Formal Methods II
Lecture Script

Fall Term 2009
University of Zurich, Switzerland

Rolf Pfeifer
Simon Bovet

Zürich, 2009

Contents

Chapter 1

Introduction

1.1 Introduction and Overview

The goal of this lecture series is to provide training in formal thinking to students of applied informatics. Students should learn to analyze and solve problems using formal methods. The class will expose students to a wide set of problems and show ways of solving them. Formal methods, as they are used in theoretical computer science, constitute an essential part of a computer science education, not only for those who target an academic or research career, but also for practitioners. Although the main goal of theoretical computer science, and the use of formal methods, is to gain insights, it turns out that many of the methods have direct practical applications. We will make an attempt to always illustrate the theoretical ideas with concrete examples and case studies – which is sometimes easier and sometimes a bit harder.

Because the curriculum is toward applied informatics, we only provide an overview of the field – on each of the chapters, we could have an entire course itself (e.g. on complexity, on logic, on automata and languages, etc.). Thus, it is not possible to always dig very deeply into the subject matter, but we do hope that this is sufficient to appreciate the issues involved and their implications.

We will cover the classical topics – formal languages, automata theory, etc. – but we have added a number of subjects that reflect our background in artificial intelligence, artificial life, and complex systems. They all require clear, formal thinking, and provide an additional valuable set of tools. For example, we have included sections on cellular automata, L-systems, and fractals which we feel, every computer scientist should know about. We also bring in the relatively recent

topic of network theory, which represents an extension of classical graph theory. Moreover, in the classical approach to artificial intelligence, the relation between computational models and theory of intelligence is often discussed; we will devote some final considerations to this question.

Theoretical computer science is built on the idea of computation as formalized by Alan Turing. While this idea of abstracting from the physical substrate and only considering the level of *computation* is extremely powerful – and has fundamentally changed society and our daily lives – it does have its limitations. This is why we also incorporated short chapters on *morphological computation* and on *quantum computing*, again concepts a computer science student should know about.

This script is only a short summary of contents treated in the lectures. Not all the material covered in class is covered here even though it may be relevant for the final examination. For certain topic areas, the students are referred to the pertinent literature for more detail.

Computer science – theoretical or applied – cannot be taught only theoretically, but active participation is extremely important. Consequently, we have included four substantial problem sheets that have the purpose to provide you with a deeper understanding of the ideas introduced. We do hope that the students will find them instructive and fun to do.

In terms of prerequisite, we expect the students to be familiar with the ideas taught during the course "Formal Methods, Part 1" which includes the following areas: information theory, Boolean algebra, predicate calculus, program verification, algorithmic complexity, trees, graphs, and relations. Some of these topics will be taken up again and deepened theoretically and through applications.

1.2 Acknowledgment

Rolf Pfeifer and his teaching assistants are grateful to Norbert Fuchs for generously providing materials and patiently answering our questions.

1.3 Outline of the Script

The topics which we will cover during the lecture – and thus the chapters of the present script – are structured and related to each other as follows (see Figure 1.1):

We will start with the **theory of formal languages**, which is about formalizing

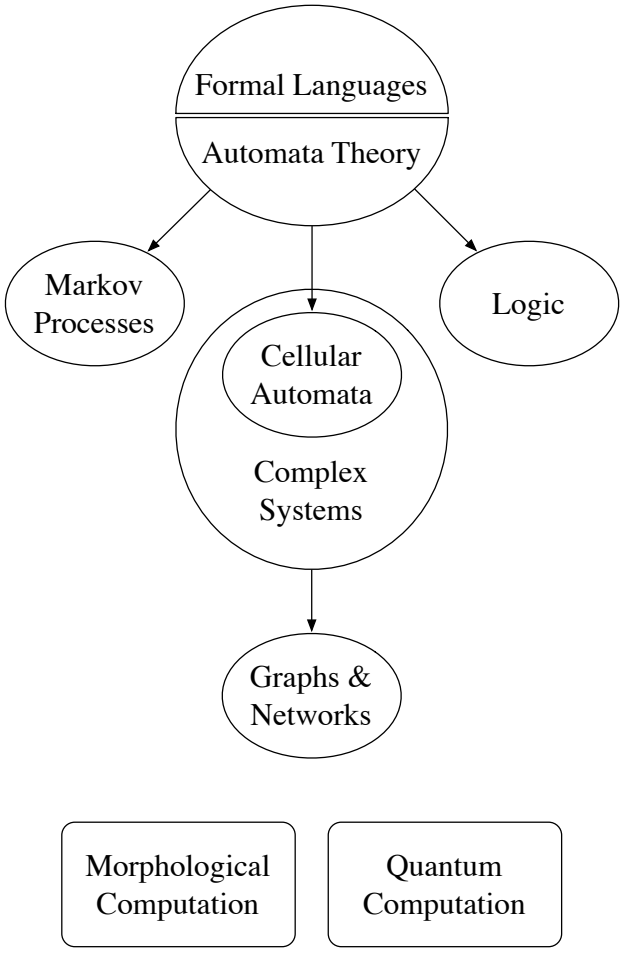


Figure 1.1: Overview of the topics that will be covered during the lecture.

the various structures found is virtually any text-based communication between any combination of humans and computers. We will not only learn the terminology, but also explore the various complexity levels found in formal languages.

The second topic, **automata theory**, is very closely related to formal languages: it deals with the various kinds of abstract machines capable of recognizing formal languages. In fact, the two topics of formal languages and automata theory constitute together the very basis of *computation*. Yet, this first part of the lecture is not only about theoretical computer science. We will also encounter various **applications**, many of which are found in the most modern software and devices.

We will explore different extensions of automata theory. The theory of **Markov processes** belongs to the probabilistic follows-up of automata theory, and has many applications, such as in text or speech recognition. **Logic** is also an extension of formal languages concerned with the question of how sentences in a formal language (such as mathematical theorems) can be logically derived from one another. We will also discuss how such a formalisms may allow computers to derive or proof theorems for us.

Another interesting extension of automata theory is the study of **cellular automata** – which are nothing else than many simple automata put on a grid next to each other. We will see that such systems have very surprising, even spectacular properties. In fact, we will see that cellular automata belong to a larger class of systems called **complex systems**.

An example of complex systems on which we will focus are **graphs and networks**. In this chapter, we will learn different techniques that enables us to deal with and characterize such kind of systems. We will also discuss some interesting effects found in different kind of static or growing networks.

Finally, we will conclude our journey through theoretical and applied computer science with two particular topics – **morphological** and **quantum computation** – which should provide insights on alternatives to the classical and prevalent theory of Turing computation.

1.4 Suggested Readings

Formal Languages and Automata Theory

- Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2001). *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, second edition.
- Sudkamp, T. A. (2006). *Languages and Machines: An introduction to the Theory of Computer Science*. Addison Wesley, third edition.

Logic and Models of Computation

- Papadimitriou, C. H. (1995). *Computational Complexity*. Addison Wesley.
- Rechenberg, P. and Pomberger, G. (2006). *Informatik Handbuch*. Hanser, fourth edition.

Fractals and Chaos

- Flake, G. W. (1998). *The Computational Beauty of Nature – Computer Explorations of Fractals, Chaos, Complex Systems, and Adaptation*. MIT Press.

Graphs and Networks

- Buchanan, M. (2002). *Nexus: Small Worlds and the Groundbreaking Science of Networks*. Norton Publishing. (Popular science)

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

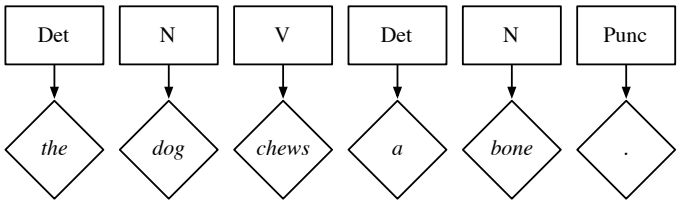
Table 2.1: Characteristics of Natural and Formal Languages.

2.2.1 Syntax

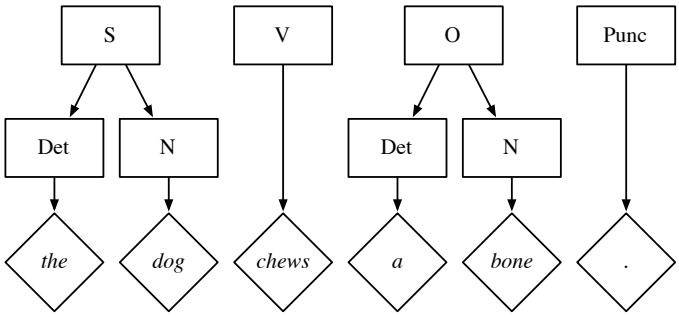
Everyone knows, after spending some years sitting on school banks, that a correct German or English sentence should respect the rules of its grammar (except if one tries to do poetry or modern art). Let's consider the following sentence:

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?

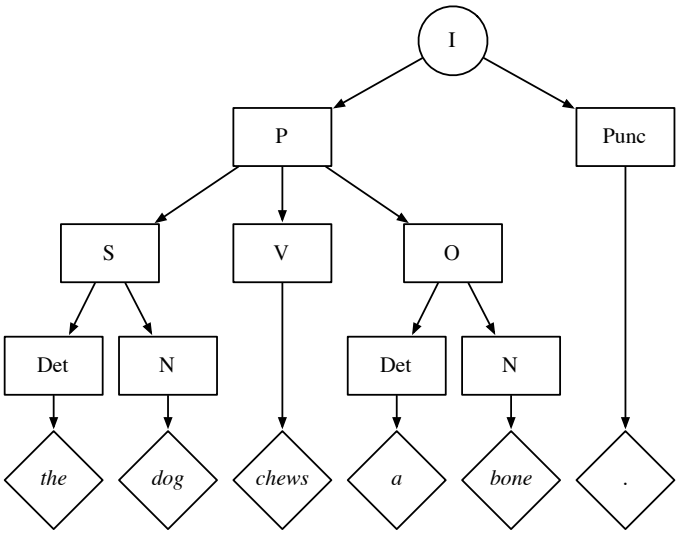
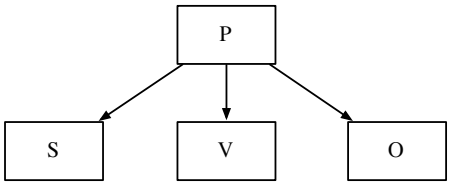


Figure 2.1: Tree representation of the syntax of an English phrase.

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:

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

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.

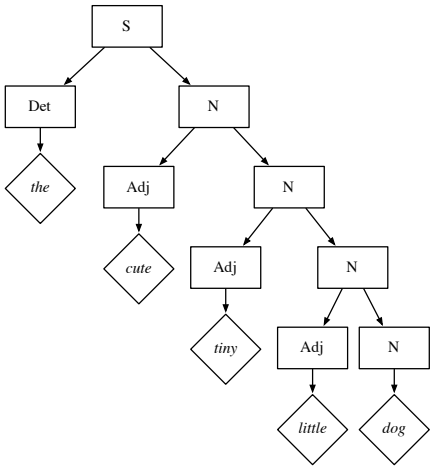


Figure 2.2: Derivation using a recursive rule.

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*:

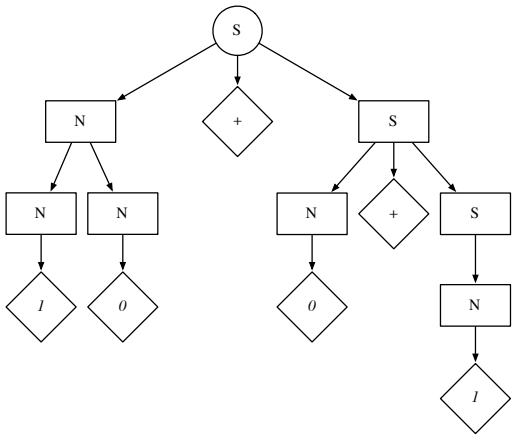


Figure 2.3: Syntax tree corresponding to one possible derivation of "10 + 0 + 1".

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: $L = \{\epsilon, ab, aabb, aaabbb, \dots\}$.

if the terminals on one of the two sides are missing, it's regular

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

$$A \to xy \ldots XY \ldots vw \ldots$$

and has to be recursive, either directly

$$A \to xAy$$

or indirectly

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

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

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

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*:

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

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>
```

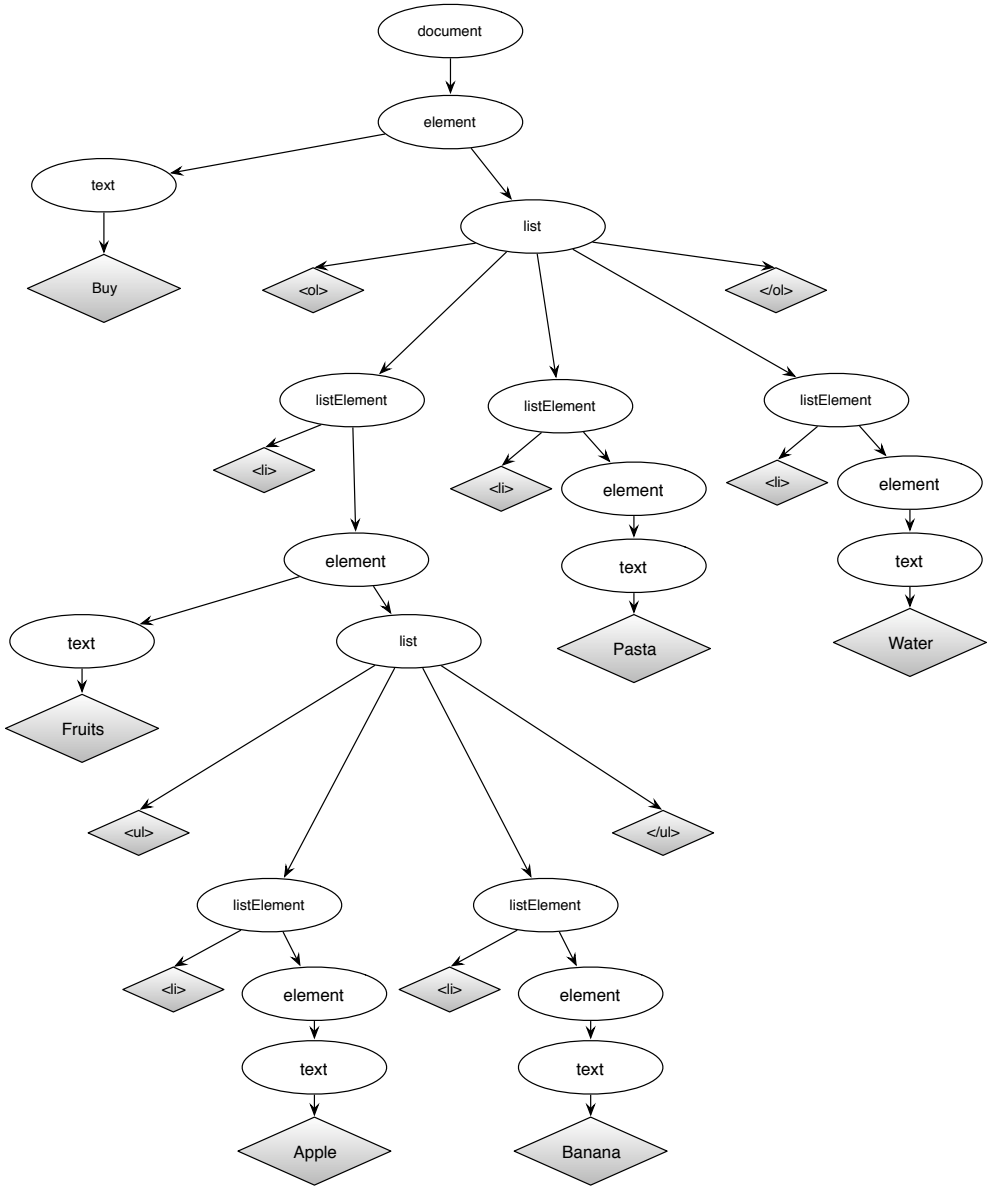
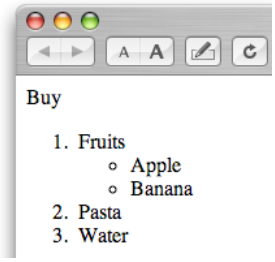


Figure 2.4: Syntax tree of a simplified HTML string.

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.

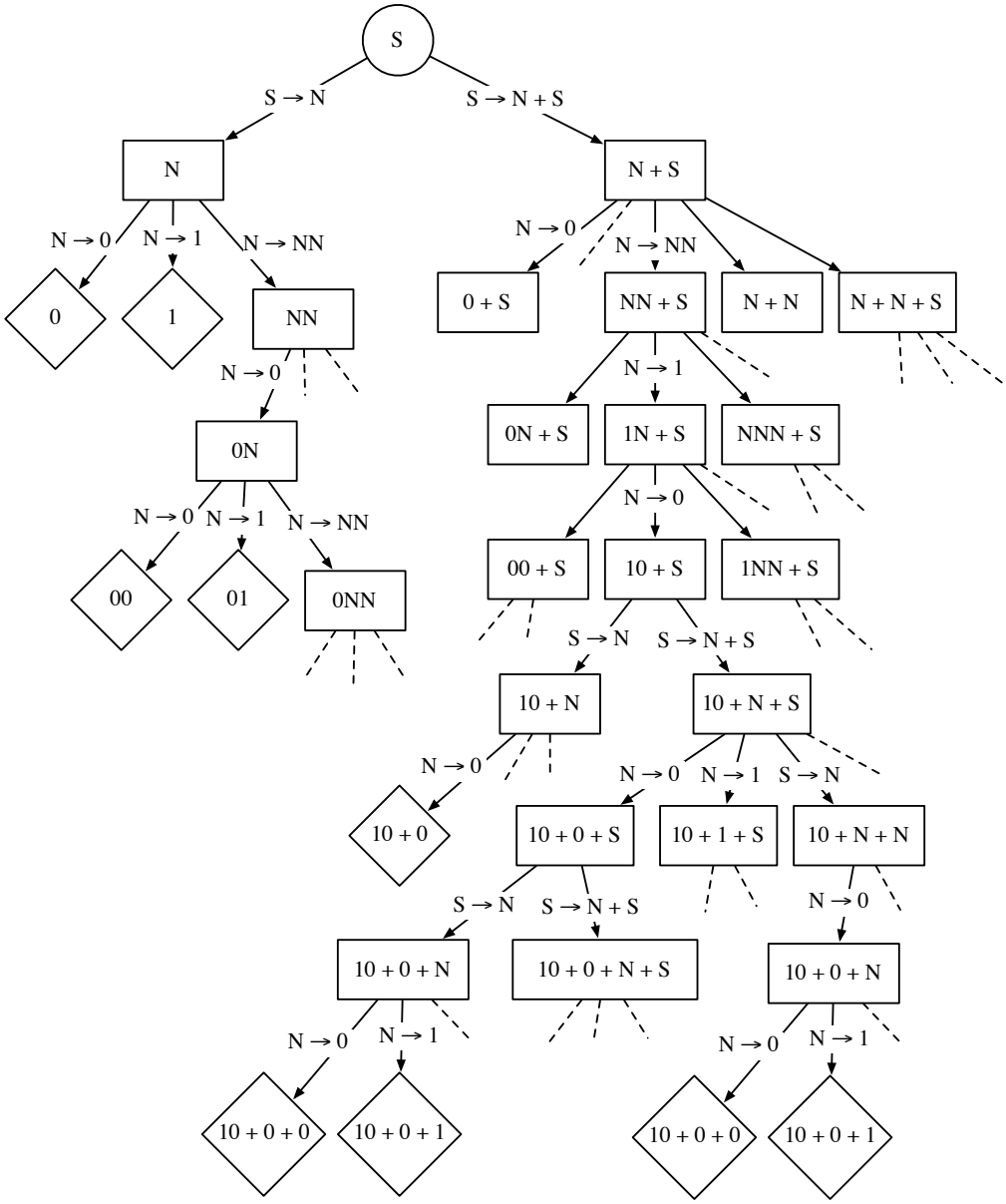


Figure 2.5: Partial grammar tree for the grammar $S \to N \mid N + S$, $N \to 0 \mid 1 \mid N$.

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.

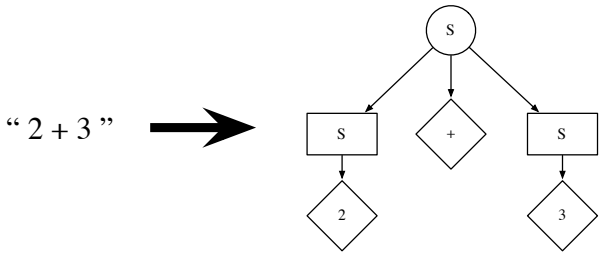


Figure 2.6: The process of parsing. Left: the input string. Right: the corresponding parse tree showing the syntactical structure.

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

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

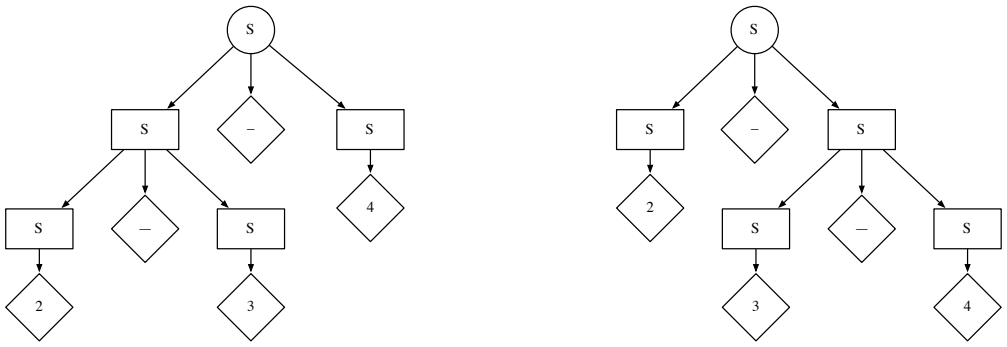


Figure 2.7: Two parse trees showing the two possible derivations of the input string "$2-3-4$".

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.

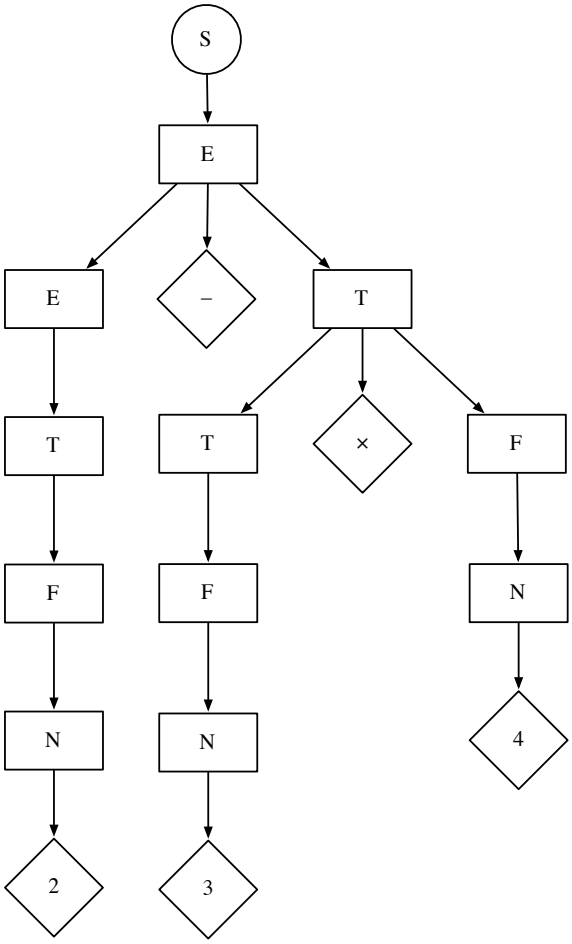


Figure 2.8: Parsing tree of the expression "$2 - 3 \times 4$" using an unambiguous grammar.

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.

Chapter 3

Automata Theory

3.1 Why Study Automata Theory?

The previous chapter provided an introduction into the theory of formal languages, a topic dealing with the syntax structures found in any text-based interaction.

The current chapter will study different kinds of abstract machines – so-called *automata* – able to recognize the elements of particular languages. As we will soon discover, these two chapters are profoundly coupled together, and constitute two parts of an almost indivisible whole.

On the one hand, this chapter will deal with many theoretical notions, such as the concept of *computation* which lies at the heart of automata theory. On the other hand, the exploration we're about to embark on will let us get to know several practical techniques and applications, such as taking advantages of compiler compilers, which will hopefully help extending one's panoply of essential tools.

3.2 Computation

The current chapter is basically concerned with the question:

"Does a particular string w belongs to a given language L or not?"

In fact, this very question constitutes the foundational definition of *computation*, whose theory gave birth to computers – those devices that changed our world.

The term *computation* carries nowadays various meanings, such as any type of information processing that can be represented mathematically.

3.2.1 Definition of Computation

Computation:
for EACH element of input set X
—> ONE element of an output set Y
|
|
reduce to…
|
|
for EACH elt. of the power set of {0,1}*
—> ONE element of {0,1}
i.e. we set X=ALL finite strings consisting of 0 and 1, and Y={0,1}
i.e. computation is the assignment of 0 or 1 to any finite string of 0 and 1
|
|
this is a decision problem
|
|
and the same as…
|
|
accepting strings of a language: if 0 is assigned to a string -> string not accepted;
if 1 is assigned to a string -> string accepted

In theoretical computer science, computation refers to a mapping that associates to each element of an input set X exactly one element in an output set Y. Without loss of generality, the definition of computation can be defined using the simple alphabet $\Sigma = \{0, 1\}$. The input set X can thus be defined as the Kleene closure of the alphabet $\{0, 1\}^*$. Interestingly, restricting the output set Y to $\{0, 1\}$ – i.e. considering only *"yes or no" decision problems* – doesn't really make any difference: even if we allow more complicated answers, the resulting problems are still equivalent to "yes or no" decision problems.

Computation thus deals mainly with decision problems of the type "given an integer n, decise whether or not n is a prime number" rather than (equivalent) problems of the type "given two real numbers x and y, what is their product $x \times y$".

Definition 3.1. A *decision problem* is a mapping

$$\{0, 1\}^* \longmapsto \{0, 1\}$$

that takes as input any finite string of 0's and 1's and assign to it an output consisting of either 0 ("no") or 1 ("yes").

Obviously, the theory computation and the theory of formal language are just two sides of the same coin: solving a decision problem is the same as accepting strings of a language (namely, the language of all strings that are mapped to 1).

3.2.2 $\mathcal{O}$ Notation

The theory of computation is also closely related to the theory of computational complexity, which is concerned with the relative computational difficulty of computable functions. The question there is how the resource requirements grow with input of increasing length. *Time* resource refers to the number of steps required, whereas *space* resource refers to the size of the memory necessary to perform the computation.

In fact, we are not interested in the *exact* number of time steps (or bits of memory) a particular computation requires (which depends on what machine and language is being used), but rather in *characterizing* how this number increases with larger input. This is usually done with help of the $\mathcal{O}$ notation:

Definition 3.2. The $\mathcal{O}$ notation (pronounce: big oh) stands for "order of" and is used to describe an asymptotic upper bound for the magnitude of a function in terms of another, typically simpler, function.

$f(x)$ is $\mathcal{O}(g(x))$ if and only if $\exists x_0, \exists c > 0$ such that $|f(x)| \leq c \cdot |g(x)|$ for $x > x_0$.

In other words: for sufficiently large x, $f(x)$ does not grow faster than $g(x)$, i.e. $f(x)$ remains smaller than $g(x)$ (up to a constant multiplicative factor).

The statement "$f(x)$ is $\mathcal{O}(g(x))$" as defined above is usually written as $f(x) = \mathcal{O}(g(x))$. Note that this is a slight abuse of notation: for instance $\mathcal{O}(x) = \mathcal{O}(x^2)$ but $\mathcal{O}(x^2) \neq \mathcal{O}(x)$. For this reason, some literature prefers the set notation and write $f \in \mathcal{O}(g)$, thinking of $\mathcal{O}(g)$ as the set of all functions dominated by g.

Example 3.1. If an algorithm uses exactly $3n^2 + 5n + 2$ steps to process an input string of length n, its time complexity is $\mathcal{O}(n^2)$.

Example 3.2. Searching a word in a dictionary containing n entries is not a problem with linear time complexity $\mathcal{O}(n)$: since the words are sorted alphabetically, one can obviously use an algorithm more efficient than searching each page starting from the first one (for instance, opening the dictionary in the middle, and then again in the middle of the appropriate half, and so on). Since doubling n requires only one more time step, the problem has a time complexity of $\mathcal{O}(\log(n))$. log has the basis 2

3.3 Finite State Automata

Imagine that you have to design a machine that, when given an input string consisting only of a's, b's and c's, tells you if the string contains the sequence "abc".

At first sight, the simplest algorithm could be the following:

1. Start with first character of the string.
2. Look if the three characters read from the current position are a, b and c.
3. If so, stop and return "yes".
4. Otherwise, advance to the next character and repeat step 2.
5. If the end of the string is reached, stop and return "no".

In some kind of pseudo-code, the algorithm could be written like this (note the two `for` loops):

```
input = raw_input("Enter a string: ")
sequence = "abc"
found = 0
for i in range(1, len(input) - 2):
    flag = 1
    for j in range(1, 3):
        if input[i + j] != sequence[j]:
           flag = 0
    if flag:
         found = 1

if found:
    print "Yes"
else:
    print "No"
```

If the length of the input string is n, and the length of the sequence to search for is k (in our case: $k = 3$), then the time complexity of the algorithm is $\mathcal{O}(n \cdot k)$.

Yet, there's room for improvement... Consider the following algorithm:

0. Move to the next character.
   If the next character is a, go to step 1.
   Otherwise, remain at step 0.

1. Move to the next character.
   If the next character is b, go to step 2.
   If the next character is a, remain at step 1.
   Otherwise, go to step 0.

2. Move to the next character.
   If the next character is c, go to step 3.
   If the next character is a, go to step 1.
   Otherwise, go to step 0.

3. Move to the next character.
   Whatever the character is, remain at step 3.

Listing 3.1: A more efficient algorithm to search for the sequence "abc".

If the end of the input string is reached on step 3, then the sequence "abc" has been found. Otherwise, the input string does not contain the sequence "abc".

Note that with this algorithm, each character of the input string is only read once. The time complexity of this algorithm is thus only $\mathcal{O}(n)$! In fact, this algorithm is nothing else than a *finite state automaton*.

3.3.1 Definition

A *finite state automaton* (plural: finite state automata) is an abstract machine that successively reads each symbols of the input string, and changes its state according to a particular control mechanism. If the machine, after reading the last symbol of the input string, is in one of a set of particular states, then the machine is said to *accept* the input string. It can be illustrated as follows:

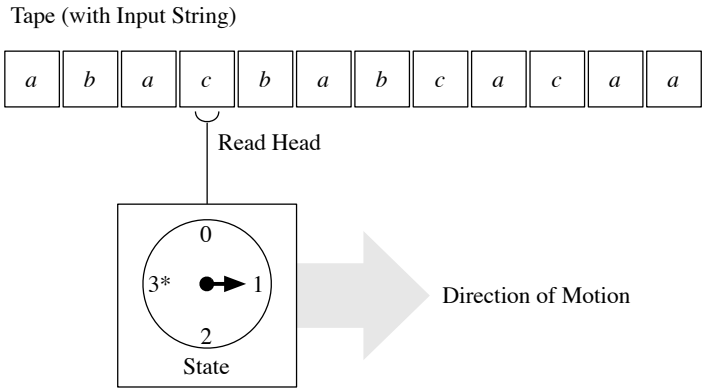


Figure 3.1: Illustration of a finite state automaton.

Definition 3.3. A *finite state automaton* is a five-tuple

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

consisting of:

1. Q: a finite set of states.
2. Σ: a finite set of input symbols.
3. $\delta : (q, s) \in Q \times \Sigma \longmapsto q' \in Q$: a transition function (or transition table) specifying, for each state q and each input symbol s, the next state q' of the automaton.

4. $q_0 \in Q$: the initial state.

5. $F \subset Q$: a set of accepting states.

The literature abounds with TLA's[1] to refer to finite state automata (FSA): they are also called *finite state machines* (FSM) or *deterministic finite automata* (DFA).

A finite state automaton can be represented with a table. The rows indicate the states Q, the columns the input symbols Σ, and the table entries the transition function δ. The initial state q_0 is indicated with an arrow $\rightarrow$, and the accepting states F are indicated with a star *.

Example 3.3. Consider the example used in the introduction of this section, namely the search for the sequence "abc" (see Listing 3.1). The corresponding finite state automaton is:

	a	b	c
$\rightarrow q_0$	q_1	q_0	q_0
q_1	q_1	q_2	q_0
q_2	q_1	q_0	q_3
$^*q_3$	q_3	q_3	q_3

3.3.2 State Diagrams

A finite state automaton can also be represented graphically with a *state* or *transition diagram*. Such as diagram is a graph where the nodes represent the states and the links between nodes represent the transitions. The initial state is usually indicated with an arrow, and the accepting states are denoted with double circles.

Example 3.4. The automata given in the previous Example 3.3 can be graphically represented with the state diagram shown in Figure 3.2.

3.3.3 Nondeterministic Finite Automata

The deterministic finite automata introduced so far are clearly an efficient way of searching some sequence in a string. However, having to specify all possible transitions is not extremely convenient. For instance, Figure 3.2 (especially among

[1] Three-Letter Abbreviations

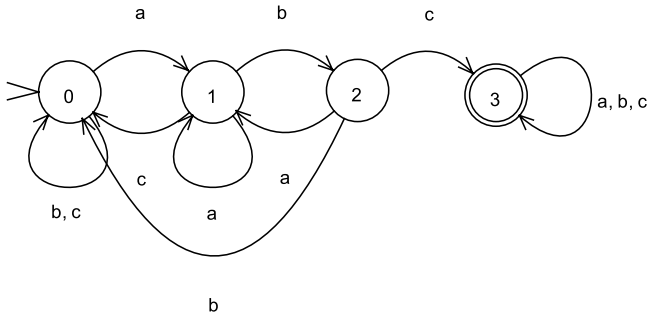


Figure 3.2: State diagram of a finite state automaton accepting the strings containing "abc".

the states q_0, q_1 and q_2) illustrates already with a simple case how this task can quickly become tedious.

A *nondeterministic* finite automaton (NFA) has the power to be in several states at once. For instance, if one state has more than one transition for a given input symbol, then the automaton follows simultaneously all the transitions and gets into several states at once. Moreover, if one state has no transition for a given input symbol, the corresponding state ceases to exist.

Figure 3.3 illustrates how a nondeterministic finite automaton can be substantially simpler than an equivalent deterministic finite automaton.

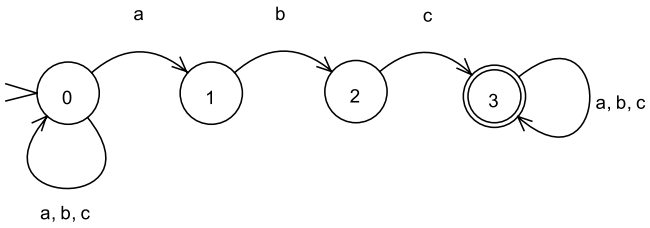


Figure 3.3: State diagram of a nondeterministic finite automaton accepting the strings containing "abc".

Definition 3.4. A *nondeterministic finite automaton* is a five-tuple

$$(Q, \Sigma, \hat{\delta}, q_0, F)$$

consisting of:

1. Q: a finite set of states.
2. Σ: a finite set of input symbols.
3. $\hat{\delta} : (q, s) \in Q \times \Sigma \longmapsto \{q_i, q_j, \ldots\} \subseteq Q$: a transition function (or transition table) specifying, for each state q and each input symbol s, the next state(s) $\{q_i, q_j, \ldots\}$ of the automaton.
4. $q_0 \in Q$: the initial state.
5. $F \subset Q$: a set of accepting states. An input w is accepted if the automaton, after reading the last symbol of the input, is in *at least one* accepting state.

Example 3.5. The transition table $\hat{\delta}$ of the nondeterministic finite automaton shown in Figure 3.3 is the following:

	a	b	c
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$	$\{q_0\}$
q_1	$\emptyset$	$\{q_2\}$	$\emptyset$
q_2	$\emptyset$	$\emptyset$	$\{q_3\}$
$^*q_3$	$\{q_3\}$	$\{q_3\}$	$\{q_3\}$

3.3.4 Equivalence of Deterministic and Nondeterministic Finite Automata

Clearly, each deterministic finite automaton (DFA) is already a *nondeterministic* finite automaton (NFA), namely one that happens to always be in just one state. A surprising fact, however, is the following theorem:

Theorem 3.1. *Every language that can be described by an NFA (i.e. the set of all strings accepted by the automaton) can also be described by some DFA.*

Proof. The proof that DFA's can do whatever NFA's can do consists in showing that for every FNA, a DFA can be constructed such that both accepts the same languages. This so-called "subset construction proof" involves constructing all subset of the set of state of the NFA.

In short, since an NFA can only be in a finite number of states simultaneously, it can be seen as a DFA where each "superstate" of the DFA corresponds to a set of states of the NFA.

Let $N = \{Q_N, \Sigma, \hat{\delta}_N, q_0, F_N\}$ be an NFA. We can construct a DFA $D = \{Q_D, \Sigma, \delta_D, q_0, F_D\}$ as follows:

- $Q_D = 2^{Q_N}$. That is, Q_D is the *power set* (the set of all subsets) of Q_N.
- $\delta_D(S, a) = \bigcup_{p \in S} \hat{\delta}_N(p, a)$
- $F_D = \left\{ S \subset Q_N \;\middle|\; S \cap F_N \neq \emptyset \right\}$. That is, F_D is all sets of N's states that include at least one accepting state of N.

It can "easily" be seen from the construction that both automata accept exactly the same input sequences. □

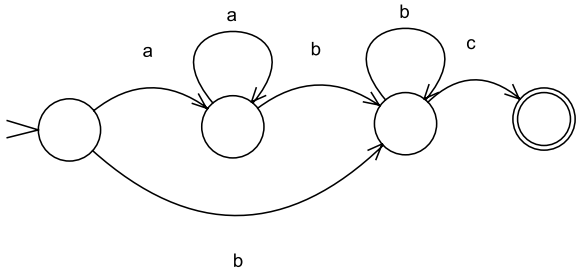
Example 3.6. Figure 3.4 shows an NFA that accepts strings over the alphabet $\Sigma = \{0, 1, 2, \ldots, 9\}$ containing dates of the form "19?0", where ? stands for any possible digit. Figure 3.5 shows an equivalent DFA.

3.3.5 Finite Automata and Regular Languages

We have seen so far different examples of finite automata that can accept strings that contain one or more substrings, such as "abc" or "19?0". Recalling what wew have learned in Chapter 2, we note that the corresponding languages are in fact regular, such as the language corresponding to the regular expression

$$(a \mid b \mid c)^* \; abc \; (a \mid b \mid c)^*$$

More generally, it can be proven that for each regular expression, there is a finite automaton that defines the same regular language. Rather than providing a proof, which can be found for instance in Hopcroft et al. (2001), we only give here an illustrating example. It can be seen that the following NFA defines the same language as the regular expression a^*b^+c: a, zero or more times, followed by b, one or more times, followed by a single c.



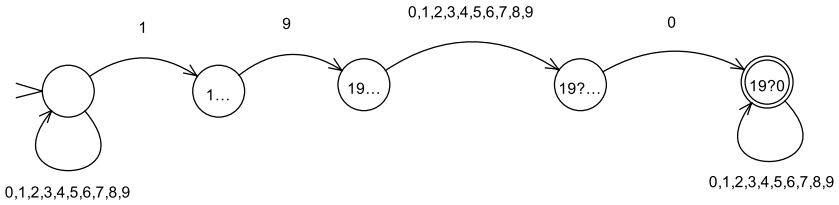


Figure 3.4: An NFA that searches for dates of the form "19?0".

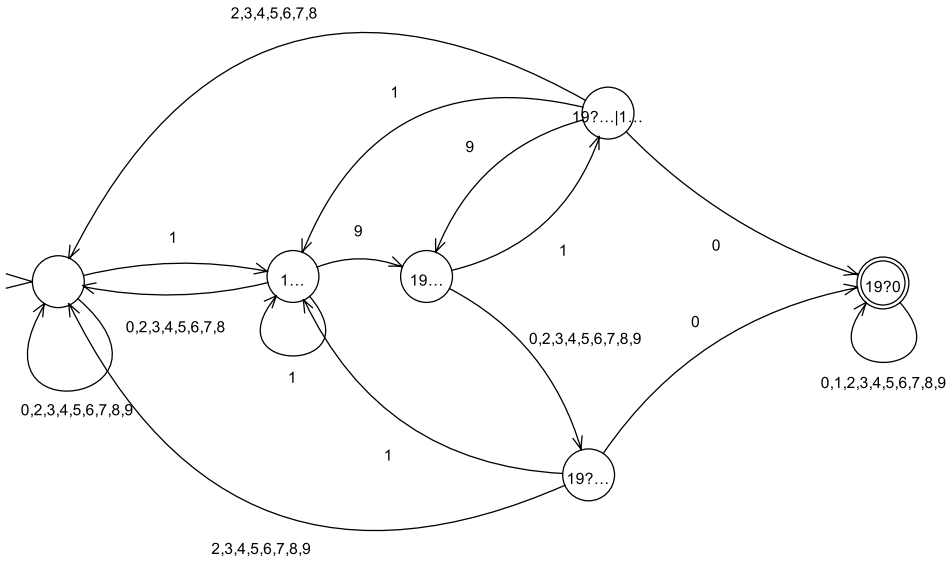


Figure 3.5: Conversion of the NFA from Figure 3.4 to a DFA.

In summary, it can be shown that any regular language satisfies the following equivalent properties:

- It can be generated by a *regular grammar*.
- It can be described by a *regular expression*.
- It can be accepted by a *deterministic finite automaton*.
- It can be accepted by a *nondeterministic finite finite automaton*.

3.3.6 The "Pumping Lemma" for Regular Languages

In the theory of formal languages, a *pumping lemma* for a given class of languages states a property that all languages in the class have – namely, that they can be "pumped". A language can be pumped if any sufficiently long string in the language can be broken into pieces, some of which can be repeated arbitrarily to produce a longer string that is still in the language.

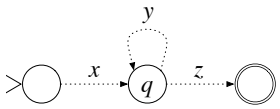
These lemmas can be used to determine if a particular language is not in a given language class. One of the most important examples is the pumping lemma for regular languages, which is primarily used to prove that there exist languages that are not regular (that's why it's called a "lemma" – it is a useful result for proving other things).

Lemma 3.1. *Let L be a regular language. Then, there exists a constant n (which depends on L) such that for every string $w \in L$ with n or more symbols ($|w| \geq n$), we can break w into three strings, $w = xyz$, such that:*

1. $|y| > 0$ *(i.e. $y \neq \epsilon$)*
2. $|xy| \leq n$
3. *For all $k \geq 0$, the string xy^kz is still in L. ($y^k := \underbrace{yy\ldots y}_{k \text{ times}}$)*

In other words, for every string w of a certain minimal length, we can always find a nonempty substring y of w that can be "pumped"; that is, repeating y any number of times, or deleting it (the case of $k = 0$), keeps the resulting string in the language L.

Proof. The proof idea uses the fact that for every regular language there is a finite state automaton (FSA) that accepts the language. The number of states in this FSA are counted, and then, that count is used as the pumping length n. If the string's length is longer than n, then there must be at least one state that is repeated (which we will call state q):



The transitions that take the automaton from state q back to state q match some string. This string is called y in the lemma, and since the machine will match a string without the y portion, or the string y can be repeated, the conditions of the lemma are satisfied. □

We're going to use this lemma to prove that a particular language L_{eq} is not regular. L_{eq} is defined over the alphabet $\{0, 1\}$ as the set of all strings that contain as many 0's and 1's:

$$L_{\text{eq}} = \{\epsilon, 01, 10, 0011, 0101, 1010, 1100, 000111, 001011, \ldots\}$$

To show: find a string in Leq for which there is NO substring that can be pumped.

We'll use this lemma in a *proof by contradiction*. If we assume that L_{eq} is regular, then the pumping lemma for regular languages must hold for all strings in L_{eq}. By showing that there exists an instance where the lemma does not hold, we show that our assumption is incorrect: L_{eq} cannot be regular.

Statements with multiple quantifiers

Let's first look at what it means for the lemma not to hold:[2]

1. There exists a language L_{eq} assumed to be regular. Then...
2. for all n...
3. there exists a string $w \in L_{\text{eq}}$, $|w| \geq n$ such that...
4. for all possible decompositions $w = xyz$ with $|y| > 0$ and $|xy| \leq n$...
5. there is a $k \geq 0$ so that the string xy^kz is *not* in L_{eq}.

[2]Remember that the opposite of "for all x, y" is "there exists an x so that not y", and vice versa: $\neg(\forall x \Rightarrow y) \equiv (\exists x \Rightarrow \neg y)$ and $\neg(\exists x \Rightarrow y) \equiv (\forall x \Rightarrow \neg y)$.

¬(pumping lemma)

x: string in Leq with length >= n
y: there is a substring that can be pumped

If is often helpful to see statements with more than one quantifier as a game between two players – one player A for the "there exists", and another player B for the "for all" – who take turns specifying values for the variables mentioned in the theorem. We'll take the stand of player A – being smart enough to find, for each "there exists" statement, a clever choice – and show that it can beat player B for any of his choices.

Player A:	We choose L_{eq} and assume it is regular.
Player B:	He returns some n.
Player A:	We choose the following, particular string: $w = 0^n 1^n$. Obviously, $w \in L_{\text{eq}}$ and $\|w\| \geq n$.
Player B:	He returns a particular decomposition $w = xyz$ with $\|y\| > 0$ and $\|xy\| \leq n$.
Player A:	Since $\|xy\| \leq n$ and xy is at the front of our w, we known that x and y only consist of 0's. We choose $k = 0$, i.e. we remove y from our string w. Now, the remaining string xz does obviously contain less 0's than 1's, since it "lost" the 0's from y (which is not the empty string). Thus, for $k = 0$, $xy^k z \notin L_{\text{eq}}$.

The lemma has been proven wrong. Thus, it can only be our initial assumption which is wrong: L_{eq} cannot be a regular language. □

3.3.7 Applications

Before moving to the next kind of abstract machine, let us briefly list a number of applications of finite state automata:

- Regular expressions
- Software for scanning large bodies of text, such as collections of web pages, to find occurrences of words or patterns
- Communication protocols (such as TCP)
- Protocols for the secure exchange of information
- Stateful firewalls (which build on the 3-way handshake of the TCP protocol)
- Lexical analyzer of compilers

3.4 Push-Down Automata

We have seen in the previous section that finite state automata are capable of recognizing elements of a language such as $L = \{a^n \mid n \text{ even}\}$ – for instance simply by switching between an "odd" state and an "even" state.

Yet they are also proven to be too simple to recognize the elements of a language such as $L = \{a^n b^n \mid n > 0\}$. At least for this language, the reason can be intuitively understood. n can be arbitrarily large, but the finite state automaton only has a finite number of states: it thus cannot "remember" the occurrences of more than a certain number of a's.

The next kind of abstract machine we will study consists of finite state automata endowed with simple external memory – namely a *stack*, whose top element can be manipulated and used by the automaton to decide which transition to take:

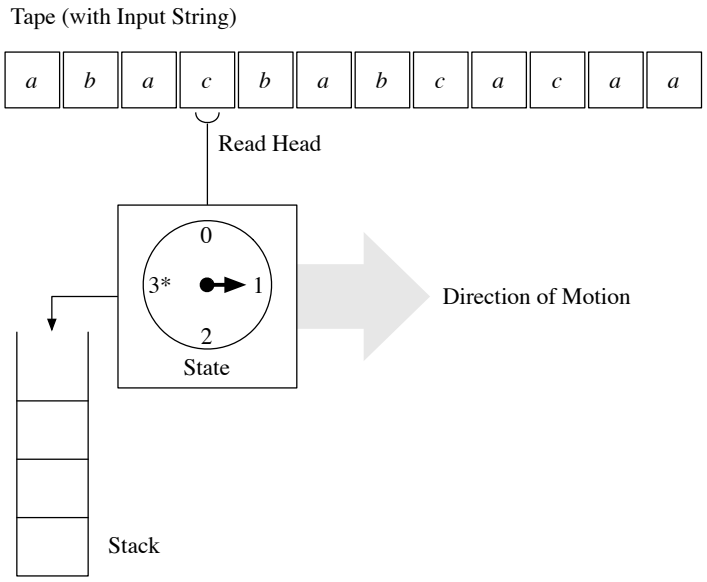


Figure 3.6: Illustration of a push-down automaton.

Push-down automata *per se* offer only little interest: what use is a machine that only recognize whether or not a text – such as the source code of a program – has a correct syntax? Yet, the concept of a push-down automaton lies very close to the concept of an *interpreter*, which not only checks the syntax, but also interprets a text or executes the instructions of a source code.

3.4.1 Parser Generator

Section 2.5.3 introduces the concept of *parsing*, i.e. the process of transforming input strings into their corresponding parse trees according to a given grammar. A *parser* is a component that carries out this task:

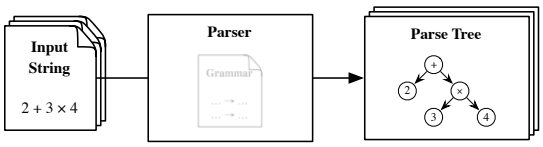


Figure 3.7: A parser transforms input strings into parse trees according to a given grammar.

From an abstract point of view, a parser implements a push-down automaton corresponding to the given grammar. An interesting, and for computer scientists very convenient consequence, which follows from the low complexity of push-down automata, is that it is possible to *automatically* generate a parser by only providing the grammar!

This is in essence what a *parser generator* does:

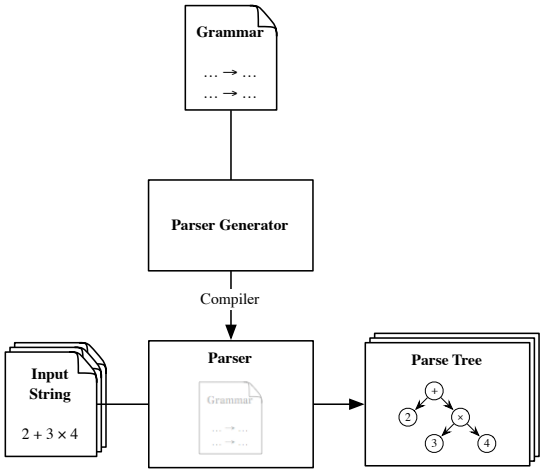


Figure 3.8: A parser generator creates a parser from a given grammar.

Note that the output of a parser generator usually needs to be compiled to create an executable parser.

Example 3.7. `yacc` is a parser generator that uses a syntax close to the EBNF. The following (partial) listing illustrates an input grammar that can be transformed into a parser for simple mathematical expressions using integers, the operators "+", "−" and "×" as well as parenthesis "(" and ")":

```
%token DIGIT

  line : expr
       ;

  expr : term
       | expr '+' term
       | expr '-' term
       ;

  term : factor
       | term 'x' factor
       ;

factor : number
       | '(' expr ')'
       ;

number : DIGIT
       | number DIGIT
```

3.4.2 Compiler Compilers

Even though parsers are more exciting than push-down automata, they nevertheless offer only limited interest, since they can only verify whether input strings comply to a particular syntax.

In fact, the only piece of missing information needed to process a parsing tree is the *semantics* of the production rules. For instance, the semantics of the following production rule

$$T \to T \times F$$

is to multiply the numerical value of the term T with the numerical value of the factor F.

A *compiler compiler* is a program that

- takes as input a grammar complemented with atomic *actions* for each of its production rule, and
- generates a compiler (or to be more exact, a interpreter) that not only checks for any input string the correctness of its syntax, but also evaluates it.

Figure 3.9 illustrates what a compiler compiler does. The very interesting advantage of a compiler compiler is that you only have to provide the specifications of the language you want to implement. In other words, you only provide the "what" – i.e. the grammar rules – and the compiler compiler automatically produces the "how" – i.e. takes care of all the implementation details to implement the language.

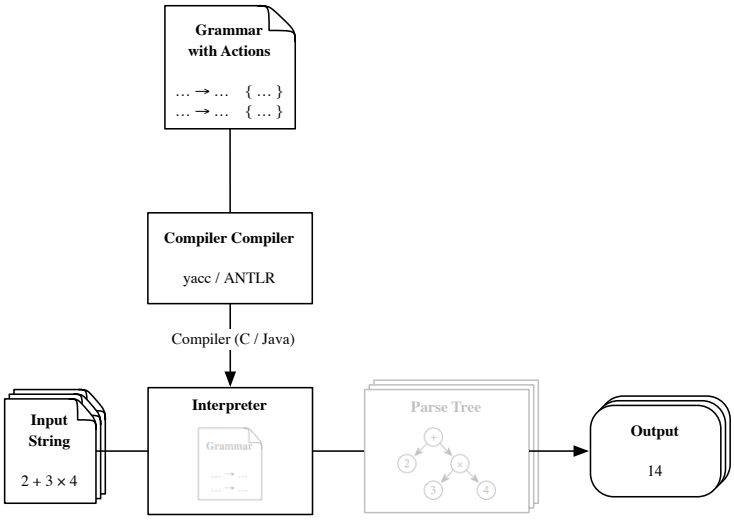


Figure 3.9: Overview of what a compiler compiler does. From a given grammar complemented with atomic actions (the "what"), it automatically generates a interpreter that checks the syntax and evaluates any number of input string (the "how").

Example 3.8. `yacc` is in fact also a compiler compiler (it is the acronym of "yet another compiler compiler"). The following lines illustrates how each production rule of the grammar provided in Example 3.7 can be complemented with atomic actions:

```
line : expr               { printf("result: %i\n", $1); }
     ;

expr : term               { $$ = $1; }
     | expr '+' term      { $$ = $1 + $3; }
     | expr '-' term      { $$ = $1 - $3; }
     ;

term : factor             { $$ = $1; }
```

```
        | term 'x' factor    { $$ = $1 * $3; }
        ;

factor : number             { $$ = $1; }
        | '(' expr ')'      { $$ = $2; }
        ;

number : DIGIT              { $$ = $1; }
        | number DIGIT      { $$ = 10 * $1 + $2; }
```

The point of this example is that the above listing provides in essence all what is required to create a calculator program that correctly evaluate any input string corresponding to a mathematical expression, such as "$2 + 3 \times 4$" or "$((12 - 4) \times 7 + 42) \times 9$".

3.4.3 From yacc to ANTLR

yacc offers a very powerful way of automatically generating a fully working program – the "how" – just from the description of the grammar rules that should be implemented – the "what". Yet, a quick glance at the cryptic code generated by yacc with immediately reveal an intrinsic problem: based on the implementation of a finite state machine, the code is completely opaque to any human programmer. As a result, the code cannot be re-used (we shouldn't expect a full-fledged program to be generated automatically, but only some parts of it), extended or even debugged (in the case that there is an error in the grammar). This severe limitation turned out to be fatal for yacc: nowadays, it is basically only used by the creators of this tool, or by some weird lecturers to implement toy problems in computer science classes.

The landscape has dramatically changed since the emergence of new, modern and much more convenient tools, such as ANTLR[3]. This tool not only provides a very powerful framework for constructing recognizers, interpreters, compilers, and translators from grammatical descriptions containing actions, but also generates a very transparent code. In addition, ANTLR has a sophisticated grammar development environment called ANTLRWorks, which we will be demoing during the lecture. Nowadays, ANTLR is used by hundreds of industrial and software companies.

[3] http://www.antlr.org/

3.4.4 Understanding Parsing as "Consuming and Evaluating"

The code generated by ANTLR is based on a general idea that is quite useful to know. In essence, the code is based on the concept of *parsing* as *consuming and evaluating*. It consists of multiple functions, each implementing a particular grammar rule. Each function parses a portion of the input string by consuming as many characters of the input string as it can, and returning the value corresponding to the evaluation of what has been consumed.

Example 3.9. The grammar rule

```
number : DIGIT              { $$ = $1; }
       | number DIGIT       { $$ = 10 * $1 + $2; }
```

can be implemented by the following code:

```
def scanNumber():
    n = 0
    ok, d = scanDigit()
    if not ok:
        return False, 0
    while ok:
        n = 10 * n + d
        ok, d = scanDigit()
    return True, n

def scanDigit()
    if scanCharacter('0'):
        return True, 0
    if scanCharacter('1'):
        return True, 1

    [...]

    if scanCharacter('9'):
        return True, 9
    return False, 0
```

Each function returns two values: 1) a boolean indicating whether anything could be scanned, and 2) a number corresponding to the evaluation of the scanned characters. The function `scanCharacter(c)` returns whether or not the character `c` could be consumed from the input string (in which case the character is consumed).

Similarly, the following grammar rules

```
  term : factor             { $$ = $1; }
       | term 'x' factor    { $$ = $1 * $3; }
       ;

factor : number             { $$ = $1; }
       | '(' expr ')'       { $$ = $2; }
       ;
```

can be implemented by a code with the same structure:

```
def scanTerm():
    ok, x = scanFactor()
    if not ok: error()
    while scanChar('x'):
        ok, y = scanFactor()
        if not ok: error()
        x = x * y
    return True, x

def scanFactor():
    if scanChar('('):
        ok, x = scanExpression()
        if not ok: error()
        if not scanChar(')'): error()
        return True, x
    else:
        ok, x = scanNumber()
        return ok, x
```

3.5 Turing Machines

Let us go back, for the remaining part of this chapter, to the formal aspect of automata theory.

We have seen that a push-down automata is an abstract machine capable of recognizing context-free languages, such as $L = \left\{a^n b^n \;\middle|\; n > 0\right\}$ – for instance by pushing (i.e. adding) a token on the stack each time an a is read, popping (i.e. removing) a token from the stack eacheach time a b is read, and verifying at the end that no token is left on the stack.

However, push-down automata also have limits. For instance, it can be shown that they cannot recognize the elements of the context-sensitive language $L = \left\{a^n b^n c^n \;\middle|\; n > 0\right\}$ – intuitively, we see that the stack only allows the automaton to verify that there are as many a's and b's, but the machine cannot "remember" what their number was when counting the c's.

Surprisingly, the next type of automaton we're about to (re)discover looks in some respects even simpler than a push-down automaton. The "trick" consists basically in allowing the machine (a) to move in both directions and (b) read *and* write on the tape (see Figure 3.10).

Definition 3.5. A *Turing machine* is an automaton with the following properties:

- A **tape** with the input string initially written on it. Note that the tape can be potentially infinite on both sides, but the number of symbols written at any time on the tape is always finite.

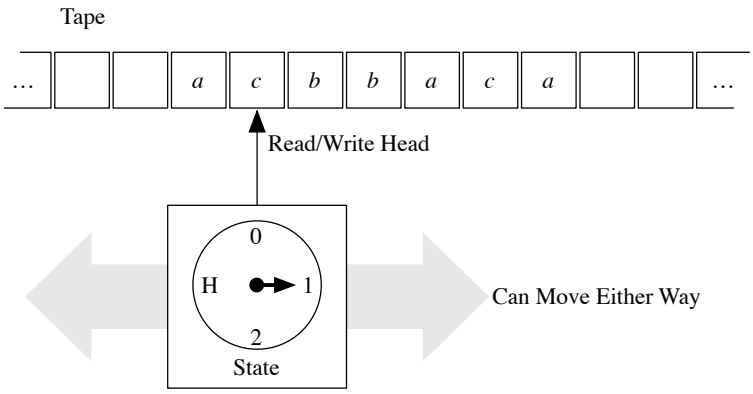


Figure 3.10: Illustration of a Turing machine.

- A **read-write head**. After reading the symbol on the tape and overwriting it with another symbol (which can be the same), the head moves to the next character, either on the left or on the right.

- A **finite controller** that specifies the behavior of the machine (for each state of the automaton and each symbol read from the tape, what symbol to write on the tape and which direction to move next).

- A **halting state**. In addition to moving left or right, the machine may also *halt*. In this case, the Turing machine is usually said to *accept* the input. (A Turing machine is thus an automaton with only one accepting state H.)

The initial position of the Turing machine is usually explicitly stated (otherwise, the machine can for instance start by moving first to the left-most symbol).

Example 3.10. The reader is left with the exercise of finding out what the following Turing machine does on a tape containing symbols from the alphabet $\{a, b, c\}$.

The rows correspond to the state of the machine. Each cell indicates the symbol to write on the tape, the direction in which to move (L or R) and the next state of the machine.

	a	b	c	$\sqcup$
0	a L 0	b L 0	c L 0	$\sqcup$ R 1
1	b R 1	a R 1	c R 1	Halt

- Go left until reading a space
- From left to right, replace each a with a b, and each b with an a
- Halt when a space is read

3.5.1 Recursively Enumerable Languages

Interestingly, a Turing machine already provides us with a formidable computational power. In fact, it can be shown that Turing machines are capable of recognizing the elements generated by any *unrestricted grammar*. Note however that the corresponding languages aren't called "unrestricted languages" – there still are many languages that just cannot be generated by any grammar! – but *recursively enumerable languages*.

Definition 3.6. A *recursively enumerable set* is a set whose members can simply be numbered. More formally, a recursively enumerable set is a set for which there exist a mapping between every of its elements and the integer numbers.

It is in fact surprising that the three definitions of recursively enumerable languages – namely (a) the languages whose elements can be enumerated, (b) the languages generated by any unrestricted grammar, and (c) the languages accepted by Turing machines – are in fact equivalent!

3.5.2 Linear Bounded Turing Machines

NFA, DFA Pushdown Automaton Turing Machine

We have seen so far that languages defined by regular, context-free and unrestricted grammars each have a corresponding automaton that can be used to recognize their elements. What about the in-between class of context-sensitive grammars?

The automaton that corresponds to context-sensitive grammars (i.e. that can recognize elements of context-sensitive languages) are so-called *linear bounded Turing machines*. Such a machine is like a Turing machine, but with one restriction: the length of the tape is only $k \cdot n$ cells, where n is the length of the input, and k a constant associated with the machine.

3.5.3 Universal Turing Machines

One of the most astonishing contribution of Alan Turing is the existence of what he called "universal Turing machines".

Definition 3.7. A *universal Turing machine* U is a Turing machine which, when supplied with a tape on which the code of some Turing machine M is written (together with some input string), will produce the same output as the machine M. on the input string

In other words, a universal Turing machine is a machine which can be *programmed* to simulate any other possible Turing machine. We now take this remarkable finding for granted. But at the time (1936), it was so astonishing that it is considered by some to have been the fundamental theoretical breakthrough that led to modern computers.

3.5.4 Multitape Turing Machines

There exists many variations of the standard Turing machine model, which appear to increase the capability of the machine, but have the same language-recognizing power as the basic model of a Turing machine.

One of these, the multitape Turing machine (see Figure 3.11), is important because it is much easier to see how a multitape Turing machine can simulate real computers, compared with the single-tape model we have been studying. Yet the extra tapes add no power to the model, as far as the ability to accept languages is concerned.

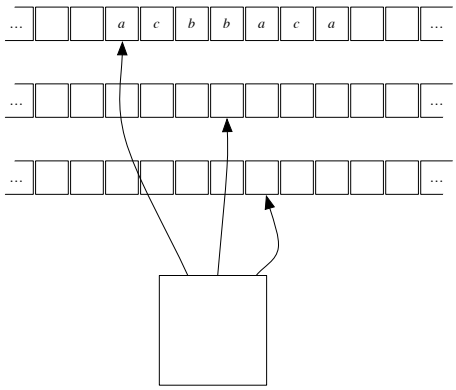


Figure 3.11: Illustration of a multitape Turing machine. At the beginning, the input is written on one of the tape. Each head can move independently.

3.5.5 Nondeterministic Turing Machines

Nondeterministic Turing machines (NTM) are to standard Turing machines (TM) what nondeterministic finite automata (NFA) are to deterministic finite automata (DFA).

A nondeterministic Turing machine may specify any finite number of transitions for a given configuration. The components of a nondeterministic Turing machine, with the exception of the transition function, are identical to those of the standard Turing machine. An input string is accepted by a nondeterministic Turing machine if there is at least one computation that halts.

As for finite state automata, nondeterminism does not increase the capabilities of Turing computation; the languages accepted by nondeterministic machines are precisely those accepted by deterministic machines:

Theorem 3.2. *If* M_N *is a nondeterministic Turing machine, then there is a deterministic Turing machine* M_D *such that* $L(\mathrm{M}_N) = L(\mathrm{M}_D)$.

Notice that the constructed deterministic Turing machine may take exponentially more time than the nondeterministic Turing machine.

The difference between deterministic and nondeterministic machines is thus a matter of time complexity (i.e. the number of steps required), which leads us to the next subsection.

3.5.6 The $\mathrm{P} = \mathrm{NP}$ Problem

An important question is theoretical computer science is how the number of steps required to perform a computation grows with input of increasing length (see Section 3.2.2). While some problems require polynomial time to compute, there seems to be some really "hard" problems that seem to require exponential time to compute.

Another way of expressing "hard" problems is to distinguish between *computing* and *verifying*. Indeed, it can be shown that decision problems solvable in polynomial time on a nondeterministic Turing machine can be "verified" by a deterministic Turing machine in polynomial time.

Example 3.11. The *subset-sum problem* is an example of a problem which is easy to verify, but whose answer is believed (but not proven) to be difficult to compute.

Given a set of integers, does some nonempty subset of them sum to 0? For instance, does a subset of the set $\{-2, -3, 15, 14, 7, -10\}$ add up to 0? The answer is yes, though it may take a while to find a subset that does, depending on its size. On the other hand, if someone claims that the answer is "yes, because $\{-2, -3, -10, 15\}$ add up to zero", then we can quickly check that with a few additions.

Definition 3.8. $\mathbf{P}$ is the complexity class containing decision problems which can be solved by a deterministic Turing machine using a polynomial amount of computation time, or polynomial time.

Definition 3.9. $\mathbf{NP}$ – nondeterministic, polynomial time – is the set of decision problems solvable in polynomial time on a nondeterministic Turing machine. Equivalently, it is the set of problems whose solutions can be "verified" by a deterministic Turing machine in polynomial time.

The relationship between the complexity classes $\mathbf{P}$ and $\mathbf{NP}$ is a famous unsolved question in theoretical computer science. It is generally agreed to be the most important such unsolved problem; the Clay Mathematics Institute has offered a US$1 million prize for the first correct proof.

In essence, the $\mathbf{P} = \mathbf{NP}$ question asks: if positive solutions to a "yes or no" problem can be verified quickly (where "quickly" means "in polynomial time"), can the answers also be computed quickly?

3.5.7 The Church–Turing Thesis

The computational power of Turing machine seems so large that it lead many mathematicians to conjecture the following thesis, known now as the *Church–Turing thesis*, which lies at the basis of the theory of computation:

> Every function which would naturally be regarded as computable can be computed by a Turing machine.

Due to the vagueness of the concept of effective calculability, the Church–Turing thesis cannot formally be proven. There exist several variations of this thesis, such as the following *Physical Church–Turing thesis*:

> Every function that can be physically computed can be computed by a Turing machine.

3.5.8 The Halting Problem

It is nevertheless possible to formally define functions that are not computable. One of the best known example is the halting problem: given the code of a Turing machine as well as its input, decide whether the machine will halt at some point or loop forever.

It can be easily shown – but constructing a machine that solves the Halting problem on itself – that a general algorithm to solve the halting problem for all possible program-input pairs cannot exist.

It is interesting to note that adjectives such as "universal" have often lead to the misconception that "Turing machines can compute everything". This is a truncated statement, missing "...that is computable" – which does actually nothing else than defining the term "computation"!

3.6 The Chomsky Hierarchy Revisited

We have seen throughout Chapters 2 and 3 that there is a close relation between languages, grammars and automata (see Figure 3.12). Grammars can be used to describe languages and generate their elements. Automata can be used to recognize elements of languages and to implement the corresponding grammars.

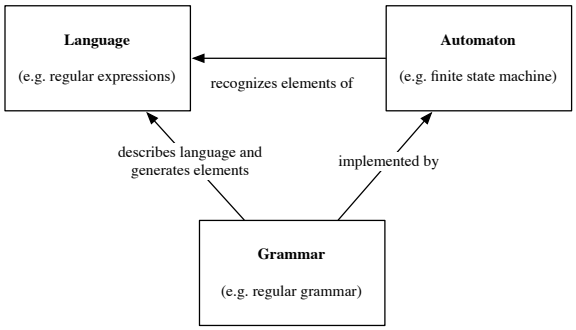


Figure 3.12: Relation between languages, grammars and automata.

The Chomsky hierarchy we have came across in Section 2.7 can now be completed, for each type of formal grammar, with the corresponding abstract machine that recognizes the elements generated by the grammar. It is summarized in Table 3.1.

Type	Grammar	Language	Automaton
0	Unrestricted $\alpha \rightarrow \beta$	Recursively Enumerable e.g. $\{a^n \mid n \in \mathbb{N}, n \text{ perfect}\}$	Turing Machine
1	Context-Sensitive $\alpha A \beta \rightarrow \alpha \gamma \beta$	Context-Sensitive e.g. $\{a^n b^n c^n \mid n \geq 0\}$	Linear Bounded Turing Machine
2	Context-Free $A \rightarrow \gamma$	Context-Free e.g. $\{a^n b^n \mid n \geq 0\}$	Push-Down Automaton
3	Regular $A \rightarrow \epsilon \mid a \mid aB$	Regular e.g. $\{a^m b^n \mid m, n \geq 0\}$	Finite State Automaton

Table 3.1: The Chomsky hierarchy.

3.7 Chapter Summary

- *Computation* is formally described as a "yes/no" decision problem.
- *Finite State Automata, Push-Down Automata and Turing Machines* are abstract machines that can recognize elements of regular, context-free and recursively enumerable languages, respectively.
- A *State Diagram* is a convenient way of graphically representing a finite state automaton.
- *Parsers* are concrete implementations of push-down automata, which not only recognize, but also evaluate elements of a language (such as mathematical expressions, or the code of a program in C).
- *Compiler Compilers* are powerful tools that can be used to automatically generate parts of your program. They only require a definition of the "what" (a grammar with actions) and take care of all details necessary to generate the "how" (some functional code that can correctly parse input strings).
- *Computation* can also be defined as all what a Turing machine can compute. There are however some well-known "yes/no" decision problems that cannot be computed by any Turing machine.

Chapter 4

Markov Processes

4.1 Why Study Markov Processes?

As we'll see in this chapter, Markov processes are interesting in more than one respects. On the one hand, they appear as a natural extension of the finite state automata we've discussed in Chapter 3. They constitute an important theoretical concept that is encountered in many different fields. We believe therefore that it is useful for anyone (being in academia, research or industry) to have heard about the terminology of Markov processes and to be able to talk about it.

On the other hand, the study of Markov processes – more precisely *hidden* Markov processes – will lead us to algorithms that find direct application in today's technology (such as in optical character recognition or speech-to-text systems), and which constitutes an essential component within the underlying architecture of several modern devices (such as cell phones).

4.2 Markov Processes

A Markov process[1] is a stochastic extension of a finite state automaton. In a Markov process, state transitions are probabilistic, and there is – in contrast to a finite state automaton – no input to the system. Furthermore, the system is only in one state at each time step. (The nondeterminism of finite state automata should thus not be confused with the stochasticity of Markov processes.)

[1] Named after the Russian mathematician Andrey Markov (1856-1922).

Before coming to the formal definitions, let us introduce the following example, which should clearly illustrate what a Markov process is.

Example. Cheezit[2], a lazy hamster, only knows three places in its cage: (a) the pine wood shaving that offers him a bedding where it sleeps, (b) the feeding trough that supplies him with food, and (c) the wheel where it makes some exercise.

After every minute, the hamster either gets to some other activity, or keeps on doing what he's just been doing. Referring to Cheezit as a process without memory is not exaggerated at all:

- When the hamster sleeps, there are 9 chances out of 10 that it won't wake up the next minute.
- When it wakes up, there is 1 chance out of 2 that it eats and 1 chance out of 2 that it does some exercise.
- The hamster's meal only lasts for one minute, after which it does something else.
- After eating, there are 3 chances out of 10 that the hamster goes into its wheel, but most notably, there are 7 chances out of 10 that it goes back to sleep.
- Running in the wheel is tiring: there is an 80% chance that the hamster gets tired and goes back to sleep. Otherwise, it keeps running, ignoring fatigue.

4.2.1 Process Diagrams

Process diagramas offer a natural way of graphically representing Markov processes – similar to the state diagrams of finite automata (see Section 3.3.2).

For instance, the previous example with our hamster in a cage can be represented with the process diagram shown in Figure 4.1.

[2]This example is inspired by the article found on `http://fr.wikipedia.org/wiki/Chaîne_de_Markov`. The name was generated by a "Computer-Assisted Hamster Naming" technology found on `http://www.coyoteslodge.com/hamform.htm`.

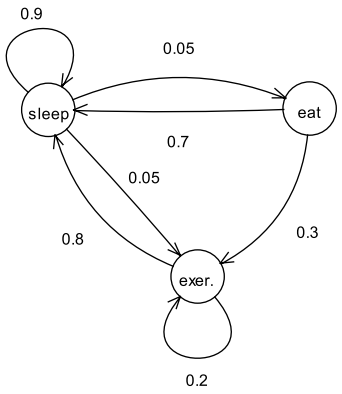


Figure 4.1: Process diagram of a Markov process.

4.2.2 Formal Definitions

Definition 4.1. A *Markov chain* is a sequence of random variables $X_1, X_2, X_3, \ldots$ with the *Markov property*, namely that the probability of any given state X_n only depends on its immediate previous state X_{n-1}. Formally:

$$P(X_n = x \mid X_{n-1} = x_{n-1}, \ldots, X_1 = x_1) = P(X_n = x \mid X_{n-1} = x_{n-1})$$

where $P(A \mid B)$ is the probability of A given B.

The possible values of X_i form a countable set S called the *state space* of the chain. If the state space is finite, and the Markov chain time-homogeneous (i.e. the transition probabilities are constant in time), the transition probability distribution can be represented by a matrix $\mathbf{P} = (p_{ij})_{i,j \in S}$, called the *transition matrix*, whose elements are defined as:

$$p_{ij} = P(X_n = j \mid X_{n-1} = i)$$

Let $\mathbf{x}^{(n)}$ be the *probability distribution* at time step n, i.e. a vector whose i-th component describe the probability of the system to be in state i at time state n:

$$\mathbf{x}_i^{(n)} = P(X_n = i)$$

Transition probabilities can be then computed as power of the transition matrix:

$$\begin{aligned} \mathbf{x}^{(n+1)} &= \mathbf{P} \cdot \mathbf{x}^{(n)} \\ \mathbf{x}^{(n+2)} &= \mathbf{P} \cdot \mathbf{x}^{(n+1)} = \mathbf{P}^2 \cdot \mathbf{x}^{(n)} \\ \mathbf{x}^{(n)} &= \mathbf{P}^n \cdot \mathbf{x}^{(0)} \end{aligned}$$

Example. The state space of the "hamster in a cage" Markov process is:

$$S = \{\text{sleep}, \text{eat}, \text{exercise}\}$$

and the transition matrix:

$$\mathbf{P} = \begin{pmatrix} 0.9 & 0.7 & 0.8 \\ 0.05 & 0 & 0 \\ 0.05 & 0.3 & 0.2 \end{pmatrix}$$

The transition matrix can be used to predict the probability distribution $\mathbf{x}^{(n)}$ at each time step n. For instance, let us assume that Cheezit is initially sleeping:

$$\mathbf{x}^{(0)} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

After one minute, we can predict:

$$\mathbf{x}^{(1)} = \mathbf{P} \cdot \mathbf{x}^{(0)} = \begin{pmatrix} 0.9 \\ 0.05 \\ 0.05 \end{pmatrix}$$

Thus, after one minute, there is a 90% chance that the hamster is still sleeping, 5% chance that he's eating and 5% that he's running in the wheel.

Similarly, we can predict that after two minutes:

$$\mathbf{x}^{(2)} = \mathbf{P} \cdot \mathbf{x}^{(1)} = \begin{pmatrix} 0.885 \\ 0.045 \\ 0.07 \end{pmatrix}$$

Definition 4.2. The *process diagram* of a Markov chain is a directed graph describing the Markov process. Each node represent a state from the state space. The edges are labeled by the probabilities of going from one state to the other states. Edges with zero transition probability are usually discarded.

4.2.3 Stationary Distribution

The theory shows that – in most practical cases[3] – after a certain time, the probability distribution does not depend on the initial probability distribution $\mathbf{x}^{(0)}$ anymore. In other words, the probability distribution converges towards a *stationary distribution*:

$$\mathbf{x}^* = \lim_{n \to \infty} \mathbf{x}^{(n)}$$

In particular, the stationary distribution $\mathbf{x}^*$ satisfies the following equation:

$$\mathbf{x}^* = \mathbf{P} \cdot \mathbf{x}^* \tag{4.1}$$

Example. The stationary distribution of the hamster

$$\mathbf{x}^* = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$$

can be obtained using Equation 4.1, as well as the fact that the probabilities add up to $x_1 + x_2 + x_3 = 1$. We obtain:

$$\mathbf{x}^* = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} x_1 \\ x_2 \\ 1 - x_1 - x_2 \end{pmatrix} = \begin{pmatrix} 0.9 & 0.7 & 0.8 \\ 0.05 & 0 & 0 \\ 0.05 & 0.3 & 0.2 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ 1 - x_1 - x_2 \end{pmatrix}$$

From the first two components, we get:

$$\begin{aligned} x_1 &= 0.9x_1 + 0.7x_2 + 0.8(1 - x_1 - x_2) \\ x_2 &= 0.05x_1 \end{aligned}$$

Combining the two equations gives:

$$0.905x_1 = 0.8$$

[3] The Markov chain must be aperiodic and irreductible.

so that:

$$\begin{aligned} x_1 &= \frac{0.8}{0.905} \approx 0.89 \\ x_2 &= 0.05x_1 \approx 0.044 \\ x_3 &= 1 - x_1 - x_2 \approx 0.072 \\ \mathbf{x}^* &\approx \begin{pmatrix} 0.89 \\ 0.044 \\ 0.072 \end{pmatrix} \end{aligned}$$

In other words, if we observe the hamster long enough, the probability that it will be sleeping is $x_1 = 89\%$, that it will be eating $x_2 = 4\%$, and that it will be doing some exercise $x_3 = 7\%$.

4.3 Hidden Markov Models

A *hidden Markov model* (HMM) is a statistical model in which the system being modeled is assumed to be a Markov process with unknown parameters. The challenge is to determine the hidden parameters from the observable parameters. Typically, the parameters of the model are given and the challenge is to find the most likely sequence of hidden states that could have generated a given sequence of observed states.

In a regular Markov model, the state is directly visible to the observer, and therefore the state transition probabilities are the only parameters. In a hidden Markov model, the state is not directly visible. Rather, the observer sees an observable (or output) token. Each hidden state has a probability distribution, called *emission probability*, over the possible observable tokens. Figure 4.2 illustrates a hidden Markov model.

Example 4.1. Assume you have a friend who lives far away and to whom you talk daily over the telephone about what he did that day. Your friend is only interested in three activities: walking in the park, shopping, and cleaning his apartment. The choice of what to do is determined exclusively by the weather on a given day. You have no definite information about the weather where your friend lives, but you know general trends. Based on what he tells you he did each day, you try to guess what the weather must have been like.

You believe that the weather operates as a discrete Markov process (i.e. a Markov chain). There are two states, "Rainy" and "Sunny", but you cannot observe them directly – they are hidden from you. On each day, there is a certain

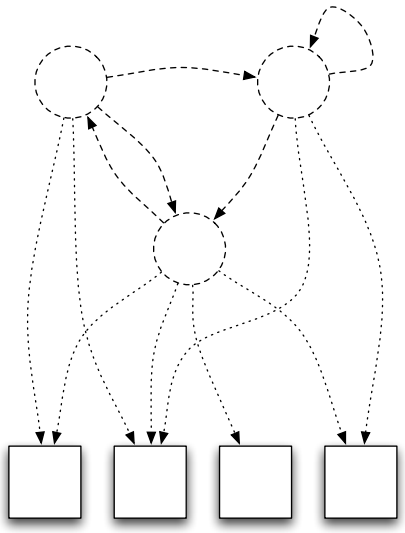


Figure 4.2: Illustration of a hidden Markov model. The upper part (dashed) represents the underlying hidden Markov process. The lower part (shadowed) represents the observable outputs. Dotted arrows represent the emission probabilities.

chance that your friend will perform one of the following activities, depending on the weather: “walk”, “shop’, or “clean”. Since your friend tells you about his activities, those are the observations.

Furthermore, you know the general weather trends in the area, and what your friend likes to do on average.

Concerning the weather, you known that:

- If one day is rainy, there is a 70% chance that the next day will be rainy too.
- If one day is sunny, there is 40% chance that the weather will degrade on the next day.

Your friend’s general habits can be summarized as follows:

- If it is rainy, there is a 50% chance that he is cleaning his apartment, and only 10% chance that he goes out for a walk.
- If it is sunny, there is a 60% chance that he is outside for a walk, 30% chance that he decides to go shopping and 10% that he stays home to clean his apartment.

The entire system is that of a hidden Markov model (HMM). The hidden state space is $S = \{\text{Rainy}, \text{Sunny}\}$, and the possible observable output state $O = \{\text{walk}, \text{shop}, \text{clean}\}$. The transition probability matrix (between hidden states) is:

$$\mathbf{P} = \begin{pmatrix} 0.7 & 0.4 \\ 0.3 & 0.6 \end{pmatrix}$$

Finally, the emission probabilities are:

	Rainy	Sunny
walk	0.1	0.6
shop	0.4	0.3
clean	0.5	0.1

The whole model can also be summarized with the following diagram:

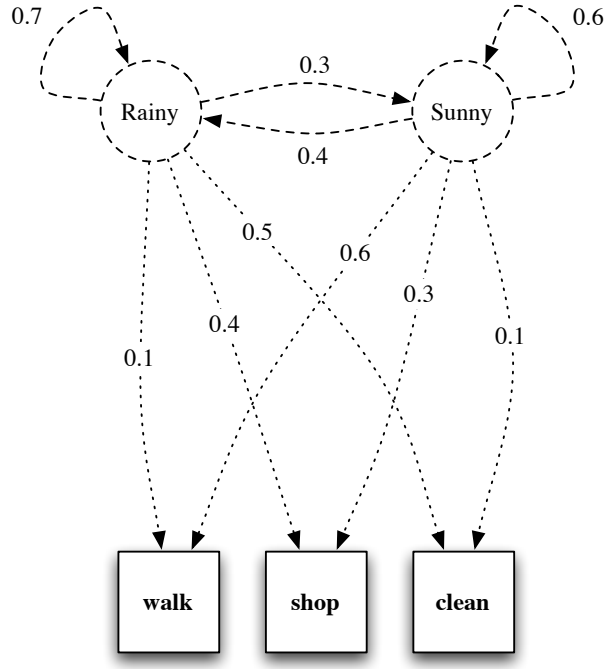


Figure 4.3: Diagram of a hidden Markov model.

4.3.1 Viterbi Algorithm

You talk to your friend three days in a row and discover that on the first day he went for a walk, on the second day he went shopping, and on the third day he cleaned his apartment.

There are many questions that can be asked from this sequence of observations, such as the overall probability of such a sequence to be observed, etc. We will however focus on the following one:

> What is the most likely sequence of rainy/sunny days that would explain these observations?

One could obviously enumerate all possible sequences of hidden states, calculate for each the probability of observing the given output sequence, and eventually pick up the most probable one.

Yet, the assumption that the underlying process (the weather) has the Markov property (i.e. the probability of each state only depends on the previous one) allows us to solve this question using a much more efficient technique: the *Viterbi algorithm*.

The algorithm

The Viterbi algorithm (also called "forward-Viterbi") works as follows. For each successive observable output and each possible hidden state, keep track of:

- the relative probability,
- and the most probable sequences of hidden states *so far* (with their corresponding probability).

The updating process from one observable output to the next includes multiplying the probabilities obtained so far with the corresponding transition probability *and* emission probability.

The algorithm is best illustrated with a concrete example. Figures 4.4 to 4.7 illustrate, for the above example (where your friend is observed to walk, then shop and finally clean his apartment), how the most probable sequence of hidden events is obtained. The most likely sequence of hidden states for the given observations turns out to be (Sunny, Rainy, Rainy).

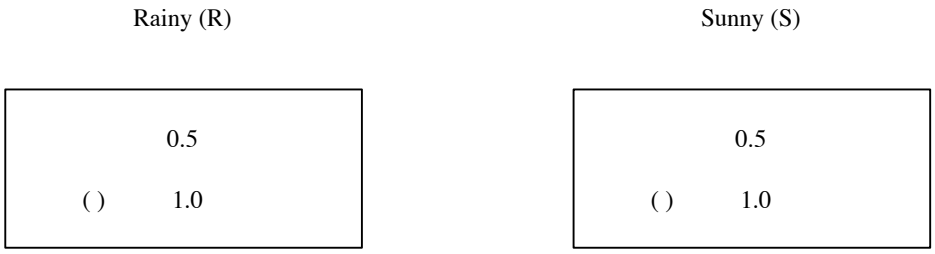


Figure 4.4: Viterbi algorithm – step 0 (initialization). All hidden states are assumed to have the same initial probability.

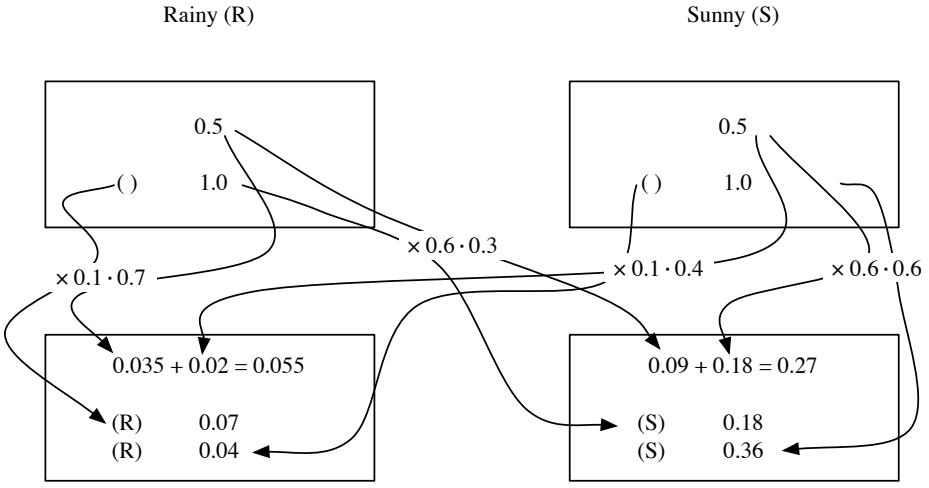


Figure 4.5: Viterbi algorithm – step 1 (observe "walk"). The new states are updated using both emission and transition probabilities (see Figure 4.3).

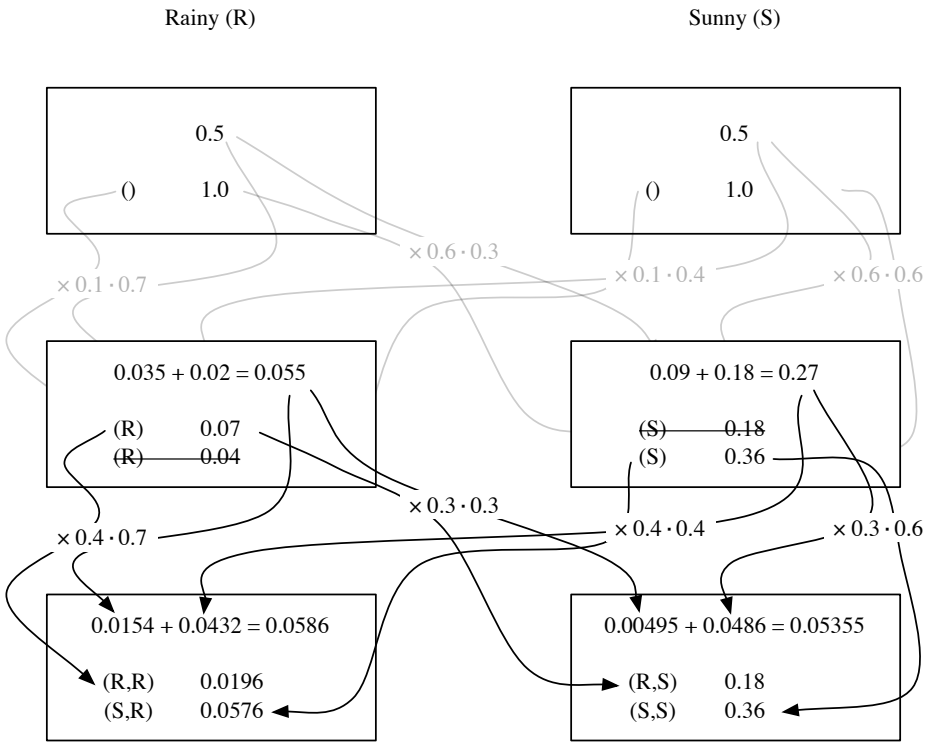


Figure 4.6: Viterbi algorithm – step 2 (observe "shop"). For each hidden state, only the most probable sequence so far is kept – all other sequences are discarded. The updating process then repeats.

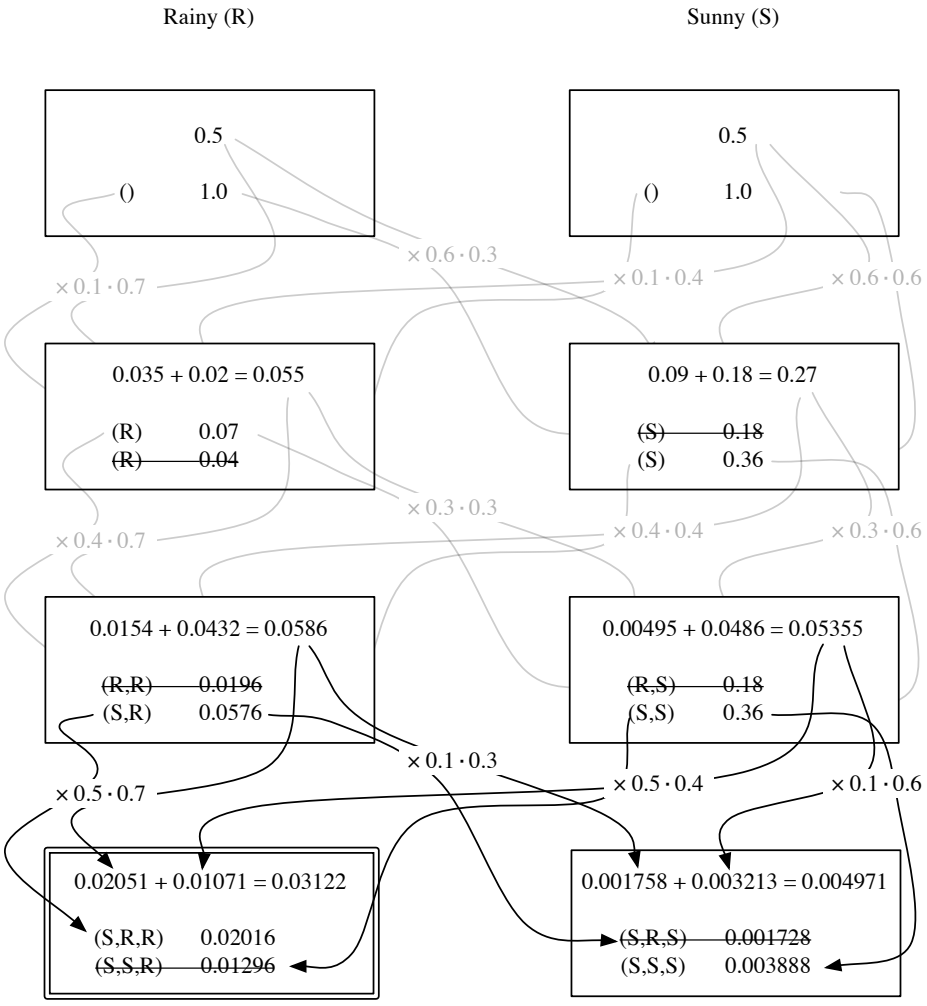


Figure 4.7: Viterbi algorithm – step 3 (observe "clean"). After the final step is processed, we only consider the most probable hidden state (here, "Rainy"). The most likely sequence of hidden states for the given observation is thus {S,R,R}, i.e. {Sunny, Rainy, Rain}.

4.3.2 Complexity of the Viterbi Algorithm

The computational complexity of the Viterbi algorithm is $\mathcal{O}(M^2 \cdot N)$, where $M = |S|$ is the number of possible hidden states, and N the number of observed outputs. In other words, the number of steps required by the Viterbi algorithm scales linearly with the length of the observed sequence.

This dramatically contrasts with a brute-foce search, which is $\mathcal{O}(M^N)$, i.e. exponential. The following Table 4.1 compares the time possibly required by the Viterbi algorithm and a brute-force algorithm, for our particular example model ($M = 2$), with observed sequences of different lengths. We assume that both algorithms take 1ms to compute the above example with $N = 3$.

	Viterbi	Brute Force
$N = 3$	1ms	1ms
$N = 10$	3.3ms	0.13s
$N = 20$	6.7ms	2.2min (!)

Table 4.1: Comparison between Viterbi and brute foce algorithms.

4.3.3 Applications

Hidden Markov models (HMM) have applications in many domains, such as temporal pattern recognition. Note however that HMM often don't offer *per se* the whole solution to a problem, but that they constitute a part of a more complete solutions, typically used to drastically speed up the processing of input sequences.

Here are some example of applications (which will be either demonstrated during the lecture or studied in some exercise sheet):

- Text recognition, optical character recognition (OCR)
  - Hidden variable: letters of the alphabet
  - Observable: simple features extracted from pixels of character maps (such as strokes and arcs)
- Predictive text input systems for portable devices such as cell phones (maybe even the iPhone?).
  - Hidden variable: letters of the alphabet

  - Observable: the sequence of keys pressed by the user

- Speech-to-Text Systems (such as IBM ViaVoice)
  - Hidden variable: phonemes, syllables of the words
  - Observable: features of frequency spectrum (such as formants, peaks, etc.)

4.4 Chapter Summary

- A *Markov process* is a stochastic extension of a finite state automaton.
- A system with the *Markov property* is a system with no memory: the transition probabilities only depend on the current state of the system.
- In a *hidden Markov model*, the underlying system is assumed to be a Markov process. The actual state of the underlying system is not directly visible. Rather, the observer sees an observable output, whose *emission probability* depends on the hidden state of the system.
- The *Viterbi algorithm* is an efficient algorithm that finds the most likely sequence of hidden states given a sequence of observations.
- *Applications* of Hidden Markov models include text recognition, predictive text input systems of portable devices and speech-to-text software.

Part II

Logic

Chapter 5

Logic

5.1 Why Study Logic?

We have seen in the previous part of this script how formal language theory studies the *syntax* of languages. Yet, this formalism is not concerned with how sentences – i.e. elements of a language – are *related* to each other. In other words, formal language theory does not answer the question of how a sentence can be *derived* from already existing sentences.

This is where *logic* comes into play. As a formal science, logic investigates how valid statements can be formally derived or inferred from other statements, or from a set of axioms. In short, logic enables us to formalize the notion of a *proof*.

Traditionally studied as a branch of philosophy, logic has become since the nineteenth century the foundation of mathematics and computer science. You may have heard of some of the key figures of mathematical logic: G. Boole, B. Bolzano, G. Frege, D. Hilbert, A. Church, G. Peano, B. Russel, A. Tarski or A. Turing.

Interestingly, the study of logic not only provides a formalism about abstract objects, but also opens the way to concrete applications. By defining how theorems – i.e. valid formulas – can be derived in a purely syntactic way, computers can then be used to derive statements about the "real" world. For instance, predicate logic is often used in expert systems to represent the knowledge and to derive new facts from the knowledge base. Or, in linguistics, declarative programming languages are often used, the most prominent one being Prolog.

5.1.1 An Experiment

Imagine that you have been asked to perform two tasks[1].

In the first, you are presented with a stack of cards. One side of each card has either the letter A or the letter D. The opposite side of each card has either the number 4 or the number 7. The cards are now stacked with either side up, at random, and shuffled, so that thumbing through the deck you would see some A's, some D's, some 4s and some 7s. Your task is to determine whether or not the cards of this deck satisfy the rule "If the letter side of a card is an A, then the number side must be a 4." To make that determination, you are to imagine that you are going through the deck, looking at the turned-up side of each card, one at a time, and turning over whichever cards you must, but only those cards, in order to verify or contradict the rule that every A must be accompanied by a 4.

Think about the task for a moment. Would you turn over only those cards with A's showing? Or those showing A's and 4s? Or showing A's and 7s? Or perhaps those showing A's, 4s and 7s? Or did you choose some other combination? Write down your choice and proceed to task two. And don't feel discouraged. Task one is difficult, and most of the English college students upon whom the experiment was first performed failed to give the correct answer.

As task two, you are to imagine that your are the cashier at a supermarket and have the checks received that day stacked before you; some face up and some face down. Your supermarket has a rule. The checkout people are to accept checks for more the $50 only if approved on the back by the manager. Imaginge that you are to go through the checks, one at a time, and turn over only those checks necessary to establish if the approval rule has been followed.

Again, think about the task for a moment. Would you turn over only checks bigger than $50? Or those, pluse checks with their face down bearing the manager's approval? Or those for over $50 and those with no approval on the back? Or perhaps checks exceeding $50, plus all checks with their faces down? Or some other combination?

As before, jot down your answer. If you are typical of most subjects of this experiment, you did not find task two nearly as difficult as task one. You probably correctly answered two by turning over checks for more than $50 and those with no approval on their backs. You were more likely to miss on task one, for which the correct solution is to turn over A's and 7s only.

Why this pair of experiments? Because the two tasks are essentially identical.

[1] This experiment is taken from (Dreyfus and Dreyfus, 1986, p.18).

If you designate "over $50" as A, "not over $50" as D, "approved on back" as 4 and "unapproved" as 7, task two becomes task one. But while they are abstractly identical, the statement of task two draws, for many, on "knowing how," whereas task one is perceived as a logical puzzle requiring the application of logical rules, that is , requiring the reduction to "knowing that." All of you who did task two easily and correctly and had trouble with task one have learned from this experience that "knowing how" is quite distinct from "knowing that" and in no way requires using conscious abstract rules.

5.2 Definition of a Formal System

Definition 5.1. In logic, a *formal system* consists of:

(a) a formal language,

(b) a set of axioms,

(c) a set of inference (or transformation) rules.

The choice of these defines the power of the system (which assertions can be expressed) and its properties (completeness, decidability, etc.).

5.3 Propositional Logic

One of the most basic types of logic (and one of the simplest formal systems) is *propositional logic* or *propositional calculus*. Propositional calculus examines the syntax and semantics of expressions which are formed by connecting atomic formulas (i.e. variables that are either true or false) by logical connectives. Propositional calculus can only make atomic statements. Thus, a sentence from natural language such as "Socrates is human" becomes `socrates_is_human`.[2] This statement can then have either the value of true or false.

5.3.1 Language

The language of propositional logic consists of:

- a set P of atomic formulas, consisting of e.g. symbols such as $p, q, r...$

[2]The statement of this archetypal example usually reads `human_socrates`.

- the following logical connectives:

  $\neg$ negation

  $\wedge$ logical "and"

  $\vee$ logical "or"

  $\rightarrow$ logical implication ("if ... then")

  $\leftrightarrow$ equivalence ("if and only if")

- auxiliary symbols: "(" and ")"

Definition 5.2. *Propositional formulas* can then be constructed from the symbols of the language by a recursive definition:

(i) Every atomic formula $p \in P$ is a propositonal formula.

(ii) If the expressions A and B are propositional formulas, then the expressions $\neg A, (A \wedge B), (A \vee B), (A \rightarrow B), (A \leftrightarrow B)$ are propositional formulas.

(iii) Every propositional formula arises from a finite number of application of (i) and (ii).

Example 5.1. If $P = \{p, q, r\}$ is a set of atomic formulas, then the expressions

- p, q and r are propositional formulas according to (i)
- $\neg p$ and $(q \wedge r)$ are propositional formulas according to (ii)
- $(\neg p \rightarrow (q \wedge r))$ is a propositional formula according to another application of (ii)

Alternatively, propositional (or well-formed) formulas can also be generated by means of a grammar:

$$
\begin{array}{rcl}
\langle \text{formula} \rangle & \rightarrow & \langle \text{atomic formula} \rangle \,|\langle \text{ propositional formula} \rangle \\
\langle \text{atomic formula} \rangle & \rightarrow & T \mid F \mid p \mid q \mid r \mid \ldots \\
\langle \text{propositional formula} \rangle & \rightarrow & (\ \langle \text{formula} \rangle\ ) \\
 & \mid & \langle \text{formula} \rangle \langle \text{connector} \rangle \langle \text{formula} \rangle \\
 & \mid & \neg \langle \text{formula} \rangle \\
\langle \text{connector} \rangle & \rightarrow & \wedge \mid \vee \mid \rightarrow \mid \leftrightarrow
\end{array}
$$

where T and F stand for the logical "true" and "false" values.

This grammar still contains some ambiguity as long as the priority of connectors is not defined. Priority is defined as follows:

1. The negation $\neg$ has the highest priority,
2. followed by the logical "and" ($\wedge$),
3. the logical "or" ($\vee$),
4. and the logical implications ($\rightarrow$, $\leftrightarrow$).

Alternatively, parenthesis can be used, which also makes a logical expression more readable. For instance, $A \wedge \neg B \vee C \rightarrow D$ can also be written as $((A \wedge (\neg B)) \vee C) \rightarrow D$.

Now we are able to construct propositional (or well-formed) formulas.

5.3.2 Semantics

The semantics of propositional logic is concerned with the truth or falsity of propositional formulas. The atomic formulas cannot be analysed further, their truth value is given, by means of a truth function. This process, assigning a value of true or false to every atomic formula, is called interpretation. The truth values of the atomic formulas can then be unambigously extended to all propositional formulas by defining the semantics of logical connectives. This can be done by means of a truth table:

P	Q	$\neg$P	$P \vee Q$	$P \wedge Q$	$P \rightarrow Q$	$P \leftrightarrow Q$
T	T	F	T	T	T	T
T	F	F	T	F	F	F
F	T	T	T	F	T	F
F	F	T	F	F	T	T

Note that the logical implication $P \rightarrow Q$ is simply defined by the above truth table. It does not contain any notion of causality relating events in the real world.

Definition 5.3. A propositional formula A is *satisfiable*, if there exists an interpretation of its atomic formulas (assigning truth values to all of them) such that A is true.

Definition 5.4. Let M be a set of formulas. If a fomula A is true in every interpretation in which all formulas of M are true, then A is a *tautologic consequence* of M and we write $M \vDash A$.

Definition 5.5. If A is true in all interpretations, it is called a *tautology*. This is written as $\vDash A$. In other words, $\vDash A$ implies that $M \vDash A$ is valid for any M. An example of a tautology is $P \vee \neg P$.

Definition 5.6. *Contradiction*: If a statement is always false for all interpretations, it is called a contradiction. An example of contradiction is $P \wedge \neg P$.

5.3.3 Formal System

We have already defined the language and propositional formulas. To complete the formal system of propositional logic we need a set of axioms and inference rules.

Why would we need a formal system? We are already able to construct well-formed formulas and decide on their truthfulness by means of a truth table. However, imagine we had a set of formulas M and we know that they are true – they represent our knowledge about a certain problem. We would then be interested in other formulas valid in this situation, i.e. in some A, such that $M \vDash A$. How would you find them? By means of a truth table, we would have to list all interpretations for which M is true and then randomly generate various formulas and check whether they are true in those interpretations. In complex situations, this would be a tedious job!

On the other hand, a formal system would allow to generate valid formulas in an automated and more effective manner. You can think of the formal system as syntax, as a complement of semantics.

Axioms

An important requirement we have on any formal system is that only valid (i.e. logically true) formulas can be derived. Such a system is then said to be *sound*. The basis of the formal system are the axioms. A logical choice thus is to choose axioms from tautologies – formulas valid in every interpretation.

A particularly compact and well-known axiom system for propositional logic is the following (after Jan Lukasiewicz):

$$
\begin{array}{rcll}
p & \rightarrow & (q \rightarrow p) & \text{(A1)} \\
(p \rightarrow (q \rightarrow r)) & \rightarrow & ((p \rightarrow q) \rightarrow (p \rightarrow r)) & \text{(A2)} \\
(\neg p \rightarrow \neg q) & \rightarrow & (q \rightarrow p) & \text{(A3)}
\end{array}
$$

Note that other axiom systems are also possible.

Inference rules

What requirements do we have on the rules of inference? They should be *correct*, i.e. from a formula that is valid in an interpretation, they can derive only a formula that is valid in the same interpretation. Propositional logic has a single inference rule: Modus ponens.

Modus Ponens

$$
\begin{array}{l}
p \rightarrow q \\
\underline{p} \\
q
\end{array}
$$

Example 5.2. Let us replace p with `bad_weather` and q with `I_stay_home`. If we assume the axioms `bad_weather` $\rightarrow$ `I_stay_home` as well as `bad_weather`, we can derive, using modus ponens, the formula `I_stay_home`:

$$
\begin{array}{l}
\texttt{bad\_weather} \rightarrow \texttt{I\_stay\_home} \\
\underline{\texttt{bad\_weather}} \\
\texttt{I\_stay\_home}
\end{array}
$$

All theorems of propositional logic can be derived from the three axioms and the single inference rule.

Proofs

Definition 5.7.

(i) A finite sequence of formulas $A_1, A_2, \ldots, A_n$ is the *proof* of a formula A, if A_n is the formula A and for any i, formula A_i is either an axiom or it is derived from previous formulas A_j $(j < i)$ by modus ponens.

(ii) If there exists a proof of formula A, we say that A is provable in propositional logic and we write $\vdash A$.

Example 5.3. To get an idea, how the formal system works, let us look at the formal proof of a seemingly trivial formula $A \to A$.

$\vdash A \to ((A \to A) \to A)$	a case of (A1) axiom
$\vdash (A \to ((A \to A) \to A)) \to$ $[(A \to (A \to A)) \to (A \to A)]$	a case of (A2) axiom
$\vdash (A \to (A \to A)) \to (A \to A)$	modus ponens
$\vdash (A \to (A \to A))$	a case of (A1) axiom
$\vdash A \to A$	modus ponens

Although this seems that it is uselessly complicated and you could get this trivially just by looking at it or from a truth table, there are also cases when the reverse is true.

Definition 5.8. *Proof from assumptions.* Let T be a set of formulas. We say that formula A is provable from T and write $T \vdash A$ if A can be proven from axioms and formulas in T, by using the inference rule. That is we have basically enriched the set of axioms by the formulas in T.

Theorems

The following are some theorems of propositional logic. They can be derived from axioms or you can check that they are tautologies by examining their truth tables.

de Morgan's Rules

1. $(\neg(P \vee Q)) \leftrightarrow (\neg P) \wedge (\neg Q)$

2. $(\neg(P \wedge Q)) \leftrightarrow (\neg P) \vee (\neg Q)$

Distributive Laws

1. $(P \vee (Q \wedge R)) \leftrightarrow ((P \vee Q) \wedge (P \vee R))$

2. $(P \wedge (Q \vee R)) \leftrightarrow ((P \wedge Q) \vee (P \wedge R))$

5.3.4 Completeness

Theorem 5.1 (Post)**.** *For every propositional formula A*

$$\vdash A \quad \textit{if and only if} \quad \models A$$

This means that, in propositional logic, tautologies are provable, and what is provable is a tautology. Thus, the formal system of propositional logic is not only sound (i.e. generates only valid formulas) but also generates all of them.

Theorem 5.2 (completeness of propositional logic)**.** *Let T be a set of formulas and A a formula. Then*

$$T \vdash A \quad \textit{if and only if} \quad T \models A$$

This is a more general version of Post's theorem. In a sense, completeness implies that loosely speaking syntax and semantics are equivalent in this case. This is by no means true for any formalism (see below).

5.3.5 Normal Forms of Propositional Formulas

Every propositional formula can be expressed in two standard or *normal* forms.

Definition 5.9 (Conjunctive normal form – CNF)**.** A formula in CNF has the following form:

$$(A_1 \vee \ldots \vee A_N) \wedge (B_1 \vee \ldots \vee B_M) \wedge \ldots$$

where the A_i can be either atomic formulas or their negations. It is thus a conjuction of clauses, i.e. disjunctions of literals (variables or their negations).

CNF is used in machine proving of theorems. Resolution in Prolog is also based on a special form of CNF.

Definition 5.10 (Disjunctive normal form – DNF)**.** A formula in DNF has the following form:

$$(A_1 \wedge \ldots \wedge A_N) \vee (B_1 \wedge \ldots \wedge B_M) \vee \ldots$$

where A_1 etc. can be either atomic formulas or their negations. It is thus a disjuction of cojunctions of literals.

Applications of DNF include databases.

Example 5.4. This example shows how the following formula can be transformed into conjunctive normal form.

$$A \rightarrow \neg(B \rightarrow C)$$

1st step: Elimination of $\rightarrow$. Using the rule $P \rightarrow Q = \neg P \vee Q$:

$$(A \rightarrow \neg(B \rightarrow C)) = (\neg A \vee \neg(\neg B \vee C))$$

2nd step: Distribution of $\neg$ onto atomic expressions.

$$\begin{array}{lll} & (\neg A \vee \neg(\neg B \vee C)) & \\ = & (\neg A \vee (\neg\neg B \wedge \neg C)) & \text{using rule } \neg(P \vee Q) = \neg P \wedge \neg Q \\ = & (\neg A \vee (\neg\neg B \wedge \neg C)) & \text{using rule } \neg\neg P = P \\ = & (\neg A \vee (B \wedge \neg C)) & \end{array}$$

3rd step: Transforming into a conjuction of disjunctions by the distributive rule. Using rule $P \vee (Q \wedge R) = (P \vee Q) \wedge (P \vee R)$:

$$\begin{array}{ll} & (\neg A \vee (B \wedge \neg C)) \\ = & ((\neg A \vee B) \wedge (\neg A \vee \neg C)) \end{array}$$

5.3.6 A "Logical" Anecdote

This example shows the power of propositional calculus by illustrating how it allows us to easily solve a problem which doesn't offer an intuitive solution. The problem reads as follows:

1. If the weather is bad, I stay at home.
2. I die if and only if my house explodes because of a gas leak and I'm at home.
3. If my house explodes or I go out, and only then, my neighbors notice something strange.
4. Whenever my neighbors notice something strange, they call home the next day (they only do so if I'm alive). They never call home otherwise.

My wife (who works at home) tells you that she received a call from the neighbors last Tuesday. Can you say something about the weather of last Monday?

Simplification

Let's rewrite the above statements as logical expressions by setting:

- a: `bad_weather`
- b: `stay_home`
- c: `I_die`
- d: `gas_leak_explosion`
- e: `neighbors_notice_something_strange`
- f: `neighbors_call_home`

This gives:

1. $a \rightarrow b$
2. $c \leftrightarrow d \wedge b$
3. $d \vee \neg b \leftrightarrow e$
4. $e \wedge \neg c \leftrightarrow f$

From the last statement of the problem, we can derive:

$$\begin{aligned} f &\leftrightarrow e \wedge \neg c \\ &\leftrightarrow (d \vee \neg b) \wedge \neg(d \wedge b) \\ &\leftrightarrow (d \vee \neg b) \wedge (\neg d \vee \neg b) \\ &\leftrightarrow (d \wedge \neg d) \vee \neg b \\ &\leftrightarrow \neg b \\ &\rightarrow \neg a \end{aligned}$$

Therefore: you know that on Monday, the weather was good! □

5.4 Predicate Calculus (First Order Logic)

Propositional calculus has a limited expressive power. It works with atomic statements that are either true or false and studies the properties of logical connectives that connect the atomic formulas.

Predicate calculus introduces variables that do not have a value of true or false, but can take up any value, depending on the model, such as an element of a set (e.g. of the set of all inhabitants of Prague), or a natural number. There are also functions operating on the variables and predicates that have a truth value. By means of predicates, formulas can be constructed. In addition, the language also contains *quantifiers*. Such a language is called *first order language*.

5.4.1 Language

Definition 5.11. *First order language* contains:

(i) An unlimited number of symbols for variables: $x, y, z, \ldots$

(ii) Symbols for logical connectives: $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$.

(iii) Symbols for quantifiers: the *universal* quantifier $\forall$ ("for all"), and the *existential* quantifier $\exists$ ("there exists").

(iv) Symbols for predicates: $p, q, ...$

Predicates are relations. The *arity* of a predicate symbol specifies the number of arguments of the predicate. For example, equality "=" is a binary predicate (its arity is 2).

(v) Symbols for functions: $f, g, \ldots$

The *arity* of a function symbol specifies the number of arguments of the function. Function symbols of arity 0 are constants.

(vi) Auxiliary symbols: "(", ")"

This language is called first order because one can only quantify over variables (for individuals), such as in $(\forall P)\texttt{inhabitant\_of\_Prague}(P) \rightarrow \texttt{mortal}(P)$. However, one cannot quantify over predicates, i.e. one cannot say "$\forall$ inhabitants of Prague".

Symbols for variables, connectives, quantifiers and equality (if present) are called *logical symbols* because they are present in every first order language. On

the other hand, symbols for predicates and functions are called special symbols and the choice of these symbols depends on what we want to study, i.e. on the particular laguage.

Examples of languages

a) Language of predicate logic. This simply is a first order language without any special symbols.

b) Language of group theory. This a first order language with equality with two special symbols: e, a constant for the unit element, and a binary function symbol '$\cdot$' for the group operation.

c) Language of set theory is a language with equality and a single special symbol $\in$ as a binary predicate symbol for belonging to a set.

d) Language of elementary number theory[3] (with equality) contains function symbols 0 (constant for zero), S (unary symbol for the consecutive natural number, e.g.), $+$ and $\times$ (binary symbols for addition and multiplication).

Example 5.5 (Colonel West)**.** The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is American.

What we wish to prove is that West is a criminal. We can represent these facts in first-order logic, and then show the proof as a sequence of applications of the inference rules.

Yet, expecting the proof to be found by a computer requires a pretty smart program. The reasons not only include the problem of formalizing common-sense knowledge (e.g. that a missile is a weapon, that an enemy of America counts as a "hostile", etc.), but also a large branching factor, and hence a potentially explosive search problem. Thus, even a simple problem for a human to solve can lead to serious difficulty when needed to be formally proven.

[3]Number theory is the branch of pure mathematics concerned with the properties of numbers in general, and integers in particular, as well as the wider classes of problems that arise from their study. Some also refers to it as "arithmetic".

Knowledge base (or axis of a special theory – the world of Colone West)

"it is a crime for an American to sell weapons to hostile nations"

$$(\forall X)\ (\forall Y)\ (\forall Z) \quad (\ \text{american}(X) \wedge \text{weapon}(Y) \wedge \text{nation}(Z) \wedge \text{hostile}(Z) \wedge \text{sells}(X, Z, Y) \rightarrow \text{criminal}(X)\ ) \tag{5.1}$$

"the country Nono"

$$\text{nation}(\text{Nono}) \tag{5.2}$$

"Nono, an enemy of America"

$$\text{enemy}(\text{Nono}, \text{America}) \tag{5.3}$$

"Nono has some missiles"

$$(\exists X)(\text{own}(\text{Nono}, X) \wedge \text{missile}(X)) \tag{5.4}$$

"All its missiles were sold to it by Colonel West"

$$(\forall X)(\text{own}(\text{Nono}, X) \wedge \text{missile}(X) \rightarrow \text{sells}(\text{West}, \text{Nono}, X)) \tag{5.5}$$

"West, who is American"

$$\text{american}(\text{West}) \tag{5.6}$$

Frame problem (i.e. common-sense knowledge):

$$\text{nation}(\text{America}) \tag{5.7}$$

$$(\forall X) \quad (\text{missile}(X) \rightarrow \text{weapon}(X)) \tag{5.8}$$

$$(\forall X) \quad (\text{enemy}(X, \text{America}) \rightarrow \text{hostile}(X)) \tag{5.9}$$

Goal (or theorem to be proved)

$$\text{criminal}(\text{West})$$

Proof

Using 5.4 and existential elimination[4]:

$$\text{own}(\text{Nono}, \text{M1}) \wedge \text{missile}(\text{M1}) \tag{5.10}$$

Using 5.10 and "and" elimination[5]:

$$\text{own}(\text{Nono}, \text{M1}) \tag{5.11}$$

and

$$\text{missile}(\text{M1}) \tag{5.12}$$

Using 5.8 and universal elimination[6]:

$$\text{missile}(\text{M1}) \rightarrow \text{weapon}(\text{M1}) \tag{5.13}$$

Using 5.12 and 5.13 and modus ponens:

$$\text{weapon}(\text{M1}) \tag{5.14}$$

Using 5.5 and universal elimination:

$$\text{own}(\text{Nono}, \text{M1}) \wedge \text{missile}(\text{M1}) \rightarrow \text{sells}(\text{West}, \text{Nono}, \text{M1}) \tag{5.15}$$

Using 5.10 and 5.15 and modus ponens:

$$\text{sells}(\text{West}, \text{Nono}, \text{M1}) \tag{5.16}$$

Using 5.1 and universal elimination (3 times):

$$\begin{aligned} &\text{american}(\text{West}) \wedge \text{weapon}(\text{M1}) \wedge \text{nation}(\text{Nono}) \wedge \text{hostile}(\text{Nono}) \\ &\wedge \text{sells}(\text{West}, \text{Nono}, \text{M1}) \rightarrow \text{criminal}(\text{West}) \end{aligned} \tag{5.17}$$

[4]The existential elimination is a rule where the $\exists$ quantifier can be instantiated with a particular variable that does not appear elsewhere, for instance:

$$(\exists X)(\text{likes}(X, \text{IceCream})) \Rightarrow \text{likes}(\text{Person1}, \text{IceCream})$$

[5]The "and" elimination is another rule where from $X \wedge Y$ follows X (and Y).

[6]The universal elimination is a rule where the $\forall$ quantifier can be instantiated with any variable:

$$(\forall X)(\text{human}(X) \rightarrow \text{mortal}(X)) \Rightarrow \text{human}(\text{Socrates}) \rightarrow \text{mortal}(\text{Socrates})$$

Using 5.9 and universal elimination:

$$\text{enemy}(\text{Nono}, \text{America}) \rightarrow \text{hostile}(\text{Nono}) \tag{5.18}$$

Using 5.3 and 5.18 and modus ponens:

$$\text{hostile}(\text{Nono}) \tag{5.19}$$

Using 5.6, 5.2, 5.14, 5.16, 5.19 and "and" introduction[7]:

$$\begin{aligned} &\text{american}(\text{West}) \wedge \text{weapon}(\text{M1}) \wedge \text{nation}(\text{Nono}) \\ &\wedge \text{hostile}(\text{Nono}) \wedge \text{sells}(\text{West}, \text{Nono}, \text{M1}) \end{aligned} \tag{5.20}$$

Using 5.17 and 5.20 and modus ponens:

$$\text{criminal}(\text{West}) \tag{5.21}$$

□

Definition 5.12 (Expression)**.** An *expression* is any sequence of symbols of a particular language.

Definition 5.13 (Term)**.** An *term* is an expression defined recursively as follows:

(i) Every variable is a term.

(ii) If the expressions $t_1, \ldots, t_n$ are terms and f is an n-ary function symbol, then $f(t_1, \ldots, t_n)$ is a term.

(iii) Every term arises from a finite number of applications of (i) and (ii).

Example 5.6. In number theory $x, x + y$, are terms. The latter term could also be written as $+(x, y)$, where + is our f (but it is conventional to write these binary predicates in infix notation).

Definition 5.14 (Formula)**.** A *formula* is defined recursively as follows:

(i) If p is an n-ary predicate symbol and the expressions $t_1, \ldots, t_n$ are terms, then the expression $p(t_1, \ldots, t_n)$ is an atomic formula.

(ii) If the expressions A and B are formulas, then the expressions $\neg A, (A \wedge B), (A \vee B), (A \rightarrow B), (A \leftrightarrow B)$ are formulas.

[7]The "and" introduction is a rule where from X and Y follows $X \wedge Y$.

(iii) If x is a variable and A a formula, then $(\forall x)A, (\exists x)A$ are formulas.

(iv) Every formula arises from a finite number of applications of (i) to (iii).

Definition 5.15 (Theorem)**.** A *theorem* is a formula that is valid, i.e. a formula that is logically true in the given formal system.

5.4.2 Semantics

Now that we have learned the basics of syntax of predicate logic, we can have a look at the semantics. This is brought about by a relational structure $\mathfrak{M}$, which realizes (or instantiates) the symbols of our language. Moreover $\mathfrak{M}$ tells us which formulas are valid. To start with, we have to provide some values to our variables. The range of the values of our variables will be a nonempty set M, called universe of discourse $\mathfrak{M}$, and its members are individuals. On this universe of discourse, the function and predicate symbols are also realized on this universe of discourse.

Example 5.7. The realization of the language of number theory (arithmetics, see previous section) can be as follows: the universe of discourse is ω (set of all natural numbers), constant 0 is realized by an empty set $\emptyset$, the successor function is realized by a function that assigns the successive natural number to every number $n \in \omega$, and the function symbols $+$ and $\cdot$ are realized by conventional addition and multiplication.

Similarly to propositional calculus, we can investigate whether a certain formula is satisfiable or whether it is valid in every interpretation. However, in predicate logic, things get a bit more complicated. First, a relational structure $\mathfrak{M}$ realizing the language has to be chosen. This specifies how the function and predicate symbols are realized and also gives the universe of discourse M, from which we can choose the values for our variables. Once we have chosen $\mathfrak{M}$, we can assign various values to our variables – an interpretation of variables in predicate logic. An analogy to satisfiability in propositional logic would be to find an interpretation of the variables for which a formula is true.

For instance, suppose we have a standard realization (also called *model*) of number theory and the formula $x > y$. Obviously, we can find values for x and y such that the formula is true.

A stronger assertion is that a formula is *valid* in a realization $\mathfrak{M}$. That means that it is valid for every interpretation. This is analogous to a tautology. However, in predicate logic, it is with respect to a chosen realization. Obviously the formula

from the previous example is not valid – we can easily find values for x and y such that it is not true.

Suppose we had a formula $(\forall x)(\forall y)x > y$. In this case, whenever we find one interpretation giving a value of true, we automatically know that it is valid. This is because all free variables in the formula are universally quantified – we have to check all possible interpretations.

Scope of a Quantifier

The definition of the scope of a quantifier is illustrated in the following example.

Example 5.8. For every human x there exists a human y that loves x. Stated formally:

$$\underbrace{\forall x, (\text{human}(x) \rightarrow \exists y \underbrace{(\text{human}(y) \wedge \text{loves}(x, y))}_{\text{scope of } y})}_{\text{scope of } x}$$

Definition 5.16.

(i) A given occurrence of a variable x in a formula A is *bounded*, if it is part of a subformula of A (i.e. a substring of A that is also a formula) of the form $(\exists x)B$ or $(\forall x)B$. If an occurrence is not bounded, it is *free*.

(ii) A variable is *free* in A, if it has a free occurrence there.

A variable is *bounded* in A, if it has a bounded occurence there.

(iii) Formula A is *open*, if it does not contain any bounded variable.

Formula A is *closed*, if it does not contain any free variable.

Example 5.9. Formula A:

$$(\forall x)(x \rightarrow y)$$

In formula A, x has a bounded occurrence by the quantifier $\forall$, and hence it is bounded in A, whereas y is not quantified and hence it has a free occurrence and thus is free in A. Formula A is neither open nor closed.

Example 5.10. Formula B:

$$(\forall x)(\forall y)(x \rightarrow y)$$

In formula B both are variables are bounded and hence this is a closed formula.

5.4.3 Formal system

For the definition of the formal system, we will use a reduced form of the language – with logical connectives $\neg$ and $\rightarrow$ only and with only a universal quantifier $\forall$. You should be able to work out, why we can do this with the connectives. In case of the quantifiers, we use the fact that for a formula A, $(\exists x)A$ is equivalent to $\neg((\forall x)\neg A)$. The following is a formal system of predicate logic without equality.

1a) Axioms for logical connectives

(A1) – (A3) from propositional calculus

Thus, the whole propositional logic becomes a 'subset' of predicate logic. Tautologies of propositional calculus are automatically theorems of predicate calculus.

1b) Inference rule: Modus ponens

2) Axioms for quantifiers

2a) Specification scheme: Let A be a formula, x a variable and t a term that can be substituted for x into A

$$(\forall x)A \rightarrow A_x[t]$$

2b) "Jump scheme:" A, B are formulas, x a variable which is not free in A, then

$$(\forall x)(A \rightarrow B) \rightarrow (A \rightarrow (\forall x)B)$$

Comment: This is a rather technical axiom, to be used in prenex operations.

3) Inference rule: Universal generalization For an arbitrary variable x, from a formula A, derive $(\forall x)A$.

Comment: This shows the role of free variables in theorems. Whenever you can prove a formula A with a free variable x, then you can prove also a formula $(\forall x)A$. This is because, from a semantic point of view, for free variables you would have to check all possible interpretations anyway.

Rules of Manipulation

Permutation

$$\forall x(\forall y(P(x,y))) \leftrightarrow \forall y(\forall x(P(x,y)))$$

A similar rule can be shown for the existential quantifier.

Negation

$$\neg(\forall x(P(x))) \leftrightarrow \exists x(\neg P(x))$$

For the negation of the universal quanitifer it suffices to show that there exists one case for which $\neg P(x)$.

Nesting/Applicability

$$(\forall x : P(x)) \wedge Q \leftrightarrow \forall x : (P(x) \wedge Q)$$

Here, x appears in P, but not in Q. Therefore it does not affect the truth value of Q when it is grouped with P with respect to x. Similar argumentation holds true for the existential quantifier.

Prenex normal form

Just normal forms are useful for propositional calculus (conjunctive normal form, disjunctive normal form), there is a normal form for predicate calculus. Because of the higher complexity of predicate calculus – we have to take care of the quantifiers – are somewhat more involved. The goal is to move all the quantifiers to the beginning of the formula. This makes the formulas more transparent and comparable, and it makes them more accessible to automated processing.

Definition 5.17. We say that formula A is in *prenex form*, if it has the following form:

$$(Q_1 x_1) \ldots (Q_n x_n) B$$

where

1. Q_i are either $\forall$ or $\exists$
2. B is an open formula (i.e. all variables are free in it)
3. $x_1 \ldots x_n$ are all different

B is called an open core of A and the sequence of quantifiers is called prefix.

Replacement (renaming) of bounded variables

Suppose we have a formula A which contains a subformula of the form $(Qx)B$ (where Q is either $\forall$ or $\exists$). Then it is possible to replace x by y (in the prefix as well as in the formula B) and we obtain an equivalent formula A', a variation of A. However, we have to take care – the original formula B could not contain free occurences of y as these would then become bounded by our replacement. The safest way is to take a completely new symbol to name our variable.

Theorem 5.3. *For every formula A, it is possible to construct an equivalent formula A' in prenex form, such that $\vdash A \leftrightarrow A'$.*

Proof. Formula A' is constructed by using prenex operations. These replace subformulas of A according to one of the following schemes (where Q is either $\forall$ or $\exists$ and $\bar{Q}$ is the other quantifier than Q).

(a) replace subformula B by a variation of it B'

(b) replace subformula $\neg(Qx)B$ by $(\bar{Q}x)\neg B$

(c) if x is not free in B, replace subformula $B \rightarrow (Qx)C$ by $(Qx)(B \rightarrow C)$

(d) if x is not free in C, replace subformula $(Qx)B \rightarrow C$ by $(\bar{Q}x)(B \rightarrow C)$

(e) if the symbol $\star$ represents either $\wedge$ or $\vee$ and x is not free in C, then replace the subformula

$$(Qx)B \star C \quad \text{or} \quad C \star (Qx)B \quad \text{by} \quad (Qx)(B \star C)$$

□

5.5 Extensions

Although FOPC (First Order Predicate Calculus) has proved extremely useful and has broad applicability in virtually all areas of mathematics, and is used widely in computer science, etc., it does have some serious limitations. For example, we cannot express ideas like “this should be the case”, “I believe this to be the case”, or “this is almost correct”. Moreover, there is no notion of time. New forms of logic such as modal logic, fuzzy logic, and temporal logic have been developed to deal with these issues.

Bibliography

Caldwell, M. (2001). *The Tipping Point: How Little Things Can Make a Big Difference*. Abacus.

Cook, M. (2004). Universality in elementary cellular automata. *Complex Systems*, 15(1):1–40.

Dreyfus, H. L. and Dreyfus, S. E. (1986). *Mind over Machine – The Power of Human Intuition and Expertise in the Era of the Computer*. Free Press, New York.

Flake, G. W. (1998). *The Computational Beauty of Nature – Computer Explorations of Fractals, Chaos, Complex Systems, and Adaptation*. MIT Press.

Guimerà, R., Mossa, S., Turtschi, A., and Amaral, L. A. N. (2005). *The worldwide air transportation network: Anomalous centrality, community structure, and cities' global roles*. Proc. Natl. Acad. Sci. U.S.A.

Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2001). *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, second edition.

Kauffman, S. (1993). *Origins of Order: Self-Organization and Selection in Evolution*. Oxford University Press.

Langton, C. G. (1990). Computation at the edge of chaos. *Physica D*, 42:12–37.

Milo, R., Shen-Orr, S., Itzkovitz, S., Kashtan, N., Chklovskii, D., and Alon, U. (2002). *Network Motifs: Simple Building Blocks of Complex Networks*, volume 298. Science.

Newman, M. E. J. (2003). The structure and function of complex networks. *SIAM Review*, 45(2):167–256.

Papadimitriou, C. H. (1995). *Computational Complexity*. Addison Wesley.

Pfeifer, R. and Bongard, J. (2007). *How the body shapes the way we think: a new view of intelligence*. MIT Press.

Rechenberg, P. and Pomberger, G. (2006). *Informatik Handbuch*. Hanser, fourth edition.

Schelling, T. C. (1969). Models of segregation. *American Economic Review, Papers and Proceedings*, 59(2):488–493.

Sporns, O. and Kötter, R. (2004). *Motifs in Brain Networks*, volume 2. PLoS Biology.

Sudkamp, T. A. (2006). *Languages and Machines: An introduction to the Theory of Computer Science*. Addison Wesley, third edition.

Watts, D. J. and Strogatz, S. H. (1998). Collective dynamics of 'small-world' networks. *Nature*, 393:409–10.

Wolfram, S. (1984). Universality and complexity in cellular automata. *Physica D*, 10:1–35.

Wolfram, S. (2002). *A New Kind of Science*. Wolfram Media, Inc.