Projet : une simple environnement interactif

Version du 6 novembre 2017

Nous allons écrire un simple programme, réminiscent de `gchi`, permettant d'évaluer des expressions numériques.

1 Première partie

Dans cette première partie, nous allons donner une implantation minimale de ce programme. Le rendu du projet contiendra un sous-répertoire `partieI` avec cette première partie—et seulement celle-ci—complètement développée.

Le but de cette première partie, assez élémentaire, est de bien structurer le travail. En particulier, on prêtera attention à organiser le code, de façon qu'il puisse s'étendre aisément. Dans la deuxième partie du projet, les expressions seront plus complexes, construites aussi grâce à des nouveaux opérateurs ; aussi nous ajouterons des nouvelles commandes. L'ajout d'un ou de plusieurs opérateurs/commandes pourra se faire aisément à partir du code existant.

Résoudre avec précision et clarté cette partie vous donnera droit à 13 des 20 points de la note finale.
En plus du module `Main` définissant l'action d'entrée `main`, votre code se composera de au moins 6 modules :

`Exrpession.hs`. On trouvera dans ce module la définition/description des types `Expression`, `Store` et `Variable`, et de la fonction `eval`.
Un objet de type `Expression` est construit à partir des variables et constantes numériques par les opérateurs de négation (opérateur unaire), somme, multiplication, et exponentiation (opérateurs binaires).
Un objet de type `Store` est une fonction (partielle) qui associe à une variable un nombre. On pourra implementer le type `Store` par des listes associatives (listes de couples clé-valeur, voir https://en.wikipedia.org/wiki/Association_list).
La fonction principale du module est la fonction

```
eval :: Store -> Expression -> Maybe Float
```

qui évalue une expression à un nombre, par rapport à un « store ». En effet, pour évaluer récursivement une expression il est nécessaire d'évaluer les variables contenues dans l'expression. L'évaluation d'une variable se fait par le store : la valeur de la variable est le nombre correspondant à la variable.
On utilise `Maybe` pour les valeurs de retour, car on ne réussit pas toujours à évaluer une expression : si une expression ne peut pas s'évaluer à cause d'une variable qui n'est pas dans le store, alors on retournera `Nothing`.

`Parse.hs`. Le but de ce module est d'écrire une fonction

```
parseExpression :: String -> Maybe Expression
```

qui fait l'analyse syntaxique d'une chaîne de caractères et construit l'expression correspondante.
On utilise `Maybe` car il n'est pas possible de parser toutes les chaînes de caractères : il est possible parser la chaîne $x + 33 * 5$, mais non la chaîne $+33$ ^.
Il est fortement conseillé d'utiliser la bibliothèque `Parsec` de Haskell, voir https://wiki.haskell.org/Parsec. Depuis cette page, on trouvera beaucoup d'exemples sur l'utilisation de cette bibliothèque ; servez vous aussi de l'exemple porté en cours.
Tous les détails autour de la syntaxe utilisateur des expressions seront concentrés dans ce module.

`EmvInteractif.hs`. Dans ce code nous allons implémenter l'environnement interactif. Notez bien que toute action `IO a` apparaîtra dans ce module.

La boucle principale de l'environnement peut se décrire de cette façon :

1. l'utilisateur rentre une ligne,
2. cette ligne est analysée pour voir si c'est une commande ou bien une expression,
3. si c'est un commande on l'exécute,
4. si c'est une expression on l'évalue ;
5. on recommence.

On pourra distinguer les commandes des expressions avec une condition lexicale (par exemple, toute commande est introduit par les doubles points verticaux, comme en `ghci`). On implantera les commandes suivants :

— `q` ou `quit` : on sort du programme.
— `h` ou `help` : on affiche la liste des commandes existants avec des explications ;
— `store` : on affiche le contenu du store ;
— `set x a`, où a est un nombre : on ajoute au store la variable x, avec valeur a ;
— `unset x` : on enlève x du store courant.

On prêtera encore une fois attention à la bonne organisation du code : ici, l'ajout d'un commande pourra se faire de façon modulaire, en modifiant un morceau minimal du code.
Par exemple, vous pouvez définir un type des commandes comme suit :

```
type Handler = [String] -> Store -> IO Store
data Command = Command {
  name :: String, -- Nom de la commande
  description :: String, -- Description de la commande
    -- utilise' par la commande -help
  exits :: Bool, -- Drapeau pour sortir de la boucle,
    -- vrai pour quit, faux pour les autres
  run :: Handler -- Le code a` executer
  }
```

Observez que le style de programmation se rapproche du OO : on peut considérer la déclaration `data` ci-dessus comme la définition d'une classe (au sens OO), `name`, `description` et `exits` étant les propriétés de la classe, et `run` comme la méthode de la classe (au sens OO). Remarquez donc la différence entre la notion de classe au sens OO et celle de classe au sens Haskell.

`ExpressionTest.hs`, `ParserTest.hs`, `EnvInteractifTest.hs`. Chaque module `X` sera accompagné par un module `XTest`. On aura donc au moins des modules `ExrpessionTest.hs`, `ParserTest.hs`, `EmvInteractifTes`
Ce modules contiendront de test unitaires. Vous pouvez utiliser la bibliothèque `HUnit`, voir https://hackage.haskell.org/package/HUnit, pour écrire et exécuter automatiquement ces tests.
En particulier, on ajoutera des tests visants à assurer que les motifs des fonctions définies utilisant le filtrage soient exhaustifs. Cela permettra d'étendre le code en assurant que de erreurs dus aux filtrages non exhaustifs se produisent lors de l'exécution.
C'est laissez à votre discrétion de décider quel tests ajouter dans chaque module `XTest` (ce n'est pas un cours de génie logiciel). Mais si vous êtes curieux, vous pouvez lire la page
https://fr.wikibooks.org/wiki/Introduction_au_test_logiciel/Qualit%C3%A9_des_tests
pour vous faire une idée de ce que c'est un bon jeu de tests.

D'autres fichiers à inclure (dans la partie I et dans la partie II)

Fichier `README.txt`. Ce fichier contiendra :

— noms et prénoms des membres du binôme ;
— description des objectifs du projet ;

— description de ce qui a été accompli par rapport au travail demandé ;
— les sources de et/ou un lien vers n'importe quel code que vous a partiellement (ou totalement) inspiré pour l'écriture de votre projet ;
— vos observations éventuelles.

Fichier `Makefile`. Ce fichier permettra au chargé de TP de compiler votre projet en tapant `make` sur la console. Votre code ne sera pas compilé à l'avance, seulement les sources et les autres fichiers texte demandés seront transmis.

2 Deuxième partie

On n'abordera pas cette deuxième partie tant que la première partie n'est pas complètement élaborée. Le rendu du projet contiendra aussi un sous-répertoire `partieII` avec le code qui sera une extension de la première partie.
On vous demande détendre le code de la première partie par :

1. l'ajout d'autres opérateurs arithmétiques, tels que la division (binaire) et d'un minimum de fonctions trigonométriques (par exemple `sin` et `arctg`). Ces fonctions trigonométriques seront pensées comme des opérateurs unaires préfixes.
2. l'ajout de la possibilité de calculer des intégrales des expressions sur un intervalle donnée et avec une certaine précision. Par exemple, une intégral de a à b avec précision n peut se définir par la formule

$$\int_{n,a}^{b} expr(x)dx = \sum_{i=1,\ldots n} expr(a_i)(a_i - a_{i-1}) \qquad \text{avec } a_i = a + i * \frac{(b-a)}{n}.$$

   On considérera une telle intégrale comme un opérateurs avec cinq opérandes : la variable x, l'entier n, les nombres a et b, et la sous-expression *expr* contenant la variable x.
3. On modifiera le store afin qu'il puisse contenir des expressions non évaluées. L'évaluation d'une expression se fera alors en détectant des cycles dans le store. Par exemple, si le store contient $x := y + 5$ et $y := x + 4$, alors l'évaluation de l'expression $x + 2$ détectera l'erreur et retournera `Nothing`.
4. On améliorera le traitement des erreurs en permettant de propager des messages d'erreur. On pourra faire ça en substituant `Maybe` par `Either`—dans le fonctions `eval`, `parseExpr`, et dans toute autre fonction qui retourne un `Nothing` en cas d'erreur.
5. On ajoutera, parmi les opérateurs sur Expressions, l'opérateur *subst* et *it*, qui se définissent par

$$subst\ x\ expr1\ expr2 := expr1[\ expr2\ /\ x\ ]$$

   où $expr1[\ expr2\ /\ x\ ]$ est le résultat de remplacer toute occurrence de x dans $expr1$ par $expr2$,

$$\begin{aligned} it\ 0\ x\ expr &:= x \\ it\ (n+1)\ x\ expr &:= subst\ x\ expr\ (it\ n\ x\ expr)\,. \end{aligned}$$

   Ainsi :
   — *subst* est un opérateur avec trois opérandes : une variable et deux expressions ;
   — *it* est un opérateur avec trois opérandes : une variable, un entier, et une expression.
6. On modifiera le comportement de la fonction d'évaluation dans le cas d'une expression qui contient des variables qui n'apparaissent pat dans le store. Dans ce cas, on retournera des expressions normalisées, dans le sens que toute opérateur *subst* et *it* sera éliminé des l'expression en utilisant les règles ci-dessus (par exemple, on éliminera d'abord tous les opérateurs *it* et ensuite tous les opérateurs *subst*.
7. On pourra ajouter des nouvelles commandes, par exemple :

  - une commande `unsetAll` qui vide le store ;
  - la commande `load` qui exécute, ligne par ligne un fichier ; chaque ligne du fichier contient une commande de type `set` (ou autres) ;
  - à vous d'imaginer d'autres commandes utiles.

3 Remarques et caveats

- Le projet se déroule en binômes. Si vous n'arrivez pas à trouver un binôme, veuillez le signaler, un binôme pourra alors vous être attribué.
- Date limite du rendu du projet : le mercredi 06/12/2017, à 23h59 (heure de Paris). Envoyez votre projet **en format compressé `zip` ou `tgz` à votre encadrant de TP**. Le nom du fichier envoyé **indiquera clairement** les deux noms (pas les prénoms) des membres du binômes.
- Afin de prévenir plagiat et copies, la similarité entre les sources du projet sont vérifiées avec des outils automatiques efficaces et fiables. Donc, afin d'éviter de recevoir la note minimum, evitez de :
  - travailler en collaboration avec d'autres binômes ;
  - transmettre votre code à d'autres étudiants ;
  - afficher votre code sur des répositoires ouverts (tels que `github`) tant que l'évaluation de votre projet n'est pas terminée.
- La note de projet est individuelle, et peut ne pas être égale pour les deux membres du binôme.
- La soutenance est une partie nécessaire de l'évaluation de votre projet ; la présence de chaque individu à la soutenance est donc obligatoire. L'absence à la soutenance implique une note de 0 à votre projet.
- Si vous utilisez, dans votre projet, du code dont vous n'êtes pas complètement les auteurs, veuillez le signaler clairement dans le fichier appelé `README.txt` et inclure les sources qui vous ont inspirés dans le projet.

  Par ailleurs :
  - On vous demande de suivre strictement les étapes décrites dans le projet.
  - On testera, lors de la soutenance et pour chaque étape, la compréhension du code dont vous êtes supposés être les auteurs. Une incompréhension manifeste (ou miminale) du code présenté à la soutenance entraînera la note 0 (ou minimale) à votre projet.
- Dates des soutenances :
  - à Luminy : le mardi 12/12/2017 (date prévisionnelle).
  - à St-Jérôme : le mercredi 13/12/2017 (date prévisionnelle).