TP Haskell #1/7 (CS 222)

Grenoble INP – Esisar

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

Objectifs :

1. Prise en main de GHCi.
2. Définition de fonctions dans un fichier `.hs`.
3. Applications partielles et fonctions récursives.

Mise en place du TP

Durant les TP, nous allons principalement utiliser **GHCi**, un programme interactif Haskell. GHCi va nous permettre de faire des calculs, tester des fonctions, et charger des fichiers de code Haskell.

```
michelse@cs222:~$ ghci
GHCi, version 9.2.5: https://www.haskell.org/ghc/  :? for help
ghci> reverse "Haskell!"
"!lleksaH"
ghci> :quit
Leaving GHCi.
michelse@cs222:~$
```

Sur les machines de l'Esisar, l'interpréteur `ghci` est déjà installé dans l'environnement **Linux**. Sur vos machines personnelles, vous pouvez l'installer à l'aide de GHCup.

- Créez un répertoire de travail `CS222/TP1` et naviguez-y dans un terminal.

Exercice 1 — Expressions et types dans l'évaluateur

Lancez la commande `ghci` dans votre terminal ; tapez-y les expressions suivantes.

```haskell
-- Opérations numériques
5 * 4 ^ 2
9 / 2
(+) 4 6
-- Fonctions numériques
div 9 2
9 `div` 2               -- notation infixe pour la fonction div
succ 7
odd 15
even 17
pi
-- Comparaisons et booléens
3 == 3
3 /= 6
True && False
not (True || True)
-- Listes et chaînes de caractères
```

```haskell
reverse [1..5]
['J'..'e']
reverse "Hello World"    -- ça marche comme une liste !
```

Même question pour les définitions et expressions suivantes. Vérifiez que le résultat obtenu est celui que vous attendiez.

```haskell
let x = 67               -- dans ghci, on peut aussi écrire x = 67
f = (+3)                 -- que fait f ?
let y = x * 2
f y
```

Rappel : la fonction `map :: (a -> b) -> [a] -> [b]` prend en paramètres une fonction `f :: a -> b` et une liste de valeurs de type `a` ; elle applique `f` à tous les éléments de la liste et renvoie la liste résultante de valeurs de type `b`.

```haskell
plusUn = map (+1)        -- que fait plusUn ?
plusUn [1 .. 5]          -- l'argument est la liste [1,2,3,4,5]
bs = ['a','b','c','d']
zip [1..9] bs

head [1 .. 10]
tail [1 .. 10]
```

Enfin, observez les types des expressions suivantes. Pour ça, on utilise la commande `:type`. Notez que toutes les commandes commençant par `:` s'adressent à GHCi et n'ont pas de sans dans un fichier source Haskell.

```haskell
:type True
:type "TP Haskell"
:type 'x'
:type map                -- voir ci-dessous
:type 42                 -- voir ci-dessous
:type (+)                -- voir ci-dessous
```

Pour `map`, le type renvoyé `(a -> b) -> [a] -> [b]` parle de deux types `a` et `b` qui ne sont pas spécifiés. En fait, la fonction peut être utilisée pour *n'importe quels types `a` et `b`*.

Pour `42` et `(+)`, la situation est similaire, mais cette fois le type est préfixé de `Num a =>`, une "contrainte" indiquant que l'expression peut être utilisée uniquement avec des types numériques. Ce mécanisme nous permet d'utiliser `+` comme une fonction d'addition pour plusieurs types (par exemple `Int -> Int -> Int`, `Integer -> Integer -> Integer`, ...).

Expliquez pourquoi les expressions suivantes renvoient une erreur de type :

```haskell
True && 1
[1, 2, True]
1 + "7"
map (*2) [True,False]
if x==y then [x,'a'] else [y,4.0]
```


Exercice 2 — Définitions : de l'interprète au fichier (`definitions.hs`)

Fichier à rendre : `definitions.hs`.

- Dans l'interpréteur, définissez une fonction `double` qui permet de doubler la valeur d'un argument numérique. Testez-la avec les arguments : 0, 5 et 42.

Pour conserver le travail, on va tout de suite prendre l'habitude d'écrire notre code Haskell dans des fichiers `.hs`. On pourra ensuite les utiliser dans GHCi de la façon suivante :

- `:load <FICHIER>` (ou `:l <FICHIER>`) charge un fichier `.hs` dans GHCi.
- `:reload` (ou `:r`) recharge le dernier fichier chargé.
- On notera aussi que `:{` et `:}` peuvent être utilisés pour écrire des définitions sur plusieurs lignes dans GHCi.

À partir de maintenant, chaque exercice sera à rendre dans un fichier dont le nom est dans le titre de la section (par exemple, celui-ci est à rendre dans `definitions.hs`).

- En utilisant un éditeur de texte, créez le fichier `definitions.hs` avec la définition de la fonction `double`. Testez la fonction `double` dans GHCi en chargeant le fichier avec `:l definitions.hs`.

Attention : GHCi est un peu plus qu'un interprète, il évalue par exemple les expressions nues ("1+2"). Une requête à GHCi n'est pas nécessairement un code source valide dans un fichier `.hs`.

- Ajoutez au fichier la définition d'une fonction `quadruple` qui permet de quadrupler un argument numérique. Définissez-la en utilisant la fonction `double`. Testez la fonction `quadruple` avec les mêmes arguments que ci-dessus en rechargent le fichier avec `:r`.

Exercice 3 — Fonctions en paramètres, applications partielles (`applis.hs`)

Fichier à rendre : `applis.hs`.

Une application de fonction est dite *partielle* si on ne fournit pas tous les arguments de la fonction. Une application partielle est une version spécialisée de la fonction, où les arguments fournis ont été fixés. Par exemple, la fonction `min 2` prend un argument, et elle associe à l'entier `y` l'entier `min 2 y`.

- L'expression `(>=)` est une fonction ; essayez-la sur des entiers. Combien a-t-elle d'arguments ? Définissez la fonction `z = (>=) 3`. Que calcule-t-elle ?
- Pouvez-vous prédire (ou expliquer) la valeur de l'expression `map z [1..5]` ?

Définissez dans GHCi la fonction `app f = map f [1..5]`.

- Quel sont les types de `app` et `app z` ?
- Montrez comment on peut utilisez `app` pour obtenir le même résultat que la deuxième question.
- Construisez un appel à `app` qui calcule le successeur de tous les éléments de la liste. De même, construisez un appel à `app` qui calcule le maximum entre 4 et chaque élément, d'abord avec une fonction auxiliaire, puis avec une application partielle de la fonction puissance `max :: Int -> Int -> Int`.

On veut maintenir définir la fonction `mod2` par application partielle de la fonction modulo, `mod`. Malheureusement, l'argument qu'on veut fixer n'est pas le premier.

- Écrire une fonction `echange :: (a -> b -> c) -> (b -> a -> c)` qui prend en argument une fonction à deux paramètres, et renvoie une copie de cette fonction où l'ordre des paramètres a été inversé.

Indice : Le type `(a -> b -> c) -> (b -> a -> c)` s'écrit aussi `(a -> b -> c) -> b -> a -> c`. Autrement dit, au lieu de renvoyer une fonction `b -> a -> c`, vous pouvez vous-même prendre deux

arguments supplémentaires de type `b` et `a`, et calculer le `c`. En vertu des applications partielles, ça revient au même.

- Utiliser `echange` pour définir la fonction `mod2` comme une application partielle de `mod`.

Interlude — Utiliser un "script GHCi" pour tester votre code

Comme vous l'avez vu, écrire des tests dans l'évaluateur est fastidieux et peu reproductible. Sans viser à produire un système de test de qualité industrielle, nous pouvons faire mieux en mobilisant la commande `:script` de GHCi pour exécuter automatiquement des commandes pré-écrites dans un fichier.

- Dans un fichier nommé `testapplis.ghci`, copiez le texte suivant. Remplacez les points de suspension dans chaque commentaire par la réponse attendue de GHCi à la commande qui précède.

```
app f = map f [1, 2, 3, 4, 5]
-- ...
mod2 27
-- ...
```

Ce code n'est pas un programme Haskell valide, comme vous pouvez le vérifier en tapant `:load testapplis.ghci`. Vous pouvez cependant activer la coloration syntaxique Haskell dans votre éditeur favori.

- Exécutez ce script dans GHCi à l'aide de la commande `:script testapplis.ghci` et vérifiez que les réponses correspondent à vos commentaires.

Vous fournirez à partir de maintenant, pour chaque exercice, un script GHCi montrant les tests que vous avez faits. Vous prendrez soin de noter après chaque calcul la réponse attendue en commentaire.

Exercice 4 — Fonctions récursives (`rec.hs`)

Fichiers à rendre : `rec.hs`, `testrec.ghci`.

- Dans le fichier `rec.hs`, créez la fonction `puissance` avec la définition récursive suivante :

  ```
  puissance a 0 = 1
  puissance a n = a * puissance a (n-1)
  ```

  Fournissez des tests pertinents dans `testrec.hs`.

- Sur le même modèle, définissez de manière récursive les fonctions suivantes :
  - Factorielle : `fact :: Integer -> Integer`
  - Somme des carrés des n premiers entiers : `somme_carres :: Integer -> Integer`
  - Suite de Fibonacci : `Integer -> Integer`

  Ajoutez des tests au fur et à mesure.

- Vérifiez que vos définitions ont bien les bons types dans GHCi, avec `:type` ou `:browse`.

On veut maintenant déterminer si la fonction `puissance` est efficace et si on ne peut pas en écrire une version plus rapide.

- Dans un commentaire, exécuter à la main un exemple d'appel à la fonction `puissance`. Combien y a-t-il d'appels récursifs (en fonction de l'exposant) ?

On observe qu'on peut calculer `a^n` en multipliant pas seulement par `a` mais aussi par des puissances de `a`. En particulier, on remarque que

$$\begin{cases} a^n = (a^{n/2})^2 & \text{si } n \text{ est pair} \\ a^n = (a^{n/2})^2 \times a & \text{si } n \text{ est impair.} \end{cases}$$

- Écrire une nouvelle version de `puissance` qui implémente cette méthode de calcul (qu'on appelle *exponentiation rapide*). Dérouler les appels récursifs pour le calcul de `2^35`.
- (Bonus) Combien y a-t-il d'appels récursifs (en fonction de l'exposant) ?

Pour aller plus loin

Si vous arrivez jusqu'ici, bravo ! Vous pouvez approfondir votre compréhension de ce sujets en lisant l'introduction au langage de *Monday Morning Haskell*.

- https://mmhaskell.com/liftoff/
- Notez que `:t` est une abbréviation de la commande GHCi `:type` et `::` peut être utilisé pour spécifier le type d'une expression.