



Scheme

- Scheme es un lenguaje funcional. Es un dialecto de LISP.
- Fue desarrollado por Steele y Sussman en 1975.
- Existen una gran cantidad de dialectos de Scheme pero usaremos el llamado Racket.
- Scheme es un intérprete y su línea de comandos está indicada por un PROMPT.
- También se puede compilar.
- Sí es sensible a las mayúsculas.

Notación prefija

- En Scheme, todas las expresiones utilizan notación prefija, es decir, el operador va primero y los operandos después.
- Además, todo se pone entre paréntesis, como una lista.
- Si no se le indica otra cosa, debido a la notación prefija, el intérprete trata el primer elemento de la lista como una función (operador) y el resto como sus parámetros (operandos).
- Ejemplo:

<code>(* 2 3)</code>	→	<code>2 * 3</code>
<code>(> 5 6)</code>	→	<code>5 > 6</code>
<code>(+ 2 3 4)</code>	→	<code>2 + 3 + 4</code>
<code>(* 4 (+ 3 2))</code>	→	<code>4 * (3 + 2)</code>

Para aprender un lenguaje imperativo

- Tipos de variables
- Asignación
- Expresiones (operadores aritméticos y lógicos)
- Lectura y escritura de datos (teclado y pantalla)
- Condicionales (if y case)
- Ciclos (for, do... until y do... while) y recursión
- Archivos
- Algunas otras instrucciones especiales

Para aprender un lenguaje funcional

- Objetos (átomos y listas)
- Expresiones
 - Aritméticas
 - Lógicas
 - Relacionales
- Funciones
 - Uso de las predefinidas
 - Definición de nuevas
- Recursión

Objetos

- Hay tres tipos de objetos en Scheme:
 - Átomos: palabras, strings, booleanos y números
 - Nombres: identificadores que inician con una letra
 - Listas: conjunto ordenado de listas o átomos
- El único tipo de dato que existe en Scheme es el tipo lista de objetos por lo que no requiere declaración.
- Scheme es especial para manipulación de símbolos.

Intérprete

- Scheme es un intérprete (no un compilador, aunque se puede compilar) por lo que todos los objetos que se encuentra tiende a EVALUARLOS.
- Esto es debido a que para Scheme todo es una expresión:
(<operador> <operando>*)
- Evaluar un objeto significa:
 - Si es una lista, piensa siempre que es una función y la trata de ejecutar. Si no está definida marca error.
 - Si es un átomo, piensa siempre que es una variable y trata de obtener su valor. Si está sin valor marca error.
- Si no se desea que evalúe algo se le debe indicar explícitamente.

Átomos

Los átomos son de dos tipos:

- Numéricos: formados por una secuencia de dígitos (y algunas veces un punto decimal)
 - Enteros: no tienen punto (e.g. 567, -32)
 - Racionales: fracciones (e.g. 4/7, 11/13)
 - Punto flotante: tienen punto o notación científica (e.g. 45.32, 4E-2)
- Simbólicos: cualquiera que no sea numérico (e.g. Aoo, VICTOR, PC, +)

Átomos especiales

- Algunos átomos sí se evalúan aunque no contengan ningún valor (no pueden ser variables):
 - `#t`: es el átomo que representa al valor booleano TRUE. También se puede interpretar como una lista NO VACÍA.
 - `#f` o `null`: es el átomo que representa al valor booleano FALSE. También se puede interpretar como lista VACÍA.
 - Los números siempre se evalúan como números.
 - Los strings siempre se evalúan como strings.

Identificadores

- Los identificadores en Racket son muy liberales.
- A diferencia de otros lenguajes de programación que restringen mucho los caracteres válidos para sus identificadores, en Racket, prácticamente no hay restricciones.
- Los únicos caracteres no válidos en los identificadores son: `()[]{} , ' ; # | \`
- Tampoco se pueden utilizar identificadores que se correspondan con literales numéricos y tampoco están permitidos los espacios dentro de los identificadores.
- Ejemplos:
 - `variable-con-guiones`, `variable+con+más-y-tildes-123abc`, `+-`, `¿variable-interrogativa?`, `23..4`, `variable/dividida`, etc.

Lista

- Una lista es un conjunto ordenado de átomos o listas, colocados entre paréntesis y separados por uno o más blancos. e.g.

(+ 3.1416 34), 3 elementos

(HOLA MUNDO), 2 elementos

(A B C D E), 5 elementos

(A B (1 2 3) C), 4 elementos, uno de los cuales es una lista de 3 elementos

Funciones

- Todas las operaciones e instrucciones que tiene Scheme son FUNCIONES.
- Una función es una lista, en donde el primer elemento indica el NOMBRE de la función y los siguientes son sus PARÁMETROS (argumentos, cuando se define), y siempre REGRESA un valor. e.g.

(ORDENA 2 4 5 1)

(+ 3.4 2.1)

(EVALUA (SUMA 3 5) (RESTA 8 5))

- Los parámetros también pueden ser funciones (el resultado que devuelven).

¿Cómo diferenciarlos?

- La pregunta obligada es ¿cómo podemos diferenciar entre una variable y un átomo o entre una lista y una función?
- La respuesta es muy simple, Scheme permite el uso del apóstrofe para indicar que un identificador o una lista no se debe EVALUAR.
- EVALUAR significa ejecutar una función o devolver el contenido de una variable.

Evaluación

- Supongamos que la variable CONT representa un 5.
- Si colocamos la palabra CONT en el sistema, automáticamente la evaluará y nos devolverá un 5.
- Si colocamos la lista (SUMA 5 6) en el sistema, pensará que queremos ejecutar la función llamada SUMA, y que debe regresar un resultado.
- Como todo en Scheme es una función, siempre que colocamos algo en el PROMPT, el sistema trata de evaluarlo.

Apóstrofe

- Si queremos que el sistema no realice la evaluación, tenemos que indicarlo con un apóstrofe.
- Si colocamos la lista '(SUMA 5 6)', el sistema simplemente la toma como una lista de 3 elementos.
- Si tecleamos 'CONT, el sistema lo toma como el átomo CONT (y no como la variable CONT)

Comentarios

- Los comentarios se utilizan para auto documentar un programa.
- Un comentario en Scheme es cualquier cosa que se escriba después de un punto y coma (;).
- El intérprete de Scheme ignora todo lo que siga a un punto y coma.
- También hay comentarios de bloque #| y |#.
- Ejemplo:
; este es un comentario
#| esto también es un comentario |#

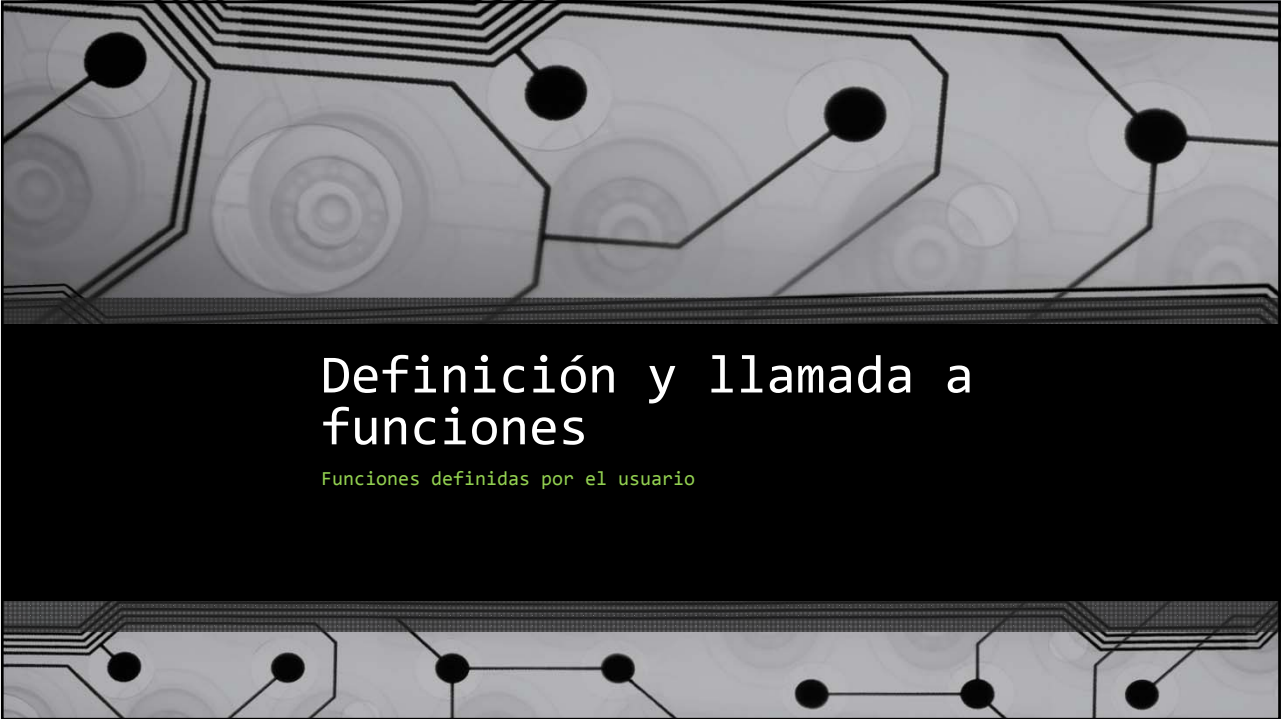


Funciones básica para el manejo de datos

Scheme (Racket)

Funciones básicas

- Scheme (Racket) define muchas funciones integradas del lenguaje.
- A continuación veremos algunas funciones básicas que sirven para hacer programas complejos.
- Sin embargo, las funciones pre definidas son muchísimas y todas vienen contenidas en el Reference Manual de Racket (seleccionando Help>Racket Documentation).
- En todos los proyectos se pueden usar todas las funciones de Scheme, aún y cuando no se hayan visto en esta introducción, solamente cuidando respetar la pureza funcional del lenguaje, es decir, NO CICLOS, NO SECUENCIAS DE INSTRUCCIONES y NO ASIGNACIÓN.



Definición y llamada a funciones

Funciones definidas por el usuario

Llamada a funciones

- Típicamente una llamada a función tiene la forma:
(<identificador> <expresión>*)
- donde la secuencia de expresiones determina el número de parámetros reales pasados a la función referenciada por el identificador.

DEFINE: Definiciones globales

- Hay 2 tipos de definiciones globales:
 - Nombres, con el siguiente formato:
(DEFINE <nombre> <expresión>)
 - Funciones, cuyo formato es:
(DEFINE (<identificador> <identificador>*) <expresión>+)
- La definición de nombres, asigna el valor que resulta de la <expresión> al <nombre>, el cual se puede usar posteriormente.
 - Ejemplo: (DEFINE CONT (+ 2 4))

Funciones del usuario

- Scheme permite que el usuario pueda definir sus propias funciones.
- Cuando un usuario define sus propias funciones, estas quedan disponibles en el sistema, mientras no se define alguna otra con el mismo nombre.
- Esto permite que existan muchos dialectos de Scheme.
- Un PROGRAMA en Scheme está formado por una serie de estas funciones que se llaman unas a otras.

DEFINE: Definiendo funciones

- Su formato es:

```
(DEFINE (<identificador> <identificador>*) <expresión>+ )
```

O más claramente:

```
(DEFINE (<nombre> <lista-de-parámetros>)
```

```
    <expresión 1>
```

```
    <expresión 2>
```

```
    ...
```

```
)
```

- Cuando se invoca la función con sus parámetros, el sistema realiza las acciones de su cuerpo y regresa el resultado de la última expresión.
- Por esta razón, lo más común es colocar sólo una expresión.

Ejemplo

- Una función que recibe dos números y nos regresa su suma:

```
• (DEFINE (suma a b)
```

```
    (+ a b)
```

```
)
```

- Desde luego que una función puede llamar a otra ya definida.

```
• (DEFINE (suma-y-resta a b)
```

```
    (LIST (suma a b) (- a b))
```

```
)
```

Endentación

- En Scheme es fácil confundirse con los paréntesis que abren y los que cierran.
- Existen algunos editores que permiten colocarse sobre un paréntesis y parpadea su paréntesis complementario.
- La recomendación principal es usar endentación y colocar el que abre a la misma altura que su correspondiente que cierra.

Cargando un archivo

- Una función se puede teclear directamente en el *prompt* del sistema, sin embargo, si hay errores se tendría que teclear nuevamente la función completa.
- Esto se puede evitar escribiendo todas las funciones en un archivo de texto.
- Se acostumbra ponerle la extensión `rkt` para indicar que es un archivo de racket.
- El archivo se carga al sistema oprimiendo `RUN` o con la instrucción
(enter! "<archivo>")
- Si está en el mismo folder que el sistema basta con el nombre, de no ser así, se requiere todo el *path*.

Módulos

- Algunas funciones de Scheme no se encuentran cargadas en el sistema principal sino que están en módulos separados.
- En ese caso, hay que cargarlas.
- Un ejemplo es la función **enter!**, la cual se encuentra en el módulo **racket/enter**.
- Para cargarlo, se ejecuta la función:
 - (require racket/enter)
- Una vez hecho esto, ya se pueden usar todas las funciones de dicho módulo.

Directorios

- Para poder ver el directorio actual se usa la instrucción:
 - (current-directory)
- Si se quiere cambiar de directorio se usa la función:
 - (current-directory <path>)
- <path> se obtiene a partir de la función:
 - (string->path <string>), donde <string> es un string que contiene el path deseado.
- Ejemplo para cambio de directorio:
 - (current-directory (string->path "C:\\Users\\Profesor\\miracket"))
 - Teniéndose que usar caracteres escapados para \\

CAR y CDR

- Dos de las funciones fundamentales para la manipulación de elementos de una lista son:
 - CAR: regresa el primer elemento de una lista (sin modificar la lista)
 - CDR: regresa la lista resultante de quitar el primer elemento a la lista actual (sin modificar la lista original)
- Su formato es:
 - (CAR <lista>) y (CDR <lista>)

Nota histórica sobre CAR y CDR

- Su nombre proviene de la implementación de LISP en la IBM 704.
- CAR la instrucción original daba el contenido de la porción de dirección de memoria de un registro (Content of the Adreess Portion of a Register).
- CDR la instrucción original daba el “Content of the Decrement portion of a Register”.

Ejemplos

- (CAR '(a b c d e)) regresa una A
- (CDR '(a b c d e)) regresa la lista (b c d e)
- (CAR '((1 2 3) a b c)) regresa la lista (1 2 3) porque es el primer elemento de la lista
- (CDR '((1 2 3) a b c)) regresa la lista
(a b c)

Ventaja de usar CAR y CDR

- La ventaja de usar CAR y CDR es que se pueden combinar usando más letras de las de en medio.
- CAAR obtiene el CAR del CAR, lo cual también se podría poner como (CAR (CAR
- CADR obtiene el CAR del CDR
- CAAAR obtiene el CAR del CAR del CAR
- CDADR obtiene el CDR del CAR del CDR

Ejemplos

- (CAR (CAR '((1 2 3) a b c))), devuelve un 1 porque el primer CAR (el más interno) regresa la lista (1 2 3) y luego a ese resultado se le aplica el CAR, lo que regresa su primer elemento que es el 1.
- (CAAR '((1 2 3) a b c)) hace lo mismo
- (CADR '((1 2 3) a b c)), regresa una A

CONS y APPEND

- Permiten agregar elementos a una lista:
 - CONS agrega un elemento al inicio de una lista
 - APPEND pega dos listas en una sola, en el orden en el que se le den
- Formato:
 - (CONS <elemento> <lista>)
 - (APPEND <lista₁> <lista₂>)
- Ejemplo:
 - (CONS 'a '(b c d)) regresa (a b c d)
 - (APPEND '(a b) '(1 2 3)) regresa (a b 1 2 3)
 - (CONS (CAR '(a b c)) (CDR '(a b c))) regresa (a b c)

LIST

- Esta instrucción toma n elementos y hace una lista con todos ellos.
- Los elementos pueden ser átomos o listas.
- Formato:
`(LIST <elem1> <elem2>... <elemn>)`
- Ejemplo:
`(LIST 'a 'b '(1 2))` regresa `(a b (1 2))`
`(LIST 'a)` regresa `(a)`

Expresiones

- Como en todo lenguaje de programación, en Scheme hay dos tipos de expresiones:
 - Aritméticas: devuelven un valor numérico
 - Lógicas: devuelven un valor booleano (predicados)
- En Scheme, todos sus operadores se usan como funciones (notación prefijo).
- Una expresión puede ser: un átomo, un número o una expresión.

Operadores

- Son símbolos especiales que se utilizan para operar dos o más expresiones.
- En realidad, todos ellos siguen siendo funciones.
- Operadores aritméticos: `*`, `+`, `/`, `-`
- Operadores lógicos: `NOT`, `AND`, `OR`
- Ejemplos:
 - `(+ 2 3)` regresa un 5
 - `(/ 3 2)` regresa 3/2
 - `(/ 3 2.0)` regresa 1.5
 - `(AND #t null)` regresa null o `#f`

Predicados

- Los predicados son expresiones que dan como resultado un valor verdadero o falso.
- Por convención, los nombres de los predicados en Scheme terminan con un signo de interrogación `?`.
- Ejemplos de algunos pre definidos:
 - **`number?`**, verifica si el argumento es un número.
 - **`symbol?`**, verifica si el argumento es un símbolo.
 - **`equal?`**, verifica si los argumentos son estructuralmente iguales.
 - **`null?`**, verdadera si el argumento es lista vacía.
 - **`list?`**, verdadera si el argumento es una lista.

Operadores relacionales

- Son un tipo especial de operadores booleanos (devuelven valores booleanos) pero pueden operar con valores numéricos.
 - equal?: expresiones
 - =, <=, <, >=, >: números
- Ejemplos:
 - (< 2 3) regresa T, (= 2 3) regresa NULL, (equal? 'a 'a) regresa T

Funciones como parámetros

- Las funciones en Scheme son objetos VIP, eso significa que tiene el mismo trato que cualquier otro objeto.
- Uno de los más impresionantes es que pueden ser mandadas como parámetros de cualquier función, como cualquier variable:
- (DEFINE (opera f a b)
 - (f a b)
 -)
- Si la función anterior se llama con (opera * a b), realiza la multiplicación, si se llama con (opera / a b), realiza la división, etc.

MEMBER

- Su formato es:
(MEMBER <elemento> <lista>)
- Regresa #t si el elemento es un miembro de la lista. En realidad, regresa la lista restante a partir del elemento dado.
- Regresa #f si el elemento no es miembro de la lista.
- Ejemplos:
 - (MEMBER 'a '(1 2 a 3 4)) regresa (a 3 4), que como es no vacía, es equivalente a #t
 - (MEMBER 'b '(1 2 a 3 4)) regresa #f

Algunas funciones útiles

- (LENGTH A), regresa la longitud de la lista A.
- (REVERSE A), regresa la lista A al revés.
- (MAP F L), aplica la función F a todos los elementos de la lista L y regresa la lista con los resultados de dicha aplicación. (ver LIST ITERATION para más funciones de este tipo).
- (FOLDL Op V L), aplica el operador Op a todos los elementos de la lista L, teniendo como valor inicial V (como REDUCE).
- Hay decenas más (ver documentación).



Importancia

- Las instrucciones de entrada y salida son fundamentales para que se pueda dar la interacción entre el usuario y el programa de computadora.
- Las instrucciones más simples realizan la entrada (lectura) de datos por medio del teclado y la salida (escritura) por medio de la pantalla.
- Estas operaciones también se pueden hacer a un archivo.

Instrucciones de E/S

- Para escribir algo a la pantalla hay dos funciones:
 - (WRITE <expresión>)
 - (DISPLAY <expresión>), para la pantalla.
- Para leer del teclado, la función es READ, la cual se usa dentro de un DEFINE:
 - (DEFINE <nombre> (READ))
 - Esta instrucción detiene el programa y espera que el usuario teclee un valor, asignándolo al nombre dado.

Escritura

- Su formato es: (WRITE <expresión>) o (DISPLAY <expresión>)
- La diferencia es que DISPLAY se usa para la pantalla y WRITE para archivos. El archivo por default es la pantalla.
- Escribe en la pantalla la expresión que tiene como parámetro (puede ser una lista).
- También se le puede colocar una expresión o una variable y lo que imprime es el resultado de la expresión o el valor de la variable.
- Ejemplos:
 - (DISPLAY '(Hola Mundo)), imprime Hola Mundo
 - (WRITE A)

Lectura

- La lectura se realiza como si fuera un DEFINE, indicándole que el valor se tomará del teclado. Esto se logra con la función (READ)
- La lectura se hace desde un archivo. El archivo por default es la pantalla.
- Su formato es: (DEFINE <nombre> (READ))
- Asigna al nombre el valor que se teclee hasta el ENTER
- Ejemplos:
 - (DEFINE A (READ)) le asigna al nombre A lo que se teclee.

Nota sobre entrada y salida

- Aunque existen las instrucciones para leer del teclado y escribir a la pantalla, su mayor uso es en la lectura o escritura a archivos, donde se usan WRITE y READ.
- Casi nunca se usan para la pantalla y el teclado.
- En este caso, normalmente, la lectura se hace por medio de los parámetros y la escritura por medio del valor regresado por una función.



Condicionales

- Los condicionales nos permiten ir por dos o más caminos diferentes dependiendo del cumplimiento o no de cierta condición.
- Hay tres condicionales principales en Scheme:
 - IF
 - CASE (varios valores para una variable)
 - COND (IF's anidados)

IF

- Su formato es:
 (IF <expresión-lógica> <expresión-para-verdadero> <expresión-para-falso>) o
 (IF <condición> <acción₁> <acción₂>)
- Si se cumple la condición ejecuta la acción₁, si no se cumple ejecuta la acción₂.
- Ejemplos:
 - (IF (< A B) (DISPLAY '(MENOR A QUE B))
 (DISPLAY '(MAYOR O IGUAL A QUE B)))
- Es importante notar que en <acción> siempre va una sola función (sólo una).

Ejercicios

- Hacer una función que reciba 3 calificaciones y nos regrese un string diciendo “aprobado” o “reprobado”, de acuerdo a su promedio (< 70, reprobado).
- Hacer una función que reciba un año y nos diga si es bisiesto o no (se debe investigar la función adecuada).
- Hacer una función que reciba 3 números y nos regrese el mayor de los 3 (extraño la asignación).
- Hacer una función que reciba una lista de 3 números y nos regrese la lista ordenada (y los ciclos también los extraño).

CASE

- Es una función que agrupa muchos IFs anidados con varios valores para la misma expresión.
- Su formato es:
 - (case <expresión-de-prueba> { [(<valores>+) <expresión>+] }*)
O más claro
 - (CASE <nombre>
 - [<valores₁> <acción₁>]
 - [<valores₂> <acción₂>]
 - ...
 - [<valores_n> <acción_n>])

Ejemplo CASE

```
(CASE A
  ['A (PRINT '(es una A))]
  ['B (PRINT '(es una B))])
```

```
(CASE CONT
  [(1) (DISPLAY "es un uno")])
  [(2) "es un dos"]
  [ELSE "no es ni 1 ni 2"])
```

```
(CASE(- 7 5)
  [(1 2 3) "pequeño "]
  [(10 11 12) "grande "])
```

Ejercicios

- Hacer una función que reciba un string y dos números. En base al string debe realizar la operación correspondiente sobre los números.

(opera “suma” a b)

regresa la suma de a y b.

- Hacer una función que reciba un string indicando un mes y nos diga cuántos días tiene.

COND

- Es una instrucción que agrupa varios IFs anidados, con diferentes condiciones.
- Su formato es:

- (cond { [<expresión-de-prueba> <expresión>*] }*)

o más claro

- (COND

[<condición₁> <acción₁₋₁> <acción₁₋₂>...]

[<condición₂> <acción₂₋₁> <acción₂₋₂>...]

...

[<condición_n> <acción_{n-1}> <acción_{n-2}>...]

)

Ejemplo COND

- Su principal ventaja es que permite tener varias condiciones y varias acciones por condición.

- (COND

```
[(= A 1) (WRITE '(es un uno))]
```

```
[(= A 2) (WRITE '(es un dos))]
```

```
)
```

Si A es igual a 1, imprime “es un uno”. Si es igual a 2, imprime “es un dos”.

Ejercicio

- Hacer una función en Scheme que represente la siguiente función matemática:

$$f(x) = \begin{cases} x + 2 & \text{si } x < -1 \\ 1 & \text{si } -1 \leq x < 0 \\ -x^2 + 1 & \text{si } x \geq 0 \end{cases}$$

Solución

```
(DEFINE (seccionada x)
  (COND [(< x -1)                ; x < -1: x+2
        (+ x 2)]
        [(and (>= x -1) (< x 0)) ; -1<=x < 0 : 1
        1]
        [(>= x 0)                ; 0<=x : -(x^2)+1
        (+ (- (sqr x)) 1) ]))
```

Asignación local

Asignación Local

- Hay al menos 3 formas de hacer asignación local en Scheme:
 - Define
 - Let
 - Let*

DEFINE

- Su sintaxis es la misma que ya se vio.
- Todas las definiciones dentro de una función son locales a ella.
- Las funciones definidas dentro de una función siempre deben estar realizadas al inicio de la función.

LET

- La sintaxis del constructor LET es:
 - $(\text{LET}(\{ [\text{<identificador> <expresión>}]^* \} \text{<expresión>}+))$
 - O más claramente:
 - $(\text{LET}([x_1 E_1] [x_2 E_2] \dots [x_k E_k]) F)$
 - Para Little Quilt sería:
 - **let val** $x_1 = E_1$ **val** $x_2 = E_2 \dots$ **in** F **end**
- Para determinar su valor primero se evalúan las expresiones E_1, E_2, \dots, E_k y al final se evalúa la expresión F donde x_i representa el valor de E_i .

Ventajas del LET

- Una ventaja de LET sobre DEFINE es que puede ser colocada en cualquier lugar dentro de una expresión y no sólo al principio de la función, como DEFINE.
- Además, con LET se pueden hacer múltiples asignaciones al mismo tiempo, en lugar de hacer un DEFINE para cada asignación.

Subexpresiones

- El constructor LET permite nombrar subexpresiones:
- Ejemplo:
 - (LET ([tres-cua (cuadrado 3)]
[cuatro-cua (cuadrado 4)])
(+ tres-cua cuatro-cua))

Ejemplo

```
(LET ([x 1] [y 2])
  (DISPLAY (STRING-APPEND "La suma de "
    (number -> string x)
    " más "
    (number -> string y)
    " es: "
    (number -> string (+ x y)))))
```

La suma de 1 más 2 es: 3

> (+ x y)

reference to an identifier before its definition : x

LET*

- Una variante secuencial del constructor LET se escribe con la palabra reservada **LET***.
- A diferencia de **LET**, la cual evalúa todas las expresiones E_1, E_2, \dots, E_k , antes de asociar cualquiera de las variables, **LET*** asocia x_i al valor E_i antes de que se evalúe E_{i+1} .

Ejemplo

```
( LET* ([x 1] [y 2] [z (+ x y)])
  (PRINTF "La suma de ~a y ~a es: ~a" x y z))
```

La suma de 1 y 2 es: 3

NOTAS:

- Así es como funciona el Let en Little Quilt.
- Se está usando la función **PRINTF** que permite introducir variables dentro de un string.

Funciones Anónimas

Funciones Anónimas: Bloques lambda

- La sintaxis de una función es:
`(define (<nombre> <parámetros>) <expresión>)`
- La asociación de un nombre de función con un valor de función es más clara a partir de la siguiente sintaxis:
`(define <nombre de función> <valor de función>)`
- Para usar esta sintaxis, necesitamos una notación para valores de función, la cual tiene Scheme:
`(lambda (<parámetros formales>) <expresión>)`
- La tradición de usar lambda para definir funciones anónimas se remonta al cálculo lambda de Alonzo Church.

Ejemplos

- La función para elevar un número al cuadrado se puede escribir:

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

- En ese caso, si se llama a (cuadrado 5), regresa 25.
- La función anónima se puede aplicar directamente a 5:

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

NOTA:

En cualquier lugar donde se pueda usar un nombre de función, se puede usar una función anónima.

Funciones de orden superior

- Se dice que una función es de orden superior, si sus argumentos o sus resultados son funciones.
- Las herramientas para construir nuevas funciones a partir de otras más simples son funciones de orden superior.
- Un ejemplo es la función de orden superior MAP, la cual es un iterador, ya que recibe una función que aplica a una lista.
- En Scheme es muy simple hacer que una función reciba como parámetros otra función.

NOTA:

Un iterador es una función que cicla o se repite a través de los elementos de una lista y hace algo con cada elemento.

Ejemplo

- La función que reciba un operador aritmético y regrese la operación indicada, aplicada a sus otros dos parámetros (que debe ser números):

```
(define (opera f a b)  
  (f a b)  
)
```



Ciclos

Recursión

- Los primeros días de LISP fueron sin asignación ni ciclos.
- La única operación con que se contaba para hacer asignación y ciclos era la recursión.
- Scheme cuenta con recursión como una de sus principales herramientas.
- Recordemos los tres puntos fundamentales de la recursión:
 - Llamarse a sí misma
 - Tener un argumento más simple
 - Condición de terminación

Ejercicios

- Hacer un programa que reciba dos números y nos regrese el primero multiplicado por el segundo pero usando sólo sumas.
- Hacer un programa que reciba un número y nos devuelva su factorial.
- Hacer un programa que reciba una lista y nos regrese la suma de todos los elementos de la lista.
- Hacer un programa que reciba una lista y un operador y nos regrese la operación de todos los elementos de la lista (como el REDUCE).
- Hacer un programa que reciba una lista y nos regrese la lista al revés (como el REVERSE).
- Hacer una función que reciba una lista y una función y nos regrese la lista formada por la aplicación de la función a todos los elementos de la lista (como el MAP).
 - (MAP F L), aplica la función F a todos los elementos de la lista L y regresa una lista con los resultados.
 - (FOLDL Op V L), aplica el operador Op a los elementos de la lista L, teniendo como valor inicial V (como REDUCE).



Archivos

- Son una de las estructuras más importantes en todo lenguaje de programación, debido a la necesidad de persistencia de datos en la mayor parte de las aplicaciones.
- Nos permiten guardar la información resultante del proceso, en un dispositivo de almacenamiento secundario (normalmente disco duro).
- La información de entrada también se puede tomar del mismo medio sin que el usuario tenga que usar el teclado.

Tipos de archivos

- Existen dos tipos principales de archivos:
 - Texto
 - Binarios
- Desde luego que existen muchos tipos de archivos más, muchos de ellos con formatos especiales para guardar:
 - Información de un software en particular, e.g. DOC para archivos Word
 - Información de un tipo especial, e.g. JPG para fotos o MPEG para videos
 - Información de un tipo especial de lenguaje que hace referencia a elementos externos, e.g. HTML para páginas Web o LaTeX para escritos matemáticos.

Instrucciones principales

- El manejo de archivo tiene 4 instrucciones fundamentales para su manejo:
 - Abrir el archivo
 - Cerrar el archivo
 - Escribir en el archivo
 - Leer del archivo
- Scheme cuenta con estas 4 instrucciones.
- El alumno tendrá que investigarlas.

Ejercicios

- Hacer un programa que lea un archivo de números y regrese todos los números en una lista.
- Hacer un programa que se le mande una lista de números y escriba los números en un archivo.
- Hacer un programa que haga lo mismos que los dos anteriores pero a un archivo específico (nombre dado por el usuario como un parámetro).

Extras

Modismos (*Idioms*) del Lenguaje

Pares

- En realidad, lo que regresa una función `cons` es un par (llamado celda cons)
- Un par es un tupla de dos elementos (pareja ordenada), separados por punto (`.`)
`(cons 'a 'b)` regresa `'(a . b)`
- El primer elemento de un par se obtiene con `car` y el segundo con `cdr`.
- Una lista es en realidad una secuencia de pares en donde el último termina en `null`.
`(cons 'a (cons 'b (cons 'c null)))` regresa `'(a b c)`
- Ejemplo:
 - `(car (cons 'a 'b))` regresa `'a`
 - `(cdr (cons 'a 'b))` regresa `'b` (no una lista como el `cdr` de una lista, a menos que el segundo sea una lista)

Pares son inmutables

- `Cons cells` (pares) son inmutables.
- ¿Se puede cambiar el contenido de un par?
 - En Racket no se puede (uno de los mayores cambios con respecto a Scheme).
 - Eso es bueno:
 - El alias de listas es irrelevante.
 - Se puede hacer que funciones como `list?` sean más rápidas porque el no ser una lista se determina desde que la cons cell es creada.
- En realidad, como no tenemos asignación esto es irrelevante.

mcons

- Existe una versión para hacer pares mutables y es mediante la función **mcons**.
- **mcons** no genera un par, genera un par mutable que regresa **#f** con **pair?** pero **#t** con **mpair?**.
- Los elementos del par mutable creado por **mcons** se obtienen con **mcar** y **mcdrr**.
- El objetivo de un par mutable es que sus valores se puedan cambiar y esto se logra con:

(set-mcar! mpar valor) y **(set-mcdr! mpar valor)**

Evaluación retrasada (*delayed evaluation*)

- Para cada constructor del lenguaje, la semántica especifica cuándo una subexpresión toma un valor.
- En ML, Racket, Java y C:
 - Los argumentos de las funciones son “eager”, es decir, se llaman por valor.
 - Se evalúan una vez antes de que la función se llame.
 - Las ramas de los condicionales no son “eager”, su evaluación es retrasada hasta que se cumple o no la condición.
- Esto es muy importante, en el siguiente ejemplo la llamada a **factorial-bad** se cicla:

```
(define (my-if-bad x y z)
  (if x y z))
(define (factorial-bad n) % nunca llama a my-if-bad porque primero evalúa los argumentos
  (my-if-bad (= n 0)
              1
              (* n (factorial-bad (- n 1)))))
```

Thunk

- Para retrasar la evaluación de una expresión bastará ponerla dentro de una función.
- Una función de cero argumentos usada para retrasar la evaluación se llama un thunk.
- Se puede usar como verbo (igual que curry), “thunk the expression”.
- La solución a lo anterior es (pero es ridículo “wrappear” el `if` de esta forma):

```
(define (my-if x y z)
  (if x (y) (z)))
(define (fact n)
  (my-if (= n 0)
    (lambda() 1)
    (lambda() (* n (fact (- n 1))))))
```

El punto clave

- Evaluar una expresión `e` para obtener un resultado:

`e`

- Una función que cuando es llamada evalúa `e` y regresa el resultado (función de cero argumentos para `thunking`):

`(lambda () e)`

- Llama el thunk y entonces evalúa `e` para dicho thunk:

`(e)`

Evitando cálculos muy costosos

- Los thunks permiten evitar cálculos costosos si estos no son necesarios.
 - Son buenos si se usan una vez
 - Son costosos si terminas usando los thunks más de una vez
- En general, no se sabe cuántas veces se va a requerir un resultado.
- Lo ideal sería que si se requiere el resultado más de una vez, no tuviéramos que calcularlo con el thunk en cada ocasión sino, simplemente guardarlo.
- Entonces, si se tiene un cálculo costoso quisiéramos:
 - No computar hasta que se requiera.
 - Recordar la respuesta para tenerla lista de inmediato en futuros usos.

Stream

- Es una secuencia infinita de valores:
 - No se puede hacer un stream generando todos los valores
 - Idea principal: usar un thunk para retrasar la creación de la mayoría de la secuencia
- La división del trabajo, un concepto poderoso:
 - El productor del stream sabe cómo crear cualquier cantidad de valores
 - El consumidor del stream sabe cuántos valores pedir

Usando streams

- Vamos a representar un stream usando pares y thunks.
- Sea el stream un thunk que cuando se llame regrese un par:
'(siguiente-respuesta . siguiente-thunk)
- De esta forma, dado un stream *s*, el cliente puede generar cualquier número de elementos:
 - Primero: `(car (s))`
 - Segundo: `(car ((cdr (s))))`
 - Tercero: `(car ((cdr ((cdr (s)))))`

Ejemplo de uso de streams

- La siguiente función regresa tantos elementos de stream como sean necesarios hasta que encuentre uno que para el cual el tester no regrese #f:
 - Se escribe con una función de ayuda (helper function) con recursión de cola.


```
(define (number-until stream tester)
  (letrec ([f (lambda (stream ans)
    (let ([pr (stream)])
      (if (tester (car pr))
          ans
          (f (cdr pr) (+ ans 1))))))]
    (f stream 1)))
```
 - `(stream)` genera el par.
 - Se pasa recursivamente `(cdr pr)` y el thunk para el resto de la secuencia infinita.

Definiendo streams

- Codificar un stream en un programa es fácil, sólo se requieren pares y thunks.
- ¿Cómo se crea el siguiente thunk?: ¡con recursión!
 - Hacer un thunk que produce un par donde el cdr es el siguiente thunk.
 - Una función recursiva puede regresar un thunk en el cual la llamada recursiva no sucede hasta que el thunk es llamado.

Ejemplos

```
(define ones (lambda () (cons 1 ones)))
(define nats
  (letrec ([f (lambda (x)
                (cons x (lambda () (f (+ x 1)))))])
    (lambda () (f 1))))
(define powers-of-two
  (letrec ([f (lambda (x)
                (cons x (lambda () (f (* x 2)))))])
    (lambda () (f 2))))
```

Stream-maker

```
(define (stream-maker fn arg)
  (letrec ([f (lambda (x)
                (cons x (lambda () (f (fn x arg))))))]
    (lambda () (f arg))))

(define nats2 (stream-maker + 1))
(define powers2 (stream-maker * 2))
```

Haciéndolo mal

- Usar una variable antes de ser definida:

```
(define ones-really-bad (cons 1 ones-really-bad))
```

- Cae en un ciclo infinito, haciendo una lista de longitud infinita:

```
(define ones-bad (lambda () cons 1 (ones-bad)))
```

```
(define (ones-bad)(cons 1 (ones-bad)))
```

- Este sí es un stream: el thunk regresa un par cuyo cdr es un thunk:

```
(define ones (lambda () (cons 1 ones)))
```

```
(define (ones)(cons 1 ones))
```


Memoization

- Si una función no tiene efectos colaterales y no lee memoria mutable, no tiene caso calcularla dos veces con los mismos argumentos:
 - Se puede mantener un cache de resultados previos.
 - La ganancia net es que 1) mantener un cache es más barato que recalcular todo, y 2) los resultados en el cache son reusados.
- Para funciones recursivas, la memoization puede llevarnos a crear programas exponencialmente más rápidos:
 - Está relacionada con la técnica algorítmica de Programación Dinámica.

Cómo hacer memoization

- Se requiere un cache (mutable) que pueda ser utilizado por todas las llamadas:
 - Debe ser definido fuera de la(s) función(s) que lo usa(n).
- Las listas asociativas (lista de pares) son estructuras de datos simples pero sub-óptimas para hacer un cache. Hay formas mejores.
- (**assoc v lst**) toma una lista **lst** de pares y localiza el primer elemento en ella cuyo **car** es igual a **v** de acuerdo a la función **is-equal?**:
 - Si el elemento existe el par es regresado
 - De otra forma regresa **#f**

Ejemplo fibonacci1: normal

```
(define (fibonacci1 x)
  (if (or (= x 1) (= x 2))
      1
      (+ (fibonacci1 (- x 1))
          (fibonacci1 (- x 2)))))
```

Ejemplo fibonacci3: memoization

```
(define fibonacci3
  (letrec([memo null]
    [f (lambda (x)
          (let ([ans (assoc x memo)])
            (if ans
                (cdr ans)
                (let ([new-ans (if (or (= x 1) (= x 2))
                                     1
                                     (+ (f (- x 1))
                                         (f (- x 2)))]))
                  (begin
                     (set! memo (cons (cons x new-ans) memo))
                     new-ans)))]))
    f))
```

Referencias

- [1] E. Navas. [Programando con Racket 5](#). Versión 1.0 (2010).
- [2] Racket 6.1. Documentation.
- [3] D. Grossman. Programming Languages Course. Coursera/University of Washington (2013).