Context-free grammars

We move on from regular languages to **context-free languages**, a language family that contains all regular languages but also some non-regular ones. For example

$$\{a^n b^n \mid n \geq 0\}$$

is a context-free language. (On the other hand there are many simple languages which are not context-free, for example $\{a^n b^n c^n \mid n \geq 0\}$.)

Context-free languages are recognized by automata with access to an infinite memory that is organized as a stack (LIFO, last-in-first-out). These automata are called **pushdown automata**.

We also have a generative model for the family, so-called **context-free grammars**.

Context-free languages have applications in compiler design (parsers). The syntax of programming languages is often given in the form of context-free grammar, or equivalent Backus-Naur form (BN-form).

Example. Here is an example of a context-free grammar. It uses **variables** A and B, and **terminal** symbols a and b. We have following **productions** (also called rewrite rules):

$$
\begin{aligned}
A &\longrightarrow AbA \\
A &\longrightarrow B \\
B &\longrightarrow aBa \\
B &\longrightarrow b
\end{aligned}
$$

In a word containing variables and terminals we can replace any variable with the word on the right hand side of a production for that variable. For example, in the word

$$aBabAbA$$

we may replace the first occurrence of variable A by AbA, obtaining word

Example. Here is an example of a context-free grammar. It uses **variables** A and B, and **terminal** symbols a and b. We have following **productions** (also called rewrite rules):

$$\begin{aligned} A &\longrightarrow AbA \\ A &\longrightarrow B \\ B &\longrightarrow aBa \\ B &\longrightarrow b \end{aligned}$$

In a word containing variables and terminals we can replace any variable with the word on the right hand side of a production for that variable. For example, in the word

$$aBabAbA$$

we may replace the first occurrence of variable A by AbA, obtaining word

$$aBabAbAbA$$

Then we can decide to replace the variable B by b which gives the word

Example. Here is an example of a context-free grammar. It uses **variables** A and B, and **terminal** symbols a and b. We have following **productions** (also called rewrite rules):

$$\begin{aligned} A &\longrightarrow AbA \\ A &\longrightarrow B \\ B &\longrightarrow aBa \\ B &\longrightarrow b \end{aligned}$$

In a word containing variables and terminals we can replace any variable with the word on the right hand side of a production for that variable. For example, in the word

$$aBabAbA$$

we may replace the first occurrence of variable A by AbA, obtaining word

$$aBabAbAbA$$

Then we can decide to replace the variable B by b which gives the word

$$ababAbAbA$$

Rewriting one variable is called a **derivation step** and we denote

$$aBabAbA \Rightarrow aBabAbAbA \Rightarrow ababAbAbA$$

Derivation is non-deterministic: usually we have many choices how to proceed.

Example. Here is an example of a context-free grammar. It uses **variables** A and B, and **terminal** symbols a and b. We have following **productions** (also called rewrite rules):

$$\begin{aligned} A &\longrightarrow AbA \\ A &\longrightarrow B \\ B &\longrightarrow aBa \\ B &\longrightarrow b \end{aligned}$$

In a word containing variables and terminals we can replace any variable with the word on the right hand side of a production for that variable. For example, in the word

$$aBabAbA$$

we may replace the first occurrence of variable A by AbA, obtaining word

$$aBabAbAbA$$

Then we can decide to replace the variable B by b which gives the word

$$ababAbAbA$$

Rewriting one variable is called a **derivation step** and we denote

$$aBabAbA \Rightarrow aBabAbAbA \Rightarrow ababAbAbA$$

Derivation is non-deterministic: usually we have many choices how to proceed.

We can continue derivation as long as there exist variables in the word. Once the word contains only terminal symbols the derivation terminates.

Example continues:

$$\begin{aligned} A &\longrightarrow AbA \\ A &\longrightarrow B \\ B &\longrightarrow aBa \\ B &\longrightarrow b \end{aligned}$$

One variable is called the **start symbol**- In this example, let us choose A as the start symbol. All derivations start with the start symbol A, *i.e.*, initially the word is A.

The **language defined by the grammar** contains all words of terminal symbols that can be obtained from the start symbol by applying the productions on the variables.

Example continues:

$$\begin{aligned} A &\longrightarrow AbA \\ A &\longrightarrow B \\ B &\longrightarrow aBa \\ B &\longrightarrow b \end{aligned}$$

One variable is called the **start symbol**- In this example, let us choose A as the start symbol. All derivations start with the start symbol A, *i.e.*, initially the word is A.

The **language defined by the grammar** contains all words of terminal symbols that can be obtained from the start symbol by applying the productions on the variables.

For example, the word *aabaababa* is in the language defined by our grammar since we have the following valid derivation:

$A \Rightarrow$

Example continues:

$$\begin{aligned} A &\longrightarrow AbA \\ A &\longrightarrow B \\ B &\longrightarrow aBa \\ B &\longrightarrow b \end{aligned}$$

One variable is called the **start symbol**- In this example, let us choose A as the start symbol. All derivations start with the start symbol A, *i.e.*, initially the word is A.

The **language defined by the grammar** contains all words of terminal symbols that can be obtained from the start symbol by applying the productions on the variables.

For example, the word $aabaababa$ is in the language defined by the grammar since we have the following valid derivation:

$$A \Rightarrow AbA \Rightarrow BbA \Rightarrow aBabA \Rightarrow aaBaabA \Rightarrow aabaabA \Rightarrow aabaabB \Rightarrow aabaabaBa \Rightarrow aabaababa$$

Example continues:

$$\begin{aligned} A &\longrightarrow AbA \\ A &\longrightarrow B \\ B &\longrightarrow aBa \\ B &\longrightarrow b \end{aligned}$$

One variable is called the **start symbol**- In this example, let us choose A as the start symbol. All derivations start with the start symbol A, *i.e.*, initially the word is A.

The **language defined by the grammar** contains all words of terminal symbols that can be obtained from the start symbol by applying the productions on the variables.

For example, the word $aabaababa$ is in the language defined by the grammar since we have the following valid derivation:

$$A \Rightarrow AbA \Rightarrow BbA \Rightarrow aBabA \Rightarrow aaBaabA \Rightarrow aabaabA \Rightarrow aabaabB \Rightarrow aabaabaBa \Rightarrow aabaababa$$

On the other hand, the word aaa is not in the language because the only way to produce letter a is to use the production

$$B \longrightarrow aBa$$

and it creates two a's, so the number of a's has to be even.

Here's the formal definition of context-free grammars.

A context-free grammar $G = (V, T, P, S)$ consists of

- two disjoint alphabets, V and T, containing **variables** (=nonterminals) and **terminals**, respectively,
- a finite set P of **productions** of the form

$$A \longrightarrow \alpha$$

  where $A \in V$ is a variable, and $\alpha \in (V \cup T)^*$ is a word of terminals and variables,
- a **start symbol** $S \in V$.

For any $\alpha, \beta \in (T \cup V)^*$ we denote

$$\alpha \Rightarrow \beta$$

if β is obtained from α by rewriting one variable in α using some production from P. This is called a **derivation step**. More precisely,

$$xAy \Rightarrow xwy$$

for $A \in V$ and $x, y, w \in (V \cup T)^*$ if and only if

$$A \longrightarrow w \ \in P.$$

For any $\alpha, \beta \in (T \cup V)^*$ we denote

$$\alpha \Rightarrow \beta$$

if β is obtained from α by rewriting one variable in α using some production from P. This is called a **derivation step**. More precisely,

$$xAy \Rightarrow xwy$$

for $A \in V$ and $x, y, w \in (V \cup T)^*$ if and only if

$$A \longrightarrow w \ \in P.$$

We write

$$\alpha \Rightarrow^* \beta$$

if there is a sequence of derivation steps

$$\alpha = \alpha_1 \Rightarrow \alpha_2 \Rightarrow \alpha_3 \Rightarrow \cdots \Rightarrow \alpha_n = \beta$$

that starts with α and leads to β. (The number of derivation steps can be zero, so always $\alpha \Rightarrow^* \alpha$.) We say that α **derives** β.

Notation $\alpha \Rightarrow^+ \beta$ means that α derives β using at least one derivation step, and $\alpha \Rightarrow^n \beta$ means a derivation with n derivation steps.

A word $\alpha \in (V \cup T)^*$ is called a **sentential form** if it can be derived from the start symbol S, *i.e.*, if

$$S \Rightarrow^* \alpha.$$

A word $\alpha \in (V \cup T)^*$ is called a **sentential form** if it can be derived from the start symbol S, *i.e.*, if

$$S \Rightarrow^* \alpha.$$

The language $L(G)$ generated by grammar G consists of all sentential forms that contain only terminals. In other words, a word is in $L(G)$ if and only if

- it can be derived from the start symbol S using the productions in P, and
- it contains only terminal symbols.

In short,

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

A language is called a **context-free language** if it is $L(G)$ for some context-free grammar G.

Example 1. Consider the grammar $G = (V, T, P, S)$ where $V = \{S\}$, $T = \{a, b\}$, and P contains productions

$$\begin{aligned} S &\longrightarrow aSb \\ S &\longrightarrow \varepsilon \end{aligned}$$

To make notations shorter we may use the following convention: productions of the same variable may be combined on one line, separated by symbol $|$. So we may write

$$S \longrightarrow aSb \mid \varepsilon$$

What is $L(G)$?

Example 2. Consider the grammar $G = (V, T, P, E)$ where $V = \{E, N\}$, $T = \{+, *, (,), 0, 1\}$, and P contains the productions

$$\begin{aligned} E &\longrightarrow E + E \mid E * E \mid (E) \mid N \\ N &\longrightarrow 0N \mid 1N \mid 0 \mid 1 \end{aligned}$$

Example 2. Consider the grammar $G = (V, T, P, E)$ where $V = \{E, N\}$, $T = \{+, *, (,), 0, 1\}$, and P contains the productions

$$\begin{aligned} E &\longrightarrow E + E \mid E * E \mid (E) \mid N \\ N &\longrightarrow 0N \mid 1N \mid 0 \mid 1 \end{aligned}$$

For example, all following words are in the language $L(G)$:

$$\begin{array}{l} 0 \\ 0 * 1 + 111 \\ (1 + 1) * 0 \\ (1 * 1) + (((0000)) * 1111) \end{array}$$

For instance $(1 + 1) * 0$ is derived by

$E \Rightarrow$

Example 2. Consider the grammar $G = (V, T, P, E)$ where $V = \{E, N\}$, $T = \{+, *, (,), 0, 1\}$, and P contains the productions

$$
\begin{array}{rcl}
E & \longrightarrow & E + E \mid E * E \mid (E) \mid N \\
N & \longrightarrow & 0N \mid 1N \mid 0 \mid 1
\end{array}
$$

For example, all following words are in the language $L(G)$:

$$
\begin{array}{l}
0 \\
0 * 1 + 111 \\
(1 + 1) * 0 \\
(1 * 1) + (((0000)) * 1111)
\end{array}
$$

For instance $(1 + 1) * 0$ is derived by

$$E \Rightarrow E * E \Rightarrow (E) * E \Rightarrow (E + E) * E \Rightarrow^3 (N + N) * N \Rightarrow^3 (1 + 1) * 0$$

Example 2. Consider the grammar $G = (V, T, P, E)$ where $V = \{E, N\}$, $T = \{+, *, (,), 0, 1\}$, and P contains the productions

$$\begin{aligned} E &\longrightarrow E + E \mid E * E \mid (E) \mid N \\ N &\longrightarrow 0N \mid 1N \mid 0 \mid 1 \end{aligned}$$

For example, all following words are in the language $L(G)$:

$$\begin{array}{l} 0 \\ 0 * 1 + 111 \\ (1 + 1) * 0 \\ (1 * 1) + (((0000)) * 1111) \end{array}$$

For instance $(1 + 1) * 0$ is derived by

$$E \Rightarrow E * E \Rightarrow (E) * E \Rightarrow (E + E) * E \Rightarrow^3 (N + N) * N \Rightarrow^3 (1 + 1) * 0$$

Here are some words that are **not** in the language $L(G)$:

$$\begin{array}{l} 1(1 \\ () \\ 1 * 1 * 1 * 1* \end{array}$$

- From variable N all nonempty strings of symbols 0 and 1 can be derived.
- From variable E we can derive all well-formed arithmetic expressions containing operations '*' and '+', parenthesises, and strings of 0's and 1's derived from variable N.

Example 3. Grammar $G = (V, T, P, S)$ where $V = \{S\}$, $T = \{a, b\}$ and P contains productions

$$S \longrightarrow aSbS \mid bSaS \mid \varepsilon$$

What is $L(G)$?

Example 3. Grammar $G = (V, T, P, S)$ where $V = \{S\}$, $T = \{a, b\}$ and P contains productions

$$S \longrightarrow aSbS \mid bSaS \mid \varepsilon$$

What is $L(G)$?

Clearly every word that belongs to $L(G)$ must contain equally many a's and b's. (Every production adds the same number of a's and b's to the sentential form.)

Let us prove (on the blackboard) that $L(G)$ contains all words $w \in \{a, b\}^*$ with equally many a's and b's, *i.e.*, $L(G)$ is the language

$$L = \{w \in \{a, b\}^* \mid \ w \text{ has equally many } a\text{'s and } b\text{'s } \}$$

Example 3. Grammar $G = (V, T, P, S)$ where $V = \{S\}$, $T = \{a, b\}$ and P contains productions

$$S \longrightarrow aSbS \mid bSaS \mid \varepsilon$$

What is $L(G)$?

Clearly every word that belongs to $L(G)$ must contain equally many a's and b's. (Every production adds the same number of a's and b's to the sentential form.)

Let us prove (on the blackboard) that $L(G)$ contains all words $w \in \{a, b\}^*$ with equally many a's and b's, *i.e.*, $L(G)$ is the language

$$L = \{w \in \{a, b\}^* \mid \ w \text{ has equally many } a\text{'s and } b\text{'s } \}$$

For example, the word $ababba$ has the derivation:

$S \Rightarrow$

Example 3. Grammar $G = (V, T, P, S)$ where $V = \{S\}$, $T = \{a, b\}$ and P contains productions

$$S \longrightarrow aSbS \mid bSaS \mid \varepsilon$$

What is $L(G)$?

Clearly every word that belongs to $L(G)$ must contain equally many a's and b's. (Every production adds the same number of a's and b's to the sentential form.)

Let us prove (on the blackboard) that $L(G)$ contains all words $w \in \{a, b\}^*$ with equally many a's and b's, *i.e.*, $L(G)$ is the language

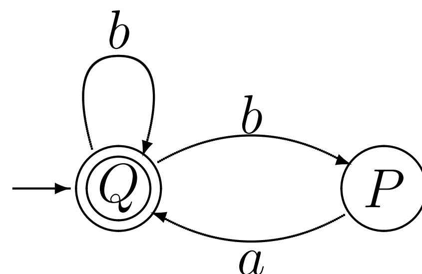
$$L = \{w \in \{a, b\}^* \mid \ w \text{ has equally many } a\text{'s and } b\text{'s } \}$$

For example, the word $ababba$ has the derivation:

$$S \Rightarrow aSbS \Rightarrow abSaSbS \Rightarrow abaSbS \Rightarrow ababS \Rightarrow ababbSaS \Rightarrow ababbaS \Rightarrow ababba$$

Let us prove that every regular language can be generated by a context-free grammar.

Example. Consider the language L recognized by the NFA A



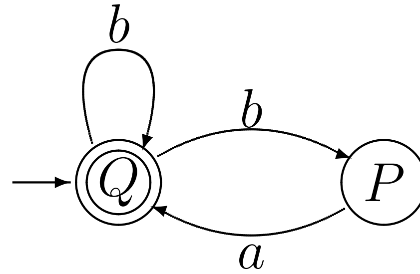
Let us construct a grammar G that generates L. The variables of G are the states $\{Q, P\}$ and the terminals are $\{a, b\}$. The start symbol is the initial state Q. Every transition of A is simulated by a production:

$$\begin{aligned} Q &\longrightarrow bQ \\ Q &\longrightarrow bP \\ P &\longrightarrow aQ \end{aligned}$$

For each transition $q \xrightarrow{x} p$ in A we have the production $q \longrightarrow xp$ in G

So whenever the automaton reads a letter x and goes to state p the grammar produces the same letter x and changes the variable to the current state p.

Example continues.



$$\begin{aligned} Q &\longrightarrow bQ \\ Q &\longrightarrow bP \\ P &\longrightarrow aQ \end{aligned}$$

Derivations by G correspond step-by-step to computations in A:

In A: $\quad Q \xrightarrow{b} Q \xrightarrow{b} P \xrightarrow{a} Q$

In G: $\quad Q \Rightarrow bQ \Rightarrow bbP \Rightarrow bbaQ$

In general, $q \xrightarrow{w} p$ in A if and only if So $q \Rightarrow^* wp$ in G.

To terminate the simulation we add the transition

$$Q \longrightarrow \varepsilon$$

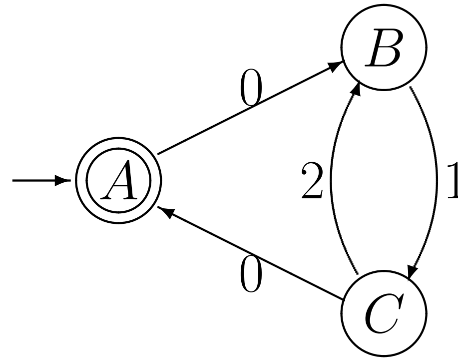
for the final state Q. Accepting computations $Q \xrightarrow{w} Q$ now have corresponding terminating derivations $Q \Rightarrow^* wQ \Rightarrow w$.

The construction can be done on any NFA, proving the following theorem:

Theorem. Every regular language is a context-free language.

Proof.

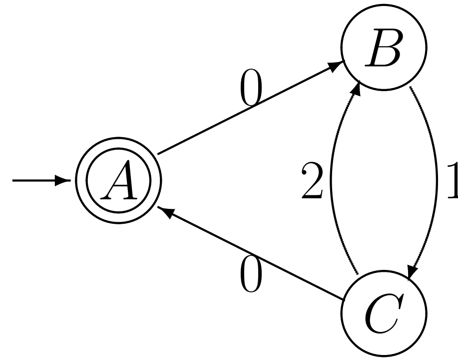
Example. Let us construct a grammar for the language recognized by



$G = (V, T, P, S)$ where

- variables are $V =$
- terminals are $T =$
- the start symbol is $S =$
- the productions are

Example. Let us construct a grammar for the language recognized by



$G = (V, T, P, S)$ where

- variables are $V = \{A, B, C\}$
- terminals are $T = \{0, 1, 2\}$
- the start symbol is $S = A$
- the productions are

$$\begin{aligned} A &\longrightarrow 0B \\ B &\longrightarrow 1C \\ C &\longrightarrow 2B \\ C &\longrightarrow 0A \\ A &\longrightarrow \varepsilon \end{aligned}$$

A grammar is called **right linear** if all productions are of forms

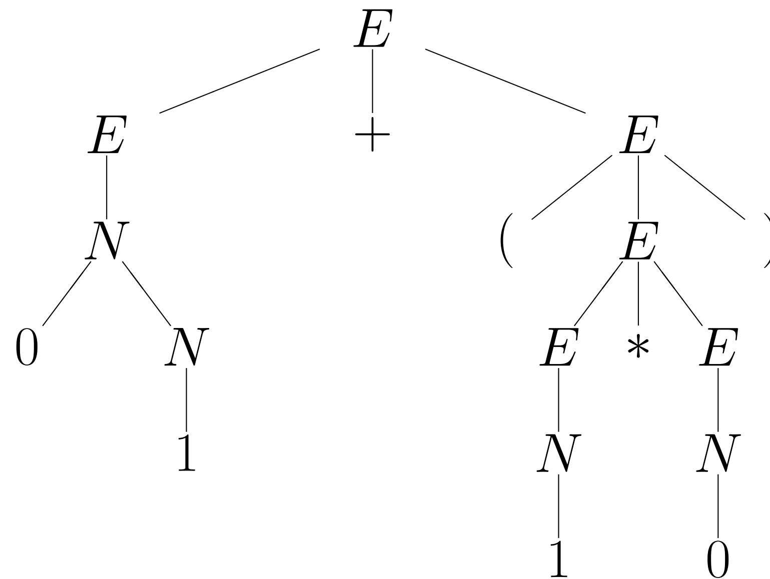
$$\begin{aligned} A &\longrightarrow wB, \\ A &\longrightarrow w \end{aligned}$$

for terminal words $w \in T^*$. In other words, the right hand side of a production may contain only one variable, and it has to be the last symbol.

Our construction always produces a right linear grammar, so every regular language is generated by a right linear grammar. The converse is also true: a language generated by a right linear grammar is always regular.

Derivation trees

Derivations by context-free grammars can be visualized using **derivation trees**, also called **parse trees**. These are rooted, oriented trees. Here is a derivation tree for one of our sample grammars:



All interior nodes are labeled using variables. The label of the root is the start symbol E of the grammar. The children of a node are the letters on the right-hand-side of some production for the parent's variable.

For example, the root has children E, $+$, E, in this order, corresponding to the production $E \longrightarrow E + E$ in the grammar.

A **derivation tree** for grammar $G = (V, T, P, S)$ is a tree whose nodes are labeled by symbols from the set

$$V \cup T \cup \{\varepsilon\}$$

in such a way that

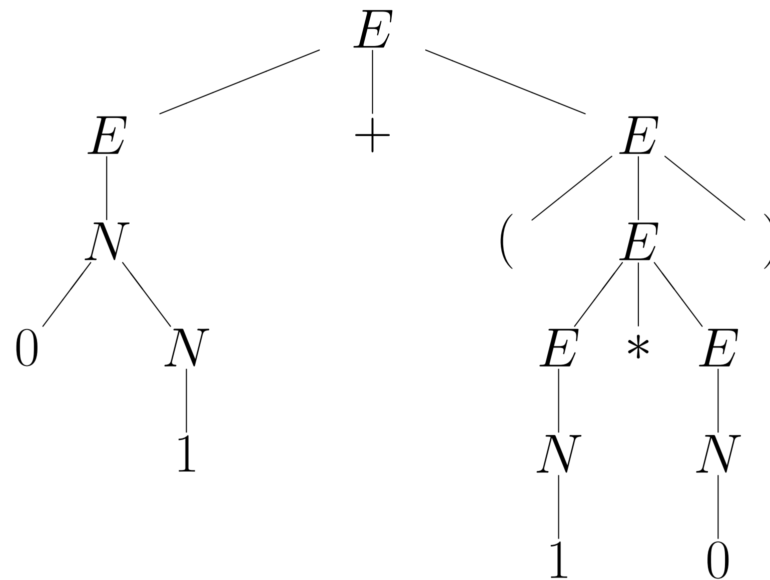
- interior nodes are labeled by variables, *i.e.*, elements of V,
- the root is labeled by the start symbol S,
- if $X_1, X_2, \ldots, X_n$ are the labels of the children of a node labeled by variable A, ordered from left to right, then

  $$A \longrightarrow X_1 X_2 \ldots X_n$$

  is a production in P. If $n \geq 2$ then all $X_i \neq \varepsilon$.

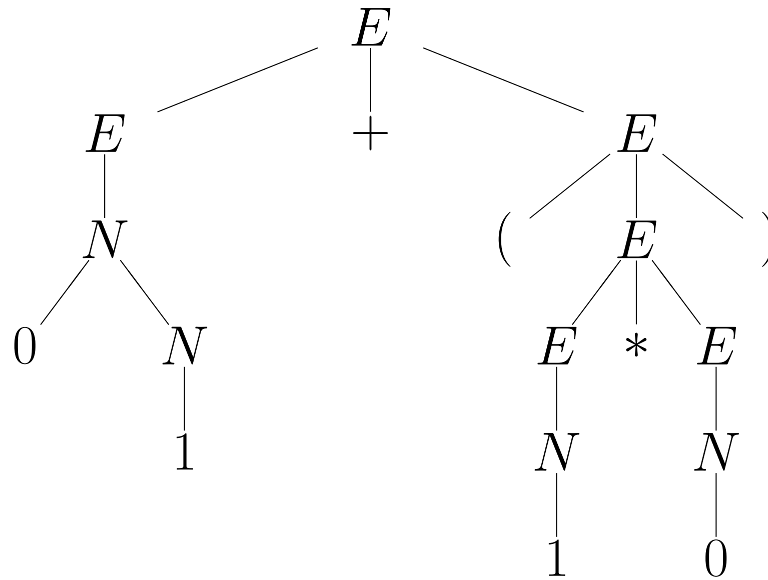
Note that label ε is needed to allow productions $A \longrightarrow \varepsilon$ in the derivation tree.

If one reads the labels of the **leaves** from left to right, one obtains a word over $(V \cup T)^*$. This word is called the **yield** of the derivation tree. For example, the yield of



is $01 + (1 * 0)$.

If one reads the labels of the **leaves** from left to right, one obtains a word over $(V \cup T)^*$. This word is called the **yield** of the derivation tree. For example, the yield of

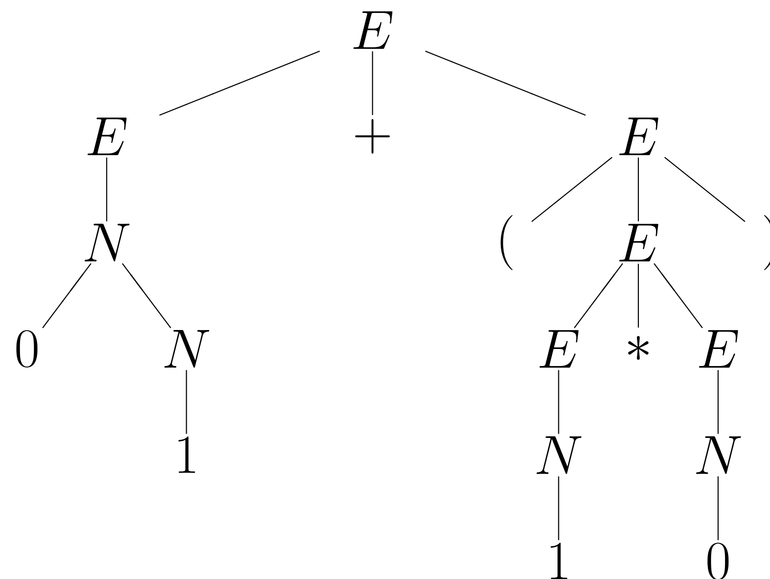


is $01 + (1 * 0)$.

The yield is always a sentential form, and every sentential form is a yield of a derivation tree (obvious?)

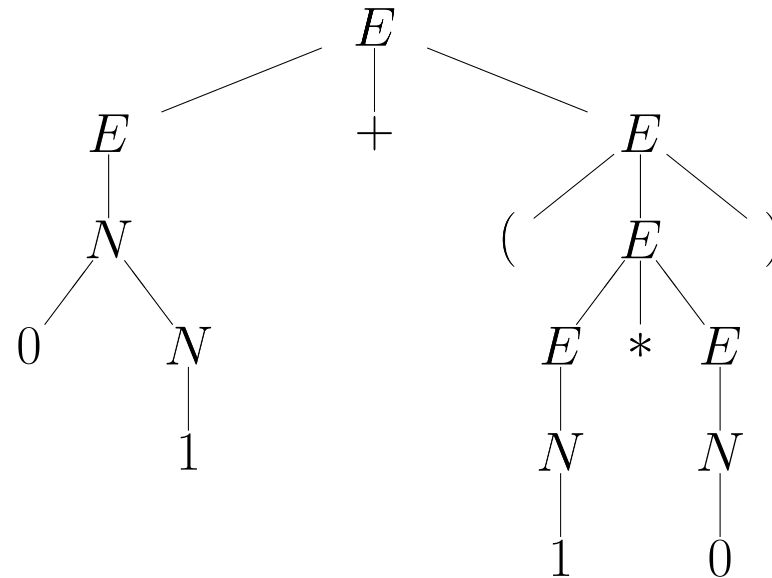
Remark: the yield is obtained by the depth first traversal of the tree, when the subtrees of each node are processed in the left-to-right order, outputting the symbols at all encountered leaves. A program to print the yield of a tree rooted at node x:

```
Yield(x)
Begin
    if x is a leaf then print the label of x
    else
       begin
          let x_1,...x_n be the children of x in the left-to-right order
          for i=1 to n call Yield(x_i)
       end
End
```

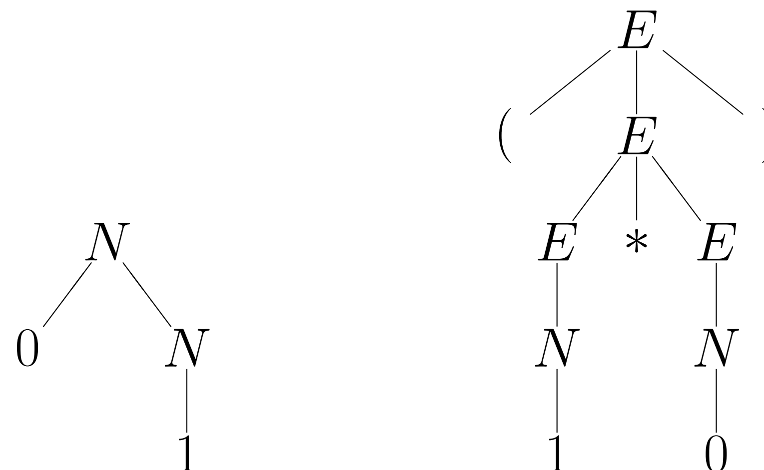


A **subtree** of a derivation tree is the tree formed by some node and all its descendants. A subtree is like a derivation tree except that its root may have a different label than the start symbol. Such a tree is called an ***A*-tree**, where A is the label of the root.

Example.



has, as subtrees, the following N-tree and E-tree



Theorem. Let $G = (V, T, P, S)$ be a context-free grammar. Then $A \Rightarrow^* \alpha$, for some $A \in V$ and $\alpha \in (V \cup T)^*$, if and only if α is the yield of an A-tree.

In particular, $w \in L(G)$ if and only if w is the yield of a derivation tree and w contains only terminal letters.

Proof. An easy induction on the length of the derivation/size of the derivation tree.

Example. Let us construct the derivation tree corresponding to the derivation

$$\begin{aligned} E &\Rightarrow E + E \Rightarrow N + E \Rightarrow 1 + E \Rightarrow 1 + E * E \\ &\Rightarrow 1 + N * E \Rightarrow 1 + 1N * E \Rightarrow 1 + 10 * E \\ &\Rightarrow 1 + 10 * N \Rightarrow 1 + 10 * 1 \end{aligned}$$

Example. Let us construct the derivation tree corresponding to the derivation

$$\begin{aligned} E &\Rightarrow E + E \Rightarrow N + E \Rightarrow 1 + E \Rightarrow 1 + E * E \\ &\Rightarrow 1 + N * E \Rightarrow 1 + 1N * E \Rightarrow 1 + 10 * E \\ &\Rightarrow 1 + 10 * N \Rightarrow 1 + 10 * 1 \end{aligned}$$

Note that the same tree represents several derivations. For example, the derivation

$$\begin{aligned} E &\Rightarrow E + E \Rightarrow E + E * E \Rightarrow E + N * E \\ &\Rightarrow N + N * E \Rightarrow N + 1N * E \Rightarrow N + 1N * N \\ &\Rightarrow 1 + 1N * N \Rightarrow 1 + 1N * 1 \Rightarrow 1 + 10 * 1 \end{aligned}$$

has the same derivation tree. The two derivations only differ in the order of rewriting the variables.

Example. Let us construct the derivation tree corresponding to the derivation

$$\begin{aligned} E &\Rightarrow E+E \Rightarrow N+E \Rightarrow 1+E \Rightarrow 1+E*E \\ &\Rightarrow 1+N*E \Rightarrow 1+1N*E \Rightarrow 1+10*E \\ &\Rightarrow 1+10*N \Rightarrow 1+10*1 \end{aligned}$$

Note that the same tree represents several derivations. For example, the derivation

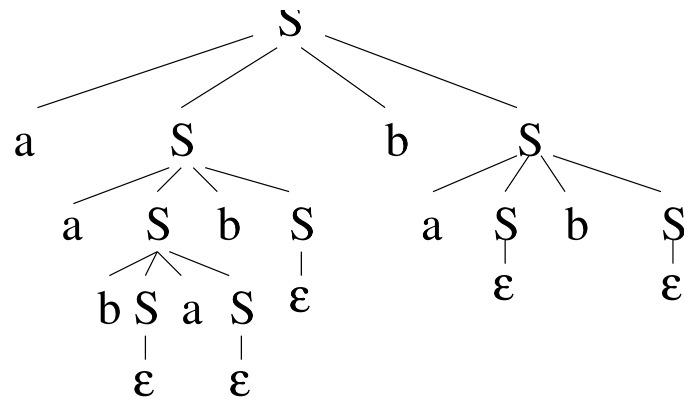
$$\begin{aligned} E &\Rightarrow E+E \Rightarrow E+E*E \Rightarrow E+N*E \\ &\Rightarrow N+N*E \Rightarrow N+1N*E \Rightarrow N+1N*N \\ &\Rightarrow 1+1N*N \Rightarrow 1+1N*1 \Rightarrow 1+10*1 \end{aligned}$$

has the same derivation tree. The two derivations only differ in the order of rewriting the variables.

A derivation is called **leftmost** if at every derivation step the leftmost variable of the sentential form is rewritten. The first derivation is leftmost, and the second one is not. **Rightmost** derivations are defined analogously: One always replaces the rightmost variable.

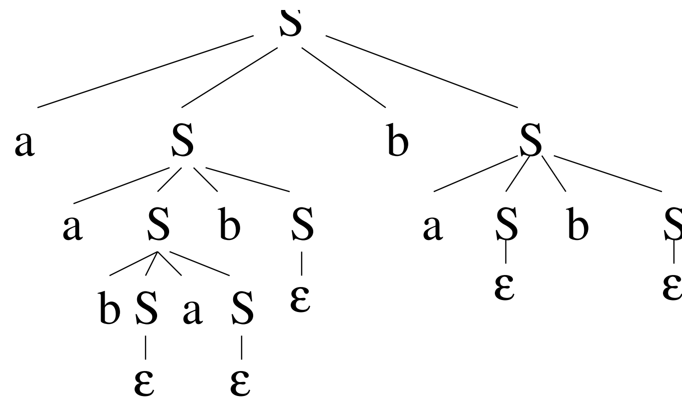
Every **derivation tree defines a unique leftmost derivation** (and a unique rightmost derivation). The leftmost derivation rewrites the variables in the order in which the depth first traversal of the tree from left to right encounters them. (The rightmost derivation corresponds to the depth first traversal from right to left.)

Example. Let us write the leftmost derivation corresponding to the derivation tree



Every **derivation tree defines a unique leftmost derivation** (and a unique rightmost derivation). The leftmost derivation rewrites the variables in the order in which the depth first traversal of the tree from left to right encounters them. (The rightmost derivation corresponds to the depth first traversal from right to left.)

Example. Let us write the leftmost derivation corresponding to the derivation tree

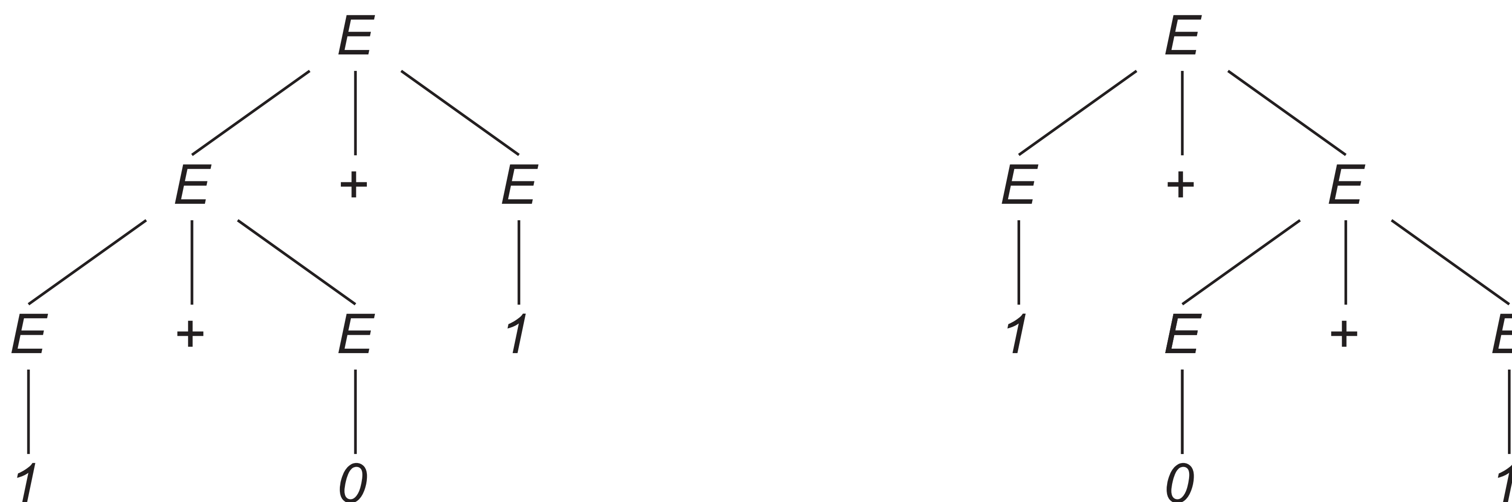


$$S \Rightarrow aSbS \Rightarrow aaSbSbS \Rightarrow aabSaSbSbS \Rightarrow aabaSbSbS \Rightarrow aababSbS \Rightarrow aababbS \Rightarrow aababbaSbS \Rightarrow aababbabS \Rightarrow aababbabS$$

Conclusion: There is a one-to-one correspondence between derivation trees and leftmost (rightmost) derivations.

Even though every derivation tree defines a unique leftmost derivation, some words may have several different leftmost (or rightmost) derivations. This happens if a word is the yield of several different derivation trees.

Example. In our sample grammar word $1+0+1$ is the yield of two different derivation trees



corresponding to two different leftmost derivations

$$E \Rightarrow E+E \Rightarrow E+E+E \Rightarrow 1+E+E \Rightarrow 1+0+E \Rightarrow 1+0+1, \text{ and}$$
$$E \Rightarrow E+E \Rightarrow 1+E \Rightarrow 1+E+E \Rightarrow 1+0+E \Rightarrow 1+0+1,$$

A context-free grammar G is called **ambiguous** if some word has more than one derivation tree.

Equivalently: G is ambiguous if some word has more than one leftmost derivation (or rightmost derivation).

A context-free grammar G is called **ambiguous** if some word has more than one derivation tree.

Equivalently: G is ambiguous if some word has more than one leftmost derivation (or rightmost derivation).

One can prove ambiguity by finding two leftmost derivations for some word.

Proving unambiguity is harder:

- There does not exist an algorithm that would determine if a given context-free grammar G is ambiguous or unambiguous. It an **undecidable** property. (We are going to prove this before then end of the semester.)
- As a consequence, there exist unambiguous grammars whose unambiguity cannot be proved.

For individual grammars one can come up with ad hoc proofs.

Example. Grammar $G = (\{S\}, \{a, b\}, P, S)$ with productions

$$S \longrightarrow aSSb \mid ab.$$

Is this grammar ambiguous or unambiguous ?

Example. Grammar $G = (\{S\}, \{a, b\}, P, S)$ with productions

$$S \longrightarrow aSSb \mid ab.$$

Is this grammar ambiguous or unambiguous ?

The grammar is unambiguous. (Proof on the blackboard)

Example. Grammar $G = (\{S\}, \{a, b\}, P, S)$ where P contains productions

$$S \longrightarrow aSb \mid aaS \mid \varepsilon.$$

Is G ambiguous or unambiguous ?

Example. Grammar $G = (\{S\}, \{a, b\}, P, S)$ where P contains productions

$$S \longrightarrow aSb \mid aaS \mid \varepsilon.$$

Is G ambiguous or unambiguous ?

Ambiguous: word aab has two different leftmost derivations

$$S \Rightarrow aaS \Rightarrow aaaSb \Rightarrow aaab$$
$$S \Rightarrow aSb \Rightarrow aaaSb \Rightarrow aaab.$$

For some context-free languages there exist only ambiguous grammars. Such languages are called **inherently ambiguous**.

(Note that inherent ambiguity is a property of a language while ambiguity is a property of a grammar.)

Also inherent ambiguity is **undecidable**. In other words, there does not exist an algorithm for determining if the language $L(G)$ generated by given context-free grammar G is inherently ambiguous or not.

An example of an inherently ambiguous context-free language:

$$\{a^n b^n c^m d^m \mid n, m \geq 1\} \cup \{a^n b^m c^m d^n \mid n, m \geq 1\}$$

Example. Grammar $G = (\{S\}, \{a, b\}, P, S)$ where P contains productions

$$S \longrightarrow aSb \mid aaS \mid \varepsilon.$$

We saw that this grammar is ambiguous because the word $aaab$ has two leftmost derivations.

But is the language $L(G)$ it generates inherently ambiguous ?

Example. Grammar $G = (\{S\}, \{a, b\}, P, S)$ where P contains productions

$$S \longrightarrow aSb \mid aaS \mid \varepsilon.$$

We saw that this grammar is ambiguous because the word $aaab$ has two leftmost derivations.

But is the language $L(G)$ it generates inherently ambiguous ?

No: the same language is generated by the unambiguous grammar $(\{S, A\}, \{a, b\}, P', S$
with productions

$$\begin{aligned} S &\longrightarrow aSb \mid A, \\ A &\longrightarrow aaA \mid \varepsilon. \end{aligned}$$

(Using variable A we force the grammar first to use productions $S \longrightarrow aSb$ before productions $S \longrightarrow aaS$.)

Example. Grammar $G = (\{S\}, \{a, b\}, P, S)$ with productions

$$S \longrightarrow aSb \mid bS \mid \varepsilon.$$

Ambiguous or unambiguous ?

Example. Grammar $G = (\{S\}, \{a, b\}, P, S)$ with productions

$$S \longrightarrow aSb \mid bS \mid \varepsilon.$$

Ambiguous or unambiguous ?

Unambiguous.

Example. Grammar $G = (\{S\}, \{a, b\}, P, S)$ with productions

$$S \longrightarrow aSbS \mid bSaS \mid \varepsilon.$$

Ambiguous or unambiguous ?

Ambiguous because $abab$ has two different leftmost derivations.

Example. Grammar $G = (\{S\}, \{a, b\}, P, S)$ with productions

$$S \longrightarrow aSbS \mid bSaS \mid \varepsilon.$$

Ambiguous or unambiguous ?

Ambiguous because $abab$ has two different leftmost derivations.

What about the corresponding language

$$L(G) = \{w \in \{a, b\}^* \mid \ w \text{ contains equally many } a\text{'s and } b\text{'s } \}.$$

Is it inherently ambiguous or not ?

Example. Grammar $G = (\{S\}, \{a,b\}, P, S)$ with productions

$$S \longrightarrow aSbS \mid bSaS \mid \varepsilon.$$

Ambiguous or unambiguous ?

Ambiguous because $abab$ has two different leftmost derivations.

What about the corresponding language

$$L(G) = \{w \in \{a,b\}^* \mid \ w \text{ contains equally many } a\text{'s and } b\text{'s }\}.$$

Is it inherently ambiguous or not ?

$L(G)$ is not inherently ambiguous. The same language is generated by the unambiguous grammar $(\{S,P,N\}, \{a,b\}, P, S)$ with productions

$$\begin{aligned} S &\longrightarrow aPbS \mid bNaS \mid \varepsilon \\ P &\longrightarrow aPbP \mid \varepsilon \\ N &\longrightarrow bNaN \mid \varepsilon \end{aligned}$$

(Proof in the homework.)

Grammars G and G' are called **equivalent** if they generate the same language $L(G) = L(G')$. Certain types of productions are undesirable in a grammar, and we would like to find an equivalent grammar that does not contain such productions.

We want to remove productions of form

$$\begin{array}{lll} A \longrightarrow \varepsilon & (\varepsilon\textbf{-production}), \text{ and} \\ A \longrightarrow B & (\textbf{unit production}). \end{array}$$

We also want to remove symbols that are unnecessary in the sense that they are not used in any terminating derivations.

The simplification is done in the following order:

1. Remove ε -productions.
2. Remove unit productions.
3. Remove variables that do not derive any terminal strings.
4. Remove symbols that cannot be reached from the start symbol.

STEP 1. Remove ε-productions.

Given a CFG G we (effectively) construct a CFG G' that does not have any ε-productions and

$$L(G') = L(G) - \{\varepsilon\}$$

(So ε is lost from the generated language. This cannot be avoided because a grammar without ε-productions cannot generate the word ε.)

The idea of G' is to remove from derivation trees all subtrees whose yield is ε.

STEP 1. Remove ε-productions.

Given a CFG G we (effectively) construct a CFG G' that does not have any ε-productions and

$$L(G') = L(G) - \{\varepsilon\}$$

(So ε is lost from the generated language. This cannot be avoided because a grammar without ε-productions cannot generate the word ε.)

The idea of G' is to remove from derivation trees all subtrees whose yield is ε.

Let us call a variable A **nullable** if $A \Rightarrow^* \varepsilon$.

We can identify the nullable variable using a simple marking procedure:

- First, mark as nullable those variables A that have a production $A \longrightarrow \varepsilon$.
- Then keep marking variables A that have productions

$$A \longrightarrow \alpha$$

  where α is a word of variables that have all been already marked nullable. Repeat this until no new variables can be marked.

Then we construct a new set of productions as follows: For every production of the original grammar

$$A \longrightarrow X_1 X_2 \ldots X_n$$

where all $X_i \in V \cup T$, the new production set will have all productions

$$A \longrightarrow \alpha_1 \alpha_2 \ldots \alpha_n$$

where

- $\alpha_i = X_i$ if X_i is not nullable, or is a terminal symbol,
- $\alpha_i = \varepsilon$ or $\alpha_i = X_i$ if X_i is nullable,
- $\alpha_1 \alpha_2 \ldots \alpha_n \neq \varepsilon$.

Remark: If several X_i are nullable, all combinations of $\alpha_i = \varepsilon$ and X_i are used. One original production can result in up to 2^k new productions if the right-hand-side contains k nullable variables.

Then we construct a new set of productions as follows: For every production of the original grammar

$$A \longrightarrow X_1 X_2 \dots X_n$$

where all $X_i \in V \cup T$, the new production set will have all productions

$$A \longrightarrow \alpha_1 \alpha_2 \dots \alpha_n$$

where

- $\alpha_i = X_i$ if X_i is not nullable, or is a terminal symbol,
- $\alpha_i = \varepsilon$ or $\alpha_i = X_i$ if X_i is nullable,
- $\alpha_1 \alpha_2 \dots \alpha_n \neq \varepsilon$.

$L(G') \subseteq L(G)$: Each production $A \longrightarrow \alpha$ in G' is obtained from a production in G by removing nullable variables. Thus in G there is a derivation $A \Rightarrow^* \alpha$.

Any derivation in G' can hence be turned into a derivation in G by replacing each derivation step $xAy \Rightarrow x\alpha y$ by a derivation $xAy \Rightarrow^* x\alpha y$.

Then we construct a new set of productions as follows: For every production of the original grammar

$$A \longrightarrow X_1 X_2 \ldots X_n$$

where all $X_i \in V \cup T$, the new production set will have all productions

$$A \longrightarrow \alpha_1 \alpha_2 \ldots \alpha_n$$

where

- $\alpha_i = X_i$ if X_i is not nullable, or is a terminal symbol,
- $\alpha_i = \varepsilon$ or $\alpha_i = X_i$ if X_i is nullable,
- $\alpha_1 \alpha_2 \ldots \alpha_n \neq \varepsilon$.

$L(G') \subseteq L(G)$: Each production $A \longrightarrow \alpha$ in G' is obtained from a production in G by removing nullable variables. Thus in G there is a derivation $A \Rightarrow^* \alpha$.

Any derivation in G' can hence be turned into a derivation in G by replacing each derivation step $xAy \Rightarrow x\alpha y$ by a derivation $xAy \Rightarrow^* x\alpha y$.

$L(G) - \{\varepsilon\} \subseteq L(G')$: Let $w \in L(G) - \{\varepsilon\}$. It has a derivation tree T. Removing from T all subtrees whose yield is ε results in a derivation tree of G' with the same yield w.

STEP 2. Remove unit productions.

Assume we have removed all ε-productions from P. The right-hand-side of every production in P is some non-empty word.

Let us find an equivalent grammar that does not contain ε-productions or unit productions (=productions $A \longrightarrow B$ for variables A, B).

The idea is similar to step 1: We anticipate all possible unit derivations, and plug them in directly.

First, for every variable A we find all variables B, such that

$$A \Rightarrow^* B.$$

(In this derivation only unit productions can be used.)

Such variable-to-variable derivations $A \Rightarrow^* B$ for any fixed A can be effectively found using a marking procedure:

- First, mark variable A.
- Then keep marking variables B such that there is a unit production

$$X \longrightarrow B$$

  where X is already marked. Repeat this until no new variables can be marked.

A non-mandatory step: At this stage, if one wants, one may simplify the grammar by removing duplicate variables.

If A and B are different variables such that $A \Rightarrow^* B$ and $B \Rightarrow^* A$, it is clear that both variables derive exactly same sentential forms. So we may trim the grammar by replacing B by A everywhere, both left- and righthand sides of all productions.

Also, productions $A \longrightarrow A$ can be directly removed as useless.

Then we construct a new set of productions as follows: For every **non-unit production**

$$B \longrightarrow \alpha$$

of the grammar and for every variable A such that $A \Rightarrow^* B$ we take in the new grammar the production

$$A \longrightarrow \alpha.$$

The new grammar G' then has no unit productions.

Then we construct a new set of productions as follows: For every **non-unit production**

$$B \longrightarrow \alpha$$

of the grammar and for every variable A such that $A \Rightarrow^* B$ we take in the new grammar the production

$$A \longrightarrow \alpha.$$

The new grammar G' then has no unit productions.

$L(G') \subseteq L(G)$: For each production $A \longrightarrow \alpha$ in G' there is a derivation $A \Rightarrow^* B \Rightarrow \alpha$ in G. Any derivation in G' can hence be turned into a derivation in G by replacing each derivation step $xAy \Rightarrow x\alpha y$ by a derivation $xAy \Rightarrow^* x\alpha y$.

Then we construct a new set of productions as follows: For every **non-unit production**

$$B \longrightarrow \alpha$$

of the grammar and for every variable A such that $A \Rightarrow^* B$ we take in the new grammar the production

$$A \longrightarrow \alpha.$$

The new grammar G' then has no unit productions.

$L(G') \subseteq L(G)$: For each production $A \longrightarrow \alpha$ in G' there is a derivation $A \Rightarrow^* B \Rightarrow \alpha$ in G. Any derivation in G' can hence be turned into a derivation in G by replacing each derivation step $xAy \Rightarrow x\alpha y$ by a derivation $xAy \Rightarrow^* x\alpha y$.

$L(G) \subseteq L(G')$: Let $w \in L(G)$. Consider terminal derivations $S \Rightarrow^* w$ that use productions of G and G'. If there is a derivation step $xAy \Rightarrow xBy$ that uses a unit production $A \longrightarrow B$ then the derivation can be shortened by removing xBy. The remaining shorter derivation still uses productions of G and G'. Repeat this until no applications of unit productions remain.

The final derivation then only uses productions of G' because all non-unit productions of G are also in G'.

STEP 3. Remove variables that do not derive any terminal strings.

Such variables can be simply removed together with all productions containing the variable on either side of the production. This does not effect the generated language since the variable is not used in any terminating derivation.

To find variables that **do** generate some terminal string, we apply similar marking procedure as with ε-productions:

- mark all variables A such that there is a production $A \longrightarrow w$ in P, where w contains only terminals.
- Then keep marking variables A that have productions

  $$A \longrightarrow \alpha$$

  where α is a word that contains only terminals and variables that have already been marked. Repeat this until no new variables can be marked.

The variables that have **not** been marked can be removed. All productions containing removable variables are unnecessary, and can be deleted.

STEP 4. Remove symbols (variables and terminals) that are not reachable from the start symbol S.

Any production that contains such an unreachable symbol can be removed.

To find symbols that **can** be reached from S are found again using a marking procedure:

- Initially, mark the start symbol S.
- Then, for all productions $A \longrightarrow \alpha$ where A is a marked variable, mark all symbols that appear in α. Repeat this until no new variables can be marked.

All symbols that have **not** been marked can not be reached from the initial symbol, and they can be removed.

Remark: After step 4 all remaining variables still derive a terminal word, so we do not need to repeat step 3 again.

(Step 3 may introduce new unreachable symbols, so if we do step 4 before step 3, we may have to do step 4 again.)

After steps 1,2,3 and 4 we have found an equivalent (except the possible loss of the the empty word ε) grammar that does not have ε-productions, unit productions, or useless symbols.

After steps 1,2,3 and 4 we have found an equivalent (except the possible loss of the the empty word ε) grammar that does not have ε-productions, unit productions, or useless symbols.

If the empty word was in the original language and we want to include it in the new grammar we may introduce a **nonrecursive start symbol**. This means that we add a new variable S', make it the start symbol, and add productions

$$S' \longrightarrow S$$
$$S' \longrightarrow \varepsilon$$

where S is the old start symbol. The new grammar is equivalent to the original one, and it has only one ε-production that can be used only once.

Example. Let us simplify the following grammar:

$$\begin{aligned} S &\longrightarrow Aa \mid CbDS \\ A &\longrightarrow \varepsilon \mid BA \\ B &\longrightarrow AA \\ C &\longrightarrow a \\ D &\longrightarrow aAD \end{aligned}$$

A grammar G is in the **Chomsky normal form** (or CNF) if all productions are of the forms

$$\begin{aligned} A &\longrightarrow BC, \text{ or} \\ A &\longrightarrow a \end{aligned}$$

where A, B and C are variables, and a is a terminal.

(Right-hand-sides of productions consist of two variables, or one terminal.)

Theorem. Every context-free language L without ε is generated by a grammar that is in the Chomsky normal form.

The conversion from a given context-free grammar to an equivalent CNF grammar is effective.

A grammar G is in the **Chomsky normal form** (or CNF) if all productions are of the forms

$$\begin{aligned} A &\longrightarrow BC, \text{ or} \\ A &\longrightarrow a \end{aligned}$$

where A, B and C are variables, and a is a terminal.

(Right-hand-sides of productions consist of two variables, or one terminal.)

Theorem. Every context-free language L without ε is generated by a grammar that is in the Chomsky normal form.

The conversion from a given context-free grammar to an equivalent CNF grammar is effective.

Proof. Let G be a grammar that generates L. We may assume that there are no ε-productions or unit productions in G. (Because they can be removed, as we have seen.)

Now all productions with one symbol on the right hand side are terminal productions $A \longrightarrow a$. These are then already in the Chomsky normal form.

The only productions $A \longrightarrow \alpha$ that are not in the Chomsky normal form are **long**, meaning that $|\alpha| \geq 2$.

For every terminal symbol a we introduce a new variable V_a and a production

$$V_a \longrightarrow a$$

Variable V_a hence only derives word a.

(Note that the production $V_a \longrightarrow a$ we add is in the Chomsky normal form.)

The only productions $A \longrightarrow \alpha$ that are not in the Chomsky normal form are **long**, meaning that $|\alpha| \geq 2$.

For every terminal symbol a we introduce a new variable V_a and a production

$$V_a \longrightarrow a$$

Variable V_a hence only derives word a.

(Note that the production $V_a \longrightarrow a$ we add is in the Chomsky normal form.)

Now in each long production $A \longrightarrow \alpha$ we replace every terminal symbol a by the corresponding variable V_a. After this change the grammar generates the same language L as before, and now every long production only contains variables.

Example. The production

$$S \longrightarrow cAbbS$$

will be replace by production

$$S \longrightarrow V_c A V_b V_b S$$

where V_a and V_b are new variables. New productions $V_a \longrightarrow a$ and $V_b \longrightarrow b$ are also added.

Clearly the new grammar is equivalent to the original one: Every application of the original production

$$S \longrightarrow cAbbS$$

is replaced by a sequence of derivation steps that first uses production

$$S \longrightarrow V_c A V_b V_b S$$

and then applies productions

$$V_c \longrightarrow c \text{ and } V_b \longrightarrow b$$

to replace all occurrences of variables V_c and V_b by terminals c and b.

So far we have constructed an equivalent grammar whose productions are of the forms

$$\begin{aligned} A &\longrightarrow B_1 B_2 \dots B_n \quad \text{and} \\ A &\longrightarrow a \end{aligned}$$

where $n \geq 2$, and $A, B_1, B_2, \dots, B_n$ are variables, and a terminal symbol.

So far we have constructed an equivalent grammar whose productions are of the forms

$$\begin{aligned} A &\longrightarrow B_1 B_2 \ldots B_n \quad \text{and} \\ A &\longrightarrow a \end{aligned}$$

where $n \geq 2$, and $A, B_1, B_2, \ldots, B_n$ are variables, and a terminal symbol.

The only productions that are not in CNF are

$$A \longrightarrow B_1 B_2 \ldots B_n$$

where $n \geq 3$. For each such production we introduce $n-2$ new variables $D_1, D_2, \ldots, D_{n-2}$, and replace the production by the productions

$$\begin{aligned} A &\longrightarrow B_1 D_1 \\ D_1 &\longrightarrow B_2 D_2 \\ D_2 &\longrightarrow B_3 D_3 \\ &\ldots \\ D_{n-3} &\longrightarrow B_{n-2} D_{n-2} \\ D_{n-2} &\longrightarrow B_{n-1} B_n \end{aligned}$$

One application of the original production gives the same result as applying the new productions one after the other. The new grammar is equivalent to the original one.

Remark: The new variables D_i have to be different for different productions.

Example. Let us find a Chomsky normal form grammar that is equivalent to

$$
\begin{aligned}
S &\longrightarrow SSaA \mid bc \mid c \\
A &\longrightarrow AAA \mid b
\end{aligned}
$$

Another normal form is so-called **Greibach normal form**, or GNF. A grammar is in the Greibach normal form if all productions are of form

$$A \longrightarrow aB_1B_2 \dots B_n$$

where $n \geq 0$ and $A, B_1, B_2, \dots, B_n$ are variables, and a is a terminal symbol.

(All productions contain exactly one terminal symbol and it is the first symbol on the right hand side of the production.)

Theorem. Every context-free language L without ε is generated by a grammar that is in Greibach normal form.

Proof. Skipped.

Pushdown automata

A **pushdown automaton** (PDA) is an NFA that has access to an **infinite memory, organized as a stack**. It turn out that see that the family of languages recogized by PDA is exactly the family of context-free languages.

A PDA consists of the following:

- **Stack**. The stack is a word. The PDA has access only to the leftmost symbol of the stack. (This is called the top of the stack.) During one move of the PDA, the leftmost symbol may be removed ("popped" from the stack) and new symbols may be added ("pushed") on the top of the stack.

- **Input tape**. Similar to finite automata: the input word is normally scanned one symbol at a time. But also ε-moves are possible.

- **Finite state control unit**. The control unit is a non-deterministic finite automaton. Transitions may depend on the next input symbol and the topmost stack symbol.

There are two types of moves: normal moves, and ε-moves.

1. **Normal moves**: Depending on
   (a) the current state of the control unit,
   (b) the next input letter, and
   (c) the topmost symbol on the stack

   the PDA may

   (A) change the state,
   (B) pop the topmost element from the stack,
   (C) push new symbols to the stack, and
   (D) move to the next input symbol.

2. **ε-moves** don't have (b) and (D), i.e. they are done spontaneously without using the input tape.

Example. Let

$$L = \{a^n b^n \mid n \geq 1\}.$$

A PDA that recognizes L has two states, S_a and S_b, and the stack alphabet contains two symbols, A and Z_0. In the beginning the machine is in initial state S_a, and the stack contains only one symbol Z_0, the start symbol of the stack.

Transitions are summarized in this table:

State	Top of stack	Input Symbol a	Input Symbol b	ε
S_a	Z_0	Add one A to the stack, stay in state S_a	—	—
S_a	A	Add one A to the stack, stay in state S_a	Remove one A from the stack, go to state S_b	—
S_b	A	—	Remove one A from the stack, stay in state S_b	—
S_b	Z_0	—	—	Remove Z_0 from the stack, stay in state S_b

A word is accepted if and only if the PDA has an empty stack after reading all input symbols.

An **instantaneous description** (ID) records the configuration of a PDA at given time. It is a triple

$$(q, w, \gamma)$$

where

- q is the **state** of the PDA,
- w is the **remaining input**, i.e. the suffix of the original input that has not been used yet, and
- γ is a word, the **content of the stack**. The first letter of γ is the topmost symbol of the stack.

ID contains all relevant information that is needed in subsequent steps of the computation.

We denote

$$(q_1, w_1, \gamma_1) \vdash (q_2, w_2, \gamma_2)$$

if there exists a move that takes the first ID into the second ID.

We denote

$$(q_1, w_1, \gamma_1) \vdash (q_2, w_2, \gamma_2)$$

if there exists a move that takes the first ID into the second ID. We denote

$$(q_1, w_1, \gamma_1) \vdash^* (q_2, w_2, \gamma_2)$$

if there is a sequence of moves (possibly empty) from the first ID to the second ID. We denote

$$(q_1, w_1, \gamma_1) \vdash^+ (q_2, w_2, \gamma_2)$$

if there is a non-empty sequence of moves (at least one move). Finally, we denote

$$(q_1, w_1, \gamma_1) \vdash^n (q_2, w_2, \gamma_2)$$

if the first ID becomes the second ID in exactly n moves.

We operate PDF under two different modes of acceptance, depending on what is more convenient:

- Acceptance by **empty stack**: An input word is accepted if all input letters are consumed and in the end the stack is empty.
- Acceptance by **final state**: We identify some states as final states. An input word is accepted if all input letters are consumed and in the end the PDA is in a final state.

We will see later that the two possible modes of acceptance are **equivalent**: If there is a PDA that recognizes language L by empty stack then there is also a PDA that accepts L by final state, and vice versa. The conversion between the two modes is effective.

Example. The sample PDA to recognaize $L = \{a^n b^n \mid n \geq 1\}$ accepts using empty stack.

State	Top of stack	Input Symbol a	b	ε
S_a	Z_0	Add one A to the stack, stay in state S_a	—	—
S_a	A	Add one A to the stack, stay in state S_a	Remove one A from the stack, go to state S_b	—
S_b	A	—	Remove one A from the stack, stay in state S_b	—
S_b	Z_0	—	—	Remove Z_0 from the stack, stay in state S_b

An accepting computation on input $aaabbb$:

$(S_a, aaabbb, Z_0) \vdash$

Example. The sample PDA to recognaize $L = \{a^n b^n \mid n \geq 1\}$ accepts using empty stack.

State	Top of stack	Input Symbol a	Input Symbol b	ε
S_a	Z_0	Add one A to the stack, stay in state S_a	—	—
S_a	A	Add one A to the stack, stay in state S_a	Remove one A from the stack, go to state S_b	—
S_b	A	—	Remove one A from the stack, stay in state S_b	—
S_b	Z_0	—	—	Remove Z_0 from the stack, stay in state S_b

An accepting computation on input $aaabbb$:

$$(S_a, aaabbb, Z_0) \vdash (S_a, aabbb, AZ_0) \vdash (S_a, abbb, AAZ_0) \vdash (S_a, bbb, AAAZ_0) \vdash (S_b, bb, AAZ_0) \vdash (S_b, b, AZ_0) \vdash (S_b, \varepsilon, Z_0) \vdash (S_b, \varepsilon, \varepsilon)$$

Formally, a pushdown automaton M consists of

$$(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

where

- Q is a finite **state set**,
- Σ is the **input alphabet**,
- Γ is the **stack alphabet**,
- $q_0 \in Q$ is the **initial state**,
- $Z_0 \in \Gamma$ is the **start symbol** of the stack,
- $F \subseteq Q$ is the set of **final states**,
- δ is a mapping from

  $$Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma$$

  to finite subsets of

  $$Q \times \Gamma^*.$$

Note that $\delta(q, a, Z)$ is a **set** of possible outcomes: PDA are non-deterministic.

1. The interpretation of a transition

$$(p, \gamma) \in \delta(q, a, Z)$$

(where $p, q \in Q$, $a \in \Sigma$, $Z \in \Gamma$, $\gamma \in \Gamma^*$):

In state q, reading input letter a, and Z on the top of the stack, the PDA may go to state p, move to next input symbol, and **replace** Z by γ on top of the stack. The leftmost symbol of γ will be the new top of the stack (if $\gamma \neq \varepsilon$).

This transition means that moves

$$(q, aw, Z\alpha) \vdash (p, w, \gamma\alpha)$$

are allowed (for any $w \in \Sigma^*$, $\alpha \in \Gamma^*$).

2. The interpretation of an ε-transition

$$(p, \gamma) \in \delta(q, \varepsilon, Z)$$

(where $p, q \in Q$, $Z \in \Gamma$, $\gamma \in \Gamma^*$):

In state q and Z on the top of the stack, the PDA may go to state p and replace Z by γ on top of the stack. Note that **no input symbol is consumed**, and the transition can be used regardless of the current input symbol.

This transition means that moves

$$(q, w, Z\alpha) \vdash (p, w, \gamma\alpha)$$

are allowed (for any $w \in \Sigma^*$, $\alpha \in \Gamma^*$).

Let us formally define the two modes of acceptance:

- A word w is accepted by PDA M using **empty stack**, iff

$$(q_0, w, Z_0) \vdash^* (q, \varepsilon, \varepsilon)$$

for some $q \in Q$. Note that q can be any state, final or non-final.

The language recognized using empty stack is denoted by $N(M)$, and it consists of all words accepted using empty stack:

$$N(M) = \{w \in \Sigma^* \mid (q_0, w, Z_0) \vdash^* (q, \varepsilon, \varepsilon)\}$$

Let us formally define the two modes of acceptance:

- A word w is accepted by PDA M using **empty stack**, iff

$$(q_0, w, Z_0) \vdash^* (q, \varepsilon, \varepsilon)$$

  for some $q \in Q$. Note that q can be any state, final or non-final.

  The language recognized using empty stack is denoted by $N(M)$, and it consists of all words accepted using empty stack:

$$N(M) = \{w \in \Sigma^* \mid (q_0, w, Z_0) \vdash^* (q, \varepsilon, \varepsilon)\}$$

- A word w is accepted by PDA M using **final states**, iff

$$(q_0, w, Z_0) \vdash^* (q, \varepsilon, \gamma)$$

  for some $q \in F$, and $\gamma \in \Gamma^*$. Now q has to be a final state, but the stack does not need to be empty.

  The language recognized using final states is denoted by $L(M)$, and it consists of all words accepted using final states:

$$L(M) = \{w \in \Sigma^* \mid (q_0, w, Z_0) \vdash^* (q, \varepsilon, \gamma)\}$$

Example. Our sample PDA is

$$M = (\{S_a, S_b\}, \{a, b\}, \{Z_0, A\}, \delta, S_a, Z_0, \emptyset)$$

where

$$\begin{aligned}
\delta(S_a, a, Z_0) &= \{(S_a, AZ_0)\} \\
\delta(S_a, a, A) &= \{(S_a, AA)\} \\
\delta(S_a, b, A) &= \{(S_b, \varepsilon)\} \\
\delta(S_b, b, A) &= \{(S_b, \varepsilon)\} \\
\delta(S_b, \varepsilon, Z_0) &= \{(S_b, \varepsilon)\}
\end{aligned}$$

and all other sets $\delta(q, a, Z)$ are empty. It does not matter which set we choose as the set of final states, since we use acceptance by empty stack. (Choose, for example $F = \emptyset$.) We have

$$N(M) = \{a^n b^n \mid n \geq 1\}.$$

Example. Let us construct a PDA M such that

$$N(M) = \{w \mid w \in \{a, b\}^* \text{ and } w \text{ is a palindrome } \}.$$

Example. Let us construct a PDA M such that

$$N(M) = \{w \mid w \in \{a, b\}^* \text{ and } w \text{ is a palindrome } \}.$$

Idea: The PDA will read symbols from the input and push them into the stack. At some point it **guesses** that it is in the middle of the input word, and starts popping letters from the stack and comparing them against the following input letters. If all letters match, and the stack and the input string become empty at the same time, the word was a palindrome.

Example. Let us construct a PDA M such that

$$N(M) = \{w \mid w \in \{a,b\}^* \text{ and } w \text{ is a palindrome }\}.$$

Idea: The PDA will read symbols from the input and push them into the stack. At some point it **guesses** that it is in the middle of the input word, and starts popping letters from the stack and comparing them against the following input letters. If all letters match, and the stack and the input string become empty at the same time, the word was a palindrome.

$$M = (\{q_1, q_2\}, \{a, b\}, \{Z_0, A, B\}, \delta, q_1, Z_0, \emptyset)$$

where δ is

$$\begin{aligned}
\delta(q_1, \varepsilon, Z_0) &= \{(q_1, \varepsilon)\} \\
\delta(q_1, a, Z_0) &= \{(q_1, AZ_0), (q_2, AZ_0), (q_2, Z_0)\} \\
\delta(q_1, b, Z_0) &= \{(q_1, BZ_0), (q_2, BZ_0), (q_2, Z_0)\} \\
\delta(q_1, a, A) &= \{(q_1, AA), (q_2, AA), (q_2, A)\} \\
\delta(q_1, b, A) &= \{(q_1, BA), (q_2, BA), (q_2, A)\} \\
\delta(q_1, a, B) &= \{(q_1, AB), (q_2, AB), (q_2, B)\} \\
\delta(q_1, b, B) &= \{(q_1, BB), (q_2, BB), (q_2, B)\} \\
\delta(q_2, b, B) &= \{(q_2, \varepsilon)\} \\
\delta(q_2, a, A) &= \{(q_2, \varepsilon)\} \\
\delta(q_2, \varepsilon, Z_0) &= \{(q_2, \varepsilon)\}
\end{aligned}$$

State q_1 is used in the first half of the input word, state q_2 in the second half.

$$
\begin{aligned}
\delta(q_1, \varepsilon, Z_0) &= \{(q_1, \varepsilon)\} \\
\delta(q_1, a, Z_0) &= \{(q_1, AZ_0), (q_2, AZ_0), (q_2, Z_0)\} \\
\delta(q_1, b, Z_0) &= \{(q_1, BZ_0), (q_2, BZ_0), (q_2, Z_0)\} \\
\delta(q_1, a, A) &= \{(q_1, AA), (q_2, AA), (q_2, A)\} \\
\delta(q_1, b, A) &= \{(q_1, BA), (q_2, BA), (q_2, A)\} \\
\delta(q_1, a, B) &= \{(q_1, AB), (q_2, AB), (q_2, B)\} \\
\delta(q_1, b, B) &= \{(q_1, BB), (q_2, BB), (q_2, B)\} \\
\delta(q_2, b, B) &= \{(q_2, \varepsilon)\} \\
\delta(q_2, a, A) &= \{(q_2, \varepsilon)\} \\
\delta(q_2, \varepsilon, Z_0) &= \{(q_2, \varepsilon)\}
\end{aligned}
$$

The three possible outcomes of some transitions have the following roles in accepting computations:

- use the first transition, if the input letter is before the end of the first half of the input word.
- use the second transition, if the input word has even length and the current input letter is the last letter of the first half, and
- use the third transition, if the input word has odd length and the current input letter is exactly in the middle of the input word.

$$
\begin{aligned}
\delta(q_1, \varepsilon, Z_0) &= \{(q_1, \varepsilon)\} \\
\delta(q_1, a, Z_0) &= \{(q_1, AZ_0), (q_2, AZ_0), (q_2, Z_0)\} \\
\delta(q_1, b, Z_0) &= \{(q_1, BZ_0), (q_2, BZ_0), (q_2, Z_0)\} \\
\delta(q_1, a, A) &= \{(q_1, AA), (q_2, AA), (q_2, A)\} \\
\delta(q_1, b, A) &= \{(q_1, BA), (q_2, BA), (q_2, A)\} \\
\delta(q_1, a, B) &= \{(q_1, AB), (q_2, AB), (q_2, B)\} \\
\delta(q_1, b, B) &= \{(q_1, BB), (q_2, BB), (q_2, B)\} \\
\delta(q_2, b, B) &= \{(q_2, \varepsilon)\} \\
\delta(q_2, a, A) &= \{(q_2, \varepsilon)\} \\
\delta(q_2, \varepsilon, Z_0) &= \{(q_2, \varepsilon)\}
\end{aligned}
$$

Accepting computations for inputs aba and $abba$:

$(q_1, bab, Z_0) \quad \vdash$

$(q_1, abba, Z_0) \vdash$

$$
\begin{aligned}
\delta(q_1, \varepsilon, Z_0) &= \{(q_1, \varepsilon)\} \\
\delta(q_1, a, Z_0) &= \{(q_1, AZ_0), (q_2, AZ_0), (q_2, Z_0)\} \\
\delta(q_1, b, Z_0) &= \{(q_1, BZ_0), (q_2, BZ_0), (q_2, Z_0)\} \\
\delta(q_1, a, A) &= \{(q_1, AA), (q_2, AA), (q_2, A)\} \\
\delta(q_1, b, A) &= \{(q_1, BA), (q_2, BA), (q_2, A)\} \\
\delta(q_1, a, B) &= \{(q_1, AB), (q_2, AB), (q_2, B)\} \\
\delta(q_1, b, B) &= \{(q_1, BB), (q_2, BB), (q_2, B)\} \\
\delta(q_2, b, B) &= \{(q_2, \varepsilon)\} \\
\delta(q_2, a, A) &= \{(q_2, \varepsilon)\} \\
\delta(q_2, \varepsilon, Z_0) &= \{(q_2, \varepsilon)\}
\end{aligned}
$$

Accepting computations for inputs aba and $abba$:

$$
(q_1, bab, Z_0) \vdash (q_1, ab, BZ_0) \vdash (q_2, b, BZ_0) \vdash (q_2, \varepsilon, Z_0) \vdash (q_2, \varepsilon, \varepsilon)
$$

$$
(q_1, abba, Z_0) \vdash (q_1, bba, AZ_0) \vdash (q_2, ba, BAZ_0) \vdash (q_2, a, AZ_0) \vdash (q_2, \varepsilon, Z_0) \vdash (q_2, \varepsilon, \varepsilon
$$

A PDA is called **deterministic** (or DPDA) if every ID has at most one possible move. This means that

- if $\delta(q, \varepsilon, Z)$ is non-empty then $\delta(q, a, Z)$ is empty for every input letter a, and
- All $\delta(q, a, Z)$ and $\delta(q, \varepsilon, Z)$ contain at most one element.

The first condition states that there is no choice between ε-move and non-ε-move. If one can make a move without reading an input letter, then that is the only possible move.

A PDA is called **deterministic** (or DPDA) if every ID has at most one possible move. This means that

- if $\delta(q, \varepsilon, Z)$ is non-empty then $\delta(q, a, Z)$ is empty for every input letter a, and
- All $\delta(q, a, Z)$ and $\delta(q, \varepsilon, Z)$ contain at most one element.

The first condition states that there is no choice between ε-move and non-ε-move. If one can make a move without reading an input letter, then that is the only possible move.

Remark: There exist languages that are recognized by nondeterministic PDA but not by any deterministic PDA. The situation is different than in case of finite automata where determinism was equivalent to non-determinism.

For example, the language of palindromes cannot be recognized by any DPDA.

Theorem. If $L = L(M)$ for some PDA M then there (effectively) exists a PDA M' such that $L = N(M')$.

Proof.

Theorem. If $L = L(M)$ for some PDA M then there (effectively) exists a PDA M' such that $L = N(M')$.

Proof. Idea: M accepts L by final state. We construct a PDA M' that simulates M, with the additional option that when M enters a final state, M' may enter a special **erase the stack** -state q_e and remove all symbols from the stack. Them M' accepts w by empty stack if M entered a final state.

Theorem. If $L = L(M)$ for some PDA M then there (effectively) exists a PDA M' such that $L = N(M')$.

Proof. Idea: M accepts L by final state. We construct a PDA M' that simulates M, with the additional option that when M enters a final state, M' may enter a special **erase the stack** -state q_e and remove all symbols from the stack. Them M' accepts w by empty stack if M entered a final state.

But: We have to be careful to make sure that M' does not accept a word accidentally if M empties the stack without entering an accepting state. We prevent this by inserting a **new bottom of the stack symbol** X_0 below the old start symbol Z_0. Even if M empties the stack, M' will have the symbol X_0 in the stack. We have a new initial state q_0' that simply places Z_0 above X_0 and starts the simulation of M.

If M empties the stack without entering the final state then it is stuck. Corresponding computation in M' gets stuck with X_0 on the stack.

Theorem. If $L = N(M)$ for some PDA M then there (effectively) exists a PDA M' such that $L = L(M')$.

Proof.

Theorem. If $L = N(M)$ for some PDA M then there (effectively) exists a PDA M' such that $L = L(M')$.

Proof. Idea: Now M recognizes L by empty stack. We construct a PDA M' that simulates M and detects when the stack is empty. When that happens the PDA enters a final state. In order to be able to detect the empty stack and change the state, we again introduce a new **bottom of the stack symbol** X_0. As soon as X_0 is revealed the new PDA enters a new final state q_f.

Example. Let us modify our first sample PDA so that it recognizes the language

$$L = \{a^n b^n \mid n \geq 1\}$$

using final states instead of the empty stack.

Next we prove that **PDA recognize all context-free languages**. In fact, all context-free languages are recognized by PDA having only **one state**. We prove this by showing how any context-free grammar can be simulated using only the stack.

Example. Consider the grammar $G = (V, T, P, S)$ where $V = \{S\}$, $T = \{a, b\}$ and P contains productions

$$S \longrightarrow aSbS \mid bSaS \mid \varepsilon.$$

Let us construct a PDA M that recognizes $L(G)$ using empty stack.

Example. Consider the grammar $G = (V, T, P, S)$ where $V = \{S\}$, $T = \{a, b\}$ and P contains productions

$$S \longrightarrow aSbS \mid bSaS \mid \varepsilon.$$

Let us construct a PDA M that recognizes $L(G)$ using empty stack.

The stack symbols of M are the terminals and variables of G. At all times, the content of the stack is a suffix of a sentential form by G, from which the remaining input can be generated.

Initially the stack contains the start symbol of the grammar.

- If the topmost symbol of the stack is a terminal symbol it is compared against next input letter. If they are identical, the symbol is popped from the stack and the PDA moves to the next input letter. If they are different the simulation halts.
- If the topmost symbol of the stack is a variable the PDA replaces it by a right-hand-side of a production using an ε-move.

Theorem. If $L = L(G)$ for some context-free grammar G then there (effectively) exists a PDA M such that $L = N(M)$.

Proof.

Example. Consider the grammar

$$G = (\{S\}, \{a, b\}, P, S)$$

with productions

$$S \longrightarrow \varepsilon \mid aSb$$

that generates $L = \{a^n b^n \mid n \geq 0\}$. The construction gives a PDA M such that $N(M) = L$

Next we prove that PDA accept **only** context-free languages: For a given PDA M we construct an equivalent context-free grammar G.

Theorem. If $L = N(M)$ for some PDA M then there exists a context-free grammar G such that $L = L(G)$.

Proof.

Next we prove that PDA accept **only** context-free languages: For a given PDA M we construct an equivalent context-free grammar G.

Theorem. If $L = N(M)$ for some PDA M then there exists a context-free grammar G such that $L = L(G)$.

Proof. This construction is made complicated by the fact that the PDA may have more than one state. We split the prof in two stages:

Theorem A. If $L = N(M_1)$ for some PDA M_1 then there (effectively) exists a PDA M_2 that has only **one state** such that $L = N(M_2)$.

Theorem B. If $L = N(M_2)$ for some PDA M_2 with **one state**, then there (effectively) exists a context-free grammar G such that $L = L(G)$.

Theorem A. If $L = N(M_1)$ for some PDA M_1 then there (effectively) exists a PDA M_2 that has only **one state** such that $L = N(M_2)$.

Proof.

Theorem A. If $L = N(M_1)$ for some PDA M_1 then there (effectively) exists a PDA M_2 that has only **one state** such that $L = N(M_2)$.

Proof. Idea: The computations of M_1 will be simulated in such a way that the state of M_1 will be stored in the topmost element of the stack in M_2. So at all times the topmost element of the stack contains both the stack symbol of M_1 and the current state of M_1.

Then every move of M_1 can be simulated by M_2 since it knows the state and the topmost stack symbol of M_1. After each step, the next state is written into the new topmost element.

Theorem A. If $L = N(M_1)$ for some PDA M_1 then there (effectively) exists a PDA M_2 that has only **one state** such that $L = N(M_2)$.

Proof. Idea: The computations of M_1 will be simulated in such a way that the state of M_1 will be stored in the topmost element of the stack in M_2. So at all times the topmost element of the stack contains both the stack symbol of M_1 and the current state of M_1.

Then every move of M_1 can be simulated by M_2 since it knows the state and the topmost stack symbol of M_1. After each step, the next state is written into the new topmost element.

But: This is fine as long as at least one new symbol is written in the stack. However, there is one big problem: Where is the next state stored when the stack is only popped and nothing new is written into the stack ? New top of the stack is the symbol that used to be second highest on the stack, and it is not accessible for writing. The new state must have been stored in the stack already when the second highest symbol was pushed into the stack — and this was possibly long before it becomes the topmost symbol.

How do we know long before what is going to be the state of M_1 when a particular stack symbol is revealed to the top of the stack ? Answer: we guess it using non-determinism of M_2. At the time when the symbol becomes the topmost element of the stack we only need to verify that the earlier guess was correct.

In order to be able to verify the guess it has to be stored also on the stack symbol above. Therefore stack symbols of M_2 need to contain two states of M_1: one indicates the state of the machine when the symbol is the topmost element of the stack, and the other state that indicates the state of M_1 when the the element below is the topmost element of the stack.

Therefore the stack of M_2 stores triplets

$$Q \times \Gamma \times Q.$$

Symbol $[q, Z, p]$ on top of the stack indicates that that M_1 is in state q with Z is on top of the stack and p will be the state of M_1 when the element below becomes the topmost symbol of the stack.

The stack is kept **consistent**, meaning that the content of the stack will always look like

$$[q_1, Z_1, q_2][q_2, Z_2, q_3][q_3, Z_3, q_4]\ldots$$

(The third component of a stack symbol is the same as the first component of the next stack symbol below.)

The values $q_2, q_3, q_4 \ldots$ are nondeterministic 'guesses' done by M_2 about the state of M_1 when the corresponding stack elements will get exposed on top of the stack. As the stack shrinks the correctness of the guesses gets verified.

Detailed construction on the blackboard.

Example. Let us construct a one state PDA that is equivalent to the PDA

$$M = (\{S_a, S_b\}, \{a, b\}, \{Z_0, A\}, \delta, S_a, Z_0, \emptyset)$$

where

$$\begin{aligned} \delta(S_a, a, Z_0) &= \{(S_a, AZ_0)\} \\ \delta(S_a, a, A) &= \{(S_a, AA)\} \\ \delta(S_a, b, A) &= \{(S_b, \varepsilon)\} \\ \delta(S_b, b, A) &= \{(S_b, \varepsilon)\} \\ \delta(S_b, \varepsilon, Z_0) &= \{(S_b, \varepsilon)\} \end{aligned}$$

that recognizes language

$$N(M) = \{a^n b^n \mid n \geq 1\}.$$

Theorem B. If $L = N(M_2)$ for some PDA M_2 with **one state**, then there (effectively) exists a context-free grammar G such that $L = L(G)$.

Proof.

Example. Consider the one state PDA

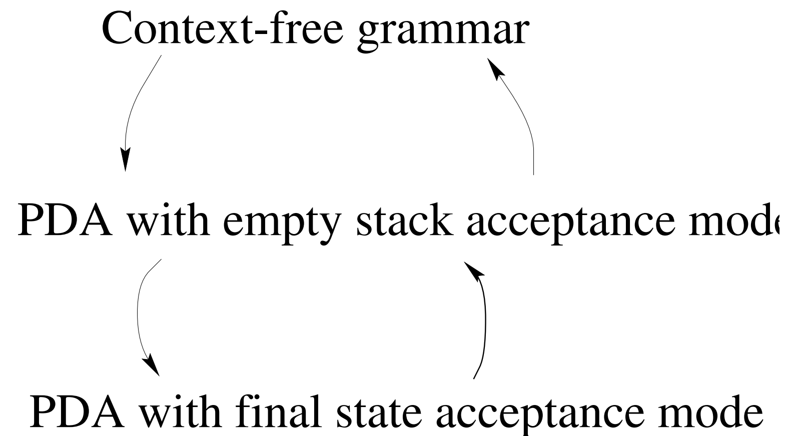
$$M = (\{\#\}, \{a, b\}, \{A, B, Z\}, \delta, \#, Z, \emptyset)$$

where δ contains transitions

$$\begin{aligned}
\delta(\#, \varepsilon, Z) &= \{(\#, AZZA), (\#, B)\} \\
\delta(\#, a, A) &= \{(\#, \varepsilon)\} \\
\delta(\#, b, B) &= \{(\#, \varepsilon)\}
\end{aligned}$$

Let us construct the equivalent grammar.

We have proved four theorems providing effective equivalence of three devices:



The constructions were all effective which means that when investigating closure properties of context-free languages or decision algorithms we may assume that the input is given in any of the three forms.

Pumping lemma for CFL

Not all languages are context-free. A technique for proving that some language is not context-free is a pumping lemma.

Just as in case of regular languages, pumping lemma states a property that every context-free language satisfies. If a language does not satisfy the pumping lemma then the language is not context-free.

Pumping lemma for CFL: Let L be a context-free language. Then there exists a positive number n such that every word $z \in L$ satisfying $|z| \geq n$ can be divided into **five** segments

$$z = uvwxy$$

in such a way that

$$\begin{cases} |vwx| \leq n, \text{ and} \\ v \neq \varepsilon \text{ or } x \neq \varepsilon \end{cases}$$

and for all $i \geq 0$ the word uv^iwx^iy is in the language L.

(The difference to the pumping lemma of regular languages is that now word z contains two subwords v and x that are pumped. Note that subwords v and x are always pumped the same number of times.)

Example. The language $L = \{a^m b^m \mid m \geq 0\}$ satisfies the pumping lemma.

Number $n = 2$ can be used. Let $z = a^m b^m$ be an arbitrary word of the language such that $|z| \geq 2$. This means $m \geq 1$. We break z into five parts as follows:

$$z = \underbrace{a^{m-1}}_{u} \underbrace{a}_{v} \underbrace{\varepsilon}_{w} \underbrace{b}_{x} \underbrace{b^{m-1}}_{y}$$

This division is good since $vx = ab \neq \varepsilon$ and $|vwx| = |ab| \leq 2$.

Subwords v and x can be pumped arbitrarily many times: for every $i \geq 0$

$$uv^i wx^i y = a^{m-1}\ a^i\ \varepsilon\ b^i\ b^{m-1} = a^{m+i-1} b^{m+i-1}$$

is in language L.

Let's analyze in detail why $L = \{a^m b^m \mid m \geq 0\}$ satisfies the pumping lemma.

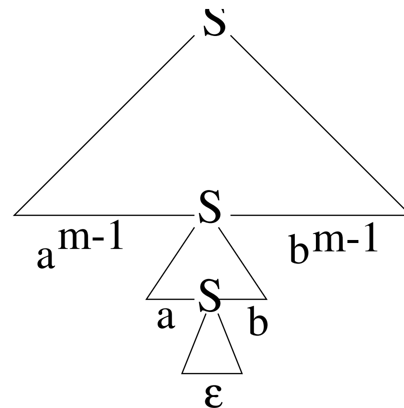
L is generated by a grammar with two productions

$$S \longrightarrow aSb \mid \varepsilon.$$

The word $a^m b^m$, $m \geq 1$, has a derivation

$$S \Longrightarrow^* a^{m-1}Sb^{m-1} \Longrightarrow a^{m-1}aSbb^{m-1} \Longrightarrow a^{m-1}a\varepsilon bb^{m-1}$$

corresponding to the derivation tree



We used three subderivations

$$S \Longrightarrow^* a^{m-1} S b^{m-1},$$
$$S \Longrightarrow^* aSb,$$
$$S \Longrightarrow^* \varepsilon.$$

The middle subderivation $S \Longrightarrow^* aSb$ derives from variable S the variable itself.

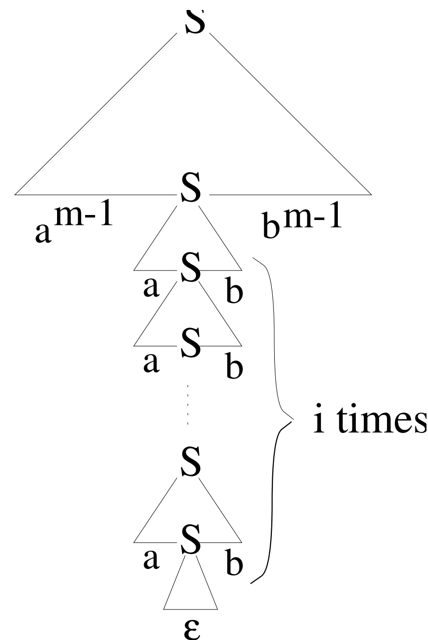
We used three subderivations

$$S \Longrightarrow^* a^{m-1}Sb^{m-1},$$
$$S \Longrightarrow^* aSb,$$
$$S \Longrightarrow^* \varepsilon.$$

The middle subderivation $S \Longrightarrow^* aSb$ derives from variable S the variable itself. It means that the subderivation can be iterated arbitrarily many times. Repeating it i times gives

$$S \Longrightarrow^* a^{m-1}Sb^{m-1} \Longrightarrow^i a^{m-1}a^iSb^ib^{m-1} \Longrightarrow a^{m-1}a^i\varepsilon b^ib^{m-1}$$

corresponding to the derivation tree



In other words, pumping i times words $v = a$ and $x = b$ corresponds to iterating the middle subderivaton i times.

Let us prove that the argument above can be done for any context-free language.

Proof of the pumping lemma. Idea: The proof is based on the fact that during derivations of sufficiently long words, some variable derives a sentential form containing the variable itself. In other words, we have derivations

$$\begin{aligned} S &\Rightarrow^* uAy \\ A &\Rightarrow^* vAx \\ A &\Rightarrow^* w \end{aligned}$$

for some words u, v, w, x, y and variable A. But the derivation $A \Rightarrow^* vAx$ may be repeated arbitrarily many times, say i times:

$$\begin{gathered} S \Rightarrow^* uAy \Rightarrow^* uvAxy \Rightarrow^* uvvAxxy \Rightarrow^* \dots \\ \dots \Rightarrow^* uv^iAx^iy \Rightarrow^* uv^iwx^iy \end{gathered}$$

This shows that uv^iwx^iy is in language L for every $i \geq 0$.

Let us prove that the argument above can be done for any context-free language.

Proof of the pumping lemma. Idea: The proof is based on the fact that during derivations of sufficiently long words, some variable derives a sentential form containing the variable itself. In other words, we have derivations

$$\begin{aligned} S &\Rightarrow^* uAy \\ A &\Rightarrow^* vAx \\ A &\Rightarrow^* w \end{aligned}$$

for some words u, v, w, x, y and variable A. But the derivation $A \Rightarrow^* vAx$ may be repeated arbitrarily many times, say i times:

$$S \Rightarrow^* uAy \Rightarrow^* uvAxy \Rightarrow^* uvvAxxy \Rightarrow^* \ldots$$
$$\cdots \Rightarrow^* uv^iAx^iy \Rightarrow^* uv^iwx^iy$$

This shows that uv^iwx^iy is in language L for every $i \geq 0$.

A precise proof: Let G be a grammar in Chomsky normal form such that $L = L(G)$ (or $L = L(G) \cup \{\varepsilon\}$ if $\varepsilon \in L$).

Define: the **depth** of a derivation tree = the length of the longest path from the root to a leaf.

Lemma. If a derivation tree by a CNF grammar for a word w has depth d then $|w| \leq 2^{d-1}$.

Proof.

Lemma. If a derivation tree by a CNF grammar for a word w has depth d then $|w| \leq 2^{d-1}$.

Proof. Mathematical induction on d:

1° **(base case)** If $d = 1$ then the tree must be the trivial one:

and the word it derives has length 2^{d-1}.

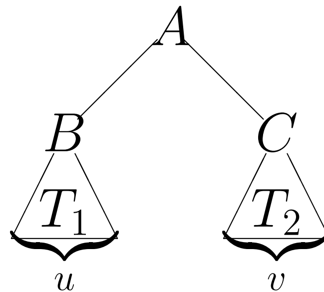
Lemma. If a derivation tree by a CNF grammar for a word w has depth d then $|w| \leq 2^{d-1}$.

Proof. Mathematical induction on d:

1° **(base case)** If $d = 1$ then the tree must be the trivial one:

and the word it derives has length 2^{d-1}.

2° **(inductive step)** Let $d > 1$. The tree has the form



and $w = uv$.

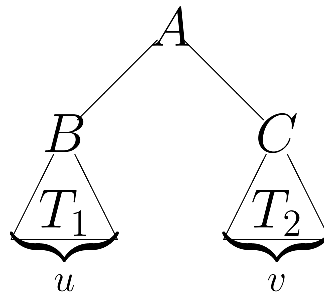
Lemma. If a derivation tree by a CNF grammar for a word w has depth d then $|w| \leq 2^{d-1}$.

Proof. Mathematical induction on d:

1° **(base case)** If $d = 1$ then the tree must be the trivial one:

and the word it derives has length 2^{d-1}.

2° **(inductive step)** Let $d > 1$. The tree has the form



and $w = uv$. The depths of the trees T_1 and T_2 are at most $d-1$, so that (by the inductive hypothesis) $|u| \leq 2^{d-2}$ and $|v| \leq 2^{d-2}$. Therefore

$$|w| = |u| + |v| \leq 2^{d-2} + 2^{d-2} = 2^{d-1}.$$

Let k be the **number of variables** in G, a CNF grammar for L.

Let us choose the number n in the pumping lemma as $n = 2^k$.

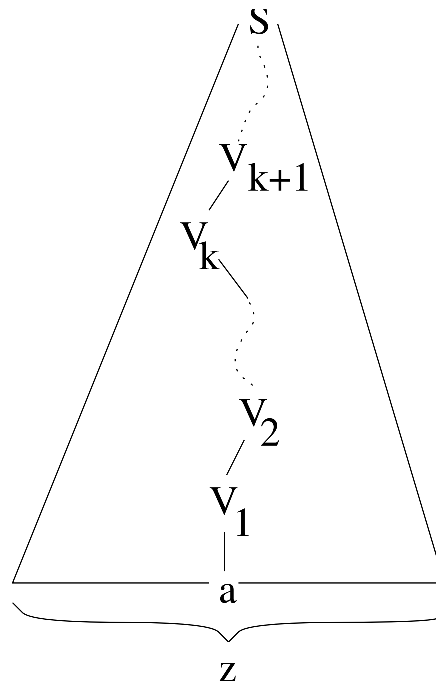
Let $z \in L$ be any word such that $|z| \geq n$, and let T be a derivation tree for z.

The depth of T has to be at least $k + 1$, since a tree of smaller depth can yield only shorter words.

Pick one maximum length path π from the root to a leaf in T. Reading from the leaf up, let

$$a, V_1, V_2, V_3, \ldots, V_{k+1}$$

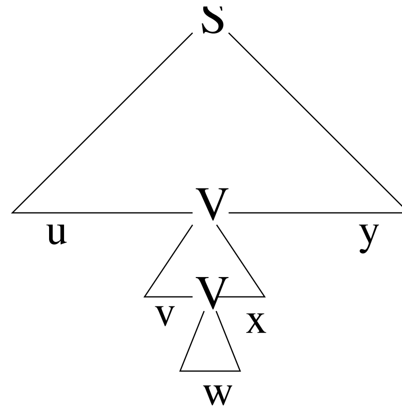
be the first $k+2$ labels along path π:



Since the grammar has only k variables, it follows from the pigeon hole principal that two of the variables V_1, V_2, $\ldots$, V_{k+1} must be identical, say

$$V_s = V_t = V, \text{ for } s < t.$$

So the derivation tree for z looks like:

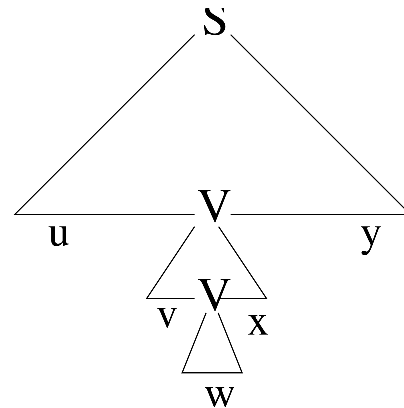


We have valid derivations

$$\begin{aligned} S &\Rightarrow^* uVy \\ V &\Rightarrow^* vVx \\ V &\Rightarrow^* w. \end{aligned}$$

Clearly $vx \neq \varepsilon$: Since the grammar G is in CNF, the lower V must have a sibling, and the yield of that sibling is part of either v or x.

So the derivation tree for z looks like:



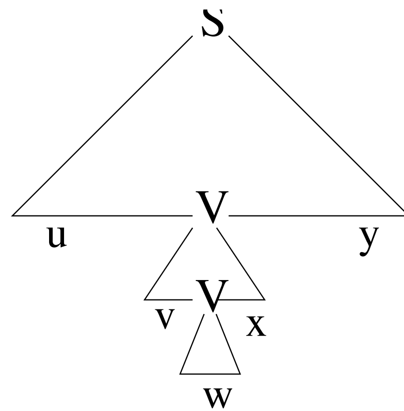
We have valid derivations

$$\begin{aligned} S &\Rightarrow^* uVy \\ V &\Rightarrow^* vVx \\ V &\Rightarrow^* w. \end{aligned}$$

Clearly $vx \neq \varepsilon$: Since the grammar G is in CNF, the lower V must have a sibling, and the yield of that sibling is part of either v or x.

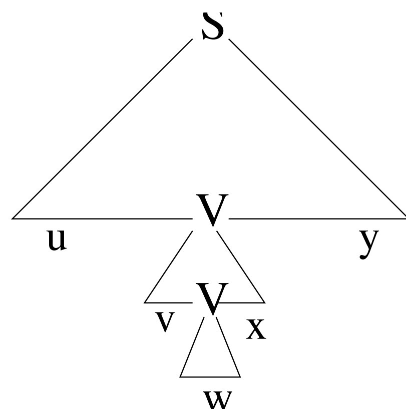
Now, there are derivations for words uv^iwx^iy for all $i \geq 0$:

$$S \Rightarrow^* uVy \Rightarrow^* uvVxy \Rightarrow^* uv^2Vx^2y \Rightarrow^* \ldots$$
$$\cdots \Rightarrow^* uv^iVx^iy \Rightarrow^* uv^iwx^iy$$



To complete the proof we still have to show that

$$|vwx| \leq n = 2^k.$$



To complete the proof we still have to show that

$$|vwx| \leq n = 2^k.$$

But vwx it is the yield of the subtree rooted at V_t (=the higher V), and the depth of that subtree is at most $k+1$.

(If there would be longer path from V_t to a leaf, then that path combined with the path from the root S to V_t would be longer than the longest path π, a contradiction.)

We know how long can the yields of trees of depth $\leq k+1$ be: they can have at most 2^k letters.

We use the pumping lemma to prove that certain languages are **not context-free**. It works the same way as the pumping lemma for regular languages. To show that a language is not a CFL we show that it does not satisfy the pumping lemma. So we are more interested in the negation of the pumping lemma.

Here is the mathematical formulation for the pumping lemma:

$$
\begin{aligned}
&(\exists n)\\
&\ (\forall z \in L, |z| \geq n)\\
&\ \ (\exists u, v, w, x, y : z = uvwxy, |vwx| \leq n, vx \neq \varepsilon)\\
&\ \ \ (\forall i \geq 0)\\
&\ \ \ \ uv^i wx^i y \in L,
\end{aligned}
$$

Its negation is the statement:

$$
\begin{aligned}
&(\forall n)\\
&\ (\exists z \in L, |z| \geq n)\\
&\ \ (\forall u, v, w, x, y : z = uvwxy, |vwx| \leq n, vx \neq \varepsilon)\\
&\ \ \ (\exists i \geq 0)\\
&\ \ \ \ uv^i wx^i y \notin L.
\end{aligned}
$$

Here is the mathematical formulation for the pumping lemma:

$$(\exists n)$$
$$(\forall z \in L, |z| \geq n)$$
$$(\exists u, v, w, x, y : z = uvwxy, |vwx| \leq n, vx \neq \varepsilon)$$
$$(\forall i \geq 0)$$
$$uv^iwx^iy \in L,$$

Its negation is the statement:

$$(\forall n)$$
$$(\exists z \in L, |z| \geq n)$$
$$(\forall u, v, w, x, y : z = uvwxy, |vwx| \leq n, vx \neq \varepsilon)$$
$$(\exists i \geq 0)$$
$$uv^iwx^iy \notin L.$$

To prove that a language is not context-free we do the following two steps:

(1) For every n select a word $z \in L$, $|z| \geq n$,

(2) for any division $z = uvwxy$ of z into five segments satisfying $|vwx| \leq n$ and $vx \neq \varepsilon$, find a number $i \geq 0$ such that

$$uv^iwx^iy$$

is **not** in the language L.

Example. Consider the language

$$L = \{a^m b^m c^m \mid m \geq 1\}.$$

Let us prove that L is **not** context-free.

Example. Consider the language

$$L = \{a^m b^m c^m \mid m \geq 1\}.$$

Let us prove that L is **not** context-free.

(1) For any given n choose word $z =$

Example. Consider the language

$$L = \{a^m b^m c^m \mid m \geq 1\}.$$

Let us prove that L is **not** context-free.

(1) For any given n choose word $z = a^n b^n c^n$. We have $z \in L$ and $|z| \geq n$, as required.

Example. Consider the language

$$L = \{a^m b^m c^m \mid m \geq 1\}.$$

Let us prove that L is **not** context-free.

(1) For any given n choose word $z = a^n b^n c^n$. We have $z \in L$ and $|z| \geq n$, as required.

(2) Let $z = uvwxy$ be an arbitrary division of z into 5 segments such that

$$|vwx| \leq n \text{ and } vx \neq \varepsilon.$$

We must analyze possible choices of u, v, w, x, y: Since $|vwx| \leq n$, words v and x cannot contain both letters a and c. (In z, letters a are separated from all c's by a string of n b's.)

Example. Consider the language

$$L = \{a^m b^m c^m \mid m \geq 1\}.$$

Let us prove that L is **not** context-free.

(1) For any given n choose word $z = a^n b^n c^n$. We have $z \in L$ and $|z| \geq n$, as required.

(2) Let $z = uvwxy$ be an arbitrary division of z into 5 segments such that

$$|vwx| \leq n \quad \text{and} \quad vx \neq \varepsilon.$$

We must analyze possible choices of u, v, w, x, y: Since $|vwx| \leq n$, words v and x cannot contain both letters a and c. (In z, letters a are separated from all c's by a string of n b's.)

So letter a or c does not exist in v and x. Therefore uv^2wx^2y contains more some letters than others: By adding one v and x we have increased the number of some letters, while the number of one letter has remained n.

Conclusion: Choice $i = 2$ gives

$$uv^i wx^i y \notin L.$$

Another example.

$$L = \left\{ a^m b^k c^m d^k \mid m, k \geq 1 \right\}$$

Closure properties of CFL

Recall: We say that the family of context-free languages is **closed under the language operation** Op if

$$L_1, L_2, \ldots \text{ are CFL } \implies Op(L_1, L_2, \ldots) \text{ is CFL}$$

That is, if operation Op is applied to context-free languages, the result is also context-free.

Closure properties of CFL

Recall: We say that the family of context-free languages is **closed under the language operation** Op if

$$L_1, L_2, \dots \text{ are CFL} \implies Op(L_1, L_2, \dots) \text{ is CFL}$$

That is, if operation Op is applied to context-free languages, the result is also context-free.

We say the **closure is effective** if there is a mechanical procedure (=algorithm) that constructs result $Op(L_1, L_2, \dots)$ for any context-free input languages L_1, L_2, $\dots$.

Inputs and outputs are given in the form of PDA or context-free grammar – it does not matter which format is used since both devices can be mechanically converted into each other.

Theorems. The family of CFL is effectively closed under the following operations:

- Union (if L_1, L_2 are CFL, so is $L_1 \cup L_2$),
- Concatenation (if L_1, L_2 are CFL, so is $L_1 L_2$),
- Kleene closure (if L is CFL, so is L^*),
- Substitutions with CFL (if L is CFL, and f is a substitution such that for every letter $a \in \Sigma$ language $f(a)$ is CFL, then $f(L)$ is CFL),
- Homomorphisms (if L is CFL, and h is a homomorphism then $h(L)$ is CFL),
- Inverse homomorphisms (if L is CFL, and h is a homomorphism then $h^{-1}(L)$ is CFL),
- Intersections with **regular** languages (if L is CFL and R is a regular language, then $L \cap R$ is CFL).

More theorems. The family of CFL is **NOT** closed under the following operations:

- Intersection (for some CFL L_1 and L_2 the language $L_1 \cap L_2$ is **not** CFL),
- Complementation (for some CFL L the language $\bar{L}$ is **not** CFL),
- Quotient (for some CFL L_1 and L_2 the language L_1/L_2 is **not** CFL).

Proofs.

1. **Union and concatenation**. Let L_1 be generated by grammar

$$G_1 = (V_1, T_1, P_1, S_1),$$

and L_2 by

$$G_2 = (V_2, T_2, P_2, S_2).$$

We may assume that the two variable sets V_1 and V_2 are disjoint, i.e. $V_1 \cap V_2 = \emptyset$. (If necessary we just rename some variables.)

Proofs.

1. **Union and concatenation**. Let L_1 be generated by grammar

$$G_1 = (V_1, T_1, P_1, S_1),$$

and L_2 by

$$G_2 = (V_2, T_2, P_2, S_2).$$

We may assume that the two variable sets V_1 and V_2 are disjoint, i.e. $V_1 \cap V_2 = \emptyset$. (If necessary we just rename some variables.)

Union $L_1 \cup L_2$ is generated by grammar

$$G_3 = (V_1 \cup V_2 \cup \{S_3\}, T_1 \cup T_2, P_3, S_3)$$

where S_3 is a new start symbol, and P_3 contains all productions in P_1 and P_2, and additional "initializing" productions

$$S_3 \longrightarrow S_1 \mid S_2.$$

The first derivation step by G_3 produces either S_1 or S_2, and after that a derivation by G_1 or G_2 is simulated.

Proofs.

1. **Union and concatenation**. Let L_1 be generated by grammar

$$G_1 = (V_1, T_1, P_1, S_1),$$

and L_2 by

$$G_2 = (V_2, T_2, P_2, S_2).$$

We may assume that the two variable sets V_1 and V_2 are disjoint, i.e. $V_1 \cap V_2 = \emptyset$. (If necessary we just rename some variables.)

Concatenation L_1L_2 is generated by grammar

$$G_4 = (V_1 \cup V_2 \cup \{S_4\}, T_1 \cup T_2, P_4, S_4)$$

where S_4 is a new start symbol, and P_4 contains all productions in P_1 and P_2, and additional initializing production

$$S_4 \longrightarrow \; S_1S_2.$$

After first derivation step the sentential form is S_1S_2, and from S_1 and S_2 we can derive words of L_1 and L_2, respectively.

2. **Kleene ***. Let L be generated by grammar

$$G = (V, T, P, S).$$

Then L^* is generated by grammar

$$G' = (V \cup \{S'\}, T, P', S')$$

where S' is a new start symbol, and P' contains all productions in P plus productions

$$S' \longrightarrow \; SS' \mid \varepsilon.$$

From S' one gan generate an arbitrarily long sequence of S's:

$$S' \Rightarrow SS' \Rightarrow \cdots \Rightarrow SS \ldots SS' \Rightarrow SS \ldots S,$$

and each S produces a word of L.

3. **Substitution**. Let $L \subseteq \Sigma^*$ be generated by grammar

$$G = (V, \Sigma, P, S),$$

and for each $a \in \Sigma$ let $f(a)$ be generated by grammar

$$G_a = (V_a, T_a, P_a, S_a).$$

Assume that all variable sets V and V_a are disjoint.

3. **Substitution**. Let $L \subseteq \Sigma^*$ be generated by grammar

$$G = (V, \Sigma, P, S),$$

and for each $a \in \Sigma$ let $f(a)$ be generated by grammar

$$G_a = (V_a, T_a, P_a, S_a).$$

Assume that all variable sets V and V_a are disjoint.

The following grammar generates $f(L)$:

$$G' = (V', T', P', S),$$

where

$$V' = V \cup \left(\bigcup_{a \in \Sigma} V_a \right)$$

contains all variables of all grammars G and G_a,

$$T' = \bigcup_{a \in \Sigma} T_a$$

contains all terminals of grammars G_a.

Productions P' include all productions that are in P_a's, plus for every production in P a production obtained by replacing all occurences of terminals a by the start symbol S_a of the corresponding grammar G_a.

Example. Let $L = L(G)$ be generated by

$$G = (\{X\}, \{a, b\}, P, X)$$

with productions

$$X \longrightarrow aXb \mid bXa \mid \varepsilon.$$

Let f be a substitution such that

- the language $f(a)$ is generated by

  $$G_a = (\{Y\}, \{0, 1\}, P_a, Y)$$

  with

  $$Y \longrightarrow 0YY0 \mid 1,$$

- the language $f(b)$ is generated by

  $$Z \longrightarrow 0Z1Z \mid \varepsilon.$$

Then $f(L)$ is generated by the grammar $G' = (V', T', P', S')$ where

Example. Let $L = L(G)$ be generated by

$$G = (\{X\}, \{a, b\}, P, X)$$

with productions

$$X \longrightarrow aXb \mid bXa \mid \varepsilon.$$

Let f be a substitution such that

- the language $f(a)$ is generated by $G_a = (\{Y\}, \{0, 1\}, P_a, Y)$ with productions
$$Y \longrightarrow 0YY0 \mid 1,$$
- the language $f(b)$ is generated by $G_b = (\{Z\}, \{0, 1\}, P_b, Z)$ with productions
$$Z \longrightarrow 0Z1Z \mid \varepsilon.$$

Then $f(L)$ is generated by the grammar $G' = (V', T', P', S')$ where

$$\begin{aligned} V' &= \\ T' &= \\ S' &= \\ P' &= \end{aligned}$$

4. **Homomorphism**. Follows from previous proof because homomorphism is a special type of substitution.

4. **Homomorphism**. Follows from previous proof because homomorphism is a special type of substitution.

We can also make a direct construction where we replace in the productions all occurrences of terminal letters by their homomorphic images.

Example. If L is generated by a grammar with productions

$$\begin{aligned} E &\longrightarrow E + E \mid N \\ N &\longrightarrow 0N \mid 1N \mid 0 \mid 1 \end{aligned}$$

and homomorphism h is given by $h(+) = \times$, $h(0) = ab$, $h(1) = ba$ then $h(L)$ is generated by

$$\begin{aligned} E &\longrightarrow \\ N &\longrightarrow \end{aligned}$$

4. **Homomorphism**. Follows from previous proof because homomorphism is a special type of substitution.

We can also make a direct construction where we replace in the productions all occurrences of terminal letters by their homomorphic images.

Example. If L is generated by a grammar with productions

$$\begin{array}{lcl} E & \longrightarrow & E + E \mid N \\ N & \longrightarrow & 0N \mid 1N \mid 0 \mid 1 \end{array}$$

and homomorphism h is given by $h(+) = \times$, $h(0) = ab$, $h(1) = ba$ then $h(L)$ is generated by

$$\begin{array}{lcl} E & \longrightarrow & E \times E \mid N \\ N & \longrightarrow & abN \mid baN \mid ab \mid ba \end{array}$$

5. **Inverse homomorphism**. In this case it is easier to use PDA instead of grammars. Let L be recognized by the PDA

$$M = (Q, \Delta, \Gamma, \delta, q_0, Z_0, F)$$

by final state, and let

$$h : \Sigma \longrightarrow \Delta^*$$

be a homomorphism. Let us construct a PDA M' that recognizes the language $h^{-1}(L)$ by final state, that is, it accepts all words w such that $h(w) \in L$.

5. **Inverse homomorphism**. In this case it is easier to use PDA instead of grammars. Let L be recognized by the PDA

$$M = (Q, \Delta, \Gamma, \delta, q_0, Z_0, F)$$

by final state, and let

$$h : \Sigma \longrightarrow \Delta^*$$

be a homomorphism. Let us construct a PDA M' that recognizes the language $h^{-1}(L)$ by final state, that is, it accepts all words w such that $h(w) \in L$.

Idea: On an input w, machine M' computes $h(w)$ and simulates M with input $h(w)$. If $h(w)$ is accepted by M then w is accepted by M'.

The result of h on w is stored on a buffer, or **"virtual input tape"**, inside the control unit. The simulation of M reads its input from this virtual tape. $h(w)$ is computed from w letter-by-letter as needed: as soon as the virtual tape becomes empty the next "real" input letter a is scanned, and $h(a)$ is added on the virtual tape.

In detail: Let

$$B = \{u \mid \ u \text{ is a suffix of some } h(a)\}.$$

Set B consists of all possible contents of the "virtual" input tape. Set B is of course a finite set.

Let $M = (Q, \Delta, \Gamma, \delta, q_0, Z_0, F)$ recognize L. For $h^{-1}(L)$ we construct the PDA

$$M' = (Q', \Sigma, \Gamma, \delta', q'_0, Z_0, F'),$$

where

$$\begin{aligned} Q' &= Q \times B, \\ q'_0 &= [q_0, \varepsilon], \text{ and} \\ F' &= F \times \{\varepsilon\}. \end{aligned}$$

δ' contains two types of productions:

- For every transition of the original machine M we have the simulating transition in M': If

$$(p, \gamma) \in \delta(q, a, Z)$$

  then

$$([p, x], \gamma) \in \delta'([q, ax], \varepsilon, Z)$$

  for every $ax \in B$. Here $a \in \Sigma \cup \{\varepsilon\}$. Notice that this is an ε-move: M' reads the input letter a from the virtual tape.

- When the virtual tape becomes empty, we have a transition for reading the next real input letter a, and loading its homomorphic image $h(a)$ to the virtual tape: For all $q \in Q$, $a \in \Sigma$, and $Z \in \Gamma$ we have the transition

$$([q, h(a)], Z) \in \delta'([q, \varepsilon], a, Z).$$

Initially the virtual tape is empty, so the initial state of M' is

$$[q_0, \varepsilon].$$

The input word is accepted if in the end the virtual input tape is empty (M has consumed the whole input word), and the state of M is a final state. Therefore the final states of M' are elements of

$$F \times \{\varepsilon\}.$$

Example. Consider the PDA

$$M = (\{q\}, \{a, b\}, \{A\}, \delta, q, A, \{q\})$$

where

$$\begin{aligned} \delta(q, a, A) &= \{(q, AA)\} \\ \delta(q, b, A) &= \{(q, \varepsilon)\} \end{aligned}$$

and let h be the homomorphism $h(0) = b$, $h(1) = aa$. Let us construct a PDA for $h^{-1}(L(M))$:

Possible contents of the virtual tape are $B =$

so the state set of the new PDA will be $Q' =$

The start state is $S' =$

and there is only one final state $F' =$

Example. Consider the PDA

$$M = (\{q\}, \{a, b\}, \{A\}, \delta, q, A, \{q\})$$

where

$$\begin{aligned} \delta(q, a, A) &= \{(q, AA)\} \\ \delta(q, b, A) &= \{(q, \varepsilon)\} \end{aligned}$$

and let h be the homomorphism $h(0) = b$, $h(1) = aa$. Let us construct a PDA for $h^{-1}(L(M))$:

Possible contents of the virtual tape are $B = \{b, \varepsilon, aa, a\}$

so the state set of the new PDA will be $Q' = \{[q, b], [q, \varepsilon], [q, aa], [q, a]\}$

The start state is $S' = [q, \varepsilon]$

and there is only one final state $F' = \{[q, \varepsilon]\}$

Example. Consider the PDA

$$M = (\{q\}, \{a, b\}, \{A\}, \delta, q, A, \{q\})$$

where

$$\begin{aligned} \delta(q, a, A) &= \{(q, AA)\} \\ \delta(q, b, A) &= \{(q, \varepsilon)\} \end{aligned}$$

and let h be the homomorphism $h(0) = b$, $h(1) = aa$. Let us construct a PDA for $h^{-1}(L(M))$:

Possible contents of the virtual tape are $B = \{b, \varepsilon, aa, a\}$

so the state set of the new PDA will be $Q' = \{[q, b], [q, \varepsilon], [q, aa], [q, a]\}$

The start state is $S' = [q, \varepsilon]$

and there is only one final state $F' = \{[q, \varepsilon]\}$

From the original transition $(q, AA) \in \delta(q, a, A)$ we get two simulating transitions, and from $(q, \varepsilon) \in \delta(q, b, A)$ one transition:

Finally, we have transitions for loading the virtual tape:

Example. Let us show that the language

$$L = \{a^n b^{2n} c^{3n} \mid n \geq 1\}$$

is not context-free.

6. **NOT closed under Intersection**. It is enough to find one counter example.

6. **NOT closed under Intersection**. It is enough to find one counter example. Let

$$L_1 = \{a^n b^n c^m \mid n, m \geq 1\}$$

and

$$L_2 = \{a^m b^n c^n \mid n, m \geq 1\}.$$

Both L_1 and L_2 are context-free: L_1 is concatenation of context-free languages

$$\{a^n b^n \mid n \geq 1\} \text{ and } c^+,$$

and L_2 is concatenation of

$$a^+ \text{ and } \{b^n c^n \mid n \geq 1\}.$$

(We may also directly construct grammars for both L_1 and L_2.)

But even though L_1 and L_2 are context-free, their intersection

$L_1 \cap L_2 =$

is not. So the family of CFL is not closed under intersection.

6. **NOT closed under Intersection**. It is enough to find one counter example. Let

$$L_1 = \{a^n b^n c^m \mid n, m \geq 1\}$$

and

$$L_2 = \{a^m b^n c^n \mid n, m \geq 1\}.$$

Both L_1 and L_2 are context-free: L_1 is concatenation of context-free languages

$$\{a^n b^n \mid n \geq 1\} \text{ and } c^+,$$

and L_2 is concatenation of

$$a^+ \text{ and } \{b^n c^n \mid n \geq 1\}.$$

(We may also directly construct grammars for both L_1 and L_2.)

But even though L_1 and L_2 are context-free, their intersection

$$L_1 \cap L_2 = \{a^n b^n c^n \mid n \geq 1\}$$

is not. So the family of CFL is not closed under intersection.

7. **NOT closed under complementation**. If they were, then the family of CFL would be closed under intersection as well:

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}},$$

a contradiction with 6 above.

8. **NOT closed under quotients (with context-free languages)**. Homework.

8. **NOT closed under quotients (with context-free languages)**. Homework.

An easy to prove variant: there exists a context-free language L and some (non-context-free) language K such that L/K is not context-free. Choose

$$L = \{a^n b^m c^k \# d^k e^m f^n \mid n, m, k \geq 1\},$$

$$K = \{\# d^n e^n f^n \mid n \geq 1\}.$$

Here L is context-free but

$L/K =$

is not context-free.

8. **NOT closed under quotients (with context-free languages)**. Homework.

An easy to prove variant: there exists a context-free language L and some (non-context-free) language K such that L/K is not context-free. Choose

$$L = \{a^n b^m c^k \# d^k e^m f^n \mid n, m, k \geq 1\},$$

$$K = \{\# d^n e^n f^n \mid n \geq 1\}.$$

Here L is context-free but

$$L/K = \{a^n b^n c^n \mid n \geq 1\}$$

is not context-free.

9. **Intersection with regular languages**. Even though the intersection of two context-free languages may be non-context-free, the intersection of any context-free language with a regular language is always context-free.

Proof.

9. **Intersection with regular languages**. Even though the intersection of two context-free languages may be non-context-free, the intersection of any context-free language with a regular language is always context-free.

Proof. Let L be recognized by the PDA

$$M = (Q_M, \Sigma, \Gamma, \delta_M, q_M, Z_0, F_M)$$

by final state, and let R be recognized by the DFA

$$A = (Q_A, \Sigma, \delta_A, q_A, F_A).$$

We construct a PDA M' that runs both M and A in parallel. The state set of M' is

$$Q' = Q_M \times Q_A.$$

The states are used to store the states of both M and A for the input that has been read so far.

9. **Intersection with regular languages**. Even though the intersection of two context-free languages may be non-context-free, the intersection of any context-free language with a regular language is always context-free.

Proof. Let L be recognized by the PDA

$$M = (Q_M, \Sigma, \Gamma, \delta_M, q_M, Z_0, F_M)$$

by final state, and let R be recognized by the DFA

$$A = (Q_A, \Sigma, \delta_A, q_A, F_A).$$

We construct a PDA M' that runs both M and A in parallel. The state set of M' is

$$Q' = Q_M \times Q_A.$$

The states are used to store the states of both M and A for the input that has been read so far.

All ϵ-transitions of M are simulated as such, without changing the Q_A-component of the state. But whenever M reads next input letter, the Q_A-component is changed according to DFA A. Clearly, if in the end of the input word both Q_M- and Q_A-components are final states, the word is accepted by both M and A.

$$\begin{aligned} M &= (Q_M, \Sigma, \Gamma, \delta_M, q_M, Z_0, F_M) \\ A &= (Q_A, \Sigma, \delta_A, q_A, F_A) \end{aligned}$$

More precisely, we construct the PDA

$$M' = (Q_M \times Q_A, \Sigma, \Gamma, \delta', [q_M, q_A], Z_0, F_M \times F_A)$$

where δ' is described below:

Each transition

$$(q', \gamma) \in \delta_M(q, a, Z)$$

by the original PDA, and each state $p \in Q_A$ of the DFA provide the transition

$$([q', \delta_A(p, a)], \gamma) \in \delta'([q, p], a, Z)$$

in the new machine M'.

(The first state component and the stack follow PDA M. The second state component simulates DFA A: On input a state p is changed to $\delta_A(p, a)$. Note that if $a = \varepsilon$ then $\delta_A(p, a) = p$.)

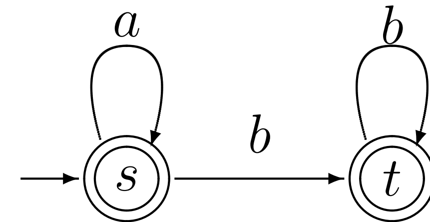
Example. Consider the PDA

$$M = (\{q\}, \{a,b\}, \{A\}, \delta, q, A, \{q\})$$

where

$$\begin{aligned}\delta(q,a,A) &= \{(q,AA)\} \\ \delta(q,b,A) &= \{(q,\varepsilon)\}\end{aligned}$$

and let $R = a^*b^*$. Then R is recognized by the DFA



The intersection $L(M) \cap R$ is recognized by the PDA

$$M' = (Q', \{a,b\}, \{A\}, \delta', q_0, A, F)$$

where

$$\begin{aligned}Q' &= \\ q_0 &= \\ F &= \end{aligned}$$

and δ' contains transitions

$$\begin{aligned}\delta'([q,s],a,A) &= \\ \delta'([q,s],b,A) &= \\ \delta'([q,t],a,A) &= \\ \delta'([q,t],b,A) &= \end{aligned}$$

Example. Let us prove that

$$L = \{0^n 1 0^n 1 0^n \mid n \geq 1\}$$

is not context-free.

Example. Let us prove that

$$L = \{0^n 1 0^n 1 0^n \mid n \geq 1\}$$

is not context-free.

Assume it is a CFL. Define a substitution $f(0) = \{a, b, c\}$, $f(1) = \{\#\}$. Since we assumed that L is context-free, so is

$L_1 = f(L) \cap (a^+ \# b^+ \# c^+) =$

Example. Let us prove that

$$L = \{0^n 1 0^n 1 0^n \mid n \geq 1\}$$

is not context-free.

Assume it is a CFL. Define a substitution $f(0) = \{a, b, c\}$, $f(1) = \{\#\}$. Since we assumed that L is context-free, so is

$L_1 = f(L) \cap (a^+ \# b^+ \# c^+) = \{a^n \# b^n \# c^n \mid n \geq 1\}$

Example. Let us prove that

$$L = \{0^n 1 0^n 1 0^n \mid n \geq 1\}$$

is not context-free.

Assume it is a CFL. Define a substitution $f(0) = \{a, b, c\}$, $f(1) = \{\#\}$. Since we assumed that L is context-free, so is

$L_1 = f(L) \cap (a^+ \# b^+ \# c^+) = \{a^n \# b^n \# c^n \mid n \geq 1\}$

Define then a homomorphism $h(a) = a$, $h(b) = b$, $h(c) = c$, $h(\#) = \varepsilon$. Then the language

$L_2 = h(L_1) =$

is context-free.

Example. Let us prove that

$$L = \{0^n 1 0^n 1 0^n \mid n \geq 1\}$$

is not context-free.

Assume it is a CFL. Define a substitution $f(0) = \{a, b, c\}$, $f(1) = \{\#\}$. Since we assumed that L is context-free, so is

$$L_1 = f(L) \cap (a^+ \# b^+ \# c^+) = \{a^n \# b^n \# c^n \mid n \geq 1\}$$

Define then a homomorphism $h(a) = a$, $h(b) = b$, $h(c) = c$, $h(\#) = \varepsilon$. Then the language

$$L_2 = h(L_1) = \{a^n b^n c^n \mid n \geq 1\}$$

is context-free.

But we know that L_2 is not context-free, a contradiction. Our initial assumption that L is a CFL, must be incorrect.

Example. Let us show that the language

$$L = \{ww \mid w \in \{a, b\}^*\}$$

is not context-free.

Example. Let us show that the language

$$L = \{ww \mid w \in \{a, b\}^*\}$$

is not context-free.

If it were CFL, so would be

$L_1 = L \cap (a^+b^+a^+b^+) =$

Example. Let us show that the language

$$L = \{ww \mid w \in \{a, b\}^*\}$$

is not context-free.

If it were CFL, so would be

$$L_1 = L \cap (a^+b^+a^+b^+) = \{a^m b^k a^m b^k \mid m, k \geq 1\}$$

Example. Let us show that the language

$$L = \{ww \mid w \in \{a,b\}^*\}$$

is not context-free.

If it were CFL, so would be

$L_1 = L \cap (a^+b^+a^+b^+) = \{a^m b^k a^m b^k \mid m, k \geq 1\}$

Define then substitution $f(a) = \{a, c\}$, $f(b) = \{b, d\}$. If L_1 is a CFL so would be

$L_2 = f(L_1) \cap (a^+b^+c^+d^+) =$

Example. Let us show that the language

$$L = \{ww \mid w \in \{a,b\}^*\}$$

is not context-free.

If it were CFL, so would be

$$L_1 = L \cap (a^+b^+a^+b^+) = \{a^m b^k a^m b^k \mid m,k \geq 1\}$$

Define then substitution $f(a) = \{a,c\}$, $f(b) = \{b,d\}$. If L_1 is a CFL so would be

$$L_2 = f(L_1) \cap (a^+b^+c^+d^+) = \{a^m b^k c^m d^k \mid m,k \geq 1\}$$

But we have proved earlier using the pumping lemma that L_2 is not context-free, so the original language L is not context-free either.

Example. Let us show that family of context-free languages is closed under quotient with regular languages. Let L be a context-free language, and let R be a regular language.

Example. Let us show that family of context-free languages is closed under quotient with regular languages. Let L be a context-free language, and let R be a regular language.

Let Σ be the union alphabets of L and R, and let

$$\Sigma' = \{a' \mid a \in \Sigma\}$$

be a new alphabet obtained by marking symbols of Σ.

We have shown previously that the quotient L/R can be implemented as

$$L/R = g(s(L) \cap \Sigma^* h(R))$$

where

- homomorphism h marks all letters of a word,
- substitution s may mark any letters of a word,
- homomorphism g erases all marked letters.

All operations here preserve context-freeness of L, so that L/R is context-free. (Note that $\Sigma^* h(R)$ is a regular language.)

Decision algorithms

Next we discuss algorithms for deciding if a given CFL is

- empty,
- finite,
- infinite.

We also have an algorithm for deciding if a given word belongs to a given CFL.

Decision algorithms

Next we discuss algorithms for deciding if a given CFL is

- empty,
- finite,
- infinite.

We also have an algorithm for deciding if a given word belongs to a given CFL.

Later on, we'll see that there are many questions concerning CFL that are **undecidable**. For example, there do not exist any algorithms for deciding if a given CFG is ambiguous, if given two CFG's are equivalent (=define the same language), if a given CFG generates Σ^*, if the intersection of two CFL's is empty, etc.

Decision algorithms

Next we discuss algorithms for deciding if a given CFL is

- empty,
- finite,
- infinite.

We also have an algorithm for deciding if a given word belongs to a given CFL.

Later on, we'll see that there are many questions concerning CFL that are **undecidable**. For example, there do not exist any algorithms for deciding if a given CFG is ambiguous, if given two CFG's are equivalent (=define the same language), if a given CFG generates Σ^*, if the intersection of two CFL's is empty, etc.

A CFL can be represented in different ways: as a **grammar** or as a **pushdown automaton** that recognizes the language. It does not matter which representation we use since we have algorithms for converting a CFG into equivalent PDA and vice versa.

Theorem. There are algorithms to determine if a given CFL is (a) empty, (b) finite (contains finitely many words), (c) infinite (contains infinitely many words).

Theorem. There are algorithms to determine if a given CFL is (a) empty, (b) finite (contains finitely many words), (c) infinite (contains infinitely many words).

Proof. (a) We know how to simplify a given grammar by removing all variables that do not derive a terminal word. (We used this algorithm to simplify grammars.) Language $L(G)$ is empty if and only if start symbol S gets removed, $i.e.$, if and only if the grammar does not generate any terminal strings.

Theorem. There are algorithms to determine if a given CFL is (a) empty, (b) finite (contains finitely many words), (c) infinite (contains infinitely many words).

Proof. (a) We know how to simplify a given grammar by removing all variables that do not derive a terminal word. (We used this algorithm to simplify grammars.) Language $L(G)$ is empty if and only if start symbol S gets removed, $i.e.$, if and only if the grammar does not generate any terminal strings.

Recall the algorithm: We mark variables to detect if S is removable.

`Empty(`G`)`

```
1.  Initially unmark every variable A
2.  Repeat
3.     For every production A ⟶ α do
4.        If every variable appearing in α is marked
5.           then mark A
6.  Until no new variables were marked on line 5
7.  If start symbol S is marked then return FALSE
8.     else return TRUE
```

Proof. (b) and (c). To test whether a given grammar generates finitely or infinitely many words, we simplify the grammar, and convert it into Chomsky normal form. After this we only have productions of types

$$\begin{aligned} A &\longrightarrow BC, \\ A &\longrightarrow a, \end{aligned}$$

and we do not have useless symbols, *i.e.*, symbols that do not derive terminal words and/or are not reachable from the start symbol.

(We may loose ε in the conversion but it does not matter since L is finite if and only if $L - \{\varepsilon\}$ is finite.)

Proof. (b) and (c). To test whether a given grammar generates finitely or infinitely many words, we simplify the grammar, and convert it into Chomsky normal form. After this we only have productions of types

$$\begin{aligned} A &\longrightarrow BC, \\ A &\longrightarrow a, \end{aligned}$$

and we do not have useless symbols, *i.e.*, symbols that do not derive terminal words and/or are not reachable from the start symbol.

(We may loose ε in the conversion but it does not matter since L is finite if and only if $L - \{\varepsilon\}$ is finite.)

So we can assume the given grammar G is in CNF, and all variables are useful. We use the fact that grammar G generates an infinite language if and only if some variable can derive a word containing the variable itself. Let us call a variable A **self-embedded** if

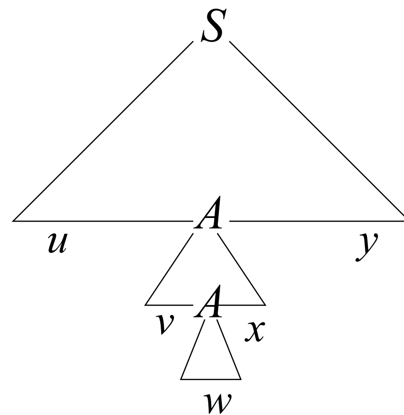
$$A \Rightarrow^{+} \alpha A \beta$$

for some words α and β. Since the grammar is in CNF, α and β are not both empty words.

Let us prove that $L = L(G)$ is infinite if and only if G contains a self-embedded variable $A \Rightarrow^+ \alpha A \beta$:

$\Longrightarrow$ Assume that L is infinite. Let k be the number of variables in G and let $n = 2^k$. Because L is infinite, there exists a word $z \in L$ such that $|z| \geq n$. As in the proof of the pumping lemma, $z = uvwxy$ and there are derivations

$$\begin{aligned} S &\Rightarrow^* uAy \\ A &\Rightarrow^+ vAx \\ A &\Rightarrow^* w \end{aligned}$$



The subderivation $A \Rightarrow^+ vAx$ means that the variable A is self-embedded.

Let us prove that $L = L(G)$ is infinite if and only if G contains a self-embedded variable $A \Rightarrow^+ \alpha A \beta$:

$\Longleftarrow$ Assume that $A \Rightarrow^+ \alpha A \beta$ is self-embedded. Since the grammar does not contain useless symbols, the variable A is reachable from S: there exist γ, ν such that $S \Longrightarrow^* \gamma A \nu$.

Again, since the grammar does not contain useless symbols every word of terminals and variables derives some terminal word: There are terminal words u, v, w, x, y such that

$$\gamma \Rightarrow^* u, \qquad \alpha \Rightarrow^* v, \qquad A \Rightarrow^* w, \qquad \beta \Rightarrow^* x, \qquad \nu \Rightarrow^* y.$$

Then we have derivations

$$\begin{array}{lll} S & \Rightarrow^* & uAy \\ A & \Rightarrow^+ & vAx \\ A & \Rightarrow^* & w \end{array}$$

Note that $vx \neq \emptyset$ because $A \Rightarrow^+ vAx$ uses at least one derivation step. It follows that $L(G)$ is infinite because for all i

$$S \Rightarrow^+ uv^i w x^i y$$

The algorithm: To decide if $L(G)$ is infinite we have to find out whether any variable is self-embedded. We can use marking procedure to determine if variable A is self-embedded:

Self-embedded(G, A)
1. Initially unmark every variable X
2. For every production $A \longrightarrow \alpha$ of A do
3. mark every variable X that appears in α
4. Repeat
5. For every production $X \longrightarrow \alpha$ in G do
6. If X is marked then mark every variable of α
7. Until no new variables were marked on line 6
8. If A is marked then return TRUE
9. else return FALSE

Then it is a simple matter to determine if G is finite or infinite:

Infinite(G)
1. Construct CNF grammar G' without useless variables such that $L(G') = L(G) \setminus \{\varepsilon\}$
2. For every variable A of G' do
3. If Self-embedded(G', A) then return TRUE
4. return FALSE

The process of determining if a grammar contains self-embedded variables can be visualized using a directed graph indicating which variables derive each other.

We draw a directed graph whose nodes are the variables, and there is an arrow from A to B if the grammar has production $A \longrightarrow \alpha$ for some α containing variable B.

Clearly variable A is self-embedded if and only if it is on a loop in the directed graph. Therefore: $L(G)$ is infinite if and only if there is a loop in the directed graph.

Example. Consider the CNF grammar G_1 with productions

$$\begin{aligned} S &\longrightarrow BC \mid BB \mid a \\ B &\longrightarrow CC \mid b \\ C &\longrightarrow a \end{aligned}$$

It does not contain useless symbols. The corresponding directed graph has three nodes, corresponding to variables S, B and C.

The graph does not contain a loop, so the language generated by G_1 is finite.

Example. Consider the CNF grammar G_2 with productions

$$\begin{aligned} S &\longrightarrow BC \mid BB \mid a \\ B &\longrightarrow CC \mid b \\ C &\longrightarrow SC \mid a \end{aligned}$$

It does not contain useless symbols. The corresponding directed graph contains a loop so the variables on the loop are self-embedded, and $L(G_2)$ is infinite. For example,

$$S \Rightarrow^4 bSa$$

so

$$S \Rightarrow^* b^i S a^i \Rightarrow b^i a a^i$$

for every $i \geq 1$.

Theorem. There is an algorithm for determining whether a given word w is in a given CFL L.

Proof. A simple but slow algorithm is provided by CNF grammars. We first construct a CNF grammar G for language L. To determine if $w \in L$ try all possible derivations that use $2n - 1$ derivation steps, where n is the length of the word w. We have $w \in L$ if and only if one of these derivations generate w.

Namely: as proved in the homework, in CNF grammars deriving a word of length n takes exactly $2n - 1$ steps.

Theorem. There is an algorithm for determining whether a given word w is in a given CFL L.

Proof. A simple but slow algorithm is provided by CNF grammars. We first construct a CNF grammar G for language L. To determine if $w \in L$ try all possible derivations that use $2n-1$ derivation steps, where n is the length of the word w. We have $w \in L$ if and only if one of these derivations generate w.

Namely: as proved in the homework, in CNF grammars deriving a word of length n takes exactly $2n-1$ steps.

(The case $w = \varepsilon$ has to be handled separately, since we loose ε when we convert the grammar into CNF. To test whether $\varepsilon \in L$ we can check whether the start symbol S is nullable in the original grammar.)

The algorithm is very **inefficient**: it may require an amount of time which is exponential with respect to the length of w.

In this course we are usually only interested in the existence of algorithms — not their efficiency. However, since parsing context-free grammars is an important problem with applications in compiler design, let us develop a faster algorithm.

CYK-algorithm for the membership problem of CFL: Again we start with a grammar G that is in the Chomsky normal form. Let

$$w = x_1 x_2 x_3 \ldots x_n$$

be the word for which we want to find out whether $w \in L$. Here, x_i are letters.

The CYK uses **dynamic programming** to find for each subword of w all variables of the grammar that derive that subword.

Example. Consider the grammar

$$\begin{aligned} S &\longrightarrow XY \\ X &\longrightarrow XZ \mid a \mid b \\ Y &\longrightarrow ZY \mid a \\ Z &\longrightarrow a \end{aligned}$$

and the word $w = babaa$.

We start with shorter subwords of w and work our way up to longer subwords.

Example. Consider the grammar

$$\begin{array}{rcl} S & \longrightarrow & XY \\ X & \longrightarrow & XZ \mid a \mid b \\ Y & \longrightarrow & ZY \mid a \\ Z & \longrightarrow & a \end{array}$$

and the word $w = babaa$.

We start with shorter subwords of w and work our way up to longer subwords.

Initially, we determine for each subword of **length 1** which variables derive it: A variable A derives word x_i if and only if $A \longrightarrow x_i$ is a production in P. In our example we get the following lists of variables:

$$\begin{array}{rcl} b & : & X \\ a & : & X, Y, Z \end{array}$$

Example. Consider the grammar

$$
\begin{aligned}
S &\longrightarrow XY \\
X &\longrightarrow XZ \mid a \mid b \\
Y &\longrightarrow ZY \mid a \\
Z &\longrightarrow a
\end{aligned}
$$

and the word $w = babaa$.

$b \; : \; X$

$a \; : \; X, Y, Z$

Next we move to subwords of **length 2**.

Example. Consider the grammar

$$\begin{aligned} S &\longrightarrow XY \\ X &\longrightarrow XZ \mid a \mid b \\ Y &\longrightarrow ZY \mid a \\ Z &\longrightarrow a \end{aligned}$$

and the word $w = babaa$.

b : X
a : X, Y, Z

Next we move to subwords of **length 2**.

Word $x_i x_{i+1}$ can be derived from a variable A if and only if there is a production $A \longrightarrow BC$ and B and C derive x_i and x_{i+1}, respectively. In our example we have

$$\begin{aligned} ba &: S, X \\ ab &: \\ aa &: S, X, Y \end{aligned}$$

Remark: We do not need to find variables that derive bb since bb is not a subword of w.

Example. Consider the grammar

$$\begin{array}{rcl} S & \longrightarrow & XY \\ X & \longrightarrow & XZ \mid a \mid b \\ Y & \longrightarrow & ZY \mid a \\ Z & \longrightarrow & a \end{array}$$

and the word $w = babaa$.

$$\begin{array}{lcl} b & : & X \\ a & : & X, Y, Z \\ ba & : & S, X \\ ab & : & \\ aa & : & S, X, Y \end{array}$$

We move up to **length 3**. Word $x_i x_{i+1} x_{i+2}$ can be derived from variable A if and only if there are variables B and C such that $A \longrightarrow BC$, and either B derives $x_i x_{i+1}$ and C derives x_{i+2} OR B derives x_i and C derives $x_{i+1} x_{i+2}$.

In our example:

$$\begin{array}{lcl} bab & : & \\ aba & : & \\ baa & : & S, X \end{array}$$

We move up to longer and longer subwords. To find out which variables derive subword

$$u = x_i x_{i+1} \dots x_{i+k}$$

of length $k+1$, we try all possible ways of dividing the subword into two non-empty parts:

$$\begin{aligned} u_1 &= x_i x_{i+1} \dots x_{i+s} \\ u_2 &= x_{i+s+1} x_{i+s+2} \dots x_{i+k} \end{aligned}$$

We find all pairs of variables (B, C) that derive the two parts:

$$\begin{aligned} B &\longrightarrow u_1 \\ C &\longrightarrow u_2 \end{aligned}$$

(We have found all such variables on the earlier rounds, since u_1 and u_2 are shorter words.) Finally we find all variables A such that

$$A \longrightarrow BC$$

and B and C derive u_1 and u_2, respectively.

Example. Consider the grammar

$$\begin{aligned} S &\longrightarrow XY \\ X &\longrightarrow XZ \mid a \mid b \\ Y &\longrightarrow ZY \mid a \\ Z &\longrightarrow a \end{aligned}$$

and the word $w = babaa$.

In our example, we move up to subwords of **length 4**. Consider first the subword $baba$. It can be factored into two parts in three different ways

$$\begin{array}{ll} b & aba \\ ba & ba \qquad\qquad SS, SX, XS, XX \\ bab & a \end{array}$$

But SS, SX, XS or XX does not appear on the right hand side of any production, so $baba$ is not derived by any variable.

Example. Consider the grammar

$$\begin{aligned} S &\longrightarrow XY \\ X &\longrightarrow XZ \mid a \mid b \\ Y &\longrightarrow ZY \mid a \\ Z &\longrightarrow a \end{aligned}$$

and the word $w = babaa$.

In our example, we move up to subwords of **length 4**. Consider first subword *baba*. It can be factored into two parts in three different ways

$$\begin{array}{llc} b & aba & \\ ba & ba & SS, SX, XS, XX \\ bab & a & \end{array}$$

But SS, SX, XS or XX does not appear on the right hand side of any production, so *baba* is not derived by any variable.

Similarily we see that the other subword *abaa* of length 4 cannot be derived from any variable:

$$\begin{array}{llc} a & baa & XS, YS, ZS, XX, YX, ZX \\ ab & aa & \\ aba & a & \end{array}$$

and XS, YS, ZS, XX, YX or ZX is not the right hand side of any production.

Example. Consider the grammar

$$\begin{aligned} S &\longrightarrow XY \\ X &\longrightarrow XZ \mid a \mid b \\ Y &\longrightarrow ZY \mid a \\ Z &\longrightarrow a \end{aligned}$$

and the word $w = babaa$.

Finally we get to subwords of **length 5**, *i.e.*, the word w itself:

$$\begin{array}{lrl} b & abaa & \\ ba & baa & SS, SX, XS, XX \\ bab & aa & \\ baba & a & \end{array}$$

But SS, SX, XS or XX is not the right hand side of any production, so no variables derive *babaa*, not even the start symbol. Therefore the grammar does not generate the word *babaa*.

Below is a pseudocode for the CYK-algorithm. It uses for each variable A a boolean array $A_{i,j}$ to indicate whether variable A derives the subword

$$w_i w_{i+1} \ldots w_{i+j-1}$$

of length j starting at the i'th letter of the input word w.

CYK(G, w)

```
1.   Let G be in Chomsky normal form, let n = |w|
2.   For every variable A and i,j ≤ n do
3.       Initialize A_{i,j} ⟵ FALSE
4.   For i ⟵ 1 to n do
5.       For all productions A ⟶ a do
6.           If a = w_i then A_{i,1} ⟵ TRUE
7.   For j ⟵ 2 to n do
8.       For i ⟵ 1 to n − j + 1 do
9.           For k ⟵ 1 to j − 1 do
10.              For all productions A ⟶ BC do
11.                  If B_{i,k} and C_{i+k,j−k} then A_{i,j} ⟵ TRUE
12.  Return S_{1,n}.
```

The time complexity of this algorithm for a **fixed CFL** L is $O(n^3)$, where n is the length of the input word.

Example. Let the grammar G be

$$\begin{aligned} S &\longrightarrow AB \mid BC \\ A &\longrightarrow BA \mid a \\ B &\longrightarrow CC \mid b \\ C &\longrightarrow AB \mid a \end{aligned}$$

and we want to find out whether word

$$w = baaba$$

is in the language $L(G)$.