Université de Lille

2026–2027
Licence Informatique

Programmation fonctionnelle

PF : TD 2

Exercice 1 : Échauffement : arbres binaires d'entiers

On considère des arbres binaires dont les feuilles portent une valeur entière. La Figure 1 donne des exemples de tels arbres. Il n'existe aucune règle particulière quant aux valeurs que ces arbres peuvent porter (hormis leur type) et leur emplacement dans l'arbre.

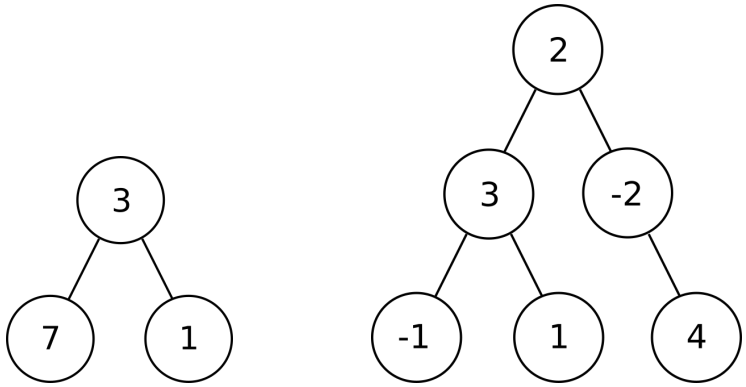


FIGURE 1 – Exemples d'arbres à instancier.

Pour représenter ces arbres, on définit le type algébrique suivant :

```haskell
data ArbreBin
    = Noeud Int ArbreBin ArbreBin
    | ArbreVide
```

Q 1.1 Instanciation d'arbres
Écrivez les expressions Haskell permettant d'instancier les arbres dessinés dans la Figure 1.

Q 1.2 Existence d'un élément
Écrivez un prédicat `existe :: ArbreBin -> Int -> Bool` tel que `existe a v` renvoie vrai si la valeur `v` est présente dans l'arbre `a`, ou faux sinon.

Q 1.3 Somme des éléments
Écrivez une fonction `sommePos :: ArbreBin -> Int` qui renvoie la somme des valeurs strictement positives portées dans l'arbre binaire passé en argument.

Q 1.4 Ajouter une valeur aux éléments
Écrivez une fonction `ajouter :: Int -> ArbreBin -> ArbreBin` telle que `ajoute v a` ajoute la valeur `v` à chaque nœud de l'arbre `a` et renvoie l'arbre ainsi obtenu.

Exercice 2 : Compter le nombre d'occurrences des éléments d'une liste

On traite à nouveau un problème vu au TD1 : compter le nombre d'occurrences des éléments d'une liste d'entiers. Au TD1, nous avons utilisé des listes associatives de type `(Int, Int)`, dont chaque couple `(v, n)` indique que la valeur `v` apparaît `n` fois dans la liste d'entrée. Dans cet exercice, pour calculer ces listes associatives, nous utiliserons des dictionnaires contenant des couples *(clef, valeur)*. Les clefs sont les entiers de la liste d'entrée, et les valeurs sont leur nombre d'occurrences. On implémente le dictionnaire par un arbre binaire de recherche dont les nœuds portent un couple (clef, valeur) de type `(Int, Int)`. On rappelle l'invariant des arbres binaires de recherche :

— les nœuds situés dans le sous-arbre gauche d'un nœud lui sont tous inférieurs,
— les nœuds situés dans le sous-arbre droit d'un nœud lui sont tous supérieurs.

Il doit donc exister une relation d'ordre total entre les valeurs portées par les nœuds de l'arbre. Ici, cette relation d'ordre porte sur la clef portée par les nœuds : cette clef est de type `Int`, donc la relation d'ordre entre les nœuds est celle qui existe sur les valeurs de type `Int`. Cette structuration de l'arbre permet de retrouver un élément dans l'arbre en ne parcourant qu'un unique chemin dans l'arbre, plutôt que l'ensemble des nœuds : à chaque nœud de l'arbre que l'on parcourt, on sait si la valeur recherchée doit se trouver à gauche ou à droite.

Q 2.1 Type `Dictionnaire`

Donnez la définition d'un type algébrique `Dictionnaire` qui représente un dictionnaire par un arbre binaire de recherche portant des couples (clef, valeur) de type `(Int, Int)`.

Q 2.2 Accès à une valeur

Écrivez une fonction `decompte :: Dictionnaire -> Int -> Int` telle que `decompte a c` renvoie le nombre d'occurrences associé à la clef `c` dans le dictionnaire `a`. Si une valeur est absente de l'arbre, c'est que son décompte d'occurrences vaut 0 : `decompte a c` renverra donc la valeur `0` si la clef `c` n'est pas présente dans l'arbre.

Q 2.3 Incrémentation d'un élément

On veut écrire une fonction `incrementer :: Dictionnaire -> Int -> Dictionnaire` telle que `incrementer a c` ajoute une occurrence à la clef `c` dans le dictionnaire `a`. Deux cas peuvent se produire :
— si `a` contient déjà la clef `c`, alors il suffit d'incrémenter la valeur associée dans le nœud concerné,
— sinon, il faut insérer une nouvelle feuille contenant le couple `(c, 1)`, en respectant l'invariant des arbres binaires de recherche.

Écrivez la fonction `incrementer`. On ne cherchera pas à équilibrer l'arbre binaire de recherche.

Q 2.4 Résoudre le problème initial

Écrire une fonction `compte :: [Int] -> [(Int, Int)]` telle que `compte l` :

1. construit un dictionnaire associant à chaque valeur de `l` son nombre d'occurrences,
2. renvoie la liste de tous les couples contenus dans le dictionnaire.

Q 2.5 Complexité

On veut comparer la complexité de cette solution à la complexité de celles vues au TD1.

1. Quelle est la complexité au meilleur cas de la fonction `compte` ?
2. Quelle est la complexité au pire cas de la fonction `compte` ?

Exercice 3 : Sommaire de documents

Cet exercice propose de manipuler et d'afficher une représentation de sommaire qui structure les différentes parties d'un document. Cette représentation est implémentée par le type suivant :

```
data Section = Section String [Section]
```

Le type `Section t som` représente une section avec un titre `t` et le sommaire de ses sous-parties `som`. Le type `Section` définit un arbre n-aire : chaque section est un nœud de l'arbre, et les enfants de ce nœud sont les sous-sections que la section contient. Une feuille est implicitement un nœud dont la liste d'enfants est vide.

Un sommaire est une liste de `Section`. Chaque section peut donc être vue comme contenant un sommaire, constitué des sous-arbres enfants du nœud représentant la section.

Étant donné un sommaire de la forme :

```
lyah = [Section "Introduction" [
            Section "About this tutorial" [],
            Section "So what's Haskell?" [],
            Section "What you need to dive in" []],
        Section "Starting Out" [
            Section "Ready, set, go!" [],
            Section "Baby's first functions" [],
            Section "An intro to lists" [],
            Section "Texas ranges" [],
            Section "I'm a list comprehension" [],
            Section "Tuples" []]]
```

nous cherchons à produire une chaîne de caractères qui permettra d'afficher le sommaire au format texte, numéroté de la façon suivante :

```
1 Introduction
..1.1 About this tutorial
..1.2 So what's Haskell?
..1.3 What you need to dive in
2 Starting Out
..2.1 Ready, set, go!
..2.2 Baby's first functions
..2.3 An intro to lists
..2.4 Texas ranges
..2.5 I'm a list comprehension
..2.6 Tuples
```

Q 3.1 Transformer sections et sommaires en chaînes de caractères

Nous allons implémenter deux fonctions mutuellement récursives `sectionToStrings :: Section -> [String]` et `sommaireToStrings :: [Section] -> [String]` qui transforment respectivement :

— une section en la liste des lignes utilisées pour son affichage, et

— un sommaire en la liste des lignes utilisées pour son affichage.

Ces deux fonctions se contentent d'assembler dans une liste les titres des sections, sans les modifier. Nous ajouterons une numérotation aux titres avec la prochaine question.

Écrivez les fonctions `sectionToStrings` et `sommaireToStrings`.

Q 3.2 Numérotation des sections et sommaires

Nous allons implémenter la numérotation des sommaires à l'aide de deux fonctions mutuellement récursives `numSommaire :: String -> [Section] -> [Section]` et `numSection :: String -> Int -> Section -> Section` telles que :

— `numSommaire s som` renvoie le sommaire `som` dont tous les titres de sections ont été modifiés pour faire figurer devant la chaîne de caractère `s`, suivie du numéro de la section et de son titre,

— `numSection s k sec` renvoie la section `sec` avec

  — un nouveau titre obtenu en ajoutant au début la chaîne `s`, le numéro de section `k`, et

  — un sommaire numéroté et dont les titres sont précédés de `".."` pour les décaler dans l'affichage.

Écrivez les fonctions `numSommaire` et `numSection`. Vous pourrez utiliser la fonction `show :: Int -> String` telle que `show n` renvoie la chaîne de caractères représentant l'entier `n`.

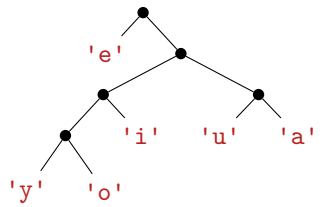
Q 3.3 Afficher un sommaire

Écrivez une fonction `showSommaire :: [Section] -> String` qui numérote, puis transforme en chaîne de caractères un sommaire.

Exercice 4 : Codage de Huffman

Le codage de Huffman peut être utilisé pour compresser une séquence de valeurs (par exemple un texte, qui est une séquence de caractères) en remplacant 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 à prendre à chaque nœud interne pour atteindre la feuille : `G` pour continuer dans le sous-arbre gauche ou `D` pour continuer dans le sous-arbre droit. Ce chemin constitue ainsi le code binaire représentant la valeur. Pour l'arbre ci-dessus, on a donc pour la lettre `'e'` le code `[G]`et pour la lettre `'i'` le code `[D,G,D]`. 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. Dans l'exemple donné, la voyelle la plus fréquente,

`'e'`, a un code à un bit (une direction), alors que `'i'`, qui est moins fréquente, a un code plus long, à trois bits (trois directions).

Une liste de directions menant à une feuille sera appelée *chemin valide* pour l'arbre considéré. Dans notre exemple ni `[G,D]` ni `[D,D]` ne sont des chemins valides, alors que `[G]` et `[D,G,D]` sont des chemins valides.

Dans cet exercice, on cherchera à écrire des fonctions de codage et décodage de valeurs de type `Char` à partir d'un arbre existant.

Pour représenter l'arbre et les codes de Huffman, on définit les types de données suivants :

```
data ArbreH
    = Feuille Char
    | Noeud ArbreH ArbreH
    deriving Show

data Direction = G | D
    deriving Show

data Chemin = Valide [Direction] | Invalide
    deriving Show
```

L'arbre de notre exemple peut donc être défini comme :

```
voyellesH :: ArbreH
voyellesH =
    Noeud (Feuille 'e')
        (Noeud (Noeud (Noeud (Feuille 'y') (Feuille 'o'))
            (Feuille 'i'))
            (Noeud (Feuille 'u') (Feuille 'a')))
```

Le chemin valide de la racine vers la valeur `'i'` sera représenté par `Valide [D,G,D]`, et un chemin invalide par `Invalide`.

Q 4.1 Existence d'une valeur dans l'arbre

Écrivez un prédicat `existe :: ArbreH -> Char -> Bool` tel que `existe a v` renvoie vrai si l'arbre `a` possède une feuille portant la valeur `v`, faux sinon.

Exemples d'exécution :

```
>>> existe voyellesH 'e'
True
>>> existe voyellesH 'y'
True
>>> existe voyellesH 'r'
False
```

Q 4.2 Encodage d'une valeur

Écrivez une fonction `chemin :: ArbreH -> Char -> Chemin` telle que `chemin a v` renvoie le chemin dans `a` vers la feuille portant la valeur `v` si `v` est dans `a`, ou `Invalide` sinon.

Exemples d'exécution :

```
>>> chemin voyellesH 'e'
Valide [G]
>>> chemin voyellesH 'i'
Valide [D,G,D]
>>> chemin voyellesH 'r'
Invalide
```

Q 4.3 Encodage d'une séquence de valeurs

On veut écrire une fonction `encoder :: ArbreH -> String -> Chemin` telle que `encoder a s` renvoie le code correspondant à la chaîne de caractères `s` selon l'arbre `a`. Si tous les caractères de `s` ont une chemin valide dans `a`, alors le code de `s` est obtenu en concaténant ensembles ces chemins. Sinon, le code pour `s` est invalide.

Voici quelques exemples d'exécution de `encoder` :

```
>>> encoder voyellesH ""
Valide []
>>> encoder voyellesH "aie"
```

```
Valide [D,D,D,D,G,D,G]
>>> encoder voyellesH "air"
Invalide
```

Pour écrire `encoder`, nous aurons besoin d'un opérateur `(+++) :: Chemin -> Chemin -> Chemin` permettant de concaténer des chemins valides. Exemples d'exécution :

```
>>> Valide [D,D,D] +++ Valide [D,G,D]
Valide [D,D,D,D,G,D]
>>> Valide [D,D,D] +++ Invalide
Invalide
>>> Invalide +++ Valide [D,G,D]
Invalide
```

1. Écrivez l'opérateur `(+++) :: Chemin -> Chemin -> Chemin`.
2. Écrivez la fonction `encoder :: ArbreH -> String -> Chemin`. Puisque toute chaîne contenant un caractère invalide produira un chemin invalide, `encoder` devra s'arrêter et renvoyer `Invalide` dès le premier caractère invalide rencontré.

Q 4.4 Décodage d'un chemin

On veut écrire une fonction `decoder` telle que `decoder a l` renvoie la valeur contenue dans `a` qui est codée par la liste de directions `l`. On ne sait pas si la liste de directions est valide : elle peut ne pas mener à une feuille de `a`. Pour représenter la validité du résultat, on introduit le type `ResChar` suivant :

```
data ResChar = Present Char | Absent
    deriving Show
```

La fonction `decoder` aura donc le type `ArbreH -> [Direction] -> ResChar`. Voici quelques exemples d'exécution de cette fonction :

```
>>> decoder voyellesH [G]
Present 'e'
>>> decoder voyellesH [D,G,D]
Present 'i'
>>> decoder voyellesH [D]
Absent
>>> decoder voyellesH [D,G,D,G]
Absent
```

Écrivez la fonction `decoder :: ArbreH -> [Direction] -> ResChar`.

Q 4.5 Décodage partiel d'un chemin

Comme on l'a vu précédemment, la fonction `encoder` renvoie une seule liste de directions contenant le code d'une chaîne de caractères. Cette liste est la concaténation des codes des caractères successifs de la chaîne. Par exemple, `"aie"` est encodée par `[D,D,D,D,G,D,G]`. Pour décoder ce code, il faut donc d'abord décoder `[D,D,D]`, qui correspond au caractère `'a'`, puis décoder le caractère suivant à partir du code restant `[D,G,D,G]`, et ainsi de suite. La fonction `decoder` écrite à la question précédente ne permet pas cela, car elle échoue s'il reste des directions dans le code. Il faudrait plutôt qu'elle renvoie la partie du code qui reste à traiter. On va donc remplacer `decoder` par une fonction `decoderUn :: ArbreH -> [Direction] -> (ResChar, [Direction])`. `decoderUn a ds` renvoie `(Present c, ds')` où `c` est le caractère encodé par les premiers éléments de `ds`, et `ds'` contient les éléments de `ds` qui n'ont pas été consommés pour décoder `c`. Si `ds` est un chemin invalide, `decoder a ds` renverra `(Absent, [])`.

Exemples d'exécution :

```
>>> decoderUn voyellesH [D,D,D,D,G,D,D]
(Present 'a', [D,G,D,D])
>>> decoderUn voyellesH [D,G,D,D]
(Present 'i', [D])
>>> decoderUn voyellesH [D]
(Absent, [])
```

Écrivez la fonction `decoderUn :: ArbreH -> [Direction] -> ResChar`.

Q 4.6 Décodage d'une chaîne de caractères

On souhaite maintenant utiliser `decoderUn` pour écrire une fonction `decoderTous :: ArbreH -> [Direction] -> String` qui décode une chaîne de caractères complète. `decoderTous a ds` renvoie la chaîne de caractères encodée par `ds`, si `ds` contient un code valide. Sinon, la chaîne de caractères renvoyée ne contiendra que les premiers caractères qui ont pu être décodés. Exemples d'exécution :

```
>>> decoderTous voyellesH [D,D,D,D,G,D,G]
"aie"
>>> decoderTous voyellesH [D,D,D,D,G,D,D]
"ai"
>>> decoderTous voyellesH []
""
```

Écrivez la fonction `decoderTous :: ArbreH -> [Direction] -> String`.