Examen final

Programmation IV

20 juin 2007

Nom : _____

Prénom : _____

Section : _____

Ne perdez pas de points bêtement, soyez attentifs:

- Une feuille sans nom est une feuille qui ne sera pas prise en compte.
- Les quatre exercices influencent le résultat de la même façon; nous ne pensons toutefois pas qu'ils aient la même difficulté. Choisissez-en judicieusement l'ordre.

Exercice	**Points**	**Points obtenus**
1	20	
2	20	
3	20	
4	20	
Total	80	

Exercice 1 : Compréhensions "**for**" (20 points)

Traduisez chacune des expressions Scala suivantes en une expressions équivalente utilisant des constructions "**for**" à la place de "`map`", "`flatMap`", "`filter`" et `foreach`".

1. `xs.zip(ys).foreach(p => println(p))`

2. ```
   xs.zip(xs.indices)
     .filter(q => ys.contains(q._1)).map(p => p._2)
   ```
   n.b. : pour deux listes "`xs`" et "`ys`", l'expression retourne la liste des indices "`i`" sur "`xs`" tels que "`xs(i)`" soit présent dans "`ys`".

3. ```
   xs.zip(xs.indices)
     .flatMap(p => ys.zip(ys.indices)
       .filter(y => p._1 + y._1 <= 10)
         .map(q => (p._2, q._2)))
   ```
   n.b. : pour deux listes "`xs`" et "`ys`", l'expression retourne la liste des paires "`(i, j)`" d'indice "`i`" sur "`xs`" et "`j`" sur "`ys`" tels que $\texttt{xs}(\texttt{i}) + \texttt{ys}(\texttt{j}) \leq 10$.

Exercice 2 : Flots (20 points)

Vous disposez du flot infini des nombres premiers en ordre croissant dont la signature est la suivante.

```
val primes: Stream[Int]
```

La décomposition d'un nombre n en produit de facteurs premiers consiste à réécrire n sous forme d'un produit de nombres premiers. Par exemple $90 = 2 \cdot 3 \cdot 3 \cdot 5$.

Implantez la fonction "`factors`" qui, pour un nombre n, retourne le flot de nombres premiers tels que le produit de tous les éléments du flot soit égal à n, c'est-à-dire une représentation de sa décomposition en produit de facteurs premiers. `factors` a la signature suivante.

```
def factors(n: Int): Stream[Int] = ...
```

Par exemple, "`factors(90).print`" imprimera "`2, 3, 3, 5, Stream.empty`".

Exercice 3 : Lisp et Prolog (20 points)

Lisp

En Scala, la méthode "`zip`" de la classe "`List`" peut être implantée de la façon suivante.

```
abstract class List[A] {
  def zip[B](that: List[B]): List[(A, B)] =
    if (this.isEmpty || that.isEmpty) Nil
    else (this.head, that.head) :: this.tail.zip(that.tail)
  ...
}
```

Elle retourne une liste de paires "`(x, y)`" tels que "`x`" provient de la liste elle-même ("`this`") et "`y`" est l'élément correspondant (de même indice) de "`that`". Si une des deux listes est plus longue que l'autre, les éléments en plus sont ignorés.

Implantez une fonction équivalente à `zip` en Lisp. Représentez la paire "`(x, y)`" comme "`(cons x y)`". Votre fonction sera basée sur le modèle suivant.

```
(def (zip xs ys) ...)
```

Prolog

Ecrivez un programme Prolog qui implante le prédicat "`reverse`". L'application de "`reverse`" à une liste d'éléments est égal à une liste dont les éléments apparaissent dans l'ordre inverse.
Vous pouvez considérer dans votre solution que tout prédicat Prolog vu au cours est disponible.

Exercice 4 : Décomposition fonctionelle (20 points)

Représentons en Scala les expressions régulières (e.r.) sur des chaînes de caractères par des fonctions. Si "`cs`" a le type "`Chaîne`" et "`er`" est une fonction de type "`ExpRég`" représentant une e.r., les propriétés suivantes sont vérifiées.

$$\texttt{er}(\texttt{cs}) = \begin{cases} \texttt{Some}(\texttt{cs2}) & \text{si } \exists \texttt{cs1} \begin{cases} \texttt{cs} = \texttt{cs1} ::: \texttt{cs2} \wedge \\ \texttt{cs1} \text{ est consommée par l'e.r.} \end{cases} \\ \texttt{None} & \text{dans tous les autres cas} \end{cases}$$

```
type Chaîne = List[Char]
type ExpRég = Chaîne => Option[Chaîne]
```

On peut définir une fonction qui, pour un caractère c donné, retourne la fonction représentant l'e.r. qui consomme le caractère c.

```
def caractère(car: Char)(cs: Chaîne) = cs match {
  case c :: cs1 if c == car => Some(cs1)
  case _ => None
}
```

Par exemple, "`caractère('a')`" définit l'*expression régulière "a"* (notez l'application partielle de la fonction currifiée). L'exécution du code

```
val a = caractère('a'); val b = caractère('b')
println(a("abc".toList))
println(b("abc".toList))
```

imprimera "`Some(List('b', 'c'))`" puis "`None`".

1. Implantez une fonction qui, pour deux e.r. e_1 et e_2 données, retourne l'e.r. *séquence* qui consomme e_1 *puis* e_2 ; elle a la signature suivante.
   `def séq(e1: ExpRég, e2: ExpRég)(ch: Chaîne): Option[Chaîne]`

   Par exemple, "`séq(a, b)`" définit l'e.r. "ab".

2. Implantez ensuite une fonction qui, pour deux e.r. e_1 et e_2 données, retourne l'e.r. *alternative* qui consomme *soit* e_1 *soit* e_2 ; elle a la signature suivante.
   `def alt(e1: ExpRég, e2: ExpRég)(ch: Chaîne): Option[Chaîne]`

   Par exemple, "`alt(a, b)`" définit l'e.r. "$a|b$"

3. Implantez finalement une fonction qui, pour une e.r. e donnée, retourne l'e.r. *répétition* qui consomme e autant de fois que possible ; elle a la signature suivante.
   `def rép(e: ExpRég)(ch: Chaîne): Option[Chaîne]`

   Par exemple, "`rép(a)`" définit l'e.r. "$a*$"