Programmation fonctionnelle

1 Premier contact : quelques définitions et types

1.1 Définitions de fonctions

Q 1. On veut définir une fonction `sommeDeXaY` qui prend en arguments deux entiers et calcule la somme des entiers compris entre ses arguments.

Quels types peut-elle avoir ?

Donnez sa définition.

Q 2. Donnez une fonction qui calcule la somme des éléments d'une liste.

Q 3. Donnez une fonction qui calcule le produit des éléments d'une liste.

1.2 Types d'expressions

Q 4. Donnez les types des expressions suivantes :

```
['a', 'b', 'c']
[(False, '0'), (True, '1')]
([False, True], ['0', '1'])
(['a', 'b'], 'c')
[tail, init, reverse]
take 5
```

Même question après avoir remplacé toutes les parenthèses par des crochets et réciproquement. Par exemple `(['a', 'b'], 'c')` devient `[('a', 'b'), 'c']`. Sont-elles toutes encore bien typées ?

Q 5. Donnez les types des fonctions ainsi définies :

```
second xs     = head (tail xs)
swap (x, y)   = (y, x)
pair x y      = (x, y)
fst (x, _)    = x
double x      = x * 2
palindrome xs = reverse xs == xs
twice f x     = f (f x)
```

1.3 Formes curryfiée et décurryfiée d'une fonction

Q 6. Réalisez une fonction nommée `curryfie`

```
curryfie :: ((t1,t2) -> t) -> t1 -> t2 -> t
```

qui transforme une fonction f à deux paramètres sous forme décurryfiée en une fonction g équivalente curryfiée. Autrement dit, pour tout x et y (du bon type) on doit avoir

$$f\ (x, y) = g\ x\ y.$$

Q 7. Réalisez la fonction `decurryfie`, réciproque de la fonction précédente.

1.4 Un itérateur

Q 8. Réalisez une fonction `itere :: (t -> t) -> Int -> t -> t` telle que la valeur de l'expression `itere f n x` est égale à $f(f(f \ldots (f\ x) \ldots))$, le nombre de f étant égal à n.

Par exemple `itere (\x -> x + 1) 10 0` sera égal à 10.

1.5 Expressions fonctionnelles polymorphes

Q 9. Définissez des fonctions ayant les types suivants :

```
— t -> t
— a -> b -> a
— (a -> b -> c) -> b -> a -> c
— a -> b -> b
— (t1 -> t) -> t1 -> t
— (t2 -> t1) -> (t1 -> t) -> t2 -> t
— Eq a => a -> a -> Bool
— Num a => t -> a
— Eq a => (t -> a) -> (t1 -> a) -> t -> t1 -> Bool
— (Eq a, Num a1, Num a2) => (a -> a2 -> a1) -> a -> a -> a1
— (t2 -> t1 -> t) -> (t2 -> t1) -> t2 -> t
— t -> t1
```

1.6 Expressions fonctionnelles numériques

Q 10. Définissez des fonctions ayant les types suivants :

```
— Num a => a -> a -> a
— (Num a, Num a1) => (a1 -> a) -> a
— (Num a) => a -> (a -> a)
— Num a => a -> a -> a -> a
— Num a => a -> (a -> a) -> a
— Num a => (a -> a) -> a -> a
— (Num a, Num a1, Num a2) => (a1 -> a2 -> a) -> a
```

1.7 Typage d'expressions fonctionnelles polymorphes

Q 11. Expliquez pourquoi le type des expressions fonctionnelles qui suivent ne dépend pas de l'environnement dans lequel elles sont évaluées, et déterminez ce type.

```
— h1 f x y = f x y
— h2 f x y = (f x) y
— h3 f x y = f (x y)
— h4 f x y = f (x y f)
— h5 f x y = (f x) (y f)
— h6 f x y = (f x) + (f y)
— h7 f x y = x f (f y)
— h8 f x y = f (f x y)
```

2 Quelques fonctions

2.1 Retour sur le TP 1, encore des listes et de la récursivité…

Q 12. Re-programmez les fonctions standard `last` et `init` en utilisant uniquement les fonctions `head`, `tail`, `!!`, `take`, `drop`, `length`, `++`, `reverse`.

Q 13. Re-programmez les fonctions standard `!!`, `++`, `concat`, `map` (en leur donnant un autre nom pour éviter le conflit avec les fonctions du `Prelude`) en n'utilisant que du filtrage de motifs.

Q 14. Définissez une fonction `myReverse` qui renvoie le miroir d'une liste (le premier élément devient le dernier, etc.). Par exemple, `myReverse [1,2,3,4]` s'évaluera en `[4,3,2,1]`.

Q 15. Définissez une fonction `myButLast` qui renvoie l'avant-dernier élément d'une liste. Par exemple, `myButLast [1,2,3,4]` s'évaluera en `3`.

Q 16. Définissez une fonction `compress` qui élimine les éléments redondants consécutifs d'une liste. Par exemple, `compress [2,2,2,1,3,3,4,4,4,4,1]` s'évaluera en `[2,1,3,4,1]`.

2.2 Expressions fonctionnelles

Q 17. Que calculent les expressions suivantes ?

```
(\x -> x * 2) 3
(\x -> \y -> x + y) ((\x -> x + x) 1) 1
(\x -> x 3) (\x -> x * 2)
(\x -> x) (\x -> x) 1
(\x -> x 1) (\x -> x)
```

Une des forces de la programmation fonctionnelle est de permettre de combiner des fonctions, pour en créer de plus complexes, sans avoir à appliquer ces fonctions à des arguments. Le plus simple des combinateurs est la composition de deux fonctions.

Q 18. Définissez et donnez le type de la fonction `compose` équivalente à `(.)` (sans utiliser `.` !).

2.3 Vol de suite cyclique

On veut étudier la fonction somFac qui à chaque entier associe la somme des factorielles de ses chiffres, par exemple :

- somFac$(156) = 1! + 5! + 6! = 1 + 120 + 520 = 841$,
- somFac$(5) = 5! = 120$
- somFac$(4985) = 4! + 9! + 8! + 5! = 24 + 362880 + 40320 + 120 = 403334$.

Q 19. Écrivez la fonction somFac.

Q 20. Comment calculeriez-vous tous les nombres tels que $n = \text{somFac}(n)$? (Il n'est pas nécessaire d'aller plus loin que 9999999 : pouvez-vous le prouver ?)

On peut répéter plusieurs fois l'opération somFac, par exemple :

$$132 \xrightarrow{\text{somFac}} 9 \xrightarrow{\text{somFac}} 362880 \cdots \xrightarrow{\text{somFac}} 169 \xrightarrow{\text{somFac}} 363601 \xrightarrow{\text{somFac}} 1454 \xrightarrow{\text{somFac}} 169 \xrightarrow{\text{somFac}} \cdots$$

On appelle *vol* de n la suite des entiers construits par applications successives de la fonction à un entier n.

$$\text{vol}(132) = [132, 9, 362880, 81369, 403927, \cdots]$$

Q 21. Écrivez une fonction qui retourne la suite (potentiellement infinie) des applications successives de somFac à un entier.

Tous les vols finissent par boucler sur un cycle (voir l'exemple de 132).

Q 22. Écrivez une fonction qui repère le début d'un cycle dans un vol (c'est-à-dire la première fois où on retrouve une valeur déjà présente dans le début du vol).

Q 23. Écrivez une fonction qui calcule tous les cycles des entiers de 1 à n.

Q 24. Comment généraliseriez-vous votre code pour qu'il traite toute fonction finalement cyclique (cf. suite de Syracuse, etc.) ?

3 Types algébriques

3.1 Pliages

On rappelle que `foldr` est défini ainsi :

```
foldr _  v []     = v
foldr op v (x :xs) = x `op` foldr op v xs
```

et `foldl` ainsi :

```
foldl _  v []     = v
foldl op v (x :xs) = foldl op (v `op` x) xs
```

Q 25. Donnez deux implémentations de la fonction `sum` l'une à l'aide de `foldr` et l'autre à l'aide de `foldl`. Pour chacune d'elles dépliez la liste des appels récursifs pour l'évaluation du terme `sum [1,2,3]`.

Q 26. Redéfinissez les fonctions `(++)` et `concat` à l'aide d'un *fold*.

Q 27. Redéfinissez les fonctions `map` et `filter` à l'aide de `foldr`.

Q 28. En utilisant `foldl` définissez une fonction `dec2int ::[Int]->Int` qui convertit une liste de chiffres en un entier. Par exemple,

```
> dec2int [2, 3, 4, 5]
2345
```

On considère les fonctions `unfold` et `int2bin` suivantes :

```
unfold :: (t -> Bool) -> (t -> a) -> (t -> t) -> t -> [a]
unfold p h t x | p x       = []
               | otherwise = h x  : unfold p h t (t x)

int2bin = unfold (== 0) (`mod` 2) (`div` 2)
```

Q 29. Que fait la fonction `int2bin` ? Implémentez la fonction `iterate` à l'aide de `unfold`.

3.2 Type de données algébriques pour les piles

Q 30. Définissez le type des structures de piles `Pile a`.

Q 31. Définissez les fonctions suivantes :

```
estVide :: Pile a -> Bool
sommet  :: Pile a -> Maybe a
depiler :: Pile a -> Pile a
```

Q 32. Définissez les fonctions suivantes :

```
empiler      ::  a  -> Pile a -> Pile a
empilerTout  :: [a] -> Pile a -> Pile a
empilerTout' :: [a] -> Pile a -> Pile a
```

telles que `empilerTout [1,2,3]` s'évalue en une pile ayant `1` au sommet alors que `empilerTout' [1,2,3]` aura `3` au sommet. Donnez deux versions de ces fonctions : l'une récursive et l'autre qui utilise un pliage.

3.3 Expressions arithmétiques et notation polonaise

On souhaite représenter les expressions arithmétiques sous forme d'*arbres de syntaxe abstraite* c'est-à-dire des arbres (ici binaires) dont les feuilles contiennent des valeurs (par exemple flottantes) et les nœuds un opérateur (binaire : `+`, `-`, `*`, `/`). L'avantage d'une telle représentation est qu'elle ne nécessite pas de connaître les priorités relatives des différents opérateurs pour pouvoir interpréter l'expression en l'absence de parenthèses (contrairement à la représentation littérale).

Q 33. Proposez un type de données algébriques nommé `Expression` pour représenter de telles expressions.

Q 34. Définissez la fonction `tree2string :: Expression -> String` qui convertit une représentation arborescente en représentation littérale.

Q 35. Définissez la fonction `eval :: Expression -> Float` qui évalue une expression arithmétique. Donnez une implémentation récursive et une implémentation qui utilise un pliage préalablement défini pour ce type de données.

Une autre représentation non ambiguë des expressions arithmétiques (binaires) est la *notation polonaise* qui empile les opérateurs sur leurs arguments. L'expression `(2 + 3) * 4` est ainsi représentée par `* + 2 3 4` où les parenthèses ne sont plus nécessaires.

Q 36. Définissez le type `ExpressionNP` des expressions représentées en notation polonaise.

Q 37. Définissez une fonction `parse :: ExpressionNP -> Expression` qui donne l'arbre de syntaxe abstraite d'une expression en notation polonaise.

4 Type `Maybe` et fonctions « sûres »

Nous voulons définir des fonctions « sûres », qui ne peuvent pas déclencher une exception même si leurs arguments sont incohérents. Par exemple, lorsque `(!!)` reçoit en argument un indice supérieur à la longueur de la liste, nous obtenons :

```
ghci> [0..10] !! 11
*** Exception : Prelude.!!: index too large
```

Nous allons pour cela utiliser le type `Maybe` : nous aurons ainsi une fonction sûre `atSafe` :

```
ghci> [0..10] `atSafe` 5
Just 5
ghci> [0..10] `atSafe` 11
Nothing
```

Q 38. Donnez le type de la fonction `atSafe` et définissez-la.

Q 39. Définissez une fonction `tailSafe` qui retourne `Nothing` pour la liste vide, et `Just ...` avec la queue de la liste sinon.

Q 40. Donnez le type de la fonction `minimumSafe` qui calcule le minimum d'une liste et définissez-la.

Vous pourrez utiliser la fonction `min` qui prend deux arguments et retourne le plus petit des deux.

La bibliothèque standard définit une fonction `foldr1` telle que :

```
foldr1 f [x1,x2,x3] = x1 `f` (x2 `f` x3)
```

Cette fonction lève une exception si la liste est vide.

Q 41. Donnez le type de `foldr1` et définissez-la.

Q 42. Définissez une variante sûre `foldr1Safe` de `foldr1`.

Q 43. Redéfinissez `minimumSafe` en utilisant `foldr1Safe` et `min`.

5 Analyse syntaxique

Le mini-haddock contient notamment la documentation sur le module `Parser` défini en cours.

Q 44. Expliquez le type `Parser`.

Q 45. Expliquez l'évaluation des expressions suivantes et donnez leur résultat :

```haskell
runParser empty "haskell"
runParser (pure 5) "haskell"
runParser unCaractereQuelconque "haskell"
runParser (empty <|> pure 5) "haskell"
runParser (pure 5 <|> empty) "haskell"
runParser (empty >>= \_ -> pure 5) "haskell"
runParser (pure 5 >>= \_ -> empty) "haskell"
runParser (unCaractereQuelconque >>= \c ->
           unCaractereQuelconque >>= \c' ->
           pure [c',c]) "haskell"
```

Q 46. Donnez le type d'un parseur qui sert à consommer tous les espaces au début de l'entrée et définissez-le. *(Penser à une fonction `trim`).*

Q 47. Donnez le type et définissez un parseur qui reconnaît les nombres entiers. Vous envisagerez des variations usuelles (signé ou non, décimal ou autre base, accepte un préfixe d'espaces ou pas, etc.).

5.1 Analyse de listes et d'arbres

On peut convertir un caractère qui est un chiffre en l'entier correspondant :

```haskell
intOfChar :: Char -> Int
intOfChar c = read [c]
```

Q 48. Définissez un parseur qui retourne la somme d'une suite de chiffres.

On représente des arbres binaires sous forme textuelle de la façon suivante : `f` est une feuille, `(f,(f,f))` est un arbre ayant une feuille `f` pour sous-arbre gauche et `(f,f)` pour sous-arbre droit.

Q 49. Définissez un parseur `hauteurArbre` qui retourne la hauteur d'un arbre.

Q 50. Définissez un parseur d'expressions arithmétiques écrites en notation polonaise. On supposera que l'expression ne comporte que des entiers et que opérateurs et entiers sont séparés d'un espace. Par exemple, l'expression `"- 23 + 2 4"` donnera, après analyse et évaluation, `17.0`.

On représente des arbres *au plus* binaires sous forme textuelle de la façon suivante : `f` est une feuille, `(f)` est un arbre unaire, `((f),(f,f))` est un arbre ayant un arbre unaire `(f)` pour sous-arbre gauche et un arbre binaire `(f,f)` pour sous-arbre droit.

Q 51. Définissez un parseur `hauteurArbre'` qui retourne la hauteur d'un arbre au plus binaire.

6 Interprète

Nous désirons faire un petit programme interactif `echo` affiche une `invite>` puis attendant que l'utilisateur entre un message. Lorsque ce message est entré (avec un retour à la ligne final), ce message est raffiché. Lorsque l'entrée standard est fermée (l'utilisateur tape Ctrl+D), le programme se termine proprement. Par exemple, nous pourrons avoir la session :

```
$ ./echo
invite> test
Vous avez entré  : «  test  »
invite> ^D
$
```

Q 52. Implémentez la commande `echo`.

Vous pourrez aussi ajouter le traitement d'une entrée particulière pour arrêter le programme (`:quit`, comme GHCi, ou autre).

7 Manipulations de matrices

Quand elle est possible, la meilleure approche pour résoudre un problème est d'utiliser les structures de données standard (en particulier les listes) et leurs fonctions associées (`map`, `zip`, `foldr`, etc.). Le code est alors plus court (on évite de récrire encore une fois les fonctions de parcours, …) et plus efficace car la bibliothèque standard est conçue pour que ces opérations soient les plus optimisées possible.

Nous allons nous exercer à cette façon de programmer sur des fonctions de manipulation de matrices. Nous allons ainsi considérer des vecteurs représentés comme des listes de nombres, et des matrices représentées par des listes de vecteurs (un pour chaque ligne de la matrice).

On pourra par exemple avoir les vecteurs `[1,2,0]` et `[4,3,-1]`, et la matrice `[[1,2,0],[4,3,-1]]` ayant 2 lignes et 3 colonnes.

Q 53. Complétez les déclarations de synonymes de types :

```
type Vecteur …
type Matrice …
```

Q 54. Définissez une fonction

```
scalaireVecteur :: Num a => a -> Vecteur a -> Vecteur a
```

qui multiplie chaque composante d'un vecteur par un nombre donné.
Par exemple `scalaireVecteur 2.5 [-2,-1,0,1,2]` donne comme résultat `[-5.0,-2.5,0.0,2.5,5.0]`.

Q 55. Définissez une fonction

```
plusVecteur :: Num a => Vecteur a -> Vecteur a -> Vecteur a
```

qui additionne deux vecteurs de même taille, c'est-à-dire additionne leurs composantes deux à deux. Par exemple `plusVecteur [1,2,3] [6,-4,3]` donne comme résultat `[7,-2,6]`.

Q 56. Définissez une fonction

```
plusMatrice :: Num a => Matrice a -> Matrice a -> Matrice a
```

qui additionne deux matrices de même taille, c'est-à-dire additionne leurs lignes deux à deux. Par exemple

```
plusMatrice [[1,1],[3,0],[1,0]] [[0,7],[0,5],[5,0]]
```

donne comme résultat `[[1,8],[3,5],[6,0]]`.

Q 57. Définissez une fonction

```
colonne :: Matrice a -> Int -> Vecteur a
```

qui extrait la colonne d'indice i d'une matrice. Par exemple

```
colonne [[1,2,3],[0,-6,7]] 1
```

donne comme résultat `[2,-6]`.

Q 58. Définissez une fonction

```
colonnes :: Matrice a -> [Vecteur a]
```

qui calcule la liste des colonnes d'une matrice. Par exemple

```
colonnes [[1,2,3],[0,-6,7]]
```

donne comme résultat `[[1,0],[2,-6],[3,7]]`.

Q 59. Définissez une fonction

```
prodScalaire :: Num a => Vecteur a -> Vecteur a -> a
```

qui, pour deux vecteurs de même taille, multiplie deux à deux les composantes puis additionne les résultats. Par exemple

```
prodScalaire [4,3,-1] [5,2,3]
```

donne comme résultat `23` (c'est-à-dire `4*5 + 3*2 + (-1)*3`).

Q 60. Définissez enfin une fonction qui multiplie deux matrices :

```
multMatrice :: Num a => Matrice a -> Matrice a -> Matrice a
```