

Programmation Fonctionnelle

Cours 2

Expressions simples et définitions

Delia Kesner

Définitions de valeurs globales et locales

- ▶ Définition **globale** simple: `let id = expr1`
- ▶ Définition **globale** multiple: `let id1 = expr1 and id2 = expr1`
- ▶ Définition **locale** simple: `let id = expr1 in expr2`
- ▶ Définition **locale** multiple: `let id1 = expr1 and id2 = expr2 in expr2`
- ▶ Les noms des identificateurs sont formés de lettres, chiffres, `_`, apostrophe `'`, et commencent toujours par une lettre en minuscule (mais pas les tirets `-`).

Exemples définitions globales I

```
let x = 2+3;;
```

```
# val x : int = 5
```

```
x*5;;
```

```
# - : int = 25
```

```
let y = 2*x;;
```

```
# val y : int = 10
```

```
let x = 42 and y=12;;
```

```
# val x : int = 42
```

```
val y : int = 12
```

```
y;;
```

```
# - : int = 12
```

Exemples définitions globales II

```
z;;
```

```
z;;
```

```
^
```

```
Error: Unbound value z
```

```
let z=5;;
```

```
# val z : int = 5
```

```
x=z;;
```

```
# - : bool = false
```

```
let x=0;;
```

```
# val x : int = 0
```

Exemples définitions globales III

```
y;;  
# - : int = 12
```

```
let z=x+1;;  
# val z : int = 1
```

Exemples définitions locales I

```
let x = 4+5 in 2*x;;
```

```
# - : int = 18
```

```
x;;
```

```
x;;
```

```
^
```

```
Error: Unbound value x
```

```
let x = 17;;
```

```
# val x : int = 17
```

```
x;;
```

```
# - : int = 17
```

Exemples définitions locales II

```
let y = x+1 in y/3;;
```

```
# - : int = 6
```

```
let x=2;;
```

```
# val x : int = 2
```

```
let x=5 and y=7 in x+7;;
```

```
let x=5 and y=7 in x+7;;  
      ^
```

```
Warning 26: unused variable y.
```

```
- : int = 12
```

```
x;;
```

```
# - : int = 2
```

Visibilité des liaisons

```
let x = 1;;  
:  
let x = 2 in  
  :  
  let x = 3 in  
    :  
    :  
    :  
  :  
  :  
  :
```

}x = 1
}x = 2
}x = 3
}x = 2
}x = 1

Seulement la liaison la plus locale est visible.

Les fonctions

- ▶ Définition **anonyme** d'une fonction à un argument :
`function id -> expr`
- ▶ Définition **anonyme** d'une fonction à plusieurs arguments :
`fun id1 ... idn -> expr`
- ▶ Définition **globale** d'une fonction à un argument :
`let id1 id2 = expr`
- ▶ Définition **locale** d'une fonction à un argument :
`let id1 id2 = expr1 in expr2`
- ▶ Règles de portées comme avant

Fonctions anonymes à un argument

▶ Syntaxe: `function id -> expr`

▶ Exemple:

```
# function x -> x+1;;  
-: int -> int = <fun>
```

▶ Application:

```
# (function x -> x+1) 5;;  
-: int = 6
```

▶ Associativité de l'application à gauche:

```
# (function x -> x+1) ((function x -> 3*x) 6);;  
-: int = 19
```

```
# (function x -> x+1) (function x -> 3*x) 6 ;;
```

```
Error: This function has type int -> int  
It is applied to too many arguments
```

Le type d'une fonction à un argument

► Le type d'une fonction à un argument est `typ1 -> typ2`, où `typ1` est le type de l'argument de la fonction et `typ2` est le type du résultat de la fonction.

► Exemples:

```
# function x -> x + 1;;  
-:  int -> int = <fun>  
  
# function x -> x +. 1.;;  
-:  float -> float = <fun>  
  
# function x -> x > 2.;;  
-:  float -> bool = <fun>  
  
# function x -> x > 0;;  
-:  int -> bool = <fun>  
  
# function x -> x ;;  
-:  'a -> 'a = <fun>
```

L'application d'une fonction et son type

- ▶ L'argument d'une fonction de type `typ1 -> typ2` doit nécessairement être de type `typ1`

```
# (function x -> x+1) 5;;
```

```
-: int = 6
```

```
# (function x -> x+1) 5.;;
```

```
Error: This expression has type float but an expression was  
expected of type int
```

Fonctions anonymes à plusieurs arguments

► Syntaxe:

```
# function x -> function y -> x+y;;  
-: int -> int -> int = <fun>  
# (function x -> function y -> x+y) 2 3;;  
-: int = 5
```

► Syntaxe alternative (à privilégier):

```
# fun x y -> x+y;;  
-: int -> int -> int = <fun>  
# (fun x y -> x+y) 2 3;;  
-: int = 5
```

La flèche associe à droite!

Les fonctions sont des entités de première classe

- ▶ L'argument d'une fonction peut être une fonction:

```
# function f -> f(f(2));  
-: (int -> int) -> int = <fun>  
# (function f -> f(f(2))) (function x -> x+1);;  
-: int = 4
```

- ▶ Le résultat d'une fonction peut être une fonction:

```
# (fun x y -> x+y) 1 ;;  
-: int -> int = <fun>
```

Définition globale d'une fonction

- ▶ Syntaxe: `let id = fun var1 ... varn -> expr`
`let f = fun x -> x*2;;`
`val f : int -> int = <fun>`
`f 2;;`
- : `int = 4`
`f (f 2);;`
- : `int = 8`
- ▶ Syntaxe alternative: `let id var1 ... varn = expr`
`let f x = x*2;;`
`val f : int -> int = <fun>`
`let g y = y+99;;`
`val g : int -> int = <fun>`
`g (f 2);;`
- : `int = 103`
`f (g 2);;`
- : `int = 202`

Définition locale d'une fonction

- Syntaxe: `let id var1 ... varn = expr1 in expr2`
 - # `let f x y = x+y in f 3 4;;`
 - : `int = 7`
 - # `let f x y = let g z = z*3 in (g x) + (g y);;`
 - `val f : int -> int -> int = <fun>`
 - # `f 4 8;;`
 - : `int = 36`

Définition de fonctions récursives

- ▶ Syntaxe: `let rec id = fun var1 ... varn -> expr`
let rec fact =
 fun x -> if (x=0) then 1 else x*fact (x-1);;
- ▶ Syntaxe alternative (à privilégier): `let rec id var1 ... varn = expr`
let rec fact x =
 if (x=0) then 1 else x*fact (x-1);;

Factorielle I

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

```
fact 0;;  
# - : int = 1
```

```
fact 4;;  
# - : int = 24
```

Tracer une fonction

```
# let rec fact x = if (x=0) then 1 else x*fact (x-1);;
# # trace fact;;
# fact 4;;
fact <- 4
fact <- 3
fact <- 2
fact <- 1
fact <- 0
fact -> 1
fact -> 1
fact -> 2
fact -> 6
fact -> 24
- : int = 24
```

Définition de fonctions mutuellement récursives

```
# let rec
  pair x = if (x=0) then true else impair (x-1) and
  impair x = if (x=0) then false else pair (x-1);;
val pair : int -> bool = <fun>
val impair : int -> bool = <fun>
# pair 4;;
- : bool = true
# impair 4;;
- : bool = false
```

Valeurs fonctionnelles en OCaml

Des valeurs fonctionnelles en OCaml

- ▶ Tout opérateur *infix* (opérateur binaire qui est écrit entre ses deux arguments) définit une fonction qui est dénotée par son nom entre parenthèses. Exemple : (+).
- ▶ Les constructeurs des types sommes ne sont pas de valeurs fonctionnelles (comme par exemple : :).

Des valeurs fonctionnelles I

```
( + );;
```

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

```
( * );;
```

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

```
let rec expo n f = function x ->
```

```
    if n=1 then x
```

```
        else f (expo (n-1) f x) (expo (n-1) f x);;
```

```
# val expo : int -> ('a -> 'a -> 'a) -> 'a -> 'a = <fun>
```

```
expo 5 (+) 1;;
```

```
# - : int = 16
```

```
type mytype = A | B of int;;
```

```
# type mytype = A | B of int
```

```
A;; (* OK *)
```

```
# - : mytype = A
```

Des valeurs fonctionnelles II

```
function x -> B x;; (* OK *)
```

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

```
B;; (* erreur *)
```

```
B;; (* erreur *)
```

```
^
```

```
Error: The constructor B expects 1 argument(s),  
      but is applied here to 0 argument(s)
```


Quelques remarques

- Utilisation de définitions locales pour meilleure efficacité:

```
# let f x = (x/2) + (x/2)*3;;
```

```
val f : int -> int = <fun>
```

```
# let f x = let a = x/2 in a + a*3;;
```

```
val f : int -> int = <fun>
```

- Deux let imbriqués dans une définition locale:

```
# let x =2 in let x =1 in x;;
```

```
Warning 26: unused variable x.
```

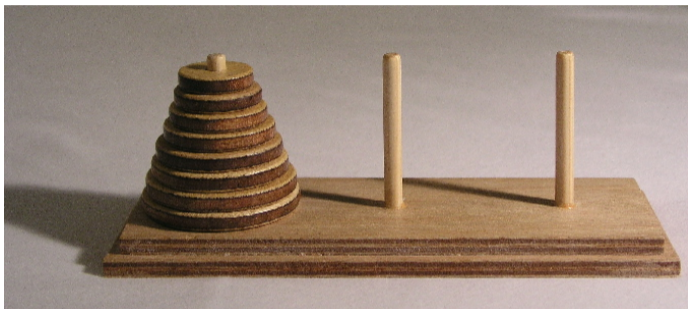
```
- : int = 1
```

Quelques remarques

- ▶ Deux let imbriqués dans une définition globale:

```
# let a =1;;  
val a : int = 1  
# let f x = x +a;;  
val f : int -> int = <fun>  
# f 3;;  
- : int = 4  
# let a = 555;;  
val a : int = 555  
# f 3 ;;  
- : int = 4
```

Les Tours de Hanoi



- ▶ déplacer la tour d'un pilier vers un autre
- ▶ déplacer un disque à la fois
- ▶ jamais placer un disque au-dessus d'un disque plus petit

Les Tours de Hanoi I

```
let bouge_disque origine destination =  
  (* deplace un seul disque de origine vers destination *)  
  print_string "Deplacer_un_disque_du_pilier";  
  print_int origine;  
  print_string "vers_le_pilier";  
  print_int destination;  
  print_endline ".";;  
# val bouge_disque : int -> int -> unit = <fun>
```

Les Tours de Hanoi II

```
let rec hanoi initial terminal auxiliaire disques =  
  (* Tours Hanoi de initial vers terminal en utilisant auxiliaire *)  
  if disques = 1  
  then bouge_disque initial terminal  
  else begin  
    hanoi initial auxiliaire terminal (disques-1);  
    bouge_disque initial terminal;  
    hanoi auxiliaire terminal initial (disques-1)  
  end;;  
# val hanoi : int -> int -> int -> int -> unit = <fun>
```

Les Tours de Hanoi III

```
hanoi 1 2 3 3;;  
# Déplacer un disque du pilier 1 vers le pilier 2.  
Déplacer un disque du pilier 1 vers le pilier 3.  
Déplacer un disque du pilier 2 vers le pilier 3.  
Déplacer un disque du pilier 1 vers le pilier 2.  
Déplacer un disque du pilier 3 vers le pilier 1.  
Déplacer un disque du pilier 3 vers le pilier 2.  
Déplacer un disque du pilier 1 vers le pilier 2.  
- : unit = ()
```