14/01/2013
Durée : 2 heures
Documents autorisés

Programmation fonctionnelle
Master 1 Informatique

Attention : donnez autant de détails que vous jugez nécessaire ; des simples copies-collers-miroirs de vos notes de cours ne seront pas jugés satisfaisants. En bref, montrez que vous avez compris.

Types

Exercice 1. Calculez le type et, éventuellement, les contraintes de classe du type, de chacune des expressions et fonctions suivantes :

```
tailleAlphabet = ord 'z' - ord 'A'
encoder [] = 0
encoder (x:xs) = (ord x) - (ord 'A') + tailleAlphabet * (encoder xs)
decoder 0 = []
decoder x =
    (chr (x `mod` tailleAlphabet + ord 'A')):(decoder (x `div` tailleAlphabet))

absMax [] = -1
absMax (x:xs) = max (abs x) (absMax xs)

fG f g [] = -1
fG f g (x:xs) = g (f x) (fG f g xs)
```

Rappel :

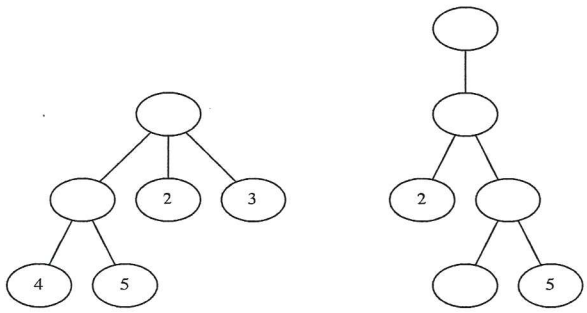
```
ord :: Char -> Int -- Transforme un caractere dans son code ascii
chr :: Int -> Char -- Transforme un entier dans le caractere correspondant
                   -- (selon ascii)
```

Types récursifs

Pour les exercices suivants, considérez le type (récursif) des arbres dont chaque noeud possède une liste ordonnée d'enfants, et dont chaque feuille est étiquetée par un entier. Le code Haskell qui définit ce type est le suivant :

```
data Arbre = Feuille Int | Noeud [Arbre]
```

Exercice 2. Écrivez deux expressions qui codent en Haskell ces deux arbres :



Exercice 3. Écrivez un script Haskell contenant la définition de :
- une fonction `maximum`, qui calcule l'étiquette maximum d'une feuille de l'arbre passé en paramètre,

- une fonction `somme`, qui calcule la somme de toutes les étiquettes des feuilles de l'arbre passé en paramètre,
- une fonction `profondeur`, qui calcule la longueur d'une branche de longueur de maximale de l'arbre en paramètre.

Exercice 4.
- Complétez le script de l'exercice précédent en y définissant une fonction

```
foldArbre :: (Int -> a) -> ([a] -> a) -> Arbre -> a
```

  telle que :
  - `foldArbre f g` appliquée à une feuille, produira la valeur de `f` sur l'étiquette,
  - `foldArbre f g` appliquée à un noeud produira la valeur de `g` appliquée à la liste obtenue des appels récursifs sur les enfants.

  Par exemple, `foldArbre (+1) sum`, appliqué à l'arbre sur la gauche de l'exercice 2, produira `18` comme résultat.
- Montrez comment les fonctions `maximum`, `somme` et `profondeur` (de l'exercice précédent) peuvent se définir à partir de la fonction `foldArbre`.

Évaluation

Exercice 5. Évaluez, par nom (stratégie call-by-name) *et* par valeur (stratégie call-by-value), les expressions `e1`, `e2` et `e3`, du script Haskell suivant :

```
e1 = (\x y -> x + y) (1 + 2) (3 + 4)
omega = omega + 1
e2 = (\x y -> y) omega 33
ifthenelse x y z = if x then y else z
e3 = ifthenelse True 0 omega
```

Polynômes (divertissement)

Exercice 6. Écrivez un script Haskell qui contiendra :
- la définition d'un type `Pol`, pour représenter les polynômes à coefficients entiers,
- une fonction `eval :: Pol -> Int -> Int`, qui évalue un polynôme en un entiere donné,
- une fonction `toPol :: Int -> Int -> Pol` telle que, appliquée à deux entier n et m, produira le polynôme $P(x) = \sum_i a_i x^i$ tel que $P(m) = n$.

Dans cet exercice, on vous demande de faire preuve de votre maîtrise du langage Haskell : des erreurs de syntaxe ou de typage seront tenu en compte pour la correction.

25/06/2013
Durée : 2 heures
Documents autorisés

Programmation fonctionnelle
Master 1 Informatique

Attention : donnez autant de détails que vous jugez nécessaire ; des simples copies-collers-miroirs de vos notes de cours ne seront pas jugés satisfaisants. En bref, montrez que vous avez compris.

Types

Exercice 1. Calculez le type et, éventuellement, les contraintes de classe du type, de chacune des fonctions définies ci-dessous :

```
distance x y = sqrt (x^2 + y^2)
f (x:xs) = map (\y -> y + x) xs
g f ('a',x) = 'a':f x
tri plsptt [] =  []
tri plsptt [x] = [x]
tri plsptt (x:xs) = if (plsptt x y) then x:y:ys else y:(tri plsptt (x:ys))
    where
      y:ys = tri plsptt xs
sort [] = []
sort [x] = [x]
sort (x:xs) = if x < y then x:y:ys else y:(sort (x:ys))
    where
      y:ys = sort xs
```

Types récursifs

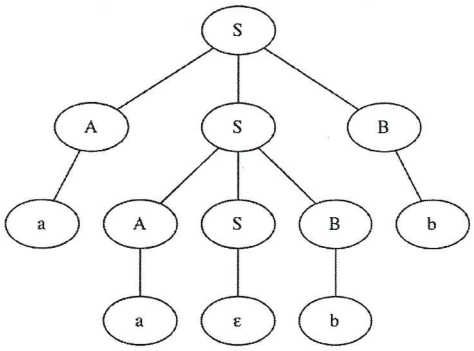
Exercice 2. Considérez la grammaire donnée par les quatre productions (ou règles) suivantes :

$$S \Rightarrow ASB \mid \epsilon, \qquad A \Rightarrow a, \qquad B \Rightarrow b.$$

Nous allons débuter l'écriture d'un script Haskell pour manipuler cette grammaire.

Question 2.1 Définissez un type inductif Haskell (qui sera nommé `Arbre_der`) pour représenter les arbres de dérivation de cette grammaire. Le définition sera telle que tout valeur ayant type `Arbre_der` soit le codage d'un arbre de dérivation de cette grammaire.

Question 2.2 En utilisant le type inductif `Arbre_der` défini dans la question précédente, définissez une expression Haskell qui code cet arbre de dérivation :



Question 2.3 Ajoutez au script la définition d'une fonction `frontiere : Arbre_der -> String` qui produit le mot qui se trouve sur les feuilles d'un arbre de dérivation.

Question 2.4 Écrivez une fonction `appartient : String -> Bool` qui retourne `true` ssi le mot passé en paramètre appartient au langage de la grammaire.

Question 2.5 Modifiez la fonction `appartient` pour écrire une fonction `appartientArbre : String -> Maybe Arbre_der` qui retourne `Just a` où `a` est un arbre de dérivation du mot passé en paramètre, et `Nothing` si un tel arbre de dérivation n'existe pas (c'est à dire, si le mot passé en paramètre n'appartient pas au langage de la grammaire).
Rappels :
- Le type `Maybe a` est défini comme suit :

```
data Maybe a = Nothing | Just a
```

- Un *arbre de dérivation* est un arbre fini étiqueté par les symboles d'une grammaire donnée, tel que : a) la racine est étiquetée par le symbole de départ (ou axiome) ; b) si X est l'étiquette d'un noeud et w est le mot composé par les étiquettes des fils du noeud, alors $X \Rightarrow w$ est une production (ou règle) de la grammaire.

Évaluation

Exercice 3. Évaluez, par nom (stratégie call-by-name) *et* par valeur (stratégie call-by-value), les expressions `e1`, `e2` et `e3`, du script Haskell suivant :

```
e1 = (\x -> \y -> x y) (\z -> z + 1) (3 + 4)
e2 = if True then (1 + 1) else (1 + 2)
zeros = 0:zeros
prendre 0 x = []
prendre n [] = []
prendre n (x:xs) = x:(prendre (n-1) xs)
e3 = prendre 1 zeros
```

Divertissements

Exercice 4. Les fonctions `tri` et `sort` de l'exercice 1 ont une complexité inutile.
Modifiez le code des fonctions `tri` et `sort` de pour obtenir deux fonctions de tri, `tri_a_bulles` et `bubble_sort`, qui soient une implémenterions de l'algorithme de tri à bulles.
Le type de la fonction `tri_a_bulles` sera le même que celui de la fonction `tri` ; le type de la fonction `bubble_sort` sera le même que celui de la fonction `sort`.

Exercice 5. La suite de Fibonacci $\{f_n \mid n \geq 0\}$ est définie par $f_0 = 0$, $f_1 = 1$, et $f_n = f_{n-1} + f_{n-2}$ pour $n \geq 2$.
Écrivez un script Haskell où sera définie une expression qui code la liste infinie $[f_0, f_1, f_2, \ldots, f_n, \ldots]$.
Conseil : dans le script vous pouvez aussi définir d'autres objets (par exemple : la liste de tous les entiers, la fonction qui envoie n vers f_n, etc.) qui contribueront à réaliser l'objectif de l'exercice.

Date : 09/01/2014
Durée : 2 heures
Documents authorisés
Ordinateurs interdits

Cours Programmation Fonctionnelle
Master 1 Informatique, Aix-Marseille Université

Examen

Attention : donnez autant de détails que vous jugez nécessaire ; des simples copies-collers-miroirs de vos notes de cours ne seront pas jugés satisfaisants. En bref, montrez que vous avez compris.

Types, évaluation

Exercice 1. Rappelons la définition des fonctions `foldr` et `foldl` :

```
foldr :: (a -> b -> b) ->  b ->  [a] -> b
foldr f v [] = v
foldr f v (x:xs) = f x (foldr f v xs)

foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f v [] = v
foldl f v (x:xs) = foldl f (f v x) xs
```

Considérez maintenant les deux expressions suivantes :

```
exp1 = foldr (&&) True (map even [0..])
exp2 = foldl (&&) True (map even [0..])
```

Question 1.1. Donnez le type (incluant les contraintes de type éventuelles) de chacune des expressions suivantes :

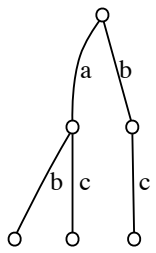
`(&&), True, map, even, [0..], expr1, expr2.`

(FQu 1.1)

Question 1.2. En Haskell, quelle expression parmi `expr1` et `expr2` produit (s'évalue à) un valeur ? Justifiez de manière circonstanciée votre réponse. (FQu 1.2)

Types récursifs

Exercice 2. Un ensemble de mots L est *préfixe* si $\epsilon \in L$ et, pour tous mots w et u, si $wu \in L$ alors $w \in L$.[1] En partageant les préfixes des mots par une structure arborescente, on peut économiser l'espace nécessaire à représenter sur ordinateur un tel ensemble. Par exemple, l'ensemble $\{\epsilon, a, ab, ac, b, bc\}$ peut se présenter par l'arbre à la droite. Le type récursif `EnsPref` defini par




```
data EnsPref = Noeud [(Char,EnsPref)]
```

permet de coder des tels arbres, où les arêtes sont étiquetées par des caractères, en Haskell (et donc aussi les ensembles préfixes). Par exemple, l'arbre à la droite sera codé par l'expression Haskell

```
Noeud [('a',Noeud [('b',Noeud []),('c',Noeud [])]),
                                  ('b',Noeud [('c',Noeud [])])]
```

Question 2.1. Comment peut on optimiser la définition du type `EnsPref` ? Justifiez votre réponse. (FQu 2.1)

1. ϵ denote ici le mot vide, et wu est la concatenation des mots w et u.

Considérez maintenant les fonctions suivantes :

```
singleton :: EnsPref
singleton = Noeud []

ajouterMot :: EnsPref -> String -> EnsPref
ajouterMot dict [] = dict
ajouterMot (Noeud []) (x:xs) = Noeud [(x,ajouterMot (Noeud []) xs)]
ajouterMot (Noeud ((c,d):ds)) (x:xs)
    | c == x = Noeud ((c,ajouterMot d xs):ds)
    | otherwise =
        let
            Noeud queue = ajouterMot (Noeud ds) (x:xs)
        in Noeud ((c,d):queue)

ajouterMots :: EnsPref -> [String] -> EnsPref
ajouterMots = foldl ajouterMot
```

Question 2.2. A quelle valeur s'évalue l'expression `ajouterMots singleton ['''a''','''bc''']` ? Représentez cette valeur comme une expression Haskell et aussi sous la forme d'arbre, comme dans la figure. (FQu 2.2)

Question 2.3. Listez tous les patterns que vous voyez dans la code ci-dessous. A quelle ligne se trouvent les conditions de garde ? Quelles sont les expressions définies par récursion ? Lesquelles par filtrage ? Lesquelles utilisent les conditions de garde dans leur définition ? (FQu 2.3)

Question 2.4. Écrivez une fonction `mots : EnsPref -> [String]`, qui liste tous les mots qui appartiennent à un ensemble préfixe passé en paramètre. (FQu 2.4)

Chaînes compressées (divertissement)

Exercice 3. Pour une chaîne de caractères $w = w_1 w_2 \dots w_n$ (avec w_i des caractères) et des entiers i, j avec $1 \leq i \leq j \leq n$, le facteur $w[i,j]$ est la sous-chaîne $w_i w_{i+1} \dots w_j$ de w.

Question 3.1. Définissez une fonction Haskell `facteur` qui prends en paramètre une chaîne de caractères w et une paire d'entiers (i,j), et retourne `Just` $w[i,j]$ si la chaîne $w[i,j]$ est bien définie. Si $w[i,j]$ n'est pas définie, la fonction retournera `Nothing`. (FQu 3.1)

Une *chaîne compressée* est une suite $W = w_1 w_2 \dots w_n$ où, pour tout $i = 1, \dots, n$, w_i est ou bien un caractère, ou bien une paire (i,j) d'entiers.

Question 3.2. Définissez un type de données `ZipString` pour représenter en Haskell les chaînes compressées. (FQu 3.2)

Une chaîne compressée $W = w_1 w_2 \dots w_n$ représente son expansion $\ulcorner W \urcorner$, définie par induction comme suit :

$$\ulcorner \epsilon \urcorner = \epsilon\,, \quad \ulcorner W x \urcorner = \ulcorner W \urcorner\, u\,, \text{ où } \epsilon \text{ est la chaîne vide, et } \quad u = \begin{cases} x & \text{si } x \text{ est un caractère,} \\ \ulcorner W \urcorner\,[i,j] & \text{si } x = (i,j) \text{ est une paire d'entiers.} \end{cases}$$

Question 3.3. Définissez une fonction Haskell `unzip : ZipString -> Maybe String`, qui retourne `Just` $\ulcorner W \urcorner$, si la chaîne compressée W passée en paramètre peut se décompresser, et retourne `Nothing` sinon. (FQu 3.3)

Question 3.4. Définissez une fonction Haskell `bienformee : ZipString -> Bool`, qui retourne `True` si et seulement si la chaîne compressée W passée en paramètre est bien formée, c'est à dire si elle peut se décompresser. La fonction `bienformee` sera optimisée (par rapport à la fonction `unzip`), car elle n'essayera pas de décompresser la chaîne passée en paramètre. (FQu 3.4)

Date : 25/06/2014
Durée : 2 heures
Documents authorisés
Ordinateurs interdits

Cours Programmation Fonctionnelle
Master 1 Informatique, Aix-Marseille Université

Examen

Typage, évaluation

Exercice 1. (6 pts). Considérez les trois définitions suivantes :

```
nombre_c [] = 0
nombre_c ('c':xs) = 1 + nombre_c xs
nombre_c ([x]:xs) = nombre_c xs
monMap f [] = []
monMap f (x:xs) = f x (monMap f xs)
trier [] p = []
trier [x] p = [x]
trier (x:xs) p = if p x y then x:(y:ys)
       else
           y:(trier (x:ys))
    where
        y:ys = trier xs
```

Pour chaque fonction définie, si l'analyse de son type produit une erreur :

1. expliquez pourquoi on a cette erreur ;
2. corrigez la définition par conséquent ;
3. donnez le type de l'expression définie ;

sinon, donnez simplement le type de l'expression définie.

Exercice 2. (4 pts). Considérez l'expression Haskell suivante :

```
(\x -> \f -> f x) 1 ((\y -> \z -> y + z) 1)
```

Question 2.1. Utilisez la stratégie par valeur (« *call-by-value* ») pour évaluer cette expression. (Fin question 2.1)

Question 2.2. Utilisez la stratégie par nom (« *call-by-name* ») pour évaluer cette expression. (Fin question 2.2)

Types récursifs

Exercice 3. (6 pts). Un *arbre binaire* est ou bien une feuille, ou bien un noeud ayant exactement deux enfants (l'un à gauche, l'autre à droite) ; chaque enfant est lui même un arbre binaire.

Question 3.1. Définissez en `Haskell` un type de données, nommé `ArbreBin`, pour coder les arbres binaires. (Fin question 3.1)

Question 3.2. Utilisez la réponse à la question précédente pour écrire une expression qui code l'arbre binaire sur la gauche de la figure 1. (Fin question 3.2)

Un *arbre à branchement fini* est un noeud ayant une liste ordonnée (de gauche à droite) d'enfants ; chaque enfant est lui même un arbre à branchement fini, il est possible que la liste d'enfants soit vide.

Question 3.3. Définissez en `Haskell` un type de données, nommée `ArbreBrf`, pour coder les arbres à branchement fini. (Fin question 3.3)

Question 3.4. Utilisez la réponse à la question précédente pour écrire une expression qui code l'arbre à branchement fini sur la droite de la figure 1. (Fin question 3.4)

On peut transformer un arbre binaire en un arbre à branchement fini, récursivement, de la façon suivante. Une feuille est transformée dans un noeud ayant une liste vide d'enfants ; un noeud avec deux enfants g et d (pour gauche et droite) est traité comme suit : on applique la procédure à g et d pour obtenir deux arbres à branchement fini l et r ; on retourne l'arbre qui est un noeud et dont la liste des enfants est obtenue de la liste des enfants de r en lui ajoutant en tête l'arbre l.
Par exemple, l'arbre à branchement fini en figure 1 sur la droite est obtenu de l'arbre binaire sur la gauche.
Question 3.5. Implémentez cette transformation en `Haskell` en écrivant la définition d'une fonction

```
arbreBintoArbreBrf ::
      ArbreBin -> ArbreBrf
```

(Fin question 3.5)

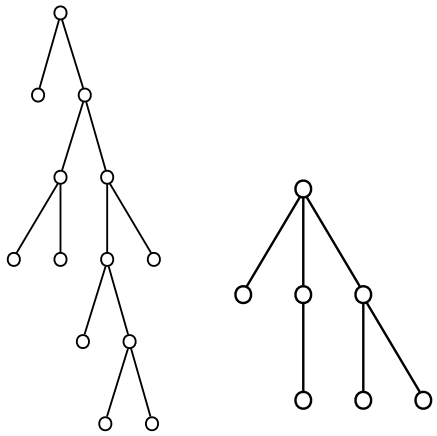


FIGURE 1 – Arbre binaire (gauche) et arbre à branchement fini (droite)

Sous-ensembles finis

Exercice 4. Le type `EnsFin` défini ci-dessous sert à manipuler des sous-ensembles finis (donc, pas d'éléments répétés) dont les élements sont tirés d'un type génerique `a` :

```
data EnsFin a = EnsFin [a] deriving Show
```

Question 4.1. (1 pts). Peut on optimiser la définition du type `EnsFin` ? Comment ? Justifiez votre réponse.

(Fin question 4.1)

Question 4.2. (1 pts). Expliquez, avec précision, ce qui se passe avec le mot clés `deriving` dans la définition du type `EnsFin`. (Fin question 4.2)

Considérez maintenant les définitions suivantes :

```
ensFinVersListe (EnsFin xs) = xs
vide = EnsFin []
appartient x (EnsFin []) = False
appartient x (EnsFin (y:ys)) =
    if x == y then True else appartient x (EnsFin ys)
ajouter_el x (EnsFin []) = EnsFin [x]
ajouter_el x (EnsFin (y:ys)) =
    if x == y then (EnsFin (y:ys)) else EnsFin (y:zs)
        where zs = ensFinVersListe (ajouter_el x (EnsFin ys))
test = ajouter_el 3 (EnsFin [1,3,4])
```

Question 4.3. (2 pts). Donnez un type à chacune des expressions définies ci-dessus.
Attention : n'oubliez pas les contraintes de classe quand elles sont nécessaires ; précisez aussi lesquelles parmi ces expressions sont polymorphes. (Fin question 4.3)

Question 4.4. (3 pts). Écrivez les définitions des fonctions

```
listeVersEnsFin :: Eq a => [a] -> EnsFin a
union :: Eq a => EnsFin a -> EnsFin a -> EnsFin a
intersection :: Eq a => EnsFin a -> EnsFin a -> EnsFin a
```

`listeVersEnsFin` transforme une liste vers un ensemble finis (attention : pas d'éléments répétés dans un ensemble), `union` et `intersection` sont les opérations ensemblistes usuelles. (Fin question 4.4)

Points totaux : 23

Date : 09/01/2015
Durée : 2 heures
Documents autorisés
Ordinateurs interdits

Cours Programmation Fonctionnelle
Master 1 Informatique, Aix-Marseille Université

Examen

Remarque : cet examen présente une choix d'exercices et questions ; pour obtenir la note maximum, ce n'est pas nécessaire de tous les résoudre de façon correcte.

Exercice 1 : typage. Considérez le script `Haskell` suivant :

```
class MyNum a where
    zero :: a
newtype MyInt = MyInt Int deriving Show
instance MyNum MyInt where
    zero = MyInt 0
newtype MyReal = MyReal Float deriving Show
instance MyNum MyReal where
    zero = MyReal 0
if1 = if True then show (zero::MyInt) else show (zero::MyReal)
if2 = if True then (zero::MyInt) else (zero::MyReal)
sum x y = MyReal (x + y)
mult (MyReal x) (MyReal y) = MyReal (x*y)
power x (MyInt 0) = MyReal 1
power x n = mult (power x (n-1)) x
```

Pour chaque expression parmi `if1, if2, sum, mult, power`, si lors de la compilation l'analyse du type produit une erreur :

1. expliquez pourquoi on a cette erreur,
2. corrigez la définition par conséquent,
3. donnez le type de l'expression corrigée ;

sinon, donnez simplement le type de l'expression définie.

Exercice 2 : évaluation, λ-calcul. Considérez l'expression du λ-calcul suivante :

$$(\lambda x.\lambda f.(f\,x))\,z\,((\lambda g.\lambda y.(g\,y))\,w)$$

Question 2.1. Quels sont les redexes qui apparaissent dans cette expression ? (Fin question 2.1)

Question 2.2. Évaluez cette expression en utilisant l'ordre applicatif (évaluation par l'intérieur). (Fin question 2.2)

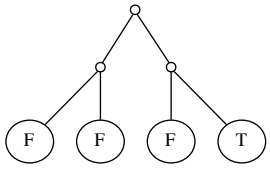
Question 2.3. Évaluez cette expression en utilisant l'ordre normal (évaluation par l'extérieur). (Fin question 2.3)

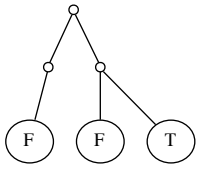
Exercice 3 : types récursifs et λ-calcul. Définissez en `Haskell` un type de données—qui sera appélé `LambdaTerm`—pour coder les termes (ou expressions) du λ-calcul. Écrivez ensuite une expression `Haskell` de type `LambdaTerm` qui soit le codage du terme

$$\lambda f.\lambda x.(f\,x)\,.$$

Exercice 4 : types récursifs. Un arbre de décision est un arbre dont les noeuds internes possèdent un ou deux enfants et dont les feuilles sont étiquetées par des valeurs d'un domaine donné. Ces arbres peuvent s'utiliser pour représenter des fonctions dont les arguments sont des tuplets de Booléens. Par exemple, la fonction AND, dont la table se trouve ci-dessous sur la gauche, peut se représenter par l'arbre ci-dessous au centre :

x_1	x_2	x_1 AND x_2
F	F	F
F	T	F
T	F	F
T	T	T





La méthode est la suivante : à partir de la racine, on parcourt l'arbre en lisant le tuplet de Booléens (False=descendre à gauche, True=descendre à droite) en entrée pour découvrir sur la feuille la valeur de la fonction quand appliquée à ce tuplet. On exploite la possibilité que les noeuds internes aient un seul enfant pour réduire la taille de la représentation d'une fonction lorsque la valeur à calculer ne dépend pas d'une variable. Par exemple, la fonction AND peut aussi se représenter par l'arbre de décision ci-dessus sur la droite, en utilisant le fait que quand la première variable vaut Faux, alors quelque soit la valeur de la deuxième variable, on obtient le résultat Faux.

Voici la définition en `Haskell` d'un type de données récursif pour les abres de décision binaires :

```
data ArbreDec a = Value a
                  | FalseTrue (ArbreDec a) (ArbreDec a)
                  | DontCare (ArbreDec a)
```

Question 4.1. Représentez la fonction AND par une expression `Haskell` ayant type `ArbreDec Bool`.

(Fin question 4.1)

Question 4.2. Dessinez l'arbre de décision qui représente la fonction Booléenne de trois variables qui retourne `Vrai` si et seulement si la première et la deuxième variables ont la même valeur, ou la deuxième et la troisième variables ont la même valeur. Représentez ensuite cet arbre par une expression `Haskell` de type `ArbreDec Bool`. (Fin question 4.2)

Question 4.3. Écrivez la définition d'une fonction

```
eval :: ArbreDec a ->  [Bool] -> a
```

qui prend en paramètre un arbre de décision (représentant une fonction) et une liste de Booléens (représentant un tuplet), et évalue cette fonction avec ce tuplet comme entrée. (Fin question 4.3)

Question 4.4. La fonction `eval` demande deux paramètres, un arbre de décision qui code une fonction demandant un tuplet de *longueur fixée* de paramètres Booléens, et une liste de *longueur variable* de Booléens. On pourrait donc avoir que la longueur de la liste ne correspond pas à la longueur du tuplet demandé par la fonction. Modifiez votre définition (et, si nécessaire, le type) de la fonction `eval` pour considérer cette éventualité comme une erreur. (Fin question 4.4)

Exercice 5 : suites d'entiers. On s'amuse, dans le script suivant, à définir trois listes infinies d'entiers :

```
suite1 = suite 1 2
    where suite n m = n:suite m (n+m)
suite2 = suite 2 1
    where suite n m = n^m:suite n (m+1)
suite3 = suite 1
    where suite n = n:map (n*) (suite (n+1))
```

Question 5.1. Décrivez en langue française (ou en utilisant une notation mathématique appropriée) quelles sont les suites définies par `suite1,suite2,suite3`. (Fin question 5.1)

Question 5.2. Avec les définitions ci-dessus, l'évaluation de l'expression

```
concat (map (\(x,xs) -> take x xs) (zip [4..6] [suite1,suite2,suite3]))
```

produit, en `Haskell`, une valeur. Pouvez vous deviner quelle est cette valeur ? Donnez une justification détaillée de votre réponse. (Fin question 5.2)

Faculté des Sciences
Aix-Marseille Université

Année universitaire 2014/2015

Site : ☐ Luminy ☐ St-Charles ☐ St-Jérôme ☒ Cht-Gombert ☐ Aix-Montperrin ☐ Aubagne-SATIS
Sujet de : ☐ 1er semestre ☐ 2ème semestre ☒ Session 2 Durée de l'épreuve : 2h
Examen de : M1 Nom du diplôme : Master Informatique
Code du module : ENSINAU4 Libellé du module : Programmation fonctionnelle
Calculatrices autorisées : NON Documents autorisés : OUI, notes de Cours/TD/TP

Remarque : cet examen présente une choix d'exercices et questions ; il n'est pas nécessaire, afin d'obtenir la note maximum, de tous les résoudre.

Exercice 1. Écrivez le type (n'oubliez pas les contraintes de classe) de chaque expression définie dans le script suivant :

```
succ = \x -> x + 1
thrice f = \x ->  f (f (f x))
geom a q n = take n (xs a q) where xs a q = a:(xs (a*q)  q)
graph f a b n = [ (x,f x) | x <- [g i | i <- [0..n-1] ] ]
    where  g i = a + fromIntegral i*(b-a)/fromIntegral (n-1)
```

Exercice 2 : types et constructeurs, classes et methodes. Considérez le script `Haskell` suivant :

```
1  class MyNum a where
2      zero :: a
3  newtype MyInt = MyInt Int deriving Show
4  instance MyNum MyInt where
5      zero = MyInt 0
6  newtype MyReal = MyReal Float deriving Show
7  instance MyNum MyReal where
8      zero = MyReal 0
9  if1 = if True then show (zero::MyInt) else show (zero::MyReal)
10 sum x y = MyReal (x + y)
11 mult (MyReal x) (MyReal y) = MyReal (x*y)
12 power x (MyInt 0) = MyReal 1
13 power x (MyInt n) = mult (power x (MyInt (n-1))) x
```

1. Listez les noms des *constructeurs* qui apparaissent dans ce script ; à coté de chaque nom de constructeur de votre liste, écrivez les numéros de ligne où ce constructeur est utilisé dans le script.
2. Listez les noms des *types* qui apparaissent dans ce script ; à coté de chaque nom de type dans votre liste, écrivez le numéros de ligne où le type apparaît explicitement dans le script.
3. Listez les noms des *classes* qui apparaissent dans ce script ; à coté de chaque nom de classe de votre liste, écrivez le numéro de ligne où cette classe apparaît explicitement dans le script.
4. Quels sont les *méthodes* de classe qui apparaissent dans ce script ? Listez–les, et à coté de chaque méthode classe de votre liste, écrivez le numéro de ligne où cette méthode apparaît dans le script.

Exercice 3 : λ-calcul, évaluation. Considérez l'évaluation suivante :

$$
\begin{aligned}
1 \quad (\lambda x.\lambda f.(f\,x))\,z\,((\lambda g.\lambda y.(g\,y))\,w) &\rightsquigarrow (\lambda f.(f\,z))\,((\lambda g.\lambda y.(g\,y))\,w) \\
2 \quad &\rightsquigarrow ((\lambda g.\lambda y.(g\,y))\,w)\,z \\
3 \quad &\rightsquigarrow (\lambda y.(w\,y))\,z \\
4 \quad &\rightsquigarrow w\,z\,.
\end{aligned}
$$

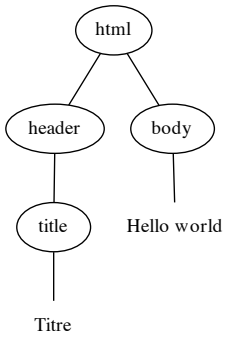
Question 3.1. Combien de β-réductions il y a dans cette évaluation ? (Fin question 3.1)

Question 3.2. Dire, pour chaque β-réduction, quel est le redex utilisé pour la réduction. (Fin question 3.2)

Question 3.3. Quelle est la stratégie d'évaluation utilisée ? Justifiez soigneusement votre réponse (une réponse sans justification ne sera pas appréciée). (Fin question 3.3)

Exercice 4 : Types récursif. Un document `html` peut se représenter en forme arborescente. Par exemple, le document suivant sur la gauche peut se représenter par l'arbre sur la droite :

```
<html>
  <header>
    <title>Titre</title>
  </header>
  <body>
    Hello world
  </body>
</html>
```



Une première définition d'un type de donnée `Haskell` pour représenter les documents `html` est la suivante :

```
data Tag = Html | Header | Title | Body
data HtmlDoc = Node Tag [HtmlDoc] | Leaf String
```

Question 4.1. Écrivez une expression `Haskell` de type `HtmlDoc` qui représente le document `html` ci-dessus. (Fin question 4.1)

Question 4.2. Écrivez une fonction `showHtmlDoc` qui prend en paramètre un objet de type `HtmlDoc` et construit la représentation usuelle d'un tel document comme chaîne de caractères. (Fin question 4.2)

Question 4.3. Modifiez la définition des types afin que chaque *tag* puisse porter l'information d'une suite de couples attributs/valeurs [1]. (Fin question 4.3)

Question 4.4. Modifiez la définition du type `HtmlDoc`, afin qu'on puisse insérer d'autres *tags* entre le texte d'une feuille. Par exemple, il sera possible de représenter le code `html` suivant :

```
<body>La solution est <a>ici</a> et non par <a>la</a>.</body>
```

(Fin question 4.4)

Exercice 5 : intégrales. On vous demande, dans cet exercice, d'écrire une fonction `integral` qui, étant donnée en entrée une fonction réelle f, calculera l'intégrale

$$\int_a^b f(x)d(x)\,.$$

Il s'agit, évidemment, de calculer une approximation de cette intégrale, de la forme (par exemple)

$$\sum_{i=1}^{n} f(a+i\delta_n)\delta_n \quad \text{où} \quad \delta_n = \frac{b-a}{n}\,.$$

Tout est à vous : le choix des paramètres (écrivez le type de la fonction `integral` explicitement), le code qui réalise ce calcul.

1. Par exemple, dans `<a href="http://blabla.com" target=''top'' >...</a>`, le *tag* `a` porte les attributs `href` et `target` dont les valeurs respectifs sont entre guillemets.

Faculté des Sciences
Aix*Marseille Université

Année universitaire 2015/2016

Site : ☒ Luminy ☐ St-Charles ☒ St-Jérôme ☐ Cht-Gombert ☐ Aix-Montperrin ☐ Aubagne-SATIS
Sujet de : ☐ 1er semestre ☐ 2ème semestre ☒ Session 2 Durée de l'épreuve : 2h
Examen de : M1 Nom du diplôme : Master Informatique
Code du module : ENSINAU4 Libellé du module : Programmation fonctionnelle
Calculatrices autorisées : NON Documents autorisés : OUI, notes de Cours/TD/TP

Exercice 1 : Typage, types et constructeurs, classes et methodes. Considérez le script `Haskell` suivant :

```
1  type Resultat a = Maybe (a,String)
2  newtype Parser a = MkParser (String -> Resultat a)
3  parse (MkParser f) = f
4  p >>> q =  MkParser pThenQ where
5      pThenQ = \inputString -> case parse p inputString of
6          Just (resultP, inputStringLeft) -> parse (q resultP) inputStringLeft
7          _ -> Nothing
8  instance Monad Parser where
9      (>>=) = (>>>)
10     return x = MkParser (\str -> Just(x,str))
11     fail _ = MkParser (\_ -> Nothing)
12 anyChar = MkParser getHead where
13     getHead (x:xs) = Just (x,xs)
14     getHead _ = Nothing
15 thisChar c = anyChar >>= \x ->
16     if x == c then return x else fail ""
```

1. (1 pts). Listez les noms des *constructeurs* qui apparaissent dans ce script ; écrivez, à coté de chaque nom de constructeur, les numéros des lignes où ce constructeur est utilisé dans le script.
2. (1 pts). Listez les noms des *types* qui apparaissent dans ce script ; écrivez, à coté de chaque nom de type, le numéros des lignes où le type apparaît explicitement dans le script.
3. (2 pts). Quels sont les *méthodes* de classe qui apparaissent dans ce script ? Listez–les et, à coté de chaque nom de méthode, écrivez le nom de la classe de la méthode et les numéros des toutes les lignes où la méthode apparaît dans le script.
4. (3 pts). Écrivez le type de `(>>>)`, `anyChar` et `thisChar`. Pour le dernier, justifiez avec précision votre réponse.

Exercice 2 : Types récursifs. Rappelons (voir le TD no. 5) un type de données `Haskell` adapté à coder les λ-termes du λ-calcul non typé :

```
type Variable = String
data LambdaTerm = Var Variable
    | Appl LambdaTerm LambdaTerm
    | Abstr Variable LambdaTerm
```

Question 2.1. (1 pts). Dessinez l'arbre abstrait du λ-terme $(\lambda x.x\,(\lambda y.x\,y))\,z\,z$. (Rappelez-vous des règles de priorité et associativité de l'opérateur d'application). (Fin question 2.1)

Question 2.2. (2 pts). Donnez ensuite une expression `Haskell` de type `LambdaTerm` représentant ce λ-terme. (Fin question 2.2)

Question 2.3. (3 pts). Dans la définition suivante de la substitution (d'une variable par un λ-terme dans un λ-terme) on ignore—pour simplicité—le problème de la capture des variables (c'est-à-dire le fait qu'une variable libre puisse devenir liée après substitution) :

$$x[r/y] = \begin{cases} r, & \text{si } y = x, \\ x, & \text{sinon,} \end{cases} \qquad (\,t\ s\,)[r/y] = t[r/y]\ s[r/y]\,, \qquad (\lambda x.t)[r/y] = \lambda x.(\,t[r/y]\,).$$

Donnez une définition de cette fonction en `Haskell`. Cette fonction, nommée `subst`, aura le type suivant :

```
subst :: LambdaTerm -> Variable -> LambdaTerm -> LambdaTerm
```

(Fin question 2.3)

Exercice 3 : Évaluation en Haskell. (3 pts). Considérez les deux expressions définies comme suit :

```
expr1 = take 100 [1..1000]
expr2 = take 99 [1..1000000000000000000000000000000]
```

On demande à `ghci` d'évaluer ces deux expressions. Selon vous, quelle est l'expression qui s'évalue en moins de temps ? Justifiez avec précision votre réponse.

Exercice 4 : Autour des nombres premiers. On souhaite, avec cet exercice, écrire un script `Haskell` pour calculer le décomposition en produit de facteurs premiers d'un nombre naturel (nombre entier plus grand ou égal à 0) passé en paramètre, ainsi que utiliser cette décomposition pour représenter des listes de nombres naturels par un nombre naturel. On procédera comme suit.

Question 4.1. (2 pts). Écrire une fonction `pptDiviseur :: Integer -> Integer` qui retourne le plus petit diviseur plus grand que 1 d'un nombre passé en paramètre. (Fin question 4.1)

Question 4.2. (2 pts). Écrire une fonction `decomposer :: Integer -> [Integer]` qui construit la liste des diviseurs premiers (avec multiplicités) d'un nombre n selon la méthode suivante : la tête de la liste retournée est le plus petit diviseur d de n ; sa queue est la liste représentant la décomposition du nombre n/d. (Fin question 4.2)

Par la suite, nous supposons définies ces deux fonctions

```
niemePremier :: Integer-> Integer
-- retourne le n-ième nombre premier

indice :: Integer -> Integer
-- si n est premier, il retourne l'indice de n
```

L'indice d'un nombre premier est sa position dans la liste des nombres premiers triée par ordre croissant. Ainsi 2 a pour indice 0, 3 a pour indice 1, 5 a pour indice 2, 7 a pour indice 3... Autrement dit : 2 est le 0e nombre premier, 3 est le 1er nombre premier, 5 est le 2e nombre premier, 7 est le 3e nombre premier, ...on compte à partir de 0. Nous exploitons maintenant la fait que la décomposition en produit de facteurs premiers est unique pour coder (et de façon inverse, décoder) des listes triées de nombres naturels comme des entiers. Par exemple : le nombre 90 code la liste $[0, 1, 1, 2]$ car

$$90 = 2 \times 3 \times 3 \times 5 \quad \text{et}$$

2 a pour indice 0, 3 a pour indice 1, 5 a pour indice 2.

Question 4.3. (1 pts). Écrire une fonction `coder :: [Integer] -> Integer` qui associe à une liste de nombres naturels le produit des nombres premiers indexés par les nombres de la liste. Par example, on aura `coder [0,1,1,2] = 90`. (Fin question 4.3)

Question 4.4. (1 pts). Écrire une fonction `decoder :: Integer -> [Integer]` qui associe à un nombre n la liste des entiers dont le produit des premièrs correspondants est égal à n. Par example, on aura `decoder 90 = [0,1,1,2]`. (Fin question 4.4)

Pour finir :

Question 4.5. (3 pts). Pouvez vous écrire les définitions des fonctions `niemePremier` et `indice` ? Pour ce faire, on pourra supposer que l'on dispose de `premiers :: [Integer]`, la liste de tous les premiers. (Fin question 4.5)

Points totaux : 25

Faculté des Sciences
Aix*Marseille Université

Année universitaire 2015/2016

Site : ☒ Luminy ☐ St-Charles ☒ St-Jérôme ☐ Cht-Gombert ☐ Aix-Montperrin ☐ Aubagne-SATIS
Sujet de : ☐ 1er semestre ☐ 2ème semestre ☒ Session 2 Durée de l'épreuve : 2h
Examen de : M1 Nom du diplôme : Master Informatique
Code du module : ENSINAU4 Libellé du module : Programmation fonctionnelle
Calculatrices autorisées : NON Documents autorisés : OUI, notes de Cours/TD/TP

Exercice 1 : compréhension, typage. (7 pts). Le script suivant est une implémentation (incomplète) en `Haskell` de l'algorithme de Davis-Putnam-Longemann-Loveland (pour la satisfiabilité d'un ensemble de clauses de la logique propositionnelle) :

```
1  type Variable = Int
2  data Litteral = Positif Variable | Negatif Variable deriving Eq
3  neg :: Litteral -> Litteral
4  neg (Positif n) = Negatif n
5  neg (Negatif n) = Positif n
6  newtype Clause = Clause [Litteral] deriving Eq
7  faux = Clause []
8  type Clauses = [Clause]

10 dPLL :: Clauses -> Bool
11 dPLL [] = True
12 dPLL clauses = not (elem faux clauses)
13                &&
14                case chercherClauseUnitaire clauses of
15                  Just l -> dPLL (affecterAVrai l clauses)
16                  Nothing ->
17                    let l = choisirLitteral clauses in
18                      dPLL (affecterAVrai l clauses)
19                      || dPLL (affecterAVrai (neg l) clauses)
```

Question 1.1. (Sous-pointage : 2 points). Quels sont les nouveaux noms de types définis dans ce script ? Quels sont les nouveaux types définis dans le script ? Quels sont les constructeurs définis dans le script ?

(Fin question 1.1)

Question 1.2. (Sous-pointage : 2 points). Dans le script, les types `Litteral` et `Clause` sont instanciés à une classe.

1. De quelle classe s'agit et à quelles lignes ces instanciations sont faites ?
2. Pour quelle raison on a besoin de faire ces instanciations ? (Fin question 1.2)

Question 1.3. (Sous-pointage : 3 points). La fonction `chercherClauseUnitaire` recherche dans une liste de clauses la première clause ayant exactement un seul littéral.

1. Inférerez, à partir du script, le type exacte de cette fonction (écrivez le type de la fonction).
2. Écrivez la définition de cette fonction. (Fin question 1.3)

Exercice 2 : types récursifs. (6 pts). Un *arbre binaire* est ou bien une feuille, ou bien un noeud ayant exactement deux enfants (l'un à gauche, l'autre à droite) ; chaque enfant est lui même un arbre binaire.

Question 2.1. Définissez en `Haskell` un type de données, nommé `ArbreBin`, pour coder les arbres binaires.

(Fin question 2.1)

Question 2.2. Utilisez la réponse à la question précédente pour écrire une expression qui code l'arbre binaire sur la gauche de la figure 1. (Fin question 2.2)

Un *arbre à branchement fini* est un noeud ayant une liste ordonnée (de gauche à droite) d'enfants ; chaque enfant est lui même un arbre à branchement fini, il est possible que la liste d'enfants soit vide.

Question 2.3. Définissez en `Haskell` un type de données, nommée `ArbreBrf`, pour coder les arbres à branchement fini. (Fin question 2.3)

Question 2.4. Utilisez la réponse à la question précédente pour écrire une expression qui code l'arbre à branchement fini sur la droite de la figure 1. (Fin question 2.4)

On peut transformer un arbre binaire en un arbre à branchement fini, récursivement, de la façon suivante. Une feuille est transformée dans un noeud ayant une liste vide d'enfants ; un noeud avec deux enfants g et d (pour gauche et droite) est traité comme suit : on applique la procédure à g et d pour obtenir deux arbres à branchement fini l et r ; on retourne l'arbre qui est un noeud et dont la liste des enfants est obtenue de la liste des enfants de r en lui ajoutant en tête l'arbre l.

Par exemple, l'arbre à branchement fini en figure 1 sur la droite est obtenu de l'arbre binaire sur la gauche.

Question 2.5. Implémentez cette transformation en `Haskell` en écrivant la définition d'une fonction

```
arbreBintoArbreBrf ::
      ArbreBin -> ArbreBrf
```

(Fin question 2.5)

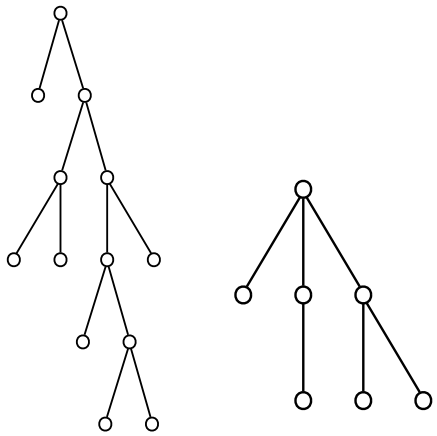


FIGURE 1 – Arbre binaire (gauche) et arbre à branchement fini (droite)

Exercice 3 : λ-calcul, évaluation. (4 pts). Considérez l'évaluation suivante :

$$
\begin{array}{ll}
1 & (\lambda x.\lambda f.(f\,x))\,z\,((\lambda g.\lambda y.(g\,y))\,w) \rightsquigarrow (\lambda f.(f\,z))\,((\lambda g.\lambda y.(g\,y))\,w) \\
2 & \rightsquigarrow ((\lambda g.\lambda y.(g\,y))\,w)\,z \\
3 & \rightsquigarrow (\lambda y.(w\,y))\,z \\
4 & \rightsquigarrow w\,z\,.
\end{array}
$$

Question 3.1. Combien de β-réductions il y a dans cette évaluation ? (Fin question 3.1)

Question 3.2. Dire, pour chaque β-réduction, quel est le redex utilisé pour la réduction. (Fin question 3.2)

Question 3.3. Quelle est la stratégie d'évaluation utilisée ? Justifiez soigneusement votre réponse (une réponse sans justification ne sera pas appréciée). (Fin question 3.3)

Exercice 4. (6 pts). Voici une possible définition de la classe des ensembles finis :

```
class EnsFin s where
    vide :: s a -- l'ensemble vide
    est_vide :: s a -> Bool --est l'esemble vide ?
    singleton :: a -> s a --le signleton
    construire :: Eq a => [a] -> s a -- transforme une liste en ensemble
    union :: Eq a => s a -> s a -> s a   -- operations Booleennes
    intersection :: Eq a => s a -> s a -> s a
    difference :: Eq a => s a -> s a -> s a
```

Instanciez à cette classe le type ainsi défini :

```
data Ens a = Ens [a]
```

Points totaux : 23

Faculté des Sciences
Aix*Marseille Université

Année universitaire 2016/2017 (le 12/12/2016)

Site : ☒ Luminy ☐ St-Charles ☒ St-Jérôme ☐ Cht-Gombert ☐ Aix-Montperrin ☐ Aubagne-SATIS
Sujet de : ☒ 1er semestre ☐ 2ème semestre ☐ Session 2 Durée de l'épreuve : 2h
Examen de : M1 Nom du diplôme : Master Informatique
Code du module : ENSINAU4 Libellé du module : Programmation fonctionnelle
Calculatrices autorisées : NON Documents autorisés : OUI, notes de Cours/TD/TP

Exercice 1. Dans ce script :

```
zip [] ys = ys
zip xs [] = xs
zip [x:xs] [y:ys] = x:y:xs++ys
iter f 0  = \x -> x
iter f n = \x -> iter f (n-1) f x
isSingleton xs
  | [x] = True
  | _ = False
```

chaque définition pose un problème, de syntaxe ou de type.
Pour chaque expression définie :

1. expliquez quel est le problème (soyez le plus précis possible),
2. corrigez la définition,
3. écrivez le type de l'expression définie (n'oubliez pas que la fonction pourrait être polymorphe ou surchargée).

Exercice 2. Un arbre binaire de recherche sert à stocker un ensemble de clés pour les rechercher ensuite en temps optimal. En supposant que les clés sont des entiers et les valeurs associées aux clés sont des chaînes de caractères, les arbres de recherche binaires peuvent se définir en Haskell par le type `Arbre` comme suit :

```
type Cle = Int
type Valeur = String
type ListeAssoc = [(Cle,Valeur)]
data Arbre = Feuille | Noeud Arbre (Cle,Valeur) Arbre
```

Donc, un tel arbre est ou bien une feuille (sans clé) ou bien un nœud interne (avec clé et valeur) avec un sous-arbre à gauche et un sous-arbre à droite. En plus, on suppose que *(**) la clé d'un nœud quelconque est plus grande que toutes les clés dans le sous-arbre à gauche et plus petite que toutes les clés dans le sous-arbre à droite.*

Question 2.1. Faites un dessin de l'arbre décrit par l'expression Haskell suivante :

```
Noeud
(Noeud Feuille (0,"b") Feuille)
(1,"d")
(Noeud (Noeud Feuille (2,"a") Feuille) (3,"c") Feuille)
```

(Fin question 2.1)

Question 2.2. Implémentez les fonctions suivantes :

```
taille :: Arbre -> Int
estEquilibre :: Arbre -> Bool
```

```
chercher :: Cle -> Arbre -> Maybe Valeur
ajouter :: (Cle,Valeur) -> Arbre -> Arbre
depuisListeAssoc :: ListeAssoc -> Arbre -> Arbre
```

Ici :

- `taille` calcule le nombre de clés dans un arbre ;
- `estEquilibre` détermine si un arbre est *équilibré*, c'est-à-dire, si pour tout nœud de l'arbre, le nombre de clés dans le sous-arbre à gauche et le nombre de clés dans le sous-arbre à droite diffèrent au plus de 1 ;
- `chercher` cherche la valeur associée à une clé dans un arbre ;
- `ajouter` ajoute un couple clé/valeur à un arbre binaire de recherche, en préservant la condition (**)
- `depuisListeAssoc` ajoute tous les couples clé/valeur d'une liste associative à un arbre.

(Fin question 2.2)

Exercice 3. Posons

$$\texttt{TRUE} := \lambda x.\lambda y.x\,, \qquad \texttt{FALSE} := \lambda x.\lambda y.y\,, \qquad \texttt{AND} := \lambda p.\lambda q.p\,q\,p\,.$$

et considérez le λ-terme

$$\texttt{AND FALSE TRUE}\,.$$

Question 3.1. Dessinez l'arbre abstrait de ce λ-terme. (Fin question 3.1)

Question 3.2. Évaluez ce λ-terme d'abord jusqu'à obtenir un valeur, en utilisant l'ordre normal (par l'extérieur) et puis en utilisant l'ordre applicatif (par l'intérieur).
Avant chaque étape de l'évaluation, listez tous les redexes dans le terme à évaluer. (Fin question 3.2)

Question 3.3. Décrivez brièvement ce qui se passe si, en plus d'utiliser l'ordre normal (resp. l'ordre applicatif) on s'empêche d'évaluer sous un λ. C'est-à-dire, décrivez ce qui se passe si on utilise la stratégie call by name (resp. call by value) en évaluant ce terme. (Fin question 3.3)

Exercice 4. Considérez les deux fonctions définies ci-dessous :

```
join x [] = []
join x ys = foldr (\z w -> z++x++w) (last ys) (init ys)
pad n ys = map Just ys ++ replicate k Nothing
  where k = n - (length ys `mod` n)
```

Rappel :

```
last :: [a] -> a -- dernier element d'une liste
init :: [a] -> [a] -- la liste sans son dernier element
foldr :: (a -> b -> b) -> b -> [a] -> b
```

Question 4.1. Écrivez le type de ces deux fonctions. (Fin question 4.1)

Question 4.2. Expliquez ce qui est accompli par ces deux fonctions. Apportez des exemples à vos explications. (Fin question 4.2)

Question 4.3. En utilisant les fonctions `join` et `pad`, écrivez deux fonctions, `cutAndPad` et `formatText`, dont le type sera :

```
cutAndPad :: Int -> [a] -> [[Maybe a]]
formatText :: Int -> String -> String
```

de facon que :

- `cutAndPad n xs` divise une liste `xs` en une liste de listes, chacune de longueur `n`. Si la longueur de la dernière liste est moindre que `n`, cette liste sera complétée par des `Nothing`.
- `formatText n str` transforme la chaîne de caractères `str` en une autre chaîne où le même texte est organisé en lignes de `n` caractères.

(Fin question 4.3)

Faculté des Sciences
Aix*Marseille Université

Année universitaire 2016/2017 (juin 2017)

Site : ☒ Luminy ☐ St-Charles ☒ St-Jérôme ☐ Cht-Gombert ☐ Aix-Montperrin ☐ Aubagne-SATIS
Sujet de : ☐ 1er semestre ☐ 2ème semestre ☒ Session 2 Durée de l'épreuve : 2h
Examen de : M1 Nom du diplôme : Master Informatique
Code du module : ENSINAU4 Libellé du module : Programmation fonctionnelle
Calculatrices autorisées : NON Documents autorisés : OUI, notes de Cours/TD/TP

Exercice 1. Considérez le script `Haskell` suivant :

```
encode ::  String -> [(Char,[Int])]

encode [] = []
encode (c:cs) =
  let
    rest = [ x | x <- cs, x /= c ]
    positions = [ i | (x,i) <- zip (c:cs) [0 .. length cs], x == c]
  in
    (c,positions):encode rest

insertAtPositions :: (Char,[Int]) -> String -> String
insertAtPositions (c,[]) ys = ys
insertAtPositions (c,(x:xs)) ys =
  insertAtPositions (c,xs) (take x ys ++ [c] ++ drop x ys)

decode :: [(Char,[Int])] -> String
decode [] = []
decode (keyValues:ys) = insertAtPositions keyValues (decode ys)
```

Question 1.1. Listez tous les méthodes de classe qui apparaissent dans ce script en mentionnant la classe à laquelle ils appartiennent. (Fin question 1.1)

Question 1.2. Listez toutes les motifs (patterns) qui apparaissent dans ce script en mentionnant la ligne où ils apparaissent. (Attention : la même expression, selon le contexte, peut ou non être un motif). (Fin question 1.2)

Question 1.3. Que donne l'évaluation de l'expression `encode "abracadabra''`? (Fin question 1.3)

Question 1.4. Que donne l'évaluation de l'expression
`decode [('a',[0,3,5,7,10]),('r',[0,4]),('b',[0,3]),('d',[0]),('c',[0])]` ? (Fin question 1.4)

Question 1.5. Redéfinissez la fonction `decode` en utilisant le schéma d'itération `foldr`.
Rappel : le type de `foldr` est `(a -> b -> b) -> b -> [a] -> b` (Fin question 1.5)

Exercice 2. Posons

$$\texttt{TRUE} := \lambda x.\lambda y.x\,, \qquad \texttt{FALSE} := \lambda x.\lambda y.y\,, \qquad \texttt{OR} := \lambda p.\lambda q.p\,p\,q\,.$$

et considérez le λ-terme

$$\texttt{OR FALSE TRUE}\,.$$

Question 2.1. Dessinez l'arbre abstrait de ce λ-terme. (Fin question 2.1)

Question 2.2. Évaluez ce λ-terme jusqu'à obtenir un valeur, d'abord en utilisant l'ordre normal (par l'extérieur) et ensuite en utilisant l'ordre applicatif (par l'intérieur).
Avant chaque étape de l'évaluation, listez tous les redexes dans le terme à évaluer. (Fin question 2.2)

Question 2.3. Décrivez brièvement ce qui se passe si, en plus d'utiliser l'ordre normal (resp. l'ordre applicatif) on s'empêche d'évaluer sous un λ. C'est-à-dire, décrivez ce qui se passe si on utilise la stratégie call by name (resp. call by value) en évaluant le terme `OR FALSE TRUE`. (Fin question 2.3)

Exercice 3. Considérez ce script `Haskell` :

```
data Expr a = Leaf Char | Node (Expr a) a (Expr a)
test = Node (Node (Leaf 't') '*' (Leaf 's'))
  '*' (Node (Leaf 'e') '*'  (Leaf 't'))

weight (Leaf _) = 0
weight (Node l x r) = 1 + weight l + x + weight r

foldExpr f  g (Leaf x) = f x
foldExpr f  g (Node l x r) = g (foldExpr f g l) x (foldExpr f g r)

word = foldExpr (\x -> [x]) (\xs _ zs -> xs++zs)

f x y = if weight x == 0 then Node x 1 (Leaf y)
         else if weight x == 1 then Node (Leaf y) 0 x
              else Leaf y
```

Question 3.1. Écrivez le type de chaque expression définie dans ce script (n'oubliez pas qu'une expression peut être polymorphe ou surchargée). (Fin question 3.1)

Question 3.2. Réécrivez la définition de la fonction `f` en utilisant des définitions avec des conditions de garde à la place des conditionnels. (Fin question 3.2)

Question 3.3. Réécrivez la définition de cette même fonction `f` en utilisant le filtrage par motifs à la place des conditionnels. (Aide : en supposant que toutes les étiquettes des `Noeud`s sont positives, alors `weight expr == 1` si et seulement si l'expression est un `Node` étiqueté par `0` avec deux fils qui sont des `Leaf`s.) (Fin question 3.3)

Question 3.4. Quel est le résultat de l'évaluation de l'expression `word test` ? (Fin question 3.4)

Question 3.5. Donnez une définition récursive explicite de la fonction `word` (donc sans utiliser le schéma récursif `foldExpr`). (Fin question 3.5)

Exercice 4. Dans cet exercice on vous demande de faire preuve de votre maîtrise du langage `Haskell` : des erreurs de syntaxe ou de type seront tenu en compte pour la correction.
Écrivez un script `Haskell` contenant :

- la définition d'un type `Pol`, pour représenter les polynômes à coefficients entiers,
- une fonction `eval : Pol -> Int -> Int`, qui évalue un polynôme en un entier donné,
- une fonction `toPol : Int -> Int -> Pol` telle que, appliquée à deux entiers n et m, construira un polynôme $P(x) = \sum_{i=0}^{k} a_i x^i$ tel que $P(n) = m$ et $0 \leq a_i < n - 1$, pour $i = 0, \ldots k$.

  C'est-à-dire, `toPol n m` construit la représentation en base n de l'entier m. Par exemple, `toPol 2 5` sera le polynôme $x^2 + 1$ (correspondant à la représentation binaire de 5 : 101) et `toPol 2 7` sera le polynôme $x^2 + x + 1$ (correspondant à la représentation binaire de 7 : 111).

Faculté des Sciences
Aix*Marseille Université

Année universitaire 2017/2018 (décembre 2017)

Site : ☒ Luminy ☐ St-Charles ☒ St-Jérôme ☐ Cht-Gombert ☐ Aix-Montperrin ☐ Aubagne-SATIS
Sujet de : ☒ 1er semestre ☐ 2ème semestre ☐ Session 2 Durée de l'épreuve : 2h
Examen de : M1 Nom du diplôme : Master Informatique
Code du module : ENSINAU4 Libellé du module : Programmation fonctionnelle
Calculatrices autorisées : NON Documents autorisés : OUI

Exercice 1 : Typage. La compilation du script suivant :

```
containsEmptyLine = any null
columns lines
  | containsEmptyLine lines = []
  | otherwise =
      let pairs = map (\xs -> (head xs,tail xs)) lines
      in
      fst pairs:columns (snd pairs)
padRight n x xs
  | l < n  = xs ++ replicate (n-l) x
  | otherwise = take n xs
    where l = length xs
safeColumns xss = columns (map (padRight m ' ') xss)
  where m = maximum (map length) xss
```

produit des erreurs de type. Ces erreurs sont contenues dans la définition de deux des quatre fonctions définies.

Question 1.1. Parmi les définitions de ces quatre fonctions, lesquelles sont responsables des erreurs ? (Fin question 1.1)

Question 1.2. Corrigez ces erreurs. (Fin question 1.2)

Question 1.3. Après correction des erreurs, écrivez le type de chaque fonction définie. (Fin question 1.3)

Question 1.4. Donnez un exemple d'une liste `xss` telle que `columns xss` et `safeColumns xss` ne donnent pas le même résultat. Quels sont ces deux résultats ? (Fin question 1.4)

Exercice 2 : Types, constructeurs, classes, methodes, patterns. Considérez le script suivant :

```
data List a = Nil | Cons a (List a) deriving (Eq,Show)
bubbleSort Nil = Nil
bubbleSort (Cons x xs) =  case bubbleSort xs of
  Nil -> Cons x Nil
  Cons y ys -> if x <= y then Cons x (Cons y ys)
               else Cons y (bubbleSort (Cons x ys))
```

Question 2.1. Combien et quels sont : (a) les types utilisés dans ce script ; (b) les noms de constructeurs apparaissant dans ce script ; (c) les noms de classes apparaissant dans ce script ; (d) les méthodes de classes utilisées dans le script ; (e) les classes utilisées dans ce script ; (f) les motifs (patterns) utilisés dans ce script. (Fin question 2.1)

Question 2.2. Quel est le type de la fonction `bubbleSort` ? (Fin question 2.2)

Question 2.3. Peut on remplacer le mot clés `data` par (a) `type` ? (b) `newtype` ? Justifiez vos réponses. (Fin question 2.3)

Exercice 3 : λ-calcul. Dans la suite de réductions suivantes :

$$
\begin{aligned}
&(\ (\lambda x.x)\ (\lambda y.y)\ )\ (\ (\lambda z.z)\ (\lambda v.(\ (\lambda u.u)\ v\ ))\ ) \\
&\qquad \rightsquigarrow\ (\lambda y.y)\ (\ (\lambda z.z)\ (\lambda v.(\ (\lambda u.u)\ v\ ))\ ) \\
&\qquad \rightsquigarrow\ (\lambda z.z)\ (\lambda v.(\ (\lambda u.u)\ v\ )) \\
&\qquad \rightsquigarrow\ \lambda v.(\ (\lambda u.u)\ v\ )
\end{aligned}
$$

on a complètement évalué du premier λ-terme de la suite au dernier λ-terme en utilisant une certaine stratégie d'évaluation que vous allez reconnaître.

Question 3.1. Combien et lesquels sont les redexes dans le deuxième λ-terme de la suite ? Lequel est utilisé pour faire la β-réduction ? (Fin question 3.1)

Question 3.2. Quelle est la stratégie utilisée dans cette suite de réductions : (a) la stratégie par l'intérieur ? (b) la stratégie par l'extérieur ? (c) la stratégie (ou appel) par valeur (call by value) ? (d) la stratégie (ou appel) par nom (call by name) ? Justifiez votre réponse. (Fin question 3.2)

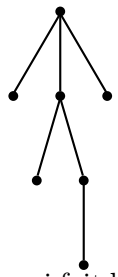
Question 3.3. Expliquez pourquoi, dans la question précédente, une seule réponse est correcte. (Fin question 3.3)

Question 3.4. Donnez un exemple d'une suite de réductions (à partir d'un λ-terme de votre choix) qui utilise à la fois la stratégie d'évaluation par l'intérieur et la stratégie d'évaluation par l'extérieur. (Fin question 3.4)

Exercice 4 : Types récursifs. Un arbre planaire est un noeud dont les enfants sont organisés dans une liste ; chaque enfant est lui aussi un arbre planaire. Une feuille est un noeud qui ne possède aucun enfant, donc sa liste d'enfants est vide. Le type des arbres planaires peut donc se définir en `Haskell` ainsi :

```
data APlanaire = Noeud [APlanaire]
```

Étant donné un arbre planaire, on obtient un mot bien parenthésé en faisant un parcours en profondeur gauche de l'arbre : on ouvre une parenthèse quand on visite pour la première fois un noeud, on la ferme quand on visite le noeud pour la deuxième et dernière fois.
Par exemple, un parcours profondeur gauche de l'arbre ci-dessous à droite produira le mot `(()(()(())))()`.



Question 4.1. Représentez l'arbre ci-dessus sur la gauche comme une expression `Haskell` de type `APlanaire`. (Fin question 4.1)

Question 4.2. Écrivez une fonction `arbreVersMot :: APlanaire -> String` qui, étant donnée en entrée un arbre planaire, produit un mot bien parenthésé selon l'algorithme décrit auparavant. (Fin question 4.2)

Question 4.3. Modifiez le type des arbres planaires donné ci-dessus de façon à pouvoir coder en `Haskell` des arbres planaires dont les noeuds sont étiquetés par un type générique `a`. Le nom du type sera `APLanaireE a`. (Fin question 4.3)

Question 4.4. Définissez une fonction `somme :: APlainaireE Int -> APlainaireE Int` qui fait la somme de tous les étiquettes des noeuds. (Fin question 4.4)

Question 4.5. Montrez que l'on peut définir les fonctions `arbreVersMots` et `somme` comme des instances d'un même algorithme général. C'est-à-dire, définissez une fonction

```
foldAPlanaireE :: (a -> [b] -> b) -> APlanaireE a -> b
```

qui calcule un résultat grâce à une fonction `f :: a -> [b] -> b` passée en paramètre, qui s'applique à la valeurs de l'étiquette et aux résultats des calculs récursif sur les fils. Ainsi, on aura :

```
arbreVersMot :: APlanaireE () -> Int
arbreVersMot = foldAPlanaireE f
  where f = \_ strs -> "(" ++ concat strs ++ ")"
somme :: APlanaireE Int -> Int
somme = foldAPlanaireE f
  where f = \n ns -> n+ sum ns
```

(Fin question 4.5)