

# Functional Approaches to Programming

## ~ Evaluation and Scoping ~

Didier Verna

EPITA / LRDE

didier@lrde.epita.fr



lrde/~didier



@didierverna



didier.verna



in/didierverna



# Outline



## Evaluation Techniques

Strict Evaluation / Applicative Order

Lazy Evaluation / Normal Order

Comparative Merits

## Scoping Forms

Block Structure / Locality

Lexical vs. Dynamic Scoping





# Plan



## Evaluation Techniques

Strict Evaluation / Applicative Order

Lazy Evaluation / Normal Order

Comparative Merits

## Scoping Forms



# Strict Evaluation (Lisp)

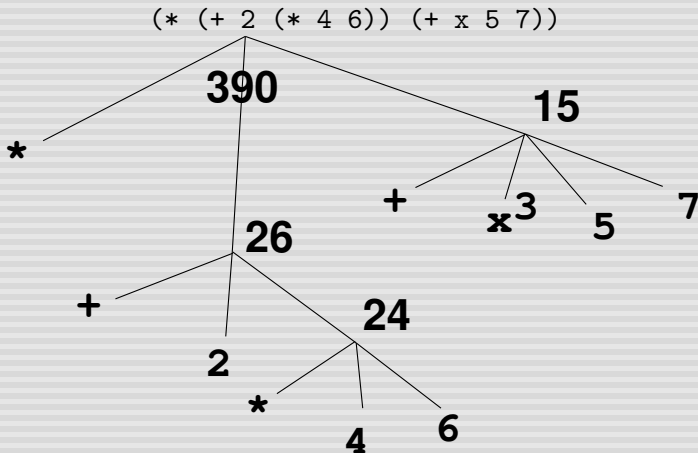
- ▶ A.k.a. “applicative order”
- ▶ **Process**
  1. Evaluate the sub-expressions from left to right
  2. Apply the operator to its arguments

*Recursive process (“tree accumulation”)*

- ▶ **Leaves:** literals / primitive operators
- ▶ **Environment**
  - ▶ Abstract expressions values
  - ▶ Special case: primitive operators



# Exemple



# Substitution Model

## ▶ For abstract expressions

- ▶ Identical to the previous case
- ▶ *Substitution* (preliminary step)  
*Replacement of formal parameters with their corresponding value*

## ▶ Remarks

- ▶ Theoretical model
- ▶ Substitution  $\Rightarrow$  local environment
- ▶ Complicated! Cf.  $\lambda$ -calcul,  $\alpha$ -conversion, name collision



# Example

```
(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 (* 5 2))  
(ssq 6 10)  
(+ (sq x) (sq y))  
(+ (sq 6) (sq 10))  
(+ (* x x) (sq 10))  
(+ (* 6 6) (sq 10))  
(+ 36 (sq 10))  
(+ 36 (* x x))  
(+ 36 (* 10 10))  
(+ 36 100)  
136
```



# Special Operators

- ▶ **Problem:** non strict idioms  
*Conditionals, boolean logic, etc.*
- ▶ **Solution:** special *primitive* operators  
*Particular evaluation technique*
- ▶ 25 in Lisp (`setq`, `if`, `quote`, `function`, *etc.*)
- ▶ Test: `(special-operator-p 'if)`





# Lazy Evaluation (Haskell)

- ▶ A.k.a. “normal order”
- ▶ **Process**
  - ▶ Same substitution model
  - ▶ But *on-demand* evaluation
- ▶ **Performance:** inefficient model  
*Work on an evaluation graph (“memoization”)*
- ▶ **Constraint:** purely functional only



# Example

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

```
6 * 6 + (5 * 2) * (5 * 2)
```

```
36 + (5 * 2) * (5 * 2)
```

```
36 + 10 * (5 * 2)
```

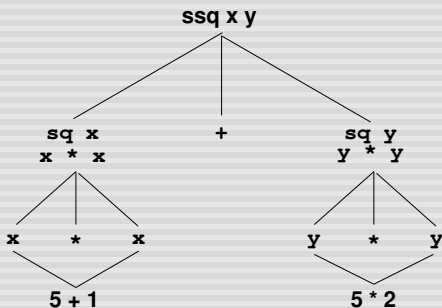
```
36 + 10 * 10
```

```
36 + 100
```

```
136
```



# Evaluation Graph / Memoization



$ssq\ (5 + 1)\ (5 * 2)$

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

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

$6 * 6 + 10 * 10$



# Equation Evaluation

f p1 p2 p3 ... = e1

f q1 q2 q3 ... = e2

...

- ▶ **Selected equation:** first “match”
- ▶ **Matching**
  - ▶ Only the arguments *necessary* to the decision process
  - ▶ Only the arguments *parts* necessary to the decision process



# Examples

## ► Minimal evaluation of the arguments

```
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
```

## ► Partial evaluation of the arguments

```
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
```



# Guards Evaluation

```
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
```



# nrsqrt IV, Apocalypse

## ▶ Reminder

### Lisp

```
(defun intlist (s)                ;; KO
  (cons s (intlist (1+ s))))
```

### Haskell

```
intlist :: Int -> [ Int ]      -- OK
intlist s = s : intlist (s + 1)
```

- ▶ **New view:** work directly on the (infinite) Newton-Raphson sequence

## Haskell

```
limit :: [ Float ] -> (Float -> Bool) -> Float
```

```
limit (y:ys) found
  | found y = y
  | otherwise = limit ys found
```

```
nrsqrt :: Float -> Float -> Float
```

```
nrsqrt x delta = limit (nrlist x 1) (nrfound x delta) -- partial application!
```

```
nrlist :: Float -> Float -> [ Float ]
```

```
nrlist x yn = yn : nrlist x ((yn + x / yn) / 2)
```

```
nrfound :: Float -> Float -> Float -> Bool
```

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

# Strict vs. Lazy Evaluation

- ▶ **Strict evaluation is more efficient**  
No computation redundancy, but *cf.* normal order + memoization
- ▶ **Lazy evaluation is more expressive**  
*E.g.* infinite data structures, but *cf.* Lisp macros

## 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))
```

- ▶ **No laziness without purity, no impurity without determinism**  
But strict evaluation is not the only deterministic form
- ▶ **Church-Rosser** (“global confluence”, *cf.*  $\lambda$ -calcul)  
Unicity of the canonical form, whatever the evaluation method  
(purely functional only)





# Strict vs. Lazy Evaluation

## ► Semantic diff quizz

### Lisp

```
(setq bar (sqrt (- (* 3 (+ 6 7)) 8)))
```

```
'(foo bar baz) ≠ (list foo bar baz)
```

```
(let* ((a (* x x)) ;; order important  
       (b (+ a 1)))  
  (+ (/ a b) (/ b a)))
```

### Haskell

```
bar = sqrt (3 * (6 + 7) - 8)
```

```
[foo, bar, baz]
```

```
let b = a + 1 -- order not important  
    a = x * x  
in a / b + b / a
```

# Pseudo-Lazyness in Imperative

## ► Conditionals, *etc.*

### C

```
if (something_that_turns_out_to_be_true)
{
    /* Computed */
}
else
{
    /* Not computed */
}
```





# Plan



Evaluation Techniques

Scoping Forms

Block Structure / Locality

Lexical vs. Dynamic Scoping



# Block Structure / Locality

- ▶ **Block:** set of  $\{name \rightarrow expression\}$  “bindings”
- ▶ **Evaluation environment:** nested blocks
- ▶ **Explicit vs. implicit blocks**
  - ▶ `let`, `where`, `flet`, *etc.*
  - ▶ Global declarations, function parameters
- ▶ **Theoretical problems** (*cf.*  $\alpha$ -conversion)

## Lisp

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

```
(defun sq (x) (* x x))  
(defun f (x) (sq (* 2 x)))
```

## Haskell

```
sq x = x * x  
sq y = y * y
```

```
sq x = x * x  
f x = sq (2 * x)
```

# Concept of Scoping

## ▶ Context

- ▶ *Bound* variable: defined in the local block
- ▶ *Free* variable: not locally defined

## ▶ Scoping

- ▶ Binding lookup in the “nearest” block
- ▶ But what do we mean by “near”?



# Scoping and Collision

## ► Local bindings take precedence

### Lisp

```
(defvar x 5)

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

### Haskell

```
x :: Int
x = 5

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

# Scoping and Collision

## ▶ But beware the language semantics!

- ▶ Lisp: local values computed *outside* of let

### Lisp

```
(defvar x 2)

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

- ▶ Nonsensical in Haskell (closer to Lisp's let\*)

### 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
     z = x + 2    -- z = 3 + 2
     in x * z     -- y = 3 * 5
```



## Scoping Forms

- ▶ **Reminder:** “closest” binding lookup  
*Deals with free variables only*
- ▶ **Lexical (static):** lookup in the *definition* environment
- ▶ **Dynamic:** lookup in the *calling* environment
- ▶ **Remark:** `let` does two different things at the same time!

### Lisp

```
(let ((x 10)).           ;; lexical scope
  (defun foo () x))

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

### Lisp

```
(defvar x 10)           ;; dynamic scope
(defun foo () x)

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



# Concept of Lexical Closure

## ▶ Definition

Combination of a function and its defining environment

*Values of free variables at definition time*

## ▶ Advantages

- ▶ Foundation of the *whole* higher-order machinery!
- ▶ Cf.  $\lambda$ -calcul / chapter 2



# Examples

## ▶ Functional arguments

### Lisp

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

### Haskell

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

## ▶ Functional Returns

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

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

## ▶ Locality + mutation

```
(let ((cnt 0))
  (defun get-uid () (incf cnt))
  (defun reset-uids () (setq cnt 0)))
```



# Dynamic Scoping (Lisp)

- ▶ Historical form in Lisp
- ▶ Since Scheme: lexical scoping
- ▶ Common Lisp: lexical scoping by default, dynamic possible
  - ▶ Global variables (`defvar`, *etc.*)
  - ▶ Local variables declared “special”

```
(let ((x 10)).           ;; lexical scope
  (defun foo () x))
```

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

```
(defvar x 10)           ;; dynamic scope
(defun foo () x)
```

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

```
(let ((x 10))
  (defun foo ()
    (declare (special x)) ;; dynamic scope
    x))
```

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



# Pro, or Anti Dynamic Scoping?

## ▶ Advantages

- ▶ Several paradigms (Context-Oriented Programming, *etc.*)
- ▶ Exception management (handlers)
- ▶ Global variables (*e.g.* user options, *cf.* Emacs)

```
(defvar *default-background* 'blue) ;; note the earmuffs!
```

```
(defun create-42-red-windows ()  
  (let ((*default-background* 'red))  
    (loop :repeat 42 :do (create-window))))
```

## ▶ Drawbacks

- ▶ Unsuitable to higher order
- ▶ Origin of very nasty bugs  
*Name collision problem*
- ▶ The very first example of higher order function from John McCarthy was wrong!



# McCarthy's (bad) example

## ► First historical mapping function

```
(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)) ;; <- name clash on n!!
    (reverse n)))

(defun list+ (lst n)
  (my-mapcar (lambda (x)
              (declare (special n)) ;; Do that and...
              (+ x n)) ;; ... barf, name clash on n!!
            lst))
```

