Examen final

Programmation IV

17 juin 2003

Nom : _____

Prénom : _____

Section : _____

Exercice	Points	Points obtenus
1	5	
2	10	
3	15	
4	10	
5	15	
6	15	
7	15	
8	15	
Total	100	

Exercice 1 : Suppression d'un élément d'une liste (5 points)

Le but de la fonction `removeFirst` ci-dessous est de supprimer la première occurrence d'un élément dans une liste. Voici un exemple d'utilisation :

```
> removeFirst(List(7,9,2,1,5,2,1,8), 9)
List(7,2,1,5,2,1,8): scala.List[scala.Int]
> removeFirst(List(7,9,2,1,5,2,1,8), 1)
List(7,9,2,5,2,1,8): scala.List[scala.Int]
```

Complétez la définition suivante aux *deux* endroits indiqués par des points d'interrogation :

```
def removeFirst[a](lst: List[a], elem: a): List[a] =
  lst match {
    case Nil =>
      ???









    case head :: tail =>
      ???









  }
```

Exercice 2 : Multi-ensembles (10 points)

Un *multi-ensemble* de valeurs est un ensemble dans lequel une même valeur peut apparaître plusieurs fois, contrairement aux ensembles classiques.

On décide de représenter les multi-ensembles en SCALA au moyen de listes. Par exemple, le multi-ensemble {1, 2, 2, 3} peut être représenté par n'importe quelle permutation de la liste `List(1,2,2,3)`.

La fonction `intersectM` ci-dessous calcule l'intersection de deux multi-ensembles. Par exemple, pour les multi-ensembles {1, 2, 2, 3, 3, 3} et {2, 3, 3, 5}, cette fonction doit retourner le multi-ensemble {2, 3, 3}. Complétez sa définition aux *deux* endroits indiqués par des points d'interrogation.

Indication : vous pouvez utiliser la fonction `removeFirst` de l'exercice précédent.

```
def intersectM[a](m1: List[a], m2: List[a]): List[a] =
  m1 match {
    case Nil => Nil
    case head :: tail =>
      if (m2 contains head) {
        ???




      } else {
        ???




      }
  }
```

Exercice 3 : LISP (15 points)

Soit le programme LISP suivant :

```
(def (f x)
     (if (= (car x) (quote num))
         (car (cdr x))
       (if (= (car x) (quote plus))
           (+ (f (car (cdr x))) (f (car (cdr (cdr x)))))
         (error)))
     (f (quote (plus (num 6) (plus (num 4) (num 3))))))
```

dans lequel la fonction `error` signale une erreur à l'utilisateur.

Question 1 [5 points] Que fait ce programme ?

Question 2 [10 points] Traduisez-le en SCALA en remplaçant les listes utilisées dans la version LISP par des classes cas (*case classes*).

Exercice 4 : Graphes acycliques dirigés (10 points)

Un *graphe acyclique dirigé* est composé d'un ensemble de *sommets* et d'un ensemble d'*arêtes* dirigées liant un sommet source à un sommet destination. Un *chemin* dans un tel graphe est une suite d'au moins deux sommets telle qu'il existe une arête allant de chaque sommet du chemin à celui qui le suit. Le fait qu'un tel graphe soit acyclique signifie qu'il n'existe pas de chemin incluant plus d'une fois un sommet donné. En particulier, *il n'existe pas de chemin allant d'un sommet à lui-même*.

On représente de tels graphes en PROLOG au moyen du prédicat `edge` qui déclare l'existence d'une arête entre deux sommets. Par exemple, les déclarations suivantes :

```
edge(lausanne, yverdon).
edge(yverdon, neuchatel).
edge(neuchatel, bienne).
```

définissent un graphe composé des sommets `lausanne`, `yverdon`, `bienne` et `neuchatel`, avec une arête de Lausanne à Yverdon, une de Yverdon à Neuchâtel et une de Neuchâtel à Bienne. On peut imaginer qu'un tel graphe représente des lignes de chemin de fer.

Question 1 [5 points] Définissez le prédicat `pathExists` qui teste s'il existe un chemin allant d'un sommet à un autre. Il devrait pouvoir être utilisé ainsi :

```
prolog> ?pathExists(lausanne,bienne).
yes
prolog> ?pathExists(bienne,lausanne).
no
prolog> ?pathExists(yverdon,yverdon).
no
```

Question 2 [5 points] Définissez le prédicat `path` qui teste si un chemin, représenté sous forme de liste de sommets, lie deux sommets. Il devrait pouvoir être utilisé ainsi :

```
prolog> ?path(yverdon,bienne,P).
List(P = cons(yverdon,cons(neuchatel,cons(bienne,nil))))
prolog> ?path(bienne,lausanne,P).
no
```

Exercice 5 : Cavalier d'échecs (15 points)

Aux échecs, le cavalier se déplace « en L », c'est-à-dire qu'à partir d'une case donnée il peut aller à n'importe quelle case atteignable en bougeant de deux positions dans une direction et une position dans une direction perpendiculaire. La figure 1 montre toutes les cases atteignables en un coup par un cavalier aux échecs : il y en a huit, indiquées par une croix.

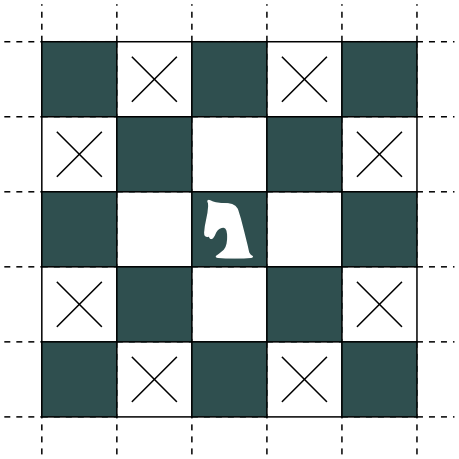

FIG. 1 – Cases atteignables en un coup par un cavalier aux échecs

On désire calculer l'ensembles des cases atteignables par un cavalier en au plus n coups, pour une taille d'échiquier et une position initiale donnés. On décide de représenter les positions sur l'échiquier par des paires d'entiers : la première composante donne la ligne, la seconde la colonne. Les lignes et les colonnes sont numérotées à partir de 0.

Complétez la solution partielle de la page suivante en écrivant le corps de la fonction `reachable`. Cette fonction retourne l'ensemble des positions atteignables par un cavalier en au plus un nombre de coups donné, en fonction de la taille de l'échiquier, qui est carré, et de la position initiale du cavalier. La liste des positions retournée par cette fonction peut contenir des doublons si une position est atteignable de plusieurs manières.

Indication : une utilisation judicieuse de la construction `for` de SCALA peut simplifier la solution.

```
type Position = Pair[int, int];

def row(p: Position) = p._1;
def col(p: Position) = p._2;

def legalPos(boardSize: int)(pos: Position): boolean =
  row(pos) >= 0 && row(pos) < boardSize
  && col(pos) >= 0 && col(pos) < boardSize;

def nextPos(pos: Position): List[Position] = {
  val r = row(pos), c = col(pos);
  List(Pair(r + 2, c + 1),
       Pair(r + 2, c - 1),
       Pair(r - 2, c + 1),
       Pair(r - 2, c - 1),
       Pair(r + 1, c + 2),
       Pair(r + 1, c - 2),
       Pair(r - 1, c + 2),
       Pair(r - 1, c - 2))
}

def reachable(boardSize: int)(initial: Position, moves: int)
  : List[Position] =
```

Exercice 6 : Preuve par induction (15 points)

On représente les listes d'entiers au moyen des classes suivantes :

```
abstract class IntList {
  def map(f: int => int): IntList;
}

case class IntNil() extends IntList {
  def map(f: int => int): IntList = IntNil();
}

case class IntCons(head: int, tail: IntList)
  extends IntList {
    def map(f: int => int): IntList =
      IntCons(f(head), tail map f);
  }
```

Montrez par induction qu'on a l'équivalence suivante, pour toute liste d'entiers l et toutes fonctions f et g :

$$(l \text{ map } g) \text{ map } f = l \text{ map } \{ \, x \Rightarrow f(g(x)) \, \}$$

Exercice 7 : XML (15 points)

On représente les documents XML en SCALA au moyen des classes suivantes :

```
abstract class XML;
case class Tagged(tag: String, contents: List[XML])
  extends XML;
case class Text(contents: String) extends XML;
```

La classe `Tagged` est utilisée pour représenter des parties entourées par une étiquette, alors que la classe `Text` est utilisée pour représenter le texte pur. Par exemple, l'expression SCALA suivante produit un objet décrivant une personne nommée Marie Dupond :

```
Tagged("person", List(Tagged("fname", List(Text("Marie"))),
                      Tagged("lname", List(Text("Dupond")))))
```

(Note : pour les personnes qui connaissent XML, cette expression SCALA correspond au document XML suivant :

```
<person><fname>Marie</fname><lname>Dupond</lname></person>
```

mais savoir cela n'est pas indispensable pour résoudre l'exercice.)

Écrivez une fonction `flatten` qui, étant donné un objet de type `XML`, retourne sous forme de chaîne de caractères le texte contenu dans cet objet. Pour l'exemple ci-dessus, cette fonction retourne donc la chaîne `MarieDupond`.

Exercice 8 : Flot infini (15 points)

Écrivez une fonction qui retourne le flot infini de tous les couples d'entiers positifs ordonnés selon le schéma de la figure 2. Ce flot commence donc ainsi : $(0,0)$, $(1,0)$, $(0,1)$, $(2,0)$, $(1,1)$, $(0,2)$, $(3,0)$, $(2,1)$, $(1,2)$, $(0,3)$, ...

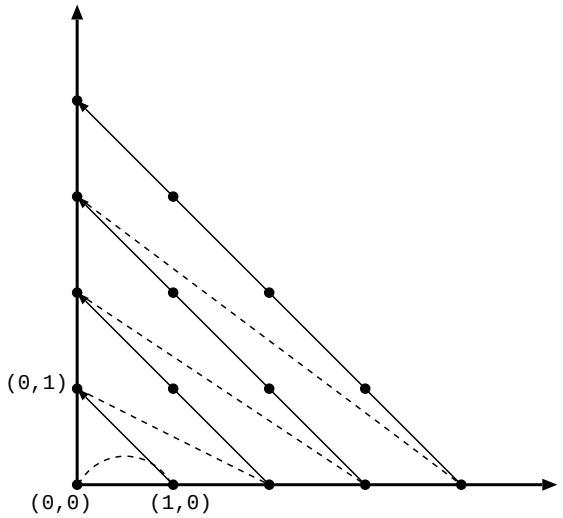

FIG. 2 – Énumération des couples d'entiers