Chapter 4

Normal Forms for CFGs
4.5 Chomsky Normal Form

- **Defn 4.4.1** A CFG $G = (V, \Sigma, P, S)$ is in *Chomsky normal form* if each rule in $G$ has one of the following forms:
  
  i) $A \rightarrow BC$
  
  ii) $A \rightarrow a$
  
  iii) $S \rightarrow \lambda$

  where $A, B, C \in V$ and $B, C \in V - \{S\}$ and $a \in \Sigma$

- A *simplified* normal form which restricts the length & composition of the R.H.S. of a rule in CFG

- The *derivation tree* for a string generated by a CFG in Chomsky normal form is a *binary tree*
4.5 Chomsky Normal Form

Theorem 4.4.2 Let $G = (V, \Sigma, P, S)$ be a CFG. There is an algorithm to construct a grammar $G' = (V', \Sigma', P', S')$ in chomsky normal form that is equivalent to $G$.

Proof (sketch):

(i) For each rule $A \rightarrow w$, where $|w| > 1$, replace each terminal $a$ in $w$ by a distinct variable $Y$ & create new rule $Y \rightarrow a$.

(ii) For each modified rule $X \rightarrow w'$, $w$ is either a terminal or a string in $V^*$. Rules in the latter form must be broken into a sequence of rules, each of whose R.H.S. consists of two variables.

Example 4.4.1

One of the applications of using CFGs that are in Chomsky Normal Form:

- Constructing binary search trees to accomplish “optimal” time & space search complexity for parsing an input string.
4.1 Grammar Transformations

- **Lemma 4.1.1** Let $G = (V, \Sigma, P, S)$ be a CFG. There is a
  CFG $G' = (V', \Sigma, P', S')$ that satisfies
  
  i) $L(G) = L(G')$

  ii) Rules in $P'$ are of the form

  $$A \rightarrow w$$

  where $A \in V'$ and $w \in ((V - \{S\}) \cup \Sigma)^*$.

- **Proof.** If $S$ is a *recursive variable*, then construct $G'$ by creating a new *start symbol* $S'$ & adding $S' \rightarrow S$ to $P'$, i.e.,

  $$G' = (V \cup \{S'\}, \Sigma, P \cup \{S' \rightarrow S\}, S').$$

  If $S \xrightarrow{G} u$, then $S' \xrightarrow{G'} S \xrightarrow{G'}^* u$, where $u \in \Sigma^*$.

- **Example.** 4.1.1 Assume that $P$ in $G$ includes

  $$S \rightarrow aS \mid AB \mid AC,$$

  then $P'$ in $G'$ should include $S' \rightarrow S$, $S \rightarrow aS \mid AB \mid AC$
4.2 Elimination of $\lambda$-rules

- Nullable variables are variables that can derive $\lambda$.

- A grammar w/o nullable variables is called non-contracting grammar since the application of a rule cannot decrease the length of the sentential form.

- It is desirable to avoid the generation of (nullable) variables that are subsequently removed by $\lambda$-rules.

- The removal of nullable variables in a grammar guarantees that during process of the deriving a terminal string, each variable generates terminal symbol(s).

- The derivation of a terminal string in a grammar $G$ is more cost-effective if $G$ is noncontracting than if $G$ is contracting.
4.2 Elimination of $\lambda$-rules

Algorithm 4.2.1 Construction of Sets of Nullable Variables

*Input*: A CFG $G = (V, \Sigma, P, S)$

*Output*: Set of Nullable Variables

1. $\text{NULL} := \{ A \mid A \rightarrow \lambda \in P \}$
2. Repeat
   2.1. $\text{PREV} := \text{NULL}$
   2.2. For each variable $A \in V$ do
         If $\exists A \rightarrow w$, where $w \in \text{PREV}^*$ do
             $\text{NULL} := \text{NULL} \cup \{ A \}$
   Until $\text{NULL} = \text{PREV}$.

(Note: $w \in \text{PREV}^*$ indicates that $A (\rightarrow w)$ produces entirely *nullable* variables)
4.2 Elimination of $\lambda$-rules

Example 4.2.1 Given the following CFG

\[
S \rightarrow ACA \\
A \rightarrow aAa \mid B \mid C \\
B \rightarrow bB \mid b \\
C \rightarrow cC \mid \lambda
\]

Using Algorithm 4.1.2, the set of nullable variables can be computed

<table>
<thead>
<tr>
<th>Iteration</th>
<th>NULL</th>
<th>PREV</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{ C }</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>{ A, C}</td>
<td>{ C }</td>
</tr>
<tr>
<td>2</td>
<td>{ S, A, C}</td>
<td>{ A, C}</td>
</tr>
<tr>
<td>3</td>
<td>{ S, A, C}</td>
<td>{ S, A, C}</td>
</tr>
</tbody>
</table>
4.2 Elimination of $\lambda$-rules

- An essentially noncontracting grammar $G$
  - includes $S \to \lambda$, if $\lambda \in L(G)$
  - excludes any $\lambda$-rules
  - yields only noncontracting derivations, with the exception of $S \Rightarrow \lambda$.

- **Theorem 4.2.3** Let $G = (V, \Sigma, P, S)$ be a CFG. There is a CFG $G_L = (V_L, \Sigma, P_L, S_L)$ w/o $\lambda$-rules that satisfies
  - $L(G_L) = L(G)$.
  - $S_L$ is not a recursive variable.
  - $A \to \lambda \in P_L$ iff $\lambda \in L(G) \& A = S_L$. 
4.2 Elimination of $\lambda$-rules

- **Constructing $G_L$ from $G$.**

  (i) $S_L$ is **not** a recursive variable. Any recursive start variable can be made nonrecursive according to Lemma 4.1.1.

  (ii) If $\lambda \in L(G)$, then $S_L \to \lambda \in P_L$.

  (iii) Delete all the $\lambda$-rules in $G$, except $S_L \to \lambda$, from $P_L$ as follows:

  If $A \to w \in P$, $w = w_1 A_1 w_2 A_2 \ldots w_k A_k w_{k+1}$, and $A_1, \ldots, A_k$ are nullable variables, then create $2^k - 1$ new rules in $P_L$, i.e., a set of $2^k$ possible combinations with the inclusion and exclusion of $A_1, A_2, \ldots, A_k$.

- **Example 4.2.2** \{ $S$, $A$, $C$ \} are nullable variables of $G$, where

  $$
  G: S \to ACA \\
  A \to aAa | B | C \\
  B \to bB | b \\
  C \to cC | \lambda
  $$

  $$
  G_L: S \to ACA | AC | CA | AA | A | C | \lambda \\
  A \to aAa | aa | B | C \\
  B \to bB | b \\
  C \to cC | c
  $$
4.2 Elimination of $\lambda$-rules

Example 4.2.3 Let $G$ be the grammar

$$
\begin{align*}
S & \rightarrow ABC \\
A & \rightarrow aA | \lambda \\
B & \rightarrow bB | \lambda \\
C & \rightarrow cC | \lambda
\end{align*}
$$

$G$ generates $a^*b^*c^*$. The nullable variables of $G$ are $S$, $A$, $B$, and $C$. The equivalent grammar of $G$ w/o $\lambda$-rules is $G_L$, where

$$
\begin{align*}
S & \rightarrow ABC | AB | AC | BC | A | B | C | \lambda \\
A & \rightarrow aA | a \\
B & \rightarrow bB | b \\
C & \rightarrow cC | c
\end{align*}
$$
4.3 Elimination of Chain rules

- **Definition.** A rule of the form \( A \rightarrow B \), where \( A, B \in V \), which simply renames a variable in a derivation, is a *chain rule*.

- The removal of chain rules
  - *increase* the number of *rules* in the grammar, but
  - *reduce* the *length* of derivations

- **Example.** Consider the set of rules \( P \)
  
  \[
  A \rightarrow aA \mid a \mid B \\
  B \rightarrow bB \mid b \mid C
  \]

  Eliminating the chain rule \( A \rightarrow B \) by (i) adding \( A \rightarrow w \), for every rule \( B \rightarrow w \), and (ii) delete \( A \rightarrow B \).

  Hence, \( P \) is modified as
  
  \[
  A \rightarrow aA \mid a \mid bB \mid b \mid C \\
  B \rightarrow bB \mid b \mid C
  \]

  However, another *chain rule* \( A \rightarrow C \) was created.
4.3 Elimination of Chain rules

**Algorithm 4.3.1. Construction of the Set CHAIN(A)**

*Input: A CFG \( G = (V, \Sigma, P, S) \) and variable \( A \)*

*Output: The set of chain rules of \( A \)*

1. \( \text{CHAIN}(A) := \{ A \} \)
2. \( \text{PREV} := \emptyset \)
3. **Repeat**
   3.1. \( \text{NEW} := \text{CHAIN}(A) – \text{PREV} \)
   3.2. \( \text{PREV} := \text{CHAIN}(A) \)
   3.3. **For each variable** \( B \in \text{NEW} \) **do**
       **For each rule** \( B \rightarrow C \) **do**
       \( \text{CHAIN}(A) := \text{CHAIN}(A) \cup \{ C \} \)
   **Until** \( \text{CHAIN}(A) = \text{PREV} \).
4.3 Elimination of Chain rules

Theorem 4.3.3 Let \( G = (V, \Sigma, P, S) \) be a CFG. There is a CFG \( G_C = (V, \Sigma, P_C, S) \) that satisfies

1) \( L(G_C) = L(G) \).
2) \( G_C \) has no chain rules.

Proof. Using \( P & CHAIN(A) \), we compute the \( A \) rules in \( G_C \).

The rule \( A \to w \) is in \( P_C \) if \( \exists \) variable \( B \) & string \( w \).

i) \( B \in CHAIN(A) \)

ii) \( B \to w \in P \)

iii) \( w \notin V \) (ensures that \( P_C \) does not contain chain rules).

Let \( w \in L(G) & A \Rightarrow_B \) be a maximal sequence of chain rules used to derive \( w \), which can be generated by

\[
\begin{align*}
S & \Rightarrow^{*} uAv \Rightarrow^{*} uBv \Rightarrow^{*} upv \Rightarrow^{*} w, \\
S & \Rightarrow^{*} uAv \Rightarrow^{*} upv \Rightarrow^{*} w
\end{align*}
\]

where \( B \to p \in P \) is not a chain rule.
4.3 Elimination of Chain rules

Example 4.3.1. Given the following CFG

\[
S \rightarrow ACA \mid CA \mid AA \mid AC \mid A \mid C \mid \lambda
\]
\[
A \rightarrow aAa \mid aa \mid B \mid C
\]
\[
B \rightarrow bB \mid b
\]
\[
C \rightarrow cC \mid c
\]

Using Algorithm 4.3.1, we generate

\[
\text{CHAIN}(S) = \{ S, A, C, B \}
\]
\[
\text{CHAIN}(A) = \{ A, C, B \}
\]
\[
\text{CHAIN}(B) = \{ B \}
\]
\[
\text{CHAIN}(C) = \{ C \}
\]

Using these CHAIN sets, we generate \( G_C \), where \( P_C \in G_C \) is

\[
S \rightarrow ACA \mid CA \mid AA \mid AC \mid aAa \mid aa \mid bB \mid b \mid cC \mid c \mid \lambda
\]
\[
A \rightarrow aAa \mid aa \mid bB \mid b \mid cC \mid c
\]
\[
B \rightarrow bB \mid b
\]
\[
C \rightarrow cC \mid c
\]
4.7 Removal of Direct Left Recursion

- Recursion is necessary to generate strings of arbitrary length. Left recursion causes problems, not recursion in general.

- *Directly left recursive rules* (e.g. $A \rightarrow Aa$) introduce the possibility of unending computations since repeated applications of them fail to generate a *prefix* that can terminate the parse.

- To avoid the possibility of a non-terminating parse, directly left recursive rules must be removed.

- **Example.**

<table>
<thead>
<tr>
<th>Direct left-recursive rules</th>
<th>No direct left-recursive rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G_1$: $A \rightarrow Aa \mid b$</td>
<td>$G_1'$: $A \rightarrow bZ \mid b, Z \rightarrow aZ \mid a$</td>
</tr>
<tr>
<td>$G_2$: $A \rightarrow Aa \mid Ab \mid b \mid c$</td>
<td>$G_2'$: $A \rightarrow bZ \mid cZ \mid b \mid c$</td>
</tr>
<tr>
<td>$G_3$: $A \rightarrow AB \mid BA \mid a$</td>
<td>$G_3'$: $A \rightarrow BAZ \mid aZ \mid BA \mid a$</td>
</tr>
<tr>
<td>$B \rightarrow b \mid c$</td>
<td>$Z \rightarrow BZ \mid B$</td>
</tr>
<tr>
<td></td>
<td>$B \rightarrow b \mid c$</td>
</tr>
</tbody>
</table>
4.7 Removal of Direct Left Recursion

- The removal of any *direct left recursion* requires the addition of a new variable $V$, which introduces a set of *directly right recursive rules*.

- To remove direct left recursion on variable $A$, $A$ is divided into:
  - *directly left recursive rules*: $A \rightarrow A u_1 | A u_2 | \ldots | A u_j$
  - *non-directly left recursive rules*: $A \rightarrow v_1 | v_2 | \ldots | v_k$, where $A$ is not the first symbol in $v_i$ ($1 \leq i \leq k$)

- **Strategy**: build a string in a left-to-right manner by applying (i) *non-recursive* rules first, followed by (ii) constructing the remaining symbols by right recursion.
  $$A \rightarrow v_1 | \ldots | v_k | v_1 Z | \ldots | v_k Z$$
  $$Z \rightarrow u_1 | \ldots | u_j | u_1 Z | \ldots | u_j Z$$

- **Example 4.5.1** Given the rules $A \rightarrow Aa | Aa b | bb | b$
  
The $A$ rules are: $A \rightarrow bb | b | bb Z | b Z$
  
The $Z$ rules are: $Z \rightarrow a | ab | a Z | ab Z$