Approches Fonctionnelles de la Programmation

Lisp / Haskell: Tutoriel des Différences

Didier Verna

didier@lrde.epita.fr
http://www.lrde.epita.fr/~didier

Table des matières

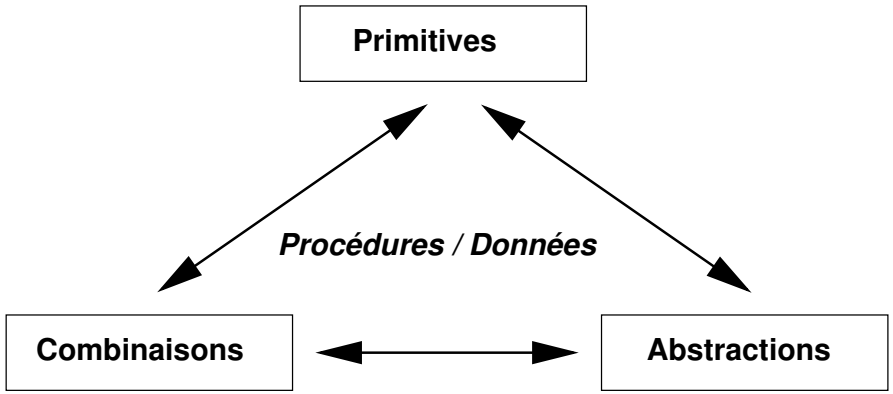
Syntaxe

Ou « abscence de . . . »

- **Mise en forme :**
  - **Haskell :** Indentation (« offside-rule », Peter J. Landin).
    Séparateur implicite : `';'`.
    Quelques mots réservés.
  - **Lisp :** Parenthèses (mais Cf. reader-macros).
    Aucun mot réservé.
- **Nommage :**
  - **Haskell :** Indentique au C, plus apostrophe.
    Première lettre capitalisée pour les types.
  - **Lisp :** N'importe quoi.
    Syntaxe spéciale pour les symboles ésothériques :
    `|...|`

Rappel : 3 types d'Expressions

3 aspects langagiers



- **Expressions littérales :**
  s'évaluent à elles-mêmes
- **Expressions combinées :**
  application de *procédures* (opérateurs, fonctions) à des *arguments* (expressions)
- **Expressions abstraites :**
  nommage et assignation d'expressions (simples, combinées, fonctionnelles)

Combinaison
Opérateurs et fonctions

- **Lisp :** Pas de distinction
  (sauf opérateurs spéciaux)
  - `(f arg...)`
  - Notation exclusivement préfixe (ou pas : Cf. macros)
  - Opérateurs variadiques
  - Imbrication naturelle (pas d'ambiguïté)
- **Haskell :** Distinction
  (mais ponts entre les deux notations)
  - `3 + 4` ⇔ `(+) 3 4`
  - `div 3 4` ⇔ `3 `div` 4`
  - Problèmes d'associativité et de précédence
    (`f n+1`, `f -12` *etc.*)
  - Définition d'opérateurs par notation préfixe
    `! # $ % & * + > / < = > ? \ ^ | : - ~`

Variadicité en Lisp
De l'influence de la syntaxe...


```
(defun mklist (head &rest tail)
  (cons head tail))
;; (mklist 'a 'b 'c)

(defun msg (str &optional (prefix "error: ") postfix)
  (concatenate 'string prefix str postfix))
;; (msg "hello" nil "!")

(defun msg* (str &key prefix (postfix "."))
  (concatenate 'string prefix str postfix))
;; (msg* "hello" :prefix "Me: ")
```

- Plus &-combinaisons ...

Abstraction
Nommage et assignation

Haskell

```
foo, bar, boo :: Float
foo = 10
bar = sqrt (3 * (6 + 7) - 8)
boo = bar

baz :: Float -> Float -> Float
baz a b = sqrt (3 * (a + 7) - b)
```

Lisp

```
(defvar *foo* 10)
(setq *foo* 10)
(setq bar (sqrt (- (* 3 (+ 6 7)) 8)))
(setq boo bar)

(defun baz (a b)
  (sqrt (- (* 3 (+ a 7)) b))))
```

Remarque :

- **Haskell :** abstraction syntaxique (déclarations)
- **Lisp :** abstraction fonctionnelle (expressions)

La boucle « read-eval-print »

Développement interactif

- **Boucle d'évaluation** (REPL) :
  - **Read :** saisir une *expression*
  - **Eval :** calculer (« évaluer ») sa valeur
  - **Print :** présenter le résultat *sous forme affichable*
- **Remarques :**
  - Haskell : REPL limitée (expressions *vs.* déclarations)
  - Lisp : interprétation / byte- [JIT] compilation au choix

Typage dans les langages fonctionnels
Et dans les autres. . .

Problème orthogonal à la programmation fonctionnelle
Il n'y a qu'à regarder les politiques d'Ada, C, Ruby, PHP. . .

- **Lisp : Typage dynamique**
  - Les *valeurs* sont typées
  - Vérification de type à l'*exécution*
- **Haskell : Typage statique**
  - Les *variables* sont typées
  - Vérification de type à la *compilation*

Lisp : Typage Dynamique

Et « fort »

- **Variables non typées :**
  variable = pointeur vers un *objet* Lisp
- **Valeurs typées :**
  information contenue dans chaque objet
- **Information de typage accessible :** (fonctionnelle)

```
(type-of this)
(typep that 'integer)
```

- **Attention au vocabulaire :**
  - Typage *implicite* (au moins dans le code)
  - *Vérification* de type dynamique (à l'exécution)
  - Typage *fort*
    Toute erreur de type est détectée (mais à l'exécution)

Lisp : Typage Statique

Et « faible »

- **La légende de la lenteur :**
  Typage dynamique ⇒ vérification au vol ⇒ lenteur
- **Sauf que :**
  - Structures de données « modernes »
    (tableaux, tables de hash *etc.*)
  - Déclarations *explicites* de type

```
(declare (type fixnum foobar))
```

  - Niveaux d'optimisation des compilateurs

```
(declare (optimize (speed 3) (safety 0) (debug 0)))
```

- **Remarque :** *déclaration* ≠ *expression* ; -)

Haskell : Typage statique

Le typage reste « fort »

- **Variables typées :**
  Rappel : variable $=$ constante
  $\Longrightarrow$ Type constant
- **Information de typage accessible :**
  Mais construction non langagière (Hugs `:type`)
- **Attention au vocabulaire :**
  - Typage *explicite ou implicite*
    Inférence / Polymorphisme
  - *Vérification* de type statique (à la compilation)
  - Typage *fort*
    Toute erreur de type est détectée (à la compilation)

Haskell : Typage statique et polymorphisme

Travailler sur plusieurs types

Lisp

```
(defun length (l)
  (if (null l) 0
      (1+ (length (cdr l)))))
```

Haskell

```
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs
```

- **Quel est le type Haskell de `length` ?**
  - `[Int] -> Int` ? `[String] -> Int` ? ... ?
- **Typage polymorphe :**
  - Variable de type : `length :: [a] -> Int`
  - Mais les listes restent homogènes...
  - `:type` retourne le type le *plus général*.
- **Polymorphisme *vs.* Surcharge** :
  - Polymorphisme : définition *unique* $\forall$ type
  - Surcharge : (overloading) définitions $\neq$ selon le type

Typage Lispien *vs.* Haskellien

Mérites comparés

- **Lisp :**
  - Typage très libéral
  - Polymorphisme *de-facto* ou explicite (introspection, OOP, macros *etc.*)
  - Compromis efficacité / sûreté
  - **Mais :** Cf. Qi / Racket (typage statique et fort)
- **Haskell :**
  - Typage très contrai(gna)nt
  - Polymorphisme « rigide » (variables de type, types algébriques *etc.*)
  - Sûreté *de-facto* sans compromis
- **Le vrai défi du 21^{e} siècle :**
  - Languages statiques *et* dynamiques

Booléens

Vrai ou faux ?

- **Haskell :**
  - Type `Bool`
  - Valeurs `True` et `False`
  - Opérateurs : `&&`, `||`, `not`
  - Fonctions : `and, or :: [Bool] -> Bool`
- **Lisp :**
  - Pas de type
  - Valeurs `t` et tout sauf `nil`, et `nil` / `()`
  - Opérateurs spéciaux : (variadiques) `and`, `or`, `not`

Expressions conditionnelles

P'têt ben qu'oui, p'têt ben qu'non...

`if then else` :

Haskell

```
max3 :: Int -> Int -> Int -> Int
max3 m n p = if (m >= n) && (m >= p) then m
             else if (n >= m) && (n >= p)
                  then n else p
```

Lisp

```
(defun max3 (m n p)
  (if (and (>= m n) (>= m p))
      m
    (if (and (>= n m) (>= n p))
        n
      p)))
```

Guardes et `cond` :

Haskell

```
max3 :: Int -> Int -> Int -> Int
max3 m n p
     | (m >= n) && (m >= p) = m
     | (n >= m) && (n >= p) = n
     | otherwise = p
```

Lisp

```
(defun max3 (m n p)
  (cond ((and (>= m n) (>= m p))
         m)
        ((and (>= n m) (>= n p))
         n)
        (t p)))
```

Haskell : Conditionnels *vs.* Équations

Quand on peut pattern matcher...


```
ssq :: Int -> Int
ssq n = if (n == 1) then 1
        else n*n + ssq (n-1)
```

```
ssq :: Int -> Int
ssq n
    | (n == 1) = 1
    | otherwise = n*n + ssq (n-1)
```

- Mise en équations préférable

```
ssq :: Int -> Int
ssq 1 = 1
ssq n = n*n + ssq (n-1)
```

- Conditionnel par pattern matching :

```
last :: [a] -> a
last x = case (reverse x) of
           [] -> error "Empty list"
           (r:rs) -> r
```

Lisp : Autres conditionnels

More than one way to do it. . .

Sur des objets :

```
(defun month-length (month)
  (case month
    ((jan mar may jul aug oct dec) 31)
    ((apr jun sept nov) 30)
    (feb 28) ;; Y2K bug !!!
    (otherwise "Unknown month !")))
```

Sur des types :

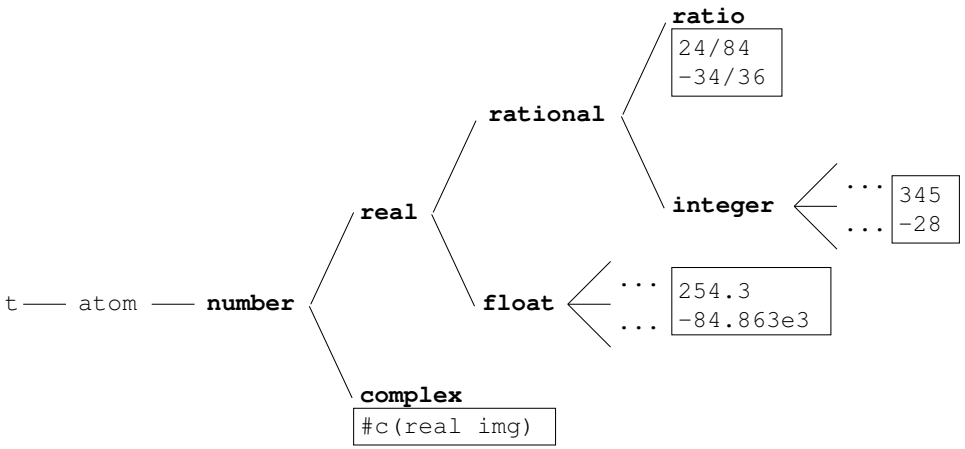
```
(defun +func (x)
  (typecase x
    (number #'+)
    (list #'append)
    (t #'list)))

(defun my+ (&rest args)
  (apply (+func (car args)) args))
```

Autres :

- `when`, `unless` *etc.*

Lisp : Types numériques

Un deux trois nous irons au bois. . .



- Fonctions `floatp`, `integerp` *etc.*
- **Transtypage implicite :**
  ratios vers entiers, complexes vers réels
- **Transtypage opérationnel :**
  - automatique : vers flottants ou complexes
  - explicite : `float`, `coerce`

Haskell : Types numériques

Quatre cinq six cueillir des cerises

- Même notation qu'en Lisp (ou autre)
- Hiérarchie similaire (`Int`, `Float` *etc.*)

- Surcharge des littéraux
  (2 est à la fois un `Int` et un `Float`)
- Surcharge des opérateurs

```
(==) :: Int -> Int -> Bool
(==) :: Float -> Float -> Bool
```

- Pas de transtypage opérationnel automatique

```
fromIntegral :: <integral> -> <number>
```

Opérations

Sept huit neuf dans mon panier neuf

- **Extraction :**
  - `truncate`, `floor`, `ceiling`, `round`
- **Arithmétique :**
  - `+ - *`, `div` (Haskell), `mod`, `abs`, `signum`
- **Relationnel :**
  - `< > <= >= /=`, `==` (Haskell), `= eql` (Lisp)
- **Exponentiation :**
  - `exp`, `log`, `logBase` (Haskell), `expt` (Lisp), `^ **` (Haskell), `sqrt`
- **Trigonometrie :**
  - `pi sin cos tan acos...`

Listes

Type natif dans tous les langages fonctionnels

- **Syntaxe :**
  - Haskell : `[e1, e2, e3, ...]`
  - Lisp : `(e1 e2 e3 ...)`
- **Type :**
  - Haskell : Listes homogènes.
    $\forall$ `t`, $\exists$ `[t]` (liste d'éléments de type `t`).
  - Lisp : Listes hétérogènes. Prédicat `listp`.
- **Liste vide :**
  - Haskell : `[]`. Appartient à tous les types de liste.
  - Lisp : `()`, `nil` (Cf. latin *nihil*).

Constructeurs de listes

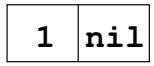
Au commencement, il y avait `cons`...

- **Représentation :**
  - Toute liste peut être décrite comme un élément de tête (head) et le reste / la queue de la liste (tail).
  - En théorie, les listes sont représentées par constructions successives au dessus de la liste vide.
- **Construction :**
  - Haskell : opérateur `(:) :: a -> [a] -> [a]`
  - Lisp : fonction `cons`, prédicat `consp`
  - « Constructeur » $\Rightarrow$ unicité de la construction
    Cf. `++`, `append` *etc.*
- **Génération :**
  - **Lisp seul :** Fonction `list` (Aucun sens en Haskell)
  - **Autres :**
    - Lisp : `(make-list n &key initial-element)`
    - Haskell : `replicate :: Int -> a -> [a]`

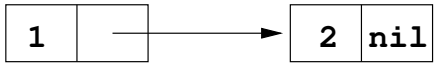
Représentation emboîtée

Lisp : « cons » = paire de pointeurs

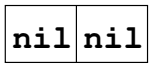

```
(1) (cons 1 nil)
[1] 1:[]
```



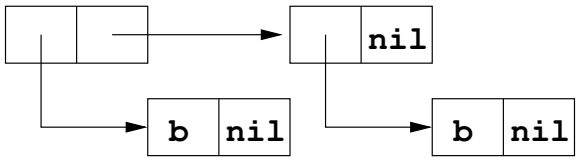

```
(1 2)  (cons 1 (cons 2 nil))
[1, 2] 1:(2:[])
```



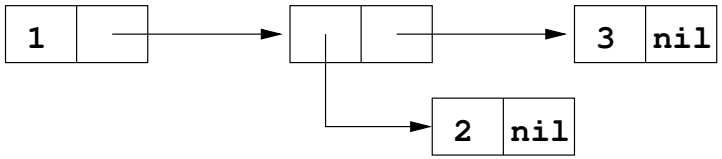

```
(nil) (cons nil nil)
[[]]  []:[]
```




```
((1) (2))  (cons (cons 1 nil) (cons (cons 2 nil) nil))
[[1], [2]] (1:[]):((2:[]):[])
```




```
(1 (2) 3) (cons 1 (cons (cons 2 nil) (cons 3 nil)))
```



Haskell : Remarques sur le constructeur

En vrac

- **Associativité :** à droite.
  `[3,4] = 3:[4] = 3:(4:[]) = 3:4:[]`
- **Précédence :** l'application lie toujours plus fort.
  $\Rightarrow$ toujours parenthéser !

```
tail :: [a] -> [a]
tail [] = []
tail (x:xs) = xs
```

- **Pattern matching :** répétition de variable interdite.

```
elt :: a -> [a] -> Bool
elt x []     = False
elt x (x:xs) = True       -- Barf !!
elt x (y:ys) = elem x ys
```

Accesseurs de listes

Éléments, ou `cons`

- **Lisp :**
  - `car`, `cdr`, `c[ad]+r` (jusqu'à 4) en termes de `cons` (Cf. IBM-704 : **C**ontents of **A**dress/**D**ecrement **R**egister)
  - `last`, `(nthcdr n lst)` en termes de `cons`
  - `first`...`tenth`, `(nth n lst)` en termes d'éléments
- **Haskell :**
  - Pattern matching : `(x:xs)`
  - `head`, `tail` en termes de `cons`
  - `(!!) :: [a] -> Int -> a`, `last` ($\neq$ Lisp) en termes d'éléments

Haskell : Énumérateurs

Passe-moi l'sucre...

- **2 formes d'énumération :**

```
-- [n .. m]

[2 .. 7]
[3.1 .. 7.0]
```

```
-- [n,p .. m]

[7,6 .. 3]
[0.0,0.3 .. 1.0]
```

- **Remarque :** sucre syntaxique

Haskell : « Compréhensions » de listes

Générateurs par extraction, test et transformation

$$[f(n)|n \in lst, C_1(n), C_2(n), \ldots]$$

- **Littéralement :** la liste des $f(n)$ telle que $n \in lst$ et $\forall i, C_i(n)$ est vraie.
- **Générateur :** `n <- lst` en Haskell.
- **Exemples :**

```
[ 2*n     | n <- [2,3,4] ]
[ n == 3  | n <- [2,3,4] ]
[ 2*n     | n <- [2,3,4], n == 3]
```

- **Pattern matching :** dans le générateur

```
[ m*n     | [m,n] <- [[2,3],[4,5],[6,7]], m > 2]
```

Remarques sur les compréhensions

En vrac...

- Utilisation comme filtre :

```
import Char

digits :: String -> String
digits str = [ c | c <- str, isDigit c ]
```

- Utilisation en corps de fonction :

```
isEven :: Int -> Bool
isEven n = (n `mod` 2 == 0)

allEven :: [Int] -> Bool
allEven xs = (xs == [e | e <- xs, isEven e])

allOdd :: [Int] -> Bool
allOdd xs = ([] == [e | e <- xs, isEven e])
```

Remarques sur les compréhensions (suite)

Suite du vrac. . .

- **Attention :** variables locales aux compréhensions

```
bogusFind :: Int -> [[Int]] -> [[Int]]
bogusFind x ys = ([ x:zs | x:zs <- ys])
```

- Sucre syntaxique

Séquences
Autre vision des listes

- **Généralités :**
  - Vision des listes comme séries ordonnées d'éléments
  - Concept optionnel du point de vue théorique
  - Exemples : chaînes de caractères, vecteurs (natifs en Lisp)...
  - Typage : séquences homogènes en Haskell, hétérogènes en Lisp
- **Lisp :**
  - Fonction `(elt seq pos)`, équivalent à `nth` mais pour tout type de séquence.

Chaînes de caractères

« chaîne » ⇔ liste ou vecteur

- **Caractères :**
  - Expression littérale : `'c'` (Haskell), `#\c` (Lisp)
  - Type Haskell : `Char`, prédicat Lisp : `characterp`
  - Encodage : `char-code` et `code-char` (Lisp), `ord` et `chr` (Haskell)
  - Comparaison (Lisp) : `char=`, `char>=`, `char/=` *etc.*
- **Chaînes :**
  - Expression littérale : `"abcde"`
  - Type Haskell : `String` ⇔ `[Char]`
  - Type Lisp : *vecteur* de caractères. Prédicat `stringp`.
  - Caractères spéciaux : `\n` (Haskell), `format` (Lisp)

Listes *vs.* vecteurs de caractères

Haskell *vs.* Lisp

- **Haskell :**
  - Le type `String` est un *alias* du type `[Char]`.
  - Tout ce qui marche sur les listes s'applique (constructeur, générateurs, compréhensions).

Haskell

```
'a':'b':'c':[]
['a' .. 'c']
['a','c' .. 'z']
[c | c <- "abc", c > 'b']
```

- **Lisp :**
  - Listes et chaînes (vecteurs) sont deux choses différentes.

Lisp

```
"abc"
(#\a #\b #\c)
```

Recherche / indexation / filtrage

Manipulation de séquences

- **Recherche :**
  - Lisp : `(member elt lst &key ...)` (réservé aux listes), `(find elt seq &key ...)`
  - Haskell : `elem :: Eq a => a -> [a] -> Bool`
- **Indexation :**
  - Lisp : `(position elt seq &key ...)`
  - Haskell :
    `elemIndex :: Eq a => a -> [a] -> Int`
- **Filtrage :**
  - Lisp : `(remove elt seq &key ...)`
    Note : clé `:count`
  - Haskell : `delete :: Eq a => a -> [a] -> [a]`
    Note : seulement le premier élément

Contraction, Extraction

Sur-/sous-séquences

- **Concaténation :**
  - Haskell : opérateur `(++) :: [a] -> [a] -> [a]`
    Fonction `concat :: [[a]] -> [a]`
  - Lisp : `append`, `(concatenate type &rest elts)`
    Attention : `append` ne copie pas son dernier argument, `concatenate` copie tout.
- **Sous-séquences :**
  - Haskell : `take, drop :: Int -> [a] -> [a]`
  - Lisp : `(subseq seq first &optional last)`
- **Autres :**
  - `reverse`, `length`

La méthode de Newton-Raphson

Approche par itérations successives

$$\sqrt{x} = \lim_{n \to +\infty} y_n \quad | \quad y_n = \frac{y_{n-1} + \frac{x}{y_{n-1}}}{2}$$

- Itérer la suite y_n jusqu'à ce l'approximation soit satisfaisante.

`nrsqrt` en Lisp

Approche fonctionnelle (pure)

Lisp

```
(defun nrsqrt (x delta)
  (nrfind x 1f0 delta))

(defun nrfind (x yn delta)
  (if (nrhappy yn x delta)
      yn
      (nrfind x (nrnext yn x) delta)))

(defun nrhappy (yn x delta)
  (<= (abs (- x (* yn yn))) delta))

(defun nrnext (yn x)
  (/ (+ yn (/ x yn)) 2))
```

nrsqrt en Haskell

Approche fonctionnelle (pure)

Haskell

```
nrsqrt :: Float -> Float -> Float
nrsqrt x delta = nrfind x 1.0 delta

nrfind :: Float -> Float -> Float -> Float
nrfind x yn delta
    | nrhappy yn x delta = yn
    | otherwise = nrfind x (nrnext yn x) delta

nrhappy :: Float -> Float -> Float -> Bool
nrhappy yn x delta = abs (x - yn * yn) <= delta

nrnext :: Float -> Float -> Float
nrnext yn x = (yn + x / yn) / 2
```

Ou bien ? . . .

Approche carrément pas fonctionnelle pas pure du tout

Lisp

```
(defun nrsqrt (x delta)
  (loop for y = 1f0 then (/ (+ y (/ x y)) 2f0)
        until (<= (abs (- x (* y y))) delta)
        finally (return y)))
```

- **Rappel :** « quoi faire » *vs.* « comment faire »
- **Question :** un algorithme contient-il son propre paradigme d'expression ?

Les opinions varient. . .