Initiation Caml – TD1 : Premiers pas

Sylvain.Perifel@ens-lyon.fr

19 septembre 2006

0 Prise en main du *top-level* OCaml

0.1 Dans un terminal

Pour lancer OCaml à partir d'un terminal, il suffit de taper la commande `ledit ocaml` dans le terminal en question (si vous ne savez pas ce qu'est un terminal ou comment en ouvrir un, demandez au TDman).

Remarque : le *top-level* OCaml est juste `ocaml`, mais le lancer avec `ledit` permet de bénéficier de plein de petites fonctionnalités très utiles (gestion des flèches du clavier, pour éditer une ligne ou remonter dans l'historique des commandes).

Une fois la commande `ledit ocaml` entrée, vous vous retrouvez devant un *prompt*, ou invite de commandes : `#`. Vous pouvez alors taper directement et évaluer des expressions OCaml.

Pour interrompre un calcul, il suffit de faire Ctrl-C. Pour quitter le *top-level*, Ctrl-D.

0.2 Dans Emacs

Cette solution est bien plus pratique car elle vous permettra de sauver vos programmes dans un fichier. Cependant, pour pouvoir utiliser OCaml dans Emacs, il faut avoir installé SoftÉlèves (`/soft/eleves/bin/install-softeleves`). Si vous ne l'avez pas encore fait, si vous souhaitez savoir comment faire ou si vous ne savez pas ce qu'est SoftÉlèves, demandez au TDman.

Lancez donc tout d'abord Emacs, et ouvrez un nouveau fichier, avec l'extension `.ml` (typiquement `td1.ml` pour aujourd'hui). Emacs va reconnaître l'extension et charger le mode correspondant (pour les plus curieux d'entre vous, il s'agit du mode *Tuareg*) : un nouveau menu Tuareg apparaît.

Dans ce menu, sélectionnez Interactive Mode puis Run Caml Toplevel (vous pouvez aussi faire Ctrl-C Ctrl-S pour aller plus vite). Emacs vous demande alors quel est le programme à lancer, et vous propose `ocaml` : c'est ce qu'on veut, donc validez. Emacs charge alors le *top-level* OCaml dans la fenêtre du bas.

Vous pouvez donc interagir directement avec le *top-level* comme s'il était lancé depuis un terminal, ou bien taper les expressions OCaml dans votre fichier `td1.ml` (qui est dans la fenêtre du haut), et les envoyer au top-level par le menu Tuareg, Interactive Mode puis Evaluate Phrase (ou Ctrl-X Ctrl-E).

Le menu Tuareg vous permet aussi d'interrompre un calcul ainsi que de quitter le *top-level* (Ctrl-C Ctrl-K).

1 Premiers pas

Commençons doucement : 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 à l'adresse `http://perso.ens-lyon.fr/sylvain.perifel/td_caml_01.ml`. Dans ce répertoire se trouve aussi le fichier d'exemples du prof, `debut_caml.ml`.

1.1 Évaluation d'expressions

```
51;;

16 - 64;;

32 *
(51+1);;

8 / 3;;

3.14;;

2 * 3.14;;

2. *. 3.14;;

"Hello world!";;

("haha",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 ! rappelez-vous comment interrompre le calcul *)

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 $u_{\texttt{n}}$ 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 $(u_{\texttt{n}-1}, u_{\texttt{n}})$ 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 que 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 si 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`).