UMB CS 420

Pushdown Automata (PDAs)

Wednesday, March 8, 2023
Announcements

• HW 5 out
  • Due Sun 3/19 11:59pm EST
  • (After Spring Break)

• No lecture next week
  • (Spring Break)

Quiz Preview
1. Which of the following are possible representations of a CFL?
2. Which of the following are characteristics of a PDA?
   • Infinite or finite “memory”?
   • Infinite or finite states?
   • Deterministic or nondeterministic?
Last Time: Generating Strings with a CFG

A CFG represents a context free language!

Strings in CFG’s language = all possible generated strings

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

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

\[
A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000B111 \Rightarrow 000\#111
\]
Last Time:

<table>
<thead>
<tr>
<th>Regular Languages</th>
<th>Context-Free Languages (CFLs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular Expression</td>
<td><strong>Context-Free Grammar (CFG)</strong></td>
</tr>
<tr>
<td>A Reg Expr describes a Regular lang</td>
<td>A CFG describes a CFL</td>
</tr>
</tbody>
</table>
**Today:**

<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>A Reg Expr describes a Regular lang</td>
<td>A CFG describes a CFL</td>
</tr>
<tr>
<td><strong>TODAY:</strong></td>
<td></td>
</tr>
<tr>
<td>Finite Automaton (FSM)</td>
<td><strong>Push-down automaton</strong> (PDA)</td>
</tr>
<tr>
<td>An FSM recognizes a Regular lang</td>
<td>A PDA recognizes a CFL</td>
</tr>
</tbody>
</table>
### Today:

<table>
<thead>
<tr>
<th><strong>Regular</strong> Languages</th>
<th><strong>Context-Free</strong> Languages (CFLs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular Expression</td>
<td>Context-Free Grammar (CFG)</td>
</tr>
<tr>
<td>A Reg Expr <strong>describes</strong> a Regular lang</td>
<td>A CFG <strong>describes</strong> a CFL</td>
</tr>
</tbody>
</table>

**TODAY:**

<table>
<thead>
<tr>
<th><strong>Finite Automaton</strong> (FSM)</th>
<th><strong>Push-down automaton</strong> (PDA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>An FSM <strong>recognizes</strong> a Regular lang</td>
<td>A PDA <strong>recognizes</strong> a CFL</td>
</tr>
</tbody>
</table>

**KEY DIFFERENCE:**

<table>
<thead>
<tr>
<th>A Regular lang is <strong>defined</strong> with a FSM</th>
<th>A CFL is <strong>defined</strong> with a CFG</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Must prove: Reg Expr ↔ Reg lang</em></td>
<td><em>Must prove: PDA ↔ CFL</em></td>
</tr>
</tbody>
</table>
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
  - Can only read/write top location
    - Push/pop

```
NFA-like states

input

aabbb

stack

x
y
z
```
An Example PDA

$ = \text{special symbol, indicating empty stack}$

Can only pop this \text{and accept} when stack is empty, i.e., when \# 0s matches \# 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 can have special stack symbols, e.g., $.

Non-deterministic: produces a set of (State, Stack Char) pairs.

Input, Pop, Push
PDA Formal Definition Example

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

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>( { (q_3, \varepsilon) } )</td>
<td>( { (q_4, \varepsilon) } )</td>
</tr>
<tr>
<td>( q_2 )</td>
<td>( { (q_2, 0) } )</td>
<td>( { (q_3, \varepsilon) } )</td>
<td>2</td>
<td>( { (q_2, \varepsilon) } )</td>
</tr>
<tr>
<td>( q_3 )</td>
<td>( { (q_3, \varepsilon) } )</td>
<td>3</td>
<td>( { (q_4, \varepsilon) } )</td>
<td></td>
</tr>
<tr>
<td>( q_4 )</td>
<td>4</td>
<td>( { (q_2, $) } )</td>
<td>5</td>
<td>( { (q_2, $) } )</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>$</td>
<td>( \varepsilon )</td>
<td>0</td>
</tr>
</tbody>
</table>

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

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

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

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

<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>{ ((q_4, \varepsilon))}</td>
<td>{ ((q_2, $))}</td>
</tr>
<tr>
<td>( q_2 )</td>
<td>{ ((q_3, \varepsilon))}</td>
<td>{ ((q_3, \varepsilon))}</td>
<td>{ ((q_4, \varepsilon))}</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.
\[
Q = \{q_1, q_2, q_3, q_4\},
\]
\[
\Sigma = \{0, 1\},
\]
\[
\Gamma = \{0, \$\},
\]
\[
F = \{q_1, q_4\},
\]

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

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

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.
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>( \epsilon )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack:</td>
<td>0</td>
<td>$</td>
<td>( \epsilon )</td>
</tr>
</tbody>
</table>

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

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.
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>$</td>
<td>\varepsilon</td>
<td>0</td>
</tr>
</tbody>
</table>

\begin{align*}
q_1 & \rightarrow ((q_2, 0)) \\
q_2 & \rightarrow \{(q_2, 1)\} \\
q_3 & \rightarrow \{(q_3, \varepsilon)\} \\
q_4 & \rightarrow \{(q_4, \varepsilon)\}
\end{align*}

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

**Informally**
- "Program" = a finite automata
- Input = string of chars, e.g. "1101"

To run a “program”:
- **Start** in “start state”

Repeat:
- Read 1 char;
- Change state according to the transition table

Result =
- “Accept” if last state is “Accept” state
- “Reject” otherwise

**Formally (i.e., mathematically)**
- $M = (Q, \Sigma, \delta, q_0, F)$
- $w = w_1 w_2 \cdots w_n$

- $r_0 = q_0$

- $r_i = \delta(r_{i-1}, w_i)$, for $i = 1, \ldots, n$

- $M$ accepts $w$ if there exists a sequence of states $r_0, r_1, \ldots, r_n$ in $Q$ with $r_n \in F$.

A sequence of states represents a DFA computation.
Flashback: A DFA Extended Transition Fn

Define **extended transition function:**

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

- **Domain:**
  - Beginning state \( q \in Q \) (not necessarily the start state)
  - Input string \( w = w_1w_2 \cdots w_n \) where \( w_i \in \Sigma \)
- **Range:**
  - Ending state (not necessarily an accept state)

(Defined recursively)

- **Base case:** \( \hat{\delta}(q, \varepsilon) = q \)
- **Recursive case:** \( \hat{\delta}(q, w) = \hat{\delta}(\hat{\delta}(q, w_1), w_2 \cdots w_n) \)

This specifies the **sequence of states** for a DFA computation.
Last Time: 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**
- **Push**

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

- \(q_1, q_2 \in Q\)
- \(a \in \Sigma\)
- \(w \in \Sigma^*\)
- \(X \in \Gamma\)
- \(\beta, \alpha \in \Gamma^*\)

### Extended

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

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

---

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

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

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

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

State  | Remaining Input  | Stack
--- | --- | ---
\[\text{State} \quad \text{Remaining Input} \quad \text{Stack}\]
\[\text{Read 1, pop 0}\]

81
PDA Running Input String Example

\[(q_1, 0011, \varepsilon) \vdash (q_2, 0011, \$)\]

\[(q_2, 011, 0\$)\]

\[(q_2, 11, 00\$)\]

\[(q_3, 1, 0\$)\]

\[(q_3, \varepsilon, \$)\]

Read 1, pop 0
PDA Running Input String Example

\[(q_1, 0011, \varepsilon) \vdash (q_2, 0011, \$)\]
\[\vdash (q_2, 011, 0\$)\]
\[\vdash (q_2, 11, 00\$)\]
\[\vdash (q_3, 1, 0\$)\]
\[\vdash (q_3, \varepsilon, \$)\]
\[\vdash (q_4, \varepsilon, \varepsilon)\]
**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 $M = L(M) = \{ w | M \text{ accepts } w \}$.
Language of a PDA

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

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

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
  • Can only read/write top location: Push/pop

• **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
Shorthand: Multi-Symbol Read Transition

\[ x, \epsilon \rightarrow \epsilon \]
\[ y, \epsilon \rightarrow \epsilon \]
\[ z, \epsilon \rightarrow \epsilon \]

Read 1
Read multi char

89
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

```
  q_{start} \rightarrow \epsilon, \epsilon \rightarrow S$

  q_{loop} \rightarrow \epsilon, A \rightarrow w \text{ for rule } A \rightarrow w
  \quad \epsilon, a \rightarrow \epsilon \text{ for terminal } a

  q_{accept} \rightarrow \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

```
q_{start}  ε, ε→S$
q_{loop}   ε, A→w  for rule A→w
           a, a→ε  for terminal a
q_{accept} ε, $→ε
```

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

- \textbf{push} start variable onto stack
- If: stack top is variable $S$, pop $S$ and ...
- ... push rule right-sides (in rev order)
Example \textbf{CFG→PDA}

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

Diagram:

- Start state: $q_{\text{start}}$
- transitions:
  - $\varepsilon, \varepsilon \rightarrow S$
  - $\varepsilon, \varepsilon \rightarrow \varepsilon$
  - $\varepsilon, \varepsilon \rightarrow S$
  - $\varepsilon, S \rightarrow b$
  - $\varepsilon, T \rightarrow a$
  - $\varepsilon, T \rightarrow \varepsilon$
  - $a, a \rightarrow \varepsilon$
  - $b, b \rightarrow \varepsilon$

- Accept state: $q_{\text{accept}}$

- Accept transitions:
  - $\varepsilon, \varepsilon \rightarrow a$
  - $\varepsilon, \varepsilon \rightarrow T$
  - $\varepsilon, \varepsilon \rightarrow \varepsilon$

Note: Diagram shows transitions and states corresponding to the CFG rules.
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 $T \rightarrow \epsilon$)

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>$S$$</td>
<td>$S \rightarrow aTb$</td>
</tr>
<tr>
<td>$q_{loop}$</td>
<td>aab</td>
<td>aTb$</td>
<td>$S \rightarrow aTb$</td>
</tr>
<tr>
<td>$q_{loop}$</td>
<td>ab</td>
<td>Tb$</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>$T \rightarrow \epsilon$</td>
</tr>
<tr>
<td>$q_{accept}$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example Derivation using CFG:

\[ S \rightarrow a T b \] (using rule \( S \rightarrow a T b \))
\[ \Rightarrow a T a b \] (using rule \( T \rightarrow T a \))
\[ \Rightarrow a a b \] (using rule \( T \rightarrow \varepsilon \))

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

**Example CFG→PDA**

**PDA Example**

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
<th>Stack</th>
<th>Equiv Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q_{\text{start}} )</td>
<td>aab</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( q_{\text{loop}} )</td>
<td>aab</td>
<td>$S$</td>
<td></td>
</tr>
<tr>
<td>( q_{\text{loop}} )</td>
<td>aab</td>
<td>aTb$</td>
<td>( S \rightarrow aTb )</td>
</tr>
<tr>
<td>( q_{\text{loop}} )</td>
<td>ab</td>
<td>Tb$</td>
<td></td>
</tr>
<tr>
<td>( q_{\text{loop}} )</td>
<td>ab</td>
<td>Tab$</td>
<td>( T \rightarrow Ta )</td>
</tr>
<tr>
<td>( q_{\text{loop}} )</td>
<td>ab</td>
<td>ab$</td>
<td>( T \rightarrow \varepsilon )</td>
</tr>
<tr>
<td>( q_{\text{loop}} )</td>
<td>b</td>
<td>b$</td>
<td></td>
</tr>
<tr>
<td>( q_{\text{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$</td>
<td>( S \rightarrow aTb )</td>
</tr>
<tr>
<td>( q_{loop} )</td>
<td>aab</td>
<td>a?b$</td>
<td>( S \rightarrow aTb )</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$</td>
<td>( T \rightarrow \varepsilon )</td>
</tr>
<tr>
<td>( q_{loop} )</td>
<td>b</td>
<td>b$</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 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_{\text{start}} )</td>
<td>aab</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( q_{\text{loop}} )</td>
<td>aab</td>
<td>$</td>
<td>( S \rightarrow aTb )</td>
</tr>
<tr>
<td>( q_{\text{loop}} )</td>
<td>aab</td>
<td>aTb$</td>
<td>( S \rightarrow aTb )</td>
</tr>
<tr>
<td>( q_{\text{loop}} )</td>
<td>ab</td>
<td>Tb$</td>
<td></td>
</tr>
<tr>
<td>( q_{\text{loop}} )</td>
<td>ab</td>
<td>Tab$</td>
<td>( T \rightarrow Ta )</td>
</tr>
<tr>
<td>( q_{\text{loop}} )</td>
<td>ab</td>
<td>ab$</td>
<td>( T \rightarrow \varepsilon )</td>
</tr>
<tr>
<td>( q_{\text{loop}} )</td>
<td>b</td>
<td>b$</td>
<td></td>
</tr>
<tr>
<td>( q_{\text{loop}} )</td>
<td></td>
<td>$</td>
<td></td>
</tr>
<tr>
<td>( q_{\text{accept}} )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</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, $q_{\text{accept}}$.
2. It empties its stack before accepting.
3. Each transition either pushes a symbol onto the stack (a *push* move) or pops one off the stack (a *pop* move), but it does not do both at the same time.

**Important:**
This doesn’t change the language recognized by the PDA
PDA $P \rightarrow$ CFG $G$: Variables

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

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

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

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

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

• **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 $\{A_{pq} | p, q \in Q\}$

- **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 $\delta(s, b, \varepsilon)$ 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 \( \{A_{pq} \mid p, q \in Q\} \)

- 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 $G$
A language is a CFL $\iff$ A PDA recognizes it

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

- $\Leftarrow$ If a PDA recognizes a language, then it’s a CFL
  - Convert PDA $\rightarrow$ CFG
Regular Languages are CFLs: 3 Proofs

• DFA \rightarrow CFG
  • HW?

• NFA \rightarrow CFG
  • NFA \rightarrow PDA (with no stack moves) \rightarrow CFG
  • Just now

• Regular expression \rightarrow CFG
  • HW?
Check-in Quiz 3/8

On Gradescope