

MPSI

# Option Informatique

## TP 1

David Baelde  
david.baelde@ens-lyon.org

Le but de ce premier TP est de se familiariser avec le langage OCaml. Vous apprendrez ici les bases, que nous compléterons lors des prochains TP tout en commençant l'algorithmique à proprement parler.

### Problèmes techniques

Loguez vous en utilisant la session dédiée à votre MPSI. Nous utiliserons Emacs, qui se mettra automatiquement en mode Caml (appelé Tuareg-mode) quand vous ouvrirez un fichier doté de l'extension `.ml`. Vous enregistrerez votre travail pour le TP  $N$  dans le fichier `~/Pub/06-tp{n}-{nom}.ml`.

Voici quelques raccourcis utiles dans Emacs, où  $C$  dénote `Ctrl` :

- `C-x C-f` pour ouvrir un fichier existant ou nouveau ;
- `C-x C-s` pour sauver ;
- sélection à la souris pour copier, clic du milieu pour coller ;
- `F12` pour évaluer une expression Caml (*oplevel phrase*) avec le curseur dans l'expression.

Le double point-virgule `;;` force la fin d'une phrase toplevel. Il n'est pas toujours nécessaire, par exemple si l'on commence la phrase suivante par un `let`, mais on commencera par s'imposer cette sureté.

Pour accéder à la documentation des bibliothèques Caml, il est conseillé d'installer dans Mozilla Firefox la *sidebar* OCaml, disponible à <http://caml.inria.fr/resources/mozilla-sidebar.en.html>

D'autres source de documentation possibles. Pour trouver de la documentation, notamment sur la bibliothèque standard de Caml, vous pouvez utiliser le **site web officiel** : <http://caml.inria.fr/pub/docs/manual-ocaml> Plus particulièrement si vous cherchez de l'aide sur une fonction d'un module donné (disons `List.map` pour ceux qui liront le corrigé) ou si vous cherchez une fonction particulière dans un module donné : [http://caml.inria.fr/pub/docs/manual-ocaml/libref/index\\_modules.html](http://caml.inria.fr/pub/docs/manual-ocaml/libref/index_modules.html) **Sous Unix**, la documentation des modules OCaml est disponible sous forme de pages `man`, tapez par exemple `man List` dans un terminal.

# 1 Premiers pas

Vous allez écrire vos premières expressions Caml et comprendre le système de type, ce qui est primordial pour avancer sans entrave par la suite. A chaque fois que vous évaluez une des phrases suivantes, Caml vous informe du type et de la valeur de l'expression donnée. Par exemple pour `52-10 ; ;`, OCaml répond `- : int = 42`. Notez la syntaxe et lisez attentivement les types pour les phrases suivantes.

```
(),false,'o',"tutu" ; ;
[42],[|false>true|] ; ;
1+1=2 ; ;
3.14 ; ;
3.14 +. 2. ; ;
(+) ; ;
(+.) ; ;
(<>) ; ;
```

► Pourquoi ne peut-on pas écrire `1 +. 2.14` ni `2 * 3.14`, ni même `1 <> "1"` ? Ces expressions ne sont pas *typables*. De même, exécutez les phrases suivantes et comprenez l'erreur de type à la fin.

```
let f x = log (float_of_int x) /. log 2. ; ;
f 1024 ; ;
f (f 64) ; ; <-- invalide !
```

**Attention au piège** pour ceux qui pratiquent d'autres langages : ne pas confondre les opérateurs de comparaison. L'égalité et l'inégalité sont respectivement `=` et `<>`. Les opérateurs `==` et `!=` existent mais correspondent à des comparaisons physiques. Nous n'utiliserons en TP que les comparaisons structurelles. Vous serez par exemple surpris de découvrir que `"a" == "a"` est faux. Les deux chaînes ont le même contenu mais correspondent à deux objets différents en mémoire.

## 1.1 Des fonctions

Nous allons bientôt définir nos premiers programmes sous la forme de fonctions. Voyons rapidement différentes façons de définir ou utiliser une fonction. En Caml il n'y a pas de différence entre `(f(x))` et `(f x)` mais il y en a une entre `(f x y)` et `(f (x,y))`. Comparons les deux définitions suivantes, toutes deux valides :

```
let f1 x y = x+y ; ;
let f2 (x,y) = x+y ; ;
f1 2 3 ; ;
f2 (2,3) ; ;
f1 (2,3) ; ; <-- invalide !
```

- Mais ce n'est pas tout. Évaluez `let f3 = f1 1`, comprenez ce que fait `f3`.  
 Pour mieux comprendre, il faut savoir que `f1` peut aussi s'écrire `fun x -> fun y -> x+y`, ce qui met bien en évidence la possibilité d'*application partielle*. Cette version de la fonction est dite *curryfiée*, et c'est la forme qu'on préférera en général. Dans le même ordre d'idée on notera l'associativité à droite de la flèche : `a -> b -> c = a -> (b -> c)`.

## 1.2 La factorielle

**Pourquoi faut-il déclarer les fonctions récursives ?** En Caml, on procède par définition plutôt que par instantiation. Quand on écrit `let a = ..` il ne s'agit pas de modifier `a` mais de le (re-)définir. On écrit ainsi souvent `let a = .. in let a = ..a.. in ..` pour expliciter des étapes du calcul de `a`. Il est donc pratique que la référence à `a` dans la redéfinition de `a` soit relative à la première définition. C'est pourquoi la récursivité doit être spécifiée explicitement au moyen de `let rec a = ..`, quand on veut que les références à `a` parlent de la définition en cours.

- Définissez la fonction factorielle, et testez-la sur quelques valeurs. Remarquez que les valeurs trop grandes deviennent fantaisistes, négatives ou nulles. Vous devez avoir une vague idée de pourquoi.

**Pour les curieux rapides :** essayez de calculer  $100000!$ . Une implémentation naïve échoue avec le message `Stack overflow during evaluation (looping recursion?)`.

On peut éviter ces erreurs en programmant ses fonctions récursives de façon à ce qu'elles soient *récursives terminales*. Dans ce cas précis ça ne nous avancerait pas, car on a compris que  $100000! = 0$ , mais cette technique sert aussi à rendre les calculs plus rapides. Le corrigé contient une solution récursive terminale.

## 1.3 En finir avec les types

Vous n'en aurez probablement pas vraiment fini de vous débattre avec les erreurs de type, mais voici en tout cas le premier et dernier exercice dédié au sujet.

- Pour chacun des types suivants, définissez une valeur ayant ce type :
  - `'a -> 'a`
  - `'a -> 'b -> 'a`
  - `'a -> 'b -> 'a * 'b`
  - `'a * 'b -> 'a`
  - `('a -> 'b -> 'c) -> 'a * 'b -> 'c`
  - `('a * 'b -> 'c) -> 'a -> 'b -> 'c`
  - `('a -> 'b -> 'c) -> 'b -> 'a -> 'c`

Faites les premiers, mais ne vous attardez pas trop sur cet exercice. Il cache des remarques importantes, mais gardez vous une heure pour la suite qui vous sera plus utile.

## 2 Variants

Pour le premier groupe qui n'a pas encore vu le cours d'introduction, nous verrons ensemble au tableau ce que sont les types variants et comment les utiliser. Dans ce cas, la question suivante est fortement recommandée.

Les types variants sont un outil très pratique pour définir des types de donnée. L'utilisation des filtrages permet de les traiter de façon intuitive, et le compilateur Caml vous aide à ne pas oublier de traiter un cas. Peu de langages fournissent un outil aussi expressif.

► Définir à l'aide de types variants les entiers de Peano, puis les listes. Si vous ne vous sentez pas à l'aise, écrivez la traduction des entiers de Peano vers `int`, ou la fonction `map : ('a -> 'b) -> 'a list -> 'b list` telle que :

```
map f [ a_1; ..; a_n ] = [ f a_1; ..; f a_n ].
```

Sinon vous pouvez passer tout de suite à la suite, plus originale.

### 2.1 Entiers naturels en binaire

On représente les entiers naturels strictement positifs codés en binaire par le type variant suivant : `type entier = H | 0 of entier | I of entier`  
Le nombre 10010 sera représenté par `0(I(0(0(H))))`.

► Définissez les fonctions de conversion entre `entier` et `int` :

```
int_of_entier : entier -> int  
entier_of_int : int -> entier
```

► Définissez l'addition sur les `entier`. Vous n'utiliserez pas de conversion vers `int` mais pourrez définir une fonction auxiliaire de type `entier -> entier -> bool -> entier` dont l'argument booléen indique s'il faut propager une retenue.

Vérifiez quelques additions en utilisant les conversions :

```
let _ =  
  for i = 1 to 16 do  
    for j = 1 to 16 do  
      let k = int_of_entier  
        (somme (entier_of_int i) (entier_of_int j)) in  
      if k <> i+j then  
        Printf.printf "Erreur: %d+%d = %d ?!\n" i j k  
      done  
    done ;  
  Printf.printf "Fin du test.\n"  
;;
```

### 3 Bonus : Amusons-nous avec Pascal

► Écrire une fonction `next : int list -> int list` qui à une ligne du triangle de pascal associe la suivante.

► Á l'aide de la fonction précédemment définie, affichez le triangle de Pascal, représentant les valeurs paires par un ' ' et les impaires par un '#'.  
Vous utiliserez les fonctions suivantes :

```
mod : int -> int -> int
print_char : char -> unit
print_newline : unit -> unit
```

► Affichez ainsi les 31 premières lignes du triangle. Si vous ne reconnaissez pas la figure obtenue, essayez de diminuer la police, plisser les yeux ou afficher plus de lignes.

► Quel type utiliseriez vous pour réécrire la fonction `next` spécialisée pour le calcul des parités?

## 4 Corrigé

### 4.1 La factorielle

La solution simple :

```
let rec fact n = if n = 0 then 1 else n * (fact (n-1))
```

Pour les curieux, voici l'écriture récursive terminale de la factorielle :

```
let factorielle n =  
  let rec aux n acc =  
    if n = 0 then acc else aux (n-1) (n*acc)  
  in  
    acc n 1
```

### 4.2 En finir avec les types

Type	Fonction
'a -> 'a	fun x -> x
'a -> 'b -> 'a	fun x y -> x
'a -> 'b -> 'a * 'b	fun x y -> x,y
'a * 'b -> 'a	fun (x,y) -> x
('a -> 'b -> 'c) -> ('a * 'b -> 'c)	fun f (a,b) -> f a b
('a * 'b -> 'c) -> ('a -> 'b -> 'c)	fun f a b -> f (a,b)
('a -> 'b -> 'c) -> ('b -> 'a -> 'c)	fun f a b -> f b a

### 4.3 Une représentation des entiers naturels

```
let rec int_of_entier = function  
  | H -> 1  
  | 0 n -> 2 * (int_of_entier n)  
  | I n -> 2 * (int_of_entier n) + 1  
;;  
  
let rec entier_of_int n =  
  if n = 1 then H else  
    if n mod 2 = 0 then 0 (entier_of_int (n/2)) else  
      I (entier_of_int (n/2))  
;;  
  
let somme n m =  
  let rec somme n m retenue =  
    match n,m with  
    | 0 n, 0 m -> if retenue then I (somme n m false) else 0 (somme n m false)  
    | I n, I m -> if retenue then I (somme n m true) else 0 (somme n m true)  
    | I n, 0 m
```

```

    | 0 n, I m -> if retenue then 0 (somme n m true) else I (somme n m false)
    | H, H      -> if retenue then I H else 0 H
    | 0 n, H
    | H, 0 n   -> if retenue then 0 (somme n H false) else I n
    | H, I n
    | I n, H   -> if retenue then I (somme n H false) else 0 (somme n H false)
in
    somme n m false
;;

```

#### 4.4 Bonus : Amusons-nous avec Pascal

Ensuite la définition récursive dans le triangle :

```

let next l =
  let rec aux prev = fonction
    | [] -> [1]
    | a::l -> (a+prev)::(aux a l)
  in
    match l with
    | [] -> [1]
    | hd::tl -> 1 :: aux hd tl
;;

let print l =
  List.iter print_char
    (List.map (fun e -> if e mod 2 = 0 then ' ' else '#') l) ;
  print_newline ()
;;

let rec tri n l =
  if n > 0 then begin
    let l = next l in
      print l ;
      tri (n-1) l
    end
  ;;

tri 31 []
;;

```

La même chose, spécialisée pour le calcul des parités. On code la parité par un booléen, `true` pour pair. On redéfinit l'addition sur les parités, on remplace les `1` par des `false` et `e mod 2 = 0` par `e`.

```

let (+) a b = (not a && not b) || (b && a) ;;

```

```

let next l =
  let rec aux prev = function
    | [] -> [false]
    | a::l -> (a+prev)::(aux a l)
  in
    match l with [] -> [false] | hd::tl -> false :: aux hd tl
  ;;

let print l =
  List.iter print_char
    (List.map (fun e -> if e then ' ' else '#') l) ;
  print_newline ()
  ;;

let rec sierpinski n l =
  if n > 0 then begin
    let l = next l in
      print l ;
      tri (n-1) l
    end
  ;;

```