Programmation avancée

Examen intermédiaire

Vendredi 18 novembre 2011

Nom : _____

Prénom : _____

Lisez attentivement le suivant pour ne pas gaspiller vos points *précieux* !

Votre nom Le travail qui ne peut pas vous être attribué est perdu: écrivez votre nom sur chaque feuille que vous rendez.

Votre temps Tous les points ne sont pas égaux. En effet, nous ne pensons pas que tous les exercices ont la même difficulté, même s'ils ont le même nombre de points. Si vous êtes bloqués sur un exercise, mettez-le à coté pour d'abord travailler sur les autres.

Votre attention La donnée de chaque exercice est précisément formulée, et parfois subtile. Si vous ne la comprenez pas, vous ne pourrez pas en tirer tous les points.

Exercice	**Points**	**Points obtenus**
1	5	
2	10	
3	10	
Total	25	

Exercice 1 : La notation for (5 points)

Dans cet exercice, implémentez une méthode `divisors` qui prend une liste d'entiers en argument, et qui retourne, pour chaque nombre de la liste, tous les nombres de la liste qui le divisent. Utilisez l'opérateur modulo (`%`) pour tester si un entier divise un autre.

La méthode `divisors` a la signature suivante :

```
def divisors(xs: List[Int]): List[(Int, Int)]
```

Elle retourne une liste de tous les couples `(x, y)` tel que

- `x,y` $\in$ `xs`, et
- `x != y`, et
- `x` divise `y`

Vous pouvez assumer que tous les éléments `x` $\in$ `xs` de la liste d'entrée sont positifs, `x > 0`. Donnez deux implémentations de la méthode `divisors` :

1. En utilisant une compréhension **for**
2. En utilisant les méthodes *filter*, *map* et *flatMap*

Exercice 2 : Filtrage de motifs et récursion (10 points)

Partie 1 Run-Length Encoding (5 points)

Le run-length encoding (RLE) est un algorithme simple de compression de données. Il est utile pour comprimer des données contenant des longues séquences d'un certain symbole.

En RLE, chaque pièce de donnée est un couple composé d'un symbole et un d'entier représentant le nombre de fois que ce symbole apparait en séquence. Par exemple, la liste

```
List('a','a','a','a','a','z','z','z','a','c','c','c','c')
```

est représentée en RLE comme

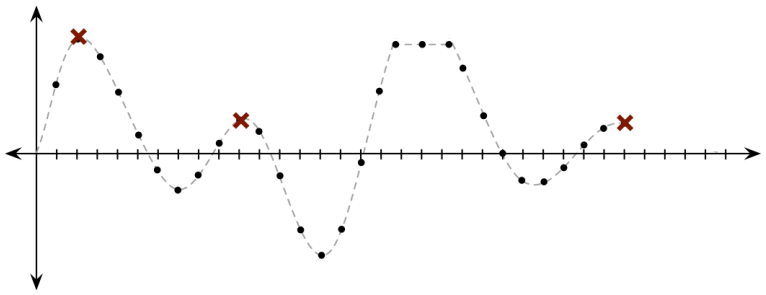
```
List(('a',5),('z',3),('a',1),('c',4))
```

Écrivez une méthode `encode` pour encoder une liste de caractères en utilisant RLE. Le résultat est une liste de paires `(Char, Int)`. La solution doit être récursive et utiliser le filtrage de motifs.

```
def encode(xs: List[Char]): List[(Char, Int)] = /* ??? */
```

Partie 2 Maxima locaux (5 points)

Dans une séquence de nombres, chaque élément qui est strictement plus grand que ces deux voisins est un maximum local. Dans la figure ci-dessous, les maxima sont marqués avec un "X".



- Le premier et le dernier élément sont considérés comme des maxima s'ils sont plus grand que leur voisin unique.
- Si plusieurs éléments égaux se suivent, aucun n'est considéré comme maximum. Par exemple, la liste `List(1, 2, 2, 0, 3, 4)` a un seul maximum local `List(4)`.
- Les listes de taille 1 n'ont pas de maximum.

Écrivez une méthode `findMaxima` qui prend une liste d'entiers en argument, et retourne la liste des maxima locaux. La solution doit être récursive et utiliser le filtrage de motifs.

```
def findMaxima(in: List[Int]): List[Int] = /* ??? */
```

Exercice 3 : Tries (10 points)

Un trie est une structure de données arbre utilisée pour stocker des ensembles de chaînes de caractères.

Introduction L'idée générale est de stocker les chaînes de caractères de mainère efficace en partageant les préfixes communs. Voici deux exemples :

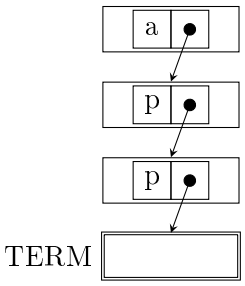


FIGURE 1 – `"app"`

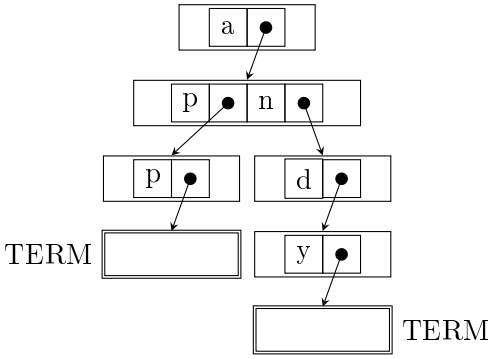


FIGURE 2 – `"app","andy"`

Figure 1 montre un trie contenant un seul mot `"app"`. Il consiste en quatre noeuds. Le premier noeud, appelé la *racine*, associe avec la lettre `'a'` un sous-trie qui représente la chaîne de caractères `"pp"` (le reste du mot `"app"`). Ceci se continue jusqu'à ce qu'on atteint le dernier noeud marqué TERM - ce noeud est appelé un noeud terminal. Un noeud terminal marque que la chaîne de caractère traversée pour y arriver fait partie du trie. Par conséquent, les mots `""`, `"a"` et `"ap"` ne sont pas représentés par ce trie.

Figure 2 montre un trie contenant les mots `"app"` et `"andy"`. Le deuxième noeud a deux enfants, l'un associé à la lettre `'p'`, l'autre associé à `'n'`.

Définition précise La liste suivante donne une définition plus formelle des tries.

- Le premier noeud d'un trie est appelé la *racine*.
- Chaque noeud peut avoir au plus un enfant pour chaque lettre de l'alphabet.
- Chaque noeud est soit terminal, soit non-terminal.
- Une chaîne de caractères est stockée dans un trie si et seulement si un noeud terminal est atteignable en traversant le trie depuis la racine, choisissant le prochain noeud selon le prochain caractère de la chaîne.

Notez que le nombre de chaînes de caractères stocké dans un trie correspond au nombre de noeuds terminaux.

Cas limites Voici quelques exemples en plus pour démontrer des cas limites.

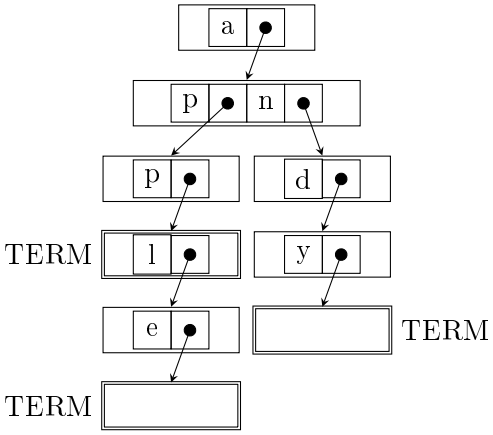


FIGURE 3 – `"app","andy","apple"`

FIGURE 4 – trie vide

FIGURE 5 – trie contenant `""`

- Figure 3 montre un trie contenant les mots `"app"`, `"apple"` et `"andy"`.
- Figure 4 représente un trie vide. Un trie vide ne contient aucune chaîne de caractère.
- Figure 5 montre un trie contenant le string vide (`""`), la chaîne avec aucun caractère. Notez que les deux derniers ne sont pas égaux.

La classe Trie Voici la definition des noeuds d'un trie.

```
case class Trie(isTerminal: Boolean, children: List[(Char, Trie)]) {
  def next(c: Char): Trie = children.find(p => p._1 == c) match {
    case Some(pair) => pair._2
    case None => emptyTrie
  }
  def updateChild(c: Char, t: Trie): Trie = {
    val pos = children.indexWhere(p => p._1 == c)
    val newChildren = if (pos == -1) (c, t) :: children
                      else children.updated(pos, (c, t))
    Trie(isTerminal, newChildren)
  }
}
```

Chaque trie a un membre booléen `isTerminal` qui est vrai si le noeud est terminal, et une liste `children` contenant des couples d'un caractère et le sous-trie correspondant à ce caractère.

La méthode `next` retourne le noeud enfant correspondant à un caractère. S'il y a pas de noeud associé avec ce caractère, un trie vide est retourné. Nous assumons qu'il existe une méthode `emptyTrie` qui crée un trie vide.

Cette implémentation d'un trie est immuable. La méthode `updateChild` retourne une nouvelle version du trie en mettant à jour la liste d'enfants pour le caractère spécifié. S'il n'y a pas d'enfant associé au caractère, un nouveau couple de caractère et sous-trie est ajouté à la liste. Sinon, le noeud enfant associé auparavant est remplacé.

Il y a deux méthodes utilisées sur le type `List[T]` :

- `def indexWhere(pred: T => Boolean): Int`. Par exemple :
  `List("in","on").indexWhere(s => s.contains("o")) ==> 1`.

- `def updated(pos: Int, v: T): List[T]`. Par exemple :
  `List("in","on").updated(1,"un") ==> List("in","un")`.

Votre but est d'écrire des méthodes pour vérifier si une chaîne de caractères fait partie d'un trie. Après, vous allez écrire des méthodes pour construire un trie étant donné une liste de mots.

Partie 1 (1 points)

Écrivez une procédure `emptyTrie` qui retourne un trie vide, un trie contenant aucun mot.

```
def emptyTrie = /* ??? */
```

Partie 2 (2 points)

On définit un nouveau type `Word` pour représenter des mots comme suit :

```
type Word = List[Char]
```

Écrivez une méthode `parseWord` qui vérifie si un mot existe dans un trie donné.

```
def parseWord(dict: Trie, word: Word): Boolean = /* ??? */
```

Partie 3 (2 points)

Une phrase est représentée comme une liste de mots :

```
type Sentence = List[Word]
```

En utilisant la méthode `parseWord` de la partie précédente, écrivez une méthode `parse` qui vérifie si tous les mots d'une phrase sont contenus dans le trie.

```
def parse(dict: Trie, sentence: Sentence): Boolean = /* ??? */
```

Partie 4 (4 points)

Implémentez une méthode `add` qui, étant donné un trie et un mot, produit un nouveau trie contenant le nouveau mot, et tous les mots du trie existant. Sa signature est la suivante :

```
def add(trie: Trie, word: Word): Trie = /* ??? */
```

Partie 5 (1 points)

Utilisez la méthode `add` pour implémenter une méthode `adAll` qui ajoute tous les mots d'une liste donnée à un trie.

```
def addAll(trie: Trie, words: List[Word]): Trie = /* ??? */
```