Aide-mémoire OCaml

Cet aide mémoire rappelle les types de bases en OCaml ainsi que les fonctions utilitaires associées ainsi que leurs types. Attention, toutes ces fonctions ne sont pas forcément utiles pour les exercices. Dans la suite, lorsqu'une fonction est marquée comme « opérateur binaire » (par exemple `+`) cela signifie qu'il faut l'écrire `a op b`. Sinon c'est une fonction qu'il faut appeler avec `op a b`.

Entiers

Le type `int` représente des entiers signés. Les constantes entières s'écrivent simplement `0`, `42`, `-233`. Les opérations et fonctions sur les entiers sont :

`+ : int -> int -> int` addition entre deux entiers (opérateur binaire).
`- : int -> int -> int` soustraction entre deux entiers (opérateur binaire).
`* : int -> int -> int` multiplication entre deux entiers (opérateur binaire).
`/ : int -> int -> int` division **entière** entre deux entiers (opérateur binaire).
`mod : int -> int -> int` reste dans la division entière (opérateur binaire).
`int_of_string : string -> int` conversion d'une chaîne en entier. Lève une exception si la chaîne n'est pas au bon format.
`int_of_float : float -> int` conversion d'un flottant en entier (la partie décimale est tronquée).

Flottants

Le type `float` représente des nombre flottants. Les constantes flottantes s'écrivent en notation scientifique `0.5`, `42e3`, `-233.8e-20`. Les opérations et fonctions sur les flottants sont :

`+. : float -> float -> float` addition entre deux flottants (opérateur binaire).
`-. : float -> float -> float` soustraction entre deux flottants (opérateur binaire)
`*. : float -> float -> float` multiplication entre deux flottants (opérateur binaire)
`/. : float -> float -> float` division entre deux flottants (opérateur binaire)
`** : float -> float -> float` puissance entre deux flottants (opérateur binaire)
`float_of_string : string -> float` conversion d'une chaîne en flottant. Lève une exception si la chaîne n'est pas au bon format.
`float : int -> float` conversion d'un flottant en entier (la partie décimale est tronquée).
`sqrt : float -> float` racine carrée d'un flottant.

Booléens

Le type `bool` représente des booléens. Les constantes booléennes sont `true` et `false`. Les opérations et fonctions sur les booléens sont :

`&& : bool -> bool -> bool` « et » logique entre deux booléens (opérateur binaire).
`|| : bool -> bool -> bool` « ou » logique entre deux booléens (opérateur binaire).
`not : bool -> bool` négation d'un booléen.

Chaînes de caractères

Le type `string` représente des chaînes de caractères. Les chaînes de caractères constantes sont délimitées par des guillemets :`"Hello, world !"`. De façon usuelle, la séquence d'échappement `\n` représente un retour à la ligne. Les opérations et fonctions sur les chaînes sont :

s`.[`i`]` accède au $i^{\text{ème}}$ caractère de la chaîne s. Les chaînes ne sont pas modifiables et les indicies commencent à 0.
`^ : string -> string -> string` concaténation entre deux chaînes (opérateur binaire).

`String.length : string -> int` longueur d'une chaîne.
`String.trim : string -> string` renvoie une copie de la chaîne où les blancs (espaces, tabulations, retours à la ligne) en début et en fin de chaîne ont été supprimés.
`String.split_on_char : c -> string -> string list` découpe une chaîne donnée en une liste de chaînes, en utilisant le caractère donné comme séparateur.
`String.concat : string -> string list -> string` concatène une liste de chaînes en les séparant par la chaîne de séparation donnée.

Affichage

La fonction `Printf.printf fmt arg1 arg2 ... argn`, permet d'afficher `n` arguments en utilisant la chaîne de format `fmt`. Cette dernière est une chaîne de caractères contenant des séquences spéciales :
`%d` Affichage d'un entier.
`%s` Affichage d'une chaîne.
`%f` Affichage d'un flottant.

Exemple : `Printf.printf "Salut, mon nom est %s et j'ai %d ans" "Toto" 42` affiche :

```
Salut, mon nom est Toto et j'ai 42 ans
```

Comparaisons

En OCaml les opérations de comparaison permettent de comparer n'importe quelles valeurs du même type :
`<, <=, >, >=, =, <> : 'a -> 'a -> bool` comparaisons entre deux valeurs (respectivement, inférieur, inférieur ou égal, supérieur, supérieur ou égal, égal et différent), (opérateur binaire).
`compare : 'a -> 'a -> int` comparaison générique : `compare x y` renvoie un entier négatif si `x < y`, nul si `x = y` et positif si `x > y`.
`min : 'a -> 'a -> 'a` renvoie la plus petite de deux valeurs du même type.
`max : 'a -> 'a -> 'a` renvoie la plus grande de deux valeurs du même type.

n-uplets

Les n-uplets ou produits sont délimités par des parenthèses et des virgules. Dans le cas particulier des paires, deux fonctions `fst` et `snd` permettent d'accéder à la première et seconde composante. Dans les autres cas, on peut utiliser un `let` multiple :

```
let p1 = (10, 12)
let p2 = (-1, false)
let t3 = ("A", "B", 24)

let x = fst p1 (* 10 *)
let y = snd p2 (* false *)
let a, b, n = t3  (* a vaut "A", b vaut "B" et n vaut 24 *)
```

Définitions de types

La directive `type t = ...` permet de définir un type OCaml nommé `t`. Ce type peut être :

Un produit nommé est défini en donnant pour chaque étiquette le type des valeurs associées :

```
type point_colore = { x : float; y : float; couleur : string }
```

On peut créer des valeurs enregistrement avec des accolades et accéder aux champs avec la notation `.f` :

```
let prouge = { x = 0.5; y = 10.3; couleur = "rouge" }

(* Fonction pour afficher un point coloré *)
let pr_point p = Printf.printf "<x = %f, y = %f, couleur = %s>"
        p.x p.y p.couleur
```

Un type somme est défini en donnant la liste de cas possibles :

```
type val_carte = Roi | Dame | Valet | Val of int
```

On peut créer des valeurs de ces types en utilisant les constantes ou en leur donnant un argument. On peut tester les valeurs en utilisant l'opérateur de filtrage :

```
let dix = Val 10
let as = Val 1

(* Fonction pour afficher une valeur de carte *)
let pr_val v = match v with
      Roi -> Printf.printf "%s"  "Roi"
    | Dame -> Printf.printf "%s"  "Dame"
    | Valet -> Printf.printf "%s"  "Valet"
    | Val (1) -> Printf.printf "%s"  "As"
    | Val (n) -> Printf.printf "%d"  n
```

Exceptions

En OCaml, les exceptions sont des objets particuliers permettant de signaler une erreur. On peut définir une exception avec la directive **exception** `E` **of** `...` :

```
exception MonErreur of string (* un message *)
```

On peut « lever » une exception, c'est à dire signaler une erreur au moyen de la fonction prédéfinie `raise` :

```
let err_arg_invalide () = raise (MonErreur "argument invalide")
```

Une exception non rattrapée interrompt immédiatement le programme. On peut rattraper une exception avec la construction **try with** :

```
try
  18 + (f 42)  (* f peut lever une exception *)
with
   Not_found -> (* si l'exception Not_found est levée par f *)
                10
 | MonErreur msg ->
                12
```

Si on veut signaler une erreur avec un message, la fonction prédéfinie `failwith` permet de lever une exception avec ce message en argument.

```
if x < 0.0 then
     failwith "valeur négative interdite"
else
    sqrt x
```

Listes

Le type OCaml des listes est `'a list` et permet de représenter une collection ordonnée de valeurs du type `'a`. Les listes constantes sont délimitées par des crochets et des points-virgules : `[1; 2; 42; -5]`. La liste vide est représentée par `[]`. Les opérations et fonctions sur les listes sont :

`:: : 'a -> 'a list -> 'a list` ajout en tête de liste : `1 :: l` (opérateur binaire).

`@ : 'a list -> 'a list -> 'a list` concaténation de deux listes.

`List.rev : 'a list -> 'a list` renvoie la liste avec les éléments dans l'ordre inverse.

`List.iter : ('a -> unit) -> 'a list -> unit` `List.iter f l` applique `f` à tous les éléments de `l`. La fonction `f` ne renvoie pas de résultat (par exemple elle fait un affichage).

`List.map : ('a -> 'b) -> 'a list -> 'b list` `List.map f l` applique `f` à tous les éléments de `l` et renvoie la liste des images par `f`.

`List.fold_left: ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a` `List.fold_left f init l` applique la fonction de combinaison `f` à `init` et tous les éléments de `l` dans l'ordre. Si

```
l = [ v1; v2; ... ; vn]
```

alors

```
List.fold_left f init l = (f ... (f init v1) v2) ... vn)
```

`List.filter : ('a -> bool) -> 'a list -> 'a list` `List.filter f l` renvoie la liste de tous les éléments de `l` pour lesquels `f` renvoie `true`.

`List.assoc : 'a -> ('a * 'b) list -> 'b` `List.assoc a l` renvoie la seconde composante de la première paire dans `l` qui possède `a` comme première composante. La fonction lève l'exception `Not_found` si une telle paire n'existe pas.

`List.sort : ('a -> 'a -> int) -> 'a list -> 'a list` `List.sort f l` renvoie une copie triée de `l` selon la fonction de comparaison `f`. Cette dernière suit les mêmes conventions que la fonction prédéfinie `compare`.

Enfin, on peut inspecter les listes au moyen de l'opérateur de filtrage :

```
1     (* teste si une liste est de longueur paire *)
2   let rec liste_long_paire l =
3     match l with
4     []    -> true (* liste vide est de longueur 0, pair *)
5   | [ _ ] -> false (* liste à un élément est de longueur 1, imapair *)
6   | _ :: _ :: ll -> liste_long_paire ll (* cas récursif *)
```

Boucles et séquences d'instructions

On a deux types de boucles en OCaml :

```
1   for i = 0 to n do
2     ...
3   done
```

```
1   while c do
2     ...
3   done
```

Dans le boucle **for** les deux bornes sont incluses. Dans la boucle **while**, la condition `c` devrait pouvoir varier et donc utiliser des références ou autre valeur mutable.

On peut mettre en séquence deux expressions OCaml en utilisant le « `;` ». Enfin, on peut grouper des séquences d'instructions au moyen de **begin** `...` **end**.

Modules et foncteurs

Module Un module est définit par la construction **module** `X` = **struct** `...` **end**, dans laquelle on trouve des définitions de valeurs et de types :

```
1  module Point =
2  struct
3    type t = int * int
4
5    let compare (x1, y1) (x2, y2) =
6      let c = compare x1 x2 in
7      if c = 0 then compare y1 y2 else c
8  end
```

Type de modules Un type de module est définit par la construction **module type** `T` = **sig** `...` **end**, dans laquelle on trouve des déclarations de valeurs et de types :

```
1  module type COMPARABLE =
2  sig
3    type t
4
5    val compare : t -> t - int
6  end
```

Foncteur Un foncteur est un module paramétré par un ou plusieurs autres modules. On le définit comme un module normal, mais en donnant en plus des paramètres associés à un type de module. Une fois définit, le foncteur peut être appliqué à un module concret pour obtenir un module résultat :

```
1  module SetBuilder (E : COMPARABLE) =
2  struct
3    type t = Leaf | Node of (int * t * E.t * t)
4
5    let empty = Leaf
6    let singleton e = Node (1, Leaf, e, Leaf)
7    (* ...
8      La suite peut utiliser E.compare pour comparer
9      des valeurs de types E.t
10   *)
11 end
12
13 (* On applique le foncteur *)
14 module PointSet = SetBuilder(Point)
```

Références

Le type `'a` **ref** représente les références vers des valeurs de type `'a`.

`ref : 'a -> 'a ref` création d'une référence.
r `:=` v mise à jour de la référence r avec la valeur v.
`!` r accès au contenu de la référence r.

Tableaux

Les tableaux sont des collections ordonnées de valeurs. On peut accéder aux indices d'un tableau (à partir de 0) et modifier le contenu des cases.

t`.(`i`)` accès à la $i^{\text{ème}}$ case du tableau t.

t`.(`i`) <-v` accès à la $i^{\text{ème}}$ case du tableau t.

`Array.length : 'a array -> int` renvoie la longueur du tableau t.

`Array.map : ('a -> 'b) -> 'a array -> 'b array` renvoie le tableau des images par la fonction donnée.

`Array.fold_left : ('a -> 'b -> 'a) -> 'a -> ' b array -> 'a` comme `List.map` mais sur les tableaux :

```
Array.fold_left f init t = (f ... (f init t.(0) t.(1)) ... t.(n-1))
```

Tables de hachage

Le type des tables de hachage est `('a,'b) Hashtbl.t` où `'a` est le type des clés et `'b` le type des valeurs.
`Hashtbl.create : int -> ('a, 'b) Hashtbl.t` crée une table de hachage avec une capacité initiale donnée (on peut utiliser n'importe quelle valeur, la tablea est redimentionnée automatiquement).

`Hashtbl.replace : ('a, 'b) Hashtbl.t -> 'a -> 'b -> unit` associe la clé à la valeur données dans la table (et ne renvoie rien). Si la clé était déjà présente, l'ancienne valeur est écrasée.

`Hashtbl.mem : ('a, 'b) Hashtbl.t -> 'a -> bool` teste si une clé est présente dans la table.

`Hashtbl.find : ('a, 'b) Hashtbl.t -> 'a -> 'b` renvoie la valeur associée à la clé donnée, ou lève l'exception `Not_found` si la clé est absente.

`Hashtbl.length : ('a, 'b) Hashtbl.t -> int` le nombre d'entrées présentes dans la table.

`Hashtbl.iter : ('a -> 'b -> unit) -> ('a, 'b) Hashtbl.t -> unit` appelle la fonction passée en paramètre sur toues les clés et valeurs de la table.

`Hashtbl.fold : ('a -> 'b -> 'c -> 'c) -> ('a, 'b) Hashtbl.t -> 'c -> 'c` similaire à `fold_left`. Si la table contient des clés et valeurs (`k1: v1, k2 : v2, ... ,kn : vn`) alors

```
Hashtbl.fold f table init = (f kn vn ... (f k2 v2 (f k1 v1 init)))
```