LANGAGE OCaml – TP 2
Fonctions et Filtrage

Introduction

Le langage OCaml obéit au paradigme déclaratif fonctionnel. Des fonctions peuvent donc être écrites, de manière similaire aux autres langages de programmation : ces dernières servent à découper le code de manière logique, en éléments de taille raisonnable, et à éviter la duplication de parties de code.

Pour rappel, la programmation déclarative fonctionnelle va mettre en avant la définition et l'évaluation de fonctions pour résoudre un problème, en interdisant toute opération d'affectation. Dans ce mode de programmation, la mémoire est donc gérée automatiquement et de manière transparente pour le programmeur.

Contrairement au style de programmation impérative où le programmeur réfléchit aux différentes instructions à exécuter dans un ordre donné pour résoudre un problème, la programmation fonctionnelle est organisée autour du concept de fonctions, souvent récursives. Le programmeur doit décrire de propriétés mathématiques de la fonction, qui prend en entrée des arguments d'un certains type et retourne des valeurs d'un certain type, après avoir effectué un calcul. L'utilisation des références et des affectations n'est donc plus utile, car propre au paradigme impératif.

La programmation impérative et la programmation déclarative obéissent à des paradigmes différents. L'approche impérative s'attache à décrire le processus permettant d'aboutir au résultat attendu. L'approche privilégiée en programmation déclarative est la description du résultat attendu plutôt que les étapes permettant l'obtention de ce résultat. Son niveau d'abstraction est donc très élevé, ce qui rend les codes plus synthétiques à l'écriture mais plus difficiles à déchiffrer pour un autre programmeur.

Syntaxe

Déclaration d'une fonction

On rappelle les syntaxes d'une déclaration d'une fonction vue dans le premier TP :

`let carre x = x**2. ;;`	Définit une fonction nommée `carre`, prenant un unique argument `x` et renvoyant le carré de `x`. Le corps de la fonction correspond à l'expression qui doit être évaluée lors de l'appel de la fonction. Aucun return n'apparaît puisqu'il n'y a pas de distinction entre expression et instruction en OCaml. Les arguments ne sont pas mis entre parenthèses et les types sont inférés.

L'envoi de cette fonction provoque la sortie : `val carre : float -> float = <fun>`

Pour des fonctions recevant plusieurs arguments :

`let moyenne x y = (x +. y) /. 2.0 ;;`	Le type est inféré : cette fonction prend deux arguments de type `float` et renvoie une valeur de type `float`. C'est la présence du `+.` qui permet au compilateur d'inférer le type.
`val moyenne : float -> float -> float = <fun>`	Résultat de l'envoi de la déclaration ci-dessus.

Fonction et type unit

Le type `unit` est un type particulier contenant un unique élément : `()`. Ce type constitue une valeur de retour comme une autre :

`let f x =`
`  print_int x ;`
`  print_newline() ;;`
 	
`f 5;;`
	Affichage en sortie :
	`# val f : int -> unit * unit = <fun>`
	`#`
	`5- : unit * unit = ((), ())`

Ce type est utilisé pour procéder à des modifications par effet de bord.

`let x = ref 0;;`
`let affecte v =`
`  x := v;;`
 	
`print_int !x;;`
`affecte 3;;`
`print_int !x;;`
	Affichage en sortie :
	`# val x : int ref = {contents = 0}`
	`# val affecte : int -> unit = <fun>`
	`# 0- : unit = ()`
	`# - : unit = ()`
	`# 3- : unit = ()`
	
	La fonction affecte ne retourne pas vraiment une valeur, comme l'indique son type (`unit`). Elle procède par effet de bord pour modifier le contenu de la référence x.

`let remise_a_0 () =`
`  x := 0 ;;`
 	
`remise_a_0 () ;;`
`print_int !x;;`
	Inversement, une fonction sans argument va s'écrire comme prenant un seul argument de type unit, noté ().
	
	Affichage en sortie :

Type fonctionnel et polymorphisme

Lorsque l'on écrit une fonction, OCaml infère les types en fonction des contraintes présentes dans le corps de la fonction.

En cas d'absence de contraintes, OCaml utilise les notations `'a, 'b`, … (prononcer alpha et bêta) pour désigner un type quelconque.

Expressions	Sortie obtenue

`let f x n =`
`  if n = x then x`
 	
`  else n;;`	

`f 2 3 ;;`	
`f 2. 3. ;;`	

L'opérateur d'égalité est polymorphe : il n'introduit aucune contrainte sur le type de n et de x. Par contre, les deux variables n et x doivent être du même type.

Les opérateurs de comparaison (< <= = <> <= <) sont des opérateurs polymorphes.

Exemples

Tester les instructions suivantes ; observer les sorties obtenues.

Exemple 1 :

```
let concatene mot1 mot2  = mot1^mot2;;
concatene "cou" "cou";;
```

Exemple 2 : une fonction peut appeler une autre fonction.

```
let surface rayon = 3.14159 *. rayon*.rayon;;
surface 1.;;
let volume rayon hauteur = surface rayon *. hauteur;;
volume 1. 1.;;
volume 1 1.;;
```

Exemple 3

```
let fonction x y z =
   if x>3 then y+x else z-x;;
fonction 1 2 3;;
fonction 4 2 3;;
```

Exemple 4 : une fonction étant une valeur comme une autre, elle peut être déclarée localement, comme une variable de n'importe quel autre type. Ainsi, on peut écrire :

```
let carre x = x*x
      in carre 3 + carre 4 = carre 5 ;;
```

Ou bien :

```
let pythagore x y z =
   let carre n = n*n
   in carre x + carre y = carre z;;
pythagore 3 4 5;;
```

Notion de fonction de première classe

Les fonctions définies ci-dessus sont des fonctions appelées de première classe, c'est-à-dire des valeurs pouvant être créées par des calculs, passées en argument à des fonctions, ou retournées, comme n'importe quelles valeurs.

Une fonction peut être une expression comme une autre, alors anonyme, et introduite par le mot clé `fun` : `fun x -> x+1;;` est la fonction associant à un entier x son successeur.

Il s'agit d'une expression ayant pour type `int -> int`, qui peut donc être appliquée à un entier : `(fun x -> x+1)5;;`

provoque l'affichage : `# - : int = 6`

Une déclaration de la forme : `let f x = x+1 ;;` est un sucre syntaxique (extension à la syntaxe d'un langage pour le rendre plus agréable à lire) pour la déclaration :

```
let f = fun x -> x+1 ;; (*La flèche -> désigne une fonction *)
```

Il n' y a donc pas un `let` pour les variables, et un autre pour les fonctions, mais un unique déclaration `let` pour introduire une variable pouvant contenir des variables de types quelconque, entier, booléen ou fonction…

Remarque : `tableau[i]`, en C, est un sucre syntaxique pour `*(tableau+i)`.

Ecrire les déclarations équivalentes aux déclarations suivantes :

`(fun x y -> x * x + y * y) 2 1 ;;`
 	
Affichage en sortie :
`# - : int = 5`	

`(fun x -> fun y -> x * x + y * y) 2 1;;`
 	
Affichage en sortie :
`# - : int = 5`	

La seconde écriture de la fonction suggère que l'on peut appliquer une telle fonction à un seul argument. En effet, l'instruction :

```
(fun x -> fun y -> x * x + y * y) 2 ;;
```

provoque l'affichage :

```
# - : int -> int -> <fun>
```

On parle alors d'application partielle.
On peut alors définir une fonction f :

```
let f x y = x*x + 2*y ;;
(sortie : val f : int -> int -> int = <fun> *)
```

Puis construire une fonction g en appliquant f partiellement :

```
let g = f 3 ;; (* sortie : val g : int -> int = <fun> *)
```

Le type de g est celui d'une fonction prenant en argument un entier :

```
g 4 ;; (* applique f avec x = 3 et y = 4 *)
```

donnera la sortie : `- int 17`

La fonction `g` est comparable à : `fun y -> 3*3 + 2*y`. Son corps est identique à celui de `f` auquel on aurait substitué 3 à la variable formelle `x`.

L'application partielle d'une fonction est une expression qui est encore une fonction ; c'est donc une manière de retourner une fonction.

On peut noter que l'évaluation ne se fait pas de manière paresseuse. Par exemple, dans la déclaration

```
(let x = 1+2 in fun y -> x*x + 2*y) 1;;
(* sortie obtenue : (2+1)*(2+1) + 2*1 = 11 de type int *)
```

La sortie correspond à 3×3 + 2×1 = 11 et non à (2+1) × (2+1) + 2×1 = 11 : l'évaluation est effectuée avant l'appel de la fonction.

Fonctions d'ordre supérieur

De même qu'une fonction peut retourner une fonction comme résultat, elle peut prendre une ou plusieurs fonction(s) en argument. Cette capacité à manipuler les fonctions comme des valeurs de premières classes est appelée ordre supérieur.

On peut alors envisager une fonction composant deux fonctions :

```
let f1 x = x ;;
let f2 x = x*x;;

let compose f g  x = f (g x);;
(* peut aussi s'écrire :
let compose f g  = fun x -> f (g x);;
*)

(compose f1 f2) 2;;
Compose f1 f2 ;;
```

Signatures :

Sortie :

On note que le type retourné par f1 est de type `'a`, soit un type non défini. La fonction `f1` est une fonction polymorphe.

Filtrage

Le filtrage du langage OCaml s'apparente au `switch` du langage C. Il permet d'écrire des fonctions en envisageant des cas correspondant à des motifs. Par exemple, la fonction sur des entiers définie par

$$\begin{cases} f(0) = 0 \\ f(1) = 2 \\ \forall n > 2, f(n) = 2n + 2 \end{cases}$$

Peut s'écrire :

`let f n =`
`  match n with`
`  |0->0`
`  |1->2`
`  |_->2*n+2`
`;;`
 	
`f 0 ;;`
`f 1 ;;`
`f 4 ;;`
	Signature
	
	Exécution

Le filtrage est défini par le mot clé `match`. La fonction compare alors l'argument à filtrer (ici n) avec le premier motif (ici, 0) ; si l'argument correspond, la fonction renvoie l'évaluation du membre de droite (ici, 0) ; si l'argument passé ne correspond pas, la fonction va tester successivement les autres valeurs pour renvoyer la valeur adéquate. Le motif _ correspond à la valeur quelconque. Ce motif définit ici la valeur à retourner par défaut.

Les motifs du filtrage doivent avoir le même type que la variable passée en argument.

De même, les membres de droite doivent aussi avoir le même type.

Contrairement au `switch`, le filtrage permet de filtrer des couples de valeurs.

`let filtre (x, y) =`
`  match (x, y) with`
`  | (0,_)->0`
`  | (_,0)->0`
`  | _->x+y`
`  ;;`
 	
`filtre (0,1);;`
`filtre (2,0);;`
`filtre (3,2);;`
	Signature :
	
	L'étoile indique un n-uplet
	
	Sortie

`(* Autre possibilité *)`
`let filtre x y =`
`  match (x, y) with`
`  | (0,_)->0`
`  | (_,0)->0`
`  | _->x+y`
`  ;;`
 	
`filtre 0 1;;`
`filtre 2 0;;`
`filtre 3 2;;`
	Signature :
	
	Sortie :

La fonction suivante indique si l'une des composantes d'un couple est nulle ou si les deux éléments sont égaux :

```
let egalite_ou_zero x y =
   match (x, y) with
   | (0,_)-> true
   | (_,0)->true
   | (x, y) when x = y -> true
   | _ -> false
     ;;

egalite_ou_zero 0 1;;
egalite_ou_zero 1 0;;
egalite_ou_zero 0 1;;
egalite_ou_zero 3 1;;
```

Signature :

Sortie :

Une variable ne peut être utilisée à plusieurs reprises dans l'instruction match :

```
let filtre x y =
   match (x, y) with
   | (0,_)->0
   | (_,0)->0
   | (x, x)->2*x
   | _->x+1
     ;;
```

Message retourné :

Motif par combinaison de motifs

IL est possible de construire un motif par combinaison de motifs p_1, p_2, ...p_n. La seule contrainte forte est de refuser tout nommage à l'intérieur de ces motifs. Donc chacun d'eux ne devra contenir que des valeurs constantes ou le motif universel.

```
let est_voyelle c = match c with
   |'a'|'e'|'i'|'o'|'u'|'y' -> true
   | _ -> false ;;

est_voyelle 'a';;
est_voyelle 'p';;
```

Sortie écran :

```
# val est_voyelle : char -> bool =
<fun>
# - : bool = true
# - : bool = false
```

Motif avec garde (garde de filtrage)

Le filtrage permet d'écrire des instructions dépendant de cas correspondant à des motifs. Les instructions générées par chaque cas doivent impérativement renvoyer une évaluation fournissant le même type. Le mot clé `match` a déjà été vu lors d'un TP précédent.

Dans un motif, une même variable ne peut apparaître à plusieurs reprises, ce qui peut parfois limiter les possibilités.

Dans un tel cas, on peut ajouter une garde à un motif, c'est-à-dire une condition booléenne qui doit être vérifiée en plus du filtrage

`let egalite_composantes(x, y) =`
`  match (x, y) with`
`  | (x,x) -> true`
`  | _ -> false ;;`
	Sortie écran :
	
	`Error: Variable x is bound several times in this matching`

`let egalite_composantes (x, y) =`
`  match (x, y) with`
`  | (x, y) when x = y -> true`
`  | _ -> false ;;`
 	
`egalite_composantes (0,1);;`
`egalite_composantes (1,1);;`
	Sortie écran :
	
	`# val egalite_composantes :`
	`'a * 'a -> bool = <fun>`
	
	`# - : bool = false`
	`# - : bool = true`

`let egalite_composantes = function`
`  | (x, y) when x = y -> true`
`  | _ -> false ;;`
 	
`egalite_composantes (0,1);;`
`egalite_composantes (1,1);;`
 	
Sortie écran :
`# val egalite_composantes2 :`
`'a * 'a -> bool = <fun>`
`# - : bool = false`
`# - : bool = true`
	Dans le contexte d'une définition d'une fonction à un seul paramètre, une autre écriture du filtrage de motif est possible qui n'utilise pas l'expression `match`.
	
	La syntaxe est alors :
	`let` *une_fonction* `= function`
	`  |` $motif_1$ `->` $expr_1$
	`  |` $motif_1$ `->` $expr_1$
	`  ...`
	`  |` $motif_n$ `->` $expr_n$

Bilan

- Les fonctions sont des valeurs comme les autres : elles peuvent être locales, anonymes, arguments d'autres fonctions, …
- Les fonctions peuvent être appliquées partiellement ;
- Les fonctions peuvent être polymorphes.
- Le filtrage permet d'adapter une sortie en fonction d'un motif appliqué aux arguments passés à la fonction. Les différentes sorties doivent être du même type.
- Le filtrage permet d'appliquer des instructions sur des motifs particuliers
- Les motifs gardés permettent d'émettre des conditions supplémentaires sur les motifs de filtrage.

Exercices

Exercice 1

Prévoir les sorties dans l'interpréteur de commande après les instructions suivantes :

`let a = 2;;`
`let f x = a*x ;;`
`(*REPONSE ATTENDUE *)`
`let a = 3 in f 1 ;;`
`(*REPONSE ATTENDUE *)`
`let a = 3 ;;`
`f 1 ;;`
`(* REPONSE ATTENDUE *)`	

`let a =`
`  let a = 3 and b = 2 in`
`  let a = a+b and b = a-b in`
`  a-b;;`
`(* REPONSE ATTENDUE *)`
 	
`let b = 2 in a-b*b;;`
`(* REPONSE ATTENDUE *)`	

Exercice 2 (révision des affectations)

1. En utilisant une affectation locale, calculer :

$$\frac{1+\sqrt{7}+\sqrt{7}^3}{1+e^{\sqrt{7}}} \quad \text{et} \quad \frac{ln\big(cos(1)\big)+sin\big(ln(5)\big)}{cos(1)+ln(5)}$$

2. En utilisant une affectation locale, donner un calcul efficace (i.e. avec un seul appel à la fonction exponentielle) de $th(x)$.

Exercice 3

Ecrire une fonction polymorphe de signature `val choix : bool -> 'a -> 'a -> 'a = <fun>`, recevant 3 arguments, qui renvoie le second argument si le premier argument est true, le second sinon. Proposer les expressions permettant d'obtenir les sorties écran suivantes :

```
# val choix : bool -> 'a -> 'a -> 'a = <fun>
# - : float = 1.
# - : string = "bon"
# - : int = 2
```

Exercice 4

Ecrire une fonction de signature `val sinc : float -> float = <fun>`, prenant une variable x de type `float` en argument et renvoyant un message d'erreur indiquant la division par 0 si x vaut 0, $\frac{\sin x}{x}$ sinon. Le filtrage devra être utilisé.

Un message d'erreur peut être retourné en utilisant l'instruction `failwith " message d'erreur "`.