Langage CAML

Initiation à la programmation fonctionnelle pure

O. Bailleux - Version 2014

Fonctions simples a un paramètre

Exemple de définition d'une fonction

```
let abs x = if x<0 then -x else x ;;
```

- `abs x` : La fonction abs appliquée à x
- `=` : retourne
- `if x<0 then -x else x` : - x si x est négatif, sinon x

Signature de la fonction : `abs : int -> int = <fun>`

- `int -> int` : fonction qui à un int associe un int
- Les types sont inférés automatiquement par le compilateur

Appel de la fonction :

```
abs 3;;      ⟶ #- : int = 3
abs (-5);;   ⟶ #- : int = 5
```

⚠ `abs -5` ⟶ est interprété comme ⟶ `abs - 5` ⟶ Erreur !

Typage des opérateurs

Exemple d'appel qui échoue :

abs 3.8;; ⟶ Erreur : un type int est attendu.

Solution :

```
let abs x = if x < 0. then -. x else x ;;
```

Constante 0 de type float

moins unaire pour float

Signature : abs : float -> float= <fun>

Quelques opérateurs :

Pour int : - + * / mod

Pour float : -. +. *. /. **

Conversions :

float_of_int 2;; ⟶ float = 2.0

int_of_float 2.1;; ⟶ int = 2

Curryfication

Les fonctions ne peuvent avoir qu'un seul paramètre !

let sum a b = a+b;; ⟶ sum : int -> int -> int = <fun>

int -> (int -> int) : fonction qui a un **int** associe une fonction qui à un **int** associe un **int**

sum 10 15;; int = 25

sum (10 15);; ⟶ Erreur !

(sum 10) 15;; int = 25

let f = sum 10;; ⟶ f : int -> int = <fun>

f 15;; ⟶ int = 25

La fonction f associe à tout entier **a** la valeur **a+10**

Autres notations pour définir une fonction

On peut aussi utiliser la syntaxe suivante pour définir sum :

```
let sum = fun a -> fun b -> a+b;;   ⟶   sum : int -> int -> int = <fun>
```

Inspiré du λ-calcul :
λ a . λ b . (terme qui calcule a+b)

signatures différentes

Cette notation définit une *autre* fonction :

```
let sum (a,b) = a+b;;   ⟶   sum : int * int -> int = <fun>
```

Fonction qui a un couple d'entiers associe un entier

```
sum (10,15);;   ⟶   int = 25
sum (10 15);;
sum 10 15;;     ⟶   Erreur !
```

Petit test

La signature de la fonction ci-dessous est correcte :

```
let f a b c = a+(b*c);;   ⟶   f: int -> int -> int -> int = <fun>
```

1. Oui
2. Non
3. Je ne sais pas

Les deux évaluations ci-dessous sont équivalentes et donnent le même résultat :

```
(f 10) 20 30;;        f 10 20 30;;
```

1. Oui
2. Non
3. Je ne sais pas

Listes

Une liste peut être construite ou décomposée avec [] et ::

```
let m = 1 :: 2 :: 3 :: [];;   ⟶  m : int list = [1; 2; 3]
let p = 4 :: m;;              ⟶  p : int list = [4; 1; 2; 3]
```

La reconnaissance de motifs permet de décomposer une liste

```
let rec longueur w =
  match w with
  | [] -> 0;
  | x::q -> 1 + longueur q;;
```

⟶ longueur : 'a list -> int = <fun>

Fonction qui a une liste d'éléments d'un type 'a quelconque associe un entier

```
longueur [1;2;7];;  ⟶  int = 3
```

Exhaustivité des motifs

Il faut que toutes les valeurs possibles soient couvertes :

```
let rec dernier w =
  match w with
  |[e] -> e
  |x :: q -> dernier q;;
```

⟶ Erreur ! La liste vide n'est pas couverte.

Gestion simplifiée d'une erreur :

```
let rec dernier w =
  match w with
  |[] -> failwith "liste vide"
  |[e] -> e
  |x :: q -> dernier q;;
```

⟶ dernier : 'a list -> 'a = <fun>

Fonction qui a toute liste d'éléments d'un certain type 'a associe un élément de type 'a

Exemple

Une fonction qui insère un élément dans une liste triée :

```
let rec insert e w =
  match w with
   | [] -> [e]
   | x :: q ->
     if e < x then e :: w
     else x :: (insert e q);;
```

→ insert : 'a -> 'a list -> 'a list = <fun>

```
insert 7 [1;5;6;9;12];;
```
→ int list = [1; 5; 6; 7; 9; 12]

Question concernant la ligne ci-dessous :

```
let f = insert 7;;
```

1. Cette ligne est correcte, elle ne produit aucune erreur
2. Elle produit une erreur
3. Je ne sais pas

Fonctions d'ordre supérieur

Cette fonction accepte une fonction en paramètre :

```
let rec sigma f w =
  match w with
   | [] -> 0
   | x :: q -> f x + sigma f q;;
```

→ sigma : ('a -> int) -> 'a list -> int = <fun>

type de f | type de w | type retourné par (sigma f) w

Exemple d'utilisation :

```
let f x = x*x;;
```
→ f : int -> int = <fun>

```
let w = [1;2;3;4];;
```
→ int list = [1; 2; 3; 4]

```
sigma f w;;
```
→ int = 30

Quelle est la signature de sigma f ?

Variantes et application aux arbres

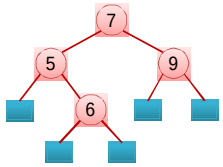
Cette déclaration définit un type arbre binaire étiqueté par des entiers :

```
type btree = E | N of int * btree * btree ;;
```

- `E` : Un arbre binaire est soit vide, représenté par **E**
- `N of int * btree * btree` : soit un arbre non vide représenté par N (x,g,d) où x est un entier, g et des arbres binaires.

Exemple :

```
let t = N (7,N (5,E ,N (6,E ,E )),N (9,E ,E ));;
```



Exemple

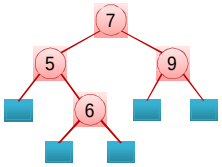
Cette fonction vérifie si un entier se trouve dans un arbre étiqueté :

```
let rec member x t =
  match t with
   |E -> false
   |N(y, left, right) ->
      if x=y then true
      else if member x left then true
      else member x right;;
```

→ member : int -> btree -> bool = <fun>

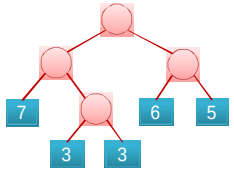
Exemple d'utilisation :

```
let t = N (7,N (5,E ,N (6,E ,E )),N (9,E ,E ));;
member 6 t;;   ⟶ bool = true
member 8 t;;   ⟶ bool = false
```

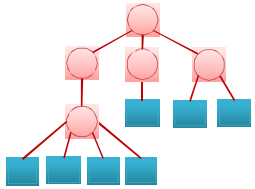


Quelques exercices

Définissez un type permettant de représenter des arbres binaire dont les nœuds n'ont pas d'étiquette, et dont les feuilles ont une étiquette de type int.



Définissez un type permettant de représenter des arbres sans étiquettes dont chaque nœud peut avoir un nombre quelconque de fils



Calculs intermédiaires

Il est parfois utile de calculer une valeur intermédiaire qui sera ensuite réutilisée plusieurs fois dans la définition d'une fonction.

```
let f x =
  let y = x*x in
  if y = 0 then 0
  else if y>0 then y-1
  else y+1;;
```

y est le résultat d'un calcul intermédiaire. Il pourra être réutilisé plusieurs fois (gain en efficacité et lisibilité)