

# Objective CAML for Scientists

Chapter 1  
© Flying Frog Consultancy Ltd.  
By Dr. Jon D. Harrop

Buy the whole book at <http://www.ffconsultancy.com>

# Chapter 1

## Introduction

For the first time in history, and thanks to the exponential growth rate of computing power, an increasing number of scientists are finding that more time is spent creating, rather than executing, working programs. Indeed, much effort is spent writing small programs to automate otherwise tedious forms of analysis. In the future, this imbalance will doubtless be addressed by the adoption and teaching of more efficient programming techniques at the cost of less efficient programs. An important step in this direction is the use of higher-level programming languages, such as OCaml, in place of more conventional languages for scientific programming such as Fortran, C, C++ and Java.

In this chapter, we shall begin by laying down some guidelines for good programming which are applicable in any language before briefly reviewing the history of the OCaml language and outlining some of the features of the language which enforce some of these guidelines and other features which allow the remaining guidelines to be met. As we shall see, these aspects of the design of OCaml greatly improve reliability and development speed. Coupled with the fact that a freely available, efficient compiler already exists for this language, no wonder OCaml is already being adopted by scientists of all disciplines.

### 1.1 Good programming style

Regardless of the choice of language, some simple, generic guidelines can be productively adhered to. We shall now examine the most relevant such guidelines in the context of scientific computing:

**Avoid premature optimisation** Programs should be written correctly first and optimised last.

**Structure programs** Complicated programs should be hierarchically decomposed into progressively smaller, constituent components.

**Factor programs** Complicated or common operations should be factored out into separate functions.

**Explicit interfaces** Interfaces should always be made as explicit as possible.

**Avoid magic numbers** Numeric constants should be defined once and referred back to, rather than explicitly “hard-coding” their value multiple times at different places in a program.

We shall now examine some of the ways OCaml can help in enforcing these guidelines and how the OCaml compiler can exploit well-designed code.

## 1.2 A Brief History of OCaml

The Meta-Language (**ML**) was originally developed at Edinburgh University in the 1970’s as a language designed to efficiently represent other languages. The language was pioneered by Robin Milner for the *Logic of Computable Functions* (**LCF**) theorem prover. The original ML, and its derivatives, were designed to stretch theoretical computer science to the limit, yielding remarkably robust and concise programming languages which can also be very efficient.

The *Categorical Abstract Machine Language* (**CAML**) was the acronym originally used to describe what is now known as the Caml family of languages. Gérard Huet designed and implemented Caml at *Institut National de Recherche en Informatique et en Automatique* (**INRIA**) in France, until 1994. Since then, development has continued as part of *projet Cristal*, now led by Xavier Leroy.

*Objective Caml* (**OCaml**<sup>1</sup>) is the current flagship language of projet Cristal. The Cristal group have produced freely available tools for this language. Most notably, an interpreter which runs OCaml code in a *virtual machine* (**VM**) and two compilers, one which compiles OCaml to a machine independent byte-code which can then be executed by a byte-code interpreter and another which compiles OCaml directly to native code. At the time of writing, the native-code compiler is capable of producing code for Alpha, Sparc, x86, MIPS, HPPA, PowerPC, ARM, ia64 and x86-64 CPUs and the associated run-time environment has been ported to the Linux, Windows, MacOS X, BSD, Solaris, HPUNIX, IRIX and Tru64 operating systems.

## 1.3 Benefits of OCaml

Before delving into the syntax of the language itself, we shall list the main, advantageous features offered by the OCaml language:

**Safety** OCaml programs are thoroughly checked at compile-time such that they are proven to be entirely safe to run, e.g. a compiled OCaml program cannot segfault.

**Functional** Functions may be nested, passed as arguments to other functions and stored in data structures as values.

**Strongly typed** The types of all values are checked during compilation to ensure that they are well defined and validly used.

---

<sup>1</sup>Pronounced oh-camel.

**Statically typed** Any typing errors in a program are picked up at compile-time by the compiler, instead of at run-time as in many other languages.

**Type inference** The types of values are automatically inferred during compilation by the context in which they occur. Therefore, the types of variables and functions in OCaml code does not need to be specified explicitly, dramatically reducing source code size.

**Polymorphism** In cases where any of several different types may be valid, any such type can be used. This greatly simplifies the writing of generic, reusable code.

**Pattern matching** Values, particularly the contents of data structures, can be matched against arbitrarily-complicated patterns in order to determine the appropriate action.

**Modules** Programs can be structured by grouping their data structures and related functions into modules.

**Objects** Data structures and related functions can also be grouped into objects (object-oriented programming).

**Separate compilation** Source files can be compiled separately into object files which are then linked together to form an executable. When linking, object files are automatically type checked and optimized before the final executable is created.

## 1.4 Running OCaml programs

OCaml provides three different ways to execute code. We shall now examine each of these three approaches, explaining how code can be executed using them and noting their relative advantages and disadvantages.

### 1.4.1 Top-level

The OCaml *top-level* interactively interprets OCaml code and is started by running the program `ocaml`:

```
$ ocaml
      Objective Caml version 3.08.0

#
```

OCaml code may then be entered at this `#` prompt, the end of which is delimited by `;;`. For example, the following calculates  $1 + 3 = 4$ :

```
# 1 + 3;;
- : int = 4
#
```

The top-level will also print the type of the result as well as its value (when the result has a value). For example, the following defines a variable called `sqr` which is a function:

```
# let sqr x = x *. x;;
val sqr : float -> float = <fun>
#
```

This response indicates that a function called `sqr` has been defined which accepts a `float` and returns a `float`. In general, the response of the top-level is either of the form:

```
- : type = value
```

or consisting of one or more descriptions of the form:

```
val name : type = value
```

where `-` indicates that a value has been returned but was not bound to a variable name, `name` is the name of a variable which has been bound, `type` is the type of the value and `value` is the value itself. Values are described explicitly for many data structures, such as `4` in the former case, but several other kinds of value are simply classified, such as `<fun>` to indicate that the value is a function in the latter case<sup>2</sup>.

Programs entered into the top-level execute almost as quickly as byte-code compiled programs (which is often quite a bit slower than native-code compiled programs). However, the interactivity of the top-level makes testing the validity of code segments much easier.

In the remainder of this book, we shall write numerous code snippets in this style, as if they had been entered into the top-level.

### 1.4.2 Byte-code compilation

When stored in a plain text file with the suffix “.ml”, an OCaml program can be compiled to a machine independent byte-code using the `ocamlc` compiler. For example, for a file “test.ml” containing the code:

```
let _ = print_endline "Hello world!"
```

This file may be compiled at the Unix shell `$` prompt into a byte-code executable called “test”:

```
$ ocamlc test.ml -o test
```

and then executed:

```
$ ./test
Hello world!
```

In this case, the result was to print the string “Hello world!” onto the screen. Byte-code compilation is an adequate way to execute OCaml programs which do not perform intensive computations. If the time taken to execute a program needs to be reduced then native-code compilation can be used instead.

---

<sup>2</sup>Abstract types are denoted `<abstr>`, as we shall see in chapter 2.

### 1.4.3 Native-code compilation

The “test.ml” program could equivalently have been compiled to native code, creating a stand-alone, native-code executable called “test”, using:

```
$ ocamlpt test.ml -o test
```

The resulting executable runs in exactly the same way:

```
$ ./test
Hello world!
```

Programs compiled to native code, particularly in the context of numerically intensive programs, can be considerably faster to execute.

## 1.5 OCaml syntax

Before we consider the features offered by OCaml, a brief overview of the syntax of the language is instructive, so that we can provide actual code examples later. Other books give more systematic, thorough and formal introductions to the whole of the OCaml language.

### 1.5.1 Language overview

In this section we shall evolve the notions of values, types, variables, functions, simple containers (lists and arrays) and program flow control. These notions will then be used to introduce more advanced features in the later sections of this chapter.

When presented with a block of code, even the most seasoned and fluent programmer will not be able to infer the purpose of the code. Consequently, programs should contain additional descriptions written in plain English, known as *comments*. In OCaml, comments are enclosed between `(*` and `*)`. They may be nested, i.e. `(* (* ... *) *)` is a valid comment. Comments are treated as whitespace, i.e. `a(* ... *)b` is understood to mean `a b` rather than `ab`.

Just as numbers can be defined to be members of sets such as integer ( $\in \mathbb{Z}$ ), real ( $\in \mathbb{R}$ ), complex ( $\in \mathbb{C}$ ) and so on, so *values* in programs are also defined to be members of sets. These sets are known as *types*.

#### 1.5.1.1 Types

Fundamentally, languages provide basic types and, often, allow more sophisticated types to be defined in terms of the basic types. OCaml provides a number of built-in types: `unit`, `int`, `float`, `char`, `string` and `bool`. We shall examine these built-in types before discussing the compound *tuple*, *record* and *variant* types.

Only one value is of type `unit` and this value is written `()` and, therefore, conveys no information. This is used to implement functions which require no input or expressions which return

no value. For example, a new line can be printed by calling the `print_newline` function as `print_newline ()`. This function requires no input, so it accepts a single argument `()` of type `unit`, and returns the value `()` of type `unit`.

Integers are written `-2`, `-1`, `0`, `1` and `2`. Floating-point numbers are written `-2.`, `-1.`, `-0.5`, `0.`, `0.5`, `1.` and `2.`. For example:

```
# 3;;
- : int = 3
# 5.;;
- : float = 5.
```

Arithmetic can be performed using the conventional `+`, `-`, `*`, `/`, `mod` binary infix<sup>3</sup> operators over the integers<sup>4</sup>. For example, the following expression is evaluated according to usual mathematical convention regarding operator precedence, with multiplication taking precedence over addition:

```
# 1 * 2 + 2 * 3;;
- : int = 8
```

The floating-point infix functions have slightly different names, suffixed by a full-stop: `+. .`, `-. .`, `*. .`, `/. .` as well as `**` (raise to the power). For example, the following calculates  $(1 \times 2) + (2 \times 3) = 8$ :

```
# 1. *. 2. +. 2. *. 3.;;
- : float = 8.
```

The distinct names of the operators for different types arises as the most elegant solution to allowing the unambiguous inference of types in the presence of different forms of arithmetic. The definition of new operators is discussed later, in section A.3. In order to perform arithmetic using mixed types, functions such as `float_of_int` can be used to convert between types.

Unlike other languages, OCaml is phenomenally pedantic about types. For example, the following fails because `*` denotes the multiplication of a pair of integers and cannot, therefore, be applied to a value of type `float`:

```
# 2 * 2.;;
This expression has type float but is here used with type int
```

Note that the OCaml top-level underlines the erroneous portion of the code.

Explicitly converting the value of type `float` to a value of type `int` using the built-in function `int_of_float` results in a valid expression which the top-level will execute:

```
# 2 * (int_of_float 2.);;
- : int = 4
```

---

<sup>3</sup>An infix function is a function which is called with its name and arguments in a non-standard order. For example, the arguments  $i$  and  $j$  of the conventional addition operator  $+$  appear on either side  $i + j$ .

<sup>4</sup>As well as bit-wise binary infix operators `lsl`, `lsr`, `asl`, `asr`, `land`, `lor` and `lxor` described in the manual.

In general, arithmetic is typically performed using a single number representation (e.g. either `int` or `float`) and conversions between representations are, therefore, comparatively rare.

Single characters (of type `char`) are written in single quotes, e.g. `'a'`, which may also be written using a 3-digit decimal code, e.g. `'\097'`.

Strings are written in double quotes, e.g. `"Hello World!"`. Characters in a string of length  $n$  may be extracted using the notation `s.[i]` for  $i \in \{0 \dots n - 1\}$ . For example, the fifth character in this string is “o”:

```
# "Hello world!".[4];;
- : char = 'o'
```

The character at index `i` in a string `s` may be set to `c` using the notation `s.[i] <- c`.

A pair of strings may be concatenated using the `^` operator<sup>5</sup>:

```
# "Hello " ^ "world!";;
- : string = "Hello world!"
```

Booleans are either `true` or `false`. Booleans are created by the usual comparison functions `=`, `<>` (not equal to), `<`, `>`, `<=`, `>=`. These functions are polymorphic, meaning they may be applied to pairs of values of the same type for any type<sup>6</sup>. The usual, short-circuit-evaluated<sup>7</sup> logical comparisons `&&` and `||` are also present. For example, the following expression tests that one is less than three and 2.5 is less than 2.7:

```
# 1 < 3 && 2.5 < 2.7;;
- : bool = true
```

Values may be assigned, or *bound*, to names. As OCaml is a functional language, these values may be expressions mapping values to values — functions. We shall now examine the binding of values and expressions to variable and function names.

### 1.5.1.2 Variables and functions

Variables and functions are both defined using the `let` construct. For example, the following defines a variable called `a` to have the value 2:

```
# let a = 2;;
val a : int = 2
```

Note that the language automatically infers types. In this case, `a` has been inferred to be of type `int`.

Definitions using `let` can be defined locally using the syntax:

<sup>5</sup>A list of strings may be concatenated more efficiently than repeated application of the `^` operator by using the `String.concat` function.

<sup>6</sup>Any attempt to evaluate a comparison function over a value which has the type of a function raises an `Invalid_argument` exception at run-time.

<sup>7</sup>Short-circuit evaluation refers to the premature escaping of a sequence of operations (in this case, boolean comparisons). For example, the expression `false && expr` need not evaluate `expr` as the result of the whole expression is necessarily `false` due to the preceding `false`.

```
let name = expr1 in expr2
```

This evaluates *expr1* and binds the result to the variable *name* before evaluating *expr2*. For example, the following evaluates  $a^2$  in the context  $a = 3$ , giving 9:

```
# let a = 3 in a * a;;
- : int = 9
```

Note that the value 3 bound to the variable *a* in this example was local to the expression *a \* a* and, therefore, the global definition of *a* is still 2:

```
# a;;
- : int = 2
```

More recent definitions shadow previous definitions. For example, the following supersedes the definition  $a = 2$  with  $a = a \times a$  in order to calculate  $2 \times 2 \times 2 \times 2 = 16$ :

```
# let a = 2 in
  let a = a * a in
    a * a;;
- : int = 16
```

As OCaml is a functional language, values can be functions and variables can be bound to them in exactly the same way as we have just seen. Specifically, function definitions append a list of arguments between the name of the function and the = in the *let* construct. For example, a function called *sqr* which accepts an argument called *x* and returns  $x * x$  may be defined as:

```
# let sqr x = x * x;;
val sqr : int -> int = <fun>
```

In this case, the use of the integer multiply *\** results in OCaml correctly inferring the type of *sqr* to be *int -> int*, i.e. the *sqr* function accepts a value of type *int* and returns a value of type *int*.

The function *sqr* may then be applied to an integer as:

```
# sqr 5;;
- : int = 25
```

Typically, more sophisticated computations require the use of more complicated types. We shall now examine the three simplest ways by which more complicated types may be constructed.

### 1.5.1.3 Tuples, records and variants

Tuples are the simplest form of compound types, containing a fixed number of values which may be of different types. The type of a tuple is written analogously to conventional set-theoretic style, using `*` to denote the cartesian product between the sets of possible values for each type. For example, a tuple of three integers, conventionally denoted by the triple  $(i, j, k) \in \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$ , can be represented by values `(i, j, k)` of the type `int * int * int`. When written, tuple values are comma-separated and enclosed in parentheses. For example, the following tuple contains three different values of type `int`:

```
# (1, 2, 3);;
- : int * int * int = (1, 2, 3)
```

A tuple containing  $n$  values is described as an  $n$ -tuple, e.g. the tuple `(1, 2, 3)` is a 3-tuple.

Records are essentially tuples with named components, known as *fields*. Records and, in particular, the names of their fields must be defined using a `type` construct before they can be used. When written, record fields are written `name : type` where `name` is the name of the field (which must start with a lower-case letter) and `type` is the type of values in that field, and are semicolon-separated and enclosed in curly braces. For example, a record containing the  $x$  and  $y$  components of a 2D vector could be defined as:

```
# type vec2 = { x:float; y:float };;
type vec2 = { x:float; y:float }
```

A value of this type representing the zero vector can then be defined using:

```
# let zero = { x=0.; y=0. };;
val zero : vec2 = {x = 0.; y = 0.}
```

Note that the use of a record with fields `x` and `y` allowed OCaml to infer the type of `zero` as `vec2`.

Whereas the tuples are order-dependent, i.e.  $(1, 2) \neq (2, 1)$ , the named fields of a record may appear in any order, i.e.  $\{x = 1, y = 2\} \equiv \{y = 2, x = 1\}$ . Thus, we could, equivalently, have provided the `x` and `y` fields in reverse order:

```
# let zero = { y=0.; x=0. };;
val zero : vec2 = {x = 0.; y = 0.}
```

The fields in this record can be extracted individually using the notation `record.field` where `record` is the name of the record and `field` is the name of the field within that record. For example, the `x` field in the variable `zero` is 0:

```
# zero.x;;
- : float = 0.
```

Also, a shorthand `with` notation exists for the creation of a new record from an existing record with a single field replaced. This is particularly useful when records contain many fields. For example, the record `{x=1., y=0.}` may be obtained by replacing the field `x` in the variable `zero` with 1:

```
# let x_axis = { zero with x=1. };;
val x_axis : vec2 = {x = 1.; y = 0.}
```

Although OCaml is a functional language, OCaml does support imperative programming. Fundamentally, record fields can be marked as mutable, in which case their value may be changed. For example, the type of a mutable, two-dimensional vector called `vec2` may be defined as:

```
# type vec2 = { mutable x:float; mutable y:float };;
type vec2 = { mutable x : float; mutable y : float; }
```

A value `r` of this type may be defined:

```
# let r = { x=1.; y=2. };;
val r : vec2 = {x = 1.; y = 2.}
```

The  $x$ -coordinate of the vector `r` may be altered in-place using an imperative style:

```
# r.x <- 3.;;
- : unit = ()
```

The side-effect of this expression has mutated the value of the variable `r`, the  $x$ -coordinate of which is now 3 instead of 1:

```
# r;;
- : vec = {x = 3.; y = 2.}
```

However, a record with a single, mutable field can often be useful. This data structure, called a *reference*, is already provided by the type `ref`. For example, the following defines a variable named `a` which is a reference to the integer 2:

```
# let a = ref 2;;
val a : int ref = {contents = 2}
```

The type of `a` is then `int ref`. The value referred to by `a` may be obtained using `!a`:

```
# !a;;
- : int = 2
```

The value of `a` may be set using `:=`:

```
# a := 3;
- : unit = ()
# a;;
- : int ref = {contents = 3}
```

In the case of references to integers, two additional functions are provided, `incr` and `decr`, which increment and decrement references to integers, respectively:

```
# incr a;;
- : unit = ()
# a;;
val a : int ref = {contents = 4}
```

The types of values stored in tuples and records are defined at compile-time. OCaml completely verifies the correct use of these types at compile-time. However, this is too restrictive in many circumstances. These requirements can be slightly relaxed by allowing a type to be defined which can acquire one of several possible types at run-time. These are known as *variant types*. OCaml still verifies the correct use of variant types as far as is theoretically possible.

Variant types are defined using the `type` construct with the possible constituent types referred to by *constructors* (the names of which must begin with upper-case letters) separated by the `|` character. For example, a variant type named `button` which may adopt the values `On` or `Off` may be written:

```
# type button = On | Off;;
type button = On | Off
```

The constructors `On` and `Off` may then be used as values of type `button`:

```
# On;;
- : button = On
# Off;;
- : button = Off
```

In this case, the constructors `On` and `Off` convey no information in themselves (i.e. like the type `unit`, `On` and `Off` do not carry data) but the choice of `On` or `Off` does convey information. Note that both expressions were correctly inferred to have results of type `button`.

More usefully, constructors may take arguments, allowing them to convey information by carrying data. The arguments are defined using `of` and are written in the same form as that of a tuple. For example, a replacement `button` type which provides an `On` constructor accepting two arguments may be written:

```
# type button = On of int * string | Off;;
type button = On of int * string | Off
```

The `On` constructor may then be used to create values of type `button` by appending the argument in the style of a tuple:

```
# On (1, "mine");;
- : button = On (1, "mine")
# On (2, "hers");;
- : button = On (2, "hers")
# Off;;
- : button = Off
```

Types can also be defined recursively, which is very useful when defining more sophisticated data structures, such as trees. For example, a binary tree contains either zero or two binary trees and can be defined as:

```
# type binary_tree = Leaf | Node of binary_tree * binary_tree;;
type binary_tree = Leaf | Node of binary_tree * binary_tree
```

A value of type `binary_tree` may be written in terms of these constructors:

```
# Node (Node (Leaf, Leaf), Leaf);;
- : binary_tree = Node (Node (Leaf, Leaf), Leaf)
```

Of course, we could also place data in the nodes to make a more useful data structure. This line of thinking will be pursued in chapter 3. In the mean time, let us consider two special data structures which have notations built into the language.

#### 1.5.1.4 Lists and arrays

Lists are written `[a; b; c]` and arrays are written `[|a; b; c|]`. As we shall see in chapter 3, lists and arrays have different merits.

The types of lists and arrays of integers, for example, are written `int list` and `int array`, respectively:

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

In the case of lists, the infix cons operator `::` provides a simple way to prepend an element to the front of a list. For example, prepending `1` onto the list `[2; 3]` gives the list `[1; 2; 3]`:

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

In the case of arrays, the notation `array.(i)` may be used to extract the  $i + 1^{\text{th}}$  element. For example, `[|1; 2; 3|].(1)` gives the second element `2`:

```
# [|1; 2; 3|].(1);;
- : int = 2
```

Also, a short-hand notation can be used to represent lists or arrays of tuples by omitting the parentheses. For example, `[(a, b); (c, d)]` may be written `[a, b; c, d]`:

```
# [1, 2; 3, 4];;
- : (int * int) list = [(1, 2); (3, 4)]
# [|1, 2; 3, 4|];;
- : (int * int) array = [|1, 2; 3, 4|]
```

The use and properties of lists, arrays and several other data structures will be discussed in chapter 3. In the mean time, we shall examine programming constructs which allow more interesting computations to be performed.

### 1.5.1.5 If

Like many other programming languages, OCaml provides an `if` construct which allows a boolean “predicate” expression to determine which of two expressions is evaluated and returned, as well as a special `if` construct which optionally evaluates an expression of type `unit`:

```
if expr1 then expr2
if expr1 then expr2 else expr3
```

In both cases, *expr1* must evaluate to a value of type `bool`. In the former case, *expr2* is expected to evaluate to the value of type `unit`. In the latter case, both *expr2* and *expr3* must evaluate to values of the same type.

The former evaluates the boolean expression *expr1* and, only if the result is `true`, evaluates the expression *expr2* which is expected to return the value of type `unit`.

The latter similarly evaluates *expr1* but returning the result of either *expr2*, if *expr1* evaluated to `true`, or of *expr3* otherwise.

For example, the following function prints “Less than three” if the given argument is less than three:

```
# let f x = if x < 3 then print_endline "Less than three";;
val f : int -> unit = <fun>
# f 1;;
Less than three
- : unit = ()
# f 5;;
- : unit = ()
```

The following function returns the string “Less” if the argument is less than 3 and “Greater” otherwise:

```
# let f x = if x < 3 then "Less" else "Greater";;
val f : int -> string = <fun>
# f 1;;
- : string = "Less"
# f 5;;
- : string = "Greater"
```

The parts of the language we have covered can already be used to write some interesting programs. However, attention should be paid to the way in which programs are constructed from these parts.

### 1.5.1.6 Program composition

As we have seen, program segments may be written in the top-level which replies by reciting the automatically inferred types and executing expressions. However, the `;;` used to force the top-level into producing output is not necessary in programs compiled with `ocamlc` and `ocamlopt`. For example, the two previous functions can be defined simultaneously, with only a single `;;` at the end:

```
# let f1 x = if x < 3 then print_endline "Less than three"
  let f2 x = if x < 3 then "Less" else "Greater";;
val f1 : int -> unit = <fun>
val f2 : int -> string = <fun>
```

Note that OCaml has determined that this input corresponds to two separate function definitions. In fact, when written for the `ocamlc` or `ocamlopt` compilers, programs can be written entirely without `;;`, such as:

```
let f1 x = if x < 3 then print_endline "Less than three"
let f2 x = if x < 3 then "Less" else "Greater"
```

As we have seen, expressions which act by way of a side-effect (such as printing) produce the value `()` of type `unit`. Many situations require a sequence of such expressions to be evaluated. Expressions of type `unit` may be concatenated into a compound expression by using the `;` separator. For example, a function to print “A”, “B” and then “C” on three separate lines could be written:

```
# let f () =
  print_endline "A";
  print_endline "B";
  print_endline "C";;
val f : unit -> unit = <fun>
```

Note that there is no final `;`, only the delimiting `;;`, so the value `()` of type `unit` produced by the final call to `print_endline` is returned by our `f` function.

### 1.5.1.7 More about functions

Functions can also be defined anonymously, known as  $\lambda$ -abstraction in computer science. For example, the following defines a function  $f(x) = x \times x$  which has a type representing<sup>8</sup>  $f : \mathbb{Z} \rightarrow \mathbb{Z}$ :

```
# fun x -> x * x;;
- : int -> int = <fun>
```

This is an anonymous equivalent to the `sqr` function defined earlier. The type of this expression is also inferred to be `int -> int`. This anonymous function can be called as if it were the name of a conventional function. For example, applying the function `f` to the value 2 gives  $2 \times 2 = 4$ :

```
# (fun x -> x * x) 2;;
val : int = 4
```

Consequently, we could have defined `sqr` equivalently as:

---

<sup>8</sup>We say “representing” because the OCaml type `int` is, in fact, a finite subset of  $\mathbb{Z}$ , as we shall see in chapter 4.

```
# let sqr = fun x -> x * x;;
val sqr : int -> int = <fun>
```

Once defined, this version of the `sqr` function is indistinguishable from the original.

The `let ... in` construct allows definitions to be nested, including function definitions. For example, the following function `ipow3` raises a given `int` to the power three using a `sqr` function nested within the body of the `ipow3` function:

```
# let ipow3 x =
  let sqr x = x * x in
  x * sqr x;;
val ipow3 : int -> int = <fun>
```

Note that the function application `sqr x` takes precedence over the multiplication.

The `let` construct may also be used to define the elements of a tuple simultaneously. For example, the following defines two variables, `a` and `b`, simultaneously:

```
# let (a, b) = (3, 4);;
val a : int = 3
val b : int = 4
```

This is particularly useful when factoring code. For example, the following definition of the `ipow4` function contains an implementation of the `sqr` function which is identical to that in our previous definition of the `ipow3` function:

```
# let ipow4 x =
  let sqr x = x * x in
  (sqr x) * (sqr x);;
val ipow4 : int -> int = <fun>
```

Just as common subexpressions in a mathematical expression can be factored, so the `ipow3` and `ipow4` functions can be factored by sharing a common `sqr` function and returning the `ipow3` and `ipow4` functions simultaneously in a 2-tuple:

```
# let (ipow3, ipow4) =
  let sqr x = x * x in
  ((fun x -> x * (sqr x)), fun x -> (sqr x) * (sqr x));;
val ipow3 : int -> int = <fun>
val ipow4 : int -> int = <fun>
```

Factoring code is an important way to keep programs manageable. In particular, programs can be factored much more aggressively in the presence of higher-order functions — something which can be done in OCaml but not Java, C++ or Fortran. We shall discuss such factoring of OCaml programs as a means of code structuring in chapter 2. In the mean time, we shall examine functions which perform computations by applying themselves.

As we have already seen, variable names in variable and function definitions refer to their previously defined values. This default behaviour can be overridden using the `rec` keyword,

which allows a variable definition to refer to itself. This is necessary to define a recursive function<sup>9</sup>. For example, the following implementation of the `ipow` function, which computes  $n^m$  for  $n, m \geq 0 \in \mathbb{Z}$ , calls itself recursively with smaller  $m$  to build up the result until the base-case  $n^m = 1$  for  $m = 0$  is reached:

```
# let rec ipow n m = if m = 0 then 1 else n * ipow n (m - 1);;
val ipow : int -> int -> int = <fun>
```

For example,  $2^{16} = 65,536$ :

```
# ipow 2 16;;
- : int = 65536
```

All of the programming constructs we have just introduced may be structured into *modules*.

### 1.5.1.8 Modules

In OCaml, modules are the most commonly used means to encapsulate related definitions. For example, many function definitions pertaining to lists are encapsulated in the `List` module. Visible definitions in modules may be referred to by the notation `module.name` where `module` is the name of the module and `name` is the name of the type or variable definition. For example, the `List` module contains a function `length` which returns the number of elements in the given list:

```
# List.length ["one", "two", "three"];;
- : int = 3
```

The `Pervasives` module contains many common definitions, such as `sqrt`, and is automatically opened before a program starts so these definitions are available immediately.

The OCaml module system and program structuring in general are examined in chapter 2. We shall now examine some of the more advanced features of OCaml in more detail.

## 1.5.2 Pattern matching

As a program is executed, it is quite often necessary to choose the future course of action based upon the value of a previously computed result. As we have already seen, a two-way choice can be implemented using the `if` construct. However, the ability to choose from several different possible actions is often desirable. Although such cases can be reduced to a series of `if` tests, languages typically provide a more general construct to compare a result with several different possibilities more succinctly, more clearly and sometimes more efficiently than manually nested `ifs`. In Fortran, this is the `SELECT CASE` construct. In C and C++, it is the `switch case` construct.

Unlike conventional languages, OCaml allows the value of a previous result to be compared against various patterns - *pattern matching*. As we shall see, this approach is considerably more powerful than the conventional approaches.

The most common pattern matching construct in OCaml is in the `match ... with ...` expression:

<sup>9</sup>A recursive function is a function which calls itself, possibly via other functions.

```

match expr with
  pattern1 -> expr1
| pattern2 -> expr2
| pattern3 -> expr3
| ...

```

This evaluates *expr* and compares the resulting value firstly with *pattern1* then with *pattern2* and so on, until a pattern is found which matches the value of *expr*, in which case the corresponding expression (e.g. *expr2*) is evaluated and returned. A pattern is an expression composed of constants and variable names. When a pattern matches an argument, the variables are bound to values of the corresponding expressions.

Patterns may contain arbitrary data structures (tuples, records, variant types, lists and arrays) and, in particular, the cons operator `::` may be used in a pattern to decapitate a list. Also, the pattern `_` matches any value without assigning a name to it. This is useful for clarifying that part of a pattern is not referred to in the corresponding expression.

For example, the following function compares its argument with several possible patterns of type `int`, returning the expression of type `string` corresponding to the pattern which matches:

```

# let f i = match i with
  0 -> "Zero"
| 3 -> "Three"
| _ -> "Neither zero nor three";;

```

Applying this function to some expressions of type `int` demonstrates the functionality of the `match` construct:

```

# f 0;;
- : string = "Zero"
# f 1;;
- : string = "Neither zero nor three"
# f (1 + 2);;
- : string = "Three"

```

As pattern matching is such a fundamental concept in OCaml programming, we shall provide several more examples using pattern matching in this section.

A function `is_empty_list` which examines a given list and returns `true` if the list is empty and `false` if the list contains any elements, may be written without pattern matching by simply testing equality with the empty list:

```

# let is_empty_list l =
  l = [];;
val is_empty_list : 'a list -> bool = <fun>

```

Using pattern matching, this example may be written using the `match ... with ...` construct as:

```
# let is_empty_list l = match l with
  [] -> true
  | _ -> false;;
val is_empty_list : 'a list -> bool = <fun>
```

Note the use of the anonymous `_` pattern to match any value, in this case accounting for all other possibilities.

The `is_empty_list` function can also be written using the `function ...` construct, used to create one-argument  $\lambda$ -functions which are pattern matched over their argument:

```
# let is_empty_list = function
  [] -> true
  | _ -> false;;
val is_empty_list : 'a list -> bool = <fun>
```

In general, functions which pattern match over their last argument may be rewritten more succinctly using `function`. Let us now consider some additional sophistication supported by OCaml's pattern matching.

### 1.5.2.1 Guarded patterns

Patterns can also have arbitrary tests associated with them, written using the `when` construct. Such patterns are referred to as *guarded patterns* and are only allowed to match when the associated boolean expression evaluates to true. For example, the following function evaluates to `true` only for lists which contain three integers,  $i$ ,  $j$  and  $k$ , satisfying the equality  $i - j - k = 0$ :

```
# let f = function
  [i; j; k] when i - j - k = 0 -> true
  | _ -> false;;
val f : int list -> bool = <fun>
# f [2; 3];;
- : bool = false
# f [5; 2; 3];;
- : bool = true
# f [1; 2; 3];;
- : bool = false
```

Subsequent patterns sharing the same variable bindings and corresponding expression may be written in the short-hand notation:

```
match ... with
  pattern1 | pattern2 | ... -> ...
  | ...
```

For example, the following function returns `true` if the given integer is in the set  $\{-1, 0, 1\}$  and false otherwise:

```
# let is_sign = function
  -1 | 0 | 1 -> true
  | _ -> false;;
val is_sign : int -> bool = <fun>
```

The sophistication provided by pattern matching may be misused. Fortunately, the OCaml compilers go to great lengths to enforce correct use, even brashly criticising the programmers style when appropriate.

### 1.5.2.2 Erroneous patterns

Sequences of patterns which match to the same corresponding expression are required to share the same set of variable bindings. For example, although the following function makes sense to a human, the OCaml compilers object to the patterns `(a, 0.)` and `(0., b)` binding different sets of variables ( $\{a\}$  and  $\{b\}$ , respectively):

```
# let product a b = match (a, b) with
  (a, 0.) | (0., b) -> 0.
  | (a, b) -> a *. b;;
Variable a must occur on both sides of this | pattern
```

In this case, this function could be corrected by using the anonymous `_` pattern as neither `a` nor `b` is used in the first case:

```
# let product a b = match (a, b) with
  (_, 0.) | (0., _) -> 0.
  | (a, b) -> a *. b;;
val product : float -> float -> float = <fun>
```

This actually conveys useful information about the code. Specifically, that the values matched by `_` are not used in the corresponding expression.

OCaml uses type information to determine the possible values of expression being matched over. If the set of pattern matches fails to cover all of the possible values of the input then, at compile-time, the compiler emits:

```
Warning: this pattern-matching is not exhaustive
```

followed by examples of values which could not be matched. If a program containing such pattern matches is executed and no matching pattern is found at run-time then the `Match_failure` exception is raised. Exceptions will be discussed in section 1.5.3.

For example, in the context of the variant type:

```
# type int_option = None | Some of int;;
type int_option = None | Some of int
```

The OCaml compiler will warn of a function matching only `Some ...` values and neglecting the `None` value:

```
# let extract = function Some i -> i;;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
None
val extract : int_option -> int = <fun>
```

This `extract` function then works as expected on `Some ...` values:

```
# extract (Some 3);;
- : int = 3
```

but causes a `Match_failure` exception to be raised at run-time if a `None` value is given, as none of the patterns in the pattern match of the `extract` function match this value:

```
# extract None;;
Exception: Match_failure ("", 5, -40).
```

As some approaches to pattern matching lead to more robust programs, some notions of good and bad programming styles arise in the context of pattern matching.

### 1.5.2.3 Good style

The compiler cannot prove<sup>10</sup> that any given pattern match covers all eventualities in the general case. Thus, some style guidelines may be productively adhered to when writing pattern matches, to aid the compiler in its proofs:

- Guarded patterns should be used only when necessary. In particular, in any given pattern matching, at least one pattern should be unguarded.
- Unless all eventualities are clearly covered (such as `[]` and `h::t` which, between them, match any list) the last pattern should be general.

As proof generation cannot be automated in general, the OCaml compilers do not try to prove that a sequence of guarded patterns will match all possible inputs. Instead, the programmer is expected to adhere to a good programming style, making the breadth of the final match explicit by removing the guard. For example, the OCaml compilers do not prove that the following pattern match covers all possibilities:

```
# let sign = function
  | i when i < 0. -> -1
  | 0. -> 0
  | i when i > 0. -> 1;;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
1.
(However, some guarded clause may match this value.)
val sign : float -> int = <fun>
```

<sup>10</sup>Indeed, it can be proven that the act of proving cannot be automated in the general case.

In this case, the function should have been written without the guard on the last pattern:

```
# let sign = function
  i when i < 0. -> -1
  | 0. -> 0
  | _ -> 1;;
val sign : float -> int = <fun>
```

Also, the OCaml compilers will try to determine any patterns which can never be matched. If such a pattern is found, the compiler will emit a warning. For example, in this case the first match accounts for all possible input values and, therefore, the second match will never be used:

```
# let product a b = match (a, b) with
  (a, b) -> a *. b
  | (_, 0.) | (0., _) -> 0.;;
Warning: this match case is unused.
val product : float -> float -> float = <fun>
```

When matching over the constructors of a type, all eventualities should be caught explicitly, i.e. the final pattern should not be made completely general. For example, in the context of a type which can represent different number representations:

```
# type number = Integer of int | Real of float;;
type number = Integer of int | Real of float
```

A function to test for equality with zero could be written in the following, poor style:

```
# let bad_is_zero = function
  Integer 0 -> true
  | Real 0. -> true
  | _ -> false;;
val bad_is_zero : number -> bool = <fun>
```

When applied to various values of type `number`, this function correctly acts a predicate to test for equality with zero:

```
# bad_is_zero (Integer (-1));;
- : bool = false
# bad_is_zero (Integer 0);;
- : bool = true
# bad_is_zero (Real 0.);;
- : bool = true
# bad_is_zero (Real 2.6);;
- : bool = false
```

Although the `bad_is_zero` function works in this case, a better style would be to extract the numerical values from the constructors and test their equality with zero, avoiding the final catch-all case in the pattern match:

```
# let good_is_zero = function
  Integer i -> i = 0
  | Real x -> x = 0.;;
val good_is_zero : number -> bool = <fun>
```

Not only is the latter style more concise but, more importantly, this style is more robust. For example, if whilst developing our program, we were to supplement the definition of our `number` type with a new representation, say of the complex numbers  $z = x + iy \in \mathbb{C}$ :

```
# type number = Integer of int | Real of float | Complex of float * float;;
type number = Integer of int | Real of float | Complex of float * float
```

the `bad_is_zero` function, which is written in the poor style, would compile without warning to give incorrect functionality:

```
# let bad_is_zero = function
  Integer 0 -> true
  | Real 0. -> true
  | _ -> false;;
val bad_is_zero : number -> bool = <fun>
```

Specifically, this function treats all values which are not zero-integers and zero-reals as being non-zero. Thus, zero-complex  $z = 0 + 0i$  is incorrectly deemed to be non-zero:

```
# bad_is_zero (Complex (0., 0.));;
- : bool = false
```

In contrast, the `good_is_zero` function, which was written using the good style, would allow the compiler to spot that part of the `number` type was no longer being accounted for in the pattern match:

```
# let good_is_zero = function
  Integer i -> i = 0
  | Real x -> x = 0.;;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
Complex (_, _)
val good_is_zero : number -> bool = <fun>
```

The programmer could then supplement this function with a case for complex numbers:

```
# let good_is_zero = function
  Integer i -> i = 0
  | Real x -> x = 0.
  | Complex (x, y) -> x = 0. && y = 0.;;
val good_is_zero : number -> bool = <fun>
```

The resulting function would then provide the correct functionality:

```
# good_is_zero (Complex (0., 0.));;
- : bool = true
```

Clearly, the ability have such safety checks performed at compile-time can be very valuable. This is another important aspect of safety provided by the OCaml language, which results in considerably more robust programs.

Due to the ubiquity of pattern matching in OCaml programs, the number and structure of pattern matches can be non-trivial. In particular, patterns may be nested and may be performed in parallel.

#### 1.5.2.4 Nested patterns

In some cases, nested pattern matches may be desirable. Inner pattern matches may be bundled either into parentheses (...) or, equivalently, into a `begin ... end` construct. When split across multiple lines, the `begin ... end` construct is the conventional choice. For example, the following function tests equality between two values of type `number`<sup>11</sup>:

```
# let number_equal a b = match a with
  Integer i ->
    begin
      match b with
        Integer j when i = j -> true
      | Complex (x, 0.) | Real x when x = float_of_int i -> true
      | Integer _ | Real _ | Complex _ -> false
    end
  | Real x | Complex (x, 0.) ->
    begin
      match b with
        Integer i when x = float_of_int i -> true
      | Complex (y, 0.) | Real y when x = y -> true
      | Integer _ | Real _ | Complex _ -> false
    end
  | Complex (x1, y1) ->
    begin
      match b with
        Complex (x2, y2) when x1 = x2 && y1 = y2 -> true
      | Integer _ | Real _ | Complex _ -> false
    end;;
val number_equal : number -> number -> bool = <fun>
```

In many cases, nested patterns may be written more succinctly and, in fact, more efficient, when presented as a single pattern match which matches different values simultaneously.

#### 1.5.2.5 Parallel pattern matching

In many cases, nested pattern matches may be combined into a single pattern match. This functionality is often obtained by combining variables into a tuple which is then matched over. This is known as *parallel pattern matching*. For example, the previous function could have been written:

---

<sup>11</sup>Note that the built-in, polymorphic equality = could be used to compare values of type `number` but this would perform a structural comparison rather than a numerical comparison, e.g. the expression `Real 1. = Complex (1., 0.)` evaluates to `false`.

```

# let number_equal a b = match (a, b) with
  (Integer i, Integer j) -> i = j
  | (Integer i, (Real x | Complex (x, 0.)))
  | ((Real x | Complex (x, 0.)), Integer i) -> x = float_of_int i
  | ((Real x1 | Complex (x1, 0.)), (Complex (x2, 0.) | Real x2)) -> x1 = x2
  | ((Integer _ | Real _), Complex _)
  | (Complex _, (Integer _ | Real _)) -> false
  | (Complex (x1, y1), Complex (x2, y2)) -> x1 = x2 && y1 = y2;;
val number_equal : number -> number -> bool = <fun>

```

As a core feature of the OCaml language, pattern matching will be used extensively in the remainder of this book, particularly when dissecting data structures in chapter 3. One remaining form of pattern matching in OCaml programs appears in the handling of exceptions.

### 1.5.3 Exceptions

In many programming languages, program execution can be interrupted by the raising<sup>12</sup> of an exception. This is a useful facility, typically used to handle problems such as failing to open a file or an unexpected flow of execution (e.g. due to a program being given invalid input) but exceptions are also useful as an efficient means to escape a computation, as we shall see in section 7.3.3.3.

Like a variant constructor in OCaml, the name of an exception must begin with a capital letter and an exception may or may not carry an associated value. Before an exception can be used, it must be declared. An exception which does not carry associated data may be declared as:

```
exception Name
```

An exception which carries associated data of type *type* may be declared:

```
exception Name of type
```

Exceptions are raised using the `raise` construct. For example, the following raises a built-in exception called `Failure` which carries a string:

```

# raise (Failure "My problem");;
Exception: Failure "My problem".

```

Exceptions may also be caught using the syntax:

```

try
  expr
with
  pattern1 -> expr1
  | pattern2 -> expr2
  | pattern3 -> expr3
  | ...

```

---

<sup>12</sup>Sometimes known as *throwing* an exception, e.g. in the context of the C++ language.

where *expr* is evaluated and its result returned if no exception was raised. If an exception was raised then the exception is matched against the patterns and the value of the corresponding expression (if any) is returned instead.

Note that, unlike other pattern matching constructs, patterns matching over exceptions need not account for all eventualities — any uncaught exceptions simply continue to propagate.

For example, an exception called `ZeroLength`, which does not carry associated data, may be declared as:

```
# exception ZeroLength;;
exception ZeroLength
```

A function to normalise a 2D vector  $\mathbf{r} = (x, y)$  to create a unit 2D vector  $\hat{\mathbf{r}} = \mathbf{r}/|\mathbf{r}|$ , catching the erroneous case of a zero-length vector, may then be written:

```
# let norm (x, y) =
  let l = sqrt (x*.x +. y*.y) in
  if l = 0. then raise ZeroLength else
  let il = 1. /. l in
  (il*.x, il*.y);;
val norm : float * float -> float * float = <fun>
```

Applying the `norm` function to a non-zero-length vector produces the correct result to within numerical error (a subject discussed in chapter 4):

```
# norm (3., 4.);;
- : float * float = (0.600000000000000089, 0.8)
```

Applying the `norm` function to the zero vector raises the `ZeroLength` exception:

```
# norm (0., 0.);;
Exception: ZeroLength.
```

A “safe” version of the norm function might catch this exception and return some reasonable result in the case of a zero-length vector:

```
# let safe_norm r = try norm r with ZeroLength -> (0., 0.);;
val safe_norm : float * float -> float * float = <fun>
```

Applying the `safe_norm` function to a non-zero-length vector causes the result of the expression `norm r` to be returned:

```
# safe_norm (3., 4.);;
- : float * float = (0.600000000000000089, 0.8)
```

However, applying the `safe_norm` function to the zero vector causes the `norm` function to raise the `ZeroLength` exception which is then caught within the `safe_norm` function which then returns the zero vector:

```
# safe_norm (0., 0.);;
- : float * float = (0., 0.)
```

The use of exceptions to handle unusual occurrences, such as in the `safe_norm` function, is one important application of exceptions. This functionality is exploited by many of the functions provided by the core OCaml library, such as those for handling files (discussed in chapter 5). The `safe_norm` function is a simple example using exceptions which could have been written using an `if` expression. However, exceptions are much more useful in more complicated circumstances, where many `if` expressions would be required in order to achieve the same effect.

Another important application is the use of exceptions to escape computations. The usefulness of this way of exploiting exceptions cannot be fully understood without first understanding data structures and algorithms and, therefore, this topic will be discussed in much more detail in chapter 3 and again, in the context of performance, in chapter 7.

The `Pervasives` module defines two exceptions, `Invalid_argument` and `Failure`, as well as two functions which simplify the raising of these exceptions. Specifically, the `invalid_arg` and `failwith` functions raise the `Invalid_argument` and `Failure` exceptions, respectively, using the given string.

Support for exceptions is not uncommon in modern languages. However, the automatic generalisation of functions over all types of data for which they are valid is rather unusual and is discussed next.

### 1.5.4 Polymorphism

As we have seen, OCaml will infer types in a program. But what if a specific type cannot be inferred? In this case, OCaml will create a polymorphic function which can act on any suitable type. For example, the following defines a higher-order function `f` which accepts function `g` and a value `x`, and applies `g` to the result of applying `g` to `x`:

```
# let f g x = g (g x);;
val f : ('a -> 'a) -> 'a -> 'a = <fun>
```

Note that OCaml uses the notation `'a` (conventionally written  $\alpha$ ) when writing the type of the function `f`. This OCaml function may then be used for any type `'a`. For example, the following uses the polymorphic function to calculate  $2^4$ , first using the type `int` and then using the type `float`:

```
# f (fun x -> x * x) 2;;
- : int = 16
# f (fun x -> x *. x) 2.;;
- : float = 16.
```

Types may be constrained by specifying types in a definition using the syntax `(expr : type)`. For example, specifying the type of the argument `x` to be a floating-point value in the definition of the function `f` results in OCaml inferring all of the previously polymorphic types to be `float`:

```
# let f g (x : float) = g (g x);;
val f : (float -> float) -> float -> float = <fun>
```

Although omitting the brackets results in the same types being inferred in this case:

```
# let f g x : float = g (g x);;
val f : (float -> float) -> float -> float = <fun>
```

The syntax of this latter form actually constrains the return type of the function `f` to be `float`, rather than constraining the type of the last argument, as in the former example.

Variant types may contain polymorphic types, in which case the name of the variant type must be preceded by its polymorphic type arguments. For example, the polymorphic option:

```
# type 'a option = None | Some of 'a
type 'a option = None | Some of 'a
```

can have values such as `Some 1` or `Some 2` (for which the type is written `int option`) and the value `None` (for which the type defaults to `'a option`). For example, this 3-tuple is allowed to have elements of different types:

```
# (Some 1, Some 2, None);;
- : int option * int option * 'a option = (Some 1, Some 2, None)
```

In contrast, the elements of a list must all be of the same type and, therefore, a `None` presented as an alternative to a `Some 1` will be inferred to be of type `int option`:

```
# [Some 1; Some 2; None];;
- : int option list = [Some 1; Some 2; None]
```

The polymorphic option type `'a option` is actually already in the `Pervasives` module.

Many polymorphic functions are provided by the language. Most notably the comparison operators `=`, `<>`, `<`, `>`, `<=`, `>=`. Also the total ordering function `compare` which provides conventional ordering over the `int` and `float` types as well as lexicographic ordering over lists, arrays and strings:

$$\text{compare } a b = \begin{cases} -1 & a < b \\ 0 & a = b \\ 1 & a > b \end{cases}$$

```
# compare 1 2;;
- : int = -1
# compare "slug" "plug";;
- : int = 1
```

The `min` and `max` functions use polymorphic comparison to find the smaller and larger of two given arguments, respectively:

```
# max 1 2;;
- : int = 2
# min "slug" "plug";;
- : string = "plug"
```

Before completing this introduction to OCaml, we have one remaining exotic topic to cover.

### 1.5.5 Currying

A curried function is a function which returns a function as its result. Curried functions are best introduced as a more powerful alternative to the conventional (non-curried) functions provided by imperative programming languages.

Effectively, imperative languages only allow functions which accept a single value (often a tuple) as an argument. For example, a raise-to-the-power function for integers would have to accept a single tuple as an argument which contained the two values used by the function:

```
# let rec ipow (x, n) = if n = 0 then 1. else x *. ipow (x, n - 1);;
val ipow : float * int -> float = <fun>
```

But, as we have seen, OCaml also allows:

```
# let rec ipow x n = if n = 0 then 1. else x *. ipow x (n - 1);;
val ipow : float -> int -> float = <fun>
```

This latter approach is actually a powerful generalization of the former, only available in functional programming languages.

The difference between these two styles is subtle but important. In the latter case, the type can be understood to be:

```
val ipow : float -> (int -> float)
```

i.e. this `ipow` function accepts an floating-point number and returns a function which raises that number to the given power. A function which returns a function is referred to as a *curried function*. As the curried style is more general than the non-functional style, functions are written in curried form by default in functional languages.

Now that we have examined the syntax of the OCaml language, we shall explain why the exotic programming styles offered by OCaml are highly relevant in the context of scientific computing.

## 1.6 Functional vs Imperative programming

The vast majority of current programmers write in imperative languages and use an imperative style. This refers to the use of statements or expressions which are designed to act by way of a *side-effect*.

For example, the following declares a mutable variable called `x`, executes a statement which has the effect of modifying the value of `x` (squaring it) and then examines the resulting value of `x`:

```
# let x = ref 2;;
val x : int ref = {contents = 2}
# x := !x * !x;;
- : unit = ()
# !x;;
- : int = 4
```

The only action of the statement “`x := !x * !x`” is to modify the value of an existing variable. This is its *side-effect*. In this case, the statement has no other effect and, consequently, returns the value `()` of type `unit`.

The functional equivalent to this imperative style is to define a new value (in this case, of the same name so that the old value is superseded):

```
# let x = 2;;
val x : int = 2
# let x = x * x;;
val x : int = 4
# x;;
- : int = 4
```

Purely functional programming has several advantages over imperative programming:

- easier to determine variable values, as they cannot be altered
- easier proofs of correctness
- typically more concise in terms of the quantity of source code required to perform a given task
- the ability to reuse old data structures (known as *persistence*) without having to worry about undoing state changes and unwanted interactions
- trivial multi-threading of programs due to the lack of data structure interdependencies

OCaml supports both functional and imperative programming and, hence, is known as an *impure* functional programming language. In particular, the OCaml core library provides implementations of several imperative data structures (strings, arrays and hash tables) as well as functional data structures (lists, sets and maps). We shall examine these data structures in detail in chapter 3.

In addition to mutable data structures, the OCaml language provides looping constructs for imperative programming. The `while` loop executes its body repeatedly while the condition is `true`, returning the value of type `unit` upon completion. For example, this `while` loop repeatedly decrements the mutable variable `x`, until it reaches zero:

```
# let x = ref 5;;
val x : int ref = {contents = 5}
# while !x > 0 do
  decr x;
done;;
- unit = ()
# !x;;
- : int = 0
```

The `for` loop introduces a new loop variable explicitly, giving the initial and final values of the loop variable. For example, this `for` loop runs a loop variable called `i` from 1 to five, incrementing the mutable value `x` five times in total:

```

# for i = 1 to 5 do
  incr x;
done;;
- unit = ()
# !x;;
- : int = 5

```

Thus, `while` and `for` loops in OCaml are analogous to those found in most imperative languages.

In practice, the ability to choose between imperative and functional styles when programming in OCaml is very productive. Many programming tasks are naturally suited to either an imperative or a functional style. For example, portions of a program which deal with user input, such as mouse movements and key-presses, are likely to benefit from an imperative style where the program maintains a state and user input may result in a change of state. In contrast, functions dealing with the manipulation of complex data structures, such as trees and graphs, are likely to benefit from being written in a functional style, using recursive functions and immutable data, as this greatly simplifies the task of writing such functions correctly. In both cases, functions can refer to themselves — *recursive* functions. However, recursive functions are pivotal in functional programming, where they are used to implement functionality equivalent to the `while` and `for` looping constructs we have just examined.

## 1.7 Recursion

When a programmer is introduced to the concept of functional programming for the first time, the way to implement simple programming constructs such as loops does not appear obvious. If the loop variable cannot be changed then how can the loop proceed?

In essence, the answer to this question lies in the ability to convert looping constructs, such as mathematical sums and products, into recursive constructs, such as recurrence relations. For example, the factorial function is typically considered to be a product with the special case  $0! = 1$ :

$$n! = \begin{cases} 1 & n = 0 \\ \prod_{i=1}^n i = 1 \times 2 \times \dots \times (n-1) \times n & n > 0 \end{cases}$$

However, this may also be expressed as a recurrence relation:

$$n! = \begin{cases} 1 & n = 0 \\ n \times (n-1)! & n > 0 \end{cases}$$

Both the product and recurrence-relation forms of the factorial function may be expressed in OCaml. The product form is most obviously implemented in an imperative style, using mutable variables which are iteratively updated to accumulate the value of the product:

```

# let factorial n =
  let ans = ref 1 and n = ref n in
  while (!n > 1) do ( ans := !ans * !n; decr n ) done;
  !ans;;
val factorial : int -> int = <fun>
# factorial 5;;
- val int = 120

```

In contrast, the recurrence relation can be implemented more simply, as a recursive function:

```
# let rec factorial n = if n < 1 then 1 else n * factorial (n - 1);;
val factorial : int -> int = <fun>
# factorial 5;;
- val int = 120
```

In the case of the factorial function, the functional style is considerably more concise and, more importantly, is much easier to reason over, i.e. the functional version is more easily seen to be correct. For sophisticated and intrinsically complicated computations, these advantages result in functional programs often being both simpler and more reliable than their imperative equivalents.

However, functional programming is not always preferable to imperative programming. Many problems naturally lend themselves to either imperative or functional styles. Clearly the factorial function is most easily implemented when considered as a recurrence relation. Other computations are most naturally represented as sums and products. For example, the dot product  $\mathbf{a} \cdot \mathbf{b}$  of a pair of  $d$ -dimensional vectors  $\mathbf{a}$  and  $\mathbf{b}$  is most naturally represented as a sum:

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^d a_i \times b_i$$

This sum can be computed by a rather obfuscated recursive function:

```
# let dot a b =
  let len = Array.length a in
  if len <> Array.length b then invalid_arg "dot" else
  let rec aux i accu =
    if i < len then aux (i+1) (accu +. a.(i) *. b.(i)) else accu in
  aux 0 0.;;
val dot : float array -> float array -> float = <fun>
```

or by a clearer iterative function:

```
# let dot a b =
  let len = Array.length a in
  if len <> Array.length b then invalid_arg "dot" else
  let r = ref 0. in
  for i = 0 to len - 1 do
    r := !r +. a.(i) *. b.(i)
  done;
  !r;;
val dot : float array -> float array -> float = <fun>
```

For example,  $(1, 2, 3) \cdot (2, 3, 4) = 20$ :

```
# dot [|1.; 2.; 3.|] [|2.; 3.; 4.|];;
- : float = 20.
```

In this case, the imperative form of the vector dot product is easier to understand than the recursive form. Regardless of the choice of functional or imperative style, structured design and implementation is an important way to manage complicated problems.

Finally, this introductory chapter would not be complete without providing a taste of the value of OCaml in the context of scientific computing.

## 1.8 Applicability

Conventional languages vehemently separate functions from data. In contrast, OCaml allows the seamless treatment of functions as data. Specifically, OCaml allows functions to be stored as values in data structures, passed as arguments to other functions and returned as the results of expressions, including the return-values of functions. As we shall now demonstrate, this ability can be of direct relevance to scientific applications.

Many numerical algorithms are most obviously expressed as a function which accepts and acts upon another function. For example, consider a function called  $d$  which calculates a numerical approximation to the derivative of a given, one-argument function. The function  $d$  accepts a function  $f : \mathbb{R} \rightarrow \mathbb{R}$  and a value  $x$  and returns a function to compute an approximation to the derivative  $\frac{df}{dx}$  given by  $d : (\mathbb{R} \rightarrow \mathbb{R}) \rightarrow (\mathbb{R} \rightarrow \mathbb{R})$ :

$$d[f](x) = \frac{f(x + \delta) - f(x - \delta)}{2\delta} \simeq \frac{\partial f}{\partial x}$$

This is easily written in OCaml as the curried function `d`<sup>13</sup>:

```
# let d f x =
  let eps = sqrt epsilon_float in
  ((f (x +. eps)) -. (f (x -. eps))) /. (2. *. eps);;
val d : (float -> float) -> float -> float = <fun>
```

For example, consider the function  $f(x) = x^3 - x - 1$ :

```
# let f x = x *. x *. x -. x -. 1.;;
val f : float -> float = <fun>
```

The higher-order function `d` can be used to approximate  $\left. \frac{\partial f}{\partial x} \right|_{x=2} = 11$ :

```
# d f 2.;;
- : float = 10.9999999701976776
```

More importantly, as `d` is a curried function, we can use `d` to create derivative functions. For example, the derivative  $f'(x) = \frac{\partial f}{\partial x}$ :

```
# let f' = d f;;
val f' : float -> float = <fun>
```

The function  $f'$  can now be used to calculate a numerical approximation to the derivative of  $f$  for any  $x$ . For example,  $f'(2) = 11$ :

```
# f' 2.;;
- : float = 10.9999999701976776
```

As this demonstrates, functional programming languages such as OCaml offer many considerable improvements over conventional languages used for scientific computing. Before continuing, readers should be warned that, once learned, the techniques presented in this book soon become indispensable and, therefore, there is no going back after this chapter.

<sup>13</sup>The value `epsilon_float`, defined in the `Pervasives` module, is the smallest floating-point number which, when added to 1, does not give 1. The square root of this value can be shown to give optimal properties when used in this way.