Programmation fonctionnelle

Devoir surveillé final

27 mars 2017

Durée : 2h30
Documents (notes de cours, TD, TP et mini-haddock) autorisés

La clarté et la simplicité de vos réponses seront prises en compte dans l'évaluation. La définition d'une fonction récursive lorsque vous auriez dû identifier ce cas comme une instance d'une fonction standard (`map`, `zipWith`, `foldr`, etc.) pourra être pénalisée.

Dès que votre code n'est pas trivial, commentez-le ! En particulier, si vous définissez une fonction auxiliaire, spécifiez-la (que calcule-t-elle, quels arguments prend-elle, etc.).

Vous composerez votre DS sur deux copies séparées afin de paralléliser la correction.

1 Première copie

À composer sur la première copie

Q 1. *(0,5 point)* Indiquez votre numéro de place sur votre première copie.

1.1 Parcours d'arbre

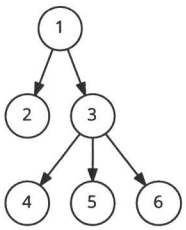


Fig. 1: Exemple d'arbre

L'objectif de cet exercice est de définir une structure de données arborescente et quelques fonctions de parcours de cette structure. Lorsque vous définissez une fonction, précisez toujours son type.

Q 2. Définissez une structure de données `Arbre t` pour représenter des arbres de sorte que :

- les nœuds portent une valeur de type `t`,
- les nœuds ont un nombre quelconque d'enfants (y compris 0 ou 1),
- il n'y a pas de constructeur de feuille (cela simplifiera les opérations suivantes).

Q 3. Définissez `exArb :: Arbre Int` qui correspond à l'arbre de la figure 1.

Q 4. Définissez une fonction qui calcule la hauteur d'un arbre. La hauteur de l'arbre constitué juste d'un nœud, sans enfants, sera 1.

Q 5. Définissez une fonction `treemap` qui prend en arguments une fonction et un arbre et retourne l'arbre dans lequel la fonction a été appliquée à toutes les valeurs de tous les nœuds.

Par exemple `treemap (1000+) exArb` donnera un arbre de même forme mais dont les nœuds portent les valeurs 1001 à 1006.

Q 6. Définissez une fonction `extrForet` qui prend en arguments un entier `n` et un arbre `a` et retourne la liste de tous les sous-arbres à profondeur `n` dans `a`.

Par exemple :

- `extrForet 0 exArb` retournera la liste ne contenant que `exArb`,
- `extrForet 1 exArb` retournera la liste contenant le sous-arbre ayant pour racine 2 et celui ayant pour racine 3.

Q 7. Définissez une fonction `extrValeurs` qui prend en arguments un entier `n` et un arbre `a` et retourne la liste de toutes les valeurs à profondeur `n` dans `a`.

Par exemple : `extrValeurs 1 exArb` retournera `[2,3]`.

1.2 Manipulation de listes

Dans cet exercice, il vous est demandé d'écrire des fonctions sur les listes, mais vous ne devez pas écrire de fonctions récursives. Il vous faut utiliser les fonctions qui sont indiquées.

La fonction « `all :: (a -> Bool) -> [a] -> Bool` » permet de vérifier que tous les éléments d'une liste vérifient une propriété. Par exemple : « `all (\e -> e `mod` 2 == 0) ns` » permet de vérifier que `ns` ne contient que des nombres pairs.

De même « `any :: (a -> Bool) -> [a] -> Bool` » permet de vérifier qu'il existe au moins un élément de la liste qui vérifie une propriété.

Q 8. Construire une fonction « `noMultiple5 :: [Int] -> Bool` » qui vérifie qu'une liste ne contient pas de multiple de 5.

Q 9. Construire une fonction « `oneGT50 :: [Int] -> Bool` » qui vérifie qu'une liste contient au moins un entier plus grand que 50.

La fonction « `filter :: (a -> Bool) -> [a] -> [a]` » renvoie la liste des éléments qui vérifient une propriété. Par exemple :

```
ghci> filter (\e -> e `mod` 2 /= 0) [1,2,3,4,5,6]
[1,3,5]
```

Q 10. En utilisant la fonction `filter` écrire une fonction « `count :: (Int -> Bool) -> [Int] -> Int` » telle que « `count p ns` » renvoie le nombre d'éléments de `ns` qui vérifient la propriété `p`.

2 Seconde copie

À composer sur la seconde copie

Q 11. *(0,5 point)* Indiquez votre numéro de place sur votre seconde copie.

2.1 Chemins

L'objectif de cet exercice est de proposer une API de cartographie pour trouver les chemins d'un endroit à un autre.

Nous manipulerons trois types :

- Un **point** est un couple de deux nombres flottants représentant ses coordonnées (x et y).
- Un **segment** est défini par deux points, ses deux extrémités (c'est-à-dire son départ et son arrivée).
- Un **chemin** est une liste de segments que l'on supposera contigus : pour deux segments successifs de la liste, vous pourrez supposer (sans le vérifier) que le point d'arrivée du premier est le point de départ du suivant.

Q 12. Donnez les synonymes de types `Point`, `Segment`, `Chemin` correspondant aux définitions ci-dessus.

Vous utiliserez dans la suite une fonction « `longueur` » qui calcule[1] la longueur d'un `Segment` AB sans avoir besoin de la définir.

Pour toutes les fonctions ci-dessous, vous donnerez explicitement leurs types.

Q 13. Implémentez une fonction `taille` donnant la taille d'un chemin c'est-à-dire la somme des longueurs de ses segments.

Q 14. Définissez une fonction `minChemin` qui prend en arguments deux chemins et retourne le plus court des deux.

Q 15. Programmez une fonction `cheminLePlusCourt` recherchant le chemin le plus court dans une liste de chemins. Ce plus court chemin pouvant ne pas exister, adaptez le type de votre fonction en conséquence.

On appelle **étape** un point intermédiaire d'un chemin. Par exemple, un chemin constitué de deux segments AB (départ : A ; arrivée : B) et BC a pour unique étape **B**.

Q 16. Définissez une fonction `etapes` fournissant la liste des étapes d'un `chemin`.

Q 17. Implémentez alors la fonction `cheminsAvecEtape` déterminant à partir d'une liste de chemins `cs` et d'une étape `e`, la liste des chemins de `cs` ayant `e` pour étape.

Q 18. Généralisez la fonction précédente en une fonction `cheminsAvecEtapes` qui prend une liste de chemins `cs` et une liste d'étapes `es` et retourne la liste des chemins de `cs` passant par toutes les étapes de `es`.

Q 19. Pour finir, programmez la fonction `cheminLePlusCourtEtapes` calculant à partir d'une liste de chemins `cs` et d'une liste d'étapes `es`, le plus court chemin de `cs` passant par toutes les étapes de `es`.

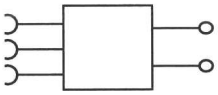
2.2 Assemblage de composants

Dans cet exercice, il s'agit tout d'abord de faire l'analyse syntaxique d'un langage de description d'assemblage de composants[2] décrit ci-dessous. Ce langage permet de décrire des *composants atomiques*, des *mises en série* de composants et des *mises en parallèle* de composants.

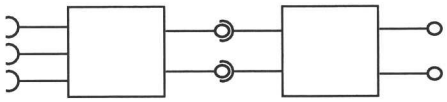
Un *composant* est caractérisé par un nombre de connexions d'entrée et un nombre de connexions de sortie. Textuellement, un composant atomique sera noté par un couple « < *in* ; *out* > » où *in* est le nombre d'entrées et *out* est le nombre de sorties du composant.

1. La longueur se calcule simplement avec la formule habituelle $\sqrt{(x_A - x_B)^2 + (y_A - y_B)^2}$.
2. Cela peut être par exemple des composants logiciels, mais peu importe.

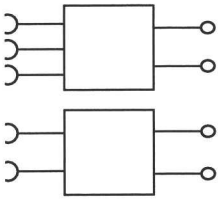
Par exemple, « <3 ; 2> » correspond au composant :



Nous pouvons *mettre en série* deux composants en connectant toutes les sorties du premier aux entrées du second. Cette opération n'est donc correcte que si le premier a autant de sorties que le second a d'entrées. Nous utiliserons le symbole « & » pour noter cette opération. Par exemple « <3 ; 2> & <2 ; 2> » correspond à :



Nous pouvons également *mettre en parallèle* deux composants. Nous utiliserons le symbole « | » pour noter cette opération. Par exemple « <3 ; 2> | <2 ; 2> » peut se représenter :



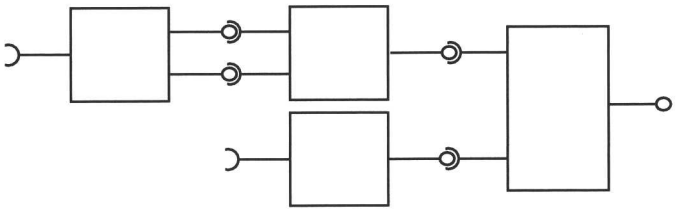
Le résultat de la mise en parallèle est donc un nouveau composant ayant toutes les entrées et toutes les sorties des deux composants.

Parseur

Nous allons tout d'abord définir un parseur pour la description que nous avons donnée pour les composants. Ajoutons que, pour simplifier le parseur :

- nous utiliserons systématiquement des parenthèses pour grouper les sous-expressions, c'est-à-dire que nous autoriserons uniquement « (<1 ; 2> & <2 ; 3>) & <3 ; 4> » et « <1 ; 2> & (<2 ; 3> & <3 ; 4>) » (mais pas « <1 ; 2> & <2 ; 3> & <3 ; 4> »),
- nous considérerons qu'il n'y a pas d'espaces dans la chaîne à parser, même si nous en utilisons dans les exemples pour les rendre plus lisibles,
- le parseur réussira même si la description ne correspond pas à un composant correct ; nous accepterons par exemple « <1 ; 2> & <4 ; 1> » ; la validation sera faite dans un deuxième temps.

Q 20. Donner une description textuelle pour l'assemblage de composants suivant :



On définit tout d'abord le synonyme de type :

```
type InOut = (Int , Int)
```

qui permet de représenter un couple (nombre d'entrées, nombre de sorties) et on se donne le parseur :

```
chiffre :: Parser Char
chiffre = carQuand (`elem` "012345679")
```

utilisant le module `Parser` vu pendant le semestre et documenté dans votre mini-Haddock.

L'objectif des questions suivantes est de construire un parseur `composant :: Parser Composant` tel que :

```
ghci> runParser composant "((<1;2>&<2;1>)&<1;4>)|<3;1>"
Just (Par (Ser (Ser (Atm (1,2)) (Atm (2,1))) (Atm (1,4))) (Atm (3,1)),"")
```

où le type `Composant` est défini par :

```
data Composant = Atm InOut                  -- ^ Composant atomique
               | Ser Composant Composant    -- ^ Mise en série
               | Par Composant Composant    -- ^ Mise en parallèle
```

Q 21. Donner un parseur `nombre :: Parser Int` tel que :

```
ghci> runParser nombre "123ABC"
Just (123,"ABC")
ghci> runParser nombre ""
Nothing
ghci> runParser nombre "ABC"
Nothing
```

Q 22. Donner un parseur `atomique :: Parser Composant` tel que :

```
ghci> runParser atomique "<2;3>123"
Just (Atm (2,3),"123")
ghci> runParser composant "<2;3>|<3;1>"
Just (Atm (2,3),"|<3;1>")
ghci> runParser atomique "(<1;2>&<2;1>)&<1;4>"
Nothing
```

Q 23. Donner un parseur `serie :: Parser Composant` tel que :

```
ghci> runParser serie "<1;2>"
Nothing
ghci> runParser serie "<1;2>&<4;5>123"
Just (Ser (Atm (1,2)) (Atm (4,5)),"123")
ghci> runParser serie "(<1;2>&<2;1>)&<1;4>|<3;1>"
Just (Ser (Ser (Atm (1,2)) (Atm (2,1))) (Atm (1,4)), "|<3;1>")
```

Il utilisera le parseur global `composant`.

Définir le parseur `parallele :: Parser Composant` serait un copié-collé quasi-complet.

Q 24. Donner un combinateur de parseur `operateur`, dont le type sera précisé, permettant de définir `serie` et `parallele` sans recopie de code.

Donner les définitions de `serie` et `parallele` utilisant `operateur`.

Q 25. Donner le parseur global `composant :: Parser Composant`, en définissant au passage un parseur auxiliaire pour les composants parenthésés.

Validation

On souhaite maintenant vérifier les contraintes liées aux mises en série de composants, c'est-à-dire que le nombre de sorties du premier composant est égal au nombre d'entrées du second pour toutes les mises en série dans un assemblage de composants.

Q 26. Écrire une fonction `validation :: Composant -> Maybe InOut` qui retourne `Nothing` si les contraintes de mises en séries sont mises en défaut par le composant passé en paramètre sinon retourne le nombre d'entrées et sorties du composant :

```
ghci> validation (Ser (Atm (1,2)) (Atm (2,3)))
Just (1,3)
ghci> validation (Ser (Atm (1,2)) (Atm (4,3)))
Nothing
```