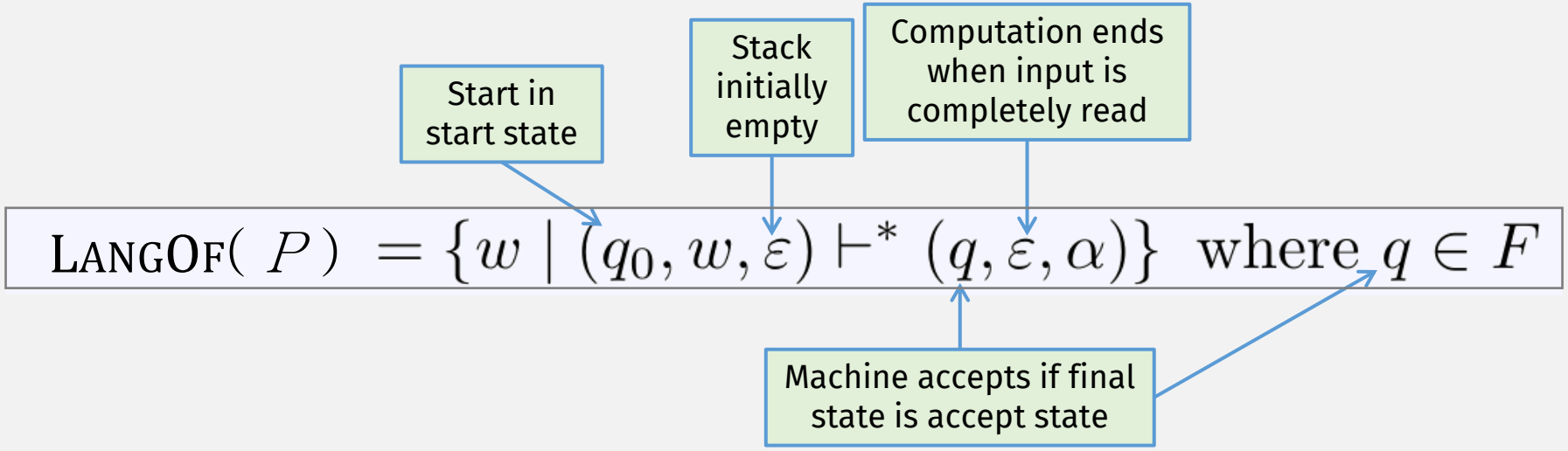
pop empty stack symbol

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 = $LangOf(M)$ = { w | M **accepts** w }

Language of a PDA

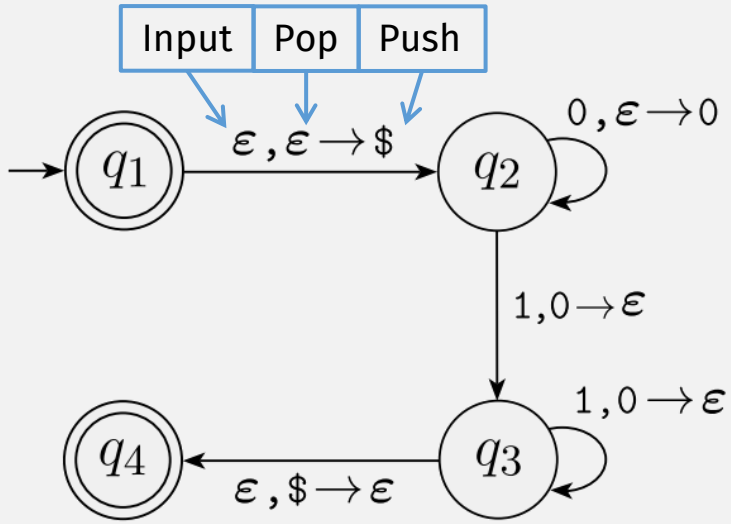
let P = **mk-PDA** (Q, Σ, Γ, δ, q_0, F)
where …



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