Résumé OCaml 1 : variables, récursivité, listes, match

Quentin Fortier

September 22, 2021

let local

Pour définir une **variable** `a` **locale** (qui existe seulement dans le `in ...`) :

```
let a = ... in ...
```

let local

Pour définir une **variable** `a` **locale** (qui existe seulement dans le `in ...`) :

```
let a = ... in ...
```

De même pour définir une fonction `f` locale (qui existe seulement dans le `in ...`) :

```
let f x y z = ... in ...
```

let local

Pour définir une **variable** `a` **locale** (qui existe seulement dans le `in ...`) :

```
let a = ... in ...
```

De même pour définir une fonction `f` locale (qui existe seulement dans le `in ...`) :

```
let f x y z = ... in ...
```

En DS/concours on utilisera quasiment toujours des variables locales et non pas globales.

float

Si `a` est une variable de type `float`, il ne faut pas de `.` sur `a` (pas de `a.`).

ref

Opération sur les **références** (variables **mutables**) :

Création	Obtenir la valeur	Modifier la valeur
`let a = ref 3 in ...`	`!a`	`a := 7`

Pour ajouter 1 à une référence sur un entier : `incr a`

ref

Opération sur les **références** (variables **mutables**) :

Création	Obtenir la valeur	Modifier la valeur
`let a = ref 3 in ...`	`!a`	`a := 7`

Pour ajouter 1 à une référence sur un entier : `incr a`

Rappel : on ne peut pas modifier une variable si ce n'est pas une référence

let local

`let a = ... in ...` renvoie une valeur qui est la dernière instructions du `in ....`

let local

`let a = ... in ...` renvoie une valeur qui est la dernière instructions du `in ....`

```
let a = 2 in 3*a + 1
```

let local

`let a = ... in ...` renvoie une valeur qui est la dernière instructions du `in ....`

```
let a = 2 in 3*a + 1
```

Cette instruction renvoie la valeur 7, que l'on peut stocker :

```
let b = (let a = 2 in 3*a + 1)
```

`b` contient alors la valeur 7

; et ;;

; sert à séparer deux instructions qui font parties du même bloc :

```
let a = ref 3 in
a := 4;
!a  (* renvoie 4 *)
```

; et ;;

Lorsqu'on écrit ... ; ... c'est la valeur du deuxième ... qui est renvoyé.

; et ;;

Lorsqu'on écrit `...` `;` `...` c'est la valeur du deuxième `...` qui est renvoyé.

```
let b = (let a = ref 3 in a := 4; !a)
(* b vaut 4 *)
```

; et ;;

;; arrête complètement le bloc d'instruction en cours et renvoie un résultat.

; et ;;

`;;` arrête complètement le bloc d'instruction en cours et renvoie un résultat.

Il est impossible d'utiliser `;;` dans le `in` d'une fonction ou d'une variable.
J'utilise `;;` seulement pour séparer des exercices, mais ce ne sera jamais utile en DS/concours.

Fonctions récursives

Pour écrire une fonction récursive, on cherche souvent une formule de récurrence.

Par exemple, pour calculer $S_n = \sum_{k=0}^{n} k^4$, on peut utiliser le fait que :

$$S_n = \underbrace{S_{n-1}}_{\text{appel récursif}} + n^4$$

```
let rec s n =
    if n = 0 then 0
    else s (n - 1) + n*n*n*n
```

Fonctions récursives

Si on a une équation du type :

$$u_0 = 3$$

$$u_{n+1} = 2u_n + 5n$$

Il suffit de remplacer n par $n - 1$:

```
let rec u n =
    if n = 0 then 3
    else 2*(u (n - 1)) + 5*(n - 1)
```

Fonctions récursives

$$u_0 = 3$$
$$u_n = 2u_{n-1} + u_{n-1}^2$$

Au lieu de faire plusieurs appels récursifs pour calculer u_{n-1}, le stocker dans une variable :

```
let rec u n =
    if n = 0 then 3
    else let a = u (n - 1) in
         2*a + a*a
```

Suite de Fibonacci

$$u_0 = 1$$

$$u_1 = 1$$

$$u_n = u_{n-1} + u_{n-2}$$

```
let rec fibo n =
    if n <= 1 then 1
    else fibo (n - 1) + fibo (n - 2) in
fibo 10
```

Implémentation très inefficace à cause des 2 appels récursifs.

Suite de Fibonacci

```
let rec fibo n =
    if n <= 1 then 1
    else fibo (n - 1) + fibo (n - 2) in
fibo 10
```

Soit C_n = nombre d'appels récursifs de `fibo n`.

$$C_n = 2 + \underbrace{C_{n-1}}_{\text{appels récursifs de fibo (n-1)}} + \underbrace{C_{n-2}}_{\text{appels récursifs de fibo (n-2)}}$$

Suite de Fibonacci

```
let rec fibo n =
    if n <= 1 then 1
    else fibo (n - 1) + fibo (n - 2) in
fibo 10
```

Soit C_n = nombre d'appels récursifs de `fibo n`.

$$C_n = 2 + \underbrace{C_{n-1}}_{\text{appels récursifs de fibo (n-1)}} + \underbrace{C_{n-2}}_{\text{appels récursifs de fibo (n-2)}}$$

Comme $C_n \geq C_{n-1} + C_{n-2}$ et $C_{n-1} \geq C_{n-2}$:

Suite de Fibonacci

```
let rec fibo n =
    if n <= 1 then 1
    else fibo (n - 1) + fibo (n - 2) in
fibo 10
```

Soit C_n = nombre d'appels récursifs de `fibo n`.

$$C_n = 2 + \underbrace{C_{n-1}}_{\text{appels récursifs de fibo (n-1)}} + \underbrace{C_{n-2}}_{\text{appels récursifs de fibo (n-2)}}$$

Comme $C_n \geq C_{n-1} + C_{n-2}$ et $C_{n-1} \geq C_{n-2}$:

$$C_n \geq 2C_{n-2} \geq 2^2 C_{n-4} \geq 2^3 C_{n-6} \geq ... \geq 2^{\frac{n}{2}} C_{n-2\frac{n}{2}} = 2^{\frac{n}{2}}$$

$$\boxed{\forall n \in \mathbb{N}, C_n \geq 2^{\frac{n}{2}}}$$

Suite de Fibonacci

On peut utiliser un accumulateur (argument qu'on utilise pour construire le résultat) :

```
let rec fibo2 n a b =
  (* n : nombre de termes restants à calculer *)
  (* a : dernier terme calculé de la suite *)
  (* b : avant-dernier terme calculé *)
  if n = 0 then b
  else fibo2 (n - 1) (a + b) a
  (* les derniers termes deviennent a+b et a *)
```

Le nombre d'appels récursifs est alors environ n ($\ll 2^{\frac{n}{2}}$).

Tuples

Décomposer un tuple :

```
let a, b, c = t
```

ou:

```
let f (a, b, c) = ...
```

Fonctionnel vs impératif

Écrire un code de ce genre ne **sert à rien** :

```
let l = [] in  (* l ne pourra jamais être modifiée *)
for i=0 to 5 do
    i::l  (* renvoie une nouvelle liste *)
          (* mais ne modifie pas l *)
done;
l
```

Fonctionnel vs impératif

Juste mais très moche :

```
let l = ref [] in
for i=0 to 5 do
    l := i::!l
done;
!l
```

Fonctionnel vs impératif

Juste mais très moche :

```
let l = ref [] in
for i=0 to 5 do
    l := i::!l
done;
!l
```

Juste et idiomatique :

```
let rec f i =
    if i = 0 then [0]
    else i::f (i - 1) in
f 5
```

Fonctionnel vs impératif

Programmation fonctionnelle	Programmation impérative
Structures de données persistantes	Structures de données mutables
Listes	Tableaux
Fonctions récursives et match	Boucles
Variables (non mutables)	Références

Pattern matching

Pattern matching de base sur une liste `l` :

```
match l with
    | [] -> ... (* si la liste est vide *)
    | e::q -> ... (* sinon, décomposer l en e::q)
```

Pattern matching

Pour regarder les deux premiers éléments d'une liste `l` :

```
match l with
    | [] -> ...
    | [e] -> ...
    | e1::e2::q -> ...
```

Exercice

Écrire une fonction pour savoir si une liste est triée par ordre croissant.

Pattern matching

_ (underscore) permet de considérer tous les autres cas :

```
exception NotExist;

let deuxieme l = match l with
    | e1::e2::q -> e2
    | _ -> raise NotExist
```

Pattern matching

Autre façon de réunir plusieurs cas :

```
exception NotExist;

let deuxieme l = match l with
    | e1::e2::q -> e2
    | [] | [e] -> raise NotExist
```

Pattern matching

De façon plus générale, underscore permet d'éviter de nommer une variable dont on a pas besoin :

```
let rec dernier l = match l with
    | [] -> raise NotExist
    | [e] -> e
    | _::q -> dernier q
```

Pattern matching

Il est possible d'utiliser `when` pour mettre une condition sur un cas d'un match :

```
let rec appartient e l = match l with
    | [] -> false
    | x::q when x = e -> true
    | _::q -> appartient e q
```

Pattern matching

Attention : dans `|e::q -> ...`, e et q sont des nouvelles variables.

```
let rec appartient e l = match l with
    | [] -> false
    | e::q -> true (* ne marche pas *)
    | _::q -> appartient e q
```

Le e dans `| e::q ->` n'est pas le même que le e en argument de `appartient`

Pattern matching

Il est possible de décomposer à une profondeur arbitraire avec un match.

Exercice

Écrire une fonction `somme : float*float list -> float*float` pour sommer une liste de points

Pattern matching

Il est possible de décomposer à une profondeur arbitraire avec un match.

Exercice

Écrire une fonction `somme : float*float list -> float*float` pour sommer une liste de points

```
let rec somme l = match l with
    | [] -> 0., 0.
    | (x, y)::q -> let xq, yq = somme q in
                   x +. xq, y +. yq
```

Pattern matching

Si on a besoin de décomposer 2 choses, on peut match un couple.

Exercice

On représente un vecteur par une liste de flottants.
Écrire une fonction
`add : float list -> float list -> float list` pour
additionner deux vecteurs.

Pattern matching

Si on a besoin de décomposer 2 choses, on peut match un couple.

Exercice

On représente un vecteur par une liste de flottants.
Écrire une fonction
`add : float list -> float list -> float list` pour additionner deux vecteurs.

```
let rec add l1 l2 = match l1, l2 with
    | [],  [] -> []
    | e1::q1, e2::q2 -> let q = add q1 q2 in
                        (e1 +. e2)::q
    | _ -> failwith "Tailles différentes"
```