Chapter 4

Basic types

We examine in this chapter the Caml basic types.

4.1 Numbers

Caml Light provides two numeric types: integers (type `int`) and floating-point numbers (type `float`). Integers are limited to the range $-2^{30} \ldots 2^{30} - 1$. Integer arithmetic is taken modulo 2^{31}; that is, an integer operation that overflows does not raise an error, but the result simply wraps around. Predefined operations (functions) on integers include:

`+`	addition
`-`	subtraction and unary minus
`*`	multiplication
`/`	division
`mod`	modulo

Some examples of expressions:

```
#3 * 4 + 2;;
- : int = 14

#3 * (4 + 2);;
- : int = 18

#3 - 7 - 2;;
- : int = -6
```

There are precedence rules that make `*` bind tighter than `+`, and so on. In doubt, write extra parentheses.

So far, we have not seen the type of these arithmetic operations. They all expect two integer arguments; so, they are functions taking one integer and returning a function from integers to integers. The (functional) value of such infix identifiers may be obtained by taking their *prefix* version.

```
#prefix + ;;
- : int -> int -> int = <fun>
```

```
#prefix * ;;
- : int -> int -> int = <fun>

#prefix mod ;;
- : int -> int -> int = <fun>
```

As shown by their types, the infix operators `+`, `*`, ..., do not work on floating-point values. A separate set of floating-point arithmetic operations is provided:

`+.`	addition
`-.`	subtraction and unary minus
`*.`	multiplication
`/.`	division
`sqrt`	square root
`exp`, `log`	exponentiation and logarithm
`sin`, `cos`,...	usual trigonometric operations

We have two conversion functions to go back and forth between integers and floating-point numbers: `int_of_float` and `float_of_int`.

```
#1 + 2.3;;
Toplevel input:
>1 + 2.3;;
>    ^^^
This expression has type float,
but is used with type int.

#float_of_int 1 +. 2.3;;
- : float = 3.3
```

Let us now give some examples of numerical functions. We start by defining some very simple functions on numbers:

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

#square 2.0;;
- : float = 4.0

#square (2.0 /. 3.0);;
- : float = 0.444444444444

#let sum_of_squares (x,y) = square(x) +. square(y);;
sum_of_squares : float * float -> float = <fun>

#let half_pi = 3.14159 /. 2.0
#in sum_of_squares(cos(half_pi), sin(half_pi));;
- : float = 1.0
```

We now develop a classical example: the computation of the root of a function by Newton's method. Newton's method can be stated as follows: if y is an approximation to a root of a function f, then:

$$y - \frac{f(y)}{f'(y)}$$

is a better approximation, where $f'(y)$ is the derivative of f evaluated at y. For example, with $f(x) = x^2 - a$, we have $f'(x) = 2x$, and therefore:

$$y - \frac{f(y)}{f'(y)} = y - \frac{y^2 - a}{2y} = \frac{y + \frac{a}{y}}{2}$$

We can define a function `deriv` for approximating the derivative of a function at a given point by:

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

Provided `dx` is sufficiently small, this gives a reasonable estimate of the derivative of f at x.

The following function returns the absolute value of its floating point number argument. It uses the "`if` ... `then` ... `else`" conditional construct.

```
#let abs x = if x >. 0.0 then x else -. x;;
abs : float -> float = <fun>
```

The main function, given below, uses three local functions. The first one, `until`, is an example of a *recursive* function: it calls itself in its body, and is defined using a `let rec` construct (`rec` shows that the definition is recursive). It takes three arguments: a predicate `p` on floats, a function `change` from floats to floats, and a float `x`. If `p(x)` is false, then `until` is called with last argument `change(x)`, otherwise, `x` is the result of the whole call. We will study recursive functions more closely later. The two other auxiliary functions, `satisfied` and `improve`, take a float as argument and deliver respectively a boolean value and a float. The function `satisfied` asks wether the image of its argument by `f` is smaller than `epsilon` or not, thus testing wether `y` is close enough to a root of `f`. The function `improve` computes the next approximation using the formula given below. The three local functions are defined using a cascade of `let` constructs. The whole function is:

```
#let newton f epsilon =
#  let rec until p change x =
#            if p(x) then x
#            else until p change (change(x)) in
#  let satisfied y = abs(f y) <. epsilon in
#  let improve y = y -. (f(y) /. (deriv f y epsilon))
#in until satisfied improve;;
newton : (float -> float) -> float -> float -> float = <fun>
```

Some examples of equation solving:

```
#let square_root x epsilon =
#           newton (function y -> y*.y -. x) epsilon x
```

```
#and cubic_root x epsilon =
#           newton (function y -> y*.y*.y -. x) epsilon x;;
square_root : float -> float -> float = <fun>
cubic_root : float -> float -> float = <fun>

#square_root 2.0 1e-5;;
- : float = 1.41421569553

#cubic_root 8.0 1e-5;;
- : float = 2.00000000131

#2.0 *. (newton cos 1e-5 1.5);;
- : float = 3.14159265359
```

4.2 Boolean values

The presence of the conditional construct implies the presence of boolean values. The type `bool` is composed of two values `true` and `false`.

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

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

The functions with results of type `bool` are often called *predicates*. Many predicates are predefined in Caml. Here are some of them:

```
#prefix <;;
- : 'a -> 'a -> bool = <fun>

#1 < 2;;
- : bool = true

#prefix <.;;
- : float -> float -> bool = <fun>

#3.14159 <. 2.718;;
- : bool = false
```

There exist also `<=`, `>`, `>=`, and similarly `<=.`, `>.`, `>=.`.

4.2.1 Equality

Equality has a special status in functional languages because of functional values. It is easy to test equality of values of base types (integers, floating-point numbers, booleans, ...):

```
#1 = 2;;
- : bool = false

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

But it is impossible, in the general case, to decide the equality of functional values. Hence, equality stops on a run-time error when it encounters functional values.

```
#(fun x -> x) = (fun x -> x);;
Uncaught exception: Invalid_argument "equal: functional value"
```

Since equality may be used on values of any type, what is its type? Equality takes two arguments of the same type (whatever type it is) and returns a boolean value. The type of equality is a *polymorphic type*, i.e. may take several possible forms. Indeed, when testing equality of two numbers, then its type is `int -> int -> bool`, and when testing equality of strings, its type is `string -> string -> bool`.

```
#prefix =;;
- : 'a -> 'a -> bool = <fun>
```

The type of equality uses *type variables*, written `'a`, `'b`, etc. Any type can be substituted to type variables in order to produces specific *instances* of types. For example, substituting `int` for `'a` above produces `int -> int -> bool`, which is the type of the equality function used on integer values.

```
#1=2;;
- : bool = false

#(1,2) = (2,1);;
- : bool = false

#1 = (1,2);;
Toplevel input:
>1 = (1,2);;
>     ^^^
This expression has type int * int,
but is used with type int.
```

4.2.2 Conditional

Conditional expressions are of the form "`if` e_{test} `then` e_1 `else` e_2", where e_{test} is an expression of type `bool` and e_1, e_2 are expressions possessing the same type. As an example, the logical negation could be written:

```
#let negate a = if a then false else true;;
negate : bool -> bool = <fun>

#negate (1=2);;
- : bool = true
```

4.2.3 Logical operators

The classical logical operators are available in Caml. Disjunction and conjunction are respectively written `or` and `&`:

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

```
#(1<2) & (2>1);;
- : bool = true
```

The operators `&` and `or` are not functions. They should not be seen as regular functions, since they evaluate their second argument only if it is needed. For example, the `or` operator evaluates its second operand only if the first one evaluates to `false`. The behavior of these operators may be described as follows:

e_1 `or` e_2	is equivalent to	`if` e_1 `then true else` e_2
e_1 `&` e_2	is equivalent to	`if` e_1 `then` e_2 `else false`

Negation is predefined:

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

The `not` identifier receives a special treatment from the parser: the application "`not f x`" is recognized as "`not (f x)`" while "`f g x`" is identical to "`(f g) x`". This special treatment explains why the functional value associated to `not` can be obtained only using the `prefix` keyword:

```
#prefix not;;
- : bool -> bool = <fun>
```

4.3 Strings and characters

String constants (type `string`) are written between `"` characters (double-quotes). Single-character constants (type `char`) are written between `‘` characters (backquotes). The most used string operation is string concatenation, denoted by the `^` character.

```
#"Hello" ^ " World!";;
- : string = "Hello World!"
```

```
#prefix ^;;
- : string -> string -> string = <fun>
```

Characters are ASCII characters:

```
#‘a‘;;
- : char = ‘a‘
```

```
#‘\065‘;;
- : char = ‘A‘
```

and can be built from their ASCII code as in:

```
#char_of_int 65;;
- : char = 'A'
```

Other operations are available (`sub_string`, `int_of_char`, etc). See the Caml Light reference manual [20] for complete information.

4.4 Tuples

4.4.1 Constructing tuples

It is possible to combine values into tuples (pairs, triples, ...). The *value constructor* for tuples is the "," character (the comma). We will often use parentheses around tuples in order to improve readability, but they are not strictly necessary.

```
#1,2;;
- : int * int = 1, 2

#1,2,3,4;;
- : int * int * int * int = 1, 2, 3, 4

#let p = (1+2, 1<2);;
p : int * bool = 3, true
```

The infix "*" identifier is the *type constructor* for tuples. For instance, $t_1 * t_2$ corresponds to the mathematical cartesian product of types t_1 and t_2.

We can build tuples from any values: the tuple value constructor is *generic*.

4.4.2 Extracting pair components

Projection functions are used to extract components of tuples. For pairs, we have:

```
#fst;;
- : 'a * 'b -> 'a = <fun>

#snd;;
- : 'a * 'b -> 'b = <fun>
```

They have polymorphic types, of course, since they may be applied to any kind of pair. They are predefined in the Caml system, but could be defined by the user (in fact, the cartesian product itself could be defined — see section 6.1, dedicated to user-defined product types):

```
#let first (x,y) = x
#and second (x,y) = y;;
first : 'a * 'b -> 'a = <fun>
second : 'a * 'b -> 'b = <fun>

#first p;;
- : int = 3

#second p;;
- : bool = true
```

We say that `first` is a *generic* value because it works uniformly on several kinds of arguments (provided they are pairs). We often confuse between "generic" and "polymorphic", saying that such value is polymorphic instead of generic.

4.5 Patterns and pattern-matching

Patterns and pattern-matching play an important role in ML languages. They appear in all real programs and are strongly related to types (predefined or user-defined).

A *pattern* indicates the *shape* of a value. Patterns are "values with holes". A single variable (formal parameter) is a pattern (with no shape specified: it matches any value). When a value is *matched against* a pattern (this is called *pattern-matching*), the pattern acts as a filter. We already used patterns and pattern-matching in all the functions we wrote: the function body `(function x -> ...)` does (trivial) pattern-matching. When applied to a value, the formal parameter `x` gets bound to that value. The function body `(function (x,y) -> x+y)` does also pattern-matching: when applied to a value (a pair, because of well-typing hypotheses), the `x` and `y` get bound respectively to the first and the second component of that pair.

All these pattern-matching examples were trivial ones, they did not involve any test:

- formal parameters such as `x` do not impose any particular shape to the value they are supposed to match;
- pair patterns such as `(x,y)` always match for typing reasons (cartesian product is a *product type*).

However, some types do not guarantee such a uniform shape to their values. Consider the `bool` type: an element of type `bool` is either `true` or `false`. If we impose to a value of type `bool` to have the shape of `true`, then pattern-matching may fail. Consider the following function:

```
#let f = function true -> false;;
Toplevel input:
>let f = function true -> false;;
>        ^^^^^^^^^^^^^^^^^^^^^^
Warning: this matching is not exhaustive.
f : bool -> bool = <fun>
```

The compiler warns us that the pattern-matching may fail (we did not consider the `false` case).

Thus, if we apply `f` to a value that evaluates to `true`, pattern-matching will succeed (an equality test is performed during execution).

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

But, if `f`'s argument evaluates to `false`, a run-time error is reported:

```
#f (1>2);;
Uncaught exception: Match_failure ("", 1346, 1368)
```

Here is a correct definition using pattern-matching:

```
#let negate = function true -> false
#                     | false -> true;;
negate : bool -> bool = <fun>
```

The pattern-matching has now two cases, separated by the "|" character. Cases are examined in turn, from top to bottom. An equivalent definition of `negate` would be:

```
#let negate = function true -> false
#                     | x -> true;;
negate : bool -> bool = <fun>
```

The second case now matches any boolean value (in fact, only `false` since `true` has been caught by the first match case). When the second case is chosen, the identifier `x` gets bound to the argument of `negate`, and could be used in the right-hand part of the match case. Since in our example we do not use the value of the argument in the right-hand part of the second match case, another equivalent definition of `negate` would be:

```
#let negate = function true -> false
#                     | _ -> true;;
negate : bool -> bool = <fun>
```

Where "_" acts as a formal paramenter (matches any value), but does not introduce any binding: it should be read as "anything else".

As an example of pattern-matching, consider the following function definition (truth value table of implication):

```
#let imply = function (true,true) -> true
#                   | (true,false) -> false
#                   | (false,true) -> true
#                   | (false,false) -> true;;
imply : bool * bool -> bool = <fun>
```

Here is another way of defining `imply`, by using variables:

```
#let imply = function (true,x) -> x
#                   | (false,x) -> true;;
imply : bool * bool -> bool = <fun>
```

Simpler, and simpler again:

```
#let imply = function (true,x) -> x
#                   | (false,_) -> true;;
imply : bool * bool -> bool = <fun>
```

```
#let imply = function (true,false) -> false
#                   | _ -> true;;
imply : bool * bool -> bool = <fun>
```

Pattern-matching is allowed on any type of value (non-trivial pattern-matching is only possible on types with *data constructors*).

For example:

```
#let is_zero = function 0 -> true | _ -> false;;
is_zero : int -> bool = <fun>

#let is_yes = function "oui" -> true
#                    | "si" -> true
#                    | "ya" -> true
#                    | "yes" -> true
#                    | _ -> false;;
is_yes : string -> bool = <fun>
```

4.6 Functions

The type constructor "`->`" is predefined and cannot be defined in ML's type system. We shall explore in this section some further aspects of functions and functional types.

4.6.1 Functional composition

Functional composition is easily definable in Caml. It is of course a polymorphic function:

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

The type of `compose` contains no more constraints than the ones appearing in the definition: in a sense, it is the *most general* type compatible with these constraints.

These constraints are:

- the codomain of `g` and the domain of `f` must be the same;
- `x` must belong to the domain of `g`;
- `compose f g x` will belong to the codomain of `f`.

Let's see `compose` at work:

```
#let succ x = x+1;;
succ : int -> int = <fun>

#compose succ list_length;;
- : '_a list -> int = <fun>

#(compose succ list_length) [1;2;3];;
- : int = 4
```

4.6.2 Currying

We can define two versions of the addition function:

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

#let add = function x -> (function y -> x+y);;
add : int -> int -> int = <fun>
```

These two functions differ only in their way of taking their arguments. The first one will take an argument belonging to a cartesian product, the second one will take a number and return a function. The `add` function is said to be *the curried version* of `plus` (in honor of the logician Haskell Curry).

The currying transformation may be written in Caml under the form of a higher-order function:

```
#let curry f = function x -> (function y -> f(x,y));;
curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
```

Its inverse function may be defined by:

```
#let uncurry f = function (x,y) -> f x y;;
uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c = <fun>
```

We may check the types of `curry` and `uncurry` on particular examples:

```
#uncurry (prefix +);;
- : int * int -> int = <fun>

#uncurry (prefix ^);;
- : string * string -> string = <fun>

#curry plus;;
- : int -> int -> int = <fun>
```

Exercises

Exercise 4.1 *Define functions that compute the surface area and the volume of well-known geometric objects (rectangles, circles, spheres, ...).*

Exercise 4.2 *What would happen in a language submitted to call-by-value with recursion if there was no conditional construct (*`if ... then ... else ...`*)?*

Exercise 4.3 *Define the* `factorial` *function such that:*

$$\texttt{factorial}\ n = n * (n-1) * \ldots * 2 * 1$$

Give two versions of `factorial`*: recursive and tail recursive.*

Exercise 4.4 *Define the* `fibonacci` *function that, when given a number n, returns the nth Fibonacci number, i.e. the nth term u_n of the sequence defined by:*

$$\begin{aligned} u_1 &= 1 \\ u_2 &= 1 \\ u_{n+2} &= u_{n+1} + u_n \end{aligned}$$

Exercise 4.5 *What are the types of the following expressions?*

- `uncurry compose`
- `compose curry uncurry`
- `compose uncurry curry`