Programmation Avancée (INF441)
Contrôle classant

9 juin 2015

Les parties 1 et 2 sont indépendantes l'une de l'autre. Elles peuvent être traitées dans l'ordre de votre choix.

1 Graphes, itération et parcours

Dans cette partie, on demande d'écrire **du code Java** aussi **correct** que possible.

Les trois questions sont **entièrement indépendantes les unes des autres**.

On souhaite écrire une fois pour toutes un algorithme de parcours d'un graphe orienté. On souhaite donc que cet algorithme soit applicable quelle soit la façon dont le graphe est représenté.

La seule opération que doit permettre le graphe est l'énumération des successeurs d'un sommet `v` arbitraire. Pour exprimer cela, on définit une interface `ISuccessors<V>` (figure 1). Le paramètre `V` représente le type (a priori inconnu) des sommets. La méthode `successors` doit renvoyer un objet de type `Iterable<V>`, qui permet d'énumérer les successeurs du sommet `v`. La figure 2 rappelle la définition de toutes les classes et interfaces dont on aura besoin dans cette partie, dont `Iterable`.

On considère d'abord un graphe représenté explicitement en mémoire à l'aide de listes d'adjacence. La classe `AdjacencyGraph` (figure 3) représente un tel graphe. Ses sommets sont les entiers compris entre 0 inclus et `n` exclus. Ses arêtes sont données par le vecteur `successors`, qui contient une liste chaînée des successeurs de chaque sommet. (On a omis le constructeur, qui initialise les champs `n` et `successors` de façon à ce qu'ils représentent le graphe souhaité.)

Question 1 On souhaite que la classe `AdjacencyGraph` implémente l'interface `ISuccessors<V>` pour un certain type `V`. Indiquez quel doit être ce type `V`, puis ajoutez à la classe `AdjacencyGraph` les éléments nécessaires. (La solution est très courte ! On rappelle que la classe `LinkedList<E>` implémente l'interface `Iterable<E>`.) ◇

On considère maintenant un graphe représenté explicitement en mémoire à l'aide d'une matrice d'adjacence. La classe `MatrixGraph` (figure 4) représente un tel graphe. Ses sommets sont les entiers compris entre 0 inclus et `n` exclus. Ses arêtes sont données par la matrice `matrix`, qui aux sommets `i` et `j` associe un Booléen indiquant s'il existe ou non une arête de `i` vers `j`. (À nouveau, on a omis le constructeur de la classe `MatrixGraph`.)

Question 2 On souhaite que la classe `MatrixGraph` implémente l'interface `ISuccessors<V>` pour un certain type `V`. Indiquez quel doit être ce type `V`. Ajoutez ensuite à la classe `MatrixGraph` les éléments nécessaires. (Vous devrez à un certain moment écrire un objet de classe `Iterator<V>`. On conseille d'écrire la méthode `hasNext` avant la méthode `next`.) ◇

```
public interface ISuccessors<V> {
  Iterable<V> successors (V v);
}
```

FIGURE 1 – L'interface `ISuccessors`

```
interface Iterable<T> {
  Iterator<T> iterator ();
}
interface Iterator<T> {
  boolean hasNext ();
  T next () throws NoSuchElementException;
}
class Vector<E> {
  E get (int i);
  int size ();
}
class Stack<E> {
  boolean empty ();
  E peek ();
  E pop ();
  void push (E e);
}
class HashSet<E> {
  boolean add (E e);
  boolean contains (E e);
}
```

FIGURE 2 – Définitions (simplifiées) issues de la bibliothèque de Java

```
public class AdjacencyGraph {
  // The number of vertices.
  private final int n;
  // The adjacency lists.
  private final Vector<LinkedList<Integer>> successors;
}
```

FIGURE 3 – Un graphe représenté en mémoire à l'aide de listes d'adjacence (question 1)

```
public class MatrixGraph {
  // The number of vertices.
  private final int n;
  // The adjacency matrix, of size n * n.
  private final boolean[][] matrix;
}
```

FIGURE 4 – Un graphe représenté en mémoire à l'aide d'une matrice d'adjacence (question 2)

```
public class Traversal<V> {
  public Iterable<V> traverse (ISuccessors<V> g, V root) {
    ... // à compléter
  }
}
```

FIGURE 5 – La classe `Traversal` (question 3)

Dans la dernière question de cette partie, on souhaite implémenter un parcours de graphe en s'appuyant uniquement sur l'interface `ISuccessors<V>`, donc sans savoir de quelle manière le graphe est implémenté.

On souhaite utiliser le parcours de graphe comme une manière de découvrir et d'énumérer les sommets du graphe (plus précisément, les sommets accessibles à partir d'un sommet `root` fixé). Pour faciliter cette utilisation, on souhaite donner à l'utilisateur un objet de type `Iterable<V>` qui énumère ces sommets. On souhaite donc écrire l'algorithme sous la forme présentée dans la figure 5. Le paramètre `g` représente le graphe ; le paramètre `root` est le sommet à partir duquel le parcours doit avoir lieu.

Question 3 Implémentez la méthode `traverse`. Vous utiliserez un parcours en profondeur d'abord. Pour marquer les sommets, vous pourrez supposer que le type `V` est muni de méthodes `equals` et `hashCode` appropriées, et utiliser la classe `HashSet` (figure 2). Pour stocker les sommets en attente, vous pourrez utiliser la classe `Stack` (figure 2). ◇

2 Arbres binaires et séquences de poids

Dans cette partie, on demande d'écrire **du code OCaml** aussi **correct** que possible.

Certaines questions sont indépendantes de celles qui précèdent.

On se donne un type `tree` (figure 7) pour représenter des arbres composés de feuilles et de nœuds internes binaires (figure 6, partie gauche). Ces arbres ne contiennent pas d'éléments : seule leur forme nous intéresse. On appelle **poids** d'un arbre le nombre de ses feuilles.

Question 4 Écrivez une fonction `weight` de type `tree -> int` qui, étant donné un arbre, calcule son poids. Quelle est sa complexité asymptotique en temps ? ◇

Question 5 Écrivez une fonction `enumerate` de type `int -> (tree -> unit) -> unit` telle que, si `n` est un entier supérieur ou égal à 1, alors `enumerate n f` applique la fonction `f` sucessivement à chacun des arbres de poids `n`, dans un ordre arbitraire. ◇

La **séquence de poids** d'un arbre t est définie par la fonction `weights` (figure 8). On rappelle que la fonction `@` effectue la concaténation de deux listes simplement chaînées. Si t est un arbre de poids n, alors sa séquence de poids est une liste de $n-1$ nombres entiers strictement positifs. Par exemple, la séquence de poids de l'arbre situé à gauche de la figure 6 est 1, 2, 1, 1, 5, 1, 1, 3.

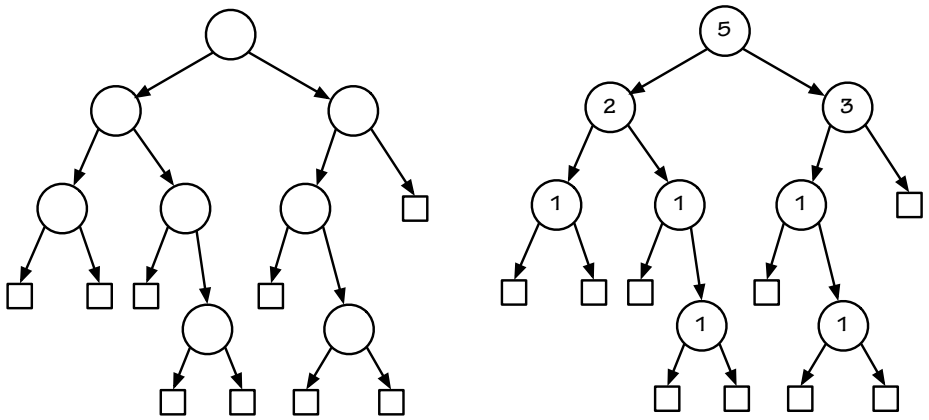

FIGURE 6 – À gauche, un arbre de type `tree`. À droite, un arbre de type `wtree` (question 7).

```
type tree =
| Leaf
| Node of tree * tree
```

FIGURE 7 – Un type d'arbres binaires

```
let rec weights (t : tree) : int list =
  match t with
  | Leaf ->
      []
  | Node (t1, t2) ->
      weights t1 @ ([ weight t1 ] @ weights t2)
```

FIGURE 8 – Calcul de la séquence de poids

Question 6 Quelle est la complexité asymptotique en temps de la fonction `weights` ? ◇

On note que la séquence de poids d'un arbre t peut être obtenue par un parcours infixe de l'arbre t où chaque nœud a été préalablement annoté par le poids de son sous-arbre gauche. (On ne demande pas de démonstration de cette affirmation.) On rappelle que le mot « infixe » signifie ici que lorsqu'on parcourt un nœud binaire, on considère d'abord son sous-arbre gauche, puis le nœud lui-même, puis son sous-arbre droit.

L'affirmation ci-dessus suggère que l'on peut calculer la séquence de poids d'un arbre t en deux phases successives : d'abord, on construit un arbre où chaque nœud est annoté par le poids de son sous-arbre gauche (figure 6, partie droite) ; puis, on parcourt ce nouvel arbre et on énumère les annotations que l'on rencontre.

Question 7 Définissez un type `wtree` représentant un arbre où chaque noeud binaire est annoté par un poids (qui est en principe le poids de son sous-arbre gauche). ◇

Question 8 Écrivez une fonction `annotate` de type `tree -> wtree * int` telle que `annotate t` renvoie une paire de (1) l'arbre `t`, où chaque nœud est annoté par le poids de son sous-arbre gauche ; (2) le poids de l'arbre `t`. Vous prendrez soin que la complexité asymptotique en temps de cette fonction soit linéaire. ◇

Question 9 Écrivez une fonction `wtree_iter` de type `wtree -> (int -> unit) -> unit` telle que `wtree_iter wt f` applique la fonction `f` successivement à toutes les annotations rencontrées lors du parcours infixe de l'arbre annoté `wt`. ◇

Question 10 À l'aide des fonctions `annotate` et `wtree_iter` demandées dans les deux questions précédentes, écrivez une nouvelle fonction `weights` de type `tree -> int list` et de complexité linéaire qui, étant donné un arbre, construit sa séquence de poids. ◇

Question 11 Écrivez une fonction `weights_iter` de type `tree -> (int -> unit) -> int` qui effectue en une seule passe (donc sans construire aucun arbre annoté) le travail des fonctions `annotate` et `wtree_iter` des questions 8 et 9. En d'autres termes, il faut que `weights_iter t f` applique la fonction `f` successivement à chacun des éléments de la séquence de poids de l'arbre `t` et renvoie le poids de l'arbre `t`. ◇