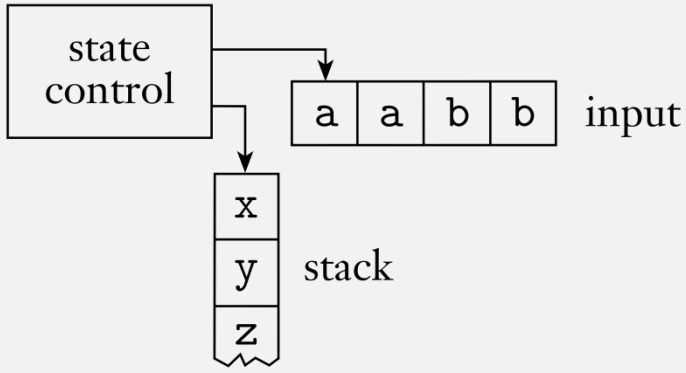
UMB CS 420

CFL ⇔ PDA

Thursday, October 20, 2022

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.



Announcements

- HW 5 out
  - Due Sun 10/23 11:59pm EST

- Reminder:
  - Use “Gradescope Regrade Request” for regrade requests
  - Do not email hw to course staff

Flashback: DFA Computation Model

Informally	*Formally (i.e., mathematically)*
• "Program" = a finite automata	• $M = (Q, \Sigma, \delta, q_0, F)$
• Input = string of chars, e.g. "1101"	• $w = w_1 w_2 \cdots w_n$
To run a "program":	
• Start in "start state"	• $r_0 = q_0$

• Repeat:
  • Read 1 char;
  • Change state according to the transition table
	• $r_i = \delta(r_{i-1}, w_i)$, for $i = 1, \ldots, n$

• Result =
  • "**Accept**" if last state is "Accept" state
  • "**Reject**" otherwise
	• M ***accepts*** w if
	sequence of states $r_0, r_1, \ldots, r_n$ in Q exists . . .
	with $r_n \in F$

A **sequence of states** represents a **DFA** computation

Flashback: A DFA Extended Transition Fn

Define **extended transition function:** $\hat{\delta} : Q \times \Sigma^* \to Q$

- Domain:
  - Beginning state $q \in Q$ (not necessarily the start state)
  - Input string $w = w_1 w_2 \cdots w_n$ where $w_i \in \Sigma$
- Range:
  - Ending state (not necessarily an accept state)

(Defined recursively)

This specifies the **sequence of states** for a **DFA** computation

- Base case: $\hat{\delta}(q, \varepsilon) = q$
- Recursive case: $\hat{\delta}(q, w) = \hat{\delta}(\delta(q, w_1), w_2 \cdots w_n)$

Last Time: PDA Configurations (IDs)

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

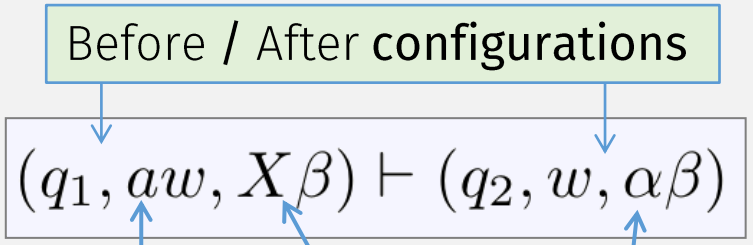
- 3 components (q, w, γ) :
  q = the **current state**
  w = the **remaining input string**
  γ = the **stack contents**

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

PDA Computation, Formally

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

Single-step



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

Extended

- Base Case

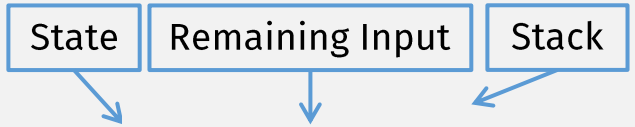
$$I \vdash^* I \text{ for any ID } I$$

- Recursive Case

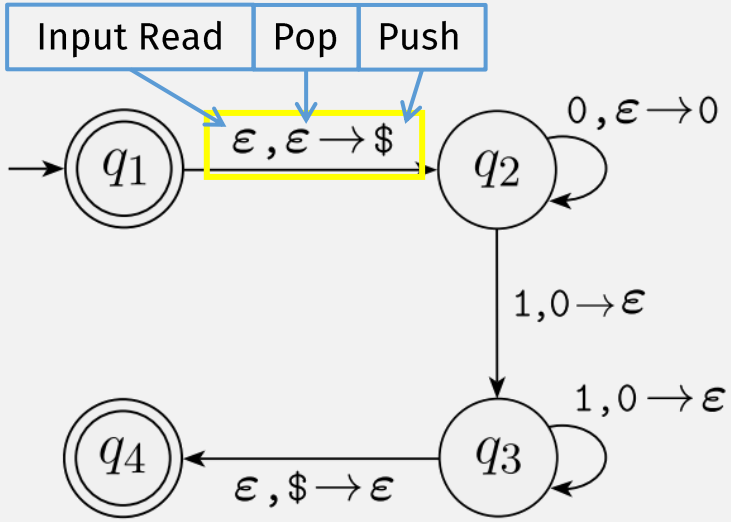
$I \vdash^* J$ if there exists some ID K such that $I \vdash K$ and $K \vdash^* J$

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

PDA Running Input String Example



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



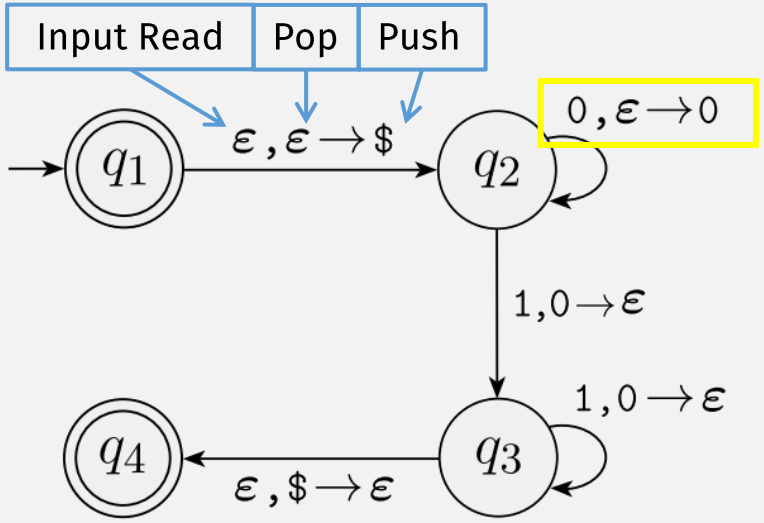
PDA Running Input String Example

State | Remaining Input | Stack

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

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

Read 0, push 0

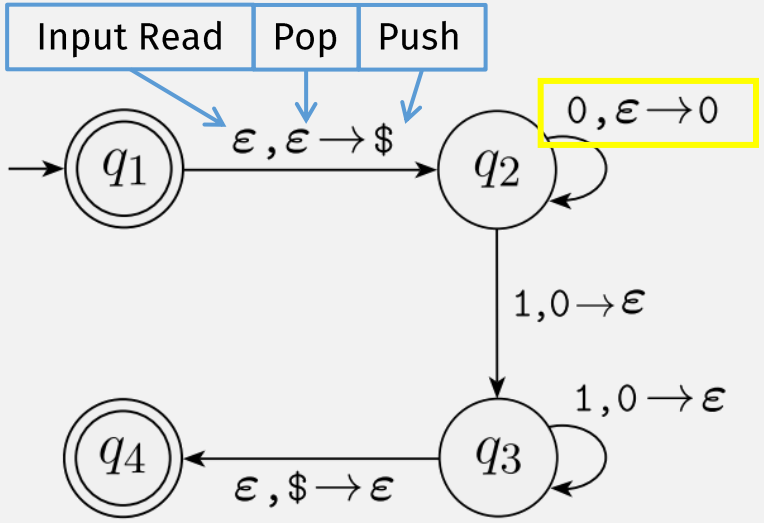


PDA Running Input String Example

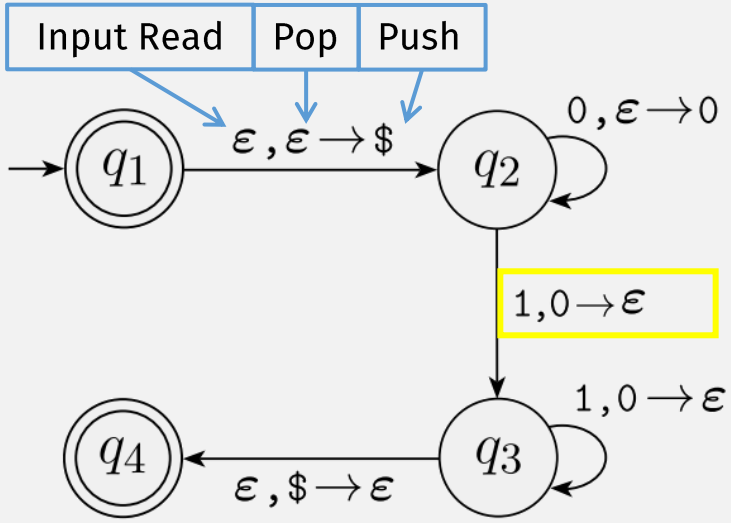
State | Remaining Input | Stack

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

Read 0, push 0



PDA Running Input String Example

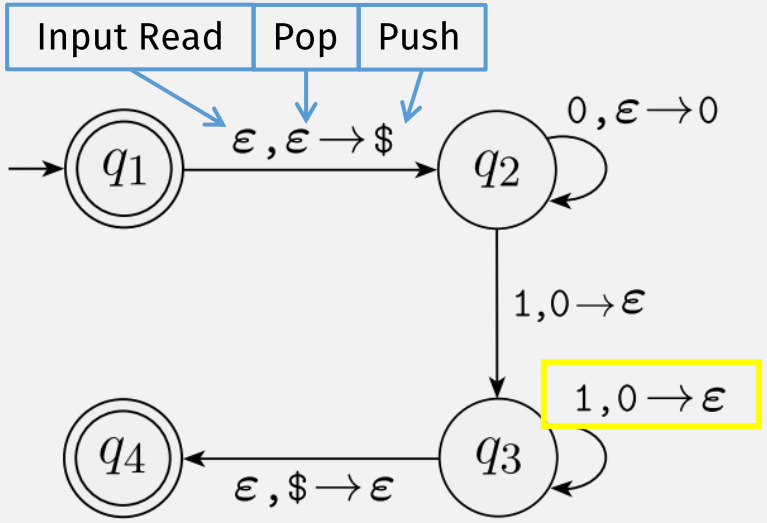


State | Remaining Input | Stack

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

Read 1, pop 0

PDA Running Input String Example

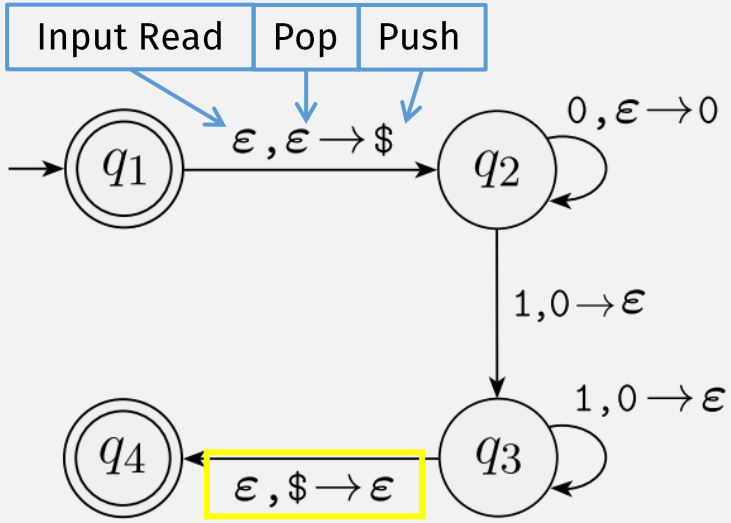


State | Remaining Input | Stack

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

Read 1, pop 0

PDA Running Input String Example



State | Remaining Input | Stack

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

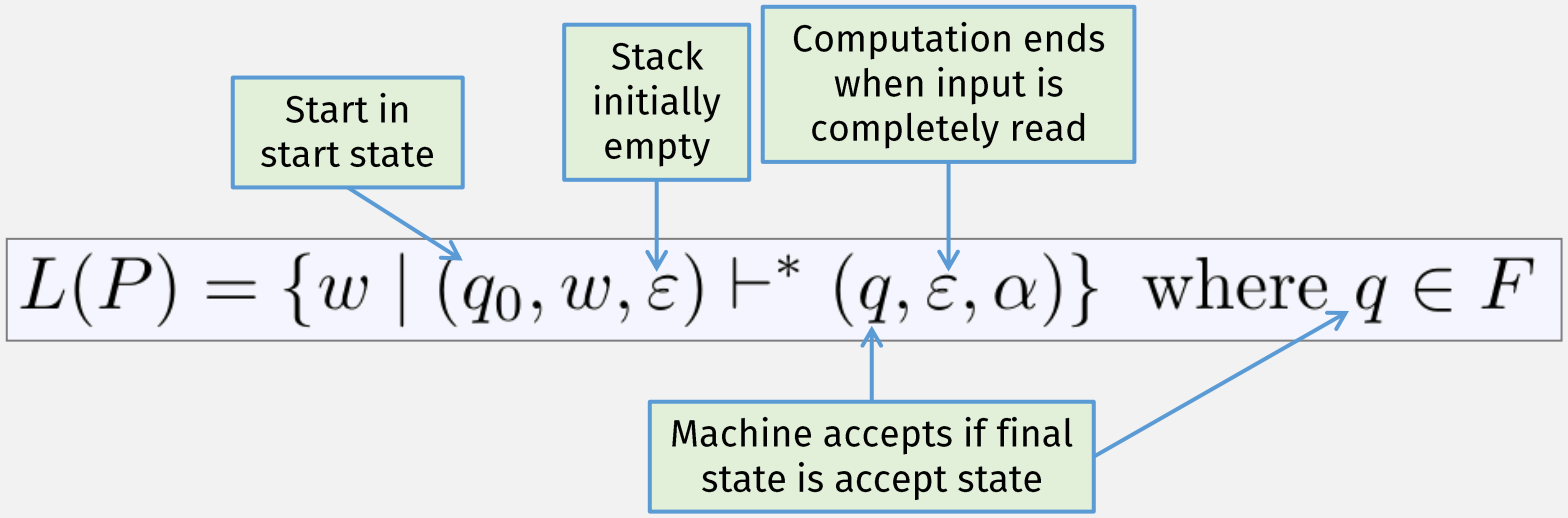
pop empty stack symbol

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 $M = L(M) = \{ w \mid M$ **accepts** $w \}$

Language of a PDA

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

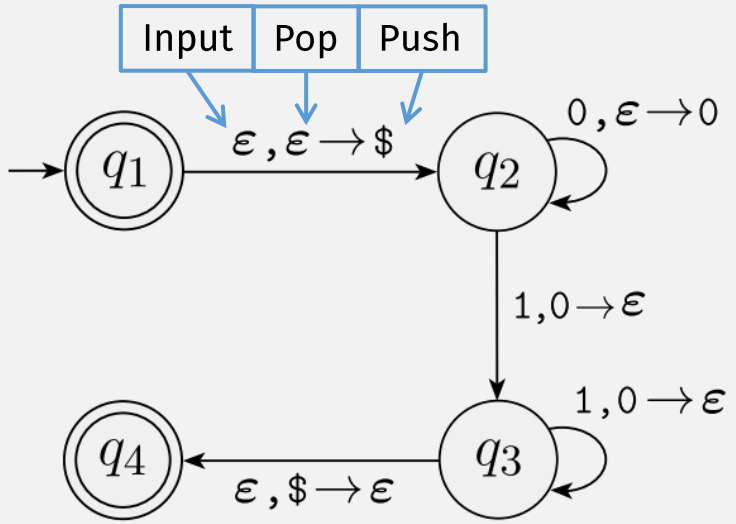


A **configuration** (q, w, γ) has three components

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

PDAs and CFLs?

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



- 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

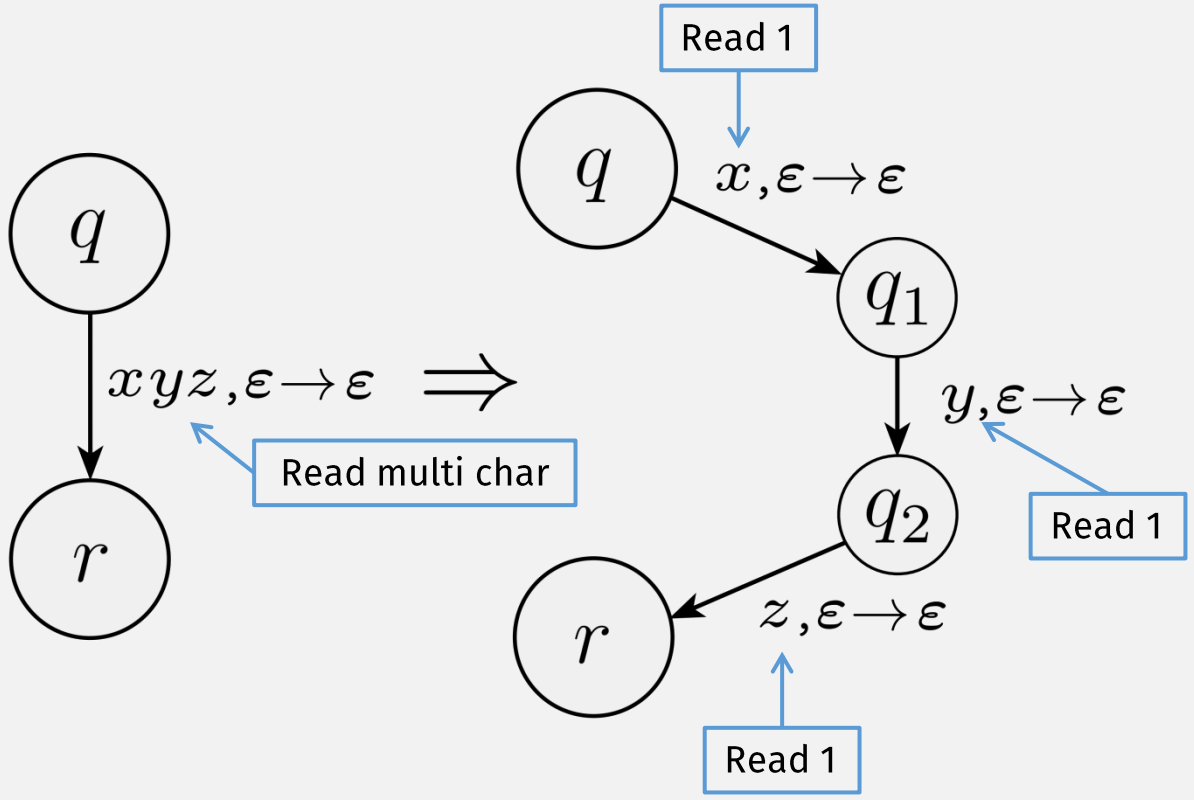
A lang is a CFL **iff** some PDA recognizes it

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

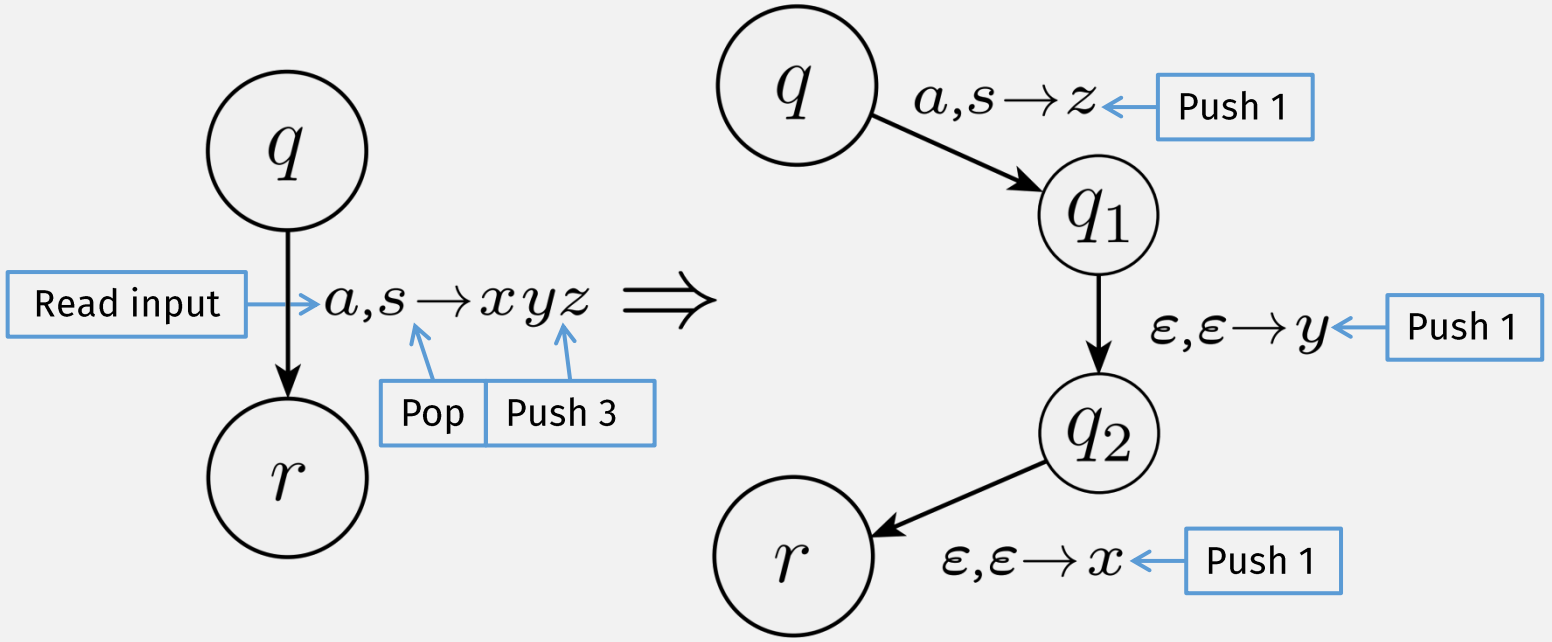
- We know: **A CFL has a CFG describing it** (definition of CFL)
- To prove this part: **show the CFG has an equivalent PDA**

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

Shorthand: Multi-Symbol Read Transition



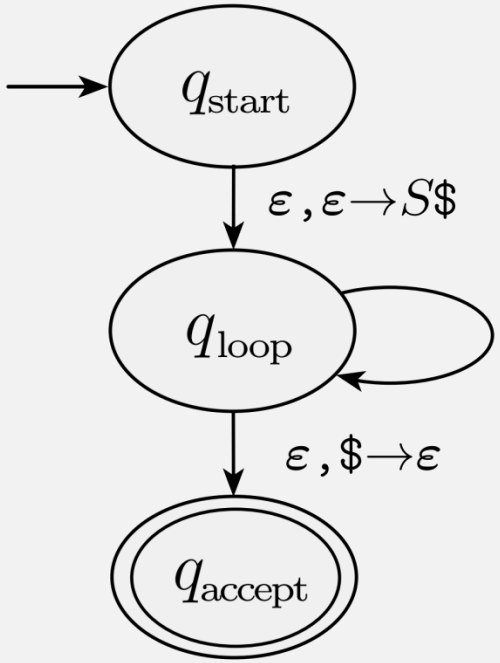
Shorthand: Multi-Stack Push Transition



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

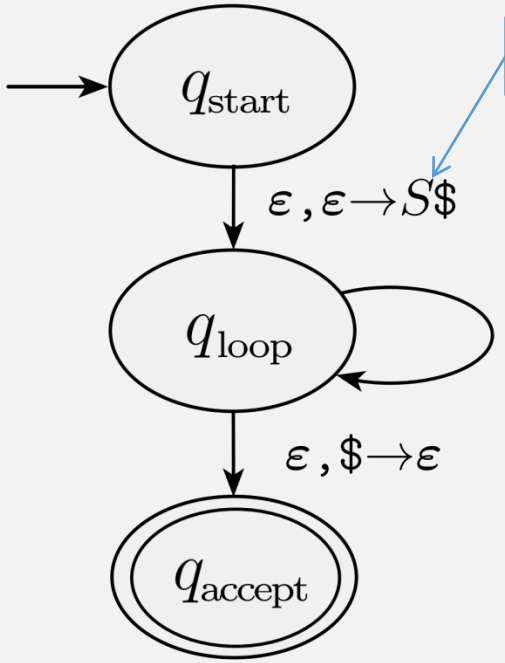


$\varepsilon, A \to w$ for rule $A \to w$

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



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 $A \to w$

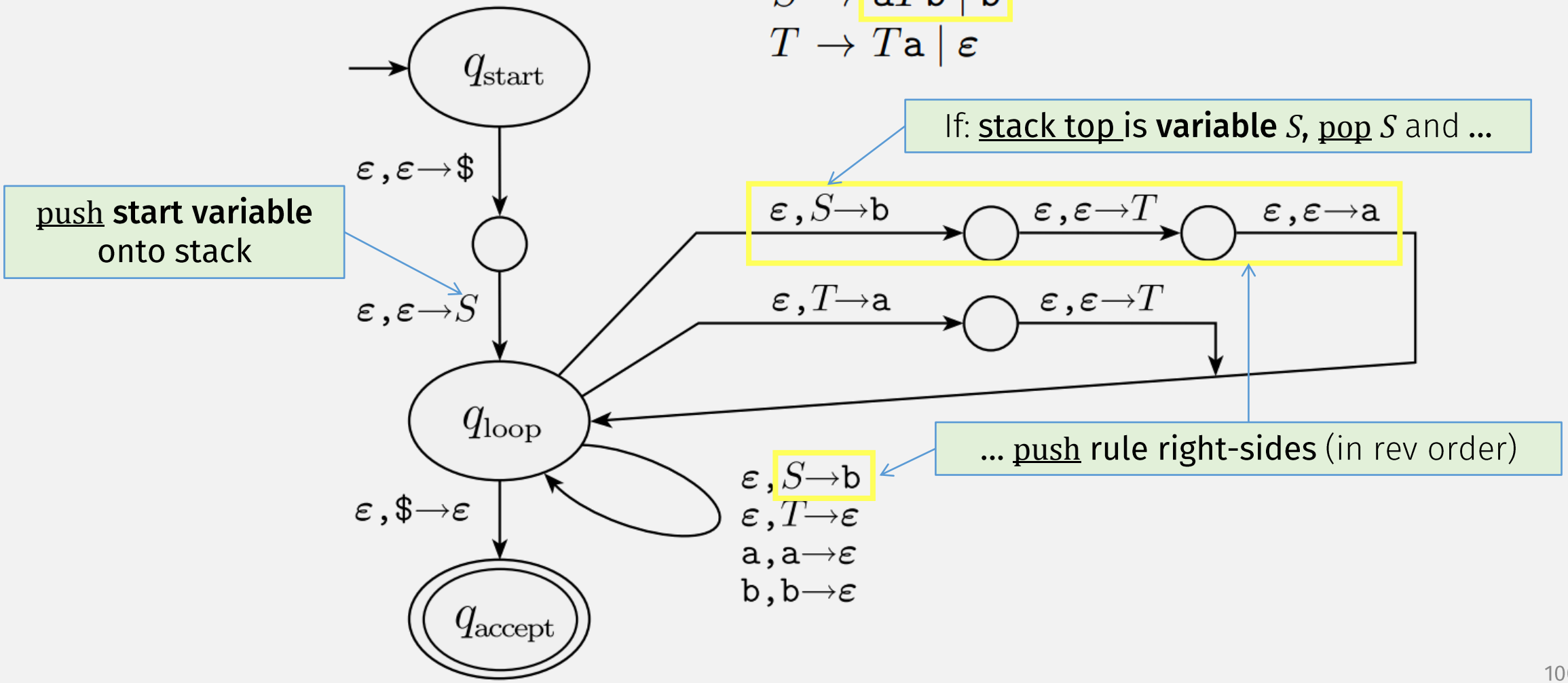
$\mathtt{a}, \mathtt{a} \to \varepsilon$ for terminal a

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

… read matching input

Example **CFG→PDA**

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

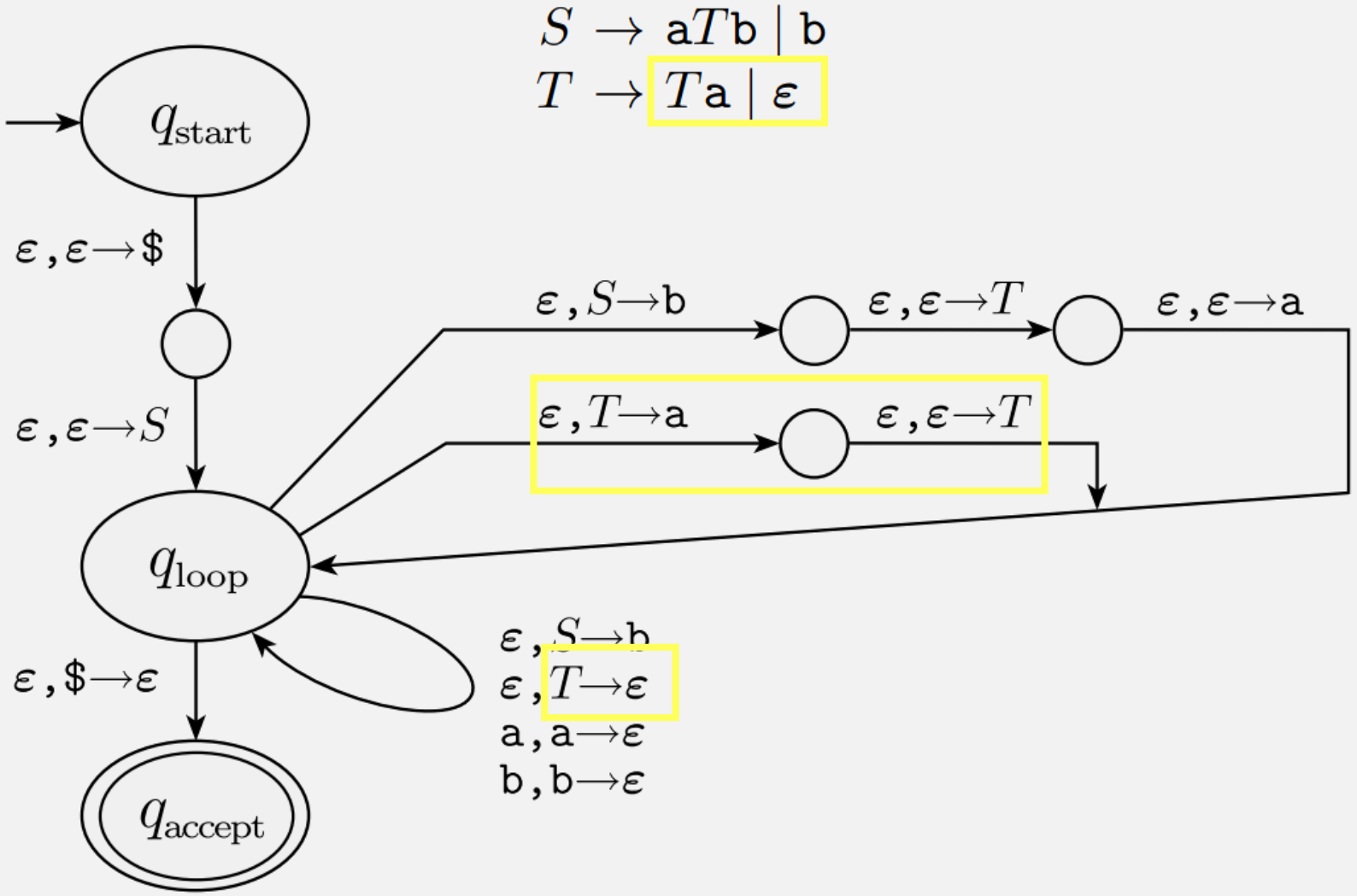


push **start variable** onto stack

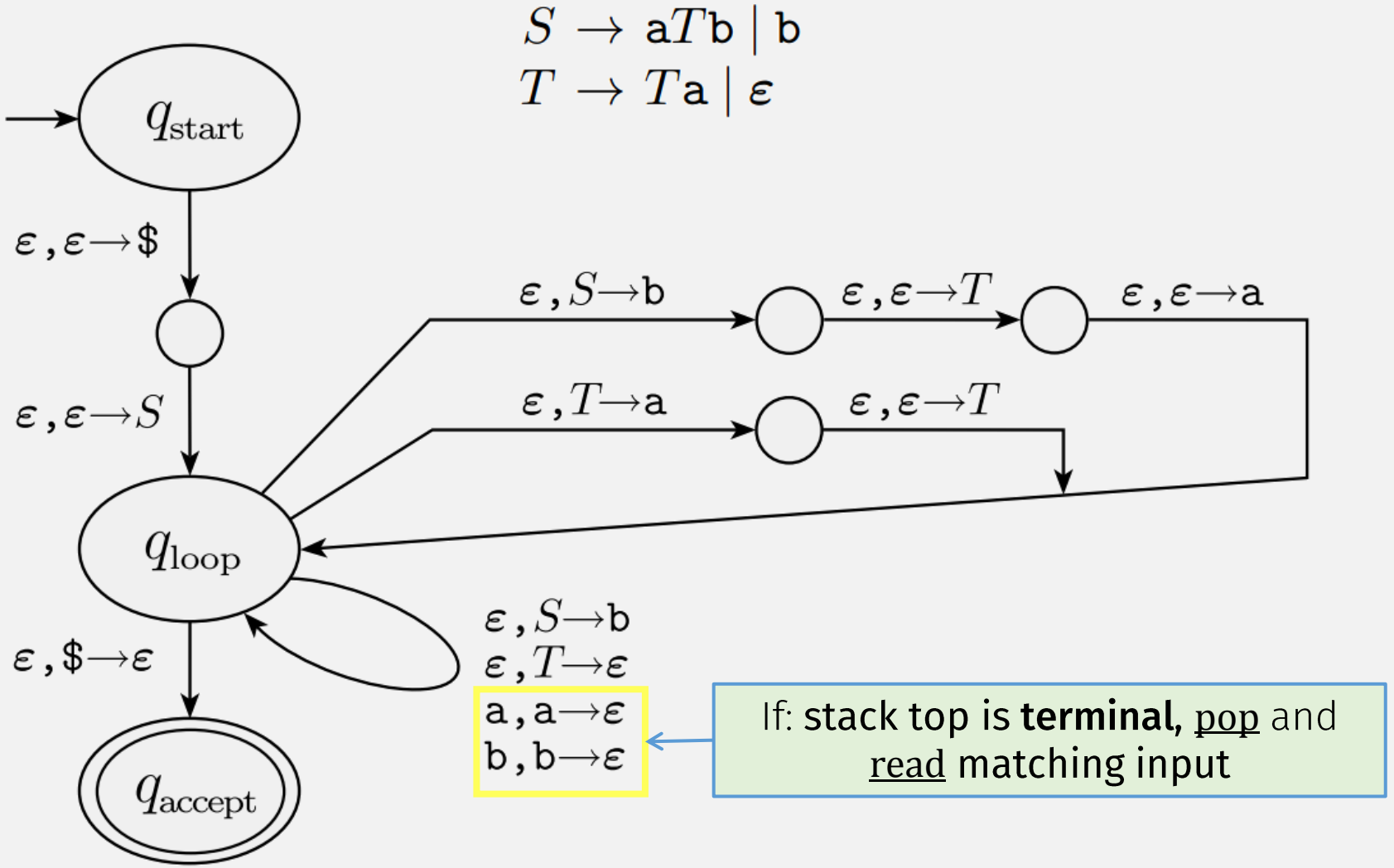
If: stack top is **variable** S, pop S and ...

... push rule right-sides (in rev order)

Example **CFG→PDA**



Example **CFG→PDA**

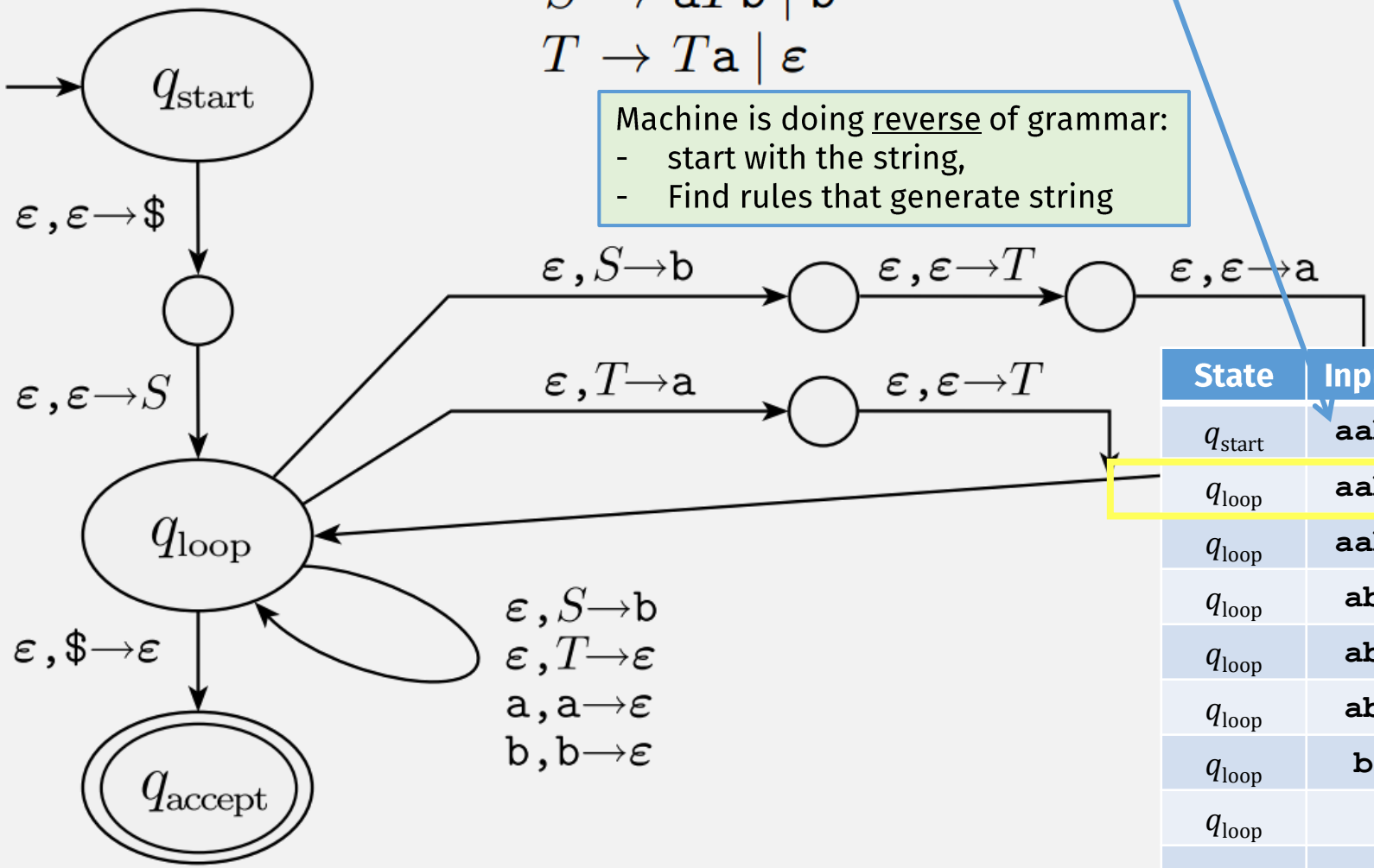


Example **CFG→PDA**

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



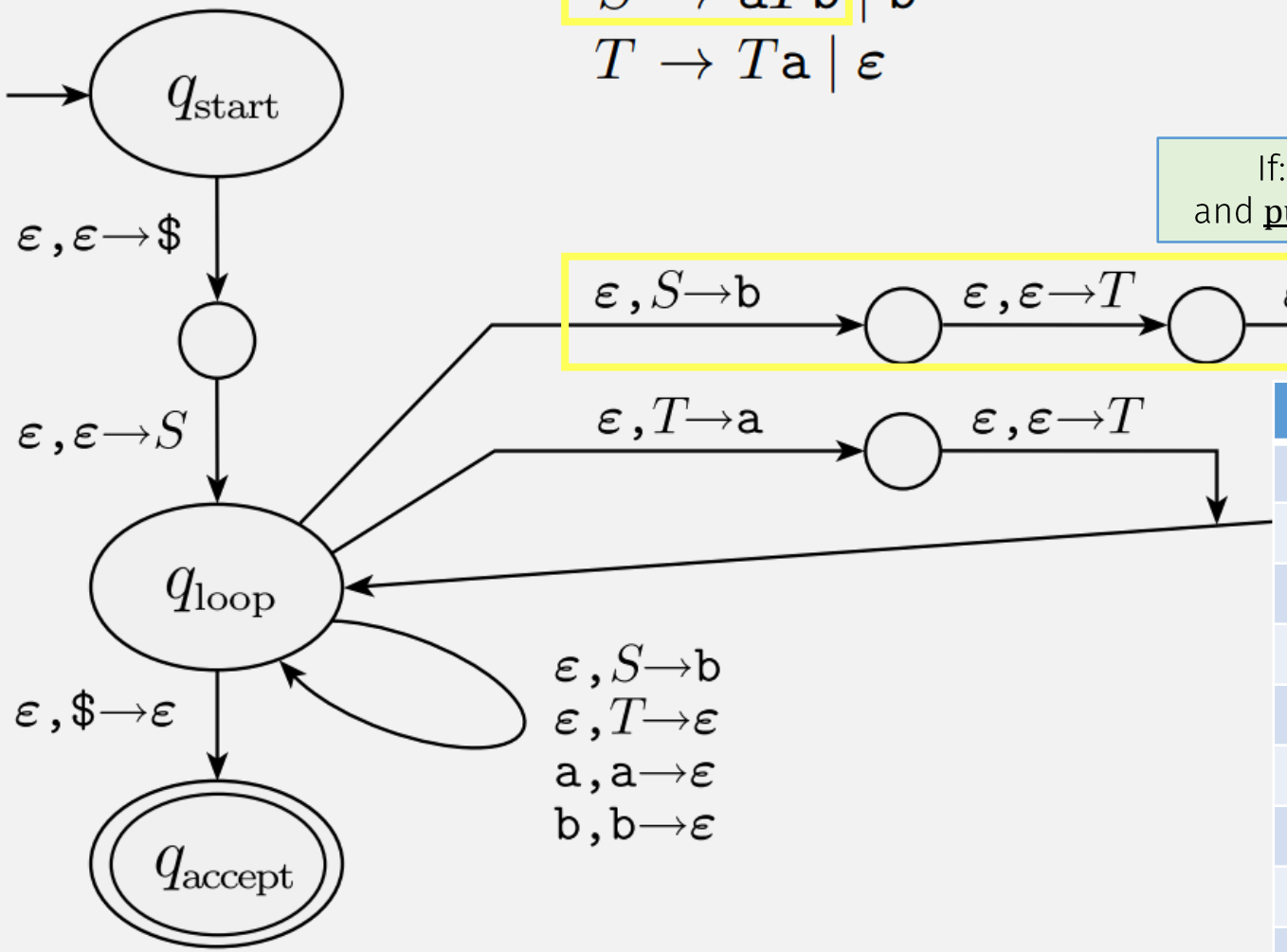
PDA Example

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	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 $T \to \varepsilon$)

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



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

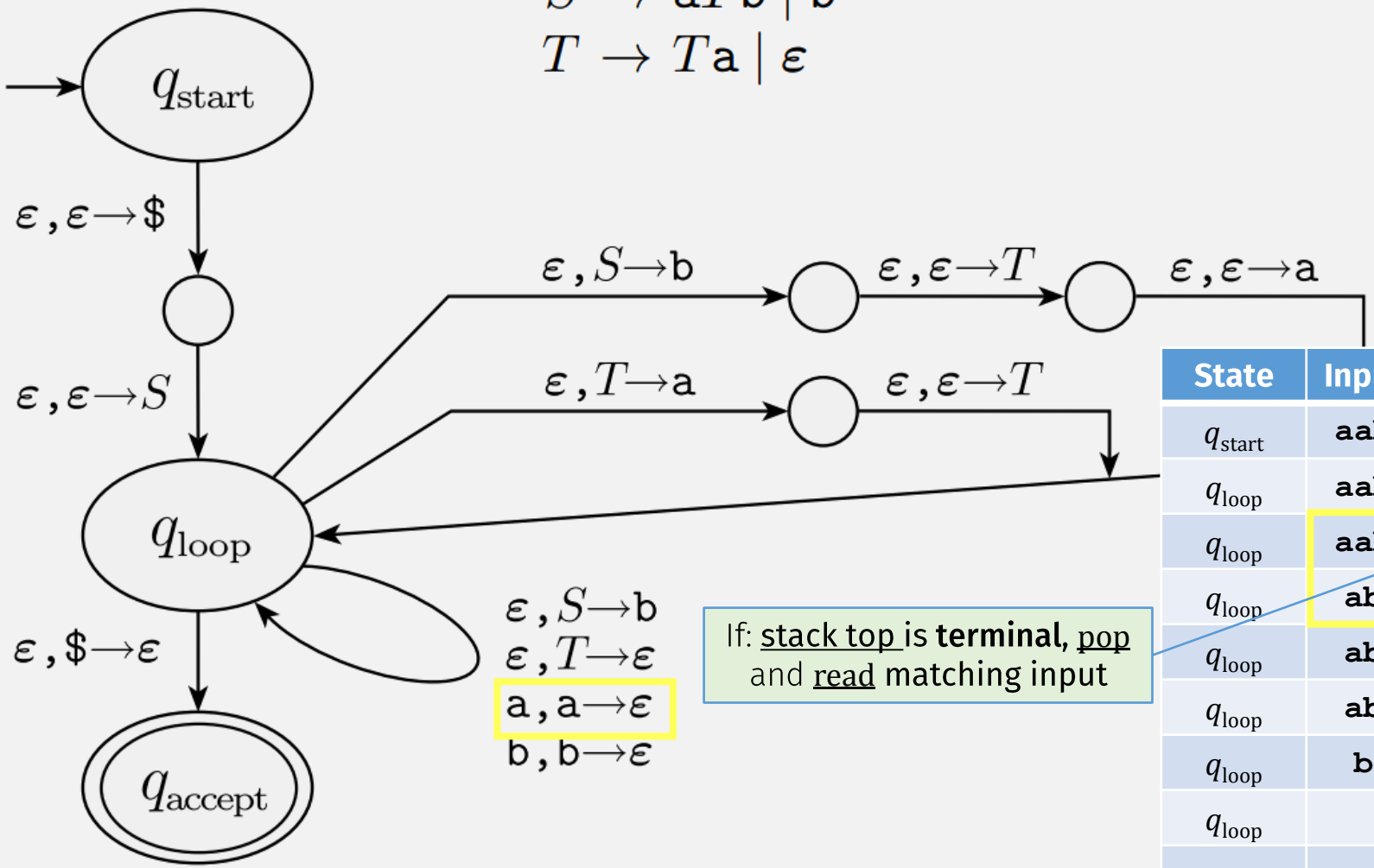
PDA Example

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

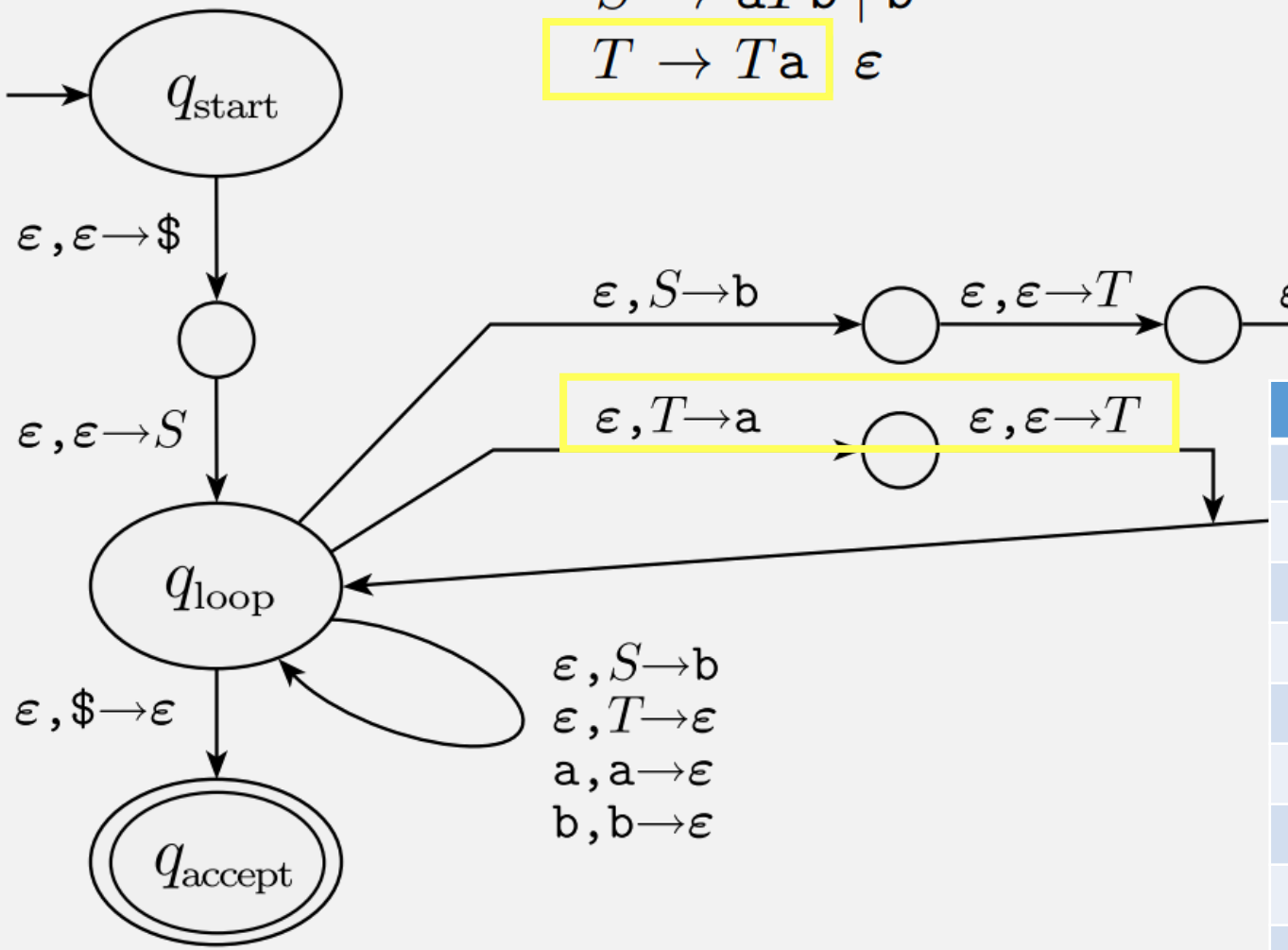
PDA Example

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	$\mathtt{ab}\$$	$T \to \varepsilon$
q_{loop}	b	$\mathtt{b}\$$	
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

- To prove this part: 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 $\{A_{pq} \mid p, q \in Q\}$

- 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

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

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

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

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

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

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**

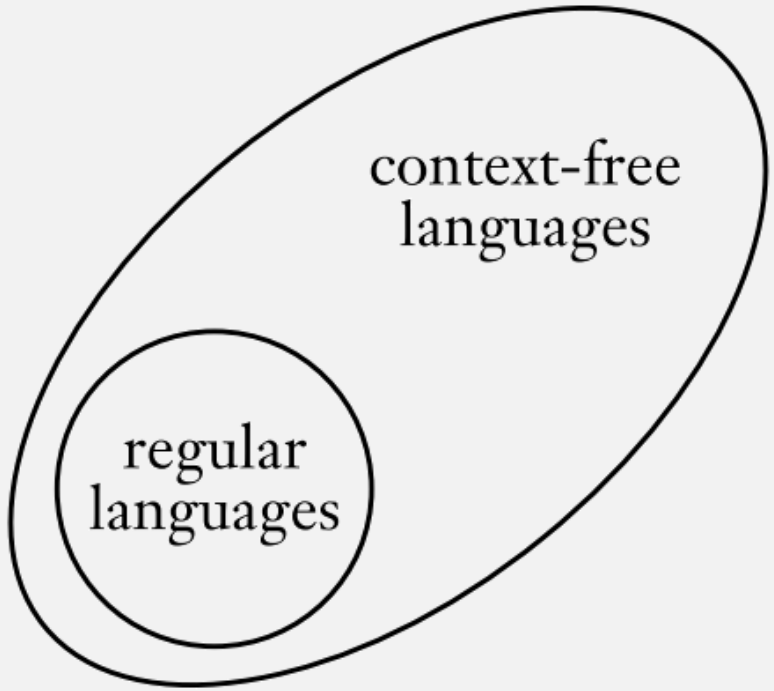
☑⇐ If **a PDA recognizes a language,** then **it's a CFL**

- Convert **PDA→CFG**

■

Regular Languages are CFLs: 3 Proofs

- DFA → CFG
  - HW?
- NFA → CFG
  - NFA → PDA (with no stack moves) → CFG
  - Just now
- Regular expression → CFG
  - HW?



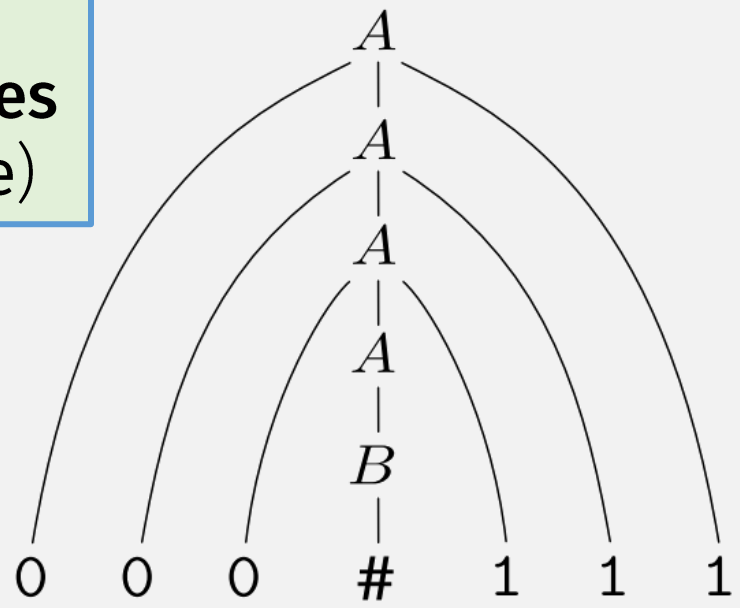
Check-in Quiz 10/20

On Gradescope

Previously: CFLs, CFGs, and Parse Trees

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

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



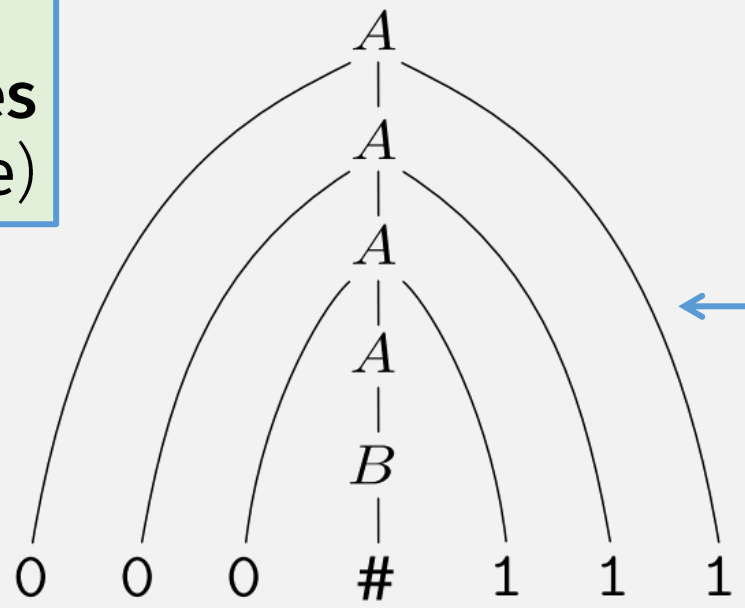
$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000B111 \Rightarrow 000\#111$$

Generating vs Parsing

Generating strings:
- Start with **start variable,**
- Repeatedly *apply* **CFG rules**

to get string (and **parse tree**)

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



In practice, **the opposite** is more interesting: start with a string, then **parse** it into parse tree

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

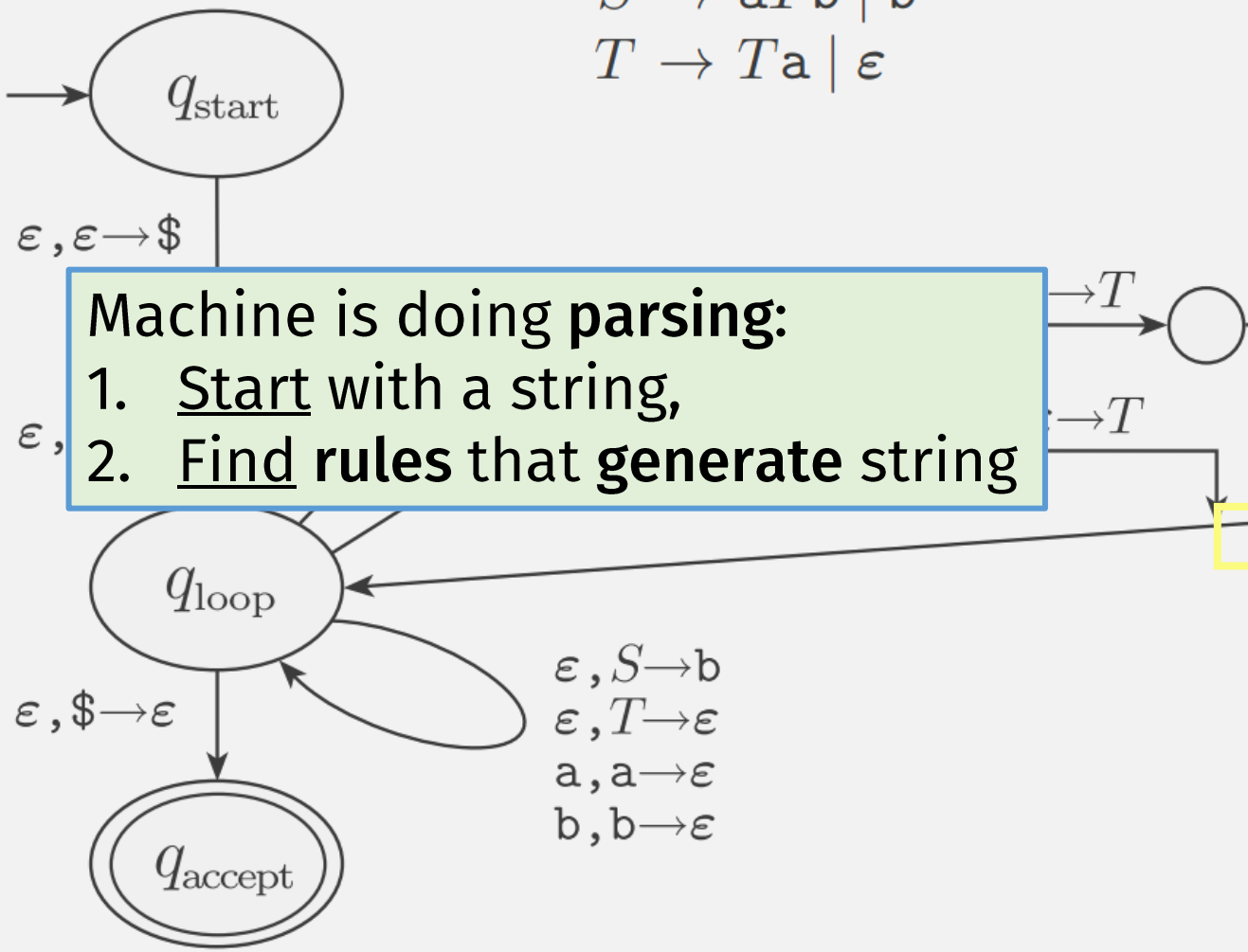
Generating vs Parsing

- In practice, **parsing** a string more important than **generating** one
  - E.g., a **compiler** (first step) parses source code 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 $T \to \varepsilon$)

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



Machine is doing **parsing**:
1. Start with a string,
2. 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}			

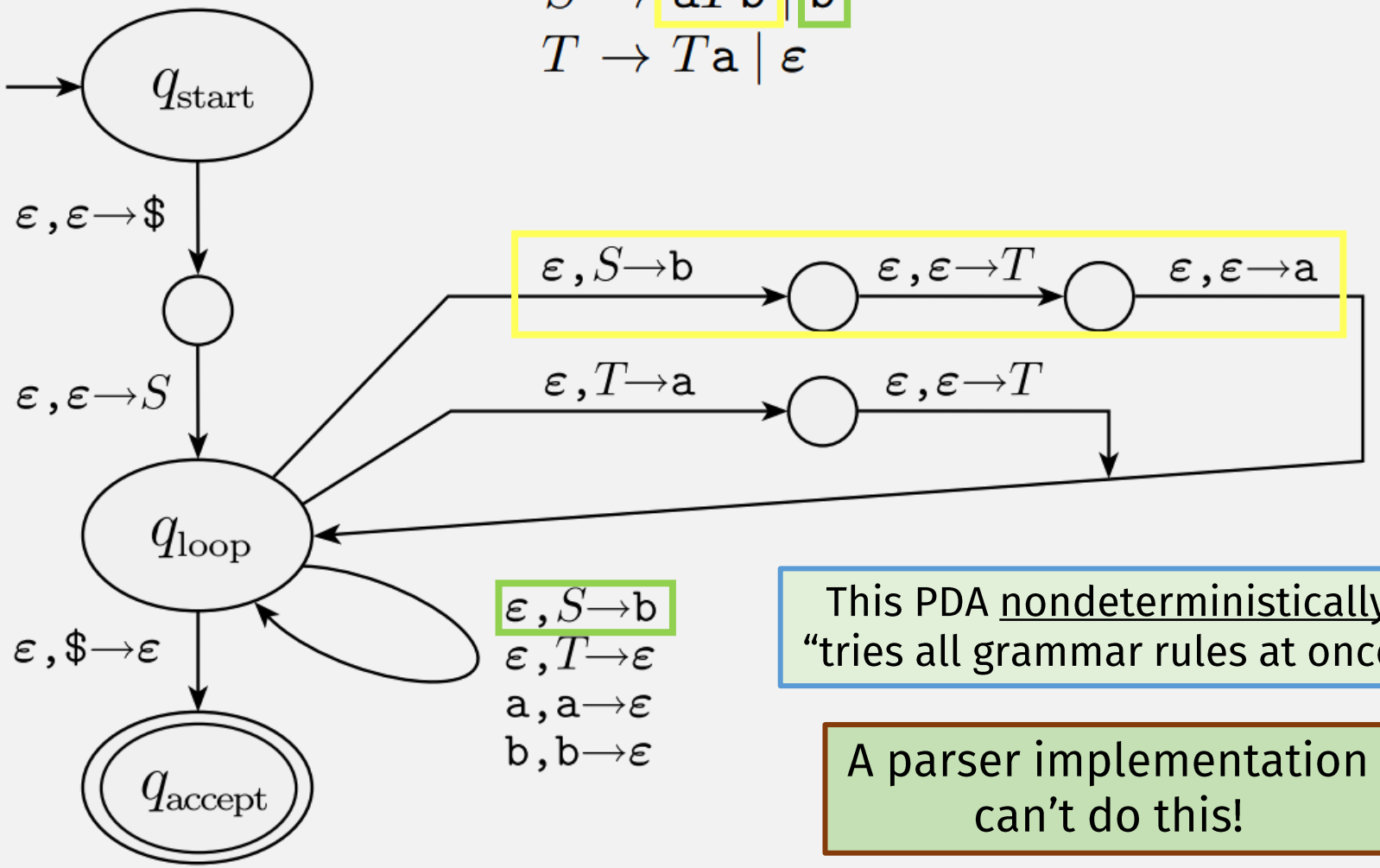
Generating vs Parsing

- 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)

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 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. $\delta: Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow (Q \times \Gamma_\varepsilon) \cup \{\emptyset\}$ is the transition function
5. $q_0 \in Q$ is the start state, and
6. $F \subseteq Q$ is the set of accept states.

A ***pushdown automaton*** is a 6-tuple

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)

This must take into account ε reads or stack ops! E.g., if $\delta(q, a, X)$ is valid, then $\delta(q, \varepsilon, X)$ must not be

DPDAs are Not Equivalent to PDAs!

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

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

Should use S rule

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

aaabbb ⟼ aaSbb

When parsing reaches this input position, which rule to use, S or T?

Should use T rule

aaabbbbbb ⟼ aaTbbbb

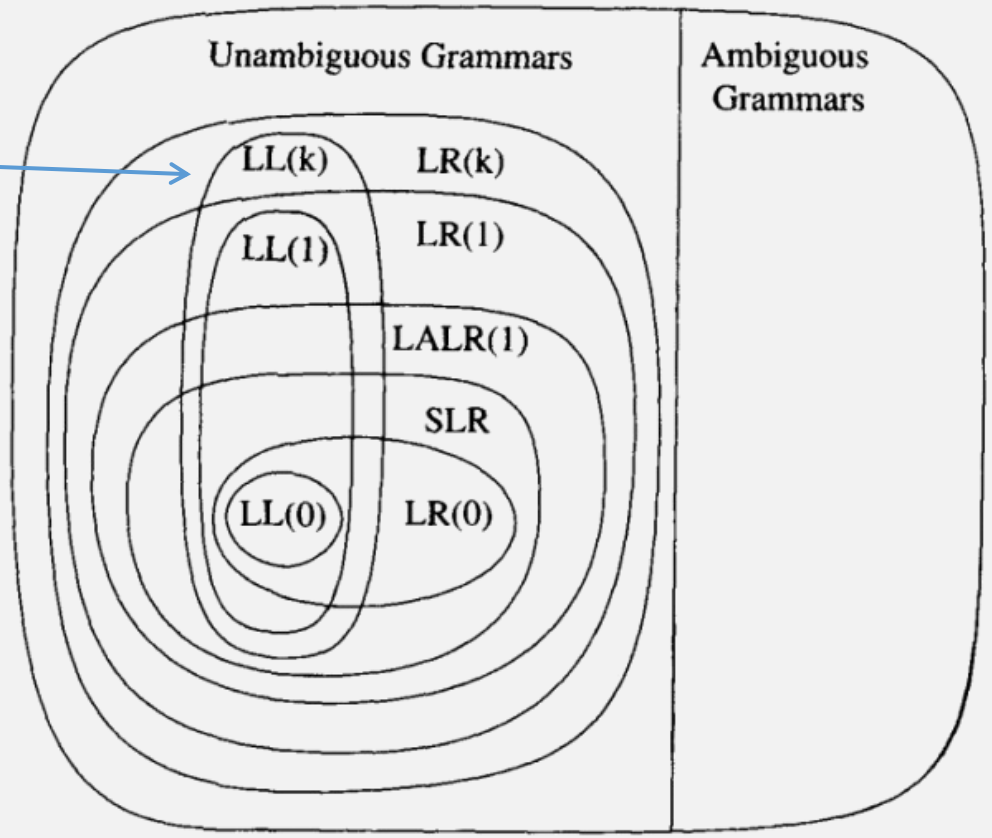
Choosing "correct" rule depends on rest of the input!

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

Subclasses of CFLs

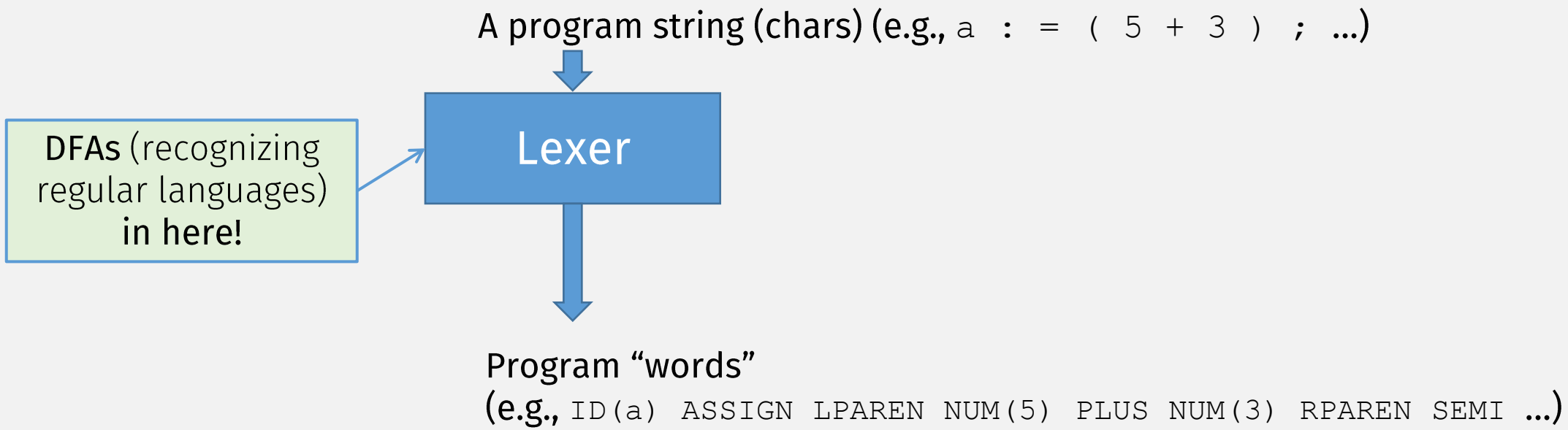
DCFLs

Programming language parsers / compilers are ideally in here



All CFLS

Compiler Stages



A Lexer Implementation

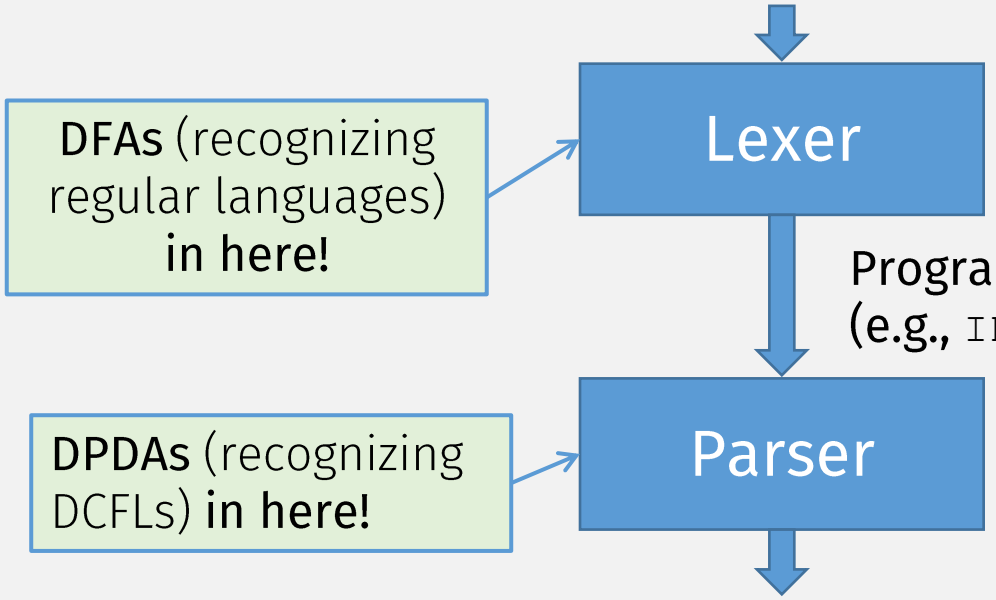
```
%{
/* C Declarations: */
#include "tokens.h"    /* definitions of IF, ID, NUM, ... */
#include "errormsg.h"
union {int ival; string sval; double fval;} yylval;
int charPos=1;
#define ADJ  (EM_tokPos=charPos, charPos+=yyleng)
%}
/* Lex Definitions: */
digits  [0-9]+
%%
/* Regular Expressions and Actions: */
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");}
```

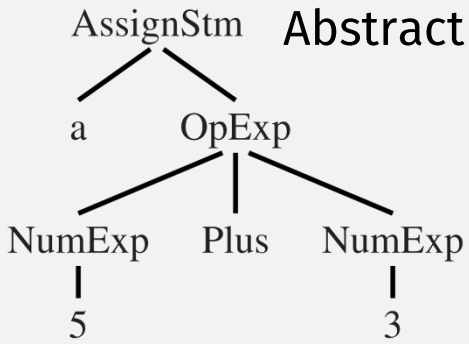
DFAs (represented as **regular expressions**)!

A "`lex`" tool translates this to a (C program) implementation of a lexer

Compiler Stages

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





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!

A "yacc" tool translates this to a (C program) implementation of a parser

Parsing

$$
\begin{aligned}
R &\to S \mid T \\
S &\to \mathtt{a}S\mathtt{b} \mid \mathtt{ab} \\
T &\to \mathtt{a}T\mathtt{bb} \mid \mathtt{abb}
\end{aligned}
$$

$\mathtt{aa\underline{ab}bb} \rightarrowtail \mathtt{a\underline{aSb}b}$

A parser must be able to <u>choose the one correct rule</u>, when reading input left-to-right

$\mathtt{aa\underline{abb}bbbb} \rightarrowtail \mathtt{a\underline{aTbb}bb}$

LL parsing

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

Game: "You're the Parser": Guess which rule applies?

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 2 = 3 begin print 1; print 2; end else print 0
```

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 2 = 3 begin print 1; print 2; end else print 0
```

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 2 = 3 begin print 1; print 2; end else print 0
```

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 2 = 3 begin print 1; print 2; end else print 0
```

"Prefix" languages (like Scheme/Lisp) are easily parsed with LL parsers

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 to some temporary memory, like a **stack**: this is a job for a (D)PDA!!

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* $E \rightarrow$ num
1 id_4 :=$_6$ E_{11}	; b := c + (d := 5 + 6 , d) $	*reduce* $S \rightarrow$ id := E
1 S_2	; b := c + (d := 5 + 6 , d) $	*shift*

push

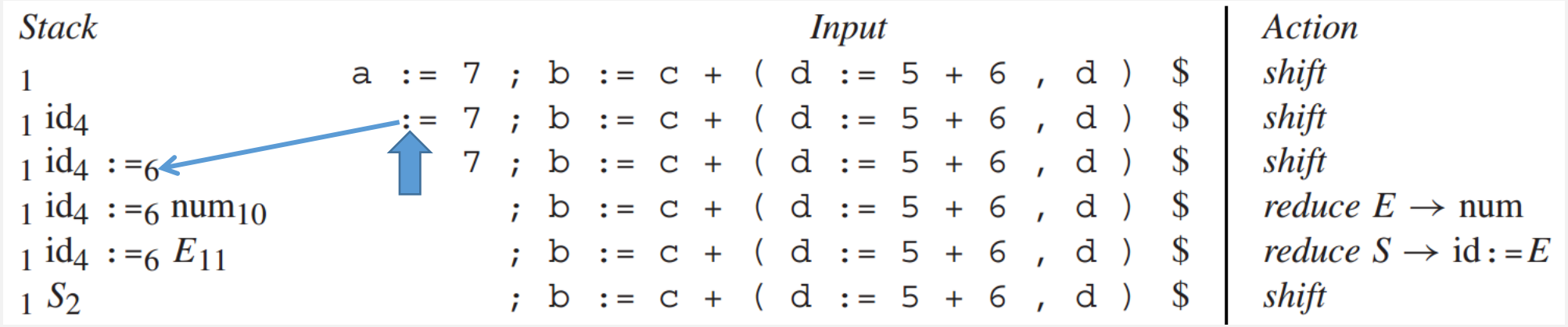
"push"

State name

LR parsing

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

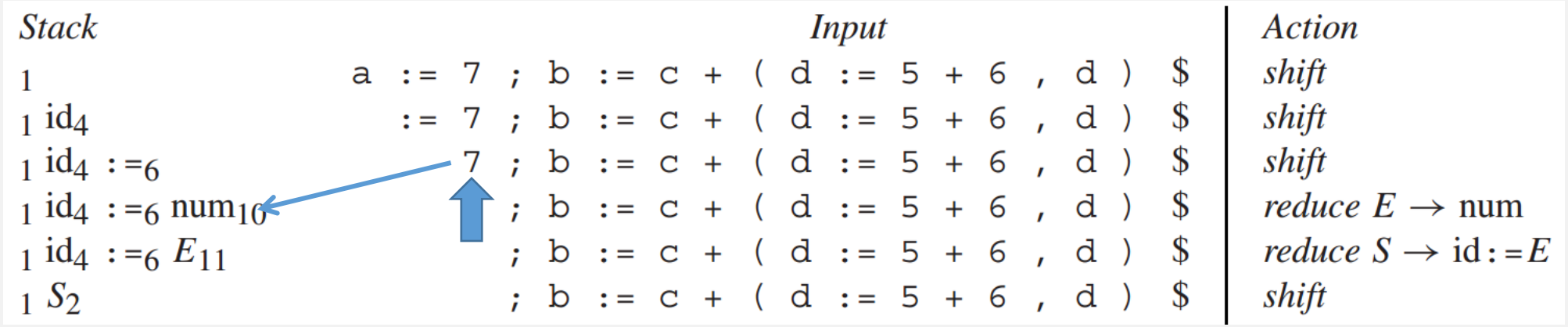
$$S \rightarrow S\ ;\ S \qquad E \rightarrow \text{id}$$
$$S \rightarrow \text{id} := E \qquad E \rightarrow \text{num}$$
$$S \rightarrow \text{print}\ (\ L\ ) \qquad E \rightarrow E + E$$



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* $E \rightarrow \text{num}$
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*

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* $E \rightarrow \text{num}$
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*

Can determine (rightmost) rule

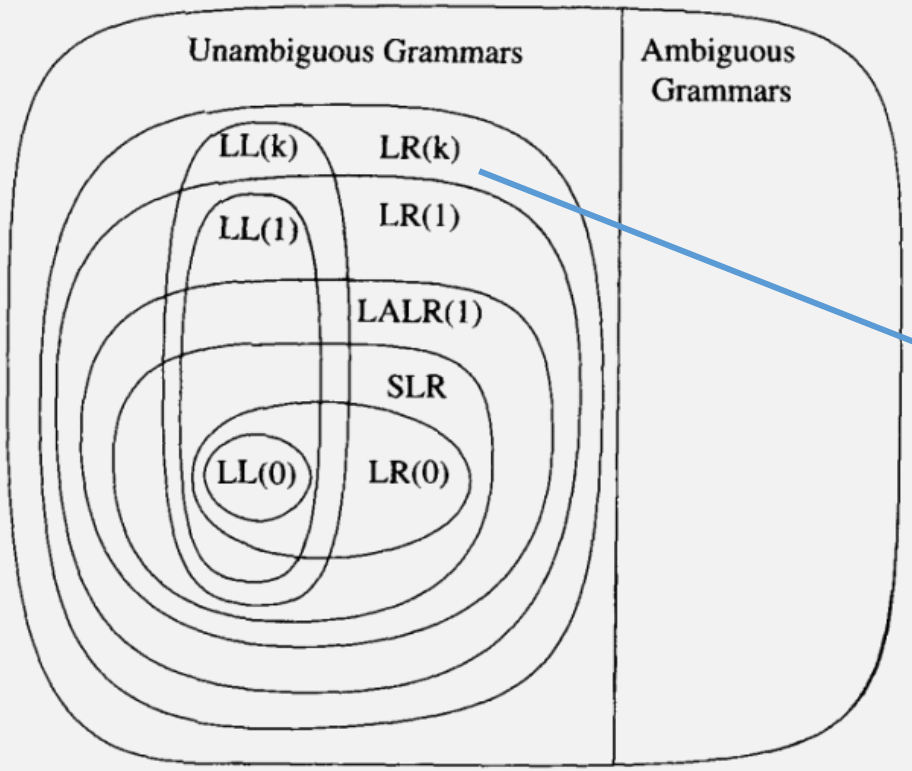
LR parsing

- **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* $E \rightarrow$ num
1 id_4 :=6 E_{11}	; b := c + (d := 5 + 6 , d) $	*reduce* $S \rightarrow$ id:=E
1 S_2	; b := c + (d := 5 + 6 , d) $	*shift*

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