**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 $\Sigma$ 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 $\Sigma$.

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 $\Sigma^*$, 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 $\Sigma^*$, 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

$$A \longrightarrow BC,$$
$$A \longrightarrow a,$$

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 $\varepsilon$ 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

$$A \longrightarrow BC,$$
$$A \longrightarrow a,$$

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 $\varepsilon$ 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

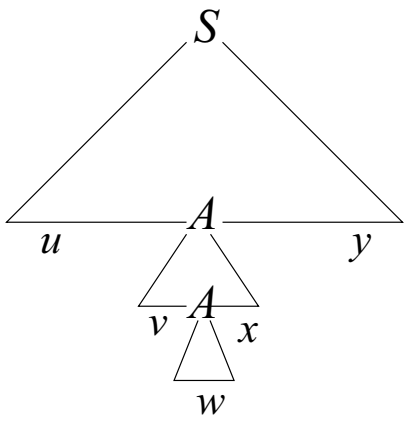
$$A \Rightarrow^+ \alpha A \beta$$

for some words $\alpha$ and $\beta$. Since the grammar is in CNF, $\alpha$ and $\beta$ 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 $\gamma, \nu$ 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

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

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 $i$

$$S \Rightarrow^+ uv^i wx^i y$$

**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 ⟶ α of A do
3.     mark every variable X that appears in α
4.  Repeat
5.     For every production X ⟶ α 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) \ {ε}
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 $\alpha$ 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: 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: 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 $\varepsilon$ 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.)

Here's another slow approach. The following algorithm determines if a given variable $A$ derives a given non-empty word $a_1 \dots a_n$ in a given CNF normal form grammar:

```
Derives(G, A, a_1 a_2 ... a_n)
1.  If n = 1 then if A ⟶ a_1 in G then return TRUE else return FALSE
2.  For i ⟵ 2 to n do
3.     For all productions A ⟶ BC in G do
4.        If Derives(G, B, a_1 ... a_{i-1}) and Derives(G, C, a_i ... a_n) then return TRUE
5.  return FALSE
```

The call `Derives(`$G, S, w$`)` with the start symbol $S$ returns `TRUE` if and only if $w$ is in $L$.

```
Derives(G, A, a_1 a_2 ... a_n)
1.  If n = 1 then if A ⟶ a_1 in G then return TRUE else return FALSE
2.  For i ⟵ 2 to n do
3.      For all productions A ⟶ BC in G do
4.          If Derives(G, B, a_1 ... a_{i-1}) and Derives(G, C, a_i ... a_n) then return TRUE
5.  return FALSE
```

The algorithm is very **inefficient**: it may require an amount of time which is exponential with respect to the length of $w$.

This is because during the recursion `Derives` may be called several times with the same inputs, this repeating the same computations over and over again.

In such situations **dynamic programming** may lead to a faster algorithm.

(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** is the dynamic programming version of the previous algorithm.

Again we start with a grammar $G$ that is in the Chomsky normal form. Let

$$w = a_1 a_2 a_3 \dots a_n$$

be the word for which we want to find out whether $w \in L$. Here, $a_i$ are letters.

The CYK uses **dynamic programming** to find for each subword $u$ of $w$ all variables of the grammar that derive $u$. We do this in the order of **increasing length** of $u$.

**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

$$\begin{aligned} S &\longrightarrow XY \\ X &\longrightarrow XZ \mid a \mid b \\ Y &\longrightarrow ZY \mid a \\ Z &\longrightarrow a \end{aligned}$$

and the word $w = babaa$.

We start with shorter subwords of $w$ and work our way up to longer subwords.

Initially, we determine for each subword of **length 1** which variables derive it: A variable $A$ derives word $a_i$ if and only if $A \longrightarrow a_i$ is a production in $P$. In our example we get the following lists of variables:

$$\begin{aligned} b &: X \\ a &: X, Y, Z \end{aligned}$$

**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 $w = babaa$.

$$\begin{aligned} b &: X \\ a &: X, Y, Z \end{aligned}$$

Next we move to subwords of **length 2**.

Word $a_i a_{i+1}$ can be derived from a variable $A$ if and only if there is a production $A \longrightarrow BC$ and $B$ and $C$ derive $a_i$ and $a_{i+1}$, 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

$$\begin{aligned} S &\longrightarrow XY \\ X &\longrightarrow XZ \mid a \mid b \\ Y &\longrightarrow ZY \mid a \\ Z &\longrightarrow a \end{aligned}$$

and the word $w = babaa$.

$$\begin{aligned} b &: X \\ a &: X, Y, Z \\ ba &: S, X \\ ab &: \\ aa &: S, X, Y \end{aligned}$$

We move up to **length 3**. Word $a_i a_{i+1} a_{i+2}$ 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 $a_i a_{i+1}$ and $C$ derives $a_{i+2}$ OR $B$ derives $a_i$ and $C$ derives $a_{i+1} a_{i+2}$.

In our example:

$$\begin{aligned} bab &: \\ aba &: \\ baa &: S, X \end{aligned}$$

We move up to longer and longer subwords. To find out which variables derive subword

$$u \;=\; a_i a_{i+1} \ldots a_{i+k}$$

of length $k+1$, we try all possible ways of dividing the subword into two non-empty parts:

$$\begin{aligned} u_1 &= a_i a_{i+1} \ldots a_{i+s} \\ u_2 &= a_{i+s+1} a_{i+s+2} \ldots a_{i+k} \end{aligned}$$

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

$$\begin{array}{cc} b & aba \\ ba & ba \\ bab & a \end{array} \qquad SS, SX, XS, XX$$

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

$$\begin{array}{cc} b & aba \\ ba & ba \\ bab & a \end{array} \qquad SS, SX, XS, XX$$

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:

$$\begin{array}{cc} a & baa \\ ab & aa \\ aba & a \end{array} \qquad XS, YS, ZS, XX, YX, ZX$$

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:

| | | |
|---|---|---|
| $b$ | $abaa$ | |
| $ba$ | $baa$ | $SS, SX, XS, XX$ |
| $bab$ | $aa$ | |
| $baba$ | $a$ | |

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

$$a_i a_{i+1} \dots a_{i+j-1}$$

of length $j$ starting at the $i$'th letter of the input word $w = a_1 a_2 \dots a_n$.

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 = a_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 $L(G)$.