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 useless if

\[ S \Rightarrow^{*} \alpha X \beta \Rightarrow^{*} w, \text{ with } w \in V_{T}^{*}, \alpha, \beta \in V^{*} \]

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: right order is important

Example: $\begin{cases} S \rightarrow AB | b \\
A \rightarrow e \end{cases}$

- we eliminate unreachable symbols: all are reachable
- non-generating

we eliminate $B$ and $S \rightarrow AB$

$\Rightarrow$ we obtain: $S \rightarrow b$
$A \rightarrow e$

But, if we do it in right order:

1) Eliminate non-generating symbols: $B$ and $S \rightarrow AB$
2) unreachable: $A$ and $A \rightarrow e$

$\Rightarrow$ we obtain: $S \rightarrow b$

1) Eliminating non-generating symbols:

Recursively:

- basis: mark all terminals as generating
- induction: for each production $A \rightarrow X_1 \ldots X_n$ if all of $X_1 \ldots X_n$ are marked as generating then mark $A$ as generating
- terminate: when no new generating symbol is found
Example: \( G_1 : \) 
\[
\begin{align*}
S &\rightarrow AB \mid AC \mid CD \\
A &\rightarrow BB \\
B &\rightarrow AC \mid ab \\
C &\rightarrow Ca \mid CC \\
D &\rightarrow Bc \mid b \mid d \\
\end{align*}
\]
\[
\{e, b, d\} \\
\{\_\_\_\_\_\_\_\_, b, D\} \\
\{\_\_\_\_\_\_\_, A\} \\
\{\_\_\_\_\_\_\_, S\} \Rightarrow C \text{ is non-generating}
\]
\( \Rightarrow \) Remove \( C \) and all productions involving \( C \).

2) Eliminating unreachable symbols

Recursively

Basis: mark \( S \) as reachable

Induction: for each production \( A \rightarrow X_0 \ldots X_n \)

if \( A \) is marked as reachable

then mark \( X_0, \ldots, X_n \) as reachable

terminate when no new reachable symbol is found

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

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

**Definition:** $X \in V_N$ is **millable** if $X \Rightarrow^{*} E$

We first need to find all millable symbols:

*Recursively:*

- **basis:** if $P$ contains $A \rightarrow E$, then mark $A$ as millable
- **induction:** for each production $A \rightarrow X_1 \cdots X_n$
  - if all of $X_1, \ldots, X_n$ are marked as millable
    - then mark $A$ as millable

**Example:** $G_1$:

\[
\begin{align*}
S & \rightarrow ABC \mid BCB \\
A & \rightarrow aB \mid e \\
B & \rightarrow CC \mid b \\
C & \rightarrow S \mid E
\end{align*}
\]

\[
\begin{align*}
\{ C \} \\
\{ C, B \} \\
\{ C, B, S \}
\end{align*}
\]

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

**Example:** in $G_1$, since $B$ and $C$ are millable, 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 \( A \rightarrow X_{i_1} \ldots X_{i_k} \)
   where \( X_{i_i} = X_i \), if \( X_i \) is not nullable
   \( X_{i_i} = X_i \) or \( E \), if \( X_i \) is nullable

3) Remove all E-productions

Example: for \( G_1 \)

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

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 \( A, B \in V_N \)
   if \( A \rightarrow^* B \) and \( B \rightarrow \alpha \) is not unit
   then add \( A \rightarrow \alpha \)

3) Eliminate all unit-productions

1/12/2004
How do we find $A \Rightarrow^* B$?

Since we have no $E$-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, $k \leq |V_N|$ and we can use reachability in directed graphs.

**Example:** $G_1:

$$\begin{align*}
S & \rightarrow A \mid B \\
A & \rightarrow S \mid a \\
B & \rightarrow S \mid b
\end{align*}$$

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

We get:

$$\begin{align*}
S & \rightarrow S \mid e \mid b \mid A \mid B \\
A & \rightarrow S \mid a \\
B & \rightarrow S \mid e \mid b \mid S
\end{align*}$$

Removing unit-productions

$$\begin{align*}
S & \rightarrow S \mid e \mid b \\
A & \rightarrow S \mid e \\
B & \rightarrow S \mid e \mid b
\end{align*}$$

Note: $A, B$ are now unreachable, and hence useless.
We have seen: removal of : useless symbols, E-productions, unit-productions.

Does the order of the steps matter?

Observation:

- removing useless : does not add productions at all
  (and therefore not E-productions or unit-productions)
- removing E-productions: could add unit-productions
- removing unit-productions: needs removing E-productions first
  - could create useless symbols
  - cannot create E-productions

⇒ 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 $a \in V_T$, $A, B, C \in V_N$

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

1) Eliminate $E$-prod. and unit-prod.
   \[ \Rightarrow \text{ all productions are of the form} \]
   \[ A \rightarrow e \]
   \[ A \rightarrow X_1 \cdots X_k \quad (k \geq 2) \]
   \[ \text{with } A \in V_M, \quad e \in V_T, \quad X_1, \ldots, X_k \in V \]

2) Remove "mixed bodies"
   
   for each $e \in V_T$, add a new nonterminal $V_e$ and production $V_e \rightarrow e$
   
   in each production $A \rightarrow X_1 \cdots X_k$, replace $e$ with $V_e$

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

3) "Factor" long productions
   
   for each $A \rightarrow A_1 \cdots A_k$ with $k \geq 3$
   
   - add new nonterminals $B_1, \ldots, B_{k-2}$
   
   - replace $A \rightarrow A_1 \cdots A_k$
     
     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: \( G, \) \[
\begin{align*}
S &\rightarrow ABB \mid \varepsilon b \\
A &\rightarrow B \varepsilon \mid \varepsilon \varepsilon \\
B &\rightarrow aA \varepsilon B
\end{align*}
\]

Step 1: nothing to do

Step 2: \[
\begin{align*}
V_e &\rightarrow \varepsilon \\
V_b &\rightarrow b \\
S &\rightarrow ABB \mid V_e V_b \\
A &\rightarrow BV_e \mid V_b V_e \\
B &\rightarrow V_e A V_b B
\end{align*}
\]

Step 3: \[
\begin{align*}
V_e &\rightarrow \varepsilon \\
V_b &\rightarrow b \\
S &\rightarrow AB \varepsilon \mid V_e V_b \\
B_n &\rightarrow BB \\
A &\rightarrow BV_e \mid V_b V_e \\
B &\rightarrow V_e C_1 \\
C_1 &\rightarrow AC_2 \\
C_2 &\rightarrow V_b B
\end{align*}
\]
Pumping lemma for CFLs:

Example: \[ L_1 = \{ a^m b^m \mid m \geq 1 \} \]
\[ L_2 = \{ a^n b^m c^m \mid m, n \geq 1 \} \]
\[ L_3 = \{ a^m b^{2m} c^m \mid m \geq 1 \} \]

Which are CFLs?

- \( L_1 = \{ S \rightarrow aSb \} \Rightarrow L_1 \) is a CFL

- \( L_2 = \{ a^n b^m c^m \mid m, n \geq 1 \} = \{ a^n b^m \mid m \geq 1 \} \cap \{ b^m c^m \mid m \geq 1 \} \)
  \( S \rightarrow AC \)
  \( A \rightarrow aSb \mid aA \)
  \( C \rightarrow bC \mid bCC \) \Rightarrow \( L_2 \) is a CFL

- \( L_3 \) is not a CFL, because number of \( b \)'s depends on its context, both to the left and to the right.

How can we prove that?

Pumping lemma for CFLs

Let \( L \) be a CFL

Then there exists a constant \( m > 0 \) such that

for all \( z \in L \) with \( |z| \geq m \)

there exist a decomposition \( z = uvwx \ y \) such that

\[ |vwx| \leq m \]
\[ |v|x| > 0 \]

for all \( i \geq 0 \), \( uv^iwx^i y \in L \)
Example: $L_3 = \{ a^n b^{2^n} c^n \ | \ n > 1 \}$ is not a CFL.

Assume $L_3$ is a CFL.

Pick P.L.: there exists $m > 0$

Choose $z = b^m c^m$ (|z| > m)

Take an arbitrary decomposition $z = uvwx$ such that $|vwx| \leq m$ and $|vx| > 0$.

Choose $i = 0$, and claim $uvw^ixy = uvwx \notin L$.

We get a contradiction.

Let's show that $uvw^ixy \notin L$:

Since $|vwx| \leq m$, either $vwx$ has no $a$'s or no $c$'s.

Case 1: $vwx$ has no $a$'s

Let $z' = uv^0 w^ix y = u w^i y$

$|z'| = |z| - |vwx| = 4m - |vx| < 4m$

$> 0$

But $z'$ has no $a$'s, and since $|z'| < 4m$ we cannot have $z' = b^m c^m$.

$\Rightarrow z' \notin L$

Case 2: $vwx$ has no $c$'s

Similar.

Exercise 7.1: Show that $L = \{ a^n b^n c^n \ | \ n > 1 \}$ is not a CFL.

Exercise 7.2: Show that $L = \{ww \ | \ w \in \{a,b\}^* \}$ is not a CFL. [Hint: Consider $a^nb^nc^n$.]
Proof of pumping lemma:

Let \( L \) be a CFL and \( G = (V_N, V_T, P, S) \) a CNF grammar for \( L = \{ \varepsilon \} \).

We exploit the following fact:

If \( z \in L \) and \( z \) has a parse tree \( T \) for \( G \) and \( |T| \leq \frac{2^k}{k} \) for some \( k \),

then \( |\text{yield}(T)| \leq 2^{k-1} \) symbols.

Intuitively: productions of CNF grammars have

only two nonterminals on LHS,

\( \Rightarrow T \) is a binary tree

Formally: by induction on \( k \)

Let \( k = |V_N| \) and \( n = 2^k \).

Consider any \( z \in L \) with \( |z| > n = 2^k \).

By fact, parse tree \( T \) for \( z \) has a root-leaf path \( Q \)
of length \( q \geq k+1 \).

Since \( |V_N| = k < q \), by pigeonhole principle, some
non-terminal appears more than once on \( Q \).

More precisely: \( V_k, V_k, V_{k+1}, \ldots, V_{q-1}, V_q \) will contain
a repetition.
Let $i, j$ be such that $q - k < i < j < q$
and $V_i = V_j = A$

We have $z = mwxy$

$$S \Rightarrow mAy \Rightarrow mAvAxy \Rightarrow mwyxy$$

Hence: $A \Rightarrow vAxy$ and $A \Rightarrow w$

Therefore: $A \Rightarrow vAxy \Rightarrow v^2Axy \Rightarrow \ldots \Rightarrow v^iAxy \Rightarrow mwyxy$

and also: $S \Rightarrow mAy \Rightarrow mAvAxy \Rightarrow mwyxy$

or $S \Rightarrow mAy \Rightarrow mwy$

We get: $V_i \geq 0 \quad S \Rightarrow mwyxy$

and hence $V_i \geq 0 \quad mw^iwx^iy \in L(A)$

In pictures

![Diagram 1](image1)

![Diagram 2](image2)

![Diagram 3](image3)

Moreover:
1) $|v| > 0$ since $G$ has no $E$-productions
2) $|vw| < m = 2^k$ since $V_i = A$ is at height $\leq k + 1$

and by fact has yield $vw$ of length $\leq 2^{k+1} - 1 = 2^k$. Read.
Closure properties of context-free languages

Fundamental operation: substitution \( s(w) \) of a string \( w \).

Consider an alphabet \( \Sigma \); for every \( a \in \Sigma \) we define a language \( L_a \) on any alphabet (not necessarily \( \Sigma \) of course). We shall denote
\[
L_a = s(a)
\]
Consider now a string \( w = a_1 a_2 \ldots a_n \in \Sigma^* \); the substitution \( s(w) \)
of \( w \) in the language
\[
\{ x_1 x_2 \ldots x_n \mid x_i \in s(a_i) \ \text{for all} \ i \in \{1, \ldots, n\} \}
\]
which is the concatenation of all \( s(a_i) \) for \( i \in \{1, \ldots, n\} \):
\[
s(w) = s(a_1)^o s(a_2)^o \ldots s(a_n)^o
\]
The definition extends to languages: given a language \( L \),
we define \( s(L) = \bigcup_{w \in L} s(w) \).

Example

Let \( s(o) = \{ a^m b^m \mid m \geq 1 \} \)
\( s(1) = \{ aa, bb \} \)
\( s \) is a substitution on \( \Sigma = \{0,1\} \).

Let \( w = 01 \); therefore \( s(w) = s(0)s(1) \); we have
\[
s(w) = \{ a^m b^m a a \mid m \geq 2 \} \cup \{ a^m b^{m+2} \mid m \geq 1 \}.
\]
**Theorem**

Let $L$ be a context-free language over an alphabet $\Sigma$, and $s$ a substitution on $\Sigma$ such that $s(a)$ is a context-free language for every $a \in \Sigma$. Then $s(L)$ is a context-free language.

**Proof (sketch)**

The idea is that we consider a context-free grammar for $L$ and we replace each terminal $a \in \Sigma$ with the start symbol (axiom) of the CFG for $s(a)$.

Formally, let $G = (V, \Sigma, P, S)$ the CFG for $L$ and $G_a = (V_a, T_a, P_a, S_a)$ the CFG for each $a \in \Sigma$.

We choose disjunctive variables (non-terminals) to avoid mixing the productions from different grammars.

The grammar $G' = (V', T', P', S)$ for $s(L)$ is as follows:

(i) $V'$ is the union of $V$ and all the $V_a$'s for all $a \in \Sigma$.

(ii) $T'$ is the union of all $T_a$'s for all $a \in \Sigma$.

(iii) $P'$ consists of

1. all productions in all $P_a$'s for all $a \in \Sigma$.
2. the productions of $G$, where each terminal symbol $a$ has been replaced (in the body) by $S_a$.

The typical parse tree for $G'$ starts like a parse tree for $G$, but in the frontier, instead of having symbols of $\Sigma$, has start symbols of $G_a$'s, from which parse-trees for the $G_a$'s stem.
The formal proof that $L(G') = s(L)$ is left to the reader (remember that we need to prove both inclusions $L(G') \subseteq s(L)$ and $L(G') \supseteq s(L)$).

We now show how the substitution theorem can be used for proving familiar closure properties we have already seen for regular languages.

**Theorem.** The class of context-free languages is closed under:

1. Union
2. Concatenation
3. Closure (+) and positive closure (+).

**Proof.**

Let $L_1$ and $L_2$ be CFL's. Consider the language $L = \{1,2\}$ on $\Sigma = \{1,2\}$. If we define $s(1) = L_1$ and $s(2) = L_2$, then $L_1 \cup L_2 = s(L)$. This follows from substitution theorem.
(2) Let $L_1$ and $L_2$ be CFL's. Consider the language $L = \{12\}$ on $\Sigma = \{1,2\}$ and $s(1) = L_1$, $s(2) = L_2$; we have $s(L) = L_1 \cdot L_2$, and again this follows from substitution theorem.

(3) Similarly, to the previous cases, if we choose $L = \{1\}^*$ and $s(1) = L_1$, we have $L_1^* = s(L)$; if $L = \{1\}^+$ and $s(1) = L_1$, then $L_1^+ = s(L)$. Again, the theorem follows from substitution theorem.

# Interaction with a regular language

Unlike regular languages, CFL's are not closed under intersection. The following example provides a proper counterexample.

We know that $L = \{0^m1^m2^m | m \geq 1\}$ is now a CFL. On the contrary, the following languages are context-free:

$L_1 = \{0^m1^m2^k | m \geq 1, i \geq j\}$
$L_2 = \{0^k1^m2^m | m \geq 1, k \geq 1\}$

A grammar for $L_1$ is in fact

$S \rightarrow AB$
$A \rightarrow 0A | 01$
$B \rightarrow 2B | 2$

and one for $L_2$ is

$S \rightarrow AB$
$A \rightarrow 0A | 0$
$B \rightarrow 1B2 | 2$
Also, PDA's accepting $L_1$ and $L_2$ can be constructed (left as exercise to the reader).

Now, notice that $L = L_1 \cap L_2$, in fact $L_2$ requires that the 0's are as many as the 1's, $L_2$ requires that the 1's are as many as the 2's, and $L$ requires both properties.

Now, if CFL's were closed under intersection, we could prove that $L$ is CF, which is a contradiction. So CFL's are not closed under intersection.

On the contrary, if we intersect a CFL with a regular language, we obtain a CFL.

**Theorem**: If $L$ is a CFL and $R$ is a regular language, then $L \cap R$ is a CFL.

**Proof**: The proof is obtained by making a PDA accepting $L$ and a FA accepting $R$ run in parallel; the overall automaton is a PDA, shown in figure:
The result is acceptance if both automata accept the input string.

Formally, let $P = (Q_p, \Sigma, \Gamma, \delta_p, q_p, Z_0, F_p)$ the PDA accepting $L$ by final state, and $A = (Q_A, \Sigma, \delta_A, q_A, F_A)$ a DFA accepting $R$.

The PDA accepting $L \cap R$ is

$P' = (Q_p \times Q_A, \Sigma, \Gamma, \delta, (q_p, q_A), Z_0, F_p \times F_A)

$ with $\delta((q, q'), a, x)$ is the set of all pairs $((r, s), \delta)$ such that:

(i) $s = \delta_A((r, a))$ and

(ii) $(r, t)$ is in $\delta_p(q, a, x)$.

Intuitively, for each move of $P$, we make the same move in $A$, and at the same time we carry along the corresponding state of $A$.

The formal proof of the fact that $L(P') = L \cap R$ is left to the reader.

---

**Theorem**

Let $L, L_1, L_2$ be CFL's and $R$ be a regular language; then we have

1. $L - R$ is a CFL
2. $\overline{L}$ is not necessarily a CFL
3. $L_1 - L_2$ is not necessarily a CFL
Proof

(1) Observe that \( L - R = L \cap \overline{R} \); \( \overline{R} \) is regular because \( R \) is regular (closure property); from the theorem at page 7, we have the thesis.

(2) By contradiction, assume that the complement of a CFL is in general a CFL. Then \( \overline{L_1} \cup \overline{L_2} \) is a CFL. But we observe that \( L_1 \cap L_2 = \overline{L_1} \cup \overline{L_2} \), which is not a CFL in general. Since this is a contradiction, we have the thesis.

(2) The language \( \Sigma^* \) is known to be a CFL (it is easy to construct a CFL for it or a PDA accepting it). By contradiction, assume that the difference of two CFLs is a CFL. Then \( \Sigma^* - L \) is in general a CFL. But \( \Sigma^* - L = \overline{L} \), which in general is not a CFL; this is a contradiction, so thesis follows.
Decision properties of context-free languages

Complexity of converting among CFG's and PDA's.

In the following we will consider, as size of a PDA or CFG, the entire length of the representation of the PDA or of the CFG.

The following conversions are linear on the input size (i.e. the time needed is a linear function of the input size):

1. Converting a CFG into a PDA
2. Converting a PDA accepting by final state to one accepting by empty stack
3. Converting a PDA accepting by empty stack to one accepting by final state.
Conversion to Chomsky normal form

**Theorem** Given a CFG $G$ of length $m$, we can find an equivalent grammar in Chomsky normal form in time $O(m^2)$; the resulting grammar has length $O(m^2)$.

**Proof** Observe that:

1. Detecting reachable and generating symbols can be done in time $O(m)$, so an elimination of useless symbols.

2. Construction of unit pairs takes $O(m^2)$; the resulting grammar has size $O(m^2)$.

3. The replacement of terminals by variables takes $O(m)$, resulting in a grammar of length $O(m)$.

4. Breaking productions with bodies longer than two takes $O(m)$ and results in a grammar of length $O(m)$.

Unfortunately, the elimination of $\epsilon$-productions clearly requires time exponential in the length of the production bodies; however, we can apply this step only after having broken productions with body longer than 2. In this case, the running time is $O(m)$ and the resulting grammar has length $O(m)$. This ends the proof.
Testing emptiness of a CFG

It is immediate to see that we can test emptiness of a CFG by checking whether the start symbol is generating.

A naive implementation of this test takes time $O(m^2)$, where $m$ is the length of the grammar. In fact, each pass of the discovery of generating symbols can take $O(m)$, and there are $O(m)$ passes in the worst case.

However, a more careful algorithm can take time $O(m)$- Such algorithm makes use of a data structure which is built in advance. The data structure starts with an entry indexed on the non-terminals (variables), and reports whether variables are generating or not.

For each variable, there is a chain of arrows linking positions in which such variable appears. In the figure, productions $A \rightarrow B c D B$ and $C \rightarrow B A$ are shown.

However, dashed arrows suggest links from productions to the corresponding counts; each count represents the number of variables in the production for which the capability of producing terminals has not yet been verified.

If we discover that a variable is generating, we decrement every body where such variable appears by the number of positions in which it appears; for example, if $B$ is discovered to be generating, we decrement the count of the production $A \rightarrow B c D B$ by 2.

When a count reaches 0, then the variable in the head is generating, and if it is not known to be generating, it is put on a queue (from the head of which variables are taken one by one).

We make the following observations:
(1) The creation and initialization of the array takes time $O(m)$, since there are not more than $m$ variables.

(2) Initializations of links and counts takes $O(m)$, since there are at most $m$ productions, with total length $m$.

(3) When a variable is discovered to be generating, there are two things to do:

(a) for each production: checking whether the count is 0, and putting the variable in the head of the queue if necessary; this takes $O(1)$ for each production, and therefore $O(m)$ overall;

(b) visiting the positions of the production bodies where the variable appears: work taking time proportional to the length of the production bodies, i.e. $O(m)$.

We can conclude that checking emptiness can be done in time $O(m)$. 
Testing membership in a context-free language

Consider a string $w$, with $|w|=n$. A naive algorithm for testing membership of $w$ in a CFL $L$ proceeds as follows. We construct a CFG $G$ for $L$ in Chomsky normal form. We notice that for a CFG derivation tree an binary, end a binary tree having no leaves (i.e., producing a word of length $n$) has $2n-1$ non-terminal nodes. In fact, if we add a leaf to a tree with $m$ leaves, we need to add exactly one non-leaf node. Therefore we can test membership of $w$ in $L$ by constructing all trees with $2n-1$ leaves; this of course takes time $O(2^n)$.

A much more clever algorithm is CYK (Cocke–Younger–Kasami), or table-filling algorithm.

We start from a grammar $G=(V,T,P,S)$ in Chomsky normal form. The input is

$$w = a_1 a_2 \ldots a_n \in T^*$$

We build a table as follows.

$$\begin{array}{c}
X_{15} \\
X_{14} X_{25} \\
X_{13} X_{24} X_{35} \\
X_{12} X_{13} X_{34} X_{45} \\
X_{11} X_{22} X_{33} X_{44} X_{55} \\
a_1 a_2 a_3 a_4 a_5 \ldots
\end{array}$$

The table cell $X_{ij}$ is the set of variables $\{A \mid A \Rightarrow^* a_i a_{i+1} \ldots a_j\}$; so we are interested to know whether $S$ appears in $X_{1m}$. We work now by row from bottom to top.
**BASE STEP** (first row)

\[ X_{ij} = \{ A \mid A \rightarrow a_i \text{ is a production of } G \} \]

**INDUCTIVE STEP**

To compute \( X_{ij} \), we exploit the \( X_{ik} \)'s in the row below, which tell us everything on all strings shorter than \( a_i a_{i+1} \ldots a_j \).

Every derivation \( A \Rightarrow^* a_i a_{i+1} \ldots a_j \) must start with a step of the form \( A \Rightarrow BC \). Therefore, we have

\[
\begin{align*}
B & \Rightarrow^* a_i a_{i+1} \ldots a_k \\
C & \Rightarrow^* a_{k+1} a_{k+2} \ldots a_j
\end{align*}
\]

So, in order to have \( A \) in \( X_{ij} \), we must find \( B \) and \( C \) in the lower row, and \( k \geq 1 \) such that:

(i) \( i \leq k < j \)
(ii) \( B \in X_{ik} \)
(iii) \( C \in X_{k+1,j} \)
(iv) \( A \Rightarrow BC \) is a production in \( G \).

**Complexity**: The technique requires to compare at most \( n^2 \) pairs of previously computed sets. There are \( n(n+1)/2 \) cells in the table, so the algorithm takes time \( O(n^3) \).

(Note: \( n \) is the length of the input string, while we consider the size of \( G \) to be constant.)

**Example**

\[
\begin{align*}
S & \rightarrow AB \mid BC \\
A & \rightarrow BA \mid e \\
B & \rightarrow CC \mid A \\
C & \rightarrow AB \mid e
\end{align*}
\]

\[
\begin{align*}
S & \rightarrow AC \\
S & \rightarrow BC \\
B & \rightarrow BB \\
S & \rightarrow AB \mid SC \mid SA \\
B & \rightarrow AC \mid AC \mid B \mid AC
\end{align*}
\]

\[
\begin{array}{c}
A \quad C \\
2 \quad e \quad e \quad A
\end{array}
\]
Undecidable CFL problems

As we will see in the following, there are some problems regarding CFL's that cannot be solved by an algorithm. This means that there may be an algorithm, but in some cases the execution of the algorithm may not terminate.

1. Is a given CFG ambiguous?
2. Is a given CFG inherently ambiguous?
3. Is the intersection of two CFL's empty?
4. Are two given CFL's equal?
5. Is a given CFL on an alphabet $\Sigma$ equal to $\Sigma^*$?