TP n°8 - Listes en OCaml

Notions abordées

- Type paramétré récursif prédéfini `'a list`
- Constructeurs de listes `[]` et `::`
- Notion de liste à traiter et liste résultat ou accumulateur
- Réduction, filtrage et transformation de listes
- Quelques fonctions du module List (`fold_left, rev, rev_append, map, filter_map`

Toutes les fonctions doivent être commentées et **testées**. Elles doivent notamment être munies d'une description faisant suite à d'éventuelles hypothèses sur leurs arguments.
En OCaml, les commentaires s'écrivent comme suit `(*Commentaires*)`.

Le type `'a pile` défini en classe est en fait déjà prédéfini en Ocaml, et s'appelle alors `'a list`. Afin de faciliter son usage, les constructeurs de ce type suivent une syntaxe particulière, mais comme pour notre type `'a pile`, le type prédéfini `'a list` est un type somme paramétré et récursif, qui admet deux constructeurs, l'un sans argument pour construire un objet vide, et l'autre qui construit un nouvel objet à partir d'un nouvel élément de type `'a` et d'un objet déjà existant.
Au lieu d'écrire `let p0:'a pile = PileVide` , on écrit `let l0:'a list =[]`, et au lieu d'écrire `let p1 = PileNonVide (elem,pile)` , on écrit `let l1 = elem :: liste`.

Ces nouveaux constructeurs induisent aussi un *nouveau* schéma de filtrage, au lieu d'écrire

```
match p with
| PileVide -> ...
| PileNonVide (elem, pile) -> ...
```

on écrit

```
match l with
| [] -> ...
| elem::liste -> ...
```

Il est aussi possible de construire directement une liste en séparant ses éléments par des points-virgules et en encadrant le tout par des crochets. Par exemple `let l1 : int list = [1; 2 ; 48; -3]`.
Attention à ne pas utiliser des virgules sinon OCaml comprend qu'on donne une liste à un seul élément qui est un n-uplet dont les composantes sont les éléments qu'on espérait mettre dans la liste.

Enfin, il est aussi possible de construire une liste par concaténation de listes de même type grâce à l'opérateur `@`, par exemple `[4; 3] @ [1; 2 ; 48; -3]` est la liste `[4; 3; 1; 2 ; 48; -3]`. Dans le cas d'un ajout d'un élément `elem` en tête d'une liste `l`, on préférera `elem::l` à `[elem]@l`. Notez bien que `l::elem` n'est pas syntaxiquement correct si `l` est une liste (de type `'a list`) et `elem` un élément (de type `'a`).

Exercice 1 Calculer une valeur par réduction d'une liste

Question 1 💻
Donner une fonction récursive terminale `somme_entiers` qui calcule la somme d'une liste d'entiers.

Question 2 💻
Donner une fonction récursive terminale `somme_float` qui calcule la somme d'une liste de flottants.

Question 3 💻
Donner une fonction récursive terminale `conjonction` qui calcule la conjonction des booléens d'une liste. Justifier le choix de la valeur retournée pour une liste vide. *Cette fonction peut être récursive terminale sans définir une fonction auxiliaire.*

Question 4 💻
Donner une fonction récursive terminale `pgcd_entiers` qui calcule le PGCD des entiers naturels d'une liste. Par exemple le PGCD de `[12; 36; 10]` est `2`. Justifier le choix de la valeur retournée pour une liste vide.

Question 5 💻
Donner une fonction récursive terminale `compose_fonctions` qui calcule la composée des fonctions d'une liste. Justifier le choix de la valeur retournée pour une liste vide. *Avant de coder, réfléchissez au type possible pour les éléments de la liste afin que la composition ait un sens. Veillez ensuite à être assez précis dans la description pour qu'on comprenne dans quel ordre les fonctions sont composées.*

Question 6 ✍
Peut-on extraire un schéma commun des définitions précédentes. Quels paramètres suffit-il de fixer dans ce schéma pour définir une fonction ? Quelles sont alors les hypothèses sur ces paramètres.

Ce schéma étant assez courant dans les manipulations de listes, il existe en OCaml une fonction qui prend en argument une fonction f, un élément de départ a et une liste L, dont on note $L_1, L_2, \dots L_n$ les éléments, et qui calcule $f(\dots f(f(a, L_1), L_2), \dots L_n)$. Plus précisément, la fonction `List.fold_left` est de type `('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`.

Question 7 ✍
Pour chacune des fonctions précédentes, identifier quelle fonction de deux arguments est itérée (*i.e.* le f) et quel élément est utilisé initialement (*i.e.* le a).

Question 8 💻
Donner un nouvelle définition de la fonction `pgcd_entiers` nommée `pgcd_entiers_bis`, se réduisant à un appel à la fonction `List.fold_left`. Même question pour `compose_fonctions`.

Exercice 2 Calculer une liste par filtrage d'une liste

Question 1 💻
Donner une fonction récursive terminale `entiers_pairs` qui prend en argument une liste d'entiers `l` et qui calcule la liste des entiers pairs contenus dans `l` (avec ordre et multiplicité). Par exemple `entiers_pairs [1; 2; 3; 4; 2]` vaut `[2; 4; 2]`, tandis que `entiers_pairs [3; 5; 3; 1]` vaut `[]`. *Pour faire ça de manière récursive terminale, il peut être utile d'envisager de maintenir deux listes, l'une des éléments restant à traiter, l'autre des éléments déjà traités, ou plutôt déjà sélectionnés.*

Question 2 💻
Donner une fonction récursive terminale `renverse` qui prend en argument deux listes `l1` et `l2` dont les éléments sont de même type, et qui calcule la liste obtenue en déplaçant successivement l'élément en tête de `l1` en tête de `l2`. Par exemple `renverse [1; 2; 3] [10; 11; 12]` donne `[3; 2; 1; 10; 11; 12]`

Question 3 💻
Donner une fonction récursive terminale `miroir` qui prend en argument une liste `l1` et qui calcule la liste obtenue en inversant l'ordre des ses éléments. *Cette fonction doit appeler* `renverse`.

Question 4 💻
Si ce n'est déjà fait, simplifier la définition de `entiers_pairs` en utilisant les fonctions `renverse` et `miroir`.

Question 5 💻
Donner une fonction récursive terminale `entiers_positifs` qui prend en argument une liste d'entiers `l` et qui calcule la liste des entiers positifs contenus dans `l` (avec ordre et multiplicité). Par exemple `entiers_positifs [-1; 2; -3;- 4; 2; 13]` vaut `entiers_pairs [2; 2]; 13`, tandis que `entiers_positifs [-3; -12; 0]` vaut `[0]`.

Question 6 💻
Donner une fonction récursive terminale `caracteres_alpha` qui prend en argument une liste de caractères `l` et qui calcule la liste des caractères qui sont des lettres, minuscules ou majuscules, contenus dans `l` (avec ordre et multiplicité). Autrement dit la fonction `caracteres_alpha` calcule la liste obtenue en éliminant les caractères spéciaux de la liste qu'elle prend en argument.

Question 7 ✍
Peut-on extraire un schéma commun des définitions de `caracteres_alpha` et `entiers_pairs`. Quels paramètres suffit-il de fixer dans ce schéma pour définir une fonction ? Quelles sont alors les hypothèses sur ces paramètres.

Ce schéma étant assez courant dans les manipulations de listes, il existe en OCaml une manière de le reproduire très simplement. On explique comment à la fin du prochain exercice.

Exercice 3 Calculer une liste par transformation d'une liste

Dans cet exercice, on aura probablement à nouveau besoin des fonctions `miroir` et `renverse`, cependant on ne les recodera pas dans le fichier `exo3.ml` car ces fonctions sont déjà définies en Ocaml dans le module `List`. On utilisera donc `List.rev` de signature `'a list -> 'a list` au lieu de `miroir` et `List.rev_append` de signature `'a list -> 'a list -> 'a list` au lieu de `renverse`.

Question 1 💻
Définir une fonction `oppose` qui prend en argument une liste d'entiers, et qui calcule la liste de leurs opposés.

Question 2 💻
Définir une fonction `intervertit` qui prend en argument une liste de couples `l`, et qui calcule la liste obtenue en intervertissant la première et la deuxième composant de chaque couple de `l`. *Votre fonction doit être indépendante des types des éléments es couples.*

Question 3 💻
Définir une fonction `démultiplie` qui prend en argument une liste de couples composés d'un entier et d'un élément, et qui calcule la liste où chaque élément est répété autant de fois qu'indiqué par la première composante du couple. Par exemple `démultiplie [('a', 2); ('b', 3) ; ('a',1)]` vaut `['a'; 'a'; 'b'; 'b'; 'b'; 'a']`. *Votre fonction doit être indépendante du type des éléments.*

Question 4 💻
Définir une fonction `compresse` réciproque de la fonction `démultiplie` (pour les listes sans couple de la forme `(elem,0)`). *Votre fonction doit être indépendante du type des éléments.*

Question 5 💻
Définir une fonction `moyennes_listes` qui prend en argument une liste de liste d'entiers non vides, et qui calcule la liste de leurs moyennes.

Question 6 💻
Quel type vu en cours permettrait de se débarrasser de l'hypothèse de non vacuité des éléments de la liste prise en entrée. Définir alors une nouvelle version de la fonction `moyennes_listes` nommée `moyennes_listes_bis`.

Il existe en OCaml une fonction qui prend en argument une fonction f et une liste L, dont on note $L_1, L_2, \dots L_n$ les éléments, et qui calcule la liste $[f(L_1), f(L_2), \dots, f(L_n)]$. Plus précisément, la fonction `List.map` est de type `('a -> 'b ) -> 'a list -> 'b list`.

Question 7 ✍ puis 💻
Parmi les fonctions précédentes, repérer lesquelles peuvent être réalisées par un simple appel à `List.map`. Pour celles-ci, proposer une nouvelle définition simplifiée. Pour les autres, identifier ce qui fait défaut, et imaginer un nouveau schéma.

Dans le cas où la fonction `f` (que l'on veut appliquer à chacun des éléments d'une liste `l` de type `'a list`) est **partiellement définie** sur le type `'a`, il est possible d'obtenir la liste des images par `f` des éléments de `l` pour lesquels `f` est définie seulement.
Pour cela il faut un peu transformer `f` de type `'a -> 'b` en une fonction `f_bis` de type `'a -> 'b option` qui vaut `None` sur les entrées pour lesquelles `f` n'était pas définie. On peut alors utiliser la fonction `List.filter_map` qui est de type `('a -> 'b option) -> 'a list -> 'b list`.

Question 8 💻

Si ce n'est déjà fait, définir une fonction `moyennes_bis` de type `int list-> float option`. Pour une liste d'entiers non vide `l`, `moyennes_bis l` vaut `Some m` où `m` est la moyenne de `l`, et `moyennes_bis []` vaut `None`. On peut alors définir une nouvelle version de la fonction `moyennes_listes` nommée `moyennes_listes_ter`, de type `int list list -> float list option`, qui se réduit à un appel à `List.filter_map`.

Question 9 ✍ puis 💻

On suppose qu'on veut simplement filtrer une liste, c'est-à-dire calculer la liste obtenue en ne conservant que les éléments vérifiant une certaine propriété qu'on sait évaluer, c'est-à-dire les éléments dont l'image par `f` est `true`. Proposer une méthode générique faisant appel à `List.filter_map`.

Exercice 4 Autres manipulations sur les listes

Question 1 💻

Définir une fonction récursive terminale qui teste si une liste d'entiers est croissante.

Question 2 💻

Définir une fonction récursive terminale qui teste si une liste de caractères est croissante pour l'ordre alphabétique.

Question 3 ✍

Que remarquez-vous sur les deux fonctions précédentes ? Quelle signature et quelles hypothèses proposer alors pour une fonction `est_croissante` générique ?

Question 4 💻

Définir une fonction récursive terminale qui teste si une liste de flottants est uniquement constituée uniquement de valeurs de l'intervalle $]0, 1]$. *Vous pouvez utiliser l'un des schémas introduits ci-avant.*

Question 5 💻

Définir une fonction `bin_packing_glouton` qui prend en argument une liste `l` de valeurs de l'intervalle $]0, 1]$, et la subdivise en sous-listes de valeurs consécutives de somme inférieure ou égale à 1. Plus précisément, l'ordre et la multiplicité doivent être respectés de sorte que la concaténation des sous-listes rend exactement `l`. De plus, chaque sous-liste est la plus grande que l'on peut obtenir sans dépasser 1. Par exemple `bin_packing_glouton [0.1; 0.8; 0.3; 0.7; 0.1; 0.1; 0.5]` donne `[[0.1; 0.8];[0.3; 0.7]; [0.1; 0.1; 0.5]]`.