**Chomsky Normal Form (C.N.F.)**

**Def** A c.f. grammar is in Chomsky normal form if every rule is of the form:

- \( S \rightarrow \varepsilon \) (\( S \) is the start variable)
- \( A \rightarrow a \) (\( A \) is a variable, \( a \) is a terminal)
- \( A \rightarrow BC \) (\( A, B, C \) are variables, \( B, C \) are not start variable)

We note the following theorem.

**Theorem** Every c.f. language is generated by a c.f. grammar in the Chomsky normal form.

Having the grammar in a "normal" (i.e. standardized) form often makes reasoning about the grammar more convenient.

E.g. If the grammar is in C.N.F., then a string of terminals of length \( n \) is
generated in about \(2n\) steps: \(n\) applications of the rule of the type \(A \rightarrow BC\) and then \(n\) applications of the rule of the type \(A \rightarrow a\), converting all variables to terminals.

We will prove the theorem by giving a procedure that takes a c.f. grammar and converts it into an equivalent grammar in c.n.f. We skip the formal proof and only illustrate through an example.

Suppose the given grammar is:

\[
G: \\
S \rightarrow ASA | aB \\
A \rightarrow B | S \\
B \rightarrow b | \varepsilon
\]

The rules of type \(B \rightarrow \varepsilon\) (\(B \neq\) start var.) are called \(\varepsilon\)-rules and those of type \(A \rightarrow B\) are called unit rules. We need to remove them.
We begin by introducing a new start variable $S_0$ and adding the rule $S_0 \rightarrow S$.

**Step 1:** Add a new start variable

- $S_0 \rightarrow S$
- $S \rightarrow ASA | aB$
- $A \rightarrow B | S$
- $B \rightarrow b | \varepsilon$

**Step 2:** Remove $\varepsilon$-rules

To remove a rule $B \rightarrow \varepsilon$, we delete it and replace every rule $A \rightarrow uBv$ by $A \rightarrow uBv | uv$

Thus removing the rule $B \rightarrow \varepsilon$ from above gives

- $S_0 \rightarrow S$
- $S \rightarrow ASA | aB | a$
- $A \rightarrow B | \varepsilon | S$
- $B \rightarrow b$

Now removing $A \rightarrow \varepsilon$ gives
$S_0 \rightarrow S$
$S \rightarrow aB \mid a \mid ASA \mid SA \mid AS \mid S$
$A \rightarrow B \mid S$
$B \rightarrow b$

Whenever we have a rule $S \rightarrow S$, $S$ is a variable, it is removed (safely).

**Step 3: Remove unit rules**

To remove a rule $A \rightarrow B$, delete it and for every rule $B \rightarrow u$, add the rule $A \rightarrow u$.

Thus removing the rule $A \rightarrow B$ gives

$S_0 \rightarrow S$
$S \rightarrow aB \mid a \mid ASA \mid SA \mid AS$
$A \rightarrow b \mid S$
$B \rightarrow b$

Removing the rule $A \rightarrow S$ gives
\[ S_0 \rightarrow S \]
\[ S \rightarrow aB \mid a \mid ASA \mid SA \mid AS \]
\[ A \rightarrow b \mid aB \mid a \mid ASA \mid SA \mid AS \]
\[ B \rightarrow b \]

Removing the rule \( S_0 \rightarrow S \) gives
\[ S_0 \rightarrow aB \mid a \mid ASA \mid SA \mid AS \]
\[ S \rightarrow aB \mid a \mid ASA \mid SA \mid AS \]
\[ A \rightarrow b \mid aB \mid a \mid ASA \mid SA \mid AS \]
\[ B \rightarrow b \]

**Step 4: Breaking rules and Dummy variables.**

We can break a rule such as
\[ S \rightarrow ASA \]
into two rules
\[ S \rightarrow A_1A \]
\[ A_1 \rightarrow AS \].

Also a rule such as \( S \rightarrow aB \) can be replaced by \[ S \rightarrow X_aB \] where \( X_a \) is
a dummy variable that stands for the terminal \( a \). This gives

\[
S_0 \rightarrow X_a B \mid a \mid A_1 A \mid SA \mid AS \\
A_1 \rightarrow AS \\
X_a \rightarrow a \\
S \rightarrow X_a B \mid a \mid A_1 A \mid SA \mid AS \\
A \rightarrow b \mid X_a B \mid a \mid A_1 A \mid SA \mid AS \\
B \rightarrow b
\]

The grammar is now in Chomsky normal form.
Push-Down Automata (PDA)

We now study PDAs, a computational model that characterizes c.f. languages.

<table>
<thead>
<tr>
<th>Regular Languages</th>
<th>Context Free lang.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntactic Characterization</td>
<td>Regular expressions</td>
</tr>
<tr>
<td>Computational Characterization</td>
<td>Finite Automata</td>
</tr>
<tr>
<td>Push-Down Automata</td>
<td>Context free grammars</td>
</tr>
</tbody>
</table>

- Read input symbol
- Enter (possibly) new state
- Move input pointer to the right (so input is 1-way, read-only)
- Read input symbol and top stack symbol
- Enter (possibly) new state
- Replace or push or pop top stack symbol
- Move input pointer to the right
In both FA and PDA, accept if the input is exhausted and the m/c is in an accept state.

Example \( L = \{ 0^n 1^n \mid n \geq 1 \} \) is accepted by a PDA.

- Keep pushing 0's onto stack.
- After reading 1, pop a 0 from the stack and keep popping a 0 for every 1 read from the input.
- When the input is exhausted, accept if the stack is empty.

**Non-Deterministic PDA**

Example \( L = \{ W \cdot W^{\text{Reverse}} \mid W \in \{a,b,c\}^* \} \) is recognized by a non-deterministic PDA.
The proposed PDA should:
- push w onto the stack.
- After the string $w$ begins, pop the stack matching the top stack symbol with the input symbol, in every step.

However, how does the PDA "know" when w ends and $w^{Reverse}$ begins?

Answer: Non-deterministically "guess" the midpoint of the string $w^R$.

The non-det PDA is then:
- Keep pushing input symbols onto the stack.
- Non-deterministically enter a new state.
- Keep matching input symbols to top stack symbols and popping.

* Henceforth, PDA always means non-deterministic PDA. *
Formal Definition of a PDA

A PDA is a 6-tuple $M= (Q, \Sigma, \Gamma, S, q_0, F)$ where

- $Q$ is a finite set of states.
- $\Sigma$ is the set of input symbols.
- $\Gamma$ is the set of stack symbols.
- $q_0$ is the start state.
- $F \subseteq Q$ is the set of accept states.
- $S : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \rightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$ is the transition function where

$$\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}, \quad \Gamma_\varepsilon = \Gamma \cup \{\varepsilon\}.$$

The interpretation is that if $\delta(q, a, b)$ contains $(q', c)$ then

In state $q$, input $a$, top of stack $b$,

the PDA changes state to $q'$, replaces top stack symbol $b$ with $c$. 
Note Formally $S(q, a, b)$ is a set of possible moves, since the PDA is non-deterministic. It may be that $S(q, a, b) = \emptyset$.

Any of $a, b, c$ can be $\varepsilon$ as allowed by the definition of the transition function. This gives rise to 8 possible types of moves, 4 by reading an input symbol and 4 without " (i.e., $\varepsilon$-move).

We will note down all these moves below. Below, $a, b, c$ will denote input symbols and $\varepsilon$ will be written explicitly.

4- Moves by Reading an Input Symbol

Replace $S(q, a, b)$ contains $(q', c)$.
Pop  $S(q, a, b)$ contains $(q', \varepsilon)$.

Push  $S(q, a, \varepsilon)$ contains $(q', c)$

Stack  Unchanged / Untouched  $S(q, a, \varepsilon)$ contains $(q', \varepsilon)$. 
4 Moves Without Reading an Input Symbol

Replace \( S(q, \varepsilon, b) \) contains \( (q', c) \).

Pop \( S(q, \varepsilon, b) \) contains \( (q', \varepsilon) \).

Push \( S(q, \varepsilon, \varepsilon) \) contains \( (q', c) \).
Untouched $S(q, \epsilon, \epsilon)$ contains $(q', \epsilon)$.

Computation of a PDA

A PDA $M = (Q, \Sigma, \Gamma, S, q_1, F)$ on input $w \in \Sigma^*$:

- Starts in the initial configuration

  Where State = $q_1$, Stack is empty.

- Makes moves according to transition $f^*$.

- Accepts if input is exhausted and the state is an accept state (i.e., in $F$).
Keeping in mind that the PDA is non-deterministic, the language recognized by the PDA $M$ is:

$$ L(M) = \{ w \in \Sigma^* \mid \text{There exists a computation of } M \text{ on } w \text{ that accepts} \} $$

Clarification - The language $L(M)$ is defined only over the input alphabet $\Sigma$.

- Formally, input alphabet $\Sigma$ and the stack alphabet $\Gamma$ are disjoint. However, one can argue as if $\Sigma \subset \Gamma$ since for every $a \in \Sigma$, one can have corresponding symbol $X_a \in \Gamma$ that stands for $a \in \Sigma$.

PDAs are represented by state diagrams.

$$ a, b \rightarrow c $$

represents the move $\delta(q, a, b) \text{ contains } (q', c)$. 
Example State diagram for the PDA that recognizes the language
\[ L = \{ 0^n 1^n \mid n \geq 0 \} \].

\[ q_1 \xrightarrow{\varepsilon, \varepsilon \rightarrow \$} q_2 \xrightarrow{0, 3 \rightarrow 0 \text{ (push)}} q_3 \xrightarrow{1, 0 \rightarrow 3 \text{ (pop)}} q_1 \xrightarrow{\varepsilon, \$ \rightarrow \varepsilon} \]

There is no formal mechanism to test whether the stack is empty, so the PDA begins by pushing \$ symbol onto the stack, later, whenever the top stack symbol is \$, it effectively amounts to the stack being empty. On input 00001111

\[
\begin{array}{c}
\text{Read} \\
\text{0000} \rightarrow \\
\$
\end{array}
\quad
\begin{array}{c}
\text{Read} \\
\text{1111} \\
\$
\end{array}
\rightarrow \text{Accept.}
\]
Example State diagram for the PDA that recognizes the language

\[ L = \{ \text{Reverse} \mid \text{Reverse} \in \{0,1\}^* \} \]

\[ q_1 \xrightarrow{\epsilon, \epsilon \rightarrow \$} q_2 \xrightarrow{0, \epsilon \rightarrow 0} \text{(push)} \]
\[ q_2 \xrightarrow{1, \epsilon \rightarrow 1} (\text{non-det}) \]
\[ q_2 \xrightarrow{\epsilon, \epsilon \rightarrow \epsilon} q_3 \xrightarrow{0, 0 \rightarrow \epsilon} \text{(match and pop)} \]
\[ q_2 \xrightarrow{1, 1 \rightarrow \epsilon} q_3 \]

The move \( q_2 \rightarrow q_3 \) is a non-deterministic move where the PDA "believes" that the midpoint of the string \( \text{WW Reverse} \) has been reached.

On input \( 0110 \), several possible computations are possible as:

\[ \text{depending on "guess" of the midpoint} \]
\[ q, \]
\[ 0110 \]
"incorrect guess"