Initiation Caml

aurelien.pardon@ens-lyon.fr

13 septembre 2006

0 *Top-level* OCaml

0.1 Dans un terminal

Dans un terminal (xterm ou quoi que ce soit), il suffit de lancer la commande `ocaml` pour lancer le *top-level*. L'utilitaire *ledit* permet d'utiliser les flèches pour se déplacer dans la ligne de commande ou rappeler une ligne de l'historique, on lancera donc plutôt `ledit ocaml` dans le terminal.

Une fois OCaml lancé, un *prompt* (ou *invite de commandes*) `#` et le curseur s'affichent. On entre le programme et on valide avec *Entrée*. On arrête un calcul avec CTRL-C et on quitte avec CTRL-D.

0.2 Dans *Emacs*

Emacs est un éditeur de texte (on va donc pouvoir sauvegarder les programmes) qui peut interagir avec OCaml. La commande `emacs` permet de lancer cette éditeur avec une interface graphique. Le raccourci pour créer un nouveau fichier est CTRL-X CTRL-F, on spécifie ensuite un nom de fichier avec l'extension `.ml`, *Emacs* lance alors le mode *Tuareg* et son menu.

Dans ce menu, sélectionnez Interactive Mode puis Run Caml Toplevel (raccourci CTRL-C CTRL-S) ; *Emacs* propose de lancer `ocaml` : on valide. La fenêtre du bas qui s'ouvre correspond au top-level décrit précédemment.

On peut alors utiliser le *top-level* dans la fenêtre du bas ou alors écrire dans le fichier (fenêtre du haut) et envoyez le code au top-level pour qu'il l'évalue en faisant Tuareg → Interactive Mode → Evaluate Phrase (raccourci CTRL-X CTRL-E).

On ferme le *top-level* avec CTRL-C CTRL-K.

1 Premiers pas

Rentrez les expressions suivantes dans le *top-level*, et essayez à chaque fois d'interpréter et d'expliquer la réponse d'OCaml.

Remarque : pour vous éviter d'avoir à tout recopier, allez chercher le fichier sur la page `perso.ens-lyon.fr/aurelien.pardon/td0.ml`.

1.1 Evaluation d'expressions

```
51;;

16 - 64;;

32 *
(51+1);;

8 / 3;;

3.14;;

2 * 3.14;;

2. *. 3.14;;

"Bonjour Monde ! lol";;

("mdr",3);;

true;;

true && false;;

2+2 = 4;;

if 3*3 > 8 then "Bravo" else "Bouh";;

if false || not true then "un" else 2;;

2 * (* ceci est un commentaire *) 3;;
```

1.2 Déclaration de variables : `let` et `let ... in`

```
let a = 4;;
a;;

let b = 13 * a;;
b;;

let a = a * a;;
a;;
b;;
```

```
let c = 2 in a * b * c;;
c;;

let a = 21 in a * 2;;
a;;

let (a,b) = (3,4) in a * b;;
```

1.3 Fonctions

```
let carre x = x * x;;
carre 4;;

let fois x y = x * y;;
fois 3 4;;

let fois_3 = fois 3;;
fois_3 14;;

let rec plus x y = if y = 0 then x else plus (x+1) (y-1);;
plus 3 0;;
plus 3 4;;
plus 3 (-1);;   (* boum ! *)

let rec f x = f x;;
f 0;;           (* re-boum ! *)

(fun x -> x*x*x) 4;;

let apply_rev f x y = f y x;;
let g x y = (x,y);;
apply_rev g 64 16;;
apply_rev (fun x y -> x / y) 3 18;;

let rec f x = (x x);;
```

1.4 Listes

```
let est_vide l = (l = []);;

let rec produit l = match l with
  | [] -> 1
  | x::xs -> x * (produit xs);;
produit [2;8;4;2;13];;
```

2 À vous maintenant !

2.1 Fonctions

1. Écrivez récursivement la fonction factorielle `fact : int -> int` telle que `fact n` renvoie `n`!. (Attention si `n` est négatif !)

2.a. Écrivez récursivement la fonction `fibo : int -> int` telle que `fibo n` renvoie le terme u_n de la suite de Fibonacci definie par :

$$\begin{cases} u_0 &= 0 \\ u_1 &= 1 \\ u_n &= u_{n-1} + u_{n-2} \end{cases} .$$

2.b. Essayez de compter le nombre d'appels récursifs à cette fonction lors de l'évaluation de `fibo n`.

2.c. Écrivez une fonction récursive `fibo_aux : int -> int * int` telle que `fibo_aux n` renvoie le couple (u_{n-1}, u_n) en seulement `n` appels récursifs.

2.d. Écrivez une fonction `fibo2 : int -> int` faisant appel à `fibo_aux` pour calculer les termes de la suite de Fibonacci.
Remarque : dans la vraie vie, on utilisera la construction :

```
let fibo2 n =
  let fibo_aux n =
    ...
  in
  ...;;
```

Ainsi, `fibo_aux` n'existe pas en dehors de `fibo2`.

3.a. Écrivez une fonction ayant pour type `'a -> 'a -> 'a`.

3.b. Écrivez une fonction ayant pour type `('a -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c`.

2.2 Manipulation de listes

Le langage OCaml fournit une fonction `failwith : string -> 'a` permettant de lancer des exceptions en cas d'erreur.

Exemple : gestion de la division par zéro :

```
let div x y =
  if y = 0 then failwith "Division par zéro !"
  else x / y;;
div 9 3;;
div 9 0;;
```

1. À l'aide de la construction `match ... with`, écrivez la fonction `hd : 'a list -> 'a` qui renvoie le premier élément de la liste. (Pensez à utiliser `failwith` si la liste est vide.)

2. De même, écrivez la fonction `tl : 'a list -> 'a list` qui renvoie la queue de la liste (c'est-à-dire la liste privée de son premier élément).

3. Écrivez la fonction `length : 'a list -> int` qui calcule la longueur de la liste.

4. Écrivez la fonction `nth : 'a list -> int -> 'a` qui renvoie le $n^{\text{ème}}$ élément de la liste.

5. Écrivez la fonction `rev : 'a list -> 'a list` qui renverse la liste.
Astuce : pensez à écrire une fonction `rev_aux : 'a list -> 'a list -> 'a list`, telle qu'un appel à `rev_aux x::l1 l2` fasse lui-même un appel récursif à `rev_aux l1 x::l2`. La liste `l2` joue ici le rôle d'un *accumulateur*.

6. Écrivez la fonction `append : 'a list -> 'a list -> 'a list` qui concatène les deux listes.

7.a. Écrivez la fonction `map : ('a -> 'b) -> 'a list -> 'b list` telle que `map f [a1; ...; an]` renvoie la liste `[f a1; ...; f an]`.

7.b. Écrivez la fonction `map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list` telle que `map2 f [a1; ...; an] [b1; ...; bn]` renvoie `[f a1 b1; ...; f an bn]`, ou lève une exception si les listes n'ont pas la même taille.

8. Écrivez la fonction `fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a` telle que
`fold_left f a [b1; ...; bn]` renvoie `f (... (f (f a b1) b2) ...) bn`.
Exemple : `fold_left (fun s x -> 10*s+x) 0 [1; 2; 3; 4]` renvoie `1234`.

9. Écrivez la fonction `fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b` telle que
`fold_right f [a1; ...; an] b` renvoie `f a1 (f a2 (... (f an b) ...))`.
Exemple : `fold_right (fun x s -> 10*s+x) [1; 2; 3; 4] 0` renvoie `4321`.

10. En utilisant `fold_left` ou `fold_right`, écrivez la fonction `for_all : ('a -> bool) -> 'a list -> bool` telle que `for_all p [a1; ...; an]` renvoie `true` si et seulement si tous les éléments la liste satisfont le prédicat `p`, soit `(p a1) && (p a2) && ... && (p an)`.

11. Même question pour la fonction `exists : ('a -> bool) -> 'a list -> bool` qui renvoie `true` si et seulement s'il existe un élément de la liste qui satisfait `p`.

2.3 Tri fusion d'une liste

1. Écrire une fonction `split : 'a list -> 'a list * 'a list` qui partage une liste `l` en deux listes `l1` et `l2` telles que les tailles de `l1` et `l2` ne diffèrent que d'un au maximum.

2. Écrire une fonction `merge : int list -> int list -> int list` qui prend en argument deux listes ordonnées d'entiers `l1` et `l2` et renvoie une liste ordonnée `l` contenant tous les éléments de `l1` et de `l2`.

3. Enfin, écrire une fonction `merge_sort : int list -> int list` utilisant les deux fonctions précédentes pour trier une liste d'entier.

Indication : l'algorithme *merge sort* (ou tri fusion) repose sur une approche *diviser pour régner* : si une liste contient au plus un élément, elle est trivialement ordonnée. Par contre, si elle contient deux éléments ou plus, il suffit de la partager en deux listes plus petites (`split`) qui seront alors chacune triée par un appel récursif à `merge_sort`, puis enfin de fusionner les listes résultantes (`merge`).