Examen

Programmation IV

17 juin 2004

Nom : _____

Prénom : _____

Exercice	Points	Points obtenus
1	10	
2	15	
3	10	
4	10	
5	10	
6	15	
Total	70	

Exercice 1 : Récursion terminale (10 points)

Voici une fonction `power` qui élève un nombre à une puissance entière :

```
def power(x: double, y: int): double =
  if (y == 0) 1
  else if (y % 2 == 0) power(x * x, y / 2)
  else x * power(x, y - 1);
```

Transformez cette fonction de manière à la rendre récursive terminale.

Indication : définir une fonction auxiliaire dont l'un des paramètres joue le rôle d'accumulateur. Voici son profil :

```
def iter(x: double, y: int, acc: double): double
```

Exercice 2 : Multiplication de grands entiers (15 points)

On décide de représenter les grands entiers positifs au moyen de listes d'entiers SCALA. Chaque élément de la liste est un « chiffre » du grand nombre, en base 10. Les chiffres les moins significatifs apparaissent en *tête* de liste. On suppose de plus que tous les nombres sont normalisés, c.-à-d. qu'ils n'ont aucun zéro inutile.

Ainsi, 0 est représenté par la liste vide `List()`, et 12340 est représenté par la liste `List(0,4,3,2,1)`.

1) Définissez une fonction `shift` qui multiplie un grand entier par 10 (5 points).

```
def shift(x: List[int]): List[int] =
```

2) Définissez une fonction `scale` qui multiplie un grand entier `x` par un chiffre `d` (un entier compris entre 0 et 9) (5 points) :

```
def scale(x: List[int], d: int): List[int] = {
  def iter(x: List[int], d: int, carry: int): List[int] = x match {

  }
  iter(    ,     ,    )
}
```

3) Enfin, définissez une fonction `prod` qui multiplie deux grands entiers. Vous pouvez supposez l'existence d'une fonction `sum` qui additionne deux grands entiers (5 points).

```
def prod(x: List[int], y: List[int]): List[int] =
  y match {

  }
```

Exercice 3 : Les quatre carrés (10 points)

On peut montrer, en théorie des nombres, que tout entier positif est la somme de quatre carrés d'entiers.

Par exemple,

$$\begin{array}{rcl} 0 & = & 0^2 + 0^2 + 0^2 + 0^2 \\ 3 & = & 0^2 + 1^2 + 1^2 + 1^2 \\ 15 & = & 1^2 + 1^2 + 2^2 + 3^2 \\ 88 & = & 0^2 + 4^2 + 6^2 + 6^2 \quad = \quad 2^2 + 2^2 + 4^2 + 8^2 \end{array}$$

Le but de cet exercice est d'écrire une fonction `fourSquares` qui renvoie la liste de toutes les décompositions possibles d'un entier `n` en somme de quatre carrés. Voici son profil :

```
def fourSquares(n: int): List[Tuple4[int, int, int, int]]
```

Le type `Tuple4` représente les tuples à 4 éléments. Rappelons que pour construire un tuple contenant les éléments x_1, x_2, x_3 et x_4, on écrit Tuple4(x_1, x_2, x_3, x_4). Voici quelques exemples d'utilisation de cette fonction :

$fourSquares(0)$	retourne	$List(Tuple4(0,0,0,0))$
$fourSquares(3)$	retourne	$List(Tuple4(0,1,1,1))$
$fourSquares(15)$	retourne	$List(Tuple(1,1,2,3))$
$fourSquares(88)$	retourne	$List(Tuple4(0,4,6,6), Tuple4(2,2,4,8))$

Pour être précis, un tuple $Tuple4(m_1, m_2, m_3, m_4)$ appartient à $fourSquares(n)$ si et seulement si $m_1^2 + m_2^2 + m_3^2 + m_4^2 = n$. De plus, un même tuple modulo permutation ne doit apparaître qu'une seule fois.

Indication : il existe une solution courte utilisant la construction `for`.

Exercice 4 : Lisp (10 points)

1) Écrivez en LISP une fonction `drop` qui prend en argument une liste `l` et un entier `n` et qui renvoie la liste `l` sans ses `n` premiers éléments. Si `n` est plus grand que la longueur de la liste, la fonction doit retourner la liste vide (5 points).

2) Écrivez une version SCALA de la fonction LISP suivante (5 points).

```
(def (f xs)
  (def (g xs p)
    (if (null? xs) (quote ())
      (if (p (car xs)) (cons (car xs) (g (cdr xs) p))
        (g (cdr xs) p))))
  (g xs (lambda (x) (>= x 0))))
```

Indication : vous pouvez utiliser les fonctions de la bibliothèque standard de SCALA.

Exercice 5 : Prolog (10 points)

On choisit de représenter les entiers en Prolog en utilisant un constructeur `zero` d'arité 0 pour zéro, et un constructeur `succ` d'arité 1 pour le successeur d'un entier donné.

Par exemple, l'entier 2 est représenté par le terme `succ(succ(zero))` en PROLOG.

1) Définissez en PROLOG un prédicat `plus` qui représente l'addition sur les entiers (5 points).

2) Définissez l'analogue en PROLOG de la fonction SCALA `sum` suivante (5 points) :

```
def sum(l: List[int]): int = l match {
  case Nil => 0
  case x :: xs => x + sum(xs)
}
```

Exercice 6 : Entiers inductifs (15 points)

Voici une définition inductive des entiers :

```
trait Nat {
  def + (n: Nat): Nat = match {
    case Z    => n
    case S(m) => S(m + n)
  }
  def * (n: Nat): Nat = match {
    case Z => Z
    case S(m) => (m * n) + n
  }
}
case object Z extends Nat;
case class S(n: Nat) extends Nat;
```

Il y a une constante (`Z`) pour représenter l'entier 0, un constructeur unaire (`S`) pour construire le successeur d'un entier donné, et les définitions de l'addition (`+`) et de la mutliplication (`*`) entre deux entiers.

Étant données ces définitions, démontrez, en justifiant chaque étape, les deux propriétés suivantes :

$$\begin{array}{lllcll} 1) & \forall m, n : Nat, & m + n & = & n + m & \text{(6 points)} \\ 2) & \forall m, n, p : Nat, & m * (n + p) & = & (m * n) + (m * p) & \text{(9 points)} \end{array}$$

Indications

Vous pouvez utiliser, sans les redémontrer, les propriétés suivantes :

$$\begin{array}{lllcl} (P1) & \forall m, n, p : Nat, & (m + n) + p & = & m + (n + p) \\ (P2) & \forall n : Nat, & n + Z & = & n \\ (P3) & \forall m, n : Nat, & m + S(n) & = & S(m + n) \end{array}$$

Rappelez-vous aussi que `m + n` est juste du sucre syntaxique pour l'expression `m.+(n)`.