TP Haskell #2/7 (CS 222)

Grenoble INP – Esisar – année 2022-23

Durée : 1h30
Consignes et date de rendu : Voir Chamilo

Objectifs :

1. Pratique de la définition de types algébriques (sommes et produits).
2. Fonctions récursives et filtrage de motifs classiques.

Mise en place du TP

Réalisez le TP dans un répertoire de travail `CS222/TP2_VotreNom`. Chaque exercice `XXX.hs` sera accompagné d'un script GHCi, nommé `test_XXX.ghci`, contenant les expressions sur lesquelles vous avez testé votre code, et exécutable de la façon suivante dans GHCi :

```
:load XXX.hs
[... affichage terminant par Ok, one module loaded.]
:script test_XXX.ghci
[... affichage de l'évaluation de vos tests]
```

Consignes communes pour l'écriture des fonctions Haskell :

- Chaque définition de fonction sera précédée de sa déclaration de type.
- Privilégier la réutilisation des fonctions écrites précédemment.
- Privilégier le filtrage de motif (*pattern-matching*) à du code conditionnel.
- N'hésitez pas à utiliser les fonctions de la bibliothèque standard de Haskell.

Exercice 1 — Types somme et filtrage de motifs

Fichiers à rendre : `automate.hs`, `test_automate.ghci`.

On considère une machine à café qui peut être dans trois états : Vide, Percolation (préparation du café), et Café Prêt. La machine peut préparer l'équivalent de 4 tasses à chaque percolation.

- Dans un fichier `automate.hs`, définissez un type somme `EtatMachine` qui encode les trois états possibles de la machine avec trois contructeurs : `Vide` (sans paramètre), `Percolation` (sans paramètre), et `CafePret` (avec un paramètre entier indiquant le nombre de tasses restantes).

On rappelle la syntaxe de déclaration d'un type somme :

```
data <nom> = <option1> | <option2> ...
```

À la fin de la définition, ajoutez `deriving Show` pour permettre à GHCi d'afficher les valeurs de type `EtatMachine` dans le terminal.

On voudra aussi garder le compte, au fur et à mesure de la journée, du nombre de tasses de café consommées par les étudiant·e·s. On se donne donc le type

```
type InfoMachine = (Int, EtatMachine)
```

qui fait de `InfoMachine` un alias du type `(Int, EtatMachine)`, dont on se servira fréquemment. Dans cette paire, l'entier représente le nombre de cafés servis et le second membre indique l'état actuel de la machine.

- Écrivez trois fonctions `lancerMachine`, `attendre` et `servirCafe`, toutes trois du même type `InfoMachine -> InfoMachine`, qui modifient l'état de la machine pour réagir aux actions suivantes :
  - `lancerMachine` démarrera une percolation, sauf si la machine est déjà pleine.
  - `attendre` patientera jusqu'à la fin de la percolation en cours (après une attente la machine contiendra donc 4 doses de café).
  - `servirCafe` servira une tasse de café, s'il y a du café disponible.

  Dans les autres cas, ces fonctions renverront le paramètre `InfoMachine` inchangé.
- On encode une séquence de ces actions à l'aide d'une chaîne de caractères composée des lettres `'L'`, `'A'` et `'S'`. Écrire une fonction `executerActions :: InfoMachine -> [Char] -> InfoMachine` qui exécute toutes les actions d'une chaîne sur l'état initial fourni, et renvoie l'état final. (On pourra, optionnellement, s'aider d'une fonction auxiliaire écrite avec `where`).
- Réalisez si ce n'est pas déjà fait un script de tests nommé `test_automate.ghci` contenants vos tests.
- Sachant que la machine est vide au début de la journée, combien de cafés auront été servis et combien restera-t-il de tasses prêtes après la séquence `"LASSASSLSSASSSLSASSAASSLSAS"` ?

Exercice 2 — Fonctions classiques sur les listes

Fichiers à rendre : `listes.hs`, `test_listes.ghci`.

Dans cet exercice, on redéfinit quelques fonctions classiques sur les listes pour se familiariser avec. Dans les exercices et TP suivants, n'hésitez pas à recourir à ces fonctions de la bibliothèque standard (`map`, `filter`, etc) dès qu'elles vous semblent utiles.

- Écrire une fonction `my_map :: (a -> b) -> [a] -> [b]` qui applique une fonction de type `a -> b` à tous les éléments d'une liste. Écrivez quelques tests dans votre script `test_listes.ghci` et vérifiez que le résultat correspond à la fonction standard `map`.
- Écrire une fonction `my_filter :: (a -> Bool) -> [a] -> [a]` qui extrait de la liste donnée en paramètres tous les éléments pour lequel la fonction `a -> Bool` donnée renvoie `True`. De même, écrire des tests et comparer à la fonction `filter`.
- Écrire une fonction `my_unzip :: [(a,b)] -> ([a], [b])` qui sépare une liste de paires en deux listes, une contenant les éléments gauches des paires, l'autre contenant les éléments droits. Tester et comparer avec `unzip`. Tester également la fonction `zip` qui lui est associée.

Exercice 3 — Chaînes de caractères

Fichiers à rendre : `chaines.hs`, `test_chaines.ghci`.

Les chaînes de caractère ont pour type `String`, qui est (par définition) une liste de caractères, `[Char]`.

- Écrire une fonction `repeter_n_fois :: Int -> Char -> String` qui prend en arguments un entier `n` et un caractère `c` et qui renvoie la chaîne obtenue en répétant le caractère `c` `n` fois de suite.
- Adaptez la fonction pour répéter un `Float` au lieu d'un caractère. La définition de la fonction était-elle finalement spécifique au type `Char` ?
- Définir la fonction `etoiles :: Int -> String -> String` qui prend en arguments un entier `n` et une chaîne, et entoure la chaîne de `n` étoiles de chaque côté. On utilisera `repeter_n_fois`, ou pas (en critiquant).
- Définir la fonction `slashes` qui entoure une chaîne d'une barre oblique (`/`) de chaque côté.
- En utilisant la composition de fonctions avec l'opérateur `.` (point), en déduire une fonction `commentaire_documentation :: String -> String` qui prend une chaîne et qui en fait un commentaire de documentation C (commentaire avec deux étoiles : `/** ... **/`).

Fournissez vos tests de ces fonctions dans un fichier `test_chaines.ghci`.

Exercice 4 — Jeu de tarot

Cet exercice est facultatif ; vous pourrez rendre les fichiers `tarot.hs` et `test_tarot.ghci`.

Au tarot (le jeu de cartes), il y a 14 cartes de chaque couleur (par ordre de valeur : 1–10, Valet, Cavalier, Dame et Roi), plus 21 atouts (1–21) et une carte spéciale, l'excuse.

- Définir un type somme `Couleur` avec `data` pour représenter les 4 couleurs (Pique, Coeur, Carreau, Trèfle) et un autre type somme `Carte` avec trois constructeurs `Standard`, `Atout` et `Excuse` pour représenter toutes les cartes. On représentera les Valets, Cavaliers, Dames et Rois par les valeurs 11–14.

Ajoutez `deriving Show` à la fin de ces définitions si vous voulez pouvoir afficher les valeurs de type `Couleur` et `Carte` dans le terminal.

Les cartes ont un ordre de valeur : les cartes de chaque couleur sont comparées normalement et les atouts sont ordonnés par numéro (le 21 étant le plus fort). N'importe quel atout bat n'importe quelle carte standard, et l'excuse perd contre n'importe quelle autre carte.

- Écrire une fonction `comparer :: Carte -> Carte -> Bool` qui indique si la première carte est plus forte que la seconde. Si les cartes sont incomparables (par exemple deux `Standard` de différentes couleur), la fonction devra renvoyer `False`. Pensez bien à écrire des tests pour tous les cas pertinents de la fonction.

À chaque pli, un premier joueur pose une carte, et cette carte limite ce que les autres joueurs peuvent joueur.

Si une carte standard a été posée, le deuxième joueur est obligé de poser une carte standard de la même couleur (s'il en a), à défaut un atout (s'il en a), et à défaut n'importe quelle autre carte.

Si un atout a été posé, le deuxième joueur est obligé de poser un atout plus grand (s'il en a), à défaut un autout plus faible (s'il en a), et à défaut n'importe quelle autre carte.

Si l'excuse a été posée, n'importe quelle carte peut être jouée. De même, un joueur qui a l'excuse peut la jouer dans n'importe quelle situation, en plus des cartes citées précédemment.

- Écrire trois fonctions

```
meme_couleur :: Couleur -> Carte -> Bool
atout_plus_grand :: Int -> Carte -> Bool
atout :: Carte -> Bool
```

implémentant les trois critères : `meme_couleur` acceptera les cartes standards de la couleur donnée ; `atout_plus_grand` acceptera les atouts d'une valeur supérieure à la valeur donnée ; et `atout` acceptera les atouts. Ajouter les tests qui vont bien.

On s'apprête à écrire une fonction `cartes_possibles :: Carte -> [Carte] -> [Carte]` qui prend en paramètre la carte posée par le premier joueur, et indique à la deuxième joueuse quelles cartes elle peut jouer parmi sa main. On remarque un motif récurrent (hors excuse) : s'il y a des cartes d'une certaine catégorie, alors c'est celles-ci qu'elle peut jouer, sinon il faut essayer une catégorie suivante.

- Écrire une fonction `(||||) :: [a] -> [a] -> [a]` qui renvoie la première liste si elle est non vide, et la deuxième sinon.
- En utilisant `(||||)` et `filter`, écrire une fonction `cartes_possibles_hors_excuse` qui liste les cartes possibles en ignorant l'excuse. En déduire la fonction `cartes_possibles` qui ajoute l'excuse quand la joueuse la possède.

TP Haskell #3/7 (CS 222)

Grenoble INP – Esisar – année 2022-23

Durée : 1h30
Consignes et date de rendu : Voir Chamilo

Objectifs :

1. Pratique de la définition de fonctions.
2. Définitions de fonctions récursives.
3. Arbres et filtrage de motifs.

Mise en place du TP

Réalisez le TP dans un répertoire de travail `CS222/TP3_VotreNom`. Chaque exercice `XXX.hs` sera accompagné d'un script GHCi, nommé `test_XXX.ghci`, contenant les expressions sur lesquelles vous avez testé votre code, et exécutable de la façon suivante dans GHCi :

```
:load XXX.hs
[... affichage terminant par Ok, one module loaded.]
:script test_XXX.ghci
[... affichage de l'évaluation de vos tests]
```

Consignes communes pour l'écriture des fonctions Haskell :

- Chaque définition de fonction sera précédée de sa déclaration de type.
- Privilégier la réutilisation des fonctions écrites précédemment.
- Privilégier le filtrage de motif (*pattern-matching*) à du code conditionnel.
- N'hésitez pas à utiliser les fonctions de la bibliothèque standard de Haskell.

Exercice 1 — Accumulation sur les listes

Fichiers à rendre : `fold.hs`, `test_fold.ghci`.

- Écrire une fonction `somme_liste :: [Int] -> Int` qui calcule la somme des entiers d'une liste.

Ce motif consistant à *accumuler* les valeurs d'une liste est très récurrent ; on aimerait bien le factoriser. Commençons par généraliser l'opération.

- Écrire une fonction `acc_ints :: (Int -> Int -> Int) -> Int -> [Int] -> Int` qui prend en arguments une opération binaire, une valeur initiale, et une liste d'entiers. Elle devra accumuler les entiers de la liste, à partir de la valeur initiale, en appliquant l'opérateur binaire.
- En analysant votre code, déterminez si `acc_ints (-) 0 [x,y]` calcule `(0-x)-y` ou `x-(y-0)` (les deux sont possibles selon la façon dont vous avez écrit la fonction). Vérifiez avec un test.
- Écrire une fonction `max_liste :: [Int] -> Int` qui calcule le maximum d'une liste d'entiers positifs en utilisant une application partielle de `acc_ints`.

Finalement, le code de `acc_ints` ne semble plus être spécifique aux entiers, car l'addition est maintenant devenue une fonction générique.

- Recopier la définition de `acc_ints` sous le nom `acc` (penser à renommer l'appel récursif), cette fois sans annoncer son type. Quel type GHCi infère-t-il pour la fonction ?
- En déduire la fonction `my_foldl :: (b -> a -> b) -> b -> [a] -> b` qui accumule une liste de valeurs de type `a` et produit un résultat accumulé de type `b`. L'ordre d'association est imposé : `my_foldl (-) 0 [x,y]` devra calculer `(0-x)-y`.
- Écrire de même la fonction `my_foldr :: (a -> b -> b) -> b -> [a] -> b` qui associe dans l'ordre inverse : `my_foldr (-) 0 [x,y]` calculera donc `x-(y-0)`.
- Utiliser une de ces deux fonctions pour écrire la fonction `join :: [String] -> String` qui concatène les éléments de la liste en les séparant par des espaces. *On s'autorisera un espace supplémentaire au début ou à la fin.*
- Dans le cas où la valeur initiale est neutre pour la fonction d'accumulation, donner une condition suffisante sur la fonction pour que `foldl` et `foldr` soient équivalentes.

À partir de maintenant, n'hésitez pas à utiliser les fonctions standards `foldl` et `foldr` quand elles vous semblent appropriées (ainsi que les fonctions `sum`, `minimum`, `elem`, ... qui sont construites avec).

Exercice 2 — Arbres binaires

Fichiers à rendre : `arbres.hs`, `test_arbres.ghci`.

Dans cet exercice, nous allons manipuler des *arbres binaires*, une structure de données classique en algorithmique. Un arbre binaire est une structure de ce style :

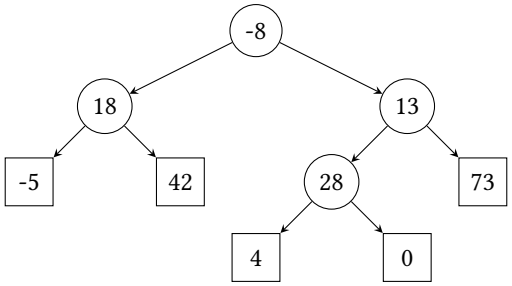

Figure 1: Un arbre binaire `arbre_exemple`

L'arbre est composé de *noeuds internes* (les ronds) et de *feuilles* (les carrés). Les noeuds internes ont chacun deux enfants qui sont eux-même des arbres (d'où le "binaire" — 2 enfants). Les feuilles n'ont pas d'enfant. Ici, on s'intéresse à un arbre dans lequel tous les noeuds portent une valeur (il y a de nombreuses variations).

On peut définir ce type ainsi en Haskell :

```haskell
data Bintree a =
  Leaf a |
  Node a (Bintree a) (Bintree a)
  deriving Show
```

- Expliquer la définition fournie, en commentaire du code. Que représente `a` ? À quel type de noeud de l'arbre correspond chaque constructeur ? Quels paramètres du constructeur `Node` correspondent aux deux enfants des noeuds internes ?

- Définir une constante `arbre_exemple :: Bintree Int` qui représente l'arbre du schéma ci-dessus.

La nature "imbriquée" des arbres binaires fait que la structure se prête très bien à l'écriture de fonctions récursives.

- Écrire une fonction `nombre_feuilles :: Bintree a -> Int` qui compte le nombre de feuilles dans l'arbre.
- Écrire une fonction `hauteur :: Bintree a -> Int` calculant la *hauteur* de l'arbre, ie. la plus grande distance entre le sommet de l'arbre et une des feuilles. (Par exemple, la hauteur de `arbre_exemple` est 4.)

On se demande maintenant si on peut avoir un arbre d'entiers (un `Bintree Integer`) qui est "trié" dans le sens où il vérifierait les deux propriétés suivantes :

1. Les valeurs associées aux noeuds sont uniques (ie. chaque valeur n'apparaît qu'une fois dans l'arbre).
2. Dans un noeud interne `Node v t1 t2`, toutes les valeurs présentes dans `t1` (le sous-arbre de gauche) sont inférieures à v, et toutes les valeurs présentes dans `t2` (le sous-arbre de droite) sont supérieures à v.

Un tel arbre s'appelle un *Arbre Binaire de Recherche (ABR)*.

- Construire un ABR `abr_exemple :: Bintree Integer` qui contient les mêmes valeurs que `arbre_exemple`.
- Écrire une fonction `rechercher_abr :: Bintree Integer -> Integer -> Bool` qui exploite la propriété d'ABR pour déterminer rapidement si un entier donné est présent dans l'arbre.
- Écrire une fonction `aplatir_abr :: Bintree a -> [a]` qui génère la liste triée des éléments de l'arbre.
- Comparer intuitivement la vitesse de `rechercher_abr` avec une recherche dans une liste et une recherche dichotomique dans un tableau. Y a-t-il différentes façons d'écrire `abr_exemple` qui impacteraient l'efficacité de la fonction `rechercher_abr` ?

Exercice 3 — Découpage en mots

Cet exercice est facultatif ; vous pourrez rendre les fichiers `mots.hs` et `test_mots.ghci`.

Le but de cet exercice est de lister les mots d'une chaîne de caractères comprenant des espaces, comme la méthode `split` de Python :

```
mots "  You shall   not pass!" -- ["You", "shall", "not", "pass!"]
```

Dans cet exercice, vous n'avez pas le droit aux fonctions standard sur les listes. ;D

- Écrire une fonction `espace :: Char -> Bool` qui détermine si le caractère passé en paramètre est un espace.
- Écrire une fonction `separe :: (Char -> Bool) -> String -> (String, String)` qui prend une chaîne et un prédicat sur les caractères, et sépare la chaîne en deux morceaux : le plus long préfixe de la chaîne dont tous les caractères vérifient le prédicat, et le reste.

  Par exemple `separe (== 'a') "aaabaaba"` renverra `("aaa", "baaba")`.

Privilégiez l'utilisation d'un `let` ou un `where` plutôt que `fst` et `snd` pour découper la paire obtenue lors de l'appel récursif.

- Utiliser `separe` et `espace` pour écrire la fonction `grignote_espaces :: String -> String` qui grignote les espaces en tête d'une chaîne.

On a maintenant tout ce qu'il faut pour extraire rapidement un mot, puis tous les mots.

- Écrire une fonction `un_mot :: String -> (String, String)` qui prend une chaîne ne commençant pas par un espace, et renvoie un couple contenant son premier mot (ie. la série de non-espaces au début) ainsi que le reste (dont les espaces initiaux seront retirés avec `grignote_espace`). Utilisez le plus possible les fonctions précédentes.
- Écrire une fonction `mots :: String -> [String]` qui sépare la chaîne donnée en mots. Il vous est suggéré de définir et appeler une fonction auxiliaire qui travaille sur des chaînes ne commençant pas par des espaces.

Et si ça vous démange depuis le début :

- Bonus : écrire une version plus simple de `mots` en utilisant les fonctions standard `dropWhile` et `span` (possible en 3 lignes).

Exercice 4 — Système de fichiers

Cet exercice est facultatif ; vous pourrez rendre les fichiers `fichiers.hs` et `test_fichiers.ghci`.

On revisite maintenant les arbres pour modéliser un système de fichiers. Dans un système de fichiers, les noeuds internes sont les dossiers tandis que les feuilles sont les fichiers. Cette fois, les arbres ne sont plus nécessairement *binaires* : un dossier peut contenir plus que 2 éléments (et même aucun).

- En s'inspirant de la définition de `Bintree a`, définissez le type `Tree a` qui a les mêmes constructeurs mais dans lequel `Node` prend en paramètres une valeur de type `a` et une liste de sous-arbres.

En observant le dossier de travail du TP, on remarque la hiérarchie suivante :

```
CS222/
├── TP1/
│   ├── CS222_TP1.pdf
│   ├── applis.hs
│   ├── rec.hs
│   └── test_rec.ghci
├── TP2/
│   ├── CS222_TP2.pdf
│   ├── chaines.hs
│   └── tarot.hs
└── TP3/
```

- Construire un arbre `arbre_TP :: Tree String` représentant les contenus du dossier de TP. Sur chaque noeud vous indiquerez le nom du dossier (sans `/` final) ou du fichier.
- Écrire une fonction `descendre :: Tree String -> String -> Maybe (Tree String)` prenant en paramètre un arbre (supposé être un dossier), le nom d'une entrée, et renvoie le sous-arbre correspondant à cette entrée du dossier. Si l'arbre n'est pas un dossier ou l'entrée demandée n'existe pas, la fonction devra renvoyer `Nothing`.

- Utiliser `descendre` dans un appel bien choisi à une fonction `fold` pour obtenir une fonction `descendre_chemin :: Tree String -> [String] -> Maybe (Tree String)` qui descende le long d'une série de dossiers/fichiers. Par exemple, `descendre_chemin arbre_TP ["TP2", "tarot.hs"]` renverra le noeud du fichier `tarot.hs`.
- Écrire une fonction `liste_fichiers :: Tree String -> [String]` qui génère la liste des chemins de tous les fichiers de l'arbre, c'est-à-dire `["CS222/TP1/CS222_TP1.pdf", "CS222/TP1/applis.hs", ...]`.

TP Haskell #4/7 (CS 222)

Grenoble INP – Esisar – année 2022-23

Durée : 1h30
Consignes et date de rendu : Voir Chamilo

Objectifs :

1. Pratique de définition et utilisation des classes de types.

Mise en place du TP

Réalisez le TP dans un répertoire de travail `CS222/TP4_VotreNom`. Chaque exercice `XXX.hs` sera accompagné d'un script GHCi, nommé `test_XXX.ghci`, contenant les expressions sur lesquelles vous avez testé votre code, et exécutable de la façon suivante dans GHCi :

```
:load XXX.hs
[... affichage terminant par Ok, one module loaded.]
:script test_XXX.ghci
[... affichage de l'évaluation de vos tests]
```

Consignes communes pour l'écriture des fonctions Haskell :

- Chaque définition de fonction sera précédée de sa déclaration de type.
- Privilégier la réutilisation des fonctions écrites précédemment.
- Privilégier le filtrage de motif (*pattern-matching*) à du code conditionnel.
- N'hésitez pas à utiliser les fonctions de la bibliothèque standard de Haskell.

Exercice 1 — Agrégation générique avec la classe `Foldable`

Fichiers à rendre : `Foldable.hs`, `test_foldable.ghci`.

Dans le TP précédent, nous avons utilisé les fonctions `foldl` et `foldr` pour calculer des agrégations sur des listes:

```
my_foldl :: (b -> a -> b) -> b -> [a] -> b
my_foldr :: (a -> b -> b) -> b -> [a] -> b
```

L'intuition est que beaucoup de structures qui ne sont pas des listes peuvent être *parcourues* comme des listes, et ainsi se prêter à une agrégation. Par exemple, dans l'exercice sur les arbres binaires de recherche, on a parcouru un ABR d'une façon bien choisie pour obtenir une liste triée de ses éléments. On peut écrire une fonction d'agrégation sur l'arbre qui se comporte comme `foldr` sur la liste triée.

- Redéfinir le type `Bintree a` des arbres binaires portant des valeurs de type `a`, vu au TP3 (sans relire le sujet, si vous y arrivez !).
- Écrire la fonction `parcours_infixe :: Bintree a -> [a]` qui calcule la liste des valeurs de l'arbre dans l'ordre infixe (c'est-à-dire que pour un noeud interne on met d'abord les valeurs du sous-arbre de gauche, ensuite la valeur portée par le noeud lui-même, ensuite les valeurs du sous-arbre de droite). Cette fonction devra générer la même séquence que la fonction `aplatir_abr` du TP3. Écrire les tests associés.

- Écrire maintenant une fonction `foldr_arbre :: (a -> b -> b) -> b -> Bintree a -> b`, qui accumule de la même façon que `foldr` les éléments de l'arbre pris dans l'ordre infixe. Pour cette fonction, vous n'utiliserez pas `parcours_infixe` (écrivez une nouvelle fonction récursive).
- Dans vos tests dans `test_foldable.ghci`, vous prendrez soin de comparer `foldr_arbre` avec `foldr` appliqué au résultat de `pacours_infixe` (qui doivent être identiques).

Il est courant, à la fois dans la bibliothèque standard Haskell et dans les applications, d'écrire des fonctions qui parcourent et accumulent des valeurs en utilisant `foldr` ou un équivalent. Haskell nous fournit la classe de types `Foldable` pour capturer de façon générique ce concept ; ainsi, n'importe quelle fonction qui appelle `foldr` marche sur les arbres sans avoir besoin de mentionner `foldr_arbre`.

- Inspecter le type de `foldr` dans GHCi avec la commande `:type`. Remarquez l'argument de type `t a`. Quelles sont les deux "valeurs" de `t` (constructeurs de types) qu'on a étudiées jusqu'ici ?
- Écrire une fonction `to_list :: Foldable t => t a -> [a]` qui transforme une structure de données en liste en utilisant `foldr` appliqué à un opérateur binaire bien choisi.
- Définir une nouvelle instance de `Foldable` pour le constructeur de types `Bintree` en utilisant `foldr_arbre` comme définition de la fonction `foldr`. Cela suffit pour obtenir accès à *l'intégralité* des fonctions de la classe `Foldable` (dont vous pouvez obtenir la liste avec la commande `:info Foldable`), ainsi que la fonction `to_list` définie précédemment.

On rappelle ici la syntaxe pour les instances de type:

```
instance <classedeType> <untype> where
    <fundef1> = ...
```

- Sans définir de nouvelle fonction, calculer la somme des éléments d'un arbre d'entiers à l'aide de la fonction standard `sum`. Expliquer en commentaire le type de `sum`.

Interlude — Un programme d'ordonnancement de tâches

À titre d'application de Haskell, on va maintenant construire un programme de traitement de tâches dont le rôle sera de lire et exécuter des tâches de calcul. On imagine que ce programme sera lancé sur un serveur ; il aura une liste de tâches en attente, qu'il exécutera selon un ordre de priorité. Dans le cas où le programme a encore des tâches en attente lorsqu'on l'arrête, il les sauvegardera dans un fichier pour reprendre leur exécution plus tard.

Ce programme sera construit progressivement : on va explorer divers aspects comme le stockage dans des fichiers ou la gestion des priorités dans des exercices individuels sur chaque TP, et on réunira le tout pour construire l'application finale au dernier TP.

Exercice 2 — Sérialisation JSON

Fichiers à rendre : `Json.hs`, `test_json.ghci`.
Cet exercice fait partie du programme d'ordonnancement de tâches.

JSON est un format de données extrêmement répandu (né comme une description d'objets en Javascript mais généralisé depuis), utilisé pour représenter sous forme de texte des données structurées. Une valeur JSON peut être :

- Une constante : entier (`42`), flottant (`7.3`), chaîne (`"JSON!"`), booléen (`true`), ou `null`.
- Un tableau de valeurs, noté entre crochets : `[42, true, "JSON!", [false, 0]]`.

- Un “objet” (en réalité un dictionnaire), noté entre accolades, avec des chaînes de caractères comme clés : `{"entier": 42, "texte": "Haskell!", "tableau": [true]}`.

JSON est fréquemment utilisé pour *sérialiser* des données, c’est-à-dire les représenter sous forme d’une simple séquence (ici de caractères) sans pointeurs. La sérialisation est très utile pour stocker des données (par exemple une sauvegarde dans un jeu) ou pour les transmettre (par exemple répondre à une requête web sur le réseau). Le format est de plus indépendant des langages de programmation et des architectures. L’opération consistant à reconstruire la donnée originale s’appelle *désérialiser*.

Dans le programme d’ordonnancement de tâches, la sérialisation nous sera utile pour sauvegarder dans un fichier la liste des tâches en attente à la fin de l’exécution.

On se donne pour représenter des données JSON dans un format simplifié avec des constantes de type `Integer`, `Bool` et `String` :

```haskell
data JSON =
  JSON_Int Integer |
  JSON_Bool Bool |
  JSON_String String |
  JSON_Array [JSON] |
  JSON_Object [(String, JSON)]
```

- Définir les constantes `exemple_tableau :: JSON` et `exemple_objet :: JSON` encodant le tableau et l’objet donnés en exemples ci-dessus. Les valeurs construites avec `JSON_Array` contiennent d’autres valeurs de type `JSON`. Quelle structure de données vue précédemment avait aussi cette caractéristique ?

À ce stade, on n’a pas de quoi afficher les valeurs de type `JSON` dans le terminal. Pour cela, il nous faudrait une instance de la classe de types `Show`, dont la fonction principale est `show :: a -> String`.

- Définir une fonction `show_json :: JSON -> String` qui donne la notation d’une valeur JSON comme dans les exemples introductifs. On utilisera `show` pour obtenir la représentation textuelle des entiers et des chaînes de caractères, et la fonction intercalate peut aider.
- Instancier `JSON` dans la classe en `Show` en fournissant une unique fonction `show` égale à `show_json`. Constater qu’on peut maintenant évaluer `exemple_tableau` ou `exemple_objet` directement dans GHCi et obtenir en réponse leur notation JSON. Ajouter des tests de ce comportement dans `test_json.ghci`.

On se donne maintenant le type suivant correspondant à une tâche de calcul : soit afficher un nombre, soit afficher la somme de deux nombres.

```haskell
data Task =
  PrintVal Integer |
  PrintSum Integer Integer
```

- Écrire une fonction `serialize_task :: Task -> JSON` qui encode une tâche en JSON :
  - `PrintVal <x>` sera encodé par `{"op": "PrintVal", "x": <x>}` ;
  - `PrintSum <x> <y>` sera encodé par `{"op": "PrintSum", "x": <x>, "y": <y>}`.

  Ajouter quelques tests dans `test_json.ghci` si ce n’est pas encore fait.
- Écrire à l’inverse une fonction `deserialize_task :: JSON -> Maybe Task` qui essaie de décoder une valeur JSON correspondant à une tâche. Vous renverrez `Nothing` si la valeur JSON ne correspond à aucun des deux encodages valides. (Remarquez que grâce au filtrage de motifs il suffit presque d’inverser les arguments et valeurs de retour de `serialize_task`.)

Comme la séralisation est une tâche très courante, on aimerait bien ne pas avoir une fonction `serialize_X` pour tous les types intéressants (on aura notamment besoin de sauvegarder une *liste* de tâches dans le programme). On définit donc une classe de types pour avoir les mêmes fonctions `serialize` et `deserialize` pour tous les types:

```
class Serializable a where
  serialize :: a -> JSON
  deserialize :: JSON -> Maybe a
```

- Construire une instance `Serializable Task` en utilisant les fonctions définies précédemment.
- Construire une instance `Serializable a => Serializable [a]` que l'on utilisera pour sérialiser des listes de valeurs sous la forme d'un tableau JSON. La contrainte `Serializable a` vous permet d'utiliser `serialize` et `deserialize` sur les éléments de la liste. Dans la fonction de désérialistion, vous renverrez un résultat (`Just`) uniquement si tous les éléments ont pu être désérialisés ; si l'un d'eux échoue, renvoyez `Nothing`.

Exercice 3 — Tri fusion

Cet exercice est facultatif ; vous pourrez rendre les fichiers `Fusion.hs` et `test_fusion.ghci`.

L'objectif de cet exercice est de coder en Haskell un tri fusion sur des listes d'entiers, sous la forme d'une fonction `tri_fusion :: [Int] -> [Int]`.

- Écrire la fonction `halve :: [Int] -> ([Int], [Int])` qui coupe une liste en deux listes de tailles égales (à une unité près). Ne pas utiliser la longueur de la liste. Il n'est pas nécessaire de garder l'ordre des éléments de l'entrée !
- Écrire la fonction `combine :: [Int] -> [Int] -> [Int]` qui prend deux listes triées par ordre croissant en entrée, et combine ces deux listes en une unique liste triée par ordre croissant.
- En utilisant les fonctions `halve` et `combine`, écrire la fonction `tri_fusion` Il faudra traiter séparément les cas des listes à 0 et 1 élément.

TP Haskell #6/7 (CS 222)

Grenoble INP – Esisar – année 2022-23

Durée : 1h30
Consignes et date de rendu : Voir Chamilo

Objectifs :

1. Obtenir l'intuition d'une monade comme un nouvel environnement d'exécution.
2. Utiliser les opérateurs `fmap` (`<$>`) et `bind` (`>>=`) dans des cas simples.

Mise en place du TP

Réalisez le TP dans un répertoire de travail `CS222/TP6_VotreNom`. Chaque exercice `XXX.hs` sera accompagné d'un script GHCi, nommé `test_XXX.ghci`, contenant les expressions sur lesquelles vous avez testé votre code, et exécutable de la façon suivante dans GHCi :

```
:load XXX.hs
[... affichage terminant par Ok, one module loaded.]
:script test_XXX.ghci
[... affichage de l'évaluation de vos tests]
```

Consignes communes pour l'écriture des fonctions Haskell :

- Chaque définition de fonction sera précédée de sa déclaration de type.
- Privilégier la réutilisation des fonctions écrites précédemment.
- Privilégier le filtrage de motif (*pattern-matching*) à du code conditionnel.
- N'hésitez pas à utiliser les fonctions de la bibliothèque standard de Haskell.

Exercice 1 — Propagation automatique des erreurs

Fichiers à rendre : `Erreurs.hs`, `test_Erreurs.ghci`.

Une des monades les plus courantes est la monade associée au type `Maybe`, qui permet de gérer des erreurs. L'objectif de cet exercice est de manipuler des fonctions qui peuvent échouer, et de montrer que les monades en simplifient l'usage au point où on peut s'en servir sans se préoccuper constamment des erreurs.

On se donne la liste des planètes du système solaire, accompagnées de leur distance au Soleil moyenne en Unités Astronomiques (UA), dans l'ordre croissant.

```haskell
planetes :: [(Double, String)]
planetes = [
  (0.39, "Mercure"),    (0.72, "Venus"),      (1.00, "Terre"),
  (1.52, "Mars"),       (5.20, "Jupiter"),    (9.54, "Saturne"),
  (19.2, "Uranus"),     (30.1, "Neptune")]
```

- Écrire une fonction `distance :: String -> Maybe Double` qui prend en paramètre un nom de planète et renvoie sa distance au Soleil. Si le nom demandé n'est pas présent dans la liste, la fonction renverra `Nothing`. *(Vous aurez sans doute besoin d'une sous-fonction qui prend en paramètre la liste dans laquelle on cherche.)*

- Écrire une fonction `suivante :: Double -> Maybe String` qui prend en paramètre une distance `d` en UA et renvoie le nom de la première planète dont la distance au Soleil est strictement supérieure à `d`. S'il n'y en a pas, la fonction renverra `Nothing`. Il vous est conseillé :
  1. D'utiliser `filter` pour obtenir la liste des planètes à distance strictement supérieure à `d` du Soleil ;
  2. Dans le cas où la liste est vide, renvoyer `Nothing`, sinon renvoyer le nom de la première planète.
- Écrire une fonction `ua_vers_km :: Double -> Double` qui convertit une distance d'UA en km (multiplication par `1.496e+8`).

Nous avons maintenant deux fonctions dont le type est de la forme `a -> Maybe b`, que l'on peut comprendre intuitivement comme : "une fonction de type `a -> b` mais qui peut échouer". C'est un bon début, mais on ne peut pas facilement les composer. En effet, une fonction `f :: a -> Maybe b` ne peut être composée ni avec une fonction `g :: b -> c` ni avec une fonction `h :: b -> Maybe c`, puisque `b` et `Maybe b` sont deux types différents. Plus concrètement, *on ne peut pas continuer le calcul avec `g` ou `h` tant qu'on n'a pas vérifié si `f` a renvoyé une erreur.*

Cas où l'on continue avec une fonction qui ne peut pas échouer

- Écrire une fonction `distance_km_1 :: String -> Maybe Double` qui associe à un nom de planète sa distance au Soleil en km, en appelant `distance` puis `ua_vers_km`.

Remarquez que vous avez besoin de filtrer la valeur renvoyée par `distance_planete` pour déterminer si une erreur s'est produite ; c'est un peu fastidieux.

- Écrire une fonction `fmap_Maybe :: (a -> b) -> Maybe a -> Maybe b` qui prend en argument une fonction et une valeur de type `Maybe a`. Elle renverra `Nothing` si la valeur est `Nothing` et appliquera la fonction sous le `Just` sinon.

Cette fonction est disponible dans la bibliothèque standard ; vous l'appelerez par son nom `fmap` ou via l'opérateur infixe `<$>`. Intuitivement, `<$>` *applique une fonction dans le cas `Just` et ne fait rien dans le cas `Nothing`.*

- En déduire une version améliorée `distance_km_2 :: String -> Maybe Double` qui ne fait pas de filtrage de motif.

Cas où l'on continue avec une fonction qui peut échouer

- Écrire une fonction `distance_suivante_1 :: String -> Maybe Double` qui prend en paramètre le nom d'une planète et renvoie la distance au Soleil de la planète qui la suit. Par exemple, sur l'entrée `"Terre"` la fonction renverra la distance entre le Soleil et Mars.

  Pour cette fonction, vous devrez appeler `distance`, puis `suivante`, et de nouveau `distance`. Ces appels pouvant échouer, il vous est conseillé d'inspecter chaque valeur de retour à l'aide d'une expression `case`, comme ceci :

  ```
  case distance p of
    Nothing -> ... {- erreur dans le calcul de la distance -}
    Just d  -> ... {- la distance est d -}
  ```

  Si vous ne l'avez pas encore fait, ajoutez des tests dans `test_erreurs.ghci`.

La fonction précédente est un bon début, mais ça fait beaucoup de code très répétitif pour propager le `Nothing`. Comme précédemment, on veut donc écrire une fonction auxiliaire qui ferait ça à notre place.

- Expliquer pourquoi l'opérateur `<$>` (`fmap`) n'est pas suffisant pour cette tâche. Vous pourrez analyser les types des fonctions et comparer aux fonctions `f`, `g` et `h` de l'introduction.
- Écrire une fonction `bind_Maybe :: Maybe a -> (a -> Maybe b) -> Maybe b` qui renvoie `Nothing` sur l'entrée `Nothing` et appelle la fonction fournie sur l'entrée `Just a`. Expliquer la différence entre `fmap` et `bind_Maybe` en termes de capacité à renvoyer des erreurs.

Cette fonction est disponible dans la bibliothèque standard ; vous l'appelerez par son nom `bind` ou via l'opérateur infixe `>>=`. Intuitivement, `>>=` sert à *poursuivre un calcul en restant dans le monde où les fonctions peuvent renvoyer des erreurs.*

- Écrire une version améliorée `distance_suivante_2 :: String -> Maybe Double` qui utilise `>>=` et des lambdas à la place des `case`.

On rappelle qu'en Haskell un bloc `do` peut être utilisé pour enchaîner plusieurs calculs connectés par des bind (`>>=`). La syntaxe est la suivante :

```haskell
do x <- un_calcul
   y <- un_autre_calcul
   valeur_retour

-- Équivalent à:
un_calcul >>= (\x ->
  un_autre_calcul >>= (\y ->
    valeur_retour))
```

- Écrire une dernière version `distance_suivante_3 :: String -> Maybe Double` qui utilise un bloc `do`. Remarquez que le code ne fait plus du tout de filtrage : les erreurs sont propagées automatiquement !

Exercice 2 — Gestion des entrées/sorties

Fichiers à rendre : `IO.hs`, `test_IO.ghci`.
Les concepts de cet exercice seront utilisés dans le programme d'ordonnancement de tâches.

Avec les monades, nous pouvons enfin aborder les entrées-sorties. Leur existence semble contre-intuitive car Haskell est un langage pur (sans effets de bord). Cette pureté a plein d'avantages ; par exemple, deux appels de fonction identiques renvoient forcément la même valeur, et on peut calculer les expressions dans l'ordre qu'on veut. Pourtant deux appels à `scanf()` en C ne renvoient pas forcément la même valeur, et changer l'ordre de deux appels à `printf()` change le comportement du programme. Alors comment concilier ces deux aspects ?

La monade des entrées-sorties, `IO`, résoud ce problème. L'idée majeure c'est qu'une fonction comme `print :: Show a => a -> IO ()` qui affiche une valeur dans le terminal, n'affiche en fait pas la valeur, mais renvoie une *action* qui peut être exécutée plus tard. Une valeur de type `IO T` c'est donc une action qui, quand elle est exécutée, fait des entrées-sorties puis renvoie une valeur de type `T`.

Les actions restent non exécutées jusqu'à ce que le code soit lancé dans GHCi. Vous avez déjà vu que si vous tapez une expression comme `1+2`, GHCi vous affiche leur valeur. En plus de ce comportement, si vous saisisez une action comme `print (1+2)` dont le type est `IO T`, alors GHCi exécute l'action puis affiche le résultat.

- Écrire une fonction `afficher_avec_etoiles :: Show a => a -> IO ()` qui utilise `show` et `print` pour afficher une valeur avec trois étoiles de chaque côté. Testez-la ensuite dans GHCi comme suit :

```
ghci> afficher_avec_etoiles 3.14
***3.14***
```

Ajoutez dans `test_io.ghci` un test similaire.

- Écrire une fonction `echo :: IO ()` qui lit une chaîne de caractères dans le terminal avec `getLine :: IO String` puis l'affiche avec `putStrLn :: String -> IO ()`. Contrairement au `Maybe` de l'exercice précédent vous ne pouvez pas filtrer une valeur de type `IO ()`, vous devez utiliser `bind` (`>>=`) ou un bloc `do`.

On veut maintenant écrire une fonction similaire au programme Unix `wc` (*word count*), qui compte le nombre de lignes, mots et octets dans un fichier.

- Écrire une fonction `wc :: String -> IO ()` qui prend en paramètre le nom d'un fichier et affiche ses statistiques au format suivant :

  `<nombre de lignes> <nombre de mots> <nombre d'octets> <nom du fichier>`

  Par exemple:

  `22 96 518 IO.hs`

  On utilisera la fonction `readFile :: FilePath -> IO String` (où `FilePath` est un alias de `String`) pour obtenir les contenus du fichier. Ajoutez de plus `import Data.List` au début de votre fichier pour accéder aux fonctions `words` et `lines` qui génèrent la liste des mots et lignes d'une chaîne de caractères, respectivement.

- Ajouter une fonction `main :: IO ()` qui récupère les arguments de ligne de commande avec `getArgs :: IO [String]` (disponible après `import System.Environment`) et appelle `wc`.

De cette façon, vous pouvez compiler un exécutable et l'appeler comme le `wc` classique d'Unix.

```
% ghc --make IO.hs
(...)
% ./IO IO.hs
22 96 518 IO.hs
```


Exercice 3 — Monade des listes et problème du cavalier

Cet exercice est facultatif ; vous pourrez rendre les fichiers `Cavalier.hs` et `test_cavalier.hs`.

Dans le premier exercice, on a vu comment la monade `Maybe` modélise le comportement des programmes qui peuvent échouer. Une autre monade intéressante est la monade des listes, qui modélise des fonctions *non-déterministes* (ie. des fonctions qui peuvent faire des *choix*). Une fonction `a -> b` mais qui est non-déterministe sera codée par une fonction `a -> [b]` ; la liste représente toutes les valeurs possibles que la fonction peut renvoyer à l'issue de ses choix (qui sont invisibles quand on l'appelle).

Ces fonctions sont notamment utiles pour résoudre les problèmes *d'exploration* comme le problème du cavalier. Dans ce problème, un cavalier est placé sur un échiquier de m lignes et n colonnes. Le cavalier se déplace « en L » comme illustré ci-dessous. La question est la suivante : est-il possible de déplacer successivement le cavalier pour lui faire visiter toutes les cases de l'échiquier sans passer deux fois par la même case ?

Un programme pour résoudre ce problème pourra *explorer* les différentes options de déplacement du cavalier à chaque étape du jeu. Généralement il y en a plusieurs, chacune donnant un résultat différent, et offrant plusieurs nouvelles options. Un programme non-déterministe utile dans cette situation est

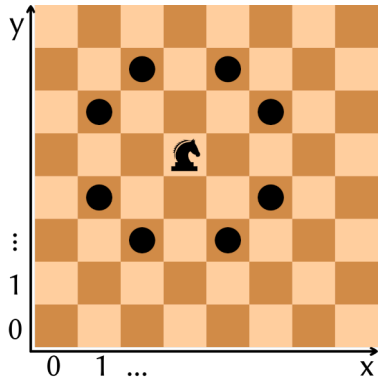

Figure 1: Mouvement du cavalier et coordonnées sur l'échiquier.

celui qui *choisit* un déplacement à chaque étape, et continue jusqu'à trouver une solution ou tomber à court d'options. Pour savoir s'il y a une solution, il suffit de regarder toutes les exécutions possibles du programme non-déterministe, et la monade des listes en Haskell nous aide à calculer ça.

On vous fournit le fichier `ProblemeCavalier.hs` modélisant le problème pour un échiquier de taille 4×3 ; les quelques fonctions fournies sont expliquées dans les commentaires. (Il ne vous est pas demandé de comprendre le code, simplement d'utiliser les fonctions.) Dans `Cavalier.hs` vous ajouterez `import ProblemeCavalier` pour utiliser le module.

- Écrire une fonction `mouvements :: Etat -> [Position]` qui sur l'entrée `(p, (x, y))` indique toutes les cases que le cavalier placé en `(x, y)` peut atteindre. Il y a 8 mouvements en L possibles, mais vous ne devez garder que les cases qui sont dans les bornes de l'échiquier et pas déjà visitées dans `p`.
- En déduire une fonction `etats_suivants :: Etat -> [Etat]` qui liste tous les états possibles du jeu après un mouvement du cavalier. On notera que si le cavalier est bloqué (et donc la partie perdue) la liste renvoyée est vide.

On peut voir `etats_suivants` comme une fonction non-déterministe qui *choisit* un mouvement possible et renvoie l'état correspondant. L'opérateur `>>=` (même notation que pour `Maybe`) nous permet dans la suite de continuer ce calcul avec une autre fonction non-déterministe et ainsi d'enchaîner les choix. Dans ce monde non-déterministe, `l >>= g` est la liste de toutes les valeurs que `g` peut renvoyer (en fonction de ses choix internes) après avoir été appelée sur n'importe laquelle des valeurs de `l`.

- Écrire une fonction `explorer_etats :: Etat -> [Etat]` qui détermine tous les états gagnants accessibles à partir de l'état fourni.
  - Si l'état fourni est gagnant (ie. `plateau_plein p` est vrai), alors on renverra une liste contenant juste cet état ; la recherche s'arrête.
  - Sinon, on calculera les états suivants (dans la vision non-déterministe, on *choisira* un de ces états) et on utilisera un appel récursif pour continuer l'exploration.
- En déduire une fonction `etats_gagnants :: Position -> [Etat]` qui liste tous les états gagnants quand le cavalier commence à la position indiquée.
- Calculer la liste `positions_initiales_realisables :: [Position]` de toutes les positions initiales à partir desquelles le cavalier *peut* parcourir tout l'échiquier en passant exactement une fois par chaque case. Il vous est conseillé d'utiliser une compréhension de listes pour énumérer les coordonnées initiales.

Exercice 4 — Dessin du dragon de Heighway avec les entrées/sorties

Cet exercice exploratoire est facultatif ; vous pourrez rendre `Dragon.hs` et `test_Dragon.ghci`.

On vous fournit le module `Pics` contenant une infrastructure de dessin utilisant la bibliothèque Haskell SVG. (Vous pouvez l'installer avec `cabal install svg-builder --lib` sur vos machines perso, en espérant que la librairie existe en salle de TP.)

- Lire et prendre en main le code fourni. Dessiner des choses dans `test_Dragon.hs`, par exemple des séries de maisons. :)

Vous pouvez générer un fichier SVG à partir d'une `Picture` avec la fonction `output` de `Pics.hs`. Pour visualiser, vous pouvez utiliser ImageMagick pour faire un rendu JPG :

```
% convert image.svg image.jpg
```

- Pour comprendre le code fourni, on pourra se reporter à doc wiki qui explique la structure mathématique simple de monoïde, et on pourra répondre aux questions suivantes (GHCi est votre ami):
  - quels dessins sont codés par le type de données `Picture`?
  - que fait la fonction `polyline`?
  - que fait `(°/)`?
  - que fait `rotateP`?
  - à quoi servent les arguments de `mkLandscape` ?
  - (il n'est pas nécessaire de comprendre toSvg)
- Écrire une fonction récursive `dragon :: Int -> Float -> Picture` qui calcule un dragon de Heighway. Le premier argument désigne le nombre d'appels récursifs, le deuxième la taille de chaque segment.

On pourra utiliser une fonction auxiliaire qui retourne non seulement la figure courante mais aussi le dernier point calculé.

- Générez un dragon avec 14 appels récursifs et des segments de longueur 8 pour obtenir la figure ci-dessous.

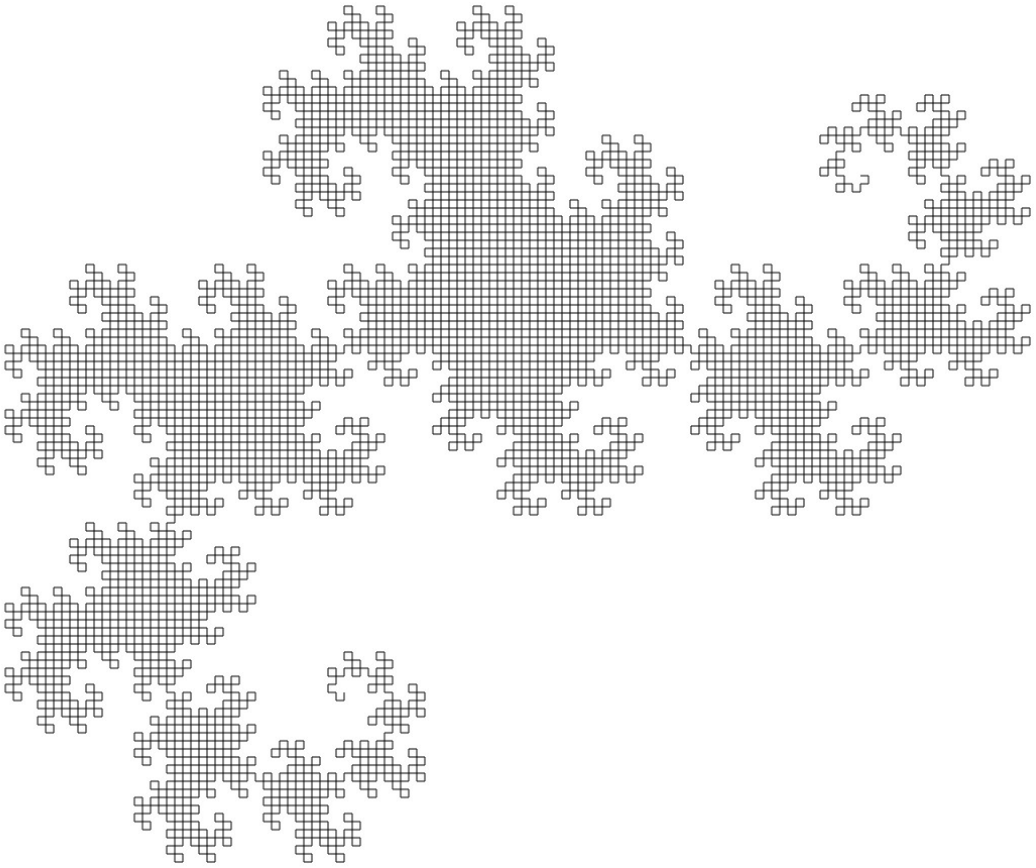

Figure 2: Dragon de Heighway

TP Haskell #7/7 (CS 222)

Grenoble INP – Esisar – année 2022-23

Durée : 1h30
Consignes et date de rendu : Voir Chamilo

Objectifs :

1. Construire des parsers utiles en combinant des parsers simples.
2. Combiner les notions vues en TP précédemment dans un gros programme.

Mise en place du TP

Réalisez le TP dans un répertoire de travail `CS222/TP7_VotreNom`. Chaque exercice `XXX.hs` sera accompagné d'un script GHCi, nommé `test_XXX.ghci`, contenant les expressions sur lesquelles vous avez testé votre code, et exécutable de la façon suivante dans GHCi :

```
:load XXX.hs
[... affichage terminant par Ok, one module loaded.]
:script test_XXX.ghci
[... affichage de l'évaluation de vos tests]
```

Consignes communes pour l'écriture des fonctions Haskell :

- Chaque définition de fonction sera précédée de sa déclaration de type.
- Privilégier la réutilisation des fonctions écrites précédemment.
- Privilégier le filtrage de motif (*pattern-matching*) à du code conditionnel.
- N'hésitez pas à utiliser les fonctions de la bibliothèque standard de Haskell.

Exercice 1 — Parser JSON

Fichiers à rendre : `JSONParser.hs`, `test_JSONParser.ghci`.
Cet exercice fait partie du programme d'ordonnancement de tâches.

Un *parser* est un programme qui analyse une chaîne de caractères et construit une valeur correspondant au texte. En un sens, c'est l'inverse de `show` : `show` prend une valeur et renvoie une chaîne de caractères qui la représente ; un parser prend cette chaîne de caractères et reconstruit la valeur.

L'objectif de cet exercice est de construire une fonction `pJSON` qui inverse `show` sur les valeurs de type `JSON` utilisées au TP4 :

```haskell
pJSON "[12, true, [-8, -73]]"
-- Just (JSON_Array [JSON_Int 12, JSON_Bool True,
--                   JSON_Array [JSON_Int (-8), JSON_Int (-73)]],
--       "")
```

Note : Les parsers sont un sujet classique en Haskell et un exemple important de monade ; ici on utilise la monade `Maybe` mais pas celle des parsers, donc si vous croisez des parsers Haskell en ligne ou dans des livres la présentation sera sans doute différente.

Pour cela, on vous fournit le module `Parser.hs` qui présente la notion de parser, plusieurs exemples, et rappelle comment on peut utiliser `do` pour propager les erreurs (comme dans le TP6).

- Lire `Parser.hs` ; écrire dans `test_JSONParser.ghci` des tests des fonctions fournies.
- Dans `JSONParser.hs`, écrivez le parser `pBooleen :: Parser Bool` qui lit un booléen (en minuscules) au début d'une chaîne, après avoir sauté les espaces. Par exemple `pBooleen " true2"` renverra `Just (True, "2")`.

La représentation textuelle du JSON qu'on a vue au TP4 ne contient que des nombres, chaînes, booléens et des caractères de ponctuation (`[]{}:,`). Avec `pNombre`, `pChaine`, `pBooleen` et `pCaractere` on a donc tous les parsers élémentaires qu'il nous faut. On veut maintenant les mettre bout-à-bout pour construire des parsers plus compliqués comme celui des tableaux JSON.

Traditionnellement on dit qu'on « combine » les parsers et on appelle les fonctions de combinaison des *combinateurs*. `Parser.hs` fournit par exemple le combinateur `(>>>)` qui séquence deux parsers en ignorant la valeur de retour du premier.

Note : tous nos parsers élémentaires sautent les espaces en début de chaîne, ce sera donc automatiquement le cas des parsers combinés. Du coup on peut ignorer les espaces à partir de maintenant.

Pour les valeurs JSON, on a bien de quoi lire les nombres, entiers et booléens, mais on ne sait pas à l'avance quel type de valeur va être utilisé. Par exemple après avoir lu "`[1, true,`" on sait que le tableau doit continuer avec une nouvelle valeur JSON, mais on ne sait pas quel parser utiliser pour la lire. Il nous faut donc un moyen de « tenter » plusieurs parsers et de prendre le premier qui marche.

- Écrire un combinateur « alternative » `(<||>) :: Parser a -> Parser a -> Parser a` qui essaie deux parsers sur le même texte. `(p1 <||> p2) str` sera le résultat de la lecture avec `p1`, sauf si ce résultat est `Nothing`, auquel cas ce sera le résultat de la lecture avec `p2`.

On vous fournit un combinateur `cRepeter :: Parser a -> Parser [a]` qui répète le parser donné en argument jusqu'à ce qu'il échoue, et collecte les valeurs lues dans une liste. Par exemple sur des entiers, `cRepeter pNombre "1 2 3 x"` renvoie `Just ([1,2,3], " x")` et `cRepeter pNombre "x"` renvoie `Just ([], "x")`. Remarquez que `cRepeter p` n'échoue jamais.

- En déduire un combinateur `cRepeterVirgules :: Parser a -> Parser [a]` qui répète le parser donné en argument mais avec des virgules entre chaque occurrence. Par exemple, `cRepeterVirgules pNombre "1,2,3y"` renverra `Just ([1,2,3], "y")`.

  Vous pouvez recopier et ajuster le code de `cRepeter`, ou bien vous pouvez essayer *d'appeler* `cRepeter` avec un argument bien choisi.

On a maintenant tout ce qu'il nous faut pour lire du JSON et reconstruire les valeurs. Redéfinissez le type `data JSON` du TP4, cette fois avec `deriving (Eq, Show)` pour utiliser l'affichage par défaut :

```haskell
data JSON =
  JSON_Int Integer |
  JSON_Bool Bool |
  JSON_String String |
  JSON_Array [JSON] |
  JSON_Object [(String, JSON)]
  deriving (Eq, Show)
```

- Écrire trois parsers `pNombreJSON`, `pBooleenJSON` et `pChaineJSON` de type `Parser JSON` qui appellent `pNombre`, `pBooleen` et `pChaine` puis transforment le résultat en un `JSON` en utilisant le constructeur approprié. (L'opérateur `<$$>` de `Parser.hs` permet de le faire en une ligne.)

Il nous reste maintenant à traiter les tableaux et les objets, puis à tout combiner dans un seul parser `pJSON` qui testera les 5 types de valeurs. Comme les tableaux/objets contiennent des valeurs JSON qui peuvent elle-même être des tableaux/objets, les fonctions qui nous restent seront *mutuellement récursives* (les parsers vont s'appeler les uns les autres).

- Écrivez quatre parsers mutuellement récursifs :

```
pTableauJSON :: Parser JSON
pAttributJSON :: Parser (String, JSON)
pObjetJSON :: Parser JSON
pJSON :: Parser JSON
```

  - `pTableauJSON` lira un crochet ouvrant `[`, une liste séparée par des virgules de valeurs JSON avec `pJSON`, et un crochet fermant `]`.
  - `pAttributJSON` lira un attribut d'un objet (une chaîne avec `pChaine`, le séparateur `:`, puis une valeur JSON avec `pJSON`).
  - `pObjetJSON` lira une accolade ouvrante, `{`, une liste séparée par des virgules d'attributs, et une accolade fermante `}`.
  - `pJSON` lira un nombre, un booléen, une chaine, un tableau ou un objet JSON en utilisant (plusieurs fois) le combinateur `<|>`.

- Ajoutez des tests dans `test_JSONParser.hs`.

Exercice 2 — Programme d'ordonnancement de tâches

Cet exercice est facultatif ; vous pourrez rendre `Ordonnancement.hs` et `test_Ordonnancement.ghci`.

Dans cet exercice, on va finalement combiner les fichiers de plusieurs TP précédents pour construire une application complète. Dans votre dossier du TP7, réunissez les fichiers nécessaires :

- Une copie de `Json.hs` du TP4. Ajoutez `module Json where` au tout début du fichier ainsi que `deriving (Eq, Ord)` au type `Task`.
- Une copie de `Tas.hs` du TP5. Ajoutez `module Tas where` au tout début du fichier s'il n'y est pas déjà.
- Ajoutez `module JSONParser where` au début de `JSONParser.hs` et supprimez la copie de `data JSON` au profit d'un `import Json` au début du fichier.

Vous devez pouvoir maintenant utiliser le fichier `Ordonnancement.hs` fourni. Au lieu d'utiliser GHCi, pour conclure le TP on va générer un exécutable directement en compilant sur la ligne de commande :

```
% ghc --make Ordonnancement.hs
Loaded package environment from (...)
[1 of 5] Compiling Json             ( Json.hs, Json.o )
[2 of 5] Compiling Parser           ( Parser.hs, Parser.o )
[3 of 5] Compiling JSONParser       ( JSONParser.hs, JSONParser.o )
[4 of 5] Compiling Tas              ( Tas.hs, Tas.o )
[5 of 5] Compiling Main             ( Ordonnancement.hs, Ordonnancement.o )
Linking Ordonnancement ...
```

Notez que le programme a besoin d'un paquet qui s'appelle `strict`. S'il n'est pas installé, vous pouvez l'installer avec cabal :

```
% cabal install --lib strict
```

En plus de quelques fichiers `.o` et `.hi`, la compilation produit le fichier exécutable `Ordonnancement` que vous pouvez lancer avec `./Ordonnancement`. Vous pouvez toujours charger le projet dans GHCi avec `:load Ordonnancement.hs` si vous voulez tester interactivement.

Le but du programme d'ordonnancement de tâches est de traiter des tâches de calcul (le type `Task` de `Json.hs`) avec un ordre de priorité. Les tâches seront stockées dans un tas (de `Tas.hs`) avec leur

priorité. L'utilisateur commandera interactivement l'exécution ou l'ajout de nouvelles tâches. À la fin du programme, s'il reste des tâches non traitées, elles seront sérialisées (avec `serialize` de `Json.hs`) et stockées dans le fichier `taches.txt`, d'où elles seront rechargées à l'exécution suivante.

- Lisez `Ordonnancement.hs` et identifiez globalement à quoi sert chaque fonction. (Vous n'avez pas besoin de comprendre tous les détails pour faire l'exercice.)

Le programme s'utilise de façon interactive. Au démarrage, l'aide suivante est affichée :

```
Commandes:
  e: Exécuter une tâche (s'il y en a)
  a <priorité> <valeur>: Ajouter une tâche PrintVal
  a <priorité> <valeur>+<valeur>: Ajouter une tâche PrintSum
  q: Quitter
```

Par exemple, si au début de l'exécution la file de tâches est vide :

- `a 4 2+3` ajoutera une première tâche `PrintSum 2 3` de priorité 4
- `a 2 1` ajoutera une tâche `PrintVal 1` de priorité 2 (ie. élevée, ça marche à l'envers)
- `e` exécutera `PrintVal 1` et afffichera donc 1
- `e` exécutera `PrintSum 2 3` et affichera donc 5
- Enfin, `q` quittera le programme.

Pour l'instant, aucune fonction n'est codée, donc n'importe quelle commande arrête le programme.

Vous avez quatre fonctions à compléter :

- `afficher_prio_tache` doit simplement afficher dans le terminal une paire (priorité, tâche). La fonction `interactif` affiche automatiquement la liste des tâches en attente (par ordre de priorité) avant chaque demande de commande en appelant `afficher_prio_tache`.
- `executer_tache` doit exécuter une tâche en affichant soit une valeur entière soit la somme de deux valeurs entières.
- La fonction la plus intéressante est `action`, qui prend en entrée une commande saisie par l'utilisateur (à savoir `e`, `q` ou une version de `a`) et l'exécute. `action` renvoie une paire ; le booléen indique si l'exécution doit continuer (c'est donc `False` sur l'entrée `q` et vrai le reste du temps) et la file de tâches restant après la commande. Vous pouvez utiliser `decoder_tache` pour convertir l'argument de `a` en une paire `(Integer, Task)`.
- Enfin, `sauver_file` devra sauver dans le fichier `"taches.txt"` la file restant à la fin de l'exécution, en utilisant la fonction standard `writeFile`. Il n'y a pas grand intérêt à stocker un `Tas` ; à la place, générez la liste des tâches avec `deconstruit` et sérialisez-la en JSON. `charger_file` se chargera de décoder tout ça au prochain lancement.

Vous pouvez alors tester le programme complet et même enregistrer des tâches en attente d'une exécution à la suivante.

```
% ./Ordonnancement
(...)
Liste des tâches :
> a 4 2+3
Liste des tâches :
  4: PrintSum 2 3
> a 2 1
Liste des tâches :
  2: PrintVal 1
  4: PrintSum 2 3
```

```
> e
1
Liste des tâches :
  4: PrintSum 2 3
> q
% ./Ordonnancement
(...)
Liste des tâches :
  4: PrintSum 2 3
> e
5
Liste des tâches :
> q
```

Voilà qui conclut cette série de TP. Vous êtes maintenant solidement équipé·es pour lire et écrire du code en Haskell. Ce détour par la programmation fonctionnelle vous permettra d'apprécier, on l'espère, la force de monter en abstraction en factorisant les motifs récurrents.

SQL, Lisp et Haskell sont les seuls langages de programmation que j'ai vus où l'on passe plus de temps à réfléchir qu'à écrire.
— Philip Greenspun