Introducción a Scheme

Victor Ramiro
cc41a

Clase pasada

- Funciones como valor
- Funciones como parámetro
  (Funciones de orden superior)
- lambda calculus

lambda calculus

```
<def> := (define <id> <expr>)

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

Definiendo Funciones

```
f(x) = x + 3
f(5)
```

```
f = λ(x).x + 3
f(5)
```

```
((λ(x) x + 3) 5)
```

Parámetro Formal

Argumento

Funciones en javascript

```
function first(param) {
 alert(param + "method to define a function");
}
```

```
second = function(param){
 alert(param + "method to define a function")};
```

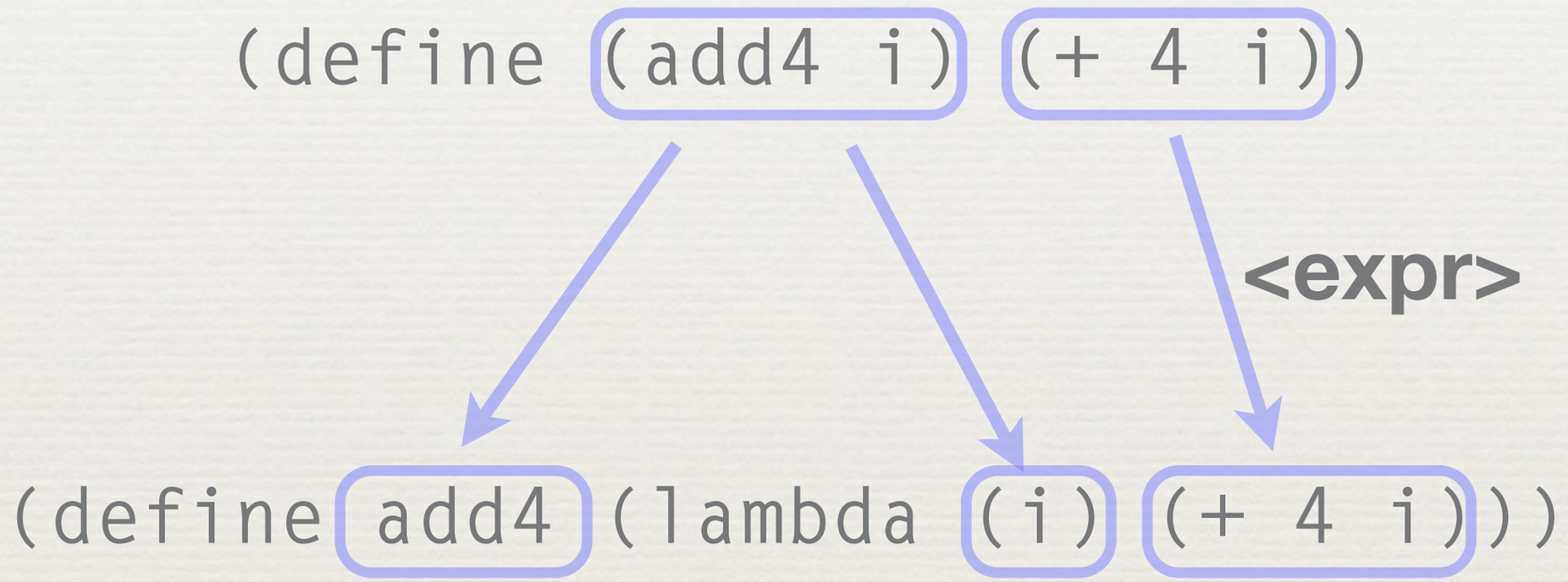
Funciones en Ruby

```
def inc(x)
 return x+1
end
```

```
add1 = lambda { |x| x + 1 }
```

```
print inc(1)
print add1[1]
```

Definiendo funciones



Sustitución

```
((lambda(x) e) ev) -> e [x/ev]
```

```
((λ(x) x + 3) 5)

((λ(5) x + 3))

(5 + 3)

8
```

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))
```

sum-sq (3)

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

Filter-<

```
;; filter-<-pivot:
;; number list(number) → list(number)
(define (filter-<-pivot pivot l)
  (cond
    [(empty? l) empty]
    [(cons? l)
     (if (< (first l) pivot)
         (cons (first l)
               (filter-<-pivot pivot (rest l)))
         (filter-<-pivot pivot (rest l)))]))
```

Filter->

```
;; filter->-pivot:
;; number list(number) → list(number)
(define (filter->-pivot pivot l)
  (cond
    [(empty? l) empty]
    [(cons? l)
     (if (> (first l) pivot)
         (cons (first l)
               (filter-<-pivot pivot (rest l)))
         (filter-<-pivot pivot (rest l)))]))
```

Filter (2)

```
;; filter-pivot:
;; X (X X → boolean) list(X) → list(X)
(define (filter-pivot pivot comp l)
  (cond
    [(empty? l) empty]
    [(cons? l)
      (if (comp (first l) pivot)
        (cons (first l)
          (filter-pivot pivot comp (rest l)))
        (filter-pivot pivot comp (rest l)))]))
```

Filter (3)

```
;; filter-any:
;; (X → boolean) list(X) → list(X)
(define (filter-any comp l)
 (cond
  [(empty? l) empty]
  [(cons? l)
   (if (comp (first l))
     (cons (first l)
           (filter-any comp (rest l)))
     (filter-any comp (rest l)))]))
```

Quicksort

```
(define (qs comp l)
 (cond
  [(empty? l) empty]
  [(cons? l)
   (local ([define pivot (first l)])
   (append
    (qs comp
        (filter-any
            (lambda(x)(comp x pivot))
            (rest l)))
    (list pivot)
    (qs comp
        (filter-any
            (lambda(x)(not (comp x pivot)))
            (rest l)))))]))
```

diff

```
(define H 0.0001)

(define (d/dx f )
 (lambda (x)
  (/ (- (f (+ x H)) (f x))
     H)))
```

diff (2)

```
(define H 0.0001)

(define d/dx
  (lambda(f)
    (lambda (x)
      (/ (- (f (+ x H)) (f x))
         H))))
```

Uso de diff

```
(define diff-square (d/dx (lambda (x) (* x x))))
```

```
(diff-square 10) => 20.000099999890608
```

Composición

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

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

Composición (2)

```
(define id (compose square sqrt))
(define id2 (compose sqrt square))

(id 10)   =>  10.000000000000002
(id2 10)  =>  10
```

Taxonomía de funciones

- Primer Orden: No son valores en el lenguaje. Solo pueden ser definidas en una parte especifica del código, donde son nombradas para su uso posterior
- Orden Superior: Funciones que pueden retornar funciones como valores
- Primera Clase: Son valores en el lenguaje. Es decir, pueden ser: argumentos, valor de retornos y guardados en alguna estructura de datos