TRAVAUX PRATIQUES VIII

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. Recréer la fonction `affiche_int` du cours qui affiche un entier suivi d'un reto ur à la ligne (à l'aide d'un `let ... = ... in ...`).
2. Écrire une fonction `print_bool` qui, **à l'aide d'un filtrage**, affiche la valeur d'une expression booléenne.

B Rappels de cours

Rappel : une liste OCaml est une *pile* : elle est soit vide, soit constituée d'un élément au-dessous duquel se trouve une autre liste.

Définition 1

Le type `'a list` est le type des listes dont le contenu est de type `'a` . Une liste est :

- Soit la liste vide `[]`
- Soit de la forme `h::t` , où `h` est un élément de type `'a` appelé tête (*head*), et `t` est le reste de la liste appelée queue (*tail*) de la liste.

Toute manipulation de liste doit être faite en filtrant à l'aide de ces deux motifs : `[]` et `h::t` . Parfois, nous ajouterons quelques motifs supplémentaires, mais ces deux-ci sont l'alpha et l'oméga des manipulations des suites.
Exemple :

- `7 :: (5 :: [])` est la liste contenant 7 puis 5. On peut également la noter `[7;5]` (notez les point-virgule et non les virgules !).
- 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'est donc un type somme récursif [2], qui doit être manipulé avec des filtrages (et des récursions).

On ne peut accéder qu'à l'élément de la tête de pile (et on y accède via un filtrage). Ainsi, le premier élément que l'on met dans une pile se retrouve tout en bas de la pile, et sera le dernier à en ressortir. Au contraire, la dernière entrée est la première à sortir.

Proposition 1

L'ordre des éléments dans une pile suit le principe **LIFO** : **L**ast **I**n **F**irst **O**ut (en VF, dernier arrivé premier servi). Autrement dit, l'ordre de sortie des éléments d'une pile est l'inverse de l'ordre d'entrée.

Dans ce TP, on recode des fonctions de la librairie OCaml. Bien sûr, plus tard vous ne les recoderez plus mais utiliserez les versions de OCaml.

1. Cette métaphore vous est offerte par quelqu'un qui n'a jamais fait de musique en dehors de l'école.
2. `[]` et `::` sont peu ou prou des constructeurs, mais qui ne se notent pas avec une majuscule comme ceux du cours.

C Premières fonctions

3. Écrire une fonction `is_empty` qui prend en argument une liste et renvoie si elle est vide ou non.
4. Écrire une fonction `hd` qui prend en argument une liste et en renvoie l'élement de tête (sa *head*).
5. Écrire une fonction `print_int_list` qui affiche successivement les éléments d'une liste d'entiers (sans 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 `length` qui prend en argument une liste et renvoie son nombre d'éléments.
7. Écrire une fonction `int_sum` qui prend en argument une liste d'entiers et renvoie la somme de ses éléments.
8. Écrire une fonction `mem : 'a -> 'a list -> bool` qui prend en argument un élément, une liste et teste si cet élément est dans la liste.
   `mem x [a1; ...; an]` renvoie vrai si l'un des éléments `a1` , ..., `an` est égal à x, et faux sinon.
9. Écrire une fonction `append : 'a list -> 'a list -> 'a list` qui prend en argument deux listes (de même type) et renvoie la nouvelle liste constituée de la première suivie de la seconde.
   `append [a1; ...; an] [b1; ...; bm]` renvoie `[a1; ...; an; b1; ...; bm]` .

D Interlude mi hors-programme

Voici deux points confortables de OCaml qui sont mi hors-programme. Je me servirai souvent du premier, et parfois du second.

Définition 2 : when

On peut ajouter des condtions sur les motifs d'un filtrage. La syntaxe est la suivante :

```
| motif when condition -> expr
```

Le filtrage va associer `expr` si le motif est vérifié ET si la condition est vraie.

*Exemple :*La fonction ci-dessous prend en argument un deux-uplets d'entiers et en renvoie la première coordonnée si elle est paire, la seconde sinon :

OCaml
```
let demo = fun a -> match a with
| (x,y) when x mod 2 = 0 -> x
| (x,y)                  -> y
```

10. Sans utiliser de `if` (mais en utiliser des `when`), écrire une fonction qui prend en argument une liste d'entiers et n'en affiche que les entiers pairs.

Passons au second point. Il est un peu plus délicat, et ce n'est pas grave si vous ne le mémorisez pas :

Définition 3 : function

Le mot clef `function` est une abréviation pour `fun x -> match x with` , sauf que l'on ne peut pas accéder à x ensuite. Autrement dit, `function` définit une fonction qui attend un argument puis le filtre aussitôt.

*Exemple :*On peut ainsi réécrire la fonction de l'exemple précédent comme :

OCaml
```
let demo = function
| (x,y) when x mod 2 = 0 -> x
| (x,y)                  -> y
```

Ce mot-clef a l'avntage de raccourcir la déclaration en évitant de prendre un argument pour l'oublier immédiatement en le filtrant... mais l'inconvénient de masquer le fait que l'on attend un argument ! Je le réserverai aux questions difficiles.

E Fonctions de listes et booléens

Techniquement, `mem` devrait être ici. Mais elle faisait une bonne introduction.

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.

F 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 `()` .

G 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.

H 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)))`.