TP4 - Les visiteurs... sauce Scala

L'objectif de ce TP est de programmer en Scala un pretty printer et un évaluateur pour des programmes impératifs basiques. Le sujet est donc similaire au TD d'ACO qui vise le même objectif mais nous allons le traiter à l'aide de Scala pour illustrer certaines spécificités du langage. En particulier, l'utilisation des `case class` et des `match ... case` offre une alternative élégante à l'utilisation du patron de conception Visiteur.

1 Préambule

Vous allez utiliser Eclipse avec le plugin Scala. **Il est conseillé de vous créer un nouveau répertoire Workspace avant de lancer Eclipse Scala sur celui-ci**. Sur les postes ISTIC Linux, vous devez utiliser la version d'Eclipse accessible par le menu : Applications>Programmation>Eclipse 4.7 Scala GEN-ACF(ou exécuter dans un terminal : `/usr/local/eclipse47scala-gen-acf`). Ensuite, la marche à suivre est la suivante :

1. Importer le projet se trouvant dans l'archive `/share/m1info/ACF/TP4/TP4.zip` :

   File>Import>General>Existing Projects into Workspace>Select archive file

2. Pour créer d'autres objets/classes : Clic droit sur le projet puis Nouveau>Scala object (ou classe). Notez que contrairement à Java, un fichier Scala peut contenir plusieurs classes, objets, traits, ... Libre à vous d'organiser les fichiers comme bon vous semble !

2 Le type des arbres de syntaxes abstraits des programmes impératifs

Soit la grammaire suivante définissant des programmes impératifs basiques.

Grammaire	Exemple de programme

`Expression::= BinExpression | IntegerValue | VariableRef`
`BinExpression::= Expression; String; Expression`
`IntegerValue::= Int`
`VariableRef::= String`
 	
`Statement::= Assignment | Print | While | Seq | If | Read`
`Assignment::= String; Expression`
`Print::= Expression`
`While::= Expression; Statement`
`Seq::= Statement; Statement`
`If::= Expression; Statement; Statement`
`Read::= String`
	`{`
	`  x:= 0`
	`  y:= 1`
	`  read(z)`
	`  while ((x < z)) do`
	`  {`
	`  x:= (x + 1)`
	`  y:= (y * x)`
	`  print(x)`
	`  }`
	`  print(y)`
	`}`

On vous donne le code Scala définissant les classes nécessaires pour la représentation de ces programmes ainsi qu'un objet représentant une expression et un objet représentant le programme ci-dessus. Le code Scala définissant le type des expressions et les trois classes l'implémentant est le suivant :

```
sealed trait Expression
case class IntegerValue(i:Int) extends Expression
case class VariableRef(s:String) extends Expression
case class BinExpr(op:String, e1: Expression,e2: Expression) extends Expression
```

3 Un pretty printer

Voici un objet `PrettyPrinter` avec un extrait de l'opération `stringOf(e: Expression):String` permettant de produire la chaîne de caractère représentant l'expression `e`.

```
object PrettyPrinter{
  def stringOf(e:Expression):String={
    e match {
      case IntegerValue(i) => i.toString
      case VariableRef(v) => v
      ...
```

Définissez l'objet `PrettyPrinter` et équipez le d'une opération `stringOf(p: Statement):String` qui donne une chaîne de caractère représentant un programme. Tester votre fonction sur le programme `prog` donné. Il est conseillé d'utiliser la construction `match ... case`.

4 Un évaluateur

Définir un objet `Interpret` qui dispose d'une opération `eval(p:Statement,inList:List[Int]):List[Int]` qui permette d'évaluer un programme et retourne la liste des entiers affichés successivement par les instructions `print` du programme. La liste `inList` contient, elle, la liste des entiers successivement saisis par l'utilisateur, *i.e.* lus par les instructions `read`. Une table de type `Map[String,Int]` vous sera nécessaire pour associer des valeurs entières à des noms de variables. Il est également conseillé d'utiliser la construction `match ... case` pour la définition de `eval`.

5 Bonus... intégration du code généré par Isabelle et test aléatoire/exhaustif

Importez le projet se trouvant dans l'archive `/share/m1info/ACF/TP4/TP4bonus.zip`. Dans l'objet `tp3` du package `exportScala`, copiez l'objet Scala produit par Isabelle/HOL à partir de votre théorie du TP2/3. Pour que l'intégration de ce code se passe bien, il peut être nécessaire de décommenter les lignes suivantes dans `MainTest` :

```
implicit def equal_t[T]: HOL.equal[T] = new HOL.equal[T] {
    val `HOL.equal` = (a: T, b: T) => a==b
}
```

On va se servir de cette implantation (certifiée) pour découvrir des erreurs dans 5 autres implantations (non certifiées) : `Imp1, Imp2, Imp3, Imp4` et `Imp5`. Vous pouvez lancer l'application de test `MainTest` du package `tester`. Celle-ci compare les résultats pour `egal`, `inter` et `union` entre votre implantation et `Imp1, Imp2, Imp3, Imp4, Imp5`. Pour l'instant les tests ne sont effectués que sur un seul cas de test. Pour découvrir les bugs affectant ces 5 implantations, construisez un générateur produisant d'autres cas de tests en vous servant d'une approche aléatoire ou exhaustive.