Paradigmes et Langages de Programmation

Haute École d'Ingénierie et de Gestion du Canton de Vaud

Devoir 1

2022

1 Introduction

Ce devoir va consister pour vous en l'implémentation Haskell d'une série de cinq exercices de difficulté variée. Certains de ces exercices nécessitent d'être familier avec les entrées-sorties, les exceptions et la compilation en Haskell. Ces sujets seront introduits lors de la prochaine séance de cours. Il n'en demeure pas moins que vous pouvez d'ores et déjà commencer l'implémentation de tous les exercices et terminer ce qu'il manque une fois le matériel restant introduit. Vous êtes toutefois libres de prendre de l'avance sur le cours et de lire les chapitres correspondants de l'ouvrage Apprendre Haskell vous fera le plus grand bien !.

2 Exercices

Cinq exercices vous sont proposés dans ce devoir, que vous êtes libres de réaliser seul ou en binôme. Il s'agit de problèmes de programmation de taille moyenne qui, ensembles, ont pour but de couvrir autant que possible les caractéristiques de la programmation fonctionnelle et les fonctionnalités du langage Haskell. Les exercices sont présentés dans ce document par ordre de difficulté croissante. La donnée de chaque exercice se veut être délibérément concise et exige de votre part un certain degré d'investissement quant à l'élaboration de vos solutions et la recherche voire demande de compléments d'information. Notez que tout comportement non-spécifié dans l'énoncé signifie que vous pouvez l'adresser comme bon vous semble. En cas de doute, utilisez sans autre le forum de discussions pour poser vos questions.

2.1 Numération romaine

Écrivez un module Haskell, **roman.hs**, qui permet de convertir un nombre entier positif (≤ 3999) vers sa représentation en numération romaine, et vice versa. Les chiffres romains modernes sont écrits en exprimant chaque chiffre séparément, en commençant par le chiffre le plus à gauche et en sautant tout chiffre ayant une valeur de zéro.

Votre module doit mettre à disposition deux fonctions permettant de faire la conversion dans les deux sens. Ces fonctions doivent générer une erreur au cas où un argument incohérent leur est transmis. À vous de déterminer quel mécanisme de gestion d'erreur s'applique le mieux dans votre implémentation (il n'existe pas une seule manière de faire).

Voici quelques correspondances entre nombres entiers et nombres romains :

0	∅
1	I
12	XII
123	CXXIII
1234	MCCXXXIV
3999	MMMCMXCIX

2.2 Décompte de mots

Écrivez un programme Haskell, **word.hs**, qui affiche à l'écran le nombre de lignes, de mots et de caractères figurant dans chaque fichier donné, dont le nom est fourni en argument de ligne de commande. Certaines fonctions du module Data.Text vous seront utiles à cet égard. Votre programme doit en outre afficher le total de lignes, mots et caractères tout fichier confondu.

Voici un exemple d'exécution du programme demandé avec deux fichiers d'entrée et la sortie attendue, que vous devez respectez scrupuleusement :

```
> printf 'un deux\n' > tmp1
> printf 'trois\nquatre\n' > tmp2
> ghc word.hs
> ./word tmp1 tmp2
file          word   line   byte
tmp1             2      1      8
tmp2             2      2     13
total            4      3     21
```

La sortie attendue est constituée d'une entête, suivie d'une ligne par fichier laquelle indique le nom du fichier, le nombre de mots, le nombre de lignes et le nombres de caractères, et enfin le total tout fichier compris.

Votre programme doit finalement appliquer un formatage flexible lors de l'affichage du résultat ; il doit concrètement veiller au maintien ordonné des colonnes affichées. Pour ce faire, il vous est demandé d'utiliser la fonction *printf* du module Text.Printf.

Libre à vous de gérer ou non les erreurs d'entrées-sorties, c'est-à-dire d'éventuels problèmes d'exécution pouvant survenir lors de l'ouverture ou la lecture d'un fichier. En dehors de ces cas précis, votre programme doit fonctionner en toute circonstance.

2.3 Entiers de Peano

Écrivez un module Haskell, **natural.hs**, qui permet de manipuler des entiers de Peano. Un entier de Peano est un entier naturel exprimé en termes de zéro et de successeurs de zéro, un peu à la manière de la fonction standard *succ* du langage Haskell :

0 = Z
1 = S(Z)
2 = S(S(Z))
...
n = S(S(...S(Z)))

Afin de manipuler des entiers de Peano, il vous est demandé de définir un type algébrique *Nat* qui permet de représenter et exprimer de tels entiers. Votre module doit en plus exposer les fonctionnalités suivantes que vous devez implémenter uniquement sous formes de fonctions, sans utiliser des classes de types à l'exception peut-être de *Show* :

Arithmétiques	$+, -, *, \wedge$
Comparaisons	$=, \neq, <, \leq, >, \geq$
Conversions	Int $\longrightarrow$ Nat, Nat $\longrightarrow$ Int, Nat $\longrightarrow$ String
Fonctions	zero, succ, pred, isZero

On admettra que toute opération susceptible de retourner une valeur négative produira comme résultat le zéro de Peano. Inutile donc de lever une quelconque exception ou de générer une erreur d'exécution.

2.4 Symboles JSON

Écrivez un programme Haskell, **json.hs**, qui permet d'analyser lexicalement (*tokenizer*) un document JSON, dont le nom est passé en argument de ligne de commande. L'analyse lexicale consiste à convertir une chaîne de caractères en une liste de symboles (*tokens*), certains caractères pouvant être ignorés suivant leur pertinence. Le JSON (JavaScript Object Notation), lui, est un format léger d'échange de données qui est basé sur un sous-ensemble du langage de programmation JavaScript.

On se propose d'apporter un certain nombre de simplifications sur l'analyse lexicale d'un document JSON :

1. Les nombres considérés sont des entiers positifs constitués uniquement de digits.
2. Le support de caractères d'échappement, par exemple \", dans les chaînes peut être ignoré.

En dehors de ces simplifications, vous devez considérer tous les caractères qui constituent un document JSON. Pour ce faire, vous devez en premier lieu définir un type algébrique décrivant les différents symboles du format JSON de manière aussi précise que possible. Vous devez ensuite implémenter une fonction qui effectue l'analyse lexicale à proprement parler.

À titre d'exemple, voici les symboles d'un document JSON donné :

{ "str" : [123, true, null] } → '{', 'str', ' :', '[', '123', ',', 'true', ',', 'null', ']', '}'

2.5 Le loup, la chèvre et les choux

Écrivez un programme Haskell, **river.hs**, qui permet de jouer au fameux jeu de raisonnement mathématique *le loup, la chèvre et les choux*, connu également sous le nom du problème de passage de rivière qui est formulé comme suit :

"*Un fermier doit passer la rivière dans une barque juste assez grande pour lui et son loup, ou lui et sa chèvre, ou lui et ses choux. Les choux seront mangés s'il les laisse seuls avec la chèvre, et la chèvre sera mangée s'il la laisse seule avec le loup. Comment faire passer tout ce monde sans dégâts ?*". Wikipédia

Le but de votre programme est de fournir donc un environnement interactif permettant de disputer une partie du jeu. Pour ce faire, votre programme doit implémenter un interpréteur de commandes qui s'exécute à chaque tour du jeu, pour autant que le joueur souhaite poursuivre la partie, et qui supporte les commandes suivantes :

:p	afficher l'état du jeu
:l <passenger>	charger la barque avec un passager
:u	décharger la barque
:m	déplacer la barque
:r	réinitialiser le jeu
:q	quitter le jeu
:h	afficher l'aide

Pour cet exercice, il vous faudra définir un certain nombre de structures de données lesquelles vous permettront de décrire le jeu, l'état d'une partie, ses protagonistes ainsi que ses emplacements. Il vous faudra également modulariser votre implémentation autant que possible ; les commandes devant être supportées sont une bonne indication sur la manière de s'y prendre.

Il se peut que lorsque vous essayez d'afficher un message à l'écran celui-ci n'apparaisse pas immédiatement. Cela est dû au fait que la sortie standard de Haskell est tamponnée (*buffered*). Désactivez ce comportement avec *hSetBuffering* du module System.IO si besoin.

3 Évaluation

L'évaluation de votre implémentation pour chacun des exercices proposés dans ce devoir s'appuiera sur des critères bien précis, qui sont :

- Exactitude — Le code est correct au sens des exigences, du comportement et de l'exécution
- Complexité — Le code est compréhensible à la première lecture
- Performance — Le code est dépourvu de surcharges inutiles de performance
- Modularité — Le code est structuré de façon pertinente
- Style — Le code est écrit dans un style fonctionnel

Chaque critère aura le même poids sur le nombre de points obtenus pour un exercice donné. De manière similaire, chaque exercice vous rapportera au maximum un point sur le total de la note du devoir. Autrement dit, la note du devoir sera calculée selon la formule :

$$N = 1 + \sum_{x=1}^{5} E_x + C_x + P_x + M_x + S_x$$

où E_x, C_x, P_x, M_x *et* S_x valent chacun un cinquième de point.

4 Rendu

Le rendu du devoir comprend cinq fichiers sources Haskell, un fichier par exercice nommé tel qu'indiqué dans les énoncés. Chaque fichier doit inclure un entête, un commentaire Haskell, lequel indique le(s) auteur(s) du code. Le code rendu doit être le résultat de votre propre production. Le plagiat de code, de quelque façon que ce soit et quelle qu'en soit la source, sera considéré comme de la tricherie et puni en conséquence. La date de rendu pour ce devoir est fixée au **vendredi 29 avril 2022 à 16h30**. Aucun retard ne sera admis et la note de 1 vous sera automatiquement attribuée si cela devait se produire.

5 Conclusion

Ce devoir n'est pas difficile en soi dans la mesure où les algorithmes que vous devez concocter ne sont pas compliqués. De plus, un délai plus que suffisant vous est mis à disposition pour le réaliser. Toutefois, puisque ce cours est votre première expérience avec la programmation fonctionnelle, en Haskell qui plus est, il est important pour vous de commencer l'implémentation relativement tôt. À vous d'y investir le temps que vous jugerez nécessaire. Le forum de discussions est à votre entière disposition pour poser des questions et demander des conseils sur les différents exercices du devoir. Toutefois, ce forum ne doit pas devenir le lieu pour déboguer votre code. En cas de bug dans vos implémentations, il est de votre responsabilité d'identifier la source du problème et de la corriger. À cet égard, les outils de debugging sont vos meilleurs alliés.

Bon travail !