deptinfo.unice.fr/~roy/PANORAMA.pdf

(Apprendre à) Programmer avec Scheme

MCours.com

Pourquoi Scheme ?

qualités pédagogiques

- Syntaxe minimale
- Environnement intégré
- Multi-paradigmes
- Données complexes (listes, arbres, vecteurs, images) immédiatement disponibles
- Peu répandu :-)

qualités théoriques

- Syntaxe minimale
- Auto-référent et
- facile à analyser
- Bases : λ-calcul
- Sémantique propre

Fonctionnel
Impératif
Par objets
Strict/Paresseux
etc.

Points d'histoire

- Origine : LISP, (McCarthy, MIT, 1958)
- Motivation : λ-calcul (fondements) + IA (symbolique)
- Crise vers 1970. Liaison dynamique ou statique ?

  Emacs-Lisp Pascal, C
- Scheme fait (avec Common-Lisp) le choix statique.

  Enseignement & recherche Industrie
- Scheme [norme R⁶RS, 2007] devient de plus en plus :
  - un outil d'enseignement
  - un laboratoire de langages
  - un langage de scripts

Quelques pôles d'enseignement

6.001 - Structure and Interpretation of Computer Programs
Fall 2007 at MIT

PROGRAMMATION RÉCURSIVE
(EN SCHEME)

Anne Brygoo
Titou Durand
Maryse Pelletier
Christian Queinnec
Michèle Soria

DUNOD

Paris VI Jussieu

L'équipe de C. Queinnec

PLT-Scheme

Programming Language Team

(the knights of λ-calculus)

Brown, Northeastern, Utah, Chicago...

Les images sont des liens vers des pages Web !...

MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Course # 6.946
Classical Mechanics: A Computational Approach

```
(define ((L-free-particle mass) local)
  (let ((v (velocity local)))
    (* 1/2 mass (dot-product v v))))
```

$$L(t, x, v) = \frac{1}{2} m(v \cdot v)$$

The advantage of Scheme over other languages is that the manipulation of procedures that implement mathematical functions is easier and more natural in Scheme. Indeed, many theorems of mechanics are directly representable as Scheme programs.

...et de recherche

HOP is the winner of the ACM MULTIMEDIA'07 !

Open Source Software Competition

Hop, a Language for Programming the Web 2.0

Manuel Serrano

Inria Sophia Antipolis, Projet Mimosa

HOP is a new Software Development Kit for the Web 2.0. It relies a new higher-order language for programming interactive web applications such as web agendas, web galleries, music players, etc. HOP can be viewed as a replacement for traditional graphical toolkits. HOP is implemented as a Web broker, i.e., a Web server that may act indifferently as a regular Web server or Web proxy.

Expressions préfixées

- Oui, il faut s'y habituer un peu...

a + b + 2	→	`(+ a b 2)`
a + 3b + 5	→	`(+ a (* 3 b) 5)`
sin(ωt+φ)	→	`(sin (+ (* ω t) φ))`
si x>0 alors 3 sinon y+1	→	`(if (> x 0) 3 (+ y 1))`
(x,y) ⟼ x + sin y	→	`(λ (x y) (+ x (sin y)))`
f o g	→	`(compose f g)`

- Avantages : aucune ambigüité, faciles à analyser.

Arithmétique

- Précision "infinie" sur les entiers :

```
(define fac          ; le style puriste
  (λ (n) (if (= n 0) 1 (* n (fac (- n 1))))))
```

```
(define (fac n)  ; le style MIT
  (if (= n 0) 1 (* n (fac (- n 1)))))
```

```
> (fac 40)
815915283247897734345611269596115894272000000000
> (time (even? (fac 10000)))    ; en interprété
time : 0.3 sec
#t
```

• Calcul de a^b mod n

$a^0 = 1$
$a^b = (a^2)^{b/2}$ si b pair
$a^b = a*(a^2)^{(b-1)/2}$ sinon

```
(define (exptmod a b n)
  (cond ((= b 0) 1)
        ((even? b) (exptmod (modulo (sqr a) n) (quotient b 2) n))
        (else (modulo (* a (exptmod (sqr a) (quotient b 2) n)) n))))
```

```
> (time (exptmod 123456789 123456789 654))
time: 0 sec
477
```

$123456789^{123456789}$ [654]

• Génération de grands nombres premiers

$$n \text{ premier} \Rightarrow a^{n-1} \equiv 1 \ [n] \quad \forall a,\ a \wedge n = 1$$

La réciproque est fausse mais "presque" vraie...
D'où un test de pseudo-primalité.

```
(define (random-base n)          ; retourne a ∈ [2,n-1] premier avec n
  (let ((a (+ 2 (srandom (- n 2)))))
    (if (= 1 (gcd a n)) a (random-base n))))
```

- La fonction ci-dessus est ITERATIVE [pas de pile] car l'appel récursif est en position terminale ! Cette optimisation de la récurrence est garantie...

```
> (random-base 10)
3
> (random-base 10)
7
```

```
> (random-base 10)
9
> (random-base 10)
7
```

- La fonction exptmod de la page précédente n'était pas itérative ! Bof.

```
(define (fermat-premier? n)          ; presque certainement...
  (define (ok? a)                    ; n passe-t-il le test en base a ?
    (= (exptmod a (- n 1) n) 1))
  (or (= n 2)
      (= n 3)
      (and (>= n 2) (ok? 2) (ok? 3) (ok? (random-base n)))))


(define (ppp>=  n)                   ; le + petit premier ≥ n (n grand)
  (define (iter n)
    (if (fermat-premier? n)
        n
        (iter (+ n 2))))
  (iter (if (odd? n) n (+ n 1))))
```

une boucle locale !

```
> (ppp>= (expt 10 50))
100000000000000000000000000000000000000000000000151
```

Les Listes

- (a b c d) est un appel de fonction : $a(b,c,d)$
- '(a b c d) est une donnée littérale : une liste.

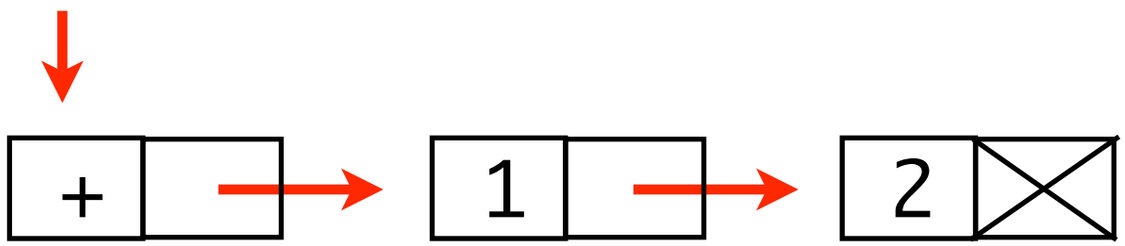
```
> (+ 1 2)
3
```

```
> '(+ 1 2)
(+ 1 2)
```

- Le premier élément est le CAR, le reste le CDR.

```
> (car '(+ 1 2))
+
```

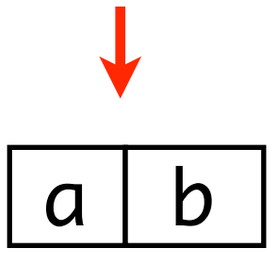
```
> (cdr '(+ 1 2))
(1 2)
```



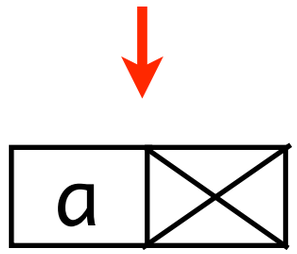
(define first car) *(define rest cdr)*

- La fonction (cons a b) construit une cellule :

(cons a b)



(cons a '())



$(car\ (cons\ a\ b)) \equiv a$

$(cdr\ (cons\ a\ b)) \equiv b$

```
(define L '(+ 1 2))
(define L (cons '+ (cons 1 (cons 2 '()))))
```

- Et oui, un texte Scheme est une liste :

Programmes == Données !

- Représentation d'un polynôme :

```
(define P '((2 0) (-5 1) (3 12)))      ; 2-5x+3x^12
```

- Tri par insertion :

$c_n = c_{n-1} + O(n)$

$c_n = O(n^2)$

```
(define (tri-ins L)
  (if (null? L)
      L
      (insertion (first L) (tri-ins (rest L)))))

(define (insertion x LT)
  (cond ((null? LT) (list x))
        ((≤ x (first LT)) (cons x LT))
        (else (cons (first LT)
                    (insertion x (rest LT))))))
```

- Analyse de complexité facilitée.

Arbres et calcul formel

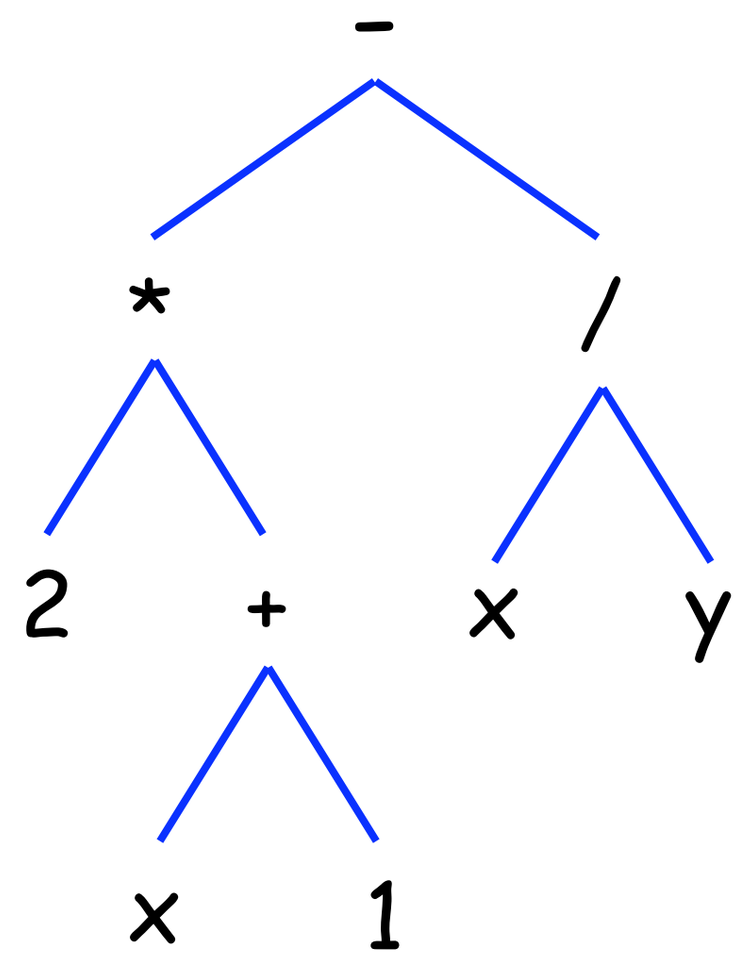
Analyser,
transformer,
compiler !

2*(x+1)-x/y

une feuille

```
(- (* 2 (+ x 1)) (/ x y))
```

racine — fils gauche — fils droit



- Implémentation du type abstrait par une liste

```
(define (feuille? obj)
  (or (number? obj)
      (and (symbol? obj) (not (operateur? obj)))))

(define (opérateur? obj)
  (member obj '(+ * - /)))
```

```
(define (fg A)
  (if (feuille? A)
      (error "pas de fg pour une feuille" A)
      (second A)))

(define (arbre r Ag Ad)
  (list r Ag Ad))
```

etc.

- Dérivation d'un arbre

```
(define (diff+- A v)                    ; A = (± Ag Ad)
  (arbre (racine A) (diff (fg A) v) (diff (fd A) v)))


(define (diff* A v)                     ; A = (* Ag Ad)
  (arbre '+
         (arbre '* (diff (fg A) v) (fd A))
         (arbre '* (fg A) (diff (fd A) v))))
```

```
> (diff* '(* y (+ x 1)) 'x)
(+ (* 0 (+ x 1)) (* y (+ 1 0)))
```

à simplifier !

- Simplification d'un arbre *en présence d'inconnues*

```
(define (simplif+ A)                    ; A = (+ Ag Ad)
  (let ((A1 (simplif (fg A))) (A2 (simplif (fd A))))
    (cond ((and (number? A1) (number? A2)) (+ A1 A2))
          ((equal? 0 A1) A2)
          ((equal? 0 A2) A1)
          (else (arbre '+ A1 A2)))))
```

```
> (simplif+ '(+ (+ x 0) (+ 2 3)))
(+ x 5)
```

- On peut alors simplifier la dérivée. Puis calculer la série de Taylor à coeffs. exacts, etc.

```
> (taylor '(/ 1 (cos x)) 'x 0 6)
(1 0 1/2 0 5/24 0)
```

Et Taylor n'attendra pas...

Compilation d'un arbre conditionnel

```
(if (> x 0) (+ x y) (- y 5))
```

```
((push x)
 (push 0)
 (call >)
 (brf e1)
```

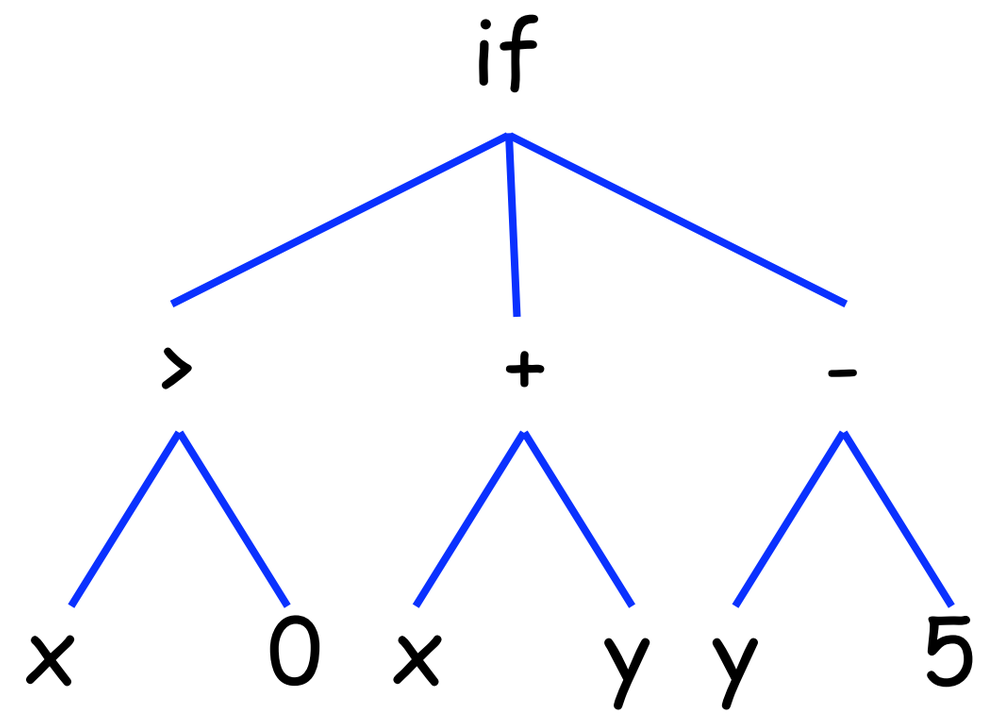
Compilation du fg

```
 (push x)
 (push y)
 (call +)
```

Compilation du fd

```
 (jmp e2)
 e1
 (push y)
 (push 5)
 (call -)
 e2)
```

Compilation du fdd



puis on interprète le code produit sur une machine à pile...

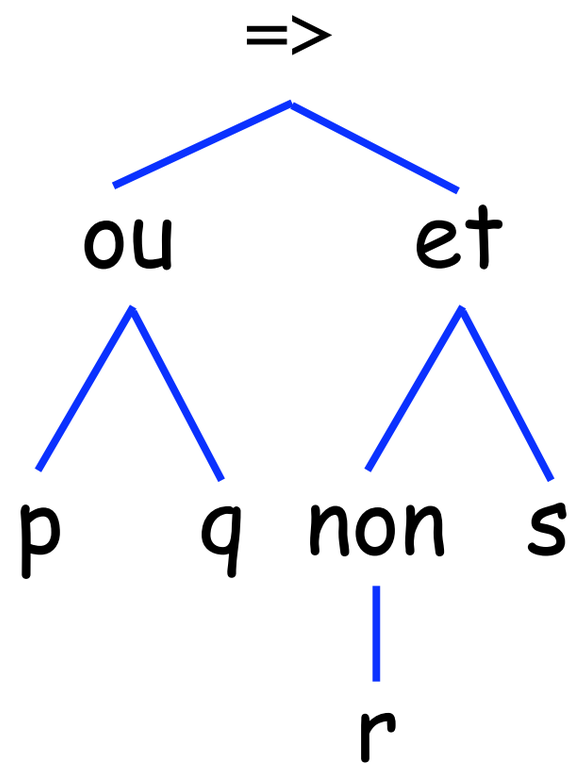
- Le temps du défi : décompiler !

```
((push 2) (push x) (push 0) (call =) (brf
etiq78) (push x) (push 1) (call +) (jmp
etiq79) etiq78 (push y) (push x) (push 2)
(call -) (call *) etiq79 (push y) (push 3)
(call -) (call >) (brf etiq76) (push x) (push
x) (push 1) (call -) (push 0) (call >) (brf
etiq80) (push y) (push z) (call +) (jmp
etiq81) etiq80 (push 5) etiq81 (call *) (jmp
etiq77) etiq76 (push y) (push z) (call *)
(push y) (call +) etiq77 (call +))
```

Des Arbres et de la Logique

- L'algorithme de Wang. Démonstration automatique en logique des propositions.

$p \vee q \Rightarrow \neg r \wedge s$



- Une formule est encore un arbre, représenté par une liste :

```
((p ou q) => ((non r) et s))
```

arg1 connecteur arg2

- On cherche à savoir si une formule est valide :

```
> (valide? '((p et (p => q)) => (p et q)))
#t
> (valide? '(#f => p))
#t
> (valide? '(p => (p et q)))
#f
```

- Un séquent est de la forme $H_1 \wedge \ldots \wedge H_n \rightarrow C_1 \vee \ldots \vee C_k$

- Représentation d'un séquent par une liste :

$((H_1 \ \ldots \ H_n) \ (C_1 \ \ldots \ C_k))$ *(lhs rhs)*

- On va converger vers une situation décidable en appliquant des règles.

- Réduction du séquent à l'aide de 4 règles :

R1: $\zeta \rightarrow \phi, \neg \lambda, \rho \quad \rightsquigarrow \quad \zeta, \lambda \rightarrow \varphi, \rho$

R2: $\phi \rightarrow \lambda, \rho \vee \pi \quad \rightsquigarrow \quad \phi \rightarrow \lambda, \rho, \pi$

$\phi, \zeta \wedge \rho \rightarrow \lambda \quad \rightsquigarrow \quad \phi, \zeta, \rho \rightarrow \lambda$

R3: $\phi, \zeta \vee \rho \rightarrow \lambda \quad \rightsquigarrow \quad \begin{cases} \phi, \zeta \rightarrow \lambda \\ \phi, \rho \rightarrow \lambda \end{cases}$

R4: $\phi \rightarrow \lambda, \zeta \wedge \rho \quad \rightsquigarrow \quad \begin{cases} \phi \rightarrow \lambda, \zeta \\ \phi \rightarrow \lambda, \rho \end{cases}$

- On applique les règles jusqu'à :
  - trouver une même formule dans les membres gauche et droit du séquent (CQFD)
  - saturation (ECHEC).

```
(define (valide? fbf)
  (if (not (fbf? fbf))
      (error "Syntaxe incorrecte" fbf)
      (wang '() (list (eliminer-implications fbf)))))



; LHS et RHS sont des listes de fbf sans implications

(define (wang LHS RHS)                ; le moteur de réécriture
  (if (intersection? LHS RHS)
      #t
      (let ((fbf (first-molecule LHS)))
        (if fbf
            (case (connecteur fbf)
              ((non) ; R1 : une négation à gauche
               (wang (remove fbf LHS) (add (arg1 fbf) RHS)))
              ((ou)  ; R3 : une disjonction à gauche
               (and (wang (add (arg1 fbf) (remove fbf LHS)) RHS)
                    (wang (add (arg2 fbf) (remove fbf LHS)) RHS)))
              ..........)))))
```

Programmer la Syntaxe !

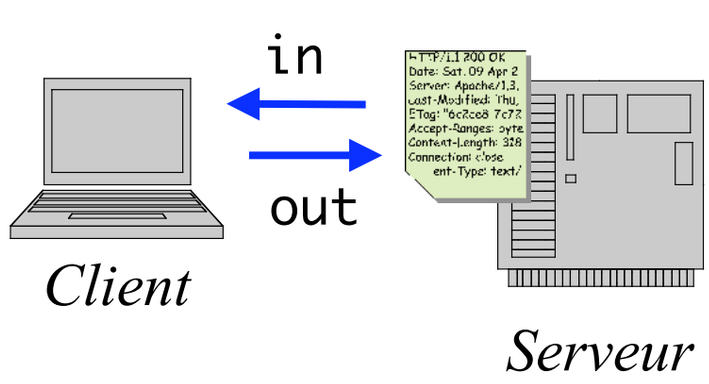
- Ou l'Art subtil des Macros (*hygiéniques*)

```
(define-syntax for
  (syntax-rules (to in)
    ((for i from a to b e1 e2 ...) (let ((vb b))
                                     (define (iter i)
                                       (if (> i vb)
                                           (void)
                                           (begin e1 e2 ...
                                                  (iter (+ i 1)))))
                                     (iter a)))
    ((for x in L e1 e2 ...) <analogue>)))
```

```
> (for k from 1 to 5 (write (sqr k)) (display " "))
1 4 9 16 25
> (for x in (iota 50 100)
    (if (premier? x) (printf "~a " x)))
101 103 107 109 113 127 131 137 139 149
```

Programmer un client Web

```
> (define-values (in out)
    (tcp-connect "www.boursorama.com" 80))
> (fprintf out
    "GET http://www.boursorama.com/index.html HTTP/1.0\n\n")
> (write (read-line in))
"HTTP/1.1 302 Found"
> (write (read-line in))
"Date: Thu, 31 Jan 2008 20:56:41 GMT"
> (write (read-line in))
"Server: Apache"
> ......
```



- On lit ligne à ligne la page Web convoitée et on extrait l'information !

expressions régulières

```
> (nasdaq)          ; en temps réel
+1.56%
```

Créer une Animation

- Modèle de type ActionScript ou Processing

big-bang : largeur hauteur fréq-horloge Monde → #t

end-of-time : String → Monde

on-redraw : (Monde → Scène) → #t

on-tick-event : (Monde → Monde) → #t

on-mouse-event : Monde x y MouseEvent → #t

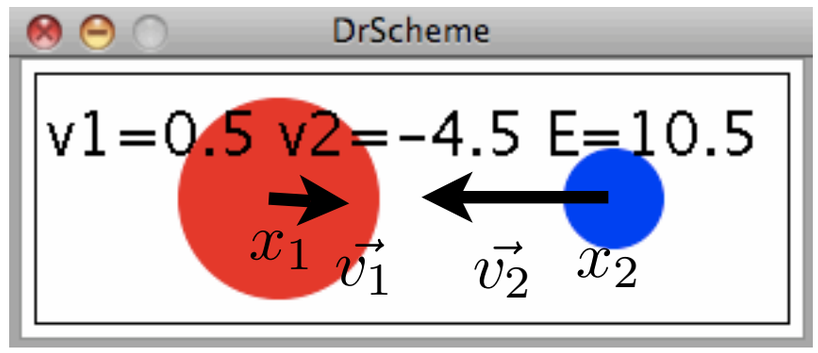
on-key-event : Monde KeyEvent → #t

Model-View-Controller

- On spécifie l'origine et la fin du Monde, la manière dont le Monde s'affiche et évolue, comment il réagit au clavier et à la souris, etc.

- Ex: collision et conservation de l'énergie cinétique

```
(define LARG 300)
(define HAUT 100)
(define HAUT/2 (/ HAUT 2))
(define R1 40)
(define R2 20)
(define M1 3.0)
(define M2 1)
(define DISK1 (circle R1 'solid "red"))
(define DISK2 (circle R2 'solid "blue"))
(define BACKGROUND (empty-scene LARG HAUT))

(define-struct Monde (x1 v1 x2 v2))

(define (energie m)          ; Monde → Number
  (* 0.5 (+ (* M1 (sqr (Monde-v1 m)))
            (* M2 (sqr (Monde-v2 m))))))
```



les struct de C...

```
(define (show-monde m)          ; Monde → Image
  (place-image (text "..." 24 "black") 5 10
               (place-image DISK2 (Monde-x2 m) HAUT/2
                            (place-image DISK1 ...))))

(define (next m)                ; Monde → Monde
  (let ((x1 (Monde-x1 m)) (v1 (Monde-v1 m)) (x2 ...) (v2 ...))
    (let ((X1 (+ x1 v1)) (X2 (+ x2 v2)))
      (cond ((< X1 R1) (if (> X2 (- LARG R2))
                           (make-Monde R1 (- v1) (- LARG R2) (- v2))
                           (make-Monde R1 (- v1) X2 v2)))
            ((> X2 (- LARG R2)) (make-Monde X1 v1 (- LARG R2) (- v2)))
            (else (let ((dist (abs (- X2 X1))))
                    (if (< dist (+ R1 R2))
                      (let ((V1 <nouvelle vitesse de la balle 1>
                                (V2 <nouvelle vitesse de la balle 2>))
                        (make-Monde (+ X1 V1) V1 (+ X2 V2) V2))
                      (make-Monde X1 v1 X2 v2)))))))))

(big-bang LARG HAUT 0.05 (make-Monde 50 2 300 -3))
(on-redraw show-monde)
(on-tick-event next)
```

- Ah, oui, les nouvelles vitesses ! Mmmm…
- Je délègue le travail à Mathematica™ via un appel système sur la ligne de commande (Unix™) :

```
(call-with-output-file "foo.m"
  (lambda (out)
    (fprintf out
      "Solve[{m1*v1+m2*v2==m1*V1+m2*V2, m1*v1^2+m2*v2^2==m1*V1^2+m2*V2^2},
            {V1,V2}] >> foo.txt;")))
```

```
(system "MathKernel -run << foo.m")

(call-with-input-file "foo.txt"
  (lambda (in)
    (printf "~a\n~a\n" (read-line in) (read-line in))))
```

```
Bienvenue dans DrScheme, version 371
Mathematica 5.2 for Mac OS X x86
Copyright 1988-2006 Wolfram Research, Inc.
{{V1 -> v1, V2 -> v2}, {V1 -> (m1*v1 - m2*v1 + 2*m2*v2)/(m1 + m2),
  V2 -> (2*m1*v1 - m1*v2 + m2*v2)/(m1 + m2)}}
```

à traduire (automatiquement) en Scheme ! (avec Lex/Yacc)

Classes, Objets, Messages (soft)

```
(define (point2d x0 y0)
  (let ((x x0) (y y0))
    (define (this method . Largs)
      (case method
        ((getX) x)
        ((getY) y)
        ((setX) (set! x (car Largs)))
        ((setY) (set! y (car Largs)))
        ((reset) (this 'setX x0) (this 'setY y0))
        ((toString) (format "point2d(~a ~a)" x y))
        (else (error "Unknown method" method))))
    this))

(define a (point2d 10 20))     ; Java:  a = new Point2d(10,20);
(a 'setY (+ (a 'getY) 1))      ; Java:  a.setY(a.getY()+1)
(a 'toString)                  ; Java:  a.toString()
```

Scheme

=

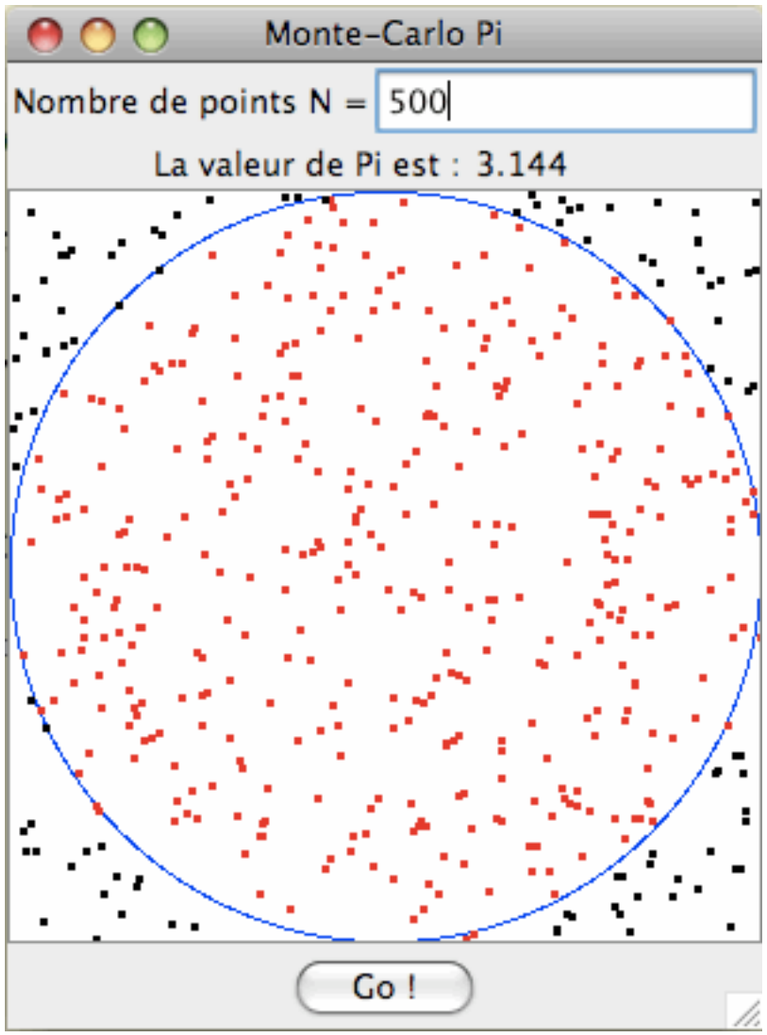
Scheme++

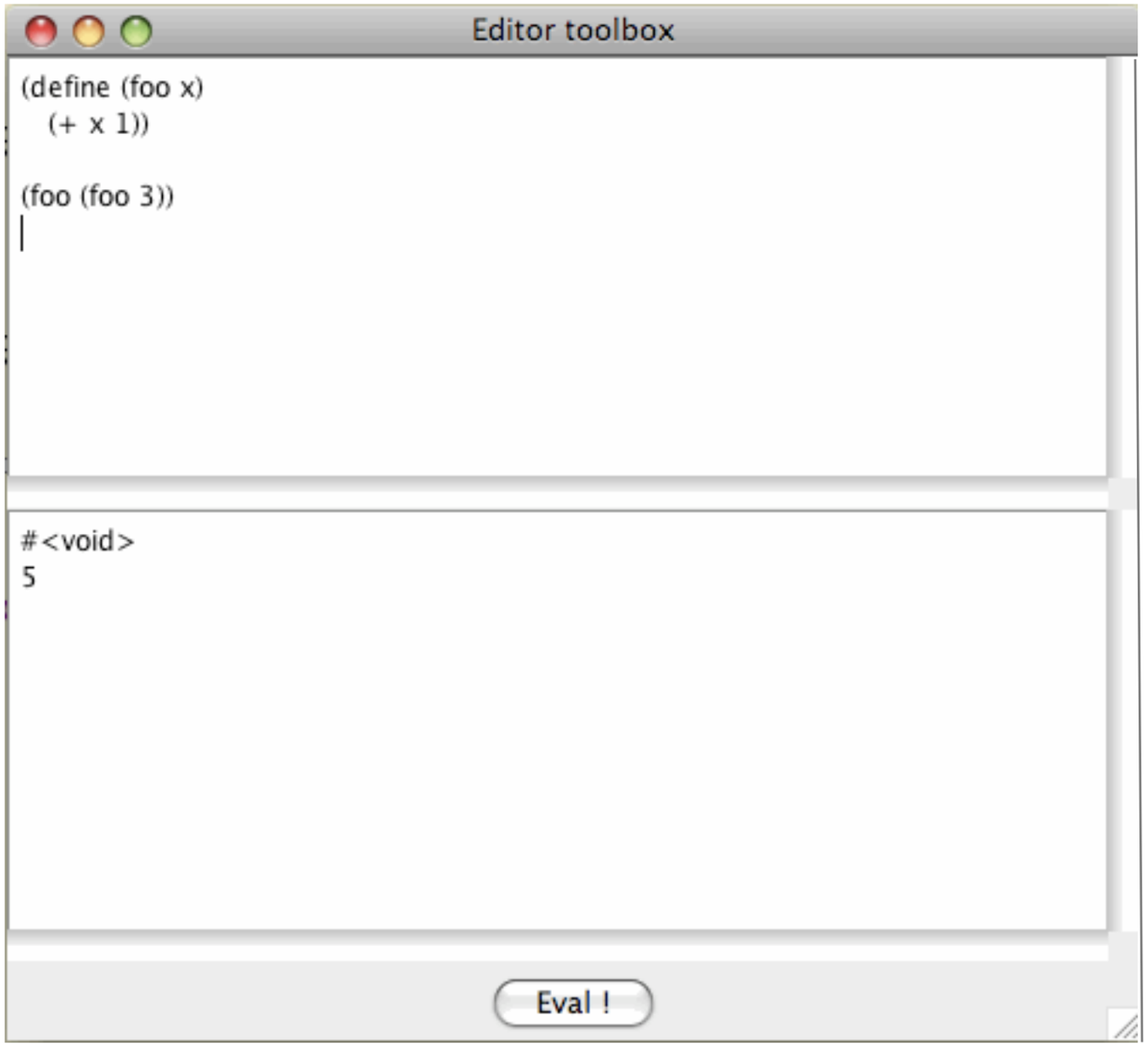
Classes, Objets, Messages (hard)

```
(define Lpoints '())                    ; la liste globale de tous les points

(define point2d%
  (class object%                        ; en Java :     extends Object
    (init-field (x 0) (y 0))
    (define/public (getX) x)
    (define/public (getY) y)
    (define/public (setX new-x)
      (set! x new-x))
    ...
    (super-new)
    (set! Lpoints (cons this Lpoints))))


(define a (new point2d% (x 10) (y 20)))
(send a setY (+ (send a getY) 1))
(send a toString)
```

La vraie couche objet !

Interfaces graphiques (objets)





Canvas, boutons, text-fields, menus,
souris, éditeurs de texte, navigateurs...

Et ces slides, elles sont en quoi ?

- Elles sont réalisées avec *Keynote* (Apple), cool :-)

- Mais j'aurais pu les écrire en Scheme avec l'outil de présentation SlideShow !

```
(module test-slideshow (lib "slideshow.ss" "slideshow")
  (require (lib "code.ss" "slideshow"))
  (slide/title "Un exemple de slide"
   (page-item "La valeur de " (code (log 2))
              "est environ " (number->string (log 2)))
   (colorize (page-item "Mais vous le saviez !") "blue")
   (page-item "Et bla-bla-bla..."))
  (slide/title "Le dessin"
   (page-para "On peut programmer des dessins et des animations.")
   (page-para "Par exemple," (code (disk 100)) " produit :")
   (disk 100)))
```

Un exemple de slide

- La valeur de `(log 2)` est environ 0.6931471805599453
- Mais vous le saviez !
- Et bla-bla-bla...

```
(slide/title "Un exemple de slide"
   (page-item "La valeur de " (code (log 2))
              "est environ " (number->string (log 2)))
   (colorize (page-item "Mais vous le saviez !") "blue")
   (page-item "Et bla-bla-bla..."))
```

Le dessin

On peut programmer des dessins et des animations.

Par exemple, `(disk 100)` produit :

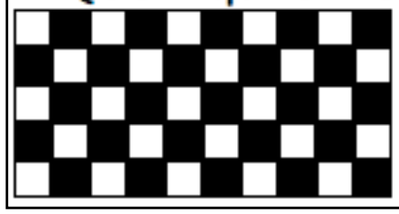
```
(slide/title "Le dessin"
   (page-para "On peut programmer des dessins et des animations.")
   (page-para "Par exemple," (code (disk 100)) " produit :")
   (disk 100))
```

Et débuter avec SlideShow ?...

```
#lang slideshow

(define r1 (rectangle 10 10))
(define r2 (filled-rectangle 10 10))

(define (other r)
  (if (equal? r r1) r2 r1))

(define (colonne n r)
  (if (= n 1)
      r
      (vc-append r (colonne (- n 1) (other r)))))

(define (échiquier n p r)
  (if (= p 1)
      (colonne n r)
      (hc-append (colonne n r) (échiquier n (- p 1) (other r)))))
```

```
> (list r1 r2)
(□ ■)
> (colonne 5 r1)
```

```
> (échiquier 5 10 r1)
```



Ah oui , l'efficacité !...

- En principe la vitesse en mode interprété est bien plus lente que celle de C mais largement suffisante.

- Il existe de bons compilateurs (bigloo, chicken...).

- Dans la lignée des compilateurs LISP (MacLisp).

- Et souvenez-vous : un benchmark c'est comme un sondage : on peut lui faire dire tout et n'importe quoi. Soignez vos algorithmes !...

- Exemple page suivante. Accrochez vos ceintures !

gcc

```
int ack(int x, int y) {
    if (x==0) return y+1;
    if (y==0) return ack(x-1,1);
    return ack(x-1,ack(x,y-1));
}
```

```
$ gcc ack-c.c -o ack-c
$ ./ack-c
ack(3,13) = 65533 (time = 37811 ms)
$ gcc ack-c.c -O9 -o ack-c
$ ./ack-c
ack(3,13) = 65533 (time = 10929 ms)
```

bigloo

```
$ bigloo -Obench ack-bigloo.scm -o ack-bigloo
$ ./ack-bigloo
(ack 3 13) = 65533 (time = 4660 ms)
```

```
(define (ack x::int y::int)::int
  (cond ((= x 0) (+ y 1))
        ((= y 0) (ack (- x 1) 1))
        (else (ack (- x 1) (ack x (- y 1))))))
```

The spirit of Lisp hacking can be expressed in two sentences.

-Programming should be fun.

-Programs should be beautiful.

Paul Graham