CS 420

CFGs vs PDAs
subCFLs and DPDAs

Thursday October 29, 2026

UMass Boston Computer Science



Announcements

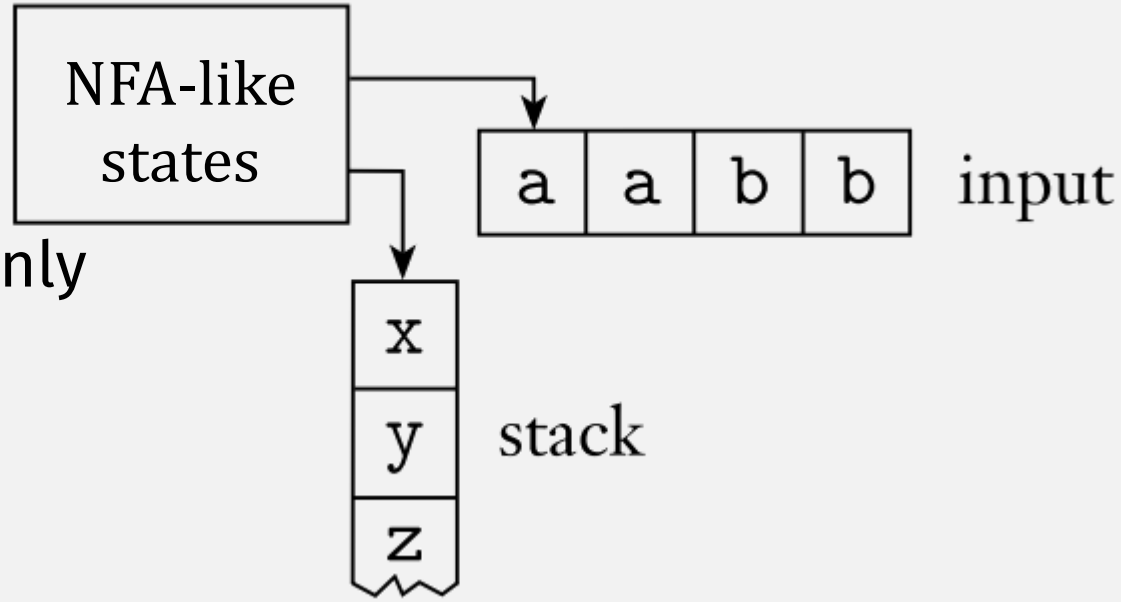
- HW 5 grades
  - Out: Fri 10/29
- HW 6
  - ~~Due: Thu 10/28 12:30pm~~
- HW 7
  - Out: Thu 10/28
  - Due: Thu 11/5 12:30pm



Last Time

Pushdown Automata (PDA)

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



Last Time

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 …

PDA Formal Definition Example

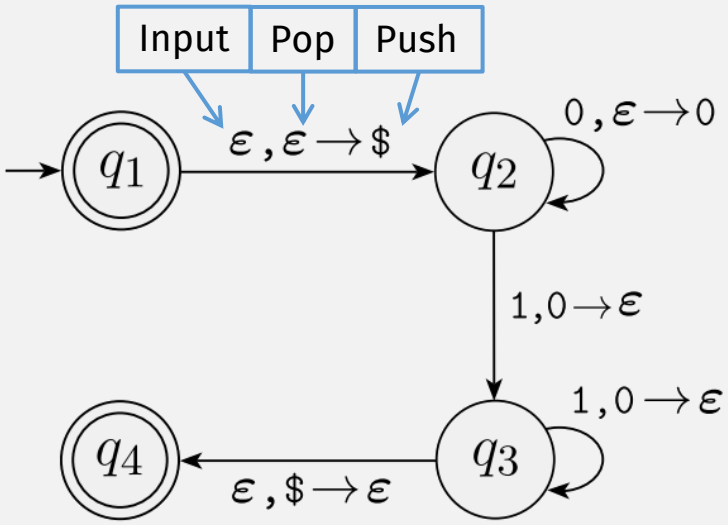
$$Q = \{q_1, q_2, q_3, q_4\}$$

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

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

Stack alphabet has special stack symbol $

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



A ***pushdown automaton*** is a `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 ...

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

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

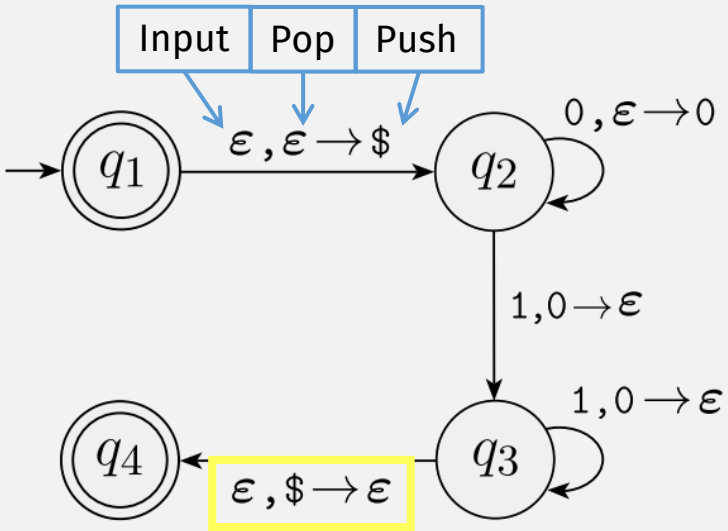
$$\Gamma = \{0, \$\}$$

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

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

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

Input
Pop
Push



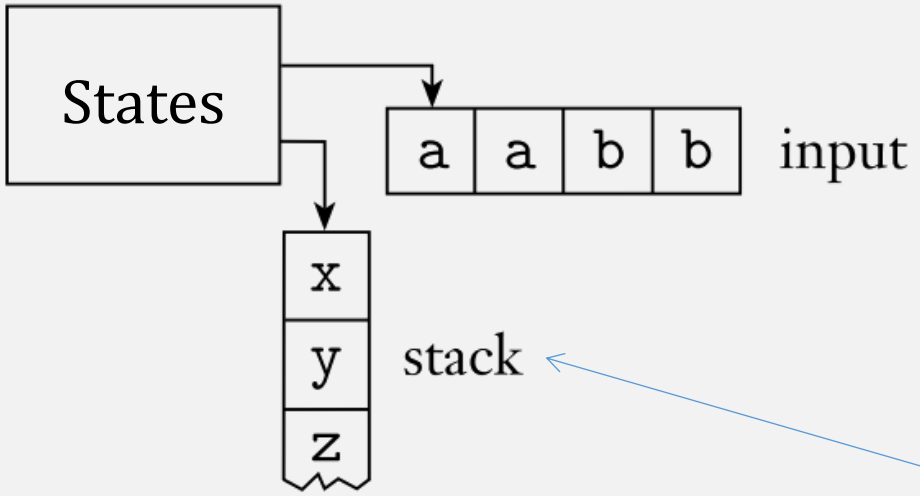
A ***pushdown automaton*** is a **mk-PDA** $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where Q, Σ, Γ, and F are all finite sets, and

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

PDA Computation?

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



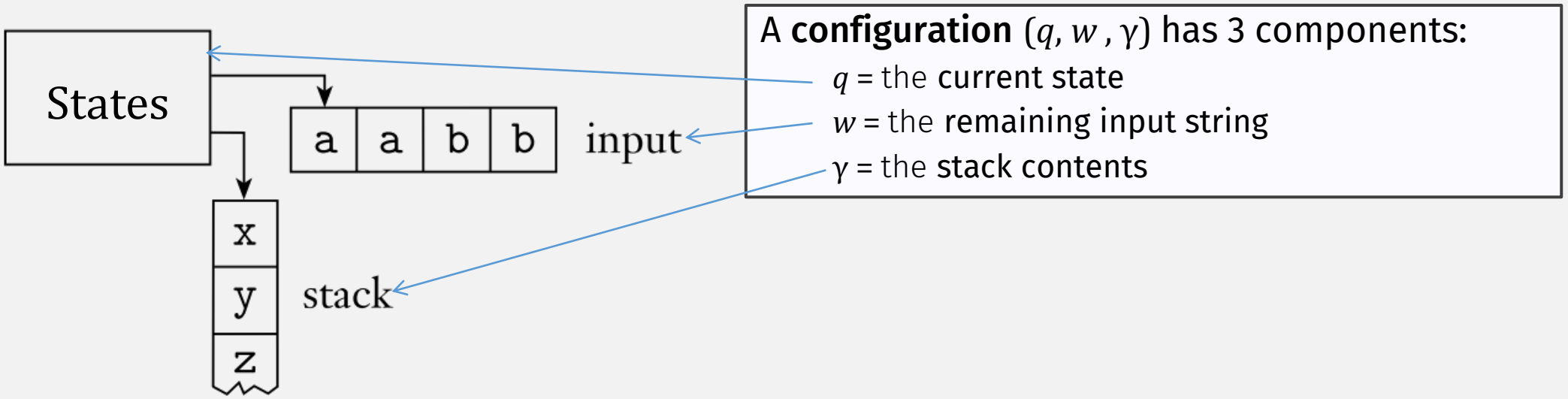
A DFA **computation** is a sequence of states …

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**

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

Read Input | Pop | Less 1 char | Push

"code" if $\delta(q_1, a, X)$ contains (q_2, α)

$q_1, q_2 \in Q$

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

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

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

$I \vdash^* J$ if there exists some ID K such that $I \vdash K$ and $K \vdash^* J$

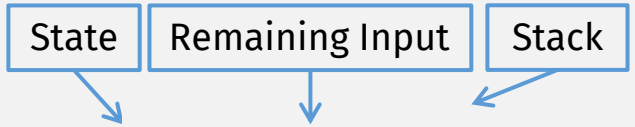
Single step | Recursive "call"

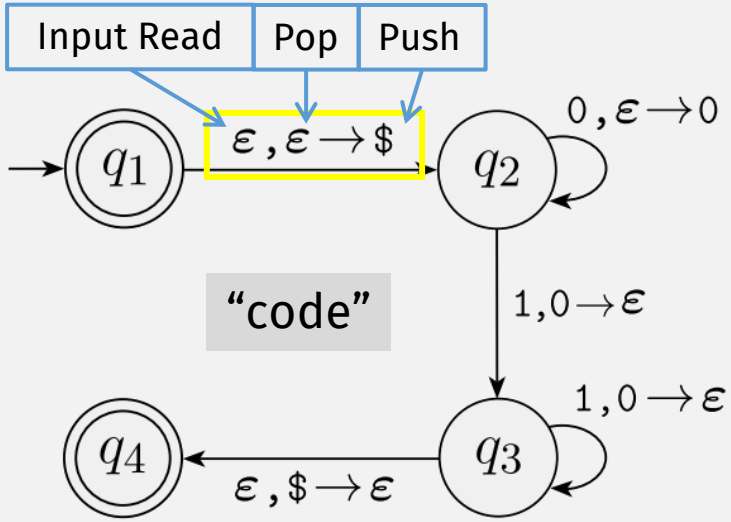
This specifies the **sequence of configurations** for a **PDA** computation

PDA Running Input String Example

"run"

$(q_1, 0011, \varepsilon)$





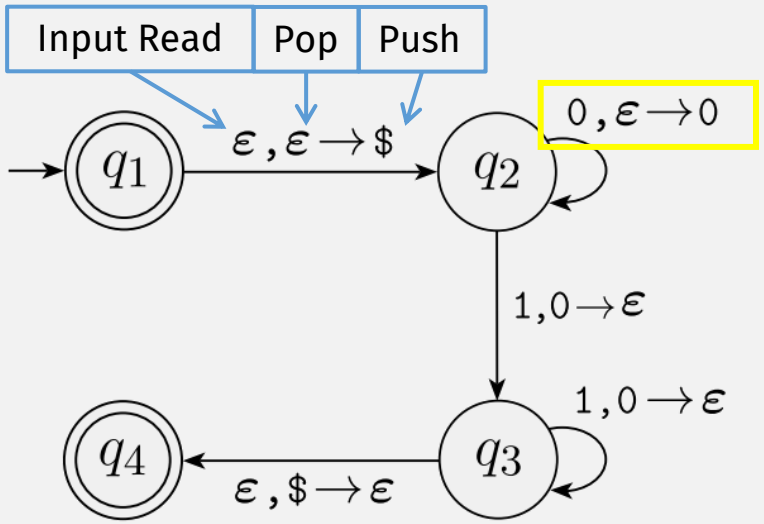
PDA Running Input String Example

State | Remaining Input | Stack

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

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

Read 0, push 0

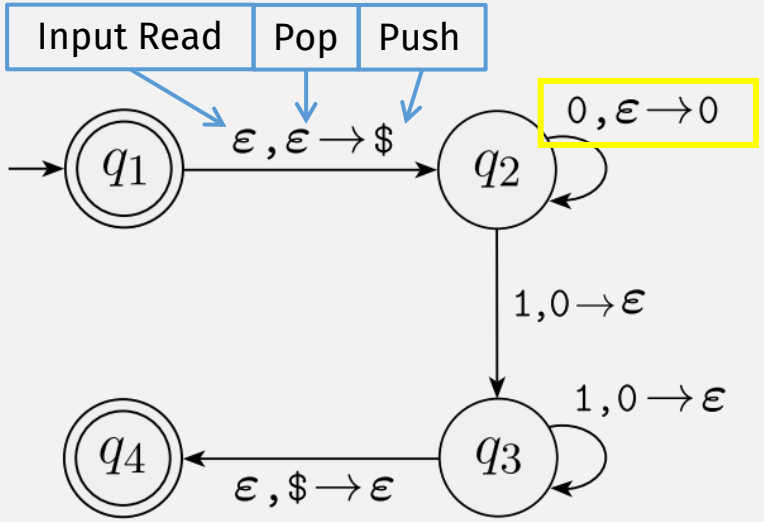


PDA Running Input String Example

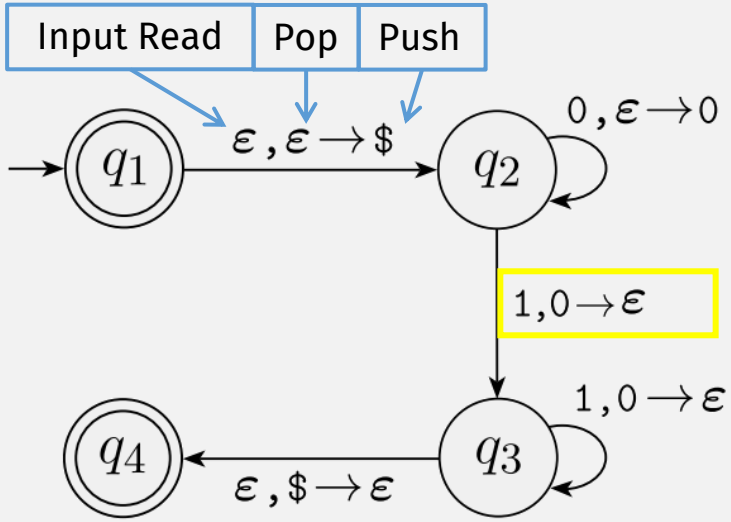
State | Remaining Input | Stack

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

Read 0, push 0



PDA Running Input String Example

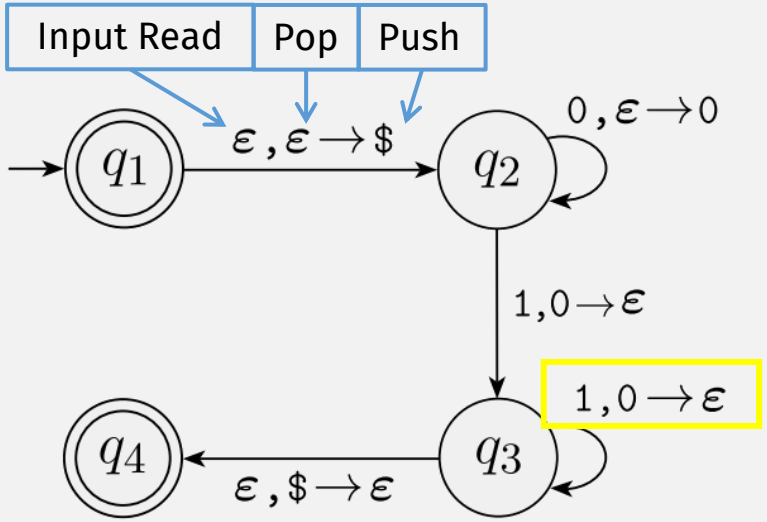


State | Remaining Input | Stack

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

Read 1, pop 0

PDA Running Input String Example

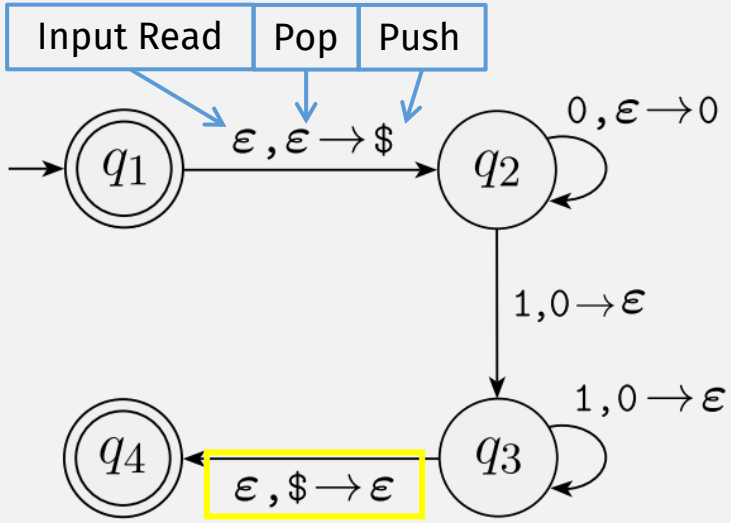


State | Remaining Input | Stack

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

Read 1, pop 0

PDA Running Input String Example



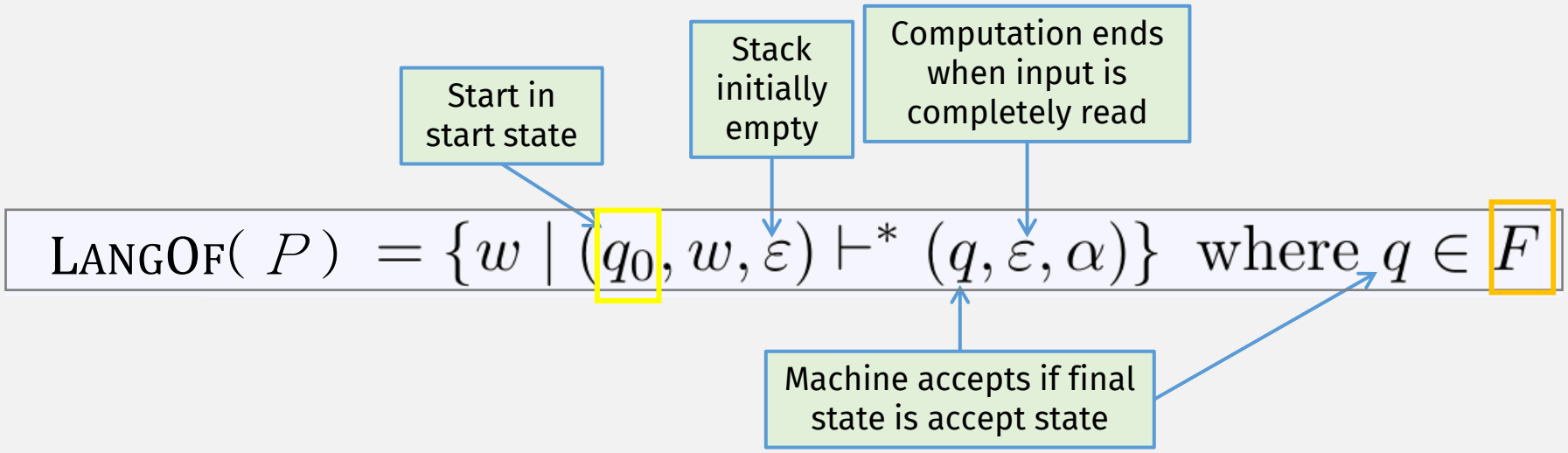
State | Remaining Input | Stack

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

pop empty stack symbol

Language of a PDA

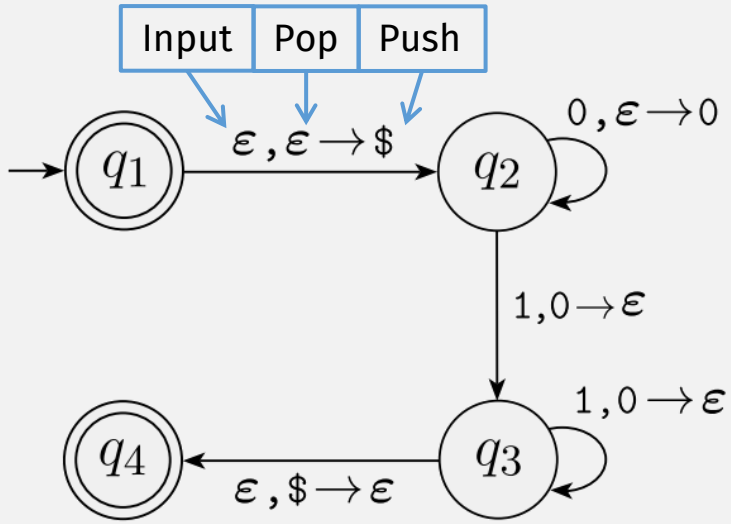
let P = **mk-PDA** (Q, Σ, Γ, δ, q_0, F)
where …



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 ⇔ CFL
- Then, to prove that a language is a CFL, we can either:
  - Create a CFG, or
  - Create a PDA

Regular Language vs CFL Comparison

Regular Languages	Context-Free Languages (CFLs)
Regular Expression	Context-Free Grammar (CFG)
describes a **Regular Lang**	describes a **CFL**
Det. Finite State Automaton (DFA)	**Push-down Automata** (PDA)
recognizes a **Regular Lang**	recognizes a **CFL**
Proved:	Must Prove:
Regular Lang ⇔Regular Expr ☑	CFL ⇔ PDA ???

thm

def

def

thm

L IS-A **CFL** CFLIFFPDA

IF AND ONLY IF

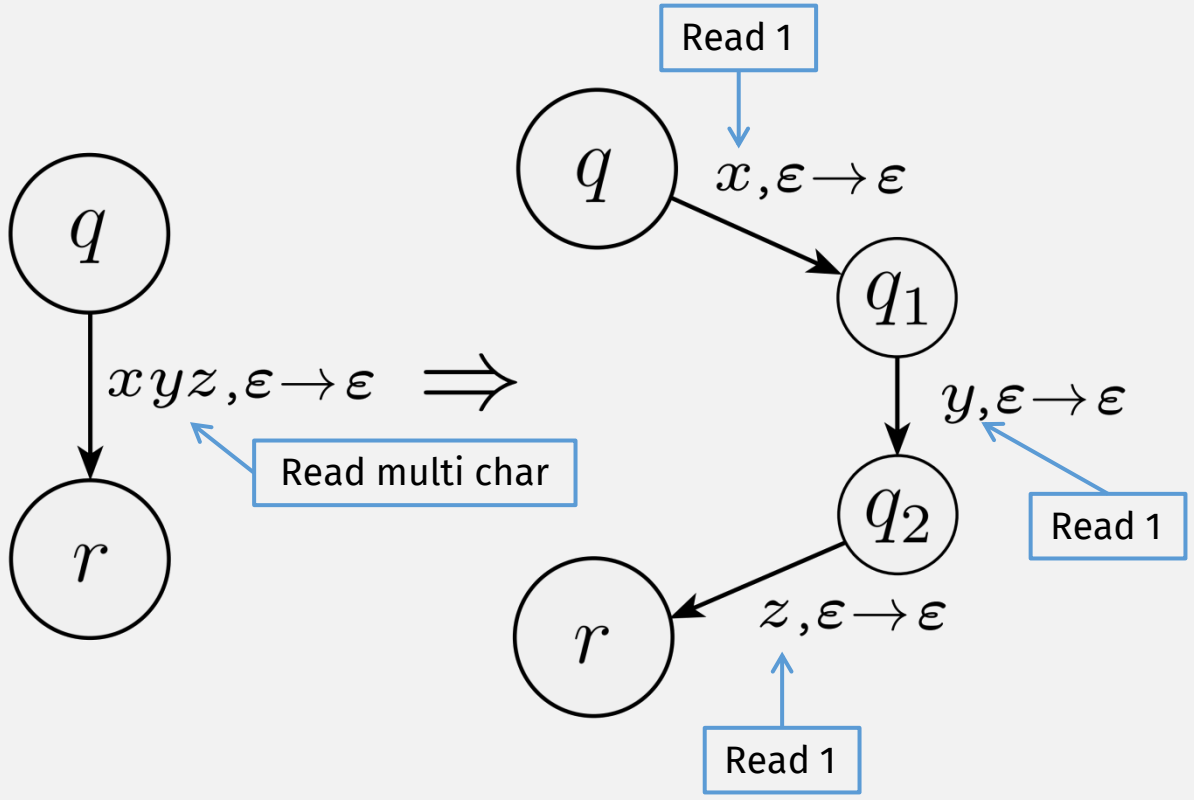
P IS-A **PDA** Machine AND P **recognizes** L

⇒ IF L IS-A **CFL** THEN P IS-A **PDA** Machine AND P **recognizes** L

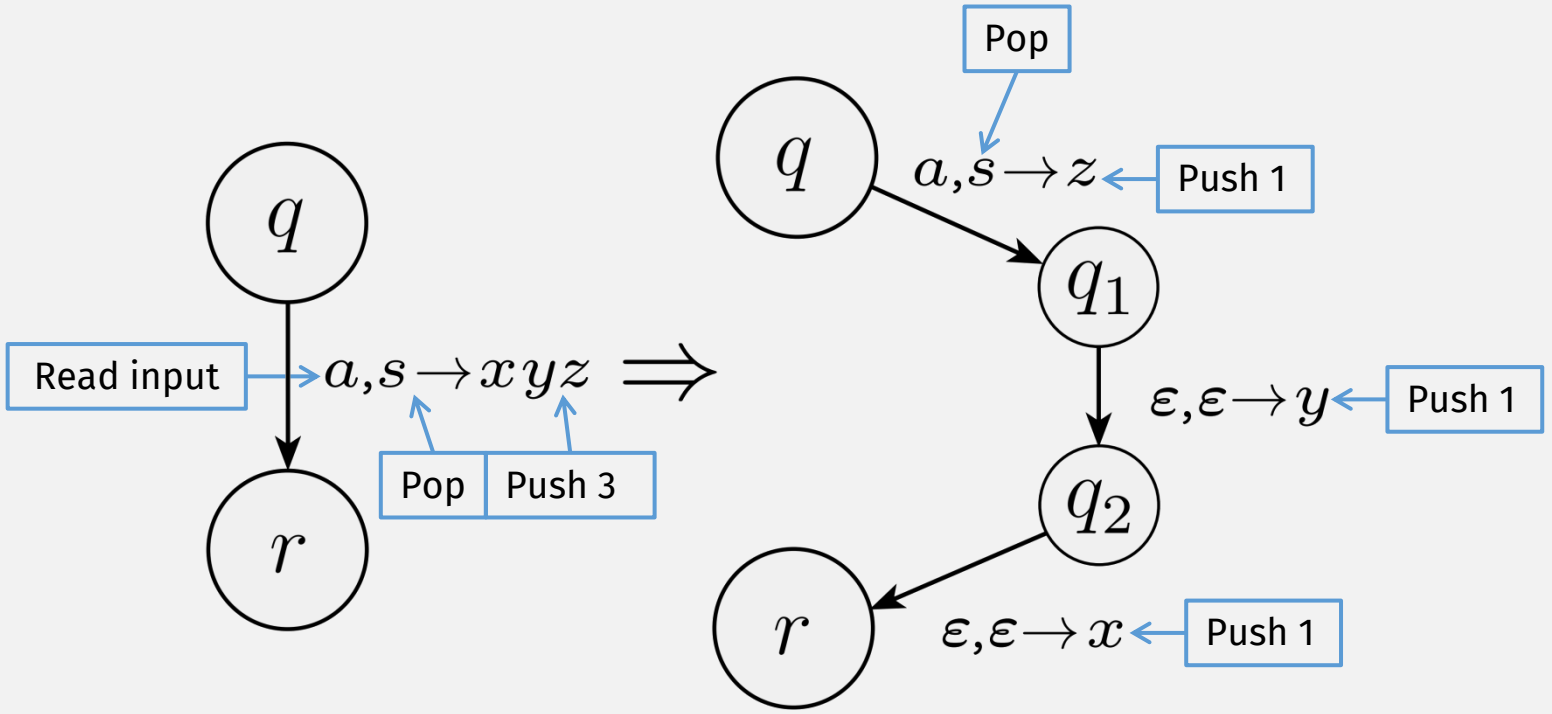
- We know: A **CFL** has a **CFG describing it** (DefCFL)
- To prove this part, show: the **CFG** has an **equivalent** PDA

⇐ IF P IS-A **PDA** Machine AND P **recognizes** L THEN L IS-A **CFL**

Shorthand: Multi-Symbol Read Transition



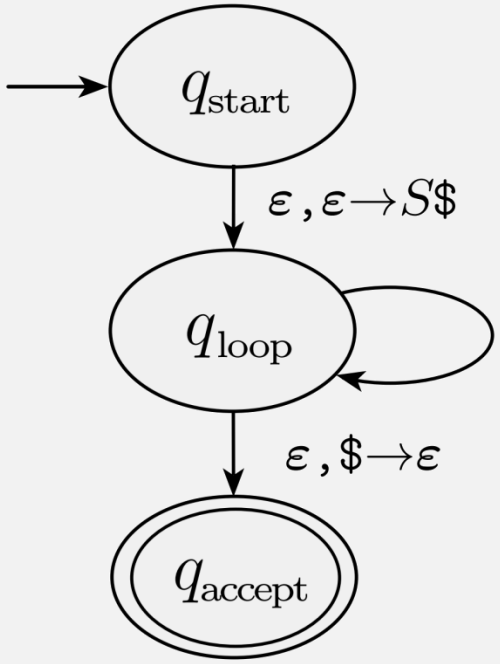
Shorthand: Multi-Stack Push Transition



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

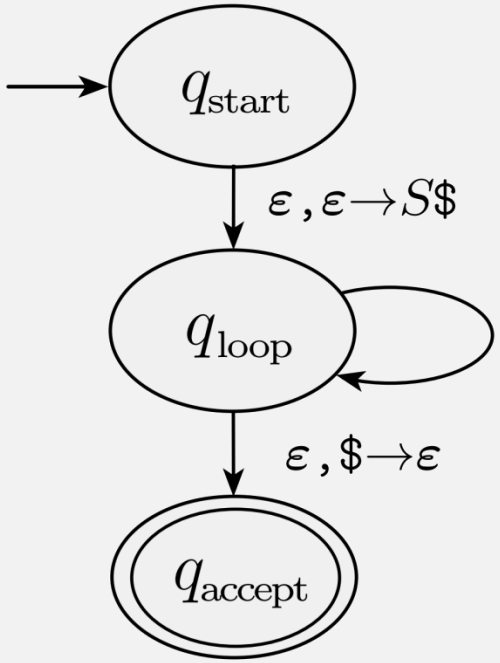


$\varepsilon, A \to w$ for rule $A \to w$

$\mathtt{a}, \mathtt{a} \to \varepsilon$ for terminal a

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



Convert: every CFG rule to PDA "loop" transition that:
- Pops LHS variable
- Pushes RHS

(Stack is "workspace" containing intermediate string of vars + terminals)

$\varepsilon, A \to w$ for rule $A \to w$

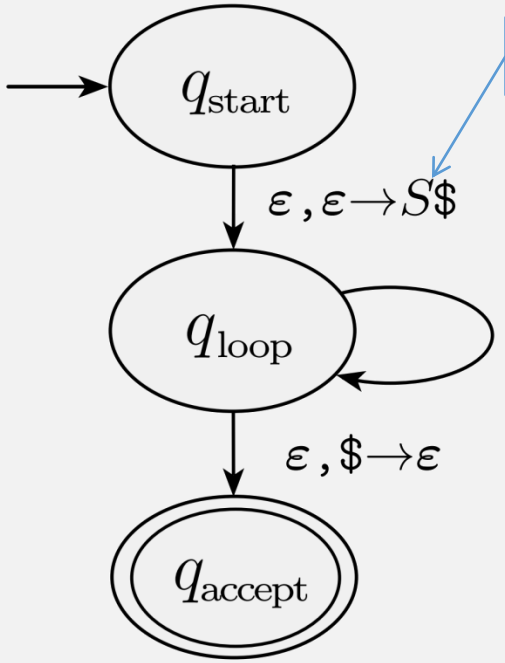
$\mathtt{a}, \mathtt{a} \to \varepsilon$ for terminal a

Convert: every terminal to "loop" transition that:
- Reads input char
- Pops matching char on stack

(Read the terminals as they become known)

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



push **start variable** onto stack

If: stack top is **variable** A, pop and …

… push **rule's right-sides** (nondeterministically)

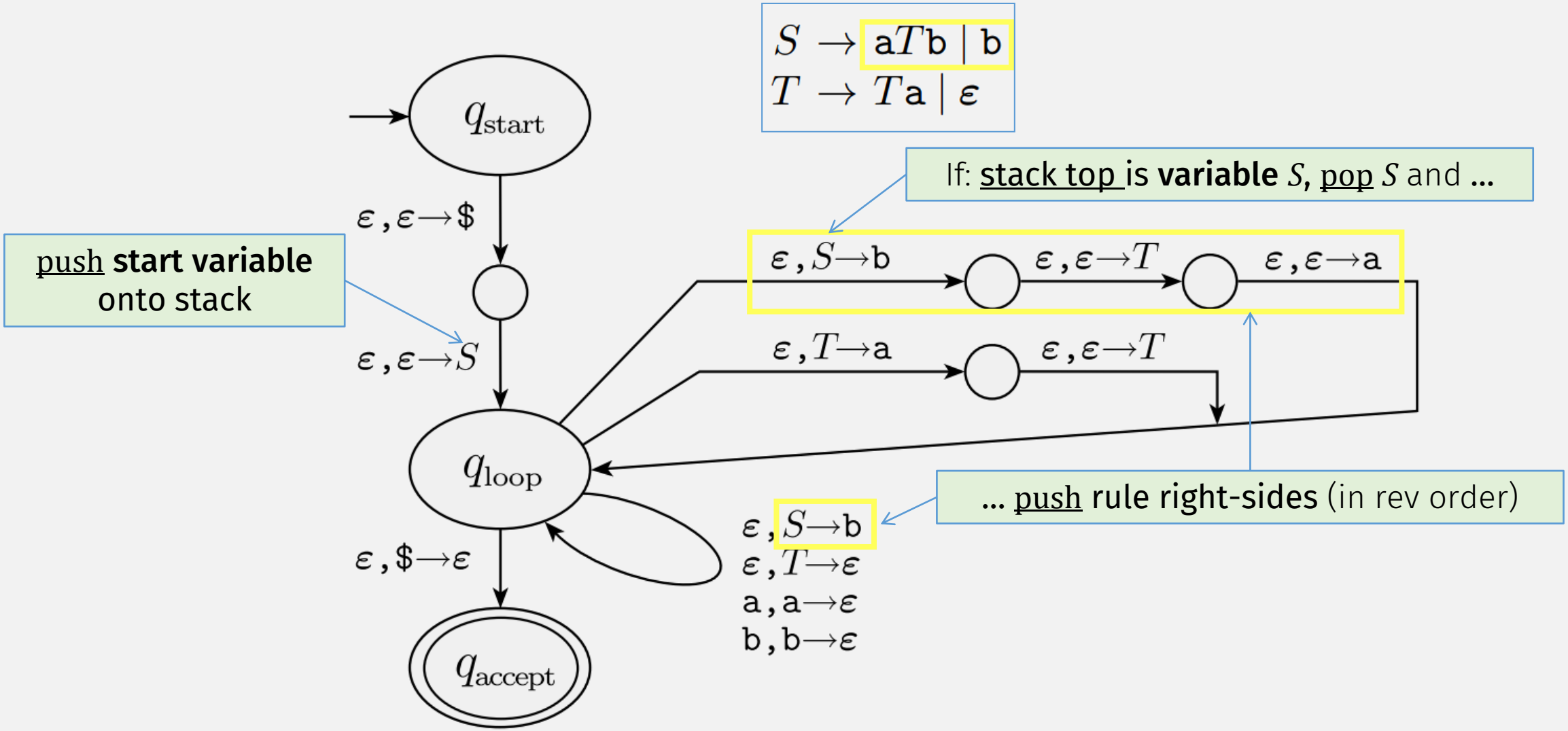
$\varepsilon, A \to w$ for rule $A \to w$

$\mathtt{a}, \mathtt{a} \to \varepsilon$ for terminal a

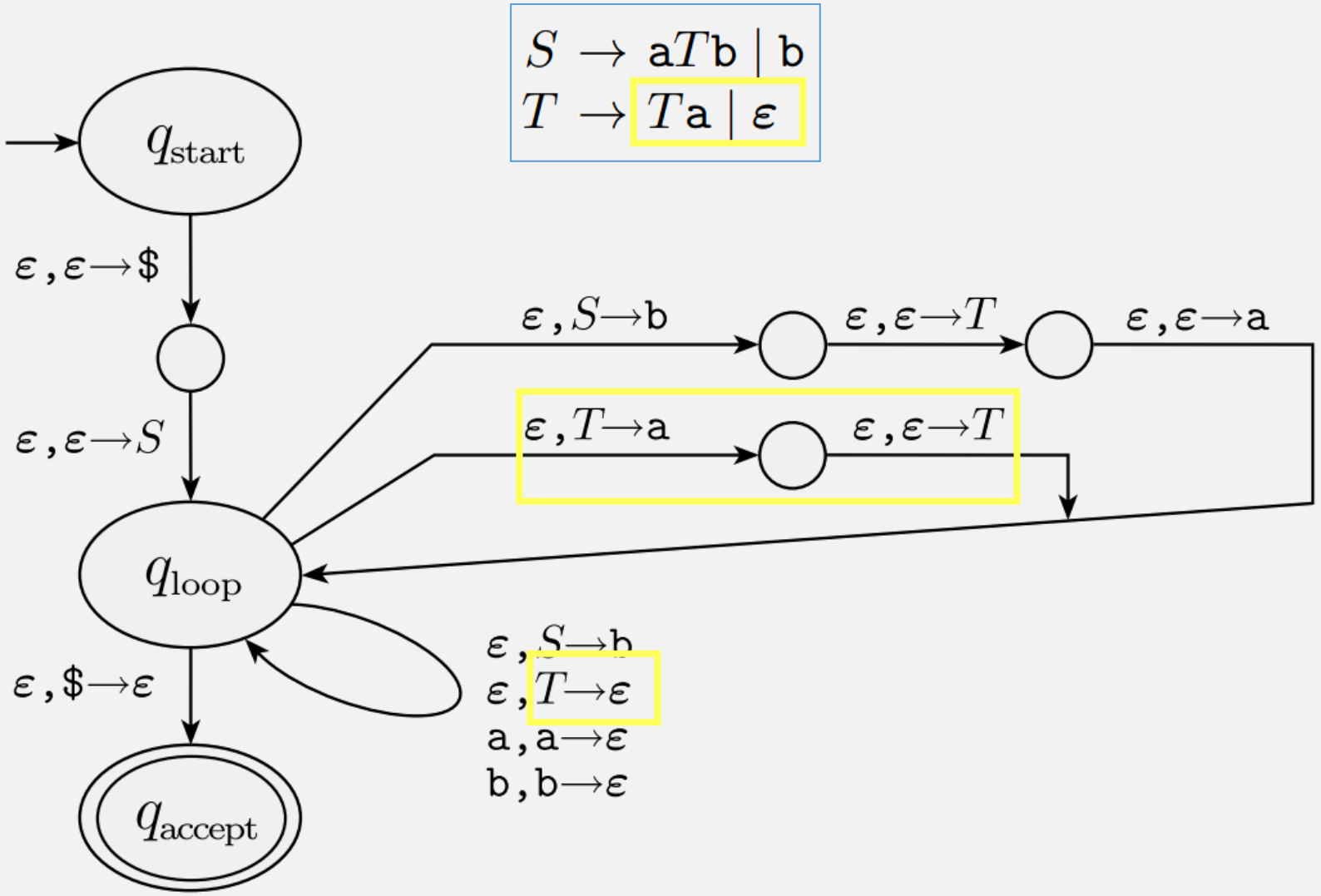
If: stack top is **terminal** a, pop and …

… read matching input

Example **CFG→PDA**

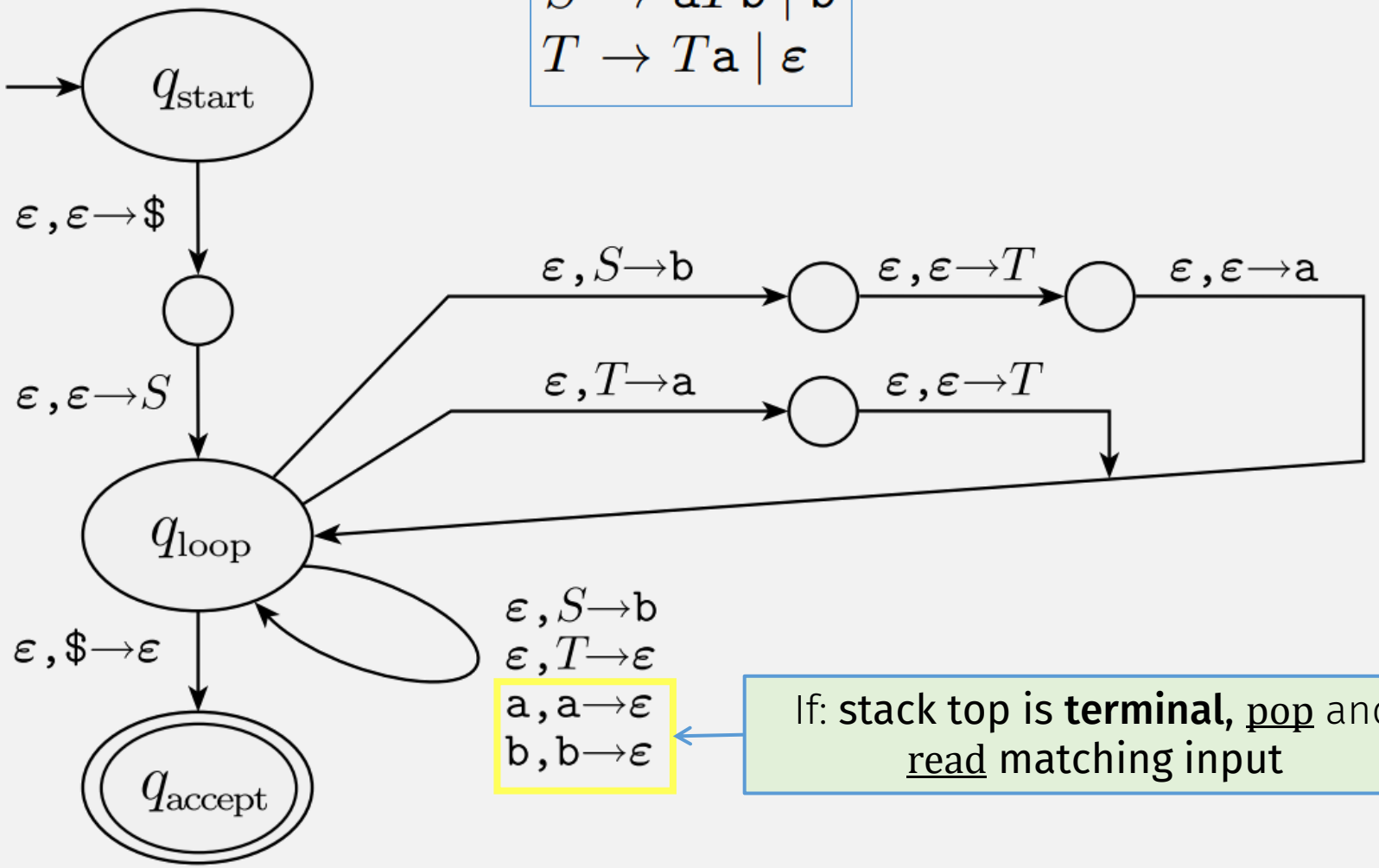


Example **CFG→PDA**



Example **CFG→PDA**

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



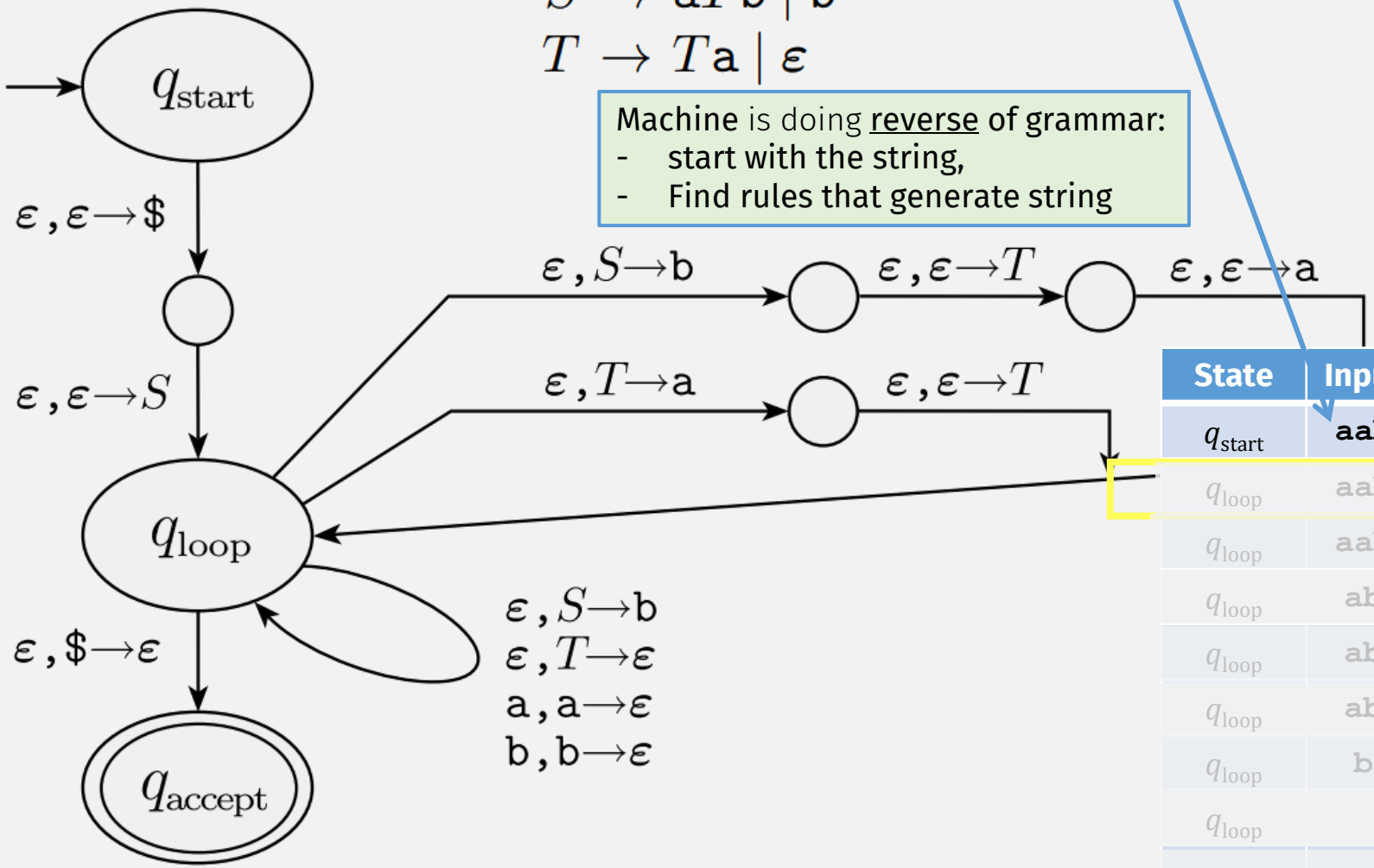
If: stack top is **terminal,** pop and read matching input

Example **CFG→PDA**

Example Derivation using CFG:
$S \Rightarrow \mathtt{a}T\mathtt{b}$ (using rule $S \to \mathtt{a}T\mathtt{b}$)
$\Rightarrow \mathtt{a}T\mathtt{ab}$ (using rule $T \to T\mathtt{a}$)
$\Rightarrow \mathtt{aab}$ (using rule $T \to \varepsilon$)

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

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



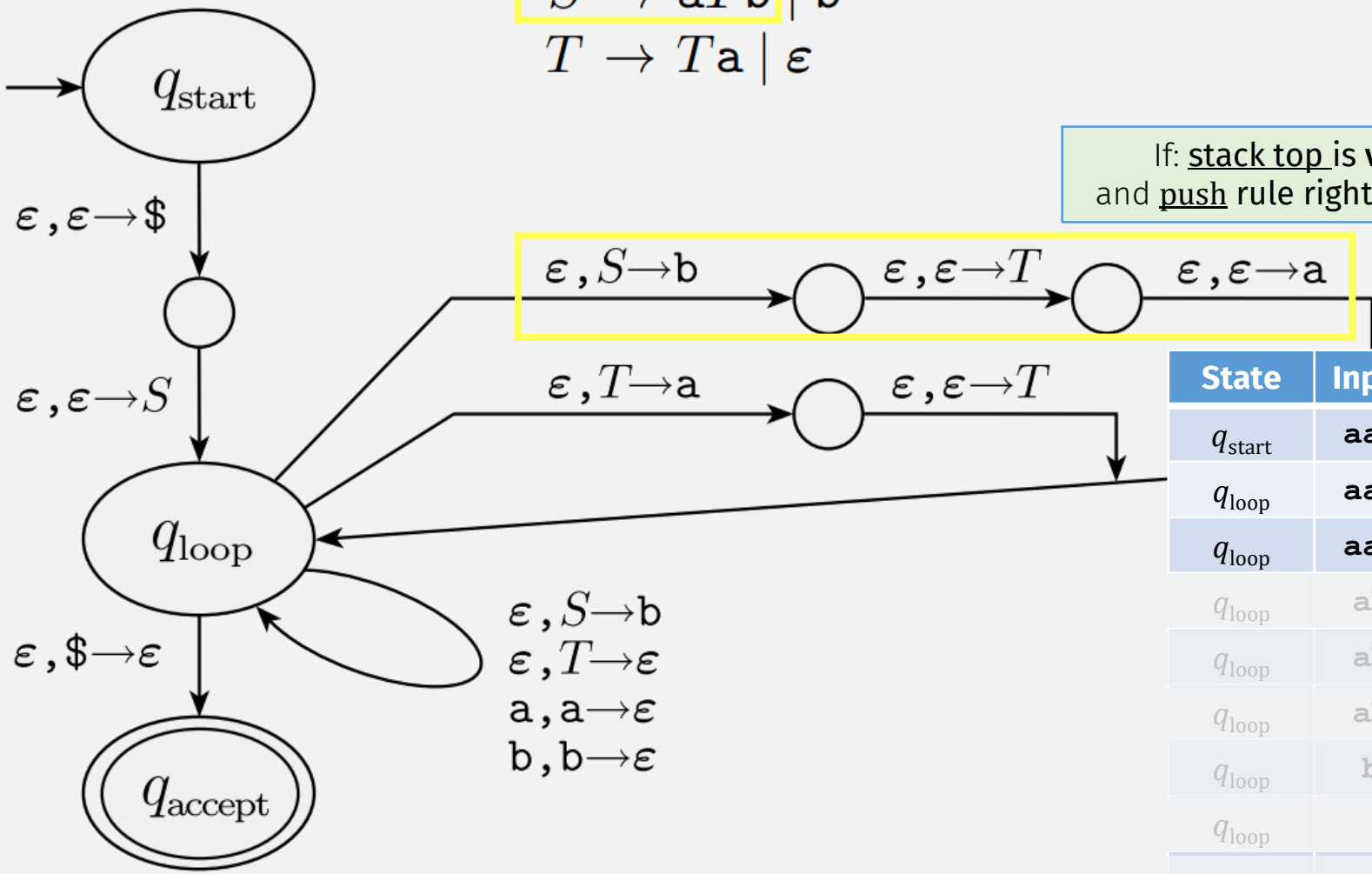
PDA Example

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

Example **CFG→PDA**

Example Derivation using CFG:
$S \Rightarrow \mathtt{a}T\mathtt{b}$ (using rule $S \to \mathtt{a}T\mathtt{b}$)
$\Rightarrow \mathtt{a}T\mathtt{ab}$ (using rule $T \to T\mathtt{a}$)
$\Rightarrow \mathtt{aab}$ (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)

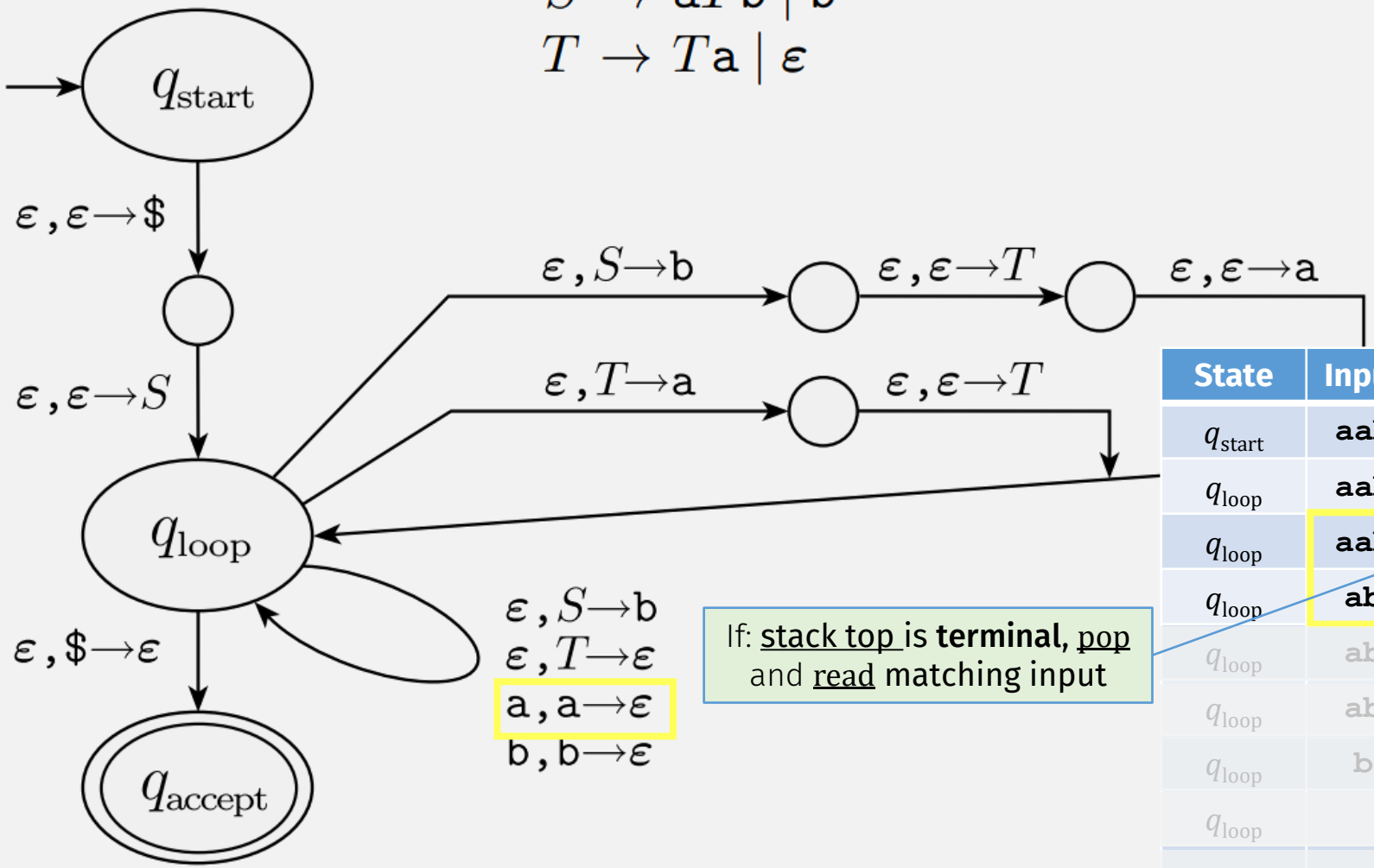
PDA Example

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

Example **CFG→PDA**

Example Derivation using CFG:
$S \Rightarrow \mathtt{a}T\mathtt{b}$ (using rule $S \to \mathtt{a}T\mathtt{b}$)
$\Rightarrow \mathtt{a}T\mathtt{ab}$ (using rule $T \to T\mathtt{a}$)
$\Rightarrow \mathtt{aab}$ (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 **terminal**, pop and read matching input

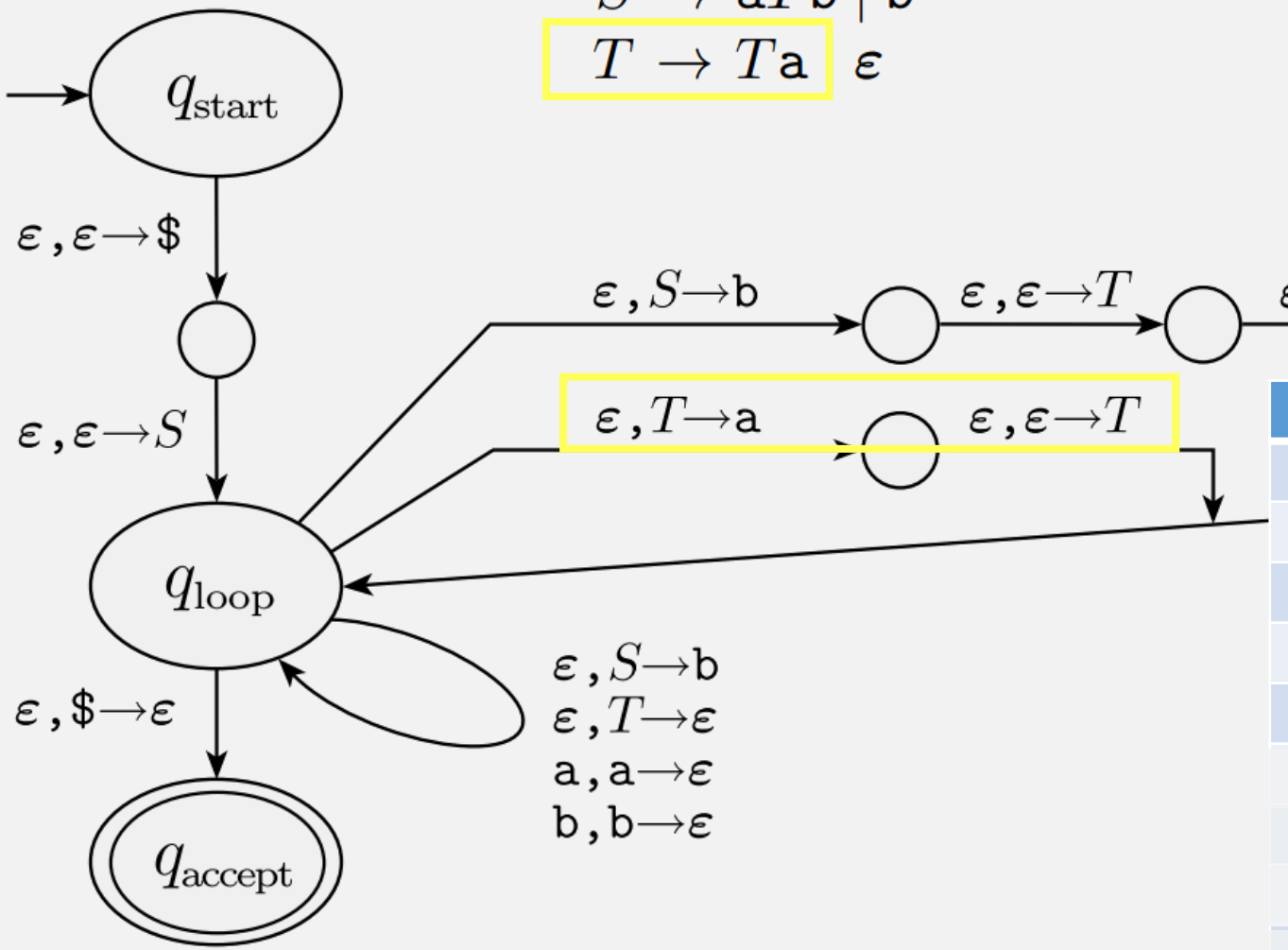
PDA Example

State	Input	Stack	Equiv Rule
q_{start}	**aab**		
q_{loop}	**aab**	S\$	
q_{loop}	**aab**	**a**T**b**\$	$S \to \mathtt{a}T\mathtt{b}$
q_{loop}	**ab**	T**b**\$	
q_{loop}	**ab**	T**ab**\$	$T \to T\mathtt{a}$
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 $S \to \mathtt{a}T\mathtt{b}$)
$\Rightarrow \mathtt{a}T\mathtt{ab}$ (using rule $T \to T\mathtt{a}$)
$\Rightarrow \mathtt{aab}$ (using rule $T \to \varepsilon$)

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



PDA Example

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

L IS-A **CFL** CFLIFFPDA

IF AND ONLY IF

P IS-A **PDA** Machine AND P **recognizes** L

☑ ⇒ IF L IS-A **CFL** THEN P IS-A **PDA** Machine AND P **recognizes** L

⇐ IF P IS-A **PDA** Machine AND P **recognizes** L THEN L IS-A **CFL**

To prove this part: show PDA has an equivalent CFG

PDA→CFG: Prelims

Before converting PDA to CFG, <u>modify</u> 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.

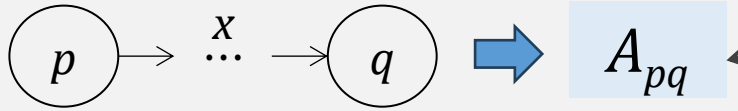
<u>Important</u>:
This **doesn't change** the **language recognized by the PDA**

PDA P -> CFG G: Transitions and Variables

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

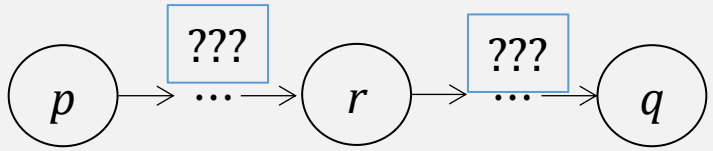
variables of G are $\{A_{pq} \mid p, q \in Q\}$

- 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 = `mk-PDA` $(Q, \Sigma, \Gamma, \delta, q_0, \{q_{\text{accept}}\})$

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

PDA P -> CFG G: Generating Strings

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

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

PDA P -> CFG G : Generating Strings

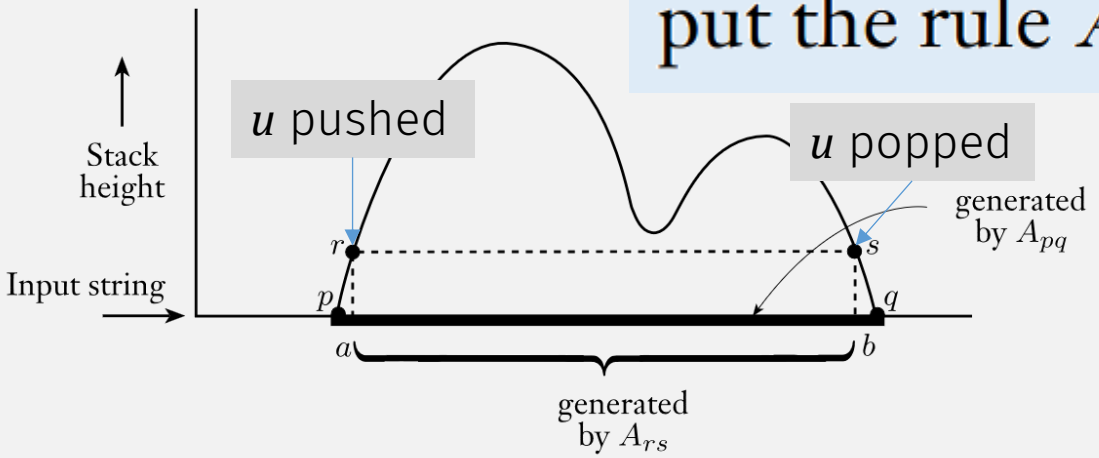
P = **mk-PDA** $(Q, \Sigma, \Gamma, \delta, q_0, \{q_{\text{accept}}\})$

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



L IS-A **CFL** CFLIFFPDA

IF AND ONLY IF

P IS-A **PDA** Machine AND P **recognizes** L

☑ ⇒ IF L IS-A **CFL** THEN P IS-A **PDA** Machine AND P **recognizes** L

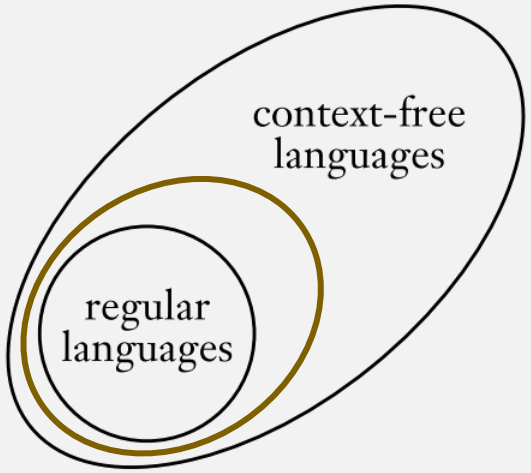
key step: **Convert CFG→PDA**

☑ ⇐ IF P IS-A **PDA** Machine AND P **recognizes** L THEN L IS-A **CFL**

key step: **Convert PDA→CFG**

■

Regular vs Context-Free Languages
(and others?)

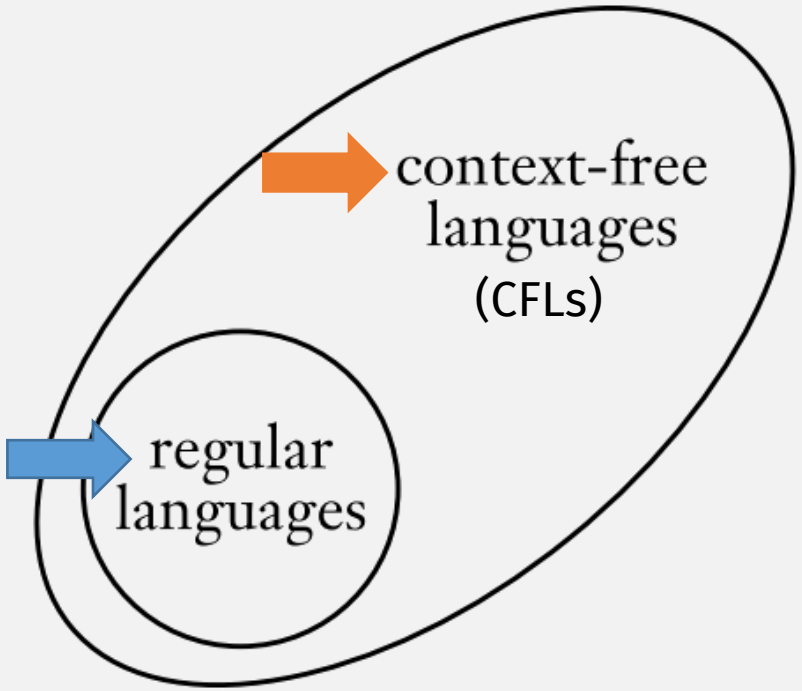


Is This Diagram “Correct”?

(What are the statements implied by this diagram?)

1. Every regular language is a CFL

2. Not every CFL is a regular language



How to Prove This Diagram "Correct"?

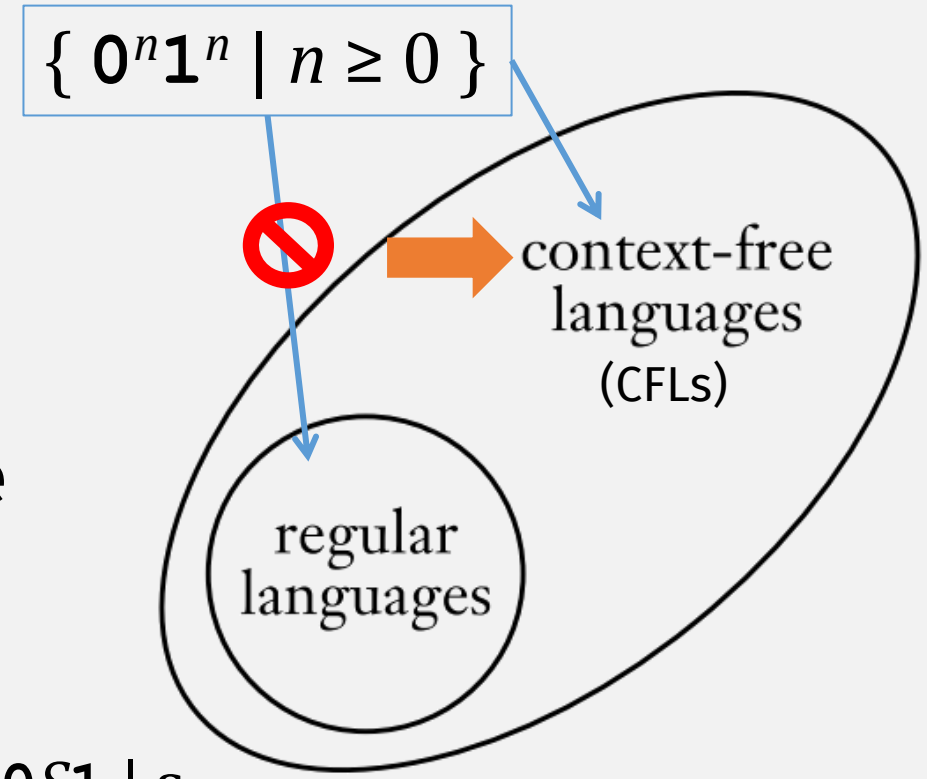
1. Every regular language is a CFL

2. Not every CFL is a regular language

Find a counterexample CFL that is not regular

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

- It's a CFL
  - *Prove with* : CFG $S \rightarrow \mathtt{0}S\mathtt{1} \mid \varepsilon$
- It's not a Regular Lang
  - *Proof*: by contradiction using ThePumpingLemma



How to Prove This Diagram "Correct"?

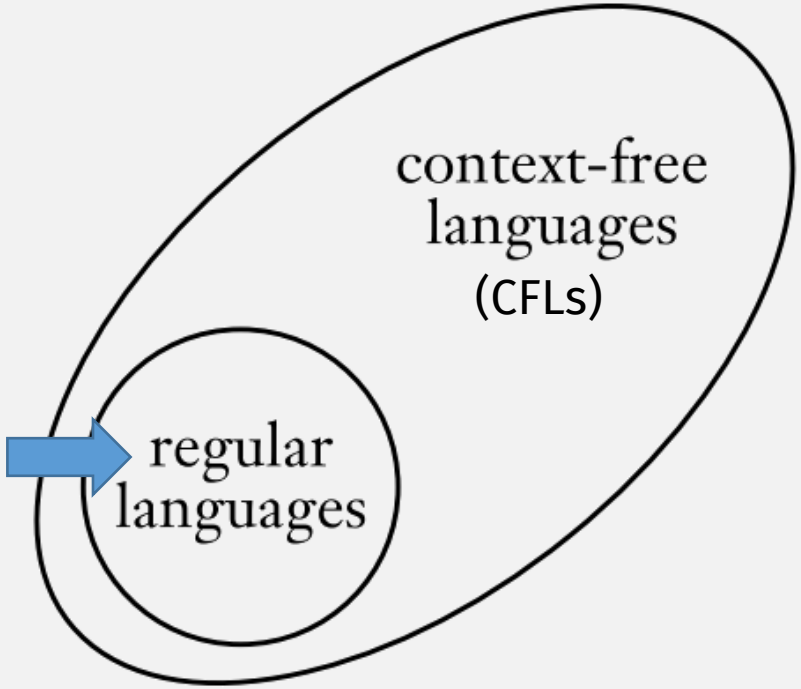
1. Every regular language is a CFL

For any regular language A, show …

… it has a **CFG** or **PDA**

☑ 2. Not every CFL is a regular language

A regular language is represented by a:
- DFA
- NFA
- Regular Expression



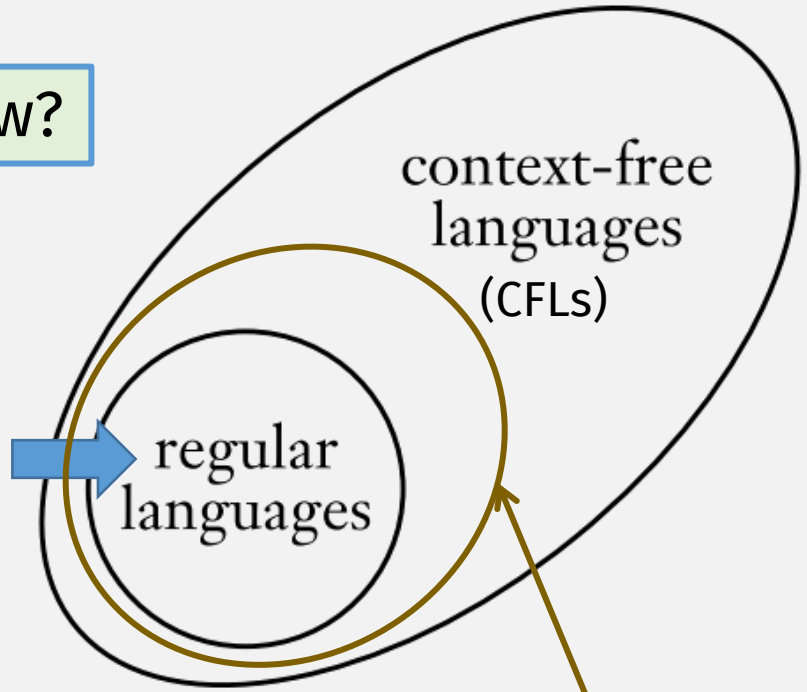
Regular Languages are CFLs: 3 Ways to Prove

- DFA → CFG or PDA

Coming soon to a future hw?

- NFA → CFG or PDA

- Regular expression → CFG or PDA



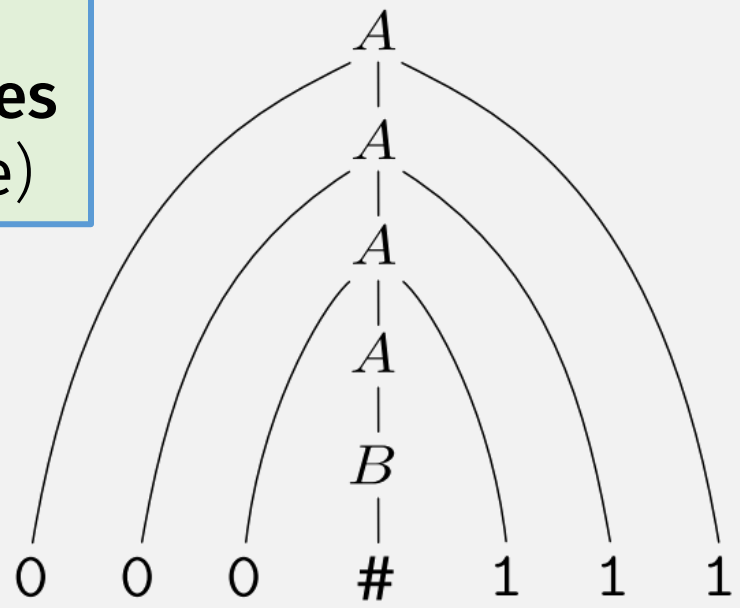
Are there other interesting subsets of CFLs?

Deterministic CFLs and DPDAs

Previously: Generating Strings

Generating strings:
1. Start with **start variable,**
2. Repeatedly *apply* **CFG rules**
to **get string** (and **parse tree**)

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

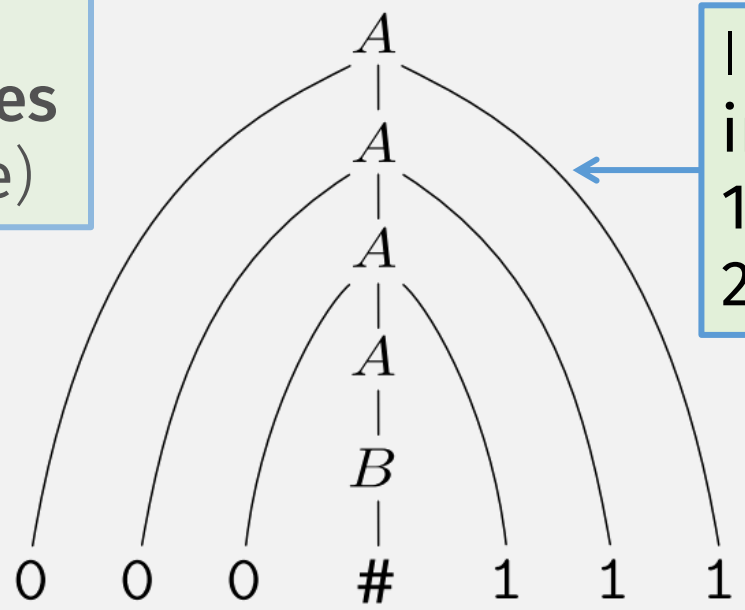


$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000B111 \Rightarrow 000\#111$$

Generating vs Parsing

Generating strings:
1. Start with **start variable,**
2. Repeatedly *apply* **CFG rules**
to get string (and **parse tree**)

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



In practice, opposite is more interesting:
1. Start with **string,**
2. Then **parse** into **parse tree**

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

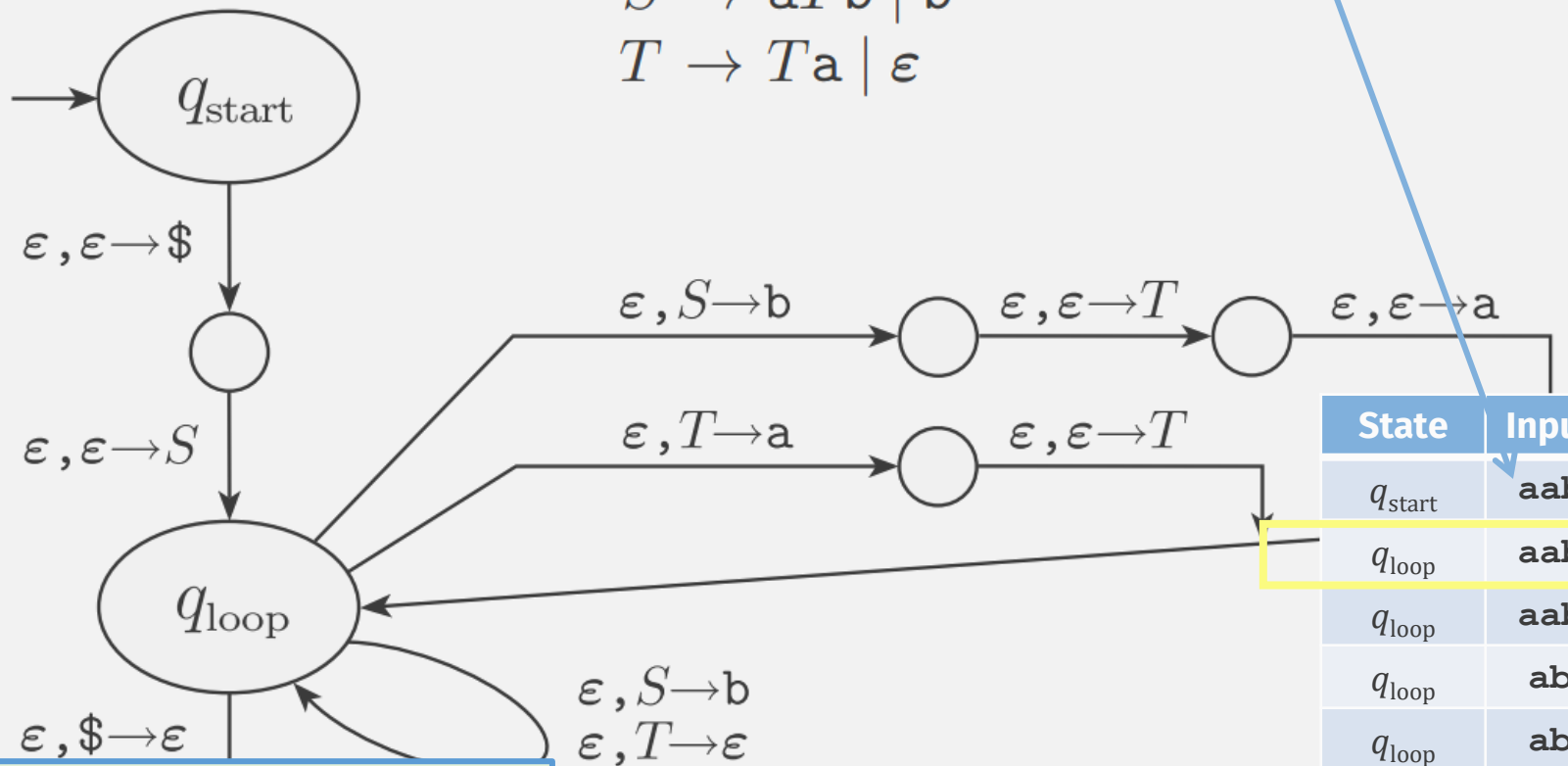
Generating vs Parsing

- In practice, **parsing a string more important** than **generating**
  - E.g., **a compiler** (first) **parses source code string** into a **parse tree**
  - (Actually, *any* program with string inputs must first parse it)

Previously: Example **CFG→PDA**

Example Derivation using CFG:
$S \Rightarrow \mathtt{a}T\mathtt{b}$ (using rule $S \to \mathtt{a}T\mathtt{b}$)
$\Rightarrow \mathtt{a}T\mathtt{ab}$ (using rule $T \to T\mathtt{a}$)
$\Rightarrow \mathtt{aab}$ (using rule $T \to \varepsilon$)

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



PDA Example

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

This Machine is **parsing**!
1. Start with (input) string,
2. Find **rules** that **generate** string

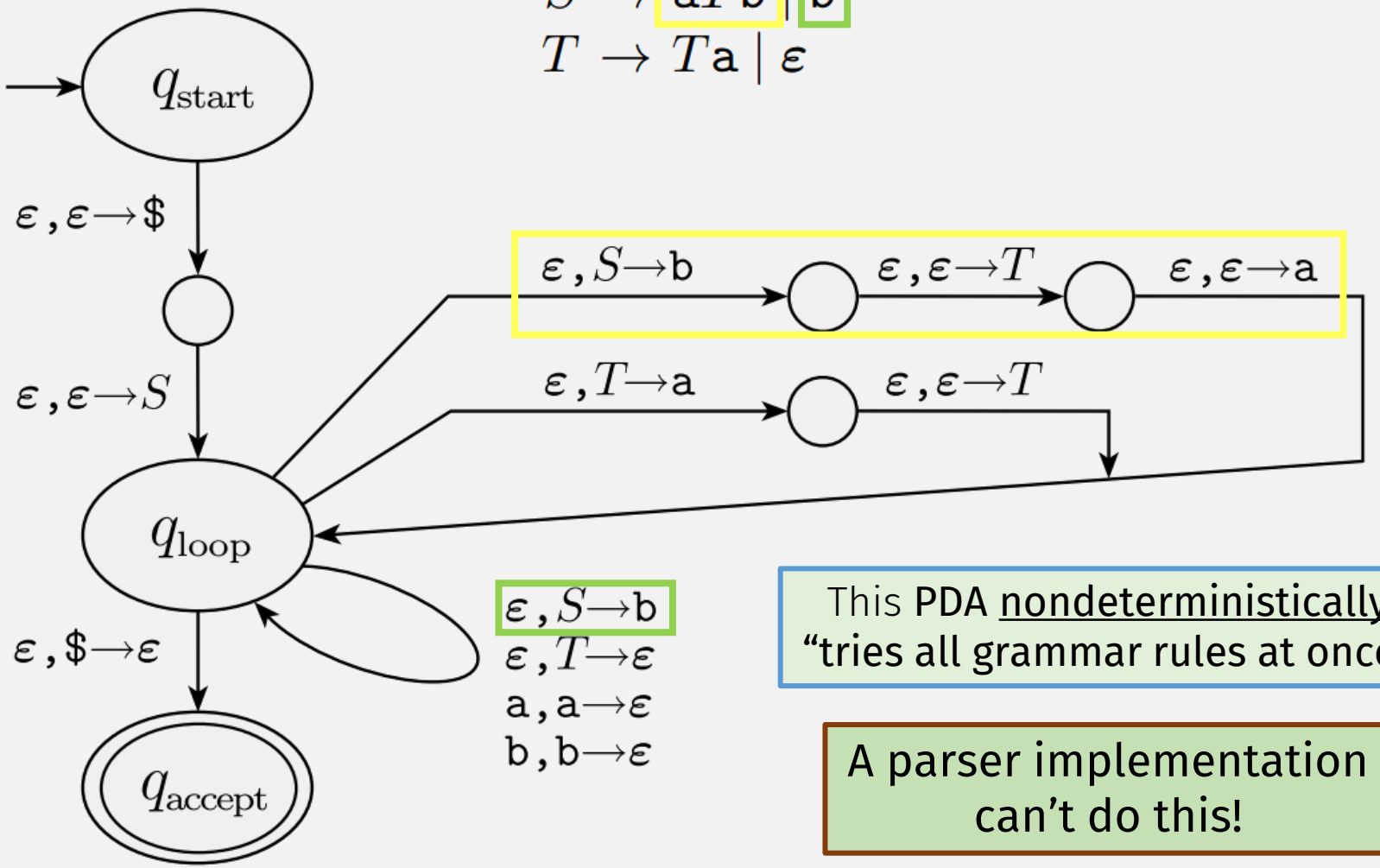
Generating vs Parsing

- In practice, **parsing** a string more important than **generating**
  - E.g., a **compiler** (first step) **parses source code string** into a **parse tree**
  - (Actually, *any* program with string inputs must first parse it)

- But: the **PDAs** we've seen are **non-deterministic** (like **NFAs**)

Previously: (Nondeterministic) PDA

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



This PDA nondeterministically "tries all grammar rules at once"

A parser implementation can't do this!

Generating vs Parsing

- In practice, **parsing** a string more important than **generating** one
  - E.g., a **compiler** (first step) **parses source code into a parse tree**
  - (Actually, *any* program with string inputs must first parse it)

- But: **the PDAs we've seen are** non-deterministic (like NFAs)

- **Compiler's parsing algorithm must be** deterministic

- So: to **model parsers,** we need a **Deterministic PDA** (DPDA)

DPDA: Formal Definition

The language of a DPDA is called a ***deterministic context-free language***.

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

"do nothing"

Not power set!

A ***pushdown automaton*** is a `mk-PDA`

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

Difference: DPDA has only one possible action, for any given state, input, and stack op (similar to DFA vs NFA)

Must consider: ε reads or stack ops!
E.g., if $\delta(q, a, X)$ does "something", then $\delta(q, \varepsilon, X)$ must "do nothing"

DPDAs are Not Equivalent to PDAs!

$$R \to S \mid T$$
$$S \to \mathtt{a}S\mathtt{b} \mid \mathtt{ab}$$
$$T \to \mathtt{a}T\mathtt{bb} \mid \mathtt{abb}$$

- A PDA can non-deterministically "try all rules" (abandoning failed attempts)
- A DPDA must choose one rule at each step! (cant go back after reading input!)

Parsing = deriving reversed: start with string, end with parse tree

used S rule

used T rule

$\mathtt{aaa}\underline{\mathtt{a}}\mathtt{bbb} \rightarrowtail \mathtt{a}\underline{\mathtt{a}}\mathtt{S}\underline{\mathtt{b}}\mathtt{b}$

$\mathtt{aaa}\underline{\mathtt{a}}\mathtt{bbbbbb} \rightarrowtail \mathtt{a}\underline{\mathtt{a}}\mathtt{T}\underline{\mathtt{bb}}\mathtt{bb}$

When parsing this string, when does it know which rule was used, S or T?

Choosing "correct" rule depends on rest of the input!

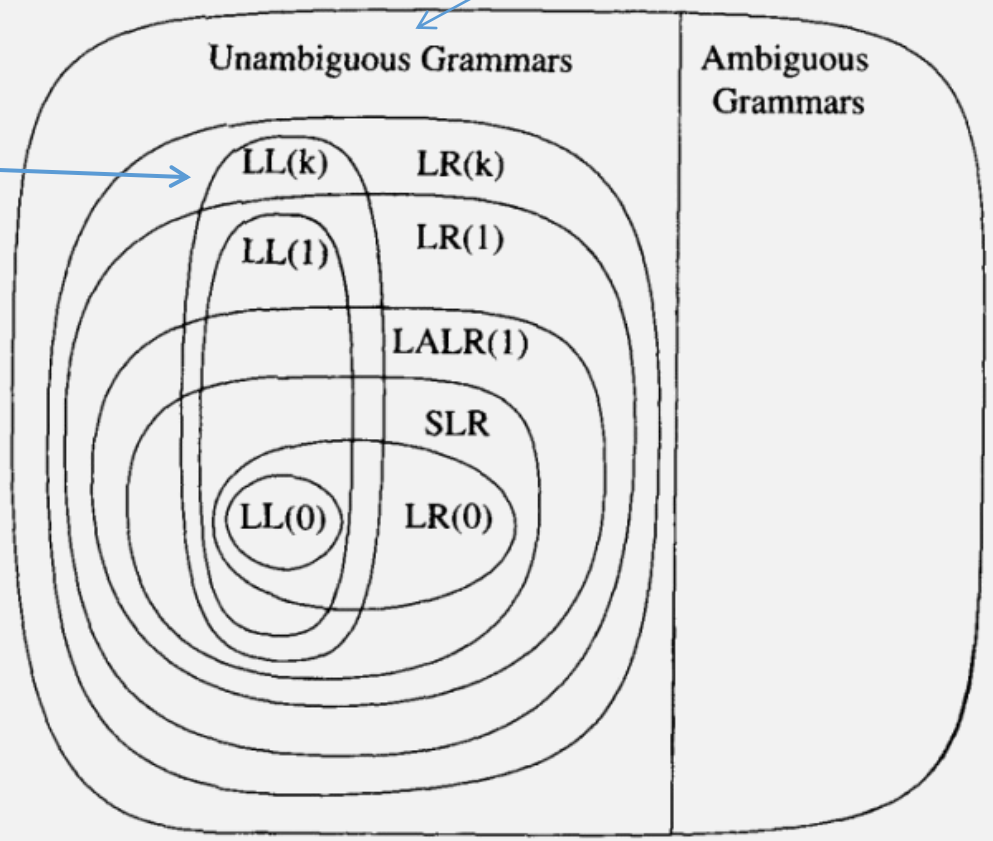
PDAs recognize CFLs, but **DPDAs** only recognize **DCFLs!** (a subset of CFLs)

Subclasses of CFLs

Umambiguous CFLs / PDAs

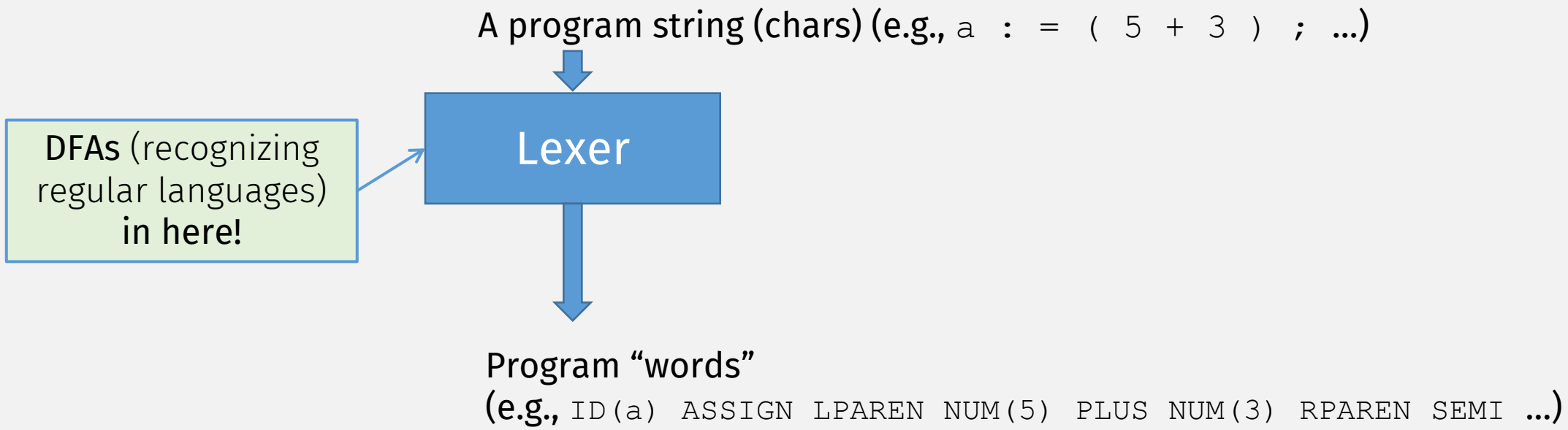
DCFLs

Programming language parsers / compilers are ideally in here



All CFLS

Compiler Stages



A Lexer Implementation

Remember our analogy:
- **DFAs** are like **programs**
- **All possible DFA tuples** is like a **programming language**

It's more than an analogy!

This DFA is a real program!

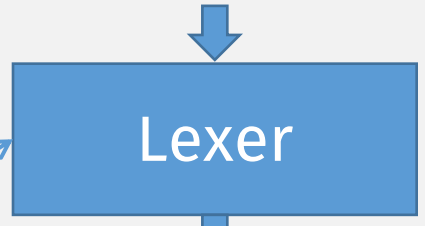
A "`lex`" tool converts the program:
- from "DFA Lang" ...
- to an **equivalent** one in C !

DFAs (represented as **regular expressions**)!

```
%{
/* C Declarations: */
#include "tokens.h"    /* definitions of IF, ID, NUM, ... */
#include "errormsg.h"
union {int ival; string sval; double fval;} yylval;
int charPos=1;
#define ADJ  (EM_tokPos=charPos, charPos+=yyleng)
%}
/* Lex Definitions: */
digits   [0-9]+
%%
/* Regular Expressions and Actions: */
if                          {ADJ; return IF;}
[a-z][a-z0-9]*              {ADJ; yylval.sval=String(yytext);
                              return ID;}
{digits}                   {ADJ; yylval.ival=atoi(yytext);
                              return NUM;}
({digits}"."[0-9]*)|([0-9]*"."{digits})      {ADJ;
                              yylval.fval=atof(yytext);
                              return REAL;}
("--"[a-z]*"\n")|(" "|"\n"|"\t")+     {ADJ;}
.                           {ADJ; EM_error("illegal character");}
```

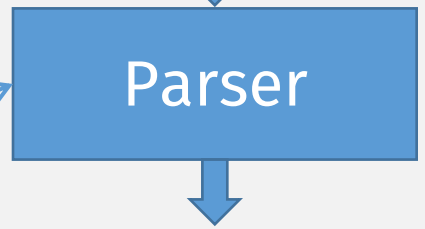
Compiler Stages

A program (chars) (e.g., `a : = ( 5 + 3 ) ; ...`)

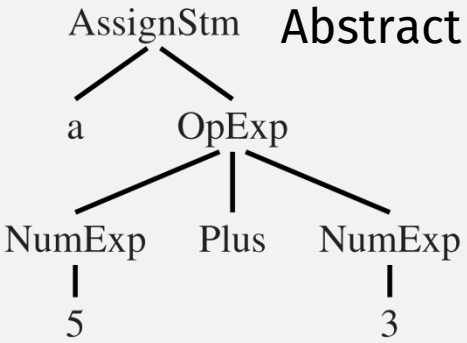


DFAs (recognizing regular languages) **in here!**

Program "words"
(e.g., `ID(a) ASSIGN LPAREN NUM(5) PLUS NUM(3) RPAREN SEMI ...`)



DPDAs (recognizing DCFLs) **in here!**



Abstract Syntax tree (AST), i.e., a **parse tree!**

A Parser Implementation

```
%{
int yylex(void);
void yyerror(char *s) { EM_error(EM_tokPos, "%s", s); }
%}
%token ID WHILE BEGIN END DO IF THEN ELSE SEMI ASSIGN
%start prog
%%

prog: stmlist

stm : ID ASSIGN ID
    | WHILE ID DO stm
    | BEGIN stmlist END
    | IF ID THEN stm
    | IF ID THEN stm ELSE stm

stmlist : stm
        | stmlist SEMI stm
```

Just write the CFG!

Remember our analogy:
CFGs are like **programs**

It's more than an analogy!

This CFG **is** a **real program!**

A "`yacc`" tool converts the program:
- from "CFG Lang" ...
- to an **equivalent** one in C !

DPDAs are Not Equivalent to PDAs!

Parsing = generating reversed:
- start with string
- end with parse tree

$$R \to S \mid T$$
$$S \to \mathrm{a}S\mathrm{b} \mid \mathrm{ab}$$
$$T \to \mathrm{a}T\mathrm{bb} \mid \mathrm{abb}$$

- **PDA**: can non-deterministically "try all rules" (abandoning failed attempts);
- **DPDA**: must choose one rule at each step!

Should use S rule

aaabbb ⟼ aaSbb

aaa

Should use T rule

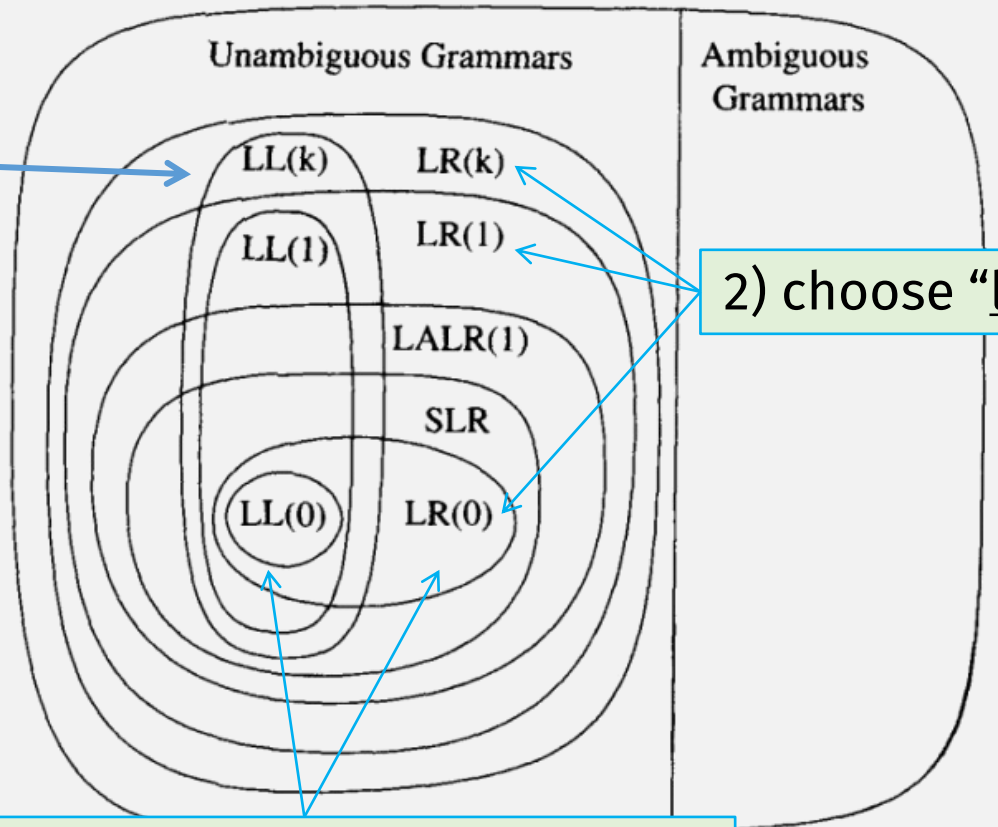
aaabbbbbb ⟼ aaTbbbb

When parsing reaches this position, does it know which rule, S or T?

To choose "correct" rule, need to "look ahead" at rest of the input!

PDAs recognize CFLs, but **DPDAs** only recognize **DCFLs!** (a subset of CFLs)

Subclasses of CFLs



DCFLs

Programming language parsers / compilers are ideally in here

2) choose "look ahead" amount

2 parser design decisions:
1) Parse from left, or from right

All CFLS

LL parsing

- **L** = left-to-right
- **L** = leftmost derivation

Let's play a game: "You're the Parser": Guess which rule applies?

(and how much did you have to "look ahead"?)

1 $S \rightarrow$ if E then S else S

2 $S \rightarrow$ begin S L

3 $S \rightarrow$ print E

4 $L \rightarrow$ end

5 $L \rightarrow$; S L

6 $E \rightarrow$ num = num

```
if 0 = 1 then begin print 2 = 3; print 4 = 5 end else print 6 = 7
```



LL parsing

- **L** = left-to-right
- **L** = leftmost derivation

1 $S \rightarrow \text{if } E \text{ then } S \text{ else } S$

2 $S \rightarrow \text{begin } S\ L$

3 $S \rightarrow \text{print } E$

4 $L \rightarrow \text{end}$

5 $L \rightarrow ;\ S\ L$

6 $E \rightarrow \text{num} = \text{num}$

```
if 0 = 1 then begin print 2 = 3; print 4 = 5 end else print 6 = 7
```



LL parsing

- **L** = left-to-right
- **L** = leftmost derivation

1 $S \rightarrow \text{if } E \text{ then } S \text{ else } S$

2 $S \rightarrow \text{begin } S\ L$

3 $S \rightarrow \text{print } E$

4 $L \rightarrow \text{end}$

5 $L \rightarrow ;\ S\ L$

6 $E \rightarrow \text{num} = \text{num}$

```
if 0 = 1 then begin print 2 = 3; print 4 = 5 end else print 6 = 7
```



LL parsing

- **L** = left-to-right
- **L** = leftmost derivation

1 $S \rightarrow \text{if } E \text{ then } S \text{ else } S$

2 $S \rightarrow \text{begin } S\ L$

3 $S \rightarrow \text{print } E$

4 $L \rightarrow \text{end}$

5 $L \rightarrow ;\ S\ L$

6 $E \rightarrow \text{num} = \text{num}$

```
if 0 = 1 then begin print 2 = 3; print 4 = 5 end else print 6 = 7
```



“Prefix” languages (Scheme/Lisp) **are easily parsed with LL parsers** (zero lookahead)

LR parsing

- **L** = left-to-right
- **R** = rightmost derivation

1 $S \rightarrow S\ ;\ S$

2 $S \rightarrow \text{id}\ :=\ E$

3 $S \rightarrow \text{print}\ (\ L\ )$

4 $E \rightarrow \text{id}$

5 $E \rightarrow \text{num}$

6 $E \rightarrow E\ +\ E$

```
a := 7;
b := c + (d := 5 + 6, d)
```

When parse is here, can't determine whether it's an assign (:=) or addition (+)

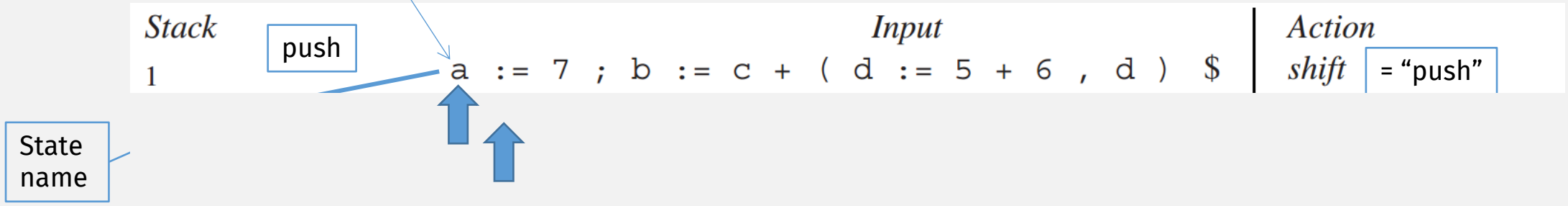
Need to save input (lookahead) to some memory, like a **stack**! this is a job for a (D)PDA!

LR parsing

- **L** = left-to-right
- **R** = rightmost derivation

$$S \rightarrow S\ ;\ S \qquad E \rightarrow \text{id}$$
$$S \rightarrow \text{id} := E \qquad E \rightarrow \text{num}$$
$$S \rightarrow \text{print}\ (\ L\ ) \qquad E \rightarrow E + E$$

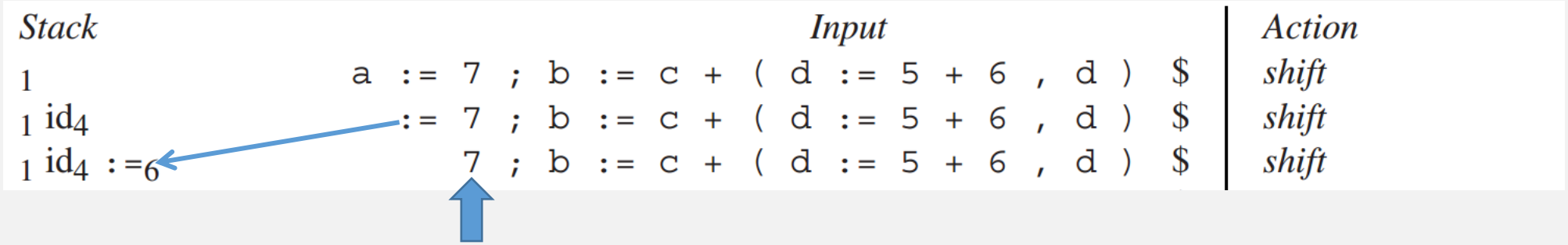
```
a := 7;
b := c + (d := 5 + 6, d)
```



LR parsing

- **L** = left-to-right
- **R** = rightmost derivation

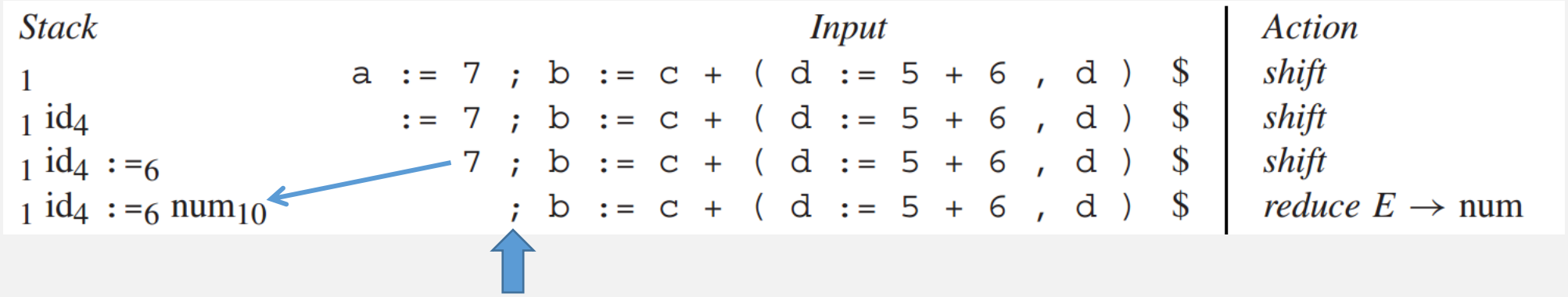
$$S \rightarrow S\ ;\ S \qquad E \rightarrow \text{id}$$
$$S \rightarrow \text{id} := E \qquad E \rightarrow \text{num}$$
$$S \rightarrow \text{print}\ (\ L\ ) \qquad E \rightarrow E + E$$



LR parsing

- **L** = left-to-right
- **R** = rightmost derivation

$$S \rightarrow S\ ;\ S \qquad E \rightarrow \text{id}$$
$$S \rightarrow \text{id} := E \qquad E \rightarrow \text{num}$$
$$S \rightarrow \text{print}\ (\ L\ ) \qquad E \rightarrow E + E$$



LR parsing

- **L** = left-to-right
- **R** = rightmost derivation

1 $S \rightarrow S\ ;\ S$

2 $S \rightarrow \text{id} := E$

3 $S \rightarrow \text{print}\ (\ L\ )$

4 $E \rightarrow \text{id}$

5 $E \rightarrow \text{num}$

6 $E \rightarrow E + E$

Stack	*Input*	*Action*
1	a := 7 ; b := c + (d := 5 + 6 , d) $	*shift*
1 id_4	:= c + (d := 5 + 6 , d) $	*shift*
1 id_4 :=6	:= c + (d := 5 + 6 , d) $	*shift*
1 id_4 :=6 num_{10}	; b := c + (d := 5 + 6 , d) $	*reduce* $E \rightarrow$ num

Can determine (rightmost) rule



LR parsing

- **L** = left-to-right
- **R** = rightmost derivation

1 $S \rightarrow S\ ;\ S$

2 $S \rightarrow \text{id} := E$

3 $S \rightarrow \text{print}\ (\ L\ )$

4 $E \rightarrow \text{id}$

5 $E \rightarrow \text{num}$

6 $E \rightarrow E + E$

Stack	*Input*	*Action*
1	a := 7 ; b := c + (d := 5 + 6 , d) $	*shift*
1 id_4	:= 7 ; b := c + (d := 5 + 6 , d) $	*shift*
1 id_4 :=$_6$	= c + (d := 5 + 6 , d) $	*shift*
1 id_4 :=$_6$ num_{10}	= c + (d := 5 + 6 , d) $	*reduce* $E \rightarrow$ num
1 id_4 :=$_6$ E_{11}	; b := c + (d := 5 + 6 , d) $	*reduce* $S \rightarrow \text{id} := E$

Can determine (rightmost) rule

LR parsing

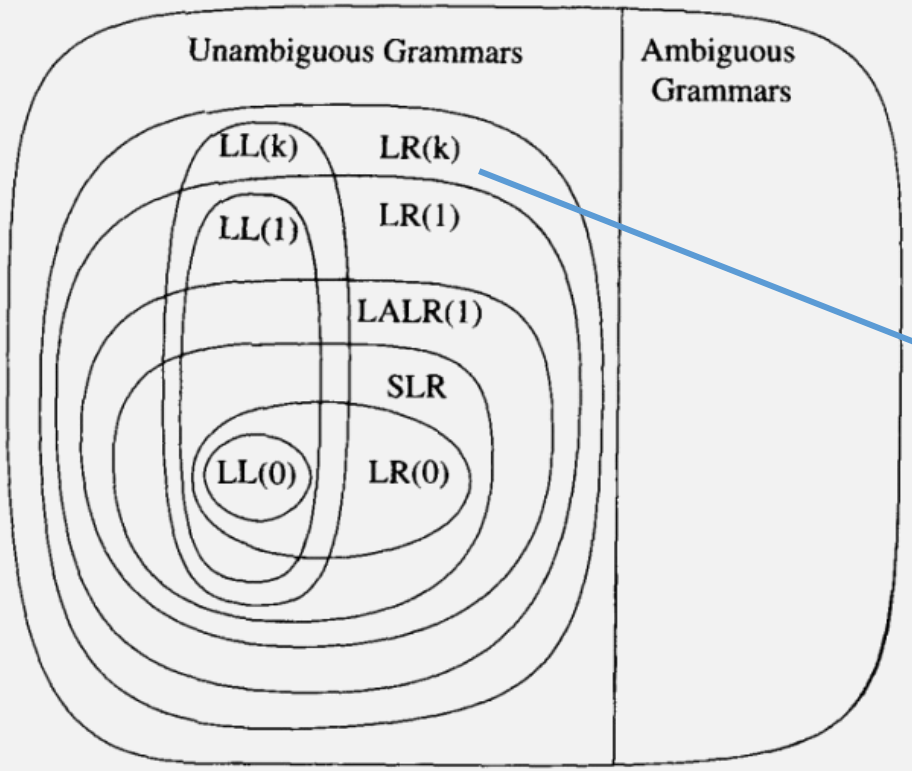
- **L** = left-to-right
- **R** = rightmost derivation

$$S \rightarrow S\ ;\ S \qquad E \rightarrow \text{id}$$
$$S \rightarrow \text{id} := E \qquad E \rightarrow \text{num}$$
$$S \rightarrow \text{print}\ (\ L\ ) \qquad E \rightarrow E + E$$

Stack	*Input*	*Action*
1	a := 7 ; b := c + (d := 5 + 6 , d) $	*shift*
1 id_4	:= 7 ; b := c + (d := 5 + 6 , d) $	*shift*
1 id_4 :=$_6$	7 ; b := c + (d := 5 + 6 , d) $	*shift*
1 id_4 :=$_6$ num_{10}	; b := c + (d := 5 + 6 , d) $	*reduce* $E \rightarrow \text{num}$
1 id_4 :=$_6$ E_{11}	; b := c + (d := 5 + 6 , d) $	*reduce* $S \rightarrow \text{id} := E$
1 S_2	; b := c + (d := 5 + 6 , d) $	*shift*

LR Parsers also called "Shift-Reduce" Parsers

To learn more, take a Compilers Class!



A program (string of chars)

Lexer
(DFAs / NFAs)

Program “words”

Parser
(DPDAs)

Abstract Syntax tree (AST)

???

This phase needs computation that goes beyond CFLs