Programmation fonctionnelle, listes, variables libres et liées

Objectifs du TP
- Traitement sur les listes,
- stratégie d'évaluation de CAML,
- traitement sur les termes de Core ML.

1 Les listes

1.1 Le type liste

Les listes en CAML peuvent contenir un nombre quelconque d'éléments, y compris aucun, mais tous de même type[1]

Les listes peuvent être construites par énumération de leurs éléments séparés par des `;`, énumération entourée par des crochets.

```
# [3; 1; 4; 1; 5; 9; 2] ;;
- : int list = [3; 1; 4; 1; 5; 9; 2]
# ["timoleon"; "cunegonde"] ;;
- : string list = ["timoleon"; "cunegonde"]
# [true ; false] ;;
- : bool list = [true; false]
```

Comme on peut le voir sur ces exemples, les listes ont pour type τ `list` où τ est le type commun des éléments de la liste.

A priori la liste vide est de type polymorphe :

```
 # [] ;;
- : 'a list = []
```

mais comme on va le voir peu après ce n'est pas toujours le cas.

Le constructeur `::` permet d'ajouter en tête un élément à une liste donné.

```
# 1 :: [4;1;4] ;;
- : int list = [1; 4; 1; 4]
# true :: [false ; true] ;;
- : bool list = [true; false; true]
 # 1 :: 2 :: 3 :: 4 :: [] ;;
- : int list = [1; 2; 3; 4]
```

Ce constructeur permet d'effectuer du filtrage. Par exemple, il est possible d'extraire le premier élément d'une liste (non vide) donnée par un filtrage.

```
# let tete = function x :: _ -> x ;;
Warning P: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val tete : 'a list -> 'a = <fun>
# tete [3 ; 1 ; 4] ;;
- : int = 3
# tete [true ; false] ;;
- : bool = true
```

Question 1 Expliquez l'avertissement qui suit la déclaration de la fonction `tete`.

De la même façon, on peut extraire le reste d'une liste (non vide) par filtrage.

[1] C'est le cas de tous les langages de la famile ML, mais ce n'est pas le cas dans d'autres langages fonctionnels n'ayant pas d'inférence de type statique comme LISP, SCHEME, ...

```
# let reste = function _ :: l -> l ;;
Warning P: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val reste : 'a list -> 'a list = <fun>
# reste [3 ; 1 ; 4];;
- : int list = [1; 4]
# reste [true ; false] ;;
- : bool list = [false]
```

Et c'est maintenant qu'on peut remarquer que la liste vide n'est pas toujours d'un type polymorphe.

```
# reste [1] ;;
- : int list = []
```

1.2 Fonctions sur les listes

Vous allez récrire quelques fonctions pour la manipulation de listes, en utilisant exhaustivement le filtrage de motifs. Si possible, votre solution doit être récursive terminale.

Il est donc INTERDIT

1. d'utiliser la structure conditionelle **if then else**, sauf dans l'unique cas d'un test booleen (1 fois dans ce TP) ;
2. d'utiliser les fonctions prédéfinies du module `List`, comme `List.hd`, `List.tl`, `List.map`, `@`.

Consultez la doc du module `List` d'OCaml à l'adresse `http ://caml.inria.fr/pub/docs/manual-ocaml/libref/Lis`

Question 2 La fonction `length` qui détermine la longueur d'une liste.

```
val length : 'a list -> int
```

Question 3 La fonction `nth` qui donne le n-ième élément d'une liste.

```
val nth : 'a list -> int -> 'a
```

Assurez-vous que les indices soient utilisés comme dans `List.nth` : la tête de la liste a pour indice 0, etc ...

Votre fonction devra déclencher une exception pour des indices illégaux. Une exception non paramétrée se déclare à l'aide du mot clé `exception`.

```
exception Liste_trop_courte ;;
```

et se déclenche à l'aide de la fonction `raise`.

```
# raise Liste_trop_courte ;;
Exception: Liste_trop_courte.
```

Question 4 La fonction `rev` qui renvoie une liste d'ordre inverse.

```
val rev : 'a list -> 'a list
```

Question 5 La fonction `map` applique une fonction à chaque élément d'une liste, et renvoie la liste des valeurs obtenues.

```
val map : ('a -> 'b) -> 'a list -> 'b list
```

```
let rec map f = function
   | []   ->  []
   | a::l -> f a :: map f l ;;
```

Cette implantation n'est pas récursive terminale. Expliquez pourquoi !

Question 6 Essayez d'écrire une version plus efficace... après avoir attentivement lu la doc, notamment la dernière phrase.

Question 7 Écrivez une fonction récursive terminale `rev_map` (voir documentation). Vous la trouverez automatiquement en ajoutant un accumulateur et un emballeur à la déclaration précédente de `map`.

```
val rev_map : ('a -> 'b) -> 'a list -> 'b list

List.rev_map f l gives the same result as List.rev (List.map f l), but is tail-recursive and more efficient.
```

Question 8 Utilisez `rev_map` et votre `rev` pour définir la version la plus efficace possible de `map` (un appel à une fonction récursive terminale, suivi d'un appel à une autre fonction récursive terminale).

Question 9 Écrivez une fonction `filter` qui fait ce que fait `List.filter`. (C'est ici qu'il est permis d'utiliser **if then else**.)

```
val filter : ('a -> bool) -> 'a list -> 'a list

filter p l returns all the elements of the list l that satisfy the predicate p. The order of the elements in the in
```

Question 10 Écrivez la fonction `append`. Expliquez pourquoi `List.append` n'est pas récursive terminale. Déterminez la complexité de votre implantation.

```
val append : 'a list -> 'a list -> 'a list

Catenate two lists. Same function as the infix operator @. Not tail-recursive (length of the first argument). The @
```

Question 11 Une question sans programmation : décrivez la différence des résultats des fonctioné prédéfinies `flatten` en PROLOG, et `List.flatten` en OCAML. Testez des exemples pour déterminer laquelle est plus puissante.

```
val flatten : 'a list list -> 'a list

Same as concat. Not tail-recursive (length of the argument + length of the longest sub-list).
```

Question 12 OPTIONNELLE : Essayez d'écrire une fonction `flatten` en CAML qui fournit le résultat de sa cousine PROLOG. Expliquez.

Question 13 Utilisez la fonction `List.fold_left` pour calculer la somme des éléments d'une liste d'entiers.

```
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a

List.fold_left f a [b1; ...; bn] is f (... (f (f a b1) b2) ...) bn.
```

Les 3 arguments que vous passerez à `fold_left` seront :

1. une fonction de type `int->int->int`,
2. un entier,
3. et une liste d'entiers.

Et le résultat sera un entier.

Question 14 Programmez la fonction `fold_left`.

Question 15 Expliquez pourquoi `List.fold_right` n'est pas récursive terminale.

```
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b

List.fold_right f [a1; ...; an] b is f a1 (f a2 (... (f an b) ...)).
Not tail-recursive.
```

Question 16 Vous utilisez maintenant la fonction List.map2 f $[a_1;...;a_n][b_1;...;b_n]$, qui calcule$[fa_1b_1;...;fa_nb_n]$. Son type est

```
('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list
```

Dans d'autres langages de programmation, map2 s'appelle zip (c.a.d. *tirette*... devinez pourquoi !).

A faire : utilisez `List.map2` et votre `fold_left` pour calculer le produit scalaire de deux listes d'entiers de meme longueur. Par exemple pour `[3;4;5]` et `[2;2;2]` le résultat sera 24 ($= 3*2+4*2+5*2$), et pour `[1;1;1]` et `[2;2;2]` le résultat sera 6 ($= 1*2+1*2+1*2$).

Question 17 Écrivez une fonction `buildlist` qui, pour un entier positif n rend la liste des entiers de 0 à n.

```
val buildlist : int -> int list
```

Question 18 Écrivez une fonction `repeatlist` récursive terminale, du type polymorphe

```
val repeatlist : 'a -> int -> 'a list
```

qui construit une liste contentant n fois le premier élément passé lors de l'appel. Exemples :

```
# reapeatlist "a" 3;;
- : string list = ["a"; "a"; "a"]
#   reapeatlist 4 10;;
- : int list = [4; 4; 4; 4; 4; 4; 4; 4; 4; 4]
# let makeonelist = reapeatlist 1 ;;
  val makeonelist : int -> int list = <fun>
# makeonelist 13;;
 - : int list =
[1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1]
```

2 Stratégie d'évaluation de CAML

2.1 Évaluation du corps d'une fonction

Question 19 À votre avis, l'expression qui suit est-elle correcte ?

```
function x -> 1/0
```

Si oui quel est son type ? Que donnent les appels à cette fonction ? Et que peut-on en déduire sur l'évaluation du corps d'une fonction ?

Question 20 Si maintenant on considère la déclaration

```
let f x y = 1/x
```

que donne l'expression `f 0` ? et `f 0 1` ?

2.2 Ordre d'évaluation des arguments

Question 21 Dans un premier temps, sans utiliser l'interprète du langage, tentez de déterminer la valeur de l'expression CAML ci-dessou, ainsi que les effets de bord (impressions) que cette évaluation provoque.

```
let f = fun y z t-> ((function x -> y+z+t) (print_string "e"))
in
  f ((function x -> 1) (print_string "f"))
    ((function x -> 2) (print_string "l"))
    ((function x -> 3) (print_string "E"))
```

Puis vérifiez votre réponse.

Que peut-on en déduire sur la stratégie d'évaluation des expressions utilisée en CAML ?

Remarque : l'expression qui précède a été écrite dans un style purement fonctionnel. Elle aurait pu être écrite dans un style plus impératif en utilisant la séquence qui en CAMLest marquée par le simple point-virgule (`;`). Voici donc l'expression réécrite dans ce style.

```
let f = fun y z t-> ((function x -> y+z+t) (print_string "e"))
in
  f (print_string "f"; 1)
    (print_string "l"; 2)
    (print_string "E"; 3)
```

Question 22 Et si on remplace la définition de la varable locale par une séquence, obtient-on la même chose ?

```
let f = (print_string "e"; fun y z t-> y+z+t)
in
  f (print_string "f"; 1)
    (print_string "l"; 2)
    (print_string "E"; 3)
```

Explication.

3 Core ML

3.1 Variables libres et liées

Question 23 Réalisez une fonction nommée `fv` qui calcule la liste des variables libres d'une expression Core ML. Cette fonction doit avoir pour type

```
val fv : expression -> string list
```

Question 24 Même question pour les variables liées.

```
val bv : expression -> string list
```

3.2 Substitution

Question 25 Réalisez une fonction qui calcule le terme obtenu par substitution d'une variable par un terme dans un autre. Attention aux éventuelles captures de variables, pensez à la nécessité de renommer éventuellement les variables.

```
val subst : variable -> expression -> expression -> expression

subst x u t = [u/x] t
```