Programmation fonctionnelle en Haskell

Licence Informatique

3e année option GL

UFR SET – Université de THIES

Pr Mouhamadou THIAM

Objectifs généraux

- Découvrir un « autre » type de programmation : la programmation fonctionnelle
  - fondé sur la notion de *fonction calculable (au sens mathématique)*
  - le typage (des données, des fonctions)
  - la récursivité
- Applications/initiation au calcul symbolique : manipulation d'expressions formelles, logique

Pr Mouhamadou THIAM

Plan

I. Bases du langage
   1. Evaluation d'expressions
   2. Types primitifs
   3. Fonctions : définition, typage, récursivité
   4. Types construits : paires et tuples

II. Listes
   1. Fonctions primitives
   2. Fonctions gardes
   3. Appel par filtrage
   4. Récursivité

Bases du langage

1. Evaluation d'expressions
2. Types primitifs
3. Fonctions : définition, typage, récursivité
4. Types construits : paires et tuples

Définitions

- Un langage interactif
- Le « prompt » >
- Type = ensemble de « données », muni d'opérations (fonctions)

Evaluation d'une expression

```
> 42
42
> 6*7
42
> sqrt 65
8.06225774829855
> 2 == 1+1
True
> 5 > 3*4
False
```

Typage des expressions

```
:set +t
> 6*7
42 it :: Integer
> 2<3
True it :: Bool
> sqrt 23
4.795831523312719 it :: Double
```


Caractère & types complexes

- Caractères

```
> 'a'
'a' it :: Char
```

- Paires

```
> (1,2)
(1,2) it :: (Integer, Integer)
```

Caractère & types complexes

- Listes

> *[1..5]*

[1,2,3,4,5] it :: [Integer]

- Strings = listes de caractères

> *”niania"++”dembele"*

”nianiadembele" it :: [Char]

Fonctionnelles (fonctions de fonctions)

> *(1+) 5*

6 it :: Integer

> *map (1+) [1..5]*

[2,3,4,5,6] it :: [Integer]

Fonctionnelles (fonctions de fonctions)

> even 4

True it :: Bool

> map even [1..10]

[False,True,False,True,False,True,False,True,False, True]

Types primitifs

- Types primitifs
  - Les entiers (types Int, Integer) : 1, -3...
  - Les booléens (type Bool) : True, False
  - Les caractères (type Char) : 'a', 'b'...
  - Les nombres « réels » (types Float, Double) : -1.2, 3.1416

Types dérivés

- Types dérivés (définis par des *constructeurs*)
  - Paires : (1,8) (‘a’,’b’)
  - Listes : [21,02,2008]
  - Fonctions : moyenne :: Float -> Float -> Float
- Types définis par le programmeur

Type entier : Int et Integer

- Types: Int = précision fixée – Integer = précision infinie
- Opérateurs : +, -, *, div, mod
- Comparateurs : <, <=, ==, /, >, >=, /=
  !! Ne pas confondre = (def de fonction) et == (arithmétique)
- Priorités: - (opposé) >> / >> * >> + et - (soustraction)
- Associativité : à gauche
  2*3-4+5 = ? -2+8/4*2 =? ...

Type entier : Int et Integer

```
fact :: Int -> Int -- factorielle
fact2 :: Integer -> Integer
> fact 20
-2102132736
> fact2 20
2432902008176640000
```


Les « réels » ou nombres *flottants*

- Types : Float et Double
- Approximation des Réels : Double meilleure (+ précis) que Float
- Opérateurs : +, *, -, sqrt, ...
- Comparaison : <, ==, ...
- Priorité, associativité : même ordre que les entiers

Les booléens

- *Des « valeurs » à part entières (comme les entiers, les réels, les caractères, ...)*
- Type : Bool
- 2 valeurs : True et False
- Opérateurs : && (et) || (ou) not
- Egalité: ==
- Priorité : not >> && >> ||

Les booléens : exemples

> True || False && False

??

> not False && False

??

Les caractères

- Type : Char
- Notation : ‘a’, ‘2’, ‘(‘, ‘+’
- Comparaison : ==, <, >
  - > *'a' < 'b'* → *True*
  - > *'Z' < 'a'* → *True*

Les caractères

- Intervalles
  - > ['a'..'z'] → "abcdefghijklmnopqrstuvwxyz"
- successeur :
  - > succ ‘a’ → ‘b’
  - > pred ‘f’ → ‘e’

Les fonctions (1)

- Programme Haskell = un ensemble de fonctions à (0), 1, 2, …, n arguments
- -- Exemple carré d'un nombre (entier)

```
carre :: Int -> Int      -- Typage
carre x = x*x            -- Définition
```

- -- moyenne arithmétique

```
moyenne :: Float -> Float -> Float
moyenne x y = (x+y)/2
```


Les fonctions (2)

- -- constante de gravité terrestre (m/s2)

```
g = 9.81 :: Float
```

- Une fonction peut être appliquée à des arguments

```
> carre 5
25
> moyenne 2.6 7.8
5.2
```

Les fonctions (3)

```
quad :: Int -> Int
quad x = carre (carre x)
```

- Définition fonction = « équation » (orientée) :
  `foo x1 x2 ... xn = Expr`
- foo = nom de la fonction (commence par une minuscule !)
- x1 x2 ... xn = arguments formels (des variables, en 1ère approx.)
- Expr = expression formée à partir de :
  - variables (x, y, toto), et constantes (pi, 23, 'a', ...)
  - fonctions prédéfinies : +, /, ...
  - fonctions définies par le programmeur : carre...

Les fonctions (4)

- Remarques
  - Notation préfixe : fonction avant ses arguments
  - Sauf « opérateurs infixes » (arithmétiques ou autres : =, *, <, &&, ...)
  - Arguments évalués avant que la fonction elle-même soit évaluée
  - Parenthèses : (f x), (f x y ...), (x+y), ... si nécessaire
  - Priorités :

  > carre 3 + 4 → ?

Les fonctions (5)

- Les fonctions sont typées
- Si type source de f = a et type arrivée = b, alors la fonction f a pour type : a -> b

  f :: a -> b
- Si f a plusieurs arguments de types a, b, ... et le type d'arrivée est q, alors f a pour type

  a -> b -> ... -> q

  f :: a -> b -> ... -> q

Les fonctions (6)

- Position d'un mobile soumis à une accélération constante

```
position :: Float -> Float -> Float -> Float -> Float
position x0 v0 gamma t = (1/2)*gamma*t^2 + v0*t + x0
```

- chute d'un corps, vitesse initiale nulle

```
altitude x0 t = position x0 0 (- g) t
```

Conditionnelles (1)

- **IF … THEN … ELSE**
- plusPetit :: Int -> Int -> Int -- f. à 2 arguments Entiers
- plusPetit x y = if x < y then x else y

  > plusPetit 7 4

  4

  > plusPetit (carre 3) (carre 4) → ?

- Remarques
  - Pas de if … Then «sans else »

Conditionnelles (2)

- Fonctions à valeur booléenne (prédicats)

  positif :: Int -> Bool

  positif x = if x >= 0 then True else False

  negatif :: Int -> Bool

  negatif x = if x <= 0 then True else False

  > positif 6 → True

  > positif (-6) → False

Conditionnelles (3)

- Égalité booléenne (exprime) équivalence logique

```
positif x = (x >= 0)
negatif x = (x <= 0)

:t (<=) -- <= est une fonction booléenne
(<=) :: (Ord a) => a -> a -> Bool
```

- test de crash

```
crash x0 t = (altitude x0 t) == 0
```


Conditionnelles (4)

- Division entière

```
> mod 7 3
1
> div 7 3
2
```

- -- divise x y : x divise y

```
divise :: Int -> Int -> Bool
divise x y = (mod y x == 0)
> divise 3 12
True
```

Conditionnelles (5)

- -- pair x : x est pair

```
pair :: Int -> Bool
pair x = (divise 2 x)

> pair 6
True
```


Récursivité (1)

- **Définition** : une fonction récursive (directe) est une fonction qui s'appelle soi-même
- La récursivité est
  - un mode de programmation *très sûr et élégant*
  - pour *certains types de données* : entiers, listes, arbres

Récursivité (1)

- + ou - présente ou marginale dans certains langages (impératifs, objet), c'est un mode *fondamental* en programmation fonctionnelle.
- Parenté forte avec la notion de ***raisonnement par récurrence*** (cf. cours de maths)

Récursivité (2)

- 1er exemple : Fonction factorielle
  !n = n * (n-1) * (n-2) ... 2 *1 pour n>0
  !0 = 1 (par convention)
- Relation de récurrence entre !n et !(n-1)
  !n = n * !(n-1)
  !0 = 1
- Se programme quasi texto en Haskell

```
fact :: Int -> Int
fact n = if n==0 then 1 else n*(fact (n-1))
```

  > fact 12 → 479001600

Récursivité (3)

- Remarque : Définition non circulaire car (fact n) appelle fact, mais sur une donnée de taille inferieure : (n-1)
- Exemple : Une suite définie par

  $u_0 = -2$ et $u_n = 3 + 4 * u_{n-1}$
- Série de Riemann : 1/n^a : converge ssi a >1

```
riemann1 n = if n==1 then 1
             else (1/n) + (riemann1 (n-1))
```


Récursivité (3)

- Série de Riemann : 1/n^a : converge ssi a >1

```
riemann2 n = if n==1 then 1
             else (1/n)^2 + (riemann2 (n-1))

riemann :: Int -> Float -> Float
riemann n a = ??
```

Récursivité (4)

- Fibonacci

  $u_0=u_1=1$ et $u_n = u_{n-1} + u_{n-2}$ pour n>1

  ```
  fibo n = if n==0 || n==1 then 1
           else (fibo (n-1)) + (fibo (n-2))
  ```

- Fonction 91 de McCarthy – (=> 91 si n<=101)

  ```
  mac :: Int -> Int
  mac n = if n > 100 then n-10 else (mac (mac (n+11)))
  ```


Récursivité (5)

- Déroulement des appels récursifs : factorial

  ```
  fact n = if n == 0 then 1 else n * fact (n-1)
  ```

  fact 3
  ⇓
  3 * (fact 2)
  ⇓
  2 * (fact 1)
  ⇓
  1 * (fact 0)
  ⇓
  1

- Résultat : 3 * 2 * 1 *1

Récursivité (6)

- Raisonnement par récurrence
  fact n = if n == 0 then 1 else n * fact (n-1)
- Si la donnée = 0 que doit valoir la fonction fact ?
  Réponse : alors fact vaut 1
- Supposons que la fonction calcule correctement
  fact n-1 = v [= (n-1)!]
- Comment construire fact n ?
  Réponse : fact n = n * v

Récursivité (6)

- ***On s'appuie donc sur une relation de récurrence mathématique.***
- ***Extension à d'autres données, notamment les listes, les arbres...***

Types paire, triplet, …

- Si type(x) = a et type(y) = b, alors on peut former la paire ou couple tel que type(x, y) = (a, b)
- Autrement dit : (a, b) est le type des couples « d'éléments de a et de b »,
- De même (a, b, c) est le type des triplets « d'éléments de a , b et c»
- Etc : (a, b, c, d) ...

Opérateurs **fst** et **snd**

- fst = premier élément d'une paire
- snd = second élément d'une paire

> snd (1,2) → 2

> ((fst (1,2)), (snd (1,2))) → (1,2)

> let h="hello" :: String

> let w="world" :: String

> snd (h, w) → "world »

Division entière

- division :: Int -> Int -> (Int, Int)
- division x y = ((div x y), (mod x y))
- > division 31 7
  (4,3)

Opérations vectorielles (1)

- dimension 2

```
type Vect2 = (Float, Float) -- Définition de type
-- somme
sommeVectDim2 :: Vect2 -> Vect2 -> Vect2
sommeVectDim2 v1 v2 = ( fst v1 + fst v2, snd v1 + snd v2 )

-- Produit extérieur
prodVectDim2 :: Float -> Vect2 -> Vect2
prodVectDim2 k v = (k*(fst v), k*(snd v))
```

Opérations vectorielles (2)

- Exemples

```
> sommeVectDim2 (3, -1) (1, 5)
(4.0, 4.0)
it :: Vect2

> scalaireVectDim2 3 it
(12.0, 12.0)
it :: Vect2
```


GHCI (1)

- Appeler Haskell sous un shell (Glasgow Haskell Compiler)

  $ ghci
- Charger un fichier contenant des programmes

  > :l toto.hs -- Notez l'expansion .hs
- Recharger le dernier fichier

  > :r
- Changer de répertoire (complétion ok!)

  > :cd niania/haskell/programs

GHCI (2)

- Entrer en mode « typage »
  > :set +t
- Demander le type d'une expression
  > :t monExp
- Quotes et autres conventions:
  'x' :: Char
  "xyz" :: String (= [Char])
  x... est une variable
  x' est une autre variable

GHCI (3)

- X... est un identificateur de type
- -- ceci est un commentaire
- Arrêter un programme qui boucle : ^C
- Sortir: :q
- Connaître cette liste : :?

GHCI (fin)

- Compilateurs et interprètes : Plusieurs implémentations de **Haskell**, en particulier : GHC et Hugs en version :
  - Interprète interactif : le chargement d'un programme produit un « pseudo code » chargé dans l'espace de travail et les lignes de commandes à la console sont interprétées « à la volée »
  - Compilateur : produit du code compilé, qui peut être intégré dans une application
  - Pour en savoir plus sur le langage et charger un compilateur Haskell : http://haskell.org/
- Sur les machines du libre service : GHCI est la version interprétée de GHC
- Recommandation : si vous chargez un Haskell sur votre machine personnelle, choisissez GHCI pour raisons de compatibilité avec les TP

Listes

1. Fonctions primitives
2. Fonctions gardes
3. Appel par filtrage
4. Récursivité