Programmation Fonctionnelle : TD1

EFREI - Paris

Principes de base du langage Ocaml

*
* *

Préalable

Pour programmer en Ocaml, vous pouvez utiliser l'éditeur intéractif en ligne https://try.ocamlpro.com/.

Certains passages de ce TD sont directement empruntés au livre "*Le Langage Caml*" (deuxième édition) de P. Weis et X. Leroy. Celui-ci est disponible en intégralité sur *Moodle* pour les plus curieux.

Consignes

Vous pouvez réaliser les devoirs en binôme. Pour chaque TD et TP, il vous est demandé de déposer un compte-rendu sur Moodle. Le compte-rendu devra être sous la forme d'un fichier Ocaml nom1-nom2_nomSeance.ml unique où:

- nom1 et nom2 sont les noms du binôme (juste votre nom si le travail est fait seul).
- nomSeance désigne le nom et numéro de la séance (ex. TD1, TP3, etc.).

Comme par exemple le fichier moreau-lirzin_TD1.ml.
Également, le fichier devra suivre la même chronologie que les différents énoncés et contiendra:

- Les ***spécifications*** et le ***code source*** accompagné de ***commentaires explicatifs***.
- Les ***réponses*** aux questions de l'énoncé sous forme de commentaires.

Enfin, il est fortement conseillé de ***lire*** attentivement le document... ;)

Premiers pas en Ocaml

Mes premières commandes en Ocaml

L'évaluation d'une expression entraîne l'affichage de son type et de sa valeur.
Testez les expressions suivantes et copiez le résultat dans votre compte-rendu en les expliquant si nécessaire (si le résultat vous surprend...):

```
2 + 2;;
"il etait " ^ "une fois";;
29 / 4;;
29 /. 4;;
29.0 /. 4.0;;
29 mod 4;;
0.5 + 20.2;;
0.5 +. 20.2;;
(42, 'A');;
fst(42, 'A');;
snd(42, 'A');;
```

```
2 = 2;;
2 <> 2;;
'a' = 'A';;
false || true;;
true && true;;
(2 = 3) || ('A' <> 'a' && true);;
1 ;; (* Commentaires *)
[];;
[1 ; 2 ; 3];;
```

Mes premières définitions en Ocaml

Les définitions (ou *liaisons*) permettent de désigner une valeur par un nom de manière *globale*. De même qu'en mathématiques on écrit : « Soit x la somme des nombres 1, 2 et 3. », on écrit en Ocaml (« soit » qui se traduit par `let` en anglais):

```
let x = 1 + 2 + 3;;
```

Maintenant que le nom x est défini, il est utilisable dans d'autres calculs ; par exemple, pour définir le carré de x, on écrirait:

```
let x2 = x * x;;
```

Une définition n'est pas modifiable : un nom donné fait toujours référence à la même valeur, celle qu'on a calculée lors de la définition du nom. Le mécanisme du `let` est donc fondamentalement différent du mécanisme d'affectation des autres langage de programmation, comme en Java.
Il est impossible de changer la valeur liée à un nom ; on peut seulement redéfinir ce nom par une nouvelle définition, donc un nouveau `let`. Testez les expressions suivantes et commentez-les:

```
let a = 1;;
let translation = fun x -> x + a;;
translation 3;;
let a = 2;;
translation 3;;
```

Il est également possible de définir temporairement des variables. Ces définitions temporaires sont les définitions *locales* (par opposition aux précédentes qui sont évaluées globalement), qui disparaissent à la fin de l'évaluation de la phrase dans laquelle elles se trouvent.
Ces définitions locales ne sont donc plus valides après le calcul de l'expression qui les suit (après le mot-clé `in`, qui signifie « dans »):

```
let x = 20 in x * 4;;
x;;
```

La définition locale s'affranchie également du type actuel de la variable, par exemple:

```
let x = "Le langage " and x2 = " Ocaml" in x ^ x2;;
```

Notez également que les définitions multiples sont possibles et consistent en une simple succession de définitions séparées par le mot-clé `and` (qui signifie « et »).

Mes premières fonctions en Ocaml

Les fonctions forment les ***constituants élémentaires*** des programmes en Ocaml. Un programme n'est rien d'autre qu'une collection de définitions de fonctions, suivie d'un appel à la fonction qui déclenche le calcul voulu.
La définition d'une fonction suit la formulation mathématique classique. Notons que contrairement aux fonctions anonymes celles-ci possède un nom ! Par exemple « Soit *succ* la fonction définie par $succ(x) = x + 1$. » se traduit en Ocaml par:

```
let succ(x) = x + 1;;
```

L'application d'une fonction à son argument suit aussi la convention mathématique:

```
succ(2);;
```

N.B : Les parenthèses peuvent être omises lors de la dénition/utilisation des fonctions. Essayez !

Une fonction peut aussi définir localement une autre fonction. Par exemple, pour définir la fonction `puissance4` qui élève son argument à la puissance quatre, il est naturel d'utiliser la formule $x^4 = \left(x^2\right)^2$, donc d'élever au carré le carré de l'argument. Pour cela, on définit localement la fonction `carre` et on l'utilise deux fois:

```
let puissance4 x =
    let carre y = y * y in (* def locale d'une fonction *)
    carre (carre x);;
puissance4 2;;
```

De même, il est possible de définir des fonctions à plusieurs arguments. Par exemple la fonction moyenne de deux nombres a et b se traduit telle que:

```
let moyenne a b = (a +. b) /. 2.0;;
```

Attention ! Cette définition n'est pas équivalente à la fonction suivante:

```
let moyenne2 (a, b) = (a +. b) /. 2.0;;
```

La première définition est une fonction *curryfiée* tandis que la seconde est non-curryfiée, elle prend un couple de nombres en argument. Vous pouvez observer la différence au niveau de la signature des fonctions. Testez et expliquez les appels suivants:

```
moyenne 3.0 2.0;;
moyenne 3.0;;
(moyenne 3.0) 2.0;;
moyenne (3.0, 2.0);;
moyenne2 (3.0, 2.0);;
moyenne2 3.0;;
```

Il existe ainsi plein de manières d'écrire une même fonction. En outre, les écritures suivantes sont équivalentes. Vérifiez-le en les testant:

— Par utilisation successive de fonctions anonymes.

```
let f = fun x1 -> (fun x2 -> fun x3 -> x1 * x2 - x3);;
f 2 4 6;;
```

— Par déclarant une unique fonction anonyme.

```
let f = fun x1 x2 x3 -> x1 * x2 - x3;;
f 2 4 6;;
```

— En définissant une fonction localement (ce qu'on fait en général):

```
let f x1 x2 x3 = x1 * x2 - x3;;
f 2 4 6;;
```

Fonctions anonymes

Les fonctions anonymes sont des expressions comme les autres et sont appliquées à des arguments. Cependant, il s'agit de fonction qui ne possèdent pas de liaision / définition. Elles sont donc évanescentes et disparaissent une fois déclarée car elles ne sont liées à aucun nom.
Testez les expressions suivantes et *expliquez* le résultat dans votre compte-rendu:

```
fun x -> x + 1;;
(fun x -> x + 1) 3;;
fun x -> (fun y -> x + y);;
(fun x -> (fun y -> x + y) 1) 3;;
fun (x, y) -> x;;
(fun (x, y) -> x) (42, 'A');;
```

N.B : On précise que `fun` est l'abrégé du mot-clé `function` et désigne une fonction anonyme.

Expressions conditionnelles

Ocaml fournit une construction pour faire des calculs qui dépendent d'une condition : le classique « `if... then... else...` ».
Formellement, ceci correspond au calcul « Si *cond* alors exp_1 sinon exp_2. », où la condition *cond* est une valeur booléenne et les expressions exp_1 et exp_2 sont de même type ; ainsi, la valeur de l'expression est exp_1 si *cond* est vraie et exp_2 sinon.
On peut illustrer cette utilisation en représentant la fonction sgn : $\mathbb{Z} \to \{-1, 1\}$ qui retourne le signe d'un nombre (on retournera 1 si x vaut 0).

$$\text{sgn}(x) = \begin{cases} 1 & \text{si } x \geq 0 \\ -1 & \text{sinon} \end{cases}$$

La fonction en Ocaml s'écrit:

```
let sgn x = if( x >= 0) then 1 else (-1);;
sgn (-2);;
sgn 42;;
```

Écrire la fonction valeur absolue $|\cdot| : \mathbb{Z} \to \mathbb{N}$ nommée `abs` et définie telle que:

$$|x| = \begin{cases} x & \text{si } x \geq 0 \\ -x & \text{sinon} \end{cases}$$

Exception `failwith`

L'utilisation du mot `failwith` provoque l'arrêt de l'évaluation en déclenchant une exception. Par exemple, on considère la fonction factorielle suivante:

```
let rec fact n = match n with
  0 -> 1 (* Si n = 0, alors on retourne 1 *)
| 1 -> 1 (* Si n = 1, alors on retourne 1 *)
| n -> n * fact (n-1);;
```

Cette fonction est syntaxiquement et sémantiquement correcte mais ne se termine pas pour l'appel suivant:

```
fact (-1);;
```

car la factorielle n'est pas définie pour les nombres négatifs.
On peut utiliser `failwith` pour signaler une erreur ou une entrée non prévue par le programme. Par exemple:

```
let rec fact n = match n with
  0 -> 1 (* Si n = 0, alors on retourne 1 *)
| 1 -> 1 (* Si n = 1, alors on retourne 1 *)
| n when (n < 0) -
> failwith "ERR. : Nombre negatif" (* Si n < 0, attention... *)
| n -> n * fact (n-1);;
```

Dans ce cas, on constate que la commande suivante produit une erreur et est alors bien gérée par notre programme.

```
fact (-1);;
```

Écrire une fonction `division` qui prend deux nombres entiers a et b en argument et retourne la division entière de a par b. On utilisera la notion de `failwith` pour gérer les cas de division par 0.

À votre tour...

1. Écrire la *spécification* et le *code* d'une fonction `max` qui prend en argument deux nombres entiers x et y et retourne le plus grand des deux.
2. Écrire la *spécification* et le *code* d'une fonction `maxi3` qui prend en arguments trois nombres entiers x, y et z et retourne le plus grand des trois.

   On pourra pour se faire s'aider de la fonction `max` préalablement codée.
3. (Falcultatif) Comment pourrait-on généraliser notre démarche pour calculer le maximum de 4 entiers, le maximum de 5 entiers, le maximum de n entiers ?