Escuela de Ingeniería de Sistemas y Computación

Curso: Estructura de Lenguajes

Agosto - Diciembre de 2001

El Paradigma de Programación Funcional

Juan Francisco Díaz Frias

Profesor Titular

Escuela de Ingeniería de Sistemas y Computación

Universidad del Valle

Plan

- Introducción
- Elementos de programación
- Procedimientos y los procesos que generan
- Procedimientos de Alto Orden
- Estructuras de Datos
- Modularidad, Objetos y Estado

Introducción

- Principios
- Elementos básicos
- Cómo definir funciones
- *Scheme*, un dialecto de LISP
- Un poco más sobre LISP
- Otros Lenguajes Funcionales

Principios

- El valor de una expresión depende sólo de los valores de sus subexpresiones, si las tiene.
  - ⋆ No efectos de borde No asignación

- Manejo de almacenamiento implícito
  - ⋆ Procedimiento de asignación de memoria especial
  - ⋆ Recolección de "basura "

- Funciones como "Ciudadanos de primera clase"
  - ⋆ Valor o argumento de una expresión
  - ⋆ Hacen parte de las estructuras de datos

Elementos Basicos de la Programación Funcional

- Las funciones

$$\begin{array}{rcl} Dominio & \stackrel{f}{\rightarrow} & Rango \\ x \in Domino & \mapsto & f(x) \in Rango \end{array}$$

- La composición de funciones

$$\begin{array}{rcccl} A & \stackrel{g}{\rightarrow} & B & \stackrel{f}{\rightarrow} & C \\ x & \mapsto & g(x) & \mapsto & f(g(x)) \end{array}$$

- Destacar:
  - ⋆ Noción de tipo
  - ⋆ Noción de abstracción funcional: $h(x) = f(g(x))$

Cómo Definir Funciones

- Tabulación o enumeración:

x	a	b	c
f(x)	1	1	2

- Reglas conocidas

$$Suc(x) = x + 1, Cubo(y) = y \times y \times y$$

- Composición

$$h(x) = Suc(Cubo(x)) = x^3 + 1$$

- Casos

$$max(x, y) = \begin{cases} x & \text{Si } x \geq y \\ y & \text{En caso contrario} \end{cases}$$

- Recursividad: $0! = 1, n! = n \times (n-1)!$Si $n > 0$

SCHEME, un Dialecto de LISP

- Qué es LISP
  - ⋆ LISP = LISt Processing
  - ⋆ Un lenguaje funcional para el procesamiento de listas
  - ⋆ Data de 1950: J. McCarthy
  - ⋆ Uno de los lenguajes más usados en el mundo
  - ⋆ Familia de dialectos
  - ⋆ Interpretado y compilado

- Por qué LISP
  - ⋆ Fácil disponibilidad
  - ⋆ Lenguaje principal en IA.
  - ⋆ Gran capacidad para manipulación de símbolos
  - ⋆ Procedimientos LISP $\equiv$ DATOS
    - ◦ Incrementa el poder de programación
    - ◦ Facilidad de escribir programas que manipulan otros programas como datos
- Por qué SCHEME
  - ⋆ Dialecto pequeño
  - ⋆ Funciones, verdaderamente, de primera clase

Un poco más sobre LISP

- Aplicaciones en IA:
  - ⋆ Sistemas expertos
    Razonamiento humano y Aprendizaje
    Interfaces en lenguaje natural, visión y habla

- Mitos
  - ⋆ LISP es lento
  - ⋆ Programas LISP son grandes
  - ⋆ Programas LISP requiere computadores caros
  - ⋆ LISP es dificil de leer, de depurar y de aprender

Otros Lenguajes Funcionales

- Otros dialectos de LISP
  - ⋆ IQLISP
  - ⋆ Common-LISP
  - ⋆ Franz-LISP
- Lenguajes funcionales modernos
  - ⋆ Lenguajes con tipos
  - ⋆ Familia Miranda: Evaluación perezosa
  - ⋆ Familia ML: Evaluación por valor. Por ejemplo CAML.

Elementos de programación

- Generalidades
- Expresiones simples
- Identificadores y el ambiente
- Evaluación de expresiones
- Definición de procedimientos
- Modelo de Substitución

- Modelo de Orden Normal

- Expresiones Condicionales y Predicados

- Procedimientos como cajas negras

Generalidades

- Elementos principales de un lenguaje:
  - ⋆ Expresiones primitivas
  - ⋆ Medios de combinación
  - ⋆ Medios de abstracción
- Elementos de la programación
  - ⋆ Procedimientos (o funciones)
  - ⋆ Datos

Expresiones Simples

- Los numerales

  >578

  578

- Procedimientos primitivos: +,-,*,/.

  >(+ 115 226)

  341

  >(- 1000 334)

  666

  >(* 5 99)

  495

  >(/ 10 6)

  1.66667

- Combinación: $\underbrace{(\underbrace{< op >}_{\text{Operador}} \overbrace{< op_1 >< op_2 > \ldots < op_n >}^{\text{Operandos}})}_{\text{NotaciónPrefija}}$

- Ventajas Notación prefija:
  - ⋆ Procedimientos con número arbitrario de argumentos

    >(+ 21 35 12 7)

    75

  - ⋆ Facilidad de anidamiento

    >(+ (* 3 5) (- 10 6))

    19

- Para facilitar la lectura,

$$(+\ (*\ 3\ (+\ (*\ 2\ 4)\ (+\ 3\ 5)))\ (+\ (-\ 10\ 7)\ 6))$$

se escribirá:

```
(+ (*  3
       (+ (*  2  4)
          (+  3  5)))
   (+ (-  10  7)
      6))
```

} Pretty printing

- Interpretador:Read-Eval-Print Loop

Identificadores y el Ambiente

- Identificador: identifica una variable cuyo valor es un objeto.

- Operador utilizado: *define*.

  >(define peso 75)

  peso

  Nota: En LISP toda función devuelve un valor

  >peso

  75

  >(* 2 peso)

  150

- En general:

  (define <ident> <expresión>)

- Más ejemplos:

  >(define pi 3.14159)

  pi

  >(define radio 10)

  radio

  >(define circunferencia (* 2 *pi radio*))

  circunferencia

  >circunferencia

  62.8318

- *define* es el medio más simple de abstracción en SCHEME.

- Guardar en memoria los pares nombre-objeto: *Ambiente Global*.

Evaluación de Expresiones

- En general la regla de evaluación de la expresión

$$(exp\ exp_1\ exp_2\ \ldots\ exp_n)$$

es la siguiente:

  - ⋆ **Evaluar** las subexpresiones $exp_1, \ldots, exp_n$:

$$\Rightarrow a_1, \ldots, a_n$$

  - ⋆ **Evaluar** la subexpresión exp :

$$\Rightarrow o$$

  - ⋆ **Aplicar** el operador o a los argumentos $a_1, \ldots, a_n$.

- Observaciones:

  - ⋆ Regla de evaluación es recursiva

  - ⋆ Evaluar es un proceso complicado

- Existen casos particulares, llamados *formas especiales*, con regla de evaluación propia.
  - ⋆ Numerales: números que representan
  - ⋆ Operadores predefinidos: Sucesión de instrucciones de máquina
  - ⋆ Identificadores: Objetos asociados en el ambiente
  - ⋆ *define*: Ver más adelante

Definición de Procedimientos

- Elevar al cuadrado:

  >(define (cuadrado x) (* x x))

  cuadrado

- En general:

  (define (<ident> <params. formales>) <cuerpo>)

  ⋆ <ident>: símbolo

  ⋆ <params. formales>: nombres usados en el cuerpo

  ⋆ <cuerpo>: expresión

- Continuando el ejemplo:

  >(cuadrado 11)

  121

  >(cuadrado (+ 6 5))

  121

  >(cuadrado (cuadrado 3))

  81

- Uso de procedimientos definidos para definir otros:

  > (define (suma_de_cuadrados x y)
  (+ (cuadrado x) (cuadrado y)))

  suma_de_cuadrados

  >(suma_de_cuadrados 3 4)

  25

Modelo de Substitución para Evaluación de Procedimientos

- Qué quiere decir:

  Aplicar el operador o a los argumentos $a_1, \ldots, a_n$?

  ⋆ Si o es primitivo:

  $\Rightarrow$ El interpretador sabe qué hacer

  ⋆ Si o es definido con *define*:

  $\Rightarrow$ **Aplicar** es **evaluar** el cuerpo del procedimiento con cada parámetro formal reemplazado por el correspondiente argumento.

- Ejemplo: evaluar (f 5) si f está definida por:

```
(define  (f a)
         (suma_de_cuadrados(+ a 1) (* a 2)))
```

La evaluación sigue los siguientes pasos:

⋆ Valor de f $\Rightarrow o$: (suma_de_cuadrados(+ a 1) (* a 2))
Valor de 5 $\Rightarrow a_1 : 5$
Aplicar o a a_1 consiste en
Evaluar (suma_de_cuadrados(+ 5 1) (* 5 2))

⋆ Valor de suma_de_cuadrados$\Rightarrow o$: (+ (cuadrado x) (cuadrado y))
Valor de (+ 5 1) $\Rightarrow a_1 : 6$
Valor de (* 5 2) $\Rightarrow a_2 : 10$
Aplicar o a a_1, a_2 consiste en
Evaluar (+ (cuadrado 6) (cuadrado 10))

⋆ Valor de + $\Rightarrow o$: instrucciones de máquina
Valor de (cuadrado 6) $\Rightarrow a_1 : 36$
Valor de (cuadrado 10) $\Rightarrow a_2 : 100$
Aplicar o a a_1, a_2 da 136.

- El modelo de substitución permite pensar sobre aplicación de procedimientos.

- El modelo de substitución no es suficientemente poderoso.

- El modelo de substitución es llamado **Orden aplicativo**.

- Otro modelo: **Orden normal**:
  - ⋆ Expresar cada definición en términos de las más simples
  - ⋆ Cuando todo sea primitivo: **Evaluar**
  - ⋆ Ejercicio: Qué pasa en el caso anterior?

Sobre el Modelo de Orden Normal

- Ejemplo: evaluar (f 5) si f está definida por:

```
(define  (f a)
         (suma_de_cuadrados(+ a 1) (* a 2)))
```

- Solución:

(f 5)
⇓
(suma_de_cuadrados(+ 5 1) (* 5 2))
⇓
(+ (cuadrado (+ 5 1)) (cuadrado (* 5 2)))
⇓
(+ (* (+ 5 1) (+ 5 1)) (* (* 5 2) (* 5 2)))
⇓
(+ (* 6 6) (* 10 10))
⇓
(+ 36 100)
⇓
136

Expresiones Condicionales y Predicados

- Forma especial *cond*:

$$\begin{array}{ll}(\text{cond} & (< p_1 >< e_1 >) \\ & (< p_2 >< e_2 >) \\ & \vdots \\ & (< p_n >< e_n >))\end{array}$$

Cada $(< p >< e >)$ es llamado una cláusula.

$< p >$ es un predicado : $\begin{cases} \text{falso} \equiv nil \\ \text{cierto} \equiv \text{Cualquier otro valor} \end{cases}$

- Evaluación de la forma *cond*:
  - ⋆ Evaluar consecutivamente los $< p_i >$ hasta encontrar el primero que evalúe a *cierto*.
    Supongamos que es $< p_k >$.
    El interpretador $\Rightarrow$ evaluación de $< e_k >$.
  - ⋆ Si todos los $< p_i >$ evaluaron a *falso*,
    el interpretador $\Rightarrow$ *nil*.

- Ejemplo: definir la función:

$$abs(x) = \begin{cases} x & \text{Si } x > 0 \\ 0 & \text{Si } x = 0 \\ -x & \text{Si } x < 0 \end{cases}$$

  - ⋆
```
(define  (abs x)
         (cond   ( (> x 0) x)
                 ( (= x 0) 0)
                 ( (< x 0) (- x))))
```

  - ⋆ Otra forma de hacerlo:
```
(define  (abs x)
         (cond   ( (< x 0) (- x))
                 (else x)))
```

    Nota: *else* es un símbolo especial, usado opcionalmente al final del *cond*.

- Otra forma de hacerlo:
```
(define  (abs x)
         (if       (< x 0)
                   (- x)
                   x))
```

- Forma especial *if*:

  (if <pred> <cons> <alt>)

- Evaluación de *if*:
  - ⋆ Evaluar el <pred> .
  - ⋆ Si es *cierto* ⇒ Evaluación de la <cons> .
  - ⋆ En caso contrario ⇒ Evaluación de la <alt> .

- Operadores lógicos:*and, or, not*.

Ejercicio: más sobre el Modelo de Orden Normal

- Considere los siguientes dos procedimientos:

```
(define (p) (p))
(define (test x y)
        (if (= x 0)
            0
            y ))
```

Evaluar

(test 0 (p))

utilizando como orden de evaluación para formas no especiales:

⋆ el orden aplicativo

⋆ el orden normal

Ejemplo: la Raíz Cuadrada

- Ilustrar:
  - ⋆ Función matemática v.s. Procedimiento funcional
  - ⋆ Declarativo v.s. Imperativo
  - ⋆ Qué v.s. Cómo
- Declaración:

$$\sqrt{x} = y : y \geq 0, \text{ y } y^2 = x$$

- Cómo calcular $\sqrt{x}$?

- Cómo calcular $\sqrt{x}$?
  - ⋆ Método de Newton ($\sqrt{2}$):

Adivinanza	Cociente	Promedio
1	$\frac{2}{1} = 2$	$\frac{2+1}{2} = 1.5$
1.5	$\frac{2}{1.5} = 1.3333$	$\frac{1.3333+1.5}{2} = 1.4167$
1.4167	$\frac{2}{1.4167} = 1.4118$	$\frac{1.4118+1.4167}{2} = 1.4142$
1.4142	...	...

  - ⋆ Definición del proceso general en SCHEME:

```
(define (raiz_cuad_iter adiv x)
        (if (buena_aprox? adiv x)
            adiv
            (raiz_cuad_iter (mejorar adiv x)
                            x)))
```

  - ⋆ Donde buena_aprox? y mejorar están definidos así:

```
(define (buena_aprox? adiv x)
        (< (abs (- (cuadrado adiv)
                   x))
           0.001))
(define (mejorar adiv x)
        (promedio adiv (/ x adiv)))
```

⋆ Y, la función que calcula la raíz cuadrada es:

```
(define (raiz_cuad x)
        (raiz_cuad_iter 1 x))
```

⋆ Ejemplos:

```
>(raiz_cuad 9)
3.0001
>(raiz_cuad (+ 100 37))
11.7047
>(cuadrado (raiz_cuad 1000))
1000.0003
```

⋆ **Nota:** Procesos iterativos sin *while*, ni *for*, ni ...

Ejercicio: porque IF Necesita una forma Especial de Evaluación?

- Definamos *if* a partir de *cond*:

```
(define (nuevo_if p cons alt)
        (cond (p cons)
              (else alt)))
```

  - ⋆ Evaluar las expresiones siguientes:
    - ∘ (nuevo_if (= 2 3) 0 5)
    - ∘ (nuevo_if (= 1 1) 0 5)
  - ⋆ Reescribir *raiz_cuad_iter* con *nuevo_if*. Funciona bien? Qué pasa?

Procedimientos como Cajas Negras

- El problema de calcular $\sqrt{x}$ fué descompuesto en subproblemas:

$$\text{raiz\_cuad} \begin{cases} \text{raiz\_cuad\_iter} \begin{cases} \text{mejorar} \{\text{promedio} \\ \text{buena\_aprox?} \begin{cases} \text{cuadrado} \\ \text{abs} \end{cases} \end{cases} \end{cases}$$

- Crucial: división en tareas modulares.

$$\Rightarrow \text{Procedimientos como cajas negras}$$

- Observaciones
  - ⋆ Nombres locales: procedimientos independientes de nombres de parámetros formales.

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

$$\equiv$$

```
(define (cuadrado y) (* y y))
```

⋆ Nombres locales: la x de *buena_aprox?* diferente de la x de *cuadrado*.

⋆ Un parámetro formal es una **variable acotada**.

x está acotada en *cuadrado*

cuadrado acota a x

⋆ x está acotada en una expresión si el valor de la expresión es el mismo cuando x es cambiado uniformemente por otro nombre dentro de la expresión.

⋆ Variable libre: no acotada.

⋆ Alcance de x: Conjunto de expresiones que la definen.

- Algunos problemas con *raiz_cuad* :
  - ⋆ Procedimientos separados y globales
  - ⋆ Problemas en proyectos grandes con programadores independientes

- Solución: Estructura de bloque.

```
(define (raiz_cuad x)
        (define (buena_aprox? adiv x)
                (< (abs (- (cuadrado adiv)
                           x))
                   0.001))
        (define (mejorar adiv x)
                (promedio adiv (/ x adiv)))
        ⋮
        (raiz_cuad_iter 1 x))
```

- Aún mejor: x global a todos los procs. locales.

Procedimientos y los procesos que generan

- Generalidades
- Recursión lineal e iteración
- Recursión de árbol
- Recursión y complejidad

Generalidades

- Procedimientos v.s. Procesos computacionales:
  - ⋆ Un procedimiemto especifica la evolución de un proceso computacional
  - ⋆ Las reglas de interpretación de un procedimiento determinan el siguiente estado del proceso.
- Objetivo: Hacer observaciones globales sobre el comportamiento de un proceso.
- Procesos más comunes:
  - ⋆ Recursión lineal e iteración
  - ⋆ Recursión en árbol

Recursión Lineal e Iteración

- Consideremos la función *Factorial*:

$$n! = 1 \times 2 \times 3 \ldots \times (n-1) \times n$$

- Diferentes formas de calcularla:
  - ⋆ Observando que $n! = n \times (n-1)!$

```
(define (factorial n)
        (if (= 1 n)
            1
            (* n (factorial (- n 1)))))
```

  - ⋆ Multiplicando en su orden 1,2,. . . ,n

```
(define (fact n) (fact_iter 1 1 n))
(define (fact_iter cont prod n)
        (if (> cont n)
            prod
            (fact_iter (+ cont 1)
                       (* prod cont)
                       n)))
```

- Procesos generados para (factorial 6) y (fact 6):
  - ⋆ Para el caso de factorial

(factorial 6)
(* 6 (factorial 5))
(* 6 (* 5 (factorial 4)))
(* 6 (* 5 (* 4 (factorial 3))))
(* 6 (* 5 (* 4 (* 3 (factorial 2)))))
(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1))))))
(* 6 (* 5 (* 4 (* 3 (* 2 1)))))
(* 6 (* 5 (* 4 (* 3 2))))
(* 6 (* 5 (* 4 6)))
(* 6 (* 5 24))
(* 6 120)
720

  - ⋆ Para el caso de fact :

(fact 6)
(fact_iter 1 1 6)
(fact_iter 2 1 6)
(fact_iter 3 2 6)
(fact_iter 4 6 6)
(fact_iter 5 24 6)
(fact_iter 6 120 6)
(fact_iter 7 720 6)

- Comparemos los dos procesos:

	factorial	fact
Tiempo	$\sim 2n$	$\sim n$
Forma	Expansión-Contracción	Constante
Espacio	$\sim n$	$\sim$cte
	$\Uparrow$	$\Uparrow$
	Recursivo Lineal	Iterativo Lineal

- OJO!

Proceso Recursivo $\neq$ Procedimiento Recursivo

Ejercicio

Considere las dos definiciones siguientes:

- ```
  (define (+ a b)
          (if (= a 0)
              b
              (1+ (+ (1- a)
                     b))))
  ```

- ```
  (define (+ a b)
          (if (= a 0)
              b
              (+ (1 - a)
                 (1 + b))))
  ```

donde 1+ y 1- son dos funciones predefinidas que calculan el sucesor y el predecesor de un entero, respectivamente.

Ilustrar el proceso generado por cada procedimiento al evaluar (+ 4 5). Cómo son estos procesos?

Recursión de Arbol

- Consideremos la función de *Fibonacci*:

$$fib(n) = \begin{cases} 0 & \text{Si } n = 0 \\ 1 & \text{Si } n = 1 \\ fib(n-1) + fib(n-2) & \text{Sino} \end{cases}$$

- Aplicando directamente la definición:

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

- El proceso generado al calcular (fib 5):

- Observaciones:
  - ⋆ Tiempo $\equiv$ Nodos del árbol
    Espacio $\equiv$ Profundidad del árbol
    Fácil de programar
    mayor eficiencia $\Rightarrow$ menos natural

- Una solución más eficiente:
  - ⋆ Observar que: $0, 1, \underbrace{1, 2, 3}_{+}, \underbrace{5, 8, 13}_{+}, 21, \ldots$

  - ⋆ Dos acumuladores y un contador:

```
(define (fib_rapido n) (fib_iter 1 0 n))

(define (fib_iter a b cont)
        (if (= cont 0)
            b
            (fib_iter (+ a b) a (- cont 1))))
```

Otro Ejemplo de Recursión de Arbol

- Problema: Cuantas formas diferentes existen de completar \$100 con monedas de \$1, \$5, \$10, \$25, \$50 ?

- Clave 1:

$$\underbrace{\#\text{ de formas de cambiar } a \text{ con } n \text{ clases de monedas}}$$

$$\begin{array}{l} \#\text{ de formas de cambiar} \\ a \text{ con } n-1 \text{ clases de mo-} \\ \text{nedas} \end{array} \quad + \quad \begin{array}{l} \#\text{ de formas de cambiar} \\ a-d \text{ con } n \text{ clases de mo-} \\ \text{nedas} \end{array}$$

- Clave 2: Casos degenerados
  - ⋆ Si a es 0, responder 1
  - ⋆ Si $a < 0$, responder 0
  - ⋆ Si n es 0, responder 0

- Solución:

```
(define (contar_cambio cantidad)
        (cc cantidad 5))

(define (cc cantidad tip_de_mon)
        (cond ((= cantidad 0) 1)
              ((or (< cantidad 0)
                   (= tip_de_mon 0)) 0)
              (else (+ (cc (- cantidad
                              (denom tip_de_mon))
                           tip_de_mon)
                       (cc cantidad
                           (- tip_de_mon 1))))))

(define (denom tip_de_mon)
        (cond ((= tip_de_mon 1) 1)
              ((= tip_de_mon 2) 5)
              ((= tip_de_mon 3) 10)
              ((= tip_de_mon 4) 2)
              ((= tip_de_mon 5) 50)))
```

- Observaciones:
  - ⋆ contar_cambio genera un proceso de recursión de árbol.
  - ⋆ El proceso tiene redundancias: Cómo evitarlas?
  - ⋆ En general: Cómo evitar las redundancias en los procesos de recursión de árbol?
    Idea: Tabulación de redundancias

Ejercicio

Diseñar un procedimiento que genere un proceso iterativo para resolver el problema precedente.

Mas Sobre Recursión y Complejidad

- Consideremos el problema de calcular b^n.

- Idea: utilizar la definición:

$$b^0 = 1$$

$$b^n = bb^{n-1}, n > 0$$

La solución, inmediata es:

```
(define (exp b n)
        (if (= n 0)
            1
            (* b (exp b (- n 1)))))
```

El proceso generado es recursivo lineal, con requerimiento lineal de espacio.

- Otra idea: estilo factorial

```
(define (expt b n)
        (exp_iter b n 1))

(define (exp_iter b cont prod)
        (if (= cont 0)
            prod
            (exp_iter b
                      (- cont 1)
                      (* b prod))))
```

El proceso generado es iterativo lineal, con requerimiento constante de espacio.

- Aún mejor:
  - ⋆ Utilizar que:

$$b^8 = (b^4)(b^4)$$

$$b^4 = (b^2)(b^2)$$

$$b^2 = bb$$

⋆ En general:

$$b^n = (b^{\frac{n}{2}})(b^{\frac{n}{2}}), \text{Si } n \text{ es par}$$

$$b^n = bb^{n-1}, \text{Si } n \text{ es impar}$$

⋆ Solución:

```
(define (f_exp b n)
        (cond ((= n 0) 1)
              ((par? n) (cuadrado (f_exp b (/ n 2))))
              (else (* b (f_exp b (- n 1))))))

(define (par? n)
        (= (remainder n 2) 0))
```

⋆ El proceso generado es iterativo logarítmico, con requerimiento logarítmico de espacio.

Ejercicio

Comparar el tiempo de evaluación de 1^{100000} entre cada una de las soluciones propuestas. Utilizar la función *runtime* que da el tiempo, en milisegundos, del reloj de la máquina.

Procedimientos de Alto Orden

- Generalidades

- Procedimientos como parámetros

- Uso de *lambda* y *let*

- Procedimientos como respuesta

Generalidades

- Los parámetros de los procedimientos no son solo números.

- Frecuentemente el mismo patrón de programación es usado en diferentes procedimientos.

- Expresar esos patrones como conceptos implica poder pasar procedimientos como parámetros.

$\Rightarrow$ Procedimientos de alto orden

Procedimientos como Parametros

- Considere los tres procedimientos siguientes para calcular:

$$a + (a+1) + (a+2) + \ldots + b$$

$$a^2 + (a+1)^2 + (a+2)^2 + \ldots + b^2$$

$$a^2 + (a+2)^2 + (a+4)^2 + \ldots + c^2, (b-1) \leq c \leq b$$

```
(define (suma_enteros a b)
        (if (> a b)
            0
            (+ a (suma_enteros (+ a 1) b))))

(define (suma_cuads a b)
        (if (> a b)
            0
            (+ (cuadrado a) (suma_cuads (+ a 1) b))))

(define (suma_alt a b)
        (if (> a b)
            0
            (+ (cuadrado a) (suma_alt (+ a 2) b))))
```

- El esquema general es:

```
(define (<nombre> a b)
        (if (> a b)
            0
            (+ (<term> a)
               (<nombre> (<prox> a) b))))
```

En SCHEME

⇓

```
(define (suma term a prox b)
        (if (> a b)
            0
            (+ (term a)
               (suma term (prox a) prox b))))
```

- Por ejemplo:

```
(define (suma_enteros a b)
        (suma ident a 1+ b))

(define (ident x) x)
```

Ejercicios

- El procedimiento *suma* genera un proceso recursivo lineal. Escribir un procedimiento *suma* que genere más bien un proceso iterativo lineal.

- Escriba un procedimiento *producto* análogo al procedimiento *suma*. Defina *factorial* en función de este nuevo procedimiento.

Uso de *lambda* y *let*

- Existe una manera de definir funciones anónimas:

  (lambda (<Parámetros formales>) <cuerpo>)

  Por ejemplo:

  (lambda (x) x)

- Ejemplos:

  >($\underbrace{\text{(lambda (x) x)}}_{Procedimiento}$ 3)

  3

  >($\underbrace{\text{(lambda (x) (+ x 2))}}_{Procedimiento}$ (+ 3 5))

  10

- Más ejemplos:

  (define (suma_enteros a b)
  (suma (lambda (x) x) a 1+ b))

- Otro uso de *lambda*: def. de variables locales.
  - ⋆ Calcular
    $f(x,y) = x\underbrace{(1+xy)^2}_{a} + y\underbrace{(1-y)}_{b} + \underbrace{(1+xy)}_{a}\underbrace{(1-y)}_{b}$
  - ⋆ Ayudaría tener a, b como variables intermedias:

```
(define (f x y)
        (define a (+ 1 (* x y)))
        (define b (- 1 y))
        (+ (* x (cuadrado a))
           (* y b)
           (* a b)))
```

  - ⋆ Una alternativa:

```
(define (f x y)
        (define (g a b)
                (+ (* x (cuadrado a))
                   (* y b)
                   (* a b)))
        (g (+ 1 (* x y))
           (- 1 y)))
```

⋆ Mejor si g es anónima:

```
(define (f x y)
        ((lambda (a b)
                 (+ (* x (cuadrado a))
                    (* y b)
                    (* a b))) (+ 1 (* x y))
                              (- 1 y)))
```

⋆ Por comodidad sintáctica: *let*

```
(define (f x y)
        (let ((a (+ 1 (* x y)))
              (b (- 1 y)))
             (+ (* x (cuadrado a))
                (* y b)
                (* a b))))
```

⋆ La forma general del *let* es:

(let ((< v_1 > < e_1 >)
(< v_2 > < e_2 >)
⋮
(< v_n > < e_n >))
<cuerpo>)

Sintaxis alternativa para:

((lambda (< v_1 > < v_2 > . . . < v_n >)
<cuerpo>) < e_1 >
< e_2 >
⋮
< e_n >)

⋆ OJO! *let* no es un nuevo mecanismo.

- Ventajas de *let* frente al *define*:
  - ⋆ Con *define*, alcance global
    Con *let*, alcance local al cuerpo del *let*
    Ejemplo:

```
(+ (let ((x 3))
         (+ x (* x 10)))
    x)
```

  - ⋆ .

Paralelismo del *let*

v.s.

Secuencialidad del *define*:

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

Qué valor devuelve la expresión anterior?

Procedimientos como Respuesta

- Calcular $f'(x)$:

$$f'(x) \approx \frac{f(x+dx) - f(x)}{dx}, dx \to 0$$

Esto se puede expresar:

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

- Definimos:

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

- >((deriv cube 0.001) 5)
  75.015

Estructuras de Datos

- Generalidades
- Atomos y pares
- Representación de sucesiones
- Operaciones sobre listas
- Representando árboles
- Símbolos y QUOTE

Generalidades

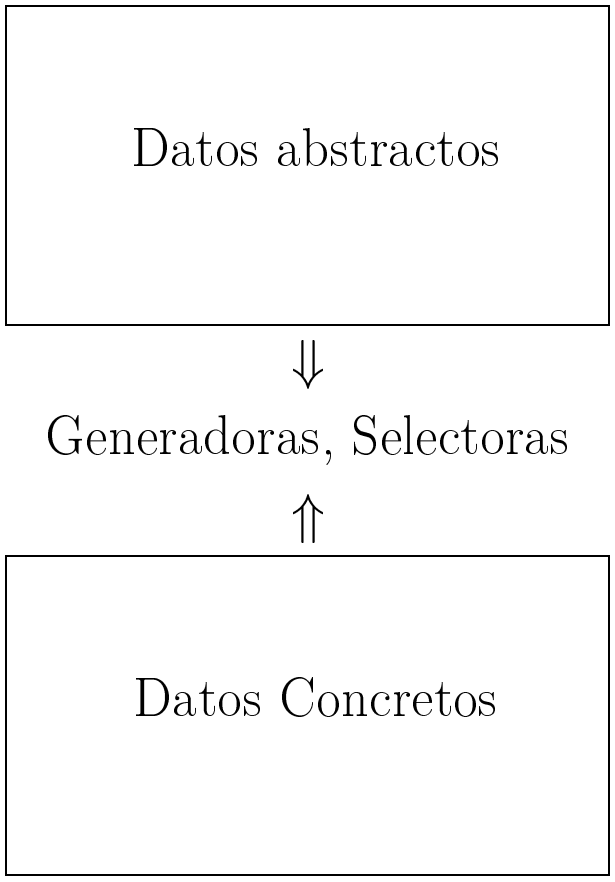
- Abstracciones de estructuras de datos simples en estructuras de datos más complejas.

- Para qué estructuras de datos compuestas?
  - ⋆ Elevar el nivel conceptual
    Incrementar la modularidad
    Fortalecer el poder expresivo del lenguaje

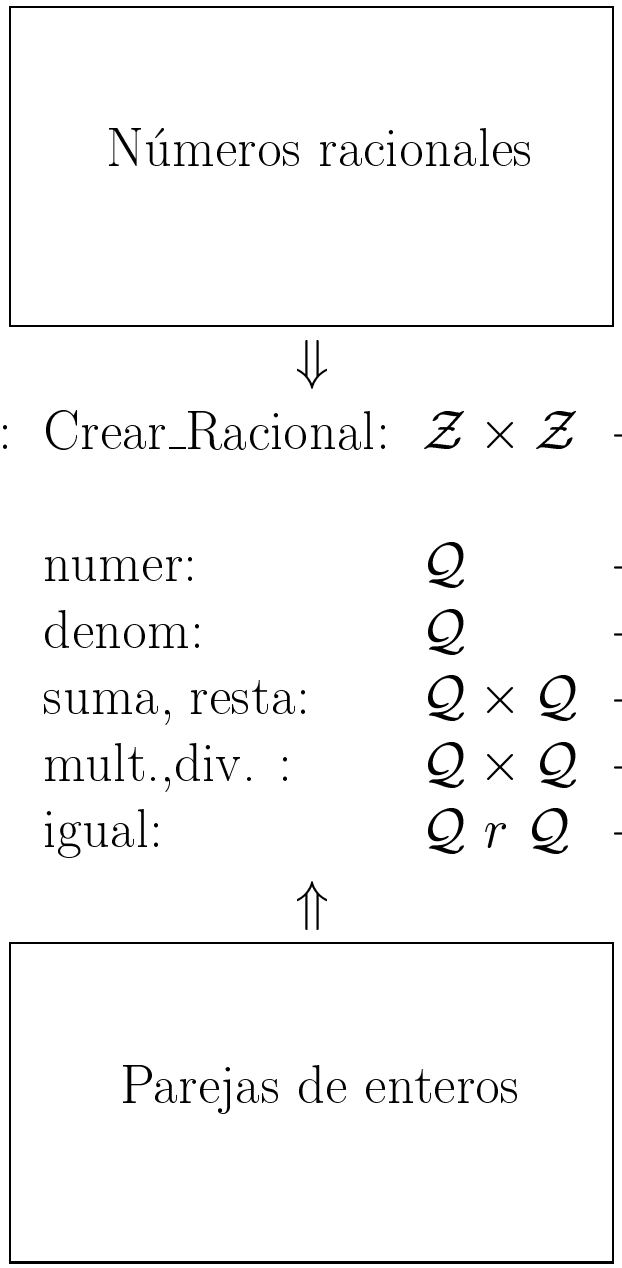
- Metodología: Tipos abstractos de datos.
  Permite independizar
  - ⋆ Cómo son usadas las estructuras de datos.
  - ⋆ Cómo son implementadas las estructuras de datos.

Sobre los Tipos Abstractos de Datos

- IDEA: Estructurar los programas para que operen sobre "datos abstractos"

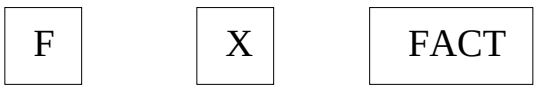


- Ejemplo:

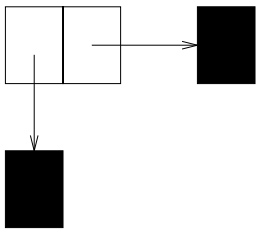

Números racionales
Generadoras: Crear_Racional: $\mathcal{Z} \times \mathcal{Z} \rightarrow \mathcal{Q}$
Selectoras: numer: $\mathcal{Q} \rightarrow \mathcal{Z}$
denom: $\mathcal{Q} \rightarrow \mathcal{Z}$
suma, resta: $\mathcal{Q} \times \mathcal{Q} \rightarrow \mathcal{Q}$
mult.,div. : $\mathcal{Q} \times \mathcal{Q} \rightarrow \mathcal{Q}$
igual: $\mathcal{Q}\ r\ \mathcal{Q} \rightarrow$ Booleano
Parejas de enteros

Atomos y Pares

- 2 tipos de estructuras:
  - ⋆ Atomos
    - ◦ Números
    - ◦ Símbolos
    - ◦ Todo lo que no sea un *par*
    - ◦ Ejemplo: X, Y, F, FACT, EXP, 1, 290, "hola"
  - ⋆ Pares
    - ◦ Parejas de Estructuras
    - ◦ Notación (par-punto): (.)
  - ⋆ Notación (Caja-apuntador):
    - ◦ Atomos



- Pares



- Constructora de pares: *cons*
  - ⋆ Ejemplos:

    >(define x (cons 1 2))

    x

    >x

    (1 . 2)

    >(cons 3 x)

    (3 . (1 . 2))

    >(cons x 3)

    ((1 . 2) . 3)

  - ⋆ En general

    (cons x y) produce el par (x . y)

- Selectoras de pares: *car, cdr*

  ⋆ Ejemplos:

  >(car x)

  1

  >(cdr x)

  2

  ⋆ En general

  (car (cons x y)) produce x
  (cdr (cons x y)) produce y

- Toda estructura más compleja es construida a partir de átomos y pares.

- Función ⇒ Diferentes procedimientos
  Estructura de datos ⇒ Diferentes representaciones

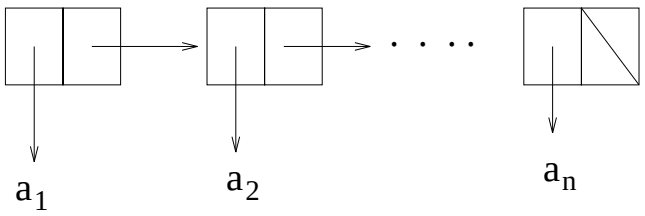
  ⇒ EFICIENCIA

Representación de Sucesiones

- Sucesión: colección ordenada de objetos.

- Una representación: Listas.

$$a_1, a_2, \ldots, a_n \Rightarrow (a_1.(a_2.\ldots(a_n.nil)\ldots))$$
$$\equiv (\text{cons } a_1 \; (\text{cons } a_2 \; \ldots \; (\text{cons } a_n \; nil)\ \ldots))$$



- Primitiva para crear listas: *list*

⋆ Ejemplo:
>(list 1 2 3 4)
(1 2 3 4)
>(list (list 1 2) (list 3 4))
((1 2) (3 4))

⋆ (list a_1 a_2 ... a_n) ≡ expresión precedente.

⋆ Ojo: (list a_1, a_2) ≠ (cons a_1 a_2)

- Si l es una lista,
  ⋆ (car l) es el primer elemento de la lista
  (cdr l) es la sublista resultante de eliminar el primer elemento
  caddr, caaadr, etc ...

- El símbolo *nil* es la lista vacía.
  El predicado *null?* permite saber si una lista es vacía o no.

Operaciones Sobre Listas

- Calcular el tamaño de una lista.

$$tam(l) = \begin{cases} 0 & \text{Si } l \text{ es vacía} \\ n & \text{Si } l = (a_1\ a_2\ \ldots\ a_n) \end{cases}$$

  - ⋆ De manera recursiva:

$$tam(l) = \begin{cases} 0 & \text{Si } l \text{ es vacía} \\ 1 + tam((a_2\ \ldots\ a_n)) & \text{Si } l = (a_1\ a_2\ \ldots\ a_n) \end{cases}$$

  - ⋆ Lo que nos da el procedimiento:

```
(define (tam l)
        (if (null? l)
            0
            (+ 1 (tam (cdr l)))))
```

  - ⋆ Ejemplo:

```
>(define pares (list 2 4 6 8))
pares
>(tam pares)
4
```

- Dadas dos listas l_1 y l_2 calcular la lista resultante de concatenar las dos listas.

$$concat(l_1, l_2) = \begin{cases} l_1 & \text{Si } l_2 \text{ es vacía} \\ l_2 & \text{Si } l_1 \text{ es vacía} \\ (a_1 \ldots a_n b_1 \ldots b_m) & \text{Si } l_1 = (a_1 \ \ldots \ a_n) \\ & \text{y } l_2 = (b_1 \ \ldots \ b_m) \end{cases}$$

  ⋆ De manera recursiva:

$$concat(l_1, l_2) = \begin{cases} l_2 & \text{Si } l_1 \text{ es vacía} \\ cons(\ car(l_1), & \\ \quad concat(cdr(l_1), l_2)) & \text{Si no} \end{cases}$$

  ⋆ Lo que nos da:

```
(define (concat l1 l2)
        (if (null? l1)
            l2
            (cons (car l1)
                  (concat (cdr l1)
                          l2))))
```

Ejercicios

- Definir la función *last* que dada una lista devuelva el último elemento de la lista:

$$last((a_1 \ldots a_n)) = a_n$$

- Definir la función *reverse* que dada una lista devuelva la lista en orden inverso:

$$reverse((a_1 \ldots a_n)) = (a_n \ldots a_1)$$

- Hacer una versión iterativa de *concat*. Qué problemas encuentra?

Mas Ejemplos Sobre Listas

- Dada una lista de números, construir la lista de los cuadrados de esos números.

```
(define (cuad_list l)
        (if (null? l)
            nil
            (cons (cuadrado (car l))
                  (cuad_list (cdr l)))))
```

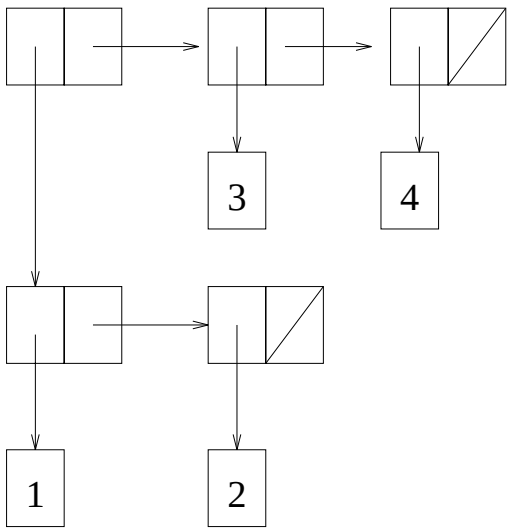
- Ejercicios:
  - ⋆ Tratar de escribir una versión iterativa de *cuad_list*. Qué problemas encuentra?
  - ⋆ Escriba un procedimiento de alto orden *aplicar_car* tal que:

$$\text{aplicar\_car}(f, (a_1 \ldots a_n)) = (f(a_1) \ldots f(a_n))$$

Representando Arboles

- Una manera de modelar árboles: listas de listas.

- Por ejemplo: ((1 2) 3 4) en notación caja-apuntador:



- Un árbol es entonces:
  - ⋆ Un átomo, o
  - ⋆ Una lista de subárboles.

- Ejercicio: Dibujar el árbol:

  (list 1 (list 2 (list 3 4)))

- Recursión: manera natural de operar sobre árboles. Una nueva primitiva: *atom?*.

- Ejemplo: Contar las hojas de un árbol.

$$hojas(a) = \begin{cases} 0 & \text{Si } a \text{ es vacío} \\ 1 & \text{Si } a \text{ es una hoja} \\ hojas(car(a)) + & \\ hojas(cdr(a)) & \text{si no} \end{cases}$$

Lo que sugiere el procedimiento:

```
(define (hojas a)
        (cond ((null? a) 0)
              ((atom? a) 1)
              (else (+ (hojas (car a))
                       (hojas (cdr a))))))
```

Simbolos y *QUOTE*

- Trabajar con cualquier clase de símbolos y no solo con números.

(a b c d)

((carlos 10) (pilar 15) (sonia ?) (jose 23))

- Cualquier expresión es una lista de listas de símbolos.

(* (+ 23 45) (+ x 9))

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

- Cómo construir la lista (a b) ?
  (list a b) no funciona. Por qué ?

- *quote*: indicar al evaluador que lo que sigue son datos.
  $'a \equiv$ (quote a)

- Ejemplos:

  >(define a 1)

  a

  >(define b 2)

  b

  >(list a b)

  (1 2)

  >(list *'a 'b*)

  (a b)

  >(list *'a b*)

  (a 2)

  >(car '(a b c))

  a

  >(cdr '(a b c))

  (b c)

- Nuevo predicado: *eq?*

  (eq? s_1 s_2) dice si el s_1 es igual a s_2 como símbolos.

Ejemplo: Derivación Simbolica

- Queremos hacer un procedimiento tal que:

$$\xrightarrow{f(x)} \boxed{\text{DERIV}} \xrightarrow{f'(x)}$$

  Por ejemplo:

$$\xrightarrow{ax^2+bx+c} \boxed{\text{DERIV}} \xrightarrow{2ax+b}$$

- Estructura abstracta de las expresiones algebráicas (ExpAlg):

Constantes	$\subset$ ExpAlg
Variables	$\subset$ ExpAlg

  Constructoras:

haga-suma:	ExpAlg $\times$ ExpAlg $\rightarrow$ ExpAlg
haga-prod:	ExpAlg $\times$ ExpAlg $\rightarrow$ ExpAlg

Analizadoras:

constante?:	ExpAlg	$\rightarrow$	Boolean
variable?:	ExpAlg	$\rightarrow$	Boolean
suma?:	ExpAlg	$\rightarrow$	Boolean
producto?:	ExpAlg	$\rightarrow$	Boolean
misma-var?:	Var $\times$ Var	$\rightarrow$	Boolean

Destructoras:

sumando-1:	ExpAlg	$\rightarrow$	ExpAlg
sumando-2:	ExpAlg	$\rightarrow$	ExpAlg
multip-1:	ExpAlg	$\rightarrow$	ExpAlg
multip-2:	ExpAlg	$\rightarrow$	ExpAlg

- Reglas de derivación:
  - $\star$ $\frac{dc}{dx} = 0$, para c const. o var. dif. de x
  - $\star$ $\frac{dx}{dx} = 1$
  - $\star$ $\frac{d(u+v)}{dx} = \frac{du}{dx} + \frac{dv}{dx}$ $\Leftarrow$ Recursiva

⋆ $\frac{d(uv)}{dx} = \frac{du}{dx}v + \frac{dv}{dx}u$ $\Leftarrow$ Recursiva

- Un procedimiento Lisp:

```
(define (deriv exp var)
        (cond ((constante? exp) 0)
              ((variable? exp)  (if (misma-var? exp var)
                                    1
                                    0 ))
              ((suma? exp)      (haga-suma (deriv (sumando-1 exp)
                                                  var)
                                           (deriv (sumando-2 exp)
                                                  var)))
              ((producto? exp) (haga-suma (haga-prod (deriv (mult-1 exp)
                                                            var)
                                                     (mult-2 exp))
                                          (haga-prod (deriv (mult-2 exp)
                                                            var)
                                                     (mult-1 exp))))))
```

- Solo falta implementar las funciones de EXPALG:
  - ⋆ haga-suma: ExpAlg $\times$ ExpAlg $\rightarrow$ ExpAlg

```
(define (haga-suma e1 e2)
        (list '+ e1 e2))
```

⋆ haga-prod: ExpAlg × ExpAlg → ExpAlg

```
(define (haga-prod e1 e2)
        (list '* e1 e2))
```

⋆ constante?: ExpAlg → Boolean

```
(define (constante? e)
        (number? e))
```

⋆ variable?: ExpAlg → Boolean

```
(define (variable? e)
        (symbol? e))
```

⋆ suma?: ExpAlg → Boolean

```
(define (suma? e)
        (if (atom? e)
        nil
            (eq? (car e) '+)))
```

⋆ producto?: ExpAlg → Boolean

```
(define (producto? e)
        (if (atom? e)
        nil
            (eq? (car e) '*)))
```

⋆ misma-var?: Var × Var → Boolean

```
(define (misma-var e1 e2)
        (and (variable? e1)
             (variable? e2)
             (eq? e1 e2)))
```

⋆ sumando-1: ExpAlg → ExpAlg

```
(define (sumando-1 e)
        (if (suma? e)
            (cadr e)
            nil))
```

⋆ sumando-2: ExpAlg → ExpAlg

```
(define (sumando-2 e)
        (if (suma? e)
            (caddr e)
            nil))
```

⋆ multip-1: ExpAlg → ExpAlg

```
(define (multip-1 e)
        (if (prod? e)
            (cadr e)
            nil))
```

⋆ multip-2: ExpAlg → ExpAlg

```
(define (multip-2 e)
        (if (prod? e)
            (caddr e)
            nil))
```

- Ejemplos:
  >(deriv '(+ x 3) 'x)
  (+ 1 0)
  >(deriv '(* x y) 'x)
  (+ (x 0) (* 1 y))*
  Probar con (deriv '(* (* x y) (+ x 3)) 'x)

  Respuestas no simplificadas!!!

- Ejercicio: Mejorar *haga-suma, haga-prod* de manera que:
  a+0=0+a sea a
  a*0=0*a sea 0
  a*1=1*a sea a

Modularidad, Objetos y Estado

- Generalidades
- Asignación y estado local
- Costos de introducir la asignación
- Nuevo modelo de evaluación: modelo de ambientes
- Reglas de evaluación

Generalidades

- Estrategia de diseño de un sistema:
  Basar la estructura del programa en la del sistema.
  Objeto del sistema ⇒ Objeto computacional
  Acción del sistema ⇒ Operación simbólica en el modelo

  ⇓

  ⋆ Extensión fácil del sistema

  ⋆ Modularidad

- Sistema: Colección de objetos distintos cuyo comportamiento puede cambiar en el tiempo.

  ⇓

  Objetos que cambian pero mantienen su identidad.

Asignación y Estado Local

- Un objeto tiene estado si su comportamiento depende de su historia.

  Ejemplo: Cuenta de banco, al tratar de retirar una cantidad.

- Objeto del sistema con estado $\equiv$ Objeto comput. con variables locales.

- Objetos del sistema cambian de valor con el tiempo

  $\Rightarrow$ Objetos comput. deben poder cambiar su valor mientras el programa corre.

$$\Downarrow$$

Necesitamos la asignación

- Ejemplo: hacer un procedimiento *retirar* con un argumento *cantidad*, tal que:
  - ⋆ Devuelve el saldo de la cuenta después del retiro, si hay fondos suficientes.
  - ⋆ Devuelve el mensaje "Fondos Insuficientes" en caso contrario.
  - ⋆ Por ejemplo, si incialmente hay $100.000:
    >(retirar 25000)
    75000
    >(retirar 25000)
    50000
    >(retirar 60000)
    Fondos Insuficientes
    >(retirar 15000)
    35000
    Nota: La misma expresión, evaluada dos veces, da resultados diferentes!!!

⋆ Una solución: tener una variable global *saldo*

```
(define saldo 100000)

(define (retirar cantidad)
        (if (>= saldo cantidad)
            (sequence (set! saldo
                            ( - saldo cantidad))
                      saldo)
            "Fondos Insuficientes"))
```

⋆ Forma especial Asignación:

(set! <nombre> <valor>)

⋆ Forma especial Secuenciación:

(sequence $< exp_1 > \ldots < exp_k >$)

⋆ Esta solución funciona, pero *saldo* es global.

⋆ Para que *saldo* sea local al procedimiento:

```
(define nuevo-retirar
        (let ((saldo 100000))
            (lambda (cantidad)
                    (if (>= saldo cantidad)
                        (sequence (set! saldo
                                        (- saldo cantidad))
                                  saldo)
                        "Fondos Insuficientes"))))
```

⋆ Notar:

- Combinación de *set!* con variables locales
  ⇒ objetos computacionales con estado local

- El modelo de substitución no es adecuado para aplicación de procedimientos

- Por qué *nuevo-retirar* funciona?

Algunas Observaciones Mas

- Crear objetos "procesadores de retiros":

```
(define (hacer-retirar saldo)
        (lambda (cantidad)
                (if (>= saldo cantidad)
                    (sequence (set! saldo (- saldo cantidad))
                              saldo)
                    "Fondos Insuficientes")))

(define R1 (hacer-retirar 100000))
(define R2 (hacer-retirar 100000))
>(R1 50000)
50000
>(R2 70000)
30000
>(R2 40000)
Fondos Insuficientes
>(R1 40000)
10000
```

R1 y R2 son objetos independientes con estado local !!!

- Crear objetos "Cuenta":

```
(define (hacer-cuenta saldo)
        (define (retirar cantidad)
                (if (>= saldo cantidad)
                    (sequence (set! saldo (-saldo cantidad))
                              saldo)
                    "Fondos Insuficientes"))
        (define (depositar cantidad)
                (set! saldo (+ saldo cantidad))
                saldo)
        (define (atender m)
                (cond ((eq? m 'retirar) retirar)
                      ((eq? m 'depositar) depositar)
                      (else (error "No entiendo"m))))
        atender)

(define cuenta (hacer-cuenta 100000))
>((cuenta 'retirar) 50000)
50000
>((cuenta 'retirar) 60000)
Fondos Insuficientes
>((cuenta 'depositar) 40000)
90000
```

Costos de Introducir la Asignación

- El modelo de substitución no funciona.

- Ningún modelo con propiedades matemáticas "bonitas" es adecuado.

  Ahora no se puede remplazar "iguales por iguales".

- El modelo de substitución no funciona.
  - ⋆ Consideremos una versión, más simple, de *hacer-retirar*:

```
(define (hacer-simple-retirar saldo)
        (lambda (cantidad)
                (set! saldo (- saldo cantidad))
                saldo))

(define R (hacer-simple-retirar 25000))
>(R 20000)
5000
>(R 10000)
-5000
```

  - ⋆ Consideremos el procedimiento:

```
(define (hacer-decrem saldo)
        (lambda (cantidad)
                (- saldo cantidad)))

(define D (hacer-decrem 25000))
>(D 20000)
5000
>(D 10000)
15000
```

  - ⋆ Cómo trabaja *hacer-decrem*?

((hacer-decrem 25000) 20000)

$\Downarrow$

((lambda (cantidad) (- 25000 cantidad)) 20000)

$\Downarrow$

(- 25000 20000)

$\Downarrow$

5000

$\star$ Cómo trabaja *hacer-simple-retirar*?

((hacer-simple-retirar 25000) 20000)

$\Downarrow$

((lambda (cantidad)
(set! 25000 (- 25000 cantidad))
25000) 20000)

$\Downarrow$

(set! 25000 (- 25000 20000)) 25000

$\Uparrow$

no tiene sentido!!!

⋆ Qué pasó?

- En el modelo de substitución los símbolos del lenguaje son esencialmente nombres para valores.

- Con la aparición de *set!* **una variable no puede ser simplemente un nombre**.

- Una variable debe hacer referencia a un lugar donde el valor sea almacenado, y el valor almacenado allí pueda cambiar.

- Ningún modelo con propiedades matemáticas "bonitas" es adecuado.

  ⋆ Consideremos el siguiente principio de *transparencia referencial*

  > "iguales pueden ser substituidos por iguales" sin cambiar el valor de las expresiones

  Este lenguaje, con asignación, lo respeta?

  ⋆ Miremos primero el problema de la igualdad:
  (define D1 (haga-decrem 25000))
  (define D2 (haga-decrem 25000))
  (define R1 (haga-simple-retirar 25000))
  (define R2 (haga-simple-retirar 25000))

  D1 y D2 son iguales?

  R1 y R2 son iguales?

⋆ D1 y D2 son iguales !!!

- Dos nombres para la misma expresión
- Su comportamiento computacional es idéntico

⋆ R1 y R2 no son iguales !!!

- >(R1 20000)
  5000
- >(R1 20000)
  -15000
- >(R2 20000)
  5000

⋆ Qué pasó?

- Se perdió la transparencia referencial
- En qué casos se pueden substituir expresiones por expresiones equivalentes? Difícil de decir
- Analizar programas que usan la asignación es más difícil.
- Aparece la posibilidad de cometer errores que no pueden ocurrir sin la asignación: Ojo a los efectos laterales.
- Difícil capturar la noción de igualdad de manera formal.

El Nuevo Modelo de Evaluación: Modelo de Ambientes

- Recordemos el modelo de substitución:
  Evaluar la expresión

$$(exp\ exp_1\ exp_2\ \ldots\ exp_n)$$

  consiste en:

  ⋆ **Evaluar** las subexpresiones $exp, exp_1, \ldots, exp_n$:

$$\Rightarrow o, a_1, \ldots, a_n$$

  ⋆ **Aplicar** el operador o a los argumentos $a_1, \ldots, a_n$.

  ◦ Si o es primitivo: $\Rightarrow$ El interpretador sabe qué hacer

  ◦ Si o es definido con *define*:
  $\Rightarrow$ **Aplicar** es **evaluar** el cuerpo del procedimiento con cada parámetro formal reemplazado por el correspondiente argumento.

- Revisar el concepto de **aplicación**.

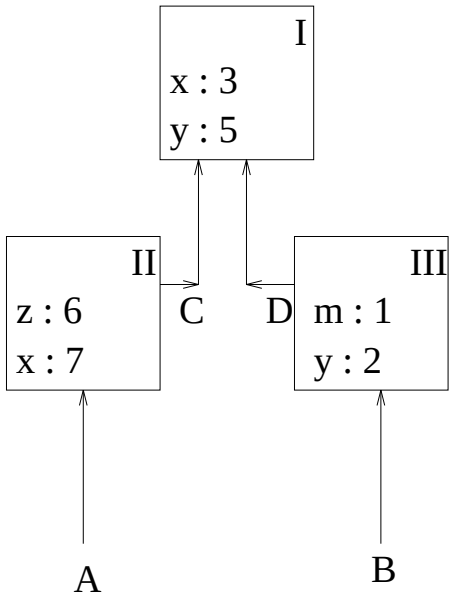
- Ahora una variable debe designar un lugar en donde se puedan almacenar valores:

  AMBIENTES

  ⋆ Ambiente ≡ Sucesión de Marcos

  ⋆ Marco ≡ (Tabla de Corresp., apuntador)
  Tabla de correspondencias: { Var ↔ Valor }
  Apuntador: ambiente padre

- Ejemplo:

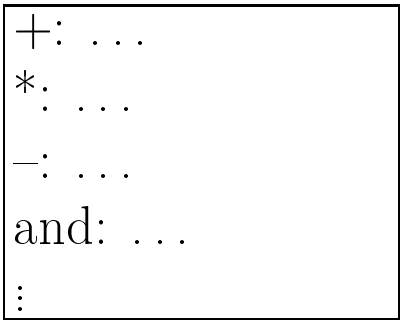


- Ambiente es crucial en el proceso de evaluación: Determina el contexto de evaluación de una expresión.

- Una expresión adquiere un significado sólo con respecto a un ambiente en el cual sea evaluada.

  Aún en casos como (+ 1 1),

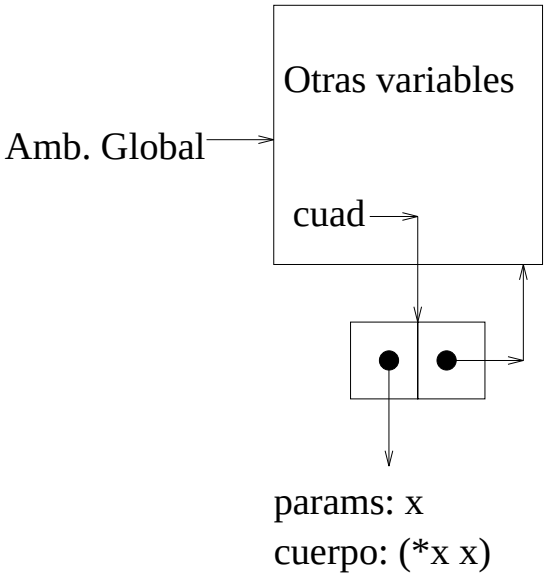
  $\Rightarrow$ Ambiente Global



Las Reglas de Evaluación

- Qué cambia?
  - ⋆ Evaluación de formas no especiales: igual.
  - ⋆ Evaluación de formas especiales: puede cambiar.
  - ⋆ Aplicación de procedimientos: **lo nuevo**.
- Qué es un procedimiento?
  - ⋆ Evaluación de (define (<proc> pars) cuerpo)
  - ⋆ Procedimiento ≡ (código, apuntador)
    Código:(parámetros, cuerpo)
    Apuntador:ambiente donde el proc. está definido

- Ejemplo: (define (cuad x) (* x x))

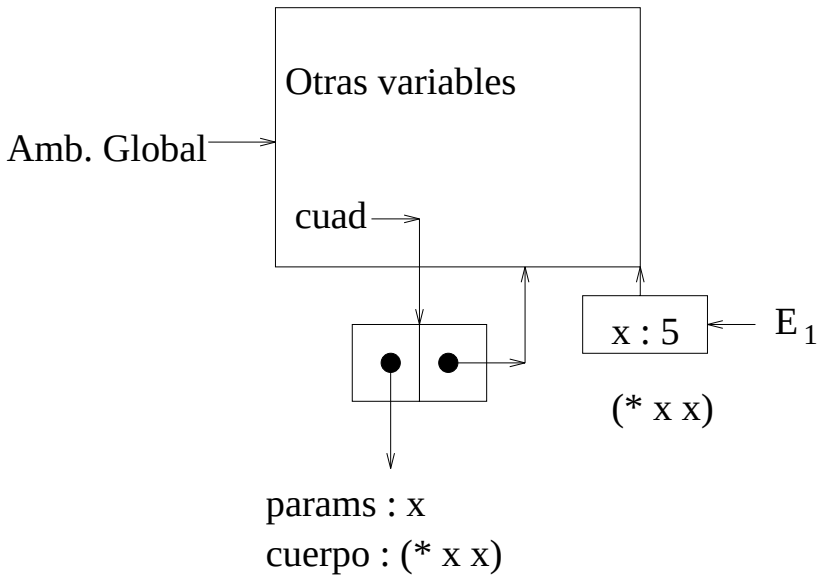


- Y cuando se utiliza *define* para definir una variable, qué ocurre?

(define <var> <exp>)

- Cómo se aplica un procedimiento?
  - ⋆ Crear un nuevo ambiente con un marco:
    - ◦ Corresp. params. formales y actuales
    - ◦ Apuntador al ambiente especificado por el proc.
  - ⋆ Evaluar el cuerpo en el nuevo ambiente

- Ejemplo: (cuad 5)



- Cómo se evalúa la forma especial *set!*?

  Evaluar (set! <var> <exp>) en el ambiente A:

  ⋆ Localizar <var> en A, y cambiar su valor en el marco en que se encontró.

  ⋆ Si <var> no existe en el ambiente: error, "unbound variable".

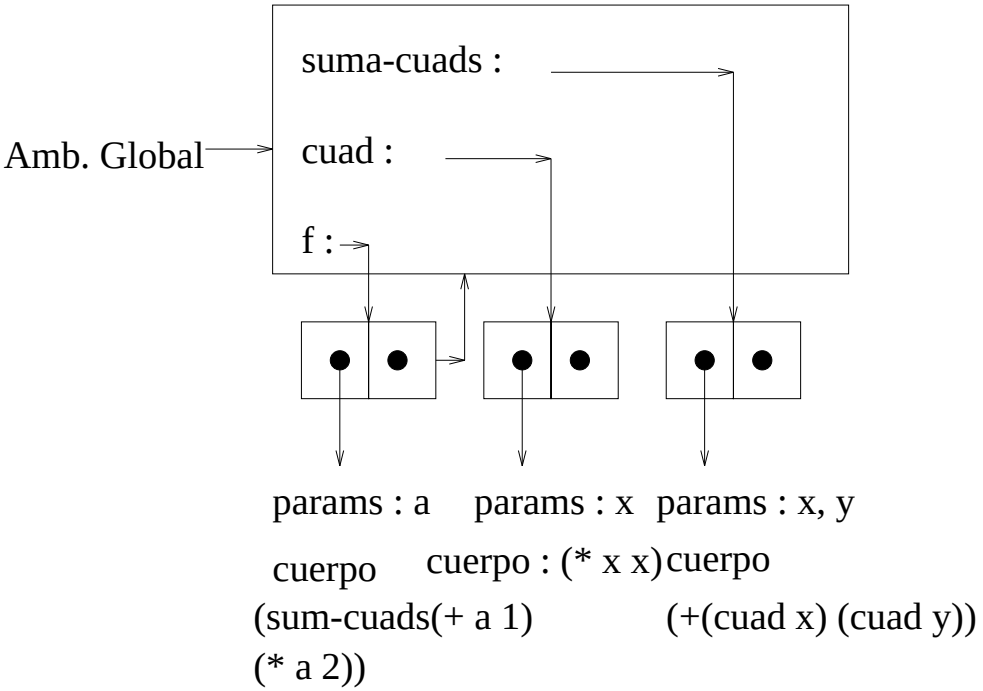
Evaluando Procedimientos Simples

- Evaluación de defs. de *cuad*, *sum-cuads*, *f*:

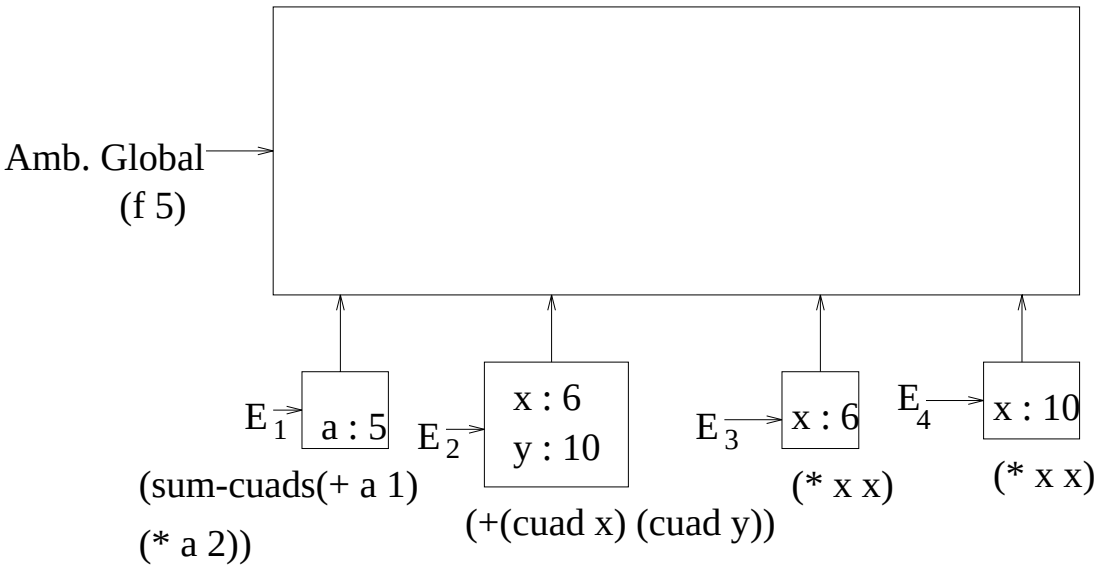
```
(define (cuad x) (* x x))

(define (sum-cuads x y)
        (+ (cuad x) (cuad y)))

(define (f a) (sum-cuads (+ a 1) (* a 2)))
```



- Evaluación de (f 5)

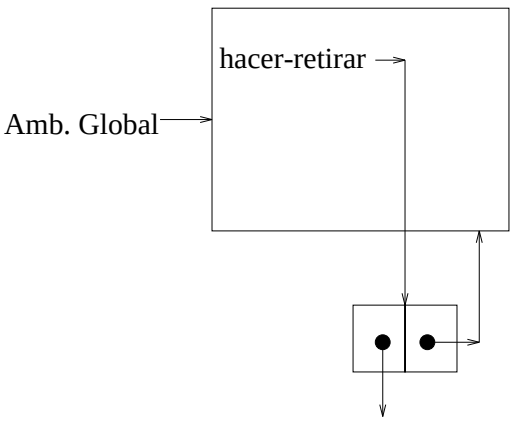


Evaluando Procedimientos con Asignación

- Recordemos la función *hacer-retirar*:

```
(define (hacer-retirar saldo)
        (lambda (cantidad)
                (if (>= saldo cantidad)
                    (sequence (set! saldo (- saldo cantidad))
                              saldo)
                    "Fondos Insuficientes")))
```

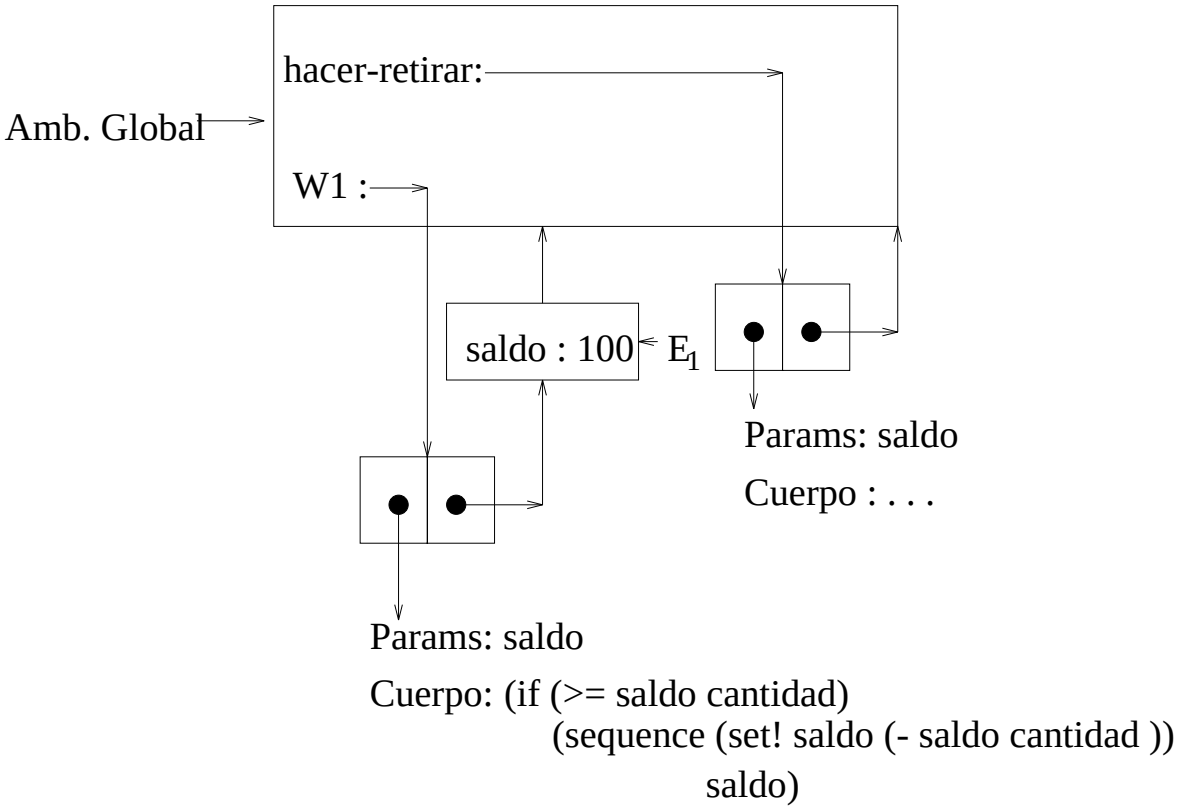
Una vez evaluado el *define*:



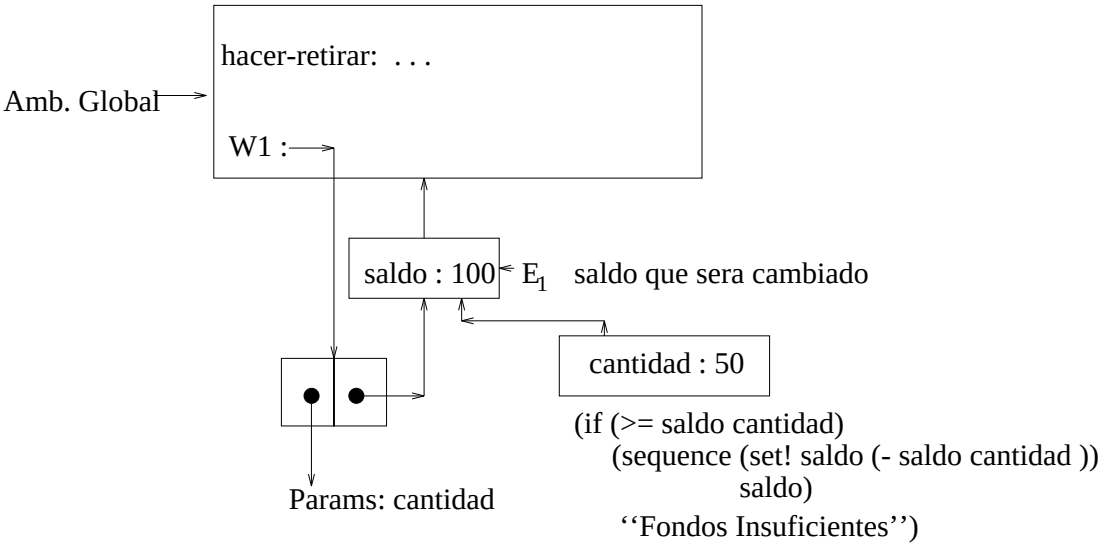
Params: saldo

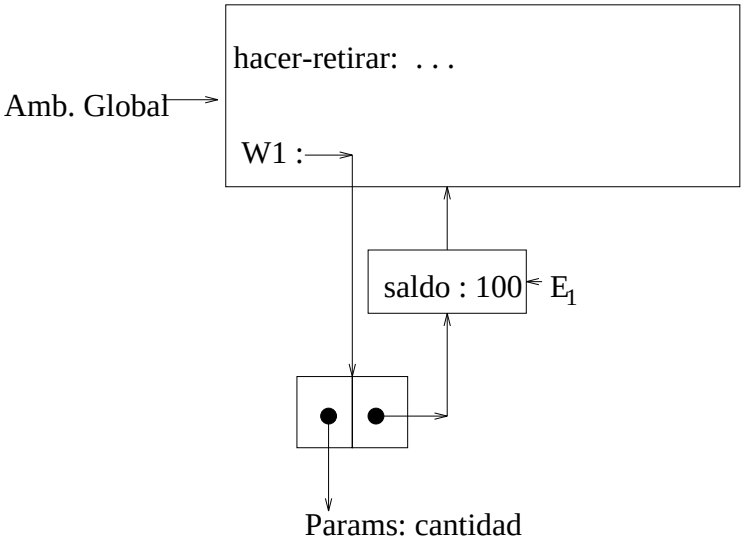
Cuerpo: (lambda (cantidad)
(if (>= saldo cantidad)
(sequence (set! saldo (- saldo o
saldo)
‘‘Fondos Insuficientes’’))

- Evaluación de (define W1 (hacer-retirar 100))

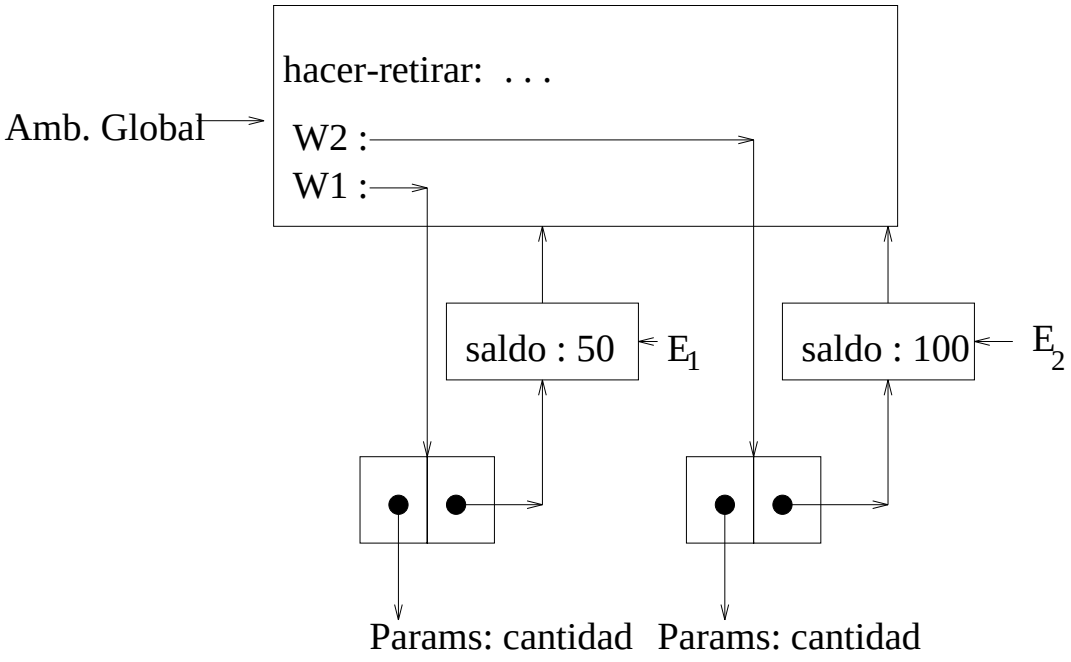


- Evaluación de (W1 50)





- Evaluación de (define W2 (hacer-retirar 100))



Ejercicio

Recuerde la primera solución que se dió para tener *saldo* local y no global:

```
(define nuevo-retirar
        (let ((saldo 100000))
            (lambda (cantidad)
                    (if (>= saldo cantidad)
                        (sequence (set! saldo
                                        (- saldo cantidad))
                                  saldo)
                        "Fondos Insuficientes"))))
```

Muestre porqué esta solución funciona, evaluándola con el modelo de ambientes.

Recuerde que

(let ((<var> <exp>)) cuerpo)

es sólo una forma de abreviar:

((lambda (<var>) cuerpo) <exp >)