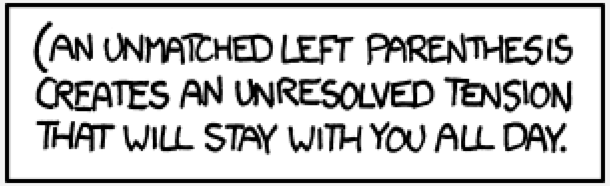
UMB CS 622

PDA ⇔ CFL

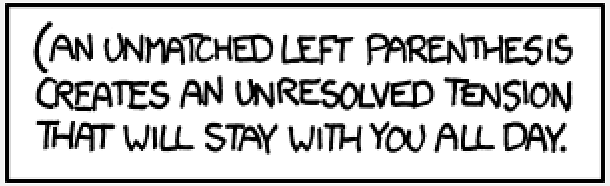
Monday, March 25, 2024



Announcements

- HW 5 in
  - ~~Due Mon 3/25 12pm noon~~

- HW 6 out
  - Due Mon 4/1 12pm noon



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)
<u>describes</u> a Regular Lang	<u>describes</u> a CFL
Finite State Automaton (FSM)	???
<u>recognizes</u> a Regular Lang	<u>recognizes</u> a CFL

Regular Language vs CFL Comparison

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

Regular Language vs CFL Comparison

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

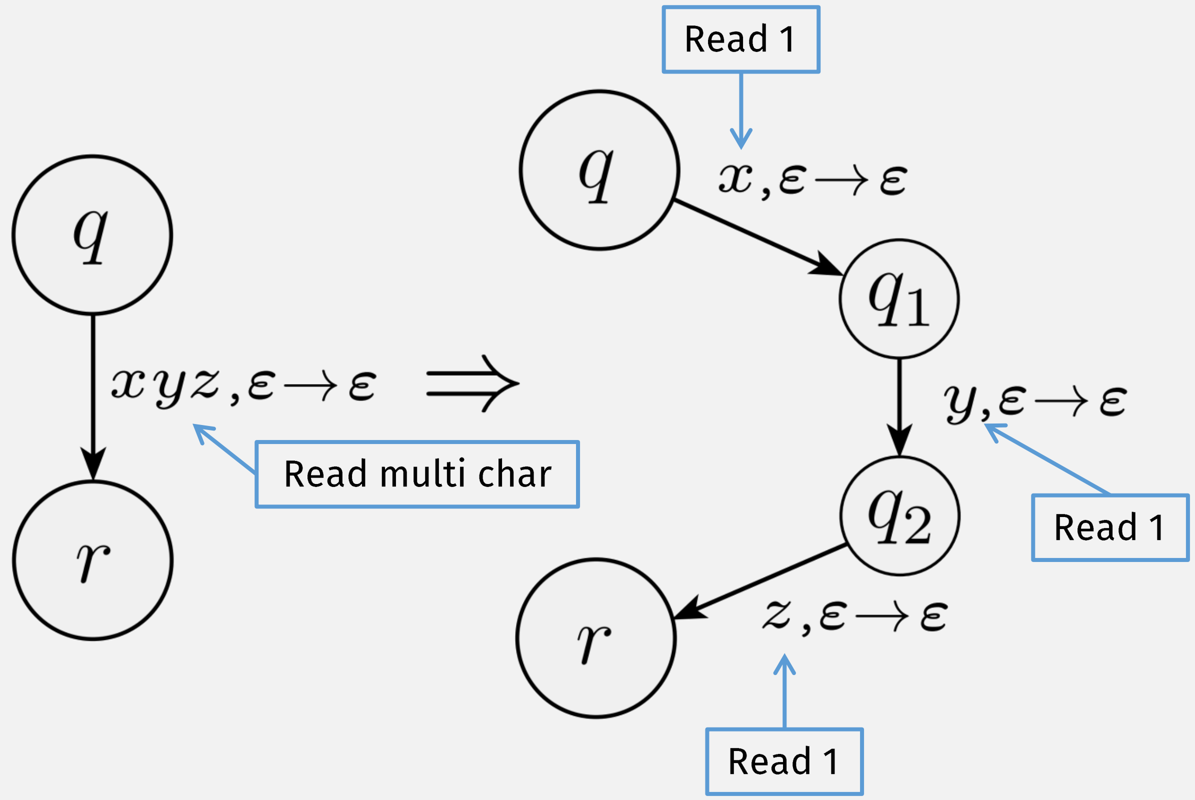
A lang is a CFL **iff** some PDA recognizes it

⇒ If **a language is a** $\boxed{\text{CFL}}$, then **a PDA recognizes it**

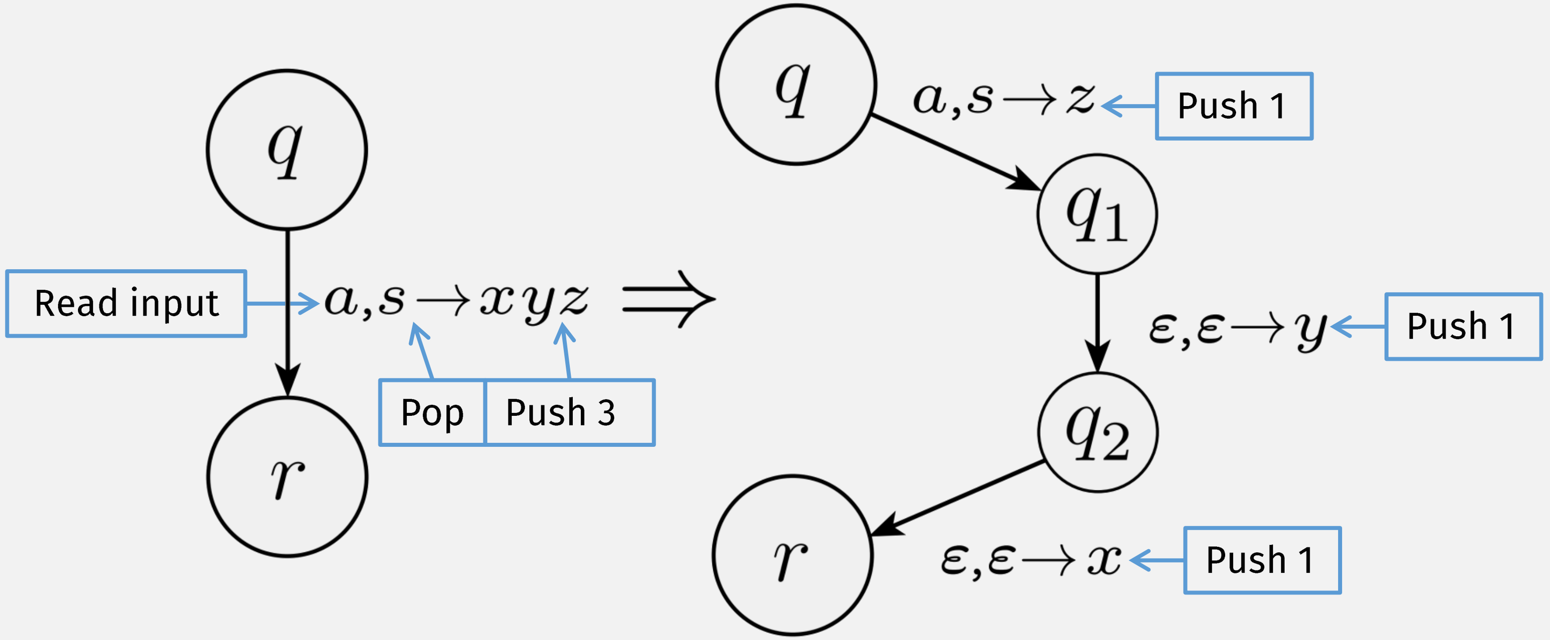
- We know: A **CFL** has a **CFG describing it** (definition of CFL)
- To prove this part, show: the **CFG** has an **equivalent** **PDA**

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

Shorthand: Multi-Symbol Read Transition



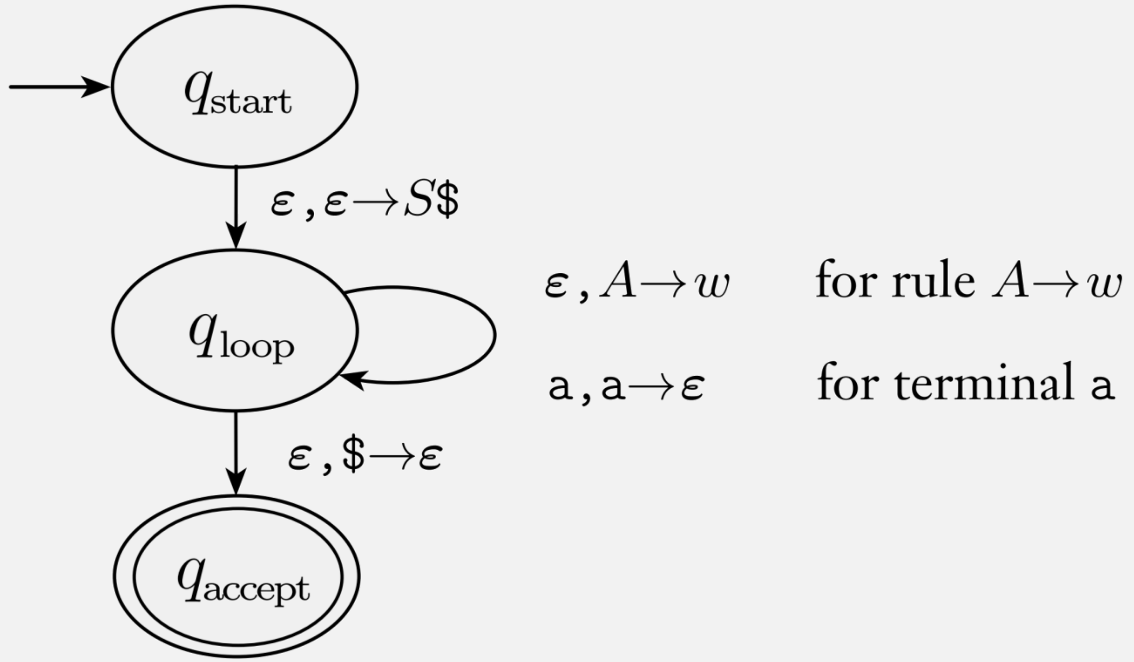
Shorthand: Multi-Stack Push Transition



Note the reverse order of pushes

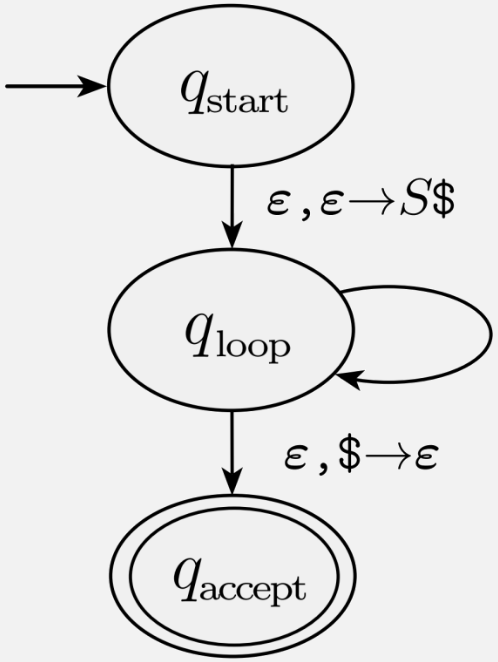
CFG→PDA (sketch)

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



CFG→PDA (sketch)

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



Convert: **every <u>CFG rule</u> to PDA "loop" transition(s)** that:
- Pops LHS variable
- Pushes RHS

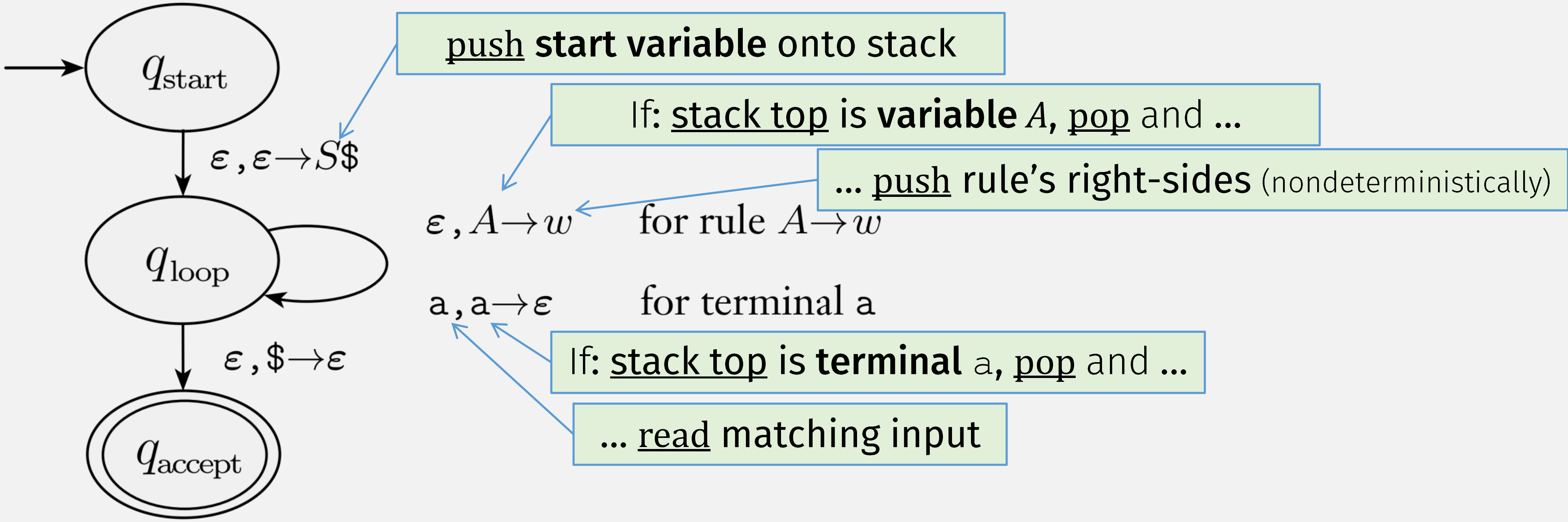
$\varepsilon, A \to w$ for rule $A \to w$

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

Convert: **<u>every terminal</u>** to "loop" **transition** that:
- Reads input char
- Pops <u>matching</u> char on stack

CFG→PDA (sketch)

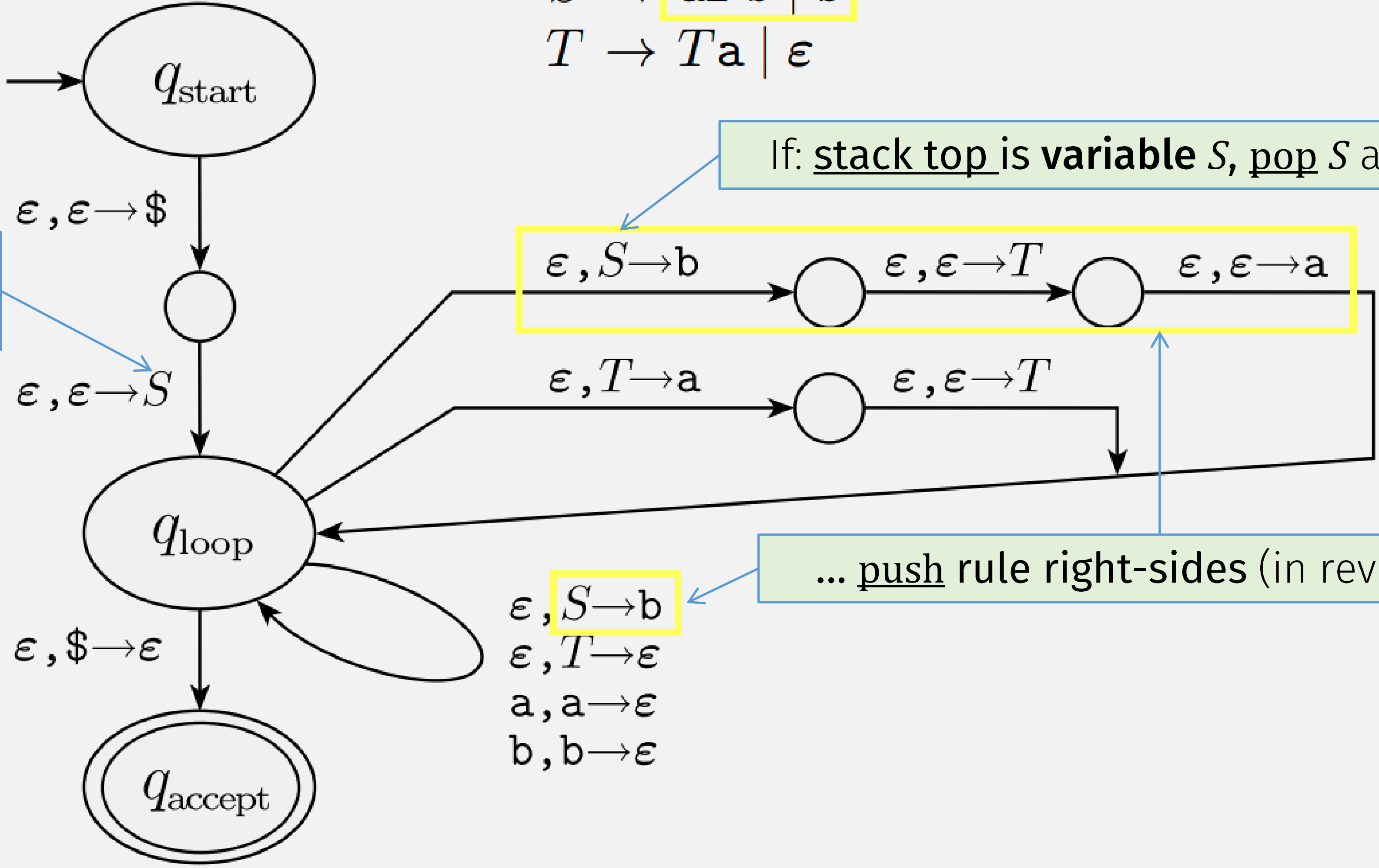
- **Construct PDA from CFG** such that:
  - **PDA accepts input only if CFG generates it**
- **PDA:**
  - **simulates generating a string with CFG rules**
  - **by** (nondeterministically) **trying all rules** to find the right ones



Example **CFG→PDA**

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

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



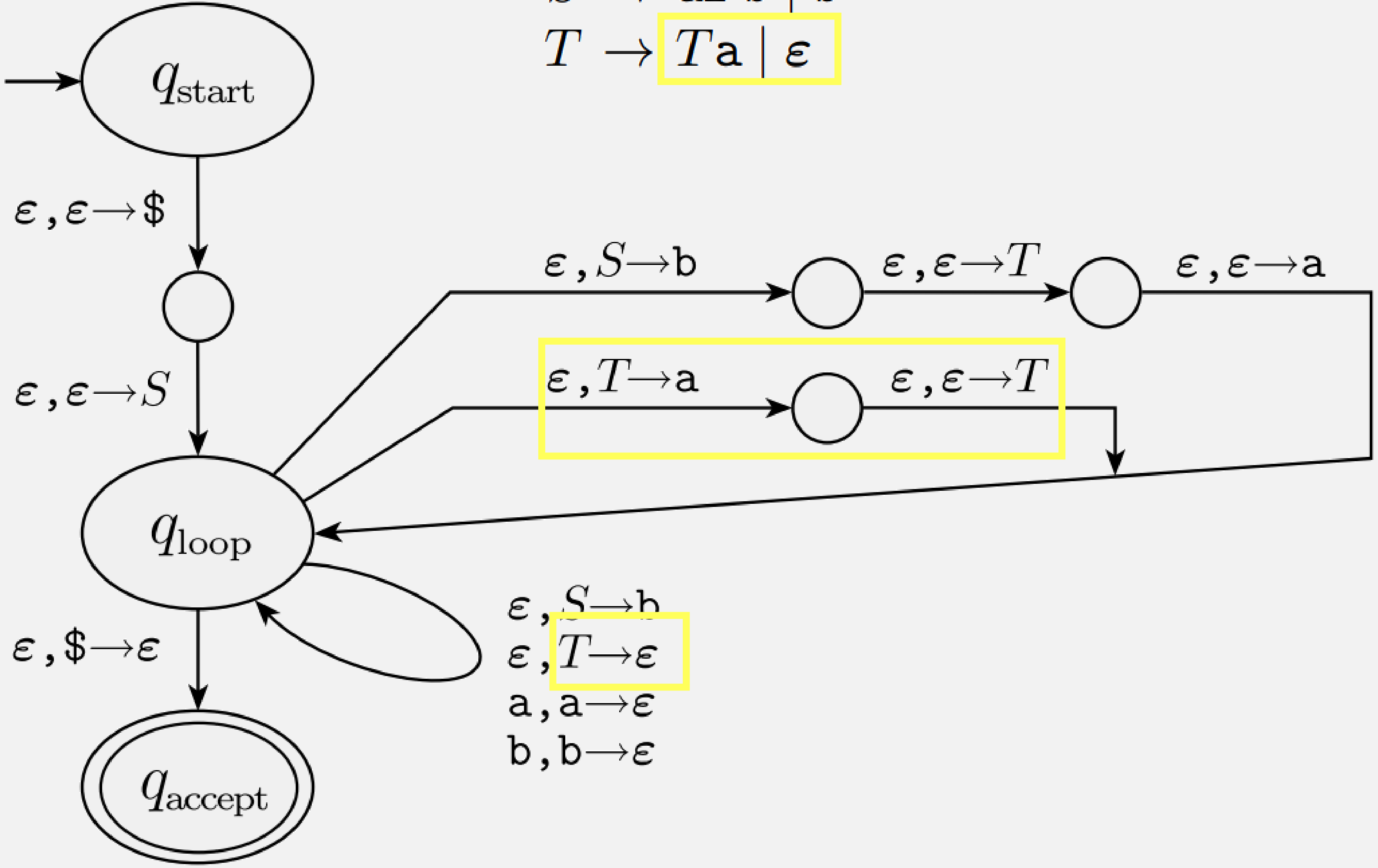
<u>push</u> **start variable** onto stack

If: <u>stack top</u> **is variable** S, <u>pop</u> S and …

… <u>push</u> **rule right-sides** (in rev order)

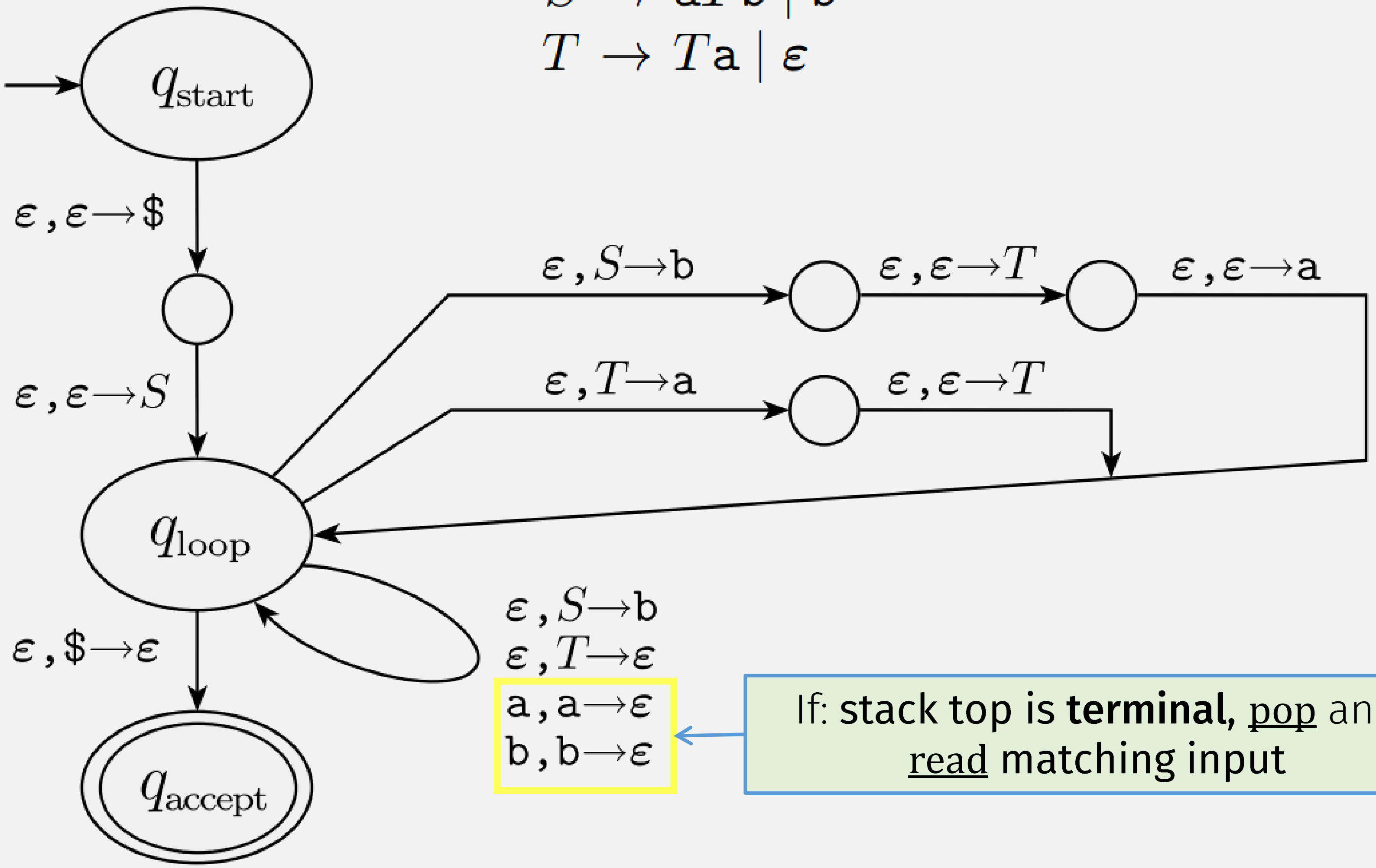
Example **CFG→PDA**

$$
\begin{aligned}
S &\to \mathtt{a}T\mathtt{b} \mid \mathtt{b} \\
T &\to \boxed{T\mathtt{a} \mid \varepsilon}
\end{aligned}
$$



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**, <u>pop</u> and <u>read</u> matching input

Example **CFG→PDA**

Example Derivation using CFG:

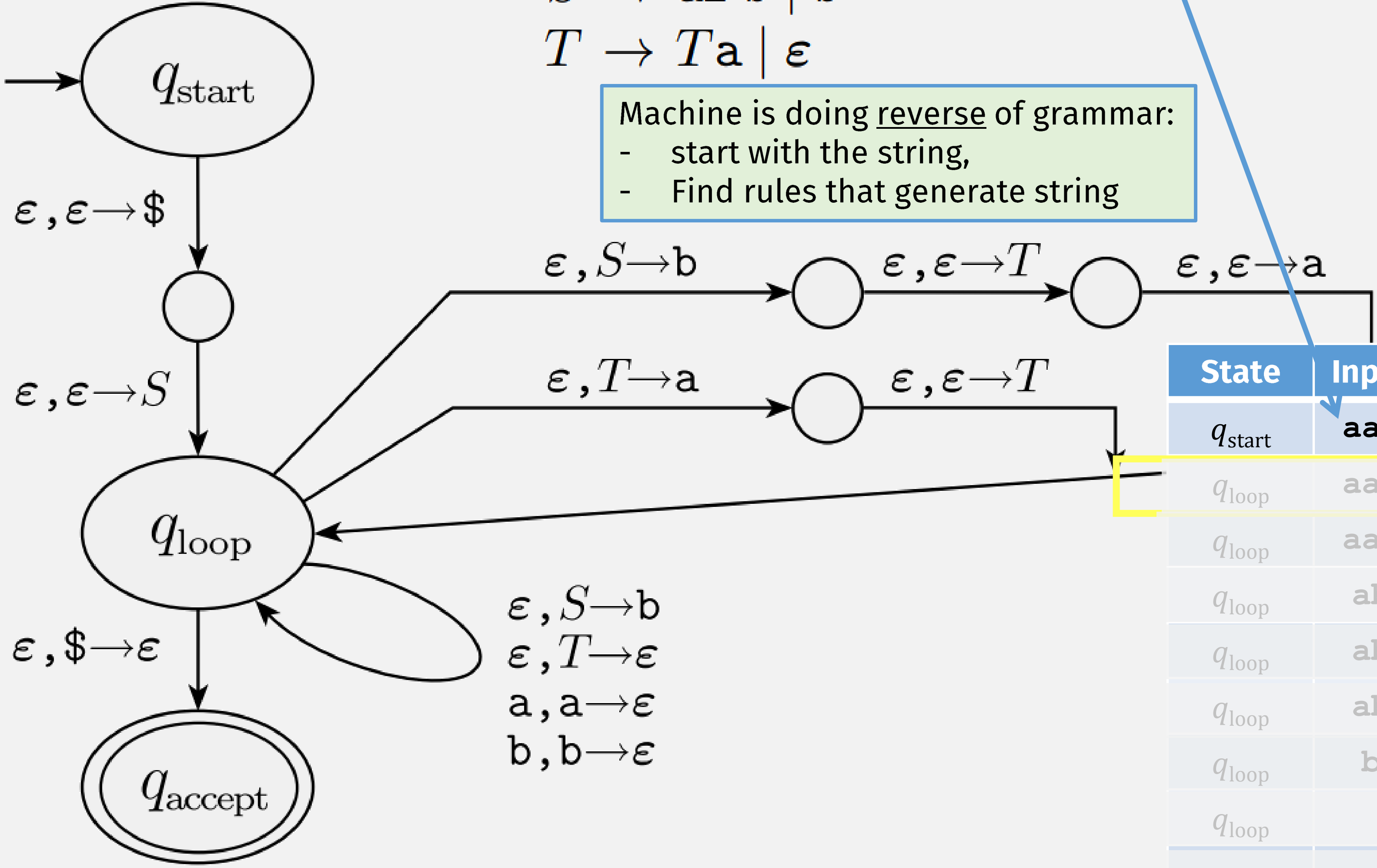
$S \Rightarrow \mathbf{a}T\mathbf{b}$ (using rule $S \rightarrow \mathbf{a}T\mathbf{b}$)

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

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

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

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



PDA Example

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

Example CFG→PDA

Example Derivation using CFG:

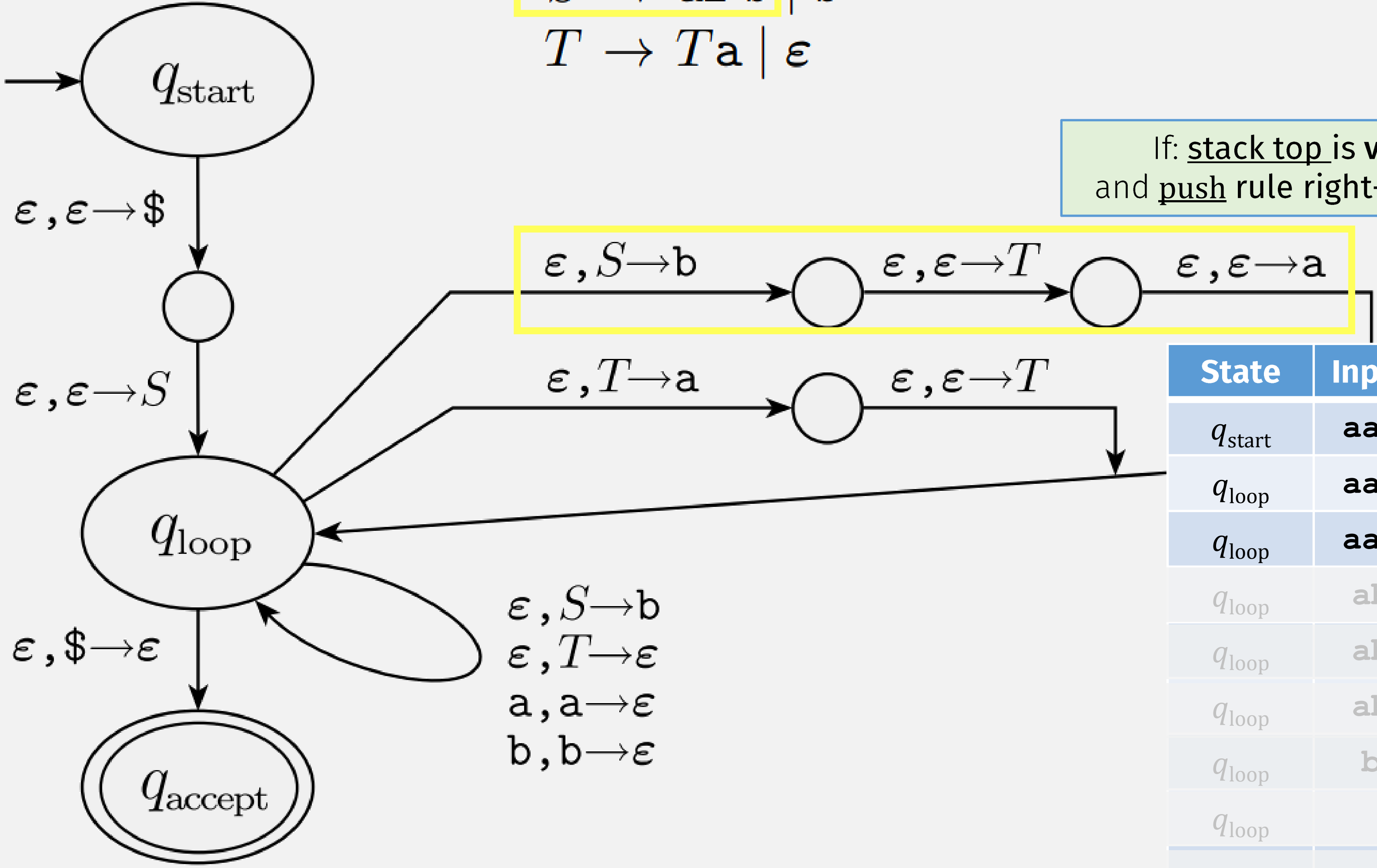
$S \Rightarrow \mathtt{a}T\mathtt{b}$ (using rule $S \to \mathtt{a}T\mathtt{b}$)

$\Rightarrow \mathtt{a}T\mathtt{ab}$ (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

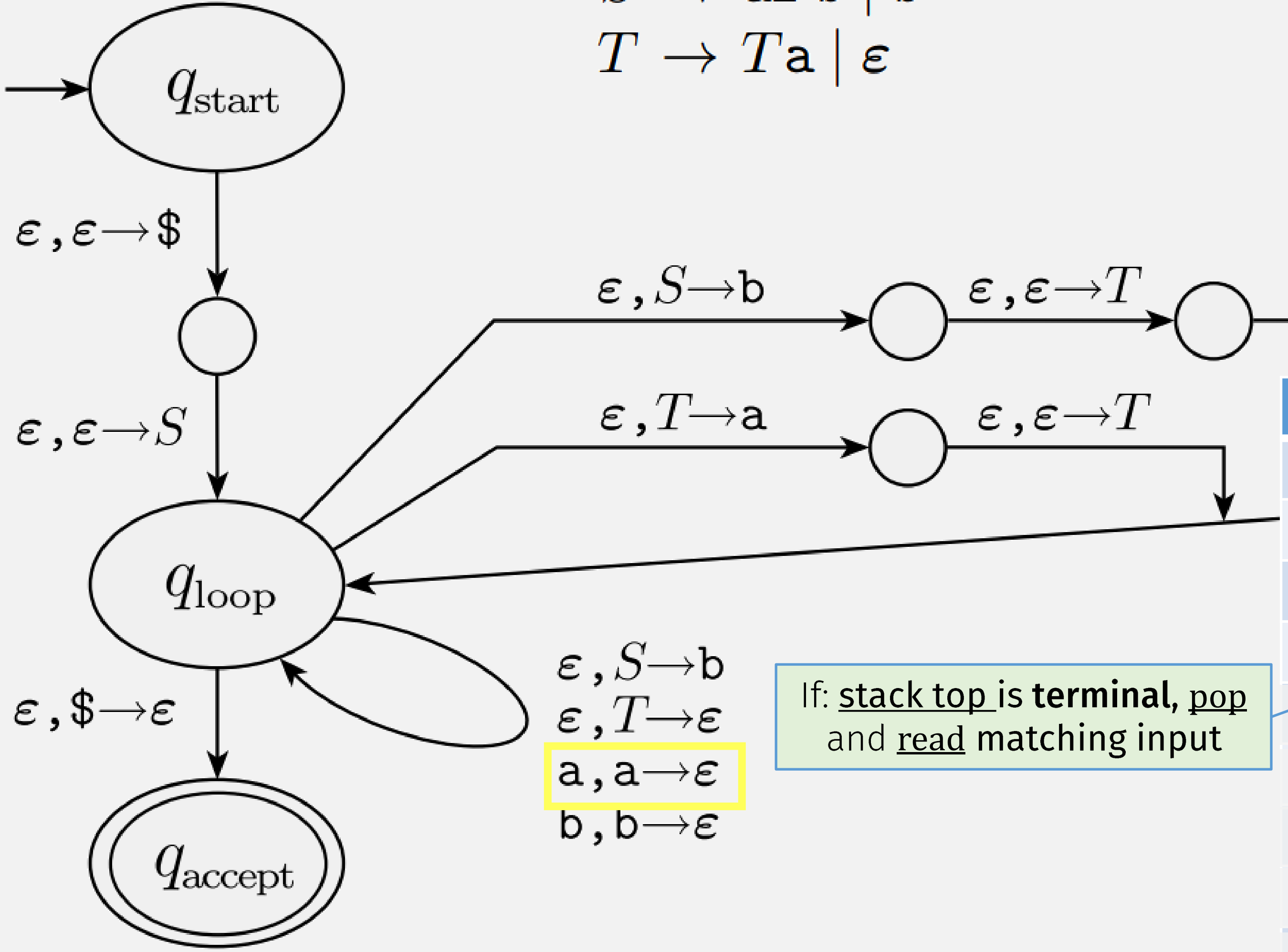
State	Input	Stack	Equiv Rule
q_{start}	aab		
q_{loop}	aab	$S\$$	
q_{loop}	aab	$\mathtt{a}T\mathtt{b}\$$	$S \to \mathtt{a}T\mathtt{b}$
q_{loop}	ab	$T\mathtt{b}\$$	
q_{loop}	ab	$T\mathtt{ab}\$$	$T \to T\mathtt{a}$
q_{loop}	ab	$\mathtt{ab}\$$	$T \to \varepsilon$
q_{loop}	b	$\mathtt{b}\$$	
q_{loop}		$\$$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

Example CFG→PDA

<u>Example Derivation using CFG:</u>

$S \Rightarrow \mathtt{a}T\mathtt{b}$ (using rule $S \rightarrow \mathtt{a}T\mathtt{b}$)
$\Rightarrow \boxed{\mathtt{a}}T\mathtt{ab}$ (using rule $T \rightarrow T\mathtt{a}$)
$\Rightarrow \mathtt{aab}$ (using rule $T \rightarrow \varepsilon$)

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



<u>PDA Example</u>

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

Example **CFG→PDA**

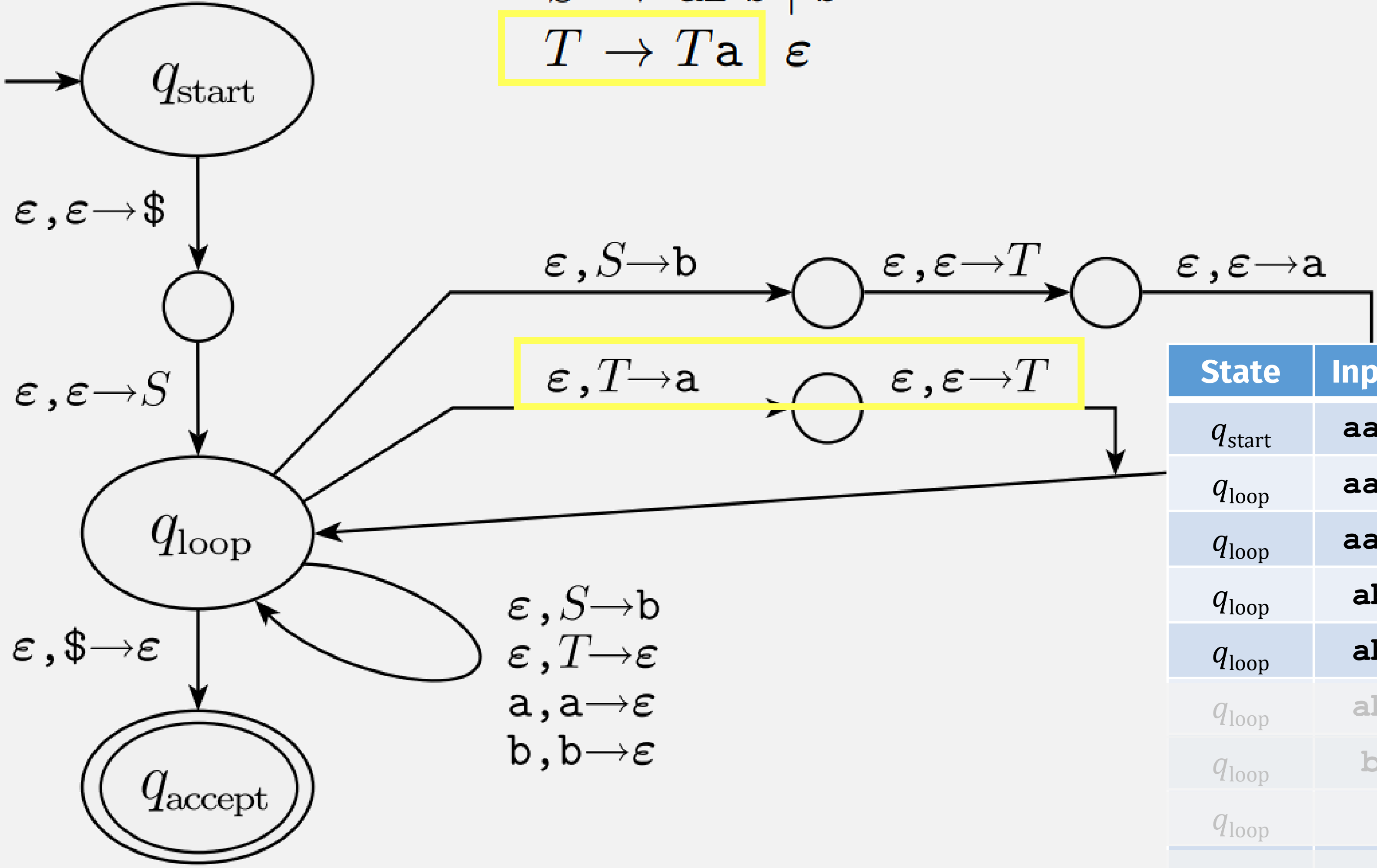
Example Derivation using CFG:

$S \Rightarrow \mathtt{a}T\mathtt{b}$ (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}			

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 $\{A_{pq} | \ p, q \in Q\}$

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

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

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

The Key IDEA

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

PDA P -> CFG G: Generating Strings

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

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

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

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

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

PDA P -> CFG G: Generating Strings

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

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

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

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

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

PDA P -> CFG G: Generating Strings

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

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

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

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

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

A language is a CFL ⇔ A PDA recognizes it

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

- Convert **CFG→PDA**

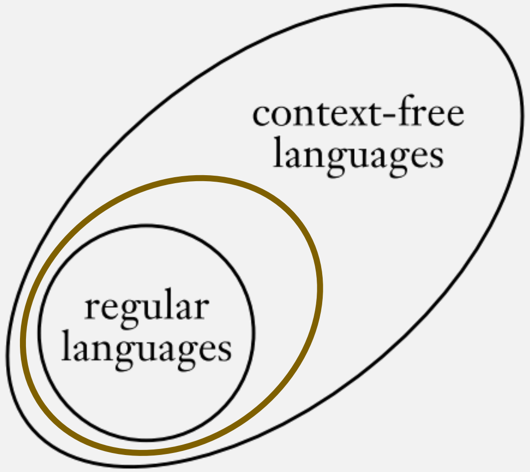
☑⇐ If **a PDA recognizes a language,** then **it's a CFL**

- Convert **PDA→CFG**

∎

Regular vs Context-Free Languages

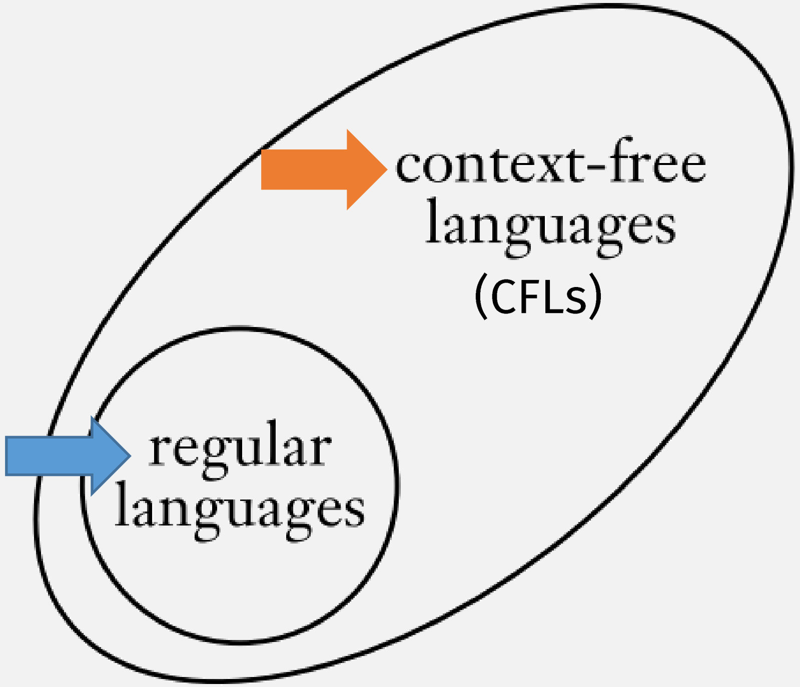
(and others?)



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"?

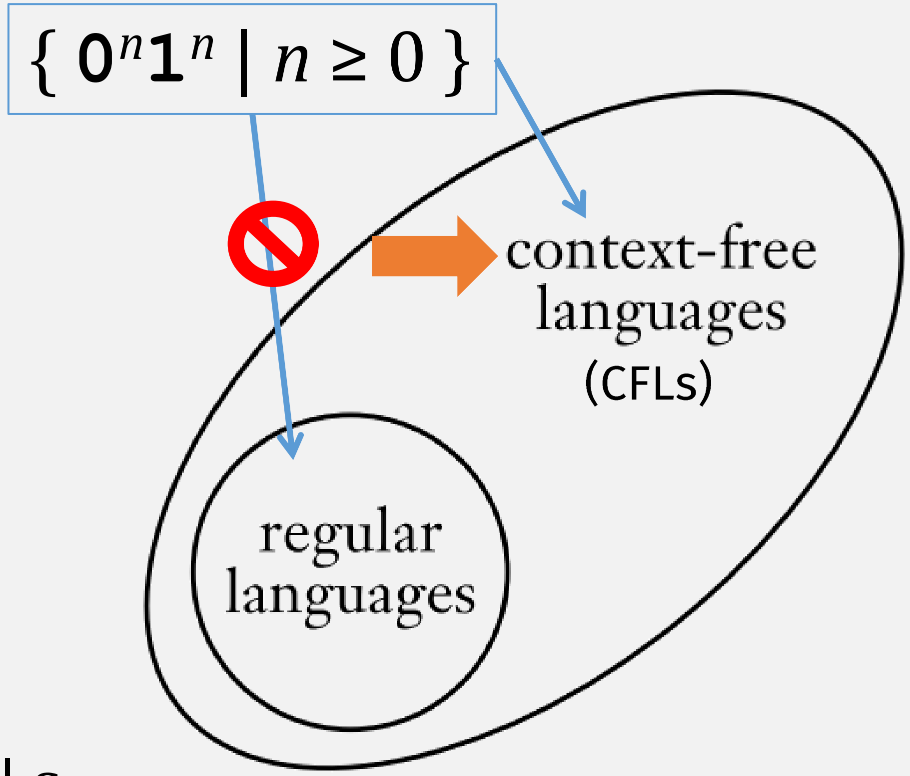
1. Every regular language is a CFL

2. Not every CFL is a regular language

Find a CFL that is not regular

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

- It's a CFL
  - *Proof*: CFG $S \rightarrow \mathbf{0}S\mathbf{1} \mid \varepsilon$
- It's not regular
  - *Proof*: by contradiction using the Pumping Lemma



How to Prove This Diagram “Correct”?

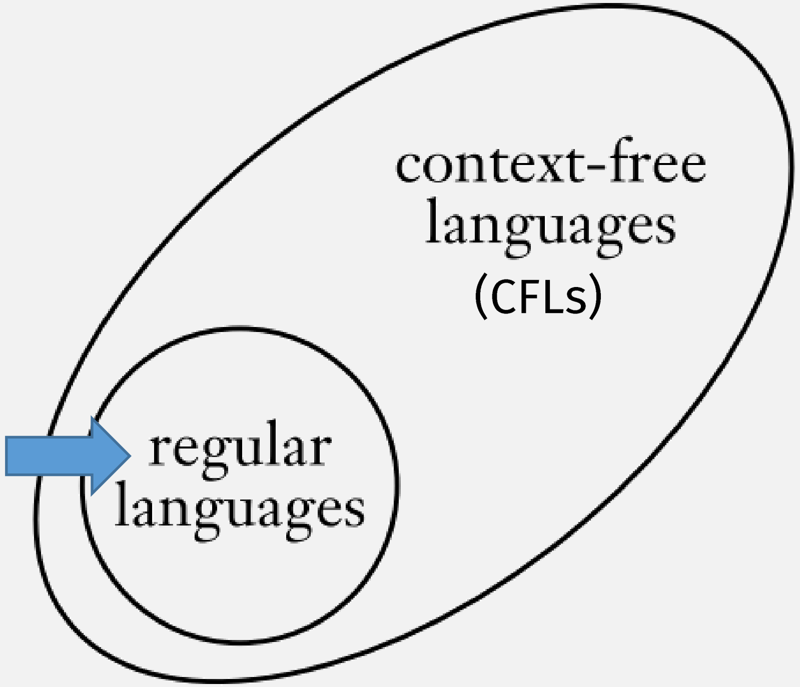
1. Every regular language is a CFL

For any regular language A, show ...

... it has a CFG or PDA

☑ 2. Not every CFL is a regular language

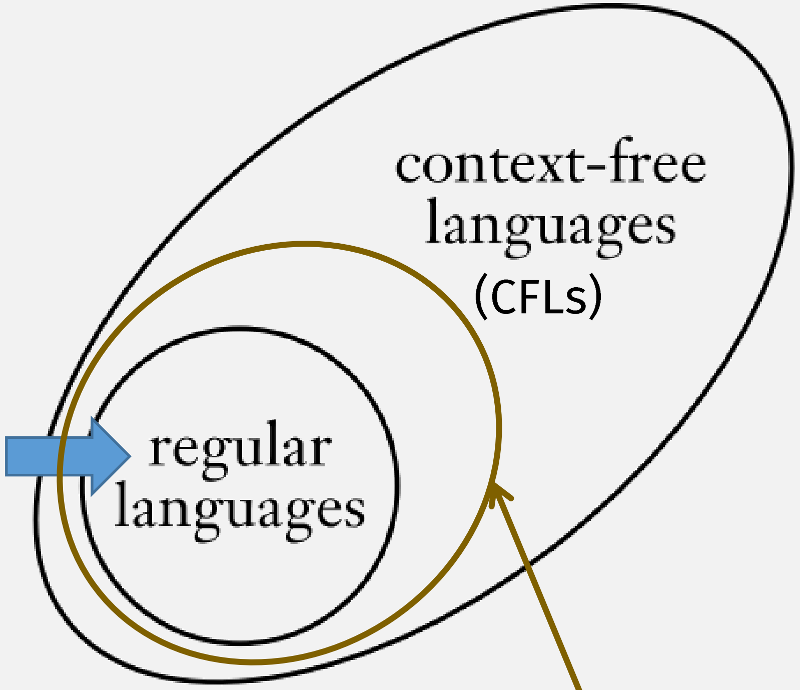
A regular language is represented by a:
- DFA
- NFA
- Regular Expression



Regular Languages are CFLs: 3 Ways to Prove

- DFA → CFG or PDA
- NFA → CFG or PDA
- Regular expression → CFG or PDA

See HW 6!



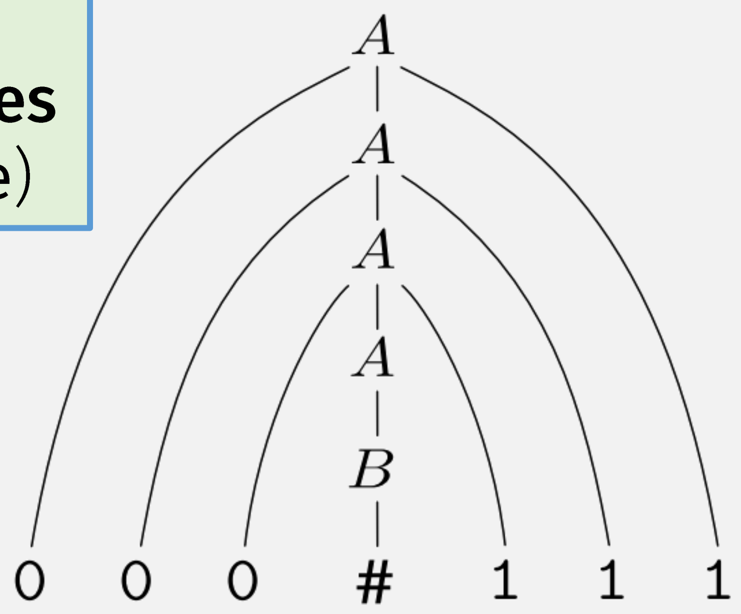
Are there other interesting subsets of CFLs?

Deterministic CFLs and DPDAs

Previously: Generating Strings

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

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

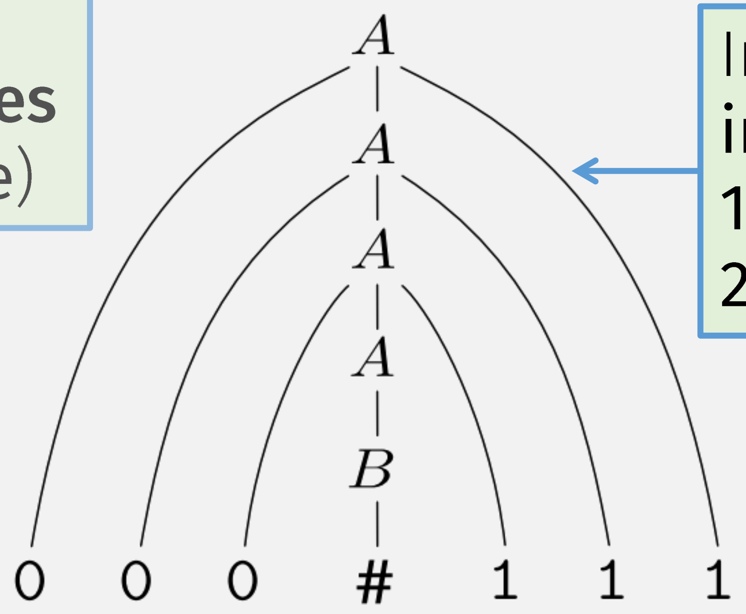


$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000B111 \Rightarrow 000\#111$$

Generating vs Parsing

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

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



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** one
  - E.g., a **compiler** (first) parses source code into a parse tree
  - (Actually, *any* program with string inputs must first parse it)

Previously: Example **CFG→PDA**

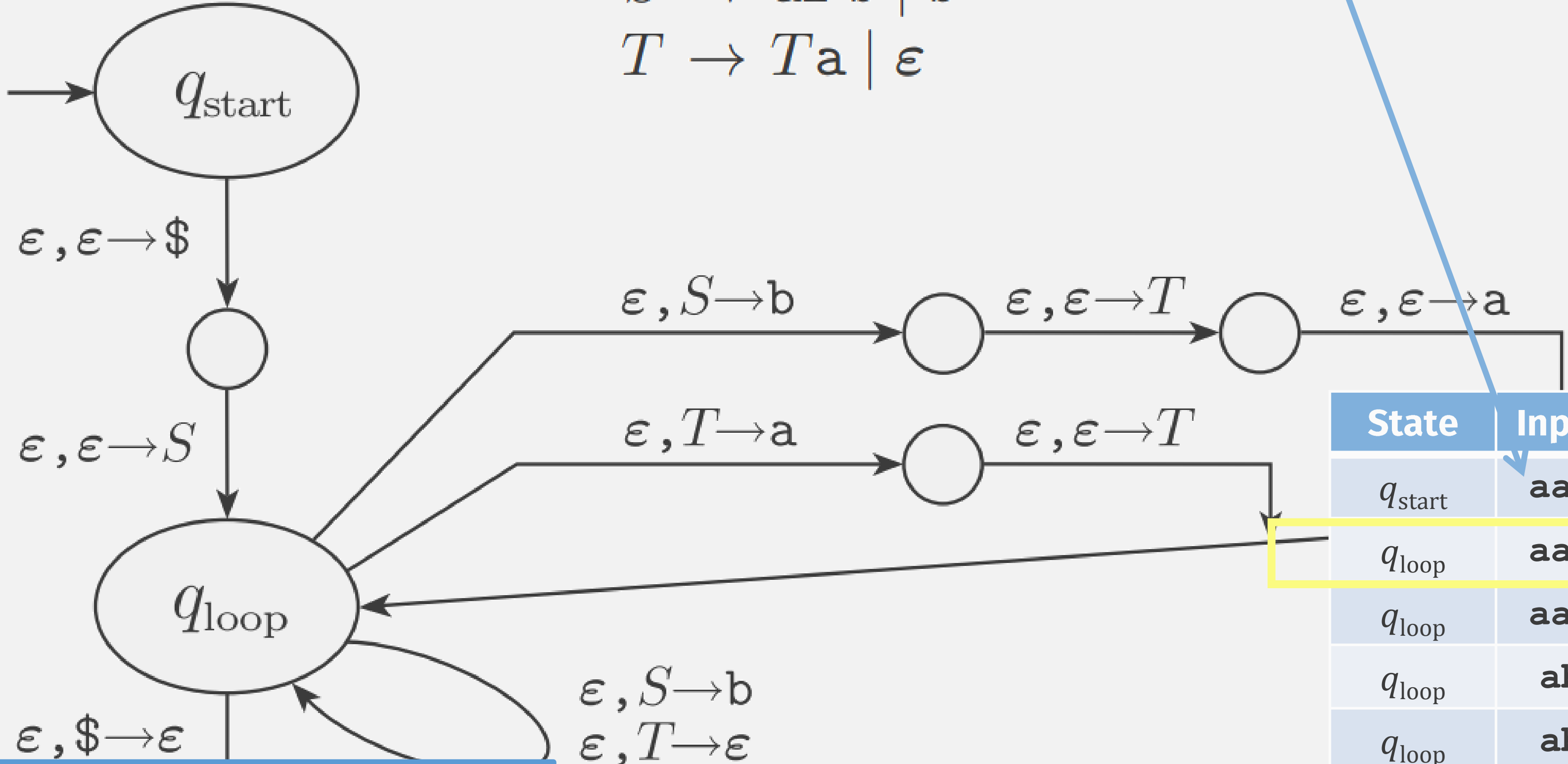
Example Derivation using CFG:

$S \Rightarrow \mathtt{a}T\mathtt{b}$ (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 $T \to \varepsilon$)

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



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}			

This Machine is **parsing:**

1. Start with (input) string,
2. Find **rules** that **generate** string

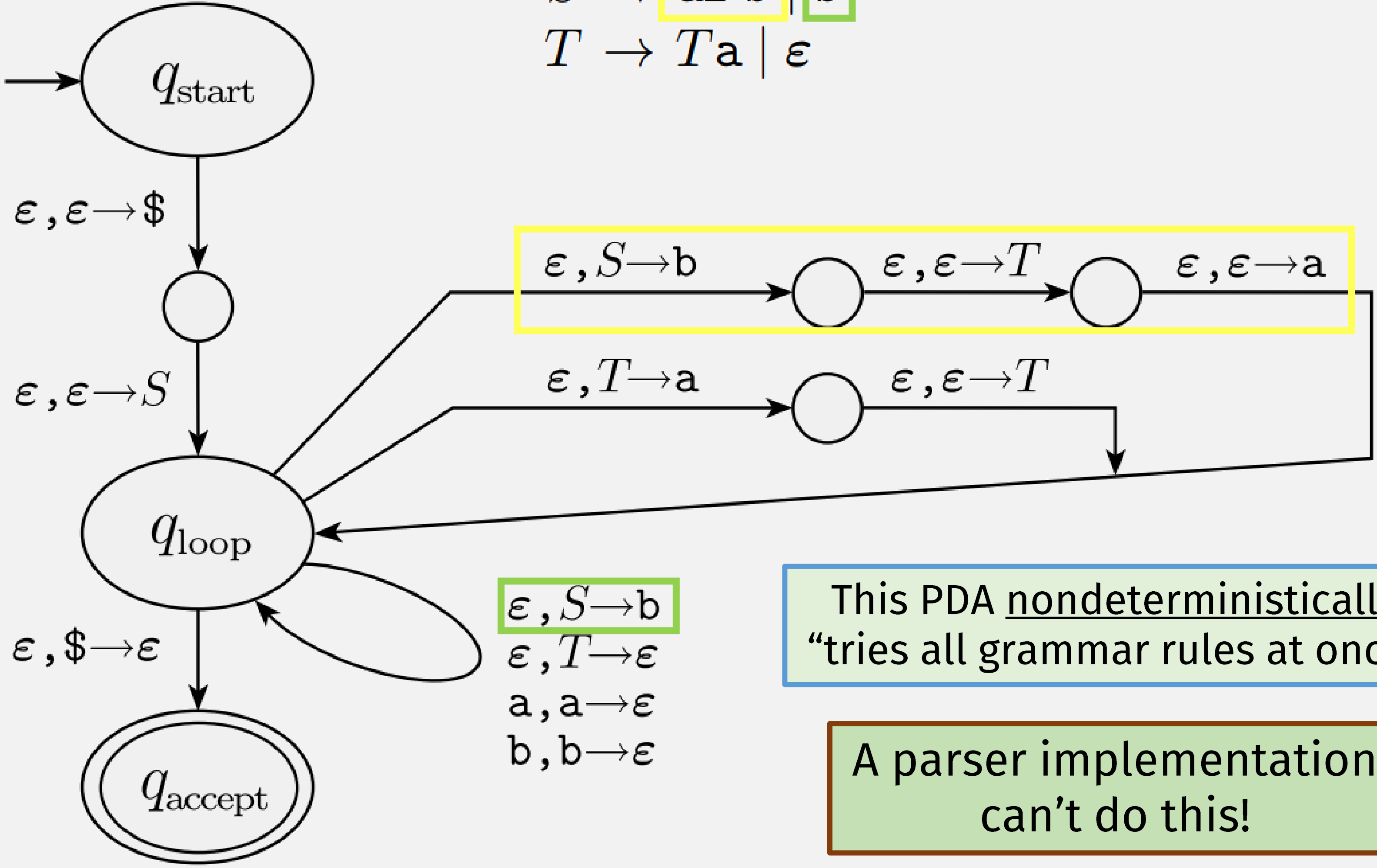
Generating vs Parsing

- In practice, **parsing** a string more important than **generating** 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 <u>non-deterministic</u> (like NFAs)

Previously: (Nondeterministic) PDA

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



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 <u>non-deterministic</u> (like NFAs)

- Compiler's parsing algorithm must be <u>deterministic</u>

- <u>So</u>: 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. $\delta \colon Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow \boxed{(Q \times \Gamma_\varepsilon) \cup \{\emptyset\}}$ 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"

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. $\delta \colon Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow \boxed{\mathcal{P}}(Q \times \Gamma_\varepsilon)$
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 take into account ε reads or stack ops!
E.g., if $\delta(q, \mathrm{a}, X)$ 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

$\mathtt{aaabbb} \rightarrowtail \mathtt{aaSbb}$

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

used T rule

$\mathtt{aaabbbbbb} \rightarrowtail \mathtt{aaTbbbb}$

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

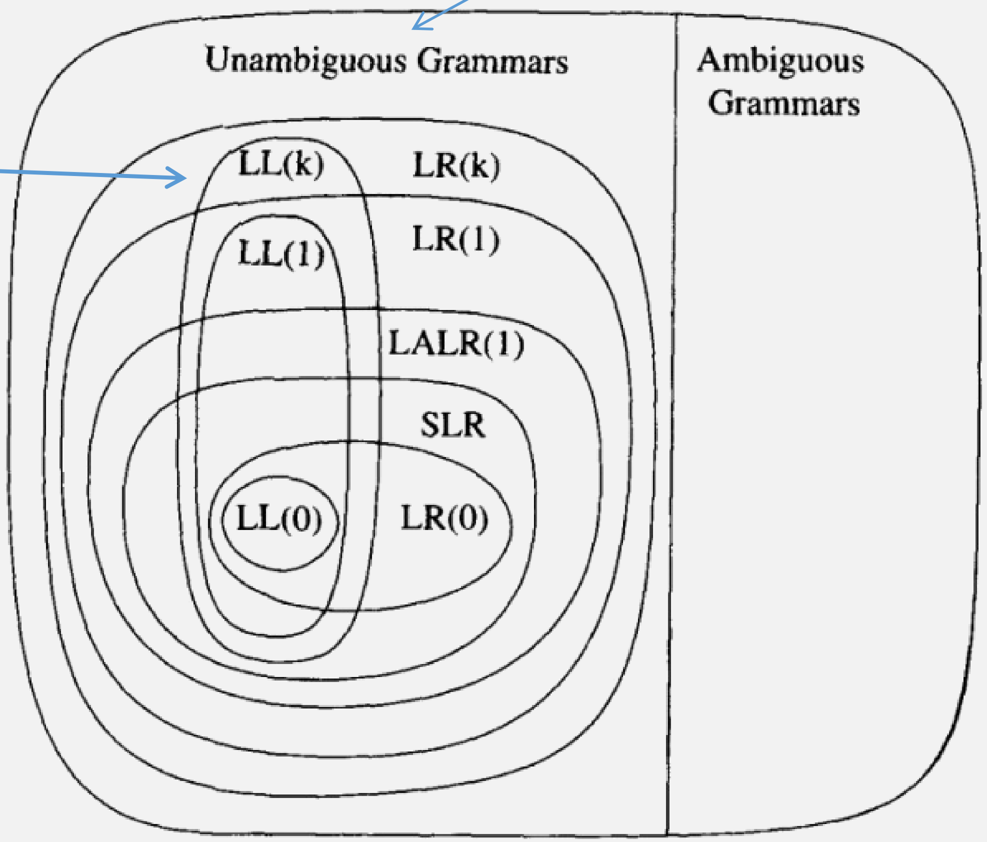
PDAs recognize CFLs, but **DPDAs only recognize DCFLs!** (a subset of CFLs)

Subclasses of CFLs

Umambiguous CFLs / PDAs

DCFLs

Programming language parsers / compilers are ideally in here



All CFLS

Compiler Stages

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

Lexer

DFAs (recognizing regular languages) in here!

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

A Lexer Implementation

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

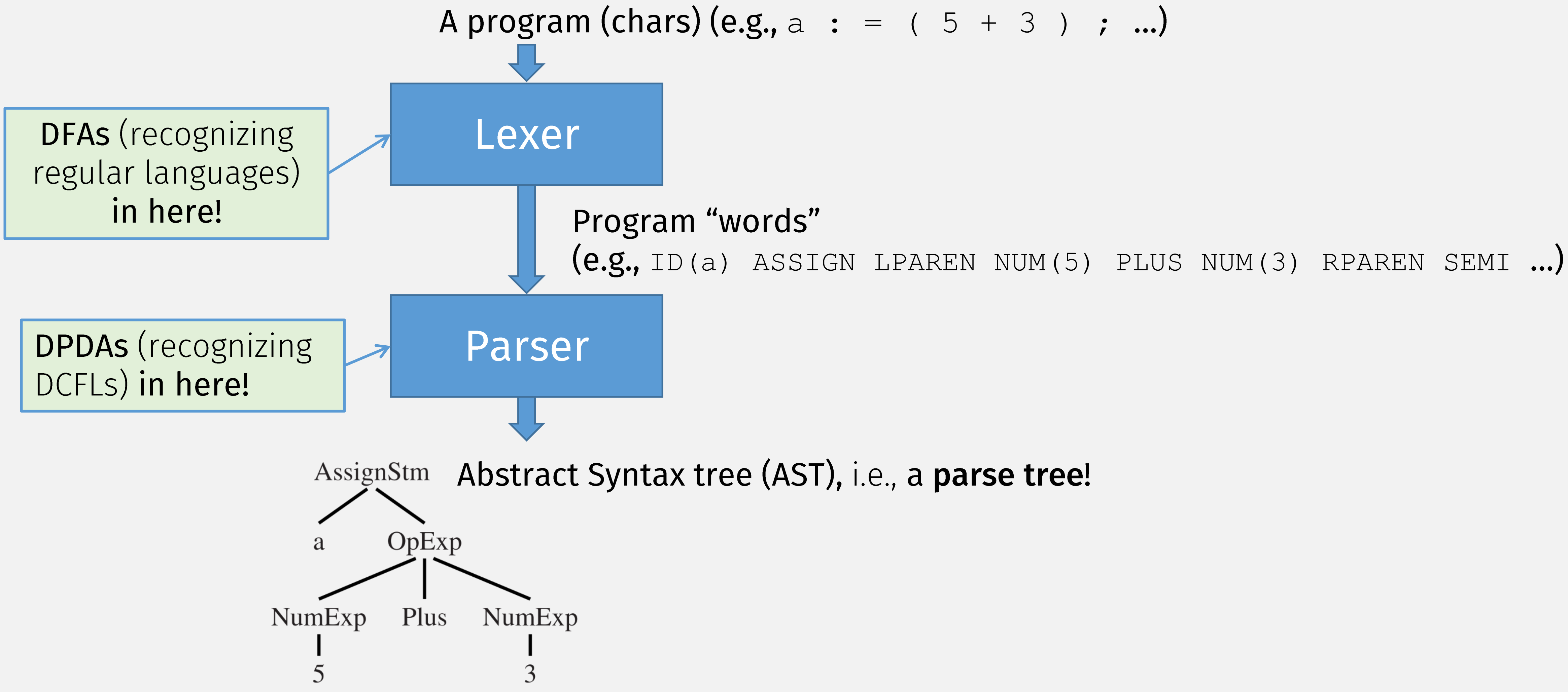
<u>This</u> DFA <u>is</u> a real program!

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 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**

This CFG is a real program!

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

DPDAs are <u>Not</u> Equivalent to PDAs!

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

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

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

Should use S rule

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

$\mathtt{aa\underline{a}}$

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

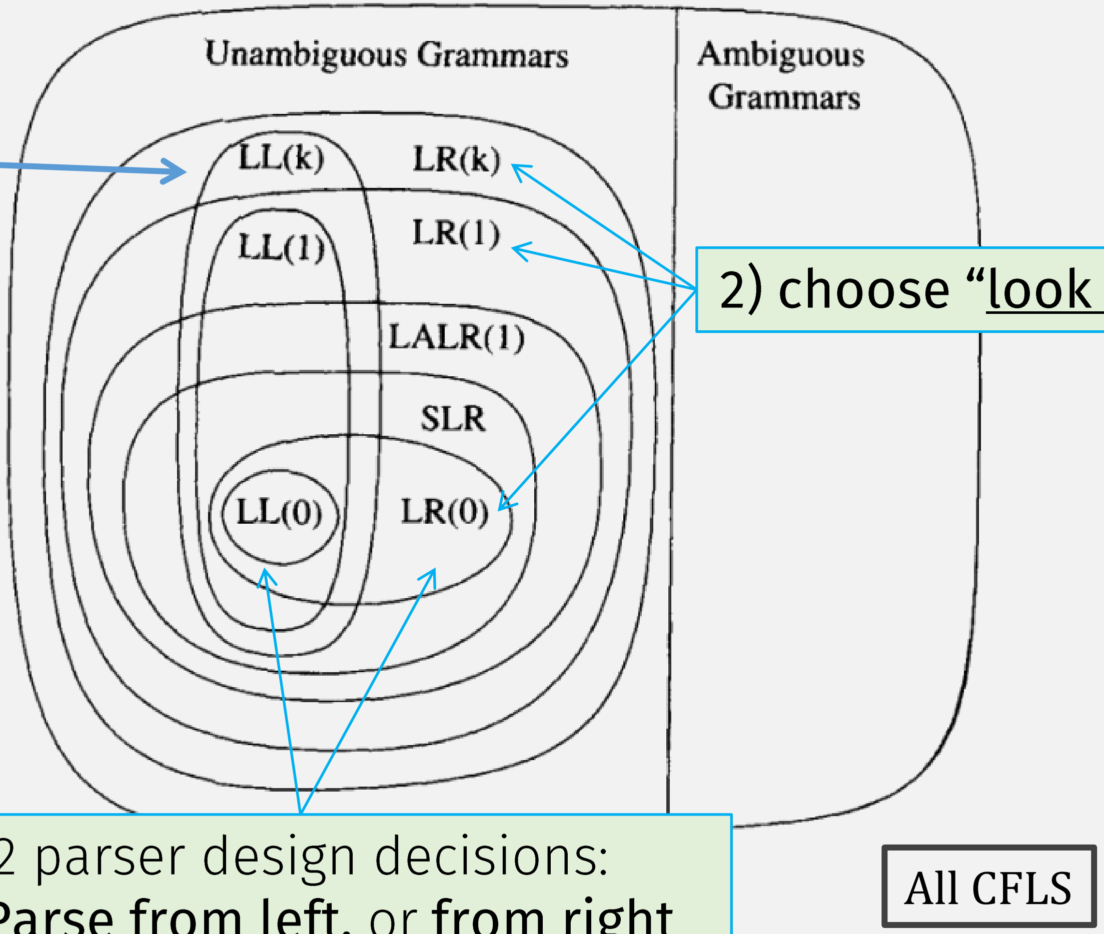
Should use T rule

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

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

PDAs recognize CFLs, but **DPDAs** only recognize **DCFLs!** (a <u>subset</u> 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

LL parsing

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

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

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

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

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

```
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 $S \rightarrow \boxed{\text{begin } S\ L}$

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

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 (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 $\boxed{S \rightarrow \text{id} := E}$
3 $S \rightarrow \text{print} \; (\; L \;)$
4 $\boxed{E \rightarrow \text{id}}$
5 $E \rightarrow \text{num}$
6 $\boxed{E \rightarrow E + E}$

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

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

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

C

LR parsing

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

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

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

Stack	*Input*	*Action*
1	`a := 7 ; b := c + ( d := 5 + 6 , d ) $`	*shift*

push

= “push”

State name

LR parsing

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

$$
\begin{array}{ll}
S \rightarrow S\ ;\ S & E \rightarrow \mathrm{id} \\
S \rightarrow \mathrm{id}\ :=\ E & E \rightarrow \mathrm{num} \\
S \rightarrow \mathrm{print}\ (\ L\ ) & E \rightarrow E\ +\ E
\end{array}
$$

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*

LR parsing

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

$$
\begin{array}{ll}
S \rightarrow S \,;\, S & E \rightarrow \mathrm{id} \\
S \rightarrow \mathrm{id} := E & E \rightarrow \mathrm{num} \\
S \rightarrow \mathrm{print}\,(\,L\,) & E \rightarrow E + E
\end{array}
$$

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

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

6 $E \rightarrow E + E$

Can determine (rightmost) rule

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$_4$:=$_6$ $\boxed{\text{num}_{10}}$	; b := c + (d := 5 + 6 , d) $	*reduce* $E \rightarrow$ num

LR parsing

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

1 $S \rightarrow S \; ; \; S$

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

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*
$_1 \; \text{id}_4$	`:= 7 ; b := c + ( d := 5 + 6 , d ) $`	*shift*
$_1 \; \text{id}_4 \; :=_6$	`= c + ( d := 5 + 6 , d ) $`	*shift*
$_1 \; \text{id}_4 \; :=_6 \; \text{num}_{10}$	`= c + ( d := 5 + 6 , d ) $`	*reduce* $E \rightarrow \text{num}$
$_1 \; \text{id}_4 \; :=_6 \; E_{11}$	`; b := c + ( d := 5 + 6 , d ) $`	*reduce* $S \rightarrow \text{id} := E$

Can determine (rightmost) rule

LR parsing

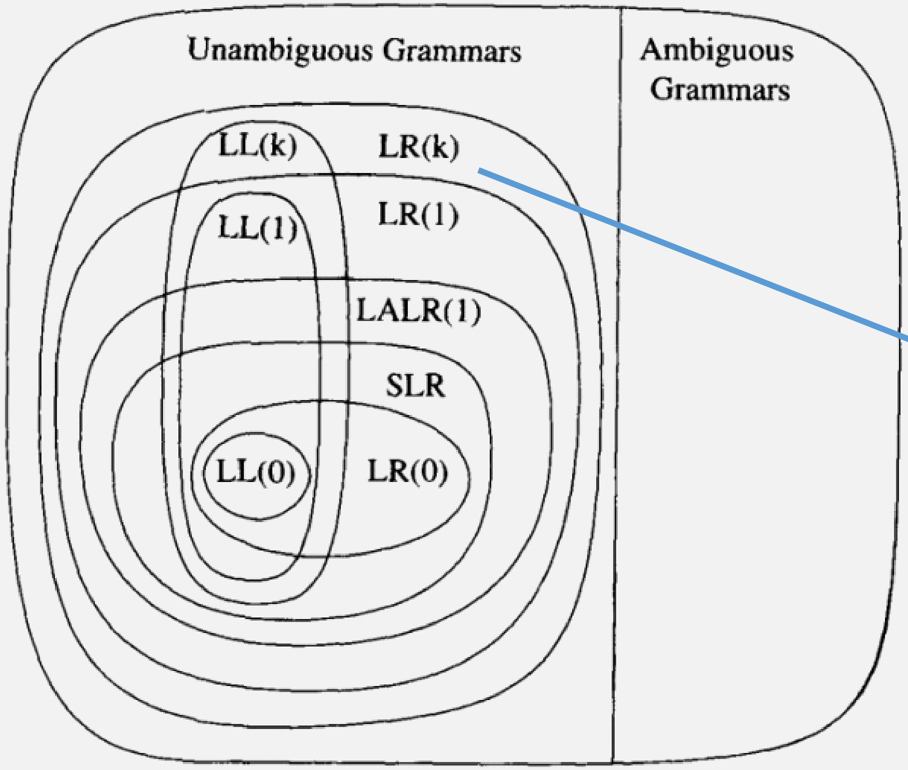
- **L** = left-to-right
- **R** = rightmost derivation

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

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* $E \to$ num
$_1$ $\boxed{\text{id}_4 :=_6 E_{11}}$	; b := c + (d := 5 + 6 , d) \$	*reduce* $S \to$ id:=E
$_1$ $\boxed{S_2}$	; 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