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 1950, various formal methods based on the notion of rewriting or derivation have been proposed by Axel Néven, 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 $(V_N \cup V_T)^* \times (V_N \cup V_T)^*$

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. (AILATIDITALIA, 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. \( S \rightarrow \varepsilon \)  
2. \( S \rightarrow 0 \)  
3. \( S \rightarrow 1 \)  
4. \( S \rightarrow 0SO \)  
5. \( S \rightarrow 1S1 \)

Basis: 0, 1 are palindromes

Induction: if \( S \) is a palindrome, so are \( 0SO \) and \( 1S1 \)

Example of derivation:

- \( 0110 : S \xrightarrow{\varepsilon} 0SO \xrightarrow{5} 01S10 \xrightarrow{4} 0110 \)
- \( 1011 : S \xrightarrow{\varepsilon} 1S1 \xrightarrow{5} 11S11 \xrightarrow{2} 1011 \)

Example: Exercise 5.4.1

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 \( \rightarrow \) VerbPhrase

NounPhrase \( \rightarrow \) Adjective \( \rightarrow \) NounPhrase

Noun \( \rightarrow \) cat

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 \( V = V_N \cup V_T \)

A production of the form \( \alpha \rightarrow \varepsilon \), with \( \alpha \in V^* \cup V_N \cup V^* \)
   is called \( \varepsilon \)-production.

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

We have already seen that this language is not regular.

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

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

Induction: \( \text{If } A \varepsilon w \text{ in } L_{eq} \text{ then } \text{one more } 1 \text{ than } 0 \)
   \( \text{and } w \varepsilon A \text{ and } w \varepsilon B \text{ in } L_{eq} \text{ then } 0 \text{ or } 1 \)

Characterize also languages for \( w_A \) and \( w_B \) inductively

Grammar \( G_{eq} = (\{ S, A, B \}, \{ 0, 1 \}, \{ 0, 1 \}, S, P) \) with \( P \)

\[ S \rightarrow \varepsilon \mid OA \mid 1B \]
\[ A \rightarrow 1S \mid 0AA \]
\[ B \rightarrow 0S \mid 1BB \]

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 $(V^* \circ V_\text{m} \circ V^*) \subseteq V^*$ defined as follows:

$(\gamma, \psi)$ is in the relation if there exist

$\alpha \in V^* \circ V_\text{m} \circ V^*$, $\beta, \gamma, \delta \in V^*$

such that $\gamma = \rho \alpha \delta$, $\psi = \rho \beta \delta$ and $\alpha \rightarrow \beta \in \mathcal{P}$.

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

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

a) $\gamma = \psi$, or

b) there are $\gamma_1, \ldots, \gamma_m \in V^*$ such that

$\gamma_1 = \gamma$, $\gamma_m = \psi$, and $\gamma_i \Rightarrow \gamma_{i+1}$, $\forall i, 1 \leq i < m$.

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

$L(G) = \{w \in V_\tau^* \mid S \Rightarrow^* w\}$

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

Terminology:

- sentence: any word $w \in V_\tau^*$ s.t. $S \Rightarrow^* w$, i.e. $w \in L(G)$
- sentential form: any $\alpha \in V^* = (V_\tau \cup V_\text{m})^*$ s.t. $S \Rightarrow \alpha$

Notation: terminals: $a$, $b$, $c$, ...

nonterminals: $A$, $B$, $C$, ...

strings of terminals: $a$, $a^2$, $w$, $x$, $y$, $z$

symbols of $V = V_\text{m} \cup V_\tau$: $X$, $Y$, $Z$

sentential forms: $\alpha$, $\beta$, $\gamma$
Example: Productions for G_{eq}:

\[ S \rightarrow E | 10A | 1B \]
\[ A \rightarrow 1S | 10AA \]
\[ B \rightarrow 0S | 1BB \]

Derivation:

1) \[ 001SA \Rightarrow 001SA \] (using \( A \rightarrow 1S \))
2) \[ 001SA \Rightarrow 001AS \] (using \( S \rightarrow E \))
3) \[ 001SA \Rightarrow 001AS \] (using (1) and (2))
4) \[ S \Rightarrow 001110 \]

Example: Grammar for \( L_{eq} = \{ a^n b^m | m \geq 0 \} \)

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

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) \( CC \rightarrow cC \)

Note: we cannot simply have \( B \rightarrow b \), \( C \rightarrow c \) because this would generate many words not in \( L_{eq} \)

Example of derivation of \( aaaa bb bb cccc \):

\[ S \quad 1 \Rightarrow aSBC \quad 2 \Rightarrow aaaSBCBC \quad 2 \Rightarrow aaaa BCBBCBC \]
\[ 3 \Rightarrow aaaa BCBBBC \quad 3 \Rightarrow aaaa BBBCBC \\
3 \Rightarrow aaaa BBBCCC \quad 4 \Rightarrow aaaa BBBCCC \\
5 \Rightarrow aaaa bbBCCC \quad 5 \Rightarrow aaaa bbbbCC \\
6 \Rightarrow aaaa bb bb CCC \quad 7 \Rightarrow aaaa bb bb ccc \\
7 \Rightarrow aaaa bb bb ccc \]
Note: not each sequence of direct derivations leads to a sentence in $S(G_{3n})$

E.g. with the previous grammar we could generate

$S \Rightarrow aSBC \Rightarrow aaSBCBC \Rightarrow aaoBCBCBC$

$\Rightarrow aaaoBCBCBC \Rightarrow aaoBCBCCC$

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 $\alpha \rightarrow \beta$,

with $\alpha \in V^* \cup V_M \cup V^*, \quad \beta \in V^*$

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 $\alpha \rightarrow \beta$, 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: $G_{3m}$ is context sensitive. Obviously, it is also
of type 0.

Definition: grammar of type 2, or context-free
Productions have the form $A \rightarrow \beta$, 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: $L_{2m} = \{a^m b^m | m \geq 1\}$ is of type 1, since the
following grammar $G'_{2m}$ generates $L_{2m}$

$S \rightarrow aB \mid SAB$
$BA \rightarrow AB$
$aA \rightarrow aA$
$AB \rightarrow aB$
$bB \rightarrow bB$

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

$S \rightarrow aSB \mid aB$
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 \( \gamma_1 A \gamma_2 \rightarrow \gamma_3 \beta \gamma_2 \)
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 \( \gamma_1 A \gamma_2 \rightarrow \gamma_3 \beta \gamma_2 \), we have \( \beta \in V^+ \) and hence \( |\gamma_1 A \gamma_2| \leq |\gamma_3 \beta \gamma_2| \))

"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 \( \alpha \in V_T \), add a new nonterminal \( N_\alpha \)
2) replace in each production of \( G_1 \), each \( \alpha \in V_T \) by \( N_\alpha \)

Now all productions have the form

\[ A_1 A_2 \ldots A_m \rightarrow B_1 B_2 \ldots B_m \]
with \( m \leq m \)
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 non-terminal \( N \), and replace the production by the following ones:

\[
\begin{align*}
A_1 A_2 \cdots A_m & \rightarrow NA_2 \cdots A_m \\
NA_2 \cdots A_m & \rightarrow NB_2 A_3 \cdots A_m \\
NB_2 A_3 \cdots A_m & \rightarrow NB_2 B_3 A_4 \cdots A_m \\
\vdots & \\
NB_2 \cdots B_{m-1} A_m & \rightarrow NB_2 \cdots B_{m-1} B_m \cdots B_n \\
NB_2 \cdots B_m & \rightarrow B_1 B_2 \cdots B_n
\end{align*}
\]

(\textit{note that, due to the presence of } \( N \), \textit{these productions will not "interact" with other ones})

Observe that all such productions are of the form

\( \gamma_1 \gamma_2 \rightarrow \gamma_1 \beta \gamma_2 \) with \( \gamma_1, \gamma_2 \in V^* \), \( \alpha \in V_N \), \( \beta \in V^+ \)

4) For each \( a \in V_1 \), add the production

\( N_a \rightarrow a \) (where \( N_a \) is the new non-terminal associated to \( a \))

It is not difficult to see that \( S(G_1) = S(G_2) \)

(\textit{the proof is by induction on the length of the derivation of a string } \( w \in S(G_1) \) (resp., \( S(G_2) \))

\textit{END OPTIONAL}
Definition: grammar of type 3, or regular, or right-linear.

Productions have the form \( A \to \delta \) with \( A \in V_N \)

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

(i.e., \( A \to \alpha \beta \) or \( A \to \alpha \), with \( \alpha, \beta \in V_N \), \( \alpha \in V_T \))

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

Example: \( \{ a^m b^m | m \geq 0 \} \) is of type 3, since it is generated by the grammar:

\[
\begin{align*}
S & \to aS \\
S & \to b
\end{align*}
\]

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

Optional

Exercise: E5.3:

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

Hint: Given \( G = (V_N, V_T, P, S) \), construct an NFA

\( A_\epsilon = (V_N, V_T, \{F\}, S, S, \{F\}) \) with

\( B \in \delta(A, \epsilon) \) iff \( A \to \alpha \beta \) and

\( F \in \delta(A, \epsilon) \) iff \( A \to \epsilon \)

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

Conversely, given an NFA \( A \), construct a grammar \( G_A \) by having unique nonterminals 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 $S_{new}$, make it the new start symbol, add a production $S_{new} \rightarrow S$, and allow for $S_{new} \rightarrow E$.

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 $N_e$ that is eliminated through an $E$-production $N_e \rightarrow E$, and use $N_e$ to make the right-hand side of productions as long as the left-hand side.
In a CFG, the productions have the form a → b, with a ∈ V_1, b ∈ V*. (note: we allow for E-productions)

Example: CFG for arithmetic expressions over variable n

G = ( {E, T, F}, {i, +, *, (, )}, P, E ), where P is

E → T | E + T  
T → F | T * F  
F → i | (E)  

This grammar generates, e.g., i + i * i

E → E + T → T + i → E + i → i + T  
→ i + T * F → i + i * i

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

To a tree whose nodes are labeled by elements of V_1∪{E} satisfying:

1) each internal node is labeled by a non-terminal
2) each leaf is labeled by a non-terminal, a terminal, or E. If it is labeled by E, then it is the only child of its parent
3) If an internal node is labeled A, and its children from left to right are labeled X_1, X_2, ..., X_k, then there is a production A → X_1X_2...X_k in P.

Example: parse tree for i + i * i
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:**

![Diagram of a parse tree]

**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.

  \[ E \Rightarrow E + T \Rightarrow E + T \Rightarrow E + T \Rightarrow \cdots \]

- **Rightmost derivation:** defined similarly.

  \[ E \Rightarrow E + T \Rightarrow E + T \times F \cdots \]
Theorem: The following are all equivalent statements for a CFG \( G = (V, T, P, S) \) and a string \( w \in T^* 

1) \( w \in \mathcal{L}(G) \) (or \( S \Rightarrow^* w \))
2) \( S \xrightarrow{\text{lm}} w \)
3) \( S \xrightarrow{\text{m}} w \)
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 \( \text{lm} \)-derivation as a canonical way to derive any \( w \in \mathcal{L}(G) \); i.e., as a canonical way to interpret a parse tree for \( w \).

Ambiguous grammars:

- \( w \in \mathcal{L}(G) \) could have two distinct parse trees, and hence two distinct \( \text{lm} \)-derivations.

Example: another grammar for arithmetic expressions

\[
E \rightarrow i \mid (E) \mid E + E \mid E \cdot E
\]

\( w = i + i \cdot i \)

\[
\text{These parse trees correspond to two different \( \text{lm} \)-derivations, and also to two ways of interpreting \( w \).}
\]
**Definition:** A CFG $G$ 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 CF language is (inherently) ambiguous if all its grammars are ambiguous.

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

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

Consider strings of the form $a^k b^k c^k d^k$.
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 useless if

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

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

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 $S \Rightarrow^* \alpha X \beta$, for $\alpha, \beta \in V^*$

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

1) eliminating non-generating symbols and all their production
2) unreachable

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

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

Let us consider what happens if we do first (2) and then (1)

We eliminate unreachable symbols: all are reachable

we eliminate B and \( S \rightarrow AB \)

\[ \Rightarrow \text{we obtain: } S \rightarrow b \]

\[ A \rightarrow a \]

But, if we do it in reverse order:

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

\[ \Rightarrow \text{we obtain: } S \rightarrow b \]

1) Eliminating non-generating symbols:

Recursive algorithm to construct the set \( G \) of generating symbols:

basis: mark all terminals as generating

recursive step: for each production \( A \rightarrow X_1 \ldots X_h \)

if all of \( X_1 \ldots X_h \) are marked as generating

then mark \( A \) as generating

terminate: when no new generating symbol is found
Example: \( G_1: \)

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

\[
\begin{aligned}
\{e, b, d\} \\
\{e, b, c, B, D\} \\
\{b, c, A\} \\
\{e, b, c, S\} & \Rightarrow C \text{ is non-generating}
\end{aligned}
\]

\[
\Rightarrow \text{Remove } C \text{ and all productions involving } C
\]

2) Eliminating unreachable symbols

Recursive algorithm to construct the set of reachable symbols:

- Basis: mark \( S \) as reachable

- Recursive step: for each production \( A \rightarrow K_1 \ldots K_n \)
  - if \( A \) is marked as reachable
  - then mark \( K_1, \ldots, K_n \) as reachable

Terminate when no new reachable symbol is found

Example: \( G_2: \)

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

\[
\begin{aligned}
\{S\} \\
\{S, A, B\} \\
\{S, A, B, e, b\}\Rightarrow D, d \text{ are unreachable}
\end{aligned}
\]

\[
\Rightarrow \text{Remove } D, d \text{ and all productions involving them}
\]
2) Eliminate $E$-productions

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

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

We first need to find all nullable symbols:

Recursive algorithm to construct the set of nullable symbols:

basis: if $P$ contains $A \rightarrow E$, then mark $A$ as nullable

inductive: for each production $A \rightarrow X_1 \ldots X_n$

step: if all of $X_1, \ldots, X_n$ are marked as nullable
then mark $A$ as nullable

terminate when no new nullable symbol is found

Example: $G_1$: \[
\begin{align*}
S &\rightarrow ABC \mid BCB \\
A &\rightarrow cB \mid e \\
B &\rightarrow CC \mid b \\
C &\rightarrow S \mid E
\end{align*}
\]

\{
\{C\} \\
\{C, B\} \\
\{C, B, S\}\}

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

Example: in $G_1$, since $B$ and $C$ are nullable, we can derive

\[
\begin{align*}
S &\Rightarrow^* BCB, \quad S \Rightarrow^* CB, \quad S \Rightarrow^* BC, \quad S \Rightarrow^* BB, \\
S &\Rightarrow^* C, \quad S \Rightarrow^* B, \quad S \Rightarrow^* E
\end{align*}
\]

Hence, if we eliminate $C \rightarrow E$, we have to add direct productions for the above derivations.
Algorithm to eliminate $E$-productions

1) Identify all nullable symbols

2) Replace each production $A \rightarrow X_1 \ldots X_k$
   by the set of all productions of the form $A \rightarrow \alpha_1 \ldots \alpha_k$
   where
   $\alpha_i = X_i$, if $X_i$ is not nullable
   $\alpha_i = X_i$ or $E$, if $X_i$ is nullable

3) If the resulting grammar contains $S \rightarrow E$, introduce a new start symbol $S'$ and add the productions $S' \rightarrow S | E$

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

Example: by applying steps (1) and (2) to $G_1$, we get

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

Since we have $S \rightarrow E$, we add $S' \rightarrow S | E$

and remove the remaining $E$-productions.

3) Eliminate unit productions

Unit production: $A \rightarrow B$ slows down parsing

Algorithm to eliminate unit-productions

1) Remove $E$-productions

2) For all $A, B \in V_N$
   \[ \text{if } A \Rightarrow^* B \text{ and } B \rightarrow \alpha \text{ is not unit} \]
   \[ \text{then add } A \rightarrow \alpha \]

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

Since we have our $E$-productions, we have that

$A \Rightarrow^* B$ if and only if

$A \Rightarrow B_1 \Rightarrow B_2 \Rightarrow \cdots \Rightarrow B_{k-1} \Rightarrow B_k \Rightarrow B$

where all $B_i$'s are pairwise distinct, hence $k \leq |V_H|$. 

(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 $\text{A} \rightarrow \text{B}$ for each unit prod. $A \rightarrow B$

Example: $G_1:\{S \rightarrow A \mid B$
$A \rightarrow Sa \mid a$
$B \rightarrow S \mid b$

Graph of unit productions:

Reachability: $S \Rightarrow^* A$  $S \Rightarrow^* B$
$B \Rightarrow^* S$  $B \Rightarrow^* A$

We get:

$S \rightarrow A \mid B \mid Sa \mid a \mid S \mid b$
$A \rightarrow Sa \mid a$
$B \rightarrow S \mid b \mid A \mid B \mid Sa \mid a$

Removing unit productions, we get:

Note: A and B have become unreachable.

\{ $S \rightarrow Sa \mid a$
$A \rightarrow Sa \mid a$
$B \rightarrow Sa \mid a$
}
We have seen removal of: useless right, $E$-prod, unit-prod.

Does the order of the steps matter?

Observation:
- Removing useless: 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

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

Note: since a CFG in CNF cannot generate $\epsilon$, if $G$ generates $\epsilon$, then we cannot have that $L(G_c) = L(G)$. However, apart from $\epsilon$, 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 $S' \rightarrow S'E$)
2) Eliminate unit-productions

   => All productions are of the form

   $A \rightarrow a$
   $A \rightarrow \lambda_1 \ldots \lambda_k$ with $k \geq 2$
   with $A \in V_N$, $e \in V_T$, $\lambda_1, \ldots, \lambda_k \in V$

3) Remove non-generating symbols
4) Remove unreachable symbols
5) Remove "mixed bodies"

   for each $e \in V_T$, add a new nonterminal $V_e$ and production $V_e \rightarrow e$

   in each production $A \rightarrow \lambda_1 \ldots \lambda_k$, replace $e$ with $V_e$

   => All productions are of the form

   $A \rightarrow e$
   $A \rightarrow A_1 \ldots A_k$ (k \geq 2)
   with $e \in V_T$, $A_1, A_2, \ldots, A_k \in V_N$

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: \( G, \) \[
S \rightarrow ABB \mid \varepsilon b \\
A \rightarrow B \varepsilon \mid b \varepsilon \\
B \rightarrow \varepsilon A B B
\]

Steps 1-4: nothing to do

Step 5: \[
\begin{align*}
V_e & \rightarrow \varepsilon \\
V_b & \rightarrow b \\
S & \rightarrow ABB \mid V_e V_b \\
A & \rightarrow B V_e \mid V_b V_e \\
B & \rightarrow V_e A V_b B
\end{align*}
\]

Step 6: \[
\begin{align*}
V_e & \rightarrow \varepsilon \\
V_b & \rightarrow b \\
S & \rightarrow A B, \mid V_e V_b \\
B_n & \rightarrow B B \\
A & \rightarrow B V_e \mid V_b V_e \\
B & \rightarrow V_e C_1 \\
C_1 & \rightarrow A C_2 \\
C_2 & \rightarrow V_b B
\end{align*}
\]

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

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 CHF, except for the production \( S' \rightarrow \varepsilon \).