Université de Lille

2026–2027
Licence Informatique

Programmation fonctionnelle

PF : TD 3

Dans cette feuille de TD, pour chaque fonction que vous implémenterez, vous en indiquerez le type le plus général.

Échauffement minimum

Exercice 1 : Minimum d'une liste

Nous souhaitons implémenter une fonction `monMinimum` qui renvoie le minimum des éléments d'une liste, mais qui se termine sans erreur lorsque la liste est vide.

1. Quel type donner à cette fonction ?
2. Proposez une implémentation récursive basée sur le filtrage de motif.

Arbres binaires de recherche génériques

Exercice 2 : Arbres binaires de recherche

On souhaite implémenter un type `ABR a` représentant un arbre binaire de recherche polymorphe portant une valeur d'un type générique `a`.

Nous allons considérer que le type `a` est muni d'un ordre total et que toute donnée de type `ABR a` vérifie l'invariant des arbres binaires de recherche, à savoir que pour tout nœud portant la valeur `a`, un sous-arbre gauche `t1` et un sous-arbre droit `t2` :

— les éléments contenus dans `t1` sont tous *strictement inférieurs* à `a`,
— les éléments contenus dans `t2` sont tous *strictement supérieurs* à `a`.

Q 2.1 Type `ABR a`

Donnez la définition du type `ABR a`.

Q 2.2 Insertion d'un élément

On souhaite écrire une fonction `inserer` telle que `inserer e a` insère l'élément `e` dans l'arbre binaire de recherche `a` et renvoie le nouvel arbre obtenu.

1. Déterminez le type de `inserer`.
2. Donnez la définition de `inserer`.

Q 2.3 Arbre vérifiant l'invariant

Nous allons maintenant implémenter un prédicat `estABR` qui vérifie si un arbre de type `ABR a` vérifie l'invariant des arbres binaires de recherche.

Pour cela, nous allons procéder en trois étapes :

— implémenter une fonction `abrToList` telle que `abrToList a` renvoie la liste contenant les éléments de l'arbre `a` ; les éléments de la liste résultat doivent être triés en ordre croissant quand `a` vérifie l'invariant des arbres binaires de recherche ;
— Implémenter ensuite un prédicat `estOrdonnee` tel que `estOrdonnee l` renvoie vrai si la liste `l` est ordonnée par ordre croissant, ou faux sinon ;
— En utilisant `abrToList` et `estOrdonnee`, implémenter le prédicat `estABR`.

Donnez la définition de `abrToList`, `estOrdonnee` et `estABR`.

Exercice 3 : Implémentation d'un dictionnaire

Nous souhaiterions utiliser le type `ABR a` défini dans l'exercice précédent pour implémenter un *dictionnaire*, c'est-à-dire un ensemble (fini) d'associations `(clé, valeur)`. On note `a` le type des clés et `b` le type des valeurs. L'arbre représentant le dictionnaire devra donc porter des valeurs de type `(a,b)`. Les nœuds de l'arbre seront triés selon la valeur des clés : si un nœud porte un couple clé-valeur `(c,v)`,

— tous les nœuds de son sous-arbre gauche porteront des clés strictement inférieures à `c`,

— tous les nœuds de son sous-arbre droit porteront des clés strictement supérieures à `c`.

Enfin, chaque clé ne peut apparaître qu'une seule fois dans le dictionnaire.

Q 3.1 Spécialisation du type `ABR a`

Définir un synonyme de type `Dictionnaire a b` qui spécialise le type `ABR a` en un arbre portant des couples `(clé, valeur)`.

Q 3.2 Insertion d'un couple `(clé, valeur)`

On souhaite écrire une fonction qui insère un couple `(clé, valeur)` dans un dictionnaire.

1. On a défini dans l'exercice précédent une fonction `insere` qui insère une valeur dans un arbre de type `ABR a`. Pourquoi cette fonction ne convient-elle pas pour les dictionnaires ?
2. Donnez la définition d'une fonction `insereD` telle que `insereD cv d` insère le couple clé-valeur `cv` dans le dictionnaire `d` et renvoie le nouveau dictionnaire obtenu.

Q 3.3 Recherche d'une valeur par clé

Implémenter une fonction `assoc` telle que `assoc c d` renvoie la valeur associée à la clé `c` dans le dictionnaire `d`, si `c` est présente dans le dictionnaire.

Q 3.4 Compatibilité avec `ABR a`

Comment aurait-on dû définir le type `Dictionnaire` pour pouvoir réutiliser directement les fonctions définies sur le type `ABR a` de l'exercice précédent ?

Exercice 4 : Arbres équilibrés

Dans cet exercice, on s'intéresse à l'équilibrage des arbres binaires de recherche. Sauf mention contraire, on utilisera le type `ABR a` défini dans l'Exercice 2.

Q 4.1 Vérifier l'équilibre d'un arbre

Pour que les arbres de recherches constituent des implémentations efficaces des ensembles ou des dictionnaires, il est nécessaire qu'ils soient équilibrés. Cela signifie que, pour un arbre ayant deux sous-arbres `g` et `d`, il faudrait que :

1. les sous-arbres `g` et `d` aient des hauteurs qui diffèrent d'au plus 1 ; et
2. tout sous-arbre vérifie cette propriété.

Il y a plusieurs remarques importantes concernant cette propriété :

— dès que l'on a trouvé un sous-arbre de l'arbre testé qui ne vérifie pas la propriété 1, on sait que l'arbre n'est pas équilibré,

— si, par contre, un sous-arbre est équilibré, il faut connaître sa hauteur afin de pouvoir la comparer à celle de son voisin (si il en a un) et vérifier que son nœud parent est équilibré.

Ainsi, pour vérifier qu'un arbre est équilibré, on peut s'appuyer sur une fonction auxiliaire qui renvoie `Nothing` lorsqu'il n'est pas équilibré et `Just h` lorsque l'arbre est équilibré et que sa hauteur est `h`.

Donnez une implémentation de la fonction `estEquilibre` qui vérifie qu'un arbre est équilibré.

Q 4.2 Un type pour les arbres binaires de recherche équilibrés

Lorsque l'on manipule des arbres binaires de recherche que l'on veut équilibrés, on ajoute généralement aux nœuds une information concernant l'équilibrage.

1. Proposez une nouvelle structure de données générique proche de la définition de `ABR a` qui permette de faire cela.
2. Définissez un type qui permette d'indiquer de quel côté penche un nœud :
   — soit son fils gauche a une plus grande hauteur (de 1) que son fils droit,
   — soit son fils droit a une plus grande hauteur (de 1) que son fils gauche,
   — soit ses deux enfants ont la même hauteur.
3. Définissez finalement, à partir des deux types précédents, le type `AVL a`, qui est un arbre binaire de recherche équilibré et qui contient l'information d'équilibrage qui indique de quel côté chaque nœud penche.

Files de priorité

Dans les exercices suivants, on s'intéresse aux *files de priorité*. Une file de priorité est une structure de données portant des valeurs (d'un type unique quelconque) associée à une valeur de priorité. Les éléments d'une file de priorité sont défilés par ordre croissant de priorité [1] : lorsque l'on extrait un élément de la file, c'est l'élément de priorité minimale qui est retiré de la file et renvoyé.

Les files de priorités peuvent être implémentées de différentes manières : par des listes, des arbres binaires de recherche, ou des tas [2]. L'enjeu de ces implémentations est de minimiser la complexité des opérations d'insertion et d'accès aux éléments de la file. L'objectif de cet exercice est de comparer différentes implémentations des files de priorités.

On définit la classe de types `FileP` qui fournit les opérations élémentaires sur une file de priorité :

```
class FileP f where
    - renvoie une file vide
    vide :: f p v

    - estVide f renvoie vrai si la file f est vide, ou faux sinon
    estVide :: f p v -> Bool

    - inserer p v f insère une valeur de priorité p dans la file f
    - et renvoie la nouvelle file obtenue
    inserer :: Ord p => p -> v -> f p v -> f p v

    - pop f renvoie Just (v, f'), avec v un élément de priorité
    - minimale, et f' la file f privée de v.
    - pop f renvoie Nothing si f est vide.
    pop :: Ord p => f p v -> Maybe (v, f p v)
```

Dans cette classe de type, le type `f` ayant un instance de `FileP` est un type paramétré par deux variables de type `p` et `v` : `p` est le type des valeurs de priorité, et `v` le type des valeurs stockées dans la file.

Exercice 5 : Implémentation par listes triées

On souhaite d'abord implémenter une file de priorité à l'aide d'une liste chaînée de couples `(priorité, valeur)` triée par ordre croissant de priorité. L'élément de priorité minimale est ainsi toujours en tête de liste. La file pourra contenir plusieurs éléments de même priorité. S'il existe plusieurs éléments de priorité minimale, n'importe lequel de ces éléments pourra être renvoyé lors d'un appel à `pop`.

Q 5.1 Définition du type `FileL p v`

Définir un type `FileL p v` représentant une liste chaînée dont chaque maillon porte une valeur de type `v` associée à une priorité de type `p`.

Q 5.2 Instance de `FileL p v` dans `FileP`

Écrivez une instance de la classe de types `FileP` pour le type `FileL p v` défini précédemment.

Q 5.3 Complexité des opérations

Quelle est la complexité en temps au pire cas des fonctions `inserer` et `pop` dans cette implémentation ?

Exercice 6 : Implémentation (limitée) par arbres binaires de recherche

Dans cet exercice, on cherche à implémenter les files de priorité par un arbre binaire de recherche. Les nœuds de l'arbre binaire de recherche porteront une valeur et la priorité qui lui est associée. Pour garantir l'invariant des arbres binaires de recherche, on n'autorisera pas l'insertion de plusieurs valeurs de priorité égale (ce sera l'objet de l'exercice suivant). Si une valeur `v` de priorité `p` est présente dans l'arbre `a`, l'appel à `inserer p v' a` remplacera `v` par `v'` dans `a`.

Q 6.1 Définition du type `FileA p v`

Définir un type `FileA p v` représentant un arbre binaire portant des valeurs de type `v` associées à des priorités de type `p`.

1. On peut définir de manière équivalente des files de priorité donnant accès aux éléments par ordre décroissant de priorité.
2. `https://en.wikipedia.org/wiki/Heap_(data_structure)`

Q 6.2 Instance de `FileA p v` dans `FileP`

Écrivez une instance de la classe de types `FileP` pour le type `FileA p v` défini précédemment. On ne cherchera pas à équilibrer l'arbre binaire de recherche.

Q 6.3 Complexité des opérations

Quelle est la complexité en temps au pire cas des fonctions `inserer` et `pop` dans cette implémentation ?

Exercice 7 : Implémentation (complète) par arbres binaires de recherche

Dans cet exercice, on utilise une implémentation par arbre binaire de recherche qui lève la restriction précédente sur l'unicité des valeurs de priorité. Pour cela, chaque nœud de l'arbre portera une priorité, et une liste des valeurs associées à cette priorité. Lors de la recherche d'un élément de priorité minimale, n'importe quel élément de la liste correspondante pourra être renvoyé.

Q 7.1 Définition du type `FileAL p v`

Définir un type `FileAL p v` représentant un arbre binaire dont chaque nœud porte une priorité de type `p` associée à une liste de valeurs de type `v`.

Q 7.2 Instance de `FileAL p v` dans `FileP`

Écrivez une instance de la classe de types `FileP` pour le type `FileAL p v` défini précédemment. On ne cherchera pas à équilibrer l'arbre binaire de recherche. Il faudra veiller à ce que les filtrages de motif soient exhaustifs ; pour les cas qui ne sont pas censés se produire, on pourra utiliser la fonction `error :: String -> a` pour lever une exception.

Q 7.3 Complexité des opérations

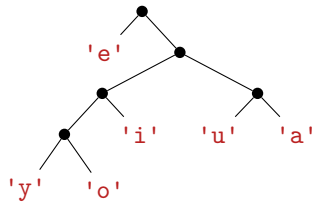
Quelle est la complexité en temps au pire cas des fonctions `inserer` et `pop` dans cette implémentation ?

Arbres de Huffman

Exercice 8 : Construction d'arbres de Huffman

Dans cet exercice, on travaille sur les arbres de Huffman vus au TD précédent. Pour rappel, le codage de Huffman peut être utilisé pour compresser une séquence de valeurs (par exemple un texte, qui est une séquence de caractères) en remplaçant chaque valeur par son code, qui est une séquence de bits. Les valeurs les plus fréquentes sont associées à des codes plus courts, ce qui permet de minimiser l'espace mémoire requis pour stocker les données.

Les codes sont construits à l'aide d'un arbre binaire. Les feuilles de l'arbre portent les valeurs à coder. L'arbre suivant est un exemple d'un tel arbre, contenant les voyelles de l'alphabet :



Dans un tel arbre, chaque valeur peut être identifiée par le chemin qui va de la racine à la feuille correspondante. Ce chemin est représenté par la liste des directions (gauche ou droite) à prendre à chaque nœud interne pour atteindre la feuille. L'arbre est construit de manière à ce que les valeurs les plus fréquentes soient situées plus haut dans l'arbre ; elles ont ainsi un code (chemin racine-feuille) plus court.

L'arbre de l'exemple a été produit à partir des fréquences d'occurrence (exprimées en %) suivantes des voyelles : `[('a', 17.5), ('e', 42.2), ('i', 14.3), ('o', 10.4), ('u', 14.7), ('y', 0.8)]`. Le caractère le plus fréquent, `'e'`, est placé le plus haut dans l'arbre, ce qui lui attribue le code le plus court. Inversement, les voyelles les moins fréquentes, `'y'` et `'o'`, sont placées au niveau le plus profond.

L'objectif de cet exercice est d'écrire des fonctions permettant de construire un arbre de Huffman portant un type quelconque de données. Pour représenter les couples (valeur, fréquence) des données, on définit le type `Freq a` suivant :

```
data Freq a = F a Float
```

où le premier champ, de type `a`, est la valeur, et le second champ, de type `Float`, est la fréquence de cette valeur.

Pour les exemples, on définit également la liste `voyellesF :: [Freq Char]` qui contient les voyelles et les fréquences qui leur sont associées :

```
voyellesF :: [Freq Char]
voyellesF = [F 'a' 17.5, F 'e' 42.2, F 'i' 14.3, F 'o' 10.4, F 'u' 14.7, F 'y' 0.8]
```

Q 8.1 Type `ArbreH a`

Donnez la définition du type algébrique d'un arbre de Huffman générique dont les feuilles portent des valeurs de type `a`.

Q 8.2 Tri des valeurs de type `Freq a`

Pour construire un arbre de Huffman à partir d'une liste de fréquences, il sera nécessaire de manipuler des listes de type `[Freq a]` triées par ordre croissant de fréquence, puisque la position des valeurs dans l'arbre dépend de leur fréquence.

Nous avons ainsi besoin de définir deux fonctions :

— `sortF :: [Freq a] -> [Freq a]`, qui permet de trier une liste d'éléments de type `Freq a` par fréquence croissante,

— `insertF :: Freq a -> [Freq a] -> [Freq a]`, qui permet d'insérer dans une liste triée par fréquence croissante un élément de type `Freq a` pour obtenir une liste triée.

Voici des exemples d'exécution de ces fonctions :

```
ghci> sortF voyellesF
[F 'y' 0.8, F 'o' 10.4, F 'i' 14.3, F 'u' 14.7, F 'a' 17.5, F 'e' 42.2]
ghci> insertF (F 'r' 12.3) (sortF voyellesF)
[F 'y' 0.8, F 'o' 10.4, F 'r' 12.3, F 'i' 14.3, F 'u' 14.7, F 'a' 17.5, F 'e' 42.2]
```

Pour écrire ces fonctions, on peut s'appuyer sur les fonctions suivantes du module `Data.List` :

— `sort :: Ord a => [a] -> [a]`, telle que `sort l` renvoie la liste `l` triée par ordre croissant,

— `insert :: Ord a => a -> [a] -> [a]`, telle que `insert e l` insère dans la liste *triée* `l` la valeur `e` et renvoie la nouvelle liste triée obtenue.

1. Déclarez la (ou les) instance(s) de classes de types nécessaires pour utiliser `sort` et `insert` sur des listes de type `[Freq a]`, de sorte à ce que le tri soit effectué en ordre croissant des fréquences.
2. Écrire la fonction `sortF :: [Freq a] -> [Freq a]`. Vous pouvez utiliser la fonction `sort`.
3. Écrire la fonction `insertF :: Freq a -> [Freq a] -> [Freq a]`. La liste d'entrée est supposée triée. Vous pouvez utiliser la fonction `insert`.

Q 8.3 Initialisation de l'algorithme

Écrire une fonction `initH :: [Freq a] -> [Freq (ArbreH a)]` qui transforme chaque association valeur-fréquence de la liste d'entrée en une association arbre-fréquence dont l'arbre est une feuille contenant la valeur initiale.

Exemple d'exécution :

```
ghci> initH voyellesF
[F (Feuille 'a') 17.5, F (Feuille 'e') 42.2, F (Feuille 'i') 14.3, F (Feuille 'o') 10.4,
F (Feuille 'u') 14.7, F (Feuille 'y') 0.8]
```

L'algorithme de construction de l'arbre de Huffman à partir d'une liste de type `[Freq (ArbreH a)]` triée (suivant les fréquences) est le suivant :

— si la liste ne contient qu'un seul élément, alors cet élément contient l'arbre de Huffman résultat ;

— sinon, on extrait de la liste les deux arbres `t1` et `t2` de fréquences `f1` et `f2` minimales, et on construit un nouvel arbre ayant `t1` pour fils gauche et `t2` pour fils droit, et associé à la fréquence `f1 + f2` ; ce nouvel élément est inséré dans la liste de manière à ce qu'elle soit toujours triée par fréquences croissantes ; cette liste contient un élément de moins que la liste d'origine.

Q 8.4 Fusion d'arbres

Ecrire une fonction `etapeH` qui prend en argument deux éléments `t1` et `t2` de type `Freq (ArbreH a)` et une liste de `Freq (ArbreH a)` triée dans l'ordre croissant des fréquences, et renvoie cette liste augmentée de l'élément construit à partir de `t1` et `t2` comme décrit ci-dessus. Pour rappel, la liste résultat doit être triée.

Exemple d'exécution :

```
ghci> etapeH (F (Feuille 'a') 17.5) (F (Feuille 'i') 14.3) [F (Feuille 'e') 42.2]
[F (Noeud (Feuille 'a') (Feuille 'i')) 31.8,F (Feuille 'e') 42.2]
```

Q 8.5 Construction de l'arbre

Ecrire une fonction `arbreH` qui construit un arbre de Huffman à partir d'une liste d'éléments de type `Freq a` (en ordre quelconque) donnée en entrée. Vous utiliserez `Maybe` pour éviter que la fonction ne déclenche une erreur dans le cas où la liste est vide.

Exemples d'exécution :

```
ghci> arbreH []
Nothing
ghci> arbreH voyellesF
Just (Noeud (Feuille 'e') (Noeud (Noeud (Noeud (Feuille 'y')
       (Feuille 'o')) (Feuille 'i')) (Noeud (Feuille 'u') (Feuille 'a'))))
```