Programmation fonctionnelle

Examen

Licence 3 Informatique 2020–2021

jeudi 27 mai 2021

♥ ♣ ♠ ♦

Tout d'abord, ne pas prendre peur à cause de la longueur apparente du sujet : il contient beaucoup d'exemples et d'explications.

Ce devoir est composé de deux exercices indépendants. Les questions au sein des exercices sont elles-mêmes très souvent indépendantes. Il est donc conseillé de ne pas rester bloqué sur une question.

Il est demandé dans certaines questions d'utiliser des fonctions d'ordre supérieur ou de réutiliser des fonctions précédentes : cela est préférable pour avoir la meilleure évaluation mais n'est pas obligatoire.

Les étudiants qui disposent d'un **tiers temps** sont exemptés des questions qui commencent par le symbole (TT).

Le travail consiste à remplir le fichier `Squelette.ml` qui contient un squelette à compléter pour regrouper les réponses des deux exercices.

♥ ♣ ♠ ♦

Exercice 1. (**Arbres unaires binaires**)

Un *arbre unaire binaire* est défini récursivement comme étant

— soit une feuille ;

— soit un nœud unaire par où est attaché un arbre unaire binaire ;

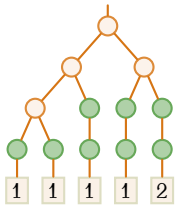
— soit un nœud binaire par où sont attachés deux arbres unaires binaires.

Les feuilles portent des valeurs de nature quelconque. Nous pouvons réaliser ces objets par le type paramétré

```
type 'v arbre =
   |Feuille of 'v
   |Unaire of 'v arbre
   |Binaire of 'v arbre * 'v arbre
```

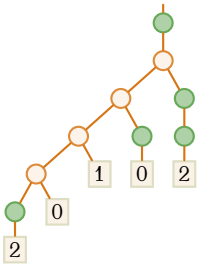
Voici à titre d'exemple deux arbres unaires binaires avec leur codage (les nœuds unaires sont dessinés par des cercles sombres ●, les nœuds binaires par des cercles clairs ○ et les feuilles par des carrés [v] où v est la valeur portée par la feuille) :

$a_1 :=$




```
# let a1 =
  Binaire (
      Binaire (
          Binaire (Unaire (Feuille 1), Unaire (Feuille 1)),
          Unaire (Unaire (Feuille 1))),
      Binaire (
          Unaire (Unaire (Feuille 1)),
          Unaire (Unaire (Feuille 2))));;
```

$a_2 :=$




```
# let a2 =
  Unaire (
      Binaire (
          Binaire (
              Binaire (Binaire (Unaire (Feuille 2), Feuille 0),
                  Feuille 1),
              Unaire (Feuille 0)),
          Unaire (Unaire (Feuille 2))));;
```

Nous ferons référence à ces arbres dans les exemples suivants respectivement par `a1` et par `a2`.

Les objectifs de cet exercice sont d'écrire un générateur exhaustif et des fonctions de factorisation et de développement (dans un sens donné dans la suite) de tels arbres.

Q1. Écrire une fonction

```
val decomposer_entier : int -> (int * int) list = <fun>
```

paramétrée par un entier `n` et qui renvoie la liste de tous les couples d'entiers positifs[1] distincts dont la somme est `n`. L'ordre des éléments dans la liste n'est pas important.

```
# decomposer_entier 0;;
- : (int * int) list = [(0, 0)]
# decomposer_entier 1;;
- : (int * int) list = [(0, 1); (1, 0)]
# decomposer_entier 4;;
- : (int * int) list = [(0, 4); (1, 3); (2, 2); (3, 1); (4, 0)]
```

Q2. Écrire, **en utilisant une fonction d'ordre supérieur**, une fonction

1. Donc, positifs ou nuls.

```
val ajouter_noeud_unaire : 'a arbre list -> 'a arbre list = <fun>
```

paramétrée par une liste d'arbres unaires binaires `arbres` et qui renvoie la liste des arbres obtenus en ajoutant à chaque arbre de `arbres` une racine unaire.

```
# ajouter_noeud_unaire
    [Feuille 1; Unaire (Feuille 2); Binaire (Feuille 1, Unaire (Feuille 4))];;
- : int arbre list =
    [
       Unaire (Feuille 1);
       Unaire (Unaire (Feuille 2));
       Unaire (Binaire (Feuille 1, Unaire (Feuille 4)))
    ]
```

Q3. Écrire une fonction

```
val ajouter_noeud_binaire : 'a arbre list -> 'a arbre list -> 'a arbre list = <fun>
```

paramétrée par deux listes d'arbres unaires binaires `arbres1` et `arbres2` et qui renvoie la liste des arbres dont la racine est binaire et le sous-arbre gauche est un arbre d'`arbre1` et le sous-arbre droit est un arbre d'`arbre2`. Cette fonction considère bien toutes les possibilités. Ainsi, le nombre d'arbres dans le résultat est égal aux nombre d'arbres de la liste `arbre1` multiplié par le nombre d'arbres de la liste `arbre2`.

```
# ajouter_noeud_binaire
    [Feuille 1; Feuille 2]
    [Unaire (Feuille 2); Binaire (Feuille 0, Feuille 1)];;
- : int arbre list =
    [
       Binaire (Feuille 1, Unaire (Feuille 2));
       Binaire (Feuille 1, Binaire (Feuille 0, Feuille 1));
       Binaire (Feuille 2, Unaire (Feuille 2));
       Binaire (Feuille 2, Binaire (Feuille 0, Feuille 1))
    ]
```

Q4. La *taille* d'un arbre unaire binaire est son nombre de nœuds unaires additionné de son nombre de nœuds binaires. Écrire une fonction

```
val taille : 'a arbre -> int = <fun>
```

paramétrée par un arbre unaire binaire `arb` et qui renvoie sa taille.

```
# taille (Feuille 512);;
- : int = 0
# taille a1;;
- : int = 12
# taille a2;;
- : int = 9
```

Q5. (TT) Écrire une fonction

```
val generer_arbres : 'a -> int -> 'a arbre list = <fun>
```

paramétrée par une valeur `v` de type quelconque et un entier `n` et qui renvoie la liste des arbres unaires binaires de taille `n` et dont les feuilles portent toutes la valeur `v`.

Nous utiliserons pour cela l'algorithme récursif suivant.

(a) Si n = 0, la liste qui contient uniquement la feuille portant `v` est renvoyée ;

(b) sinon,

i. créer une nouvelle liste `lst1` contenant les arbres unaires binaires dont la racine est unaire et ayant comme fils un arbre unaire binaire de taille $n - 1$;

ii. créer une nouvelle liste `lst2` contenant les arbres unaires binaires dont la racine est binaire et dont le sous-arbre gauche est de taille i, le sous-arbre droit de taille j, où (i, j) est une décomposition de $n - 1$. Cette étape doit bien considérer toutes les décompositions.

(c) Renvoyer la concaténation de `lst1` et `lst2`.

```
# generer_arbres 0 1;;
- : int arbre list = [Unaire (Feuille 0); Binaire (Feuille 0, Feuille 0)]
# generer_arbres 0 2;;
- : int arbre list =
  [
     Unaire (Unaire (Feuille 0));
     Unaire (Binaire (Feuille 0, Feuille 0));
     Binaire (Feuille 0, Unaire (Feuille 0));
     Binaire (Feuille 0, Binaire (Feuille 0, Feuille 0));
     Binaire (Unaire (Feuille 0), Feuille 0);
     Binaire (Binaire (Feuille 0, Feuille 0), Feuille 0)
  ]
```

Q6. Écrire une fonction

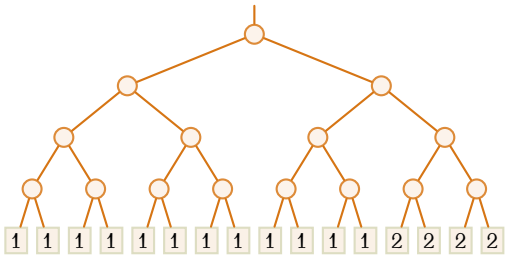
```
val developper : 'a arbre -> 'a arbre = <fun>
```

paramétrée par un arbre unaire binaire `arb` et qui renvoie l'arbre unaire binaire obtenu récursivement en remplaçant chaque nœud unaire de `arb` par un nœud binaire dont les sous-arbres gauche et droit sont les mêmes et sont égaux à celui qui figure comme fils du nœud unaire.

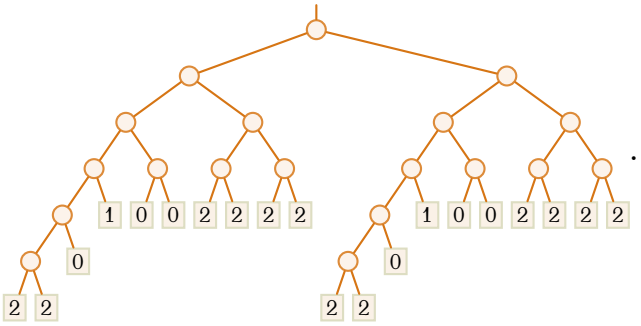
```
# developper a1;;
- : int arbre =
 Binaire
  (Binaire
    (Binaire (Binaire (Feuille 1, Feuille 1), Binaire (Feuille 1, Feuille 1)),
     Binaire (Binaire (Feuille 1, Feuille 1), Binaire (Feuille 1, Feuille 1))),
```

```
  Binaire
   (Binaire (Binaire (Feuille 1, Feuille 1), Binaire (Feuille 1, Feuille 1)),
   Binaire (Binaire (Feuille 2, Feuille 2), Binaire (Feuille 2, Feuille 2))))
# developper a2;;
- : int arbre =
 Binaire
  (Binaire
    (Binaire
      (Binaire (Binaire (Binaire (Feuille 2, Feuille 2), Feuille 0), Feuille 1),
      Binaire (Feuille 0, Feuille 0)),
     Binaire (Binaire (Feuille 2, Feuille 2), Binaire (Feuille 2, Feuille 2))),
   Binaire
    (Binaire
      (Binaire (Binaire (Binaire (Feuille 2, Feuille 2), Feuille 0), Feuille 1),
      Binaire (Feuille 0, Feuille 0)),
    Binaire (Binaire (Feuille 2, Feuille 2), Binaire (Feuille 2, Feuille 2))))
```

Le développement de `a1` se dessine par



et celui de `a2` par



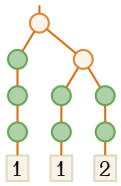
Q7. (TT) Écrire une fonction

```
val factoriser : 'a arbre -> 'a arbre = <fun>
```

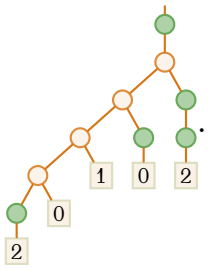
paramétrée par un arbre unaire binaire `arb` et qui renvoie l'arbre unaire binaire obtenu récursivement en remplaçant chaque nœud binaire de `arb`, sous la condition que son fils gauche et son fils droit soient égaux, par un nœud unaire dont le fils est celui qui figurait comme fils (gauche ou droit) du nœud binaire.

```
# factoriser a1;;
- : int arbre =
 Binaire (Unaire (Unaire (Unaire (Feuille 1))),
   Binaire (Unaire (Unaire (Feuille 1)), Unaire (Unaire (Feuille 2))))
# factoriser a2;;
- : int arbre =
 Unaire
  (Binaire
    (Binaire (Binaire (Binaire (Unaire (Feuille 2), Feuille 0), Feuille 1),
      Unaire (Feuille 0)),
    Unaire (Unaire (Feuille 2))))
```

La factorisation de a1 se dessine par



et celle de a2 par



Nous pouvons observer que la factorisation de a2 est égale à a2.

Exercice 2. (Le problème du siècle)

Un *siècle* est une façon de décomposer le nombre 100 par une expression faisant intervenir des nombres formés de chiffres allant de 1 à 9, où chaque chiffre apparaît dans l'ordre et en tout et pour tout exactement une fois, en utilisant additions et multiplications. Par exemple,

$$1 \times 2 \times 3 \times 4 + 5 + 6 + 7 \times 8 + 9,$$

et

$$1 + 2 \times 3 + 4 + 5 + 67 + 8 + 9$$

sont deux solutions à ce problème. Observons bien que dans le 2^{e} exemple, les chiffres 6 et 7 ont été associés pour former le nombre 67. Rappelons aussi que la multiplication est prioritaire face à l'addition. Ainsi, l'expression $1 \times 2 + 3$ se comprend comme l'expression $(1 \times 2) + 3$.

L'objectif de cet exercice est de générer tous les siècles possibles.

Un siècle est donc une expression particulière, qui sera représentée à l'aide des types

```
type chiffre = int
type ecriture = chiffre list
type terme = ecriture list
type expression = terme list
```

de la façon suivante :

- le type `chiffre` est un alias pour le type `int` et sert à représenter un chiffre (qui est donc ici une valeur comprise entre 1 et 9) ;
- le type `ecriture` sert à représenter une valeur en base dix par la liste de ses chiffres, ordonnés de manière naturelle. Par exemple, la liste `[1; 5; 3]` représente le nombre 153 ;
- le type `terme` sert à représenter un produit de nombres par la liste de ses nombres. Par exemple, la liste `[[1; 3; 5]; [2; 8; 1; 1]]` représente le produit 135×2811 ;
- le type `expression` sert à représenter une expression par la somme de ses termes. Par exemple, la liste `[[[1; 3; 5]; [2; 8; 1; 1]]; [[2; 2]; [1]; [1; 4; 4]]]` représente l'expression $135 \times 2811 + 22 \times 1 \times 144$.

Nous allons commencer par écrire quelques fonctions de base.

Q1. Définir deux alias `s1` et `s2` de type `expression` respectivement pour les expressions des deux siècles en exemple.

Q2. Écrire, **en utilisant des fonctions d'ordre supérieur**, des fonctions

```
val somme : int list -> int = <fun>
```

et

```
val produit : int list -> int = <fun>
```

paramétrées par une liste d'entiers `lst` et qui renvoient respectivement la somme et le produit des éléments de `lst`.

```
# somme [1; 2; 3; 1];;
- : int = 7
# produit [1; 2; 3; 1];;
- : int = 6
```

Q3. Écrire, **en utilisant une fonction d'ordre supérieur**, une fonction

```
val valeur_ecriture : int list -> int = <fun>
```

paramétrée par une écriture `ecr` et qui renvoie l'entier représenté par cette dernière.

```
# valeur_ecriture [];;
- : int = 0
# valeur_ecriture [1; 5; 3];;
- : int = 153
```

Q4. Écrire, **en utilisant la fonction** `produit` **et une fonction d'ordre supérieur**, une fonction

```
val valeur_terme : int list list -> int = <fun>
```

paramétrée par un terme `ter` et qui renvoie l'entier représenté par ce dernier.

```
# valeur_terme [];;
- : int = 1
# valeur_terme [[1; 5; 3]];;
- : int = 153
# valeur_terme [[1; 3; 5]; [2; 8; 1; 1]];;
- : int = 379485
```

Q5. Écrire, **en utilisant la fonction** `somme` **et une fonction d'ordre supérieur**, une fonction

```
val valeur_expression : int list list list -> int = <fun>
```

paramétrée par une expression `exp` et qui renvoie l'entier représenté par cette dernière.

```
# valeur_expression [];;
- : int = 0
# valeur_expression [[[1; 3; 5]; [2; 8; 1; 1]]; [[2; 2]; [1]; [1; 4; 4]]];;
- : int = 382653
# valeur_expression [[[1]]; [[2]; [3]]; [[4]]; [[5]]; [[6; 7]]; [[8]]; [[9]]];;
- : int = 100
```

Q6. Écrire, **en utilisant une fonction d'ordre supérieur**, une fonction

```
val concatener_images : ('a -> 'b list) -> 'a list -> 'b list = <fun>
```

paramétrée par une fonction `f` de type `'a -> 'b list` et une liste `lst` d'éléments de type `'a` et qui renvoie la liste d'éléments de type `'b` obtenus en concaténant les listes obtenues en appliquant `f` aux éléments de `lst`.

```
# concatener_images (fun x -> [x; 2 * x; 4 * x]) [0; 1; 2; 3];;
- : int list = [0; 0; 0; 1; 2; 4; 2; 4; 8; 3; 6; 12]
# concatener_images (fun u -> ["*"; u ^ u]) ["a"; "abc"];;
- : string list = ["*"; "aa"; "*"; "abcabc"]
```

Nous allons maintenant mettre au point un générateur d'expressions. Il devra générer la liste de toutes les expressions possibles sur une liste de chiffres spécifiée.

Q7. Étant donné un chiffre `ch` et une expression `exp` non vide, nous allons considérer les trois façons suivantes d'incorporer `ch` dans `exp` pour former une expression plus grande :

- insérer `ch` de sorte qu'il forme à lui seul un nouveau terme. Par exemple, si `ch` est le chiffre **1** et `exp` est l'expression $23\times4+5\times6$, l'expression obtenue est $\mathbf{1}+23\times4+5\times6$;
- insérer `ch` de sorte qu'il s'associe au premier terme d'`exp`. Par exemple, si `ch` est le chiffre **1** et `exp` est l'expression $23\times4+5\times6$, l'expression obtenue est $\mathbf{1}\times23\times4+5\times6$;
- insérer `ch` de sorte qu'il s'associe à la première écriture du premier terme d'`exp`. Par exemple, si `ch` est le chiffre **1** et `exp` est l'expression $23\times4+5\times6$, l'expression obtenue est $\mathbf{1}23\times4+5\times6$.

Écrire une fonction

```
val incorporer_chiffre : 'a -> 'a list list list -> 'a list list list list = <fun>
```

paramétrée par un chiffre `ch` et une expression `exp` non vide et qui renvoie la liste des trois nouvelles expressions issues de l'incorporation de `ch` dans `exp`. Dans le cas où `exp` est vide, l'exception `ExpressionVide` pourra être levée par `raise ExpressionVide`.

```
# incorporer_chiffre 1 [[[2; 3]; [4]]; [[5]; [6]]];;
- : int list list list list =
  [
     [[[1]]; [[2; 3]; [4]]; [[5]; [6]]];
     [[[1]; [2; 3]; [4]]; [[5]; [6]]];
     [[[1; 2; 3]; [4]]; [[5]; [6]]]
  ]
# incorporer_chiffre 1 [[[2]]];;
- : int list list list list =
  [
     [[[1]]; [[2]]];
     [[[1]; [2]]];
     [[[1; 2]]]
  ]
```

Q8. Écrire, **en utilisant les fonctions** `concatener_images` **et** `incorporer_chiffre`, une fonction

```
val etendre_expressions : 'a -> 'a list list list list -> 'a list list list list = <fun>
```

paramétrée par un chiffre `ch` et une liste d'expressions non vides `exp_lst` et qui renvoie la liste des expressions obtenues en incorporant `ch` dans chaque expression de `exp_lst`.

```
# let exp1 = [[[2; 3]; [4]]; [[5]; [6]]] in
  let exp2 = [[[2]]] in
  etendre_expressions 1 [exp1; exp2];;
```

```
- : int list list list list =
  [
      [[[1]]; [[2; 3]; [4]]; [[5]; [6]]];
      [[[1]; [2; 3]; [4]]; [[5]; [6]]];
      [[[1; 2; 3]; [4]]; [[5]; [6]]];
      [[[1]]; [[2]]];
      [[[1]; [2]]];
      [[[1; 2]]]
  ]
```

Q9. Écrire une fonction

```
val generer_expressions : 'a list -> 'a list list list list = <fun>
```

paramétrée par une liste de chiffres `ch_lst` et qui renvoie la liste de toutes les expressions qu'il est possible de construire sur les chiffres de `ch_lst`. Un algorithme possible consiste à étendre petit à petit les expressions obtenues en insérant un nouveau chiffre à chaque itération. Il est préférable que les chiffres apparaissent dans le même ordre dans `ch_lst` que dans les expressions générées mais ce n'est pas obligatoire.

```
# generer_expressions [1];;
- : int list list list list =
  [
      [[[1]]] (* expression 1 *)
  ]
# generer_expressions [1; 2];;
- : int list list list list =
  [
      [[[1]]; [[2]]]; (* expression 1 + 2 *)
      [[[1]; [2]]]; (* expression 1 x 2 *)
      [[[1; 2]]] (* expression 12 *)
  ]
# generer_expressions [1; 2; 3];;
- : int list list list list =
  [
      [[[1]]; [[2]]; [[3]]]; (* expression 1 + 2 + 3 *)
      [[[1]; [2]]; [[3]]]; (* expression 1 x 2 + 3 *)
      [[[1; 2]]; [[3]]]; (* expression 12 + 3 *)
      [[[1]]; [[2]; [3]]]; (* expression 1 + 2 x 3 *)
      [[[1]; [2]; [3]]]; (* expression 1 x 2 3 *)
      [[[1; 2]; [3]]]; (* expression 12 x 3 *)
      [[[1]]; [[2; 3]]]; (* expression 1 + 23 *)
      [[[1]; [2; 3]]]; (* expression 1 x 23 *)
      [[[1; 2; 3]]] (* expression 123 *)
  ]
```

Il est maintenant temps d'assembler tout cela pour répondre au problème de départ.

Q10. Écrire une fonction

```
val siecles : int list -> int -> int list list list list = <fun>
```

paramétrée par une liste de chiffres `ch_lst` et un entier `k` et qui renvoie la liste des expressions dont la valeur est `k`.

```
# siecles [1; 2; 3; 4; 5; 6; 7; 8; 9] 100;;
- : int list list list list =
  [
      [[[1; 2]]; [[3; 4]]; [[5]; [6]]; [[7]]; [[8]]; [[9]]];
      [[[1]; [2]]; [[3; 4]]; [[5]]; [[6]; [7]]; [[8]]; [[9]]];
      [[[1]]; [[2]; [3]]; [[4]]; [[5]]; [[6; 7]]; [[8]]; [[9]]];
      [[[1; 2]]; [[3]; [4]]; [[5]]; [[6]]; [[7]; [8]]; [[9]]];
      [[[1]; [2]; [3]; [4]]; [[5]]; [[6]]; [[7]; [8]]; [[9]]];
      [[[1]]; [[2]]; [[3]]; [[4]]; [[5]]; [[6]]; [[7]]; [[8]; [9]]];
      [[[1]; [2]; [3]]; [[4]]; [[5]]; [[6]]; [[7]]; [[8]; [9]]]
  ]
```