TD 2: Premiers programmes en SCALA

Université Paris Diderot – Master 2 Informatique

21 septembre 2015

1 Environnement de développement pour Scala

1.1 Installation de Scala

Sur `lucien` La dernière version du compilation SCALA est installée sur `lucien`. Pour utiliser le compilateur en ligne de commande, il pourrait être nécessaire de mettre à jour votre variable d'environnement $PATH :

```
~% export PATH=$PATH:/usr/local/scala/bin
```

Après avoir relancé votre shell, vous devriez alors obtenir :

```
~% scala
Welcome to Scala version 2.9.1.final (Java HotSpot(TM) 64-Bit Server VM, Java 1.6.0_26).
Type in expressions to have them evaluated.
Type help for more information.
scala>
```

qui est la boucle interactive de SCALA.

Sur votre machine personnelle En fonction de votre environnement de travail et votre système d'exploitation plusieurs solutions s'offrent à vous pour installer SCALA. Plus d'informations ici : `http://www.scala-lang.org/downloads` .

Si vous rencontrez des difficultés, posez des questions sur la mailing-list du cours.

La distribution standard de Scala Voici les commandes principales de la distribution de SCALA :

Commande	Description
`scala`	la boucle interactive / interprète.
`scalac`	le compilateur.
`fsc`	le serveur de compilation.
`scaladoc`	le générateur de documentation de code.

1.2 Emacs

Pour les utilisateurs d'EMACS, le mode SCALA est pré-installé sur `lucien`. Sur votre machine personnelle il suffit de telecharger le paquet `scala-mode-el` ou bien de rajouter les lignes suivantes dans votre fichier `.emacs` (ici SCALA se trouve a `/usr/local/src/scala/`) :

```
(add-to-list 'load-path "/usr/local/src/scala/misc/scala-tool-support/emacs")
(require 'scala-mode-auto)
(add-hook 'scala-mode-hook '(lambda () (yas/minor-mode-on)))
```

Pour utiliser efficacement ce mode Emacs, consultez : `http://www.scala-lang.org/node/354` .

1.3 Eclipse

Il existe un *plugin* SCALA pour ECLIPSE : `http://www.scala-ide.org/` . Pour l'installer, suivre dans Eclipse `Help > Install New Software...` et ajouter le repository de plugins `http://download.scala-ide.org/releases-juno-29/milestone/site/` .

Une fois installé, pour créer un projet SCALA dans Eclipse, assurez vous que la *perspective* SCALA soit active, en suivant `Window > Open Perspective > Other` et choisissant `Scala`. Dans la salle TP, le plugin SCALA pour Eclipse est déjà installé, mais :

— il faut lancer `Eclipse Classic`, et

— vérifier que la *perspective* SCALA soit active, comme indiqué ci-dessus.

1.4 Outils essentiels

Comme pour l'apprentissage de tout langage de programmation, il faut consulter régulièrement la documentation de la bibliothèque standard pour se l'approprier progressivement : `http://www.scala-lang.org/api/current/` .

Par ailleurs, le langage SCALA est spécifié très précisément dans le document suivant : `http://www.scala-lang.org/docu/files/ScalaReference.pdf` .

Pour finir, le site des créateurs du langage fourmille de renseignements sur SCALA : `http://www.scala-lang.org` .

2 Bonjour le monde !

Exercice 1 *Voici un classique « Hello World ! » écrit en* SCALA *:*

```
01 object Hello extends App {println ("Hello World !")}
```

1. Compilez et exécutez ce programme.

2. Voici une autre façon de définir « Hello World ! », qui prend aussi en argument des chaînes de caractères de la ligne de commande

```
01 object HelloWorld {
02   def main (args :Array[String]) {
03     println ("Hello World !")
04   }
05 }
```

Sachant que `args.length` renvoie la taille du table `args`, modifiez ce programme pour qu'il affiche `Hello World!` si aucun argument n'est passé en ligne de commande et `Hello X_1 X_2 ... X_n!` si `X_1`, ..., `X_n` sont les arguments passés en ligne de commande. Écrivez plusieurs versions de ce programme, en utilisant les boucles while, les expressions conditionnelles, les notations « `for x` ← `...` ». Quelle est la solution la plus concise ? □

3 Table de hachage

Exercice 2 *On souhaite implémenter une structure de données qui associe des images de type `V` à des clés de type `K` en s'appuyant sur une fonction de hachage définie sur les clés de type `K`. On vous fournit un squelette de code dans la page suivante.*
1. Consultez l'API de la bibliothèque standard de SCALA *pour prendre connaissance des méthodes proposées par les types `List`, `Pair` (qui est un alias pour `Tuple2`), `Option` et `Array`.*

2. Donnez une implantation stupide de chacune de méthodes marquées `/*?*/` pour que ce programme compile.

3. Implantez la fonction `alloc_data` qui créé un objet de type `Table` dont tous les éléments ont été initialisés à `Nil`.

4. Implantez la fonction `add_in (table : Table, k : K, v : V)` qui calcule la valeur de hachage de `k` et insère le couple `(k, v)` dans la liste des couples de l'élément `k % table.length` de `table`.

5. Implantez la fonction `rehash ()` qui alloue une nouvelle table `new_data` de taille `2 * data.length + 1` et y réinsère tous les éléments de `data` avant de remplacer `data` par `new_data`

6. Implantez la fonction `add (k : K, v : V)` qui insère un nouveau couple `(k, v)` dans la table et vérifie ensuite que le nombre d'éléments de la table est inférieur à `data.length / 2`. Dans le cas contraire, on utilise la méthode `rehash` pour s'assurer que cet invariant est maintenu.

7. Implantez la fonction `find (k : K)` qui renvoie `None` si aucune image n'est associée à `k` et `Some (v)` si `(k, v)` est la dernière association de `k` insérée dans la table.

8. Bonus. Implémentez une nouvelle classe de table de hachage, alternative à SimpleHashTable, qui implémente la résolution de collision dite du coucou comme décrite ici : `http://en.wikipedia.org/wiki/Cuckoo_hashing` .

```
01 abstract class HashTable {
02   type K
03   type V
04
05   def hash (k :K) :Int
06   def add (k:K, v:V)
07   def find (k :K) :Option[V]
08 }
09
10 abstract class SimpleHashTable extends HashTable {
11   type Bucket = List[Pair[K, V]]
12   type Table = Array[Bucket]
13
14   def alloc_data (size :Int) :Table = { /*? */ }
15   def initial_size :Int
16   var data = alloc_data (initial_size)
17   var count = 0
18   def add_in (data :Table, k :K, v :V) { /*? */}
19   def rehash () { /*? */ }
20   def add (k :K, v :V) { /*? */ }
21   def find (k :K) :Option[V] = { /*? */ }
22 }
23
24 object Test extends App {
25   override def main (args :Array[String]) = {
26     val t = new SimpleHashTable {
27       type K = String;
28       type V = String;
29       def hash (k :K) = k.hashCode ()
30       def initial_size = 31
31     }
32     t.add ("Luke", "Darth");
33     t.add ("Leila", "Darth");
34     t.add ("George W", "George");
35     println (t.find ("Abama"));
36     println (t.find ("Leila"));
37   }
38 }
```

□

4 Modélisation objet d'une bibliothèque

Exercice 3 *On modélise une application devant servir à l'inventaire d'une bibliothèque. Elle devra traiter des documents de nature diverse : des livres, des dictionnaires, et autres types de documents qu'on ne connaît pas encore précisément mais qu'il faudra certainement ajouter un jour (articles, bandes dessinées...). Tous les documents possèdent un numéro d'enregistrement et un titre. À chaque livre est associé, un auteur et un nombre de pages, les dictionnaires ont, eux, pour attributs supplémentaires une langue et un nombre de tomes. On veut manipuler tous les articles de la bibliothèque au travers de la même représentation : celle d'un document.*

1. Après avoir dessiné le diagramme de classe UML de cette modélisation objet, définissez en SCALA *les classes* `Document`, `Book` *et* `Dictionary`. *(Essayez de rester le plus abstrait possible.)*

2. Définissez une classe `Library` réduite à une méthode main permettant de tester les classes précédentes (ainsi que les suivantes).

3. On souhaite autoriser des extensions futures de fonctionnalité des objets de la hiérarchie des documents sans que cela ne nécesse de modification dans le code des classes existantes. La nouvelle fonctionnalité est une méthode `keywords () : Set[String]` qui produit un ensemble de mot-clés à associer au document. Pour autoriser ces extensions futures, on modifie (pour la dernière fois !) les classes `Document`, `Book` et `Dictionary` en rajoutant une méthode « `def accept[Result] (v : Visitor[Result]) : Result` » où la classe `Visitor` est définie comme suit :

```
01 abstract class Visitor[Result] {
02   def visitBook (b :Book) :Result
03   def visitDictionary (d :Dictionary) :Result
04 }
```

Écrivez le corps des méthodes `accept` des classes `Dictionary` et `Book` puis implémentez un objet `Visitor` particulier qui récolte les mots-clés des documents présents dans leur titre et leur nom d'auteur (si il existe).

4. Implémentez la fonctionnalité précédente à l'aide d'une analyse de motif (pattern matching). Comparez ces deux approches. En particulier, comment faire évoluer ces implémentations lorsque l'on rajoute un nouveau type de document ?

□

5 Outils Logiques - trois ans après

Exercice 4 *Une formule de la logique propositionnelle est soit une variable propositionnelle numerotée, soit une negation d'une formule, soit une conjonction de deux formules, soit une disjonction de deux formules.*

1. Définir, en utilisant le case classes, *un classe `Formule` qui modélise les formules propositionnelles. Definir l'objet compagnon.*

2. Écrire une méthode qui renvoie l'ensemble des numéros de variables d'une formule. Écrivez deux versions, une dans l'objet compagnon qui utilise le pattern matching, *et une autre en redefinissant une méthode abstraite de la classe `Formule`. Dans l'objet compagnon, écrire une méthode qui prend en entrée un ensemble de formules et renvoie l'ensemble des numéros de variables des formule de l'ensemble. Utilisez `yield`.*

3. Un litéral *est une variable ou une négation d'une variable. Une* clause conjonctive *est une conjonction de litéraux. Une* forme normale de disjonction *est une disjonction de clauses conjonctives. Dans l'objet compagnon, écrire une méthode qui prend en entrée une formule et renvoie une formule equivalente en forme normale de conjonction. D'abord on poussera la négation vers l'intérieur, ensuite on distribuera la conjonction sur la disjonction.*

4. Une affectation *est une fonction (totale) qui prend en entrée un entier et qui renvoie une valeur booléenne. Intuitivement une affectation donne une valeur de vérité aux variables propositionnelles. Définissez une méthode d'évaluation d'une formule propositionnelle par rapport à une affectation. Ici encore, deux versions sont possibles.*

5. Le support *d'une affectation est l'ensemble des variables pour lesquelles l'affectation vaut `true`. Dans l'objet compagnon, écrire une méthode qui prend en entrée un ensemble X de numéros de variables et qui renvoie l'ensemble des affectations dont le support est contenu dans X.*

6. Une formule A est une tautologie *si A est vraie pour toute affectation. Pour vérifier cela il suffit de le vérifier pour les affectations dont les* support *est contenu dans l'ensemble de variables de A. Écrire une méthode qui dit si une*

formule est un tautologie. Vous pouvez aussi implémenter l'algorithme DPLL (`http://en.wikipedia.org/wiki/DPLL_algorithm`*) sur une forme normale de disjonction.*

7*. *Une formule* A implique *une formule* B *si* B *est vraie pour toute affectation pour la quelle* A *est vraie. Pour vérifier cela il suffit de le vérifier pour les affectations dont les* support *est contenu dans l'ensemble de variables de* A *et de* B*. Écrire une méthode qui dit si une formule implique une autre. Étant donné un ensemble de formules et une formule* A*, définissez une méthode qui dit s'il existe une formule qui est impliquée par* A*. Écrire une méthode qui prend en entrée un ensemble de formules* V *et qui renvoi un ensemble* W *qui est un sous-ensemble de* V *et qui est tel que aucune formule de* W *n'implique une autre formule de* W*.*

□