TP2 : Listes, Typage Persistant

Semaine du 18 septembre

Dans le cas où une entrée n'est pas valide, vous pouvez utiliser la fonction `failwith` qui est de signature `str -> 'a` et qui permet d'interrompre le calcul avec un message passé en argument. (Cette fonction est paramétré pour sa sortie afin de ne jamais poser de problème de typage.)

Exercice 1. Calcul sur les listes

1. Proposer une fonction de signature `'a list -> bool` qui renvoie si la liste en entrée est vide.
2. Proposer une fonction de signature `int list -> int` qui renvoie la somme des éléments d'une liste.
3. Proposer une fonction de signature `'a list -> bool` qui renvoie `true` si et seulement si la liste en entrée a un nombre pair d'éléments.
4. Proposer une fonction de signature `float list -> float` qui renvoie le produit des éléments d'une liste.
5. Proposer une fonction de signature `int list -> int` qui renvoie le maximum des éléments d'une liste.
   On pourra renvoyer `min_int` si la liste est vide.
6. Proposer une fonction de signature `int list -> int` qui renvoie le minimum des éléments d'une liste.
7. Proposer une fonction `dernier_element` de signature `'a list -> 'a` qui renvoie le dernier élément de la liste en entrée.

Exercice 2. Construction de Liste

1. Proposer une fonction de signature `int -> float list` qui avec une entrée n renvoie une liste de taille n qui ne contient que zéros.
2. Proposer une fonction de signature `int -> 'a -> 'a list` qui sur une entrée n et x renvoie une liste de taille n et dont tous les éléments sont égaux à x.
3. Proposer une fonction de signature `int -> int list` qui avec une entrée n renvoie une liste des éléments des entiers de 0 à n par ordre décroissant.
4. Proposer une fonction de signature `int -> int list` qui avec une entrée n renvoie une liste des éléments des entiers de 0 à n par ordre croissant.
5. Proposer une fonction de signature `int ->int list` qui pour une entrée n renvoie la liste des diviseurs de n.
6. Proposer une fonction de signature `int -> int list` qui pour une entrée n renvoie la liste des entiers premiers avec n inférieurs à n.

Exercice 3. Des listes à partir de listes

1. Proposer une fonction de signature `int list -> int liste` qui renvoie une liste de tous les éléments non nuls de la liste en entrée.
2. Proposer une fonction `concat` de signature `'a list -> 'a list -> 'a list` qui renvoie la concaténation des éléments des deux listes en entrée.

```
concat [1;2] [3;4;5] (* [1;2;3;4;5] *)
```

OCaml propose l'opérateur @ qui fait la même chose.

3. Proposer une fonction `inverse` de signature `'a list -> 'a list` de sorte à ce que `inverse l` renvoie la liste `l` inversée.

```
inverse [1;2;3] (* [3;2;1] *)
```

4. Proposer une fonction `inserer` de signature `'a list -> int -> int -> 'a list` de sorte à ce que `inserer l n x` renvoie la liste `l` mais avec l'élément `x` inséré à la position `n`.

```
inserer [1;2;3;4] 2 0 (* [1;2;0;3;4] *)
```

Dans le cas où la liste est trop petite, on pourra lever une exception, ou ajouter l'élément à la fin de la liste.

5. Proposer une fonction `zip` de signature `'a list -> b' list -> ('a * 'b) list` de sorte à ce que, pour deux listes de même longueurs `[a1; ...; an]` et `[b1 ;...; bn]`, `zip [a1; ...; an] [b1; ...; bn]` renvoie la liste de couple `[(a1,b1) ;... ; (an; bn)]`.
6. Proposer une fonction `unzip` de signature `('a * 'b) list -> 'a list * b' list` qui sur une entrée `[(a1,b1) ;... ; (an; bn)]` renvoie le couple de listes `([a1; ...; an], [b1; ...; bn])`.
7. Proposer une fonction `mirroir` de type `('a*'b) list -> ('b*'a) list` qui inverse l'ordre de tous les couples de la liste en argument :

```
mirroir [(1,2);(3,4)] (* [(2,1);(4,3)] *)
```

8. Proposer une fonction `pairs_avec` de type `'a -> 'b list -> ('a*'b) list` qui réalise une liste des couples dont le premier élément est le premier argument, et dont les deuxièmes éléments sont les éléments de la liste en argument.

```
pairs_avec 1 [2;3;4] (*[(1,2); (1, 3); (1,4)]*)
```

9. Proposer une fonction `pairs` de signature `'a list -> ('a * 'a) list` qui renvoie la liste de toutes les pairs d'éléments de la liste en entrée.

```
pairs [1;2;3] (* [(1,2); (1,3); (2,3); (2, 1); (3, 1); (3,2)]*)
```

L'ordre n'est pas important. On pourra utiliser l'exercice 9.

10. Proposer une fonction `ajouter_a_toutes` de signature `'a -> 'a list list -> 'a list list` qui, à partir d'un élément et d'une liste de listes, rajoute l'élément dans toutes les listes de la liste.

```
ajouter_a_toutes 1 [[];[4];[2;3]] (* [[1];[1;4];[1;2;3]]*)
```

11. Proposer une fonction `produit_cartesien` de signature `'a list -> 'b list -> ('a * 'b) list` qui renvoie la liste des couples dont le premier élément est dans la liste en premier argument, et dont le second élément est dans la liste en second argument.

```
produit_cartesien [1;2] [3;4;5] (*[(1,3); (1,4); (1,5); (2,3); (2,4); (2,5)
```

12. Proposer une fonction `prefixes` de signature `'a list -> 'a list list` qui renvoie la liste de tous les préfixes de la liste en entrée.

```
prefixes [1;2;3] (* [[];[1];[1;2];[1;2;3]]*)
```

13. Proposer une fonction sous_ensembles de signature 'a list -> 'a list list qui renvoie la liste de toutes les sous-liste incluses dans l'entrée, en gardant l'ordre des éléments de l'entrée.

```
sous_ensembles [1;2;3] (*[[]; [1]; [2]; [3]; [1; 2]; [2; 3]; [1; 3]; [1; 2; 3]]*)
```

Exercice 4. Ordre Supérieur

1. Proposer une fonction verifier_tous de signature ('a -> bool) -> 'a list -> bool qui vérifie si tous les éléments de la liste en entrée valent true par la fonction passée en entrée.

```
verifier_tous (fun x -> x>0) [1;2;-2] (*false*)
```

```
verifier_tous (fun x -> x>0) [1;2;2] (*true*)
```

```
verifier_tous (fun x -> x>0) [] (*true*)
```

2. Proposer une fonction verifier_un de signature ('a -> bool) -> 'a list -> bool qui vérifie si au moins un élément de la liste en entrée vaut true par la fonction passée en entrée.

```
verifier_un (fun x -> x >0) [-2; 0; -1] (*false*)
```

```
verifier_un (fun x -> x >0) [-2; 1; -1] (*true*)
```

```
verifier_un (fun x -> x >0) [] (*false*)
```

3. Proposer une fonction filtrer de signature ('a -> bool) -> 'a list -> 'a list qui prend en entrée une fonction et une liste et qui renvoie la liste des éléments de la liste en entrée dont la valeur par la fonction est true.

```
filtrer (fun x -> x > 0) [-2; 1; 2] (*[1;2]*)
```

4. Proposer une fonction map de sorte à ce que map f [x1; x2; ....; xn] calcule [f x1; f x2; ....; f xn].

```
map (fun x -> x*2) [1; 2; 3] (*[2; 4; 6]*)
```

5. Proposer une fonction tous_les_entiers de signature (int -> bool) -> int -> int list de sorte à ce que tous_les_entiers f n renvoie la liste de tous les entiers k inférieurs à n tels que f k soit égal à true.

Exercice 5. Compression

OCaml propose un module List qui comprend beaucoup des fonctions déjà présentées dans ce TD. L'une en particulier, List.fold_left permet de compresser une liste en applicant successivent une fonction passée en argument.

La fonction List.fold_left f init [a1; a2 ;... ;an] nous donne f (... (f (f init b1) b2) ...) bn. Ainsi, pour calculer la somme, d'une liste, on peut faire List.fold_left (fun x y -> x + y) 0 liste.

1. Quelle est la signature de List.fold_left ? Justifier.
2. À l'aide de List.fold_left, proposer une fonction de signature float list -> float qui calcule le produit des éléments d'une liste de flottants.
3. À l'aide de List.fold_left, proposer une foction de signature bool list -> bool qui renvoie true si et seulement si tous les éléments de la liste en entrée sont égaux à true.

4. À l'aide de `List.fold_left`, proposer une fonction qui inverse la liste en entrée.
5. Proposer une implémentation de la fonction `fold_left`.

Exercice 6. Des listes triées

1. Proposer une fonction `est_triee` de signature `'a list -> bool` qui renvoie si la liste en entrée est triée.

```
est_triee [1 ; 5; 23] (* true *)
```

```
est_triee [2 ; 8; 7; 11] (* false*)
```

On cherche désormais à pouvoir préciser pour quelle comparaison la liste est triée : on veut pouvoir spécifier une fonction de comparaison de notre choix.

2. Proposer une fonction `est_triee2` de type `('a -> 'a -> bool) -> 'a list -> bool` telle que `est_triee2 f l` vérifie si, pour toute paire consécutive d'élément `x` et `y` dans `l`, on a bien que `f x y` vaut `true`.

```
let comp1 x y = x<y in
est_triee2 comp1 [1; 5; 23] (* true *)
```

```
let comp2 x y = x>y in
est_triee2 comp2 [1; 5; 23] (* false *)
```

Au lieu de préciser une fonction de comparaison, on peut préciser une clé qui permette de calculer, à partir de chaque élément, la valeur qui va nous servir pour l'ordonner.

3. Proposer une fonction `est_triee3` de type `('a -> 'b) -> 'a list -> bool` telle que `est_triee3 clef l` vérifie si, pour toute paire consécutive d'éléments `x` et `y` dans `l`, on a bien que `clef x <= clef y`.

Exercice 7. Tri Fusion

On cherche à trier les éléments d'une liste. Pour ce faire, on propose d'utiliser un type de tri diviser-pour-régner. L'idée est de procéder par en trois étapes :

— On divise la liste en deux listes de taille égale à un élément près ;
— On trie récursivement ces deux listes ;
— On fusionne ces listes en gardant l'ordre.

1. Proposer une fonction `diviser` de signature `'a list -> 'a list * 'a list` qui sépare la liste en entrée en deux listes dont les tailles diffèrent d'au plus un, et renvoie le couple formé des deux listes.

```
diviser [3;5;2;4;1] (* ([3;5;2],[4;1])*)
```

2. Proposer une fonction `fusionner` de signature `'a list -> 'a list -> 'a list` qui à partir de deux listes triées par ordre croissant en entrée retourne la liste triée qui contient les éléments des deux listes.

On pourra parcourir les deux listes en même temps et à chaque fois ne prendre que l'élément le plus petit des têtes des deux listes.

3. Proposer une fonction `tri_fusion` de signature `'a list -> 'a list` qui renvoie la liste triée contenant les mêmes éléments que dans la liste en entrée.

Pour l'instant, nous avons utilisé les comparaisons usuelles pour comparer les éléments deux à deux. On chercher à rajouter un argument aux fonctions `fusionner` et `tri_fusion` de sorte à pouvoir préciser quelle comparaison nous utilisons.

4. Proposer une fonction fusionner2 de signature ('a -> 'a -> bool) -> 'a list -> 'a list -> 'a list qui prend en argument une fonction de comparaison (fonction qui détermine si le premier élément est plus petit que le second élément) et deux listes triées au sens de cette fonction de comparaison, et qui renvoie une liste triées avec les deux éléments des deux listes.

```
let comp1 x y = x < y in
fusionner comp1 [1;2;4] [3;5] (* [1;2;3;4;5]*)
```

```
let comp2 x y = x > y in
fusionner comp2 [4;2;1] [5;3] (* [5;4;3;2;1] *)
```

5. Proposer une fonction tri_fusion2 de signature ('a -> 'a -> bool) -> 'a list -> 'a list qui prend en argument une fonction de comparaison et une liste, et qui retourne la liste des éléments de sorte à ce que les éléments soit triés au sens
6. En déduire une fonction de type tri_listes_par_somme de signature int list list -> int list list qui renvoie la liste des listes en entrée triées par somme croissante.

```
tri_listes_par_somme [[1;43]; [2]; [5;7;2]] (* [[2]; [5;7;2]; [1;43]]*)
```

7. Au lieu de préciser la fonction de comparaison, parfois, on précise une clef de signature ('a -> int) qui permet d'associer à chaque élément de la liste une valeur dont on se sers pour ordonner les éléments.

   Proposer une fonction tri_fusion3 de sorte à ce que tri_fusion clef liste trie la liste liste de sorte à ce que les éléments dans la liste de sortie soit croissants pour leur valeur par clef.

Exercice 8. Relation d'Ordre

Une relation d'ordre sur un ensemble E est une relation R qui vérifie les trois propriétés suivantes pour tout x, y et z dans E :
- Réflexivité : xRx ;
- Antisymmétrie : si xRy et yRx alors $x = y$;
- Transitivé : si xRy et yRz alors xRz.

On peut modéliser une relation R par une fonction r de signature 'a -> 'a -> bool de sorte à ce que r x y ait pour valeur true si et seulement si xRy.

1. Proposer une fonction est_reflexive_sur de signature ('a -> 'a -> bool) -> 'a list -> bool qui vérifie si la relation définie par le premier argument est réflexive sur les éléments de la liste en second argument.
2. Proposer une fonction est_antysimmetrique_sur de signature ('a -> 'a -> bool) -> 'a list -> bool qui vérifie si la relation définie par le premier argument est antisymmétrique sur les éléments de la liste en second argument.
3. Proposer une fonction est_transitive_sur de signature ('a -> 'a -> bool) -> 'a list -> bool qui vérifie si la relation définie par le premier argument est transitive sur les éléments de la liste en second argument.
4. En déduire une fonction est_une_relation_d_ordre_sur de signature ('a -> 'a -> bool) -> 'a list -> bool qui détermine si la relation dans le premier argument est une relation d'ordre sur l'ensemble des éléments du second argument.