Introducción a Scheme

Victor Ramiro
cc41a

Clase pasada

- Tipos basicos de scheme (numero, booleans, etc)
- Definición de variables en el top level y variables locales usando let
- Condicionales: if, cond
- Definición de funciones (sin lambda aún)

```
(define global_var expr)

(let ( (k1 v1)
       (k2 v2)
       )
  expr_k1_k2
  )

(if btest texpr fexpr)

(cond
  (btest1 expr1)
  (btest2 expr2)
  (btest3 expr3)
  (else expr)
  )

(define (fun arg1 arg2 ...) expr )
```

Clase pasada

- Axiomas de listas:

```
(car (cons h t)) => h
(cdr (cons h t)) => t
```

- Manejo de listas según su estructura
- Tomar la gramática
  - lon := '() | '(h t)
- y llevarlo al patrón:

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

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)))))
```

(map fun list)

```
(define (map f l)
  (if (null? l)
      '()
      (cons (f (car l)) (map f (cdr l)))))

(define (add1 x) (+ 1 x))

(map add1 '(1 2 3)) --> (2 3 4)
```

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)))
```

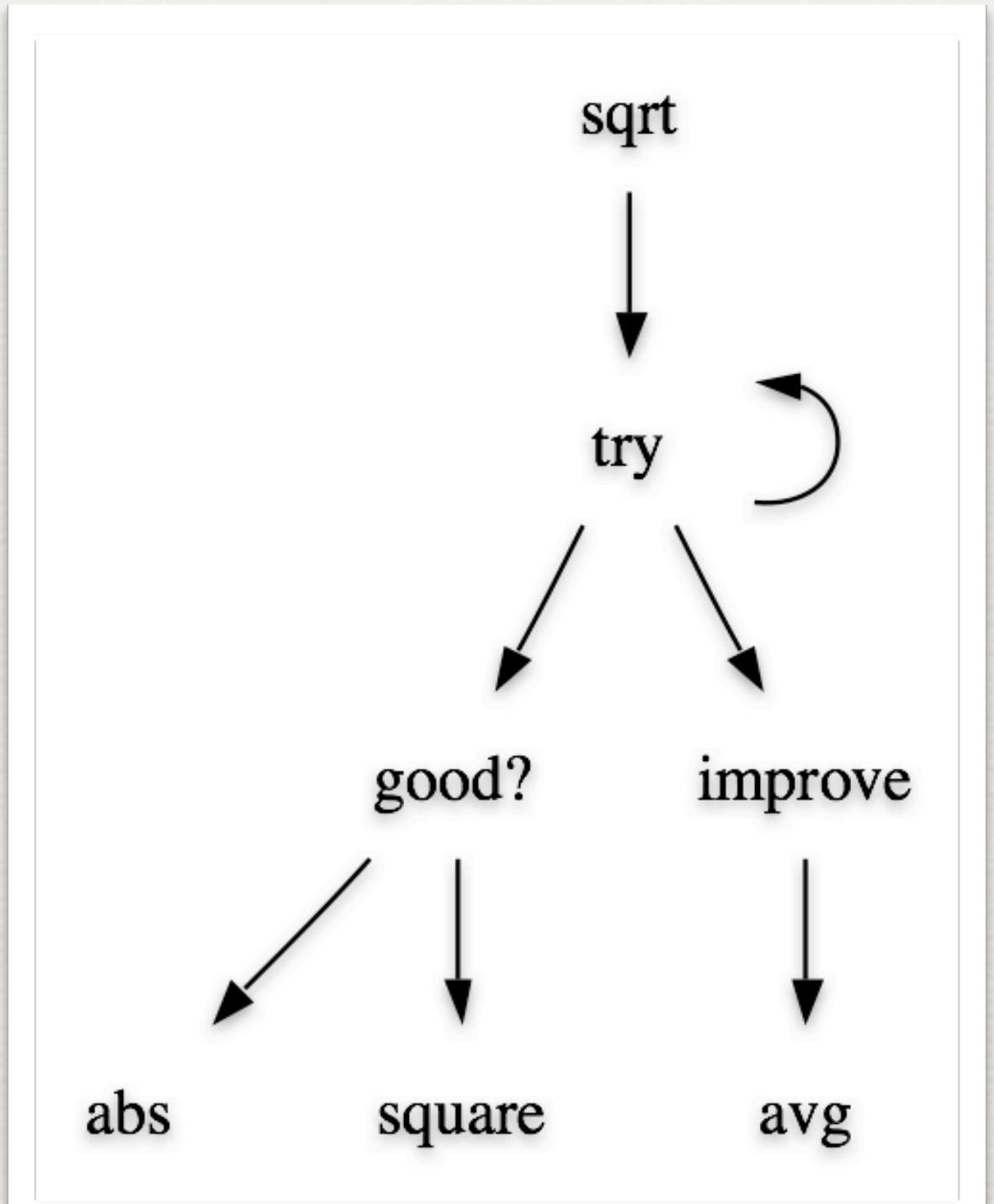
(sqrt x) (3)

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 (1)

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

```
(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))
```

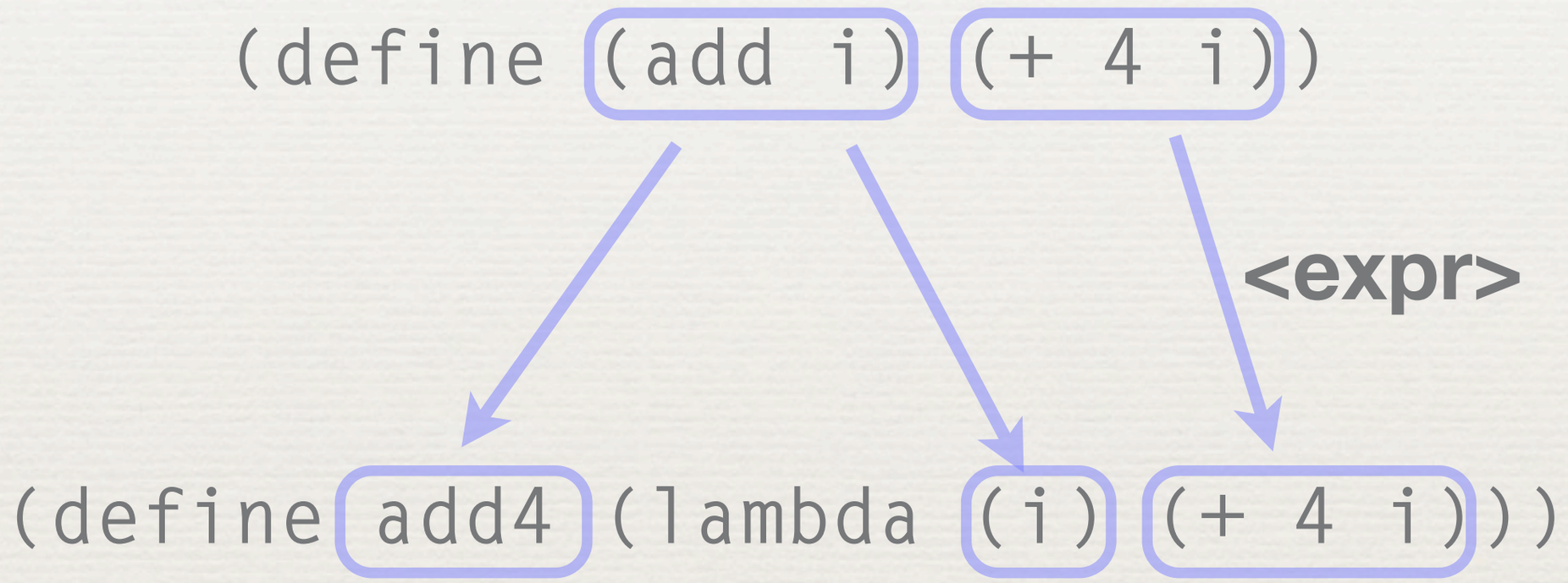
pi-sum (3)

```
(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



Funciones como Valores

- al igual que números, listas, estructuras, etc.
- basado en lambda calculo

lambda calculus

```
<expr> := <id>
        | (<expr> <expr>)
        | (lambda (<id>) <expr>)
```

Gramática de Scheme

```
<def>  =    (define (<var> <var> ... <var>) <exp>)
          | (define <var> <exp>)
          | (define-struct <var0> (<var-1> ... <var-n>))

<exp>  =    <var>
          | <con>
          | <prm>
          | (<exp> <exp> ... <exp>)
          | (cond (<exp> <exp>) ... (<exp> <exp>))
          | (cond (<exp> <exp>) ... (else <exp>))
          | (local (<def> ... <def>) <exp>)
          | (lambda (<var> ... <var>) <exp>)
          | (set! <var> <exp>)
          | (begin <exp> ... <exp>)
```

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);
   }
}
```

- En Scheme, las funciones son valores:
  - Pueden ser definidas dentro de otras funciones (funciones como valor)
  - Pueden ser pasadas como parámetro
  - Pueden ser definidas anónimamente

- Que implicancias tiene esto?