TRAVAUX PRATIQUES VI

Listes OCaml

Ce TP se concentre sur les listes OCaml. Je vous enjoins à **tester vos codes**.

Les questions de ce TP sont plutôt abstraites : nous implémentons des fonctions qui pourront servir plus tard. Le but est de bien comprendre comment les listes et les filtrages fonctionnent. Ce sont un peu vos gammes OCaml : les faire n'est pas le plus réjouissant, mais c'est en en faisant et refaisant que l'on acquiert la technique nécessaire pour jouer de jolis morceaux[1].

A Introduction

1. Code une fonction `better_print_int` qui affiche un entier suivi d'un retour à la ligne (à l'aide d'un `let ... = ... in ...`).
2. Écrire une fonction `print_bool` qui affiche la valeur d'une expression booléenne.

B Présentation de l'interpréteur utop

OCaml dispose d'un REPL (**R**ead **E**xecute **P**rint **L**oop-back), c'est à dire d'un programme où l'on peut taper des déclarations globales (resp. des expressions), et visualiser en direct ce qu'elles déclarent (resp. leur valeur).

Ce REPL (je parlerai d'interpréteur pour simplifier[2]) se nomme `utop`, et vous pouvez le lancer en tapant son nom dans le terminal. Pour le quitter, utilisez `Ctrl+D`.

Dans `utop`, vous pouvez taper des déclarations globales ou des expressions OCaml et visualiser leur résultat. Attention, il faut ajouter `;;` à la fin de votre déclaration ou expression (c'est comme ça que `utop` comprend que vous avez terminé).

3. Dans utop, recopiez la déclaration de `better_print_int` dans l'interpréteur (en ajoutant `;;` à la fin). Ensuite, testez cette fonction dans l'interpréteur.

C Cours rapide

C.1 Listes

Définition 1 (Listes OCaml, informel).
Une *liste* est une succession d'éléments (ayant le même type).
Le premier élément est appelé la **tête**. La *liste* des éléments suivants est appelée la **queue**.
Cas particulier : la liste vide est une liste, qui n'a ni tête ni queue.

Exemple.

- « `3` puis `5` puis `-2` puis `42` ». Cette suite se noterait `[3; 5; -2; 42]` en OCaml.
- la tête de la liste précédente est `3` et sa queue est `[5; -2; 42]`.

1. Cette métaphore vous est offerte par quelqu'un qui n'a jamais fait de musique en dehors de l'école.
2. Il y a techniquement une légère différence entre les deux : un interpréteur exécute du code, un REPL affiche la « valeur » d'un code et est interactif.

Définition 2 (Listes OCaml, formel).
Une liste est :

- Soit la liste vide, notée `[]` .
- Soit une tête `t` à laquelle succède la queue `q` , notée `h::q` .

Exemple.

- `7 :: (5 :: [])` est la liste contenant 7 puis 5 puis plus rien. On peut également la noter `[7;5]`.
- De même, `'c' :: ('o' :: ('e' :: ('u' :: ('r' :: []))))` est la liste contenant `'c'` puis `'e'` puis `'u'` puis `'r'`. On peut également la noter `['c'; 'o'; 'e'; 'u'; 'r']`.

C.2 Filtrage par motif

Définition 3 (Filtrage).
En OCaml, on peut faire une disjonction de cas sur le « motif » d'une expression. Par motif, j'entends :

- une valeur (par exemple `3.14` , ou `[5 ; 2]`) .
- l'application d'une règle de construction, par exemple `h :: q` .

L'expression ci-dessous permet de faire l'équivalent d'un `if-then-else`, mais en faisant une disjonction de cas sur « à quel motif correspind `truc` ? »

```
match truc with
| motif0 -> expr0
| motif1 -> expr1
...
| motifK -> exprK
```

On appelle cela un **filtrage par motif**.

Exemple. La fonction ci-dessous renvoie `true` si une liste est vide, et `false` sinon :

```
let is_empty = fun l ->
  match l with
  | [] -> true
  | h :: q -> false
```

Proposition 4 (Comment manipuler des listes).
Toute manipulation de liste doit être faite en filtrant à l'aide de ces deux motifs : `[]` et `h::t` .

Remarque. En particulier, les listes **ne sont pas** des tableaux : on ne peut pas y accéder par indice. Cela ne sert à rien d'essayer.

C.3 `failwith`

On peut volontairement déclencher une erreyr à l'aide de l'expression `failwith "message"` . Cela arrête immédiatement le code, et affiche le message donné (qui doit expliquer l'erreur).

D Premières fonctions

4. Écrire une fonction `hd` qui prend en argument une liste et en renvoie l'élement de tête (sa *head*).
5. Écrire une fonction récursive `print_int_list` qui affiche les éléments d'une liste d'entiers, séparés par des espaces (sans plus se prendre la tête sur la mise en forme).
   (Bonus) Si vous êtes motivé·e·s, vous pouvez faire en sorte que votre fonction affiche [... ;... ;...] à la place.

6. Écrire une fonction récursive `length` qui prend en argument une liste et renvoie son nombre d'éléments.
   Par exemple, `length []` vaut `0` et `length [5; 3; 7]` vaut `3`
7. Écrire une fonction récursive `int_sum` qui prend en argument une liste d'entiers et renvoie la somme de ses éléments.
   Par exemple, `length []` vaut `0` et `length [5; 3; 7]` vaut `15`
8. Écrire une fonction récursive `mem` qui prend en argument un élément, une liste et s'évalue à `true` si et seulement si cet élément est dans la liste.
   Par exemple, `mem 3 [5;3;7]` vaut `true` et `length 4 [5; 3; 7]` vaut `false`
9. Écrire une fonction `append` qui prend en argument deux listes (de même type) et renvoie la nouvelle liste constituée de la première suivie de la seconde.
   Ainsi, `append [a1; ...; an] [b1; ...; bm]` vaut `[a1; ...; an; b1; ...; bm]` .

E Typage pour les nuls

En OCaml, si `e` est une expression, on peut écrire `e : type` pour préciser que `type` est le type de `e`

Exemple.

```
let x : int =
  5-17+3
```

Les types de bases s'appellent `int` , `float` (nombres à virgule), `bool` , `char` et `string` (pour les mots, c'est à dire les successions de caractères).
Pour une fonction à 1 argument, son type est `type_entree -> type_sortie` .

Exemple. La fonction `fun x -> x+1` est de type `int -> int`

Si la fonction a plusieurs arguments, ils sont écrits « les uns après les autres, de manière curryfiée » :
`type_entree0 -> type_entree1 -> ... -> type_entreeK -> type_sortie`

Enfin, il peut arriver que le type d'une entrée/sortie ne puisse pas être déterminé : pensez par exemple à `fun x -> x` . Dans cette fonction, rien ne permet de deviner le type de `x` . Dans ce cas, on donne un type « joker » : cela signifie qu'on peut donner un `x` de n'importe quel type, cela marchera toujours. Le premier type joker utilisé s'appelle `'a` , le second `'b` , etc.

Exemple. `fun x y -> x` est de type `'a -> 'b -> 'a` .

10. Appelez-moi si vous n'avez pas compris.

Dans la suite de ces énoncés, j'indique le type des fonctions demandés.

F Fonctions de listes et booléens

Dorénavant, je n'indique plus si une fonction a besoin d'être récursive ou non.

11. Écrire une fonction `exists : ('a -> bool) -> 'a list -> bool` qui prend en argument une fonction de type `'a -> bool` (dans l'idée, une fonction qui teste si un élément vérifie une certain propriété) et une liste, et qui teste s'il existe un élément de la liste sur lequel la fonction s'évalue à `true` (dans l'idée, s'il existe un élément qui vérifie la propriété).
    `exists f [a1; ...; an]` renvoie vrai si l'un des `f a1` , ..., `f an` est vrai, et faux sinon.
12. Écrire une fonction `for_all : ('a -> bool) -> 'a list -> bool` qui prend en argument une fonction de type `'a -> bool` et une liste et qui teste si sur tous les éléments de la liste la fonction s'évalue à `true` (dans l'idée, si tous les éléments vérifient la propriété).
    `for_all [a1; ...; an]` renvoie vrai si tous les `f a1` , ..., `f an` sont vrai, et faux sinon.
13. Écrire une fonction `filter` qui prend en argument une fonction de type `'a -> bool` et une liste et qui renvoie la liste uniquement composée des éléments de la liste sur lesquels la fonction s'évalue à `true`.
    `filter f [a1; ...; an]` renvoie la liste des `ai` tels que `f ai` est vrai.

G Fonctions avancées

14. Écrire une fonction `map : ('a -> 'b) -> 'a list -> 'b list` qui prend en argument une fonction et une liste et renvoie la liste composée des images de la première par la fonction (dans le même ordre).

`map f [a1; ...; an]` renvoie la liste `[f a1; ...; f an]`.

15. Écrire une fonction `iter : ('a -> unit) -> 'a list -> unit` qui prend en argument une procédure (fonction de sortie `unit`) et une liste et applique la fonction successivement à tous les éléments de la liste.

`iter f [a1; ...; an]` a les mêmes effets secondaires que `map f [a1; ...; an]` mais renvoie uniquement `()`.

H Comparaisons et Tris

Dans cette section, on utilise le comportement suivant pour une fonction de comparaison : elles renvoient un entier négatif si le premier argument est strictement inférieur au second, 0 s'ils sont égaux, et 1 sinon.

16. Écrire une fonction `compare : ('a -> 'a -> int) -> 'a list -> 'a list -> int` qui prend en argument une fonction de comparaison, deux listes, et compare lexicographiquement les deux listes (c'est l'ordre du dictionnaire : on compare d'abord le premier élément, puis le second, etc).

Elle doit renvoyer un entier négatif si la première liste est strictement inférieure (lexicographiquement) à la seconde, 0 si elles sont égales, et un entier strictement positif sinon.

17. Écrire une fonction `sort : ('a -> 'a -> int) -> 'a list -> 'a list` qui prend en argument une fonction de comparaison, une liste, et renvoie une nouvelle liste égale à sa liste argument triée par ordre croissant. *On pourra faire un tri par insertion récursif (en $O(n^2)$) : à chaque étape, on rajoute un élément dans une liste déjà triée, comme pour trier un jeu de cartes.*

I Pour occuper les plus rapides

18. Écrire une fonction `rev` qui prend en argument une liste et renvoie cette même liste mais en ordre inverse.

`rev [a1; ...; an]` renvoie la liste `[an; ...; a1]`.

19.
a. Écrire une fonction `isole : int -> 'a list -> 'a list` telle que `isole n lst` renvoie les n premiers éléments de `lst`. Si `lst` contient moins de n éléments, elle les renvoie tous.

b. Écrire une fonction `enleve : int -> 'a list -> 'a list` telle que `enleve n lst` renvoie la liste `lst` privée de ses n premiers éléments. Si `lst` contient moins de n éléments, elle renvoie une liste vide.

20. Écrire une fonction `fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a` qui étant donnée une fonction `f`, un élément a et une liste $[b_0; b_1; ...; b_{k-1}]$ calcule `f (... (f (f a` b_0`)` b_1`) ...)` b_{k-1}. *Inspirez-vous de la recherche de maximum dans une liste, et recodez là si besoin.*

21. Faire de même une fonction `fold_right ('a -> 'b -> 'a) -> 'a list -> 'b -> 'a` qui étant donnée une fonction `f`, une liste $[a_0; a_1; ...; a_{k-1}]$ calcule `f` a_0 `(f` a_1 `(... (f` a_{k-1} `b)))`.