Tutoriel OCAML

Hugues Cassé <casse@irit.fr>

Ce document est inspiré du site http://sdz.tdct.org/sdz/ocaml-pour-les-zeros.html. Vous pouvez vous y référer pour plus de détails. Pour une documentation complète d'OCAML, on se référera à http://caml.inria.fr/pub/docs/manual-ocaml/.

Les exemples donnés dans le document pourront être entrées dans l'interprète OCAML tout au long de la lecture.

OCAML est un langage:

- fonctionnel - il n'y a pas de variable : tout est expression et les 2 seules structures de contrôle sont l'appel de fonction et la sélection ;
- fortement typé - toute expression et opération a un type bien précis et aucune conversion implicite n'est réalisée ;
- automatiquement typé - les types des constantes sont automatiquement déduits et les types des expressions et des fonctions sont automatiquement inférés ;
- incluant un ramasse-miettes - signifie que nous n'avez pas besoin de gérer la mémoire.

1 Premiers pas

On peut utiliser OCAML comme un interpréteur en ligne de commande :

```
> ./ocaml
Objective Caml version X.XX.X
#
```

A partir, on peut lancer un calcul :

```
# 15 * 3 ;;
- : int = 45
```

Non seulement il réalise le calcul et l'affiche mais calcule également le type. Le int supporte les opérations classique +, -, *, /. Si on veut utiliser des nombres flottants, le typage fort nous impose d'utiliser les opérateurs +., -., *. et /. :

```
# 12.4 +. 3.14 ;;
- : float = 15.54000000
```

On peut aussi concaténer des chaînes de caractères :

```
# "Hello, " ^ "World!"
- : string = "Hello, World!"
```

Comparer des valeurs :

```
# 3 > 4 ;;
- : bool = false
```

Ou faire des calculs logiques avec des booléens (`&&` - ET, `||` - OR, `not`) :

```
# (true && true) || false;;
- : bool = true
```

2 Définition de valeur

Il est possible de donner un nom au résultat d'une expression qu'on utilisera plus tard :

```
# let x = 4 ;;
val x : int = 4
```

On pourra utiliser ce symbole plus tard :

```
# x * x ;;
- : int = 16
```

On peut également réaliser une définition qui sera connu seulement dans l'expression qui la suit :

```
# let r = 3 in r * r ;;
- : int = 9
# r ;;
Error: Unbound value r
```

3 Définition de fonction

Une des structures les plus utilisé est la définition de fonction :

```
# let f x = 2 * x + 1 ;;
val f : int -> int = <fun>
```

Ici, on obtient une nouvelle définition pour l'identificateur f qui est de type fonction int -> int : il prend en entrée un entier et renvoie un entier. On peut l'appeler de la manière suivante :

```
# f 3 ;;
- : int = 7
```

En OCaml, on n'a pas besoin de mettre des parenthèses pour passer des paramètres à une fonction. En revanche, si un des paramètres est le résultat d'un calcul, le calcul doit être mise entre parenthèses :

```
# f (1 + 1);;
- : int = 5
```

Si on ne met pas les parenthèses, on obtient (f 1) + 1 :

```
# f 1 + 1 ;;
- : int = 4
```

Exercice :

1. définir la valeur étant égale à 3.1415
2. définir la fonction `peri` qui prend en paramètre un rayon `r` et renvoie le périmètre du cercle de rayon `r`.

Une autre manière d'écrire une fonction est d'utiliser une notation λ-lambda calcul :

```
# let f = fun x -> 2 * x + 1;;
val f : int -> int = <fun>
```

On peut se rendre compte qu'on obtient le même résultat.

NOTE : vous allez maintenant être amené à écrire des fonctions plus complexes mais l'éditeur OCAML n'est pas très ergonomique pour cela. Je vous propose de les mettre dans un fichier .ml de la manière suivante.

1. Depuis la ligne de commande, lancez votre éditeur préféré pour créer un fichier test.ml (n'oubliez le & final !) :
   `> EDITEUR test.ml &`
2. Entrez vos définitions de fonction à l'intérieur, par exemple :
   `let print s = print_string s`
3. Revenez à la ligne de commande et tapez :
   `> ocaml`
4. Dans l'interpréteur, chargez votre fichier :
   `#use "test.ml";;`
5. Appelez votre fonction de la manière habituelle :
   `print "Hello, World !\n"`

4 Les conditions

L'instruction `if` … `then` … `else` … permet de réaliser une conditionnelle :

```
# if 3 < 4 then "petit" else "grand";;
- : string = "petit"
```

On remarquera néanmoins que (a) le then et le else s'appliquent sur des expressions et renvoient des valeur (de type `string` ici) et (b) par conséquent, qu'il n'existe pas de `if-then` sans `else` !

La fonction définie ci-dessous calcule la valeur absolue d'un nombre entier :

```
# let abs x = if x < 0 then -x else x ;;
val abs : int -> int = <fun>
# abs 3;;
- : int = 3
# abs (-4);;
- : int = 4
```

Exercice
Ecrire une fonction signe qui prend en entier un paramètre `x` entier et renvoie -1 si `x` est négatif, 0 si `x` est nul, +1 si `x` est positif.

5 Les fonctions récursives

En OCAML, il n'y a pas d'instruction de répétition : il faut utiliser des fonctions récursives. Une fonction récursive est déclarée avec `let rec` :

```
# let rec sum n = if n = 0 then 0 else n + sum (n - 1);;
val sum : int -> int = <fun>
# sum 10;;
- : int = 55
```

La fonction sum fait la somme des nombres de 0 à n. On notera que, pour faire une fonction récursive, il est nécessaire d'avoir une condition avec, au moins, un cas trivial et un cas récursif.

Exercice

1. Ecrire la fonction `fact n` qui calcule factorielle de n.
   On pourra la tester avec fact 5, fact 1 et fact 0.
2. Ecrire la fonction `pgcd` a b qui calcule le PGCD(a, b) avec la méthode d'Euclide :
   PGCD(a, 0) = a
   PGCD(a, b) = PGCD(b, a) si a < b
   PGCD(a, b) = PGCD(b, a mod b) sinon
3. Ecrire la fonction `fib` n qui calcule le nombre de Fibonacci sur n générations en se rappelant que :
   fib(0) = 0
   fib(1) = 1
   fib(n) = fib(n - 1) + fib(n - 2)
   Calculer fib(5), fib(1), fib(0), fib(10).
4. Ecrire une fonction `puis` x n qui la calcule x^n.
   On pourra le tester pour calculer 4^3, 3^3, 3^1 2^0.
5. Ecrire une fonction `puis_rap` x n qui calcule de manière beaucoup plus rapide x^n en remarquant que :
   $x^0 = 1$
   $x^{2n} = x^n * x^n$ (on ne calculera x^n qu'1 fois !)
   $x^n = x^{n-1} * x$

6 Encore plus sur les fonctions

On peut définir des fonctions dans des fonctions et la fonction aura la visibilité sur toutes les définitions de la fonction externe : les paramètres et les définitions.

```
# let puis x n =
            let rec comp n =
            if n = 0 then 1 else x * (comp (n - 1)) in
      comp n;;
val puis : int -> int -> int = <fun>
# puis 2 7;;
- : int = 128
```

On notera également qu'une fonction peut être chargée partiellement pour former une fermeture de la fonction :

```
# let add x y =  x + y;;
val add : int -> int -> int = <fun>
# let inc = add 1 ;;
val inc : int -> int = <fun>
# inc 5;;
- : int = 6
```

Observez bien que le type de `add` est `int -> int -> int` et que le type de `inc` est `int -> int` : c'est le type de `add` auquel on a fourni un premier paramètre, 1. La fonction ne s'exécute qu'une fois que tous les paramètres sont fournis. Dans certains, cela peut être très utile en évitant du code glu entre diverses fonctions.

On peut également passer des fonctions en paramètre pour réaliser des fonctions d'ordre supérieur :

```
# let rec apply f x n =
    if n = 1 then x else f x (apply f x (n - 1));;
    val apply : ('a -> 'a -> 'a) -> 'a -> int -> 'a = <fun>
```

On notera que les types de `f` et de `x` dépendent d'un type générique 'a qui sera remplacé par un type précis lors de l'appel de apply :

```
# let add x y = x + y;;
val add : int -> int -> int = <fun>
# apply add 4 5;;
- : int = 20
```

A partir de la fonction d'addition, la fonction `apply` nous permet de réaliser une multiplication : on répète 5 fois la somme de 4 !

```
# apply (fun x y -> x * y) 2 7;;
- : int = 128
```

On n'a même pas besoin de définir un nom pour la fonction passée en paramètre (ici la multiplication) en utilisant les fonctions anonymes avec `fun`. Ici, on transforme la multiplication en calcul de puissance.

Exercice

1. Ecrire une fonction `forall` p n qui teste si un prédicat p est vrai pour tous les entiers entre 1 et n. On pourra la tester avec :
   ```
   forall (fun x -> x >= 0) 100
   forall (fun x -> x mod 2 = 1) 100
   forall (fun x -> x * x = x) 100
   forall (fun x -> x < 0) 100
   ```
2. Ecrire une fonction `exists` p n qui teste le prédicat p est vrai pour au moins 1 des entiers entre 0 et n.

7 Les tuples

En OCAML, les paramètres sont toujours passés par valeur : il n'est donc pas possible de renvoyer un résultat par les paramètres. Par contre, il est facile de construire des tuples (paire ou ordre supérieur pour renvoyer plusieurs résultats à la fois) :

```
# (3, "ok");;
- : int * string = (3, "ok")
```

Pour accéder au résultat d'un tuple, on peut utiliser la reconnaissance de modèle avec le let :

```
# let t = (3, "ok");;
```

```
val t : int * string = (3, "ok")
# let (i, s) = t;;
val i : int = 3
val s : string = "ok"
```

Le tuple vide () représente la valeur nulle de type nul unit :

```
# ();;
- : unit = ()
```

8 Les listes

En OCAML, il existe un type ‘a `list` qui permet de gérer de type générique ‘a. Tous les éléments d’une liste doivent avoir le même type, ‘a.
Pour définir une liste, on utilisera la syntaxe suivante :

```
# let l = [1; 2; 3];;
val l : int list = [1; 2; 3]
```

La liste vide est simplement définie par :

```
# let vide = [ ];;
val vide : 'a list = [ ]
```

Il est possible de construire une nouvelle liste en ajoutant un élément en tête en écrivant :

```
# 0 :: l;;
- : int list = [0; 1; 2; 3]
```

Il est possible d’examiner une liste en utilisant la reconnaissance de modèle fournie par l’instruction let :

```
# let h::t = l;;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[ ]
val h : int = 1
val t : int list = [2; 3]
```

h (head) va recevoir le premier élément de la liste et t (tail) va recevoir la suite de la liste. Cependant, une telle opération peut échouer si la liste est vide d’où l’avertissement affiché par OCAML.

Une autre manière d’utiliser les liste passe par l’instruction de reconnaissance de modèle :

```
# match l with
  | [ ] -> -1
  | t::_ -> h;;
- : int = 1
```

Cette instruction permet de prendre en compte de manière exhaustive tous les cas (retour de -1 en cas de liste vide) et d’ignorer la suite de la liste (en utilisant le caractère _) si elle n’est pas utile pour mon calcul.

Ainsi, on pourra calculer la longueur d’une liste avec la fonction :

```
# let rec long l =
      match l with
      | [ ] -> 0
      | _::t -> 1 + (long t);;
val long : 'a list -> int = <fun>
# long [1; 2; 3];;
- : int = 3
# long ["a"; "b"; "c"; "d"];;
- : int = 4
```

On notera que la fonction ainsi obtenue fonctionne pour tout type de liste (son type dépend du type générique ‘a list).

Il existe de nombreuses fonctions pour travailler sur les listes documentées sur la page http://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html. Pour utiliser ce module, nommé List, on tapera :

```
# open List;;
# List.hd l;;
- : int = 1
```

Exercice

1. Ecrire la fonction make n qui construit une liste formée des n premiers entiers: [n-1; n-2; …; 2; 1; 0].
   Testez là avec make 10 et make 0.
2. Ecrire la fonction sum l qui fait la somme des entiers contenus dans la liste l.
   On pourra la tester avec sum (make 10).
3. Ecrire la fonction makerev n qui construit une liste formée des n premiers entiers dans l’ordre [0; 1; …; n-1; n-2].
4. Ecrire la fonction map f l qui, si l = [e_1; e_2; …], renvoie [f(e_1); f(e_2); …].
   Testez là avec map (fun x -> x + 1) (make 10).
5. Ecrire une fonction fold f i l qui renvoie i si la liste l est vide, sinon si l = [e_1; e_2; …], elle renvoie f(e_n, f(e_{n-1}, … f(e_2, f(e_1, i)) …)).
   Utilisez là sur fold (fun x y -> x + y) 0 (make 10).

9 Les types somme

Ils permettent tout d’abord de faire des énumération de valeurs (le nom des valeurs énumérées doit avoir une majuscule en première lettre) :

```
# type color = RED | GREEN | BLUE;;
type color = RED | GREEN | BLUE
# GREEN;;
- : color = GREEN
```

Mais chaque élément de l’énumération peut supporter une valeur :

```
# type data =
      | STR of string
```

```
      | INT of int;;
type data = STR of string | INT of int
# let x = STR "ok";;
val x : data = STR "ok"
```

On peut utiliser le match pour exploiter un type somme :

```
match x with
  | STR s -> print_string s
  | INT i -> print_int i
 ;;
ok- : unit = ()
```

Ici, le type de retour est `unit` et la valeur `()` car les fonctions `print_string` and `print_int` affichent leur paramètre mais ne renvoient aucun valeur. Par contre, on peut voir que la chaîne “ok” de x a été affichée !

On peut faire des types somme récursifs. Cela est pratique pour définir des structures de données récursives comme un arbre d’entier :

```
# type tree =
      | LEAF of int
      | NODE of tree * tree;;
type tree = LEAF of int | NODE of tree * tree
```

La valeur énumérée `LEAF` prend comme paramètre 1 entier et la valeur énumérée `NODE` prend une paire de sous-arbres : on a donc un arbre binaire. La fonction ci-dessous fait la somme de tous les entiers feuilles de l’arbre :

```
# let rec sum t =
     match t with
     | LEAF i -> i
     | NODE (t1, t2) -> (sum t1) + (sum t2);;
     val sum : tree -> int = <fun>
```

On pourra l’utiliser de la manière suivante :

```
# let my_tree = NODE (LEAF 3, NODE (LEAF 1, LEAF 10));;
val t : tree = NODE (LEAF 3, NODE (LEAF 1, LEAF 10))
# sum my_tree;;
- : int = 14
```

Exercice

1. Ecrire la fonction count t qui calcule le nombre de feuilles de l’arbre t.
   On pourra la tester avec `count my_tree`
2. Ecrire la fonction max t qui renvoie le maximum des valeurs des feuilles de l’arbre t.
   On pourra la tester avec `max my_tree`
3. Ecrire la fonction leaves t qui renvoie une liste formée des entiers feuilles de l’arbre t.
   On pourra la tester avec `leaves my_tree`
4. Ecrire la fonction forall p t qui renvoie vraie si toutes les feuilles de l’arbre sont vraies pour le prédicat p.

On pourra la tester avec `forall (fun x -> x >= 0) my_tree`
et `forall (fun x -> x mod 2 = 1) my_tree`.

10 Compilation

Le langage OCAML peut interprété mais peut-être aussi compilé en utilisant le compilateur ocamlc.

Utilisez votre éditeur préféré pour éditer un fichier hello.ml où vous taperez :

```
print_string "Hello, World!\n"
```

Sauvegardez le fichier et, en ligne de commande, compilez le avec :

```
> ocamlc hello.ml -o hello
```

Vous pouvez maintenant l'exécuter comme n'importe fichier exécutable :

```
> ./hello
Hello, World!
```

Dans ce cas là, il est possible d'avoir un programme formé de plusieurs fichiers : chaque fichier formera un module.

Avec votre éditeur préféré, créez un fichier nommé out.ml et tapez :

```
let print = print_string
```

Sauvez le et compilez le avec :

```
> ocamlc -c out.ml -o out.cmo
```

On peut maintenant réécrire le fichier hello.ml de la manière suivante :

```
Out.print "Hello, World!\n"
```

On remarquera que OCAML va chercher un fichier nommé out.cmo pour résoudre le nom du module Out et va vérifier que le module Out contient bien la fonction print.
Après avoir sauvé hello.ml, on le compile partiellement avec :

```
> ocamlc -c hello.ml -o hello.cmo
```

Et enfin, on fait l'édition des liens du programme complet :

```
ocamlc out.cmo hello.cmo -o hello
```

Et on peut l'exécuter :

```
> ./hello
Hello, World!
```