



Programmation
Fonctionnelle

Didier Verna
EPITA

Techniques
d'évaluation

Stricte

Lazy

Comparaison

Scoping

Blocs

Scoping

Approches Fonctionnelles de la Programmation Évaluation et Scoping

Didier Verna

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



Table des matières

Programmation
Fonctionnelle

Didier Verna
EPITA

Techniques
d'évaluation

Stricte

Lazy

Comparaison

Scoping

Blocs

Scoping

1 Techniques d'évaluation

- Lisp : Évaluation stricte
- Haskell : Évaluation lazy
- Mérites comparés

2 Scoping

- Structure de blocs
- Scoping lexical vs. dynamique



Lisp : évaluation stricte

Ordre applicatif

Programmation
Fonctionnelle

Didier Verna
EPITA

Techniques
d'évaluation

Stricte

Lazy

Comparaison

Scoping

Blocs

Scoping

■ Déroulement : « tree accumulation »

- ▶ Évaluer les sous-expressions de gauche à droite
⇒ Processus récursif
- ▶ Appliquer l'opérateur à ses arguments

■ En bas de l'arbre :

- ▶ Données primitives (littéraux)
- ▶ Opérateurs primitifs (built-in)

■ Environnement :

- ▶ Valeurs d'expressions abstraites
- ▶ Les opérateurs primitifs en sont des cas particuliers



Exemple

$(2 + 4 * 6) * (x + 5 + 7)$

Programmation
Fonctionnelle

Didier Verna
EPITA

Techniques
d'évaluation

Stricte

Lazy

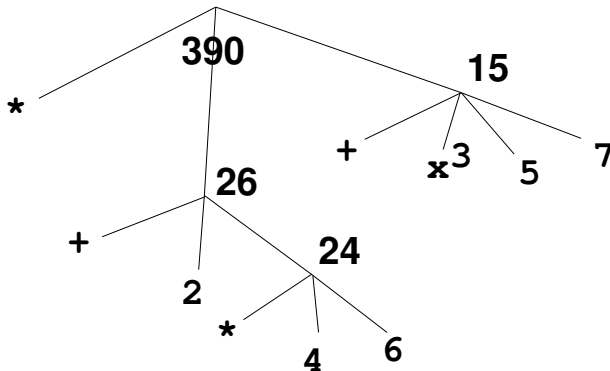
Comparaison

Scoping

Blocs

Scoping

$(* (+ 2 (* 4 6)) (+ x 5 7))$





Modèle de substitution

Évaluation des expressions fonctionnelles

Programmation
Fonctionnelle

Didier Verna
EPITA

Techniques
d'évaluation

Stricte

Lazy

Comparaison

Scoping

Blocs

Scoping

■ Définition :

- ▶ Identique au cas précédent
- ▶ *Substitution* : (étape préalable)
Remplacement des paramètres formels par les arguments correspondants.

■ Remarques :

- ▶ Substitution = vue de l'esprit.
Utilisation d'un environnement local pour les paramètres formels.
- ▶ Modèle formel mathématique de la substitution très complexe (collision de noms). Cf. λ -calcul.



Exemple

$f(5)$

Programmation
Fonctionnelle

Didier Verna
EPITA

Techniques
d'évaluation

Stricte

Lazy

Comparaison

Scoping

Blocs

Scoping

```
(defun sq (x) (* x x))  
(defun ssq (x y) (+ (sq x) (sq y)))  
(defun f (a)  
  (ssq (+ a 1) (* a 2)))
```

```
(f 5)  
(ssq (+ a 1) (* a 2))  
(ssq (+ 5 1) (* 5 2))  
(ssq 6 10)  
(+ (sq x) (sq y))  
(+ (sq 6) (sq 10))  
(+ (* x x) (* x x))  
(+ (* 6 6) (* 10 10))  
(+ 36 100)
```

136



Opérateurs spéciaux

Expressions à évaluation particulière

Programmation
Fonctionnelle

Didier Verna
EPITA

Techniques
d'évaluation

Stricte

Lazy

Comparaison

Scoping

Blocs

Scoping

■ Contexte :

- ▶ Problème : évaluation d'idiomes non stricts (ex. branchements conditionnels)
- ▶ Solution : opérateurs *primitifs spéciaux* (comportement d'évaluation particulier)
- ▶ Exemples : `if`, `setq`

■ Corollaire :

- ▶ *Quotation* : empêcher l'évaluation d'une expression
- ▶ Opérateur spécial `quote`, aussi noté `'`
- ▶ Exemple : `(special-operator-p 'if)`



Puissance de la quotation

– Dispatch

Programmation
Fonctionnelle

Didier Verna
EPITA

Techniques
d'évaluation

Stricte

Lazy

Comparaison

Scoping

Blocs

Scoping

■ Méta-Expressions

(expressions manipulant des expressions)

- Dites-moi votre nom `(println your-name)`
- Thierry Chmonfiss `=> "Thierry Chmonfiss"`
- Dites-moi « votre nom » `(println 'your-name)`
- Votre nom `=> your-name`

■ Utilité :

- ▶ Réflexivité
- ▶ Code \iff Données
- ▶ Macros
- ▶ Calcul symbolique
`(list a b c) \neq ' (a b c)`



Méandres de la quotation

– « Spatch » !

Programmation
Fonctionnelle

Didier Verna
EPITA

Techniques
d'évaluation

Stricte

Lazy

Comparaison

Scoping

Blocs

Scoping

■ Propagation de la notion d'égalité :

– Trois égal deux plus un ?

$(= \ 3 \ (+ \ 2 \ 1)) \Rightarrow T$

– « Trois » égal « deux plus un » ?

$(= \ '3 \ ' (+ \ 2 \ 1)) \Rightarrow nil$

■ Inférence sur des prédicats :

« Les jazzmen sont d'excellents musiciens. »

– John Scofield est un jazzman.

\Rightarrow John Scofield est un excellent musicien.

– Thierry sait que John Scofield est un jazzman.

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



Haskell : Évaluation lazy

Ordre normal

Programmation
Fonctionnelle

Didier Verna
EPITA

Techniques
d'évaluation

Stricte

Lazy

Comparaison

Scoping

Blocs

Scoping

■ Modèle de substitution :

- ▶ Remplacement des paramètres formels par les arguments correspondants

■ Mais :

- ▶ Arguments non évalués
- ▶ Évaluation « à la demande »
(aussi vrai pour les agrégats)

■ Attention :

- ▶ Évaluation *unique* des expressions
(travail sur un *graphe* d'évaluation)
- ▶ Fonctionnel pur uniquement



Exemple

$f(5)$

Programmation
Fonctionnelle

Didier Verna
EPITA

Techniques
d'évaluation

Stricte

Lazy

Comparaison

Scoping

Blocs

Scoping

```
sq :: Float -> Float
sq x = x * x
```

```
ssq :: Float -> Float -> Float
ssq x y = sq x + sq y
```

```
f :: Float -> Float
f a = ssq (a + 1) (a * 2)
```

f 5

ssq (a + 1) (a * 2)

ssq (5 + 1) (5 * 2)

sq x + sq y

sq (5 + 1) + sq (5 * 2)

(x * x) + (y * y)

(5 + 1) * (5 + 1) + (5 * 2) * (5 * 2)

6 * 6 + 10 * 10

36 + 100

136



Graphe d'évaluation

Évaluation unique des paramètres

Programmation
Fonctionnelle

Didier Verna
EPITA

Techniques
d'évaluation

Stricte

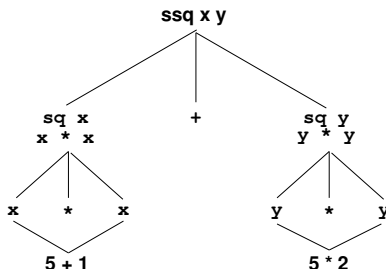
Lazy

Comparaison

Scoping

Blocs

Scoping



`ssq (5 + 1) (5 * 2)`

`sq (5 + 1) + sq (5 * 2)`

`(5 + 1) * (5 + 1) + (5 * 2) * (5 * 2)`

`6 * 6 + 10 * 10`



Évaluation d'équations multiples

Pattern Matching

Programmation
Fonctionnelle

Didier Verna
EPITA

Techniques
d'évaluation

Stricte

Lazy

Comparaison

Scoping

Blocs

Scoping

$f\ p1\ p2\ p3\ \dots = e1$

$f\ q1\ q2\ q3\ \dots = e2$

\dots

- **Équation utilisée** : la *première* offrant un matching entre les arguments et les formes paramétriques
- **Évaluation** :
 - ▶ Seuls les arguments nécessaires à la décision
 - ▶ Seuls les *morceaux* d'agrégats nécessaires



Évaluation *minimum* des arguments

Exemple 1

Programmation
Fonctionnelle

Didier Verna
EPITA

Techniques
d'évaluation

Stricte

Lazy

Comparaison

Scoping

Blocs

Scoping

```
foo :: Float  
foo = 2 * foo
```

```
prod :: Float -> Float -> Float  
prod x 0 = 0  
prod 0 y = 0  
prod x y = x * y
```

```
prod 3 4 => (3) 12.0
```

```
prod 4 0 => (1) 0.0
```

```
prod foo 0 => (1) 0.0
```

```
prod 0 foo => Stack overflow
```



Évaluation d'équations multiples

Pattern Matching

Programmation
Fonctionnelle

Didier Verna
EPITA

Techniques
d'évaluation

Stricte

Lazy

Comparaison

Scoping

Blocs

Scoping

`f p1 p2 p3 ... = e1`

`f q1 q2 q3 ... = e2`

`...`

- **Équation utilisée** : la *première* offrant un matching entre les arguments et les formes paramétriques
- **Évaluation** :
 - ▶ Seuls les arguments nécessaires à la décision
 - ▶ Seuls les *morceaux* d'agrégats nécessaires



Évaluation *partielle* des arguments

Exemple 2

Programmation
Fonctionnelle

Didier Verna
EPITA

Techniques
d'évaluation

Stricte

Lazy

Comparaison

Scoping

Blocs

Scoping

```
sh :: [Int] -> [Int] -> Int
sh [] ys = 0
sh (x:xs) [] = 0
sh (x:xs) (y:ys) = x + y
```

```
sh [1..3] [5..8]
```

```
(1) => sh (1:[2..3]) [5..8]
```

```
(2) => sh (1:[2..3]) (5:[6..8])
```

```
(3) => 1 + 5
```

```
=> 6
```




Évaluation des guardes

C'est pareil

Programmation
Fonctionnelle

Didier Verna
EPITA

Techniques
d'évaluation

Stricte

Lazy

Comparaison

Scoping

Blocs

Scoping

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

max3 (2+3) (4-1) (3+9)

(1) => (2+3) >= (4-1) && (2+3) >= (3+9)

=> 5 >= (4-1) && 5 >= (3+9)

=> 5 >= 3 && 5 >= (3+9)

=> True && 5 >= (3+9)

=> True && 5 >= 12

=> True && False

=> False

(2) => 3 >= 5 && 3 >= 12

=> False && 3 >= 12

=> False

(3) => 12



Application

Newton-Raphson

Programmation
Fonctionnelle

Didier Verna
EPITA

Techniques
d'évaluation

Stricte

Lazy

Comparaison

Scoping

Blocs

Scoping

```
nrsqrt :: Float -> Float -> Float
nrsqrt x delta = nrfind x (nrlist x) delta
```

```
nrfind :: Float -> [Float] -> Float -> Float
nrfind x (yn:ys) delta
  | nrhappy yn x delta = yn
  | otherwise = nrfind x ys delta
```

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

```
nrlist :: Float -> [Float]
nrlist x = nrbuild 1.0 (nrnext x)
```

```
nrbuild :: Float -> (Float -> Float) -> [Float]
nrbuild yn f = yn : nrbuild (f yn) f
```

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



Évaluation stricte vs. lazy

Les vrai-faux arguments

Programmation
Fonctionnelle

Didier Verna
EPITA

Techniques
d'évaluation

Stricte

Lazy

Comparaison

Scoping

Blocs

Scoping

■ L'évaluation stricte évite la redondance de calculs

- ▶ **Vrai**, mais l'ordre normal aussi
(Cf. graphes d'évaluation)

■ L'évaluation lazy dispense d'opérateurs spéciaux

- ▶ **Vrai**, mais les macros (programmatiques) de Lisp rattrappent le coup... (ex. `unless`)

Haskell

```
ifnot :: Bool -> a -> a -> a  
ifnot test e1 e2 = if test then e2 else e1
```

Lisp

```
(defmacro ifnot (test e1 e2)  
  (list 'if test e2 e1))
```



Évaluation stricte vs. lazy

Les vrai-vrais arguments

Programmation
Fonctionnelle

Didier Verna
EPITA

Techniques
d'évaluation

Stricte

Lazy

Comparaison

Scoping

Blocs

Scoping

- **Théorème de Church-Rosser** : (Cf. λ -calcul)
Les deux méthodes sont équivalentes (donnent le même résultat) en fonctionnel pur.
- **Évaluation stricte** :
 - ▶ De l'intérieur vers l'extérieur ; de gauche à droite
 - ▶ seul utilisable en fonctionnel impur
(dépendance vis-à-vis de l'ordre d'évaluation)
- **Évaluation lazy** :
 - ▶ De l'extérieur vers l'intérieur ; de gauche à droite
 - ▶ abstractions supplémentaires
(ex. types infinis)



Un point sur le vocabulaire

strict, lazy, applicatif, normal. . .

Programmation
Fonctionnelle

Didier Verna
EPITA

Techniques
d'évaluation

Stricte

Lazy

Comparaison

Scoping

Blocs

Scoping

- **Ordre applicatif/ normal** : sémantique des langages
- **Strict** : se dit surtout d'une procédure / fonction
- **Lazy** : se dit surtout d'un évaluateur
- Dans un langage d'ordre applicatif, toutes les procédures sont strictes.
- Dans un langage d'ordre normal, toutes les procédures non primitives sont non strictes (puisque l'évaluateur est lazy), et les procédures primitives peuvent être strictes, ou pas.

Vous me suivez ?



Environnement global vs. local

L'environnement global est insuffisant

Programmation
Fonctionnelle

Didier Verna
EPITA

Techniques
d'évaluation

Stricte

Lazy

Comparaison

Scoping

Blocs

Scoping

■ Contextes locaux implicites :

- ▶ Arguments des fonctions (α -conversion, imbrication)

Lisp

```
(defun sq (x) (* x x))  
(defun sq (y) (* y y))
```

Lisp

```
(defun sq (x) (* x x))  
(defun f (x) (sq (/ 1 x)))
```

■ Contextes locaux explicites :

- ▶ Données locales
(éviter la redondance d'évaluation)
- ▶ Fonctions locales
(éviter la pollution des espaces de noms)



Structure de blocs

Où chercher une expression nommée

Programmation
Fonctionnelle

Didier Verna
EPITA

Techniques
d'évaluation

Stricte

Lazy

Comparaison

Scoping

Blocs

Scoping

- **Bloc :**
ensemble de liaisons [nom – expression]
- **Environnement d'évaluation :**
structure de blocs imbriqués
- **Variable liée :**
définie dans le contexte (bloc) local
- **Variable libre :**
non définie localement
- **Scoping :** capture d'une variable libre
(recherche d'une liaison dans le bloc le plus « proche »)

Remarque : la notion de « proximité » reste à définir. . .



Contextes locaux explicites

Trop d'imbrication tue l'imbrication

Programmation
Fonctionnelle

Didier Verna
EPITA

Techniques
d'évaluation

Stricte

Lazy

Comparaison

Scoping

Blocs

Scoping

Lisp

```
(defun f (x)
  (let ((a (* x x))
        (b (+ (* x x) 1)))
    (+ (/ a b) (/ b a))))
```

Haskell

```
f :: Float -> Float
f x = let a = x * x
      b = a + 1
      in a / b + b / a
```

Lisp

```
(defun f (x)
  (let* ((a (* x x))
         (b (+ a 1)))
    (+ (/ a b) (/ b a))))
```

Haskell

```
f :: Float -> Float
f x = a / b + b / a
      where a = x * x
            b = a + 1
```

- **Lisp** : pas de références mutuelles dans `let` (utiliser `let*`).
- **Haskell** : ordre des définitions sans importance



Contextes locaux fonctionnels

Rappel : les fonctions sont des objets de 1^{re} classe...

Programmation
Fonctionnelle

Didier Verna
EPITA

Techniques
d'évaluation

Stricte

Lazy

Comparaison

Scoping

Blocs

Scoping

- **Haskell** : `let` et `where` peuvent contenir des fonctions (y compris les déclarations de type).
- **Lisp** : utiliser `labels`

Lisp

```
(defun ssq (x y)
  (labels ((square (x) (* x x)))
    (+ (square x) (square y))))
```

Haskell

```
ssq :: Float -> Float -> Float
ssq x y = square x + square y
  where square x = x * x
```

- **Remarque** : référencement mutuel dans `label` possible



Pattern matching sur les déclarations

En Lisp comme en Haskell

Programmation
Fonctionnelle

Didier Verna
EPITA

Techniques
d'évaluation

Stricte

Lazy

Comparaison

Scoping

Blocs

Scoping

- **Haskell** : assignation par pattern matching
- **Lisp** : `destructuring-bind`
(formes correspondantes à des listes d'arguments de fonctions)

Lisp

```
(defun foo (l)  
  (destructuring-bind (a . b) l  
    (mapcar (lambda (x) (* x a)) b)))
```

Haskell

```
foo :: [Int] -> [Int]  
foo l = map (*a) b  
      where (a:b) = l
```



Portée des noms

Attention aux collisions

Programmation
Fonctionnelle

Didier Verna
EPITA

Techniques
d'évaluation

Stricte
Lazy
Comparaison

Scoping

Blocs

Scoping

- `let` lie plus fort les variables déjà liées

Lisp

```
(defvar x 5)
```

```
(defvar y  
  (+ (let ((x 3)) x)  
      x))  
      ;; y = 8  
      ;; x = 3  
      ;; x = 5
```

Haskell

```
x :: Int  
x = 5
```

```
y :: Int  
y = (let x = 3 in x) + x  
      — y = 8  
      — x = 3  
      — x = 5
```



Portée des noms (suite)

Attention aux collisions

Programmation
Fonctionnelle

Didier Verna
EPITA

Techniques
d'évaluation

Stricte
Lazy
Comparaison

Scoping

Blocs
Scoping

- **Lisp** : valeurs locales calculées à l'*extérieur* de `let` (pas en Haskell)
- `let` (Haskell) plus proche de `let*` (Lisp)

Lisp

```
(defvar x 2)

(defvar y
  (let ((x 3)           ;; x = 3
        (z (+ x 2)))    ;; z = 2 + 2
    (* x z)))           ;; y = 3 * 4
```

Lisp

```
(defvar x 2)

(defvar y
  (let* ((x 3)           ;; x = 3
         (z (+ x 2)))    ;; z = 3 + 2
    (* x z)))           ;; y = 3 * 5
```

Haskell

```
x :: Int
x = 2

y :: Int
y = let x = 3      — x = 3
     in z = x + 2  — z = 3 + 2
     in x * z      — y = 3 * 5
```



Formes de scoping

Recherche d'une liaison [nom – expression]

- **Lexical** : recherche dans l'environnement de *définition*
- **Dynamique** : recherche dans l'environnement d'*appel*

Lisp

```
(let ((x 10))  
  (defun foo () x))
```

```
(let ((x 20))  
  (foo)) ;; => 10
```

Lisp

```
(defparameter x 10)  
(defun foo () x)
```

```
(let ((x 20))  
  (foo)) ;; => 20
```

Programmation
Fonctionnelle

Didier Verna
EPITA

Techniques
d'évaluation

Stricte

Lazy

Comparaison

Scoping

Blocs

Scoping



Fermetures lexicales

AKA lexical closures

Programmation
Fonctionnelle

Didier Verna
EPITA

Techniques
d'évaluation

Stricte
Lazy
Comparaison

Scoping

Blocs

Scoping

■ Définition :

Combinaison entre une fonction et son environnement de définition (valeurs des variables libres au moment de la définition).

■ Intérêts :

- ▶ Opérations génériques par fonctions anonymes mapping, folding *etc.* : fermetures lexicales qui s'ignorent
- ▶ Création dynamique de fonctions à état local
- ▶ Encapsulation (portée restreinte)

■ Remarque : (Lisp) Environnement lexical *modifiable*



Intérêts des fermetures lexicales

Elles sont partout. . .

Programmation
Fonctionnelle

Didier Verna
EPITA

Techniques
d'évaluation

Stricte

Lazy

Comparaison

Scoping

Blocs

Scoping

■ Opérations génériques par fonctions anonymes :

Lisp

```
(defun list+ (lst n)
  (mapcar #'(lambda (x) (+ x n))
    lst))
```

Haskell

```
(+++) :: [Int] -> Int -> [Int]
(+++) lst n = map (\x -> x + n) lst
             — map (+n) lst
```

■ Création dynamique de fonctions à état local :

Lisp

```
(defun make-adder (n)
  #'(lambda (x) (+ x n)))
```

Haskell

```
makeAdder :: Int -> Int -> Int
makeAdder n = \x -> x + n
```

■ État local modifiable :

Lisp

```
(let ((cnt 0))
  (defun newtag () (incf cnt))
  (defun resettag () (setq cnt 0)))
```



Lisp : scoping dynamique

C'est mon choix (et je le partage)

Programmation
Fonctionnelle

Didier Verna
EPITA

Techniques
d'évaluation

Stricte

Lazy

Comparaison

Scoping

Blocs

Scoping

- Historiquement : première forme de scoping
- Depuis `scheme` : scoping lexical par défaut
- Common Lisp : scoping dynamique possible
 - ▶ Variables globales (`defvar`, `defparameter`)
 - ▶ Variables locales déclarées « spéciales »

```
(let ((x 10))  
  (defun foo () x))  
  
(let ((x 20))  
  (foo))      ;; => 10
```

```
(let ((x 10))  
  (defun foo ()  
    (declare (special x))  
    x))  
  
(let ((x 20))  
  (foo))      ;; => 20
```




Pour ou contre le scoping dynamique ?

Ça dépend...

Programmation
Fonctionnelle

Didier Verna
EPITA

Techniques
d'évaluation

Stricte
Lazy
Comparaison

Scoping

Blocs
Scoping

■ Avantages :

- ▶ Variables globales !
Options Emacs (`ex. case-fold-search` *etc.*)

■ Inconvénients :

- ▶ Énorme source de bugs très difficiles à pister
Problème de collision de noms (« name clash »)
- ▶ Le premier exemple de fonction d'ordre supérieur
donné par Mc Carthy était faux !



Le (mauvais) exemple de Mc Carthy

L'ancêtre de `mapcar`

Programmation
Fonctionnelle

Didier Verna
EPITA

Techniques
d'évaluation

Stricte
Lazy
Comparaison

Scoping

Blocs

Scoping

Lisp

```
(defmacro while (test &rest body)
  '(do () ((not ,test))
        ,@body))

(defun my-mapcar (func lst)
  (let (elt n)
    (while (setq elt (pop lst))
      (push (funcall func elt) n))
    (nreverse n)))

(defun list+ (lst n)
  (my-mapcar #'(lambda (x)
                  (declare (special n))
                  (+ x n)) ;; Barf !!
             lst))
```