

Révisions OCaml

Quentin Fortier

September 27, 2023

Attention

Il n'est pas possible d'accéder au i ème élément d'une liste en OCaml (pas de `l.(i)`).

À la place, utiliser une fonction récursive.

Attention

Il n'est pas possible de modifier une liste en OCaml, seulement de renvoyer de nouvelles valeurs.

Par exemple, ne pas faire :

```
let l = [] in (* inutile car l ne peut pas être modifié *)  
...
```

Mais utiliser une fonction récursive, ou une référence :

```
let l = ref [] in  
l := ...
```

Attention

Il est impossible de renvoyer une valeur dans une boucle.

Ne pas écrire :

```
let appartient e t =  
  for i = 0 to Array.length t - 1 do  
    if t.(i) = e then true (* !!!??? *)  
  done;  
false
```

Mais :

```
let appartient e t =  
  let r = ref false in  
  for i = 0 to Array.length t - 1 do  
    if t.(i) = e then r := true  
  done;  
  !r
```

match

Attention

Un cas dans un **match** définit de nouvelles variables et ne permet pas de comparer des valeurs. Utiliser **if** ou **when** pour comparer des valeurs.

Ne pas écrire :

```
let rec supprime e l = match l with
| [] -> []
| e::q -> q (* e est une nouvelle variable *)
| t::q -> t::(supprime e q) (* on ne passe jamais ici *)
```

Mais :

```
let rec supprime e l = match l with
| [] -> []
| t::q -> if t = e then q
          else t::(supprime e q)
```

Attention

Ne pas confondre OCaml et Python.

	OCaml	Python
test d'égalité	=	==
test de différence	<>	!=
division euclidienne	/	//
modulo	mod	%
et	&&	and
ou		or

Syntaxe des références :

Opération	OCaml
Définition	<code>let r = ref ...</code>
Accéder à la valeur	<code>!r</code>
Modifier la valeur	<code>r := ...</code>

Exercice

Quels sont les types et complexités des fonctions suivantes ?

```
List.mem  
List.length  
List.rev  
List.filter  
List.init  
List.map  
List.iter  
List.fold_left
```

```
Array.length  
Array.make  
Array.make_matrix  
Array.copy
```

Fonctions classiques

```
List.mem : 'a -> 'a list -> bool (* O(n) *)
List.length : 'a list -> int (* O(n) *)
List.rev : 'a list -> 'a list (* O(n) *)
List.filter : ('a -> bool) -> 'a list -> 'a list (* O(n) *)
List.init : int -> (int -> 'a) -> 'a list (* O(n) *)
List.map : (a -> b) -> 'a list -> 'b list (* O(n) *)
List.iter : (a -> unit) -> 'a list -> unit (* O(n) *)
List.fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a (* O(n) *)
```

```
Array.length : 'a array -> int (* O(1) *)
Array.make : int -> 'a -> 'a array (* O(n) *)
Array.make_matrix : int -> int -> 'a -> 'a array array (* O(np) *)
```

Fonctions classiques

```
List.mem : 'a -> 'a list -> bool (* O(n) *)
List.length : 'a list -> int (* O(n) *)
List.rev : 'a list -> 'a list (* O(n) *)
List.filter : ('a -> bool) -> 'a list -> 'a list (* O(n) *)
List.init : int -> (int -> 'a) -> 'a list (* O(n) *)
List.map : (a -> b) -> 'a list -> 'b list (* O(n) *)
List.iter : (a -> unit) -> 'a list -> unit (* O(n) *)
List.fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a (* O(n) *)
```

```
Array.length : 'a array -> int (* O(1) *)
Array.make : int -> 'a -> 'a array (* O(n) *)
Array.make_matrix : int -> int -> 'a -> 'a array array (* O(np) *)
```

Remarque : Les fonctions de `List` ci-dessus existent aussi dans `Array`.

Fonctions classiques

`List.map` `f` [`e1`; `e2`; ...] renvoie [`f e1`; `f e2`; ...].

`List.filter` `f` `l` renvoie la liste des éléments `a` de `l` tels que `f a` est `true`.

`List.init` `n` `f` renvoie [`f 0`; `f 1`; ... `f (n-1)`].

Exercice

- 1 Réimplémenter ces fonctions.

Fonctions classiques

`List.map f [e1; e2; ...]` renvoie `[f e1; f e2; ...]`.

`List.filter f l` renvoie la liste des éléments `a` de `l` tels que `f a` est `true`.

`List.init n f` renvoie `[f 0; f 1; ... f (n-1)]`.

Exercice

- 1 Réimplémenter ces fonctions.
- 2 Calculer la liste des carrés des entiers pairs entre 0 et 10.

$|>$ est une notation pour appeler plusieurs fonctions à la suite.

$e \ |> \ f$ est équivalent à $f \ e$.

$e \ |> \ f \ |> \ g$ est équivalent à $g \ (f \ e)$.

Exercice

Calculer la liste des carrés des entiers pairs entre 0 et 10, en utilisant $|>$.

Application partielle

Si f est une fonction à deux arguments, alors $f \text{ a}$ est une fonction à un argument, qui fixe le premier argument de f .

Application partielle

Si f est une fonction à deux arguments, alors $f\ a$ est une fonction à un argument, qui fixe le premier argument de f .

Exemple :

```
let sum x y = x + y in
let f = sum 42 in (* f est la fonction y -> 42 + y *)
f 3 (* 45 *)
```

Application partielle

Si f est une fonction à deux arguments, alors $f\ a$ est une fonction à un argument, qui fixe le premier argument de f .

Exemple :

```
let sum x y = x + y in  
let f = sum 42 in (* f est la fonction y -> 42 + y *)  
f 3 (* 45 *)
```

Remarque : $(+)$ (version préfixe de $+$) est la même fonction que `sum`.
De même pour $(=)$, `(mod)`...

Application partielle

Si f est une fonction à deux arguments, alors $f\ a$ est une fonction à un argument, qui fixe le premier argument de f .

Exemple :

```
let sum x y = x + y in
let f = sum 42 in (* f est la fonction y -> 42 + y *)
f 3 (* 45 *)
```

Remarque : $(+)$ (version préfixe de $+$) est la même fonction que `sum`.
De même pour $(=)$, `(mod)`...

Exercice

Que vaut `List.filter ((<) 0) 1 ?`

`List.fold_left f acc l` renvoie :

- Si $l = []$: acc .
- Si $l = [e1; e2; \dots]$: $f (\dots (f (f acc e1) e2) \dots) l$

List.fold

`List.fold_left f acc l` renvoie :

- Si $l = []$: `acc`.
- Si $l = [e1; e2; \dots]$: `f (... (f (f acc e1) e2) ...) l`

Exercice

Que renvoie `List.fold_left ((+) 0) [3; 1; 4] ?`

`List.fold_left f acc l` renvoie :

- Si `l = []` : `acc`.
- Si `l = [e1; e2; ...]` : `f (... (f (f acc e1) e2) ...) l`

Exercice

Redéfinir `List.rev` à l'aide de `List.fold_left`.