Université Paris Diderot — Programmation Fonctionnelle
L2 Math-Info/L3 Informatique/M1 Linguistique — Année 2011-2012

TP n°1

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'*évaluation*. 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 :

`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`

Par abus de langage, on dit que `x` est une *variable*. `let ...;;` est appelé une *définition*. Sur le même modèle, on peut definir 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 [⋆] 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;;
```

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
```

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;;`

Booléens et conditionnelle

Expressions de type `bool`

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

- *constantes booléennes* :
  `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 Ecrire une fonction `est_majeur` 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.

Exercice 4 Par convention, une année est bissextile : si elle est divisible par 4 et non divisible par 100, ou, si elle divisible par 400. En vous aidant de l'opérateur `mod` sur les entiers, écrivez la fonction `est_bissextile` qui prend comme argument une année (un entier), renvoyant `true` si elle et bissextile et `false` sinon.

Exercice 5 [⋆] 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).

Ecrire votre propre fonction `max`, en ne vous servant que de `if then else` et des opérateurs de comparaisons. Votre fonction doit être 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_triplet` de type `` `a * `a * `a -> `a `` prenant en argument un triplet `(x,y,z)`, et renvoyant le maximum de ses composantes.

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 $n > 0$. Ecrire cette fonction en OCaml. Que donne cette fonction appliquée à un nombre négatif ?

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

Exercice 8 [$\star$] Ecrire une fonction `binome` p n renvoyant C_n^p, donné par :

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

$$C_n^0 = 1$$

$$C_n^p = 0 \text{ si } p > n$$

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 abrégée* 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`) : l'abus de langage consistant à dire que `x` est une *variable* et `f` une *fonction* est seulement dû au fait que `x` est de type de base (`int`) et `f` de type fonctionnel (`int -> int`). Dans les deux cas, on se contente de donner un nom à une valeur d'expression.

Exercice 9 Puisque les fonctions sont des valeurs comme les autres, rien n'empêche d'écrire une fonction prenant en argument une autre fonction. On peut par exemple écrire :

```
let somme_fonct f g x =  (f x) + (g x);;
```

où `f`, `g` sont ici des *paramètres de fonctions*, utilisés à droite comme des fonctions ordinaires. La fonction `somme_fonct` peut être appliquée à deux fonctions "réelles" en écrivant par exemple :

```
somme_fonct (fun n -> n + 1) (fun m -> m * 2) 3;;
```

soit `(fun n -> n + 1) 3 + (fun m -> m * 2) 3`
soit `(3 + 1) + (3 * 2)`
soit `4 + 6`
soit `10`

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