

Nom _____

Prénom _____

Code permanent _____

Solution de l'examen intra

Date : 29 octobre 2019

Titre du cours : Programmation fonctionnelle et logique

Sigle et groupe : INF6120

Enseignant : Alexandre Blondin Massé

Instructions

- 1) Vous avez trois heures pour répondre à l'examen ;
 - 2) Vous avez droit à toute votre documentation papier ;
 - 3) Vous ne devez pas dégrafer le questionnaire ;
 - 4) Il est interdit d'utiliser un ordinateur, peu importe sa taille et sa forme (téléphone portable, agenda électronique, etc.) ;
 - 5) Il est interdit de parler et de prêter de la documentation à un autre étudiant ;
 - 6) Indiquez clairement vos réponses finales ;
-

Question	1	2	3	4	5	6	Total
Sur	15	15	10	15	25	20	100
Note							

Question 1. (15 points)
(Pour cette question, aucune justification n'est requise. Dans le cas d'une mauvaise réponse, vous pourriez obtenir quelques points si vous avez laissé des traces de votre démarche.)

Pour chacune des expressions suivantes, indiquez quelle est la valeur retournée par l'interpréteur.

(1) (3 points)

```
>>> map (:"oi") "cflmrst"
```

Solution:

```
["coi", "foi", "loi", "moi", "roi", "soi", "toi"]
```

(2) (3 points)

```
>>> filter odd [x * x | x <- [1..10]]
```

Solution:

```
[1, 9, 25, 49, 81]
```

(3) (3 points)

```
>>> foldr1 (*) [1, 3, 5]
```

Solution:

```
15
```

(4) (3 points)

```
>>> zipWith (\x y -> x * y - 1) [1, 2, 3] [4, 5, 6]
```

Solution:

```
[3, 9, 17]
```

(5) (3 points)

```
>>> all ((==1) . (`mod` 7)) [22, 1, 14]
```

Solution:

```
False
```

Question 2. (15 points)

Écrivez une fonction qui prend en entrée une liste xs et qui retourne $True$ si et seulement si xs est une *répétition double*, c'est-à-dire qu'il existe une liste non vide plus courte ys telle que

$$xs == ys ++ ys.$$

Complétez directement le code ci-bas :

```
1  -- | Retourne vrai si une liste est une répétition double
2  --
3  -- >>> estDouble []
4  -- False
5  -- >>> estDouble [1,1]
6  -- True
7  -- >>> estDouble "abc"
8  -- False
9  -- >>> estDouble "abab"
10 -- True
11 -- >>> estDouble [1,2,3,1,2]
12 -- False
13 -- >>> estDouble [1,2,3,1,2,3]
14 -- True
15 estDouble :: Eq a => [a] -> Bool
16 estDouble xs = n > 0 && take h xs == drop h xs
17   where n = length xs
18         h = n `div` 2
```

Question 3. (10 points)

Nous avons vu en classe que, par défaut, le type `[a]` est une instance de `Ord`, en utilisant l'ordre lexicographique. On souhaiterait proposer une instance alternative de `Ord` pour les listes qui utilise l'ordre *radix*, défini comme suit :

- Si les deux listes sont de longueurs différentes, alors on compare la longueur des listes ;
- Si les listes sont de même longueur, alors on se rabat sur l'ordre lexicographique.

Complétez l'implémentation suivante pour obtenir l'ordre radix :

```
1 module RadixList where
2
3 newtype RadixList a = RadixList [a]
4     deriving (Eq, Show)
5
6 -- | L'ordre radix
7 --
8 -- Si les listes sont de longueurs différentes, on compare les longueurs:
9 --
10 -- >>> RadixList [5,6] < RadixList [1,2,3,4]
11 -- True
12 -- >>> RadixList "abc" >= RadixList "de"
13 -- True
14 --
15 -- Si elles sont de même longueur, on utilise l'ordre lexicographique:
16 --
17 -- >>> RadixList [1,4] < RadixList [2,3]
18 -- True
19 -- >>> RadixList "trois" < RadixList "trait"
20 -- False
21 instance Ord a => Ord (RadixList a) where
22     (RadixList xs) `compare` (RadixList ys)
23     | m == n    = xs `compare` ys
24     | otherwise = m `compare` n
25     where m = length xs
26           n = length ys
```

Question 4. (15 points)
(Pour cette question, aucune justification n'est requise. Dans le cas d'une mauvaise réponse, vous pourriez obtenir quelques points si vous avez laissé des traces de votre démarche.)

Qu'est-ce qui devrait être affiché par l'interpréteur si on entre les expressions suivantes ?

(1) (3 points)

```
>>> fmap (*4) Nothing
```

Solution:

```
Nothing
```

(2) (3 points)

```
>>> pure (*4) <*> Just 5
```

Solution:

```
Just 20
```

(3) (3 points)

```
>>> pure (:) <*> [1,2,3] <*> [[4],[5,6]]
```

Solution:

```
[[1,4],[1,5,6],[2,4],[2,5,6],[3,4],[3,5,6]]
```

(4) (3 points)

```
>>> replicate <$> [1,2,3] <*> "ab"
```

Solution:

```
["a","b","aa","bb","aaa","bbb"]
```

(5) (3 points)

```
>>> (<=>) <$> ZipList "banane" <*> ZipList "fraise"
```

Solution:

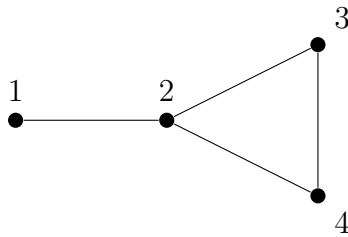
```
ZipList {getZipList = [True,True,False,True,True,True]}
```

Question 5. (25 points)

En théorie des ensembles, si E est un ensemble, alors on désigne par $\mathcal{P}_2(E)$ l'ensemble de toutes les paires de E , c'est-à-dire $\mathcal{P}_2(E) = \{\{a, b\} \mid a, b \in E, a \neq b\}$. Par exemple, si $E = \{1, 2, 3, 4\}$, alors

$$\mathcal{P}_2(E) = \{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 3\}, \{2, 4\}, \{3, 4\}\}.$$

Un graphe *simple* est un couple $G = (V, E)$, où V est un ensemble dont les éléments sont appelés *sommets* et $E \subseteq \mathcal{P}_2(V)$ est un ensemble dont les éléments sont appelés *arêtes*. Par exemple, si $V = \{1, 2, 3, 4\}$ et $E = \{\{1, 2\}, \{2, 3\}, \{2, 4\}, \{3, 4\}\}$, alors on obtient le graphe ci-bas, représenté par un dessin (il n'y a pas de flèche car le graphe est non orienté) :



Considérez les déclarations Haskell suivantes :

```

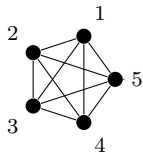
1 type Sommet = Int
2 type Arete = (Int, Int)
3 type Graphe = ([Sommet], [Arete])
4
5 g1 :: Int -> Graphe
6 g1 n = (v, e)
7     where v = [1..n]
8           e = [(i, j) | i <- [1..n], j <- [i+1..n]]
9
10 g2 :: Int -> Graphe
11 g2 n = (v, e)
12     where v = [1..n]
13           e = zip [1..n] (drop 1 . cycle $ [1..n])
14
15 g3 :: Int -> Int -> Graphe
16 g3 m n = (v, e)
17     where v = [1..m+n]
18           e = [(i, j) | i <- [1..m], j <- [m+1..m+n]]
19
20 g4 :: Int -> Graphe
21 g4 n = (v, e)
22     where v = [0..n]
23           e = zip [1..n] (repeat 0)
24
25 -- Rappel: la fonction snd retourne le 2e élément d'un couple
26 g5 :: Int -> Graphe
27 g5 n = (v, e1 ++ e2)
28     where v = [0..n]
29           e1 = snd $ g2 n
30           e2 = snd $ g4 n

```

Dessinez le graphe correspondant à chacune des expressions suivantes.

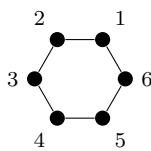
(1) (3 points) $g1\ 5$

Solution:



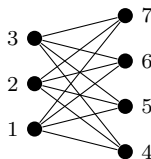
(2) (3 points) $g2\ 6$

Solution:



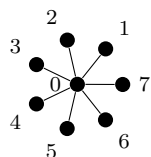
(3) (3 points) $g3\ 3\ 4$

Solution:



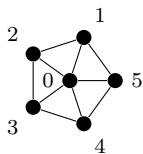
(4) (3 points) $g4\ 7$

Solution:



(5) (3 points) $g5\ 5$

Solution:

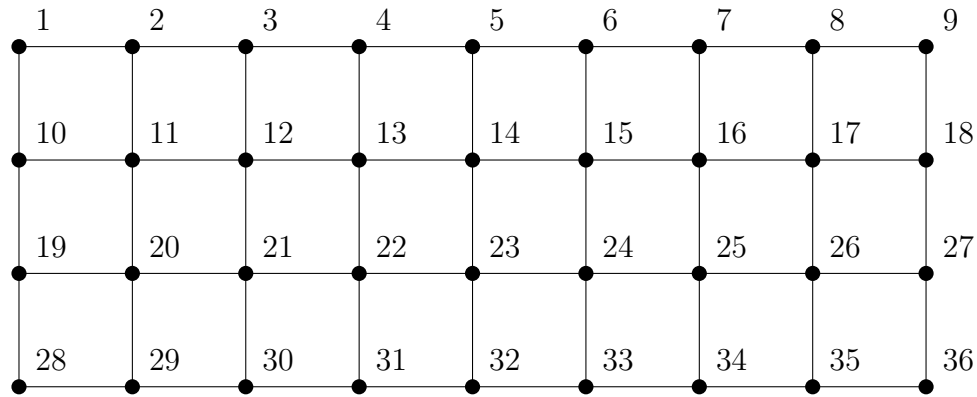


(6) (10 points) Écrivez une fonction Haskell dont la signature est

```
grille :: Int -> Int -> Graphe
```

qui génère le graphe grille de m lignes et de n colonnes.

Par exemple, on souhaiterait que l'appel `grille 4 9` génère le graphe dessiné ci-bas :



Aide : ce graphe possède mn sommets et $m(n-1) + n(m-1)$ arêtes. De plus, il peut être pratique d'introduire une fonction auxiliaire qui calcule le numéro d'un sommet $k(i, j)$ à partir du numéro de ligne i et du numéro de colonne j .

Solution:

```
1 grille :: Int -> Int -> Graphe
2 grille m n = (v, e1 ++ e2)
3   where v = [1..m*n]
4         e1 = [(k i j, k i (j + 1)) | i <- [1..m], j <- [1..n-1]]
5         e2 = [(k i j, k (i + 1) j) | i <- [1..m-1], j <- [1..n]]
6         k i j = (i - 1) * n + j
```

Question 6. (20 points)

Un *arbre rouge-noir* est un arbre binaire de recherche dans lequel chaque noeud est colorié en rouge ou en noir (avec la convention que la couleur d'un arbre vide est noire).

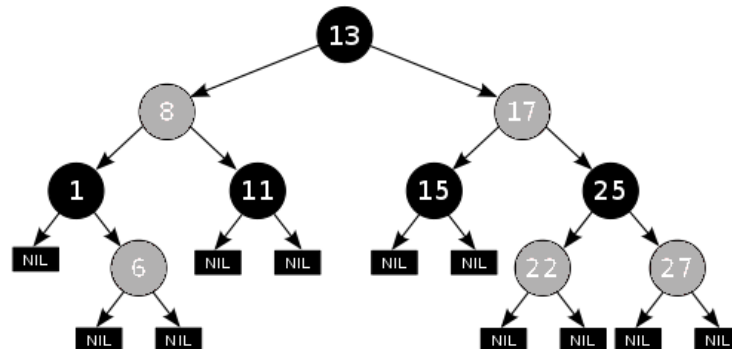
Étant donné un noeud N dans un arbre rouge-noir, on appelle *hauteur noire* de N le nombre maximal de noeuds noirs qu'on rencontre à partir de N jusqu'à n'importe quelle feuille. Plus précisément,

- La hauteur noire d'un noeud vide est 1 ;
- La hauteur noire d'un noeud noir ayant G et D comme enfants est égale à 1 plus le maximum des hauteurs noires de G et de D ;
- La hauteur noire d'un noeud rouge ayant G et D comme enfants est égale au maximum des hauteurs noires de G et de D .

On dit qu'un arbre rouge-noir est *valide* s'il vérifie les 4 propriétés suivantes :

1. Un arbre vide est considéré de couleur noire ;
2. La racine de l'arbre doit être noire ;
3. Les deux enfants d'un noeud rouge sont nécessairement noirs ;
4. Pour tout noeud dont les enfants sont G et D , la hauteur noire de G doit être égale à la hauteur noire de D .

Voici un exemple d'arbre rouge-noir valide (les noeuds rouges apparaissent en gris dans la version imprimée et les arbres vides sont identifiés par *NIL*) :



Complétez le module *RedBlack.hs* suivant en tenant compte des observations données plus haut et en lisant attentivement la documentation (*docstring*). Écrivez directement dans le code. Les points sont répartis comme suit :

- (1) (4 points) L'instance *Functor* pour *RBN* ;
- (2) (4 points) La fonction *color* ;
- (3) (6 points) La fonction *blackHeight* et
- (4) (6 points) La fonction *isValid*.

Ci-bas le programme complet :

```

1 module RougeNoir where
2
3 -- | Data types
4 data Color = Red | Black
5     deriving (Eq, Show)
6
7 data RBT a = RBT { -- Red-black tree
8     root :: RBN a -- Root node
9 } deriving Show
10
11 data RBN a -- Red-black node
12     = Empty -- Empty tree
13     | Node Color -- Color of node
14         a -- Content of node
15         (RBN a) -- Left subtree
16         (RBN a) -- Right subtree
17     deriving Show
18
19 -- | Making a red-black leaf
20 makeLeaf :: Color -> a -> RBN a
21 makeLeaf c x = Node c x Empty Empty
22
23 -- | Valid red-black tree
24 tree1 = RBT
25     (Node Black 13 (
26         Node Red 8 (
27             Node Black 1 (
28                 Empty)
29             (makeLeaf Red 6)) (
30                 makeLeaf Black 11)) (
31             Node Red 17 (
32                 makeLeaf Black 15) (
33                 Node Black 25 (
34                     makeLeaf Red 22) (
35                     makeLeaf Red 27))))))
36
37 -- | Invalid: root is not black
38 tree2 = RBT (Node Red 8 (makeLeaf Black 4) (makeLeaf Black 9))
39
40 -- | Invalid: red node cannot have red child
41 tree3 = RBT
42     (Node Black 8 (
43         Node Red 4
44             Empty (
45                 makeLeaf Red 5)) (
46                 makeLeaf Black 9))
47
48 -- | Invalid: wrong black height at the root
49 tree4 = RBT (Node Black 8 (makeLeaf Red 4) (makeLeaf Black 9))
50
51 -- | Invalid: wrong black height at the root

```

```

52 tree5 = RBT (Node Black 8 (makeLeaf Black 4) Empty)
53
54 -- | A red-black node is a functor
55 --
56 -- >>> fmap (+1) (makeLeaf Red 5)
57 -- Node Red 6 Empty Empty
58 instance Functor RBN where
59     fmap _ Empty          = Empty
60     fmap f (Node c x l r) = Node c (f x) (fmap f l) (fmap f r)
61
62 -- | A red-black tree is a functor
63 --
64 -- >>> fmap (+1) tree5
65 -- RBT {root = Node Black 9 (Node Black 5 Empty Empty) Empty}
66 instance Functor RBT where
67     fmap f = RBT . fmap f . root
68
69 -- | Returns the color of a node
70 --
71 -- Note: an empty node is considered black
72 --
73 -- >>> color Empty
74 -- Black
75 -- >>> color $ makeLeaf Red 4
76 -- Red
77 -- >>> color $ root tree1
78 -- Black
79 color :: RBN a -> Color
80 color Empty          = Black
81 color (Node c _ _ _) = c
82
83 -- | Returns the black height of a node
84 --
85 -- This is the maximum number of black nodes that are visited from the current
86 -- node to any leaf. It is defined recursively as follows:
87 --
88 -- * An empty node has black height 1
89 -- * A node of color black has height 1 + the maximum black height between the
90 --   left and right subtrees.
91 -- * A node of color red has height equal to the maximum black height between
92 --   the left and right subtrees.
93 --
94 -- >>> blackHeight Empty
95 -- 1
96 -- >>> blackHeight $ makeLeaf Red 4
97 -- 1
98 -- >>> blackHeight $ makeLeaf Black 4
99 -- 2
100 -- >>> map (blackHeight . root) [tree1, tree2, tree3, tree4, tree5]
101 -- [3,2,3,3,3]
102 blackHeight :: RBN a -> Int
103 blackHeight Empty = 1

```

```
104 blackHeight (Node c _ l r) = (if c == Black then 1 else 0)
105     + max (blackHeight l) (blackHeight r)
106
107 -- | A red-black tree is valid if it satisfies the following 3 properties:
108 --
109 -- 1. The root is black
110 -- 2. A red node has only black children.
111 -- 3. For any node, the black height of its left and right subtrees are equal.
112 --
113 -- >>> map isValid [tree1, tree2, tree3, tree4, tree5]
114 -- [True,False,False,False,False]
115 isValid :: RBT a -> Bool
116 isValid (RBT root) = color root == Black && isValid' root
117     where isValid' :: RBN a -> Bool
118           isValid' Empty = True
119           isValid' (Node c _ l r)
120               = (c == Black || (color l == Black && color r == Black)) &&
121                 (blackHeight l == blackHeight r) && isValid' l && isValid' r
```