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 :

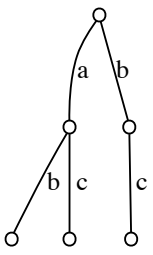
```
(&&), True, map, even, [0..], expr1, expr2.
```

(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 :

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

 4 ajouterMot :: EnsPref -> String -> EnsPref
 5 ajouterMot dict [] = dict
 6 ajouterMot (Noeud []) (x:xs) = Noeud [(x,ajouterMot (Noeud []) xs)]
 7 ajouterMot (Noeud ((c,d):ds)) (x:xs)
 8     | c == x = Noeud ((c,ajouterMot d xs):ds)
 9     | otherwise =
10         let
11             Noeud queue = ajouterMot (Noeud ds) (x:xs)
12         in Noeud ((c,d):queue)

14 ajouterMots :: EnsPref -> [String] -> EnsPref
15 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)