Functional programming and controlling complexity of software systems using the Objective Caml system

René Thoraval - Université de Nantes - France
ERASMUS
Eötvös Loránd University - Faculty of Informatics - Budapest - Hungary

October 5, 2006

Contents

About these pages

This is a draft, especially in its tutorial part.

There are many grammatical errors and so on!

This only a part of an *introduction* to *Functional programming and controlling complexity of software systems using the Objective Caml system* designed for only *a ten-hours curse* involving students whose levels in computer science are not known well enough by the author at the time he is writing these pages.

This pages are mainly an introduction to *Functional programming using the Objective Caml system*.

The systematic use of abstract types, specifications and separately compiled modules is the way the author chooses to shortly introduce *Controlling complexity of software systems*.

1 Introduction

1.1 Tutorial (draft)

1.1.1 Programming paradigms

Programming paradigms[1]: *imperative*, *functional*, *logic*, *object-oriented*, and so on.

A programming language can support multiple paradigms.

The Objective Caml language supports several paradigms.

Here *pure* functional programming (*no side-effects*) is only considered. Pure functional programming enforces *referential transparency*, which is the notion that *equals can be substituted for equals*.

1.1.2 Functional programming

Functional programming[2]: *a computation is viewed as the application of a mathematical function.*

Functional programming avoids state and mutable data (whereas imperative programming emphasizes changes in state and the execution of sequential commands).

Lambda-calculus[3] provides a theoretical framework for describing functions and their application.

1.1.3 Functional languages

Most of them can support several paradigms.

See *http://www.cs.nott.ac.uk/˜gmh/faq.html#functional-languages* for instance.

Examples: Scheme, Haskell, Clean, Miranda, Erlang, languages of the ML family as for instance Caml.

There are sequential languages and concurrent ones (Erlang for instance).

They also can be classified according other criteria: *purity*[4] (no side-effects), *strictness*[5] (*eager* or *lazy* evaluation), ...

1.1.4 The ML family

ML[6] is a general-purpose functional programming language developed by *Robin Milner* and others in the late 1970s at the University of Edinburgh.

[1] See *http://en.wikipedia.org/wiki/Programming_paradigm* for instance.
[2] See *http://en.wikipedia.org/wiki/Functional_programming* for instance.
[3] See *http://en.wikipedia.org/wiki/Lambda_calculus* for instance.
[4] See *http://www.cs.nott.ac.uk/˜gmh/faq.html#purity* for instance.
[5] See *http://www.cs.nott.ac.uk/˜gmh/faq.html#strictness* for instance.
[6] See *http://en.wikipedia.org/wiki/ML_programming_language* for instance.

ML is often referred to as an impure functional language, because it permits side-effects, and therefore imperative programming, unlike purely functional programming languages such as Haskell.

Features of ML include a *call-by-value evaluation strategy*, *first class functions*, *automatic memory management through garbage collection*, *parametric polymorphism*, *static typing*, *type inference*, *algebraic data types*, *pattern matching*, and *exception handling*.

The ML family includes *OCaml*[7] (Objective Caml), *Standard ML of New Jersey*[8], ...

[7] See *http://en.wikipedia.org/wiki/OCaml* for instance.
[8] See *http://en.wikipedia.org/wiki/Standard_ML_of_New_Jersey* for instance.

2 Expressions and functions

2.1 Tutorial (draft)

2.1.1 Expressions

In Objective Caml

a program is an *expression*.

Even functions and loops are expressions.

There exist

- Expressions that contain *illegal characters*.
- Expressions without illegal characters that are *not syntactically correct*.
- Syntactically correct expressions that contain *values names* that are *not bound to values*.
- Syntactically correct expressions without unbound value names that are *not well-typed*.

If an expression is *well-typed* then it contains no illegal character, is syntactically correct and contains no unbound value name.

2.1.2 Binding value names

Local definition: the `let [rec]... in...` construct is used *to bind value names locally*.

Definition: the `let [rec]...` construct is used *to bind value names not locally*.

Binding several value names in some *unspecified order* (that is, *in parallel*) is possible.

2.1.3 Types

There are *built-in types*, for instance `int` and `bool`.

New types can be defined.

Recursive definitions of types are allowed, and even *mutually* recursive ones.

2.1.4 Automatic type inference

Given an expression the compiler checks its syntax and whether all value names it contains are bound. Of course!

Moreover, if the expression is not well-typed

- Then the compiler rejects it.
- Else

  the compiler computes the type of the expression (along with its value).

Thus in Objective Caml

there is no need for type declaration!

Yet it is a *strictly-typed* language. That is, *implicit conversions are forbidden.*

2.1.5 Well-typed expressions and computations - Values and types

In Objective Caml

an expression is well-typed if and only it describes a computation.

Evaluation of a well-typed expression *returns a value* at the end of the computation.

Each value has a type.

Each well-typed expression has a type: the type of the value returned by its evaluation.

2.1.6 Functional values

In Objective Caml

functional values are so-called *first-class citizens.*

That is, they are considered as any other values, except for *equality test* ... and except the fact that they *can be applied to other values*!

For instance `exp` is a (functional) value.

Therefore

- A functional value can be an argument of a function.
- A functional value can be the result of applying a function.

Thus

higher-order **functional values can be written.**

That is, functions taking functions as arguments and functions whose application return functions.

2.1.7 Polymorphism

In Objective Caml

a type or a function can be written *generically*

so that it can deal equally well with any values without depending on their type.
For instance:

- Lists, pairs and binary trees can be written generically.
- List reversal, computing the first component of a pair or the number of nodes of a binary tree can be written generically.

2.1.8 Functions - Abstraction and application

Consider the following code.
First three values:

```
2 + (3 * 4) ;;

5 + (0 * 8) ;;

4 + (1 * 7) ;;
```

Then the same values expressed using local definitions:

```
let
  a = 2 and b = 3 and c = 4
in
  a + (b * c) ;;

let
  a = 5 and b = 0 and c = 8
in
  a + (b * c) ;;

let
  a = 4 and b = 1 and c = 7
in
  a + (b * c) ;;
```

Now the abstraction step:

```
function a, b, c -> a + (b * c) ;;
```

Finally the application step (three times):

```
(function a, b, c -> a + (b * c)) (2, 3, 4) ;;

(function a, b, c -> a + (b * c)) (5, 0, 8) ;;

(function a, b, c -> a + (b * c)) (4, 1, 7) ;;
```

Now consider this code.
First three values:

```
3.  *. sin (2.)       +. 0.5 *. cos (4.) ;;

7.2 *. abs_float (3.) +. 8.  *. exp (6.1) ;;

5.  *. log (10.5)     +. 3.1 *. sin (0.7) ;;
```

Then the same values expressed using local definitions:

```
let
  a = 3. and b = sin and c = 2. and d = 0.5 and e = cos and f = 4.
in
  a *. b (c) +. d *. e (f) ;;

let
  a = 7.2 and b = abs_float and c = 3. and d = 8. and e = exp and f = 6.1
in
  a *. b (c) +. d *. e (f) ;;

let
  a = 5. and b = log and c = 10.5 and d = 3.1 and e = sin and f = 0.7
in
  a *. b (c) +. d *. e (f) ;;
```

Now the abstraction step:

```
function a, b, c, d, e, f -> a  *. b (c) +. d *. e (f) ;;
```

Finally the application step (three times):

```
(function a, b, c, d, e, f -> a  *. b (c) +. d *. e (f)) (3., sin, 2., 0.5, cos, 4.) ;;

(function a, b, c, d, e, f -> a  *. b (c) +. d *. e (f)) (7.2, abs_float, 3., 8., exp, 6.1) ;;

(function a, b, c, d, e, f -> a  *. b (c) +. d *. e (f)) (5., log, 10.5, 3.1, sin, 0.7) ;;
```

In both cases, there is first an *abstraction step* from three expressions.

Abstraction consists of:

1. Noting that the structures of the expressions are similar.
2. Identifying what is identical and what is not.
3. Transforming
   - What is identical into an expression having a *functional value*. Let f denote this expression.
   - What is not identical into the *formal parameters* of the expression f.

So, at the end of the abstraction step a *functional value* is written.

Then in both cases there is (three times) an *application step*.

An application step consists of applying the (functional) value of the expression f to *arguments* that correspond to what is particular in the initial expression considered. This results in obtaining the value of this initial expression.

Application consists of:

1. Bounding each of the formal parameters in the expression f to the values of the corresponding arguments.
2. Evaluating the right-hand part of the expression f in this *context*.

Clearly both cases are similar, though the initial values are functional in the second one, which gives a *higher-order function*.

2.2 Exercises

2.2.1 Exercise 1 - Leap years

Recall that a year is *leap* (or *bissextile*) if either it is divisible by 4 but not by 100 or it is divisible by 400. For instance 2000 and 2004 are leap years but 1900 and 2006 are not.

Write two functional values that test whether a non-negative year is leap and whose type is `int -> bool`:

1. Using `if` but no boolean operator.

2. Using boolean operators but not `if`.

2.2.2 Exercise 2 - Type inference

The following expression is syntactically correct. It has a value. Write its type.

```
function (x, y) -> (if x then 2 + 3 else 0)
                   +
                   (let a = "false" and b = 13 in if y = a then 1 else b)
```

2.2.3 Exercise 3 - Local definitions

The three expressions in the second column are syntactically correct and have values of the same type:

2	`function (a, b, c)`	`->`	`((a + b) + c) / ((a + b) - c)`
3	`function (a, b, c)`	`->`	`((a + b) / (b + c)) + ((b + c) / (a + b))`
3	`function (a, b, c)`	`->`	`(((c + (a + b)) / (c - (a + b))) / a)` `+ (a / ((c + (a + b)) / (c - (a + b))))`

1. What is this type?

2. For each of them use the notion of *local definition* to write an expression having the same value and containing as many occurrences of the `+` operator as written in the first column.

2.2.4 Exercise 4 - Closed intervals whose bounds are integers

The non-built-in type `closed_interval` will be used. It is provided by the `Closed_interval` module.

Specification of the type `closed_interval`. This type gives each closed interval whose bounds are integers a *unique representation*. It is *equipped with* the five following functions:

- The *constructor* `cons_closed` constructs the representation of the interval $[a, b]$ from a pair (a, b) of integers where $a \leq b$.
- The *accessors* `lower_bound` et `upper_bound` access the bounds of an interval.
- The *conversion* function `string_of_closed_interval` gives a string representation of an interval.
- The *printing* function `print_closed_interval` prints such a string representation.

The interface definition of the module (i.e. the content of the `Closed_interval.mli` file) shows the types of these five functions:

```
type closed_interval
val cons_closed : int * int -> closed_interval
val lower_bound : closed_interval -> int
val upper_bound : closed_interval -> int
val string_of_closed_interval : closed_interval -> string
val print_closed_interval : closed_interval -> unit
```

It also indicates that the type `closed_interval` is *abstract*. The type definition phrase `type closed_interval` only contains the keyword `type` followed by the name `closed_interval`, indeed. After this there is nothing. Therefore the type representation is hidden.

Write definitions of functional values according to the following specifications:

1. Name: `width`. Type: `closed_interval -> int`. Computes the width of a closed interval.

2. Name: `is_member`. Type: `closed_interval -> bool`. Checks whether z belongs to i given a pair (z, i) where z is an integer and i is an interval.

3. Name: `cons_closed_1`. Type: `int * int -> closed_interval`. Computes the interval whose lower bound is z and width is n given a pair (z, n) where z is an integer and n is a non-negative integer.

4. Name: `are_disjoint`. Type: `closed_interval * closed_interval -> bool`. Checks whether two intervals are disjoint.

5. Name: `intersection`. Type: `closed_interval * closed_interval -> closed_interval`. Computes the intersection of two non-disjoint intervals.

6. Name: `is_subset`. Type: `closed_interval * closed_interval -> bool`. Checks whether $i \subseteq j$ given a pair (i, j) of intervals ($\subseteq$ means *is a proper subset of or is equal to*).

7. Name: `envelope`. Type: `closed_interval * closed_interval -> closed_interval`. Computes the envelope of two intervals, that is, the smallest interval that is a superset of both.

Test all these definitions:

8. First create a directory. Create in this directory a file containing all these definitions and whose name has extension `ml`. For instance `Use_closed_interval.ml`. Write also in this directory the content of the archive `Closed_interval`, that is, the files `Closed_interval.cmi` and `Closed_interval.cmo` (in other words: the module `Closed_interval`). Enter this directory. Build a Caml toplevel that contains the compiled implementation of the module `Closed_interval` preloaded at start-up. To do this, execute the Unix command

```
ocamlmktop -o mytoplevel Closed_interval.cmo
```

To run this new toplevel execute the Unix command

```
mytoplevel
```

In this toplevel first evaluate the Caml phrase

```
open Closed_interval
```

This opens the module `Closed_interval`, which allows to use short names instead of long ones in dotted notation (that is, for instance `lower_bound` instead of `Closed_interval.lower_bound`). Then evaluate the Caml phrase

```
#use "Use_closed_interval.ml"
```

This reads, compiles and executes all the source phrases from the given file. Now test each of the definitions, that is, apply each of the function values several times.

Type-inference by yourself

1. All the following Caml expressions are syntactically correct. Consider each of them. Has it a value? If it has a value then which one and of which type? Else why?

```
5 * 4 * 3 * 2 * 1 ;;
true ;;
if 2 > 0 then "true"
         else "false" ;;
if 2 > 0 then true
         else false ;;
if 2 > 0 then "yes"
         else "no" ;;
if 2 > 0 then yes
         else no ;;
if 2 > 0 then true
         else "false" ;;
3.14 +. 2. ;;
3.14 +. 2 ;;
3.14 + 2.0 ;;
3.14 > 2. ;;
3.14 > 2 ;;

if false then 5 / 0
         else 3 ;;
if false then 3
         else 5 / 0 ;;
"s" ^  "i" ;;
‘s‘ ^  "i" ;;
‘s‘ ^  ‘i‘ ;;
"s" ^ i ;;
si ;;
"if" ;;
function x -> x > 0 ;;
(function x -> x > 0) (12) ;;
function x -> 2 * x ;;
(function x -> 2 * x) (7) ;;
```

```
int_of_float (3.14) > 2 ;;
3.14 > float_of_int (2) ;;
3 > 2 ;;
"three" > "two" ;;
"two" > "one" ;;
5 / 2 ;;
5. /. 2. ;;
sqrt (4.) ;;
sqrt (4) ;;
sqrt ;;
exp (1.) ;;
log (exp (1.)) ;;
5 / 0 ;;
if true then 5 / 0
        else 3 ;;
if true then 3
        else 5 / 0 ;;

function (x, y) -> (x + y, x - y) ;;
function x -> x / 0 ;;
(function x -> x / 0) (12) ;;
(function x -> x) (5) ;;
(function x -> x) ("yes") ;;
function x -> x ;;
(function x -> 1) (5) ;;
(function x -> 1) (5.) ;;
function x -> 1 ;;
(function (x, y) -> (y, x)) (1, 2) ;;
(function (x, y) -> (y, x)) (1, "yes") ;;
(function (x, y) -> (y, x)) ("yes", "yes") ;;
function (x, y) -> (y, x) ;;
function x -> (x, x) ;;
(function x -> (x, x)) (1) ;;
```

2. All the following Caml phrases are syntactically correct. They are assumed to be evaluated in order. Consider each of them. Is it a value definition `let` $\ldots = e$ or an expression e? Has e a value? If it has a value then which one and of which type? Else why?

```
let absolute = function x -> if x >= 0 then x else - x ;;
absolute (absolute (-12)) ;;
let comp = function (x, y, z) -> (x <= y, y <= z) ;;
(comp (3, 2, 4), comp (2, 3, 4)) ;;
comp ("yellow", "red", "green") ;;
let approx = function (a, b) -> let (b1, b2) = comp (a - 1, b, a + 1) in b1 & b2 ;;
approx (12, 13) ;;
approx ("purple", "violet") ;;
let ff = function x -> x + 1 in ff ;;
ff (5) ;;
let ff = function x -> x + 1 ;;
ff (5) ;;
let ff = function x -> x + 1. ;;
ff (5) ;;
let ff = function x -> x +. 1. ;;
```

```
ff (5) ;;
(let ff = function x -> x + 1 in ff) (5) ;;
ff (5) ;;
ff ;;
```

3. Consider each of the following types. Write a value having this types (naming it is useless), evaluate it, then apply it:

```
int -> int
int * int -> bool
string -> bool
float * float -> string
bool * bool -> bool
int -> int * bool
```

4. Write values named `f` and `g` in such a manner that the following expression has a value:

```
f (f (g (2, true), 4))
```

5. All the following Caml phrases are syntactically correct. They are assumed to be evaluated in order (those in the left column being evaluated first). Consider each of them. Is it a value definition `let` $\ldots = e$ or an expression e? Has e a value? If it has a value then which one and of which type? Else why?

```
let a = 5
in 2 * a ;;
a ;;
let a = 7
in (a, a) ;;
let a = 1 and b = 3 and c = 4
in a + b + c ;;
let (a, b, c) = (1, 3, 4)
in a + b + c ;;
let a = 5 ;;
a ;;
let a = 3
in 2 * a ;;
a ;;
let a = "yes" ;;
a ;;
let x = 5 ;;
let x = 3
in 2 * x ;;
let y = x + 1 ;;
let x = 4 and y = 2
in x * y ;;
y ;;
let y = let x = 7 and y = 9
        in y - x ;;
let y = y = 9 ;;
failwith ;;
```

6. Same question.

```
let f = function
  x -> if x <> 0 then x / abs (x)
                 else failwith ("f : requires a non-zero value as argument") ;;
(f (-7), f (5)) ;;
f (0) ;;
let rec times = function (a, b) -> if b = 0 then 0
                                            else a + times (a, b - 1) ;;
times (5, 7) ;;
let rec modulo_8 = function n -> if n < 0 then modulo_8 (n + 8)
                                         else if n >= 8 then modulo_8 (n - 8)
                                                        else n ;;
modulo_8 (-13) ;;
let rec repeat = function (word, n) -> if n = 0 then "!"
                                                else word ^ repeat (word, n - 1) ;;
repeat ("Ah ", 5) ;;
let rec f = function x -> f (x) ;;
f (3) ;;
let rec power = function (x, n) ->
  if n = 0 then 1.0
```

```
          else if n < 0 then 1.0 /. power (x, - n)
                       else x *. power (x, n - 1) ;;
power (1.2, 4) ;;
power (5.0, -3) ;;
let rec is_even = function n -> if n = 0 then true
                                     else is_even (n - 2) ;;
is_even (12) ;;
is_even (11) ;;
```

3 Induction and recursion

3.1 Tutorial (draft)

3.1.1 The factorial function

In mathematics, the *factorial* of a natural number n is the product of all positive integers less than or equal to n, with the convention that the product of no numbers at all is 1. This is written as $n!$ and pronounced n *factorial*.

The *factorial function* is the function that maps any natural number n to $n!$.

This is a *recursive relation*

$n! = n(n-1)!$ for any positive natural number n.

Yet this recursive relation does not give a *definition* of the factorial function.
To get such a definition a *basis relation* is also needed.
The correct *basis relation* is

$0! = 1$

From the above *basis* and *recursive* relations this Objective Caml *recursive definition* of the factorial function immediately follows:

```
let rec f = function
  n -> (* n >= 0 *)
       if n = 0
         then 1
         else n * f (n - 1)
in
  f
```

The type of this value is `int -> int`.

This code shows that in Objective Caml

A functional value can be recursively defined with no name bound to it.

In the written code the name `f` is only *locally* bound to the functional value.
So, even *anonymous* functions can be recursively defined.

Yet a name, for instance `fact`, can be bound to such a functional value:

```
let rec fact = function
  n -> (* n >= 0 *)
       if n = 0
         then 1
         else n * fact (n - 1)
```

Consider the evaluation of `fact (i)` where `i` is bound to a non-negative value (obeying the specification given by the comment `(* n >= 0 *)`).
Look at the sub-expression `fact (n - 1)` in the code.
During the evaluation of `fact (i)`, each time the sub-expression `n - 1` is evaluated, its value is *closer to the basis* than the value of the formal parameter `n`.
This guarantees the termination of the evaluation of `fact (i)`.
Note that the assumption of `i` bound to a non-negative value is crucial.

3.1.2 A recursive definition of a type

In Objective Caml, recursive definitions can be given not only to functions but also to types.
For instance evaluating this code

```
type int_bin_tree = Empty                                     (* basis *)
                  | Plant of int * int_bin_tree * int_bin_tree  (* induction *)
```

defines a type whose values can be used to represent binary trees of integers.

So, there exist recursive definitions of types and of functions as well.
Clearly, a type is a sets of values. Moreover a function can be considered as a set, too, that is, as the set of pairs (x, y) where y is the image of x under the function.
Thus the notion of a recursive definition, usually called an *inductive definition*, of a set is basic.

3.1.3 Warning

From this *valid* mathematical formula

$\boxed{0! = 1 \text{ and } n! = n(n-1)! \text{ for any positive natural number } n}$

immediately follows a correct recursive definition of the factorial function.

Now consider another *valid* mathematical formula

$\boxed{a + b = b + a \text{ for any pair } (a, b) \text{ of integers}}$

from which follows this recursive definition

```
let rec add = function a, b -> add (b, a)
```

Any application of `add` loops.

So, consider both formulas and the meaning of the word *valid* in both cases.
The first formula is a *valid* ***definition*** of factorials.
The second formula is a *valid* expression of the commutativity of addition of integers.
But it is not a ***definition*** of the addition of integers.
She is also a *valid* expression of the commutativity of multiplication of integers or of addition of real numbers for instance.
Look at the type of `add`

```
val add : 'a * 'b -> 'c
```

It is fully polymorphic then it does not refer to a type whose values could represent integers.
Yet `add` is (bound to) a well-typed functional value! Therefore it can be applied ... but never returns a result. Its range is empty, that is, it is defined nowhere. Thus it is useless.

Given a functional value (or a type)

the useful formulas are those that **express a way to construct** this functional value (or this type).

To be such

a formula must contain a ***basis (case)***.

The formula related to `add` has no basis.
Similarly this code

```
type int_bin_tree = Plant of int * int_bin_tree * int_bin_tree  (* induction *)
```

does not contain a basis.

Yet evaluating this phrase does not fail. But no value of this type `int_bin_tree` can be expressed! Thus it is useless.

The existence of a *basis* is **necessary** but it is **not sufficient**.

For instance this does not define the predicate function checking whether a natural number is even

```
let rec is_even = function
  n -> (* n >= 0 *)
       if n = 0
         then true              (* basis *)
         else is_even (n - 2)   (* induction *)
```

This codes only defines the function indicating that an even number is even!

To get a correct definition the *basis* must be widened:

```
let rec is_even = function
  n -> (* n >= 0 *)
       if n <= 1
         then if n = 0
                then true       (* basis *)
                else false      (* basis *)
         else is_even (n - 2)   (* induction *)
```

3.1.4 Mutually recursive definitions

This code gives a correct definition of both the predicate functions checking whether a natural number is even and whether a natural number is odd

```
let rec
  is_even = function
    n -> (* n >= 0 *)
         if n = 0
           then true             (* basis *)
           else is_odd (n - 1)   (* induction *)
  and
  is_odd = function
    n -> (* n >= 0 *)
         if n = 0
           then false            (* basis *)
           else is_even (n - 1)  (* induction *)
```

This is a *mutually recursive definition.*

Similarly mutually recursive definitions of types can be written.

3.1.5 Writing correct recursive definitions of functions

A function can be considered as the set of pairs (x, y) where y is the image of x under the function.

Therefore from a mathematical standpoint writing correct recursive definitions of functions amounts to writing correct *inductive definitions* of sets.

Thus *inductive definitions* of sets are first considered.

3.1.6 Inductive definition of a set

An inductive definition of a set S is as follows, where B is a set of 0-*ary rules* and I is a set of *rules of positive arity*:

- **Basis**: if b is a rule in B then $b()$ is an element of S.
- **Induction**: if $(x_1, \ldots, x_n)$ is a tuple of elements of S and i is an n-ary rule in I that can be applied to $(x_1, \ldots, x_n)$ then $i(x_1, \ldots, x_n)$ is an element of S.
- **Closure**: any element in S results from applying finitely many times rules in B or in I.

As the *closure* part of the definition is always the same, its is usually omitted, as in the sequel.

3.1.7 Some inductive definitions of sets

The set $\mathbb{N}$ of all the **natural numbers** can be defined as follows

- **Basis**: $\mathit{zero}()$ is in $\mathbb{N}$.
- **Induction**: if x is in $\mathbb{N}$ then $\mathit{suc}(x)$ is in $\mathbb{N}$.

Here $B = \{\mathit{zero}\}$ and $I = \{\mathit{suc}\}$.

The set $\mathbb{N}$ of all the **natural numbers** can also be defined as follows

- **Basis**: $\mathit{zero}()$ and $\mathit{one}()$ are in $\mathbb{N}$.
- **Induction**: if x is in $\mathbb{N}$ then $\mathit{sucsuc}(x)$ is in $\mathbb{N}$.

Here $B = \{\mathit{zero}, \mathit{one}\}$ and $I = \{\mathit{suc}\}$.

The set $\mathbb{N}$ of all the **natural numbers** can be defined as follows, too

- **Basis**: $\mathit{zero}()$ is in $\mathbb{N}$.
- **Induction**:
  - If x is in $\mathbb{N} \setminus \{\mathit{zero}()\}$ then $d_0(x)$ is in $\mathbb{N}$.
  - If x is in $\mathbb{N}$ then $d_1(x)$ is in $\mathbb{N}$.

Here $B = \{\mathit{zero}\}$ and $I = \{d_0, d_1\}$.

The set of all the **lists of integers**, here denoted by L, can be defined as follows

- **Basis**: $\mathit{emptyList}()$ is in L.
- **Induction**: if x is in L and i is in $\mathbb{Z}$ then $\mathit{consList}(i)(x)$ is in L.

Here $B = \{\mathit{emptyList}\}$ and $I = \{\mathit{consList}(i), i \in \mathbb{Z}\}$.

The set $\mathbb{Z}[X]$ of all the **polynomials in one variable X and whose coefficients are rational numbers** can be defined as follows

- **Basis**: $\mathit{nullPoly}()$ is in $\mathbb{Z}[X]$.
- **Induction**: if p is in $\mathbb{Z}[X]$ and i is in $\mathbb{Z}$ then $\mathit{consPoly}(i)(p)$ is in $\mathbb{Z}[X]$.

Here $B = \{\mathit{nullPoly}\}$ and $I = \{\mathit{consPoly}(i), i \in \mathbb{Z}\}$.

Note that if $\mathit{consPoly}(i)(p)$ is interpreted as the polynomial $i + Xp$ then this inductive definition of $\mathbb{Z}[X]$ is well-suited for easily defining (using the Horner scheme) the function that evaluates a polynomial.

The set of all the **binary trees of integers**, here denoted by T, can be defined as follows

- **Basis**: $emptyTree()$ is in T.
- **Induction**: if (x_1, x_2) is a pair of elements of T and i is in $\mathbb{Z}$ then $plant(i)(x_1, x_2)$ is in T.

Here $B = \{emptyTree\}$ and $I = \{plant(i), i \in \mathbb{Z}\}$.

The set of all the **strings over an alphabet** A, here denoted by Str, can be defined as follows

- **Basis**: $emptyString()$ is in Str and if a in A then $LengthOneString(a)$ is in Str.
- **Induction**: if x is in Str and (a_1, a_2) is a pair of elements of A then $AddTwo(a_1, a_2)(x)$ is in Str.

Here $B = \{emptyString\} \cup \{LengthOneString(a), a \in A\}$ and $I = \{AddTwo(a_1, a_2), (a_1, a_2) \in A \times A\}$.

Note that if $AddTwo(a_1, a_2)(x)$ is interpreted as the string whose characters are a_1 followed by those of x in order followed by a_2 then this inductive definition of strings is well-suited for easily defining the predicate function that checks whether reading a string from left to right is as reading it from right to left.

3.1.8 Inductive structures

An inductive definition of a set S gives a way to construct any element of S.

This is useful when defining a function whose application returns results in S (or results that are tuples at least one component of which is in S).

But this is not immediately useful when recursively defining a function whose arguments are in S (or are tuples at least one component of which is in S).

While the inductive definition of S gives a way to ***synthesize*** any element of S, what is required here is a way to ***analyze*** any element of S.

What is required is called from now on

an ***inductive structure over*** S, that is, a set of functions:

- ***Constructors***: a 0-ary *basic constructor* (function) for each rule in the basis, a n-ary *inductive constructor* (function) for each n-ary inductive rule and if necessary ***predicate functions*** that check whether inductive constructors can be applied.
- ***Accessors***: functions that give access to the components of the non-atomic elements of S (i.e., not in the basis of S) and ***predicate functions*** that check whether these functions can be applied. At least one such predicate function is required in order to check whether an element of S is atomic. Moreover if there is more than one atomic element then other predicate functions are required in order to identify each atomic element.

In the sequel several inductive structures over types (i.e., sets of values) are given.

An inductive structure over a type makes it of practical use because its values can then be synthesized or analyzed.

Yet this is true only if the internal representation of values is accessible.

However as most of the types given in the sequel will be made *abstract*, the functions in their inductive structures will not be sufficient for practical use (internal representations will be hidden). Functions that return string representations of values will be necessary. Thus each type will be equipped with a set of functions: those in its inductive structure and one or several functions that return string representations of its values.

3.1.9 Natural numbers

The non-built-in type `nat_1` is provided by the `Nat_1` module.

Specification of the type `nat_1`. This type gives each natural number a *unique representation*. It is *equipped with* these five functions:

- These *constructors*:
  - The *basic constructor* `zero` constructs the representation of the natural number zero.
  - The *inductive constructor* `suc` constructs the representation of the successor of a natural number.
- The *predicate function* `is_zero` checks whether a natural number is zero.
- The *accessor* `pre` accesses the predecessor of a positive natural number.
- The *printing* function `print_nat` prints a string representation of a natural number.

The interface definition of the module (i.e. the content of the `Nat_1.mli` file) shows the types of these five functions:

```
type nat_1
val zero : unit -> nat_1
val suc : nat_1 -> nat_1
val is_zero : nat_1 -> bool
val pre : nat_1 -> nat_1
val print_nat : nat_1 -> unit
```

It also indicates that `nat_1` is an *abstract* type.

Use `nat_1`. For each of the following specifications of functional values first write a mathematical formula then write a recursive definition based on the formula and meeting the specification:

Name: `add`. Type: `nat_1 * nat_1 -> nat_1`. Adds two natural numbers.

A mathematical formula:

For all natural numbers m: $m + 0 = m$

For all natural numbers m: for all positive natural numbers n: $m + n = (m + (n - 1)) + 1$

This recursive definition of `add` follows:

```
let rec add = function
  m, n -> if is_zero (n)
            then m
            else suc (add (m, pre (n)))
```

Name: `mult`. Type: `nat_1 * nat_1 -> nat_1`. Multiply two natural numbers.

A mathematical formula:

For all natural numbers m: $(m)(0) = 0$

For all natural numbers m: for all positive natural numbers n: $mn = (m(n-1)) + m$

This recursive definition of `add` follows:

```
let rec mult = function
  m, n -> if is_zero (n)
            then zero ()
            else add (mult (m, pre (n)), m)
```

Name: `greater_than`. Type: `nat_1 * nat_1 -> bool`. Tests whether $m > n$ given a pair (m, n) of natural numbers.

A mathematical formula:

For all natural numbers n: $0 \leq n$

For all positive natural numbers m: $m > 0$

For all positive natural numbers m: for all positive natural numbers n: $m > n$ if and only if $m - 1 > n - 1$

This recursive definition of `add` follows:

```
let rec greater_than = function
  m, n -> if is_zero (m)
            then false
            else
              if is_zero (n)
                then true
                else greater_than (pre (m), pre (n))
```

Name: `subtract`. Type: `nat_1 * nat_1 -> nat_1`. Returns the natural number $m - n$ given a pair (m, n) of natural numbers where m is greater than of equal to n.

A mathematical formula:

For all natural numbers m: $m - 0 = m$

For all natural numbers m: for all positive natural numbers n less than or equal to m: $m - n = (m - (n-1)) - 1$

This recursive definition of `add` follows:

```
let subtract =
  let rec f = function
    m, n -> if is_zero (n)
              then m
              else pre (f (m, pre (n)))
  in
    function
      m, n -> if greater_than (n, m)
                then failwith ("subtract : the first argument must be >= to the second one")
                else f (m, n)
```

The function `failwith` has type `string -> 'a`. Its application stops the computation and prints its argument.

In Objective Caml there exist ways of dealing with run-time errors much better than using `failwith`!

Name: `nat_of_int`. Type: `int -> nat_1`. Converts a non-negative `int` value to the corresponding `nat_1` value.

A mathematical formula:

0 is the first natural number

For all positive natural numbers n: n is the successor of $n - 1$

This recursive definition of `add` follows:

```
let nat_of_int =
  let rec f = function n -> (* n >= 0 *)
                          if n = 0 then zero ()
                                   else suc (f (n - 1))
in
  function
    z -> if z >= 0
           then f (z)
           else failwith ("nat_of_int : requires a non-negative integer as its argument") ;;
```

Name: `int_of_nat`. Type: `nat_1 -> int`. Converts a `nat_1` value to the corresponding `int` value.

A mathematical formula:

The first natural number is 0

For all non-zero natural numbers n: n is $m + 1$ where m is the predecessor of n

This recursive definition of `add` follows:

```
let rec int_of_nat = function
  n -> if is_zero (n)
         then 0
         else int_of_nat (pre (n)) + 1
```

3.1.10 Writing correct recursive definitions of functions - Again

To say a recursive definition of a functional value (assume it is named `f`) is *correct* has no meaning if there does not exist a specification for `f`.

So, assume there exists a specification for `f`.
Then:

Saying a recursive definition of `f` is *correct* means that it meets this specification.

A specification for `f` says . . .

Correctness means both:

- *Partial correctness*: if an application of `f` to an argument in its specified domain terminates then its result meets the specification.
- *Termination*: any application of `f` to an argument in its specified domain produces a result, which means:
  - *No infinite looping.*
  - *No run-time error.*

To prove the correctness of a recursively-defined function, use induction!

Proving *partial correctness* is usually more difficult than proving *termination*.
Here *termination* only will be considered.
However there will be no proof.
Rules to ensure termination are given below.

For each value `x` write `f (x)` as an expression e such that:

- *Basis rule*: If `x` is *in the basis*: no application of `f` in e.
- *Induction rule*: If `x` is <u>not</u> *in the basis*: any application of `f` in e has argument *closer to the basis than* `x`.

3.2 Exercises

Each time an expression f with tuple of formal parameters p intended to have a functional value is written, consider each sub-expression $g(e)$ in f (that is, each application) and do the following:

- *To avoid run-time errors: Look at the specification of g and check whether g can be applied to the tuple e of arguments in $g(e)$.*
- *To avoid infinite looping: If g is f (recursive call of f) then check whether the tuple e of arguments in $f(e)$ is closer to the basis than the tuple p of formal parameters of f.*

3.2.1 Exercise 1 - Natural numbers again

The non-built-in type `nat_2` is provided by the `Nat_2` module.

Specification of the type `nat_2`. This type gives each natural number a *unique representation*. It is *equipped with* these seven functions:

- These *constructors*:
  - The *basic constructors* `zero` and `one` construct the representations of the natural numbers zero and one.
  - The *inductive constructor* `sucsuc` constructs the representation of the successor of the successor of a natural number.

- The *predicate functions* `is_zero` and `is_one` check whether a natural number is zero and whether a natural number is one.
- The *accessor* `prepre` accesses the predecessor of the predecessor of a natural number greater than one.
- The *printing* function `print_nat` prints a string representation of a natural number.

The interface definition of the module (i.e. the content of the `Nat_2.mli` file) shows the types of these seven functions:

```
type nat_2
val zero : unit -> nat_2
val one : unit -> nat_2
val sucsuc : nat_2 -> nat_2
val is_zero : nat_2 -> bool
val is_one : nat_2 -> bool
val prepre : nat_2 -> nat_2
val print_nat : nat_2 -> unit
```

It also indicates that `nat_2` is an *abstract* type.

Use `nat_2`. For each of the following specifications of functional values first write a mathematical formula then write a recursive definition based on the formula and meeting the specification:

1. Name: `is_even`. Type: `nat_2 -> bool`. Tests whether a natural number is even.

2. Name: `two_times`. Type: `nat_2 -> nat_2`. Returns two times a natural number.

3. Name: `suc`. Type: `nat_2 -> nat_2`. Returns the successor of a natural number.

4. Name: `half`. Type: `nat_2 -> nat_2`. Returns the half of a natural number.

5. Name: `add`. Type: `nat_2 * nat_2 -> nat_2`. Adds two natural numbers.

6. Name: `pre`. Type: `nat_2 -> nat_2`. Returns the predecessor of positive natural number.

7. Name: `nat_of_int`. Type: `int -> nat_2`. Converts a non-negative `int` value to the corresponding `nat_2` value.

3.2.2 Exercise 2 - Natural numbers once more

The non-built-in type `nat_3` is provided by the `Nat_3` module.

Specification of the type `nat_3`. This type gives each natural number a *unique representation*. It is *equipped with* these eight functions:

- These *constructors*:
  - The *basic constructor* `zero` constructs the representation of the natural number zero.
  - The *inductive constructor* `d0` constructs the representation of the natural number that is two times a positive natural number, and the *inductive constructor* `d1` constructs that of the natural number that is the successor of two times a natural number.

- The *predicate functions* `is_even_non_zero` and `is_odd` check whether a natural number is even and positive, and whether a natural number is odd.
- The *accessors* `inv_d0` and `inv_d1` are the inverses of the functions `d0` and `d1`.
- The *printing* function `print_nat` prints a string representation of a natural number.

The interface definition of the module (i.e. the content of the `Nat_3.mli` file) shows the types of these eight functions:

```
type nat_3
val zero : unit -> nat_3
val d0 : nat_3 -> nat_3
val d1 : nat_3 -> nat_3
val is_even_non_zero : nat_3 -> bool
val is_odd : nat_3 -> bool
val inv_d0 : nat_3 -> nat_3
val inv_d1 : nat_3 -> nat_3
val print_nat : nat_3 -> unit
```

It also indicates that `nat_3` is an *abstract* type.

Use `nat_3`. For each of the following specifications of functional values first write a mathematical formula then write a recursive definition based on the formula and meeting the specification:

1. Write the value of type `nat_3` that represents the natural number two.

2. Write the value of type `nat_3` that represents the natural number three.

3. Name: `is_zero`. Type: `nat_3 -> bool`. Tests whether a natural number is zero.

4. Name: `is_power_of_two`. Type: `nat_3 -> bool`. Tests whether a natural number is a power of two.

5. Name: `suc`. Type: `nat_3 -> nat_3`. Returns the successor of a natural number.

6. Name: `add`. Type: `nat_3 * nat_3 -> nat_3`. Adds two natural numbers.

7. Name: `pre`. Type: `nat_3 -> nat_3`. Returns the predecessor of <u>positive</u> natural number.

8. Name: `nat_of_int`. Type: `int -> nat_3`. Converts a <u>non-negative</u> `int` value to the corresponding `nat_3` value.

3.2.3 Exercise 3 - Lists

For any type `t` (giving a representation of a set A) the type `t list` is provided by the `List_1` module.

Specification of the type `t list`. This type gives each list of elements of A a *representation* (or a *unique representation* if and only if `t` gives each element of A a unique representation). It is *equipped with* these five functions:

- These *constructors*:
  - The *basic constructor* `empty_list` constructs the representation of the empty list.
  - The *inductive constructor* `cons_list` constructs a (or the) representation of the list whose first element is x and whose rest is lx given a pair (x, lx) where x is an element of A and lx is a list of elements of A.
- The *predicate function* `is_empty_list` checks whether a list of elements of A is empty.
- The *accessors* `head` and `tail` access the first element and the rest of a non-empty list of elements of A.

The interface definition of the module (i.e. the content of the `List_1.mli` file) shows the types of these five functions:

```
val empty_list : unit -> 'a list
val cons_list : 'a * 'a list -> 'a list
val is_empty_list : 'a list -> bool
val head : 'a list -> 'a
val tail : 'a list -> 'a list
```

It also indicates that these functions are *polymorphic*: `t` can be *any* type, which is expressed by the *type variable* `'a`.

Use lists. For each of the following specifications of functional values write a recursive definition that meets the specification:

1. Name: `length`. Type: `a list -> int`. Computes the length of a list. Examples:

`[ ]` $\longmapsto$ `0` `[1 ; 2 ; 3]` $\longmapsto$ `3` `["" ; ""]` $\longmapsto$ `2` `[[ ]]` $\longmapsto$ `1`

2. Name: `is_member`. Type: `a * 'a list -> bool`. Tests whether x is a member of the list lx given the pair (x, lx). Examples:

`('a', [ ])` $\longmapsto$ `false` `(3, [2 ; 2])` $\longmapsto$ `false` `("", ["" ; "la" ; ""])` $\longmapsto$ `true`

3. Name: `map_twice`. Type: . Computes the list of twice each element in a list of integers. Examples:

`[ ]` $\longmapsto$ `[ ]` `[0 ; 0 ; -5 ; 3 ; 1]` $\longmapsto$ `[0 ; 0 ; -10 ; 6 ; 2]`

4. Name: `less_than`. Type: `int -> int list`. Computes the strictly decreasing list of the non-negative integers less than an integer. Examples:

`0` $\longmapsto$ `[ ]` `5` $\longmapsto$ `[4 ; 3 ; 2 ; 1 ; 0]`

5. Name: `even_only`. Type: `int list -> int list`. Computes the sublist of the even numbers in a list of integers. Examples:

`[ ]` $\longmapsto$ `[ ]` `[3 ; 5]` $\longmapsto$ `[ ]` `[2 ; 2]` $\longmapsto$ `[2 ; 2]` `[-3 ; 0 ; -3]` $\longmapsto$ `[0]`

6. Name: `map_length`. Type: `'a list list -> int list`. Computes the list of the lengths of the elements in a list of lists. Examples:

`[ ]` $\longmapsto$ `[ ]`   `[[1] ; [ ]]` $\longmapsto$ `[1 ; 0]`   `[[1 ; 2] ; [3]]` $\longmapsto$ `[2 ; 1]`   `[[[ ]]]` $\longmapsto$ `[1]`

7. Name: `combine`. Type: `'a list * 'b list -> ('a * 'b) list`. Transforms a pair of lists having the same length into a list of pairs. Examples:

`([ ], [ ])` $\longmapsto$ `[ ]`   `([0 ; 2], [1 ; 3])` $\longmapsto$ `[(0, 1) ; (2, 3)]`

8. Name: `split`. Type: `('a * 'b) list -> 'a list * 'b list`. Transforms a list of pairs into a pair of lists. Examples:

`[ ]` $\longmapsto$ `([ ], [ ])`   `[(0, 1) ; (2, 3)]` $\longmapsto$ `([0 ; 2], [1 ; 3])`

9. Name: `insert`. Type: `'a * 'a list -> 'a list`. Order-preserving insertion of a value into an increasing list (according to `<`). Examples:

`("yes", [ ])` $\longmapsto$ `["yes"]`   `(7, [2 ; 3 ; 3 ; 6 ; 8])` $\longmapsto$ `[2; 3; 3; 6; 7; 8]`

10. Name: `sort`. Type: `'a list -> 'a list`. Sorts a list in increasing order (according to `<`).

3.2.4 Exercise 4 - Lists of closed intervals whose bounds are integers

Consider lists of closed interval whose bounds are integers. To deal with such intervals use the non-built-in type `closed_interval` provided by the `Closed_interval` module. To deal with lists use the functions provided by the `List_1` module.

Use lists of closed interval whose bounds are integers. For each of the following specifications of functional values write a recursive definition that meets the specification:

1. Name: `map_width`. Type: `closed_interval list -> int list`. Computes the list of the widths of the elements in a list of intervals.

2. Name: `are_embedded`. Type: `closed_interval list -> bool`. Tests whether a list of intervals is in increasing order (according to set inclusion).

Type-inference by yourself

To check your solution enter an Objective Caml toplevel from a directory containing the files `List_1.cmi` and `List_1.cmo` (in other words: the module `List_1`) to be extracted from `this archive`. Then in this toplevel first evaluate this Caml code:

```
#load "List_1.cmo" ;;

open List_1 ;;
```

This loads the module `List_1` then opens it, which allows to use short names instead of long ones in dotted notation (that is, for instance `cons_list` instead of `List_1.cons_list`).

All the following Caml phrases are syntactically correct. They are assumed to be evaluated in order. Consider each of them. Is it a value definition `let` $\ldots = e$ or an expression e? Has e a value? If it has a value then which one and of which type? Else why?

```
let li_1 = cons_list (7, cons_list (3, cons_list (11, empty_list ()))) ;;
cons_list (2, 3) ;;
cons_list (empty_list (), 7) ;;
let li_2 = cons_list (9, li_1)
in li_2 ;;
li_1 ;;
li_2 ;;
empty_list () ;;
empty_list ;;
let li_2 = cons_list (6, cons_list (5, cons_list (9, empty_list ()))) ;;
cons_list (li_2, cons_list (li_1, empty_list ())) ;;
cons_list (6, cons_list (li_1, empty_list ())) ;;
let li_3 = cons_list (6, cons_list (5.3, cons_list (9, empty_list ()))) ;;
let li_4 = cons_list ("bear", cons_list ("cow", cons_list ("fox", empty_list ()))) ;;
let li_5 = cons_list ("honey", cons_list ("grass", cons_list ("hen", empty_list ()))) ;;
cons_list (li_4, li_5) ;;
tail (tail (cons_list ("yes", empty_list ()))) ;;
cons_list ("yes", empty_list ()) ;;
cons_list ((2, 3.5), cons_list ((0, 4.), empty_list ())) ;;
head (tail (cons_list ((if 4 > 2 then 3 < 5
                                 else 3 > 5),
         l             empty_list ()))) ;;
cons_list ((if 4 > 2 then 3 < 5
                     else 3 > 5),
           empty_list ()) ;;
cons_list (li_1, (cons_list (empty_list (), cons_list (cons_list (4, li_2), empty_list ())))) ;;
tail (cons_list (5, empty_list ())) ;;
let rec f1 = function li -> if is_empty_list (tail (li)) then head (li)
                                                         else f1 (tail (li)) ;;
let rec f2 = function (li, n) -> if n = 1 then head (li)
                                          else f2 (tail (li), n - 1) ;;
cons_list (empty_list (), empty_list ()) ;;
cons_list (cons_list (empty_list (), empty_list ()), empty_list ()) ;;
cons_list (sqrt, cons_list (log, cons_list (exp, empty_list ()))) ;;
cons_list (empty_list, empty_list ()) ;;
cons_list (empty_list (), empty_list) ;;
cons_list (empty_list, empty_list) ;;
let rec f3 = function (li, li1) ->
  if is_empty_list (li)
    then li1
```

```
    else if is_empty_list (li1)
           then li
           else if head (li) <= head (li1)
                  then cons_list (head (li), f3 (tail (li), li1))
                  else cons_list (head (li1), f3 (li, tail (li1))) ;;
let rec f4 = function li ->
  if is_empty_list (li)
    then (li, empty_list ())
    else if is_empty_list (tail (li))
           then (li, empty_list ())
           else let (h1, h2) = f4 (tail (tail (li)))
                in (cons_list (head (li), h1), cons_list (head (tail (li)), h2)) ;;
let rec f5 = function li ->
  if is_empty_list (li) then li
                        else if is_empty_list (tail (li))
                                then li
                                else let (h1, h2) = f4 (li)
                                     in f3 (f5 (h1), f5 (h2)) ;;
let rec f6 = function (li, li1 ) ->
  if is_empty_list (li) & is_empty_list (li1)
    then empty_list ()
    else cons_list ((head (li), head (li1)),
                f6 (tail (li), tail (li1))) ;;
let rec f7 = function li -> if is_empty_list (li) then f7 (cons_list (0, li))
                                                  else f7 (li) ;;
f7 (cons_list (2, empty_list ())) ;;
let rec f8 = function li -> if is_empty_list (li) then tail (li)
                                                  else f8 (tail (li)) ;;
f8 (cons_list (2, empty_list ())) ;;
let rec f9 = function
  li -> if is_empty_list (li)
          then empty_list ()
          else if is_empty_list (tail (li)) then head (li) else tail (li) ;;
f9 (cons_list (2, empty_list ())) ;;
let rec f10 = function li ->
  if is_empty_list (li) then li
                        else cons_list (head (li), f10 (tail (li))) ;;
f10 (cons_list (3, cons_list (2, empty_list ()))) ;;
head (tail (tail (li_1))) ;;
is_empty_list (cons_list (3.5, empty_list ())) ;;
is_empty_list (tail (tail (tail (li_4)))) ;;
is_empty_list (empty_list ()) ;;
(f1 (li_1), f1 (li_4)) ;;
let li = f5 (f3 (li_1, li_2))
in (f2 (li, 1), f2 (li, 2), f2 (li, 3), f2 (li, 4), f2 (li, 5), f2 (li, 6)) ;;
li ;;
let li = f5 (f3 (li_4, li_5))
in (f2 (li, 1), f2 (li, 2), f2 (li, 3), f2 (li, 4), f2 (li, 5), f2 (li, 6)) ;;
let li = f6 (li_4, li_5)
in (f2 (li, 1), f2 (li, 2), f2 (li, 3)) ;;
```

4 Higher-order functions

4.1 Tutorial (draft)

Assume an Objective Caml toplevel in which the module `List_1` has been loaded and opened. This provides functions as `cons_list` and so on which can be used with short names instead of long ones in dotted notation (that is, for instance `cons_list` instead of `List_1.cons_list`).

4.1.1 Functions - Abstraction and application again

Recall.

First three values:

```
2 + (3 * 4) ;;

5 + (0 * 8) ;;

4 + (1 * 7) ;;
```

Then the same values expressed using local definitions:

```
let
  a = 2 and b = 3 and c = 4
in
  a + (b * c) ;;

let
  a = 5 and b = 0 and c = 8
in
  a + (b * c) ;;

let
  a = 4 and b = 1 and c = 7
in
  a + (b * c) ;;
```

Now the abstraction step:

```
function a, b, c -> a + (b * c) ;;
```

Finally the application step (three times):

```
(function a, b, c -> a + (b * c)) (2, 3, 4) ;;

(function a, b, c -> a + (b * c)) (5, 0, 8) ;;

(function a, b, c -> a + (b * c)) (4, 1, 7) ;;
```

Recall again. Here *the startpoint consists* of *values of expressions containing applications of functions*. As the *abstraction step deals with these functions* it leads to a *higher-order functional value* whose *parameters are functional*.

First three values:

```
3.  *. sin (2.)       +. 0.5 *. cos (4.) ;;

7.2 *. abs_float (3.) +. 8.  *. exp (6.1) ;;

5.  *. log (10.5)     +. 3.1 *. sin (0.7) ;;
```

Then the same values expressed using local definitions:

```
let
  a = 3. and b = sin and c = 2. and d = 0.5 and e = cos and f = 4.
in
  a *. b (c) +. d *. e (f) ;;

let
  a = 7.2 and b = abs_float and c = 3. and d = 8. and e = exp and f = 6.1
in
  a *. b (c) +. d *. e (f) ;;

let
  a = 5. and b = log and c = 10.5 and d = 3.1 and e = sin and f = 0.7
in
  a *. b (c) +. d *. e (f) ;;
```

Now the abstraction step:

```
function a, b, c, d, e, f -> a  *. b (c) +. d *. e (f) ;;
```

Finally the application step (three times):

```
(function a, b, c, d, e, f -> a  *. b (c) +. d *. e (f)) (3., sin, 2., 0.5, cos, 4.) ;;

(function a, b, c, d, e, f -> a  *. b (c) +. d *. e (f)) (7.2, abs_float, 3., 8., exp, 6.1) ;;

(function a, b, c, d, e, f -> a  *. b (c) +. d *. e (f)) (5., log, 10.5, 3.1, sin, 0.7) ;;
```

More about higher-order functions.

Here *the startpoint consists of functional values*. As *the abstraction step deals with these functions* it leads to a *higher-order functional value* whose *results of applications are functional values*.

First three values:

```
function x -> 1 + x ;;

function x -> 2 + x ;;

function x -> 3 + x ;;
```

Then the same values expressed using local definitions:

```
let
  z = 1
in
  function x -> z + x ;;

let
  z = 2
in
  function x -> z + x ;;

let
  z = 3
in
  function x -> z + x ;;
```

Now the abstraction step:

```
function z -> (function x -> z + x) ;;
```

Then the application step (three times):

```
(function z -> (function x -> z + x)) (1) ;;

(function z -> (function x -> z + x)) (2) ;;

(function z -> (function x -> z + x)) (3) ;;
```

Finally compare applications of results of applications of this functional value with applications of the initial functional values:

```
(function x -> 1 + x) (5) ;;

((function z -> (function x -> z + x)) (1)) (5) ;;

(function x -> 2 + x) (7) ;;

((function z -> (function x -> z + x)) (2)) (7) ;;

(function x -> 3 + x) (9) ;;

((function z -> (function x -> z + x)) (3)) (9) ;;
```

Three examples of a function whose applications result in functions (note that the value bound to `unary_of_int_add` is that in the previous example):

```
(** Returns the function that adds x and an integer, given the integer x *)
(* *)
let unary_of_int_add = function x -> (function z -> x + z) ;;

(*****)

(** Returns the function that multiplies x and a real number, given the real number x *)
(* *)
let unary_of_real_mult = function x -> (function z -> x *. z) ;;

(*****)

(** Returns the function that concatenates x before a string, given the string x *)
(* *)
let unary_of_string_concat = function x -> (function z -> x ^ z) ;;
```

In the sequel will also be written *higher-order functions whose parameters are functions and whose results of applications are functions*, too.

4.1.2 Higher-order: two well-known examples

Function composition

First consider mathematics.

Recall that the *function composition* of two or more functions uses the output of one function as the input of another. For instance, f such that $f(x) = \cos(x^2)$ is the composition of the cos function and the squaring function (here denoted by sq), which is usually written $\cos \circ sq$.

Now consider Objective Caml:

```
let compose = function (g, f) -> (function x -> g (f (x)))
```

is a high-order function whose type is

```
('a -> 'b) * ('c -> 'a) -> ('c -> 'b)
```

Then

```
compose (cos, (function x -> x *. x))
```

represents the mathematical function $\cos \circ sq$.

Its type is

```
float -> float
```

Here is an example of applying it:

```
(compose (cos, (function x -> x *. x))) (-4.3)
```

Derivation

Here *numerical* derivation is considerd rather than *symbolic* (also called *formal*) derivation.

A functional value that computes an approximate derivative of a function from real numbers to real numbers is bound to `deriv`:

```
let deriv = function
  f -> (function x -> let
                         dx = 1e-7
                       in
                         (f (x +. dx) -. f (x -. dx)) /. (2.0 *. dx))
```

The type of `deriv` is

```
(float -> float) -> (float -> float)
```

Here is an example of applying it. The result approximately represents the mathematical function cos:

```
let approx_cos = deriv (sin)
```

Applying `approx_cos` to the `float` value `0.` returns:

```
0.999999999999998335
```

4.1.3 Higher-order functions

This is an approximate definition of the notion of a *higher-order function*:

A *higher-order function* is a function whose argument is a function and/or whose result of application is a function.

To get a correct definition, replace all occurrences of *a function* but the first one with *a tuple at least one component of which is a function*:

A *higher-order function* is a function whose argument is a tuple at least one component of which is a function and/or whose result of application is a tuple at least one component of which is a function.

This is another correct definition:

A *higher-order function* is a function whose type contains at last two arrows.

For instance here are the types of functional values previously written:

```
val unary_of_int_add : int -> (int -> int)

val unary_of_real_mult : float -> (float -> float)

val unary_of_string_concat : string -> (string -> string)
```

Here are the types of the function `make` provided by the Objective Caml `String` module (`(String.make (n)) (c)` returns a fresh string of length `n`, filled with the character `c`) and of the function `append` (catenates two lists and has the same meaning as the infix operator `@`) provided by its `List` module:

```
val make : int -> (char -> string)

val append : 'a list -> ('a list -> 'a list)
```

4.1.4 Associativity

The `->` *operator* is *right-associative* in type expressions as well as in function definitions.

For instance

```
int -> char -> string
```

means

```
int -> (char -> string)
```

For instance

```
function x -> function z -> x + z
```

means

```
function x -> (function z -> x + z)
```

Function application is *left-associative.*

For instance, assuming `unary_of_int_add` is bound to the value of `function x -> (function z -> x + z)`.

Then:

```
unary_of_int_add 5 8
```

means

```
(unary_of_int_add 5) 8
```

which can also be written

```
(unary_of_int_add (5)) (8)
```

4.1.5 Mapping

Consider the following codes

```
(** Computes the list of twice each element in a list of integers *)
(* *)
let rec map_twice = function
  li -> if is_empty_list (li)
          then empty_list ()
          else cons_list (2 * head (li), map_twice (tail (li)))

(** Computes he list of the lengths of the elements in a list of lists *)
(* *)
let rec map_length = function
  llx -> if is_empty_list (llx)
           then empty_list ()     (* ----- llx : type error ----- *)
           else cons_list (List.length (head (llx)), map_length (tail (llx)))
```

Look at the definitions of `map_twice` and `map_length`. Clearly their structures are similar.

Transform what is identical in them into a functional value (named `map_to_all`) that can be applied to obtain the value of `map_twice` or that of `map_length`:

```
let map_to_all = function
  f -> (let rec g = function
          lx -> if is_empty_list (lx)
                  then empty_list ()
                  else cons_list (f (head (lx)), g (tail (lx)))
        in
          g)
```

The type of `map_to_all` is

```
('a -> 'b) -> ('a list -> 'b list)
```

An example of application of the *higher-order* function `map_to_all`:

```
(** Computes the list of three times each element in a list of integers *)
(* *)
map_to_all (function x -> 3 * x)
```

Now an example of application of the result of this application of the *higher-order* function `map_to_all`:

```
(** Computes the list of three times each element in a list of integers *)
(* *)
(map_to_all (function x -> 3 * x)) ([1 ; 2 ; 3])
```

Note that `map_to_all` and `List.map` (the function `map` provided by the Objective Caml module `List`) have the same value.

Finally note that the value of `map_to_all` can also be *recursively* defined:

```
let rec map_to_all_1 = function
  f -> (function
          lx -> if is_empty_list (lx)
                  then empty_list ()
                  else cons_list (f (head (lx)), (map_to_all_1 (f)) (tail (lx))))
```

4.1.6 An accumulator

Consider the following codes

```
(** Summation of all the integers in a list *)
(* *)
let rec sum_int_list = function
  lx -> if is_empty_list (lx)
          then 0
          else (function x, y -> x + y) (head (lx), sum_int_list (tail (lx)))

(** Product of all the reals in a list *)
(* *)
let rec prod_float_list = function
  lx -> if is_empty_list (lx)
          then 1.
          else (function x, y -> x *. y) (head (lx), prod_float_list (tail (lx)))
```

```
(** Logical conjunction of all the booleans in a list *)
(* *)
let rec and_list = function
  lx -> if is_empty_list (lx)
          then true
          else (function x, y -> x & y) (head (lx), and_list (tail (lx)))
```

Look at the definitions of `sum_int_list`, `prod_float_list` and `and_list`. Clearly their structures are similar.

Transform what is identical in them into a functional value (named `accumul_right`) that can be applied to obtain the values of `sum_int_list`, of `prod_float_list` and of `and_list` :

```
let accumul_right = function
  (op, neutral) -> (let rec f = function
                      lx -> if is_empty_list (lx)
                              then neutral
                              else op (head (lx), f (tail (lx)))
                    in
                      f)
```

The type of `accumul_right` is

```
('a * 'b -> 'b) * 'b -> ('a list -> 'b)
```

An example of application of the *higher-order* function `accumul_right`:

```
(** Logical disjunction of all the booleans in a list *)
(* *)
accumul_right ((function x, y -> x or y), false)
```

Now an example of application of the result of this application of the *higher-order* function `accumul_right`:

```
(** Computes the list of three times each element in a list of integers *)
(* *)
(accumul_right ((function x, y -> x or y), false)) ([2 + 2 = 5 ; 6 - 3 = 9])
```

4.1.7 Currying

Currying is the technique of transforming a function that takes multiple arguments into a function that takes a single argument (the first of the arguments to the original function) and returns a new function that takes the remainder of the arguments and returns the result.

Uncurrying is the reverse transformation.

This a definition of the function (named `curry`) that computes the curried version of a function with two arguments:

```
let curry = function f -> (function x -> (function y -> f (x, y)))
```

Here are applications of `curry` that return functional values previously written:

```
(** The same function as unary_of_int_add. *)
(* *)
curry (function x, y -> x + y) ;;

(*****)

(** The same function as unary_of_real_mult. *)
(* *)
curry (function x, y -> x *. y) ;;

(*****)

(** The same function as unary_of_string_concat. *)
(* *)
curry (function x, y -> x ^ y) ;;
```

This a definition of the function (named `uncurry`) that uncurries the curried version of a function with two arguments:

```
let uncurry = function f -> (function (x, y) -> (f (x)) (y))
```

Almost all the functions provided by the modules in the libraries of the Objective Caml system only take a single argument.

Multiple-arguments functions can be considered as *syntactic sugar* for nested one-argument functions.

Here are applications of `curry` that computes addition of two integers, multiplication of two real numbers and concatenation of two strings:

```
uncurry (unary_of_int_add) ;;

uncurry (unary_of_real_mult) ;;

uncurry (unary_of_string_concat) ;;
```

4.2 Exercises

Assume an Objective Caml toplevel in which the module `List_1` has been loaded and opened. This provides functions as `cons_list` and so on and they can be used with short names instead of long ones in dotted notation (that is, for instance `cons_list` instead of `List_1.cons_list`).

4.2.1 Exercise 1 - Logarithms

First consider mathematics.

Given $a \in \mathbb{R}^{+*} \setminus \{1\}$ the domain of the *logarithm* function $\log_a$ (logarithm in *base* a) is $\mathbb{R}^{+*}$ and $\log_a(x) = \frac{\ln(x)}{\ln(a)}$ for any x in this domain (where ln denotes the natural logarithm function).

Now consider programming.

Using `float` values to represent real numbers write the type of the functional value that computes $\log_a$ given $a \in \mathbb{R}^{+*} \setminus \{1\}$. Then write a definition of this functional value (using `log` whose type is `float -> float` and which represents the natural logarithm function). Name this functional value `log_in_base`.

Use `log_in_base` to write values that represent $\log_{10}$ and $\log_2(1024)$ and write their types.

4.2.2 Exercise 2 - An accumulator

Consider `accumul_right`.

1. Write a recursive definition of `accumul_right_1` (i.e., a phrase that begins with `let rec accumul_right_1`, not with `let accumul_right_1`) that binds this name to the same value as that bounded to `accumul_right`.

2. Use `string` values to represent strings and `accumul_right` to write a functional value that computes the concatenation of all the strings in a list.

3. Use the higher-order functions `compose` and `accumul_right` to write the functional value that computes the composition of all the functions in a list. Name this value as `compose_functions`.
Write applications of `compose_functions`.
Write applications of results of applications of `compose_functions`.

4.2.3 Exercise 3 - Operations over functions

1. First consider mathematics.
Let f and g be functions that map real numbers to real numbers. Then $f + g$ denotes the function that maps x to $f(x) + g(x)$ and is called the *sum* of f and g. Similarly fg denotes the function that maps x to $f(x)g(x)$ and is called the *product* of f and g.
Now consider programming.
Use `float` values to represent real numbers. Write functional values that compute the sum and the product of functions that map real numbers to real numbers. Name them `sum` and `product`.
Is `(float -> float) * (float -> float) -> (float -> float)` the type of `sum` and of `product`?
Write applications of `sum` and of `product`.
Write applications of results of applications of each of them.

2. Using `accumul_right` and `product` write the functional value that computes the product of all the elements in a list of function mapping real numbers to real numbers.
Write applications of this functional value.
Write applications of results of such applications.

3. Now climb up in abstraction.
Look at the definitions of `sum` and `product`. Clearly their structures are similar.
Identify what is identical in them and what is not.
Then transform what is identical into a functional value that can be applied to obtain the value of `sum` or that of `product`. Name `phi` this functional value.
Write the type of `phi`.

4. Write applications of `phi` whose results are the values of `sum` and of `product`.
Write another application of `phi`, then an application of the result of this application and finally an application of its result.

4.2.4 Exercise 4 - Filter

1. Write a function named `sublist` that computes from a given predicate the function that extracts the sublist of the elements in a list satisfying the predicate. Its type must be `('a -> bool) -> ('a list -> 'a list)`.

2. Use `sublist` to write a function that computes the sublist of the positive integers in a list.

4.2.5 Exercise 5 - Newton's method

Let a and b two real numbers such that $a < b$. Let $I = [a,\, b]$. Let $f\ :\ \mathbb{R} \longrightarrow \mathbb{R}$ a differentiable function defined and strictly monotonic on the interval I, and such that $f(a)\,f(b) < 0$.

It can easily be shown that f has a unique zero in I (let it be denoted by z).

Let $h\ :\ \mathbb{R} \longrightarrow \mathbb{R}$ defined by $h(x) = x - \frac{f(x)}{f'(x)}$.

Let $(u_n)_{n\in\mathbb{N}}$ the sequence defined by $u_0 = b$ if both f and f' are increasing, or both decreasing, $u_0 = a$ else, and by $u_{n+1} = h(u_n)$ for any natural number n.

It can be shown that if f' is differentiable on I, f'' continuous on I and f'' with no zero in I, then the sequence $(u_n)_{n\in\mathbb{N}}$ converges to z. Then an approximate value for z can be computed by the Newton's method, which consists of computing the successive elements in the sequence $(u_n)_{n\in\mathbb{N}}$ until a value considered closed enough to z is reached.

1. Using the higher-order function `deriv`, write a higher-order function (named `newton`) which, given a function f from real numbers to real numbers and two real numbers a and b, computes by the Newton's method an approximation of the unique zero f has on the closed interval defined by a and b (f will be assumed to satisfy the conditions stated above).

2. Compute an approximation of the square root of 2 by the Newton's method.

Type-inference by yourself

To check your solution enter an Objective Caml toplevel from a directory containing the files `List_1.cmi` and `List_1.cmo` (in other words: the module `List_1`) to be extracted from `this archive`. Then in this toplevel first evaluate this Caml code:

```
#load "List_1.cmo" ;;

open List_1 ;;
```

This loads the module `List_1` then opens it, which allows to use short names instead of long ones in dotted notation (that is, for instance `cons_list` instead of `List_1.cons_list`).

1. Consider each of the following types. Write a value having this types (naming it is useless), evaluate it, then apply it:

`(int -> int) -> int -> int`	that is: `(int -> int) -> (int -> int)`
`(int -> bool) -> int -> bool`	that is: `(int -> bool) -> (int -> bool)`

2. All the following Caml phrases are syntactically correct. They are assumed to be evaluated in order. Consider each of them. Is it a value definition `let` ... $= e$ or an expression e? In each case e has a value. Which one and of which type?

```
let mul = function f -> (function (x, y) -> exp (f (log (x), log (y)))) ;;
let integer = function f -> (function x -> int_of_float (f (float_of_int (x)))) ;;
let comp = function
  (f1, f2, f3) -> (function x -> let (a, b) = f1 (x) in f3 (f2 (a), f2 (b))) ;;
let times = mul (function (a, b) -> a +. b) ;;
times (3., 5.) ;;
let root = integer (sqrt) ;;
root (81) ;;
let c = comp ((function x -> x / 4, x mod 4),
              (function x -> x * 3),
              (function q, r -> 4 * q + r)) ;;
c (9) ;;
let map_to_each = function
  f -> (let rec g = function
          lx ->  if is_empty_list (lx)
                 then empty_list ()
                 else cons_list (f (head (lx)), g (tail (lx)))
        in
          g) ;;
map_to_each (function x -> 2 * x) ;;
(map_to_each (function x -> 2 * x)) ([1 ; 2 ; 3]) ;;
map_to_each (function x -> 3 * x + 1) ;;
(map_to_each (function x -> 3 * x + 1)) ([1 ; 2 ; 3]) ;;
List.map ;;
List.map (function x -> 2 * x) ;;
(List.map (function x -> 2 * x)) ([1 ; 2 ; 3]) ;;
List.map (function x -> 3 * x + 1) ;;
(List.map (function x -> 3 * x + 1)) ([1 ; 2 ; 3]) ;;
let curry = function f -> (function x -> (function y -> f (x, y))) ;;
let uncurry = function f -> (function (x, y) -> (f (x)) (y)) ;;
let plus = function x, y -> x + y ;;
let plus_curry = function x -> (function y -> x + y) ;;
plus (3, 7) ;;
curry (plus) ;;
```

```
(curry (plus)) (3) ;;
((curry (plus)) (3)) (7) ;;
plus_curry (3) ;;
(plus_curry (3)) (7) ;;
uncurry (plus_curry) ;;
(uncurry (plus_curry)) (3, 7) ;;
let exchange = function f -> (function x -> (function y -> (f (y)) (x))) ;;
let rec accum = function
  (op, neutral) -> (let rec
                      f = function
                        lx ->  if is_empty_list (lx)
                                 then neutral
                                 else op (head (lx), f (tail (lx)))
                    in
                      f) ;;
let accum_1 = function
  f -> exchange (function neutral -> accum (uncurry (f), neutral)) ;;
accum_1 (function x -> (function y -> x + y)) ;;
(accum_1 (function x -> (function y -> x + y))) ([1 ; 2 ; 3]) ;;
((accum_1 (function x -> (function y -> x + y))) ([1 ; 2 ; 3])) (7) ;;
((accum_1 (function x -> (function y -> x - y))) ([1 ; 2 ; 3])) (7) ;;
List.fold_right ;;
((List.fold_right (function x -> (function y -> x + y))) ([1 ; 2 ; 3])) (7) ;;
((List.fold_right (function x -> (function y -> x - y))) ([1 ; 2 ; 3])) (7) ;;
let rec fp = function f -> (function x -> if x = f (x) then x else (fp (f)) (f (x))) ;;
fp (function x -> (x +. 2.) /. (x +. 1.)) ;;
(fp (function x -> (x +. 2.) /. (x +. 1.))) (0.) ;;
```

5 Case study - Polynomials in one variable whose coefficients are rational

5.1 Exercises

Given a non-zero polynomial p in one variable X whose coefficients are *real numbers* (i.e., in $\mathbb{R}[X]$), how to compute the number of its *real* roots and how to get an approximation of each of them as accurate as desired?

Sturm's theorem defines, given any open interval i of $\mathbb{R}$ none of the bounds of which is a root of p, the number n_i of *real* roots of p in i. Another theorem gives an open interval out of which p has no *real* root.

Unfortunately if at least one coefficient of p or one bound of i is an *irrational number* (i.e., belongs to $\mathbb{R} \setminus \mathbb{Q}$) then *approximations* when trying to compute n_i can lead to an incorrect result or to no result at all (error or looping).

But *exact* results can be obtained when computing with *rational numbers* (i.e., belongs to $\mathbb{Q}$) using *arbitrary-precision numbers*.

This is why only polynomials whose coefficients are *rational numbers* are here considered.

The aim here is to compute the number of *real* roots of any non-zero polynomial p in $\mathbb{Q}[X]$ and to get for each of them an interval (whose bounds are rational numbers) that contains it and is as smallest as desired.

5.1.1 Exercise 1 - Usual operations over polynomials

Assume an Objective Caml toplevel in which the following modules have been loaded and opened:

- `List_1`: to deal with lists.
- `Rat`: provides the type `rat` to deal with rational numbers.
- `Rat_poly`: provides the type `rat_poly` to deal with polynomials in one variable and whose coefficients are rational numbers.
- `Ex_rat_poly`: provides some `rat_poly` values.

Write definitions of functional values according to the following specifications:

1. Name: `eq_poly`. Type: `rat_poly * rat_poly -> bool`. Tests two polynomials for equality.

2. Name: `add_poly`. Type: `rat_poly * rat_poly -> rat_poly`. Adds two polynomials.

3. Name: `x_times`. Type: `rat_poly -> rat_poly`. Multiply a polynomial by the monomial X.

4. Name: `deriv_poly`. Type: `rat_poly -> rat_poly`. Computes the derivative of a polynomial.

5. Name: `rat_times_poly`. Type: `rat * rat_poly -> rat_poly`. Computes the polynomial ap given the pair (a, p) where a is a rational and p is a polynomial.

6. Name: `mult_poly`. Type: `rat_poly * rat_poly -> rat_poly`. Multiplies two polynomials.

7. Name: `degree`. Type: `rat_poly -> int`. Computes the degree of a non-zero polynomial. Fails if the polynomial is zero.

8. Name: `leading_coef`. Type: `rat_poly -> rat`. Computes the leading coefficient of a non-zero polynomial. Fails if the polynomial is zero.

9. Name: `monomial`. Type: `rat * int -> rat_poly`. Computes the monomial aX^d from a pair (a, d) where a is a non-zero rational and d is a non-negative integer. Fails if $a = 0$ or $d < 0$.

10. Name: `euclid_div_poly`. Type: `rat_poly * rat_poly -> rat_poly * rat_poly`. Computes the pair (q, r) where q is the quotient and r is the remainder in the euclidean division of p_1 by p_2, given a pair (p_1, p_2) of polynomials where the divisor p_2 is non-zero. Fails if p_2 is zero.

11. Name: `quotient_poly`. Type: `rat_poly * rat_poly -> rat_poly`. Computes the quotient in the euclidean division of p_1 by p_2, given a pair (p_1, p_2) of polynomials where the divisor p_2 is non-zero. Fails if p_2 is zero.

12. Name: `mod_poly`. Type: `rat_poly * rat_poly -> rat_poly`. Computes the remainder in the euclidean division of p_1 by p_2, given a pair (p_1, p_2) of polynomials where the divisor p_2 is non-zero. Fails if p_2 is zero.

13. Name: `gcd_poly`. Type: `rat_poly * rat_poly -> rat_poly`. Computes the gcd (greatest common divisor) of a pair of non-zero polynomials. Fails if a component of the pair is zero.

14. Name: `eval_poly`. Type: `rat_poly * rat -> rat`. Evaluation of a polynomial.

15. Name: `function_of_poly`. Type: `rat_poly -> (rat -> rat)`. Computes the polynomial function associated with a polynomial.

16. Name: `poly_of_coef_list`. Type: `rat list -> rat_poly`. Computes the polynomial given a list of its coefficients ordered according to decreasing degrees (it is written *a* list rather than *the* list because this list can begin with useless zeros).

5.1.2 Exercise 2 - The roots of a polynomial

Assume an Objective Caml toplevel in which the following modules have been loaded and opened:

- All the the modules listed here.
- `Rat_inter`: provides the type `rat_inter` to deal with intervals whose bounds are rational numbers (non-empty open intervals and singletons).
- `Rat_poly_common`: provides the functions specified in exercise 5.1.1.

The *irreducible* polynomials in $\mathbb{R}[X]$ are those whose degree is 1 and those whose degree is 2 and whose corresponding quadratic equation has a negative discriminant.

Moreover any non-zero polynomial in $\mathbb{R}[X]$ can be *factorized* as follows and this factorization is *unique* (not taking into account the order of the factors):

$$p \;=\; a_n\, p_1^{h_1} \,\ldots\, p_a^{h_a}\, s_1^{k_1} \,\ldots\, s_b^{k_b} \qquad \text{avec} \qquad h_1 \,+\, \ldots \,+\, h_a \,+\, 2\,(k_1 \,+\, \ldots \,+\, k_b) \;=\; n$$

where:

- n denotes the degree of p.
- a_n denotes its leading coefficient.
- Each p_i (s_j respectively) denotes an irreducible polynomial whose leading coefficients is 1 and whose degrees is 1 (2 respectively). It is called an *irreducible factors* of p.
- $p_i \neq p_{i'}$ for $i \neq i'$ ($s_j \neq s_{j'}$ for $j \neq'$ respectively).
- Each h_i (k_j respectively) denotes a positive integer. It is called the *multiplicity* of p_i (s_j respectively).

A polynomial in $\mathbb{R}[X]$ is said to be *square-free* if the multiplicity of each of its irreducible factors is 1.

The *Sturm chain* (or *Sturm sequence*) of a *square-free* polynomial in $\mathbb{R}[X]$ *whose degree is positive* is the sequence $q_1 \ldots q_r$ of polynomials in $\mathbb{R}[X]$ defined as follows (where p' denotes the derivative of p):

$$\begin{cases} q_1 = p \\ q_2 = p' \\ \forall i \in [2, r-1[\; : \; q_{i+1} = -mod(q_{i-1}, q_i) \\ \text{the degree of } q_r \text{ is } 0 \end{cases}$$

It can be shown that $r \geq 2$, that each q_i is non-zero, and that for all i in $[1, r[$ the degree of q_i is greater than that of q_{i+1}. Clearly if p belongs to $\mathbb{Q}[X]$ then each q_i belongs to $\mathbb{Q}[X]$, too.

For instance, the Sturm chain of $p_1 = X^3 - 2\,X^2 - X + 2$ consists of p_1, followed by $p_1' = 3\,X^2 - 4\,X - 1$, then by $\frac{14}{9}\,X - \frac{16}{9}$ and finally by $\frac{81}{49}$.

That of $p_2 = X^7 - 1$ consists of p_2, followed by $p_2' = 7\,X^6$, and finally by 1.

That of $p_3 = 7\,X + 4$ consists of p_3, followed by $p_3' = 7$.

From now on let $\mathcal{S}$ denote the set of all *square-free* polynomials in $\mathbb{Q}[X]$ *whose degree is positive.*

Let I denote the set of intervals of $\mathbb{R}$ whose bounds are *rational* numbers and that are either non-empty and closed or singletons.

1. Write a definition of the function (named `sturm_chain`) that computes the Sturm chain of a polynomial p in $\mathcal{S}$. Its type must be `rat_poly -> rat_poly list`.

Let w denote the function that maps any *square-free* polynomial p in $\mathbb{R}[X]$ *whose degree is positive* to the function from $\mathbb{R}$ to $\mathbb{N}$ that maps any real number X to the number of sign changes in the list $q_1(X) \ldots q_r(X)$ of real numbers, where $q_1 \ldots q_r$ denotes the Strum chain of p.

2. Write a definition of the function (named `nb_chang`) that computes the number of sign changes in a list of rational numbers. Its type must be `rat list -> int`.

3. Write a definition of the (Objective Caml) function (named `w`) that, given a polynomial p in $\mathcal{S}$, computes the restriction to $\mathbb{Q}$ of the (mathematical) function $w(p)$ (which is from $\mathbb{R}$ to $\mathbb{N}$). Its type must be `rat_poly -> (rat -> int)`.

> Let p denote a non-zero polynomial in $\mathbb{R}[X]$. Let n denote its degree. Let $(a_i)_{i=n\ldots0}$ denote its coefficients in degree decreasing order. Let $M = \frac{1}{|a_n|}\sum_{i=0}^{n}|a_i|$.
>
> Note that $M_p > 0$. Thus $(-M_p, M_p)$ is *non-empty.*
>
> It can be shown that there exists no *real* root of p out of the non-empty open interval $(-M_p, M_p)$ of $\mathbb{R}$.
>
> Moreover note that M_p is a *rational* number if the coefficients of p are *rational* numbers.

4. Write a definition of the function (named `sum_of_coef_abs_val`) that sums the absolute values of all the coefficients of a non-zero polynomial in de $\mathbb{Q}[X]$. Its type must be `rat_poly -> rat`.

5. Write a definition of the function (named `no_root_out`) that, given a non-zero polynomial p in $\mathbb{Q}[X]$, computes the non-empty open interval $(-M_p, M_p)$ of I (its bounds are *rational* numbers and there exists no *real* root of p out of it.). The type of `no_root_out` must be `rat_poly -> rat_inter`.

> It can be shown (Sturm's theorem) that for any *square-free* polynomial p in $\mathbb{R}[X]$ *whose degree is positive* and for any *non-empty open* interval (a, b) of $\mathbb{R}$ such that *neither a nor b is a zero of* p, the number of *real* roots of p belonging to (a, b) is $(w(p))(a) - (w(p))(b)$.
>
> It can also be shown that, given a polynomial p in $\mathbb{R}[X]$ *whose degree is positive*, the polynomial $q_p = div(p, pgcd(p, p'))$ is *square-free* (therefore all its *real* roots are of multiplicity 1) and has exactly the same *real* roots as p. This polynomial q_p will be called the *square-free polynomial associated with p.*

6. Write a definition of the function (named `variation`) that, given a function f from *rational* numbers to *integers*, computes the function that, given an interval i of I whose lower bound a and upper bound b are (*rational* numbers) such that $f(a)$ and $f(b)$ exist, computes the *integer* $f(a) - f(b)$ (note that $a = b$ is possible). Its type must be `(rat -> int) -> (rat_inter -> int)`.

7. Write a definition of the function (named `nb_roots_in`) that, given a polynomial p in $\mathcal{S}$, computes the function that, given a (non-empty) open interval i of I none of the bounds of which is a root of p, computes the number of *real* roots of p in I. Its type must be `rat_poly -> (rat_inter -> int)`.

8. Write a definition of the function (named `square_free_with_the_same_roots`) that computes the square-free polynomial q_p associated with a polynomial p of $\mathbb{Q}[X]$ *whose degree is positive.* Its type must be `rat_poly -> rat_poly`.

9. Write a definition of the function (named `nb_roots`) that computes the number of *real* roots of a non-zero polynomial of $\mathbb{Q}[X]$.. Its type must be `rat_poly -> int`.

6 Controlling complexity of software systems

6.1 Tutorial (draft)