

4.7 Operaciones sobre Objetos en Scheme

Equivalencias y predicados de
tipos, listas, números, caracteres,
strings y vectores

Ejemplos de Equivalencia

```
(eq? 'a 'a)           ;=> #t
(eq? 3.1 3.1)         ;=> ()
(eq? (cons 'a 'b)(cons 'a 'b)) ;=> ()

(eqv? 'a 'a)          ;=> #t
(eqv? 3.1 3.1)        ;=> #t
(eqv? (cons 'a 'b)(cons 'a 'b)) ;=> ()

(equal? 'a 'a)         ;=> #t
(equal? 3.1 3.1)       ;=> #t
(equal? (cons 'a 'b)(cons 'a 'b)) ;=> #t
```

a) Equivalencias

- `(eq? obj1 obj2)`
retorno: *#t si son idénticos*
- `(eqv? obj1 obj2)`
retorno: *#t si son equivalentes*
- `eqv?` es similar a `eq?`, salvo que no es dependiente de la implementación, pero es algo más costoso.
- `eq?` no permite comparar en forma fiable números

b) Predicados de Tipos

- `(boolean? obj)`
retorno: *#t si obj #t o #f*
- `(null? obj)`
retorno: *#t si obj #f o lista vacia*
- `(pair? obj)`
retorno: *#t si obj es un par*

equal?

- `(equal? obj1 obj2)`
retorno: *#t si tienen la misma estructura y contenido*
- `equal?` es similar a `eqv?`, salvo que se aplica también para strings, pares y vectores.
- Es más costoso que `eqv?` y `eq?`,

Tipos de Números

- `(number? obj)`
retorno: *#t si obj es número*
- `(complex? obj)`
retorno: *#t si obj es número complejo*
- `(real? obj)`
retorno: *#t si obj es número real*
- `(rational? obj)`
retorno: *#t si obj es número racional*
- `(integer? obj)`
retorno: *#t si obj es número entero*

Otros Tipos

- `(char? obj)`
retorno: `#t` si `obj` es un caracter
- `(string? obj)`
retorno: `#t` si `obj` es un string
- `(vector? obj)`
retorno: `#t` si `obj` es un vector
- `(symbol? obj)`
retorno: `#t` si `obj` es un símbolo
- `(procedure? obj)`
retorno: `#t` si `obj` es un procedimiento

Otros Operadores con Listas

- `(list-ref list n)`
retorno: elemento `#n` (basado en 0) de `list`
- `(list-tail list n)`
retorno: el resto de `list` a partir de elemento `#n`
- `(append list ...)`
retorno: concatenación de `list ...`
- `(reverse list)`
retorno: `list` en orden invertido

c) Listas

- El par es la estructura más fundamental de los tipos de objetos
- Los pares normalmente se usan para construir listas
- Las listas son secuencias ordenadas donde cada elemento ocupa el `car` de un par y el `cdr` del par del último elemento es la lista vacía.
- Si no se cumple lo anterior, es una lista impropia

Membresía en Listas

- `(memq obj list)`
retorno: el resto de `list` a partir de elemento `obj` si es encontrado, sino `#f`
- `(memv obj list)`
retorno: el resto de `list` a partir de elemento `obj` si es encontrado, sino `#f`
- `(member obj list)`
retorno: el resto de `list` a partir de elemento `obj` si es encontrado, sino `#f`
- Usan `eq?` y `eqv?` y `equal?` respectivamente

Operadores Básicos con Listas

- `(list obj ...)`
retorno: una lista de elementos `obj ...`
- `(list? obj)`
retorno: `#t` si `obj` es una lista propia
- `(length list)`
retorno: el número de elementos de `list`

d) Listas Asociativos (Hash tables)

- Una lista asociativa (**`alist`**) es una lista propia cuyos elementos son pares.
- Un par tiene la forma (clave . valor)
- Asociaciones son útiles para almacenar información (valor) relacionada con un objeto (clave)

Operadores con Listas Asociativos

- `(assq obj alist)`
retorno: primer elemento de `alist` cuyo `car` es equivalente a `obj`, sino `#f`
- `(assv obj alist)`
retorno: primer elemento de `alist` cuyo `car` es equivalente a `obj`, sino `#f`
- `(assoc obj alist)`
retorno: primer elemento de `alist` cuyo `car` es equivalente a `obj`, sino `#f`
- Usan `eq?` y `eqv?` y `equal?` respectivamente

Operadores Aritméticos

- `(+ num ...)`
retorno: la suma de los argumentos `num...`
- `(- num)`
retorno: cambio de signo de `num`
- `(- num1 num2 ...)`
retorno: sustracción a `num1` de `num2 ...`
- `(* num ...)`
retorno: la multiplicación de `num...`
- `(/ num)`
retorno: recíproco multiplicativo de `num`
- `(/ num1 num2 num3 ...)`
retorno: la división de `num1` por la multiplicación de `num2 ...`

e) Números

- Scheme soporta enteros, racionales, reales y complejos (que definen una jerarquía)
- Los números pueden ser exactos o inexactos
- Enteros y racionales pueden soportar precisión arbitraria
- Números inexactos se representan en punto flotante
- Los complejos se representan como un par, usando coordenadas polares o rectangulares

Predicados con Números

- `(zero? num)`
retorno: `#t` si `num` es cero
- `(positive? num)`
retorno: `#t` si `num` es positivo
- `(negative? num)`
retorno: `#t` si `num` es negativo
- `(even? num)`
retorno: `#t` si `num` es par
- `(odd? num)`
retorno: `#t` si `num` es impar

Operadores con Números

- `(exact? num)`
retorno: `#t` si `num` es exacto
- `(inexact? num)`
retorno: `#t` si `num` es inexacto
- `(= num1 num2 ...)`
retorno: `#t` si los números son iguales
- `(op real1 real2 ...)`
retorno: `#t` si los números cumplen la relación `op`, con `op` en `{<, >, <=, >=}`
(no está definido para complejos)

Operadores con Enteros

- `(quotient int1 int2)`
retorno: el cociente de `int1` y `int2`
- `(remainder int1 int2)`
retorno: el residuo de `int1` y `int2`
- `(modulo int1 int2)`
retorno: el módulo entero de `int1` y `int2`
- `(gcd int ...)`
retorno: máximo común divisor de `num...`
- `(lcm int ...)`
retorno: mínimo común múltiplo de `num...`

Operadores con Reales

- `(truncate real)`
retorno: *truncar a entero hacia cero*
- `(round real)`
retorno: *redondear a entero más cercano*
- `(abs real)`
retorno: *valor absoluto*
- `(max real ...)`
retorno: *máximo de real ...*
- `(min real ...)`
retorno: *mínimo de real ...*

Predicados de Caracteres

- `(char=? char1 char2 ...)`
- `(char<? char1 char2 ...)`
- `(char>? char1 char2 ...)`
- `(char<=? char1 char2 ...)`
- `(char>=? char1 char2 ...)`
retorno: *#t si relación es verdadera, sino #f*
- Relaciones son sensibles a mayúsculas
- Operadores `char-ci@?` son insensibles

Funciones Matemáticas

- `(sqrt num)`
retorno: *raíz cuadrada de num*
- `(exp num)`
retorno: *e elevado a la potencia de num*
- `(log num)`
retorno: *el logaritmo natural de num*
- `(sin num)`
retorno: *el seno de num*
- `(cos num)`
retorno: *el coseno de num*
- `(tan num)`
retorno: *la tangente de num*

Otros Operadores con Caracteres

- `(char-alphabetic? char)`
retorno: *#t si char es letra*
- `(char-number? char)`
retorno: *#t si char es número*
- `(char-lower-case? letter)`
retorno: *#t si char es letra minúscula*
- `(char-upper-case? letter)`
retorno: *#t si char es letra mayúscula*
- `(char->integer char)`
retorno: *código del carácter char*
- `(integer->char int)`
retorno: *carácter con código int*

f) Caracteres

- Representan objetos atómicos (letras, números, caracteres especiales)
- Se escriben con el prefijo `#\`
- Ejemplos:

<code>#\a</code>	carácter 'a'
<code>#\newline</code>	cambio de línea
<code>#\space</code>	espacio en blanco

g) Strings

- Representan cadenas o secuencias de caracteres
- Se emplean en mensajes o buffers de caracteres
- Se escriben entre citación doble; doble citación se introduce en el texto con `\`
- Se indexan con base en 0
- Ejemplo:

```
"Hola"
"\ "Ohhh\ "
```

Predicados de Strings

- `(string=? string1 string2 ...)`
- `(string<? string1 string2 ...)`
- `(string>? string1 string2 ...)`
- `(string<=? string1 string2 ...)`
- `(string=? string1 string2 ...)`
retorno: `#t` si relación es verdadera,
sino `#f`
- Relaciones son sensibles a mayúsculas
- Operadores `string-ci@?` son insensibles

Operadores con Vectores

- `(vector obj ...)`
retorno: un vector con objetos `obj ...`
- `(make-vector n)`
retorno: un vector de largo `n`
- `(vector-length vector)`
retorno: largo de vector
- `(vector-ref vector n)`
retorno: el elemento `n` de vector

Otros Operadores con Strings

- `(string char ...)`
retorno: un string que contiene `char ...`
- `(make-string n)`
retorno: un string de `n` caracteres (indefinido)
- `(string-length string)`
retorno: largo del string
- `(string-ref string n)`
retorno: el carácter `n` (desde 0) del string
- `(string-set! string n char)`
acción: cambia el carácter `n` de `string` a `char`
- `(string-copy string)`
retorno: una copia de `string`
- `(string-append string ...)`
retorno: concatenación de `string ...`

Más Operadores con Vectores

- `(vector-set! vector n obj)`
acción: define elmtó `n` de vector como `obj`
- `(vector-fill! vector obj)`
acción: reemplaza cada elemento de vector con `obj`
- `(vector->list vector)`
retorno: una lista con elementos de vector
- `(list->vector list)`
retorno: una vector con elementos de lista `list`

h) Vectores

- Forma más conveniente y eficiente de listas en algunas aplicaciones
- A diferencia de listas, el acceso a un elemento se hace en tiempo constante
- Al igual que strings, se referencian con base 0
- Elementos pueden ser de cualquier tipo
- Se escriben como lista, pero precedidos por `#`
- **Ejemplo:**
 `#(a b c)`

4.8 Operaciones de Entrada y Salida en Scheme

Entrada y Salida

- Toda E/S se realiza por puertos.
- Un puertos es un puntero a un flujo (posiblemente infinito) de caracteres (e.g. archivo)
- Un puerto es un objeto de primera clase
- El sistema tiene dos puertos por defecto: *entrada* y *salida estándar* (normalmente es el terminal en un ambiente interactivo)
- Cuando se agotan los datos de entrada se retorna el objeto especial `eof`

Ejemplos de Entrada

```
(input-port? `(a b c))      => unspecified
(input-port? (current-input-port)) => #t
(input-port? (open-input-file "f.dat")) => #t

(let ((p (open-input-file "mi_archivo")))
  (let leer ((x (read p)))
    (if (eof-object? x)
        (begin
          (close-input-port p)
          '())
        (cons x (leer (read x)))))))
```

Operaciones de Entrada (1/2)

- `(input-port? obj)`
retorno: `#t` si *obj* es puerto de entrada
- `(current-input-port)`
retorno: puerto de entrada actual
- `(open-input-file filename)`
retorno: nuevo puerto de entrada asociado a archivo *filename*
- `(call-with-input-file file proc)`
acción: abre *file* y llama a *proc* con puerto abierto como parámetro; luego cierra puerto
retorno: valor de evaluación de *proc*
- `(close-input-port input-port)`
acción: cierra puerto de entrada

Otro Ejemplo de Entrada

```
(define read-word
  (lambda (p)
    (list->string
     (let leer_entrada ()
       (let ((c (peek-char p)))
         (cond
          ((eof-object? c) '())
          ((char-alphabetic? c)
           (read-char p)
           (cons c (leer_entrada)))
          (else '()))))))))

;=> read-word

(read-word (current-input-port))
hola
;=> "hola"
```

Operaciones de Entrada (2/2)

- `(eof-object? obj)`
retorno: `#t` si *obj* es fin de archivo
- `(read)`
`(read input-port)`
retorno: próximo objeto de *input-port*
- `(read-char)`
`(read-char input-port)`
retorno: próximo carácter de *input-port* (con consumo)
- `(peek-char)`
`(peek-char input-port)`
retorno: próximo carácter de *input-port* (sin consumo)

Operaciones de Salida (1/2)

- `(output-port? obj)`
retorno: `#t` si *obj* es puerto de salida
- `(current-output-port)`
retorno: puerto de salida actual
- `(open-output-file file)`
retorno: nueva puerto de salida asociado a archivo *file*
- `(call-with-output-file file proc)`
acción: abre *file* y llama a *proc* con puerto abierto como parámetro; luego cierra puerto
retorno: valor de evaluación de *proc*
- `(close-output-port output-port)`
acción: cierra puerto de salida

Operaciones de Salida (2/2)

- `(write obj)`
`(write obj output-port)`
acción: escribe obj a output-port
- `(display obj)`
`(display obj output-port)`
acción: escribe obj a output-port
- `(write-char char)`
`(write-char char output-port)`
acción: escribe char a output-port
- `(newline)`
`(newline output-port)`
acción: escribe nueva línea a output-port

Operaciones Especiales de E/S

- `(transcript-on file)`
acción: inicia registro en file la sesión interactiva de Scheme
- `(transcript-off)`
acción: termina de registrar sesión
- `(load file)`
acción: carga file leyendo y evaluando secuencialmente las expresiones contenidas.

Observaciones a E/S

- **write** escribe objetos que luego pueden ser leídos con **read**
 - Strings se escriben con citación doble
 - Un carácter se escribe con prefijo `#\`
- **display** se usa normalmente para escribir objetos que no serán leídos
 - Strings y caracteres no escriben doble citación o `#\`
- **write-char** y **read-char** escriben y leen un único carácter sin usar `#\`

Ejemplo de E/S

```
(define copy-file
  (lambda (infile outfile)
    (call-with-input-file infile
      (lambda (iport)
        (call-with-output-file outfile
          (lambda (oport)
            (do ((c (read-char iport))
                (read-char iport)))
              ((eof-object? c))
              (write-char c oport)))))))

;=> copy-file

(copy-file "infile" "outfile")
;=> #t
```