Programmation fonctionnelle

Devoir surveillé intermédiaire

13 février 2015

Durée : 1h
Documents (notes de cours, TD et TP) autorisés

L'énoncé est long, la notation en tiendra compte.
Vous composerez votre DS sur deux copies séparées afin de paralléliser la correction.

1 Typage et définitions de fonctions

À rédiger sur la première copie

Considérons la liste de définitions suivante :

```
v1   = 'a' : ('b','c')
v2   = 1 : [2+3]
v3   = (1,2) : [(2,3), (3,4,5)]
v4   = [(True,'a'), ('b',False)]
v5   = (\x -> x+1) : []
v6 f = \a b -> f a
```

Q 1. Pour chaque v*n*, indiquez s'il est bien ou mal typé : s'il est bien typé, donnez son type ; s'il est mal typé, expliquez pourquoi.

Le type utilisé dans le cours pour décrire un analyseur syntaxique est

```
newtype Parser a = MkParser (String -> Maybe (a, String))
```

Q 2. Expliquez ce que signifie ce type en détaillant ce que sont `Parser`, `a`, `MkParser`, `String` et `Maybe`.
Donnez un exemple de valeur simple ayant ce type.
Pourquoi ce type est-il pertinent pour faire de l'analyse syntaxique ?

Nous avons vu que l'on peut écrire `[1..4]` pour obtenir la liste `[1,2,3,4]`. Mis à part le *sucre syntaxique* (c'est-à-dire la notation), ce résultat est produit par une fonction qui énumère toutes les valeurs de l'intervalle indiqué.

Q 3. Définissez une fonction `enumereDeA :: Integer -> Integer -> [Integer]` qui calcule la liste de tous les entiers compris entre son premier et son second arguments (inclus). Cette fonction sera récursive et ne pourra bien évidemment pas faire appel à la notation `[ .. ]` !
Par exemple :

```
ghci> enumereDeA 1 10
[1,2,3,4,5,6,7,8,9,10]
ghci> enumereDeA 11 10
[]
```

Q 4. Définissez une fonction similaire `enumereDeA'` telle que, si son premier argument `a` est supérieur à son second argument `b`, calcule la liste des entiers de l'intervalle de `b` à `a` en ordre décroissant. Sinon, elle se comporte comme la fonction précédente.

Par exemple :

```
ghci> enumereDeA' 10 0
[10,9,8,7,6,5,4,3,2,1,0]
ghci> enumereDeA' 10 20
[10,11,12,13,14,15,16,17,18,19,20]
```

Rappelons que `map :: (a -> b) -> [a] -> [b]` applique une fonction à tous les éléments d'une liste, c'est-à-dire telle que `map f [x1, x2, x3,...]` vale `[f x1, f x2, f x3, ...]`.

Nous voulons une fonction `pam :: a -> [a -> b] -> [b]` telle que `pam x [f1, f2, f3, ...]` vale `[f1 x, f2 x, f3 x, ...]`.

Q 5. Définissez la fonction `pam`. Vous donnerez une version récursive et une version utilisant la fonction `map`.

2 Addition booléenne

À composer sur la seconde copie

Nous allons considérer ici des nombres entiers positifs représentés comme des listes de `Bit` en commençant par le bit de poids faible, où chaque `Bit` sera représenté par un booléen.

Par exemple, le nombre 2 sera représenté par `[False,True]` et 13 par `[True,False,True,True]`.

Q 6. Complétez les déclarations de synonymes de types :

```
type Bit = ...
type Nombre = ...
```

Q 7. Définissez une fonction `deNombre :: Nombre -> Integer` qui calcule l'entier correspondant à son argument de type `Nombre`. Vous donnerez deux versions : une récursive et une utilisant un `foldr`.

L'objectif des questions qui suivent est de définir l'addition sur cette représentation des nombres. Par exemple :

```
ghci> additionne [False,True] [True,False,True,True]
[True,True,True,True]
ghci> additionne [True,True] [True,False,True,True]
[False,False,False,False,True]
```

correspondent aux additions $2 + 13 = 15$ et $3 + 13 = 16$.

Vous pourrez utiliser les opérateurs `&&`, `||` et `not` sur les booléens. Le ou-exclusif peut aussi être utile.

Q 8. Définissez une fonction `xor :: Bool -> Bool -> Bool` qui calcule le ou-exclusif entre ses deux arguments, c'est-à-dire vrai si et seulement si exactement un de ses arguments est vrai.

Nous allons « poser l'addition » comme à l'école. Nous aurons donc besoin d'additionner trois bits (deux chiffres et une retenue).

Q 9. Définissez une fonction `add3Bits :: Bit -> Bit -> Bit -> (Bit,Bit)` qui additionne trois bits. Le résultat prend deux bits, que vous représenterez par une paire. Le premier de la paire sera le bit de poids fort.

Le bit de poids fort d'une addition deviendra une retenue pour l'addition suivante.

Q 10. Définissez une fonction `add2Bits :: Bit -> Bit -> (Bit,Bit)` qui ajoute deux bits, par application partielle de `add3Bits`.

Q 11. Définissez une fonction `additionne :: Nombre -> Nombre -> Nombre` qui additionne deux nombres.