The OCaml system release 5.3

Documentation and user's manual

Xavier Leroy,
Damien Doligez, Alain Frisch, Jacques Garrigue,
Didier Rémy, KC Sivaramakrishnan and Jérôme Vouillon

January 8, 2025

Contents

Foreword

This manual documents the release 5.3 of the OCaml system. It is organized as follows.

- Part I, "An introduction to OCaml", gives an overview of the language.
- Part II, "The OCaml language", is the reference description of the language.
- Part III, "The OCaml tools", documents the compilers, toplevel system, and programming utilities.
- Part IV, "The OCaml library", describes the modules provided in the standard library.
- Part V, "Indexes", contains an index of all identifiers defined in the standard library, and an index of keywords.

Conventions

OCaml runs on several operating systems. The parts of this manual that are specific to one operating system are presented as shown below:

Unix:
This is material specific to the Unix family of operating systems, including Linux and macOS.

Windows:
This is material specific to Microsoft Windows (Vista, 7, 8, 10, 11).

License

Availability

The complete OCaml distribution can be accessed via the website https://ocaml.org/. This site contains a lot of additional information on OCaml.

Part I

An introduction to OCaml

Chapter 1

The core language

This part of the manual is a tutorial introduction to the OCaml language. A good familiarity with programming in a conventional languages (say, C or Java) is assumed, but no prior exposure to functional languages is required. The present chapter introduces the core language. Chapter 2 deals with the module system, chapter 3 with the object-oriented features, chapter 4 with labeled arguments, chapter 5 with polymorphic variants, chapter 6 with the limitations of polymorphism, and chapter 8 gives some advanced examples.

1.1 Basics

For this overview of OCaml, we use the interactive system, which is started by running `ocaml` from the Unix shell or Windows command prompt. This tutorial is presented as the transcript of a session with the interactive system: lines starting with `#` represent user input; the system responses are printed below, without a leading `#`.

Under the interactive system, the user types OCaml phrases terminated by `;;` in response to the `#` prompt, and the system compiles them on the fly, executes them, and prints the outcome of evaluation. Phrases are either simple expressions, or `let` definitions of identifiers (either values or functions).

```
# 1 + 2 * 3;;
- : int = 7

# let pi = 4.0 *. atan 1.0;;
val pi : float = 3.14159265358979312

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

# square (sin pi) +. square (cos pi);;
- : float = 1.
```

The OCaml system computes both the value and the type for each phrase. Even function parameters need no explicit type declaration: the system infers their types from their usage in the function. Notice also that integers and floating-point numbers are distinct types, with distinct operators: `+` and `*` operate on integers, but `+.` and `*.` operate on floats.

```
# 1.0 * 2;;
Error: The constant 1.0 has type float but an expression was expected of type
         int
```

Recursive functions are defined with the `let rec` binding:

```
# let rec fib n =
    if n < 2 then n else fib (n - 1) + fib (n - 2);;
val fib : int -> int = <fun>

# fib 10;;
- : int = 55
```

1.2 Data types

In addition to integers and floating-point numbers, OCaml offers the usual basic data types:

- booleans

```
# (1 < 2) = false;;
- : bool = false

# let one = if true then 1 else 2;;
val one : int = 1
```

- characters

```
#  'a';;
- : char = 'a'

#  int_of_char '\n';;
- : int = 10
```

- immutable character strings

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

# {|This is a quoted string, here, neither \ nor " are special characters|};;
- : string =
"This is a quoted string, here, neither \\ nor \" are special characters"

# {|"\\"|}="\"\\\\\"";;
- : bool = true

#   {delimiter|the end of this|}quoted string is here|delimiter}
  =           "the end of this|}quoted string is here";;
- : bool = true
```

Predefined data structures include tuples, arrays, and lists. There are also general mechanisms for defining your own data structures, such as records and variants, which will be covered in more detail later; for now, we concentrate on lists. Lists are either given in extension as a bracketed list of semicolon-separated elements, or built from the empty list `[]` (pronounce "nil") by adding elements in front using the `::` ("cons") operator.

```
# let l = ["is"; "a"; "tale"; "told"; "etc."];;
val l : string list = ["is"; "a"; "tale"; "told"; "etc."]

# "Life" :: l;;
- : string list = ["Life"; "is"; "a"; "tale"; "told"; "etc."]
```

As with all other OCaml data structures, lists do not need to be explicitly allocated and deallocated from memory: all memory management is entirely automatic in OCaml. Similarly, there is no explicit handling of pointers: the OCaml compiler silently introduces pointers where necessary.

As with most OCaml data structures, inspecting and destructuring lists is performed by pattern-matching. List patterns have exactly the same form as list expressions, with identifiers representing unspecified parts of the list. As an example, here is insertion sort on a list:

```
# let rec sort lst =
    match lst with
      [] -> []
    | head :: tail -> insert head (sort tail)
  and insert elt lst =
    match lst with
      [] -> [elt]
    | head :: tail -> if elt <= head then elt :: lst else head :: insert elt tail
  ;;
val sort : 'a list -> 'a list = <fun>
val insert : 'a -> 'a list -> 'a list = <fun>

# sort l;;
- : string list = ["a"; "etc."; "is"; "tale"; "told"]
```

The type inferred for `sort`, `'a list -> 'a list`, means that `sort` can actually apply to lists of any type, and returns a list of the same type. The type `'a` is a *type variable*, and stands for any given type. The reason why `sort` can apply to lists of any type is that the comparisons (`=`, `<=`, etc.) are *polymorphic* in OCaml: they operate between any two values of the same type. This makes `sort` itself polymorphic over all list types.

```
# sort [6; 2; 5; 3];;
- : int list = [2; 3; 5; 6]

# sort [3.14; 2.718];;
- : float list = [2.718; 3.14]
```

The `sort` function above does not modify its input list: it builds and returns a new list containing the same elements as the input list, in ascending order. There is actually no way in OCaml to modify a list in-place once it is built: we say that lists are *immutable* data structures. Most OCaml

data structures are immutable, but a few (most notably arrays) are *mutable*, meaning that they can be modified in-place at any time.

The OCaml notation for the type of a function with multiple arguments is `arg1_type -> arg2_type -> ... -> return_type`. For example, the type inferred for `insert`, `'a -> 'a list -> 'a list`, means that `insert` takes two arguments, an element of any type `'a` and a list with elements of the same type `'a` and returns a list of the same type.

1.3 Functions as values

OCaml is a functional language: functions in the full mathematical sense are supported and can be passed around freely just as any other piece of data. For instance, here is a `deriv` function that takes any float function as argument and returns an approximation of its derivative function:

```
# let deriv f dx = fun x -> (f (x +. dx) -. f x) /. dx;;
val deriv : (float -> float) -> float -> float -> float = <fun>

# let sin' = deriv sin 1e-6;;
val sin' : float -> float = <fun>

# sin' pi;;
- : float = -1.00000000013961143
```

Even function composition is definable:

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

# let cos2 = compose square cos;;
val cos2 : float -> float = <fun>
```

Functions that take other functions as arguments are called "functionals", or "higher-order functions". Functionals are especially useful to provide iterators or similar generic operations over a data structure. For instance, the standard OCaml library provides a `List.map` functional that applies a given function to each element of a list, and returns the list of the results:

```
# List.map (fun n -> n * 2 + 1) [0;1;2;3;4];;
- : int list = [1; 3; 5; 7; 9]
```

This functional, along with a number of other list and array functionals, is predefined because it is often useful, but there is nothing magic with it: it can easily be defined as follows.

```
# let rec map f l =
    match l with
      [] -> []
    | hd :: tl -> f hd :: map f tl;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

1.4 Records and variants

User-defined data structures include records and variants. Both are defined with the `type` declaration. Here, we declare a record type to represent rational numbers.

```
# type ratio = {num: int; denom: int};;
type ratio = { num : int; denom : int; }

# let add_ratio r1 r2 =
    {num = r1.num * r2.denom + r2.num * r1.denom;
     denom = r1.denom * r2.denom};;
val add_ratio : ratio -> ratio -> ratio = <fun>

# add_ratio {num=1; denom=3} {num=2; denom=5};;
- : ratio = {num = 11; denom = 15}
```

Record fields can also be accessed through pattern-matching:

```
# let integer_part r =
    match r with
      {num=num; denom=denom} -> num / denom;;
val integer_part : ratio -> int = <fun>
```

Since there is only one case in this pattern matching, it is safe to expand directly the argument `r` in a record pattern:

```
# let integer_part {num=num; denom=denom} = num / denom;;
val integer_part : ratio -> int = <fun>
```

Unneeded fields can be omitted:

```
# let get_denom {denom=denom} = denom;;
val get_denom : ratio -> int = <fun>
```

Optionally, missing fields can be made explicit by ending the list of fields with a trailing wildcard `_`::

```
# let get_num {num=num; _ } = num;;
val get_num : ratio -> int = <fun>
```

When both sides of the `=` sign are the same, it is possible to avoid repeating the field name by eliding the `=field` part:

```
# let integer_part {num; denom} = num / denom;;
val integer_part : ratio -> int = <fun>
```

This short notation for fields also works when constructing records:

```
# let ratio num denom = {num; denom};;
val ratio : int -> int -> ratio = <fun>
```

At last, it is possible to update few fields of a record at once:

```
# let integer_product integer ratio = { ratio with num = integer * ratio.num };;
```

```
val integer_product : int -> ratio -> ratio = <fun>
```

With this functional update notation, the record on the left-hand side of `with` is copied except for the fields on the right-hand side which are updated.

The declaration of a variant type lists all possible forms for values of that type. Each case is identified by a name, called a constructor, which serves both for constructing values of the variant type and inspecting them by pattern-matching. Constructor names are capitalized to distinguish them from variable names (which must start with a lowercase letter). For instance, here is a variant type for doing mixed arithmetic (integers and floats):

```
# type number = Int of int | Float of float | Error;;
type number = Int of int | Float of float | Error
```

This declaration expresses that a value of type `number` is either an integer, a floating-point number, or the constant `Error` representing the result of an invalid operation (e.g. a division by zero).

Enumerated types are a special case of variant types, where all alternatives are constants:

```
# type sign = Positive | Negative;;
type sign = Positive | Negative

# let sign_int n = if n >= 0 then Positive else Negative;;
val sign_int : int -> sign = <fun>
```

To define arithmetic operations for the `number` type, we use pattern-matching on the two numbers involved:

```
# let add_num n1 n2 =
    match (n1, n2) with
      (Int i1, Int i2) ->
        (* Check for overflow of integer addition *)
        if sign_int i1 = sign_int i2 && sign_int (i1 + i2) <> sign_int i1
        then Float(float i1 +. float i2)
        else Int(i1 + i2)
    | (Int i1, Float f2) -> Float(float i1 +. f2)
    | (Float f1, Int i2) -> Float(f1 +. float i2)
    | (Float f1, Float f2) -> Float(f1 +. f2)
    | (Error, _) -> Error
    | (_, Error) -> Error;;
val add_num : number -> number -> number = <fun>

# add_num (Int 123) (Float 3.14159);;
- : number = Float 126.14159
```

Another interesting example of variant type is the built-in `'a option` type which represents either a value of type `'a` or an absence of value:

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

This type is particularly useful when defining function that can fail in common situations, for instance

```
# let safe_square_root x = if x >= 0. then Some(sqrt x) else None;;
val safe_square_root : float -> float option = <fun>
```

The most common usage of variant types is to describe recursive data structures. Consider for example the type of binary trees:

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

This definition reads as follows: a binary tree containing values of type `'a` (an arbitrary type) is either empty, or is a node containing one value of type `'a` and two subtrees also containing values of type `'a`, that is, two `'a btree`.

Operations on binary trees are naturally expressed as recursive functions following the same structure as the type definition itself. For instance, here are functions performing lookup and insertion in ordered binary trees (elements increase from left to right):

```
# let rec member x btree =
    match btree with
      Empty -> false
    | Node(y, left, right) ->
        if x = y then true else
        if x < y then member x left else member x right;;
val member : 'a -> 'a btree -> bool = <fun>

# let rec insert x btree =
    match btree with
      Empty -> Node(x, Empty, Empty)
    | Node(y, left, right) ->
        if x <= y then Node(y, insert x left, right)
                  else Node(y, left, insert x right);;
val insert : 'a -> 'a btree -> 'a btree = <fun>
```

1.4.1 Record and variant disambiguation

(This subsection can be skipped on the first reading)

Astute readers may have wondered what happens when two or more record fields or constructors share the same name

```
# type first_record  = { x:int; y:int; z:int }
  type middle_record = { x:int; z:int }
  type last_record   = { x:int };;

# type first_variant = A | B | C
  type last_variant  = A;;
```

The answer is that when confronted with multiple options, OCaml tries to use locally available information to disambiguate between the various fields and constructors. First, if the type of the record or variant is known, OCaml can pick unambiguously the corresponding field or constructor. For instance:

```
# let look_at_x_then_z (r:first_record) =
    let x = r.x in
    x + r.z;;
val look_at_x_then_z : first_record -> int = <fun>

# let permute (x:first_variant) = match x with
    | A -> (B:first_variant)
    | B -> A
    | C -> C;;
val permute : first_variant -> first_variant = <fun>

# type wrapped = First of first_record
  let f (First r) = r, r.x;;
type wrapped = First of first_record
val f : wrapped -> first_record * int = <fun>
```

In the first example, `(r:first_record)` is an explicit annotation telling OCaml that the type of `r` is `first_record`. With this annotation, Ocaml knows that `r.x` refers to the `x` field of the first record type. Similarly, the type annotation in the second example makes it clear to OCaml that the constructors `A`, `B` and `C` come from the first variant type. Contrarily, in the last example, OCaml has inferred by itself that the type of `r` can only be `first_record` and there are no needs for explicit type annotations.

Those explicit type annotations can in fact be used anywhere. Most of the time they are unnecessary, but they are useful to guide disambiguation, to debug unexpected type errors, or combined with some of the more advanced features of OCaml described in later chapters.

Secondly, for records, OCaml can also deduce the right record type by looking at the whole set of fields used in a expression or pattern:

```
# let project_and_rotate {x; y; _} = { x= - y; y = x; z = 0} ;;
val project_and_rotate : first_record -> first_record = <fun>
```

Since the fields `x` and `y` can only appear simultaneously in the first record type, OCaml infers that the type of `project_and_rotate` is `first_record -> first_record`.

In last resort, if there is not enough information to disambiguate between different fields or constructors, Ocaml picks the last defined type amongst all locally valid choices:

```
# let look_at_xz {x; z} = x;;
val look_at_xz : middle_record -> int = <fun>
```

Here, OCaml has inferred that the possible choices for the type of `{x;z}` are `first_record` and `middle_record`, since the type `last_record` has no field `z`. Ocaml then picks the type `middle_record` as the last defined type between the two possibilities.

Beware that this last resort disambiguation is local: once Ocaml has chosen a disambiguation, it sticks to this choice, even if it leads to an ulterior type error:

```
# let look_at_x_then_y r =
    let x = r.x in (* Ocaml deduces [r: last_record] *)
    x + r.y;;
```

```
Error: This expression has type last_record
       There is no field y within type last_record

# let is_a_or_b x = match x with
    | A -> true (* OCaml infers [x: last_variant] *)
    | B -> true;;
Error: This variant pattern is expected to have type last_variant
       There is no constructor B within type last_variant
```

Moreover, being the last defined type is a quite unstable position that may change surreptitiously after adding or moving around a type definition, or after opening a module (see chapter 2). Consequently, adding explicit type annotations to guide disambiguation is more robust than relying on the last defined type disambiguation.

1.5 Imperative features

Though all examples so far were written in purely applicative style, OCaml is also equipped with full imperative features. This includes the usual `while` and `for` loops, as well as mutable data structures such as arrays. Arrays are either created by listing semicolon-separated element values between `[|` and `|]` brackets, or allocated and initialized with the `Array.make` function, then filled up later by assignments. For instance, the function below sums two vectors (represented as float arrays) componentwise.

```
# let add_vect v1 v2 =
    let len = min (Array.length v1) (Array.length v2) in
    let res = Array.make len 0.0 in
    for i = 0 to len - 1 do
      res.(i) <- v1.(i) +. v2.(i)
    done;
    res;;
val add_vect : float array -> float array -> float array = <fun>

# add_vect [| 1.0; 2.0 |] [| 3.0; 4.0 |];;
- : float array = [|4.; 6.|]
```

Record fields can also be modified by assignment, provided they are declared `mutable` in the definition of the record type:

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

# let translate p dx dy =
    p.x <- p.x +. dx; p.y <- p.y +. dy;;
val translate : mutable_point -> float -> float -> unit = <fun>

# let mypoint = { x = 0.0; y = 0.0 };;
val mypoint : mutable_point = {x = 0.; y = 0.}
```

```
# translate mypoint 1.0 2.0;;
- : unit = ()

# mypoint;;
- : mutable_point = {x = 1.; y = 2.}
```

OCaml has no built-in notion of variable – identifiers whose current value can be changed by assignment. (The `let` binding is not an assignment, it introduces a new identifier with a new scope.) However, the standard library provides references, which are mutable indirection cells, with operators `!` to fetch the current contents of the reference and `:=` to assign the contents. Variables can then be emulated by `let`-binding a reference. For instance, here is an in-place insertion sort over arrays:

```
# let insertion_sort a =
    for i = 1 to Array.length a - 1 do
      let val_i = a.(i) in
      let j = ref i in
      while !j > 0 && val_i < a.(!j - 1) do
        a.(!j) <- a.(!j - 1);
        j := !j - 1
      done;
      a.(!j) <- val_i
    done;;
val insertion_sort : 'a array -> unit = <fun>
```

References are also useful to write functions that maintain a current state between two calls to the function. For instance, the following pseudo-random number generator keeps the last returned number in a reference:

```
# let current_rand = ref 0;;
val current_rand : int ref = {contents = 0}

# let random () =
    current_rand := !current_rand * 25713 + 1345;
    !current_rand;;
val random : unit -> int = <fun>
```

Again, there is nothing magical with references: they are implemented as a single-field mutable record, as follows.

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

# let ( ! ) r = r.contents;;
val ( ! ) : 'a ref -> 'a = <fun>

# let ( := ) r newval = r.contents <- newval;;
val ( := ) : 'a ref -> 'a -> unit = <fun>
```

In some special cases, you may need to store a polymorphic function in a data structure, keeping its polymorphism. Doing this requires user-provided type annotations, since polymorphism is only introduced automatically for global definitions. However, you can explicitly give polymorphic types to record fields.

```
# type idref = { mutable id: 'a. 'a -> 'a };;
type idref = { mutable id : 'a. 'a -> 'a; }

# let r = {id = fun x -> x};;
val r : idref = {id = <fun>}

# let g s = (s.id 1, s.id true);;
val g : idref -> int * bool = <fun>

# r.id <- (fun x -> print_string "called id\n"; x);;
- : unit = ()

# g r;;
called id
called id
- : int * bool = (1, true)
```

1.6 Exceptions

OCaml provides exceptions for signalling and handling exceptional conditions. Exceptions can also be used as a general-purpose non-local control structure, although this should not be overused since it can make the code harder to understand. Exceptions are declared with the `exception` construct, and signalled with the `raise` operator. For instance, the function below for taking the head of a list uses an exception to signal the case where an empty list is given.

```
# exception Empty_list;;
exception Empty_list

# let head l =
    match l with
      [] -> raise Empty_list
    | hd :: tl -> hd;;
val head : 'a list -> 'a = <fun>

# head [1; 2];;
- : int = 1

# head [];;
Exception: Empty_list.
```

Exceptions are used throughout the standard library to signal cases where the library functions cannot complete normally. For instance, the `List.assoc` function, which returns the data associated with a given key in a list of (key, data) pairs, raises the predefined exception `Not_found` when the key does not appear in the list:

```
# List.assoc 1 [(0, "zero"); (1, "one")];;
- : string = "one"

# List.assoc 2 [(0, "zero"); (1, "one")];;
Exception: Not_found.
```

Exceptions can be trapped with the `try…with` construct:

```
# let name_of_binary_digit digit =
    try
      List.assoc digit [0, "zero"; 1, "one"]
    with Not_found ->
      "not a binary digit";;
val name_of_binary_digit : int -> string = <fun>

# name_of_binary_digit 0;;
- : string = "zero"

# name_of_binary_digit (-1);;
- : string = "not a binary digit"
```

The `with` part does pattern matching on the exception value with the same syntax and behavior as `match`. Thus, several exceptions can be caught by one `try…with` construct:

```
# let rec first_named_value values names =
    try
      List.assoc (head values) names
    with
    | Empty_list -> "no named value"
    | Not_found -> first_named_value (List.tl values) names;;
val first_named_value : 'a list -> ('a * string) list -> string = <fun>

# first_named_value [0; 10] [1, "one"; 10, "ten"];;
- : string = "ten"
```

Also, finalization can be performed by trapping all exceptions, performing the finalization, then re-raising the exception:

```
# let temporarily_set_reference ref newval funct =
    let oldval = !ref in
    try
      ref := newval;
      let res = funct () in
      ref := oldval;
      res
    with x ->
      ref := oldval;
      raise x;;
val temporarily_set_reference : 'a ref -> 'a -> (unit -> 'b) -> 'b = <fun>
```

An alternative to `try…with` is to catch the exception while pattern matching:

```
# let assoc_may_map f x l =
    match List.assoc x l with
    | exception Not_found -> None
    | y -> f y;;
val assoc_may_map : ('a -> 'b option) -> 'c -> ('c * 'a) list -> 'b option =
  <fun>
```

Note that this construction is only useful if the exception is raised between `match…with`. Exception patterns can be combined with ordinary patterns at the toplevel,

```
# let flat_assoc_opt x l =
    match List.assoc x l with
    | None | exception Not_found -> None
    | Some _ as v -> v;;
val flat_assoc_opt : 'a -> ('a * 'b option) list -> 'b option = <fun>
```

but they cannot be nested inside other patterns. For instance, the pattern `Some (exception A)` is invalid.

When exceptions are used as a control structure, it can be useful to make them as local as possible by using a locally defined exception. For instance, with

```
# let fixpoint f x =
    let exception Done in
    let x = ref x in
    try while true do
        let y = f !x in
        if !x = y then raise Done else x := y
      done
    with Done -> !x;;
val fixpoint : ('a -> 'a) -> 'a -> 'a = <fun>
```

the function `f` cannot raise a `Done` exception, which removes an entire class of misbehaving functions.

1.7 Lazy expressions

OCaml allows us to defer some computation until later when we need the result of that computation.

We use `lazy (expr)` to delay the evaluation of some expression `expr`. For example, we can defer the computation of `1+1` until we need the result of that expression, `2`. Let us see how we initialize a lazy expression.

```
# let lazy_two = lazy (print_endline "lazy_two evaluation"; 1 + 1);;
val lazy_two : int lazy_t = <lazy>
```

We added `print_endline "lazy_two evaluation"` to see when the lazy expression is being evaluated.

The value of `lazy_two` is displayed as `<lazy>`, which means the expression has not been evaluated yet, and its final value is unknown.

Note that `lazy_two` has type `int lazy_t`. However, the type `'a lazy_t` is an internal type name, so the type `'a Lazy.t` should be preferred when possible.

When we finally need the result of a lazy expression, we can call `Lazy.force` on that expression to force its evaluation. The function `force` comes from standard-library module Lazy[29.30].

```
# Lazy.force lazy_two;;
lazy_two evaluation
- : int = 2
```

Notice that our function call above prints "lazy_two evaluation" and then returns the plain value of the computation.

Now if we look at the value of `lazy_two`, we see that it is not displayed as `<lazy>` anymore but as `lazy 2`.

```
# lazy_two;;
- : int lazy_t = lazy 2
```

This is because `Lazy.force` memoizes the result of the forced expression. In other words, every subsequent call of `Lazy.force` on that expression returns the result of the first computation without recomputing the lazy expression. Let us force `lazy_two` once again.

```
# Lazy.force lazy_two;;
- : int = 2
```

The expression is not evaluated this time; notice that "lazy_two evaluation" is not printed. The result of the initial computation is simply returned.

Lazy patterns provide another way to force a lazy expression.

```
# let lazy_l = lazy ([1; 2] @ [3; 4]);;
val lazy_l : int list lazy_t = <lazy>

# let lazy l = lazy_l;;
val l : int list = [1; 2; 3; 4]
```

We can also use lazy patterns in pattern matching.

```
# let maybe_eval lazy_guard lazy_expr =
    match lazy_guard, lazy_expr with
    | lazy false, _ -> "matches if (Lazy.force lazy_guard = false); lazy_expr not forced"
    | lazy true, lazy _ -> "matches if (Lazy.force lazy_guard = true); lazy_expr forced";;
val maybe_eval : bool lazy_t -> 'a lazy_t -> string = <fun>
```

The lazy expression `lazy_expr` is forced only if the `lazy_guard` value yields `true` once computed. Indeed, a simple wildcard pattern (not lazy) never forces the lazy expression's evaluation. However, a pattern with keyword `lazy`, even if it is wildcard, always forces the evaluation of the deferred computation.

1.8 Symbolic processing of expressions

We finish this introduction with a more complete example representative of the use of OCaml for symbolic processing: formal manipulations of arithmetic expressions containing variables. The following variant type describes the expressions we shall manipulate:

```
# type expression =
      Const of float
    | Var of string
    | Sum of expression * expression    (* e1 + e2 *)
    | Diff of expression * expression   (* e1 - e2 *)
    | Prod of expression * expression   (* e1 * e2 *)
    | Quot of expression * expression   (* e1 / e2 *)
  ;;
type expression =
    Const of float
  | Var of string
  | Sum of expression * expression
  | Diff of expression * expression
  | Prod of expression * expression
  | Quot of expression * expression
```

We first define a function to evaluate an expression given an environment that maps variable names to their values. For simplicity, the environment is represented as an association list.

```
# exception Unbound_variable of string;;
exception Unbound_variable of string

# let rec eval env exp =
    match exp with
      Const c -> c
    | Var v ->
        (try List.assoc v env with Not_found -> raise (Unbound_variable v))
    | Sum(f, g) -> eval env f +. eval env g
    | Diff(f, g) -> eval env f -. eval env g
    | Prod(f, g) -> eval env f *. eval env g
    | Quot(f, g) -> eval env f /. eval env g;;
val eval : (string * float) list -> expression -> float = <fun>

# eval [("x", 1.0); ("y", 3.14)] (Prod(Sum(Var "x", Const 2.0), Var "y"));;
- : float = 9.42
```

Now for a real symbolic processing, we define the derivative of an expression with respect to a variable `dv`:

```
# let rec deriv exp dv =
    match exp with
      Const c -> Const 0.0
    | Var v -> if v = dv then Const 1.0 else Const 0.0
    | Sum(f, g) -> Sum(deriv f dv, deriv g dv)
    | Diff(f, g) -> Diff(deriv f dv, deriv g dv)
    | Prod(f, g) -> Sum(Prod(f, deriv g dv), Prod(deriv f dv, g))
    | Quot(f, g) -> Quot(Diff(Prod(deriv f dv, g), Prod(f, deriv g dv)),
                         Prod(g, g))
  ;;
```

```
val deriv : expression -> string -> expression = <fun>

# deriv (Quot(Const 1.0, Var "x")) "x";;
- : expression =
Quot (Diff (Prod (Const 0., Var "x"), Prod (Const 1., Const 1.)),
 Prod (Var "x", Var "x"))
```

1.9 Pretty-printing

As shown in the examples above, the internal representation (also called *abstract syntax*) of expressions quickly becomes hard to read and write as the expressions get larger. We need a printer and a parser to go back and forth between the abstract syntax and the *concrete syntax*, which in the case of expressions is the familiar algebraic notation (e.g. `2*x+1`).

For the printing function, we take into account the usual precedence rules (i.e. `*` binds tighter than `+`) to avoid printing unnecessary parentheses. To this end, we maintain the current operator precedence and print parentheses around an operator only if its precedence is less than the current precedence.

```
# let print_expr exp =
    (* Local function definitions *)
    let open_paren prec op_prec =
      if prec > op_prec then print_string "(" in
    let close_paren prec op_prec =
      if prec > op_prec then print_string ")" in
    let rec print prec exp =      (* prec is the current precedence *)
      match exp with
        Const c -> print_float c
      | Var v -> print_string v
      | Sum(f, g) ->
          open_paren prec 0;
          print 0 f; print_string " + "; print 0 g;
          close_paren prec 0
      | Diff(f, g) ->
          open_paren prec 0;
          print 0 f; print_string " - "; print 1 g;
          close_paren prec 0
      | Prod(f, g) ->
          open_paren prec 2;
          print 2 f; print_string " * "; print 2 g;
          close_paren prec 2
      | Quot(f, g) ->
          open_paren prec 2;
          print 2 f; print_string " / "; print 3 g;
          close_paren prec 2
    in print 0 exp;;
```

```
val print_expr : expression -> unit = <fun>

# let e = Sum(Prod(Const 2.0, Var "x"), Const 1.0);;
val e : expression = Sum (Prod (Const 2., Var "x"), Const 1.)

# print_expr e; print_newline ();;
2. * x + 1.
- : unit = ()

# print_expr (deriv e "x"); print_newline ();;
2. * 1. + 0. * x + 0.
- : unit = ()
```

1.10 Printf formats

There is a `printf` function in the `Printf`[29.44] module (see chapter 2) that allows you to make formatted output more concisely. It follows the behavior of the `printf` function from the C standard library. The `printf` function takes a format string that describes the desired output as a text interspersed with specifiers (for instance `%d`, `%f`). Next, the specifiers are substituted by the following arguments in their order of apparition in the format string:

```
# Printf.printf "%i + %i is an integer value, %F * %F is a float, %S\n"
  3 2 4.5 1. "this is a string";;
3 + 2 is an integer value, 4.5 * 1. is a float, "this is a string"
- : unit = ()
```

The OCaml type system checks that the type of the arguments and the specifiers are compatible. If you pass it an argument of a type that does not correspond to the format specifier, the compiler will display an error message:

```
# Printf.printf "Float value: %F" 42;;

Error: The constant 42 has type int but an expression was expected of type
         float
  Hint: Did you mean 42.?
```

The `fprintf` function is like `printf` except that it takes an output channel as the first argument. The `%a` specifier can be useful to define custom printers (for custom types). For instance, we can create a printing template that converts an integer argument to signed decimal:

```
# let pp_int ppf n = Printf.fprintf ppf "%d" n;;
val pp_int : out_channel -> int -> unit = <fun>

# Printf.printf "Outputting an integer using a custom printer: %a " pp_int 42;;
Outputting an integer using a custom printer: 42 - : unit = ()
```

The advantage of those printers based on the `%a` specifier is that they can be composed together to create more complex printers step by step. We can define a combinator that can turn a printer for `'a` type into a printer for `'a optional`:

```
# let pp_option printer ppf = function
    | None -> Printf.fprintf ppf "None"
    | Some v -> Printf.fprintf ppf "Some(%a)" printer v;;
val pp_option :
  (out_channel -> 'a -> unit) -> out_channel -> 'a option -> unit = <fun>

# Printf.fprintf stdout
    "The current setting is %a. \nThere is only %a\n"
    (pp_option pp_int) (Some 3)
    (pp_option pp_int) None
  ;;
The current setting is Some(3).
There is only None
- : unit = ()
```

If the value of its argument its `None`, the printer returned by pp_option printer prints `None` otherwise it uses the provided printer to print `Some` .

Here is how to rewrite the pretty-printer using `fprintf`:

```
# let pp_expr ppf expr =
    let open_paren prec op_prec output =
      if prec > op_prec then Printf.fprintf output "%s" "(" in
    let close_paren prec op_prec output =
      if prec > op_prec then Printf.fprintf output "%s" ")" in
    let rec print prec ppf expr =
        match expr with
        | Const c -> Printf.fprintf ppf "%F" c
        | Var v -> Printf.fprintf ppf "%s" v
        | Sum(f, g) ->
            open_paren prec 0 ppf;
            Printf.fprintf ppf "%a + %a" (print 0) f (print 0) g;
            close_paren prec 0 ppf
        | Diff(f, g) ->
            open_paren prec 0 ppf;
            Printf.fprintf ppf "%a - %a" (print 0) f (print 1) g;
            close_paren prec 0 ppf
        | Prod(f, g) ->
            open_paren prec 2 ppf;
            Printf.fprintf ppf "%a * %a" (print 2) f (print 2) g;
            close_paren prec 2 ppf
        | Quot(f, g) ->
            open_paren prec 2 ppf;
            Printf.fprintf ppf "%a / %a" (print 2) f (print 3) g;
            close_paren prec 2 ppf
    in print 0 ppf expr;;
val pp_expr : out_channel -> expression -> unit = <fun>
```

```
# pp_expr stdout e; print_newline ();;
2. * x + 1.
- : unit = ()

# pp_expr stdout (deriv e "x"); print_newline ();;
2. * 1. + 0. * x + 0.
- : unit = ()
```

Due to the way that format strings are built, storing a format string requires an explicit type annotation:

```
# let str : _ format =
      "%i is an integer value, %F is a float, %S\n";;

# Printf.printf str 3 4.5 "string value";;
3 is an integer value, 4.5 is a float, "string value"
- : unit = ()
```

1.11 Standalone OCaml programs

All examples given so far were executed under the interactive system. OCaml code can also be compiled separately and executed non-interactively using the batch compilers `ocamlc` and `ocamlopt`. The source code must be put in a file with extension `.ml`. It consists of a sequence of phrases, which will be evaluated at runtime in their order of appearance in the source file. Unlike in interactive mode, types and values are not printed automatically; the program must call printing functions explicitly to produce some output. The `;;` used in the interactive examples is not required in source files created for use with OCaml compilers, but can be helpful to mark the end of a top-level expression unambiguously even when there are syntax errors. Here is a sample standalone program to print the greatest common divisor (gcd) of two numbers:

```
(* File gcd.ml *)
let rec gcd a b =
  if b = 0 then a
  else gcd b (a mod b);;

let main () =
  let a = int_of_string Sys.argv.(1) in
  let b = int_of_string Sys.argv.(2) in
  Printf.printf "%d\n" (gcd a b);
  exit 0;;
main ();;
```

`Sys.argv` is an array of strings containing the command-line parameters. `Sys.argv.(1)` is thus the first command-line parameter. The program above is compiled and executed with the following shell commands:

```
$ ocamlc -o gcd gcd.ml
$ ./gcd 6 9
```

```
3
$ ./gcd 7 11
1
```

More complex standalone OCaml programs are typically composed of multiple source files, and can link with precompiled libraries. Chapters 13 and 16 explain how to use the batch compilers `ocamlc` and `ocamlopt`. Recompilation of multi-file OCaml projects can be automated using third-party build systems, such as dune.

Chapter 2

The module system

This chapter introduces the module system of OCaml.

2.1 Structures

A primary motivation for modules is to package together related definitions (such as the definitions of a data type and associated operations over that type) and enforce a consistent naming scheme for these definitions. This avoids running out of names or accidentally confusing names. Such a package is called a *structure* and is introduced by the `struct`…`end` construct, which contains an arbitrary sequence of definitions. The structure is usually given a name with the `module` binding. For instance, here is a structure packaging together a type of FIFO queues and their operations:

```
# module Fifo =
    struct
      type 'a queue = { front: 'a list; rear: 'a list }
      let make front rear =
        match front with
        | [] -> { front = List.rev rear; rear = [] }
        | _  -> { front; rear }
      let empty = { front = []; rear = [] }
      let is_empty = function { front = []; _ } -> true | _ -> false
      let add x q = make q.front (x :: q.rear)
      exception Empty
      let top = function
        | { front = []; _ } -> raise Empty
        | { front = x :: _; _ } -> x
      let pop = function
        | { front = []; _ } -> raise Empty
        | { front = _ :: f; rear = r } -> make f r
    end;;
module Fifo :
  sig
    type 'a queue = { front : 'a list; rear : 'a list; }
```

```
    val make : 'a list -> 'a list -> 'a queue
    val empty : 'a queue
    val is_empty : 'a queue -> bool
    val add : 'a -> 'a queue -> 'a queue
    exception Empty
    val top : 'a queue -> 'a
    val pop : 'a queue -> 'a queue
  end
```

Outside the structure, its components can be referred to using the "dot notation", that is, identifiers qualified by a structure name. For instance, `Fifo.add` is the function `add` defined inside the structure `Fifo` and `Fifo.queue` is the type `queue` defined in `Fifo`.

```
# Fifo.add "hello" Fifo.empty;;
- : string Fifo.queue = {Fifo.front = ["hello"]; rear = []}
```

Another possibility is to open the module, which brings all identifiers defined inside the module into the scope of the current structure.

```
# open Fifo;;
```

```
# add "hello" empty;;
- : string Fifo.queue = {front = ["hello"]; rear = []}
```

Opening a module enables lighter access to its components, at the cost of making it harder to identify in which module an identifier has been defined. In particular, opened modules can shadow identifiers present in the current scope, potentially leading to confusing errors:

```
# let empty = []
  open Fifo;;
val empty : 'a list = []

# let x = 1 :: empty ;;
Error: The value empty has type 'a Fifo.queue
       but an expression was expected of type int list
```

A partial solution to this conundrum is to open modules locally, making the components of the module available only in the concerned expression. This can also make the code both easier to read (since the open statement is closer to where it is used) and easier to refactor (since the code fragment is more self-contained). Two constructions are available for this purpose:

```
# let open Fifo in
  add "hello" empty;;
- : string Fifo.queue = {front = ["hello"]; rear = []}
```

and

```
# Fifo.(add "hello" empty);;
- : string Fifo.queue = {front = ["hello"]; rear = []}
```

In the second form, when the body of a local open is itself delimited by parentheses, braces or bracket, the parentheses of the local open can be omitted. For instance,

```
# Fifo.[empty] = Fifo.([empty]);;
- : bool = true

# Fifo.[|empty|] = Fifo.([|empty|]);;
- : bool = true

# Fifo.{ contents = empty } = Fifo.({ contents = empty });;
- : bool = true
```

This second form also works for patterns:

```
# let at_most_one_element x = match x with
  | Fifo.{ front = ([] | [_]); rear = [] } -> true
  | _ -> false ;;
val at_most_one_element : 'a Fifo.queue -> bool = <fun>
```

It is also possible to copy the components of a module inside another module by using an `include` statement. This can be particularly useful to extend existing modules. As an illustration, we could add functions that return an optional value rather than an exception when the queue is empty.

```
# module FifoOpt =
  struct
    include Fifo
    let top_opt q = if is_empty q then None else Some(top q)
    let pop_opt q = if is_empty q then None else Some(pop q)
  end;;
module FifoOpt :
  sig
    type 'a queue = 'a Fifo.queue = { front : 'a list; rear : 'a list; }
    val make : 'a list -> 'a list -> 'a queue
    val empty : 'a queue
    val is_empty : 'a queue -> bool
    val add : 'a -> 'a queue -> 'a queue
    exception Empty
    val top : 'a queue -> 'a
    val pop : 'a queue -> 'a queue
    val top_opt : 'a queue -> 'a option
    val pop_opt : 'a queue -> 'a queue option
  end
```

2.2 Signatures

Signatures are interfaces for structures. A signature specifies which components of a structure are accessible from the outside, and with which type. It can be used to hide some components of a structure (e.g. local function definitions) or export some components with a restricted type. For instance, the signature below specifies the queue operations `empty`, `add`, `top` and `pop`, but not the auxiliary function `make`. Similarly, it makes the `queue` type abstract (by not providing its actual

representation as a concrete type). This ensures that users of the `Fifo` module cannot violate data structure invariants that operations rely on, such as "if the front list is empty, the rear list must also be empty".

```
# module type FIFO =
    sig
      type 'a queue                 (* now an abstract type *)
      val empty : 'a queue
      val add : 'a -> 'a queue -> 'a queue
      val top : 'a queue -> 'a
      val pop : 'a queue -> 'a queue
      exception Empty
    end;;
module type FIFO =
  sig
    type 'a queue
    val empty : 'a queue
    val add : 'a -> 'a queue -> 'a queue
    val top : 'a queue -> 'a
    val pop : 'a queue -> 'a queue
    exception Empty
  end
```

Restricting the `Fifo` structure to this signature results in another view of the `Fifo` structure where the `make` function is not accessible and the actual representation of queues is hidden:

```
# module AbstractQueue = (Fifo : FIFO);;
module AbstractQueue : FIFO

# AbstractQueue.make [1] [2;3] ;;
Error: Unbound value AbstractQueue.make

# AbstractQueue.add "hello" AbstractQueue.empty;;
- : string AbstractQueue.queue = <abstr>
```

The restriction can also be performed during the definition of the structure, as in

```
module Fifo = (struct ... end : FIFO);;
```

An alternate syntax is provided for the above:

```
module Fifo : FIFO = struct ... end;;
```

Like for modules, it is possible to include a signature to copy its components inside the current signature. For instance, we can extend the `FIFO` signature with the `top_opt` and `pop_opt` functions:

```
# module type FIFO_WITH_OPT =
    sig
      include FIFO
      val top_opt: 'a queue -> 'a option
      val pop_opt: 'a queue -> 'a queue option
    end;;
```

```
module type FIFO_WITH_OPT =
  sig
    type 'a queue
    val empty : 'a queue
    val add : 'a -> 'a queue -> 'a queue
    val top : 'a queue -> 'a
    val pop : 'a queue -> 'a queue
    exception Empty
    val top_opt : 'a queue -> 'a option
    val pop_opt : 'a queue -> 'a queue option
  end
```

2.3 Functors

Functors are "functions" from modules to modules. Functors let you create parameterized modules and then provide other modules as parameter(s) to get a specific implementation. For instance, a `Set` module implementing sets as sorted lists could be parameterized to work with any module that provides an element type and a comparison function `compare` (such as `OrderedString`):

```
# type comparison = Less | Equal | Greater;;
type comparison = Less | Equal | Greater

# module type ORDERED_TYPE =
    sig
      type t
      val compare: t -> t -> comparison
    end;;
module type ORDERED_TYPE = sig type t val compare : t -> t -> comparison end

# module Set =
    functor (Elt: ORDERED_TYPE) ->
      struct
        type element = Elt.t
        type set = element list
        let empty = []
        let rec add x s =
          match s with
            [] -> [x]
          | hd::tl ->
             match Elt.compare x hd with
               Equal   -> s         (* x is already in s *)
             | Less    -> x :: s    (* x is smaller than all elements of s *)
             | Greater -> hd :: add x tl
        let rec member x s =
          match s with
            [] -> false
```

```
          | hd::tl ->
              match Elt.compare x hd with
                Equal   -> true     (* x belongs to s *)
              | Less    -> false    (* x is smaller than all elements of s *)
              | Greater -> member x tl
      end;;
module Set :
  (Elt : ORDERED_TYPE) ->
    sig
      type element = Elt.t
      type set = element list
      val empty : 'a list
      val add : Elt.t -> Elt.t list -> Elt.t list
      val member : Elt.t -> Elt.t list -> bool
    end
```

By applying the `Set` functor to a structure implementing an ordered type, we obtain set operations for this type:

```
# module OrderedString =
    struct
      type t = string
      let compare x y = if x = y then Equal else if x < y then Less else Greater
    end;;
module OrderedString :
  sig type t = string val compare : 'a -> 'a -> comparison end

# module StringSet = Set(OrderedString);;
module StringSet :
  sig
    type element = OrderedString.t
    type set = element list
    val empty : 'a list
    val add : OrderedString.t -> OrderedString.t list -> OrderedString.t list
    val member : OrderedString.t -> OrderedString.t list -> bool
  end

# StringSet.member "bar" (StringSet.add "foo" StringSet.empty);;
- : bool = false
```

2.4 Functors and type abstraction

As in the `Fifo` example, it would be good style to hide the actual implementation of the type `set`, so that users of the structure will not rely on sets being lists, and we can switch later to another, more efficient representation of sets without breaking their code. This can be achieved by restricting `Set` by a suitable functor signature:

```
# module type SETFUNCTOR =
```

```
    (Elt: ORDERED_TYPE) ->
      sig
        type element = Elt.t       (* concrete *)
        type set                   (* abstract *)
        val empty : set
        val add : element -> set -> set
        val member : element -> set -> bool
      end;;
module type SETFUNCTOR =
  (Elt : ORDERED_TYPE) ->
    sig
      type element = Elt.t
      type set
      val empty : set
      val add : element -> set -> set
      val member : element -> set -> bool
    end

# module AbstractSet = (Set : SETFUNCTOR);;
module AbstractSet : SETFUNCTOR

# module AbstractStringSet = AbstractSet(OrderedString);;
module AbstractStringSet :
  sig
    type element = OrderedString.t
    type set = AbstractSet(OrderedString).set
    val empty : set
    val add : element -> set -> set
    val member : element -> set -> bool
  end

# AbstractStringSet.add "gee" AbstractStringSet.empty;;
- : AbstractStringSet.set = <abstr>
```

In an attempt to write the type constraint above more elegantly, one may wish to name the signature of the structure returned by the functor, then use that signature in the constraint:

```
# module type SET =
    sig
      type element
      type set
      val empty : set
      val add : element -> set -> set
      val member : element -> set -> bool
    end;;
module type SET =
  sig
    type element
    type set
```

```
    val empty : set
    val add : element -> set -> set
    val member : element -> set -> bool
  end

# module WrongSet = (Set : (Elt: ORDERED_TYPE) -> SET);;
module WrongSet : (Elt : ORDERED_TYPE) -> SET

# module WrongStringSet = WrongSet(OrderedString);;
module WrongStringSet :
  sig
    type element = WrongSet(OrderedString).element
    type set = WrongSet(OrderedString).set
    val empty : set
    val add : element -> set -> set
    val member : element -> set -> bool
  end

# WrongStringSet.add "gee" WrongStringSet.empty ;;
Error: This constant has type string but an expression was expected of type
         WrongStringSet.element = WrongSet(OrderedString).element
```

The problem here is that `SET` specifies the type `element` abstractly, so that the type equality between `element` in the result of the functor and `t` in its argument is forgotten. Consequently, `WrongStringSet.element` is not the same type as `string`, and the operations of `WrongStringSet` cannot be applied to strings. As demonstrated above, it is important that the type `element` in the signature `SET` be declared equal to `Elt.t`; unfortunately, this is impossible above since `SET` is defined in a context where `Elt` does not exist. To overcome this difficulty, OCaml provides a `with type` construct over signatures that allows enriching a signature with extra type equalities:

```
# module AbstractSet2 =
    (Set : (Elt: ORDERED_TYPE) -> (SET with type element = Elt.t));;
module AbstractSet2 :
  (Elt : ORDERED_TYPE) ->
    sig
      type element = Elt.t
      type set
      val empty : set
      val add : element -> set -> set
      val member : element -> set -> bool
    end
```

As in the case of simple structures, an alternate syntax is provided for defining functors and restricting their result:

```
module AbstractSet2(Elt: ORDERED_TYPE) : (SET with type element = Elt.t) =
  struct ... end;;
```

Abstracting a type component in a functor result is a powerful technique that provides a high degree of type safety, as we now illustrate. Consider an ordering over character strings that is

different from the standard ordering implemented in the `OrderedString` structure. For instance, we compare strings without distinguishing upper and lower case.

```
# module NoCaseString =
    struct
      type t = string
      let compare s1 s2 =
        OrderedString.compare (String.lowercase_ascii s1) (String.lowercase_ascii s2)
    end;;
module NoCaseString :
  sig type t = string val compare : string -> string -> comparison end

# module NoCaseStringSet = AbstractSet(NoCaseString);;
module NoCaseStringSet :
  sig
    type element = NoCaseString.t
    type set = AbstractSet(NoCaseString).set
    val empty : set
    val add : element -> set -> set
    val member : element -> set -> bool
  end

# NoCaseStringSet.add "FOO" AbstractStringSet.empty ;;
Error: The value AbstractStringSet.empty has type
         AbstractStringSet.set = AbstractSet(OrderedString).set
       but an expression was expected of type
         NoCaseStringSet.set = AbstractSet(NoCaseString).set
```

Note that the two types `AbstractStringSet.set` and `NoCaseStringSet.set` are not compatible, and values of these two types do not match. This is the correct behavior: even though both set types contain elements of the same type (strings), they are built upon different orderings of that type, and different invariants need to be maintained by the operations (being strictly increasing for the standard ordering and for the case-insensitive ordering). Applying operations from `AbstractStringSet` to values of type `NoCaseStringSet.set` could give incorrect results, or build lists that violate the invariants of `NoCaseStringSet`.

2.5 Modules and separate compilation

All examples of modules so far have been given in the context of the interactive system. However, modules are most useful for large, batch-compiled programs. For these programs, it is a practical necessity to split the source into several files, called compilation units, that can be compiled separately, thus minimizing recompilation after changes.

In OCaml, compilation units are special cases of structures and signatures, and the relationship between the units can be explained easily in terms of the module system. A compilation unit A comprises two files:

- the implementation file A`.ml`, which contains a sequence of definitions, analogous to the inside of a `struct`…`end` construct;

- the interface file *A*`.mli`, which contains a sequence of specifications, analogous to the inside of a `sig`…`end` construct.

These two files together define a structure named *A* as if the following definition was entered at top-level:

```
module A: sig (* contents of file A.mli *) end
        = struct (* contents of file A.ml *) end;;
```

The files that define the compilation units can be compiled separately using the `ocamlc -c` command (the `-c` option means "compile only, do not try to link"); this produces compiled interface files (with extension `.cmi`) and compiled object code files (with extension `.cmo`). When all units have been compiled, their `.cmo` files are linked together using the `ocamlc` command. For instance, the following commands compile and link a program composed of two compilation units `Aux` and `Main`:

```
$ ocamlc -c Aux.mli                     # produces aux.cmi
$ ocamlc -c Aux.ml                      # produces aux.cmo
$ ocamlc -c Main.mli                    # produces main.cmi
$ ocamlc -c Main.ml                     # produces main.cmo
$ ocamlc -o theprogram Aux.cmo Main.cmo
```

The program behaves exactly as if the following phrases were entered at top-level:

```
module Aux: sig (* contents of Aux.mli *) end
          = struct (* contents of Aux.ml *) end;;
module Main: sig (* contents of Main.mli *) end
           = struct (* contents of Main.ml *) end;;
```

In particular, `Main` can refer to `Aux`: the definitions and declarations contained in `Main.ml` and `Main.mli` can refer to definition in `Aux.ml`, using the `Aux.`*ident* notation, provided these definitions are exported in `Aux.mli`.

The order in which the `.cmo` files are given to `ocamlc` during the linking phase determines the order in which the module definitions occur. Hence, in the example above, `Aux` appears first and `Main` can refer to it, but `Aux` cannot refer to `Main`.

Note that only top-level structures can be mapped to separately-compiled files, but neither functors nor module types. However, all module-class objects can appear as components of a structure, so the solution is to put the functor or module type inside a structure, which can then be mapped to a file.

Chapter 3

Objects in OCaml

(Chapter written by Jérôme Vouillon, Didier Rémy and Jacques Garrigue)

This chapter gives an overview of the object-oriented features of OCaml.

Note that the relationship between object, class and type in OCaml is different than in mainstream object-oriented languages such as Java and C++, so you shouldn't assume that similar keywords mean the same thing. Object-oriented features are used much less frequently in OCaml than in those languages. OCaml has alternatives that are often more appropriate, such as modules and functors. Indeed, many OCaml programs do not use objects at all.

3.1 Classes and objects

The class `point` below defines one instance variable `x` and two methods `get_x` and `move`. The initial value of the instance variable is `0`. The variable `x` is declared mutable, so the method `move` can change its value.

```
# class point =
    object
      val mutable x = 0
      method get_x = x
      method move d = x <- x + d
    end;;
class point :
  object val mutable x : int method get_x : int method move : int -> unit end
```

We now create a new point `p`, instance of the `point` class.

```
# let p = new point;;
val p : point = <obj>
```

Note that the type of `p` is `point`. This is an abbreviation automatically defined by the class definition above. It stands for the object type `<get_x : int; move : int -> unit>`, listing the methods of class `point` along with their types.

We now invoke some methods of `p`:

```
# p#get_x;;
```

```
- : int = 0

# p#move 3;;
- : unit = ()

# p#get_x;;
- : int = 3
```

The evaluation of the body of a class only takes place at object creation time. Therefore, in the following example, the instance variable `x` is initialized to different values for two different objects.

```
# let x0 = ref 0;;
val x0 : int ref = {contents = 0}

# class point =
    object
      val mutable x = incr x0; !x0
      method get_x = x
      method move d = x <- x + d
    end;;
class point :
  object val mutable x : int method get_x : int method move : int -> unit end

# new point#get_x;;
- : int = 1

# new point#get_x;;
- : int = 2
```

The class `point` can also be abstracted over the initial values of the `x` coordinate.

```
# class point = fun x_init ->
    object
      val mutable x = x_init
      method get_x = x
      method move d = x <- x + d
    end;;
class point :
  int ->
  object val mutable x : int method get_x : int method move : int -> unit end
```

Like in function definitions, the definition above can be abbreviated as:

```
# class point x_init =
    object
      val mutable x = x_init
      method get_x = x
      method move d = x <- x + d
    end;;
```

```
class point :
  int ->
  object val mutable x : int method get_x : int method move : int -> unit end
```

An instance of the class `point` is now a function that expects an initial parameter to create a point object:

```
# new point;;
- : int -> point = <fun>

# let p = new point 7;;
val p : point = <obj>
```

The parameter `x_init` is, of course, visible in the whole body of the definition, including methods. For instance, the method `get_offset` in the class below returns the position of the object relative to its initial position.

```
# class point x_init =
    object
      val mutable x = x_init
      method get_x = x
      method get_offset = x - x_init
      method move d = x <- x + d
    end;;
class point :
  int ->
  object
    val mutable x : int
    method get_offset : int
    method get_x : int
    method move : int -> unit
  end
```

Expressions can be evaluated and bound before defining the object body of the class. This is useful to enforce invariants. For instance, points can be automatically adjusted to the nearest point on a grid, as follows:

```
# class adjusted_point x_init =
    let origin = (x_init / 10) * 10 in
    object
      val mutable x = origin
      method get_x = x
      method get_offset = x - origin
      method move d = x <- x + d
    end;;
class adjusted_point :
  int ->
  object
    val mutable x : int
    method get_offset : int
```

```
    method get_x : int
    method move : int -> unit
  end
```

(One could also raise an exception if the `x_init` coordinate is not on the grid.) In fact, the same effect could be obtained here by calling the definition of class `point` with the value of the `origin`.

```
# class adjusted_point x_init =  point ((x_init / 10) * 10);;
class adjusted_point : int -> point
```

An alternate solution would have been to define the adjustment in a special allocation function:

```
# let new_adjusted_point x_init = new point ((x_init / 10) * 10);;
val new_adjusted_point : int -> point = <fun>
```

However, the former pattern is generally more appropriate, since the code for adjustment is part of the definition of the class and will be inherited.

This ability provides class constructors as can be found in other languages. Several constructors can be defined this way to build objects of the same class but with different initialization patterns; an alternative is to use initializers, as described below in section 3.4.

3.2 Immediate objects

There is another, more direct way to create an object: create it without going through a class.

The syntax is exactly the same as for class expressions, but the result is a single object rather than a class. All the constructs described in the rest of this section also apply to immediate objects.

```
# let p =
    object
      val mutable x = 0
      method get_x = x
      method move d = x <- x + d
    end;;
val p : < get_x : int; move : int -> unit > = <obj>

# p#get_x;;
- : int = 0

# p#move 3;;
- : unit = ()

# p#get_x;;
- : int = 3
```

Unlike classes, which cannot be defined inside an expression, immediate objects can appear anywhere, using variables from their environment.

```
# let minmax x y =
    if x < y then object method min = x method max = y end
    else object method min = y method max = x end;;
```