Relax, Program in Scheme

Ludovic Courtès
Service expérimentation & développement

8 November 2010

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
BORDEAUX - SUD-OUEST

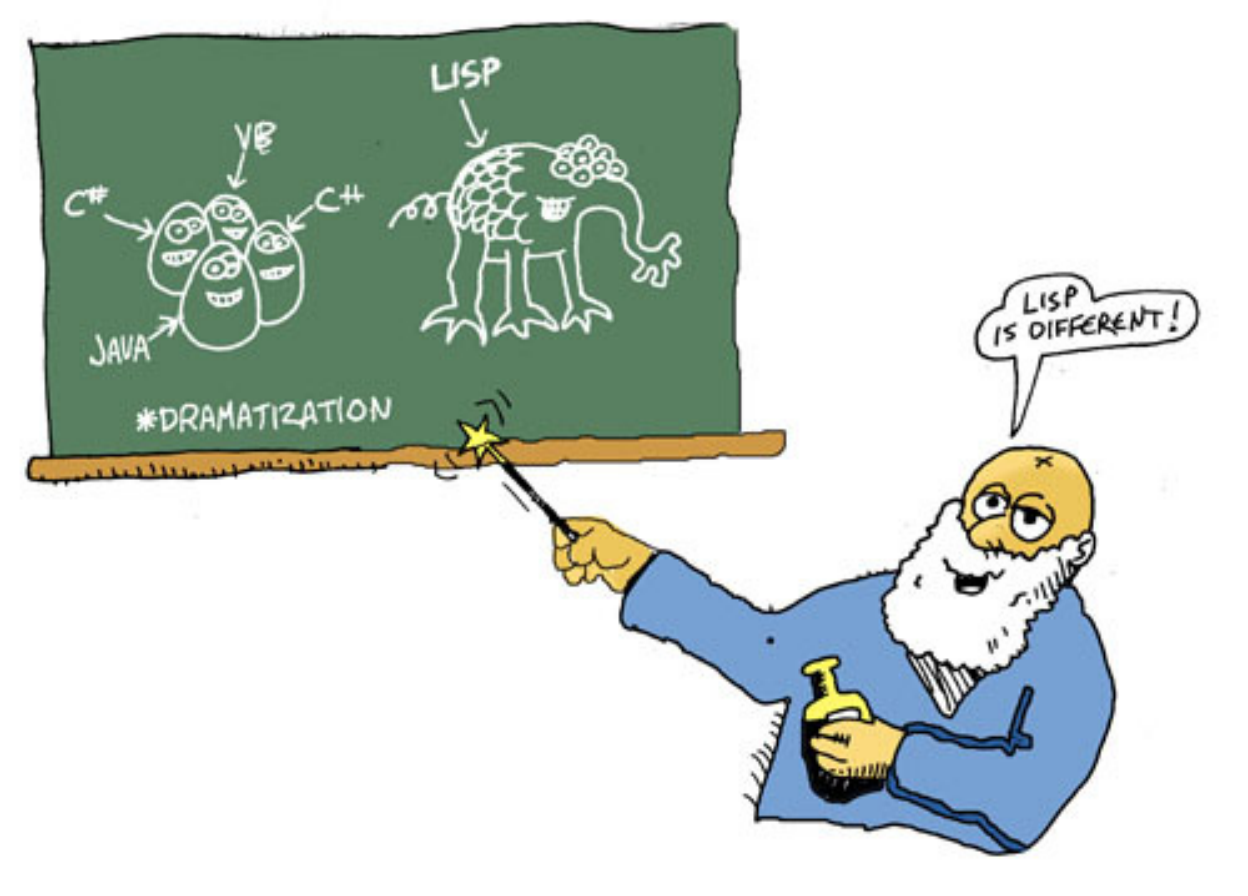
- **Introduction**
- Getting Started
- Functional Programming
- Bonuses
- Practice

The Perception of Lisp

Why Bother?

http://xkcd.com/297/

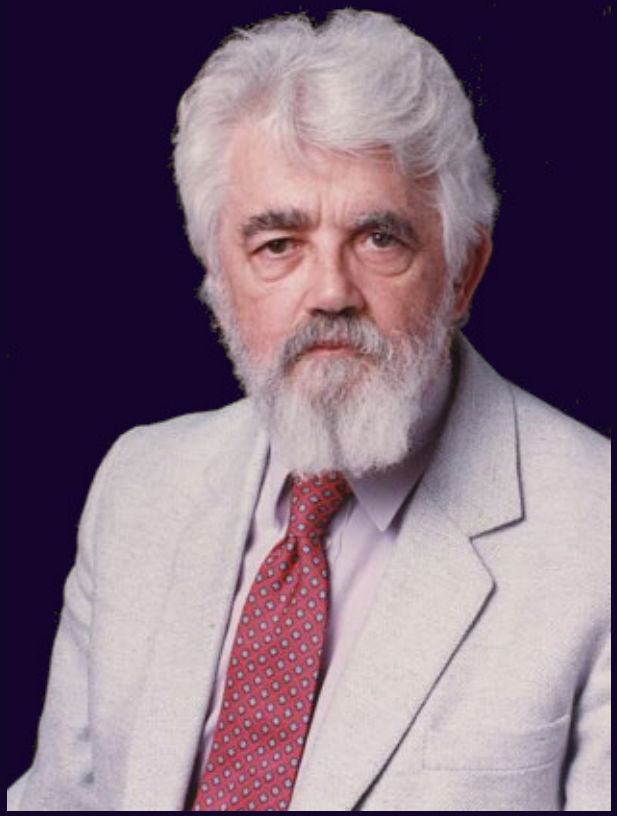
Why Lisp?



http://www.lisperati.com/casting.html

Calculus?

Scheme is a Modern Lisp



Scheme is Minimalist

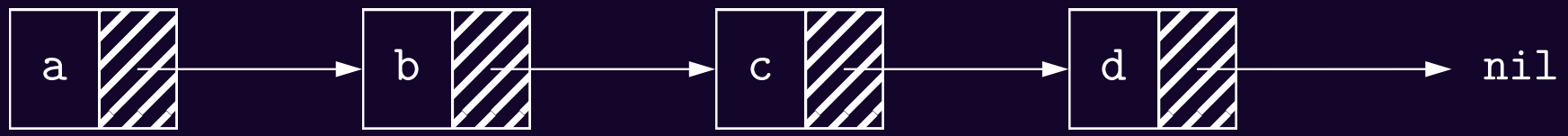
"Programming languages should be designed
not by piling feature on top of feature,
but by removing the weaknesses and restrictions
that make additional features appear necessary."

- Introduction
- **Getting Started**
  - Basic Data Types
  - Syntax (Or Lack Thereof)
  - Procedures
  - Procedure Application
- Functional Programming
- Bonuses
- Practice

Basic Data Types

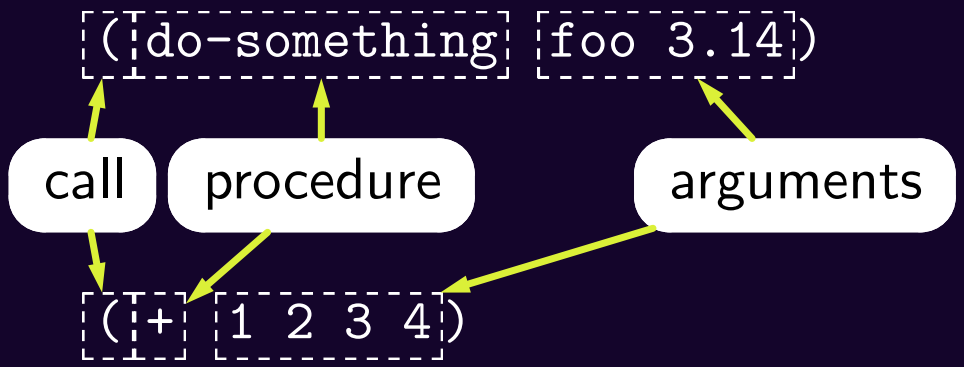
```
#f              ;; false
#t              ;; true
1               ;; integer
3.14            ;; floating point number
2/3             ;; rational
"hello!"        ;; string
call/cc         ;; symbol
another-one     ;; symbol
#(1 2 a b)      ;; vector
```

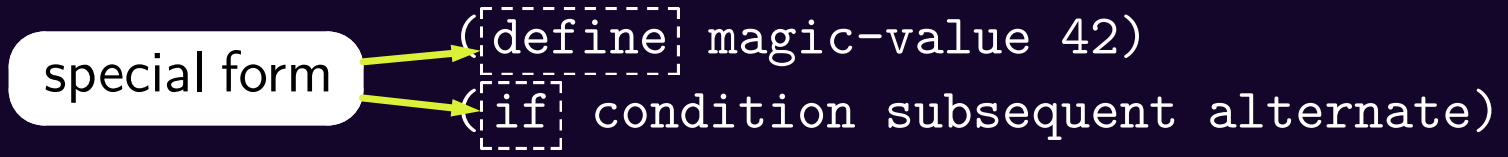
Pairs & Lists



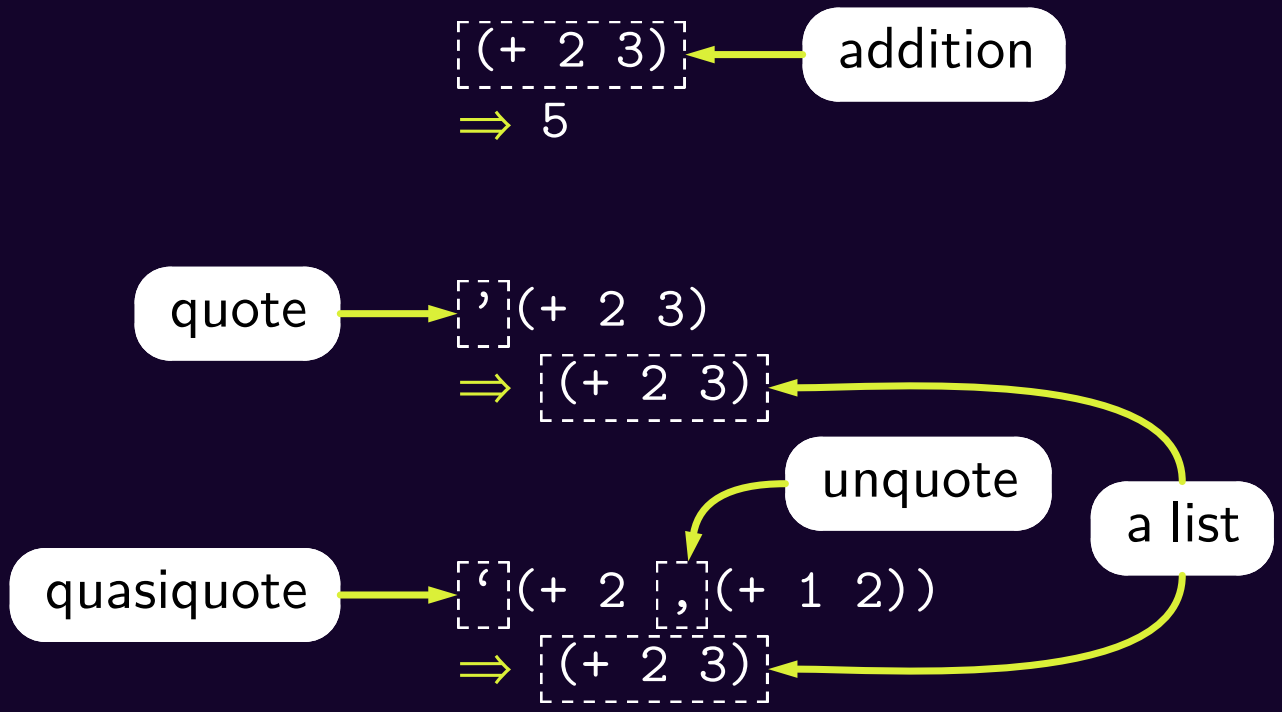

```
(a . (b . (c . d)))  ≡  (a b c d)
```

Syntax (Or Lack Thereof)





Quotes & Co.



Expressions

```
(if (> 2 3) 'yes 'no)
⇒ no

(pair? '(a b c))
⇒ #t

(third '(a b c))
⇒ c

(let ((p '(a . b))) (cons (car p) (cdr p)))
⇒ (a . b)
```

Procedures

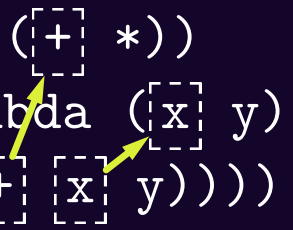
```
(lambda (x y)  ;; number -> number -> number
  (+ x y))

(lambda (x y)  ;; string -> string -> string
  (string-append x y))

(lambda (f g) ;; proc -> proc -> proc
  (lambda (x)
    (f (g x))))
```

Lexical Scope

```
(define f
  (let ((x 42)
        (+ *))
    (lambda (x y)
      (+ x y))))
```



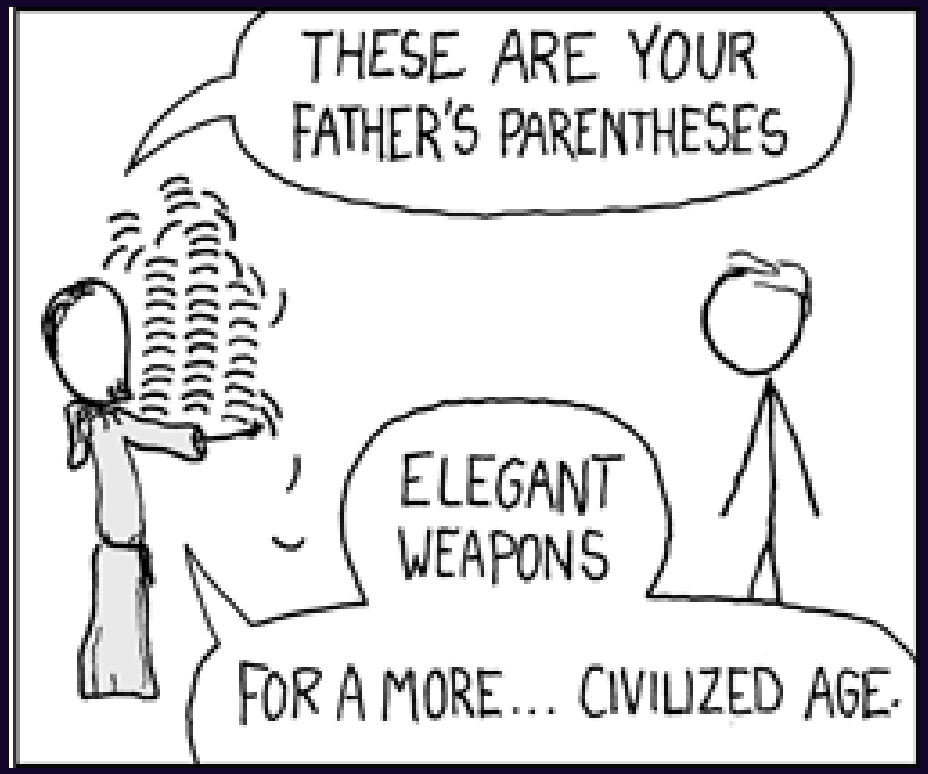
```
(f 2 3)
⇒ 6
```

Procedure Application

```
((lambda (x) (+ x 2)) 5)
⇒ 7

(define add2 (lambda (x) (+ x 2)))
(add2 40)
⇒ 42

(apply * '(1 2 3))
⇒ 6
```



http://xkcd.com/297/

- Introduction
- Getting Started
- **Functional Programming**
  - Referential Transparency
  - Higher-Order Functions
  - Pattern Matching
- Bonuses
- Practice

Referential Transparency

"An expression is *referentially transparent* if it can be replaced with its value without changing the program."

Referential Transparency in Maths

$$f(x,y) = x + y$$

$$g(x,y) = (f(x,y))^2$$

$f(x,y)$ can be substituted by its value:

$$g(x,y) = (x+y)^2 = x^2 + y^2 + 2xy$$

Referential Transparency in /bin/sh

```
$ printf "apple automn\ntomato summer\norange winter\n" | \
  cut -f 1 -d ' ' | \
  sort | uniq | \
  tr '\n' ' '
⇒ apple orange tomato

$ printf "apple\ntomato\norange" | \
  sort | uniq | \
  tr '\n' ' '
⇒ apple orange tomato

$ printf "apple\norange\ntomato" | \
  tr '\n' ' '
⇒ apple orange tomato
```

Referential Opaqueness vs. Transparency

```
int
foo (int y)
{
  int x = y;

  x += 2 * x + 4;
  x += (++y) - 5;
  x -= 4 * (y - 1);

  return x;  /* ??? */
}
```

```
(define (foo y)
  (let ((x y))
    (+ (+ x (+ 4 (* 2 x)))
       (+ y 1 -5)
       (- (* 4 y)))))
≡
(define (foo x)
  (+ (+ (* 3 x) 4)
     (- x 4)
     (- (* 4 x))))
≡
(define (foo x) 0)
```

Referential Transparency & Optimization

```
(define (frob lst new)
  ;; O(n)
  (reverse (append (reverse lst) (list new))))
≡
(define (frob lst new)
  (reverse (reverse (cons new list))))
≡
(define (frob lst new)
  ;; O(1)
  (cons new lst))
≡
(define frob cons)
```

Iteration Through Recursion

```
(define (reverse lst)
  (if (null? lst)
    lst
    (append
      (reverse (cdr lst))
      (list (car lst)))))

(reverse '(1 2 3 4))
⇒ (4 3 2 1)
```

Referential Transparency Helps!

- reasoning
- static code analysis
- common sub-expression elimination
- automatic parallelization
- ...

Higher-Order Functions

```
(define (make-adder c) (lambda (x) (+ x c)))
((make-adder 40) 2)
⇒ 42

(define add1 (make-adder 1))
(add1 -1)
⇒ 0
```

Higher-Order Functions

```
(define (compose f g)
  (lambda (x) (f (g x))))

((compose add1 add2) 0)
⇒ 3
```

Mapping Lists to Other Lists

```
(map add1 '(0 1 2 3 4))
⇒ (1 2 3 4 5)

(map (lambda (x) (expt x 2)) '(0 1 2))
⇒ (0 1 4)

(map + '(1 2 3) '(-1 -2 -3))
⇒ (0 0 0)
```

Filtering Lists

```
(procedure? odd?)
⇒ #t

(filter odd? '(1 2 3 4))
⇒ (1 3)

(filter even? '(1 2 3 4))
⇒ (2 4)
```

List Iteration

```
(define (count pred lst)
  (fold (lambda (element
                 total)
          (if (pred element)
            (+ total 1)
            total))
        0
        lst))

(count odd? '(1 2 3 4 5))
⇒ 3
(count even? '(1 2 3 4 5))
⇒ 2
```

```
(count #<procedure odd?> (1 2 …))
(fold #<procedure _> 0 (1 2 …))
|  (#<procedure _> 1 0)
|  |  (odd? 1)
|  |  #t
|  1
|  (#<procedure _> 2 1)
|  |  (odd? 2)
|  |  #f
|  1
|  (#<procedure _> 3 1)
|  |  (odd? 3)
|  |  #t
|  2
…
```

Recursive functions are nice, but...

```
(define (fact n)
  (if (= n 0)
    1
    (* n (fact (- n 1)))))

(fact 3)
⇒ 6

(fact 20000) ;; an important number
=| ERROR: Stack overflow
```

Tail Call Optimization To The Rescue!

```
(define (fact n)
  (if (= n 0)
      1
      (* n (fact (- n 1)))))
```

consumes stack space

```
(define (fact* n)
  (define (fact-aux n result)
    (if (= n 0)
        result
        (fact-aux (- n 1) (* result n))))
  (fact-aux n 1))
```

tail call ⇒ goto

"λ, the Ultimate Goto", Guy Steele, 1977

Pattern Matching

```
(define lst '(if (= x 0) 'zero x))

(match lst
  ((elt …)
   (list 'it-is-a-list! elt))
  ((? string?)
   'it-is-a-string))
⇒ (it-is-a-list! (if (= x 0) 'zero x))

(match lst (('if condition _ _) condition))
⇒ (= x 0)
(match lst
       (('if ('= _ (and n (? number?))) _ _)
        n))
⇒ 0
```

pattern that matches a list

pattern that matches a string

Pattern Matching & XML

```
(define album
  (xml->sxml
    (open-input-string
      (string-append
        "<album title=\"Yeah!\" artist=\"John Smith\">"
        "<track num=\"1\" title=\"Chbouib\"/>"
        "<track num=\"2\" title=\"Shbweeb\"/>"
        "</album>"))))
```

Pattern Matching & XML

```
(define (album->html tree)
  (sxml-match
    tree
    ((*TOP* ,body …)
     `(html (body ,@(append-map album->html body))))
    ((album (@ (title ,t) (artist ,a)) ,body …)
     `((h1 ,(format #f "‘~a’ by ~a" t a))
       (ul ,@(map (lambda (item) `(li ,(album->html item)))
                  body))))
    ((track (@ (num ,n) (title ,t)))
     (format #f "~2d. ~a" (string->number n) t))))
```

Pattern Matching & XML

```
(with-output-to-string
  (lambda () (sxml->xml (album->html album))))
```

⇒ `"<html><body><h1>‘Yeah!’ by John Smith</h1><ul><li> 1. Chbouib</li><li> 2. Shbweeb</li></ul></body></html>"`

- Introduction
- Getting Started
- Functional Programming
- **Bonuses**
  - The Numerical Tower
  - Homoiconicity
  - Hygienic Macros
- Practice

The Tower of Numbers

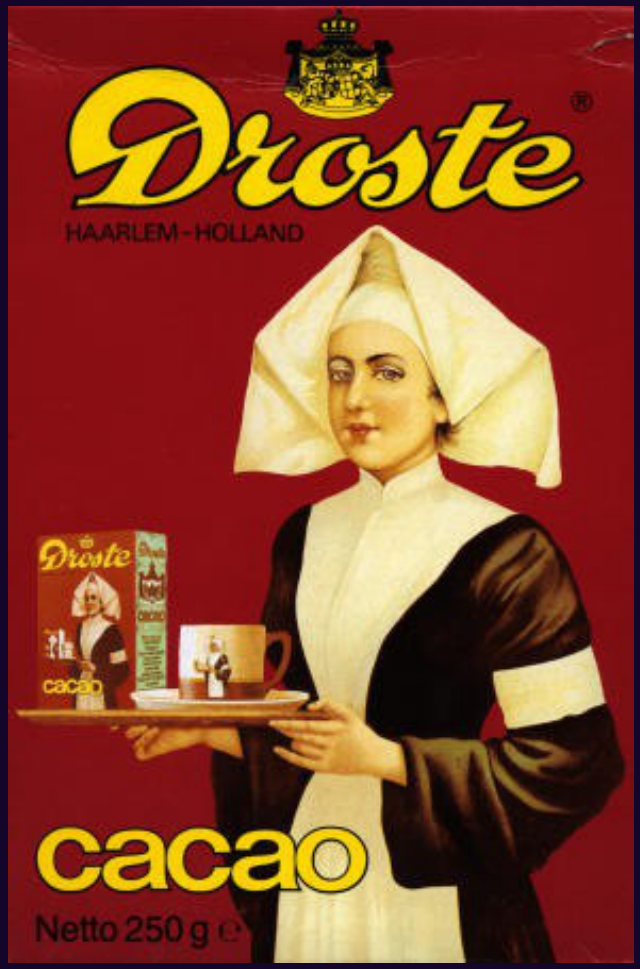
The Numerical Tower

```
(integer? 1)
⇒ #t
(expt 12 34)
⇒ 4922235242952026704037113243122008064
(integer? (expt 12 34))
⇒ #t
(+ (/ 2 3) (/ 1 4))
⇒ 11/12
(rational? 2/3)
⇒ #t
(exact->inexact 2/3)
⇒ 0.6666666666666667
(sqrt -16)
⇒ 0.0+4.0i
```

Homoiconicity

```
(define lst
  '(define lst '(define lst '())))

(eval lst
      (interaction-environment))
lst
⇒ (define lst '())
```



Hygienic Macros

http://www.inpes.fr/grippeAH1N1/communication.html

What's With Macro Hygiene?

```
#define swap(x, y)                              \
  {                                             \
    int z = x;                                  \
    x = y;                                      \
    y = z;                                      \
  }

int
foo (int x, int y, int z)
{
  if (y > z)
    swap (y, z);

  /* … */
}
```

What's With Macro Hygiene?

```
#define swap(x, y)                                  \
  {                                                 \
    int z = x;                                      \
    x = y;                                          \
    y = z;                                          \
  }

int
foo (int x, int y, int z)
{
  if (y > z)
    { int z = y; y = z; z = z; };  /* !!! */

  /* … */
}
```

Macros With Hygiene

1. avoid name clashes
2. resolve macro references local to their definition context

Hygienic Macros By Example

```
(define-syntax swap!
  (syntax-rules ()
    ((_ x y)
     (let ((z x)) (set! x y) (set! y z)))))

(macro-expand
  '(let ((y 1) (z 2)) (swap! y z)))
⇒ (let ((#{y\ 3546}# 1) (#{z\ 3547}# 2))
  (let ((#{z\ 3550}# #{y\ 3546}#))
    (begin
      (set! #{y\ 3546}# #{z\ 3547}#)
      (set! #{z\ 3547}# #{z\ 3550}#))))
(let ((y 1) (z 2)) (swap! y z) (list y z))
⇒ (2 1)
```

Hygienic Macros By Example

```
(define-syntax plus
  (syntax-rules () ((_ x y) (+ x y))))

(macro-expand '(let ((+ /)) (plus 1 2)))
⇒ (let ((#{+\ 3589}# /)) (+ 1 2))

(let ((+ *)) (plus 1 2))
⇒ 3
```

The Minimalist's Holly Grail

```
(define-syntax and
  (syntax-rules ()
    ((_) #t)
    ((_ x) x)
    ((_ x y …) (if x (and y …) #f))))

(macro-expand '(and 1 2 3 4))
⇒ (if 1 (if 2 (if 3 4 #f) #f) #f)

(define-syntax or
  (syntax-rules ()
    ((_) #f)
    ((_ x) x)
    ((_ x y …) (let ((t x)) (if t t (or y …))))))
```

Macro-Generating Macros

```
(define-syntax define-inline
  (syntax-rules ()
    ((_ (name args …) body …)
     (define-syntax name
       (syntax-rules ()
         ((_ args …) body …))))))

(define-inline (inline-me x y) (+ x y))

(macro-expand '(inline-me foo bar))
⇒ (+ foo bar)
(inline-me 2 3)
⇒ 5
```

Arbitrary Computations in Macros

```
(define (fact n)
  (let loop ((n n) (f 1))
    (if (= n 0) f (loop (- n 1) (* f n)))))
(define-syntax fact*
  (lambda (stx)
    (syntax-case stx ()
      ((_ n)
       (fact (syntax->datum (syntax n)))))))

(macro-expand '(fact 17))
⇒ (fact 17)
(macro-expand '(fact* 17))
⇒ 355687428096000
```

Define your own language!

`syntax-rules`!

- Introduction
- Getting Started
- Functional Programming
- Bonuses
- **Practice**
  - "Standards"
  - Implementations
  - Performance

"Standards"

minimalist

- R1RS (1978) ... R5RS (1998)
- 50 page spec, consensual

practical

- 100+ Scheme Requests for Implementation (SRFIs)
- R6RS (2007) $\Rightarrow 90 + 71$ page spec, controversial

Implementations

native compilers

- MIT/GNU Scheme, Larceny, Ikarus
- *via* C: **Bigloo** (INRIA Sophia), Chicken, Gambit, Stalin
- JIT: Racket

bytecode interpreters etc.

- GNU Guile, Gauche, Scheme48, Kawa, Ypsilon, etc.

Performance

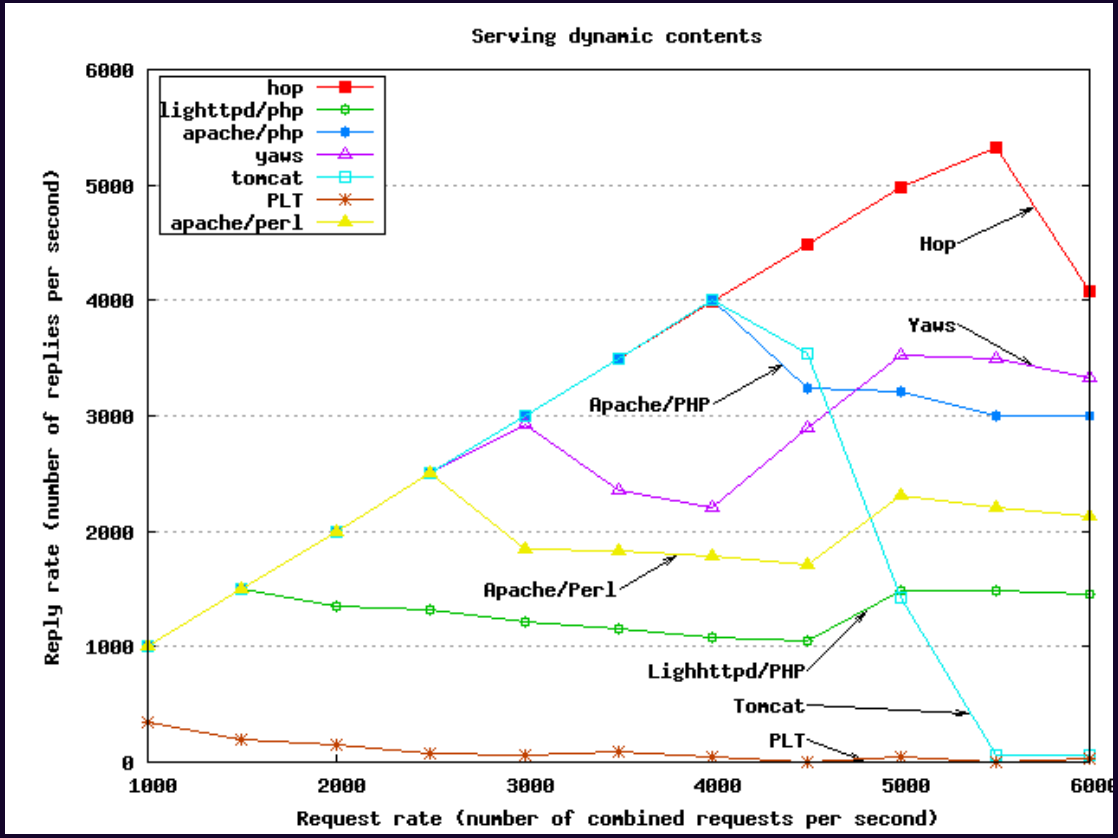
```
(module odd)

;; Not exported.
(define (odd? x)
  (define (even? n)
    (if (= n 0)
        #t
        (odd? (- n 1))))
  (define (odd? n)
    (if (= n 0)
        #f
        (even? (- n 1))))
  (odd? x))

;; Only use is with fixnum.
(odd? 77)
```

```
/* bigloo -O2 -cgen */
bool_t BGl_odd (long x)
{
  long a, b;
  b = x;
inner_odd:
  if ((b == ((long) 0)))
    return ((bool_t) 0);
  else { /* tail call to 'even?' */
    a = (b - ((long) 1));
    if ((a == ((long) 0)))
      return ((bool_t) 1);
    else {
      long c;
      c = (a - ((long) 1));
      b = c;
      goto inner_odd;
    }
  }
}
```

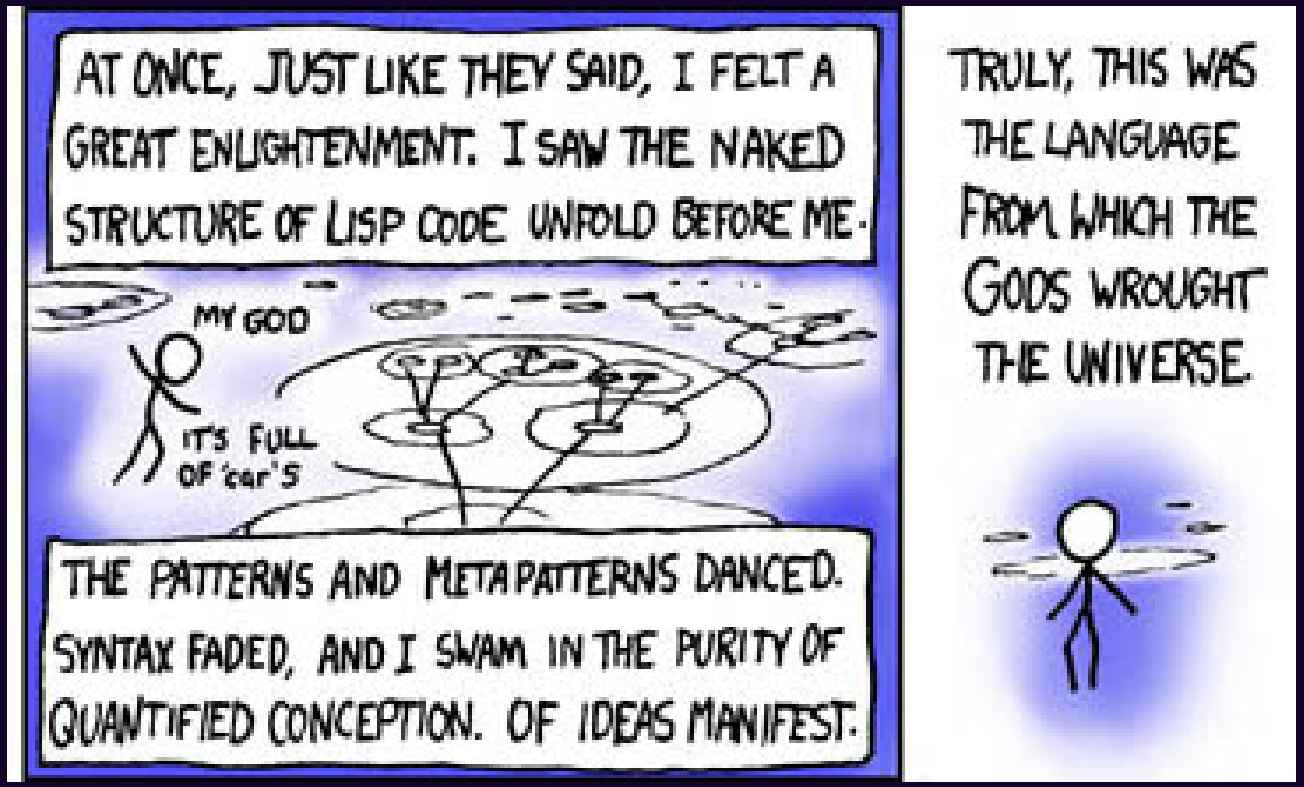
Performance of a Web Server in Bigloo Scheme



http://hop.inria.fr/

"Hop, a Fast Server for the Diffuse Web", Serrano, 2009

Enlightenment!



http://xkcd.com/224/

Summary

Scheme is good for you!

Summary

- functional programming eases **reasoning**
- Scheme has **simple & clean semantics**
- macros for the definition of **domain-specific languages**
- lots of **implementations + libs that rock!**
- performance bottleneck is (mostly) between keyboard and chair

ludovic.courtes@inria.fr

http://schemers.org/

http://sed.bordeaux.inria.fr/

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
BORDEAUX - SUD-OUEST

Made with Skribilo on GNU Guile 1.9.13.11-3a1a8-dirty; typeset with Lout.