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 Noam Chomsky, Emil Post, and others.

In the mid 1950s, the linguist Noam Chomsky introduced the notion of formal grammars 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 \( a \in V_N \cap \Sigma = \emptyset \), 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

\[
(V_N \cup V_T)^* \times V_N \times (V_N \cup V_T)^* \times (V_N \cup V_T)^*
\]

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

Note: \( U \) ... sequence of terminals and nonterminals with at least one non-terminal.

\( \beta \) ... 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 \( U \rightarrow \beta \) consists in replacing an occurrence of \( U \) with \( \beta \).
Example: Palindromes:

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

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

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

1) \( S \rightarrow \epsilon \)  
2) \( S \rightarrow 0 \)  
3) \( S \rightarrow 1 \)  
4) \( S \rightarrow 0S0 \)  
5) \( S \rightarrow 1S1 \)

Example of derivation:

\[ 0110 : S \rightarrow 0S0 \rightarrow 01S10 \rightarrow 0110 \]
\[ 11011 : S \rightarrow 1S1 \rightarrow 11S11 \rightarrow 11011 \]

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 VerbalPhrase
NounPhrase \( \rightarrow \) Adjective NounPhrase
NounPhrase \( \rightarrow \) Noun
Noun \( \rightarrow \) can
Noun \( \rightarrow \) train
Adjective \( \rightarrow \) big
Adjective \( \rightarrow \) broken
Notation:
1) To denote the set of productions
   \[ \alpha \rightarrow \beta_1, \quad \alpha \rightarrow \beta_2, \quad \ldots, \quad \alpha \rightarrow \beta_n \]
   we use
   \[ \alpha \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n \]
2) We use \( V = V_N \cup V_T \)

A production of the form \( \alpha \rightarrow \varepsilon \), with \( \varepsilon \in V^* \cdot V_N \cdot V^* \)
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. \( L(G_{eq}) = L_{eq} \): use induction

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

induction: Own \( A \in L_{eq} \) if \( w_A \) has one more 1 than 0
          Own \( B \in L_{eq} \) if \( w_B \) has one more 0 than 1

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

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

\[ S \rightarrow \varepsilon \mid OA \mid AB \]  \( (A \text{ generates strings with one more 1 than 0}) \)
\[ A \rightarrow 1S \mid 0AA \]  \( (B \text{ generates strings with one more 0 than 1}) \)
\[ B \rightarrow OS \mid 1BB \]

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

$\gamma \Rightarrow \delta$ is in the relation if there are

$\alpha \in V^* \circ V_M \circ V^*$, \hspace{1em} \beta, \gamma, \delta \in V^*$

such that $\gamma = \gamma \alpha \delta$, $\gamma = \gamma \beta \delta$ and $\alpha \Rightarrow \beta \in P$.

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

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

a) $\gamma = \delta$, or

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

$\gamma_1 = \gamma$, $\gamma_n = \delta$, and $\gamma_i \Rightarrow^* \gamma_{i+1}$, $\forall i$, $1 \leq i \leq n$.

Definition: Given a grammar $G$, the language generated by $G$ in
$L(G) = \{ w \in V_T^* \mid S \Rightarrow^* w \}$

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

Terminology:

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

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

Notation: terminals: $A, B, C, \ldots$

nonterminals: $A, B, C, \ldots$

strings of terminals: $w, w_1, \ldots, w, \ k, y, z, \ldots$

symbols of $V = V_M \cup V_T$: $A, B, Y, Z, \ldots$

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

\[ S \rightarrow e | OA | 1B \]
\[ A \rightarrow 1S | 0AA \]
\[ B \rightarrow 0S | 1BB \]

Derivation:
1) $001SA \Rightarrow 001SA$ (using $A \rightarrow 1S$)
2) $001SA \Rightarrow 0011S$ (using $S \rightarrow e$)
3) $001SA \Rightarrow 0011S$ (using (1) and (2))
4) $S \Rightarrow 001110$

Example: Grammar for $L_{eq} = \{ a^n b^m c^n \mid m \geq 0 \}$

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

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) $CC \rightarrow cc$

Example of derivation of $aaabbbbccc$:

\[ S \Rightarrow aSBC \Rightarrow aaSBCBC \Rightarrow aaeBCBCBC \]
\[ \Rightarrow aeeBCBCC \]
\[ \Rightarrow aeeBBBCC \]
\[ \Rightarrow aeeBBBCCC \]
\[ \Rightarrow aeeBBBCCC \]
\[ \Rightarrow aeeBBBCCC \]
\[ \Rightarrow aeeBBBCCC \]
\[ \Rightarrow aeeBBBCCC \]
\[ \Rightarrow aeeBBBCCC \]
\[ \Rightarrow aeeBBBCCC \]
\[ \Rightarrow aeeBBBCCC \]
Note: not each sequence of direc derivations leads to a sentence in \( L(G_{3n}) \)

E.g. with the previous grammar we could generate

\[
S \Rightarrow aSBC \Rightarrow aeaSBCBC \Rightarrow aeaBCBCBCC \Rightarrow aeaBBBCBCC \\
\Rightarrow aeaBBCC
\]

we cannot apply any other production

Also, the application of productions could go on forever
(E.g. rule 4 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: grammars of type 0.

Productions have the most general form \( \alpha \rightarrow \beta \),
with \( \alpha \in V \cup V^* \) \( \rightarrow \) \( \beta \in V^* \)

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 \( A \rightarrow \beta \), with
\[
\alpha \in \mathcal{V}^*, \quad \gamma \in \mathcal{V}^* \quad \beta \in \mathcal{V}^+ \quad |\alpha| \leq |\beta|
\]

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 \( A \rightarrow \beta \), with \( A \in \mathcal{V}_N \), \( \beta \in \mathcal{V}^+ \).

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 \( L_{2n} \):

\[
S \rightarrow aB \mid SAB \\
B \rightarrow AB \\
aA \rightarrow aA \\
aB \rightarrow aB \\
bB \rightarrow bb
\]

\( L_{2n} \) 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 \( \phi_1 A \phi_2 \to \phi_1 \beta \phi_2 \)

with \( \phi_1, \phi_2 \in \Sigma^*, \ A \in V_N, \ \beta \in \Sigma^+ \)

Intuitively, \( A \) is replaced by \( \beta \) only if it appears "in the context" of \( \phi_1 \) and \( \phi_2 \).

**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 \( \phi_1 A \phi_2 \to \phi_1 \beta \phi_2 \) we have \( \beta \in \Sigma^+ \) and hence \( |\phi_1 A \phi_2| \leq |\phi_1 \beta \phi_2| \))

"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 \( \epsilon \in \Sigma^* \), add a new nonterminal \( N_{\epsilon} \)
2) replace in each production of \( G_1 \), each \( \epsilon \in \Sigma^* \) by \( N_{\epsilon} \)

Now all productions have the form:

\[ A_1 A_2 \cdots A_m \to B_1 B_2 \cdots B_n \]

with \( m \leq n \)

and all \( A_i, B_j \in V_N \)
3) For each such production $A_1 \ldots A_m \rightarrow B_1 \ldots B_n$, introduce a new non-terminal $N$, and replace the production by the following ones:

$$A_1 A_2 \ldots A_m \rightarrow NA_2 \ldots A_m$$
$$NA_2 \ldots A_m \rightarrow NB_2 A_3 \ldots A_m$$
$$NB_2 A_3 \ldots A_m \rightarrow NB_2 B_3 A_4 \ldots A_m$$

$$\vdots$$

$$NB_2 \ldots B_{m-1} A_m \rightarrow NB_2 \ldots B_{m-1} B_m \ldots B_n$$
$$NB_2 \ldots B_m \rightarrow B_1 B_2 \ldots B_n$$

Observe that all such productions are of the form $y_1 A y_2 \rightarrow y_1 B y_2$ with $y_1, y_2 \in V^*$, $A \in V_N$, $B \in V^+$.

4) For each $e \in V_7$, add the production

$$N_e \rightarrow e$$

(where $N_e$ is the new, non-terminal associated to $e$)

It is not difficult to see that $L(G_1) = L(G_e)$

(Proof is by induction on the length of the derivation of a string $w \in L(G_1)$ (resp., $L(G_e)$))

END OPTIONAL
Definition: grammar of type 3, or regular, or right linear

Productions have the form $A \rightarrow \sigma$ with $A \in V_N$

$\delta \in V_T \cup (V_T \times V_N)$

(i.e., $A \rightarrow \alpha B$ or $A \rightarrow \alpha$, with $A, B \in V_N$, $\alpha \in V_T$)

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

Example: $\{a^n b^n | n \geq 0\}$ 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.

Exercise: E5.3: Show that grammars of type 3 generate the class of regular languages that do not contain $\varepsilon$.

Hint: given $G = (V_N, V_T, P, S)$, construct an NFA $A_\varepsilon = (V_N \cup \{F\}, V_T, \delta, S, \{F\})$ with

$\delta(C, \alpha) \text{ iff } A \rightarrow \alpha B$ and

$\delta(A, \varepsilon) \text{ iff } A \rightarrow B$

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

Conversely, given an NFA $A$, construct a grammar $G_A$ by having again non-terminals correspond to states of $A$. 

End Optional
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

  $$S \rightarrow E \quad (E\text{-production})$$

- if the start symbol $S$ appears on the right-hand side of productions, we introduce a new non-terminal $S_{\text{new}}$, make it the new start symbol, add a production

  $$S_{\text{new}} \rightarrow S$$

  and allow for $S_{\text{new}} \rightarrow E$.

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.

Optional

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.

END OPTIONAL
In a CFG, the productions have the form \( A \rightarrow B \) with \( A \in V_N \), \( B \in V^* \) (note: we allow for \( E \)-productions).

Example: CFG for arithmetic expressions over variables \( i \)

\[
G_i = (\{ E, T, F \}, \{ i, +, *, (, ) \}, \{ P, E \}, \text{where } P \text{ in } P \text{ is }

E \rightarrow T \mid E + T \\
T \rightarrow F \mid T * F \\
F \rightarrow i \mid (E)
\]

This grammar generates, e.g., \( i + i \times i \)

\[
E \rightarrow E + T \rightarrow T + T \rightarrow E + T \rightarrow i + i \\
\rightarrow i + i \times F \rightarrow i + i \times E \rightarrow i + i \times i
\]

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 \( V = \{ E, F \} \) satisfying:

1) each interior 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 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 \( i + i \times i \)
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:

```
    E
   /|
  /  |
E   T
/   |
T   +
    |
T   T
    |
T   F
    |
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).

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 $\Rightarrow$ T + T $\Rightarrow$ E + T $\Rightarrow$ $\alpha + T$ $\Rightarrow$ ...

- Rightmost derivation: defined similarly: E $\Rightarrow$ E + T $\Rightarrow$ E + T + E
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) \quad \text{(or } S \Rightarrow^* w \text{)} \\
2) \ S \xrightarrow{l} w \\
3) \ S \xrightarrow{r} 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 \( l \text{-}r \)-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:

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

Example: another grammar for arithmetic expressions

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

\( w = i + i \times i \)

\[
\begin{array}{c}
E \\
| \\
E + E \\
| \\
E \times E
\end{array}
\]

\[
\begin{array}{c}
E \\
| \\
E + E \\
| \\
E \times E
\end{array}
\]

These parse trees correspond to two different \( l \text{-}r \)-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 string.

Sometimes grammar 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: $L = \{a^n b^n c^n d^n \mid n, m \geq 1\} \cup \{a^n b^n c^n d^n \mid n, m \geq 1\}$

$L$ 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 \Rightarrow 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

1) **Eliminate useless symbols.**

We say that \( X \in V \) is useful 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 productions
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
- ... non-generating ...

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 of generating symbols:

basis: mark all terminals as generating

recursive step: for each production \( A \rightarrow X_1 \ldots X_k \)
if all of \( X_1 \ldots X_k \) are marked as generating
then mark \( A \) as generating

terminate: when no new generating symbol is found
Example: \( G_1: \)
\[
\begin{align*}
S & \rightarrow AB \mid AC \mid CD \\
A & \rightarrow BB \\
B & \rightarrow AC \mid aB \\
C & \rightarrow Ca \mid CC \\
D & \rightarrow Bc \mid b \mid d
\end{align*}
\]
\[
\{e, b, d\} \\
\{e, b, d\} \\
\{e, b, d\} \\
\{e, b, d\} \\
\{e, b, d\} \Rightarrow C \text{ is non-generating}
\]
\[
\Rightarrow \text{Remove } C \text{ and all productions involving } C
\]

2) Eliminating unreachable symbols

Recursive algorithm to construct the set of reachable symbols:

- **Basis:** Mark \( S \) as reachable
- **Recursive step:** For each production \( A \rightarrow X_1 \cdots X_n \)
  - If \( A \) is marked as reachable
  - Then mark \( X_1, \ldots, X_n \) as reachable

Terminate when no new reachable symbol is found

Example: \( G_2: \)
\[
\begin{align*}
S & \rightarrow AB \\
A & \rightarrow BB \\
B & \rightarrow cB \\
D & \rightarrow b \mid d
\end{align*}
\]
\[
\{S\} \\
\{S, A, B\} \\
\{S, A, B, c, b\} \Rightarrow D, d \text{ are unreachable}
\]
\[
\Rightarrow \text{Remove } D, d \text{ and all productions involving them}
\]
2) Eliminate $E$-productions

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

**Definition:** $X \in V_K$ is *nulleable* if $X \Rightarrow^* E$

We first need to find all nulleable symbols:

**Recursive algorithm to construct the set of nulleable symbols:**

- **basis:** if $P$ contains $A \rightarrow E$, then mark $A$ as nulleable
- **inductive step:** for each production $A \rightarrow X_1 \ldots X_n$,
  - if all of $X_1, \ldots, X_n$ are marked as nulleable
    - then mark $A$ as nulleable
  terminate when no new nulleable symbol is found

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

- $\{C\}$
- $\{C, B\}$
- $\{C, B, S\}$

Eliminating the nulleable symbols allows us to compensate for the elimination of $E$-transitions.

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

- $S \Rightarrow^* BCB, S \Rightarrow^* CB, S \Rightarrow^* BC, S \Rightarrow^* BB$
- $S \Rightarrow^* C, S \Rightarrow^* B, S \Rightarrow^* E$

Since 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 $A \rightarrow \alpha_1 \ldots \alpha_k$ by the set of all productions of the form $A \rightarrow \alpha_1 \ldots \alpha_k$
   where $\alpha_i = \xi_i$, if $\xi_i$ is not nullable
   $\alpha_i = \xi_i \cdot E$, if $\xi_i$ is nullable

3) Remove all $E$-productions

Example: for $G_4$

$$
S \rightarrow ABC \mid AB \mid AC \mid A \\
BCB \mid BC \mid BB \mid CB \mid B \mid C \mid E
$$

$$
A \rightarrow aB \mid e
B \rightarrow CC \mid C \mid E \mid e
C \rightarrow S \mid E
$$

Finally, remove all $E$-productions

Note: the grammar no longer generates $E$. (This is unavoidable.)

3) Eliminate unit-productions

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

Algorithm to eliminate unit-productions

1) Remove $E$-productions

2) For all $A, B \in V_K$
   if $A =^* B$ and $B \rightarrow \alpha$ is not unit
   then add $A \rightarrow \alpha$

3) Eliminate all unit-productions

11/12/2004
How do we find $A \Rightarrow^* B$?

Since we have no $\varepsilon$-productions: $A \Rightarrow^* B$ only if

$$A \Rightarrow B_1 \Rightarrow B_2 \Rightarrow \ldots \Rightarrow B_k \Rightarrow B$$

where all $B_i$'s are distinct.

Hence, $k \leq |V_M|$, and we can use reachability in directed graphs.

Example: $G_1 : \begin{cases} S \rightarrow A \mid B \\ A \rightarrow S \mid a \\ B \rightarrow S \mid b \end{cases}$

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

We get: \begin{cases} S \rightarrow S \mid e \mid b \mid A \mid B \\ A \rightarrow S \mid a \\ B \rightarrow S \mid e \mid b \mid S \end{cases}

Removing unit-productions: \begin{cases} S \rightarrow S \mid e \mid b \\ A \rightarrow S \mid a \\ B \rightarrow S \mid e \mid b \end{cases}

Note: $A, B$ are now unreachable, and hence useless.
We have seen: removal of \( \rho \), \( \epsilon \)-prod, \( \text{mut} \)-prod

Does the order of the steps matter?

Observation:

- Removing \( \rho \): does not add productions at all (and therefore not \( \epsilon \)-prod or \( \text{mut} \)-prod)
- Removing \( \epsilon \)-prod: could add \( \text{mut} \)-prod
- Removing \( \text{mut} \)-prod: needs removing \( \epsilon \)-prod first
  - could create \( \rho \) symbols
  - cannot create \( \epsilon \)-prod

\[
\implies \text{The right order for removal is}
\]

1) \( \epsilon \)-productions
2) \( \text{mut} \)-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

\[
A \rightarrow a, \quad A \rightarrow BC
\]

with \( a \in V_T \)

\( A, B, C \in V_N \)

**Theorem**: Given a CFG \( G \) with \( \epsilon \in L(G) \), there is a CNF grammar \( G' \) with \( L(G') = L(G) \).
Proof. Constructive, in 3 steps

1) Eliminate $\varepsilon$-prod. and unit-prod.
   $\Rightarrow$ all productions are of the form
   
   $A \rightarrow \varepsilon$
   $A \rightarrow A_1 \ldots A_k \quad (k \geq 2)$
   
   with $A \in V_N$, $\alpha \in V_T$, $A_1, \ldots, A_k \in V$

2) Remove "mixed bodies"
   
   for each $\varepsilon \in V_T$, add a new nonterminal $V_\varepsilon$ and
   production $V_\varepsilon \rightarrow \alpha$
   
   in each production $A \rightarrow A_1 \ldots A_k$, replace $\varepsilon$ with $V_\varepsilon$

   $\Rightarrow$ all productions are of the form
   
   $A \rightarrow \varepsilon$
   $A \rightarrow A_1 \ldots A_k \quad (k \geq 2)$
   
   with $\varepsilon \in V_T$, $A, A_1, \ldots, A_k \in V_N$

3) "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 $A \rightarrow A_1 B_1$
     $B_1 \rightarrow A_2 B_2$
     $\vdots$
     $B_{k-2} \rightarrow A_{k-1} A_k$

   The grammar we get is in CNF by construction.

It is easy to show that the language is preserved
Example: \[ G, \{ \begin{align*} S & \rightarrow ABB | e \& b \\ A & \rightarrow Be | b \& a \\ B & \rightarrow aA \& B \end{align*} \]  

Step 1: nothing to do

Step 2: \[ \{ \begin{align*} V_e & \rightarrow e \\ V_b & \rightarrow b \\ S & \rightarrow ABB | V_eV_b \\ A & \rightarrow BV_e | V_bV_e \\ B & \rightarrow V_eAV_bV_b \end{align*} \]  

Step 3: \[ \begin{align*} V_e & \rightarrow e \\ V_b & \rightarrow b \\ \{ \begin{align*} S & \rightarrow ABV_e | V_eV_b \\ B_1 & \rightarrow BB \\ A & \rightarrow BV_e | V_bV_e \\ B & \rightarrow V_eC_1 \\ C_1 & \rightarrow AC_2 \\ C_2 & \rightarrow V_bV_b \end{align*} \]