UMB CS 420

Pushdown Automata (PDAs)

Wednesday, March 20, 2024
Announcements

• HW 5 out
  • Due Mon 3/25 12pm noon
A context-free grammar is a 4-tuple \((V, \Sigma, R, S)\), where

1. \(V\) is a finite set called the **variables**,
2. \(\Sigma\) is a finite set, disjoint from \(V\), called the **terminals**,
3. \(R\) is a finite set of **rules**, with each rule being a variable and a string of variables and terminals, and
4. \(S \in V\) is the start variable.

\[ V = \{A, B\}, \]
\[ \Sigma = \{0, 1, \#\}, \]
\[ S = A, \]
Generating Strings with a CFG

Grammar $G_1 = (V, \Sigma, R, S)$

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

Strings in CFG’s language = all possible generated / derived strings

$L(G_1)$ is $\{0^n#1^n | n \geq 0\}$

A CFG generates a string, by repeatedly applying substitution rules:

Example:

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

This sequence of steps is called a derivation
Last Time:

Derivations: Formally

Let $G = (V, \Sigma, R, S)$

**Single-step**

$$\alpha A \beta \Rightarrow_G \alpha \gamma \beta$$

Where:

$\alpha, \beta \in (V \cup \Sigma)^*$ — sequence of terminals or variables

$A \in V$ — Variable

$A \rightarrow \gamma \in R$ — Rule

A context-free grammar is a 4-tuple $(V, \Sigma, R, S)$, where
1. $V$ is a finite set called the variables,
2. $\Sigma$ is a finite set, disjoint from $V$, called the terminals,
3. $R$ is a finite set of rules, with each rule being a variable and a string of variables and terminals, and
4. $S \in V$ is the start variable.
Last Time:

Derivations: Formally

Let $G = (V, \Sigma, R, S)$

**Single-step**

$$\alpha A \beta \Rightarrow_G \alpha \gamma \beta$$

Where:

- $\alpha, \beta \in (V \cup \Sigma)^*$  
  sequence of terminals or variables
- $A \in V$  
  Variable
- $A \rightarrow \gamma \in R$  
  Rule

**Multi-step (recursively defined)**

**Base case:**

$$\alpha \Rightarrow^*_G \alpha \quad (0 \text{ steps})$$

**Recursive case:**  

$$\alpha \Rightarrow^*_G \gamma$$

Where:

- $\alpha \Rightarrow^*_G \beta$  
  (smaller) Recursive “call”
- $\beta \Rightarrow^*_G \gamma$

**Single step**

A context-free grammar is a 4-tuple $(V, \Sigma, R, S)$, where:

1. $V$ is a finite set called the **variables**,  
2. $\Sigma$ is a finite set, disjoint from $V$, called the **terminals**,  
3. $R$ is a finite set of **rules**, with each rule being a variable and a string of variables and terminals, and  
4. $S \in V$ is the start variable.
Formal Definition of a CFL

A context-free grammar is a 4-tuple \( (V, \Sigma, R, S) \), where
1. \( V \) is a finite set called the variables,
2. \( \Sigma \) is a finite set, disjoint from \( V \), called the terminals,
3. \( R \) is a finite set of rules, with each rule being a variable and a string of variables and terminals, and
4. \( S \in V \) is the start variable.

\[
G = (V, \Sigma, R, S)
\]

The language of a grammar \( G \) is ...

... all possible sequences of terminal symbols (i.e., strings) ...

... that can be generated with rules of grammar \( G \)

\[
L(G) = \left\{ w \in \Sigma^* \mid S \xrightarrow[G]{*} w \right\}
\]

If a CFG generates all strings in a language \( L \), then \( L \) is a context-free language (CFL)
Designing Grammars : Basics

1. Think about what you want to “link” together
   - E.g., $0^n1^n$
     - $A \rightarrow 0A1$
     - # 0s and # 1s are “linked”
   - E.g., XML
     - ELEMENT $\rightarrow \langle\text{TAG}\rangle\text{CONTENT}\langle/\text{TAG}\rangle$
     - Start and end tags are “linked”

2. Start with small grammars and then combine
   - just like with FSMs, and programming!
Example: Creating CFG

alphabet $\Sigma$ is $\{0, 1\}$

$\{ w \mid w \text{ starts and ends with the same symbol} \}$

1) come up with examples: In the language: $010, 101, 11011 \quad 1, 0 ? \quad \checkmark$

Not in the language: $10, 01, 110 \quad \varepsilon ? \quad \times$

2) Create CFG:

$S \rightarrow 0M0 \mid 1M1 \mid 0 \mid 1$

Needed Rules:

"start/end symbol are “linked” (ie, same); middle can be anything"

$M \rightarrow MT \mid \varepsilon$

"middle: all possible terminals, repeated (ie, all possible strings)"

$T \rightarrow 0 \mid 1$

"all possible terminals"

3) Check CFG: generates examples in the language; does not generate examples not in language
## Regular Language vs CFL Comparison

<table>
<thead>
<tr>
<th>Regular Languages</th>
<th>Context-Free Languages (CFLs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular Expression</td>
<td>Context-Free Grammar (CFG)</td>
</tr>
<tr>
<td>describes a Regular Lang</td>
<td>describes a CFL</td>
</tr>
</tbody>
</table>

*Note: The table compares regular languages and context-free languages, highlighting how regular expressions describe regular languages and context-free grammars describe context-free languages.*
## Regular Language vs CFL Comparison

<table>
<thead>
<tr>
<th>Regular Languages</th>
<th>Context-Free Languages (CFLs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular Expression</td>
<td>Context-Free Grammar (CFG)</td>
</tr>
<tr>
<td><strong>describes</strong> a Regular Lang</td>
<td><strong>describes</strong> a CFL</td>
</tr>
<tr>
<td>Finite State Automaton (FSM)</td>
<td>???</td>
</tr>
<tr>
<td><strong>recognizes</strong> a Regular Lang</td>
<td><strong>recognizes</strong> a CFL</td>
</tr>
</tbody>
</table>
## Regular Language vs CFL Comparison

<table>
<thead>
<tr>
<th>Regular Languages</th>
<th>Context-Free Languages (CFLs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular Expression</td>
<td>Context-Free Grammar (CFG)</td>
</tr>
<tr>
<td>describes a Regular Lang</td>
<td>describes a CFL</td>
</tr>
<tr>
<td>Finite State Automaton (FSM)</td>
<td>Push-down Automata (PDA)</td>
</tr>
<tr>
<td>recognizes a Regular Lang</td>
<td>recognizes a CFL</td>
</tr>
</tbody>
</table>

**Definitions:**
- **Regular Language:** A language that can be described by a regular expression or recognized by a finite state automaton.
- **Context-Free Language (CFL):** A language that can be described by a context-free grammar or recognized by a push-down automaton.

**Theorems:**
- Theorem (def): A regular expression describes a regular language.
- Theorem (thm): A context-free grammar describes a context-free language.
- Theorem (thm): A finite state automaton recognizes a regular language.
- Theorem (thm): A push-down automaton recognizes a context-free language.
## Regular Language vs CFL Comparison

<table>
<thead>
<tr>
<th>Regular Languages</th>
<th>Context-Free Languages (CFLs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular Expression</td>
<td>Context-Free Grammar (CFG)</td>
</tr>
<tr>
<td><em>describes</em> a Regular Lang</td>
<td><em>describes</em> a CFL</td>
</tr>
<tr>
<td>Finite State Automaton (FSM)</td>
<td><strong>Push-down Automata</strong> (PDA)</td>
</tr>
<tr>
<td><em>recognizes</em> a Regular Lang</td>
<td><em>recognizes</em> a CFL</td>
</tr>
</tbody>
</table>

**Proved:**

- Regular Lang $\Leftrightarrow$ Regular Expr
- CFL $\Leftrightarrow$ PDA
Pushdown Automata (PDA)

$$\text{PDA} = \text{NFA} + \text{a stack}$$
What is a Stack?

- A **restricted** kind of (infinite!) memory
- Access to top element only
- 2 Operations only: **push**, **pop**
Pushdown Automata (PDA)

• **PDA = NFA + a stack**
  • Infinite memory
  • read/write top location only
    • Push/pop
An Example PDA

A PDA transition has 3 parts:
- Read
- Pop
- Push

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

This machine can only pop $\$$ (and accept) when stack is empty, i.e., when # 0s = # 1s
A *pushdown automaton* is a 6-tuple \( (Q, \Sigma, \Gamma, \delta, q_0, F) \), where \( Q \), \( \Sigma \), \( \Gamma \), \( \delta \), \( q_0 \), and \( F \) are all finite sets, and

1. \( Q \) is the set of states,
2. \( \Sigma \) is the input alphabet,
3. \( \Gamma \) is the stack alphabet,
4. \( \delta: Q \times \Sigma \varepsilon \times \Gamma \varepsilon \rightarrow \mathcal{P}(Q \times \Gamma \varepsilon) \) is the transition function,
5. \( q_0 \in Q \) is the initial state, and
6. \( F \subseteq Q \) is the set of accept states.

Non-deterministic! Result of a step is set of (State, Stack Char) pairs.
\[ Q = \{q_1, q_2, q_3, q_4\}, \]
\[ \Sigma = \{0, 1\}, \]
\[ \Gamma = \{0, \$\}, \]
\[ F = \{q_1, q_4\}, \]

A pushdown automaton is a 6-tuple \((Q, \Sigma, \Gamma, \delta, q_0, F)\), where \(Q, \Sigma, \Gamma, \) and \(F\) are all finite sets, and

1. \(Q\) is the set of states,
2. \(\Sigma\) is the input alphabet,
3. \(\Gamma\) is the stack alphabet,
4. \(\delta: Q \times \Sigma \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma)\) is the transition function,
5. \(q_0 \in Q\) is the start state, and
6. \(F \subseteq Q\) is the set of accept states.
\[ Q = \{q_1, q_2, q_3, q_4\}, \]
\[ \Sigma = \{0,1\}, \]
\[ \Gamma = \{0, \$\}, \]
\[ F = \{q_1, q_4\}, \]

and

\( \delta \) is given by the following table, wherein blank entries signify \( \emptyset \).

<table>
<thead>
<tr>
<th>Input:</th>
<th>0</th>
<th>( \varepsilon )</th>
<th>1</th>
<th>( \varepsilon )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack:</td>
<td>0</td>
<td>$</td>
<td>( \varepsilon )</td>
<td>0</td>
</tr>
<tr>
<td>( q_1 )</td>
<td>{ ( q_2, 0 ) }</td>
<td>{ ( q_3, \varepsilon ) }</td>
<td>2</td>
<td>{ ( q_3, \varepsilon ) }</td>
</tr>
</tbody>
</table>

A **pushdown automaton** is a 6-tuple \((Q, \Sigma, \Gamma, \delta, q_0, F)\), where \(Q, \Sigma, \Gamma,\) and \(F\) are all finite sets, and

1. \(Q\) is the set of states,
2. \(\Sigma\) is the input alphabet,
3. \(\Gamma\) is the stack alphabet,
4. \(\delta: Q \times \Sigma \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma)\) is the transition function,
5. \(q_0 \in Q\) is the start state, and
6. \(F \subseteq Q\) is the set of accept states.
\[ Q = \{q_1, q_2, q_3, q_4\}, \]
\[ \Sigma = \{0,1\}, \]
\[ \Gamma = \{0,\$\}, \]
\[ F = \{q_1, q_4\}, \] and

\( \delta \) is given by the following table, wherein blank entries signify \( \emptyset \).

<table>
<thead>
<tr>
<th>Input:</th>
<th>0</th>
<th>( \varepsilon )</th>
<th>1</th>
<th>( \varepsilon )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack:</td>
<td>0</td>
<td>$ \varepsilon $</td>
<td>0</td>
<td>$ \varepsilon $</td>
</tr>
<tr>
<td>( q_1 )</td>
<td>{(q_2, 0)}</td>
<td>{(q_3, \varepsilon)}</td>
<td>2</td>
<td>{(q_4, \varepsilon)}</td>
</tr>
<tr>
<td>( q_2 )</td>
<td>{(q_3, \varepsilon)}</td>
<td>{(q_3, \varepsilon)}</td>
<td>3</td>
<td>{(q_4, \varepsilon)}</td>
</tr>
<tr>
<td>( q_3 )</td>
<td>{(q_2, \varepsilon)}</td>
<td>{(q_2, \varepsilon)}</td>
<td>4</td>
<td>{(q_4, \varepsilon)}</td>
</tr>
<tr>
<td>( q_4 )</td>
<td>{(q_2, \varepsilon)}</td>
<td>{(q_2, \varepsilon)}</td>
<td>5</td>
<td>{(q_4, \varepsilon)}</td>
</tr>
</tbody>
</table>

A *pushdown automaton* is a 6-tuple \((Q, \Sigma, \Gamma, \delta, q_0, F)\), where \(Q, \Sigma, \Gamma, \) and \(F\) are all finite sets, and

1. \(Q\) is the set of states,
2. \(\Sigma\) is the input alphabet,
3. \(\Gamma\) is the stack alphabet,
4. \(\delta: Q \times \Sigma \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma)\) is the transition function,
5. \(q_0 \in Q\) is the start state, and
6. \(F \subseteq Q\) is the set of accept states.
\[ Q = \{ q_1, q_2, q_3, q_4 \}, \]
\[ \Sigma = \{0, 1\}, \]
\[ \Gamma = \{0, \$\}, \]
\[ F = \{ q_1, q_4 \}, \]

and \( \delta \) is given by the following table, wherein blank entries signify \( \emptyset \).

<table>
<thead>
<tr>
<th>Input: ( \epsilon )</th>
<th>0</th>
<th>( \epsilon )</th>
<th>1</th>
<th>( \epsilon )</th>
<th>( \epsilon )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack: ( \epsilon )</td>
<td>( (q_2, \epsilon) )</td>
<td>( (q_3, \epsilon) )</td>
<td>2</td>
<td>( (q_3, \epsilon) )</td>
<td>3</td>
</tr>
</tbody>
</table>

A **pushdown automaton** is a 6-tuple \( (Q, \Sigma, \Gamma, \delta, q_0, F) \), where \( Q, \Sigma, \Gamma, \) and \( F \) are all finite sets, and

1. \( Q \) is the set of states,
2. \( \Sigma \) is the input alphabet,
3. \( \Gamma \) is the stack alphabet,
4. \( \delta: Q \times \Sigma \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma) \) is the transition function,
5. \( q_0 \in Q \) is the start state, and
6. \( F \subseteq Q \) is the set of accept states.
\[
Q = \{q_1, q_2, q_3, q_4\},
\]
\[
\Sigma = \{0, 1\},
\]
\[
\Gamma = \{0, \$\},
\]
\[
F = \{q_1, q_4\},
\]

\(\delta\) is given by the following table, wherein blank entries signify \(\emptyset\).

<table>
<thead>
<tr>
<th>Input:</th>
<th>0</th>
<th>1</th>
<th>(\varepsilon)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack:</td>
<td>0</td>
<td>$</td>
<td>(\varepsilon)</td>
</tr>
<tr>
<td>(q_1)</td>
<td></td>
<td></td>
<td>{(q_2, 0)}</td>
</tr>
<tr>
<td>(q_2)</td>
<td></td>
<td></td>
<td>{(q_3, \varepsilon)}</td>
</tr>
<tr>
<td>(q_3)</td>
<td>{(q_3, \varepsilon)}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(q_4)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A **pushdown automaton** is a 6-tuple \((Q, \Sigma, \Gamma, \delta, q_0, F)\), where \(Q\), \(\Sigma\), \(\Gamma\), and \(F\) are all finite sets, and

1. \(Q\) is the set of states,
2. \(\Sigma\) is the input alphabet,
3. \(\Gamma\) is the stack alphabet,
4. \(\delta: Q \times \Sigma \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma)\) is the transition function,
5. \(q_0 \in Q\) is the start state, and
6. \(F \subseteq Q\) is the set of accept states.
$Q = \{q_1, q_2, q_3, q_4\},$
$\Sigma = \{0, 1\},$
$\Gamma = \{\text{0, } $\}$,$
$F = \{q_1, q_4\}$, and

$\delta$ is given by the following table, wherein blank entries signify $\emptyset$.

<table>
<thead>
<tr>
<th>Input:</th>
<th>0</th>
<th>$\varepsilon$</th>
<th>1</th>
<th>$\varepsilon$</th>
<th>$\varepsilon$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack:</td>
<td>0</td>
<td>$$</td>
<td>$\varepsilon$</td>
<td>0</td>
<td>$$</td>
</tr>
</tbody>
</table>

$q_1$
$q_2$
$q_3$
$q_4$

$\{(q_2, 0)\}$
$\{(q_3, \varepsilon)\}$
$\{(q_3, \varepsilon)\}$
$\{(q_4, \varepsilon)\}$

A pushdown automaton is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where $Q$, $\Sigma$, $\Gamma$, and $F$ are all finite sets, and

1. $Q$ is the set of states,
2. $\Sigma$ is the input alphabet,
3. $\Gamma$ is the stack alphabet,
4. $\delta: Q \times \Sigma \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma)$ is the transition function,
5. $q_0 \in Q$ is the start state, and
6. $F \subseteq Q$ is the set of accept states.
In-class exercise:
Fill in the blanks

\[ Q = \]
\[ \Sigma = \]
\[ \Gamma = \]
\[ F = \]

\( \delta \) is given by the following table, wherein blank entries signify \( \emptyset \).

<table>
<thead>
<tr>
<th>Input</th>
<th>0</th>
<th>1</th>
<th>( \varepsilon )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>0</td>
<td>$</td>
<td>\varepsilon</td>
</tr>
</tbody>
</table>

PDA \( M_3 \) recognizing the language \( \{ w w^R | w \in \{0,1\}^* \} \)

A **pushdown automaton** is a 6-tuple \( (Q, \Sigma, \Gamma, \delta, q_0, F) \), where \( Q, \Sigma, \Gamma, \) and \( F \) are all finite sets, and

1. \( Q \) is the set of states,
2. \( \Sigma \) is the input alphabet,
3. \( \Gamma \) is the stack alphabet,
4. \( \delta : Q \times \Sigma_e \times \Gamma_e \rightarrow \mathcal{P}(Q \times \Gamma_e) \) is the transition function,
5. \( q_0 \in Q \) is the start state, and
6. \( F \subseteq Q \) is the set of accept states.
In-class exercise: Fill in the blanks

\[ Q = \{q_1, q_2, q_3, q_4\}, \]
\[ \Sigma = \{0,1\}, \]
\[ \Gamma = \{0,1,\$\}, \]
\[ F = \{q_4\} \]

\[ \delta \text{ is given by the following table, wherein blank entries signify } \emptyset. \]

<table>
<thead>
<tr>
<th>Input:</th>
<th>0</th>
<th>1</th>
<th>$</th>
<th>\varepsilon</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack:</td>
<td>$</td>
<td>\varepsilon</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>q_1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>q_2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>q_3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>q_4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ \delta(q_1, 0, \varepsilon) = \{q_2, \varepsilon\} \]
\[ \delta(q_1, 1, \varepsilon) = \{q_2, 1\} \]
\[ \delta(q_2, 0, \varepsilon) = \{q_2, 0\} \]
\[ \delta(q_2, 1, \varepsilon) = \{q_2, 1\} \]
\[ \delta(q_3, \varepsilon) = \{(q_3, \varepsilon)\} \]
\[ \delta(q_4, \varepsilon) = \{(q_4, \varepsilon)\} \]

PDA \( M_3 \) recognizing the language \( \{ww^R | w \in \{0,1\}^*\} \)
DFA Computation Rules

**Informally**

Given
- A DFA (~ a “Program”)
- and Input = string of chars, e.g. “1101”

A DFA computation (~ “Program run”):
- **Start** in *start state*

**Repeat:**
- **Read 1 char** from Input, and
- **Change state** according to *transition rules*

**Result** of computation:
- **Accept** if last state is *Accept state*
- **Reject** otherwise

---

**Formally (i.e., mathematically)**

- \( M = (Q, \Sigma, \delta, q_0, F) \)
- \( w = w_1w_2 \cdots w_n \)

A DFA computation is a sequence of states:

- specified by \( \hat{\delta}(q_0, w) \) where:

  - \( M \) **accepts** \( w \) if \( \hat{\delta}(q_0, w) \in F \)
  - \( M \) **rejects** otherwise
DFA Multi-step Transition Function

\[ \hat{\delta} : Q \times \Sigma^* \rightarrow Q \]

- **Domain** (inputs):
  - state \( q \in Q \)
  - string \( w = w_1 w_2 \cdots w_n \) where \( w_i \in \Sigma \)

- **Range** (output):
  - state \( q \in Q \)

(Defined recursively)

**Base case** \[ \hat{\delta}(q, \varepsilon) = q \]

**Recursive Case**
\[ \hat{\delta}(q, w'w_n) = \delta(\hat{\delta}(q, w'), w_n) \]
where \( w' = w_1 \cdots w_{n-1} \)
PDA Computation?

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

A DFA computation is a sequence of states ...

A PDA computation is a **not** just a sequence of states ...

... because the stack contents can change too!
PDA Configurations (IDs)

• **A configuration** (or **ID**) is a “snapshot” of a PDA’s computation

• 3 components \((q, w, \gamma)\):
  - \(q\) = the current state
  - \(w\) = the remaining input string
  - \(\gamma\) = the stack contents

A sequence of configurations represents a PDA computation
PDA Computation, Formally

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

**Single-step**

Before / After configurations

\[(q_1, aw, X\beta) \vdash (q_2, w, \alpha\beta)\]

Read Input  Pop  Less 1 char  Push

if \( \delta(q_1, a, X) \) contains \( (q_2, \alpha) \)

\[ q_1, q_2 \in Q \]
\[ a \in \Sigma \]
\[ w \in \Sigma^* \]
\[ X \in \Gamma \]
\[ \beta, \alpha \in \Gamma^* \]

A configuration \((q, w, \gamma)\) has three components

- \( q \) = the current state
- \( w \) = the remaining input string
- \( \gamma \) = the stack contents

**Multi-step**

- **Base Case**
  
  \( I \vdash^* I \) for any ID \( I \)
  
  0 steps

- **Recursive Case**
  
  \( I \vdash^* J \) if there exists some ID \( K \) such that \( I \vdash K \) and \( K \vdash^* J \)
  
  > 0 steps

This specifies the sequence of configurations for a PDA computation
PDA Running Input String Example

\((q_1, 0011, \varepsilon)\)
PDA Running Input String Example

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

Read 0, push 0
PDA Running Input String Example

\[(q_1, 0011, \varepsilon) \vdash (q_2, 0011, \$$)\]
\[(q_2, 011, 0\$$) \vdash (q_2, 11, 00\$$)\]
PDA Running Input String Example

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

Input Read | Pop | Push
--- | --- | ---
$\varepsilon, \varepsilon \rightarrow \$ | 0, $\varepsilon \rightarrow 0$

1, $\varepsilon \rightarrow \varepsilon$

$\varepsilon, \$, $\rightarrow \varepsilon$

State | Remaining Input | Stack
--- | --- | ---

Read 1, pop 0
PDA Running Input String Example

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

Read 1, pop 0
PDA Running Input String Example

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

**Input Read** | **Pop** | **Push**
--- | --- | ---
\[\varepsilon, \varepsilon \rightarrow \$\] | \[0, \varepsilon \rightarrow 0\] | \[1, 0 \rightarrow \varepsilon\]
\[\varepsilon, \$ \rightarrow \varepsilon\]

**Stack**

pop empty stack symbol
Flashback: Computation and Languages

- The **language** of a machine is the set of all strings that it accepts.

- E.g., A DFA $M$ accepts $w$ if $\delta(q_0, w) \in F$.

- Language of $M = L(M) = \{ w \mid M \text{ accepts } w \}$.
Language of a PDA

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

\[ L(P) = \{ w \mid (q_0, w, \varepsilon) \vdash^* (q, \varepsilon, \alpha) \} \text{ where } q \in F \]

A configuration \((q, w, \gamma)\) has three components:
- \(q\) = the current state
- \(w\) = the remaining input string
- \(\gamma\) = the stack contents
PDAs and CFLs?

• **PDA** = NFA + a stack
  • Infinite memory
  • Push/pop top location only

• **Want to prove**: PDAs represent CFLs!

• **We know**: a CFL, by definition, is a language that is generated by a CFG

• **Need to show**: PDA ⇔ CFG

• Then, **to prove that a language is a CFL**, we can either:
  • Create a CFG, or
  • Create a PDA
A lang is a CFL iff some PDA recognizes it

⇒ If a language is a **CFL**, then a PDA recognizes it
  • We know: A CFL has a CFG describing it (definition of CFL)
  • To prove this part: show the CFG has an equivalent PDA

⇐ If a PDA recognizes a language, then it’s a CFL
Shorthand: Multi-Symbol Read Transition
Shorthand: Multi-Stack Push Transition

Note the reverse order of pushes
CFG→PDA (sketch)

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

- Diagram:

  ![Diagram of PDA transitions]

  - $q_{start}$
  - $q_{loop}$
  - $q_{accept}$
  - Transitions:
    - $\epsilon, \epsilon \rightarrow S$ for $S\rightarrow \epsilon$
    - $\epsilon, A \rightarrow w$ for rule $A \rightarrow w$
    - $a, a \rightarrow \epsilon$ for terminal $a$
    - $\epsilon, \epsilon \rightarrow \epsilon$
**CFG→PDA (sketch)**

- Construct PDA from CFG such that:
  - PDA accepts input only if CFG generates it

- PDA:
  - simulates generating a string with CFG rules
  - by (nondeterministically) trying all rules to find the right ones

\[
\begin{align*}
q_{\text{start}} & \xrightarrow{\varepsilon, \varepsilon \rightarrow S\$} q_{\text{loop}} \\
q_{\text{loop}} & \xrightarrow{\varepsilon, \$ \rightarrow \varepsilon} q_{\text{accept}}
\end{align*}
\]

- **Push start variable onto stack**
- **If: stack top is variable** \( A \), **pop and** ...
- **... push rule’s right-sides** (nondeterministically)
- **\( \varepsilon, A \rightarrow w \) for rule** \( A \rightarrow w \)
- **\( a, a \rightarrow \varepsilon \) for terminal** \( a \)
- **If: stack top is terminal** \( a \), **pop and** ...
- **... read matching input**
Example \textbf{CFG→PDA}

- \( q_{\text{start}} \)
  - \( \epsilon, \epsilon \rightarrow S \)
  - \( \epsilon, \epsilon \rightarrow \epsilon \)
  - \( \epsilon, S \rightarrow \epsilon \)
  - \( \epsilon, T \rightarrow \epsilon \)
  - \( a, a \rightarrow \epsilon \)
  - \( b, b \rightarrow \epsilon \)

- \( q_{\text{loop}} \)
  - \( \epsilon, \epsilon \rightarrow T \)
  - \( \epsilon, \epsilon \rightarrow a \)
  - \( \epsilon, \epsilon \rightarrow T \)
  - \( \epsilon, S \rightarrow b \)
  - \( T \rightarrow T \alpha | \epsilon \)
  - \( S \rightarrow aTb | b \)

- \( q_{\text{accept}} \)

**push** start variable onto stack

If: stack top is variable \( S \), pop \( S \) and ...

... push rule right-sides (in rev order)
Example **CFG→PDA**

\[ S \rightarrow aTb \mid b \]
\[ T \rightarrow Ta \mid \epsilon \]

![Diagram of a PDA](image)

- **States**: \( q_{start} \), \( q_{loop} \), \( q_{accept} \)
- **Transitions**:
  - \( \epsilon, \epsilon \rightarrow \$ \)
  - \( \epsilon, \epsilon \rightarrow S \)
  - \( \epsilon, \$ \rightarrow \epsilon \)
  - \( \epsilon, S \rightarrow b \)
  - \( \epsilon, T \rightarrow a \)
  - \( \epsilon, T \rightarrow \epsilon \)
  - \( a, a \rightarrow \epsilon \)
  - \( b, b \rightarrow \epsilon \)
  - \( \epsilon, \epsilon \rightarrow T \)
  - \( \epsilon, \epsilon \rightarrow a \)
Example **CFG→PDA**

\[
S \rightarrow aTb \mid b \\
T \rightarrow Ta \mid \varepsilon
\]

If: stack top is terminal, pop and read matching input
Example CFG→PDA

### Example Derivation using CFG:
- $S \rightarrow aTb$ (using rule $S \rightarrow aTb$)
- $\Rightarrow aTab$ (using rule $T \rightarrow Ta$)
- $\Rightarrow aab$ (using rule $T \rightarrow \varepsilon$)

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

### PDA Example

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
<th>Stack</th>
<th>Equiv Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_{start}$</td>
<td>$aab$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$q_{loop}$</td>
<td>$aab$</td>
<td>$S$</td>
<td>$S \rightarrow aTb$</td>
</tr>
<tr>
<td>$q_{loop}$</td>
<td>$aab$</td>
<td>$aTb$</td>
<td></td>
</tr>
<tr>
<td>$q_{loop}$</td>
<td>$ab$</td>
<td>$ab$</td>
<td>$T \rightarrow Ta$</td>
</tr>
<tr>
<td>$q_{loop}$</td>
<td>$ab$</td>
<td>$\varepsilon$</td>
<td></td>
</tr>
<tr>
<td>$q_{loop}$</td>
<td>$b$</td>
<td>$b$</td>
<td>$T \rightarrow \varepsilon$</td>
</tr>
<tr>
<td>$q_{loop}$</td>
<td>$$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example **CFG→PDA**

Example Derivation using CFG:

- $S \Rightarrow aTb$ (using rule $S \rightarrow aTb$)
- $\Rightarrow aTab$ (using rule $T \rightarrow Ta$)
- $\Rightarrow aab$ (using rule $T \rightarrow \epsilon$)

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

PDA Example

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
<th>Stack</th>
<th>Equiv Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_{start}$</td>
<td>aab</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$q_{loop}$</td>
<td>aab</td>
<td>$S$</td>
<td>$S \rightarrow aTb$</td>
</tr>
<tr>
<td>$q_{loop}$</td>
<td>aab</td>
<td>aTb$</td>
<td></td>
</tr>
<tr>
<td>$q_{loop}$</td>
<td>ab</td>
<td>Tb$</td>
<td></td>
</tr>
<tr>
<td>$q_{loop}$</td>
<td>ab</td>
<td>Tab$</td>
<td>$T \rightarrow Ta$</td>
</tr>
<tr>
<td>$q_{loop}$</td>
<td>ab</td>
<td>ab$</td>
<td>$T \rightarrow \epsilon$</td>
</tr>
<tr>
<td>$q_{loop}$</td>
<td>b</td>
<td>b$</td>
<td></td>
</tr>
<tr>
<td>$q_{accept}$</td>
<td></td>
<td>$\ $</td>
<td></td>
</tr>
</tbody>
</table>
Example CFG → PDA

Example Derivation using CFG:

- $S \rightarrow aTb$ (using rule $S \rightarrow aTb$)
- $\Rightarrow aTab$ (using rule $T \rightarrow Ta$)
- $\Rightarrow aab$ (using rule $T \rightarrow \varepsilon$)

PDA Example

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
<th>Stack</th>
<th>Equiv Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_{start}$</td>
<td>aab</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$q_{loop}$</td>
<td>aab</td>
<td>$S$$</td>
<td>$S \rightarrow aTb$</td>
</tr>
<tr>
<td>$q_{loop}$</td>
<td>aab</td>
<td>$aTb$$</td>
<td></td>
</tr>
<tr>
<td>$q_{loop}$</td>
<td>ab</td>
<td>$7b$$</td>
<td></td>
</tr>
<tr>
<td>$q_{loop}$</td>
<td>ab</td>
<td>Tab$$</td>
<td>$T \rightarrow Ta$</td>
</tr>
<tr>
<td>$q_{loop}$</td>
<td>ab</td>
<td>ab$$</td>
<td>$T \rightarrow \varepsilon$</td>
</tr>
<tr>
<td>$q_{loop}$</td>
<td>b</td>
<td>b$$</td>
<td></td>
</tr>
<tr>
<td>$q_{accept}$</td>
<td></td>
<td>$$$</td>
<td></td>
</tr>
</tbody>
</table>

If stack top is terminal, pop and read matching input.
Example \textbf{CFG\textarrow{\rightarrow}PDA}

\textbf{Example Derivation using CFG:}
\begin{align*}
S & \Rightarrow aTb \quad \text{(using rule } S \rightarrow aTb) \\
& \Rightarrow aTab \quad \text{(using rule } T \rightarrow Ta) \\
& \Rightarrow aab \quad \text{(using rule } T \rightarrow \varepsilon)
\end{align*}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
\textbf{State} & \textbf{Input} & \textbf{Stack} & \textbf{Equiv Rule} \\
\hline
q_{start} & aab & & \\
\hline
q_{loop} & aab & \text{S$} & S \rightarrow aTb \\
\hline
q_{loop} & aab & aTb$ & T \rightarrow Ta \\
\hline
q_{loop} & ab & Tb$ & T \rightarrow \varepsilon \\
\hline
q_{accept} & b & b$ & \\
\hline
\end{tabular}
\end{table}
A lang is a CFL iff some PDA recognizes it

⇒ If a language is a CFL, then a PDA recognizes it
  • Convert CFG→PDA

⇐ If a PDA recognizes a language, then it’s a CFL
  • To prove this part: show PDA has an equivalent CFG
PDA→CFG: Prelims

Before converting PDA to CFG, modify it so:

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

\textbf{Important:}
This doesn’t change the language recognized by the PDA.
PDA $P \rightarrow$ CFG $G$: Variables

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

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

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

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

\textbf{To add terminals:} pair up stack pushes and pops (essence of a CFL)
PDA $P \rightarrow$ CFG $G$: Generating Strings

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

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

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

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

put the rule $A_{pq} \rightarrow aA_{rs}b$ in $G$
PDA $P \rightarrow$ CFG $G$: Generating Strings

$P = (Q, \Sigma, \Gamma, \delta, q_0, \{q_{\text{accept}}\})$  \hspace{1cm} \text{variables of } G \text{ are } \{A_{pq} \mid p, q \in Q\}$

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

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

put the rule $A_{pq} \rightarrow aA_{rs}b$ in $G$
PDA $P \rightarrow$ CFG $G$: Generating Strings

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

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

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

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

  put the rule $A_{pq} \rightarrow aA_{rs}b$ in $G$
A language is a CFL $\iff$ A PDA recognizes it

- If a language is a CFL, then a PDA recognizes it
  - Convert CFG $\rightarrow$ PDA

- If a PDA recognizes a language, then it’s a CFL
  - Convert PDA $\rightarrow$ CFG
Submit in-class work 3/20

On Gradescope