Part I

Language Core

The first part of this book is a complete introduction to the core of the Objective Caml language, in particular the expression evaluation mechanism, static typing and the data memory model.

An expression is the description of a computation. Evaluation of an expression returns a value at the end of the computation. The execution of an Objective Caml program corresponds to the computation of an expression. Functions, program execution control structures, even conditions or loops, are themselves also expressions.

Static typing guarantees that the computation of an expression cannot cause a run-time type error. In fact, application of a function to some arguments (or actual parameters) isn't accepted unless they all have types compatible with the formal parameters indicated in the definition of the function. Furthermore, the Objective Caml language has type infererence: the compiler automatically determines the most general type of an expression.

Finally a minimal knowledge of the representation of data is indispensable to the programmer in order to master the effects of physical modifications to the data.

Outline

Chapter 2 contains a complete presentation of the purely functional part of the language and the constraints due to static typing. The notion of expression evaluation is illustrated there at length. The following control structures are detailed: conditional, function application and pattern matching. The differences between the type and the domain of a function are discussed in order to introduce the exception mechanism. This feature of the language goes beyond the functional context and allows management of computational breakdowns.

Chapter 3 exhibits the imperative style. The constructions there are closer to classic languages. Associative control structures such as sequence and iteration are presented there, as well as mutable data structures. The interaction between physical modifications and sharing of data is then detailed. Type inference is described there in the context of these new constructions.

Chapter 4 compares the two preceding styles and especially presents different mixed styles. This mixture supports in particular the construction of lazy data structures, including mutable ones.

Chapter 5 demonstrates the use of the `Graphics` library included in the language distribution. The basic notions of graphics programming are exhibited there and immediately put into practice. There's even something about GUI construction thanks to the minimal event control provided by this library.

These first four chapters are illustrated by a complete example, the implementation of a calculator, which evolves from chapter to chapter.

Chapter 6 presents three complete applications: a little database, a mini-BASIC interpreter and the game Minesweeper. The first two examples are constructed mainly in a functional style, while the third is done in an imperative style.

The rudiments of syntax

Before beginning we indicate the first elements of the syntax of the language. A program is a sequence of phrases in the language. A phrase is a complete, directly executable syntactic element (an expression, a declaration). A phrase is terminated with a double semi-colon (;;). There are three different types of declarations which are each marked with a different keyword:

value declaration	:	**let**
exception declaration	:	**exception**
type declaration	:	**type**

All the examples given in this part are to be input into the interactive toplevel of the language.

Here's a first (little) Objective Caml program, to be entered into the toplevel, whose prompt is the pound character (#), in which a function *fact* computing the factorial of a natural number, and its application to a natural number 8, are defined.

```
# let rec fact n = if n < 2 then 1 else n * fact(n-1) ;;
val fact : int -> int = <fun>
# fact 8 ;;
- : int = 40320
```

This program consists of two *phrases*. The first is the declaration of a function value and the second is an expression. One sees that the toplevel prints out three pieces of information which are: the name being declared, or a dash (-) in the case of an expression; the inferred type; and the return value. In the case of a function value, the system prints <fun>.

The following example demonstrates the manipulation of functions as values in the language. There we first of all define the function *succ* which calculates the successor of an integer, then the function *compose* which composes two functions. The latter will be applied to *fact* and *succ*.

```
# let succ x = x+1 ;;
val succ : int -> int = <fun>
# let compose f g x = f(g x)  ;;
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
# compose fact succ 8 ;;
- : int = 362880
```

This last call carries out the computation *fact*(*succ* 8) and returns the expected result. Let us note that the functions *fact* and *succ* are passed as parameters to compose in the same way as the natural number 8.

2

Functional programming

The first functional language, Lisp, appeared at the end of the 1950's. That is, at the same time as Fortran, the first representative of the imperative languages. These two languages still exist, although both have evolved greatly. They are used widely for numerical programming (in the case of Fortran) and symbolic applications in the case of Lisp. Interest in functional programming arises from the great ease of writing programs and specifying the values which they manipulate. A program is a function applied to its arguments. It computes a result which is returned (when the computation terminates) as the output of the program. In this way it becomes easy to combine programs: the output of one program becomes an input argument to another, in the sense of function composition.

Functional programming is based on a simple computation model with three constructions: variables, function definitions, and applications of a function to an argument. This model is called the λ-calculus and it was introduced by Alonzo Church in 1932, thus before the first computer. It was created to offer a general theoretical model of the notion of *computability*. In the λ-calculus, all functions are values which can be manipulated. They can be used as arguments to other functions, or returned as the result of a call to another function. The theory of λ-calculus asserts that everything which is computable (i.e., programmable) can be written in this formalism. Its syntax is too limited to make its use as a programming language practical, so primitive values (such as integers or character strings), operations on these primitive values, control structures, and declarations which allow the naming of values or functions and, in particular, recursive functions, have all been added to the λ-calculus to make it more palatable.

There are several classifications of functional languages. For our part, we will distinguish them according to two characteristics which seem to us most salient:

- Without side effects (pure) or with side effects (impure): a pure functional language is a language in which there is no change of state. There everything is simply a computation and the way it is carried out is unimportant. Impure func-

tional languages, such as Lisp or ML, integrate imperative traits such as change of state. They permit the writing of algorithms in a style closer to languages like Fortran, where the order of evaluation of expressions is significant.

- Dynamically typed or statically typed: typing permits verification of whether an argument passed to a function is indeed of the type of the function's formal parameter. This verification can be made during program execution. In that case this verification is called *dynamic typing*. If type errors occur the program will halt in a consistent state. This is the case in the language Lisp. This verification can also be done before program execution, that is, at compilation time. This *a priori* verification is called *static typing*. Having been carried out once and for all, it won't slow down program execution. This is the case in the ML language and its dialects such as Objective Caml. Only correctly typed programs, i.e., those accepted by the type verifier, will be able to be compiled and then executed.

Chapter outline

This chapter presents the basic elements of the functional part of the Objective Caml language, namely its syntactic elements, its language of types and its exception mechanism. This will lead us to the development of a first example of a complete program.

The first section describes the core of the language, beginning with primitive values and the functions which manipulate them. We then go on to structured values and to function values. The basic control structures are introduced as well as local and global value declarations. The second section deals with type definitions for the construction of structured values and with pattern matching to access these structures. The third section compares the inferred type of functions and their domain of definition, which leads us to introduce the exception mechanism. The fourth section illustrates all these notions put together, by describing a simple application: a desktop calculator.

Functional core of Objective Caml

Like all functional languages, Objective Caml is an expression oriented language, where programming consists mainly of creating functions and applying them. The result of the evaluation of one of these expressions is a value in the language and the execution of a program is the evaluation of all the expressions which comprise it.

Primitive values, functions, and types

Integers and floating-point numbers, characters, character strings, and booleans are predefined in Objective Caml.

Numbers

There are two kinds of numbers: integers[1] of type ***int*** and floating-point numbers of type ***float***. Objective Caml follows the IEEE 754 standard[2] for representing double-precision floating-point numbers. The operations on integers and floating-point numbers are described in figure 2.1. Let us note that when the result of an integer operation is outside the interval on which values of type ***int*** are defined, this does not produce an error, but the result is an integer within the system's interval of integers. In other words, all integer operations are operations *modulo* the boundaries of the interval.

integer numbers		floating-point numbers	
`+`	addition	`+.`	addition
`-`	subtraction and unary negation	`-.`	subtraction and unary negation
`*`	multiplication	`*.`	multiplication
`/`	integer division	`/.`	division
`mod`	remainder of integer division	`**`	exponentiation

```
# 1 ;;
- : int = 1
# 1 + 2 ;;
- : int = 3
# 9 / 2 ;;
- : int = 4
# 11 mod 3 ;;
- : int = 2
(* limits of the representation  *)
(* of integers                   *)
# 2147483650 ;;
- : int = 2
```

```
# 2.0 ;;
- : float = 2
# 1.1 +. 2.2 ;;
- : float = 3.3
# 9.1 /. 2.2 ;;
- : float = 4.13636363636
# 1. /. 0. ;;
- : float = inf
(* limits of the representation  *)
(* of floating-point numbers     *)
# 222222222222.11111 ;;
- : float = 222222222222
```

Figure 2.1: Operations on numbers.

Differences between integers and floating-point numbers Values having different types such as ***float*** and ***int*** can never be compared directly. But there are functions for conversion (`float_of_int` and `int_of_float`) between one and the other.

```
# 2 = 2.0 ;;
Characters 5-8:
This expression has type float but is here used with type int
# 3.0 = float_of_int 3 ;;
```

1. In the interval $[-2^{30}, 2^{30} - 1]$ on 32-bit machines and in the interval $[-2^{62}, 2^{62} - 1]$ on 64-bit machines

2. The floating point number $m \times 10^n$ is represented with a 53-bit mantissa m and an exponent n in the interval $[-1022, 1023]$.

```
- : bool = true
```

In the same way, operations on floating-point numbers are distinct from those on integers.

```
# 3 + 2 ;;
- : int = 5
# 3.0 +. 2.0 ;;
- : float = 5
# 3.0 + 2.0 ;;
Characters 0-3:
This expression has type float but is here used with type int
# sin 3.14159 ;;
- : float = 2.65358979335e-06
```

An ill-defined computation, such as a division by zero, will raise an exception (see page 54) which interrupts the computation. Floating-point numbers have a representation for infinite values (printed as `Inf`) and ill-defined computations (printed as `NaN`[3]). The main functions on floating-point numbers are described in figure 2.2.

functions on floats		trigonometric functions	
`ceil`		`cos`	cosine
`floor`		`sin`	sine
`sqrt`	square root	`tan`	tangent
`exp`	exponential	`acos`	arccosine
`log`	natural log	`asin`	arcsine
`log10`	log base 10	`atan`	arctangent

```
# ceil 3.4 ;;
- : float = 4
# floor 3.4 ;;
- : float = 3
# ceil (-.3.4) ;;
- : float = -3
# floor (-.3.4) ;;
- : float = -4
```

```
# sin 1.57078 ;;
- : float = 0.999999999867
# sin (asin 0.707) ;;
- : float = 0.707
# acos 0.0 ;;
- : float = 1.57079632679
# asin 3.14 ;;
- : float = nan
```

Figure 2.2: Functions on floats.

3. Not a Number

Characters and Strings

Characters, type *char*, correspond to integers between 0 and 255 inclusive, following the ASCII encoding for the first 128. The functions `char_of_int` and `int_of_char` support conversion between integers and characters. Character strings, type *string*, are sequences of characters of definite length (less than $2^{24} - 6$). The concatenation operator is `^` . The functions `int_of_string`, `string_of_int`, `string_of_float` and `float_of_string` carry out the various conversions between numbers and character strings.

```
# 'B' ;;
- : char = 'B'
# int_of_char 'B' ;;
- : int = 66
# "is a string" ;;
- : string = "is a string"
# (string_of_int 1987) ^ " is the year Caml was created" ;;
- : string = "1987 is the year Caml was created"
```

Even if a string contains the characters of a number, it won't be possible to use it in operations on numbers without carrying out an explicit conversion.

```
# "1999" + 1 ;;
Characters 1-7:
This expression has type string but is here used with type int
# (int_of_string "1999") + 1 ;;
- : int = 2000
```

Numerous functions on character strings are gathered in the `String` module (see page 217).

Booleans

Booleans, of type *bool*, belong to a set consisting of two values: `true` and `false`. The primitive operators are described in figure 2.3. For historical reasons, the "and" and "or" operators each have two forms.

`not`	negation	`&`	synonym for `&&`
`&&`	sequential and	`or`	synonym for `||`
`||`	sequential or		

Figure 2.3: Operators on booleans.

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

```
# true && false ;;
- : bool = false
```

The operators `&&` and `||`, or their synonyms, evaluate their left argument and then, depending on its value, evaluate their right argument. They can be rewritten in the form of conditional constructs (see page 18).

`=`	structural equality	`<`	less than
`==`	physical equality	`>`	greater than
`<>`	negation of `=`	`<=`	less than or equal to
`!=`	negation of `==`	`>=`	greater than or equal to

Figure 2.4: Equality and comparison operators.

The equality and comparison operators are described in figure 2.4. They are polymorphic, i.e., they can be used to compare two integers as well as two character strings. The only constraint is that their two operands must be of the same type (see page 28).

```
# 1<=118 && (1=2 || not(1=2)) ;;
- : bool = true
# 1.0 <= 118.0 && (1.0 = 2.0 || not (1.0 = 2.0)) ;;
- : bool = true
# "one" < "two" ;;
- : bool = true
# 0 < '0' ;;
Characters 4-7:
This expression has type char but is here used with type int
```

Structural equality tests the equality of two values by traversing their structure, whereas physical equality tests whether the two values occupy the same region in memory. These two equality operators return the same result for simple values: booleans, characters, integers and constant constructors (page 45).

Warning Floating-point numbers and character strings are considered structured values.

Unit

The *unit* type describes a set which possesses only a single element, denoted: `()`.

```
# () ;;
- : unit = ()
```

This value will often be used in imperative programs (see chapter 3, page 67) for functions which carry out side effects. Functions whose result is the value `()` simulate the notion of procedure, which doesn't exist in Objective Caml, just as the type `void` does in the C language.

Cartesian product, tuple

Values of possibly different types can be gathered in pairs or more generally in tuples. The values making up a tuple are separated by commas. The type constructor `*` indicates a tuple. The type *`int * string`* is the type of pairs whose first element is an integer (of type *`int`*) and whose second is a character string (of type *`string`*).

```
# ( 12 , "October" ) ;;
- : int * string = 12, "October"
```

When there is no ambiguity, it can be written more simply:

```
# 12 , "October" ;;
- : int * string = 12, "October"
```

The functions `fst` and `snd` allow access to the first and second elements of a pair.

```
# fst ( 12 , "October" ) ;;
- : int = 12
# snd ( 12 , "October" ) ;;
- : string = "October"
```

These two functions accept pairs whose components are of any type whatsoever. They are polymorphic, in the same way as equality.

```
# fst;;
- : 'a * 'b -> 'a = <fun>
# fst ( "October", 12 ) ;;
- : string = "October"
```

The type *`int * char * string`* is that of triplets whose first element is of type *`int`*, whose second is of type *`char`*, and whose third is of type *`string`*. Its values are written

```
# ( 65 , 'B' , "ascii" ) ;;
- : int * char * string = 65, 'B', "ascii"
```

Warning The functions `fst` and `snd` applied to a tuple, other than a pair, result in a type error.

```
# snd ( 65 , 'B' , "ascii" ) ;;
Characters 7-25:
This expression has type int * char * string but is here used with type
  'a * 'b
```

There is indeed a difference between the type of a pair and that of a triplet. The type *`int * int * int`* is different from the types *`(int * int) * int`* and *`int * (int * int)`*. Functions to access a triplet (and other tuples) are not defined by the core library. One can use pattern matching to define them if need be (see page 34).

Lists

Values of the same type can be gathered into a list. A list can either be empty or consist of elements of the same type.

```
# [] ;;
- : 'a list = []
```

```
# [ 1 ; 2 ; 3 ] ;;
- : int list = [1; 2; 3]
# [ 1 ; "two" ; 3 ] ;;
Characters 14-17:
This expression has type int list but is here used with type string list
```

The function which adds an element at the head of a list is the infix operator :: . It is the analogue of Lisp's *cons*.

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

Concatenation of two lists is also an infix operator `@`.

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

The other list manipulation functions are defined in the `List` library. The functions `hd` and `tl` from this library give respectively the head and the tail of a list when these values exist. These functions are denoted by `List.hd` and `List.tl` to indicate to the system that they can be found in the module `List`[4].

```
# List.hd [ 1 ; 2 ; 3 ] ;;
- : int = 1
# List.hd [] ;;
Uncaught exception: Failure("hd")
```

In this last example, it is indeed problematic to request retrieval of the first element of an empty list. It is for this reason that the system raises an *exception* (see page 54).

Conditional control structure

One of the indispensable control structures in any programming language is the structure called *conditional* (or branch) which guides the computation as a function of a condition.

Syntax : **if** $expr_1$ **then** $expr_2$ **else** $expr_3$

The expression $expr_1$ is of type `bool`. The expressions $expr_2$ and $expr_3$ must be of the same type, whatever it may be.

```
# if 3=4 then 0 else 4 ;;
- : int = 4
# if 3=4 then "0" else "4" ;;
```

4. The `List` module is presented on page 217.

```
- : string = "4"
# if 3=4 then 0 else  "4";;
Characters 20-23:
This expression has type string but is here used with type int
```

A conditional construct is itself an expression and its evaluation returns a value.

```
# (if 3=5 then 8 else 10) + 5 ;;
- : int = 15
```

Note

The **else** branch can be omitted, but in this case it is implicitly replaced by **else** (). Consequently, the type of the expression $expr_2$ must be ***unit*** (see page 79).

Value declarations

A declaration binds a name to a value. There are two types: *global declarations* and *local declarations*. In the first case, the declared names are known to all the expressions following the declaration; in the second, the declared names are only known to one expression. It is equally possible to *simultaneously* declare several name-value bindings.

Global declarations

Syntax : **let** *name* **=** *expr* **;;**

A global declaration defines the binding between the name *name* and the value of the expression *expr* which will be known to all subsequent expressions.

```
# let yr = "1999" ;;
val yr : string = "1999"
# let x = int_of_string(yr) ;;
val x : int = 1999
# x ;;
- : int = 1999
# x + 1 ;;
- : int = 2000
# let new_yr = string_of_int (x + 1) ;;
val new_yr : string = "2000"
```

Simultaneous global declarations

Syntax :

let $name_1$ **=** $expr_1$
and $name_2$ **=** $expr_2$
⋮
and $name_n$ **=** $expr_n$ **;;**

A simultaneous declaration declares different symbols at the same level. They won't be known until the end of all the declarations.

```
# let x = 1 and y = 2 ;;
val x : int = 1
val y : int = 2
# x + y ;;
- : int = 3
# let z = 3 and t = z + 2 ;;
Characters 18-19:
Unbound value z
```

It is possible to gather several global declarations in the same phrase; then printing of their types and their values does not take place until the end of the phrase, marked by double "**;;**". These declarations are evaluated sequentially, in contrast with a simultaneous declaration.

```
# let x = 2
  let y = x + 3  ;;
val x : int = 2
val y : int = 5
```

A global declaration can be masked by a new declaration of the same name (see page 26).

Local declarations

Syntax : **let** *name* **=** $expr_1$ **in** $expr_2$**;;**

The name *name* is only known during the evaluation of $expr_2$. The local declaration binds it to the value of $expr_1$.

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

The local declaration binding `xl` to the value `3` is only in effect during the evaluation of `xl * xl`.

```
# xl ;;
Characters 1-3:
Unbound value xl
```

A local declaration masks all previous declarations of the same name, but the previous value is reinstated upon leaving the scope of the local declaration:

```
# let x = 2 ;;
```

```
val x : int = 2
# let x = 3 in x * x ;;
- : int = 9
# x * x ;;
- : int = 4
```

A local declaration is an expression and can thus be used to construct other expressions:

```
# (let x = 3 in x * x) + 1 ;;
- : int = 10
```

Local declarations can also be simultaneous.

Syntax :

```
let   name_1 = expr_1
and   name_2 = expr_2
 ⋮
and   name_n = expr_n
in    expr ;;
```

```
# let a = 3.0 and b = 4.0 in sqrt (a*.a +. b*.b) ;;
- : float = 5
# b ;;
Characters 0-1:
Unbound value b
```

Function expressions, functions

A function expression consists of a *parameter* and a *body*. The formal parameter is a variable name and the body an expression. The parameter is said to be *abstract*. For this reason, a function expression is also called an *abstraction*.

Syntax : **function** *p* -> *expr*

Thus the function which squares its argument is written:

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

The Objective Caml system deduces its type. The *function type* ***int -> int*** indicates a function expecting a parameter of type ***int*** and returning a value of type ***int***.

Application of a function to an argument is written as the function followed by the argument.

```
# (function x → x * x) 5 ;;
- : int = 25
```

The evaluation of an application amounts to evaluating the body of the function, here

`x * x`, where the formal parameter, `x`, is replaced by the value of the argument (or the *actual parameter*), here `5`.

In the construction of a function expression, ***expr*** is any expression whatsoever. In particular, ***expr*** may itself be a function expression.

```
# function x → (function y → 3*x + y) ;;
- : int -> int -> int = <fun>
```

The parentheses are not required. One can write more simply:

```
# function x → function y → 3*x + y ;;
- : int -> int -> int = <fun>
```

The type of this expression can be read in the usual way as the type of a function which expects two integers and returns an integer value. But in the context of a functional language such as Objective Caml we are dealing more precisely with the type of a function which expects an integer and returns a *function value* of type ***int -> int***:

```
# (function x → function y → 3*x + y) 5 ;;
- : int -> int = <fun>
```

One can, of course, use the function expression in the usual way by applying it to two arguments. One writes:

```
# (function x → function y → 3*x + y) 4 5 ;;
- : int = 17
```

When one writes ***f a b***, there is an implicit parenthesization on the left which makes this expression equivalent to: ***(f a) b***.

Let's examine the application

```
(function x → function y → 3*x + y) 4 5
```

in detail. To compute the value of this expression, it is necessary to compute the value of

```
(function x → function y → 3*x + y) 4
```

which is a *function expression* equivalent to

```
function y → 3*4 + y
```

obtained by replacing ***x*** by `4` in ***3*x + y***. Applying this value (which is a function) to `5` we get the final value `3*4+5 = 17`:

```
# (function x → function y → 3*x + y) 4 5 ;;
- : int = 17
```

Arity of a function

The number of arguments of a function is called its *arity*. Usage inherited from mathematics demands that the arguments of a function be given in parentheses after the name of the function. One writes: $f(4,5)$. We've just seen that in Objective Caml, one more usually writes: f 4 5. One can, of course, write a function expression in Objective Caml which can be applied to $(4,5)$:

```
# function (x,y) → 3*x + y ;;
- : int * int -> int = <fun>
```

But, as its type indicates, this last expression expects not two, but only one argument: a pair of integers. Trying to pass two arguments to a function which expects a pair or trying to pass a pair to a function which expects two arguments results in a type error:

```
# (function (x,y) → 3*x + y) 4 5 ;;
Characters 29-30:
This expression has type int but is here used with type int * int
# (function x → function y → 3*x + y) (4, 5) ;;
Characters 39-43:
This expression has type int * int but is here used with type int
```

Alternative syntax

There is a more compact way of writing function expressions with several parameters. It is a legacy of former versions of the Caml language. Its form is as follows:

Syntax : **fun** $p_1 \ldots p_n$ -> *expr*

It allows one to omit repetitions of the **function** keyword and the arrows. It is equivalent to the following translation:

function p_1 -> ...-> **function** p_n -> *expr*

```
# fun x y → 3*x + y ;;
- : int -> int -> int = <fun>
# (fun x y → 3*x + y) 4 5 ;;
- : int = 17
```

This form is still encountered often, in particular in the libraries provided with the Objective Caml distribution.

Closure

Objective Caml treats a function expression like any other expression and is able to compute its value. The value returned by the computation is a function expression and is called a *closure*. Every Objective Caml expression is evaluated in an *environment*

consisting of name-value bindings coming from the declarations preceding the expression being computed. A closure can be described as a triplet consisting of the name of the formal parameter, the body of the function, and the environment of the expression. This environment needs to be preserved because the body of a function expression may use, in addition to the formal parameters, every other variable declared previously. These variables are said to be "free" in the function expression. Their values will be needed when the function expression is applied.

```
# let m = 3 ;;
val m : int = 3
# function x → x + m ;;
- : int -> int = <fun>
# (function x → x + m) 5 ;;
- : int = 8
```

When application of a closure to an argument returns a new closure, the latter possesses within its environment all the bindings necessary for a future application. The subsection on the scope of variables (see page 26) details this notion. We will return to the memory representation of a closure in chapter 4 (page 103) as well as chapter 12 (page 332).

The function expressions used until now are *anonymous*. It is rather useful to be able to name them.

Function value declarations

Function values are declared in the same way as other language values, by the **let** construct.

```
# let succ = function x → x + 1 ;;
val succ : int -> int = <fun>
# succ 420 ;;
- : int = 421
# let g = function x → function y → 2*x + 3*y ;;
val g : int -> int -> int = <fun>
# g 1 2;;
- : int = 8
```

To simplify writing, the following notation is allowed:

Syntax : **let** *name* $p_1 \ldots p_n$ **=** *expr*

which is equivalent to the following form:

let *name* **= function** p_1 -> ...-> **function** p_n -> *expr*

The following declarations of `succ` and `g` are equivalent to their previous declaration.

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

```
val succ : int -> int = <fun>
# let g x y = 2*x + 3*y ;;
val g : int -> int -> int = <fun>
```

The completely functional character of Objective Caml is brought out by the following example, in which the function `h1` is obtained by the application of `g` to a single integer. In this case one speaks of *partial application*:

```
# let h1 = g 1 ;;
val h1 : int -> int = <fun>
# h1 2 ;;
- : int = 8
```

One can also, starting from *g*, define a function *h2* by fixing the value of the second parameter, *y*, of *g*:

```
# let h2 = function x → g x 2 ;;
val h2 : int -> int = <fun>
# h2 1 ;;
- : int = 8
```

Declaration of infix functions

Certain functions taking two arguments can be applied in infix form. This is the case with addition of integers. One writes `3 + 5` for the application of `+` to `3` and `5`. To use the symbol `+` as a regular function value, this must be syntactically indicated by surrounding the infix symbol with parentheses. The syntax is as follows:

Syntax : `( op )`

The following example defines the function `succ` using `( + )`.

```
# ( + ) ;;
- : int -> int -> int = <fun>
# let succ = ( + ) 1 ;;
val succ : int -> int = <fun>
# succ 3 ;;
- : int = 4
```

It is also possible to define new operators. We define an operator `++`, addition on pairs of integers

```
# let ( ++ ) c1 c2 = (fst c1)+(fst c2), (snd c1)+(snd c2) ;;
val ++ : int * int -> int * int -> int * int = <fun>
# let c = (2,3) ;;
val c : int * int = 2, 3
# c ++ c ;;
- : int * int = 4, 6
```

There is an important limitation on the possible operators. They must contain only *symbols* (such as `*`, `+`, `@`, etc.) and not letters or digits. Certain functions predefined as infixes are exceptions to the rule. They are listed as follows: `or mod land lor lxor lsl lsr asr`.

Higher order functions

A function value (a closure) can be returned as a result. It can equally well be passed as an argument to a function. Functions taking function values as arguments or returning them as results are called *higher order*.

```
# let h = function f → function y → (f y) + y ;;
val h : (int -> int) -> int -> int = <fun>
```

Note

Application is implicitly parenthesized to the left, but function types are implicitly parenthesized to the right. Thus the type of the function `h` can be written

(int -> int) -> int -> int or *(int -> int) -> (int -> int)*

Higher order functions offer elegant possibilities for dealing with lists. For example the function `List.map` can apply a function to all the elements of a list and return the results in a list.

```
# List.map ;;
- : ('a -> 'b) -> 'a list -> 'b list = <fun>
# let square x = string_of_int (x*x) ;;
val square : int -> string = <fun>
# List.map square [1; 2; 3; 4] ;;
- : string list = ["1"; "4"; "9"; "16"]
```

As another example, the function `List.for_all` can find out whether all the elements of a list satisfy a given criterion.

```
# List.for_all ;;
- : ('a -> bool) -> 'a list -> bool = <fun>
# List.for_all (function n → n<>0) [-3; -2; -1; 1; 2; 3] ;;
- : bool = true
# List.for_all (function n → n<>0) [-3; -2; 0; 1; 2; 3] ;;
- : bool = false
```

Scope of variables

In order for it to be possible to evaluate an expression, all the variables appearing therein must be defined. This is the case in particular for the expression `e` in the dec-

laration **let** *p* = *e*. But since p is not yet known within this expression, this variable can only be present if it refers to another value issued by a previous declaration.

```
# let p = p ^ "-suffix" ;;
Characters 9-10:
Unbound value p
# let p = "prefix" ;;
val p : string = "prefix"
# let p = p ^ "-suffix" ;;
val p : string = "prefix-suffix"
```

In Objective Caml, variables are statically bound. The environment used to execute the application of a closure is the one in effect at the moment of its declaration (static scope) and not the one in effect at the moment of application (dynamic scope).

```
# let p = 10 ;;
val p : int = 10
# let k x = (x, p, x+p) ;;
val k : int -> int * int * int = <fun>
# k p ;;
- : int * int * int = 10, 10, 20
# let p = 1000 ;;
val p : int = 1000
# k p ;;
- : int * int * int = 1000, 10, 1010
```

The function k contains a free variable: p. Since the latter is defined in the global environment, the definition of k is legal. The binding between the name p and the value 10 in the environment of the closure k is static, i.e., does not depend on the most recent definition of p.

Recursive declarations

A variable declaration is called *recursive* if it uses its own identifier in its definition. This facility is used mainly for functions, notably to simulate a definition by recurrence. We have just seen that the **let** declaration does not support this. To declare a recursive function we will use a dedicated syntactic construct.

Syntax : **let rec** *name* **=** *expr* **;;**

We can equally well use the syntactic facility for defining function values while indicating the function parameters:

Syntax : **let rec** *name* $p_1 \ldots p_n$ **=** *expr* **;;**

By way of example, here is the function sigma which computes the sum of the (non-negative) integers between 0 and the value of its argument, inclusive.

```
# let rec sigma x = if x = 0 then 0 else x + sigma (x-1) ;;
val sigma : int -> int = <fun>
```

```
# sigma 10 ;;
- : int = 55
```
It may be noted that this function does not terminate if its argument is strictly negative.

A recursive value is in general a function. The compiler rejects some recursive declarations whose values are not functions:
```
# let rec x = x + 1 ;;
Characters 13-18:
This kind of expression is not allowed as right-hand side of 'let rec'
```
We will see however that in certain cases such declarations are allowed (see page 52).

The **let rec** declaration may be combined with the **and** construction for simultaneous declarations. In this case, all the functions defined at the same level are known within the bodies of each of the others. This permits, among other things, the declaration of *mutually recursive* functions.
```
# let rec even n = (n<>1) && ((n=0) or (odd (n-1)))
  and     odd  n = (n<>0) && ((n=1) or (even (n-1)))    ;;
val even : int -> bool = <fun>
val odd : int -> bool = <fun>
# even 4 ;;
- : bool = true
# odd 5 ;;
- : bool = true
```

In the same way, local declarations can be recursive. This new definition of `sigma` tests the validity of its argument before carrying out the computation of the sum defined by a local function `sigma_rec`.
```
# let sigma x =
    let rec sigma_rec x =  if x = 0 then 0 else x + sigma_rec (x-1) in
     if (x<0) then "error: negative argument"
     else "sigma = " ^ (string_of_int (sigma_rec x)) ;;
val sigma : int -> string = <fun>
```

Note

The need to give a return value of the same type, whether the argument is negative or not, has forced us to give the result in the form of a character string. Indeed, what value should be returned by `sigma` when its argument is negative? We will see the proper way to manage this problem, using exceptions (see page 54).

Polymorphism and type constraints

Some functions execute the same code for arguments having different types. For example, creation of a pair from two values doesn't require different functions for each type

known to the system[5]. In the same way, the function to access the first field of a pair doesn't have to be differentiated according to the type of the value of this first field.

```
# let make_pair a b = (a,b) ;;
val make_pair : 'a -> 'b -> 'a * 'b = <fun>
# let p = make_pair "paper" 451 ;;
val p : string * int = "paper", 451
# let a = make_pair 'B' 65 ;;
val a : char * int = 'B', 65
# fst p ;;
- : string = "paper"
# fst a ;;
- : char = 'B'
```

Functions are called polymorphic if their return value or one of their parameters is of a type which need not be specified. The type synthesizer contained in the Objective Caml compiler finds the most general type for each expression. In this case, Objective Caml uses variables, here `'a` and `'b`, to designate these general types. These variables are instantiated to the type of the argument during application of the function.

With Objective Caml's polymorphic functions, we get the advantages of being able to write generic code usable for values of every type, while still preserving the execution safety of static typing. Indeed, although `make_pair` is polymorphic, the value created by `(make_pair 'B' 65)` has a well-specified type which is different from that of `(make_pair "paper" 451)`. Moreover, type verification is carried out on compilation, so the generality of the code does not hamper the efficiency of the program.

Examples of polymorphic functions and values

The following examples of polymorphic functions have functional parameters whose type is parameterized.

The `app` function applies a function to an argument.

```
# let app = function f → function x → f x ;;
val app : ('a -> 'b) -> 'a -> 'b = <fun>
```

So it can be applied to the function `odd` defined previously:

```
# app odd 2;;
- : bool = false
```

The identity function (`id`) takes a parameter and returns it as is.

```
# let id x = x ;;
val id : 'a -> 'a = <fun>
# app id  1 ;;
- : int = 1
```

5. Fortunately since the number of types is only limited by machine memory

The `compose` function takes two functions and another value and composes the application of these two functions to this value.

```
# let compose f g  x = f (g x) ;;
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
# let add1 x = x+1 and mul5 x = x*5 in compose  mul5 add1  9 ;;
- : int = 50
```

It can be seen that the result of `g` must be of the same type as the argument of `f`.

Values other than functions can be polymorphic as well. For example, this is the case for the empty list:

```
# let l = [] ;;
val l : 'a list = []
```

The following example demonstrates that type synthesis indeed arises from resolution of the constraints coming from function application and not from the value obtained upon execution.

```
# let t = List.tl [2] ;;
val t : int list = []
```

The type of `List.tl` is *`'a list -> 'a list`*, so this function applied to a list of integers returns a list of integers. The fact that upon execution it is the empty list which is obtained doesn't change its type at all.

Objective Caml generates parameterized types for every function which doesn't use the form of its arguments. This polymorphism is called *parametric polymorphism*[6].

Type constraint

As the Caml type synthesizer generates the most general type, it may be useful or necessary to specify the type of an expression.

The syntactic form of a type constraint is as follows:

Syntax : **`(`** *expr* **`:`** *`t`* **`)`**

When it runs into such a constraint, the type synthesizer will take it into account while constructing the type of the expression. Using type constraints lets one:

- make the type of the parameters of a function visible;
- forbid the use of a function outside its intended context;
- specify the type of an expression, which will be particularly useful for mutable values (see page 68).

The following examples demonstrate the use of such type constraints

```
# let add (x:int) (y:int) = x + y ;;
```

6. Some predefined functions do not obey this rule, in particular the structural equality function (`=`) which is polymorphic (its type is *`'a -> 'a -> bool`*) but which explores the structure of its arguments to test their equality.

```
val add : int -> int -> int = <fun>
# let make_pair_int (x:int) (y:int) = x,y;;
val make_pair_int : int -> int -> int * int = <fun>
# let compose_fn_int (f : int → int) (g : int → int) (x:int) =
    compose f g  x;;
val compose_fn_int : (int -> int) -> (int -> int) -> int -> int = <fun>
# let nil = ( [] : string list);;
val nil : string list = []
# 'H' :: nil;;
Characters 5-8:
This expression has type string list but is here used with type char list
```

Restricting polymorphism this way lets us control the type of an expression better by constraining the polymorphism of the type deduced by the system. Any defined type whatsoever may be used, including ones containing type variables, as the following example shows:

```
# let llnil = ( [] : 'a list list) ;;
val llnil : 'a list list = []
# [1;2;3]:: llnil ;;
- : int list list = [[1; 2; 3]]
```

The symbol `llnil` is a list of lists of any type whatsoever.

Here we are dealing with constraints, and not replacing Objective Caml's type synthesis with explicit typing. In particular, one cannot generalize types beyond what inference permits.

```
# let add_general (x:'a) (y:'b) = add x y ;;
val add_general : int -> int -> int = <fun>
```

Type constraints will be used in module interfaces (see chapter 14) as well as in class declarations (see chapter 15).

Examples

In this section we will give several somewhat elaborate examples of functions. Most of these functions are predefined in Objective Caml. We will redefine them for the sake of "pedagogy".

Here, the test for the terminal case of recursive functions is implemented by a conditional. Hence a programming style closer to Lisp. We will see how to give a more ML character to these definitions when we present another way of defining functions by case (see page 34).

Length of a list

Let's start with the function `null` which tests whether a list is empty.

```
# let null l =  (l = [] ) ;;
```

```
val null : 'a list -> bool = <fun>
```
Next, we define the function `size` to compute the length of a list (*i.e.*, the number of its elements).
```
# let rec size l =
    if null l then 0
    else 1 + (size (List.tl l)) ;;
val size : 'a list -> int = <fun>
# size [] ;;
- : int = 0
# size [1;2;18;22] ;;
- : int = 4
```
The function `size` tests whether the list argument is empty. If so it returns `0`, if not it returns `1` plus the value resulting from computing the length of the tail of the list.

Iteration of composition

The expression *iterate n f* computes the value `f` iterated n times.
```
# let rec iterate n f =
    if n = 0 then (function x → x)
    else compose f (iterate (n-1) f) ;;
val iterate : int -> ('a -> 'a) -> 'a -> 'a = <fun>
```
The `iterate` function tests whether *n* is 0, if yes it returns the identity function, if not it composes `f` with the iteration of `f n-1` times.

Using `iterate`, one can define exponentiation as iteration of multiplication.
```
# let rec power i n =
   let i_times = ( * ) i in
    iterate n i_times 1 ;;
val power : int -> int -> int = <fun>
# power 2 8 ;;
- : int = 256
```
The `power` function iterates n times the function expression `i_times`, then applies this result to 1, which does indeed compute the nth power of an integer.

Multiplication table

We want to write a function *multab* which computes the multiplication table of an integer passed as an argument.

First we define the function `apply_fun_list` such that, if *f_list* is a list of functions, *apply_fun_list x f_list* returns the list of results of applying each element of *f_list* to *x*.
```
# let rec apply_fun_list x f_list =
    if null f_list then []
    else ((List.hd f_list) x) :: (apply_fun_list x (List.tl f_list)) ;;
val apply_fun_list : 'a -> ('a -> 'b) list -> 'b list = <fun>
# apply_fun_list 1 [( + ) 1;( + ) 2;( + ) 3] ;;
```

```
- : int list = [2; 3; 4]
```

The function *mk_mult_fun_list* returns the list of functions multiplying their argument by *i*, for *i* varying from 0 to *n*.

```
# let mk_mult_fun_list n =
    let rec mmfl_aux p =
      if p = n then [ ( * ) n ]
      else (( * ) p) :: (mmfl_aux (p+1))
    in  (mmfl_aux 1) ;;
val mk_mult_fun_list : int -> (int -> int) list = <fun>
```

We obtain the multiplication table of 7 by:

```
# let multab n = apply_fun_list n (mk_mult_fun_list 10) ;;
val multab : int -> int list = <fun>
# multab 7 ;;
- : int list = [7; 14; 21; 28; 35; 42; 49; 56; 63; 70]
```

Iteration over lists

The function call *fold_left f a* [*e1*; *e2*; ... ; *en*] returns *f* ... (*f* (*f a e1*) *e2*) ... *en*. So there are n applications.

```
# let rec fold_left f a l =
    if null l then a
    else fold_left f ( f a (List.hd l)) (List.tl l) ;;
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

The function `fold_left` permits the compact definition of a function to compute the sum of the elements of a list of integers:

```
# let sum_list = fold_left (+) 0 ;;
val sum_list : int list -> int = <fun>
# sum_list [2;4;7] ;;
- : int = 13
```

Or else, the concatenation of the elements of a list of strings:

```
# let concat_list = fold_left (^) "" ;;
val concat_list : string list -> string = <fun>
# concat_list ["Hello "; "world" ; "!"] ;;
- : string = "Hello world!"
```

Type declarations and pattern matching

Although Objective Caml's predefined types permit the construction of data structures from tuples and lists, one needs to be able to define new types to describe certain data structures. In Objective Caml, type declarations are recursive and may be parameterized by type variables, in the same vein as the type `'a list` already encountered. Type inference takes these new declarations into account to produce the type of an expression. The construction of values of these new types uses the *constructors* described in their definition. A special feature of languages in the ML family is *pattern matching*. It allows simple access to the components of complex data structures. A function definition most often corresponds to pattern matching over one of its parameters, allowing the function to be defined by cases.

First of all we present pattern matching over the predefined types, and then go on to describe the various ways to declare structured types and how to construct values of such types, as well as how to access their components through pattern matching.

Pattern matching

A pattern is not strictly speaking an Objective Caml expression. It's more like a correct (syntactically, and from the point of view of types) arrangement of elements such as constants of the primitive types (`int`, `bool`, `char`, ..), variables, constructors, and the symbol _ called the *wildcard pattern*. Other symbols are used in writing patterns. We will introduce them to the extent needed.

Pattern matching applies to values. It is used to recognize the form of this value and lets the computation be guided accordingly, associating with each pattern an expression to compute.

Syntax :

match *expr* **with**
| p_1 -> $expr_1$
⋮
| p_n -> $expr_n$

The expression *expr* is matched sequentially to the various patterns $p_1, \ldots, p_n$. If one of the patterns (for example p_i) is consistent with the value of *expr* then the corresponding computation branch ($expr_i$) is evaluated. The various patterns p_i are of the same type. The same goes for the various expressions $expr_i$. The vertical bar preceding the first pattern is optional.

Examples

Here are two ways to define by pattern matching a function `imply` of type `(bool * bool) -> bool` implementing logical implication. A pattern which matches pairs has the form `( , )`.

The first version of `imply` enumerates all possible cases, as a truth table would:

```
# let imply v = match v with
      (true,true)   → true
    | (true,false)  → false
    | (false,true)  → true
    | (false,false) → true;;
val imply : bool * bool -> bool = <fun>
```

By using variables which group together several cases, we obtain a more compact definition.

```
# let imply v = match v with
      (true,x)  → x
    | (false,x) → true;;
val imply : bool * bool -> bool = <fun>
```

These two versions of `imply` compute the same function. That is, they return the same values for the same inputs.

Linear pattern

A pattern must necessarily be *linear*, that is, no given variable can occur more than once inside the pattern being matched. Thus, we might have hoped to be able to write:

```
# let equal c = match c with
     (x,x) → true
   | (x,y) → false;;
Characters 35-36:
This variable is bound several times in this matching
```

But this would have required the compiler to know how to carry out equality tests. Yet this immediately raises numerous problems. If we accept physical equality between values, we get a system which is too weak, incapable of recognizing the equality between two occurrences of the list `[1; 2]`, for example. If we decide to use structural equality, we run the risk of having to traverse, ad infinitum, circular structures. Recursive functions, for example, are circular structures, but we can construct recursive, hence circular, values which are not functions as well (see page 52).

Wildcard pattern

The symbol _ matches all possible values. It is called a wildcard pattern. It can be used to match complex types. We use it, for example, to further simplify the definition of the function `imply`:

```
# let imply v = match v with
          (true,false) → false
        |   _          → true;;
val imply : bool * bool -> bool = <fun>
```

A definition by pattern matching must handle the entire set of possible cases of the values being matched. If this is not the case, the compiler prints a warning message:

```
# let is_zero n = match n with  0 → true ;;
Characters 17-40:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
1
val is_zero : int -> bool = <fun>
```

Indeed if the actual parameter is different from `0` the function doesn't know what value to return. So the case analysis can be completed using the wildcard pattern.

```
# let is_zero n = match n with
     0 → true
   | _ → false ;;
val is_zero : int -> bool = <fun>
```

If, at run-time, no pattern is selected, then an exception is raised. Thus, one can write:

```
# let f x = match x with 1 → 3 ;;
Characters 11-30:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
0
val f : int -> int = <fun>
# f 1 ;;
- : int = 3
# f 4 ;;
Uncaught exception: Match_failure("", 11, 30)
```

The `Match_Failure` exception is raised by the call to `f 4`, and if it is not handled induces the computation in progress to halt (see 54)

Combining patterns

Combining several patterns lets us obtain a new pattern which can match a value according to one or another of the original patterns. The syntactic form is as follows:

Syntax : p_1 **|** ... **|** p_n

It constructs a new pattern by combining the patterns p_1, ... and p_n. The only strong constraint is that all naming is forbidden within these patterns. So each one of them must contain only constant values or the wildcard pattern. The following example demonstrates how to verify that a character is a vowel.

```
# let is_a_vowel c = match c with
     'a' | 'e' | 'i' | 'o' | 'u' | 'y' → true
   | _ → false ;;
val is_a_vowel : char -> bool = <fun>
```

```
# is_a_vowel 'i' ;;
- : bool = true
# is_a_vowel 'j' ;;
- : bool = false
```

Pattern matching of a parameter

Pattern matching is used in an essential way for defining functions by cases. To make writing these definitions easier, the syntactic construct **`function`** allows pattern matching of a parameter:

Syntax :

`function`	**`|`**	p_1 `->` $expr_1$
	`|`	p_2 `->` $expr_2$
		$\vdots$
	`|`	p_n `->` $expr_n$

The vertical bar preceding the first pattern is optional here as well. In fact, like Mr. Jourdain, each time we define a function, we use pattern matching[7]. Indeed, the construction `function x -> expression`, is a definition by pattern matching using a single pattern reduced to one variable. One can make use of this detail with simple patterns as in:

```
# let f = function (x,y) → 2*x + 3*y + 4 ;;
val f : int * int -> int = <fun>
```

In fact the form

`function` p_1 `->` $expr_1$ **`|`** $\ldots$ **`|`** p_n `->` $expr_n$

is equivalent to

`function` $expr$ `->` **`match`** `expr` **`with`** p_1 `->` $expr_1$ **`|`** $\ldots$ **`|`** p_n `->` $expr_n$

Using the equivalence of the declarations mentioned on page 24, we write:

```
# let f (x,y) = 2*x + 3*y + 4 ;;
val f : int * int -> int = <fun>
```

But this natural way of writing is only possible if the value being matched belongs to

7. Translator's note: In Molière's play *Le Bourgeois Gentilhomme* (*The Bourgeois Gentleman*), the character Mr. Jourdain is amazed to discover that he has been speaking prose all his life. The play can be found at
Link: http://www.site-moliere.com/pieces/bourgeoi.htm
and
Link: http://moliere-in-english.com/bourgeois.html
gives an excerpt from an English translation, including this part.

a type having only a single constructor. If such is not the case, the pattern matching is not exhaustive:

```
# let is_zero 0 = true ;;
Characters 13-21:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
1
val is_zero : int -> bool = <fun>
```

Naming a value being matched

During pattern matching, it is sometimes useful to name part or all of the pattern. The following syntactic form introduces the keyword **as** which binds a name to a pattern.

Syntax : **(** *p* **as** *name* **)**

This is useful when one needs to take apart a value while still maintaining its integrity. In the following example, the function `min_rat` gives the smaller rational of a pair of rationals. The latter are each represented by a numerator and denominator in a pair.

```
# let min_rat pr = match pr with
    ((_,0),p2) →  p2
  | (p1,(_,0)) →  p1
  | (((n1,d1) as r1), ((n2,d2) as r2)) →
          if (n1 *  d2 ) < (n2 * d1) then r1 else r2;;
val min_rat : (int * int) * (int * int) -> int * int = <fun>
```

To compare two rationals, it is necessary to take them apart in order to name their numerators and denominators (`n1`, `n2`, `d1` and `d2`), but the initial pair (`r1` or `r2`) must be returned. The **as** construct allows us to name the parts of a single value in this way. This lets us avoid having to reconstruct the rational returned as the result.

Pattern matching with guards

Pattern matching with guards corresponds to the evaluation of a conditional expression immediately after the pattern is matched. If this expression comes back `true`, then the expression associated with that pattern is evaluated, otherwise pattern matching continues with the following pattern.

Syntax :

match *expr* **with**
⋮
| p_i **when** $cond_i$ -> $expr_i$
⋮

The following example uses two guards to test equality of two rationals.

```
# let eq_rat cr = match cr with
```

```
    ((_,0),(_,0)) →  true
  | ((_,0),_) →  false
  | (_,(_,0)) →  false
  | ((n1,1), (n2,1)) when n1 = n2 → true
  | ((n1,d1), (n2,d2)) when ((n1 * d2) = (n2 * d1)) → true
  | _ → false;;
val eq_rat : (int * int) * (int * int) -> bool = <fun>
```

If the guard fails when the fourth pattern is matched, matching continues with the fifth pattern.

Note

The verification carried out by Objective Caml as to whether the pattern matching is exhaustive assumes that the conditional expression in the guard may be false. Consequently, it does not count this pattern since it is not possible to know, before execution, whether the guard will be satisfied or not.

It won't be possible to detect that the pattern matching in the following example is exhaustive.

```
#  let f = function x when x = x  → true;;
Characters 10-40:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:

-
val f : 'a -> bool = <fun>
```

Pattern matching on character intervals

In the context of pattern matching on characters, it is tedious to construct the combination of all the patterns corresponding to a character interval. Indeed, if one wishes to test a character or even a letter, one would need to write 26 patterns at a minimum and combine them. For characters, Objective Caml permits writing patterns of the form:

Syntax : `'c1' .. 'cn'`

It is equivalent to the combination: `'c1' | 'c2' | ...| 'cn'`.

For example the pattern `'0' .. '9'` corresponds to the pattern `'0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'`. The first form is nicer to read and quicker to write.

Warning This feature is among the extensions to the language and may change in future versions.

Using combined patterns and intervals, we define a function categorizing characters according to several criteria.

```
# let char_discriminate c = match c with
```

```
      'a' | 'e' | 'i' | 'o' | 'u' | 'y'
    | 'A' | 'E' | 'I' | 'O' | 'U' | 'Y'  → "Vowel"
    | 'a'..'z' | 'A'..'Z' → "Consonant"
    | '0'..'9' → "Digit"
    |   _ → "Other" ;;
val char_discriminate : char -> string = <fun>
```

It should be noted that the order of the groups of patterns has some significance. Indeed, the second set of patterns includes the first, but it is not examined until after the check on the first.

Pattern matching on lists

As we have seen, a list can be:

- either empty (the list is of the form **`[]`**),
- or composed of a first element (its head) and a sublist (its tail). The list is then of the form `h::t`.

These two possible ways of writing a list can be used as patterns and allow pattern matching on a list.

```
# let rec size x = match x with
      [] → 0
    | _::tail_x → 1 + (size tail_x) ;;
val size : 'a list -> int = <fun>
# size [];;
- : int = 0
# size [7;9;2;6];;
- : int = 4
```

So we can redo the examples described previously (see page 31) using pattern matching, such as iteration over lists for example.

```
# let rec fold_left f a = function
     [] → a
   | head::tail → fold_left f (f a head) tail ;;
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
# fold_left (+) 0 [8;4;10];;
- : int = 22
```

Value declaration through pattern matching

Value declaration in fact uses pattern matching. The declaration **`let`** *x* `=` `18` matches the value `18` with the pattern *x*. Any pattern is allowed as the left-hand side of a declaration; the variables in the pattern are bound to the values which they match.

```
# let (a,b,c) = (1, true, 'A');;
val a : int = 1
```

```
val b : bool = true
val c : char = 'A'
# let (d,c) = 8, 3 in d + c;;
- : int = 11
```

The scope of pattern variables is the usual static scope for local declarations. Here, `c` remains bound to the value `'A'`.

```
# a + (int_of_char c);;
- : int = 66
```

As with any kind of pattern matching, value declaration may not be exhaustive.

```
# let [x;y;z] = [1;2;3];;
Characters 5-12:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val x : int = 1
val y : int = 2
val z : int = 3
# let [x;y;z] = [1;2;3;4];;
Characters 4-11:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
Uncaught exception: Match_failure("", 4, 11)
```

Any pattern is allowed, including constructors, wildcards and combined patterns.

```
# let head :: 2 :: _ =  [1; 2; 3] ;;
Characters 5-19:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val head : int = 1
# let _ = 3. +. 0.14 in "PI" ;;
- : string = "PI"
```

This last example is of little use in the functional world insofar as the computed value `3.14` is not named and so is lost.

Type declaration

Type declarations are another possible ingredient in an Objective Caml phrase. They support the definition of new types corresponding to the original data structures used in a program. There are two major families of types: *product* types for tuples or records; and *sum* types for unions.

Type declarations use the keyword **type**.

Syntax : **type** *name* **=** *typedef* **;;**

In contrast with variable declarations, type declarations are recursive by default. That is, type declarations, when combined, support the declaration of mutually recursive types.

Syntax :

type $name_1$ **=** $typedef_1$
and $name_2$ **=** $typedef_2$
⋮
and $name_n$ **=** $typedef_n$ **;;**

Type declarations can be parameterized by type variables. A type variable name always begins with an apostrophe (the **'** character):

Syntax : **type** *'a name* **=** *typedef* **;;**

When there are several of them, the type parameters are declared as a tuple in front of the name of the type:

Syntax : **type (**$'a_1$ ... $'a_n$**)** *name* **=** *typedef* **;;**

Only the type parameters defined on the left-hand side of the declaration may appear on the right-hand side.

Note

Objective Caml's type printer renames the type parameters encountered; the first is called *'a*, the second *'b* and so forth.

One can always define a new type from one or more existing types.

Syntax : **type** *name* **=** *type expression*

This is useful for constraining a type which one finds too general.

```
# type 'param paired_with_integer = int * 'param ;;
type 'a paired_with_integer = int * 'a
# type specific_pair = float paired_with_integer ;;
type specific_pair = float paired_with_integer
```

Nevertheless without type constraints, inference will produce the most general type.

```
# let x = (3, 3.14) ;;
val x : int * float = 3, 3.14
```

But one can use a type constraint to see the desired name appear:

```
# let (x:specific_pair) = (3, 3.14) ;;
val x : specific_pair = 3, 3.14
```

Records

Records are tuples, each of whose fields is named in the same way as the Pascal *record* or the C *struct*. A record always corresponds to the declaration of a new type. A record type is defined by the declaration of its name and the names and types of each of its fields.

Syntax : **type** *name* **=** { *name*$_1$ **:** *t*$_1$**;** ... **;** *name*$_n$ **:** *t*$_n$ } **;;**

We can define a type representing complex numbers by:

```
# type complex = { re:float; im:float } ;;
type complex = { re: float; im: float }
```

The creation of a value of record type is done by giving a value to each of its fields (in arbitrary order).

Syntax : { *name*$_{i_1}$ **=** *expr*$_{i_1}$**;** ... **;** *name*$_{i_n}$ **=** *expr*$_{i_n}$ } **;;**

For example, we create a complex number with real part `2.` and imaginary part `3.`:

```
# let c = {re=2.;im=3.} ;;
val c : complex = {re=2; im=3}
# c = {im=3.;re=2.} ;;
- : bool = true
```

In the case where some fields are missing, the following error is produced:

```
# let d = { im=4. } ;;
Characters 9-18:
Some labels are undefined
```

A field can be accessed in two ways: by the dot notation or by pattern matching on certain fields.

The dot notation syntax is as usual:

Syntax : *expr.name*

The expression *expr* must be of a record type containing a field *name*.

Pattern matching a record lets one retrieve the value bound to several fields. A pattern to match a record has the following syntax:

Syntax : { *name*$_i$ **=** *p*$_i$ **;** ... **;** *name*$_j$ **=** *p*$_j$ }

The patterns are to the right of the **=** sign (p_i, ..., p_j). It is not necessary to make all the fields of a record appear in such a pattern.

The function `add_complex` accesses fields through the dot notation, while the function `mult_complex` accesses them through pattern matching.

```
# let add_complex c1 c2 = {re=c1.re+.c2.re; im=c1.im+.c2.im};;
val add_complex : complex -> complex -> complex = <fun>
# add_complex c c ;;
- : complex = {re=4; im=6}
# let mult_complex c1 c2 = match (c1,c2) with
    ({re=x1;im=y1},{re=x2;im=y2}) → {re=x1*.x2-.y1*.y2;im=x1*.y2+.x2*.y1} ;;
val mult_complex : complex -> complex -> complex = <fun>
# mult_complex c c ;;
- : complex = {re=-5; im=12}
```

The advantages of records, as opposed to tuples, are at least twofold:

- descriptive and distinguishing information thanks to the field names: in particular this allows pattern matching to be simplified;
- access in an identical way, by name, to any field of the record whatsoever: the order of the fields no longer has significance, only their names count.

The following example shows the ease of accessing the fields of records as opposed to tuples:

```
# let a = (1,2,3) ;;
val a : int * int * int = 1, 2, 3
# let f tr = match tr with  x,_,_ → x ;;
val f : 'a * 'b * 'c -> 'a = <fun>
# f a ;;
- : int = 1
# type triplet = {x1:int; x2:int; x3:int} ;;
type triplet = { x1: int; x2: int; x3: int }
# let b = {x1=1; x2=2; x3=3} ;;
val b : triplet = {x1=1; x2=2; x3=3}
# let g tr = tr.x1 ;;
val g : triplet -> int = <fun>
# g b ;;
- : int = 1
```

For pattern matching, it is not necessary to indicate all the fields of the record being matched. The inferred type is then that of the last field.

```
# let h tr = match tr with  {x1=x} → x;;
val h : triplet -> int = <fun>
# h b;;
- : int = 1
```

There is a construction which lets one create a record identical to another except for some fields. It is often useful for records containing many fields.

Syntax : { *name* **with** $name_i$**=** $expr_i$ **;** ... **;** $name_j$**=**$expr_j$}

```
# let c = {b with x1=0} ;;
val c : triplet = {x1=0; x2=2; x3=3}
```

A new copy of the value of `b` is created where only the field `x1` has a new value.

Warning This feature is among the extensions to the language and may change in future versions.

Sum types

In contrast with tuples or records, which correspond to a Cartesian product, the declaration of a sum type corresponds to a union of sets. Different types (for example integers or character strings) are gathered into a single type. The various members of the sum are distinguished by *constructors*, which support on the one hand, as their name indicates, *construction* of values of this type and on the other hand, thanks to pattern matching, *access* to the components of these values. To apply a constructor to an argument is to indicate that the value returned belongs to this new type.

A sum type is declared by giving the names of its constructors and the types of their eventual arguments.

Syntax :

type *name* **=** ...
| $Name_i$ ...
| $Name_j$ **of** t_j ...
| $Name_k$ **of** t_k ***** ...***** t_l ... **;;**

A constructor name is a particular identifier:

Warning The names of constructors always begin with a capital letter.

Constant constructors

A constructor which doesn't expect an argument is called a *constant constructor*. Constant constructors can subsequently be used directly as a value in the language, as a constant.

```
# type coin = Heads | Tails;;
type coin = | Heads | Tails
# Tails;;
- : coin = Tails
```

The type `bool` can be defined in this way.

Constructors with arguments

Constructors can have arguments. The keyword **of** indicates the type of the constructor's arguments. This supports the gathering into a single type of objects of different types, each one being introduced with a particular constructor.

Here is a classic example of defining a datatype to represent the cards in a game, here Tarot[8]. The types *suit* and *card* are defined in the following way:

```
# type suit = Spades | Hearts | Diamonds | Clubs ;;
# type card =
       King of suit
     | Queen of suit
     | Knight of suit
     | Knave of suit
     | Minor_card of suit * int
     | Trump of int
     | Joker ;;
```

The creation of a value of type *card* is carried out through the application of a constructor to a value of the appropriate type.

```
# King Spades ;;
- : card = King Spades
# Minor_card(Hearts, 10) ;;
- : card = Minor_card (Hearts, 10)
# Trump 21 ;;
- : card = Trump 21
```

And here, for example, is the function `all_cards` which constructs a list of all the cards of a suit passed as a parameter.

```
# let rec interval a b =  if a = b then [b] else a :: (interval (a+1) b) ;;
val interval : int -> int -> int list = <fun>
# let all_cards s =
     let face_cards = [ Knave s; Knight s; Queen s; King s ]
     and other_cards  = List.map (function n → Minor_card(s,n)) (interval 1 10)
     in face_cards @ other_cards ;;
val all_cards : suit -> card list = <fun>
# all_cards Hearts ;;
- : card list =
[Knave Hearts; Knight Hearts; Queen Hearts; King Hearts;
 Minor_card (Hearts, 1); Minor_card (Hearts, 2); Minor_card (Hearts, 3);
 Minor_card (Hearts, ...); ...]
```

8. Translator's note: The rules for French Tarot can be found, for example, at
Link: http://www.pagat.com/tarot/frtarot.html

To handle values of sum types, we use pattern matching. The following example constructs conversion functions from values of type *suit* and of type *card* to character strings (type *string*):

```
# let string_of_suit = function
        Spades   → "spades"
      | Diamonds → "diamonds"
      | Hearts   → "hearts"
      | Clubs    → "clubs"  ;;
val string_of_suit : suit -> string = <fun>
# let string_of_card = function
        King c            → "king of " ^ (string_of_suit c)
      | Queen c           → "queen of " ^ (string_of_suit c)
      | Knave c           → "knave of " ^ (string_of_suit c)
      | Knight c          → "knight of " ^ (string_of_suit c)
      | Minor_card (c, n) → (string_of_int n) ^ " of "^(string_of_suit c)
      | Trump n           → (string_of_int n) ^ " of trumps"
      | Joker             → "joker" ;;
val string_of_card : card -> string = <fun>
```

Lining up the patterns makes these functions easy to read.

The constructor `Minor_card` is treated as a constructor with *two* arguments. Pattern matching on such a value requires naming its two components.

```
# let is_minor_card c = match c with
      Minor_card v →  true
    | _ → false;;
Characters 41-53:
The constructor Minor_card expects 2 argument(s),
but is here applied to 1 argument(s)
```

To avoid having to name each component of a constructor, one declares it to have a single argument by parenthesizing the corresponding tuple type. The two constructors which follow are pattern-matched differently.

```
# type t =
        C of int * bool
      | D of (int * bool) ;;
# let access v = match v with
        C (i, b) → i,b
      | D x  → x;;
val access : t -> int * bool = <fun>
```

Recursive types

Recursive type definitions are indispensable in any algorithmic language for describing the usual data structures (lists, heaps, trees, graphs, etc.). To this end, in Objective Caml type definition is recursive by default, in contrast with value declaration (**let**).

Objective Caml's predefined type of lists only takes a single parameter. One may wish to store values of belonging to two different types in a list structure, for example, integers (*int*) or characters (*char*). In this case, one defines:

```
# type int_or_char_list =
        Nil
      | Int_cons of int * int_or_char_list
      | Char_cons of char * int_or_char_list ;;

# let l1 =  Char_cons ( '=', Int_cons(5, Nil) )  in
     Int_cons ( 2, Char_cons ( '+', Int_cons(3, l1) ) )  ;;
- : int_or_char_list =
Int_cons (2, Char_cons ('+', Int_cons (3, Char_cons ('=', Int_cons (...)))))
```

Parametrized types

A user can equally well declare types with parameters. This lets us generalize the example of lists containing values of two different types.

```
# type ('a, 'b) list2 =
        Nil
      | Acons of 'a * ('a, 'b) list2
      | Bcons of 'b * ('a, 'b) list2 ;;

# Acons(2, Bcons('+', Acons(3, Bcons('=', Acons(5, Nil))))) ;;
- : (int, char) list2 =
Acons (2, Bcons ('+', Acons (3, Bcons ('=', Acons (...)))))
```

One can, obviously, instantiate the parameters *'a* and *'b* with the same type.

```
# Acons(1, Bcons(2, Acons(3, Bcons(4, Nil)))) ;;
- : (int, int) list2 = Acons (1, Bcons (2, Acons (3, Bcons (4, Nil))))
```

This use of the type *list2* can, as in the preceding example, serve to mark even integers and odd integers. In this way we extract the sublist of even integers in order to construct an ordinary list.

```
# let rec extract_odd = function
       Nil → []
     | Acons(_, x) → extract_odd x
     | Bcons(n, x) → n :: (extract_odd x) ;;
val extract_odd : ('a, 'b) list2 -> 'b list = <fun>
```

The definition of this function doesn't give a single clue as to the nature of the values stored in the structure. That is why its type is parameterized.

Scope of declarations

Constructor names obey the same scope discipline as global declarations: a redefinition masks the previous one. Nevertheless values of the masked type still exist. The interactive toplevel does not distinguish these two types in its output. Whence some unclear error messages.

In this first example, the constant constructor `Nil` of type *int_or_char* has been masked by the constructor declarations of the type *('a, 'b) list2*.

```
# Int_cons(0, Nil) ;;
Characters 13-16:
This expression has type ('a, 'b) list2 but is here used with type
  int_or_char_list
```

This second example provokes a rather baffling error message, at least the first time it appears. Let the little program be as follows:

```
# type t1 = Empty | Full;;
type t1 = | Empty | Full
# let empty_t1 x = match x with Empty → true | Full → false ;;
val empty_t1 : t1 -> bool = <fun>
# empty_t1 Empty;;
- : bool = true
```

Then, we redeclare the type *t1*:

```
# type t1 = {u : int; v : int} ;;
type t1 = { u: int; v: int }
# let y = { u=2; v=3 } ;;
val y : t1 = {u=2; v=3}
```

Now if we apply the function `empty_t1` to a value of the new type *t1*, we get the following error message:

```
# empty_t1 y;;
Characters 10-11:
This expression has type t1 but is here used with type t1
```

The first occurrence of *t1* represents the first type defined, while the second corresponds to the second type.

Function types

The type of the argument of a constructor may be arbitrary. In particular, it may very well contain a function type. The following type constructs lists, all of whose elements except the last are function values.

```
# type 'a listf =
      Val of 'a
    | Fun of ('a → 'a) * 'a listf ;;
```

```
type 'a listf = | Val of 'a | Fun of ('a -> 'a) * 'a listf
```

Since function values are values which can be manipulated in the language, we can construct values of type *listf*:

```
# let eight_div = (/) 8 ;;
val eight_div : int -> int = <fun>
# let gl = Fun (succ, (Fun (eight_div, Val 4))) ;;
val gl : int listf = Fun (<fun>, Fun (<fun>, Val 4))
```

and functions which pattern-match such values:

```
# let rec compute = function
      Val v → v
    | Fun(f, x) → f (compute x) ;;
val compute : 'a listf -> 'a = <fun>
# compute gl;;
- : int = 3
```

Example: representing trees

Tree structures come up frequently in programming. Recursive types make it easy to define and manipulate such structures. In this subsection, we give two examples of tree structures.

Binary trees We define a binary tree structure whose nodes are labelled with values of a single type by declaring:

```
# type 'a bin_tree =
      Empty
    | Node of 'a bin_tree * 'a * 'a bin_tree ;;
```

We use this structure to define a little sorting program using binary search trees. A binary search tree has the property that all the values in the left branch are less than that of the root, and all those of the right branch are greater. Figure 2.5 gives an example of such a structure over the integers. The empty nodes (constructor `Empty`) are represented there by little squares; the others (constructor `Node`), by a circle in which is inscribed the stored value.

A sorted list is extracted from a binary search tree via an inorder traversal carried out by the following function:

```
# let rec list_of_tree = function
      Empty → []
    | Node(lb, r, rb) → (list_of_tree lb) @ (r :: (list_of_tree rb)) ;;
val list_of_tree : 'a bin_tree -> 'a list = <fun>
```

Figure 2.5: Binary search tree.

To obtain a binary search tree from a list, we define an insert function.

```
# let rec insert x = function
      Empty  →  Node(Empty, x, Empty)
    | Node(lb, r, rb)  →  if x < r then Node(insert x lb, r, rb)
                          else Node(lb, r, insert x rb) ;;
val insert : 'a -> 'a bin_tree -> 'a bin_tree = <fun>
```

The function to transform a list into a tree is obtained by iterating the function `insert`.

```
# let rec tree_of_list = function
        []    →  Empty
     | h :: t  →  insert h (tree_of_list t)  ;;
val tree_of_list : 'a list -> 'a bin_tree = <fun>
```

The sort function is then simply the composition of the functions `tree_of_list` and `list_of_tree`.

```
# let sort x = list_of_tree (tree_of_list x) ;;
val sort : 'a list -> 'a list = <fun>
# sort [5; 8; 2; 7; 1; 0; 3; 6; 9; 4] ;;
- : int list = [0; 1; 2; 3; 4; 5; 6; 7; 8; 9]
```

General planar trees In this part, we use the following predefined functions from the `List` module (see page 217):

- `List.map`: which applies a function to all the elements of a list and returns the list of results;

- `List.fold_left`: which is an equivalent version of the function `fold_left` defined on page 33;
- `List.exists`: which applies a boolean-valued function to all the elements of a list; if one of these applications yields `true` then the result is `true`, otherwise the function returns `false`.

A *general planar tree* is a tree whose number of branches is not fixed *a priori*; to each node is associated a list of branches whose length may vary.

```
# type 'a tree = Empty
                | Node of 'a * 'a tree list ;;
```

The empty tree is represented by the value `Empty`. A leaf is a node without branches either of the form `Node(x,[])`, or of the degenerate form `Node(x, [Empty;Empty; ..])`. It is then relatively easy to write functions to manipulate these trees, e.g., to determine whether an element belongs to a tree or compute the height of the tree.

To test membership of an element `e`, we use the following algorithm: if the tree is empty then `e` does not belong to this tree, otherwise `e` belongs to the tree if and only if either it is equal to the label of the root, or it belongs to one of its branches.

```
# let rec belongs e = function
     Empty → false
   | Node(v, bs) → (e=v) or (List.exists (belongs e) bs) ;;
val belongs : 'a -> 'a tree -> bool = <fun>
```

To compute the height of a tree, we use the following definition: an empty tree has height `0`, otherwise the height of the tree is equal to the height of its highest subtree plus `1`.

```
# let rec height =
   let max_list l = List.fold_left max 0 l in
    function
      Empty → 0
    | Node (_, bs) → 1 + (max_list (List.map height bs)) ;;
val height : 'a tree -> int = <fun>
```

Recursive values which are not functions

Recursive declaration of non-function values allows the construction of circular data structures.

The following declaration constructs a circular list with one element.

```
# let rec l = 1 :: l ;;
val l : int list =
  [1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; ...]
```

Application of a recursive function to such a list risks looping until memory overflows.

```
# size l ;;
Stack overflow during evaluation (looping recursion?).
```

Structural equality remains usable with such lists only when physical equality is first verified:

```
# l=l ;;
- : bool = true
```

In short, if you define a new list, even an equal one, you must not use the structural equality test on pain of seeing your program loop indefinitely. So we don't recommend attempting to evaluate the following example:

```
let rec l2 = 1::l2 in l=l2 ;;
```

On the other hand, physical equality always remains possible.

```
# let rec l2 = 1 :: l2 in l==l2 ;;
- : bool = false
```

The predicate `==` tests equality of an immediate value or sharing of a structured object (equality of the address of the value). We will use it to verify that in traversing a list we don't retraverse a sublist which was already examined. First of all, we define the function `memq`, which verifies the presence of an element in the list by relying on physical equality. It is the counterpart to the function `mem` which tests structural equality; these two functions belong to the module `List`.

```
# let rec memq a l = match l with
      [] → false
    | b :: l → (a==b) or (memq a l) ;;
val memq : 'a -> 'a list -> bool = <fun>
```

The size computation function is redefined, storing the list of lists already examined and halting if a list is encountered a second time.

```
# let special_size l =
    let rec size_aux previous l = match l with
        [] → 0
      | _::l1 → if memq l previous then 0
                        else 1 + (size_aux (l :: previous) l1)
    in size_aux [] l ;;
val special_size : 'a list -> int = <fun>
# special_size [1;2;3;4] ;;
- : int = 4
# special_size l ;;
- : int = 1
# let rec l1 = 1 :: 2 :: l2 and l2 = 1 :: 2 :: l1 in special_size l1 ;;
- : int = 4
```

Typing, domain of definition, and exceptions

The inferred type of a function corresponds to a subset of its domain of definition. Just because a function takes a parameter of type *int* doesn't mean it will know how to compute a value for all integers passed as parameters. In general this problem is dealt with using Objective Caml's exception mechanism. Raising an exception results in a computational interruption which can be intercepted and handled by the program. For this to happen program execution must have registered an exception handler before the computation of the expression which raises this exception.

Partial functions and exceptions

The domain of definition of a function corresponds to the set of values on which the function carries out its computation. There are many mathematical functions which are partial; we might mention division or taking the natural log. This problem also arises for functions which manipulate more complex data structures. Indeed, what is the result of computing the first element of an empty list? In the same way, evaluation of the `factorial` function on a negative integer can lead to an infinite recursion.

Several exceptional situations may arise during execution of a program, for example an attempt to divide by zero. Trying to divide a number by zero will provoke at best a program halt, at worst an inconsistent machine state. The *safety* of a programming language comes from the guarantee that such a situation will not arise for these particular cases. Exceptions are a way of responding to them.

Division of `1` by `0` will cause a specific exception to be raised:

```
# 1/0;;
Uncaught exception: Division_by_zero
```

The message `Uncaught exception: Division_by_zero` indicates on the one hand that the `Division_by_zero` exception has been raised, and on the other hand that it has not been handled. This exception is among the core declarations of the language.

Often, the type of a function does not correspond to its domain of definition when a pattern-matching is not exhaustive, that is, when it does not match all the cases of a given expression. To prevent such an error, Objective Caml prints a message in such a case.

```
# let head l = match l with h::t -> h ;;
Characters 14-36:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val head : 'a list -> 'a = <fun>
```

If the programmer nevertheless keeps the incomplete definition, Objective Caml will use the exception mechanism in the case of an erroneous call to the partial function:

```
# head [] ;;
Uncaught exception: Match_failure("", 14, 36)
```

Finally, we have already met with another predefined exception: `Failure`. It takes an argument of type *string*. One can raise this exception using the function `failwith`. We can use it in this way to complete the definition of our `head`:

```
# let head = function
      [] → failwith "Empty list"
   | h :: t → h;;
val head : 'a list -> 'a = <fun>
# head [] ;;
Uncaught exception: Failure("Empty list")
```

Definition of an exception

In Objective Caml, exceptions belong to a predefined type *exn*. This type is very special since it is an *extensible* sum type: the set of values of the type can be extended by declaring new constructors[9]. This detail lets users define their own exceptions by adding new constructors to the type *exn*.

The syntax of an exception declaration is as follows:

Syntax : **exception** *Name* **;;**

or

Syntax : **exception** *Name* **of** *t* **;;**

Here are some examples of exception declarations:

```
# exception MY_EXN;;
exception MY_EXN
# MY_EXN;;
- : exn = MY_EXN
# exception Depth of int;;
exception Depth of int
# Depth 4;;
- : exn = Depth(4)
```

Thus an exception is a full-fledged language value.

9. Translator's note: Thanks to the new "polymorphic variants" feature of Objective Caml `3.00`, some other sum types can now be extended as well

Warning The names of exceptions are constructors. So they necessarily begin with a capital letter.

```
# exception lowercase ;;
Characters 11-20:
Syntax error
```

Warning Exceptions are monomorphic: they do not have type parameters in the declaration of the type of their argument.

```
# exception Value of 'a ;;
Characters 20-22:
Unbound type parameter 'a
```

A polymorphic exception would permit the definition of functions with an arbitrary return type as we will see further on, page 58.

Raising an exception

The function `raise` is a primitive function of the language. It takes an exception as an argument and has a completely polymorphic return type.

```
# raise ;;
- : exn -> 'a = <fun>
# raise MY_EXN;;
Uncaught exception: MY_EXN
# 1+(raise MY_EXN);;
Uncaught exception: MY_EXN
# raise (Depth 4);;
Uncaught exception: Depth(4)
```

It is not possible to write the function `raise` in Objective Caml. It must be predefined.

Exception handling

The whole point of raising exceptions lies in the ability to handle them and to direct the sequence of computation according to the value of the exception raised. The order of evaluation of an expression thus becomes important for determining which exception is raised. We are leaving the purely functional context, and entering a domain where the order of evaluation of arguments can change the result of a computation, as will be discussed in the following chapter (see page 85).

The following syntactic construct, which computes the value of an expression, permits the handling of an exception raised during this computation:

Syntax :

try *expr* **with**
| p_1 -> $expr_1$
⋮
| p_n -> $expr_n$

If the evaluation of *expr* does not raise any exception, then the result is that of the evaluation of *expr*. Otherwise, the value of the exception which was raised is pattern-matched; the value of the expression corresponding to the first matching pattern is returned. If none of the patterns corresponds to the value of the exception then the latter is propagated up to the next outer **try-with** entered during the execution of the program. Thus pattern matching an exception is always considered to be exhaustive. Implicitly, the last pattern is `| e -> raise e`. If no matching exception handler is found in the program, the system itself takes charge of intercepting the exception and terminates the program while printing an error message.

One must not confuse *computing* an exception (that is, a value of type ***exn***) with *raising* an exception which causes computation to be interrupted. An exception being a value like others, it can be returned as the result of a function.

```
# let return x = Failure x ;;
val return : string -> exn = <fun>
# return "test" ;;
- : exn = Failure("test")
# let my_raise x = raise (Failure x) ;;
val my_raise : string -> 'a = <fun>
# my_raise "test" ;;
Uncaught exception: Failure("test")
```

We note that applying `my_raise` does not return any value while applying `return` returns one of type ***exn***.

Computing with exceptions

Beyond their use for handling exceptional values, exceptions also support a specific programming style and can be the source of optimizations. The following example finds the product of all the elements of a list of integers. We use an exception to interrupt traversal of the list and return the value `0` when we encounter it.

```
# exception Found_zero ;;
exception Found_zero
# let rec mult_rec l = match l with
       [] → 1
    | 0 :: _ → raise Found_zero
    | n :: x → n * (mult_rec x) ;;
val mult_rec : int list -> int = <fun>
# let mult_list l =
     try mult_rec l with Found_zero → 0 ;;
val mult_list : int list -> int = <fun>
# mult_list [1;2;3;0;5;6] ;;
```

```
- : int = 0
```

So all the computations standing by, namely the multiplications by n which follow each of the recursive calls, are abandoned. After encountering **raise**, computation resumes from the pattern-matching under **with**.

Polymorphism and return values of functions

Objective Caml's parametric polymorphism permits the definition of functions whose return type is completely unspecified. For example:

```
# let id x = x ;;
val id : 'a -> 'a = <fun>
```

However, the return type depends on the type of the argument. Thus, when the function `id` is applied to an argument, the type inference mechanism knows how to instantiate the type variable *'a*. So for each particular use, the type of `id` can be determined.

If this were not so, it would no longer make sense to use strong static typing, entrusted with ensuring execution safety. Indeed, a function of completely unspecified type such as *'a -> 'b* would allow any type conversion whatsoever, which would inevitably lead to a run-time error since the physical representations of values of different types are not the same.

Apparent contradiction

However, it is possible in the Objective Caml language to define a function whose return type contains a type variable which does not appear in the types of its arguments. We will consider several such examples and see why such a possibility is not contradictory to strong static typing.

Here is a first example:

```
# let f x = [] ;;
val f : 'a -> 'b list = <fun>
```

This function lets us construct a polymorphic value from anything at all:

```
# f () ;;
- : '_a list = []
# f "anything at all" ;;
- : '_a list = []
```

Nevertheless, the value obtained isn't entirely unspecified: we're dealing with a list. So it can't be used just anywhere.

Here are three examples whose type is the dreaded *'a -> 'b*:

```
# let rec f1 x = f1 x ;;
val f1 : 'a -> 'b = <fun>
# let f2 x = failwith "anything at all" ;;
val f2 : 'a -> 'b = <fun>
# let f3 x = List.hd [] ;;
val f3 : 'a -> 'b = <fun>
```

These functions are not, in fact, dangerous vis-a-vis execution safety, since it isn't possible to use them to construct a value: the first one loops forever, the latter two raise an exception which interrupts the computation.

Similarly, it is in order to prevent functions of type *'a -> 'b* from being defined that new exception constructors are forbidden from having arguments whose type contains a variable.

Indeed, if one could declare a polymorphic exception `Poly_exn` of type *'a -> exn*, one could then write the function:

```
let f = function
   0 → raise (Poly_exn false)
 | n → n+1 ;;
```

The function `f` being of type *int -> int* and `Poly_exn` being of type *'a -> exn*, one could then define:

```
let g n = try f n with Poly_exn x → x+1 ;;
```

This function is equally well-typed (since the argument of `Poly_exn` may be arbitrary) and now, evaluation of `(g 0)` would end up in an attempt to add an integer and a boolean!

Desktop Calculator

To understand how a program is built in Objective Caml, it is necessary to develop one. The chosen example is a desktop calculator—that is, the simplest model, which only works on whole numbers and only carries out the four standard arithmetic operations.

To begin, we define the type *key* to represent the keys of a pocket calculator. The latter has fifteen keys, namely: one for each operation, one for each digit, and the `=` key.

```
# type key = Plus | Minus | Times | Div | Equals | Digit of int ;;
```

We note that the numeric keys are gathered under a single constructor *Digit* taking an integer argument. In fact, some values of type *key* don't actually represent a key. For example, `(Digit 32)` is a possible value of type *key*, but doesn't represent any of the calculator's keys.

So we write a function `valid` which verifies that its argument corresponds to a calculator key. The type of this function is *key -> bool*, that is, it takes a value of type *key* as argument and returns a value of type *bool*.

The first step is to define a function which verifies that an integer is included between 0 and 9. We declare this function under the name `is_digit`:

```
# let is_digit = function x → (x>=0) && (x<=9) ;;
val is_digit : int -> bool = <fun>
```

We then define the function `valid` by pattern-matching over its argument of type ***key***:

```
#   let valid ky = match ky with
      Digit n → is_digit n
    | _ → true ;;
val valid : key -> bool = <fun>
```

The first pattern is applied when the argument of `valid` is a value made with the ***Digit*** constructor; in this case, the argument of ***Digit*** is tested by the function `is_digit`. The second pattern is applied to every other kind of value of type ***key***. Recall that thanks to typing, the value being matched is necessarily of type ***key***.

Before setting out to code the calculator mechanism, we will specify a *model* allowing us to describe from a formal point of view the reaction to the activation of one of the device's keys. We will consider a pocket calculator to have four registers in which are stored respectively the last computation done, the last key activated, the last operator activated, and the number printed on the screen. The set of these four registers is called the state of the calculator; it is modified by each keypress on the keypad. This modification is called a transition and the theory governing this kind of mechanism is that of automata. A state will be represented in our program by a record type:

```
# type state = {
    lcd : int; (* last computation done   *)
    lka : key; (* last key activated      *)
    loa : key; (* last operator activated *)
    vpr : int  (* value printed           *)
  } ;;
```

Figure 2.6 gives an example of a sequence of transitions.

	state	key
	$(0, =, =, 0)$	3
$\longrightarrow$	$(0, 3, =, 3)$	$+$
$\longrightarrow$	$(3, +, +, 3)$	2
$\longrightarrow$	$(3, 2, +, 2)$	1
$\longrightarrow$	$(3, 1, +, 21)$	$\times$
$\longrightarrow$	$(24, *, *, 24)$	2
$\longrightarrow$	$(24, 2, *, 2)$	$=$
$\longrightarrow$	$(48, =, =, 48)$	

Figure 2.6: Transitions for $3 + 21 * 2 =$.

In what follows we need the function `evaluate` which takes two integers and a value of type `key` containing an operator and which returns the result of the operation corresponding to the key, applied to the integers. This function is defined by pattern-matching over its last argument, of type ***key***:

```
# let evaluate x y ky = match ky with
      Plus   → x + y
    | Minus  → x - y
    | Times  → x * y
    | Div    → x / y
    | Equals → y
    | Digit _ → failwith "evaluate : no op";;
val evaluate : int -> int -> key -> int = <fun>
```

Now we give the definition of the transition function by enumerating all possible cases. We assume that the current state is the quadruplet $(a, b, \oplus, d)$:

- a key with digit x is pressed, then there are two cases to consider:
  - the last key pressed was also a digit. So it is a number which the user of the pocket calculator is in the midst of entering; consequently the digit x must be affixed to the printed value, i.e., replacing it with $d \times 10 + x$. The new state is:
    $$(a, \texttt{(Digit x)}, \oplus, d \times 10 + x)$$
  - the last key pressed was not a digit. So it is the start of a new number which is being entered. The new state is:
    $$(a, \texttt{(Digit x)}, \oplus, x)$$
- a key with operator $\otimes$ has been pressed, the second operand of the operation has thus been completely entered and the calculator has to deal with carrying out this operation. It is to this end that the last operation (here $\oplus$) is stored. The new state is:
  $$(\oplus d, \otimes, \otimes, a \oplus d)$$

To write the function `transition`, it suffices to translate the preceding definition word for word into Objective Caml: the definition by cases becomes a definition by pattern-matching over the key passed as an argument. The case of a key, which itself is made up of two cases, is handled by the local function `digit_transition` by pattern-matching over the last key activated.

```
# let transition st ky =
    let digit_transition n = function
        Digit _ → { st with lka=ky; vpr=st.vpr*10+n }
      | _       → { st with lka=ky; vpr=n }
    in
       match ky with
           Digit p  →  digit_transition p st.lka
         | _        →  let res = evaluate st.lcd st.vpr st.loa
                       in { lcd=res; lka=ky; loa=ky; vpr=res }  ;;
```

```
val transition : state -> key -> state = <fun>
```
This function takes a *state* and a *key* and computes the new *state*.

We can now test this program on the previous example:
```
# let initial_state = { lcd=0; lka=Equals; loa=Equals; vpr=0 } ;;
val initial_state : state = {lcd=0; lka=Equals; loa=Equals; vpr=0}
# let state2 = transition initial_state (Digit 3) ;;
val state2 : state = {lcd=0; lka=Digit 3; loa=Equals; vpr=3}
# let state3 = transition state2 Plus ;;
val state3 : state = {lcd=3; lka=Plus; loa=Plus; vpr=3}
# let state4 = transition state3 (Digit 2) ;;
val state4 : state = {lcd=3; lka=Digit 2; loa=Plus; vpr=2}
# let state5 = transition state4 (Digit 1) ;;
val state5 : state = {lcd=3; lka=Digit 1; loa=Plus; vpr=21}
# let state6 = transition state5 Times ;;
val state6 : state = {lcd=24; lka=Times; loa=Times; vpr=24}
# let state7 = transition state6 (Digit 2) ;;
val state7 : state = {lcd=24; lka=Digit 2; loa=Times; vpr=2}
# let state8 = transition state7 Equals ;;
val state8 : state = {lcd=48; lka=Equals; loa=Equals; vpr=48}
```

This run can be written in a more concise way using a function applying a sequence of transitions corresponding to a list of keys passed as an argument.

```
# let transition_list st ls = List.fold_left transition st ls ;;
val transition_list : state -> key list -> state = <fun>
# let example = [ Digit 3; Plus; Digit 2; Digit 1; Times; Digit 2; Equals ]
  in transition_list initial_state example ;;
- : state = {lcd=48; lka=Equals; loa=Equals; vpr=48}
```

Exercises

Merging two lists

1. Write a function `merge_i` which takes as input two integer lists sorted in increasing order and returns a new sorted list containing the elements of the first two.
2. Write a general function `merge` which takes as argument a comparison function and two lists sorted in this order and returns the list merged in the same order. The comparison function will be of type *'a → 'a → bool*.
3. Apply this function to two integer lists sorted in decreasing order, then to two string lists sorted in decreasing order.
4. What happens if one of the lists is not in the required decreasing order?

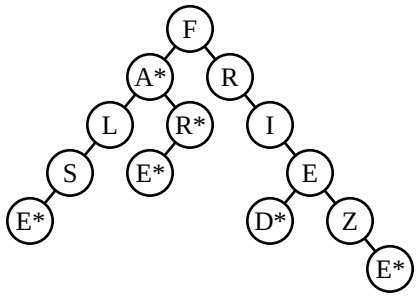
5. Write a new *list* type in the form of a record containing three fields: the conventional list, an order function and a boolean indicating whether the list is in that order.
6. Write the function `insert` which adds an element to a list of this type.
7. Write a function `sort` which insertion sorts the elements of a list.
8. Write a new function `merge` for these lists.

Lexical trees

Lexical trees (or *tries*) are used for the representation of dictionaries.

```
# type lex_node = Letter of char * bool * lex_tree
  and  lex_tree = lex_node  list;;
# type word = string;;
```

The boolean value in *lex_node* marks the end of a word when it equals **true**. In such a structure, the sequence of words "fa, false, far, fare, fried, frieze" is stored in the following way:



An asterisk (*) marks the end of a word.

1. Write the function `exists` which tests whether a word belongs to a dictionary of type *lex_tree*.
2. Write a function `insert` which takes a word and a dictionary and returns a new dictionary which additionally contains this word. If the word is already in the dictionary, it is not necessary to insert it.
3. Write a function `construct` which takes a list of words and constructs the corresponding dictionary.
4. Write a function `verify` which takes a list of words and a dictionary and returns the list of words not belonging to this dictionary.
5. Write a function `select` which takes a dictionary and a length and returns the set of words of this length.

Graph traversal

We define a type *'a graph* representing directed graphs by adjacency lists containing for each vertex the list of its successors:

```
# type 'a graph = ( 'a * 'a  list) list ;;
```

1. Write a function `insert_vtx` which inserts a vertex into a graph and returns the new graph.
2. Write a function `insert_edge` which adds an edge to a graph already possessing these two vertices.
3. Write a function `has_edges_to` which returns all the vertices following directly from a given vertex.
4. Write a function `has_edges_from` which returns the list of all the vertices leading directly to a given vertex.

Summary

This chapter has demonstrated the main features of functional programming and parametric polymorphism, which are two essential features of the Objective Caml language. The syntax of the expressions in the functional core of the language as well as those of the types which have been described allowed us to develop our first programs. Moreover, the profound difference between the type of a function and its domain of definition was underlined. Introducing the exception mechanism allowed us to resolve this problem and already introduces a new programming style in which one specifies how computations should unfold.

To learn more

The computation model for functional languages is λ-calculus, which was invented by Alonzo Church in 1932. Church's goal was to define a notion of effective computability through the medium of λ-definability. Later, it became apparent that the notion thus introduced was equivalent to the notions of computability in the sense of Turing (Turing machine) and Gödel-Herbrand (recursive functions). This coïncidence leads one to think that there exists a universal notion of computability, independent of particular formalisms: this is Church's thesis. In this calculus, the only two constructions are abstraction and application. Data structures (integers, booleans, pairs, ...) can be coded by λ-termes.

Functional languages, of which the first representative was Lisp, implement this model and extend it mainly with more efficient data structures. For the sake of efficiency, the first functional languages implemented physical modifications of memory, which among other things forced the evaluation strategy to be immediate, or strict, evaluation. In this strategy, the arguments of functions are evaluated before being passed to the

function. It is in fact later, for other languages such as Miranda, Haskell, or LML, that the strategy of delayed (lazy, or call-by-need) evaluation was implemented for pure functional languages.

Static typing, with type inference, was promoted by the ML family at the start of the 80's. The web page

Link: http://www.pps.jussieu.fr/~cousinea/Caml/caml_history.html

presents a historical overview of the ML language. Its computation model is typed λ-calculus, a subset of λ-calculus. It guarantees that no type error will occur during program execution. Nevertheless "completely correct" programs can be rejected by ML's type system. These cases seldom arise and these programs can always be rewritten in such a way as to conform to the type system.

The two most-used functional languages are Lisp and ML, representatives of impure functional languages. To deepen the functional approach to programming, the books [ASS96] and [CM98] each present a general programming course using the languages Scheme (a dialect of Lisp) and Caml-Light, respectively.