Exercice 1 : Récursion terminale

La fonction récursive suivante n'est pas *récursive terminale* :

```
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);
```

En effet, l'appel récursif `power(x, y - 1)` ne se situe en *position terminale* de la fonction. Il s'en suit que chaque appel récursif de la fonction va faire grandir la pile à l'exécution, ce qui peut entraîner un dépassement de mémoire (stack overflow) pour de grandes valeurs initiales du paramètre `y`. Le but de cette exercice est de rendre la fonction récursive terminale.

La manière standard de transformer une fonction récursive pour la rendre récursive terminale est de lui ajouter un paramètre. Celui-ci va jouer un rôle d'accumulateur en *récoltant* les résultats intermédiaires obtenus à chaque itération. Avec la fonction `power`, cela donne le code suivant :

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

Comme on le voit, on utilise une fonction récursive auxiliaire `iter` qui reprend le code de la fonction initiale `power` avec les différences suivantes :

1. elle a un paramètre additionnel `acc` pour accumuler les résultats intermédiaires.
2. dans le cas de base (`y == 0`), elle renvoie l'accumulateur.
3. l'appel récursif `power(x * x, y / 2)` devient `iter(x * x, y /2, acc)`. En effet, cet appel était déjà en position terminale de la fonction, dans ces cas-là, il suffit de passer tel quel l'accumulateur.
4. l'expression `x * power(x, y - 1)` devient `iter(x, y - 1, x * acc)`. Maintenant l'appel récursif se trouve bien en position terminale, et l'action de multiplier par `x` se répercute sur l'accumulateur.

Pour terminer, on a englobé la fonction auxiliaire `iter` dans une fonction `power` avec la même signature que la fonction originale. Le résultat de cette nouvelle fonction est alors simplement un appel à la fonction `iter` avec une valeur d'accumulateur initialisée à `1`.

Exercice 2 : Multiplication de grands entiers

1) Pour multiplier par dix un grand nombre en base dix, il suffit de rajouter un zéro en tête de la liste, de manière à décaler tous les chiffres d'un rang. Mais il faut traiter différemment le cas du grand nombre 0, représenté par la liste vide, sinon l'on obtient le grand nombre `List(0)` qui n'est pas normalisé.

```
def shift(x: List[int]): List[int] = x match {
  case Nil => Nil
  case d :: ds => 0 :: x
}
```

2) Pour multiplier un grand nombre par un chiffre `d` compris entre 0 et 9, on multiplie chacun des nombres du grand chiffre par `d` et on se sert d'une retenue pour reporter sur le chiffre suivant l'excédent issu de chaque multiplication. On définit une fonction récursive auxiliaire `iter` pour itérer sur tous les chiffres du grand nombre :

```
def scale(x: List[int], d: int): List[int] ={
  def iter(x: List[int], d: int, carry: int): List[int] = x match {
    case Nil => if (carry == 0) Nil else List(carry)
    case d1 :: rest => {
      val p = d * d1 + carry;
      (p % 10) :: iter(rest, d, p / 10)
    }
  }
  if (d == 0) Nil else iter(x, d, 0)
}
```

Comme on le voit, cette fonction emprunte l'idée de la retenue à la fonction définissant l'addition entre grands nombres. Mais elle est en fait plus simple car il n'est plus nécessaire d'itérer sur deux nombres en même temps. Il faut juste faire attention de ne pas retourner un résultat qui ne soit pas normalisé.

3) Il ne reste plus maintenant qu'à recoller les morceaux pour obtenir la multiplication entre grands nombres. La solution se base sur les deux propriétés suivantes de la multiplication :

$$\begin{aligned} x * 0 &= 0 \\ x * (ds * 10 + d) &= (x * d) + (x * ds) * 10 \end{aligned}$$

dans lesquelles x et ds sont des entiers, et d un chiffre.

En utilisant la fonction d'addition `sum`, ainsi que les fonctions déjà définies dans les questions précédentes, on obtient la définition suivante :

```
def prod(x: List[int], y: List[int]): List[int] =
  y match {
    case Nil => Nil
    case d :: ds => sum(scale(x, d), shift(prod(x, ds)))
  }
```

Exercice 3 : Les quatre carrés

Si l'on adopte la notation mathématique, on peut définir ainsi l'ensemble des solutions au problème des quatre carrés :

$$fourSquares(n) = \{(i_1, i_2, i_3, i_4) \in \mathbb{N}^4 \quad | \quad i_1^2 + i_2^2 + i_3^2 + i_4^2 = n\}$$

Autrement dit, $fourSquares(n)$ est l'ensemble des tuples de quatre entiers dont la somme des carrés est égale à n.

En fait, aucun entier plus grand que n ne peut apparaître dans un tuple solution. On obtient donc la formulation équivalente suivante pour chaque ensemble $fourSquares(n)$:

$$fourSquares(n) = \{(i_1, i_2, i_3, i_4) \in [0, n]^4 \quad | \quad i_1^2 + i_2^2 + i_3^2 + i_4^2 = n\}$$

En mathématique, on dit que l'on a défini l'ensemble $fourSquares(n)$ par *compréhension*, c'est-à-dire qu'on l'a défini comme sous-ensemble d'un autre ensemble ($[0, n]^4$), dont tous les éléments vérifient une certaine proprpriété ($i_1^2 + i_2^2 + i_3^2 + i_4^2 = n$).

La notation `for` de SCALA permet d'exprimer élégamment les ensembles ayant une définition par compréhension.

```
def fourSquares(n: int): List[Tuple4[int, int, int, int]] =
  for(
    val i1 <- List.range(0, n + 1);
    val i2 <- List.range(0, n + 1);
    val i3 <- List.range(0, n + 1);
    val i4 <- List.range(0, n + 1);
    square(i1) + square(i2) + square(i3) + square(i4) == n
  ) yield Tuple4(i1, i2, i3, i4);

def square(i: int): int = i * i;
```

Comme on le voit, la notation SCALA est très proche de la notation mathématique.

Il faut maintenant remarquer que si un tuple est une solution du problème des quatre carrés, alors n'importe quelle permutation de ce tuple est aussi une solution du problème. Pour simplifier le résultat, on choisit donc de ne prendre en compte que le *représentant trié* de chaque classe de permutations :

```
def fourSquares(n: int): List[Tuple4[int, int, int, int]] =
  for(
    val i1 <- List.range(0, n + 1);
    val i2 <- List.range(i1, n + 1);
    val i3 <- List.range(i2, n + 1);
    val i4 <- List.range(i3, n + 1);
    n == square(i1) + square(i2) + square(i3) + square(i4)
  ) yield Tuple4(i1, i2, i3, i4);
```

Maintenant, toutes les solutions générées ont bien la propriété d'être ordonnées, c'est-à-dire qu'on a toujours $i_1 \leq i_2 \leq i_3 \leq i_4$.

Exercice 4 : Lisp

1) La fonction `drop` permet de retirer les `n` premiers éléments d'une liste. Si celle-ci contient moins de `n` éléments, `drop` retourne la liste vide.

Commencons par une formulation de cette fonction en SCALA :

```
def drop[A](l: List[A], n: int): List[A] =
  if (n == 0)
    l
  else l match {
    case Nil => Nil
    case x :: xs => drop(xs, n - 1)
  }
```

Si n est égal à zéro, cela signifie qu'on ne doit retirer aucun élément de la liste, on renvoie donc la liste elle-même. Sinon, dans le cas où la liste est vide, il n'y a plus rien à retirer donc on renvoie simplement la liste vide. Dans le dernier cas, où la liste n'est pas vide et où le nombre d'éléments n à retirer n'est pas vide, on obtient le résultat en retirant $n - 1$ éléments à la queue de la liste.

Voici maintenant la traduction de cette fonction en LISP :

```
(def (drop l n)
  (if (= n 0) l
      (if (null? l) (quote ()) (drop (cdr l) (- n 1)))))
```

La seule différence significative entre les deux formulations (mise à part la syntaxe) est la façon de tester si une liste est vide et de la décomposer en deux parties, sa tête et sa queue, si elle ne l'est pas. En SCALA, le filtrage de motif nous permet de réaliser ces deux opérations en même temps. En LISP, on doit d'abord tester si la liste est vide à l'aide de la fonction `null?`, puis utiliser les fonctions prédéfinies `car` et `cdr` pour accéder respectivement à la tête et à la queue de la liste.

2) Voici la traduction en SCALA de la fonction LISP `f`.

```
def f(xs: List[int]): List[int] = {
  def g(xs: List[int], p: int => boolean): List[Int] =
    xs match {
      case Nil => Nil
      case y :: ys => if (p(y)) y :: g(ys, p) else g(ys, p)
    }
  g(xs, x => 0 <= x)
}
```

Le corps de la fonction `f` contient une définition de fonction imbriquée `g`. Par rapport à la version LISP, on a utilisé en SCALA le filtrage de motif qui permet de déconstruire élégamment une liste en la partie constituée de sa tête et la partie constituée de sa queue. Sinon, seule la syntaxe diffère, avec notamment l'utilisation de la forme spéciale `lamdba` en LISP pour représenter les fonctions anonymes.

On aurait aussi pu remarquer que la fonction `g` ne fait rien de plus que la méthode `filter` dans la classe `List`. Une solution valide aurait alors été tout simplement :

```
def f(xs: List[int]): List[int] = xs filter (0 <=);
```

Exercice 5 : Prolog

1) En PROLOG, on représente une fonction par un prédicat avec un paramètre additionnel pour le résultat de la fonction.

Il s'agit ici de représenter en PROLOG la fonction qui prend en arguments deux entiers et qui renvoie leur somme. On doit donc avoir un prédicat `plus` à trois places, deux pour les arguments, et un pour le résultat.

Par exemple, voici le résultat de la requête correspondant à l'addition des entiers 3 et 2 :

```
prolog> ? plus(succ(succ(succ(zero))), succ(succ(zero)), S).
List(S = succ(succ(succ(succ(succ(zero))))))
```

Pour définir ce prédicat, on raisonne par cas sur le premier argument. Si c'est l'entier `zero`, alors la somme est égale au deuxième argument :

```
plus(zero, N, N).
```

Si c'est le successeur d'un entier `M`, alors la somme est égale au successeur de l'addition de `M` avec le deuxième argument :

```
plus(succ(M), N, succ(P)) :- plus(M, N, P).
```

Sans le savoir on a aussi défini la soustraction entre entiers. En effet, en fixant la valeur du premier et du dernier argument du prédicat dans une requête, on obtient la différence des deux comme solution pour le deuxième argument. Par exemple, pour obtenir la différence entre 3 et 1, on peut faire la requête suivante :

```
prolog> ? plus(succ(zero), D, succ(succ(succ(zero)))).
List(D = succ(succ(zero)))
```

2) Pour effectuer la somme d'une liste d'entiers en PROLOG, on peut partir de la définition SCALA suivante :

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

et la traduire en PROLOG :

```
sum(nil, zero).
sum(cons(X, Xs), S) :- sum(Xs, S1), plus(X, S1, S).
```

Les deux branches du filtrage de liste SCALA correspondent en PROLOG à deux règles définissant le prédicat `sum`. Le membre gauche de la première règle s'unifiera uniquement avec les requêtes `sum` dont le premier argument est une liste vide. La deuxième règle traite le cas des listes non vides. Sinon, la logique intrinsèque est très similaire. Remarquez toutefois qu'en PROLOG il est nécessaire de représenter explicitement le résultat de l'appel d'une fonction au moyen de variables auxiliaires (`S1` et `S`). En SCALA cela correspondrait à nommer des résultats intermédiaires au moyen de variables locales.

Exercice 6 : Entiers inductifs

1) La première propriété à montrer est la commutativité de l'opérateur + :

$$\forall m, n : Nat, \quad m + n \quad = \quad n + m$$

On fait la preuve par induction structurelle sur m.

Cas [$m = Z$].
À gauche, on a :

	$m + n$	
=	$Z + n$	remplacement de m par sa valeur
=	n	définition de +, premier cas

À droite, on a :

	$n + m$	
=	$n + Z$	remplacement de m par sa valeur
=	n	propriété (P2)

Cas [$m = S(m')$].
On dispose de l'hypothèse d'induction suivante :

$$\forall n : Nat, \quad m' + n \quad = \quad n + m'$$

À gauche, on a :

	$m + n$	
=	$S(m') + n$	remplacement de m par sa valeur
=	$S(m' + n)$	définition de +, deuxième cas
=	$S(n + m')$	hypothèse d'induction

À droite, on a :

	$n + m$	
=	$n + S(m')$	remplacement de m par sa valeur
=	$S(n + m')$	propriété (P3)

2) La deuxième propriété à montrer est la distributivité de la multiplication sur l'addition :

$$\forall m, n, p : Nat, \quad m * (n + p) \quad = \quad (m * n) + (m * p)$$

On fait la preuve par induction structurelle sur m.

Cas [$m = Z$].
À gauche, on a :

	$m * (n + p)$	
=	$Z * (n + p)$	remplacement de m par sa valeur
=	Z	définition de *, premier cas

À droite, on a :

	$(m * n) + (m * p)$	
=	$(Z * n) + (Z * p)$	remplacement de m par sa valeur
=	$Z + (Z * p)$	définition de *, premier cas
=	$Z + Z$	définition de *, premier cas
=	Z	définition de +, premier cas

Cas [$m = S(m')$].

On dispose de l'hypothèse d'induction suivante :

$$\forall n, p : Nat, \quad m' * (n + p) \quad = \quad (m' * n) + (m' * p)$$

À gauche, on a :

	$m * (n + p)$	
=	$S(m') * (n + p)$	remplacement de m par sa valeur
=	$(m' * (n + p)) + (n + p)$	définition de *, deuxième cas
=	$((m' * n) + (m' * p)) + (n + p)$	hypothèse d'induction

À droite, on a :

	$(m * n) + (m * p)$	
=	$(S(m') * n) + (S(m') * p)$	remplacement de m par sa valeur
=	$((m' * n) + n) + (S(m') * p)$	définition de *, deuxième cas
=	$((m' * n) + n) + ((m' * p) + p)$	définition de *, deuxième cas
=	$(m' * n) + (n + ((m' * p) + p))$	propriété (P1)
=	$(m' * n) + (n + (p + (m' * p)))$	question 6.1
=	$(m' * n) + ((n + p) + (m' * p))$	propriété (P1)
=	$(m' * n) + ((m' * p) + (n + p))$	question 6.1
=	$((m' * n) + (m' * p)) + (n + p)$	propriété (P1)

Il aurait en fait été suffisant de dire que grâce à la commutativité et à l'associativité de l'addition, on peut permuter et regrouper les opérandes d'une imbrication de + comme on veut.