25/06/2013 — Programmation fonctionnelle
Durée : 2 heures — Master 1 Informatique
Documents autorisés

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

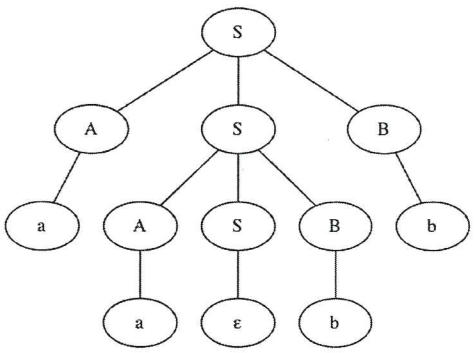
Exercice 2. Considérez la grammaire donnée par les quatre productions (ou règles) suivantes :

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

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.