Introducción a Scheme

Victor Ramiro

Tipos en Scheme

- Booleans: `#t` `#f`
- Números enteros: 0, 1, 123, -5
- Números reales: 3.14159
- Strings: `"hola que tal", ...`
- Caracteres: `#\a #\;`

Tipos en Scheme

- Símbolos: `a, b, +, *valor*`
  - La evaluación de un símbolo es su valor asociado. Se usan como variables en los programas.

Tipos en Scheme

- Listas: `(1 #t #\. "hoola" a)`
- Una lista se evalúa de la siguiente forma:
  - Primero se evalúan todos sus elementos
    - El resultado de evaluar el primer elemento debe ser un procedimiento P
    - El resultado de evaluar los demás elementos son los argumentos
    - Se invoca P sobre los argumentos
- Los programas en Scheme se representan mediante listas!

Quote

- Se puede evitar la evaluación con la forma especial **quote**:

```
(quote 1) => 1
(quote a) => a
(quote (a b c)) => (a b c)
```

- Azúcar sintáctico: '**exp** es equivalente a (quote exp)

```
'1 => 1
'a => a
'(a b c) => (a b c)
```

Variables

- Las variables son símbolos.
- Variables locales: son los argumentos de un procedimiento.
- Variables globales: se definen con **define**

```
(define var exp)
(define pi 3.14159)
(define msg "hello world")
(define pi/2 (/ pi 2))
```

Condicionales

- Para evaluar en forma condicional se usa la forma especial if:

```
(if (> 2 1) (display "si") (display "no")) => "si"
```

- En general: `(if bexp texp fexp)`
- Se evalúa **bexp**.
- Si es #t: se evalúa **texp** y se entrega su resultado (sin evaluar **fexp**)
- Si es #f: se evalúa **fexp** y se entrega su resultado (sin evaluar **texp**)

Condicionales

```
(cond
  ((< x y) "menor")
  ((> x y) "mayor")
  (else    "igual"))
```

- En general:

```
(cond
        (bexp1   texp1)
        (bexp2   texp2)
        ...
        (else    eexp) )
```

- Se evalúa bexp1. Si no es #f se evalúa texp1 y se entrega el resultado. En este caso nunca se evalúan bexp2, texp2, etc. En caso contrario, se evalúa bexp2 y se opera como en el caso anterior. Si todos los bexpk son falsos, se evalúa eexp y se entrega el resultado.

Procedimientos

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

- En general:

```
(define (fun arg1 arg2 ...)
  body-exp)
```

Variables Locales

- Let: introduce variables locales.

```
(let ((x (+ a b))
      (y (/ c d)))
  (+ (* x x) (* y y)))
```

- En general:

```
(let ((sym1 ini1)
      (sym2 ini2)
        ... )
  exp)
```

- Introduce las variables sym1 sym2 ... con valores iniciales ini1 ini2.

Variables Locales

- El alcance de sym1 sym2 es únicamente exp. Esto significa que en:

```
(let ((x 1))
  (let ((x (+ x 2))
        (y (+ x 3)))
    (+ x y)))
```

- Equivale a:

```
(let ((x0 1))
  (let ((x1 (+ x0 2))
        (y  (+ x0 3)))
    (+ x1 y)))
```

Funciones Básicas

```
(car '(1 2 3)) => 1

(cdr '(1 2 3)) => (2 3)

(car '()) o (cdr '())   => error

(cons 0 '(1 2 3)) => (0 1 2 3)

(car (cons x l)) <=> x  y  (cdr (cons x l)) <=> l

(cons 1 2) => ( 1 .  2 )  ;; lo que no es *propiamente* una lista

;;  En realidad (1 2 3) <=> ( 1 . ( 2 . ( 3 . () ) ) )

(cdr '( 1 . 2 ) ) => 2

(list? '(1 . 2)) => #f  ;;verdadero si se trata de una lista *propiamente*

(pair? '(1 . 2)) => #t

(null? '())      => #t  ;;falso si no es ()
```

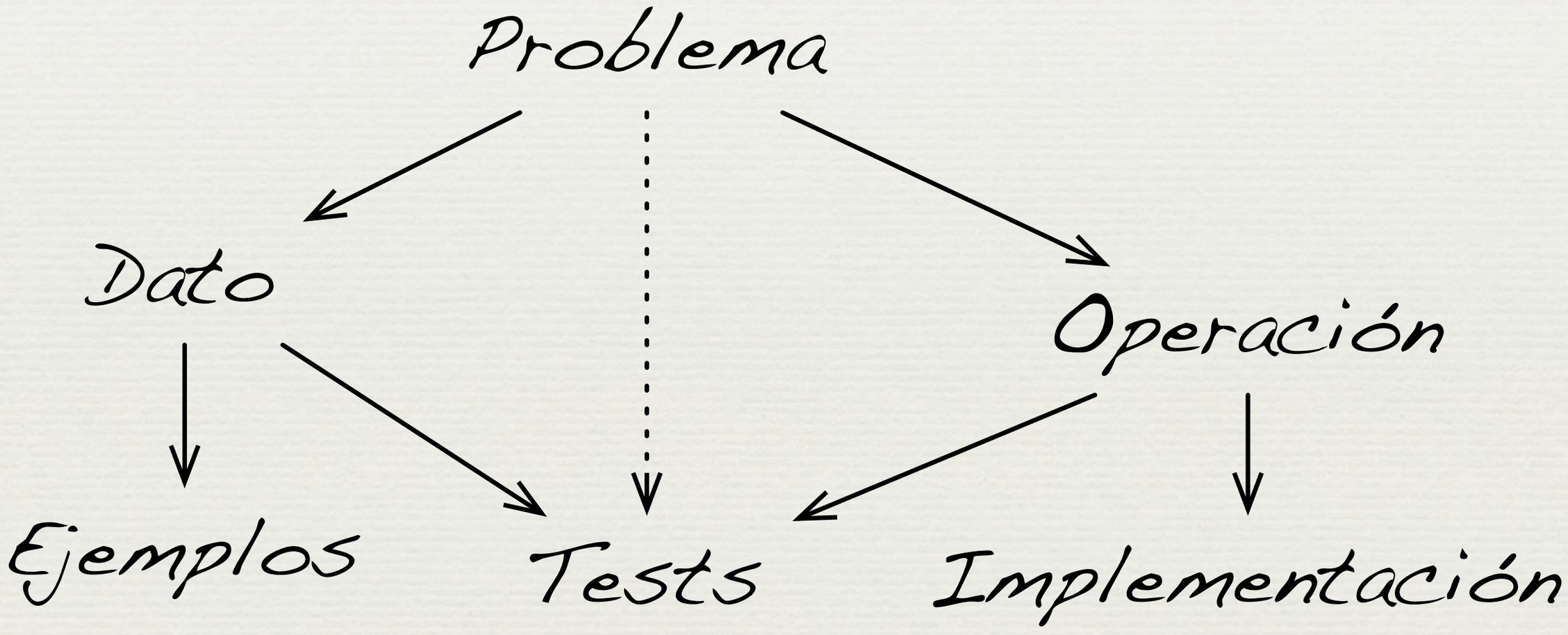
Funciones Básicas

```
(list 'a "hola" 3) => (a "hola" 3)

(length '(1 2 3)) => 3

(length '())      => 0

(append '(a (b)) '((c))) =>  (a (b) (c)) ;;los argumentos deben ser listas

(reverse '(1 2 3)) => (3 2 1)

(member 'b '(a b c)) => (b c)   ;; membresía

(member 'd ('a b c)) => #f

(assoc 'b '((a 1) (b 2) (c 3))) => (b 2) ;; manejo de listas de asociación
```

Funciones Básicas

```
number?  ;;determina si un objeto es un número (real o entero)

integer? ;;determina si un objeto es un número entero

= < <= > >=  ;; para comparar números

(max x1 x2 ...)  y (min x1 x2 ...)

(+ x1 x2 ...) y (* x1 x2 ...)

(- x1 x2 ...) y (/ x1 x2 ...)

 ;;(/ 3 2) => 1.5 pero (quotient 3 2) => 1 y (remainder 3 2) => 1

sqrt exp log sin cos tan asin acos atan

(expt x y) calcula x elevado a y
```

HTDP



Listas

- Problema: Sumar los elementos de una lista
- Datos: Definidos por la siguiente gramática BNF

```
lon := '() | '(h t)
h   := NUMBER
t   := lon
```

- Ejemplos de Datos: `'(),'(1 2 3)`
- Operación: `(define (suma-l l) 6)`
- Test: `(suma-l '(1 2 3)) => 6`

Axiomas de Listas

- Axiomas de listas: `lon := '() | '(h t)`

```
(null? `()) => #t
(car (cons h t)) => h
(cdr (cons h t)) => t
```

- Patrón:

```
(define (fun l)
 (if (null? l) ....
       (... (car l) .... (cdr l).....)))
```

```
;; Contract: suma-l : list-of-numbers -> number
;; Purpose: sum-up all elements on a list
;; Example: (suma-l '(1 2 3)) should produce 6
;; Definition: [refines the header]
(define (suma-l l)
  (if (null? l)
      0
      (+ (car l) (suma-l (cdr l)))))

;; Tests:
(suma-l '(1 2 3))
;; expected value 6
```

Ejercicio

```
;; Contract: rev : list-> list
;; Purpose: new list in reverse order
;; Example: (rev '(a b c)) => '(c b a)

;; Definition:
(define (rev l) '(c b a))

;; Tests:
(rev '(a b c))
;; expected value '(c b a)

;; hint: (append l1 l2) => (l1 l2)
```

Ejemplos

```
(define (fib n)
  (if (= n 0)
      1
      (+ (fib (- n 1)) (fib (- n 2)))))

(define (rev l)
  (if (null? l)
      '()
      (append (rev (cdr l)) (list (car l)))))

(define (suma-l l)
  (if (null? l)
      0
      (+ (car l) (suma-l (cdr l)))))
```

Calcular sqrt(x)

(sqrt 4.0) => 2.0

Algoritmo de aproximación:

$$x_0 = 1$$

$$x_n = \frac{x_{n-1} + \frac{x}{x_{n-1}}}{2}$$

(sqrt x) (1)

```
(define (sqrt x) (try 1 x))
```

Algoritmo:

```
(define (try guess x)
  (if (good? guess x)
      guess
      (try (improve guess x) x)))
```

(sqrt x) (2)

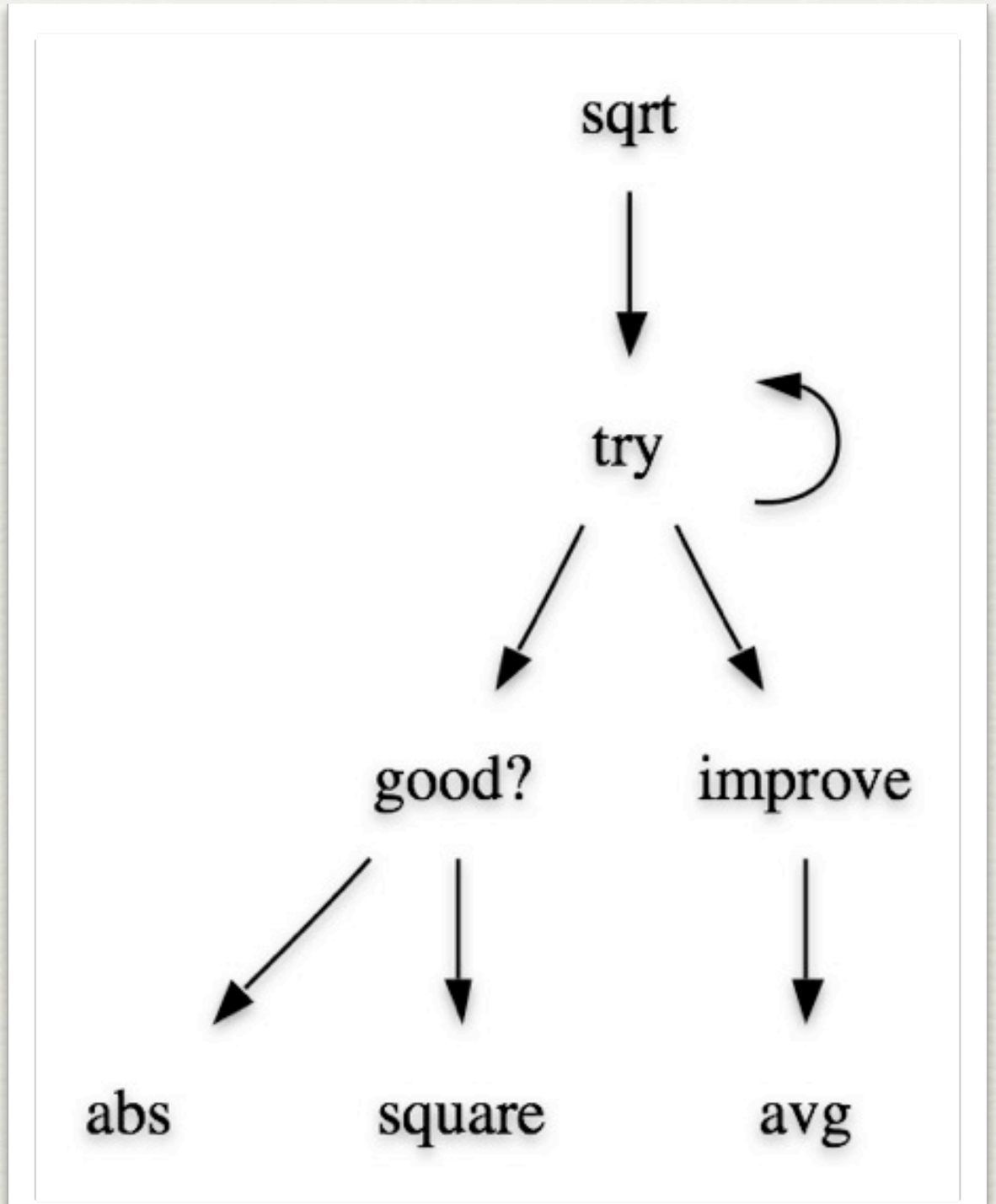
```
(define (good? g x)
  (< (abs (- x (square g)))  *epsilon*))

(define *epsilon* .0001)

(define (improve g x)
  (average g (/ x g)))
```

(sqrt x) (3)

Funciones Auxiliares:

```
(define (square x) (* x x))

(define (average x y) (/ (+ x y) 2))
```

Black-box Abstraction



(sqrt x) (4)

```
(define (sqrt x)
  (define (improve g) (average g (/ x g)))
  (define (good? g)
    (< (abs (- x (square g))) .001))
  (define (try g)
    (if (good? g)
        g
        (try (improve g))))
  (try 1))
```

funciones

- Hasta el momento:
  - Pueden ser definidas funciones dentro de funciones (funciones como valor)

sum-int (l)

$$\sum_{i=a}^{b} i$$

```
(define (sum-int a b)
  (if (> a b)
      0
      (+ a (sum-int (+ 1 a) b))))
```

sum-sq (1)

$$\sum_{i=a}^{b} i^2$$

```
(define (sum-sq a b)
  (if (> a b)
      0
      (+ (* a a) (sum-sq (+ 1 a) b))))
```

sum-pi (1)

$$\sum_{i=a,a+=4}^{b} \frac{1}{i(i+2)}$$

```
(define (sum-pi a b)
  (if (> a b)
      0
      (+ (/ 1 (* a (+ a 2)))
         (sum-pi (+ 4 a) b))))
```

sum-sq (2)

$$\sum_{i=a}^{b} i^2$$

```
(define (sum-sq a b)
  (if (> a b)
      0
      (+ (square a)
         (sum-sq (add1 a) b))))

(define (square x) (* x x))
(define (add1 x) (+ 1 x))
```

sum-pi (2)

$$\sum_{i=a,a+=4}^{b} \frac{1}{i(i+2)}$$

```
(define (sum-pi a b)
  (if (> a b)
      0
      (+ (f a) (sum-pi (add4 a) b))))


(define (f a)(/ 1 (* a (+ a 2))))
(define (add4 i)( + 4 i))
```

- Es lo mejor que podemos hacer?

Patrón Común

```
(define (<nombre> a b)
  (if (> a b)
      0
      (+ (<termino> a)
         (<nombre>
              (<next> a)
              b))))
```

sum (1)

```
(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)
         (sum term
              (next a)
              next
              b))))
```

sum-int (2)

```
(define (sum-int a b)
  (define (identity a) a)
  (define (next a) (+ 1 a))
  (sum identity a next b))
```

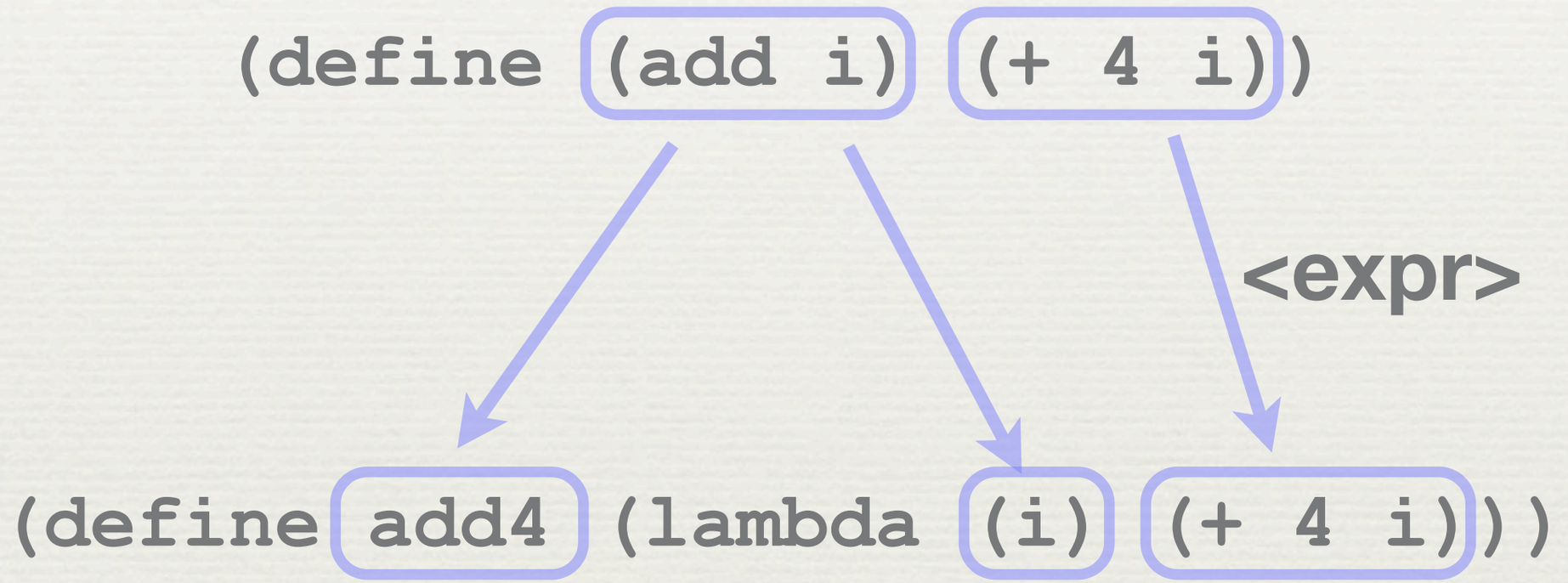
pi-sum (3)

```
(define (pi-sum a b)
  (sum (lambda(i)(/ 1.0 (* i (+ i 2))))
a
(lambda(i)(+ 4 i))
b))
```

sum-sq (3)

```
(define (sum-sq a b)
(define next (lambda(x)(+ 1 x)))
(sum square a next b))
```

Definiendo funciones



- ¿Cómo haríamos lo mismo en Java?

Como haríamos lo mismo en Java...

```
interface Function{
  public double eval(int x);
}
```

```
class Square implements Function{
   public double eval(int x){
      return x * x;
   }
}
```

Como haríamos lo mismo en Java...(2)

```
class Sum {
   static public double sum(Function f,
                            int a, int b, Function next) {
      double s = 0;
      for (int i = a; i <= b;) {
         s += f.eval(i);
         i = (int) next.eval(i);
      }
      return s;
   }
}
```

Como haríamos lo mismo en Java...(2)

```
public class Main {
   static public void main(String[] args) {
      Function f;
      f = new Square();
      Function add1 = new Function() {
         public double eval(int x) {
            return 1 + x;
         }
      };
      Sum s = new Sum();
      double suma = s.sum(f, 1, 100, add1);
      System.out.println(suma);
   }
}
```

Bibliografía

- DrScheme
  - http://www.plt-scheme.org/software/drscheme/
- How to Design Programs
  - http://www.htdp.org/
- Little Schemer
  - http://www.ccs.neu.edu/home/matthias/BTLS/