

# Introduction à OCaml

OPTION INFORMATIQUE - TP n° 1.0 - Olivier Reynet

## À la fin de ce chapitre, je sais :

- ☞ expliquer le fonctionnement de l'inférence de type
- ☞ lire la signature d'une fonction
- ☞ définir un type simple en OCaml
- ☞ définir un type somme simple
- ☞ utiliser le filtrage de motif sur un cas simple
- ☞ utiliser une référence pour programmer impérativement
- ☞ programmer une fonction récursive simple

## A Coder en OCaml?

**En ligne** Pour utiliser OCaml en ligne, il suffit d'utiliser les bacs à sable des sites [OCaml](#) ou [TryOcaml](#).

**Sur votre machine locale** Sur votre machine, le plus simple est d'utiliser l'interprète interactifs OCaml. [On peut facilement l'installer sur n'importe quel système d'exploitation en suivant ces instructions.](#)

**Pour travailler avec un éditeur de texte** Emacs facilite l'édition de code OCaml grâce au mode Tuareg. Les commandes `M-x tuareg-mode` et `M-x run-ocaml` permettent d'activer ce mode. [Le résumé \(sic!\) des commandes est accessible en ligne. Ce tutoriel vous montre quelques commandes de base pour survivre avec Emacs.](#) Pour les puristes de la ligne de commande, ce mode Tuareg est également disponible sous Vim.

**Utiliser un IDE** Enfin, il est possible d'utiliser OCaml avec la plupart des environnements de développement : Eclipse, Visual Studio ou IntelliJ (Jet Brains).

## B Tester les commandes via l'interprète interactif

OCaml dispose d'un interprète interactif. Par défaut, c'est l'exécutable `ocaml` mais il existe également `utop`.

B1. L'inférence de type est un mécanisme puissant. Sur les éléments suivants, tenter de deviner les types de ces variables ou expressions et vérifier avec l'interprète :

- (a) `let n = 3`
- (b) `let x = 3.14`
- (c) `let c = 'a'`

- (d) `let s = "ocaml"`
- (e) `let b = n > 3`
- (f) `let bi = if n > 2 then true else false`
- (g) `let m = if bi then 4.5 else 7.`
- (h) `let elements = [1;3;5;42]`
- (i) `let elements = [true;false;true]`
- (j) `()`
- (k) `print_int 3`
- (l) `let t = (3,"clef")`
- (m) `let st = (4,5)`
- (n) `let t = (3.,"clef")`
- (o) `let tt = (true, 42, 'z')`

B2. Tenter de deviner les signatures des fonctions suivantes :

- (a) `let u n = 3*n + 2`
- (b) `let f x y = (x+.y)*.(x-.y)`
- (c) `let g n x = (float_of_int n)*. x`
- (d) 

```
let h n x =
    let r = ref 1. in
    for k = 1 to n do
        r := !r *. x
    done;
    !r;;
```

B3. S'entraîner à utiliser ces variables et ces fonctions dans un programme principal.

### Solution :

#### Code 1 –

```
let n = 3;;
let x = 3.14;;
let c = 'a';;
let s = "ocaml";;
let b = n > 3;;
let bi = if n > 2 then true else false;;
let m = if bi then 4.5 else 7.;;
let elements = [1;3;5;42];;
let elements = [true;false;true];;
let elements = [2.3;0.3;1.];;

();;
let st = (4,5);;
let t = (3.,"clef");;
let tt = (true, 42, 'z');
```

```

let u n = 3*n + 2;;
let f x y = (x+.y)*.(x-.y);;
let g n x = (float_of_int n) *. x;;
let h n x = let r = ref 1. in
  for k = 0 to n do
    r := !r *. x
  done;
  !r;;

(* MAIN PROGRAM *)

u n;;
f x x;;
g (n+2) (x -. 0.54);;
h (n+5) (3. *. x);;

```

## C Variables globales et locales

- C1. Créer un fichier `vars.ml`.
- C2. Définir une variable globale `rayon` et l'initialiser à la valeur 1.21.
- C3. Coder une fonction de signature `disk_area : float -> float` qui renvoie l'aire d'un disque en fonction de son rayon.
- C4. Modifier la fonction précédente pour faire apparaître une variable locale à la fonction nommée `pi` et valant 3.14159265359.
- C5. Quelles sont les variables locales et globales de ce code?

### Solution :

#### Code 2 –

```

let rayon = 1.21;;
let disk_area r = 3.14159265359*.r*.r;;
let disk_area r = let pi = 3.14159265359 in pi*.r*.r;;

(* MAIN PROGRAM *)

disk_area rayon;;

```

## D Fonctions et fonctions récursives

- D1. Créer un fichier `fact.ml`
- D2. Coder une fonction impérative de signature `i_fact : int -> int` qui renvoie  $n!$ .
- D3. Coder une fonction **récursive** de signature `fact : int -> int` qui renvoie  $n!$ .

- D4. Coder une fonction **récursive terminale** de signature `tr_fact : int -> int` qui renvoie  $n!$ . On utilisera une fonction auxiliaire interne à la fonction.
- D5. Peut-on calculer  $42!$ . Pourquoi?

**Solution :****Code 3 –**

```
(* imperative *)
let i_fact n = let i = ref 1 and f = ref 1 in
  while !i < n do
    incr i; (* same as : i := !i + 1 *)
    f := !f * !i
  done;
  !f;;

(* recursive match with *)
let rec fact n =
  match n with
  | 0 -> 1
  | _ -> n * fact (n - 1);;

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

(* tail recursive (récursif terminal) *)
let tr_fact n =
  let rec aux k acc = match k with
    | 0 -> acc
    | _ -> aux (k - 1) (acc * k) in
  aux n 1;;

(* MAIN PROGRAM *)
i_fact 6;;
fact 6;;
tr_fact 6;;
fact 42;;
tr_fact 42;;
fact 39;;
```

**E Jouer à la bataille**

On souhaite coder les fonctions élémentaires pour jouer à la bataille. On choisit de modéliser les cartes avec des types algébriques.

- E1. Créer un fichier `bataille.ml`.
- E2. Coder un type `somme couleur` capable de représenter les quatre couleurs d'un jeu de carte : trèfle, pique, carreau, cœur.

- E3. Coder un type somme `figure` capable de représenter les figures, c'est à dire le roi, la dame et le valet.
- E4. Coder un type algébrique `carte` capable de représenter une carte quelconque.
- E5. Dans le programme principal, créer quelques cartes.
- E6. **En utilisant le filtrage de motif**, coder une fonction de signature `get_value : carte -> int` qui renvoie la valeur associée à une carte. On considère que l'as vaut 14, le roi 13, la dame 12, le valet 11 et les numéros leur propre nombre.
- E7. Coder une fonction de signature `comparer : carte -> carte -> int` qui compare deux cartes d'après leur valeur. La fonction renvoie un entier valant la différence entre les valeurs des cartes.
- E8. Coder une fonction de signature `bataille : carte -> carte -> bool` qui teste s'il y a bataille entre deux cartes.

**Solution :****Code 4 –**

```
(* DEFINE TYPES *)
type couleur = Pique | Trefle | Coeur | Carreau
type figure = Roi | Dame | Valet
type carte = Figure of figure * couleur | Numero of int * couleur

(* BASE FUNCTIONS *)
let get_value carte =
  match carte with
  | Numero (1,_) -> 14
  | Figure (Roi,_) -> 13
  | Figure (Dame,_) -> 12
  | Figure (Valet,_) -> 11
  | Numero (n,_) -> n;;

let get_value = function
  | Numero (1,_) -> 14
  | Figure (Roi,_) -> 13
  | Figure (Dame,_) -> 12
  | Figure (Valet,_) -> 11
  | Numero (n,_) -> n;;

let comparer c1 c2 = (get_value c1) - (get_value c2);;

let bataille c1 c2 = match comparer c1 c2 with
  | 0 -> true
  | _ -> false;;

(* LET'S TEST ! *)
let asc = Numero (1, Coeur);;
let rc = Figure (Roi, Pique);;
let nc = Numero (9, Carreau);;
let tc = Numero (3, Trefle);;

comparer asc nc;;
comparer tc rc;;
bataille asc asc;;
bataille rc asc;;
```

En deuxième année, vous pourrez programmer un automate qui implémentera la logique du jeu.

## F Calculer avec les entiers en OCaml

- F1. Créer un fichier `evens.ml` et écrire une fonction de signature `even : int -> bool` qui teste si un nombre est pair. En déduire une fonction qui teste si un nombre est impair. L'opérateur modulo est `mod` en OCaml.

### Solution :

#### Code 5 –

```
let even n = n mod 2 = 0;;
let odd n = not (even n);;
let odd n = n mod 2 = 1;;

(* MAIN PROGRAM *)

even 21;;
even 42;;
odd 33;;
odd 14;;
```

- F2. Créer un fichier `syracuse.ml`. On considère la suite de Syracuse  $(u_n)_{n \in \mathbb{N}}$  définie par  $u_0 \in \mathbb{N}^*$  et :

$$\forall n \in \mathbb{N}, \quad u_{n+1} = \begin{cases} 3u_n + 1 & \text{si } u_n \text{ est impair} \\ \frac{u_n}{2} & \text{si } u_n \text{ est pair} \end{cases} \quad (1)$$

- (a) Écrire une fonction de signature `next_u : int -> int` qui calcule le terme suivant de la suite.  
 (b) Écrire une fonction de signature `syracuse : int -> int` qui renvoie le temps de vol de la suite pour un entier  $u_0$  donné<sup>1</sup>. On utilisera une boucle `while` et deux références.

### Solution :

#### Code 6 –

```
let next_u u = if (u mod 2) = 1 then 3 * u + 1 else u / 2 ;;

let syracuse u0 =
  let u = ref u0 and n = ref 0 in
  while !u > 1 do
    u := next_u !u;
    incr n
    (* same as --> n := !n + 1 *)
  done;
  !n;;

(* MAIN PROGRAM *)
```

1. c'est à dire l'indice  $N$  de la suite pour lequel  $u_N = 1$ , s'il existe...

```

next_u 3;;

syracuse 3;;
syracuse 17;;
syracuse 97;;

```

- F3. Créer un fichier `gcd.ml`. Écrire une fonction **récursive** de signature `gcd : int -> int -> int` qui calcule le PGCD de deux entiers naturels. En déduire une fonction de signature `coprime : int -> int -> bool` qui teste si deux nombres sont premiers entre eux.

**Solution :**

**Code 7 –**

```

let rec gcd a b = if b = 0 then a else gcd b (a mod b);;

let coprime a b = (gcd a b) = 1;;

(* MAIN PROGRAM *)

gcd 39 15;;
gcd 64 47;;
coprime 39 15;;
coprime 64 47;;

```

- F4. On souhaite tester la conjecture de Goldbach qui affirme que *tout entier naturel pair plus grand que 2 est la somme de deux nombres premiers*. Créer un fichier `goldbach.ml`
- Écrire une fonction de signature `is_prime : int -> bool` qui teste si un nombre est premier. On rappelle que 1 n'est pas un nombre premier.
  - Écrire une fonction de signature `goldbach : int -> int * int` qui teste la conjecture de Goldbach. Si le nombre entier fourni en paramètre est impair, le programme échoue en imprimant le message `"Goldbach's conjecture only on even numbers"`. Sinon, il renvoie un tuple contenant les deux nombres solution.

**Solution :**

**Code 8 –**

```

let is_prime n =
  if n = 1
  then false
  else
    begin
      let d = ref 2 and result = ref true in
      while !d * !d <= n && !result do
        if n mod !d > 0 then incr d else result := false
      done;
    end

```

```
        !result
    end;;

let is_prime n =
    let rec not_divided_by d =
        if d * d > n then true else (n mod d <> 0 && not_divided_by (d + 1))
    in
    n <> 1 && not_divided_by 2;;

let goldbach n =
    if n mod 2 > 0
    then failwith "Goldbach's conjecture only on even numbers !"
    else
        begin
            let rec aux d =
                if is_prime d && is_prime (n - d)
                then (d, n - d)
                else aux (d + 1)
            in aux 2
        end;;

(* MAIN PROGRAM *)

is_prime 17;;
is_prime 43;;
is_prime 45;;
is_prime 1;;
is_prime 2;;
is_prime 101;;

goldbach 14;;
goldbach 21;;
goldbach 42;;
goldbach 101;;
goldbach 102;;
```