Context-Free Grammars

Context-Free Grammars

Example: We would like to describe a language of arithmetic expressions, containing expressions such as:

`175` `(9+15)` `(((10-4)*((1+34)+2))/(3+(-37)))`

For simplicity we assume that:

- Expressions are fully parenthesized.
- The only arithmetic operations are "`+`", "`-`", "`*`", "`/`" and unary "`-`".
- Values of operands are natural numbers written in decimal — a number is represented as a non-empty sequence of digits.

Alphabet: $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, *, /, (,)\}$

Context-Free Grammars

Example (cont.): A description by an inductive definition:

- **Digit** is any of characters `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`.
- **Number** is a non-empty sequence of digits, i.e.:
  - If α is a digit then α is a number.
  - If α is a digit and β is a number then also $\alpha\beta$ is a number.
- **Expression** is a sequence of symbols constructed according to the following rules:
  - If α is a number then α is an expression.
  - If α is an expression then also $(-\alpha)$ is an expression.
  - If α and β are expressions then also $(\alpha+\beta)$ is an expression.
  - If α and β are expressions then also $(\alpha-\beta)$ is an expression.
  - If α and β are expressions then also $(\alpha*\beta)$ is an expression.
  - If α and β are expressions then also (α/β) is an expression.

Context-Free Grammars

Example (cont.): The same information that was described by the previous inductive definition can be represented by a **context-free grammar**:

New auxiliary symbols, called **nonterminals**, are introduced:

- D — stands for an arbitrary digit
- C — stands for an arbitrary number
- E — stands for an arbitrary expression

$$
\begin{array}{llll}
D \longrightarrow 0 & D \longrightarrow 5 & & E \longrightarrow C \\
D \longrightarrow 1 & D \longrightarrow 6 & & E \longrightarrow (-E) \\
D \longrightarrow 2 & D \longrightarrow 7 & C \longrightarrow D & E \longrightarrow (E+E) \\
D \longrightarrow 3 & D \longrightarrow 8 & C \longrightarrow DC & E \longrightarrow (E-E) \\
D \longrightarrow 4 & D \longrightarrow 9 & & E \longrightarrow (E*E) \\
 & & & E \longrightarrow (E/E)
\end{array}
$$

Context-Free Grammars

Example (cont.): Written in a more succinct way:

$$
\begin{aligned}
D &\rightarrow \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \\
C &\rightarrow D \mid DC \\
E &\rightarrow C \mid \texttt{(-}E\texttt{)} \mid \texttt{(}E\texttt{+}E\texttt{)} \mid \texttt{(}E\texttt{-}E\texttt{)} \mid \texttt{(}E\texttt{*}E\texttt{)} \mid \texttt{(}E\texttt{/}E\texttt{)}
\end{aligned}
$$

Context-Free Grammars

Example: A language where words are (possibly empty) sequences of expressions described in the previous example, where individual expressions are separated by commas (the alphabet must be extended with symbol "$\texttt{,}$"):

$$
\begin{aligned}
S &\rightarrow T \mid \varepsilon \\
T &\rightarrow E \mid E\texttt{,}T \\
D &\rightarrow \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \\
C &\rightarrow D \mid DC \\
E &\rightarrow C \mid \texttt{(-}E\texttt{)} \mid \texttt{(}E\texttt{+}E\texttt{)} \mid \texttt{(}E\texttt{-}E\texttt{)} \mid \texttt{(}E\texttt{*}E\texttt{)} \mid \texttt{(}E\texttt{/}E\texttt{)}
\end{aligned}
$$

Context-Free Grammars

Example: Statements of some programming language (a fragment of a grammar):

$$
\begin{aligned}
S &\rightarrow E\texttt{;} \mid T \mid \texttt{if (}E\texttt{)}\ S \mid \texttt{if (}E\texttt{)}\ S\ \texttt{else}\ S \\
&\quad \mid \texttt{while (}E\texttt{)}\ S \mid \texttt{do}\ S\ \texttt{while (}E\texttt{);} \mid \texttt{for (}F\texttt{;}F\texttt{;}F\texttt{)}\ S \\
&\quad \mid \texttt{return}\ F\texttt{;} \\
T &\rightarrow \texttt{\{}\ U\ \texttt{\}} \\
U &\rightarrow \varepsilon \mid SU \\
F &\rightarrow \varepsilon \mid E \\
E &\rightarrow \quad \ldots
\end{aligned}
$$

Remark:

- S — statement
- T — block of statements
- U — sequence of statements
- E — expression
- F — optional expression that can be omitted

Context-Free Grammars

Formally, a **context-free grammar** is a tuple

$$\mathcal{G} = (\Pi, \Sigma, S, P)$$

where:

- Π is a finite set of **nonterminal symbols** (**nonterminals**)
- Σ is a finite set of **terminal symbols** (**terminals**), where $\Pi \cap \Sigma = \emptyset$
- $S \in \Pi$ is an **initial nonterminal**
- $P \subseteq \Pi \times (\Pi \cup \Sigma)^*$ is a finite set of **rewrite rules**

Context-Free Grammars

Remarks:

- We will use uppercase letters A, B, C, ... to denote nonterminal symbols.
- We will use lowercase letters a, b, c, ... or digits 0, 1, 2, ... to denote terminal symbols.
- We will use lowercase Greek letters α, β, γ, ... do denote strings from $(\Pi \cup \Sigma)^*$.
- We will use the following notation for rules instead of (A, α)

$$A \rightarrow \alpha$$

A – left-hand side of the rule
α – right-hand side of the rule

Context-Free Grammars

Example: Grammar $\mathcal{G} = (\Pi, \Sigma, S, P)$ where

- $\Pi = \{A, B, C\}$
- $\Sigma = \{a, b\}$
- $S = A$
- P contains rules

$$
\begin{aligned}
A &\longrightarrow aBBb \\
A &\longrightarrow AaA \\
B &\longrightarrow \varepsilon \\
B &\longrightarrow bCA \\
C &\longrightarrow AB \\
C &\longrightarrow a \\
C &\longrightarrow b
\end{aligned}
$$

Context-Free Grammars

Remark: If we have more rules with the same left-hand side, as for example

$$A \longrightarrow \alpha_1 \qquad\qquad A \longrightarrow \alpha_2 \qquad\qquad A \longrightarrow \alpha_3$$

we can write them in a more succinct way as

$$A \longrightarrow \alpha_1 \mid \alpha_2 \mid \alpha_3$$

For example, the rules of the grammar from the previous slide can be written as

$$\begin{aligned} A &\longrightarrow aBBb \mid AaA \\ B &\longrightarrow \varepsilon \mid bCA \\ C &\longrightarrow AB \mid a \mid b \end{aligned}$$

Context-Free Grammars

Grammars are used for generating words.

Example: $\mathcal{G} = (\Pi, \Sigma, A, P)$ where $\Pi = \{A, B, C\}$, $\Sigma = \{a, b\}$, and P contains rules

$$\begin{aligned} A &\rightarrow aBBb \mid AaA \\ B &\rightarrow \varepsilon \mid bCA \\ C &\rightarrow AB \mid a \mid b \end{aligned}$$

For example, the word $abbabb$ can be in grammar $\mathcal{G}$ generated as follows:

Context-Free Grammars

Grammars are used for generating words.

Example: $\mathcal{G} = (\Pi, \Sigma, A, P)$ where $\Pi = \{A, B, C\}$, $\Sigma = \{a, b\}$, and P contains rules

$$\begin{aligned} A &\rightarrow aBBb \mid AaA \\ B &\rightarrow \varepsilon \mid bCA \\ C &\rightarrow AB \mid a \mid b \end{aligned}$$

For example, the word $abbabb$ can be in grammar $\mathcal{G}$ generated as follows:

A

Context-Free Grammars

Grammars are used for generating words.

Example: $\mathcal{G} = (\Pi, \Sigma, A, P)$ where $\Pi = \{A, B, C\}$, $\Sigma = \{a, b\}$, and P contains rules

$$\begin{aligned} \underline{A} &\rightarrow aBBb \mid AaA \\ B &\rightarrow \varepsilon \mid bCA \\ C &\rightarrow AB \mid a \mid b \end{aligned}$$

For example, the word $abbabb$ can be in grammar $\mathcal{G}$ generated as follows:

$\underline{A}$

Context-Free Grammars

Grammars are used for generating words.

Example: $\mathcal{G} = (\Pi, \Sigma, A, P)$ where $\Pi = \{A, B, C\}$, $\Sigma = \{a, b\}$, and P contains rules

$$\begin{aligned} \underline{A} &\rightarrow \underline{aBBb} \mid AaA \\ B &\rightarrow \varepsilon \mid bCA \\ C &\rightarrow AB \mid a \mid b \end{aligned}$$

For example, the word $abbabb$ can be in grammar $\mathcal{G}$ generated as follows:

$\underline{A} \Rightarrow \underline{aBBb}$

Context-Free Grammars

Grammars are used for generating words.

Example: $\mathcal{G} = (\Pi, \Sigma, A, P)$ where $\Pi = \{A, B, C\}$, $\Sigma = \{a, b\}$, and P contains rules

$$\begin{aligned} A &\rightarrow aBBb \mid AaA \\ B &\rightarrow \varepsilon \mid bCA \\ C &\rightarrow AB \mid a \mid b \end{aligned}$$

For example, the word $abbabb$ can be in grammar $\mathcal{G}$ generated as follows:

$A \Rightarrow aBBb$

Context-Free Grammars

Grammars are used for generating words.

Example: $\mathcal{G} = (\Pi, \Sigma, A, P)$ where $\Pi = \{A, B, C\}$, $\Sigma = \{a, b\}$, and P contains rules

$$\begin{aligned} A &\longrightarrow aBBb \mid AaA \\ \underline{B} &\longrightarrow \varepsilon \mid bCA \\ C &\longrightarrow AB \mid a \mid b \end{aligned}$$

For example, the word $abbabb$ can be in grammar $\mathcal{G}$ generated as follows:

$A \Rightarrow a\underline{B}Bb$

Context-Free Grammars

Grammars are used for generating words.

Example: $\mathcal{G} = (\Pi, \Sigma, A, P)$ where $\Pi = \{A, B, C\}$, $\Sigma = \{a, b\}$, and P contains rules

$$\begin{aligned} A &\longrightarrow aBBb \mid AaA \\ \underline{B} &\longrightarrow \varepsilon \mid \underline{bCA} \\ C &\longrightarrow AB \mid a \mid b \end{aligned}$$

For example, the word $abbabb$ can be in grammar $\mathcal{G}$ generated as follows:

$A \Rightarrow a\underline{B}Bb \Rightarrow a\underline{bCA}Bb$

Context-Free Grammars

Grammars are used for generating words.

Example: $\mathcal{G} = (\Pi, \Sigma, A, P)$ where $\Pi = \{A, B, C\}$, $\Sigma = \{a, b\}$, and P contains rules

$$\begin{aligned} A &\longrightarrow aBBb \mid AaA \\ B &\longrightarrow \varepsilon \mid bCA \\ C &\longrightarrow AB \mid a \mid b \end{aligned}$$

For example, the word $abbabb$ can be in grammar $\mathcal{G}$ generated as follows:

$A \Rightarrow aBBb \Rightarrow abCABb$

Context-Free Grammars

Grammars are used for generating words.

Example: $\mathcal{G} = (\Pi, \Sigma, A, P)$ where $\Pi = \{A, B, C\}$, $\Sigma = \{a, b\}$, and P contains rules

$$\begin{aligned} \underline{A} &\longrightarrow aBBb \mid AaA \\ B &\longrightarrow \varepsilon \mid bCA \\ C &\longrightarrow AB \mid a \mid b \end{aligned}$$

For example, the word $abbabb$ can be in grammar $\mathcal{G}$ generated as follows:

$A \Rightarrow aBBb \Rightarrow abC\underline{A}Bb$

Context-Free Grammars

Grammars are used for generating words.

Example: $\mathcal{G} = (\Pi, \Sigma, A, P)$ where $\Pi = \{A, B, C\}$, $\Sigma = \{a, b\}$, and P contains rules

$$\begin{aligned} \underline{A} &\rightarrow \underline{aBBb} \mid AaA \\ B &\rightarrow \varepsilon \mid bCA \\ C &\rightarrow AB \mid a \mid b \end{aligned}$$

For example, the word $abbabb$ can be in grammar $\mathcal{G}$ generated as follows:

$A \Rightarrow aBBb \Rightarrow abC\underline{A}Bb \Rightarrow abC\underline{aBBb}Bb$

Context-Free Grammars

Grammars are used for generating words.

Example: $\mathcal{G} = (\Pi, \Sigma, A, P)$ where $\Pi = \{A, B, C\}$, $\Sigma = \{a, b\}$, and P contains rules

$$\begin{aligned} A &\rightarrow aBBb \mid AaA \\ B &\rightarrow \varepsilon \mid bCA \\ C &\rightarrow AB \mid a \mid b \end{aligned}$$

For example, the word $abbabb$ can be in grammar $\mathcal{G}$ generated as follows:

$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb$

Context-Free Grammars

Grammars are used for generating words.

Example: $\mathcal{G} = (\Pi, \Sigma, A, P)$ where $\Pi = \{A, B, C\}$, $\Sigma = \{a, b\}$, and P contains rules

$$
\begin{aligned}
A &\rightarrow aBBb \mid AaA \\
\underline{B} &\rightarrow \varepsilon \mid bCA \\
C &\rightarrow AB \mid a \mid b
\end{aligned}
$$

For example, the word $abbabb$ can be in grammar $\mathcal{G}$ generated as follows:

$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaB\underline{B}bBb$

Context-Free Grammars

Grammars are used for generating words.

Example: $\mathcal{G} = (\Pi, \Sigma, A, P)$ where $\Pi = \{A, B, C\}$, $\Sigma = \{a, b\}$, and P contains rules

$$
\begin{aligned}
A &\rightarrow aBBb \mid AaA \\
\underline{B} &\rightarrow \underline{\varepsilon} \mid bCA \\
C &\rightarrow AB \mid a \mid b
\end{aligned}
$$

For example, the word *abbabb* can be in grammar $\mathcal{G}$ generated as follows:

$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaB\underline{B}bBb \Rightarrow abCaBbBb$

Context-Free Grammars

Grammars are used for generating words.

Example: $\mathcal{G} = (\Pi, \Sigma, A, P)$ where $\Pi = \{A, B, C\}$, $\Sigma = \{a, b\}$, and P contains rules

$$\begin{aligned} A &\rightarrow aBBb \mid AaA \\ B &\rightarrow \varepsilon \mid bCA \\ C &\rightarrow AB \mid a \mid b \end{aligned}$$

For example, the word $abbabb$ can be in grammar $\mathcal{G}$ generated as follows:

$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb$

Grammars are used for generating words.

Example: $\mathcal{G} = (\Pi, \Sigma, A, P)$ where $\Pi = \{A, B, C\}$, $\Sigma = \{a, b\}$, and P contains rules

$$\begin{aligned} A &\rightarrow aBBb \mid AaA \\ B &\rightarrow \varepsilon \mid bCA \\ \underline{C} &\rightarrow AB \mid a \mid b \end{aligned}$$

For example, the word $abbabb$ can be in grammar $\mathcal{G}$ generated as follows:

$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow ab\underline{C}aBbBb$

Context-Free Grammars

Grammars are used for generating words.

Example: $\mathcal{G} = (\Pi, \Sigma, A, P)$ where $\Pi = \{A, B, C\}$, $\Sigma = \{a, b\}$, and P contains rules

$$\begin{aligned} A &\rightarrow aBBb \mid AaA \\ B &\rightarrow \varepsilon \mid bCA \\ \underline{C} &\rightarrow AB \mid a \mid \underline{b} \end{aligned}$$

For example, the word *abbabb* can be in grammar $\mathcal{G}$ generated as follows:

$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow ab\underline{C}aBbBb \Rightarrow ab\underline{b}aBbBb$

Context-Free Grammars

Grammars are used for generating words.

Example: $\mathcal{G} = (\Pi, \Sigma, A, P)$ where $\Pi = \{A, B, C\}$, $\Sigma = \{a, b\}$, and P contains rules

$$\begin{aligned} A &\rightarrow aBBb \mid AaA \\ B &\rightarrow \varepsilon \mid bCA \\ C &\rightarrow AB \mid a \mid b \end{aligned}$$

For example, the word $abbabb$ can be in grammar $\mathcal{G}$ generated as follows:

$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBbBb$

Context-Free Grammars

Grammars are used for generating words.

Example: $\mathcal{G} = (\Pi, \Sigma, A, P)$ where $\Pi = \{A, B, C\}$, $\Sigma = \{a, b\}$, and P contains rules

$$
\begin{aligned}
A &\rightarrow aBBb \mid AaA \\
\underline{B} &\rightarrow \varepsilon \mid bCA \\
C &\rightarrow AB \mid a \mid b
\end{aligned}
$$

For example, the word $abbabb$ can be in grammar $\mathcal{G}$ generated as follows:

$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBb\underline{B}b$

Context-Free Grammars

Grammars are used for generating words.

Example: $\mathcal{G} = (\Pi, \Sigma, A, P)$ where $\Pi = \{A, B, C\}$, $\Sigma = \{a, b\}$, and P contains rules

$$\begin{aligned} A &\rightarrow aBBb \mid AaA \\ \underline{B} &\rightarrow \underline{\varepsilon} \mid bCA \\ C &\rightarrow AB \mid a \mid b \end{aligned}$$

For example, the word $abbabb$ can be in grammar $\mathcal{G}$ generated as follows:

$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBb\underline{B}b \Rightarrow abbaBbb$

Context-Free Grammars

Grammars are used for generating words.

Example: $\mathcal{G} = (\Pi, \Sigma, A, P)$ where $\Pi = \{A, B, C\}$, $\Sigma = \{a, b\}$, and P contains rules

$$
\begin{aligned}
A &\rightarrow aBBb \mid AaA \\
B &\rightarrow \varepsilon \mid bCA \\
C &\rightarrow AB \mid a \mid b
\end{aligned}
$$

For example, the word $abbabb$ can be in grammar $\mathcal{G}$ generated as follows:

$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBbBb \Rightarrow abbaBbb$

Context-Free Grammars

Grammars are used for generating words.

Example: $\mathcal{G} = (\Pi, \Sigma, A, P)$ where $\Pi = \{A, B, C\}$, $\Sigma = \{a, b\}$, and P contains rules

$$
\begin{aligned}
A &\rightarrow aBBb \mid AaA \\
\underline{B} &\rightarrow \varepsilon \mid bCA \\
C &\rightarrow AB \mid a \mid b
\end{aligned}
$$

For example, the word $abbabb$ can be in grammar $\mathcal{G}$ generated as follows:

$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBbBb \Rightarrow abba\underline{B}bb$

Context-Free Grammars

Grammars are used for generating words.

Example: $\mathcal{G} = (\Pi, \Sigma, A, P)$ where $\Pi = \{A, B, C\}$, $\Sigma = \{a, b\}$, and P contains rules

$$\begin{aligned} A &\rightarrow aBBb \mid AaA \\ \underline{B} &\rightarrow \underline{\varepsilon} \mid bCA \\ C &\rightarrow AB \mid a \mid b \end{aligned}$$

For example, the word $abbabb$ can be in grammar $\mathcal{G}$ generated as follows:

$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBbBb \Rightarrow abba\underline{B}bb \Rightarrow abbabb$

Context-Free Grammars

Grammars are used for generating words.

Example: $\mathcal{G} = (\Pi, \Sigma, A, P)$ where $\Pi = \{A, B, C\}$, $\Sigma = \{a, b\}$, and P contains rules

$$
\begin{aligned}
A &\rightarrow aBBb \mid AaA \\
B &\rightarrow \varepsilon \mid bCA \\
C &\rightarrow AB \mid a \mid b
\end{aligned}
$$

For example, the word $abbabb$ can be in grammar $\mathcal{G}$ generated as follows:

$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBbBb \Rightarrow abbaBbb \Rightarrow abbabb$

Context-Free Grammars

On strings from $(\Pi \cup \Sigma)^*$ we define relation $\Rightarrow \subseteq (\Pi \cup \Sigma)^* \times (\Pi \cup \Sigma)^*$ such that

$$\alpha \Rightarrow \alpha'$$

iff $\alpha = \beta_1 A \beta_2$ and $\alpha' = \beta_1 \gamma \beta_2$ for some $\beta_1, \beta_2, \gamma \in (\Pi \cup \Sigma)^*$ and $A \in \Pi$ where $(A \rightarrow \gamma) \in P$.

Example: If $(B \rightarrow bCA) \in P$ then

$$aCBbA \Rightarrow aCbCAbA$$

Remark: Informally, $\alpha \Rightarrow \alpha'$ means that it is possible to derive α' from α by one step where an occurrence of some nonterminal A in α is replaced with the right-hand side of some rule $A \rightarrow \gamma$ with A on the left-hand side.

On strings from $(\Pi \cup \Sigma)^*$ we define relation $\Rightarrow \subseteq (\Pi \cup \Sigma)^* \times (\Pi \cup \Sigma)^*$ such that

$$\alpha \Rightarrow \alpha'$$

iff $\alpha = \beta_1 A \beta_2$ and $\alpha' = \beta_1 \gamma \beta_2$ for some $\beta_1, \beta_2, \gamma \in (\Pi \cup \Sigma)^*$ and $A \in \Pi$ where $(A \rightarrow \gamma) \in P$.

Example: If $(B \rightarrow bCA) \in P$ then

$$aC\underline{B}bA \Rightarrow aC\underline{bCA}bA$$

Remark: Informally, $\alpha \Rightarrow \alpha'$ means that it is possible to derive α' from α by one step where an occurrence of some nonterminal A in α is replaced with the right-hand side of some rule $A \rightarrow \gamma$ with A on the left-hand side.

Context-Free Grammars

A **derivation** of length n is a sequence $\beta_0, \beta_1, \beta_2, \cdots, \beta_n$, where $\beta_i \in (\Pi \cup \Sigma)^*$, and where $\beta_{i-1} \Rightarrow \beta_i$ for all $1 \leq i \leq n$, which can be written more succinctly as

$$\beta_0 \Rightarrow \beta_1 \Rightarrow \beta_2 \Rightarrow \ldots \Rightarrow \beta_{n-1} \Rightarrow \beta_n$$

The fact that for given $\alpha, \alpha' \in (\Pi \cup \Sigma)^*$ and $n \in \mathbb{N}$ there exists some derivation $\beta_0 \Rightarrow \beta_1 \Rightarrow \beta_2 \Rightarrow \ldots \Rightarrow \beta_{n-1} \Rightarrow \beta_n$, where $\alpha = \beta_0$ and $\alpha' = \beta_n$, is denoted

$$\alpha \Rightarrow^n \alpha'$$

The fact that $\alpha \Rightarrow^n \alpha'$ for some $n \geq 0$, is denoted

$$\alpha \Rightarrow^* \alpha'$$

Remark: Relation $\Rightarrow^*$ is the reflexive and transitive closure of relation $\Rightarrow$ (i.e., the smallest reflexive and transitive relation containing relation $\Rightarrow$).

Sentential forms are those $\alpha \in (\Pi \cup \Sigma)^*$, for which

$$S \Rightarrow^* \alpha$$

where S is the initial nonterminal.

Context-Free Grammars

A **language** $\mathcal{L}(\mathcal{G})$ generated by a grammar $\mathcal{G} = (\Pi, \Sigma, S, P)$ is the set of all words over alphabet Σ that can be derived by some derivation from the initial nonterminal S using rules from P, i.e.,

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

Context-Free Grammars

Example: We want to construct a grammar generating the language

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

Context-Free Grammars

Example: We want to construct a grammar generating the language

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

Grammar $\mathcal{G} = (\Pi, \Sigma, S, P)$ where $\Pi = \{S\}$, $\Sigma = \{a, b\}$, and P contains

$$S \rightarrow \varepsilon \mid aSb$$

Context-Free Grammars

Example: We want to construct a grammar generating the language

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

Grammar $\mathcal{G} = (\Pi, \Sigma, S, P)$ where $\Pi = \{S\}$, $\Sigma = \{a, b\}$, and P contains

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

$$\begin{aligned}
&S \Rightarrow \varepsilon \\
&S \Rightarrow aSb \Rightarrow ab \\
&S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb \\
&S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbb \\
&S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaaaSbbbb \Rightarrow aaaabbbb \\
&\qquad \cdots
\end{aligned}$$

Context-Free Grammars

Example: We want to construct a grammar generating the language consisting of all palindroms over the alphabet $\{a, b\}$, i.e.,

$$L = \{w \in \{a, b\}^* \mid w = w^R\}$$

Remark: w^R denotes the **reverse** of a word w, i.e., the word w written backwards.

Example: We want to construct a grammar generating the language consisting of all palindroms over the alphabet $\{a, b\}$, i.e.,

$$L = \{w \in \{a, b\}^* \mid w = w^R\}$$

Remark: w^R denotes the **reverse** of a word w, i.e., the word w written backwards.

Solution:

$$S \rightarrow \varepsilon \mid a \mid b \mid aSa \mid bSb$$

Example: We want to construct a grammar generating the language consisting of all palindroms over the alphabet $\{a, b\}$, i.e.,

$$L = \{w \in \{a, b\}^* \mid w = w^R\}$$

Remark: w^R denotes the **reverse** of a word w, i.e., the word w written backwards.

Solution:

$$S \to \varepsilon \mid a \mid b \mid aSa \mid bSb$$

$$S \Rightarrow aSa \Rightarrow abSba \Rightarrow abaSaba \Rightarrow abaaaba$$

Context-Free Grammars

Example: We want to construct a grammar generating the language L consisting of all correctly parenthesised sequences of symbols '$($' and '$)$'.

For example $(()())(()) \in L$ but $)()) \notin L$.

Context-Free Grammars

Example: We want to construct a grammar generating the language L consisting of all correctly parenthesised sequences of symbols '$($' and '$)$'.

For example $(()())(()) \in L$ but $)()) \notin L$.

Solution:

$$S \rightarrow \varepsilon \mid (S) \mid SS$$

Context-Free Grammars

Example: We want to construct a grammar generating the language L consisting of all correctly parenthesised sequences of symbols '$($' and '$)$'.

For example $(()())(()) \in L$ but $)()) \notin L$.

Solution:

$$S \rightarrow \varepsilon \mid (S) \mid SS$$

$S \Rightarrow SS \Rightarrow (S)S \Rightarrow (S)(S) \Rightarrow (SS)(S) \Rightarrow ((S)S)(S) \Rightarrow (()S)(S) \Rightarrow (()(S))(S) \Rightarrow (()())(S) \Rightarrow (()())((S)) \Rightarrow (()())(())$

Context-Free Grammars

Example: We want to construct a grammar generating the language L consisting of all correctly constructed arithmetic experessions where operands are always of the form 'a' and where symbols $+$ and $*$ can be used as operators.

For example $(a + a) * a + (a * a) \in L$.

Context-Free Grammars

Example: We want to construct a grammar generating the language L consisting of all correctly constructed arithmetic experessions where operands are always of the form 'a' and where symbols $+$ and $*$ can be used as operators.

For example $(a + a) * a + (a * a) \in L$.

Solution:

$$E \rightarrow a \mid E + E \mid E * E \mid (E)$$

Example: We want to construct a grammar generating the language L consisting of all correctly constructed arithmetic experessions where operands are always of the form 'a' and where symbols $+$ and $*$ can be used as operators.

For example $(a + a) * a + (a * a) \in L$.

Solution:

$$E \rightarrow a \mid E + E \mid E * E \mid (E)$$

$E \Rightarrow E + E \Rightarrow E * E + E \Rightarrow (E) * E + E \Rightarrow (E + E) * E + E \Rightarrow (a + E) * E + E \Rightarrow (a + a) * E + E \Rightarrow (a + a) * a + E \Rightarrow (a + a) * a + (E) \Rightarrow (a + a) * a + (E * E) \Rightarrow (a + a) * a + (a * E) \Rightarrow (a + a) * a + (a * a)$

Derivation Tree

$$
\begin{aligned}
A &\rightarrow aBBb \mid AaA \\
B &\rightarrow \varepsilon \mid bCA \\
C &\rightarrow AB \mid a \mid b
\end{aligned}
$$

Derivation Tree

A

$$A \longrightarrow aBBb \mid AaA$$
$$B \longrightarrow \varepsilon \mid bCA$$
$$C \longrightarrow AB \mid a \mid b$$

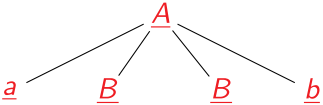
A

Derivation Tree

$\underline{A}$

$$\begin{aligned} \underline{A} &\rightarrow aBBb \mid AaA \\ B &\rightarrow \varepsilon \mid bCA \\ C &\rightarrow AB \mid a \mid b \end{aligned}$$

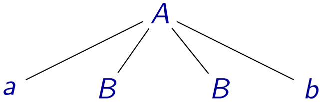
$\underline{A}$



$$\underline{A} \longrightarrow \underline{aBBb} \mid AaA$$
$$B \longrightarrow \varepsilon \mid bCA$$
$$C \longrightarrow AB \mid a \mid b$$

$$\underline{A} \Rightarrow \underline{aBBb}$$

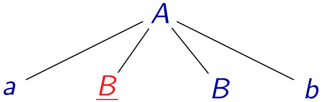
Derivation Tree



$$A \rightarrow aBBb \mid AaA$$
$$B \rightarrow \varepsilon \mid bCA$$
$$C \rightarrow AB \mid a \mid b$$

$$A \Rightarrow aBBb$$

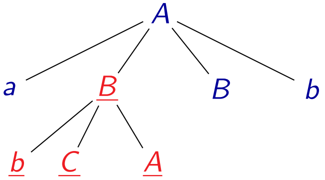
Derivation Tree



$$
\begin{aligned}
A &\rightarrow aBBb \mid AaA \\
\underline{B} &\rightarrow \varepsilon \mid bCA \\
C &\rightarrow AB \mid a \mid b
\end{aligned}
$$

$$A \Rightarrow a\underline{B}Bb$$

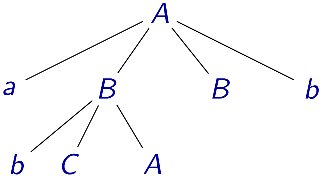
Derivation Tree



$$
\begin{aligned}
A &\rightarrow aBBb \mid AaA \\
\underline{B} &\rightarrow \varepsilon \mid \underline{bCA} \\
C &\rightarrow AB \mid a \mid b
\end{aligned}
$$

$$A \Rightarrow a\underline{B}Bb \Rightarrow a\underline{bCA}Bb$$

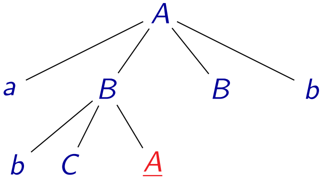
Derivation Tree



$$A \longrightarrow aBBb \mid AaA$$
$$B \longrightarrow \varepsilon \mid bCA$$
$$C \longrightarrow AB \mid a \mid b$$

$$A \Rightarrow aBBb \Rightarrow abCABb$$

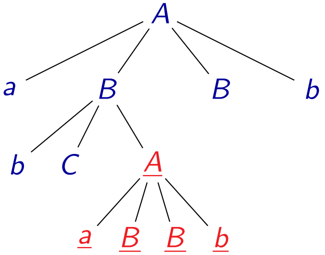
Derivation Tree



$$\underline{A} \rightarrow aBBb \mid AaA$$
$$B \rightarrow \varepsilon \mid bCA$$
$$C \rightarrow AB \mid a \mid b$$

$$A \Rightarrow aBBb \Rightarrow abC\underline{A}Bb$$

Derivation Tree

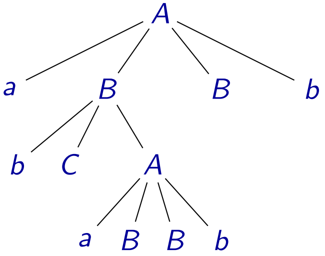


$\underline{A} \longrightarrow \underline{aBBb} \mid AaA$
$B \longrightarrow \varepsilon \mid bCA$
$C \longrightarrow AB \mid a \mid b$

$$A \Rightarrow aBBb \Rightarrow abC\underline{A}Bb \Rightarrow abC\underline{aBBb}Bb$$

Derivation Tree

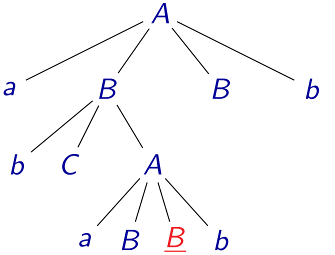
$A \rightarrow aBBb \mid AaA$
$B \rightarrow \varepsilon \mid bCA$
$C \rightarrow AB \mid a \mid b$



$$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb$$

Derivation Tree

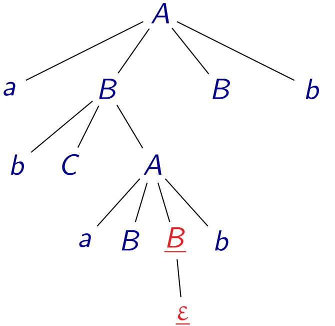
$$\begin{aligned} A &\longrightarrow aBBb \mid AaA \\ \underline{B} &\longrightarrow \varepsilon \mid bCA \\ C &\longrightarrow AB \mid a \mid b \end{aligned}$$



$$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaB\underline{B}bBb$$

Derivation Tree

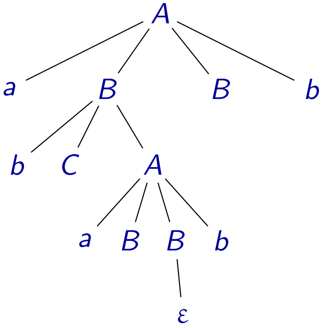
$A \longrightarrow aBBb \mid AaA$
$\underline{B} \longrightarrow \underline{\varepsilon} \mid bCA$
$C \longrightarrow AB \mid a \mid b$



$$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaB\underline{B}bBb \Rightarrow abCaBbBb$$

Derivation Tree

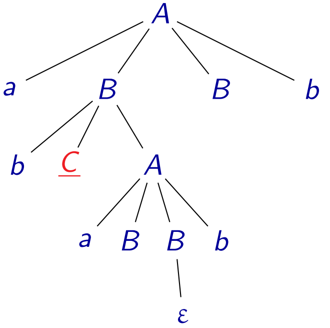
$$
\begin{aligned}
A &\rightarrow aBBb \mid AaA \\
B &\rightarrow \varepsilon \mid bCA \\
C &\rightarrow AB \mid a \mid b
\end{aligned}
$$



$$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb$$

Derivation Tree

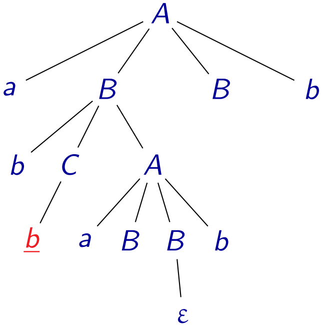
$A \longrightarrow aBBb \mid AaA$
$B \longrightarrow \varepsilon \mid bCA$
$\underline{C} \longrightarrow AB \mid a \mid b$



$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow ab\underline{C}aBbBb$

Derivation Tree

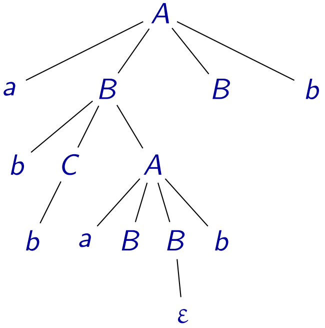
$$
\begin{aligned}
A &\longrightarrow aBBb \mid AaA \\
B &\longrightarrow \varepsilon \mid bCA \\
\underline{C} &\longrightarrow AB \mid a \mid \underline{b}
\end{aligned}
$$



$$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow ab\underline{C}aBbBb \Rightarrow ab\underline{b}aBbBb$$

Derivation Tree

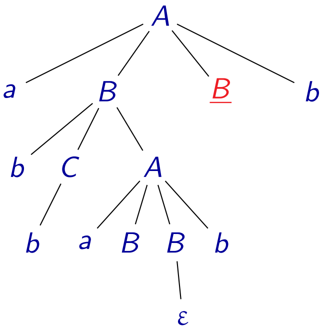
$$
\begin{aligned}
A &\longrightarrow aBBb \mid AaA \\
B &\longrightarrow \varepsilon \mid bCA \\
C &\longrightarrow AB \mid a \mid b
\end{aligned}
$$



$$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBbBb$$

Derivation Tree

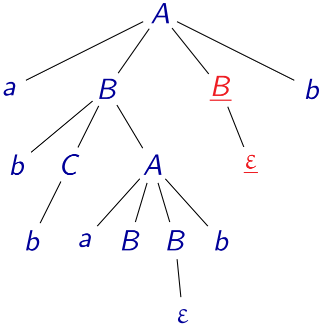
$A \longrightarrow aBBb \mid AaA$
$\underline{B} \longrightarrow \varepsilon \mid bCA$
$C \longrightarrow AB \mid a \mid b$



$$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBb\underline{B}b$$

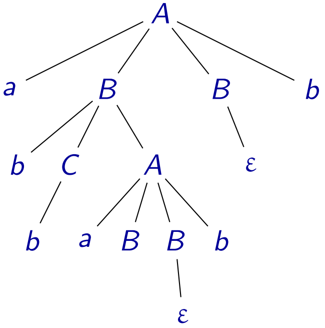
Derivation Tree

$A \rightarrow aBBb \mid AaA$
$\underline{B} \rightarrow \underline{\varepsilon} \mid bCA$
$C \rightarrow AB \mid a \mid b$



$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBb\underline{B}b \Rightarrow abbaBbb$

Derivation Tree

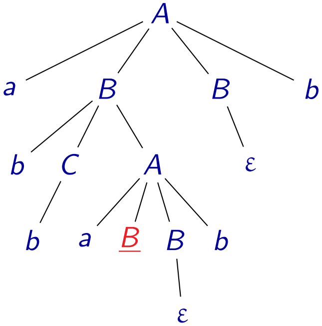


$A \to aBBb \mid AaA$
$B \to \varepsilon \mid bCA$
$C \to AB \mid a \mid b$

$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBbBb \Rightarrow abbaBbb$

Derivation Tree

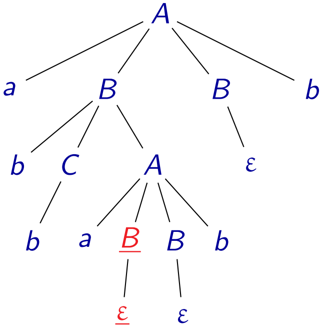
$A \longrightarrow aBBb \mid AaA$
$\underline{B} \longrightarrow \varepsilon \mid bCA$
$C \longrightarrow AB \mid a \mid b$



$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBbBb \Rightarrow abba\underline{B}bb$

Derivation Tree

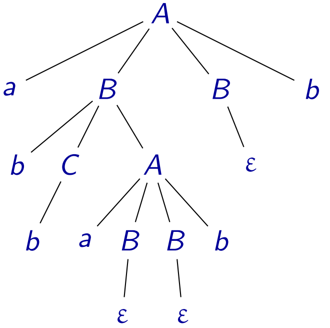
$A \longrightarrow aBBb \mid AaA$
$\underline{B} \longrightarrow \underline{\varepsilon} \mid bCA$
$C \longrightarrow AB \mid a \mid b$



$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBbBb \Rightarrow$
$abba\underline{B}bb \Rightarrow abbabb$

Derivation Tree

$$A \longrightarrow aBBb \mid AaA$$
$$B \longrightarrow \varepsilon \mid bCA$$
$$C \longrightarrow AB \mid a \mid b$$



$$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBbBb \Rightarrow abbaBbb \Rightarrow abbabb$$

Derivation Tree

For each derivation there is some **derivation tree**:

- Nodes of the tree are labelled with terminals and nonterminals.
- The root of the tree is labelled with the initial nonterminal.
- The leafs of the tree are labelled with terminals or with symbols ε.
- The remaining nodes of the tree are labelled with nonterminals.
- If a node is labelled with some nonterminal A then its children are labelled with the symbols from the right-hand side of some rewriting rule $A \rightarrow \alpha$.

Left and Right Derivation

$$E \longrightarrow a \mid E + E \mid E * E \mid (E)$$

A **left derivation** is a derivation where in every step we always replace the leftmost nonterminal.

$$\underline{E} \Rightarrow \underline{E} + E \Rightarrow \underline{E} * E + E \Rightarrow a * \underline{E} + E \Rightarrow a * a + \underline{E} \Rightarrow a * a + a$$

A **right derivation** is a derivation where in every step we always replace the rightmost nonterminal.

$$\underline{E} \Rightarrow E + \underline{E} \Rightarrow \underline{E} + a \Rightarrow E * \underline{E} + a \Rightarrow \underline{E} * a + a \Rightarrow a * a + a$$

A derivation need not be left or right:

$$\underline{E} \Rightarrow \underline{E} + E \Rightarrow E * \underline{E} + E \Rightarrow E * a + \underline{E} \Rightarrow \underline{E} * a + a \Rightarrow a * a + a$$

Left and Right Derivation

- There can be several different derivations corresponding to one derivation tree.
- For every derivation tree, there is exactly one left and exactly one right derivation corresponding to the tree.

Equivalence of Grammars

Grammars $\mathcal{G}_1$ and $\mathcal{G}_2$ are **equivalent** if they generate the same language, i.e., if $\mathcal{L}(\mathcal{G}_1) = \mathcal{L}(\mathcal{G}_2)$.

Remark: The problem of equivalence of context-free grammars is algorithmically undecidable. It can be shown that it is not possible to construct an algorithm that would decide for any pair of context-free grammars if they are equivalent or not.

Even the problem to decide if a grammar generates the language Σ^* is algorithmically undecidable.

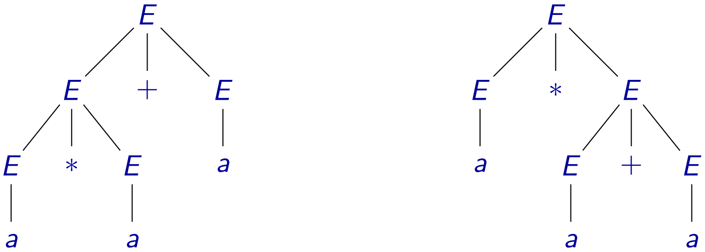
Ambiguous Grammars

A grammar $\mathcal{G}$ is **ambiguous** if there is a word $w \in \mathcal{L}(\mathcal{G})$ that has two different derivation trees, resp. two different left or two different right derivations.

Example:

$E \Rightarrow E + E \Rightarrow E * E + E \Rightarrow a * E + E \Rightarrow a * a + E \Rightarrow a * a + a$

$E \Rightarrow E * E \Rightarrow E * E + E \Rightarrow a * E + E \Rightarrow a * a + E \Rightarrow a * a + a$



Ambiguous Grammars

Sometimes it is possible to replace an ambiguous grammar with a grammar generating the same language but which is not ambiguous.

Example: A grammar

$$E \longrightarrow a \mid E + E \mid E * E \mid (E)$$

can be replaced with the equivalent grammar

$$\begin{aligned} E &\longrightarrow T \mid T + E \\ T &\longrightarrow F \mid F * T \\ F &\longrightarrow a \mid (E) \end{aligned}$$

Remark: If there is no unambiguous grammar equivalent to a given ambiguous grammar, we say it is **inherently ambiguous**.

Context-Free Languages

Definition

A language L is **context-free** if there exists some context-free grammar $\mathcal{G}$ such that $L = \mathcal{L}(\mathcal{G})$.

The class of context-free languages is closed with respect to:

- concatenation
- union
- iteration

The class of context-free languages is not closed with respect to:

- complement
- intersection

Context-Free Languages

We have two grammars $\mathcal{G}_1 = (\Pi_1, \Sigma, S_1, P_1)$ and $\mathcal{G}_2 = (\Pi_2, \Sigma, S_2, P_2)$, and can assume that $\Pi_1 \cap \Pi_2 = \emptyset$ and $S \notin \Pi_1 \cup \Pi_2$.

- Grammar $\mathcal{G}$ such that $\mathcal{L}(\mathcal{G}) = \mathcal{L}(\mathcal{G}_1) \cdot \mathcal{L}(\mathcal{G}_2)$:

$$\mathcal{G} = (\Pi_1 \cup \Pi_2 \cup \{S\},\ \Sigma,\ S,\ P_1 \cup P_2 \cup \{S \ \rightarrow \ S_1 S_2\})$$

- Grammar $\mathcal{G}$ such that $\mathcal{L}(\mathcal{G}) = \mathcal{L}(\mathcal{G}_1) \cup \mathcal{L}(\mathcal{G}_2)$:

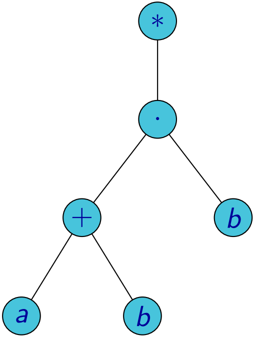
$$\mathcal{G} = (\Pi_1 \cup \Pi_2 \cup \{S\},\ \Sigma,\ S,\ P_1 \cup P_2 \cup \{S \ \rightarrow \ S_1, S \ \rightarrow \ S_2\})$$

- Grammar $\mathcal{G}$ such that $\mathcal{L}(\mathcal{G}) = \mathcal{L}(\mathcal{G}_1)^*$:

$$\mathcal{G} = (\Pi_1 \cup \{S\},\ \Sigma,\ S,\ P_1 \cup \{S \ \rightarrow \ \varepsilon, S \ \rightarrow \ S_1 S\})$$

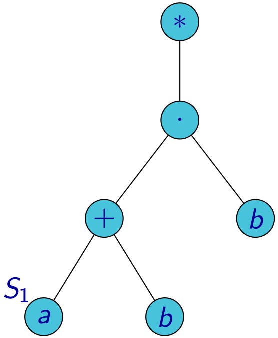
A Context-Free Grammar for a Regular Expression

Example: The construction of a context-free grammar for regular expression $((a+b)\cdot b)^*$:



A Context-Free Grammar for a Regular Expression

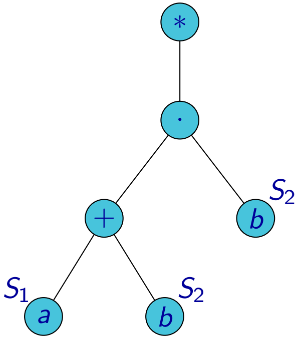
Example: The construction of a context-free grammar for regular expression $((a + b) \cdot b)^*$:



$S_1 \longrightarrow a$

A Context-Free Grammar for a Regular Expression

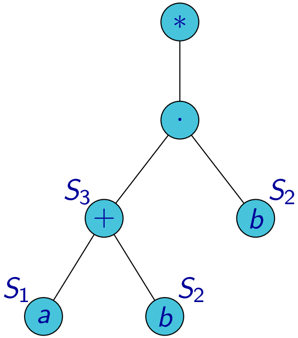
Example: The construction of a context-free grammar for regular expression $((a + b) \cdot b)^*$:



$$S_2 \longrightarrow b$$
$$S_1 \longrightarrow a$$

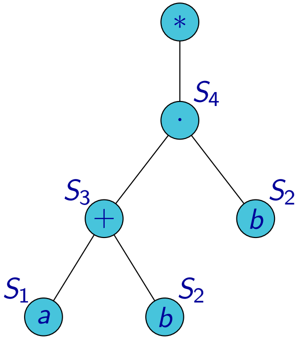
A Context-Free Grammar for a Regular Expression

Example: The construction of a context-free grammar for regular expression $((a+b)\cdot b)^*$:



$$
\begin{aligned}
S_3 &\rightarrow S_1 \mid S_2 \\
S_2 &\rightarrow b \\
S_1 &\rightarrow a
\end{aligned}
$$

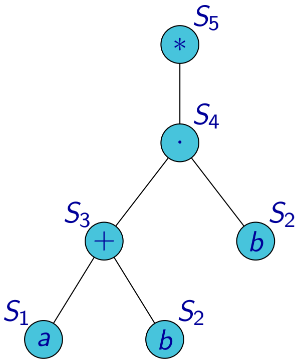
Example: The construction of a context-free grammar for regular expression $((a + b) \cdot b)^*$:



$$
\begin{aligned}
S_4 &\longrightarrow S_3 S_2 \\
S_3 &\longrightarrow S_1 \mid S_2 \\
S_2 &\longrightarrow b \\
S_1 &\longrightarrow a
\end{aligned}
$$

A Context-Free Grammar for a Regular Expression

Example: The construction of a context-free grammar for regular expression $((a + b) \cdot b)^*$:



$$
\begin{aligned}
S_5 &\longrightarrow \varepsilon \mid S_4 S_5 \\
S_4 &\longrightarrow S_3 S_2 \\
S_3 &\longrightarrow S_1 \mid S_2 \\
S_2 &\longrightarrow b \\
S_1 &\longrightarrow a
\end{aligned}
$$

Lexical and Syntactic Analysis — an example

Example: We would like to recognize a language of arithmetic expressions containing expressions such as:

`34`   `x+1`   `-x * 2 + 128 * (y - z / 3)`

- The expressions can contain number constants — sequences of digits `0`, `1`, ... , `9`.
- The expressions can contain names of variables — sequences consisting of letters, digits, and symbol "`_`", which do not start with a digit.
- The expressions can contain basic arithmetic operations — "`+`", "`-`", "`*`", "`/`", and unary "`-`".
- It is possible to use parentheses — "`(`" and "`)`", and to use a standard priority of arithmetic operations.

Lexical and Syntactic Analysis — an example

The problem we want to solve:

- **Input:** a sequence of characters (e.g., a string, a text file, etc.)
- **Output:** an abstract syntax tree representing the structure of a given expression, or an information about a syntax error in the expression

Lexical and Syntactic Analysis — an example

It is convenient to decompose this problem into several parts:

- **Lexical analysis** — recognizing of **lexical elements** (so called **tokens**) such as for example identifiers, number constants, operators, etc.

- **Syntactic analysis** — determining whether a given sequence of tokens corresponds to an allowed structure of expressions; basically, it means finding corresponding derivation (resp. derivation tree) for a given word in a context-free grammar representing the given language (e.g., in our case, the language of all well-formed expressions).

- **Construction of an abstract syntax tree** — this phase is usually connected with the syntax analysis, where the result, actually produced by the program, is typically not directly a derivation tree but rather some kind of abstract syntax tree or performing of some actions connected with rules of the given grammar.

Lexical and Syntactic Analysis — an example

Terminals for the grammar representing well-formed expressions:

$\langle \mathit{ident} \rangle$	— identifier, e.g. "`x`", "`q3`", "`count_r12`"
$\langle \mathit{num} \rangle$	— number constant, e.g. "`5`", "`42`", "`65535`"
"`(`"	— left parenthesis
"`)`"	— right parenthesis
"`+`"	— plus
"`-`"	— minus
"`*`"	— star
"`/`"	— slash

Remark: Recognizing of sequences of symbols that correspond to individual terminals is the goal of lexical analysis.

Lexical and Syntactic Analysis — an example

Example: Expression `-x * 2 + 128 * (y - z / 3)` is represented by the following sequence of symbols:

`- x ␣ * ␣ 2 ␣ + ␣ 1 2 8 ␣ * ␣ ( y ␣ - ␣ z ␣ / ␣ 3 )`

The following sequence of **tokens** corresponds to this sequence of symbols; these tokens are terminal symbols of the given context-free grammar:

$$- \ \langle \mathit{ident} \rangle \ * \ \langle \mathit{num} \rangle \ + \ \langle \mathit{num} \rangle \ * \ (\ \langle \mathit{ident} \rangle \ - \ \langle \mathit{ident} \rangle \ / \ \langle \mathit{num} \rangle \ )$$

The context-free grammar for the given language — the first try:

$$E \rightarrow \langle ident \rangle \mid \langle num \rangle \mid (\, E\,) \mid -E \mid E + E \mid E - E \mid E * E \mid E \,/\, E$$

Lexical and Syntactic Analysis — an example

The context-free grammar for the given language — the first try:

$$E \rightarrow \langle ident \rangle \mid \langle num \rangle \mid (\, E\,) \mid -E \mid E + E \mid E - E \mid E * E \mid E \,/\, E$$

This grammar is ambiguous.

Lexical and Syntactic Analysis — an example

The context-free grammar for the given language — the second try:

$$
\begin{aligned}
E &\rightarrow T \mid T + E \mid T - E \\
T &\rightarrow F \mid F * T \mid F \,/\, T \\
F &\rightarrow \langle \mathit{ident} \rangle \mid \langle \mathit{num} \rangle \mid (\, E \,) \mid - F
\end{aligned}
$$

Different levels of priority are represented by different nonterminals:

- E — expression
- T — term
- F — factor

This grammar is unambiguous.

Lexical and Syntactic Analysis — an example

The context-free grammar for the given language — the third try:

$$
\begin{aligned}
E &\rightarrow T \mid T\,A\,E \\
A &\rightarrow \texttt{+} \mid \texttt{-} \\
T &\rightarrow F \mid F\,M\,T \\
M &\rightarrow \ast \mid / \\
F &\rightarrow \langle ident \rangle \mid \langle num \rangle \mid (\,E\,) \mid \texttt{-}\,F
\end{aligned}
$$

We create separate nonterminals for operators on different levels of priority:

- A — additive operator
- M — multiplicative operator

Lexical and Syntactic Analysis — an example

The context-free grammar for the given language — the fourth try:

$$
\begin{aligned}
S &\longrightarrow E\ \langle eof \rangle \\
E &\longrightarrow T \mid T\ A\ E \\
A &\longrightarrow + \mid - \\
T &\longrightarrow F \mid F\ M\ T \\
M &\longrightarrow * \mid / \\
F &\longrightarrow \langle ident \rangle \mid \langle num \rangle \mid (\ E\ ) \mid -\ F
\end{aligned}
$$

- It is useful to introduce special nonterminal $\langle eof \rangle$ representing the end of input.
- Moreover, in this grammar the initial nonterminal S does not occur on the right hand side of any grammar.

Implementation of Lexical Analysis

Enumerated type *Token_kind* representing different kinds of **tokens**:

T_EOF	—	the end of input
T_Ident	—	identifier
T_Number	—	number constant
T_LParen	—	"("
T_RParen	—	")"
T_Plus	—	"+"
T_Minus	—	"-"
T_Star	—	"*"
T_Slash	—	"/"

Implementation of Lexical Analysis

Variable c : a currently processed character (resp. a special value $\langle eof \rangle$ representing the end of input):

- at the beginning, the first character in the input is read to variable c
- function $\textsc{Next-Char}()$ returns a next charater from the input

Some helper functions:

- $\textsc{Error}()$ — outputs an information about a syntax error and aborts the processing of the expression
- **is-ident-start-char**(c) — tests whether c is a charater that can occur at the beginning of an identifier
- **is-ident-normal-char**(c) — tests whether c is a character that can occur in an identifier (on other positions except beginning)
- **is-digit**(c) — tests whether c is a digit

Implementation of Lexical Analysis

Some other helper functions:

- $\text{CREATE-IDENT}(s)$ — creates an identifier from a given string s
- $\text{CREATE-NUMBER}(s)$ — creates a number from a given string s

Auxiliary variables:

- *last-ident* — the last processed identifier
- *last-num* — the last processed number constant

Function $\text{NEXT-TOKEN}()$ — the main part of the lexical analyser, it returns the following token from the input

Implementation of Lexical Analysis

```
NEXT-TOKEN ():
    while c ∈ {" ", "\t"} do
        c := NEXT-CHAR();
    if c == ⟨eof⟩ then return T_EOF
    else switch c do
        case "(": do c := NEXT-CHAR(); return T_LParen
        case ")": do c := NEXT-CHAR(); return T_RParen
        case "+": do c := NEXT-CHAR(); return T_Plus
        case "−": do c := NEXT-CHAR(); return T_Minus
        case "*": do c := NEXT-CHAR(); return T_Star
        case "/": do c := NEXT-CHAR(); return T_Slash
        otherwise do
            if is-ident-start-char(c) then return SCAN-IDENT()
            else if is-digit(c) then return SCAN-NUMBER()
            else ERROR()
```

Implementation of Lexical Analysis

```
SCAN-IDENT ():
    s := c
    c := NEXT-CHAR()
    while is-ident-normal-char(c) do
        s := s · c
        c := NEXT-CHAR()
    last-ident := CREATE-IDENT(s)
    return T_Ident
```

Implementation of Lexical Analysis

```
SCAN-NUMBER ():
    s := c
    c := NEXT-CHAR()
    while is-digit(c) do
        s := s · c
        c := NEXT-CHAR()
    last-num := CREATE-NUMBER(s)
    return T_Number
```

Implementation of Syntactic Analysis

Variable t :

- the last processed token

A helper function:

- INIT-SCANNER():
  - initializes the lexical analyser
  - reads the first character from the input into variable c, aby tam byl nachystán pro následná volání funkce NEXT-TOKEN()

Reading a next token:

- NEXT-TOKEN():
  - this is the previously described main function of the lexical analyser
  - by repeatedly calling this function we read the tokens
  - variable c always contains the symbol that has been read last

Implementation of Syntactic Analysis

One of the often used methods of syntactic analysis is **recursive descent**:

- For each nonterminal there is a corresponding function — the function corresponding to nonterminal A implements all rules with nonterminal A on the left-hand side.
- In a given function, the next token is used to select between corresponding rules.
- Instructions in the body of a function correspond to processing of right-hand sides of the rules:
  - an occurrence of nonterminal B — the function corresponding to nonterminal B is called
  - an occurrence of terminal a — it is checked that the following token corresponds to terminal a, when it does, the next token is read, otherwise an error is reported

Implementation of Syntactic Analysis

The previously described grammed is not very suitable for the recursive descent because it is not possible for nonterminals E and T to determine in a deterministic way one of the given pair of rules by use of just one following symbol:

$$
\begin{aligned}
S &\rightarrow E\,\langle eof\,\rangle \\
E &\rightarrow T \mid T\,A\,E \\
A &\rightarrow + \mid - \\
T &\rightarrow F \mid F\,M\,T \\
M &\rightarrow * \mid / \\
F &\rightarrow \langle ident\rangle \mid \langle num\,\rangle \mid (\,E\,) \mid -F
\end{aligned}
$$

For example, if we want to rewrite nonterminal T and we know that the following terminal in the input is $\langle num\,\rangle$, this terminal can be generated by use of any of the rules

$$T \rightarrow F \qquad\qquad T \rightarrow F\,M\,T$$

Implementation of Syntactic Analysis

The following modified grammar does not have this problem:

$$
\begin{aligned}
S &\longrightarrow E\,\langle eof \rangle \\
E &\longrightarrow T\,G \\
G &\longrightarrow \varepsilon \mid A\,T\,G \\
A &\longrightarrow \texttt{+} \mid \texttt{-} \\
T &\longrightarrow F\,U \\
U &\longrightarrow \varepsilon \mid M\,F\,U \\
M &\longrightarrow \texttt{*} \mid \texttt{/} \\
F &\longrightarrow \texttt{-}\,F \mid \texttt{(}\,E\,\texttt{)} \mid \langle ident \rangle \mid \langle num \rangle
\end{aligned}
$$

Implementation of Syntactic Analysis

```
PARSE ():
    INIT-SCANNER()
    t := NEXT-TOKEN()
    PARSE-S()
```

$$S \longrightarrow E \, \langle eof \rangle$$

```
PARSE-S ():
    PARSE-E()
    if t ≠ T_EOF then ERROR()
```

Implementation of Syntactic Analysis

$$E \longrightarrow T\ G$$

```
PARSE-E ():
    PARSE-T()
    PARSE-G()
```

$$G \longrightarrow \varepsilon \mid A\ T\ G$$

```
PARSE-G ():
    if t ∈ {T_Plus, T_Minus} then
        PARSE-A()
        PARSE-T()
        PARSE-G()
```

Implementation of Syntactic Analysis

$$T \longrightarrow F\ U$$

```
PARSE-T ():
    PARSE-F()
    PARSE-U()
```

$$U \longrightarrow \varepsilon \mid M\ F\ U$$

```
PARSE-U (e1):
    if t ∈ {T_Star, T_Slash} then
        PARSE-M()
        PARSE-F()
        PARSE-U()
```

$$A \rightarrow \texttt{+} \mid \texttt{-}$$

```
PARSE-A ():
    switch t do
        case T_Plus do
            t := NEXT-TOKEN()
        case T_Minus do
            t := NEXT-TOKEN()
        otherwise do ERROR()
```

Implementation of Syntactic Analysis

$$M \longrightarrow * \mid /$$

```
Parse-M ():
    switch t do
        case T_Star do
            t := next-token()
        case T_Slash do
            t := next-token()
        otherwise do  error()
```

Implementation of Syntactic Analysis

$$
\begin{aligned}
F \;\rightarrow\; & \langle ident \rangle \\
\mid\; & \langle num \rangle \\
\mid\; & (\ E\ ) \\
\mid\; & -\ F
\end{aligned}
$$

```
PARSE-F ():
    switch t do
        case T_Ident do
            t := NEXT-TOKEN()
        case T_Number do
            t := NEXT-TOKEN()
        case T_LParen do
            t := NEXT-TOKEN()
            PARSE-E()
            if t ≠ T_RParen then  ERROR()
            t := NEXT-TOKEN()
        case T_Minus do
            t := NEXT-TOKEN()
            PARSE-F()
        otherwise do  ERROR()
```

Implementation of Syntactic Analysis

- If a function ends with a recursive call of itself, as for example function $\textsc{Parse-G}()$, it is possible to replace this recursion with an iteration.
- Functions $\textsc{Parse-E}()$ and $\textsc{Parse-G}()$ can be merged into one function.
- Similarly, it is possible to replace a recursion with an iteration in function $\textsc{Parse-U}()$, and functions $\textsc{Parse-T}()$ and $\textsc{Parse-U}()$ can be merged into one function.

$$E \longrightarrow T\ G$$
$$G \longrightarrow \varepsilon \mid A\ T\ G$$

```
PARSE-E ():
    PARSE-T()
    while t ∈ {T_Plus, T_Minus} do
        PARSE-A()
        PARSE-T()
```

$$T \longrightarrow F\ U$$
$$U \longrightarrow \varepsilon \mid M\ F\ U$$

```
PARSE-T ():
    PARSE-F()
    while t ∈ {T_Star, T_Slash} do
        PARSE-M()
        PARSE-F()
```

Implementation of Syntactic Analysis

- The implementation described above just finds out whether the given input corresponds to some word that can be generated by the given grammar.
- If this is the case, it reads whole input and finishes successfully.
- If it is not the case, function $\text{ERROR}()$ is called.
- In real implementation, it is useful to provide function $\text{ERROR}()$ with error messages describing the kind of error together with the information about a position in the input where the error occurred (e.g., this line and column where the currently processed token starts).

  Function $\text{ERROR}()$ can use this information to create error messages that are displayed to a user.

Implementation of Syntactic Analysis

- Typically, we do not want to use syntactic analysis just to check that the input is correct but also to create abstract syntax tree or to perform some other types of actions connected with individual rules of the grammar.

- The previously presented code can be used as a base that can be extended with other actions such as construction of an abstract syntax tree, modifications of read expressions, and possibly some other types of computation.

- When the functions that correspond to nonterminals should create the corresponding abstract syntax tree, they can return the constructed subtree, corresponding to the part of the expression generated from the given nonterminal, as a return value.

Implementation of Syntactic Analysis

Construction of an **abstract syntax tree**:

- An enumerated type representing binary arithmetic operations:
  enum *Bin_op* { **Add**, **Sub**, **Mul**, **Div** }
- An enumerated type representing unary arithmetic operations:
  enum *Un_op* { **Un_minus** }
- Functions for creation of different kinds of nodes of an abstract syntax tree:
  - MK-VAR(*ident*) — creates a leaf representing a variable
  - MK-NUM(*num*) — creates a leaf representing a number constant
  - MK-UNARY(*op*, *e*) — creates a node with one child *e*, on which a unary operation *op* (of type *Un_op*) is applied
  - MK-BINARY(*op*, *e1*, *e2*) — creates a node with two children *e1* and *e2*, on which a binary operation *op* (of type *Bin_op*) is applied

Implementation of Syntactic Analysis

$$S \longrightarrow E \, \langle eof \rangle$$

```
PARSE ():
    INIT-SCANNER()
    t := NEXT-TOKEN()
    e := PARSE-E()
    if t ≠ T_EOF then ERROR()
    return e
```

Implementation of Syntactic Analysis

$$E \to T\ G$$
$$G \to \varepsilon \mid A\ T\ G$$

```
PARSE-E():
    e1 := PARSE-T()
    while t ∈ {T_Plus, T_Minus} do
        op := PARSE-A()
        e2 := PARSE-T()
        e1 := MK-BINARY(op, e1, e2)
    return e1
```

Implementation of Syntactic Analysis

$$A \longrightarrow \texttt{+} \mid \texttt{-}$$

```
PARSE-A ():
    switch t do
        case T_Plus do
            t := NEXT-TOKEN()
            return Add
        case T_Minus do
            t := NEXT-TOKEN()
            return Sub
        otherwise do  ERROR()
```

Implementation of Syntactic Analysis

$$T \longrightarrow F\ U$$
$$U \longrightarrow \varepsilon \mid M\ F\ U$$

```
PARSE-T ():
    e1 := PARSE-F()
    while t ∈ {T_Star, T_Slash} do
        op := PARSE-M()
        e2 := PARSE-F()
        e1 := MK-BINARY(op, e1, e2)
    return e1
```

Implementation of Syntactic Analysis

$$M \longrightarrow * \mid /$$

```
PARSE-M ():
    switch t do
        case T_Star do
            t := NEXT-TOKEN()
            return Mul
        case T_Slash do
            t := NEXT-TOKEN()
            return Div
        otherwise do  ERROR()
```

$$
\begin{aligned}
F \rightarrow\ & \langle ident \rangle \\
\mid\ & \langle num \rangle \\
\mid\ & (\ E\ ) \\
\mid\ & -\ F
\end{aligned}
$$

```
PARSE-F ():
  switch t do
    case T_Ident do
      e := MK-VAR(last-ident)
      t := NEXT-TOKEN()
      return e
    case T_Number do
      e := MK-NUM(last-num)
      t := NEXT-TOKEN()
      return e
    case T_LParen do
      t := NEXT-TOKEN()
      e := PARSE-E()
      if t ≠ T_RParen then  ERROR()
      t := NEXT-TOKEN()
      return e
    case T_Minus do
      t := NEXT-TOKEN()
      e := PARSE-F()
      return MK-UNARY(Un_minus, e)
    otherwise do  ERROR()
```