TME1 - Prise en main de GHC (Glasgow Haskell Compiler)

Sorbonne Université - Master Informatique M1 - STL

MU4IN510 Programmation Avancée en Fonctionnel - 2021

Dans ce premier TME, notre objectif est de démarrer notre apprentissage du langage Haskell qui sert de support au cours PAF.

En terme d'outillage, nous utiliserons les outils suivants :

- le gestionnaire de projets **stack** (cf. instructions d'installation sur le forum discord)
- le compilateur **GHC** (*Glasgow Haskell Compiler*) dans sa version 8.10.3 (installé par **stack**)
- un éditeur de texte avec support minimal (au moins coloration syntaxique et indentation) pour le langage Haskell, par exemple vscode ou emacs (pour vscode, cf. les instructions d'installation sur le forum).

On se placera dans le cadre du projet `paf-playground` créé après l'installation de **stack** (toujours cf. les instruction sur discord). Nous travaillerons également avec un shell dans ce répertoire : un shell unix sous Linux ou Macos et un powershell sous windows.

Pour vérifier la présence du compilateur et de son numéro de version, on pourra par exemple taper :

```
$ stack ghc -- --version
The Glorious Glasgow Haskell Compilation System, version 8.10.3
```

Lancement du REPL

En terme d'outillage, un avantage indéniable des langages fonctionnels est de fournir un interprète de commandes ou REPL (*Read, Eval, Print Loop*) qui permet une programmation très dynamique. L'implémentation GHC de Haskell ne déroge pas à la règle, son interprète de commandes est très puissant et nous tenterons dans ce cours d'y faire appel le plus souvent possible.

Dans une fenêtre de terminal, lançons `ghci` de la façon suivante :

```
$ stack ghci
... blablabla ...
*Main Lib Paths_paf_playground>
```

On va raccourcir l'invite (*prompt*) pour rendre les interations un peu plus lisibles, en tapant la commande :

```
:set prompt "> "
```

Après ce *prompt* `>` on peut sasir des commandes, en particulier des expressions Haskell et des commandes "spéciales" préfixées par deux-points. La commande `:help` liste les très nombreuses possibilités.

Essayons une petite opération arithmétique :

```
> 3 + 4 * 12 - 5
46
```

Nous pouvons également demander le type d'une expression plutôt que sa valeur. Et pour cela on utilise la commande `:type` ou `:t` :

```
> :type 3 + 4 * 12 - 5
3 + 4 * 12 - 5 :: Num a => a
```

Le résultat est peut-être un peu différent de ce à quoi on pourrait s'attendre. Ici plutôt qu'un type entier par exemple, on a un type générique `Num a => a` qui signifie :

Tout type `a` implémentant les opérations arithmétiques usuelles

Vu autrement, même les litéraux “entiers” sont compatibles avec des types numériques définis par l'utilisateur.

On peut bien sûr forcer le type des litéraux :

```
> :t 3 + 4 * 12 - 5 :: Integer
3 + 4 * 12 - 5 :: Integer :: Integer
```

Editeur

Pour éditer des programmes Haskell, on utilise le plus courant un éditeur “classique” comme Vim ou Emacs, ou alors un éditeur plus récent comme Atom ou VSCode. Le minimum vital nécessaire pour éditer du code Haskell est de disposer d'un support pour la coloration syntaxique, et surtout un support l'aide à l'indentation. En effet, Haskell est un langage indenté avec des règles un peu plus complexes que celles de Python. Sur le discord on explique l'installation de VSCode et pour le lancer, il suffit de taper : `code .` (dans le répertoire `paf-playground`).

Une première fonction : le maximum entre deux entiers

Nous allons maintenant travailler dans le fichier `src/Lib.hs` du projet `paf-playground`.

Dans ce fichier, définir une fonction `maxInt` prenant deux entiers (`Integer`) et retournant le maximum de ces deux entiers. On utilisera une expression `if ... then ... else` pour implémenter la fonction.

Il faudra ensuite exporter la fonction en changeant la déclaration du module `Lib`, on écrira ceci au début du fichier :

```
module Lib
    ( someFunc
    , maxInt
    ) where
```

Dans ghci (que l'on garde ouvert), on rechargera les modules avec la commande `:reload` et on testera la fonction, de telle sorte que :

```
> :reload
[2 of 3] Compiling Lib              ( ... )
Ok, three modules loaded.
> maxInt 2 4
4
> maxInt 4 2
4
> maxInt 4 4
4
```

On essaierai également, pour celles et ceux qui utilisent VSCode, de placer ces exemples directement dans le fichier avec des commentaires spéciaux `-- >>>` et le bouton *Evaluate* (cf. les explications sur le discord).

Une deuxième fonction : fibonacci

Définir une fonction `fibo` retournant le n-ième nombre de *Fibonacci*.

```
*Main> fibo 0
1
*Main> fibo 1
1
*Main> fibo 2
2
*Main> fibo 3
3
*Main> fibo 4
5
```

```
*Main> fibo 5
8
```

etc...

Si cela vous amuse, vous pouvez faire une version récursive terminale, mais ce n'est pas si important...

Un premier jeu (il y en aura d'autres !)

Pour ce dernier exercice de TME, nous souhaitons compléter l'implémentation d'un jeu extrêmement simple à deux joueurs: «Devine le nombre». Pour cela on va créer un nouveau module/fichier `app/Main.hs` fourni (enfin on fourni juste le fichier qu'il faudra placer dans le repértoire `app/` en écrasant l'ancien) et il faudra remplacer toutes les occurrences de `undefined` par des expressions adéquates.

Pour comprendre ce que fait le programme, il faudra bien lire les commentaires bien sûr...

Pour compiler le programme, on utilisera simplement la commande suivante :

```
$ stack build
```

et pour «jouer» :

```
$ stack run
```

Remarque : le programme est déjà compilable et exécutable, donc on peut par exemple interagir dans ghci ou avec le bouton `Evaluate` ... L'évaluation d'un `undefined` produit juste une erreur d'exécution.

Défi 1 : Choix dans un intervalle

Ajouter à votre jeu une première demande d'intervalle pour le choix du nombre à deviner. Par exemple, si on a saisit l'intervalle `1 - 1000` il faudra d'abord choisir un nombre dans cet intervalle, puis le faire deviner. On modifiera l'algorithme de triche pour exploiter au mieux cet intervalle explicite.

Défi 2 : Choix aléatoire (dans un intervalle)

On ajoutera le choix aléatoire en demandant initialement une graine pour le générateur aléatoire (que l'on choisira donc purement fonctionnel). Et bien sûr il faudra implémenter un algorithme simple de génération aléatoire uniforme. (pour l'instant on ne peut pas utiliser la bibliothèque tierce `random` dédiée à cette problématique en Haskell).

Important : les défis sont facultatifs, c'est juste pour nous impressionner...