Learn Programming with OCaml

Algorithms and Data Structures

Sylvain Conchon & Jean-Christophe Filliâtre

Translated by Urmila Nair

February 18, 2025

The OCaml Software Foundation funded the translation of this book into English by Urmila Nair, and suggested that the result be made Open Access. The present English version is thus available under the license CC BY SA 4.0.

Compiled from commit `89f4126` on February 18, 2025.

For our wives and daughters

Contents

Preface

Learning to program is difficult. To program well requires knowledge of algorithms, imagination, foresight, mastery of a programming language. Most of all, it requires experience, since the difficulties are often hidden in the details. This book offers a synthesis based on our own experience, both as programmers and as teachers of programming.

It is important to remember that style is essential to programming. The same algorithm can be written in a given language in multiple ways; only some of these are simultaneously elegant and efficient, which is what the programmer should strive for, above all else. This is why we chose to use a concrete programming language, OCaml, rather than pseudocode.

The book is divided into three parts. Part I is an introduction to the OCaml language. It is aimed at beginners, including both novice programmers and more experienced programmers who do not know OCaml. Various small programs will enable the reader to discover the basic concepts of programming and the OCaml language. Parts II and III focus on fundamental algorithmic concepts, with a view to enabling readers to write their own programs in an efficient and structured manner. Algorithmic concepts are presented directly in OCaml syntax, and all the programs in this book can be readily reused.

Who Should Read This Book

The book is aimed at teachers, students, and programmers seeking to discover and use the OCaml language effectively. For teachers and students, the book may be used as a textbook in courses on algorithms or functional programming. It contains numerous code samples and exercises. For OCaml programmers, it may serve as a reference text in the study of specific data structures or algorithms.

The Book's Companion Website

All the programs in this book have been compiled and tested with OCaml version 4.14.0. The programs are available at the following address:

`http://programmer-avec-ocaml.lri.fr/`

Acknowledgements

We would like to thank François Pottier for discussions regarding the creation of this book. We are also very grateful to our colleagues Arthur Charguéraud and Frédéric Voisin for their careful and constructive revision of the text. We would also like to thank everyone who read and commented on it: Thomas Braibant, Martin Clochard, David Declerck, Léon Gondelman, Louis Mandel, Simão Melo de Sousa, Mattias Roux. Finally, we thank the Eyrolles editorial team for their precious help in putting the book together.

Part I

Programming with OCaml

1 The Working Environment

The OCaml language is available for use with the main operating systems (Linux, macOS, Windows). It can be obtained at the following address:

https://ocaml.org/

The site contains installation instructions for these systems. Once you have installed OCaml, you will have access to the following tools: an interpreter, `ocaml`; two compilers, `ocamlc` and `ocamlopt`; and a package manager, `opam`. We briefly present these four tools below.

1.1 The Compiler and the Interpreter

The OCaml system includes two compilers: `ocamlc` and `ocamlopt`. The first generates portable code, independent of the hardware architecture; the second generates more efficient native code (X86, ARM, etc.). You can use either of these two tools to compile the programs given in this book. The system also includes an interactive version, namely, the program `ocaml`. This is an interpreter or *toplevel* that executes a read-eval-print loop (REPL).

1.2 A First OCaml Program

We illustrate the use of these tools with a simple program that displays the message `Hello world!` on the screen. It can be written as follows:

```
let () = print_string "Hello world!\n"
```

This is itself a complete program. We will go into the details of such programs in the next chapter. If the program is stored in a file `hello.ml`, it can be compiled from within a terminal[1] using `ocamlc`, as follows:

```
> ocamlc hello.ml
```

We can then execute the resulting binary, `a.out`, with the expected result:

```
> ./a.out
Hello world!
```

The binary can be given a different name using the compiler's `-o` option:

```
> ocamlc -o hello hello.ml
> ./hello
Hello world!
```

To use the interpreter, it suffices to type the command `ocaml` in a terminal. The program then displays a prompt in the form of the character `#`, prompting the user to enter an expression.

```
> ocaml
OCaml version 4.14.0
#
```

To be evaluated, the expression must end with two semicolons `;;` followed by a carriage return. The toplevel verifies that the expression is syntactically correct and well-typed. It then evaluates the expression and displays the result. Thus, in order to display the message `Hello world!` on the screen, it suffices to type:

[1]The symbol > designates the terminal prompt from which commands are launched.

```
> ocaml
OCaml version 4.14.0

# let () = print_string "Hello world!\n";;
Hello world!
#
```

In addition to the result, the toplevel also displays the type inferred by OCaml for the expression that has been entered. For example, if you enter the mathematical expression $1+4\times 2$, the interpreter returns the following response:

```
# 1 + 4 * 2;;
- : int = 9
#
```

This shows that the expression is of type `int` (the type of integers), and that its result is 9. From now on, we will present the expression entered by the user and the interpreter's response against a gray background. Thus, the evaluation of the preceding expression will be presented, as follows:

```
# 1 + 4 * 2 ;;
- : int = 9
```

To exit the toplevel, type the command `#quit`, as follows:

```
# #quit;;
>
```

A simpler way to do this is to press the keys `ctrl` and `D` simultaneously.

1.3 Programming Environments

As with many other programming languages, you can work with OCaml using your preferred text editor (Emacs, Visual Studio Code, etc.). The text editor may be configured with OCaml-specific syntax coloring, error highlighting, and so forth.

In this text, we will use the tool `dune` to automate the compilation process. This tool can be obtained via the `opam` package manager, and can be installed by typing the following command in a terminal:

```
> opam install dune
```

Another solution is to use the tool `make`. This requires writing a configuration file (`Makefile`) specific to OCaml.

Online Environments

There are also online solutions for using OCaml without installing it on your machine. One example is the online interpreter TryOCaml, available at the following address:

`https://try.ocamlpro.com/`

Another example is the website

`https://replit.com/`

which offers online environments for a number of languages, among them OCaml.

1.4 Installing OCaml Libraries

The community of OCaml users has developed a number of OCaml libraries. The easiest way to install them is to use the package manager `opam`. In the next chapter, we will use a graphics library, which may be installed as follows:

```
> opam install graphics
```

We will explain later how to use this library and, more generally, any library installed using `opam`.

2 First Steps with OCaml

The best way to learn a programming language is by programming. We therefore present below fourteen (little) programs to get started with OCaml. The aim is to introduce the basic constructions and ideas of the language.

Program 1 [leap_year.ml] — Leap Year

```
let year = read_int ()
let leap =
  (year mod 4 = 0 && year mod 100 <> 0) || year mod 400 = 0
let msg = if leap then "is" else "is not"
let () = Printf.printf "%d %s a leap year\n" year msg
```

2.1 Leap Years

Ideas introduced

- the general form of a program
- the `let` construct
- calling a function
- basic types (`int`, `bool`, `string`)
- standard library (`Printf`, `Sys`, etc.), module system
- access to program arguments (`Sys.argv`)
- access to the elements of an array (notation `t.(i)`)

Our first program, `leap_year.ml`, determines if a given year is a leap year. We can compile it using `ocamlc`.

```
> ocamlc -o leap_year leap_year.ml
```

When the program is launched, it waits for the user to enter a year. It then prints whether the year is a leap year or not.

```
> ./leap_year
2013
```

```
2013 is not a leap year
```

Let us now consider the structure of this program. The first line reads an integer from standard input:

```
let year = read_int ()
```

This line alone contains several notions of the OCaml language. Firstly, it is a *variable declaration* of the form:

```
let year = ...
```

This initializes a new variable `year` with the result of the expression to the right of the symbol `=`. The type of the variable `year` does not have to be declared; it is automatically deduced by the compiler or the interpreter. The expression to the right of the symbol `=` is a *function call*. It is a call to the predefined function `read_int`.

```
          ... = read_int ()
```

Here, `()` indicates that the function `read_int` does not take any meaningful argument. It only returns an integer, read from standard input.

The second line introduces a boolean variable `leap` whose value is `true` if and only if `year` is a leap year, that is, if it is divisible by 4 but not by 100, or if it is divisible by 400.

```
let leap =
  (year mod 4 = 0 && year mod 100 <> 0) || year mod 400 = 0
```

The infix operator `mod` returns the remainder of integer division. The operators `&&` and `||` represent the boolean AND and OR, respectively. As with `year`, the type of the variable `leap` is deduced automatically. The following line introduces a third variable, `msg`, which contains a string:

```
let msg = if leap then "is" else "is not"
```

The variable `msg` will contain the string `"is"` if `leap` is true, and the string `"is not"` if it is false. (Strings are delimited by double quotes `"`.) Note that the construction `if-then-else` is used here to construct an expression, namely, a string.

Finally, the last line prints a message that indicates whether or not the year entered is a leap year.

```
let () = Printf.printf "%d %s a leap year\n" year msg
```

This is achieved by calling the library function `Printf.printf` with three arguments: a formatting string `"%d %s a leap year\n"` together with the two values, `year` and `msg`, which respectively replace `%d` and `%s` in the printed message. As you can see, the function call is denoted by simply juxtaposing the function and its arguments. Unlike other languages, OCaml does not use parentheses around arguments.

```
    ... = Printf.printf "%d %s a leap year\n" year msg
```

We refer the reader to the OCaml manual for further information on the standard library and, in particular, on the function `Printf.printf`. The manual is available online at the address: `https://ocaml.org/manual`. Finally, note that the call to the function `printf` is contained in a declaration of a specific form:

```
let () = ...
```

This is somewhat enigmatic, and will be left so for the present. It will be explained when we discuss pattern matching in sections 2.6 *Drawing a Curve* and 2.12 *Playing a Musical Score*. For now, suffice it to say that in general an OCaml program is a series of declarations that are evaluated from top to bottom, that is, in the order in which they appear in the source file. Unlike languages like C or Java, there is no `main` entry point. The specific form `let () = ...` is used in order to maintain the same format in case of expressions that do not return a value. When this form is used, OCaml verifies that the expression does indeed not return a value.

Additional Information

The Command Line

Rather than reading the year from standard input, we can pass it on the command line. In that case, the program is invoked as follows:

```
> ./leap_year 2013
2013 is not a leap year
```

To do this, we replace the line `let year = read_int ()` with:

```
let year = int_of_string Sys.argv.(1)
```

This line calls the predefined function `int_of_string` with `Sys.argv.(1)` as argument. The value `Sys.argv.(1)` contains the year passed as parameter on the command line. As is the case with other programming languages (Java or C, for example), we can retrieve the values passed to the program in the form of an array of strings, designated `Sys.argv` in OCaml. More precisely, the array is called `argv`, and it is available within the `Sys` module, which consists of an interface to the operating system. The notation `Sys.argv` involves the *module system* of OCaml, that organizes libraries in terms of units, called modules, which themselves contain different values. Without getting into the details of the module system, suffice it to say for now that we refer to a value in a module by writing the name of the module, followed by the name of the value, separated by a dot. (We will discuss this in greater detail in section *2.11 Logo Turtle*.) Thus, `Sys.argv` must be read as "the array `argv` that belongs to the module `Sys`."

We use the notation `t.(i)` to access the `i`-th element of an array `t`. We will return to the notion of arrays in section *2.5 The Sieve of Eratosthenes*. The first element of `Sys.argv` is `Sys.argv.(0)` since arrays are indexed starting from 0. This is the program's name (here `leap_year`). The second element is `Sys.argv.(1)`. This is the first parameter passed to the program (`2013` in our example). Since it is a string, it must be converted into an integer by calling the function `int_of_string`, which belongs to the module `Stdlib`. This is a special module for which it is not necessary to write `Stdlib.`*f*.

Type-Checking

Let us now consider some additional details regarding the datatypes introduced in our first program. While the types of variables are inferred automatically, it is also possible to display them using the `-i` option of the compiler:

```
> ocamlc -i leap_year.ml
val year : int
val leap : bool
val msg : string
```

We thus see that the variables `year`, `leap`, and `msg` are respectively of type `int`, `bool`, and `string`. These are three predefined types in OCaml. Let us examine these and other related types using the toplevel.

Types `int`, `int32`, and `int64`

Let us begin with the type `int`, which corresponds to processor-native integers, also called machine integers. An integer constant can be written in decimal, binary (prefix `0b`), octal (prefix `0o`) or hexadecimal (prefix `0x`) notations. Thus, the decimal constant `91` can be written as `0b1011011` in binary, as `0o133` in octal, and as `0x5B` in hexadecimal.

The usual operations on the type `int` are addition (`+`), subtraction (`-`), multiplication (`*`), integer division (`/`), and remainder (`mod`).

```
# 2 * (7 / 2) + 7 mod 2;;
- : int = 7
```

For greater clarity, the digits may be separated using an underscore (`_`).

```
# 0x2f_ff_ff_ff + 268_435_456;;
- : int = 1073741823
```

The precision of the type `int` depends on the machine architecture. The minimum and maximum values of the type `int` are given by the two constants `min_int` and `max_int`.

```
# min_int;;
- : int = -1073741824
```

On a 32-bit machine, values of type `int` are encoded as 31-bit signed integers[1], so that `min_int` and `max_int` have the values -2^{30} and $2^{30}-1$, respectively. On

[1]One bit is used by OCaml's automatic memory management system; see section *3.2 Runtime Model* in chapter 3.

a 64-bit machine, the values of type `int` are encoded as 63-bit signed integers. The machine representation of integers uses the two's complement representation and may be manipulated with the help of shift operations (`lsl`, `lsr`, and `asr`) and bitwise operations (`land`, `lor`, `lxor`, and `lnot`). For example, you can test the fifth bit of the representation of the integer 42 using:

```
# 42 land (1 lsl 5);;
- : int = 32
```

To manipulate 32-bit or 64-bit signed integers, OCaml offers two specific types: `int32` and `int64`. The integer constants of type `int32` (or `int64`) are written with the suffix `l` (or `L`).

```
# 123l;;
- : int32 = 123l
# 10_000_000_000_000L;;
- : int64 = 10000000000000L
```

The arithmetic operations on these two types may be found in the OCaml modules `Int32` and `Int64`.

The Type `bool`

The type `bool` represents boolean values, that is, `true` and `false`. The operations on the type `bool` are conjunction (`&&`), disjunction (`||`), and negation (`not`).

```
# not true && false || true;;
- : bool = true
```

Negation has the highest priority, followed by conjunction, and then disjunction. The preceding expression is therefore read as:

```
# ((not true) && false) || true;;
- : bool = true
```

The evaluation order of the operations `&&` and `||` is fixed from left to right. Furthermore, the second argument is evaluated only if necessary, that is, only if the first argument of the operation `&&` (respectively, `||`) is true (respectively, false). These are the only operations for which the evaluation order is specified. Furthermore, these are the only operations whose evaluation is lazy.

The comparison operations `=`, `<>`, `<`, `>`, `<=`, and `>=` return a boolean.

```
# 1 < 2 && 3 = 4;;
- : bool = false
```

Boolean expressions are mainly used in the conditional construction `if e1 then e2 else e3`. Depending on whether the expression `e1` is true or false, either the expression `e2` or the expression `e3` is evaluated.

```
# if 1 < 2 then 3 else 4;;
- : int = 3
```

As is clear in this example, the construction `if then else` is an expression like any other. In particular, for this expression to be well-typed, the two branches `e2` and `e3` must have the same type, which is then the type of the expression as a whole.

Types `char` and `string`

Characters are represented by the type `char`. They are written between single quotes: `'a'`. Characters that are not printable are entered using a notation of the form `'\`*c*`'`, where *c* can be a character (`n`, `r`, `t`, `\`), or an ASCII code (between 0 and 255), written with three decimal digits (`\ddd`) or two hexadecimal digits (`\xdd`).

```
# 'a';;
- : char = 'a'
# '\n';;
- : char = '\n'
# '\\';;
- : char = '\\'
# '\126';;
- : char = '~'
# '\x7E';;
```

```
- : char = '~'
```

You can convert a character to its ASCII code and vice versa with the help of the functions `Char.code` and `Char.chr` of the `Char` module.

Strings are represented by the type `string`. They are written within double quotes: `"abc"`. Within a string, characters that are not printable may be entered with the same syntax used for the type `char`.

```
# "a";;
- : string = "a"
# "hello world\n";;
- : string = "hello world\n"
# "";;
- : string = ""
# "abc\126def";;
- : string = "abc~def"
```

As is clear in this example, the string `"a"` does not have the same type as the character `'a'`. In general, there is no conversion between these two types. You can access the *i*-th character of a string *s* with the notation *s*`.[`*i*`]`, the first character having index 0.

```
# "abc".[1];;
- : char = 'b'
```

The length of a string is given by the function `String.length` of the module `String`. Finally, two strings can be concatenated using the operator `^`.

```
# String.length ("abc" ^ "def");;
- : int = 6
```

The reader is referred to the OCaml manual for further details regarding the modules presented in this chapter.

Input/Output

The simplest input/output functions are those that print values of basic types, such as `print_string`, which was used above to print `Hello world`. Other such functions include `print_char`, `print_int`, and `print_float`; `print_newline`, to print a carriage return; and `print_endline`, to print a string followed by a carriage return. These six printing functions write to standard output. To write to standard error, there are six other functions: `prerr_char`, `prerr_string`, etc. Likewise, there are functions that read from standard input, namely `read_int`, `read_float`, and `read_line`. There is also a variant of the function `printf` to write to standard error, `Printf.eprintf`. You will find more details regarding input/output in section *2.7 Copying a File*.

2.2 The Monte-Carlo Method

Ideas introduced

- side effects, expression sequences, references (`ref` keyword)
- local variables (`let-in` construct)
- `for` loops
- floating-point numbers (type `float`)
- random number generation using the module `Random`

Our second program (see page 18), `approx_pi.ml`, calculates the value of π experimentally using the Monte-Carlo method. This method consists in randomly generating points in a square of side 1, whose area is therefore 1, and then counting the number of those points that lie within a quarter circle of radius 1 inscribed in this square. The area of the quarter circle is $\pi/4$. Figure 2.1 illustrates the idea.

We compile the program `approx_pi.ml`, for instance using `ocamlopt` (for greater efficiency), as follows:

```
> ocamlopt -o approx_pi approx_pi.ml
```

We then execute the program and give it the number of points to be generated. If this number is sufficiently large, the proportion of points contained within the

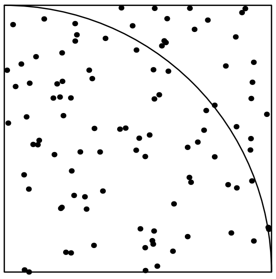


Figure 2.1: Computation of π using the Monte-Carlo method.

quarter circle approaches the ratio of the area of the quarter circle to that of the square, that is, $\pi/4$. We obtain the following results:

```
> ./approx_pi
500
3.136000
> ./approx_pi
5000
3.113600
> ./approx_pi
50000
3.149920
> ./approx_pi
500000
3.142184
```

In practice, however, this method yields a value of π that is accurate only to a few decimal places. For example, with 10 million points, the value is accurate only to the third decimal place.

As in our first program, the first line of `approx_pi.ml` obtains the number `n` of points to be generated by reading an integer from standard input:

Program 2 [approx_pi.ml] — Approximation of π

```
let n = read_int ()

let () =
  let p = ref 0 in
  for k = 1 to n do
    let x = Random.float 1.0 in
    let y = Random.float 1.0 in
    if x *. x +. y *. y <= 1.0 then
      p := !p + 1
  done;
  let pi = 4.0 *. float !p /. float n in
  Printf.printf "%f\n" pi
```

```
let n = read_int ()
```

The rest of the program uses a loop to generate `n` points at random. In each iteration of the loop, we increase by 1 the contents of a variable initialized with 0 if the generated point lies within the quarter circle of radius 1. At the end, we display the approximate value of π, which equals four times the ratio of the number of points within the quarter circle to the total number of points.

Computing in this manner, by modifying the contents of a variable, is called computation by *side effects*. More generally, the term side effect is used in relation to expressions that do not return a value when evaluated. As we have mentioned in the previous section, a declaration of the form `let () = ...` is used to evaluate such expressions.

The rest of the program consists in the expression that follows the `=` sign. It begins with the initialization of a variable `p` using a declaration of the form:

```
let p = ... in
...
```

This declaration introduces a *local variable*, whose use or *scope* is restricted to the expression that follows the keyword `in`. This kind of declaration is useful when one wishes to restrict the visibility of a variable, whether to prevent its usage in other parts of the program or because one wishes to hide temporarily another variable with the same name.

The variable `p` is initialized with a *reference*, that is, a memory cell, the contents of which can be modified, and which initially contains the value 0. This reference will be used to count the number of points that fall within the quarter circle.

```
let p = ref 0 in
...
```

The variable `p` is created by calling the predefined function `ref` (defined in the `Stdlib` module) with the integer 0 as argument. The application of this function has the effect of allocating a new memory cell and initializing its contents to 0. The type of `p` inferred by OCaml is `int ref`, read as "a reference containing a value of type `int`."

The next line introduces a `for` *loop* that evaluates the expression situated between the keywords `do` and `done` `n` times, while varying the contents of a variable `k` from `1` to `n`:

```
for k = 1 to n do
  ...
done
```

The variable `k` is the *index* of the loop, and the expression between `do` and `done` is the *body* of the loop. The index `k` is initialized with the value `1`. Then, the body is evaluated. This is the first iteration of the loop. Then, `k` is increased, and the body is evaluated again. This is the second iteration of the loop. And so on and so forth. Following the evaluation for which `k` equals `n`, the loop terminates.

The body of the loop begins by generating the coordinates of a point (x, y) randomly within the square of side 1.

```
  let x = Random.float 1.0 in
  let y = Random.float 1.0 in
```

To this end, two local variables, `x` and `y`, are initialized by two calls to the function `Random.float` with the *floating-point number* `1.0` as argument. Each of these calls returns a floating-point number between 0 and 1 (including 0 and excluding 1). As we will see later, OCaml makes a strong distinction between floating-point numbers, which are of type `float`, and integers, which are of type `int`. Crucially, a dot must be inserted after the number `1` to represent the floating-point number 1. We thus write `1.0` or `1.` to distinguish it from the integer `1` of type `int`.

The conditional that follows increases the contents of the reference `p` if the point (x, y) is within the quarter circle, that is, if $x^2 + y^2 \leq 1$.

```
if x *. x +. y *. y <= 1.0 then
  p := !p + 1
```

As we see in the expression `x *. x +. y *. y`, the arithmetic operators on floating-point numbers are also distinguished *syntactically* from operators on integers: A dot `.` must be added after the sign of each operator (`+`, `-`, etc.). Thus, `+.` is the addition operator on floats, `*.` the multiplication operator, etc. However, the same operator `<=` is used to compare floats and integers. (See chapter 3 for further details on comparison operators.)

The expression `p := !p + 1` is used to increase the reference `p` by 1. Here, the assignment operator `:=` serves to modify the contents of a reference, and the prefix unary *dereference* operator `!` is used to retrieve the contents of a reference. Thus, if `p` is a reference of type `int ref`, the value `!p` is of type `int`. The loop ends after the iteration for which `k` equals `n`. The evaluation then continues with the expression that follows the `;` operator.

```
for k = 1 to n do
 ...
done;
```

Such a series of evaluations is called an *expression sequence*. For an expression sequence to be accepted by OCaml, the expression before the semicolon must not return a value, that is, it must only perform side effects.

The declaration that follows the `for` loop defines a local variable `pi` containing the approximation of π.

```
    let pi = 4.0 *. float !p /. float n in
```

In order to perform a division of floating-point numbers, the integer values `!p` and `n` are converted into floats using the function `float` (defined in the `Stdlib` module). Finally, we display the approximation of π thus calculated using the function `Printf.printf`:

```
    Printf.printf "%f\n" pi
```

Just as we used `%d` in a formatting string to display an integer, here we use `%f` to display a float.

Additional Information

Here, we present further explanations regarding the ideas introduced in this second program.

The Type `float`

Real numbers are represented in the computer using a specific kind of encoding called *floating-point numbers*, or more simply *floats*. In OCaml, floating-point numbers are encoded using the double-precision IEEE 754 standard (64 bits). There are no single-precision (32 bits) floating-point numbers in OCaml.

As we noted earlier, the type of floating-point numbers is `float`. A constant of type `float` is written either in decimal notation (for example, `3.14`) or in scientific notation (for example, `6.02214e23`). A dot must always be used in the decimal notation, to indicate that the constant is not of type `int`. In case of scientific notation, however, the dot is not compulsory.

```
# 3 ;;
- : int = 3
# 3. ;;
- : float = 3.
# 1e6 ;;
- : float = 1000000.
```

Operations on the type `float` are distinct from those on the type `int`. They are followed by a dot, as in case of floating-point constants: `+.`, `-.`, `*.`, `/.`. Unlike in other languages, arithmetic operations thus have different names depending on whether integers or floats are involved, and there are no implicit conversions between the types `int` and `float`.

```
# 2 +. 3.14;;
Error: This expression has type int but an expression was
expected of type float
```

The conversion functions `float` and `truncate` must be used to convert integers to floats and vice versa.

```
# 3.14 /. float 2;;
- : float = 1.57
# truncate 3.141592;;
- : int = 3
```

Numerous operations on the type `float` are available in the `Stdlib` module, such as exponentiation `**`, square root `sqrt`, trigonometic functions, logarithmic functions, etc. Thus, the value of π, which is not predefined in OCaml, could be calculated using the following expression:

```
# 4. *. atan 1.;;
- : float = 3.14159265358979312
```

Unlike the type `int`, the type `float` includes three additional values, `neg_infinity`, `infinity`, and `nan`, that represent calculations that have no meaningful result.

```
# -1. /. 0.;;
- : float = neg_infinity
# 1. /. 0.;;
- : float = infinity
# 0. /. 0.;;
- : float = nan
```

Unlike other languages, OCaml offers only one type of floating-point numbers, namely, 64-bit double-precision floats conforming to the IEEE 754 standard. Floats lie between a minimum value `min_float` and a maximum value `max_float`.

```
# min_float;;
- : float = 2.22507385850720138e-308
# max_float;;
- : float = 1.79769313486231571e+308
```

The `let` and `let-in` Constructs

The `let` and `let-in` constructs are used, respectively, to define global and local variables. Two important points must be noted regarding variables:

- Variables are necessarily initialized.
- The type of a variable is automatically inferred.

The first point results simply from the fact that variables can only be defined using the constructions `let` and `let-in`, whose syntax requires an expression for the initial value. The second point is specific to the OCaml language.

You can use the toplevel to find out the type inferred for a variable. For instance, if we type the following declaration:

```
# let x = 1+2;;
```

The interpreter displays the response:

```
val x : int = 3
```

This indicates that we have defined a global variable `x`, initialized with the integer `3`. The type of `x` is inferred from the expression `1+2`, namely, `int`.

It is equally important to master the *scoping rules* of variables declared by a `let` or `let-in` construct. The scope of a global variable `x` begins immediately after the declaration `let x = ...` used to define it. This scope extends until the end of the program. The scope of a local variable introduced with the `let-in` construct is limited to the expression that follows the keyword `in`. Thus, each of the three declarations below triggers a scoping error:

```
# let x = x + 1;;
Error: Unbound value x
# (let v = 1 + 2 in v * v) + v;;
Error: Unbound value v
# let z = 1 + z in z * 2;;
Error: Unbound value z
```

The three declarations above illustrate an important rule regarding variable usage: To be used, a variable must have been declared beforehand. The following program is therefore correct:

```
let x = 10
let y = x+2
```

By contrast, the program below is wrong:

```
let y = x+2
let x = 10
```

It is important to note that in the expression `let x = e1 in e2; e3`, the scope of the variable `x` extends to the entire expression `e2; e3`. The precedence of the operator `;` is higher than that of the construction `let`. In other words, this expression must be read as: `let x = e1 in (e2; e3)`. Program 2 uses this scoping rule in the declaration of the local variable `p`.

```
let () =
  let p = ... in
  for k = 1 to n do
    ...
    p := !p + 1
  done;
  let pi = 4. *. float !p /. float n in
  ...
```

Thus, the variable `p` is visible both in the `for` loop and in the body of the declaration of the variable `pi`.

You can use the same name for several variables, including those of different types. The use of a variable `x` always refers to the nearest declaration. Consider the following program as an example:

```
let x = 10
let y = x + 2
let x = 20
let () = Printf.printf "x=%d y=%d\n" x y
```

The declaration `let x = 20` defines a *new* variable `x`, which hides the scope of the previous one. The message displayed when you execute the program is therefore `x=20 y=12`.

The `for` Loop

As we have seen in our example, `for` loops are used to evaluate a given expression a certain number of times. They have the following form:

```
for i = e1 to e2 do
  e
done
```

The index of the loop, the variable `i`, is introduced by the `for` construct. Its value cannot be modified by the user. Its scope is limited to the body of the loop (the expression `e`), and therefore it cannot be used in the expressions `e1` and `e2`. These two expressions are evaluated only once, before the execution of the loop. This can be observed with the following program:

```
let () =
  for i = (Printf.printf "*"; 0) to (Printf.printf "."; 5) do
    Printf.printf "%d" i
  done
```

When you execute this program, it displays the string `*.012345`. The evaluation of the `for` loop begins with the initialization of the index `i` with the value of the expression `(Printf.printf "*"; 0)`. The evaluation of this sequence first causes the symbol `*` to be printed. Then, the integer `0` is returned. The fact that the second printed character is a dot `.` indicates that the expression `(Printf.printf "."; 5)`, which corresponds to the final value of the index, is evaluated next. As the rest of the message contains only the numbers displayed by the expression `Printf.printf "%d" i` of the body of the loop, it may be deduced that the expressions `e1` and `e2` are not evaluated again. It is also important to note that if the value of `e1` is strictly greater than that of `e2`, the body of the loop is never executed.

There is also a variant of the `for` construct that goes backwards. For example, the following program displays the numbers `9 8 7 6 5 4 3 2 1 0`, in this order.

```
let () =
  for i = 9 downto 0 do
    Printf.printf "%d " i
  done
```

Note that the index can only be increased or decreased by 1 in each iteration. For other increments, it is possible to use a `while` loop, which will be discussed later.

For the sake of completeness, let us note that the body of the `for` loop in program 2 does not make use of the loop index `k`. The OCaml compiler can detect this and emit a warning, if we add the command-line option `-w +a`:

```
> ocamlopt -w +a -o approx_pi approx_pi.ml
Warning 35 [unused-for-index]: unused for-loop index k.
```

A solution is to avoid naming the index of the loop, by replacing it with an underscore `_`.

```
for _ = 1 to n do
```

This does not change the behavior of the `for` loop in any way.

2.3 Drawing a Cardioid

Ideas introduced

- `open` directive
- library (extension `.cma`)
- library `Graphics`
- function `ignore`

Our third program (see page 28), `cardioid.ml`, draws a cardioid, that is, a curve that represents the trajectory of a fixed point on a circle that turns around a second circle of the same diameter, as shown in figure 2.2.

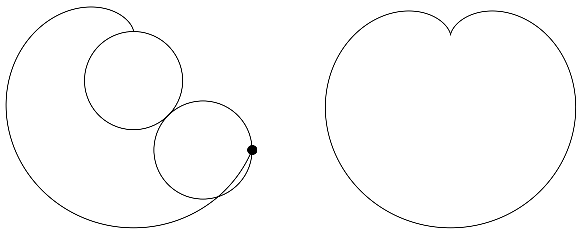


Figure 2.2: Drawing a cardioid.

Mathematically, a cardioid is an algebraic plane curve that may be defined by the following parametric equations:

$$\begin{cases} x(\theta) = a\,(1 - \sin(\theta))\,\cos(\theta) \\ y(\theta) = a\,(1 - \sin(\theta))\,\sin(\theta) \end{cases}$$

where a is the radius of each circle.

To trace the cardioid, we use the OCaml library `Graphics`, which allows you to open a two-dimensional graphical window and draw in it using elementary graphical functions. This library contains a single module, also called `Graphics`. Since we will be using a number of functions of this module, the program begins with a directive to open the module:

Program 3 [cardioid.ml] — Drawing a cardioid

```
open Graphics

let () = open_graph " 300x200"

let () =
  moveto 200 150;
  for i = 0 to 200 do
    let th = atan 1. *. float i /. 25. in
    let r = 50. *. (1. -. sin th) in
    lineto (150 + truncate (r *. cos th))
           (150 + truncate (r *. sin th))
  done;
  ignore (read_key ())
```

```
open Graphics
```

We will thus be able to use the values and functions of this library without having to prefix them each time with the name of the module, `Graphics`.

The following line opens a graphical window with the function `open_graph`. This function takes as argument a string indicating the window's dimensions.

```
let () = open_graph " 300x200"
```

This statement opens a window of size 300×200, that is, 300 pixels wide and 200 pixels high. (Note that the space at the start of the string is mandatory.) We then draw within this window using integer coordinates, the origin being located at the bottom left of the window, the x-coordinates taking values from 0 to 299 and the y-coordinates from 0 to 199 (see figure 2.3).

The rest of the program plots the cardioid. Since this part of the code does not return a value, we place it within a declaration of the form:

```
let () = ...
```

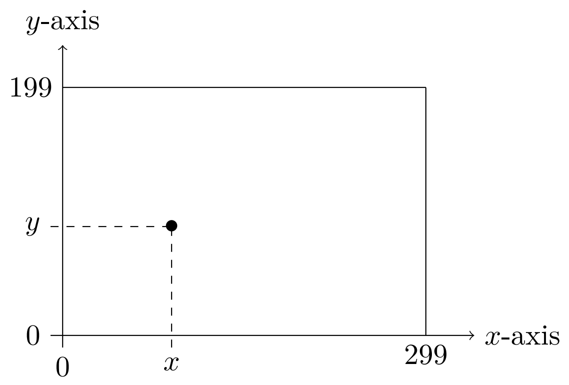


Figure 2.3: Coordinate system of the library `Graphics`.

The functions we will use to draw the cardioid rely on a notion of *current point*. This point can be positioned using `moveto`. A call to `lineto x y` traces a line segment from the current point to the point with coordinates `(x,y)`, which then becomes the new current point.

We begin by placing the current point at $(200, 150)$, by calling the function `moveto`, with a view to drawing a cardioid beginning at this point:

```
moveto 200 150;
```

We use a `for` loop to vary the angle θ between 0 and 2π. For this, we make use of an integer index `i` varying from 0 to 200, and we use the fact that $\arctan(1) = \pi/4$ to calculate θ as $\arctan(1) \times i/25$.

```
for i = 0 to 200 do
 let th = atan 1. *. float i /. 25. in
 ...
done
```

In OCaml, the arctangent of an angle, expressed in radians, is calculated using the function `atan` of the standard library `Stdlib`.

The following line defines a local variable `r` with the value of the intermediate subexpression $a\,(1 - \sin(\theta))$, where the radius a of the cardioid is fixed at 50 pixels:

```
let r = 50. *. (1. -. sin th) in
```

This variable is used to factor out one part of the calculation of the coordinates of the point $(x(\theta), y(\theta))$.

Finally, using the function `lineto`, we trace a segment between the current point and the point with coordinates $(x(\theta), y(\theta))$. Since graphical coordinates are integers, we use the function `truncate` to extract the integer part of the expressions used to calculate the coordinates.

```
lineto (150 + truncate (r *. cos th))
       (150 + truncate (r *. sin th))
```

After the `for` loop, the last line of the code pauses the program, waiting for a key to be pressed. This is in order to keep the graphical window from closing immediately after the drawing finishes.

```
ignore (read_key())
```

Since we are only waiting for a key to be pressed, we ignore the actual character returned by the function `read_key` using the predefined function `ignore`.

Compilation

In order to compile the program `cardioid.ml`, we must explicitly tell the compiler that we wish to use the library `Graphics`. This is not needed in case of the modules we saw earlier, such as `Sys`, `Arg` or `Printf`, which belong to the OCaml *standard library*. The `Graphics` library is contained in the file `graphics.cmxa`, which must be added to the command line. Furthermore, we must tell the compiler the location of this file, which was installed by `opam`. This can be done using the following command:

```
  > ocamlopt -I ‘ocamlfind query graphics‘ graphics.cmxa cardioid.ml -
o cardioid
```

The tool `ocamlfind` allows you to find the location of the library `graphics`, which is then passed to the OCaml compiler using the `-I` option. (Note that the command `ocamlfind query graphics` has to be placed between *backquotes*.) If you wish to use the `ocamlc` compiler, you should replace the suffix `.cmxa` by `.cma`. As we will see in section *2.10 Breakout without Bricks*, the order of the files on the command line of the compiler is important: Here, `graphics.cmxa` must appear before `cardioid.ml` since the latter uses the library `Graphics`.

Another way of compiling our program consists in using `dune`. To do this, we write a configuration file, which must be named `dune`, in the directory containing `cardioid.ml`. This file should contain the following:

```
(executable
 (name cardioid)
 (libraries graphics))
```

This specifies that we wish to construct an executable that is named `cardioid`, and that uses the library `graphics`. Assuming that our two files `cardioid.ml` and `dune` are in the directory `my-first-project/`, we must first perform an initialization step with the following command:

```
> dune init project my-first-project/
```

We can then shift to the directory `my-first-project/` and build the executable there:

```
> cd my-first-project/
> dune build
```

Finally, we can execute the program with the following command:

```
> dune exec ./cardioid.exe
```

The executable is named `cardioid.exe`, where the extension `.exe` is added by `dune`. It is stored in a subdirectory `_build/default/`, and the command `dune exec` allows us to launch it directly.

To use the library `Graphics` from within the toplevel `ocaml`, it suffices to type the following command in the terminal:

```
> ocaml -I ‘ocamlfind query graphics‘ graphics.cma
OCaml version 4.14.0
#
```

Note that the library `Graphics` can also be used directly from within TryOCaml.

The Library Graphics

In addition to the functions `moveto` and `lineto` used in our program `cardioid.ml`, the library `Graphics` offers various other functions: `plot`, to plot a point; `set_color`, to modify the display color; `draw_circle`, to draw a circle; `set_line_width`, to change the line thickness; `fill_rect`, to shade a rectangle; `draw_string`, to display a string, etc. The reader is referred to the OCaml manual's web page for a complete description of this library. It is important to note that, although the features of `Graphics` are limited, this library has the advantage of being available on many architectures. If you wish to use a more sophisticated library, you can use (depending on your operating system):

- OCaml-Canvas: a portable Canvas framework for OCaml, with an interface similar to HTML5 Canvas.

- LablTk: a Tcl/Tk library that allows you to create graphical user interfaces (GUI) with drop-down menus, buttons, etc.

- LablGtk: similar to LablTk but based on the graphical library Gtk.

- OCamlsdl: an SDL library mainly used to create video games.

This list is not exhaustive.

2.4 The Mandelbrot Set

Ideas introduced

- function declarations
- recursive functions
- type `unit` and the value `()`

Program 4 (see page 35) draws the Mandelbrot set (see figure 2.4). It is defined as the set of points (a, b) of the plane for which neither of the following two recursive sequences tends to infinity in absolute value.

$$\begin{array}{lcl} x_0 & = & 0 \\ y_0 & = & 0 \\ x_{n+1} & = & x_n^2 - y_n^2 + a \\ y_{n+1} & = & 2x_n y_n + b \end{array}$$

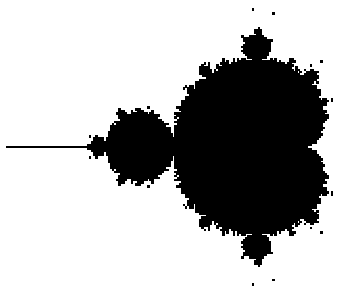


Figure 2.4: Mandelbrot set.

Although there is no exact method to determine whether this condition holds, it can be proven that one of these sequences must tend to infinity as soon as $x_n^2 + y_n^2 > 4$. This result allows us, firstly, to deduce that the points of the Mandelbrot set belong to a disc of radius 2 and center $(0, 0)$. Secondly, it enables us to draw an approximation of the Mandelbrot set, defined by the set of points (a, b) such that $x_n^2 + y_n^2 \leq 4$ for the first k terms of these sequences. Note that the precision of this approximation depends solely on the number of terms k.

The program begins with an `open Graphics` directive to simplify the use of the functions of the graphical library. It continues with the definition of two global variables `width` and `height`, containing respectively the width and height of the graphical window we wish to open (here, 800×800):

```
let width = 800
let height = 800
```

Next, we fix k, the maximum number of terms to be calculated (100 terms sufficing for a good approximation):

```
let k = 100
```

The following declaration defines a function `norm2` that takes two *arguments* x and y and returns the value of the expression $x^2 + y^2$.

```
let norm2 x y = x *. x +. y *. y
```

We discuss this syntactic construction in greater detail at the end of this section. For now, note that, as in the case of a function call, the names of the arguments are only separated by spaces. Note also that the *body* of the function, the expression to the right of the symbol `=`, may be arbitrary.

The following function, `mandelbrot`, determines whether a point with coordinates $(\mathtt{a}, \mathtt{b})$ belongs to the Mandelbrot set by calculating the first `k` terms of the recursive sequences (x_n) and (y_n). In a manner similar to the declaration of `norm2`, we declare the function `mandelbrot` with two arguments `a` and `b` as follows:

```
let mandelbrot a b = ...
```

To calculate the terms of the two recursive sequences, we define a local *recursive* function `mandel_rec` with three arguments `x`, `y`, and `i`.

```
    let rec mandel_rec x y i =
     ...
    in
```

Program 4 [mandelbrot.ml] — Drawing the Mandelbrot Fractal

```
open Graphics

let width = 800
let height = 800
let k = 100

let norm2 x y = x *. x +. y *. y

let mandelbrot a b =
  let rec mandel_rec x y i =
    if i = k || norm2 x y > 4. then i = k
    else
      let x' = x *. x -. y *. y +. a in
      let y' = 2. *. x *. y +. b in
      mandel_rec x' y' (i + 1)
  in
  mandel_rec 0. 0. 0

let draw () =
  for w = 0 to width - 1 do
    for h = 0 to height - 1 do
      let a = 4. *. float w /. float width -. 2. in
      let b = 4. *. float h /. float height -. 2. in
      if mandelbrot a b then plot w h
    done
  done

let () =
  let dim = Printf.sprintf " %dx%d" width height in
  open_graph dim;
  draw ();
  ignore (read_key ())
```

The first two arguments `x` and `y` of `mandel_rec` contain the i-th values x_i and y_i of the recursive sequences, and the third argument contains the index i.

Before we discuss the body of this function let us note that, syntactically speaking, a declaration of the form `let rec` is used to define a recursive function. This is a function in which you can use its own name within its body. Furthermore, as with local variable declarations, the scope of a *local* function, whether it is introduced by `let-rec-in` or simply `let-in`, is limited to the expression that follows the keyword `in`.

The body of the function `mandel_rec` begins with a test to determine whether we have reached the calculation of the k-th term of the sequence, or if the stopping condition $x_i^2 + y_i^2 > 4$ holds:

```
if i = k || norm2 x y > 4. then i = k
else
```

In both cases, the `mandel_rec` function ends and returns a boolean indicating whether or not the point is in the set. If neither of these conditions holds, we calculate x_{i+1} and y_{i+1} by applying the equations of the recursive sequences, and store them in the local variables `x'` and `y'`:

```
  let x' = x *. x -. y *. y +. a in
  let y' = 2. *. x *. y +. b in
```

We then call the function `mandel_rec` again to calculate the following terms:

```
  mandel_rec x' y' (i + 1)
in
```

Finally, the `mandelbrot` function starts the calculation by calling the local function `mandel_rec` with the initial values x_0 and y_0 as arguments:

```
in
mandel_rec 0. 0. 0
```

To draw the Mandelbrot set, it suffices to traverse each row and each column of the graphical window and to display the pixels $(\texttt{a}, \texttt{b})$ for which `mandelbrot a b` returns the value `true`.

The drawing code is contained in a function `draw` that is declared as follows:

```
let draw () = ...
```

A declaration of this form is used to define functions without arguments. To be precise, this declaration defines a function with *a single argument*, denoted by `()`. This is the unique value of a particular type, `unit`. As we will see in section 2.6 *Drawing a Curve*, this is an example of a function defined by *pattern matching*.

The function `draw` traverses the rows and columns of the graphical window using two nested `for` loops:

```
    for w = 0 to width - 1 do
      for h = 0 to height - 1 do
        ...
      done
    done
```

To draw the Mandelbrot set in a window of size `width` $\times$ `height`, we scale the window coordinates to points $(\mathtt{a}, \mathtt{b})$ in $[-2, 2] \times [-2, 2]$, where `a` and `b` are variables defined as follows:

```
        let a = 4. *. float w /. float width -. 2. in
        let b = 4. *. float h /. float height -. 2. in
```

All that remains is to determine whether the point $(\mathtt{a}, \mathtt{b})$ belongs to the set, by calling the function `mandelbrot` and plotting the point using the function `plot`:

```
        if mandelbrot a b then plot w h
      done
    done
```

To complete the program, we open a graphical window of size `width` $\times$ `height` by building a string of the form `" `*w*`x`*h*`"` where *w* and *h* are replaced by the values contained in the variables `width` and `height`.

```
  let () =
    let dim = Printf.sprintf " %dx%d" width height in
    open_graph dim;
```

The string is constructed by calling the function `Printf.sprintf` with a formatting string `" %dx%d"`, and the two arguments `width` and `heigth`. The function `Printf.sprintf` is similar to the function `printf`, except that it returns the string as result instead of displaying it.

Finally, we draw the Mandelbrot set by calling the function `draw`. We then wait until a key is pressed to end the program:

```
draw ();
ignore (read_key ())
```

Note that the same syntax is used to call the function `draw`, as is used in its declaration.

Function Declarations

Let us return to the notion of function in OCaml. A function is declared with the keyword `let`. Thus, a global function `f` that associates x with $x + 1$ is written simply as:

```
# let f x = x + 1;;
val f : int -> int = <fun>
```

As with variables, its type is inferred. More precisely, the type of its argument (here `int`) and that of its result (also `int`) are both inferred. The type of `f` has the form `int -> int`, where the type to the left of the arrow `->` is that of the argument and the type to the right is that of the result. However, the value of `f` is not displayed. Only `<fun>` is displayed, indicating that it is a function.

Functions may be local and follow the same scoping rules as variables.

```
# let sqr x = x * x in sqr 3 + sqr 4;;
- : int = 25
# (let sqr x = x * x in sqr 3) + sqr 4;;
Error: Unbound value sqr
```

By default, functions are not recursive.

```
# let fact x = if x = 0 then 1 else x * fact (x-1);;
Error: Unbound value fact
```

The construction `let rec` must be explicitly used to introduce a recursive function.

```
# let rec fact x = if x = 0 then 1 else x * fact (x-1);;
val fact : int -> int = <fun>
# fact 10;;
- : int = 3628800
```

Notice that in the previous example, we wrote `fact (x-1)` and not `fact x-1`. The second expression is understood as `(fact x) - 1` because function application is the operation with the highest precedence.

Mutually recursive functions must be defined *simultaneously* using the keyword `and`. For example, we can define two mutually recursive functions `f` and `g` as follows:

```
let rec f x = ... g ...
and     g x = ... f ...
```

A function may take several arguments. These are simply juxtaposed both in the definition and when it is called. For example, the following declaration defines a function `plus` with two arguments:

```
# let plus x y = x + y ;;
val plus : int -> int -> int = <fun>
# plus 3 4;;
- : int = 7
```

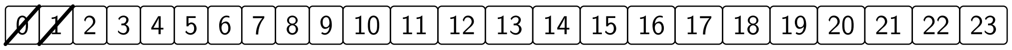
The type inferred by OCaml for this function is `int -> int -> int`. Adding parentheses yields the type `int -> (int -> int)` (the operator `->` is *right* associative), which is read as: "a function that takes a value of type `int` and returns a function of type `int` to `int` ". The function `plus` is therefore seen as a function with *one* argument that returns a function that awaits the *second* argument. Thus, strictly speaking, OCaml does not have functions that take multiple arguments. There are only *higher-order* functions with one argument, that is, functions that take a *single* argument and return another function. We will discuss such functions later on (section *2.6 Drawing a curve*).

2.5 Sieve of Eratosthenes

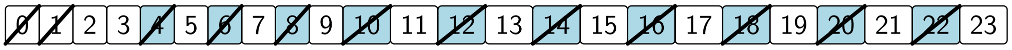
Ideas introduced

- arrays
- `while` loop
- `begin-end` block

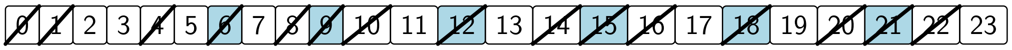
Our next program (see page 41) determines the primality of integers $n \leq N$ for a given integer N. To do this, it uses an algorithm called *the sieve of Eratosthenes*. Let us illustrate how it works for $N = 23$. We write down all the integers from 0 to N and progressively eliminate all the integers that are not prime. Hence the name *sieve*. We begin by noting that 0 and 1 are not prime.



Next, we determine the first integer that has not yet been eliminated, namely, 2. We eliminate all its multiples, that is, all even integers greater than 2. These are shaded below.



Then, we repeat the process. The next integer that has not yet been eliminated is 3. We eliminate all the multiples of 3, shown shaded below.



Note that certain integers have already been eliminated (the multiples of 6 in this case).

The next integer to be considered is 5. Since $5 \times 5 > 23$, the sieve is finished: All multiples of 5, that is, $k \times 5$, have either been eliminated (if $k < 5$) or are greater than 23 (if $k \geq 5$). The integers that remain are the prime numbers less than or equal to N: 2, 3, 5, 7, 11, 13, 17, 19 and 23.

Program 5 [sieve.ml] — Sieve of Eratosthenes

```
let max = read_int ()

let prime = Array.make (max + 1) true

let () =
  prime.(0) <- false;
  prime.(1) <- false;
  let limit = truncate (sqrt (float max)) in
  for n = 2 to limit do
    if prime.(n) then begin
      let m = ref (n * n) in
      while !m <= max do
        prime.(!m) <- false;
        m := !m + n
      done
    end
  done

let () =
  for n = 2 to max do
    if prime.(n) then Printf.printf "%d\n" n
  done
```

The program `sieve.ml` implements the sieve of Eratosthenes for a value of N entered on standard input. We store this value in a variable `max`.

```
let max = read_int ()
```

We then create an array of booleans `prime` of size `max+1`. To this end, we use the library function `Array.make`. It takes as arguments the size of the array and a default value for its elements, here `true`.

```
let prime = Array.make (max + 1) true
```

Recall that arrays are indexed starting from 0. The indices of the array `prime` are therefore the integers from 0 to `max`, both included. We begin by indicating that the integers 0 and 1 are not primes, by assigning the value `false` to the two corresponding elements of the array `prime`.

```
let () =
  prime.(0) <- false;
  prime.(1) <- false;
```

For this, we have used the construction t`.(`i`) <-` v, which assigns the value v to the cell with index i in the array t. This is a special syntax for the library function `Array.set`. We could also have written `Array.set prime 0 false`.

We continue the program by specifying the largest number to be considered. This is $\lfloor\sqrt{\texttt{max}}\rfloor$, which can be calculated thus:

```
  let limit = truncate (sqrt (float max)) in
```

The main loop of the sieve then iterates over the integers from 2 to `limit`, testing whether each is prime. The loop therefore has the following structure:

```
  for n = 2 to limit do
    if prime.(n) then begin
      ...
    end
```

The construction `begin-end` introduces a *block*, that is, a piece of program that is delimited, indicated here by the ellipses. In OCaml, there is no distinction between expressions and instructions; there are only expressions. The delimited piece of code is thus nothing but an expression; it could have been delimited by parentheses. Nevertheless, the use of begin-end highlights the imperative character of this expression.

Let us now discuss this block. It consists of a second loop that eliminates all multiples of n in the array prime. The same reasoning that allows us to stop the sieve as soon as $\texttt{n} \times \texttt{n} > N$ also allows us to begin the elimination from $\texttt{n} \times \texttt{n}$ (rather than $2\texttt{n}$), the smaller multiples having already been eliminated. To iterate over all the integer multiples of n from $\texttt{n}^2$ onwards, we initialize a reference m to the starting value:

```
    let m = ref (n * n) in
```

Then, we use a loop that assigns false to prime.(!m) and increases m by n, as long as the expression !m <= max holds. Such a loop is written using a while construct, as follows:

```
    while !m <= max do
      prime.(!m) <- false;
      m := !m + n
    done
```

Note that, as in case of the for loop, the body of the while loop is delimited by the keywords do and done.

This completes the sieve. The values of the array prime now indicate which numbers less than or equal to max are prime. We can display these prime numbers using another loop.

```
let () =
  for n = 2 to max do
    if prime.(n) then Printf.printf "%d\n" n
  done
```

We could instead have displayed the prime numbers upon finding them, that is, while executing the sieve. However, the sieve's loop stops when $\texttt{n} = \lfloor\sqrt{\texttt{max}}\rfloor$, so that another loop would have been necessary anyway to display the prime numbers between $\lfloor\sqrt{\texttt{max}}\rfloor + 1$ and max.

Additional Information

We return here to the constructions `while` and `begin-end` introduced in this section, and then discuss arrays.

The `while` Loop

In general, the syntax of the `while` loop is:

```
while e1 do
 e2
done
```

The expression e_1 must be of type `bool` and the expression e_2 of type `unit`.

The `begin-end` Block

The construction `begin-end` is equivalent to a pair of parentheses. Thus, you can write:

```
# 2 * begin 1 + 2 end;;
- : int = 6
```

Nevertheless, the convention is to limit its use to expressions that perform side effects, by analogy with the notion of block in languages such as C or Java.

Arrays

An array can be constructed explicitly by specifying the values that make it up.

```
# let t = [| 12; 32; 3; 8 |];;
val t : int array = [|12; 32; 3; 8|]
```

To allocate an array of an arbitrary and, in particular, statically unknown size, we use the library function `Array.make`. It takes as arguments the size of the array and a value used to initialize all its cells. (As with any OCaml value, an array must be initialized.)

```
# let u = Array.make 1024 'a';;
val u : char array = [|'a'; 'a'; 'a'; ... |]
```

The size of an array can be obtained using the library function `Array.length`.

```
# Array.length u;;
- : int = 1024
```

Note finally that accessing an array with an index that lies outside the range of valid indices produces a runtime error, signaled by an exception:

```
# u.(4012);;
Exception: Invalid_argument "index out of bounds".
```

Exceptions will be discussed in detail later on.

Matrices

There is no predefined type for matrices nor, more generally, for multidimensional arrays. We simply use arrays of arrays. Consider, for instance, the matrix:

$$M = \begin{pmatrix} 1 & 0 \\ 2 & 3 \end{pmatrix}$$

It is represented by the expression:

```
# let m = [| [| 1; 0 |];
             [| 2; 3 |] |];;
  val m : int array array = [|[|1; 0|]; [|2; 3|]|]
```

The element $M_{i,j}$ is accessed with `m.(i).(j)`, which is nothing more than two successive accesses: First, we access row `i` with `m.(i)`, and then, its element at index `j`.

```
# m.(1).(0);;
- : int = 2
```

All operations on matrices are implemented using those of arrays. The module `Array` nonetheless offers an operation to create a matrix, `make_matrix`, which takes as arguments the dimensions of the matrix and the initial value of its elements. We will return to matrices in the section *10.4 Matrix Calculus* of chapter 10.

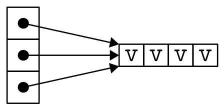
Aliasing

Imperative structures must be handled with caution. In particular, whenever it is possible to reference an imperative structure in multiple ways, you may think you are modifying different data even though this need not be the case.

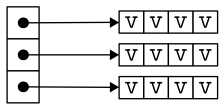
A typical example of aliasing is obtained by allocating an array with all its elements initialized to the same array. Consider the following expression:

```
Array.make 3 (Array.make 4 v)
```

You may think you are constructing a 3×4 matrix, each of whose elements is `v`. In reality, this expression constructs a *single* array of size 4—let us call it a—whose elements have the value `v`, and an array of size 3, whose three elements are the *same* array a. In other words, the result is as follows:



Of course, this is not what we want, as it has only four elements instead of twelve. To construct an array of arrays correctly, you can use the function `Array.make_matrix` of the standard library, by writing `Array.make_matrix 3 4 v`, which has the effect of constructing the desired matrix, that is:



2.6 Drawing a Curve

Ideas introduced

- pairs, n-tuples
- evaluation order
- first-class, higher order, anonymous functions
- definition by pattern matching, wildcard pattern

Our sixth program (see page 50), `plot.ml`, draws a curve specified by a set of points in the plane. The algorithm consists in joining these points with line segments, proceeding in increasing order of their x-coordinates. For example, the curve drawn for the set of points $\{(20,15),(2,2),(30,10),(10,15)\}$ is shown in figure 2.5.

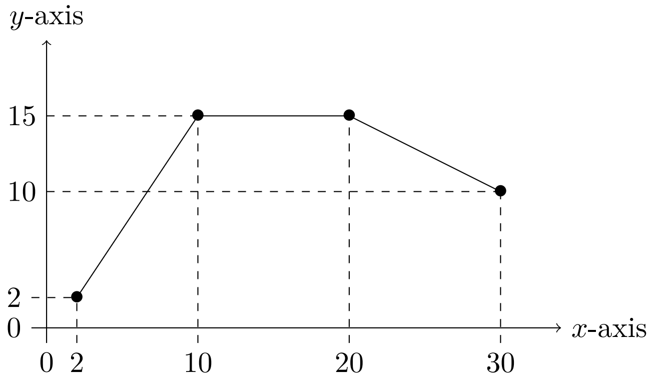


Figure 2.5: Drawing a curve.

To draw this curve, we begin by reading from standard input an integer that indicates the number of points in the set.

```
let n = read_int ()
```

Next, we define a function that reads the coordinates of a point from standard input:

```
let read_pair () =
  let x = read_int () in
  let y = read_int () in
  (x, y)
```

After reading the first integer, `x`, and the second, `y`, from standard input using two calls to `read_int`, the function `read_pair` returns the pair of integers (`x`, `y`), constructed very naturally using the syntax `(x,y)`. The type of this pair is written as `int * int` in OCaml. We will discuss pairs and, more generally, n-tuples in greater detail at the end of this section.

We then create an array `data` of size `n` that we initialize with the pairs of integers read from standard input. To do this, we use the library function `Array.init` that takes as arguments the size of the array and a function used to initialize the array's cells:

```
let data = Array.init n (fun i -> read_pair ())
```

The function call `Array.init` illustrates two new notions. First, the expression `fun i -> ...` passed as an argument to `Array.init` is an *anonymous* function. The symbol `i` to the left of the arrow `->` represents the argument of the function, the expression to the right, its body. Second, this call shows that functions in OCaml are values like any other. We also say they are *first-class* values. They can be passed as arguments and, as we will see later, returned as the results of other functions. A function like `Array.init`, which takes a function as an argument, is called a *higher-order* function.

The function `Array.init` returns an array obtained by initializing the cell at index k (ranging from 0 to $\mathtt{n} - 1$) with the value obtained by applying the function `(fun i -> read_pair ())` to k.

Thus, if the integers entered on standard input consist of the sequence `4 20 15 2 2 30 10 10 15`, the preceding declarations define a variable `n` containing the integer `4` and an array `data` whose contents are graphically represented as follows:

(20,15)	(2,2)	(30,10)	(10,15)

To draw the curve, we must begin by sorting the cells of this array in increasing order of x-coordinates. To this end, we begin by defining a function that compares two pairs by their first component:

```
let compare (x1, y1) (x2, y2) = x1 - x2
```

This function returns an integer. Its sign is interpreted by the sorting algorithm as follows:

- if `x1 - x2` $= 0$, the two pairs are considered to be equal;
- if `x1 - x2` > 0, the first pair is considered to be greater;
- if `x1 - x2` < 0, the first pair is considered to be smaller.

Before continuing with the description of our program, let us stop an instant and consider the form of this declaration.

The arguments of the function `compare` are not simple identifiers, but rather *patterns*. They each have the form of a pair $(\mathtt{x}_i, \mathtt{y}_i)$. The use of patterns allows us here to easily retrieve the components $\mathtt{x}_i$ and $\mathtt{y}_i$ of the pairs passed as arguments of the function. This is a declaration by *pattern matching*. This form of declaration is very frequently used in OCaml programs, and we will use it on numerous occasions in this book.

To sort the array, we use the higher-order function `Array.sort` which, in addition to the array to be sorted, takes as its first argument a function to compare the elements of the array. This sorting is done in-place, that is, directly within the array `data`. The call to `Array.sort` only performs side effects.

```
let () = Array.sort compare data
```

At the end of this call, the contents of `data` are as follows:

(2,2)	(10,15)	(20,15)	(30,10)

All that remains is to draw the curve by joining the points thus ordered, with line segments. We begin with a directive `open Graphics`. Then, we open a graphical window of size 200 × 200. The call `set_line_width 3` fixes the thickness of the line segments.

Program 6 [plot.ml] — Drawing a Curve

```
let n = read_int ()

let read_pair () =
  let x = read_int () in
  let y = read_int () in
  (x, y)

let data = Array.init n (fun i -> read_pair ())

let compare (x1, y1) (x2, y2) = x1 - x2
let () = Array.sort compare data

open Graphics
let () =
  open_graph " 200x200";
  set_line_width 3;
  let (x0,y0) = data.(0) in moveto x0 y0;
  for i = 1 to n-1 do
    let (x,y) = data.(i) in
    lineto x y
  done;
  ignore (read_key ())
```

```
open Graphics
let () =
 open_graph " 200x200";
 set_line_width 3;
```

We then retrieve the first pair `(x0, y0)` from the `0`-th cell of the array `data`, and use it to fix the coordinates of the current point using the function `moveto`:

```
let (x0,y0) = data.(0) in moveto x0 y0;
```

Here again, we use a declaration by pattern matching to extract the components of a pair stored in the array. The value contained in `data.(0)` is matched by means of the pattern `(x0, y0)`, which defines two local variables `x0` and `y0`.

We then iterate over all the elements of the array `data` with the help of a `for` loop to join the points:

```
for i = 1 to n-1 do
  let (x,y) = data.(i) in
  lineto x y
done;
```

The body of the loop retrieves the coordinates `(x,y)` of each point using a local declaration by pattern matching (identical to the one above) and draws a line from the current point to the point with coordinates `(x,y)`.

Additional Information

Pairs and n-Tuples

A pair of two values `v1` and `v2` can be constructed using the traditional notation `(v1, v2)`. The type of pairs uses the mathematical notation of the Cartesian product.

```
# (1, true);;
- : int * bool = (1, true)
```

You will note that the comma is used to construct *the pair* while the symbol `*` is used to construct *the type* of the pair. The two projections are written `fst` and `snd` respectively.

```
# fst (1 + 2, true);;
- : int = 3
# snd (1, not true);;
- : bool = false
```

Pairs may be arbitrarily nested.

```
# ((1, true), (3.14, false));;
- : (int * bool) * (float * bool) = ((1, true), (3.14, false))
```

Pairs are a special case of n-tuples, and use the same syntax for both values and types.

```
# (1, true, 3.14);;
- : int * bool * float = (1, true, 3.14)
```

It should be noted that the type `(int * int) * int`, the type `int * (int * int)`, and the type `int * int * int` are not the same:

- the first is that of a pair whose first component is a pair;
- the second is that of a pair whose second component is a pair;
- the third is that of a triple.

Projections by Pattern Matching and Wildcard Pattern

There are no projection functions for n-tuples. A declaration by pattern matching must be used to retrieve each component. For example:

```
# let (x, (y, z), t) = (1, (true, "hello"), 3.4);;
val x : int = 1
val y : bool = true
val z : string = "hello"
val t : float = 3.4
```

If you only wish to access certain components of an n-tuple, you may use a *wildcard pattern*, which is written as `_`, to avoid having to introduce unnecessary variable names. For example, if you wish only to access the components `x` and `z` of the above triple, you can write the following pattern matching:

```
# let (x, (_, z), _) = (1, (true, "hello"), 3.4);;
val x : int = 1
val z : string = "hello"
```

Wildcard patterns also allow you to represent arbitrary values. Thus, if you only wish to access components `x` and `t`, you may use a wildcard pattern to represent the second component of the above triple:

```
# let (x, _, t) = (1, (true, "hello"), 3.4);;
val x : int = 1
val t : float = 3.4
```

Functions with Several Arguments and n-Tuples

It is equally important to differentiate clearly between a function with several arguments and a function that takes a single argument which is an n-tuple. Consider, for instance, the following function `pyth`:

```
# let pyth (x, y, z) = x*x + y*y = z*z;;
val pyth : int * int * int -> bool = <fun>
```

It takes a triple of integers as argument. You therefore call it using a *single* argument of the form `(`e_1`,` e_2`,` e_3`)`, as in the following call:

```
# pyth (3, 4, 5);;
- : bool = true
```

On the other hand, the same function may be written as a function of several arguments, as follows:

```
# let pyth x y z = x*x + y*y = z*z;;
val pyth : int -> int -> int -> bool = <fun>
```

This function must be called by juxtaposing its name with *three* arguments, as in the following call:

```
# pyth 3 4 5;;
- : bool = true
```

The types `int * int * int -> bool` and `int -> int -> int -> bool` illustrate clearly the difference between these two variants. In this text, we adopt the *Curry* style, wherein functions take n arguments rather than an n-tuple. In particular, this enables the *partial* application of functions: You can pass the function fewer arguments than it expects and obtain another function in return. For example, by applying the function `pyth` to an integer, you obtain a function that takes two arguments:

```
# let f = pyth 3;;
val f : int -> int -> bool = <fun>
```

You can then apply it several times:

```
# f 4 5;;
- : bool = true
# f 6 7;;
- : bool = false
```

Records

Tuples serve to group several values together. However, it is easy to confuse the different components of an n-tuple when they have the same type. Thus, a date represented by a triple of integers, of type `int * int * int`, does not indicate clearly which integer designates the day, the month, and the year. To correct this defect, the OCaml language provides the notion of *record*, that is, an n-tuple whose fields are named. To create a record, you must declare a new type beforehand, describing the names and the types of the different fields.

```
# type date = { day : int; month : int; year : int };;
type date = { day : int; month : int; year : int; }
```

We can then construct a value of the type `date` with the following syntax.

```
# let valentine's_day = { day = 14; month = 2; year = 2014 };;
val valentine's_day : date = { day = 14; month = 2; year = 2014 }
```

We access a field f of a record e using the notation $e.f$.

```
# valentine's_day.month;;
- : int = 2
```

A new record can be constructed based on another record of the same type, preserving the values of certain fields and giving new values to others. We use the construction `with` for this.

```
# let d = { valentine's_day with day = 15 };;
val d : date = {day = 15; month = 2; year = 2014}
```

This is identical to the following declaration:

```
# let d = { day = 15;
            month = valentine's_day.month;
            year = valentine's_day.year };;
val d : date = {day = 15; month = 2; year = 2014}
```

Another advantage of records over n-tuples is that the order of fields is not significant. We can therefore write:

```
# let us_valentine's_day = { month = 2; day = 14; year = 2014; };;
val us_valentine's_day : date = {day = 14; month = 2; year = 2014}
# us_valentine's_day = valentine's_day;;
- : bool = true
```

It is also possible to use a pattern to retrieve the values of each field, once again in any order.

```
# let { day; month; year } = valentine's_day;;
val day : int = 14
val month : int = 2
val year : int = 2014
```

We can omit certain fields by using the wildcard pattern _ to indicate the fields to be ignored.

```
# let { day; month; _ } = valentine's_day;;
```

Tuples are nevertheless useful, particularly because they do not require us to declare a type.

Records with Mutable Fields

The keyword mutable declares a record field that can be modified.

```
# type student = { number : int; mutable age : int };;
type student = { number : int; mutable age : int; }
```

A record field is modified using the construction <-.

```
# let birthday e = e.age <- e.age + 1;;
val birthday : student -> unit = <fun>
```

This modification does not return a value. Hence its type is unit. We observe this side effect in the following example:

```
# let e = { number = 123456; age = 21 };;
val e : student = {number = 123456; age = 21}
# birthday e;;
- : unit = ()
# e;;
- : student = {number = 123456; age = 22}
```

We can use a record with a mutable field to simulate the traditional notion of a mutable variable. For example, in case of a variable of type int, it suffices to declare a type with a single mutable field value.

```
# type variable = { mutable value : int };;
type variable = { mutable value : int; }
```

We then use x.value to access the contents of the variable x and the assignment x.value <- e to modify it.

```
# let x = { value = 41 };;
val x : variable = {value = 41}
# x.value <- x.value + 1;;
- : unit = ()
# x.value;;
- : int = 42
```

Strictly speaking, we are not modifying the variable x. We modify only the value contained in the record that x points to. So, more precisely, it is the notion of pointer that is being used here.

To avoid having to define such a type each time we need a mutable variable, OCaml offers the predefined type of *references*, that we have already used (see section 2.2 *Approximation of* π). A reference is created using the keyword `ref`.

```
# let x = ref 41;;
val x : int ref = {contents = 41}
```

As is clear, the type of a reference, written as `int ref`, is that of a record, with a single field `contents`. We access this field using the notation `!` and modify its contents using the notation `:=`.

```
# x := !x + 1;;
- : unit = ()
# !x;;
- : int = 42
```

Evaluation Order

The OCaml language does not specify the evaluation order of the components of n-tuples and records. Thus, in the pair `(e1, e2)`, it is not possible to know *a priori* if the expression `e1` will be evaluated before `e2` or the other way around.

Of course, we can easily figure out the evaluation order of a specific implementation of OCaml using an expression that creates a pair, as follows:

```
# (read_int (), read_int ());;
4
5
- : int * int = (5, 4)
```

We can see here that it is the expression `e2` that is evaluated first, since the first integer typed in, here `4`, is stored in the second component of the pair.

In general, it is not a good idea to write a program that relies on the evaluation order implemented by a specific compiler. It is better to use local declarations to specify the evaluation order. Thus, what we ought to write is:

```
# let x = read_int () in let y = read_int () in (x, y);;
4
5
- : int * int = (4, 5)
```

Anonymous Functions

The definitions of anonymous functions are not limited to one argument. Functions that take several arguments may be defined as follows:

```
# fun x y -> x + y;;
- : int -> int -> int = <fun>
```

It is interesting to note that the following two declarations are strictly equivalent:

```
# let f = fun x -> x + 1;;
val f : int -> int = <fun>
# let f x = x + 1;;
val f : int -> int = <fun>
```

2.7 Copying a File

Ideas introduced

- input/output
- exceptions

Our next program, `copy_file.ml`, copies the contents of one file into another, with the names of the two files passed on the command line.

For this, we write a function `copy_file` that takes as arguments the names of the files, as strings `f1` and `f2`. We begin by opening channels `c1` and `c2` on the files `f1` and `f2`, respectively.

```
let copy_file f1 f2 =
  let c1 = open_in f1 in
  let c2 = open_out f2 in
```

The first, `c1`, is opened for reading and the second, `c2`, for writing.

Then, we execute an infinite loop, (`while true`), which reads characters from `c1` and writes them to `c2`.

```
    while true do output_char c2 (input_char c1) done
```

When there are no more characters left to read, the function `input_char` raises a predefined *exception*, namely, `End_of_file`. This interrupts the execution of the `while` loop and, more generally, of the program being executed, until the exception is *caught*. Here, we catch this exception just outside the `while` loop. We do this with the construction `try-with`.

```
let () =
  try
    while true do ... done
  with End_of_file ->
    close_in c1; close_out c2
```

The meaning of the construction `try` e_1 `with E ->` e_2 is as follows: We begin by evaluating the expression e_1. If a value is obtained, it is the value of the entire `try-with` expression. If, however, the evaluation of e_1 raises the exception `E`, then we evaluate the expression e_2, and that is the result of the expression as a whole. Finally, if e_1 raises an exception other than `E`, then the exception is not caught, and it is the expression `try-with` as a whole that raises it.

In our case, the expression e_1 is an infinite loop, whose evaluation will never end. However, it ends by raising the exception `End_of_file`, which is then caught, and the two files `c1` and `c2` are closed.

All that remains then is to call the function `copy_file` with the first two arguments of the command line, namely, `Sys.argv.(1)` and `Sys.argv.(2)`.

Program 7 [copy_file.ml] — Copying a file

```
let copy_file f1 f2 =
  let c1 = open_in f1 in
  let c2 = open_out f2 in
  try
    while true do output_char c2 (input_char c1) done
  with End_of_file ->
    close_in c1; close_out c2

let () = copy_file Sys.argv.(1) Sys.argv.(2)
```

Additional Information

Input/Output Channels

Input/output devices are represented by *channels*. Input channels are of the predefined type `in_channel`, and output channels of the type `out_channel`. Three predefined values represent, respectively, standard input, standard output, and standard error of the program being executed: `stdin` of type `in_channel`, and `stdout` and `stderr` of type `out_channel`.

Functions are available for reading and writing on channels. The function `input_char`, of type `in_channel -> char`, reads a single character from the channel passed as argument. We can thus read a character from standard input, `stdin`, in the following manner:

```
# let c = input_char stdin;;
a
val c : char = 'a'
```

A more general function, `input`, can be used to read n characters from an input channel and store them at a certain position within a string.

Similarly, the `output_char` function, of type `out_channel -> char -> unit`, writes a character to an output channel. A more general function, `output`, writes a given substring. There is also a function `Printf.fprintf` that writes to an output channel, in a manner analogous to the function `Printf.printf` that we used earlier.

Files can also be manipulated as channels. Thus, a file is opened for reading with the function `open_in`, of type `string -> in_channel`, and for writing with the function `open_out`, of type `string -> out_channel`. The string designates the file name. Symmetrically, we close a channel with the functions `close_in` and `close_out`. Input/output operations on channels can fail (reading or writing on a closed channel, invalid permissions, etc.). This is systematically signalled by raising an exception. For instance, reading beyond the end of a file raises the predefined exception `End_of_file`.

Marshalling

Besides characters, it is possible to use channels to read and write OCaml values of arbitrary types. This is known as *marshalling*. The function `output_value` writes a value of an arbitrary type to an output channel.

```
# let c = open_out "foo" in
  output_value c (1, 3.14, true);
  close_out c;;
- : unit = ()
```

The format used is specific to OCaml (and even to its version). The function that performs the inverse operation is `input_value`. When you read a marshalled value with the function `input_value`, its type is *inferred* according to how it is used. It is good practice to indicate the type of the value that is read with a type annotation. Below, we read back the value written to the file `"foo"`, specifying that its type is `int * float * bool`:

```
# let c = open_in "foo";;
val c : in_channel = <abstr>
# let v : int * float * bool = input_value c;;
val v : int * float * bool = (1, 3.14, true)
```

There is no type information in the marshalled value itself. In particular, runtime type safety is not guaranteed if we use a marshalled value in a way that is incompatible with the type of the value that was actually marshalled. In the above example, we could pretend that the value read is a pair whose first component is also a pair. This leads to a fatal program error.

```
# let c = open_in "foo";;
val c : in_channel = <abstr>
# let v = input_value c in fst (fst v);;

Process caml-toplevel segmentation fault
```

This error is not an exception that we could have caught. It is, rather, a lower-level error corresponding to an unrecoverable illegal memory access.

Exceptions

Certain operations are partial, that is, they are not defined for all possible values of their arguments. For instance, integer division by zero is not defined. If you try, nevertheless, to perform this operation, an *exception is raised.*

```
# 1/0;;
Exception: Division_by_zero.
```

The evaluation of the expression does not result in a value; it fails when the exception is raised, as the message beginning with `Exception` indicates. Here, `Division_by_zero` is a predefined exception. Another predefined exception in OCaml, `Invalid_argument`, is often used to signal the use of a function outside its domain of definition.

```
# Random.int (-4);;
Exception: Invalid_argument "Random.int".
# Char.chr 257;;
Exception: Invalid_argument "Char.chr".
```

As is clear from these examples, a string is associated with the exception `Invalid_argument`, which specifies the name of the function that raised the exception.

Raising an exception *interrupts* the calculation, as we can see by evaluating the following expression.

```
# print_endline "before"; print_int (1/0); print_endline "after";;
before
Exception: Division_by_zero.
```

It is, nonetheless, possible to catch an exception, so as to continue the evaluation with another expression. You use the construction `try with` for this, as follows:

```
# let test x y =
    try let q = x / y in Printf.printf "quotient = %d\n" q
    with Division_by_zero -> Printf.printf "error\n";;
val test : int -> int -> unit = <fun>
# test 4 0;;
error
- : unit = ()
```

Here, we attempt to calculate `x / y` and display its value. In case of failure, that is, when the division triggers the exception `Division_by_zero`, we catch the exception and display an error message. The evaluation of the function `test` therefore always ends with a value of type `unit`.

In general, the expression `try` e_1 `with E ->` e_2 is evaluated as follows:

- The expression e_1 is evaluated first. If it does not raise an exception, the calculation is complete, and its value is that of e_1.
- If, however, the expression raises an exception, then there are two possibilities:
  - if the exception is E, then evaluation continues with e_2;
  - if the exception is something other than E, then it is *propagated*, without evaluating e_2.

In the expression `try` e_1 `with E ->` e_2, the two sub-expressions e_1 and e_2 must have the same type, which is also the type of the expression as a whole.

Users may define their own exceptions with the declaration `exception` followed by the exception's name, which must begin with a capital letter. The exception need not take any arguments, for example, `Division_by_zero`:

```
# exception Stop;;
exception Stop
```

Alternatively, the exception may have one or more arguments, for example, `Invalid_argument`:

```
# exception Error of string;;
exception Error of string
```

To raise an exception, you use the construction `raise`. This construction takes an exception as argument.

```
# let f x =
    if x < 0 then raise (Error "negative argument");
    123 mod x;;
val f : int -> int = <fun>
```

Here, the exception is constructed by applying the constructor `Error` to the string `"negative argument"`. You can observe the exception being raised by calling `f` with a negative argument.

```
# f (-1);;
Exception: Error "negative argument".
```

Exceptions are, in fact, values like any other and belong to the predefined type `exn`. This type can be seen as a concrete type with an unbounded number of constructors. Predefined exceptions are merely predefined constructors of this type. Each `exception` declaration adds a new constructor. You can construct a value of type `exn` and pass it as an argument to `raise`:

```
# let e = Error "invalid argument";;
val e : exn = Error "invalid argument"
# raise e;;
Exception: Error "invalid argument".
```

The construction `raise` is simply a function that takes an argument of type `exn`. The expression `raise` *e* can take any type. In the previous example, the expression `raise (Error "invalid argument")` takes the type `unit`, but can equally take the type `int` if we wish to write the function `f` with an `else`:

```
# let f x =
    if x < 0 then raise (Error "invalid argument") else 123 mod x;;
val f : int -> int = <fun>
```

Exceptions can be used to signal exceptional behaviors (as in the preceding examples), as well as to change the program's control flow. A typical example is that of an infinite loop that you exit using an exception. For example, this is the case of interactive programs that you exit by pressing a key. With the library `Graphics`, such a loop can be written as follows:

```
try
  while true do
    let st = wait_next_event [Key_pressed] in
    if st.keypressed && st.key = 'q' then raise Exit;
    ...
  done
with Exit ->
  close_graph ();
  ...
```

Raising the exception `Exit` interrupts the infinite loop `while true`. We catch the exception, close the graphical window with `close_graph`, and then continue the program. Other examples of the use of exceptions to change the flow of control are given in exercises 2.15 and 5.7.

Finally, note that the OCaml standard library provides two functions, `failwith` and `invalid_arg`. When called with a string `s`, these raise the exceptions `Failure s` and `Invalid_argument s`, respectively. OCaml also provides a specific construction `assert e`, which evaluates an expression `e` of type `bool` and raises the exception `Assert_failure` if `e` is `false`.

2.8 Reversing the Order of Lines in a Text

Ideas introduced

- lists
- pattern matching
- call stack, tail call

Our next program, `tac.ml`, reads lines of text from standard input and then displays them in reverse order. It is easy to use. Having compiled the program, we execute it and enter, for example, the following three lines on standard input:

```
first line
second line
third line
```

Once we signal the end of input (for example, by pressing the keys `ctrl` and `D`), the program displays the lines in reverse order:

```
third line
second line
first line
```

To achieve this, the program must store all the lines read before it is able to display them, since the first line to be displayed is the last one to be read. We therefore need a data structure to store the lines that have been read. An array is not suitable since we do not know the total number of lines[2]. We will therefore use a *list*.

OCaml provides a built-in type for lists. These are constructed from the empty list, represented by `[]`, and by adding an element x at the front of an existing list l, represented as x `::` l. Lists in OCaml are *immutable*: once a list is constructed, it is no longer possible to modify it.

[2]A resizeable array could also be used here. This notion will be presented in chapter 4.

Program 8 [tac.ml] — Reversing the Order of Lines in a Text

```
let lines = ref []

let () =
  try
    while true do lines := read_line () :: !lines done
  with End_of_file ->
    ()

let rec print l =
  match l with
  | []     -> ()
  | s :: r -> print_endline s; print r

let () = print !lines
```

Our program begins by introducing a reference, `lines`, that will contain the list of lines that have been read. Initially, this list is empty.

```
let lines = ref []
```

We then proceed to read the lines, using the library function `read_line`, which reads a line of text from standard input and returns it as a string. To read all the lines, we write an infinite loop:

```
    while true do lines := read_line () :: !lines done
```

Each line read using `read_line` is added to the list contained in the reference `lines`. More precisely, we construct a new list whose first element is the most recently read line, followed by the elements already present in `lines`, that is, `!lines`. Next, we set the value of the reference `lines` to this new list. When there are no more lines to be read, the function `read_line` raises the exception `End_of_file`.

We catch this exception just outside the loop:

```
    try  while true do ... done
    with End_of_file -> ()
```

With this, we are done with the reading. The reference `lines` now contains the list of all the lines read. By construction, this list is in reverse order, the most recently read line having been added at the head of the list. This is convenient since this is precisely the order in which we wish to display the lines. To display the lines, we begin by writing a recursive function `print`, which displays a list `l` of strings, one per line.

```
  let rec print l =
```

This function examines the list `l`, treating the case of an empty list and that of a list containing at least one element separately. We discriminate between the two cases using the OCaml construct `match-with`, as follows:

```
    match l with
    | []     -> ...case 1...
    | s :: r -> ...case 2...
```

This pattern matching construct is read as follows: If the list `l` is of the form `[]`, that is, if it is the empty list, then we evaluate the code denoted above by `...case 1...`. If, however, the list `l` is of the form `s :: r`, that is, if it contains a first element `s` and is followed by other elements forming a list `r`, then we evaluate the code denoted above by `...case 2...`. Here, the variables `s` and `r` take the value of the first element and the rest of the elements of `l`. In the function `print`, we do nothing in the first case. In the second case, we display the string `s` and then call `print` recursively on the list `r`. Therefore, the body of the function `print` is:

```
    match l with
    | []     -> ()
    | s :: r -> print_endline s; print r
```

All that remains to complete our program is to apply this function to the list contained in the reference `lines`, that is:

```
  let () = print !lines
```

Additional Information

Lists

The type of lists is predefined in OCaml; it is `list`. This type is “generic,” in the sense that we can construct lists of values of any type at all, provided that all the elements of a given list are of the same type.

```
# 1 :: 2 :: 3 :: [];;
- : int list = [1; 2; 3]
```

```
# 1 :: 'a' :: 3 :: [];;
Error: This expression has type char but an expression
       was expected of type int
```

As we see in the first example, the toplevel displays a list in the form `[1; 2; 3]`. This notation can also be used to construct a list given its elements.

```
# ['a'; 'b'; 'c'];;
- : char list = ['a'; 'b'; 'c']
```

Pattern Matching

When we presented n-tuples, we explained that pairs can be destructured using the construction `let (x,y) =` e_1 `in` e_2. In fact, that pattern matching is identical in every way to the one we discussed above in relation to lists. The construction `let` is merely a shorthand for the expression `match` e_1 `with (x,y) ->` e_2. In general, the construct `let` may be used with an arbitrary pattern as its first argument, even if the latter is often reduced to a variable.

The function `print` proceeds by pattern matching on its argument `l`. Such functions are so common in OCaml that there is a keyword, `function`, which introduces a function that proceeds by pattern matching on its argument. Thus, we may write:

```
let rec print = function
  | []     -> ()
  | s :: r -> print_endline s; print r
```

In other words, the keyword `function` is a shorthand for `fun x -> match x with`. Note also that `function` introduces a function with only *one* argument, where the pattern matching may have one or more patterns. By contrast, `fun` defines a function with several arguments, where each argument is matched by *a single* pattern. Thus, we may write:

```
fun x (y, z) -> x + y * z
```

List Traversals

The function `print` that we wrote to display all the elements of the list `!lines` is an example of a list *traversal*, where all the elements of the list are treated the same way, in the order in which they appear. Such traversals are so frequent that a library function exists for them, namely, `List.iter`. This is a higher-order function, like `Array.init` or `Array.sort`, which we presented earlier. `List.iter` takes as argument the function to be applied to each element. Thus, we can rewrite `print` quite simply as:

```
let print l = List.iter print_endline l
```

We can even replace the last five lines by:

```
let () = List.iter print_endline !lines
```

Call Stack

Suppose we want our program to display the total number of lines that have been read and reversed, in a message of the form:

```
113 lines read
```

For this, we write a function `length` that calculates the length of a list. As with `print`, we write it in the form of a recursive function that proceeds by pattern matching:

```
let rec length = function
  | []     -> 0
  | _ :: r -> 1 + length r
```

Note the wildcard pattern _ used in place of the first element of the list in the second pattern. This value does not need to be named because it is not used in the calculation of the length of the list. All that remains now is to display the length thus calculated:

```
let () = Printf.printf "%d lines read\n" (length !lines)
```

If we run the program several times, we would observe that it does indeed give the expected result. However, if we were to attempt feeding the program a very large number of lines—say, 700,000 lines—we would be in for a nasty surprise: The program would give us an error[3] even though everything worked just fine up until the point where we tried to display the number of lines:

```
seq 700000 | ./tac
Fatal error: exception Stack_overflow
```

To explain this phenomenon, it is necessary to understand the principle behind the execution of a recursive function like `length`.

Consider as an example the execution of the call `length [1; 2; 3]`. We can "symbolically" unroll this call as follows:

```
length [1; 2; 3] = 1 + length [2; 3]
                 = 1 + (1 + length [3])
                 = 1 + (1 + (1 + length []))
                 = 1 + (1 + (1 + 0))
                 = 1 + (1 + 1)
                 = 1 + 2
                 = 3
```

Each call to `length` allocates a memory cell to store the function's argument. In general, a function call, whether recursive or not, allocates the memory space necessary for its arguments and its local variables. This memory space is freed once the function call ends. In case of the recursive function `length`, the call does not end until the recursive call completes and the addition `1 + ...` is performed. In particular, the call `length [1; 2; 3]` will require up to four

[3]The program `seq` is a Unix tool that displays all the integers from 1 to n, one integer per line. It is the simplest way to obtain exactly n lines of text.

nested function calls to reach the final call `length []`. Thus, four memory cells will be needed to store the arguments of each of the four calls, as represented in figure 2.6.

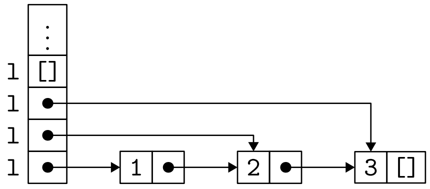


Figure 2.6: Representation of a call stack.

As is clear in the diagram, the nesting of function calls allows us to organize the memory required for them as a *stack*. Hence the term *call stack*. Each call to a function pushes the memory space required for its variables (here, the single variable `l`) onto the call stack, and removes this space when the call ends.

The error `Stack_overflow` triggered earlier when calculating the length of a list of 700,000 elements is due to a limit on the size of the call stack. This limit depends on the operating system and the compiler. It is generally fairly low, of the order of several MB. When the stack reaches this limit, we call it a *stack overflow*. The program is then interrupted. This is signaled either by the exception `Stack_overflow` in OCaml or by a fatal crash of the program. If, on the other hand, the size of the stack is large (or even unlimited), the program will not trigger a stack overflow, but will consume more and more memory, which could eventually cause the machine to crash.

Tail Calls

You may wonder why the call `print !lines` does not provoke a stack overflow, unlike `length !lines`. The difference between the two functions is that, in the function `print`, the recursive call is the very last expression evaluated in the body of the function. This is not the case in the function `length`, where an addition remains to be performed after the recursive call. In the case of `print`, the compiler performs an *optimization*, which consists in removing the current call from the call stack, before performing the recursive call. Thus, the call stack only ever contains the final call to `print` and, therefore, does not increase in size.

In general, when a function call is the last expression to be evaluated in the body of a function, we say it is a *tail call*. Note that this notion applies to both recursive and non-recursive functions. In the former case, if every recursive call is a tail call, the function is said to be *tail recursive*. Thus, `print` is tail recursive and `length` is not. Note that a recursive function can contain both tail calls and non-tail calls. This is the case of McCarthy's famous 91 function.

```
let rec f91 n =
  if n > 100 then n - 10 else f91 (f91 (n + 11))
```

We can avoid the stack overflow in the function `length` by writing it differently, using an additional argument that represents the number of elements already traversed. Thus, we can write:

```
let rec length_term acc = function
  | []     -> acc
  | _ :: r -> length_term (1 + acc) r
```

In this version, the recursive call to `length_term` is a tail call. To calculate the length of a list, it suffices to call `length_term` with `0` as the first argument.

```
let length l = length_term 0 l
```

We can even define the function `length` as the result of the partial application of `length_term` to `0`.

```
let length = length_term 0
```

2.9 Converting Integers from an Arbitrary Base

Ideas introduced

- iterators
- polymorphism
- function `exit`

Our next program (see page 75), `radix.ml`, takes numbers written in some base B, where $2 \leq B \leq 36$, and converts them to base 10. The base is passed on the command line. The program then reads numbers from standard input and displays them in base 10.

Here is a first example in base 16:

```
> radix 16
7FFF
 -> 32767
A0
 -> 160
```

After launching the program with the given base, we enter the number `7FFF` on standard input, and the program displays the result of the conversion, namely `32767`. Next, we enter the number `A0` and obtain `160`.

Here is a second example, with numbers written in base 36:

```
> radix 36
ZORRO
 -> 59942292
```

As you see in these examples, the numbers are written using the characters `0` to `9`, and `A` (for 10) to `Z` (for 35).

Let us now pass to the implementation of the program. To begin, we retrieve the base from the command line and store it in a variable `base`.

```
let base = int_of_string Sys.argv.(1)
```

Program 9 [radix.ml] — Converting Integers from an Arbitrary Base

```
let base = int_of_string Sys.argv.(1)

let list_of_string s =
  let digits = ref [] in
  for i = 0 to String.length s - 1 do
    digits := s.[i] :: !digits
  done;
  !digits

let digit_of_char c =
  match c with
    | '0'..'9' -> Char.code c - Char.code '0'
    | 'A'..'Z' -> 10 + Char.code c - Char.code 'A'
    | c -> Printf.eprintf "invalid character %c\n" c; exit 1

let check_digit d =
  if d < 0 || d >= base then begin
    Printf.eprintf "invalid digit %d\n" d; exit 1
  end

let () =
  while true do
    let s = read_line () in
    let cl = list_of_string s in
    let dl = List.map digit_of_char cl in
    List.iter check_digit dl;
    let v = List.fold_right (fun d acc -> d + base * acc) dl 0 in
    Printf.printf " -> %d\n" v
  done
```

It is good practice to verify that the integer does indeed lie between 2 and 36, and to fail otherwise.

We continue by defining a function, `list_of_string`, which converts a string (of type `string`) to a list of characters (of type `char list`).

```
let list_of_string s =
  let digits = ref [] in
  for i = 0 to String.length s - 1 do
    digits := s.[i] :: !digits
  done;
  !digits
```

We traverse the string `s`, beginning with its first character, so as to construct a list in which the last character of `s`—that is, the least significant digit of the number—is found at the head of the list.

Next, we write a function `digit_of_char` to convert a character representing a digit into the corresponding integer.

```
let digit_of_char c =
  match c with
    | '0'..'9' -> Char.code c - Char.code '0'
    | 'A'..'Z' -> 10 + Char.code c - Char.code 'A'
    | c -> Printf.eprintf "invalid character %c\n" c; exit 1
```

The first pattern matching case `| '0'..'9' ->` treats the case of a character `c` between `'0'` and `'9'`. Its numeric value is obtained simply as the difference between the ASCII code of `c` and that of the character `0`. The second case proceeds in the same manner, for the characters between `'A'` and `'Z'`. For all other characters, the last case displays a message on standard error using `Printf.eprintf` and terminates the program using the predefined function `exit`.

We then write a function `check_digit` that verifies that `d` is a valid digit for the given base, that is, that it lies between 0 and `base` − 1.

```
let check_digit d =
  if d < 0 || d >= base then begin
    Printf.eprintf "invalid digit %d\n" d; exit 1
  end
```

This function does not return a result. It interrupts the program if the digit `d` is not valid.

The main part of the program is an infinite loop that reads a string `s` from standard input and converts the corresponding number to base 10.

```
let () =
  while true do
    let s = read_line () in
    ...
  done
```

We begin by converting the string `s` into a list of characters `cl` using the function `list_of_string`.

```
    let cl = list_of_string s in
```

In order to convert the list `cl` into a list of digits, we apply the function `digit_of_char` to *each* element of `cl`. For this, we use the library function `List.map`, which constructs a new list by applying a given function to all the elements of a list.

```
    let dl = List.map digit_of_char cl in
```

For example, if `cl` is the list `['A'; '0']`, then `dl` is the list `[10; 0]`. More generally, given a function f and a list l `[`e_1`;`e_2`;`...`;`e_n`]`, we have:

$$\texttt{List.map}\ f\ l = \texttt{[}f\ e_1\texttt{;}f\ e_2\texttt{;}\ \ldots\ \texttt{;}f\ e_n\texttt{]}$$

We then verify that each digit of the list `dl` is valid by applying the function `check_digit` successively to each element of `dl`.

```
    List.iter check_digit dl;
```

In this case we do not wish to construct a new list but rather to apply `check_digit`. The whole program will be interrupted if `check_digit` detects an invalid digit. In general, given a function f and a list l equal to `[`e_1`;`e_2`;`...`;`e_n`]`, the application `List.iter` $f\ l$ is equivalent to the following sequence:

$$\texttt{List.iter}\ f\ l = f\ e_1\texttt{;}f\ e_2\texttt{;}\ \cdots\ \texttt{;}f\ e_n$$

Finally, we will calculate the base 10 notation and display it. The list `dl` is of the form `[`d_0`;`d_1`; ... ;`d_{n-1}`]`, where d_0 is the digit corresponding to the last character in the string, that is, the least significant digit. It is therefore necessary to calculate the following value:

$$\sum_{i=0}^{n-1} d_i \times \texttt{base}^i$$

The most efficient method to minimize the number of multiplications (in particular to avoid the expensive calculation of $\texttt{base}^i$) is Horner's rule, which rewrites the preceding sum as follows:

$$d_0 + \texttt{base} \times (d_1 + \texttt{base} \times (\ldots (d_{n-2} + \texttt{base} \times (d_{n-1} + \texttt{base} \times 0)) \ldots))$$

We can easily program the above using the library function `List.fold_right`, which allows us to traverse the list `dl` from the last element d_{n-1} to the first element d_0, applying a function f to each element as follows:

$$\texttt{List.fold\_right}\ f\ l\ acc = f\ d_0\ (\ldots (f\ d_{n-2}\ (f\ d_{n-1}\ acc) \ldots))$$

The value computed for each element of the list is systematically passed to the function f as the second argument, called the *accumulator*. Its initial value, used when calling f on the last element d_{n-1} of the list, is the third argument of `List.fold_right`, here called *acc*. To obtain Horner's formula, it suffices to let f be the function `fun d acc -> d + base * acc`. Thus, the end of the program is simply:

```
let v = List.fold_right (fun d acc -> d + base * acc) dl 0 in
Printf.printf " -> %d\n" v
```

Additional Information

Polymorphism

To be useful, the functions of the library `List`, such as `iter`, `map`, or `fold_right`, must be generic with respect to both the type of the elements of the list to which they are applied, and the operation performed on them. This genericity is called *polymorphism*. The notion may be illustrated using the function `length` to calculate the length of a list:

```
let rec length l =
  match l with
  | []      -> 0
  | _ :: r -> 1 + length r
```

This function may, of course, be applied to a list of integers. For example:

```
# length [1; 6; 2; 8; 3] ;;
- : int = 5
```

However, this function does not directly use the elements of the list `l` passed as argument (as indicated by the pattern matching `_ :: r`). Therefore, this function may in fact be applied to any list, be it a list of strings or one of floating-point numbers. For example:

```
# length ["hello"; "world"; "!"] ;;
- : int = 3
# length [ [3.4; .2]; []; [1.2]; [5.]] ;;
- : int = 4
```

For the above to be possible, the function `length` must accept arguments of type `int list`, `string list`, and even `float list list`. This function is, in fact, applicable to lists of *any type*. In OCaml, it is given the type:

```
'a list -> int
```

where the *type variable* `'a` represents any type. A type such as that of `length`, which contains at least one type variable, is said to be *polymorphic*.

Apart from representing any type, type variables are also useful in establishing relations between the generic parts of a polymorphic type. For example, in the function `f`, defined by `let f g x = (g x) + 1`, the types of the arguments `g` and `x` must be related because the function `g` is applied to `x`. Nevertheless, `x` (and hence `g`) remains polymorphic, because there is no other type constraint involving this value in the body of `f`. Thus, in the type of `f`, that is, `('a -> int) -> 'a -> int`, the variable `'a` relates the type of the argument of `g` with the type of `x`, without adding any extra constraints. Similarly, the function `iter` of the library `List` has a polymorphic type:

```
iter  :  ('a -> unit) -> 'a list -> unit
```

This relates the type of the function passed as argument to that of the elements of the list that the function iterates over.

A polymorphic type can contain several type variables, to represent unrelated generic types. Thus, consider the function `make_pair`, defined by:

```
let make_pair x y = (x, y)
```

Here, the arguments `x` and `y` may be of any type, and they need not be related. Hence this function has a polymorphic type with two variables, that is, `'a -> 'b -> 'a * 'b`.

In the same way, the functions `map` and `fold_right` of the library `List` have the following polymorphic types:

```
map         :  ('a -> 'b) -> 'a list -> 'b list
fold_right  :  ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

These types are the most general possible.

Right-to-Left and Left-to-Right Iterators on Lists

Combining higher-order functions and polymorphic types allows us to define *iterators*, that is, functions that traverse generic data structures, such as lists.

For example, the iterator `fold_right` used in the program `radix.ml` can be defined as follows:

```
# let rec fold_right f l e =
    match l with
    | [] -> e
    | x :: r -> f x (fold_right f r e);;
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

As we have seen in the program `radix.ml`, the use of such an iterator allows us to avoid writing recursive functions. Suppose, for example, that we wish to multiply all the elements of a list of floats. We can simply write a function `mult` as follows:

```
# let sum l = fold_right (fun x y -> x *. y) l 1. ;;
val mult : float list -> float = <fun>
```

The reason for calling this iterator `fold_right` is that it treats the elements of the list from right to left. The list is, in fact, traversed from left to right, but the recursive call is performed *before* applying the function passed as argument. This means that the rest of the list is treated before its first element. We can confirm this behaviour by applying `fold_right` to a function that prints its first argument:

```
# fold_right (fun x () -> print_int x) [1;2;3;4] ();;
4321- : unit = ()
```

It is natural at this point to consider the other iterator, which traverses and treats elements from left to right. It is naturally called `fold_left` and is written as follows:

```
# let rec fold_left f e = function
    | [] -> e
    | x :: r -> fold_left f (f e x) r;;
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

It is clear that the elements are effectively treated from left to right:

```
# fold_left (fun () x -> print_int x) () [1;2;3;4];;
1234- : unit = ()
```

We can write the function `mult` with `fold_left` as we did with `fold_right`:

```
# let mult l = fold_left (fun x y -> x *. y) 1. l ;;
val mult : float list -> float
```

There is, nevertheless, a slight difference. On a list containing hundreds of thousands of elements, the function `mult` written with `fold_right` may trigger a stack overflow, which results either in an exception `Stack_overflow` or in the sudden halt of the program. This is because the recursive call to `fold_right` is not a tail call. (Since the call must be carried out before performing the operation, it is necessary to save the value of the first element of the list.) By contrast, in the case of `fold_left`, the recursive call is a tail call. (The stack is not needed since the value `x` is used before the recursive call to `fold_left`).

It is important to note that an iterator like `fold_left` or `fold_right` traverses *all* the elements of the list passed as argument. If we wish to stop the calculation at a specific element, we have at least two possibilities[4]: Either we write a recursive function that traverses the list and stops when we want it to. Alternatively, we can use an iterator and raise an exception to interrupt the list traversal. Consider again the example of the function `mult`. We may wish to stop the calculation when the value 0 is encountered. The version with the exception may be written as follows:

```
let mult l =
  try
    List.fold_left
      (fun x y -> if y = 0. then raise Exit else x *. y) 1. l
  with Exit -> 0.
```

However, in this case, writing a recursive function that terminates when it encounters 0 is simpler and more elegant.

2.10 Breakout without Bricks

Ideas introduced

- compilation unit, separate compilation
- module, interface

Our tenth program (see page 85) is a simplified version of the game Breakout, without the bricks. The game consists in making a ball bounce within a frame,

[4]Chapter 9 will present another type of iterator which can solve this problem.

by means of a paddle that is controlled using the mouse.

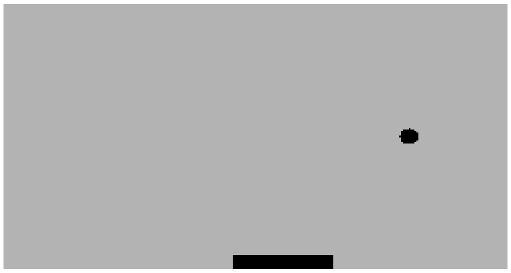


Figure 2.7: Breakout without bricks.

Graphically, the game window is as shown in figure 2.7, represented by a gray frame, with a black circle representing the ball, and a black rectangle at the base of the game window representing the paddle.

Programming such a game involves three main ingredients. First, we must define the *constants* of the game, that is, the dimensions of the window (its height and width, in pixels), the dimensions of the ball (that is, the radius of the circle), and the paddle (the height and width of the rectangle). Next, we must determine the *state* of the game, that is, the set of values that characterize it at any instant. Here, these are the coordinates of the center of the ball, its velocity, and the position of the paddle, which can be specified by the coordinates of its bottom-left corner. To simplify matters, we may use the same coordinate system as that of the library `Graphics`. That is, the point with coordinates $(0, 0)$ is located at the bottom left of the window, and the x- and y-axes are respectively the window's bottom and left edges. Finally, the game is played using an algorithm that executes the following steps:

1. Initialize the state of the game (the position and velocity of the ball, the position of the paddle).

2. Erase the graphical window.

3. Calculate the position of the paddle as a function of the x-coordinate of the mouse.

4. Display the ball and the paddle.
5. Calculate the new state of the game, that is:
   - the new position of the ball, as a function of its current position and velocity vector;
   - the new velocity vector of the ball, as a function of its bouncing off the sides of the window and the position of the paddle.
6. Return to step 2.

As you can see, the actions performed in these steps are of two types: those that are purely graphical (steps 2, 3, 4) and those that manipulate the state of the game (steps 1, 5, 6). We choose to divide the OCaml code into two files: `draw.ml` and `breakout.ml`. The first contains all the constants and graphical functions of the game; the second maintains the current state and implements the algorithm. In general, to clarify the structure of larger programs, it is a good idea to divide them into several files, called compilation units.

To compile this program, which is divided into two files, we use the following command line:

```
> ocamlc -I ‘ocamlfind query graphics‘ -o breakout graphics.cma draw.ml bre
```

As we will see shortly, the order of file names on the command line matters.

The graphical part (file `draw.ml`, program 10), begins by defining the dimensions of the game window, with the declaration of four constants: `left`, `right`, `down`, and `up`.

```
let left = 0.
let right = 300.
let down = 0.
let up = 200.
```

These constants are values of type `float`, as we will later be using floating-point numbers in trajectory calculations so as to guarantee a high degree of precision.

The following three constants define respectively the radius of the ball, the length of the paddle, and its width (in pixels).

Program 10 [draw.ml] — Breakout without Bricks (1/2)

```
open Graphics

let left = 0.
let right = 300.
let down = 0.
let up = 200.

let ball = 5
let paddle = 50
let thick = 8

let gray = rgb 220 220 220

let init () =
  let s = Printf.sprintf " %dx%d" (truncate right) (truncate up) in
  open_graph s;
  auto_synchronize false

let clear () =
  set_color gray;
  fill_rect 0 0 (truncate right) (truncate up)

let get_paddle_pos () =
  let x = fst (mouse_pos ()) in
  max 0 (min x (truncate right - paddle))

let game x y =
  clear ();
  set_color black;
  fill_circle (truncate x) (truncate y) ball;
  let x = get_paddle_pos () in
  fill_rect x 0 paddle thick;
  synchronize ();
  x
```

```
let ball = 5
let paddle = 50
let thick = 8
```

Finally, we declare a color `gray`, obtained by specifying the values of its three components—red, green, and blue—using the function `rgb`.

```
let gray = rgb 240 240 240
```

The function `init` initializes the game window by opening a graphical canvas of dimensions `right` $\times$ `up`.

```
let init () =
  let s = Printf.sprintf " %dx%d" (truncate right) (truncate up) in
  open_graph s;
  auto_synchronize false
```

Since the constants `right` and `up` are values of type `float`, we use the predefined function `truncate` from the standard library to truncate their values and convert them into integers. The call `auto_synchronize false` activates the *double buffering* mode of the graphical device. When this mode is activated, all display operations (drawing a circle, a rectangle, etc.) are performed in a memory zone called the *backing store*, and not directly on the screen. The function `synchronize` must then be used explicitly to copy the contents of the backing store to the screen. As this copy is instantaneous, it eliminates any "flickering" effects during the animation.

We now write a function for each graphical step of the algorithm. Step 2 is implemented by a function `clear` that erases the canvas by drawing a gray rectangle of the same dimensions as the graphical window.

```
let clear () =
  set_color gray;
  fill_rect 0 0 (truncate right) (truncate up)
```

Step 3 of the algorithm is implemented by a function `get_paddle_pos`. It begins by retrieving the `x`-coordinate of the current position of the mouse (with the function `mouse_pos`). To ensure that the paddle does not go outside the game window on the right side, we take the minimum between the value obtained and `truncate right - paddle`.

```
let get_paddle_pos () =
  let x = fst (mouse_pos ()) in
  max 0 (min x (truncate right - paddle))
```

The main graphical function, `game`, takes as arguments the `x`- and `y`-coordinates of the ball and implements steps 2, 3, and 4 of the algorithm.

```
  let game x y =
    clear ();
    set_color black;
    fill_circle (truncate x) (truncate y) ball;
    let x = get_paddle_pos () in
    fill_rect x 0 paddle thick;
    synchronize ();
    x
```

The second file, `breakout.ml` (see page 89), implements steps 1, 5, and 6 of the algorithm. The state of the game is encoded as two pairs of floating-point numbers: `(x, y)` and `(vx, vy)`, which represent respectively the coordinates of the ball and its velocity.

The first function of this file, `bounce`, calculates the new velocity of the ball, taking into account that the ball may bounce off the sides of the window or the paddle. The function `bounce` therefore also takes the position `xp` of the paddle as an argument.

```
  let bounce (x, y) (vx, vy) xp =
    ...
```

To calculate the new component `vx`, the `x`-coordinate of the ball must be compared with the left and right edges of the game window. The constants `left` and `right` having been defined in the file `draw.ml`, the dot notation `Draw.left` and `Draw.right` must be used to refer to them. Each compilation unit effectively defines a module bearing the same name as the file, but with the first letter in upper case.

```
    let vx =
      if x <= Draw.left || x >= Draw.right then -. vx else vx in
```

In the same way, to calculate the new component `vy`, we must compare the `y`-coordinate of the ball with the top of the window `Draw.up`. The direction of the ball's velocity also changes if the ball hits the paddle, that is, if the `x`-coordinate lies between `xp` and `xp +. float Draw.paddle`, and if the `y`-coordinate is less than the paddle thickness `Draw.thick`.

```
    let vy =
      if y <= float Draw.thick && x >= xp &&
         x <= xp +. float Draw.paddle || y >= Draw.up
      then -. vy else vy
    in
    (vx, vy)
```

The following function returns the ball's new position by adding the `x`- and `y`-coordinates, respectively, to the velocity's coordinates `vx` and `vy`.

```
  let new_position (x, y) (vx, vy) = x +. vx, y +. vy
```

The function `play` implements the loop between steps 2 and 6. Starting from the current state of the game, passed as argument, we begin by verifying that the ball does not go beyond the bottom of the game window, by comparing the `y`-coordinate to the constant `Draw.down`. If the ball does so, the game ends with a call to the function `failwith` of the standard library:

```
  let rec play (x, y) (vx, vy) =
    if y <= Draw.down then failwith "game over";
```

If the game has not ended, we display the ball and the paddle using the function `Draw.game`, which returns the `x`-coordinate of the paddle.

```
    let xp = Draw.game x y in
```

Next, we calculate the new coordinates of the velocity vector, and then those of the ball.

```
    let vx, vy = bounce (x, y) (vx, vy) (float xp) in
    let x', y' = new_position (x, y) (vx, vy) in
```

Program 11 [breakout.ml] — Breakout without Bricks (2/2)

```
let bounce (x, y) (vx, vy) xp =
  let vx =
    if x <= Draw.left || x >= Draw.right then -. vx else vx in
  let vy =
    if y <= float Draw.thick && x >= xp &&
       x <= xp +. float Draw.paddle || y >= Draw.up
    then -. vy else vy
  in
  (vx, vy)

let new_position (x, y) (vx, vy) = x +. vx, y +. vy

let rec play (x, y) (vx, vy) =
  if y <= Draw.down then begin Printf.eprintf "Game over!\n"; exit 0 end;
  let xp = Draw.game x y in
  let vx, vy = bounce (x, y) (vx, vy) (float xp) in
  let x', y' = new_position (x, y) (vx, vy) in
  play (x', y') (vx, vy)

let () =
  Draw.init();
  let speed = 0.1 in
  let vx = speed *. Random.float 1. in
  let vy = speed *. Random.float 1. in
  play (Draw.right /. 2., float Draw.thick) (vx, vy)
```

We return to step 2 by a recursive call to the function `play` with the new state of the game as argument.

```
    play (x', y') (vx, vy)
```

The main part of the program opens the game window by a call to `Draw.init`.

```
  let () =
    Draw.init();
```

It then creates a random velocity vector `(vx, vy)`:

```
    let speed = 0.1 in
    let vx = speed *. Random.float 1. in
    let vy = speed *. Random.float 1. in
```

Finally, the game begins with a call to the function `play`, placing the ball on top of the paddle and at the center of the window:

```
    play (Draw.right /. 2., float Draw.thick) (vx, vy)
```

Additional Information

Separate Compilation and Linking

The command line given above to compile Breakout can be decomposed into several commands. We can begin by compiling the file `draw.ml` alone, with the command:

```
  > ocamlc -I `ocamlfind query graphics` -c draw.ml
```

The compiler option `-c` specifies that we do not wish to construct an executable, but only to compile the code. The result consists of two files, `draw.cmi` and `draw.cmo`. The first contains typing information, and the second contains code. This code is not self-contained. In particular, it references functions from the module `Graphics`, which are not included in this file.

We can then compile the second file, with a similar command.

```
  > ocamlc -c breakout.ml
```

Since this file references the module `Draw`, we had to compile the file `draw.ml` first. The information needed for the type-checking of `breakout.ml` is contained in the file `draw.cmi`. Here, too, the code obtained is not self-contained; the file `breakout.cmo` references values and functions of the module `Draw` that are contained in `draw.cmo`.

To obtain an executable, we must perform a *linking* operation, which consists in putting together several pieces of code while verifying that every reference can be resolved. Here, the three pieces involved are those of the OCaml library `graphics.cma` (which groups a set of `cmo` files together) and the two files, `draw.cmo` and `breakout.cmo`.

```
> ocamlc -I ‘ocamlfind query graphics‘ -o breakout graphics.cma draw.cmo breakout
```

As we explained above, the order of file names on this command line is significant. In particular, it must be compatible with the dependencies between compilation units. Here, `breakout.cmo` *depends* on `draw.cmo`, which itself depends on `graphics.cma`. Furthermore, it must be understood that an OCaml program does not have a specific entry point. The code obtained after linking simply executes the code of the different files passed on the command line, in the order in which they appear.

Interfaces

The file `draw.cmi` produced by compilation contains the typing information of all the values defined in the file `draw.ml`. This same information can be obtained using the compiler option `-i`.

```
> ocamlc -I ‘ocamlfind query graphics‘ -i draw.ml
val left : float
val right : float
...
val game : float -> float -> int
```

We may, however, wish to make some of these values inaccessible to the rest of the program. For example, we may wish to hide the function `clear`. To this end,

Program 12 [draw.mli] — Interface for Module Draw

```
val left : float
val right : float
val down : float
val up : float

val paddle : int
val thick : int

val init : unit -> unit
val game : float -> float -> int
```

we can define an *interface* for `draw.ml`, in the form of a file `draw.mli`. Such a file is given in program 12. We write it using the same syntax as that used by the compiler option `-i`. We compile `draw.mli` before compiling `draw.ml`.

```
> ocamlc -c draw.mli
> ocamlc -I ‘ocamlfind query graphics‘ -c draw.ml
```

The first command produces the file `draw.cmi`. The second produces `draw.cmo` and verifies that it is compatible with `draw.cmi`, that is, that it defines values having the names and types specified in the file `draw.cmi`. When there is a file `draw.mli`, the compiler refuses to compile `draw.ml` until `draw.mli` has been compiled. If we then try to use a value in `breakout.ml` that is not mentioned in `draw.mli`, for example `clear`, an error is triggered.

```
> ocamlc -c breakout.ml
Error: Unbound value Draw.clear
```

Interfaces are not only a means of limiting the visibility of values defined in a compilation unit. They also enable *separate compilation*. This means that the code of each compilation unit may be compiled independantly of the code of other units, because it depends only on their interfaces. Once `draw.mli` is compiled, we can compile either `draw.ml` or `breakout.ml`, in any order. Notably, this makes it possible to divide the work amongst several developers once the interfaces are in place. Furthermore, separate compilation helps avoid unnecessary recompilation of code. For example, a change in the code of the function `Draw.clear` would only call for recompiling `draw.ml`, and then redoing the linking operation. No other unit using `Draw` needs to be recompiled (for example, `breakout.ml`).

Separate Compilation with `dune`

To compile our program with `dune`, we write the following configuration file:

```
(executable
 (name breakout)
 (modules draw breakout)
 (libraries graphics))
```

The line `(modules draw breakout)` lists the modules necessary for the compilation of the program. The tool `dune` will successively compile the modules `Draw` and `Breakout`, as we did manually above. In particular, if an `.mli` file exists, it is compiled before the corresponding `.ml` file. Furthermore, the line `(libraries graphics)` indicates that the files must be compiled with the library `Graphics`.

Modules

We have used various modules thus far, including those of the standard library, like `Array` and `List`, and the modules `Draw` and `Breakout` of the preceding example. These modules always corresponded to compilation units, that is, to pairs of `.ml` and `.mli` files. The notions of module and interface are, in fact, more precise than the notion of file; they correspond to constructs of the language. Thus, we can define an interface `I` that contains a constant `a` and a function `f` with the following syntax:

```
module type I = sig
```

```
    val a: int
    val f: int -> int
  end
```

We can then define a module M having this interface, with the following syntax:

```
  module M : I = struct
    let a = 42
    let b = 3
    let f x = a * x + b
  end
```

The compiler then performs the same operations as if we had written the interface I in a file `m.mli` and the module M in a file `m.ml`. In particular, we access the constant `a` with the notation `M.a`, while the constant `b` is not accessible outside the module M.

2.11 Logo Turtle

Ideas introduced

- abstract types
- private types
- encapsulation
- functors

Our next program, `turtle.ml`, is a simplified version of the Logo turtle. Although the programming language Logo has fallen into disuse today, conceptually, it remains as interesting as ever. Its most salient feature is a turtle that can be displaced using instructions of the form “forward 3 units” or “turn right 30 degrees,” and whose path is then displayed on the screen.

We can thus easily trace a square by repeating the sequence “forward, then

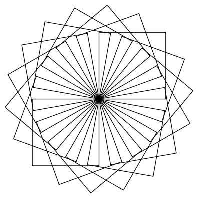


Figure 2.8: Example of a drawing using the Logo turtle.

turn left 90 degrees," or even draw figure 2.8 through several repetitions of the sequence "trace a square, then turn by 20 degrees."

Our aim here is to implement some of the basic turtle operations, such as moving forward, turning, and lifting the pen.

The following question arises naturally when writing the code of the turtle: how do we represent the angle that determines the turtle's direction?

We may choose to represent the angle in degrees or in radians, as an integer or a floating-point number, and so forth. Here, we decided not to choose amongst these options, and instead to *parametrize* the code by a module `A` that provides the type of angles.

We begin accordingly by creating a signature for such a module, namely:

```
module type ANGLE = sig
  type t
  val of_degrees: float -> t
  val add: t -> t -> t
  val cos: t -> float
  val sin: t -> float
end
```

The type `t` is that of angles. It has no definition; we say it is an *abstract type*. We can, nevertheless, use it in the implementation of our turtle because we are provided with the following: a function `of_degrees`, to convert an angle expressed in degrees to a value of type `t`; a function `add`, to add two angles; and, finally, `sin` and `cos` functions, to calculate the sine and cosine of an angle.

We can therefore write the Logo turtle as a module `Turtle`, parametrized by a module A with signature ANGLE. Such a module is called a *functor* and is defined thus:

```
module Turtle(A: ANGLE) = struct
```

Inside the module Turtle, the module A of signature ANGLE is visible and may be used like any other module. For example, we can introduce a reference angle, containing the current direction of the turtle, using:

```
  let angle = ref (A.of_degrees 0.)
```

We can also write a function rotate_left to make the turtle turn d degrees to the left:

```
  let rotate_left d = angle := A.add !angle (A.of_degrees d)
```

The complete code of the parametrized turtle is given in program 13 (see below). There, we find references tx and ty which keep track of the the current position of the turtle, and a function advance to make the turtle move forward. The boolean reference draw indicates whether the turtle's pen is up or down, and is modified using the functions pen_up and pen_down.

To use the functor Turtle, we must begin by providing a specific module with interface ANGLE. If we choose to represent angles in radians and floating-point numbers, one such module Angle may be defined as follows:

```
module Angle: ANGLE = struct
  type t = float
  let add = (+.)
  let pi_over_180 = atan 1. /. 45.
  let of_degrees d = d *. pi_over_180
  let cos = Stdlib.cos
  let sin = Stdlib.sin
end
```

We then obtain a module T by applying the functor Turtle to the module Angle, which is written thus:

```
module T = Turtle(Angle)
```

Program 13 [turtle.ml] — A Logo turtle

```
module type ANGLE = sig
  type t
  val of_degrees: float -> t
  val add: t -> t -> t
  val cos: t -> float
  val sin: t -> float
end

module Turtle(A: ANGLE) = struct

  let draw = ref true
  let pen_down () = draw := true
  let pen_up   () = draw := false

  let angle = ref (A.of_degrees 0.)
  let rotate_left d = angle := A.add !angle (A.of_degrees d)
  let rotate_right d = rotate_left (-. d)

  open Graphics
  let tx = ref 400.
  let ty = ref 300.
  let () = open_graph " 800x600"; set_line_width 2;
    moveto (truncate !tx) (truncate !ty)

  let advance d =
    tx := !tx +. d *. A.cos !angle;
    ty := !ty +. d *. A.sin !angle;
    if !draw then lineto (truncate !tx) (truncate !ty)
             else moveto (truncate !tx) (truncate !ty)

end
```

Finally, we can use the module T to trace the figure 2.8 by writing, for example:

```
let square d =
  for k = 1 to 4 do T.advance d; T.rotate_left 90. done
let squares d a =
  for k = 1 to truncate (360. /. a) do
    square d; T.rotate_left a
  done
let () = squares 100. 20.
```

Here, `square d` traces a square of side `d` and `squares d a`, a set of squares of side `d`, with a rotation of `a` degrees between any two successive squares.

The reason for writing the module `Turtle` as a functor, parametrized by the representation of angles, is that we can then apply it to other modules of signature `ANGLE`, and thus obtain other turtles where the angles are represented differently.

Additional Information

Abstract Types

In the above example, the type of angles `A.t` is an abstract type, because we do not yet know how it will be implemented. We may equally use the notion of abstract types to *hide* a concrete implementation, even when it is already known. This is what we call *encapsulation*. Suppose, for example, that we wish to define a module to manipulate integers between 0 and 30, and ensure that all the integers we use lie within this interval. We begin by defining a signature `INT31` for such a module:

```
module type INT31 = sig
  type t
  val create : int -> t
  val value : t -> int
end
```

This signature declares an abstract type `t`, and two functions `create` and `value`. We note that, by virtue of the abstract nature of the type `t`, a value of this type can only be constructed using the function `create`. Next, we can construct a module `Int31` with this signature.

```
module Int31 : INT31 = struct
  type t = int
  let check x = if x < 0 || x > 30 then invalid_arg "Int31.create"
  let create x = check x; x
  let value x = x
end
```

Inside this module, we give a definition to the type `t`, namely, `int`. Outside the module `Int31`, by contrast, the type `t` remains abstract, that is, we do not know that the values of type `Int31.t` are integers. In particular, we can enforce the invariant that any value of type `Int31.t` lies between 0 and 30. We would not have been able to do this if we had written `type t = int` in the interface `INT31`.

The abstract nature of type `t` is illustrated by the way in which the values are displayed by the OCaml interpreter:

```
# let x = Int31.create 7;;
x : Int31.t = <abstr>
```

Here, the value displayed for `x` is `<abstr>`, which denotes the value of an abstract type. Thus, `x` cannot be used like an integer:

```
# x + 10;;
Error: This expression has type Int31.t
       but an expression was expected of type int
```

The type system therefore distinguishes the two types `Int31.t` and `int`. Nevertheless, the value of `x` (namely, the integer 7) is exactly the same as if `x` had type `int`. Using an abstract type does not introduce any extra cost at runtime. If we wish to add the value of `x` to 10, it is necessary to apply the function `Int31.value` to `x` in order to get back a value of type `int`.

```
# Int31.value x + 10;;
```

```
- : int = 17
```

Private Types

As we have seen with the type `Int31.t`, an abstract type can be used to enforce an invariant. Let us consider the example of a module `Polar`, used to represent complex numbers in polar coordinates, with the type:

```
type t = { rho : float; theta : float; }
```

If we wish to enforce the invariant $0 \leq$ `rho` on this type, or to ensure that $0 \leq$ `theta` $< 2\pi$, one solution is to define an abstract type. However, in that case, we would no longer be able to access the fields `rho` and `theta` from outside the module `Polar`, and would therefore have to provide two accessor functions:

```
val rho : t -> float
val theta : t -> float
```

A more elegant solution consists in making the type `t` a *private type*. We give the module a signature in which the definition of the type `t` is qualified with `private`.

```
type t = private { rho : float; theta : float; }
```

The definition of the type `t` inside the module remains unchanged. The private nature of the type `t` entails that it is no longer possible to construct a record of type `Polar.t` outside the module. Thus, if we write the expression `{ Polar.rho = 1.; Polar.theta = 0. }`, the following error is triggered:

```
Error: Cannot create values of the private type Polar.t
```

However, it remains possible to construct values of this type inside the module `Polar`. We may therefore provide one—or several—functions to create values of type `t`, for example:

```
val create : float -> float -> t
```

We may implement this function inside the module in a way that enforces the invariant: We may either choose to fail if the values of `rho` and `theta` do not satisfy the invariant, or normalize the values so that they do. Private types are useful because their definition is not hidden, so that the structure of their values is accessible, but the construction of these values is still disallowed. In our example, we can access the fields `rho` and `theta` with the usual notation in an expression or a pattern matching. Furthermore, it would not be possible to modify a `mutable` field of a private record. The notion of privacy is not limited to record types.

Iterators and Abstract Types

Iterators allow the traversal of a data structure (see section *2.9 Converting Integers from an Arbitrary Base*). In case of concrete types, such as lists, we always have the option of directly defining a recursive function. By contrast, in case of *abstract* types, this is no longer possible. It is therefore a good idea to provide an iterator as well, in the form of a function `iter` or `fold`. If, for example, a module `S` provides an abstract type for sets of integers—let us call it `set`—it could also provide a function:

```
fold: (int -> 'a -> 'a) -> set -> 'a -> 'a
```

We can use this function to calculate the sum of the elements of a given set, even though we do not know its representation.

Functors

Functors allow us to construct data structures parametrized by other data structures. The OCaml standard library contains four examples of data structures defined as functors: `Hashtbl.Make`, `Set.Make`, `Map.Make`, and `Weak.Make`. Part II of this book also contains numerous such examples. Functors can also help us write algorithms parametrized by data structures or even by other algorithms (see the various examples in Part III). In general, functors are an elegant way of reusing code, by writing it in the most generic way possible. Despite numerous differences, they are comparable to C++ *templates*.

2.12 Playing a Musical Score

Ideas introduced

- algebraic types
- pattern matching

Our next program (see page 104), `music.ml`, plays a musical score. The simple scores that we are going to play are represented grapically as in figure 2.9.

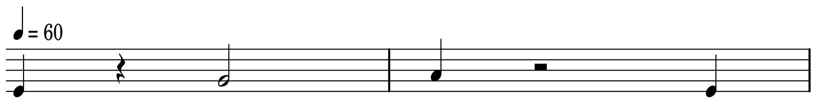


Figure 2.9: Musical score.

There are several types of *signs* here:

- *Notes* are determined by their pitch (or position) on the score and their duration. We are only interested in notes of two specific durations: half notes and quarter notes. Half notes are twice as long as quarter notes.
- *Silences,* for which we similarly consider only those that have the durations of half and quarter notes.

To these signs, we add the *tempo*, which determines the number of quarter notes per minute. In the example above, the tempo is 60 quarter notes per minute.

To play the notes, we need to know their respective frequencies (in Hz). To facilitate this calculation, we represent the pitch of a note not by its position on the score, but rather by using the following two pieces of information:

- a main *note*—*do*, *re*, *mi*, *fa*, *sol*, *la*, or *si*;
- an octave 0, 1, 2, 3, etc.

This representation allows us to determine the frequency of a note easily, using the following formula:

$$f = f_0 \times 2^o$$

Here, f_0 is the frequency of its main note in the octave 0 and o is its octave. Table 2.10 gives the frequencies in octave 0 for the main notes.

Note	Frequency (Hz) for octave 0
do	33
re	37
mi	41
fa	44
sol	49
la	55
si	62

Figure 2.10: Frequencies for octave 0 of the main musical notes.

In order to represent all the elements of a score, our program begins by defining several types. The main notes are represented by the type `note` as follows [5]:

```
type note = Do | Re | Mi | Fa | Sol | La | Si
```

This is an *enumerated* type, which defines a finite domain made up of seven elements (`Do`, `Re`, etc.), called *constructors*. Syntactically, OCaml requires that the name of a constructor begin with an uppercase letter.

Once we have defined this type, we may manipulate values of type `note` by simply using the names of their constructors:

```
# Re;;
- : note = Re
```

[5] Here, we use the French naming system of musical notes, which is used in many other countries as well.

Program 14 [music.ml] — Playing a Musical Score

```
type note = Do | Re | Mi | Fa | Sol | La | Si
type pitch = { note : note; octave : int }
type duration = Half | Quarter
type symbol = Note of pitch * duration | Rest of duration
type score = { symbols : symbol list;  metronome : int }

let frequency { note = n; octave = o} =
  let f0 =
    match n with
      | Do -> 33
      | Re -> 37
      | Mi -> 41
      | Fa -> 44
      | Sol -> 49
      | La -> 55
      | Si -> 62
  in
  f0 * truncate (2. ** float o)

let millisecondes d t =
  let quarter = 60000 / t in
  match d with
    | Half -> quarter * 2
    | Quarter -> quarter

let sound t s =
  match s with
    | Note (p, d) ->
        let f = frequency p in
        Graphics.sound f (millisecondes d t)
    | Rest r ->
        Graphics.sound 0 (millisecondes r t)

let play_score { symbols = l; metronome = t } =
    List.iter (sound t) l
```

The pitch of the notes is represented using records of type `pitch` with two fields, `note` and `octave`, which must contain non-negative integers.

```
type pitch = { note : note; octave : int }
```

The durations, which are of type `duration`, are represented by the two constructors `Half` (half notes) and `Quarter` (quarter notes).

```
type duration = Half | Quarter
```

To represent the two kinds of signs that may appear on musical scores, we define the type `symbol` as follows:

```
type symbol = Note of pitch * duration | Rest of duration
```

This is an *algebraic type*, which allows us to distinguish between notes and silences by means of two constructors, `Note` and `Rest`. Unlike the constructors of enumerated types `note` and `duration`, the constructors `Note` and `Rest` have arguments. In case of `Note`, the arguments are of type `pitch` and `duration`. `Rest` has an argument of type `duration`. For example, to create a silence of quarter-note duration, we write:

```
# Rest Quarter;;
- : symbol = Rest Quarter
```

Similarly, to create a note re in octave 1 of half-note duration, we write:

```
# Note ({ note = Re; octave = 1 }, Half);;
- : signe = Note ({note = Re; octave = 1}, Half)
```

Finally, scores are represented by the record type `score`, composed of a list of signs and a tempo that gives the number of quarter notes per minute.

```
type score = { symbols : symbol list;  metronome : int }
```

The first function of the program calculates the frequency associated with the pitch of a note based on the formula and the table of frequencies given above.

```
let frequency { note = n; octave = o } =
```

Given a note `n` of type `note` and an octave `o`, the function `frequency` begins by determining the frequency `f0` of the main note `n` using the pattern matching construct `match n with` as follows:

```
let f0 =
 match n with
   | Do -> 33
   | Re -> 37
   | Mi -> 41
   | Fa -> 44
   | Sol -> 49
   | La -> 55
   | Si -> 62
in
...
```

This construct returns different values depending on the constructor used to create `n`. Thus, if `n` is the constructor `Do`, the construct `match n with` returns the integer `33`. Else, if `n` is the constructor `Re`, it is the integer `37` that is returned, and so forth. The main frequency `f0` is then used to calculate the frequency of the note:

```
f0 * truncate (2. ** float o)
```

The second function calculates the time (in milliseconds) corresponding to a duration `d` (a half or quarter note), for a given tempo `t`.

```
let milliseconds d t = ...
```

We begin by calculating the time of a quarter note, (`quarter`), for the given tempo:

```
let quarter = 60000 / t in
```

Next, depending on the duration `d`, we return `quarter * 2` milliseconds for a quarter note, and `quarter` milliseconds for a half note:

```
match d with
| Half -> quarter * 2
| Quarter -> quarter
```

The following function, `sound`, either plays a musical note or a silence:

```
let sound t s = ...
```

Given a tempo `t` and a symbol `s`, `sound` pattern matches on `s` to determine if it is a musical note or a silence:

```
    match s with
      | Note (p, d) ->
          ...
      | Rest r ->
          ...
```

This pattern matching construct allows us not only to discern the constructor used to create `s`, but also to retrieve the arguments of this constructor. Thus, if `s` corresponds to the first *pattern* `Note (p,d)`, the two variables `p` and `d` represent, respectively, the pitch and duration associated with this constructor. In the second pattern, the variable `r` represents the silence associated with the constructor `Rest`. In both cases, the variables introduced by the patterns have a scope limited to the expression on the right-hand side of the arrow `->`.

When `s` is of the form `Note (p, d)`, we calculate the frequency `f` associated with the pitch `p` of the note using the function `frequency`. Then, we use the function `Graphics.sound` to emit a sound of frequency `f` for a duration of `(milliseconds d t)`:

```
      | Note (p, d) ->
        let f = frequency p in
        Graphics.sound f (millisecondes d t)
```

When `s` is a silence of the form `Rest r`, we emit a sound of zero frequency during `(milliseconds r t)` milliseconds.

```
      | Rest r ->
        Graphics.sound 0 (millisecondes r t)
```

Finally, the last function plays the notes of a score one by one, using the function `sound`:

```
  let play_score { symbols = l; metronome = t } =
    List.iter (sound t) l
```

Additional Information

Pattern Matching

In general, the pattern matching construct `match v with` generalizes what we saw earlier in case of lists (section 2.8 *Reversing the Order of Lines in a Text*). It contains `k` branches, each branch associating a pattern with an expression:

```
match v with
  | pattern_1 -> e_1
  | pattern_2 -> e_2
  | ...
  | pattern_k -> e_k
```

The value `v` is first evaluated and is then compared, from top to bottom, with each pattern. The only expression evaluated by this construct is the `e_i` corresponding to the first pattern *compatible* with the shape of `v`.

The syntax of patterns is very expressive. It allows us to decompose complex values with ease, for example values defined using the following types `t` and `u`:

```
type t = A of int * float | B of string
type u = { a : t; b : int * t }
```

Thus, the pattern `{ b = (_, A (x, _)) }` allows us to retrieve the integer `1` passed as argument to the constructor `A` in the following value:

```
{ a = B "foo"; b = (10, A(1, 4.5)) }
```

Informally, the pattern `{ b = (_, A (x, _)) }` is read as "a record whose field `b` contains a pair, the second component of which is a value constructed using `A` with two arguments, the first of which is named `x`."

OCaml can also verify the exhaustivity of pattern matching, ensuring that all possible cases are indeed covered. Thus, for any value `v` of type `u`, the compiler detects that the following pattern matching is not exhaustive:

```
match v with
  | { a = A(_, x); b = (_, B y) } -> ...
  | { a = A(_, x); b = (_, A (y, _)) } -> ...
  | { a = B x; b = (_, B y) } -> ...
```

It even indicates that values of the form `{ a=B _; b=(_, A (_, _)) }` are not handled by this pattern matching, which could lead to a runtime error. By contrast, the compiler verifies that the following construct *is* exhaustive, although this is not obvious *a priori*:

```
match v with
  | { a = A(_, x); b = (_, B y) } -> ...
  | { a = A(_, x); b = (_, A (y, _)) } -> ...
  | { a = B x; b = (_, B y) } -> ...
  | { a = _;  b = (_, A(_, y)) } -> ...
```

The pattern matching construct also allows us to associate the same expression with several patterns. These patterns, called *or* patterns, take the following form:

```
match v with
  | pattern_1 | pattern_2 | ... | pattern_k -> e
  | ...
```

The only syntactic constraint is that each pattern must introduce the same variable names, and these must have the same type. For example, we may write the following *or* pattern on values of type `u`:

```
match v with
  | { a = A(x, _) } | { b = (x, _) } -> ...
  | ...
```

It is equally possible to associate variable names to the sub-parts of a pattern using the `as` notation, as in the following example:

```
match v with
  | { a = B _ ; b = (_, (A (y, _) as z)) } -> (y, z)
  | ...
```

Here, the variable `z` references the value that is matched by the (sub-)pattern `A (y, _)`. Thus, the application of this pattern matching to `{ a = B "foo"; b = (10, A(1, 4.5)) }` will construct the pair `(1, A(1, 4.5))`.

Finally, pattern matching may be extended with arbitrary boolean conditions, using the `when` notation as follows:

```
match v with
  | pattern_1 when e -> e_1
  | ...
```

For example, we can match values of type `u` having a field `a` of the form `A(x, _)`, with `x` greater than `10`, as follows:

```
match v with
  | { a = A(x, _) } when x > 10 -> ...
  | ...
```

It is important to note that the exhaustivity analysis does not take into account these boolean conditions and that all patterns constrained by such expressions are simply ignored.

Algebraic Types

It is also possible to define polymorphic algebraic types. To do so, we must indicate the list of type variables (or parameters) that appear in the definition. Consider the following example:

```
type ('a, 'b) t = C of 'a * int | D of 'b | E of int
```

Here, the type `t` is parametrized by two variables `'a` and `'b`. The constructors `C` and `D` are polymorphic. Thus, as in the case of a polymorphic function, these constructors may be applied to arguments of any type:

```
# let v = C(3, "foo");;
val v : (int, string) t = C(3, "foo")
```

The type `'a list` of polymorphic lists predefined in the OCaml standard library is an example of a polymorphic algebraic type. If the syntax allowed it, it could be defined by:

```
type 'a list = [] | :: of 'a * 'a list
```

Another example of a polymorphic algebraic type is the type `option` of the standard library defined by:

```
type 'a option = None | Some of 'a
```

It serves in particular to represent a value not yet initialized, or an optional result. For example, the following function returns either `None`, when it is not possible to divide `x` by `y`, or else `Some (x/y)`:

```
let division x y = if y = 0 then None else Some (x / y)
```

```
# division 2 0;;
- : int option = None
```

2.13 Quadtrees

Ideas introduced

- trees, binary trees
- sharing

Our next program (page 114), `quad.ml`, manipulates black-and-white square images of size $2^n \times 2^n$. If the image is completely black or white, it is represented using a constant denoting its color. Otherwise, it is decomposed into four sub-images, each of size $2^{n-1} \times 2^{n-1}$, following the order indicated in figure 2.11.

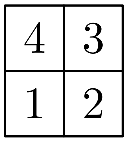


Figure 2.11: Breaking up an image.

The image is the union of the four sub-images. In our program, the type `quad` corresponds to this representation.

```
type quad = White | Black | Node of quad * quad * quad * quad
```

The constants `White` and `Black` respectively denote an image that is either completely white or completely black. The constructor `Node` corresponds to an image decomposed into four sub-images, which are the four arguments of this constructor.

Consider the image in figure 2.12. It is represented by the following value of type `quad`:

```
Node (Node (Black, Black, Black, White),
      Black,
      Node (Black, Black, Black, White),
      White)
```

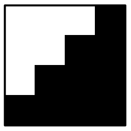
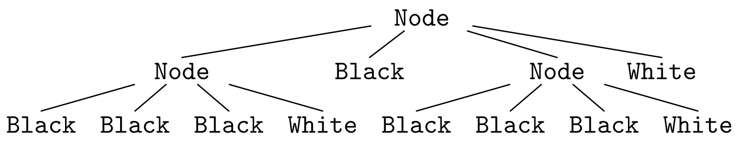


Figure 2.12: An image of size 4×4.

Graphically, we may also represent this value as follows:



This is why we speak of *trees* in relation to such values.

Let us now write a function `checker_board`, which constructs a quadtree corresponding to a $2^n \times 2^n$ checkerboard. Thus, `checker_board 3` corresponds to the image in figure 2.13.

We proceed by recursion on n. If n is 0, we make an arbitrary choice and return a black square.

```
let rec checker_board = function
  | 0 -> Black
```

If $n = 1$, we return a checkerboard of size 2×2:

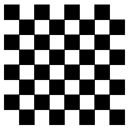


Figure 2.13: An 8×8 checkerboard.

```
  | 1 -> Node (White, Black, White, Black)
```

Finally, if $n > 1$, we construct a $2^{n-1} \times 2^{n-1}$ checkerboard that we use four times over to construct a checkerboard of size $2^n \times 2^n$.

```
  | n -> let q = checker_board (n - 1) in Node (q, q, q, q)
```

Let us now write a function `draw` that draws the image represented by a quadtree. This function takes as arguments the coordinates of the square in which we wish to draw the image: the position $(\mathtt{x}, \mathtt{y})$ of the bottom-left point of the image, and the side length `w` of the square. We proceed recursively on the structure of the quadtree.

```
  let rec draw x y w = function
```

If the tree consists of only one leaf, we distinguish between two cases: For a white leaf, we do nothing; for a black leaf, we shade the square defined by `x`, `y`, and `w`.

```
    | White -> ()
    | Black -> Graphics.fill_rect x y w w
```

If, however, we are in the case of the constructor `Node`, we begin by calculating the side length of the four sub-images, namely, $\mathtt{w}/2$.

```
    | Node (q1, q2, q3, q4) ->
        let w = w / 2 in
```

Then, we draw the sub-images with four recursive calls to `draw`. In each case, we pass the coordinates of the bottom-left corner of the corresponding sub-image.

```
        draw x       y       w q1;
        draw (x + w) y       w q2;
```

Program 15 [quad.ml] — Quadtrees

```
type quad = White | Black | Node of quad * quad * quad * quad

let rec checker_board = function
  | 0 -> Black
  | 1 -> Node (White, Black, White, Black)
  | n -> let q = checker_board (n - 1) in Node (q, q, q, q)

let rec draw x y w = function
  | White ->
      ()
  | Black ->
      Graphics.fill_rect x y w w
  | Node (q1, q2, q3, q4) ->
      let w = w / 2 in
      draw x       y       w q1;
      draw (x + w) y       w q2;
      draw (x + w) (y + w) w q3;
      draw x       (y + w) w q4

let () = draw 0 0 256 (checker_board 3)
```

```
        draw (x + w) (y + w) w q3;
        draw x       (y + w) w q4
```

In this way, we can draw an 8×8 checkerboard in a square of side 256 by calling `draw 0 0 256 (checker_board 3)`.

Additional Information

Constructors with Records

With the type `quad` of quadtrees, it is easy to make a mistake in the order of the sub-trees of the constructor `Node`. We have to be very careful and constantly keep in mind the figure 2.11. An alternative consists in naming the four sub-trees, as we do with record types:

```
type quad =
  | White
  | Black
  | Node of { sw: quad; se: quad; ne: quad; nw: quad }
```

Both within a pattern and when constructing a value, we use the syntax of records to manipulate the arguments of the constructor `Node`. Thus, we construct a node as follows:

```
Node { nw = a; sw = b; se = c; ne = d }
```

As we see here, the declaration order of the fields is not important. As in case of records, we can use pattern matching to retrieve their values:

```
match q with
  | ...
  | Node { sw; se; ne; nw } -> ...
```

However, we cannot write the following:

```
match q with
  | ...
  | Node r -> ...
```

This is because there is no record as such, only a syntax to name the four arguments of the constructor `Node`.

To understand the difference, let us compare the constructor `Node` above with the constructor `C` of the type `t` below:

```
type r = { a: int; b: bool; c: int }
type t = C of r | ...
```

Here, the constructor `C` has a single argument of type `r`, which is a record. This value can be retrieved and, for example, passed to a function which takes a value of type `r`:

```
match ... with
| C r -> f r
```

This, however, has a cost. The constructor `C` contains a pointer to a record allocated previously, which means two allocations are needed instead of one. And accessing the components of the constructor `C` requires two memory accesses, the first to access the record and the second to access the field.

Invariants

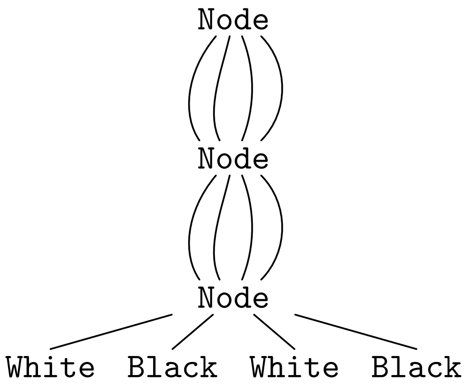
We may wish to guarantee the property that a quadtree is never made up of four leaves of the same color, since that would be an unnecessarily complicated representation. To enforce this invariant, we must make `quad` either an abstract or a private type. As alternatives to the constructors `Black`, `White`, and `Node`, we must then provide two constants `black` and `white`, and a function `node`, defined so as to guarantee the invariant.

```
let node = function
  | White, White, White, White -> White
  | Black, Black, Black, Black -> Black
  | q1, q2, q3, q4 -> Node (q1, q2, q3, q4)
```

We is known as a *smart constructor*.

Sharing

The attentive reader will have noticed that the function `checker_board` makes *a single* recursive call, reusing its result four times. What we have constructed is thus not a tree, but rather a DAG. A depiction of the result of `checker_board 3` that more accurately reflects its memory representation is therefore:



In particular, `checker_board` n executes in $O(n)$ time and space. By contrast, *drawing* its result takes $O(4^n)$ time because the function `draw` traverses the four sub-images successively. In other words, the function `draw` traverses a tree without taking into account the sharing that exists in memory. Section *11.4 Hash-consing* explains how to exploit such sharing.

Binary Trees

In a binary tree, each internal node contains exactly two sub-trees. The leaves, like the nodes, can be labeled. For example, binary trees with internal nodes labeled by integers correspond to the type:

```
type tree = Leaf | Node of tree * int * tree
```

Similarly, trees with leaves labeled by strings correspond to the type:

```
type tree = Leaf of string | Node of tree * tree
```

The reader will find numerous examples of such trees in the pages that follow.

n-ary Trees

The number of sub-trees of an internal node is not necessarily fixed. It may vary and can even be unbounded. In this case, we may represent the sub-trees of a node by a list of trees. Thus we write:

```
type tree = Node of tree list
```

It is interesting to note that we no longer need a specific constructor to represent leaves because the value `Node []` fulfills this role. Of course, we may label the nodes of such a tree if we wish. An example may be found in section 5.4 *Prefix Trees*.

2.14 Solving the N-Queens Problem

Ideas introduced

- backtracking
- persistence

We are interested here in the classic N-queens problem: N queens are to be placed on an $N \times N$ chessboard such that none of them are threatened by the others. Figure 2.14 shows one of the 92 solutions when $N = 8$.

More precisely, we are interested in the problem of enumerating all solutions, without taking into consideration the symmetries of the problem. We proceed in a relatively brute fashion by exploring all possibilities, noting nevertheless that a solution must necessarily have one and only one queen on each line of the chessboard. In light of this, we try to fill the chessboard line by line, placing a queen each time in such a way that it is not threatened by the queens already placed. Thus, if we have placed three queens on the first three lines of the chessboard, we then search for a valid placement on the fourth line as follows:

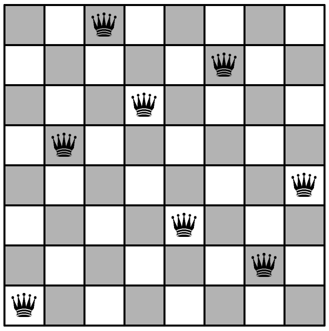
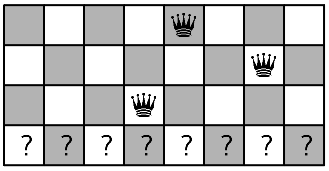


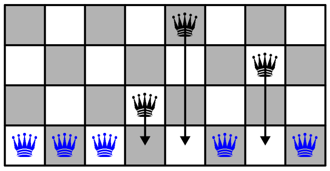
Figure 2.14: One solution of the 8-queens problem.



If we find one, we place a queen there and continue the search on the following line. Otherwise, we go back one step and begin again. A solution is obtained each time we manage to place a queen on the last line. By proceeding in such a systematic manner, we are sure to find all solutions. This technique is called *backtracking*.

For each line of the chessboard, our program will keep track of the columns in which a queen may be placed. Thus, instead of trying all the N columns of the current line, we would be able to examine far fewer than N and would therefore backtrack sooner. We illustrate this idea for $N = 8$.

Let us suppose that we have already placed queens on the first three lines. Only five columns need be considered in the fourth line (at the bottom of the chessboard).

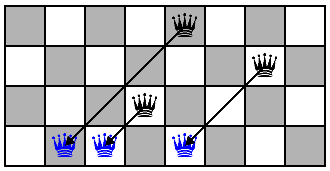


(1)

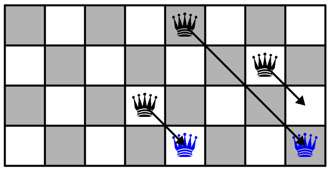
Program 16 [queens.ml] — The *N*-Queens Problem

```
module S = Set.Make(struct type t = int let compare = compare end)

let map f s = S.fold (fun x s -> S.add (f x) s) s S.empty

let rec upto n = if n < 0 then S.empty else S.add n (upto (n-1))

let rec count cols d1 d2 =
  if S.is_empty cols then
    1
  else
    S.fold
      (fun c res ->
        let d1 = map succ (S.add c d1) in
        let d2 = map pred (S.add c d2) in
        res + count (S.remove c cols) d1 d2)
      (S.diff (S.diff cols d1) d2)
      0

let () =
  let n = int_of_string Sys.argv.(1) in
  Format.printf "%d@." (count (upto (n - 1)) S.empty S.empty)
```

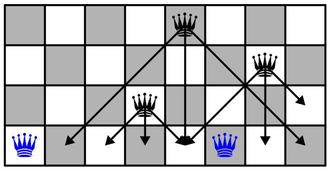
Furthermore, three positions in the fourth line are threatened by queens already placed along a left diagonal. These three positions need not be considered.


(2)

Similarly, two positions in the fourth line are threatened by queens already placed along a right diagonal. These two positions need not be considered.


(3)

There are thus six positions on the fourth line that cannot be used, leaving only two to be consider, rather than eight.


(4)

Our program will proceed recursively, keeping track at each instant of three sets of columns: the set `cols` of columns that remain to be considered; the set `d1` of columns that should be avoided since queens placed there would be threatened along a left diagonal; and the set `d2` of columns that should be avoided since queens placed there would be threatened along a right diagonal. In our example, numbering the columns starting from the right, we have `cols` $= \{0, 2, 5, 6, 7\}$, `d1` $= \{3, 5, 6\}$, and `d2` $= \{0, 3\}$, as shown in the figures above (respectively in (1), (2), and (3)). The set of columns where queens can be placed is obtained by calculating the set-theoretic difference `cols`\\`d1`\\`d2`, which gives the set $\{2, 7\}$, as shown in figure (4).

To manipulate such sets of integers, we use the module `Set.Make` of the OCaml standard library. This is a functor that we instantiate with a type equipped with a total order. (The code of such a functor will be explained later in this book, in the section *AVL* of chapter 5.)

We choose here the usual order on the integers, provided by the function `compare` of the library `Stdlib`.

```
module S = Set.Make(struct type t = int let compare = compare end)
```

The resulting module `S` provides a data structure `S.t` that represents a set of integers, a constant `S.empty` corresponding to the empty set, and operations such as `S.add` (adding an element), `S.remove` (removing an element), `S.diff` (set-theoretic difference), etc. An important property of this data structure is its *persistence*. This means that the operations applied to this data structure do not modify it, but instead return *new* data structures. Thus, if we have a set `s` of type `S.t`, then the expression `S.add 4 s` denotes a new set (containing 4 and all the elements of `s`), while the set `s` remains unmodified.

Let us now write our program using a recursive function `count` that takes as arguments the three sets `cols`, `d1`, and `d2` described above:

```
let rec count cols d1 d2 =
```

This function returns the number of solutions that are compatible with these arguments. The search ends when `cols` is empty. We then indicate that a solution has been found.

```
if S.is_empty cols then 1 else
```

Otherwise, we calculate the set of columns in which queens can be placed using the expression `S.diff (S.diff cols d1) d2`, as explained above. Then, we traverse the elements of this set using the iterator `S.fold`, where the accumulator is the current number of solutions found.

```
  S.fold
    (fun c res -> ...)
    (S.diff (S.diff cols d1) d2)
    0
```

For each column `c` where a queen can be placed, it suffices to call `count` recursively with the three sets adjusted accordingly. To adjust `cols`, it suffices to remove the element `c`, using `S.remove`. To adjust `d1` and `d2`, we use `S.add` to add the column `c` to each of them, and then shift their elements by one unit as appropriate. To do this, we define a function `map` that applies a function f to all the elements of a set s, that is, it constructs the set $\{f(x) \mid x \in s\}$. We may write it, for instance, using `S.fold`.

```
let map f s = S.fold (fun x s -> S.add (f x) s) s S.empty
```

Given this function, the set d1 may be adjusted using map succ, where succ is the predefined function fun x -> x + 1. Likewise, the set d2 is adjusted with map pred, where pred is the predefined function fun x -> x - 1. The function passed as argument to S.fold therefore has the following form:

```
(fun c res ->
  let d1 = map succ (S.add c d1) in
  let d2 = map pred (S.add c d2) in
  res + count (S.remove c cols) d1 d2)
```

With this, the function count is complete. It is important to note here the key role played by the persistence of the sets cols, d1, and d2. Indeed, these sets are reused for each value of c, and must therefore not be modified by the operations S.add and S.remove.

To solve the N-queens problem, it suffices to call count with cols equal to the set $\{0, 1, \ldots, N-1\}$, and d1 and d2 each equal to the empty set. To this end, we define a function upto that constructs the set $\{0, 1, \ldots, n\}$.

```
let rec upto n = if n < 0 then S.empty else S.add n (upto (n-1))
```

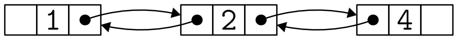
The main part of the program retrieves the value of N from the command line and then displays the result obtained by count.

```
let () =
  let n = int_of_string Sys.argv.(1) in
  Format.printf "%d@." (count (upto (n - 1)) S.empty S.empty)
```

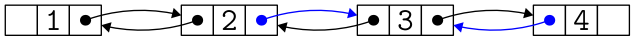
We can thus enumerate the 365,596 solutions of the 14-queens problem in less than a minute.

Additional Information

The data structures that you find in the literature on algorithms are, for the most part, *imperative* in nature: They are modified in-place by the operations they provide. Arrays and linked lists are the first examples that come to mind. Thus, writing to an array's cell replaces the previous value by a new one. Something similar occurs when we add an element to a linked list such as:



The operation consists in modifying two of the pointers, as follows:

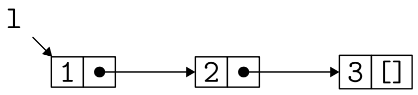


As in case of the array, we have modified the structure of the list *in-place*. In both cases, it is possible to return to the previous state, but it requires another in-place modification.

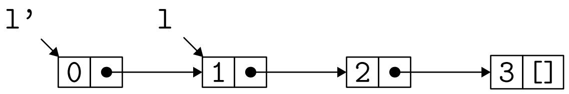
A *persistent* data structure, by contrast, is a data structure whose operations never modify their arguments; *new* values are returned instead. Of course, we could render a data structure persistent by making a copy of it systematically, but this would be grossly inefficient. There are other, more efficient means of rendering an imperative data structure persistent. See, for instance, section *4.4 Persistent Arrays*.

There is an entire class of data structures for which persistence is both possible and efficient, namely, *immutable* data structures, which cannot be modified once constructed. For such data structures, it is possible to avoid unnecessary copies through sharing. This is easiest to illustrate in case of lists.

If we define a list `l` by `let l = [1; 2; 3]`, then, in terms of its memory representation, `l` is a pointer. It points to a first block that contains `1` and a pointer to a second block, and so on and so forth:



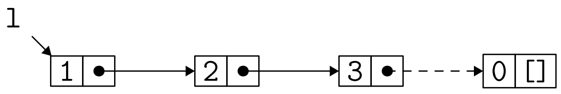
If we now define a list `l'` by adding another element to the list `l`, with the declaration `let l' = 0 :: l`, we have the following situation:



The application of the constructor `::` has the effect of allocating a new block, whose first element is `0` and whose second element is a pointer having the same value as `l`. The variable `l` continues to point to the same blocks as before. In general, any function that we may write on lists has the property of not modifying the lists passed as arguments.

It is very important to understand that the above involves *sharing*. The declaration of `l'` allocates a single block (because only one constructor is applied), and the blocks forming `l` are reused without modification. We have two lists of three and four elements respectively, namely `[1;2;3]` and `[0;1;2;3]`, but only four memory blocks. In particular, *no copy* is made. In general, OCaml never makes copies of values, unless you explicitly write a function to do so. Such a function would be useless in case of lists because a list cannot be modified in-place. Functions that make copies of values are not useful unless the data structures involved are mutable.

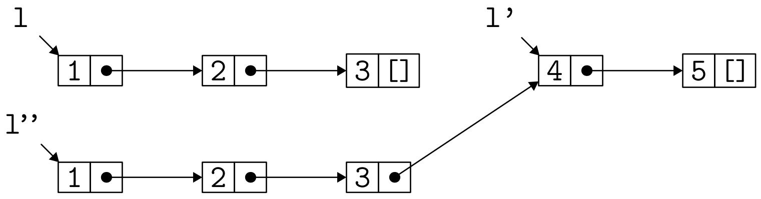
It is now clear why it is harder to add an element to the end of a list rather than to its head. This is because an in-place modification of the list `l` would be required:



To add an element to the end of a list, we must make a copy of every block of the list. This is in fact what is done by the following function `append`, which concatenates two lists (see exercise 2.24):

```
let rec append l1 l2 = match l1 with
  | [] -> l2
  | x :: l -> x :: append l l2
```

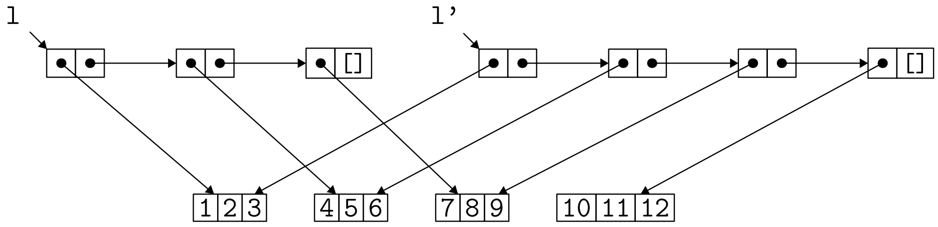
We observe that this function creates as many blocks as there are in `l1` and shares only those of `l2`. Thus, if we declare `let l' = [4; 5]` and concatenate `l` and `l'` with `let l'' = append l l'`, we will have the following situation:



Copies of the the blocks of `l` have been made and those of `l'` have been shared.

Accordingly, lists must be used whenever the relevant operations are addition and removal from the front (*stack* structure). When we need to access and/or make modifications at arbitrary positions, it is preferable to use another data structure.

Crucially, the *elements* of the list are not themselves copied by the function `append`. In fact, the variable `x` in the code of `append` denotes an element of an arbitrary type and is not itself copied. This does make a difference when dealing with lists of integers. However, if we have a list `l` with three elements of a more complex type, for example the list `[(1,2,3); (4,5,6); (7,8,9)]`, then these will be shared between `l` and `append l [(10,11,12)]`:



All this may appear unnecessarily expensive if we are used to lists that are modified in-place, which is the traditional approach in the context of imperative languages. However, this would be to underestimate the practical advantages of persistence. Besides, it is important to note that the concept of persistence can be easily implemented in an imperative language. We only have to manipulate linked lists exactly as the OCaml compiler does. Conversely, we can certainly manipulate lists that are modifiable in-place in OCaml, for example by defining the following type:

```
type 'a mlist = Nil | Cons of 'a * 'a mlist ref
```

Here, the second argument of the constructor `Cons` is a reference. Unlike imperative languages, OCaml provides the possibility of defining immutable data structures in a natural manner that is also safe. (Even if we code a persistent data structure in C, the type system cannot prevent in-place modification, since all data in C is mutable).

Finally, we must not forget that unused memory is automatically recovered. Consider for instance an expression such as:

```
let l = [1;2;3] in append l [4;5;6]
```

The three blocks of `l` are effectively copied during the construction of the list `[1;2;3;4;5;6]` but may be recovered immediately as they are no longer referenced anywhere. (Chapter 3 offers more details on memory management.)

The Practical Advantages of Persistence

Persistence has many practical advantages. To begin with, it aids reasoning about the code and its correctness: Since the values manipulated by the program are immutable, we can reason mathematically about them. We can use equational reasoning without even worrying about the evaluation order. It is thus easy to verify the correctness of the function `append` once we have stated what it is supposed to do (namely, that `append l1 l2` constructs the list formed from the elements of `l1`, followed by the elements of `l2`). A simple recursion on the structure of `l1` suffices. If, however, we are dealing with lists that are modifiable in-place and a function `append` that modifies the last pointer of `l1` to make it point to `l2`, then the correctness argument is clearly more difficult. The contrast is even more flagrant when a list is to be reversed. The correctness of a program is important and must always take precedence over its efficiency. After all, nobody would want a program that is fast but incorrect.

Persistence is not only useful for correctness, it is also a powerful tool in contexts where *backtracking* is necessary. Suppose, for example, we write a program to find the way out of a maze using a function `find` that takes as argument a persistent state and returns a boolean indicating a successful search. A function `possible_moves` returns the possible moves from a particular state as a list. As the datatype is persistent, another function, `move`, returns the state resulting from a move and the current state. We also assume that a boolean function `is_exit` returns whether a particular state corresponds to the exit. We can then write the function `find` in the following way:

```
let rec find e =
  is_exit e || try_move e (possible_moves e)
and try_move e = function
  | [] -> false
  | d :: r -> find (move d e) || try_move e r
```

Here, `try_move` is a function that tries the possible moves in a list, one by one. It is the persistence of the data structure encoding the state that allows us to write such concise code. If the state was a global data structure modified in-place, we would have to perform the move before calling `find` recursively in `try_move`. We would also have to *undo* this move in case of failure before trying other possible moves. The code would then look something like this:

```
let rec find () =
  is_exit () || try_move (possible_moves ())
and try_move = function
  | [] -> false
  | d :: r -> (move d; find ()) || (undo_move d; try_move r)
```

This is indubitably less clear and more error-prone. Note that this is not an artifical example; *backtracking* is a programming technique regularly used when implementing graph traversals, coloring, enumeration of solutions, etc.

Consider a second example of the usefulness of persistence. Imagine a program that manipulates a database. At any moment, there is only one instance of this database. A priori, there is therefore no need to use a persistent data structure for it. Let us assume that the modifications performed on the database are complex, that is, involving a large number of operations, some of which can fail. We then find ourselves in a difficult situation if we need to *undo* the effects already performed before the failure. Schematically, the code could look like this:

```
  try
    ... perform modifications ...
```

```
with e ->
  ... restore the database to a coherent state,
      then handle the error ...
```

If we use a persistent data structure for the database, it may be stored in a reference. Let us call it `db`. The operation that modifies the database sets the value of this reference:

```
let db = ref ( ... initial database ... )
...
try
  db := ... operation that modifies the database !db ...
with e ->
  ... handle the error ...
```

From then on, there is no need to undo anything. The modification, however complex it may be, constructs a brand new database. Only once this construction is complete is the reference `db` modified so that it points to the new database. This final modification is atomic and cannot fail. If any exception is raised during the modification itself, the reference `db` remains unchanged. The automatic memory manager will then recover the memory that was unnecessarily allocated during the modification.

Interfaces and Persistence

The datatype of lists is obviously persistent because it is a type with a known definition, that is, it is concrete and immutable. When an OCaml module implements a data structure, declared as an *abstract* type, its persistent or imperative nature is not evident. Of course, the programmer can be informed of this fact by means of an appropriate comment in the interface. In practice, however, it is the types of the operations that provide this information. Consider the example of a persistent data structure that represents finite sets of integers. The interface of such a module looks like this:

```
type set
val empty : set
val add : int -> set -> set
val remove : int -> set -> set
...
```

The persistent nature of sets is implicit in the interface: The operations `add` and `remove` return a value of type `set`, that is, a new set. The fact that the empty set `empty` is a constant and not a function makes the persistence even clearer. All occurrences of `empty` will be shared irrespective of its representation. This would not be possible with a mutable data structure.

A data structure for sets of integers that are modified in-place would instead have an interface of the form:

```
type set
val create : unit -> set
val add : int -> set -> unit
val remove : int -> set -> unit
...
```

Here, the function for addition, `add`, does not return anything because it adds an element in-place to the data structure, and it is the same for the other operations. The value `empty` is replaced by a *function* `create` that takes an argument of type `unit`. Each call to `create` must construct a new instance of the data structure so that the in-place modifications of one do not affect the others.

Unfortunately, the type system of OCaml does not prevent the mixing of both kinds of data structures. Thus, you can give the type `int -> set -> set` to a function that adds an element to a set through side effects, for example by returning the set passed as argument. Conversely, you can give the type `unit -> set` to a function `empty` that returns a persistent empty set. In both cases, it is useless and potentially dangerous.

This does not mean, however, that a persistent data structure must necessarily be coded without any side effects. The good definition of *persistent* is:

$$\textit{persistent} = \textit{observationally immutable}$$

In other words, it is not *purely applicative*, in the sense of an absence of side effects. We only have an implication in one direction:

$$\textit{purely applicative} \Rightarrow \textit{persistent}$$

The opposite is false: There are persistent data structures that make use of side effects. This book contains several examples of such data structures. The situation is illustrated in figure 2.15.

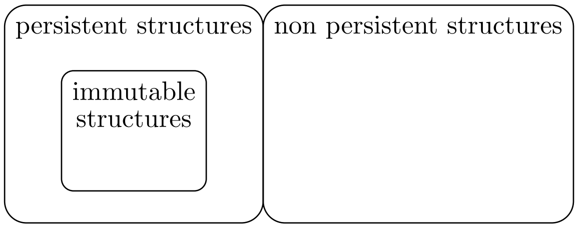


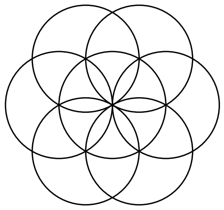
Figure 2.15: Different classes of data structures.

2.15 Exercises

Drawing a Cardioid

2.1 Modify program 2 to plot the set of points in figure 2.1.

2.2 Write a program that draws the figure below.

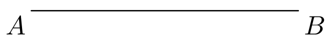


The Mandelbrot Set

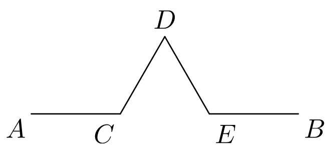
2.3 Modify the drawing of the Mandelbrot set by coloring points that are not in the set based on the first value of i for which $x_i^2 + y_i^2 > 4$. The color may be chosen, for example, by linearly interpolating between two predefined colors, and then using the function `Graphics.rgb` (see section *2.10 Breakout without Bricks*) and a simple rule of three.

2.4 Modify the code of the preceding exercise as follows: Once the drawing is done, choose a point in the set using the mouse. Redo the drawing centered around this point and at one-tenth the scale.

2.5 Write a program that draws a Koch snowflake. The snowflake is obtained by tracing three Koch curves along the three sides of an equilateral triangle. A Koch curve of depth n between two points A and B is defined as follows: For $n = 0$, it is the segment linking A and B.

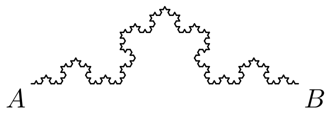


For $n > 0$, the segment $[A, B]$ is divided into three segments of equal length, $[A, C]$, $[C, E]$, and $[E, B]$. A point D is defined as the third vertex of an equilateral triangle CED:



We then draw four Koch curves of depth $n - 1$ along the four segments $[A, C]$, $[C, D]$, $[D, E]$, and $[E, B]$.

By choosing an initial depth that is sufficiently large, we get a drawing of the following shape:

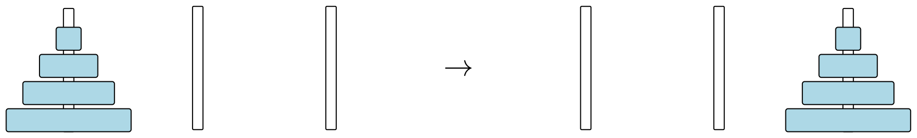


2.6 Let u_0 be an integer greater than 1 and (u_n) the sequence defined by:

$$u_{n+1} = \begin{cases} u_n/2 & \text{if } u_n \text{ is even} \\ 3u_n + 1 & \text{otherwise} \end{cases}$$

Write a program that reads the value of u_0 using `read_int` and displays the successive values of the sequence (u_n) as long as $u_n > 1$. (The Collatz conjecture states that for any value of u_0, we always reach the cycle $1 \to 4 \to 2 \to 1 \to \ldots$. Stated in 1928, this conjecture continues to defy the best efforts of mathematicians.)

2.7 The problem of the tower of Hanoi consists in shifting n discs of decreasing diameter, stacked one on top of the other, from one location to another, by using a third location temporarily. Only one disc may be moved at a time, by taking it off the top of a stack, and it may only be placed over a larger disc.



Write a program `hanoi` that reads the value of n and then displays a valid sequence of moves to solve the problem, for example as follows:

```
> ./hanoi
4
moving disk from 1 to 3
moving disk from 1 to 2
moving disk from 3 to 2
moving disk from 1 to 3
...
```

Sieve of Eratosthenes

2.8 Write a function `sum: int array -> int` that calculates the sum of the elements of an array, first with a `for` loop, then with a recursive function.

2.9 Write a function `occurs: int array -> int -> bool` that, for an array of integers a and an integer value v, determines if v appears in a. Write two variants of the program, one with a `while` loop and the other with a recursive function.

2.10 Write a function `binary_search: int array -> int -> bool` that, for an array of integers a sorted in increasing order, and an integer value v, determines if v appears in a. Use binary search so as to have complexity $O(\log n)$, where n is the length of the array a.

2.11 Write a function `shuffle: int array -> unit` that randomly shuffles the elements of an array using the following algorithm, called the "Knuth shuffle," where n is the size of the array:

```
for $i$ from 1 to $n-1$
  let $j$ be a random integer between 0 and $i$ (inclusive)
  exchange the elements at indices $i$ and $j$
```

You may use `Random.int` k to obtain a random integer between 0 and $k-1$. Of course, this algorithm may be used for arrays of any type.

Drawing a Curve

2.12 Write a function `sum: (int -> int) -> int -> int -> int` that takes a function f, and two integers i and j as arguments, and calculates the following sum:

$$\sum_{k=i}^{k=j} f(k).$$

2.13 Write a function `dicho` of type:

```
dicho : (float -> float) -> float -> float -> float
```

that searches for the zero of a monotonic function f on an interval $[a,b]$ by dichotomic search. The idea is as follows: Calculate the midpoint x of $[a,b]$, and compare the signs of $f(a)$ and $f(x)$. Depending on the result, repeat the calculation on the interval $[a,x]$ or $[x,b]$. Stop when a and b are sufficiently close to each other. You can assume that $[a,b]$ contains a zero of f.

Copying a File

2.14 Modify program 7 so that it correctly handles the case in which one of the two files cannot be opened. This triggers an exception of the form `Sys_error` *s*, where *s* is a string explaining the nature of the error (file does not exist, permission denied, etc.). This message should be displayed before terminating the program using `exit 1`.

2.15 In languages like C and Java, the statement `break` exits the nearest enclosing loop, and the statement `continue` jumps to the next iteration of the loop. Thus, the following C code displays `124567`:

```
for (int i = 0; i < 10; i++) {
  if (i == 3) continue;
  if (i == 8) break;
  printf("%d", i);
}
```

Explain how to translate these two statements into OCaml. You may use two exceptions, `Break` and `Continue`, to this end.

Inverting the Lines of a Text

2.16 Write a function `mult : int list -> int` that calculates the product of all the elements of a list of integers. Make sure to return 0 at the first occurrence of the integer 0.

2.17 Redo the previous exercise, this time raising an exception if 0 is encountered, so as to avoid unnecessarily multiplying the first elements of the list.

2.18 Redo exercise 2.16 by writing the function `mult` as a tail-recursive function.

2.19 Write a function `insert : int -> int list -> int list` that inserts an integer into a list of integers sorted in increasing order. Avoid making an unnecessary copy of the part of the initial list that lies beyond the point of insertion.

2.20 Using the above function `insert`, write a function `insertion_sort : int list -> i` that sorts a list of integers in increasing order using following algorithm, known as *insertion sort:* Beginning with the empty list, insert each element successively, using the function `insert`. Insertion sort is described in detail in chapter 12.

Converting Integers from an Arbitrary Base

2.21 Write a function `mem : 'a -> 'a list -> bool` that determines whether an element is present in a list, using the equality operator `=`. This function exists in the standard library, under the name `List.mem`.

2.22 Write a function `count : 'a -> 'a list -> int` that counts the number of occurrences of an element in a list, using the equality operator `=`.

2.23 Rewrite the function `count` of the last exercise using an iterator.

2.24 Write a function `append : 'a list -> 'a list -> 'a list` that concatenates two lists. If $l_1 = [x_1; \dots; x_n]$ and $l_2 = [y_1; \dots; y_m]$, then the result of `append` l_1 l_2 must be the list $[x_1; \dots; x_n; y_1; \dots; y_m]$ of length $n + m$. Proceed by recursion on the first list, thus performing n recursive calls. The OCaml standard library provides this function both under the name `List.append` and as the infix operator `@`.

2.25 Using the above function `append`, write a function `rev` of type `'a list -> 'a list` that reverses the order of elements of a list, that is, such that `rev` $[x_1; \dots; x_n] = [x_n; \dots; x_1]$. In the next section, we will discuss a more efficient method of implementing this function.

2.26 Write a function `subseq: 'a list -> 'a list -> bool` that determines if a list w_1 is a sub-sequence of another list w_2, that is, if w_1 can be obtained by removing zero or more elements of w_2. For example, $[1; 5; 4; 1]$ is a sub-sequence of $[3; 1; 1; 5; 0; 4; 1]$.

2.27 Write a function `exists: ('a -> bool) -> 'a list -> bool` that takes a boolean function p and a list l as arguments and determines if l contains at least one element x such that $p(x)$ is `true`.

2.28 Using `List.fold_left`, write a function `forall: ('a -> bool) -> 'a list -> boo` that determines whether all the elements of a list of type `'a list` satisfy a condition given by a function of type `'a -> bool`. What is the problem with this solution? Propose a more efficient version.

2.29 Write a function `filter: ('a -> bool) -> 'a list -> 'a list` that takes a boolean function p and a list l as arguments, and returns the list of all the elements x of l for which $p(x)$ is `true`. Preserve the order of the elements. This function exists in the standard library under the name `List.filter`.

2.30 Using an iterator, write a function `max_seq: bool list -> int` that returns the length of the longest sequence of consecutive `true` values in a given list.

2.31 Write a function `first: ('a -> bool) -> 'a list -> 'a` that returns the first element of the list passed as argument satisfying the given condition. If such an element does not exist, the function should raise the exception `Not_found`.

2.32 Write an iterator `fold_pairs` of type:

```
fold_pairs: ('a -> 'b -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

The iterator takes as argument a function f, a value v, and a list $[x_1; \ldots; x_n]$, and returns:

$$f(f(\ldots f(f(v, x_1, x_2), x_2, x_3), \ldots, x_{n-1}, x_n), x_n, x_1)$$

In other words, the function traverses all pairs of consecutive elements (x_i, x_{i+1}), where the list is considered to be circular.

2.33 Propose a function to create a matrix `init_matrix`, analogous to `Array.init`, of type:

```
init_matrix: (int -> int -> 'a) -> int -> int -> 'a array array
```

2.34 Write a traversal function:

```
iter_matrix: (int -> int -> 'a -> unit) -> 'a array array -> unit
```

that applies a given function to all the elements of a matrix. The first two arguments of the function are the indices of the corresponding element. Explain why it is more efficient to traverse the matrix row-wise rather than column-wise.

Breakout without Bricks

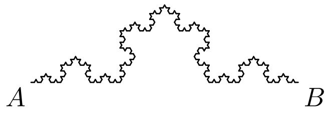
2.35 Given a file `a.ml` that contains `let () = Printf.printf "A\n"` and a file `b.ml` that contains `let () = Printf.printf "B\n"`, compile these two files separately and verify that different results are obtained depending on the order in which the files are passed to the linker.

2.36 Given six files `a.mli`, `a.ml`, `b.ml`, `c.ml`, `d.mli`, and `d.ml`, and knowing that `A` and `B` depend on `C`, and that `C` depends on `D`, give all possible ways of compiling and linking a program containing the four compilation units `A`, `B`, `C`, and `D`.

2.37 Rewrite the code of Breakout in a single file with two submodules `Draw` and `Breakout`. Give the module `Draw` the same interface as that defined in `draw.mli`.

Logo Turtle

2.38 Redo exercise 2.5 that draws a Koch snowflake using the Logo turtle.



Hint: Write a recursive function `von_koch: int -> float -> unit` whose first parameter is the number of recursive steps and the second the length of the segment $[A, B]$.

2.39 Rewrite the functor `Turtle` so that it is also parametrized by a module of signature:

```
val open_graph: int -> int -> unit
val draw_line: int -> int -> int -> int -> unit
```

Here, `open_graph` $w\ h$ opens a window of dimensions $w \times h$ and `draw_line` $x_1\ y_1\ x_2\ y_2$ draws a segment between the points (x_1, y_1) and (x_2, y_2). Then, implement a module with such an interface using `Graphics`.

2.40 We would like to define a module to represent files endowed with reading and writing permissions. To this end, we introduce a type `permission`.

```
type permission = Read | Write
```

Define a module `Access` to manipulate access rights. This module will provide an abstract type `t`, a value of type `t`, `default`, representing read and write access, and two functions `get` and `set` that allow, respectively, to get and set a permission.

```
val get : t -> permission -> bool
val set : t -> permission -> bool -> t
```

Next, define another module `File` to represent a file endowed with access rights. This module will export a private record type `t`, containing the file name in a field of type `string` and the access rights in a `mutable` field of type `Access.t`. You will provide a function `create : string -> t` taking the file name as argument, a function `chmod` to modify the access rights, a function `open_in : t -> in_channel` to open a file for reading, and a function `open_out : t -> out_channel` to open a file for writing. The last two functions will raise an exception `PermissionDenied` in case of insufficient permissions.

2.41 Write multiple modules to manipulate monetary values in different currencies. All the modules should have the same signature:

```
module type MONEY = sig
  type t = private { i : int; f : int }
  val create : int -> int -> t
  val add : t -> t -> t
end
```

A monetary value is represented by an integer `i` and a fraction `f`, between 0 and 99, representing the number of cents. The integer `i` is signed. Write a module `Money` implementing the signature `MONEY`. Derive from it two modules `Euro` and `Dollar`, both having the signature `MONEY` and the structure `Money`, but with distinct types `Euro.t` and `Dollar.t`. Finally, write a function `euros_to_dollars : float -> Euro.t -> Dollar.t` that, given an exchange rate, converts euros to dollars.

2.42 Write a module to represent sets of integers by lists sorted in increasing order, with the following signature:

```
module type ISET = sig
  type t
  val empty : t
  val add : int -> t -> t
  val union : t -> t -> t
  val mem : int -> t -> bool
end
```

What is the advantage of making the type `t` an abstract type?

2.43 Rewrite the code of the previous exercise in the form of a functor parametrized by the type of elements and a function defining a total order on them, that is:

```
module type ELT = sig type t val compare: t -> t -> int end
module Set(E: ELT) : ... = struct ... end
```

Here, the value of `compare` x y is strictly negative if $x < y$, zero if $x = y$, and strictly positive if $x > y$.

2.44 Apply the functor of the previous exercise to obtain sets of even integers, giving these an arbitrary total order.

2.45 Write a data structure for polynomials of one variable. The idea is to represent a polynomial using the sorted list of its monomials, each monomial being represented by a pair of a coefficient and an exponent. Thus, the polynomial $X^7 - 3X^4 + 2$ is represented by the list $[(1,7); (-3,4); (2,0)]$. In order to be generic with respect to the ring of coefficients, write the structure of polynomials as a functor `Poly` parametrized by a module of signature:

```
module type Ring = sig
  type t
  val zero : t
  val one : t
  val add : t -> t -> t
  val mul : t -> t -> t
  val equal : t -> t -> bool
end
```

If `R` designates the argument of the functor `Poly`, the latter must provide the following functions:

```
type t
val create: (R.t * int) list -> t
val add: t -> t -> t
val eval: t -> R.t -> R.t
```

Here, `t` is the type of polynomials, `create` constructs a polynomial based on a list of monomials, `add` adds two polynomials, and `eval` evaluates a polynomial at a point. Hint: Begin with the function `add` (see exercise 2.42) and then use it to write the function `create`.

2.46 Continue the previous exercise by adding two constants, `zero` and `one`, and two functions, `mul` and `equal`, to the signature of the functor `Poly`. This yields polynomials endowed with the structure of a ring. In other words, the signature of the functor `Poly` can be written in terms of the signature `Ring`. The construct `include` is a syntactic shortcut to include the definition of one signature in another:

```
sig
  include Ring
  val create: (R.t * int) list -> t
  val eval: t -> R.t -> R.t
end
```

It is thus possible to obtain polynomials of several variables by applying the functor `Poly` several times in succession. For instance, the structure of polynomials of three variables and integer coefficients is obtained as:

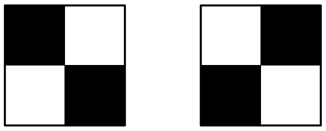
```
module P = Poly(Poly(Poly(Int)))
```

Here, it is assumed that `Int` implements the signature `Ring` for the integers.

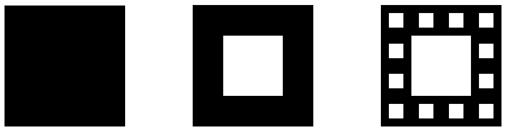
Quadtrees

2.47 Write a function `rotate: quad -> quad` that turns an image anticlockwise by 90 degrees.

2.48 Write a function `mirror: quad -> quad` that constructs the mirror symmetry along the vertical axis of an image represented by a quadtree. Thus, the function `mirror` allows switching between the following two images:



2.49 Write a function `fractal: int -> quad` that constructs a quadtree representing a Sierpiński carpet fractal, by iterating the following process of decomposition:



Solving the N-Queens Problem

2.50 Modify program 16 to return the first solution found, in the form of an array a giving the column $a[i]$ of the queen located on row i. When there is no solution, the exception `Not_found` should be raised.

3 OCaml Concepts: A Closer Look

What do you do if your program is ill-typed or consumes too many resources (memory or time)? To make the right diagnosis, it is necessary to understand the functioning of the compiler and the code it produces.

In this chapter, we address problems related to type-checking, memory use, and execution time.

3.1 Type-Checking Algorithm

While the compiler automatically infers the types of all values defined in a program, it may nevertheless be useful for programmers to have at least an intuitive understanding of the typing rules of OCaml. This would help them understand any error messages that may arise.

To compute the type of an expression, it suffices to traverse its definition from left to right, beginning with the most generic type possible, `'a`, and accumulating type constraints.

Consider, for example, the following function definition:

```
let f x y = if x y then [] else [y + 1]
```

We begin by rewriting this definition as follows:

```
let f = fun x -> fun y -> if x y then [] else [y + 1]
```

Next, we let:

```
val f : 'a
```

Since `f` is an expression of the form `fun x -> e1`, we deduce that `'a` must be the type of a function, `'a1 -> 'a2`:

```
val f : 'a1 -> 'a2
```

Similarly, `e1` is a function of the form `fun y -> e2`, so that `'a2` is equal to a type `'b1 -> 'b2`:

```
val f : 'a1 -> 'b1 -> 'b2
```

We now extract the type constraints on the variables `'a1`, `'b1`, and `'b2` originating from the expression `e2`, which is equal to:

```
if x y then [] else [y + 1]
```

From the function application `x y`, we deduce that `x` is a function. In other words, `'a1` is equal to a type `'c1 -> 'c2`. We therefore get:

```
val f : ('c1 -> 'c2) -> 'b1 -> 'b2
```

Next, since `y` is passed as argument to `x`, we deduce that the variable `'c1` is equal to `'b1`, so that the type of `f` is:

```
val f : ('b1 -> 'c2) -> 'b1 -> 'b2
```

Furthermore, since `x y` is used as the condition in an `if` construct, we deduce that `'c2` is equal to `bool`:

```
val f : ('b1 -> bool) -> 'b1 -> 'b2
```

The first branch of the `if` returns the empty list `[]`. Therefore, `'b2` is equal to `'d list`:

```
val f : ('b1 -> bool) -> 'b1 -> 'd list
```

Finally, the second branch `[y + 1]` provides the last two constraints. From the addition `y + 1`, we deduce that `y` is of type `int`. The list `[y + 1]` therefore has the type `int list`, and the variable `'d` is equal to `int`. The complete type of the function `f` is therefore:

```
val f : (int -> bool) -> int -> int list
```

Let us now study the case of an ill-typed function. The function `g` below takes two arguments, `x` and `y`. The type inference algorithm detects the following error:

```
# let g x y = if x y then [x] else [y] ;;
Error: This expression has type 'a but an expression was expected
       of type 'a -> bool
       The type variable 'a occurs inside 'a -> bool
```

The initial steps in the typing of `g` are the same as those in case of `f`. Thus, having scrutinized the expression `x y`, we deduce that `g` has the following type:

```
val g : ('b1 -> bool) -> 'b1 -> 'b2
```

The first branch of the `if` returns the list `[x]`. Therefore, the variable `'b2` must be equal to `('b1 -> bool) list`. Similarly, the second branch returns the list `[y]`. So `'b2` must be of type `'b1 list`. Therefore, for `g` to be well typed, the types `('b1 -> bool) list` and `'b1 list` must be equal. This would require the equation of types `'b1 = 'b1 -> bool` to have a solution, which is not possible in OCaml (unless we enable the option `-rectypes` of the compiler or interpreter). We now understand the above error message: It indicates that the variable `y` is of type `'a`, whereas it should have type `'a -> bool`, which is not possible because the variable `'a` appears within the type `'a -> bool`.

In general, to determine the type of a program, or understand its typing errors, it is necessary to know the type constraints imposed by the constructs of the language. Here we present a (non-exhaustive) list of these constraints.

If-then-else

In a conditional `if e1 then e2 else e3`, the expression `e1` must be of type `bool`, and the expressions `e2` and `e3` must be of the same type, whatever that may be. The type of the conditional is that of `e2` (and therefore also that of `e3`).

If-then

A conditional expression `if e1 then e2` (without the `else` branch) has type `unit`. The expression `e1` must be of type `bool` and `e2` of type `unit`.

While

In a loop `while e1 do e2 done`, the expression `e1` must be of type `bool` and `e2` of type `unit`. The `while` loop is of type `unit`.

For

In a loop `for i = e1 to e2 do e3 done`, the expressions `e1` and `e2` must be of type `int` and the expression `e3` of type `unit`. The `for` loop is of type `unit`.

Sequence

In a sequence `e1; e2`, the expression `e1` must be of type `unit` while `e2` may be of any type. The type of `e2` gives the type of the sequence. If `e1` is not of type `unit`, we may write `let _ = e1 in e2` or `ignore e1; e2`.

Declaration

In a *local* declaration `let p = e1 in e2`, the pattern `p` and the expression `e1` must be of the same type. The expression `e2` may be of any type. This is also the type of the declaration. In a *global* declaration `let p = e`, the type of `p` must be that of `e`.

Function

A function of the form `function p -> e` or `fun p -> e` is of type `t1 -> t2`, where `t1` is the type of the pattern `p` and `t2` is the type of the expression `e`.

Application

The application `e1 e2` requires that `e1` be of type `t1 -> t2` and that `e2` be of type `t1`. The type of the result of the application is `t2`.

Pattern Matching

Consider the pattern matching construct:

```
match e with
  | p1 when b1 -> e1
  | ...
  | pk when bk -> ek
```

The associated constraints are that the expressions `e1`, . . . , `ek` must all have the same type, and that the expressions `b1`, . . . , `bk` must all be of type `bool`. The patterns `p1`, . . . , `pk` must all have the same type as `e`. The global type of this construct is that of the expressions `e1`, . . . , `ek`.

Exception

Consider a `try-with` block of the form:

```
try
  e
with
  | p1 -> e1
  | ...
  | pk -> ek
```

Here, the expressions `e`, `e1`, . . . , `ek` must all be of the same type. The patterns `p1`, . . . , `pk` must designate exceptions (of the predefined type `exn`). The type of the `try-with` block is that of the expression `e`.

3.2 Runtime Model

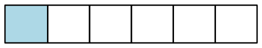
As a rule, to make the best use of a programming language, it helps to understand its runtime model. OCaml is no exception in this regard. By runtime model, we mean the time (see the next section) and space complexity of the different constructs of the language. Even if we can often ignore these complexities by remaining at a higher level of abstraction, it is worthwhile to understand them.

Value Representation

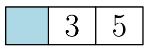
We begin by explaining the memory representations of OCaml values. An OCaml value occupies exactly one memory word (32 or 64 bits, depending on the architecture). If we consider this word as an integer, we may distinguish two cases, depending on whether the integer is even or odd:

- if it is an odd integer $2n + 1$, we interpret it as the integer value n;
- if it is an even integer, we interpret it as a pointer.

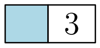
In short, an OCaml value is either an integer or a pointer. If it is an integer, it can represent a value of type `int`, a character, a boolean, or even a constant constructor like `[]` or `None`. Note that this explains the limitation of the type `int` to 31 bits (63 in a 64-bit machine), mentioned previously. If the value is a pointer, it points to a *block* allocated in memory. A block occupies $k + 1$ memory words, with $k \geq 1$. The first word of the block, called the *header*, plays a particular role that we will explain later. The following k words contain OCaml values, and we say that k is the size of the block. We will represent a block of size 5 as follows:



Thus, the pair `(1,2)` is represented by a pointer to a block containing two values, namely, the integer values 3 and 5 representing respectively the integers 1 and 2:



More generally, an n-tuple is represented by a pointer to a block of size n. Arrays and records are also represented in memory by pointers to blocks. Thus, the array `[|1; 2|]` and the record `{a=1; b=2}` have *exactly* the same representation in memory as the pair `(1,2)`. We explained previously that a reference was nothing but a particular case of records, with a single field `contents` (see the additional information in section 2.6 *Tracing a Curve*). We deduce from this that the value `ref 1` is represented in memory in the form of a block of size 1, that is:

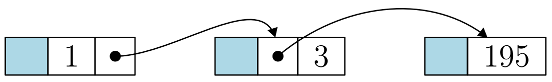


The `mutable` nature of the field `contents` means that the value contained in this block, here 3, can be modified in place. The same is true for the values of an array.

Let us now consider the representation of concrete types. The idea is simple: Constant constructors are represented by integers and other constructors by (pointers to) blocks. Consider, for example, the type:

```
type t = A | B of t * t | C | D of char
```

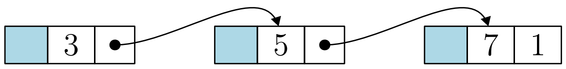
The two constant constructors `A` and `C` are represented by the integers 1 and 3, respectively. The two non-constant constructors `B` and `D` are represented by pointers to blocks of size 2 and 1, respectively. Thus, the value `B (A, B (D 'a', C))` corresponds to three blocks, as follows:



Here, the value 195 corresponds to $2 \times 97 + 1$, where 97 is the code of the character `'a'`.

For faster access, a pointer to a block points to its first field rather than its header. The latter contains the tag of the constructor, here `B` or `D`. This information is used particularly when pattern matching.

Consider the type of lists as a second example: The constant constructor `[]` is represented by the integer 1. The constructor `::` is represented by (a pointer to) a block of size 2. The list `1 :: 2 :: 3 :: []` is therefore represented in memory as follows:

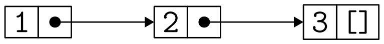


We observe that OCaml lists are not fundamentally different from the linked lists used in C or Java. When an OCaml program manipulates lists (passing them as arguments of functions, returning them, etc.), it only manipulates the pointers to the blocks, exactly like a C or Java program would. The essential difference is that OCaml only permits the construction of well-formed lists since pointers are not explicit in the language. Where programmers using C or Java have to *think* about checking whether a pointer is `null`, programmers using OCaml would use a pattern matching construct that would *force* them to consider this case, and with a very concise syntax at that.

In general, the header contains the tag of the block. Beyond the cases we have already illustrated, the header can also identify a floating-point number, an array of floats, a string, an object, a function, etc. Note that the tag provides far less information than the type of the value. In general, the memory representation of OCaml values does not include typing information. The OCaml standard library provides a module `Obj` that can be used to explore this representation. We can thus test whether a value is represented as a block and, if so, retrieve its tag, size, fields, etc. This module must be used with caution given its strongly untyped nature.

Notation

In what follows, we will simplify the graphical representation of OCaml values: Headers will not be drawn, and integer values will not be transformed using $n \mapsto 2n + 1$. Thus, the list `[1;2;3]` will be represented simply as:



Comparison Operators

The equality operator = that we have already used several times may be applied to any two values, provided that they are of the same type. Thus, we can use it to compare values of the following type:

```
# type ty = D | A of int | B of bool | C;;
type ty = D | A of int | B of bool | C
# A 1 = A 1 ;;
- : bool = true
# D = B true ;;
- : bool = false
```

This operator compares the structures of two values and therefore corresponds to the usual mathematical notion of equality. It is called the *structural equality* operator. Its negation is written as <>.

OCaml also provides another equality operator, ==. It is called the *physical equality* operator. Its negation is written as !=. The operator == compares values as integers, even if these integers represent memory addresses. Accordingly, we can observe that two constructor applications allocate two different memory blocks:

```
# A 1 == A 1 ;;
- : bool = false
```

Now that we know how OCaml values are represented, we can further deduce that = and == coincide on values like integers, characters, and constant constructors. In case of more complex types, predicting the result of a comparison using physical equality is more difficult. It would be necessary to have a precise understanding of how the compiler manages memory allocation.

The physical equality operator *always* finishes in constant time, whereas structural equality can take an arbitrary amount of time and even loop on cyclic data. Let us illustrate this point with an example. We construct a cyclic list l using a feature of OCaml that permits us to define a value using let-rec:

```
# let rec l = 1 :: l ;;
l : int list = [1; 1; 1; ... ]
```

The evaluation of `l == l` terminates whereas that of `l = l` does not.

```
# l == l ;;
- : bool = true
# l = l ;;
```

OCaml provides other structural comparison operators: on the one hand, `<`, `>`, `<=`, and `>=` of type `'a -> 'a -> bool`; and on the other hand, an order relation `Stdlib.compare` of type `'a -> 'a -> int`, compatible with the preceding operators.

```
# C < A 1 ;;
- : bool = true
# A 1 < B true ;;
- : bool = true
# Stdlib.compare (A 1) (B true);;
- : int = -1
```

The comparison operators are based on the uniform representation of values explained previously. However, their definition remains arbitrary and entirely linked to the internal representation of values in the language. It remains, nevertheless, useful to have a "generic " order relation, even if it is arbitrary.

For example, we can use `Stdlib.compare` to construct complex structures that require an order relation, such as sets:

```
# module S = Set.Make(
     struct type t = ty
            let compare = Stdlib.compare
     end);;
```

Here, the definition of `Stdlib.compare` is not important as long as it is an order relation. However, if a specific notion of comparison is needed, it would have to be defined.

Other Generic Functions

OCaml provides other generic functions, such as:

- a hash function `Hashtbl.hash` of type `'a -> int`;
- marshalling functions, mentioned previously (see the additional information in section 2.7 *Copying a File*).

Automatic Memory Management

As in other languages, like Lisp and Java, memory management in OCaml is automated. This means that the programmer explicitly indicates allocations (by constructing an array or a record, applying a constructor, etc.) but not *deallocations*. Deallocation is performed automatically by the *garbage collector* (GC). The functioning of the GC is intimately linked with the representation of OCaml values. On the one hand, the GC can distinguish between pointers and other values by examining the parity of the value. On the other, in case of a block, the header contains the information needed by the GC: the tag (as was explained above), the size of the block, and a few bits to mark the block as being in use. Thus, in a 32-bit architecture, the header is of the following form:

31 . . . 10	9 8	7 . . . 0
size	GC	tag

As the size of a block is encoded in 22 bits, it is relatively limited (up to $2^{22} - 1$, that is 4 194 303). This is, for instance, the maximum size of OCaml arrays. It can be obtained using `Sys.max_array_length`. The maximum size of strings is four times larger, because strings have a more compact representation than arrays. Thus, `Sys.max_string_length` $= 2^{24} - 5 = 16\,777\,211$. In a 64-bit machine, by contrast, these limits are well beyond what can be allocated in memory (respectively $2^{54} - 1$ and $2^{57} - 9$).

Representation of Functions

Whenever a function may be used as a first-class value (see section 2.6 *Tracing a Curve*), it is represented by a pointer to a block. We speak then of *closure*. A closure contains two things:

- a pointer to the code to be executed;

- the values of variables susceptible of being used by this code, termed the environment.

The construction of a closure therefore consists in allocating a block whose size is determined by that of the environment, that is, by the number of free variables in the body of the function. Conversely, the application of a closure f to a value v consists in calling the function indicated by the code pointer of f and passing two arguments to the function, namely, the environment (in the form of the closure f itself) and the value v. Consider the following example:

```
let diff_quotient dx f x = (f (x +. dx) -. f x) /. dx
let derivative = diff_quotient 1e-10
let my_cos = derivative sin
```

The value `derivative` is a function obtained by partially applying the function `diff_quotient`. It is represented by a closure whose code pointer corresponds to `fun f -> fun x -> ...`, and whose environment contains the value of `dx`, namely, `1e-10`. Similarly, the value of `my_cos` is a closure whose code pointer corresponds to `fun x -> ...`, and whose environment contains the values of `dx` (namely, `1e-10`) and `f` (namely, `sin`). The value `sin` is another example of a closure, whose environment is empty.

3.3 Analyzing the Execution Time of a Program

From the preceding discussion, we may deduce a certain number of facts concerning the time complexity of OCaml operations:

- The application of a non-constant constructor is proportional to the number of fields, since it consists in assigning the constructor arguments to the fields of a block. As the number of arguments of a constructor is known statically, this operation takes constant time and space.

- The application of a function also takes constant time, irrespective of how the function was constructed. Of course, we speak here of the application operation only, and not of the cost of evaluating the arguments or the function itself; these may be arbitrarily expensive.

In what follows, we offer several practical and theoretical tools to measure and analyze the execution time of a program.

Measuring Execution Time

It is easy to measure the total execution time of a program using a watch or, better yet, with a tool provided by the operating system. The execution time of *part* of an OCaml program, can be measured just as easily, and with great precision, using the function `Unix.times` of the module `Unix`. This function returns a record whose field `tms_utime` contains the time that has elapsed since the start of the program. This is a floating-point number, expressed in seconds, with precision greater than a millisecond. By calculating the difference between the two values returned by this function at two different points of the program, we deduce the precise execution time of that part of the code. We would typically write something like:

```
let start = (times ()).tms_utime in
...
let stop = (times ()).tms_utime in
```

We can then calculate the elapsed time using `stop -. start`. Exercises 3.3 and 3.4 focus on writing generic code for such measurements.

Informal Notions of Complexity

It is useful to measure the execution time of a program to evaluate its efficiency, for example, to compare it with another program, as well as to determine whether it can be used to solve a given problem. If we write a program that sorts an array of n entries, we may measure its execution time for small values of n and then try to extrapolate to larger values. If, for example, we are able to sort $n = 1\,000$ entries in a hundredth of a second, and if we observe that the execution time is multiplied by a factor of 4 each time that n is doubled, then we may estimate that we would need over two and a half hours to sort a million entries.

The *complexity* of a program, or more generally of an algorithm, is defined as the number of elementary operations that it must perform as a function of the

	$f(n) =$					
n	$\log n$	n	$n \log n$	n^2	n^3	2^n
10^1	–	–	–	–	–	–
10^2	–	–	–	–	–	∞
10^3	–	–	–	–	1 s	∞
10^4	–	–	–	–	17 m	∞
10^5	–	–	–	10 s	12 j	∞
10^6	–	–	–	17 m	∞	∞
10^7	–	–	–	1 j	∞	∞
10^8	–	–	2 s	116 j	∞	∞
10^9	–	1 s	21 s	∞	∞	∞

Figure 3.1: Execution time of $f(n)$ operations, assuming a billion operations per second. Note that "–" indicates that the time is less than one second and "∞" that it is greater than one year.

size of the input data. In the above example, the complexity may be expressed as a function f of the number n of entries: $f(n)$ is the number of elementary operations performed to sort n inputs. By elementary operations, we mean the operations that are atomic for the machine, for example, addition. If we assume that a machine these days performs a billion operations per second, figure 3.1 gives the execution time of $f(n)$ operations for different functions f.

In practice, the precise nature of the elementary operations and their respective cost is not important. We can term any operation whose time cost is constant an elementary operation. This includes arithmetic operations, function and constructor applications, assignments, etc. It suffices to retain the following principle: A million elementary operations can be performed instantaneously, a billion in a few dozen seconds, and a trillion over several hours.

Complexity Analysis

Rather than experimental evaluations, we can analyze the code of a program to calculate the number of elementary operations that it will perform. Let us suppose, for example, that our sorting program is written as two nested loops, such as:

```
for i = 1 to n - 1 do
  for j = i downto 1 do
    ...
```

Here, we assume that the omitted code (denoted by `...`) contains only a finite number of elementary operations, say C. We can then deduce that the total complexity is exactly:

$$f(n) = \sum_{i=1}^{n-1} C \times i = C\frac{n(n-1)}{2}.$$

This confirms our experimental observation that doubling n causes $f(n)$ to be multiplied by approximately four. In practice, it is rarely this simple to analyze the complexity. Certain operations are only performed in certain cases, depending on checks performed by the program, which in turn depend on the inputs. We generally settle for the calculation of the *worst-case* complexity, which provides an upper bound on the total number of operations performed. We may also calculate the *average* complexity over the set of possible inputs of size n, for a given distribution of these inputs.

Big O Notation

In practice, we use the big O notation to bound the complexity of a program. To express that a function f, defined on the natural numbers, does not grow faster than another function g, also defined on the natural numbers, we write:

$$f(n) = O(g(n))$$

This means that there is a constant C such that $f(n) \le Cg(n)$ for large enough n, or, more formally:

$$\exists C \exists N, \forall n \ge N, f(n) \le Cg(n)$$

If $f(n)$ denotes the complexity of an algorithm in terms of a parameter n, we say that "the complexity of the algorithm is in $O(g(n))$" if $f(n) = O(g(n))$. In the example involving sorting, the complexity is in $O(n^2)$.

An algorithm whose complexity is in $O(1)$ is executed in constant time, independently of the size of the data. It therefore involves a constant number of elementary operations, for example, two additions and a multiplication. An algorithm is said to be: *logarithmic* if its complexity is in $O(\log n)$; *linear* if it is in $O(n)$; *linearithmic* if it is in $O(n \log n)$; *quadratic* if it is in $O(n^2)$ and, more generally, *polynomial* if there is a constant k such that its complexity is in $O(n^k)$; and *exponential* if it is in $O(2^n)$.

In general, we do not seek to exhibit the constants C and N hidden in the notation O. We are only concerned with the asymptotic order of magnitude, for example, when deciding whether one algorithm is better than another. Of course, for a *specific* value of n, an algorithm in $O(n^2)$ can certainly take less time to execute than another in $O(\log n)$, by virtue of the constants behind the O's.

Amortized Complexity

It is often worthwhile to calculate the total complexity of a sequence of n operations and then divide by n, in order to obtain the average cost of each operation. This average cost may prove inferior to the worst-case complexity. We call the average complexity thus obtained *amortized complexity*, as opposed to the worst-case complexity. We give examples of amortized complexity in chapters 4 and 6.

3.4 Exercises

Type-checking Algorithm

3.1 Are the following functions well typed? If so, what is their type? If not, explain why.

```
let f1 x = let y = !x + 2 in y
let f2 n = for i = 1 to n do i + 4 done
let f3 m n = if n = m then 1 else 2
let f4 l = match l with [] -> [] | y :: s -> [s]
let f5 x y = let z = x + 1 in y || z > 10
let f6 x = let z = ref x in z + 1
let f7 x y = for i = x to 10 do y := x :: !y done; !y
let f8 x y z = x y z
```

3.2 Consider the two functions below:

```
let foo x = x.a := (snd x.b) + 1
let bar x = x.c <- (fst x.b) :: x.c
```

Define `'a t` as a record type so that we have:

```
val foo : 'a t -> unit
val bar : 'a t -> unit
```

Analyzing the Execution Time of a Program

3.3 Write a function `time: (unit -> unit) -> float` that measures the execution time of the function passed as argument. Use the function `Unix.times` as explained above.

3.4 Write a function `time5: (unit -> unit) -> float` that does the following: executes the function passed as argument five times, measuring the execution time each time; eliminates the smallest and largest of the values obtained; and returns the mean of the three remaining values.

3.5 Write a program that produces the table in figure 3.1.

Part II

Data Structures

4 Arrays

One of the simplest data structures is the array, represented by the OCaml type `array`. In this chapter, we present several variations on this data structure. We show how to construct resizeable arrays, compact arrays of booleans, arrays with an efficient concatenation operation, and persistent arrays.

4.1 Resizeable Arrays

A resizeable array is an array whose size can be increased or decreased at any time by means of an operation `resize`. The signature of such arrays is given in program 17. (We will use resizeable arrays in chapters 5 and 6.)

The idea behind the implementation of resizeable arrays is simple: We use a normal array to store the elements, and if it becomes too small, we allocate a larger array into which we copy the existing elements. To avoid spending too much time allocating and copying, we allow the backing array to be larger than needed, with the elements beyond a certain index left unused.

The type of resizeable arrays is therefore a record containing an ordinary array in a field `data` and the number of elements in use in a field `size`.

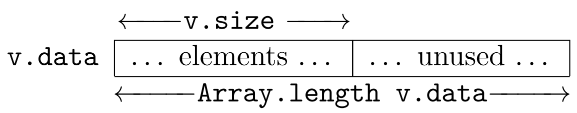
```
  type 'a t = {
```

Program 17 — Minimal Signature for Resizeable Arrays

```
module type ResizeableArray = sig
  type 'a t
  val length : 'a t -> int
  val make : int -> 'a -> 'a t
  val resize : 'a t -> int -> unit
  val get : 'a t -> int -> 'a
  val set : 'a t -> int -> 'a -> unit
end
```

```
    mutable size: int;
    mutable data: 'a array;
  }
```

Thus, if `v` is such a resizeable array, we may represent the situation as follows:



Accordingly, we will maintain the following invariant throughout:

$$0 \leq \texttt{v.size} \leq \texttt{Array.length v.data}$$

If we wish to increase the size of the array `v`, there are two possibilities to consider. If the new size does not exceed the capacity of the array `v.data`, it suffices to modify `v.size`. Otherwise, we allocate a new, sufficiently large array and copy the elements of `v.data` into it. We then assign this new array to `v.data`. Whenever we wish to decrease the size of the array `v`, it suffices to modify `v.size`.

There is, however, a slight difficulty regarding this last point. Pointers to elements that are no longer in use should not be retained, so that the GC is able to collect them whenever possible. We therefore need a value of the right type to replace the elements that are no longer in use. To this end, we will require that a default value be provided when the array is created. We will save it in a third field of the record. We thus arrive at the following type:

```
type 'a t = {
  default: 'a;
  mutable size: int;
  mutable data: 'a array;
}
```

Let us now write the code for the different operations. The length of the resizeable array is given by the field `size`.

```
let length a = a.size
```

To create a new resizeable array, it suffices to use `Array.make`, and to store the size and the default value.

```
let make n d = { default = d; size = n; data = Array.make n d }
```

In general, we will maintain the following invariant on the type `t`: Every element beyond index `size`, that is, every unused element, contains the value stored in the field `default`.

To access the `i`-th element of a resizeable array `a`, it is necessary to verify that the index is valid, as the array `a.data` may contain more than `a.size` elements.

```
let get a i =
  if i < 0 || i >= a.size then invalid_arg "get";
  a.data.(i)
```

Note that once this verification has been done, we could use `Array.unsafe_get` to access `a.data` for greater efficiency. The assignment operation is analogous:

```
let set a i v =
  if i < 0 || i >= a.size then invalid_arg "set";
  a.data.(i) <- v
```

All the subtlety lies in the function `resize`, which modifies the size of a resizeable array `a`. Several scenarios are possible. If the new size `s` is less than

Program 18 — Resizeable Arrays

```
type 'a t = {
  default: 'a;
  mutable size: int;
  mutable data: 'a array;
}

let length a = a.size

let make n d = { default = d; size = n; data = Array.make n d }

let get a i =
  if i < 0 || i >= a.size then invalid_arg "get";
  a.data.(i)

let set a i v =
  if i < 0 || i >= a.size then invalid_arg "set";
  a.data.(i) <- v

let resize a s =
  if s <= a.size then
    Array.fill a.data s (a.size - s) a.default
  else begin
    let n = Array.length a.data in
    if s > n then begin
      let n' = max (2 * n) s in
      let a' = Array.make n' a.default in
      Array.blit a.data 0 a' 0 a.size;
      a.data <- a'
    end
  end;
  a.size <- s
```

or equal to `a.size`, it suffices to replace the elements between `s` and `a.size-1` by `a.default`.

```
  let resize a s =
    if s <= a.size then
      Array.fill a.data s (a.size - s) a.default
```

We could also reallocate the elements into a smaller array (see exercise 4.2). If, however, `s` is larger than `a.size`, it is necessary to check whether the array `a.data` is large enough to contain `s` elements. We therefore compute the size `n` of the array `a.data` and compare it with `s`.

```
    else begin
      let n = Array.length a.data in
      if s > n then begin
```

If `s` is greater than `n`, it is necessary to resize the array. Here, we choose the strategy of doubling the size of the array `a.data`. (See exercise 4.1 for another strategy.) Since that may not suffice, we calculate the new size as the maximum of `2 * n` and `s`.

```
        let n' = max (2 * n) s in
```

To be perfectly safe, it would be necessary to limit the new size to the maximum size of arrays, that is, `Sys.max_array_length`, and to fail if the value `s` is too large. This is left as an exercise for the reader.

We then allocate a new array of size `n'` into which we copy the elements of `a.data` that are in use. For this, we use the function `Array.blit`, which copies a portion of one array into another.

```
        let a' = Array.make n' a.default in
        Array.blit a.data 0 a' 0 a.size;
```

Then, we replace the array `a.data` by the new array.

```
        a.data <- a'
      end
    end;
```

When `s <= n`, there is nothing more to be done. Finally, in all scenarios, we end by updating the field `size`.

```
    a.size <- s
```

This completes the implementation of resizeable arrays. The code is given in full in program 18 (see page 166).

Complexity

Resizeable arrays are typically used when the number of elements to be stored is not known beforehand (see exercise 4.3). Let us suppose that we begin with a resizeable array of length 0 and increase its length n times by one unit, to end up with an array of length n. The naive strategy of giving the array `data` exactly the same length as the resizeable array entails that each `resize` with a length i would have cost i. The total cost would then be quadratic:

$$1 + 2 + \cdots + n = \frac{n(n+1)}{2} = O(n^2).$$

However, the strategy implemented by `resize` here is more subtle, consisting in (at least) doubling the size of the array whenever it needs to be increased. The total cost is then linear, which may be shown as follows: Let us assume, without loss of generality, that $n \geq 2$. Let $k = \lfloor \log_2(n) \rfloor$, that is, $2^k \leq n < 2^{k+1}$. Beginning with an empty array, we will have executed a total of $k+2$ resizings to arrive at an array `data` of the final size 2^{k+1}. After the i-th resizing, where $i = 0, \ldots, k+1$, the size of the array is 2^i. The i-th resizing therefore has cost 2^i. The total cost is then:

$$\sum_{i=0}^{k+1} 2^i = 2^{k+2} - 1 < 4n.$$

In other words, some of the `resize` operations have constant cost (when resizing is not actually necessary). Others, by contrast, have non-constant cost, but the total complexity remains linear. Averaging over the n operations, it is as if each addition had constant cost. The extension of a resizeable array by one unit therefore has *amortized complexity* $O(1)$.

4.2 Bit Vectors

If a program needs to manipulate a set of integers included within a reasonable interval, and if persistence is not required, then an array of booleans may be used. This is what we did, for instance, in section 2.5 *Sieve of Eratosthenes*. However, given that a boolean is represented by one memory word (32 or 64 bits depending on the architecture), a lot of memory is wasted. While a single bit per element would suffice, an array of booleans occupies 32 or 64 times that amount. The bit vector data structure is useful precisely in this regard: it enables the use of a single bit per element.

The signature of such a data structure is exactly the same as that of arrays, except that elements are of type `bool`. The beginning of such a signature is given in program 19 (see below).

The implementation consists in using an ordinary array of integers and storing several booleans in each of the integers: 31 booleans in a 32-bit architecture and 63 booleans in a 64-bit architecture. (Recall that OCaml reserves one bit of each integer for use by the GC.) We begin by letting `bpi` be the number of booleans stored in each integer.

```
let bpi = Sys.word_size - 1
```

The maximum size of a bit vector is therefore given by the following value:

```
let max_length = Sys.max_array_length * bpi
```

To represent a bit vector, we introduce the following type `t`:

```
type t = {
  length : int;
  bits : int array;
}
```

The field `length` contains the actual number of booleans present in the bit vector, as this is not necessarily a multiple of `bpi`. The field `bits` contains the array of integers. If $b_0, b_1, \ldots, b_{\texttt{bpi}-1}$ are the first `bpi` elements of an array of booleans, they are represented in the first cell of the array `bits` by the integer

$$b_0 + 2b_1 + \cdots + 2^{\texttt{bpi}-1} b_{\texttt{bpi}-1}$$

Program 19 — Minimal Signature of Bit Vectors

```
module type Bitv = sig
  type t
  val create : int -> bool -> t
  val length : t -> int
  val get : t -> int -> bool
  val set : t -> int -> bool -> unit
end
```

The following `bpi` elements are represented in the same manner by the second integer of the array `bits`, and so forth. When the number of elements is not a multiple of `bpi`, some of the high-order bits of the last integer of the array are not used. To allow comparing bit vectors structurally, we enforce the following invariant on the representation:

$$\textit{unused high-order bits are 0} \tag{4.1}$$

We can thus use the OCaml operations `=` or `<>` on bit vectors, irrespective of how they are constructed.

Creation

Let us begin with the function `create`, which constructs a bit vector of size `n`, where all the elements have the same value `b`. Except for the last integer, all the elements of the array `bits` will contain the same value: 0 if `b` is `false` or -1 if `b` is `true`. This is because in the two's-complement representation, -1 is written in binary using only the digit 1, that is, $-1 = (1 \cdots 1)_2$. The function `create` therefore begins by defining this initial value, 0 or -1, depending on the value of `b`.

```
let create n b =
  let initv = if b then -1 else 0 in
```

The number of elements of the array `bits` is then determined by the Euclidean division of `n` by `bpi`.

```
    let q = n / bpi and r = n mod bpi in
```

If the remainder `r` is 0, then all elements of the array are fully used, and it suffices to construct an array of size `q` initialized with `initv`.

```
    if r = 0 then
      { length = n; bits = Array.make q initv }
```

If the remainder `r` is not 0, we need to construct an array of $\mathtt{q}+1$ integers. All the elements contain `initv`, except for the last one, whose high-order bits must be 0 to respect the invariant. More precisely, if `b` is `true`, then the last element of the array must have exactly `r` low-order set bits. This is easily obtained using the expression `(1 lsl r) - 1`. This completes the function `create`:

```
    else begin
      let a = Array.make (q + 1) initv in
      if b then a.(q) <- (1 lsl r) - 1;
      { length = n; bits = a }
    end
```

The complete code is given in program 20.

Reading

To read the n-th element of a bit vector we must first determine the index i of the corresponding integer in the array, and then the index j of the corresponding bit within this integer. The values of i and j are obtained from the Euclidean division of n by `bpi`. Given the i-th element of the array, in order to extract the j-th bit, it suffices to shift this integer j positions to the right using `lsr`, and then test its least significant bit. We end up with the following function:

```
  let get v n =
    let i = n / bpi and j = n mod bpi in
    (v.bits.(i) lsr j) land 1 <> 0
```

Program 20 — Creation of a Bit Vector

```
let create n b =
  let initv = if b then -1 else 0 in
  let q = n / bpi and r = n mod bpi in
  if r = 0 then
    { length = n; bits = Array.make  q initv }
  else begin
    let a = Array.make  (q + 1) initv in
    if b then a.(q) <- (1 lsl r) - 1;
    { length = n; bits = a }
  end
```

Equivalently, we could also perform a bitwise *and* operation against the integer containing a single set bit in the j-th position, which is obtained using the expression `1 lsl j`:

```
    v.bits.(i) land (1 lsl j) <> 0
```

Writing

To assign a value `b` of type `bool` to the n-th element of a bit vector `v`, we begin by determining its position exactly as we did for reading, via the Euclidean division of n by `bpi`.

```
  let set v n b =
    let i = n / bpi and j = n mod bpi in
```

We then consider the two possible cases separately. If `b` is `true`, that is, if the j-th bit of `v.bits.(i)` must be set to 1, we perform a *logical or* against the integer containing a single set bit in the j-th position.

```
    if b then
      v.bits.(i) <- v.bits.(i) lor (1 lsl j)
```

Program 21 — Reading and Writing in a Bit Vector

```
let get v n =
  let i = n / bpi and j = n mod bpi in
  (v.bits.(i) lsr j) land 1 <> 0

let set v n b =
  let i = n / bpi and j = n mod bpi in
  if b then
    v.bits.(i) <- v.bits.(i) lor (1 lsl j)
  else
    v.bits.(i) <- v.bits.(i) land lnot (1 lsl j)
```

Otherwise, we must set the j-th bit to 0, which can be done using a *logical and* against an integer with all bits set, except the j-th.

```
  else
    v.bits.(i) <- v.bits.(i) land lnot (1 lsl j)
```

The complete code of the functions `get` and `set` is given in program 21. The two operations clearly have cost $O(1)$.

Set-Theoretic Operations

Bit vectors come into their own when they are used to represent sets. In particular, the union, intersection and complement operations can be implemented efficiently by the corresponding bitwise operations.

Consider, for example, the *logical and* operation of two bit vectors, `inter`, which also computes the intersection of the corresponding sets. This operation is only meaningful when the two bit vectors are of the same size. We begin by verifying this.

```
let inter v1 v2 =
  let l1 = v1.length in
  if l1 <> v2.length then invalid_arg "Bitv.inter";
```

Program 22 — Operations *and* and *not* on bit vectors

```
let inter v1 v2 =
  let l1 = v1.length in
  if l1 <> v2.length then invalid_arg "Bitv.inter";
  let b = Array.mapi (fun i ei -> ei land v2.bits.(i)) v1.bits in
  { length = l1; bits = b }

let normalize v =
  let r = v.length mod bpi in
  if r > 0 then
    let s = Array.length v.bits - 1 in
    v.bits.(s) <- v.bits.(s) land (1 lsl r - 1)

let compl v =
  let b = Array.map lnot v.bits in
  let r = { length = v.length; bits = b } in
  normalize r;
  r
```

Once we know that both vectors are of the same size, we implement the *logical and* operation itself. For this, we may use the function `Array.mapi`, which constructs an array by applying a function to all the elements of another array.

```
let b = Array.mapi (fun i ei -> ei land v2.bits.(i)) v1.bits in
{ length = l1; bits = b }
```

This completes the function `inter`. We can similarly write the function `union`, which computes the set-theoretic union, by performing a *logical or*.

Note that the invariant (4.1) is respected, because *and* and *or* preserve the unused high-order 0s. This is not the case with bitwise operations like *negation* or *exclusive or*. In these cases, it is necessary to re-establish the invariant by zeroing the unused high-order bits *a posteriori*. Let us write a function `normalize` to do this. We begin by letting `r` be the number of bits in use in the last integer of the array.

```
let normalize v =
  let r = v.length mod bpi in
```

If this number is not zero, we must zero out all bits of index `r` and higher. This can be implemented using a mask containing exactly `r` low-order set bits, that is, `1 lsl r - 1`.

```
  if r > 0 then
    let s = Array.length v.bits - 1 in
    v.bits.(s) <- v.bits.(s) land (1 lsl r - 1)
```

This completes the function `normalize`.

We can easily derive from this a function `compl` that computes the bitwise negation of a bit vector, that is, the complement of the corresponding set.

```
let compl v =
  let b = Array.map lnot v.bits in
  let r = { length = v.length; bits = b } in
  normalize r;
  r
```

The functions `inter` and `compl` are summarized in program 22 (see page 174). They have cost $O(n)$ in time and space, where n is the size of the bit vectors concerned.

Among other useful set-theoretic operations, we may also consider iteration over all elements. For a bit vector, this boils down to iterating over the set of indices for which the corresponding boolean is `true`. Such an iteration may be expressed as a function of the form:

```
val iteri_true : (int -> unit) -> t -> unit
```

To implement this function, we iterate over the elements of the array using `Array.iteri`. For each element, we iterate over the set bits with a loop that successively tests each bit. We obtain the following code:

```
let iteri_true f v =
  Array.iteri
```

Program 23 — Iteration Over the Set Bits of a Bit Vector

```
let iteri_true f v =
  Array.iteri
    (fun i ei ->
       let index = i * bpi in
       let rec visit x =
         if x <> 0 then begin
           let b = x land -x in
           f (index + ntz b);
           visit (x - b)
         end
       in
       visit ei)
    v.bits
```

```
    (fun i ei ->
       let index = i * bpi in
       for j = 0 to bpi - 1 do
         if ei land (1 lsl j) <> 0 then f (index + j)
       done)
    v.bits
```

It is not necessary to consider the case of the final element of the array separately, because the invariant guarantees precisely that the unused bits are 0.

Although correct, this solution is relatively inefficient, because it successively tests *all* bits of the array. We would like the cost of the iteration to decrease if there are fewer set bits. In other words, we need an efficient means of iterating over the set bits of an integer.

As it so happens, the two's-complement representation allows extracting the low-order set bit of an integer `x` easily by performing a *logical and* of `x` and `-x` (see exercise 4.5). This gives an efficient means of iterating over the set bits of an integer, by successively extracting and zeroing each set bit in increasing order.

There remains, however, one difficulty: The expression `x land -x` does not return the *index* i of the low-order set bit in `x`, but instead returns 2^i. We therefore need to calculate the base 2 logarithm of this value or, equivalently, the number of low-order 0s. For this, let us suppose that we are given a function `ntz`, for *number of trailing zeros*, that calculates the number of low-order 0s of an integer.

We may then rewrite the part of the code of `iteri_true` that examines the set bits of `ei` with the help of a recursive function `visit` instead of a `for` loop.

```
let rec visit x =
  if x <> 0 then begin
    let b = x land -x in
    f (index + ntz b);
    visit (x - b)
  end
in
visit ei
```

The function `visit` ends as soon as its argument is 0, that is, when it does not contain any set bit. Otherwise, we extract the low-order set bit, `b`. We call `f` on the corresponding index `index + ntz b`, and then call `visit` again on `x-b`, which is `x` with the `b`-th bit zeroed out. The complete code of `iteri_true` is given in program 23 (see page 176). The function `ntz` is the focus of exercise 4.6.

Finally, let us write a function `cardinal: t -> int`, which calculates the number of set bits in the bit vector, that is, the cardinal of the set. It suffices to calculate the sum of the number of set bits in each element of the array. If we assume we are given a function `pop` of type `int -> int` that calculates the number of set bits of an integer, then the function `cardinal` is easily obtained.

```
let cardinal v =
  Array.fold_left (fun n x -> n + pop x) 0 v.bits
```

The function `pop` (short for *population count*) remains to be written. We can write it by extracting the bits one by one, again thanks to the expression `x land -x`:

```
let pop x =
  let rec count n x =
    if x = 0 then n else count (n + 1) (x - (x land -x)) in
  count 0 x
```

We can render this function more efficient by tabulating it (see exercise 4.12).

Other Operations on Bit Vectors

To be complete, a bit vector library should also provide other operations. On the one hand, it should provide the same operations that we use for arrays, like `append`, `sub`, `fill`, and `blit`. On the other hand, it should also offer arithmetic operations that interpret a bit vector as an integer of n bits, such as addition, shifts, and conversions to and from the different integer types of OCaml. Some of these operations are proposed in exercises 4.8–4.11.

For Further Information

Warren's *Hacker's Delight* [24] details numerous techniques to manipulate the bits of an integer, such as the ones we have used to implement Patricia trees and bit vectors. In particular, the book gives several variations of the functions `ntz` and `pop`.

4.3 Ropes

This section presents the *rope* data structure. This data structure was originally introduced as an alternative to strings, to address the following issues:

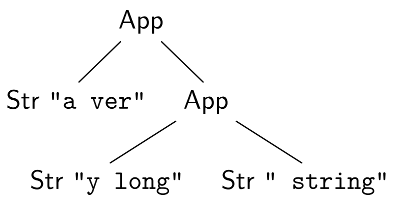
- Strings are not persistent; they are immutable in OCaml, but this is not necessarily the case in other languages, like C for instance.
- Strings are limited in size, especially when OCaml is used on a 32-bit machine, where the limit is $2^{24} - 5$ characters.

- Concatenation and substring extraction have a significant cost, particularly in space, as they involve copying the contents of the underlying string.

To address these issues, the rope data structure draws on a simple idea: A rope is nothing but a binary tree whose leaves are (ordinary) strings and whose inner nodes should be interpreted as concatenations. A rope therefore corresponds to the following type:

```
type t =
  | Str of string
  | App of t * t
```

For example, one possible value by which the string `"a very long string"` may be represented is the following:



Two considerations lead us to refine the proposed type `t`. On the one hand, numerous algorithms require efficient access to the length of a rope, particularly to decide whether to descend along the left or the right subtree of an `App` node. It is therefore advisable to decorate each internal node with the length of the rope. On the other hand, it is important to share substrings between ropes themselves as well as with the ordinary strings used to construct the ropes. Hence, rather than having the `Str` node point to a complete OCaml string, we prefer to have it designate a substring of an OCaml string. A substring of a string s is represented by a triple (s, o, n), denoting `s`$[o..o + n - 1]$, that is, the portion of s of length n starting at offset o. We thus obtain the following type:

```
type t =
  | Str of string * int * int
  | App of t * t * int
```

A node `App`(r_1, r_2, n) represents the concatenation of the two ropes r_1 and r_2, whose total length is n. We could also have stored the sizes of r_1 and r_2 in the node. However, retaining only the total size is more economical in terms of memory space, and there is no loss of efficiency in practice.

Genericity

We notice that the notion of rope is not related to that of string. Instead of the type `string`, we could equally use arrays or lists of characters or, more generally still, sequences of values of a type other than that of characters. In fact, all we need is a data structure of *sequences* of a certain *type of characters*. We can then construct ropes whose leaves consist of these sequences.

The resulting data structure has *the same signature* as that with which we started, namely, the signature of sequences for the same type of characters.

Program 24 gives a minimal signature `STRING` for a data structure of strings. The abstract type `t` is that of strings, and the type `char`, that of their characters. This signature contains only the operations that are needed to implement the rope structure. They are compatible with those of the OCaml module `String`.

Program 24 — Generic String Signature

```
module type STRING = sig
  type t
  type char
  val length : t -> int
  val empty : t
  val make : int -> char -> t
  val get : t -> int -> char
  val append : t -> t -> t
  val sub : t -> int -> int -> t
end
```

We introduce a signature `ROPE` for ropes in program 25. This signature includes the module `S` of strings that will be the leaves of the ropes. The

Program 25 — Signature of Ropes

```
module type ROPE = sig
  module S : STRING
  include STRING with type char = S.char
  val of_string : S.t -> t
  val set : t -> int -> char -> t
  val delete_char : t -> int -> t
  val insert_char : t -> int -> char -> t
  val insert : t -> int -> t -> t
end
```

signature `ROPE` provides the same operations as those of strings, for the same type of characters, which can be translated as follows:

```
include STRING with type char = S.char
```

Ropes are immutable. Update operations (`set`, etc.) return new ropes.

The rope structure is naturally implemented as a functor parametrized by a module `X` of signature `STRING`:

```
module Make(X : STRING) : (ROPE with module S = X) = struct
```

The module obtained by applying this functor has the signature `ROPE with module S = X`, which specifies that its module `S` is the module `X` passed as argument of the functor. The code thus begins by defining `S` to be equal to `X`:

```
module S = X
```

The type of characters of the rope is the same as that of strings:

```
type char = S.char
```

The type of ropes is as presented earlier:

```
type t =
  | Str of S.t * int * int
  | App of t * t * int
```

We will enforce several invariants on this type. On the one hand, for every rope of the form `Str (s, o, n)`, we have the inequalities:

$$0 \leq \texttt{o}, \quad 0 \leq \texttt{n} \quad \text{and} \quad \texttt{o} + \texttt{n} \leq \texttt{S.length s}$$

On the other hand, for every rope of the form `App (u, v, n)`, we have the inequalities:

$$0 < \texttt{length u}, \quad 0 < \texttt{length v} \quad \text{and} \quad \texttt{n} = \texttt{length u} + \texttt{length v}$$

Basic Operations

By definition of the type `t`, the length of a rope can be computed in constant time.

```
let length = function
  | Str (_,_,n)
  | App (_,_,n) -> n
```

The construction of the empty rope is straightforward using the empty string:

```
let empty =
  Str (S.empty, 0, 0)
```

More generally, the rope corresponding to a string `s` is given by:

```
let of_string s =
  Str (s, 0, S.length s)
```

Importantly, the type `S.t` must itself be persistent in order to guarantee both the persistent nature of ropes and the correctness of rope operations. This is the case when `S.t` is the type `string`, as strings in OCaml are immutable.

To access the i-th character of a rope, it suffices to go down the tree until the target leaf is reached. We assume that the index is valid. The recursive part of the operation is then written as follows:

```
let rec unsafe_get t i = match t with
  | Str (s, ofs, _) ->
      S.get s (ofs + i)
```

Program 26 — Basic Operations on Ropes

```
module Make(X : STRING) : (ROPE with module S = X) = struct

  module S = X

  type char = S.char

  type t =
    | Str of S.t * int * int
    | App of t * t * int

  let empty = Str (S.empty, 0, 0)

  let length = function
    | Str (_,_,n)
    | App (_,_,n) -> n

  let of_string s = Str (s, 0, S.length s)

  let make n c = of_string (S.make n c)

  let rec unsafe_get t i = match t with
    | Str (s, ofs, _) ->
        S.get s (ofs + i)
    | App (t1, t2, _) ->
        let n1 = length t1 in
        if i < n1 then unsafe_get t1 i else unsafe_get t2 (i - n1)

  let get t i =
    if i < 0 || i >= length t then invalid_arg "get";
    unsafe_get t i
    | App (t1, t2, _) ->
        let n1 = length t1 in
        if i < n1 then unsafe_get t1 i else unsafe_get t2 (i - n1)
```

We see here the importance of being able to retrieve the size of `t1` in constant time. The complexity of this operation is therefore bound by the height of the tree. As we will see later, ropes can be balanced so as to minimize this height.

We define a function `get` that also checks whether the index is valid, as a wrapper around `unsafe_get`.

```
let get t i =
  if i < 0 || i >= length t then invalid_arg "get";
  unsafe_get t i
```

These basic operations on ropes are presented together in program 26 (see page 183).

Concatenation

A priori, the concatenation of two ropes `t1` and `t2` is as simple as applying the constructor `App` and calculating the total length, that is:

```
let append t1 t2 =
  App (t1, t2, length t1 + length t2)
```

We note that this operation thus takes constant time. However, iterated concatenations may cause the number of nodes, and hence the height of the tree, to increase rapidly, to the detriment of other operations.

Two ideas allow us to control the number of nodes and the height of the tree. The first idea consists in actually concatenating "small" leaves (i.e. short strings) when they are found side by side in the tree. We may choose, for example, to concatenate the leaves `s1` and `s2` in the three situations illustrated in figure 4.1. The first situation corresponds to the concatenation of a rope whose right branch is the small leaf `s1`, with another small leaf `s2`. The second corresponds to the concatenation of two small leaves, `s1` and `s2`. The third is the mirror image of the first.

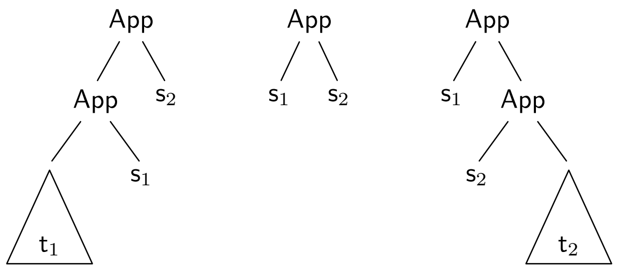


Figure 4.1: Possible situations with adjacent leaves.

To determine whether the leaves are small enough to be concatenated, we use a constant:

```
let small_length = 256
```

The constant is arbitrary and should be chosen depending on the specific use of the rope structure. Exercise 4.15 proposes making this constant a parameter of the functor.

To implement the concatenation of two leaves, we define the following general function that concatenates fragments of two strings, `s1` and `s2`:

```
let append_string s1 ofs1 len1 s2 ofs2 len2 =
  Str (S.append (S.sub s1 ofs1 len1) (S.sub s2 ofs2 len2),
       0, len1 + len2)
```

We now give the code of the function `append`, which implements the concatenation of two ropes `t1` and `t2`.

We begin by considering the case in which one of the two ropes is empty.

```
let append t1 t2 = match t1, t2 with
  | Str (_,_,0), t | t, Str (_,_,0) ->
      t
```

Next, we consider the three possible cases in which two small leaves are found side by side. The simplest case is that in which `t1` and `t2` are the leaves in question. We then use the function `append_string`:

```
| Str (s1, ofs1, len1), Str (s2, ofs2, len2)
 when len1 <= small_length && len2 <= small_length ->
    append_string s1 ofs1 len1 s2 ofs2 len2
```

The two other cases are similar:

```
| App (t1, Str (s1, ofs1, len1), _), Str (s2, ofs2, len2)
  when ... -> ...
| Str (s1, ofs1, len1), App (Str (s2, ofs2, len2), t2, _)
  when ... -> ...
```

Finally, the general case is simply the application of the constructor `App`, as we discussed above:

```
| t1, t2 ->
    App (t1, t2, length t1 + length t2)
```

The entire code of `append` is given in program 27.

In the rest of the section, we will use the infix operator `++` to denote rope concatenation.

```
let (++) = append
```

The second idea that allows us to control the height of ropes consists in *rebalancing* the underlying trees. We could implement balancing incrementally as the rope is constructed, using for example the same technique as for AVL trees (see chapter 5). It would then be necessary to modify the last line of the function `append`, to invoke a balancing function instead of the constructor `App`. We may equally conceive of an *a posteriori* rebalancing implemented either selectively, when the height of the tree becomes too large, or explicitly, when the user requests it. Exercises 4.17–4.20 propose different ways of implementing an *a posteriori* rebalancing.

Subrope Extraction

The next operation consists in extracting a rope fragment, defined by an offset and a length. Extraction consists in retaining only those parts of the rope that are within this fragment, possibly reconstructing `Str` nodes that partially

Program 27 — Concatenation of Two Ropes

```
let small_length = 256

let append_string s1 ofs1 len1 s2 ofs2 len2 =
  Str (S.append (S.sub s1 ofs1 len1) (S.sub s2 ofs2 len2),
       0, len1 + len2)

let append t1 t2 = match t1, t2 with
  | Str (_,_,0), t | t, Str (_,_,0) ->
      t
  | Str (s1, ofs1, len1), Str (s2, ofs2, len2)
   when len1 <= small_length && len2 <= small_length ->
      append_string s1 ofs1 len1 s2 ofs2 len2
  | App (t1, Str (s1, ofs1, len1), _), Str (s2, ofs2, len2)
   when len1 <= small_length && len2 <= small_length ->
      App (t1, append_string s1 ofs1 len1 s2 ofs2 len2,
           length t1 + len1 + len2)
  | Str (s1, ofs1, len1), App (Str (s2, ofs2, len2), t2, _)
   when len1 <= small_length && len2 <= small_length ->
      App (append_string s1 ofs1 len1 s2 ofs2 len2, t2,
           len1 + len2 + length t2)
  | t1, t2 ->
      App (t1, t2, length t1 + length t2)

let (++) = append
```

overlap it. It is in fact preferable to specify the fragment to be extracted as extending from an offset `start` (inclusive) to the offset `stop` (exclusive). We assume that this is a valid and non-empty fragment, that is:

$$0 \leq \texttt{start} < \texttt{stop} \leq \texttt{length t}.$$

The extraction function, `mksub`, begins by considering the particular case in which the fragment is the whole rope.

```
let rec mksub start stop t =
  if start = 0 && stop = length t then
    t
```

This allows, to the extent possible, the sharing of subropes between the result of the extraction and the initial rope. Without this special case, these subropes would all be reconstructed.

In the general case, we examine the shape of the rope `t`. If `t` is a leaf, it suffices to modify the fragment specification:

```
  else match t with
    | Str (s, ofs, _) ->
        Str (s, ofs+start, stop-start)
```

Note that this is a constant-time operation, which does not call `S.sub`, and that it shares `s` between the initial rope and the result of the extraction. For a rope of form `App`, there are three cases to be considered. The first is when the fragment is completely contained in the left subrope. It then suffices to make a recursive call.

```
    | App (t1, t2, _) ->
        let n1 = length t1 in
        if stop <= n1 then mksub start stop t1
```

Symmetrically, the fragment may be completely contained in the right subrope.

```
        else if start >= n1 then mksub (start-n1) (stop-n1) t2
```

Finally, the fragment may overlap both `t1` and `t2`. It suffices then to concatenate the fragment of `t1` with that of `t2`.

```
        else mksub start n1 t1 ++ mksub 0 (stop-n1) t2
```

Note that neither of these fragments is empty, which respects the invariant of `mksub`. Note also that `t1` or `t2` may be completely contained in the result, in which case they will be shared by virtue of the first two lines of the function `mksub`.

Next, we define an extraction function `sub` specified by an offset `ofs` and a length `len` (similar to `String.sub` or `Array.sub`). It verifies the validity of the arguments, and then handles the special case of an empty fragment or calls the recursive function `mksub`.

```
let sub t ofs len =
  let stop = ofs + len in
  if ofs < 0 || len < 0 || stop > length t then invalid_arg "sub";
  if len = 0 then empty else mksub ofs stop t
```

The entire code is given in program 28.

Update Operations

We now consider several operations to insert or delete characters in a rope. Since ropes are persistent, these operations do not mutate them, but instead return new ropes.

Let us consider for example the operation `set`, which inserts the character `c` at offset `i` in a rope `t`. We begin by verifying that the offset `i` is valid, that is, $0 \leq$ `i` $<$ `length t`.

```
let set t i c =
  let n = length t in
  if i < 0 || i >= n then invalid_arg "set";
```

Next, it suffices to extract the subropes to the left and right of the offset `i`, and use concatenation to insert the character `c`.

```
  sub t 0 i ++ make 1 c ++ sub t (i + 1) (n - i - 1)
```

We note, however, that this solution is not optimal, since it performs two recursive traversals of the rope where one would have sufficed. Exercise 4.16 proposes an improvement in this regard.

Program 28 — Subrope Extraction

```
let rec mksub start stop t =
  if start = 0 && stop = length t then
    t
  else match t with
    | Str (s, ofs, _) ->
        Str (s, ofs+start, stop-start)
    | App (t1, t2, _) ->
        let n1 = length t1 in
        if stop <= n1 then mksub start stop t1
        else if start >= n1 then mksub (start-n1) (stop-n1) t2
        else mksub start n1 t1 ++ mksub 0 (stop-n1) t2

let sub t ofs len =
  let stop = ofs + len in
  if ofs < 0 || len < 0 || stop > length t then invalid_arg "sub";
  if len = 0 then empty else mksub ofs stop t
```

Program 29 — Update Operations on Ropes

```
let set t i c =
  let n = length t in
  if i < 0 || i >= n then invalid_arg "set";
  sub t 0 i ++ make 1 c ++ sub t (i + 1) (n - i - 1)

let insert t i r =
  let n = length t in
  if i < 0 || i > n then invalid_arg "insert";
  sub t 0 i ++ r ++ sub t i (n - i)

let insert_char t i c =
  insert t i (make 1 c)

let delete_char t i =
  let n = length t in
  if i < 0 || i >= n then invalid_arg "delete_char";
  sub t 0 i ++ sub t (i + 1) (n - i - 1)
```

We may similarly define operations `insert` (which inserts a rope within another at a given offset), `insert_char` (which inserts a character into a rope), and `delete_char` (which deletes the *i*-th character of a rope). The entire code of the four operations is given in program **??** (see previous page). This concludes our discussion of the functor defining ropes.

Application: Text Editor

Ropes are an ideal structure for text editors, in particular to support very long texts, which are rarely handled well by text editors, including the best-known ones.

Rather than using a single rope to represent the text to be edited in its entirety, we will use our functor for greater flexibility. Handling lines of text using a single rope is awkward: Line breaks have to be found, or their offsets recorded in a table that must be maintained in sync with the text. Instead, we can use ropes whose elements are characters to represent lines, and a rope whose elements are lines to represent the entire text.

We therefore begin by constructing a rope structure for lines. The argument of the functor `Make` is based on the OCaml module `String`:

```
module Str = struct
  include String
  let get = unsafe_get
  type char = Char.t
  let empty = ""
  let append = (^)
end
```

Note that we use `String.unsafe_get` rather than `String.get`, because every access to the characters of a string contained in a rope is guaranteed to be valid. We can thus avoid unnecessary checks. Ropes are obtained by applying the functor `Make` as follows:

```
module Line = Make(Str)
```

Next, we construct another rope structure for the whole text, whose "characters" are lines. For this, it suffices to apply the functor `Make` once again, this time using arrays for the leaves of our ropes.

```
module Text = Make(struct
                     type char = Line.t
                     type t = Line.t array
                     let empty = [||]
                     let length = Array.length
                     let make = Array.make
                     let append = Array.append
                     let get = Array.unsafe_get
                     let sub = Array.sub
                   end)
```

We see here the usefulness of defining ropes as a functor, which allows us to apply it twice on different arguments.

Let us assume, for simplicity's sake, that the text manipulated by the editor is stored in a reference containing a rope of type `Text.t`.

```
let text = ref ...
```

To insert a character `c` at offset `ofs` in line `l` of the text, it suffices to retrieve the line with `Text.get`, to modify it using `Line.insert`, and finally to update the text using `Text.set`:

```
let insert_char l ofs c =
  let line = Text.get !text l in
  let line' = Line.insert_char line ofs c in
  text := Text.set !text l line'
```

Similarly, to delete a character:

```
let delete_char l ofs =
  let line = Text.get !text l in
  let line' = Line.delete_char line ofs in
  text := Text.set !text l line'
```

Finally, to insert a new line at offset `ofs` in line `l` of the text, it suffices to cut the line `l` at offset `ofs` with two calls to `Line.sub`, then insert the prefix in line `l` using `Text.insert`, and to place the suffix in line `l+1` using `Text.set`:

```
let insert_newline l ofs =
  let line = Text.get !text l in
  let prefix = Line.sub line 0 ofs in
  let suffix = Line.sub line ofs (Line.length line - ofs) in
  let r = Text.insert_char !text l prefix in
  text := Text.set r (l + 1) suffix
```

Note that it is not necessary to "shift" all the lines: we use insertion in the rope of lines, just as we used insertion in the ropes of characters in `insert_char`.

Remark

Just as concatenation operations are suspended in ropes, extraction operations too may be suspended. The resulting OCaml type would be as follows:

```
type t =
  | Str of S.t
  | Sub of t * int * int
  | App of t * t * int
```

Everything then depends on the strategy used to choose between performing the concatenation and subrope operations immediately or suspending them.

The extraction operation may, in fact, be suspended "for free," by applying the functor `Make` repeatedly to its own result, as follows:

```
module R1 = Make(Str)
module R2 = Make(R1)
...
```

In the ropes of the module `R2`, the leaves are ropes of `R1`. Hence, the function `R1.sub` is only called when the leaf becomes sufficiently small. As long as this is not the case, the operation remains suspended. With a small abuse of notation, the types `R1.t` and `R2.t` could be said to correspond to the following definitions:

```
type R1.t =
  | R1.Str of string * int * int
  | R1.App of R1.t * R1.t * int
type R2.t =
  | R2.Str of R1.t * int * int
  | R2.App of R2.t * R2.t * int
```

It is clear that the node `R2.Str` suspends the extraction on a value of type `R1.t`. By iterating this process, we obtain a type equivalent to the preceding type `t`.

For Further Information

The rope data structure was introduced by Boehm, Atkinson, and Plass [5] in relation to the development of the Cedar language. Their article notably proposes an *a posteriori* rope rebalancing algorithm based on the numbers of the Fibonacci sequence (see exercise 4.18).

4.4 Persistent Arrays

This section presents a *persistent array* data structure. Throughout this section, we use the term "array" in its usual sense, to designate arrays that may be modified in-place. In all other cases, we use the term "persistent array."

The signature of persistent arrays is given in program 30. This is exactly the same signature as for arrays, except that the function `set` returns a new persistent array, without changing its argument.

A desirable property of persistent arrays is that they provide operations `set` and `get` with the same efficiency—$O(1)$—as arrays, at least as long as their persistent nature is not used. However, we accept that there is a price to be paid for access to previous versions of the array. This section presents a data structure that has this property.

The basic idea consists in using an array for the most recent version of the persistent array, together with extra information to allow us to go back to previous versions. To this end, we introduce the following two mutually recursive types:

Program 30 — Signature of Persistent Arrays

```
module type PersistentArray = sig
  type 'a t
  val init : int -> (int -> 'a) -> 'a t
  val length : 'a t -> int
  val get : 'a t -> int -> 'a
  val set : 'a t -> int -> 'a -> 'a t
  val iteri : (int -> 'a -> unit) -> 'a t -> unit
end
```

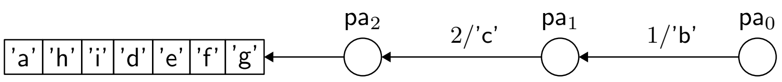
```
type 'a t = 'a data ref
and 'a data =
  | Arr of 'a array
  | Diff of int * 'a * 'a t
```

The type `'a t` is that of persistent arrays. It involves a reference to data of type `'a data`, which indicates its nature: either an immediate value `Arr` a, where a is an array, or an indirection `Diff` (i, v, p), representing a persistent array elementwise identical to the persistent array p, except at index i where the value v is to be found.

We illustrate the use of this data structure with an example. Consider the following series of declarations that define a persistent array `pa0`, then two others, `pa1` and `pa2`, obtained by two successive updates:

```
let pa0 = init 7 (fun i -> Char.chr (Char.code 'a' + i))
let pa1 = set pa0 1 'h'
let pa2 = set pa1 2 'i'
```

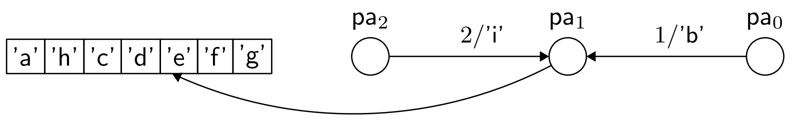
The situation after these three declarations is the following:



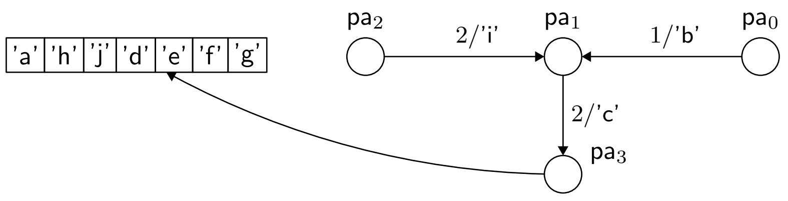
The last constructed persistent array, `pa2`, is a reference containing the value `Arr [|'a';'h';'i';'d';'e';'f';'g'|]`. The persistent array `pa1` is a reference containing `Diff (2, 'c', pa2)`, and the array `pa0`, a reference containing `Diff (1, 'b', pa1)`. Let us assume now that we construct a fourth array `pa3` as follows:

```
let pa3 = set pa1 2 'j'
```

We proceed in two steps: We begin by making sure that the persistent array on which we perform the operation, `pa1`, is of the form `Arr a`. For this, we invert the chain of `Diff` nodes that leads from `pa1` to the array `a`. We then have the following situation:



In the second step, we create a new reference for `pa3`, containing `Arr a`, where `a` has been modified to contain `'j'` in the second cell. We finish by modifying the reference `pa1` so that it now contains `Diff (2, 'c', pa3)`. The final situation is therefore as follows:



We will now describe the code of the different operations on persistent arrays. The creation of a new persistent array is straightforward:

```
let init n f = ref (Arr (Array.init n f))
```

For all other operations, we need a function `reroot` to invert a chain of `Diff` nodes so as to ensure that a persistent array `pa` is of the form `Arr a`. We write it as a recursive function `reroot : 'a t -> 'a array` that ensures this inversion of pointers and returns the array a. If `pa` is a reference to a value of form `Arr a`, it suffices to return `a`.

```
let rec reroot pa = match !pa with
  | Arr a ->
      a
```

If, however, `pa` is a reference to a value of form `Diff (i, v, pa’)`, we begin by calling `reroot` recursively on the persistent array `pa’`.

```
  | Diff (i, v, pa’) ->
      let a = reroot pa’ in
```

It is then guaranteed that `pa’` is a reference containing `Arr a`. It now suffices to exchange the roles of `pa’` and `pa`, that is, to store the value `Arr a` in `pa`, after having modified `a` in place, and to store a value `Diff` in `pa’`, indicating the previous value contained in `a`.

```
      let old = a.(i) in
      a.(i) <- v;
      pa := Arr a;
      pa’ := Diff (i, old, pa);
      a
```

The definitions of the `length` and `get` operations are straightforward if we use their equivalents in the `Array` module together with the `reroot` function.

```
let length pa = Array.length (reroot pa)
let get pa i = (reroot pa).(i)
```

The operation `iteri` is implemented by using the function `Array.iteri`, which applies a function to all the elements of an array and to their indices.

```
let iteri f pa = Array.iteri f (reroot pa)
```

The function `set` remains to be written. It constructs a new persistent array based on an existing persistent array `pa`, an index `i` and value `v`. We begin by calling `reroot` on `pa`.

```
let set pa i v =
  let a = reroot pa in
```

We then proceed as we did in case of the function `reroot`, except that we construct a *new* reference for the result:

```
let old = a.(i) in
a.(i) <- v;
let res = ref (Arr a) in
pa := Diff (i, old, res);
res
```

Note that at the end of this operation, the persistent array `pa` is one indirection away from the array `Arr a`. The entire code of persistent arrays is given in program 31.

For Further Information

The persistent array data structure is attributed to Henry Baker, who used it to represent environments in Lisp closures efficiently [11, 4].

4.5 Exercises

Resizeable Arrays

4.1 Modify the function `create` so that it takes an argument `increment` of type `int option` specifying a resizing strategy. If this argument is `None`, the function `resize` applies the preceding strategy (doubling the size of the array). If, on the other hand, `increment` is `Some n`, then `resize` increases the size of the array by `n` elements (or more if necessary).

4.2 We may sometimes wish to reduce the size of the array, for example when it is large compared with the number of elements actually in use or if it occupies a lot of memory. Modify the function `resize` so that it halves the size of the array when the number of elements is less than a quarter of the size of the array. Show that the amortized complexity remains $O(1)$.

4.3 Use a resizeable array to propose a stack structure with the following interface:

Program 31 — Persistent Arrays

```
type 'a t = 'a data ref
and 'a data =
  | Arr of 'a array
  | Diff of int * 'a * 'a t

let init n f = ref (Arr (Array.init n f))

let rec reroot pa = match !pa with
  | Arr a ->
      a
  | Diff (i, v, pa') ->
      let a = reroot pa' in
      let old = a.(i) in
      a.(i) <- v;
      pa := Arr a;
      pa' := Diff (i, old, pa);
      a

let length pa = Array.length (reroot pa)

let get pa i = (reroot pa).(i)

let iteri f pa = Array.iteri f (reroot pa)

let set pa i v =
  let a = reroot pa in
  let old = a.(i) in
  a.(i) <- v;
  let res = ref (Arr a) in
  pa := Diff (i, old, res);
  res
```

```
module type Stack = sig
  type 'a t
  val make : int -> 'a -> 'a t
  val length : 'a t -> int
  val push : 'a t -> 'a -> unit
  val pop : 'a t -> 'a
end
```

The call to `make n d` creates a stack containing `n` copies of the value `d`. The function `pop` raises an exception when the stack is empty.

4.4 In the context of the preceding exercise, show that n calls to the operations `push` and `pop` on a given queue have total cost $O(n)$.

Bit Vectors

4.5 Justify the fact that the expression `x land -x` extracts the low-order set bit of `x` for a non-zero integer `x`.

4.6 Write a function `ntz: int -> int` that calculates the number of low-order zero bits of its argument. Hint: You can tabulate this function over a byte and then use binary search.

4.7 Write a function `blit_bits` that copies bits i to $i+n-1$ of an integer x to the position j of a bit vector v, that is, so that we have for every index k in v:

$$(\texttt{blit\_bits}\ x\ i\ n\ v\ j)_k = \begin{cases} x_{i+k-j} & \text{if } j \le k < j+n, \\ v_k & \text{otherwise} \end{cases}$$

We will assume that the intervals $i..i+n-1$ and $j..j+n-1$ are valid for x and v, respectively.

4.8 Using the previous exercise, derive a function `blit` that copies the bits $i_1..i_1+n-1$ of a bit vector v_1 into a bit vector v_2 at position i_2.

4.9 Using the preceding function `blit`, write a function `sub` that extracts the bits $i..i+n-1$ of an array in the form of a new bit vector of size n.

4.10 Still using `blit`, write a function `append` that implements the concatenation of two bit vectors.

4.11 Write a function `fill` that assigns a constant value (0 or 1, given in the form of a boolean) to bits $i..i+n-1$ of an array. You can reuse the function `blit_bits` with x equal to 0 or -1.

4.12 Write a function `pop: int -> int` that counts the number of set bits of an integer by tabulating its values for all 8-bit integers. This function will be four times faster than the one given in section *4.2 Bit Vectors*.

4.13 The idea of a bit vector may be exploited to represent a subset of $\{0, 1, \ldots, 30\}$ (respectively, $\{0, 1, \ldots, 62\}$ in a 64-bit architecture) by a single integer. This is, moreover, a persistent data structure. Use this idea to write a module of the following form:

```
module Bitset : Set.S type elt = int
```

Ropes

4.14 Add a function to the functor `Make` that allows you to iterate over all leaves of a rope in infix order:

```
iter_leaves : (S.t -> int -> int -> unit) -> t -> unit
```

Each leaf is a string fragment corresponding to the three arguments of the constructor `Str`.

4.15 Add the parameter `small_length` as an argument of the functor `Make`.

4.16 Rewrite the functions `set`, `insert`, and `delete_char` so that they perform only one recursive traversal of the rope.

4.17 In this and the following two exercises, we propose methods to rebalance ropes *a posteriori*, that is, functions of the type:

```
val balance : t -> t
```

A simple method consists in constructing the list of all the leaves of the rope in infix order and then constructing a complete binary tree based on this list. Write a function `balance` following this idea.

4.18 The balancing algorithm proposed in the above exercise minimizes the height of the rope (as a tree) but does not take into consideration the length of the different leaves. If a leaf contains a large number of characters, it would be worthwhile to bring it closer to the root.

The article introducing ropes [5] proposes the following balancing method: Use an array `a` of ropes, verifying the following invariant: if the rope `a.`(i) is non-empty, its length belongs to the interval $[F_i, F_{i+1}[$, where F_i is the i-th term of the Fibonacci series, defined by $F_0 = F_1 = 1$ and $F_{n+2} = F_{n+1} + F_n$ for all $n \geq 0$.

Initially, all the ropes in `a` are empty. We then successively insert in `a` every leaf of the rope to be balanced, in infix order, in the following manner, starting from $i = 2$:

1. Let r be the rope obtained by concatenating the rope to be inserted with `a.`(i).
2. If the length of r is in $[F_i, F_{i+1}[$, assign r to `a.`(i) and then stop.
3. Otherwise, assign the empty rope to `a.`(i), increase i, and go back to step 1 with r as the rope to be inserted.

Once all leaves are inserted, the result is the concatenation of all the ropes in `a`. Write a function `balance` that implements this algorithm.

4.19 Let r be the rope obtained by the algorithm described in the preceding exercise, n its length, and h its height, where the height of a rope is defined by $h(\texttt{Str \_}) = 0$ and $h(\texttt{App}(r_1, r_2, \_)) = 1 + \max(h(r_1), h(r_2))$. Show that $n \geq F_{h+1}$. (Hint: Show that $h(\texttt{a.}(i)) \leq i - 2$ for every i.) From this, deduce that the average distance of a character to the root of the rope r is less than or equal to $\log_\phi(n) + K$ for a certain constant K, where ϕ is the golden ratio $(1 + \sqrt{5})/2$.

4.20 This exercise presents an optimal strategy to rebalance a rope c_0, known as the Garsia-Wachs algorithm. The algorithm acts on a list of ropes q equal to $\langle q_0, q_1, \ldots, q_m \rangle$, as follows:

1. Initially, the list q is the list of leaves of c_0, in infix order.

2. As long as the list q contains at least two elements,
   (a) determine the smallest index i, if it exists, such that `length` $q_i \leq$ `length` q_{i+1}; otherwise let $i = m$;
   (b) remove q_{i-1} and q_i from the list q and form their concatenation c;
   (c) determine the largest index j, if it exists, such that $j < i$ and `length` $q_{j-1} \geq$ `length` c; otherwise let $j = 0$;
   (d) insert c in the list q just after q_{j-1}.
3. Let c_1 be the unique element left in q. This rope is optimal, but its leaves are not in the same order as in c_0. The result is the rope c_2 that has the same leaves as c_0 and c_1, in the same order as in c_0, and at the same depth as in c_1.

Write a function `balance` that implements this algorithm. For more details on this algorithm, consult *The Art of Computer Programming* [15, vol. 3, sec. 6.2.2].

Persistent Arrays

4.21 You may have noticed that the code of `reroot` and of `set` unnecessarily recreates the value `Arr a`. Modify the two operations to avoid this.

4.22 Simplify the function `set` in the case of a call `set pa i v`, where the index `i` already maps to `v` in `pa`.

4.23 The recursive call in the function `reroot` is not a tail call. This can create problems if a persistent array is the result of a large number of modifications. Rewrite the function `reroot` so that it contains only tail-recursive calls.

5 Sets and Dictionaries

Sets and dictionaries are the most frequently used data structures. This chapter presents several ways of implementing sets. In each case, we discuss how they may be adapted for dictionaries. Some data structures may be used with elements of any type while others are more specialized, applicable to elements of a particular shape (for example, lists) or type (for example, integers).

Some of these data structures are presented in a persistent version and others in an imperative version. This is an arbitrary choice. Other choices are often possible, such as persistent hash tables or imperative AVL trees. Some of these variations are proposed in the exercises.

The choice of a data structure does not depend solely on its persistent or imperative character. Other criteria determine this choice, such as the available operations on the elements (for example, the existence of a total order), the operations provided by the data structure (for example, a union operation), or their respective cost (for example, the possibility of constructing a set in linear time).

5.1 Binary Search Trees

In this section, we implement persistent sets. A minimal signature for such sets, `PersistentSet`, is given in program 32. The type of the elements is `elt` and that of the sets is `t`. The empty set is represented by the value `empty`. The operations `add` and `mem` respectively add an element and test the membership of an element in a set. The function `min_elt` returns the smallest element of a set, if such an element exists, and raises the exception `Not_found` otherwise. The function `remove` removes an element from a set, if such an element is present, and returns the same set otherwise. Finally, the operation `cardinal` returns the number of elements of a set. Of course, a more realistic signature would contain additional operations, such as union, intersection, and difference.

The data structure chosen here to implement this signature is that of a *binary search tree*, that is, a binary tree such that every element contained in a node is larger than every element in its left subtree and smaller than every element in its right subtree. This organization allows searching for an element in time proportionate to the height of the tree. This tree structure assumes, however, that the elements can be compared to one another. For this, we require a comparison function `compare : elt -> elt -> int` such as:

$$\texttt{compare x y} \text{ est } \begin{cases} < 0 & \text{if } \texttt{x} \text{ is strictly smaller than } \texttt{y}, \\ = 0 & \text{if } \texttt{x} \text{ is equal to } \texttt{y}, \\ > 0 & \text{if } \texttt{x} \text{ is strictly larger than } \texttt{y}. \end{cases}$$

We gather the type `elt` and the declaration of the function `compare` in a signature `Ordered` shown in program 33. The code is then written as a functor parametrized by a module of signature `Ordered`:

```
module Make(X : Ordered) : PersistentSet with type elt = X.t =
struct
```

The signature of the module returned by this functor is `PersistentSet`, where we specify that the type `elt` of the elements is the type `X.t`.

Accordingly, in the body of the functor, we begin by introducing the type `elt` as a synonym of the type `X.t`.

Program 32 — Signature of Persistent Sets

```
module type PersistentSet = sig
  type elt
  type t
  val empty : t
  val add : elt -> t -> t
  val mem : elt -> t -> bool
  val min_elt : t -> elt
  val remove : elt -> t -> t
  val cardinal : t -> int
end
```

Program 33 — Signature of Ordered Types

```
module type Ordered = sig
  type t
  val compare: t -> t -> int
end
```

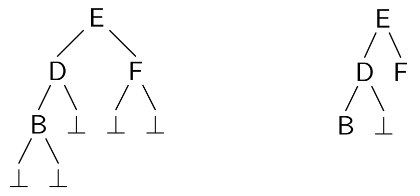


Figure 5.1: Two representations of the same binary search tree.

```
type elt = X.t
```

We then define the type `t` of sets as that of binary trees whose internal nodes contain a value of type `elt`:

```
type t = Empty | Node of t * elt * t
```

Consider the following OCaml value:

```
Node (Node (Node (Empty, 'B', Empty), 'D', Empty),
      'E',
      Node (Empty, 'F', Empty))
```

It corresponds to the left tree in figure 5.1, where only the value of type `elt` in each node is shown, and where `Empty` is represented by ⊥.

In what follows, we will not draw the children of a node when they are both empty, as depicted in the right tree of the figure.

The value `empty`, denoting the empty set, is simply a synonym for the constructor `Empty`, that is, the empty tree.

```
let empty = Empty
```

Smallest Element

The structure of a binary search tree makes it easy to retrieve its smallest element. It suffices to descend along the left branch as long as possible. The function `min_elt` implements this traversal.

```
let rec min_elt = function
```

```
    | Empty -> raise Not_found
    | Node (Empty, v, _) -> v
    | Node (l, _, _) -> min_elt l
```

Rather than the predefined exception `Not_found`, we could have declared and used an exception specific to binary search trees.

Searching for an Element

Searching for an element `x` in a binary search tree proceeds recursively as follows: If the tree is empty, we return `false`.

```
  let rec mem x = function
    | Empty ->
        false
```

Otherwise, we compare `x` with the element `v` situated at the root of the tree. If they are equal, the search finishes successfully. If `x` $<$ `v`, we continue recursively down the left subtree and, otherwise, down the right subtree.

```
    | Node (l, v, r) ->
        let c = X.compare x v in
        c = 0 || if c < 0 then mem x l else mem x r
```

Inserting an Element

Inserting an element `x` in a binary search tree `t` consists in finding the location of `x` in `t`, following the same principle as when searching for an element. If `t` is empty, it suffices to construct a tree that contains only `x`.

```
  let rec add x t =
    match t with
    | Empty ->
        Node (Empty, x, Empty)
```

Otherwise, we compare the element `x` with the root `v` of `t`. If they are equal, we return the tree `t` unchanged to avoid introducing a duplicate.

```
  | Node (l, v, r) ->
      let c = X.compare x v in
      if c = 0 then t
```

Otherwise, we recursively pursue the insertion to the left or to the right, depending on the result of the comparison.

```
      else if c < 0 then Node (add x l, v, r)
      else Node (l, v, add x r)
```

Of course, this implementation of the insertion function may lead to highly unbalanced trees. For example, if we successively add `F`, `E`, `D`, and `B` to the empty tree, we get the "comb" shown in figure 5.2.

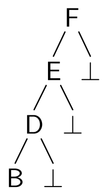


Figure 5.2: A binary search tree shaped like a comb.

Such shapes degrade the efficiency of subsequent search and insertion operations. The worst case corresponds to that of the comb where, as shown in the example, each internal node has only one non-empty subtree. Operations in such trees are then proportionate to the number of elements. In the following section, we will see that it is possible to limit the height of a binary search tree so that it remains logarithmic with respect to the number of elements. The complete code of the insertion function is given in program 34.

Removing an Element

To remove an element `x` from a binary search tree `t`, we proceed in the same manner as when searching or inserting an element, that is, by recursive descent towards the possible location of `x`. If `t` is empty, it suffices to return the empty tree.

Program 34 — Binary Search Trees (1/2)

```
type t = Empty | Node of t * elt * t

let rec min_elt = function
  | Empty -> raise Not_found
  | Node (Empty, v, __) -> v
  | Node (l, _, _) -> min_elt l

let rec mem x = function
  | Empty ->
      false
  | Node (l, v, r) ->
      let c = X.compare x v in
      c = 0 || if c < 0 then mem x l else mem x r

let rec add x t =
  match t with
  | Empty ->
      Node (Empty, x, Empty)
  | Node (l, v, r) ->
      let c = X.compare x v in
      if c = 0 then t
      else if c < 0 then Node (add x l, v, r)
      else Node (l, v, add x r)
```

```
let rec remove x = function
  | Empty ->
      Empty
```

Otherwise, we compare x to the root v of t. If they are equal, we face a problem: It is necessary to remove v, that is, to construct a tree from the left and the right subtrees of t. To implement this operation, we assume that we have written a function merge that performs this merging.

```
  | Node (l, v, r) ->
      let c = X.compare x v in
      if c = 0 then merge l r
```

If x is different from v, we descend recursively along the left or the right subtree to remove x. The code is similar to that of an insertion.

```
      else if c < 0 then Node (remove x l, v, r)
      else Node (l, v, remove x r)
```

The function merge, which merges two trees l and r, remains to be written. The difficulty lies in determining a root for the resulting tree. One way of proceeding consists in choosing the smallest element of r. Its value is obtained using the function min_elt written earlier. We must then remove this element from r, but this is precisely the operation that we are trying to implement.

Fortunately, the removal of the smallest element of a binary search tree is much simpler to implement than the removal of an arbitrary element. Let us write a function remove_min_elt for this. If the tree is empty, there is nothing more to be done.

```
let rec remove_min_elt = function
  | Empty -> Empty
```

If not, and if there is no left subtree, then the root is the smallest element, and we return the right subtree.

```
  | Node (Empty, _, r) -> r
```

Otherwise, we recursively remove the smallest element from the left subtree.

```
  | Node (l, v, r) -> Node (remove_min_elt l, v, r)
```

Program 35 — Binary Search Trees (2/2)

```
let rec remove_min_elt = function
  | Empty -> Empty
  | Node (Empty, _, r) -> r
  | Node (l, v, r) -> Node (remove_min_elt l, v, r)

let merge t1 t2 = match t1, t2 with
  | Empty, t | t, Empty -> t
  | _ -> Node (t1, min_elt t2, remove_min_elt t2)

let rec remove x = function
  | Empty ->
      Empty
  | Node (l, v, r) ->
      let c = X.compare x v in
      if c = 0 then merge l r
      else if c < 0 then Node (remove x l, v, r)
      else Node (l, v, remove x r)
```

We are now ready to write the function `merge`. If one of its two arguments is empty, we return the other. Otherwise, we use `min_elt` and `remove_min_elt`, and apply the idea mentioned previously.

```
let merge t1 t2 = match t1, t2 with
  | Empty, t | t, Empty -> t
  | _ -> Node (t1, min_elt t2, remove_min_elt t2)
```

The complete code for removing an element is given in program 35. Exercise 5.7 proposes a slight improvement, which consists in avoiding the reconstruction of the tree when the element to be removed is not actually present. This optimization is also applicable in the function `add`.

Dictionaries

We do not show here how to adapt this data structure to implement dictionaries since this will be done in the following section for balanced binary trees.

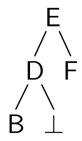
5.2 AVL Trees

As we saw in the previous section, the height of a binary search tree can be as high as the number of elements it contains, which is the case of the comb. The efficiency of each operation is directly affected by this. The problem can be resolved by *balancing* the binary search tree, that is, by trying to make its height as small as possible. In practice, we look for a solution that is not too costly at construction time, but which guarantees that the two subtrees of each node contain the same number of elements up to a small multiplicative factor. The solution discussed here is that of AVL trees, where the difference in heights of the left and right subtrees of each node is never allowed to exceed 1.

We begin by modifying the type of binary search trees to store the height of the tree as the fourth argument of the constructor `Node`.

```
type t = Empty | Node of t * elt * t * int
```

Consider the following tree BDEF:



It is represented by the following OCaml value:

```
Node (Node (Node (Empty, 'B', Empty, 1), 'D', Empty, 2),
      'E',
      Node (Empty, 'F', Empty, 1),
      3)
```

To manipulate the height of an AVL tree, it is convenient to introduce a function to retrieve it:

```
let height = function
  | Empty -> 0
  | Node (_, _, _, h) -> h
```

It is similarly useful to introduce a function that creates a new node while simultaneously computing its height:

```
let node l v r =
  Node (l, v, r, 1 + max (height l) (height r))
```

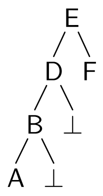
Such a function is what we call a *smart constructor*. It will be used in what follows instead of the constructor `Node`. Of course, the height computed by `node` is correct only if the heights stored in `l` and `r` are also correct. This will be guaranteed by making the type of AVL trees abstract.

Smallest Element and Membership Testing

As an AVL tree is a binary search tree, the functions `min_elt` and `mem` remain identical to those of program 34 (see page 211). The only difference is that they discard the fourth argument of the `Node` constructor.

Balancing

We illustrate the balancing of an AVL tree with the help of an example. If we insert the element `A` into the tree BDEF, we obtain the following tree ABDEF:



This tree is not balanced since the difference in height between the left and right subtrees of the node `E` is two. Nevertheless, it is easy to restore the balance of the tree. This can be done by performing local transformations that

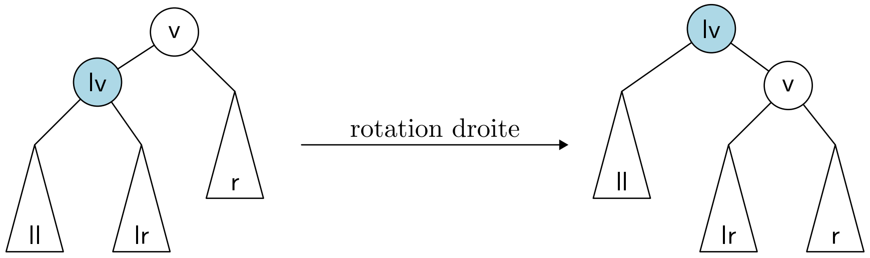
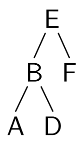


Figure 5.3: Right rotation in an AVL tree.

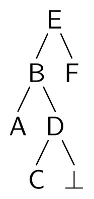
preserve the binary search tree property. An example of such an operation is *right rotation*, illustrated in figure 5.3.

This operation replaces the root `v` by the root `lv` of the left subtree and shifts the subtree `lr` containing the elements between `lv` and `v`. Note that this operation modifies only two nodes in the tree structure. The binary search tree property is preserved: the subtree `ll` remains to the left of `lv`; the subtree `r` remains to the right of `v`; and the subtree `lr` remains to the right of `lv` and to the left of `v`. A left rotation may be performed symmetrically.

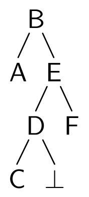
Thus, the tree ABDEF may be rebalanced by performing a right rotation on the subtree with root `D`. We then obtain the following tree, which satisfies the AVL condition.

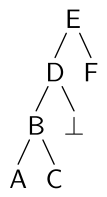


A simple left or right rotation does not necessarily suffice to rebalance the tree. For example, if we insert `C`, we obtain the following tree, which is not an AVL tree.

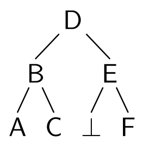


We can then try to perform a right rotation at the root `E` or a left rotation at the node `B`. However, neither operation yields an AVL tree.





Nevertheless, the tree on the right can be easily rebalanced by performing a further right rotation at the root `E`. We obtain the following AVL tree:



This double operation is called *left-right rotation*. Needless to say, there is also a symmetrical right-left rotation.

These four operations, that is, the two simple rotations and the two double rotations, suffice to rebalance any AVL tree following an insertion operation. We gather the rebalancing code in a *smart constructor* `balance` with the following type:

```
val balance : t -> elt -> t -> t
```

It behaves exactly like the function `node`, except that it also guarantees that the result will be balanced.

We begin by calculating the heights `hl` and `hr` of the left and right subtrees, and consider first the case in which the imbalance is due to the left subtree `l`:

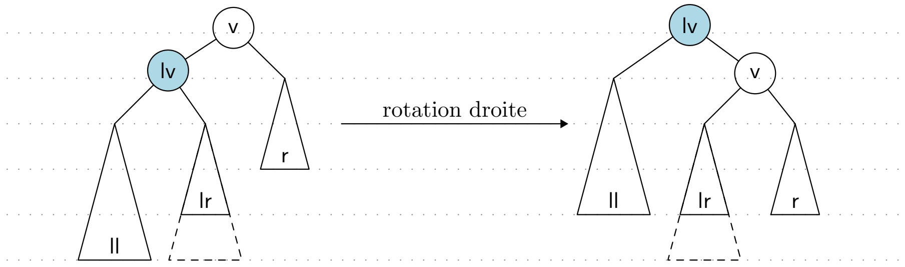


Figure 5.4: Simple rotation in an AVL tree.

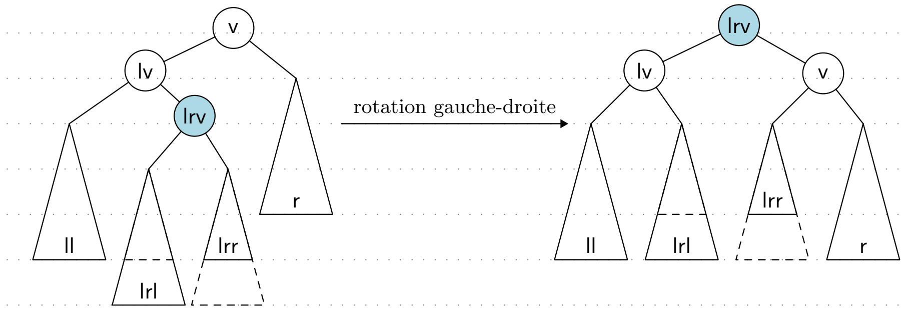


Figure 5.5: Double rotation in an AVL tree.

```
let balance l v r =
  let hl = height l in
  let hr = height r in
  if hl > hr + 1 then begin
```

As illustrated in figure 5.4, a simple right rotation suffices if the left subtree `ll` of `l` is at least as tall as the right subtree `lr`:

```
      match l with
        | Node(ll, lv, lr, _) when height ll >= height lr ->
          node ll lv (node lr v r)
```

On the other hand, if `ll` is shorter than `lr`, we must perform a left-right double rotation, as illustrated in figure 5.5.

```
    | Node(ll, lv, Node(lrl, lrv, lrr, _),_)->
        node (node ll lv lrl) lrv (node lrr v r)
```

Here too, the AVL property is guaranteed. Note that the imbalance can be due to either `lrl` or `lrr`, and that in both cases a left-right double rotation suffices to restore the balance.

As `l` and `r` are both assumed to be AVL trees, there is no other possible cause for imbalance when `l` is taller than `r`. This completes the pattern matching:

```
    |  _ -> assert false
```

The case where the imbalance is due to `r` is handled symmetrically:

```
end else if hr > hl + 1 then begin
  match r with
    | Node (rl, rv, rr, _) when height rr >= height rl ->
        node (node l v rl) rv rr
    | Node (Node(rll, rlv, rlr, _), rv, rr, _) ->
        node (node l v rll) rlv (node rlr rv rr)
    |  _ ->
        assert false
```

Finally, if the heights of `l` and `r` differ by at most one, we construct the node without rebalancing:

```
end else
  node l v r
```

The complete code of the function `balance` is given in program 36.

Insertion and Removal

We can now adapt the code for inserting an element (function `add` in program 34), replacing each application of the constructor `Node` with an application of the function `balance`. Similarly, we can adapt the code for removing

Program 36 — Balancing an AVL Tree

```
let balance l v r =
  let hl = height l in
  let hr = height r in
  if hl > hr + 1 then begin
    match l with
      | Node (ll, lv, lr, _) when height ll >= height lr ->
          node ll lv (node lr v r)
      | Node (ll, lv, Node (lrl, lrv, lrr, _),_)->
          node (node ll lv lrl) lrv (node lrr v r)
      | _ ->
          assert false
  end else if hr > hl + 1 then begin
    match r with
      | Node (rl, rv, rr, _) when height rr >= height rl ->
          node (node l v rl) rv rr
      | Node (Node(rll, rlv, rlr, _), rv, rr, _) ->
          node (node l v rll) rlv (node rlr rv rr)
      | _ ->
          assert false
  end else
    node l v r
```

an element (functions `remove_min_elt`, `merge`, and `remove`), making the same modification.

The code thus obtained is given in program 37. We emphasize that the adaptions above only involve simple substitutions.

Comparison of Sets

The implementation of a total order on sets is important. It has the following type:

```
val compare : t -> t -> int
```

In particular, this allows us to easily construct sets of sets by instantiating the functor `Make` several times in succession, as follows:

```
module Int = struct
  type t = int
  let compare = Stdlib.compare
end
module IntSet = Make(Int)
module IntSetSet = Make(IntSet)
```

The comparison on sets cannot be done simply by using `Stdlib.compare`. Indeed, two binary search trees may contain the same elements without having the same structure. A simple solution would be to construct the list of elements of each tree in the infix order and then compare them. This is, however, very inefficient: On the one hand, constructing the list requires a lot of memory. On the other, this construction can be wasteful if the lists differ near the beginning. Chapter 9 proposes an efficient solution to this problem.

Complexity

To justify the claim that AVL trees are balanced, we now show that the height of such a tree is always logarithmic in the number of elements.

Consider an AVL tree of height h. Let us try to estimate the number of its elements, n.

Program 37 — Insertion and Removal in an AVL Tree

```
let rec add x = function
  | Empty ->
      Node (Empty, x, Empty, 1)
  | Node (l, v, r, _) as t ->
      let c = X.compare x v in
      if c = 0 then t
      else if c < 0 then balance (add x l) v r
      else balance l v (add x r)

let rec remove_min_elt = function
  | Empty -> Empty
  | Node (Empty, _, r, _) -> r
  | Node (l, v, r, _) -> balance (remove_min_elt l) v r

let merge t1 t2 = match t1, t2 with
  | Empty, t | t, Empty -> t
  | _ -> balance t1 (min_elt t2) (remove_min_elt t2)

let rec remove x = function
  | Empty ->
      Empty
  | Node (l, v, r, _) ->
      let c = X.compare x v in
      if c = 0 then merge l r
      else if c < 0 then balance (remove x l) v r
      else balance l v (remove x r)
```

If the tree is perfectly balanced, then $n = 2^h - 1$. However, in general, we have only $n \leq 2^h - 1$. The smallest possible value of n would be attained in a tree having one subtree of height $h-1$ and the other of height $h-2$. (The reason is that, otherwise, we would still be able to remove elements from one of the two subtrees while maintaining the AVL property.) Thus, if we let N_h be the smallest number of elements in an AVL tree of height h, we have $N_h = 1 + N_{h-1} + N_{h-2}$, which can be rewritten as $N_h + 1 = (N_{h-1} + 1) + (N_{h-2} + 1)$. Equivalently, letting $G_h = N_h + 1$, we have $G_h = G_{h-1} + G_{h-2}$. Here, we recognize the recurrence relation that defines the Fibonacci sequence. As we also have $N_0 = 0$ and $N_1 = 1$, that is, $G_0 = 1$ and $G_1 = 2$, we deduce that $G_h = F_{h+2}$, where (F_i) is the Fibonacci sequence.

A mathematical result tells us that $F_i > \phi^i/\sqrt{5} - 1$, where $\phi = \frac{1+\sqrt{5}}{2}$ is the golden ratio. Hence:

$$n \geq N_h = F_{h+2} - 1 > \phi^{h+2}/\sqrt{5} - 2,$$

that is,

$$\phi^{h+2}/\sqrt{5} < n + 2.$$

Taking the base 2 logarithm of this inequality, we deduce the desired bound on the height h as a function of the number of elements n:

$$\begin{array}{rcl} h & \leq & \dfrac{1}{\log_2 \phi} \log_2(n+2) + \dfrac{\log_2 \sqrt{5}}{\log_2 \phi} - 2 \\ & \approx & 1,44 \log_2(n+2) - 0,33 \end{array}$$

An AVL tree therefore has a height that is logarithmic in the number of elements. This guarantees, in particular, $O(\log n)$ complexity for the searching, insertion, and removal operations. This also guarantees that these operations will not trigger a stack overflow.

Dictionary

A minimal signature for a persistent dictionary is given in program 38. It is very similar to that of a set (program 32, at the start of the chapter). The type `key`

Program 38 — Minimal Signature for Persistent Dictionaries

```
module type PersistentMap = sig
  type key
  type 'a t
  val empty : 'a t
  val add : key -> 'a -> 'a t -> 'a t
  val mem : key -> 'a t -> bool
  val find : key -> 'a t -> 'a
  val remove : key -> 'a t -> 'a t
end
```

of keys replaces the type `elt` of elements, for greater clarity. The type `t` is now polymorphic, its argument of type `'a` being the type of the values associated with the keys. The function `add` takes an additional argument. Finally, a new function `find` returns the value associated with a key, and raises the exception `Not_found` otherwise.

The adaptation of the structure of AVL trees to dictionaries consists in adding the value associated with the key to each node, that is:

```
type 'a t = Empty | Node of 'a t * key * 'a * 'a t * int
```

Program 39 — Searching in an AVL Dictionary

```
  let rec find k = function
    | Empty ->
        raise Not_found
    | Node (l, k', v, r, _) ->
        let c = X.compare k k' in
        if c = 0 then v else if c < 0 then find k l else find k r
```

The function `mem` is unchanged. The function `find` follows exactly the same schema as `mem`, except that the exception `Not_found` is raised if the key is not present in the dictionary. Its code is given in program 39. All the other functions on AVL trees (`balance`, `add`, etc.) are unchanged, except that elements are replaced by key/value pairs.

Modules Set and Map

The OCaml standard library provides the modules `Set` and `Map`, implemented using AVL trees, that are almost identical to those presented in this section. The only difference is that the condition on the height of the subtrees is relaxed (2 for OCaml versus 1 here). This choice is a compromise between the construction cost and the efficiency of the operations. These libraries provide all set-theoretic operations, such as union, intersection, inclusion testing, etc.

For Further Information

AVL trees were introduced by Georgy Adelson-Velskyĭ and Evgenii Landis in 1962 [2]. The union and intersection operations on AVL trees are detailed in an article by Adams [1]. There are other techniques to balance binary search trees, notably red-black trees [7, 19] and 2-3-4 trees [21].

5.3 Hash Tables

An *imperative* data structure for sets can be implemented using an array of booleans of size n if the elements are integers between 0 and $n-1$. However, this idea is not applicable to other kinds of elements, such as integers in a different range, elements of a different type, or if the number of elements is unbounded. We could also use a list, but searching is then costly.

We present here the *hash table* data structure, which combines these two approaches. The idea consists in using a function f that maps each element to an integer between 0 and $n-1$. In general, it is difficult to find an f that is *injective*. We will thus have to handle *collisions*, that is, cases where several elements map to the same value under f. Rather than using an array of booleans, we will use an array where each cell contains a “bucket” of elements represented as a list. Thus, an element x will belong to the set if and only if it belongs to the bucket at index $f(x)$. If the function f distributes the elements uniformly across the different buckets, then the search operation will be efficient.

Consider, for example, a hash table representing the following set of strings:

$$\{\texttt{""}, \texttt{"we like"}, \texttt{"the codes"}, \texttt{"in"}, \texttt{"ocaml"}\}$$

Let us assume the table has $n = 7$ buckets. We also assume that there is a function h, called the *hash function*, which associates a positive integer (or zero) to each element. We can then define the function f as:

$$f(s) = h(s) \bmod n$$

The operation *modulo* guarantees that the value of f is indeed in the range $0..n-1$. In the case of strings, we can take h to be the length of the string. If we use lists to represent buckets, we then obtain the structure shown in figure 5.6. Thus, for example, bucket 2 contains the two strings `"the codes"` and `"in"`, respectively of length 9 and 2.

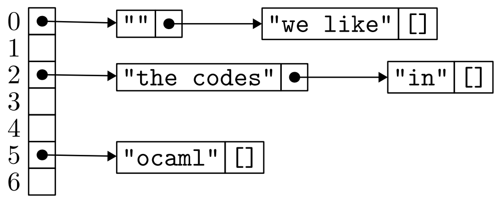


Figure 5.6: A hash table containing strings.

The choice of the number of buckets n and of the hash function h is important. For example, if we wish to represent the set of the 80 000 words of the French dictionary, we must choose a value of n that is sufficiently large, of the same order of magnitude as the number of elements, with a view to having small buckets. It is also necessary to choose a subtler function h than simply the length of the string, or else only a few buckets will be used.

Program 40 — Signature for Imperative Sets

```
module type ImperativeSet = sig
  type elt
  type t
  val create : unit -> t
  val add : elt -> t -> unit
  val mem : elt -> t -> bool
  val remove : elt -> t -> unit
  val cardinal : t -> int
end
```

We will now implement hash tables having the signature given in program 40. The type `elt` is that of the elements and the type `t` is that of hash tables. The function `create` returns a new table that is initially empty. The function `add` adds an element and the function `remove` removes an element. The function `mem` tests for the presence of an element and `cardinal` returns the number of elements in the set.

To add an element to a hash table, it is necessary to have a hash function `hash` of type `elt -> int`. To search for an element, it is also necessary to have an equality function `equal` of type `elt -> elt -> bool`. Rather than using the OCaml equality operator `=`, we prefer to let users choose their own comparison function. The two functions `hash` and `equal` must verify the following condition:

$$\forall x, y.\ \texttt{equal}\ x\ y \Rightarrow \texttt{hash}\ x = \texttt{hash}\ y$$

It is therefore natural to write hash tables as a functor parametrized by a module of type `HashType`, whose signature contains these two functions (see program 41 below).

The functor then takes the following form:

```
module Make(X: HashType): ImperativeSet with type elt = X.t = struct
  type elt = X.t
```

Program 41 — Signature Required for the Elements of a Hash Table

```
module type HashType = sig
  type t
  val hash : t -> int
  val equal : t -> t -> bool
end
```

The type `t` of hash tables could be defined as follows:

```
type t = (elt list) array
```

However, it is also preferable to store the number of elements in the set, so as to be able to implement the function `cardinal` efficiently. We therefore use a record type containing, on the one hand, the number of elements in a field `size` and, on the other, the array of buckets in a field `buckets`

```
type t = {
  mutable size : int;
  buckets : (elt list) array;
}
```

The function `cardinal` is then immediate.

```
let cardinal h =
  h.size
```

To create a new hash table, it is necessary to choose the size n of the array. Ideally, this size should be of the same order of magnitude as the number of elements that will be stored in the table. The user could provide this information, for example in the form of an additional argument to `create`, but this is not always possible. For the present, let us therefore consider a simplified situation in which the size is an arbitrary constant.

```
let array_length = 5003
```

We will see later how to eliminate the need for this arbitrary choice. The creation of an empty hash table consists in the creation of an array of size `array_length` containing only empty lists, and in initializing the field `size` to 0.

```
let create () =
  { size = 0;
    buckets = Array.make array_length []; }
```

Searching for an Element

To search for an element `x` within a hash table `h`, we begin by writing a function `bucket_of` that returns the bucket corresponding to `x` in `h`.

```
let bucket_of x h =
  (X.hash x land max_int) mod (Array.length h.buckets)
```

Since the result of `mod` has the same sign as its first argument, we make sure that the latter is not negative by clearing the sign bit[1].

To search for an element `x` in a table `h`, we begin by determining the index `i` of the bucket containing `x` with the function `bucket_of`.

```
let mem x h =
  let i = bucket_of x h in
```

Next, we search for `x` in the list `h.buckets.(i)`. For this we use the equality function on the type `elt`, namely, `X.equal`.

```
    List.exists (X.equal x) h.buckets.(i)
```

The complete code of `mem` is given below in program 42. Since the operation of searching within a bucket is reused several times subsequently, we turn it into a function, `mem_bucket`.

[1]Another solution, notably adopted in the hash tables of the OCaml standard library, consists is using an array whose size `n` is a power of 2. The modulo operation can then be implemented with `land (n-1)`, which has the added advantage of being much cheaper than a division.

Program 42 — Searching in a Hash Table

```
let bucket_of x h =
  (X.hash x land max_int) mod (Array.length h.buckets)

let mem_bucket x b =
  List.exists (X.equal x) b

let mem x h =
  let i = bucket_of x h in
  mem_bucket x h.buckets.(i)
```

Inserting an Element

To insert an element `x` in a table `h`, we begin by retrieving the list `b` of the bucket that may contain `x`.

```
let add x h =
  let i = bucket_of x h in
  let b = h.buckets.(i) in
```

If `x` is present in the bucket, then there is nothing more to be done. Otherwise, we increase the field `size` and add `x` at the head of the list `b`.

```
  if not (mem_bucket x b) then begin
    h.size <- h.size + 1;
    h.buckets.(i) <- x :: b
  end
```

The complete code of `add` is given below in program 43.

Note: We could have directly added `x` to the list `b` without checking for its presence beforehand. This would not modify the contents of the set (from the point of view of `mem`), and insertion would then happen in $O(1)$ time. On the other hand, it would have become impossible to keep track of the field `size`, and the operation `cardinal` would have become more costly. Furthermore, repeated insertions of the same element would have occupied more and more space in the hash table, thereby reducing the efficiency of search operations. Alternatively, to prevent duplicates, buckets may be represented using a more efficient data structure than lists. We could, for example, use AVL trees, presented in the previous section. This would nevertheless require an additional comparison function on the type `elt`.

Program 43 — Inserting an Element into a Hash Table

```
let add x h =
  let i = bucket_of x h in
  let b = h.buckets.(i) in
  if not (mem_bucket x b) then begin
    h.size <- h.size + 1;
    h.buckets.(i) <- x :: b
  end
```

Removing an Element

Removing an element from a hash table proceeds as in the insertion operation. The code of `remove` is given in program 44.

Program 44 — Deletion from a Hash Table

```
let remove x h =
  let i = bucket_of x h in
  let b = h.buckets.(i) in
  if mem_bucket x b then begin
    h.size <- h.size - 1;
    h.buckets.(i) <- List.filter (fun y -> not (X.equal y x)) b
  end
```

When `x` is present in the bucket `b`, it is necessary to decrease the field `size` and delete the occurrence of `x` in `b`. For this last operation, we use the function `List.filter` so as to preserve only the elements of the bucket that are distinct from `x`.

Resizing the Array Dynamically

The efficiency of the code that we presented above is satisfactory when the order of magnitude of the number of elements of the set is known beforehand. However, our structure is too simplistic if the number of elements can increase arbitrarily. Efficiency decreases rapidly once the *load factor* of the hash table—that is, the ratio between the number of elements in the table and the size of the array—becomes large. To remedy this, it is necessary to resize the array dynamically, depending on the load factor. For example, we may double the size of the array once the load factor is greater than 1/2.

To do this, it is necessary to modify the definition of the type `t` slightly, so as to make the field `buckets` `mutable`.

```
type t = {
  mutable size : int;
  mutable buckets : elt list array;
}
```

The initial value of the constant `array_length` remains arbitrary but will no longer impact the efficiency of the operations. The user may, nevertheless, continue to indicate an order of magnitude so as to avoid too many resizings.

The main modification is to the function `add`, where the array must be resized when the load factor becomes too large.

```
let add x h =
  let n = Array.length h.buckets in
  ...
  if not (mem_bucket x b) then begin
    ...
    if h.size > n/2 then resize h
  end
```

The test `h.size > n/2` is done immediately after the insertion of `x`. If the condition holds, we call a function `resize` that will take care of resizing the array. This function proceeds as in section *4.1 Resizeable Arrays*. It begins by allocating a new array `a`, of size $m = 2 \times n$.

```
let resize h =
  let n = Array.length h.buckets in
  let m = 2 * n in
  let a = Array.make m [] in
```

The size of the array having changed, the bucket number of an element is no longer necessarily the same due to the change in the *modulus*. It is therefore necessary to reassign each of the elements of `h` to their respective new buckets. For this, we introduce a function `rehash` that adds the element `x` into its new bucket.

```
    let rehash x =
      let i = (X.hash x land max_int) mod m in
      a.(i) <- x :: a.(i)
    in
```

It then suffices to execute `rehash` on all the buckets of the previous array `h.buckets`:

```
    Array.iter (List.iter rehash) h.buckets;
```

Finally, we replace the array `h.buckets` by `a`.

```
    h.buckets <- a
```

The complete code of `resize` is given in program 45.

As the size of arrays is limited to `Sys.max_array_length`, it is a good idea to verify that the length of the array created by the function `resize` is not too large (see exercise 5.11).

Program 45 — Resizing a Hash Table

```
let resize h =
  let n = Array.length h.buckets in
  let m = 2 * n in
  let a = Array.make m [] in
  let rehash x =
    let i = (X.hash x land max_int) mod m in
    a.(i) <- x :: a.(i)
  in
  Array.iter (List.iter rehash) h.buckets;
  h.buckets <- a
```

Complexity

The complexity of the operations on a hash table obviously depends on the hash function. If, for example, this function always returns the same value, then the hash table becomes a simple list, and the operations `mem`, `add`, and `remove` all have cost $O(N)$ if the table contains N elements. If, by contrast, the hash function distributes the elements uniformly over the different buckets, and if the number of buckets is sufficiently large, then we can hope that the size of each bucket will be bounded by a small constant. In this case, the complexity of each operation will be $O(1)$.

The tuning of the hash function is done empirically, for example by measuring maximal and average bucket sizes. On values such as integers, strings, or even arrays of integers, we can use the polymorphic hash function `Hashtbl.hash`, provided by the OCaml library, which gives good results. In the section *11.4 Hash-consing*, we will see how to define an efficient hash function for more complex values, such as lists or trees.

To maintain a bounded load factor, we have shown above how to dynamically resize the hash table. Each operation that triggers a resizing obviously has cost $O(N)$, since each element has to be reintroduced individually into the new array. However, if we follow the above strategy, consisting in doubling the size of the array each time, then the cost is *amortized* over the entire sequence of operations, exactly as in section 4.1 *Resizeable Arrays*. The operations mem, add, and remove therefore have amortized cost $O(1)$. A hash table is consequently an extremely efficient data structure.

Dictionary

To adapt the hash table structure to represent dictionaries, we add the values associated with the keys into the buckets. The lists become association lists and the type t becomes:

```
type 'a t = {
  mutable size : int;
  mutable buckets : (key * 'a) list array;
}
```

To check for the presence of a key in a list of type (key * 'a) list, we modify the function mem_bucket slightly, as follows:

```
let mem_bucket x b =
  List.exists (fun (y,_) -> X.equal x y) b
```

All the other functions are subject to similar minor modifications. The only new function that needs to be added is find. It looks for the value associated with a key in the corresponding bucket, which boils down to searching in an association list. We cannot use the predefined function List.assoc, because here we must use the equality predicate X.equal. We therefore write a specific function, lookup, for this. The code of find is given in program 46.

Module Hashtbl

The standard library of OCaml provides dictionaries implemented as hash tables in the form of a functor Hashtbl.Make, analogous to the one we just wrote.

This data structure may be used to associate several values with a single key. More precisely, it allows us to call the function add several times with the same key, without losing the values with which the key was previously associated. For example, if we execute the additions add h x v1 and add h x v2 sequentially

Program 46 — Searching in an Associative Hash Table

```
let find x h =
  let rec lookup = function
    | [] -> raise Not_found
    | (k, v) :: _ when X.equal x k -> v
    | _ :: b -> lookup b
  in
  let i = bucket_of x h in
  lookup h.buckets.(i)
```

for a key `x` in a table `h`, then `find h x` will return `v2`. If we then perform `remove h x`, this has the effect of deleting the last binding of `x`, and `find h x` will then return `v1`. On the other hand, if we wish to replace the value `v1` by `v2`, we can use the operation `replace`.

The module `Hashtbl` also provides a polymorphic hash table data structure that relies on the polymorphic hash function `Hashtbl.hash` that has type `'a -> int` and that is compatible with the structural equality operation `=`. This way, we avoid having to use the functor `Hashtbl.Make` when the keys are, for example, values of type `int` or `string`.

5.4 Prefix Trees

We are concerned in this section with a data structure used to represent sets of *words*. By "word," we mean here any OCaml value that can be seen as a sequence of *letters*. The first data type that comes to mind for such values is obviously the type `string` of strings, where the letters are values of type `char`. However, other values may also be considered as words. For example, an integer written in base 2 can be understood as a word formed of the letters `0` and `1`. More generally, we will assume that words are of type `L.t list`, where `L` is a module of signature `Letter`, given in program 47, containing a type `t` that represents letters. Apart from this type `t`, this signature assumes the existence

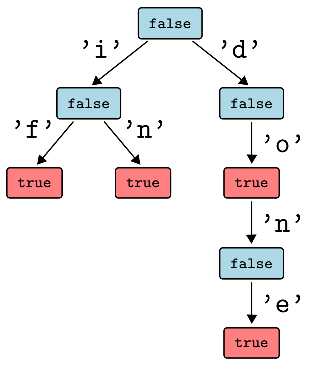


Figure 5.7: Prefix tree for the set {`if`, `in`, `do`, `done`}.

Program 47 — Minimal Signature of Letter Module

```
module type Letter = sig
  type t
  val compare: t -> t -> int
end
```

of a comparison function that we will need in what follows.

We use this decomposition into letters to represent sets of words with the help of a data structure called a *prefix tree*, also known as *trie*[2]. In these trees, each branch is labeled by a letter and each node contains a boolean indicating whether the sequence of letters leading from the root of the tree to this node is a word belonging to the set. For example, the prefix tree corresponding to the set of words {`if`, `in`, `do`, `done`} is represented in figure 5.7.

This data structure is useful to bound the time needed to search for a word in a set by the length of the word, independent of the number of words in the set.

In order to avoid depending on a particular implementation of the mod-

[2]This comes from the word re*trie*val.

Program 48 — Signature of Persistent Sets

```
module type PersistentSet = sig
  type elt
  type t
  val empty : t
  val add : elt -> t -> t
  val mem : elt -> t -> bool
  val remove : elt -> t -> t
  val inter : t -> t -> t
  val compare : t -> t -> int
end
```

ule L, we define the prefix tree data structure as a functor parametrized by L and returning a persistent set whose signature, PersistentSet, is given in program 48.

```
module Make(L: Letter): PersistentSet with type elt = L.t list =
struct
  type elt = L.t list
```

As mentioned earlier, the type elt of elements is defined as a list of letters, that is, a list of type L.t list.

The idea behind prefix trees is to represent each node as a dictionary that associates letters with subtrees, that is, with other nodes. We begin by introducing a dictionary M, whose keys are letters, by applying the functor Map.Make to the module L.

```
module M = Map.Make(L)
```

We can then define the type t of nodes as follows:

```
  type t = { word : bool; branches : t M.t }
```

The field word contains the boolean value indicating the presence of a word in the set. The field branches contains the children of the node.

Program 49 — Structure of a Prefix Tree

```
module Make(L : Letter) : PersistentSet with type elt = L.t list =
struct

  module M = Map.Make(L)

  type elt = L.t list
  type t = { word : bool ; branches : t M.t; }

  let empty = { word = false; branches = M.empty }

  let is_empty t = not t.word && M.is_empty t.branches

  let rec mem x t =
    match x with
      | [] ->
          t.word
      | i::l ->
          try mem l (M.find i t.branches)
          with Not_found -> false
```

The empty set is represented by a tree `empty` consisting of a single node, where the field `word` is `false`, and `branches` is the empty dictionary:

```
let empty = { word = false; branches = M.empty }
```

The code is given in program 49 along with a function `is_empty` that tests whether a set is empty.

Searching for an Element

Searching for an element `x` proceeds recursively as follows: If the word `x` is the empty list, the search ends, returning the boolean value contained in the root of the tree t.

```
let rec mem x t =
  match x with
  | [] -> t.word
```

Otherwise, we pursue the search recursively within the subtree associated with the branch labeled with the first letter i of the word x. If this branch does not exist, the search ends immediately with a failure.

```
  | i :: l ->
      try mem l (M.find i t.branches)
      with Not_found -> false
```

The code of the function mem is given in program 49.

Inserting an Element

Inserting a word x in a prefix tree t consists in descending along the branches labeled with the letters of x, as was done when searching for an element.

If x is the empty word, the insertion ends and returns a tree t with a field word set to true, indicating that x now belongs to the set.

```
let rec add x t =
  match x with
  | [] ->
      if t.word then t else { t with word = true }
```

We avoid reconstructing the node when t.word is already true.

If x is a non-empty list of the form i::l, we recursively insert l in the subtree b associated with the letter i in t. If b does not exist, we perform the insertion starting with an empty tree. The insertion ends with the creation of a new node in which we associate the letter i with the subtree obtained recursively.

```
  | i :: l ->
      let b = try M.find i t.branches with Not_found -> empty in
      { t with branches = M.add i (add l b) t.branches }
```

The code of the function `add` is given in program 50.

Removing an Element

Removing an element `x` in a tree `t` proceeds, once again, with the same recursive traversal. There is, however, a subtlety. Removing an element can leave the tree in a state in which one branch is completely empty, that is, where the fields `word` are all `false`. To avoid wasting space in this manner, we will enforce the property that all leaves of a prefix tree must represent a word belonging to the set. In other words, the boolean `word` of a leaf must always be `true`, with the exception, of course, of the empty set, represented by a single node with the boolean `word` set to `false`.

Let us now write the code of the function `remove`. If the argument `x` is the empty list, it is deleted from `t` by switching the field `word` to `false`.

```
let rec remove x t =
  match x with
    | [] ->
        { t with word = false }
```

Otherwise, `x` is of the form `i::l` and we begin by recursively deleting the rest of the word `l` from the subtree associated with `i`. If this subtree does not exist, the function ends, directly returning `t`.

```
    | i :: l ->
        try
          let s = remove l (M.find i t.branches) in
          ...
        with Not_found -> t
```

Next, if the tree `s` exists and is empty, we remove the branch associated with `i` in `t`. Otherwise, we create a new binding between `i` and `s`.

```
          let new_branches =
            if is_empty s then M.remove i t.branches
            else M.add i s t.branches
          in
```

Program 50 — Insertion and Removal in a Prefix Tree

```
let rec add x t =
  match x with
  | [] ->
      if t.word then t else { t with word = true }
  | i::l ->
      let b = try M.find i t.branches with Not_found -> empty in
      { t with branches = M.add i (add l b) t.branches }

let rec remove x t =
  match x with
  | [] ->
      { t with word = false }
  | i::l ->
      try
        let s = remove l (M.find i t.branches) in
        let new_branches =
          if is_empty s then M.remove i t.branches
          else M.add i s t.branches
        in
        { t with branches = new_branches }
      with Not_found -> t
```

This guarantees that no branch of t will point to an empty tree. Finally, the function ends by updating t with the new branches.

```
            { t with branches = new_branches }
```

The code of the function remove is given above in program 50.

Intersection of Prefix Trees

The intersection of two prefix trees is implemented with the help of two mutually recursive functions inter and inter_branches. The function inter computes the intersection of two trees t1 and t2 that are assumed to correspond to the same prefix p. The word p belongs to the intersection if the roots of t1 and t2 both contain the word p. The branches of the intersection are calculated by the function inter_branches.

```
let rec inter t1 t2 =
  { word = t1.word && t2.word ;
    branches = inter_branches t1.branches t2.branches }
```

The function inter_branches implements the intersection of two dictionaries m1 and m2. The idea is to enumerate the branches of m1 while checking, for each branch, if a corresponding branch exists in m2. For this we use the function M.fold, starting from an empty dictionary and considering each binding i $\mapsto$ ti of m1 in turn.

```
and inter_branches m1 m2 =
  M.fold
    (fun i ti m -> ... )
    m1 M.empty
```

For each binding, we recursively compute the intersection t of ti with the tree bound to i in m2, if it exists. We add the binding i $\mapsto$ t to m, taking care to verify that t is not empty. If m2 does not contain the branch i, the intersection is empty and m is left unchanged.

```
    (fun i ti m ->
       try
```

Program 51 — Intersection of Prefix Trees

```
let rec inter t1 t2 =
  { word = t1.word && t2.word;
    branches = inter_branches t1.branches t2.branches; }
and inter_branches m1 m2 =
  M.fold
    (fun i ti m ->
       try
         let t = inter ti (M.find i m2) in
         if is_empty t then m else M.add i t m
       with Not_found -> m)
    m1 M.empty
```

```
         let t = inter ti (M.find i m2) in
         if is_empty t then m else M.add i t m
       with Not_found -> m)
```

The complete code of the intersection function is given in program 51. The union operation is left as an exercise.

Comparison

Prefix trees can be compared easily. To compare two trees `t1` and `t2`, we begin by comparing the two booleans `t1.word` and `t2.word`. In case of equality, it is necessary to compare the two dictionaries `t1.branches` and `t2.branches`. For this, we use the comparison function provided by the module `M`. This takes as argument a function to compare the values in the two dictionaries, which corresponds precisely to the function `compare` that we seek to define. This function is therefore recursive.

```
let rec compare t1 t2 =
  let c = Stdlib.compare t1.word t2.word in
  if c<>0 then c else M.compare compare t1.branches t2.branches
```

Complexity

Consider a prefix tree s containing N words in all, and consider a word x of length M. The search for x in s necessitates at most M recursive calls to the function `mem`, each having the same cost as that of searching in the dictionary `branches` of the corresponding node. Since our dictionaries use the module `Map` of the standard library of OCaml, the cost of searching is logarithmic in the number of elements. The total cost is therefore proportionate to $\sum \log(B_i)$, where the B_i are the sizes of the dictionaries encountered during the search. We can bound each B_i by N, since the property that each leaf ends with `true` guarantees that a node with a dictionary of size B contains at least B different words. Hence the complexity is at worst $O(M \log N)$. However, it can be a lot less. If, for example, each B_i equals two, then the complexity will be $O(M)$, while N can be as large as 2^M. More generally, if the set of letters considered has a bounded size, as in case of the 26 letters of the alphabet for example, then searching has complexity $O(M)$. The insertion and removal operations have similar complexity.

For the intersection of two prefix trees containing respectively N_1 and N_2 elements of length less than M, we can similarly bound the complexity by $O(N_1 M \log N_2)$. If the number of letters is bounded, the complexity reduces to $O(N_1 M)$.

Dictionaries

A prefix tree encodes the presence of an element with a boolean contained in each node. Adapting this structure to represent dictionaries therefore consists in replacing the boolean by an optional value. We therefore define the following type:

```
  type 'a t = { value : 'a option ; branches : 'a t M.t }
```

The field `value` replaces the boolean field `word`. The necessary adaptations are immediate: the function `mem` checks if the field `value` is different from `None`; the function `add` replaces the current value (if present) by a new value; etc. The code of the function `find` is given in program 52.

Program 52 — Searching in a Prefix Tree

```
let rec find x t =
  match x, t.value with
    | [], None   -> raise Not_found
    | [], Some v -> v
    | i :: l, _  -> find l (M.find i t.branches)
```

5.5 Patricia Trees

We presented prefix trees in the preceding section. This data structure can be used to represent sets of integers by identifying an integer with the word formed by its binary digits. We call this a *Patricia tree*. Thus, the set $\{4, 5, 17\}$ can be seen as the following set of three binary words:

$$\{(100)_2, (101)_2, (10001)_2\}$$

We can construct a prefix tree for these three words. We have two possible Patricia trees, depending on whether we begin reading from the high- or the low-order bit. In the first case, we speak of *big-endian* Patricia trees and in the second, of *little-endian* Patricia trees[3]. In what follows, we consider only little-endian Patricia trees, which means that we examine the bits starting from the least significant one, that is, from right to left. Adopting the convention that a 0 bit leads to the left subtree while a 1 bit leads to the right subtree, the prefix tree obtained for the integers $\{4, 5, 17\}$ is represented in figure 5.8.

[3]The terms *little-endian* and *big-endian* were borrowed from Jonathan Swift's *Gulliver's Travels*.

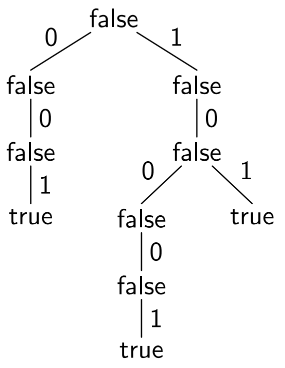


Figure 5.8: Prefix tree for the integers $\{4, 5, 17\}$.

We observe immediately that there is a lot of redundant information. For instance, the second bit of the three integers is always 0. Yet the prefix tree contains a branching node at this level. This is also the case for the fourth bit in the right subtree. We observe equally that it is useless to branch off when there is only one element left in the tree. To obtain a more compact representation, we will add a branching node to the tree only when necessary, that is, when there is a difference in the bits of the two subsets of elements. More precisely, a node will contain the bit to be tested as well as all the bits situated to its right. We thus obtain the tree of figure 5.9, where the bit to be tested is underlined.

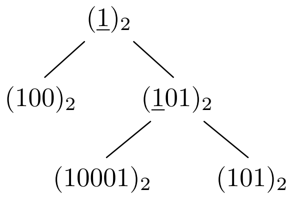


Figure 5.9: Patricia tree for the set $\{4, 5, 17\}$.

This tree signifies that we can begin by testing the first bit. If it is 0, we go to the left subtree, which contains a single element, $4 = (100)_2$. If it is 1, we go to the right subtree, which indicates that the third bit is to be tested, and that the two low-order bits are $(01)_2$ for this entire subtree. If the third bit is 0, we reach the subtree containing only $17 = (10001)_2$. If it is 1, we reach the subtree containing only $5 = (101)_2$. Note that the order of elements is not preserved in the tree structure. (This would, however, be the case in big-endian Patricia trees containing unsigned integers.)

It is important to note that a branching node such as $(\underline{1}01)_2$ contains two distinct pieces of information: the bit that must be tested (here the third) and the bits lying to its right (here $(01)_2$). Only the first piece of information is necessary to search for an element in the tree. Insertion, however, requires the second piece as well. We will use two integers to represent a branching node such as $(\underline{1}01)_2$: a power of two to represent the bit to be tested (here $(100)_2$) and an integer representing the prefix, that is, the bits of order less than this power of two (here $(01)_2$). One integer alone would suffice (here $(101)_2$), but since it is not easy to extract the high-order bit, it is simpler to maintain the two pieces of information separately. We adopt therefore the following type to represent Patricia trees:

```
type t =
  | Empty
  | Leaf of int
  | Node of int * int * t * t
```

The constructor `Empty` represents the empty tree; `Leaf x` is a leaf containing an element `x`; and, finally, `Node (p, b, l, r)` represents a branching point, where `p` is the prefix, `b` the bit to test (a power of 2), and `l` and `r` the two subtrees. In what follows we will guarantee the following well-formedness invariant: A tree of the form `Node` does not contain a subtree of the form `Empty`.

Searching for an Element

To search for an element `x` in a Patricia tree, it suffices to descend along the tree according to the branching bits, until a leaf is reached. We begin by writing a function `zero_bit` that tests whether the bit `b` of `x` is 0, using a *logical and* and assuming that the integer `b` is a power of 2.

```
let zero_bit x b =
    x land b == 0
```

We can now write the function `mem`. Its definition is clear if the tree is empty.

```
  let rec mem x = function
    | Empty -> false
```

If we reach a leaf `j`, we check whether `x` is equal to `j`.

```
    | Leaf j -> x = j
```

Finally, on a branching node, it suffices to determine if the bit corresponding to `x` is 0 or 1 by using the function `zero_bit`.

```
    | Node (_, b, l, r) -> mem x (if zero_bit x b then l else r)
```

The code of `mem` is given in program 53.

Program 53 — Searching in a Patricia Tree

```
let zero_bit x b =
  x land b == 0

let rec mem x = function
  | Empty -> false
  | Leaf j -> x = j
  | Node (_, b, l, r) -> mem x (if zero_bit x b then l else r)
```

Note that this code descends systematically until a leaf is reached, though it could sometimes stop earlier if the element `x` does not have the prefix expected by a branching node. We have chosen here an approach that performs fewer checks, which privileges situations in which a majority of searches have a positive outcome. Exercise 5.22 considers the other option.

Inserting an Element

Inserting an element into a Patricia tree consists in descending along the tree until the insertion position is found. We use an example to illustrate the two

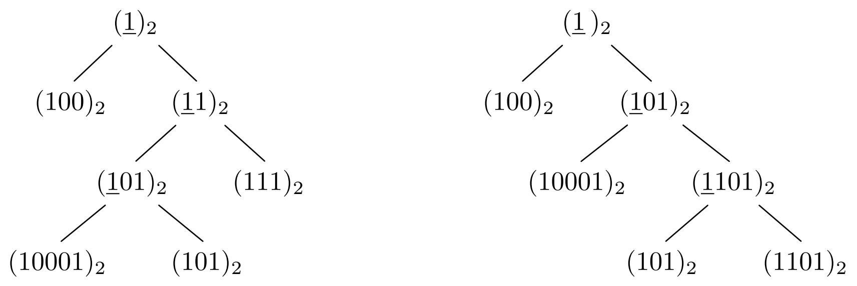


Figure 5.10: Insertion of $7 = (111)_2$ (on the left) or $13 = (1101)_2$ (on the right) in the Patricia tree of figure 5.9.

cases that may come up. Let us assume that we are inserting $7 = (111)_2$ into the tree of figure 5.9. The first branching node leads to the right subtree, because the low-order bit is 1. However, in the second branching node, namely, $(\underline{1}01)_2$, the prefix does not coincide: It is $(11)_2$ in the element to be inserted and $(10)_2$ in the subtree. It is therefore necessary to create a new branching node, namely, $(\underline{1}1)_2$, with the previous subtree to the left and the leaf 7 to the right. In the end, we obtain the left tree of figure 5.10. On the other hand, if we insert not 7 but $13 = (1101)_2$, then we reach the leaf containing 5, which must be separated into two distinct leaves, containing 5 and 13, which gives the tree on the right in figure 5.10.

We begin by implementing a function `branch` that takes two trees `t1` and `t2` as arguments, and creates the corresponding branching node. The prefixes of the two trees, `p1` and `p2`, are assumed to be different. It is therefore necessary to determine the lowest-order bit at which `p1` and `p2` differ. For this, we proceed in two steps: We begin by determining all the bits that differ, using the operation `p1 lxor p2`. Then, we extract the lowest-order set bit, which must exist. We have already seen how to do this using a *logical and* between `x` and `-x` in section *4.2 Bit Vectors*. Accordingly, we introduce a function `rightmost_1_bit` for this.

```
let rightmost_1_bit x =
  x land -x
```

Program 54 — Insertion in a Patricia Tree

```
let rightmost_1_bit x =
  x land -x

let branch p1 t1 p2 t2 =
  let b = rightmost_1_bit (p1 lxor p2) in
  let p = p1 land (b-1) in
  if zero_bit p1 b then
    Node (p, b, t1, t2)
  else
    Node (p, b, t2, t1)

let matches_prefix x p b =
  x land (b-1) == p

let rec add x = function
  | Empty ->
      Leaf x
  | Leaf j as t ->
      if j == x then t else branch x (Leaf x) j t
  | Node (p, b, l, r) as t ->
      if matches_prefix x p b then
        if zero_bit x b then
          Node (p, b, add x l, r)
        else
          Node (p, b, l, add x r)
      else
        branch x (Leaf x) p t
```

Let us now write the function `branch`. We begin by calculating the branching bit using `rightmost_1_bit`.

```
let branch p1 t1 p2 t2 =
  let b = rightmost_1_bit (p1 lxor p2) in
```

Then, we calculate the common prefix `p` of `p1` and `p2`. This is the set of bits of order less than `b`, that we extract using a *logical and* with the mask `b-1`.

```
  let p = p1 land (b-1) in
```

We use `p1` here to determine this prefix, but we could have also used `p2`. The only thing that remains now is to determine which of the two, `t1` or `t2`, must be placed on the left, depending on the value of the bit `b`.

```
  if zero_bit p1 b then
    Node (p, b, t1, t2)
  else
    Node (p, b, t2, t1)
```

Let us illustrate how the function `branch` works with `p1` $= (011101)_2$ and `p2` $= (110101)_2$. The result of `p1 lxor p2` is $(101000)_2$. The bit `b` extracted using `rightmost_1_bit` is therefore $(00\underline{1}000)_2$. The mask `b-1` is then $(000111)_2$, which gives the prefix `p` $= (000101)_2$.

We next define a function `matches_prefix` that determines if an integer `x` has the prefix defined by `p` and `b`.

```
let matches_prefix x p b =
  x land (b-1) == p
```

We are now able to write the function `add` that inserts an element `x` in a Patricia tree. The case of an empty tree is clear.

```
let rec add x = function
  | Empty ->
      Leaf x
```

In the case of a leaf `j`, we begin by checking if `j` is equal to `x`. If it is, there is nothing more to be done. Otherwise, a branching node must be created for the two leaves, using `branch`.

```
    | Leaf j as t ->
      if j == x then t else branch x (Leaf x) j t
```

Finally, in case of a `Node`, we determine whether `x` has the prefix of this tree.

```
    | Node (p, b, l, r) as t ->
      if matches_prefix x p b then
```

If this is the case, we pursue the insertion recursively in the left or the right subtree, depending on whether the bit of `x` defined by `b` equals 0 or 1.

```
        if zero_bit x b then
          Node (p, b, add x l, r)
        else
          Node (p, b, l, add x r)
```

If, on the other hand, `x` does not have the prefix `p`, then it is necessary to create a branching node with two subtrees, `t` and the leaf `x`. The function `branch` is made precisely for this.

```
      else
        branch x (Leaf x) p t
```

This concludes the function `add`. The complete code is given in program 54 (see page 251).

Removing an Element

Removing an element in a Patricia tree proceeds exactly as in case of insertion, the only difference being that it is necessary to maintain the well-formedness invariant. To this end, it suffices to introduce a *smart constructor* `node` that behaves like `Node` when its arguments are not `Empty` and performs a simplification otherwise.

```
  let node = function
    | (_, _, Empty, t)
    | (_, _, t, Empty) -> t
    | (p, b, l, r) -> Node (p, b, l, r)
```

Program 55 — Removing an Element in a Patricia Tree

```
let node = function
  | (_, _, Empty, t)
  | (_, _, t, Empty) -> t
  | (p, b, l, r) -> Node (p, b, l, r)

let rec remove x = function
  | Empty ->
      Empty
  | Leaf j as t ->
      if x == j then Empty else t
  | Node (p, m, t0, t1) as t ->
      if matches_prefix x p m then
        if zero_bit x m then
          node (p, m, remove x t0, t1)
        else
          node (p, m, t0, remove x t1)
      else
        t
```

By using the function `node` instead of the constructor `Node`, the definition of `remove` is clear. It is given in program 55.

Union

One of the advantages of Patricia trees compared to AVL trees is that the former lend themselves more easily to set-theoretic operations such as union. Indeed, two Patricia trees that have many elements in common will typically have the same, or similar, branching structures, thereby allowing an approach that uses recursive descent. This stands in contrast with AVL trees, where the root depends strongly on the manner in which the tree was constructed.

Let us implement the union of two Patricia trees, `t1` and `t2`. We begin by considering the trivial cases. The simplest is that in which one of the trees is empty.

```
let rec union t1 t2 = match t1, t2 with
  | Empty, t | t, Empty  ->
      t
```

Similarly, we can easily treat the case in which one of the two trees is a leaf, by calling the function `add`.

```
  | Leaf x, t | t, Leaf x ->
      add x t
```

Next we must consider the general case, where `t1` and `t2` are both branching nodes.

```
  | Node (p1, b1, l1, r1), Node (p2, b2, l2, r2) ->
```

There are three possible scenarios. The simplest case is that in which the prefixes of `t1` and `t2` coincide exactly. In this case, it suffices to recursively perform the union of the left and right subtrees respectively.

```
      if b1 == b2 && p1 = p2 then
        Node (p1, b1, union l1 l2, union r1 r2)
```

In the second case, one of the two prefixes is included in the other. Assume for example that the prefix of `t1` is included in the prefix of `t2`. This means that `t1` tests a bit that is of lower order than that of `t2`, that is, `b1 < b2`, and that the bits of `p2` below `b1` coincide with `p1`. In this case, all of `t2` must be recursively merged with the left or right subtree of `t1`, depending on the bit `b1` of the prefix `p2`.

```
      else if b1 < b2 && matches_prefix p2 p1 b1 then
        if zero_bit p2 b1 then
          Node (p1, b1, union l1 t2, r1)
        else
          Node (p1, b1, l1, union r1 t2)
```

We treat the symmetrical case, when the prefix of `t2` is included in that of `t1`, in a similar manner.

```
    else if b1 > b2 && matches_prefix p1 p2 b2 then
      ...
```

The last case is that in which the prefixes differ completely, neither being included in the other. This means that the elements of `t1` are disjoint from the elements of `t2`, and it suffices to create a new branching node with `t1` on one side and `t2` on the other, which is exactly what is done by the function `branch`.

```
    else
      branch p1 t1 p2 t2
```

The complete code of `union` is given below in program 56. The intersection, difference, and inclusion test operations are written in a similar manner (see exercise 5.23).

Comparison

Patricia trees can be compared easily. Indeed, two Patricia trees containing the same elements must have the same structure. Hence, the structural comparison functions of OCaml can be directly used on Patricia trees. In this way, we obtain an equality and a total order on the type `t` with the operator `=` and the function `Stdlib.compare`.

Complexity

The cost of searching, inserting, and removing elements in Patricia trees is proportionate to the number of bits of the integer considered. The cost is thus constant. Note, however, that this cost can reach 64 comparisons on a 64-bit machine, which, in the case of an AVL tree, would correspond to a structure containing more elements than can be represented in the memory of the computer. We therefore cannot deduce from this that a Patricia tree is better than an AVL tree in general. A Patricia tree can require more comparisons than an AVL tree. Indeed, a Patricia tree containing 30 elements can have a comb structure while the corresponding AVL tree would be balanced. Nonetheless, Patricia

Program 56 — Union of Two Patricia Trees

```
let rec union t1 t2 = match t1, t2 with
  | Empty, t | t, Empty  ->
      t
  | Leaf x, t | t, Leaf x ->
      add x t
  | Node (p1, b1, l1, r1), Node (p2, b2, l2, r2) ->
      if b1 == b2 && p1 = p2 then
        Node (p1, b1, union l1 l2, union r1 r2)
      else if b1 < b2 && matches_prefix p2 p1 b1 then
        if zero_bit p2 b1 then
          Node (p1, b1, union l1 t2, r1)
        else
          Node (p1, b1, l1, union r1 t2)
      else if b1 > b2 && matches_prefix p1 p2 b2 then
        if zero_bit p1 b2 then
          Node (p2, b2, union t1 l2, r2)
        else
          Node (p2, b2, l2, union t1 r2)
      else
        branch p1 t1 p2 t2
```

trees remain an interesting structure when we have to implement operations like union, intersection, or comparison.

Dictionaries

It is easy to adapt the type of Patricia trees to make dictionaries out of them. Indeed, it suffices to add a second argument to the constructor `Leaf`.

```
type key = int
type 'a t =
  | Empty
  | Leaf of key * 'a
  | Node of int * int * 'a t * 'a t
```

The adaptations needed for the different functions are clear. Exercise 5.27 proposes writing the function `find`.

For Further Information

Patricia trees were introduced in 1968 by Morrison, [18], who forged the term *Patricia* as an acronym for *Practical Algorithm To Retrieve Information Coded In Alphanumeric*. More recently, Patricia trees have been studied in the context of a functional language by Okasaki and Gill [20].
The technique of *hash-consing*, presented in chapter 11, allows the association of unique integers with values of certain types. We can then exploit Patricia trees to represent sets of values of this type, notably if we have to implement costly operations like the union or comparison of sets.

5.6 Exercises

Binary Search Trees

5.1 Write a function `height` that calculates the height of a tree.

5.2 Write a function `cardinal: t -> int` that returns the number of elements of a binary search tree.

5.3 Write a function `max_elt: t -> elt` analogous to `min_elt`.

5.4 Write a function `floor: elt -> t -> elt` that returns the largest element of a binary search tree less than or equal to a given element, if it exists, and raises the exception `Not_found` otherwise.

5.5 Write a function `iter: (elt -> unit) -> t -> unit` that traverses the elements of a binary search tree in increasing order.

5.6 Write a function `elements: t -> elt list` that returns the elements of a binary search tree in increasing order.

5.7 When calling `add` with an element already present or `remove` with an element that is absent, the efficiency of the two functions can be improved by directly returning the tree passed as argument. Rewrite the functions `add` and `remove` using this idea. You can raise an exception to signal that the tree is unchanged, taking care not to catch it at every recursive call, but only at the level of the initial call.

5.8 If you use the idea of the preceding exercise, it becomes easy to keep track of the cardinal of a binary search tree. You can, for example, use a record that stores the cardinal next to the tree. Rewrite the type `t`, and the functions `add` and `remove` in this way.

AVL Trees

All the previous exercises on binary search trees may be taken up again with AVL trees.

5.9 Write the function `add` for a dictionary implemented with an AVL tree. What should be done if a key is already associated with a value?

Hash Tables

5.10 The computation of `(X.hash x) mod (Array.length h.buckets)` would be incorrect if `X.hash x` returns a negative value, since the result would be negative. Explain why `(abs (X.hash x)) mod (Array.length h.buckets)` is not a solution.

5.11 Modify the function `resize` so that the size `m` of the new array does not exceed `Sys.max_array_length`.

5.12 In case of a dictionary, rather than using an association list of type `(key * 'a) list` to represent the buckets, you could use a more compact type, namely:

```
type 'a bucket = Nil | Cons of key * 'a * 'a bucket
```

Show that you would thus economize a third of the memory words. Rewrite the functions on hash tables using the type `'a bucket`.

5.13 Using persistent arrays, implement a data structure of persistent hash tables.

Prefix Trees

5.14 Write the function `cardinal` on prefix trees.

5.15 Write a function `min_elt` that returns the smallest element of a prefix tree with respect to the lexicographic order derived from `L.compare`. You can use the fact that the iterators of the module `Map` traverse the bindings in increasing order of the keys.

5.16 Improve the efficiency of the functions `add` and `remove` using the idea of exercice 5.7.

5.17 The function `mem` is not tail recursive since the recursive call is contained in a `try-with`. Remedy this problem by only catching the exception `Not_found` at the top of the function.

5.18 Rather than using the module `Map` of the OCaml standard library, the functor `Make` can take an additional argument `M` of signature

```
Map.S with type key = L.t
```

Rewrite the functor `Make` in this manner.

5.19 Write the function `union` on prefix trees. You can use two mutually recursive functions as we did in case of the function `inter`.

5.20 Write a variant of prefix trees where the dictionary `branches` is implemented with an array, for example, when words are formed from the characters `'a'..'z'`.

5.21 The `T9` input method of early mobile phones facilitates typing text on their numeric keyboards: Instead of pressing several times on a key to produce each letter, a single press suffices, and the phone proposes the words that correspond to the sequence of pressed keys, based on a dictionary that the phone has in memory.

For example, when you press the keys 2, 6, 6, 7, 8, 8, 3, 7 successively, you obtain the word *computer*, assuming the following mapping from keys to letters,

$$2 \mapsto \text{abc}, 3 \mapsto \text{def}, 4 \mapsto \text{ghi}, 5 \mapsto \text{jkl}, 6 \mapsto \text{mno}, 7 \mapsto \text{pqrs}, 8 \mapsto \text{tuv}, 9 \mapsto \text{wxyz}.$$

A sequence of keys may correspond to more than one word. For instance, the sequence 2, 2, 5, 3, 7 corresponds to both *baker* and *cakes*, and the sequence 3, 3, 2, 7 corresponds to both *dear* and *fear*.

You can use prefix trees to represent such dictionaries. We thus have a data structure that allows us to search efficiently for all the words in the telephone's dictionary that correspond to a given sequence of key presses.

To this end, it is necessary to modify the data structure presented in this chapter slightly. First of all, we must assume that the decomposition of words is not necessarily injective. In other words, two different words may now be represented by the same sequence of letters. We will assume the existence of a module implementing this decomposition, with the following signature:

```
module type Word = sig
  type t
  type letter
  val decomposition: t -> letter list
  val compare : t -> t -> int
end
```

Next, in order to store all the words whose decomposition corresponds to the same prefix, we must replace the boolean field `word`, which indicates a complete word, by a field containing a set of matching words. The beginning of the module `Make` will therefore have the following form:

```
module Make(L : Letter)(W : Word with type letter = L.t)
:  PersistentSet with type elt = W.t =
struct
  module S = Set.Make(W)
  module M = Map.Make(L)

  type elt = W.t
  type t = { words : S.t ; branches : t M.t }
```

Complete the code of the functor `Make`.

Patricia Trees

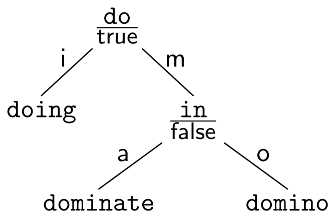
5.22 Modify the function `mem` on Patricia trees (program 53) so that it fails as soon as `x` does not possess the prefix corresponding to the node `Node`.

5.23 Write functions `inter`, `diff`, and `subset` on Patricia trees, implementing respectively the intersection, the difference, and the inclusion test, taking the function `union` as a model.

5.24 Write all the operations of *big-endian* Patricia trees, that is, where the bits are examined from left to right rather than from right to left. The main difficulty is writing the function `leftmost_1_bit` which is not as simple as `rightmost_1_bit`.

5.25 Write a library of Patricia trees, independently of the little-endian or big-endian nature, as a functor parameterized by those elements that differ between the two implementations.

5.26 Improve the representation of prefix trees by factoring out common prefixes as is done in Patricia trees. Thus, for the set of words {`do`, `doing`, `dominate`, `domino`} will have the following representation:



The root indicates that the first two letters must be `do` and that the corresponding word, `do`, is in the set. We move down the tree according to the value of the third letter. For `i`, we reach a leaf containing the word `doing`. For `m`, we reach a new branching node that indicates that the two following letters must be `in` and that the corresponding word, `domin`, is not in the set. According to the value of the sixth letter, we obtain either the leaf `dominate` or the leaf `domino`.

5.27 Write the function `find` for dictionaries represented using Patricia trees.

6 Queues

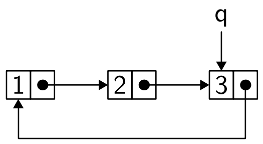
This chapter presents several data structures known as *queues*. A queue is a data structure whose elements are taken out in the order in which they were inserted, thus corresponding precisely to the usual notion of queue. We can also associate elements with priorities, in which case the elements are no longer taken out in the order in which they were inserted. This is called a *priority queue*.

6.1 Imperative Queues

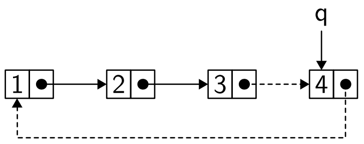
This section presents a data structure of imperative queues based on linked lists. The signature of this structure is given in program 57 (see following page). The idea is as follows: If a queue `q` contains the elements 1, 2, 3, inserted in this order, then it is represented by a circular linked list in which each element points to the following one in the queue, and the last element points to the first. To be able to insert and remove elements in constant time, it suffices to keep a pointer to the final element of the queue, here, 3. We therefore have the following situation:

Program 57 — Minimal signature for imperative queues.

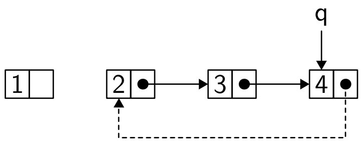
```
module type ImperativeQueue = sig
  type 'a t
  val create : unit -> 'a t
  val is_empty : 'a t -> bool
  val push : 'a -> 'a t -> unit
  val pop : 'a t -> 'a
end
```



Thus, to insert a new element, 4, it suffices to insert it after 3, that is, between 3 and 1, and to shift the pointer to 4.



We access the first element, here 1, by following the pointer contained in the final element. To remove the first element, it suffices to make the final element point to the second one, here 2, which is obtained by following two pointers.



We begin by introducing a type `'a cell` to represent the cells of the linked list, in the usual way.

```
type 'a cell = { elt : 'a; mutable next : 'a cell }
```

We then represent a queue by a reference to the last cell. A problem arises when it comes to representing the empty queue, for which we would not want to have any list cells at all. We choose therefore to represent a queue with a reference to a value of type `'a cell option`, where `None` represents the empty queue.

```
type 'a t = (('a cell) option) ref
```

The implementation of the functions `create` and `is_empty` is clear.

```
let create () =
  ref None
let is_empty q =
  !q = None
```

Inserting an Element

Inserting an element `x` into a queue `q` follows the scheme mentioned above. If `q` is empty, we create a circular list of one element and modify `q` so that it points to this element. We use here the possibility afforded by OCaml of constructing a cyclic value with `let rec`.

```
let push x q = match !q with
  | None ->
      let rec c = { elt = x; next = c } in
      q := Some c
```

If `q` is not empty, we create a new cell `c` containing `x`, that we insert between the final and first elements, and then modify `q` so that it points to this new cell.

```
  | Some last ->
      let c = { elt = x; next = last.next } in
      last.next <- c;
      q := Some c
```

Extracting an Element

To extract the first element of a queue `q`, we begin by testing whether `q` is empty. In this case, we raise an exception.

```
let pop q = match !q with
  | None ->
      invalid_arg "pop"
```

When q is not empty, it is convenient to treat the case in which q contains only one element separately. Here, the final element points to itself. It thus suffices to empty the queue, giving it the value None and returning the only element that it contained.

```
  | Some last when last.next == last ->
      q := None;
      last.elt
```

In the general case, when q contains at least two elements, we begin by determining the first element of the queue, first. We then remove it from the linked list by making the final element point to the second one. The value to be returned is contained in the cell first.

```
  | Some last ->
      let first = last.next in
      last.next <- first.next;
      first.elt
```

The complete code is given in program 58 (see following page).

Complexity

It is clear that the operations create, is_empty, push, and pop all execute in constant time. A queue containing N elements occupies three words per element in memory (being a record with two fields, elt and next), plus two words for the reference.

Program 58 — Imperative queues using linked lists.

```
type 'a cell = { elt : 'a; mutable next : 'a cell }

type 'a t = 'a cell option ref

let create () =
  ref None

let is_empty q =
  !q = None

let push x q = match !q with
  | None ->
      let rec c = { elt = x; next = c } in
      q := Some c
  | Some last ->
      let c = { elt = x; next = last.next } in
      last.next <- c;
      q := Some c

let pop q = match !q with
  | None ->
      invalid_arg "pop"
  | Some last when last.next == last ->
      q := None;
      last.elt
  | Some last ->
      let first = last.next in
      last.next <- first.next;
      first.elt
```

For Further Information

The module `Queue` of the OCaml standard library also uses a singly linked list, but in a slightly different way. First, it uses a type with two constructors, `Nil` and `Cons`, instead of using the combination of a record type and an option type. Second, it maintains two pointers, to the first and last elements, rather than using a circular list.

6.2 Persistent Queues

In this section, we present a data structure of *persistent* queues. The signature of such a structure is given in program 59. We see, in particular, that the operations `push` and `pop` return a modified version of the data structure.

A naive idea would consist in representing a queue by a list. This solution is unfortunately very inefficient, because either the insertion or the extraction operation would have to be performed at the end of the list. To implement these two operations more efficiently, a simple idea consists in using not one but two lists. The first list contains the elements at the front of the queue, in order. The second list contains the elements at the back of the queue, in reverse order. In this way, both insertion and extraction are done at the head of a list. For example, the queue containing the elements 1, 2, 3, 4, 5, inserted in this order, may be represented by the two lists `[1; 2]` and `[5; 4; 3]`. However, the queue could equally be represented by the two lists `[1]` and `[5;4;3;2]`, or even by `[1;2;3;4;5]` and `[]`, which are all equivalent representations of the same queue.

We therefore introduce the type of persistent queues simply as a synonym for a pair of lists.

```
type 'a t = 'a list * 'a list
```

The first list represents the front of the queue. The second represents the back of the queue. The definitions of the empty queue `empty` and the function `is_empty` are clear.

```
let empty = [], []
```

Program 59 — Minimal signature for persistent queues.

```
module type PersistentQueue = sig
  type 'a t
  val empty : 'a t
  val is_empty : 'a t -> bool
  val push : 'a -> 'a t -> 'a t
  val pop : 'a t -> 'a * 'a t
end
```

```
let is_empty = function
  | [], [] -> true
  | _ -> false
```

To insert an element `x`, it suffices to add it to the head of the second list.

```
let push x (o, i) =
  (o, x :: i)
```

Extraction is a more delicate operation. There are three possible cases. First, when the queue is empty, we raise an exception.

```
let pop = function
  | [], [] ->
      invalid_arg "pop"
```

Second, when the first list, which represents the front of the queue, contains at least one element, it suffices to extract this element.

```
  | x :: o, i ->
      x, (o, i)
```

Third, when all the elements are in the back of the queue, it suffices to reverse the list `i` and proceed as in the previous case, where the element to be extracted was at the head of the list. The tail of the list thus obtained becomes the new front of the queue, and the back of the new queue no longer contains any elements.

```
  | [], i ->
      match List.rev i with
      | x :: o -> x, (o, [])
```

Since the list `i` is not empty, the list `List.rev i` is also not empty. Rather than write a non-exhaustive pattern matching, which would trigger a compiler warning, we indicate explicitly that the case of an empty list cannot occur.

```
      | [] -> assert false
```

The complete code is given in program 60.

Complexity

It is clear that the operations `is_empty` and `push` execute in constant time. The operation `pop` can, however, have cost $O(N)$ for a queue containing N elements. This is the case when all the elements are at the back of the queue so that the list has to be inverted. Nevertheless, if we consider a sequence of M successive `push` and `pop` operations, starting from an empty queue, then the total cost cannot be greater than $O(M)$. Indeed, each element may be involved in, at most, one list reversal. The `pop` operation therefore has amortized cost $O(1)$. The space complexity is the same as in case of imperative queues, with three words per element of the queue.

6.3 Imperative Priority Queues

In this section and the next, we will consider queues in which the elements are associated with priorities. In such queues, called *priority queues*, the elements are removed in the order fixed by their priority rather than in the order in which they were entered. We present here an *imperative* data structure of priority queues, whose signature is given in program 61 (see the following page).

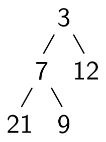
In this interface, the notion of minimality coincides with the notion of greatest priority. Unlike with ordinary queues, here we distinguish the access and

Program 60 — Persistent queues represented by pairs of lists.

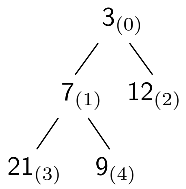
```
type 'a t = 'a list * 'a list

let empty =
  ([], [])

let is_empty = function
  | [], [] -> true
  | _ -> false

let push x (o, i) =
  (o, x :: i)

let pop = function
  | [], [] ->
      invalid_arg "pop"
  | x :: o, i ->
      x, (o, i)
  | [], i ->
      match List.rev i with
      | x :: o -> x, (o, [])
      | [] -> assert false
```

removal of the first element using two separate operations. This is for the sake of efficiency, as will be explained later. Accordingly, the function `get_min` returns the element of the queue with the highest priority, and the function `remove_min` removes it.

To implement an efficient priority queue, it is necessary to use a more complex data structure than in case of ordinary queues. One solution consists in organizing the elements as a *heap*. A heap is a binary tree in which the element stored at each node has a higher priority than the two elements situated immediately below it. The element with the highest priority is therefore found at the root. Thus, a heap containing the elements $\{3, 7, 9, 12, 21\}$, ordered by magnitude, may have the following shape:



We note that there are other heaps containing the same elements. However, for the sake of efficiency, it is preferable to choose a heap of minimal height. Heaps represented by complete binary trees (in which all levels are filled except, perhaps, the last one) have minimal height. It turns out that a complete binary tree can be easily represented using an array. The idea consists in numbering the nodes of the tree from top to bottom and left to right, beginning from 0. The result of this numbering of the heap above yields the following labeling:



This numbering permits the representation of the heap in an array of five elements as follows:

0	1	2	3	4
3	7	12	21	9

Program 61 — Minimal signature for imperative priority queues.

```
module type ImperativePriorityQueue = sig
  type t
  type elt
  val create : unit -> t
  val is_empty : t -> bool
  val add : elt -> t -> unit
  val get_min : t -> elt
  val remove_min : t -> unit
end
```

In general, the root of the tree occupies the cell at index 0, and the roots of the two subtrees of the node stored at cell i are stored in cells $2i + 1$ and $2i + 2$, respectively. Conversely, the parent of node i is stored at $\lfloor (i-1)/2 \rfloor$.

One problem remains: We do not know the size of the priority queue beforehand and therefore cannot fix in advance the maximum size of the array. An elegant solution consists in using resizeable arrays. Such arrays were presented in section *4.1 Resizeable Arrays*. It then suffices to write a functor parametrized by a structure of resizeable arrays. A signature for such arrays, `ResizeableArray`, is given in program 17, at the start of chapter 4. As far as the elements are concerned, we require a type equipped with a total order and a default value. The corresponding signature, `OrderedWithDummy`, is given in program 62.

The functor of priority queues then takes the following form:

```
module Make(X: OrderedWithDummy)(A: ResizeableArray)
  : ImperativePriorityQueue with type elt = X.t =
struct
```

Module `X` is then that of elements, and module `A` that of resizeable arrays.

The types `elt` and `t` are respectively synonyms for `X.t` and `elt A.t`.

```
  type elt = X.t
  type t = elt A.t
```

Program 62 — Ordered elements with a default value.

```
module type OrderedWithDummy = sig
  type t
  val compare: t -> t -> int
  val dummy : t
end
```

In fact, a heap is nothing but a resizeable array. The function `create` constructs an empty array with `A.make`, and the function `is_empty` tests whether the array is empty.

```
let create () = A.make 0 X.dummy
let is_empty h = A.length h = 0
```

The function `get_min` returns the root of the heap, if it exists, and raises an exception otherwise.

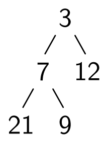
```
let get_min h =
  if A.length h = 0 then invalid_arg "get_min";
  A.get h 0
```

The code is given in program 63.

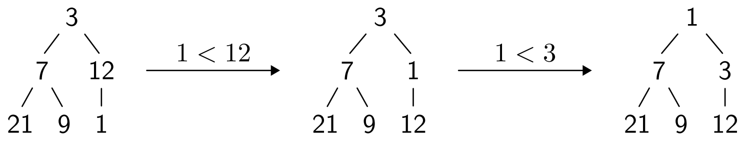
Inserting an Element

Inserting an element `x` in a heap `h` consists in adding `x` into the tree that represents `h` while preserving the heap structure. One approach consists in adding `x` at the bottom right and then moving it up the tree as long as is necessary. In our implementation, this boils down to extending the array by one cell, inserting the value `x` there, and then moving `x` up until it reaches the correct position. For this, we use the following algorithm: if `x` is smaller than its parent, we exchange the two values and repeat.

For example, let us consider again the following heap:



The insertion of the element 1 in this heap is performed in three steps:



We begin by writing a function `move_up` that inserts an element `x` in the heap `h`, starting at position `i`. This function assumes that the tree of root `i`, obtained by placing `x` at `i`, is a heap. The function `move_up` first considers the case in which `i` is 0, that is, when we have reached the root. It then suffices to insert `x` at position `i`.

```
let rec move_up h x i =
  if i = 0 then
    A.set h i x
```

If, however, we are at an internal node, we calculate the index `fi` of the parent of `i` and the value `y` stored in this node. If `y` is greater than `x`, we need to move `x` up while moving `y` down to position `i`. We then call `move_up` recursively beginning from `fi`.

```
  else
    let fi = (i - 1) / 2 in
    let y = A.get h fi in
    if X.compare y x > 0 then begin
      A.set h i y;
      move_up h x fi
    end
```

If, however, `y` is less than or equal to `x`, then `x` is at its final position, and it suffices to store it there.

```
      else A.set h i x
```

The function `add` proceeds in two steps. It increases the size of the array by one unit, adding a cell at the end of the array. Then it calls the function `move_up`, beginning from this cell.

```
let add x h =
  let n = A.length h in
  A.resize h (n + 1);
  move_up h x n
```

The code is given in program 63 (see next page).

Removing the Smallest Element

Removing the smallest element of a heap is more delicate. The reason for this is that we remove the root of the tree and must, therefore, find an element to replace it. The idea consists in choosing the element at the bottom right of the heap, that is, the element occupying the last cell of the array, placing it at the root, and then moving it down the heap until it reaches its place. This is roughly what we did when inserting an element, where we moved the new element up.

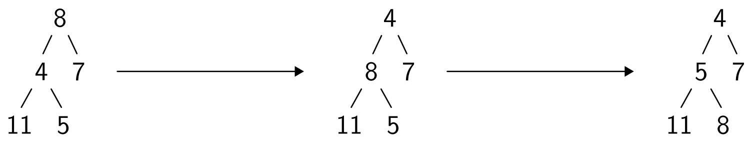
Let us assume, for example, that we wish to remove the smallest element of the following heap:

```
      1
     / \
    4   7
   / \  |
  11  5 8
```

We replace the root, 1, by the element at the bottom right, 8. Then we make 8 move down until it reaches its place. For this, we compare 8 with the roots a and b of the two subtrees. If a and b are both greater than 8, the descent is complete. Otherwise, we exchange 8 with the smaller of the two nodes a and b, and continue the descent. In the example, 8 is successively exchanged with 4 and 5:

Program 63 — An imperative heap data structure (1/2).

```
module type OrderedWithDummy = sig
  type t
  val compare: t -> t -> int
  val dummy: t
end

module Make(X: OrderedWithDummy)(A: ResizeableArray)
  : ImperativePriorityQueue with type elt = X.t =
struct
  type elt = X.t
  type t = elt A.t

  let create () =
    A.make 0 X.dummy

  let is_empty h =
    A.length h = 0

  let get_min h =
    if A.length h = 0 then invalid_arg "get_min";
    A.get h 0

  let rec move_up h x i =
    if i = 0 then A.set h i x else
      let fi = (i - 1) / 2 in
      let y = A.get h fi in
      if X.compare y x > 0 then begin
        A.set h i y;
        move_up h x fi
      end else
        A.set h i x

  let add x h =
    let n = A.length h in A.resize h (n + 1); move_up h x n
```



We begin by writing a function that compares two nodes, given by their indices `l` and `r`, and returns the smaller of the two.

```
let min h l r =
  if X.compare (A.get h r) (A.get h l) < 0 then r else l
```

Next, we write a function `smallest_node` that determines if a value `x`, situated at node `i`, needs to move down the heap or not. The value `x` is not that of the node `i` because we do not know yet if this will be its final position. However, the function `smallest_node` acts as if the node `i` had the value `x`. The function `smallest_node` accordingly takes as parameters the heap `h`, the value `x`, and the index `i`. We begin by calculating the left child `l` of node `i`.

```
let smallest_node h x i =
  let l = 2 * i + 1 in
```

If the node `i` has no children, we return `i`.

```
  let n = A.length h in
  if l >= n then
    i
```

If, however, the node `l` is part of the heap, we determine the index `j` of the smallest child of `i`. For this, it is convenient to determine if the right child `r = l+1` exists. If it does, we compare the nodes `l` and `r` with the function `min`. Otherwise, the index `j` is equal to `l`.

```
  else
    let r = l + 1 in
    let j = if r < n then min h l r else l in
```

Finally, we compare the value of the node `j` with that of `x`.

```
    if X.compare (A.get h j) x < 0 then j else i
```

Moving down the heap is implemented by a recursive function `move_down` that takes as parameters the heap `h`, the value `x` that is moving down, and the index `i` of the candidate node. We begin by calling the function `smallest_node` to calculate the index `j` that should receive the value `x`.

```
let rec move_down h x i =
  let j = smallest_node h x i in
```

If `j = i`, the descent is complete and it suffices to store the value `x` at index `i`.

```
  if j = i then
    A.set h i x
```

Otherwise, we move the value of the node `j` up to the node `i` and then continue moving down from `j`.

```
  else begin
    A.set h i (A.get h j);
    move_down h x j
  end
```

The function for removing the smallest element of a heap `h` is then the following:

```
let remove_min h =
  let n = A.length h - 1 in
  if n < 0 then invalid_arg "remove_min";
  let x = A.get h n in
  A.resize h n;
  if n > 0 then move_down h x 0
```

We consider three cases: If the heap is empty, we raise an exception. Next, if it contains only one element, it is not necessary to call the function `move_down`. Finally, if the heap contains more than one element, it is worthwhile to resize the array with `A.resize`. The complete code is given in programs 63 (see page 279) and 64.

Program 64 — An imperative heap data structure (2/2).

```
  let min h l r =
    if X.compare (A.get h r) (A.get h l) < 0 then r else l

  let smallest_node h x i =
    let l = 2 * i + 1 in
    let n = A.length h in
    if l >= n then i else
      let r = l + 1 in
      let j = if r < n then min h l r else l in
      if X.compare (A.get h j) x < 0 then j else i

  let rec move_down h x i =
    let j = smallest_node h x i in
    if j = i then A.set h i x
    else begin A.set h i (A.get h j); move_down h x j end

  let remove_min h =
    let n = A.length h - 1 in
    if n < 0 then invalid_arg "remove_min";
    let x = A.get h n in
    A.resize h n;
    if n > 0 then move_down h x 0
end
```

Complexity

It is clear that the functions `create`, `is_empty`, and `get_min` have constant cost. In case of the functions `add` and `remove_min`, their cost is bounded by the height of the heap. Indeed, in one case, we move from the leaves towards the root, and in the other, we move from the root towards the leaves. Since the heap is a complete binary tree, its height is logarithmic in the number of elements. For a priority queue containing N elements, the functions `add` and `remove_min` therefore have cost $O(\log N)$. A simpler way of looking at it consists in noting that the function `move_up` divides its argument `i` by two at each recursive call. Therefore, it cannot be called more than $O(\log N)$ times. Similarly, the function `move_down` multiplies its argument by two at each recursive call and, therefore, cannot be called more than $O(\log N)$ times. Of course, all of this applies only in an amortized manner by virtue of the calls to `resize`, which are amortized $O(1)$, as we showed in chapter 4.

The space complexity depends on the implementation of the resizeable array. If we use the implementation presented at the start of chapter 4, where the size of the array was doubled when it had to be increased, the array contains at most two times more elements than required. A priority queue containing N elements therefore occupies at most $2N$ memory words.

6.4 Persistent Priority Queues

In this section, we present a data structure of *persistent* priority queues, whose signature is given in program 65.

As in the previous section, we use a heap data structure, except that here it is directly represented by a binary tree. Unlike in case of AVL trees, no extra information is stored in the nodes for the purposes of balancing. We will see later that these heaps nevertheless have amortized logarithmic complexity. We say these trees are *self-balancing*.

The functor of persistent priority queues therefore takes the following form:

```
module Make(X: Ordered) :
  PersistentPriorityQueue with type elt = X.t =
```

Program 65 — Minimal signature for persistent priority queues.

```
module type PersistentPriorityQueue = sig
  type t
  type elt
  val empty : t
  val is_empty : t -> bool
  val add : elt -> t -> t
  val get_min : t -> elt
  val remove_min : t -> t
end
```

```
struct
  type elt = X.t
```

The argument `X`, of type `Ordered`, defines the type of elements. These must be equipped with a comparison function corresponding to the priority order (see program 33 of chapter 5 for the signature `Ordered`). The type `elt` of the elements of the queue is thus a synonym for `X.t`. The type `t` is the type of binary trees.

```
  type t = Empty | Node of t * elt * t
```

The definitions of the empty heap and the function `is_empty` are clear.

```
  let empty = Empty
  let is_empty h =
    h = Empty
```

The structure of the heap gives immediate access to the smallest element.

```
  let get_min = function
    | Empty -> invalid_arg "get_min"
    | Node (_, x, _) -> x
```

All the subtlety of this structure lies in the function `merge` that merges two heaps. Assuming we have already written this function, it is easy to write the functions `add` and `remove_min`. Indeed, the addition of an element `x` to a heap `h` consists in merging the heap containing the single element `x` with `h`.

```
let add x h =
  merge (Node (Empty, x, Empty)) h
```

Similarly, the removal of the smallest element of a heap consists in merging the two child heaps of the root.

```
let remove_min = function
  | Empty -> invalid_arg "remove_min"
  | Node (a, _, b) -> merge a b
```

All we have to do now is write the function `merge` that merges two heaps `ha` and `hb`. If one of the two heaps is empty, how to do so is clear.

```
let rec merge ha hb = match ha, hb with
  | Empty, h | h, Empty ->
      h
```

If, however, none of the heaps is empty, we construct the resulting heap as follows. Its root is the smallest of the two roots of `ha` and `hb`. Assume that the root of `ha` is smaller.

```
    | Node (la, xa, ra), Node (lb, xb, rb) ->
        if X.compare xa xb <= 0 then
```

We must now determine the two subtrees of `Node (_, xa, _)`. There are several possibilities that result from recursively calling `merge` on two of the three trees `la`, `ra`, and `hb`, and from choosing the result as the left or right subtree. Among these possibilities, we choose the one that performs a rotation of the subtrees from right to left, so as to ensure self-balancing. Thus, `ra` takes the position of `la`, and `la` is merged with `hb`.

```
          Node (ra, xa, merge la hb)
```

The other situation, in which the root of `hb` is smaller, is symmetrical.

```
        else Node (rb, xb, merge lb ha)
```

The complete code of persistent priority queues is given in program 66.

Program 66 — Persistent priority queues.

```
module Make(X: Ordered) :
  PersistentPriorityQueue with type elt = X.t =
struct
  type elt = X.t
  type t = Empty | Node of t * elt * t

  let empty =
    Empty

  let is_empty h =
    h = Empty

  let get_min = function
    | Empty -> invalid_arg "get_min"
    | Node (_, x, _) -> x

  let rec merge ha hb = match ha, hb with
    | Empty, h | h, Empty ->
        h
    | Node (la, xa, ra), Node (lb, xb, rb) ->
        if X.compare xa xb <= 0 then
          Node (ra, xa, merge la hb)
        else
          Node (rb, xb, merge lb ha)

  let add x h =
    merge (Node (Empty, x, Empty)) h

  let remove_min = function
    | Empty -> invalid_arg "remove_min"
    | Node (a, _, b) -> merge a b
end
```

Complexity

The functions `empty`, `is_empty`, and `get_min` clearly have constant cost. The analysis is subtler in case of the function `merge`, and hence `add` and `remove_min`. In the worst case, `merge` can have a cost as large as the total number of elements. However, for a sequence of `add` and `remove_min` operations, executed successively on a heap, this worst case cannot occur each time. We can show that the *amortized* cost of each operation is, in fact, $O(\log N)$, where N is the number of elements in the heap. For an analysis of this complexity, we refer the reader to the article by Sleator and Tarjan introducing self-balancing heaps [22] or the chapter on heaps by Okasaki in *The Fun of Programming* [10].

6.5 Exercises

Linked Lists

6.1 Write a function `iter : ('a -> unit) -> 'a t -> unit` that traverses the elements of a queue in the order in which they were inserted.

6.2 Write a function `transfer : 'a t -> 'a t -> unit` that takes two queues `q1` and `q2` as arguments, moves all the elements of `q1` into `q2`, and empties the queue `q1`. Make sure that this operation executes in constant time.

6.3 Modify the queue structure of program 58 to keep track of the total number of elements throughout. You can make type `t` into a record with a mutable field containing the number of elements of the queue.

Pairs of Lists

6.4 Consider the case of a queue `q`, in which the list representing the front is empty. If the operation `pop` is performed multiple times, then the reversal of the list `i`, representing the back of the queue, will also be performed multiple times. To remedy this, we can memoize the reversal of `i` in the original queue `q` by means of a side effect. Accordingly, we adopt an equivalent representation of the queue `q` that is still persistent. Modify the type `'a t` as follows:

```
type 'a t = { mutable o : 'a list; mutable i : 'a list }
```

Reimplement the operations on these queues. The mutable nature of the fields of this type does not allow defining a polymorphic value `empty`. It suffices to replace it by a function `create : unit -> 'a t`.

6.5 Note that the representation of queues as pairs of lists is symmetrical. We can therefore perform insertion and removal operations from the two ends of the queue with equal facility. We call the resulting structure a double-ended queue or *dequeue*. We can name the four operations as follows:

```
val push_front : 'a -> 'a t -> 'a t
val push_back : 'a -> 'a t -> 'a t
val pop_front : 'a t -> 'a * 'a t
val pop_back : 'a t -> 'a * 'a t
```

The functions `push_back` and `pop_front` correspond to the functions `push` and `pop` already written. Write the functions `push_front` and `pop_back`.

6.6 The double-ended queues of the previous exercise may prove inefficient. Indeed, alternating removals from the two ends of the queue can end up requiring many list reversals. To remedy this, we can try to balance the lengths of the two lists, so as to ensure a minimum proportion of elements at each end. For this, we let $c \geq 2$ be a constant and impose the following invariant on the lengths l_i and l_o of the two lists:

$$l_i \leq c \times l_o + 1 \quad \text{and} \quad l_o \leq c \times l_i + 1$$

Modify the type `'a t` as follows, to keep track of the length of each list.

```
type 'a t = { li : int; i : 'a list; lo : int; o : 'a list }
```

Reimplement the operations for double-ended queues. For more details concerning this structure, and notably for its analysis, you can consult the book by Okasaki [19, sec. 8.4].

Heap Structures

6.7 You can use the heap structure to easily implement an efficient sorting algorithm, called *heapsort*. The idea is as follows: We insert all the elements to be sorted in a heap, and then extract them successively with the functions `get_min` and `remove_min`. Add a function `sort : X.t array -> unit` to the functor `Make` to sort an array using this algorithm. If the complexity of the heap operations is logarithmic (which is the case when you use resizeable arrays of constant amortized complexity), then the complexity of this sorting algorithm is $O(n \log n)$, that is, optimal. Heapsort will be described in detail in chapter 12.

6.8 Several different elements of a heap can have the same priority. This is, for example, the case when the elements are pairs for which the function `compare` takes into account only one of the two components. Show that the order of insertion in a priority queue is preserved, that is, that the elements of equal priority are extracted from the queue in the same order in which they were inserted.

6.9 We wish to add an operation `remove` to remove an element from a priority queue. Several problems arise. On the one hand, the same element may occur several times in the queue. On the other, to implement the removal of an element x efficiently, it is necessary to determine at which index it appears without traversing the entire array, which is not possible with the current structure.

We propose to resolve these two problems as follows. When an element is added in the priority queue, we return a "pointer" to the user, that may be used subsequently to remove this element. The signature is accordingly modified as follows:

```
type pointer
val add: elt -> t -> pointer
```

Internally, the type `pointer` contains the index where the corresponding element of the array is stored. This index is updated when the element is moved within the array. In practice, it is simplest to also include the value of the element along with the index:

```
type pointer = { elt: elt; mutable index: int }
```

The resizeable array then contains values of type `pointer` rather than `elt`. Modify the priority queue structure following this idea and provide a function `remove: pointer -> t -> unit`. Hint: To remove the element situated at index i, replace it with the final element of the resizeable array. It suffices then to re-establish the heap property, by using either `move_up` or `move_down`. Show that the complexity of `remove` is in $O(\log n)$.

6.10 In the context of the above exercise, a value of type `pointer` may reference an element that is no longer in the queue. A call to `remove` could therefore corrupt the structure. Remedy this problem by invalidating all pointers corresponding to an element that is no longer in the queue, so that the function `remove` is able to fail cleanly.

6.11 It can be interesting to have priority queues that provide an operation `change_priority` to modify the priority of an existing element. Reusing the idea of exercise 6.9, provide the following new operation:

```
val change_priority : pointer -> elt -> t -> unit
```

This operation modifies the value of an element already present in the queue. Show that the complexity of `change_priority` is in $O(\log n)$.

6.12 In certain cases, we may wish to distinguish the priority of an element from its value. Modify the implementation of priority queues to give them the following signature:

```
module Make(P: Ordered)(V: WithDummy) :
sig
  type t
  type priority = P.t
  type value = V.t
  val create : unit -> t
  val is_empty : t -> bool
  val add : priority -> value -> t -> unit
  val get_min : t -> priority * value
  val remove_min : t -> unit
end
```

The type `P.t` is that of priorities and `V.t`, that of values. As the signatures indicate, the first is equipped with a function `compare` and the second with a default value.

Self-Balancing Heaps

6.13 Redo exercise 6.7 (heapsort) with self-balancing heaps. What is the complexity of this sorting algorithm?

6.14 Using a reference, wrap the code of program 66 in a module with the imperative signature of program 57.

7 Graphs

Graphs are a fundamental data structure in computer science. A graph consists of a set of *vertices* linked by *edges*. We are accustomed to visualizing graphs as illustrated in figure 7.1.

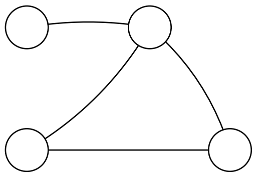


Figure 7.1: Example of a graph.

Formally, a graph is a set V of vertices and a set E of edges, which are *unordered pairs* of vertices. If $\{x, y\} \in E$, we say that the vertices x and y are adjacent. As the adjacency relation is symmetric, we speak of an *undirected graph*.

We can also define the notion of *directed graph* by taking E to be a set of *ordered pairs* of vertices rather than unordered pairs. We then speak of *arcs* rather than edges. If $(x, y) \in E$, we say that y is a successor of x and write

$x \to y$. An example of a directed graph is given in figure 7.2.

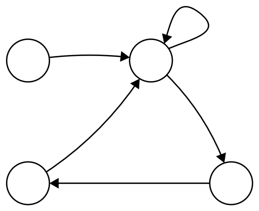


Figure 7.2: Example of a directed graph.

An arc from a vertex to itself, as in this example, is called a *cycle*. The *indegree* (resp. *outdegree*) of a vertex is the number of arcs that point towards it (resp. that point away from it). A *path* from a vertex u to a vertex v is a sequence $x_0, \ldots, x_n$ of vertices such as $x_0 = u$, $x_n = v$ and $x_i \to x_{i+1}$ for $0 \leq i < n$. Such a path is of length n; it contains n arcs.

Vertices, like arcs, may carry information. We then speak of a *labeled graph*. An example of a directed, labeled graph is given in figure 7.3.

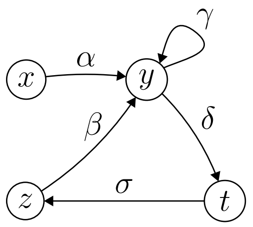


Figure 7.3: Example of a directed, labeled graph.

It is important to note that the label of a vertex is not the same as the vertex itself. In particular, two vertices may bear the same label. Formally, a labeled graph therefore comprises two additional functions giving, respectively, the label of a vertex in V and the label of an arc in E.

In what follows, we will use the term edge for both directed and undirected graphs.

Basic Operations on Graphs

Before defining data structures to represent graphs, it is necessary to consider the operations that these structures should provide. We do this independently of any particular representation, with a view to constructing generic algorithms. We may distinguish the following basic operations:

- *construction* operations, such as the creation of a graph, the addition or removal of a vertex or an edge;
- *access* operations, such as testing whether a vertex or an edge belongs to a graph, accessing the degree of a vertex, its label, the number of vertices or edges;
- basic *traversal* operations, such as the traversal of all vertices, of all successors of a given vertex, or of all edges.

As we will see shortly, there is no single structure of graphs that provides all these operations in an optimal manner. At the end of this chapter, we compare different graph data structures according to the complexity of the different operations above.

7.1 Adjacency Matrix

We consider in this section the case in which vertices are represented by integers and, more precisely, by consecutive integers $0, \dots, N-1$. In other words, we have $V = \{0, \dots, N-1\}$.

Program 67 (see following page) gives a minimal signature for such graphs. The type `t` is that of graphs. It is deliberately made abstract as we are going to present two different implementations of this signature. The creation function, `create`, takes the value N as argument, and the function `nb_vertex` gives it back. Crucially, we assume here that the graph contains N vertices from the outset. Consequently, we do not provide operations to add or remove vertices, or to test whether they belong to the graph. In case of edges, however, we do have these operations, namely, `add_edge`, `remove_edge`, and `mem_edge`. Note that

Program 67 — Minimal signature for graphs with integer vertices.

```
type vertex = int
type t

val create : int -> t
val nb_vertex : t -> int

val mem_edge : t -> vertex -> vertex -> bool
val add_edge : t -> vertex -> vertex -> unit
val remove_edge : t -> vertex -> vertex -> unit

val iter_succ : (vertex -> unit) -> t -> vertex -> unit
val iter_edge : (vertex -> vertex -> unit) -> t -> unit
```

the functions to add and remove edges correspond to an imperative structure. The signature provides only the functions to traverse the successors of a vertex (`iter_succ`) and all edges (`iter_edge`). Traversing all vertices can be easily achieved with a `for` loop from 0 to $N-1$. Finally, note that this signature works equally well for both directed and undirected graphs.

The signature is deliberately minimal. In particular, graphs are not labeled. Some of the exercises of this section propose extensions.

A straightforward representation of such a graph is a matrix M of size $N \times N$, where each element $M_{i,j}$ indicates the presence of an edge between the vertices i and j. As the graphs are assumed to be unlabeled, it suffices to use a matrix of booleans:

```
type vertex = int
type t = bool array array
```

The definition of the function that creates a graph without any edges is clear: We construct a square matrix, initialized with `false`.

```
let create n = Array.make_matrix n n false
```

Program 68 — Graphs via adjacency matrices.

```
type vertex = int
type t = bool array array

let create n = Array.make_matrix n n false
let nb_vertex = Array.length

let mem_edge g v1 v2 = g.(v1).(v2)
let add_edge g v1 v2 = g.(v1).(v2) <- true
let remove_edge g v1 v2 = g.(v1).(v2) <- false

let iter_succ f g v =
  Array.iteri (fun w b -> if b then f w) g.(v)
let iter_edge f g =
  for v = 0 to nb_vertex g - 1 do iter_succ (f v) g v done
```

```
let nb_vertex = Array.length
```

In what follows, we assume that all graphs are directed. Exercise 7.4 indicates the modifications required for undirected graphs. Adding, removing, or testing for the presence of an edge can all be done in constant time:

```
let mem_edge g v1 v2 = g.(v1).(v2)
let add_edge g v1 v2 = g.(v1).(v2) <- true
let remove_edge g v1 v2 = g.(v1).(v2) <- false
```

To apply a function `f: vertex -> unit` to all successors of a vertex `v` of a graph `g`, it suffices to traverse the row `g.(v)` of the matrix, for example, with `Array.iteri`. We test whether an edge is present before applying the function `f` to the corresponding vertex.

```
let iter_succ f g v =
  Array.iteri (fun w b -> if b then f w) g.(v)
```

To apply a function `f: vertex -> vertex -> unit` to all edges of a graph `g`, we begin by traversing the vertices of `g` with a `for` loop. Then, for each vertex `v`, we traverse its outgoing edges using `iter_succ`. The function passed to `iter_succ` is obtained by partially applying `f` to `v`.

```
let iter_edge f g =
  for v = 0 to nb_vertex g - 1 do iter_succ (f v) g v done
```

The complete code for graphs via adjacency matrices is given in program 68. The structure of the adjacency matrix may be adapted to the case of labeled graphs (see exercise 7.3) and/or undirected graphs (see exercise 7.4).

Complexity

The table in figure 7.4 summarizes the complexity of the main operations on adjacency matrices. In general, the cost is expressed as a function of the number N of vertices and the number E of edges. In the particular case of adjacency matrices, however, the cost depends only on N.

operation	cost
`mem_edge`	$O(1)$
`add_edge`	$O(1)$
`iter_succ`	$O(N)$
`iter_vertex`	$O(N)$
`iter_edge`	$O(N^2)$

Figure 7.4: Cost of operations on adjacency matrices.

Of course, we assume that the functions passed as arguments of `iter_succ`, `iter_vertex`, and `iter_edge` have constant cost. From this table, we deduce that adjacency matrices are well adapted to *dense* graphs, that is, graphs in which the number of edges E is of the order of N^2. Indeed, the outdegree of each vertex is then of the order of N, so that `iter_succ` is optimal. Similarly, `iter_edge` must traverse a number of edges of the order of N^2 and is therefore also optimal.

As for the cost in space, an adjacency matrix clearly occupies quadratic space, namely $(N + 1)^2$ words in case of a boolean matrix (see section *3.2 Runtime Model*). We can reduce this cost by a constant factor by using bit vectors (see section *4.2 Bit Vectors*). Exercise 7.5 proposes such an optimization. It is important to note that this optimization is only possible if the edges are not labeled.

7.2 Adjacency Lists

In case of sparse graphs, an alternative to adjacency matrices consists in using an array containing the list of the successors of each vertex. We speak of *adjacency lists*. The type of graphs follows directly from this definition.

```
type vertex = int
type t = vertex list array
```

An empty graph is represented by an array containing only empty lists.

```
let create n = Array.make n []
let nb_vertex = Array.length
```

To test whether an edge is present between vertices `v1` and `v2` in a graph `g`, we traverse the adjacency list `g.(v1)` in search of `v2`. We do this here with the function `List.mem`.

```
let mem_edge g v1 v2 =
  List.mem v2 g.(v1)
```

We add an edge between `v1` and `v2` by adding `v2` to the list `g.(v1)`. We only do this if the edge does not already exist, to avoid unnecessary duplicates in the adjacency list.

```
let add_edge g v1 v2 =
  if not (mem_edge g v1 v2) then g.(v1) <- v2 :: g.(v1)
```

We remove the edge `v1` $\rightarrow$ `v2` by removing the possible instance of `v2` in the list `g.(v1)`. We can do this easily using `List.filter`.

```
let remove_edge g v1 v2 =
  g.(v1) <- List.filter ((<>) v2) g.(v1)
```

Note, nevertheless, that this use of `List.filter` has two defects: On the one hand, it does not take advantage of the absence of duplicates in the list to stop early. On the other, it unnecessarily reconstructs the entire list `g.(v1)` in the case where `v2` does not appear. Exercise 7.8 proposes a remedy for these issues. Similarly, the fact that the vertices are integers could be used to optimize the three preceding operations by keeping the adjacency lists sorted (see exercise 7.9).

The traversal of the successors of a vertex `v` of a graph `g` is simpler than in case of adjacency matrices. It is trivially implemented by applying `List.iter` to the adjacency list `g.(v)`.

```
let iter_succ f g v = List.iter f g.(v)
```

The code of `iter_edge`, by contrast, is the same as that for adjacency matrices.

```
let iter_edge f g =
  for v = 0 to nb_vertex g - 1 do iter_succ (f v) g v done
```

The complete code for graphs via adjacency lists is given in program 69.

Complexity

The table in figure 7.5 summarizes the complexity of the main operations on adjacency lists. Here, δ denotes the outdegree of a vertex.

Note that, unlike in case of adjacency matrices, the operations `mem_edge` and `add_edge` are no longer in $O(1)$ but in $O(\delta)$. By contrast, `iter_succ` and `iter_edge` now have optimal complexity.

As for the cost in space, adjacency lists have optimal complexity $O(N+E)$. To be precise, the cost is $N+3E+1$ words (see section *3.2 Runtime Model*).

7.3 Adjacency Dictionaries

One clear drawback of the two preceding data structures is the condition that vertices must be represented by integers and, moreover, by consecutive integers.

Program 69 — Graphs via adjacency lists.

```
type vertex = int
type t = vertex list array

let create n = Array.make  n []
let nb_vertex = Array.length

let mem_edge g v1 v2 =
  List.mem v2 g.(v1)

let add_edge g v1 v2 =
  if not (mem_edge g v1 v2) then g.(v1) <- v2 :: g.(v1)

let remove_edge g v1 v2 =
  g.(v1) <- List.filter ((<>) v2) g.(v1)

let iter_succ f g v =
  List.iter f g.(v)

let iter_edge f g =
  for v = 0 to nb_vertex g - 1 do iter_succ (f v) g v done
```

operation	cost
`mem_edge`	$O(\delta)$
`add_edge`	$O(\delta)$
`iter_succ`	$O(\delta)$
`iter_vertex`	$O(N)$
`iter_edge`	$O(E)$

Figure 7.5: Cost of operations on adjacency lists.

This representation is adapted for graphs whose sets of vertices do not change. If, on the other hand, we wish to add or remove vertices dynamically, we have to modify the preceding structures. We can, for example, use resizeable arrays to permit the addition of vertices. To remove vertices, we could imagine marking them so as to exclude them from the graph, but this would have an impact on the complexity of the functions `iter_succ`, `iter_vertex`, and `iter_edge`.

Ideally, we would like a graph structure whose vertices are not necessarily integers and can be added or removed dynamically. The structure would also combine the respective advantages of adjacency matrices and adjacency lists in time and in space. Such a structure exists: It suffices to combine the structure of a dictionary with that of a set. A graph is thus nothing but a dictionary in which each vertex is associated with the set of its successors. In other words, we retain the idea of adjacency lists, but replace the array with a dictionary, and lists by sets. We will call this structure an *adjacency dictionary*. In this way, vertices are not limited to integers, and may be easily added and removed. As for efficiency, it suffices to use a hash table for the dictionary to obtain optimal complexity. We may also use a hash table to represent the set of successors of a vertex.

We can therefore parametrize our adjacency dictionary structure by a module `H` of hash tables (such as the one in section *5.3 Hash Tables* or the module `Hashtbl` of the OCaml standard library).

```
module Graph(H: HashTable) = struct
```

For clarity, we introduce a module `V` for the dictionary, as a synonym of the module `H`. The type of the vertices is then that of the keys of the module `V`.

```
module V = H
type vertex = V.key
```

Similarly, we introduce a module `E` for the sets of adjacent vertices, also a synonym of the module `H`. The idea here is to represent a set of vertices by a hash table, by associating each element of the set with the value `()`.

```
module E = H
```

The type of graphs is therefore the following:

```
type t = (unit E.t) V.t
```

If the edges were labeled, it would suffice to associate each successor w of v with the label of the edge $v \to w$ rather than with the value `()` (see exercise 7.10).

Creating a graph becomes merely a matter of constructing an empty dictionary. The number of vertices is obtained simply as the number of keys of the dictionary.

```
let create () = V.create ()
let nb_vertex g = V.length g
```

We check if a vertex `v` belongs to the graph `g` with `V.mem`, that is, by testing if it is a key in the dictionary. We add `v` by creating a binding associating `v` with a new hash table.

```
let mem_vertex g v = V.mem g v
let add_vertex g v = V.add g v (E.create ())
```

We note that the function `add_vertex` assumes that `v` is not yet present in the graph `g`. Otherwise, we would have to first verify the presence of `v` using `mem_vertex`. The removal of a vertex `v` is more delicate: Not only must its binding be removed from the dictionary, it is also necessary to remove every instance of `v` from the adjacency sets of other vertices (here written as `s`).

```
let remove_vertex g v =
  V.remove g v;
  V.iter (fun _ s -> E.remove s v) g
```

For greater efficiency, we should access only the predecessors of `v`. Exercise 7.14 proposes such an improvement.

To add, remove, or test for the presence of an edge between two vertices `v1` and `v2`, we begin by retrieving the adjacency set of `v1` using `V.find g v1`. We then use respectively the functions `E.replace`, `E.remove`, and `E.mem`.

```
let mem_edge g v1 v2 = E.mem (V.find g v1) v2
let add_edge g v1 v2 = E.replace (V.find g v1) v2 ()
let remove_edge g v1 v2 = E.remove (V.find g v1) v2
```

These three functions assume that `v1` is already present in the graph, that is, that `V.find g v1` does not fail. As for the function `add_edge`, we use `E.replace` rather than `E.add`, to avoid duplicates in the adjacency set.

To traverse the set of vertices of a graph `g`, it suffices to traverse the set of keys of the dictionary using `V.iter`. Similarly, we traverse the successors of a vertex `v` by traversing its adjacency set using `E.iter`.

```
let iter_vertex f g = V.iter (fun v _ -> f v) g
let iter_succ f g v = E.iter (fun w _ -> f w) (V.find g v)
```

Here too, we assume that `iter_succ` is called on an existing vertex `v`. To traverse all edges, it suffices to compose `iter_vertex` and `iter_succ` as we did for adjacency matrices and adjacency lists. However, it would be unnecessarily costly to access the adjacency set of `v` using `V.find g v`. Instead, it suffices to compose directly `V.iter` and `E.iter`.

```
let iter_edge f g = V.iter (fun v s -> E.iter (f v) s) g
```

The complete code of this graph structure is given in program 70.

For undirected, labeled graphs or graphs with multiple edges between two vertices, the reader is referred to exercises 7.10–7.12.

Complexity

If we assume that dictionaries and sets are implemented by hash tables, we obtain the performances given in the table of figure 7.6.

Program 70 — Graphs via adjacency dictionaries.

```
module Graph(H: HashTable) = struct
  module V = H
  type vertex = V.key

  module E = H

  type t = (unit E.t) V.t

  let create () = V.create ()
  let nb_vertex g = V.length g

  let mem_vertex g v = V.mem g v
  let add_vertex g v = V.add g v (E.create ())
  let remove_vertex g v =
    V.remove g v;
    V.iter (fun _ s -> E.remove s v) g

  let mem_edge g v1 v2 = E.mem (V.find g v1) v2
  let add_edge g v1 v2 = E.replace (V.find g v1) v2 ()
  let remove_edge g v1 v2 = E.remove (V.find g v1) v2

  let iter_vertex f g = V.iter (fun v _ -> f v) g
  let iter_succ f g v = E.iter (fun w _ -> f w) (V.find g v)
  let iter_edge f g = V.iter (fun v s -> E.iter (f v) s) g
end
```

operation	cost	operation	cost	operation	cost
`mem_vertex`	$O(1)$	`mem_edge`	$O(1)$	`iter_succ`	$O(\delta)$
`add_vertex`	$O(1)$	`add_edge`	$O(1)$	`iter_vertex`	$O(N)$
`remove_vertex`	$O(N)$	`remove_edge`	$O(1)$	`iter_edge`	$O(E)$

Figure 7.6: Cost of operations on adjacency dictionaries.

We observe that the complexities are optimal, except in case of `remove_vertex` (see exercise 7.14). As for the cost in space, it is necessary to make an assumption regarding the sizes of the hash tables. If we assume that the number of buckets is at least twice the number of bindings in each hash table then the cost in space of a hash table containing K inputs is $6K+4$ words. Hence, for a graph containing N vertices and E edges, the total cost is:

$$(6N+4)+\sum_{\mathtt{v}}(6\delta_{\mathtt{v}}+4)$$

This is $10N+6E+4$ words in total.

Persistent Graphs

Graphs implemented using adjacency dictionaries as presented above are imperative due to the imperative nature of hash tables. We naturally obtain persistent graphs by substituting hash tables with a persistent data structure. For example, we may use any dictionary having the signature `Map.S` of OCaml (see exercise 7.15), in particular all those discussed in chapter 5.

7.4 Performance Comparison

We recapitulate here the performance of the different graph structures. The time complexity of the main operations are summarized in the table of figure 7.7. As we have already highlighted above, adjacency dictionaries offer the best possible complexity for each operation.

As for the cost in space, recall the cost in number of words for each graph structure:

operation	adjacency matrices	adjacency lists	adjacency dictionaries
`mem_vertex`	—	—	$O(1)$
`add_vertex`	—	—	$O(1)$
`remove_vertex`	—	—	$O(N)$
`mem_edge`	$O(1)$	$O(\delta)$	$O(1)$
`add_edge`	$O(1)$	$O(\delta)$	$O(1)$
`remove_edge`	$O(1)$	$O(\delta)$	$O(1)$
`iter_succ`	$O(N)$	$O(\delta)$	$O(\delta)$
`iter_vertex`	$O(N)$	$O(N)$	$O(N)$
`iter_edge`	$O(N^2)$	$O(E)$	$O(E)$

Figure 7.7: Comparison of different graph data structures.

- adjacency matrix: $(N+1)^2$ words;
- adjacency lists: $N+3E+1$ words;
- adjacency dictionary: $10N+6E+4$ words.

It is clear that adjacency lists are more economical in terms of memory than adjacency dictionaries. The comparison with adjacency matrices is more difficult. Indeed, it depends on the density of the graph, that is, the ratio between the number of vertices N and the number of edges E. For a very dense graph, in which E is close to N^2, the adjacency matrix is more economical. In the case of a sparse graph, for example where $E \sim N$, adjacency lists are more economical. In general, lists are preferable to matrices when $E < N^2/3$ and, similarly, dictionaries are preferable to matrices when $E < N^2/6$. Lists are always preferable to dictionaries *in terms of space.*

Of course, in choosing a structure for graphs, the complexity in space of the different operations must be weighed against their complexity in time.

7.5 Exercices

Adjacency Matrices

7.1 The transposition of a directed graph G, denoted by G^R, is the graph having the same vertices as G and which contains an edge $x \to y$ if and only if there exists an edge $y \to x$ in G. Write a function `reverse: graph -> graph` that calculates the transposition of a given graph.

7.2 Add an operation `nb_edge: t -> int` that gives the number of edges in constant time. Hint: Keep track of the number of edges in the structure of the graph by updating its value in `add_edge` and `remove_edge`.

7.3 Modify adjacency matrices for graphs in which edges are labeled by a given type `label`.

7.4 The easiest way to represent undirected graphs is by preserving the same structure as for directed graphs, but by maintaining the invariant that, for each edge $a \to b$, there is also an edge $b \to a$. Modify the operations `add_edge` and `remove_edge` of adjacency matrices and lists accordingly. Also modify `iter_edge` so that it traverses each edge only once.

7.5 Modify the adjacency matrix structure to use bit vectors (see section *4.2 Bit Vectors*) rather than boolean arrays. What is the gain in terms of space?

7.6 The transitive closure of a graph G is a graph T having the same vertices as G, and an edge between i and j when there is a *path* between i and j in G. When the graph is represented by a boolean matrix, the transitive closure may be calculated in $O(N^3)$ time using the Floyd-Warshall algorithm. The pseudo-code is as follows:

```
$T \leftarrow G$
for $k$ from 0 to $N-1$
   for $i$ from 0 to $N-1$
      for $j$ from 0 to $N-1$
         $T_{i,j} \leftarrow T_{i,j}$ or $(T_{i,k}$ and $T_{k,j})$
```

The idea consists in determining, for increasing values of k, if there exists a path between i and j that uses only intermediate vertices smaller than k. The last line of the pseudo-code considers the two cases of a path between i and j, passing via k, or not. Write a function `transitive_closure: t -> t` implementing this algorithm.

Adjacency Lists

7.7 Redo exercises 7.2 to 7.4 for adjacency lists.

7.8 Modify the function `remove_edge` on adjacency lists so that (1) it does not modify the adjacency list if the edge to be removed does not exist, and (2) it makes use of the fact that adjacency lists do not contain duplicates to stop once the edge is found.

7.9 Modify the operations on adjacency lists to maintain the invariant that adjacency lists are sorted. Make use of this property to optimize some of the operations.

Adjacency Dictionaries

7.10 Modify the code of program 70 for graphs whose edges are labeled by values of any type. More precisely, give graphs the polymorphic type `type 'a t = ('a E.t) V.t`, where `'a` is the type of labels.

7.11 Modify the adjacency dictionary structure for undirected graphs drawing on exercise 7.4.

7.12 Modify the adjacency dictionary structure for graphs with *multiple edges*, that is, where two vertices may be linked by more than one edge. For unlabeled multi-edges, the dictionary must associate each vertex with the *multi-set* of its successors. For labeled multi-edges, we may distinguish two cases, depending on whether the edges between v_1 and v_2 can bear the same label or not.

7.13 Modify the functor of program 70 so that, instead of taking a module `H` of hash tables as argument, it takes:

- either a module `V` introducing a type `t` equipped with hash and comparison functions;
- or two modules, `M` and `S`, that respectively implement dictionaries and sets on the same type of vertices.

7.14 Modify the adjacency dictionary structure to keep track of the set of predecessors of a vertex in an efficient manner. In other words, provide an operation

```
    val iter_pred: (vertex -> unit) -> t -> vertex -> unit
```

whose cost is proportionate to the indegree. Use this function to reimplement the function `remove_vertex`.

7.15 Modify the functor of program 70 to obtain persistent graphs by replacing the argument `H` by a module `M` of signature `Map.S`.

8 Disjoint Sets

This chapter presents an imperative data structure that solves the problem of partitioning a finite set into disjoint subsets, called "classes." We wish to determine whether two elements belong to the same class and merge two classes into one. It is these two operations that give the structure its name: *union-find*.

8.1 The Basic Idea

Without loss of generality, we may assume that the set to be partitioned consists of the n integers $\{0, 1, \ldots, n-1\}$. Program 71 (see page 313) gives the signature of such a structure. The operation `create` n constructs a new partition of $\{0, 1, \ldots, n-1\}$ in which each element forms a class by itself. The operation `find` determines the class of an element by returning one of the integers as a distinguished representative. In particular, we can determine if two elements are in the same class by comparing the result given by `find` for each one. Finally, the operation `union` merges two classes of the partition, modifying the data structure in place.

The basic idea is to link the elements of a class between themselves. In each class, these links form a graph in which all paths lead to the representative,

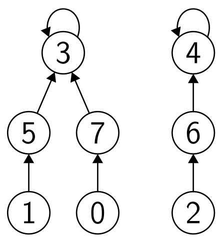


Figure 8.1: A partition of $\{0, 1, \ldots, 7\}$ into two sets.

which is the only element linked to itself. Figure 8.1 illustrates a situation in which the set $\{0, 1, \ldots, 7\}$ is partitioned into two classes, whose representatives are respectively 3 and 4.

It is possible to represent such a structure using individually allocated nodes (see exercise 8.5). Nevertheless it is simpler, and often more efficient, to use an array that links each integer to another integer of the same class. These links lead to the class representative, which is associated with itself in the array. Thus, the partition of figure 8.1 is represented by the following array:

0	1			…			7
7	5	6	3	4	3	4	3

The `find` operation simply follows the links until the representative is found. The `union` operation begins by finding the representatives of both elements and then links them together. We propose two improvements in the interest of efficiency. The first involves performing *path compression* during the search carried out by `find`: All the elements found along the path traversed to reach the representative are linked directly to it. The second consists in keeping track of the *rank* of each representative, that is, the maximum length of a path in its class. This information is stored in a second array and is used by the function `union` to choose the representative for the union of the classes.

Program 71 — Signature of the *union-find* structure.

```
module type UnionFind = sig
  type t
  val create : int -> t
  val find : t -> int -> int
  val union : t -> int -> int -> unit
end
```

8.2 Implementation

We now describe the code of the *union-find* structure. The type `t` is a record containing two arrays: `rank`, containing the rank of each class; and `link`, containing the links between elements.

```
type t = {
  rank: int array;
  link: int array;
}
```

The information contained in `rank` is only meaningful for those elements i that are representatives, that is, for which `link.`$(i) = i$. Initially, each element forms a class in itself (that is, it is its own representative), and the rank of every class is 0.

```
let create n =
  { rank = Array.make n 0;
    link = Array.init n (fun i -> i) }
```

The function `find` computes the representative of an element `i`. It is naturally written as a recursive function. We begin by calculating the element `p` linked to `i` in the array `t.link`. If it is `i` itself, `i` is the class representative, and we are done.

```
let rec find t i =
```

Program 72 — *Union-find* structure.

```
type t = {
  rank: int array;
  link: int array;
}

let create n =
  { rank = Array.make n 0;
    link = Array.init n (fun i -> i) }

let rec find t i =
  let p = t.link.(i) in
  if p = i then
    i
  else begin
    let r = find t p in
    t.link.(i) <- r;
    r
  end

let union t i j =
  let ri = find t i in
  let rj = find t j in
  if ri <> rj then begin
    if t.rank.(ri) < t.rank.(rj) then
      t.link.(ri) <- rj
    else begin
      t.link.(rj) <- ri;
      if t.rank.(ri) = t.rank.(rj) then
        t.rank.(ri) <- t.rank.(ri) + 1
    end
  end
```

```
    let p = t.link.(i) in
    if p = i then
      i
```

Otherwise, we recursively compute the representative `r` of `p` with `find t p`. However, before returning `r`, we perform path compression, that is, we link `i` to `r` directly.

```
    else begin
      let r = find t p in
      t.link.(i) <- r;
      r
    end
```

This way, the next time we call `find` on `i`, we will immediately find `r`.

The operation `union` merges the classes of two elements, `i` and `j`. We begin by calculating their representatives, `ri` and `rj`, respectively. If they are equal, there is nothing more to be done.

```
  let union t i j =
    let ri = find t i in
    let rj = find t j in
    if ri <> rj then begin
```

Otherwise, we compare the ranks of the two classes. If that of `ri` is strictly less than that of `rj`, we make `rj` the representative of the union.

```
      if t.rank.(ri) < t.rank.(rj) then
        t.link.(ri) <- rj
```

The rank does not have to be updated for this new class. Indeed, only the paths in the previous class of `ri` have had their lengths increased by one unit, and this new length does not exceed the rank of `rj`. When the rank of `rj` is smaller, we proceed symmetrically.

```
      else begin
        t.link.(rj) <- ri;
```

When the two classes have the same rank, we choose one of the two representatives arbitrarily as the representative of the union (here, `ri`). The rank information must then be updated, because in this case the length of the longest path in the class may increase by one unit.

```
    if t.rank.(ri) = t.rank.(rj) then
     t.rank.(ri) <- t.rank.(ri) + 1
   end
  end
```

Importantly the function `union` uses the function `find`, and therefore performs path compression even in the case when `i` and `j` belong to the same class. The complete code of the *union-find* structure is given in program 72 (see page 314).

Complexity

We can show that, thanks to path compression and the rank associated with each class, a sequence of m `find` and `union` operations, performed on a structure containing n elements, is executed in $O(m\,\alpha(n,m))$ total time, where α is a function that grows extremely slowly. In fact, it grows so slowly that we may consider it to be constant for all practical purposes, given the constraints that memory and time impose on the values of n and m. We may therefore assume that each operation takes amortized constant time. This analysis is complex and lies beyond the scope of this book. A detailed explanation may be found in *Introduction to Algorithms* [7, chap. 22].

For Further Information

The data structure presented in this chapter is attributed to McIlroy and Morris [3], and its complexity has been analyzed by Tarjan [23].

8.3 Exercises

8.1 The signature of the *union-find* structure is constraining because it requires prior knowledge of the number of elements. For greater flexibility, we may wish to add new elements dynamically, and without any restriction on the number of elements. The signature of such a structure could be the following:

```
type t
val create: unit -> t
val add: t -> int
val find: t -> int -> int
val union: t -> int -> int -> unit
```

The function `create` creates an empty structure. The function `add` adds a new element and returns it. (Elements are always integers). The signatures of `find` and `union` remain unchanged. Implement such a data structure using resizeable arrays (see section *4.1 Resizeable Arrays*).

8.2 Extend the *union-find* structure with an operation `num_classes` that returns the total number of classes.

```
val num_classes: t -> int
```

Ensure that `num_classes` executes in constant time by keeping track of this value as an additional field of the record.

8.3 Extend the *union-find* structure with an operation `iter_classes` that permits the traversal of the set of all class representatives.

```
val iter_classes: (int -> unit) -> t -> unit
```

Ensure that `iter_classes` executes in time proportionate to the total number of classes.

8.4 If the elements are not consecutive integers, we can replace the two arrays `rank` and `link` by a pair of hash tables. Reimplement the operations `create`, `find`, and `union` using this idea.

8.5 Instead of using arrays, another way of implementing the *union-find* structure consists in representing each class directly as an acyclic graph whose nodes are records containing the values `rank` and `link`. Each node is of the following type:

```
type elt = { mutable rank: int; mutable link: elt }
```

If we wish to keep track of extra information for each element, we may use the following type instead:

```
type 'a elt =
  { mutable rank: int; mutable link: 'a elt; data: 'a }
```

It is no longer necessary to keep track of any global information in the *union-find* structure, because each node contains all the necessary information. The signature of the *union-find* structure is then modified as follows:

```
type 'a elt
val create_node: 'a -> 'a elt
val find: 'a elt -> 'a elt
val union: 'a elt -> 'a elt -> unit
```

The function `create_node` constructs a class consisting of a single element, that is, a record whose field `link` points to itself. Write the operations `make`, `find`, and `union`. Note: The comparison of values of type `'a elt` must be done using physical equality.

8.6 Replace the arrays `rank` and `link` with persistent arrays (see section *4.4 Persistent Arrays*), and write a persistent *union-find* structure with the following signature:

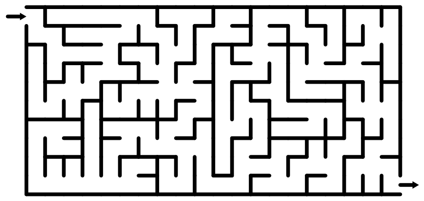
```
type t
val create: int -> t
val find: t -> int -> int
val union: t -> int -> int -> t
```

Note: To perform path compression, it is convenient to modify the contents of the field `link` using side effects. We accordingly define the following type:

```
type t = { rank: int A.t; mutable link: int A.t }
```

Here, the type `A.t` is the type of persistent arrays, and the mutability of the field `link` can be used to record the effect of path compression before returning the result of the function `find`. Although the data structure is modified by a side effect, the modification cannot be observed: the representative of each element remains the same.

8.7 The *union-find* structure can be used to efficiently construct a perfect maze, that is, one in which there is one and only one path between any two cells. Here is an example of such a maze:



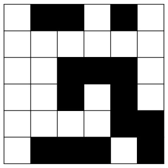
You may proceed as follows: Create a *union-find* structure whose elements are the different cells. The idea is that two cells are in the same class if and only if they are linked by a path. Initially, all the cells of the maze are separated from each other by walls. Next, consider all pairs of adjacent cells (vertically and horizontally) in random order. For each pair (c_1, c_2), compare the classes of the cells c_1 and c_2. If they are identical, there is nothing more to be done. Otherwise, delete the wall that separates c_1 and c_2, and merge the two classes using `union`. Write code that constructs a maze using this method.

Hint: The easiest way to traverse all pairs of adjacent cells in random order is by constructing an array containing all pairs, and then mixing them randomly using the *Knuth shuffle* (exercice 2.11, page 134).

Justify that at the end of the construction every cell is linked to every other cell by a single path.

8.8 The aim of this exercise is to color the *connected* components of a black-and-white image using different colors. A connected component is formed by pixels that have the same color as their neighbors. Two pixels are neighbors if they have a common border.

For example, in the following image of 6×6 pixels, there are two white connected components, and four black ones.



A black-and-white image is represented by a matrix of booleans. To compute its connected components, we use a *union-find* structure to group neighboring pixels having the same color in a single class. Write a function that takes as argument a matrix of booleans and returns a *union-find* structure containing its equivalence classes.

To assign a different color to each connected component, use a hash table that associates a different color with each representative. Write a function that displays the image with its connected components colored. Assume that you are given a function `new_color` of type `unit -> color` and a function `draw_pixel` of type `int -> int -> color -> unit`.

9 Zippers

In this chapter, we present a general technique to traverse data structures. This technique does not involve a specific traversal order and allows for the possibility of local modifications (inserting, removing, etc.). It goes by the name of *zipper*. We present it here in the context of lists and trees, but it can be adapted to many other data structures.

9.1 Zippers on Lists

Suppose we wish to "navigate" within a list of type `'a list`, that is, move from element to element, occasionally performing modifications. The image that comes to mind here is that of a cursor in a text editor, and the action of the keys of the keyboard, used to move, insert, delete characters, etc.

Consider a list containing seven elements, with the cursor placed after the third, as illustrated in figure 9.1.

We represent this situation with two lists, one containing the elements situated to the left of the cursor, and another containing the elements situated to the right. It is convenient to represent the first of these lists in reverse order, so that the element situated immediately to the left of the cursor is easily

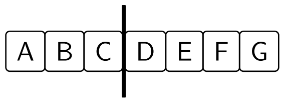


Figure 9.1: A cursor on a list.

accessible. Figure 9.2 illustrates this representation on the preceding example.

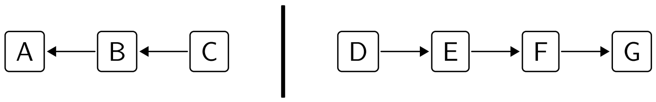


Figure 9.2: A zipper on a list.

This structure consisting of two lists is called a *zipper*. We define it using the following type:

```
type 'a zipper = { left: 'a list; right: 'a list; }
```

To construct such a zipper from a list `l`, with the cursor placed all the way to the left, we initialize the left list to `[]` and the right list to `l`.

```
let of_list l =
  { left = []; right = l }
```

Let us consider next the navigation and modification operations for the zipper.

Navigation Operations

The first navigation operation consists in moving the cursor to the right. This is only possible if the right list contains at least one element. In this case, we move the first element of this list to the head of the left list.

```
let move_right z = match z.right with
  | [] -> invalid_arg "move_right"
  | x :: r -> { left = x :: z.left; right = r }
```

The application of the function `move_right` to the example of figure 9.2 returns the zipper illustrated in figure 9.3.

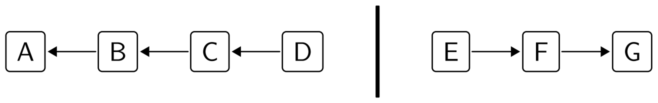


Figure 9.3: The zipper after a move to the right.

We similarly write a function `move_left` to move the cursor to the left (see program 73, following page).

To convert a zipper `z` into a list, it suffices to concatenate the two lists `z.left` and `z.right`, reversing the former. This is precisely what the function `List.rev_append` does.

```
let to_list z =
  List.rev_append z.left z.right
```

Equivalently, we could have moved the cursor all the way to the left and then retrieved the right list.

Modification Operations

To insert an element at the cursor position, it suffices to place that element at the head of one of the two lists. If we insert it at the head of the left list, the cursor will be to the right of the new element (as in case of a text editor).

```
let insert z x =
  { z with left = x :: z.left }
```

If we wish instead to leave the cursor to the left of the inserted element, it suffices to replace `left` by `right` in the code above.

Another modification operation consists in removing an element. For example, we may remove the element to the left of the cursor, if such an element exists. (This is what the *backspace* key does in a text editor.) This operation is equivalent to removing the head of the `left` list.

Program 73 — Structure of a zipper on a list.

```
type 'a zipper = { left: 'a list; right: 'a list; }

let of_list l =
  { left = []; right = l }

let move_right z = match z.right with
  | [] -> invalid_arg "move_right"
  | x :: r -> { left = x :: z.left; right = r }

let move_left z = match z.left with
  | [] -> invalid_arg "move_left"
  | x :: l -> { left = l; right = x :: z.right }

let to_list z =
  List.rev_append z.left z.right

let insert z x =
  { z with left = x :: z.left }

let delete_left z =  match z.left with
  | [] -> invalid_arg "delete_left"
  | _ :: l -> { z with left = l }

let delete_right z =  match z.right with
  | [] -> invalid_arg "delete_right"
  | _ :: r -> { z with right = r }
```

```
let delete_left z = match z.left with
  | [] -> invalid_arg "delete_left"
  | _ :: l -> { z with left = l }
```

We can similarly write an operation `delete_right` that removes the element immediately to the right of the cursor.

9.2 Zippers on Trees

The technique of the zipper also applies to other data structures, such as trees. Consider, for example, the case of polymorphic binary trees defined by the following type:

```
type 'a tree = E | N of 'a tree * 'a * 'a tree
```

What is the analog of the cursor in case of a tree? It is the designation of a particular node together with the possibility of navigating within the tree, that is, moving up, or moving down along the left or right subtrees.

Consider the tree in figure 9.4, in which we place the cursor on the node containing `5`. The zipper contains the position of the node in question and the subtree at that position. The former is represented by the path from the root to the node, that is, the sequence of moves indicating at each step whether we descend along the left or the right subtree. For example, the path to reach node `5` is the sequence `[`*left*`;` *right*`]`.

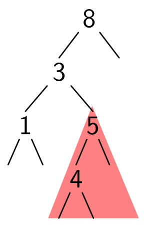


Figure 9.4: A cursor on a tree.

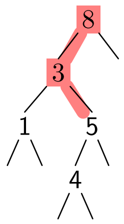
We introduce the following type to represent such paths. The constructor `Top` represents the empty path, that is, the position of the root of the tree. The constructors `Left` and `Right` indicate, respectively, a move along the left or right subtrees.

```
type 'a path =
  | Top
  | Left of 'a path * 'a * 'a tree
  | Right of 'a tree * 'a * 'a path
```

For each move, we keep track of the subtree along which we did not descend. This is necessary in order to be able to move back up the tree. The zipper on binary trees is then defined by the following type:

```
type 'a zipper = { path : 'a path; tree : 'a tree }
```

The field `tree` contains the subtree at the cursor. The field `path` contains the path leading to this subtree. Just as in case of the zipper on lists, where we chose to represent the left list in reverse order, we choose here to store the path of the node *towards* the root, rather than *from* the root. Figure 9.5 gives the zipper for the preceding example.



```
{ path = Right (N (E, 1, E), 3, Left (Top, 8, E));
  tree = N (N (E, 4, E), 5, E) }
```

Figure 9.5: A zipper on a tree.

The first constructor of the path is `Right` because the cursor designates a right subtree. The next step of the path is `Left` because node 3 is the root of a left subtree. Finally, the path reaches the root of the tree, 8.

The zipper with the cursor at the root of a tree is constructed using the empty path.

```
let of_tree t = { path = Top; tree = t }
```

Let us now write the navigation functions. We begin with a function `down_left` to move down the left subtree. For this, we examine the form of the tree at the cursor. We fail if it is empty.

```
let down_left z = match z.tree with
  | E -> invalid_arg "down_left"
```

Otherwise, we return the zipper where the subtree at the cursor is the left subtree, and the path is extended with the constructor `Left` to signify that we have moved down the left.

```
  | N (l, x, r) -> { path = Left (z.path, x, r); tree = l }
```

The node `x` and its right subtree `r` are kept in the path, as arguments of the constructor `Left`. We similarly write a function `down_right` that moves down the right subtree.

The function to move up the tree examines the current path. We fail if the path is empty.

```
let up z = match z.path with
  | Top ->
      invalid_arg "up"
```

Otherwise, we must reconstruct the node of the tree situated above the cursor. For this, we use the information contained in the path as well as the subtree contained in the field `tree`. If we move down the left subtree, the path would be of the form `Left (p, x, r)`, and the reconstructed node would have `z.tree` as its left subtree, `x` as its root, and `r` as its right subtree.

```
  | Left (p, x, r) ->
      { path = p; tree = N (z.tree, x, r) }
```

If we move down the right subtree, we proceed symmetrically.

```
  | Right (l, x, p) ->
      { path = p; tree = N (l, x, z.tree) }
```

The function that reconstructs the entire tree from a given zipper moves the cursor up the tree until it reaches the top. We stop when the path is `Top` and return the tree contained in the field `tree`.

Program 74 — A zipper structure on a binary tree.

```
type 'a tree = E | N of 'a tree * 'a * 'a tree

type 'a path =
  | Top
  | Left of 'a path * 'a * 'a tree
  | Right of 'a tree * 'a * 'a path

type 'a zipper = { path: 'a path; tree: 'a tree }

let of_tree t = { path = Top; tree = t }

let down_left z = match z.tree with
  | E -> invalid_arg "down_left"
  | N (l, x, r) -> { path = Left (z.path, x, r); tree = l }

let down_right z = match z.tree with
  | E -> invalid_arg "down_right"
  | N (l, x, r) -> { path = Right (l, x, z.path); tree = r }

let up z = match z.path with
  | Top ->
      invalid_arg "up"
  | Left (p, x, r) ->
      { path = p; tree = N (z.tree, x, r) }
  | Right (l, x, p) ->
      { path = p; tree = N (l, x, z.tree) }

let rec to_tree z =
  if z.path = Top then z.tree else to_tree (up z)
```

```
let rec to_tree z =
  if z.path = Top then z.tree else to_tree (up z)
```

The complete code of the zipper on binary trees is given in program 74.

Note: You may wonder about the commonalities between zippers on lists and those on trees. Indeed, the zipper on lists seems to be a completely symmetric structure, its two fields `left` and `right` being of the same type. However, you may also understand it as being asymmetrical, with `left` playing the same role as `path`, and `right`, that of `tree`. Thus, we could have defined the following type to represent a position in a list:

```
type 'a path = Top | Right of 'a * 'a path
```

Since this type is clearly identical to `'a list`, we did not consider it worthwhile to define a new type. Zippers on lists and trees thus embody the very same idea.

Application: Comparing Two Binary Search Trees

In chapter 5, we presented binary search trees, which can represent sets of elements equipped with a total order. If we wish to construct a set of sets, it is therefore necessary to equip binary search trees themselves with a total order.

One solution is to construct the list of elements for the two trees in infix order and then compare the two lists, for example, lexicographically. This is nevertheless somewhat naive, since the two trees may contain many elements but may differ rapidly, in which case we would have constructed the two lists in vain.

A more efficient solution is preferable, working directly on the trees. However, this is not straightforward since two binary search trees can contain the same elements without having the same structure, as illustrated in figure 9.6. One solution consists in using the zipper structure to execute a *simultaneous* infix traversal of the two trees.

We begin by writing a function that descends all the way to the bottom left of a tree and returns the zipper corresponding to this position. This amounts to iterating the function `down_left` as long as possible.

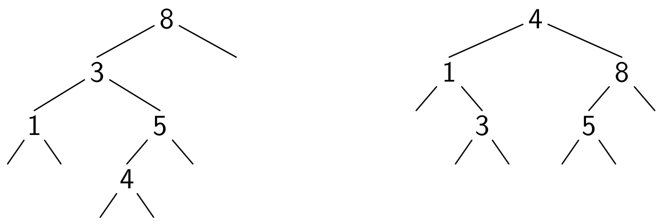


Figure 9.6: Two binary search trees containing the same elements.

Concretely, we write a function `leftmost` by passing it a zipper as accumulator. If the tree is empty, we return the zipper passed as argument:

```
let rec leftmost z = function
  | E -> z
```

Otherwise, we move down the left subtree, accumulating the visited node in the zipper.

```
  | N (l, x, r) -> leftmost (Left (z, x, r)) l
```

If we consider the two trees of figure 9.6, `t1` and `t2`, then the zippers obtained with `leftmost Top t1` and `leftmost Top t2` have the following form:

```
leftmost Top t1 = Left (Left (..., 3, ...), 1, ...)
leftmost Top t2 = Left (Left (..., 4, ...), 1, ...)
```

Figure 9.7 depicts the two zippers. We can verify that both start off with the smallest value, namely, 1.

Next, we have to write a function `compare` to perform the actual comparison. This function takes as argument two zippers and a function `cmp` to compare the elements. If the two zippers are `Top`, the comparison is over, and we return `0` to indicate equality.

```
let rec compare cmp z1 z2 = match z1, z2 with
  | Top, Top ->
      0
```

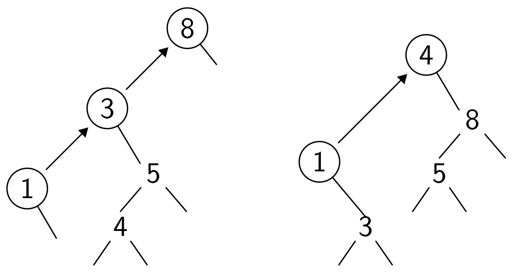


Figure 9.7: The two zippers for the trees of figure 9.6.

If the two zippers are of the form `Left`, we compare the elements that they designate using the function `cmp`. If they differ, the comparison is over, and the result given by `cmp` is returned.

```
    | Left (z1, x1, r1), Left (z2, x2, r2) ->
        let c = cmp x1 x2 in
        if c <> 0 then c
```

If they are equal, however, the comparison must continue. We look for the following elements, by calling `leftmost` on the two right subtrees, `r1` and `r2`. We then call `compare` recursively.

```
        else compare cmp (leftmost z1 r1) (leftmost z2 r2)
```

If one of the two zippers equals `Top` and the other `Left`, this means that one of the two traversals has reached the end, but not the other. We therefore return `-1` or `1`, as the case may be.

```
    | Top, Left _ ->
        -1
    | Left _, Top ->
        1
```

Finally, by construction, the case of a zipper of the form `Right` cannot occur. We dispense with it as follows:

```
    | Right _, _ | _, Right _ ->
        assert false
```

Program 75 — Lexicographic comparison of binary trees.

```
let rec leftmost z = function
  | E -> z
  | N (l, x, r) -> leftmost (Left (z, x, r)) l

let rec compare cmp z1 z2 = match z1, z2 with
  | Top, Top ->
    0
  | Left (z1, x1, r1), Left (z2, x2, r2) ->
    let c = cmp x1 x2 in
    if c <> 0 then c
    else compare cmp (leftmost z1 r1) (leftmost z2 r2)
  | Top, Left _ ->
    -1
  | Left _, Top ->
    1
  | Right _, _ | _, Right _ ->
    assert false

let compare_tree cmp t1 t2 =
  compare cmp (leftmost Top t1) (leftmost Top t2)
```

To solve the initial problem of how to compare two trees `t1` and `t2`, it suffices to construct the two zippers with the function `leftmost`, and then compare them using the function `compare`.

```
let compare_tree cmp t1 t2 =
  compare cmp (leftmost Top t1) (leftmost Top t2)
```

The complete code is given in program 75. Note that the two functions `leftmost` and `compare` are tail recursive.

Program 76 — Cursor structure.

```
type 'a enum
val start: 'a t -> 'a enum
val step: 'a enum -> 'a * 'a enum (* raise Exit when done *)
```

9.3 Cursors

When discussing the problem of comparing binary trees, we saw the importance of traversing a data structure step by step. This method of traversal is different from that of higher-order iterators like `iter` and `fold`, with which we would not have been able to solve that problem.

In general, for an arbitrary data structure of type `'a t`, we would like a *cursor* interface in order to traverse its elements one by one. Such an interface is given in program 76.

The interface contains an abstract polymorphic type, `'a enum`, for the cursor, where the variable `'a` is the type of the elements. The function `start` returns a new cursor. The function `step` returns the current element and the cursor corresponding to the next element, or raises the exception `Exit` if the iteration is complete.

This interface defines persistent cursors, which means that the cursor passed as input to the function `step` is not modified. We may thus reuse the cursor, for example, in an algorithm that uses *backtracking*. Another possibility is that of imperative cursors, where the function `step` returns only the current element and advances the cursor to the following element by a side effect (see exercise 9.12).

We will now implement persistent cursors on lists and trees.

Cursors on Lists

The zipper on lists is itself a convenient cursor. It is, nevertheless, unnecessarily complex since we do not need to keep track of the elements situated to the left

Program 77 — Cursors on lists.

```
type 'a enum = 'a list

let start l =
  l

let step = function
  | [] -> raise Exit
  | x :: r -> x, r
```

of the cursor. In other words, the cursor for lists is nothing but the right list, that is:

```
type 'a enum = 'a list
```

The function `start` reduces to the identity. The function `step` raises the exception `Exit` if the cursor is empty. Otherwise, it returns the head of the list and the rest of the list as the new cursor.

```
let step = function
  | [] -> raise Exit
  | x :: r -> x, r
```

Note that the persistent nature of lists ensures the persistence of the cursor. The complete code is given in program 77.

Cursors on Trees

To define a cursor on trees, it is first necessary to choose a traversal order. We arbitrarily choose infix order here. (See exercises 9.8 et 9.9 for other traversal orders.)

Unlike zippers on lists, zippers on trees do not immediately provide us with an operation to produce the next element in the infix traversal. However, we have seen how the zipper permits this operation when dealing with the problem

Program 78 — Cursors on trees (infix traversal).

```
type 'a enum = Top | Left of 'a * 'a tree * 'a enum

let rec leftmost t e = match t with
  | E -> e
  | N (l, x, r) -> leftmost l (Left (x, r, e))

let start t =
  leftmost t Top

let step = function
  | Top -> raise Exit
  | Left (x, r, e) -> x, leftmost r e
```

of comparing binary trees. Only the two constructors `Top` and `Left` of the zipper are necessary. We therefore define the following type for the cursor:

```
type 'a enum = Top | Left of 'a * 'a tree * 'a enum
```

The function `start` places the cursor on the element situated at the bottom left of the tree. For this, we use the function `leftmost` that was written for program 75.

```
let start t =
  leftmost t Top
```

The function `step` raises the exception `Exit` if the cursor is `Top`. Otherwise, it returns the element `x` contained in the constructor `Left` and constructs a new cursor by calling `leftmost` on the right subtree of `x`.

```
let step = function
  | Top -> raise Exit
  | Left (x, r, e) -> x, leftmost r e
```

This is exactly what we did in program 75 to move to the next element.

As in case of lists, the immutable nature of the type `enum` guarantees the persistence of the cursor. The complete code is given in program 78 (see previous page).

For Further Information

Zippers were invented by Gérard Huet and presented in the article *The Zipper* [12]. As the author noted, this concept was surely already known to other programmers.
The OCaml standard library provides a module `Seq` that implements delayed lists. This is an alternative to cursors that may be used to traverse data structures step by step.

9.4 Exercises

Zippers on Lists

9.1 Equip the zipper on lists with functions `to_start` and `to_end`, that permit moving to the beginning and end of a list, respectively.

9.2 Rewrite the function `to_list` in the same manner as the function `to_tree`.

9.3 Consider the following problem: Given a non-empty list of integers, determine if there exists a binary tree whose leaves, considered in infix order, are located at the depths given by the list. For example, the list $[1; 3; 3; 2]$ corresponds to a binary tree, but not the list $[1; 3; 3]$. We propose the following algorithm to solve this problem:

- If the list contains a single element, the algorithm terminates. The result is positive if and only if the element is 0, which corresponds to the empty tree.
- Otherwise, we search for the first two consecutive elements of the list that are equal. If there are none, we fail. Otherwise, calling n the common

value of the two integers, we replace the two occurrences of n by the integer $n-1$, which corresponds to an internal node.

- Start again with the resulting list.

Write a function `is_tree: int list -> bool` that implements this algorithm by using the zipper structure so as to avoid starting at the beginning of the list each time. The total complexity must be linear in the length of the list.

9.4 It is possible to program an efficient text editor using the zipper structure. Each line of text is a zipper on a list of characters, as explained earlier. Similarly, a file is viewed as a zipper on the list of lines. We use the following types to represent a line and a text, respectively:

```
type line = { left: char list; right: char list }
type text = { up: line list; current: line; down: line list }
```

The field `current` of the type `text` represents the current line on which the cursor is found. The fields `up` and `down` are the analogs of `left` and `right` for lines. Write the functions `insert_char`, `return`, `backspace`, etc. for such a text editor.

Zippers on Trees

9.5 Write a function `remove_leaf: 'a zipper -> 'a zipper` that verifies that the zipper passed as argument points to a node that has no children, and removes it. Such a node is of the form `N(E,x,E)`. Conversely, write a function `insert_leaf: 'a -> 'a zipper -> 'a zipper` that verifies that the zipper passed as argument points to a leaf node (that is, `E`), and which inserts in its place a new node containing the value passed as argument.

9.6 Write a function `remove_leftmost: 'a zipper -> 'a zipper` that verifies that the zipper passed as argument points to a node of the tree that has no left child, and removes it. Make sure to retain the right subtree, if it is present.

9.7 Consider n-ary trees labeled by strings, defined by the following type:

```
type tree = N of string * tree list
```

Define a zipper structure on such trees.

Cursors

9.8 Write a cursor on binary trees corresponding to a prefix traversal, that is, a traversal in which each node is visited before its two subtrees.

9.9 Write a cursor on binary trees corresponding to a postfix traversal, that is, a traversal in which each node is visited after its two subtrees.

9.10 Rewrite the comparison of binary trees (program 75) using the cursor on trees (program 78).

9.11 Rather than raising an exception `Exit` when the cursor has reached the end of the traversal, the function `step` can return a value of type `option`, that is, `step: 'a enum -> ('a * 'a enum) option`. Rewrite the cursors on lists and trees with this new interface.

9.12 If the persistence of the cursor is not used, we may consider an imperative version in which the function `step` returns the current element only and advances to the next element by a side effect, that is, `step: 'a enum -> 'a`. Rewrite the cursors on lists and trees with this new interface.

9.13 Implement a cursor on arrays.

Part III

Algorithmic Techniques and Applications

10 Arithmetic

10.1 Euclid's Algorithm

The most famous of all algorithms is certainly that of Euclid. It calculates the greatest common divisor of two integers, x and y, written $x \wedge y$, and called the "gcd of x and y". Given two non-negative integers x and y, Euclid's algorithm replaces the pair (x, y) by the pair $(y, x \bmod y)$, continuing until y reaches zero. It then returns the value of x, which is the gcd of the initial values of x and y. We write this algorithm in the form of a recursive function `gcd`:

```
let rec gcd x y =
  if y = 0 then x else gcd y (x mod y)
```

The correctness of the algorithm follows from the fact that each iteration preserves the greatest common divisor of x and y. When we reach $\mathtt{y} = 0$, we return `x`, that is, $\mathtt{x} \wedge 0$, which is then the gcd of the initial values of `x` and `y`. In other words, we have used the following two properties of the gcd:

$$\begin{array}{rcl} x \wedge y & = & y \wedge (x \bmod y), \\ x \wedge 0 & = & x. \end{array}$$

The termination of the algorithm follows from the fact that `y` decreases strictly in each iteration, remaining non-negative throughout. Indeed, we have the invariant that $\mathtt{x}, \mathtt{y} \geq 0$.

The complexity of Euclid's algorithm is given by Lamé's theorem. It states that, if the algorithm performs s iterations with $x > y > 0$, then $x \geq F_{s+1}$ and $y \geq F_s$, where (F_n) is the Fibonacci sequence. We deduce from this that Euclid's algorithm is logarithmic, that is, $s = O(\log x)$. In the general case, the complexity is $O(\log(\max(x, y)))$; see exercise 10.1. A detailed analysis may be found in *The Art of Computer Programming* [14, sec. 4.5.3].

Exercise 10.2 generalizes the function `gcd` to integers x and y that are not necessarily non-negative.

The Extended Euclidean Algorithm

We can easily modify Euclid's algorithm to compute the Bézout coefficients, that is, two integers u and v satisfying the following equation:

$$ux + vy = x \wedge y. \tag{10.1}$$

This is known as the extended Euclidean algorithm. We implement it in the form of a function `extended_gcd` that takes the integers x and y as arguments and returns the triple $(u, v, x \wedge y)$. The code is given in program 79.

It is easy to see that the third component of the returned triple is the gcd of x and y. Indeed, if we focus on this component, we find the same computations as those performed in the function `gcd` with the variables x and y, since $x - \lfloor \frac{x}{y} \rfloor y$ is nothing but $x \bmod y$. To verify the equation 10.1, it suffices to proceed by induction on y. The base case $y = 0$ is clear. For the induction step, the induction hypothesis tells us that:

$$uy + v(x - \lfloor \frac{x}{y} \rfloor y) = x \wedge y$$

This is equivalent to:

$$vx + (u - \lfloor \frac{x}{y} \rfloor v)y = x \wedge y,$$

which corresponds to the code of the function `extended_gcd`.

Program 79 — The extended Euclidean algorithm.

```
let rec extended_gcd x y =
  if y = 0 then
    (1, 0, x)
  else
    let q = x / y in
    let (u, v, g) = extended_gcd y (x - q * y) in
    (v, u - q * v, g)
```

The complexity of `extended_gcd` is the same as that of the function `gcd`. The number of operations performed in each iteration is certainly greater, but remains bounded, and the number of iterations is exactly the same. Therefore, the complexity is still $O(\log x)$.

10.2 Exponentiation by Squaring

For $n \in \mathbb{N}$, a naive computation of x^n performs $n-1$ multiplications. The algorithm of exponentiation by squaring consists in computing x^n while performing only $O(\log n)$ multiplications. It makes use of the following identities:

$$x^n = \begin{cases} (x^2)^{n/2} & \text{if } n \text{ is even,} \\ x(x^2)^{(n-1)/2} & \text{if } n \text{ is odd.} \end{cases}$$

Its translation in OCaml, for x of type `int`, is easy. We may write, for example, the code given in program 80. It uses the fact that `n/2` returns $\lfloor \mathtt{n}/2 \rfloor$, irrespective of whether `n` is even or odd.

There are multiple variants. We may, for example, treat $\mathtt{n} = 1$ as a special case, but this is not particularly useful. (See also exercise 10.4.) In any case, the central idea remains the following: If we divide the exponent by at least two in each step, the total number of recursive calls is proportionate to $\log(n)$, and so is the number of multiplications.

Program 80 — Exponentiation by squaring.

```
let rec exp x n =
  if n = 0 then
    1
  else
    let r = exp (x * x) (n / 2) in
    if n mod 2 = 0 then r else x * r
```

More precisely, we can show by induction on k that, if $2^{k-1} \leq n < 2^k$, then the function `exp` performs exactly k recursive calls. As each recursive call to `exp` performs one or two multiplications, we deduce that the total number of multiplications is bound by $2 \log n$.

The applications of this algorithm are innumerable, especially since x does not necessarily have to be of type `int`. Once we have a unit and an associative operation, that is, a monoid M, we may apply this algorithm to compute x^n for $x \in M$ and $n \in \mathbb{N}$. Exercise 10.6 proposes one such generalization.

10.3 Modular Arithmetic

In this section, we assume that we are looking to perform arithmetic modulo m for a fixed value of m. This is interesting for several reasons. We may, for instance, wish to compute a very large value but be interested only in its final k digits in base 10. In this case, we will have $m = 10^k$. We could also use the Chinese Remainder Theorem to compute a given value modulo several pairwise relative prime integers and represent in this way integers potentially larger than those provided by the machine. See exercise 10.8.

In the rest of this section, we assume `m` is a constant of type `int` such that $\mathtt{m} > 0$. We will write a certain number of functions that manipulate integers modulo `m`, that is, integers x such that:

$$0 \leq x < \mathtt{m}. \tag{10.2}$$

Exercise 10.7 proposes to gather together the various functions in a functor parametrized by m. We begin with a function of_int that takes an arbitrary integer x and returns its residue modulo m. We cannot simply do x mod m because if x is negative, then x mod m is also negative. In general, a mod b has the same sign as a. We therefore write:

```
let of_int x =
  let r = x mod m in if r < 0 then r + m else r
```

Addition modulo m is easily written if we assume that the operands are already integers modulo m. To guarantee the property (10.2), it suffices to subtract m if the value of x+y is greater than or equal to m.

```
let add x y =
  let r = x + y in if r >= m then r - m else r
```

The integers x and y being non-negative, the sign is not a problem here. Nevertheless, for the addition to be correct, the operation x+y must not overflow, that is, we must have x+y $\leq$ max_int. An easy way of guaranteeing this is to impose a maximum value on m, namely, m $\leq$ max_int$/2+1$. We can do this by placing the following assertion at the beginning of the code:

```
let () = assert (0 < m && m <= max_int/2 + 1)
```

Subtraction is written just as easily, if we take care to return a non-negative value, as we did above for the function of_int.

```
let sub x y =
  let r = x - y in if r < 0 then r + m else r
```

Arithmetic overflow is not possible here, thanks to the assumption that we made earlier for the addition operation. (It is, nevertheless, worth convincing yourself of this.) The code of these first three operations is given in program 81.

Multiplication

Multiplication is a subtler affair. For operands that are sufficiently small, that is, less than $\sqrt{\texttt{m}}$, it suffices to perform the multiplication and take its residue:

Program 81 — Arithmetic modulo m (addition and subtraction).

```
let () = assert (0 < m && m <= max_int/2 + 1)

let of_int x = let r = x mod m in if r < 0 then r + m else r

let add x y = let r = x + y in if r >= m then r - m else r

let sub x y = let r = x - y in if r < 0 then r + m else r
```

```
let mul x y =
  (x * y) mod m
```

However, if we do not wish to restrict the range of the operands, and keep only the restriction already imposed, namely, m $\leq$ max_int$/2 + 1$, we cannot use ordinary multiplication as it could trigger an overflow. We can, however, work around this difficulty by performing the multiplication as one would by hand. Suppose x is written in base 2 as:

$$(x_k x_{k-1} \dots x_1 x_0)_2$$

where x_k is the most significant digit and x_0, the least significant digit. We will then write a loop that calculates $(x_k \dots x_i)_2 \times$ y (mod m), for i varying from $k+1$ to 0. We write it as a for loop, accumulating the result in a reference r. Given the assumption on m and the fact that the OCaml type int is represented with Sys.word_size-1 signed bits (see chapter 3), we can begin with $i =$ Sys.word_size-4.

```
let mul x y =
  let r = ref 0 in
  for i = Sys.word_size - 4 downto 0 do
```

In each iteration of the loop, we begin by multiplying r by 2, modulo m, which we do with the function add.

```
    r := add !r !r;
```

Program 82 — Arithmetic modulo m (multiplication and division).

```
let mul x y =
  let r = ref 0 in
  for i = Sys.word_size - 4 downto 0 do
    r := add !r !r;
    if x land (1 lsl i) <> 0 then r := add !r y
  done;
  !r

let div x y =
  let u, _, g = extended_gcd y m in
  if g <> 1 then invalid_arg "div";
  mul x (of_int u)
```

We then have $\texttt{!r} \equiv (x_k \dots x_i)_2 \times 2 \times \texttt{y} \pmod{\texttt{m}}$. Next, we test the bit x_i and, if it equals 1, we add `y` to `r` to re-establish the loop invariant.

```
    if x land (1 lsl i) <> 0 then r := add !r y
```

Once out of the loop, we have $\texttt{!r} \equiv x_k x_{k-1} \dots x_1 x_0 \times \texttt{y} \pmod{\texttt{m}}$, that is, $\texttt{!r} \equiv \texttt{x} \times \texttt{y} \pmod{\texttt{m}}$, which is the desired result. The code is given in program 82.

Division

The division of `x` by `y` modulo `m` assumes that `y` and `m` are relatively prime, and returns a result `q` such that $\texttt{q} \times \texttt{y} = \texttt{x} \pmod{\texttt{m}}$. We use the extended Euclidean algorithm (program 79). Indeed, if `y` and `m` are relative prime integers, then the extended Euclidean algorithm gives us two integers, `u` and `v`, such that:

$$\texttt{u} \times \texttt{y} + \texttt{v} \times \texttt{m} = 1$$

This follows from equation 10.1. Multiplying this equality by `x`, we get:

$$(\texttt{x} \times \texttt{u}) \times \texttt{y} = \texttt{x} \pmod{\texttt{m}}$$

The result of the division is therefore $\texttt{x} \times \texttt{u}$. The code is given in program 82. It uses `of_int u` because `u` may be negative, but could be made more efficient since Euclid's algorithm guarantees that $|\texttt{u}| < \texttt{m}$, so that the use of `of_int` is unnecessarily complicated.

10.4 Matrix Calculus

In this section, we implement some basic matrix operations. For the sake of simplicity, we assume that the matrices have integer coefficients. Exercise 10.9 proposes a generalization to matrices with arbitrary coefficients.

As we explained in section *2.5 Sieve of Eratosthenes*, a matrix is simply an array of arrays, that is:

```
type matrix = int array array
```

We recall that we must not write `Array.make 3 (Array.make 4 v)` to create a 3×4 matrix, but rather `Array.make_matrix 3 4 v`, or equivalently, `Array.init 3 (fun _ -> Array.make 4 v)`.

Other than `Array.make_matrix`, the OCaml standard library does not provide operations on matrices. Let us begin therefore by writing a function `init_matrix`, analog of `Array.init` for matrices. The function takes as arguments the dimensions `n` and `m` of the matrix, and a function `f` to initialize each element.

```
let init_matrix n m f =
  Array.init n (fun i -> Array.init m (fun j -> f i j))
```

In other words, we return the matrix M of size $\texttt{n} \times \texttt{m}$ such that $M_{i,j} = \texttt{f}\ i\ j$. We derive from this a function `id` to construct the identity matrix of size $\texttt{n} \times \texttt{n}$.

```
let id n =
  init_matrix n n (fun i j -> if i = j then 1 else 0)
```

Next, let us write a function `size` that returns the dimensions of a matrix, that is, the number of rows and columns, as a pair. Assuming that a matrix always contains at least one row, we may write:

Program 83 — Basic matrix operations.

```
type matrix = int array array

let init_matrix n m f =
  Array.init n (fun i -> Array.init m (fun j -> f i j))

let id n =
  init_matrix n n (fun i j -> if i = j then 1 else 0)

let size a =
  (Array.length a, Array.length a.(0))

let add a b =
  let (n, m) as s = size a in
  if size b <> s then invalid_arg "add";
  init_matrix n m (fun i j -> a.(i).(j) + b.(i).(j))

let mul a b =
  let n, p = size a in
  let q, m = size b in
  if q <> p then invalid_arg "mul";
  let product i j =
    let s = ref 0 in
    for k = 0 to p - 1 do s := !s + a.(i).(k) * b.(k).(j) done;
    !s
  in
  init_matrix n m product
```

```
let size a =
  (Array.length a, Array.length a.(0))
```

Let us now consider the addition of two matrices A and B. We begin by verifying that they have the same size.

```
let add a b =
  let (n, m) as s = size a in
  if size b <> s then invalid_arg "add";
```

It then suffices to use `init_matrix` to construct the matrix whose general term is $A_{i,j} + B_{i,j}$.

```
  init_matrix n m (fun i j -> a.(i).(j) + b.(i).(j))
```

The product of a matrix A of size $n \times p$ and a matrix B of size $p \times m$ is a matrix C of size $n \times m$ whose general term is given by:

$$C_{i,j} = \sum_{k=0}^{k<p} A_{i,k} B_{k,j} \quad \text{for } 0 \leq i < n \text{ and } 0 \leq j < m. \tag{10.3}$$

We begin by determining the dimensions of the two matrices and by verifying that the number of columns of A is indeed equal to the number of rows of B.

```
let mul a b =
  let n, p = size a in
  let q, m = size b in
  if q <> p then invalid_arg "mul";
```

We then write a local function `product` that computes the coefficient $C_{i,j}$ following equation (10.3).

```
  let product i j =
    let s = ref 0 in
    for k = 0 to p - 1 do s := !s + a.(i).(k) * b.(k).(j) done;
    !s
```

It only remains to create a matrix of size $n \times m$ with the function `init_matrix` by passing it the function `product` as argument.

```
    in
    init_matrix n m product
```

The complete code is given aboev in program 83. An array of arrays is not the only possibility for representing a matrix. There are other representations that are more economical in terms of memory when a large number of elements are equal. We then speak of *sparse* matrices. Exercise 10.13 proposes one such representation.

10.5 Exercises

Euclid's Algorithm

10.1 The complexity result given in section 10.1 assumes $x > y$. Show that in the general case the complexity is $O(\log(\max(x, y)))$.

10.2 Euclid's algorithm as presented above assumes $x, y \geq 0$. If x or y is negative, the algorithm could return a negative result. (The operation `mod` returns a value of the same sign as its first argument.) Write a second version of the function `gcd` that always returns a non-negative result irrespective of the sign of its arguments. In which case will the result be zero?

10.3 Let x, y, m be three positive integers such that $y \wedge m = 1$. The quotient of x by y modulo m is any integer w such that:

$$0 \leq w < m \text{ and } x \equiv yw \pmod{m}.$$

Write a function to compute the quotient of x by y modulo m.

Exponentiation by Squaring

10.4 Write a variant of the function `exp` based on the following identities:

$$x^n = \begin{cases} (x^{n/2})^2 & \text{if } n \text{ is even,} \\ x(x^{(n-1)/2})^2 & \text{if } n \text{ is odd.} \end{cases}$$

Is there a difference in terms of efficiency?

10.5 Write a tail-recursive version of the function `exp`. This is not very interesting if n is a machine integer since $\log(n)$ is then very small compared to the maximum number of nested calls supported by the stack. However, if n is very large, for example $n = 2^{2^{20}}$, tail recursion is necessary.

10.6 Write the code of program 80 as a functor computing x^n for x of an arbitrary type `t`, provided that this type `t` is equipped with a constant `one` and an operation `mul`.

Modular Arithmetic

10.7 Write the code of programs 81 and 82 as a functor parametrized by the value of `m`, that is:

```
module Modulo(M: sig val m: int end): sig ... end = struct ... end
```

Why is it useful to write such a functor? Is it useful to have this functor export an *abstract* type of integers modulo m?

10.8 If m_1 and m_2 are two relatively prime integers, the Chinese Remainder Theorem allows us to represent every integer x between 0 (inclusive) and m_1m_2 (exclusive) by a pair (x_1, x_2), where $x \equiv x_1 \pmod{m_1}$ and $x \equiv x_2 \pmod{m_2}$. This representation is useful when m_1 and m_2 are representable in the machine but m_1m_2 is not (being too large). Write the addition, subtraction, and multiplication operations on this representation of integers. Given the pair (x_1, x_2), how can we recover the integer x that it represents?

More generally, we may consider k integers $m_1, m_2, \dots, m_k$ pairwise relatively prime and represent arbitrary integers by k-tuples. For more details, consult *The Art of Computer Programming* (vol. 2, sec. 4.3.2).

Matrix Calculus

10.9 Write the code of figure 83 as a functor parametrized by the type of matrix elements, constants `zero` and `one` of this type, and operations `add` and `mul` on this type.

10.10 Write a function `power: t -> int -> t` that raises a matrix to power n using the algorithm of exponentiation by squaring.

10.11 The numbers of the Fibonacci sequence (F_n) satisfy the following identity:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}.$$

In other words, we may compute F_n by raising a 2×2 matrix to power n. Derive a program from the preceding exercise that computes F_n in $O(\log n)$ elementary arithmetic operations.

Note, however, that this does not mean that that we are able to compute F_n for large values of n. In fact, the terms of the Fibonacci sequence grow exponentially. If we make use of arbitrary-precision integers, the cost of the arithmetic operations themselves must be taken into account, and the complexity will not be $O(\log n)$. Otherwise, the calculation will quickly overflow.

10.12 Combining the preceding exercise and the modular arithmetic of the preceding section, write a program that computes the final seven digits of F_{10^6}. The computation should be instantaneous.

10.13 When a matrix contains many identical elements (for example, many zero elements), it can be interesting to represent it more compactly than as an array of arrays. One solution consists in representing each row of the matrix by a dictionary that associates some of the column indices with their corresponding elements, the others taking a default value. For example, for matrices with integer coefficients whose zero elements are not represented, we may define the following type:

```
type matrix = { cols: int; rows: M.t array }
```

where `M.t` is the type of a dictionary associating integers with integers (for example, obtained using the module `Map`), and where the field `cols` stores the number of columns. Write the functions `id`, `size`, `add`, and `mul` for this type.

11 Dynamic Programming and Memoization

Dynamic programming and memoization are two closely related techniques based on the idea that it is better to avoid computing the same thing twice.

11.1 Basic Idea

We illustrate the two techniques in the simple example of the Fibonacci sequence. Recall that this sequence of integers (F_n) is defined by:

$$\begin{cases} F_0 & = & 0 \\ F_1 & = & 1 \\ F_n & = & F_{n-2} + F_{n-1} \text{ if } n \geq 2. \end{cases}$$

It is easy enough to write a recursive function that calculates F_n based on this definition:

```
let rec fib n =
  if n <= 1 then n else fib (n - 2) + fib (n - 1)
```

But this is also rather simplistic. Indeed, it would take more than a minute to compute F_{50}. It is not the recursion that is problematic here, but rather the fact that we compute the same values of F_n several times over. For instance, to compute F_5, F_3 will be computed twice and F_2 three times, as shown in the call graph in figure 11.1.

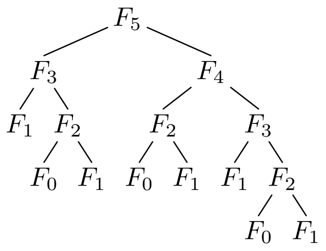


Figure 11.1: Call graph of F_5.

It is easy to see that, in general, the computation of F_n requires $F_{n+1} - 1$ additions because the recursion equation is exactly the same as that which defines F_n. We can observe this empirically. On a computer circa 2011, it takes 2 seconds to compute F_{42}, 3 seconds to compute F_{43}, 5 seconds to compute F_{44}, etc. We recognize here the terms of the Fibonacci sequence. Extrapolating, we can say that it would take 89 seconds to compute F_{50}, and this is correct to within a half-second! As we know that F_n grows exponentially, we cannot hope to go much further in the computation of the sequence using this method.

11.1.1 Memoization

The fact that we are computing the same thing several times leads us, quite naturally, to the idea that we could store the intermediate results in a table. Such a table would associate integers n with the value F_n. Accordingly, we proceed as follows: To compute `fib n`, we begin by checking whether there is an entry in the table for `n`, in which case we return the corresponding value. Otherwise, we calculate `fib n` as `fib (n-2) + fib (n-1)`, that is, recursively. We return the result after having added it to the table. This technique, consisting in using a table to store intermediate results, is called *memoization*, a term coined by the researcher Donald Michie in 1968.

Let us implement this idea in a function `fib_memo`. We begin by introducing a hash table to store the intermediate results, choosing an arbitrary initial size for it:

```
let memo = Hashtbl.create 17
```

The function `fib_memo` begins by checking whether the value of `fib n` is already present in the table `memo`. If it is, it is returned:

```
let rec fib_memo n =
  try
    Hashtbl.find memo n
```

Otherwise, we compute the result exactly as for the function `fib`, that is, using two recursive calls:

```
  with Not_found ->
    let fn = if n <= 1 then n
             else fib_memo (n-2) + fib_memo (n-1) in
```

Then, we store it in the table `memo` before returning it:

```
    Hashtbl.add memo n fn;
    fn
```

This concludes the function `fib_memo`. Its efficiency is far greater than that of `fib`. The computation of F_{50}, for example, becomes instantaneous (in contrast to the 89 seconds taken by `fib`). We can show that the complexity of `fib_memo` is linear. Intuitively, we understand that computing F_n now involves computing F_i only once for $i \leq n$. The complete code of the function `fib_memo` is given in program 84 below. Note that the table `memo` is defined *outside* the function `fib_memo`, since it must be the same for all recursive calls.

11.1.2 Dynamic Programming

Using a hash table to memoize the computation of the Fibonacci sequence may seem unnecessarily costly. Indeed, in the end we will store only the values of

Program 84 — Computing F_n via memoization.

```
let memo = Hashtbl.create 17
let rec fib_memo n =
  try
    Hashtbl.find memo n
  with Not_found ->
    let fn =
      if n <= 1 then n else fib_memo (n-2) + fib_memo (n-1) in
    Hashtbl.add memo n fn;
    fn
```

F_i for $i \leq n$, for which an array of size $n+1$ would suffice. If we wished to rewrite the function `fib_memo` using this idea, we could, for example, begin by assigning the value -1 to each cell of the array, to indicate that the value has not yet been computed. Another possibility would be to fill the array in a specific order. In the present case, we see that it suffices to fill the array in increasing order since the computation of F_i requires the values of F_{i-2} and F_{i-1}. This technique, consisting in using a table and progressively filling it with the results of intermediate computations, is called *dynamic programming*, often abbreviated as DP.

Let us implement this idea in a function `fib_dp`. We begin by treating the special case $n = 0$. In the general case, when $n > 0$, we allocate an array `f` of size $\mathtt{n}+1$, which will contain the values of the F_i.

```
let fib_dp n =
  if n = 0 then 0 else
  let f = Array.make (n+1) 0 in
```

This array may be allocated *inside* the function `fib_dp` since, unlike `fib_memo`, it will not be recursive. Next, we fill the cells of the array `f` in increasing order, treating `f.(1)` as a special case.

```
  f.(1) <- 1;
```

Program 85 — Computing F_n via dynamic programming.

```
let fib_dp n =
  if n = 0 then 0 else
  let f = Array.make (n+1) 0 in
  f.(1) <- 1;
  for i = 2 to n do f.(i) <- f.(i-2) + f.(i-1) done;
  f.(n)
```

```
for i = 2 to n do f.(i) <- f.(i-2) + f.(i-1) done;
```

Once the array is filled, we have only to return the value contained in its last cell.

```
f.(n)
```

This concludes the function `fib_dp`. As for `fib_memo`, its complexity is linear. This can be seen easily since the code boils down to a simple `for` loop. The complete code of the function `fib_dp` is given in program 85.

Remark

We have chosen the Fibonacci sequence as an example for pedagogical reasons. Of course, it is very simple to compute the terms of this sequence in linear time without using either memoization or dynamic programming (see exercise 11.1).

11.2 Mechanical Memoization

The use of an array in `fib_dp`, rather than the hash table in `fib_memo`, may leave us thinking that dynamic programming is simpler to implement than memoization. This is indeed the case in that example.

It is nevertheless important to understand that in order to write `fib_dp` we made use of two facts: we had to compute the F_i for all $i \leq n$; and we could compute them in increasing order. In general, the inputs of the function that we wish to compute may not necessarily be consecutive integers. They may not be integers at all, or the dependencies between the different values may not be as simple.

Memoization, by contrast, is simpler to implement. It suffices to add a few lines of code to consult and fill the hash table, without modifying the structure of the function. We can do this mechanically.

Better yet, we can implement the memoization procedure *once and for all*, as a higher-order function `memo_fun` that takes as argument a function `f` to be memoized. We begin by creating a hash table in which to store the already computed results.

```
let memo_fun f =
  let h = Hashtbl.create 17 in
```

We then return a function that performs the memoization, that is, consults the table and fills it as needed.

```
  fun x ->
    try Hashtbl.find h x
    with Not_found -> let v = f x in Hashtbl.add h x v; v
```

The type of the function `memo_fun` is as follows:

```
memo_fun : ('a -> 'b) -> 'a -> 'b
```

As we see, `memo_fun` takes as argument a function of an arbitrary type and returns a function of the same type. The polymorphic nature of the function follows from the use of the generic hash tables of OCaml (see exercise 11.2).

In order to memoize the function `fib`, it suffices to write:

```
let fib_memo2 = memo_fun fib
```

If we compute `fib_memo2 50` twice, the second computation is instantaneous because the result is present in the table. However, the first computation takes just as long as it does without memoization. Indeed, during the first call, we calculate `fib 50` because the value is not present in the table. During this computation, recursive calls are made to the function `fib` and not to its memoized version `fib_memo2`.

We might think that a solution consists in writing the following code:

```
let rec fib_memo3 =
  memo_fun (fun n -> if n <= 1 then n
                     else fib_memo3 (n-2) + fib_memo3 (n-1))
```

Unfortunately, this still does not work. Each call to `fib_memo3` involves a new call to `memo_fun`, which has the effect of allocating a new hash table. Thus the previously calculated results are lost and cannot be found.

To solve this problem, that is, to allocate a single hash table and perform recursive calls to the function being memoized, it is necessary to rewrite the function `memo_fun` in a subtler manner. We do this in the form of a function `memo_rec` that may be used as follows:

```
let fib_memo4 =
  memo_rec (fun f n -> if n <= 1 then n else f (n-2) + f (n-1))
```

Here, the single call to `memo_rec` ensures the creation of a single table. The function being memoized is passed as an additional argument `f`.

The function `memo_rec` is written as follows. We begin, as in case of `memo_fun`, by creating a table:

```
let memo_rec ff =
  let h = Hashtbl.create 5003 in
```

Next, we define the *recursive* function `f` that performs the memoization:

```
  let rec f x =
    try Hashtbl.find h x
    with Not_found -> let v = ff f x in Hashtbl.add h x v; v
  in
  f
```

When the result is not in the table, we compute the result `v` using `ff f x`, that is, by passing the function `f` and the argument `x` to the function `ff` provided by the user. The complete code of `memo_rec` is given in program 86.

Program 86 — Generic memoization operator.

```
let memo_rec ff =
  let h = Hashtbl.create 5003 in
  let rec f x =
    try Hashtbl.find h x
    with Not_found -> let v = ff f x in Hashtbl.add h x v; v
  in
  f
```

As in case of `memo_fun`, the type of `memo_rec` is polymorphic, but it is more complex:

```
memo_rec:  (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b
```

The function `ff` has type `('a -> 'b) -> 'a -> 'b`, which may be read as `('a -> 'b) -> ('a -> 'b)`. That is, it takes as argument a function of type `'a -> 'b` and returns a function of the same type.

11.3 Differences between Memoization and Dynamic Programming

In certain situations, dynamic programming may be preferred to memoization. Take the example of the computation of the binomial coefficients $C(n, k)$, the recursive definition of which is as follows:

$$\begin{cases} C(n,0) & = & 1 \\ C(n,n) & = & 1 \\ C(n,k) & = & C(n-1,k-1)+C(n-1,k) \text{ if } 0<k<n. \end{cases}$$

Both memoization and dynamic programming may be applied to this definition. In the first case, we will have a hash table indexed by the pair (n, k) and, in the second case, a matrix indexed by n and k. However, if we seek to compute

$C(n,k)$ for $n = 2 \times 10^5$ and $k = 10^5$, we are probably going to exceed the available memory of the machine (assuming an ordinary desktop computer). In the first case, this would happen when filling the hash table and in the second, when trying to allocate the matrix. The reason is that the complexity in time *and in space* is $O(nk)$. In the previous example, we would have to store at least 15 billion intermediate results.

Yet, if we look closer, the computation of $C(n,k)$ for a given value of n only requires the values of $C(n-1,k)$. Accordingly, we may carry out the computation for increasing values of n, even if not all the values computed are going to be useful in the end. We can visualize the situation by drawing Pascal's triangle as in figure 11.2. We will compute the values line by line, keeping track of only a single line at any given moment.

$$\begin{array}{cccccc}
1 \\
1 & 1 \\
1 & 2 & 1 \\
1 & 3 & 3 & 1 \\
1 & 4 & 6 & 4 & 1 \\
1 & 5 & 10 & 10 & 5 & 1 \\
\vdots
\end{array}$$

Figure 11.2: Pascal's triangle.

Let us implement this idea in a function `comb_smart_dp`. We begin by allocating an array `row` of size `n+1` that will contain the line of Pascal's triangle being computed:

```
let comb_smart_dp n k =
  let row = Array.make (n+1) 0 in
```

In fact, an array of size `k+1` would suffice (see exercise 11.3).

Initially, this array is filled with zeroes. We initialize the first cell with 1.

```
  row.(0) <- 1;
```

This value will not change in what follows, because the first column of Pascal's triangle contains only ones. Next, we write a loop to compute the `i`-th line of Pascal's triangle:

```
  for i = 1 to n do
```

This computation will be done in-place within the array `row`, assuming that it contains the values of the (`i` − 1)-th line. To avoid overwriting values that have yet to be used, we will compute the new values from right to left, since these values do not depend on those situated further to the right. We continue with a second loop:

```
    for j = i downto 1 do row.(j) <- row.(j) + row.(j-1) done
```

Here, we make use of the fact that `row.(i)` contains 0, which permits us to avoid introducing a special case. We stop when `j` = 1 because the value `row.(0)` does not need to be recomputed, as explained above. Once we have exited the double loop, the array `row` contains the `n`-th line of Pascal's triangle, and we only have to return `row.(k)`:

```
  done;
  row.(k)
```

The time complexity remains $O(nk)$ but the space complexity is only $O(n)$. In the example given above, with $n = 2 \times 10^5$ and $k = 10^5$, the result is now obtained in a few seconds. Note, however, that it causes an arithmetic overflow (see exercise 11.4).

The conclusion of this little exercise is that dynamic programming, which allows for a finer control of the use of memory, can sometimes be more advantageous than memoization. Conversely, in many situations where this extra measure of control is not necessary, memoization is far simpler to implement.

11.4 Hash-consing

Just as we can avoid computing the same thing twice, as we did above, we may similarly avoid constructing the same data several times, to save on memory. We can do this safely as long as we are dealing with values that will not be subsequently modified. If, for example, we have already constructed the list `[1; 2; 3]`, it is pointless to reconstruct anew. We can instead reuse it.

The technique of reusing values that have already been constructed is known as *hash-consing*. It is quite simply the application of the technique of memoization to the construction of values. The term hash-consing combines precisely the idea of hashing used in memoization with that of allocation[1]. Here, we will demonstrate how to implement this technique, taking as an example binary trees containing characters, that is:

```
type tree = E | N of tree * char * tree
```

The idea is to provide an alternative to the constructor `N`, in the form of a function `node` that behaves exactly as this constructor, but that returns a structurally identical value, if one has already been constructed. One possible solution consists in directly memoizing the function `node`, that is, by storing the values of type `tree` that have already been constructed in a hash table. However, proceeding in this manner would not be very efficient. Indeed, both computing the hash keys and comparing values of the type `tree` require traversals that may be costly.

A much better solution consists in avoiding these traversals by associating an integer, to be used in the hash function, with each value of type `tree`. We begin, therefore, by modifying the type `tree` slightly in order to store an integer as the first argument of the constructor `N`.

```
type tree = E | N of int * tree * char * tree
```

We will see below how this integer is computed. We write a function that returns this integer when it is given the constructor `N`, and the integer 0 otherwise.

```
let unique = function
  | E -> 0
  | N (u, _, _, _) -> u
```

[1]The term *cons* stands for allocation in the language Lisp, where the technique of hash-consing was first introduced.

We will justify the name of this function below, by explaining why this integer is in fact unique. For the present, this is not important since we wish only to define a hash function. Here is the definition of the hash function:

```
let hash = function
  | E ->
      0
  | N (_, l, c, r) ->
      (19 * (19 * unique l + Char.code c) + unique r) land max_int
```

This function does not make use of the integer stored in the constructor N. Indeed, hash will be used precisely to determine if a tree with root c and subtrees l and r has already been constructed. It must therefore only depend on the values l, c, and r. To compute the result in constant time, the function hash uses the integers stored in the subtrees l and r, via the function unique. The operation land max_int is used here to guarantee a non-negative result.

The second idea consists in comparing two values of type tree in constant time by using the fact that their subtrees have already been shared if they are equal and are therefore physically equal. For this we write a function equal that uses the physical equality == on the subtrees.

```
let equal t1 t2 = match t1, t2 with
  | E, E -> true
  | N (_, l1, c1, r1), N (_, l2, c2, r2) ->
      l1 == l2 && c1 == c2 && r1 == r2
  | _ -> false
```

Equipped with these two functions, we can now construct a hash table to memoize the construction of trees. We can do this using the functor Hashtbl.Make of the OCaml standard library. However, such a table would keep the constructed values forever. What we want, by constrast, is for the GC to be able to recover the space allocated for values that are no longer used, as it usually does.

For this, we will use another hash table structure included in the OCaml standard library, namely, the functor Weak.Make. Its signature is identical to that of the functor Hashtbl.Make.

```
module X = struct
  type t = tree
  let hash = ...
  let equal = ...
end
module W = Weak.Make(X)
```

The resulting module `W` provides a set structure of values of type `tree`, in which the elements disappear when they are not referenced anywhere except by a pointer from this structure. Such pointers, which are not taken into account by the GC, are called *weak references*. Hence the name of the module: `Weak`.

We may now construct the table that will contain the constructed values of type `tree`. We initialize it with an arbitrary size.

```
let nodes = W.create 5003
```

It remains only to write the function `node` that will perform the actual memoization. It uses a global counter, initialized with the value 1.

```
let node =
  let cpt = ref 1 in
  fun l c r ->
```

We begin by constructing the desired tree, using the current value of the counter.

```
    let n0 = N (!cpt, l, c, r) in
```

Next, we consult the hash table to determine if such a value has already been constructed. The module `W` conveniently provides a function `merge` that determines if an equal value is already present in the set and, if this is not the case, adds it. In both cases, the returned value is contained in the table.

```
    let n = W.merge nodes n0 in
```

It only remains to increment the value of the counter if a new node has been added to the table, which is easily determined by comparing `n` and `n0` using physical comparison. In all cases, we return the tree `n`.

```
    if n == n0 then incr cpt;
    n
```

Program 87 — Hash-consing (code).

```
type tree = E | N of int * tree * char * tree

let empty =
  E

let unique = function
  | E -> 0
  | N (u, _, _, _) -> u

module X = struct
  type t = tree
  let hash = function
    | E ->
       0
    | N (_, l, c, r) ->
       (19 * (19 * unique l + Char.code c) + unique r)
       land max_int
  let equal t1 t2 = match t1, t2 with
    | E, E -> true
    | N (_, l1, c1, r1), N (_, l2, c2, r2) ->
       l1 == l2 && c1 == c2 && r1 == r2
    | _ -> false
end
module W = Weak.Make(X)
let nodes = W.create 5003

let node =
  let cpt = ref 1 in
  fun l c r ->
    let n0 = N (!cpt, l, c, r) in
    let n = W.merge nodes n0 in
    if n == n0 then incr cpt;
    n
```

Program 88 — Hash-consing (interface).

```
type tree = private E | N of int * tree * char * tree
val empty: tree
val node: tree -> char -> tree -> tree
```

The complete code is given in program 87.

Of course, all of this is correct only if two trees that have the same integer are indeed equal. To guarantee it, we must encapsulate the type `tree` behind an interface that prevents users from arbitrarily setting this integer. One possibility is to make `tree` a private type. Such an interface is given in program 88 (see following page). In this way, we can justify *a posteriori* that the integer stored in each tree is unique in the sense that two trees bearing the same integer are equal. However, two equal values can bear a different integer. Indeed, a value can be constructed, then reclaimed by the GC, then reconstructed identically later, with a different integer.

The advantages of hash-consing are numerous. Firstly, it permits us to economize on memory. Certainly, there is an extra cost linked to memoization, but this cost is of constant time. Indeed, we have been careful to write an equality and a hash function that are of constant cost. The second advantage of hash-consing is that we can now use physical equality `==` instead of structural equality. Indeed, two values of type `tree` are equal if and only if they are physically equal. This is guaranteed by the encapsulation of the type `tree` behind a private type. Thirdly, we have a total order on the type `tree`, of constant time, by comparing the values returned by the function `unique`. In particular, we can now construct data structures, for example by using the functors `Set.Make` or `Map.Make`, that will be as efficient as in case of integers. Finally, we have an excellent hash function on the type `tree`, namely, the function `unique`. Exercise 11.6 proposes using it to memoize a recursive function on trees.

It is important to reiterate here that the hash-consing technique is only applicable to structures that can be shared. The technique is thus applicable to persistent structures, but not to mutable ones.

11.5 Exercises

Dynamic Programming and Memoization

11.1 Modify the function `fib_dp` so that it uses only two integers instead of an array.

11.2 Reimplement the function `memo_rec` as a functor parametrized by the type of the arguments of the memoized function, so as to be able to use hash tables other than the generic tables of OCaml.

11.3 Modify the function `comb_smart_dp` so that it does not compute the values of Pascal's triangle beyond the `k`-th column.

11.4 Modify the function `comb_smart_dp` so that it returns an arbitrary-precision integer (type `Num.num` of the OCaml library). Note: The complexity is no longer $O(nk)$ because additions are no longer atomic operations. Their cost depends on the size of the operands, and these grow rapidly in Pascal's triangle.

11.5 The edit distance between two strings is defined as the smallest number of insertions, deletions, and substitutions of characters required to pass from one to the other. For example, the distance between `"duck"` and `"duke"` is two (one deletion, one insertion). Write a recursive function that takes as arguments two strings `a` and `b`, and two indices `i` and `j`, and returns the distance between the strings `a`[0..`i`[and `b`[0..`j`[. Explain why we end up making the same calls several times. Improve efficiency by using either memoization or dynamic programming.

Hash-consing

11.6 Write a memoized function `height` that computes the height of a tree making use of the fact that trees have been hash-consed.

11.7 Write a structure of lists of integers making use of the hash-consing technique.

11.8 Generalize the above exercise to a functor that takes as argument the type of the elements, equipped with an equality and a hash function.

11.9 How can the application of the constructor `N` in the function `node` in program 87 be avoided when the value has already been constructed?

12 Sorting Algorithms

This chapter presents several sorting algorithms on lists and arrays. We assume that the elements to be sorted are of type `elt`, equipped with a total order given in the form of a function `compare` of type `elt -> elt -> int`. Sorting functions on lists and arrays will respectively have type `elt list -> elt list` and `elt array -> unit`. We assume that we sort lists and arrays in increasing order. For the sake of convenience, we define two functions `lt` and `le` as follows:

```
let lt x y = compare x y < 0
let le x y = compare x y <= 0
```

Throughout this chapter, N will denote the number of elements to be sorted. For each sorting algorithm presented, we indicate its complexity not only in terms of the number of comparisons performed, but also in terms of the number of basic allocations in the case of lists and the number of assignments in the case of arrays. For each one, we give the complexity in the best, worst, and average cases. For the average case, we assume that the elements to be sorted are distinct and that the $N!$ possible permutations of the input are all equally likely. Recall that the optimal complexity of a sorting algorithm only performing comparisons of elements is $O(N \log N)$ (see section 12.5).

12.1 Insertion Sort

Insertion sort consists in successively inserting each element into the set of elements already sorted. This is what we do, for example, when we sort a pack of cards.

12.1.1 Lists

Insertion sort on a list consists in traversing it while successively inserting each element in the part already sorted. We begin therefore by writing a function `insert` that inserts an element `x` in a list that is assumed to be sorted. The code is as follows:

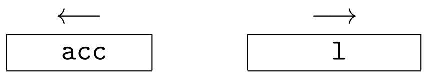
```
let rec insert x = function
  | y :: l when lt y x -> y :: insert x l
  | l -> x :: l
```

The first pattern matching case corresponds to a recursive insertion when the first element of the list is smaller than `x`. The second pattern matching case corresponds to that in which `x` is smaller than all the elements of `l`, whether `l` is empty or not.

Sorting then consists in traversing the list using `List.fold_left`, with the portion of the list already sorted as accumulator.

```
let insertion_sort l =
  List.fold_left (fun acc x -> insert x acc) [] l
```

This code may prove problematic for long lists: Since the function `insert` is not tail recursive, it can overflow the stack. It is, however, easy to modify the function `insert` to make it tail recursive. We can do this by adding an accumulator containing the list, in reverse order, of elements smaller than `x` that have already been considered. If we call this accumulator `acc`, we then have the following situation when we try to insert `x` in `l`:



Once the position of x is determined, it suffices to reverse the accumulator acc and concatenate the result with l. The function List.rev_append does exactly this and is tail recursive. The final code of insertion sort is given in program 89.

Program 89 — Insertion sort on lists.

```
let rec insert acc x = function
  | y :: l when lt y x -> insert (y :: acc) x l
  | l -> List.rev_append acc (x :: l)

let insertion_sort l =
  List.fold_left (fun r x -> insert [] x r) [] l
```

Complexity

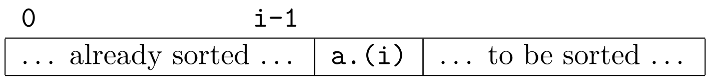
When the function insert inserts an element x at position k in l, it performs k comparisons and $2k$ allocations (k to construct the accumulator and k in List.rev_append). At best, k is always 1 and, at worst, it equals the length of the list, which in the end gives the table in figure 12.1.

	best case	average	worst case
comparisons	N	$N^2/4$	$N^2/2$
allocations	N	$N^2/2$	N^2

Figure 12.1: Complexity of insertion sort on lists.

12.1.2 Arrays

Insertion sort on an array a is performed in-place. It consists in successively inserting each element a.(i) in the portion of the array a[0..i-1] that is already sorted, which corresponds to the following situation:



We begin with a `for` loop to traverse the array:

```
let insertion_sort a =
  for i = 1 to Array.length a - 1 do
    let v = a.(i) in
```

To insert the element `a.(i)` in the right place, we use a `while` loop that shifts the elements to the right as long as they are greater than `a.(i)`.

```
    let j = ref i in
    while 0 < !j && lt v a.(!j - 1) do
      a.(!j) <- a.(!j - 1);
      decr j
    done;
```

Once out of the loop, all that remains is to put `a.(i)` in its place.

```
    a.(!j) <- v
  done
```

The complete code of insertion sort on arrays is given in program 90.

Complexity

Note that the function `insertion_sort` performs as many comparisons as assignments. When the `while` loop inserts the element `a.(i)` at position $\texttt{i} - k$, it performs $k + 1$ comparisons. At best, k equals 0 and, at worst, it equals `i`, which in the end gives the table in figure 12.2.

12.2 Quicksort

Quicksort relies on the method of *divide and conquer*: The elements to be sorted are divided into two subsets, all the elements of the first being smaller than those

Program 90 — Insertion sort on arrays.

```
let insertion_sort a =
  for i = 1 to Array.length a - 1 do
    let v = a.(i) in
    let j = ref i in
    while 0 < !j && lt v a.(!j - 1) do
      a.(!j) <- a.(!j - 1);
      decr j
    done;
    a.(!j) <- v
  done
```

	best case	average	worst case
comparisons	N	$N^2/4$	$N^2/2$
assignments	N	$N^2/4$	$N^2/2$

Figure 12.2: Complexity of insertion sort on arrays.

of the second. Then, we sort each subset recursively. In practice, the subdivision is performed with the help of an arbitrary element p of the set, called the *pivot*. The two subsets are then respectively the elements less than or equal to p, and the elements greater than p.

12.2.1 Lists

To implement quicksort on lists, we begin by writing a function `partition` that partitions a list at a given pivot `p`. To ensure tail recursion, we use an accumulator consisting of a pair of lists. The function `partition` therefore takes the following form:

```
let rec partition ((left, right) as acc) p = function
```

The lists `left` and `right` represent the elements that are, respectively, less than or equal to `p` and greater than `p`. When we reach the empty list, it suffices to return the accumulator.

```
    | [] -> acc
```

Otherwise, we consider the first element `x` and compare it with the pivot `p`. If `x` is less than or equal to `p`, we add it to `left` and call `partition` recursively on the rest of the list, `s`.

```
    | x :: s when le x p ->
        partition (x :: left, right) p s
```

On the other hand, if `x` is greater than `p`, we add it to `right` and similarly call `partition` recursively.

```
    | x :: s ->
        partition (left, x :: right) p s
```

Quicksort takes the form of a recursive function `quicksort`. The base case is that of the empty list.

```
  let rec quicksort = function
    | [] ->
        []
```

If the list is not empty, we choose its first element as pivot `p` and partition the remaining elements using the function `partition`.

```
    | p :: s ->
        let (left, right) = partition ([], []) p s in
```

Finally, we sort `left` and `right` recursively and concatenate the results, remembering to insert the pivot `p` in the middle.

```
        (quicksort left) @ (p :: quicksort right)
```

The complete code of quicksort on lists is given in program 91.

It is important to note that neither of the two recursive calls of the function `quicksort` is tail recursive. A stack overflow is therefore possible. Exercise 12.4 discusses a solution to this problem.

Program 91 — Quicksort on lists.

```
let rec partition ((left, right) as acc) p = function
  | [] -> acc
  | x :: s when le x p ->
      partition (x :: left, right) p s
  | x :: s ->
      partition (left, x :: right) p s

let rec quicksort = function
  | [] ->
      []
  | p :: s ->
      let (left, right) = partition ([], []) p s in
      (quicksort left) @ (p :: quicksort right)
```

Complexity

The function `partition` makes as many comparisons and allocations as there are elements in the list passed as argument. Let $C(N)$ be the number of comparisons performed in the function `quicksort` for a list of length N.

If `partition` returns a list of length K and a list of length $N-1-K$, we deduce:

$$C(N) = N - 1 + C(K) + C(N-1-K).$$

The worst case corresponds to $K = 0$, which gives $C(N) = N-1+C(N-1)$, and hence $C(N) \sim \frac{N^2}{2}$. The best case corresponds to a list split into two equal halves, that is, $K = N/2$. From this we easily deduce $C(N) \sim N \log N$. For the number of comparisons on average, we consider that the N possible final places for the pivot are all equally probable, which gives:

$$\begin{aligned} C(N) &= N-1+\frac{1}{N}\sum_{0\le K\le N-1} C(K)+C(N-1-K) \\ &= N-1+\frac{2}{N}\sum_{0\le K\le N-1} C(K). \end{aligned}$$

After a bit of algebra, left to the reader, we obtain:

$$\frac{C(N)}{N+1}=\frac{C(N-1)}{N}+\frac{2}{N+1}-\frac{2}{N(N+1)}.$$

This is a telescoping sum, which leads us to conclude that:

$$\frac{C(N)}{N+1}\sim 2\log N$$

We therefore have $C(N)\sim 2N\log N$. For the number of allocations $A(N)$, we proceed in a similar fashion, with a slightly different equation:

$$A(N)=N+K+A(K)+A(N-1-K).$$

In the end, we obtain the results given in the table of figure 12.3.

	best case	average	worst case
comparisons	$N\log N$	$2N\log N$	$N^2/2$
allocations	$\frac{3}{2}N\log N$	$4N\log N$	N^2

Figure 12.3: Complexity of quicksort on lists.

12.2.2 Arrays

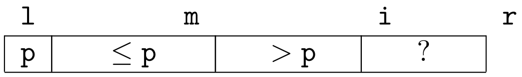
Quicksort on arrays is performed in-place. The partition and sorting functions take as arguments the array and two indices that delimit the portion to be considered. We retain the idea of a function `partition` that organizes the elements around a pivot, and a sorting function that proceeds recursively.

The function `partition` takes an array `a` as argument, and two indices `l` and `r`, that delimit a portion of the array. This portion corresponds to the elements between the indices `l` (inclusive) and `r` (exclusive). We begin by choosing `a.(l)` as pivot.

```
let partition a l r =
  let p = a.(l) in
```

This choice is arbitrary and can influence performance. Exercise 12.8 proposes a better method of choosing the pivot.

The idea consists in traversing the array from left to right, between indices `l` (inclusive) and `r` (exclusive), with a `for` loop. In each iteration of the loop, the situation is as follows:



The index `i` of the loop gives the position of the next element to consider. The index `m` partitions the part that has already been traversed. More precisely, it is the index of the last cell containing a value less than or equal to `p`. We represent it with a reference.

```
  let m = ref l in
  for i = l + 1 to r - 1 do
```

If `a.(i)` is strictly greater than `p`, we leave it in its place. Otherwise, to maintain the loop invariant, it suffices to increment `m` and swap `a.(i)` and `a.(!m)`.

```
    if le a.(i) p then begin incr m; swap a i !m end
  done;
```

The code of the function `swap` is given in program 92 below. Once all the elements have been traversed, we perform one more swap to put the pivot in its place, that is, at position `!m`, and we return this index.

```
  if l <> !m then swap a l !m;
  !m
```

We write the main function of quicksort as a recursive function quick_rec that takes the same arguments as the function partition. If l $\geq$ r-1, the number of elements to be sorted is either zero or one, and there is nothing to do.

```
let rec quick_rec a l r =
  if l < r - 1 then begin
```

Otherwise, we partition the elements between l and r.

```
    let m = partition a l r in
```

After this call, the pivot a.(m) is at its final place. We then perform two recursive calls to sort, respectively, the elements smaller than and larger than the pivot.

```
    quick_rec a l m;
    quick_rec a (m + 1) r
  end
```

To sort an array, it suffices to call quick_rec on all of its elements.

```
let quicksort a = quick_rec a 0 (Array.length a)
```

The complete code is given in program 92.

As with quicksort on lists, a stack overflow could occur. Indeed, the first call to quick_rec is not tail recursive and, in the worst case, the number of these calls may be equal to the length of the array. Exercice 12.6 discusses a solution to this problem.

Complexity

The function partition always performs exactly $(\texttt{r} - \texttt{l}) - 1$ comparisons. The complexity in number of comparisons is therefore the same as in case of quicksort on lists. As far as the number of assignments is concerned, we note that the function partition performs as many calls to swap as increments of m. The best case occurs when, at each partition stage, the pivot remains in the first position. Then no assignment is performed. It is important to note that this case does not correspond to the best complexity in terms of comparisons, which is quadratic in this case. Thus, on average, and assuming that all final positions for the

Program 92 — Quicksort on arrays.

```
let swap a i j =
  let t = a.(i) in
  a.(i) <- a.(j);
  a.(j) <- t

let partition a l r =
  let p = a.(l) in
  let m = ref l in
  for i = l + 1 to r - 1 do
    if le a.(i) p then begin incr m; swap a i !m end
  done;
  if l <> !m then swap a l !m;
  !m

let rec quick_rec a l r =
  if l < r - 1 then begin
    let m = partition a l r in
    quick_rec a l m;
    quick_rec a (m + 1) r
  end

let quicksort a = quick_rec a 0 (Array.length a)
```

pivot are equally likely, we have fewer than $(\texttt{r}-1)+1$ assignments, each call to `swap` performing two assignments. This leads to a calculation analogous to that of the average number of comparisons. We have already done this computation for lists and obtained $2N \log N$. In the worst case, the pivot is always found at position `r-1`. The function `partition` then performs $2(\texttt{r}-1)$ assignments. There is, thus, a total of N^2 assignments. In the end, we obtain the results given in the table of figure 12.4.

	best case		average	worst case
comparisons	$N \log N$	-	$2N \log N$	$N^2/2$
assignments	-	0	$2N \log N$	N^2

Figure 12.4: Complexity of quicksort on arrays.

To avoid the worst case, to the extent possible, the two optimizations proposed in exercises 12.8 and 12.9 must be implemented.

12.3 Merge Sort

As with quicksort, merge sort also applies the principle of divide and conquer. It partitions the elements to be sorted into two parts of the same size, without trying to compare their elements. It then sorts the two parts recursively and merges them; hence the name *merge sort*. Thus, the worst case of quicksort, in which the two parts have disproportionate sizes, is avoided.

12.3.1 Lists

We begin by writing a function `split` that partitions a list into two lists of length differing by at most one. To ensure tail recursion, we pass two lists `l1` and `l2` as accumulators. When we reach the end of the list, we return the pair `(l1, l2)`.

```
let rec split l1 l2 = function
  | [] -> (l1, l2)
```

Otherwise, we consider the first element `x` and place it alternatively in `l1` or `l2`. The simplest way to achieve this is to exchange `l1` and `l2` at each call.

```
  | x :: l -> split (x :: l2) l1 l
```

The second step consists in writing a function `merge` that merges two sorted lists. For this, we traverse the two lists simultaneously.

```
let rec merge l1 l2 = match l1, l2 with
```

If one of the two is empty, we return the other.

```
  | [], l | l, [] ->
      l
```

Otherwise, we compare the first elements `x1` and `x2` of each list. If `x1` is smaller, it is placed at the head of the result, and we recursively merge the remainder `s1` of `l1` with `l2`. We proceed symmetrically if `x2` is smaller.

```
  | x1 :: s1, x2 :: s2 ->
      if le x1 x2 then x1 :: merge s1 l2 else x2 :: merge l1 s2
```

Note: We observe that `merge` is not tail recursive. We can fix this easily by adding an additional argument in which we accumulate the smaller elements.

Finally, we write a function `mergesort` that implements merge sort on a list `l`. If the list contains at most one element, we return it directly.

```
let rec mergesort l = match l with
  | [] | [_] ->
      l
```

Otherwise, we partition the list `l` into two lists using the function `split`:

```
  | _ ->
      let l1, l2 = split [] [] l in
```

We then sort the two lists `l1` and `l2` recursively, and merge the results using `merge`.

```
      merge (mergesort l1) (mergesort l2)
```

Program 93 — Merge sort on lists.

```
let rec split l1 l2 = function
  | [] -> (l1, l2)
  | x :: l -> split (x :: l2) l1 l

let rec merge acc l1 l2 = match l1, l2 with
  | [], l | l, [] ->
      List.rev_append acc l
  | x1 :: s1, x2 :: s2 ->
      if le x1 x2 then merge (x1 :: acc) s1 l2
      else merge (x2 :: acc) l1 s2

let rec mergesort l = match l with
  | [] | [_] ->
      l
  | _ ->
      let l1, l2 = split [] [] l in
      merge [] (mergesort l1) (mergesort l2)
```

Unlike the function `merge`, the function `mergesort` does not risk triggering a stack overflow: Since the list `l` is partitioned into two equal halves, the size of the stack is logarithmic. Program 93 below contains the complete code, with the function `merge` in its tail-recursive version.

We can avoid allocating the lists `l1` and `l2` by cleverly reusing the list `l` itself. Exercise 12.10 proposes such an optimization. This idea is notably used in the function `List.sort` of the OCaml standard library.

Complexity

The function `split` makes N allocations and no comparisons. The function `merge` makes twice as many allocations as comparisons: In the second branch of the pattern matching, there are as many comparisons as allocations, and in the first branch, the number of allocations performed by `List.rev_append` is equal to the length of `acc`, itself equal to the number of allocations/comparisons performed in the recursive part. Let $C(N)$ (resp. $f(N)$) be the total number of comparisons performed by `mergesort` (resp. `merge`). We then have the following equation:

$$C(N) = 2\,C(N/2) + f(N)$$

Indeed, the two recursive calls are made on lists of the same length $N/2$. In the best case, the function `merge` examines only the elements of one of the two lists, since they are all smaller than those of the other list. In this case, $f(N) = N/2$ and hence $C(N) \sim \frac{1}{2}N \log N$. In the worst case, `merge` examines every element, so that $f(N) = N - 1$; hence $C(N) \sim N \log N$. The analysis on the average case is subtler (see [15, ex 2 p. 646]) and gives $f(N) = N + O(1)$; hence $C(N) \sim N \log N$ as well. The number of assignments is easily deduced from the number of comparisons. In the end, we obtain the results given in the table of figure 12.5.

	best case	average	worst case
comparisons	$\frac{1}{2}N \log N$	$N \log N$	$N \log N$
allocations	$2N \log N$	$3N \log N$	$3N \log N$

Figure 12.5: Complexity of merge sort on lists.

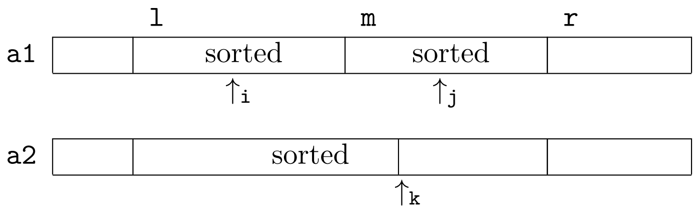
12.3.2 Arrays

The idea of merge sort on arrays is the same as for lists: We partition the elements to be sorted into two equal halves. We sort the two halves, and then merge them. We delimit the portion to be sorted by two indices `l` (inclusive) and `r` (exclusive). For the partition, it suffices to compute the middle index $\texttt{m} = \frac{\texttt{l}+\texttt{r}}{2}$. We then recursively sort the two parts delimited by `l` and `m` on the one hand, and `m` and `r` on the other. We have only to merge them. This proves extremely difficult to do in-place. It is simpler to use a second array, allocated once and for all at the start of the sort.

We begin by writing the function `merge` that implements the merging. It takes as arguments two arrays, `a1` and `a2`, and the three indices `l`, `m`, and `r`. The portions `a1`[`l`..`m`[and `a1`[`m`..`r`[are sorted. The objective is to merge them in `a2`[`l`..`r`[. For this, we traverse the two portions of `a1` (with two references `i` and `j`) and the portion of `a2` that is to be filled (with a `for` loop).

```
let merge a1 a2 l m r =
  let i = ref l in
  let j = ref m in
  for k = l to r - 1 do
```

In each iteration of the loop, the situation is therefore as follows:



We must determine the next value to put in `a2.(k)`. It is the smallest of the two values `a1.(!i)` and `a1.(!j)`. It is important, however, to also treat the case in which there are no more elements in one of the two halves. We determine whether the element is to be taken from the left half with the following test:

```
    if !i < m && (!j = r || le a1.(!i) a1.(!j)) then begin
```

This test checks whether the left half is non-empty and its first element is smaller than the first element of the right half, if such an element exists. In both cases, we copy the element into `a2.(k)` and increment the corresponding index.

```
      a2.(k) <- a1.(!i); incr i
    end else begin
      a2.(k) <- a1.(!j); incr j
    end
  done
```

Next, we implement merge sort as a function `mergesort`. We begin by allocating a temporary array `tmp` obtained by copying the array to be sorted.

```
let mergesort a =
  let tmp = Array.copy a in
```

The recursive part of merge sort is materialized via a recursive function `mergesort_rec` that takes as arguments the indices `l` and `r`, that delimit the portion to be sorted.

```
  let rec mergesort_rec l r =
```

If the segment contains at most one element, that is, if `l` ≥ `r-1`, there is nothing more to be done.

```
    if l < r - 1 then begin
```

Otherwise, we divide the interval into two equal halves by computing the middle position `m`, and then recursively sort `a`[`l`..`m`[and `a`[`m`..`r`[.

```
      let m = (l + r) / 2 in
      mergesort_rec l m;
      mergesort_rec m r;
```

Note that the computation of `(l+r)/2` cannot trigger an integer overflow, since the maximum size of an OCaml array is much smaller than the largest representable integer (see section *3.2 Runtime Model* for further details).

It remains to perform the merge. For this, we copy the entire portion `a`[`l`..`r`[into the array `tmp` and then call the function `merge`.

```
      Array.blit a l tmp l (r - l);
      merge tmp a l m r
    end
```

Finally, we sort the full array `a` by calling `mergesort` on all its elements.

```
  in
  mergesort_rec 0 (Array.length a)
```

Program 94 contains the complete code. Its efficiency can be further improved: As with quicksort, we can switch to insertion sort when the portion to be sorted becomes sufficiently small (see exercise 12.7). Another idea consists in avoiding the copy of `a` to `tmp`, which uses `Array.blit` (see exercise 12.12).

Program 94 — Merge sort on arrays.

```
let merge a1 a2 l m r =
  let i = ref l in
  let j = ref m in
  for k = l to r - 1 do
    if !i < m && (!j = r || le a1.(!i) a1.(!j)) then begin
      a2.(k) <- a1.(!i); incr i
    end else begin
      a2.(k) <- a1.(!j); incr j
    end
  done

let mergesort a =
  let tmp = Array.copy a in
  let rec mergesort_rec l r =
    if l < r - 1 then begin
      let m = (l + r) / 2 in
      mergesort_rec l m;
      mergesort_rec m r;
      Array.blit a l tmp l (r - l);
      merge tmp a l m r
    end
  in
  mergesort_rec 0 (Array.length a)
```

Complexity

The number of comparisons is exactly the same as in case of lists. The number of assignments is also the same: N assignments in the function `merge` (each element being copied from `a1` to `a2`) and N assignments performed by `Array.blit`.

Let $A(N)$ be the total number of assignments in `mergesort`. We then have:

$$A(N) = 2A(N/2) + 2N,$$

that is, $2N \log N$ assignments in total. In the end, we obtain the results of figure 12.6.

	best case	average	worst case
comparisons	$\frac{1}{2}N \log N$	$N \log N$	$N \log N$
assignments	$2N \log N$	$2N \log N$	$2N \log N$

Figure 12.6: Complexity of merge sort on arrays.

12.4 Heapsort

Heapsort is based on the use of a priority queue. Such a structure permits the addition of elements and the extraction of its smallest element. The idea behind heapsort is the following: We construct a priority queue containing all the elements to be sorted and then extract the elements one by one. Since the extraction operation returns the smallest element each time, the elements come out in increasing order.

In principle, any priority queue structure could be used. However, a structure in which the addition and extraction operations have cost $O(\log N)$ gives an optimal sorting algorithm in $O(N \log N)$. Both of the heap structures presented in chapter 6 have such complexity.

12.4.1 Lists

We assume we are given a functor `PriorityQueue` that implements a priority queue structure. In principle, we can choose between imperative and persistent structures. However, there is no advantage to choosing a persistent structure here. We therefore use the imperative heap structure described in section *6.3 Imperative Priority Queues*. Its interface is given again in program 95.

Program 95 — Minimal signature for imperative priority queues.

```
type t
type elt
val create : unit -> t
val is_empty : t -> bool
val add : elt -> t -> unit
val get_min : t -> elt
val remove_min : t -> unit
```

We apply the functor `PriorityQueue` by passing as argument a module that compares elements with `compare`. To avoid having to reverse the list at the end, we order elements in decreasing order, that is, the element of highest priority for the priority queue will be the largest element for the order relation `lt`. We therefore write:

```
module Heap = PriorityQueue(struct
                type t = elt
                let compare x y = compare y x
              end)
```

The function `heapsort` takes a list `l` of elements to be sorted and begins by creating an empty heap `h`.

```
let heapsort l =
  let h = Heap.create () in
```

The function then adds all the elements of the list `l` to the heap.

```
  List.iter (fun x -> Heap.add x h) l;
```

Next, we create a reference `res` that will receive the sorted result. We extract the elements from the heap with the help of a `while` loop.

Program 96 — Heapsort on lists.

```
let heapsort l =
  let h = Heap.create () in
  List.iter (fun x -> Heap.add x h) l;
  let res = ref [] in
  while not (Heap.is_empty h) do
    let x = Heap.get_min h in
    Heap.remove_min h;
    res := x :: !res
  done;
  !res
```

```
let res = ref [] in
while not (Heap.is_empty h) do
```

If the heap is not empty, we extract its smallest element `x` using `Heap.get_min` and then remove it from the heap using `Heap.remove_min`.

```
  let x = Heap.get_min h in
  Heap.remove_min h;
```

We add `x` at the head of the list contained in `res` and then repeat.

```
  res := x :: !res
done;
```

Once the heap becomes empty, it suffices to return the contents of `res`.

```
!res
```

Note that the fact of having reversed the order relation in the heap structure means that we do not have to reverse the list `!res` at the end of `heapsort`. Program 96 contains the complete code.

Complexity

The complexity of heapsort depends on the complexity of the heap structure used. As we mentioned in the introduction to this section, if the operations `add` and `remove_min` of the heap structure have at most logarithmic cost, then the sort itself has cost $O(N \log N)$ in the worst case, which is optimal. If we now assume that the heap structure used is that of section *6.3 Imperative Priority Queues*, we can give a more precise result. Indeed, we then know that `add` and `remove_min` each make a maximum of $2 \log k$ comparisons to add or remove an element from a heap containing k elements, so that the total number of comparisons in the worst case is at most:

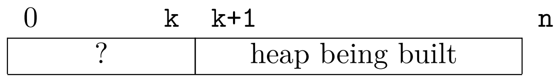
$$4(\log 1 + \log 2 + \cdots + \log N) \sim 4N \log N$$

In fact, we will see in the next section that the construction of the heap has only linear cost, so that the total is equivalent to $2N \log N$.

12.4.2 Arrays

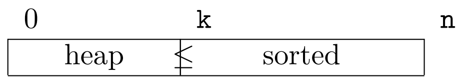
To implement heapsort on an array `a`, we will construct the heap structure directly inside this array. The organization of the heap in the array is exactly the same as in section *6.3 Imperative Priority Queues*: The left and right children of a node stored at index i are respectively stored at indices $2i + 1$ and $2i + 2$. As in the previous section, we construct a heap for the reverse order relation, that is, a heap in which the largest element is to be found at the root.

To construct the heap, we consider the elements of the array from right to left. In each iteration, we have the following situation:



The portion `a`[`k+1`..`n`[contains the bottom of the heap being constructed, that is, it is a forest of heaps whose roots are situated at indices i such that $\mathtt{k} < i < 2\mathtt{k} + 3$. We then make the value `a.(k)` descend to its place in the heap rooted at k. Once all the elements have been traversed, we have a single heap rooted at 0.

The second step consists in deconstructing the heap. For this, we exchange its root r in a.(0) with the element v in a.(n-1). The value r is then in its final place. We then reestablish the heap structure on a[0..n-1[by making v descend to its place in the heap of size n-1 rooted at 0. We repeat the operation for positions n-1, n-2, etc. In each iteration k, we have the following situation:



There is a heap in the portion a[0..k[, all of whose elements are smaller than those of the portion a[k..n[that is already sorted.

The two steps of the above algorithm use the same operation consisting in making a value descend until it reaches its position in a heap. This is implemented using a recursive function move_down that takes as arguments an array a, an index k, a value v, and an upper bound on the indices, n.

```
  let rec move_down a k v n =
```

We assume that we already have a heap h_1, rooted at 2k+1, so that 2k+1 $<$ n, and similarly a heap h_2, rooted at 2k+2, so that 2k+2 $<$ n. The objective is to construct a heap rooted at k, containing v and all the elements of h_1 and h_2. We begin by determining whether the heap rooted at k will be reduced to a leaf, that is, whether the heap h_1 is empty. If this is so, we assign the value v to a.(k), and we are done.

```
    let r = 2 * k + 1 in
    if r >= n then
      a.(k) <- v
```

Otherwise, we determine the index rmax of the largest of the two roots of h_1 and h_2, by carefully handling the case in which h_2 is empty.

```
    else
      let rmax =
        if r+1 < n then if lt a.(r) a.(r+1) then r+1 else r
        else r in
```

If the value v is greater than or equal to a.(rmax), the descent is complete, and it suffices to assign v to a.(k).

```
      if le a.(rmax) v then a.(k) <- v
```

Otherwise, we move a.(rmax) up and continue the descent of v recursively at the position rmax.

```
      else begin a.(k) <- a.(rmax); move_down a rmax v n end
```

The sorting function takes an array a as argument:

```
  let heapsort a =
    let n = Array.length a in
```

We begin by constructing the heap bottom-up by calling move_down. We avoid unnecessary calls for heaps reduced to single leaves by beginning the loop at $\lfloor \frac{n}{2} \rfloor - 1$. (Indeed, for every strictly greater index k, we have 2k+1 $\geq$ n.)

```
    for k = n/2 - 1 downto 0 do move_down a k a.(k) n done;
```

Once the heap is fully constructed, we extract its elements one by one, in decreasing order. As explained above, for each index k, we swap a.(0) with the value v in a.(k) and then move v down to its place in the heap.

```
    for k = n-1 downto 1 do
      let v = a.(k) in a.(k) <- a.(0); move_down a 0 v k
    done
```

Note that the specification of move_down allows us to avoid assigning v to a.(0) before beginning the descent. Program 97 below contains the complete code.

Complexity

Let us first consider the cost of a call to move_down a k v n. The number of recursive calls is bounded by $\log$ n since the value of k is doubled at each call. Furthermore, move_down performs at most two comparisons and an assignment in each call. Thus, in the worst case, we have a total of $2 \log$ n comparisons and $\log$ n assignments.

Program 97 — Heapsort on arrays.

```
let rec move_down a k v n =
  let r = 2 * k + 1 in
  if r >= n then
    a.(k) <- v
  else
    let rmax =
      if r+1 < n then if lt a.(r) a.(r+1) then r+1 else r
      else r in
    if le a.(rmax) v then a.(k) <- v
    else begin a.(k) <- a.(rmax); move_down a rmax v n end

let heapsort a =
  let n = Array.length a in
  for k = n/2 - 1 downto 0 do move_down a k a.(k) n done;
  for k = n-1 downto 1 do
    let v = a.(k) in a.(k) <- a.(0); move_down a 0 v k
  done
```

For the sorting itself, we can roughly bound the number $C(N)$ of comparisons in each call to `move_down` by $2 \log N$, that is, at worst a total of $3N \log N$ comparisons ($N \log N$ comparisons for the first step and $2N \log N$ for the second). Similarly, the total number of assignments is, at worst, $\frac{3}{2} N \log N$.

In fact, we can be more precise and show that the first step of the algorithm, namely, the construction of the heap, has linear cost. (See for example [7, Sec. 7.3].) It follows that only the second part of the algorithm contributes to the asymptotic complexity so that $C(N) \sim 2N \log N$. For an analysis on average of heapsort, the reader is referred to *The Art of Computer Programming* [15, vol. 3, p. 152].

Note that heapsort executes in constant memory. Indeed, the function `move_down` performs only tail-recursive calls and, moreover, all computations are performed in-place inside the array `a`.

12.5 Optimal Complexity

We present here a quick proof of the fact that the optimal complexity of a sorting algorithm that performs only element comparisons is in $O(N \log N)$.

We can visualize such an algorithm as a binary tree. Each internal node symbolizes a comparison that has been performed, the left (resp. right) subtree representing the rest of the algorithm when the test is positive (resp. negative). Each leaf constitutes a possible result, that is, a permutation performed on the initial sequence. If we assume N distinct elements, there are $N!$ possible permutations and, hence, at least $N!$ leaves in this tree. Its height is consequently at least $\log N!$. Now the longest path from the root to a leaf corresponds to the largest number of comparisons performed by the algorithm on an input. There is, therefore, an input for which the number of comparisons is at least $\log N!$. By Stirling's formula, we know that $\log N! \sim N \log N$. For a more detailed proof, the reader is referred to *The Art of Computer Programming* [15, Sec. 5.3].

12.6 Experimental Evaluation

In this section, we compare the different sorting algorithms presented in this chapter empirically.

Evaluation Protocol

For a given list or array, we evaluate the performance of several algorithms on the same input as follows. We execute each sorting algorithm five times, measuring the computation time in each case. We exclude the smallest and the largest values, and take the average of the three remaining values.

For lists, we consider randomly constructed as well as already sorted lists, with lengths $2^i \times 1000$ for i ranging from 0 to 10. For arrays, we consider randomly filled as already sorted arrays, with lengths $2^i \times 1000$ for i ranging from 0 to 12. It is important to consider the case of data that is already sorted because that represents a realistic situation for which we wish to verify that our sorting algorithms are efficient.

Evaluation on Lists

We compare the algorithms listed in figure 12.7. Figures 12.8 and 12.9 present the results.

For random lists (figure 12.8), we observe the following: Insertion sort (`insertion`) can only be used in case of small lists, given its quadratic complexity. (We do not give the computation time for lists longer than 16,000.) Two of the three quicksort algorithms, (`quicksort` and `quicksort (rand)`) triggered a stack overflow on lists of length 2^{10}. The algorithm `heapsort (trees)`, which uses binary trees, is penalized due to the construction of the trees, even if its complexity is optimal. The other sorting algorithms have comparable efficiency, the fastest being that of the OCaml standard library.

`insertion`	insertion sort of program 89
`quicksort`	quicksort of program 91
`quicksort (log stack)`	quicksort with the first call made to the smaller of the two lists after partitioning (see exercise 12.4)
`quicksort (rand)`	quicksort with a randomly chosen pivot (see exercise 12.5)
`mergesort`	merge sort of program 93
`mergesort (reuse)`	merge sort with optimization consisting in not constructing intermediate lists when partitioning (see exercise 12.10)
`List.sort`	the function `List.sort` of the OCaml standard library (version 3.11.2)
`heapsort (arrays)`	heapsort of program 96, using the structure based on arrays presented in section 6.3
`heapsort (trees)`	heapsort of program 96, using the structure based on binary trees presented in section 6.4

Figure 12.7: Sorting algorithms on lists.

In case of already sorted lists (figure 12.9), we observe the following: Insertion sort (`insertion`) has quadratic complexity. (The favorable case would have been that of a list sorted in decreasing order.) The first two quicksort algorithms, `quicksort` and `quicksort (log stack)`, have quadratic complexity, which is explained by the arbitrary choice of the first element of the list as pivot. (We do not give the computation time for lists of length greater than 16,000.) The third quicksort algorithm, `quicksort (rand)`, behaves well but triggers a stack overflow on lists of size $2^{10} \times 1000$. The algorithm `heapsort (trees)` remains penalized due to the construction of the trees, as in case of random lists.

We may draw the following conclusions: Merge sort is the one that best combines simplicity and efficiency. Even its simplest version has an efficiency comparable to the optimized version of the OCaml standard library. Heapsort always performs well, but its implementation is considerably more complex than that of merge sort. Finally, we observe that using quicksort on lists is not straightforward because we must, at a minimum, combine the optimizations of exercises 12.4 and 12.5, to avoid both stack overflows and the unfavorable case of quadratic complexity.

Evaluation on Arrays

We now compare the different techniques for sorting arrays of integers based on algorithms listed in figure 12.10. Figures 12.11 and 12.12 present the results.

For random arrays (figure 12.11), we observe the following: As in case of lists, insertion sort can only be used for small arrays. The two heapsort algorithms, `heapsort` and `Array.sort`, are slightly less efficient than merge sort and quicksort. The two most efficient algorithms are both merge sorts.

For already sorted arrays (figure 12.12), we observe the following: Here, the first three quicksort algorithms, `quicksort`, `quicksort (insertion)`, and `quicksort (log stack)`, are in their most unfavorable case, which may be explained by the arbitrary choice of the left-most element as pivot. (We do not give the computation time for arrays of length greater than 16,000.) The version of quicksort with a randomly chosen pivot achieves a good efficiency, comparable to that of `mergesort`.

Of all the sorting algorithms in $O(N \log N)$, heapsort is among the least efficient while merge sort is among the most efficient. Here, insertion sort (`insertion`) is the most efficient, since we are in the favorable case of linear complexity. (The unfavorable case would have been that of an array sorted in decreasing order.)

We may draw the following conclusions: If we are allowed to allocate an intermediate array, in other words, to temporarily double memory use, then merge sort is the best choice. As in case of lists, it combines simplicity and efficiency. If, however, we wish to sort an array in-place, we must either use heapsort or quicksort, taking care to combine it with the optimizations of exercises 12.6 and 12.8, to avoid both stack overflows and quadratic behavior due to a bad choice of pivot.

12.7 Exercises

12.1 Write a function `two_way_sort: bool array -> unit` that sorts a boolean array in-place, performing only swaps of elements, with the convention that `false` $<$ `true`. The complexity should be proportionate to the number of elements.

12.2 (Dijkstra's Dutch national flag problem) Write a function that sorts an array in-place. The array contains values representing the three colors of the Dutch flag, namely:

```
type color = Blue | White | Red
```

Only swaps of elements in the array may be performed. The complexity should be proportionate to the number of elements.

12.3 More generally, we consider the case of an array containing k distinct values. To simplify matters, we assume that we are dealing with integers $0, \dots, k-1$. Write a function that sorts such an array in $O(\max(k, N))$ time, where N is the size of the array.

Quicksort

12.4 One way to avoid a potential stack overflow in the function `quicksort` on lists is by first implementing a non-tail-recursive call to sort the smaller of the two lists, followed by a tail-recursive call to sort the larger one. Hint: For the tail-recursive call, generalize the function `quicksort` so that it takes as additional arguments the two lists `left` and `right`, so that `quicksort left l right` returns the concatenation, in this order, of `left`, the sorted list `l`, and `right`. Show that the size of the stack remains logarithmic even in the worst case.

12.5 Modify the function `partition` on lists to use a randomly chosen element of the list as pivot. The aim is to avoid the unfavorable case of a sorted list, where quicksort would then have quadratic complexity.

12.6 One way of avoiding a potential stack overflow in the function `quick_rec` on arrays is to begin by performing the recursive call on the smaller of the two portions. Show that the size of the stack remains logarithmic even in the worst case.

12.7 A classic idea to speed up sorting algorithms consists in switching to insertion sort when the number of elements to be sorted is small, that is, less than a constant fixed in advance (for example, 5). Modify the quicksort implementation on arrays taking this idea into consideration. Use the function `insertion_sort` on arrays (program 90) and generalize it by passing two indices, `l` and `r`, to delimit the portion of the array to be sorted.

12.8 Modify the quicksort implementation on arrays to use a pivot chosen randomly in the segment `a`[`l`..`r`[. As with lists, the idea is to avoid the unfavorable case of an already sorted array, where quicksort would then have quadratic complexity. One simple solution is to shuffle the array *before* sorting it, for example using exercise 2.11.

12.9 When numerous elements of the array are equal, the preceding exercise does not always suffice to guarantee a good choice of pivot. Modify the function `partition` so that it separates the elements that are strictly smaller than the pivot (to the left), those equal to the pivot (in the center), and those strictly greater than the pivot (to the right). Instead of two indices `m` and `i` dividing the segment of the array into three parts, as illustrated in figure 12.2.2, we will use three indices to divide the segment of the array into four parts. The new function `partition` must now return two indices. Modify the function `quick_rec` accordingly. Partitioning into three parts in this way is the subject of exercise 12.2.

Merge Sort

12.10 One source of inefficiency of merge sort on lists is the function `split`. Given a list `l` containing n elements, `split l` constructs two completely new lists, `l1` and `l2`, containing all the elements of `l`. We can avoid this construction by using only the list `l`, without allocating any new lists. Rather than taking every other element of `l` to construct `l1` and `l2`, we choose for `l1` the $\lfloor n/2 \rfloor$ first elements of `l` and for `l2` the $\lceil n/2 \rceil$ last elements. To construct `l2`, write a function `chop: int -> 'a list -> 'a list` such that `chop n1 l` returns the suffix of `l` obtained by removing its `n1` first elements. As for `l1`, it suffices to represent it as a prefix of `l`, that is, to add an argument `n1` to the function `mergesort`, indicating that the first `n1` elements of `l` must be sorted. Rewrite the function `mergesort` using this idea. Reuse the function `merge` of program 93. Calculate the complexity in number of allocations of the function `mergesort`.

12.11 One way of optimizing the function `mergesort` on almost sorted arrays is by avoiding the merging when, after both recursive calls, the elements of the left half are all smaller than the elements of the right half. We can easily test this by comparing the right-most element of the left half with the left-most element of the right half. Modify the function `mergesort` using this idea.

12.12 To avoid copying `a` into `tmp` with `Array.blit` in the function `mergesort` on arrays, one possibility is to sort the two halves of the array `a` while simultaneously shifting them to the array `tmp`, then merging `tmp` back into `a` as is already done. However, sorting the elements of `a` into `tmp` requires, conversely, that the two halves be sorted in-place, then merged into `tmp`. We therefore need two mutually recursive sorting functions. We can make do with only one by passing an additional parameter indicating whether the sorting must be done in-place or in `tmp`. Modify the functions `mergesort` and `mergesort_rec` following this idea.

12.13 In merge sort, as in quicksort, we can switch to insertion sort when the number of elements to be sorted is small (see exercise 12.7).

$\frac{N}{1000}$	insertion	quicksort	quicksort (log stack)	quicksort (rand)	mergesort	mergesort (reuse)	List.sort	heapsort (arrays)	heapsort (trees)
2^4	6.03	0.02	**0.01**	0.02	0.02	**0.01**	**0.01**	**0.01**	0.02
2^5	-	0.05	**0.03**	0.05	0.05	**0.03**	**0.03**	0.04	0.05
2^6	-	0.10	0.07	0.14	0.12	0.08	**0.05**	0.09	0.13
2^7	-	0.30	0.17	0.27	0.28	0.19	**0.12**	0.19	0.37
2^8	-	0.63	0.38	0.67	0.63	0.45	**0.27**	0.42	0.91
2^9	-	1.39	0.82	1.38	1.40	0.99	**0.65**	0.93	2.08
2^{10}	-	-	1.75	-	3.09	2.20	**1.41**	1.92	4.84

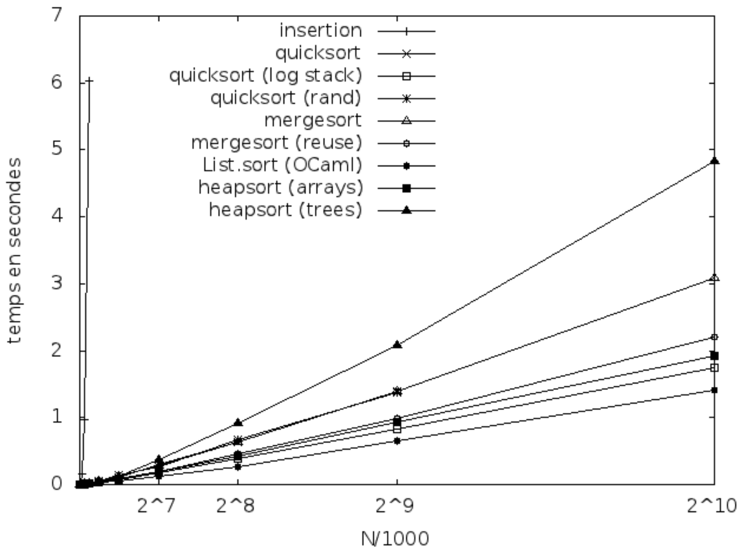


Figure 12.8: Comparison of sorting algorithms on random lists.

$\frac{N}{1000}$	insertion	quicksort	quicksort (log stack)	quicksort (rand)	mergesort	mergesort (reuse)	List.sort	heapsort (arrays)	heapsort (trees)
2^4	14.94	-	8.94	0.02	0.02	**0.01**	**0.01**	0.03	0.03
2^5	-	-	-	0.06	0.05	**0.02**	**0.02**	0.06	0.07
2^6	-	-	-	0.13	0.11	0.05	**0.04**	0.12	0.19
2^7	-	-	-	0.26	0.27	**0.11**	**0.11**	0.25	0.49
2^8	-	-	-	0.77	0.62	**0.25**	0.27	0.54	1.18
2^9	-	-	-	1.66	1.38	**0.58**	0.59	1.16	2.76
2^{10}	-	-	-	-	3.07	1.28	**1.26**	2.43	6.36

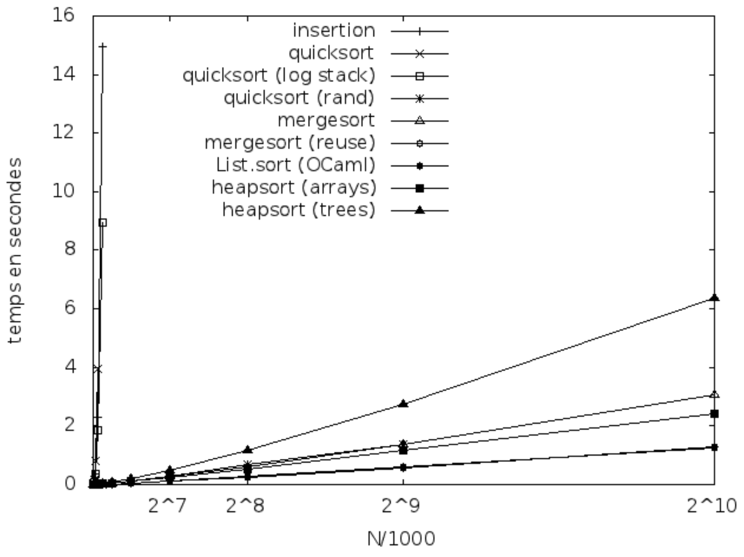


Figure 12.9: Comparison of sorting algorithms on already sorted lists.

`insertion`	insertion sort of program 90
`quicksort`	quicksort of program 92
`quicksort (insertion)`	quicksort using an insertion sort on small segments (see exercise 12.7)
`quicksort (log stack)`	quicksort with the first call on the smallest of the two segments (see exercise 12.6)
`quicksort (rand)`	quicksort with a pivot chosen randomly (see exercise 12.8)
`mergesort`	merge sort of program 94
`merge sort (swap)`	merge sort with the optimization consisting in avoiding the use of `Array.blit` (see exercise 12.12)
`Array.stable_sort`	the function `Array.stable_sort` of the OCaml standard library (version 3.11.2)
`heapsort`	heapsort of program 97
`Array.sort`	heapsort of the OCaml standard library

Figure 12.10: Sorting algorithms on arrays of integers.

$\frac{N}{1000}$	insertion	quicksort	quicksort (insertion)	quicksort (log stack)	quicksort (rand)	mergesort	mergesort (swap)	Array.stable_sort	heapsort	Array.sort
2^4	1.65	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
2^5	-	0.02	0.02	0.02	0.02	0.02	0.02	**0.01**	0.02	0.02
2^6	-	0.04	0.04	**0.03**	0.04	0.04	**0.03**	**0.03**	0.04	0.05
2^7	-	0.08	0.08	0.08	0.08	0.09	0.07	**0.06**	0.10	0.10
2^8	-	0.16	0.15	0.16	0.18	0.19	0.14	**0.13**	0.21	0.21
2^9	-	0.33	0.33	0.33	0.36	0.41	0.29	**0.26**	0.46	0.46
2^{10}	-	0.74	0.70	0.71	0.72	0.84	0.61	**0.57**	1.01	1.02
2^{11}	-	1.43	1.40	1.41	1.64	1.76	1.28	**1.10**	2.36	2.35
2^{12}	-	3.05	3.01	3.05	3.41	3.74	2.69	**2.34**	5.37	5.40

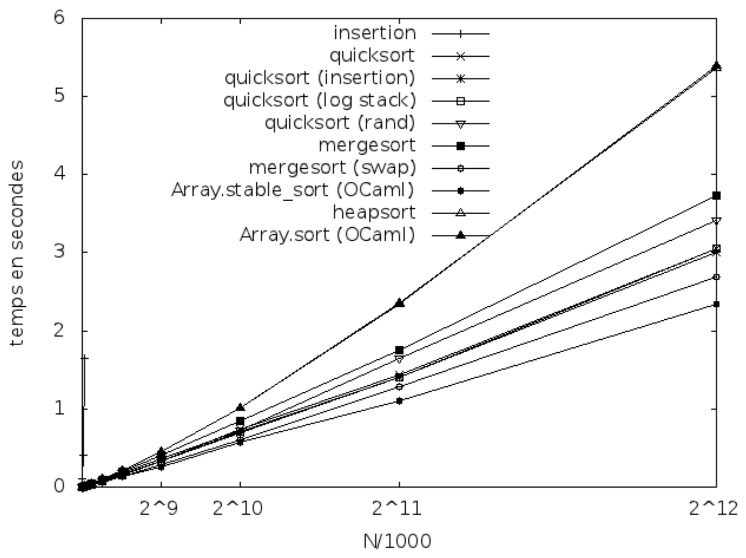


Figure 12.11: Comparison of sorting on random arrays.
— openly licensed via CC BY SA 4.0 —

$\frac{N}{1000}$	insertion	quicksort	quicksort (insertion)	quicksort (log stack)	quicksort (rand)	mergesort	mergesort (swap)	Array.stable_sort	heapsort	Array.sort
2^4	**0.00**	1.98	2.06	2.07	0.01	0.01	0.01	**0.00**	0.01	0.01
2^5	**0.00**	-	-	-	0.02	0.02	**0.01**	**0.01**	0.02	0.02
2^6	**0.00**	-	-	-	0.04	0.04	**0.02**	**0.02**	0.04	0.04
2^7	**0.00**	-	-	-	0.07	0.07	0.05	**0.04**	0.10	0.08
2^8	**0.00**	-	-	-	0.16	0.15	0.10	**0.08**	0.21	0.16
2^9	**0.01**	-	-	-	0.32	0.33	0.21	**0.18**	0.45	0.37
2^{10}	**0.02**	-	-	-	0.71	0.67	0.43	**0.37**	0.94	0.73
2^{11}	**0.05**	-	-	-	1.47	1.38	0.90	**0.76**	1.98	1.53
2^{12}	**0.11**	-	-	-	3.11	2.98	1.94	**1.65**	4.28	3.34

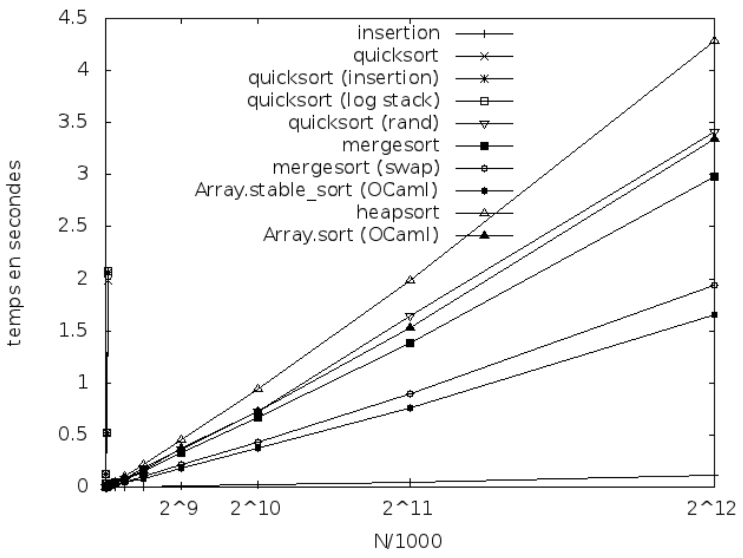


Figure 12.12: Comparison of sorting on already sorted arrays.
— openly licensed via CC BY SA 4.0 —

13 Graph Algorithms

In this chapter, we present graph algorithms. These algorithms are independent of the underlying graph data structure. They can be used, in particular, with all the graph structures of chapter 7. Below, we will take as given a module `G` whose signature includes at least the types, functions, and modules of program 98. Graphs are of type `graph`, and their vertices are of type `vertex`. The function `iter_vertex` iterates over the vertices of a given graph, applying a function passed as argument to each of them. The function `iter_succ` iterates over the successors of a given vertex. The module `H` provides a hash table structure with vertices as keys.

13.1 Breadth-First Search

The first traversal that we present is called *breadth-first search* (BFS). It consists in exploring the graph by moving outwards "in concentric circles," from a particular vertex s, called the root. We first traverse the vertices that are exactly one arc away from the root s, then those situated two arcs away, and so forth. Consider, for example, the graph on the left in figure 13.1. Beginning from vertex 5, we first visit vertices 4 and 1, which are directly linked to vertex 5.

Program 98 — Minimal signature of a graph structure.

```
type graph
type vertex
val iter_vertex: (vertex -> unit) -> graph -> unit
val iter_succ: (vertex -> unit) -> graph -> vertex -> unit
module H: HashTable with type key = vertex
```

We then visit vertices 0, 6 and 2, followed by vertex 7, and finally vertex 3. The graph, with distances to the root indicated as exponents, is shown on the right.

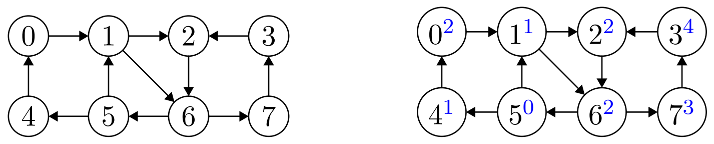


Figure 13.1: Breadth-first search of a graph.

We write breadth-first search as a function `iter_bfs` that takes as arguments a graph `g`, a root `s`, and a function `f` that will be applied to the elements of `g` as these are discovered. We begin by creating a hash table that will contain the vertices already discovered by the traversal.

```
let iter_bfs f g s =
  let visited = H.create () in
```

The search itself is based on the use of a *queue* into which vertices are inserted as they are discovered. The idea is that, at any point in time, the queue will contain the vertices situated at distance d from the root, followed by the vertices situated at distance $d+1$:

$$\leftarrow \boxed{\text{vertices at distance } d \;\Big|\; \text{vertices at distance } d+1} \leftarrow$$

This is the concretization of our notion of “concentric circles,” more precisely of the two consecutive concentric circles that are being considered. This property is crucial for the correctness of breadth-first search[1]. Here, we use the module `Queue` to implement the queue. Initially, it contains only the root. It is useful to write a function `add` that adds a vertex simultaneously in the hash table and the queue. We begin by using it on the root `s`.

```
let queue = Queue.create () in
let add w = H.add visited w (); Queue.add w queue in
add s;
```

Another important property is that each vertex contained in the queue is also contained in the table `visited`, which the function `add` guarantees in an obvious manner. We use a loop to examine vertices in the order in which they are discovered, until all vertices reachable from the root have been examined. As long as the queue is not empty, we extract the first element `v` of the queue and apply the visitor function `f` to it.

```
while not (Queue.is_empty queue) do
  let v = Queue.pop queue in
  f v;
```

For each successor `w` of `v` not yet found, that is, not contained in the table `visited`, we use `add` to add it to both the queue and the table.

```
  iter_succ (fun w -> if not (H.mem visited w) then add w) g v
done
```

This completes the `while` loop and the function `iter_bfs`. The complete code is given in program 99 below.

This code applies equally well to undirected and directed graphs. Note that some vertices may not be discovered by this algorithm. These are the vertices v for which there is no path between the root and v. In other words, breadth-first search determines the set of vertices that can be reached from the root and even determines their minimal distance from the root in number of arcs (see exercise 13.1).

[1]For a detailed proof of the correctness of breadth-first search, consult [7, chap. 23].

Program 99 — Breadth-first search.

```
let iter_bfs f g s =
  let visited = H.create () in
  let queue = Queue.create () in
  let add w = H.add visited w (); Queue.add w queue in
  add s;
  while not (Queue.is_empty queue) do
    let v = Queue.pop queue in
    f v;
    iter_succ (fun w -> if not (H.mem visited w) then add w) g v
  done
```

The complexity of breadth-first search is easy to determine. Each vertex is placed in the queue at most once and is therefore examined at most once. Each arc is considered at most once, when its starting vertex is examined. For a graph with N vertices and E arcs, the complexity is therefore $O(N+E)$, which is optimal. The complexity in space is $O(N)$, since the queue, like the hash table, may contain (almost) all vertices in the worst case.

There is another way to implement breadth-first search, without using a queue. As noted above, the queue has a very particular structure, with vertices at distance d followed by vertices at distance $d+1$. We see, therefore, that the queue structure is not really necessary. All we need are two "buckets," one containing the vertices at distance d, and the other, those at distance $d+1$. We may concretize these, for instance, using lists. When we are done with bucket d, we swap the two buckets. This does not affect the complexity.

13.2 Depth-First Search

The second graph traversal that we present is *depth-first search* (DFS). It uses a backtracking algorithm: As long as we can follow an arc, we do so, and when it is no longer possible, we backtrack. As with breadth-first search, we mark vertices as they are discovered, to avoid getting stuck in a cycle. Consider the example of the graph on the left in figure 13.2. We begin a depth-first search from vertex 2. Two arcs come out of this vertex, $2 \to 4$ and $2 \to 5$. We arbitrarily choose to consider the arc $2 \to 5$ first and, accordingly, proceed to vertex 5. No arc comes out of 5, so we cannot continue. We therefore backtrack to vertex 2, and consider the second outgoing arc, $2 \to 4$. From 4, we can only follow the arc $4 \to 3$. Similarly, from 3, we can only follow the arc $3 \to 1$. Two arcs emerge from vertex 1: $1 \to 0$ and $1 \to 4$. We choose to follow the arc $1 \to 4$ first. It leads to a vertex that has already been visited. We therefore backtrack. Returning to 1, we consider the other arc, $1 \to 0$, which leads to 0. From there, the only outgoing arc leads to 3, which has also already been visited. We therefore return to 0, then to 1, then to 3, then to 4, and finally to 2. The traversal is complete. If we redraw the graph with the order in which the vertices were discovered as exponents, we get the graph on the right in figure 13.2.

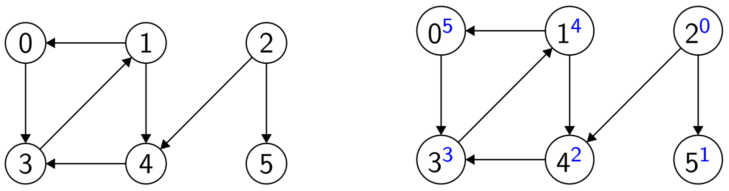


Figure 13.2: Example of depth-first search.

As with breadth-first search, we will make use of a hash table containing the vertices already discovered.

```
let iter_dfs f g s =
  let visited = H.create () in
```

Depth-first search is written as a recursive local function `visit` taking a vertex `v` as argument. Its code is a few lines long:

```
  let rec visit v =
    if not (H.mem visited v) then begin
      H.add visited v ();
      f v;
```

Program 100 — Depth-first search.

```
let iter_dfs f g s =
  let visited = H.create () in
  let rec visit v =
    if not (H.mem visited v) then begin
      H.add visited v ();
      f v;
      iter_succ visit g v
    end
  in
  visit s
```

```
      iter_succ visit g v
    end
```

If vertex `v` has already been discovered, there is nothing more to be done. Otherwise, we mark it and visit it with the function `f`. Then, we consider each successor `w`, recursively triggering a depth-first search on each one. One detail is crucial: We added `v` to the table `visited` *before* considering its successors. This is what keeps us from getting stuck in a cycle. To conclude the function `iter_dfs`, we have only to call `visit` on the root `s`:

```
  in
  visit s
```

The complete code is given in program 100.

The complexity is $O(N + E)$, by the same logic as in case of breadth-first search. Space complexity is subtler since it is the stack of recursive calls that contains the vertices being visited, and that plays the role of the queue in breadth-first search. Since in the worst case all the vertices can be present in the stack or hash table, it follows that the space complexity is $O(N)$.

Like breadth-first search, depth-first search allows us to determine the set of vertices that can be reached from the root `s`. Figure 13.3 illustrates another example, where depth-first search is triggered from vertex 1. Vertices 2 and 5 are not discovered. In particular, depth-first search, like breadth-first search, is a way to determine the existence of a path between a particular vertex, namely, the root, and other vertices of the graph. If that is the only objective (if, for example, the minimal distance does not interest us), depth-first search is generally more efficient. Indeed, its memory use (that is, the call stack) is often inferior to that of breadth-first search. The typical example is that of a tree, in which memory use is limited to the height of the tree for depth-first search, but could be as large as the entire tree in case of breadth-first search. Depth-first search has many other applications that lie beyond the scope of this book. See, for example, *Introduction to Algorithms* [7].

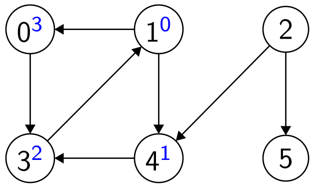


Figure 13.3: Another depth-first search.

13.3 Shortest Path

We consider here graphs whose arcs are labeled with weights (representing distances, for example). We seek the shortest path between two vertices, where the length is no longer the number of arcs but rather the sum of weights along the path. Thus, if we consider the weighted graph of figure 13.4, the shortest path from vertex 2 to vertex 0 is of length 5. This is the path $2 \to 4 \to 3 \to 1 \to 0$, which is shorter than the path $2 \to 1 \to 0$, of length 6, even if the latter contains fewer arcs. Similarly, the shortest path from vertex 2 to vertex 5, via vertex 4, is of length 2.

As with breadth- and depth-first search, we take as given a vertex `s`, from which we seek to determine the shortest paths to the other vertices. We also require the weight of each arc. One possibility is to modify the graph structure

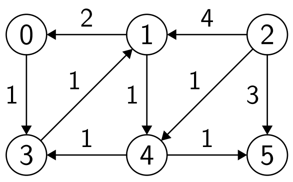


Figure 13.4: Example of a weighted graph.

to include this. Here, however, we take as given a weight function `weight` of the following type, where weights are of type `float`:

```
val weight: graph -> vertex -> vertex -> float
```

This function will only be called with arguments `g`, `x`, and `y` if there is an arc between `x` and `y` in the graph `g`.

We present two algorithms to solve the problem. The first, Dijkstra's algorithm, assumes that the weights are always non-negative. The second, the Bellman-Ford algorithm, involves no such hypothesis regarding the weights.

Dijkstra's Algorithm

Dijkstra's algorithm is a generalization of breadth-first search. Here, too, we proceed in "concentric circles." The difference is that the radii of these circles represent distance in terms of total weight rather than number of arcs. Thus, in the example above, starting from the root 2, we first reach the vertices at distance 1 (namely, 4), then those at distance 2 (namely, 3 and 5), then those at distance 3 (namely, 1), and finally those at distance 5 (namely, 0). A possible complication is that we may reach a vertex at a certain distance (for example, vertex 5 with the arc $2 \to 5$), and then later find a shorter path by taking different arcs (for example, $2 \to 4 \to 5$). A queue no longer suffices as it did in breadth-first search. Instead, we will use a *priority queue* (see chapter 6). The priority queue will contain the vertices already found, ordered by distance from the root. When a better path to a vertex is found, the vertex is reinserted in the queue with a higher priority, that is, a shorter distance[2].

We now describe the code of Dijkstra's algorithm. To implement the priority queue, we assume a functor `PriorityQueue`, parametrized by a type `t` equipped with a total order. (Such functors were presented in chapter 6.) Here, the type `t` will be that of pairs (v, d), where v is a vertex and d is its distance from the root. The order on these pairs is that obtained by comparing the distances. We introduce the following module:

```
module VertexDistance = struct
  type t = vertex * float
  let compare (_, d1) (_, d2) = Stdlib.compare d1 d2
end
```

It only remains to apply the functor `PriorityQueue` to the module `VertexDistance`, that is:

```
module P = PriorityQueue(VertexDistance)
```

We assume here that the module `P` provides a function `create` to construct a new priority queue, a function `add` to add an element, and a function `extract_min` that removes and returns the smallest element for the order defined by the function `compare`.

We now arrive at the code of the algorithm itself. We write it as a function `dijkstra` that takes as arguments a graph `g`, a root `s`, and a function `f`. The function `f` will be applied to elements of `g` as their shortest distance to `s` is determined, the vertex being passed to `f` as first argument, and its distance as second argument. We begin by creating a hash table that will contain the vertices already reached by the traversal.

```
let rec dijkstra f g s =
  let visited = H.create () in
```

We also create a hash table, `distance`, and the priority queue, `queue`. The hash table will contain the distance currently known for each vertex.

[2]Another solution would be to use a priority queue structure where it is possible to *modify* the priority of an element already in the queue. Such structures exist but are complex to implement and, although asymptotically better, do not necessarily bring any practical gains. The solution presented here is a good compromise.

```
    let distance = H.create () in
    let queue = P.create () in
```

It is convenient to introduce a function `add` that will add a vertex to both these data structures at the same time. We begin by applying it to the root `s`.

```
    let add v d = H.replace distance v d; P.add queue (v, d) in
    add s 0.;
```

We will see below why `H.replace` is preferred over `H.add`. We then continue with a loop as long as the queue is not empty.

```
    while not (P.is_empty queue) do
```

We extract the first element of the queue, `u`, together with the distance to the root, `du`. If the vertex `u` is already in `visited`, then we have already determined the distance of `s` to `u`, and there is nothing more to be done.

```
      let (u, du) = P.extract_min queue in
      if not (H.mem visited u) then begin
```

This situation can indeed occur when a first path is found and another, shorter, path is found subsequently. The latter then passes before the former in the priority queue. When the longer path finally comes out of the queue, it should be ignored. If the vertex is not in `visited`, however, that means that we have just found a shortest path from `s` to `u`, which we indicate by adding `u` to the table `visited` and by applying the visitor function `f` to `u` and `du`.

```
        H.add visited u ();
        f u du;
```

Next, we will examine all arcs emerging from `u`. For each arc `u` $\rightarrow$ `v`, the distance to `v` obtained by following the corresponding arc is the sum of the distance `du` and the arc weight, that is, `weight u v`. Several scenarios are possible for the vertex `v`. Either it is the first time that it is reached, or it was already in `distance`. In the latter case, we may have decreased the distance to `v` by passing through `u`. If so, it suffices to use the function `add` to update the tables. We handle each successor `v` in the function `visit` below:

```
        let visit v =
          let d = du +. weight g u v in
          if not (H.mem distance v) || d < H.find distance v then
            add v d
```

It only remains to apply this function to all arcs emerging from `u`, which concludes the body of the `while` loop.

```
        in
        iter_succ visit g u
      end
    done
```

Once we exit the `while` loop, all vertices that may be reached from `s` have been visited. The complete code is given in program 101 above.

Figure 13.5 shows the result of Dijkstra's algorithm on the graph given in the previous example, with 2 as root. The graph is drawn with the final distances obtained for each vertex as exponents. In this example, all vertices have been reached by the traversal. As for the breadth- and depth-first searches, this is not always the case: only vertices for which there is a path from the root will be reached.

Complexity

Let us evaluate the cost of Dijkstra's algorithm in the worst case. The priority queue can contain up to E elements, because the algorithm visits each arc at most once, and each arc can lead to the insertion of an element in the queue. Assuming that the priority queue operations `add` and `extract_min` have logarithmic cost (as is the case for the priority queues described in chapter 6), each operation on the queue will have cost $O(\log E)$, that is, $O(\log N)$, since $E \leq N^2$. The total cost is therefore $O(E \log N)$.

Bellman-Ford Algorithm

We present here a second solution to the shortest path problem, this time without assuming that the weights are non-negative. Omitting this assumption

Program 101 — Shortest path (Dijkstra's algorithm).

```
module VertexDistance = struct
  type t = vertex * float
  let compare (_, d1) (_, d2) = Stdlib.compare d1 d2
end
module P = PriorityQueue(VertexDistance)

let rec dijkstra f g s =
  let visited = H.create () in
  let distance = H.create () in
  let queue = P.create () in
  let add v d = H.replace distance v d; P.add queue (v, d) in
  add s 0.;
  while not (P.is_empty queue) do
    let (u, du) = P.extract_min queue in
    if not (H.mem visited u) then begin
      H.add visited u ();
      f u du;
      let visit v =
        let d = du +. weight g u v in
        if not (H.mem distance v) || d < H.find distance v then
          add v d
      in
      iter_succ visit g u
    end
  done
```

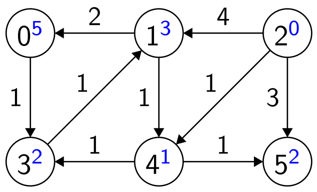


Figure 13.5: Result of Dijkstra's algorithm.

introduces a subtlety. If the graph contains a cycle with strictly negative total weight, and if this cycle can be reached from the root, then none of the vertices that can be reached from the cycle will have a shortest path because we can take this cycle as many times as we wish, reducing the total weight of the path each time. The Bellman-Ford algorithm computes the shortest paths from the root and detects any negative cycle that may be reached from the root.

This algorithm consists in repeating the following operation N times: Consider each arc `u` $\rightarrow$ `v` of the graph and determine if taking it will reduce the distance currently known between the source and `v`. The idea is that a shortest path between `s` and `v` cannot contain more than $N-1$ arcs, without which this path would contain a cycle, which we could remove from the path if it is of non-negative weight. This is why N iterations suffice.

Let us now write the code of the Bellman-Ford algorithm. We begin by defining a function `iter_edge` that traverses all the arcs of a graph `g`.

```
let iter_edge f g =
  iter_vertex (fun u ->
    iter_succ (fun v -> f u v (weight g u v)) g u) g
```

For each arc `u` $\rightarrow$ `v`, the function `iter_edge` applies the function `f` to the vertices `u` and `v`, and the distance of the arc `u` $\rightarrow$ `v` given by the function `weight`.

The function `bellman_ford` takes a graph `g` and a root vertex `s` as arguments. It begins by creating a hash table `h` containing the known distances for each vertex.

```
let bellman_ford g s =
  let h = H.create () in
```

We begin by initializing this table, giving `s` distance 0 and every other vertex an infinite distance.

```
iter_vertex (fun v -> H.add h v max_float) g;
H.add h s 0.;
```

We use here the constant `max_float` to represent an infinite distance. Then we repeat the following operation $N - 1$ times: We examine each arc, using a `for` loop and the function `iter_edge`.

```
for i = 1 to nb_vertex g - 1 do
  iter_edge (fun u v w ->
```

For each arc `u` $\to$ `v`, we compute the distance `d` from the root to `v` that is obtained by taking this arc. If it is smaller than the best distance currently known, we record it.

```
      let d = H.find h u +. w in
      if d < H.find h v then H.replace h v d
```

Once this `for` loop is complete, we redo the operation an N-th and final time. This time, the shortest paths can only be improved by the presence of a negative cycle. We will signal this case by raising an exception.

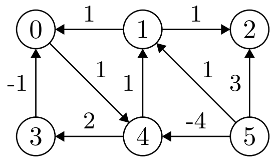
```
done;
iter_edge (fun u v w ->
  if H.find h u +. w < H.find h v then
    raise NegativeCycle
```

This completes the Bellman-Ford algorithm. We can return the table `h` that gives the distance of each vertex to the source. This is what is done in program 102.

We illustrate the Bellman-Ford algorithm with an example. Consider the graph of figure 13.6, for which we seek to determine the distances from vertex 5. The table to the right of the graph gives the distance found for each vertex after each step. Initially, all the vertices are at an infinite distance, except the root 5. At each step, we assume the vertices are traversed in the order of their numbering. We observe, for example, that the distance to vertex 2 initially

Program 102 — The Bellman-Ford algorithm.

```
let iter_edge f g =
  iter_vertex (fun u ->
    iter_succ (fun v -> f u v (weight g u v)) g u) g

exception NegativeCycle

let bellman_ford g s =
  let h = H.create () in
  iter_vertex (fun v -> H.add h v max_float) g;
  H.add h s 0.;
  for i = 1 to nb_vertex g - 1 do
    iter_edge (fun u v w ->
      let d = H.find h u +. w in
      if d < H.find h v then H.replace h v d
    ) g
  done;
  iter_edge (fun u v w ->
    if H.find h u +. w < H.find h v then
      raise NegativeCycle
  ) g;
  h
```



vertex	distance after i steps					
0	∞	∞	2	-3	-3	-3
1	∞	1	-3	-3	-3	-3
2	∞	3	2	-2	-2	-2
3	∞	∞	-2	-2	-2	-2
4	∞	-4	-4	-4	-4	-4
5	0	0	0	0	0	0

Figure 13.6: Illustration of the Bellman-Ford algorithm.

takes the value 3 (the arc $5 \to 2$), then the value 2 (passing through vertex 1), and finally the value -2 (passing through the vertices 4 and 1). After that, it no longer varies. More generally, we observe here that none of the distances is improved during the last two steps of the algorithm. In particular, there is no negative cycle.

Complexity

The cost of the Bellman-Ford algorithm in the worst case is easy to evaluate. We perform exactly N passes, and each pass examines each of the E arcs once and only once. The complexity is therefore $O(EN)$, that is, $O(N^3)$ in the worst case. In practice, however, we can significantly improve performance by observing that many arcs are examined unnecessarily when the distance from the root to the starting vertex has not been modified. Another simple optimization consists in stopping once a pass has not improved any of the distances. The complexity in the worst case remains, nevertheless, the same.

13.4 Minimum spanning tree

We consider here an undirected graph whose arcs are labeled with weights. We also assume that the graph is connected, that is, that each pair of vertices is linked by a path. A *spanning tree* is a subset of the arcs, that does not contain a cycle, and that links all vertices together. It is easy to convince ourselves that such a tree exists. For example, a depth-first search beginning at an arbitrary vertex will determine such a tree: it is formed by all the arcs that led to the discovery of a new vertex.

We seek then to solve the problem of finding a spanning tree whose total weight is minimal. We present here a solution known as *Kruskal's algorithm*. This algorithm consists in traversing the arcs in increasing order of weight. For each arc $u - v$, we determine if a cycle is created by adding the arc to the set of arcs already selected to form the spanning tree. If this is so, we ignore the arc. Otherwise, we select it. Once all the arcs have been examined, the set of selected arcs form a minimum spanning tree (see exercises 13.24 and 13.25).

To implement this algorithm, we need to find an efficient means to determine if selecting a new arc introduces a cycle among the already selected arcs. Luckily, there is a simple solution to this problem, in the form of the structure of disjoint sets studied in chapter 8. Indeed, it suffices to construct an initial structure in which each vertex constitutes a class by itself. For any selected arc $u - v$, we merge the classes of u and v. This materializes the fact that there now exists a path between every pair of vertices in the new class. To test if an arc introduces a cycle, it suffices therefore to test whether its extremities belong to the same class.

Let us now write the code for Kruskal's algorithm, as a function `spanning_tree` that takes a graph as argument and returns a minimum spanning tree in the form of a list of arcs. We take as given a module `UF` for the structure of disjoint sets (see chapter 8), with the following signature:

```
module UF: sig
  val create : vertex list -> t
  val find : t -> vertex -> vertex
  val union : t -> vertex -> vertex -> unit
end
```

We assume there is a function `vertices` that returns the list of vertices of a graph. Such a function is easy to obtain from `iter_vertex`. Similarly, we assume there is a function `edges` that returns the list of all arcs of a graph, each arc being a triple (x, y, w), where w is the weight of the arc $x - y$. Here, too, this function is easy to obtain from `iter_edge` and `weight`.

The code of Kruskal's algorithm begins by creating a *union-find* structure based on the list of vertices of the graph:

```
let spanning_tree g =
  let uf = UF.create (vertices g) in
```

Next, we sort the list of arcs of the graph in increasing order of weight, using the library function `List.sort`. The comparison used here is that of weights, with the usual order on floating-point numbers, that is, `Stdlib.compare`.

```
  let compare (_,_,w1) (_,_,w2) = Stdlib.compare w1 w2 in
  let edges = List.sort compare (edges g) in
```

The list of arcs selected to form the spanning tree is stored in a local reference `st`.

```
  let st = ref [] in
```

We then write a function `cover` that examines an arc `(u,v,w)` and determines if it should be selected. For this, we compare the classes of vertices `u` and `v` using the function `UF.find`. If they are different, we add the arc to the list `st` and merge the two classes, using the function `UF.union`.

```
  let cover ((u, v, w) as e) =
    if UF.find uf u <> UF.find uf v then begin
      UF.union uf u v;
      st := e :: !st
    end
```

Finally, it only remains to examine each arc of the list `edges` using this function, and then return the resulting list.

```
  in
  List.iter cover edges;
  !st
```

The code of Kruskal's algorithm is given in program 103.

We illustrate Kruskal's algorithm with an example. Consider the graph on the top left of figure 13.7. Arcs are considered in increasing order of weight, as presented in the table on the right in the figure. For each arc, we indicate

Program 103 — Kruskal's algorithm.

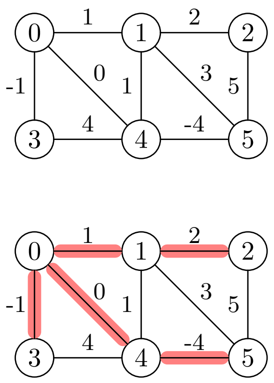
```
let spanning_tree g =
  let uf = UF.create (vertices g) in
  let compare (_,_,w1) (_,_,w2) = Stdlib.compare w1 w2 in
  let edges = List.sort compare (edges g) in
  let st = ref [] in
  let cover ((u, v, w) as e) =
    if UF.find uf u <> UF.find uf v then begin
      UF.union uf u v;
      st := e :: !st
    end
  in
  List.iter cover edges;
  !st
```

if it is selected (column "added") and the resulting partition in the *union-find* structure. The spanning tree obtained in the end is illustrated at the bottom left of the figure. Its total weight is -2. It is important to note that arcs of the same weight may be considered in any order. Thus, we have examined here the arc $0 - 1$ before the arc $1 - 4$, but we could equally have done it the other way around. The spanning tree would have been different, but its total weight would have been the same.

A number of variations and optimizations are possible. Exercise 13.26 thus proposes interrupting the arc traversal once we have selected $N - 1$ arcs. Similarly, exercise 13.27 proposes using a priority queue rather than a sorting algorithm. Be that as it may, the key idea remains the use of the disjoint-set data structure.

Complexity

The cost of Kruskal's algorithm is split between that of sorting the set E of arcs and that of examining each arc. The first cost is $O(E \log E)$ in the worst case, if



weight	arc	added	vertex partition
			$\{0\},\{1\},\{2\},\{3\},\{4\},\{5\}$
-4	$4-5$	yes	$\{0\},\{1\},\{2\},\{3\},\{4,5\}$
-1	$0-3$	yes	$\{0,3\},\{1\},\{2\},\{4,5\}$
0	$0-4$	yes	$\{0,3,4,5\},\{1\},\{2\}$
1	$0-1$	yes	$\{0,1,3,4,5\},\{2\}$
1	$1-4$	no	$\{0,1,3,4,5\},\{2\}$
2	$1-2$	yes	$\{0,1,2,3,4,5\}$
3	$1-5$	no	$\{0,1,2,3,4,5\}$
4	$3-4$	no	$\{0,1,2,3,4,5\}$
5	$2-5$	no	$\{0,1,2,3,4,5\}$

Figure 13.7: Illustration of Kruskal's algorithm.

we use a sorting algorithm of optimal complexity (see chapter 12). The second cost is $O(E)$ if we consider that the operations on the *union-find* structure are constant time (see chapter 8). The complexity of Kruskal's algorithm is therefore $O(E \log E)$.

13.5 Exercises

Breadth-first Search

13.1 Modify the function `iter_bfs` to compute the distance, in number of arcs, between the root and each vertex discovered by the traversal. A simple approach consists in associating this distance with each vertex in the table `visited`. Return this table as the result of the function `iter_bfs`.

13.2 Modify the function `iter_bfs` so as to return a shortest path between the root and the vertex, for each vertex discovered by the traversal. For each vertex discovered, store the vertex that led to it, for instance, in a hash table. We can then recover the path from the vertex discovered to the root "by going backwards."

13.3 Write a function that performs a breadth-first traversal of the nodes of a *tree* by imitating the breadth-first search of a graph.

13.4 Implement a cursor for binary trees (see section *9.3 Cursors*) corresponding to breadth-first search.

13.5 Use the previous exercise to write a program that colors a graph with k colors without using the same color for two vertices linked by an arc. Proceed by *backtracking*, using the persistent nature of the cursor to backtrack. If no color assignment is possible, raise the exception `Not_found`.

Depth-first Search

13.6 Redo exercise 13.2 for depth-first search.

13.7 In many applications of depth-first search, we wish to traverse *all* the vertices of the graph and not merely those that can be reached from a vertex `s`. Using the function `iter_vertex` that traverses all vertices of the graph, modify the function `iter_dfs` so that every vertex of the graph is visited exactly once.

13.8 For an undirected graph, a *connected component* is a maximal set of vertices pairwise linked by a path. Explain how the variant of depth-first search proposed in the previous exercise determines the connected components of an undirected graph. Write the corresponding code.

13.9 Rewrite the function `iter_dfs` using a `while` loop rather than a recursive function. Hint: Use a *stack* containing the vertices from which the depth-first search is to be performed. The code should resemble that of a breadth-first search, with the stack replacing the queue. There is, however, a difference in the treatment of the vertices already visited. Show that the vertices are not necessarily visited in the same order as in the recursive version.

13.10 Depth-first search can be modified to detect the presence of a cycle in the graph. When the function `iter_dfs` comes across an already visited vertex, we do not know *a priori* if we have found a cycle. It could be that the vertex has already been reached by a parallel path. Instead of two states (reached/not reached), the marking of the vertices should therefore be modified to use three: reached/being visited/visited. Write a function `has_cycle` that determines the presence of a cycle in a graph. Modify the function so that it returns the list of vertices of the cycle found, if any.

13.11 Let G be a directed graph that does not contain a cycle. We call such graphs *directed acyclic graphs* (DAG). A *topological sort* of G is a traversal of its vertices compatible with the arcs, that is, where a vertex x is visited before a vertex y if there is an arc $x \to y$. Modify program 100 so that it performs a topological sort, in the form of a function with the following type:

```
topological_sort: (vertex -> unit) -> graph -> unit
```

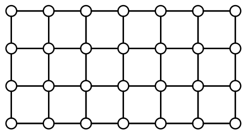
Use a stack to which a vertex `v` is added once the call to `visit v` terminates. Once the depth-first search is completed, apply the function passed as argument to all the elements of the stack.

13.12 For a directed graph, a *strongly connected component* is any maximal set of vertices pairwise linked by a path. Kosaraju-Sharir's algorithm computes the set of strongly connected components of a graph G, in time $O(N+E)$, using two depth-first searches. It proceeds as follows:

- We begin by constructing the transposition G^R of the graph G, that is, the graph having the same vertices as G but with the arcs reversed (see exercise 7.1).
- Next, we traverse the vertices of G^R according to a topological sort (see previous exercise). For each vertex v, we perform a depth-first search in G, starting from v, if v is not already part of a strongly connected component. All the vertices that are reached by this search are then placed in a new component.

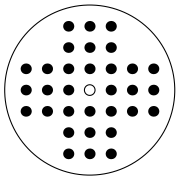
Write a function `scc: graph -> vertex list list` that uses this algorithm to compute the strongly connected components of a graph.

13.13 We can use depth-first search to construct a perfect maze in an $n \times m$ grid. A perfect maze is one in which there is one and only one path between any two cells. We begin by considering the graph in which all cells of the grid are linked to their neighbors:

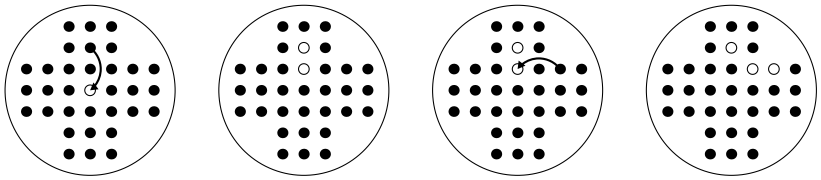


We then perform a depth-first search beginning from an arbitrary vertex (for example, the top left one). When we traverse the successors of a vertex, we do so in random order. Once the traversal is completed, the maze is obtained by declaring that we can pass from one vertex to another if the corresponding arc was taken during the depth-first search. Exercise 8.7 proposes another way of constructing a perfect maze.

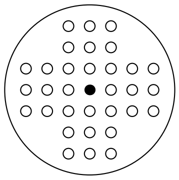
13.14 We consider here the problem of playing peg solitaire. This is a game of patience consisting of 32 pegs on a board with 33 holes. Initially, the pegs occupy all the holes, except the one at the center of the board.



A peg can be removed from the board by having one of the four adjacent pegs jump over it, provided that the hole where the peg is to land is empty. Here are two possible moves starting from the initial position.



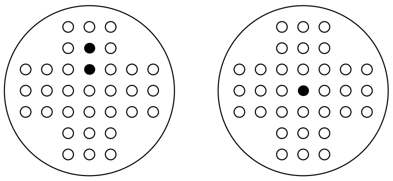
The aim of the game is to arrive at a situation in which there is only one peg remaining, which occupies the central position.



We can easily solve this problem using depth-first search on a graph in which each vertex represents a board configuration.

- A board configuration can be represented by an integer whose $(i + 7j)$-th bit indicates the presence of a peg at position (i, j) for $0 \leq i, j < 7$. Assuming a 64-bit machine, we let `type state = int`. Define two constants, `initial` and `final`, representing respectively the initial and final configurations of the game.

- A move can be represented by two configurations, the first one indicating the two pegs involved (the one that is moved and the one jumped over), and the second indicating the final position of the peg that was moved. For example, the first move above is represented as follows:



  Let `type move = state * state`. Construct a list containing the 88 possible moves (independent of what the configuration may be).

- Write a function `possible_move` to test if a move is possible in a given configuration and a function `move` that performs the move, if one is possible. Derive from it a function `iter_succ` that traverses all possible moves in a given configuration.

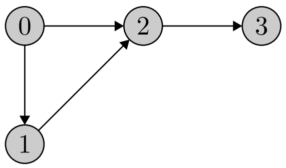
- Modify the function `iter_dfs` of program 100 to interrupt the traversal once the configuration `final` is reached and return the path found, as in exercise 13.6 above.

- Finally, write a function `print` that displays a configuration on the screen, and use it to display the solution found, in the form of 32 successive configurations.

Note: The problem of peg solitaire is one of the problems discussed in the book by Cousineau and Mauny *The Functional Approach to Programming* [9, page 216].

13.15 In the preceding exercise, depth-first search was a good solution because all the paths were of the same length. When this is not the case, and if we seek a path of minimal length, it is necessary to turn to breadth-first search instead. However, this may require a lot of memory. In this case, *Iterative deepening search* (IDS) offers a solution that has low memory requirements, but which is still able to find a path of minimal length. The idea is to execute successive depth-first searches, limited to maximal depths that are larger and larger.

- Explain why we can no longer memoize the vertices already visited during the depth-first searches. Consider the following graph and assume that we traverse it beginning at vertex 0.



- Implement iterative deepening search as a function `ids: (vertex -> bool) -> graph -> ve` that stops once it reaches a vertex that satisfies the function passed as argument.

- In case of the traversal of a complete binary tree starting at the root, show that iterative deepening search is no costlier than breadth-first and depth-first searches.

Shortest Paths

13.16 Generalize the function `dijkstra` for distances of an arbitrary type, for example in the form of a functor.

13.17 Redo exercise 13.2 for Dijkstra's algorithm. Note: When a path is improved, the corresponding table must be updated.

13.18 Redo the steps of the Bellman-Ford algorithm, illustrated in figure 13.6, assuming that the weight of the arc $0 \to 4$ is -3.

13.19 Warshall's algorithm (exercice 7.6) can be easily adapted to compute *all* shortest paths in a graph. This is the Floyd-Warshall algorithm. Let $V = \{0, \dots, N-1\}$. Initially, the distance $d_{i,j}$ between vertices i and j equals the weight of the corresponding arc, if there is one, and is otherwise ∞. The algorithm then computes the minimal distance $d_{i,j}$ from i to j in $O(N^3)$ time as follows:

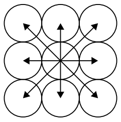
```
for $k$ from 0 to $N-1$
  for $i$ from 0 to $N-1$
    for $j$ from 0 to $N-1$
      $d_{i,j} \leftarrow \min(d_{i,j}, d_{i,k} + d_{k,j})$
```

Write a function that implements this algorithm. How can we treat the general case, when vertices are not necessarily the integers $0, \dots, N-1$?

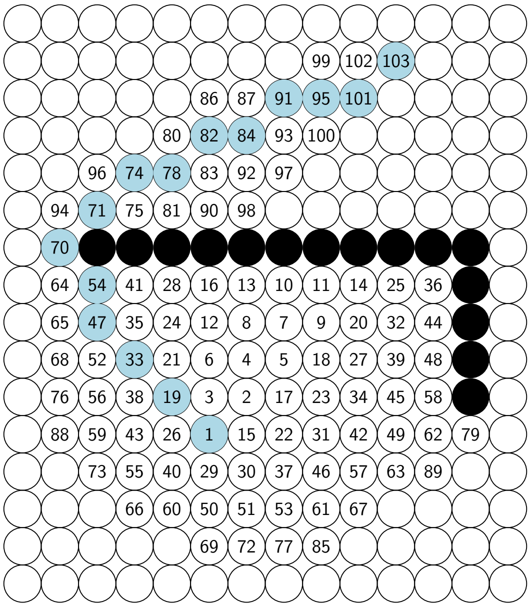
13.20 Although Dijkstra's algorithm allows us to find a path of minimal cost, it may visit a number of vertices in the graph unnecessarily. Suppose we use Dijkstra's algorithm to find the shortest path (in terms of distance) between two cities on a road network. The algorithm will explore the network in larger and larger concentric circles centered around the starting city, until it reaches the destination city. If we take the example of the French road network, and if we seek the shortest path between Paris and Nice, it is likely that almost all the cities of France will have been visited before Nice is reached.

The A* algorithm allows us to solve this problem through a finer control of the order in which vertices are visited. This control takes the form of a heuristic that estimates the distance remaining between a vertex and the destination. The A* algorithm is identical to Dijkstra's algorithm, except that the queue no longer uses distance to the source as priority, but rather the sum of this distance and the value given by the heuristic. If the heuristic is *admissible*, that is, if it never overestimates the remaining distance, then the A* algorithm will find the shortest path. An admissible heuristic for the road-network example is the distance as a crow flies. If the heuristic always returns zero, we recover Dijkstra's algorithm.

Let us consider the example of an infinite grid in which each point is linked to its eight direct neighbors.



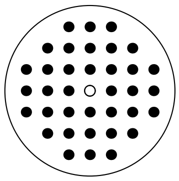
The cost here is the Euclidean distance, that is, four neighbors are at distance 1, and the other four at distance $\sqrt{2}$. We can therefore use as heuristic the Euclidean distance between the point in question (x, y) and the destination point (x_d, y_d), that is, $\sqrt{(x - x_d)^2 + (y - y_d)^2}$.



The figure above shows the result of the A* algorithm finding a path from point $(0, 0)$, labelled 1, to point $(5, 10)$, labelled 103, the black nodes being excluded from the grid. As we see, only 103 nodes were visited. Dijkstra's algorithm would also have found the shortest path, but the total number of visited nodes would have been much larger (over 800).

Assume we are given a function `heuristic: graph -> vertex -> vertex -> float`, and implement the A* algorithm by modifying the code of program 101. Apply this to the preceding example and verify that we do indeed arrive at the result of the figure above.

13.21 "French solitaire" differs from that of exercise 13.14, called "English solitaire," by virtue of four additional pegs present in positions $(1,1)$, $(5,1)$, $(1,5)$, and $(5,5)$.



French solitaire is, surprisingly, much harder to solve than English solitaire. Furthermore, there exist combinations of starting/final configurations that do not admit of a solution, for example, the configuration in which the central peg is removed at the start and the last peg must be placed at the center[3]. We will accordingly loosen the conditions on the positions of the initial and final pegs.

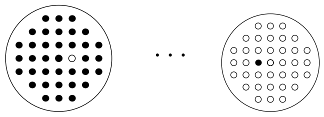
One approach to solving French solitaire consists in using the A* algorithm presented in the preceding exercise. Modify the code of exercise 13.14 as follows:

- Add the 8 new moves linked to the 4 new pegs to the the list of all possible moves.
- Write a heuristic function that evaluates how "compactly" pegs are laid out on the board. Given the representation of the grid as an integer, we may compute a rough approximation by counting the number of bit

[3]It is only recently that we have rediscovered how French solitaire was played in the 18th century: Initially, *all* cells contain a peg, and the player removes a peg of her choice. This peg can then be placed back on the board at any moment during the game. With these rules, there is then a solution in which the central peg is taken out at the start and the final peg is placed in the center. Here, however, we maintain the same rules as in English solitaire.

inversions, that is, the number of times that we pass from a 1 bit to a 0 bit, and vice versa.

- Finally, replace depth-first search with the A*algorithm. Here is an example of initial and final configurations for which there exists a solution.



As in exercise 13.14, present the solution by displaying all successive configurations of the board.

13.22 It is possible to combine the ideas of iterative deepening search (exercise 13.15) and the A* algorithm (exercise 13.20), to obtain an algorithm known as IDA* (*iterative deepening A**). The idea is as follows: At each iteration, we execute a depth-first search of all the states where the heuristic does not exceed a given threshold, without seeking to memoize already visited states. Once the search is over, we take as the new threshold the smallest value of the heuristic obtained during the previous search. In other words, each iteration pushes the threshold of the heuristic a little further until a path is found or no vertex takes a value greater than the threshold. To initiate the process, it suffices to take as initial threshold the heuristic for the root vertex. As with the A*algorithm, an admissible heuristic, that is, one that never overestimates the remaining distance, guarantees that IDA* will find a shortest path. The IDA* algorithm was discovered by Korf [17] in 1985.

Assume we are given a function `heuristic: graph -> vertex -> vertex -> float`, and modify the code of exercise 13.15 to implement the IDA*algorithm. The following exercise proposes an application of this algorithm.

13.23 A good application of the IDA* algorithm proposed in the preceding exercise is the *game of fifteen*. The game consists in a 4×4 grid containing 15 tiles, numbered from 1 to 15. The sixteenth position is empty, and we may place in it one of the adjacent tiles. Initially, the tiles are scrambled, for example as follows:

14	1	9	6
4	8	12	5
7	2	3	▨
10	11	13	15

The aim is to place the tiles in increasing order[4].

▨	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

To solve the game, we will use the following heuristic: For each tile, we compute the distance that separates it from its final position, in terms of number of rows and columns. This is called the Manhattan distance. In the preceding example, tile 7 is initially at distance 4 from its final position because it has to be moved up by one row, and right by three columns. For a given configuration, we then take the sum of the Manhattan distances of the fifteen tiles, which is a lower bound on the number of moves that remain to be done. This is an admissible heuristic, and we will thus be able to find a smallest solution. Solve the game using the IDA* algorithm. The example of the game of fifteen is presented in the article by Korf that introduced the IDA* algorithm.

Minimum Spanning Tree

13.24 Show that the set of arcs returned by Kruskal's algorithm does indeed form a tree.

13.25 Show that the result of Kruskal's algorithm is indeed a minimum spanning tree.

13.26 Improve the code of program 103 so that it stops as soon as $N-1$ arcs have been added to the spanning tree.

[4]Not all initial configurations admit a solution.

13.27 Reimplement program 103 using a priority queue rather than a sorting algorithm. How does it differ from the previous exercise?

Bibliography

[1] Stephen Adams. Functional pearls: Efficient sets – a balancing act. *Journal of Functional Programming*, 3(4):553–561, October 1993. Expanded version available as Technical Report CSTR 92-10, University of Southampton.

[2] G. M. Adel'son-Vel'skiĭ and E. M. Landis. *An algorithm for the organization of information*. *Soviet Mathematics–Doklady*, 3(5):1259–1263, September 1962.

[3] Alfred V. Aho, John E. Hopcroft, and Jeffrey Ullman. *Data Structures and Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[4] Henry G. Baker. Shallow binding makes functional arrays fast. *SIGPLAN Not.*, 26(8):145–147, 1991.

[5] Hans-Juergen Boehm, Russell R. Atkinson, and Michael F. Plass. Ropes: An alternative to strings. *Software - Practice and Experience*, 25(12):1315–1330, 1995.

[6] John Conway. Doomsday rule. http://en.wikipedia.org/wiki/Doomsday_algorithm.

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition.* The MIT Press, September 2001.

[8] Guy Cousineau and Michel Mauny. *Approche fonctionnelle de la programmation.* Ediscience International, 1995.

[9] Guy Cousineau and Michel Mauny. *The Functional Approach to Programming.* Cambridge University Press, 1998. Traduction anglaise de [8].

[10] Jeremy Gibbons and Oege de Moor, editors. *The Fun of Programming.* Cornerstones in Computing. Palgrave, 2003.

[11] Jr. Henry G. Baker. Shallow binding in Lisp 1.5. *Commun. ACM*, 21(7):565–569, 1978.

[12] Gérard Huet. The Zipper. *Journal of Functional Programming*, 7(5):549–554, September 1997.

[13] D. E. Knuth. *The Art of Computer Programming. Volume 1: Fundamental Algorithms.* Addison-Wesley, 1968.

[14] D. E. Knuth. *The Art of Computer Programming. Volume 2: Seminumerical Algorithms.* Addison-Wesley, 1969.

[15] D. E. Knuth. *The Art of Computer Programming. Volume 3: Sorting and Searching.* Addison-Wesley, 1973.

[16] Donald E. Knuth. *The Art of Computer Programming, volume 4A: Combinatorial Algorithms, Part 1.* Addison-Wesley Professional, 1st edition, 2011.

[17] Richard E. Korf. Iterative-deepening-A*: An optimal admissible tree search. In *IJCAI*, pages 1034–1036, 1985.

[18] Donald R. Morrison. *PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric*. *J. ACM*, 15(4):514–534, 1968.

[19] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[20] Chris Okasaki and Andrew Gill. *Fast Mergeable Integer Maps*. In *Workshop on ML*, pages 77–86, September 1998.

[21] Robert Sedgewick and Kevin Wayne. *Algorithms, 4th Edition*. Addison-Wesley, 2011.

[22] Daniel Dominic Sleator and Robert Endre Tarjan. *Self Adjusting Heaps*. *SIAM J. Comput.*, 15(1):52–69, February 1986.

[23] Robert Endre Tarjan. *Efficiency of a Good But Not Linear Set Union Algorithm*. *J. ACM*, 22(2):215–225, 1975.

[24] Henry S. Warren. *Hacker's Delight*. Addison-Wesley Professional, July 2002.

Index