Mettez un peu d'ordre supérieur dans CAML

"Functionals" or "higher-order functions"

1 Ordre supérieur ?

Exercice 1.1 (Try those examples)

```
# let succ = function x -> x + 1 ;;
# succ 5 ;;
# (function x -> x + 1) 5 ;;

# let times = function x -> function y -> x * y ;;
# let double = times 2 ;;
# double 15 ;;

# let double_fun f x = double (f x) ;;
# double_fun succ 2 ;;
# double_fun (function x -> x + 1) ;;
# let double_succ = double_fun succ ;;
# double_succ 2 ;;
```

Exercice 1.2 (Sigma)

1. Écrire une fonction qui calcule la somme suivante (sans multiplication...) :

$$\sum_{i=0}^{n} i$$

2. Écrire une fonction qui calcule la somme des carrés suivante :

$$\sum_{i=0}^{n} i^2$$

3. À l'aide d'une fonction d'ordre supérieur, nous pouvons généraliser le calcul d'une somme à n'importe qu'elle fonction. Écrire une fonction qui calcule, pour toute fonction f :

$$\sum_{i=0}^{n} f(i)$$

Exercice 1.3 (Loop - Bonus)

1. Écrire une fonction `loop` qui, appliquée à un prédicat p, une fonction f et un entier x, rend la valeur $f^n(x)$ où n est le plus petit entier qui vérifie $p(f^n(x)) = Vrai$.
2. Utiliser `loop` pour définir une fonction `find_power` qui, étant donnés deux entiers x et n, retourne la première puissance de x supérieure ou égale à n.

2 The one where functions apply to lists' elements

Pour chaque fonction, donner plusieurs exemples d'utilisation.

Iterators

Exercice 2.1 (`map`)

Écrire la fonction `map` qui applique une fonction f donnée à tous les éléments d'une liste et qui construit la liste des valeurs retournées.

`map f` $[a_1;\ \ldots;\ a_n]$ retourne `[f` a_1`; f` a_2`;...; f` a_n`]`

Exercice 2.2 (`iter`)

Écrire la fonction `iter` qui applique séquentiellement une fonction f donnée à tous les éléments d'une liste.

`iter f` $[a_1;\ \cdots\ ;\ a_n]$ $\equiv$ `f` a_1`; f` a_2`;...; f` a_n `; ()`

Exercice 2.3 (`mapi` – *Midterm 1# - 2017*)

1. Écrire la fonction `mapi` dont les spécfications sont les suivantes :
   — Elle prend en paramètre une fonction à deux paramètres f, ainsi qu'une liste $[a_0; a_1; a_2; \cdots; a_n]$.
   — La fonction f prend l'index de l'élément dans la liste et l'élément en paramètres.
   — Elle retourne la liste des résultats retournés par f.

   `mapi f` $[a_0;\ a_1;\ \ldots;\ a_n]$ retourne `[f 0` a_0`; f 1` a_1`; ...; f` n a_n`]`

2. Par quoi peut-on remplacer le ? dans les évaluations ci-dessous (un seul choix possible) ?

```
# mapi (?) [2;4;6;8;10;12;14;16;18];;
- : int list = [-2; 4; -6; 8; -10; 12; -14; 16; -18]
# mapi (?) [1;2;3;4;5;6;7;8;9;0] ;;
- : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 0]
# mapi (?) [7;7;7;7;7;7;7;7;7;7] ;;
- : int list = [7; -7; 7; -7; 7; -7; 7; -7; 7; -7]
# mapi (?) [2;3;4;5;6;7;8;9] ;;
- : int list = [-2; -3; -4; -5; -6; -7; -8; -9]
```

List scanning

prédicat = fonction booléenne représentant une propriété

Exercice 2.4 (`for_all`)

Écrire la fonction `for_all` qui vérifie si tous les éléments d'une liste l vérifient le prédicat p donné.

Exercice 2.5 (`exists`)

Écrire la fonction `exists` qui vérifie si au moins un élément d'une liste l vérifie le prédicat p donné.

Exercice 2.6 (Combien ? – *Midterm 1 - 2016*)

1. Écrire la fonction CAML `how_many` dont les spécifications sont les suivantes :
   — Elle prend en paramètre une fonction booléenne f ainsi qu'une liste : $[a_1; a_2; \cdots; a_n]$.
   — Elle recherche dans la liste les valeurs a_i telle que $f(a_i)$ soit vrai et retourne le nombre de valeurs trouvées.
2. Utiliser la fonction `how_many` pour définir une fonction qui retourne le nombre de valeurs multiples d'un entier n donné dans une liste d'entiers.

Exercice 2.7 (Test it)

1. Écrire la fonction `test` qui vérifie si une liste est triée selon un ordre donné par une fonction de comparaison de type `'a -> 'a -> bool`.
   Par exemple pour tester si une liste est strictement croissante, la fonction `test` sera utilisée de la manière suivante :

```
# let increase l = test (<) l ;;
  val increase : 'a list -> bool = <fun>
```

2. Utiliser la fonction `test` pour définir une fonction qui vérifie si une liste d'entiers suit une progression géométrique de raison q.

List searching

Exercice 2.8 (`find`)

Modifier la fonction `exists` pour qu'elle retourne le premier élément de la liste vérifiant le prédicat.

Exercice 2.9 (Find it)

1. Écrire la fonction `find_place` dont les spécifications sont les suivantes :
   — Elle prend en paramètre une fonction f ainsi qu'une liste : $[a_1; a_2; \cdots; a_n]$.
   — Elle recherche dans la liste la première valeur a_i telle que $f(a_i)$ soit vrai et retourne le rang i correspondant.
   — Elle déclenche une exception si aucune valeur n'a été trouvée.
2. Utiliser la fonction `find_place` pour définir une fonction qui retourne le rang de la première valeur impaire d'une liste d'entiers.

Exercice 2.10 (`filter`)

Écrire une fonction qui, à partir d'une liste, construit une nouvelle liste contenant les éléments qui satisfont un prédicat p donné.

Exercice 2.11 (`partition`)

Écrire une fonction qui, à partir d'une liste l, construit une paire de listes (l_1, l_2), où l_1 est la liste contenant les éléments qui satisfont un prédicat p donné, et l_2 la liste des éléments ne vérifiant pas p.

On two lists

Exercice 2.12 (exists2 – *Midterm 1 - 2016*)

1. Écrire la fonction CAML `exists2` dont les spécifications sont les suivantes :
   — Elle prend en paramètre une fonction booléenne à deux paramètres : `p` ainsi que deux listes : $[a_1; a_2; \cdots; a_n]$ et $[b_1; b_2; \cdots; b_n]$.
   — Elle retourne le booléen : il existe au moins un couple (a_i, b_i) tel que $p\ a_i\ b_i$ est vrai.
   — Elle déclenche une exception si les deux listes sont de longueurs différentes (si aucun couple (a_i, b_i) tel que $p\ a_i\ b_i$ est vrai n'a été trouvé...).
2. Utiliser la fonction `exists2` pour définir une fonction qui vérifie si deux listes sont identiques.

Exercice 2.13 (Compte – *Midterm 1# - 2017*)

Écrire la fonction CAML `count2` dont les spécifications sont les suivantes :

— Elle prend en paramètre une fonction à deux paramètres : f
ainsi que deux listes : $[a_1; a_2; ...; a_n]$ et $[b_1; b_2; ...; b_n]$.
— Elle retourne le nombre de paires d'éléments (a_i, b_i) tels que $f\ a_i\ b_i$ est vrai.
— Elle déclenche une exception si les deux listes sont de longueurs différentes.

3 The last one

Exercice 3.1 (Traitement récursif)

On peut généraliser de nombreux traitements récursifs (non terminaux) d'une liste par la fonction suivante :

```
# let rec process_list f init =
    function
        [] -> init
      | e :: l -> f e (process_list f init l);;
```

— Dans le cas d'une liste vide, la fonction retourne une valeur d'arrêt : `init`
— Dans le cas d'une liste de la forme `e::l`, le résultat dépend de `e` et du traitement de la liste `l`.

On peut utiliser la fonction `process_list` pour calculer de nouvelles fonctions, sans récursivité. Par exemple, la fonction qui calcule la longueur d'une liste sera définie par :

```
# let length l = process_list (function e -> function call -> 1 + call) 0 l ;;
 val length : 'a list -> int = <fun>
```

1. (a) Quel est le type de `process_list` ?

   (b) Que calcule `process_list` f *init* `[`e_1`;...;`e_n`]` ?

2. Utiliser la fonction `process_list` pour définir, sans récursivité, les fonctions suivantes :

   (a) `sum` calcule la somme des éléments d'une liste.

   (b) `concatenate` prend en paramètre une liste de chaînes de caractères et qui retourne la chaîne de caractères obtenue en concaténant les unes aux autres dans l'ordre les chaînes de la liste. Donner des exemples d'applications de `concatenate`.

   (c) `append` concatène deux listes.

   (d) `map`, appliquée à f et $[e_1;...;e_n]$, calcule $[f(e_1);...;f(e_n)]$.

   (e) `for_all` , appliquée à une fonction booléenne p et une liste, vérifie si tous les éléments de la liste vérifient le prédicat p.

   (f) **Bonus :** `sum_list` prend en paramètre une liste de fonctions réelles $[f_1;...;f_n]$ et retourne la fonction $f_1 + \cdots + f_n$. Donner des exemples d'application de `sum_list`. Donner des exemples d'application de résultats d'applications de `sum_list`.

Exercice 3.2 (`fold_left`)

Écrire la fonction `fold_left` définie par :
`val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`
`fold_left f a [b1;...; bn]` calcule `f (...(f (f a b1) b2)...) bn`.