Core OCaml For C/C++ Programmers

Part I – Writing Programs

OCaml PRO

March 9-13 2015

Outline

A Complete OCaml Program

Don't panic, we're just tuning the rythm.

Let's rewrite grep

Let's compile and run it

Let’s rewrite grep

We write `grep_lines` that returns the lines in a file that match a given regexp.

```
let rec grep_lines rex fp =
  match input_line fp with
  | line ->
    if Re.execp rex line then
      line :: grep_lines rex fp
    else
      grep_lines rex fp
  | exception End_of_file -> [] ;;
```

Let's rewrite grep

Then we call it from a main function.

```
let main () = match Array.to_list Sys.argv with
  | exe :: pat :: [] ->
    let rex = Re_posix.compile_pat pat in
    let lines = grep_lines rex stdin in
    List.iter (Printf.printf "%s\n") lines
  | exe :: pat :: files ->
    let rex = Re_posix.compile_pat pat in
    List.iter (fun file ->
        let fp = open_in file in
        let lines = grep_lines rex fp in
        List.iter (Printf.printf "%s: %s\n" file) lines ;
        close_in fp)
      files
  | _ ->
    Printf.eprintf "%s <regexp> files\n%!"
      Sys.executable_name ;
    exit 1 ;;
```

Let's compile and run it

And we call our main function.

```
main () ;;
```

We can finally compile our program:

```
ocamlfind ocamlopt -package re.posix grop.ml -o grop -linkpkg
```

And run it!

```
./grop let *.ml
grop.ml: let main () =
grop.ml:     let rex = Re_posix.compile_pat pat in
grop.ml:     let lines = grep_lines rex stdin in
grop.ml:     let rex = Re_posix.compile_pat pat in
grop.ml:         let fp = open_in file in
grop.ml:         let lines = grep_lines rex fp in
```

Writing and Running Basic Programs

Invoking OCaml
General Program Structure
Definitions and expressions
Control Structures

The `ocamlopt` native compiler

- Compiles sources to a standalone executable.
  `ocamlopt hello.ml -o hello`
- Can compile source units separately to `.cmx` s.
  `ocamlopt -c hello.ml ; ocamlopt hello.cmx -o hello`
- Compiles to platform specific programs.
- Produces pretty fast code.

Several automated build systems and tools exist.
We'll begin with direct invocation and simple Makefiles.

The `ocamlc` bytecode compiler

- Compiles sources to a bytecode program.
- Can compile source units separately to `.cmo` s (`-c`).
- Compiles quickly to portable programs.

Bytecode programs can be

- Run by `ocamlrun` .
- Debugged with `ocamldebug` .
- Recompiled to JavaScript using `js_of_ocaml`
- Run on microcontrollers using `ocapic` .

Invoking OCaml

For starters, we'll work with the `ocaml` REPL

- Can run a source file phrase by phrase.
- Can read phrases and display their results interactively.
- Displays the types of results and error messages.
- Useful for beginners, and for interactive testing.

Example session:

```
        OCaml version 4.02.1

# 2 + 2 ;;
- : int = 4
# ^D
```

Structure of an OCaml source unit:

- A file ending in `.ml`.
- A succession of phrases executed in sequence.
- Optionally terminated by `;;` (recommended at first).

We'll start with single source unit, but a program can have many.

Indentation and whitespace:

- Are completely ignored.
- Can be standardized using `ocp-indent`.

Comments:

- Are `(* enclosed like this *)`, no end of line comments.
- Can be nested and know of strings.

General Program Structure 2 / 2

Three kinds of toplevel phrases (for this lecture):

- Global variable definitions.
- Global function definitions.
- In programs, toplevel imperative expressions for their effects.
- In the REPL, toplevel functional expressions, for their result.

Let's write a `hello_user.ml`:

```
let greetings name =
  "Hello " ^ String.capitalize name ^ "!" ;;
let name =
  Sys.getenv "USER" (* FIXME: What if no USER ? *) ;;
print_endline (message name) ;;
```

Defining global variables

```
let first_variable = [| 'O' ; 'C' ; 'P' |] ;;
let second_variable = "OCP" ;;
let first_variable = second_variable
and third_variable = first_variable ;;
```

Syntax:

- Structure `let id = expr ;;` no matter the expression or value type.
- Case sensitive identifiers, starting with lowercase.
- A variable is not visible in its definition.
- Redefining a variable (only) shadows the previous definition.
- Keyword `and` for defining more variables at the same scope level.

Behaviour:

- Variables are immutable (no exceptions).
- Only the contents of allocated values is mutable.
- Allocated values are aliased, never copied.
- The stack contains only immediates and pointers to the heap.

Calling functions & operators

```ocaml
let roll = Random.int 100 ;;
print_endline
  ("Here is your percentage dice roll: "
   ^ string_of_int (roll + 1)
   ^ "%") ;;
Printf.printf
  "Or as two D10: %d and %d\n"
  (roll mod 10 + 1)
  (roll / 10 + 1) ;;
```

To remember:

- Arguments are separated by spaces `f x y z`.
- Application has higher priority than infix operators `f x + 1 = (f x) + 1`.
- Use notation to `Module.ident` access a value of a standard library module.
- Values from the `Pervasives` module are directly available.
- Usual operators with usual priorities for arithmetics.

Defining functions

Syntax:

- Keyword `let` as for variables.
- Space separated arguments follow the name.
- No `return`, the return value is the body expression.

Examples:

```
let succ x = x + 1 ;;
let avg a b = (a + b) / 2 ;;
let greet user = print_endline ("Hello " ^ user) ;;

avg 3 7 (* Useful only in the REPL *) ;;
greet "Jean" ;;
```

Handling effects in an expression language

In OCaml,

- Everything is either a definition or an expression.
- There is no notion of instruction.

But we have imperative traits.

The trick resides in `()` (pronounce *unit*):

- The result of effectful only expressions (assignment, I/O, etc.)
- Functions with no real argument take a placeholder `()`.
- Expressions with `()` result can be assigned to `()` or called at toplevel.
- Bonus: unused results can be assigned to `_` or passed to `ignore`.

Example:

```
let now = Unix.gettimeofday () ;;
let () = print_endline ("It's " ^ string_of_float now) ;;
let () = print_endline "Is it too early for a beer ?" ;;
let _ = read_line () ;;
let () = print_endline "It's always time for a beer !" ;;
```

Sequences and imperative functions

A sequence is an expression:

- It combines several expressions using `;` .
- Its value is the one of the terminal expression..
- Non terminal expressions must return `()` .
- Actually, `let () = a in b` $\Leftrightarrow$ `a ; b` .

Example, rewritten:

```
let now =
  Unix.gettimeofday () ;;
let ask () =
  print_endline ("Now is " ^ string_of_float now ^ ".") ;
  print_endline "Is it too early for a beer ?" ;
  ignore (read_line ()) ;
  print_endline "It's always time for a beer !" ;;
ask () ;;
```

Local variables

Syntax

- Keyword `let` as for globals followed by `in` expression.
- Local definitions can be mixed with sequences.
- A variable is local to an expression, not necessarily a function.
- `let (* ... *) in (* ... *)` is an expression, its value is the body.

Example, rewritten again:

```
let ask () =
  let now = Unix.gettimeofday () in
  print_endline ("Now is " ^ string_of_float now ^ ".") ;
  print_endline "Is it too early for a beer ?" ;
  let _ = read_line () in
  print_endline "It's always time for a beer !" ;;
ask () ;;
```

If then else

```
let greetings () =
  if Random.bool () then
    print_endline ("Hello "^ Sys.getenv "USER")
  else
    print_endline ("Hi "^ Sys.getenv "USER") ;;
let half_polite () =
  if Random.bool () then
    print_endline ("Hello "^ Sys.getenv "USER") ;;
let greetings () =
  print_endline
    ((if Random.bool () then "Hello " else "Hi ")
     ^ Sys.getenv "USER")
```

To remember:

- Conditionals are expressions.
- The `else` branch can be omitted, but the `then` branch must return `()` .
- Branches cannot contain sequences, unless wrapped in parentheses.
- `begin` and `end` are parentheses.

Case Analysis

Example on ints:

```
let very_bad_succ x = match x with
  | 1 -> 2
  | 2 -> 3
  | default ->
    failwith (string_of_int default ^ " not supported") ;;
```

Example on characters:

```
let isalpha x = match x with
  | 'a'..'z' | 'A'..'Z' -> true
  | _ -> false
```

OCaml will warn if the default case is omitted.

- Each case is a pattern followed by a `->` and an expression.
- Several patterns with the same outcome can be grouped.
- Use `..` to express character ranges.
- The default case can be named or ignored with `_`.

Matching is much more powerful, but can be used as a `switch` as here.

Recursive functions

Examples:

```
let rec fact x =
  if x <= 1 then 1 else x * fact (x - 1) ;;

let rec odd n =
  if n = 0 then false else even (pred n)
and even n =
  if n = 0 then true else odd (pred n) ;;
```

Basics:

- Keyword `rec` just after `let`.
- Recursion is your friend in OCaml.
- Recursive definition of non functional values is very restricted.

Control Structures

Tail recursion

Recursive (auto- or not) call as the last expression of a function.

- Optimized, compiled to non stack consuming loops.
- Rewriting often done using a register argument to store partial results.
- Register hidden by wrapping the tail recursive version in another function.

Example:

```
let fact x =
  let rec fact acc x =
    if x <= 1 then acc else fact (x * acc) (x - 1) in
  fact 1 x ;;
```

Predefined Data Types

Types in OCaml
Primitive types
Imperative compound types
Functional compound types

Strong static typing with type inference

Facts:

- All the previous examples where statically checked.
- We never wrote a single type.

OCaml has type inference (almost) everywhere:

- The compiler synthetizes all the types, no need to write them down.
- A generalized version of `auto` of C++, `var` of Java.
- It checks that the whole program is consistent.

OCaml's type checker is very strict:

- No `null` no explicit pointers.
- No unsafe type cast.
- No implicit type conversions.

Syntax of types

Used:

- For annotations.
- By the REPL to indicate the types of results.

Main type syntax categories:

- Primitive types.
  `int`, `string`, `unit`,
- Generic types.
  `'a list`, `'a array`, `('a, 'b) Hashtbl.t`
- Generic type instances.
  `int list`, `string array`
- Function types.
  `int -> string` (int parameter, string result)
  `int -> int -> unit` (two int parameters, `()` rresult)
- Generic function types.
  `'a -> 'a` (result and parameter of the same type)
  `'a -> 'a -> bool` (two parameters of the same type)
  `'a -> 'b -> ('a, 'b) Hashtbl.t)`

Syntax of type annotations

Variables:

```
let three : int = 3
```

Functions:

```
let avg (x : int) (y : int) : int = x + y
```

Expressions (parentheses are mandatory):

```
((3 : int) + (4 : int) : int)
```

Polymorphism inference

OCaml will always infer the most generic correct type.
Intuition: if a value is not examined by the code, its type stays generic.

Parameter used but not introspected:

```
let id x = x ;; (* : 'a -> 'a *)
```

Parameter not used at all:

```
let whatever x = () ;; (* : 'a -> unit *)
```

Use of a polymorphic function:

```
let what x = whatever (id x) ;; (* : 'a -> unit *)
let what x = id (whatever x) ;; (* : 'a -> unit *)
```

Primitive types

Numbers

Examples:

```
let i : int = 3 ;;
let f : float = 3. ;;

let r : int = 3 + (int_of_float 3.5) ;;
let rf : float = 3. +. (float_of_int 4) ;;

let i : int32 = 3l ;;
let i : int64 = 3L ;;
```

To remember:

- Ints and float must be explicitly converted.
- Floats are double precision IEEE754 numbers.
- Operators for floats are postfixed by a period.
- Variables can be annotated with a `':'` ..
- Specific integers `int32` and `int64` with specific operators.
- No unsigned variants (libraries exist).

Working with numbers

See module `Pervasives` for `int` and `float`.

- Integer constants `min_int`, `max_int`.
- Float constants `infinity`, `nan`, etc..
- Math functions `exp`, `cos`, etc.

See module `Int64` for `int64` and `Int32` for `int32`.

Conversions:

- Syntactic convention for conversions `type_of_type`.
- `int_of_float`, `float_of_int`.
- `Int64.of_int`, `Int64.to_float`, etc.

Comparison:

- Generic predefined comparison: `compare: 'a -> 'a -> int`.
- Used by operators: `(=): 'a -> 'a -> bool`.
- And other functions: `max : 'a -> 'a ->'a`, etc.

Works on all numbers with usual semantics.

Strings and Characters

Examples:

```
let s : string = "hello world\n" ;;
let c : char = '\000' ;;
let ls : string = "This is a \
                   continued \
                  \    string".
```

To remember:

- String are immutable byte sequences (recent), no encoding is enforced.
- Characters are single bytes.
- C-like special escapes (`'\n'`, etc.)
- Character escapes are in three digit decimal notation.

Working with strings and characters

See modules `Pervasives` and `String` for strings.

- `String.length : string -> int` gives the length.
- `String.get : string -> int -> char` accesses a character by index.
- `String.set : string -> int -> char -> unit` updates a character.
- Functions `string_of_*` and `*_of_string`.
- Indexes start at 0 and end at `(String.length s - 1)`.
- `(^)` is the concatenation operator.
- Strings are compared lexicographically.
- Conversions and access may fail, we'll see how to handle that later.

See modules `Pervasives` and `Char` for characters.

- Type `char` is not `int`.
- `Char.code : char -> int` gives the underlying byte.
- `Char.chr : int -> char` does the opposite.

Booleans

Type `bool` with two values `true` and `false`.

- Not 0 and 1. `bool` is not `int`.
- Usual sequential operators `&&` and `||`.
- Prefix `not` operator.
- The only type accepted by `if`.

A little example with primitive types

Let us count the number of letters in a string, and make it a program.

```
let count s =
  let len = String.length s in
  let rec loop i =
    if i = len then
      0
    else
      let letter =
        let c = String.get s i in
        (c >= 'a' && c <= 'z')
        || (c >= 'A' && c <= 'Z') in
      (if letter then 1 else 0) + loop (i + 1) in
  loop 0 ;;
print_int (count (read_line ())) ;;
```

That we can compile and run:

`ocamlopt count.l -o count && ./count <<< "hello world"`

Imperative compound types

References

A mutable container for a single value.

- Generic type `'a ref` (contains only values of a given type).
- Build a new reference with `ref : 'a -> 'a ref`.
- Update its content with `(:=) : 'a ref -> 'a -> unit`.
- Access its content with `(!) : 'a ref -> 'a`.
- Useful to simulate mutable variables, or mutable parts of data structures.
- Useful `incr` and `decr : int ref -> unit`
- Can be passed to functions to simulate ”out parameters”.

Example:

```
let ask_for_name rname =
  print_endline "Would␣you␣like␣to␣give␣your␣name␣(y/n)␣?" ;
  if read_line () = "y" then
    rname := read_line () ;;

let name = ref (Sys.getenv "USER") ;;
ask_for_name name ;;
print_endline ("Hello␣" ^ !name) ;;
```

Homogeneous arrays

Parametric type `'a array` of elements of a same type `'a`.

See module `Array`:

- `Array.make : int -> 'a -> 'a array` to create a new array
- `Array.make_matrix : int -> int -> 'a -> 'a array array` to create a new array
- `array.(index)` is the bound checked array access notation.
- Array indexes go from 0 to `(Array.length a - 1)`.

Array constants:

```
let empty = [||] ;;
let sample = [| 0 ; 1 ; 2 ; 3 |] ;;
let mat = Array.make_matrix 2 4 ' ' ;;
```

Access and update:

```
let first = sample.(0) ;;
sample.(0) <- 12 ;;
mat.(1).(1) <- 'X' ;;
```

A little example with references and arrays

Let's count the (polymorphic) zeroes in a matrix.

```
let zeroes zero mat =
  let res = ref 0 in
  let rec do_lines i =
    if i < Array.length mat then
      let rec do_columns j =
        if j < Array.length mat.(i) then begin
          if mat.(i).(j) = zero then
            res := !res + 1 ;
          do_columns (j + 1)
        end in
      do_columns 0 ;
      do_lines (i + 1) in
  do_lines 0 ;
  !res ;;
```

Functional compound types

Tuples

A collection of types `'a1 * 'a2 * ... * 'an` for $n \geq 2$.

Notation for construction and deconstruction:

```
let point : int * int = (3, 2) ;;
let (x, y) = point ;;

let space_point : int * int * int = (3, 2, 1) ;;
let (x, y, z) = space_point ;;

let address num : int * string =
  (num, "butcher street") ;;
let (_, street) = address 13 ;;
```

Homogeneous lists

Parametric type `'a list` of elements of a same type `'a`.

Examples:

```
let el : int list = [] ;;
let l2 : int list = [ 4 ; 5 ; 6 ] ;;
let l3 : int list = 1 :: 2 :: 3 :: l2 ;;
let l4 : int list = l2 @ l3 ;;
```

See modules `Pervasives` and `List`:

- Lists are immutable, single linked lists.
- `::` is prepending in $O(1)$, `@` is concatenation in $O(n)$.
- `[]` is the empty list.
- `List.hd`, `List.tl`, `List.length`, and a lot of combinators.

Optional values

A value of type `'a option` is either `None` or `Some v` where `v` is of type `'a`.
Examples:

```
let someone : int option = Some 1 ;;
let nobody : int option = None ;;
```

To remember:

- Useful for functions (optional argument, optional return value).
- `None` means no value present.
- `Some v` means the value is present and its value is v.

A little example with tuples and lists

Let's record the positions of (polymorphic) zeroes in a matrix.

```
let zeroes zero mat =
  let rec do_lines i =
    if i < Array.length mat then
      let rec do_columns j =
        if j < Array.length mat.(i) then
          if mat.(i).(j) = zero then
            (i, j) :: do_columns (j + 1)
          else do_columns (j + 1)
        else [] in
      do_columns 0 @ do_lines (i + 1)
    else [] in
  do_lines 0 ;;
```

Advanced control

Imperative loops
First class functions
Pattern matching
Exceptions

Imperative loops 1 / 2

Usual integer iterator

- Increments a local integer counter each step.
- Stepping 1 with `to`, -1 with `downto`.
- Bounds are computed at start.
- The loop has value `()`, its body must have value `()`.

Example:

```
for i = 99 downto 0 do
  print_int i ;
  print_endline " bottles on the wall"
done
```

Imperative loops 2 / 2

Usual unbounded iterator

- Tests a condition before each turn.
- The condition must evaluate to a boolean value.
- The loop has value `()`, its body must have value `()`.

Example:

```
print_endline "Welcome to Dice Simulator 2000." ;;
let continue = ref true ;;
while !continue do
  print_int (Random.int 6 + 1) ;
  print_endline "Continue (y/n) ?" ;
  continue := (read_line () = "y")
done ;;
```

Taking functions as parameters

Example: call an imperative function on a an integer range.

```
let rec call_on_interval
  (min : int) (max : int) (f : int -> unit) : unit =
  if min <= max then begin
    f i ;
    call_on_interval (min + 1) max f
  end ;;
call_on_interval 10 20 print_int ;;
```

Example: call a function on a an integer range and store its results in a list.

```
let rec map_interval
  (min : int) (max : int) (f : int -> 'a) : 'a list =
  if min > max then
    []
  else
    f i :: map_interval (min + 1) max f ;;
map_interval 10 20 succ ;;
```

Returning functions

Functions can be returned (or stored) as any other value.

Example: choose an integer operator from it name.

```
let operator (name : string) : int -> int -> int =
  match name with
  | "min" -> min
  | "max" -> max
  | "+" -> (+) (* operators in parentheses are idents *)
  | "-" -> (-)
  | _ -> failwith "unknown operator name" ;;
```

Closures

Even local functions can be returned or stored.
They embed their necessary environment.

Example: a hook initialized with a closure.

```
let print_hook = ref print_int ;;
let print num = !print_hook num ;;

let set_delimited_printer start stop =
  let printer num =
    print_string (start ^ string_of_int num ^ stop) in
  print_hook := printer (* with start, stop embedded *) ;;

set_delimited_printer "<< " " >>\n" ;;
print 3 ;;
call_on_interval 10 20 print ;;
```

Anonymous functions

There is no need to name functions that only serve once.

Two syntaxes for anonymous functions:

- `fun arg1 arg2 ... argn -> body`
- `function case1 -> body1 | ... | casen -> bodyn` for defining a function using case analysis.

Examples using the iterators we defined earlier:

```
map_interval 10 20
  (fun i -> i mod 2) ;;
call_on_interval 0 100
  (fun i -> if i mod 2 = 0 then print_int i) ;;
```

Gives an alternative, equivalent syntax for defining names functions.
`let f x y = body` ⇔ `let f = fun x y -> body`

And a shorcut for match.
`let f x = match x with ...` ⇔ `let f = function ...`

Iterators in the standard library

Modules `List` and `Array` define a lot of functional iterators.

Example on lists:

- `iter : ('a -> unit) -> 'a list -> unit`
  Apply an imperative function on all elements.
- `map : ('a -> 'b) -> 'a list -> 'b list`
  Apply a function on all elements and return the results.
- `filter : ('a -> bool) -> 'a list -> 'a list`
  Return all the elements satisfying a predicate.
- `fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`
  Apply a binary operation using an accumulator as left operand and successively all elements as right operand.

Partial application

In OCaml, as in λ-calculus, a function with n arguments

- is a single argument functions returning another with $n-1$ arguments ;
- can actually be defined so ;
- can be passed arguments one by one.

For instance, all these definitions are equivalent:

```
let f x y z = x + y + z ;;
let f x y = fun z -> x + y + z ;;
let f = fun x -> fun y -> fun z -> x + y + z ;;
let f x y =
  let partial_sum = x + y in
  fun z -> partial_sum + z ;;
```

They have type `int -> int -> int -> int`, and

- `f 0` has type `int -> int -> int`
- `f 3 8` has type `int -> int`

Partial application and effects

The body of a fun, and its effects, are evaluated when its arguments are passed.

Example: a unique integer generator:

```
let make_generator () =
  let uid = ref 0 in
  fun () ->
    uid := !uid + 1 ;
    !uid
```

`make_generator` has type `unit -> unit -> int`.

- Passing the first `()` creates the ref and returns a function of `unit -> int`;
- this function updates the reference at each `()` passed.

```
let gen = make_generator () ;;
gen () ;; (* displays 1 in the toplevel *)
gen () ;; (* displays 2 in the toplevel *)
```

Some examples with iterators and partial application

Make all the integers in a list to be at least 8:

```
let min_eight l = List.map (max 8) l ;;
```

Make all the floats in a list relative to the maximum:

```
let relativize l =
  let lower = List.fold_left min infinity l in
  List.map ((+) (-lower)) l
```

Drop all the floats of a list under its average.

```
let drop_low l =
  let total = List.fold_left (+) 0. l in
  let average = total /. List.length l in
  List.filter ((<=) average) l
```

Pattern matching

One of the key features or OCaml.

- Acts like a `switch` in simple cases.
- Can deeply destruct a value using a deconstruction pattern.
- Can name parts of the deconstructed value for reuse.
- Does case analysis by comparing with multiple patterns.
- Optimizes as much as possible the branchings.

It needs a bit of attention for beginners.

The pattern for a value looks like the expression for building it, except:

- `_` , called catch-all, can replace a subpattern, accepting any value.
- A name in a pattern is a catch-all that names the matched subvalue.

Destructing lists

Simple example:

```
let rec length l = match l with
  | [] -> 0
  | _ :: tail -> 1 + length tail
```

Deeper matching:

```
let rec even_length l = match l with
  | [] -> true
  | [ _ ] -> false
  | _ :: _ :: tail -> even_length tail
```

The `tail` variable is a fresh variable.

- Bound during deconstruction.
- Only usable in the branch on the right of the pattern.

On tuples

A single pattern is enough, is all components are catch-all.

```
let add_components (pair : int * int) : int =
  match pair with
  | (x, y) -> x + y
```

`function` if sugar for `fun x -> match x with`.

But not if subpatterns are not self exhaustive:

```
let add_components
  : int option * int option -> int option
  = function
    | (Some x, Some y) -> Some (x + y)
    | (None, Some x) | (Some x, None) -> Some x
    | (None, None) -> None
```

To match two values at the same time, simply match their pair.

```
match (x, y) with (* ... *)
```

On options

Apply a treatment only on present values:

```
let maybe_plus_one : int option -> int option = function
  | None -> None
  | Some x -> Some (x + 1)
```

Change the behaviour if some argument is present.

```
let rec concat sep l =
  match (sep, l) with
  | _, [ last ] -> last
  | Some s, first :: rest -> first ^ s ^ concat sep rest
  | None, first :: rest -> first ^ concat sep rest
  | _, [] -> ""
```

A typical beginner error

The user wants to compare a value to other reference values.

```
let is_x_or_y x y v =
  match v with
  | x -> true
  | y -> true
  | _ -> false
```

- This always returns `true`, because `x` is a newly introduced variable that is bound to any value of v. The other cases are not tested.
- The last two cases are actually unused (the compiler will tell).

Solution:

```
let is_x_or_y x y v =
  v = x || v = y
```

- Matching identifies patterns in a single value.
- Matching does not compare a value with another.

Exhaustivity checking

OCaml will always detect a missing case and suggest a completion.

```
# let f = function '\000' .. '\254' -> 2 ;;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched: '\255'
val f : char -> int = <fun>
# let f = function '\000' .. '\255' -> 2 ;;
val f : char -> int = <fun>
```

Some cases can be corrected by treating all cases:

```
match Random.bool with | true -> (* ... *) | false -> (* ...
```

Other, like strings will require a catch-all:

```
match Sys.argv.(1) with
| "-help" -> display_help
| arg -> invalid_arg ("bad␣argument␣" ^ arg)
```

Bonuses

The `let` construction actually expects a pattern and not a name.
But this pattern should be self exhaustive, otherwise OCaml will warn.

```
let (x, y, z) = (x, y, z) (* OK *) ;;
let [] = [ 1 ; 2 ] (* warns at compile time,
                      fails at runtime *) ;;
```

The same goes for arguments of `let` and `fun`:

```
let f (x, y) = x + y (* OK *) ;;
let f [] = 2 (* warns at compile time *) ;;
f [ 1 ; 2 ] (* fails at runtime *) ;;
```

Non terminal parts of patterns can be names using `as`.

```
let rec all_equal = function
  | [] -> true
  | x1 :: (x2 :: _ as rest) ->
    x1 = x2 && all_equal rest
```

Throwing exceptions

Predefined exception throwers, to abort execution:

```
failwith "error message" ;;
invalid_arg "sqrt: negative argument" ;;
assert false ;; (* indicates file + line*)
```

Generic exception thrower: `raise Exn`, with predefined exceptions:

- `Not_found` raised by `Hashtbl.find`, `List.assoc`, etc.
- `Exit`
- `End_of_file`

Exceptions can be defined, at toplevel of a file:

```
exception Error ;;
```

Exception constructors start with a capital.

An exception is a first class value of type `exn`.

Catching exceptions

The dedicated structure `try ... with cases`:

```
let rec echo () =
  try
    let line = read_line () in
    if line = "exit" then raise Exit ;
    print_string line ;
    echo ()
  with
    | End_of_file -> ()
    | Exit -> print_endline "Bye." ; ()
```

Beware, the `try` structure breaks tail call optimization.

Catching exceptions with match

Exceptional cases can be added to a match:

```
let rec echo () =
  match read_line () with
  | "exit" -> print_endline "Bye."
  | line -> print_endline line ; echo ()
  | exception End_of_file -> ()
```

Preserving tail call optimization.

Runtime exceptions

They should not be caught (or thrown):

- `Invalid_argument "reason"`
- `Failure "reason"`
- `Stack_overflow`
- `Out_of_memory`

Catching all exceptions with `_` is also a bad idea.

The Standard Library

A selection of modules without advanced features.

Primitive types
Data structures
Input and output
System interface

Primitive types

- Pervasives, The initially opened module.
- Char, Character operations.
- Bytes, Byte sequence operations.
- String, String operations.

- List, List operations.
- Array, Array operations.
- Buffer, Extensible buffers.
- Hashtbl, Hash tables and hash functions.
- Stack, Last-in first-out stacks.
- Queue, First-in first-out queues.

Input and output

- Printf, Formatted output functions.
- Str, Regular expressions and high-level string processing
- Digest, MD5 message digest.
- Graphics, Machine-independent graphics primitives.

System interface

- Pervasives, The initially opened module.
- Arg: Parsing of command line arguments.
- Filename, Operations on file names.
- Random, Pseudo-random number generators (PRNG).
- Sys, System interface.

Outline