Colle 1 : Premiers pas avec le langage Ocaml

14 septembre

1 Les types de base :

Nous allons commencer par tester un certain nombre d'instructions afin de comprendre les règles d'utilisation des entiers, des flottants (réels) et des booléens. Demandez vous ce que va faire le langage avant d'exécuter.

1.1 Entiers et flottants :

```
(* Ceci est un commentaire Ocaml, il est donc inutile de le saisir ! *)
1 + 2;;
1.0 +. 2.0;;
1-6;;
1 +. 2;;
1.0 + 2;;
1.0 +. float_of_int 2;;
5./.3.;;
5/3;;
6*.4;;
3*2+1;;
31 mod 3;;
min_int;;
max_int;;
max_int + 1 ;;
max_float;;
min_float;;
10.*max_float;;
```

On retiendra que :

1. Ocaml est fortement typé :
2. Entiers et flottants sont sujets aux dépassements de capacité.
3. Ocaml "devine" seul le type des objets grâce au contexte, c'est ce qu'on appelle l'inférence de type.

Notez ici les commandes que vous venez d'apprendre :

1.2 Booléens

```
not true;;
1 = 1;;
1 = 2;;
not (2 < 3);;
1 <> 1;;
1 <> 2;;
1 < 1;;
1 <= 1;;
true || false;;
true && false;;
(not (1 = 2)) && (3/0=1);;
true || (1/0 = 1);;
true && (1/0 = 1);;
false && (1/0=1);;
(not (1 = 2)) || (3/0=1);;
```

On retiendra que :

1. Ocaml refuse de diviser par zéro.
2. L'évaluation des formules logiques est paresseuse :

Notez ici les commandes que vous venez d'apprendre :

2 Ocaml est un langage fonctionnel

Plutôt que d'écrire des suites d'instructions, la programmation en Ocaml favorise l'utilisation de fonctions au sens mathématique, ainsi vous devez définir des fonctions de la forme :

```
let f x = x*x;;
```

ou

```
let g x y = x+y;;
```

ou

```
let h x y = x+x;;
```

Pour utiliser ces fonctions vous devez ensuite les évaluer en faisant une requête du type `f 3;;`.
On remarquera l'absence d'utilisation des parenthèses.
Tester les commandes suivantes :

```
let g x = x*x;;
g n;;
g 2;;
g -2;;
g (-2);;
(* En Ocaml les fonctions sont des objets comme les autres donc g -2 est interprété comme le
souhait d'effectuer la différence entre l'identificateur g et la valeur 2. On mettra donc
des parenthèses autour des nombres négatifs. *)
```

```
let f  = fun x-> x*x;;
f 5;;
f f 5;;
f (f 5);;
(f f) 5;;
f;;
```

▷ **Question 1.** Ecrire une fonction qui prend en entrée deux entiers naturels n et p et calcule leur somme. Evaluer cette fonction. Ecrire une fonction qui prend en entrée trois entiers n, p et q et qui renvoie le produit de q avec la somme de n et de p (vous pouvez réutiliser la fonction précédente). Evaluer cette fonction. ◁

Tester les commandes suivantes :

```
let h = fun x y -> x+y;;
(* On remarque que h est une fonction qui attend 2 entiers et retourne un entier *)
h 2 3;;
h;;
h 2;;
(h 2) 3;;
```

```
let est_egal x y = (x=y);;
est_egal;;
est_egal 2 3;;
est_egal (true) (false);;
est_egal 2. 2;;
est_egal 2. 2.;;
```

```
let h = fun x g -> g x;;
h 2 (fun x->x*x);;
h;;
h 2;;
(h 2) (fun x->x-1);;
```

Observez la signature des fonctions proposées. On retiendra que :

1. Curryfication des fonctions de plusieurs variables :
2. Ocaml est un langage qui permet le polymorphisme :
3. On appelle fonction d'ordre supérieur une fonction qui admet une fonction comme l'un de ses arguments. Il n'y a pas de limite dans la construction de ce type d'objet.

Afin de compléter ce panorama de construction de fonctions voici plein d'écritures de la fonction test qui attend un entier en entrée et qui retourne vrai si ce dernier est le nombre 2 ou le nombre 4 et faux sinon.

```
let test x =
    if (x=4) then
       true
    else
       if (x=2) then
           true
       else
           false;;
```

```
let test x =
    if (x=4) then
        true
    else if (x=2) then
        true
    else
        false;;
```

```
let test x = if (x=4) then true else if (x=2) then true else false;;

let test x =
     if (x=4 || x=2) then
        true
     else
        false;;

let test x = (x=4 || x=2);;

let test x=x=4||x=2;;

(* A présent vous avez du remarquer que les espaces et les retours à la ligne n'ont pas une
grande importance en CAML et qu'il est plutôt désagréable de lire un code non indenté.
Pensez-y lorsque vous programmerez... *)

let test x = match x with
| 4 -> true
| 2 -> true
|_ -> false;;
```

Le dernier exemple utilise le filtrage qui est une spécificité de Ocaml qui rend la lecture et l'écriture d'un programme très confortables.

```
let f x =
match x with
|0->1
|x->0;;
```

Que fait cette fonction ?

▷ **Question 2.** Ecrire les fonctions qui prennent en entrée deux booléens et qui renvoient respectivement le OR et le XOR de ces deux booléens. ◁

Tester les commandes suivantes :

```
let est_positif x = match x with
|x when x>0 -> true
|_ -> false
;;
let est_positif x = match x with
|x when x>0 -> true
|x when x<=0 -> false
;;
```

On remarque que Ocaml nous adresse un warning dans le dernier exemple, c'est-à-dire un message d'avertissement nous signifiant qu'il n'est pas certain que le filtrage opéré par la fonction match couvre l'ensemble des cas possibles. En effet, Ocaml peut vérifier la syntaxe des programmes qu'on lui soumet mais pas la logique sous jacente, c'est à vous de vous assurer que vous avez bien logiquement traité tous les cas ou d'utiliser comme dernier cas

```
|_
```

qui signifie "tout le reste".

3 Variables et références :

Tester les commandes suivantes :

```
let n = 11;;
n+1;;
n;;
n := n+1;;
n;;
let n = 23 in n+1;;
n;;
```

Comme vous pouvez le constater, il n'est pas possible de modifier n. D'ailleurs l'instruction let n = 11;; n'est pas une affectation mais plutôt une instruction disant à OCAML "Dorénavant, remplace les occurences de l'identificateur n par 11 ". Pour modifier une valeur il va falloir spécifier à OCAML que nous avons besoin d'un espace mémoire à cet effet. La fonction ref valeur permet de demander à OCAML un tel espace à remplir par valeur que nous pourrons utiliser. Ainsi en écrivant let n = ref 11;;, OCAML associe à l'identificateur n une référence vers la valeur 11. n désignera donc la "case" mémoire mais la valeur contenue dans la mémoire pourra être obtenue à l'aide de la commande !n.

Tester les commandes suivantes :

```
let n = ref 11;;
n;;
n := n+1;;
!n;;
n := !n+1;;
n;;
incr n;;
n;;
decr n;;
n;;
(* Notez que les fonctions incr et decr ont besoin de l'adresse de n et non de sa valeur *)
```

En résumé :

La commande `let x = a in ();;` définit la variable locale x. Elle signifie qu'au nom x on doit substituer a dans toute l'expression Ocaml qui suit in. En dehors de cette expression, x n'est plus définie, c'est une variable locale. Il faut considérer cette commande comme une définition et non pas comme une affectation.

Si au cours d'un programme on veut introduire une variable qui va être modifiée, on doit utiliser la commande suivante :

```
let (nom) = ref (valeur initiale) in (code)
```

Lorsque l'on veut modifier la valeur de la variable référence, on utilise la commande :

```
x := !x +1;
```

4 Opérations impératives : boucles et conditions

On favorise largement l'utilisation des fonctions et de la récursivité lorsque l'on programme en Ocaml mais on peut tout de même utiliser des conditions et des boucles à l'aide de la syntaxe suivante :

```
if (condition) then (operation1) else (operation2)
```

ou

```
for i = (depart) to (arrivee) do
(corps de la boucle)
done;
```

ou

```
while (condition) do
(corps de la boucle)
done;
```

▷ **Question 3.**

1. Ecrire une fonction qui prend en entrée un entier naturel n et qui calcule la somme de tous les entiers inférieurs ou égaux à n.
2. Ecrire une fonction qui prend en entrée un entier naturel n et qui calcule $n!$.
3. Calculer la somme des chiffres de l'écriture en base 10 de $30!$ (factorielle de 30) en utilisant une boucle while. *On pourra utiliser la fonction factorielle de la question précédente.*
4. Écrire une fonction `somme2b` qui prend en entrée un couple d'entiers naturels (m, n) et qui renvoie la somme des carrés des entiers pairs compris (au sens large) entre m et n.
5. En utilisant une boucle, écrire une fonction `puissance` qui prend en entrée un couple d'entiers naturels (x, n) et qui renvoie x^n.

◁

A ce stade, vous pouvez légitimement vous posez des questions sur l'utilisation du symbole point virgule : la fin d'un programme ou d'une fonction est marquée par deux points virgule et on sépare les différentes expressions d'un programme itératif à l'aide d'un seul point virgule. D'un point de vue pragmatique, on met toujours un point virgule après le done et si notre corps de boucle contient plusieurs instructions alors on doit les séparer avec des points virgules.

▷ **Question 4. [Entiers triangulaires]** On dit qu'un entier naturel n est triangulaire s'il existe un entier naturel k tel que $n = \frac{k(k+1)}{2}$.

1. Donner tous les entiers triangulaires inférieurs ou égaux à 10.

2. Écrire une fonction `premiers_entiers_triangulaires` qui prend en entrée un entier naturel n et qui renvoie la liste des n premiers entiers triangulaires.

3. Écrire une fonction `triangulaire` qui prend en entrée un entier naturel n et qui renvoie si n est un entier triangulaire ou non.

4. Écrire une fonction `entiers_triangulaires` qui prend en entrée un entier naturel n et qui retourne la liste des entiers triangulaires compris entre 0 et n.

◁

5 Récursivité

5.1 Récursivité simple :

En programmation on dit qu'une fonction est récursive lorsqu'elle apparaît dans sa propre définition. Cela est indiqué par le symbole rec, qui rend le nom de la fonction disponible à l'intérieur de sa définition (sans lui, le nom ne serait disponible qu'après la définition). Exemple :

```
let rec f n =
match n with
|0-> 1
|n->n*f(n-1);;
```

▷ **Question 5.** Que fait cette fonction ? Combien de multiplications fait-elle ? ◁

Tester le code suivant :

```
let factorielle n = match n with
|0 -> 1
|_ -> n*factorielle (n-1)
;;
```

▷ **Question 6. [Suite de Syracuse]**
Ecrire une fonction de signature syracuse : int -> int -> int = <fun>, qui à deux entiers naturels n et $u_0 > 0$ associe la valeur de u_n sachant que :

$$\forall n \in \mathbb{N}, u_{n+1} = \begin{cases} \frac{u_n}{2} & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{sinon.} \end{cases}$$

Vous écrirez cette fonction une première fois avec une boucle FOR, puis sous forme d'une fonction récursive.
◁

▷ **Question 7.** Ecrire une fonction récursive qui prend en entrée deux entiers naturels n et p et qui calcule $\binom{n}{p}$. ◁

▷ **Question 8.** Ecrire une fonction récursive qui prend en entrée deux entiers n et x et qui calcule x^n. Quelle est la complexité de votre algorithme ? Pouvez vous trouver un algorithme effectuant $\log_2(n)$ multiplications ? ◁

▷ **Question 9.** Ecrire une fonction récursive qui calcule le PGCD de deux entiers, à l'aide du même principe construire une fonction qui calcule les coefficients de Bezout de ces deux entiers. ◁

▷ **Question 10.** Écrire une fonction `fibo` récursive tel que `fibo n` calcule le n-ième terme de la suite de Fibonacci, avec $u_0 = 0$ et $u_1 = 1$. ◁

La fonction `fibo` que vous venez d'écrire a une complexité désastreuse, alors qu'elle ressemble fort à `fact`. Pourquoi ? Analysons la pile de récursivité de ces deux fonctions. Pour `fact`, elle a la forme :

$$\texttt{fact n} \to \texttt{fact (n - 1)} \to \cdots \to \texttt{fact 0}$$

Pour `fibo`, elle a une forme plus compliquée :

$$\texttt{fibo n} \to \begin{matrix}\texttt{fibo (n-2)}\\ \texttt{fibo (n-1)}\end{matrix} \to \begin{matrix}\texttt{fibo (n-2)}\\ \texttt{fibo (n-3)}\\ \texttt{fibo (n-2)}\end{matrix} \to \ldots$$

▷ **Question 11.** Quelle est la complexité de `fibo n` en fonction de n ? Proposer une amélioration en utilisant des couples, quelle est sa complexité ? ◁

5.2 Récursivité mutuelle

Pour que deux fonctions récursives puissent s'appeler mutuellement, il faut qu'elles soient définies en même temps comme le montre l'exemple ci-après. D'ailleurs, que font ces deux fonctions ?

```
let rec mystere1 n =
match n with
|0 -> true
|_ -> mystere2 (n-1)
and mystere2 n =
match n with
|0 -> false
|_ -> mystere1 (n-1)
;;
```