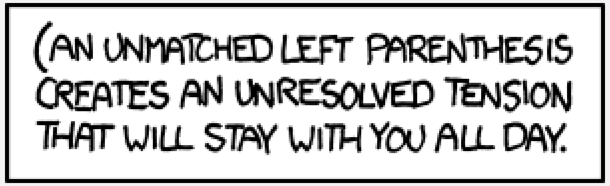
CS 420 / CS 620

CFGs vs PDAs
subCFLs and DPDAs

Monday October 27, 2025

UMass Boston Computer Science

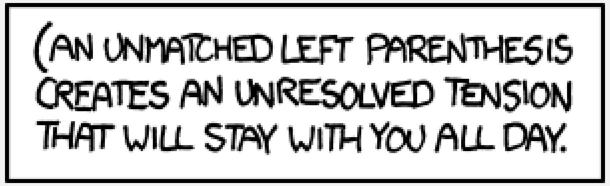


Announcements

- HW 7
  - Out: Mon 10/20 12pm (noon)
  - Due: Mon 10/27 12pm (noon)

- HW notes
  - Correct Gradescope page assignment of problems is now part of the correctness each submission

- Gradescope note
  - Regrade requests must address a specific deduction



Last Time:

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

Last Time:

A lang is a CFL iff some PDA recognizes it

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

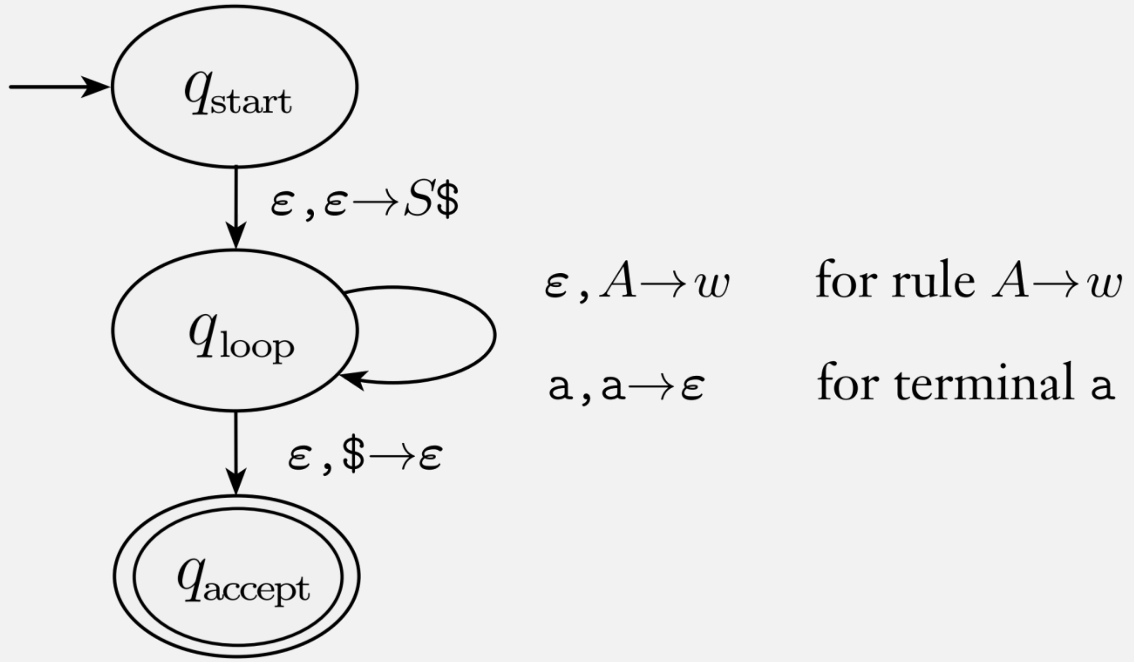
- We know: A **CFL** has a **CFG describing it** (definition of CFL)
- To prove this part, show: the **CFG** has an equivalent **PDA**

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

Last Time:

CFG→PDA (sketch)

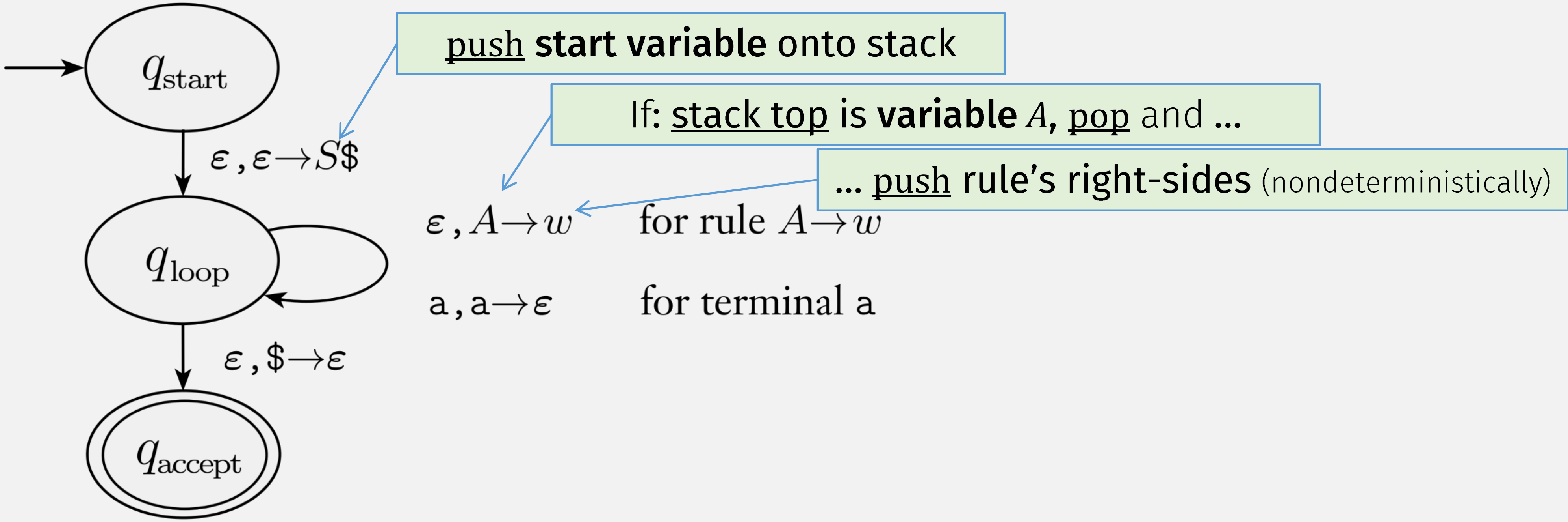
- Construct PDA from CFG such that:
  - PDA accepts input only if CFG generates it
- PDA:
  - simulates generating a string with CFG rules
  - by (nondeterministically) trying all rules to find the right ones



Last Time:

CFG→PDA (sketch)

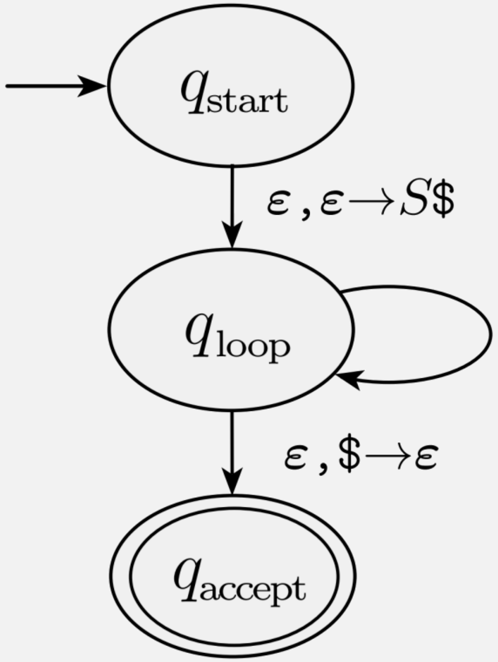
- **Construct PDA from CFG** such that:
  - PDA accepts input only if CFG generates it
- PDA:
  - simulates generating a string with CFG rules
  - by (nondeterministically) **trying all rules** to find the right ones



Last Time:

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



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

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

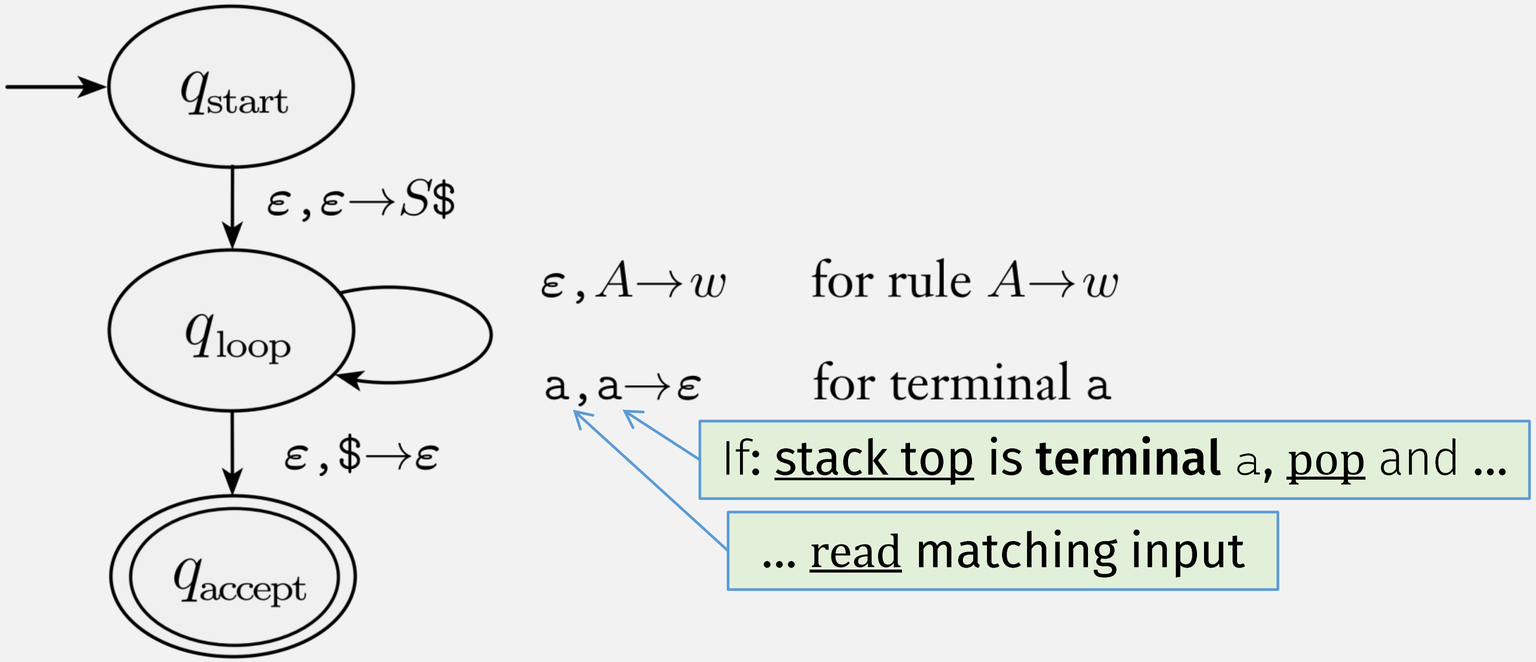
$\varepsilon, A \to w$ for rule $A \to w$

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

Last Time:

CFG→PDA (sketch)

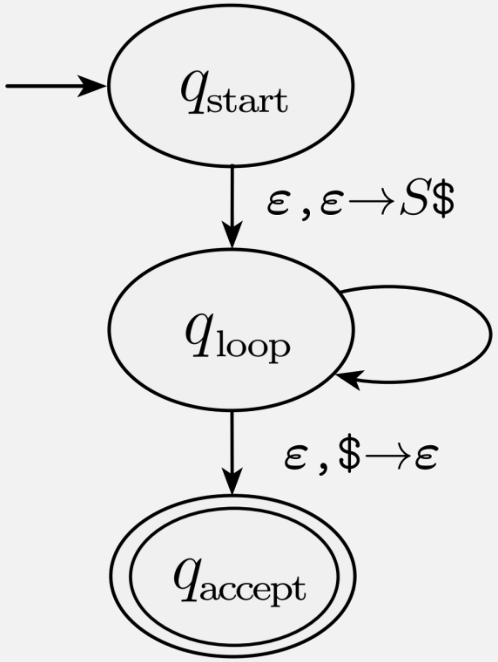
- Construct PDA from CFG such that:
  - PDA accepts input only if CFG generates it
- PDA:
  - simulates generating a string with CFG rules
  - by (nondeterministically) **trying all rules** to find the right ones



Last Time:

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

Summary: **convert <u>every terminal</u> to “loop” transition** that:
- **Reads input char**
- **Pops <u>matching</u> char on stack**

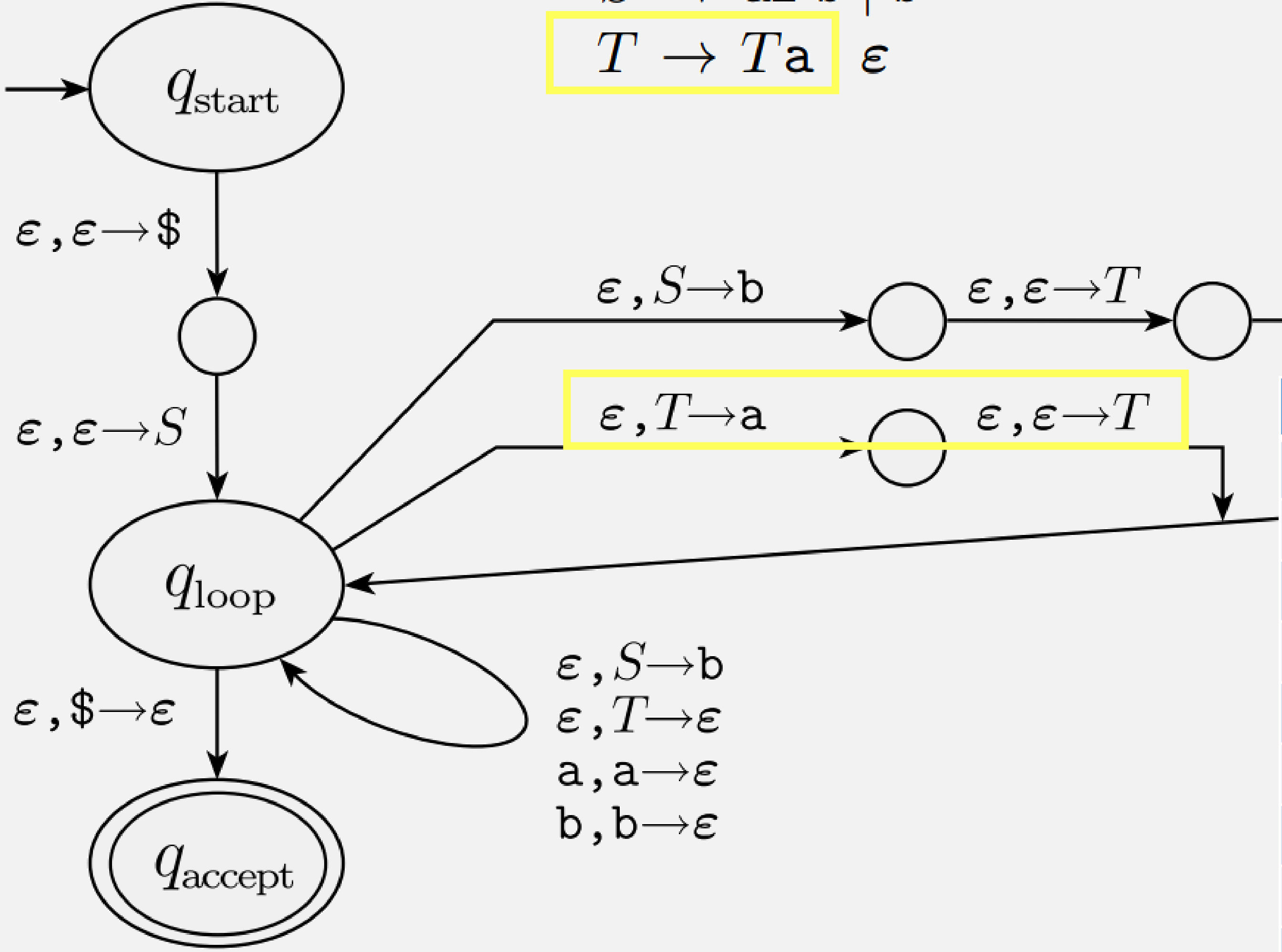
(**Read** the **terminals** as they become known)

Last Time:

Example **CFG→PDA**

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

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



Example Derivation using CFG:

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

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

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

PDA Example

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

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

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

- **Convert CFG→PDA**

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

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

PDA→CFG: Prelims

Before converting PDA to CFG, modify it so :

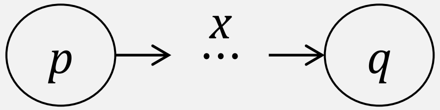
1. It has a single accept state, q_{accept}.
2. It empties its stack before accepting.
3. Each transition either pushes a symbol onto the stack (a *push* move) or pops one off the stack (a *pop* move), but it does not do both at the same time.

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

PDA P -> CFG G: Transitions and Variables

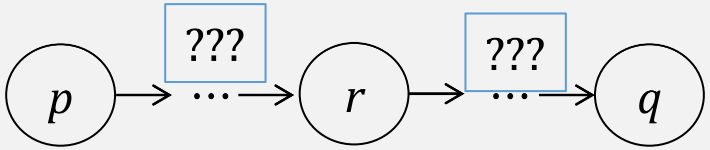
$P = (Q, \Sigma, \Gamma, \delta, q_0, \{q_{\text{accept}}\})$ variables of G are $\{A_{pq} \mid p, q \in Q\}$

- 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 $A_{pq} \to aA_{rs}b$ in G

PDA P -> CFG G: Generating Strings

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

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

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

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

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

PDA P -> CFG G: Generating Strings

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

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

- 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

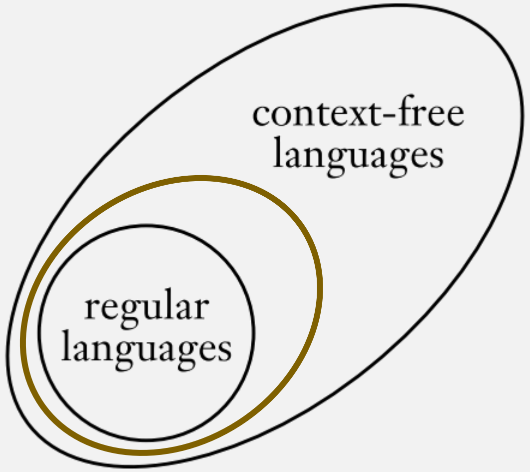
☑⇐ If **a PDA recognizes a language**, then **it's a CFL**
- Convert PDA→CFG

∎

Regular 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	Deterministic Finite-State Automata (DFA)	**Push-down Automata** (PDA)	thm
	recognizes a **Regular Lang**	recognizes a **CFL**	
	Proved:	Proved:	
	Regular Lang ⇔ Regular Expr ☑	CFL ⇔ PDA ☑	

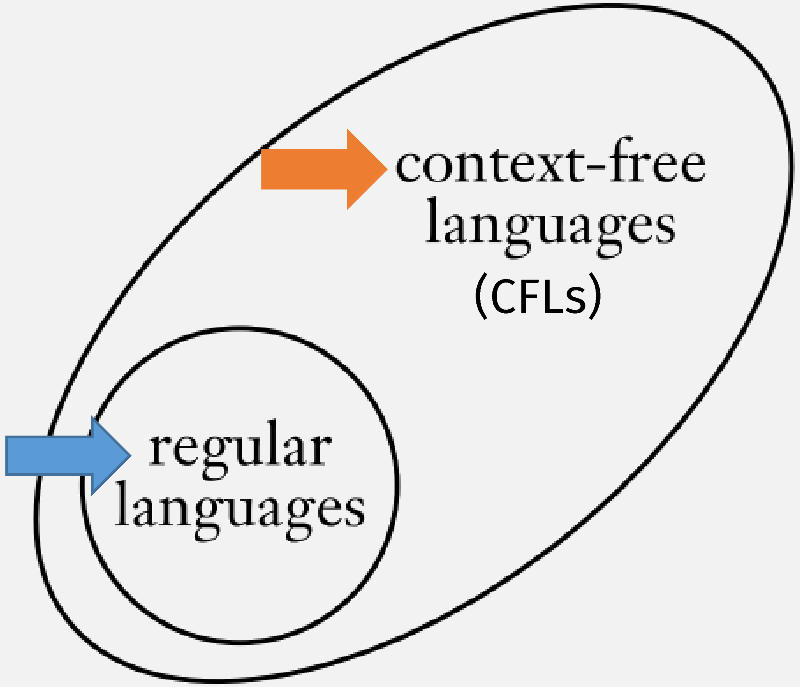
Regular vs Context-Free Languages
(and others?)



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"?

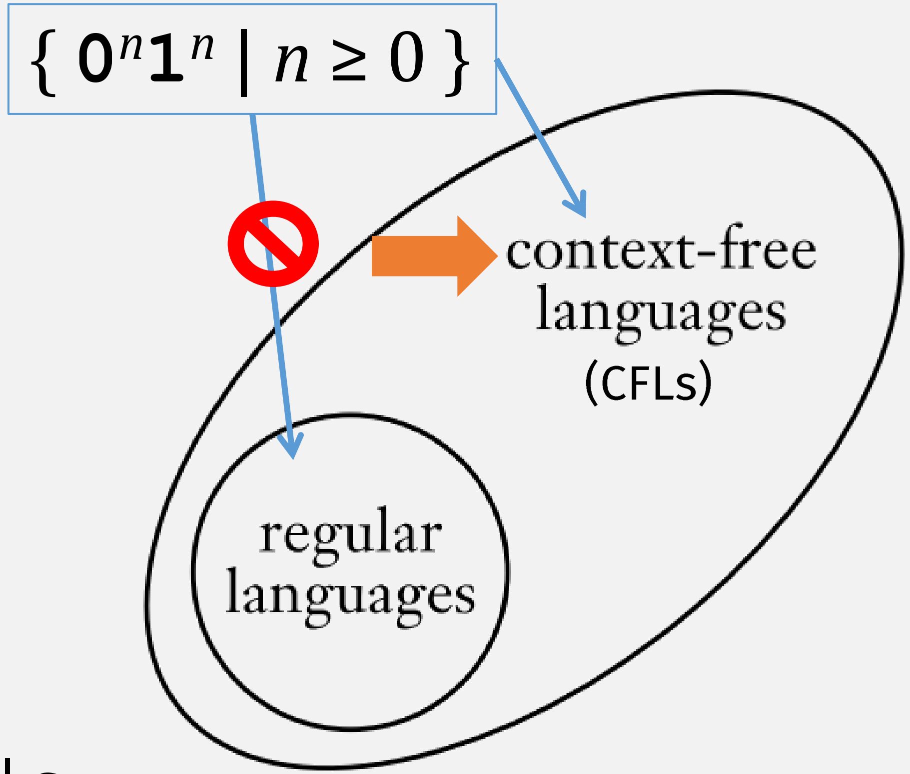
1. Every regular language is a CFL

2. Not every CFL is a regular language

Find a counterexample CFL that is not regular

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

- It's a CFL
  - *Proof*: CFG $S \rightarrow 0S1 \mid \varepsilon$
- It's not regular
  - *Proof*: by contradiction using the Pumping Lemma



How to Prove This Diagram "Correct"?

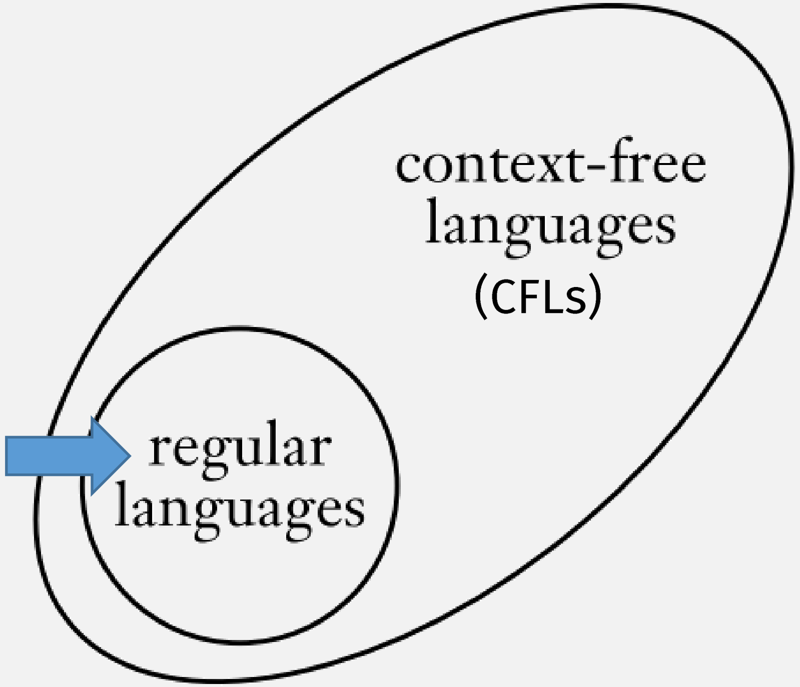
1. Every regular language is a CFL

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

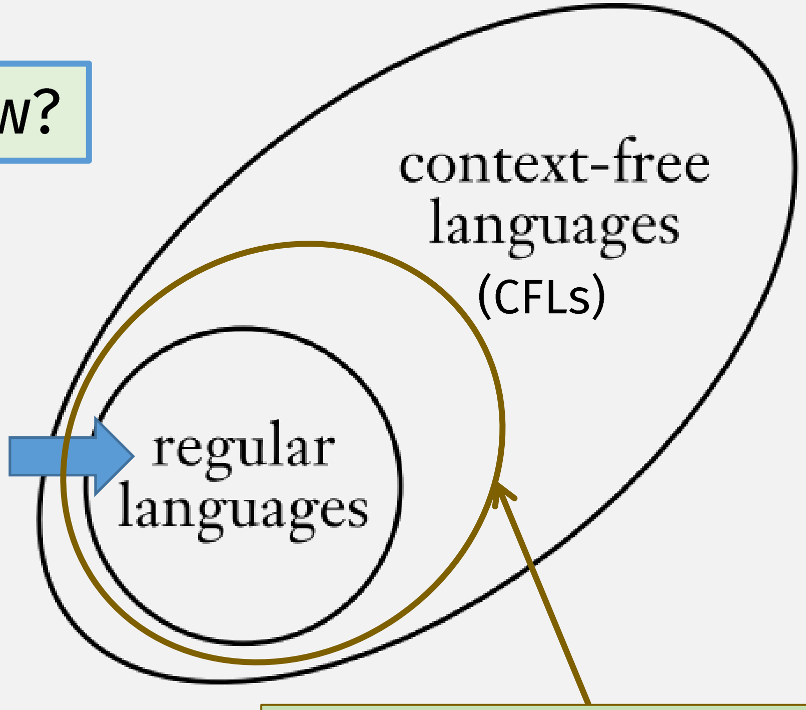


Regular Languages are CFLs: 3 Ways to Prove

- DFA → CFG or PDA

Coming soon to a future hw?

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



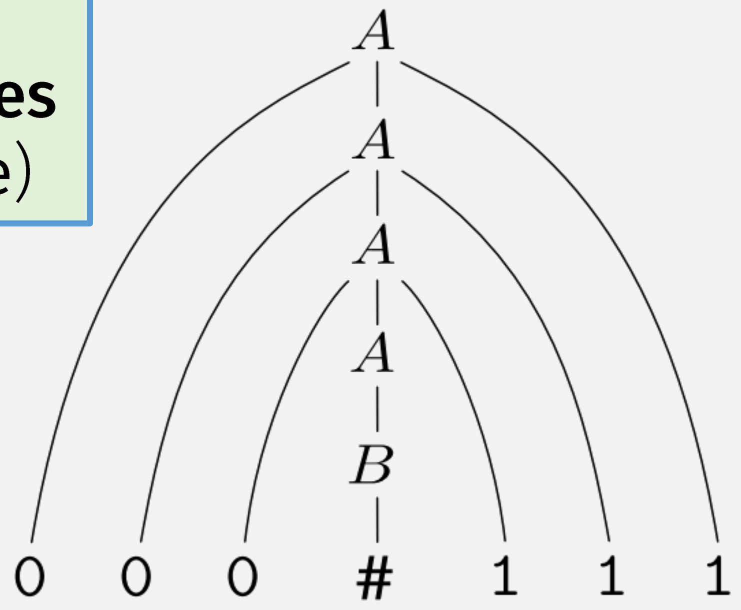
Are there other interesting <u>subsets of CFLs</u>?

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**)

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

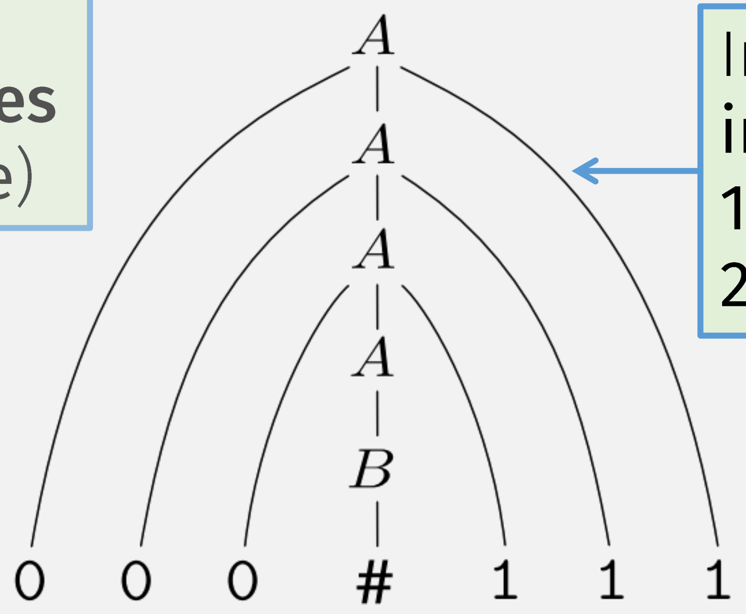


$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000B111 \Rightarrow 000\#111$$

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**

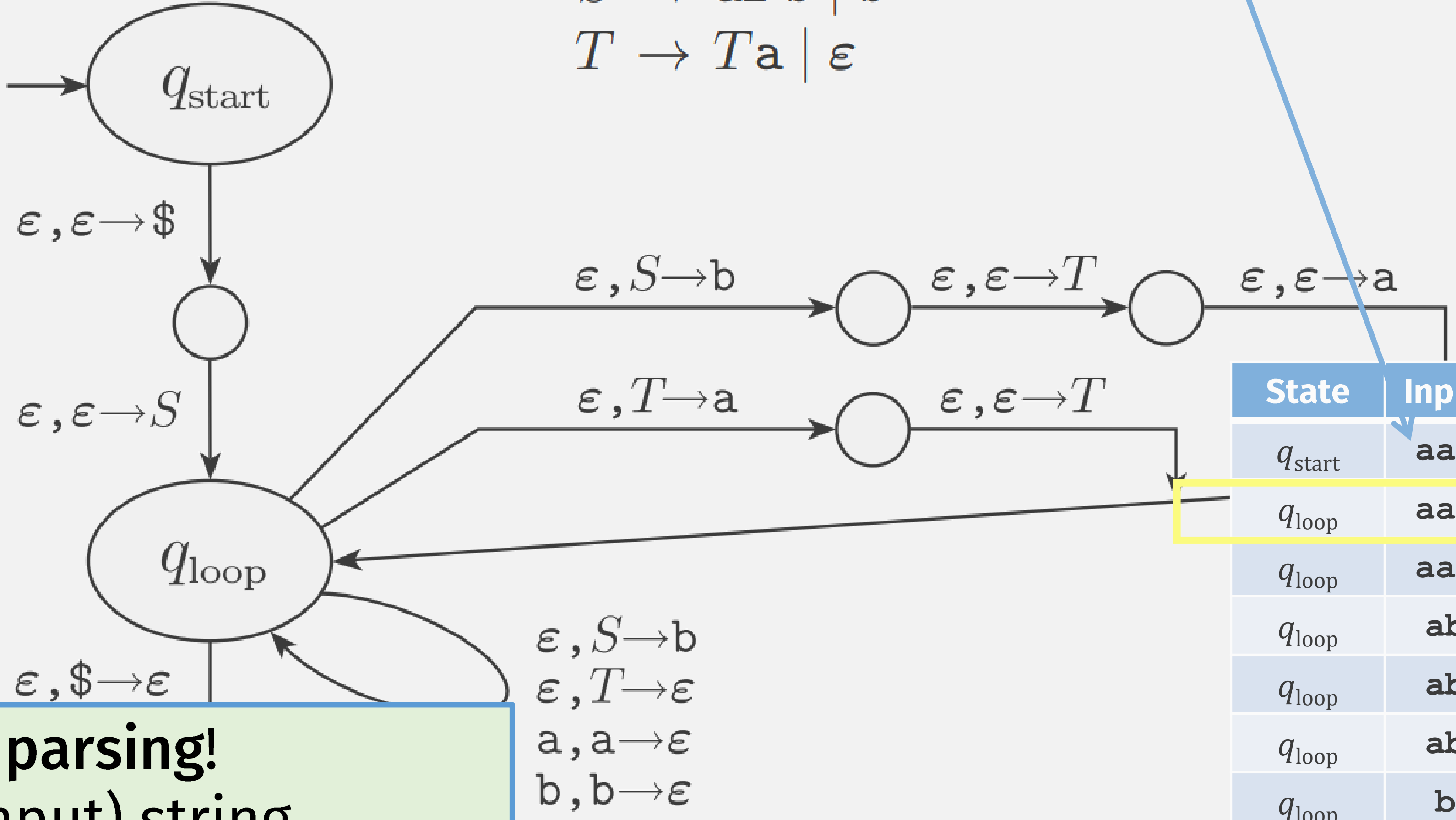
Example Derivation using CFG:

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

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

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

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



PDA Example

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

This Machine is **parsing!**

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

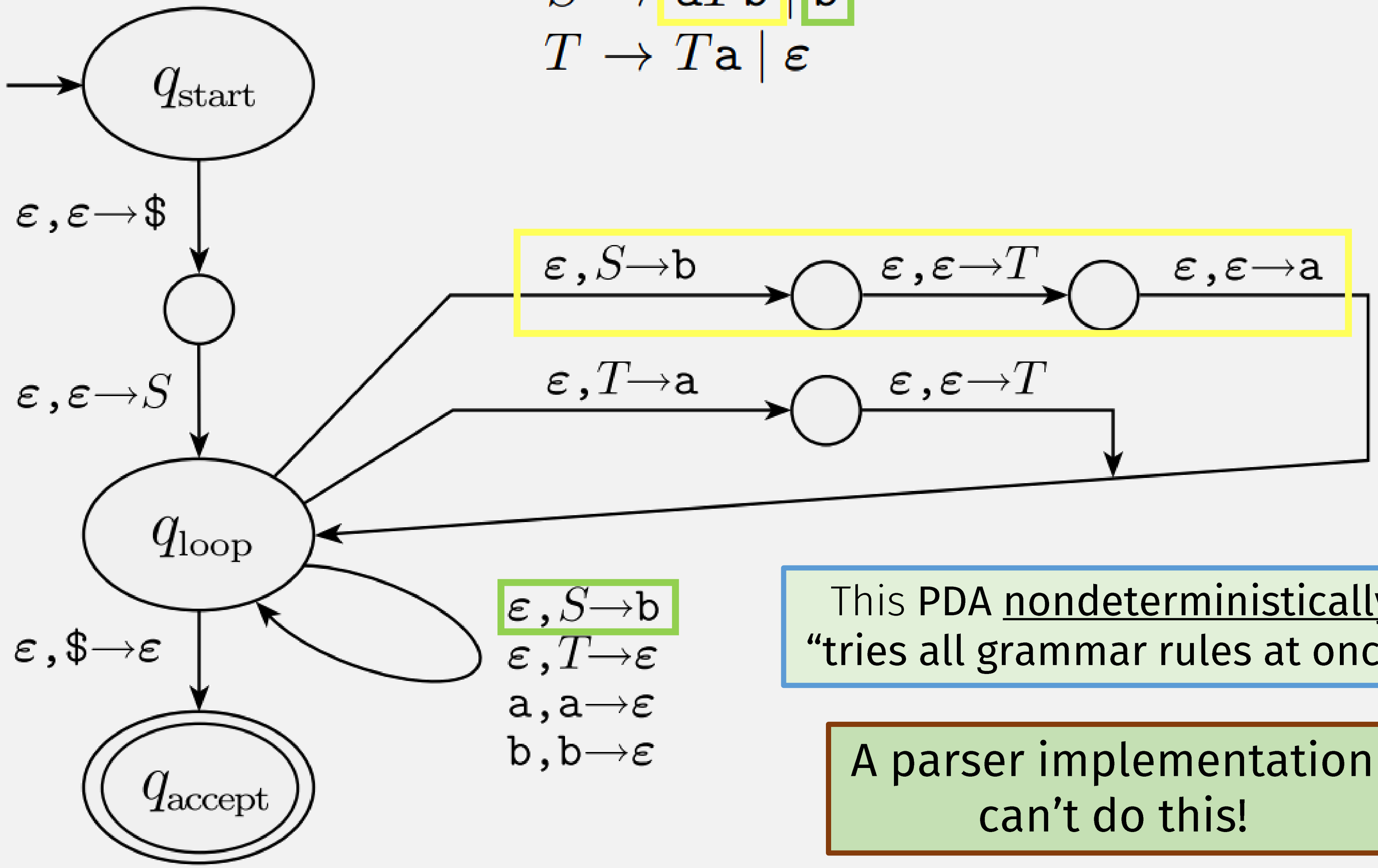
Generating vs Parsing

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

Previously: (Nondeterministic) PDA

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



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. $\delta\colon Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow (Q \times \Gamma_\varepsilon) \cup \{\emptyset\}$ 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 6-tuple

1. Q is the set of states,
2. Σ is the input alphabet,
3. Γ is the stack alphabet,
4. $\delta\colon Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$
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!

- 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!)

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

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

used S rule

aaabbb ⟼ aaSbb

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

used T rule

aaabbbbbb ⟼ aaTbbbb

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

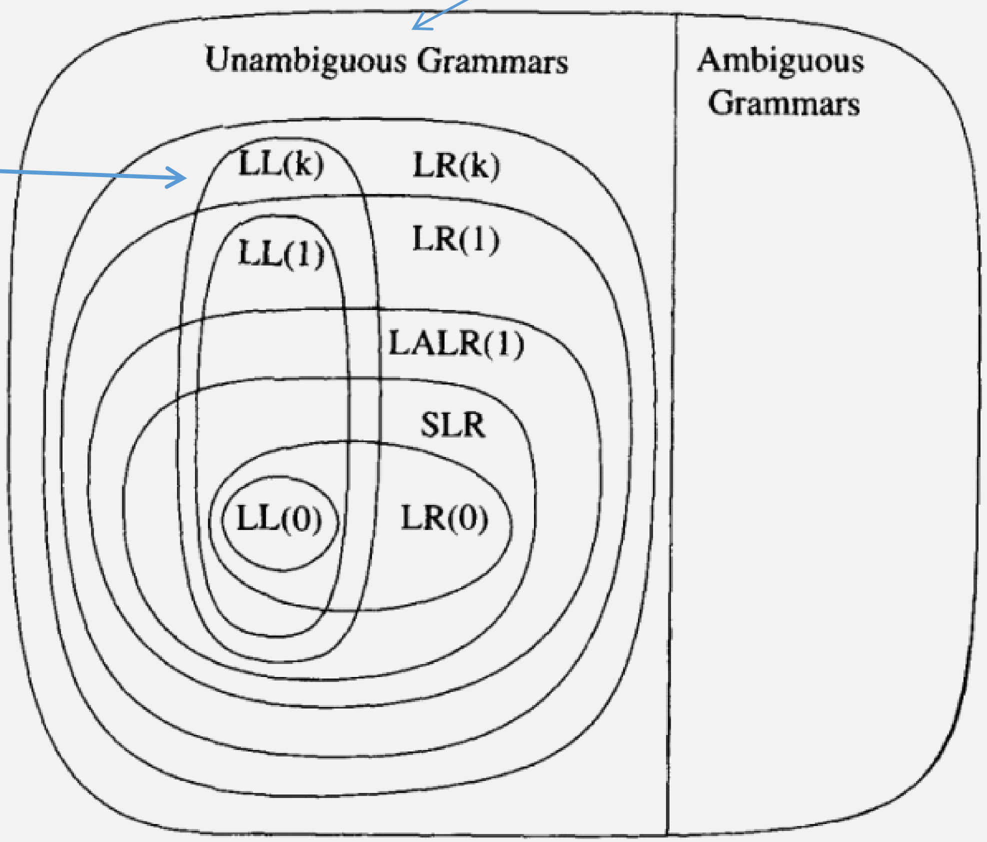
PDAs recognize CFLs, but **DPDAs only recognize DCFLs!** (a subset of CFLs)

Subclasses of CFLs

Umambiguous CFLs / PDAs

DCFLs

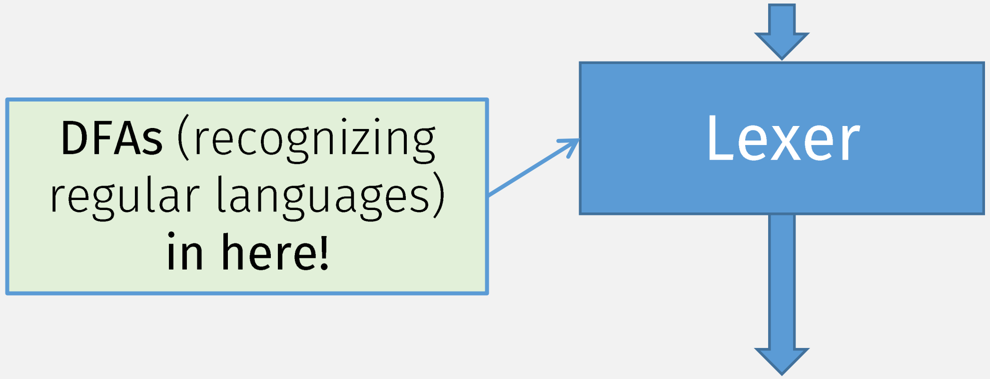
Programming language parsers / compilers are ideally in here



All CFLS

Compiler Stages

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



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

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**)!

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

It's more than an analogy!

<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 !**

Compiler Stages

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

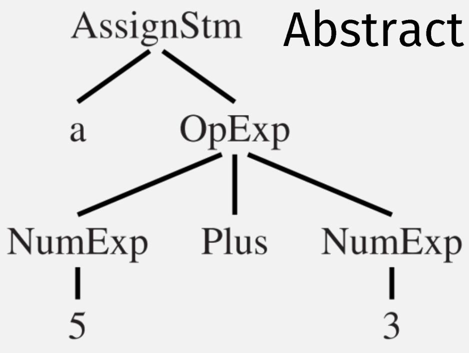
Lexer

DFAs (recognizing regular languages) in here!

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

Parser

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!

<u>This</u> CFG <u>is</u> 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

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

- **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

$\mathrm{aaabbb} \rightarrowtail \mathrm{aa}S\mathrm{bb}$

aaa

Should use T rule

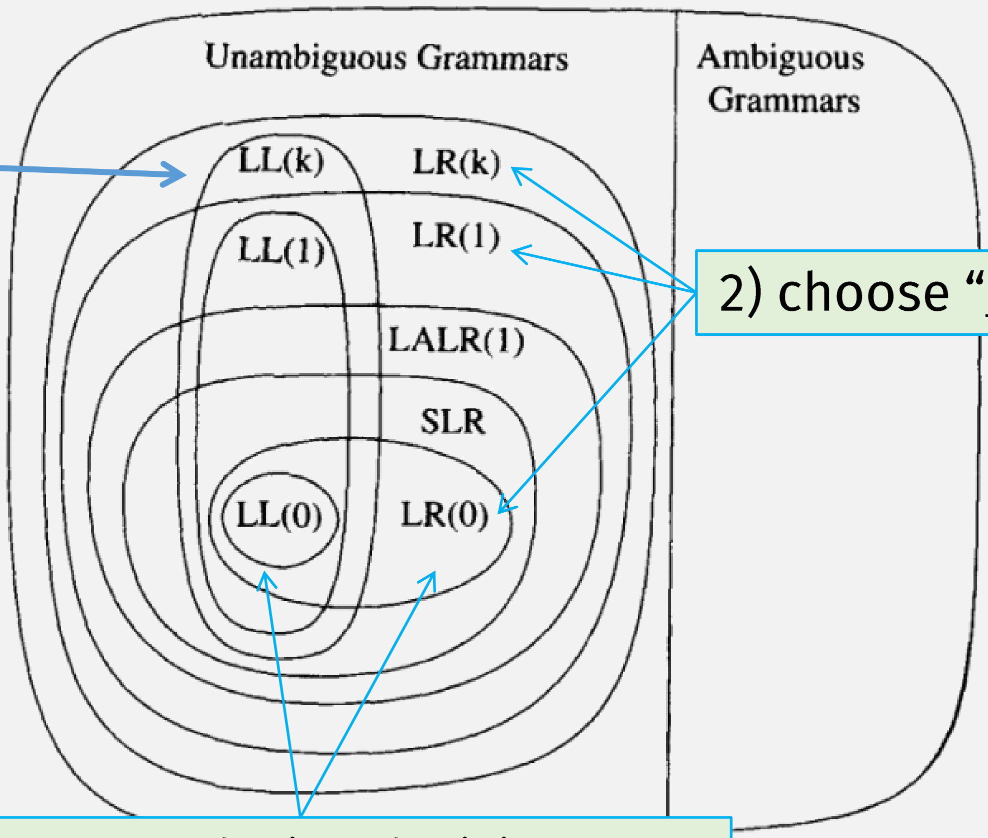
$\mathrm{aaabbbbbb} \rightarrowtail \mathrm{aa}T\mathrm{bbbb}$

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

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

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

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

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

4 $L \rightarrow \text{end}$

5 $L \rightarrow \text{; } S\ L$

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
```

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 (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

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

```
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

$$S \rightarrow S \, ; \, S \qquad E \rightarrow \text{id}$$

$$S \rightarrow \text{id} \, := \, E \qquad E \rightarrow \text{num}$$

$$S \rightarrow \text{print} \, (\, L \,) \qquad E \rightarrow E \, + \, E$$

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*

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. $E \rightarrow \text{num}$
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* $S \rightarrow$ id:=E

Can determine (rightmost) rule

LR parsing

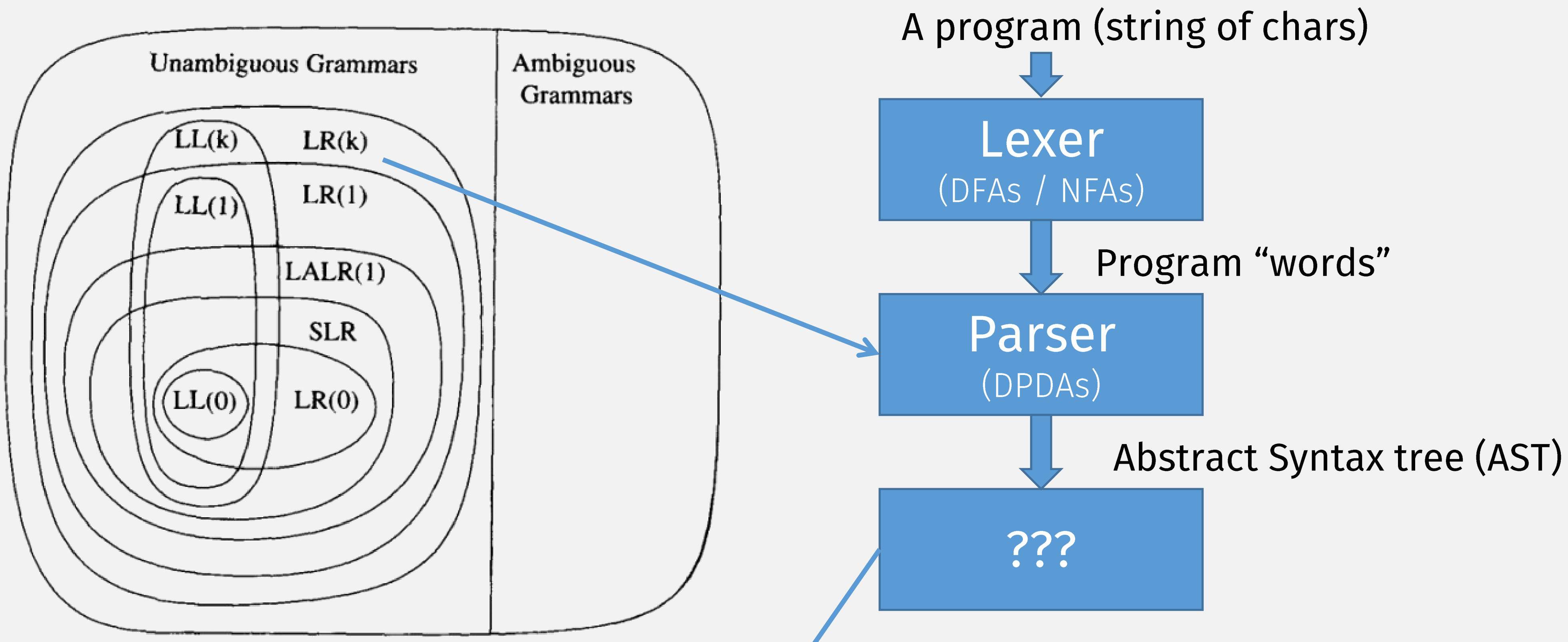
- **L** = left-to-right
- **R** = rightmost derivation

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

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* $E \rightarrow$ 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*

LR Parsers also called "Shift-Reduce" Parsers

To learn more, take a Compilers Class!

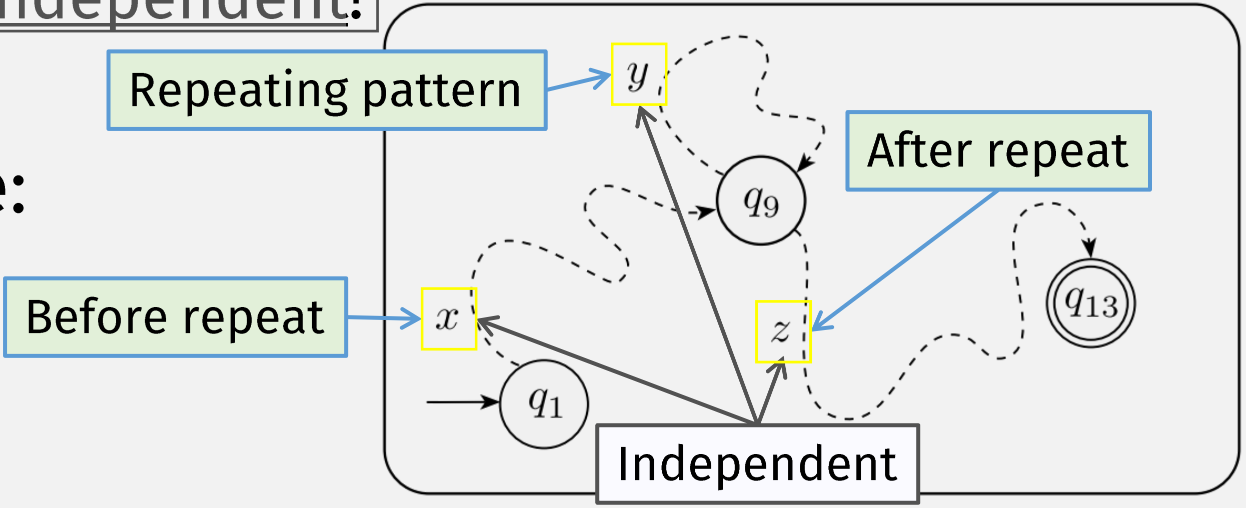


This phase needs computation that goes beyond CFLs

Flashback: Pumping Lemma for Regular Langs

- **Pumping Lemma** describes how strings repeat

- **Regular language** strings repeat using **Kleene star** operation
  - Key: 3 substrings $x\,y\,z$ independent!



- A non-regular language:

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

Kleene star can't express this pattern:
2nd part depends on (length of) 1st part

- Q: How do CFLs repeat?

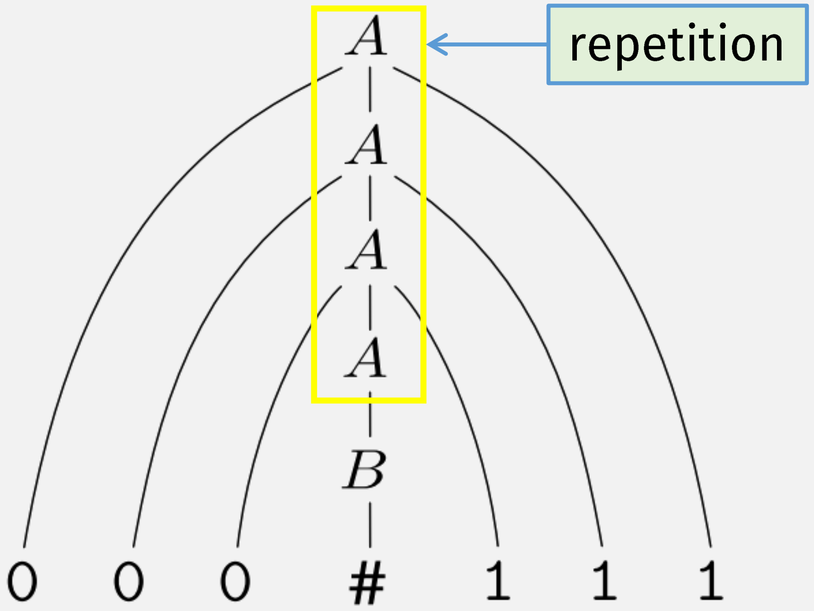
Repetition and Dependency in CFLs

Parts before/after repetition point <u>linked</u> (not independent)

Repetition

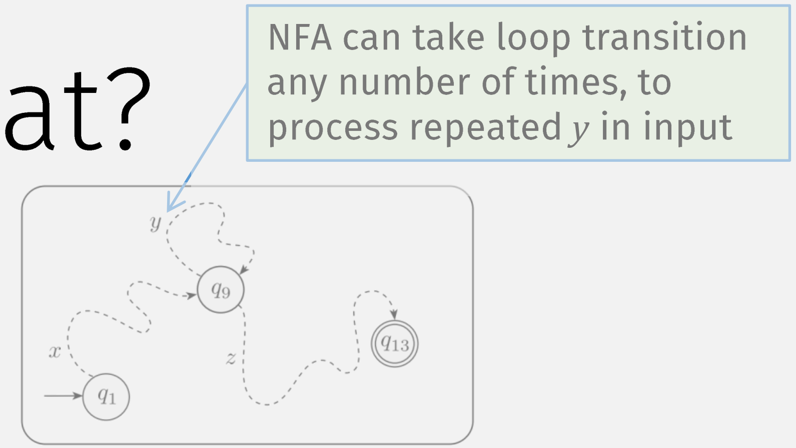
$$A \to 0A1$$
$$A \to B$$
$$B \to \#$$

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

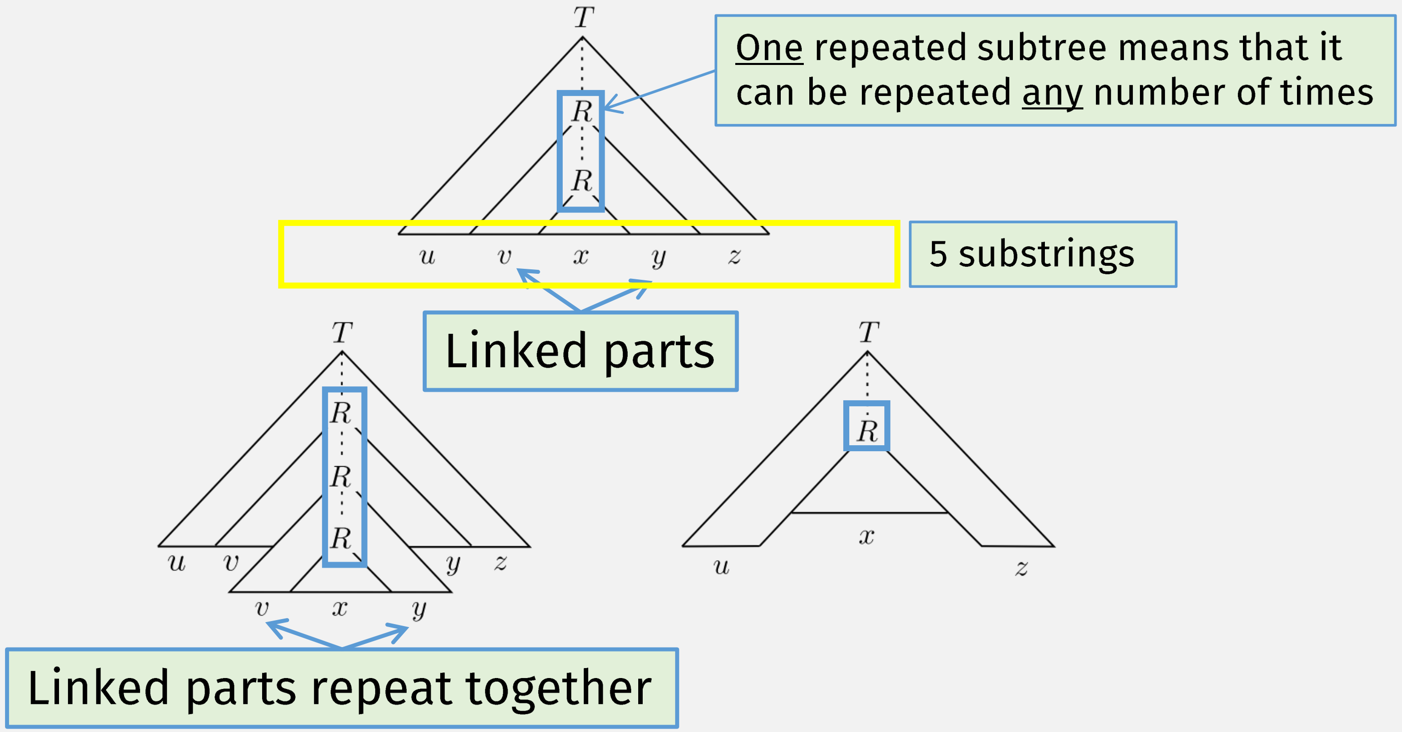


$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000B111 \Rightarrow 000\#111$$

How Do Strings in CFLs Repeat?



- Strings in regular languages <u>repeat states</u>
- Strings in CFLs <u>repeat subtrees</u> in the parse tree

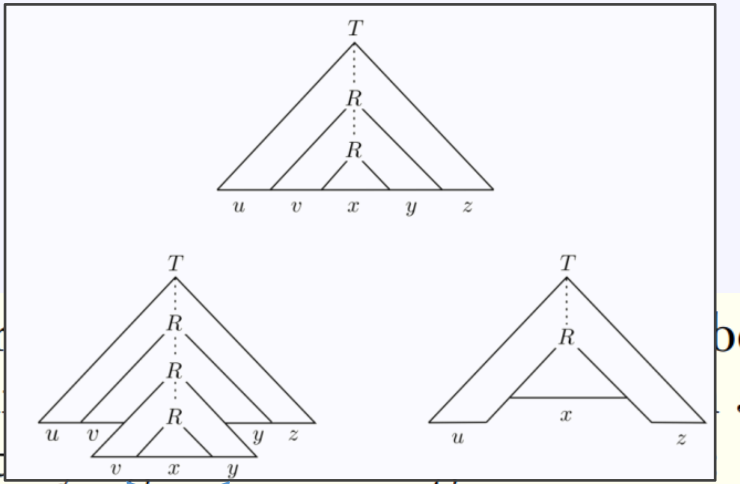


Pumping Lemma for CFLS

Pumping lemma for context-free languages If A is a context-free language, then there is a number p (the pumping length) where, if s is any string in A of length at least p, then s may be divided into five pieces $s = uvxyz$ satisfying the conditions

Two pumpable parts.
But they must be pumped together!

1. for each $i \geq 0$, $uv^ixy^iz \in A$,
2. $|vy| > 0$, and
3. $|vxy| \leq p$.



Pumping lemma If A is a regular ... ber p (the pumping length) where if s is any stri ... s may be divided into three pieces, $s = xyz$, sat ...

1. for each $i \geq 0$, $xy^iz \in A$,
2. $|y| > 0$, and
3. $|xy| \leq p$.

One pumpable part

Two pumpable parts, pumped together

Previously

A Non CFL example

$$\text{language } B = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\} \text{ is not context free}$$

Intuition

- **Strings in CFLs** can have **two parts** that are **"pumped" together**
- **Language** B **requires three parts** to be **"pumped" together**
- So **it's not a CFL!**

Proof?

Want to prove: $a^n b^n c^n$ **is not** a CFL

Pumping lemma for context-free languages If A is a context-free language, then there is a number p (the pumping length) where, if s is any string in A of length at least p, then s may be divided into five pieces $s = uvxyz$ satisfying the conditions

1. for each $i \geq 0$, $uv^ixy^iz \in A$,
2. $|vy| > 0$, and
3. $|vxy| \leq p$.

Reminder: CFL Pumping lemma says: all strings $a^n b^n c^n \geq$ length p are splittable into $uvxyz$ where v and y are pumpable

Proof (by contradiction):

Now we must find a contradiction …

- Assume: $a^n b^n c^n$ **is** a CFL
  - So it must satisfy the pumping lemma for CFLs
  - I.e., all strings $\geq$ length p are pumpable
- Counterexample = $a^p b^p c^p$

Contradiction if:
- A string in the language ☑
- $\geq$ length p ☑
- Is **not splittable** into $uvxyz$ where v and y are pumpable ???

p as p bs p bs

a … b … c …

Want to prove: $a^n b^n c^n$ **is not** a CFL

Pumping lemma for context-free languages If A is a context-free language, then there is a number p (the pumping length) where, if s is any string in A of length at least p, then s may be divided into five pieces $s = uvxyz$ satisfying the conditions

1. for each $i \geq 0$, $uv^ixy^iz \in A$,
2. $|vy| > 0$, and
3. $|vxy| \leq p$.

Possible Splits

Proof (by contradiction):

contradiction

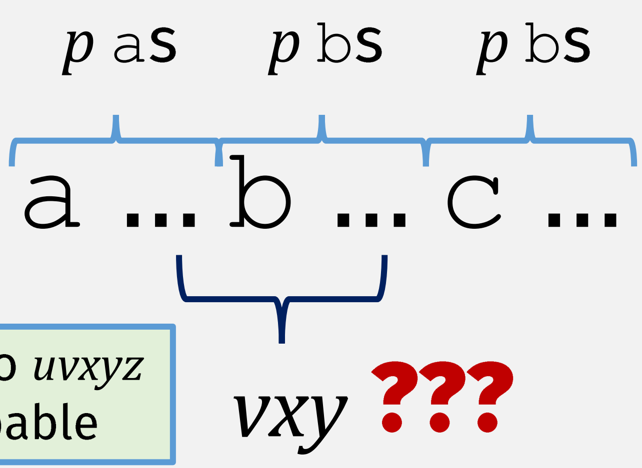
- Assume: $a^n b^n c^n$ **is** a CFL
  - So it must satisfy the pumping lemma for CFLs
  - I.e., all strings ≥ length p are pumpable
- Counterexample = $a^p b^p c^p$

Contradiction if:
- A string in the language
- ≥ length p
- Is **not splittable** into $uvxyz$ where v and y are pumpable

- Possible Splits (using condition # 3: $|vxy| \leq p$)

Not pumpable

- ☒ vxy is all as
- ☒ vxy is all bs
- ☒ vxy is all cs
- ☒ vxy has as and bs
- ☒ vxy has bs and cs
- (vxy cannot have as, bs, and cs)



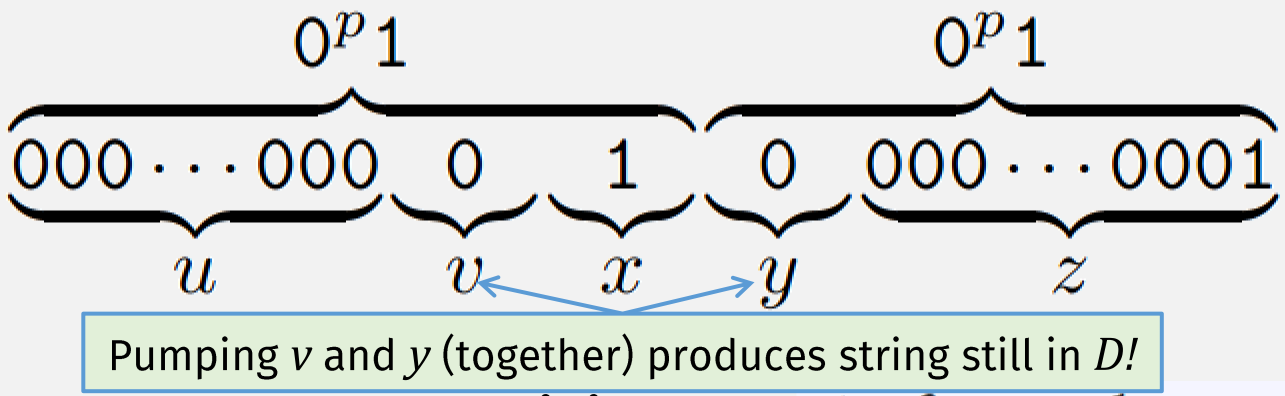
$a^p b^p c^p$ **cannot be split** into $uvxyz$ where v and y are pumpable

So $a^n b^n c^n$ **is not a CFL**

Another Non-CFL $D = \{ww \mid w \in \{0,1\}^*\}$

Be careful when choosing **counterexample** *s*: 0^p10^p1

This *s* can be pumped according to CFL pumping lemma:



- **CFL Pumping Lemma conditions:**
  - ☑ **1.** for each $i \geq 0$, $uv^ixy^iz \in A$,
  - ☑ **2.** $|vy| > 0$, and
  - ☑ **3.** $|vxy| \leq p$.

So **this attempt** to **prove** that the **language** is **not a CFL failed**.
(It doesn't prove that the language is a CFL!)

Another Non-CFL $D = \{ww \mid w \in \{0,1\}^*\}$

- Need another counterexample string *s*:

If *vyx* is contained in first or second half, then any pumping will break the match

$0^p 1^p 0^p 1^p$

So *vyx* must straddle the middle

But any pumping still breaks the match because order is wrong

- CFL Pumping Lemma conditions:

1. for each $i \geq 0$, $uv^ixy^iz \in A$,

2. $|vy| > 0$, and

3. $|vxy| \leq p$.

Now we have proven that this language is **not a CFL!**

A Practical Non-CFL

- **XML**
  - ELEMENT → <TAG>CONTENT</TAG>
  - Where TAG is any string

- XML also looks like this <u>non-CFL</u>: $D = \{ww \mid w \in \{0,1\}^*\}$

- This means XML is <u>not context-free</u>!
  - <u>Note</u>: HTML *is* context-free because …
  - … there are only a <u>finite</u> number of tags,
  - so they can be embedded into a <u>finite</u> number of rules.

<u>In practice</u>:

- XML is <u>parsed</u> as a CFL, with a CFG
- Then matching tags checked in a 2^{nd} pass with a <u>more powerful machine</u> …

Next: A More Powerful Machine …

M_1 accepts its input if it is in language: $B = \{w\#w \mid w \in \{0,1\}^*\}$

M_1 = "On input string w:

1. Zig-zag across the tape to corresponding positions on either side of the # symbol to check whether these positions contain the same symbol. If they do not, or if no # is found, *reject*. Cross off symbols as they are checked to keep track of which symbols correspond.

Infinite memory (initial contents are the **input string**)

Can move to, and **read/write from** <u>arbitrary</u> **memory locations!**