TP5: Types algébriques récursifs

1 Préliminaires

Configuration En vous aidant de la Notice ScalaIDE :

1. Lancez l'éditeur ScalaIDE (Eclipse 4.7 Scala)
2. Importez le projet ScalaIDE `TP5.zip` disponible sur Moodle.

Auto-évaluation des TP **ATTENTION !** Dans ce TP, on vous fournit uniquement **une grille** de tests unitaires, que vous devrez compléter. Consultez pour cela la notice ScalaIDE.

2 Les nombres bâtons : fichier `1-batons.scala`

Les entiers machines ont une taille bornée. Par exemple, en Scala, les entiers `Int` varient entre les deux entiers `Int.MinValue` et `Int.MaxValue`. Dans cet exercice, on souhaite modéliser des nombres entiers positifs ou nuls, sans limite de taille (autre que celle de la mémoire de votre ordinateur). Pour cela, on doit autoriser ces nombres à avoir une taille quelconque, bien que finie.

Une façon basique de s'y prendre consiste à représenter ces entiers par une suite de bâtons. Ainsi, pour représenter l'entier 5, on l'écrirait `| | | | |`, soit 5 bâtons côte-à-côte. Suivant ce principe, le type algébrique `Nombre` est défini dans le fichier source. C'est un type algébrique récursif. [1]

Exercice 1 (Construction de `val`). *Définir les différentes* `val` *indiquées par des* `TODO` *dans le fichier.*

Dans la suite, vous testerez chacune de vos fonctions sur quelques exemples bien choisis, par de simples affichages dans la console. Vous rédigerez également des **tests unitaires** dans la grille fournie, pour systématiser vos tests.

Exercice 2 (Conversion depuis les entiers). *Programmez la fonction* `fromInt`*, qui convertit un entier positif ou nul vers une valeur de type* `Nombre`*.*

Exercice 3 (Addition). *Programmez la fonction* `plus`*, qui réalise l'addition entre deux* `Nombre`*. Vous programmerez cette addition de manière récursive, et directement sur la représentation* `Nombre`*, sans re-passer par les entiers.*

Exercice 4 (Comparaison). *Programmez la fonction* `supOuEgal` *qui indique si un premier* `Nombre` *est supérieur ou égal à un deuxième* `Nombre`*. Vous programmerez cette comparaison de manière récursive, et directement sur la représentation* `Nombre`*, sans re-passer par les entiers.*

Exercice 5 (Conversion vers les entiers). *Programmez la fonction* `toInt` *qui convertit un* `Nombre` *vers un entier* `Int`*. Attention aux débordements !*

1. Cet encodage correspond aux entiers naturels de Peano. Voir `https://fr.wikipedia.org/wiki/Axiomes_de_Peano`.

3 Récursivité sur les listes : fichier `2-listes.scala`

Dans la suite, vous testerez chacune de vos fonctions sur quelques exemples bien choisis, par de simples affichages dans la console. Vous définirez également des **tests unitaires** dans la grille fournie.

Exercice 6 (Exercices faciles)**.** *Programmez les fonctions* `nfois` *et* `concat`*, repérées par des tâches* `TODO` *dans le fichier. Leur spécification précise est également indiquée dans le fichier.*

Exercice 7 (Tri par insertion)**.** *Un algorithme de tri permet de trier une collection suivant un ordre donné. Ici, on souhaite le programmer sur les listes d'entiers, et trier selon l'ordre croissant.*

L'algorithme du tri par insertion fonctionne suivant le principe suivant. Partant d'une liste non triée, chacun de ses éléments est inséré à la bonne place. La liste dans laquelle on insère les éléments est elle-même triée préalablement avec... un tri par insertion (par récursivité).

Implémentez la fonction `triInsertion` *qui, étant donnée une liste d'entiers, retourne la liste contenant les mêmes éléments, mais triée dans l'ordre croissant. La fonction doit être récursive. Vous pourrez définir une fonction auxiliaire, la fonction* `insertion` *qui, étant donnés un entier et une liste d'entiers triée dans l'ordre croissant, renvoie la liste (triée ! !) résultant de l'insertion de l'entier dans la liste à la bonne place.*

Exercice 8 (Test de `triInsertion`)**.** *On souhaite vérifier de manière systématique que la fonction de tri* `triInsertion` *est correcte sur des jeux de tests. On rappelle qu'une liste est la version triée d'une autre si :*

- *la liste est triée dans le bon ordre*
- *les deux listes sont des permutations l'une de l'autre : tous les élements de l'une sont contenus dans l'autre et réciproquement*

Implémentez la fonction `estTriee` *qui teste si une liste d'entiers est triée dans l'ordre croissant, puis la fonction* `versionTriee`*, qui détermine si une liste d'entiers est la version triée d'une autre liste d'entiers. Vous pourrez introduire des fonctions auxiliaires si nécessaire, en prenant soin de les spécifier précisément.* **Attention***, votre implémentation de* `versionTriee` *devra être* **indépendante** *de la fonction* `triInsertion`*, sans quoi vos tests ne pourraient détecter aucun bug.*

Utilisez ces fonctions sur plusieurs exemples bien choisis, puis définissez des tests unitaires de votre algorithme de tri. Vous aurez besoin de tester de manière unitaires les fonctions `estTriee` *et* `versionTriee` *!*

4 Les listes non vides : fichier `3-listesNonVides.scala`

Les listes sont disponibles de base en Scala, mais le type `List` ne permet pas de se restreindre aux listes non vides. L'objectif de cet exercice est de définir un nouveau type, le type `ListeNVE` des listes non vides d'entiers, qui interdit de construire des listes vides. Comme pour les autres exercices, vous testerez chacune de vos fonctions sur quelques exemples bien choisis, par de simples affichages dans la console. Vous rédigerez également des **tests unitaires** dans la grille fournie, pour systématiser vos tests.

Exercice 9 (Dernier sur le type `List[Int]`)**.** *Pour motiver le problème, programmez d'abord une fonction* `dernier`*, qui calcule le dernier élément d'une liste (Scala) d'entiers. Attention à la spécification !*

Qu'observez-vous ? De quels moyens dispose t-on pour contourner le problème ?

Exercice 10 (Définir le type `ListeNVE`)**.** *Définissez, au moyen d'un type algébrique récursif, le type des listes non vides. On se restreint aux listes qui contiennent des entiers. On impose le nom du type, qui devra être* `ListeNVE`*. Vous êtes en revanche libre dans le choix des noms des constructeurs de ce type.*

Exercice 11 (Définition de `val`)**.** *Définir quelques* `val` *de type* `ListeNVE`*.*

Exercice 12 (Dernier sur le type `ListeNVE`)**.** *Programmez maintenant la fonction* `dernier`*, qui calcule le dernier élément d'une liste* `ListeNVE`*. Commentez.*