UMB CS 622

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

$L(G_1) = \{0^n#1^n \mid n \geq 0\}$

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:

$\alpha, \beta \in (V \cup \Sigma)^*$

$A \in V$  \hspace{1cm} Variable

$A \rightarrow \gamma \in R$  \hspace{1cm} 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.
Derivations: Formally

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

**Single-step**

$$\alpha A \beta \xrightarrow{G} \alpha \gamma \beta$$

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:**

$$\alpha \xrightarrow{*} \alpha \quad (0 \text{ steps})$$

**Recursive case:**

$$\alpha \xrightarrow{*} \gamma$$

Where: $\alpha \xrightarrow{G} \beta$ and $\beta \xrightarrow{*} \gamma$

Single step

(smaller) Recursive “call”
Last Time:

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.

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

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

If a CFG generates all strings in a language $L$, then $L$ is a context-free language (CFL)

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

"... that can be generated with rules of grammar $G$"
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 $\rightarrow$ `<TAG>CONTENT</TAG>`
   • Start and end tags are “linked”

2. Start with small grammars and then combine
   • just like with FSMs, and programming!
Designing Grammars: Building Up

• Start with small grammars and then combine (just like FSMs)
  • To create a grammar for the language \( \{0^n1^n | n \geq 0\} \cup \{1^n0^n | n \geq 0\} \)

• First create grammar for lang \( \{0^n1^n | n \geq 0\} \):
  \[
  S_1 \rightarrow 0S_11 \mid \varepsilon
  \]

• Then create grammar for lang \( \{1^n0^n | n \geq 0\} \):
  \[
  S_2 \rightarrow 1S_20 \mid \varepsilon
  \]

• Then combine:
  \[
  S \rightarrow S_1 \mid S_2
  \]
  \[
  S_1 \rightarrow 0S_11 \mid \varepsilon
  \]
  \[
  S_2 \rightarrow 1S_20 \mid \varepsilon
  \]

New start variable and rule combines two smaller grammars
“\mid” = “or” = union (combines 2 rules with same left side)
(Closed) Operations for CFLs?

- Start with small grammars and then combine (just like FSMs)

  - “Or”: \( S \rightarrow S_1 | S_2 \)

  - “Concatenate”: \( S \rightarrow S_1 S_2 \)

  - “Repetition”: \( S' \rightarrow S'S_1 | \varepsilon \)

Could you write out the full proof of these closure operations?
Example: Creating CFG

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

\{\textit{w} | w \text{ starts and ends with the same symbol}\}

1) come up with \textbf{examples}:
   In the language: \textit{010, 101, 11011} \quad 1, 0 ? \quad \checkmark
   Not in the language: \textit{10, 01, 110} \quad \varepsilon \ ? \quad \times

2) \textbf{Create} CFG:

   \[ S \rightarrow 0M0 \mid 1M1 \mid 0 \mid 1 \]
   \[ M \rightarrow MT \mid \varepsilon \]
   \[ T \rightarrow 0 \mid 1 \]

   \textit{Needed Rules:}
   
   “start/end symbol are “linked” (ie, same); middle can be anything”
   
   “middle: all possible terminals, repeated (ie, \textit{all possible strings})”
   
   “\textit{all possible terminals}”

3) \textbf{Check} CFG: generates \textbf{examples} in the language; \textbf{does not generate} \textbf{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>

|                  |                                              |
|                  |                                              |
|                  |                                              |
|                  |                                              |
|                  |                                              |
|                  |                                              |
|                  |                                              |
|                  |                                              |
## 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>???</td>
</tr>
<tr>
<td>recognizes a Regular Lang</td>
<td>recognizes 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>
## 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>

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

PDA = NFA + 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

![Diagram of a PDA](image)
An Example PDA

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

\[ \{0^n1^n \mid n \geq 0\} \]

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 \(Q\), \(\Sigma\), \(\Gamma\), 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.

Stack alphabet has special stack symbols, e.g., 

Non-deterministic!

Result of a step is set of \((\text{State, Stack Char})\) pairs
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. \(\delta: Q \times \Sigma \times \Gamma \rightarrow P(Q \times \Gamma)\) 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: ( 0 )</th>
<th>( \varepsilon )</th>
<th>( 0 )</th>
<th>( $ )</th>
<th>( \varepsilon )</th>
<th>( $ )</th>
<th>( \varepsilon )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack: ( \emptyset )</td>
<td>( \varepsilon )</td>
<td>( \emptyset )</td>
<td>( \varepsilon )</td>
<td>( \emptyset )</td>
<td>( \varepsilon )</td>
<td>( \varepsilon )</td>
</tr>
<tr>
<td>( q_1 )</td>
<td>( { (q_2, 0) } )</td>
<td>( { (q_3, \varepsilon) } )</td>
<td>( { (q_3, \varepsilon) } )</td>
<td>( { (q_4, \varepsilon) } )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( q_2 )</td>
<td>( { (q_2, 0) } )</td>
<td>( { (q_3, \varepsilon) } )</td>
<td>( { (q_3, \varepsilon) } )</td>
<td>( { (q_4, \varepsilon) } )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( q_3 )</td>
<td>( { (q_2, 0) } )</td>
<td>( { (q_3, \varepsilon) } )</td>
<td>( { (q_3, \varepsilon) } )</td>
<td>( { (q_4, \varepsilon) } )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( q_4 )</td>
<td>( { (q_2, 0) } )</td>
<td>( { (q_3, \varepsilon) } )</td>
<td>( { (q_3, \varepsilon) } )</td>
<td>( { (q_4, \varepsilon) } )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A pushdown automaton is a 6-tuple \( (Q, \Sigma, \Gamma, \delta, q_0, F) \), where \( Q, \Sigma, \Gamma, \delta, \) 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. \( \delta : Q \times \Sigma \times \varepsilon \rightarrow \mathcal{P}(Q \times \varepsilon) \) 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>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>1</td>
<td>( (q_2, 0) )</td>
<td>( (q_3, \varepsilon) )</td>
</tr>
<tr>
<td>( q_2 )</td>
<td>( (q_3, \varepsilon) )</td>
<td>( (q_3, \varepsilon) )</td>
<td>( (q_2, $) )</td>
</tr>
<tr>
<td>( q_3 )</td>
<td>( (q_4, \varepsilon) )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( q_4 )</td>
<td>( (q_1, 0) )</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. \( \delta: Q \times \Sigma \times \Gamma \rightarrow P(Q \times \Gamma) \) is the transition function,
5. \( q_0 \in Q \) is the start state, and
6. \( F \subseteq Q \) is the set of accept states.
\[ Q = \{q_1, q_2, q_3, q_4\}, \]
\[ \Sigma = \{0,1\}, \]
\[ \Gamma = \{0,\$,\}, \]
\[ F = \{q_1, q_4\}, \text{ and} \]

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

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

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

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

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

- 1: \((q_3, \varepsilon)\)
- 2: \((q_3, \varepsilon)\)
- 3: \((q_3, \varepsilon)\)
- 4: \((q_4, \varepsilon)\)
- 5: \((q_2, \$)\)

A pushdown automaton is a 6-tuple \((Q, \Sigma, \Gamma, \delta, q_0, F)\), where \(Q, \Sigma, \Gamma, \delta, q_0, F\) 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>Stack:</th>
<th>Input:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$$ $</td>
</tr>
<tr>
<td>\cell{q_1}</td>
<td>$(q_2, 0)$</td>
</tr>
<tr>
<td>\cell{q_2}</td>
<td>$(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.
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>Stack:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 0 )</td>
<td>0 ( $ ) ( \varepsilon )</td>
</tr>
<tr>
<td>1</td>
<td>0 ( $ ) ( \varepsilon )</td>
</tr>
<tr>
<td>( \varepsilon )</td>
<td>0 ( $ ) ( \varepsilon )</td>
</tr>
</tbody>
</table>

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

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. \( \delta: Q \times \Sigma_e \times \Gamma_e \rightarrow P(Q \times \Gamma_e) \) 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

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

\( \delta \) 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 $ \varepsilon$</td>
<td>0 1 $ \varepsilon$</td>
<td>0 $ \varepsilon$</td>
</tr>
</tbody>
</table>

- \( q_1 \)
  - \( \delta(q_1, 0) = \{(q_2, \varepsilon)\} \)
  - \( \delta(q_1, \varepsilon) = \{(q_3, \varepsilon)\} \)

- \( q_2 \)
  - \( \delta(q_2, 0) = \{(q_2, 0)\} \)
  - \( \delta(q_2, 1) = \{(q_2, 1)\} \)
  - \( \delta(q_2, \varepsilon) = \{(q_2, \$)\} \)

- \( q_3 \)
  - \( \delta(q_3, \varepsilon) = \{(q_3, \varepsilon)\} \)

- \( q_4 \)
  - \( \delta(q_4, \varepsilon) = \{(q_4, \varepsilon)\} \)

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

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

- **Domain** (inputs):
  - state \( q \in Q \)
  - string \( w = w_1w_2 \cdots w_n \) 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 \( w' = w_1 \cdots w_{n-1} \)

\( \delta : Q \times \Sigma \to Q \) is the **transition function**

A DFA computation is a sequence of states:
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** \((q, w, γ)\):
  - \(q\) = the current state
  - \(w\) = the remaining input string
  - \(γ\) = 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 \((q_2, \alpha)\)

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

**Multi-step**

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

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

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

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

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

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

Diagram:
- **Input Read**
- **Pop**
- **Push**

States:
- \(q_1\)
- \(q_2\)
- \(q_3\)
- \(q_4\)

Transitions:
- \(\varepsilon, \varepsilon \rightarrow \$\) from \(q_1\) to \(q_2\)
- \(0, \varepsilon \rightarrow 0\) from \(q_2\) to \(q_2\)
- \(1, 0 \rightarrow \varepsilon\) from \(q_2\) to \(q_3\)
- \(\varepsilon, \$, \varepsilon \rightarrow \epsilon\) from \(q_4\) to \(q_3\)

Stack:
- \(\$\)

Read 0, push 0
PDA Running Input String Example

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

Read 1, pop 0
PDA Running Input String Example

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

Read 1, pop 0
PDA Running Input String Example

\[(q_1, 0011, \varepsilon) \vdash (q_2, 0011, \$)\]
\[(q_2, 011, 0\$)\]
\[(q_2, 11, 00\$)\]
\[(q_3, 1, 0\$)\]
\[(q_3, \varepsilon, \$)\]
\[(q_4, \varepsilon, \varepsilon)\]
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 \( M \) = \( L(M) \) = \{ \( w \) | \( M \) 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 $\Leftrightarrow$ 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
Submit in-class work 3/20

On gradescope