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