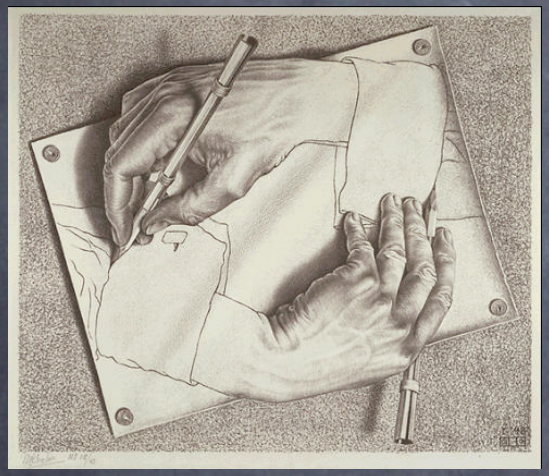
# Programación Funcional

José Manuel Gómez Soto
Universidad La Salle

CONACICO-2009
UAZ Campus-Jalpa

# Conceptos

Recursividad



Programación Funcional

$$f(g(h(x)))$$

Cálculo Lambda

Procesamiento simbólico

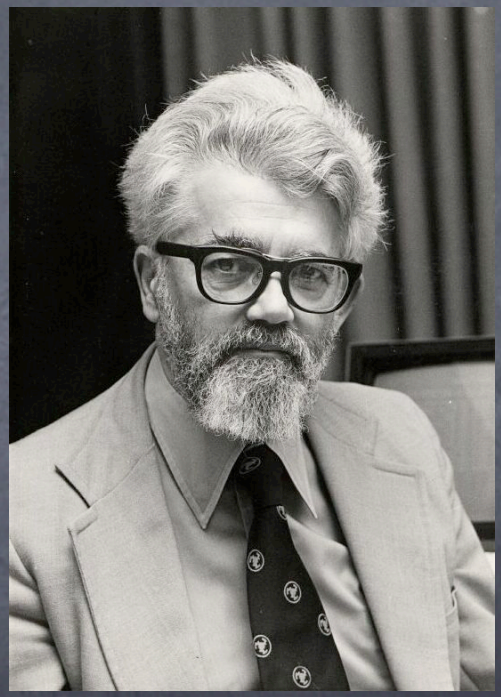
$$f(x) = 2x^3$$

$$f'(x) = 6x^2$$

# Lisp
## (LISt Processing)

Lisp fue inventado a finales de los in 50s como un formalismo para analizar ciertos tipos de modelos computacionales llamdos ecuaciones recursivas.


PDP-1

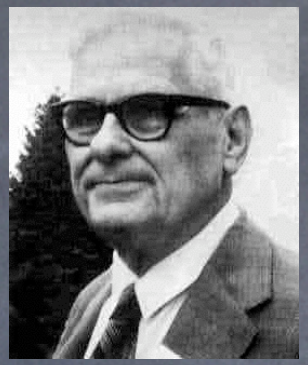


John McCarthy

J. McCarthy: Recursive Functions of Symbolic Expressions and their Computation by Machine. MIT AI Lab., AI Memo No. 8, Cambridge March 1959.

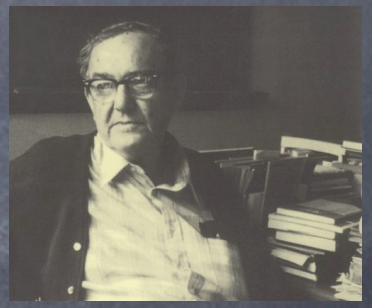
# Las ideas detrás de Lisp

Cálculo Lambda



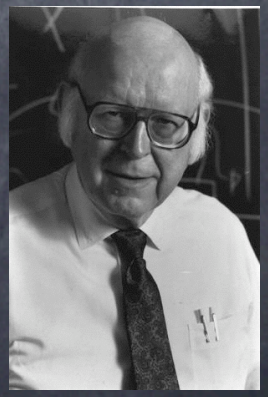
Alonzo Church

Sistemas simbólicos físicos



Hebert Simons

En algún nivel de la abstracción humana los procesos del pensamiento pueden ser modelados como relaciones entre símbolos.



Allen Newell

# expresiones primitivas

- Números
- Operadores

# Expresiones y procedimientos primitivos

Expresiones que representan

números: 512,-3.5,1.2e5, ...

cadenas: "esto es una cadena" ,
"esta es otra cadena con
^&*((^".

Valores booleanos: #T , #f.

# Expresiones y procedimientos primitivos

Expressions primitivas para manipular datos primitivos.

números: +, -, *, /, >, <, >=, < =, =.

cadenas: string-length, string=?

valores boleanos: and, or y not.

# Combinaciones

Las expresiones que representan números pueden combinarse con una expresión que representa un procedimiento primitivo para formar una expresión compuesta.

⇨(+ 137 349) ⇦

Operator Operandos

Los paréntesis delimitan el alcance del operador

# Ejemplos:

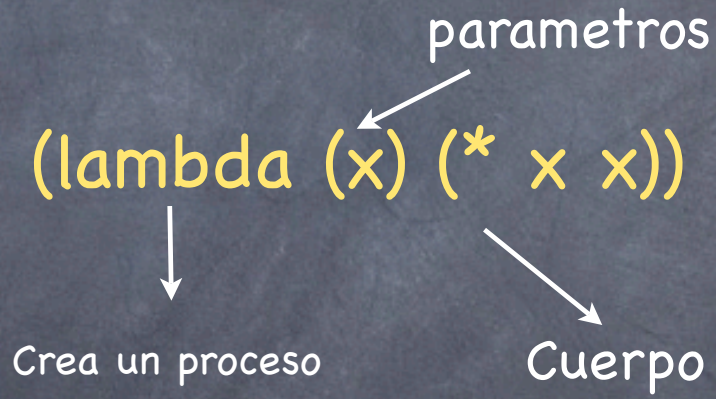
```
>(+ 137 349)
486

>(- 1000 334)
666

>(* 5 99)
495

>(/ 10 5)
2
```

Expresión lambda

Expresiones Anónimas

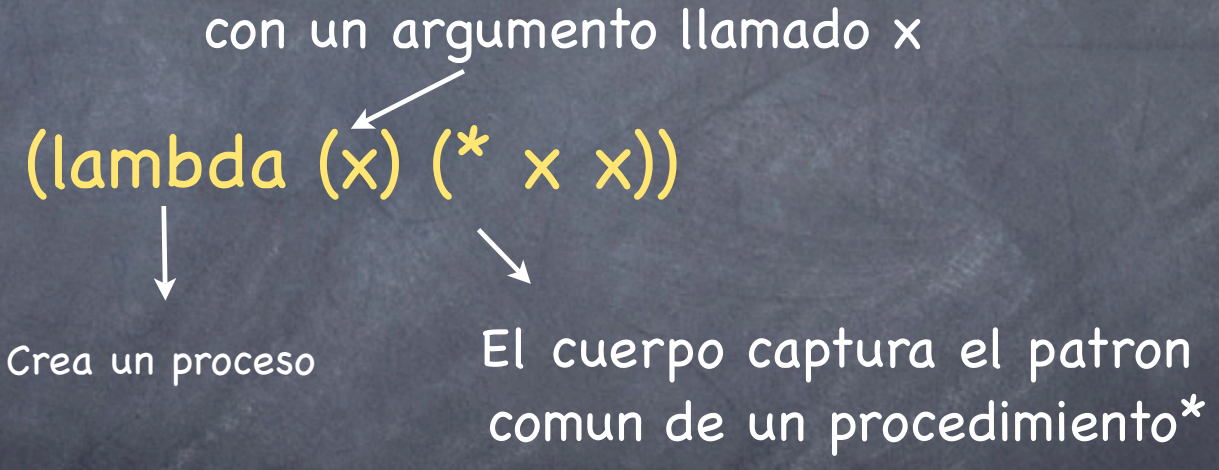
# Expresiones Lambda

En Scheme se definen los procedimientos mediante una expresion LAMBDA



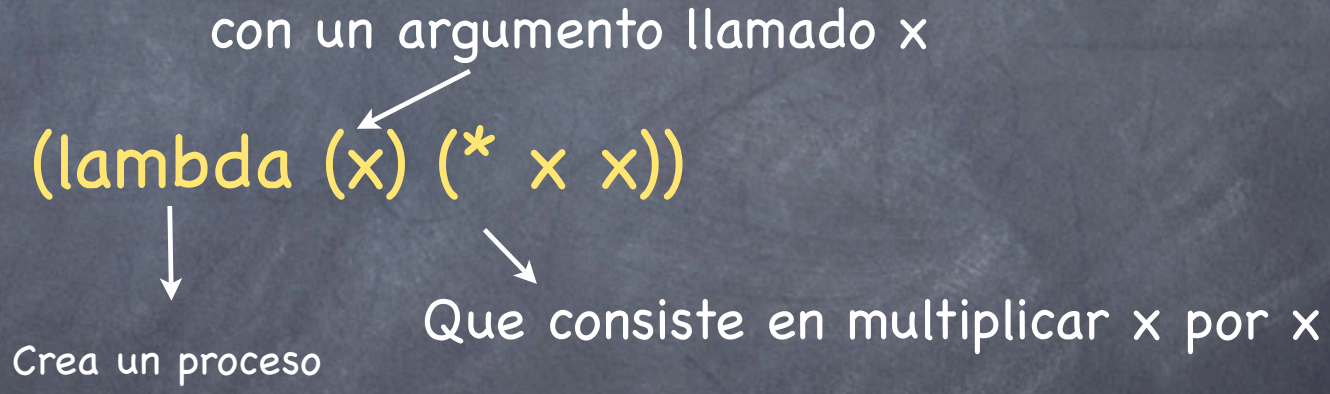
# Expresiones Lambda

En Scheme se definen los procedimientos mediante una expresion LAMBDA



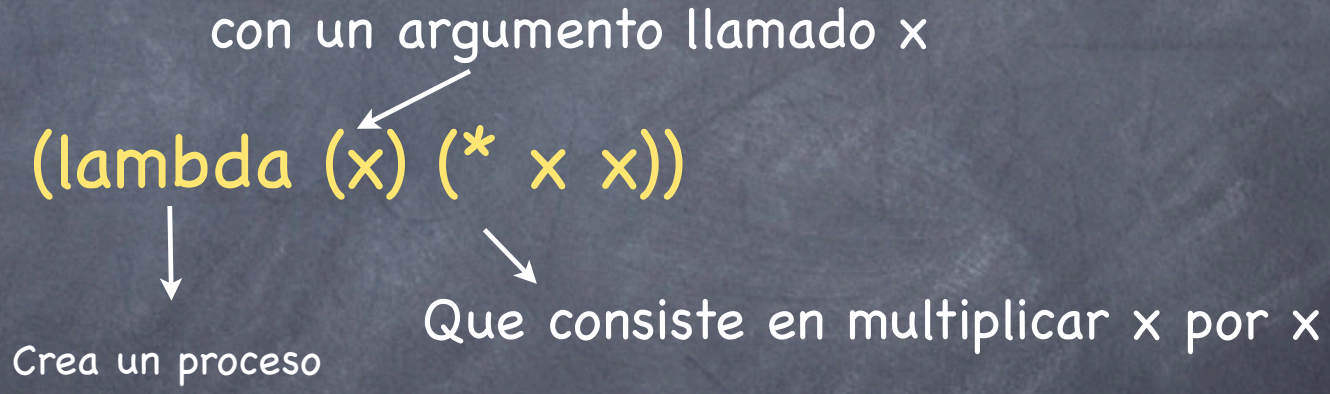
# Expresiones Lambda

En Scheme se definen los procedimientos mediante una expresion LAMBDA



# Expresiones Lambda

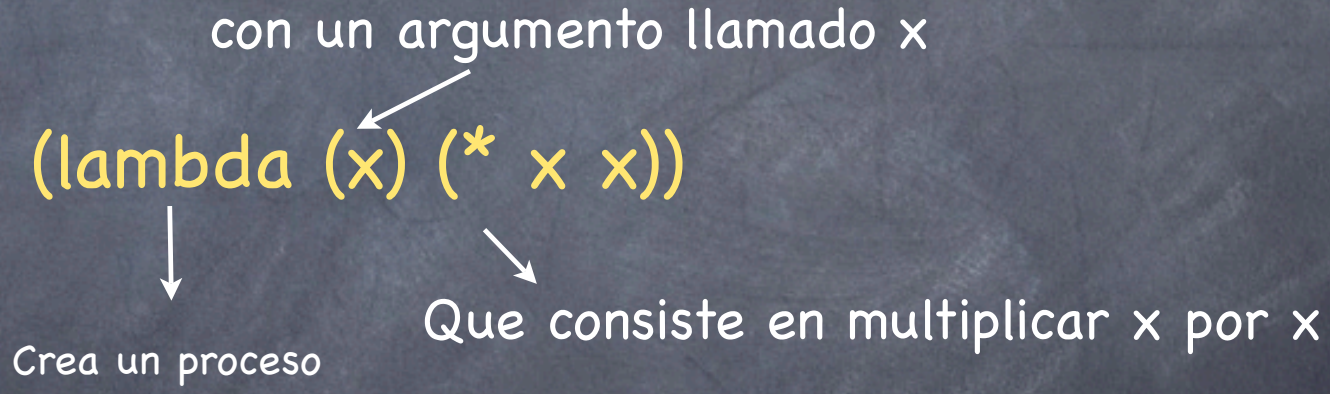
En Scheme se definen los procedimientos mediante una expresion LAMBDA




Dado un valor de x regresa el producto del valor por si mismo

# Expresiones Lambda

En Scheme se definen los procedimientos mediante una expresión LAMBDA




Note que la expresión Lamdba es una forma especial

# Expresión Lambda

Como ejecutamos un procedimiento creado por la expresión lambda?

Mediante una combinación

(operador op1 op2 op3...op4)

# Expresión Lambda

Como ejecutamos un procedimiento creado por la expresión lambda?

Mediante una combinación

(procedimiento argumentos)

# Expresión Lambda

Como ejecutamos un procedimiento creado por la expresión lambda?

Mediante una combinación

(+ 30 20 100)

# Expresión Lambda

Como ejecutamos un procedimiento creado por la expresión lambda?

Mediante una combinación

```
((lambda (x) (* x x))  5)
```

¿Cuáles son los procedimientos de primera clase

# Procedimientos de primera clase

- Pueden ser nombrados.
- Pueden ser argumentos de otros procedimientos
- Pueden ser devueltos como resultado de otro procedimiento
- Procedimientos almacenados en estructuras de datos

# Los procedimientos pueden ser nombrados

Abstracción

# Pueden ser nombrados

```
(define cuadrado
(lambda (x) (* x x)))
```

```
(define cubo (lambda
(x) (* x x)))
```

# Pueden ser nombrados

```
(define (cuadrado x) (* x x))

(define (cubo x) (* x x x))

> (cuadrado 345453)
119337775209

> (cubo 234423223)
1288255181220155937619056 7
```

# Procedimientos como argumentos de otros procedimientos

# Como argumentos de otros procedimientos

1 + 2 + 3 + 4 + 5...

PROCEDIMIENTO RECURSIVO

SUMA EL PRIMER TERMINO DE LA SERIE A LA SUMA DEL RESTO DE LOS ELEMENTOS DE LA SERIE

# Como argumentos de otros procedimientos

1 + 2 + 3 + 4 + 5...

A B

```
(define (suma-enteros a b)
  (if (> a b)
      0
      (+ a (suma-enteros (+ a 1) b))))
```

# Como argumentos de otros procedimientos

1 + 4 + 9 + 16 + 25...

A B

```
(define (suma-cuadrados a b)
  (if (> a b)
      0
      (+ (cuadrado a) (suma-enteros (+ a 1) b))))
```

# Como argumentos de otros procedimientos

```
(define (suma-enteros a b)
  (if (> a b)
      0
      (+ a (suma-enteros (+ a 1) b))))
```

```
(define (suma-cuadrados a b)
  (if (> a b)
      0
      (+ (cuadrado a) (suma-enteros (+ a 1) b))))
```

```
(define (suma a b termino siguiente)
  (if (> a b)
      0
      (+ (termino a) (suma (siguiente a) b termino siguiente))))
```

# Como argumentos de otros procedimientos

```
(suma 1 100 (lambda(x) x) (lambda(x) (+ x 1)))
```

```
(suma 1 100 (lambda(x) (* x x)) (lambda(x) (+ x 1)))
```

Pueden ser devueltos como resultado de otro procedimiento

# Devueltos como resultado de otros procedimientos

$$f(x) = 2x^3$$

```
(derivada (lambda(x) (* 2 (* x x x)))
#<procedure>
```

# Devueltos como resultado de otros procedimientos

La derivada de $f(x) = 2x^3$ evaluada en 5

((derivada (lambda(x) (* 2 (* x x x))) 5)

150

# Devueltos como resultado de otros procedimientos

$$f'(x) = \lim_{dx \mapsto 0} \frac{f(x + dx) - f(x)}{dx}$$

```
(define (derivada f)
(lambda(x) (/ (- (f (+ x dx)) (f x)) dx)))

(define dx 0.0001)
```

# Procedimientos almacenados en estructuras de datos

# Almacenados en estructuras de datos

Tabla de procedimientos

```
(define proc (list (cons 'cuadrado (lambda(x) (* x x)))
                   (cons 'cubo (lambda(x) (* x x x)))
                   (cons 'seno (lambda(x) (sin x)))))
```

```
> proc
((cuadrado . #<procedure>)
 (cubo . #<procedure>)
 (seno . #<procedure>))
```

# Almacenados en estructuras de datos

```
(define (ejecuta lista proceso x)
(cond ((null? lista) (display "procedimiento no
almacenado"))
((eq? (car (car lista)) proceso)
((cdr(car lista)) x))
(else (ejecuta (cdr lista) proceso x))))
```

# Almacenados en estructuras de datos

(ejecuta proc 'cubo 34234322)

40122242055438883554248

# Procedimientos de primera clase

- Pueden ser nombrados.
- Pueden ser argumentos de otros procedimientos
- Pueden ser devueltos como resultado de otro procedimiento
- Procedimientos almacenados en estructuras de datos

# Procesamiento simbólico

$$\frac{dc}{dx} = 0$$

$$\frac{dx}{dx} = 1$$

$$\frac{d(u+v)}{dx} = \frac{du}{dx} + \frac{dv}{dx}$$

$$\frac{d(uv)}{dx} = u\left(\frac{du}{dx}\right) + v\left(\frac{dv}{dx}\right)$$

# Procesamiento simbólico

$$\frac{dc}{dx} = 0$$

$$\frac{dx}{dx} = 1$$

$$\frac{d(u+v)}{dx} = \frac{du}{dx} + \frac{dv}{dx}$$

$$\frac{d(uv)}{dx} = u\left(\frac{du}{dx}\right) + v\left(\frac{dv}{dx}\right)$$

```
(define (derivada exp var)
  (cond
    ((espresion-es-numero?) ..)
    ((espresion-es-variable?) ..)
    ((espresion-es-suma?) ..)...)
    ((espresion-es-producto?) ..))
```

# Procesamiento simbólico

Espresión-es-número

$$\frac{dc}{dx} = 0$$

((number? exp) 0)

# Procesamiento simbólico

$$\frac{dx}{dx} = 1$$

```
((variable? exp)
 (if (misma-variable? exp var) 1 0))
```

# Procesamiento simbólico

$$\frac{d(u+v)}{dx} = \frac{du}{dx} + \frac{dv}{dx}$$

```
((suma? exp)
(crea-suma (derivada (primersumando exp) var)
           (derivada (segundosumando exp) var)))
```

# Procesamiento simbólico

$$\frac{d(uv)}{dx} = u\left(\frac{du}{dx}\right) + v\left(\frac{dv}{dx}\right)$$

```
((producto? exp)
 (crea-suma (crea-producto (primermultiplicador exp)
                           (derivada (segundomultiplicador exp) var))
            (crea-producto (segundomultiplicador exp)
                           (derivada (primermultiplicador exp) var))))
                    (else
 (error "expresion desconocida")))))
```

# Procesamiento simbólico

```
(define (derivada exp var)
  (cond ((number? exp) 0)
        ((variable? exp)
         (if (misma-variable? exp var) 1 0))
        ((suma? exp)
         (crea-suma (derivada (primersumando exp) var)
                    (derivada (segundosumando exp) var)))
        ((producto? exp)
         (crea-suma (crea-producto (primermultiplicador exp)
                                   (derivada (segundomultiplicador exp) var))
                    (crea-producto (segundomultiplicador exp)
                                   (derivada (primermultiplicador exp) var))))
        (else
         (error "expresion desconocida"))))
```

# ¿Qué sigue?

ML

Haskell

Mathematica

# Contacto

Dr. José Manuel Gómez Soto
Posgrado e Investigación
Universidad la Salle
México
www.ci.ulsa.mx/~jmgomez
jmgomezgoo@gmail.com