Patrick COUSOT

Professeur d'informatique
Computer science professor

Notes de cours d'Interprétation Abstraite
Abstract Interpretation course notes

Handout reading recommendation: [•]: compulsory, [•]: facultative, [•]: optional complement.
Handout revision after the course presentation: [!]: modifications and corrections, [+]: insertion of additional material.

---

11 An overview of the programming language O'Caml:

- **Introduction:**
  - Origins of the O'Caml programmin language [•]
  - O'Caml on the web [•]
  - (Dis-)advantage of a functional programming language [•]
  - Launching O'Caml toplevel under Unix [•]
  - O'Caml is a *functional* language [•]
  - O'Caml is also an *imperative* language [•]
  - O'Caml has a vast standard library [•]
  - O'Caml is interfaced with other languages like C [•]
  - O'Caml is interfaced with operating systems [•]
  - O'Caml has good (classical) programming tools [•]
  - O'Caml is free (at least for academics) [•]
- **Basic data types in O'Caml :** [•]
  - The `unit` type in O'Caml [•]
  - Booleans (`bool`) in O'Caml [•]
  - Infixed and prefixed binary operators in O'Caml (`int`) operations and functions in O'Caml [•]

- Integers in O'Caml [•]
- Built-in integer (`int`) operations and functions in O'Caml [•]
- `Int32` and `Int64` predefined modules [•]
- Integers and floating point numbers in O'Caml O'Caml [•]
- Built-in floating-point (`float`) operations and functions in O'Caml [•]
- Characters and strings in O'Caml [•]
- Using standard modules of O'Caml libraries [•]
- Overloaded/polymorphic comparison (of basic types) in O'Caml [•]
- Input-output of basic values [•]

- **Defining values and functions in O'Caml:**
  - Defining values and functions at the top level [•]
  - Local binding of values and functions in expressions [•]
  - Defining recursive functions [•]
  - Scope of local variables in `let` [`rec`] definitions [•]
  - Non termination of recursive functions [•]
  - Defining functions with multiple arguments [•]
  - Forms of value and function definition in O'Caml [•]
  - Simultaneous definition of [recursive] functions in O'Caml [•]
  - Polymorphic functions in O'Caml [•]
  - Examples of higher-order polymorphic functionals in O'Caml [•]
  - Various forms of function definition and call in O'Caml [•]
  - Anonymous function definition and call in O'Caml [•]
- **Matching in O'Caml :**
  - Pattern-matching/filtering in O'Caml [•]
  - Possible patterns [•]
  - Definition of functions by matching [•]
- **Imperative features in O'Caml : variables, control structures and input/output**
  - References and variables in O'Caml [•]
  - Block structure O'Caml expressions [•]
  - Exceptions in O'Caml [•]
  - Sequences in O'Caml [•]
  - Conditionals in O'Caml [•]
  - Loops in O'Caml [•]
  - File input/output in O'Caml [•]

- Formatted output in O'Caml [•]

- **Structured data types in O'Caml:** [•]
  - Pairs, [un]currying in O'Caml [•]
  - Arrays/vectors in O'Caml [•]
  - The `Array` module in O'Caml [•]
  - Lists in O'Caml [•]
  - The `List` module in O'Caml [•]
  - Variant types definition in O'Caml [•]
  - Record types in O'Caml [•]
  - [Polymorphic] recursive types in O'Caml [•]

Examples of simple O'Caml programs and theircorrectness proof:

  - Examples of list manipulation in O'Caml [•]
  - List reduction in O'Caml , with an inductive correctness proof [•]
  - Partition, QuickSort, proof sketch [•]

- **Type inference in ML [•]:**
  - The basic ML typing rules for constants, examples, type environments [•]
  - Types with variables [•]
  - Instantiation rule [•]
  - Typing conditionals, examples [•]
  - Typing (anonymous) functions and applications (calls) [•]
  - Example of function and application typing [•]
  - Typing the `let` construct, example [•]
  - Type polymorphism, polytypes, examples [•]
  - Generalization of monotypes to polytypes [•]
  - Example of correct generalization [•]
  - Example of forbidden generalization [•]
  - Example of polymorphic type verification [•]
  - Recursion is not polymorphic in ML, counter-examples [•]
  - Monomorphic typing rule for the `let rec`, example [•]
  - A note on the typing environment in typing rules [•]
  - An exponential typing examples [•]

- **Modularity and separate compilation in OCamel** [•]
  - Modules [•]
  - Signatures [•]
  - Functors [•]
  - Example modular program [•]
  - Separate compilation [•]
  - Equivalence between separately compiled modules and a single O'Caml pogram [•]
  - Example makefile and script [•]
  - (Directory containing the programs) [•]

Patrick COUSOT
Département d'Informatique (DI)
École Normale Supérieure (ENS)
45 rue d'Ulm
75230 Paris Cedex 05
France
***Métro**: Luxembourg*
***Bureau / Office**: S9, étage/floor -1 (Plan/Map)*
***Email:** cousot@ens.fr or Patrick.Cousot@ens.fr or cousotp@acm.org*
***Tél. direct :** 01 44 32 20 64 / 34 / **Direct tlph :** +33 1 44 32 20 64 / 34*
***Fax:** +33 (0)1 44 32 21 52*
***URL:** http://www.di.ens.fr/~cousot*
PGP public key

The origins of the OCAML programming language

- ML is the higher-order functional programming Language created by Robin Milner in Edinburg (Scotland) in 1976/77
  - Functional : functions are first class objects (can be passed as parameters, returned, composed, ...)
  - Higher-order : functions can take functions which take functions which take functions ... as parameters
  - The main innovation was the Milner/Damas type inference algorithm (type of objects and operations is inferred from their usage, a new idea at the time).
- starting 1983, two main developments
  - SML (Standard ML) : a standardization of ML proposed by the Bell Labs
  - CAML [Heavy-CAML] : an implementation of ML (based on the Categorical Abstract Machine, now abandonned in favor of Krivine machine). Developped by the CRISTAL project of INRIA.

- Many imperative features (pointers, assignments, ...)
- Very nice interface to LEX and YACC directly available in the language
- Complex language
- Lacks modules

- CAML-light : a new implementation of a subset of the language, with many simplifications, compilation to a virtual machine (Zinc), still lacking modules (because of typing difficulties)

- OCAML : the current implementation, with
  - separate compilation of modules;
  - object-orientation;
  - compilation to native code;

- Alternative are purely functional, such as :
  - Haskell : lazy (an operation, such as passing a parameter, is evaluated only when needed)
    see http://www.haskell.org

OCAML on the web.

- Introductions, reference manuals, books, examples, tools, etc... all freely available at URL:

  http://www.ocaml.org

- Compilers are freely available:

  http://caml.inria.fr/ocaml/distrib.html

  - Source distribution (not recommended)
  - Binary distributions:
    - Linux: Debian, Mandrake, RedHat
    - MS Windows XP, 2000, NT, ME, 98, 95
    - Mac OS X

- The user manual is available:

  - in PDF:

    http://caml.inria.fr/distrib/ocaml-3.06/ocaml-3.06-refman.pdf

  - In PostScript: ~/ocaml-3.06-refman.ps.gz
  - In a bundle of HTML files:

    ~/ocaml-3.06-refman.html.tar.gz

    which can be viewed online:

    http://caml.inria.fr/ocaml/htmlman/

(Dis-) advantages of a functional programming language

For :

- Brevity (more compact / concise programs)
- Ease of understanding (no implementation details, no side effects in purely functional languages)
- Strongly typed (many errors can be found at compile time, thus avoiding all core dumps, segmentation faults, etc.. so common e.g. in C)
- Code reuse (polymorphism allows for code to be written that works for many sorts of data structures, as opposed e.g. to C where the code must be duplicated for each possible type of data)
- built-in memory management (heap-allocation with garbage-collector)
- powerful abstraction : the expression of algorithms is close to the language of mathematics

Against :

- slow, not usable for performance-demanding applications
- can use much more memory (copy instead of in-place modification)
- not real-time (garbage collection is unpredictable)
- complex (subtleties in typing, modules, etc...).

Launching OCAML toplevel under Unix

- OCAML can be :
  - interpreted ;
  - compiled :
    - in an intermediate language later interpreted (like Java)
    - in an object language for most machines.
- The toplevel is a shell for interactive use of the OCAML interpreter ;
- Launching & quiting the top level :

```
% ocaml
        Objective Caml version 3.06

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

- At the prompt (#), one types an expression, followed by a mandatory "**;;**" and a return ⏎.

  The interpreter prints the type of the expression and its value (if printable)

- To quit, type CTRL + D (typed here ^D).

```
% ocaml
        Objective Caml version 3.06

# exit 0;;
% ocaml
        Objective Caml version 3.06

# let fin () = print_string "fin\n";;
val fin : unit -> unit = <fun>
# at_exit fin;;
- : unit = ()
# exit 1;;
fin
%
```

Program termination

```
val exit : int -> 'a

val at_exit: (unit -> unit) -> unit
```

Terminate the process, returning the given status code to the operating system
Register the given function to be called at program termination time. The functions registered with `at_exit` will be called when the program executes `exit`. They will not be called if the program terminates because of an uncaught exception. The functions are called in "last in, first out" order.

OCAML is a functional language

One can write functions (so called functionals) that take functions as argument and return a function.

A typical example of functional is the mathematical composition $\circ$ of functions :

$$(f \circ g)(x) \triangleq f(g(x))$$

which can be defined in OCAML as follows :

```
% ocaml
        Objective Caml version 3.06

# let incr x = x + 1;;
val incr : int -> int = <fun>
# let o f g x = (f (g (x)));;
val o : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
# let incr2 = o incr incr;;
val incr2 : int -> int = <fun>
# incr2 7;;
- : int = 9
# ^D
%
```

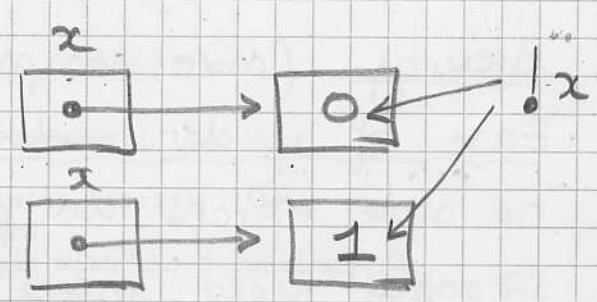
Note 1 : the value of function is some executable code which is not printable (but is stored in memory and is executable by the OCAML interpretor).

Note 2 : The infix mathematical notation $f \circ g$ is written in prefix-form in OCAML that is `o f g`, so $(f \circ g)(x)$ is `o f g x`.

OCAML is also an imperative language

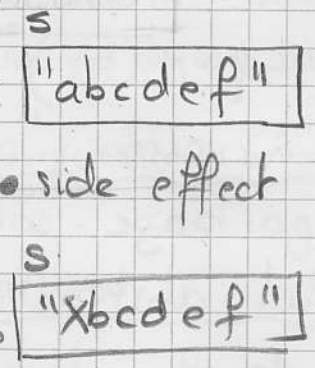
Assignment to modifiable variables :

```
% ocaml
        Objective Caml version 3.06

# let x = ref 0;;
val x : int ref = {contents = 0}
# !x;;
- : int = 0
# x := 1;;
- : unit = ()
# !x;;
- : int = 1
# ^D
%
```



Side effects

```
% ocaml
        Objective Caml version 3.06

# let s = "abcdef";;
val s : string = "abcdef"
# s;;
- : string = "abcdef"
# set s 0 'X';;
Unbound value set
# String.set s 0 'X';;
- : unit = ()
# s;;
- : string = "Xbcdef"
# ^D
%
```



OCAML has a vast standard library.

- Many data structures and algorithms are already implemented.

```
· The standard library:
Module Arg : parsing of command line arguments
Module Array : array operations
Module ArrayLabels : array operations (with labels)
Module Buffer : extensible string buffers
Module Callback : registering Caml values with the C runtime
Module Char : character operations
Module Complex : Complex numbers
Module Digest : MD5 message digest
Module Filename : operations on file names
Module Format : pretty printing
Module Gc : memory management control and statistics; finalised values
Module Genlex : a generic lexical analyzer
Module Hashtbl : hash tables and hash functions
Module Int32 : 32-bit integers
Module Int64 : 64-bit integers
Module Lazy : deferred computations.
Module Lexing : the run-time library for lexers generated by ocamllex
Module List : list operations
Module ListLabels : list operations (with labels)
Module Map : association tables over ordered types
Module Marshal : marshaling of data structures
Module MoreLabels : Include modules Hashtbl ,Map and Set with labels
Module Nativeint : processor-native integers
Module Oo : object-oriented extension
Module Parsing : the run-time library for parsers generated by ocamlyacc
Module Printexc : facilities for printing exceptions
Module Printf : formatting printing functions
Module Queue : first-in first-out queues
Module Random : pseudo-random number generator (PRNG)
Module Scanf : formatted input functions
Module Set : sets over ordered types
Module Sort : sorting and merging lists
Module Stack : last-in first-out stacks
Module StdLabels : Include modules Array ,List and String with labels
Module Stream : streams and parsers
Module String : string operations
Module StringLabels : string operations (with labels)
Module Sys : system interface
Module Weak : arrays of weak pointers
· The num library: arbitrary-precision rational arithmetic :
Module Num : operation on arbitrary-precision numbers
Module Big_int : operations on arbitrary-precision integers
Module Arith_status : flags that control rational arithmeti
· The str library: regular expressions and string processing :
Module Str : regular expressions and string processi
· The bigarray library
Module Bigarray : large, multi-dimensional, numerical arrays
```

The code of these libraries is publicly available.

OCAML is interfaced with other programming languages (like C).

See

http://caml.inria.fr/ocaml/htmlman/manual032.html

- Possibility to call C functions in OCAML ;
- Possibility to call OCAML functions in C ;
- Include files are provided to write C code that operates on OCaml values ;
- C code can be statically linked with OCAML code
- C code can be dynamically linked with OCAML code
- It is possible to interact with the OCAML garbage collector at low level in C

OCAML is interfaced with operating systems

· The unix library: Unix system calls
Module Unix : Unix system calls
Module UnixLabels : Labeled Unix system calls
· The threads library :
Module Thread : lightweight threads
Module Mutex : locks for mutual exclusion
Module Condition : condition variables to synchronize between threads
Module Event : first-class synchronous communication
Module ThreadUnix : thread-compatible s
· The graphics library
Module Graphics : machine-independent graphics primitive
· The dbm library: access to NDBM databases
Module Dbm : interface to the NDBM database
· The dynlink library: dynamic loading and linking of object files
Module Dynlink : dynamic loading of bytecode object files
· The LablTk library: Tcl/Tk GUI interface
The Tk library: Basic functions and types for Labl

OCAML has good (classical) programming tools

- Batch compilation (ocamlc)
- The toplevel system (ocaml)
- The runtime system (ocamlrun)
- Native-code compilation (ocamlopt)
- Lexer and parser generators (ocamllex, ocamlyacc)
- Dependency generator (ocamldep)
- The browser/editor (ocamlbrowser)
- The documentation generator (ocamldoc)
- The debugger (ocamldebug)
- Profiling (ocamlprof)

OCAML is free (at least for academics)

The licence is available at

http://caml.inria.fr/license/

There are no restrictions on code produced with the OCAML compiler that includes OCAML libraries:

The Caml License

Objective Caml

In the following, "the Library" refers to all files marked "Copyright INRIA" in the following directories and their sub-directories:

```
asmrun, byterun, config, maccaml, otherlibs, stdlib, win32caml
```

and "the Compiler" refers to all files marked "Copyright INRIA" in the other directories and their sub-directories.

The Compiler is distributed under the terms of the Q Public License version 1.0

The Library is distributed under the terms of the GNU Library General Public License version 2.

As a special exception to the GNU Library General Public License, you may link, statically or dynamically, a "work that uses the Library" with a publicly distributed version of the Library to produce an executable file containing portions of the Library, and distribute that executable file under terms of your choice, without any of the additional requirements listed in clause 6 of the GNU Library General Public License. By "a publicly distributed version of the Library", we mean either the unmodified Library as distributed by INRIA, or a modified version of the Library that is distributed under the conditions defined in clause 3 of the GNU Library General Public License. This exception does not however invalidate any other reasons why the executable file might be covered by the GNU Library General Public License.

Basic data types in OCAML

Predefined Types

The core library of OCaml provides the following predefined types:

Type	Meaning	Remark
int	Integer numbers	-2^{30} to $2^{30}-1$ or -2^{62} to $2^{62}-1$
char	Characters	ASCII
string	Character strings	Up to $2^{24}-6$ characters
float	Floating-point numbers	IEEE 754 (53-bit mantissa, –1022 to 1023 exponent)
bool	Booleans	`True, false`
unit	Unit values	
array	Arrays	Up to $2^{22}-1$ elements
list	Lists	
exn	Exceptions	
option	Optional parameter	
format	Format strings	C's printf-style format string

} basic types.

The Unit type

- The type unit has a unique value denoted ():

```
# ();;
- : unit = ()
```

- All functions in OCAML must take a parameter, so a parameterless function or procedure can use a () parameter

```
f ();;
```

- In OCAML all expressions must return a value, including procedure calls, which by convention return (), the unique

- The only Unit operation is:

```
# ignore;;
- : 'a -> unit = <fun>
```

which discards its argument (which can be of any type) and returns ().

```
# ignore 12345;;
- : unit = ()
```

- The are no other operators on the unit value () which can only be passed as parameters, returned (and assigned to variables of type unit):

```
# let id x = x;;
val id : 'a -> 'a = <fun>
# id ();;
- : unit = ()
```

Booleans

```
# true;;
- : bool = true
# false;;
- : bool = false
# not true;;
- : bool = false
# not false;;
- : bool = true
# true && true;;
- : bool = true
# true & false;;
- : bool = false
# false or false;;
- : bool = false
# false || true;;
- : bool = true
```

← constants

← negation

← conjunction (written && or &)

← disjunction (written or or ||)

Binary infix operators (written a op b) can be used in prefix form (written (op) a b) :

```
# (&&);;
- : bool -> bool -> bool = <fun>
# (&&) true false;;
- : bool = false
# (or) true false;;
- : bool = true
```

Boolean Operations

`val not : bool -> bool`	Boolean negation
`val (&&) : bool -> bool -> bool`	Boolean AND
`val (&) : bool -> bool -> bool`	Boolean AND
`val (or) : bool -> bool -> bool`	Boolean OR
`val (||) : bool -> bool -> bool`	Boolean AND

The evaluation of boolean operators is <u>lazy</u> (arguments are not evaluated when not necessary for providing the result). For example:

```
# failwith "fail";;
Exception: Failure "fail".
# true or (failwith "fail");;
- : bool = true
# false && (failwith "fail");;
- : bool = false
# true && (failwith "fail");;
Exception: Failure "fail".
```

[(failwith "fail") always produces a run-time error, stops the execution and prints "fail"].

Infixed and prefixed operators in OCAML

If op is a binary infixed operator (write a op b) then (op) is the same operator in prefix form (written (op) a b).

If the operator op is lazy (parameters which do not influence the result need not be evaluated), then so is the prefix operator (op).

```
# false && true;;
- : bool = false
# (&&) false true;;
- : bool = false
# true && (failwith "fail");;
Exception: Failure "fail".
# false && (failwith "fail");;
- : bool = false
# (&&) false (failwith "fail");;
- : bool = false
#
```

Integers in OCAML

- So called "native" integers.
- Integers belong to the interval min_int = $-2^{30} = -1073741824$ and max_int = $2^{30} - 1 = 1073741823$. Machine dependent.

```
% ocaml
        Objective Caml version 3.06

# 99 * 99;;
- : int = 9801
# max_int;;
- : int = 1073741823
#  (2.0 ** 30.0) -. 1.0;;
- : float = 1073741823.
# min_int;;
- : int = -1073741824
# -.(2.0 ** 30.0);;
- : float = -1073741824.
#
```

Arithmetic is modular in OCAML

All integer arithmetic operations are done modulo 2^{31}, which yields may be surprising results :

```
# max_int;;
- : int = 1073741823
# max_int + 1;;
- : int = -1073741824
# min_int;;
- : int = -1073741824
# min_int - 1;;
- : int = 1073741823
# 99999999999999 * 99999999999999999;;
- : int = 301711361
#
```

The num library implements integer arithmetic and rational arithmetic, in arbitrary precision, see

http://caml.inria.fr/ocaml/htmlman/manual036.html

Built-in integer (int) operations and functions

```
# 4 + 5;;
- : int = 9
# 4 - 5;;
- : int = -1
# 4 * 5;;
- : int = 20
# 4 / 5;;
- : int = 0
# 5 / 4;;
- : int = 1
# 5 mod 4;;
- : int = 1
```

Integer Arithematic

`val (~-) : int -> int`	Unary negation
`val succ : int -> int`	`succ x` is `x+1` (Successor)
`val pred : int -> int`	`succ x` is `x-1` (Predecessor)
`val (+) : int -> int -> int`	Integer addition
`val (-) : int -> int -> int`	Integer subtraction
`val (*) : int -> int -> int`	Integer multiplication
`val (/) : int -> int -> int`	Integer division
`val mod : int -> int -> int`	Integer modulo
`val abs : int -> int`	Integer absolute
`val max_int : int`	The greatest representable integer
`val min_int : int`	The smallest representable integer

```
# 0 lsr 1;;
- : int = 0
# 5 land 4;;
- : int = 4
# 5 lor 3;;
- : int = 7
# 5 lor 4;;
- : int = 5
# 5 lsl 2;;
- : int = 20
# 5 lsr 2;;
- : int = 1
```

Bitwise Operations

`val (land) : int -> int -> int`	Bitwise logical AND
`val (lor) : int -> int -> int`	Bitwise logical OR
`val (lxor) : int -> int -> int`	Bitwise logical XOR
`val lnot : int -> int`	Bitwise logical negation
`val (lsl) : int -> int -> int`	Bitwise logical shift to the left
`val (lsr) : int -> int -> int`	Bitwise logical shift to the right
`val (asr) : int -> int -> int`	Bitwise arithmetic shift to the right

Int32 and Int64 predefined modules

- The size of the int integers may vary from one implementation to another (16, 32, 64 bits);
- The modules Int32 and Int64 implement 32-bit / 64-bit integer whichever is the machine.

`val zero : int32`

The 32-bit integer 0.

`val one : int32`

The 32-bit integer 1.

`val minus_one : int32`

The 32-bit integer -1.

`val neg : int32 -> int32`

Unary negation.

`val add : int32 -> int32 -> int32`

Addition.

`val sub : int32 -> int32 -> int32`

Subtraction.

`val mul : int32 -> int32 -> int32`

Multiplication.

`val div : int32 -> int32 -> int32`

Integer division. Raise `Division_by_zero` if the second argument is zero.

```
val rem : int32 -> int32 -> int32
```

Integer remainder. If `x >= 0` and `y > 0`, the result of `Int32.rem x y` satisfies the following properties: `0 <= Int32.rem x y < y` and `x = Int32.add (Int32.mul (Int32.div x y) y) (Int32.rem x y)`. If `y = 0`, `Int32.rem x y` raises `Division_by_zero`. If `x < 0` or `y < 0`, the result of `Int32.rem x y` is not specified and depends on the platform.

```
val succ : int32 -> int32
```

Successor. `Int32.succ x` is `Int32.add x Int32.one`.

```
val pred : int32 -> int32
```

Predecessor. `Int32.pred x` is `Int32.sub x Int32.one`.

```
val abs : int32 -> int32
```

Return the absolute value of its argument.

```
val max_int : int32
```

The greatest representable 32-bit integer, $2^{31} - 1$.

```
val min_int : int32
```

The smallest representable 32-bit integer, -2^{31}.

```
val logand : int32 -> int32 -> int32
```

Bitwise logical and.

```
val logor : int32 -> int32 -> int32
```

Bitwise logical or.

```
val logxor : int32 -> int32 -> int32
```

Bitwise logical exclusive or.

```
val lognot : int32 -> int32
```

Bitwise logical negation

```
val shift_left : int32 -> int -> int32
```

`Int32.shift_left x y` shifts `x` to the left by `y` bits. The result is unspecified if `y < 0` or `y >= 32`.

```
val shift_right : int32 -> int -> int32
```

`Int32.shift_right x y` shifts `x` to the right by `y` bits. This is an arithmetic shift: the sign bit of `x` is replicated and inserted in the vacated bits. The result is unspecified if `y < 0` or `y >= 32`.

```
val shift_right_logical : int32 -> int -> int32
```

`Int32.shift_right_logical x y` shifts `x` to the right by `y` bits. This is a logical shift: zeroes are inserted in the vacated bits regardless of the sign of `x`. The result is unspecified if `y < 0` or `y >= 32`.

```
val of_int : int -> int32
```

Convert the given integer (type `int`) to a 32-bit integer (type `int32`).

```
val to_int : int32 -> int
```

Convert the given 32-bit integer (type `int32`) to an integer (type `int`). On 32-bit platforms, the 32-bit integer is taken modulo 2^{31}, i.e. the high-order bit is lost during the conversion. On 64-bit platforms, the conversion is exact.

```
val of_float : float -> int32
```

Convert the given floating-point number to a 32-bit integer, discarding the fractional part (truncate towards 0). The result of the conversion is undefined if, after truncation, the number is outside the range [`Int32.min_int`, `Int32.max_int`].

```
val to_float : int32 -> float
```

Convert the given 32-bit integer to a floating-point number.

```
val of_string : string -> int32
```

Convert the given string to a 32-bit integer. The string is read in decimal (by default) or in hexadecimal, octal or binary if the string begins with `0x`, `0o` or `0b` respectively. Raise `Failure "int_of_string"` if the given string is not a valid representation of an integer.

```
val to_string : int32 -> string
```

Return the string representation of its argument, in signed decimal.

```
val format : string -> int32 -> string
```

`Int32.format fmt n` return the string representation of the 32-bit integer `n` in the format specified by `fmt`. `fmt` is a `Printf`-style format containing exactly one `%d`, `%i`, `%u`, `%x`, `%X` or `%o` conversion specification. This function is deprecated; use `Printf.sprintf` with a `%lx` format instead.

Similar operations for Int64, plus conversion operations :

```
val of_int32 : int32 -> int64
```

Convert the given 32-bit integer (type `int32`) to a 64-bit integer (type `int64`).

```
val to_int32 : int64 -> int32
```

Convert the given 64-bit integer (type `int64`) to a 32-bit integer (type `int32`). The 64-bit integer is taken modulo 2^{32}, i.e. the top 32 bits are lost during the conversion.

```
val of_nativeint : nativeint -> int64
```

Convert the given native integer (type `nativeint`) to a 64-bit integer (type `int64`).

```
val to_nativeint : int64 -> nativeint
```

Convert the given 64-bit integer (type `int64`) to a native integer. On 32-bit platforms, the 64-bit integer is taken modulo 2^{32}. On 64-bit platforms, the conversion is exact.

```
val bits_of_float : float -> int64
```

Return the internal representation of the given float according to the IEEE 754 floating-point ``double format'' bit layout. Bit 63 of the result represents the sign of the float; bits 62 to 52 represent the (biased) exponent; bits 51 to 0 represent the mantissa.

```
val float_of_bits : int64 -> float
```

Return the floating-point number whose internal representation, according to the IEEE 754 floating-point ``double format'' bit layout, is the given `int64`.

Notes :

Performance notice: values of type `int64` /int32 occupy more memory space than values of type `int`, and arithmetic operations on `int64` /int32 are generally slower than those on `int`. Use int32/ `int64` only when the application requires exact 64-bit arithmetic.
32

Integer and floating point numbers in OCAML

- Numbers in OCAML can be :
  - integers of type int, between min_int and max_int, denoted in decimal notation :
    digit+
  - floating point numbers, of type float, between min_float and max_float, denoted in usual scientific notation :
    [±] digit+ . [digit+] [e [±] digit+]
  - le nombre rationels (librairie num) ;

```
# min_int;;
- : int = -1073741824
# max_int;;
- : int = 1073741823
# min_float;;
- : float = 2.22507385851e-308
# max_float;;
- : float = 1.79769313486e+308
```

- Conversions between integers and float are always explicit :

```
# int_of_float 10.5;;
- : int = 10
# float_of_int 10;;
- : float = 10.
```

- [Because of the type inference algorithm], no arithmetic operator overloading is possible :

```
# 10 / 3;;
- : int = 3
# 10.0 /. 3.0;;
- : float = 3.33333333333
# 10.0 / 3.0;;
This expression has type float but is here used with type int
```

Built-in floating point (float) operations & functions

operations:

```
# 5. +. 4.0;;
- : float = 9.
# 5. *. 4.;;
- : float = 20.
# 5. -. 4.;;
- : float = 1.
# 4. /. 5.;;
- : float = 0.8
# 2. ** 5.;;
- : float = 32.
# 5. - 4.;;
This expression has type float but is here used with type int
# (int_of_float 5.) - 4;;
- : int = 1
```

Functions:

```
# exp 1.;;
- : float = 2.71828182846
# log 10.;;
- : float = 2.30258509299
# log10 10;;
This expression has type int but is here used with type float
# sqrt 2;;
This expression has type int but is here used with type float
# sin 3.14159;;
- : float = 2.65358979335e-06
```

Floating-Point Arithmetic

`val (~-.) : float -> float`	Unary negation
`val (+.) : float -> float -> float`	Floating-point addition
`val (-.) : float -> float -> float`	Floating-point subtraction
`val (*.) : float -> float -> float`	Floating-point multiplication
`val (/.) : float -> float -> float`	Floating-point division
`val (**) : float -> float -> float`	Exponentiation
`val sqrt : float -> float`	Square root
`val exp : float -> float`	Exponential
`val log : float -> float`	Natural logarithm
`val log10 : float -> float`	Base-10 logarithm
`val sin : float -> float`	The usual trigonometric functions
`val cos : float -> float`	
`val tan : float -> float`	
`val asin : float -> float`	
`val acos : float -> float`	
`val atan : float -> float`	
`val atan2 : float -> float -> float`	
`val sinh : float -> float`	The usual hyperbolic functions
`val cosh : float -> float`	
`val tanh : float -> float`	
`val ceil : float -> float`	Round to the smallest integer greater than
`val floor : float -> float`	Round to the greatest integer less than
`val abs_float : float -> float`	Absolute
`val mod_float : float -> float -> float`	Modulo
`val frexp : float -> float * int`	`frexp f` returns the pair of the significant and the exponent of `f`.
`val ldexp : float -> int -> float`	`ldexp f n` returns `x *. 2 ** n`
`val modf : float -> float * float`	`modf f` returns the pair of the frational and integral part of `f`.
`val float : int -> float`	Convert an integer to a floating-point
`val float_of_int : int -> float`	
`val truncate : float -> int`	Convert a floating-point to a integer
`val int_of_float : float -> int`	

Characters and character strings in OCAML

- Characters have type **char** while character strings have type **string** :

```
# 'a';;
- : char = 'a'
# "a";;
- : string = "a"
# "this is a
  character
  string";;
- : string = "this is a\ncharacter\nstring"
# "a\"b\\c double quote and backslash";;
- : string = "a\"b\\c double quote and backslash"
# print_string "a\"b\\c double quote and backslash";;
a"b\c double quote and backslash- : unit = ()
# "string on a \
  single line";;
- : string = "string on a single line"
# print_string "\\\"\t\n\r\b special characters in strings";;
\"
special characters in strings- : unit = ()
#
```

Character operations

```
val int_of_char : char -> int
val char_of_int : int -> char
```

Return the ASCII code of the argument.
Return the character with the given ASCII code.

- Special characters in strings :

Caractère	Notation dans une chaîne
\	\\
"	\"
tabulation	\t
line-feed	\n ou *return* (changement de ligne)
end-of-line	\r
backspace	\b

\ followed by return ↵ can be used to continue a string on the next line without insertion of a line-feed (\n) in the character string

---

' is the left quote, for characters
" is the double quote, for strings

- conversions from and to characters (ASCII code) :

```
# int_of_char 'e';;
- : int = 101
# int_of_char 'é';;
- : int = 233
# char_of_int 45;;
- : char = '-'
# char_of_int 215;; (* not printable *)
- : char = '\215'
```

- Conversion from and to strings :

```
# string_of_int 12345;;
- : string = "12345"
# int_of_string "12345";;
- : int = 12345
# string_of_float 3.14159;;
- : string = "3.14159"
# float_of_string "3.14159";;
- : float = 3.14159
# string_of_bool true;;
- : string = "true"
# bool_of_string "false";;
- : bool = false
```

String conversion functions

`val string_of_bool : bool -> string`	Return the string representation of a boolean.
`val bool_of_string : string -> bool`	Convert the given string to a boolean.
`val string_of_int : int -> string`	Return the string representation of an integer
`val int_of_string : string -> int`	Convert the given string to an integer. The string is read in decimal (by default) or in hexadecimal, octal or binary if the string begins with 0x, 0o or 0b respectively.
`val string_of_float : float -> string`	Return the string representation of a floating-point number.
`val float_of_string : string -> float`	Convert the given string to a float.

- operations on strings (module "String") :

```
val length : string -> int
Return the length (number of characters) of the given string.
String.length "0123456789";;
- : int = 10

val get : string -> int -> char
String.get s n returns character number nin string s. The first
character is character number 0. The last character is character number
String.length s - 1. Raise Invalid_argument if nis outside the range 0
to (String.length s - 1). You can also write s.[n] instead of
String.get s n.
# String.get "0123456789" 5;;
- : char = '5'
# String.get "0123456789" 100;;
Exception: Invalid_argument "String.get".

val set : string -> int -> char -> unit
String.set s n c modifies string sin place, replacing the character
number nby c. Raise Invalid_argument if nis outside the range 0 to
(String.length s - 1). You can also write s.[n] <- c instead of
String.set s n c.
# String.set "0123456789" 5 'X';;
- : unit = ()
# String.set "0123456789" 100 'X';;
Exception: Invalid_argument "String.set".
```

```
val create : int -> string
String.create n returns a fresh string of length n. The string initially
contains arbitrary characters.
# String.create 5;;
- : string = "\000\000\015\000\000"

val make : int -> char -> string
String.make n c returns a fresh string of length n, filled with the
character c.
# String.make 5 'X';;
- : string = "XXXXX"

val copy : string -> string
Return a copy of the given string.
# String.copy "abcdef";;
- : string = "abcdef"


val sub : string -> int -> int -> string
String.sub s start len returns a fresh string of length len, containing
the characters number start to start + len - 1 of string s. Raise
Invalid_argument if start and len do not designate a valid substring of
s; that is, if start < 0, or len < 0, or start + len > String.length s.
# String.sub "0123456789" 3 2;;
- : string = "34"
# String.sub "0123456789" 3 100;;
Exception: Invalid_argument "String.sub".

val fill : string -> int -> int -> char -> unit
String.fill s start len c modifies string sin place, replacing the
characters number start to start + len - 1 by c. Raise Invalid_argument
if start and len do not designate a valid substring of s.
# let s = "0123456789";;
val s : string = "0123456789"
# String.fill s  3 5 'X';;
- : unit = ()
# s;;
- : string = "012XXXXX89"

val blit : string -> int -> string -> int -> int -> unit
String.blit src srcoff dst dstoff len copies len characters from string
src, starting at character number srcoff, to string dst, starting at
character number dstoff. It works correctly even if src and dst are the
same string, and the source and destination chunks overlap. Raise
Invalid_argument if srcoff and len do not designate a valid substring of
src, or if dstoff and len do not designate a valid substring of dst.
# let s = "0123456789";;
val s : string = "0123456789"
# String.blit "abcdefghijklmnopqrstuvwxyz" 2 s 3 4;;
- : unit = ()
# s;;
- : string = "012cdef789"
#

Concatenation:
# "0123" ^ "456" ^ "789";;
- : string = "0123456789"
val concat : string -> string list -> string
String.concat sep sl catenates the list of strings sl, inserting the
separator string sep between each.
#  String.concat "," ["a"; "bb"; "ccc"; "dddd"];;
- : string = "a,bb,ccc,dddd"
```

```
val escaped: string -> string
Return a copy of the argument, with special characters represented by
escape sequences, following the lexical conventions of OCaml.
# String.escaped "\n\t\r";;
- : string = "\\n\\t\\013"
# print_string (String.escaped "\n\t\r");;
\n\t\013- : unit = ()

val index: string -> char -> int
String.index s c returns the position of the leftmost occurrence of
character cin string s. Raise Not_found if c does not occur in s.
# String.index "0123456789876543210" '7';;
- : int = 7

val rindex: string -> char -> int
String.rindex s c returns the position of the rightmost occurrence of
character c in string s. Raise Not_found if cdoes not occur in s.
# String.rindex "0123456789876543210" '7';;
- : int = 11
# String.index "0123456789876543210" 'X';;
Exception: Not_found.

val index_from: string -> int -> char -> int
val rindex_from: string -> int -> char -> int
Same as String.index and String.rindex, but start searching at the
character position given as second argument. String.index s c is
equivalent to String.index_from s 0 c, and String.rindex s c to
String.rindex_from s (String.length s - 1) c.
# String.index_from "0123456789876543210"  4 '2';;
- : int = 16
# String.rindex_from "0123456789876543210" 10 '7';;
- : int = 7

val contains : string -> char -> bool
String.contains s c tests if character cappears in the string s.
# String.contains "0123456789" '0';;
- : bool = true
# String.contains "0123456789" 'X';;
- : bool = false

val contains_from : string -> int -> char -> bool
String.contains_from s start c tests if character c appears in the
substring of s starting from start to the end of s. Raise
Invalid_argument if start is not a valid index of s.
# String.contains_from "0123456789" 3 '0';;
- : bool = false
# String.contains_from "0123456789" 1 '2';;
- : bool = true
# String.contains_from "0123456789" 100 '0';;
Exception: Invalid_argument "String.contains_from".

val rcontains_from : string -> int -> char -> bool
String.rcontains_from s stop c tests if character c appears in the
substring of s starting from the beginning of s to index stop. Raise
Invalid_argument if stop is not a valid index of s.
# String.rcontains_from "0123456789" 7 '5';;
- : bool = true
# String.rcontains_from "0123456789" 4 '5';;
- : bool = false
# String.rcontains_from "0123456789" 100  '5';;
Exception: Invalid_argument "String.rcontains_from".
```

```
val uppercase: string -> string
Return a copy of the argument, with all lowercase letters translated to
uppercase, including accented letters of the ISO Latin-1 (8859-1)
character set.
# String.uppercase "abc123éèùàç";;
- : string = "ABC123\201\200\217\192\199"
# print_string (String.uppercase "abc123éèùàç");;
ABC123ÉÈÙÀÇ- : unit = ()

val lowercase: string -> string
Return a copy of the argument, with all uppercase letters translated to
lowercase, including accented letters of the ISO Latin-1 (8859-1)
character set.
# print_string (String.lowercase "ABC123ÉÈÙÀÇ");;
abc123éèùàç- : unit = ()


val capitalize: string -> string
Return a copy of the argument, with the first letter set to uppercase.

val uncapitalize: string -> string
Return a copy of the argument, with the first letter set to lowercase.
```

- Note : OCAML is not a purely functional language since it provides imperative features (such as procedures with side effects like set, fill, etc... which modify strings in place).

Using standard modules and libraries in OCAML

- Built-in operations are known by the compiler / interpreter / top-level and can be used freely
- Operations or functions of modules in the standard OCAML library are not known a priori by the compiler.

```
# length "0123456789";;
Unbound value length
```

→ It is necessary to indicate to the compiler in which (predefined) module to find the operation.

- Either by prefixing the module operation by the module name

```
# String.length "0123456789";;
- : int = 10
```

- Or by opening the scope of the module, so that operations & functions no longer need to be prefixed by the module name:

```
# open String;;
# length "0123456789";;
- : int = 10
# make 10 'X';;
- : string = "XXXXXXXXXX"
```

Comparison of basic types

- Comparison operators (=, <>, <, >, <=, >=, compare, min, max, ==, !=) are polymorphic (overloaded). They work on objects of the same basic type:

```
# 1=1;;
- : bool = true
# 1=2;;
- : bool = false
# 1.=1.;;
- : bool = true
# 'A'='X';;
- : bool = false
# "abc"="abc";;
- : bool = true
# "abc" < "abd";;
- : bool = true
# compare 2 3;;
- : int = -1
# min 1 2;;
- : int = 1
# max 1. 2.;;
- : float = 2.
# true = false;;
- : bool = false
# true = true;;
- : bool = true
# 1 = 1.;;
This expression has type float but is here used with type int
```

- In place / address / physical comparison:

```
# 1 == 1;;
- : bool = true
# "abc" = "abc";;
- : bool = true
# "abc" == "abc";;
- : bool = false
# let s = "abc";;
val s : string = "abc"
# s == s;;
- : bool = true
```

- **Comparison**

`val (=) : 'a -> 'a -> bool`	Structural equality
`val (<>) : 'a -> 'a -> bool`	Negation of `(=)`
`val (<) : 'a -> 'a -> bool`	Structural ordering: Less than
`val (>) : 'a -> 'a -> bool`	Structural ordering: Greater than
`val (<=) : 'a -> 'a -> bool`	Structural ordering: Less than, or equal to
`val (>=) : 'a -> 'a -> bool`	Structural ordering: Greater than, or equal to
`val compare: 'a -> 'a -> int`	`compare x y` return 0 if `x=y`, a negative number if `x<y` and a positive number if `x>y`.
`val min: 'a -> 'a -> 'a`	Return the smaller of the two arguments
`val max: 'a -> 'a -> 'a`	Return the greater of the two arguments
`val (==) : 'a -> 'a -> bool`	Physical equality
`val (!=) : 'a -> 'a -> bool`	Negation of `(==)`

Input-output of basic values

Output functions :

```
# print_char 'A';;
A- : unit = ()
# print_string "abcdef";;
abcdef- : unit = ()
# print_int 12345;;
12345- : unit = ()
# print_float 1.5;;
1.5- : unit = ()
# print_string "abcde";;
abcde- : unit = ()
# print_newline ();;

- : unit = ()
# print_endline "abcde";;
abcde
- : unit = ()
# print_string "abcde"; print_newline ();;
abcde
- : unit = ()
#
```

Input functions

```
# read_line ();;
1234567890
- : string = "1234567890"
# read_int ();;
123456789
- : int = 123456789
# read_int ();;
99999999999999999999999999999999
- : int = -1
# read_float ();;
3.14159
- : float = 3.14159
#
```

Output functions on standard output

`val print_char : char -> unit`	Print a character
`val print_string : string -> unit`	Print a string
`val print_int : int -> unit`	Print an integer, in decimal
`val print_float : float -> unit`	Print a floating-point number, in decimal
`val print_endline : string -> unit`	Print a string, followed by a newline
`val print_newline : unit -> unit`	Print a newline character

Input functions on standard input

`val read_line : unit -> string`	Flush standard output, then read characters from standard input until a newline character is encountered. Return the string of all characters read, without the newline character at the end.
`val read_int : unit -> int`	Flush standard output, then read one line from standard input and convert it to an integer.
`val read_float : unit -> float`	Flush standard output, then read one line from standard input and convert it to a floating-point number.

Defining values and functions at the top-level

- At the top level, one can bind values to identifiers :

```
# let x = 10;;
val x : int = 10
# x * x;;
- : int = 100
# y;;
Unbound value y
# X;; (* case sensitive *)
Unbound constructor X
#
```

This is a global binding, not a variable since it is not modifiable by assignment.

- Successive binding respect the block structure (a new binding hides the previous one).

```
% ocaml
        Objective Caml version 3.06

# let x = 10;;
val x : int = 10
# let incr a = x + a;;
val incr : int -> int = <fun>
# let x = 100;;
val x : int = 100
# x;;
- : int = 100
# incr 1;;
- : int = 11
# ^D
%
```

- Identifiers :

  - Les identificateurs sont des suites de lettres ou chiffres ou _ ou ', commençant par une lettre et n'étant pas un mot réservé ;
  - On distingue les minuscules des majuscules ;
  - Au maximum 256 caractères.

Local binding of values and functions :

- Instead of global / top-level bindings, one can use local bindings which bind a value to an identifier with an expression :

let x = expr1 **in** expr2

The value of x in expr2 is that of expr1 (evaluated only once) :

```
% ocaml
        Objective Caml version 3.06

# let x = 3 in
  x * x;;
- : int = 9
# x;;
Unbound value x
# ^D
%
```

- The binding is local according to the block structure (without modifying the global definitions, if any) :

```
% ocaml
        Objective Caml version 3.06

# let x = 10;;
val x : int = 10
# let x = 100 in x;;
- : int = 100
# x;;
- : int = 10
# let x = 1000 in
  (let x = 10000 in x) + x;;
- : int = 11000
# ^D
%
```

Defining recursive functions

- Recursive functions can be defined globally are locally by the "let rec":
- Global recursive definition:

```
% ocaml
        Objective Caml version 3.06

# let rec plus x y = if x = 0 then y else 1 + (plus (x - 1) y);;
val plus : int -> int -> int = <fun>
# plus 10 20;;
- : int = 30
# plus ;;
- : int -> int -> int = <fun>
```

- Local recursive definition:

```
# let rec  plus'  x y = if x = 0 then y else 1 + (plus' (x - 1) y) in
  plus' 15 25;;
  - : int = 40
# plus' 15 25;;
Unbound value plus'
```

- Let versus let rec definition:

```
# let mult x y = if x = 0 then 0 else plus y (mult (x - 1) y);;
Unbound value mult
# let rec mult x y = if x = 0 then 0 else plus y (mult (x - 1) y);;
val mult : int -> int -> int = <fun>
# mult 3 5;;
- : int = 15
```

- Bloc structure:

```
# let f x = 0;;
val f : 'a -> int = <fun>
# let f x = f x;;
val f : 'a -> int = <fun>
# f 17;;
- : int = 0
```

- Non termination:

```
# let rec g x = g x;;
val g : 'a -> 'b = <fun>
# g 0;;
^CInterrupted.
# ^D
%
```

Scope of local variables in let [rec] definitions

- The scope of x in a global, top-level definition
  let [rec] $x = e$;;
  is that of all expressions that follow.

- The scope of x in a local definition
  let $x = e_1$ in e_2
  is exactly e_2, while for a recursive definition
  let rec $x = e_1$ in e_2
  it is both e_1 (recursivity) and e_2.

```
% ocaml
        Objective Caml version 3.06

# let cube x = let sqr x = x * x in x * sqr x;;
val cube : int -> int = <fun>
# cube 5;;
- : int = 125
# sqr 5;;
Unbound value sqr
# ^D
%
```

- In "let $x = e_1$ in e_2", an "x" used in e_1 is a global one, if any (while in e_2 it is the local one with a value as given by the evaluation of e_1):

```
% ocaml
        Objective Caml version 3.06

# let x = x in x;;
Unbound value x
# let x = 10;;
val x : int = 10
# let x = x in x;;
- : int = 10
# ^D
%
```

- In "let rec $x = e_1$ in e_2", the local x is both visible in e_1 (as a recursive use) and e_2.

```
% ocaml
        Objective Caml version 3.06

# let prod x y = if x = 0 then 0 else prod (x - 1) + y;;
Unbound value prod
# let rec prod x y = if x = 0 then 0 else (prod (x - 1) y) + y;;
val prod : int -> int -> int = <fun>
# prod 3 4;;
- : int = 12
#
```

- In case of redefinition of an existing function with the same type, the omition of rec can easily be an uncaught error :

```
% ocaml
        Objective Caml version 3.06

# open Int32;;
# add;;
- : int32 -> int32 -> int32 = <fun>
# let succ x = add x one;;
val succ : int32 -> int32 = <fun>
# let pred x = sub x one;;
val pred : int32 -> int32 = <fun>
# let  add x y = if x = zero then y else succ (add (pred x) y);;
val add : int32 -> int32 -> int32 = <fun>
# add zero one;;
- : int32 = <int32 1>
# add 0 1;;
This expression has type int but is here used with type int32
# ^D
%
```

old definition

new definition

A let rec must be used to get a correct recursive definition.

Non termination of recursive functions

```
% ocaml
        Objective Caml version 3.06

# let rec loop = loop;;
This kind of expression is not allowed as right-hand side of `let rec'
# let rec loop x = loop x;;
val loop : 'a -> 'b = <fun>
# loop ();;
^CInterrupted.
# ^D
%
```

But for trivial cases, the OCAML cannot detect potential non-termination (since the problem is undecidable).

A non terminating execution must be stopped manually by ctrl + C [or will get out of memory by stack overflow].

Functions with multiple arguments

- Multiple argument are evaluated left to right

```
# let f x y z = x + y + z;;
val f : int -> int -> int -> int = <fun>
# f 10 20 30;;
- : int = 60
```

- f x y z is (((f x) y) z) :

```
# f 10 20 - 5 30;;
This expression has type int -> int but is here used with type int
# f 10 (20 - 5) 30;;
- : int = 55
#
```

- Anonymous functions :

```
# (function x -> (x + 1)) 5;;
- : int = 6
# (function x -> function y -> x + y) 5 7;;
- : int = 12
```

- Note :

  A function with two arguments f x y

is

  A function with one argument f x which returns a function (f x) which take the second argument y and returns a result ((f x) y)

hence all function have only one argument (as shown in the anonymous case).

Defining Values

Global : `let ident = expression;;`
Local : `let ident = expression in ...`
Multiple : `let ... and ident2 = expression2 and ident3 = ...`
Recursive : `let rec x = ...`

Simultaneous definition of (recursive) functions in OCAML

One can define simultaneously non recursive functions as :

```
let f1 x1 = ....
and f2 x2 = ...
...
and fn xn = .... [in e] ;;
```

(the "in e" is for the case of a local declaration in expression e).

```
# let sqr x = x * x
  and cube x = x * x * x;;
val sqr : int -> int = <fun>
val cube : int -> int = <fun>
# sqr 10;;
- : int = 100
# cube 10;;
- : int = 1000
```

The functions defined simultaneously cannot be mutually dependent or recursive :

```
# let sqr' x = x * x
  and cube' x = (sqr' x) * x;;
Unbound value sqr'
```

For mutually dependent or recursive functions use the "let rec" :

```
# let rec sqr' x = x * x
  and cube' x = (sqr' x) * x;;
  val sqr' : int -> int = <fun>
val cube' : int -> int = <fun>
# cube' 2;;
- : int = 8
#
```

Polymorphic functions in OCAML

- A function can have polymorphic formal parameters (of unknown type denoted 'a, 'b, ...) and take monomorphe actual parameters (of known type "int", "bool", ...) :

```
% ocaml
        Objective Caml version 3.06

# succ;;
- : int -> int = <fun>
# let id x = x;;
val id : 'a -> 'a = <fun>
# id true;;
- : bool = true
# id ();;
- : unit = ()
# id 1;;
- : int = 1
# id 1.5;;
- : float = 1.5
# id succ;;
- : int -> int = <fun>
# id id;;
- : '_a -> '_a = <fun>
#
```

- The compiler implement the actual parameter as a pointeur on the actual parameter (so called boxing) and manipulates only this pointer in the function body. Since the value (of unknown type) is not touched during execution of the function body, it can be of any type.
- A functional takes functions as parameters and returns functions :

```
# let o f g x = (f (g x));;
val o : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
# o succ succ;;
- : int -> int = <fun>
# ((o succ succ) 0);;
- : int = 2
# (o (o succ succ) succ) 0;;
- : int = 3
```

Examples of higher-order polymorphic functionals in OCAML

- Examples of very similar function definitions:

```
% ocaml
        Objective Caml version 3.06

# let rec sum x = if x = 0 then 0
  else (x + sum (x - 1));;
val sum : int -> int = <fun>
# sum 5;;
- : int = 15
# let rec fact x = if x = 0 then 1
  else (x + fact (x - 1));;
val fact : int -> int = <fun>
# fact 5;;
- : int = 16
```

- A functional (or higher-order function taking functions as parameters or results). For example, we can define the "Iteration functional":

```
# let rec iter f a x = if x = 0 then a
* else (f x (iter f a (x - 1)));;
val iter : (int -> 'a -> 'a) -> 'a -> int -> 'a = <fun>
```

$$iter\ f\ a\ n = (f\ n\ (f\ n-1\ (\ldots\ (f\ 1\ a)\ \ldots))).$$

- Examples:

```
# iter (+) 0 5;;
- : int = 15

# mul;;
- : int -> int -> int = <fun>
# iter mul 1 5;;
- : int = 120

# let div x y = y /. (float_of_int x);;
val div : int -> float -> float = <fun>
# (* 1/!n *)
  let factmone x = iter div 1.0 x;;
val factmone : int -> float = <fun>
# factmone 2;;
- : float = 0.5
# factmone 5;;
- : float = 0.00833333333333
```

```
# let nconc x s = (string_of_int x) ^s;;
val nconc : int -> string -> string = <fun>
# nconc 55 "ab";;
- : string = "55ab"
# iter;;
- : (int -> 'a -> 'a) -> 'a -> int -> 'a = <fun>
# iter nconc "abc" 0;;
- : string = "abc"
# iter nconc "abc" 1;;
- : string = "1abc"
# iter nconc "abc" 2;;
- : string = "21abc"
# iter nconc "abc" 3;;
- : string = "321abc"
# iter nconc "abc" 4;;
- : string = "4321abc"
# iter nconc "abc" 5;;
- : string = "54321abc"
```

Defining Functions

A single argument: `let f x = expression`
Several arguments: `let f x y = expression`
With matching: `let f = function matching`
Recursive : `let rec f x = ...`

Special cases: a **procedure** is a function with no mathematically sound result, it returns `()`.
For instance, assignment functions return `()`.

Calling Functions

Using parens: `f (x)`
Using a space: `f x`
More than one arguments: `f (x) (y)` or `f x y`

Warning: if an argument is complex you must enclose it between parens.
For instance `f (x + 1)` (and not `f x + 1`, which means `f (x) + 1`, that is `1 + f (x)`).

Special cases: functions with no argument get `()` as argument.
For instance `print_newline` is called with `print_newline ()`.

Using parentheses

In Caml, parens are **significant** and use the mathematical rules of **trigonometric functions**.

In Caml, `1 + 2 * x` **means** `1 + (2 * x)`
as in math.

In Caml, `f x` **means** `f (x)`
as in math `sin x` means `sin (x)`.

In Caml, `f x + g x` **means** `(f x) + (g x)`
as in math `sin x + cos x` means `(sin x) + (cos x)`.
Hence, `f x - y` **means** `(f x) - y`
as in math `sin x - 1` means `(sin x) - 1`.

In Caml, this rule is generalized to any infix operator:
`f x :: g x` **means** `(f x) :: (g x)`.

Anonymous functions

Anonymous functions are lamba-expressions $\lambda x.e$ with no name, written

function $x \rightarrow e$

or fun $x \rightarrow e$

For example :

```
% ocaml
        Objective Caml version 3.06

# function x -> x + 1;;
- : int -> int = <fun>
# (function x -> x + 1) 17;;
- : int = 18
# fun x -> 2. *. x;;
- : float -> float = <fun>
# (fun x -> 2. *. x) 50.;;
- : float = 100.
# ^D
%
```

Pattern-matching / filtering :

```
# let x = 2;;
val x : int = 2
# match x with
    | 0 -> 0
    | 1 -> 2
    | 2 -> 4
    | 3 -> 6
    | _ -> -1;;
```

← default case

```
- : int = 4
# match x with
    0 -> 0
  | 1 -> 2
  | 2 -> 4
  | 3 -> 6;;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
4
- : int = 4
```

← some possible cases are missing

The matching in OCAML is a generalization of a switch in C, a case statement in Pascal, etc. If the value match a case, the corresponding value is returned. If no match is possible, the default case, if any, is missing. If no match is possible, then this is an error, signaled by an exception:

```
# let x = 7;;
val x : int = 7
# match x with
      0 -> 0
    | 1 -> 2
    | 2 -> 4
    | 3 -> 6;;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
4
Exception: Match_failure ("", 0, 62).
#
```

- Several cases can be grouped together :

```
# let x = 2;;
val x : int = 2
# match x with
    0       -> 0
  | 1 | -1 -> 1
  | 2 | -2 -> 4
  | _      -> -1;;
- : int = 4
#
```

- In a matching, a variable can be used to match the argument value, which can be used to denote this value in the returned expression :

```
# let x = 2;;
val x : int = 2
# match x with
    0 -> 0.
  | x -> 1. /. (float_of_in    t x);;
- : float = 0.5

# let rec fact = function
  0 -> 1
  | n -> n * fact (n - 1);;
val fact : int -> int = <fun>
# fact 5;;
- : int = 120
```

- A pattern can be a range :

```
# let y = 'A';;
val y : char = 'A'
# match y with
    'a'..'z' -> 0
  | 'A'..'Z' -> 1
  | '1'..'9' -> 3
  | _        -> 4;;
  - : int = 1
#
```

- A pattern-matching can be filtered by tests :

```
# let x = 2;;
val x : int = 2
# match x with
   | y when y > 0 -> y
   | 0 -> 0
   | y when y < 0 -> -y;;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
1
(However, some guarded clause may match this value.)
- : int = 2
#
```

- Notice that the compiler is not able in general to check statically that the pattern matching is exhaustive (i.e. $(y>0) \vee (y=0) \vee (y<0) = \text{true}$) where the warning. This warning can be avoided (if necessary) by a default case :

```
# let x = -2;;
val x : int = -2
# match x with
   | y when y > 0 -> y
   | 0 -> 0
   |       y when y < 0 -> (- y)
   | _ -> failwith "impossibl  e";;
- : int = 2
```

- The patterns can also be used to match structured values (pairs, lists, records, etc...), see later.

Matching

A pattern matching appears after the keywords `function`, or `with`
(in `function` *matching*, `try ... with` *matching*, or `match ... with` *matching*).
Warning: a pattern matching which is nested inside another one must be enclosed by `begin end` keywords.

A pattern matching is a list of clauses `pattern -> expression`: each clause is tried in the order of presentation; the first clause for which the `pattern` corresponds to (is more general than) the value is selected; the corresponding expression is then returned.

There are constant patterns, variable patterns or compound patterns.

```
| constant_pattern -> expression
| variable_pattern -> expression
| compound_pattern -> expression;;
```

Constant pattern: the constants of the language, such as `1`, `"poi"`, `true`, or constants constructors.
For instance

```
let rec fact = function
| 0 -> 1
| n -> n * fact (n - 1);;
```

Variable pattern: identifiers or the character _
For instance

```
let rec fib = function
| 0 -> 1
| 1 -> 1
| n -> fib (n - 1) + fib (n - 2);;
```

Compound pattern: a constructor applied to a pattern `Constructor pattern`, lists `x :: l`, tuples `(pattern1, pattern2)`, records `{label1=pattern1; ...; labeln=patternn}`.

Complex patterns:

- tuple pattern: `| (pattern, pattern) -> expression`
- alias pattern: `| pattern as ident -> expression`
- or pattern: `| constant_pattern | constant_pattern -> expression`
- range pattern: `| character pattern .. character pattern -> expression`
  Example:

```
let letter = function
| `a` .. `z` | `A` .. `Z` -> true
| _ -> false;;
```

- Clause with guard: `| pattern when condition -> expression`
  Example:

```
let letter = function
| c when c >= `a` && c <= `z`
       || c >= `A` && c <= `Z` -> true
| _ -> false;;
```

Explicit call to the pattern matching

```
match expression with matching
```

For instance:

```
match f 2 with
| 1 -> expression
| n -> expression
```

Anonymous function

With a single argument: `function x -> expression`

With pattern matching on this argument:

```
function
| pattern -> expression
| ...
| pattern -> expression
```

With several arguments: `function x -> function y -> expression`

Definition of functions by matching in O'CAML

A function definition with a matching :

```
let f = function x -> match x with
        | a -> e1
        | b -> e2
        ---
        ;;
```

can be abbreviated as :

```
let f x = match x with
        | a -> e1
        | b -> e2
        --- ;;
```

or even as :

```
let f = function
        | a -> e1
        | b -> e2
        --- ;;
```

Example :

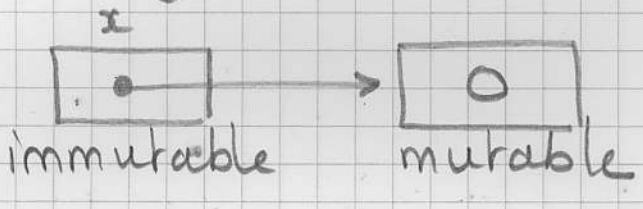
```
# let f = function x -> match x with
    | true -> 1
    | false -> 0;;
val f : bool -> int = <fun>
# let f x = match x with
    | true -> 1
    | false -> 0;;
  val f : bool -> int = <fun>
# let f = function
    | true -> 1
    | false -> 0;;
  val f : bool -> int = <fun>
#
```

References and variables in OCAML

- Bindings of values to variables (as in `let x = 1 ;;` or by actual parameter passing) is unmodifiable / immutable (even for polymorphic parameters which are boxed, that is, at the implementation level, referenced by a pointer).
- A classical variable is a reference in OCAML:

  `let x = ref 0 ;`

  The declaration must provide an initial value whence a type:



- The value pointed to is denoted "`! x`".
- This value can be changed by an assignment of the form `x := e`
- Example:

```
# let x = ref 0;;
val x : int ref = {contents = 0}
# !x;;
- : int = 0
# x := !x + 1;;
- : unit = ()
# !x;;
- : int = 1
#
```

References

Definition: `let x = ref 0 in ...`

Access: operator `!` (`! reference`)

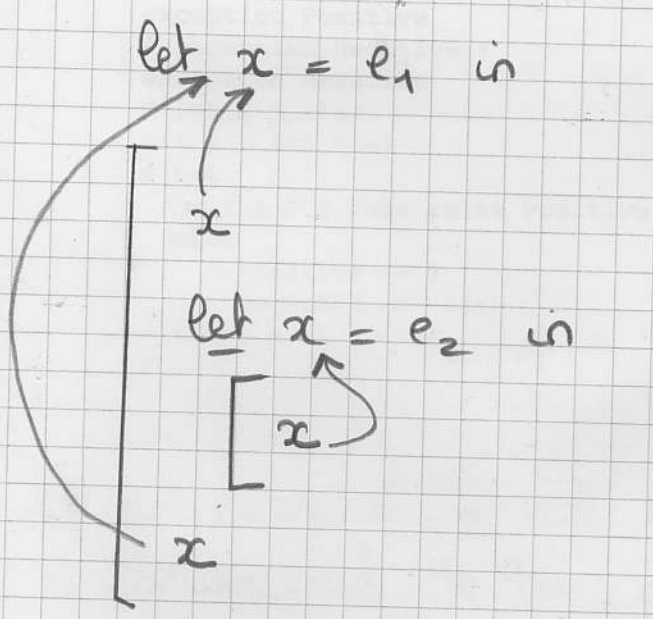
Assignment: operator `:=` (`reference := value`)
For instance: `x := !x + 1`

Block structure of OCAML expressions

The let / let rec expressions satisfy the block structure (originating from Algol 60) :

- The scope of x in let $x = e_1$ in e_2 is e_1, while in let $x = e_1$ in e_2 it is e_1 and e_2
- Outside these blocks, x is not visible. If x is defined, it is some other x.
- Inside these blocks, x hides any other external definition of x, outside the bloc
- Inside these blocks, x is hidden by any internal redeclaration of x.

Here is an example



Exceptions

- When something goes wrong in the evaluation of expressions, a standard exception is raised (which, if nothing is done by the programmer, ends the program exception after printing a message):

```
# 1/0;;
Exception: Division_by_zero.
```

- If an expression e may raise an exception, it can be caught to provide an alternative treatment:

  try e with Ex1 → e1 | Ex2 → e2 ...
  (matching)

```
# try 1/0
  with Division_by_zero -> 0;;
- : int = 0
```

- User exceptions declared and raised in programs:

```
# exception Positive;;
exception Positive
# exception Negative;;
exception Negative
# let x = -1;;
val x : int = -1
# try
    if x > 0 then raise Positive else raise Negative
  with
    | Positive -> 1
    | Negative -> -1;;
- : int = -1
#
```

If an exception is not caught in a block then it is forwarded to the enclosing block. Ultimately, it will arrive at the top level and be printed.

- Example of exception forwarded to the outer block:

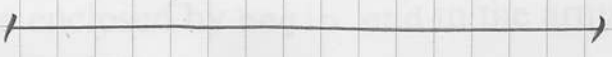
```
# exception Error;;
exception Error
try 1/0 with
  | Error -> 0;;
Exception: Division_by_zero.
```

- Exceptions can carry values, as in:

```
# exception Error of string;;
exception Error of string
# x;;
- : int = -1
# try
    if x > 0 then raise (Error "positive")
             else raise (Error "negative")
  with
   | Error s -> print_endline s
  ;;
negative
- : unit = ()
```

- A catching of an expression can reraise another or the same exception:

```
# exception Error;;
exception Error
# try 1/0 with
  Division_by_zero -> raise Error;;
Exception: Error.
# try 1/0 with
  Division_by_zero -> raise Division_by_zero;;
Exception: Division_by_zero.
#
```



Exceptions

Handling: **try** expression **with** matching
Throwing: **raise** constant_exception or raise (exception_with_argument expression)

Defining exceptions

```
exception Constant_Exception;;
exception Exception_with_argument of
type_expression;;
```

Sequences in OCAML

In a sequence $e_1 ; e_2$, e_1 is executed, its value is ignored, then e_2 is executed and its value is returned (unless an uncaught exception is raised). The type of e_1 must be unit. It is assumed that e_1 makes some side effect, like printing.

```
# let x = 1;;
val x : int = 1
# print_int x; x+1;;
1- : int = 2
#

# exception Error;;
exception Error
# raise Error; 17;;
Exception: Error.
#
```

Sequence

Syntax: `expression; expression`
Warning: must be enclosed by `begin end` in the arms of a conditional `if then else`.

Conditional in OCAML

- The conditional is

  if b then e_1 else e_2

  b must be of boolean type. e_1 and e_2 MUST BE OF THE SAME TYPE.

- There is no form if b then e since no value can be returned when b is false.

```
# if true then 1 else 2;;
- : int = 1
# if false then 1 else 2;;
- : int = 2

# if true then 0;;
This expression has type int but is here used with type unit

# if then then (); 1 else (); 2;;
Syntax error
# if true then (); 1 else (); 2;;
Syntax error
# if true then begin (); 1 end else begin (); 2 end;;
- : int = 1
# if false then ((); 1) else ((); 2);;
- : int = 2
```

Conditional

Syntax: **if** condition **then** expression **else** expression
Warning: if condition then **begin** e1; e2 **end** else **begin** e3; e4 **end**

Standard comparison operators: <, >, <=, >=, <>
Physical equality: ==, !=

Loops in OCAML

- for loops :

```
# for i = 0 to 10 do print_int i done;;
012345678910- : unit = ()
# for i = 10 downto 0 do print_int i done;;
109876543210- : unit = ()
```

- while loops :

```
# let j = ref 10;;
val j : int ref = {contents = 10}
# while !j > 0 do print_int !j; j := !j - 1 done;;
10987654321- : unit = ()
```

- Exceptions can be used to exit loops :

```
# exception Exit of int;;
exception Exit of int
# try
    for i = 0 to 10 do
      if i = 5 then raise (Exit i) else print_int i
    done
  with
  Exit i -> print_int i;;
012345- : unit = ()
```

Loops

```
for i = 0 to 10 do print_int i done
for i = 10 downto 0 do print_int i done
while !j > 0 do j := !j - 1 done
```

File input/output in OCAML

```
% ocaml
        Objective Caml version 3.06

# let file = open_out "file.txt";;
val file : out_channel = <abstr>
# output_string file "abc\n";;
- : unit = ()
# output_string file "efg\n";;
- : unit = ()
# ^D
% cat file.txt
abc
efg
% ocaml
        Objective Caml version 3.06

# let file = open_in "file.txt";;
val file : in_channel = <abstr>
# input_string;;
Unbound value input_string
# input_line file;;
- : string = "abc"
# input_line file;;
- : string = "efg"
# input_line file;;
Exception: End_of_file.
# ^D
%
```

Input-output

- Screen:
  Printing: `print_string`, `print_int`, `print_float`, ...,
  or encore `printf "%d\n" 1`
- Keyboard: `read_line`
- Files:
  - for reading: `open_in`, `close_in`,
    `input_line`, `input_string`, `input_char`
  - for writing: `open_out`, `close_out`,
    `output_string`, `output_char` (structured data:
    input_value, output_value)

Formatted output in OCAML

The C style of formatted output also exists in OCAML :

```
# Printf.fprintf stdout "(%i)-[%s]-[%c]\n" 123 "abc" 'X';;
(123)-[abc]-[X]
- : unit = ()
#
```

Module Printf

Formatted output functions.

```
val fprintf : Pervasives.out_channel -> ('a, Pervasives.out_channel, unit) format -> 'a
```

`fprintf outchan format arg1 ... argN` formats the arguments `arg1` to `argN` according to the format string `format`, and outputs the resulting string on the channel `outchan`.

The format is a character string which contains two types of objects: plain characters, which are simply copied to the output channel, and conversion specifications, each of which causes conversion and printing of one argument.

Conversion specifications consist in the `%` character, followed by optional flags and field widths, followed by one or two conversion character. The conversion characters and their meanings are:

- `d` or `i`: convert an integer argument to signed decimal.
- `u`: convert an integer argument to unsigned decimal.
- `x`: convert an integer argument to unsigned hexadecimal, using lowercase letters.
- `X`: convert an integer argument to unsigned hexadecimal, using uppercase letters.
- `o`: convert an integer argument to unsigned octal.
- `s`: insert a string argument.
- `S`: insert a string argument in Caml syntax (double quotes, escapes).
- `c`: insert a character argument.
- `C`: insert a character argument in Caml syntax (single quotes, escapes).
- `f`: convert a floating-point argument to decimal notation, in the style `dddd.ddd`.
- `e` or `E`: convert a floating-point argument to decimal notation, in the style `d.ddd e+-dd` (mantissa and exponent).
- `g` or `G`: convert a floating-point argument to decimal notation, in style `f` or `e`, `E` (whichever is more compact).
- `b`: convert a boolean argument to the string **`true`** or **`false`**
- `ld`, `li`, `lu`, `lx`, `lX`, `lo`: convert an `int32` argument to the format specified by the second letter (decimal, hexadecimal, etc).
- `nd`, `ni`, `nu`, `nx`, `nX`, `no`: convert a `nativeint` argument to the format specified by the second letter.
- `Ld`, `Li`, `Lu`, `Lx`, `LX`, `Lo`: convert an `int64` argument to the format specified by the second letter.
- `a`: user-defined printer. Takes two arguments and apply the first one to `outchan` (the current output channel) and to the second argument. The first argument must therefore have type `out_channel -> 'b -> unit` and the second `'b`. The output produced by the function is therefore inserted in the output of `fprintf` at the current

point.
- `t`: same as `%a`, but takes only one argument (with type `out_channel -> unit`) and apply it to `outchan`.
- `%`: take no argument and output one `%` character.

The optional flags include:

- `-`: left-justify the output (default is right justification).
- `0`: for numerical conversions, pad with zeroes instead of spaces.
- `+`: for numerical conversions, prefix number with a `+` sign if positive.
- space: for numerical conversions, prefix number with a space if positive.
- `#`: request an alternate formatting style for numbers.

The field widths are composed of an optional integer literal indicating the minimal width of the result, possibly followed by a dot `.` and another integer literal indicating how many digits follow the decimal point in the `%f`, `%e`, and `%E` conversions. For instance, `%6d` prints an integer, prefixing it with spaces to fill at least 6 characters; and `%.4f` prints a float with 4 fractional digits. Each or both of the integer literals can also be specified as a `*`, in which case an extra integer argument is taken to specify the corresponding width or precision.

Warning: if too few arguments are provided, for instance because the `printf` function is partially applied, the format is immediately printed up to the conversion of the first missing argument; printing will then resume when the missing arguments are provided. For example, `List.iter (printf "x=%d y=%d " 1) [2;3]` prints `x=1 y=2 3` instead of the expected `x=1 y=2 x=1 y=3`. To get the expected behavior, do `List.iter (fun y -> printf "x=%d y=%d " 1 y) [2;3]`.

```
val printf : ('a, Pervasives.out_channel, unit) format -> 'a
```

Same as `Printf.fprintf`, but output on `stdout`.

```
val eprintf : ('a, Pervasives.out_channel, unit) format -> 'a
```

Same as `Printf.fprintf`, but output on `stderr`.

```
val sprintf : ('a, unit, string) format -> 'a
```

Same as `Printf.fprintf`, but instead of printing on an output channel, return a string containing the result of formatting the arguments.

```
val bprintf : Buffer.t -> ('a, Buffer.t, unit) format -> 'a
```

Same as `Printf.fprintf`, but instead of printing on an output channel, append the formatted arguments to the given extensible buffer (see module `Buffer`).

```
val kprintf : (string -> string) -> ('a, unit, string) format -> 'a
```

`kprintf k format arguments` is the same as `sprintf format arguments`, except that the resulting string is passed as argument to `k`; the result of `k` is then returned as the result of `kprintf`.

Structured types

- pairs
- records
- lists
- vectors.

Pairs

- If a is an object of type t_1 and b is an object of type t_2 then a, b is a pair of type $t_1 * t_2$. "," is infixed.
- $a_1, a_2, \ldots, a_n$ stands for $(\ldots((a_1, a_2), a_3), \ldots, a_n)$ of type $(\ldots(t_1 * t_2) * t_3) \ldots * t_n)$.

```
# 5, 6;;
- : int * int = (5, 6)
# 5, "abc";;
- : int * string = (5, "abc")
# 6, "abc", true, 1.5;;
- : int * string * bool * float = (6, "abc", true, 1.5)
```

- **Pair operations**

`val fst : 'a * 'b -> 'a`	Return the first component of a pair.
`val snd : 'a * 'b -> 'b`	Return the second component of a pair.

```
# fst;;
- : 'a * 'b -> 'a = <fun>
# fst (1, 2);;
- : int = 1
# snd;;
- : 'a * 'b -> 'b = <fun>
# snd (1, 2);;
- : int = 2
# fst 5, 6
  ;;
This expression has type int but is here used with type 'a * 'b
# fst (5, 6);;
- : int = 5
```

- Example of definition of operations on pairs:

```
# let pair x y = x, y;;
val pair : 'a -> 'b -> 'a * 'b = <fun>
# pair 5, 6;;
- : ('_a -> int * '_a) * int = (<fun>, 6)
# pair 5 6;;
- : int * int = (5, 6)
```

- Good usage of types:

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

- `T -> T'` est le type des fonctions de `T` dans `T'` ;
- `T * T'` est le type des paires de premier élément de type `T` et de second élément `T'` ;
- `->` et `*` sont associatifs à droite ;
- Types polymorphes : les `'a`, `'b`, `'c`, ... dénotent un type monomorphe (`int`, `(int * string)`, ... quelconque.

Examples of incorrect types :

```
# pair  pair 4 5 6;;
This function is applied to too many arguments
# pair (pair 4 5) 6;;
- : (int * int) * int = ((4, 5), 6)
# fst pair 5 6;;
This expression has type 'a -> 'b -> 'a * 'b but is here used with type
  ('c -> 'd -> 'e) * 'f
# fst (pair 5 6);;
- : int = 5
```

Examples of functionals :

If $f\ a_1 \dots a_n$ is a function with n parameters then $(f\ a)$ is a function with $n-1$ parameters such that

$$(f\ a)\ a_2 \dots a_n = f\ a\ a_2 \dots a_n .$$

for example :

```
# pair;;
- : 'a -> 'b -> 'a * 'b = <fun>
# let pair5 = pair 5;;
val pair5 : '_a -> int * '_a = <fun>
# pair5 6;;
- : int * int = (5, 6)
# pair5;;
- : int -> int * int = <fun>
```

```
# let prod x y = x * y;;
val prod : int -> int -> int = <fun>
# prod 10 15;;
- : int = 150
# prod 10;;
- : int -> int = <fun>
#

# let times10 = prod 10;;
val times10 : int -> int = <fun>
# times10 20;;
- : int = 200
# times10 times10 20
  ;;
This function is applied to too many arguments
# times10 times10 20;;
This function is applied to too many arguments
# times10 (times10 20);;
- : int = 2000
#
```

- Functions with two arguments versus functions with a single pair argument

```
# let sigma x y = x + y;;
val sigma : int -> int -> int = <fun>
# let sigma' (x, y) = x + y;;
val sigma' : int * int -> int = <fun>
# sigma 10 20;;
- : int = 30
# sigma' 10, 20;;
This expression has type int but is here used with type int * int
# sigma' (10, 20);;
- : int = 30
```

- Curryfication / Decurryfication :

```
# let curry f x y = f (x, y);;
val curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
# let uncurry f (x, y) = f x y;;
val uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c = <fun>
# mul;;
- : int -> int -> int = <fun>
# uncurry mul;;
- : int * int -> int = <fun>
# (uncurry mul) (2, 3);;
- : int = 6
# let add (x, y) = x + y;;
val add : int * int -> int = <fun>
# curry add 4 5;;
- : int = 9
```

Vectors in OCAML

- Arrays are vectors of mutable values.
- Definition :

```
# [| 1; 2; 3|];;
- : int array = [|1; 2; 3|]
# Array.make;;
- : int -> 'a -> 'a array = <fun>
# Array.make 10 0;;
- : int array = [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0|]
# Array.length [| 1; 2; 3|];;
- : int = 3
```

- Access to array element (take care to out of bound indexing)

```
# [| 1; 2; 3|].(0);;
- : int = 1
# [| 1; 2; 3|].(1);;
- : int = 2
# [| 1; 2; 3|].(2);;
- : int = 3
# Array.get [| 1; 2; 3|] 2;;
- : int = 3
# [| 1; 2; 3|].(3);;
Exception: Invalid_argument "Array.get".
```

- Assignment :

```
# let t = [| 1; 2; 3|];;
val t : int array = [|1; 2; 3|]
# t.(0) <- 10;;
- : unit = ()
# t;;
- : int array = [|10; 2; 3|]
```

- Iteration :

```
# t;;
- : int array = [|10; 2; 3|]
# for i = 0 to Array.length t - 1 do t.(i) <- t.(i)+1 done;;
- : unit = ()
# t;;
- : int array = [|11; 3; 4|]
```

Module Array

```
module Array: sig  end
```

Array operations.

```
val length : 'a array -> int
```

Return the length (number of elements) of the given array.

```
val get : 'a array -> int -> 'a
```

`Array.get a n` returns the element number `n` of array `a`. The first element has number 0. The last element has number `Array.length a - 1`.

Raise `Invalid_argument "Array.get"` if `n` is outside the range 0 to `(Array.length a - 1)`. You can also write `a.(n)` instead of `Array.get a n`.

```
val set : 'a array -> int -> 'a -> unit
```

`Array.set a n x` modifies array `a` in place, replacing element number `n` with `x`.

Raise `Invalid_argument "Array.set"` if `n` is outside the range 0 to `Array.length a - 1`. You can also write `a.(n) <- x` instead of `Array.set a n x`.

```
val make : int -> 'a -> 'a array
```

`Array.make n x` returns a fresh array of length `n`, initialized with `x`. All the elements of this new array are initially physically equal to `x` (in the sense of the `==` predicate). Consequently, if `x` is mutable, it is shared among all elements of the array, and modifying `x` through one of the array entries will modify all other entries at the same time.

Raise `Invalid_argument` if `n < 0` or `n > Sys.max_array_length`. If the value of `x` is a floating-point number, then the maximum size

is only `Sys.max_array_length / 2`.

```
val create : int -> 'a -> 'a array
```

Deprecated. `Array.create` is an alias for `Array.make`.

```
val init : int -> (int -> 'a) -> 'a array
```

`Array.init n f` returns a fresh array of length `n`, with element number `i` initialized to the result of `f i`. In other terms, `Array.init n f` tabulates the results of `f` applied to the integers `0` to `n-1`.

Raise `Invalid_argument` if `n < 0` or `n > Sys.max_array_length`. If the return type of `f` is `float`, then the maximum size is only `Sys.max_array_length / 2`.

```
val make_matrix : int -> int -> 'a -> 'a array array
```

`Array.make_matrix dimx dimy e` returns a two-dimensional array (an array of arrays) with first dimension `dimx` and second dimension `dimy`. All the elements of this new matrix are initially physically equal to `e`. The element `(x,y)` of a matrix `m` is accessed with the notation `m.(x).(y)`.

Raise `Invalid_argument` if `dimx` or `dimy` is negative or greater than `Sys.max_array_length`. If the value of `e` is a floating-point number, then the maximum size is only `Sys.max_array_length / 2`.

```
val create_matrix : int -> int -> 'a -> 'a array array
```

Deprecated. `Array.create_matrix` is an alias for `Array.make_matrix`.

```
val append : 'a array -> 'a array -> 'a array
```

`Array.append v1 v2` returns a fresh array containing the concatenation of the arrays `v1` and `v2`.

```
val concat : 'a array list -> 'a array
```

Same as `Array.append`, but concatenates a list of arrays.

```
val sub : 'a array -> int -> int -> 'a array
```

`Array.sub a start len` returns a fresh array

of length `len`, containing the elements number `start` to `start + len - 1` of array `a`.

Raise `Invalid_argument "Array.sub"` if `start` and `len` do not designate a valid subarray of `a`; that is, if `start < 0`, or `len < 0`, or `start + len > Array.length a`.

```
val copy : 'a array -> 'a array
```

`Array.copy a` returns a copy of `a`, that is, a fresh array containing the same elements as `a`.

```
val fill : 'a array -> int -> int -> 'a -> unit
```

`Array.fill a ofs len x` modifies the array `a` in place, storing `x` in elements number `ofs` to `ofs + len - 1`.

Raise `Invalid_argument "Array.fill"` if `ofs` and `len` do not designate a valid subarray of `a`.

```
val blit : 'a array -> int -> 'a array -> int -> int -> unit
```

`Array.blit v1 o1 v2 o2 len` copies `len` elements from array `v1`, starting at element number `o1`, to array `v2`, starting at element number `o2`. It works correctly even if `v1` and `v2` are the same array, and the source and destination chunks overlap.

Raise `Invalid_argument "Array.blit"` if `o1` and `len` do not designate a valid subarray of `v1`, or if `o2` and `len` do not designate a valid subarray of `v2`.

```
val to_list : 'a array -> 'a list
```

`Array.to_list a` returns the list of all the elements of `a`.

```
val of_list : 'a list -> 'a array
```

`Array.of_list l` returns a fresh array containing the elements of `l`.

```
val iter : ('a -> unit) -> 'a array -> unit
```

`Array.iter f a` applies function `f` in turn to all the elements of `a`. It is equivalent to `f a.(0); f a.(1); ...; f a.(Array.length a - 1); ()`.

```
val map : ('a -> 'b) -> 'a array -> 'b array
```

`Array.map f a` applies function `f` to all the elements of `a`, and builds an array with the results returned by `f`: `[| f a.(0); f a.(1); ...; f a.(Array.length a - 1) |]`.

```
val iteri : (int -> 'a -> unit) -> 'a array -> unit
```

Same as `Array.iter`, but the function is applied to the index of the element as first argument, and the element itself as second argument.

```
val mapi : (int -> 'a -> 'b) -> 'a array -> 'b array
```

Same as `Array.map`, but the function is applied to the index of the element as first argument, and the element itself as second argument.

```
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b array -> 'a
```

`Array.fold_left f x a` computes `f (... (f (f x a.(0)) a.(1)) ...) a.(n-1)`, where `n` is the length of the array `a`.

```
val fold_right : ('a -> 'b -> 'b) -> 'a array -> 'b -> 'b
```

`Array.fold_right f a x` computes `f a.(0) (f a.(1) ( ... (f a.(n-1) x) ...))`, where `n` is the length of the array `a`.

Sorting

```
val sort : ('a -> 'a -> int) -> 'a array -> unit
```

Sort an array in increasing order according to a comparison function. The comparison function must return 0 if its arguments compare as equal, a positive integer if the first is greater, and a negative integer if the first is smaller (see below for a complete specification). For example, `Pervasives.compare` is a suitable comparison function, provided there are no floating-point NaN values in the data. After calling `Array.sort`, the array is sorted in place in increasing order. `Array.sort` is guaranteed to run in constant heap space and

(at most) logarithmic stack space.

The current implementation uses Heap Sort. It runs in constant stack space.

Specification of the comparison function: Let `a` be the array and `cmp` the comparison function. The following must be true for all x, y, z in a :

- `cmp x y` > 0 if and only if `cmp y x` < 0
- if `cmp x y` >= 0 and `cmp y z` >= 0 then `cmp x z` >= 0

When `Array.sort` returns, `a` contains the same elements as before, reordered in such a way that for all i and j valid indices of `a` :

- `cmp a.(i) a.(j)` >= 0 if and only if i >= j

```
val stable_sort : ('a -> 'a -> int) -> 'a array -> unit
```

Same as `Array.sort`, but the sorting algorithm is stable (i.e. elements that compare equal are kept in their original order) and not guaranteed to run in constant heap space.

The current implementation uses Merge Sort. It uses `n/2` words of heap space, where `n` is the length of the array. It is usually faster than the current implementation of `Array.sort`.

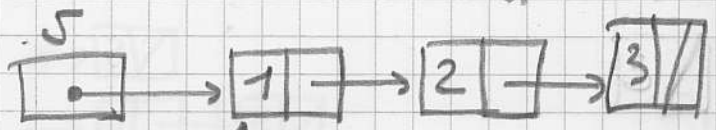
```
val fast_sort : ('a -> 'a -> int) -> 'a array -> unit
```

Same as `Array.sort` or `Array.stable_sort`, whichever is faster on typical input.

List in OCAML

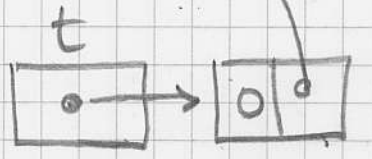
Lists are <u>immutable</u> sequences of objects of the same type. They are represented by chaining

$s = [1; 2; 3]$ is



The main idea is to have sharing so

$t = [0 : s]$ is



Consequently, side effects should be avoided (the lists are said to be immutable) to keep the program understandable. Unaccessible cells are garbage collected automatically.

- <u>List constants</u>:

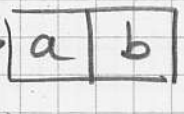
```
# [1;2;3;4];;
- : int list = [1; 2; 3; 4]
# ["a","ab","abc"];;
- : (string * string * string) list = [("a", "ab", "abc")]
# [];;
- : 'a list = []
# [[1;2]; []; [3;4;5]; [6]];;
- : int list list = [[1; 2]; []; [3; 4; 5]; [6]]
# [true; 1];;
This expression has type int but is here used with type bool
# [1;[1]];;
This expression has type 'a list but is here used with type int
```

Note : all elements must be of the <u>SAME</u> type.

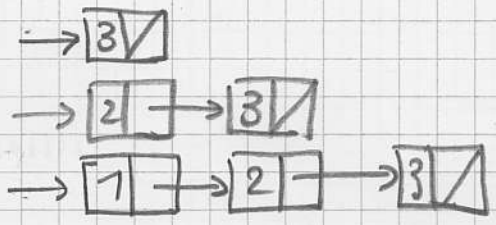
- <u>Confusing list and pair notations</u> :

```
# [1;2];;
- : int list = [1; 2]
# [1,2];;
- : (int * int) list = [(1, 2)]
```

- The "cons" operator (written "::" in OCAML)

a :: b builds a cell 


```
# 3::[];;
- : int list = [3]
# 2::3::[];;
- : int list = [2; 3]
# 1::2::3::[];;
- : int list = [1; 2; 3]
```



- Length of a list :

```
# List.length [];;
- : int = 0
# List.length [1; 2; 3];;
- : int = 3
```

- Head and tail of lists (car and cdr is LISP)

```
# List.hd [1; 2; 3];;
- : int = 1
# List.hd [];;
Exception: Failure "hd".

# List.tl [1; 2; 3];;
- : int list = [2; 3]
# List.tl [];;
Exception: Failure "tl".
```

- Selection of an element :

```
# List.nth [1; 2; 3] 0;;
- : int = 1
# List.nth [1; 2; 3] 10;;
Exception: Failure "nth".
```

- Append (concatenation of lists) :

```
# List.append [1; 2; 3] [4; 5];;
- : int list = [1; 2; 3; 4; 5]
# List.append [1; 2; 3] [];;
- : int list = [1; 2; 3]
# [1; 2; 3] @ [4; 5];;
- : int list = [1; 2; 3; 4; 5]
# [] @ [1; 2; 3];;
- : int list = [1; 2; 3]
```

- List reversal :

```
# List.rev [1; 2; 3; 4; 5];;
- : int list = [5; 4; 3; 2; 1]
# List.rev_append [1; 2; 3] [4; 5];;
- : int list = [3; 2; 1; 4; 5]
```

- Flattening lists :

```
# List.concat [[1; 2]; []; [3; 4; 5]; [6]];;
- : int list = [1; 2; 3; 4; 5; 6]
```

- Iterating on lists :

```
# let out x = print_int (x + 1);;
val out : int -> unit = <fun>
# List.iter out [1; 2; 3; 4; 5; 6];;
234567- : unit = ()
# let out2 x y = print_int (x - y);;
val out2 : int -> int -> unit = <fun>
#  List.iter2 out2 [10; 9; 8] [1; 3; 5];;
963- : unit = ()
```

- Elementwise function application (map) :

```
# let incr x = x+1;;
val incr : int -> int = <fun>
# List.map incr [1; 2; 3; 4; 5; 6];;
- : int list = [2; 3; 4; 5; 6; 7]
# List.rev_map incr [1; 2; 3; 4; 5; 6];;
- : int list = [7; 6; 5; 4; 3; 2]
# (-);;
- : int -> int -> int = <fun>
# List.map2 (-) [10; 9; 8] [1; 3; 5];;
- : int list = [9; 6; 3]
# List.rev_map2 (-) [10; 9; 8] [1; 3; 5];;
- : int list = [3; 6; 9]
```

$$\text{map } f\ [x_1; \ldots; x_n] = [(f\ x_1); \ldots (f\ x_n)]$$

- Folding lists :

$$\text{fold\_left } f\ a\ [x_1; \ldots; x_n] = f(\ldots(f(f\ a\ x_1)\ x_2)\ldots\ x_n$$

$$\text{fold\_right } f\ [x_1, \ldots, x_n]\ a = f(x_1, f(x_2, \ldots f(x_n, a)\ldots))$$

```
# (+);;
- : int -> int -> int = <fun>
# List.fold_left (+) 0 [1; 2; 3; 4; 5; 6];;
- : int = 21
# List.fold_left (-) 0 [1; 2; 3; 4; 5; 6];;
- : int = -21
# List.fold_right (-) [1; 2; 3; 4; 5; 6] 0;;
- : int = -3
```

Set-theoretic operations on lists:

```
# let pos x = x >= 0;;
val pos : int -> bool = <fun>
# List.for_all pos [1; 2; 3];;
- : bool = true
# List.for_all pos [1; -2; 3];;
# List.exists pos [-1; 2; -3];;
- : bool = true
# List.exists pos [-1; -2; -3];;
- : bool = false

# List.for_all2 (>=) [1; 2; 3] [0; 1; 2];;
- : bool = true
# List.for_all2 (>=) [1; 2; 3] [0; 1];;
Exception: Invalid_argument "List.for_all2".
# List.exists2 (>=) [1; 2; 3] [3; 4; 5];;
- : bool = false

# List.mem 0 [1; 2; 3];;
- : bool = false
# List.mem 1 [1; 2; 3];;
- : bool = true
```

```
# pos;;
- : int -> bool = <fun>
# List.find pos [-1; -2; 3; 4];;
- : int = 3
# List.filter pos [-1; -2; 3; 4];;
- : int list = [3; 4]
# List.partition pos [-1; -2; 3; 4];;
- : int list * int list = ([3; 4], [-1; -2])
```

Association lists (to represent tables as lists of pairs (entry, value)):

```
# List.assoc 2 [0,"a"; 1,"b"; 2,"c"; 3,"d"];;
- : string = "c"
# List.assoc 5 [0,"a"; 1,"b"; 2,"c"; 3,"d"];;
Exception: Not_found.
# List.mem_assoc 2 [0,"a"; 1,"b"; 2,"c"; 3,"d"];;
- : bool = true
# List.mem_assoc 5 [0,"a"; 1,"b"; 2,"c"; 3,"d"];;
- : bool = false
```

```
# List.remove_assoc 2 [0,"a"; 1,"b"; 2,"c"; 3,"d"];;
- : (int * string) list = [(0, "a"); (1, "b"); (3, "d")]
# List.remove_assoc 5 [0,"a"; 1,"b"; 2,"c"; 3,"d"];;
- : (int * string) list = [(0, "a"); (1, "b"); (2, "c"); (3, "d")]

# List.split [0,"a"; 1,"b"; 2,"c"; 3,"d"];;
- : int list * string list = ([0; 1; 2; 3], ["a"; "b"; "c"; "d"])
# List.combine [0; 1; 2; 3] ["a"; "b"; "c"; "d"];;
- : (int * string) list = [(0, "a"); (1, "b"); (2, "c"); (3, "d")]
```

Sorting lists :

```
# let cmp x y = if x < y then -1 else if x > y then 1 else 0;;
val cmp : 'a -> 'a -> int = <fun>
# List.sort cmp [1; 2; 3; 5];;
- : int list = [1; 2; 3; 5]
```

Merging sorted lists :

```
# List.merge cmp [1; 3; 5; 6] [2; 4; 7];;
- : int list = [1; 2; 3; 4; 5; 6; 7]
```

Lists

Definition: `[]`, `[ 1; 2; 3 ]`, or `x :: l`
Access:

```
match l with
| [] -> ...
| x :: l -> ...
```

Assignment: a list is immutable.
Iteration: `do_list`, `map`
Functions: `list_length`

The OCAML List library module

Module List

```
module List: sig  end
```

List operations.

Some functions are flagged as not tail-recursive. A tail-recursive function uses constant stack space, while a non-tail-recursive function uses stack space proportional to the length of its list argument, which can be a problem with very long lists. When the function takes several list arguments, an approximate formula giving stack usage (in some unspecified constant unit) is shown in parentheses.

The above considerations can usually be ignored if your lists are not longer than about 10000 elements.

```
val length : 'a list -> int
```

Return the length (number of elements) of the given list.

```
val hd : 'a list -> 'a
```

Return the first element of the given list. Raise `Failure "hd"` if the list is empty.

```
val tl : 'a list -> 'a list
```

Return the given list without its first element. Raise `Failure "tl"` if the list is empty.

```
val nth : 'a list -> int -> 'a
```

Return the n-th element of the given list. The first element (head of the list) is at position 0. Raise `Failure "nth"` if the list is too short.

```
val rev : 'a list -> 'a list
```

List reversal.

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

Catenate two lists. Same function as the infix operator `@`. Not tail-recursive (length of the first argument). The `@` operator is not tail-recursive either.

```
val rev_append : 'a list -> 'a list -> 'a list
```

`List.rev_append l1 l2` reverses `l1` and concatenates it to `l2`. This is equivalent

to `List.rev l1 @ l2`, but `rev_append` is tail-recursive and more efficient.

```
val concat : 'a list list -> 'a list
```

Concatenate a list of lists. The elements of the argument are all concatenated together (in the same order) to give the result. Not tail-recursive (length of the argument + length of the longest sub-list).

```
val flatten : 'a list list -> 'a list
```

Same as `concat`. Not tail-recursive (length of the argument + length of the longest sub-list).

Iterators

```
val iter : ('a -> unit) -> 'a list -> unit
```

`List.iter f [a1; ...; an]` applies function `f` in turn to `a1; ...; an`. It is equivalent to **`begin`** `f a1; f a2; ...; f an; ()` **`end`**.

```
val map : ('a -> 'b) -> 'a list -> 'b list
```

`List.map f [a1; ...; an]` applies function `f` to `a1, ..., an`, and builds the list `[f a1; ...; f an]` with the results returned by `f`. Not tail-recursive.

```
val rev_map : ('a -> 'b) -> 'a list -> 'b list
```

`List.rev_map f l` gives the same result as `List.rev (List.map f l)`, but is tail-recursive and more efficient.

```
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

`List.fold_left f a [b1; ...; bn]` is `f (... (f (f a b1) b2) ...) bn`.

```
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

`List.fold_right f [a1; ...; an] b` is `f a1 (f a2 (... (f an b) ...))`. Not tail-recursive.

Iterators on two lists

```
val iter2 : ('a -> 'b -> unit) -> 'a list -> 'b list -> unit
```

`List.iter2 f [a1; ...; an] [b1; ...; bn]` calls in turn `f a1 b1; ...; f an bn`. Raise `Invalid_argument` if the two lists have different lengths.

```
val map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list
```

`List.map2 f [a1; ...; an] [b1; ...; bn]` is `[f a1 b1; ...; f an bn]`. Raise `Invalid_argument` if the two lists have different lengths. Not tail-recursive.

```
val rev_map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list
```

`List.rev_map2 f l` gives the same result as `List.rev (List.map2 f l)`, but is tail-recursive and more efficient.

```
val fold_left2 : ('a -> 'b -> 'c -> 'a) -> 'a -> 'b list -> 'c list -> 'a
```

`List.fold_left2 f a [b1; ...; bn] [c1; ...; cn]` is `f (... (f (f a b1 c1) b2 c2) ...) bn cn`. Raise `Invalid_argument` if the two lists have different lengths.

```
val fold_right2 : ('a -> 'b -> 'c -> 'c) -> 'a list -> 'b list -> 'c -> 'c
```

`List.fold_right2 f [a1; ...; an] [b1; ...; bn] c` is `f a1 b1 (f a2 b2 (... (f an bn c) ...))`. Raise `Invalid_argument` if the two lists have different lengths. Not tail-recursive.

List scanning

```
val for_all : ('a -> bool) -> 'a list -> bool
```

`for_all p [a1; ...; an]` checks if all elements of the list satisfy the predicate `p`. That is, it returns `(p a1) && (p a2) && ... && (p an)`.

```
val exists : ('a -> bool) -> 'a list -> bool
```

`exists p [a1; ...; an]` checks if at least one element of the list satisfies the predicate `p`. That is, it returns `(p a1) || (p a2) || ... || (p an)`.

```
val for_all2 : ('a -> 'b -> bool) -> 'a list -> 'b list -> bool
```

Same as `List.for_all`, but for a two-argument predicate. Raise `Invalid_argument` if the two lists have different lengths.

```
val exists2 : ('a -> 'b -> bool) -> 'a list -> 'b list -> bool
```

Same as `List.exists`, but for a two-argument predicate. Raise `Invalid_argument` if the two lists have different lengths.

```
val mem : 'a -> 'a list -> bool
```

`mem a l` is true if and only if `a` is equal to an element of `l`.

```
val memq : 'a -> 'a list -> bool
```

Same as `List.mem`, but uses physical equality instead of structural equality to compare list elements.

List searching

```
val find : ('a -> bool) -> 'a list -> 'a
```

`find p l` returns the first element of the list `l` that satisfies the predicate `p`. Raise `Not_found` if there is no value that satisfies `p` in the list `l`.

```
val filter : ('a -> bool) -> 'a list -> 'a list
```

`filter p l` returns all the elements of the list `l` that satisfy the predicate `p`. The order of the elements in the input list is preserved.

```
val find_all : ('a -> bool) -> 'a list -> 'a list
```

`find_all` is another name for `List.filter`.

```
val partition : ('a -> bool) -> 'a list -> 'a list * 'a list
```

`partition p l` returns a pair of lists `(l1, l2)`, where `l1` is the list of all the elements of `l` that satisfy the predicate `p`, and `l2` is the list of all the elements of `l` that do not satisfy `p`. The order of the elements in the input list is preserved.

Association lists

```
val assoc : 'a -> ('a * 'b) list -> 'b
```

`assoc a l` returns the value associated with key `a` in the list of pairs `l`. That is, `assoc a [ ...; (a,b); ...] = b` if `(a,b)` is the leftmost binding of `a` in list `l`. Raise `Not_found` if there is no value associated with `a` in the list `l`.

```
val assq : 'a -> ('a * 'b) list -> 'b
```

Same as `List.assoc`, but uses physical equality instead of structural equality to compare keys.

```
val mem_assoc : 'a -> ('a * 'b) list -> bool
```

Same as `List.assoc`, but simply return true if a binding exists, and false if no bindings exist for the given key.

```
val mem_assq : 'a -> ('a * 'b) list -> bool
```

Same as `List.mem_assoc`, but uses physical equality instead of structural equality to compare keys.

```
val remove_assoc : 'a -> ('a * 'b) list -> ('a * 'b) list
```

`remove_assoc a l` returns the list of pairs `l` without the first pair with key `a`, if any. Not tail-recursive.

```
val remove_assq : 'a -> ('a * 'b) list -> ('a * 'b) list
```

Same as `List.remove_assoc`, but uses physical equality instead of structural equality to compare keys. Not tail-recursive.

Lists of pairs

```
val split : ('a * 'b) list -> 'a list * 'b list
```

Transform a list of pairs into a pair of lists: `split [(a1,b1); ...; (an,bn)]` is `([a1; ...; an], [b1; ...; bn])`. Not tail-recursive.

```
val combine : 'a list -> 'b list -> ('a * 'b) list
```

Transform a pair of lists into a list of pairs: `combine [a1; ...; an] [b1; ...; bn]` is `[(a1,b1); ...; (an,bn)]`. Raise `Invalid_argument` if the two lists have different lengths. Not tail-recursive.

Sorting

```
val sort : ('a -> 'a -> int) -> 'a list -> 'a list
```

Sort a list in increasing order according to a comparison function. The comparison function must return 0 if it arguments compare as equal, a positive integer if the first is greater, and a negative integer if the first is smaller (see Array.sort for a complete specification). For example, `Pervasives.compare` is a suitable comparison function. The resulting list is sorted in increasing order. `List.sort` is guaranteed to run in constant heap space (in addition to the size of the result list) and logarithmic stack space.

The current implementation uses Merge Sort. It runs in constant heap space and logarithmic stack space.

```
val stable_sort : ('a -> 'a -> int) -> 'a list -> 'a list
```

Same as `List.sort`, but the sorting algorithm is guaranteed to be stable (i.e. elements that compare equal are kept in their original order) .

The current implementation uses Merge Sort. It runs in constant heap space and logarithmic stack space.

```
val fast_sort : ('a -> 'a -> int) -> 'a list -> 'a list
```

Same as `List.sort` or `List.stable_sort`, whichever is faster on typical input.

```
val merge : ('a -> 'a -> int) -> 'a list -> 'a list -> 'a list
```

Merge two lists: Assuming that `l1` and `l2` are sorted according to the comparison function `cmp`, `merge cmp l1 l2` will return a sorted list containting all the elements of `l1` and `l2`. If several elements compare equal, the elements of `l1` will be before the elements of `l2`. Not tail-recursive (sum of the lenghts of the arguments).

Variant / Sum types in OCAML

- Because of the automatic type inference algorithm used by the OCAML compiler, it is impossible to have objects of variant types such as "either boolean or integer".

- To handle such variants, one must add a tag, which must be tested at execution to check to which case a data of the variant type does correspond;

- Example:

```
# type boolorint = BOOL of bool | INT of int;;
type boolorint = BOOL of bool | INT of int
# let x = BOOL true;;
val x : boolorint = BOOL true
# let y = INT 10;;
val y : boolorint = INT 10
# let print_boolorint = function
   | BOOL b -> if b then print_string "true"
               else print_string "false";
   | INT i -> print_int i;;
val print_boolorint : boolorint -> unit = <fun>
# print_boolorint x;;
true- : unit = ()
# print_boolorint y;;
10- : unit = ()
#
```

- Sum types can also be used to record cases (a generalization of the two-cases booleans):

```
# type case = A | B | C;;
type case = A | B | C
# let current_state = A;;
val current_state : case = A
```

Sum type:

```
type name =
  | Constant_constructor
  | Constructor_with_argument of expression_de_type;;
```

Records in OCAML

Records that is structures with named fields can be defined in OCAML. By default a field is immutable, unless declared mutable. In this case, new values can be assigned to the field.

```
# type record1 = {b:bool; mutable i:int};;
type record1 = { b : bool; mutable i : int; }
# let x = {b = true; i = 5};;
val x : record1 = {b = true; i = 5}
# x;;
- : record1 = {b = true; i = 5}
# x.i <- -10;;
- : unit = ()
# x;;
- : record1 = {b = true; i = -10}
#

# let add_record1 r1 r2 =
  { b = r1.b || r2.b; i = r1.i + r2.i};;
val add_record1 : record1 -> record1 -> record1 = <fun>
# let y = {b = false; i = 5};;
val y : record1 = {b = false; i = 5}
# x;;
- : record1 = {b = true; i = -10}
# add_record1 x y;;
- : record1 = {b = true; i = -5}
#
```

Record type:

type name = {label1 : type_of_the_field; label2 : type_of_the_field};;

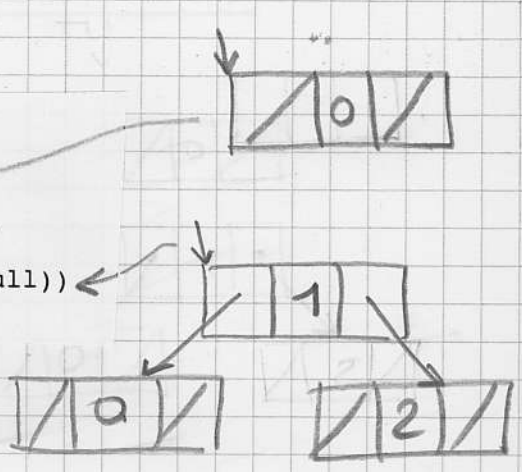
A field may be mutable, if declared as: **mutable** label : type_of_the_mutable_field

Assignment: record.label <- new_value

Recursive types in OCAML

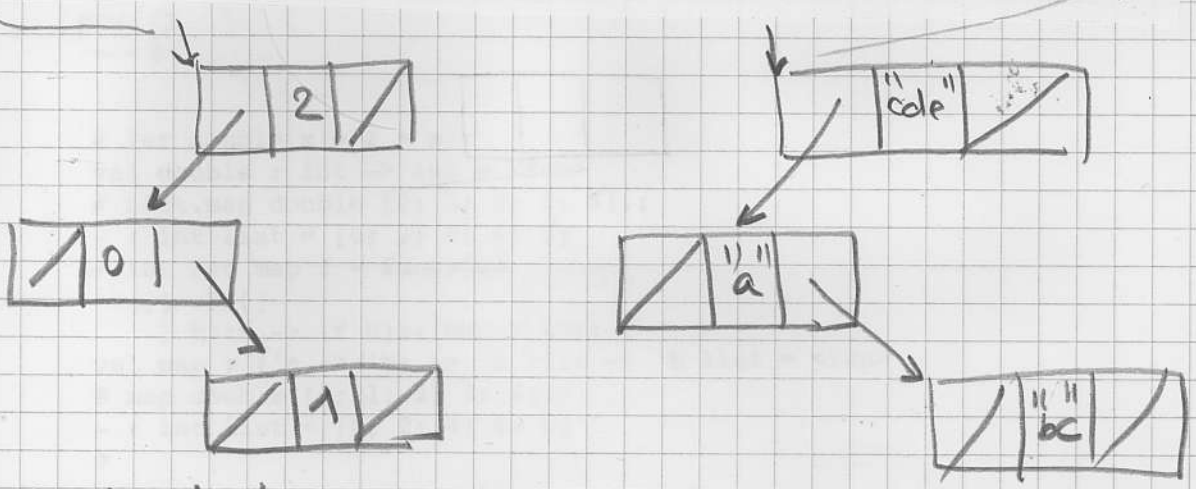
- One can declare recursive types in OCAML (to built list, trees, etc...):

```
# type inttree = Null | Rec of (int * inttree * inttree);;
type inttree = Null | Rec of (int * inttree * inttree)
# Rec (0, Null, Null);;
- : inttree = Rec (0, Null, Null)
# Rec (1, (Rec (0, Null, Null)), (Rec (2, Null, Null)));;
- : inttree = Rec (1, Rec (0, Null, Null), Rec (2, Null, Null))
#
```



- The recursive type definition can be polymorphic:

```
# type 'a btree = Empty | Node of 'a * 'a btree * 'a btree;;
type 'a btree = Empty | Node of 'a * 'a btree * 'a btree
# let rec member x btree =
   match btree with
         Empty -> false
   | Node(y, left, right) ->
           if x = y then true else
       if x < y then memb  er x left else member x right;;
val member : 'a -> 'a btree -> bool = <fun>
# let rec insert x btree =
   match btree with
         Empty -> Node(x, Empty, Empty)
   | Node(y, left,   right) ->
       if x <= y then Node(y, insert x l  eft, right)
                 else Node(y, left, in  sert x right);;
val insert : 'a -> 'a btree -> 'a btree = <fun>
# insert 1 (insert 0 (insert 2 Empty));;
- : int btree = Node (2, Node (0, Empty, Node (1, Empty, Empty)), Empty)
# insert "bc" (insert "a" (insert "cde" Empty));;
- : string btree =
Node ("cde", Node ("a", Empty, Node ("bc", Empty, Empty)), Empty)
#
```



Ordered trees -

Examples of simple programs in OCAML

Examples of list manipulation in OCAML

- appending two lists :

```
# List.append [0; 1; 2] [3; 4];;
- : int list = [0; 1; 2; 3; 4]
# let rec append x y = match x with
    []   -> y
  | h::t -> h::(append t y);;
val append : 'a list -> 'a list -> 'a list = <fun>
# append [0; 1; 2] [3; 4];;
- : int list = [0; 1; 2; 3; 4]
```

- Reversing a list :

```
# List.rev [0; 1; 2; 3; 4];;
- : int list = [4; 3; 2; 1; 0]
# let rec rev = function
   [] -> []
   | h::t -> append (rev t) [h];;
val rev : 'a list -> 'a list = <fun>
# rev [4; 3; 2; 1; 0];;
- : int list = [0; 1; 2; 3; 4]
#
```

A more efficient (tail-recursive) version :

```
# let rev l =
    let rec reverse x y = match x with
      [] -> y
    | h::t -> reverse t (h::y)
  in reverse l [];;
val rev : 'a list -> 'a list = <fun>
# rev [0; 1; 2; 3; 4];;
- : int list = [4; 3; 2; 1; 0]
#
```

- map :

```
# let double x = 2 * x;;
val double : int -> int = <fun>
# List.map double [0; 1; 2; 3; 4];;
- : int list = [0; 2; 4; 6; 8]
# let rec map f = function
   [] -> []
    | h::t -> (f h)::(map f t);;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
# map double [0; 1; 2; 3; 4];;
- : int list = [0; 2; 4; 6; 8]
#
```

- elementwise list squaring

```
# let square l = let sq x = x * x in map sq l;;
val square : int list -> int list = <fun>
# square [0; 2; 4; 6; 8];;
- : int list = [0; 4; 16; 36; 64]
#
```

- generating a list:

```
# let rec generate f p x =
  if not (p x) then [] else x :: generate f p (f x);;
val generate : ('a -> 'a) -> ('a -> bool) -> 'a -> 'a list = <fun>
# let succ x = x + 1;;
val succ : int -> int = <fun>
# let interval m n =
  let p x = (x <= n) in
  generate succ p m;;
val interval : int -> int -> int list = <fun>
# interval 5 10;;
- : int list = [5; 6; 7; 8; 9; 10]
# interval 5 0;;
- : int list = []
#
```

List reduction, with a correctness proof.

- Definition :

```
# let rec  reduce f a = function
    []    -> a
  | h::t -> reduce f  (f h a) t;;
reduce : ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b = <fun>
```

---

reduce f a $[x_1;\ x_2;\ \ldots,\ x_n] = f\ x_1\ (f\ x_2\ (\ldots(f\ x_n\ a)\ldots))$.

- Correctness proof :

reduce f a' []
= a' — par définition

reduce f a' $[x_1;\ \ldots;\ x_n]$
= (f x_n (f x_{n-1}(...(f x_1 a')...)) — hyp. ind.

reduce f a $[x_0;\ \ldots;\ x_n]$
= reduce f (f x_0 a) $[x_1;\ \ldots;\ x_n]$ — par définition
= (f x_n (f x_{n-1}(...(f x_1 (f x_0 a))...)) — par hyp. ind.

C.Q.F.D. par induction sur la longueur de la liste.

- Example :

```
# let rec reduce f a = function
  [] -> a
  | h::t -> reduce f (f h a) t;;
val reduce : ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b = <fun>
# let sigma = reduce (+) 0;;
val sigma : int list -> int = <fun>
# sigma [1; 2; 3; 4; 5];;
- : int = 15
# let prod = let f x y = x * y in reduce f 1;;
val prod : int list -> int = <fun>
# prod [1; 2; 3; 4; 5];;
- : int = 120
#

# let o f g x = (f (g x));;
val o : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
# let fact = o prod (interval 2);;
val fact : int -> int = <fun>
# fact 7;;
- : int = 5040
```

A sorting example (Quick Sort), with a proof sketch

Partition :

```
# let rec reduce f a = function
  [] -> a
  | h::t -    > reduce f (f h a) t;;
val reduce : ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b = <fun>
# let partition p =
  let aiguiller x (pos, neg) =
    if p x then (x::pos), neg else pos, (x::neg)
  in reduce aiguiller ([], []);;
val partition : ('a -> bool) -> 'a list -> 'a list * 'a list = <fun>
# let p x = x <= 5 in
  partition p [1; 10; 2; 9; 3; 8; 4; 7; 5; 6];;
- : int list * int list = ([5; 4; 3; 2; 1], [6; 7; 8; 9; 10])
```

QuickSort :

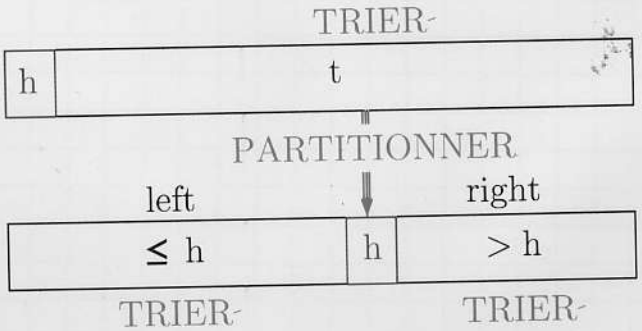
```
# let quicksort le l =
    let rec sort = function
      [] -> []
    | h::t ->
        let p x = (le x h) in
          let (left, right) = partition p t in
            append (sort left) (h :: (sort right))
  in sort l;;
```

Example :

```
val quicksort : ('a -> 'a -> bool) -> 'a list -> 'a list = <fun>
# quicksort (<) [1; 10; 2; 9; 3; 8; 4; 7; 5; 6];;
- : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
```

Proof sketch :

QUICKSORT
3 — PRINCIPE DU TRI




```
# trier h::t =
#   let (left, right) = partition le(x, h) t in
#    append (trier left) (h :: (trier  right));;
```

Type inference in ML

The basic ML typing rules for constants, examples

The OCAML compiler / interpreter automatically infer types using a generalization of the Hindley / Milner / Damas type inference algorithm. The type which is infered using polymorphic types for **let** and monomorphic types for **let rec** satisfy the typing rules given below.

- Let us start with simple monotypes:

$$t \in M ::= \text{unit} \mid \text{bool} \mid \text{int} \mid \text{float} \mid \ldots \mid M_1 \to M_2$$

The type $M_1 \to M_2$ is the type of functions mapping a single argument of type M_1 to a result of type M_2.

Other type constructors (such as pair $M_1 * M_2$) are ignored for simplicity.

- In order to determine the type of an ML expression such as $x+1$ it is necessary to know the type of global variables like x. Therefore we introduce type environments Γ, mapping variables x to their type $\Gamma(x)$.

- If x is a global integer variable then $\Gamma(x) = int$
- If f is a global function mapping integers to integers then $\Gamma(f) = int \to int$
- etc.

- A typing $\Gamma \vdash e : M$ states that in the type environment Γ, expression e has type M.

for example

$$\Gamma \vdash () : unit \quad (CONST_{()})$$

$$\Gamma \vdash true : bool \quad (CONST_{true})$$

$$\Gamma \vdash false : bool \quad (CONST_{false})$$

$$\Gamma \vdash digit^+ : int \quad (CONST_{int})$$

$$\Gamma \vdash [\pm]\, digit^+ . [digit^+][e\,[\pm]\, digit^+] : float \quad (CONST_{float})$$

$$\Gamma \vdash "char*" : string \quad (CONST_{string})$$

- Then there are rules to infer the type of a complex expression in term of its components

$$\frac{\Gamma \vdash e_1 : bool, \quad \Gamma \vdash e_2 : bool}{\Gamma \vdash e_1 \,\&\&\, e_2 : bool}$$

$$\frac{\Gamma \vdash e_1 : int, \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 + e_2 : int}$$

$$\frac{\Gamma \vdash e_1 : \text{float}, \quad \Gamma \vdash e_2 : \text{float}}{\Gamma \vdash e_1 +. e_2 : \text{float}}$$

$$\frac{\Gamma \vdash e_1 : \text{string}}{\Gamma \vdash \text{int\_of\_string } e_1 : \text{int}}$$

In fact, all these basic operations have types given in the Library :

```
# (&&);;
- : bool -> bool -> bool = <fun>
# (+);;
- : int -> int -> int = <fun>
# (+.);;
- : float -> float -> float = <fun>
# int_of_string;;
- : string -> int = <fun>
```

so we can imagine that Γ is initialized with all such declarations so as to use the general rules:

- $$\frac{\Gamma(op) = M}{\Gamma \vdash op : M} \quad (OP)$$

- $$\frac{\Gamma \vdash op : M_1 \to M_2, \quad \Gamma \vdash op : M_1}{\Gamma \vdash op\ e : M_2} \quad (OP1)$$

- $$\frac{\Gamma \vdash op : M_1 \to M_2 \to M_3, \quad \Gamma \vdash e_1 : M_1, \quad \Gamma \vdash e_2 : M_2}{\Gamma \vdash e_1\ op\ e_2 : M_3} \quad (OP2)$$

- $$\frac{\Gamma \vdash op : M_1 \to M_2 \to \dots M_n \to M_{n+1}, \quad \Gamma \vdash e_1 : M_1, \dots, \Gamma \vdash e_n : M_n}{\Gamma \vdash op\ e_1\ e_2 \dots e_n : M_{n+1}} \quad (OPn)$$

Types with variables

Some operations can work on objects of different types, including user defined functions (recall that the lambda-expression $\lambda x.e$ is written function x -> e in OCAML, and is called an anonymous function, which can be given a name, as in let f x = e):

```
# List.map;;
- : ('a -> 'b) -> 'a list -> 'b list = <fun>
# Array.length;;
- : 'a array -> int = <fun>
# function x -> x;;
- : 'a -> 'a = <fun>
# function o -> function f -> function g -> function x -> (f (g x));;
- : 'a -> ('b -> 'c) -> ('d -> 'b) -> 'd -> 'c = <fun>
```

For example the identity function $x \to x$ that is $\lambda x.x$ has type $\alpha \to \alpha$ (where α is a type variable, written 'a, 'b, ... by the compiler) which is a shorthand for infinitely many types unit -> unit, bool -> bool, ..., (int -> int) -> (int -> int), etc. Such types are polymorphic types (where the quantifier indicates which variables can be instanciated e.g. $\forall\alpha.\alpha \to \alpha$):

Types monomorphes avec variables (τ) :

$\tau \in T ::=$	num	constantes de type
	bool	
	...	
	$\alpha, \alpha_1, \ldots, \alpha_n, \ldots$	variables de type
	τ_1 -> τ_2	fonctions

Types polymorphes (π) :

$\pi \in P ::=$	τ	
	$\forall\alpha.\pi$	schéma de type

(In the compiler output, the $\forall$ is omitted and α is 'a, ...).

Instantiation rule

If an object (like $\lambda x.x$) is polymorphic (e.g. of type $\forall\alpha.\alpha \to \alpha$), this means that it has infinitely many types (e.g. $bool \to bool$, $int \to int$, $(float \to float) \to (float \to float)$, ...). These types are obtained by instantiation, that is the replacement of a type variable in a polytype by a monotype (like $\alpha \to \alpha\,[\alpha := int] = int \to int$). The instantiation rule states that if an object has the polytype $\forall\alpha.\pi$, it can be considered of type $\pi[\alpha := \tau]$ for any monotype τ :

– Un type polymorphe peut avoir des instances monomorphes multiples différentes, d'où la règle d'*instantiation* :

$$\frac{\Gamma \vdash e : \forall\alpha.\pi}{\Gamma \vdash e : \pi[\alpha := \tau]} \qquad (\mathrm{INST}_p)$$

where

$$\begin{aligned}
unit\,[\alpha := \tau] &= unit \\
bool\,[\alpha := \tau] &= bool \\
&\cdots \\
\alpha\,[\alpha := \tau] &= \tau \\
\beta\,[\alpha := \tau] &= \beta \quad \text{when } \beta \neq \alpha \\
\tau_1 \to \tau_2\,[\alpha := \tau] &= \tau_1[\alpha := \tau] \to \tau_2[\alpha := \tau]
\end{aligned}$$

Typing of conditionals, example

In a conditional if b then e_1 else e_2, b should be boolean and e_1 and e_2 should have the same type:

```
# if true then 0 else 1/0;;
- : int = 0
# if true then 0 else 0.;;
This expression has type float but is here used with type int
#
```

The corresponding rule is the following:

$$\frac{\Gamma \vdash e_1 : \text{bool} \wedge \Gamma \vdash e_2 : \tau \wedge \Gamma \vdash e_3 : \tau}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau} \quad (\text{COND}_p)$$

for example

(1) $\Gamma \vdash$ true : bool

(2) $\Gamma \vdash 0$: int

(3) $\Gamma \vdash 1$: int

(4) $\Gamma \vdash /$: int → int → int

(5) $\Gamma \vdash 1/0$: int (2) (3) (4) (OP_2)

(6) $\Gamma \vdash$ if true then 0 else 1/0 : int (1) (2) (5) (COND)

Typing of (anonymous) functions and applications (calls)

- Abstraction (anonymous function function $x \to e$) :

$$\frac{\Gamma[x := \tau] \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \to \tau'} \quad (\mathrm{ABS}_p)$$

where

$$\Gamma[x := \tau](x) = \tau$$

$$\Gamma[x : \tau](y) = \Gamma(y) \qquad y \neq x$$

- Variable : In the typing of e, there might be uses of x in the function body e, which has type $\Gamma[x := \tau](x) = \tau$. This can be derived by the rule :

$$\Gamma[x := \pi] \vdash x : \pi \quad (\mathrm{VAR}_p)$$

- Application (of a function to an argument) :

$$\frac{\Gamma \vdash e_1 : \tau \to \tau' \ \wedge \ \Gamma \vdash e_2 : \tau}{\Gamma \vdash (e_1\ e_2) : \tau'} \quad (\mathrm{APP}_p)$$

Example of function and application typing

We have:

```
% ocaml
        Objective Caml version 3.06

# (function x -> x) 1;;
- : int = 1
# ^D
%
```

This can be derived by the typing rules as follows :

(1) $\Gamma[x := \alpha] \vdash x : \alpha$ (VAR_p)

(2) $\Gamma \vdash \lambda x.x : \alpha \to \alpha$ (1) (ABS_p)

(3) $\Gamma \vdash \lambda x.x : (\alpha \to \alpha)[\alpha := int]$ (2), ($INST_p$)

and more explanations on polymorphism for the $\forall$, to come later

(4) $\Gamma \vdash \lambda x.x : int \to int$ (3), def. substitution

(5) $\Gamma \vdash 1 : int$

(6) $\Gamma \vdash (\lambda x.x\ 1) : int$ (4), (5), (APP_p)

which is valid in any typing context Γ.

Typing the let construct, example

In order to type let $x = e_1$ in e_2, the type of x is that of e_1 which is also the type of x is e_2. The typing rule is therefore:

$$\frac{\Gamma \vdash e_1 : \pi \ \wedge \ \Gamma[x := \pi] \vdash e_2 : \tau}{\Gamma \vdash \texttt{let } x = e_1 \texttt{ in } e_2 : \tau} \quad (\mathrm{LET}_p)$$

for example:

```
# let x = 1 in 1;;
- : int = 1
#
```

is typed as follows:

(1) $\Gamma \vdash 1 : int$ $(CONST_1)$

(2) $\Gamma[x := int] \vdash x : int$ (VAR_p)

(3) $\Gamma \vdash$ let $x = 1$ in $x : int$ $(1), (2), (LET_p)$

An x is e_1 is a global one, as shown below:

```
# let x = 1;;
val x : int = 1
# let x = x in x;;
- : int = 1
```

This is typed as follows

(4) $\Gamma[x : int] \vdash x : int$ (Var_p)

(5) $(\Gamma[x : int])[x : int] \vdash x : int$ (Var_p)

(6) $\Gamma[x : int] \vdash$ let $x = x$ in $x : int$ $(LET)_p$

so that now the typing requires the global variable x to be of type int.

Polymorphism

An function f has a polymorphic type if it can take parameters of different monotypes. A classical example is the composition o of functions (written in prefix form):

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

# let succ x = x + 1;;
val succ : int -> int = <fun>
# o succ succ;;
- : int -> int = <fun>
# (o succ succ) 0;;
- : int = 2

# not;;
- : bool -> bool = <fun>
# o not not;;
- : bool -> bool = <fun>
# (o not not) true;;
- : bool = true

# let inv x = 1. /. x;;
val inv : float -> float = <fun>
# (o inv inv);;
- : float -> float = <fun>
# (o inv inv) 3.;;
- : float = 3.
#
```

Polymorphism is essential to write reusable code. The cost is that of a (hidden) pointer, which would have to be explicitely manipulated in C, but in ML this can be done without loosing type safety.

formally, the type of o is

$$\forall\alpha.\forall\beta.\forall\gamma : (\alpha \to \beta) \to (\gamma \to \alpha) \to (\gamma \to \alpha)$$

meaning that α, β, γ can be instantiated with typing rule (INST_P).

Généralisation

Since (INST_p) handled polymorphic types $\forall \alpha.\tau$, we need a rule introducing the variable α which can be instanciated later. This is the so-called "generalization rule":

$$\frac{\Gamma \vdash e : \pi \ \wedge \ \alpha \notin \text{FV}(\Gamma)}{\Gamma \vdash e : \forall \alpha.\pi} \qquad (\text{GEN}_p)$$

– En combinant les règles de généralisation (GEN_p) et instantiation (INST_p), on peut, au cas où $\alpha \in \text{FV}(\pi)$ et $\alpha \notin \text{FV}(\Gamma)$, instancier :

$$\Gamma \vdash e : \pi$$

en :

$$\Gamma \vdash e : \pi[\alpha := \tau]$$

Observe that the rule has a restriction $\alpha \notin FV(\Gamma)$, that is α should not be a free variable in the environment Γ. This is defined as follow:

—

$$FV(\emptyset) = \emptyset \qquad \text{empty environment}$$
$$FV(\Gamma[x := \Pi]) = FV(\Gamma) \cup FV(\Pi)$$
$$FV(\text{int}) = \emptyset$$
$$\cdots$$
$$FV(\alpha) = \{\alpha\}$$
$$FV(\tau_1 \to \tau_2) = FV(\tau_1) \cup FV(\tau_2)$$
$$FV(\forall \alpha.\Pi) = FV(\Pi) \setminus \{\alpha\}$$

—

Example of correct generalizations:

(a) $\{(x : \alpha)\} \vdash 1 : \texttt{int}$ — constante

(b) $\emptyset \vdash \lambda x.1 : \alpha \texttt{ -> int}$ — (a), abstraction

(c) $\emptyset \vdash \lambda x.1 : \forall\alpha.\alpha \texttt{ -> int}$ — (b), généralisation

qui signifie, par la règle d'instantiation :

$$\emptyset \vdash \lambda x.1 : \texttt{bool -> int}$$
$$\emptyset \vdash \lambda x.1 : \texttt{int -> int}$$
$$\ldots$$

Observe that the judgement:

$$\{(x : \alpha)\} \vdash 1 : \text{int}$$

stands for:

$$\{(x : \text{unit})\} \vdash 1 : \text{int}$$
$$\{(x : \text{bool})\} \vdash 1 : \text{int}$$
$$\ldots$$
$$\{(x : \text{int} \to \text{int})\} \vdash 1 : \text{int}$$
$$\ldots$$

and that the type of the result does not depend on the type α of x.

Example of forbidden generalization

- In the type judgement

$$\underbrace{\{(x:\alpha)\}}_{\Gamma} \vdash x : \alpha$$

we have $\alpha \in FV(\Gamma)$ so the generalization rule cannot be applied, since this would be incorrect, as shown below:

– La variable libre α dans l'environnement :

$$\{(x:\alpha)\} \vdash x : \alpha$$

signifie :

$$\{(x:\texttt{int})\} \vdash x : \texttt{int}$$
$$\{(x:\texttt{bool})\} \vdash x : \texttt{bool}$$
$$\ldots$$

– La règle de généralisation est inapplicable. Contre exemple :

$\{(x:\alpha)\} \vdash x : \alpha$	variable
$\{(x:\alpha)\} \vdash x : \forall\alpha.\alpha$	généralisation incorrecte
$\{(x:\alpha)\} \vdash x : \texttt{int}$	instantiation

qui conduit à la conclusion incorrecte, pour $\alpha = \texttt{bool}$:

$$\{(x:\texttt{bool})\} \vdash x : \texttt{int}$$

Example of polymorphic typing

$$\texttt{let } x = \lambda y.y \texttt{ in } (x\ x)$$

$$\{(x : \forall\alpha.(\alpha \texttt{->} \alpha)),(y : \alpha)\} \vdash y : \alpha \qquad (\mathrm{VAR}_p)$$

$$\{(x : \forall\alpha.(\alpha \texttt{->} \alpha))\} \vdash \lambda y.y : \alpha \texttt{->} \alpha \qquad (\mathrm{ABS}_p)$$

$$\{(x : \forall\alpha.(\alpha \texttt{->} \alpha))\} \vdash \lambda y.y : \forall\alpha.(\alpha \texttt{->} \alpha) \qquad (\mathrm{GEN}_p)$$

$$\{(x : \forall\alpha.(\alpha \texttt{->} \alpha))\} \vdash x : \forall\alpha.(\alpha \texttt{->} \alpha) \qquad (\mathrm{VAR}_p)$$

$$\{(x : \forall\alpha.(\alpha \texttt{->} \alpha))\} \vdash x : \alpha \texttt{->} \alpha \qquad (\mathrm{INST}_p)$$

$$\{(x : \forall\alpha.(\alpha \texttt{->} \alpha))\} \vdash x : (\alpha \texttt{->} \alpha) \texttt{->} (\alpha \texttt{->} \alpha) \qquad (\mathrm{INST}_p)$$

$$\{(x : \forall\alpha.(\alpha \texttt{->} \alpha))\} \vdash (x\ x) : \alpha \texttt{->} \alpha \qquad (\mathrm{APP}_p)$$

$$\emptyset \vdash \texttt{let } x = \lambda y.y \texttt{ in } (x\ x) : \alpha \texttt{->} \alpha \qquad (\mathrm{LET}_p)$$

Recursion is not polymorphic in ML.

In ML, we have seen that a **let** can be polymorphic: The same way, a **let rec** $x = e_1$ **in** e_2 can have a polymorphic type for x in the utilization part e_2. However, x cannot have a polytype in e_1. This is shown by the following example:

```
# let z = function x -> 0 in
  let t = function x -> true in
  let rec o = function f -> function g -> function x -> (f (g x))
  in (((o z) t) ((o t) z));;
- : int = 0

# let z = function x -> 0 in
  let t = function x -> true in
  let rec o = function f -> function g -> function x -> (f (g x))
  in (((o t) z) ((o z) t));;
- : bool = true

# let rec z = function x -> 0
  and t = function x -> true
  and o = function f -> function g -> function x -> (f (g x))
  and a = (((o z) t) ((o t) z))
  and b = (((o t) z) ((o z) t))
  in ();;
This expression has type bool -> bool but is here used with type bool -> int
#
```

This limitation comes from the type inference algorithm which is unable to handle recursive functions with potentially infinite recursive calls which can be each with a different parameter type or yield infinite types, as in:

```
# let rec f = function x -> (x f) in f;;
This expression has type ('a -> 'b) -> 'c but is here used with type 'a
```

which type α should satisfy

$$\alpha = \alpha \to \beta$$

so that $\alpha = (((((\ldots \to \beta) \to \beta) \to \beta) \to \beta) \to \beta)$.

ML typing rule for the Let rec, example

$$\frac{\begin{array}{c}\Gamma[x := \tau'] \vdash e_1 : \tau' \wedge \\ \pi' = \forall\alpha.\forall\beta.\forall\gamma.\cdots.\tau' \wedge \alpha,\beta,\gamma,\ldots \notin \mathrm{FV}(\Gamma) \wedge \\ \Gamma[x := \pi'] \vdash e_2 : \tau\end{array}}{\Gamma \vdash \texttt{let rec } x = e_1 \texttt{ in } e_2 : \tau}\ (\mathrm{REC}_p)$$ [30]

- x peut apparaître dans e_1, mais avec un type monomorphe à instance unique (récursivité possible) ;
- x peut apparaître avec un type polymorphe dans e_2 (pas de récursivité).

- So the type is polymorphe in absence of recursivity but is monomorphe in case of potential recursivity.

- Example : Assume $\alpha \notin FV(\Gamma)$ (e.g. $\Gamma = \emptyset$ is empty)

(1) $\Gamma[f := \alpha \to int][x := \alpha] \vdash x : \alpha$ (VAR)

(2) $\Gamma[f := \alpha \to int][x := \alpha] \vdash f : \alpha \to int$ (VAR)

(3) $\Gamma[f := \alpha \to int][x := \alpha] \vdash (f\ x) : int$ (1), (2), (APP)

(4) $\Gamma[f := \alpha \to int][x := \alpha] \vdash 1 : int$ ($CONST_1$)

(5) $\Gamma[f := \alpha \to int][x := \alpha] \vdash + : int \to int \to int$ (OP_+)

(6) $\Gamma[f := \alpha \to int][x := \alpha] \vdash 1 + (f\ x) : int$ (3)(4)(5)(OP_2)

(7) $\Gamma[f := \alpha \to int] \vdash \lambda x.1 + (f\ x) : \alpha \to int$ (6) (ABS)

(8) $\Gamma \vdash$ Let rec $f = \lambda x.1 + (f\ x)$ in $f : \alpha \to int$ (7)(2)(REC)

(9) $\Gamma \vdash$ Let rec $f = \lambda x.1 + (f\ x)$ in $f : \forall\alpha.\alpha \to int$ (8)(GEN).

as found by the compiler :

```
# let rec f = function x -> 1 + (f x) in f;;
- : 'a -> int = <fun>
```

30. La règle donnée par J. TIURYN (Type Inference Problems: A Survey, *MFCS 1990*, B. Rovan (ed.), LNCS 452, Springer-Verlag, Berlin) est fausse.

A note on the typing environment in type rules

Constitution de l'environnement Γ

L'environnement Γ contient :

- Le type des variables prédéclarées :
  - monomorphes : +: num -> (num -> num)
  - polymorphes : map: $\forall\alpha.\forall\beta.(\alpha \texttt{ -> } \beta) \texttt{ -> } (\alpha \texttt{ list -> } \beta \texttt{ list})$

  $\Rightarrow$ pas de variables de type α, ... libres dans Γ ;
- le type π des variables locales x, qui est :
  - un type polymorphe π pour vérifier le type de e_2 dans `let` $x = e_1$ `in` e_2 ou `let rec` $x = e_1$ `in` e_2 ;

    $\Rightarrow$ pas de variables de type α, ... libres dans Γ [28] ;
  - ou, avec un type monomorphe τ pour vérifier le type de e_1 dans `let rec` $x = e_1$ `in` e_2 ou $\lambda x.e_1$;

    $\Rightarrow$ il peut y avoir des variables de type α, ... libres dans Γ.

Interdiction du polymorphisme en cas de récursivité potentielle

- Conséquence pour les variables de type α, ... figurant dans l'environnement Γ :
  - Si $\alpha \notin \mathrm{FV}(\Gamma)$ alors α n'est pas impliquée dans une définition potentiellement récursive. Dans ce cas α peut avoir des instances multiples différentes (par les règles de généralisation et d'instantiation) ;
  - Si α est impliquée dans une définition potentiellement récursive (`let rec` ou λ) alors $\alpha \in \mathrm{FV}(\Gamma)$. Dans ce cas, α ne peut donc pas avoir d'instances multiples différentes.

28. S'il y en a dans π, elle proviennent d'un `let rec` ou d'un λ englobant et correspondent donc au cas suivant.

An exponential typing example

• Dans le pire des cas, le temps de calcul et l'espace mémoire occupé sont exponentiels en fonction de la taille du programme [27].
Par exemple :

```
1   let pair x y z = z x y;;
2
3   let it_be =
4    let x1 y = pair y y in
5      let x2 y = x1(x1(y)) in
6         let x3 y = x2(x2(y)) in
7            let x4 y = x3(x3(y)) in
8               let x5 y = x4(x4(y)) in
9                  let id z = z in
10                    x5(id);;
```

En pratique, polynomial (en le degré d'imbrication des `let`).

27. H. G. Mairson, *Deciding ML Typability is Complete for Deterministic Exponential Time*, Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California, January 17-19, 1990, pp. 382–401.

1
```
# let it_be =
  let x1 y = pair y y in
  let id z = z in
  x1(id);;
val it_be : (('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b = <fun>
```
2
```
# let it_be =
  let x1 y = pair y y in
  let x2 y = x1(x1(y)) in
  let id z = z in
  x2(id);;
val it_be :
  (((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
   ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c = <fun>
```
3
```
# let it_be =
  let x1 y = pair y y in
  let x2 y = x1(x1(y)) in
  let x3 y = x2(x2(y)) in
  let id z = z in
  x3(id);;
val it_be :
  ((((((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  ((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  '_d) ->
  '_d) ->
  ((((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  ((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  '_d) ->
  '_d) ->
  '_e) ->
  '_e = <fun>
```
4
```
# let it_be =
    let x1 y = pair y y in
  let x2 y = x1(x1(y)) in
  let x3 y = x2(x2(y)) in
      let x4 y = x3(x3(y)) in
  let id z = z in
  x4(id);;
val it_be :
  ((((((((((((((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  ((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  '_d) ->
  '_d) ->
  ((((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  ((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
```



```
  '_d) ->
  '_d) ->
  '_e) ->
  '_e) ->
  ((((((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  ((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  '_d) ->
  '_d) ->
  ((((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  ((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  '_d) ->
  '_d) ->
  '_e) ->
  '_e) ->
  '_f) ->
  '_f) ->
  ((((((((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  ((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  '_d) ->
  '_d) ->
  ((((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  ((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  '_d) ->
  '_d) ->
  '_e) ->
  '_e) ->
  ((((((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  ((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  '_d) ->
  '_d) ->
  ((((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  ((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  '_d) ->
  '_d) ->
  '_e) ->
  '_e) ->
  '_f) ->
```



```
  '_f) ->
  '_g) ->
  '_g) ->
  ((((((((((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  ((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  '_d) ->
  '_d) ->
  ((((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  ((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  '_d) ->
  '_d) ->
  '_e) ->
  '_e) ->
  ((((((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  ((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  '_d) ->
  '_d) ->
  ((((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  ((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  '_d) ->
  '_d) ->
  '_e) ->
  '_e) ->
  '_f) ->
  '_f) ->
  ((((((((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  ((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  '_d) ->
  '_d) ->
  ((((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  ((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  '_d) ->
  '_d) ->
  '_e) ->
  '_e) ->
  ((((((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
```



```
  '_c) ->
  ((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  '_d) ->
  '_d) ->
  ((((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  ((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  '_d) ->
  '_d) ->
  '_e) ->
  '_e) ->
  '_f) ->
  '_f) ->
  '_g) ->
  '_g) ->
  '_h) ->
  '_h) ->
  ((((((((((((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  ((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  '_d) ->
  '_d) ->
  ((((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  ((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  '_d) ->
  '_d) ->
  '_e) ->
  '_e) ->
  ((((((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  ((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  '_d) ->
  '_d) ->
  ((((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  ((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  '_d) ->
  '_d) ->
  '_e) ->
  '_e) ->
  '_f) ->
  '_f) ->
  ((((((((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
```



```
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  ((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  '_d) ->
  '_d) ->
  ((((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  ((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  '_d) ->
  '_d) ->
  '_e) ->
  '_e) ->
  ((((((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  ((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  '_d) ->
  '_d) ->
  ((((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  ((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  '_d) ->
  '_d) ->
  '_e) ->
  '_e) ->
  '_f) ->
  '_f) ->
  '_g) ->
  '_g) ->
  ((((((((((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  ((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  '_d) ->
  '_d) ->
  ((((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  ((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  '_d) ->
  '_d) ->
  '_e) ->
  '_e) ->
  ((((((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  ((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
```



```
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  '_d) ->
  '_d) ->
  ((((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  ((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  '_d) ->
  '_d) ->
  '_e) ->
  '_e) ->
  '_f) ->
  '_f) ->
  ((((((((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  ((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  '_d) ->
  '_d) ->
  ((((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  ((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  '_d) ->
  '_d) ->
  '_e) ->
  '_e) ->
  ((((((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  ((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  '_d) ->
  '_d) ->
  ((((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  ((((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) ->
  ((('_a -> '_a) -> ('_a -> '_a) -> '_b) -> '_b) -> '_c) ->
  '_c) ->
  '_d) ->
  '_d) ->
  '_e) ->
  '_e) ->
  '_f) ->
  '_f) ->
  '_g) ->
  '_g) ->
  '_h) ->
  '_h) ->
  '_i) ->
  '_i = <fun>
```

Separate compilation in OCAML

Modules

A module is a named structure which fields define types, data and operations on data:

```
1  module B_implementation =
2  struct
3    type entier = int
4    let c = ref 1
5    let i () = c := !c + !c ; !c
6    let p v = print_int v; print_newline ()
7  end;;
```

The type of a module is a signature indicating the types of the fields of the structure:

```
1  % ocaml
2                    Objective Caml version 3.06
3
4  # module B_implementation =
5  struct
6    type entier = int
7    let c = ref 1
8    let i () = c := !c + !c ; !c
9                  let p v = print_int v; print_newline ()
10 end;;
11   module B_implementation :
12   sig
13           type entier = int
14           val c : int ref
15           val i : unit -> int
16           val p : int -> unit
17   end
18 # #quit;;
```

The fields of the structure are accessed by its name, followed by a dot and the name of the field:

```
# B_implementation.c;;
- : int ref = {contents = 1}
# i ();;
Unbound value i
# B_implementation.i ();;
- : int = 2
```

The scope of a structure can be opened to access its fields directly:

```
# open B_implementation;;
# i ();;
- : int = 4
# c;;
- : int ref = {contents = 4}
# i ();;
- : int = 8
# p (i ());;
16
- : unit = ()
# p (i ());;
32
- : unit = ()
```

Signatures

A signature specifies which components of a structure are accessible from the outside, and with which type. It can be used to hide some components of a structure (e.g. local function definitions) or export some components with a restricted type:

```
# module type A_signature =
  sig
        type entier
        val i : unit -> entier
        val p : entier -> unit
  end;;
module type A_signature =
  sig type entier val i : unit -> entier val p : entier -> unit end
```

A signature can be used to make a module "abstract" in the sense that some of its fields and its actual representation as a "concrete" type and can b is hidden:

```
# module B = (B_implementation:A_signature);;
module B : A_signature
# B.c;;
Unbound value B.c
# B.i ();;
- : B.entier = <abstr>
# B.p (B.i ());;
128
- : unit = ()
# open B;;
# p (i ());;
256
- : unit = ()
```

One can write either:

```
module B = (B_implementation:A_signature);;
```

or equivalently:

```
module B : A_signature = B_implementation;;
```

Functors

A functor maps structures to structures. More precisely, a functor F parameterized with a formal parameter A (along with the expected signature for A) maps any actual implementation B (B', B'', ...) of the expected signature for A to a structure $F(B)$ ($F(B')$, $F(B'')$, ...).

```
# module type C_signature =
  functor (A: A_signatur  e) ->
                        sig
                                              val i : unit -> A.entier
                                              val p : A      .entier -> unit
                        end;;
  module type C_signature =
  functor (A : A_signature) ->
         sig val i : unit -> A.entier val p : A.entier -> unit end
# module C_implementation =
  functor (A: A_signatur  e) ->
  struct
                        type ent = A.entier
                        let f i = (      )
                        let i () = f (A.i ()); A.i ()
                        let p = A.p
                    end;;
module C_implementation :
  functor (A : A_signature) ->
         sig
               type ent = A.entier
               val f : 'a -> unit
               val i : unit -> A.entier
               val p : A.entier -> unit
         end
# module C = (C_implementation:C_signature);;
module C : C_signature
# module D = C(B);;
module D : sig val i : unit -> B.entier val p : B.entier -> unit end
# D.p (D.i ());;
1024
- : unit = ()
# open D;;
# p (i ());;
4096
- : unit = ()
```

Example modular program

In summary, the modular program is:

```
1
2  module type A_signature =
3    sig
4      type entier
5      val i : unit -> entier
6      val p : entier -> unit
7    end;;
8
9  module B_implementation =
10 struct
11   type entier = int
12   let c = ref 1
13   let i () = c := !c + !c ; !c
14   let p v = print_int v; print_newline ()
15 end;;
16 module B = (B_implementation:A_signature);;
17
18 module type C_signature =
19   functor (A: A_signature) ->
20          sig
21                 val i : unit -> A.entier
22                 val p : A.entier -> unit
23          end;;
24
25 module C_implementation =
26   functor (A: A_signature) ->
27   struct
28          type ent = A.entier
29          let f i = ()
30          let i () = f (A.i ()); A.i ()
31          let p = A.p
32   end;;
33
34 module C = (C_implementation:C_signature);;
35
36 module D = C(B);;
37
38 D.p (D.i ());;
39
```

The program `p.ml` can be interpreted as follows:

```
1  ocaml p.ml
2  4
```

The program `p.ml` can also be compiled and executed as follows:

```
1  % ocamlc p.ml
2  % ./a.out
3  4
```

Separate Compilation

The modules `Name` of the program can be organized into separate files:

- `name.mli`, analogous to the inside of a `sig ... end` construct, for signatures specifying the module interface;
- `name.ml`, analogous to the inside of a `struct ... end` construct, specifying the module interface.

So a pair of files `name.mli` and `name.ml` is equivalent to the definition of a module `Name` (same name as the base name of the two files, with the first letter capitalized) that would be defined as follows:

```
1  module Name: sig (* contents of file name.mli *) end
2             = struct (* contents of file name.ml *) end;;
```

A single file `name.ml` (without corresponding `name.mli`) is equivalent to the following definition:

```
1  module Name = struct (* contents of file name.ml *) end;;
```

The program `p.ml` can be decomposed into the following separate files:

- `a.mli`

```
1  module type A_signature =
2    sig
3           type entier
4           val i : unit -> entier
5           val p : entier -> unit
6    end;;
```

- `b.mli`

```
1  open A;;
2  module B : A_signature;;
```

- b.ml

```
1  open A;;
2  module B_implementation =
3  struct
4    type entier = int
5    let c = ref 1
6    let i () = c := !c + !c ; !c
7    let p v = print_int v; print_newline ()
8  end;;
9  module B = (B_implementation:A_signature);;
```

- c.mli

```
1  open A;;
2  module type C_signature =
3    functor (A: A_signature) ->
4           sig
5                  val i : unit -> A.entier
6                  val p : A.entier -> unit
7           end;;
8  module C : C_signature;;
```

- c.ml

```
1  open A;;
2  module type C_signature =
3    functor (A: A_signature) ->
4           sig
5                  val i : unit -> A.entier
6                  val p : A.entier -> unit
7           end;;
8  module C_implementation =
9    functor (A: A_signature) ->
10   struct
11          type ent = A.entier
12          let f i = ()
13          let i () = f (A.i ()); A.i ()
14          let p = A.p
15   end;;
16 module C = (C_implementation:C_signature)
```

- d.ml

```
1  open B;;
2  open C;;
3  module D = C(B);;
```

- e.ml

```
1 open D;;
2 D.p (D.i ());;
```

The files defining the compilation units can be compiled separately using the `ocamlc -c` command (the `-c` option means "compile only, do not try to link"). The compilation of a `.mli` produces a compiled module interface file `.cmi` while that of a `.mlo` produces a compiled module object file `.cmi`. When all units have been compiled, their `.mlo` files must be linked together using the `ocaml -o exec` command to produce an executable file `exec` (`a.out` by default):

```
1  ocamlc -c a.mli
2  ocamlc -c b.mli
3  ocamlc -c b.ml
4  ocamlc -c c.mli
5  ocamlc -c c.ml
6  ocamlc -c d.ml
7  ocamlc -c e.ml
8  ocamlc b.cmo c.cmo d.cmo e.cmo
9  ./a.out
10 4
```

Equivalence between separately compiled modules and a single program

Notice that only top-level structures can be mapped to separately-compiled files, but not functors nor module types. So the functors or module type signatures have to be included inside a top-level structure defined in a separate file.

For example, the separately compiled files `a.mli`, `b.mli`, `b.ml`, `c.mli`, `c.ml`, `d.ml` and `e.ml` are equivalent to the following single file:

```
1
2  module A = struct
3  module type A_signature =
4    sig
5      type entier
6      val i : unit -> entier
7      val p : entier -> unit
8    end;;
9  end;;
10 module B : sig
11 open A;;
12 module B : A_signature;;
13 end = struct
14 open A;;
15 module B_implementation =
16 struct
17   type entier = int
18   let c = ref 1
19   let i () = c := !c + !c ; !c
20   let p v = print_int v; print_newline ()
21 end;;
22 module B = (B_implementation:A_signature);;
23 end;;
24 module C : sig
25 open A;;
26 module type C_signature =
27   functor (A: A_signature) ->
28          sig
29                 val i : unit -> A.entier
30                 val p : A.entier -> unit
31          end;;
32 module C : C_signature;;
33 end = struct
34 open A;;
35 module type C_signature =
36   functor (A: A_signature) ->
37          sig
38                 val i : unit -> A.entier
39                 val p : A.entier -> unit
40          end;;
```

```
41  module C_implementation =
42    functor (A: A_signature) ->
43    struct
44            type ent = A.entier
45            let f i = ()
46            let i () = f (A.i ()); A.i ()
47            let p = A.p
48    end;;
49  module C = (C_implementation:C_signature)
50  end;;
51  module D = struct
52  open B;;
53  open C;;
54  module D = C(B);;
55  end;;
56  open D;;
57  D.p (D.i ());;
```

Example script

The following script summarizes the OCaml module system.

```
Script started on Tue Jul 29 11:30:48 2003
% ls

a.mli           c.ml            e.ml            p.ml
b.ml            c.mli           makefile        scriptfile.ml
b.mli           d.ml            makefile.depend typescript
```

We show the makefile by parts, and for each part the result of its execution. Makefile:

```
############################ Documentation ######################
#
# To use this Makefile you must create a file makefile.depend,
# using "touch makefile.depend"
#
#
#################################################################

SHELL = /bin/tcsh

.PHONY :all
all : clean original generic compilesep script compile listing interactive


.PHONY : interactive
interactive :
    @echo "*** interactive mode, type:"
    @echo "  ocaml"
    @echo "and then:"
    @echo '  #use "scriptfile.ml";;'
    @echo "  #quit;;"
    @echo 'to use "ocaml" interactively with "scriptfile.ml"'

.PHONY : original
original :
    @echo "*** execution of the original program p.ml:"
    ocaml p.ml
```

Execution:

```
% make

rm: No match.
make: [clean] Error 1 (ignored)
*** execution of the original program p.ml:
ocaml p.ml
4
```

Makefile (see http://caml.inria.fr/FAQ/Makefile_ocaml-eng.html for such a generic makefile for ocaml.)

```
SOURCES = a.mli b.ml b.mli c.ml c.mli d.ml e.ml
OBJS = $(SOURCES:.ml=.cmo)

.PHONY : generic
generic : message $(OBJS)
    ocamlc $(OBJS)
    @echo "*** execution of the compiled code:"
    ./a.out

.PHONY : message
message :
    @echo "*** generic separate compilation of the modules:"

.SUFFIXES:
.SUFFIXES: .ml .mli .cmo .cmi

.ml.cmo:
    ocamlc -c $<

.mli.cmi:
    ocamlc -c $<

makefile.depend: $(SOURCES2)
    $(CAMLDEP) *.mli *.ml > makefile.depend

depend: $(SOURCES2)
    $(CAMLDEP) *.mli *.ml > makefile.depend

include makefile.depend
```

makefile.depend (created automatically)

```
b.cmi: a.cmi
c.cmi: a.cmi
b.cmo: a.cmi b.cmi
b.cmx: a.cmi b.cmi
c.cmo: a.cmi c.cmi
c.cmx: a.cmi c.cmi
d.cmo: b.cmi c.cmi
d.cmx: b.cmx c.cmx
e.cmo: d.cmo
e.cmx: d.cmx
```

Execution :

```
14  *** generic separate compilation of the modules:
15  ocamlc -c a.mli
16  ocamlc -c b.mli
17  ocamlc -c b.ml
18  ocamlc -c c.mli
19  ocamlc -c c.ml
20  ocamlc -c d.ml
21  ocamlc -c e.ml
22  ocamlc a.mli b.cmo b.mli c.cmo c.mli d.cmo e.cmo
23  *** execution of the compiled code:
24  ./a.out
25  4
```

Makefile (showing hand-made separate compilation)

```
.PHONY : compilesep
compilesep :
	@echo "*** customed separate compilation of the modules:"
	ocamlc -c a.mli
	ocamlc -c b.mli
	ocamlc -c b.ml
	ocamlc -c c.mli
	ocamlc -c c.ml
	ocamlc -c d.ml
	ocamlc -c e.ml
	ocamlc b.cmo c.cmo d.cmo e.cmo
	@echo "*** execution of the compiled code:"
	./a.out
```

Execution :

```
26 *** customed separate compilation of the modules:
27 ocamlc -c a.mli
28 ocamlc -c b.mli
29 ocamlc -c b.ml
30 ocamlc -c c.mli
31 ocamlc -c c.ml
32 ocamlc -c d.ml
33 ocamlc -c e.ml
34 ocamlc b.cmo c.cmo d.cmo e.cmo
35 *** execution of the compiled code:
36 ./a.out
37 4
```

Makefile (showing how to generate a single program file corresponding to several separately compiled modules)

```
.PHONY : script
script :
	@echo "*** creation of a script file:"
	@echo "" > scriptfile.ml
	@echo "module A = struct" >> scriptfile.ml
	@cat a.mli >> scriptfile.ml
	@echo "end;;" >> scriptfile.ml
	@echo "module B : sig" >> scriptfile.ml
	@cat b.mli >> scriptfile.ml
	@echo "end = struct" >> scriptfile.ml
	@cat b.ml >> scriptfile.ml
	@echo "end;;" >> scriptfile.ml
	@echo "module C : sig" >> scriptfile.ml
	@cat c.mli >> scriptfile.ml
	@echo "end = struct" >> scriptfile.ml
	@cat c.ml >> scriptfile.ml
	@echo "end;;" >> scriptfile.ml
	@echo "module D = struct" >> scriptfile.ml
	@cat d.ml >> scriptfile.ml
	@echo "end;;" >> scriptfile.ml
	@cat e.ml >> scriptfile.ml
	@echo "*** ocaml in script mode:"
	ocaml scriptfile.ml

.PHONY : compile
compile :
	@echo "*** compilation of the scriptfile:"
	ocamlc scriptfile.ml
	@echo "*** execution of the compiled code:"
	./a.out
```

Execution :

```
*** creation of a script file:
*** ocaml in script mode:
ocaml scriptfile.ml
4
*** compilation of the scriptfile:
ocamlc scriptfile.ml
*** execution of the compiled code:
./a.out
4
```