Chapter 2

Formal Languages

2.1 Why Study Formal Languages?

Whenever you interact with a computer using the keyboard, there are good chances that you're complying to a particular *syntax*. In fact, you're almost always using some *formal language* to communicate with a computer – maybe without you being ever aware of it!

This is obvious when you write some C or Java code, or when you create a web page. Yet, this is also true – as you'll soon discover – even when you search the web using your favourite search engine, when you enter the address of a website in your browser, when sign up and order a book on Amazon, or when you type on your good old pocket calculator.

The topic of "Formal Languages" thus constitutes the standard way of dealing with the definition, characterization, and validation of text-based communication between human and computer, as well as between computers. Not only is such kind of communication almost ubiquitous in today's world, but the formalization of this topic will allow us to smoothly penetrate some of the core areas of theoretical computer science – such as formal grammars, automata theory and computational complexity – and will naturally lead us to further exciting topics that will be discussed during the lecture – such as fractals, recursion and chaos.

This chapter is organized as follows. We will start in Section 2.2 by introducing the basic concepts and formalism of formal language theory using analogies with natural language and common knowledge of grammar. Section 2.3 follows with all sorts of useful, yet more formal definitions. The next three Sections 2.4, 2.5 and 2.6 describe, in an increasing level of complexity, the major classes of

languages and grammars, which are then summarized in Section 2.7. A brief recapitulation of the main points of this chapter is provided in the last Section 2.8.

2.2 Natural and Formal Languages: Basic Terminology

Let's first give an example of natural and formal languages, so that one understands what we're talking about. English is obviously a natural language, whereas C++ is a formal one.

Interestingly, natural and formal languages – even though they profoundly differ in many respects (see Table 2.1) – share nevertheless sufficient similarities so that a comparison between the two will help us smoothly introduce the topic and the terminology of Formal Languages.

Natural Language
+ High expressiveness
+ No extra learning
− Ambiguity
− Vagueness
− Longish style
− Consistency hard to check

Formal Language
+ Well-defined syntax
+ Unambiguous semantics
+ Can be processed by a computer
+ Large problems can be solved
− High learning effort
− Limited expressiveness
− Low acceptance

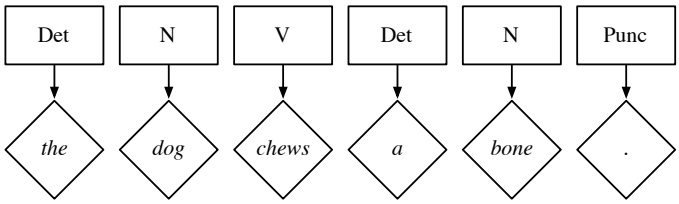
Table 2.1: Characteristics of Natural and Formal Languages.

2.2.1 Syntax

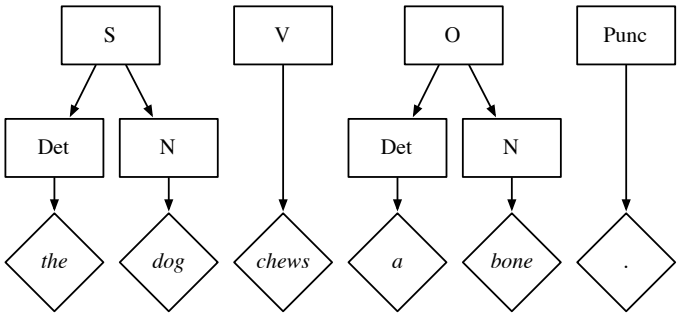
Everyone knows, after spending some years sitting on school banks, that a correct German or English sentence should respect the rules of its grammar (except if one tries to do poetry or modern art). Let's consider the following sentence:

the dog chews a bone.

This sentence can be decomposed into distinct elements (words and punctuation signs), each having a particular grammatical function. *the* is a determinent (Det), *dog* is a noun (N), *chews* is a verb (V), . is a punctuation sign (Punc), etc.



In turn, these new elements can be further combined into higher-level groups. For instance, *the dog* is the subject (S) and *a bone* is the object (O):



This hierarchy can be further expanded into a phrase (P) with the classic subject-verb-object (SVO) structure, until one element remains – the *starting* or *initial symbol* (I). Figure 2.1 illustrates how a tree representation is a convenient way of representing the *syntax* structure of a single phrase.

2.2.2 Semantics

Clearly, we're only dealing with the structure of a phrase – i.e. its *syntax* – and not about the actual meaning of the phrase – i.e. its *semantics*. In other words, we're only interested in characterizing phrases that "sounds" English, even if they don't carry much meaning – such as the phrase "*the bone chews a dog.*" which is syntacticly correct.

The reason is that verifying the semantics of a phrase is almost impossible, whereas verifying its syntax is much simpler. To draw a comparison with a formal language, each compiler can verify whether a C++ code has a correct syntax (if so, it can be compiled; otherwise, the compiler will generate an error). But no computer will ever be able to verify that the code corresponds to a meaningful program![1]

[1] In fact, it's not even clear what *meaningful* is at all. Does a simple "`printf("Hello, world!\n");`" generate anything meaningful for you?

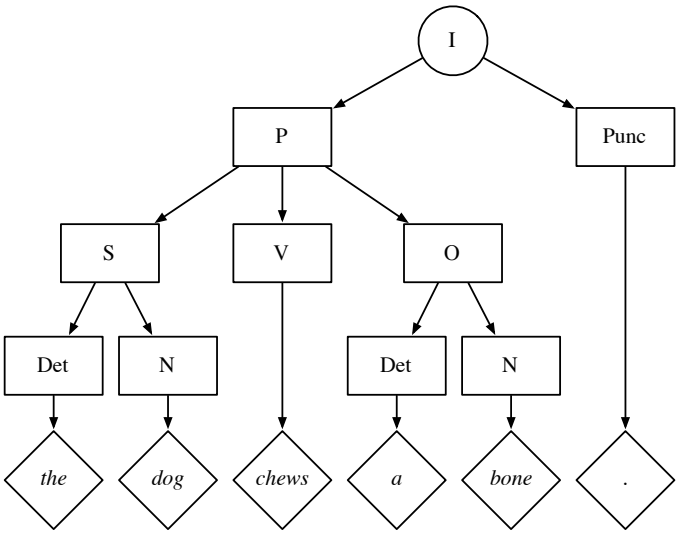
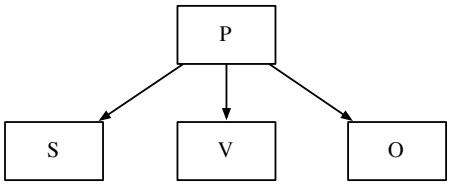


Figure 2.1: Tree representation of the syntax of an English phrase.

2.2.3 Grammar

If a syntax tree is a convenient way of representing the syntax of phrase – i.e. a *single element* of a language – it cannot be used to describe the *whole language* itself – i.e. the set of all possible elements, such as all syntactically correct English phrases.

This is where *grammar rules* come into play. Consider Figure 2.1 from top to bottom. The *starting symbol* (represented by a circle) and all *non-terminal symbols* (represented by boxes) get successively replaced by a set of non-terminal or *terminal symbols* (represented by diamonds), until there remain only terminal symbols. Each of these replacements can be represented by a *production* or *rewrite rule*:



$$P \rightarrow S\ V\ O$$

One possible set of production rules – i.e. a *generative grammar* –corresponding to our example is the following (" | " stands for "or"):

$$
\begin{aligned}
I &\rightarrow P\ Punc \\
P &\rightarrow S\ V\ O \\
S &\rightarrow Det\ N \\
O &\rightarrow Det\ N \\
V &\rightarrow \textit{chews} \\
Det &\rightarrow \textit{the} \mid \textit{a} \\
N &\rightarrow \textit{dog} \mid \textit{bone} \\
Punc &\rightarrow .
\end{aligned}
$$

Table 2.2: A simple generative grammar.

Of course, this simple grammar by far does not generate the whole set of English sentences. However, it allows us to see how this initial grammar could be extended to include more and more phrases:

$$
\begin{aligned}
I &\rightarrow P\ Punc \mid P_i\ Punc_i \\
P &\rightarrow S\ Vs\ O \\
P_i &\rightarrow Aux\ S\ V\ O \\
S &\rightarrow Det\ N \\
O &\rightarrow Det\ N \\
V &\rightarrow \textit{chew} \mid \textit{bite} \mid \textit{lick} \\
Aux &\rightarrow \textit{does} \mid \textit{did} \mid \textit{will} \\
Det &\rightarrow \textit{the} \mid \textit{a} \mid \textit{one} \mid \textit{my} \\
N &\rightarrow \textit{dog} \mid \textit{bone} \mid \textit{frog} \mid \textit{child} \\
Punc &\rightarrow . \mid ; \mid ! \\
Punc_i &\rightarrow ? \mid ?!?
\end{aligned}
$$

Table 2.3: A slightly more complex generative grammar.

Recursive Rules

What about if we want to also include adjectives into our language (i.e. into our set of possible phrases generated by our grammar)? Note that we would like to append as many adjectives as we'd like to a noun, such as:

> *the cute tiny little furry ... red-haired dog chews a big tasty ... white bone.*

Obviously, we'll need at least one production rule for all possible adjectives:

$$Adj \quad \rightarrow \quad \textit{cute} \mid \textit{tiny} \mid \textit{little} \mid \textit{furry} \mid \textit{red-haired} \mid \textit{big} \mid \textit{tasty} \mid \textit{white} \mid ...$$

However, we do not want to have a infinite number of production rules in order to append any number of adjectives to a noun, such as $S \rightarrow Det\ N$, $S \rightarrow Det\ Adj\ N$, $S \rightarrow Det\ Adj\ Adj\ N$, $S \rightarrow Det\ Adj\ Adj\ Adj\ N$, etc.

Instead, we'll simply use the following – admittedly more elegant – solution, which relies on *recursion*:

$$N \quad \rightarrow \quad Adj\ N$$

Figure 2.2 illustrates how the usage of this rule, with an arbitrary level of recursion, can lead to derivations of a noun with any number of adjectives.

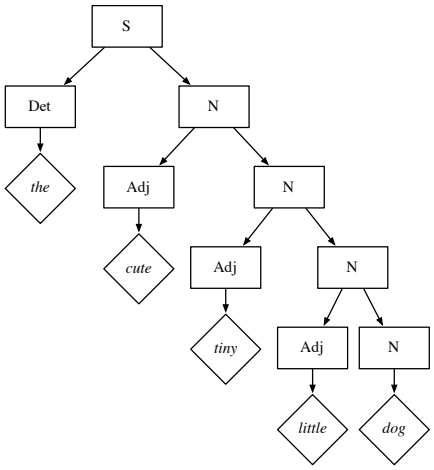


Figure 2.2: Derivation using a recursive rule.

The new generative grammar that includes the two new production rules for the adjectives is summarized in Table 2.4.

I	$\rightarrow$	$P\ Punc \mid P_i\ Punc_i$
P	$\rightarrow$	$S\ V s\ O$
P_i	$\rightarrow$	$Aux\ S\ V\ O$
S	$\rightarrow$	$Det\ N$
O	$\rightarrow$	$Det\ N$
V	$\rightarrow$	*chew* | *bite* | *lick*
Aux	$\rightarrow$	*does* | *did* | *will*
Det	$\rightarrow$	*the* | *a* | *one* | *my*
N	$\rightarrow$	$Adj\ N$ | *dog* | *bone* | *frog* | *child*
Adj	$\rightarrow$	*cute* | *tiny* | *little* | *furry* | *red-haired* | *big* | *tasty* | *white* | ...
$Punc$	$\rightarrow$	. | ; | !
$Punc_i$	$\rightarrow$	? | ?!?

Table 2.4: A generative grammar with a recursive production rule.

2.3 Formal Languages and Grammar: Definitions

We'll now abandon the analogies between formal and natural languages to enter the core of the formal language theory.

In the following definitions, ϵ stands for the *empty* or *zero-length string*. This element ϵ shouldn't be confused with the empty set $\emptyset$, which contains no elements.

Definition 2.1 (Kleene Star)**.** Let Σ be an alphabet, i.e. a finite set of distinct symbols. A string is a finite concatenation of elements of Σ. Σ^* is defined as the set of all possible strings over Σ.

Σ^* can equivalently be defined recursively as follows:

1. Basis: $\epsilon \in \Sigma^*$.
2. Recursive step: If $w \in \Sigma^*$ and $a \in \Sigma$, then $wa \in \Sigma^*$.
3. Closure: $w \in \Sigma^*$ only if it can be obtained from ϵ by a finite number of applications of the recursive step.

For any nonempty alphabet Σ, Σ^* contains infinitely many elements.

Σ	Σ^*
$\emptyset$	$\{\epsilon\}$
$\{a\}$	$\{\epsilon, a, aa, aaa, \ldots\}$
$\{0, 1\}$	$\{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011 \ldots\}$
$\{\$, \star, £\}$	$\{\epsilon, \$, \star, £, \$\$, \$\star, \$£, \star\$, \star\star, \star£, £\$, £\star, ££, \$\$\$, \$\$\star, \ldots\}$

Definition 2.2. A language L over an alphabet Σ is a subset of Σ^*:

$$L \subseteq \Sigma^*$$

A language is thus a possibly infinite set of finite-length sequences of elements (strings) drawn from a specified finite set (its alphabet Σ).

The union of two languages L and M over an alphabet Σ is defined as:

$$L \cup M := \left\{ w \in \Sigma^* \,\middle|\, w \in L \vee w \in M \right\}$$

The intersection of two languages L and M over an alphabet Σ is defined as:

$$L \cap M := \left\{ w \in \Sigma^* \,\middle|\, w \in L \wedge w \in M \right\}$$

The concatenation of two languages L and M is defined as:

$$LM := \left\{ uv \,\middle|\, u \in L \wedge v \in M \right\}$$

Example 2.1. L_1 is the language over $\Sigma = \{0, 1\}$ consisting of all strings that begins with 1.

$$L_1 = \{1, 10, 11, 100, 101, \ldots\}$$

L_2 is the language over $\Sigma = \{0, 1\}$ consisting of all strings that contains an even number of 0's.

$$L_2 = \{\epsilon, 1, 00, 11, 001, 010, 100, 0011, \ldots\}$$

L_3 is the language over $\Sigma = \{a, b\}$ consisting of all strings that contains as many a's and b's.

$$L_3 = \{\epsilon, ab, ba, aabb, abab, baba, bbaa, \ldots\}$$

L_4 is the language over $\Sigma = \{x\}$ consisting of all strings that contains a prime number of x's.

$$L_4 = \{xx, xxx, xxxxx, xxxxxxx, xxxxxxxxxxx, \ldots\}$$

Definition 2.3. A grammar G is formally defined as a quadruple

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

with

- Σ: a finite set of terminal symbols (the alphabet)
- V: a finite set of non-terminal symbols (the variables), usually with the condition that $V \cap \Sigma = \emptyset$.
- P: a finite set of production rules
- $S \in V$: the start symbol.

Non-terminal symbols are usually represented by uppercase letters, terminal symbols by lowercase letters, and the start symbol by S.

A formal grammar G defines (or *generates*) a formal language $L(G)$, which is the (possibly infinite) set of sequences of symbols that can be constructed by successively applying the production rules to a sequence of symbols, which initially contains just the start symbol, until the sequence contains only terminal symbols. A rule may be applied to a sequence of symbols by replacing an occurrence of the symbols on the left-hand side of the rule with those that appear on the right-hand side. A sequence of rule applications is called a *derivation*.

Example 2.2. Let us consider the following grammar:

$$\begin{array}{rl} \text{Alphabet } \Sigma\text{:} & \{0, 1, +\} \\ \text{Variables } V\text{:} & \{S, N\} \\ \text{Production rules } P\text{:} & S \rightarrow N \mid N + S \\ & N \rightarrow 0 \mid 1 \mid NN \\ \text{Start symbol:} & S \end{array}$$

The corresponding language contains simple expressions corresponding to the additions of binary numbers, such as $w = 10 + 0 + 1$. One possible derivation of w is as follows:

$$\begin{array}{rcl} S & \Rightarrow & N + S \\ & \Rightarrow & NN + S \\ & \Rightarrow & 1N + S \\ & \Rightarrow & 10 + S \\ & \Rightarrow & 10 + N + S \\ & \Rightarrow & 10 + N + N \\ & \Rightarrow & 10 + N + 1 \\ & \Rightarrow & 10 + 0 + 1 \end{array}$$

This derivation can be represented by the following *syntax tree*:

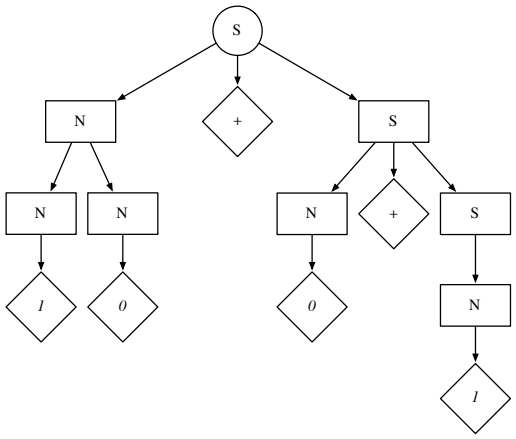


Figure 2.3: Syntax tree corresponding to one possible derivation of "10 + 0 + 1".

2.4 Regular Languages

Languages and their generative grammars can be be classified into different categories depending on the complexity of the structures of their production rules.

The first category, which correspond to the simplest kind of production rules, is called *regular*.

2.4.1 Regular Grammars

Definition 2.4. All the production rules of a *right regular grammar* are of the form:

$$A \to xyz \ldots X$$

where A is a non-terminal symbol, $xyz \ldots$ zero or more terminal symbols, and X zero or one non-terminal symbol. A *left* regular grammar have productions rules of the form:

$$A \to Xxyz \ldots$$

A regular grammar is either a left regular or a right regular grammar.

Metaphorically speaking, regular grammars generate the elements of the language by "appending" the symbols at either the right or the left during the derivation.

Example 2.3. An example of a regular grammar G with $\Sigma = \{a, b\}$, $V = \{S, A\}$, consists of the following set of rules P:

$$\begin{aligned} S &\to aS \mid aA \\ A &\to bA \mid \epsilon \end{aligned}$$

Definition 2.5. A regular language is a language that can be generated by a regular grammar.

Example 2.4. An archetypical regular language is:

$$L = \left\{ a^m b^n \;\middle|\; m, n \geq 0 \right\}$$

L is the language of all strings over the alphabet $\Sigma = \{a, b\}$ where all the a's precede the b's: $L = \{\epsilon, a, b, aa, ab, bb, aaa, aab, \ldots\}$.

2.4.2 Regular Expressions

Regular languages are typically described by *regular expressions*. A regular expression is a string that describes or matches a set of strings (the elements of the corresponding regular language), according to certain syntax rules.

There are several kinds of syntax for regular expressions. Most of them are equivalent and only differ in the symbols they use. Regular expression operators commonly used are:

Symbol	Stands for...
+	at least one occurrence of the preceding symbol
*	zero, one, or more occurrences of the preceding symbol
?	zero or one occurrence of the preceding symbol
|	logical "or"
ϵ	the empty string

Examples of regular expressions

Regular Expression	Elements of the Language
abc	abc
a*bc	bc, abc, aabc, aaabc, aaaabc, ...
go+gle	gogle, google, gooogle, ...
pfeiff?er	pfeifer, pfeiffer
pf(a|e)ifer	pfaifer, pfeifer

They are widely used for searching, e.g. in text documents or on the internet. Popular search engines/tools that utilize regular expressions are for example `grep` or – to some extent – Google.

Unix regular expressions

The traditional regular expressions, even though they are considered as obsolete by some standards, are still widely used in the Unix world, such as by the `grep` or `sed` utilities. The basic syntax is as follows:

Symbol	Matches...
`.`	any single character (wildcard)
`[ ]`	a single character that is contained within the brackets
`[^ ]`	a single character that is *not* contained within the brackets
`\n`	a digit from 0 to 9
`^`	start of the string
`$`	end of the string
`*`	zero, one or more copies of the preceding symbol (or expression)
$\{x,y\}$	at least x and not more than y copies of the preceding symbol (or expression)

Some standard defines classes or categories of characters such as:

Class	Similar to	Meaning
`[[:upper:]]`	`[A-Z]`	uppercase letters
`[[:lower:]]`	`[a-z]`	lowercase letters
`[[:alpha:]]`	`[A-Za-z]`	upper- and lowercase letters
`[[:digit:]]`	`[0-9]`	digits
`[[:alnum:]]`	`[A-Za-z0-9]`	digits, upper- and lowercase letters
`[[:space:]]`	`[ \t\n]`	whitespace characters
`[[:graph:]]`	`[^[:space:]]`	printed characters

Finally, the more modern "extended" regular expressions differs principally in that it includes the following symbols:

Symbol	Matches...
`+`	one or more copies of the preceding symbol (or expression)
`?`	zero or one copy of the preceding symbol (or expression)
`|`	either the expression before or the expression after
`\`	the next symbol literally (escaping)

2.5 Context-Free Languages

The next category of languages, which are more complex than the regular languages we've just met, yet less complex than the context-sensitive languages we'll encounter in the next section, are the so-called *context-free* languages.

Context-free languages include most programming languages, and is thus one the central category in the theory of formal languages.

Definition 2.6. A context-free grammar is a grammar whose productions rules are all of the form:

$$A \to \dots$$

where A is a *single* non-terminal symbol, and where the ellipsis "..." stands for any number of terminals and/or non-terminal symbols.

In contrast to regular grammar, a context-free grammar allows the description of *symmetries* such as the structure of unlimited balanced parenthesis.

To illustrate this, consider the following context-free grammar

$$S \;\; \to \;\; aS \mid S(S)S \mid \epsilon$$

which generates strings over the alphabet $\Sigma = \{a, (,)\}$ with any number of correctly balanced parenthesis: $(a), ((aa)aaa)a(a)(), \dots$

Definition 2.7. A language L is said to be a context-free language if there exists a context-free grammar G, such that $L = L(G)$.

In other words, a context-free language is a language that can be generated by a context-free grammar.

Example 2.5. A simple context-free grammar is

$$S \to aSb \mid \epsilon$$

It generates the context-free language

$$L = \left\{ a^n b^n \;\middle|\; n \geq 0 \right\}$$

which consists of all strings that contain some number of a's followed by the same number of b's: $L = \{\epsilon, ab, aabb, aaabbb, \dots\}$.

Note that at least one rule of a context-free grammar has to be of the form

$$A \rightarrow xy \ldots XY \ldots vw \ldots$$

and has to be recursive, either directly

$$A \rightarrow xAy$$

or indirectly

$$\begin{aligned} A &\rightarrow xBy \\ B &\rightarrow vAw \end{aligned}$$

Example 2.6. The following grammar generates strings containing any number of a's and b's within a pair of quotes.

$$\begin{aligned} S &\rightarrow \text{“}A\text{”} \\ A &\rightarrow aA \mid B \\ B &\rightarrow Bb \mid \epsilon \end{aligned}$$

It may look at first sight context-free, but is in fact equivalent to the following regular grammar, thus showing that the corresponding language is *regular*:

$$\begin{aligned} S &\rightarrow \text{“}A \\ A &\rightarrow aA \mid B \\ B &\rightarrow bB \mid \text{”} \end{aligned}$$

2.5.1 Backus-Naur Forms

A BNF (Backus-Naur form or Backus normal form) is a particular syntax used to express context-free grammars in a slightly more convenient way. A derivation rule in the original BNF specification is written as

$$< \text{symbol} > ::= < \text{expression with symbols} >$$

Terminal symbols are usually enclosed within quotation marks (“...” or ‘...’):

$$\begin{aligned} < \text{number} > &::= < \text{digit} > \mid < \text{digit} >< \text{number} > \\ < \text{digit} > &::= \text{“0”} \mid \text{“1”} \mid \text{“2”} \mid \text{“3”} \mid \text{“4”} \mid \text{“5”} \mid \text{“6”} \mid \text{“7”} \mid \text{“8”} \mid \text{“9”} \end{aligned}$$

Example 2.7. The standard BNF's syntax may be represented with a BNF like the following:

```
< syntax >        ::= < rule > | < rule >< syntax >
< rule >          ::= < opt-whitespace > " < " < rule-name > " > "
                      < opt-whitespace > " ::= " < opt-whitespace >
                      < expression >< line-end >
< opt-whitespace > ::= "" | " " < opt-whitespace >
< expression >    ::= < list > | < list > "|" < expression >
< line-end >      ::= < opt-whitespace >< EOL > | < line-end >< line-end >
< list >          ::= < term > | < term >< opt-whitespace >< list >
< term >          ::= < literal > |" < " < rule-name > " > "
< literal >       ::= """ < text > """
```

There are many extensions of and variants on BNF. For instance, the extended Backus-Naur form (EBNF), replaces the "::=" with a simple "=", allows the angle brackets "<...>" for nonterminals to be omitted, and uses a terminating character (usually a semicolon) to mark the end of a rule. In addition, the comma can be used to separate the elements of a sequence, curly braces "{...}" represent expressions that may be omitted or repeated (the regexp repetition symbols * and + can also be used), and brackets "[...]" represent optional expressions.

Example 2.8. The following grammar in EBNF notation defines a simplified HTML syntax:

```
document    = element ;
element     = ( text | list ) * ;
text        = ('A' .. 'Z' | 'a' .. 'z' | '0' .. '9' | ' ') + ;
list        = '<ul>' listElement * '</ul>'
            | '<ol>' listElement * '</ol>' ;
listElement = '<li>' element ;
```

Figure 2.4 shows the syntax tree corresponding to the following simplified HTML code:

```
Buy<ol><li>Fruits<ul><li>Apple<li>Banana</ul><li>Pasta<li>Water</ol>
```

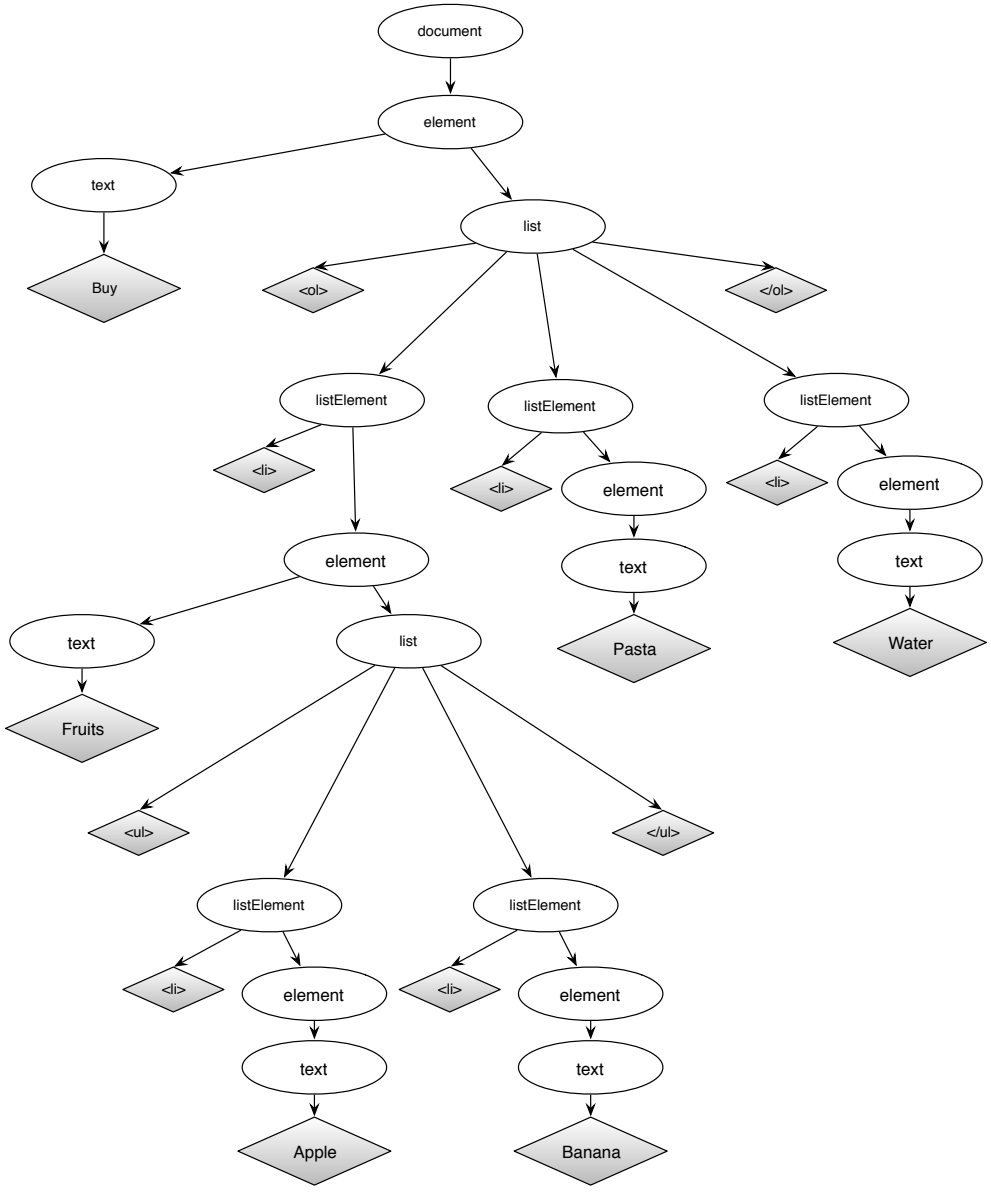
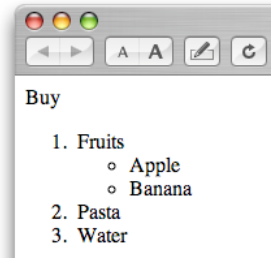


Figure 2.4: Syntax tree of a simplified HTML string.

which renders in a browser as:



2.5.2 Grammar Tree

In a previous example, we introduced the following grammar:

$$\begin{aligned} S &\rightarrow N \mid N + S \\ N &\rightarrow 0 \mid 1 \mid NN \end{aligned}$$

Recall Figure 2.3 (on page 2-10), which illustrated the syntax tree for the derivation of one element of the language (namely the string "10 + 0 + 1"). In a syntax tree, each *branching* corresponds to a particular production rule, and the leafs represent the terminal symbols (which are then read from left to right).

Definition 2.8. A *grammar tree* is a tree where each *link* corresponds to the application of one particular production rule, and where the leafs represent the elements of the language.

The path from the root element to a leaf corresponds to the *derivation* of that element. Obviously, if a language contains an infinite number of elements, its grammar tree is infinitely large.

Figure 2.5 represents the partial grammar tree corresponding to the grammar given above.

You will probably notice that there are at least two leaves in the grammar tree that corresponded to the string "10 + 0 + 1". This leads us to the next subsections about parsing ambiguities.

2.5.3 Parsing

Parsing (also referred more formally to as "syntactic analysis") is the process of analyzing a sequence of symbols (or more generally tokens) to determine its grammatical structure with respect to a given formal grammar.

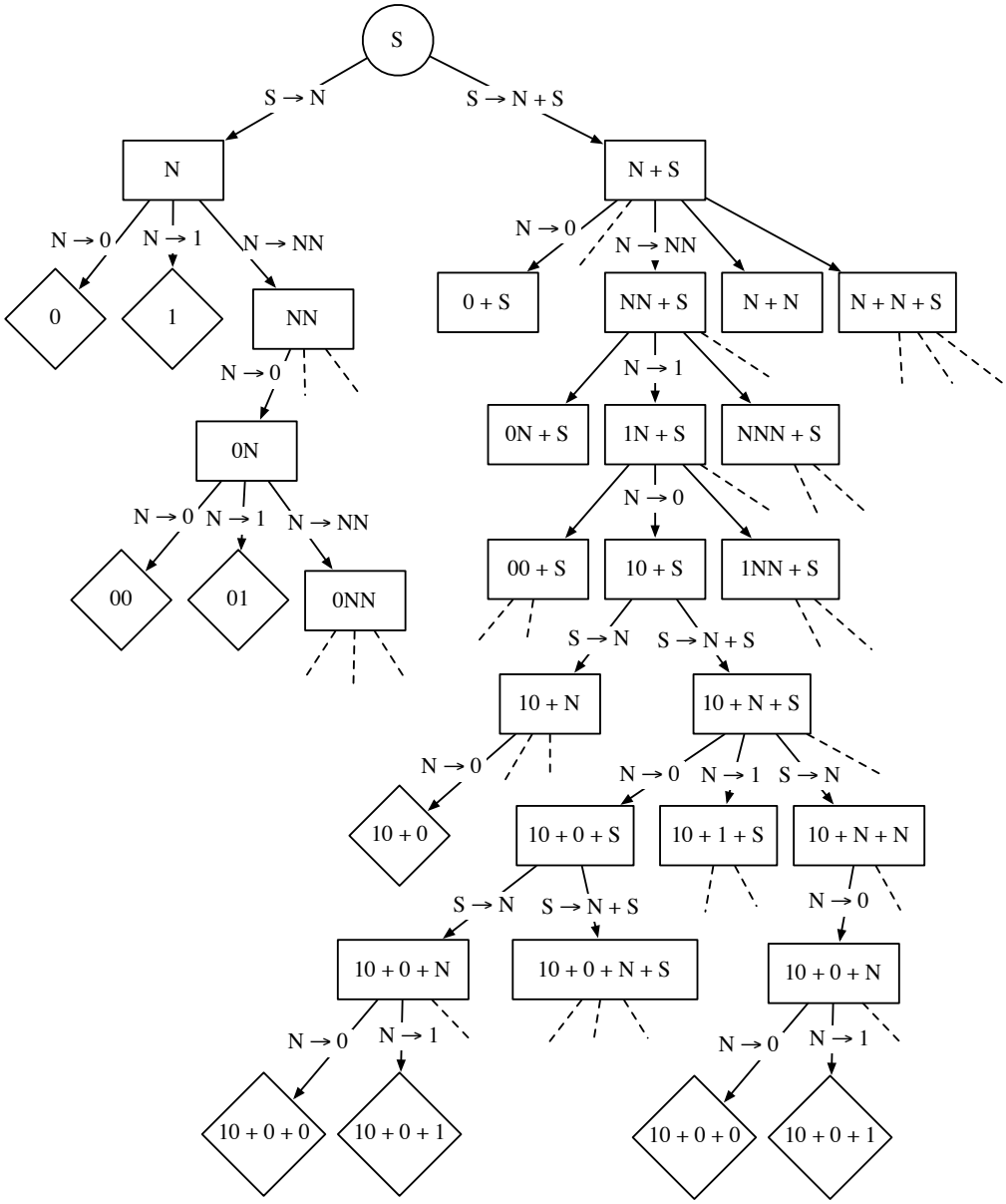


Figure 2.5: Partial grammar tree for the grammar $S \to N \mid N + S$, $N \to 0 \mid 1 \mid N$.

Parsing transforms input text into a data structure, usually a tree, which is suitable for later processing and which captures the implied hierarchy of the input.

Let us consider the following context-free grammar: The corresponding lan-

$$\begin{aligned} S &\rightarrow S + S \mid S - S \mid S \times S \mid N \\ N &\rightarrow 2 \mid 3 \mid 4 \end{aligned}$$

Table 2.5: A simple context-free grammar.

guage consists of all mathematical expressions with the operators $+$, $-$ and $\times$, and with the number 2, 3 and 4. (Obviously not a very exciting language, but this should do for our illustration purpose).

Parsing a string of this language (such "$2+3$") means in fact nothing else than the process of evaluating it. Not so surprisingly, the *syntax tree* is quite exactly the kind of data structure we're looking for (see Figure 2.6). In fact, a syntax tree is also referred to as *parse tree*, as it shows the structure necessary to parse the string.

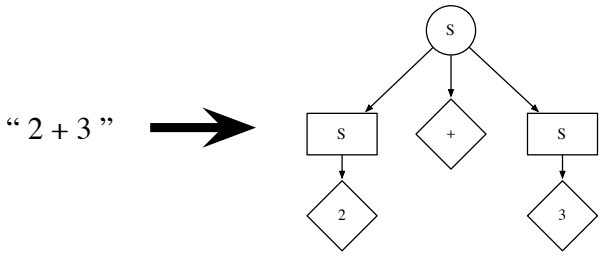


Figure 2.6: The process of parsing. Left: the input string. Right: the corresponding parse tree showing the syntactical structure.

2.5.4 Ambiguity and Leftmost Derivations

The tacit assumption so far was that each grammar uniquely determines a structure for each string in its language. However, we'll see that not every grammar does provide unique structures.

Consider for instance the string "$2-3-4$". Figure 2.7 shows that this expression can be evaluated, with respect to the grammar given above, to either

$$(2-3)-4=-5$$

or

$$2-(3-4)=3$$

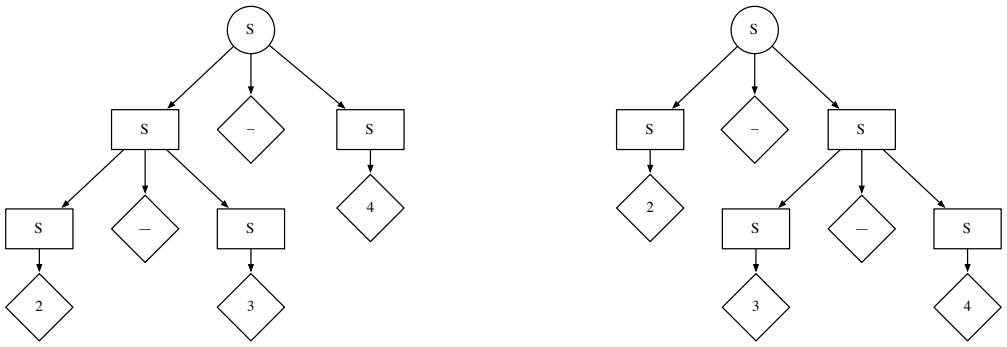


Figure 2.7: Two parse trees showing the two possible derivations of the input string "$2-3-4$".

Definition 2.9. A grammar is said to be *ambiguous* if the language it generates contains some string that have more than one possible parse tree.[2]

Note that the ambiguity is caused by a multiplicity of parse trees (and thus a multiple syntactic interpretation of the string) rather than a multiplicity of derivations. For instance, the string "$2+3$" has the two derivations

$$S \Rightarrow S+S \Rightarrow 2+S \Rightarrow 2+3$$

and

$$S \Rightarrow S+S \Rightarrow S+3 \Rightarrow 2+3$$

However, the is no real difference between the structures provided by these derivations, which both result in the parse tree shown in Figure 2.6.

[2]See also Definition 2.11.

Definition 2.10. The *leftmost derivation* of a string is the derivation obtained by always replacing the lest-most non-terminal symbols first. Similarly, the *rightmost derivation* of a string is the derivation obtained by always replacing the right-most non-terminal symbols first.

Some textbooks define a grammar as ambiguous only when the language contains some strings that have more than one leftmost (or rightmost) derivation.

Definition 2.11 (Alternative)**.** A grammar is said to be *ambiguous* if the language it generates contains some string that can be derived by two distinct lefmost derivations.

Note that the problem of grouping a sequence of (identical) operators from the left or from the right (what is referred to as the *left-* or *right-associativity*) is not the same as imposing a leftmost or rightmost derivation. Figures 2.7(a) and 2.7(b) represent a grouping of the "$-$" operators from the left and from the right, respectively.

Yet, both parse trees can be obtained with leftmost derivations:

$$S \Rightarrow S - S \Rightarrow (S - S) - S \Rightarrow (2 - S) - S \Rightarrow (2 - 3) - S \Rightarrow (2 - 3) - 4$$

and

$$S \Rightarrow S - S \Rightarrow 2 - S \Rightarrow 2 - (S - S) \Rightarrow 2 - (3 - S) \Rightarrow 2 - (3 - 4)$$

respectively. Note that the parentheses are not part of the derivation, but are only inserted to show the order of rewriting.

Removing ambiguity from grammars

In an ideal world, we would be able to give you an algorithm to remove ambiguity from context-free grammars. However, the surprising fact is that there is no algorithm whatsoever that can even tell us whether a context-free grammar is ambiguous in the first place. In fact, there are even context-free languages that have nothing but ambiguous grammars; for these languages, removal of ambiguity is impossible.

Fortunately, the situation in practice is not so grim. For the sorts of constructs that appear in common context-free languages (such as programming languages), there are well-known techniques for eliminating ambiguity. The problem encountered with the grammar of Table 2.5 is typical, and we shall now explore the elimination of its ambiguity as an important illustration.

First, let us note that there are two distinct causes of ambiguity in the grammar of Table 2.5:

1. A sequence of "$+$" or "$-$" operators, or a sequence of "$\times$" operators, can group either from the left or from the right. Most operators are left-associative in conventional arithmetics:

$$2 - 3 + 4 - 3 - 2 = (((2 - 3) + 4) - 3) - 2$$

2. The precedence of operators in not respected. In conventional arithmetics, the "$\times$" has a highest precedence than the "$+$" or "$-$" operators:

$$2 - 3 \times 4 = 2 - (3 \times 4)$$

The solution of the problem is to introduce several different variables, each of which represents those expressions that share a level of "binding strength." Specifically:

1. Any string in the language is a mathematical *expression* E.

2. Each expression consists in the sum or difference of *terms* T (or just in a single term).

3. Each term consists, in turn, in the product of *factors* F (or just in a single factor).

This hierarchy can be illustrated as follows:

$$\underbrace{\underbrace{\underbrace{2}_{\text{factor}}}_{\text{term}} - \underbrace{\underbrace{3}_{\text{factor}} \times \underbrace{4}_{\text{factor}}}_{\text{term}}}_{\text{expression}}$$

This leads us to the following unambiguous grammar:

$$\begin{aligned} S &\rightarrow E \\ E &\rightarrow T \mid E + T \mid E - T \\ T &\rightarrow F \mid T \times F \\ F &\rightarrow N \\ N &\rightarrow 2 \mid 3 \mid 4 \end{aligned}$$

Table 2.6: A simple unambiguous context-free grammar.

If we consider the expression "$2 - 3 \times 4$", we can see that the only possible parse tree for this string is the one shown in Figure 2.8, which corresponds to a derivation such as:

$$\begin{aligned} S &\Rightarrow E - T \Rightarrow T - T \\ &\Rightarrow F - T \Rightarrow N - T \Rightarrow 2 - T \\ &\Rightarrow 2 - T \times F \Rightarrow 2 - F \times F \\ &\Rightarrow 2 - N \times F \Rightarrow 2 - 3 \times F \\ &\Rightarrow 2 - 3 \times N \Rightarrow 2 - 3 \times 4 \end{aligned}$$

2.6 Context-Sensitive Languages and Unrestricted Grammars

The context-sensitive grammars represent an intermediate step between the context-free and the unrestricted grammars. No restrictions are placed on the left-hand side of a production rule, but the length of the right-hand side is required to be at least that of the left.

Note that context-sensitive grammars (and context-sensitive languages) are the least often used, both in theory and in practice.

Definition 2.12. A grammar $G = (\Sigma, V, P, S)$ is called *context-sensitive* if each rule has the form $u \rightarrow v$, where $u, v \in (\Sigma \cup V)^+$, and $|u| \leq |v|$.

In addition, a rule of the form $S \rightarrow \epsilon$ is permitted provided S does not appear on the right side of any rule. This allows the empty string ϵ to be included in context-sensitive languages.

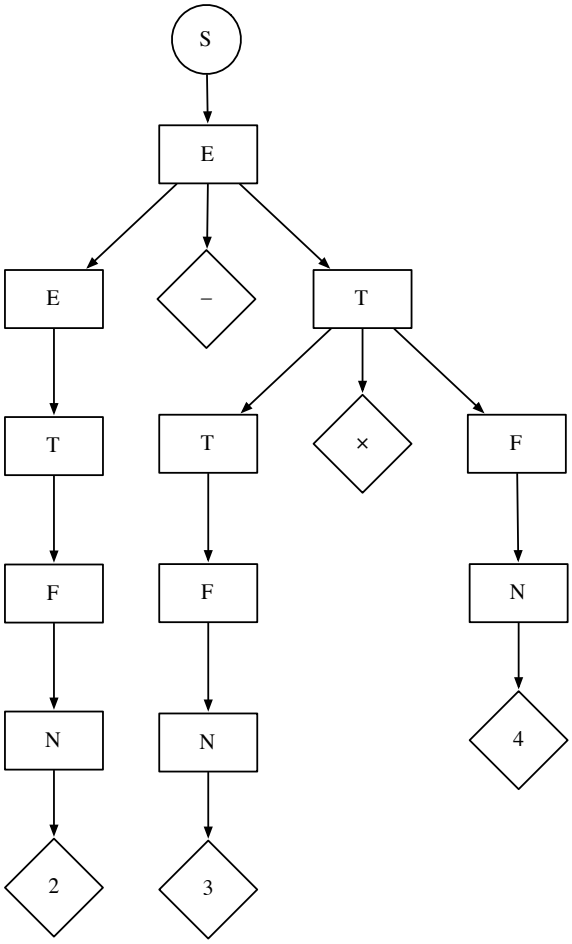


Figure 2.8: Parsing tree of the expression "$2 - 3 \times 4$" using an unambiguous grammar.

Definition 2.13. A *context-sensitive language* is a formal language that can be defined by a context-sensitive grammar.

Example 2.9. A typical example of context-sensitive language that is not context-free is the language

$$L = \left\{ a^n b^n c^n \;\middle|\; n > 0 \right\}$$

which can be generated by the following context-sensitive grammar:

$$\begin{aligned} S &\rightarrow aAbc \mid abc \\ A &\rightarrow aAbC \mid abC \\ Cb &\rightarrow bC \\ Cc &\rightarrow cc \end{aligned}$$

The last rule of this grammar illustrates the reason why such grammars are called "context-sensitive": C can only be replaced by c in a particular context – namely when it is followed by a c.

Definition 2.14. An *unrestricted grammar* is a formal grammar $G = (\Sigma, V, P, S)$ where each rule has the form $u \rightarrow v$, where $u, v \in (\Sigma \cup V)^+$, and $u \neq \epsilon$.

As the name implies, there are no real restrictions on the types of production rules that unrestricted grammars can have.

2.7 The Chomsky Hierarchy

The previous sections introduced different classes of formal grammars: regular, context-free, context-sensitive and unrestricted. These four families of grammars make up the so-called *Chomsky hierarchy*[3], with each successive family in the hierarchy permitting additional flexibility in the definition of a rule.

The nesting of the families of grammars of the Chomsky hierarchy induces a nesting of the corresponding languages:

$$\mathcal{L}_{\text{regular}} \subset \mathcal{L}_{\text{context-free}} \subset \mathcal{L}_{\text{context-sensitive}} \subset \mathcal{L}_{\text{unrestricted}} \subset \mathcal{P}(\Sigma^*)$$

where $\mathcal{L}_{type}$ is the set of all languages generated by a grammar of type *type*, and $\mathcal{P}(\Sigma^*)$ the power set of Σ^*, i.e. the set of all possible languages.

The Chomsky hierarchy is summarized in Table 2.7.

[3]Named after the renowned linguist and philosopher Noam Chomsky, who described this hierarchy of grammars in 1956.

Type	Grammar	Production Rules	Example of Language
0	Unrestricted	$\alpha \rightarrow \beta$	
1	Context-Sensitive	$\alpha A \beta \rightarrow \alpha \gamma \beta$	$\{a^n b^n c^n \mid n \geq 0\}$
2	Context-Free	$A \rightarrow \gamma$	$\{a^n b^n \mid n \geq 0\}$
3	Regular	$A \rightarrow \epsilon \mid a \mid aB$	$\{a^m b^n \mid m, n \geq 0\}$

Table 2.7: The Chomsky hierarchy.

The next chapter, which is tightly linked with the theory of formal languages, deals with the following question:

> Given a language of a particular type, what is the necessary complexity for an abstract machine to recognize the elements of that language?

2.7.1 Undecidable Problems

There are a number of interesting problems whose decidability is limited to a certain level in the Chomsky hierarchy. Here are four important ones:

1. **The recognition problem**
   Given a string w and a grammar G, is $w \in L(G)$?

2. **The emptiness problem**
   Given a grammar G, is $L(G) = \emptyset$?

3. **The equivalence problem**
   Given two grammars G_1 and G_2, is $L(G_1) = L(G_2)$?

4. **The ambiguity problem**
   Given a grammar G, is G ambiguous?

Table 2.8 indicates, for each of these problems and for each type of language in the Chomsky hierarchy, whether the problem is decidable or not.

Type	Problem: ($\surd$ = decidable, $\square$ = undecidable)			
	1. Recognition	2. Emptiness	3. Equivalence	4. Ambiguity
0	$\square$	$\square$	$\square$	$\square$
1	$\surd$	$\square$	$\square$	$\square$
2	$\surd$	$\surd$	$\square$	$\square$
3	$\surd$	$\surd$	$\surd$	$\surd$

Table 2.8: Decidability of problems in the Chomsky hierarchy.

2.8 Chapter Summary

- *Grammar and Language Terminology*: we introduced the basic notation and terminology required to describe formal grammar and languages.
- *Production Rules*: we showed how a finite set of production rules (a grammar) was a convenient way of describing a potentially infinitely large set of strings (a language).
- *Regular, Context-Free, Context-Sensitive and Unrestricted Grammars*: we introduced the four major families of formal grammars.
- *Regular Expressions*: we describe how regular expressions provide a convenient way of describing regular languages.
- *Syntax, Parse and Grammar Trees*: we defined various ways of graphically representing the syntactical structure, or the possible derivations of elements in a language.
- *Parsing and Ambiguity*: we learned some of the problems – as well as some method to remove them – encountered when trying to analyze the syntactical structure of an element in a language.
- *Chomsky Hierarchy*: we introduced the traditional hierarchy of grammars and languages, and mentioned the decidability for some problems.