3

Imperative Programming

In contrast to functional programming, in which you calculate a value by applying a function to its arguments without caring how the operations are carried out, imperative programming is closer to the machine representation, as it introduces memory state which the execution of the program's actions will modify. We call these actions of programs *instructions*, and an imperative program is a list, or *sequence*, of instructions. The execution of each operation can alter the memory state. We consider input-output actions to be modifications of memory, video memory, or files.

This style of programming is directly inspired by assembly programming. You find it in the earliest general-purpose programming languages (Fortran, C, Pascal, etc.). In Objective Caml the following elements of the language fit into this model:

- modifiable data structures, such as arrays, or records with mutable fields;
- input-output operations;
- control structures such as loops and exceptions.

Certain algorithms are easier to write in this programming style. Take for instance the computation of the product of two matrices. Even though it is certainly possible to translate it into a purely functional version, in which lists replace vectors, this is neither natural nor efficient compared to an imperative version.

The motivation for the integration of imperative elements into a functional language is to be able to write certain algorithms in this style when it is appropriate. The two principal disadvantages, compared to the purely functional style, are:

- complicating the type system of the language, and rejecting certain programs which would otherwise be considered correct;
- having to keep track of the memory representation and of the order of calculations.

Nevertheless, with a few guidelines in writing programs, the choice between several programming styles offers the greatest flexibility for writing algorithms, which is the principal objective of any programming language. Besides, a program written in a style which is close to the algorithm used will be simpler, and hence will have a better chance of being correct (or at least, rapidly correctable).

For these reasons, the Objective Caml language has some types of data structures whose values are physically modifiable, structures for controlling the execution of programs, and an I/O library in an imperative style.

Plan of the Chapter

This chapter continues the presentation of the basic elements of the Objective Caml language begun in the previous chapter, but this time focusing on imperative constructions. There are five sections. The first is the most important; it presents the different modifiable data structures and describes their memory representation. The second describes the basic I/O of the language, rather briefly. The third section is concerned with the new iterative control structures. The fourth section discusses the impact of imperative features on the execution of a program, and in particular on the order of evaluation of the arguments of a function. The final section returns to the calculator example from the last chapter, to turn it into a calculator with a memory.

Modifiable Data Structures

Values of the following types: vectors, character strings, records with mutable fields, and references are the data structures whose parts can be physically modified.

We have seen that an Objective Caml variable bound to a value keeps this value to the end of its lifetime. You can only modify this binding with a redefinition—in which case we are not really talking about the "same" variable; rather, a new variable of the same name now masks the old one, which is no longer directly accessible, but which remains unchanged. With modifiable values, you can change the value associated with a variable without having to redeclare the latter. You have access to the value of a variable for writing as well as for reading.

Vectors

Vectors, or one dimensional arrays, collect a known number of elements of the same type. You can write a vector directly by listing its values between the symbols `[|` and `|]`, separated by semicolons as for lists.

```
# let v = [| 3.14; 6.28; 9.42 |] ;;
val v : float array = [|3.14; 6.28; 9.42|]
```

The creation function `Array.create` takes the number of elements in the vector and an initial value, and returns a new vector.

```
# let v = Array.create  3  3.14;;
val v : float array = [|3.14; 3.14; 3.14|]
```

To access or modify a particular element, you give the index of that element:

Syntax : *expr*$_1$ **.** **(** *expr*$_2$ **)**

Syntax : *expr*$_1$ **.** **(** *expr*$_2$ **)** **<-** *expr*$_3$

expr$_1$ should be a vector (type ***array***) whose values have type *expr*$_3$. The expression *expr*$_2$ must, of course, have type ***int***. The modification is an expression of type ***unit***. The first element of a vector has index 0 and the index of the last element is the length of the vector minus 1. The parentheses around the index expression are required.

```
# v.(1) ;;
- : float = 3.14
# v.(0) <- 100.0 ;;
- : unit = ()
# v ;;
- : float array = [|100; 3.14; 3.14|]
```

If the index used to access an element in an array is outside the range of indices of the array, an exception is raised at the moment of access.

```
# v.(-1) +. 4.0;;
Uncaught exception: Invalid_argument("Array.get")
```

This check is done during program execution, which can slow it down. Nevertheless it is essential, in order to avoid writing to memory outside the space allocated to a vector, which would cause serious execution errors.

The functions for manipulating arrays are part of the `Array` module in the standard library. We'll describe them in chapter 8 (page 217). In the examples below, we will use the following three functions from the `Array` module:

- `create` which creates an array of the given size with the given initial value;
- `length` which gives the length of a vector;
- `append` which concatenates two vectors.

Sharing of Values in a Vector

All the elements of a vector contain the value that was passed in when it was created. This implies a sharing of this value, if it is a structured value. For example, let's create a matrix as a vector of vectors using the function `create` from the `Array` module.

```
# let v = Array.create 3 0;;
val v : int array = [|0; 0; 0|]
# let m = Array.create 3 v;;
```

```
val m : int array array = [|[|0; 0; 0|]; [|0; 0; 0|]; [|0; 0; 0|]|]
```

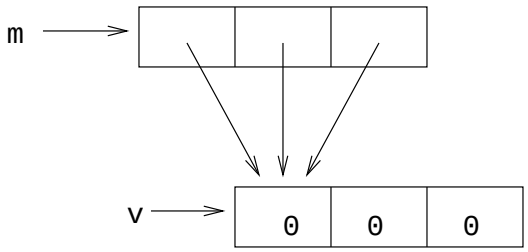

Figure 3.1: Memory representation of a vector sharing its elements.

If you modify one of the fields of vector `v`, which was used in the creation of `m`, then you automatically modify all the "rows" of the matrix together (see figures 3.1 and 3.2).

```
# v.(0) <- 1;;
- : unit = ()
# m;;
- : int array array = [|[|1; 0; 0|]; [|1; 0; 0|]; [|1; 0; 0|]|]
```

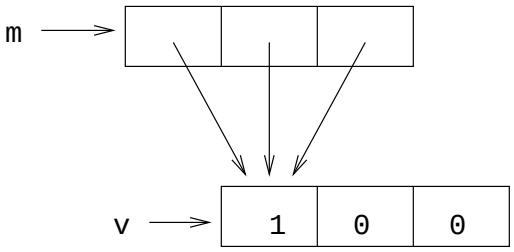

Figure 3.2: Modification of shared elements of a vector.

Duplication occurs if the initialization value of the vector (the second argument passed to `Array.create`) is an *atomic value* and there is sharing if this value is a structured value.

Values whose size does not exceed the standard size of Objective Caml values—that is, the memory word—are called atomic values. These are the integers, characters, booleans, and constant constructors. The other values—structured values—are represented by a pointer into a memory area. This distinction is detailed in chapter 9 (page 247).

Vectors of floats are a special case. Although floats are structured values, the creation of a vector of floats causes the the initial value to be copied. This is for reasons of optimization. Chapter 12, on the interface with the C language (page 315), describes this special case.

Non-Rectangular Matrices

A matrix, a vector of vectors, does not need not to be rectangular. In fact, nothing stops you from replacing one of the vector elements with a vector of a different length. This is useful to limit the size of such a matrix. The following value `t` constructs a triangular matrix for the coefficients of Pascal's triangle.

```
# let t = [|
            [|1|];
            [|1; 1|];
            [|1; 2; 1|];
            [|1; 3; 3; 1|];
            [|1; 4; 6; 4; 1|];
            [|1; 5; 10; 10; 5; 1|]
          |] ;;
val t : int array array =
  [|[|1|]; [|1; 1|]; [|1; 2; 1|]; [|1; 3; 3; 1|]; [|1; 4; 6; 4; ...|]; ...|]
# t.(3) ;;
- : int array = [|1; 3; 3; 1|]
```

In this example, the element of vector `t` with index i is a vector of integers with size $i + 1$. To manipulate such matrices, you have to calculate the size of each element vector.

Copying Vectors

When you copy a vector, or when you concatenate two vectors, the result obtained is a new vector. A modification of the original vectors does not result in the modification of the copies, unless, as usual, there are shared values.

```
# let v2 = Array.copy v ;;
val v2 : int array = [|1; 0; 0|]
# let m2 = Array.copy m ;;
val m2 : int array array = [|[|1; 0; 0|]; [|1; 0; 0|]; [|1; 0; 0|]|]
# v.(1)<- 352;;
- : unit = ()
# v2;;
- : int array = [|1; 0; 0|]
# m2 ;;
- : int array array = [|[|1; 352; 0|]; [|1; 352; 0|]; [|1; 352; 0|]|]
```

We notice in this example that copying `m` only copies the pointers to `v`. If one of the elements of `v` is modified, `m2` is modified too.

Concatenation creates a new vector whose size is equal to the sum of the sizes of the two others.

```
# let mm = Array.append m m ;;
val mm : int array array =
  [|[|1; 352; 0|]; [|1; 352; 0|]; [|1; 352; 0|]; [|1; 352; 0|];
    [|1; 352; ...|]; ...|]
# Array.length mm ;;
```

```
- : int = 6
# m.(0) <- Array.create 3 0 ;;
- : unit = ()
# m ;;
- : int array array = [|[|0; 0; 0|]; [|1; 352; 0|]; [|1; 352; 0|]|]
# mm ;;
- : int array array =
[|[|1; 352; 0|]; [|1; 352; 0|]; [|1; 352; 0|]; [|1; 352; 0|];
  [|1; 352; ...|]; ...|]
```

On the other hand, modification of `v`, a value shared by `m` and `mm`, does affect both these matrices.

```
# v.(1) <- 18 ;;
- : unit = ()
# mm;;
- : int array array =
[|[|1; 18; 0|]; [|1; 18; 0|]; [|1; 18; 0|]; [|1; 18; 0|]; [|1; 18; ...|];
  ...|]
```

Character Strings

Character strings can be considered a special case of vectors of characters. Nevertheless, for efficient memory usage[1] their type is specialized. Moreover, access to their elements has a special syntax:

Syntax : $expr_1$ **.** **[**$expr_2$**]**

The elements of a character string can be physically modified:

Syntax : $expr_1$ **.** **[**$expr_2$**]** **<-** $expr_3$

```
# let s = "hello";;
val s : string = "hello"
# s.[2];;
- : char = 'l'
# s.[2]<-'Z';;
- : unit = ()
# s;;
- : string = "heZlo"
```

1. A 32-bit word contains four characters coded as bytes

Mutable Fields of Records

Fields of a record can be declared mutable. All you have to do is to show this in the declaration of the type of the record using the keyword **mutable**.

Syntax : **type** *name* **=** { ... **; mutable** $name_i$ **:** t **;** ... }

Here is a small example defining a record type for points in the plane:

```
# type point = { mutable xc : float; mutable yc : float } ;;
type point = { mutable xc: float; mutable yc: float }
# let p = { xc = 1.0; yc = 0.0 } ;;
val p : point = {xc=1; yc=0}
```

Thus the value of a field which is declared **mutable** can be modified using the syntax:

Syntax : $expr_1$ **.** *name* **<-** $expr_2$

The expression $expr_1$ should be a record type which has the field *name*. The modification operator returns a value of type ***unit***.

```
# p.xc <- 3.0 ;;
- : unit = ()
# p ;;
- : point = {xc=3; yc=0}
```

We can write a function for moving a point by modifying its components. We use a local declaration with pattern matching in order to sequence the side-effects.

```
# let moveto p dx dy =
    let () = p.xc <- p.xc +. dx
    in p.yc <- p.yc +. dy ;;
val moveto : point -> float -> float -> unit = <fun>
# moveto p 1.1 2.2 ;;
- : unit = ()
# p ;;
- : point = {xc=4.1; yc=2.2}
```

It is possible to mix mutable and non-mutable fields in the definition of a record. Only those specified as **mutable** may be modified.

```
# type t = { c1 : int; mutable c2 : int } ;;
type t = { c1: int; mutable c2: int }
# let r = { c1 = 0; c2 = 0 } ;;
val r : t = {c1=0; c2=0}
# r.c1 <- 1 ;;
Characters 0-9:
The label c1 is not mutable
# r.c2 <- 1 ;;
- : unit = ()
```

```
# r ;;
- : t = {c1=0; c2=1}
```

On page 82 we give an example of using records with modifiable fields and arrays to implement a stack structure.

References

Objective Caml provides a polymorphic type `ref` which can be seen as the type of a pointer to any value; in Objective Caml terminology we call it a *reference* to a value. A referenced value can be modified. The type *`ref`* is defined as a record with one modifiable field:

```
type 'a ref = {mutable contents:'a}
```

This type is provided as a syntactic shortcut. We construct a reference to a value using the function `ref`. The referenced value can be reached using the prefix function **`(!)`**. The function modifying the content of a reference is the infix function **`(:=)`**.

```
# let x = ref 3 ;;
val x : int ref = {contents=3}
# x ;;
- : int ref = {contents=3}
# !x ;;
- : int = 3
# x := 4 ;;
- : unit = ()
# !x ;;
- : int = 4
# x := !x+1 ;;
- : unit = ()
# !x ;;
- : int = 5
```

Polymorphism and Modifiable Values

The type *`ref`* is parameterized. This is what lets us use it to create references to values of any type whatever. However, it is necessary to place certain restrictions on the type of referenced values; we cannot allow the creation of a reference to a value with a polymorphic type without taking some precautions.

Let us suppose that there were no restriction; then someone could declare:

```
let x = ref [] ;;
```

Then the variable x would have type *'a list ref* and its value could be modified in a way which would be inconsistent with the strong static typing of Objective Caml:

```
x := 1 :: !x ;;
x := true :: !x ;;
```

Thus we would have one and the same variable having type *int list* at one moment and *bool list* the next.

In order to avoid such a situation, Objective Caml's type inference mechanism uses a new category of type variables: *weak type variables*. Syntactically, they are distinguished by the underscore character which prefixes them.

```
# let x = ref [] ;;
val x : '_a list ref = {contents=[]}
```

The type variable *'_a* is not a type parameter, but an unknown type awaiting instantiation; the first use of x after its declaration fixes the value that *'_a* will take in all types that depend on it, permanently.

```
# x := 0::!x ;;
- : unit = ()
# x ;;
- : int list ref = {contents=[0]}
```

From here onward, the variable x has type *int list ref*.

A type containing an unknown is in fact monomorphic even though its type has not been specified. It is not possible to instantiate this unknown with a polymorphic type.

```
# let x = ref [] ;;
val x : '_a list ref = {contents=[]}
# x := (function y → ())::!x ;;
- : unit = ()
# x ;;
- : ('_a -> unit) list ref = {contents=[<fun>]}
```

In this example, even though we have instantiated the unknown type with a type which is *a priori* polymorphic (*'a -> unit*), the type has remained monomorphic with a new unknown type.

This restriction of polymorphism applies not only to references, but to any value containing a modifiable part: vectors, records having at least one field declared **mutable**, etc. Thus all the type parameters, even those which have nothing to do with a modifiable part, are weak type variables.

```
# type ('a,'b) t = { ch1 :'a list ; mutable ch2 : 'b list } ;;
type ('a, 'b) t = { ch1: 'a list; mutable ch2: 'b list }
# let x = { ch1 = [] ; ch2 = [] } ;;
val x : ('_a, '_b) t = {ch1=[]; ch2=[]}
```

Warning This modification of the typing of application has consequences for pure functional programs.

Likewise, when you apply a polymorphic value to a polymorphic function, you get a weak type variable, because you must not exclude the possibility that the function may construct physically modifiable values. In other words, the result of the application is always monomorphic.

```
# (function x → x) [] ;;
- : '_a list = []
```

You get the same result with partial application:

```
# let f a b = a ;;
val f : 'a -> 'b -> 'a = <fun>
# let g = f 1 ;;
val g : '_a -> int = <fun>
```

To get a polymorphic type back, you have to abstract the second argument of `f` and then apply it:

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

In effect, the expression which defines `h` is the functional expression **function** *x* → *f* `1` *x*. Its evaluation produces a closure which does not risk producing a side effect, because the body of the function is not evaluated.

In general, we distinguish so-called "non-expansive" expressions, whose calculation we are sure carries no risk of causing a side effect, from other expressions, called "expansive." Objective Caml's type system classifies expressions of the language according to their syntactic form:

- "non-expansive" expressions include primarily variables, constructors of non-mutable values, and abstractions;
- "expansive" expressions include primarily applications and constructors of modifiable values. We can also include here control structures like conditionals and pattern matching.

Input-Output

Input-output functions do calculate a value (often of type *unit*) but during their calculation they cause a modification of the state of the input-output peripherals: modification of the state of the keyboard *buffer*, outputting to the screen, writing in a file, or modification of a read pointer. The following two types are predefined: *in_channel* and *out_channel* for, respectively, input channels and output channels. When an end of file is met, the exception `End_of_file` is raised. Finally, the following three constants correspond to the standard channels for input, output, and error in Unix fashion: `stdin`, `stdout`, and `stderr`.

Channels

The input-output functions from the Objective Caml standard library manipulate *communication channels*: values of type ***in_channel*** or ***out_channel***. Apart from the three standard predefined values, the creation of a channel uses one of the following functions:

```
# open_in;;
- : string -> in_channel = <fun>
# open_out;;
- : string -> out_channel = <fun>
```

open_in opens the file if it exists[2], and otherwise raises the exception Sys_error. open_out creates the specified file if it does not exist or truncates it if it does.

```
# let ic = open_in "koala";;
val ic : in_channel = <abstr>
# let oc = open_out "koala";;
val oc : out_channel = <abstr>
```

The functions for closing channels are:

```
# close_in ;;
- : in_channel -> unit = <fun>
# close_out ;;
- : out_channel -> unit = <fun>
```

Reading and Writing

The most general functions for reading and writing are the following:

```
# input_line ;;
- : in_channel -> string = <fun>
# input ;;
- : in_channel -> string -> int -> int -> int = <fun>
# output ;;
- : out_channel -> string -> int -> int -> unit = <fun>
```

- ***input_line ic***: reads from input channel ic all the characters up to the first carriage return or end of file, and returns them in the form of a list of characters (excluding the carriage return).
- ***input ic s p l***: attempts to read l characters from an input channel ic and stores them in the list s starting from the p$^{\text{th}}$ character. The number of characters actually read is returned.
- ***output oc s p l***: writes on an output channel oc part of the list s, starting at the p-th character, with length l.

2. With appropriate read permissions, that is.

The following functions read from standard input or write to standard output:

```
# read_line ;;
- : unit -> string = <fun>
# print_string ;;
- : string -> unit = <fun>
# print_newline ;;
- : unit -> unit = <fun>
```

Other values of simple types can also be read directly or appended. These are the values of types which can be converted into lists of characters.

Local declarations and order of evaluation We can simulate a sequence of printouts with expressions of the form `let x = e1 in e2`. Knowing that, in general, `x` is a local variable which can be used in `e2`, we know that `e1` is evaluated first and then comes the turn of `e2`. If the two expressions are imperative functions whose results are `()` but which have side effects, then we have executed them in the right order. In particular, since we know the return value of `e1`—the constant () of type *unit*—we get a sequence of printouts by writing the sequence of nested declarations which pattern match on ().

```
# let () = print_string "and one," in
  let () = print_string " and two," in
  let () = print_string " and three" in
     print_string " zero";;
and one, and two, and three zero- : unit = ()
```

Example: Higher/Lower

The following example concerns the game "Higher/Lower" which consists of choosing a number which the user must guess at. The program indicates at each turn whether the chosen number is smaller or bigger than the proposed number.

```
#  let rec hilo n =
      let () = print_string "type a number: " in
      let i = read_int ()
      in
        if i = n then
          let () = print_string "BRAVO" in
          let () = print_newline ()
          in print_newline ()
        else
          let () =
            if i < n then
```

```
              let () = print_string "Higher"
              in print_newline ()
            else
              let () = print_string "Lower"
              in print_newline ()
          in hilo n ;;
val hilo : int -> unit = <fun>
```

Here is an example session:

```
# hilo 64;;
type a number: 88
Lower
type a number: 44
Higher
type a number: 64
BRAVO

- : unit = ()
```

Control Structures

Input-output and modifiable values produce side-effects. Their use is made easier by an imperative programming style furnished with new control structures. We present in this section the sequence and iteration structures.

We have already met the conditional control structure on page 18, whose abbreviated form **if then** patterns itself on the imperative world. We will write, for example:

```
# let n = ref 1 ;;
val n : int ref = {contents=1}
# if !n > 0 then n := !n - 1 ;;
- : unit = ()
```

Sequence

The first of the typically imperative structures is the *sequence*. This permits the left-to-right evaluation of a sequence of expressions separated by semicolons.

Syntax : $expr_1$ **;** ... **;** $expr_n$

A sequence of expressions is itself an expression, whose value is that of the last expression in the sequence (here, $expr_n$). Nevertheless, all the expressions are evaluated, and in particular their side-effects are taken into account.

```
# print_string "2 = "; 1+1 ;;
```

```
2 = - : int = 2
```

With side-effects, we get back the usual construction of imperative languages.

```
# let x = ref 1 ;;
val x : int ref = {contents=1}
# x:=!x+1 ; x:=!x*4 ; !x ;;
- : int = 8
```

As the value preceding a semicolon is discarded, Objective Caml gives a warning when it is not of type *unit*.

```
# print_int 1; 2 ; 3 ;;
Characters 14-15:
Warning: this expression should have type unit.
1- : int = 3
```

To avoid this message, you can use the function `ignore`:

```
# print_int 1; ignore 2; 3 ;;
1- : int = 3
```

A different message is obtained if the value has a functional type, as Objective Caml suspects that you have forgotten a parameter of a function.

```
# let g x y = x := y ;;
val g : 'a ref -> 'a -> unit = <fun>
# let a = ref 10;;
val a : int ref = {contents=10}
# let u = 1 in g a ; g a u ;;
Characters 13-16:
Warning: this function application is partial,
maybe some arguments are missing.
- : unit = ()
# let u = !a in ignore (g a) ; g a u ;;
- : unit = ()
```

As a general rule we parenthesize sequences to clarify their scope. Syntactically, parenthesizing can take two forms:

Syntax : **(** *expr* **)**

Syntax : **begin** *expr* **end**

We can now write the Higher/Lower program from page 78 more naturally:

```
# let rec hilo n =
    print_string "type a number: ";
    let i = read_int () in
```

```
    if i = n then print_string "BRAVO\n\n"
     else
       begin
         if i < n then print_string "Higher\n" else print_string "Lower\n"  ;
         hilo n
       end ;;
val hilo : int -> unit = <fun>
```

Loops

The iterative control structures are also from outside the functional world. The conditional expression for repeating, or leaving, a loop does not make sense unless there can be a physical modification of the memory which permits its value to change. There are two iterative control structures in Objective Caml: the **for** loop for a bounded iteration and the **while** loop for a non-bounded iteration. The loop structures themselves are expressions of the language. Thus they return a value: the constant **()** of type *unit*.

The **for** loop can be rising (**to**) or falling (**downto**) with a step of one.

Syntax :

for *name* **=** $expr_1$ **to** $expr_2$ **do** $expr_3$ **done**
for *name* **=** $expr_1$ **downto** $expr_2$ **do** $expr_3$ **done**

The expressions $expr_1$ and $expr_2$ are of type *int*. If $expr_3$ is not of type *unit*, the compiler produces a warning message.

```
# for i=1 to 10 do  print_int i; print_string " " done; print_newline() ;;
1 2 3 4 5 6 7 8 9 10
- : unit = ()
# for i=10 downto 1 do print_int i; print_string " " done; print_newline() ;;
10 9 8 7 6 5 4 3 2 1
- : unit = ()
```

The non-bounded loop is the "while" loop whose syntax is:

Syntax : **while** $expr_1$ **do** $expr_2$ **done**

The expression $expr_1$ should be of type *bool*. And, as for the **for** loop, if $expr_2$ is not of type *unit*, the compiler produces a warning message.

```
# let r = ref 1
  in while !r < 11  do
       print_int !r ;
       print_string " " ;
       r := !r+1
     done ;;
1 2 3 4 5 6 7 8 9 10 - : unit = ()
```

It is important to understand that loops are expressions like the previous ones which calculate the value **()** of type *unit*.

```
# let f () = print_string "-- end\n" ;;
val f : unit -> unit = <fun>
# f (for i=1 to 10 do  print_int i; print_string " " done) ;;
1 2 3 4 5 6 7 8 9 10 -- end
- : unit = ()
```

Note that the string `"-- end\n"` is output after the integers from 1 to 10 have been printed: this is a demonstration that the arguments (here the loop) are evaluated before being passed to the function.

In imperative programming, the body of a loop ($expr_2$) does not calculate a value, but advances by side effects. In Objective Caml, when the body of a loop is not of type *unit* the compiler prints a warning, as for the sequence:

```
# let s = [5; 4; 3; 2; 1; 0] ;;
val s : int list = [5; 4; 3; 2; 1; 0]
#  for i=0 to 5 do List.tl s done ;;
Characters 17-26:
Warning: this expression should have type unit.
- : unit = ()
```

Example: Implementing a Stack

The data structure *'a stack* will be implemented in the form of a record containing an array of elements and the first free position in this array. Here is the corresponding type:

```
# type 'a stack = { mutable ind:int; size:int; mutable elts : 'a array } ;;
```

The field `size` contains the maximal size of the stack.

The operations on these stacks will be `init_stack` for the initialization of a stack, `push` for pushing an element onto a stack, and `pop` for returning the top of the stack and popping it off.

```
# let init_stack n = {ind=0; size=n; elts =[||]} ;;
val init_stack : int -> 'a stack = <fun>
```

This function cannot create a non-empty array, because you would have to provide it with the value with which to construct it. This is why the field `elts` gets an empty array.

Two exceptions are declared to guard against attempts to pop an empty stack or to add an element to a full stack. They are used in the functions `pop` and `push`.

```
# exception Stack_empty ;;
# exception Stack_full ;;

# let pop p =
    if p.ind = 0 then raise Stack_empty
    else (p.ind <- p.ind - 1; p.elts.(p.ind)) ;;
```

```
val pop : 'a stack -> 'a = <fun>
# let push e p =
    if p.elts = [||] then
      (p.elts <- Array.create p.size e;
       p.ind <- 1)
    else if p.ind >= p.size then raise Stack_full
    else (p.elts.(p.ind) <- e; p.ind <- p.ind + 1) ;;
val push : 'a -> 'a stack -> unit = <fun>
```

Here is a small example of the use of this data structure:

```
# let p = init_stack 4 ;;
val p : '_a stack = {ind=0; size=4; elts=[||]}
# push 1 p ;;
- : unit = ()
# for i = 2 to 5 do push i p done ;;
Uncaught exception: Stack_full
# p ;;
- : int stack = {ind=4; size=4; elts=[|1; 2; 3; 4|]}
# pop p ;;
- : int = 4
# pop p ;;
- : int = 3
```

If we want to prevent raising the exception `Stack_full` when attempting to add an element to the stack, we can enlarge the array. To do this the field `size` must be modifiable too:

```
# type 'a stack =
    {mutable ind:int ; mutable size:int ; mutable elts : 'a array} ;;
# let init_stack n = {ind=0; size=max n 1; elts = [||]} ;;
# let n_push e p =
    if p.elts = [||]
    then
      begin
        p.elts <- Array.create p.size e;
        p.ind <- 1
      end
    else if p.ind >= p.size then
      begin
        let nt = 2 * p.size in
        let nv = Array.create nt e in
        for j=0 to p.size-1 do nv.(j) <- p.elts.(j) done ;
        p.elts <- nv;
        p.size <- nt;
        p.ind <- p.ind + 1
      end
    else
```

```
      begin
        p.elts.(p.ind) <- e ;
        p.ind <- p.ind + 1
      end ;;
val n_push : 'a -> 'a stack -> unit = <fun>
```

All the same, you have to be careful with data structures which can expand without bound. Here is a small example where the initial stack grows as needed.

```
# let p = init_stack 4 ;;
val p : '_a stack = {ind=0; size=4; elts=[||]}
# for i = 1 to 5 do n_push i p done ;;
- : unit = ()
# p ;;
- : int stack = {ind=5; size=8; elts=[|1; 2; 3; 4; 5; 5; 5; 5|]}
# p.stack ;;
Characters 0-7:
Unbound label stack
```

It might also be useful to allow `pop` to decrease the size of the stack, to reclaim unused memory.

Example: Calculations on Matrices

In this example we aim to define a type for matrices, two-dimensional arrays containing floating point numbers, and to write some operations on the matrices. The monomorphic type *mat* is a record containing the dimensions and the elements of the matrix. The functions `create_mat`, `access_mat`, and `mod_mat` are respectively the functions for creation, accessing an element, and modification of an element.

```
# type mat = { n:int; m:int; t: float array array };;
type mat = { n: int; m: int; t: float array array }
# let create_mat n m = { n=n; m=m; t = Array.create_matrix n m 0.0 } ;;
val create_mat : int -> int -> mat = <fun>
# let access_mat m i j = m.t.(i).(j) ;;
val access_mat : mat -> int -> int -> float = <fun>
# let mod_mat m i j e = m.t.(i).(j) <- e ;;
val mod_mat : mat -> int -> int -> float -> unit = <fun>
# let a = create_mat 3 3 ;;
val a : mat = {n=3; m=3; t=[|[|0; 0; 0|]; [|0; 0; 0|]; [|0; 0; 0|]|]}
# mod_mat a 1 1 2.0; mod_mat a 1 2 1.0; mod_mat a 2 1 1.0 ;;
- : unit = ()
# a ;;
- : mat = {n=3; m=3; t=[|[|0; 0; 0|]; [|0; 2; 1|]; [|0; 1; 0|]|]}
```

The sum of two matrices a and b is a matrix c such that $c_{ij} = a_{ij} + b_{ij}$.

```
# let add_mat p q =
    if p.n = q.n && p.m = q.m then
      let r = create_mat p.n p.m  in
      for i = 0 to p.n-1 do
        for j = 0 to p.m-1 do
          mod_mat r i j  (p.t.(i).(j) +. q.t.(i).(j))
        done
      done ;
      r
    else failwith "add_mat : dimensions incompatible";;
val add_mat : mat -> mat -> mat = <fun>
# add_mat  a a ;;
- : mat = {n=3; m=3; t=[|[|0; 0; 0|]; [|0; 4; 2|]; [|0; 2; 0|]|]}
```

The product of two matrices a and b is a matrix c such that $c_{ij} = \sum_{k=1}^{k=m_a} a_{ik}.b_{kj}$

```
# let mul_mat p q =
    if p.m = q.n then
      let r = create_mat p.n q.m in
      for i = 0 to p.n-1 do
        for j = 0 to q.m-1 do
          let c = ref 0.0 in
          for k = 0 to p.m-1 do
             c := !c +.   (p.t.(i).(k) *. q.t.(k).(j))
           done;
           mod_mat r i j !c
        done
      done;
      r
    else failwith "mul_mat : dimensions incompatible" ;;
val mul_mat : mat -> mat -> mat = <fun>
# mul_mat a a;;
- : mat = {n=3; m=3; t=[|[|0; 0; 0|]; [|0; 5; 2|]; [|0; 2; 1|]|]}
```

Order of Evaluation of Arguments

In a pure functional language, the order of evaluation of the arguments does not matter. As there is no modification of memory state and no interruption of the calculation, there is no risk of the calculation of one argument influencing another. On the other hand, in Objective Caml, where there are physically modifiable values and exceptions, there is a danger in not taking account of the order of evaluation of arguments. The following example is specific to version 2.04 of Objective Caml for Linux on Intel hardware:

```
# let new_print_string s = print_string s; String.length s ;;
val new_print_string : string -> int = <fun>
```

```
# (+) (new_print_string "Hello ") (new_print_string "World!") ;;
World!Hello - : int = 12
```
The printing of the two strings shows that the second string is output before the first.

It is the same with exceptions:
```
# try (failwith "function") (failwith "argument") with Failure s → s;;
- : string = "argument"
```

If you want to specify the order of evaluation of arguments, you have to make local declarations forcing this order before calling the function. So the preceding example can be rewritten like this:
```
# let e1 = (new_print_string "Hello ")
  in let e2 = (new_print_string "World!")
  in (+) e1 e2 ;;
Hello World!- : int = 12
```

In Objective Caml, the order of evaluation of arguments is not specified. As it happens, today all implementations of Objective Caml evaluate arguments from left to right. All the same, making use of this implementation feature could turn out to be dangerous if future versions of the language modify the implementation.

We come back to the eternal debate over the design of languages. Should certain features of the language be deliberately left unspecified—should programmers be asked not to use them, on pain of getting different results from their program according to the compiler implementation? Or should everything be specified—should programmers be allowed to use the whole language, at the price of complicating compiler implementation, and forbidding certain optimizations?

Calculator With Memory

We now reuse the calculator example described in the preceding chapter, but this time we give it a user interface, which makes our program more usable as a desktop calculator. This loop allows entering operations directly and seeing results displayed without having to explicitly apply a transition function for each keypress.

We attach four new keys: **C**, which resets the display to zero, **M**, which memorizes a result, **m**, which recalls this memory and **OFF**, which turns off the calculator. This corresponds to the following type:
```
# type key = Plus | Minus | Times | Div | Equals | Digit of int
           | Store | Recall | Clear | Off ;;
```

It is necessary to define a translation function from characters typed on the keyboard to values of type *key*. The exception `Invalid_key` handles the case of characters that do not represent any key of the calculator. The function `code` of module `Char` translates a character to its ASCII-code.

```
# exception Invalid_key ;;
exception Invalid_key
# let translation c = match c with
      '+' → Plus
    | '-' → Minus
    | '*' → Times
    | '/' → Div
    | '=' → Equals
    | 'C' | 'c' → Clear
    | 'M' → Store
    | 'm' → Recall
    | 'o' | 'O' → Off
    | '0'..'9' as c → Digit ((Char.code c) - (Char.code '0'))
    | _ → raise Invalid_key ;;
val translation : char -> key = <fun>
```

In imperative style, the translation function does not calculate a new state, but physically modifies the state of the calculator. Therefore, it is necessary to redefine the type `state` such that the fields are modifiable. Finally, we define the exception `Key_off` for treating the activation of the key **OFF**.

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

```
# exception Key_off ;;
exception Key_off
# let transition s key = match key with
      Clear →   s.vpr <- 0
    | Digit n → s.vpr <- ( if s.lka then s.vpr*10+n else n );
                s.lka <- true
    | Store   →  s.lka <- false ;
                s.mem <- s.vpr
    | Recall →  s.lka <- false ;
                s.vpr <- s.mem
    | Off → raise Key_off
    |  _  → let lcd = match s.loa with
                        Plus  → s.lcd + s.vpr
                      | Minus → s.lcd - s.vpr
                      | Times  → s.lcd * s.vpr
                      | Div   → s.lcd / s.vpr
                      | Equals  → s.vpr
```

```
                        | _ → failwith "transition: impossible match"
                    in
                s.lcd <- lcd ;
                s.lka <- false ;
                s.loa <- key ;
                s.vpr <- s.lcd;;
val transition : state -> key -> unit = <fun>
```

We define the function go, which starts the calculator. Its return value is `()`, because we are only concerned about effects produced by the execution on the environment (start/end, modification of state). Its argument is also the constant `()`, because the calculator is autonomous (it defines its own initial state) and interactive (the arguments of the computation are entered on the keyboard as required). The transitions are performed within an infinite loop (**while** `true` **do**) so we can quit with the exception `Key_off`.

```
# let go () =
    let state = { lcd=0; lka=false; loa=Equals; vpr=0; mem=0 }
    in try
         while true do
           try
             let input = translation (input_char stdin)
             in transition state input ;
                print_newline () ;
                print_string "result: " ;
                print_int state.vpr ;
                print_newline ()
           with
             Invalid_key → ()  (* no effect *)
         done
       with
         Key_off → () ;;
val go : unit -> unit = <fun>
```

We note that the initial state must be either passed as a parameter or declared locally within the function go, because it needs to be initialized at every application of this function. If we had used a value `initial_state` as in the functional program, the calculator would start in the same state as the one it had when it was terminated. This would make it difficult to use two calculators in the same program.

Exercises

Doubly Linked Lists

Functional programming lends itself well to the manipulation of non-cyclic data structures, such as lists for example. For cyclic structures, on the other hand, there are real implementation difficulties. Here we propose to define doubly linked lists, i.e., where each element of a list knows its predecessor and its successor.

1. Define a parameterized type for doubly linked lists, using at least one record with mutable fields.
2. Write the functions `add` and `remove` which add and remove an element of a doubly linked list.

Solving linear systems

This exercise has to do with matrix algebra. It solves a system of equations by Gaussian elimination (i.e., pivoting). We write the system of equations $A\ X = Y$ with A, a square matrix of dimension n, Y, a vector of constants of dimension n and X, a vector of unknowns of the same dimension.

This method consists of transforming the system $A\ X = Y$ into an equivalent system $C\ X = Z$ such that the matrix C is upper triangular. We diagonalize C to obtain the solution.

1. Define a type *`vect`*, a type *`mat`*, and a type *`syst`* .
2. Write utility functions for manipulating vectors: to display a system on screen, to add two vectors, to multiply a vector by a scalar.
3. Write utility functions for matrix computations: multiplication of two matrices, product of a matrix with a vector.
4. Write utility functions for manipulating systems: division of a row of a system by a pivot, (A_{ii}), swapping two rows.
5. Write a function to diagonalize a system. From this, obtain a function solving a linear system.
6. Test your functions on the following systems:

$$AX = \begin{pmatrix} 10 & 7 & 8 & 7 \\ 7 & 5 & 6 & 5 \\ 8 & 6 & 10 & 9 \\ 7 & 5 & 9 & 10 \end{pmatrix} * \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 32 \\ 23 \\ 33 \\ 31 \end{pmatrix} = Y$$

$$AX = \begin{pmatrix} 10 & 7 & 8 & 7 \\ 7 & 5 & 6 & 5 \\ 8 & 6 & 10 & 9 \\ 7 & 5 & 9 & 10 \end{pmatrix} * \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 32.1 \\ 22.9 \\ 33.1 \\ 30.9 \end{pmatrix} = Y$$

$$AX = \begin{pmatrix} 10 & 7 & 8.1 & 7.2 \\ 7.08 & 5.04 & 6 & 5 \\ 8 & 5.98 & 9.89 & 9 \\ 6.99 & 4.99 & 9 & 9.98 \end{pmatrix} * \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 32 \\ 23 \\ 33 \\ 31 \end{pmatrix} = Y$$

7. What can you say about the results you got?

Summary

This chapter has shown the integration of the main features of imperative programming (mutable values, I/O, iterative control structures) into a functional language. Only mutable values, such as strings, arrays, and records with mutable fields, can be physically modified. Other values, once created, are immutable. In this way we obtain read-only (RO) values for the functional part and read-write (RW) values for the imperative part.

It should be noted that, if we don't make use of the imperative features of the language, this extension to the functional core does not change the functional part, except for typing considerations which we can get around.

To Learn More

Imperative programming is the style of programming which has been most widely used since the first computer languages such as Fortran, C, or Pascal. For this reason numerous algorithms are described in this style, often using some kind of pseudo-Pascal. While they could be implemented in a functional style, the use of arrays promotes the use of an imperative style. The data structures and algorithms presented in classic algorithms books, such as [AHU83] and [Sed88], can be carried over directly in the appropriate style. An additional advantage of including these two styles in a single language is being able to define new programming models by mixing the two. This is precisely the subject of the next chapter.