Cours de programmation Java (PROG2)

Magistère MIT, 1re année
Examen terminal, 23 avril 2014

Les notes personnelles rédigées en cours, les transparents du cours et les autres documents sont autorisés.

Les trois problèmes sont indépendants. Tenez compte des indications de barème pour répartir votre temps entre eux.

Dans vos réponses aux questions vous demandant d'écrire du code, vous pourrez prendre quelques libertés avec la syntaxe du langage. En revanche, vous apporterez un soin particulier à l'orthographe et à la présentation de votre copie.

Merci de bien vous identifier sur chacune des copies, et de les numéroter.

1 Parallélisme des données (7 points)

Le Parallélisme par distribution des données ou parallélisme des données (*data parallelism* en anglais) est un paradigme de la programmation parallèle où on cherche à distribuer les données au sein des processus / threads et à y opérer les mêmes opérations (suivant un modèle SIMD – *single instruction multiple data*). Le paradigme opposé est celui du parallélisme de tâche. Ces algorithmes sont intéressants pour des calculs sur des quantités massives de données. Une approche (mais pas la seule) pour réaliser ces calculs est de prendre en entrée un sous-ensemble de données et produire en sortie un ensemble de paires clé-valeur extrait des entrées. Dans une deuxième étape, les valeurs sont agrégées, suivant les clés.

1.1 Illustrons ce mécanisme avec le problème du Word Count consistant à comptabiliser le nombre de mots contenus dans un (très grand) fichier (ou un ensemble de fichiers). L'opération de comptage des mots serait très longue si notre traitement se faisait de manière séquentielle. L'intérêt est de paralléliser les traitements sur l'ensemble des cœurs disponibles. Proposez un algorithme utilisant des threads qui résout ce problème.

1.2 Décrivez brièvement comment adapter le programme précédent pour résoudre un autre problème classique : Distributed Grep, qui consiste à la recherche d'un mot particulier (ou un document) dans un grand ensemble de données.

2 Un type option générique (5 points)

Dans cet exercice nous considérons le type suivant, analogue au type « `α option` » du langage OCaml.

```
interface Option<T> {
  boolean hasValue();
  T get();
}
```

Un objet de ce type contient au plus un objet de type `T`. La méthode `hasValue()` permet de déterminer si un tel objet est présent. Si `hasValue()` renvoie `true`, la méthode `get()` permet de récupérer ledit objet.

2.1 Programmez une fonction `some` dont la signature est la suivante. Étant donnée un objet `o` de type `T`, celle-ci construit un objet de type `Option<T>` qui renvoie `true` en réponse au message `hasValue()` et `o` en réponse au message `get()`.

```
static <T> Option<T> some(T o)
```

2.2 Programmez une fonction `none` dont la signature est la suivante. Cette fonction renvoie un *nouvel* objet qui répond `false` au message `hasValue()`.

```
static <T> Option<T> none()
```

2.3 La fonction précédente pourrait toujours renvoyer le même objet. Définissez un objet `NONE` dans ce but. Quel est son type ? Justifier.

2.4 Comment écrire alors la fonction `none()` ? Justifiez que votre programme est bien typé. Quelles vérifications de type auront lieu à l'exécution ? Peuvent-elles échouer ?

3 Paresse (8 points)

Être paresseux, c'est a) ne pas faire quelque chose tant que ce n'est pas nécessaire ; et b) ne pas faire deux fois la même chose. Il s'agit dans cet exercice de mettre en place des objets paresseux en Java, qui représentent chacun un calcul qui sera effectué au plus une fois et seulement lorsque l'on aura besoin du résultat.

Ce mécanisme sera basé sur la classe abstraite suivante.

```
abstract class Lazy<T> {
  protected abstract T get();
  T force() {/* … */}
}
```

Dans une instance de cette classe, la méthode `get` représente un calcul qui produit une valeur de type `T` ; celui-ci est exécuté la première fois que l'instance reçoit le message `force`. Tout message `force` ultérieur produira le même résultat mais sans refaire le calcul (c'est-à-dire sans appeler à nouveau la méthode `get`).

3.1 Mise en œuvre de la classe `Lazy`

a) Programmez la méthode `force`. Vous pourrez ajouter des champs privés à la classe `Lazy` si nécessaire.

b) Que se passe-t-il si la méthode `get` a des effets de bord ? Est-ce souhaitable ?

c) On souhaite généraliser la classe `Lazy` pour que la méthode `get` puisse lever une exception. Le cas échéant, le deuxième appel à `force` doit lever la même exception, mais sans appeler une deuxième fois `get`. Adaptez la classe `Lazy` en conséquence.

d) Que se passe-t-il si deux fils d'exécution concurrents envoient concurremment le message `force` à un même objet ? Adaptez (si nécessaire) la classe `Lazy` pour qu'il n'y ait pas de *data race* et que la méthode `get` ne soit pas appelée plus d'une fois. Peut-on éviter l'acquisition d'un moniteur à chaque appel de `force` ?

3.2 Application Grâce aux valeurs paresseuses, d'étonnantes listes chaînées peuvent être programmées. Voici une ébauche de classe abstraite représentant des listes paresseuses.

```
abstract class LazyList<T> {
  abstract Lazy<T> head();
  abstract Lazy<LazyList<T>> tail();
}
```

La méthode `head` renvoie le premier élément de la liste et la méthode `tail` la liste privée de son premier élément. Si la liste est vide, les deux méthodes lèvent une exception. Cependant, ces deux méthodes ne font aucun calcul (ni ne lèvent d'exception) mais renvoient un objet qui fera le calcul en réponse au message `force`.

La classe `Lazy` fait référence à celle étendue à la question 3.1.c pour permettre à la méthode `get` de lever une exception.

a) Programmez une fonction `nil` qui renvoie une liste vide.

```
static <A> LazyList<A> nil()
```

b) Programmez une fonction `cons` qui construit une liste à partir de son premier élément et de la liste des éléments suivants.

```
static <A> LazyList<A> cons(A a, LazyList<A> l)
```

c) Programmez une fonction `append` qui à partir de deux listes `l` et `m` construit une nouvelle liste qui contient d'abord les éléments de `l` puis ceux de `m`. Quelle est la complexité de cette fonction ? Commenter.

```
static <A> LazyList<A> append(LazyList<A> l, LazyList<A> m)
```

d) Programmez une fonction `repeat` qui à partir d'une valeur `a` produit une liste infinie dont tous les éléments sont `a`.

```
static <A> LazyList<A> repeat(A a)
```

e) Programmez une fonction `cycle` qui, étant donnée une liste, produit une liste infinie qui correspond à une répétition sans fin de la liste argument.

```
static <A> LazyList<A> cycle(LazyList<A> l)
```