DM : Arbres et partage

DM à faire seul(e), à rendre pour le ***7 février 2017 à 23h59*** *au plus tard (pour les débutants : 14 février à 23h59). Envoyez une archive qui compile à* `aurore.alcolei@ens-lyon.fr, bertrand.simon@ens-lyon.fr, daniel.hirschkoff@ens-lyon.fr`

1 Version "de base" (débutants)

1.1 Ce que doit faire le programme

On considère le type Caml suivant pour représenter des arbres binaires.

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

Une valeur de ce type peut faire intervenir (ou pas) du partage, comme le montrent les exemples suivants :

```
let t1 = Node(0,Node(0,Leaf,Leaf), Node(0,Leaf,Leaf))

let t2 =
  let t' = Node(0,Leaf,Leaf) in
  Node(0,t',t')

let t3 =
  let t'=Node(0,Leaf,Leaf) in
  Node(0,Node(0,t',Leaf),Node(0,Leaf,t'))
```

La fonction suivante calcule la taille d'un arbre *au sens structurel* :

```
let rec structural_size = function
  | Leaf -> 0
  | Node(_,t1,t2) -> 1+structural_size t1+structural_size t2
```

Le but de cette partie est de programmer une fonction qui calcule la taille au sens physique, autrement dit, le nombre de nœuds qui sont effectivement stockés en mémoire[1].

L'idée est de parcourir un arbre, en détectant si on tombe sur un nœud qu'on a déjà rencontré précédemment.

Pour ce faire, vous devrez programmer une librairie simple permettant de stocker les nœuds de l'arbre qui ont déjà été visités. Voici un type de module pour cette librairie :

```
module type VisitedNodes =
sig
  type t  (* the structure where nodes are stored *)
  val create : unit -> t
  val lookup : tree -> t -> bool (* when the answer is "false", the structure is extended*)
  val tolist : t -> tree list
end
```

Pour implémenter la structure mentionnée ci-dessus, vous pourrez utiliser une simple liste Caml.

Il vous est demandé pour ce DM de définir un module de type `VisitedNodes`, puis une fonction `physical_size : tree -> int` utilisant ce module pour calculer la "taille physique" d'un arbre.

Le code à produire est relativement simple, il est préférable (pour ce DM) de ne pas faire appel à des bibliothèques Caml existantes.

[1] Il faudra donc s'appuyer sur l'égalité physique de Caml, évoquée en cours, et que l'on utilise avec `==`.

1.2 Structuration en plusieurs fichiers

Votre code devra consister en *au moins trois fichiers* :

- Le fichier où est défini le module ayant pour type `VisitedNodes` ; ce fichier s'appellera `visited.ml`.
- Le fichier principal, appelé `physize.ml`, s'appuyant sur le précédent.
- Un fichier de tests, où l'on définit des arbres et on calcule leurs tailles.

  Pour ce dernier fichier, vous pouvez adopter une solution minimale, où vous définissez "à la main" un nombre non ridicule (disons à au moins deux chiffres) d'arbres. Vous pouvez également avoir davantage d'ambition, en réfléchissant à la définition d'une fonction engendrant plus ou moins aléatoirement un arbre avec partage.

La définition du type `tree` sera, en première approche, dans le fichier `visited.ml`. On pourra se convaincre dans un second temps que la structure pour les nœuds visités peut être programmée sans qu'il soit nécessaire de savoir si on stocke des arbres ou des chips (si vous êtes débutant, vous pouvez vous limiter à vous convaincre de la chose).

2 Pour les intermédiaires

Le menu pour les intermédiaires est le menu pour les débutants, modifié et enrichi par les consignes ci-dessous.
Pour les deux premières questions, il vous est demandé de programmer une version améliorée de la réponse de base. Si vous ne savez pas répondre, faites la version de base ; sinon, inutile de fournir les deux versions.

1. Redéfinissez le type `tree` de manière à ce que l'information qui décore le nœud (ci-dessus, un `int`) soit paramétrique. En effet, ce que l'on programme ici ne doit pas dépendre de cette information (que ce soit un entier, un triplet de chaînes de caractères, etc.).

2. Reprenez le type de module `VisitedNodes` pour le transformer en un type de foncteur (ou module paramétré), prenant en argument un module correspondant aux éléments qui sont stockés. Le module en paramètre doit fournir un type avec une fonction pour tester l'égalité. Les mentions de `tree` dans les types des fonctions `lookup` et `tolist` seront remplacées par `elt`, `elt` étant le type des "choses" que l'on stocke dans la structure qui, elle, a le type `t`.

   Définissez alors `structural_size` et `physical_size` de manière uniforme, en s'appuyant sur deux implémentations différentes du module correspondant aux arbres, avec deux fonctions d'égalité adaptées.

   On pourra remarquer au passage qu'il est inutile de reprogrammer ainsi l'égalité structurelle : certes, il s'agit là d'un "exercice de style".

3. *(ne traitez cette partie que si les deux précédentes sont bien traitées)*

   Ajoutez un fichier `dessine.ml` où sera programmée une fonction `affiche : tree -> unit`, qui affiche à l'écran la représentation graphique d'un arbre au format `dot/graphviz`. Bien entendu, cette représentation devra tenir compte du partage, et dessiner des DAGs comme des DAGs (et des moutons comme des moutons).

   Vous trouverez sur la page du cours un exemple simple d'utilisation de `dot`.

3 Pour les avancés

De manière générale, les avancés sont sensés faire tout ce que doivent faire les débutants + tout ce que doivent faire les intermédiaires + le travail qui leur est spécifique.

On aimerait bien pouvoir utiliser des structures de données plus malines afin que l'accès à la librairie `VisitedNodes` soit plus efficace. Proposez une solution, en reprenant éventuellement la définition du type `tree`. Comparez les performances des deux représentations.

4 Tout le monde : en quoi consiste le rendu

Vous devez envoyer avant la date limite une archive qui compile, à
`aurore.alcolei@ens-lyon.fr, daniel.hirschkoff@ens-lyon.fr, bertrand.simon@ens-lyon.fr`.

Incluez dans votre archive un fichier README, contenant des commentaires sur votre code : indiquez notamment s'il y a des parties que vous n'avez pas traitées, s'il y a des bugs/imperfections dont vous êtes conscient(e).

Expliquez également comment l'on peut tester votre code (soit une version très rustique : exécuter pas à pas le fichier `tests.ml` ; ou alors des versions plus ergonomiques : à vous de réfléchir).

On ne le répètera jamais assez : il est très préférable de rendre quelque chose de propre, qui ne traite pas toutes les questions, plutôt que quelque chose qui essaie de tout traiter, mais est en vrac et ne marche pas.

"Propre" signifie en particulier :

- du code bien structuré et bien commenté ;
- des programmes testés, avec les fichiers de test que vous avez utilisés ;
- un fichier README.