Scala objet : classes et filtrage

Nous avons précédemment vu comment utiliser certains types algébriques définis dans la librairie standard : les listes, le type option... Nous explorons aujourd'hui la définition de ces types algébriques, qui est un outil puissant pour manipuler des structures de données arborescentes. En Scala, ce mécanisme est lié aux objets et s'appelle les *case classes*.

1 Types algébriques simples

1. Déclarez une classe des booléens "maison" :

   ```
   abstract class Bool
   case class True() extends Bool
   case class False() extends Bool
   ```

2. Définissez une fonction `def not(b:Bool): Bool` par pattern matching sur `b`, et testez-la dans les deux cas.
3. Commentez maintenant un des deux cas de la fonction que vous venez d'écrire, de façon à rendre la fonction partielle. Compilez, testez et observer l'erreur à *run time*.
4. Rajoutez le modifieur `sealed` devant la déclaration de la classe `Bool`. Observez l'erreur à *compile time* ; décommentez maintenant le cas commenté en 3.
5. Définissez une classe `IOption`, qui contient soit un `Int`, soit rien. Implémentez la fonction `def map(o: IOption, f:Int=>Int): IOption`.
6. De même, définissez une classe `IList` des listes de `Int`.
7. Implémentez dans `IList` et testez une méthode `sum` qui renvoit la somme de tous les entiers présents dans la liste, et ceci de deux façons :
   — en la définissant dans le corps de `IList` par pattern-matching sur `this` ;
   — en la déclarant dans `IList` et en implémentant chaque cas dans chaque sous-classe.
   Ces deux définitions sont équivalentes. Dans le second cas, le choix à *run time* est fait grâce au mécanisme de *dispatch*.
8. Implémentez des deux façons pré-citées une méthode `length` qui renvoit la taille de la liste `this`, et une méthode `map` qui applique son argument `f:Int=>Int` à tous les éléments de la liste.
9. Modifiez maintenant le type `IList` en le type paramétrique `List`, paramétré par le type `A` des éléments qu'elle contient. Portez-y les fonctions `length`, `map` et `sum`. Vous rencontrez un problème ; lequel ?

2 Expressions arithmétique

1. Déclarer une classe `Expr` des expressions arithmétiques. Une expression arithmétique est définie comme le plus petit ensemble qui satisfait les clauses suivantes :
   — Tout entier est une expression ;

— Tout ensemble de caractères alphanumériques est une expression (ce sont les variables) ;
— Si e est une expression, alors $\square\ e$ est une expression, pour $\square$ n'importe quel opérateur unaire (-, !,...) ;
— Si e_1 et e_2 sont des expressions, alors $e_1 \circ e_2$ est une expression, pour $\circ$ n'importe quel opérateur binaire (+, -, *, /, ...). [1]

2. Définir les valeurs de type `Expr` correpondant aux expressions arithmétiques : $x+1$, $(x \times (x+1))/2$, $x \times (y \times z)$, $(x \times y) \times z$ et $(x+2)-4$.
3. Implémenter une méthode `depth` qui calcule la profondeur de `this` (le nombre d'opérateurs maximum rencontrés depuis sa racine jusqu'à une de ses feuilles)
4. Implémenter une méthode `simpl` qui transforme l'expression `this` en suivant ces règles arithmétiques :
   — $e+0=e$, $e-0=e$;
   — $-(-e)=e$;
   — $e+m+n=e+(m+n)$ avec $m,n \in$ `Int` ;
   — $m \circ n = p$ pour $m,n \in$ `Int`, $\circ$ une opération arithmétique usuelle et $p = m \circ n$;
   — $e+e=2e$; $e-e=0$;
   Testez cette fonction sur les exemples de 2. Comment se simplifie $1+(e+0)$?
5. On veut maintenant pouvoir simplifier des expressions "en profondeur". Définir une méthode `def map1(f:Expr=>Expr): Expr` qui applique `f` aux sous-expressions directes de `this`. Définir maintenant une méthode récursive `def deepSimpl(): Expr` qui simplifie en profondeur `this`. Elle n'utilisera que `simpl` et `map1`.
6. En dehors de la classe, définir une fonction `def fixpoint[A](f:A=>A, x:A): A` qui calcule `f(f(f(...(f(x))...)))` jusqu'à ce que l'application de `f` ne change plus le résultat (son point fixe).
7. Finalement, dans `Expr`, la méthode `def simplify() = fixpoint[Expr](x => x.deepSimpl(), this)` effectue toutes les simplifications possibles, en profondeur. Testez-la.

1. On représentera les opérateurs en question simplement par des chaînes de caractère.