

Expressions

Typage

Booléens

Nombres

Listes

Homoiconicité

nrsqrt

Approches Fonctionnelles de la Programmation

～ Lisp / Haskell : Tutoriel des Différences ～

Didier Verna

didier@didierverna.net



didierverna.net



[@didierverna](https://twitter.com/didierverna)



[didier.verna](https://facebook.com/didier.verna)



[in/didierverna](https://in.linkedin.com/company/didierverna)

Plan

Syntaxe et Expressions

Typage et Vérification

Logique Booléenne

Arithmétique

Listes

Corollaire: Homoiconicité (Lisp)

sqrt par Newton-Raphson

Plan

Syntaxe et Expressions

Typage et Vérification

Logique Booléenne

Arithmétique

Listes

Corollaire: Homoiconicité (Lisp)

sqrt par Newton-Raphson

Syntaxe

► Mise en forme

- **Haskell** : « offside-rule » [Landin, 1966]
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 : | . . . |

Opérateurs et Fonctions

- ▶ **Lisp** : pas de distinction (sauf « opérateurs spéciaux »)
 - ▶ `(f arg1 arg2 ...)`
 - ▶ Notation exclusivement préfixe (mais cf. macros)
 - ▶ Opérateurs variadiques
 - ▶ Pas d'ambiguïté (précédence, associativité, etc.)
- ▶ **Haskell** : distinction (mais ponts entre les deux notations)
 - ▶ `f arg1 arg2 ...`
 - ▶ `3 + 4` $\Leftrightarrow$ `(+) 3 4`
 - ▶ `div 3 4` $\Leftrightarrow$ `3 `div` 4`
 - ▶ Ambiguïtés (précédence, associativité, etc.)
(`f n+1`, `f -12` etc.)
 - ▶ Définition d'opérateurs par notation préfixe
`! # $ % & * + > / < = > ? \ ^ | : - ~`

Variadicité en Lisp

```
(defun mklist (head &rest tail)
  (cons head tail))
;; (mklist 1) => (1)
;; (mklist 1 2) => (1 2)
;; (mklist 1 2 3) => (1 2 3)
;; etc.

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

(defun msg* (str &key prefix (postfix "."))
  (concatenate 'string prefix str postfix))
;; (msg* "hello") => "hello."
;; (msg* "hello" :prefix "me: ") => "me: hello."
;; (msg* "hello" :postfix "!" :prefix "me: ") => "me: hello!"
```

- ▶ Plus &-combinaisons
- ▶ Remarque: 'string = (quote string) $\Rightarrow$ string (symbole)

Nommage et Affectation

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))))
```

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

Nommage, Affectation, et Pureté

- ▶ **Nommage** (fonctionnel pur) : *donner un nom à une valeur*
Une valeur peut donc avoir différents noms
- ▶ **Affectation** (fonctionnel impur) : *donner une valeur à un nom*
Un nom peut donc avoir différentes valeurs

« Lisp-2 »

► Chaque symbole a *deux* valeurs

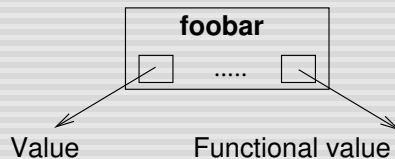
- une valeur quelconque
- une valeur fonctionnelle
- 💡 ces valeurs coexistent

► Accès

- `foobar` $\Rightarrow$ *valeur*
- `#'foobar` $\Rightarrow$ *valeur fonctionnelle*

► Remarques

- Rappel: `'foobar` $=$ `(quote foobar)` $\Rightarrow$ `foobar` (*symbole*)
- `#'foobar` $=$ `(function foobar)` $\Rightarrow$... (*fonction*)



```
(defvar foobar 3)
(defun foobar (x) (* 2 x))
;; (foobar foobar) => 6
```

Développement Interactif

► Boucle « read-eval-print » (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

Plan

Syntaxe et Expressions

Typage et Vérification

Logique Booléenne

Arithmétique

Listes

Corollaire: Homoiconicité (Lisp)

sqrt par Newton-Raphson

Typage et Vérification

- ▶ **Remarque** : question orthogonale au paradigme fonctionnel
- ▶ **Typage statique** (Haskell)
 - ▶ Typage des *variables / fonctions*
 - ▶ Vérification de type à la *compilation*
- ▶ **Typage dynamique** (Lisp)
 - ▶ Typage des *valeurs*
 - ▶ Vérification de type à l'*exécution*

Typage Statique (Haskell)

- ▶ Information de typage *inutile* à l'exécution
Vérifications faites à la compilation
- ▶ Accès dynamique potentiel (introspection, REPL)
P.ex. : type dans GHCi

! Attention au vocabulaire

- ▶ *Vérification* de type *statique*: à la compilation
- ▶ Typage *explicite* ou *implicite*: Inférence / Polymorphisme
- ▶ Typage *fort*: toute erreur de type est détectée (à la compilation)

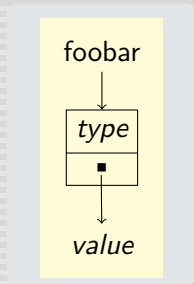
Typage Dynamique (Lisp)

- ▶ Information de typage *nécessaire* à l'exécution
Vérifications faites à l'exécution
- ▶ Boxing : typage des *valeurs*
- ▶ Accès dynamique langagier (fonctionnel)

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

⚠ Attention au vocabulaire

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



Typage Statique et Polymorphisme

Lisp

```
(defun len (l)  
  (if (null l) 0 (+ (len (cdr l)))))
```

Haskell

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

- ▶ **Type de length en Haskell**
 - ▶ `[Int] -> Int` ? `[String] -> Int` ? ... ?
- ▶ **Polymorphisme paramétrique**
 - ▶ 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 : définitions $\neq$ selon le type

Typage Statique vs. Dynamique

► Dynamique

- Plus expressif (polymorphisme *de-facto*)
- Impact sur les performances et la sûreté

► Statique

- Plus contraignant (polymorphisme paramétrique)
- Sûreté *de-facto*

► Mélange statique / dynamique (sujet brûlant !)

- Bibliothèques (*p.ex.* Shen, typage statique et fort)
- Types
 - C# `dynamic` : court-circuite les vérifications statiques
 - Haskell `Dynamic + toDyn + fromDynamic`
- Annotations statiques Common Lisp (optimisation / typage faible)
- Typage graduel [Siek, 2006]

Plan

Syntaxe et Expressions

Typage et Vérification

Logique Booléenne

Arithmétique

Listes

Corollaire: Homoiconicité (Lisp)

sqrt par Newton-Raphson

Booléens

► Haskell

- Type `Bool`
- Valeurs `True` et `False`
- Opérateurs: `&&`, `||`, `not`
- Fonctions: `and`, `or` `:: [Bool] -> Bool`

► Lisp

- Pas de type spécifique
- Vrai: `t` et tout sauf `nil`
- Faux: `nil` et `()`
- Opérateurs spéciaux (variadiques): `and`, `or`
- `not`

Branchements Conditionnels

► 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)))
```

Conditionnels vs. Équations (Haskell)

```
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 & Pattern matching

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

Autres Conditionnels (Lisp)

► 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 (error "Unknown month !"))))
```

► Sur des types

```
(defun +func (x)
  (typecase x
    (number #'+)
    (list #'append)))

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

► Autres : when, unless, etc.

Plan

Syntaxe et Expressions

Typage et Vérification

Logique Booléenne

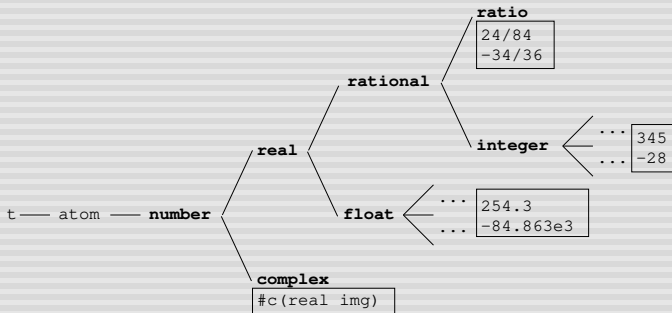
Arithmétique

Listes

Corollaire: Homoiconicité (Lisp)

sqrt par Newton-Raphson

Types Numériques (Lisp)



- ▶ **Prédicats** : `floatp`, `integerp`, *etc.*
- ▶ **Transtypage implicite** : ratios vers entiers, complexes vers réels
- ▶ **Transtypage opérationnel**
 - ▶ Automatique : vers flottants ou complexes
 - ▶ Explicite : `float`, `coerce`

Types Numériques (Haskell)

- ▶ Dénotation usuelle
- ▶ Hiérarchie similaire sous la classe Num (Int, Float etc.)
(*Inspirée de Scheme (inspirée de Common Lisp)*)
- ▶ **Surcharge des littéraux**
`42 :: Num p => p`
- ▶ **Opérateurs polymorphes**
`(+) :: Num a => a -> a -> a (==) :: Eq a => a -> a -> Bool`
- ▶ **Pas de transtypage opérationnel automatique !**
`fromIntegral :: (Integral a, Num b) => a -> b`

Plan

Syntaxe et Expressions

Typage et Vérification

Logique Booléenne

Arithmétique

Listes

Corollaire: Homoiconicité (Lisp)

sqrt par Newton-Raphson

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: `[]` (polymorphe, type `[a]` $\forall a$)
- Lisp: `()` $\Leftrightarrow$ `nil` (du latin *nihil*)

Construction

► Forme canonique

- Tête (head) + reste / queue (tail)
- Constructions successives au dessus de la liste vide

► **Haskell** : $(:) :: a \rightarrow [a] \rightarrow [a]$

► **Lisp** : cons (prédicat consp)

► **Remarque** : unicité de la « construction »
 $\neq$ « génération » (++, append, etc.)

Énumérateurs (Haskell)

```
-- Forme 1: [n .. m]
[2 .. 7]
[3.1 .. 7.0] -- [3.1,4.1,5.1,6.1,7.1]

-- Forme 2: [n,p .. m]
[7,6 .. 3]
[0.0,0.3 .. 1.0] -- [0.0,0.3,0.6,0.8999999999999999]
```

Exemples



```
(1) (cons 1 nil)
```

```
[1] 1:[]
```



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

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



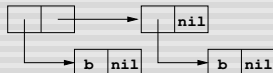
```
(nil) (cons nil nil)
```

```
[] []:[]
```

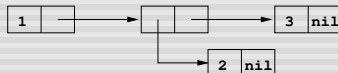


```
((1) (2)) (cons (cons 1 nil) (cons (cons 2 nil) nil))
```

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



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



► **Remarque :** pureté vs. notion(s) d'égalité

Remarques sur le Constructeur (Haskell)

► **Associativité** : droite

$[3,4] = 3:[4] = 3:(4:[]) = 3:4:[]$

► **Précédence** : l'application lie toujours plus fort

⇒ *toujours parenthéser !*

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

⚠ **Attention** : variables locales au pattern matching

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

Accessseurs

► Canoniques

- Lisp: `car / first, cdr / rest`
Cf. IBM-704: **C**ontents of **A**ddress / **D**ecrement **R**egister)
- Haskell: `head, tail`
Pattern matching: `(x:xs)`

► Positionnels

- Lisp: `first ... tenth, (nth n lst)`
- Haskell: `last, (!! ) :: [a] -> Int -> a`

► Structurels

- Lisp: `(nthcdr n lst), last (≠ Haskell!)`
- Haskell: `drop :: Int -> [a] -> [a]`

Plan

Syntaxe et Expressions

Typage et Vérification

Logique Booléenne

Arithmétique

Listes

Corollaire: Homoiconicité (Lisp)

sqrt par Newton-Raphson

Corollaire : Homoiconicité (Lisp)

► Syntaxe

- S-Expression : atome ou liste
- Atome : littéral (1.5, "foo", etc.) ou symbole
- Symbole : foobar
- Liste : (a "foo" (bar +) 2.5), etc.
En particulier : (+ 1 2), (defvar foo 1), etc.

► Homoiconicité : « même représentation »

► Opérateur crucial : quote

Lisp is the greatest single programming language ever designed.

– Alan Kay [Kay, 2017]

Puissance

► Passage au méta-niveau

- Dites-moi votre nom `(prin1 your-name)`
- Thierry Chmonfiss `=> "Thierry Chmonfiss"`

- Dites-moi « votre nom » `(prin1 'your-name)`
- Votre nom `=> your-name`

► Méta-programmation (rendue triviale)

- Homoiconicité (code $\iff$ données) + quote
- `(+ 1 2)` vs. `'(+ 1 2)`
- Réflexivité totale: introspection + intercession

Méandres

► Propagation de la notion d'égalité :

– Trois égal deux plus un ?

`(= 3 (+ 2 1)) => T`

– « Trois » égal « deux plus un » ?

`(= '3 '(+ 2 1)) => nil`

► Inférence sur des prédicats :

« Les jazzmen sont d'excellents musiciens. »

– John Scofield est un jazzman.

⇒ John Scofield est un excellent musicien.

– Thierry sait que John Scofield est un jazzman.

⇒ Thierry sait-il que John Scofield est un excellent musicien ?

Plan

Syntaxe et Expressions

Typage et Vérification

Logique Booléenne

Arithmétique

Listes

Corollaire: Homoiconicité (Lisp)

sqrt par Newton-Raphson

La méthode de Newton-Raphson

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

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

 nrsqrt (Lisp)

Lisp

```
(defun nrsqrt (x delta)
  (nrfind x delta 1.0))

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

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

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

nrsqrt (Haskell)

Haskell

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

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

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

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

nrsqrt Impératif (Lisp)

```
(defun nrsqrt (x delta)
  (loop :for y = 1.0 :then (/ (+ y (/ x y)) 2.0)
        :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...



Bibliographie

Plan

Bibliographie

Bibliographie



Peter J. Landin.

The Next 700 Programming Languages.

[Communications of the ACM 9\(3\): 157–166.](#)



Jeremy Siek and Walid Taha.

Gradual Typing for Functional Languages.

[Scheme and Functional Programming Workshop.](#)



Alan Kay.

What did Alan Kay mean by, “Lisp is the greatest single programming language ever designed” ?

[Quora.](#)