In general, to describe a language, there are two possible approaches:

1) **Recognition**: describe rules (or a mechanism) to determine whether or not a certain string belongs to a language.

   e.g. an automaton is such a mechanism.

2) **Generation**: define rules to generate all strings of a language.

A grammar is a formalism for defining a language in terms of rules that generate all strings of the language.

Since 1920, various formal methods based on the notion of rewriting or derivation have been proposed by Axel Thue, Emil Post, A.A. Markov.

In the mid 1950s, the linguist Noam Chomsky introduced the notion of formal grammar with the aim of formalizing natural language. Formal grammars are in fact too simplistic to capture natural language, but they were adopted as the main formal tool to define syntactic properties of artificial languages (e.g. programming languages).
Definition: Given an alphabet $\Sigma$, a (formal) grammar $G$ is a quadruple $G = (V_N, V_T, P, S)$ where

- $V_T \subseteq \Sigma$ is a finite nonempty set of symbols called terminals
- $V_N$ is a finite nonempty set of symbols s.t. $V_N \cap \Sigma = \emptyset$, called variables or nonterminals, or syntactic categories.
- Each variable represents a language
- $S \in V_N$ is called start symbol or axiom, and represents the language being defined by $G$
- $P$ is a binary relation over $(V_N \cup V_T)^* \times V_N \times (V_N \cup V_T)^*$

Each element $(\alpha, \beta) \in P$ is called a production or rule, and is generally written as: $\alpha \rightarrow \beta$.

Note: $\alpha$ ... sequence of terminals and nonterminals with at least one nonterminal

$\beta$ ... sequence of terminals and nonterminals

Definition: The language $L(G)$ generated by a grammar $G$ is the set of strings of terminals only that can be generated starting from the axiom by a finite sequence of rule applications. Each application of a rule $\alpha \rightarrow \beta$ consists in replacing an occurrence of $\alpha$ with $\beta$. 
Example: Palindromes:

A palindrome is a word that reads the same both forwards and backwards. (ITALIATILAJA, AMORONA)

\[ L_{pal} = \{ w \in \{0, 1\}^* \mid w^R = w \} \]

Grammar \( G_{pal} = (V_N, V_T, P, S) \), where \( P \) consists of:

1) \( S \rightarrow \epsilon \)
2) \( S \rightarrow 0 \)
3) \( S \rightarrow 1 \)

4) \( S \rightarrow 0SO \) [basis: \( \epsilon, 0, 1 \) are palindromes]
5) \( S \rightarrow 1S1 \) [induction: if \( S \) is a palindrome, so are \( OSO \) and \( 1S1 \)]

Example of derivation:

\[ 0110 : S \Rightarrow 0SO \Rightarrow 01S10 \Rightarrow 0110 \]

\[ 11011 : S \Rightarrow 1S1 \Rightarrow 11S11 \Rightarrow 11011 \]

Exercise E5.1: Prove that the above grammar generates all and only palindromes over \( \{0, 1\} \).

Hint: use induction on the length of the derivation.

Example: Natural language generation

Sentence \( \rightarrow \) NounPhrase VerbPhrase
NounPhrase \( \rightarrow \) Adjective NounPhrase
NounPhrase \( \rightarrow \) Noun
Noun \( \rightarrow \) car
Noun \( \rightarrow \) train
Adjective \( \rightarrow \) big
Adjective \( \rightarrow \) broken
Notation:

1) To denote the set of productions
\[ \alpha \rightarrow \beta_1, \alpha \rightarrow \beta_2, \ldots, \alpha \rightarrow \beta_m \]
we use
\[ \alpha \rightarrow \beta_1 | \beta_2 | \ldots | \beta_m \]

2) We use \( V = V_r \cup V_T \)

A production of the form \( \alpha \rightarrow \varepsilon \), with \( \alpha \in V^* \cdot V_r \cdot V^* \)

is called \( \varepsilon \)-production.

Example: \( L_{eq} = \{ w \in \{0,1\}^* | w \text{ has equal number of 0's and 1's} \} \)

We have already seen that this language is not regular.

Idea to define \( G_{eq} \) s.t. \( L(G_{eq}) = L_{eq} \): use induction.

Base: \( \varepsilon \in \varepsilon \cdot L_{eq} \)

Induction: \( \exists w_a \in \varepsilon \cdot L_{eq} \) if \( w_a \) has one more 1 than 0.

\( \exists w_b \in \varepsilon \cdot L_{eq} \) if \( w_b \) has one more 0 than 1.

Characterize also languages for \( w_a \) and \( w_b \) inductively.

Grammar \( G_{eq} = (\{S, A, B\}, \{0,1\}, \{1,0\}, S) \) with \( P \)
\[
S \rightarrow \varepsilon | OA | 1B \quad (A \text{ generates strings with one more 1 than 0})
A \rightarrow AS | OAA
B \rightarrow BS | 1BB \quad (B \text{ generates strings with one more 0 than 1})
\]

Exercise E5.2: Prove that \( L(G_{eq}) = L_{eq} \) (by induction).
Definition: Given $G$, the direct derivation for $G$ is the binary relation on $(V^*, V_m, V^*) \times V^*$ defined as follows:

$(\varphi, \psi)$ is in the relation if there are

\[ \alpha \in V^*, V_m, V^*, \quad \beta, \gamma, \delta \in V^* \]

such that

\[ \varphi = \alpha \beta \gamma \delta, \quad \psi = \beta \gamma \delta \quad \text{and} \quad \alpha \Rightarrow \beta \in \mathcal{P}. \]

We write $\varphi \Rightarrow \psi$ and say that $\psi$ directly derives from $\varphi$ by $G$.

Definition: We call derivation the reflexive, transitive closure of direct derivation. In other words, $\psi$ derives from $\varphi$ by $G$, written $\varphi \Rightarrow^* \psi$, if

a) $\varphi = \psi$, or

b) there are $\varphi_1, \ldots, \varphi_n \in V^*$ such that

\[ \varphi_1 = \varphi, \quad \varphi_n = \psi, \quad \text{and} \quad \varphi_i \Rightarrow \varphi_{i+1}, \quad \forall i, 1 \leq i < n \]

Definition: Given a grammar $G$, the language generated by $G$ is

\[ L(G) = \{ w \in V_T^* \mid S \Rightarrow^* w \} \]

Notice: words in $L(G)$ are constituted by terminals only.

Terminology:

- sentence: any word $w \in V_T^*$ s.t. $S \Rightarrow^* w$, i.e. $w \in L(G)

- sentential form: any $\alpha \in V^* = (V_T \cup V_N)^*$ s.t. $S \Rightarrow \alpha$

Notation: terminals: $\mathcal{A}, B, C$

nonterminals: $A, B, C$

strings of terminals: $w, \overline{w}, w, x, y, z$

symbols of $V = V_N \cup V_T$: $K, Y, 2$

sentential forms: $A, B, C$
Example: Productions for $G_{eq}$:

$$S \rightarrow \varepsilon \mid 0A \mid 1B$$
$$A \rightarrow 1S \mid 0AA$$
$$B \rightarrow 0S \mid 1BB$$

Derivation:

1) $001SA \Rightarrow 001S\varepsilon$ (using $A \rightarrow 1S$)
2) $001S\varepsilon \Rightarrow 001S\varepsilon$ (using $S \rightarrow \varepsilon$)
3) $001SA \Rightarrow 001S\varepsilon$ (using (1) and (2))
4) $S \Rightarrow 001\varepsilon\varepsilon$

Example: Grammar for $L_{eq} = \{a^n b^n c^m \mid n \geq 1\}$

$G_{eq} = (\{A, B, C, S\}, \{a, b, c\}, P, S)$

with $P$

1) $S \rightarrow aSBC$
2) $S \rightarrow aBC$
3) $CB \rightarrow BC$
4) $aB \rightarrow ab$
5) $bB \rightarrow bb$
6) $bC \rightarrow bc$
7) $CC \rightarrow cC$

Example of derivation of $aabbcccc$:

$$S \Rightarrow aSBC \Rightarrow aabSBCBC \Rightarrow aabBCBCBC \Rightarrow aabBCBCBCBC$$

Note: We cannot simply have $B \rightarrow b$, $C \rightarrow c$ because this would generate many words not in $L_{eq}$.
Note: not each sequence of direct derivations leads to a sentence in $L(G_{3n})$.

E.g. with the previous grammar we could generate

$S \Rightarrow aSBc \Rightarrow aeaBCBc \Rightarrow eeaBCBc$  
$\Rightarrow eeaBcBBc \Rightarrow eeaBcBc$  

we cannot apply any other production.

Also, the application of productions could go on forever (e.g. rule 1 in the previous example).

Classification of Chomsky grammars into 4 groups, depending on the form of the productions:

- grammars of type 0: no limitations
- 1: context-sensitive
- 2: context-free
- 3: regular (or right linear)

Definition: grammar of type 0.

Productions have the most general form $a \Rightarrow b$, with $a \in V \cup V^*$ and $b \in V^*$.

Grammars of type 0 allow for derivations that shorten the sentential form.

A language generated by a grammar of type 0 is called of type 0.
Definition: Grammar of type 1, or context-sensitive

Productions have the form $\alpha \to \beta$, with

$\alpha \in V^* \cdot V_N \cdot V^*$, $\beta \in V^+$, $|\alpha| \leq |\beta|$

These productions cannot shorten the length of the sentential form to which they are applied.

A language generated by a grammar of type 1 is called of type 1, or context-sensitive.

Example: $G_{3n}$ is context-sensitive. Obviously, it is also of type 0.

Definition: Grammar of type 2, or context-free

Productions have the form $A \to \beta$, with $A \in V_N$, $\beta \in V^+$.

These productions are productions of type 1, with the additional requirement that on the left there is a single nonterminal.

A language generated by a grammar of type 2 is called of type 2, or context-free.

Example: $L_{2n} = \{a^n b^n \mid n \geq 1\}$ is of type 1, since the following grammar $G'_{2n}$ generates $L_{2n}$

$S \to aB \mid SAa$
$BA \to AB$
$AA \to aA$
$AB \to aB$
$bB \to bb$

$L_{2n}$ is also of type 2, since it is generated by

$S \to aSb \mid ab$
We said that grammars of type 1 are also called context-sensitive (in contrast to context-free grammars). This is justified by the original definition by Chomsky for context-sensitive grammars.

**Definition:** Chomsky CS-grammar

**Productions** have the form \( \gamma_1 A \gamma_2 \rightarrow \gamma_3 \beta \gamma_4 \)
with \( \gamma_1, \gamma_2 \in V^*, \ A \in V_N, \ \beta \in V^+ \)

Intuitively, \( A \) is replaced by \( \beta \) only if it appears "in the context" of \( \gamma_1 \) and \( \gamma_2 \)

**Theorem:** Grammars of type 1 and Chomsky CS grammars generate the same class of languages

**Proof:** We show that, for every language \( L \):

There is a type-1 grammar \( G_1 \) s.t. \( L = L(G_1) \) iff there is a Chomsky CS grammar \( G_C \) s.t. \( L = L(G_C) \)

"if": immediate, since each Chomsky CS grammar is of type 1 (in \( \gamma_1 A \gamma_2 \rightarrow \gamma_3 \beta \gamma_4 \) we have \( \beta \in V^+ \) and hence \( |\gamma_1 A \gamma_2| \leq |\gamma_3 \beta \gamma_4| \))

"only-if": let \( G_1 \) be a type-1 grammar for \( L \).

We construct from \( G_1 \) a Chomsky CS grammar \( G_C \) as follows:

1) For each \( \epsilon \in V_F \), add a new non-terminal \( N_0 \).
2) Replace in each production of \( G_1 \), each \( \epsilon \in V_F \) by \( N_0 \)

Now all productions have the form

\[ A_1 A_2 \ldots A_m \rightarrow B_1 B_2 \ldots B_n \] 
with \( m \leq n \) 
and all \( A_i, B_j \in V_N \)
3) For each rule production $A_1 \cdots A_m \rightarrow B_1 \cdots B_n$, introduce a new nonterminal $N$, and replace the production by the following ones:

$$A_1 A_2 \cdots A_m \rightarrow N A_2 \cdots A_m$$

$$N A_2 \cdots A_m \rightarrow N B_2 A_3 \cdots A_m$$

$$N B_2 A_3 \cdots A_m \rightarrow N B_2 B_3 A_4 \cdots A_m$$

$$\vdots$$

$$N B_2 \cdots B_{m-1} A_m \rightarrow N B_2 \cdots B_{m-1} B_m \cdots B_n$$

$$N B_2 \cdots B_n \rightarrow B_1 B_2 \cdots B_n$$

Observe that all such productions are of the form $\gamma_1 A_2 \gamma_2 \rightarrow \gamma_1 \beta \gamma_2$ with $\gamma_1, \gamma_2 \in V^*$, $\gamma \in V_N$, $\beta \in V^+$.

4) For each $\epsilon \in V_1$, add the production $N \epsilon \rightarrow \epsilon$ (where $N \epsilon$ is the new non-terminal associated to $\epsilon$)

It is not difficult to see that $L(G_1) = L(G_\epsilon)$

(the proof is by induction on the length of the derivation of a string $w \in L(G_1)$ (resp., $L(G_\epsilon)$))
Definition: grammar of type 3, or regular, or right-linear

Productions have the form \( A \rightarrow S \) with \( A \in V_N \)

\[ \delta \in V_T \cup (V_T \circ V_N) \]

(i.e., \( A \rightarrow aB \) or \( A \rightarrow a \), with \( A, B \in V_N, a \in V_T \))

A language generated by a grammar of type 3 is called of type 3 or regular

Example: \( \{ a^n b^n | n > 0 \} \) is of type 3, since it is generated by the grammar \( S \rightarrow aS \)

\[ S \rightarrow b \]

Note: a grammar of type 3 is called linear, because on the right-hand side of a production there is at most one non-terminal. It is called right-linear because the non-terminal is on the right of the terminal

\[ \text{Exercise: E5.3:} \]

Show that grammars of type 3 generate the class of regular languages that do not contain \( E \).

\[ \text{Hint: given } G = (V_N, V_T, P, S), \text{ construct an NFA } \]

\[ A_G = (V_N \cup \{ F \}; V_T, \delta, \hat{S}, \{ F \}) \text{ with } \]

\[ B \in \delta(A, a) \text{ iff } A \rightarrow aB \text{ and } \]

\[ F \in \delta(A, a) \text{ iff } A \rightarrow a \]

Show by induction on \( |w| \) that \( w \in L(G) \) iff \( w \in L(A_G) \).

Conversely, given an NFA \( A \), construct a grammar \( G_A \) by having again non-terminals correspond to states of \( A \).
Note on E-productions (for grammars of type 1, 2, 3)

As we have defined them, grammars of type 1 (resp. 2, 3)
cannot generate the empty string E.

We could extend the definition by allowing also the
generation of E:

* if the start symbol S does not appear on the right-hand
  side of productions, we allow also for a production
  \[ S \rightarrow E \]  
  (E-production)

* if the start symbol S appears on the right-hand side
  of productions, we introduce a new non-terminal Snow,
  make it the new start symbol, add a production
  \[ Snow \rightarrow S \] , and allow for \[ Snow \rightarrow E \].

Hence, an E-production used just to generate E is harmless.

Note that, allowing for E-productions for every non-terminal
is not that harmless.

Exercise: E5.4. Show that, for every language L of type 0
there is a grammar of type 1 extended with E-productions
on arbitrary non-terminals that generates L.

Hint: introduce a new non-terminal Ne that is eliminated
through an E-production \[ Ne \rightarrow E \] , and use Ne
to make the right-hand side of productions as long
as the left-hand side.
In a CFG, the productions have the form \[ A \rightarrow \beta \]
with \[ A \in V_N, \quad \beta \in V^* \] (note: we allow for \( \epsilon \)-productions).

Example: CFG for arithmetic expressions over variable \( \alpha \)
\[
G = (\{ E, T, F \}, \{ +, *, (, ) \}, P, E) \quad \text{where } P \text{ is}
\]
\[
P =
\begin{align*}
E & \rightarrow T | E + T & \text{E...expression} \\
T & \rightarrow F | T * F & \text{T...term} \\
F & \rightarrow \alpha | (E) & \text{F...factor}
\end{align*}
\]

This grammar generates, e.g., \( \alpha + \alpha * \alpha \)
\[
E \rightarrow E + T \rightarrow T + T \rightarrow E + T \rightarrow \alpha + T \rightarrow
\]
\[
\alpha + T * F \rightarrow \alpha + \alpha * F \rightarrow \alpha + \alpha * \alpha
\]

We can also represent a derivation of a string by a CFG by means of a tree, called parse tree:

In a tree whose nodes are labeled by elements of \( V \cup \Sigma \), satisfying:

1) Each interior node is labeled by a non-terminal
2) Each leaf is labeled by a non-terminal, a terminal, or \( \epsilon \). If it is labeled by \( \epsilon \), then it is the only child of its parent
3) If an interior node is labeled \( A \), and its children, from left to right are labeled \( X_1, X_2, \ldots, X_k \), then there is a production \( A \rightarrow X_1 X_2 \ldots X_k \) in \( P \).

Example: Parse tree for
\[
\alpha + \alpha * \alpha
\]
We call A-tree a subtree of the parse tree rooted at non-terminal A.

Yield (or frontier) of a tree:

is the sequence of labels of the leaves from left to right.

Example:

```
      E
     /\  
    E  T
   /\  / \
  E  T  T
 /\  /  / \
F T F
```

Theorem: \( \alpha \in V^+ \) is the yield of an A-tree \( \Rightarrow A \Rightarrow^* \alpha \)

Proof: by induction on the height of the tree

(see textbook)

8/1/2003

Note: a parse tree does not specify a unique way to derive \( \alpha \) from A. (the order in which non-terminals are expanded is not specified).

The parse tree specifies, however, which rule is applied for each non-terminal.

Specific derivation orders:

- Leftmost derivation: obtained by traversing the tree depth-first, by first going to the left subtree, and then to the right one.
  
  Example: \( E \Rightarrow E + T \Rightarrow I + T \Rightarrow E + T \Rightarrow I + I \Rightarrow \cdots \)

- Rightmost derivation: defined similarly: \( E \Rightarrow E + T \Rightarrow E + T * E \Rightarrow \cdots \)
Theorem: the following are all equivalent statements for a CFG $G = (V, T, P, S)$ and a string $w \in T^*$

1) $w \in L(G)$ (or $S \Rightarrow^* w$)
2) $S \xrightarrow{hm} w$
3) $S \xrightarrow{rm} w$
4) There exists an $S$-tree with yield $w$.

Proof: the equivalence of (1) and (4) follows from the previous theorem. The other equivalences are obvious.

Thus, we could always use hm-derivation as a canonical way to derive any $w \in L(G)$; i.e. as a canonical way to interpret a parse tree for $w$.

Ambiguous grammars:

- $w \in L(G)$ could have two distinct parse trees, and hence two distinct hm-derivations.

Example: another grammar for arithmetic expressions

$$E \rightarrow i \mid (E) \mid E + E \mid E \times E$$

$w = i + i \times i$

These parse trees correspond to two different hm-derivations, and also to two ways of interpreting $w$. 
**Definition:** A CFG for \( G \) is ambiguous if for some \( w \in L(G) \) there exist two distinct parse trees.

Ambiguity has to be avoided in compilers, since it corresponds to different ways of interpreting a string.

Sometimes grammars can be redesigned to remove ambiguity. (e.g., for arithmetic expressions)

This is not always possible:

**Definition:** A CFG language is (inherently) ambiguous if all its grammars are ambiguous.

**Example:** \( L = \{ a^n b^n c^{2m} d^m \mid n, m \geq 1 \} \cup \{ a^n b^{2n} c^n d^m \mid n, m \geq 1 \} \)

\( L \) is CF (show for exercise)

Consider strings of the form \( a^n b^n c^n d^n \). We cannot tell whether they come from first or second types of strings in \( L \), and any CFG must allow for both possibilities.
We will study:

1) Normal forms for CFGs (useful for proving properties of CFLs)
2) Expressive power -> pumping lemma for CFLs
3) Closure and decision properties

**Normal forms for CFGs**

We look at how to simplify CFGs, while preserving the generated language.
- gain efficiency in parsing
- simplify proving properties

9) **Eliminate useless symbols**

We say that $X \in V$ is useful if

$$S \Rightarrow^* \alpha\, K\, \beta \Rightarrow^* w \quad \text{with} \quad w \in V_T^*$$

$\alpha, \beta \in V^*$

Thus, a symbol is useless (not useful) if it does not participate in any derivation of strings of the language.

$\Rightarrow$ can be eliminated.

**Definition:** $X \in V$ is generating if $X \Rightarrow^* w$, for $w \in V_T^*$

$X \in V$ is reachable if $S \Rightarrow^* \alpha\, K\, \beta$, for $\alpha, \beta \in V^*$

Hence, $X$ is useful, if it is both generating and reachable.
We identify useless symbols by

1a) eliminating non-generating symbols and all their productions
1b) \( \rightarrow \) unreachable

Note: it is important to do these two steps in the above order.

Example: 
\[
\begin{align*}
S & \rightarrow AB & b \\
A & \rightarrow & a \\
\end{align*}
\]

- we eliminate unreachable symbols: all are reachable
- \( \rightarrow \) non-generating
we eliminate B and \( S \rightarrow AB \)

\( \Rightarrow \) we obtain: \( S \rightarrow b \)
\( A \rightarrow a \)

But if we do it in the right order:

1a) eliminate non-generating symbols: B and \( S \rightarrow AB \)
1b) \( \rightarrow \) unreachable: A and \( A \rightarrow a \)

\( \Rightarrow \) we obtain: \( S \rightarrow b \)

1a) Eliminating non-generating symbols:

We construct the set \( H \) of generating symbols, and then eliminate the non-generating symbols and the productions containing them.

Algorithm to construct the set \( H \) of generating symbols
Input: grammar \( G = (V_V, V_T, P, S) \)
Output: set \( H \) of generating symbols

\( H = V_T \)

while there is a change in \( H \) do

for each production \( A \rightarrow \chi_1 \ldots \chi_k \) in \( P \) do

if \( \{\chi_1, \ldots, \chi_k\} \subseteq H \) (i.e., all of \( \chi_1, \ldots, \chi_k \) are generating)

then \( H \leftarrow H \cup \{A\} \)

return \( H \)
Example: \( G_1 = (V_N, V_T, P, S) \)

with \( V_N = \{ S, A, B, C, D \} \), \( V_T = \{ a, b, c, d \} \).

\[
P: \begin{align*}
S & \rightarrow AB \mid AC \mid CD \\
A & \rightarrow BB \\
B & \rightarrow AC \mid ab \\
C & \rightarrow Ce \mid CC \\
D & \rightarrow Bc \mid b \mid d
\end{align*}
\]

Initialisation: \( H = \{ c, b, c, d \} \)
iteration 1: \( H = H \cup \{ B, D \} \)
\( \ldots \) 2: \( H = H \cup \{ A \} \)
\( \ldots \) 3: \( H = H \cup \{ S \} \)
\( \ldots \) 4: \( H \) does not change

C is not generating \( \Rightarrow \) Remove C and all productions containing C

16) Eliminating unreachable symbols

We construct the set \( R \) of reachable symbols, and then eliminate the symbols not in \( R \) and the productions containing them.

Algorithm to construct the set \( R \) of reachable symbols

Input: grammar \( G = (V_N, V_T, P, S) \)
Output: set \( R \) of reachable symbols

\( R \leftarrow \{ S \} \)

while there is a change in \( R \) do

for each production \( A \rightarrow X_1 \ldots X_n \) in \( P \) do

\( \text{if } A \in R \) (i.e., \( A \) is reachable)

\( \text{then } H \leftarrow H \cup \{ X_1, \ldots, X_n \} \)

return \( R \)

Example: \( G_2 = (V_N, V_T, P, S) \) with \( V_N = \{ S, A, B, D \} \), \( V_T = \{ a, b, c, d \} \)

\[P: \begin{align*}
S & \rightarrow AB \\
A & \rightarrow BB \\
B & \rightarrow ab \\
D & \rightarrow b \mid d
\end{align*}\]

Initialisation: \( R = \{ S \} \)
iteration 1: \( R \leftarrow R \cup \{ A, B \} \)
\( \ldots \) 2: \( R \leftarrow R \cup \{ a, b \} \)
\( \ldots \) 3: \( R \) does not change

D, c, d are unreachable

\( \Rightarrow \) Remove D, c, d, and all productions containing them.
Eliminate $E$-productions

$E$-production: $A \rightarrow E$ slows down parsing

Definition: $A \in V_N$ is nullable if $A \Rightarrow^* E$

We first need to find all nullable symbols

Algorithm to construct the set $N$ of nullable symbols

Input: grammar $G=(V_N, V_T, P, S)$
Output: set $N$ of nullable symbols

$N = \emptyset$

for each production $A \rightarrow E$ in $P$ do $N \leftarrow N \cup \{A\}$
while there is a change in $N$ do
for each production $A \rightarrow X_1 \ldots X_k$ in $P$ do
if $\{X_1, \ldots, X_k\} \subseteq N$ (i.e., all of $X_1, \ldots, X_k$ are nullable) then $N \leftarrow N \cup \{A\}$
return $N$

Example: $G_3=(V_N, V_T, P, S)$ with $V_N=\{S, A, B, C\}, V_T=\{x, b\}$

$P: \begin{align*}
S & \rightarrow ABC | BCB \\
A & \rightarrow aB | a \\
B & \rightarrow CC | b \\
C & \rightarrow S | E
\end{align*}$

Initialization: $N = \{C\}$

Iteration 1: $N \leftarrow N \cup \{B\}$

Iteration 2: $N \leftarrow N \cup \{S\}$

Iteration 3: $N$ does not change

Knowing the nullable symbols, allows us to compensate for the elimination of $E$-productions.

Example: in $G_3$, since $B$ and $C$ are nullable, we can derive:

$S \Rightarrow^* BCB, \quad S \Rightarrow^* CB, \quad S \Rightarrow^* BC, \quad S \Rightarrow^* BB$

$S \Rightarrow^* B, \quad S \Rightarrow^* C, \quad S \Rightarrow^* E$

Hence, if we eliminate $C \rightarrow E$ (and $B$, $C$ are not nullable anymore), we have to add direct productions for the above derivations.
Algorithm to eliminate \( E \)-productions

1) Identify all malleable symbols

2) Replace each production \( A \to X_1 \cdots X_n \)
   by the set of all productions of the form \( A \to \alpha_1 \cdots \alpha_n \)
   where \( \alpha_i = X_i \), if \( X_i \) is not malleable
   \( \alpha_i = \varepsilon \) or \( E \), if \( X_i \) is malleable

3) If the resulting grammar contains \( S \to \varepsilon \), introduce a new start symbol \( S' \) and add the productions \( S' \to S \varepsilon \)

4) Remove all \( E \)-productions, except possibly the one for \( S' \).

Example: by applying steps (1) and (2) to \( G_3 \), we get

\[
\begin{align*}
S & \to ABC | AB | AC | A \\
   & \quad | BCB | BC | BB | CB | B | C | \varepsilon \\
A & \to aB | \varepsilon \\
B & \to CC | C | \varepsilon | b \\
C & \to S | \varepsilon 
\end{align*}
\]

Since we have \( S \to \varepsilon \), we add

\( S' \to S \varepsilon \)

and remove the remaining \( E \)-productions.

Eliminate unit productions

Unit-production: \( A \to B \) slows down parsing

Algorithm to eliminate unit-productions

1) Remove \( E \)-productions

2) For all \( A, B \in V_M \)
   - if \( A \Rightarrow^* B \) and \( B \to \alpha \) is not unit
     then add \( A \to \alpha \)

3) Eliminate all unit-productions
How do we determine whether \( A \Rightarrow^* B \) holds?

Since we have no \( E \)-productions, we have that

\[
A \Rightarrow^* B \quad \text{if and only if} \quad A \Rightarrow B_1 \Rightarrow B_2 \Rightarrow \cdots \Rightarrow B_{k-1} \Rightarrow B_k \Rightarrow B
\]

where all \( B_i \)'s are pairwise distinct. Hence \( k \leq |V_M| \).

(If we had a sequence where two \( B_i \)'s are the same, we could eliminate all steps in between, and get a new sequence where all \( B_i \)'s are pairwise distinct.)

Each single derivation step \( B_i \Rightarrow B_{i+1} \) must correspond to a unit production \( B_i \Rightarrow B_{i+1} \) of \( G \).

Hence, we can detect whether \( A \Rightarrow^* B \) by checking whether \( B \) is reachable from \( A \) in the graph of the unit productions:

- nodes: non-terminals
- edges: one edge \((A) \Rightarrow (B)\) for each unit prod. \( A \Rightarrow B \)

**Example:** \( G_1 \):

\[
\begin{align*}
S & \Rightarrow A \mid B \\
A & \Rightarrow S \alpha \mid \epsilon \\
B & \Rightarrow S \mid \beta
\end{align*}
\]

Graph of unit productions

Reachability:

\[
\begin{align*}
S & \Rightarrow^* A \\
B & \Rightarrow^* S \\
B & \Rightarrow^* A
\end{align*}
\]

we get:

\[
\begin{align*}
S \Rightarrow A \mid B \mid S \alpha \mid \epsilon \mid S \mid \beta \\
A \Rightarrow S \alpha \mid \epsilon \\
B \Rightarrow S \mid \beta \mid A \mid B \mid S \alpha \mid \epsilon
\end{align*}
\]

Removing unit productions, we get:

\[
\begin{align*}
S & \Rightarrow S \alpha \mid \epsilon \mid \beta \\
A & \Rightarrow S \alpha \mid \epsilon \\
B & \Rightarrow S \alpha \mid \epsilon \mid \beta
\end{align*}
\]

Note: \( A \) and \( B \) have become unreachable.
We have seen: removal of: useless symbol, E-prod, unit-prod.
Does the order of the steps matter?

Observation:
- Removing useless symbols does not add productions at all (and therefore not E-prod or unit-prod).
- Removing E-prod: could add unit-prod.
- Removing unit-prod: needs removing E-prod first.
  - Could create useless symbols.
  - Cannot create E-prod.

⇒ The right order for removal is:
1) E-productions
2) unit-productions
3) useless symbols: first non-generating then unreachable.

Chomsky Normal Form

Definition: A CFG $G$ is in Chomsky Normal (CNF) if all its productions are of the form:
- $A \rightarrow a$
- $A \rightarrow BC$
  with $a \in V_T$
  $A, B, C \in V_N$

Given a CFG $G_1$, we can always construct a CFG $G_2$ that is in CNF and such that $L(G_2) = L(G_1) \setminus \{E\}$.

Note: since a CFG in CNF cannot generate $E$, if $G$ generates $E$, then we cannot have that $L(G_2) = L(G)$. However, apart from $E$, the two languages are equal.
Starting from $G_1$, we construct $G_c$ in several steps:

1) Eliminate $E$-productions (without introducing the new start symbol $S'$ with $S' \rightarrow S_1E$)
   
2) Eliminate unit-productions
   
   \[ \Rightarrow \text{all productions are of the form} \]
   \[ A \rightarrow a \]
   \[ A \rightarrow X_1 \ldots X_k \quad \text{with} \ k \geq 2 \]
   \[ \text{with} \ A \in V_N , \ a \in V_T , \ X_1 , \ldots , X_k \in V \]

3) Remove non-generating symbols

4) Remove unreachable symbols

5) Remove "mixed bodies" for each $a \in V_T$, add a new nonterminal $Na$ and production $Na \rightarrow a$
   
   in each production $A \rightarrow X_1 \ldots X_k$, replace $a$ with $Na$

   \[ \Rightarrow \text{all productions are of the form} \]
   \[ A \rightarrow a \]
   \[ A \rightarrow A_a \ldots A_k \quad (k \geq 2) \]
   \[ \text{with} \ a \in V_T , \ A_1 , A_2 , \ldots , A_k \in V_N \]

6) "Factor" long productions

   for each $A \rightarrow A_1 \ldots A_k$ with $k \geq 3$
   
   - add new nonterminals $B_1 , \ldots , B_{k-2}$
   
   - replace $A \rightarrow A_1 \ldots A_k$
     
     with $A \rightarrow A_1 B_1$
     
     $B_1 \rightarrow A_2 B_2$
     
     $B_{k-2} \rightarrow A_{k-1} A_k$

The grammar we get is in CNF by construction.

It is easy to show that the language is preserved, except possibly for the empty string $\varepsilon$, which cannot be generated by a grammar in CNF.
Example: $G$, \[
\{ 
S \rightarrow ABB \mid \varepsilon b \\
A \rightarrow Be \mid b e \\
B \rightarrow a ABb \mid b
\}
\]

Steps 1-4: nothing to do

Step 5:
\[
\begin{align*}
N_e & \rightarrow e \\
N_0 & \rightarrow b \\
S & \rightarrow ABB \mid N_e N_0 \\
A & \rightarrow B N_e \mid N_0 N_e \\
B & \rightarrow N_e A N_0 B \mid b
\end{align*}
\]

Step 6:
\[
\begin{align*}
N_e & \rightarrow e \\
N_0 & \rightarrow b \\
S & \rightarrow A B_1 \mid N_e N_0 \\
B_0 & \rightarrow B B \\
A & \rightarrow B N_e \mid N_0 N_e \\
B & \rightarrow N_0 C_1 \mid b \\
C_0 & \rightarrow A C_2 \\
C_2 & \rightarrow N_0 B
\end{align*}
\]

Note: If the original grammar generates $E$, and we want that the grammar in CNF also generates $E$, we can execute step 1 by introducing the new start symbol $S'$ and the productions: $S' \rightarrow S \mid E$

Step 2 will then replace the unit production $S' \rightarrow S$ by other productions, but none of the transformations 2-5 will introduce a production where $S'$ is on the right side.

Therefore, in the end, we will have a grammar in CNF, apart from the production $S' \rightarrow E$. 