Programmation Fonctionnelle : des concepts aux applications web (NFP119)

Fonctionnelles

Tristan Crolard

Laboratoire CEDRIC
Equipe « Systèmes Sûrs »

tristan.crolard@cnam.fr

cedric.cnam.fr/sys/crolard

Constantes et fonctions locales

Il est possible de définir des fonctions ou des variables (constantes) locales à une expression. La syntaxe générale est : `let` *déclarations* `in` *expresssion* et la visibilité des fonctions (ou des variables) définies s'arrête à l'*expression*.

Exemples

```
# let x = 1 in
  let y = 2 in
  x + y;;
- : int = 3
```

```
# let x = 2 in
  let y = 3 in
  let f (u,  v) = x * y in
  f (x, y);;
- : int = 6
```

```
# let f (x, y) =
    let g (x, y) = 2 * x + y in
    let h x = g (x, x) in
    (g (x, y)) + (h y);;
val f : int * int -> int = <fun>
```

Fonctions en paramètre

De même que dans la plupart des langages de programmation, il est possible de passer des fonctions en paramètre à d'autres fonctions. Mais contrairement aux autres langages, cette possibilité est très utilisée dans les langages fonctionnels.

Exemples

```
# let eval (f, x) = f x;;
val eval : ('a -> 'b) * 'a -> 'b = <fun>
# let rec for_all (f, l) =
    match l with
    | [] -> true
    | (h::t) -> (f h) && (for_all (f, t));;
val for_all : ('a -> bool) * 'a list -> bool = <fun>
# let rec build (f, k) =
    match k with
    | 0 -> []
    | n -> (f n)::(build (f, n-1));;
val build : (int -> 'a) * int -> 'a list = <fun>
```

Fonctions en résultat

Une fonction peut aussi renvoyer une fonction.

Exemples

```
# let comp (g,f) =
    let h x = g (f x) in
    h;;
val comp : ('a -> 'b) * ('c -> 'a) -> ('c -> 'b) = <fun>
# let addK k =
    let f x = x + k in
    f;;
val addK : int -> (int -> int) = <fun>
```

Fonctions anonymes

Il existe une syntaxe qui permet d'éviter de donner un nom à une fonction locale. Elle correspond à l'usage mathématique dans, par exemple, $x \mapsto 2x+1$ ("la fonction qui à x associe $2x+1$") ou encore $(x, y) \mapsto 3x+4y-5$ ("la fonction qui au couple (x, y) associe $3x+4y-5$"). En OCaml, la syntaxe est la suivante :

```
# fun x -> 2 * x + 1;;
- : int -> int = <fun>
# fun (x,y) -> 3 * x + 4 * y - 5;;
- : int * int -> int = <fun>
```

Exemples

```
# let comp (g, f) = fun x -> g (f x);;
val comp : ('b -> 'c) * ('a -> 'b) -> ('a -> 'c)
# let addK k = fun x -> x + k;;
val addK : int -> (int -> int)
# let addK = fun k -> fun x -> x + k;;
val addK : int -> (int -> int)
```

Fonctions « curryfiées »

Une fonction binaire peut aussi être vue comme une fonction unaire qui renvoie une fonction unaire. Par exemple, l'addition :

```
# let add2 = fun (x,y) -> x + y;;
val add2 : int * int -> int = <fun>
```

peut aussi se définir autrement :

```
# let add = fun x -> fun y -> x + y;;
val add : int -> int -> int = <fun>
```

ce qui est équivalent à :

```
# let add x = fun y -> x + y;;
val add : int -> int -> int = <fun>
```

ce qui est équivalent à :

```
# let add x y = x + y;;
val add : int -> int -> int = <fun>
```

Remarque

Cette transformation s'appelle la curryfication (la fonction `add` est la forme curryfiée de la fonction `add2`).

Transformations « curry » et « uncurry »

curry

La transformation précédente peut se programmer :

```
# let curry f = fun x -> fun y -> f (x, y)
val curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
```

ou encore :

```
# let curry f x y = f (x, y)
val curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
```

uncurry

La réciproque se programme ainsi :

```
# let uncurry g = fun (x, y) -> (g x y)
val uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c = <fun>
```

ou encore :

```
# let uncurry g (x, y) = (g x y)
val uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c = <fun>
```

map en Python 3.12

```
def map[A, B](f: Callable[[A], B], l: list[A]) -> list[B]:
    match l:
        case [h, *t]:
            return [f(h), *map(f, t)]
        case _:
            return []
```

fold_left en Python 3.12 (récursif terminal)

```python
def fold_left[A, B](f: Callable[[A, B], A], init: A, l: list[B]) -> A:
    match l:
        case [h, *t]:
            return fold_left(f, f(init, h), t)
        case _:
            return init
```

fold_left en Python 3.12 (impératif)

```python
def fold_left[A, B](f: Callable[[A, B], A], init: A, l: list[B]) -> A:
    acc = init
    for b in l:
        acc = f(acc, b)
    return acc
```

map en Java 21 (récursif)

```
static <A, B> List<B> map(Function<A, B> f, List<A> l) {
    switch (l) {
        case Nil() -> {
            return new Nil<>();
        }
        case Cons(var h, var t) -> {
            return new Cons<>(f.apply(h), map(f, t));
        }
    }
}
```

fold_left en Java 21 (récursif)

```java
static <A, B> A fold_left(BiFunction<A, B, A> f, A init, List<B> l) {
    switch (l) {
        case Nil() -> {
            return init;
        }
        case Cons(var h, var t) -> {
            return fold_left(f, f.apply(init, h), t);
        }
    }
}
```

fold_left en Java 21 (impératif)

```java
static <A, B> A fold_left2(BiFunction<A, B, A> f, A init, List<B> l) {
    var acc = init;
    for (var b : l) {
        acc = f.apply(acc, b);
    }
    return acc;
}
```