Groupe 5.1 17 octobre 2014
Anne-Sophie Branders, Christophe Deleval, Guillaume du Roy,
Alexander Gerniers, François Palumbo, Gregory Vander Schueren

Algorithmique et structure de données
Mission 2 : Rapport

Tout au long de la conception de ce programme, nous nous sommes efforcés de construire des classes aussi génériques que possibles et qui pourraient être réutilisées dans des travaux ultérieurs. Ces classes génériques se trouvent dans le module lib.

Nous avons tout d'abord créé une classe `RBinaryTree` qui implémente l'interface d'un arbre binaire de manière récursive, où chaque noeud est lui même un arbre. Cette classe peut être utilisée pour créer et manipuler tout types d'arbre binaire. Pour cette classe, nous nous sommes basés sur l'implémentation proposée dans le DSAJ-5 mais nous l'avons adaptée en fonction de nos besoins.

Une seconde classe, `FormalExpressionTree` est une extention de la classe `RBinaryTree`. Cette classe propose des méthodes génériques pour faciliter la manipulation d'arbres binaires qui représentent des expressions analytiques. Cette classe définit 3 méthodes qui seront extrêmement utiles pour la réalisation de cette mission :

- La méthode de classe **`treeFromString()`** est un méthode factory qui renvoie un arbre binaire formé depuis une expression analytique donnée en input sous forme de chaînes de caractères.

- La méthode d'instance **`toString()`** renvoie une représentation sous forme d'une chaîne de caractère de l'expression analytique stockée dans le `FormalExpressionTree`. Cette méthode implémente un algorithme similaire au *Tour d'Euler* pour construire cette représentation en chaîne de caractères.

- Finalement, une méthode **`derive()`** qui doit être implémentée par les sous-classes de `FormalExpressionTree`.

Nous avons implémenté 9 extentions de `FormalExpressionTree` qui représentent soit un opérateur, soit une variable/valeur présente dans l'arbre. Chacune de ces classes est responsable de connaître la forme de sa propre dérivée : chaque sous-classe implémente une méthode `derive()` qui construit de manière récursive un nouvel `FormalExpressionTree` représentant la dérivée de son expression. Ainsi, toutes les méthodes de dérivées sont contenues dans les classes représentant l'expression, et le reste du programme ne doit pas s'en soucier : il suffit juste d'appeler la méthode `derive()` sur chaque sous-arbre, sans même connaître son type ou l'opérateur qu'il contient, pour obtenir sa dérivée.

Cette implémentation est particulièrement utile pour les extensions. Ainsi, si on veut créer un opérateur “exp”, par exemple, il suffit juste de créer une nouvelle classe, disons `ExpOperator`, héritant de `FormalExpressionTree`. La seule chose à faire pour que l’exponentielle soit opérationnelle est implémenter sa méthode `derive()`. Cela permet de créer ce nouvel opérateur sans devoir apporter des modifications au reste du code, mis à par le fait que le programme doit savoir reconnaître l’opérateur “exp” dans la chaîne de caractères passée en argument.

Ainsi, le programme principal n’est responsable que de très peu de choses :

- Il est responsable de lire la chaîne de caractères représentant une expression analytique qui lui est passée en argument.
- Cette chaîne de caractère est ensuite passée à la méthode factory `treeFromString()` de `FormalExpressionTree` qui renvoie un arbre binaire formé représentant cette expression analytique.
- La programme appelle ensuite la méthode `derive()` sur la racine de cet arbre qui renvoie un nouvel arbre représentant la dérivée de l’expression initiale.
- Finalement, le programme écrit sur la sortie standard la représentation en chaîne de caractères de ce nouvel arbre en appelant la méthode `toString()` dessus.

Finalement, pour tester l’exactitude de notre programme, nous avons réalisé une courte suite de tests unitaires qui se trouve dans *`FormalExpressionTreeTests`* sous le module *Tests*. Cette suite de tests ne couvre que les 3 fonctions essentielles de notre programme :

- La méthode de classe `TreeFromString()` et la méthode d’instance `toString()` de `FormalExpressionTree` sont d’abord testées de manière conjointe.
- Ensuite la méthode d’instance `derive()` de `FormalExpressionTree` est testée.

Certaines hypothèses simplificatrices énoncées dans la mission nous ont permis de simplifier notre programme. Premier exemple, toutes les expressions sont entièrement parenthésées. Cela nous permet, pour construire notre arbre, de simplement se fier aux parenthèses pour déterminer l’ordre des opérations (plutôt que devoir gérer l’ordre de précédence des opérateurs).

Evidemment, il est toujours possible de traiter le cas où cette hypothèse simplificatrice n’est pas posée en ajoutant, par exemple, une première méthode qui traiterait l’expression afin de la renvoyer sous forme parenthésée afin qu’elle corresponde aux spécifications de notre programme.

Il est important de savoir que dans notre programme, la dérivée de nombre négatifs ou de variables négatives, comme ‘-x’, est considérée comme étant la dérivée de l’opération ‘(0-x)’. Cela donnera donc ‘(0-1)’ comme résultat. Nous avons choisi de faire cela car nous ne devions pas implémenter une nouvelle expression, mais juste rajouter une constante et effectuer une soustraction.