Introduction to functional programming with OCaml (part III)

Lists $[e_1; \ldots; e_n]$, $n \geq 0$

Office Hours

- Tuesday or Thursday, 2:00 – 3:15 PM (before the class)
- By email appointment only (to avoid overcrowding at the same time):
  pcousot@cs.nyu.edu
- Can also be organized after the class
- DO NOT HESITATE!
**Lists and tuples**

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

(* empty list *)
[];;
- : 'a list = []

(* lists must have all elements of the same type *)
[1; []];;
Error: This expression has type 'a list
but an expression was expected of type int
```

**Cons, head, and tail**

```
# (* 'cons' operator: head :: tail *)
1 :: 2 :: 3 :: 4 :: 5 :: [];;
- : int list = [1; 2; 3; 4; 5]
# List.hd [1; 2; 3; 4; 5];;
- : int = 1
# List.tl [1; 2; 3; 4; 5];;
- : int list = [2; 3; 4; 5]
# (List.hd [1; 2; 3; 4; 5]) :: (List.tl [1; 2; 3; 4; 5]);;
- : int list = [1; 2; 3; 4; 5]
# List.length [1;2;3;4;5];;
- : int = 5
```

See the List module: [http://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html](http://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html)

**Operations on lists**

<table>
<thead>
<tr>
<th>Function</th>
<th>Examples</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>:: 'a -&gt; 'a list -&gt; 'a list</td>
<td>5 :: [6; 7]</td>
<td>Add an element to the front of the list. This operator is right associative.</td>
</tr>
<tr>
<td>@ : 'a list -&gt; 'a list -&gt; 'a list</td>
<td>[5] @ [6; 7]</td>
<td>List concatenation.</td>
</tr>
<tr>
<td>List.length : 'a list -&gt; int</td>
<td>List.length [5; 6; 7]</td>
<td>Number of elements in the list.</td>
</tr>
<tr>
<td>List.hd : 'a list -&gt; 'a list</td>
<td>List.hd [3; 5; 7]</td>
<td>The &quot;head&quot; of a list is its first element. Same as car in LISP.</td>
</tr>
<tr>
<td>List.tl : 'a list -&gt; 'a list</td>
<td>List.tl [3; 5; 7]</td>
<td>The &quot;tail&quot; of a list is the list with its first element removed. Same as cdr in LISP.</td>
</tr>
<tr>
<td>List.nth : 'a list -&gt; int -&gt; 'a</td>
<td>List.nth [3; 5; 7] 2</td>
<td>Returns the nth element of a list, counting from zero.</td>
</tr>
<tr>
<td>List.rev : 'a list -&gt; 'a list</td>
<td>List.rev [1; 2; 3]</td>
<td>Reverse the list.</td>
</tr>
</tbody>
</table>

- Only the empty list [], :: (LISP cons), and @ (list concatenation) operators can be used without opening the List module or prefixing the function name with List..

from [http://www.csc.villanova.edu/~dmatusze/resources/ocaml/ocaml.html](http://www.csc.villanova.edu/~dmatusze/resources/ocaml/ocaml.html)
### Pattern matching

```ocaml
# let (a, b, c) = (1, true, "abc") in c;;
- : string = "abc"
# let (a, b, c) = (1, true, "abc");;
val a : int = 1
val b : bool = true
val c : string = "abc"
# let l = [0; 1; 2];;
val l : int list = [0; 1; 2]
# match l with
| [] -> []
| hd :: tl -> tl;;
- : int list = [1; 2]
#
```

### Exhaustive pattern matching

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

### Exhaustive pattern matching

```ocaml
# let hd :: tl = [0; 1; 2];;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val hd : int = 0
val tl : int list = [1; 2]
#
```

### Default case ( _ matches everything)

```ocaml
# x;;
- : int = 4
# match x with
| 0 -> 0
| 1 -> 2
| 2 -> 4
| 3 -> 6
| _ -> -1;;
- : int = -1
#```
Matching by intervals

```ml
# let c = 'D';;
val c : char = 'D'
# match c with
  | 'a'..'z' -> 0
  | 'A'..'Z' -> 1
  | '0'..'9' -> 2
  | _ -> -1;;
- : int = 1
```

Matching with conditions

```ml
# x;;
val x : int = 2
# match x with
  | y when y > 0 -> 1
  | 0          -> 0
  | y when y < 0 -> (-1);;
Warning P: 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 = 1
```

Exhaustivity is undecidable, so compilers can always be improved.

Matching with patterns

1) pairs/tuples

```ml
# match (1, 2) with
  | (x, 1) -> x
  | (1, y) -> y
  | _   -> 0;;
- : int = 2
```

Match failure

```ml
# let x = 4;;
val x : int = 4
# match x with
  | 0   -> 0
  | 1   -> 2
  | 2   -> 4
  | 3   -> 6;;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
4
Exception: Match_failure ("//toplevel//", 4, -24).
```
Matching with patterns

II) Lists

```ocaml
# match [1;2;3] with
  | [] -> [0]
  | h :: t -> 0 :: t;;
- : int list = [0; 2; 3]
```

```ocaml
# let rec length l = match l with
  | [] -> 0
  | h :: t -> 1 + length t;;
val length : 'a list -> int = <fun>
# length [1;2;3;4;5];;
- : int = 5
```

Matching variant types

```ocaml
# (* No union type in ML *)
type t = int | bool;;
Error: Syntax error
# (* Variant types *)
type t = INT of int | BOOL of bool;;
type t = INT of int | BOOL of bool
# INT 10;;
- : t = INT 10
# let x = BOOL true;;
val x : t = BOOL true
# match x with
  | BOOL b -> 0
  | INT i -> 1;;
- : int = 0
# match x with
  | BOOL b -> b
  | INT i -> i;;
Error: This expression has type int but an expression was expected of type
  bool
```

Syntactically equivalent forms of function definitions by pattern matching

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

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

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

Operations on lists
Classical operations of lists

# (* append *)
[1;2;3] @ [4;5];;
- : int list = [1; 2; 3; 4; 5]

# (* reverse *)
List.rev [1;2;3];;
- : int list = [3; 2; 1]

# (* map *)
list.map (function n -> 2 * n) [1;2;3];;
- : int list = [2; 4; 6]

Association lists

Classical operations of lists

# (* append *)
[1;2;3] @ [4;5];;
- : int list = [1; 2; 3; 4; 5]

# (* reverse *)
List.rev [1;2;3];;
- : int list = [3; 2; 1]

# (* map *)
list.map (function n -> 2 * n) [1;2;3];;
- : int list = [2; 4; 6]

Example: expression evaluation

# type expression =
  | Num of int
  | Var of string
  | Let of string * expression * expression
  | Binop of string * expression * expression;;

# let rec eval env = function
  | Num i -> i
  | Var x -> List.assoc x env
  | Let (x, e1, in_e2) ->
    let val_x = eval env e1 in
    eval ((x, val_x) :: env) in_e2
  | Binop (op, e1, e2) ->
    let v1 = eval env e1 in
    let v2 = eval env e2 in
    eval_op op v1 v2
and eval_op op v1 v2 =
  match op with
  | "+" -> v1 + v2
  | "-" -> v1 - v2
  | "*" -> v1 * v2
  | "/" -> v1 / v2
  | _ -> failwith ("Unknown operator: " ^ op);

val eval : (string * int) list -> expression -> int = <fun>
val eval_op : (string * int) list -> expression -> int = <fun>

List generation

# 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 interval m n =
  let p x = x <= n in generate (function n -> n + 1) p m;;
val interval : int -> int -> int list = <fun>

# interval 3 10;;
- : int list = [3; 4; 5; 6; 7; 8; 9; 10]

# interval 10 3;;
- : int list = []
List reduction

reduce f a [x₁; x₂; ..., xₙ] = (f xₙ (f xₙ₋₁(...(f x₁ a)...)))

# let rec reduce f a l = match l with
| [] -> a
| h :: t -> reduce f (f h a) t;
val reduce : ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b = <fun>

Correctness proof of “reduce”

reduce f a’ [] = a’ by definition

reduce f a’ [x₁; ...; xₙ] = (f xₙ (f xₙ₋₁(...(f x₁ a’)...))) assumed by induction hypothesis

reduce f a [x₀; ...; xₙ] = reduce f (f x₀ a) [x₁; ...; xₙ] by definition
= (f xₙ (f xₙ₋₁(...(f x₁ (f x₀ a))...))) by ind. hyp.

Q.E.D. by induction on the length of the list.

Principle of Quicksort

Sort

h

Partition

left

≤ h

h

> h

right

Sort

Sort

An example: Quicksort

Quicksort is a well-known sorting algorithm developed by C. A. R. Hoare.

Example

Partition

```ocaml
# let partition p = let add x (pos, neg) = if p x then (x::pos), neg else pos, (x::neg) in reduce add ([], []);; 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

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

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

Mutable variables
Immutable variables

- The binding of a variable identifier/name and a value is immutable in parameter passing and the let construction.

These immutable variables are variables in the mathematical sense (unfortunately called constants in some programming languages such as Pascal or Ada).

References and assignments

```ocaml
# let x = ref 0;;
val x : int ref = {contents = 0}

# !x;;
- : int = 0

# x := !x + 1;;
- : unit = ()

# !x;;
- : int = 1

# x;;
- : int ref = {contents = 1}

Impossible to have “uninitialized” variables!
```

References and mutable variables

- A variable in the programming language sense is an immutable binding of a name/identifier to a mutable memory (which content can be changed through assignment).

![Immutability Diagram]

- In many languages the variable x and its content !x are both denoted x, the meaning being different on the left and righthand side of an assignment.

Impure functions

- Impure function with assignments to global variables have side effects

```ocaml
# let count = ref 0;;
val count : int ref = {contents = 0}

# let incr n =
  count := !count + n;
!count;;
val incr : int -> int = <fun>

# incr 3;;
- : int = 3

# incr 3;;
- : int = 6
```
Control structures

The value of a sequence of expressions is that of the last expression. All expressions must be evaluated to record side effects (input/output, assignments).

```ocaml
# let x = ref 0;;
val x : int ref = {contents = 0}
# print_int !x;;
x := !x + 1; !x + 1;;
0- : int = 2
#
```

Conditional

The value of a conditional expression is that of the chosen “then” or “else” expression. The evaluation is lazy in that the non-chosen expression is not evaluated.

```ocaml
# if true then 0 else failwith "error";;
- : int = 0
#
```

Bounded/for loops

Step always ≠ 1!

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

# for i = 10 downto 1 do
  print_int i
  done;;
10987654321- : unit = ()
#
```
**Unbounded/while loop**

```ocaml
let x = ref 0;;
val x : int ref = {contents = 0}
# while !x < 10 do
    print_int !x;
    x := !x + 1
# done;;
0123456789- : unit = ()
#
```

**Exceptions**

```ocaml
failwith "error";;
Exception: Failure "error".
raise (Failure "error");;
Exception: Failure "error".

1/0;;
Exception: Division_by_zero.
try 1/0 with
  | Division_by_zero -> failwith "error";;
Exception: Failure "error".
#
```

**Predefined expressions**

```ocaml
let x = -17;;
val x : int = -17
# try
    if x > 0 then raise Pos
    else if x = 0 then 0
    else raise Neg
with
  | Pos -> 1
  | Neg -> -1;;
- : int = -1
#
```

**Defining and capturing exceptions**

```ocaml
let x = -17;;
val x : int = -17
# try
    if x > 0 then raise Pos
    else if x = 0 then 0
    else raise Neg
with
  | Pos -> 1
  | Neg -> -1;;
- : int = -1
#```
Exceptions carrying values

```ocaml
# exception Exit of int;;
exception Exit of int
# try
  for k = 0 to 100 do
    if k = 5 then raise (Exit k)
    else print_int k
done
with
  | Exit i -> print_int i;;
```

012345- : unit = ()

```
```

Books on OCaml and related functional languages

- *The Objective Caml Programming Language* by Tim Rentsch
- *Practical OCaml* by Joshua B. Smith
- *The Functional Approach to Programming* by Guy Cousineau, Michel Mauny, and K. Callawa
- *ML for the Working Programmer* by Lawrence C. Paulson
- *Developing Applications With OCaml* by Emmanuel Chailloux, Pascal Manoury and Bruno Pagano (http://caml.inria.fr/pub/docs/oreilly-book/)
- *F# for Scientists* by Jon D. Harrop
- *Modern Programming Languages: A Practical Introduction* by Adam Brooks Webber

Bibliography


Online books