Résumé OCaml

September 14, 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`.

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

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`.

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

En minorant C_{n-1} par 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`.

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

En minorant C_{n-1} par C_{n-2} :

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

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

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 ($\ll 2^{\frac{n}{2}}$).