Lenguajes de Programación

CC41A

Victor Ramiro
vramiro@dcc.uchile.cl

Por qué estudiar Lenguajes?

- El objetivo de este curso es capacitar a los alumnos en los lenguajes de programación, tanto a nivel de su **semántica** e uso, como a nivel de su **implementación**.

- Para lo primero, el curso se centra en el estudio de **intérpretes de lenguajes de programación**, incluyendo en particular los paradigmas funcional y lógico, ya que éstos son en general desconocidos por los alumnos.

Por qué estudiar Lenguajes?

- Asimismo, se va llegando a mecanismos de lenguajes progresivamente más sofisticados a través de su uso y de su implementación (en el lenguaje Scheme). También se entregan elementos básicos relativos a técnicas de compilación e implementación eficiente.

Por qué estudiar Lenguajes?

- Al terminar el curso, se espera que el alumno tenga un entendimiento claro de la semántica de distintas variantes de lenguajes, siendo capaz de elegir un lenguaje de programación para un problema dado, aplicando los conceptos vistos en clase. Además el alumno tendrá un sólido manejo del lenguaje Scheme.

Por qué sirve un Lenguaje?

- Provee una forma para expresar algoritmos, de manera independiente de la máquina específica
- Provee una abstracción de alto nivel de una máquina compleja difícil de programar

Características deseables

- Rapidez de aprendizaje: Scheme vs. Cobol, Pl1
  - integridad conceptual: pocos conceptos
  - ortogonalidad: composición de las abstracciones, pocos casos especiales
- Expresividad: Smalltalk vs. Fortran
- Costos
  - rapidez de programación: Perl vs. Cobol, Java
  - legibilidad: Ml vs. Apl, perl write-only
  - mantenibilidad: Java vs. C
- Eficiencia: C vs. Basic
- Apoyo para la abstracción: tipos paramétricos
- Herramientas de verificación: assert y Ambientes de desarrollo: IDE, debugger
- API estándar para desarrollar aplicaciones: Java vs. C

Introducción a Scheme

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 #(2 3))`
- 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)
      0
      (* (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

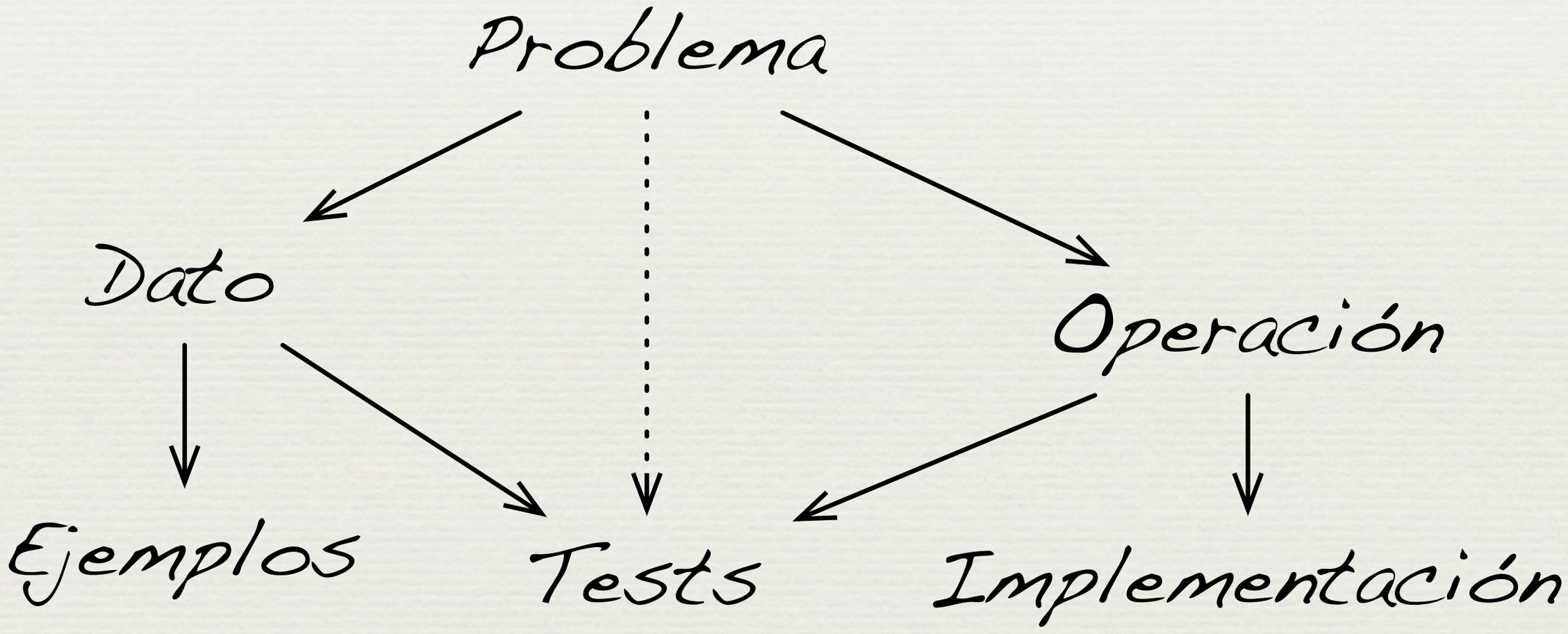
```
(list 'a "hola" 3) => (a "hola" 3)

(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



Axiomas de 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)`**

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

- Patron de Diseño:

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

```
;; Contract: suma-l : list-of-numbres -> 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
```

rev

```
;; 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)
```

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)))))
```