Fonctions, Expressions

Alain Camanes

alain.camanes@free.fr

Stanislas

Option Informatique
2021-2022

Plan

Fonctions d'une variable

```
# let f x = x * x;;
val f : int -> int = <fun>
# f 3;;
- : int = 9
```

↪ Inférence de types & Typage statique !

↪ Évaluation. Les parenthèses sont *optionnelles*.

```
# f f 1;;
Characters 0-1:
  f f 1;;
  ^
Error: This function has type int -> int
       It is applied to too many arguments;
       maybe you forgot a ';'.
```

↪ ATTENTION. Il n'y a pas d'instruction `return` ≠ Python.
Renvoie la dernière expression évaluée.

Fonctions de plusieurs variables

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

# f 7 8;;
- : int = 448
```

↪ Application partielle.

```
# let g = f 5;;
val g : int -> int = <fun>
# g 4;;
- : int = 80

# let h x = f x 3;;
val h : int -> int = <fun>
# h 5;;
- : int = 45
```

Curryfication

↪ Fonction *décurryfiée*.

```
# let f (x, y) = x * y * y;;
val f : int * int -> int = <fun>
# f (3, 4);;
- : int = 48
```

↪ Fonction *curryfiée*.

```
# let f x y = x * y * y;;
val f : int -> int -> int = <fun>
# f 3 4;;
- : int = 48
```

Ordre dans lequel sont passés les arguments + Théorie (catégories)

↪ Pas de parenthèse ; Pas de virgule !

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

```
# f (g 2) 4;;
- : int = 112
```

Plan

Expressions locales : un exemple

```
let moyenne x y =
  let somme = x +. y in
  somme /. 2.0;;
```

↪ somme est une *abréviation* pour `x +. y`

↪ On ne peut pas modifier sa valeur.

Écriture *équivalente*.

```
let moyenne x y =
  (x +. y) /. 2.0;;
```

Expressions globales

```
(* Soit pi = 3.14 un flottant. *)
# let pi = 3.14 ;;
val pi : float = 3.14
# pi ;;
- : float = 3.14
```

↪ Donner un nom à une valeur.

↪ Portée. Session du toplevel.

↪ *Abréviation* pour la valeur `3.14`.

↪ Règles de nommage. Lettres non accentuées et tiret bas.
Ne doivent pas commencer par une majuscule.

Expressions globales (II)

↪ Déclarations simultanées. `and`

```
# let x = 3 and y = 2;;
val x : int = 3
val y : int = 2
```

↪ Attention.

```
# let a = 3 and b = a * a;;
# Characters 18-19:
  let a = 3 and b = a * a;;
                    ^

Error: Unbound value a
# let b = (let a = 3 in a * a);;
val b : int = 9
```

Expressions locales

```
(* Posons y = 3 dans le calcul de y + 2 *)
# let y = 3 in y + 2;;
- : int = 5
# y;;
Characters 0-1:
  y;;
  ^
Error: Unbound value y
```

↪ Portée limitée à l'expression après le `in`.

↪ Prioritaire sur le global.

↪ Expression. Type & Valeur

```
# 4 + (let x = 2 in x * x);;
- : int = 8
# let x = (let y = 3 in y + 1) in x * 2;;
- : int = 8
```

Déclarations simultanées en local

↪ `and`

```
# let x = 3 and y = 2 in x * y;;
- : int = 6
```

Plan

Expressions conditionnelles

```
if <bool_1> then <expr_1> else <expr_2>
```

↪ Type. <expr_1> et <expr_2> de *même type*.

```
# if 3 > 0 then 1 else 2;;
- : int = 1

# if 3 > 0 then 1 else false;;
Characters 21-26:
  if 3 > 0 then 1 else false;;
                       ^^^^^
Error: This expression has type bool but
       an expression was expected of type
       int
```

Expressions conditionnelles (II)

↪ Pas de possibilité de `elif`.

```
# if -3 > 0 then 1 else
    if -3 = 0 then 2
    else 3;;
- : int = 3
```

↪ else facultatif mais expression de type `unit`.

À suivre...

↪ Possibilité d'écrire avec un filtrage par motifs.

À suivre...

Plan

Portée lexicale & Définition non dynamique

↪ Identifieurs évalués au moment de la définition.

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

# f 3;;
- : int = 5

# let a = 4;;
val a : int = 4
# f 3;;
- : int = 5
```

Fonctions locales

```
# let puissance_4 n =
    let carre n = n * n in
    carre (carre n);;
val puissance_4 : int -> int = <fun>

# puissance_4 3;;
- : int = 81
```

Fonctions anonymes

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

Exemple. La composition.

```
let compose f g =
  let h x = f (g x) in
  h;;

let p6 = compose (fun x -> x*x) (fun x -> x*x*x);;

p6 3;;
- : int = 729
```

Fonctions polymorphes

```
# let renvoie_3 x = 3;;
val renvoie_3 : 'a -> int = <fun>

# renvoie_3 "hello";;
- : int = 3

# renvoie_3 3.14;;
- : int = 3

# let f x y = x * y * y;;
val f : int -> int -> int = <fun>
# renvoie_3 f;;
- : int = 3
```

Polymorphisme utile

```
# let minimum comp x y = if comp x y then x else y;;
val minimum : ('a -> 'a -> bool) ->
          'a -> 'a -> 'a = <fun>
```

```
# let ordre1 x y = abs x < abs y;;
val ordre1 : int -> int -> bool = <fun>
# minimum ordre1 (-10) 2;;
- : int = 2
# let min1 = minimum ordre1;;
val min1 : int -> int -> int = <fun>
```

```
# let ordre2 x y =
     String.length x < String.length y;;
val ordre2 : string -> string -> bool = <fun>
# minimum ordre2 "toto" "ok";;
- : string = "ok"
# let min2 = minimum ordre2;;
val min2 : string -> string -> string = <fun>
```

Plan

Définition

↪ Tout algorithme qui fait appel à lui-même.

↪ *Exemples.*

- $s_0 = 0,\ s_n = s_{n-1} + n.$
- $0! = 1,\ n! = n \cdot (n-1)!.$
- $F_0 = F_1 = 1,\ F_{n+2} = F_{n+1} + F_n.$

Fonctions récursives

↪ Syntaxe. `let rec`.

↪ Exemple.

```
# let rec fact n =
    if n <= 0 then 1
    else n * fact(n-1);;
val fact : int -> int = <fun>

# fact 3;;
- : int = 6
```

Traçage

↪ Traçage de la fonction.

```
# #trace fact;;

# fact 3;;
fact <-- 3
fact <-- 2
fact <-- 1
fact <-- 0
fact --> 1
fact --> 1
fact --> 2
fact --> 6
- : int = 6

# #untrace fact;;
```

Un exemple plus graphique

```
# let rec f n =
  if n = 0 then []
  else (if n mod 2 = 0 then '.' :: (f (n-1))
        else '*' :: f (n-1));;
val f : int -> char list = <fun>

# f 5;;
- : char list = ['*'; '.'; '*'; '.'; '*']
```

Fonctionnement

↪ Présence d'une *pile*.

↪ À chaque appel de la fonction :

- Empilation d'un *bloc d'activation* contenant les paramètres d'appel, les variables loclaes, la valeur de retour.
- Dès que la valeur de retour est instanciée, la fonction termine et le bloc d'activation est *dépilé*.

↪ Pile pour l'appel `fact 3`.

			0 1			
		1 .	1 .	1 1		
	2 .	2 .	2 .	2 .	2 2	
3 .	3 .	3 .	3 .	3 .	3 .	3 6

Fonctions mutuellement récursives (I)

↪ Test de parité

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

```
# pair 123;;
- : bool = false
```

```
# impair 123;;
- : bool = true
```

Fonctions mutuellement récursives (II)

↪ Moyenne arithmético-géométrique

$$a_0 = u \qquad b_0 = v$$

$$a_n = \frac{a_{n-1} + b_{n-1}}{2} \qquad b_n = \sqrt{a_{n-1} b_{n-1}}$$

```
let rec a u v n =
  if n = 0 then u
  else ((a u v (n-1)) +. (b u v (n-1)))/. 2.
and
    b u v n =
  if n = 0 then v
  else sqrt((a u v (n-1)) *. (b u v (n-1))) ;;
val a : float -> float -> int -> float = <fun>
val b : float -> float -> int -> float = <fun>
```

```
# a 10. 2. 10;;
- : float = 5.20801638106
```

```
# b 10. 2. 10;;
- : float = 5.20801638106
```