

Interacciones Algorítmicas con Scheme

E. Comer B.

Las siguientes interacciones representan diálogos entre Evi, alias para un evaluador de Scheme (compatible con el estándar R5RS) y Arturo, Miriam y Jenny, estudiantes de ingeniería, informática y matemáticas¹, respectivamente. Estas reflexiones y conversaciones generarán una dinámica que esperamos muestre un poco de la riqueza de la programación, del lenguaje Scheme y de la interacción humana. La versión actualizada de este documento experimental, con licencia tipo GNU, puede encontrarla en www.matikai.com/scheme/ias.pdf

Nota: Esta versión representa sólo uno de múltiples escenarios posibles y probablemente no el más completo, esperamos que las siguientes versiones se aproximen sucesivamente al escenario más efectivo para aprender y disfrutar de la interacción tanto humana como algorítmica.

versión experimental α_8
(14 mayo, 2004)

Instituto Tecnológico de Tijuana

¹ A la fecha, nuestra Institución no ofrece esta Licenciatura. Jenny nos motivará a corregir esta grave omisión.

**Dedicado
a todos los estudiantes
del Instituto Tecnológico de Tijuana**

Día 1

Primeras interacciones de Arturo con Evi

```
>
```

[1]

[Arturo] Esta “ventana” es de "interacciones", me dijeron que aquí puedo evaluar expresiones simbólicas (llamadas también s-expresiones) y que ">" representa al evaluador (creo que yo le llamaré **Evi**, no me gustan las palabras largas, cuando debo pronunciarlas a cada rato). También me dijeron que podría funcionar como una calculadora muy potente y (junto con el editor) podría usarse para desarrollar programas tan complicados como quisiera. Veamos qué puede hacer.

```
> 1+2
error: reference to undefined identifier2: 1+2
>
```

[2]

¡Ah!, Evi piensa que 1+2 es un nombre de variable. Bueno, eso significa que puedo "darla de alta" con define (según recuerdo), probemos:

```
> (define 1+2 3)
> 1+2
3
>
```

[3]

Pero, claro está que no puedo definir así todas las sumas posibles (un número infinito no contable, como diría Jenny). Sí, ahora recuerdo: cada vez que necesite hacer una operación, debo ponerla entre paréntesis y usar primero el operador o nombre de la función, y así también para cada sub-expresión. Es decir escribirlas en ‘prefijo’. Veamos qué tal trabaja.

```
> (+ 1 2)
3
> (+ 1 2 3 4 5 6 7 8 9)
45
> (sqrt 49)
7
> (expt 2 10)
1024
> (define ^ expt)
> (^ 3 4)
81
> (^ 2 50)
1125899906842624
> (^ 2 100)
1267650600228229401496703205376
> (^ 12 70)
3488889569322095618800250852305904679826716343134391016285.1
174746322068045824
```

[4]

² Referencia a *identificador* indefinido. El identificador se utiliza para nombrar variables o funciones.

¡Hey, no está mal para un principiante!. Eso de que los resultados enteros son exactos le va a gustar a Jenny: ¡puedo usar tantos como necesite, sólo la memoria de la computadora podría limitarlos!.

Me gusta más usar el *caret* ^ que escribir **expt**, eso de definir las funciones usando alias es muy práctico. Lástima que en el estándar, se reconozca sólo a `expt` como la función para exponenciación. Luego le pregunto a Miriam cómo le hago para no tener que definir ^ en cada sesión de trabajo.

Ahora necesito experimentar con el editor. ¡Ya quiero hacer mi primer programa! Tengo curiosidad por ver cómo funciona. Me dijo el profe que en Scheme (pron. es-kím) es muy sencillo. Tomé la siguiente nota:

Un **programa** en Scheme es una colección de ‘formas’, cada una de las cuales puede ser una definición o una expresión. Estas formas se evalúan secuencialmente para generar el resultado o efecto deseado.

[N1]

Bueno, voy a capturar mi primer programa en el editor³:

```
; grados1.scm
;
; convierte grados centígrados a fahrenheit
; usa la fórmula: F = 1.8C + 32
;
(define (c->f c)
  (+ (* 1.8 c) 32))

"Hola Arturo:"
"Convierto Centígrados a Farenheit"
(display "Centígrados? ")
(define c (read))
(display c) (display "C = ")
(display (c->f c)) (display "F")

; fin de grados.scm
```

[5]. Archivo `grados1.scm`: Convierte Centígrados a Farenheit

Después de evaluar (o ‘ejecutar’) el programa anterior, obtengo de inmediato la siguiente respuesta (en la ventana de interacciones)

```
"Hola Arturo:"
"Convierto Centígrados a Farenheit"
Centígrados?
```

[6]

Al darle el número correspondiente a los grados Centígrados y oprimir [enter], se completa la siguiente línea mostrando la equivalencia a grados Farenheit:

```
"Hola Arturo:"
"Convierto Centígrados a Farenheit"
Centígrados? 23
23C = 73.4F
>
```

[7]

³ Las implementaciones del lenguaje Scheme tienen dos áreas de trabajo: la ventana del editor y la ventana del evaluador (Evi, para Arturo)

Esto se pone interesante. Mañana le preguntaré a Miriam cómo evitar escribir tantas formas display , debe haber alguna forma de simplificar este programa [5] y claro, aprovecharé para preguntarle lo del caret ^, de seguro ella me podrá ayudar.

Día 2

Arturo descubre con Jenny y Miriam, el uso de la formas `if` y `cond`

[Jenny] Hola Arturo, que milagro que te veo tan temprano en el Lab.

[Arturo] ¡Jenny!, gusto en verte. Creo que podemos programar en Scheme las funciones que me enseñaste ayer. ¿Tienes tu cuaderno de cálculo?

[J] Sí, aquí están las fórmulas, me ayudarías mucho si las pudieras programar:

$$f_1(x) = \begin{cases} \cos(x+2) - \text{sen}(x/2) & \text{si } x < 0 \\ \text{sen}(2x) + \cos(2) & \text{si } x \geq 0 \end{cases}$$

$$f_2(x) = \begin{cases} e^{-x} & \text{si } x < -1 \\ e & \text{si } -1 \leq x \leq 1 \\ e^{\sqrt{x}} & \text{si } x > 1 \end{cases}$$

[8]

[A] Creo que me lo pones difícil, porque apenas estoy empezando con eso de escribir fórmulas en prefijo. Pero mira, ahí viene Miriam, ella nos va a salvar.

[Miriam] Hola muchachos, gusto de verlos tan temprano trabajando.

[A] ¡Miriam!, qué bueno que veniste, tenemos que programar estas funciones en Scheme, ¿tendrás tiempo de ayudarnos?

[M] ¡Claro! para que son los amigos. Déjenme ver... creo que será suficiente con un `if` para `f1` y un `cond` para `f2`. Escribamos las funciones:

```
; jenny1.scm : funciones para Jenny (definidas en [8])

(define (f1 x)
  (if (< x 0)
      (- (cos (+ x 2)) (sin (/ x 2)))
      (+ (sin (* 2 x)) (cos 2))))

(define (f2 x)
  (cond ((< x -1) (exp (- x)))
        ((> x 1) (exp (sqrt x)))
        (else (exp 1))))

; fin de jenny1.scm
```

[9]. Archivo `jenny1.scm`: Codificación de las funciones definidas en [8]

[M] Listo, parece que no hay errores, resultó más sencillo de lo que pensaba.

[J] Wow Miriam, qué rápido lo hiciste. Ahora sólo nos resta hacer algunas pruebas. Necesito evaluar f_1 y f_2 en -100 y 100 . Además también en la lista de números $-2, -1, 0, 1$ y 2 .

[A] ¡Hey!, creo que yo puedo hacerlo, me servirá de práctica: Primero hay que revisar la sintaxis y después evaluar (ejecutar) las definiciones. A diferencia del programa en [5] no aparece ningún resultado, porque sólo pusimos definiciones. Así que hay que hacer las 'llamadas' con los datos que dijo Jenny:

```
> (f1 -100)
-1.0816630989953882
> (f1 100)
-1.289444133761137
> (f2 -100)
2.6881171418161614e+043
> (f2 100)
22026.465794806725
>
```

[10]

[A] Y ahora, para las listas de números, usaremos el famoso `map` :)

```
> (map f1 '(-2 -1 0 1 2))
(1.8414709848078965
 1.0197278444723428
 -0.4161468365471424
 0.4931505902785393
 -1.1729493318550706)
> (map f2 '(-2 -1 0 1 2))
(7.38905609893065
 2.718281828459045
 2.718281828459045
 2.718281828459045
 4.1132503787829275)
>
```

[11]

[J] Todo parece correcto. Luego me dicen cómo graficar estas funciones y (aunque sé que ha de ser más difícil) necesito también graficar sus primeras derivadas. Gracias por todo muchachos, nos vemos, ya tengo que irme a mi clase de Teoría de Números, luego les cuento.

[M] Arturo, qué te parece si buscas la fórmula para calcular la primera derivada de una función (p. ej. en cualquier libro de Análisis Numérico). Si gustas mañana mismo la podríamos programar y de paso impresionas a tu maestro con una aplicación muy interesante. ¡Ah! por cierto, felicidades por el manejo que hiciste del `map`, es una función muy útil. Luego te comparto algunos programas, te van a sorprender :)

[A] Muy bien Miriam, gracias por todo. Fué muy interesante lo que hiciste, pero me quedaron algunas dudas, además tengo otras cosas que te quería preguntar, pero creo que podremos verlo más tarde. ¡Nos vemos!

Día 3

Miriam comparte con Arturo la forma de manejar el load

[Arturo] Necesito entender cómo definir mis variables y funciones más comunes para reutilizarlas en cualquier programa que necesite. El profe nos dijo que había bibliotecas (colecciones o *libraries*) con código de apoyo para diferentes aplicaciones (p. ej. para gráficas o temas matemáticos especializados) pero que las veríamos más adelante mediante algunos ejemplos.

[Miriam] ¡Arturo! qué me cuentas, ¿por qué tan pensativo?.

[A] Hola Miriam. Bueno, creo que la programación me ha dado varias cosas nuevas en que pensar: Por ejemplo, ¿cómo puedo crear una biblioteca básica para mis programas de ingeniería y matemáticas (digamos, para incluir el símbolo \wedge llamado caret)?

[M] No es tan difícil, sencillamente organizas tu código de definiciones en varios archivos de acuerdo a algún criterio (p. ej. entrada/salida, conjuntos, cálculo, etc.) y los "llamas" al inicio de tus aplicaciones según se requiera. Por ejemplo, podemos crear el archivo llamado `basicas.scm`, que contenga una "miscelánea" de las funciones deseables básicas (incluyendo tu caret :). Voy a aprovechar para compartirtte algunas de mis funciones esenciales para aplicaciones de texto.

```

; basicas.scm
; funciones básicas de apoyo
; colección preliminar (versión 1)
;
; contenido:
;           ^          display-read      display-nl
;   just-der      show

; define la función potencia como ^
;
(define ^ expt)

; justifica un string a la derecha según ancho
;
(define (just-der str ancho)
  (let ((len (string-length str)))
    (if (< len ancho)
        (string-append
         (make-string (- ancho len) #\space) str)
        str)))

; captura un dato después de desplegar el string msg
; según ancho (el campo de los mensajes)
;
(define (display-read msg ancho)
  (display (just-der msg ancho))
  (read))

```

```

; despliega un string just. der (según campo)
; seguido de un obj y un newline
;
(define (display-nl msg obj campo)
  (display (just-der msg campo))
  (display obj) (newline))

; despliega cada uno de los objetos y reconoce 'nl
; como (newline)
;
(define (show obj . otros)
  (for-each (lambda(w)
             (cond ((equal? w 'nl) (newline))
                   (else (display w))))
    (cons obj otros)))

; fin de básicas.scm

```

[12]. Archivo `basicas.scm`: Funciones básicas para aplicaciones de texto

[M] Bien Arturo. Ahora, supongamos que quieres hacer un cálculo sencillo como capturar dos números y entregar su promedio. Entonces puedes hacer el siguiente programa y "llamar" las definiciones en `basicas.scm` con tan sólo escribir: `(load "basicas.scm")`. Claro, este archivo debe estar en el mismo subdirectorio de tu programa. Veamos cómo se haría:

```

; arturo1.scm
; ejemplo del uso de load y funciones básicas E/S texto

; carga las definiciones en "basicas.scm"
;
(load "basicas.scm")

(show "Promedio de dos números..." 'nl)
(define a1 (display-read "n: " 12))
(define a2 (display-read "m: " 12))
(display-nl "(n + m)/2 = " (/ (+ a1 a2) 2.0) 12)

; fin de arturo1.scm

```

[13]. Archivo `arturo1.scm`: Ejemplo del uso de `load`

[M] Es tu turno, puedes *ejecutar* el programa :)

[A] Bien, tengo muchas preguntas, pero veamos qué pasa:

```

Promedio de dos números...
  n: 4
  m: 10
(n + m)/2 = 7.0
>

```

[14]

[A] Hey! no está mal. Creo que este `load` me va a ser muy útil. Pero ve la hora, son las 9:09 AM hay que correr a la clase. Muchas gracias, nos vemos en *la Cafe*.

[M] Muy bien, pero ¡mira!: olvidabas tu diskette :)

Día 4

Arturo se sorprende al calcular la derivada numérica con ayuda de Jenny

[Jenny] Mira Arturo lo que encontré, es una fórmula para calcular la derivada numérica de una función cualquiera⁴. Ahora que estás programando, tal ves te resulte un buen ejercicio:

$$f'(x) = \frac{f(x-2h) - 8f(x-h) + 8f(x+h) - f(x+2h)}{12h}$$

[Arturo] Creo que podríamos intentarlo, pero tengo duda sobre el significado de h .

[J] Bueno, h es una medida del error que podemos esperar en el resultado, con esta fórmula, decimos que el error es del orden $O(h^4)$. Si gustas lo podemos considerar como un parámetro de la función a programar.

[A] Bien, vamos a intentarlo, hay que ser optimista. De paso, voy a probar el archivo que me dejó Miriam ayer :) Pero antes, como vamos a usar funciones, déjame sacar una nota que nos dió el profe sobre funciones anónimas:

Una **función anónima** (i.e. cuando el nombre de la función no es relevante) puede construirse utilizando la forma especial lambda, con la siguiente sintaxis:

```
(lambda (<parámetro>*)
  <cuerpo de la función>)
```

donde el asterisco indica que podemos tener cero o más parámetros.

[N2]

[J] ¿Eso significa que justo donde necesite una función, puedo escribir la forma lambda y hacer el cálculo?

[A] Así parece. Hagamos unas pruebas para estar seguros:

```
> (lambda(x)(+ x 10))
#<procedure:3:2>
> ((lambda(x)(+ x 10)) 40)
50
> ((lambda(n)(expt 2 n)) 3)
8
>
```

[15]

[J] Creo que ya entendí. Por ejemplo, usando tu función `map` y una función anónima, podríamos transformar una lista de números de distintas formas, por ejemplo, convirtiéndolos a -1, 0 y 1, dependiendo si son negativos, cero o positivos. ¿Crees que podrías hacerlo?

[A] Tal ves si me ayudas en el camino :)

⁴ Técnicamente se llama: fórmula de diferencias finitas centradas de orden $O(h^4)$

```
> (map (lambda(x)
      (cond ((< x 0) -1)
            ((= x 0) 0)
            (else 1))) '(40 -3 5 -1 0 2))
(1 -1 1 -1 0 1)
>
```

[16]

[J] ¡Funcionó!, creo que vamos a hacer muchos experimentos con esta formita lambda :)

[A] Bien, ahora sí, intentemos programar la derivada con tu fórmula:

```
; deriv1.scm
; programa básico para calcular la derivada de f en x

(load "basicas.scm")

; calcula la derivada numérica con un error O(h^4)
;
(define (derivada f x h)
  (/ (+ (f (- x h h))
        (* -8 (f (- x h)))
        (* 8 (f (+ x h)))
        (- (f (+ x h h))))
     (* 12 h)))

(show "Derivada numérica de una función..." 'nl)
(define cuerpo-fun (display-read "(f x)=" 12))
(define fun (eval (list 'lambda '(x) cuerpo-fun)))
(define x (display-read "x: " 12))
(define h (display-read "h: " 12))
(display-nl "f'(x)=" (derivada fun x h) 12)

; fin de deriv1.scm
```

[17]. Archivo deriv1.scm: Una aplicación matemática

[A] Wow, sí que me pusiste a pensar. Después de varios intentos para las variables cuerpo-fun y fun, finalmente parece que trabaja bien. El detalle es que la función hay que escribirla en forma de prefijo, para recordar eso escribí "(f x)" en lugar de "f(x)". Déjame hacer dos pruebas sencillas:

```
Derivada numérica de una función...
(f x)=(+ x 2)
x: 1
h: 0.001
f'(x)=0.99999999999999638
>
```

[18]

```
Derivada numérica de una función...
(f x)=(^ x 2)
x: 2
h: 0.0001
f'(x)=4.000000000000548
>
```

[19]

[J] Bien, los resultados exactos son 1 y 4, los errores son menores que 10^{-11} , así que es aceptable. Mira, aquí tengo una función más compleja para hacer otra prueba. ¿crees que podemos hacer los cálculos, digamos para $x=1$ y $h=0.001$?

$$f(x) = \sin(x) + 2e^{3x}$$

[A] No se ve tan difícil, veamos qué nos responde:

```
Derivada numérica de una función...
(f x)=(+ (sin x) (* 2 (exp (* 3 x))))
x: 1
h: 0.001
f'(x)=121.05352384466528
>
```

[20]

[J] Déjame resolverla simbólicamente para comparar resultados.

$$f'(x) = \cos(x) + 2(3)e^{3x} = \cos(x) + 6e^{3x}$$

¿Cuánto daría esta nueva función en $x=1$?

[A] Muy fácil:

```
> (+ (cos 1) (* 6 (exp 3)))
121.05352384499416
>
```

[21]

[J] ¡Excelente! Los resultados coinciden hasta 9 decimales. No está nada mal para hacer experimentos. Luego sería interesante poder graficar tanto la función como su derivada. Creo que voy a tener que aprender contigo algo más de Scheme :)

[A] Bueno, yo apenas estoy aprendiendo, pero junto con Miriam, creo que podemos hacer algunos programitas interesantes.

[J] De acuerdo, pero por ahora creo que ya cumplimos :) Nos vemos luego porque tengo que ir a un taller de demostraciones matemáticas. Luego te comparto el material, me han dicho que se parece mucho al proceso de hacer programas.

[A] *Sale*. Creo que me has ayudado a tener más confianza en esto de programar. Nos vemos en la tarde :)

Día 5

Miriam comparte con Arturo la idea de *programación orientada a objetos*

[Miriam] Hola Arturo, estás muy estudioso :) ¿cómo van tus clases?

[Arturo] Bien Miriam, sólo que en programación no entiendo aún las aplicaciones que puedo desarrollar con la forma `case` y con algo que llamó el maestro "funciones con estado interno". ¿Tendrías entre tus curiosidades algún ejemplo?

[M] ¿Has oído hablar de la *Programación Orientada a Objetos*?

[A] Creo que algo comentó el profe, sobre mensajes que podemos dar a un robot y cómo éste estaría programado para responder y actuar según tales mensajes.

[M] Bueno, yo tengo un ejemplo de cuentas bancarias. Te lo puedo mostrar y si gustas más adelante vemos cómo simular los movimientos de un robot elemental. Creo que por aquí traigo la descripción:

Una **objeto básico de programación** consiste de:

- a). Un conjunto de **atributos** y
- b). Un conjunto de **métodos** .

Los atributos determinan el **estado** del objeto. Sus métodos son activados a través de **mensajes** y parámetros que determinan el **comportamiento** del objeto. En Scheme, un objeto básico puede implementarse mediante **funciones con estado interno**. Ejemplo sencillo:

```
Objeto:      Cuenta bancaria elemental
Constructor: make-cuenta(nombre, cantidad)
Atributos:  Nombre, Balance
Métodos:    Balance (id)
             Retira (cantidad)
             Deposita (cantidad)
             Transfiere (cuenta, cantidad)
             Id
```

[N3]

[M] Se me ocurre que para tu robot básico, podríamos tener la siguiente descripción (bueno, tu podrías agregarle muchas otras cosas :)

```
Objeto:      Robito
Constructor: make-roboto(nombre, posición, dirección)
Atributos:  Nombre, posición, dirección
Métodos:    Posición(id)
             Dirección (id)
             Avanza (distancia)
             gira-der (grados)
             gira-izq (grados)
             Descansa (segundos)
             Id
```

[I4]

[A] Oye, ¿no sabía que supieras de robots?

[M] Bueno, se muy poco :) La idea de Robito es básicamente la misma que usan lenguajes como LOGO y también Scheme al implementar lo que llaman "geometría de tortuga" o "gráficas de tortuga". Luego te comparto algo sobre eso. Pero por el momento, aquí tienes el código de la función `make-cuenta`:

```

; cuenta.scm
; definición de una función con estado interno
; ejemplo básico de orientación a objetos
;
; crea una cuenta bancaria básica con
; atributos: id, balance
; y métodos:
; 'balance -> número
; 'retira número -> número | lista
; 'deposita número -> null
; 'transfiere cuenta número -> null | lista
; 'id -> id
; con la interpretación típica
;
(define (make-cuenta nombre cantidad)
  (let ((id nombre)
        (balance (max cantidad 0)))
    (lambda(mensaje . data)
      (define (retira cant)
        (if (> cant balance)
            (begin
              (display `(.id solo tiene ,balance))
              #f)
            (begin
              (display `(.id retira ,cant))
              (set! balance (- balance cant)))))
        (case mensaje
          ((balance) (display `(.id = ,balance)))
          ((retira) (retira (car data)))
          ((deposita) (display `(.id recibe ,(car data)))
                      (set! balance
                               (+ balance (car data))))
          ((transfiere) (let ((cta2 (car data))
                              (cant (cadr data)))
                          (if (retira cant)
                              (cta2 'deposita cant))))
          ((id) id)
          (else `(no reconozco el método ,mensaje))))))
    ))
; fin de cuenta.scm

```

[22]

[A] ¿Podríamos abrir cuentas con nuestros nombres?

[M] Claro, hagamos algunas transacciones de prueba:

```

> (define cuental (make-cuenta 'arturo 10000))
> (cuental 'balance)
(arturo = 10000)
>

```

[23.A]

```

> (define cuenta2 (make-cuenta 'miriam 7000))
> (cuenta2 'balance)
(miriam = 7000)
> (cuenta1 'retira 4000)
(arturo retira 4000)
> (cuenta1 'transfiere cuenta2 1500)
(arturo retira 1500)(miriam recibe 1500)
> (cuenta1 'balance)
(arturo = 4500)
> (cuenta2 'balance)
(miriam = 8500)
> (cuenta1 'deposita 4000)
(arturo recibe 4000)
> (cuenta1 'balance)
(arturo = 8500)
>

```

[23.B]

[A] Gracias por depositarme los 4000, así quedamos parejos :). Bueno, y en el caso de Robito, ¿cómo serían las respuestas?

[M] Si lo estamos simulando en pantalla serían gráficas: alguna figura geométrica que lo represente se movería según las instrucciones que le dieras. Pero si fuera real, entonces se podría mover en el piso, o mover un brazo mecánico según la forma que tú le indicaras.

[A] Suena muy interesante, pero ¿tendría que estar dándole las instrucciones a cada paso?

[M] Bueno, no necesariamente, podrías usar ciclos con instrucciones como `do`, `for` o `while` para repetir los comandos que tú quisieras de acuerdo a ciertas condiciones.

[A] Órale, ya quisiera estarlo haciendo, cuando menos en pantalla. Lástima que el tiempo vuela y ya nos tocan las clases. Por cierto, estamos viendo `macros` y creo que podríamos agregar algunas cosas muy útiles a tu archivo `basicas.scm`.

[M] Sí claro, creo que ya es hora de hacer la "versión 2", por allí tengo otras cosas que podrían facilitarte el trabajo con *arreglos*. Le llaman "azúcar sintáctica" :)

[A] ¡Mm! eso suena bien, aunque un poco misterioso, luego me platicas. Gracias por tu ayuda. Voy a pensar algo más en los atributos de Robito :) ¿Nos vemos a la salida para tomar algo fresco?

[M] Bueno, si tu pagas :) pero vámonos que llegaremos tarde.

D"m: Continuará...