IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

TP – Discovering OCaml

Langages et logiques – LaLog

Objectives

At the end of the activity, you should be capable of:

- building simple functional OCaml modules;
- running OCaml programs from the interpreter;
- building and running OCaml programs from the command line;
- discovering the rest of the language by yourselves.

Introduction

OCaml is a functional programming language. In other words, it is a programming language where functions are *first class values*. Functions can be manipulated like any other kind of value: passed as parameters to other functions, returned as result of a call to a function, stored in a data structure, ...

OCaml is a language:

- statically typed by inference: the use of values and variables is verified at the time of compilation; information of type does not need to be provided by the programmer, it is computed by the compiler.
- offering parametric polymorphism: if a function does not explore the whole structure of one of its arguments, it has a type not entirely determined (a variable type).
- whose allocations and deallocations of data in memory are automatically managed by a garbage collector.
- including imperative features: it is possible to use imperative control structures and to physically manipulate some values (arrays, references, ...). Imperative feature tend to use side effects.

- providing a comprehensive mechanism of exceptions.
- which has an interpreter (`ocaml`), a virtual machine (`ocamlrun`), two compilers (one for the virtual machine `ocamlc` and a native `ocamlopt` one) and development tools (execution tracing, dependencies management, performance analysis, package deployment, testing, ...).

An OCaml program can be structured using two approaches: by modules or by classes in an object oriented fashion. The choice between these two models of structuration offers a great flexibility to the language. They are dual, the modules facilitating the extension of the treatments and the objects facilitating data extension. As part of the course, we will not explore the object aspects of the language. The reader interested by this aspect is referred to the referenced documents presented below.

In ocaml, a module is a compilation unit that groups together data and code that are described by an interface. The language integrates multiple notions that allow advanced manipulation of modules (parametric modules, functors, ...). When we will practice, we will limit ourselves to simple modules *e.g.* separate files. This document will describe in more details the way to build a module in the section B.3.

The associated course is mainly focused on functional concepts which you probably don't know. Our objective, here, is to practice this functional part of OCaml. In annex A, you will find a partially redacted version of what has been seen during the course. To discover the objects and imperative parts of OCaml, you can consult:

- The official manual `http://caml.inria.fr/pub/docs/manual-ocaml`
- The OCaml portal `http://ocaml.org`
- A page of this portal containing numerous links to books on OCaml: `http://ocaml.org/learn/books.html`. Among these books, I would advise mostly to read:
  - `https://realworldocaml.org`
  - A french teaching book: `http://caml.inria.fr/pub/docs/oreilly-book/index.html`

We will also use `opam` the OCaml package manager (`https://opam.ocaml.org`). This tool allows one to download, to compile and to install libraries for OCaml. It installs software in the directory `.opam`[1] in your root directory and adds access to these libraries to your environment[2].

The various basic tools and `opam` are installed in the lab classes.

1 An introduction

In this section, we recommend using the interpreter (sometimes referred to as an interaction loop).

[1] Before the first use of `opam`, `opam init`, which is responsible for creating the directory `.opam`.

[2] The command `eval \`opam config env\`` is responsible for this initialization. If you want to run it for each new terminal, put it in your `.profile.perso`. Be careful to the enclosing characters: they are *backquotes* (or *backticks*) and they are often changed when copy-pasting them, therefore type the command instead of copy-pasting it.

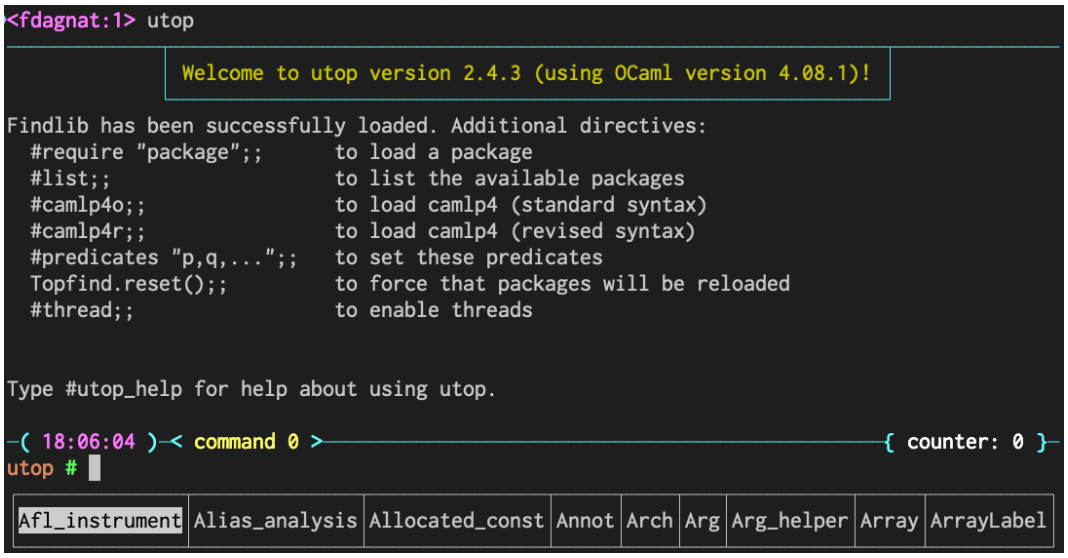

Figure 1: `utop` interpreter

1.1 The interpreter

The ocaml interpreter is started by the command `ocaml`. We are going to use a more user-friendly version of this interpreter: `utop`[3]. It should be already installed on the corporate computers. However, if it is not the case, it can be installed with `opam install utop`.

The figure 1 contains a screenshot of `utop` when launching it. The *prompt* of the interpreter is the character `#`. Once this character is displayed, the interpreter reads the data input until meeting `;;`. The sentences can be expressions (which give a result) or definitions that introduce new variables, types, ... (*e.g.* `let` statement). The interpreter reacts to an expression by displaying the result of its evaluation preceded by its type and to a definition by giving the type of the defined value and possibly its value. For example :

```
# 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
```

Directives allow you to interact with the interaction loop. They are distinguished from OCaml expressions because they begin with a sharp `#`. For example, to leave the loop use the `#quit` directive (followed by `;;`). Many directives are available, some specific to `utop` other inherited from the standard interpreter. Among those we will use:

- `#use` which loads and interprets the contents of a file,
- `#directory` to add a file search directory,

[3] `https://github.com/diml/utop`

- `#cd` to change the current directory of the interpreter,
- `#load` to load a compiled file,
- `#trace` / `untrace` to obtain / suspend traces of calls to the target function,
- ... see `http://caml.inria.fr/pub/docs/manual-ocaml/toplevel.html`.

A line management mechanism is offered by `utop` and follows bindings of `bash` [4], you can modify and move forward / backward through the history (the arrows to go up and go down). Finally, `utop` offers a mechanism for completion, its lower bar dynamically displays the current possible completions, complete by using its first proposal using the tab key.

2 Basic OCaml

Exercise 1 (*Card deck*)

▷ **Question 1.1:**
Propose a type to represent the cards of an usual card game.

▷ **Question 1.2:**
Define a function `all_the_cards` which takes a color as parameter and builds a list containing all the cards of the given color.

▷ **Question 1.3:**
Define a function `string_of_cards` which converts a card to a string representing its value and color.

Exercise 2 (*Binary trees*)

▷ **Question 2.1:**
Propose a type for binary trees.

▷ **Question 2.2:**
Use this type to realize a binary search tree for integers. Such a tree has the following invariant: for every node, the values contained in all the left sub-tree are smaller to the one of the node and the values contained in all the right sub-tree are larger.

For this define a function `add` that adds an integer in a binary search tree.

▷ **Question 2.3:**
Use the previously defined function to implement a function sorting list of integers.

[4] See `#utop_bindings` for the shortcuts of `utop`.

3 A functional data structure

Exercise 3 (*List with position*)

The idea of this exercise is to implement (efficiently) a notion of list with a position. It is a stateful data structure that maintains a list of elements together with a current position in this list of elements.

So for example, if we suppose that the elements of the list are integers. Here is such a list 117 34 55 3 where the position is indicated by the blue box.

A naive implementation could be done by a pair composed of the list of elements and an integer storing the position. It would not be efficient to access the current element as we would have to find it every time. In a language with pointers (such as Java or C), such a list would be a pair formed of the list of elements and a pointer to the current element. The complexity of accessing the current element would then be in $O(1)$. In a functional language, we do not have pointers[5].

The trick here is to see that such a list is a triple when not empty: the list of the elements before, the current element and the list of the elements after. Using our exemple, we have `[117;34],55,[3]`. Accessing current element is trivial.

▷ **Question 3.1:**
Propose a type, a constructor function to create such lists and a `current` function that returns the current element. The list must be polymorphic. The constructor function could, for example, take a list of elements and initialize the position to the first element of this list.

In order to be able to test your implementation do not hesitate to define other useful functions.

▷ **Question 3.2:**
What is the cost of moving the position one element right? What is the cost of moving the position one element left? Could we do better?

Define efficient `move_left`, `move_right`, `add_left` and `add_right` functions (it may require to modify the code already done).

▷ **Question 3.3:**
Define an iter and a map function for your list.

4 Formal calculus

This section contains a longer problem. The objective is to build a simple calculator of arithmetic expressions.

Exercise 4 (*Formal calculus*)

▷ **Question 4.1:**
Propose a type to describe simple expressions containing floating point numbers and the four basic operators ($+$, $-$, $*$, $/$).

[5] They exist in OCaml but we will stick to a pure functional style.

▷ **Question 4.2:**
Define a function `eval` to evaluate an expression.

▷ **Question 4.3:**
Extend the type of expression to make it possible to use variables. A variable is a string that appears in an expression. During evaluation, the variables will be given a value.

▷ **Question 4.4:**
Modify the fonction `eval` to receive an environment. An environment is an association list associating values to variables. Meeting a variables not defined in the environment should lead to an error.

▷ **Question 4.5:**
Write a function `string_of_expr`.

▷ **Question 4.6:**
Define a function `derive` qui that derive an expression with respect to one variable given as argument.

▷ **Question 4.7:**
Propose a function `simplify` that simplifies an expression using the following rules:

$$\begin{cases} -0 = 0 \\ \forall e \quad e + 0 = 0 + e = e \\ \forall e \quad e\ddot{O}0 = 0\ddot{O}e = 0 \\ \forall e \quad e\ddot{O}1 = 1\ddot{O}e = e \end{cases}$$

A Basics

OCaml comments are contained between `(* ... *)`. Comments may be nested inside other comments.

A.1 Types

In OCaml, the primitive types are:

- `unit` which contain a unique value `()`,
- **integers** (`int`) with their usual operations (`+`,`-`,`*`, `/`, `mod`, `int_of_float`, ...),
- **floating point numbers** (`float`) with their usual operations (`+.`,`-.`,`*.`, `/.`, `**`, `float_of_int`, ...),
- **booleans** `false` and `true` (`bool`) with classical logical operators and usual comparison operators (`=`, `<>`, `<`, `>`, `<=`, `>=`, `not`, `&&`, `||`),
- **characters** (`char`) between `'` with the usual special characters (`\t`, `\n`, ...),

- **strings** (`string`) between `"` with concatenation `^` ; all previous types may be converted to strings by function of the form `string_of_`*type* ; we can get the character at position i of a string by `tab.[i]`[6],
- **tuples** (`_ * _ * _`) the separator character is `,`, pairs have `fst` and `snd` operators ; larger tuples must be destructured using pattern matching,
- **lists** (`_ list`) between `[]` with the separator character `;`, the list constructor is `::` that add an element to the head of a list, there exists also an operator for concatenation `@`.

For more information, please consult the OCaml manual and more precisely the part on the library (part IV) of `http://caml.inria.fr/pub/docs/manual-ocaml`.

A.2 Control structure

Usual imperative control structure exists in OCaml: choice `if then else`, sequence `;`, blocks `begin ... end`, iterations `for i = e1 to e2 do e3 done` and `while e1 do e2 done`.

The main control structure is pattern matching. A pattern is a partially constructed value containing *holes* (in fact variables not yet defined). Matching is then an operation making it possible to compare a pattern with a value, if the two entities match (*i.e.* have the same form), the holes (the free variables) are filled by the corresponding sub-values (they are defined). The process is comparable to regular expressions.

The matching may fail, in which case the following case is used or an exception is raised if no other case is available (see example 2, below). Notice that the interpreter (and compiler) emits a warning if a pattern matching is incomplete and may therefore fail.

For example:

```
# let a = ([2;3;4;5],1) ;;
val a : int list * int = ([2; 3; 4; 5], 1)

# match a with (c,1) -> c ;;
Warning: this pattern-matching is not exhaustive. Here is an example of a
value that is not matched: (_, 0)
- : int list = [2; 3; 4; 5]

# match a with (c,2) -> c ;;
Warning: this pattern-matching is not exhaustive. Here is an example of a
value that is not matched: (_, 0)
Exception: Match_failure ("", 1, 0).

# match a with (2::c,1) -> c ;;
Warning: this pattern-matching is not exhaustive. Here is an example of a
value that is not matched: ([], _)
- : int list = [3; 4; 5]
```

A (free) variable can only occur once in the pattern. There exists a "hole" pattern `_` match everything but not binding the resulting data to a variable.

[6]Note that OCaml is changing to an immutable string type, you may encounter code modifying strings but we are not going to do this. If you need mutable strings use the type `bytes`.

Pattern matching can be done by `match` or `function`. It is also the basics structure for definition functions. Lastly, the handling blocks for exceptions use also pattern matching. For example, an insertion sort can be implemented like follows.

First, there is a function to insert a value in a sorted list:

```
let rec insert elt lst =
 match lst with
   [] -> [elt]
 | h::t -> if elt <= h then elt::lst else h::(insert elt t)
```

Then it is used to sort any list:

```
let rec sort lst =
 match lst with
   [] -> []
 | h::t -> insert h (sort t)
```

The value returned by a function is the value of its last expression.

This function also illustrates polymorphism. Indeed, as it does not use the structure of the elements, it is independant of it and has the following type: `'a list -> 'a list` where `'a` is a type variable.

As in all the functional languages, the approach to iterate is to use recursion. Definitions use `let rec`. During this courses, we forget about imperative control structure and focus on functional constructs. It will be a constant requirements for all the codes you produce.

Notice also that generally, in OCaml, data structure are immutable. Once a list has been defined one cannot modify its content. You only can build new lists.

A.3 Functions

In OCaml, a function is a first class value. It can be given as argument to another function. For example, it is possible to define a function `iter` that wait a function `f` and a list `l` and applies sequentially `f` to all elements of `l`.

```
let rec iter f l =
 match l with
 | [] -> ()
 | h::t -> f h; iter f t
```

This function has type `('a -> 'b) -> 'a list -> unit`. It can be used to prints the elements of a list of strings, the printing function being `print_string`:

```
# iter print_string ["a";"b";"c";"n"] ;;
abc
- : unit = ()
```

In OCaml, one can partially apply functions. It consists in providing less argument that required during the call. The result is then another function expecting the remaining arguments:

```
# let print_list = iter print_string ;;
val print_list : string list -> unit = <fun>

# print_list ["a";"b";"c";"n"] ;;
abc
- : unit = ()
```

B More OCaml

B.1 Simple compilation

OCaml programs can also be compiled. In a file, all related definitions and expressions are collected to define a module. This file must have a `.ml` extension. The definitions and the expressions it contains do not require `;;` anymore. The module has the name of the capitalized file. For example, the file `a.ml` creates the module `A` while the file `AnotherFile.ml` creates the module `AnotherFile`.

OCaml enjoys separate compilation. So you can compile modules separately even if they refer to each others. Following usual software engineering discipline, you define small modules with well-defined boundaries.

Inside a module `M`, you can use a value or a type `x` which is defined in another module `P` either by using the name `P.x` or by opening the module using `open P` and by using the name `x`.

Each file must be compiled separately. For this purpose OCaml comes with two compilers:

- a *bytecode* compiler (`ocamlc`) that produces a portable compiled object file of extension `.cmo` using an OCaml specific bytecode language; this bytecode can then be executed by a virtual machine called `ocamlrun`[7]
- a native compiler (`ocamlopt`) that produces specific machine code for the compiling device; it produces two files: one has the extension `.cmx` and the other `.o`. The first one contains information for linking and the second one contains the compiled object code

The bytecode compiler works faster and produces portable files but these bytecode files run slower than the machine code produced by the native compiler. Compilation is generally slower using the native compiler although you should not really see it on the file we will compile in this UE.

Both compilers also produce a `.cmi` file describing the element the module offers. It is used when compiling files that depend on this module to ensure we do not use undefined elements and we use them correctly.

Both compilers proceeds the same way in two steps:

1. *compilation* of each required module `ocamlc -c ...`[8], notice here that you must compile the files in the reverse order of dependencies; if `a.ml` depends on `b.ml`, you must compile `b.ml` before `a.ml`
2. *linking* all the obtained compiled artefacts to produce an executable `ocamlc -o prog ...`[8]; the result (here named `prog`) can be executed[9]

[7] The same way, Java has the Java bytecode that can be executed by the JVM.

[9] In the case of the bytecode compiler, it produces a script starting with a `#!/a/path/to/ocamlrun`.

Notice that when linking the compiled files must be given in the right order (inverse of dependencies again).

Executing the produced program consists in evaluating each module in the linking order. The evaluation of a module proceeds by evaluating each value defined in the module in the order of their definition. So, any non functional value leads to computation. This means that there is no notion of a main function in OCaml. It is replaced by any value definition. For example, it is idiomatic to use on of the following toplevel code to define a computation of the program: `let () =` *an expression* or `let _ =` *an expression*[10].

For example, if you have a file `name.ml` containing `let name ="Fabien"` and a file `main.ml` containing `let () = print_endline ("Hello "^Name.name)`. You can:

1. (byte-)compile `ocamlc -c name.ml`, it produces two files `name.cmi` and `name.cmo`
2. (byte-)compile `ocamlc -c main.ml`, it produces two files `main.cmi` and `main.cmo`
3. link the two (bytecode) files `ocamlc -o result name.cmo main.cmo`, it produces an executable file `result`
4. running `./result` prints `Hello Fabien` in your terminal; first the definition of `name` in `Name` is evaluated and then the anonymous expression of `Main` is computed

If you compile `main.ml` first, you get an error message specifying that the module `Name` is not defined (`Unbound module Name`). If you forget `name.cmo` when linking or put it after `main.cmo`, you also get an error saying that the module `Name` is not available (`Required module 'Name' is unavailable`).

The same exemple, using the native compiler gives:

1. compile `ocamlopt -c name.ml`, it produces three files `name.cmi`, `name.cmx` and `name.o`
2. compile `ocamlopt -c main.ml`, it produces two files `main.cmi`, `main.cmx` and `main.o`
3. link the two obejct files `ocamlopt -o result name.cmx main.cmx`, it produces an executable file `result`
4. running `./result` prints `Hello Fabien` in your terminal

The same errors can happen using the native compiler and the bytecode compiler.

B.2 More on compilation

An OCaml module can also have an interface. If your module is defined in the file `a.ml`, there exists a *signature* defined in a file `a.mli`. This signature defines the exported elements of the module. All external modules can only reuse the value declared in the module's signature. Signatures are useful to hide implementation details and to provide abstract well-defined interface to other modules.

Defining interfaces change slightly the compilation process. When compiling a module the compiler uses the interface (the `.mli`) to produce the compiled interface (the `.cmi`) file.

[9] or `ocamlopt`

[10] Note that the first ensure that your expression returns `()`.

The compilation of an OCaml project — as in any languages — is sufficiently complex to require specific tools. The generic `make` tools (and any of its sibling like `cmake`) can be used. The OCaml compilation process has specificity that you will have to take into account. Therefore, OCaml-specific tools have been developed. The distribution of OCaml provides an automatic building tool: `ocamlbuild`[11].

Recently, a new automation tool is becoming the *de facto* standard among the OCaml community. This tool — *Dune*[12] — supports a large panel of building activities. We will not explore it in details but we will use it. Dune reads project metadata from files named `dune`. It uses this information to setup build rules, handle installation, etc. Contrary to some of the previously cited tools, Dune is highly portable.

Using our previous example, a simple `dune` file such as:

```
;; This declares the main executable implemented by main.ml
(executable (name main))
```

Dune will explore the current directory and compile all needed files within a `_build` directory when you invoke `dune build main.exe`[13]. The name `main.exe` provides the target for the compilation process. Once done, you can run the program by invoking `dune exec ./main.exe`. If you need more information on Dune, you can consult its manual at `https://dune.readthedocs.io/en/latest/index.html`.

Dune works on the basis of a notion of *workspace*. When you run a `dune` command, it is going to search a directory root to the current workspace. It starts from current directory and search its parents until it finds a `dune-project` file. This is the root directory that contains the `_build` directory. This makes it possible to run a Dune command from any subdirectory of this root. When executing an operation Dune prints the directory in which it works.

Dune provides a command to create a workspace: `dune init`. If you give it a sub-command `exe main`, it creates an executable project where the entry point is a `main.ml` file. To get a more complex workspace there exists a `proj my_project` sub-commands that creates a whole directory structure.

Lastly, Dune advocates a composable approach where your program is often decomposed in a set of libraries and a main part. Each element being in its own subdirectory of the root. In this case, there is another sub-command of Dune which is interesting : `dune utop <dir>`. This commands launch a utop instance with all the libraries it found in the directory `<dir>` loaded enabling easy experiments.

B.3 Modules

A module is a set of definitions of types, values, functions, exceptions, ...). It has an *signature* and a *structure*. The signature defines the (external) interface of the module. Each of these public entities can be reused by other modules. The structure must define the entities declared in the signature (with compatible structure). The entities of the structure that are not declared in the signature can not be used by other modules. This makes it possible to abstract types (*i.e.* not manipulable see section B.5).

[11] The interested reader is invited to read the documentation of the tool at `https://github.com/ocaml/ocamlbuild/blob/master/manual/manual.adoc`.

[12] `https://github.com/ocaml/dune`

[13] You should have installed Dune with opam before you can run it: `opam install dune`.

In its simplest embodiment, a module is a file containing the declarations of the module (extension `ml`). Its signature shall then be in a file with the same name and extension `mli`. In this case, the name of the module corresponds to the name of the file capitalized.

When compiling, the signature is compiled into a file of extension `cmi`. If no signature is provided, all entities of the module (*i.e.* the file) are available (a `cmi` file is automatically generated).

In a program, the use of a value `toto` defined in another module `Tutu` must be prefixed by the name of the module: `Tutu.toto`.

In fact, the modules are much more powerful than presented above, you can consult `http://caml.inria.fr/pub/docs/manual-ocaml/moduleexamples.html` to read more.

B.4 Constructed types

In OCaml, it is possible to define new types (syntax `type truc =`). Theses types can be parameterized (syntax `type 'var truc =`). These new types are often aliases of already existing types. All values of the model types are also of the alias type. Beware that if the alias is then abstracted the two types (the model and the alias) become incompatibles. For example, if you declare `type id = int` and then hide the realization of `id` then `id` (which are integers) and `int` (which are also integers) won't be compatible.

One of the type constructor is the sum constructor which allow to define **variants**. Such a type is the result of a definition of the following form:

```
type name =
  | Name1 of t1
  | Name2 of t2
  | ...
  | Namek of tk
```

This type contains all the values built with the (value) **constructors** `Name1` to `Namek`[14]. Building a new value of this type is done by `Name1(toto)` for example if `toto` is a value of type `t1`.

These types are then manipulated using pattern matching.

A sum type can be recursive when one of its sub-element as a type using the sum type. The type `list` is an example of such a recursive sum type, it is defined by:

```
type 'a list =
  | []
  | :: of 'a * 'a list
```

Another sum types predefined in OCaml is the type `option`:

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

OCaml also has support to define records (named product types):

```
type ratio = { num: int; denum: int }

let add r1 r2 = { num = r1.num * r2.denum + r2.num * r1.denum; denum = r1.denum * r2.denum }
```

[14] Beware capitalization!

```
add {num=1; denum=3} {num=2; denum=5}
```

B.5 Type abstraction

One of the advantages of the concept of modules is the possibility of abstracting a type. Indeed, it is possible to declare a new type (in a signature) without showing (and making accessible) its realization. By example, the type `ratio` can be abstracted by the signature (file `ratio.mli`):

```
type ratio
val add : ratio -> ratio -> ratio
val num : ratio -> int
val denum : ratio -> int
val create : int -> int -> ratio
val print : ratio -> unit
```

The realization of the module can the be in the file `ratio.ml`:

```
type ratio = { num: int; denum: int }

let add r1 r2 = { num = r1.num * r2.denum + r2.num * r1.denum; denum = r1.denum * r2.denum }

let num r = r.num
let denum r = r.denum

let create n d = { num = n; denum = d }

open Printf
let print r =
 printf "%i/%i" r.num r.denum
```

The implementation of the type `ratio` is no more usable by other modules, the following:

```
let r = Ratio.create 1 1 in
 print_int r.num
```

generate the following compile error:

```
File "UseRatioError.ml", line 2, characters 14-17:
Error: Unbound record field num
```

To use such an abstracted type, the external code must use the functions provided by the module (the API):

```
let r1 = Ratio.create 1 1
and r2 = Ratio.create 1 2 in
  Ratio.print (Ratio.add r1 r2);
  print_newline ()
```

B.6 Exceptions

OCaml supports exceptions. An exception must be declared by the keyword `exception`, they can be raised by `raise` and they are caught by the construction `try ... with`.

For example, the function `head` below that returns the head of a list may raise an exception when it is given an empty list:

```
exception Empty_list

let head l =
 match l with
 | [] -> raise Empty_list
 | hd :: tl -> hd
```

In the standard library of OCaml, functions are defined to manipulate a notion of dictionary (called association list). The `List` module contains a function `assoc` that takes a key and association list and returns the value associated with the key in the association list. If the key is not present in the association list an exception `Not_found` is raised. Thus, writing a function `name_of_digit` that converts a digit (not a number) to string of characters can be written like follows:

```
let name_of_digit digit =
 try
  List.assoc digit [0, "zero"; 1, "one"; 2, "two"; 3, "three"; 4, "four";
                    5, "five"; 6, "six"; 7, "seven"; 8, "eight"; 9, "nine"]
 with Not_found ->
  "not a digit"
```

The `with` part contain a pattern matching and the exception can also contain data.