Pushdown Automata (PDAs)

- A new type of computational model.
- It is like a NFA but has an extra component called *stack*.
- The stack provides additional memory beyond the finite amount available in the control.
- The stack allows pushdown automata to recognize some non-regular languages.
- Pushdown automata are equivalent in power to context-free grammars.

- A PDA can write symbols on stack and read them back later
- Writing a symbol is **pushing**, removing a symbol is **popping**
- Access to the stack, for both reading and writing, may be done only at the **top** (last in, first out)
- A stack is valuable because it can hold an unlimited amount of information.
Example

• Consider the language \( \{0^n1^n : n \geq 0\} \).

• Finite automaton is unable to recognize this language.

• A PDA is able to do this.

Informal description how the PDA for this language works.

• Read symbols from the input.
• As each 0 is read, push it into the stack.
• As soon as 1s are seen, pop a 0 off the stack for each 1 read.
• If reading the input is finished exactly when the stack becomes empty of 0s, accept the input.
• If the stack becomes empty while 1s remain or
  if the 1s are finished while the stack still contains 0s or
  if any 0s appear in the input following 1s,
  reject the input.
**Formal Definition of PDAs**

- A *pushdown automaton* (PDA) is specified by a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where
  - $Q$ is a finite set of states,
  - $\Sigma$ is a finite input alphabet,
  - $\Gamma$ is a finite stack alphabet,
  - $\delta : Q \times \Sigma \times \Gamma \rightarrow P(Q \times \Gamma)$ is the transition function,
  - $q_0 \in Q$ is the initial state,
  - $F \subseteq Q$ is the set of final states.

- It computes as follows: it accepts input $w$ if $w$ can be written as $w = w_1, w_2, \ldots, w_n$, where each $w_i \in \Sigma$ and a sequence of states $r_0, r_1, r_2, \ldots, r_n \in Q$ and strings $s_0, s_1, s_2, \ldots, s_n \in \Gamma^*$ exist that satisfy the next three conditions (the strings $s_i$ represent the sequence of stack contents that PDA has on the accepting branch of the computation).
  1. $r_0 = q_0, s_0 = \varepsilon$
  2. $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$, $i = 0, \ldots, n-1$, where $s_i = at$, $s_{i+1} = bt$ for some $a, b \in \Gamma$ and $t \in \Gamma^*$.
  3. $r_n \in F$

Non-deterministic

Is a collection of all subsets

$\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$

$\Gamma_\varepsilon = \Gamma \cup \{\varepsilon\}$

---

Theory of Computation, Feodor F. Dragan, Kent State University
### Example

- Consider the language \( \{0^n1^n : n \geq 0\} \).

\[ M = (Q, \Sigma, \Gamma \delta, q_1, F) \]

- \( Q = \{q_1, q_2, q_3, q_4\} \)
- \( \Sigma = \{0, 1\} \)
- \( \Gamma = \{0, \$\} \)
- \( F = \{q_1, q_4\} \)

<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>( \emptyset )</td>
<td>( (q_2, 0) )</td>
<td>( (q_3, \varepsilon) )</td>
</tr>
<tr>
<td>( q_2 )</td>
<td>( \emptyset )</td>
<td>( (q_3, \varepsilon) )</td>
<td>( (q_4, \varepsilon) )</td>
</tr>
<tr>
<td>( q_3 )</td>
<td>( \emptyset )</td>
<td>( (q_3, \varepsilon) )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( q_4 )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
</tbody>
</table>

\( a, b \rightarrow c \) : when the machine is reading an \( a \) from the input it may replace the symbol \( b \) on the top of stack with a \( c \). Any of \( a, b, \) and \( c \) may be \( \varepsilon \).

- \( a \) is \( \varepsilon \), the machine may take this transition without reading any input symbol.
- \( b \) is \( \varepsilon \), the machine may take this transition without reading and popping any stack symbol.
- \( c \) is \( \varepsilon \), the machine does not write any symbol on the stack when going along this transition.

\[ 0, \varepsilon \rightarrow 0 \]
\[ 1, 0 \rightarrow \varepsilon \]
\[ \varepsilon, \varepsilon \rightarrow \$ \]
\[ 1, 0 \rightarrow \varepsilon \]
\[ \varepsilon, \$ \rightarrow \varepsilon \]
More Examples

- Language \( \{a^i b^j c^k : i, j, k \geq 0 \text{ and } i = j \text{ or } i = k \} \).

- Language \( \{ww^R : w \in \{0,1\}^*\} \).

\[
\begin{align*}
\text{Language: } \{a^i b^j c^k : i, j, k \geq 0 \text{ and } i = j \text{ or } i = k \},\\
\epsilon, \epsilon \rightarrow a,\\
b, a \rightarrow \epsilon,\\
c, \epsilon \rightarrow \epsilon,\\
\epsilon, \epsilon \rightarrow \epsilon,\\
b, \epsilon \rightarrow \epsilon,\\
c, a \rightarrow \epsilon,\\
\epsilon, \epsilon \rightarrow \epsilon,\\
\epsilon, \epsilon \rightarrow \epsilon,\\
\epsilon, \epsilon \rightarrow \epsilon,\\
\epsilon, \epsilon \rightarrow \epsilon,\\
\epsilon, \epsilon \rightarrow \epsilon,\\
\epsilon, \epsilon \rightarrow \epsilon,\\
\epsilon, \epsilon \rightarrow \epsilon.\\
\end{align*}
\]
Equivalence with Context-free Grammars

- Context-free grammars and pushdown automata are equivalent in their descriptive power. Both describe the class of context-free languages.
- Any context-free grammar can be converted into a pushdown automaton that recognizes the same language, and vice versa.
- We will prove the following result

**Theorem.** A language is context-free if and only if some pushdown automaton recognizes it.

- This theorem has two directions. We state each direction as a separate lemma.

**Lemma 1.** If a language is context-free, then some pushdown automaton recognizes it.

- We have a context-free grammar $G$ describing the context-free language $L$.
- We show how to convert $G$ into an equivalent PDA $P$.
- The PDA $P$ will accept string $w$ iff $G$ generates $w$, i.e., if there is a leftmost derivation for $w$.
- Recall that a derivation is simply the sequence of substitution made as a grammar generates a string.

$$S \xrightarrow{lm} AS \xrightarrow{lm} AIS \xrightarrow{lm} 011S \xrightarrow{lm} 011AS \xrightarrow{lm} 0110AIS \xrightarrow{lm} 0110011S \xrightarrow{lm} 0110011$$

$$S \rightarrow AS \mid \epsilon$$

$$A \rightarrow 0A1 \mid A1 \mid 01$$
How do we check that G generates 0110011?

\[ S \rightarrow AS \mid \varepsilon \]
\[ A \rightarrow 0A1 \mid A1 \mid 01 \]

Promising variants

Idea

- We can use stack to store an intermediate string of variables and terminals.
- It is better to keep only part (suffix) of the intermediate string, the symbols starting with the first variable.
- Any terminal symbols appearing before the first variable are matched immediately with symbols in the input string.
- Use non-determinism, make copies.
Informal description of PDA $P$

1. Place the marker symbol $\$ \text{ and the start symbol on the stack.}$

2. Repeat the following steps forever.
   a) If top of stack is a variable symbol $A$, non-deterministically select one of the rules for $A$ and substitute $A$ by the string on the right-hand side of the rule.
   b) If the top of stack is terminal symbol $a$, read the next symbol from the input and compare it to $a$. If they match, pop $a$ and repeat. If they do not match, reject on this branch of the non-determinism.
   c) If the top of the stack is the symbol $\$$, enter the accept state. Doing so accepts the input if it has all been read.
Construction of PDA $P$

- $q_{\text{start}}$
  - $\varepsilon, \varepsilon \rightarrow S\$
- $q_{\text{loop}}$
  - $\varepsilon, A \rightarrow w$ for each rule $A \rightarrow w$
  - $a, a \rightarrow \varepsilon$ for each terminal $a$
- $q_{\text{accept}}$
  - $\varepsilon, S \rightarrow \varepsilon$

For $|w| > 1$ use extensions

- For $a, s \rightarrow xyz$
  - $a, s \rightarrow z$
  - $x \rightarrow \varepsilon$
  - $y \rightarrow \varepsilon$

- $\varepsilon, \varepsilon \rightarrow x$

String of terminals and variables
Example

\[ S \rightarrow aTb \mid b \]
\[ T \rightarrow Ta \mid \varepsilon \]
Equivalence with Context-free Grammars

• We are working on the proof of the following result

**Theorem.** A language is context-free if and only if some pushdown automaton recognizes it.

• We have proved

**Lemma 1.** If a language is context-free, then some pushdown automaton recognizes it.

  • We have shown how to convert a given CFG $G$ into an equivalent PDA $P$.
  • Now we will consider the other direction

**Lemma 2.** If a pushdown automaton recognizes some language, then it is context-free.

  • We have a PDA $P$, and want to create a CFG $G$ that generates all strings that $P$ accepts.
  • That is $G$ should generate a string if that string causes the PDA to go from its start state to an accept state (*takes* $P$ from start state to an accept state).
Let $P$ be an arbitrary PDA.

For each pair of states $p$ and $q$ in $P$ the grammar will contain a variable $A_{pq}$.

This variable will generate all strings that can take $P$ from state $p$ with empty stack to $q$ with an empty stack.

Clearly, all those strings can also take $P$ from $p$ to $q$, regardless of the stack contents at $p$, leaving the stack at $q$ in the same condition as it was at $p$.

Example:

- String 000111 takes $P$ from start state to a finite state.
- String 00011 does not.

Design a Grammar:

- Let $P$ be an arbitrary PDA.
- For each pair of states $p$ and $q$ in $P$ the grammar will contain a variable $A_{pq}$.
- This variable will generate all strings that can take $P$ from state $p$ with empty stack to $q$ with an empty stack.
- Clearly, all those strings can also take $P$ from $p$ to $q$, regardless of the stack contents at $p$, leaving the stack at $q$ in the same condition as it was at $p$. 
First we modify $P$ slightly to give it the following three features.

1. It has a single accept state, $q_{accept}$.

2. It empties its stack before accepting.

3. Each transition either pushes a symbol onto stack (a *push* move) or pops one off the stack (a *pop* move), but does not do both at the same time.
Design a Grammar (ideas)

- For any string $x$ that take $P$ from $p$ to $q$, starting and ending with an empty stack, $P$’s first move on $x$ must be a push; the last move on $x$ must be a pop. (Why?)
- If the symbol pushed at the beginning is the symbol popped at the end, the stack is empty only at the beginning and the end of $P$’s computation on $x$.
  - We simulate this by the rule $A_{pq} \rightarrow aA_{rs}b$, where $a$ is the input symbol read at the first move, $b$ is the symbol read at the last move, $r$ is the state following $p$, and $s$ the state preceding $q$.
  - Else, the initially pushed symbol must get popped at some point before the end of $x$, and thus the stack becomes empty at this point.
    - We simulate this by the rule $A_{pq} \rightarrow A_{pr}A_{rq}$, $r$ is the state when the stack becomes empty.
Formal Design

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

We construct \( G \) as follows.

- The variables are \( \{A_{pq} : p, q \in Q\} \).
- The start variable is \( A_{q_0q_{accept}} \).
- Rules:
  - For each \( p,q,r,s \) from \( Q, \quad t \in \Gamma, \quad a,b \in \Sigma_e \) if we have
  - \( a, \varepsilon \rightarrow t \) \quad \( \bullet \quad \bullet \quad \bullet \) \quad \( b, t \rightarrow \varepsilon \)
  - put the rule \( A_{pq} \rightarrow aA_{rs}b \) in \( G \).
  - For each \( p,q,r \) from \( Q, \) put the rule \( A_{pq} \rightarrow A_{pr}A_{rq} \) in \( G \).
  - For each \( p \) from \( Q, \) put the rule \( A_{pp} \rightarrow \varepsilon \) in \( G \).