Université Paris Diderot — Programmation Fonctionnelle
Math-Info/Informatique/Linguistique — Année 2012-2013

TP n°1 - Correction

Premiers pas en Ocaml

Les exercices marqués d'une étoile [⋆] peuvent être laissés pour la fin – ou faits chez vous.

Pour lancer l'interpréteur OCaml sous `emacs` :

- ouvrez un nouveau fichier `tp1.ml` (l'extension `.ml` est nécessaire),
- dans le menu `Tuareg`, dans le sous-menu `Interactive Mode`, choisissez l'entrée `Run Caml Toplevel`
- confirmez le lancement de `ocaml` par un retour-chariot.

Chaque expression entrée dans la fenêtre de `tp1.ml` peut être évaluée en se plaçant sur un caractère quelconque de l'expression (avant "`;;`") , puis : ou bien par `Evaluate phrase` dans le sous-menu `Interactive Mode` du menu `Tuareg` d'emacs ; ou bien par `ctrl-x, ctrl-e`.

Expressions et types

Il n'y a pas en Caml de notion d'instruction au sens usuel : Caml ne manipule que des *expressions*. La notion d'exécution est remplacée par celle d'*evaluation*. L'évaluation d'une expression entrée entraîne l'affichage de sa valeur et de son type. Le type de l'expression est déduit à la volée par l'interpréteur avant son évaluation :

`2 + 2;;`	de valeur `4`, de type `int`
`"abcd"^"ef";;`	de valeur `"abcdef"`, de type `string`
`0.34 +. 23.12;;`	de valeur `23.46`, de type `float`
`(40 + 2,"ab"^"cd");;`	de valeur `(42,"abcd")`, de type `int * string`

Il n'y a pas de variables au sens usuel, mais on peut donner un *nom* à une valeur d'expression, quel que soit son type :

`let x = 2 + 5;;`	`x` désigne à présent la valeur `7`
`let y = x + 1;;`	`y` désigne à présent la valeur `8` (la valeur de `7 + 1`)
`x + y + 1;;`	de valeur `7 + 8 + 1 = 16`

`let ...;;` est appelé une *de nition*. Sur le même modèle, on peut définir des *fonctions* - ces fonctions sont *applicables* à des arguments du bon type :

`let succ x = x + 1;;`	de type `int -> int` (attendant un `int` et renvoyant un `int`)
`let plus x y = x + y;;`	de type `int -> (int -> int)` (attendant deux `int`)
`succ 42;;`	de valeur `42 + 1 = 43`
`plus 3 (succ 2);;`	de valeur `3 + (2 + 1) = 6`

`let` ... `in` ... permet de nommer *localement* une valeur d'expression. Les parenthèses ci-dessous ne sont pas indispensables :

```
let u = 2 in (u + 3);;
let u = 1 in (let v = 2 in  (u + v));;
```

Enfin, `let` ... `and` ... ... `and` ... permet d'effectuer plusieurs définitions simultanées :

```
let x = 10 and y = 42;;
let w = (let u = 2 and v = 3 in (u + v));;
```

Dans cet exemple, la définition de `w` est globale, et les définitions de `u` et `v` sont locales à cette définition.

Exercice 1 Écrire et évaluer quelques expressions Ocaml de votre choix.

Exercice 2 Donner le type des expressions suivantes. Vérifier le résultat sur machine.

- `let f x = x + 1 in f 6;;`
- `let f x = x + 1 in f;;`
- `let s = "a" in let f t = t^s in f;;`
- `let g h x = h (x+1) - 1 in g;;`

Correction :

```
# let f x = x + 1 in f 6;;
- : int = 7
# let f x = x + 1 in f;;
- : int -> int = <fun>
# let s = "a" in let f t = t^s in f;;
- : string -> string = <fun>
# let g h x = h (x+1) - 1 in g;;
- : (int -> int) -> int -> int = <fun>
```

Booléens et conditionnelle

Expressions de type `bool`

Les expressions de type `bool` sont de la forme :

- *constantes booleennes* :
  `true`
  `false`
- *connections logiques* :
  e_1 `&&` e_2
  e_1 `||` e_2
  `not` e
  (et, ou, négation de) avec e_1, e_2, e de type `bool`.
- *comparaisons* :
  e_1 *cmp* e_2
  où *cmp* est l'un des opérateurs `=`, `<>`, `<`, `<=`, `>`, ou `>=`, et e_1 et e_2 sont de même type.

On ne peut pas comparer des fonctions, mais on peut comparer des entiers, des chars, des listes, des n-uplets, etc. Tout type non fonctionnel est implicitement muni d'un ordre, et la valeur d'une comparaison entre éléments de même type ne dépend que de cet ordre. Pour les types numériques, il s'agit de l'ordre usuel sur les nombres, pour les `char` l'ordre alphabétique, pour les `string` l'ordre lexicographique, etc.

Conditionnelle

L'expression suivante a pour valeur la valeur de e_1 si l'expression booléenne c vaut `true`, et celle de e_2 si c vaut `false` :

`if` c `then` e_1 `else` e_2

Exercice 3 Écrire une fonction `is_major` prenant en argument une année de naissance (un entier), et renvoyant `true` si la différence entre l'année courante (écrivez-la en dur dans le code) et cette année de naissance est au moins égale à 18, `false` sinon.

Correction :

```
let is_major a = 2010 - a >= 18;;
```

Exercice 4 [⋆] La fonction `max : 'a -> 'a -> 'a` est prédéfinie en `Caml` : appliquée à x et y de même type, elle renvoie le maximum de x et y (suivant l'ordre implicite sur les valeurs de ce type, voir ci-dessus).

Écrire votre propre fonction `max`, en ne vous servant que de `if then else` et des opérateurs de comparaisons. Votre fonction doit être du même type que le `max` prédéfini, c'est-à-dire de type `'a -> 'a -> 'a`.

A partir de cette fonction, définir :

- une fonction `max_couple` de type `` `a * `a -> `a ``, prenant en argument un couple `(x,y)`, et renvoyant le maximum de ses composantes,
- une fonction `max_triple` de type `` `a * `a * `a -> `a `` prenant en argument un triplet `(x,y,z)`, et renvoyant le maximum de ses composantes.

Correction :

```
let max x y = if x > y then x else y;;
let max_couple  (x,y) = max x y;;
let max_triple (x,y,z) = max x (max y z);;
```

Portée des noms

Exercice 5 [⋆] Prévoir le résultat fourni par l'interpréteur OCaml après chacune des commandes suivantes, entrées dans l'ordre.

```
#let x = 2;;
#let x = 3
 in  let y = x + 1
      in x + y;;
#let x = 3 and y = x + 1
 in  x + y;;
```

Correction :

```
# let x = 2;;
val x : int = 2
# let x = 3
  in let y = x + 1
     in x + y;;
- : int = 7
# let x = 3 and y = x + 1
  in x + y;;
- : int = 6
```

La deuxième commande utilise la définition de l'identificateur x fraichement introduit (celle de $x = 3$), alors que la troisième commande utilise la première définition de x où elle vaut 2.

Pourquoi les deux dernières commandes ne fournissent-elles pas le même résultat ? Expliquer à présent le comportement suivant :

```
#let x = 3;;
x : int = 3
#let f y = y + x;;
f : int -> int = <fun>
#f 2;;
- : int = 5
#let x = 0;;
x : int = 0
#f 2;;
- : int = 5
```

Correction : La définition de f utilise la première définition de x. Redéfinir x dans la suite n'a aucun effet sur la définition de f qui est déjà fixée.

Récurrence

La construction

`let rec` *nom arg_1 arg_2 ...*`=` *expression*`;;`

permet de se servir du nom d'une fonction dans sa propre définition, c'est-à-dire de définir des fonctions récursives. Par exemple :

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

Noter que `and` permet de définir simultanément plusieurs fonctions mutuellement récursives :

```
let rec f x = if x > 0 then (g (x - 1)) else 1
    and g x = if x > 0 then (f (x - 1)) else 0
;;
```

Exercice 6 Rappelons que la somme des entiers de 0 à n peut être définie récursivement par $\Sigma(0) = 0$, et $\Sigma(n) = n + \Sigma(n-1)$ si $n > 0$. Écrire cette fonction en OCaml. Que donne cette fonction appliquée à un nombre négatif ?

Correction :

```
let rec somme n =
  if n <= 0 then 0 else n + (somme (n-1))
;;
(* n <= 0 au lieu de n = 0, pour eviter *)
(* un bouclage si n est negatif         *)
```

Exercice 7 Écrire en OCaml la fonction récursive définie par $f(0) = 1$, $f(1) = 1$ et $f(n) = f(n-1) + f(n-2)$.

Correction :

```
let rec fibo n =
  if n <= 1 then 1
  else (fibo (n-1)) + (fibo (n-2))
;;
(* meme remarque *)
```

Exercice 8 Écrire la fonction `ack` qui calcule la fonction mathematique f definie comme suit :

$$\begin{aligned} f\ 0\ n &= n+1 \\ f\ (m+1)\ 0 &= f\ m\ 1 \\ f\ (m+1)\ (n+1) &= f\ m\ (f\ (m+1)\ n) \end{aligned}$$

Évaluez $(f\ 0\ 0), (f\ 1\ 1), (f\ 2\ 2), (f\ 3\ 3), (f\ 4\ 4)$

Exercice 9 [$\star$] Écrire une fonction `binome` p n renvoyant C_n^p, donné par :

$$C_n^p = C_{n-1}^p + C_{n-1}^{p-1} \text{ si } 1 \leq p \leq n$$
$$C_n^0 = 1$$
$$C_n^p = 0 \text{ si } p > n$$

Correction :

```
(* en supposant n et p positifs : *)
let rec binome p n =
  if p = 0 then 1 else
  if p > n then 0 else
    binome p (n-1) + binome (p-1) (n-1)
;;
```

Exercice 10 [$\star$] Dans un bois, il y a des lapins et des renards. Au matin du premier jour, il y a 37 lapins et 5 renards. Chaque jour, à midi, chacun des renards mange 2 lapins – mais il reste toujours au moins 10 lapins qui se sauvent toujours. Chaque lapin qui n'est pas mangé fait naître 1 autre lapin au matin du jour suivant. A chaque fois que 3 lapins sont mangés, un nouveau renard naît au matin du jour suivant. Finalement, chaque soir, 1/10 des renards (arrondi à l'entier inférieur le plus proche) meurt de vieillesse.

Écrire trois fonctions mutuellement recursives `lapins`, `renards`, `manges` de type `int -> int`, telles que `(lapins n)`, `(renards n)`, `(manges n)` renvoient respectivement, les populations de lapins et de renards le matin après les naissances éventuelles, et le nombre de lapins mangés à midi, au bout de `n` jours. Évaluez ces fonctions pour les premières deux semaines. Changez les nombres de lapins et/ou renards du premier jour et observez les changements.

Correction :

```
let rec lapins n =
  if n = 0 then 37 else
     2 * ((lapins (n - 1)) - (manges (n - 1)))
     (* chaque lapin non mange la veille fait naitre un nouveau lapin *)

and renards n =
  if n = 0 then 5 else
     let rprec = (renards (n - 1)) in
     rprec - (rprec/10) + ((manges (n - 1))/3)
     (* un 10eme des renards de la veille sont morts                  *)
     (* pour chaque triplet de lapins mange, un nouveau renard est ne *)

and manges n =
     let ln = (lapins n) in       (* nombre lapins au matin *)
     let r2 = (renards n) * 2 in  (* s'il y en a suffisamment,          *)
                                  (* nombre de lapins qui seront manges *)
     if  ln - r2 < 10 then ln - 10 else r2
     (* s'il n'y a pas assez de lapins pour en manger r2, *)
     (* on les mange tous sauf 10                         *)
;;
```

Introduction aux types fonctionnels

Les fonctions en Caml sont en fait des valeurs comme les autres : "la fonction prenant en argument un entier `x` et renvoyant l'entier `x + 1`" s'écrit littéralement

`fun x -> x + 1;;` de type `int -> int` (*i.e.* attendant un `int` et renvoyant un `int`)

Tout comme les fonctions définies à l'aide d'un `let`, ces fonctions (anonymes) sont applicables à des arguments du bon type :

`fun x -> x + 1;;`	de type `int -> int`
`(fun x -> x + 1) 3;;`	de valeur `4`, de type `int`
`fun x -> (fun y -> x + y);;`	de type `int -> (int -> int)`
`((fun x -> (fun y -> x + y)) 1) 3;;`	de valeur `4`, de type `int`.

Les `fun` imbriqués peuvent être écrits avec les notations abrégées suivantes :

```
fun x -> (fun y -> (fun z -> x + y + z));;      ou
fun x y z -> x + y + z;;
```

Le `let` permet d'autre part de donner un nom à ces fonctions :

`let plus = fun x y -> x + y;;`	`plus` désigne la fonction `fun x y -> x + y`
`plus 3 7;;`	de valeur `3 + 7 = 10`

La notation `let plus x y = x + y;;` n'est rien de plus qu'une *notation abregee* pour

```
let plus = (fun x -> (fun y -> x + y));;      équivalent à
let plus = (fun x y -> x + y);;               équivalent à
let plus x = (fun y -> x + y);;               équivalent à
let plus x y = x + y;;
```

Il n'y a aucune différence essentielle entre `let x = 42` et `let f = (fun x -> x + 42)` (que l'on peut écrire `let f x = x + 42`) : on peut dire que `f` est une *fonction* pour souligner que c'est de type fonctionnel (`int -> int`).

Exercice 11 Puisque les fonctions sont des valeurs comme les autres, rien n'empêche d'écrire une fonction prenant en argument une autre fonction. Donner le type des expressions suivantes. Vérifier le résultat sur machine.

- `let plus_minus h x = h (x + 1) - 1;;`
- `let sum_funct f g x =  (f x) + (g x);;`
- `let double f x = f (f x);;`
- `let rec times n f x = if n = 0 then x else f (times (n-1) f x);;`

Évaluez les expressions suivantes :

- `plus_minus (fun x -> x * x) 3;;`
- `sum_funct succ (fun m -> m * 2) 3;;`
- `double plus_minus (fun m -> m * 2) 4;;`
- `double (plus_minus (fun m -> m * 2)) 4;;`
- `times 4 (fun m -> m * 2) 5;;`

Exercice 12 Écrire une fonction `egal_triplet` prenant en argument une fonction `f` de type `int -> 'a`, et renvoyant `true` si les valeurs `(f 0)`, `(f 1)` et `(f 2)` sont égales.

Correction :

```
let egal_triplet f =
  let v0 = (f 0) in
  v0 = (f 1) && v0 = (f 2)
;;
(* noter la definition locale, pour eviter calculer deux fois (f 0) *)
```

Exercice 13 [⋆] On souhaite généraliser la fonction précédente. Écrire une fonction `egal_segment` prenant en argument une fonction `f` de type `int -> 'a`, un entier `n` que l'on supposera au moins égal à `0`, et renvoyant `true` si les valeurs `(f 0)`, ..., `(f n)` sont deux à deux égales.

Correction :

```
(* la version suivante recalcule (f 0) a chaque appel : *)
let egal_segment f n =
  (n < 1) || ((f 0 = f n) && (egal_segment f (n - 1)))
;;

(* la version suivante est optimisee. (f 0) est calcule  *)
(* au plus une fois, et une fonction recursive locale se *)
(* charge de tester les autres valeurs                   *)
let egal_segment f n =
  (n <= 0) ||
  (let v0 = (f 0) in
   let rec test p =
    (p < 1 || ((f p = v0) && (test (p - 1))))
   in test n)
;;
```

Exercice 14 Écrire une fonction prenant en argument une fonction `f` et un entier `n`, et renvoyant $\sum_{i=0}^{n} f(i)$. Quel sera son type ? Utiliser cette fonction pour définir une nouvelle fonction calculant la somme des $n+1$ premiers carrés.

Correction :

```
let rec somme_f f n =
  if n < 0 then 0
  else (f n) + (somme_f f (n-1))
;;
let somme_carre = somme_f (fun x -> x * x)
;;
(* ou, ce qui revient au meme : *)
let somme_carre n = somme_f (fun x -> x * x) n
```