Examen final

Programmation IV

21 juin 2006

Nom : ______________________________

Prénom : ______________________________

Section : ______________________________

Exercice	**Points**	**Points obtenus**
1	15	
2	20	
3	10	
4	25	
5	30	
Total	100	

Exercice 1 : Mots les plus longs (15 points)

Partie 1 (5 pts)

En utilisant la méthode `foldLeft` des listes, écrivez une fonction qui calcule le maximum d'une liste d'entiers non négatifs.

```
def max(xs: List[Int]): Int =
  (xs foldLeft ......) (....)
  % (xs foldLeft 0)((a,b) => if (a > b) a else b)
```

Partie 2 (10 pts)

Ecrivez une fonction qui retourne le mot le plus long d'une liste de mots `xs`. Par convention, le mot le plus long d'une liste vide est la chaîne vide "".

```
def longestWord(ws: List[String]): String =
  % (ws foldLeft "")((a,b) => if (a.length > b.length) a else b)
```

Exercice 2 : Sauce au curry (20 points)

Comme vous savez, SCALA permet la définition de fonctions currifiées.

```
def multiplie(a:Int)(b:Int): Int = a * b
```

Ceci permet de définir de nouvelles fonctions résultantes de l'application partielle de la première.

```
val double: X = multiplie(2)
```

Partie 1 : Typage (3 pts)

On a omis de définir le type de la valeur `double` (noté `X`). Quel est ce type ? Quel est le type de la valeur suivante (noté `Y`) ?

```
val fois: Y = multiplie
type X = % Int => Int (or) Function2[Int, Int]
type Y = % Int => Int => Int (or) Function2[Int, Function2[Int, Int]]
```

Partie 2 : Currification explicite (5 pts)

Définissez en SCALA une fonction `curry2` qui, pour une fonction `f` à deux paramètres donnée, retourne la version currifiée de cette fonction, d'après le modèle suivant (qui retournera 16).

```
def multiplie(x: Int, y: Int): Int = x * y
def curry2[T,U,V](f: (T,U) => V): T => U => V = ...
  % x => (y => f(x,y))
def double = curry2(multiplie)(2)
double(8)
```

Partie 3 : Retour aux sources (5 pts)

Définissez en SCALA une fonction `uncurry2` qui, pour une fonction currifiée `f` à deux arguments donnée, retourne la version non-currifiée de cette fonction, d'après le modèle suivant (qui retournera 16).

```
def uncurry2[T,U,V](f: T => U => V): (T,U) => V = ...
  % (x, y) => f(x)(y)

uncurry2(curry2(multiplie))(2,8)
```

Partie 4 : Currification en LISP (7 pts)

LISP ne dispose pas de syntaxe spécifique pour définir des fonctions currifiées. Essayez quand même de définir en LISP la fonction `multiplie` pour qu'on puisse l'appliquer partiellement, comme dans l'exemple ci-dessous.

```
(val double (multiplie 2) ...)
```

Exercice 3 : Iterateur Fibonacci (10 points)

Ecrivez un itérateur en SCALA qui parcourt la suite de Fibonacci {1,1,2,3,5,...}.

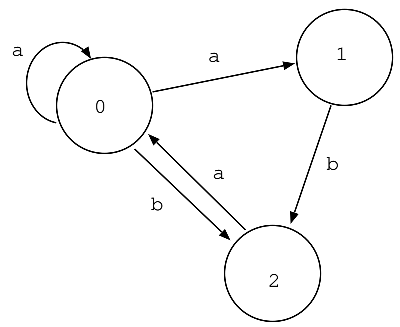
```
class FibIterator extends Iterator[Int] {

   ...



%class FibIterator extends Iterator[Int] {
%  val hasNext = true
%  private var upcomming = 1
%  private var current = 0
%  def next: Int =  {
%    val next = upcomming + current
%    current = upcomming
%    upcomming = next
%    current
%  }
%}
















}
```

Exercice 4 : Automate à états finis (25 points)

Un automate à états finis nondéterministe (AFN) est défini par
- un ensemble fini E d'états $\{0, 1, ..., n\}$
- dont 0 est l'état initial,
- un alphabet `Char` des symboles d'entrée,
- une fonction de transition `delta` qui fait correspondre à tout couple formé d'un état et d'un caractère un ensemble (éventuellement vide) d'états :

$$\texttt{delta(}e_i\texttt{,'a')=List(}e_{i_1},\ldots,e_{i_n}\texttt{)}.$$

Etant donnée la fonction `delta: (Int,Char) => List[Int]`, on peut simuler toutes les l'exécutions possibles d'un AFN pour une chaîne de caractères donnée.



Complétez le programme suivant pour obtenir un algorithme pour une telle simulation. Pour l'automate à droite, `run(List('a','a'), deltaA)` doit contenir les états 0 et 1.

```
def run(inp:List[Char], delta:(Int,Char)=>List[Int]):List[Int] = {

  def next(qs: List[Int], c: Char): List[Int] = ...

  def run1(qs: List[Int], cs: List[Char]): List[Int] =
    cs match {
      case Nil   => ...
      case d::ds => ...
  }
  run1(List(0), inp)
}
```

Partie 1 (10 pts)

Complétez la fonction `next`. Pour une liste `qs` et un caractère `c`, elle retourne une liste avec tous les états atteignables par une transition étiquetée avec `c`.

$$\texttt{next}(qs, c) \stackrel{def}{=} \{p \mid \exists q \in qs \,.\, delta(q, c) \ni p\}$$

Il n'est pas nécessaire d'éliminer les doublons éventuelles dans la liste.

Partie 2 (10 pts)

Complétez la fonction `run1`. Elle traverse la chaîne `cs` caractère par caractère, en avançant l'ensemble d'états chaque fois.

Partie 3 (5 pts)

Décrivez, dans une seule phrase, le rôle des paramètres `qs` et `cs` de `run1`

Exercice 5 : Permutations (30 points)

Une permutation de n objets rangés dans un certain ordre correspond à un changement de l'ordre de succession de ces n objets. Vous devez implanter une fonction (en SCALA) et un prédicat (en PROLOG) `permutation` qui, pour une liste d'éléments tous distincts `as`, calcule la liste de toutes les permutations possibles de `as`.
On peut se servir de la propriété suivante : si `as` s'écrit `b::bs` alors une permutation de `as` est une permutation de `bs` dans laquelle a été inséré à un certain endroit l'élément `b`.

Partie 1 : En SCALA ... (10+10 pts)

Complétez la définition de la fonction récursive `insertion` qui retourne la liste de toutes les listes résultantes de l'insertion d'un élément `x` dans une liste `xs`, comme dans l'exemple suivant.

```
insertion(1, List(2,3,4)) =
  List(List(1,2,3,4), List(2,1,3,4),
       List(2,3,1,4), List(2,3,4,1))
```

Voici le prototype de la fonction `insertion` que vous devez écrire.

```
def insertion[A](x:A, xs:List[A]): List[List[A]] = xs match {
  case Nil     => ...
    % List(List(x))

  case y :: ys => ...
    % (x::y::ys)::(insertion(x,ys) map (zs => y::zs))

}
```

Complétez la définition de la fonction récursive `permutation` qui retourne la liste de toutes les permutations d'une liste `as`, comme dans l'exemple suivant.

```
permutation(List(1,2,3)) =
  List(List(1,2,3), List(2,1,3), List(2,3,1),
       List(1,3,2), List(3,1,2), List(3,2,1))
```

Voici le prototype de la fonction `permutation` que vous devez écrire.

```
def permutation[A](as:List[A]): List[List[A]] = as match {
  case Nil     =>  ...
    % List(Nil)

  case b :: bs =>  ...
    % (permutation(bs)) flatMap (xs => insertion(b,xs))

}
```

Partie 2 : ... et en PROLOG (10 pts)

Définissez un prédicat `insertion(X,XS,ZS)` pour lequel la liste `ZS` est égale à l'insertion de l'élément `X` dans la liste `XS`, comme dans l'exemple suivant.

```
prolog> ?insertion(a, [b,c],ZS).
List(ZS = [a, b, c])
prolog> ?more.
List(ZS = [b, a, c])
prolog> ?more.
List(ZS = [b, c, a])
prolog> ?more.
no
```

Définissez un prédicat `permutation(AS,BS)` pour lequel la liste `BS` est égale à la permutation des éléments de la liste `AS`, comme dans l'exemple suivant.

```
prolog> ?permutation([a, b, c], BS).
List(BS = [a, b, c])
prolog> ?more.
List(BS = [b, a, c])
prolog> ?more.
List(BS = [b, c, a])
prolog> ?more.
List(BS = [a, c, b])
prolog> ?more.
List(BS = [c, a, b])
prolog> ?more.
List(BS = [c, b, a])
prolog> ?more.
no
```