TD : Le language OCaml

1 Introduction

2 OCaml fonctionnel

Type de base

Exercice 1 : Xor

1. Que veut dire le type `bool -> bool -> bool` ?
2. Écrire une fonction `xor : bool -> bool -> bool` implémentant un « ou » exclusif.

Type flèche

Exercice 2 : Exercice

Donner des expressions OCaml qui correspondent aux type suivants.

1. `(int -> int) -> int`
2. `int -> (int -> int)`
3. `int -> int -> int`
4. `int -> (int -> int) -> int`

Exercice 3 : Exercice

Déterminer sans utiliser l'interpète de commandes le type des expressions suivantes.

1. `fun f x y -> f x y`
2. `fun f g x -> g (f x)`
3. `fun f g x -> (f x) + (g x)`

Exercice 4 : Exercice

Deviner la réponse de l'interpète de commande lorsqu'on saisit l'expression

```
# let h (f, g) = function x -> f (g x);;
```

Exercice 5 : Exercice

Les exercices précédents ont été facilités par le choix de `f` et `g` pour désigner implicitement des fonctions et de `x` et `y` pour des variables. Oublions maintenant cette aide pour déterminer le type des expressions suivantes.

1. `fun x y z -> (x y) z`
2. `fun x y z -> x (y z)`
3. `fun x y z -> x y z`
4. `fun x y z -> x (y z x)`
5. `fun x y z -> (x y) (z x)`

Exercice 6 : Types isomorphes

On dit que deux types `t1` et `t2` sont *isomorphes* s'il existe une fonction `f : t1 -> t2` et `g : t2 -> t1` telles que pour tout `x` de type `t1`, `g (f x) = x` et pour tout `y` de type `t2`, `f (g y) = y`.

1. Montrer que `'a * 'b` et `'b * 'a` sont isomorphes.
2. Montrer que `('a * 'b) -> 'c` et `'a -> ('b -> 'c)` sont isomorphes.

Fonction récursive

Exercice 7 : Exponentiation rapide

Le but de cet exercice est d'écrire une fonction OCaml calculant l'exponentiation de x par n, c'est-à-dire x^n avec $x \in \mathbb{R}$ et $n \in \mathbb{N}$.

1. Proposer une fonction récursive `expo_naive` qui utilise la définition par récurrence de l'exponentiation suivante :
$$x^0 = 1 \qquad \text{et} \qquad x^n = x * x^{n-1} \qquad \text{pour } n > 0.$$
Combien d'appels récursifs utilise cette fonction ?
2. Cette fois ci, on utilise la relation de récurrence suivante :
$$x^0 = 1 \qquad \text{et} \qquad x^n = \begin{cases} (x^2)^{\frac{n}{2}} & \text{si } n \text{ est pair} \\ x \times (x^2)^{\frac{n-1}{2}} & \text{sinon.} \end{cases}$$
   (a) Écrire une fonction récursive `expo_rapide` qui utilise cette définition par récurrence.
   (b) Estimer le nombre d'appels récursifs utilisés par cette fonction.

Exercice 8 : Version récursive des boucles while

Pour tout $n \geqslant 0$, on définit la *somme harmonique*
$$H_n \coloneqq \sum_{k=1}^{n} \frac{1}{k}.$$

1. Écrire une fonction `harmonique : int -> float` prenant en entrée un entier n supposé supérieur ou égal à zéro et renvoyant H_n.
2. On a vu en mathématiques que $H_n \to +\infty$ quand $n \to +\infty$. Définir mathématiquement ce que renvoie la fonction suivante.
```
let premier_n x =
  let rec aux k =
    if harmonique k >= x then k
    else aux (k + 1) in
  aux 0
```
3. Si `premier_n x` vaut n, combien d'additions vont être effectuées lors du calcul (en fonction de n) ? Raisonnablement, combien devrait-on en effectuer ?
4. Combien de temps prend le calcul de `premier_n 7.` ? de `premier_n 9.` ? de `premier_n 11.` ?
Vous pouvez mesurer très approximativement « à la main » ou utiliser la fonction `Sys.time` *si vous voulez être plus précis. Pour connaitre*
5. Écrire une version plus efficace de `premier_n` et comparer les temps de calcul expérimentaux.
On pourra utiliser une fonction auxiliaire `aux k s` *en maintenant l'invariant* $s = \sum_{i=1}^{k} \frac{1}{i}$.

Exercice 9 : Exercice

Tout entier strictement positif n peut être encadré entre deux puissances de 2 consécutives : $2^k \leqslant n < 2^{k+1}$. Dans ce cas, on dira que le *logarithme discret* $\lfloor \log_2 n \rfloor$ de n est égal à k. Cela revient à définir
$$\lfloor \log_2 n \rfloor \coloneqq \max\{k \in \mathbb{N} \mid 2^k \leqslant n\}$$

1. Écrire une fonction `deux_puissance : int -> int` telle que `deux_puissance k` renvoie l'entier 2^k pour $k \geqslant 0$.
2. En utilisant cette fonction, écrire une fonction `log2 : int -> int` prenant en entrée un entier n supposé strictement positif et renvoyant son logarithme discret. On pourra compléter le squelette suivant
```
let log2 n =
  (* aux k renvoie le plus grand k' tel que
   * 2^k' <= n, en supposant 2^k <= n.      *)
  let rec aux k =
    ...
    ... in
  aux ...
```
3. Montrer que si $n \geqslant 2$, alors $\lfloor \log_2 n \rfloor = 1 + \lfloor \log_2 \lfloor n/2 \rfloor \rfloor$.
4. En déduire une version beaucoup plus simple de la fonction `log2`.

Liste

Exercice 10 : Somme et produit

1. Définir une fonction calculant la somme des éléments d'une liste d'entiers.
2. Faire de même avec le produit.
3. Quelle est leur complexité temporelle ?

Exercice 11 : Double

Écrire une fonction `double (u : 'a list) : 'a list` qui à une liste $[a_1; \ldots; a_n]$ associe la liste $[a_1; a_1; \ldots; a_n; a_n]$.

Exercice 12 : Répète

1. Écrire une fonction `repete (u : 'a list) (n : int) : 'a list` qui renvoie la liste obtenue en concaténant n listes u. On utilisera pour cela l'opérateur de concaténation `@`.

```
# repete [ 1; 2; 3 ] 4;;
- : int list = [1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3]
```

2. Après avoir rappelé la complexité de l'opérateur `@`, déterminer la complexité temporelle de cette fonction.

Exercice 13 : Le dernier élément

1. (a) Écrire une fonction qui renvoie le dernier élément d'une liste, s'il existe, et une exception signalant que la liste est vide dans le cas contraire.
   (b) Quelle est la complexité temporelle de cette fonction ?
2. Faire de même avec une fonction renvoyant l'avant-dernier élément.

Exercice 14 : Liste des préfixes

1. Écrire une fonction `prepend (x : 'a) (u : 'a list list) : 'a list list` qui ajoute x en tête de chaque liste de la liste `u`. Quelle est la complexité temporelle de cette fonction ?
2. (a) En déduire une fonction `prefixes (u : 'a list) : 'a list list` calculant la liste de tous les préfixes de la liste `u`. Par exemple :

```
# prefixes [ 1; 2; 3; 4 ];;
- : int list list = [[1]; [1; 2]; [1; 2; 3]; [1; 2; 3; 4]]
```

   (b) Quelle est la complexité temporelle de `prefixes` ?

Exercice 15 : Liste monotone

1. Écrire une fonction `est_croissante (u : 'a list) : bool` qui détermine si la liste $|u|$ est croissante. Montrer que le nombre de comparaisons $C(u)$ effectuées par cette fonction vérifie $C(u) \leqslant |u|$.
2. (a) En déduire une fonction `est_monotone (u : 'a list) : bool` qui détermine si une liste est monotone. Majorer le nombre de comparaisons effectué par cette fonction.
   (b) On suppose qu'on dispose d'une fonction `compare (x : 'a) (y : 'a) : int` qui renvoie -1 si $x < y$, 0 si $x = y$ et 1 si $x > y$. Écrire une fonction `est_monotone (u : 'a list) : bool` ayant les mêmes spécifications que celle de la question précédente et telle que le nombre d'appels à `compare` est majoré par $|u|$.

Exercice 16 : Partition

Écrire une fonction `partition (p : 'a -> bool) (u : 'a list) : ('a list * 'a list)` qui prend un prédicat p et une liste u et renvoie le couple (v, w) où v est la liste des éléments de u vérifiant le prédicat et w celle des éléments de u ne le vérifiant pas. L'ordre des éléments devra être conservé.

```
# partition (fun x -> x > 0) [ 2; -1; -5; 4; 6; 0; -4 ];;
- : int list * int list = ([2; 4; 6], [-1; -5; 0; -4])
```

Exercice 17 : Zip

1. Écrire une fonction `zip (u : 'a list) (v : 'b list) : ('a * 'b) list` qui prend deux listes $[x_1; \ldots; x_n]$ et $[y_1; \ldots; y_n]$ et renvoie $[(x_1, y_1); \ldots; (x_n, y_n)]$. La fonction lèvera une exception si les listes ne sont pas de même longueur.

```
# zip [ 1; 2; 4 ] [ "a"; "c"; "b" ];;
- : (int * string) list = [(1, "a"); (2, "c"); (4, "b")]

# zip [ 1; 2; 4; 5 ] [ "a"; "c"; "b" ];;
Exception: Failure "Longueurs differentes"
```

2. Écrire une fonction `unzip` telle que `unzip (zip u v)` renvoie `(u, v)`.

```
# unzip (zip [ 1; 2; 4 ] [ "a"; "c"; "b" ]);;
- : int list * string list = ([1; 2; 4], ["a"; "c"; "b"])
```

Exercice 18 : Sommes cumulées

On souhaite écrire une fonction `sommes_cumulees (u : int list) : int list` qui renvoie

$$[x_1; x_1 + x_2; \ldots; x_1 + \cdots + x_n]$$

lorsqu'on l'appelle sur $[x_1; \ldots; x_n]$.

```
# sommes_cumulees [ 2; 3; 5; 31 ];;
- : int list = [2; 5; 10; 41]
```

1. (a) Écrire une fonction `ajoute (x : int) (u : int list) : int list` qui ajoute x à tous les éléments de la liste u.
   (b) En déduire une première version de `sommes_cumulees`. Quelle est sa complexité temporelle ?
2. Écrire une version de `sommes_cumulees` dont la complexité temporelle est en $\Theta(|u|)$.

Exercice 19 : Run-length encoding

Le principe du *run-length encoding* est de remplacer, dans une liste, k occurrences consécutives d'une même valeur x par le couple (x, k). C'est la méthode de compression la plus simple que l'on puisse imaginer, ce qui ne l'empêche pas d'être très utilisée même si elle est très rarement utilisée seule.

1. Écrire une fonction `compresse : 'a list -> ('a * int) list` réalisant la « compression ».
2. Écrire une fonction `decompresse : ('a * int list) -> 'a list` réalisant la « décompression ».

```
# let l = compresse [ "a"; "b"; "b"; "b"; "a"; "c"; "c"; "d" ];;
val l : (string * int) list =
[("a", 1); ("b", 3); ("a", 1); ("c", 2); ("d", 1)]

# decompresse l;;
- : string list = [ "a"; "b"; "b"; "b"; "a"; "c"; "c"; "d" ]
```

Exercice 20 : Listes associatives

Dans cet exercice, l'idée est de créer une *liste associative* représentant le fait qu'on associe à des valeurs x_k de type `'a` des valeurs y_k de type `'b`. Plus précisément, une liste associative est une liste de type `('a * 'b) list`.

1. Écrire une fonction `assoc (x : 'a) (lst : 'a * 'b) : 'b` qui pour une valeur x et une liste

$$[(x_1, y_1); \ldots; (x_n, y_n)]$$

renvoie un élément y_k tel que $x_k = x$, et lève une exception `Not_Found` si un tel élément n'existe pas. *On peut lever l'exception avec la syntaxe* `raise UneException`.

2. Écrire une fonction `assoc_opt (x : 'a) (lst : 'a * 'b) : 'b option` qui a le même comportement, mais renvoie `Some yk` au lieu de y_k et `None` sinon.

Tableau

Exercice 21 : Exercice

Écrire une fonction `appartient : 'a -> 'a array -> bool` telle que `appartient x t` renvoie `true` si et seulement si l'élément `x` apparaît dans le tableau `t`.

Exercice 22 : Exercice

Écrire une fonction `croissant : 'a array -> bool` qui détermine si le tableau passé en argument est croissant (au sens large). Si possible, on arrêtera le parcours dès que la réponse est connue.

Exercice 23 : Exercice

Pour une séquence $s = s_0, \ldots, s_{n-1}$, on définit le *miroir* de s par $\overline{s} = s_{n-1}, \ldots, s_0$. Une séquence est un *palindrome* si $s = \overline{s}$. Écrire une fonction `est_palindrome : 'a array -> bool` qui détermine si le tableau passé en argument est un palindrome. On essaiera de limiter au maximum le nombre de comparaisons effectuées.

```
# est_palindrome [|2; 1; 1; 3; 1; 1; 2|];;
- : bool = true
# est_palindrome [|2; 1; 1; 3; 3; 1; 1; 2|];;
- : bool = true
# est_palindrome [|2; 1; 1; 3; 3; 1; 2; 2|];;
- : bool = false
```

Exercice 24 : Exercice

Écrire une fonction `swap : 'a array -> int -> int -> unit` telle que, après l'appel `swap t i j`, les éléments de `t` d'indices `i` et `j` aient été échangés.

Exercice 25 : Exercice

Écrire une fonction `renverse : 'a array -> unit` telle que, si l'on a initialement `t = [| t_0; ...; t_{k-1} |]`, alors, après l'appel `renverse t`, on a `[| t_{k-1}; ...; t_0 |]`. On travaillera « en place » sans créer de tableau auxiliaire, et l'on fera bien attention à ce que la fonction soit correcte indépendamment de la parité de la longueur du tableau considéré.

```
# let t = [|4; 1; 2; 5|];;
val t : int array = [|4; 1; 2; 5|]
# renverse t;;
- : unit = ()
# t;;
- : int array = [|5; 2; 1; 4|]
```

Exercice 26 : Exercice

1. Écrire une fonction `cherche_somme (t : int array) (s : int) : (int * int) option` ayant la spécification suivante :
   — si la fonction renvoie `Some (i, j)`, alors on a $0 \leqslant i \leqslant j < |t|$ et `t.(i) + t.(j) = s` ;
   — si la fonction renvoie `None`, alors il n'existe pas de couple (i, j) vérifiant $0 \leqslant i \leqslant j < |t|$ et `t.(i) + t.(j) = s`.

   ```
   # cherche_somme [|15; 1; 3; 5; 6; 7; 10; 1; 8|] 11;;
   - : (int * int) option = Some (6, 7)
   (* Some (1, 6) ou Some (2, 8) conviennent également. *)

   # cherche_somme [|15; 1; 3; 5; 6; 7; 10; 1; 8|] 14;;
   - : (int * int) option = Some (5, 5)

   # cherche_somme [|15; 1; 3; 5; 6; 7; 10; 1; 8|] 19;;
   - : (int * int) option = None
   ```

2. On suppose maintenant que le tableau `t` est trié par ordre croissant. Écrire une fonction `cherche_somme_croissant` ayant la même spécification que `cherche_somme` mais de complexité linéaire en la taille du tableau.

```
# cherche_somme_croissant [|1; 1; 3; 5; 6; 7; 7; 10; 12; 15|] 13;;
- : (int * int) option = Some (0, 8)
# cherche_somme_croissant [|1; 1; 3; 5; 6; 7; 7; 10; 12; 15|] 17;;
- : (int * int) option = Some (3, 8)
```

Exercice 27 : Exercice

Écrire une fonction `indices_mini (t : 'a array) : int list` renvoyant la liste des indices i tels que `t.(i)` $=$ min `t`. On commencera par écrire la version la plus naturelle puis on pourra essayer d'écrire une version ne faisant qu'un seul parcours du tableau `t`.

```
# indices_mini [|10; 2; 7; 1; 7; 1; 3; 7|];;
- : int list = [3; 5]
```

Exercice 28 : Exercice

Écrire une fonction `filtre (pred : 'a -> bool) (t : 'a array) -> 'a list` qui renvoie la liste des `t.(i)` tels que `pred t.(i) = true`, classée dans l'ordre des i croissants.

```
# filtre (fun n -> n mod 2 = 0) [|1; 4; 2; 5; 7; 8; 7; 10; 4; 5|];;
- : int list = [4; 2; 8; 10; 4]
```

Exercice 29 : Section équilibrée

On considère un tableau de booléens t de longueur n. Pour $0 \leqslant a \leqslant b \leqslant n$, on note $t[a, b[$ la séquence $t_a, \ldots, t_{b-1}$ (elle est vide si $a = b$). La longueur de $t[a, b[$ est définie comme $b - a$. On définit alors

— vrai(a, b) comme le nombre de `true` dans la section $t[a, b[$

$$\text{vrai}(a, b) = \text{Card}\{i \in [\![a, b[\![\mid \texttt{t.(i) = true}\}.$$

— faux(a, b) comme le nombre de `false` dans la section $t[a, b[$

$$\text{faux}(a, b) = \text{Card}\{i \in [\![a, b[\![\mid \texttt{t.(i) = false}\}.$$

— $\delta(a, b)$ comme l'écart entre les deux

$$\delta(a, b) = \text{vrai}(a, b) - \text{faux}(a, b).$$

Une section $t[a, b[$ sera dite *équilibrée* lorsque $\delta(a, b) = 0$, c'est-à-dire lorsqu'elle contient autant de `true` que de `false`. Le but de l'exercice est de déterminer efficacement la longueur maximale d'une section équilibrée d'un tableau t.

1. Écrire une fonction `est_equilibree (t : bool array) (a : int) (b : int) : bool` qui détermine si la section $t[a, b[$ est équilibrée.
2. Écrire une fonction `max_equilibree (t : bool array) : int` qui calcule la longueur maximale d'une section équilibrée de `t` en testant toutes les possibilités. Quelle est la complexité de cette fonction ?
3. Écrire une fonction `max_depuis (t : bool array) (a : int) : int` qui renvoie la longueur maximale d'une section équilibrée de t de la forme $t[a, b[$ (où a est fixé car donné en argument). Cette fonction doit avoir une complexité linéaire en $|t|$.
4. En déduire une version plus efficace de `max_equilibree` et déterminer sa complexité.
5. Trouver une relation entre $\delta(a, b)$, $\delta(0, a)$ et $\delta(0, b)$ et en déduire une version de `max_equilibree` de complexité linéaire en la taille de t.

Donner le type des fonctions suivantes ou indiquer l'incohérence de type si elle existe :

```
let f x y = x + y

let g x y z w = (x + y, z * w)

let h y =
  let f x = x *. y in
  f 0 + f 1
```