Programmation Fonctionnelle
Premiers pas

Luigi Santocanale
LIF, Aix-Marseille Université
Marseille, FRANCE

12 septembre 2016

Plan

Plan

The Glasgow Haskell Compiler (GHC)

- GHC est une implémentation de Haskell 98 se composant de :
  1. `ghci`, un interprète,
  2. `runghc`, qui (interprete et) execute un script Haskell,
  3. `ghc`, le compilateur Haskell,
  4. plein d'autres outils.

- . . . GHC est disponible sur le web de :

  `http://www.haskell.org/ghc/`

  ou mieux depuis :

  `http://hackage.haskell.org/platform/`

Utilisation de GHC, I

Le script `hello_world.hs` :

```
str = "Bonjour le monde"
main = print str
```

avec `ghci` :

```
santocan@ens1:~/Haskell$ ghci Hello_world.hs
GHCi, version 6.12.1: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
[1 of 1] Compiling Main             ( Hello_world.hs, interpreted )
Ok, modules loaded: Main.
*Main> main
"Bonjour le monde"
*Main> :q
Leaving GHCi.
santocan@ens1:~/Haskell$
```

Utilisation de GHC, I

Le script `hello_world.hs` :

```
str = "Bonjour le monde"
main = print str
```

avec `ghci` :

```
santocan@ens1:~/Haskell$ ghci Hello_world.hs
GHCi, version 6.12.1: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
[1 of 1] Compiling Main             ( Hello_world.hs, interpreted )
Ok, modules loaded: Main.
*Main> main
"Bonjour le monde"
*Main> :q
Leaving GHCi.
santocan@ens1:~/Haskell$
```

Utilisation de GHC, II

avec `runghc` (aussi nommé `runhaskell`) :

```
santocan@ens1:~/Haskell$ runghc Hello_world.hs
"Bonjour le monde"
santocan@ens1:~/Haskell$
```

avec `ghc` :

```
santocan@ens1:~/Haskell$ ghc Hello_world.hs
santocan@ens1:~/Haskell$ ls
a.out  Hello_world.hi  Hello_world.hs  Hello_world.hs~  Hello_world.o
santocan@ens1:~/Haskell$ ./a.out
"Bonjour le monde"
santocan@ens1:~/Haskell$
```

Utilisation de GHC, II

avec `runghc` (aussi nommé `runhaskell`) :

```
santocan@ens1:~/Haskell$ runghc Hello_world.hs
"Bonjour le monde"
santocan@ens1:~/Haskell$
```

avec `ghc` :

```
santocan@ens1:~/Haskell$ ghc Hello_world.hs
santocan@ens1:~/Haskell$ ls
a.out  Hello_world.hi  Hello_world.hs  Hello_world.hs~  Hello_world.o
santocan@ens1:~/Haskell$ ./a.out
"Bonjour le monde"
santocan@ens1:~/Haskell$
```

Le prompt > signifie que `ghci` est prêt à évaluer une expression. Par exemple :

```
> 2+3*4
14
> (2+3)*4
20
> sqrt (3^2 + 4^2)
5.0
```

La platforme Haskell

La plateforme Haskell

`http://hackage.haskell.org/platform/`

en plus de `ghc`, `ghci`, `runhaskell`, contient aussi d'autres outils intéressants :

- `cabal`, gestionnaire de bibliothèques
- `haddock`, gestionnaire de la documentation
- `alex` et `happy`, analise lexicale et syntaxique (i.e. `lex` et `yacc` pour Haskell)

Plan

Le `Prelude.hs` I

- Le module `Prelude.hs` est chargé en mémoire lors du démarrage de l'interprète.
- Ce module est la biliothèque standard du langage Haskell.
- Il contient la définition d'un grand nombre de fonctions usuelles (par ex. `+` et `*`).
- Pour voir son contenu, cliquez ici.

Des fonctions sur les listes I

En plus des fonctions numériques, ce module contient plusieurs fonctions sur les *listes*.

- Sélectionner le premier élément d'une liste :

```
> head [1,2,3,4,5]
1
```

- Enlever le premier élément d'une liste :

```
> tail [1,2,3,4,5]
[2,3,4,5]
```

- Sélectionner le *n*-ième élément d'une liste :

```
> [1,2,3,4,5] !! 2
3
```

Des fonctions sur les listes II

- Sélectionner les premiers *n* éléments d'une liste :

```
> take 3 [1,2,3,4,5]
[1,2,3]
```

- Enlever les premiers *n* éléments d'une liste :

```
> drop 3 [1,2,3,4,5]
[4,5]
```

- Calculer la longueur d'une liste :

```
> length [1,2,3,4,5]
5
```

- Calculer la somme d'une liste de nombres :

```
> sum [1,2,3,4,5]
15
```

Des fonctions sur les listes III

- Calculer le produit d'une liste de nombres :

```
> product [1,2,3,4,5]
120
```

- Concaténer deux listes :

```
> [1,2,3] ++ [4,5]
[1,2,3,4,5]
```

- Renverser une liste :

```
> reverse [1,2,3,4,5]
[5,4,3,2,1]
```

Plan

Application (d'une fonction), I

En mathématiques :
l'application d'une fonction est notée en utilisant les parenthèses,
la multiplication par la juxtaposition ou l'espace.

$$f(a,b) + c\,d$$

Appliquer la fonction f à a et b,
puis ajouter le résultat au produit de c et d.

En Haskell :
l'application d'une fonction est notée par l'espace,
la multiplication par $*$.

```
f a b + c*d
```

Comme auparavant,
mais avec la syntaxe de Haskell.

Application (d'une fonction), I

En mathématiques :
l'application d'une fonction est notée en utilisant les parenthèses,
la multiplication par la juxtaposition ou l'espace.

$$f(a,b) + c\,d$$

> *Appliquer la fonction f à a et b,*
> *puis ajouter le résultat au produit de c et d.*

En Haskell :
l'application d'une fonction est notée par l'espace,
la multiplication par `*`.

```
f a b + c*d
```

> *Comme auparavant,*
> *mais avec la syntaxe de Haskell.*

Application, II

L'application possède priorité plus élevée que les autres operateurs.

```
f a * b
```

signifie $(fa) * b$, au lieu que $f(a * b)$.

L'application est un opérateur associatif à gauche :

```
f g x
```

signifie :

> *la fonction f qui s'applique à l'argument g, donne la fonction $f(g)$ qui s'applique à x.*
>
> *En mathématiques on écrit cela par $(f(g))(x)$ – et non pas $f(g(x))$.*

Application, II

L'application possède priorité plus élevée que les autres operateurs.

```
f a * b
```

signifie $(fa) * b$, au lieu que $f(a * b)$.

L'application est un opérateur associatif à gauche :

```
f g x
```

signifie :

> *la fonction f qui s'applique à l'argument g, donne la fonction f(g) qui s'applique à x.*
>
> *En mathématiques on écrit cela par $(f(g))(x)$ – et non pas $f(g(x))$.*

Exemples (exercices)

Mathématiques	Haskell
f(x)	f x
f(x,y)	f x y
f(g(x))	f (g x)
f(x,g(y))	f x (g y)
f(x)g(y)	f x * g y

Plan

Les scripts en Haskell

- Vous y pouvez définir vos fonctions ;
- En général, vous y définissez des expressions ;
- Les nouvelles fonctions sont d'habitude définies dans un script,
  un fichier texte contenant une séquence de définitions ;
- Les scriptes Haskell portent (par convention) le suffixe `.hs`.
  Pas obligatoire, mais utile pour les identifier.

Un premier script (en salle TP)

Quand on développe un script, on garde ouvertes deux fenêtres :

- une pour l'éditeur de texte,
- l'autre avec l'interprète.

Exo :
Dans l'éditeur de texte, tapez les deux définitions de fonctions suivantes, et sauvegardez les avec le nom `test.hs` :

```
double x   = x + x
quadruple x = double (double x)
```

Un premier script (en salle TP)

Quand on développe un script, on garde ouvertes deux fenêtres :

- une pour l'éditeur de texte,
- l'autre avec l'interprète.

Exo :
Dans l'éditeur de texte, tapez les deux définitions de fonctions suivantes, et sauvegardez les avec le nom `test.hs` :

```
double x   = x + x
quadruple x = double (double x)
```

Laissez l'éditeur ouvert, dans une autre fenêtre démarrez `ghci` avec le nom du script en paramètre :

```
$$ ghci test.hs
```

Maintenant, `prelude.hs` et `test.hs` sont chargés, et les fonctions des tous les deux script peuvent être utilisés :

```
Main> quadruple 10
40
Main> take (double 2) [1,2,3,4,5,6]
[1,2,3,4]
```

Laissez l'éditeur ouvert, dans une autre fenêtre démarrez `ghci` avec le nom du script en paramètre :

```
$$ ghci test.hs
```

Maintenant, `prelude.hs` et `test.hs` sont chargés, et les fonctions des tous les deux script peuvent être utilisés :

```
Main> quadruple 10
40
Main> take (double 2) [1,2,3,4,5,6]
[1,2,3,4]
```

Laissez `ghci` ouvert, revenez à l'éditeur, ajoutez les définitions suivantes, sauvegardez à nouveau :

```
factorial n = product [1..n]
average ns = sum ns `div` length ns
```

Remarque :

- `div` est inclus entre guillemets simples inversés en arrière (apostrophes inversées) ;
- `x `div` y` est équivalent à `div x y`.
- Règle générale : `x `f` y` est équivalent à `f x y`.

Laissez `ghci` ouvert, revenez à l'éditeur, ajoutez les définitions suivantes, sauvegardez à nouveau :

```
factorial n = product [1..n]
average ns = sum ns `div` length ns
```

Remarque :

- `div` est inclus entre guillemets simples inversés en arrière (apostrophes inversées) ;
- `x `div` y` est équivalent à `div x y`.
- Règle générale : `x `f` y` est équivalent à `f x y`.

`ghci` ne reconnaît pas que le script a changé,
il faut exécuter la commande `:reload` avant pouvoir utiliser les nouvelles définitions.

```
Main> :reload
Reading file "test.hs"
Main> factorial 10
3628800
Main> average [1,2,3,4,5]
3
```

Plan

Conventions lexicales

- Les noms des fonctions et des arguments doivent débuter par une minuscule. Par exemple :

```
myFun   fun1   arg_2   x'
```

- Par convention, les noms des listes ont une `s` dans leur suffixe.
  Par exemple :

```
xs   ns   nss
```

Conventions lexicales

- Les noms des fonctions et des arguments doivent débuter par une minuscule. Par exemple :

```
myFun   fun1   arg_2   x'
```

- Par convention, les noms des listes ont une `s` dans leur suffixe.
  Par exemple :

```
xs   ns   nss
```

La règle d'agencement ("Layout")

- Dans une séquence de définitions, chaque définition doit débuter exactement á la même colonne :

```
a = 10
b = 20
c = 30
```

```
a = 10
 b = 20
c = 30
```

```
  a = 10
b = 20
  c = 30
```

- La règle du layout (layout rule, off-side rule) permet d'éviter le recours à une syntaxe explicite pour grouper les définitions.

```
a = b + c
  where
    b = 1
    c = 2
d = a * 2
```

(groupage implicite)

signifie

```
a = b + c
  where
    {b = 1;
     c = 2}
d = a * 2
```

(groupage explicite)

- Comme dans d'autres langages : Python, YAML, ...

Commandes utiles

Commande	Signification
`:load nom`	charger le script nom
`:reload`	recharger le script courant
`:edit nom`	editer le script nom
`:edit`	editer le script courant
`:type expr`	montrer le type of expr
`:?`	montrer tous les commandes
`:quit`	quitter