CS 341: Foundations of CS II

Marvin K. Nakayama
Computer Science Department
New Jersey Institute of Technology
Newark, NJ 07102

Chapter 2
Context-Free Languages

Contents

Context-Free Languages (CFLs)

- Consider language $\{ 0^n1^n \mid n \geq 0 \}$, which is nonregular.
- Start variable S with "substitution rules":

$$S \to 0S1$$
$$S \to \varepsilon$$

- Rules can **yield** string 0^k1^k by
  - applying rule "$S \to 0S1$" k times,
  - followed by rule "$S \to \varepsilon$" once.
- **Derivation** of string 0^31^3

$$S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 000S111 \Rightarrow 000\varepsilon111 = 000111$$

Definition of CFG

Definition: Context-free grammar (CFG) $G = (V, \Sigma, R, S)$ where

1. V is finite set of **variables** (AKA **nonterminals**)
2. Σ is finite set of **terminals** (with $V \cap \Sigma = \emptyset$)
3. R is finite set of substitution **rules** (AKA **productions**), each of the form
$$L \to X,$$
where
   - $L \in V$
   - $X \in (V \cup \Sigma)^*$
4. S is **start variable**, where $S \in V$

Example of CFG

Example: Language $\{ 0^n 1^n \mid n \geq 0 \}$ has CFG $G = (V, \Sigma, R, S)$

- Variables $V = \{S\}$
- Terminals $\Sigma = \{0, 1\}$
- Start variable S
- Rules R:

$$S \to 0S1$$
$$S \to \varepsilon$$

- Combine rules with same left-hand side in **Backus-Naur** (or **Backus Normal**) **Form (BNF)**:

$$S \to 0S1 \mid \varepsilon$$

Deriving Strings Using CFG

Definition: If

- $u, v, w \in (V \cup \Sigma)^*$, and
- $A \to w$ is a rule of the grammar,

then uAv **yields** uwv, written

$$uAv \;\Rightarrow\; uwv$$

Remark:

- A single-step derivation "$\Rightarrow$" consists of substituting a variable by a string of variables and terminals according to a substitution rule.

Example: With the rule "$A \to BC$", we can have

$$01AD0 \;\Rightarrow\; 01BCD0.$$

Language of CFG

Definition: u **derives** v, written $u \stackrel{*}{\Rightarrow} v$, if

- $u = v$, or
- $\exists\ u_1, u_2, \ldots, u_k$ for some $k \geq 0$ such that

$$u \;\Rightarrow\; u_1 \;\Rightarrow\; u_2 \;\Rightarrow\; \cdots \;\Rightarrow\; u_k \;\Rightarrow\; v$$

Remark: "$\stackrel{*}{\Rightarrow}$" denotes a sequence of ≥ 0 single-step derivations.

Example: With the rules "$A \to B1 \mid D0C$",

$$0AA \;\stackrel{*}{\Rightarrow}\; 0D0CB1$$

Definition: The **language** of CFG $G = (V, \Sigma, R, S)$ is

$$L(G) = \{ w \in \Sigma^* \mid S \stackrel{*}{\Rightarrow} w \}.$$

Such a language is called **context-free**, and satisfies $L(G) \subseteq \Sigma^*$.

Example of CFG

- CFG $G = (V, \Sigma, R, S)$ with
  1. $V = \{S\}$
  2. $\Sigma = \{0, 1\}$
  3. Rules R:

$$S \to 0S \mid \varepsilon$$

- Then $L(G) = \{ 0^n \mid n \geq 0 \}$.
- For example, S derives 0^3

$$S \Rightarrow 0S \;\Rightarrow\; 00S \;\Rightarrow\; 000S \;\Rightarrow\; 000\varepsilon \;=\; 000$$

- Note that $\to$ and $\Rightarrow$ are different.
  - $\to$ used in defining rules
  - $\Rightarrow$ used in derivation

Example of CFG

- CFG $G = (V, \Sigma, R, S)$ with
  1. $V = \{S\}$
  2. $\Sigma = \{0, 1\}$
  3. Rules R:

$$S \to 0S \mid 1S \mid \varepsilon$$

- Then $L(G) = \Sigma^*$.
- For example, S derives 0100

$$S \Rightarrow 0S \Rightarrow 01S \Rightarrow 010S \Rightarrow 0100S \Rightarrow 0100$$

Example of CFG

- CFG $G = (V, \Sigma, R, S)$ with
  1. $V = \{S\}$
  2. $\Sigma = \{0, 1\}$
  3. Rules R:

$$S \to 0S \mid 1S \mid 1$$

- Then $L(G) = \{ w \in \Sigma^* \mid w = s1 \text{ for some } s \in \Sigma^* \}$, i.e., strings that end in 1.
- For example, S derives 011

$$S \Rightarrow 0S \Rightarrow 01S \Rightarrow 011$$

Example of CFG

- CFG $G = (V, \Sigma, R, S)$ with
  1. $V = \{S, Z\}$
  2. $\Sigma = \{0, 1\}$
  3. Rules R:

$$\begin{aligned} S &\to 0S1 \mid Z \\ Z &\to 0Z \mid \varepsilon \end{aligned}$$

- Then $L(G) = \{ 0^i 1^j \mid i \geq j \}$.
- For example, S derives $0^5\,1^3$

$$\begin{aligned} S &\Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 000S111 \Rightarrow 000\,Z\,111 \\ &\Rightarrow 000\,0Z\,111 \Rightarrow 000\,00Z\,111 \Rightarrow 000\,00\varepsilon\,111 \\ &= 00000111 \end{aligned}$$

CFG for Palindrome

- PALINDROME $= \{ w \in \Sigma^* \mid w = w^{\mathcal{R}} \}$, where $\Sigma = \{a, b\}$.
- CFG $G = (V, \Sigma, R, S)$ with
  1. $V = \{S\}$
  2. $\Sigma = \{a, b\}$
  3. Rules R:

$$S \to aSa \mid bSb \mid a \mid b \mid \varepsilon$$

- Then $L(G) =$ PALINDROME
- S derives $bbaabb$

$$S \Rightarrow bSb \Rightarrow bbSbb \Rightarrow bbaSabb \Rightarrow bba\varepsilon abb = bbaabb$$

- S derives $aabaa$

$$S \Rightarrow aSa \Rightarrow aaSaa \Rightarrow aabaa$$

CFG for EVEN-EVEN

- Recall language EVEN-EVEN is the set of strings over $\Sigma = \{a, b\}$ with even number of a's and even number of b's.
- EVEN-EVEN has regular expression

$$\big(\ aa\ \cup\ bb\ \cup\ (ab \cup ba)(aa \cup bb)^*(ab \cup ba)\ \big)^*$$

- CFG $G = (V, \Sigma, R, S)$ with
  1. $V = \{S, X, Y\}$
  2. $\Sigma = \{a, b\}$
  3. Rules R:

$$\begin{aligned} S &\to aaS \mid bbS \mid XYXS \mid \varepsilon \\ X &\to ab \mid ba \\ Y &\to aaY \mid bbY \mid \varepsilon \end{aligned}$$

- Then $L(G) =$ EVEN-EVEN

CFG for Simple Arithmetic Expressions

- CFG $G = (V, \Sigma, R, S)$ with
  1. $V = \{S\}$
  2. $\Sigma = \{ +, -, \times, /, (,), 0, 1, 2, \dots, 9 \}$
  3. Rules R:

$$S \to S + S \mid S - S \mid S \times S \mid S/S \mid (S) \mid -S \mid 0 \mid 1 \mid \cdots \mid 9$$

- $L(G)$ is a set of valid arithmetic expressions over single-digit integers.
- S derives string $2 \times (3 + 4)$

$$\begin{aligned} S &\Rightarrow S \times S \Rightarrow S \times (S) \Rightarrow S \times (S + S) \\ &\Rightarrow 2 \times (S + S) \Rightarrow 2 \times (3 + S) \Rightarrow 2 \times (3 + 4) \end{aligned}$$

Derivation Tree

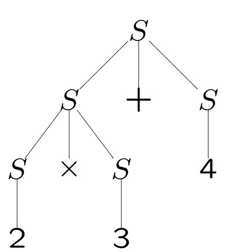
- CFG

$$S \to S + S \mid S - S \mid S \times S \mid S/S \mid (S) \mid -S \mid 0 \mid 1 \mid \cdots \mid 9$$

- Can generate string $2 \times 3 + 4$ using derivation

$$\begin{aligned} S &\Rightarrow S + S \Rightarrow S \times S + S \Rightarrow 2 \times S + S \\ &\Rightarrow 2 \times 3 + S \Rightarrow 2 \times 3 + 4 \end{aligned}$$

  - **Leftmost derivation**: leftmost variable replaced in each step.
- Corresponding **derivation** (or **parse**) **tree**



- **Depth-first** traversal of tree
  - Starting at **root**, walk around tree with left hand always touching tree.
  - string = sequence of **leaves** visited.

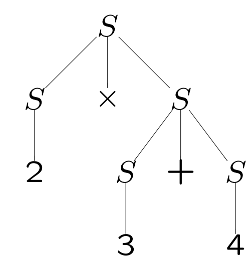
Ambiguous CFG

$$S \to S + S \mid S - S \mid S \times S \mid S/S \mid (S) \mid -S \mid 0 \mid 1 \mid \cdots \mid 9$$

- Another derivation of string $2 \times 3 + 4$:

$$\begin{aligned} S &\Rightarrow S \times S \Rightarrow S \times S + S \Rightarrow 2 \times S + S \\ &\Rightarrow 2 \times 3 + S \Rightarrow 2 \times 3 + 4 \end{aligned}$$

  which is **not a leftmost derivation**.
- Corresponding derivation tree:



Definition: CFG G is **ambiguous** if $\exists$ string $w \in L(G)$ having different parse trees (or equivalently, different leftmost derivations).

Applications of CFLs

- Model for natural languages (Noam Chomsky)

$$\begin{aligned}\langle\text{SENTENCE}\rangle &\to \langle\text{NOUN-PHRASE}\rangle\langle\text{VERB-PHRASE}\rangle\\ \langle\text{NOUN-PHRASE}\rangle &\to \langle\text{ARTICLE}\rangle\langle\text{NOUN}\rangle \mid \langle\text{ARTICLE}\rangle\langle\text{ADJ}\rangle\langle\text{NOUN}\rangle\\ \langle\text{VERB-PHRASE}\rangle &\to \langle\text{VERB}\rangle \mid \langle\text{VERB}\rangle\langle\text{NOUN-PHRASE}\rangle\\ \langle\text{ARTICLE}\rangle &\to \text{a} \mid \text{the}\\ \langle\text{NOUN}\rangle &\to \text{girl} \mid \text{boy} \mid \text{cat}\\ \langle\text{ADJ}\rangle &\to \text{big} \mid \text{small} \mid \text{blue}\\ \langle\text{VERB}\rangle &\to \text{sees} \mid \text{likes}\end{aligned}$$

Using above CFG, which has ⟨SENTENCE⟩ as start variable, can derive

$$\begin{aligned}\langle\text{SENTENCE}\rangle &\Rightarrow \langle\text{NOUN-PHRASE}\rangle\langle\text{VERB-PHRASE}\rangle\\ &\Rightarrow \langle\text{ARTICLE}\rangle\langle\text{NOUN}\rangle\langle\text{VERB-PHRASE}\rangle\\ &\Rightarrow \langle\text{ARTICLE}\rangle\langle\text{NOUN}\rangle\langle\text{VERB}\rangle\langle\text{NOUN-PHRASE}\rangle\\ &\Rightarrow \langle\text{ARTICLE}\rangle\langle\text{NOUN}\rangle\langle\text{VERB}\rangle\langle\text{ARTICLE}\rangle\langle\text{ADJ}\rangle\langle\text{NOUN}\rangle\\ &\overset{*}{\Rightarrow} \text{the girl sees a blue cat}\end{aligned}$$

Applications of CFLs

- Specification of programming languages:
  - parsing a computer program
- Describes mathematical structures, etc.
- Intermediate class between
  - regular languages (Chapter 1) and
  - computable languages (Chapters 3 and 4)

Context-Free Languages

Definition: Any language that can be generated by CFG is a **context-free language (CFL)**.

Remark: The CFL $\{ 0^n 1^n \mid n \geq 0 \}$ shows us that certain CFLs are nonregular.

Questions:

1. Are all regular languages context-free?
2. Are all languages context-free?

Chomsky Normal Form

Definition: CFG $G = (V, \Sigma, R, S)$ is in **Chomsky normal form** if each rule is in one of three forms:

$$\begin{aligned} & A \to BC \\ \text{or} \quad & A \to x \\ \text{or} \quad & S \to \varepsilon \end{aligned}$$

with

- variables $A \in V$ and $B, C \in V - \{S\}$, and
- terminal $x \in \Sigma$

Example: Rules of CFG in Chomsky normal form with $V = \{S, W, X\}$, $\Sigma = \{a, b\}$:

$$\begin{aligned} S &\to XX \mid XW \mid a \mid \varepsilon \\ X &\to WX \mid b \\ W &\to a \end{aligned}$$

Remark: Grammars in Chomsky normal form are far easier to analyze.

Can Always Put CFG into Chomsky Normal Form

Recall: CFG in Chomsky normal form if each rule has form:

$$A \to BC \quad \text{or} \quad A \to x \quad \text{or} \quad S \to \varepsilon$$

where $A \in V$; $B, C \in V - \{S\}$; $x \in \Sigma$.

Theorem 2.9

Every CFL can be described by a CFG in Chomsky normal form.

Proof Idea:

- Start with CFG $G = (V, \Sigma, R, S)$.
- Replace, one-by-one, every rule that is not "Chomsky".
- Need to take care of:
  - Start variable (not allowed on RHS of rules)
  - ε-rules ($A \to \varepsilon$ not allowed when A isn't start variable)
  - all other violating rules ($A \to B$, $A \to aBc$, $A \to BCDE$)

Converting CFG into Chomsky Normal Form

1. Start variable not allowed on RHS of rule, so introduce
   - New start variable S_0
   - New rule $S_0 \to S$
2. Remove ε-rules $A \to \varepsilon$, where $A \in V - \{S\}$.
   - Before: $B \to xAy$ and $A \to \varepsilon \mid \cdots$
   - After: $B \to xAy \mid xy$ and $A \to \cdots$
3. Remove **unit rules** $A \to B$, where $A \in V$.
   - Before: $A \to B$ and $B \to xCy$
   - After: $A \to xCy$ and $B \to xCy$

4. Replace problematic terminals a by variable T_a with rule $T_a \to a$.
   - Before: $A \to ab$
   - After: $A \to T_a T_b$, $T_a \to a$, $T_b \to b$.
5. Shorten long RHS to sequence of RHS's with only 2 variables each:
   - Before: $A \to B_1 B_2 \cdots B_k$
   - After: $A \to B_1 A_1$, $A_1 \to B_2 A_2$, ..., $A_{k-2} \to B_{k-1} B_k$
     - Thus, $A \Rightarrow B_1 A_1 \Rightarrow B_1 B_2 A_2 \Rightarrow \cdots \Rightarrow B_1 B_2 \cdots B_k$
6. Be careful about removing rules:
   - Do not introduce new rules that you removed earlier.
     - **Example:** $A \to A$ simply disappears
   - When removing $A \to \varepsilon$ rules, insert all new replacements:
     - Before: $B \to AbA$ and $A \to \varepsilon \mid \cdots$
     - After: $B \to AbA \mid bA \mid Ab \mid b$ and $A \to \cdots$

Example: Convert CFG into Chomsky Normal Form

Initial CFG G_0:

$$\begin{aligned} S &\to XSX \mid aY \\ X &\to Y \mid S \\ Y &\to b \mid \varepsilon \end{aligned}$$

1. Introduce new start variable S_0 and new rule $S_0 \to S$:

$$\begin{aligned} S_0 &\to S \\ S &\to XSX \mid aY \\ X &\to Y \mid S \\ Y &\to b \mid \varepsilon \end{aligned}$$

Example: Convert CFG into Chomsky Normal Form

From previous slide

$$
\begin{aligned}
S_0 &\to S \\
S &\to XSX \mid aY \\
X &\to Y \mid S \\
Y &\to b \mid \varepsilon
\end{aligned}
$$

2. Remove ε-rules for which left side is not start variable:

(i) remove $Y \to \varepsilon$

$$
\begin{aligned}
S_0 &\to S \\
S &\to XSX \mid aY \mid a \\
X &\to Y \mid S \mid \varepsilon \\
Y &\to b
\end{aligned}
$$

(ii) remove $X \to \varepsilon$

$$
\begin{aligned}
S_0 &\to S \\
S &\to XSX \mid aY \mid a \mid SX \mid XS \mid S \\
X &\to Y \mid S \\
Y &\to b
\end{aligned}
$$

Example: Convert CFG into Chomsky Normal Form

From previous slide

$$
\begin{aligned}
S_0 &\to S \\
S &\to XSX \mid aY \mid a \mid SX \mid XS \mid S \\
X &\to Y \mid S \\
Y &\to b
\end{aligned}
$$

3. Remove unit rules:

(i) remove unit rule $S \to S$

$$
\begin{aligned}
S_0 &\to S \\
S &\to XSX \mid aY \mid a \mid SX \mid XS \\
X &\to Y \mid S \\
Y &\to b
\end{aligned}
$$

Example: Convert CFG into Chomsky Normal Form

From previous slide

$$
\begin{aligned}
S_0 &\to S \\
S &\to XSX \mid aY \mid a \mid SX \mid XS \\
X &\to Y \mid S \\
Y &\to b
\end{aligned}
$$

(ii) remove unit rule $S_0 \to S$

$$
\begin{aligned}
S_0 &\to XSX \mid aY \mid a \mid SX \mid XS \\
S &\to XSX \mid aY \mid a \mid SX \mid XS \\
X &\to Y \mid S \\
Y &\to b
\end{aligned}
$$

Example: Convert CFG into Chomsky Normal Form

From previous slide

$$
\begin{aligned}
S_0 &\to XSX \mid aY \mid a \mid SX \mid XS \\
S &\to XSX \mid aY \mid a \mid SX \mid XS \\
X &\to Y \mid S \\
Y &\to b
\end{aligned}
$$

(iii) remove unit rule $X \to Y$

$$
\begin{aligned}
S_0 &\to XSX \mid aY \mid a \mid SX \mid XS \\
S &\to XSX \mid aY \mid a \mid SX \mid XS \\
X &\to S \mid b \\
Y &\to b
\end{aligned}
$$

Example: Convert CFG into Chomsky Normal Form

From previous slide

$$\begin{aligned} S_0 &\to XSX \mid aY \mid a \mid SX \mid XS \\ S &\to XSX \mid aY \mid a \mid SX \mid XS \\ X &\to S \mid b \\ Y &\to b \end{aligned}$$

(iv) remove unit rule $X \to S$

$$\begin{aligned} S_0 &\to XSX \mid aY \mid a \mid SX \mid XS \\ S &\to XSX \mid aY \mid a \mid SX \mid XS \\ X &\to b \mid XSX \mid aY \mid a \mid SX \mid XS \\ Y &\to b \end{aligned}$$

Example: Convert CFG into Chomsky Normal Form

From previous slide

$$\begin{aligned} S_0 &\to XSX \mid aY \mid a \mid SX \mid XS \\ S &\to XSX \mid aY \mid a \mid SX \mid XS \\ X &\to b \mid XSX \mid aY \mid a \mid SX \mid XS \\ Y &\to b \end{aligned}$$

4. Replace problematic terminals a by variable U with $U \to a$.

$$\begin{aligned} S_0 &\to XSX \mid UY \mid a \mid SX \mid XS \\ S &\to XSX \mid UY \mid a \mid SX \mid XS \\ X &\to b \mid XSX \mid UY \mid a \mid SX \mid XS \\ Y &\to b \\ U &\to a \end{aligned}$$

Example: Convert CFG into Chomsky Normal Form

From previous slide

$$\begin{aligned} S_0 &\to XSX \mid UY \mid a \mid SX \mid XS \\ S &\to XSX \mid UY \mid a \mid SX \mid XS \\ X &\to b \mid XSX \mid UY \mid a \mid SX \mid XS \\ Y &\to b \\ U &\to a \end{aligned}$$

5. Shorten long RHS to sequence of RHS's with only 2 variables each

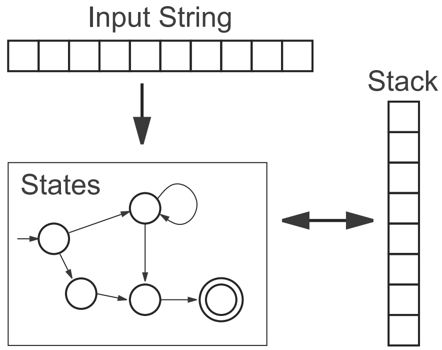
$$\begin{aligned} S_0 &\to XX_1 \mid UY \mid a \mid SX \mid XS \\ S &\to XX_1 \mid UY \mid a \mid SX \mid XS \\ X &\to b \mid XX_1 \mid UY \mid a \mid SX \mid XS \\ Y &\to b \\ U &\to a \\ X_1 &\to SX \end{aligned}$$

which is a CFG in Chomsky normal form.

Pushdown Automata (PDAs)

- Pushdown automata (PDAs) are for CFLs what finite automata are for regular languages.
  - PDA is presented with a string w over an alphabet Σ.
  - PDA accepts or doesn't accept w.
- Key Differences Between PDA and DFA:
  - PDAs have a single stack.
  - PDAs allow for nondeterminism.
- **Defn: Stack** is data structure of unlimited size with 2 operations
  - **push** adds item to top of stack,
  - **pop** removes item from top of stack.

  Last-In-First-Out (LIFO)



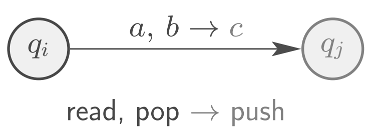
PDA has

- States
- Stack with alphabet Γ
- Transitions among states based on
  - current state
  - what is read from input string
  - what is popped from stack.
- At end of each transition, symbol may be pushed on stack.

PDA Uses Stack

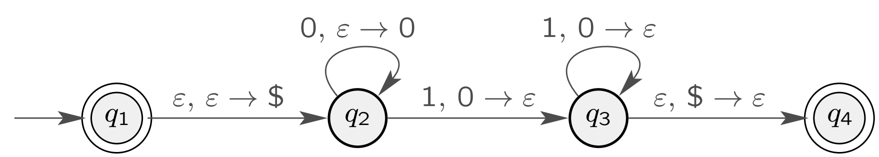
- **General idea:** CFLs are languages that can be recognized by automata that have one stack:
  - $\{ 0^n1^n \mid n \geq 0 \}$ is a CFL
  - $\{ 0^n1^n0^n \mid n \geq 0 \}$ is not a CFL
- Recall for alphabet Σ, we defined $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$.
- Let Γ be **stack alphabet**
  - Symbols in Γ can be pushed onto and popped off stack.
  - Often have $\$ \in \Gamma$ to mark bottom of stack.
- Let $\Gamma_\varepsilon = \Gamma \cup \{\varepsilon\}$.
  - Pushing or popping ε leaves stack unchanged.

PDA Transitions



- If PDA
  - currently in state q_i,
  - reads $a \in \Sigma_\varepsilon$, and
  - pops $b \in \Gamma_\varepsilon$ off the stack,
- then PDA can
  - move to state q_j
  - push $c \in \Gamma_\varepsilon$ onto top of stack
- If $a = \varepsilon$, then no input symbol is read.
- If $b = \varepsilon$, then nothing is popped off stack.
- If $c = \varepsilon$, then nothing is pushed onto stack.

How a PDA Computes

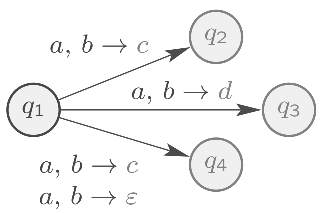


- PDA starts in start state with input string $w \in \Sigma^*$
  - stack initially empty
- PDA makes transitions among states
  - Edge label: "read, pop $\to$ push"
  - Based on current state, what from Σ_ε is next read from w, and what from Γ_ε is popped from stack.
  - Nondeterministically move to state and push from Γ_ε onto stack.
- If possible to end in accept state $\in F \subseteq Q$ after reading entire input w without crashing, then M accepts w.

Definition of PDA

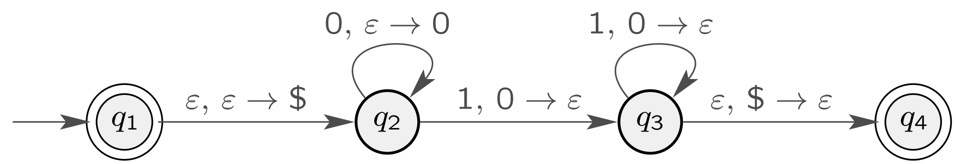
Defn: Pushdown automaton (PDA) $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$:

- Q is finite set of states
- Σ is (finite) input alphabet
- Γ is (finite) stack alphabet
- q_0 is start state, $q_0 \in Q$
- F is set of accept states, $F \subseteq Q$
- $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon)$ is transition function



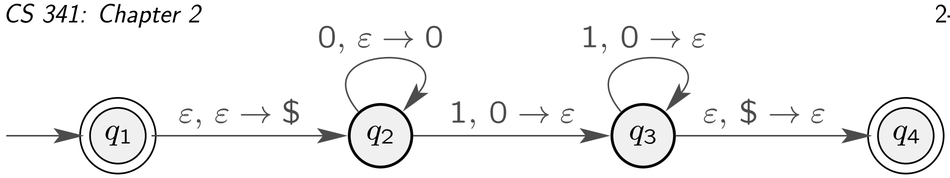
Nondeterministic: multiple choices when in state q_1, read $a \in \Sigma_\varepsilon$, and pop $b \in \Gamma_\varepsilon$;
$\delta(q_1, a, b) = \{\, (q_2, c),\ (q_3, d),\ (q_4, c),\ (q_4, \varepsilon) \,\}$

Example: PDA $M = (Q, \Sigma, \Gamma, \delta, q_1, F)$



- $Q = \{q_1, q_2, q_3, q_4\}$
- $\Sigma = \{0, 1\}$
- $\Gamma = \{0, \$\}$ (use $ to mark bottom of stack)
- q_1 is the start state
- $F = \{q_1, q_4\}$

Will see that M recognizes language $\{\, 0^n 1^n \mid n \geq 0 \,\}$.

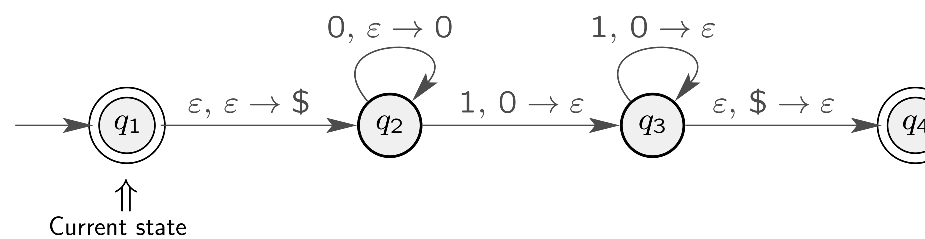


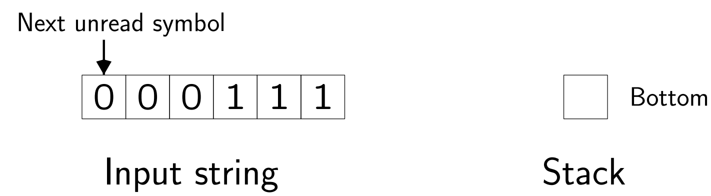
- transition function $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon)$

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

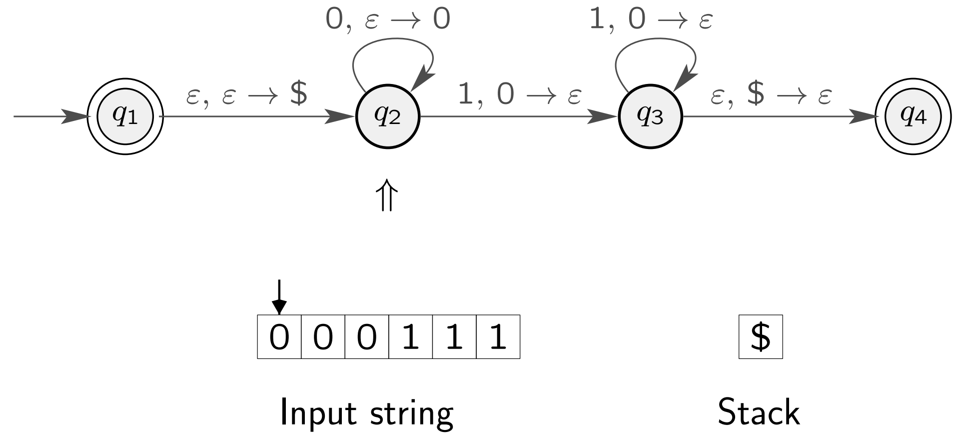
- e.g., $\delta(q_2, 1, 0) = \{\, (q_3, \epsilon) \,\}$.
- Blank entries are $\emptyset$.

- Let's process string 000111 on our PDA.
  - PDA uses stack to match each 0 to a 1.

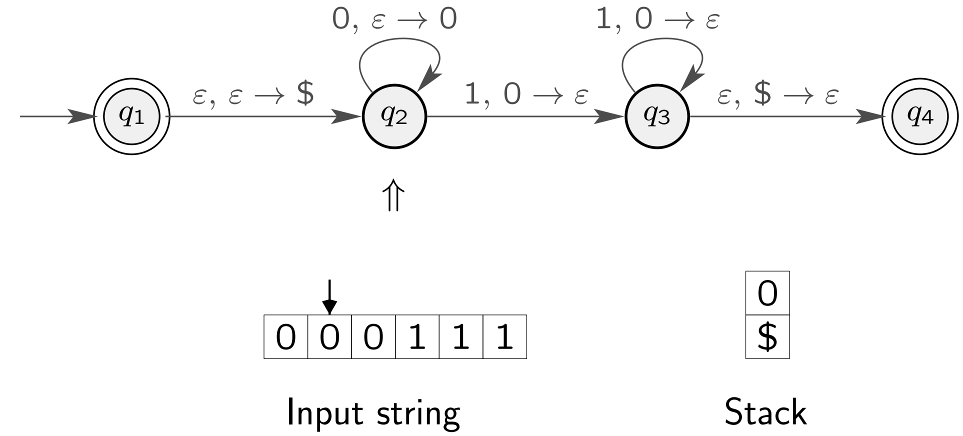




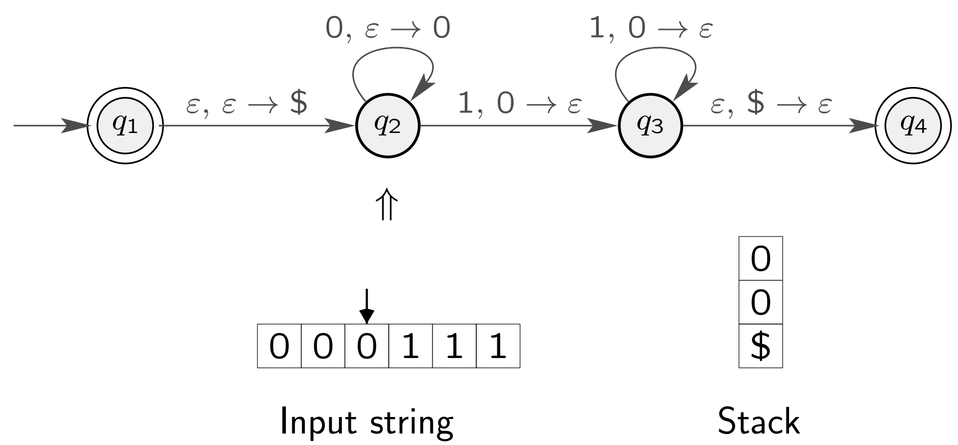
- Start in start state q_1 with stack empty.
- No input symbols read so far.
- Next go to state q_2
  - reading nothing, popping nothing, and pushing $ on stack.



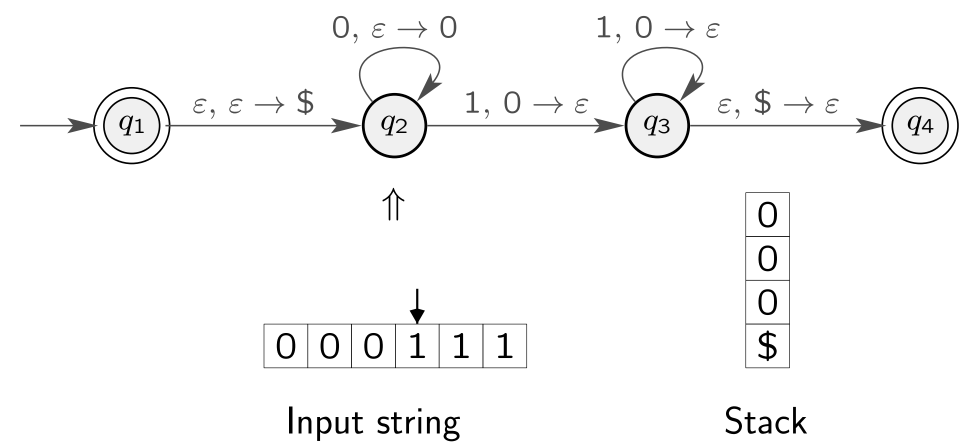
- Next return to state q_2
  - reading input symbol 0
  - popping nothing from stack
  - pushing 0 on stack.



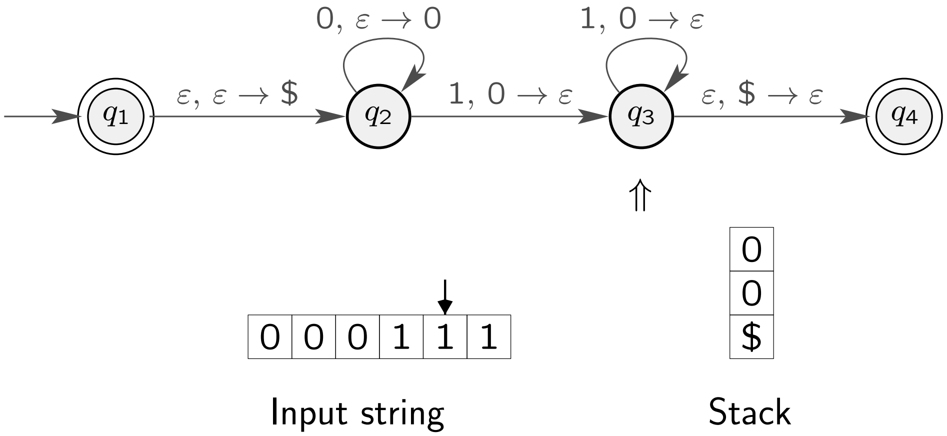
- Next return to state q_2
  - reading input symbol 0
  - popping nothing from stack
  - pushing 0 on stack.



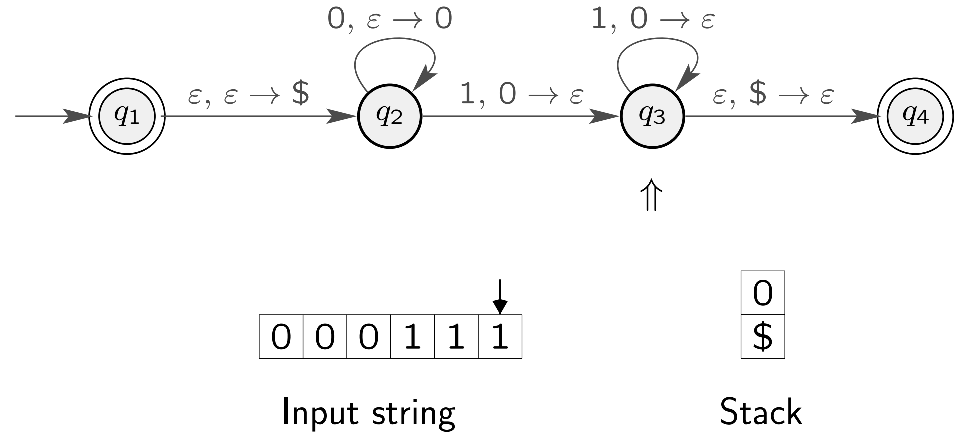
- Next return to state q_2
  - reading input symbol 0
  - popping nothing from stack
  - pushing 0 on stack.



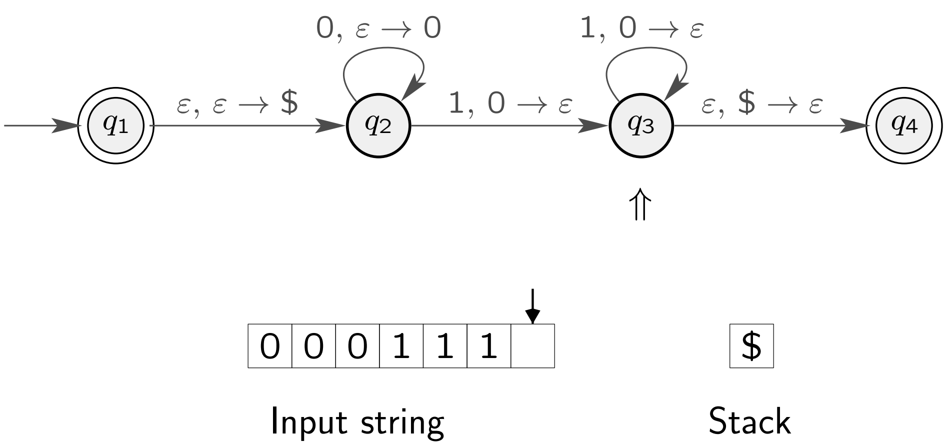
- Next go to state q_3
  - reading input symbol 1
  - popping 0 from stack
  - pushing nothing on stack.



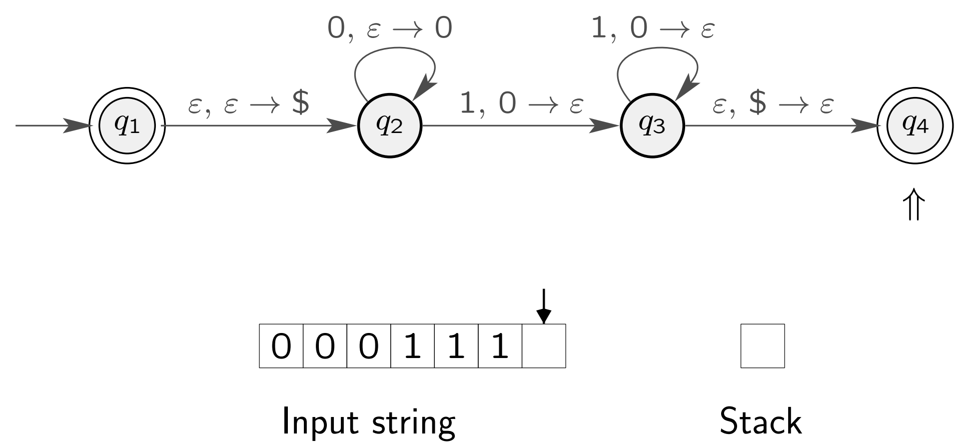
- Next return to state q_3
  - reading input symbol 1
  - popping 0 from stack
  - pushing nothing on stack.



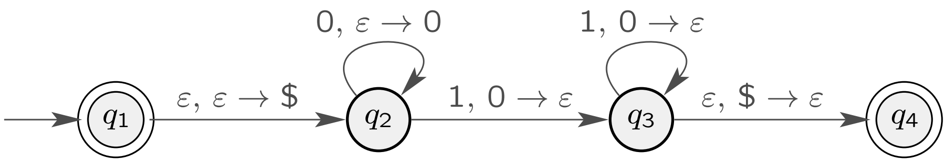
- Next return to state q_3
  - reading input symbol 1
  - popping 0 from stack
  - pushing nothing on stack.



- Next go to state q_4
  - reading nothing
  - popping $ from stack
  - pushing nothing on stack.



- String 000111 is **accepted** by PDA because
  - ended in an accept state q_4, and
  - PDA read the entire input string without crashing.



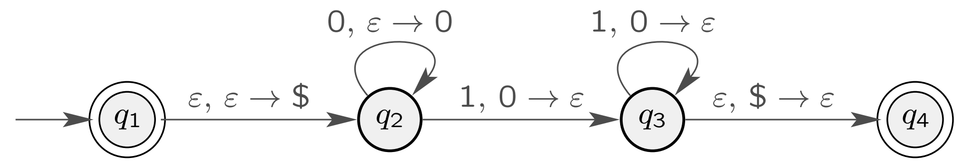
On input $w = 000111$, the (state; stack) evolution is

$$(q_1; \varepsilon) \overset{\varepsilon,\varepsilon\to\$}{\longrightarrow} (q_2; \$) \overset{0,\varepsilon\to 0}{\longrightarrow} (q_2; 0\$) \overset{0,\varepsilon\to 0}{\longrightarrow} (q_2; 00\$)$$

$$\overset{0,\varepsilon\to 0}{\longrightarrow} (q_2; 000\$) \overset{1,0\to\varepsilon}{\longrightarrow} (q_3; 00\$) \overset{1,0\to\varepsilon}{\longrightarrow} (q_3; 0\$) \overset{1,0\to\varepsilon}{\longrightarrow} (q_3; \$)$$

$$\overset{\varepsilon,\$\to\varepsilon}{\longrightarrow} (q_4; \varepsilon).$$

- Stack grows to the left, so leftmost symbol in stack is on top.
- Concatenation of what is read in sequence of transitions is $\varepsilon 000111 \varepsilon = w$.



- On input $w = 0111$, the (state; stack) evolution is

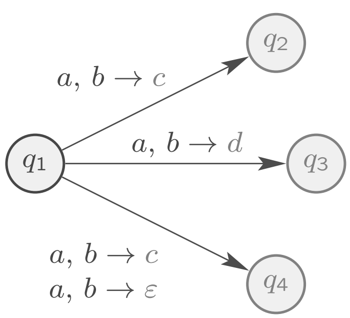
$$(q_1; \varepsilon) \overset{\varepsilon,\varepsilon\to\$}{\longrightarrow} (q_2; \$) \overset{0,\varepsilon\to 0}{\longrightarrow} (q_2; 0\$) \overset{1,0\to\varepsilon}{\longrightarrow} (q_3; \$) \overset{\varepsilon,\$\to\varepsilon}{\longrightarrow} (q_4; \varepsilon)$$

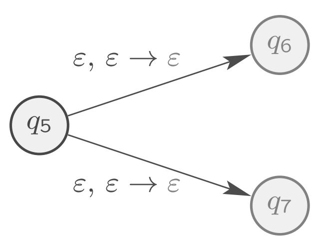
- Only first two symbols 01 were read from input $w = 0111$.
- PDA then crashes: there are still unread symbols 11 in input string w but PDA can't make any more transitions from q_4.
- No other way of processing, so string 0111 not accepted.
- Can show that PDA M recognizes language $\{\, 0^n 1^n \mid n \geq 0 \,\}$.

PDA May Be Nondeterministic

Recall: PDA transition function allows for nondeterminism

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





ε-transitions

Multiple choices when in state q_1,
read $a \in \Sigma_\varepsilon$, and pop $b \in \Gamma_\varepsilon$;
$\delta(q_1, a, b) = \{\, (q_2, c),\ (q_3, d),\ (q_4, c),\ (q_4, \varepsilon) \,\}$

Formal Definition of PDA Computation

- Recall PDA transition function $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon)$.
- PDA $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ **accepts** string $w \in \Sigma^*$ if
  - w can be written as $w = w_1 w_2 \cdots w_m$, where each $w_i \in \Sigma_\varepsilon$,
  - $\exists$ a sequence of states $r_0, r_1, \ldots, r_m \in Q$ and strings $s_0, s_1, \ldots, s_m \in \Gamma^*$ [stack contents on each transition]

  and the following hold:
  - $r_0 = q_0$ and $s_0 = \varepsilon$. [M starts in start state with empty stack.]
  - For each $i = 0, 1, \ldots, m-1$,

    $$(r_{i+1},\ b) \ \in\ \delta(r_i,\ w_{i+1},\ a),$$

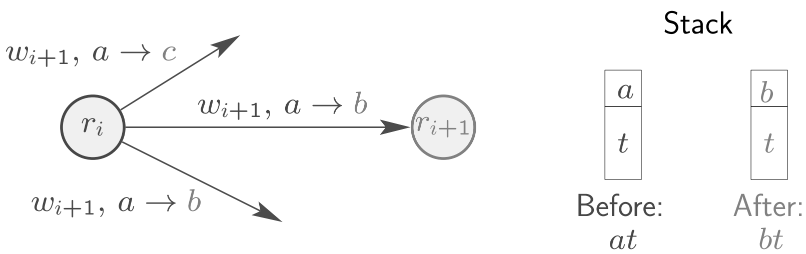
    where $s_i = at$ and $s_{i+1} = bt$ for some $a, b \in \Gamma_\varepsilon$ and $t \in \Gamma^*$. [M moves properly according to state, what's read, and stack.]
  - $r_m \in F$. [M ends in an accept state after reading entire input.]

Computation Requires Valid Sequence of Transitions

Recall for proper computation, we require for each transition i,

$$(r_{i+1},\ b)\ \in\ \delta(r_i,\ w_{i+1},\ a),$$

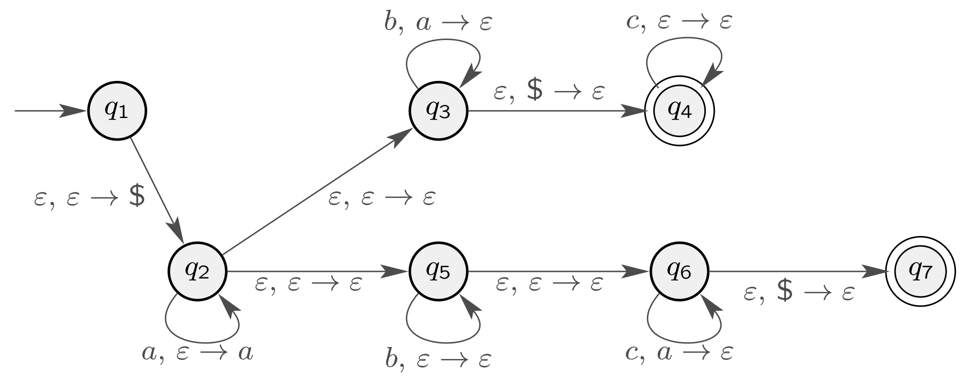
where $s_i = at$ and $s_{i+1} = bt$ for some $a, b \in \Gamma_\varepsilon$ and $t \in \Gamma^*$.



Definition: The set of all input strings that are accepted by PDA M is the **language recognized by** M and is denoted by $L(M)$.

- Note that $L(M) \subseteq \Sigma^*$.

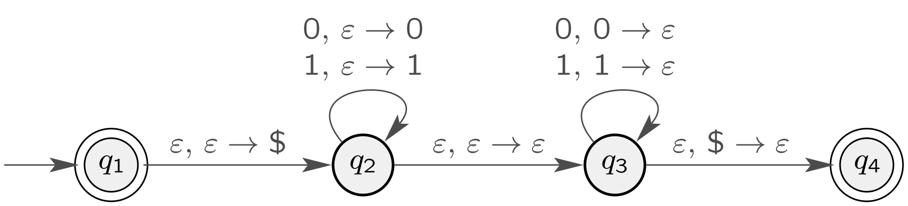
Example: PDA for language $\{\, a^i b^j c^k \mid i, j, k \geq 0 \text{ and } i = j \text{ or } i = k \,\}$



PDA guesses if it should match the a's

- with the b's (state q_3), or
- with the c's (state q_5)

Example: PDA for language $\{\, ww^{\mathcal{R}} \mid w \in \{0,1\}^* \,\}$



PDA works as follows:

- $q_1 \to q_2$: First pushes \$ on stack to mark bottom
- $q_2 \to q_2$: Reads in first half w of string, pushing it onto stack
- $q_2 \to q_3$: Guesses that it has reached middle of string
- $q_3 \to q_3$: Reads second half $w^{\mathcal{R}}$ of string, matching symbols from first half in reverse order (recall: stack LIFO)
- $q_3 \to q_4$: Makes sure that no more input symbols on stack

Equivalence of PDAs and CFGs

Theorem 2.20
A language is context free iff some PDA recognizes it.

Showing this equivalence requires two steps.

- **Lemma 2.21**
  If $A = L(G)$ for some CFG G,
  then $A = L(M)$ for some PDA M.
- **Lemma 2.27**
  If $A = L(M)$ for some PDA M,
  then $A = L(G)$ for some CFG G.

We will only show how the first lemma works.

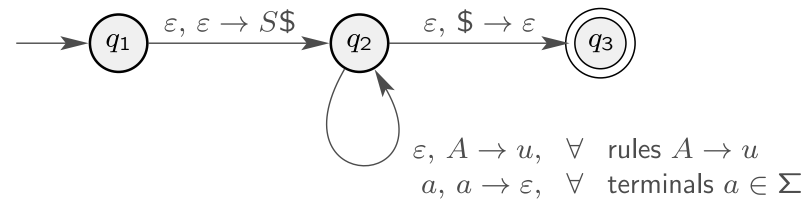
Lemma 2.21
If $A = L(G)$ for some CFG G, then $A = L(M)$ for some PDA M.

Proof Idea:

- Given CFG G, convert it into PDA M with $L(M) = L(G)$.
- Basic idea: build PDA that simulates a **leftmost derivation**.
- For example, consider CFG $G = (V, \Sigma, R, S)$
  - Variables $V = \{S, T\}$
  - Terminals $\Sigma = \{0, 1\}$
  - Rules: $S \rightarrow 0TS1 \mid 1T0, \quad T \rightarrow 1$
- Leftmost derivation of string $011101 \in L(G)$:

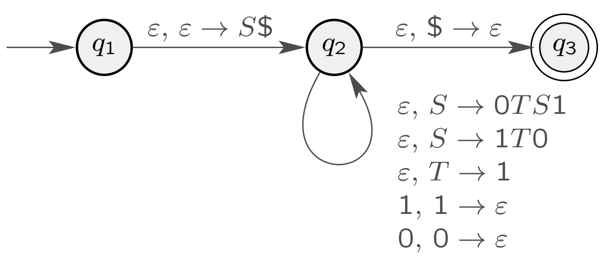
$$S \Rightarrow 0TS1 \Rightarrow 01S1 \Rightarrow 011T01 \Rightarrow 011101$$

- Convert CFG into PDA as follows:



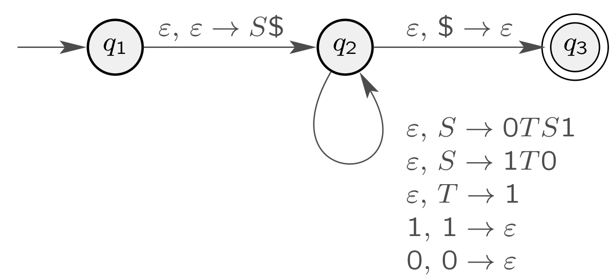
- PDA works as follows:
  1. Pushes \$ and then S on the stack, where S is start variable.
  2. Repeats following until stack empty
     (a) If top of stack is variable $A \in V$, then replace A by some $u \in (\Sigma \cup V)^*$, where $A \rightarrow u$ is a rule in R.
     (b) If top of stack is terminal $a \in \Sigma$ and next input symbol is a, then read and pop a.
     (c) If top of stack is \$, then pop it and accept.

- Recall CFG rules: $S \rightarrow 0TS1 \mid 1T0, \quad T \rightarrow 1$
- Corresponding PDA:



  - PDA is non-deterministic.
  - Input alphabet of PDA is the terminal alphabet of CFG
    - $\Sigma = \{0, 1\}$.
  - Stack alphabet consists of all variables, terminals and "\$"
    - $\Gamma = \{S, T, 0, 1, \$\}$.
  - PDA simulates a **leftmost derivation** using CFG
    - Pushes RHS of rule in **reverse order** onto stack.

- Recall CFG rules: $S \rightarrow 0TS1 \mid 1T0, \quad T \rightarrow 1$
- Corresponding PDA:

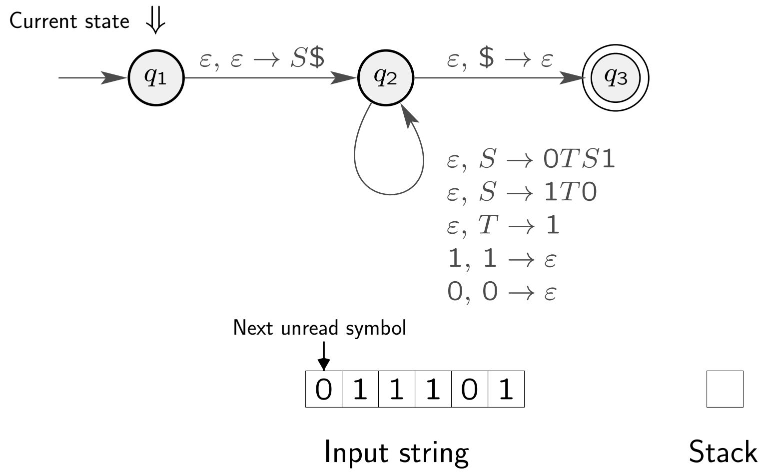


- Recall leftmost derivation of string $011101 \in L(G)$:

$$S \Rightarrow 0TS1 \Rightarrow 01S1 \Rightarrow 011T01 \Rightarrow 011101$$

- Let's now process string 011101 on PDA.
  - When in state q_2, look at top of stack to determine next transition.

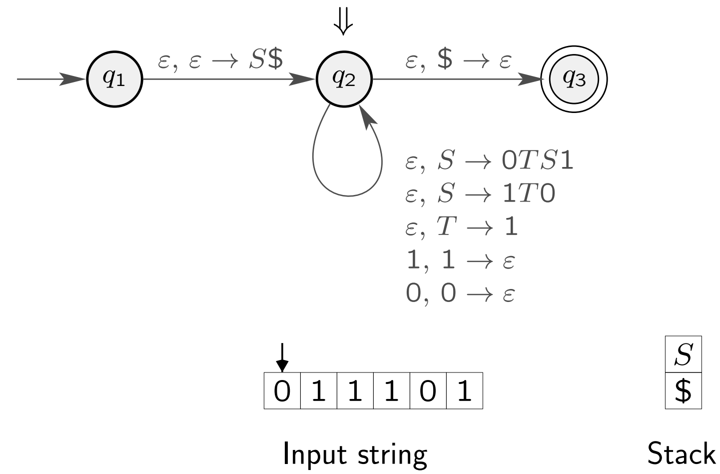
0. Start in state q_1 with 011101 on input tape and empty stack.



Leftmost derivation of string $011101 \in L(G)$:

$$S \Rightarrow 0TS1 \Rightarrow 01S1 \Rightarrow 011T01 \Rightarrow 011101$$

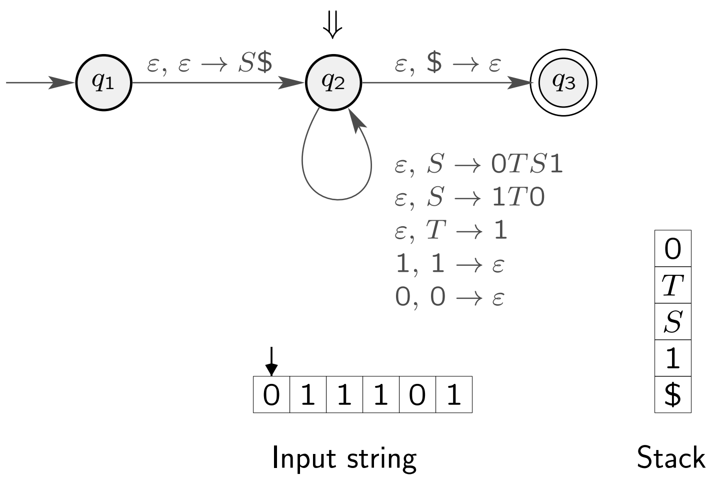
1. Read nothing, pop nothing, move to q_2, and push \$ and then S.



Leftmost derivation of string $011101 \in L(G)$:

$$\underline{S} \Rightarrow 0TS1 \Rightarrow 01S1 \Rightarrow 011T01 \Rightarrow 011101$$

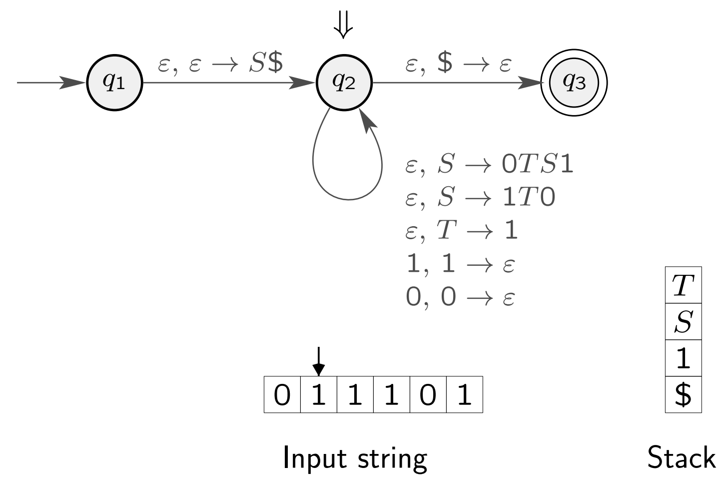
2. Read nothing, pop S, return to q_2, and push $0TS1$.



Leftmost derivation of string $011101 \in L(G)$:

$$S \Rightarrow \underline{0TS1} \Rightarrow 01S1 \Rightarrow 011T01 \Rightarrow 011101$$

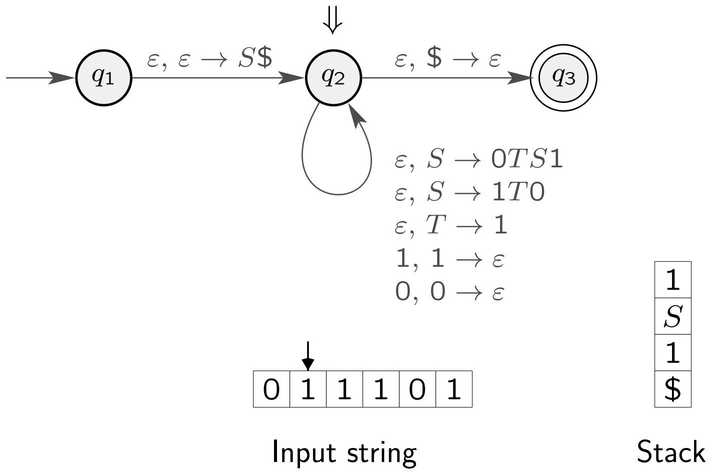
3. Read 0, pop 0, return to q_2, and push nothing.



Leftmost derivation of string $011101 \in L(G)$:

$$S \Rightarrow \underline{0TS1} \Rightarrow 01S1 \Rightarrow 011T01 \Rightarrow 011101$$

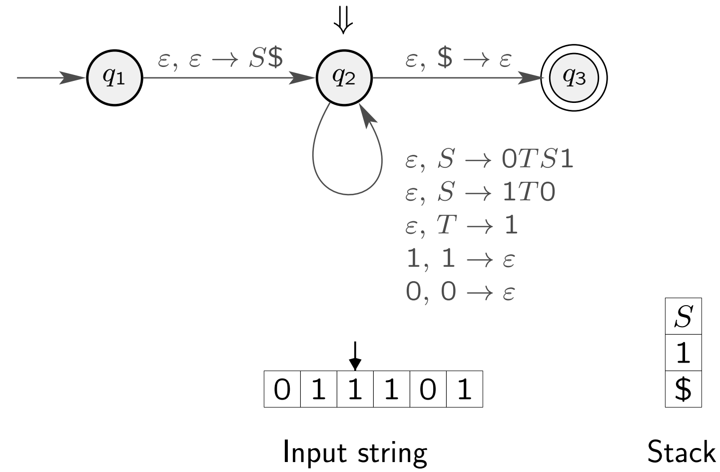
4. Read nothing, pop T, return to q_2, and push 1.



Leftmost derivation of string $011101 \in L(G)$:

$$S \Rightarrow 0TS1 \Rightarrow \underline{01S1} \Rightarrow 011T01 \Rightarrow 011101$$

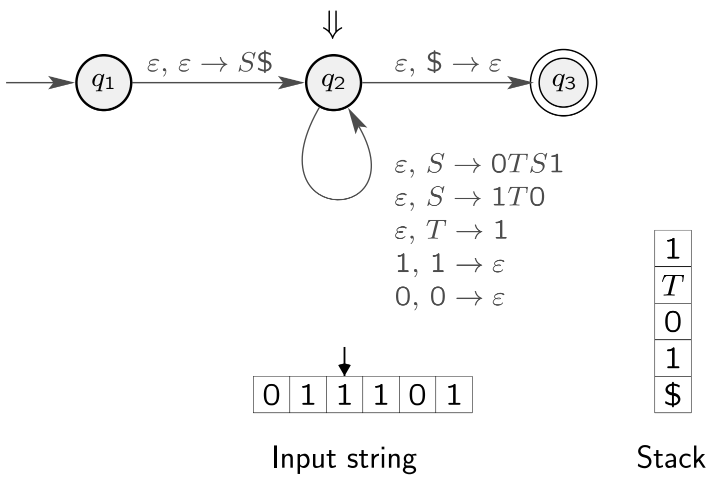
5. Read 1, pop 1, return to q_2, and push nothing.



Leftmost derivation of string $011101 \in L(G)$:

$$S \Rightarrow 0TS1 \Rightarrow \underline{01S1} \Rightarrow 011T01 \Rightarrow 011101$$

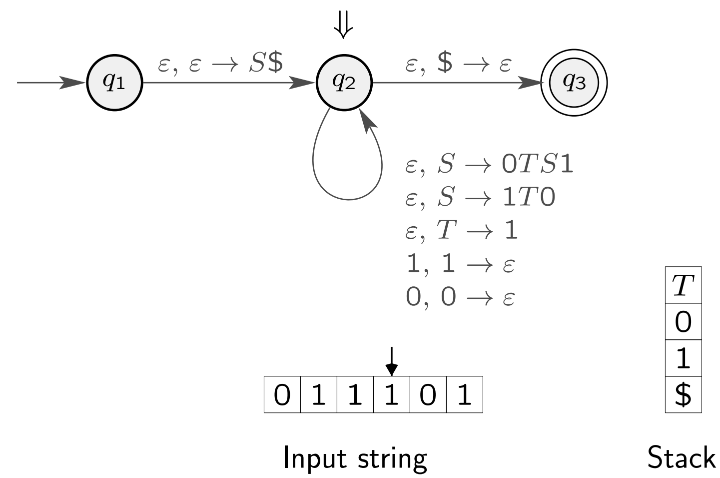
6. Read nothing, pop S, return to q_2, and push $1T0$.



Leftmost derivation of string $011101 \in L(G)$:

$$S \Rightarrow 0TS1 \Rightarrow 01S1 \Rightarrow \underline{011T01} \Rightarrow 011101$$

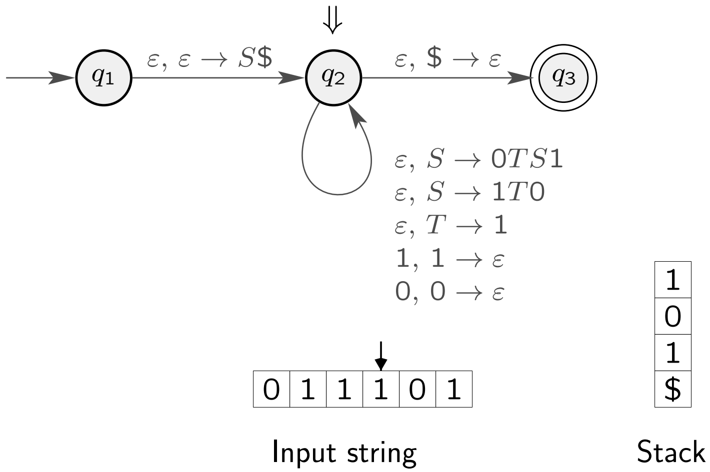
7. Read 1, pop 1, return to q_2, and push nothing.



Leftmost derivation of string $011101 \in L(G)$:

$$S \Rightarrow 0TS1 \Rightarrow 01S1 \Rightarrow \underline{011T01} \Rightarrow 011101$$

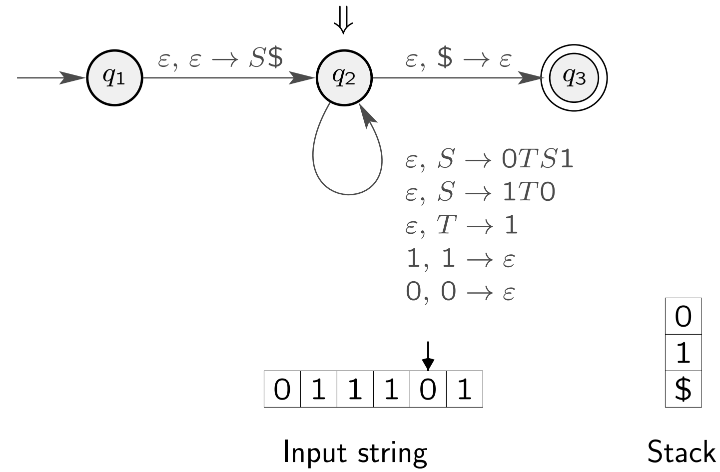
8. Read nothing, pop T, return to q_2, and push 1.



Leftmost derivation of string $011101 \in L(G)$:

$$S \Rightarrow 0TS1 \Rightarrow 01S1 \Rightarrow 011T01 \Rightarrow \underline{011101}$$

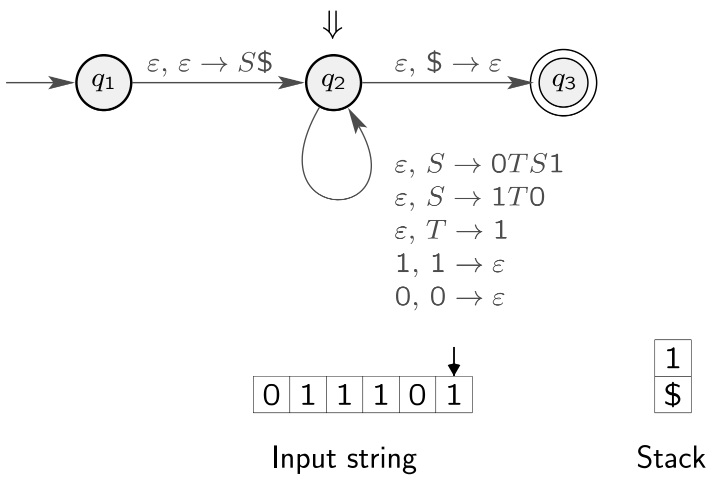
9. Read 1, pop 1, return to q_2, and push nothing.



Leftmost derivation of string $011101 \in L(G)$:

$$S \Rightarrow 0TS1 \Rightarrow 01S1 \Rightarrow 011T01 \Rightarrow \underline{011101}$$

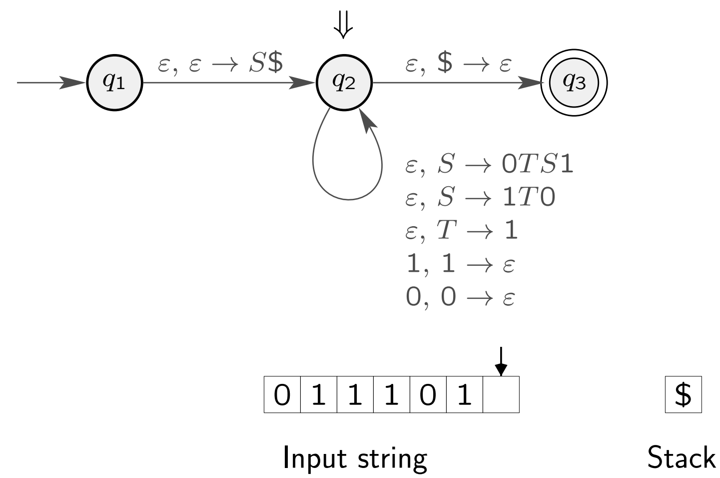
10. Read 0, pop 0, return to q_2, and push nothing.



Leftmost derivation of string $011101 \in L(G)$:

$$S \Rightarrow 0TS1 \Rightarrow 01S1 \Rightarrow 011T01 \Rightarrow \underline{011101}$$

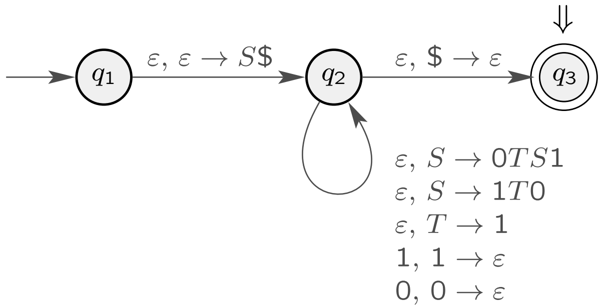
11. Read 1, pop 1, return to q_2, and push nothing.

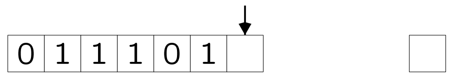


Leftmost derivation of string $011101 \in L(G)$:

$$S \Rightarrow 0TS1 \Rightarrow 01S1 \Rightarrow 011T01 \Rightarrow \underline{011101}$$

12. Read nothing, pop \$, move to q_3, push nothing, and *accept*.





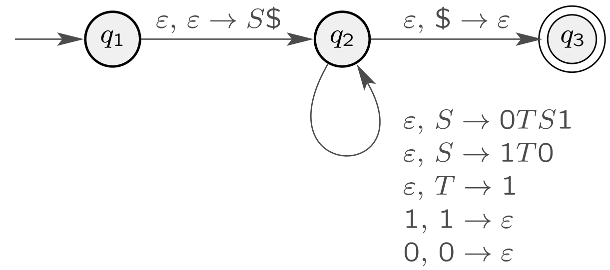
Input string Stack

Leftmost derivation of string $011101 \in L(G)$:

$$S \Rightarrow 0TS1 \Rightarrow 01S1 \Rightarrow 011T01 \Rightarrow \underline{011101}$$

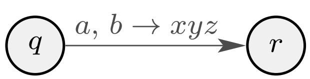
Constructed PDA is Not Compliant

- Recall CFG rules: $S \to 0TS1 \mid 1T0, \quad T \to 1$
- Corresponding PDA:

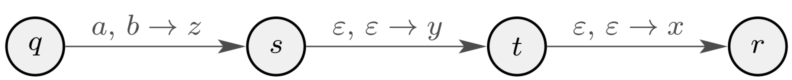


- **Problem:** pushing **strings** onto stack instead of ≤ 1 symbols, which is not allowed in PDA specification.
  - PDA transition fcn $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon)$

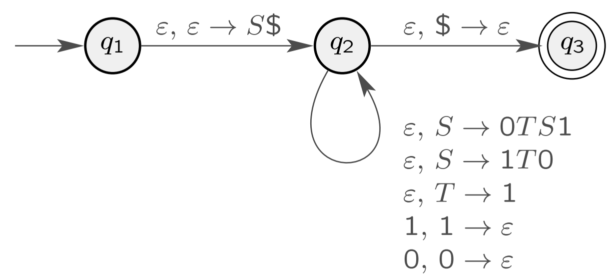
Solution: Add Extra States as Needed



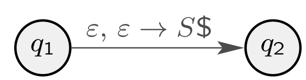
becomes



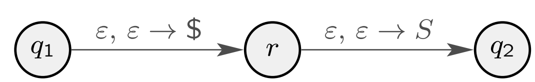
- For example, in our PDA



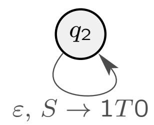
we replace



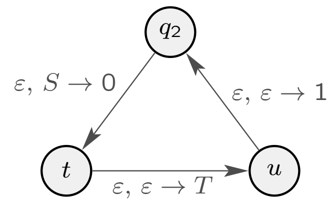
with



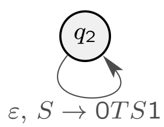
- Also, replace



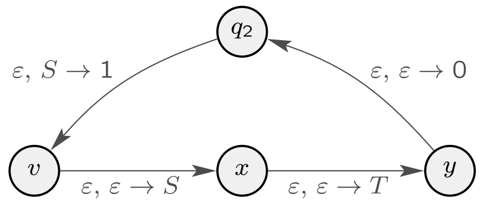
with



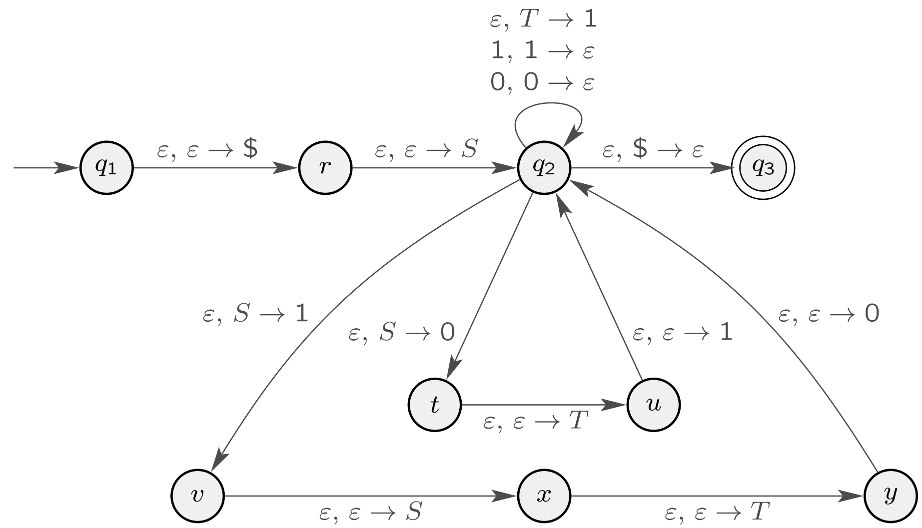
and replace



with



- So our final PDA from the CFG is



Regular $\Rightarrow$ CFL

Corollary 2.32
If A is a regular language, then A is also a CFL.

Proof.

- Suppose A is regular.
- Corollary 1.40 implies A has an NFA.
- But an NFA is just a PDA that ignores stack (always pops/pushes ε).
- So A has a PDA.
- Thus, Theorem 2.20 implies A is context-free.

Remark: Converse is not true.
For example, $\{\, 0^n1^n \mid n \geq 0 \,\}$ is CFL but not regular.

Pumping Lemma for CFLs

- Previously saw pumping lemma for regular languages.
- Analogous result for context-free language A.
- **Basic Idea:** Derivation of long string $s \in A$ has repeated variable R.
  - Long string implies tall parse tree, so must have repeated variable.
  - Can split string $s \in A$ into **5 pieces** $s = uvxyz$ based on R.
  - $uv^ixy^iz \in A$ for all $i \geq 0$.
- Consider language A with CFG G

$$\begin{aligned} S &\to CDa \mid CD \\ C &\to aD \\ D &\to Sb \mid b \end{aligned}$$

- Below "long" derivation using G repeats variable $R = D$:

$$\begin{aligned} S &\Rightarrow CDa \Rightarrow aDDa \Rightarrow ab\underline{D}a \Rightarrow abSba \Rightarrow abCDba \\ &\Rightarrow aba\underline{D}Dba \Rightarrow ababDba \Rightarrow ababbba \end{aligned}$$

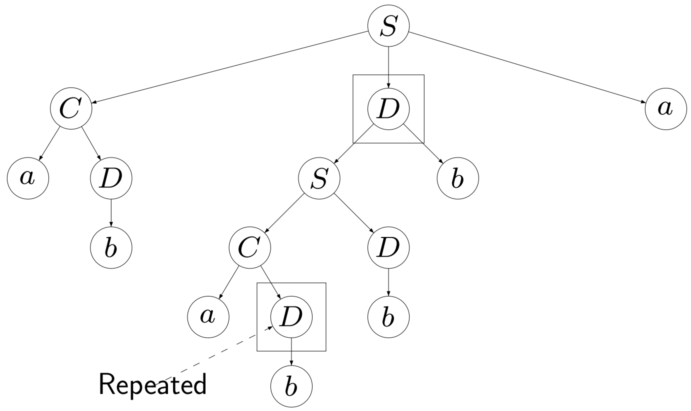
Repeated Variable in Path of Parse Tree

- Derivation of "long" string $s = ababbba \in A$ repeats variable D:

$$\boxed{\begin{array}{l} S \rightarrow CDa \mid CD \\ C \rightarrow aD \\ D \rightarrow Sb \mid b \end{array}}$$

$$S \Rightarrow CDa \Rightarrow aDDa \Rightarrow ab\underline{D}a \Rightarrow abSba \Rightarrow abCDba \Rightarrow aba\underline{D}Dba \Rightarrow ababDba \Rightarrow ababbba$$

- "Tall" parse tree repeats variable D on path from root to leaf.



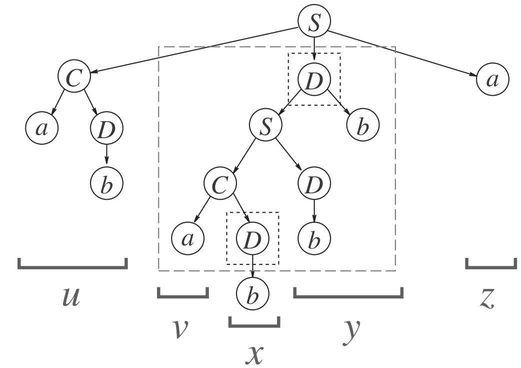
Split String Into 5 Pieces

- Split string $s \in A$ into

  $s = \underbrace{ab}_{u}\,\underbrace{a}_{v}\,\underbrace{b}_{x}\,\underbrace{bb}_{y}\,\underbrace{a}_{z}$

  using repeated variable D.

- In depth-first traversal of tree



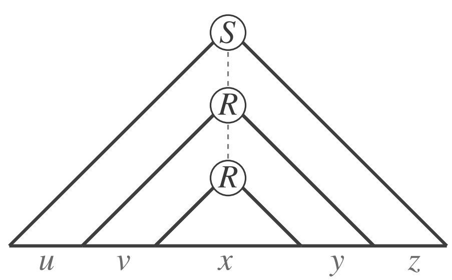
  - $u = ab$ is before D-D subtree
  - $v = a$ is before second D within D-D subtree
  - $x = b$ is what second D eventually becomes
  - $y = bb$ is after second D within D-D subtree
  - $z = a$ is after D-D subtree

Split Long String Into 5 Pieces

- More generally, consider "long" string $s \in A$.

- Parse tree is "tall"

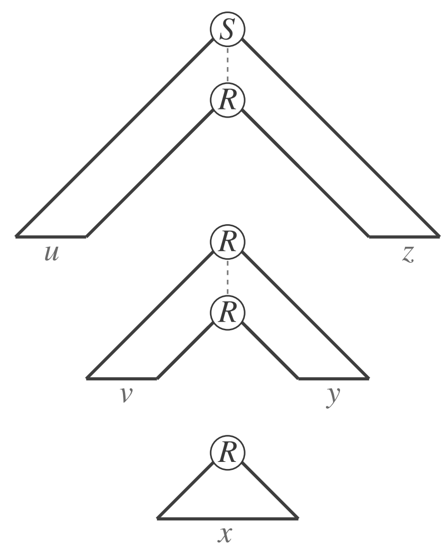
  - $\exists$ repeated variable R in path from root S to leaf.



- Split string $s = uvxyz$ into 5 pieces based on repeated variable R:

  - u is before R-R subtree (in depth-first order)
  - v is before second R within R-R subtree
  - x is what second R eventually becomes
  - y is after second R within R-R subtree
  - z is after R-R subtree

Subtrees Yield . . .



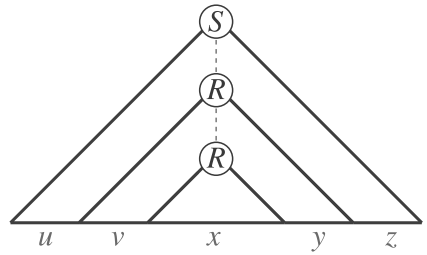
$$S \overset{*}{\Rightarrow} uRz$$

$$R \overset{*}{\Rightarrow} vRy$$

$$R \overset{*}{\Rightarrow} x$$

Can Pump To Obtain Other Strings in A

- Parse tree for string $s \in A$ implies
  - $S \overset{*}{\Rightarrow} uRz$ for $u, z \in \Sigma^*$
  - $R \overset{*}{\Rightarrow} vRy$ for $v, y \in \Sigma^*$
  - $R \overset{*}{\Rightarrow} x$ for $x \in \Sigma^*$



- Can derive string $s = uvxyz \in A$

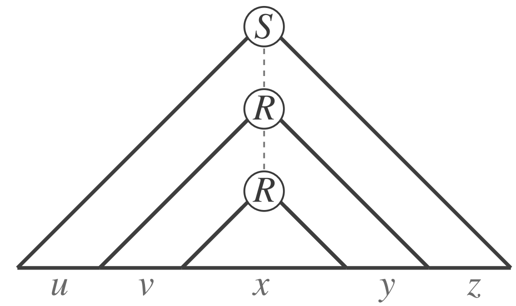
$$S \overset{*}{\Rightarrow} uRz \overset{*}{\Rightarrow} uvRyz \overset{*}{\Rightarrow} uvxyz \in A$$

- Also for each $i \geq 0$, can derive string

$$S \overset{*}{\Rightarrow} uRz \overset{*}{\Rightarrow} uvRyz \overset{*}{\Rightarrow} uvvRyyz \overset{*}{\Rightarrow} \cdots \overset{*}{\Rightarrow} uv^iRy^iz \overset{*}{\Rightarrow} uv^ixy^iz \in A$$

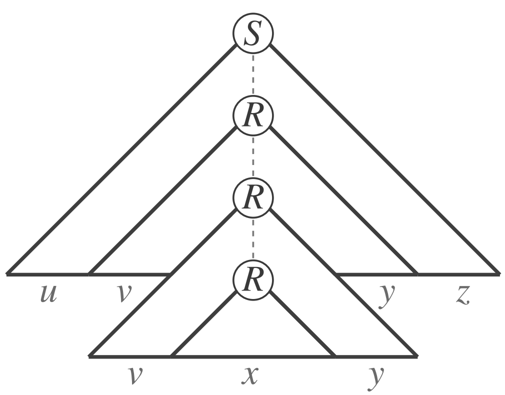
Pumping a Parse Tree

- Recall: $S \overset{*}{\Rightarrow} uRz$, $R \overset{*}{\Rightarrow} vRy$, $R \overset{*}{\Rightarrow} x$
- Consider parse tree of $uvxyz \in A$



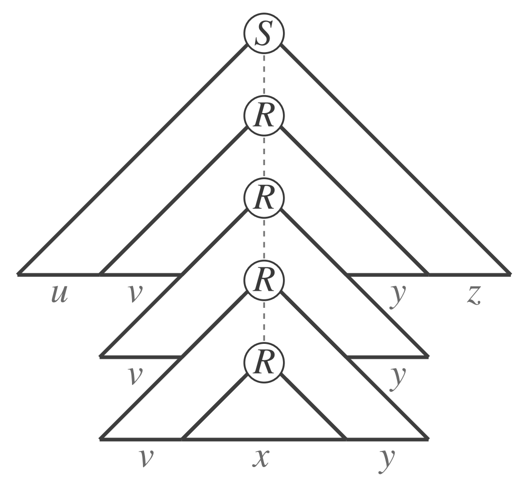
Pumping Up a Parse Tree

- Recall: $S \overset{*}{\Rightarrow} uRz$, $R \overset{*}{\Rightarrow} vRy$, $R \overset{*}{\Rightarrow} x$
- Using R-R subtree **twice** shows $uvvxyyz = uv^2xy^2z \in A$



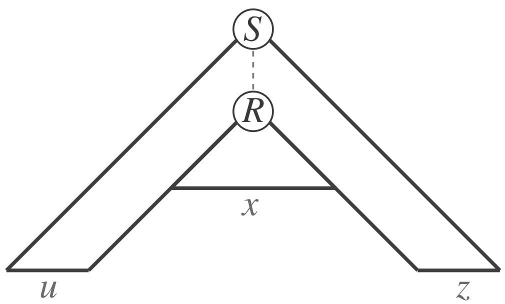
Pumping Up Multiple Times

- Recall: $S \overset{*}{\Rightarrow} uRz$, $R \overset{*}{\Rightarrow} vRy$, $R \overset{*}{\Rightarrow} x$
- Using R-R subtree **thrice** shows $uv^3xy^3z \in A$



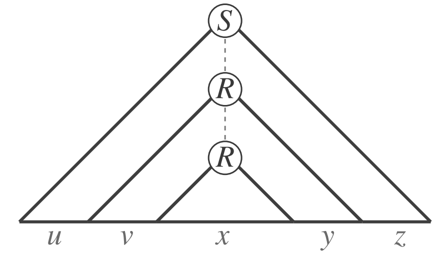
Pumping Down a Parse Tree

- Recall: $S \overset{*}{\Rightarrow} uRz,\ \ R \overset{*}{\Rightarrow} vRy,\ \ R \overset{*}{\Rightarrow} x$
- Removing R-R subtree shows $uxz = uv^0xy^0z \in A$



When Is Pumping Possible?

- **Key to Pumping**: repeated variable R in parse tree.
  - $S \overset{*}{\Rightarrow} uRz$ for $u, z \in \Sigma^*$
  - $R \overset{*}{\Rightarrow} vRy$ for $v, y \in \Sigma^*$
  - $R \overset{*}{\Rightarrow} x$ for $x \in \Sigma^*$
  - string $s = uvxyz \in A$

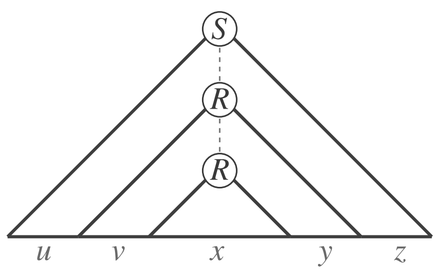


- Repeated variable $R \overset{*}{\Rightarrow} vRy$, so "$v$-$y$ pumping" possible:

$$S \overset{*}{\Rightarrow} uRz \overset{*}{\Rightarrow} uvRyz \overset{*}{\Rightarrow} uv^iRy^iz \overset{*}{\Rightarrow} uv^ixy^iz \in A$$

- If tree is **tall enough**, then repeated variable in path from root to leaf.
  - CFG has finite number $|V|$ of variables.
  - How tall does parse tree have to be to ensure pumping possible?
  - **Length** of path between two nodes = # edges in path.
  - Tree **height** = # edges on longest path from root to a leaf.

Can Pump If Parse Tree Is Tall Enough



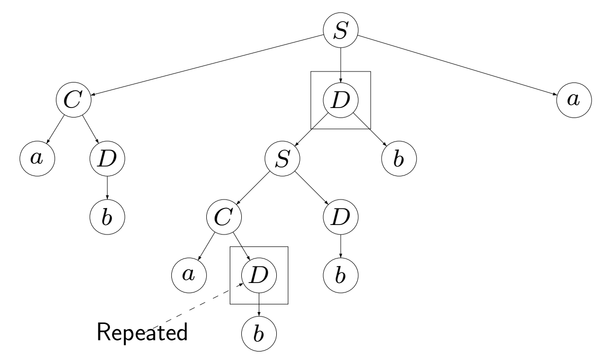
- Path from root S to leaf
  - Leaf is a terminal $\in \Sigma$
  - All other nodes along path are variables $\in V$.
- If height of tree $\geq |V| + 1$, where $|V|$ = # variables in CFG
  - then $\exists$ repeated variable on longest path from root to leaf.
- How long does string $s \in A$ have to be to ensure tall enough tree?

Previous Example

- $|V| = 3$ variables in below CFG:

$$\begin{aligned} S &\to CDa \mid CD \\ C &\to aD \\ D &\to Sb \mid b \end{aligned}$$

- In parse tree for $ababbba$, longest path has length $5 \geq |V| + 1 = 4$



If String s is Long Enough, Then Can Pump

- Let A have CFG in which longest rule has right-side length $b \geq 2$:
$$C \to D_1 \cdots D_b$$
  - So each node in tree has $\leq b$ children.
  - At most b leaves one step from root.
  - At most b^2 leaves 2 steps from root, and so on.
  - If tree has height $\leq h$, then
    - $\leq b^h$ leaves, so generated string s has length $|s| \leq b^h$.
  - Equiv: If string $s \in A$ has $|s| \geq b^h + 1$, then tree height $\geq h + 1$.
- Let $|V| = \#$ variables in CFG.
- If string $s \in A$ has length $|s| \geq p \equiv b^{|V|+1}$, then
  - tree height $\geq |V| + 1$ because $b^{|V|+1} \geq b^{|V|} + 1$.
  - some variable on longest path in tree is repeated
  - can pump parse tree.

Pumping Lemma for CFLs

Theorem 2.34
If A is context-free language, **then** $\exists$ pumping length p where,
if $s \in A$ with $|s| \geq p$, **then** s can be split into 5 pieces
$$s = uvxyz$$
satisfying the properties

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

Remarks:

- Property 1 implies that $uxz \in A$ by taking $i = 0$.
- Property 2 says that vy cannot be the empty string.
- Property 3 is sometimes useful.

Proof of Pumping Lemma for CFLs

- Let $G = (V, \Sigma, R, S)$ be CFG of A.
- Maximum size of rules is $b \geq 2$: $C \to D_1 \cdots D_b$
- From slide 2-93: If string $s \in A$ has length $|s| \geq p \equiv b^{|V|+1}$,
  - then longest path in parse tree has some repeated variable R:
$$S \stackrel{*}{\Rightarrow} uRz \stackrel{*}{\Rightarrow} uvRyz \stackrel{*}{\Rightarrow} uvxyz$$
- It follows that $uv^ixy^iz \in A$ for all $i = 0, 1, 2, \ldots$.
- Assume
  - parse tree is smallest one for string s
  - repeated R is among the bottom $|V| + 1$ variables on longest path.
- Then in tree, repeated part $R \stackrel{*}{\Rightarrow} vRy$ and $R \stackrel{*}{\Rightarrow} x$ satisfy
  - $|vy| > 0$ because tree is minimal.
  - bottom subtree with $R \stackrel{*}{\Rightarrow} vRy$ and $R \stackrel{*}{\Rightarrow} x$ has height $\leq |V| + 1$, so $|vxy| \leq b^{|V|+1} = p$.

Non-CFL

Remark: CFL Pumping Lemma (PL) mainly used to show certain languages are **not** CFL.

Example: Prove that $B = \{ a^nb^nc^n \mid n \geq 0 \}$ is non-CFL.
Proof.

- Suppose B is CFL, so PL implies B has pumping length $p \geq 1$.
- Consider string $s = a^pb^pc^p \in B$, so $|s| = 3p \geq p$.
- PL: **can** split s into 5 pieces $s = uvxyz = a^pb^pc^p$ satisfying
  1. $uv^ixy^iz \in B$ for all $i \geq 0$
  2. $|vy| > 0$
  3. $|vxy| \leq p$
- For contradiction, show **cannot** split $s = uvxyz$ satisfying 1–3.
  - Show **every** possible split satisfying Property 2 violates Property 1.

- Recall $s = uvxyz = \underbrace{aa\cdots a}_{p}\underbrace{bb\cdots b}_{p}\underbrace{cc\cdots c}_{p}$.
- Possibilities for split $s = uvxyz$ satisfying Property 2: $|vy| > 0$
  (i) Strings v and y are **uniform** [e.g., $v = a\cdots a$ and $y = b\cdots b$].
  - Then uv^2xy^2z won't have same number of a's, b's and c's because $|vy| > 0$.
  - Hence, $uv^2xy^2z \notin B$.

  (ii) Strings v and y are **not both uniform** [e.g., $v = a\cdots ab\cdots b$ and $y = b\cdots b$].
  - Then $uv^2xy^2z \notin L(a^*b^*c^*)$: symbols not grouped together.
  - Hence, $uv^2xy^2z \notin B$.
- Thus, every split satisfying Property 2 has $uv^2xy^2z \notin B$, so Property 1 violated.
- **Contradiction**, so $B = \{\, a^nb^nc^n \mid n \geq 0 \,\}$ is not a CFL.

Prove $C = \{\, a^ib^jc^k \mid 0 \leq i \leq j \leq k \,\}$ is not CFL

- Suppose C is CFL, so PL implies C has pumping length p.
- Take string $s = \underbrace{aa\cdots a}_{p}\underbrace{bb\cdots b}_{p}\underbrace{cc\cdots c}_{p} \in C$, so $|s| = 3p \geq p$.
- PL: **can** split $s = a^pb^pc^p$ into 5 pieces $s = uvxyz$ satisfying
  1. $uv^ixy^iz \in C$ for every $i \geq 0$, 2. $|vy| > 0$, 3. $|vxy| \leq p$.
- Property 3 implies vxy can't contain 3 different types of symbols.
- Two possibilities for v, x, y satisfying $|vy| > 0$ and $|vxy| \leq p$:
  (i) If $vxy \in L(a^*b^*)$, then z has all the c's
  - string uv^2xy^2z has too few c's because z not pumped
  - Hence, $uv^2xy^2z \notin C$

  (ii) If $vxy \in L(b^*c^*)$, then u has all the a's
  - string $uv^0xy^0z = uxz$ has too many a's
  - Hence, $uv^0xy^0z \notin C$
- Every split $s = uvxyz$ satisfying 2–3 violates 1, so C isn't CFL.

Prove $D = \{\, ww \mid w \in \{0,1\}^* \,\}$ is not CFL

- Suppose D is CFL, so PL implies D has pumping length p.
- Take $s = \underbrace{00\cdots 0}_{p}\underbrace{11\cdots 1}_{p}\underbrace{00\cdots 0}_{p}\underbrace{11\cdots 1}_{p} \in D$, so $|s| = 4p \geq p$.
- PL: **can** split s into 5 pieces $s = uvxyz$ satisfying
  1. $uv^ixy^iz \in D$ for every $i \geq 0$, 2. $|vy| > 0$, 3. $|vxy| \leq p$.

  (i) If vxy is entirely left of middle of $0^p1^p0^p1^p$,
  - then second half of uv^2xy^2z starts with a 1
  - so can't write uv^2xy^2z as ww because first half starts with 0.

  (ii) Similar reasoning: if vxy is entirely right of middle of $0^p1^p0^p1^p$,
  - then $uv^2xy^2z \notin D$

  (iii) If vxy straddles middle of $0^p1^p0^p1^p$,
  - then $uv^0xy^0z = uxz = 0^p1^j0^k1^p \notin D$ (because j or $k < p$)
- Every split $s = uvxyz$ satisfying 2–3 violates 1, so D isn't CFL.

Remarks on CFL Pumping Lemma

Often more difficult to apply CFL pumping lemma (Theorem 2.34) than pumping lemma for regular languages (Theorem 1.70).

- Carefully choose string s in language to get contradiction.
  - Not all strings s will give contradiction.
- CFL pumping lemma: "…**can** split s into 5 pieces $s = uvxyz$ satisfying all of Properties 1–3."
- To get contradiction, must show **cannot** split s into 5 pieces $s = uvxyz$ satisfying all of Properties 1–3.
  - Need to show **every possible** split $s = uvxyz$ **violates** at least one of Properties 1–3.

CFLs Closed Under Union

Is class of CFLs closed under standard operations?

Theorem:
If A_1 and A_2 are CFLs, then union $A_1 \cup A_2$ is CFL.

Proof.

- Assume
  - A_1 has CFG $G_1 = (V_1, \Sigma, R_1, S_1)$
  - A_2 has CFG $G_2 = (V_2, \Sigma, R_2, S_2)$.
- Assume that $V_1 \cap V_2 = \emptyset$.
- $A_1 \cup A_2$ has CFG $G_3 = (V_3, \Sigma, R_3, S_3)$ with
  - $V_3 = V_1 \cup V_2 \cup \{S_3\}$, where $S_3 \notin V_1 \cup V_2$ is new start variable
  - $R_3 = R_1 \cup R_2 \cup \{ S_3 \to S_1, S_3 \to S_2 \}$.

Example of Union of CFLs

- Suppose A_1 has CFG G_1 with rules:

$$\begin{aligned} S &\to aS \mid bXb \\ X &\to ab \mid baXb \end{aligned}$$

- Suppose A_2 has CFG G_2 with rules:

$$\begin{aligned} S &\to Sbb \mid aXba \\ X &\to b \mid XaX \end{aligned}$$

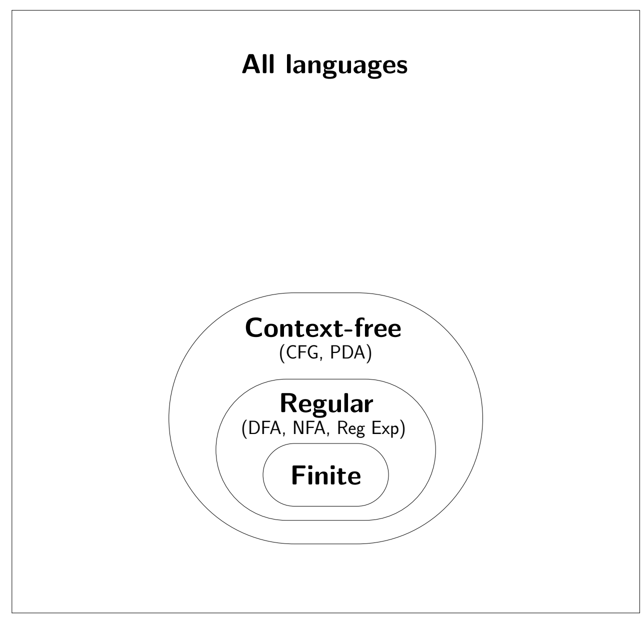
- Then $A_1 \cup A_2$ has CFG G_3 with start variable S_3 and rules:

$$\begin{aligned} S_3 &\to S_1 \mid S_2 \\ S_1 &\to aS_1 \mid bX_1b \\ X_1 &\to ab \mid baX_1b \\ S_2 &\to S_2bb \mid aX_2ba \\ X_2 &\to b \mid X_2aX_2 \end{aligned}$$

Some Closure Properties of CFLs

- Let A_1 and A_2 be two CFLs.
- Can prove that
  - union $A_1 \cup A_2$ is always CFL (slide 2-101)
  - concatenation $A_1 \circ A_2$ is always CFL
  - Kleene-star A_1^* is always CFL
- But
  - intersection $A_1 \cap A_2$ is not necessarily CFL
    - $A_1 = \{ a^n b^n c^k \mid n \geq 0, k \geq 0 \}$ and $A_2 = \{ a^k b^n c^n \mid n \geq 0, k \geq 0 \}$
  - complement $\overline{A_1} = \Sigma^* - A_1$ is not necessarily CFL.

Hierarchy of Languages (so far)



Examples

$\{ 0^n 1^n 2^n \mid n \geq 0 \}$

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

$(0 \cup 1)^*$

$\{ 110, 01 \}$

Summary of Chapter 2

- Context-free language is defined by CFG
- Parse trees
- Chomsky normal form: $A \to BC$ or $A \to x$, with $A \in V$, $B, C \in V - \{S\}$, $x \in \Sigma$. Also allow rule $S \to \varepsilon$.
- Pushdown automaton is NFA with stack for additional memory.
- Equivalence of PDAs and CFGs
- Regular $\Rightarrow$ CFL, but CFL $\not\Rightarrow$ Regular.
- Pumping lemma for CFLs: long strings in CFL can be pumped.
  - Repeat part of tall parse tree corresponding to repeated variable
  - Used to prove certain languages are non-CFL
- Class of CFLs closed under union, Kleene star, concatenation
- Class of CFLs **not** closed under intersection, complementation