Option informatique : TP 3

Dans ce TP nous allons étudier de plus près quelques fonctions récursives. Il est important de fournir une documentation précise des fonctions, afin d' expliquer pourquoi elle termine, justifier de sa correction, et détailler tout choix non trivial d'implémentation. Les fonctions seront indentées correctement.
Toutes les fonctions devront être écrites dans le style récursif, sans utiliser de référence, de boucle ni de fonction d'affichage ("print_int", etc.).

1 Suite de Fibonacci

Commençons par la seconde fonction récursive la plus classique (après la factorielle), celle qui calcule les termes de la suite de Fibonacci. On rappelle que cette suite F_n est définie par

$$F_n = \begin{cases} F_1 & = 1 \\ F_2 & = 1 \\ F_{n+2} & = F_n + F_{n+1} \end{cases}$$

(a) Écrire une fonction `fib_naive : int -> int` qui calcule le n-ième terme de cette suite, directement en suivant la définition. Que peut-on dire de ses performances quand n devient grand ?
(b) Écrire une fonction `fib` du même type qui calcule le même résultat de manière plus efficace. Cette fonction doit faire n appels récursifs pour calculer F_n, pas plus — nous verrons plus tard que sa complexité est *linéaire*. On pourra pour écrire une fonction auxiliaire.
Remarque : il est possible de faire encore plus efficace en exploitant la propriété $F_n = \frac{1}{\sqrt{5}}(\varphi^n - \varphi'^n)$ avec $\varphi = \frac{1+\sqrt{5}}{2}$ le nombre d'or et $\varphi' = -\frac{1}{\varphi}$, mais ici l'important est d'étudier la récursion.

2 Fonction d'Ackermann

La fonction d'Ackermann a eu une grande importance théorique dans l'étude de la calculabilité — la question de savoir quelles fonctions sont calculables par un ordinateur.
On la définit souvent comme une fonction $A(m, n)$ de type `int -> int -> int` [1] :

$$A(m,n) = \begin{cases} n+1 & \text{si } m = 0 \\ A(m-1,1) & \text{si } m > 0 \text{ et } n = 0 \\ A(m-1, A(m, n-1)) & \text{sinon} \end{cases}$$

1. On peut aussi écrire une version décurryfiée, plus proche de la définition mathématique.

(a) Définir une fonction Caml `ackermann` qui calcule, pour des entrées dans $\mathbb{N}$, la fonction d'Ackermann appliquée à ces entrées. Pour l'instant on ne se souciera pas de justifier sa terminaison.
(b) Définir une fonction `ackermann_row : int -> int -> int vect` qui associe, à une paire `(m, n)`, le tableau
`[|ackermann m 0; ackermann m 1; ...; ackermann m n|]`. On peut utiliser une boucle `for` ou la fonction `init_vect : int -> (int -> 'a) -> 'a vect` telle que `init_vect n f = [| f 0; f 1; f 2; ...; f (n-1) |]`.
(c) Appliquer cette fonction pour (m, n) prenant les valeurs suivantes : $(0, 4)$, $(1, 4)$, $(2, 4)$, $(3, 4)$, et enfin $(4, 0)$. Qu'observe-t'on sur cette dernière entrée ? On pourra tester la fonction sur $(4, n)$ avec $n > 0$ seulement après avoir trouvé le menu permettant d'interrompre l'exécution de Caml...
(d) Étant donnés des ordres $<_A$ et $<_B$ sur des ensembles A et B, on appelle *Ordre Lexicographique* sur $A \times B$ l'ordre $<_{lex}$ défini par :

$$(x, y) <_{lex} (x', y') \text{ si } \begin{cases} x <_A x' \\ x = x' \text{ et } y <_B y' \end{cases}$$

On peut généraliser cet ordre à des n-uplets quelconques, puis à des listes [2]. Intuitivement, l'ordre lexicographique correspond à l'ordre alphabétique qu'on utilise tous les jours pour comparer des mots (et donc les trier) dans un dictionnaire.

Justifier, à l'aide d'un ordre lexicographique, la terminaison de la fonction d'Ackermann.

3 Types Algébriques

Nous avons jusqu'ici vu un unique type permettant de représenter des séquences d'éléments, le tableau. Très adapté au style impératif, il se prête mal à la récursion et de plus est *mutable* (modifiable).
En programmation fonctionnelle, on préfère généralement travailler avec des valeurs *immuables* (non-modifiables) et la récursion. Un type extrêmement utile (depuis l'invention du langage *Lisp* en 1962) s'appelle le type des *listes*.
Ce type liste est défini en Caml par un *type algébrique*. Cette construction de types existe dans peu de langages mais est extrêmement utile et se marie particulièrement bien avec le filtrage.
Un type algébrique (ou *type somme*) se définit avec une syntaxe qui rappelle celle du filtrage. Quelques exemples :

```
type nombre =
  | Entier of int
  | Rationnel of int * int
  | Reel of float
```

2. Pour des listes, on utilisera évidemment le même ordre sur chaque élément puisqu'une liste, comme un tableau, ne contient que des objets du même type.

```
;;

type 'a option =
  | None
  | Some of 'a
;;

type direction =
  | Nord
  | Sud
  | Est
  | Ouest
;;
```

Le premier type définit un type numérique flexible capable de représenter les entiers, réels et rationnels (sous forme de paire d'entiers dont le second est non nul). On remarquera que chaque *constructeur* (par exemple, `Reel`) doit commencer par une capitale.
Le second, couramment appelé « type option », est très utile pour représenter le résultat d'une fonction partielle. Si la fonction réussit, elle renvoie `Some x` où x est son résultat, sinon, elle renvoie simplement `None`. Il est prédéfini en Caml.
Le troisième montre qu'il est pratique de pouvoir définir un type algébrique pour représenter des alternatives, dans de nombreux contextes — ici les directions cardinales. On peut ensuite raisonner par cas sur la direction.

```
# None;;
- 'a option : None
# Some "foobar";;
- string option : Some "foobar"
```

On peut filtrer sur les types algébriques, de manière assez similaire au filtrage sur les n-uplets. Par exemple, si une fonction renvoie une valeur `result : int option`, on peut filtrer cette valeur ainsi :

```
match result with
    | None -> ... (* traiter le cas None *)
    | Some i -> ... (* utiliser le resultat i *)
```

Ou une fonction qui détermine si une direction cardinale est « verticale » :

```
# let vertical d = match d with
    | Nord -> true
    | Sud -> true
    | Est
    | Ouest -> false;;
vertical : direction -> bool = <fun>
```

3.1 Listes

Maintenant que nous disposons de types algébriques, nous pouvons définir la structure de liste de la manière suivante :

```
type 'a list =
    | Nil  (* liste vide *)
    | Cons of ('a * 'a list)
```

Caml fournit un type de listes prédéfini avec une syntaxe spéciale, dans laquelle `Nil` s'écrit `[]` et `a :: l` remplace `Cons (a, l)`. Le type liste pourrait toutefois être défini par l'utilisateur comme expliqué précédemment.
La syntaxe ne le permet pas, mais la définition des listes dans Caml est donc

```
type 'a list =
    | []  (* liste vide *)
    | 'a :: 'a list
```

Pour construire une liste on peut donc soit partir de la liste vide `[]`, soit utiliser un élément au début d'une liste existante. La liste `[1; 2; 3]` est en fait un raccourci pour l'expression `1 :: (2 :: (3 :: []))`. La liste `x :: tail` a pour *tête* l'élément `x`, et pour *queue* la liste `tail`.
Le filtrage sur des listes s'écrit ainsi :

```
match l with
| [] -> ...  (* code pour traiter le cas vide *)
| head :: tail -> ... (* code utilisant head et tail *)
```

3.2 Longueur d'une liste

Écrire une fonction `length : 'a list -> int` qui calcule récursivement la longueur d'une liste, c'est-à-dire son nombre d'éléments.

3.3 Recherche dans une liste

On peut chercher, dans une liste, un élément qui vérifie une propriété particulière. Il suffit de traverser la liste récursivement en testant pour chaque élément, s'il vérifie cette propriété. Cette fonction est partielle car il est possible qu'aucun élément de la liste ne satisfasse la propriété ; on utilise donc le type `'a option` pour pouvoir renvoyer `None` dans ce dernier cas.
(a) Écrire une fonction `chercher_multiple : int -> int list -> int` qui prend un entier positif `n` et une liste `l`, et renvoie `Some i` avec `i` le premier élément de la liste qui est un multiple de `n`, ou qui renvoie `None` si aucun élément n'est divisible par `n`. Quelques exemples de tests qui doivent valider :

```
chercher_multiple 2 [4;5;6] = Some 4;;
chercher_multiple 7 [4;5;6] = None;;
chercher_multiple 3 [1;2;4;5;6;9] = Some 6
chercher_multiple 4 [2;16;8;4]= Some 16;;
chercher_multiple 2 [] = None;;
```

(b) Écrire une fonction `chercher : ('a -> bool) -> 'a list -> 'a option` qui généralise la fonction précédente en retournant le premier élément de la liste qui satisfait le prédicat passé en argument, ou `None` dans le cas contraire.
(c) Ré-écrire la fonction `chercher_multiple` à l'aide de la fonction `chercher`.

3.4 Concaténation

(a) Écrire une fonction `append : 'a list -> 'a list -> 'a list` qui concatène la première liste au début de la deuxième. Tests :

```
append [1;2;3] [4;5] = [1;2;3;4;5];;
append [] [] = [];;
append [] [1;2] = [1;2];;
append [1;2] [] = [1;2];;
```

(b) Écrire une fonction `flatten : 'a list list -> 'a list` qui vérifie

```
flatten [[1]; [2]; [3]] = [1;2;3];;
flatten [[]; [1;2;3]; [4]; [5;6]] = [1;2;3;4;5;6];;
```

3.5 Liste d'association

On appelle *Liste d'association* le type `('a * 'b) list`. C'est une structure assez naïve permettant d'associer des valeurs (de type `'b`) à des clés (de type `'a`). Par exemple, la liste `[1, "one"; 2, "two"; 42, "answer"]` associe à certains entiers une chaîne de caractères.
Écrire une fonction `associer : ('a * 'b) list -> 'a -> 'b option` qui cherche la valeur associée à une clé donnée dans la liste, ou renvoie `None` si la clé est absente.
On testera ainsi :

```
# let l = [1, "one"; 2, "two"; 42, "answer"];;
l : (int * string) list = [1, "one"; 2, "two"; 42, "answer"]
# associer l 2;;
- : string option = Some "two"
# associer l 18;;
- : string option = None
# match assoc l 3 with
    | None -> "not found"
    | Some s -> "found " ^ s;;
- : string : "not found"
```

3.6 Ordre Lexicographique sur des listes

(a) En reprenant la définition précédente de l'ordre lexicographique, étendue aux listes en spécifiant que la liste vide est plus petite qu'une liste non vide,

écrire une fonction `compare_list : int list -> int list -> bool` qui renvoie `true` si la première liste est plus petite que la seconde dans l'ordre lexicographique.
(b) Généraliser la fonction précédente aux listes polymorphes en prenant comme premier paramètre la fonction de comparaison sur les éléments de la liste. La fonction aura donc pour type `('a -> 'a -> bool) -> 'a list -> 'a list -> bool`. On notera qu'il faut parcourir les 2 listes en parallèle, pour cela il est utile de rappeler qu'il est possible de filtrer plusieurs valeurs simultanément avec

```
match l1, l2 with
    | [], [] -> ...
    | ...
```