Fonctions utiles

Cet aide-mémoire regroupe les fonctions les plus utiles du langage OCaml. Les fonctions pour lesquelles est indiqué Programme sont explicitement mentionnées dans le programme de l'option informatique, et peuvent être utilisées sans crainte **sauf mention explicite du sujet**. Les autres fonctions mentionnées devraient pouvoir être utilisées également, sous réserve qu'elles le soient de façon correcte et pertinente.

Les fonctions mathématiques (`log`, `exp`, `sin`, `cos`…) ne sont pas mentionnées ici mais peuvent évidemment être utilisées si elles sont utiles, et que le sujet n'attend pas que vous les définissiez vous-même (il serait malvenu pour de nombreuses raisons d'utiliser `log` si l'on vous demande une fonction calculant le logarithme binaire d'un entier). Pour le reste, il est probablement préférable d'essayer de s'en tenir à cette liste.

1 Général

`fst : 'a * 'b -> 'a`

Renvoie le premier élément du couple passé en argument.

`snd : 'a * 'b -> 'b`

Renvoie le second élément du couple passé en argument.

`max : 'a -> 'a -> 'a`

Renvoie le plus grand de ses deux arguments (le premier en cas d'égalité).

`min : 'a -> 'a -> 'a`

Renvoie le plus petit de ses deux arguments (le second en cas d'égalité).

`compare : 'a -> 'a -> int`

Renvoie un entier positif si le premier argument est strictement supérieur au second, un entier négatif s'il lui est strictement inférieur, et 0 si les deux arguments sont égaux. Cette fonction est typiquement destinée à être utilisée avec des fonctions de tri telles que `Array.sort`.

`abs : int -> int`

Renvoie la valeur absolue de son argument (note : « `abs min_int` » retourne un résultat négatif par débordement).

`failwith : string -> 'a` Programme

Lève une exception `Failure` paramétrée par la chaîne passée en argument. Cette fonction est destinée à provoquer l'arrêt immédiat du programme, en fournissant une information sur l'erreur rencontrée.

`ignore : 'a -> unit`

Prend en argument un objet quelconque et retourne un objet `()` de type `unit`. Cette fonction sert généralement à ignorer la valeur retournée par une fonction lorsque l'on n'en a pas l'usage.

`incr : int ref -> unit`

Incrémente la référence entière fournie en argument.

`decr : int ref -> unit`

Décrémente la référence entière fournie en argument.

`char_of_int : int -> char`

Renvoie le caractère dont le point de code est fourni en argument.

`int_of_char : char -> int`

Renvoie le point de code du caractère fourni en argument.

`string_of_int : int -> string`

Convertit un entier en une chaîne de caractères contenant sa représentation en base 10.

`int_of_string : string -> int`

Convertit une chaîne de caractères contenant la représentation d'un entier en base 10, 16 (si la chaîne commence par « `0x` ») ou 8 (si la chaîne commence par « `0o` ») en l'entier correspondant. Lève une exception `Invalid_argument` si la chaîne ne contient pas une telle représentation.

`string_of_bool : bool -> string`

Convertit un booléen en la chaîne de caractères correspondante (`"true"` ou `"false"`).

`bool_of_string`

Convertit une chaîne de caractères en un booléen. Lève une exception `Invalid_argument` si la chaîne n'est pas `"true"` ou `"false"`).

`print_int : int -> unit` Programme

Affiche dans la sortie standard la représentation en base 10 de l'entier fourni en argument.

`print_float : float -> unit` Programme

Affiche dans la sortie standard le flottant fourni en argument.

`print_string : string -> unit` Programme

Affiche dans la sortie standard la chaîne de caractères fournie en argument.

`print_char : char -> unit`

Affiche dans la sortie standard le caractère fourni en argument.

`print_newline : unit -> unit`

Provoque un retour à la ligne sur la sortie standard.

`read_line : unit -> string`

Renvoie, sous forme de chaîne, une ligne lue sur l'entrée standard (sans retour à la ligne).

2 Listes

`List.length : 'a list -> int` Programme

Renvoie le nombre d'éléments dans la liste fournie en argument. Cette fonction a une complexité $O(n)$ linéaire en la taille de la liste, et est à utiliser avec discernement. Pour tester si une liste est vide, on préférera « `lst = []` » ($O(1)$) à une condition sur sa longueur.

`List.hd : 'a list -> 'a`

Renvoie le premier élément de la liste fournie en argument. $O(1)$.

`List.tl : 'a list -> 'a`

Renvoie la liste fournie en argument privée de son premier élément. $O(1)$.

`List.nth : 'a list -> int -> 'a`

« `List.nth pos lst` » renvoie l'élément en position d'index `pos` dans la liste `lst` fournie en argument. Cette fonction a une complexité en $\Theta(\texttt{pos})$ et est donc à utiliser avec discernement.

`List.rev : 'a list -> 'a list`

Retourne une nouvelle liste contenant les éléments de la liste fournie en argument en ordre inverse.

`List.init : int -> (int -> 'a) -> 'a list`

« `List.init n f` » crée la liste `[f 0; f 1; ...; f (n-1)]` à `n` éléments obtenus en appelant la fonction `f` successivement avec les entiers de `0` à `n-1` (dans cet ordre).

```
# List.init 5 (fun i -> 2*i+1);;
- : int list = [1; 3; 5; 7; 9]
```

`List.mem : 'a -> 'a list -> bool` Programme

« `List.mem x lst` » renvoie un booléen indiquant si au moins un élément de la liste `lst` est égal à `x`. Complexité dans le pire des cas linéaire en la taille de la liste ($O(n)$).

`List.for_all : ('a -> bool) -> 'a list -> bool` Programme

« `List.for_all f lst` » renvoie un booléen indiquant si $f\,a_i$ est vrai pour tous les éléments a_i de la liste `lst`.

```
# List.for_all (fun i -> i mod 3 <> 0) [2; 3; 5; 7];;
- : bool = false
```

`List.exists : ('a -> bool) -> 'a list -> bool` Programme

« `List.exists f lst` » renvoie un booléen indiquant si $f\,a_i$ est vrai pour au moins un des éléments a_i de la liste `lst`.

```
# List.exists (fun i -> i mod 3 <> 0) [2; 3; 5; 7];;
- : bool = true
```

`List.filter : ('a -> bool) -> 'a list -> 'a list` Programme

« `List.filter f lst` » renvoie la liste des éléments a_i de la liste `lst` pour lequels $f\,a_i$ est vrai.

```
# List.filter (fun i -> i mod 3 <> 0) [2; 3; 5; 7];;
- : int list = [2; 5; 7]
```

`List.iter : ('a -> unit) -> 'a list -> unit` Programme

« `List.iter f lst` » exécute $f\,a_i$ successivement pour chacun des éléments a_i de la liste `lst` (dans l'ordre). La fonction est typiquement utilisée pour effectuer une boucle équivalente à la construction Python «`for elem in lst`».

```
# List.iter (fun i -> print_int i; print_newline ()) [2; 3; 5; 7];;
2
3
5
7
- : unit = ()
```

`List.map : ('a -> 'b) -> 'a list -> 'b list` Programme

« `List.map f lst` » renvoie la liste `[f`a_0`; f`a_1`; ...; f`a_{n-1}`]` où les a_i sont les éléments de la liste `lst`. L'ordre d'évaluation des $f\,a_i$ n'est pas spécifiée.

```
# List.map (fun i -> 2*i+1) [2; 3; 5; 7];;
- : int list = [5; 7; 11; 15]
```

`List.fold_left : ('acc -> 'a -> 'acc) -> 'acc -> 'a list -> 'acc`

« `List.fold_left f a lst` » effectue un repliement par la gauche de la liste `lst` grâce à la fonction `f` en partant de `a`. Cela revient, pour une liste `[`a_0`; `a_1`; ...; `a_{n-1}`]` au calcul de l'expression `f (... (f (f a `a_0`) `a_1`) ...) `a_{n-1}.

Si le repliement fournit parfois un moyen simple et succint de traiter une liste, il peut vite être obscur. Il peut toujours être remplacé par une récursion. Si vous optez pour l'utilisation du repliement, la syntaxe doit être irréprochable, et une explication sera attendue.

```
# List.fold_left (fun acc elem -> acc*elem) 1 [2; 3; 5; 7];;
- : int = 210
```

`List.fold_right : ('a -> 'acc -> 'acc) -> 'a list -> 'acc -> 'acc`

« `List.fold_right f lst a` » effectue un repliement par la droite de la liste `lst` grâce à la fonction `f` en partant de `a`. Cela revient, pour une liste `[`a_0`; `a_1`; ...; `a_{n-1}`]` au calcul de l'expression `f `a_0` (f `a_1` (f ... (f `a_{n-1}` a) ... ) )`.

```
# List.fold_right (fun elem acc -> acc*elem) [2; 3; 5; 7] 1;;
- : int = 210
```

3 Chaînes de caractères

`String.length : string -> int` Programme

Renvoie la longueur de la chaîne de caractère passée en argument. Contrairement aux listes, cette fonction est de complexité constante ($O(1)$), comme l'est aussi l'accès à un caractère de la chaîne.

`String.make : int -> char -> string`

« `String.make n c` » renvoie une chaîne de caractères construite par `n` répétitions du caractère `c`.

`String.init : int -> (int -> char) -> string`

« `String.init n f` » renvoie une chaîne de caractères de longueur `n` dont les caractères correspondent à la séquence `f 0`, `f 1`, ..., `f (n-1)`.

```
# String.init 26 (fun i -> char_of_int (65+i));;
- : string = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

`String.sub : string -> int -> int -> string`

« `String.sub ch i n` » retourne une nouvelle chaîne de longueur `n` correspondant aux caractères aux positions `i` à `i+n-1` de la chaîne `ch`. La fonction lève l'exception `Invalid_argument` si l'on déborde de la chaîne `ch`. Complexité linéaire en la longueur de la chaîne renvoyée.

```
# String.sub "Hello World!" 4 6;;
- : string = "o Worl"
```

`String.concat : string -> string list -> string`

« `String.concat ch lst` » retourne la chaîne de caractère obtenue par concaténation des chaînes de `lst`, en insérant la chaîne `ch` entre chacune de ces chaînes. Complexité linéaire en la longueur de la chaîne renvoyée.

```
# String.concat "**" ["Hello"; "World"; "again"];;
- : string = "Hello**World**again"
```

4 Tableaux

`Array.length : 'a array -> int` Programme

Renvoie la taille (longueur) du tableau passé en argument. Contrairement aux listes, cette fonction est de complexité constante ($O(1)$), comme l'est aussi l'accès à un élément du tableau.

`Array.make : int -> 'a -> 'a array` Programme

« `Array.make n elem` » construit et renvoie un tableau de taille `n` contenant `elem` dans chacune de ses cases. Attention, chaque case contient `elem` (et non une copie), aussi si `elem` est mutable, chaque case contient le *même* élément. Complexité linéaire en `n` ($\Theta(n)$).

`Array.init : int -> (int -> 'a) -> 'a array` Programme

« `Array.init n f` » construit et renvoie un tableau de taille `n` contenant dans la case d'index `i` le résultat de l'évaluation de `f i`. Complexité linéaire en n ($\Theta(n)$).

```
# Array.init 4 (fun i -> 2*i+1);;
- : int array = [|1; 3; 5; 7|]
```

`Array.make_matrix : int -> int -> 'a -> 'a array array` Programme

« `Array.init n p elem` » construit un tableau de taille `n`, contenant `n` tableaux *différents* de taille `p`, contenant `elem` dans chacune de leurs cases. Complexité linéaire en la taille du résultat ($\Theta(n \times p)$).

`Array.copy : 'a array -> 'a array` Programme

Renvoie un nouveau tableau, de même taille que celui passé en argument, et avec le même contenu (attention, il s'agit des *mêmes* éléments). Complexité linéaire en la taille du tableau ($\Theta(n)$).

`Array.to_list : 'a array -> 'a list`

Renvoie une liste contenant les mêmes éléments que le tableau fourni en argument, dans le même ordre. Complexité linéaire en la taille ($\Theta(n)$).

S'il est parfois pertinent de transformer un tableau en liste (ou une liste en tableau) pour un traitement, on veillera à ne pas en abuser. Ce n'est pas un moyen de se dispenser de savoir manipuler l'une ou l'autre des structures seulement !

`Array.of_list : 'a list -> 'a array`

Renvoie un tableau contenant les mêmes éléments que la liste fournie en argument, dans le même ordre. Complexité linéaire en la taille ($\Theta(n)$).

`Array.map : ('a -> 'b) -> 'a array -> 'b array` Programme

« `Array.map f arr` » renvoie un nouveau tableau dont les éléments b_i sont obtenus en évaluant `f` a_i, où les a_i sont les éléments du tableau passé en argument.

```
# Array.map (fun i -> 2*i+1) [|2; 3; 5; 7|];;
- : int array = [|5; 7; 11; 15|]
```

`Array.iter : ('a -> unit) -> 'a array -> unit` Programme

« `Array.iter f arr` » exécute `f` a_i successivement (dans l'ordre) sur chacun des éléments a_i du tableau `arr`.

```
# Array.iter (fun i -> print_int i; print_newline ()) [|2; 3; 5; 7|];;
2
3
5
7
- : unit = ()
```

`Array.for_all : ('a -> bool) -> 'a array -> bool` Programme

« `Array.for_all f arr` » renvoie un booléen indiquant si $f\,a_i$ est vrai pour tous les éléments a_i du tableau `arr`.

`Array.exists : ('a -> bool) -> 'a array -> bool` Programme

« `Array.exists f arr` » renvoie un booléen indiquant si $f\,a_i$ est vrai pour au moins un des éléments a_i du tableau `arr`.

`Array.mem : 'a -> 'a array -> bool` Programme

« `Array.mem x lst` » renvoie un booléen indiquant si au moins un élément du tableau `arr` est égal à `x`. Complexité dans le pire des cas linéaire en la taille du tableau ($O(n)$).

`Array.sort : ('a -> 'a -> int) -> 'a array -> unit`

Trie (en place) le tableau passé en argument, en utilisant la fonction de comparaison spécifiée (voir `compare` pour la fonction de comparaison). L'implémentation actuelle utilise un tri par tas, de complexité quasi-linéaire ($O(n \log n)$).

5 Pile

`Stack.create : unit -> 'a t` Programme

Crée et renvoie un objet mutable correspondant à une pile vide.

`Stack.push : 'a -> 'a t -> unit` Programme

« `Stack.push x p` » ajoute l'élément `x` au sommet de la pile `p`.

`Stack.pop : 'a t -> 'a` Programme

« `Stack.pop p` » retire et renvoie l'élément se trouvant au sommet de la pile `p`. Lève l'exception `Stack.Empty` si la pile `p` est vide.

`Stack.top : 'a t -> 'a`

« `Stack.top p` » renvoie l'élément se trouvant au sommet de la pile `p` sans le retirer. Lève l'exception `Stack.Empty` si la pile `p` est vide.

`Stack.is_empty : 'a t -> bool` Programme

Renvoie un booléen indiquant si la pile passée en argument est vide.

6 Files

`Queue.create : unit -> 'a t` Programme

Crée et renvoie un objet mutable correspondant à une file vide.

`Queue.push : 'a -> 'a t -> unit` Programme

« `Queue.push x f` » ajoute l'élément `x` à la file `f`.

`Queue.pop : 'a t -> 'a` Programme

« `Queue.pop f` » retire et renvoie l'élément se trouvant à la sortie de la file `f`. Lève l'exception `Queue.Empty` si la file `f` est vide.

`Queue.top : 'a t -> 'a`

« `Queue.top f` » renvoie l'élément se trouvant à la sortie de la file `f` sans le retirer. Lève l'exception `Queue.Empty` si la file `f` est vide.

`Queue.is_empty : 'a t -> bool` Programme

Renvoie un booléen indiquant si la file passée en argument est vide.

7 Dictionnaires

`Hashtbl.create : int -> ('a, 'b) t` Programme

Crée et renvoie un objet mutable correspondant à un dictionnaire (implémenté par une table de hachage) vide. Attends une estimation (sans importance) du nombre de clés en argument.

`Hashtbl.add : ('a, 'b) t -> 'a -> 'b -> unit` Programme

« `Hashtbl.add d c v` » ajoute une association clé-valeur (`c`, `v`) dans le dictionnaire `d`. Masque une précédente association pour `c` si elle existe. Complexité amortie constante ($O(1)$).

`Hashtbl.find : ('a, 'b) t -> 'a -> 'b` Programme

« `Hashtbl.find d c` » renvoie la valeur associée à la clé `c` dans le dictionnaire `d`. Lève l'exception `Not_found` si la clé `c` est absente. Complexité amortie constante ($O(1)$).

`Hashtbl.find_opt : ('a, 'b) t -> 'a -> 'b option` Programme

« `Hashtbl.find d c` » renvoie, sous forme d'option (`Some v`), la valeur associée à la clé `c` dans le dictionnaire `d`, et `None` si la clé `c` est absente. Complexité amortie constante ($O(1)$).

`Hashtbl.mem : ('a, 'b) t -> 'a -> bool` Programme

« `Hashtbl.find d c` » renvoie un booléen indiquant si la clé `c` est présente dans le dictionnaire `d`. Complexité amortie constante ($O(1)$).

`Hashtbl.remove : ('a, 'b) t -> 'a -> unit` Programme

« `Hashtbl.find d c` » retire la dernière association en date d'une clé `c` dans le dictionnaire `d`. Ne fait rien si la clé `c` est absente. Complexité amortie constante ($O(1)$).

`Hashtbl.replace : ('a, 'b) t -> 'a -> 'b -> unit`

« `Hashtbl.replace d c v` » ajoute une association clé-valeur (`c`, `v`) dans le dictionnaire `d`. Remplace une précédente association pour `c` si elle existe. Complexité amortie constante ($O(1)$).

`Hashtbl.iter : ('a -> 'b -> unit) -> ('a, 'b) t -> unit` Programme

« `Hashtbl.iter f d` » évalue `f` c_i v_i pour tout couple clé-valeur (c_i, v_i) présent dans le dictionnaire.