OCaml

2 - Les fonctions

http://tsi.tuxfamily.org/OCaml

vincent.maille@ac-amiens.fr

8 février 2024

Dans un programme OCaml, il s'agit bien plus de décrire les fonctions que nous souhaitons employer (**programmation fonctionnelle**) que de donner des instructions de gestion des données ou de la mémoire de l'ordinateur (**programmation impérative**).

Par exemple, on souhaite définir la fonction f : $\begin{array}{ccc} \mathbb{Z} & \longrightarrow & \mathbb{Z} \\ x & \longmapsto & 2x+1 \end{array}$

```
# let f = function x -> 2*x+1;;
val f : int -> int = <fun>
# f(3);;
- : int = 7
```

ou encore :

```
# let f = fun x -> 2*x+1;;
val f : int -> int = <fun>
# f(3);;
- : int = 7
```

plus court :

```
# let f(x) = 2*x+1;;
val f : int -> int = <fun>
# f(3);;
- : int = 7
```

ou encore plus court :

```
# let f x = 2*x+1;;
val f : int -> int = <fun>
# f 3;;
- : int = 7
```

On privilégiera surtout ce type de déclarations.

Une fonction est un nouveau type : `type1 -> type2`

Vocabulaire

$-$ $>$ se prononce "**flèche**"

On dit aussi que la **signature** de la fonction est `int -> int`.

On évite d'utiliser les parenthèses lorsque l'on peut.
Avec la fonction $f : x \mapsto 2x + 1$ définie précédemment :

```
#f 3;;
- : int = 7
#f 3 + 1;;
- : int = 8
#f (3+1);;
- : int = 9
```

```
# f f 3;;
Characters 2-3:
  f f 3;;
  ^
Error: This function has type int -> int
       It is applied to too many arguments; maybe
       you forgot a `;'.
# f ( f 3 );;
- : int = 15
# f -2;;
Characters 1-2:
  f -2;;
  ^
Error: This expression has type int -> int
       but an expression was expected of type int
# f (-2);;
- : int = -3
```

Vocabulaire

Une fonction **récursive** est une fonction dont la définition fait elle-même intervenir cette fonction.

Le mot clé `rec` indique la définition d'un objet récursif :

```
let rec fact n =
        if n = 0
        then 1
        else n*fact (n-1)
;;
```

Nous reviendrons plus longuement sur la récursivité au prochain chapitre.

On peut aussi utiliser le principe de **récursivité croisée** si besoin :

```
# let rec est_pair n =
        if n = 0
        then true
        else est_impair (n-1)
and est_impair n =
        if n = 0
        then false
        else est_pair (n-1);;
val est_pair : int -> bool = <fun>
val est_impair : int -> bool = <fun>
```

Fonctions à plusieurs paramètres

```
# let produitCouple (x,y) = x * y;;
val produitCouple : int * int -> int = <fun>
# produitCouple 2 3;;
Characters 1-14:
  produitCouple 2 3;;
  ^^^^^^^^^^^^^
Error: This function has type int * int -> int
       It is applied to too many arguments; maybe
       you forgot a `;'.
# produitCouple(2,3);;
- : int = 6
```

Deuxième solution (qui sera privilégiée)

```
# let produit x y = x * y;;
val produit : int -> int -> int = <fun>
# produit 2 3;;
- : int = 6
```

Vocabulaire

Une telle déclaration est appelée **fonction curryfiée** (H. Curry)

```
# let produit x y = x * y;;
val produit : int -> int -> int = <fun>
```

OCaml répond que `produit` est une fonction qui, à un entier x, associe une fonction $f_x : y \mapsto xy$ qui à un entier y associe l'entier xy. C'est le sens de la notation `int -> int -> int` pour signifier `int -> (int -> int)` (le typage est associatif à droite).

Vocabulaire

Version **curryfiée** de `produit` :

$$\begin{array}{rcccccc} \texttt{produit} & : & \mathbb{Z} & \longrightarrow & \mathcal{F}(\mathbb{Z};\mathbb{Z}) & & \\ & & x & \longmapsto & f_x & : \mathbb{Z} \longrightarrow & \mathbb{Z} \\ & & & & & y \longmapsto & f_x(y) = xy \end{array}$$

Dans ce cas, la fonction `produit` ne peut recevoir de couple en paramètre d'entrée

```
# produit(2,3);;
Characters 9-14:
  produit(2,3);;
          ^^^^^
Error: This expression has type 'a * 'b
       but an expression was expected of type int
```

Vocabulaire

Le procédé par lequel on passe de la fonction `produitCouple` à `produit` s'appelle la **curryfication**. Le procédé inverse, s'appelle lui la **décurryfication**

Intérêt des fonctions curryfiées :

```
# let produit x y = x * y;;
val produit : int -> int -> int = <fun>

# let fois3 = produit 3;;
val fois3 : int -> int = <fun>

# fois3 4;;
- : int = 12
```

- C'est le cas de la fonction `print_newline()` qui imprime un saut de ligne à l'écran.

- C'est aussi le cas de la fonction `time` du module `Sys` qui renvoie le nombre de secondes écoulées depuis le début de la session en cours

```
# Sys.time;;
- : unit -> float = <fun>
# let a=Sys.time();;
val a : float = 31391.935
```

Modules

Comme en Python, pour utiliser une fonction définie dans un module, on utilise un point. Les noms des modules commencent nécessairement par une majuscule.

Il n'est pas obligatoire de donner un nom aux fonctions :

```
#function x -> 2*x+1;;
- : int -> int = <fun>
#(function x -> 2*x+1) 7;;
- : int = 15
```

ou pour des fonctions curryfiées (avec le mot `fun`) :

```
# fun x y -> 2*x+y;;
- : int -> int -> int = <fun>
# fun a b c -> if b > c then a else b-c;;
- : int -> int -> int -> int = <fun>
```

Les fonctions, n'étant qu'un type parmi d'autres, peuvent à leur tour être placées comme paramètre d'une autre fonction.

```
# let comp f g x = f (g x);;
val comp : ('a -> 'b)->('c -> 'a)->'c->'b = <fun>
# let u x = 2 * x + 1;;
val u : int -> int = <fun>
# let v x = x * x;;
val v : int -> int = <fun>
# comp u v;;
- : int -> int = <fun>
# comp u v 3;;
- : int = 19
# comp (function x -> x+1)(function x -> x*x);;
- : int -> int = <fun>
```

Vocabulaire

On appelle cette possibilité la **pleine fonctionnalité**

Autre exemple de l'utilisation de fonctions anonymes : la déclaration curryfiée du produit :

```
let produit a b = a * b;;
```

Peut être déclarée ainsi :

```
let produit = function a -> function b -> a*b;;
```

Vocabulaire

On dit qu'une fonction est **polymorphe** si elle peut s'appliquer à différents types d'objets. Dans le cas contraire, on dit qu'elle est **monomorphe** ce qui est la majorité des cas pour un langage fortement typé.

```
let tri a b c =
  if a < b then
    if b < c then
      (a,b,c)
    else
      if a < c then
        (a,c,b)
      else
        (c,a,b)
  else
    if a < c then
      (b,a,c)
    else
      if b < c then
        (b,c,a)
      else
        (c,b,a)
  ;;
```

```
# let tri a b c = ...
val tri : 'a -> 'a -> 'a -> 'a * 'a * 'a = <fun>
# tri 1 5 9;;
- : int * int * int = (1, 5, 9)
# tri "cabane" "palais" "maison"  ;;
- : string * string * string =
                  ("cabane", "maison", "palais")
# tri 1 3.1 2;;
Characters 9-12:
    tri 1 3.1 2;;
          ^^^
Error: This expression has type float but an
         expression was expected of type int
```

Vocabulaire

Le **filtrage** est une manière simple et efficace de réaliser des tests. Syntaxiquement, un filtrage se présente comme une liste de clauses. Chaque clause est une construction de la forme

```
| motif -> expression.
```

Lorsqu'une clause est vérifiée, les suivantes ne sont pas testées.

Un premier exemple avec une fonction qui reçoit un entier de l'intervalle $[1;5]$ (sinon, ça ne fonctionne pas !!!) :

```
let est_premier n =
        match n with
                | 1 -> false
                | 4 -> false
                | _ -> true (* 2, 3, ou 5 *)
;;
```

L'appel

```
for i = 1 to 5 do  (* tiens.... une boucle ! *)
        print_int i;
        if est_premier i
        then print_string " est"
        else print_string " n'est pas";
        print_string " premier";
        print_newline();
done;;
```

affiche :

```
1 n'est pas premier
2 est premier
3 est premier
4 n'est pas premier
5 est premier
```

Dans une fonction à un paramètre, on peut omettre le `match...with`. Dans ce cas, on ne met pas non plus le nom de la variable (qui est déterminée par le filtrage) :

```
let est_prem = function
          1 -> false
        | 4 -> false
        | _ -> true
;;
val est_prem : int -> bool = <fun>
```

ou plus court :

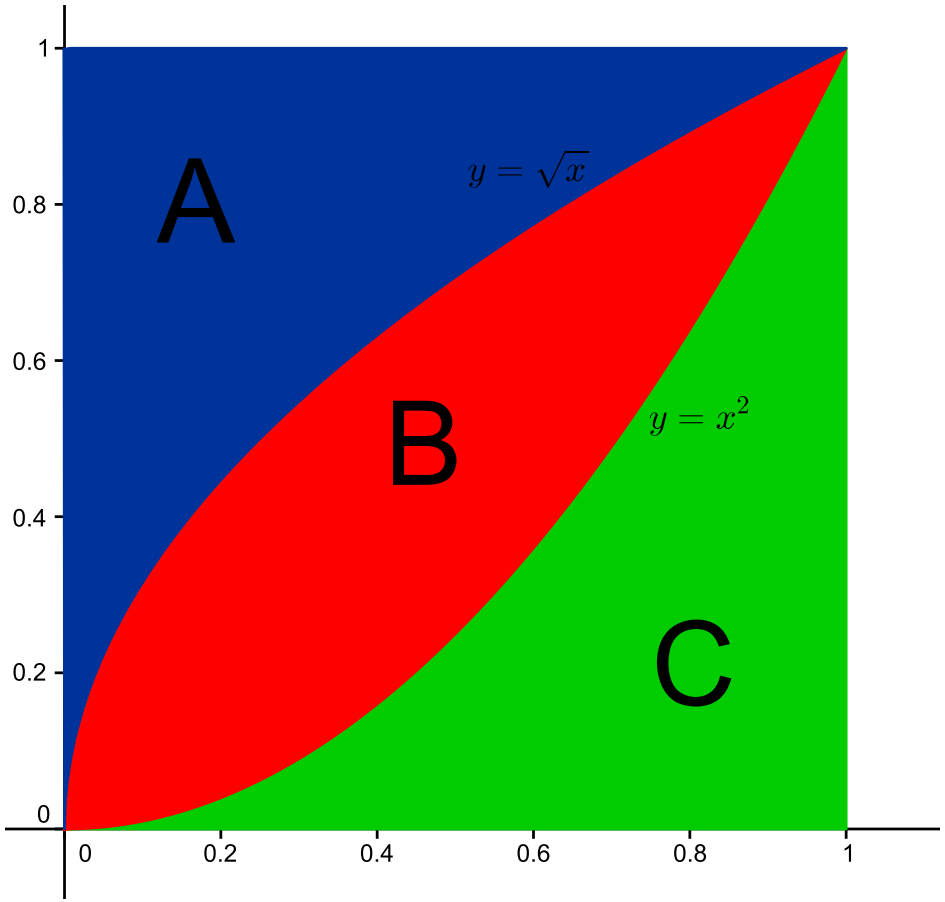
```
# let est_prem = function
          1 | 4 -> false
        | _ -> true
;;
val est_prem : int -> bool = <fun>
```

Vocabulaire

On dit que l'on réalise un **filtrage avec garde**, lorsque l'on ajoute une condition au test à l'aide du mot clé `when`.

Exercice

Écrire les version curryfiées et non curryfiées d'une fonction qui renvoie la zone d'un point en fonction de ses coordonnées.



Solution - exercice

- Version non curryfiée :

```
let zone = function
         | (x,y) when y > sqrt(x) -> "A"
         | (x,y) when y < x*.x -> "C"
         | _ -> "B"
;;
val zone : float * float -> string = <fun>
```

Solution - exercice

- Version curryfiée :

```
# let zoneC x y = match (x,y) with
        | (x,y) when y > sqrt(x) -> "A"
        | (x,y) when y < x*.x -> "C"
        | _ -> "B"
;;
val zoneC : float -> float -> string = <fun>
# zoneC 0.5 0.5;;
- : string = "B"
```

Remarque : on ne peut pas faire de filtrage implicite sur une fonction curryfiée à plusieurs paramètres.

OCaml signale si un filtrage ne contient pas tous les cas...

```
# let next = function
          | x when x mod 2 = 0 -> x / 3
          | x when x mod 3 = 0 -> 2*x
;;
Characters 13-81:
  ............function
    | x when x mod 2 = 0 -> x / 3
    | x when x mod 3 = 0 -> 2*x
Warning 8: this pattern-matching is not exhaustive.
All clauses in this pattern-matching are guarded.
val next : int -> int = <fun>
```

.... bon évidemment, il ne détecte pas tout ...

```
# let next = function
          | x when x mod 2 = 0 -> x / 3
          | x when x mod 2 = 1 -> 2*x
;;
Characters 13-82:
  ............function
    | x when x mod 2 = 0 -> x / 3
    | x when x mod 2 = 1 -> 2*x
Warning 8: this pattern-matching is not exhaustive.
All clauses in this pattern-matching are guarded.
val next : int -> int = <fun>
```

...en même temps ce filtrage n'est pas malin...