Programmation des Fonctions

MPSI - Prytanée National Militaire

Pascal Delahaye

28 mars 2018

Tandis que Python permet à la fois de créer des fonctions et des procédures, OCaml est un langage *fonctionnel* et permet donc uniquement de créer des fonctions.

1 Construction standard d'une fonction

Exercice : 1

Déclaration de fonctions :
Taper les instructions suivantes et commenter la réponse de OCaml :

OCaml
```
(* fonction à une seule variable *)

let f1 x = 3. *. x *. sin(x) ;;
f1 5. ;;
```

OCaml
```
(* fonction à plusieurs variables *)

let f2 x n = x ** (float_of_int n) ;;
f2 3. 2 ;;
```

A retenir ! !

1. Les commentaires sous OCaml prennent la forme suivants : `(* ... *)`
2. Les parenthèses ne sont pas indispensables autour des arguments (voir plus loin le typage des fonctions).
3. La fonction, renvoie la dernière valeur calculée.
4. OCaml reconnaît automatiquement les types des arguments en entrées et de la valeur de sortie.

Un programme est par définition transcrit dans un langage informatique (OCaml en l'occurence) qui n'est pas immédiatement compréhensible par le lecteur.

IMPORTANT !!! On prendra donc soin dans les copies :

1. d'accompagner tout programme d'une courte présentation faisant apparaître :
   (a) L'objectif du programme (ou de la fonction)
   (b) La description des arguments en entrée en prenant soin de préciser leur type
   (c) Une description de l'algorithme utilisé
   (d) La description de la valeur de sortie
2. d'insérer des commentaires dans le corps même du programme des commentaires.

Exercice : 2

Construire une fonction :

1. qui permet d'additionner deux nombres.
2. de variables a et b qui dit si le booléen $a \Longleftrightarrow b$ est vrai ou faux
3. de variable un caractère lettre qui transforme cette lettre en majuscule avec `int_of_char` en `char_of_int`.
4. qui calcule le module d'un nombre complexe $x + iy$.

Remarque 1. De la même façon qu'il est possible de définir localement une variable, il est possible de définir une fonction locale à une autre fonction. Voir l'exercice suivant ...

Exercice : 3

Insertion dans la fonction de calcul du module d'un nombre complexe $x+iy$ une fonction locale permettant de calculer le carré d'un réel.

OCaml
```
let f x y = let carre t = t*.t in
            sqrt (carre x +. carre y);;
```

Exercice : 4

Ordre de priorité

Instructions	Réponse	Commentaires
let s x = x +. 1. ; ;		
let c x = (x,x) ; ;		
s 2. *. 3. ; ;		
s (2. *. 3.) ; ;		
c 2,3 ; ;		
c (2,3) ; ;		
1 + 2 , 3 ; ;		

A retenir ! !

1. Les fonctions sont les opérateurs de priorité maximale.
   On pourra cependant mettre des parenthèses pour plus de clareté.
2. La virgule est l'opérateur de priorité minimale

2 Typage d'une fonction

On appelle *typage* d'une expression, la recherche de son type par OCaml.
Nous allons dans ce chapitre, comprendre le sens du typage donné par OCaml.

2.1 familiarisation

`Exemple 1.` **Familiarisation avec le typage d'une fonction par Caml**

Instructions	Réponse	Commentaires
let f p = int_of_char p ; ;		
let g (x,n) = n + int_of_float x ; ;		
let g x n = n + int_of_float x ; ;		

2.2 Constatations

I] Dans le cas d'une fonction à un seul argument (cas 1 et 2 de l'exemple précédent), OCaml retourne le type :

`type1 − > type 2.`

On interprète cela de façon simple :
- `type1` représente le type de l'argument
- `type2` représente le type de la valeur de sortie

II] Pour une fonction à n arguments (cas 3 de l'exemple précédent), Caml retourne le type :

`type arg1 − > type arg2 − > ...− > type argn − > type sortie.`

L'exemple suivant doit nous permettre de comprendre le sens de ce type.

`Exemple 2.` **Familiarisation avec le typage d'une fonction de plusieurs variables**

Instructions	Réponse	Commentaires
let produit x y z = x *. y *. z ; ;		
let prod_2var = produit 1. ; ;		
let prod_par6 = produit 2. 3. ; ;		
let carre f x = (f x) ** 2. ; ;		
carré produit 2. 3. 4. ; ;		
carré (produit 2.) 3. 4. ; ;		
carré ((produit 2.) 3.) 4. ; ;		
carré (produit 2. 3.) 4. ; ;		

Remarque 2. Les instructions précédentes montrent qu'il est possible (voir très usuel !) de rencontrer sous OCaml des fonctions dont l'un ou plusieurs arguments est lui même une fonction.

A retenir ! !

Une fonction de n variables (et non d'un "n-uplet") peut s'interpréter de plusieurs façons :

1. Soit tout simplement comme une fonction de n variables.
2. Soit comme une famille de fonctions à $n-1$ variables paramètrées par la première variable
3. Soit comme une famille de fonctions à $n-2$ variables paramètrées par les deux premières variables
4. etc ...

Ainsi, `f x y` s'interprète des deux façons suivantes : $\begin{cases} f(x,y) \\ (f(x))(y) \end{cases}$ où $f(x)$ est alors une fonction de variable y

Remarque 3. La notation `f x y` est donc bien plus polyvalente que la notation usuelle `f (x,y)`

DÉFINITION 1 : **Curry - uncurry**
Il existe deux programmations possibles pour une fonction `f` de plusieurs variables :
1. Une version *non-currifiée* `let f (x,y) = ... ;;`
2. Une version *currifiée* `let f x y = ... ;;`

Exercice : 5
Déterminer deux fonctions `curry` et `uncurry` permettant respectivement de passer d'une fonction non-currifiée à une fonction curifiée et inversement :

Exercice : 6
Sans utiliser l'ordinateur, typer la fonction suivante :

OCaml
```
let minim comp a b = if comp a b then a
                                 else b;;
```

2.3 Fonctions complémentaires

2.3.1 La fonction `List.map`

La fonction `List.map` de OCaml permet d'appliquer une même fonction f à chacune des composante d'une liste de valeurs.
La syntaxe est : `List.map f L;;`.

`Exemple 3.` Application de la fonction `map`.

OCaml
```
let carre x = x*x;;
List.map carre [1;2;3;4;5];;
```

Remarque 4. Cette fonction permet de construire rapidement et simplement des itérations d'une même fonction.

Question : Sans utiliser l'ordinateur, déterminer le type de la fonction `List.map`.

2.3.2 Les fonctions `List.mem` et `List.rev`

`Exemple 4.` **Découverte de ces deux fonctions**

Instructions	Réponse	Commentaires
List.mem ; ;		
List.rev ; ;		

A retenir ! !

1. La fonction `List.map f l` permet de transformer toutes les composantes d'une liste `l` par une fonction `f`
2. La fonction `List.mem e l` permet de savoir si un élément `e` se trouve dans une liste `l`
3. La fonction `List.rev l` permet d'inverser l'ordre des éléments d'une liste `l`

3 Conseils de présentation des procédures

La lisibilité d'un programme est extrêmement importante dans la mesure où elle permet :

1. au programmeur de rechercher facilement des erreurs de syntaxes
2. au lecteur du programme de comprendre sans effort inutile le fonctionnement du programme.

Afin de garantir une bonne lisibilité de vos programmes, vous pourrez intervenir à trois niveaux :

1. Au niveau de la structure et des conventions d'écriture :
   Il s'agit en particulier :
   (a) d'aligner et de placer les unes au dessous des autres, les instructions de même niveau
   (b) de décaler toutes les commandes locales à une même structure
   (c) de laisser des espaces entre les différents opérateurs.

2. Au niveau des commentaires :
   On pourra en particulier donner (sous forme de commentaires) un titre aux programmes conçus et ajouter des commentaires supplémentaires dans le corps du programme si celui-ci est constitué de parties distinctes.
   Les commentaires doivent respécter la syntaxe suivante : `(*... commentaires...*)`

3. Au niveau du nom des variables :
   Plus vous donnerez des noms explicites aux variables que vous utiliserez et aux fonctions que vous créerez, plus votre programme sera compréhensible plus vous serez à même de comprendre le sens des instructions qu'il contient.

`Exemple 5.` **De l'intérêt du choix des notations !..**
Attention ... Dans les manipulations suivantes, les notations choisies sont volontairement ambiguës.
Votre travail consiste à les modifier afin de rendre ces différents programmes plus intelligibles !

Instructions	Réponse	Commentaires
let z x y = 2. *. x y ; ;		
let t x y = (x y) ** 2. ; ;		
let u x y = z (t x) y ; ;		
let v x = x -. 1. ; ;		
u v 4. ; ;		

Version corrigée des instructions précédentes :

Exercice : 7

Une année est définie par un entier.
Une année est bissextile lorsqu'elle est divisible par 4 à l'exception des années du début du siècle si ces années ne sont pas divisibles par 400.
Ecrire une fonction booléenne permettant de dire si une année est bissextile ou non.

4 Programmation d'une fonction mathématique simple

On entend ici par "fonction simple", une fonction qui traite de la même façon tous les arguments quelques soient leurs valeurs.

4.1 Sous forme d'une procédure standard

`Exemple 6.` Pour définir une fonction d'une seule variable

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

`Exemple 7.` Pour définir une fonction de plusieurs variables

OCaml
```
let f x y = cos(x)*.y ;;
let g x y = (2*x,y) ;;
```

Remarque 5. Dans le cas d'une procédure, on préférera la forme currifiée à la forme non-currifiée (arguments entre parenthèses) qui donne une plus grande souplesse dans l'utilisation de celle-ci.
Avec la forme currifiée, les expressions $f\ x$ et $g\ x$ (deuxième exemple) ont un sens et représentent deux fonctions de la variable y de paramètre x.

4.2 Avec la commande `fun`

`Exemple 8.` Pour une fonction d'une seule variable

OCaml
```
let f = fun x -> x*x ;;
```

Remarque 6. Remarquez qu'avec cette syntaxe, on définit une fonction f sans préciser son argument x.

`Exemple 9.` Typer la fonction suivante : `let f = fun x -> x::[];;`

Remarque 7. la syntaxe utilisant la commande `fun` sera plus particulièrement utilisée lorsqu'on a besoin d'entrer une fonction en variable d'une autre fonction. Cette fonction sera alors entrée sous la forme : `(fun x -> 2.*.x)`. Voir l'exemple suivant :

`Exemple 10.` Fonction dont un des arguments est une fonction

OCaml
```
let carre f x = (f x)**2.;;    (* fonction qui renvoie f(x)^2 *)

carre (fun x -> 2.*.x) 3.;;    (* application de la fonction "carre" à la fonction "x -> 2x" *)
```

Exercice : 8

Soit $f : \mathbb{R} \mapsto \mathbb{R}^+$ et $a,\ b \in \mathbb{R}$.
Construire une fonction de variables f, a et b donnant l'aire du trapèze de sommet $(a,\ 0)$, $(b,\ 0)$, $(a,\ f(a))$ et $(b,\ f(b))$.

Exercice : 9

Définir la composée de deux fonctions.

Remarque 8. Comme dans l'exemple suivant, on pourra utiliser cette syntaxe pour définir une fonction locale à une fonction d'ordre supérieur.

`Exemple 11.` Pour définir une famille de fonctions

OCaml
```
let plus n = fun x -> x+n ;;   (* "plus n" est une fonction qui a x associe x+n *)
let g = plus 3;;               (* "g" est la fonction qui a x associe x+3 *)
```

5 Le filtrage en programmation fonctionnelle

Lorsque l'expression de la valeur de sortie d'un programme dépend des différentes valeurs des arguments ou de la valeur d'une fonction des arguments (c'est ce que l'on appelle le *filtrage*), la programmation usuelle utilise une structure de contrôle. Plus adaptée à ce genre de situation, OCaml offre une syntaxe plus simple d'utilisation et beaucoup plus lisible.

5.1 Premier type de filtrage

5.1.1 Syntaxe

Ce premier type de filtrage est particulièrement adapté lorsque les *motifs* de filtrage correspondent à des ensembles de valeurs du dernier argument de la fonction. Dans le cas d'une fonction de 3 variables, il se présente de la façon suivante :

```
let nom_de_la_fonction var1 var2 =   function
                                   | motif 1   -> Traitement 1
                                   | motif 1   -> Traitement 2
                                               :
                                   | motif n   -> Traitement n ;;
```

`Exemple 12.` La fonction f définie par $\begin{cases} f(0) = 1 \\ f(1) = 3 \\ f(n) = 3n+2 \end{cases}$ $\forall n > 1$ se programme alors en OCaml de la façon suivante :

OCaml
```
let f = function
        | 0 -> 1
        | 1 -> 3
        | n -> 3*n+2;;
```

`Exemple 13.` Pour filtrer des couples, on peut procéder ainsi :

```OCaml
let f = function    (* non currifiée *)
        |(0,y) -> 1
        |(x,0) -> 1
        |(x,y) -> x+y;;
```

```OCaml
let f x = function  (* currifiée *)
        |0 -> 1
        |y -> match y with
                    | 0 -> 1
                    | _ -> x + y;;
```

5.1.2 Ajout d'une condition booléenne.

`Exemple 14.` Il est possible d'ajouter une condition booléenne à l'aide du mot clé `when` :

```OCaml
let f = function
        |(x,y) when x=y -> true
        |_ -> false ;;
```

```OCaml
let f x = function
        |y when x=y -> true
        |_ -> false ;;
```

`Exemple 15.` Programmer la fonction f suivante : $\begin{cases} f(x) = 3x - 1 \text{ si } x < 0 \\ f(0) = 2 \\ f(x) = x^2 \text{ si } x > 0 \end{cases}$.

5.1.3 Message d'erreur.

Si l'on souhaite indiquer à l'utilisateur de la fonction de filtrage qu'il utilise une valeur interdite pour argument, on pourra utiliser la commande `failwith`. Si l'argument n'est pas du type adapté, Caml renvoie automatiquement un message d'erreur de la forme : `This expression has type float, but is used with type int`.

```OCaml
let f = function
        | 0. -> failwith "L'argument de cette fonction ne peut être nul"
        | x -> 1./.x;;
```

Remarque 9.

1. Les types des motifs de filtrage définissent le type de l'argument. Ils doivent donc tous être de type identique !
2. De même, puisqu'une fonction renvoie un résultat de type déterminé, les résultats correspondant à chaque motif doivent être de même type (sauf dans le cas de l'utilisation de la commande `failwith`).

Exercice : 10

Si $f : \mathbb{R} \mapsto \mathbb{R}$, on peut approcher $f'(x)$ par $\dfrac{f(x+h)-f(x)}{h}$ avec h proche de 0.
Construire une fonction d'argument f et h qui renvoie la fonction f' approximée de la façon précédente.
Votre fonction devra renvoyer un message d'erreur lorsque $h = 0$.

5.2 Deuxième type de filtrage

Il arrive parfois que le filtrage à effectuer porte non pas sur les arguments eux-même, mais sur une partie des arguments ou encore les valeurs prises par une expression dépendant des arguments de la fonction. Dans ce cas, ce deuxième type de filtrage sera plus adapté. Il se présente de la façon suivante :

let nom_de_la_fonction	*arguments*	=	match	*fonction des arguments*	with
|	motif 1	$->$	Traitement 1		
|	motif 1	$->$	Traitement 2		
			$\vdots$		
|	motif n	$->$	Traitement n ;;		

`Exemple 16.` Cas d'une fonction de plusieurs arguments dont le filtrage ne porte que sur un seul :

OCaml
```
let f x y = match y with
        | true -> x +. 1.
        | false -> x -. 1. ;;
```

Remarque 10. Dans le cas d'un filtrage sur une partie des arguments, même si le filtrage avec la commande `function` peut convenir, elle est moins adaptée que le filtrage utilisant la commande `match ... with`.

Ainsi que le montre les exemples suivants, la structure `match ... with ...` permet aussi de faire porter le filtrage sur les valeurs prises par une fonction des arguments.

`Exemple 17.`

OCaml
```
let f x y = match x=y with
        |true -> 1
        |_ -> 0 ;;

f 2 2 ;;
```

OCaml
```
let f x y = match x+y with
        |0 -> 1
        |1 -> 0
        |_ -> failwith "perdu !" ;;
f 2 2 ;;
```

Remarque 11. Remarquez que le dernier motif de filtrage est représenté par le symbole _ qui signifie "tous les arguments restants".

Exercice : 11

A l'aide de la fonction `random_int`, construire une fonction f qui renvoie la valeur "pair" si le nombre entier choisi au hasard par la machine est pair et "impair" sinon.

Exercice : 12

Noël était un mardi en 1900.
Construire une fonction donnant le jour de la semaine où a eu lieu Noël pour une année comprise entre 1900 et 1999.

6 Le polymorphisme

6.1 Définition

Dans l'exemple suivant, le type des arguments de la fonction est indéterminé.

OCaml
```
let f = function
        |(x,y) when x=y -> true
        |_ -> false ;;

f : 'a * 'a -> bool = <fun>     (* résultat renvoyé après compilation *)
```

1. Lorsque, comme dans l'exemple précédent, le type des arguments (et éventuellement de la valeur de sortie) d'une fonction (de filtrage ou pas !) est indéterminé (rien dans le programme précédent permet de définir les types de x et y), OCaml leurs attribue un type "indéterminé" qu'il appelle `'a` ou `'b` (prononcer alpha ou béta).

2. Ici, du fait de l'égalité $x = y$, les variables x et y ont nécessairement le même type : caml leur attribue donc le type `'a` à toutes les deux.

3. Cette capacité qu'a OCaml à construire des fonctions qui peuvent être utilisées sur des objets de types différents s'appelle le *polymorphisme*. On dit alors que la fonction est *polymorphe*.

Prenons un nouvel exemple :

`Exemple 18.`

OCaml
```
let f x y = if x = y then (x,y)
                     else (y,x);;

f : 'a -> 'a -> 'a * 'a = <fun>    (* résultat renvoyé par Ocaml après compilation *)
```

Remarque 12. Dans cet exemple, le type de sortie est lui-aussi indéterminé, mais est lié au type de x et y.

Exercice : 13

Définir les fonctions polymorphes suivantes :

1. fonction identité
2. fonction qui extrait la deuxième coordonnée d'un triplet.

6.2 Un exemple de fonction polymorphe : la fonction `max`

La fonction polymorphe `max` de OCaml renvoie le maximum de deux éléments.
La syntaxe est : `max x y;;`.
Son typage est : `('a -> 'a -> 'a) = <fun>`.

`Exemple 19.` Application de la fonction `max`.

OCaml
```
max 2 3;;          (* pour des entiers *)
max 2. 3.;;        (* pour des flottants *)
max (2,2) (3,7);;  (* pour des couples *)
max "tet" "afe";;  (* pour des chaînes de caractères *)
```