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`. 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. À chaque percolation la machine produit 4 tasses de café.

- 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 si la machine n'est pas pleine.
  - `attendre` patientera jusqu'à la fin de la percolation en cours (produisant ainsi jusqu'à 4 cafés).
  - `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.