Programmation Fonctionnelle en Haskell

Licence « Sciences et Technologies »

Parcours « Informatique » – 2ème année

UFR Sciences – Université de Caen

Patrice Enjalbert

Département d'Informatique

Patrice.Enjalbert@info.unicaen.fr

Objectifs de ce cours

- 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

- Poursuivi en 3eme année d'Info

Chapitre 1 : Introduction

1. Exemples Introductifs
2. Types primitifs
3. Fonctions : définition, typage, récursivité
4. Types construits : paires et tuples

Pour en savoir plus sur le langage et charger un compilateur Haskell :

http://haskell.org/

Pour retrouver les cours et TD-TP :

http://users.info.unicaen.fr/~patrice/cours/Haskell/

1. Exemples Introductifs

-- Un langage interactif. Le « prompt » >
-- Evaluation d'une expression

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

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

```
-- Caractères
> 'a'
'a'
it :: Char
-- Paires
> (1,2)
(1,2)
it :: (Integer, Integer)
-- Listes
> [1..5]
[1,2,3,4,5]
it :: [Integer]
-- Strings = listes de caractères
> "sar"++"ko"
"sarko"
it :: [Char]
```

```
-- Fonctionnelles (fonctions de fonctions)
> (1+) 5
6
it :: Integer
> map (1+) [1..5]
[2,3,4,5,6]
it :: [Integer]
> even 4
True
it :: Bool
> map even [1..10]
[False,True,False,True,False,True,False,True,False,
  True]
```

2. Types primitifs

Type = ensemble de « données », muni d'opérations (fonctions)

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

Les entiers

- 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 =? …
- `Int` = précision fixée et `Integer` = précision infinie

```
fact :: Int -> Int        -- factorielle
fact2 :: Integer -> Integer
```

```
> fact 20
-2102132736
> fact2 20
2432902008176640000
```

Les « réels » = nbs *flottants*

- Types : `Float  Double`
- Approximation des Réels : Double meilleure (+ précise) que Float
- Opérateurs, comparaison : +, *, -, sqrt,… <, ==, …
- Priorité, associativité : cf 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 >> && >> ||

```
> True || False && False
??
> not False && False
??
```

Les caractères

- Type : `Char`
- Notation : ‘a’ ‘2’ ‘(‘ ‘+’
- Comparaison : ==, <, >

```
> 'a' <'b'
True
> 'Z'<'a'
True
```

- Intervalles

```
> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
```

- successeur :

```
> succ ‘a’
‘b’
> pred ‘b’
‘a’
```

3. Fonctions

1. Programme Haskell

= un ensemble de fonctions à (0),1,2,… n arguments

```
-- 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
-- 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
```

```
carre :: Int -> Int
carre x = x*x
quad :: Int -> Int
quad x = carre (carre x)
moyenne :: Float -> Float -> Float
moyenne x y = (x+y)/2
```

- Une définition de fonction = une « équation » (orientée) :

`foo x1 x2…xn = Expr`

foo = nom de la fonction (commence par une minuscule !)

`x1 x2…xn` = arguments formels (des variables, en première approx.)

`Expr` = expression formée à partir de :

- variables (x,y,toto), et constantes (pi, 23, ‘a’…)
- fonctions prédéfinies…: +, /…
- fonctions définies par le programmeur : carre…

```
carre :: Int -> Int
carre x = x*x
quad :: Int -> Int
quad x = carre (carre x)
moyenne :: Float -> Float -> Float
moyenne x y = (x+y)/2
```

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
?

```
carre :: Int -> Int
carre x = x*x
quad :: Int -> Int
quad x = carre (carre x)
moyenne :: Float -> Float -> Float
moyenne x y = (x+y)/2
```

- Les fonctions sont typées :

- Si le type source de `f` est `a` et le type arrivée est `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
```

Exemple : 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
```

2. Une première conditionnelle : `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` »

```
-- 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
```

- Autre définition

```
-- L'égalité booléenne exprime l'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
```

```
-- 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
-- pair x  : x est pair
pair :: Int -> Bool
pair x = (divise 2 x)
> pair 6
True
```

3. 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

Plus ou moins 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)

- Premier 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 texte en Haskell :

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

> fact 12
479001600
```

Remarque

Définition non circulaire car `(fact n)` appelle `fact`, mais sur une donnée de taille inférieure `: (n-1)`

- Suites et séries (exercices)

- Une suite du cours de Python (L1)

$u_0 = -2 \qquad u_n = 3 + 4 * u_{n-1}$

```
- Série de Riemann : 1/n^a : converge ssi a >1
riemann1 n = if n==1 then 1
                else (1/n) + (riemann1 (n-1))
riemann2 n = if n==1 then 1
                      else (1/n)^2 + (riemann2 (n-1))

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

```
-- Fibonacci
```

$u_0=u_1=1$ $\qquad\qquad$ $u_n = u_{n-1} + u_{n-2}$ 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)))
```

- Déroulement des appels récursifs

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

fact 3                          -- 3 /= 0
 ⇓
3 * (fact 2)                    -- 2 /= 0
      ⇓
  2 * (fact 1)                   -- 1 /= 0
        ⇓
    1 * (fact 0)                 -- 0 == 0
        ⇓
        1
Résultat : 3 * 2 * 1 *1
```

Raisonnement par récurrence

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

- Si la donnée = 0 que doit valoir la fonstion `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`

Ici on s'appuie donc sur une relation de récurrence mathématique
Extension à d'autres données, notamment les listes, les arbres…

4. Types (2): paires et tuples

- Si `x` a le type `a` et `y` le type `b`, alors on peut former la paire ou couple `(x,y)`, qui a le type `(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) …

```
> :t ('x','y')
('x','y') :: (Char, Char)
> :t (pair 2, (impair 2))
(pair 2, (impair 2)) :: (Bool, Bool)
```

- Opérateurs

  fst,snd : premiers et 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 (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))

> sommeVectDim2 (3,-1) (1,5)
(4.0,4.0)
it :: Vect2
> scalaireVectDim2 3 it
(12.0,12.0)
it :: Vect2
```

Quelques aspects pratiques

- Appeler Haskell sous un shell (Glasgow Haskell Compiler) (salles de TP)
  $ 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 Enseignement/Haskell/Programmes
- 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
  X… est un identificateur de type
  -- ceci est un commentaire
- Arrêter un programme qui boucle : ^C
- Sortir : :q
- Connaître cette liste : :?

- 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 qui est la version interpétée de GHC
- Recommandation : si vous chargez un Haskell sur votre machine personnelle, choisissez GHCI pour raisons de compatibilité avec les TP
- Pour retrouver les cours et TD-TP : http://users.info.unicaen.fr/~patrice/cours/Haskell/