

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'*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 avant son évaluation :

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

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 :

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

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

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

`let ... in ...` permet de nommer *localement* une valeur d'expression. Les associations ainsi créées sont provisoires, et ne durent que le temps d'évaluer une expression donnée. Dans les deux exemples ci-dessous, les associations créées pour `u`, puis pour `u` et `v`, seront perdues une fois chaque expression évaluée. Les parenthèses 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. Testez, bien sûr, chacune des fonctions ci-dessous après l'avoir écrite.

1. Définir une fonction `carre` attendant un `int` et renvoyant son carré.
 2. Définir une fonction `perimetre` attendant un `float` représentant le rayon d'un cercle et renvoyant le périmètre de ce cercle. On rappelle que le périmètre d'un cercle de rayon r vaut $2\pi r$, et que π vaut environ 3.14159267. Pour définir la valeur approximative de cette constante, servez-vous d'une définition locale.
 3. Définir une fonction `bis` attendant une chaîne de caractères et renvoyant cette chaîne concaténée à elle-même. Par exemple, `"ab"` donnera `"abab"`. L'opérateur de concaténation de chaînes de caractères s'écrit `^`.
 4. Écrire une fonction `huit_fois` attendant une chaîne de caractères et renvoyant huit exemplaires de cette chaîne concaténés à la suite. On notera que :
 - en concaténant `"ab"` à elle-même, on obtient `"abab"` (deux fois `"ab"`)
 - en concaténant `"abab"` à elle-même, on obtient `"abababab"` (quatre fois `"ab"`)
 - en concaténant `"abababab"` à elle-même, on obtient `"abababababababab"` (huit fois `"ab"`)Servez-vous de cette propriété pour construire le résultat de cette fonction à l'aide d'une suite de définitions locales.
- (★) Sauriez-vous écrire une seconde version de cette fonction ne se servant que de la fonction `bis` ci-dessus, sans définitions locales, et déléguant à `bis` toutes les concaténations?

Correction :

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

let perimetre r =
  let pi = 3.1415927 in
  2. *. pi *. r
;;

let bis s = s^s;;

(* premiere version*)
let huit_fois s =
  let s2 = s^s in
  let s4 = s2^s2 in
  let s8 = s4^s4 in s8 (* ou directement : s4^s4 *)
;;

(* seconde version - noter que l'évaluation va de droite a gauche *)
let huit_fois s = bis (bis (bis s));;
```

Le problème de la portée des noms

Exercice 2. 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 fraîchement 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 ?

Expliquez 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.

Booléens et conditionnelle

Expressions de type bool

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

- *constantes booléennes* : `true` `false`
- *expressions de comparaisons* : $e_1 \text{ cmp } e_2$
où *cmp* est l'un des opérateurs `=`, `<`, `<=`, `>`, ou `>=`, et e_1 et e_2 sont de même type.
- *expressions construites avec les connecteurs logiques* :
 $e_1 \ \&\& \ e_2$
 $e_1 \ || \ e_2$
`not` e
 (et, ou, négation de) avec e_1, e_2, e de type `bool`.

On ne peut pas comparer des fonctions, mais on peut comparer des entiers, des chars, des listes, des n-uplets, etc. Pour les types numériques, l'ordre utilisé est l'ordre usuel sur les nombres, pour les char l'ordre alphabétique, pour les `string` l'ordre lexicographique, pour les n-uplets l'ordre lexicographique sur leurs composantes, etc.

Exercice 3. Comme indiqué ci-dessus, toutes les expressions de comparaison sont de type `bool`, donc s'évaluent en `true` ou en `false`. En déduire une fonction `est_nul : int -> bool` prenant un argument un entier et renvoyant `true` si cet entier est nul, `false` sinon.

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 e1 else e2
```

Exercice 4. Écrire une fonction `msg_nul : int -> string` prenant un argument un entier et renvoyant la chaîne de caractères `"nul"` si cet entier est nul, la chaîne `"non nul"` 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 .

1. Testez la fonction `max` prédéfinie sur différents types d'arguments : `int`, `float`, `char`, `string`. . . essayez-la aussi sur des couples de valeurs, de types éventuellements distincts.
2. En l'appelant `my_max`, donnez votre propre implémentation de la fonction `max` – écrite à l'aide d'un `if`, et sans vous servir de `max`. Votre fonction doit être de même type que le `max` prédéfini, et donner les mêmes résultats.

A partir de la seule fonction `max` (ou `my_max`, peu importe) et sans `if`, définir :

- une fonction `max_triple` prenant trois arguments x , y , z et renvoyant le plus grand,
- une fonction `max_quadruple` prenant quatre arguments x , y , z , t et renvoyant le plus grand.

Correction :

```
let my_max x y = if x > y then x else y;;
let max_triple x y z = max x (max y z);;
let max_quadruple x y z t = max (max x y) (max z t);;
```

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, la fonction calculant la factorielle d'un entier peut être définie ainsi :

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

Exercice 6. Évaluez `fact 2` à la main comme fait au cours pour *gcd*.

Exercice 7. 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 $n > 0$. Sur le modèle de la fonction factorielle ci-dessus, écrire cette fonction *somme* en OCaml. Vérifiez que `somme 200` donne **20100**. N'essayez pas de le faire, mais que risque-t-il de se produire si cette fonction est appliquée à un nombre négatif ? Comment éviter ce problème ?

Correction :

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

Exercice 8. La fonction de Fibonacci est définie par $fibo(0) = 1$, $fibo(1) = 1$ et $fibo(n) = fibo(n - 1) + fibo(n - 2)$ si $n \geq 2$. Définir cette fonction en OCaml. Vérifiez que `fibo 10` donne 89.

Évaluez `fibo 4` à la main. Qu'est-ce que vous constatez ?

Correction :

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

L'évaluation à la main : On remarque qu'on évalue $f(n)$ plein de fois pour certain n .

Exercice 9. Deux employés, A et B , travaillent dans un entrepôt qui contient des colis et doit être vidé. Les deux employés travaillent en alternance. Chacun utilise une stratégie d'évidement : Employé A à chaque pas enlève à chaque fois un seul colis tandis qu'employé B enlève deux colis, si le nombre de colis qui restent est pair, sinon il en enlève 1. Ecrire deux fonctions (une par employé) mutuellement récursive qui calculent après combien de pas l'entrepôt contenant n colis est vide. Vérifiez que l'entrepôt contenant 11 colis est vide après 10 pas, si employé A commence et après 11 pas, si employé B commence.

Correction :

```
let rec pasA n =  
  if n = 0 then 0 else 1 + pasB (n-1)  
and pasB n =  
  if n = 0 then 0  
  else let nbis = if n mod 2 = 0 then n-2  
                  else n-1  
        in 1 + pasA nbis;;
```

Exercice 10. (*) La fonction d'Ackermann est une fonction à deux arguments entiers, définie par :

$$\begin{aligned} f(0, n) &= n + 1 \\ f(m, 0) &= f(m - 1, 1) && \text{si } m > 0 \\ f(m, n) &= f(m - 1, f(m, n - 1)) && \text{si } m > 0 \text{ et } n > 0 \end{aligned}$$

Définir une fonction **ack** représentant cette fonction en **OCaml** – au lieu de lui donner un couple d'arguments, ses deux arguments lui seront donnés successivement : on écrira **ack n m** au lieu de $f(n, m)$. Évaluez **ack 0 0**, puis **ack 1 1**, puis **ack 2 2**, etc.

(*** Évaluez **ack 4 1** à la main.

Correction :

```
let rec ack x y =  
  if x = 0 then y+1  
  else if y = 0 then ack (x-1) 1  
  else ack (x-1) (ack x (y-1));;
```

Exercice 11 (Expressions). Qu'affiche le toplevel Caml quand on entre l'une après l'autre les expressions suivantes? Donner le type de chaque expression ou définition (ou le message d'erreur si l'expression est mal typée), ainsi que sa valeur,

1. `2. < sqrt 5.;;`

Correction :

- : bool = true

2. `(fun x -> x ^ "hello");;`

Correction :

- : string -> string = <fun>

3. **let** x = 3 **in** 2.3 + x;;

Correction :

```
let x = 3 in
  2.3 + x;;
```

Error: This expression has **type** float but an expression was expected **of type** int

4. **let** x = 3 **in** 2.3 +. (float)x;;

Correction :

```
- : float = 5.3
```

5. **let** x = 3 **in** int_of_float (2.3 +. (float)x);;

Correction :

```
- : int = 5
```

6. **let** impl x y = not x || y;;

Correction :

```
val impl : bool -> bool -> bool = <fun>
```

7. impl **true** **false**;;

Correction :

```
- : bool = false
```

8. impl (impl **true** **false**)**false**;;

Correction :

```
- : bool = true
```

9. **let** rec f i s =

```
  if i=0 then "" else s^(f (i-1) s);;
```

Correction :

```
val f : int -> string -> string = <fun>
```

10. **let** g x = x+1 **in**

```
  f (g 3) "hello";;
```

Correction :

```
- : string = "hellohellohellohello"
```

11. g 3;;

Correction :

Error: Unbound value g

12. **let** map_string f s =

```
  let rec map_string_aux f s i =
```

```
    if i >= String.length s then ""
```

```
    else (f s.[i])^(map_string_aux f s (i+1))
```

```
  in
```

```
    map_string_aux f s 0
```

```
;;
```

Correction :

```
val map_string : (char -> string) -> string -> string = <fun>
```

13. map_string (fun x -> String.make 2 x) "Hello";;

Correction :

```
- : string = "HHeelllloo"
```