

CS 341: Foundations of CS II

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

CS 341: Chapter 2

2-2

Chapter 2 Context-Free Languages

Contents

- Context-Free Grammars (CFG)
- Chomsky Normal Form
- $RL \Rightarrow CFL$
- Pushdown Automata (PDA)
- $PDA \Leftrightarrow CFG$
- Pumping Lemma for CFLs

CS 341: Chapter 2

2-3

Context-Free Languages (CFLs)

- Consider language $\{0^n 1^n \mid n \geq 0\}$, which is nonregular.
- Start variable S with “substitution rules”:

$$S \rightarrow 0S1$$

$$S \rightarrow \varepsilon$$

- Rules can **yield** string $0^k 1^k$ by
 - applying rule “ $S \rightarrow 0S1$ ” k times,
 - followed by rule “ $S \rightarrow \varepsilon$ ” once.
- **Derivation** of string $0^3 1^3$:

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

CS 341: Chapter 2

2-4

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

$$L \rightarrow X,$$

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 :

$$S \rightarrow 0S1$$

$$S \rightarrow \varepsilon$$

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

$$S \rightarrow 0S1 \mid \varepsilon$$

Deriving Strings Using CFG

Definition: If

- $u, v, w \in (V \cup \Sigma)^*$, and
- $A \rightarrow w$ 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 “ $A \rightarrow BB$ ”, we can have the derivation

$$01AB0 \Rightarrow 01BBB0.$$

Language of CFG

Definition: Write $u \xRightarrow{*} v$ if either

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

$$u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \dots \Rightarrow u_k \Rightarrow v$$

Remark: “ $\xRightarrow{*}$ ” denotes a sequence of several derivations (or none).

Example: With the rule “ $A \rightarrow BB$ ”,

$$0AA \xRightarrow{*} 0BBBB$$

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

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

Such a language is called **context-free**.

Example of CFG

- CFG $G = (V, \Sigma, R, S)$ with

1. $V = \{S\}$
2. $\Sigma = \{0, 1\}$
3. Rules R :

$$S \rightarrow 0S \mid \varepsilon$$

- Then $L(G) = \{0^n \mid n \geq 0\}$.

- For example, S yields 0^3 since

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

Example of CFG

- CFG $G = (V, \Sigma, R, S)$ with

1. $V = \{S\}$
2. $\Sigma = \{0, 1\}$
3. Rules R :

$$S \rightarrow 0S \mid 1S \mid \varepsilon$$

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

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

Example of CFG

- CFG $G = (V, \Sigma, R, S)$ with

1. $V = \{S\}$
2. $\Sigma = \{0, 1\}$
3. Rules R :

$$S \rightarrow 0S \mid 1S \mid 1$$

- 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 yields 011 since

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

Example of CFG

- CFG $G = (V, \Sigma, R, S)$ with

1. $V = \{S, Z\}$
2. $\Sigma = \{0, 1\}$
3. Rules R :

$$S \rightarrow 0S1 \mid Z$$

$$Z \rightarrow 0Z \mid \varepsilon$$

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

$$\begin{aligned} S &\Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 000S111 \Rightarrow 000Z111 \\ &\Rightarrow 0000Z111 \Rightarrow 00000Z111 \Rightarrow 00000\varepsilon111 \\ &= 00000111 \end{aligned}$$

CFG for Palindrome

- PALINDROME = $\{w \in \Sigma^* \mid w = w^R\}$, where $\Sigma = \{a, b\}$.

- CFG $G = (V, \Sigma, R, S)$ with

1. $V = \{S\}$
2. $\Sigma = \{a, b\}$
3. Rules R :

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

- Then $L(G) = \text{PALINDROME}$
- S yields $abaaba$ since

$$S \Rightarrow aSa \Rightarrow abSba \Rightarrow abaSaba \Rightarrow aba\varepsilonaba = abaaba$$
- S yields $aabaa$ since

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

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

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

- CFG $G = (V, \Sigma, R, S)$ with

- $V = \{S, X, Y\}$
- $\Sigma = \{a, b\}$
- Rules R :

$$S \rightarrow aaS \mid bbS \mid XYXS \mid \varepsilon$$

$$X \rightarrow ab \mid ba$$

$$Y \rightarrow aaY \mid bbY \mid \varepsilon$$

- Then $L(G) = \text{EVEN-EVEN}$

CFG for Simple Arithmetic Expressions

- CFG $G = (V, \Sigma, R, S)$ with

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

$$S \rightarrow S + S \mid S - S \mid S \times S \mid S / S \mid (S) \mid -S \mid 0 \mid 1 \mid \dots \mid 9$$

- $L(G)$ is a set of valid arithmetic expressions over single-digit integers.

- S yields $2 \times (3 + 4)$ since

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

Derivation Tree

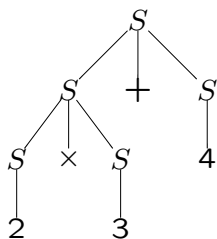
- CFG

$$S \rightarrow S + S \mid S - S \mid S \times S \mid S / S \mid (S) \mid -S \mid 0 \mid 1 \mid \dots \mid 9$$

- Can generate $2 \times 3 + 4$ 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** since leftmost variable replaced in each step.
- Corresponding **derivation (or parse) tree**:



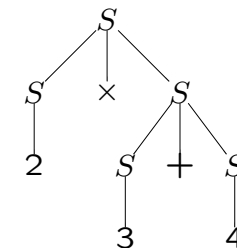
Ambiguous CFG

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

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

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)
- Specification of programming languages:
 - parsing a computer program
- Describes mathematical structures, etc.
- Intermediate class between regular languages 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. Which languages are outside the class of CFLs?

Chomsky Normal Form

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

$$A \rightarrow BC$$

or $A \rightarrow x$

with

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

For the start variable S , we also allow the rule $S \rightarrow \varepsilon$.

Example: Rules of CFG in Chomsky normal form:

$$S \rightarrow XX \mid XW \mid a \mid \varepsilon$$

$$X \rightarrow WX \mid b$$

$$W \rightarrow a$$

Can Always Put CFG into Chomsky Normal Form

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

Theorem 2.9

Every CFL can be described by a grammar 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 ($A \rightarrow \varepsilon$ not allowed when A isn't start variable)
 - all other violating rules ($A \rightarrow B$, $A \rightarrow aBc$, $A \rightarrow BCDE$)

Converting CFG into Chomsky Normal Form

1. Introduce

- New start variable S_0
- New rule $S_0 \rightarrow S$

2. Remove ε -rules $A \rightarrow \varepsilon$

- Before: $B \rightarrow xAy$ and $A \rightarrow \varepsilon$
- After: $B \rightarrow xAy \mid xy$

3. Remove **unit rules** $A \rightarrow B$

- Before: $A \rightarrow B$ and $B \rightarrow xCy$
- After: $A \rightarrow xCy$ and $B \rightarrow xCy$

4. Replace ill-placed terminals a by variable T_a with rule $T_a \rightarrow a$.

- Before: $A \rightarrow ab$
- After: $A \rightarrow T_a T_b$, $T_a \rightarrow a$, $T_b \rightarrow b$.

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

- Before: $A \rightarrow B_1 B_2 \cdots B_k$
- After: $A \rightarrow B_1 A_1$, $A_1 \rightarrow B_2 A_2$, \dots , $A_{k-2} \rightarrow B_{k-1} B_k$

6. Be careful about removing rules:

- Do not introduce new rules that you removed earlier.
- **Example:** $A \rightarrow A$ simply disappears
- When removing $A \rightarrow \varepsilon$ rules, insert all new replacements:
 - Before: $B \rightarrow AbA$ and $A \rightarrow \varepsilon$
 - After: $B \rightarrow AbA \mid bA \mid Ab \mid b$

Example: Convert CFG into Chomsky Normal Form

Initial CFG G_0 :

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

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

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

Example: Convert CFG into Chomsky Normal Form

From previous slide

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

2. Remove ε -rules:

- (i) remove $Y \rightarrow \varepsilon$ (ii) remove $X \rightarrow \varepsilon$

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

Example: Convert CFG into Chomsky Normal Form

From previous slide

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

3. Remove unit rules:

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

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

Example: Convert CFG into Chomsky Normal Form

From previous slide

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

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

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

Example: Convert CFG into Chomsky Normal Form

From previous slide

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

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

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

Example: Convert CFG into Chomsky Normal Form

From previous slide

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

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

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

Example: Convert CFG into Chomsky Normal Form

From previous slide

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

4. Replace ill-placed terminals a by variable U with $U \rightarrow a$.

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

Example: Convert CFG into Chomsky Normal Form

From previous slide

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

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

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

which is a CFG in Chomsky normal form.

Regular \Rightarrow CFL**Corollary 2.32**If A is a regular language, then A is also a CFL.**Proof Idea:**

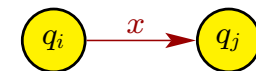
- Suppose A has DFA $M = (Q, \Sigma, \delta, q_0, F)$.
- Construct corresponding CFG $G_M = (V, \Sigma, R, S)$ with
 - $V = Q$
 - ▲ There is a variable in CFG for each state in DFA.
 - $S = q_0$
 - ▲ Start variable in CFG is start state from DFA.

■ Rules of CFG G_M :

$$\begin{aligned} q_i &\rightarrow x\delta(q_i, x) && \text{for all } q_i \in V \text{ and all } x \in \Sigma, \\ q_i &\rightarrow \varepsilon && \text{for all } q_i \in F \end{aligned}$$

CFG

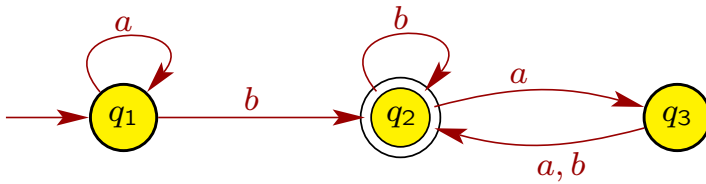
$$q_i \rightarrow xq_j$$

**DFA**

$$q_i \rightarrow \varepsilon$$



Example: DFA $M = (Q, \Sigma, \delta, q_1, F)$ with $\Sigma = \{a, b\}$



Corresponding CFG $G_M = (Q, \Sigma, R, q_1)$ with rules

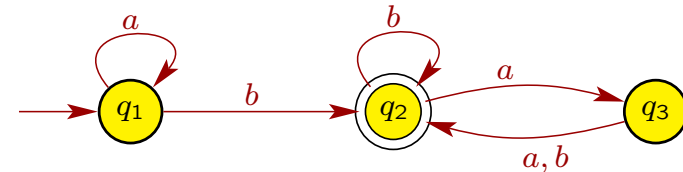
$$\begin{aligned} q_1 &\rightarrow a q_1 \mid b q_2 \\ q_2 &\rightarrow a q_3 \mid b q_2 \mid \varepsilon \\ q_3 &\rightarrow a q_2 \mid b q_2 \end{aligned}$$

Definition: Consider

- DFA $M = (Q, \Sigma, \delta, q_1, F)$
- string $w = w_1 w_2 \cdots w_k$ with each $w_i \in \Sigma$.

If M accepts w , then the **path development** of w on M is a sequence of strings s_0, s_1, \dots, s_{k+1} , each in $\Sigma^* \circ (Q \cup \{\varepsilon\})$, such that

- $s_i = w_1 w_2 \cdots w_i q$ for $0 \leq i \leq k$
 - q is state DFA is in after reading first i symbols
- $s_{k+1} = w$, the entire original string.



Path development of $abaa$ is

$$q_1, aq_1, abq_2, abaq_3, abaaq_2, abaa$$

- Recall path development of $abaa$ is

$$q_1, aq_1, abq_2, abaq_3, abaaq_2, abaa$$

- Recall CFG $G_M = (Q, \Sigma, R, q_1)$ with rules

$$\begin{aligned} q_1 &\rightarrow a q_1 \mid b q_2 \\ q_2 &\rightarrow a q_3 \mid b q_2 \mid \varepsilon \\ q_3 &\rightarrow a q_2 \mid b q_2 \end{aligned}$$

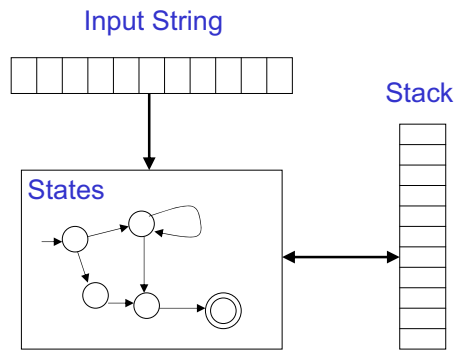
- Derivation of $abaa$ using CFG G_M is

$$q_1 \Rightarrow aq_1 \Rightarrow abq_2 \Rightarrow abaq_3 \Rightarrow abaaq_2 \Rightarrow abaa$$

- 1-1 correspondence between
 - path developments on DFA M , and
 - derivations using CFG G_M .
- Thus, $L(M) = L(G_M)$.

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 are inherently nondeterministic.
- **Defn: Stack** is data structure of unlimited size with operations
 - **push**, which adds item to top of stack,
 - **pop**, which removes item from top of stack.
 Last-In-First-Out (LIFO).

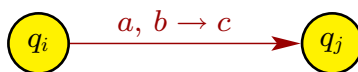


- PDA has
 - States
 - Stack with alphabet Γ
 - Transitions between states based on
 - ▲ current state
 - ▲ what is read
 - ▲ what is popped from stack.

PDA Uses Stack

- **General idea:** CFLs are languages that can be recognized by automata that have one stack:
 - $\{0^n 1^n \mid n \geq 0\}$ is a CFL
 - $\{0^n 1^n 0^n \mid n \geq 0\}$ is not a CFL
- Recall for alphabet Σ , we defined $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$.
- Let Γ be **stack alphabet**
 - Γ is set of symbols that can be pushed onto and popped off stack.
- Let $\Gamma_\varepsilon = \Gamma \cup \{\varepsilon\}$.
 - Pushing or popping ε leaves contents of stack unchanged.

PDA Transitions



Interpretation:

- 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

- String $w = w_1 w_2 \cdots w_k$ is input to PDA M , where each $w_i \in \Sigma_\varepsilon$.
- PDA reads in each w_i one at a time, from left to right.
- If PDA M
 - is currently in state $q \in Q$,
 - reads $w_i \in \Sigma_\varepsilon$, and
 - pops $s_i \in \Gamma_\varepsilon$ from stack,
- then PDA M nondeterministically
 - moves to new state in Q , and
 - pushes an element from Γ_ε on top of stack.
- After reading last w_k , PDA M will be in some state $q \in Q$.
- If possible to end in accept state $\in F \subseteq Q$ after reading entire input, then M accepts w .

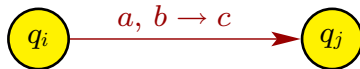
Definition of PDA

Definition: Pushdown automaton (PDA)

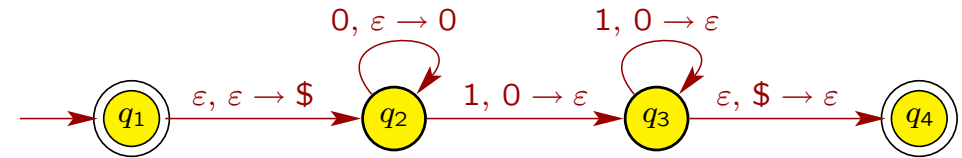
$M = (Q, \Sigma, \Gamma, \delta, q_0, F)$:

- 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, $F \subseteq Q$
- δ is transition function

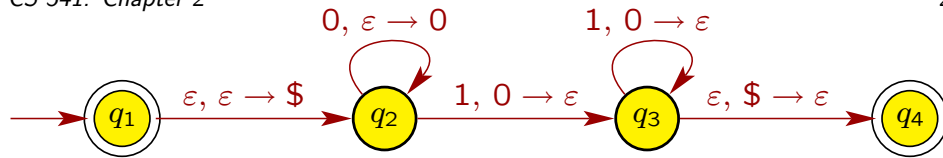
$$\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \rightarrow \mathcal{P}(Q \times \Gamma_\epsilon)$$



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

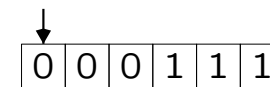
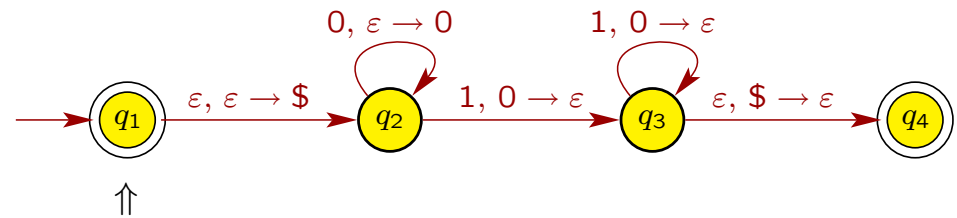


• transition function $\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \rightarrow \mathcal{P}(Q \times \Gamma_\epsilon)$

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

Blank entries are \emptyset

• Let's process string 000111 on our PDA.

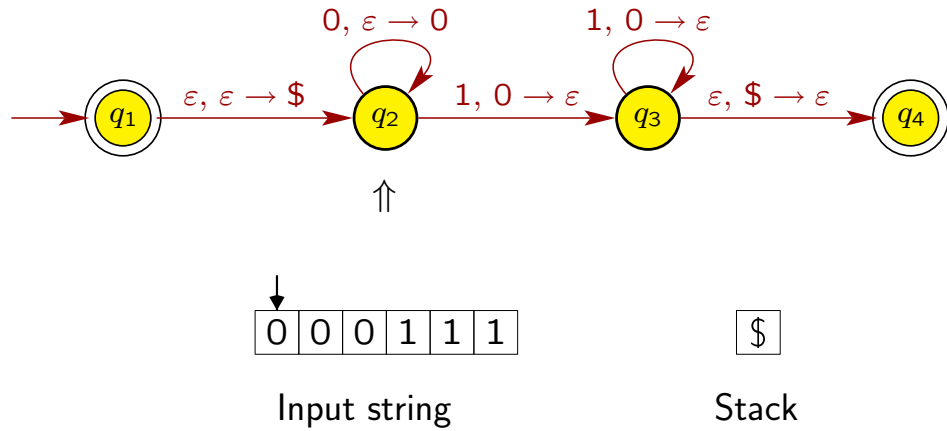


Input string

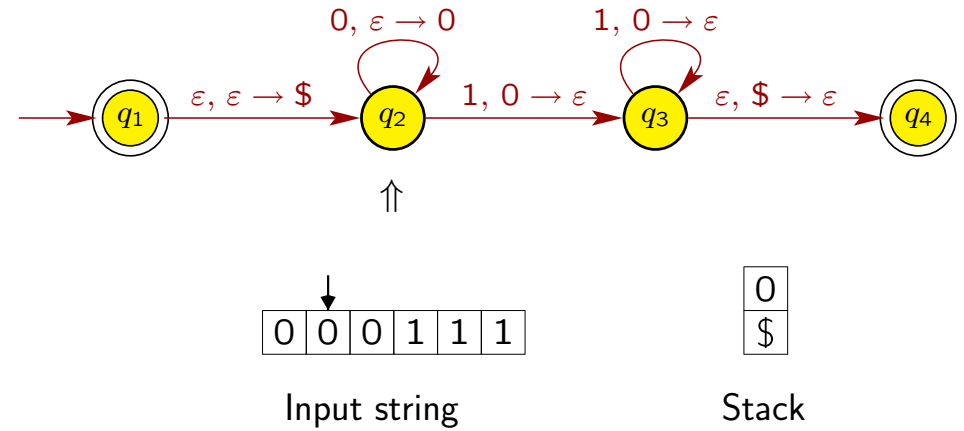


Stack

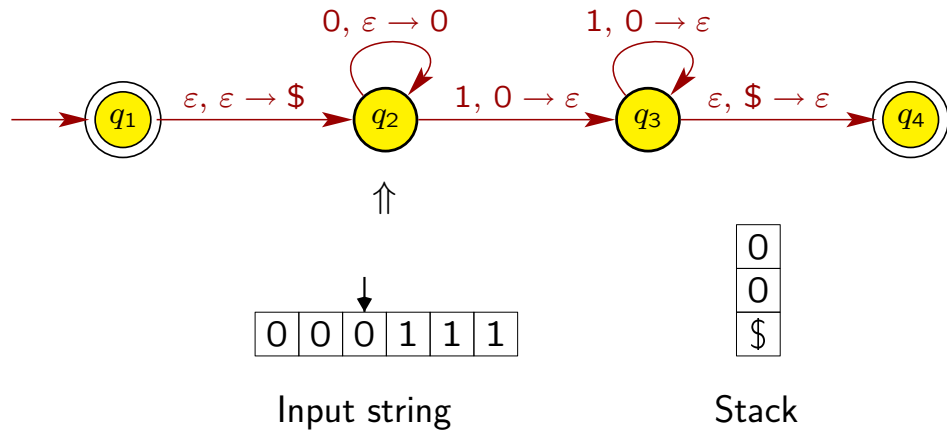
- Start in start state q_1 .
- No input symbols read so far.
- Next go to state q_2
 - reading nothing, popping nothing, and pushing \$ on stack.



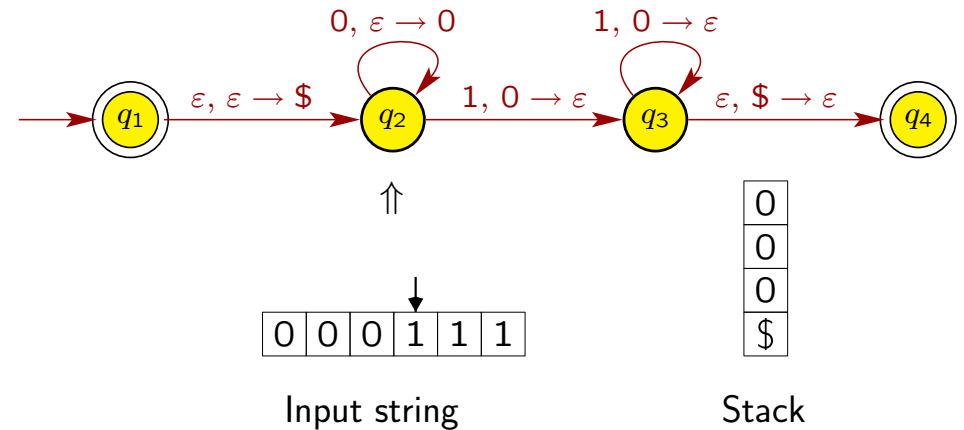
- Next return to state q_2
 - reading input symbol 0
 - popping nothing from stack
 - pushing 0 on stack.



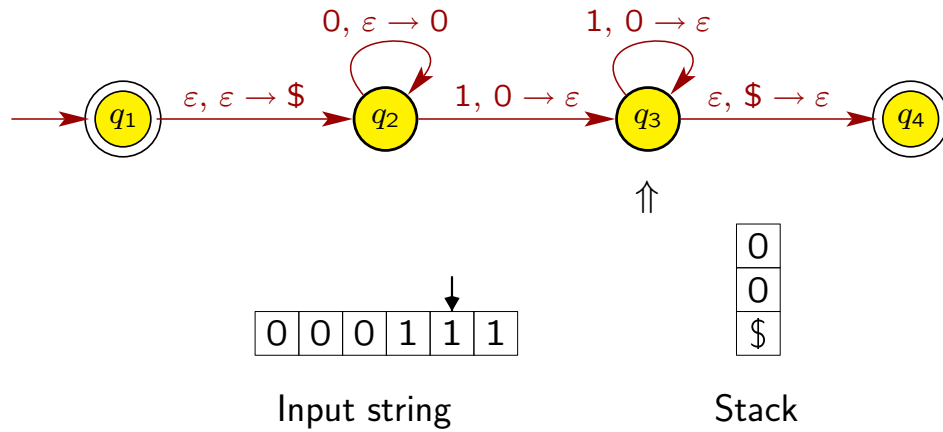
- Next return to state q_2
 - reading input symbol 0
 - popping nothing from stack
 - pushing 0 on stack.



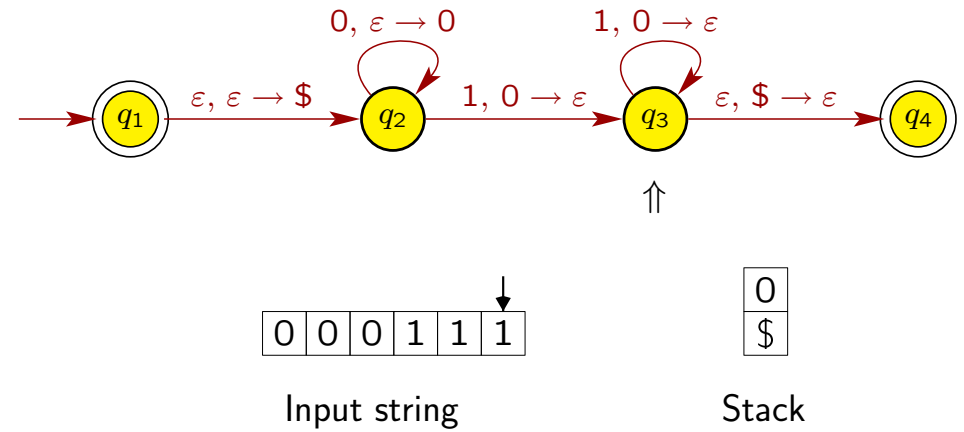
- Next return to state q_2
 - reading input symbol 0
 - popping nothing from stack
 - pushing 0 on stack.



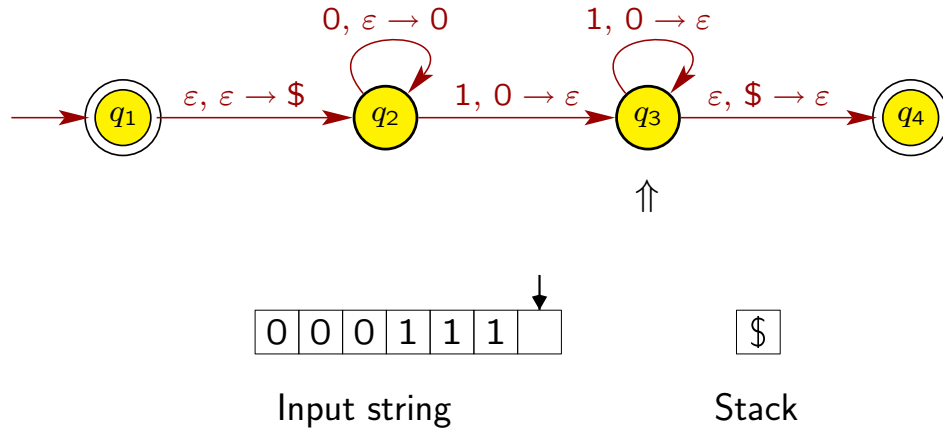
- Next go to state q_3
 - reading input symbol 1
 - popping 0 from stack
 - pushing nothing on stack.



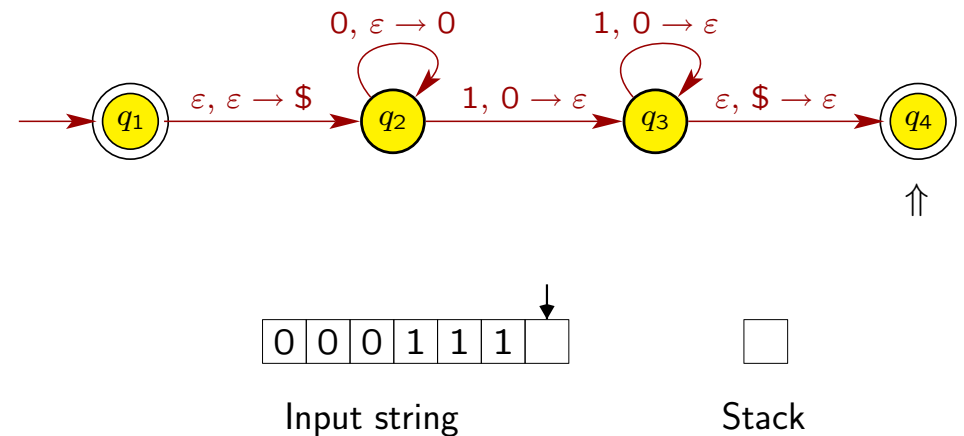
- Next return to state q_3
 - reading input symbol 1
 - popping 0 from stack
 - pushing nothing on stack.



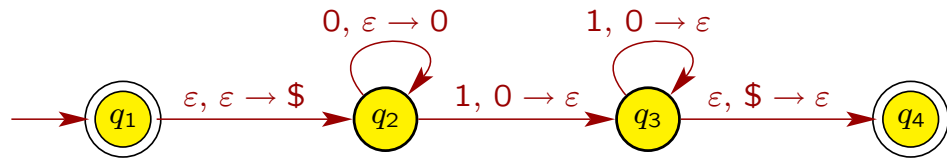
- Next go to state q_3
 - reading input symbol 1
 - popping 0 from stack
 - pushing nothing on stack.



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

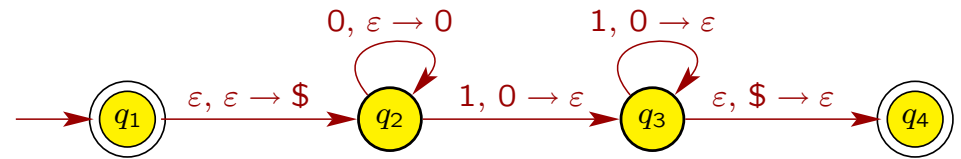


- String 000111 is accepted by PDA since
 - q_4 is an accept state and
 - we read the entire input string.



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

$$\begin{aligned}
 (q_1; \varepsilon) &\xrightarrow{\varepsilon, \varepsilon \rightarrow \$} (q_2; \$) \xrightarrow{0, \varepsilon \rightarrow 0} (q_2; 0\$) \xrightarrow{0, \varepsilon \rightarrow 0} (q_2; 00\$) \\
 &\xrightarrow{0, \varepsilon \rightarrow 0} (q_2; 000\$) \xrightarrow{1, 0 \rightarrow \varepsilon} (q_3; 00\$) \xrightarrow{1, 0 \rightarrow \varepsilon} (q_3; 0\$) \xrightarrow{1, 0 \rightarrow \varepsilon} (q_3; \$) \\
 &\xrightarrow{\varepsilon, \$ \rightarrow \varepsilon} (q_4; \varepsilon).
 \end{aligned}$$



• On input $w = 0101$, the (state; stack) evolution is

$$(q_1; \varepsilon) \xrightarrow{\varepsilon, \varepsilon \rightarrow \$} (q_2; \$) \xrightarrow{0, \varepsilon \rightarrow 0} (q_2; 0\$) \xrightarrow{1, 0 \rightarrow \varepsilon} (q_3; \$) \xrightarrow{\varepsilon, \$ \rightarrow \varepsilon} (q_4; \varepsilon)$$

- Only first two symbols 01 were read from input $w = 0101$.
- There are still unread symbols 01 from input w .
- No other way of processing, so string 0101 not accepted.
- Can show that PDA M recognizes language $\{0^n 1^n \mid n \geq 0\}$.

Formal Definition of PDA Computation

- Recall PDA transition function $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \rightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$.
- PDA $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ **accepts** string $w \in \Sigma^*$ if
 - w can be written as $w = w_1 w_2 \dots w_m$, where each $w_i \in \Sigma_\varepsilon$,
 - \exists a sequence of states $r_0, r_1, \dots, r_m \in Q$ and strings $s_0, s_1, \dots, s_m \in \Gamma^*$,
 and the following hold:
 - $r_0 = q_0$ and $s_0 = \varepsilon$. (M starts in start state with empty stack.)
 - For each $i = 0, 1, \dots, m - 1$,

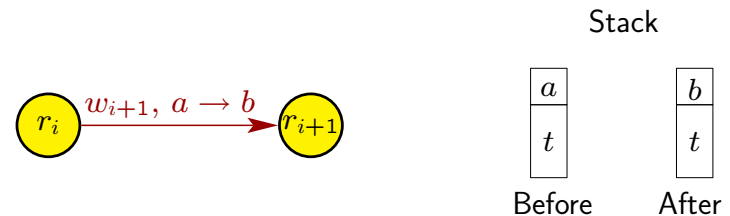
$$(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a),$$
 where $s_i = at$ and $s_{i+1} = bt$ for some $a, b \in \Gamma_\varepsilon$ and $t \in \Gamma^*$. (M moves properly according to state, stack, and what's read.)
 - $r_m \in F$. (M ends in an accept state after reaching end of input.)

Proper Computation Requires Stack Consistency

Recall for proper computation, we require for each 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 $t \in \Gamma^*$.



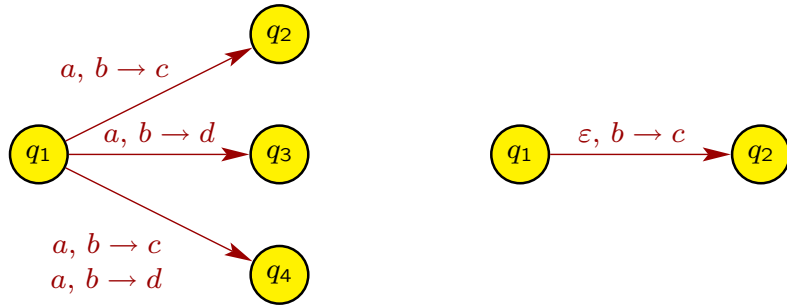
Definition: The set of all input strings that are accepted by PDA M is the language **recognized** by M and is denoted by $L(M)$.

PDA May Be Nondeterministic

Recall: PDA transition function

$$\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \rightarrow \mathcal{P}(Q \times \Gamma_\epsilon)$$

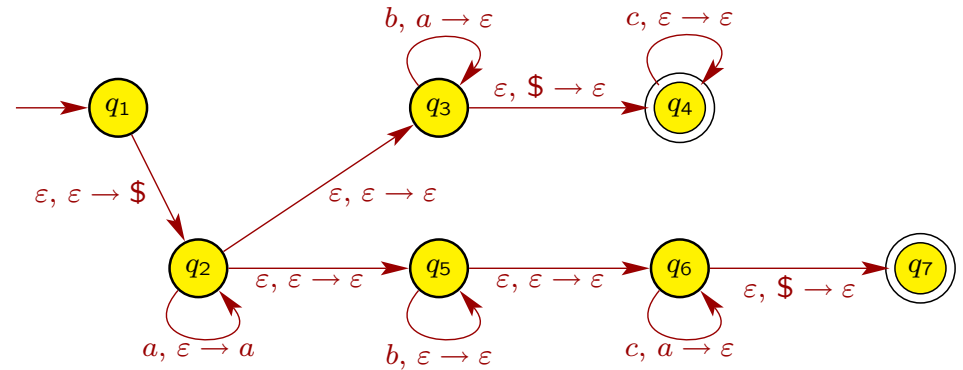
allows for nondeterminism:



Multiple choices when read input symbol is a and pop stack symbol b

ϵ -transitions

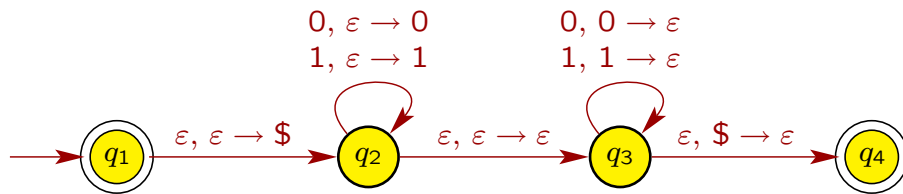
Example: PDA for language $\{ a^i b^j c^k \mid i, j, k \geq 0 \text{ and } i = j \text{ or } i = k \}$



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 $\{ ww^R \mid w \in \{0, 1\}^* \}$



PDA works as follows:

- $q_1 \rightarrow q_2$: First puts $\$$ on stack to mark bottom
- $q_2 \rightarrow q_2$: Reads in first half w of string, pushing it onto stack
- $q_2 \rightarrow q_3$: Guesses that it has reached the middle of the string
- $q_3 \rightarrow q_3$: Reads second half w^R of string, matching symbols from first half in reverse order
- $q_3 \rightarrow q_4$: Makes sure that no more 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 $L = L(G)$ for some CFG G , then $L = L(M)$ for some PDA M .

• Lemma 2.27

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

We will only show how the first lemma works.

Lemma 2.21

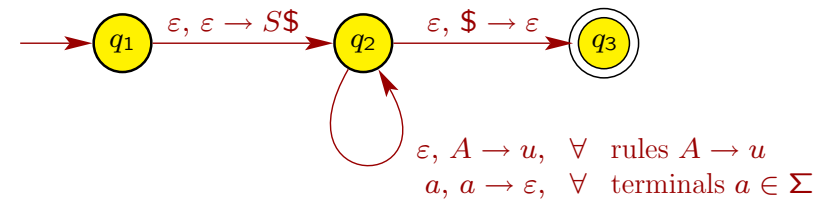
If $L = L(G)$ for some CFG G , then $L = L(M)$ for some PDA M .

Proof Idea:

- Need to show that given CFG G , we can find PDA M that recognizes the same language that G generates.
- Basic idea is to build PDA that simulates a leftmost derivation.
- For example, consider CFG $G = (V, \Sigma, R, S)$
 - Variables $V = \{S\}$
 - Terminals $\Sigma = \{0, 1\}$
 - Rules: $S \rightarrow 0S0S \mid 1S0 \mid 1$
- Leftmost derivation of string $010110 \in L(G)$:

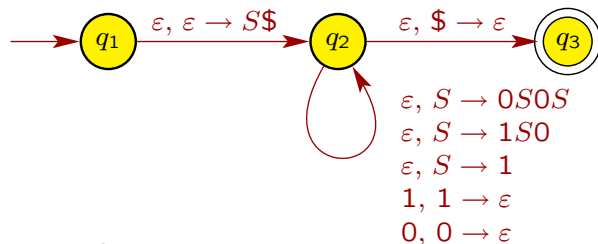
$$S \Rightarrow 0S0S \Rightarrow 010S \Rightarrow 0101S0 \Rightarrow 010110$$

- Create PDA for a CFG as follows:



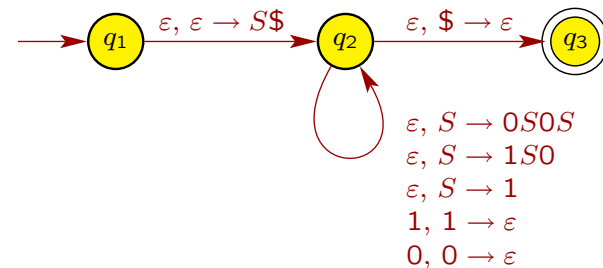
- PDA works as follows:
 1. Pushes $\$$ and then S on the stack, where S is start variable.
 2. Repeats following until stack empty
 - (a) If top of stack is variable $A \in V$, then replace A by some $u \in (\Sigma \cup V)^*$, where $A \rightarrow u$ is a rule in R .
 - (b) If top of stack is terminal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

- Recall CFG: $S \rightarrow 0S0S \mid 1S0 \mid 1$
- 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 “ $\$$ ”
 - ▲ $\Gamma = \{S, 0, 1, \$\}$.
- PDA simulates a leftmost derivation using CFG
 - ▲ Pushes RHS of rules in reverse order onto stack.

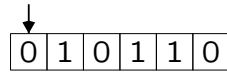
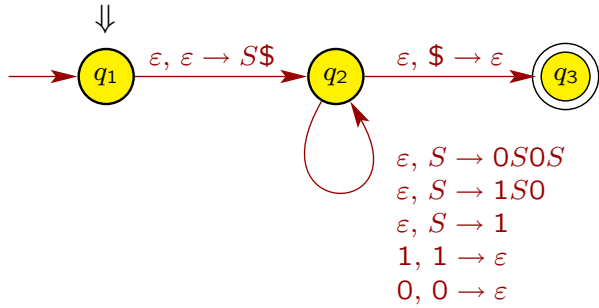
- Recall CFG: $S \rightarrow 0S0S \mid 1S0 \mid 1$
- Corresponding PDA:



- Recall leftmost derivation of string $010110 \in L(G)$:

$$S \Rightarrow 0S0S \Rightarrow 010S \Rightarrow 0101S0 \Rightarrow 010110$$
- Let's now process string 010110 on PDA.

0. Start in state q_1 with 010110 on input tape and empty stack.



Input string

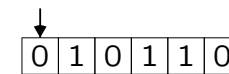
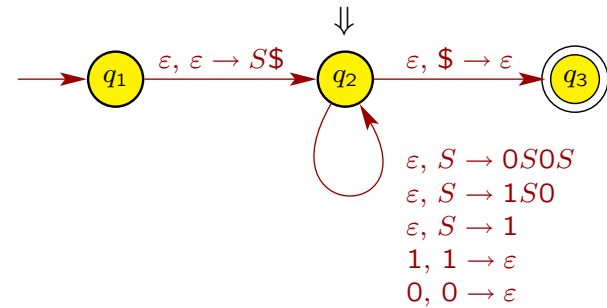


Stack

Leftmost derivation of string 010110 $\in A$:

$$S \Rightarrow OSOS \Rightarrow 010S \Rightarrow 0101SO \Rightarrow 010110$$

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



Input string

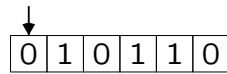
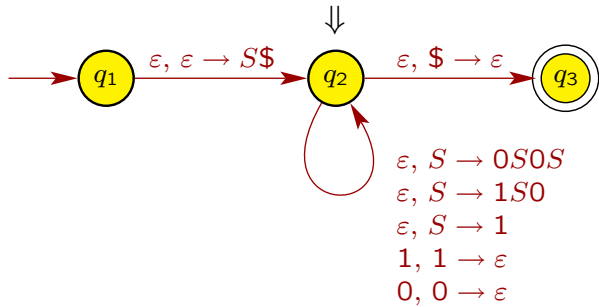


Stack

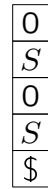
Leftmost derivation of string 010110 $\in A$:

$$\underline{S} \Rightarrow OSOS \Rightarrow 010S \Rightarrow 0101SO \Rightarrow 010110$$

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



Input string

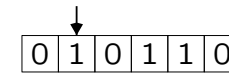
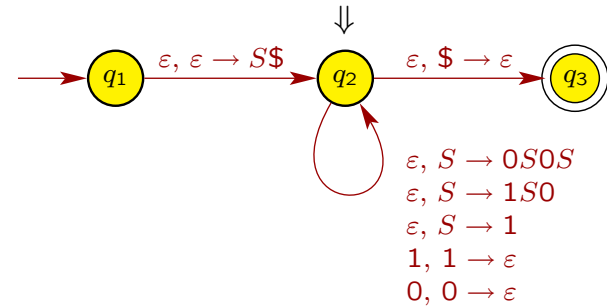


Stack

Leftmost derivation of string 010110 $\in L(G)$:

$$S \Rightarrow \underline{OSOS} \Rightarrow 010S \Rightarrow 0101SO \Rightarrow 010110$$

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



Input string

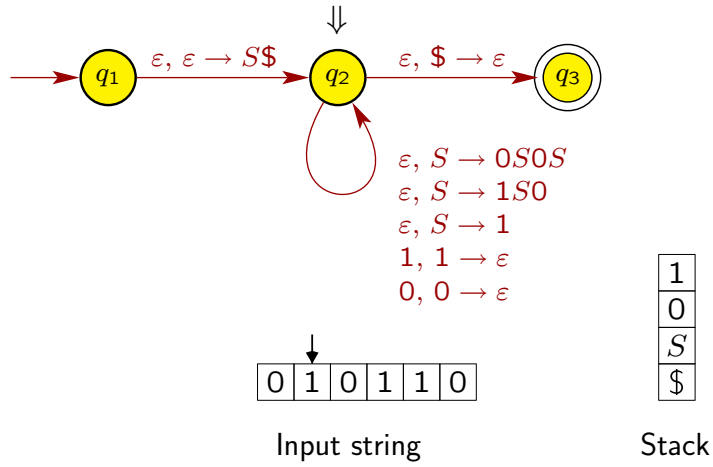


Stack

Leftmost derivation of string 010110 $\in L(G)$:

$$S \Rightarrow \underline{OSOS} \Rightarrow 010S \Rightarrow 0101SO \Rightarrow 010110$$

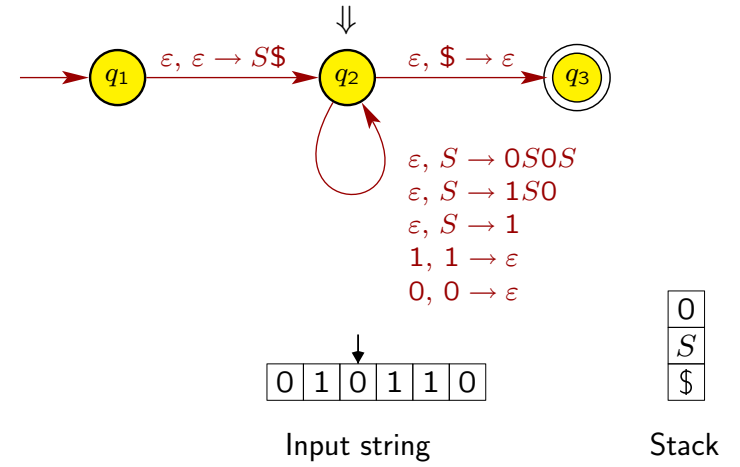
4. Read nothing, pop S , push 1, and return to q_2 .



Leftmost derivation of string $010110 \in L(G)$:

$$S \Rightarrow OSOS \Rightarrow \underline{010S} \Rightarrow 0101SO \Rightarrow 010110$$

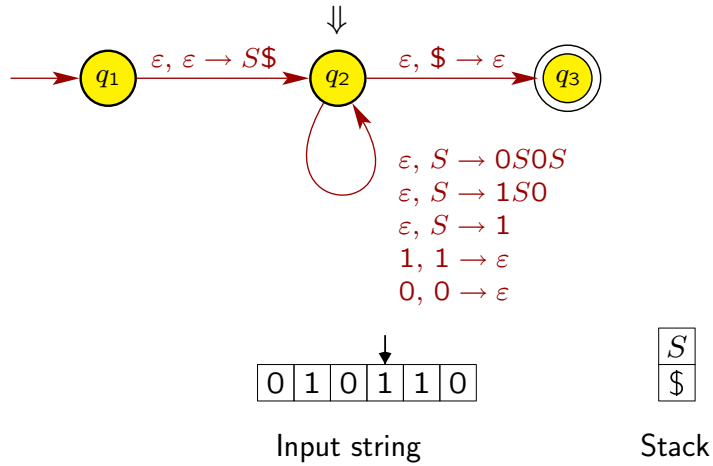
5. Read 1, pop 1, push nothing, and return to q_2 .



Leftmost derivation of string $010110 \in L(G)$:

$$S \Rightarrow OSOS \Rightarrow \underline{010S} \Rightarrow 0101SO \Rightarrow 010110$$

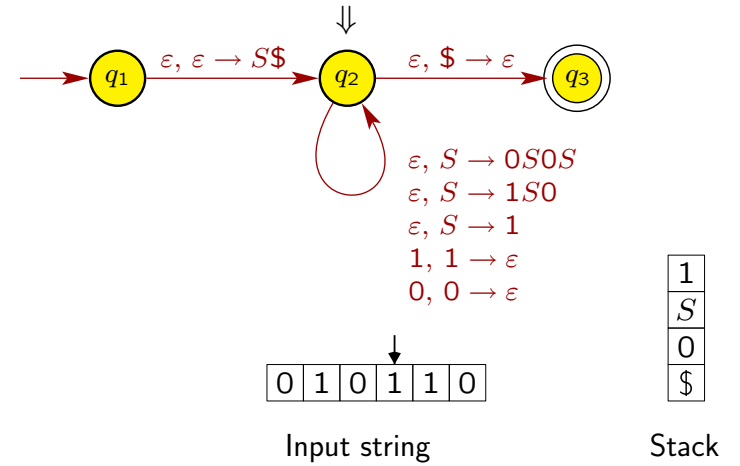
6. Read 0, pop 0, push nothing, and return to q_2 .



Leftmost derivation of string $010110 \in L(G)$:

$$S \Rightarrow OSOS \Rightarrow \underline{010S} \Rightarrow 0101SO \Rightarrow 010110$$

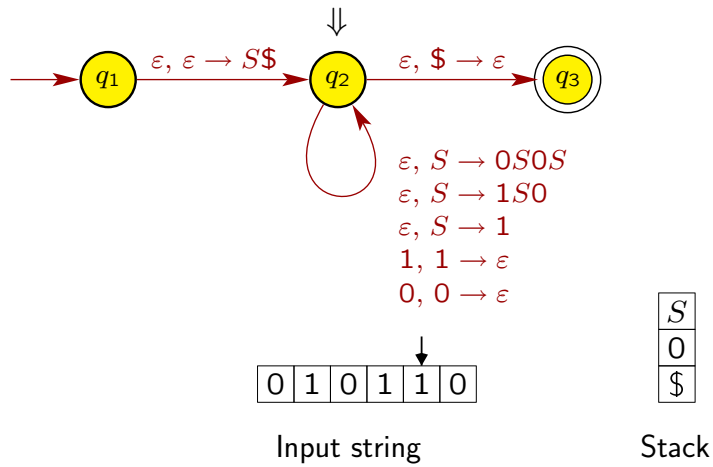
7. Read nothing, pop S , push $1SO$, and return to q_2 .



Leftmost derivation of string $010110 \in L(G)$:

$$S \Rightarrow OSOS \Rightarrow 010S \Rightarrow \underline{0101SO} \Rightarrow 010110$$

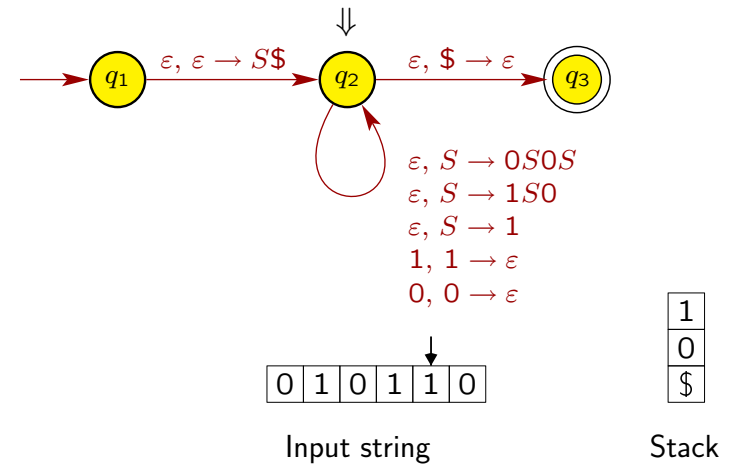
8. Read 1, pop 1, push nothing, and return to q_2 .



Leftmost derivation of string $010110 \in L(G)$:

$$S \Rightarrow OSOS \Rightarrow 010S \Rightarrow \underline{0101SO} \Rightarrow 010110$$

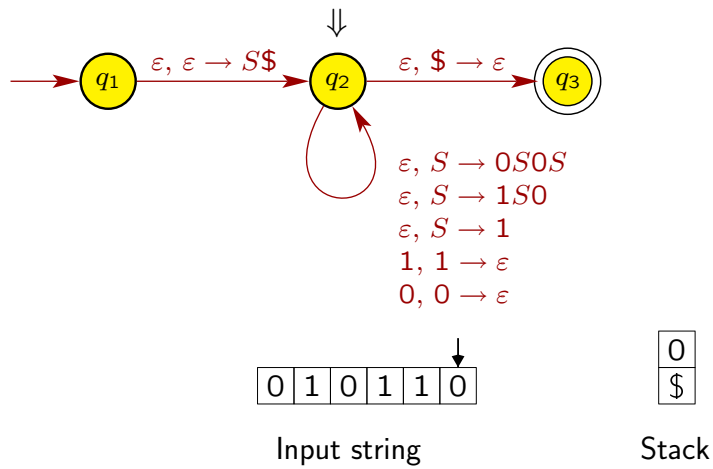
9. Read nothing, pop S , push 1, and return to q_2 .



Leftmost derivation of string $010110 \in L(G)$:

$$S \Rightarrow OSOS \Rightarrow 010S \Rightarrow 0101SO \Rightarrow \underline{010110}$$

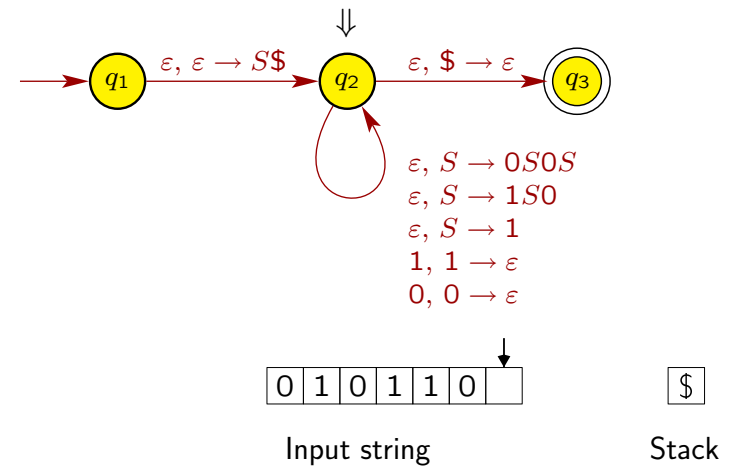
10. Read 1, pop 1, push nothing, and return to q_2 .



Leftmost derivation of string $010110 \in L(G)$:

$$S \Rightarrow OSOS \Rightarrow 010S \Rightarrow 0101SO \Rightarrow \underline{010110}$$

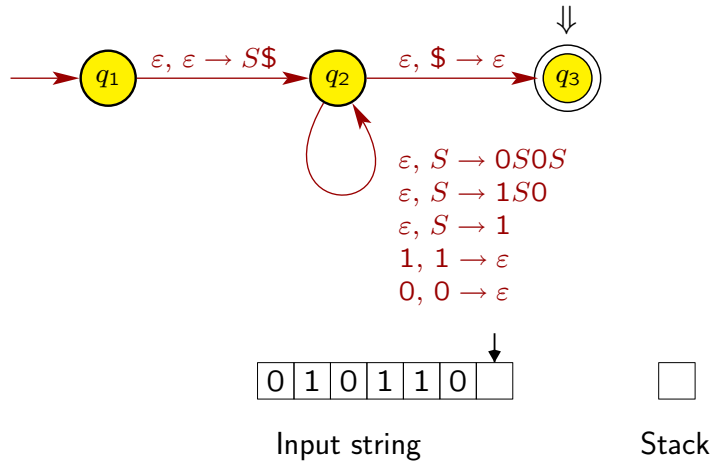
11. Read 0, pop 0, push nothing, and return to q_2 .



Leftmost derivation of string $010110 \in L(G)$:

$$S \Rightarrow OSOS \Rightarrow 010S \Rightarrow 0101SO \Rightarrow \underline{010110}$$

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

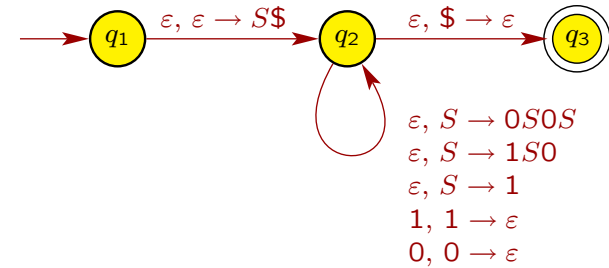


Leftmost derivation of string $010110 \in L(G)$:

$$S \Rightarrow OSOS \Rightarrow 01OS \Rightarrow 0101SO \Rightarrow \underline{010110}$$

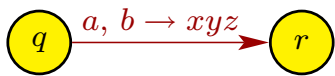
Constructed PDA is Not Compliant

- Recall CFG: $S \rightarrow OSOS \mid 1SO \mid 1$
- Corresponding PDA:

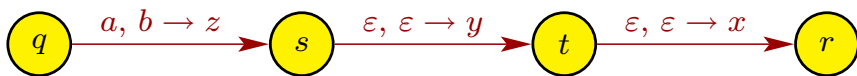


- Problem:** pushing **strings** onto stack instead of single symbols, which is not allowed in PDA specification.

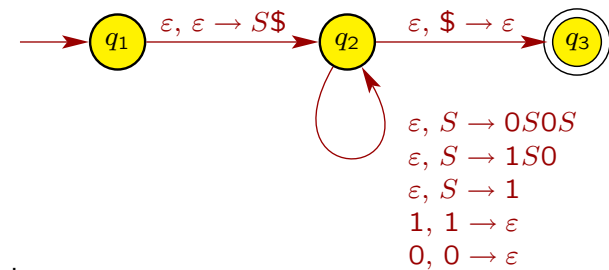
Solution: Add Extra States as Needed



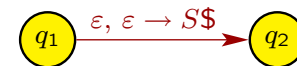
becomes



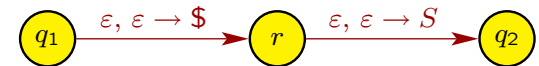
- For example, in our PDA



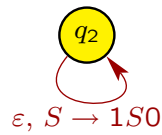
we replace



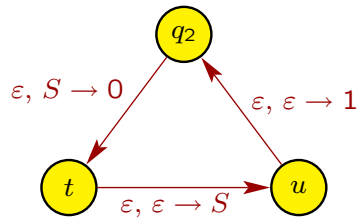
with



- Also, replace

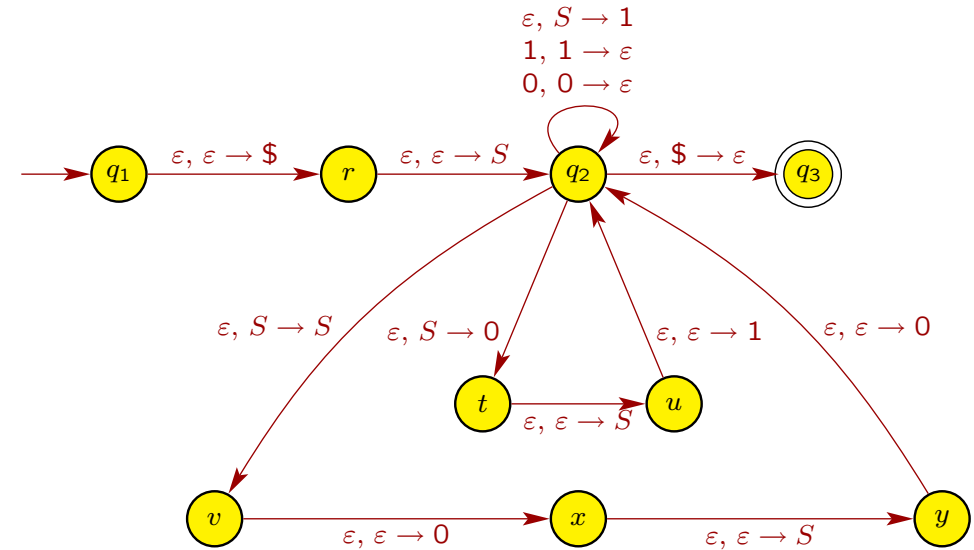


with



- Also need to replace arc from q_2 back to itself with label " $\epsilon, S \rightarrow 0S0S$ ".

- So our final PDA from the CFG is



Pumping Lemma for CFLs

- We previously saw a pumping lemma for regular languages.
- There is an analogous result for context-free languages.
- **Key Idea:** In a derivation of a long string, variables are repeated.
- Consider CFG G with rules

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

- A derivation using G :

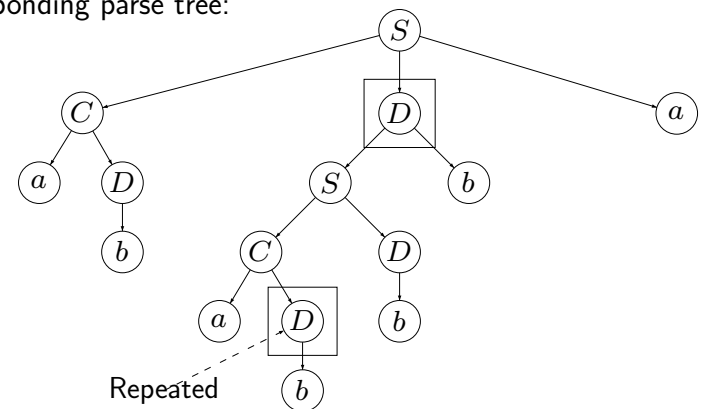
$$\begin{aligned}
 S &\Rightarrow CDa \Rightarrow aDDa \Rightarrow ab\underline{D}a \Rightarrow abSba \Rightarrow abCDba \\
 &\Rightarrow aba\underline{D}Dba \Rightarrow ababDba \Rightarrow ababbba
 \end{aligned}$$

Repeated Variable in Path of Parse Tree

- Derivation of string $ababbba \in L(G)$:

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

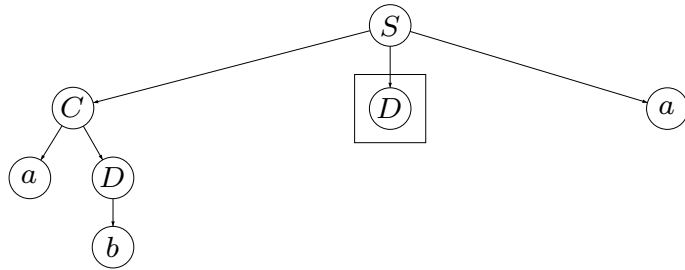
- Corresponding parse tree:



Examine Top of Tree

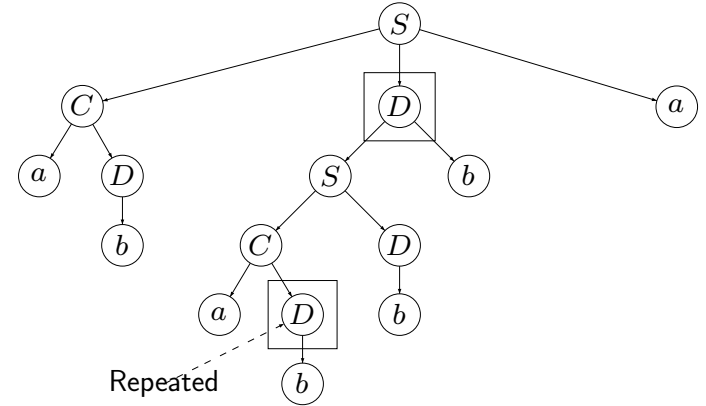
- $S \xrightarrow{*} abDa$ since

$$S \Rightarrow CDa \Rightarrow aDDa \Rightarrow abDa$$



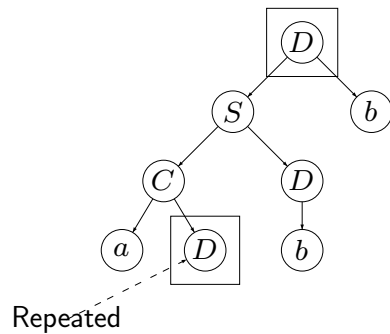
Variable D is Repeated

- $D \Rightarrow Sb \Rightarrow CDb \Rightarrow aDDb \Rightarrow aDbb$
- $D \Rightarrow b$



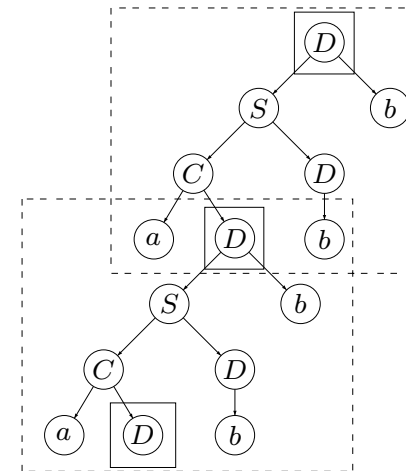
Variable D Repeated: $D \xrightarrow{*} aDbb$

- Repeated part: $D \Rightarrow Sb \Rightarrow CDb \Rightarrow aDDb \Rightarrow aDbb$
so $D \xrightarrow{*} aDbb$



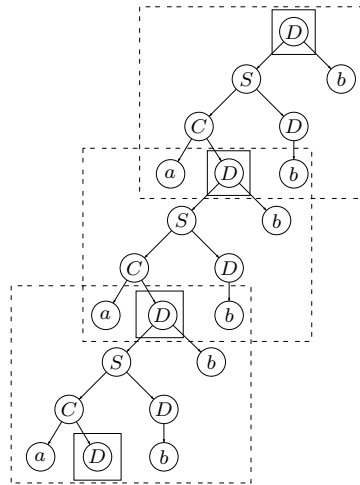
Can Repeat $D \xrightarrow{*} aDbb$

- $D \xrightarrow{*} aDbb \xrightarrow{*} aaDbbbb = (a)^2D(bb)^2$



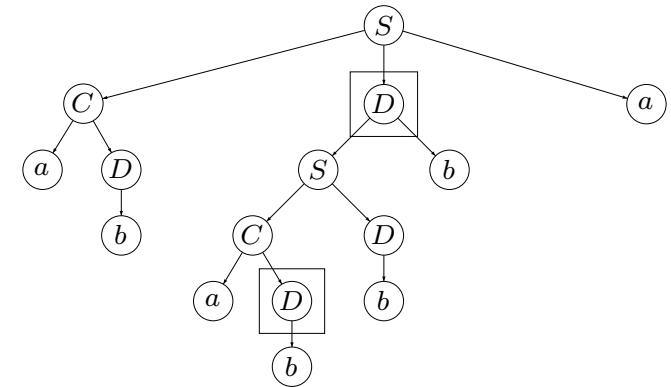
Can Repeat $D \xRightarrow{*} aDbb$ Many Times

- $D \xRightarrow{*} aDbb \xRightarrow{*} aaDbbbb \xRightarrow{*} aaaDbbbbb = (a)^3 D (bb)^3$



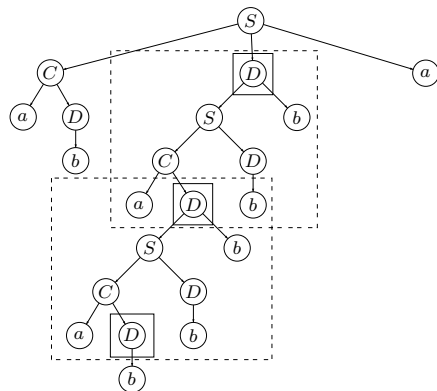
Use $D \xRightarrow{*} aDbb$ Once

- From $S \xRightarrow{*} abDa$, $D \xRightarrow{*} aDbb$ and $D \xRightarrow{*} b$, can derive
 $S \xRightarrow{*} abDa \xRightarrow{*} abaDbbba \xRightarrow{*} ababbba$
 $= ab(a)^1 b(bb)^1 a \in L(G)$

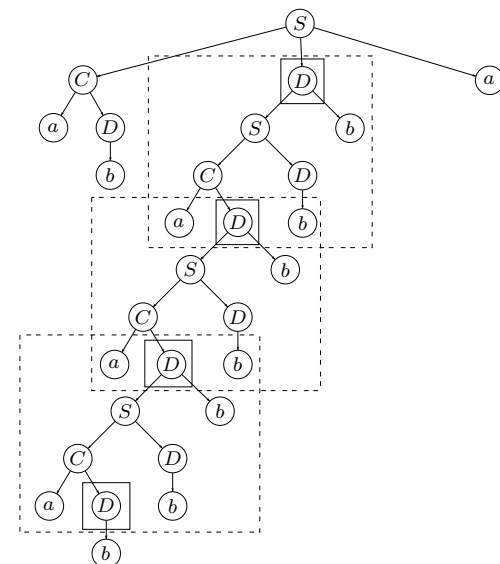


Use $D \xRightarrow{*} aDbb$ Twice

- From $S \xRightarrow{*} abDa$, $D \xRightarrow{*} aDbb$ and $D \xRightarrow{*} b$, can derive
 $S \xRightarrow{*} abDa \xRightarrow{*} abaDbbba \xRightarrow{*} abaaDbbbbb \xRightarrow{*} abaabbbba$
 $= ab(a)^2 b(bb)^2 a \in L(G)$

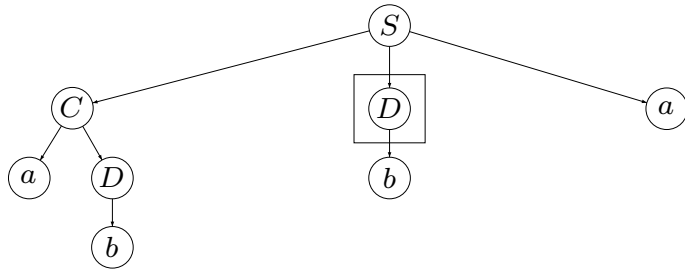


- Using $D \xRightarrow{*} aDbb$ 3 times yields $ab(a)^3 b(bb)^3 a \in L(G)$



Use $D \xRightarrow{*} aDbb$ Zero Times

- From $S \xRightarrow{*} abDa$, $D \xRightarrow{*} aDbb$ and $D \xRightarrow{*} b$, can derive $S \xRightarrow{*} abDa \xRightarrow{*} abba = ab(a)^0 b(bb)^0 a \in L(G)$



Can Use $D \xRightarrow{*} aDbb$ Any Number of Times

- In general, using

- $S \xRightarrow{*} abDa$
- $D \xRightarrow{*} aDbb$
- $D \xRightarrow{*} b$

can derive

$$S \xRightarrow{*} abDa \xRightarrow{*} ab(a)^i D(bb)^i a \xRightarrow{*} ab(a)^i b(bb)^i a \in L(G)$$

for each $i \geq 0$.

- This is a form of pumping:

$$\underbrace{ab}_u (\underbrace{a}_v)^i \underbrace{b}_x (\underbrace{bb}_y)^i \underbrace{a}_z$$

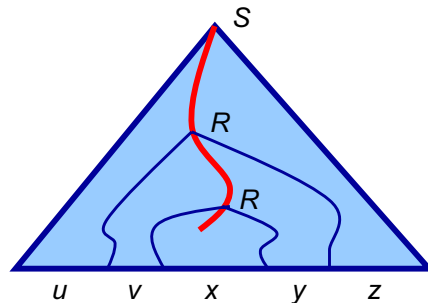
and $uv^i xy^i z \in L(G)$ for each $i \geq 0$.

- We can do the pumping because $D \xRightarrow{*} aDbb$.

Pumping Holds More Generally

- Suppose CFG for language L has

- $S \xRightarrow{*} uRz$ for $u, z \in \Sigma^*$.
- $R \xRightarrow{*} vRy$ for $v, y \in \Sigma^*$.
- $R \xRightarrow{*} x$ for $x \in \Sigma^*$.



- Then

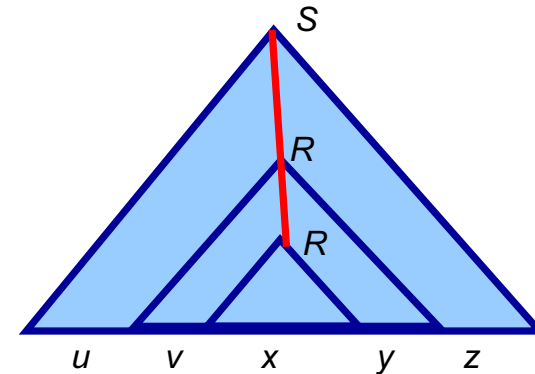
$$S \xRightarrow{*} uRz \xRightarrow{*} uvRyz \xRightarrow{*} uvxyz \in L$$

- But also for each $i \geq 0$,

$$S \xRightarrow{*} uRz \xRightarrow{*} uvRyz \xRightarrow{*} \dots \xRightarrow{*} uv^i Ry^i z \xRightarrow{*} uv^i xy^i z \in L$$

Pumping a Parse Tree

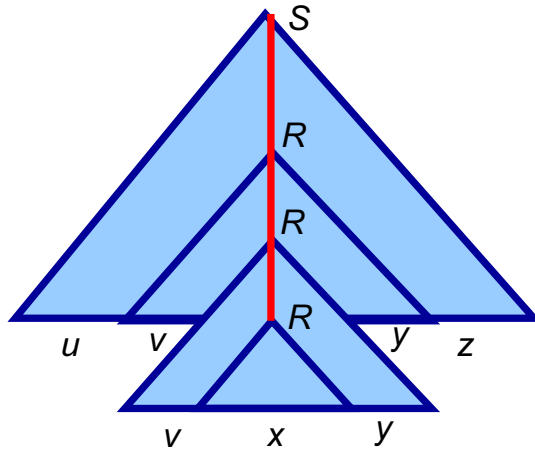
- Consider parse tree of $uvxyz \in L$



- If we repeat the R - R part, we get ...

Pumping Up a Parse Tree

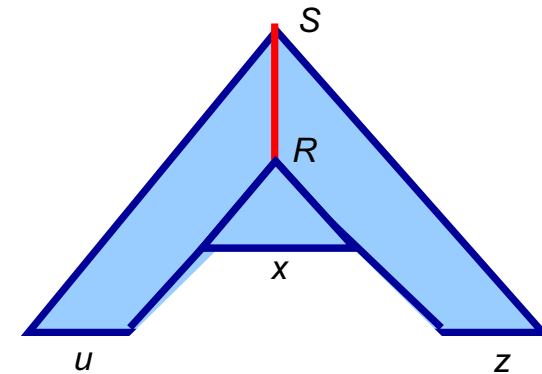
- $uv^2xy^2z \in L$



- If we remove the R - R part, we get ...

Pumping Down a Parse Tree

- $uxz \in L$



When Can We Pump?

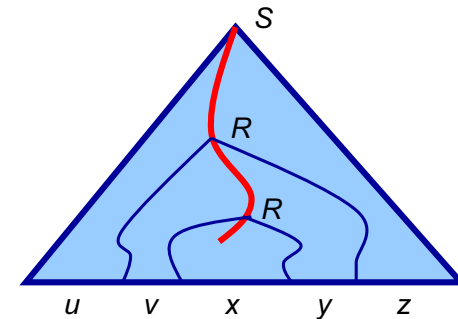
Idea:

- If we can prove the existence of derivation of string in CFL L in which $R \xRightarrow{*} vRy$,
- then “ v - y pumping” holds:

$$R \xRightarrow{*} vRy \xRightarrow{*} v^2Ry^2 \xRightarrow{*} v^3Ry^3 \xRightarrow{*} \dots$$

- We can prove this existence if the parse-tree is tall enough.

Can Pump a Parse Tree If It's Tall Enough



- If string $s = uvxyz \in L$ is long, then its parse-tree is tall.
- Hence, there is a path on which a variable R repeats.
- We can pump this R - R part, i.e., $R \xRightarrow{*} vRy$.
- If height of tree $\geq |V| + 2$, where $|V|$ is number of variables in CFG,
 - then \exists repeated variable on longest path from root to leaf.

Non-CFL

Remark: CFL pumping lemma mainly used to show certain languages are **not** CFL.

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

Proof.

- Suppose that B is CFL.
- Let $p \geq 1$ be the pumping length.
- Consider string $s = a^p b^p c^p \in B$, and note that $|s| = 3p \geq p$.
- Pumping Lemma: $s = uvxyz = a^p b^p c^p$,
 - $uv^i x y^i z \in B$ for all $i \geq 0$
 - $|vy| \geq 1$, $|vxy| \leq p$

- Recall $s = uvxyz = \underbrace{aa \cdots a}_p \underbrace{bb \cdots b}_p \underbrace{cc \cdots c}_p$.
- Possibilities for v, x, y with $|vy| \geq 1$ and $|vxy| \leq p$:
 1. 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 since $|vy| \geq 1$.
 - Hence, $uv^2xy^2z \notin B$.
 2. Strings v and y are not both uniform.
 - Then uv^2xy^2z will not be $a \cdots ab \cdots bc \cdots c$.
 - Hence, $uv^2xy^2z \notin B$.
- Thus, no options for vxy such that $uv^i x y^i z \in B$ for all i .
- This is a contradiction, so $B = \{a^n b^n c^n \mid n \geq 0\}$ is not a CFL.

Prove $C = \{a^i b^j c^k \mid 0 \leq i \leq j \leq k\}$ is not CFL

- Suppose C is CFL, and let p be pumping length.
- Consider string $s = a^p b^p c^p \in C$, so $|s| = 3p \geq p$.
- Pumping Lemma: $s = uvxyz$ and $uv^i x y^i z \in C$ for every $i \geq 0$.
- Pumping Lemma: $|vy| \geq 1$ and $|vxy| \leq p$.
- So vxy can't contain 3 different types of symbols.
- Two options for v, x, y satisfying $|vy| \geq 1$ and $|vxy| \leq p$:
 1. If $vxy \in a^*b^*$, then
 - string uv^2xy^2z doesn't have enough c 's
 - Hence, $uv^2xy^2z \notin C$
 2. If $vxy \in b^*c^*$, then
 - string $uv^0xy^0z = uxz$ has too many a 's
 - Hence, $uv^0xy^0z \notin C$
- Contradiction, so C is not a CFL.

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

- Suppose D is CFL, and let p be pumping length.
- Take $s = 0^p 1^p 0^p 1^p \in D$, so $|s| = 4p \geq p$.
- Pumping lemma: $s = uvxyz$ with $|vy| \geq 1$ and $|vxy| \leq p$:
 1. If vxy is entirely left of middle of $0^p 1^p 0^p 1^p$,
 - then second half of uv^2xy^2z starts with "1"
 - so can't write uv^2xy^2z as ww .
 2. Similar reasoning: if vxy is entirely right of middle of $0^p 1^p 0^p 1^p$,
 - then $uv^2xy^2z \notin D$
 3. If vxy straddles middle of $0^p 1^p 0^p 1^p$,
 - then $uv^0xy^0z = uxz$ equals $0^p 1^i 0^j 1^p \notin D$ (because i or $j < p$)
- Contradiction, so D is not context-free.

Remarks on CFLs

- Using the CFL pumping lemma is more difficult than the pumping lemma for regular languages.
 - Carefully choose the string s to get contradiction.
 - Carefully show all possibilities give contradictions.
- What about closure of CFLs under standard operations?

CFLs Closed Under Union

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 is the new start variable
 - $R_3 = R_1 \cup R_2 \cup \{S_3 \rightarrow S_1, S_3 \rightarrow S_2\}$.

Example of Union of CFLs

- Suppose A_1 has CFG G_1 with rules:

$$S \rightarrow aS \mid bXb$$

$$X \rightarrow ab \mid baXb$$
- Suppose A_2 has CFG G_2 with rules:

$$S \rightarrow Sbb \mid aXba$$

$$X \rightarrow b \mid XaX$$
- Then $A_1 \cup A_2$ has CFG G_3 with start variable S_3 and rules:

$$S_3 \rightarrow S_1 \mid S_2$$

$$S_1 \rightarrow aS_1 \mid bX_1b$$

$$X_1 \rightarrow ab \mid baX_1b$$

$$S_2 \rightarrow S_2bb \mid aX_2ba$$

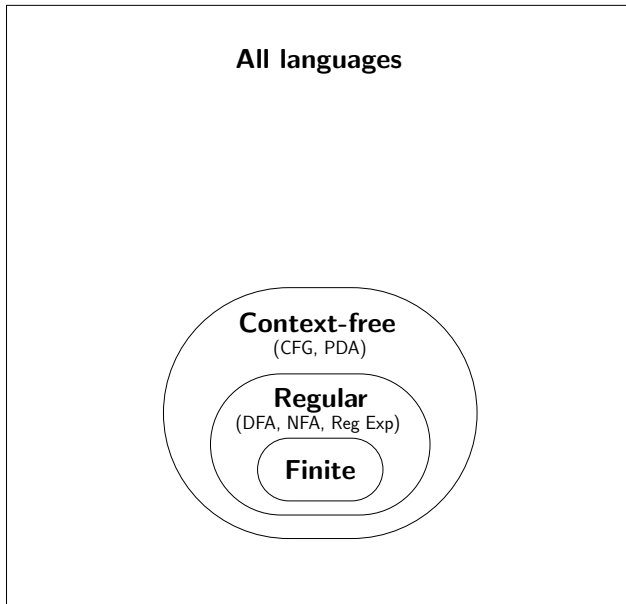
$$X_2 \rightarrow b \mid X_2aX_2$$

Other Closure Properties of CFLs

- Let A_1 and A_2 be two CFLs.
- Can prove that
 - concatenation $A_1 \circ A_2$ is always CFL
 - Kleene-star A_1^* is always CFL
 - intersection $A_1 \cap A_2$ is not necessarily CFL
 - 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: $A \rightarrow BC$ or $A \rightarrow x$, with $A \in V$, $B, C \in V - \{S\}$, $x \in \Sigma$. Also allow rule $S \rightarrow \varepsilon$.
- Regular \Rightarrow CFL, but CFL $\not\Rightarrow$ Regular.
- Pushdown automaton has stack for additional memory
- Equivalence of PDAs and CFGs
- Pumping lemma for CFLs
 - Repeat part of parse tree corresponding to repeated variable
 - Used to prove certain languages are non-CFL
- CFLs closed under union, Kleene star, concatenation
- CFLs **not** closed under intersection, complementation