Programmation fonctionnelle

Devoir surveillé final

31 mars 2016

Durée : 2h
Documents (notes de cours, TD et TP) autorisés

Vous composerez votre DS sur deux copies séparées, une pour la section 1, une pour les sections 2 et 3. Les trois parties peuvent être traitées indépendamment ; la section 3.2 peut être traitée indépendamment de 3.1.

Vous trouverez en annexe une brève documentation de fonctions des modules `Data.List` et `Parser` que vous pourrez utiliser dans vos réponses.

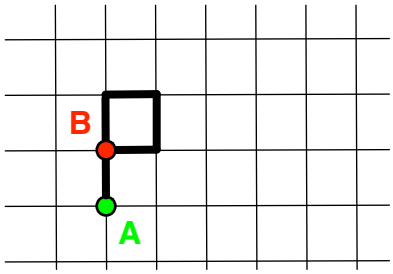
Attention : La clarté et la simplicité de vos réponses seront prises en compte dans l'évaluation. En particulier la définition d'une fonction récursive lorsque vous auriez dû identifier ce cas comme une instance d'une fonction standard (`map`, `zipWith`, `foldr`, etc.) sera pénalisée.

Par ailleurs, 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.). Le manque de clarté pourra être pénalisé.

1 Chemins sur une grille carrée

À composer sur la première copie

Une représentation classique d'un tracé ou d'un chemin dans un repère orthonormé est de donner une liste de directions correspondant à autant de pas élémentaires à faire dans les directions indiquées. Par exemple `[Nord, Nord, Est, Sud, Ouest]` veut dire : avancer deux fois au nord (c'est-à-dire en haut), une fois à l'est, une fois au sud et une fois à l'ouest comme on le voit sur la figure suivante, où le chemin a comme point origine le point `A` et comme point d'arrivée le point `B` :



Dans cet exercice, on se propose d'enrichir un module de gestion de tels chemins de quelques fonctions, notamment pour les *simplifier*. Étant donné deux chemins allant d'un point `A` à un point `B`, nous dirons que le plus court est le plus *simple*.

Voici le début de ce module :

```
data Direction = Nord | Sud | Est | Ouest
    deriving (Show, Ord, Eq)

type Chemin = [Direction]

(>-<) :: Direction -> Direction -> Bool
Nord  >-< Sud   = True
Sud   >-< Nord  = True
Est   >-< Ouest = True
Ouest >-< Est   = True
_     >-< _     = False
```

L'ordre sur le type `Direction` par le `deriving Ord` est tel que `Nord < Sud < Est < Ouest`. Vous pourrez utiliser cet ordre dans vos réponses.

L'opérateur `(>-<)` permet de savoir si deux directions données sont opposées.

Q 1. Définissez une fonction `simplifie_un_peu :: Chemin -> Chemin` qui supprime d'un chemin les allers-retours immédiats Nord-Sud, Sud-Nord, Est-Ouest, Ouest-Est rencontrés dans l'ordre de parcours du chemin. Par exemple :

```
ghci> simplifie_un_peu [Nord, Sud]
[]
ghci> simplifie_un_peu [Nord, Est, Sud, Ouest]
[Nord, Est, Sud, Ouest]
ghci> simplifie_un_peu [Ouest, Nord, Sud, Est]
[Ouest, Est]
```

Q 2. Définissez une fonction `simplifie_plus :: Chemin -> Chemin` qui continue à simplifier un peu un chemin tant que c'est possible. Par exemple :

```
ghci> simplifie_plus [Ouest,Nord,Sud,Est,Ouest,Est]
[]
ghci> simplifie_plus [Ouest,Nord,Est,Ouest,Sud,Est,Ouest,Sud,Nord,Sud]
[Ouest,Sud]
ghci> simplifie_plus [Nord,Ouest,Nord,Est,Ouest,Sud,Est,Ouest,Sud,Nord,Sud]
[Nord,Ouest,Sud]
```

On constate qu'on peut directement *simplifier plus* un chemin à la construction en vérifiant l'apparition d'un aller-retour immédiat lors de l'ajout d'une nouvelle étape dans le chemin. Par exemple, si on ajoute successivement en tête d'un chemin initialement vide les étapes `Sud`, `Nord`, `Sud`, `Ouest`, `Est`, `Sud`, `Ouest`, `Est`, `Nord`, `Ouest` et `Nord` en faisant cette vérification *à la volée*, on obtiendra successivement `[Sud]`, `[]`, `[Sud]`, `[Ouest, Sud]`, `[Sud]`, `[Sud, Sud]`, `[Ouest, Sud, Sud]`, `[Sud, Sud]`, `[Sud]`, `[Ouest, Sud]` et `[Nord, Ouest, Sud]`.

Q 3. Définissez une fonction `ajoute_en_simplifiant :: Direction -> Chemin -> Chemin` qui réalise un ajout en simplifiant *à la volée* comme expliqué ci-dessus.

Q 4. Utilisez `ajoute_en_simplifiant` pour donner une nouvelle version de `simplifie_plus` que vous nommerez `simplifie_plus'`.

Q 5. Donnez une propriété permettant de tester à l'aide de `QuickCheck`[1] l'équivalence entre `simplifie_plus` et `simplifie_plus'`.

La suppression des allers-retours immédiats ne supprime pas toutes les boucles qui peuvent apparaître dans un chemin, boucles qu'on voudrait supprimer si on souhaite vraiment simplifier le plus possible un chemin de son origine à sa destination. Par exemple, le chemin `[Nord, Nord, Est, Sud, Ouest]` devrait être simplifié en `[Nord]`, ce que ne fait pas la fonction `simplifie_plus'`.

Q 6. Définissez une fonction `simplifie :: Chemin -> Chemin` qui simplifie complètement un chemin.

On considère maintenant des représentations compressées de chemins à l'aide du type suivant :

```
type CheminCompresse = [(Direction, Int)]
```

Cela permet de compresser dans un chemin des occurrences successives d'une même direction. Par exemple, le chemin `[Nord, Sud, Sud, Sud, Ouest, Ouest, Sud]` peut être représenté par le chemin compressé `[(Nord, 1), (Sud, 3), (Ouest, 2), (Sud, 1)]`.

Q 7. Définissez une fonction `compresse :: Chemin -> CheminCompresse` qui retourne la version compressée d'un chemin. Par exemple :

```
ghci> compresse [Nord, Ouest, Nord, Nord]
[(Nord, 1), (Ouest, 1), (Nord, 2)]
```

Q 8. Définissez une fonction `decompresse :: CheminCompresse -> Chemin` qui effectue la conversion inverse de la précédente. Vous supposerez, sans le vérifier, que les valeurs entières apparaissant dans le chemin compressé en entrée sont toutes positives ou nulles. Par exemple :

```
ghci> decompresse [(Nord, 1), (Sud, 3), (Ouest, 2), (Sud, 1)]
[Nord, Sud, Sud, Sud, Ouest, Ouest, Sud]
ghci> decompresse [(Nord, 0), (Sud, 2)]
[Sud, Sud]
```

Q 9. Donnez une propriété permettant de tester à l'aide de `QuickCheck` les deux fonctions précédentes.

On définit la *version normalisée* d'un chemin par la liste composée des 4 entiers correspondant respectivement au nombre d'occurrences de chacune des directions `Nord`, `Sud`, `Est` et `Ouest` dans le chemin. Par exemple, la version normalisée du chemin `[Nord, Sud, Sud, Sud, Ouest, Ouest, Sud]` est `[1, 4, 0, 2]`.

Q 10. Définissez une fonction `normalise :: Chemin -> [Int]` qui donne la version normalisée d'un chemin. Par exemple :

```
ghci> normalise []
[0, 0, 0, 0]
ghci> normalise [Nord]
[1, 0, 0, 0]
ghci> normalise [Nord, Sud, Sud, Est, Sud]
[1, 3, 1, 0]
```

1. On supposera que `QuickCheck` sait générer aléatoirement des valeurs de type `Direction`.

2 λ-calcul et interprète

À composer sur la seconde copie

Considérons quelques expressions, écrites en utilisant la syntaxe du λ-calcul et leur équivalent en utilisant la syntaxe de Haskell et du mini-langage défini en TP.

Dans la définition de Z, il faut comprendre Y (et `Y`) comme l'expression définie à la ligne supérieure.

nom	λ-calcul	Haskell et mini-langage
X	$(\lambda x\, y.y)\, a\, b$	`(\x -> \y -> y) a b`
X′		`(\x -> \y -> y) 1 2`
Y	$\lambda f.(\lambda x.f(x\,x))(\lambda x.f(x\,x))$	`\f -> (\x -> f (x x)) (\x -> f (x x))`
Z	$(Y)(\lambda x\, y.y)$	`(Y) (\x -> \y -> y)`
W	$(\lambda x\, y.y)\,((\lambda x.x\,x)\,(\lambda y.y\, y))\, z$	`(\x -> \y -> y) ((\x -> x x) (\y -> y y)) 1`

Q 11. Pour chaque λ-terme, explicitez toutes les suites de réductions possibles en indiquant à chaque étape l'ensemble des rédex et la substitution à faire pour réduire l'expression.

Q 12. Pour chaque terme en syntaxe Haskell et mini-langage, indiquez :
- les étapes de calcul et le résultat obtenu par l'interprète du TP[2],
- le résultat donné par Haskell ; vous justifierez ce résultat.

3 JSON

JSON est un format d'échange de données structurées. Ce format a l'avantage d'être très simple, notamment parce qu'il représente les données dans un format proche de beaucoup de langages de programmation, notamment de JavaScript.

Dans la suite, nous ne considérerons que les constructions principales de JSON.

Chaîne de caractères Elles sont représentées entre guillemets :

```
"une chaîne de caractères"
```

Liste Séquence (ordonnée) de taille quelconque (zéro ou plus) de valeurs. Elles sont représentées entre crochets, avec la virgule comme séparateur :

```
["une", "liste", "de", "5", "chaînes"]
```

Objet Ensemble (non-ordonné) de taille quelconque de paires clef-valeur, où les clefs sont des chaînes. (Ils correspondent aux structures de données habituelles de *dictionnaire*, ou table de hachage). Ils sont représentés entre accolades, avec un deux-points comme séparateur entre une clef et sa valeur, et la virgule comme séparateur entre les paires de clef-valeur :

```
{
  "clef1": "une chaîne comme valeur",
  "clef2": ["une", "liste", "comme", "valeur" ],
  "clef3": {
    "descr": "un objet comme valeur"
  }
}
```

2. Vous considérerez ici l'interprète le plus simple du TP, l'`interpreteA`.

Définissons donc le type `JSON` :

```
data JSON = Objet  [(String, JSON)]
          | Liste  [JSON]
          | Chaine String
    deriving (Show)
```

3.1 Analyseur syntaxique pour JSON

L'objectif de cette partie est de définir un analyseur syntaxique pour JSON. Vous utiliserez pour cela le module `Parser` vu pendant le semestre. Vous trouverez un rappel de `Parser` en annexe.

Pour simplifier le parseur, nous supposerons que le document JSON à parser ne contient aucun espace en dehors des chaînes de caractères.

Commençons par les chaînes de caractères. Pour simplifier le problème, nous allons dans un premier temps supposer qu'elles ne peuvent pas contenir le caractère « " ». Elles sont donc représentées par un « " », suivi d'un nombre quelconque de caractères qui ne sont pas « " » puis d'un « " ».

Q 13. Définissez un parseur `toutSauf :: [Char] -> Parser Char` qui étant donné une liste de caractères accepte un caractère quelconque s'il n'est pas dans la liste.

```
ghci> runParser (toutSauf ['a','b']) "ac"
Nothing
ghci> runParser (toutSauf ['a','b']) "bc"
Nothing
ghci> runParser (toutSauf ['a','b']) "c"
Just ('c',"")
ghci> runParser (toutSauf ['a','b']) ""
Nothing
```

Q 14. Définissez un parseur `chaineP :: Parser String`. Par exemple :

```
ghci> runParser chaineP  ['"', 'a', 'b', '"']
Just ("ab","")
```

Q 15. Définissez un parseur `jsonChaine :: Parser JSON` qui reconnaît une chaîne de caractères et en fait une valeur de type `JSON`.

Nous allons maintenant définir les parseurs suivants :

- `jsonListe :: Parser JSON` pour les listes,
- `jsonObjet :: Parser JSON` pour les objets,
- `json :: Parser JSON` pour n'importe quelle valeur JSON.

Leurs définitions seront naturellement mutuellement récursives.

Q 16. Définissez un parseur `json :: Parser JSON` pour n'importe quelle valeur JSON.

Les parseurs `jsonListe` et `jsonObjet` ont un point commun : leur contenu est une séquence avec des virgules comme séparateurs. Afin d'éviter de dupliquer du code, nous allons définir deux combinateurs pour des séquences de valeurs séparées par des virgules.

Q 17. Définissez deux combinateurs de parseurs

```
zeroOuPlusVirg :: Parser a -> Parser [a]
unOuPlusVirg   :: Parser a -> Parser [a]
```

tels que :

```
ghci> runParser (zeroOuPlusVirg (car 'a')) ""
Just ("","")
ghci> runParser (zeroOuPlusVirg (car 'a')) "a"
Just ("a","")
ghci> runParser (zeroOuPlusVirg (car 'a')) "a,a"
Just ("aa","")
ghci> runParser (unOuPlusVirg (car 'a')) ""
Nothing
ghci> runParser (unOuPlusVirg (car 'a')) "a"
Just ("a","")
ghci> runParser (unOuPlusVirg (car 'a')) "a,a"
Just ("aa","")
```

Les listes sont représentées par un « [», suivi d'un nombre quelconque (y compris zéro) de valeurs JSON séparées par des virgules et terminées par un «] ».

Q 18. Définissez un parseur `jsonListe :: Parser JSON`. Par exemple :

```
ghci> runParser jsonListe "[]"
Just (Liste [],"")
ghci> runParser jsonListe "[[]]"
Just (Liste [Liste []],"")
ghci> runParser jsonListe "[{},{},[]]"
Just (Liste [Objet [],Objet [],Liste []],"")
```

Les objets JSON sont représentés par un « { », suivi d'un nombre quelconque (y compris zéro) de paires clef-valeur séparées par des virgules et terminées par un « } ».

Q 19. Définissez un parseur `clefValeur :: Parser (String, JSON)` qui parse :

— une clef, qui est une chaîne de caractère,
— le caractère « : »,
— une valeur JSON.

Par exemple :

```
ghci> runParser clefValeur "\"clef\":[]"
Just (("clef",Liste []),"")
ghci> runParser clefValeur "\"clef\":\"valeur\""
Just (("clef",Chaine "valeur"),"")
```

Q 20. Définissez un parseur `jsonObjet :: Parser JSON`.

3.2 Représentation JSON d'un type d'arbre

Q 21. Définissez un type d'arbre tel que les nœuds :

— contiennent des chaînes de caractères,
— ont un nombre quelconque de fils.

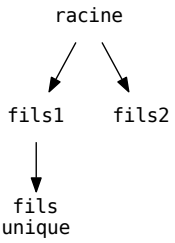
Toutes les solutions possibles qui respectent ces deux contraintes seront acceptées.

Nous décidons de sauvegarder ces arbres en JSON, en représentant un nœud par un objet JSON associant :

— à la clef « etiq » l'étiquette du nœud, c'est-à-dire la chaîne qu'il contient,
— à la clef « fils » la liste de ces fils.

Par exemple, le document JSON listé à gauche codera l'arbre de droite :

```json
{
  "etiq": "racine",
  "fils": [
    {
      "etiq": "fils1",
      "fils": [
        {
          "etiq": "fils unique",
          "fils": []
        }
      ]
    },
    {
      "etiq": "fils2",
      "fils": []
    }
  ]
}
```



Q 22. Définissez une fonction `decodeArbre` prenant en argument une valeur de type `JSON` et retournant un arbre.

3.3 Chaînes réalistes

Le format des chaînes de caractères que nous avons parsé est simpliste : nous voulons pouvoir mettre des « " » dans les chaînes. Suivant le format utilisé dans beaucoup de langages (Haskell, Java, C, JavaScript, etc.), JSON utilise « \ » comme caractère d'échappement, c'est-à-dire que les deux caractères « \" » dans la représentation d'une chaîne signifient que la chaîne contient un « " ». Il faut du coup aussi représenter « \ » par « \\ ». Cette mécanique est aussi utile pour les retours à la ligne, codés « \n », etc.

Q 23. Définissez `echappement :: (Char,Char) -> Parser Char` tel que `echappement (c,r)` réussit si et seulement s'il trouve les caractères « \ » suivi de `c` et retourne alors `r`.

Nous supposerons définie une `listeEchappements :: [(Char,Char)]` définissant tous les échappements possibles (elle contiendra par exemple `('"', '"')`, `('\\', '\\')` et `('n', '\n')`, etc.[3]).

Q 24. Définissez un parseur `echappements :: Parser Char`, et qui réussit si un des échappements représentés par la liste réussit.

Q 25. Définissez une version plus évoluée du parseur `chaineP` qui supporte les échappements.

3. Cf. http://xkcd.com/1638/

4 Annexe : extraits de la documentation

4.1 Data.List

Transformations

`map :: (a -> b) -> [a] -> [b]` Applique une fonction à tous les éléments d'une liste :

```
map f [x1, x2, ..., xn] == [f x1, f x2, ..., f xn]
map f [x1, x2, ...] == [f x1, f x2, ...]
```

Pliages

`foldl :: (b -> a -> b) -> b -> [a] -> b` Replie une liste de gauche à droite en appliquant un opérateur binaire et en utilisant une valeur initiale :

```
foldl f z [x1, x2, ..., xn] == (...((z `f` x1) `f` x2) `f`...) `f` xn
```

La liste doit être finie.

`foldl1 :: (a -> a -> a) -> [a] -> a` Variante de `foldl`, utilisant le premier élément de la liste plutôt qu'une valeur initiale.

```
foldl1 f [x1, x2, ..., xn] == (...(x1 `f` x2) `f`...) `f` xn
```

La liste ne doit pas être vide, et doit être finie.

`foldr :: (a -> b -> b) -> b -> [a] -> b` Replie une liste de droite à gauche en appliquant un opérateur binaire et en utilisant une valeur initiale :

```
foldr f z [x1, x2, ..., xn] == x1 `f` (x2 `f` ... (xn `f` z)...)
```

`foldr1 :: (a -> a -> a) -> [a] -> a` Variante de `foldr`, sans valeur initiale.

```
foldr f z [x1, x2, ..., xn_1, xn] == x1 `f` (x2 `f` ... (xn_1 `f` xn)...)
```

La liste ne doit pas être vide.

Pliages particuliers

`concat :: [[a]] -> [a]` Concatène une liste de listes.

`concatMap :: (a -> [b]) -> [a] -> [b]` Applique une fonction sur une liste et concatène les résultats.

`iterate :: (a -> a) -> a -> [a]` Retourne la liste infinie des applications itérées d'une fonction à une valeur :

```
iterate f x == [x, f x, f (f x), ...]
```

`replicate :: Int -> a -> [a]` Prend en argument une longueur `n` et une valeur `x` et retourne une liste contenant `n` fois `x`.

```
replicate 3 x == [x, x, x]
```

Recherche dans des listes

`elem :: Eq a => a -> [a] -> Bool` Teste l'appartenance d'un élément à une liste ; s'utilise en général sous forme infixe : `x `elem` xs`.

La liste doit être finie pour que le résultat soit `False`, elle peut être infinie si le résultat est `True`.

`notElem :: Eq a => a -> [a] -> Bool` Négation de `elem`.

`filter :: (a -> Bool) -> [a] -> [a]` Extrait les éléments d'une liste vérifiant un prédicat.

`zip :: [a] -> [b] -> [(a, b)]` Retourne la liste des paires correspondantes.

```
zip [x1, x2, ..., xn] [y1, y2, ..., yn] == [(x1,y1), (x2, y2), ... (xn, yn)]
```

Le résultat a la longueur de la liste la plus courte.

`zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]` Généralise `zip` en appliquant une fonction quelconque pour combiner les éléments des deux listes.

```
zipWith f [x1, x2, ..., xn] [y1, y2, ..., yn] == [f x1 y1, f x2 y2, ... f xn yn]
```

`sort :: Ord a => [a] -> [a]` Étant donné un type `a` pour lequel l'ordre (c'est-à-dire `(<=)`) est défini, trie une liste de `a` par ordre croissant.

4.2 Parser

`Parser a` Type d'un parseur qui retourne une valeur de type `a` s'il réussit.

`Parser` est une instance de `Monad`, de sorte qu'il est possible d'utiliser la notation `do` en plus de l'utilisation directe de `return` et `(>>=)`.

`runParser :: Parser a -> String -> Maybe (a, String)` Déclenche un parseur. Retourne `Nothing` en cas d'échec, `Just (v, reste)` en cas de réussite, où `v` est la valeur lue et `reste` est le reste de la chaîne à parser.

`reussit :: a -> Parser a` **et son synonyme** `return` Retourne un parseur qui réussit toujours avec le résultat donné.

`echoue :: Parser a` Parseur qui échoue toujours.

`unCaractereQuelconque :: Parser Char` Parseur qui retourne le premier caractère, et échoue s'il n'y a plus rien à parser.

`(|||) :: Parser a -> Parser a -> Parser a` Alternative : `p1 ||| p2` est le parseur qui retourne le résultat de `p1` quand `p1` réussit, sinon retourne le résultat de `p2`.

`(>>=) :: Parser a -> (a -> Parser b) -> Parser b` Séquence : `p >>= fp` retourne le résultat du parseur `fp v` où `v` est la valeur parsée par `p` s'il a réussit, et échoue sinon.

`carCond :: (Char -> Bool) -> Parser Char` Retourne le parseur qui réussit s'il trouve un caractère qui vérifie le prédicat donné.

`car :: Char -> Parser Char` Retourne le parseur qui réussit s'il trouve le caractère donné.

`chaine :: String -> Parser String` Retourne le parseur qui réussit s'il trouve la chaîne donnée.

`zeroOuPlus :: Parser a -> Parser [a]` Étant donné un parseur `p`, retourne le parseur qui itère `p` tant qu'il réussit et retourne la liste des résultats.

`unOuPlus :: Parser a -> Parser [a]` Étant donné un parseur `p`, retourne le parseur qui itère `p` tant qu'il réussit, échoue si `p` échoue initialement, sinon retourne la liste des résultats.