I.2.1. Formal Languages (10.1)

I.2.2. Grammars and Derivations (10.2)
   I.2.2.1. Grammars (10.2.1.)
   I.2.2.2. Examples of Grammars and Strings (10.2.2.)
   I.2.2.3. Derivations (10.2.3.)
   I.2.2.4. Language Generation (10.2.4.)
   I.2.2.5. Designing Syntax using Grammars (10.2.5.)

I.2.3. The Chomsky Hierarchy (12.1)

I.2.4. Modularity and BNF notation (10.3)
   I.2.4.1. A simple modular grammar (10.3.1.)
   I.2.4.2. The import construct (10.3.2.)
   I.2.4.3. BNF Notation (10.3.3.)

I.2.5. Derivation Trees for Context-Free Grammars (14.1)
I.2.1. Formal Languages (10.1)

I.2.2. Grammars and Derivations (10.2)
   I.2.2.1. Grammars (10.2.1.)
   I.2.2.2. Examples of Grammars and Strings (10.2.2.)
   I.2.2.3. Derivations (10.2.3.)
   I.2.2.4. Language Generation (10.2.4.)
   I.2.2.5. Designing Syntax using Grammars (10.2.5.)

I.2.3. The Chomsky Hierarchy (12.1)

I.2.4. Modularity and BNF notation (10.3)
   I.2.4.1. A simple modular grammar (10.3.1.)
   I.2.4.2. The import construct (10.3.2.)
   I.2.4.3. BNF Notation (10.3.3.)

I.2.5. Derivation Trees for Context-Free Grammars (14.1)
Alphabet

Definition
An alphabet is a finite non-empty set $T$. We shall consider the elements of $T$ to be symbols.

Examples
- The alphabet of decimal digits is
  \[ T_{\text{Digit}} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \]
- The lower-case alphabet of the English language is
  \[ T_{\text{English Lowercase Alphabet}} = \{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z\} \]
Alphabet

- The lower-case Welsh alphabet is

\[ T_{Welsh\text{-}Lowercase\text{-}Alphabet} = \{a, b, c, ch, d, dd, e, f, ff, g, ng, h, i, l, ll, m, n, o, p, ph, r, rh, s, t, th, u, w, y\} \]

- (This alphabet has been checked carefully).
- Notice: j, k, q, v, x, z are not elements of this alphabet.
- Therefore “Taxi” written as “Tacsi” in Welsh.

- The Unicode alphabet has over 100,000 characters.
- The Ascii alphabet has 128 characters, of which 33 are non-printing control characters.
String

Definition

A **string** or **word** over an alphabet $T$ is a finite sequence elements of $T$. The set of all strings or words over the alphabet $T$ is

$$T^* := \{ t_1 t_2 \cdots t_n \mid n \geq 0, t_1, t_2, \ldots, t_n \in T \}$$

- Note that one example is the **empty string**, which is represented by $\epsilon$.
- Many languages, e.g. Haskell, identify Strings with lists of characters.
String in Haskell

a :: String
a = "Hello"

b :: String
b = ['H','e','l','l','o']

csetzer@cspcanton:~> ghci
...
Prelude> :load testString.hs
...
*Main> a
"Hello"
*Main> b
"Hello"
Examples of Strings

1984 and 2000 are strings over the alphabet $T_{\text{Digit}}$ of decimal digits.

forwards and sdrawkcab are words over the lower-case English alphabet $T_{\text{English\_Lowercase\_Alphabet}}$. 
Length of a String

**Definition**

The **length** of a string is given by the function

\[
| \_ | : T^* \to \mathbb{N}
\]

\(|w|\) is the number of symbols from the alphabet it contains.

- If we identify strings with \(\text{List}(T)\), it is the length of the corresponding list.

- **Examples**

  \(|2000| = 4\), \(|\text{forwards}| = 8\), \(|\epsilon| = 0\)

- \(T^+\) is the set of non-empty strings:

  \[
  T^+ = T^* \setminus \{\epsilon\} = \{t_1 t_2 \cdots t_n \mid n \geq 1, t_1, \ldots, t_n \in T\}
  \]
Definition

1. The **concatenation** of two strings $u = u_1 \cdots u_m$ and $v = v_1 \cdots v_n$ is the string

   $$uv = u_1, \ldots, u_m, v_1, \ldots, v_n$$

2. The **concatenation function** is the function

   $$\cdot : T^* \times T^* \rightarrow T^*$$

- For instance, if $u = 1984$ and $v = 2000$ then $uv = 19842000$.
- We have

   $$|uv| = |u| + |v|$$
Concatenation

**Definition**

We define \( w^n := \underbrace{ww \cdots w}_{n \text{ times}} \).

For instance

\[ \text{forwards}^3 = \text{forwardsforwardsforwards} \]
A **formal language** $L$ over an alphabet $T$ is a subset $L$ of the set of strings over $T$, i.e.

$$L \subseteq T^*$$

We usually say **language** for formal language.
Example 1

- Let

\[ T_{a,b} = \{a, b\} \]

- We can enumerate \( T_{a,b}^* \) in the following way

\[ T_{a,b}^* = \{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb, \ldots\} \]

- So we write first strings of length 0, then of length 1, then of length 2 etc.
- There is only one string of length 0, namely \( \epsilon \).
- The strings of length \( n + 1 \) are obtained by adding \( a \) in front of each string of length \( n \) and then do the same with \( b \).
- **Exercise** Write a program which enumerates the strings of length \( n \) for given \( n \).
Example 1 (Cont)

Here are a few examples of languages over $T_{a,b}$:

1. $L = \emptyset$.
2. $L = \{a, b\}^*$.
3. $L = \{\epsilon\}$.
4. $L = \{a\}$.
5. $L = \{a, b\}$.
6. $L = \{a^n \mid n \text{ is even}\}$.
7. $L = \{a^n b^n \mid n \geq 0\}$.
8. $L = \{a^n b^{n+1} \mid n \geq 0\}$.
9. $L = \{(ab)^n \mid n \geq 0\}$.
Example 2: URLs

Let

\[ T = \{a, b, \ldots, z, A, B, \ldots, Z, 0, 1, \ldots, 9, -, _, /, ., m, :\}\]

Let

\[ L = \{ w \in T^* \mid w \text{ is an http address} \} \]

\( L \) contains simple addresses such as

http://www.w3.org or http://www.swansea.ac.uk

but complex examples like

http://www.google.com/search?q=swansea+university+computer+science

are not in \( L \), since for example ? and + are not in \( T \).
Example 3: Infix Numbers

Let

\[ T_{\text{Infix\_Arithmetic}} = \{0, 1, +, \cdot, )\} \]

Expressions of type natural numbers are strings such as

- \((0 + 1) + 1\),
- \((1 + 0).0\), etc.
Example 4: Prefix Numbers

Let

\[ T_{\text{Prefix\_Arithmetic}} = \{ \text{zero, succ, add, mult, } , , (, ) \} \]

Expressions of type natural numbers are expressions such as

\[ \text{succ(succ(zero))}, \]
\[ \text{add(zero, succ(succ(zero))}) \]

Let

\[ L_{\text{Prefix\_Arithmetic}} = \{ w \in T_{\text{Prefix\_Arithmetic}}^* \mid w \text{ is an arithmetic expression} \} \]
Tokens

Most parser generators (or compiler-compiler) have two phases:

- In phase one the incoming stream of characters is grouped into simple tokens. This can be words like zero, succ above. Or even longer words considered as one entity, such as `<Identifier>`.
- In phase two the text is parsed using a grammar which refers to the tokens generated in the first phase.
- Token generation will make use of regular expressions, parsing in phase two will use restricted context free grammars.
  - Notions “regular expression” and “context free grammar” will be introduced later.
- Tokens will play the role of the input alphabet for the context free grammar.
  - Token generation is the process of generating a text in the input alphabet of the context free grammar (tokens) from the input alphabet of the language (e.g. Ascii or Unicode characters).
Designing Syntax Using Formal Languages

The syntax of a language is defined in two steps:

1. Choose an alphabet $T$.
2. Define the language $L \subseteq T^*$. 
Recognition Problem

Definition

Let $L \subseteq T^*$ be a formal language over $T$. The \textit{recognition problem} for $T$ is:

Given any $w \in T^*$ decide whether or not $w \in L$. 
I.2.1. Formal Languages (10.1)

I.2.2. Grammars and Derivations (10.2)
  I.2.2.1. Grammars (10.2.1.)
  I.2.2.2. Examples of Grammars and Strings (10.2.2.)
  I.2.2.3. Derivations (10.2.3.)
  I.2.2.4. Language Generation (10.2.4.)
  I.2.2.5. Designing Syntax using Grammars (10.2.5.)

I.2.3. The Chomsky Hierarchy (12.1)

I.2.4. Modularity and BNF notation (10.3)
  I.2.4.1. A simple modular grammar (10.3.1.)
  I.2.4.2. The import construct (10.3.2.)
  I.2.4.3. BNF Notation (10.3.3.)

I.2.5. Derivation Trees for Context-Free Grammars (14.1)
Example: Parse Tree for an English Sentence

A simplified parse tree for the English Sentence “I go home”.
I, go, home, etc. are the **terminals** of the grammar.
They are elements of the alphabet of the language.
Non-terminals

Sentence, NounPhrase, VerbPhrase etc. are **non-terminals**. They are intermediate steps in the grammar. Elements derivable from them form a sublanguage, often a grammatical entity.
Sentence is the **start symbol**.
The language derived is the set of words formed from **non-terminals** derivable from them.
We need as well rules called *productions* which are used to carry out derivations.
E.g.: Sentence $\rightarrow$ NounPhrase $\sqsubseteq$ Verbphrase
Productions

NounPhrase → I
VerbPhrase → Verb NounPhrase
Grammars (10.2.1.)

Definition

A **Grammar** $G = (T, N, S, P)$ consists of

1. a finite set $T$ called the *alphabet* or set of *terminal symbols*,
2. a finite set $N$ of *non-terminal symbols* or *variable symbols* such that $T \cap N = \emptyset$,
3. a special non-terminal symbol $S \in N$ called the *start symbol*,
4. a finite set $P$ of substitution or rewrite rules, called *productions*, each of which has the form $u \rightarrow v$ where
   4.1 The left hand string $u \in (T \cup N)^+$ (esp. $u$ is non-empty),
   4.2 The right hand string $v \in (T \cup N)^\ast$. 
Remark

Note that both left and right hand strings can contain both terminals and non-terminals.
Terminals is another word for elements of the alphabet of a grammar.
Presentation of Grammars

We present a grammar as a 4-tuple $G = (T, N, S, P)$ and also use a displayed version:

| grammar    | $G$ |
| terminals  | $T$ |
| nonterminals | $N$ |
| start symbol | $S$ |
| productions | $P$ |
# Grammar for English Sentence

| grammar | SimpleEnglishGrammarExample |
| terminals | a,b,c,...,z,A,B,...,Z,\[ ] |
| nonterminals | Sentence, Nounphrase, Verbphrase, Verb |
| start symbol | Sentence |
| productions | Sentence $\rightarrow$ Nounphrase $\[ ]$ Verbphrase  
Verbphrase $\rightarrow$ Verb $\[ ]$ Nounphrase  
Verb $\rightarrow$ go  
Nounphrase $\rightarrow$ l  
Nounphrase $\rightarrow$ home |
Derivations (Informal)

A grammar $G$ defines a formal language $L(G) \subseteq T^*$. The elements of $L(G)$ are the set of strings we can obtain as follows:

- Start with start symbol $S$.
- Select a production such that the left hand string is a substring of what you derived so far.
- Replace this substring by the right hand string of the production.
- Once you have obtained a string consisting of non-terminals, possibly stop.

We will first give some examples and then a formal definition of $L(G)$. 
Example 0: English Grammar

The following derives the sentence “I go home”:

Sentence $\Rightarrow$ NounPhrase $\sqsubseteq$ VerbPhrase
$\Rightarrow$ I $\sqsubseteq$ VerbPhrase
$\Rightarrow$ I $\sqsubseteq$ Verb $\sqsubseteq$ NounPhrase
$\Rightarrow$ I $\sqsubseteq$ go $\sqsubseteq$ NounPhrase
$\Rightarrow$ I $\sqsubseteq$ go $\sqsubseteq$ home
### Example 1

Consider the grammar

\[ G^{[01]^*1} = (\{0, 1\}, \{S\}, S, \{S \rightarrow 1, S \rightarrow 0S, S \rightarrow 1S\}) \]

displayed as follows

<table>
<thead>
<tr>
<th>grammar</th>
<th>( G^{[01]^*1} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>terminals</td>
<td>0, 1</td>
</tr>
<tr>
<td>nonterminals</td>
<td>S</td>
</tr>
<tr>
<td>start symbol</td>
<td>S</td>
</tr>
<tr>
<td>productions</td>
<td>( S \rightarrow 1, S \rightarrow 0S, S \rightarrow 1S )</td>
</tr>
</tbody>
</table>
Example 1 (Generation of Strings)

We assign numbers to the production rules:

- Rule 1  \( S \rightarrow 1 \)
- Rule 2  \( S \rightarrow 0S \)
- Rule 3  \( S \rightarrow 1S \)

A derivation of \( 1001 \in L(G^{[01]*1}) \) is as follows:

\[
\begin{align*}
S & \Rightarrow 1S \quad \text{Rule 3} \\
& \Rightarrow 10S \quad \text{Rule 2} \\
& \Rightarrow 100S \quad \text{Rule 2} \\
& \Rightarrow 1001 \quad \text{Rule 1}
\end{align*}
\]

A derivation of \( 011 \in L(G^{[01]*1}) \) is as follows:

\[
\begin{align*}
S & \Rightarrow 0S \quad \text{Rule 2} \\
& \Rightarrow 01S \quad \text{Rule 3} \\
& \Rightarrow 011 \quad \text{Rule 1}
\end{align*}
\]
Derivations of Strings in $G^{\{01\}^*}$

The following shows all strings derivable in this grammar. Note that this is **not** a derivation tree, it's the tree of all possible derivations in this language.

![Diagram of derivations](image-url)
We can see that the elements of $L(G^{01}\ast 1)$ are the strings in $\{0, 1\}^*$ which end with a 1:

$$L(G^{01}\ast 1) = \{w1 \mid w \in \{0, 1\}^*\}$$

**Remark**

$G^{01}\ast 1$ is an example of a regular grammar (this notion will be introduced later).
Example 2 (Grammars)

Consider the grammar

\[
G^{a^n b^n} = (\{a, b\}, \{S\}, S, \{S \rightarrow ab, S \rightarrow aSb\})
\]

displayed as follows

<table>
<thead>
<tr>
<th>grammar</th>
<th>(G^{a^n b^n})</th>
</tr>
</thead>
<tbody>
<tr>
<td>terminals</td>
<td>(a, b)</td>
</tr>
<tr>
<td>nonterminals</td>
<td>(S)</td>
</tr>
<tr>
<td>start symbol</td>
<td>(S)</td>
</tr>
<tr>
<td>productions</td>
<td>(S \rightarrow ab, S \rightarrow aSb)</td>
</tr>
</tbody>
</table>
Derivations of strings in $G^{a_n b_n}$

Note again that this is not a derivation tree, but a tree determining all possible derivable strings in this language.
We can see that the elements of $L(G^{anbn})$ are the strings of the form $a^n b^n$: 

$$L(G^{anbn}) = \{a^n b^n \mid n \geq 1\}$$

Remark

$G^{anbn}$ is an example of a context-free grammar (this notion will be introduced later).
Example 3 (Grammars)

Consider the grammar

\[ G^{a^n b^n c^n} = (\{a, b, c\}, \{S, B, C\}, S, \]
\[ \{S \rightarrow aSBC, S \rightarrow aBC, CB \rightarrow BC, \]
\[ aB \rightarrow ab, bB \rightarrow bb, bC \rightarrow bc, cC \rightarrow cc\} ) \]

displayed as follows

<table>
<thead>
<tr>
<th>grammar</th>
<th>[ G^{a^n b^n c^n} ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>terminals</td>
<td>[ a, b, c ]</td>
</tr>
<tr>
<td>nonterminals</td>
<td>[ S, B, C ]</td>
</tr>
<tr>
<td>start symbol</td>
<td>[ S ]</td>
</tr>
</tbody>
</table>
| productions | \[ S \rightarrow aSBC, S \rightarrow aBC, \]
|             | \[ CB \rightarrow BC, \]
|             | \[ aB \rightarrow ab, bB \rightarrow bb, bC \rightarrow bc, cC \rightarrow cc. \] |
A derivation of $aabbcc$ in this grammar is as follows:

$$
\begin{align*}
S & \Rightarrow aSBC \\
& \Rightarrow aaBCBC \\
& \Rightarrow aaBBCC \\
& \Rightarrow aabBCC \\
& \Rightarrow aabbCC \\
& \Rightarrow aabbcC \\
& \Rightarrow aabbcc
\end{align*}
$$
A derivation of \textit{aaabbbccc} will now be shown on the black/whiteboard.
No Parse Trees for Context Sensitive Grammars

Note that for context sensitive grammars and unrestricted grammars we **don’t get parse trees**. They exist in general only for **context free and regular grammars**.
We can see that the elements of $L(G^{a^n b^n c^n})$ are the strings of the form $a^n b^n c^n$:

$$L(G^{a^n b^n c^n}) = \{ a^n b^n c^n \mid n \geq 1 \}$$

**Remark**

- $G^{a^n b^n c^n}$ is often in books and webpages presented as an example of a context-sensitive grammar (this notion will be introduced later).

- It is not, but can be transformed into a context sensitive one, see Sect. 1.2.3. Example 5.
I.2.2.3. Derivations (10.2.3.)

Definition

1. Let $G = (T, N, S, P)$ be a grammar, $w \in (T \cup N)^+$ be a non-empty word, $w' \in (T \cup N)^*$ be a possibly empty word.

We say that $w'$ is derived from $w$ in one step, with notation $w \Rightarrow G w'$ iff there exists a production $u \rightarrow v \in P$ such that

- $u$ occurs in $w$,
- $w'$ is the result of replacing one occurrence of $u$ in $w$ by $v$. 
2. We say as well \( w' \) is immediately generated from \( w \) or \( w' \) is one-step generated from \( w \), or

\[
\Rightarrow
\]

\[
\Rightarrow
\]

for \( w \Rightarrow_G w' \).

- So \( w \) has the form \( sut \) for some \( s, t \in (T \cup N)^* \), and \( w' = svt \) where \( u \longrightarrow v \) is a production.

- Informally \( w' \) is the result of replacing in \( w \) a substring, which is equal to the right hand string of a production by the left hand string of that production.
3. Let $G = (T, N, S, P)$ be a grammar, $w, w' \in (T \cup N)^*$ be a possibly empty words.

We say that $w'$ is derived from $w$, with notation $w \Rightarrow^*_G w'$ iff

- either $w = w'$
- or there exists a sequence of words $w_0, \ldots, w_n \in (T \cup N)^*$ s.t.
  - $w_0 = w$,
  - $w_n = w'$
  - for $1 \leq i \leq n - 1$ we have $w_i \Rightarrow_G w_{i+1}$. 
4. We say as well \( w' \) is generated from \( w \) or

\[
\Rightarrow^* \\
\]

for \( w \Rightarrow_G^* w' \).
Example

In example 3 above we had the following derivation:

\[
S \Rightarrow aSBC \\
\Rightarrow aaBCBC \\
\Rightarrow aaBBCC \\
\Rightarrow aabBCC \\
\Rightarrow aabbCC \\
\Rightarrow aabbcC \\
\Rightarrow aabbcc
\]

Therefore we have for instance

\[
S \Rightarrow^* aabbcC \\
S \Rightarrow^* aabBCC \\
aaBCBC \Rightarrow^* aabBCC \\
aaBCBC \Rightarrow^* aaBCBC
\]
Remark

The above definitions introduced relations

\[ \Rightarrow_G \subseteq (T \cup N)^+ \times (T \cup N)^* \]
\[ \Rightarrow_G^* \subseteq (T \cup N)^* \times (T \cup N)^* \]

\[ \Rightarrow_G^* \text{ is the reflexive and transitive closure of } \Rightarrow_G \]
**Language Generation (10.2.4.)**

**Definition**

Let $G = (T, N, S, P)$ be a grammar. The **language generated by the grammar** $G$, denoted by $L(G) \subseteq T^*$ is defined as the set of terminal strings generated from the start symbol, i.e.

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

**Remark**

*If a language is generated by a grammar, then there are infinitely many grammars which generate the same language.*
Equivalence of Grammars

**Definition**

We say that two grammars $G_1$ and $G_2$ are equivalent iff

$$L(G_1) = L(G_2)$$
Example

We show that

\[ L(G^{a^n b^n}) = \{a^n b^n \mid n \geq 1\} \]

One can easily show that

\[ S \Rightarrow^* t \iff \exists n \geq 0 (t = a^n S b^n \lor t = a^{n+1} b^{n+1}) \]

- “⇒” follows by induction on length of the derivation \( S \Rightarrow^* t \).
- “⇐”: show first the assertion for \( t = a^n S b^n \) by induction on \( n \). Then the assertion for \( t = a^{n+1} b^{n+1} \) follows as well.
For programming languages $L$ it is usually difficult to find a grammar $G$ s.t. $L = L(G)$.

- One problem is the fact that variables usually need to be defined before being used.
  - We will later see that languages with such a property can usually not be defined by a context-free grammar.
- Without any restrictions on the grammar it is possible, but languages with unrestricted grammars are difficult to parse.
- Instead one defines first a grammar which reflects the basic structure of the language. Then one selects those strings which fulfil the additional criteria.
Designing Syntax using Grammars

In order to define a language $L$ one usually proceeds as follows:

**Step 1** Choose an alphabet s.t. $L \subseteq T^*$.

**Step 2** Choose a simple grammar $G$ with alphabet $T$ s.t.

$$L \subseteq L(G) \subseteq T^*$$

The grammar should give a good explanation of how the language is formed.

**Step 3** Define an algorithm which determines for $t \in L(G)$ whether $t \in L$ or not.
I.2.1. Formal Languages (10.1)

I.2.2. Grammars and Derivations (10.2)
   I.2.2.1. Grammars (10.2.1.)
   I.2.2.2. Examples of Grammars and Strings (10.2.2.)
   I.2.2.3. Derivations (10.2.3.)
   I.2.2.4. Language Generation (10.2.4.)
   I.2.2.5. Designing Syntax using Grammars (10.2.5.)

I.2.3. The Chomsky Hierarchy (12.1)

I.2.4. Modularity and BNF notation (10.3)
   I.2.4.1. A simple modular grammar (10.3.1.)
   I.2.4.2. The import construct (10.3.2.)
   I.2.4.3. BNF Notation (10.3.3.)

I.2.5. Derivation Trees for Context-Free Grammars (14.1)
Chomsky Hierarchy

The **Chomsky hierarchy** is the classification of grammars by means of 4 properties of its production rules:

- **Unrestricted grammars.**
  - The limit of grammars.

- **Context-sensitive grammars.**
  - It’s usually an accident if a grammar of a language is context-sensitive. C and C++ have some context-sensitive aspects (dealt with by selecting correct strings after the parsing).

- **Context-free grammars.**
  - Easy to understand and supported by parse generators.
  - In language design one aims at languages having an underlying context-free grammar.

- **Regular grammars.**
  - Simple to parse. Used for dividing the input stream of characters into tokens.
Unrestricted Grammars

Let in the following four definitions \( G = (T, N, S, P) \) be a grammar.

**Definition**

Any grammar \( G \) is of **Type 0** or **unrestricted**, so any production

\[
u \longrightarrow v
\]

for \( u \in (T \cup N)^+ \), \( v \in (T \cup N)^* \) are allowed.
Context-Sensitive Grammars

Definition

Any grammar $G$ is of **Type 1** or **context-sensitive**, if all its productions have the form

$$uAv \rightarrow uwv$$

where $A \in N$ is a nonterminal, which rewrites to a non-empty string $w \in (T \cup N)^+$, but only where $A$ is in the context of strings $u, v \in (T \cup N)^*$. Furthermore a production

$$A \rightarrow \epsilon$$

is allowed, but only if $A$ does not occur in the right hand side of any production.
Context-Free Grammars

Definition

Any grammar $G$ is of **Type 2** or **context-free**, if all its productions have the form

$$A \rightarrow w$$

where $A \in N$ is a nonterminal, which rewrites to a string $w \in (T \cup N)^*$. 
Regular Grammars

Definition

1. A grammar $G$ is **left-linear**, iff all its productions have the form

   $$A \rightarrow Ba \text{ or } A \rightarrow a \text{ or } A \rightarrow \epsilon$$

2. A grammar $G$ is **right-linear**, iff all its productions have the form

   $$A \rightarrow aB \text{ or } A \rightarrow a \text{ or } A \rightarrow \epsilon$$

3. A grammar $G$ is of **Type 3** or **regular**, iff it is left-linear or right-linear

In the above we have $A, B \in N$ are nonterminal and $a \in T$.

Note that in a regular grammar either all productions must be left-linear or all productions must be right-linear, so no mixing of the left-linear and right-linear is allowed.
A Hierarchy of Languages

Definition
A language \( L \subseteq T^* \) is **unrestricted**, **context-sensitive**, **context-free**, or **regular**, iff there exists a grammar \( G \) of the relevant type such that \( L(G) = L \).

Remark
*For any \( L \) we have \( L \) regular \( \Rightarrow \) \( L \) context-free \( \Rightarrow \) \( L \) context-sensitive \( \Rightarrow \) \( L \) unrestricted.*
We have that

- every regular grammar is context-free.
- every context-sensitive grammar is an unrestricted grammar.

However not every context-free grammar is context sensitive, since context-sensitive languages allow only productions $A \rightarrow \epsilon$ if $A$ does not occur at the right hand side of a production. (Otherwise all unrestricted languages would be context-sensitive).

However one can construct from a context-free grammar a context-free grammar of the same language, which has only productions $A \rightarrow \epsilon$, if $A$ does not occur on the right hand side of a production. This grammar is therefore context-sensitive as well.
Hierarchy of Languages

regular ⊂ context-free ⊂ context sensitive ⊂ unrestricted
Examples of Equivalent Grammars

We give grammars of each type for defining the language

\[ L^{a^{2n}} := \{ a^i \mid i \text{ is even} \} \]
### Unrestricted Grammar for $L_{a^{2n}}$

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>grammar</td>
<td>$G_{\text{unrestricted, } a^{2n}}$</td>
</tr>
<tr>
<td>terminals</td>
<td>$a$</td>
</tr>
<tr>
<td>nonterminals</td>
<td>$S$</td>
</tr>
<tr>
<td>start symbol</td>
<td>$S$</td>
</tr>
<tr>
<td>productions</td>
<td>$S \rightarrow \epsilon$</td>
</tr>
<tr>
<td></td>
<td>$S \rightarrow aa$</td>
</tr>
<tr>
<td></td>
<td>$a \rightarrow aaa$</td>
</tr>
</tbody>
</table>
Context-Sensitive Grammar for $L^{a^{2n}}$

<table>
<thead>
<tr>
<th>grammar</th>
<th>$G_{\text{context-sensitive, } a^{2n}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>terminals</td>
<td>$a$</td>
</tr>
<tr>
<td>nonterminals</td>
<td>$S, T$</td>
</tr>
<tr>
<td>start symbol</td>
<td>$S$</td>
</tr>
<tr>
<td>productions</td>
<td>$S \rightarrow \epsilon$</td>
</tr>
<tr>
<td></td>
<td>$S \rightarrow aa$</td>
</tr>
<tr>
<td></td>
<td>$S \rightarrow aaT$</td>
</tr>
<tr>
<td></td>
<td>$aT \rightarrow aTaa$</td>
</tr>
<tr>
<td></td>
<td>$aT \rightarrow aaa$</td>
</tr>
</tbody>
</table>
# Context-Free Grammar for $L^{a^{2n}}$

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grammar</td>
<td>$G_{context-free,a^{2n}}$</td>
</tr>
<tr>
<td>Terminals</td>
<td>$a$</td>
</tr>
<tr>
<td>Nonterminals</td>
<td>$S$</td>
</tr>
<tr>
<td>Start symbol</td>
<td>$S$</td>
</tr>
<tr>
<td>Productions</td>
<td>$S \rightarrow \epsilon$</td>
</tr>
<tr>
<td></td>
<td>$S \rightarrow aSa$</td>
</tr>
</tbody>
</table>
Regular Grammar for $L_{a^{2n}}$

<table>
<thead>
<tr>
<th>grammar</th>
<th>$G_{\text{regular},a^{2n}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>terminals</td>
<td>$a$</td>
</tr>
<tr>
<td>nonterminals</td>
<td>$S, A$</td>
</tr>
<tr>
<td>start symbol</td>
<td>$S$</td>
</tr>
<tr>
<td>productions</td>
<td>$S \rightarrow \epsilon$</td>
</tr>
<tr>
<td></td>
<td>$S \rightarrow aA$</td>
</tr>
<tr>
<td></td>
<td>$A \rightarrow aS$</td>
</tr>
</tbody>
</table>
### Example 1 (Grammars of the Levels of the Chomsky Hierarchy)

<table>
<thead>
<tr>
<th>grammar</th>
<th>$G$</th>
</tr>
</thead>
<tbody>
<tr>
<td>terminals</td>
<td>$a, b$</td>
</tr>
<tr>
<td>nonterminals</td>
<td>$S$</td>
</tr>
<tr>
<td>start symbol</td>
<td>$S$</td>
</tr>
<tr>
<td>productions</td>
<td>$S \rightarrow aSa, S \rightarrow bSb, S \rightarrow \epsilon$</td>
</tr>
</tbody>
</table>

$L(G) = ?$

$G$ is of which type?
## Example 2

<table>
<thead>
<tr>
<th>Grammar</th>
<th>$G$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Terminals</td>
<td>$a$</td>
</tr>
<tr>
<td>Nonterminals</td>
<td>$S$</td>
</tr>
<tr>
<td>Start Symbol</td>
<td>$S$</td>
</tr>
<tr>
<td>Productions</td>
<td>$S \rightarrow a, S \rightarrow aS$</td>
</tr>
</tbody>
</table>

$L(G) = \square$

$G$ is of which type?
Example 3

<table>
<thead>
<tr>
<th>grammar</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>terminals</td>
<td>a, b</td>
</tr>
<tr>
<td>nonterminals</td>
<td>S</td>
</tr>
<tr>
<td>start symbol</td>
<td>S</td>
</tr>
<tr>
<td>productions</td>
<td>S → ab, S → aSb</td>
</tr>
</tbody>
</table>

$L(G) = ?$

$G$ is of which type?
Consider the grammar

<table>
<thead>
<tr>
<th>grammar</th>
<th>$G^{a^n b^n c^n}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>terminals</td>
<td>$a, b, c$</td>
</tr>
<tr>
<td>nonterminals</td>
<td>$S, B, C$</td>
</tr>
<tr>
<td>start symbol</td>
<td>$S$</td>
</tr>
</tbody>
</table>
| productions | $S \rightarrow aSBC, S \rightarrow aBC,$  
              | $CB \rightarrow BC,$       
              | $aB \rightarrow ab, bB \rightarrow bb, bC \rightarrow bc, cC \rightarrow cc.$ |

Example 4 is not a context sensitive grammar. Why?
Example 5 (Grammars)

Here is a variant which is context sensitive:

- **grammar**: \( G^{a^n b^n c^n} \)
- **terminals**: \( a, b, c \)
- **nonterminals**: \( S, B, C, H \)
- **start symbol**: \( S \)
- **productions**:
  - \( S \rightarrow aSBC \), \( S \rightarrow aBC \),
  - \( CB \rightarrow HB \), \( HB \rightarrow HC \), \( HC \rightarrow BC \),
  - \( aB \rightarrow ab \), \( bB \rightarrow bb \), \( bC \rightarrow bc \), \( cC \rightarrow cc \).
A derivation of $aabbcc$ in Example 5 is as follows:

$$
S \Rightarrow aSBC \\
\Rightarrow aaBCBC \\
\Rightarrow aaBHBC \\
\Rightarrow aaBHCC \\
\Rightarrow aaBBCC \\
\Rightarrow aabBCC \\
\Rightarrow aabbCC \\
\Rightarrow aabbcC \\
\Rightarrow aabbcc
$$
Examples

- Regular: \( \{ a^n \mid n \geq 1 \} \)
- Context-free: \( \{ a^n b^n \mid n \geq 1 \} \)
- Context-sensitive: \( \{ a^n b^n c^n \mid n \geq 1 \} \)
- Unrestricted
I.2.1. Formal Languages (10.1)

I.2.2. Grammars and Derivations (10.2)
   I.2.2.1. Grammars (10.2.1.)
   I.2.2.2. Examples of Grammars and Strings (10.2.2.)
   I.2.2.3. Derivations (10.2.3.)
   I.2.2.4. Language Generation (10.2.4.)
   I.2.2.5. Designing Syntax using Grammars (10.2.5.)

I.2.3. The Chomsky Hierarchy (12.1)

I.2.4. Modularity and BNF notation (10.3)
   I.2.4.1. A simple modular grammar (10.3.1.)
   I.2.4.2. The import construct (10.3.2.)
   I.2.4.3. BNF Notation (10.3.3.)

I.2.5. Derivation Trees for Context-Free Grammars (14.1)
Specifying Syntax using Grammars: Modularity and BNF Notation (10.3)
A Simple Modular Grammar for a Programming Language (10.3.1.)

We introduce a grammar for a simple while language for computing natural numbers. It will have the following components:

- identifiers,
- natural numbers,
- arithmetic expressions,
- Boolean expressions,
- programs.
### Grammar for Identifiers

<table>
<thead>
<tr>
<th>grammar</th>
<th>$G_{\text{Identifier}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>terminals</td>
<td>a, b, \ldots, z, A, B, \ldots, Z</td>
</tr>
<tr>
<td>nonterminals</td>
<td>Letter, Id</td>
</tr>
<tr>
<td>start symbol</td>
<td>Id</td>
</tr>
<tr>
<td>productions</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Id $\longrightarrow$ Letter</td>
</tr>
<tr>
<td></td>
<td>Id $\longrightarrow$ Letter Id</td>
</tr>
<tr>
<td></td>
<td>Letter $\longrightarrow$ a</td>
</tr>
<tr>
<td></td>
<td>Letter $\longrightarrow$ b</td>
</tr>
<tr>
<td></td>
<td>\ldots</td>
</tr>
<tr>
<td></td>
<td>Letter $\longrightarrow$ z</td>
</tr>
<tr>
<td></td>
<td>Letter $\longrightarrow$ A</td>
</tr>
<tr>
<td></td>
<td>Letter $\longrightarrow$ B</td>
</tr>
<tr>
<td></td>
<td>\ldots</td>
</tr>
<tr>
<td></td>
<td>Letter $\longrightarrow$ Z</td>
</tr>
</tbody>
</table>
# Grammar for Numbers

<table>
<thead>
<tr>
<th>Grammar</th>
<th>$G^{Number}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Terminals</td>
<td>0, 1, \ldots, 9</td>
</tr>
<tr>
<td>Nonterminals</td>
<td>$Number$, $Digit$</td>
</tr>
<tr>
<td>Start symbol</td>
<td>$Number$</td>
</tr>
</tbody>
</table>
| Productions      | $Number \rightarrow Digit$
|                  | $Number \rightarrow Digit \ Number$
|                  | $Digit \rightarrow 0$
|                  | $Digit \rightarrow 1$
|                  | \ldots
|                  | $Digit \rightarrow 9$ |
The grammar for arithmetic expressions $G^{\text{Arithmetic Expression}}$ will import $G^{\text{Identifier}}$, $G^{\text{Number}}$.

This means the following:

- The terminals/nonterminals/productions of these grammars are added to the terminals/nonterminals/productions of $G^{\text{Arithmetic Expression}}$. 

Grammar for Arithmetic Expressions

<table>
<thead>
<tr>
<th>grammar</th>
<th>$G_{Arithmetic_Expression}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>import</td>
<td>$G_{Identifier}, G_{Number}$</td>
</tr>
<tr>
<td>terminals</td>
<td>$+, -, *, /, (, )$</td>
</tr>
<tr>
<td>nonterminals</td>
<td>$AExp, AOp$</td>
</tr>
<tr>
<td>start symbol</td>
<td>$AExp$</td>
</tr>
<tr>
<td>productions</td>
<td>$AExp \rightarrow Id$</td>
</tr>
<tr>
<td></td>
<td>$AExp \rightarrow Number$</td>
</tr>
<tr>
<td></td>
<td>$AExp \rightarrow ( AExp )$</td>
</tr>
<tr>
<td></td>
<td>$AExp \rightarrow AExp AOp AExp$</td>
</tr>
<tr>
<td></td>
<td>$AOp \rightarrow +$</td>
</tr>
<tr>
<td></td>
<td>$AOp \rightarrow -$</td>
</tr>
<tr>
<td></td>
<td>$AOp \rightarrow *$</td>
</tr>
<tr>
<td></td>
<td>$AOp \rightarrow /$</td>
</tr>
</tbody>
</table>
Ambiguity

The grammar $G^{\text{Arithmetic Expression}}$ is ambiguous, a concept discussed in Section I.2.5.

- There we will discuss how to make this grammar unambiguous.

The same applies to the grammars

- $G^{\text{Boolean Expression}}$

- and $G^{\text{while}}$ (because of the sequencing operation) introduced on the next two slides.
# Grammar for Boolean Expressions

**Grammar**

\[ G_{\text{Boolean Expression}} \]

**Import**

\[ G_{\text{Arithmetic Expression}} \]

**Terminals**

true, false, not, and, or, =, <

**Nonterminals**

\( BExp, BOp1, BOp2, RelOp \)

**Start Symbol**

\( BExp \)

**Productions**

\[
\begin{align*}
BExp & \rightarrow BOp1 BExp \\
BExp & \rightarrow BExp \langle BOp2 \rangle BExp \\
BExp & \rightarrow AExp RelOp AExp \\
BExp & \rightarrow true \\
BExp & \rightarrow false \\
BOp1 & \rightarrow not \\
BOp2 & \rightarrow and \\
BOp2 & \rightarrow or \\
RelOp & \rightarrow = \\
RelOp & \rightarrow <
\end{align*}
\]
## Grammar for While Programs

<table>
<thead>
<tr>
<th>grammar</th>
<th>( G_{\text{while}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>import</td>
<td>( G_{\text{Arithmetic} _ \text{Expression}}, G_{\text{Boolean} _ \text{Expression}} )</td>
</tr>
<tr>
<td>terminals</td>
<td>skip, if, then, else, fi, while, do, od, :=, ;</td>
</tr>
<tr>
<td>nonterminals</td>
<td>Program</td>
</tr>
<tr>
<td>start symbol</td>
<td>Program</td>
</tr>
<tr>
<td>productions</td>
<td>Program ( \rightarrow ) skip</td>
</tr>
<tr>
<td></td>
<td>Program ( \rightarrow ) Id := AExp</td>
</tr>
<tr>
<td></td>
<td>Program ( \rightarrow ) Program ; Program</td>
</tr>
<tr>
<td></td>
<td>Program ( \rightarrow ) if BExp then Program else Program fi</td>
</tr>
<tr>
<td></td>
<td>Program ( \rightarrow ) while BExp do Program od</td>
</tr>
</tbody>
</table>
Definition

The **Backus-Naur-Form** or **BNF** is the presentation of a grammar using the following conventions:

1. The terminal symbols are often written in bold font.
2. The non-terminal symbols are familiar terms for syntactic components enclosed in angle brackets, e.g. `<statement>`, `<expression>`, `<identifier>`.
3. The start symbol is the non-terminal that is presented first.
4. The symbol `::=` replaces and extends `→` by listing the productions possible for a non-terminal; alternative possibilities for the right hand sides of a particular production are separated by `|`. 
Example 1

For instance the rules

\[
\begin{align*}
\langle BExp \rangle & ::= \text{true} \\
\langle BExp \rangle & ::= \text{false} \\
\langle BExp \rangle & ::= \langle BOp1 \rangle \langle BExp \rangle \\
\langle BExp \rangle & ::= \langle BExp \rangle \langle BOp2 \rangle \langle BExp \rangle \\
\langle BExp \rangle & ::= \langle AExp \rangle \langle RelOp \rangle \langle AExp \rangle
\end{align*}
\]

are condensed into one rule

\[
\langle BExp \rangle ::= \text{true} \mid \text{false} \mid \\
\langle BOp1 \rangle \langle BExp \rangle \mid \langle BExp \rangle \langle BOp2 \rangle \langle BExp \rangle \mid \langle AExp \rangle \ RelOp \langle AExp \rangle
\]
Example 2

**bnf**  

**Letter**

**rules**

\[
\langle \text{Letter} \rangle ::= \langle \text{LowerCase} \rangle | \langle \text{UpperCase} \rangle
\]

\[
\langle \text{LowerCase} \rangle ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z
\]

\[
\langle \text{UpperCase} \rangle ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z
\]
Extended BNF

Definition

Extended BNF or EBNF adds to BNF the following conventions:

1. An optional occurrence of some portion of a production rule choice is enclosed in square brackets $[\cdot]$. 
   $[u]$ means zero or one occurrence of $u$ where $u \in (T \cup N)^+$. 

2. An arbitrary number of occurrences of some portion of a production rule choice is enclosed in braces $\{\cdot\}$. 
   $\{u\}$ means zero or more occurrences of $u$ where $u \in (T \cup N)^+$. 

Example

In normal BNF, numbers can be defined as follows

<table>
<thead>
<tr>
<th>bnf</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>rules</td>
<td></td>
</tr>
<tr>
<td>⟨Number⟩</td>
<td>::= ⟨Digit⟩</td>
</tr>
<tr>
<td>⟨Digits⟩</td>
<td>::= ⟨Digit⟩</td>
</tr>
<tr>
<td>⟨Digit⟩</td>
<td>::= 0</td>
</tr>
<tr>
<td>⟨NonZeroDigit⟩</td>
<td>::= 1</td>
</tr>
</tbody>
</table>
Example

In EBNF, we can define them as follows:

\[
\text{ebnf} \\
\text{rules} \\
\langle \text{Number} \rangle \quad ::= \quad 0 \mid \langle \text{NonZeroDigit} \rangle \{\langle \text{Digit} \rangle\} \\
\langle \text{Digit} \rangle \quad ::= \quad 0 \mid \langle \text{NonZeroDigit} \rangle \\
\langle \text{NonZeroDigit} \rangle \quad ::= \quad 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\]
Theorem I.2.4.3.1

Lemma

Let $L \subseteq T^*$ be a language. $L$ is definable in EBNF iff it is definable in BNF.

Proof: Exercise.
I.2.5. Derivation Trees for Context-Free Grammars (14.1)

I.2.1. Formal Languages (10.1)

I.2.2. Grammars and Derivations (10.2)
   I.2.2.1. Grammars (10.2.1.)
   I.2.2.2. Examples of Grammars and Strings (10.2.2.)
   I.2.2.3. Derivations (10.2.3.)
   I.2.2.4. Language Generation (10.2.4.)
   I.2.2.5. Designing Syntax using Grammars (10.2.5.)

I.2.3. The Chomsky Hierarchy (12.1)

I.2.4. Modularity and BNF notation (10.3)
   I.2.4.1. A simple modular grammar (10.3.1.)
   I.2.4.2. The import construct (10.3.2.)
   I.2.4.3. BNF Notation (10.3.3.)

I.2.5. Derivation Trees for Context-Free Grammars (14.1)
Derivation Trees or Parse Trees

- Context free Grammars (abbreviated as \textit{CFG} in the following) allow to apply to a non-terminal at position without needing the context.
- Therefore we can expand the non-terminals independently of each other.
- This allows us to define derivation trees (also called parse trees).
Example

Consider the grammar

| grammar     | G |
| terminals   | a, b |
| nonterminals | S |
| start symbol | S |
| productions | S → aSb  
              S → ab |


Example Derivation

We derive $aaabbbb$ in it:

\[
S \Rightarrow aSb \\
\Rightarrow aaSbb \\
\Rightarrow aaabbb
\]
Derivation Tree

\[ S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaabbb \]
Form of the Derivation Tree

- Nodes are labelled with elements of $N \cup T \cup \{\epsilon\}$.
- A node with label $A$ has a subtree
  
  \[ 
  \begin{array}{c}
  A \\
  \ \ \ X_1 \\
  \ \ \ \ ...
  \\
  \ \ \ X_2 \\
  \ \ \ \ ...
  \\
  \ \ \ X_n \\
  \end{array}
  \]

  only if $A$ is a non-terminal and there is a production

  \[ A \rightarrow X_1 X_2 \cdots X_n \]

  where $X_i \in T \cup N$.
- All leaves of the tree together read from left to right form the string derived, namely $aaabbb$.
  This is called the **frontier** of the derivation tree.
- We will as well consider derivation trees not ending in a string of terminals, so the frontier is an element of $(T \cup N)^*$. 
Definition Derivation Tree

Let $G = (T, N, S, P)$ be a CFG. A **derivation tree** or **parse tree** for $G$ is a finite tree with

- nodes labelled by elements of $N \cup T \cup \{\epsilon\}$,
- s.t. a node $A$ has children with labels $X_1, \ldots, X_n$ only if $A \in N$ and there is a production

$$A \rightarrow X_1X_2 \cdots X_n$$

- If the node of one of the children of $A$ is $\epsilon$, then this node is the only child of this tree.

The **frontier** of the tree is the set of leaves read from left to right in sequence, which is an element $(T \cup N)^*$. The **root** of the tree is the node at the top of the derivation tree.
Left-Most and Right-Most Derivations

From a derivation tree we can obtain a derivation in various orders. Consider the grammar

```
grammar G

terminals a, b

nonterminals S, A, B

start symbol S

productions S → AB,
A → aAa, A → a
B → bBb, B → b
```
Example Derivation Tree
Different Derivations of \textit{aaabbb}

We can derive \textit{aaabbb} in different ways:

\begin{align*}
S & \Rightarrow AB \Rightarrow aAaB \Rightarrow aaaB \quad \Rightarrow aaabBb \Rightarrow aaabbb \\
\text{A \underline{left most derivation}} \\
S & \Rightarrow AB \Rightarrow AbBb \Rightarrow Abbb \quad \Rightarrow aAabbb \Rightarrow aaabbb \\
\text{A \underline{right most derivation}} \\
S & \Rightarrow AB \Rightarrow aAaB \Rightarrow aAabBb \Rightarrow aaabBb \Rightarrow aaabbb \\
S & \Rightarrow AB \Rightarrow aAaB \Rightarrow aAabBb \Rightarrow aAabbb \Rightarrow aaabbb \\
S & \Rightarrow AB \Rightarrow AbBb \Rightarrow aAabBb \Rightarrow aaabBb \Rightarrow aaabbb \\
S & \Rightarrow AB \Rightarrow AbBb \Rightarrow aAabBb \Rightarrow aAabbb \Rightarrow aaabbb
\end{align*}
Left-Most Derivation

Definition

Let $G = (T, N, S, P)$ be a CFG.
A single-step derivation $w \Rightarrow w'$ is **left-most** if a rule was applied to the left-most non-terminal in $w$, i.e.

- $w = sAt$ for some $A \in N$, $s \in T^*$ (consisting only of terminals), $t \in (S \cup T)^*$,
- and there exist a production $A \rightarrow v$
- s.t. $w' = svt$. 
Left-Most Derivation

Definition

Let $G = (T, N, S, P)$ be a CFG. A single-step derivation $w \Rightarrow w'$ is **right-most** if a rule was applied to the right-most non-terminal in $w$, i.e.

- $w = sAt$ for some $A \in N$, $s \in (S \cup T)^*$, $t \in T^*$ (consisting only of terminals),
- there exist a production $A \rightarrow v$
- s.t. $w' = svt$. 
Left/Right-Most Derivation Sequence

Definition

Let $G = (T, N, S, P)$ be a CFG

1. A derivation sequence $w_0 \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \cdots \Rightarrow w_n$ is left-most, if each derivation step $w_i \Rightarrow w_{i+1}$ is left-most.

2. Right-most derivation sequences are defined analogously.
Theorem I.2.5.1. (Derivation Trees and Language Generation)

Let $G = (T, N, S, P)$ be a CFG, $A \in T$, $w, w' \in (T \cup N)^*$, Then the following are equivalent

1. There exist a derivation tree with root labelled by $A$ and frontier $w'$.

2. $A \Rightarrow^* w'$.

In case $w' \in T^*$, the derivation sequence $w \Rightarrow^* w'$ can both be chosen as a left-most and as a right-most derivation sequence.
Proof of Theorem I.2.5.1.

- A proof of this theorem can be found in the additional material.
- We illustrate this theorem by an example. It will show both:
  - (read forwards) how to obtain from a left-most derivation a derivation tree,
  - (read backwards) how to obtain from a derivation tree a left-most derivation.
Example (Step 1)

```
S
```

```
S    S
  /   /
A    B
 /    /    /
A    a    B
 /    /    /    /
 a    a    b    b
```

```
S    S
  /   /
A    a
 /    /
 a    /
  /    
   a   b
```
Example (Step 2)

\[ S \Rightarrow AB \]
Example (Step 3)

\[ S \Rightarrow AB \Rightarrow aAaB \]

Derivation tree for \( a \) is trivial.
Example (Step 4)

\[ S \Rightarrow AB \Rightarrow aAaB \Rightarrow aaaB \]

Diagram:

- **S**
  - **A**
    - a
    - **A**
      - a
      - a
  - a
- **B**
  - b
  - **B**
    - b
    - b
Example (Step 5)

\[ S \Rightarrow AB \Rightarrow aAaB \Rightarrow aaaB \Rightarrow aaabBb \]
Example (Step 6)

\[ S \Rightarrow AB \Rightarrow aAaB \Rightarrow aaaB \Rightarrow aaabBb \Rightarrow aaabbb \]

Final derivation.
Theorem I.2.5.3. Uniqueness of Derivation (Trees)

Let $G = (T, N, S, P)$ be a CFG, $A \in N$, $w \in T^*$.

1. Assume there are two different derivation trees with root labelled by $A$ and frontier $w$. Then there exist two different left-most and two different right-most derivations of $A \Rightarrow^* w$.

2. Assume there are two different left-most derivations or two different right-most derivations of $A \Rightarrow^* w'$, Then there exist two different derivation trees of with root labelled by $A$ and frontier $w$. 
Proof of Theorem I.2.5.3

- The example above demonstrates how derivation trees are transformed into left-most derivations in a unique way.
- A more formal proof can be found in the additional material.
A direct consequence of the theorem is the following:

**Theorem**

Let $G = (T, N, S, P)$ be a CFG. The following are equivalent:

1. For every $w \in T^*$ there exist at most one derivation tree with label $S$ and frontier $w$.

2. For every $w \in T^*$ there exist at most one left-most derivation sequence $S \Rightarrow^* w$.

3. For every $w \in T^*$ there exist at most one right-most derivation sequence $S \Rightarrow^* w$.
Ambiguous Grammars

Definition

A CFG $G = (T, N, S, P)$ is **ambiguous**, if there is a string $w \in L(G)$ having more than one derivation tree (or, equivalently, having more than one left-most or right-most derivation).
Example 1

| grammar     | $G$ |
| terminals   | $S$ |
| nonterminals| $a, b$ |
| start symbol| $S$ |
| productions | $S \rightarrow aS$
              | $S \rightarrow b$
              | $S \rightarrow ab$ |
Example 1

There are two left-most derivations of $ab$:

$$S \Rightarrow aS \Rightarrow ab \text{ and } S \Rightarrow ab$$

And two derivation trees:

![Derivation Tree 1](image1)

![Derivation Tree 2](image2)
Example 2: Dangling Else

Assume the following grammar which is a cut down version of the grammar $G^{\text{while}}$ introduced in I.2.4 with \texttt{if\_then\_else\_fi} replaced by \texttt{if\_then} and a \texttt{if\_then\_else\_}:

<table>
<thead>
<tr>
<th>grammar</th>
<th>$G^{\text{Dangling_else}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>import</td>
<td>$G^{\text{Arithmetic_Expression}}, G^{\text{Boolean_Expression}}$</td>
</tr>
<tr>
<td>terminals</td>
<td>\texttt{if, then, else, ::=, ;}</td>
</tr>
<tr>
<td>nonterminals</td>
<td>$\text{Program}$</td>
</tr>
<tr>
<td>start symbol</td>
<td>$\text{Program}$</td>
</tr>
</tbody>
</table>
| productions       | $\text{Program} \rightarrow \text{Id} ::= AExp$  
|                    | $\text{Program} \rightarrow \text{if BExp then Program else Program}$  
|                    | $\text{Program} \rightarrow \text{if BExp then Program}$ |
Example 2: Dangling Else

Assume strings $b_1, b_2$ deriving from $BExp$ and string $s_1, s_2$ deriving from $Program$.
The string

$$\text{if } b_1 \text{ then if } b_2 \text{ then } s_1 \text{ else } s_2$$

has two derivation trees (we omit the derivation trees for $b_i, s_i$.)
First Derivation Tree

Program

if

BExp

then

Program

if

BExp

then

Program

else

Program

b₁

s₁

s₂

b₂
Second Derivation Tree

```
Program
  |  if  | BExp
  | then | Program
  | b_1  | if  | BExp
  |      |     | then | Program
  |      |     |      | s_1
  |      |     |      |     |
else
  | Program
  | s_2
```

I.2.5. Derivation Trees for Context-Free Grammars (14.1)
Different Interpretations of the Program

The two different derivation trees of the program

\[
\text{if } b_1 \text{ then if } b_2 \text{ then } s_1 \text{ else } s_2
\]

correspond to two different ways of executing the program:
In the first the else case belongs to the second if. It is executed if $b_1$ is true and $b_2$ is false.

The program can be using indentation be written as follows:

```plaintext
if $b_1$
  then
    if $b_2$
      then
        $s_1$
      else
        $s_2$
```
In the second derivation tree, the else case belongs to the first if. It is executed if $b_1$ is false.

The program can be using indentation be written as follows:

```plaintext
if $b_1$
    then
        if $b_2$
            then
                $s_1$
            else
                $s_2$
```
There are 2 solutions for solving this problem. The first solution is to add to `if_then` and `if_then_else` a symbol `fi` (or some other keyword such as `end − if`). Labelling the end of the statement.

<table>
<thead>
<tr>
<th>grammar</th>
<th>$G^{\text{Unambiguous _if}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>import</td>
<td>$G^{\text{Arithmetic _Expression}}, G^{\text{Boolean _Expression}}$</td>
</tr>
<tr>
<td>terminals</td>
<td><code>if, then, else, :=, ;</code></td>
</tr>
<tr>
<td>nonterminals</td>
<td><code>Program</code></td>
</tr>
<tr>
<td>start symbol</td>
<td><code>Program</code></td>
</tr>
</tbody>
</table>
| productions | `Program \rightarrow \text{Id} := AExp`  
               `Program \rightarrow \text{if } BExp \text{ then } Program \text{ else } Program \text{ fi}`  
               `Program \rightarrow \text{if } BExp \text{ then } Program \text{ fi}` |
Solution 1

Now the two interpretations of the original string would be written in as two different strings:

- “Else” belonging to the second “if” is written as

  \[
  \text{if } b_1 \text{ then if } b_2 \text{ then } s_1 \text{ else } s_2 \text{ fi fi}
  \]

- “Else” belong to the first “if” is written as

  \[
  \text{if } b_1 \text{ then if } b_2 \text{ then } s_1 \text{ fi else } s_2 \text{ fi}
  \]

- This solution has been taken for instance in Algol, in the bash shell (Linux), and in Ada (where fi is replaced by “end if”).
Solution 2

- The 2nd solution is to modify the grammar so that the derivation tree will be possible only for one of the two choices.
- For this one we modify the grammar, that the statement $s_1$ in

$$\text{if } b_1 \text{ then } s_1 \text{ else } s_2$$

is not matched by

$$\text{if } b'_1 \text{ then } s'_1$$

but only by

$$\text{if } b'_1 \text{ then } s'_1 \text{ else } s'_2$$

- This solution has been taken in most other programming languages.
Solution 2

- For this we split Programs into two categories:
  - Those derived from MatchedIf. In a program deriving from MatchedIf, each `if` is matched by an `else` clause.
  - Those derived from UnmatchedIf. These have at least one `if` with no matching `else` clause.
The grammar will make sure that a \texttt{else} will always be associated with the first \texttt{if} to the left, which has no unmatched \texttt{else} yet.

So \texttt{if\_then\_mathbf\_else} expression will be parsed as in the first derivation tree.
Here is the grammar. We omit sequencing (combining programs using “;”), since it would result in an ambiguous grammar, which needs to be resolved.

<table>
<thead>
<tr>
<th>grammar</th>
<th>$G^{Dangling_Else}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>import</td>
<td>$G^{Arithmetic_Expression}, G^{Boolean_Expression}$</td>
</tr>
<tr>
<td>terminals</td>
<td>if, then, else, :=, ;</td>
</tr>
<tr>
<td>nonterminals</td>
<td>Program</td>
</tr>
<tr>
<td>start symbol</td>
<td>Program</td>
</tr>
<tr>
<td>productions</td>
<td></td>
</tr>
<tr>
<td>Program</td>
<td>$\rightarrow$ UnmatchedIf</td>
</tr>
<tr>
<td>Program</td>
<td>$\rightarrow$ MatchedIf</td>
</tr>
<tr>
<td>MatchedIf</td>
<td>$\rightarrow$ Id := AExp</td>
</tr>
<tr>
<td>MatchedIf</td>
<td>$\rightarrow$ if BExp then MatchedIf else MatchedIf</td>
</tr>
<tr>
<td>UnmatchedIf</td>
<td>$\rightarrow$ if BExp then Program</td>
</tr>
<tr>
<td>UnmatchedIf</td>
<td>$\rightarrow$ if BExp then MatchedIf else UnmatchedIf</td>
</tr>
</tbody>
</table>
Unique Derivation Tree 2nd Solution

```
if BExp then

UnmatchedIf

if BExp then

MatchedIf

Program

else

MatchedIf

s1

s2

b1

b2
```
Example: Grammar for Arithmetic Expressions

Remember the grammar for arithmetic expressions (using elements of BNF notation)

<table>
<thead>
<tr>
<th>grammar</th>
<th>$G_{\text{Arithmetic Expression}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>import</td>
<td>$G_{\text{Identifier}}, G_{\text{Number}}$</td>
</tr>
<tr>
<td>terminals</td>
<td>$+, -, *, /, (, )$</td>
</tr>
<tr>
<td>nonterminals</td>
<td>$AExp, AOp$</td>
</tr>
<tr>
<td>start symbol</td>
<td>$AExp$</td>
</tr>
<tr>
<td>productions</td>
<td>$AExp \rightarrow Id</td>
</tr>
<tr>
<td></td>
<td>$AExp \rightarrow ( AExp )$</td>
</tr>
<tr>
<td></td>
<td>$AExp \rightarrow AExp AOp AExp$</td>
</tr>
<tr>
<td></td>
<td>$AOp \rightarrow +</td>
</tr>
</tbody>
</table>
First Parse tree for $2 + 3 \times 4$
Second Parse tree for $2 + 3 * 4$
Difference in Evaluation

- The first parse tree corresponds to parsing it as if it were \((2 + 3) \times 4\)
  Evaluation will return 20.
- The second parse tree corresponds to parsing it as if it were \(2 + (3 \times 4)\)
  Evaluation will return 14.
Unambiguous Version

<table>
<thead>
<tr>
<th>grammar</th>
<th>( G^{\text{Arithmetic Expression unambiguous}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>import</td>
<td>( G^{\text{Identifier}}, G^{\text{Number}} )</td>
</tr>
<tr>
<td>terminals</td>
<td>(+, -, *, /, (, ))</td>
</tr>
<tr>
<td>nonterminals</td>
<td>( AExp, Term, Factor )</td>
</tr>
<tr>
<td>start symbol</td>
<td>( AExp )</td>
</tr>
<tr>
<td>productions</td>
<td>( AExp \rightarrow AExp + Term \mid AExp - Term \mid Term )</td>
</tr>
<tr>
<td></td>
<td>( Term \rightarrow Term * Factor \mid Term/Factor \mid Factor )</td>
</tr>
<tr>
<td></td>
<td>( Factor \rightarrow Id \mid Number \mid ( AExp ) )</td>
</tr>
</tbody>
</table>
Unique Parse Tree for $2 + 3 \times 4$ in $G_{\text{Arithmetic Expression unambiguous}}$
Making Context Free Grammars Unambiguous

The following is known about Context Free Grammars:

- There are languages defined by context free grammars which cannot be defined by an unambiguous grammar.
  - Context free grammars, for which there exist no equivalent unambiguous grammar, are called inherently ambiguous grammars. See Hopcroft/Motwani/Ullman, 5.4.4, p. 213.

- It is undecidable whether a grammar is ambiguous. (Same book, 7.4.5, p. 307.)

- It is undecidable whether a grammar is inherently ambiguous grammars. (Same book, 7.4.5 and 9.5.2, p. 413.)