Feuille de révisions n°1 - Programmation en OCaml : manipulation de listes

1 Pied à l'étrier

R. 1-1 Écrire une fonction récursive retournant le dernier élément d'une liste d'un type quelconque. Cette fonction lèvera une exception `Invalid_argument` dans le cas où la liste est vide. On rappelle que l'exception `Invalid_argument` est prédéfinie en OCaml, elle doit être accompagnée d'une chaîne de caractères. Par exemple : l'évaluation de l'expression `raise (Invalid_argument("ceci est un message d'erreur"))` lève l'exception `Invalid_argument("ceci est un message d'erreur")`.

R. 1-2 Écrire une fonction récursive retournant le dernier élément d'une liste d'un type quelconque. Cette fonction doit retourner `None` si la liste est vide et `Some(x)` si la liste contient au moins un élément et que le dernier élément est `x`.

R. 1-3 Écrire une fonction permettant de calculer le miroir d'une liste. Une attention toute particulière sera accordée au fait de ne pas fournir une implémentation en $\mathcal{O}(n^2)$ due à une malencontreuse utilisation de `@`. On n'utilisera pas de fonctionnelle d'itération.

R. 1-4 Écrire une fonction permettant de calculer le miroir d'une liste. Une attention toute particulière sera accordée au fait de ne pas fournir une implémentation en $\mathcal{O}(n^2)$ due à une malencontreuse utilisation de `@`. On s'efforcera d'utiliser la fonctionnelle `List.fold_left`.

R. 1-5 Écrire une fonction permettant de calculer la concaténation de deux listes. On n'utilisera pas de fonctionnelle d'itération.

R. 1-6 Écrire une fonction permettant de calculer la concaténation de deux listes. On s'efforcera d'utiliser la fonctionnelle `List.fold_left`.

R. 1-7 Écrire une fonction prenant en argument une liste `l` et un élément `x` et calculant la liste `l` privée de toutes les occurrences de l'élément `x`.

R. 1-8 Écrire une fonction prenant en argument une liste `l` et un indice `i` et retournant deux listes : la sous-liste des éléments de `l` d'indice inférieurs stricts à `i` et la sous-liste des éléments de `l` d'indices supérieurs à `i`. Cette fonction devra être récursive et ne pas faire appel à une fonction récursive auxiliaire.

R. 1-9 Écrire une fonction prenant en argument une liste `l` et un indice `i` et retournant deux listes : la sous-liste des éléments de `l` d'indice inférieurs stricts à `i` et la sous-liste des éléments de `l` d'indices supérieurs à `i`. Cette fonction devra être récursive et ne pas faire appel à des fonctions récursives auxiliaires.

R. 1-10 Écrire une fonction prenant en argument une liste `l` et la découpant en deux listes, dans la première on rangera les éléments d'indices pairs dans `l`, dans la seconde on rangera les éléments d'indices impairs dans `l`. L'ordre des éléments dans les listes résultats devra être celui de la liste d'entrée.

R. 1-11 Écrire une fonction testant si une liste est un palindrome. Cette fonction devra être en $\mathcal{O}(n)$ où n est la longueur de la liste.

2 Listes de listes

R. 1-12 Écrire une fonction prenant en argument une liste de listes `l` et retournant la liste des éléments se trouvant dans une des listes de `l`. Les éléments devront être rangés dans le même ordre que dans la liste initiale. Par exemple sur la liste `[[1;2]; [3;4;5]; []; [6]]` on obtiendra `[1; 2; 3; 4; 5; 6]`. On utilisera la fonction de concaténation de listes `@`, en prenant garde à ne pas obtenir une complexité quadratique.

R. 1-13 Écrire une fonction prenant en argument une liste de listes `l` et retournant la liste des éléments se trouvant dans une des listes de `l`. Les éléments devront être rangés dans le même ordre que dans la liste initiale. Par exemple sur la liste `[[1;2]; [3;4;5]; []; [6]]` on obtiendra `[1; 2; 3; 4; 5; 6]`. On n'utilisera pas, et on ne redéfinira pas non plus, l'opération de concaténation de listes. On utilisera deux fonctions mutuellement récursives.

R. 1-14 Écrire une fonction prenant en argument une liste de listes `l` et retournant la liste des éléments se trouvant dans une des listes de `l`. Les éléments devront être rangés dans le même ordre que dans la liste initiale. Par exemple sur la liste `[[1;2]; [3;4;5]; []; [6]]` on obtiendra `[1; 2; 3; 4; 5; 6]`. On n'utilisera pas, et on ne redéfinira pas non plus, l'opération de concaténation de listes. On utilisera une unique fonction auxiliaire récursive terminale.

R. 1-15 Écrire une fonction prenant en argument une liste de listes `l` et retournant la liste des éléments se trouvant dans une des listes de `l`. Les éléments devront être rangés dans le même ordre que dans la liste initiale. Par exemple sur la liste `[[1;2]; [3;4;5]; []; [6]]` on obtiendra `[1; 2; 3; 4; 5; 6]`. On n'utilisera pas, et on ne redéfinira pas non plus, l'opération de concaténation de listes, on se limitera à des itérations au moyen de la fonctionnelle `List.fold_left`.

3 Listes et comparaisons

R. 1-16 Écrire une fonction récursive prenant en argument une liste et retournant un couple dont la première composante est le minimum de la liste et la seconde composante est le maximum. Cette fonction doit être récursive et ne pas utiliser de fonctions récursives auxiliaires. Dans le cas où la liste est vide on lèvera l'exception `Invalid_argument`.

R. 1-17 Écrire une fonction récursive prenant en argument une liste et retournant un couple dont la première composante est le minimum de la liste et la seconde composante est le maximum. Cette fonction doit être récursive terminale et peut utiliser une fonction récursive auxiliaire. Dans le cas où la liste est vide on lèvera l'exception `Invalid_argument`.

R. 1-18 Écrire une fonction récursive prenant en argument une liste et retournant un couple dont la première composante est le minimum de la liste et la seconde composante est le maximum. Cette fonction doit utiliser la fonctionnelle `List.fold_left` comme mécanisme d'itération. Dans le cas où la liste est vide on lèvera l'exception `Invalid_argument`.

R. 1-19 Écrire une fonction permettant de découper une liste en sous-séquences (non vides) croissantes maximales pour l'extension à gauche. Par exemple la liste `[1; 4; 5; 8; 7; 5; 2; 3; 4; 9; 3; 3; 3; 5; 5; 7; 2]` est découpée en `[[1; 4; 5; 8]; [7]; [5]; [2; 3; 4; 9]; [3; 3; 3; 5; 5; 7]; [2]]`.

R. 1-20 Écrire une fonction permettant de découper une liste en sous-séquences (non vides) monotones maximales pour l'extension à gauche. Par exemple la liste `[1; 4; 5; 8; 7; 2; 3; 4; 9; 3; 3; 3; 5; 5; 7; 2]` est découpée en `[[1; 4; 5; 8]; [7; 5; 2]; [3; 4; 9]; [3; 3; 3; 5; 5; 7]; [2]]`

R. 1-21 Écrire une fonction prenant en arguments une liste `(l: 'a list)`, un entier `k`, une fonction `f: 'a -> int` à valeurs dans $[\![0, k]\!]$ et retournant la liste des éléments de `l` triés par image par `f` croissante. On demande une implémentation en $\mathcal{O}(\max(n, k))$ où n est la taille de la liste `l`.

4 Quelques utilisations classiques des listes en algorithmique

R. 1-22 Écrire une fonction permettant de trier une liste au moyen de l'algorithme du tri fusion.

R. 1-23 Fournir une implémentation du type de données abstrait file au moyen de deux listes. On assurera une complexité amortie en $\mathcal{O}(1)$ pour chaque opération.

R. 1-24 Écrire une fonction prenant en argument une liste et retournant la liste des permutations de cette liste.

R. 1-25 Écrire une fonction prenant en argument un entier naturel n et retournant la liste des 2^n listes contenant n booléens. Ainsi dans la liste résultat on devra pouvoir trouver chaque liste de n booléens.