

El lenguaje de programación Scheme

En cualquier lenguaje tenemos palabras que se combinan siguiendo ciertas reglas para formar frases con significado. A su vez, estas palabras se forman uniendo las letras de un abecedario. *Scheme*, como lenguaje de programación, utiliza de manera análoga a las palabras los denominados símbolos y estos se forman uniendo las letras del alfabeto (sin distinguir mayúsculas de minúsculas), los dígitos del 0 al 9 y cualquier otro carácter que aparezca en el teclado salvo:

() [] { } ; , " ' ` # \

ya que tienen un significado especial, similar al que tienen los signos de puntuación.

Los caracteres:

+ - .

también son especiales y no deben aparecer en primer lugar en un símbolo. Los números no se consideran símbolos en *Scheme*.

Un símbolo que es usado para representar un valor se denomina variable. El intérprete determinará el significado de cada variable; los números tienen su valor usual.

Siguiendo la análoga con los lenguajes el equivalente en Scheme a las frases son las expresiones, que pueden consistir en un símbolo, un número o una lista, es decir, un paréntesis izquierdo, seguido de expresiones separadas por espacios en blanco, y para terminar un paréntesis derecho. La primera de dichas expresiones debe evaluar a un procedimiento, evaluándose las restantes como los argumentos de este.

Notación

En lo que sigue utilizaremos la siguiente notación al escribir las expresiones:

(procedimiento *expresión₁* ... *expresión_k*)

Es decir, el nombre del procedimiento aparecerá en negrita y los argumentos en itálica. Además, tendremos en cuenta que si el nombre de un argumento es el nombre de un tipo (ver el apartado PREDICADOS DE TIPO), entonces el argumento debe ser del tipo nombrado . Usaremos el siguiente convenio:

<i>z</i>	número complejo	<i>l</i>	lista
<i>x</i>	número real	<i>cter</i>	carácter
<i>n</i>	número entero	<i>cad</i>	cadena
<i>k</i>	número natural	<i>proc</i>	procedimiento
<i>obj, expresión, ex</i>	de cualquier tipo	<i>expresiones</i>	sucesión de expresiones

Utilizaremos los corchetes para denotar expresionesopcionales y los puntos suspensivos para denotar varias ocurrencias.

(procedimiento *obj₁* ... *obj_k*) Indica que procedimiento es de aridad variable.

(procedimiento *obj₁* [*obj₂*]) Indica que procedimiento tiene dos argumentos y el segundo es opcional.

A continuación, enumeraremos las expresiones más usuales seguidas de su valor.

Predicados de tipo

(symbol? <i>obj</i>)	Si <i>obj</i> es de tipo símbolo entonces #t; en otro caso #f.
(procedure? <i>obj</i>)	Si <i>obj</i> es de tipo procedimiento entonces #t; en otro caso #f.
(number? <i>obj</i>)	Si <i>obj</i> es de tipo número entonces #t; en otro caso #f.
(pair? <i>obj</i>)	Si <i>obj</i> es de tipo par punteado entonces #t; en otro caso #f.
(null? <i>obj</i>)	Si <i>obj</i> es la lista vacía entonces #t; en otro caso #f.
(boolean? <i>obj</i>)	Si <i>obj</i> es uno de los valores de verdad (o booleanos), #t o #f, entonces #t; en otro caso #f.
(vector? <i>obj</i>)	Si <i>obj</i> es de tipo vector entonces #t; en otro caso #f.
(char? <i>obj</i>)	Si <i>obj</i> es de tipo carácter entonces #t; en otro caso #f.
(string? <i>obj</i>)	Si <i>obj</i> es de tipo cadena entonces #t; en otro caso #f.
Ningún objeto verifica más de uno de los predicados anteriores; otro predicado útil es:	
(list? <i>obj</i>)	Si <i>obj</i> es de tipo ‘lista’ entonces #t; en otro caso #f.

Predicados de igualdad

(= <i>z₁</i> [<i>z₂</i> ... <i>z_k</i>])	Igualdad numérica entre los argumentos
(eq? <i>obj₁</i> <i>obj₂</i>)	Igualdad simbólica.
(eqv? <i>obj₁</i> <i>obj₂</i>)	Igualdad numérico - simbólica.
(equal? <i>obj₁</i> <i>obj₂</i>)	Igualdad de valores.

Variables y literales

expresion Valor de *expresion*.

(define <i>simbolo</i> <i>obj</i>)	Le asigna a <i>simbolo</i> el valor de <i>obj</i> .
(quote <i>obj</i>) ≡ ' <i>obj</i>	<i>obj</i> .

Los números, caracteres, cadenas y valores de verdad (o booleanos), evalúan a sí mismos por lo que no necesitan **quote**

(let	(Todos los simbolo _i deben ser distintos). Evalúa cada obj _j , en un entorno local, asigna a cada simbolo _s el valor de obj _s y a continuación evalúa de forma consecutiva ex ₁ ...ex _r , devolviendo el valor de la última.
(<i>simbolo</i>₁ <i>obj</i>₁)	
...	
(<i>simbolo</i>_k <i>obj</i>_k)	
ex₁ [ex₂ ...ex_r]	
(let*	En un entorno local, asigna, de manera secuencial, a cada simbolo _s el valor de obj _s y a continuación evalúa de forma consecutiva ex ₁ ...ex _r , devolviendo el valor de la última.
(<i>simbolo</i>₁ <i>obj</i>₁)	
...	
(<i>simbolo</i>_k <i>obj</i>_k)	
ex₁ [ex₂ ...ex_r]	
(letrec <i>variables</i> <i>cuerpo</i>)	Similar a let, pero permite hacer definiciones de procedimientos recursivos.
(set! <i>simbolo</i> <i>expresion</i>)	Asigna el valor de expresión a símbolo, que ya debe tener asignado algún valor. Devuelve un valor no específico.

Expresiones lambda

(lambda <i>argumentos</i> <i>cuerpo</i>)	Devuelve un procedimiento
argumentos puede ser:	
(<i>simbolo</i>₁ ...<i>simbolo</i>_k)	Lista de símbolos (todos distintos) que representan a cada argumento de la función, la cual será, por tanto, de aridad la longitud de dicha lista.
<i>variable</i>	Símbolo que representa a una lista con los argumentos. Por tanto, será de aridad variable
(<i>simb</i>₁ ...<i>simb</i>_k . <i>variable</i>)	Los primeros <i>k</i> argumentos se guardan en las variables simb ₁ a simb _k . El resto de los argumentos se guardan en una lista en variable. El procedimiento es, por tanto, de aridad al menos <i>k</i>

cuerpo: Sucesión de expresiones que describen la función.

(define (<i>simb</i> <i>simb</i> ₁ ... <i>simb</i> _k) <i>cuerpo</i>)	Equivalente a (define <i>simb</i> (lambda (<i>simb</i> ₁ ... <i>simb</i> _k) <i>cuerpo</i>))
(define (<i>simbolo</i> . <i>variable</i>) <i>cuerpo</i>)	Equivalente a (define <i>simbolo</i> (lambda (<i>variable</i> <i>cuerpo</i>)))

(define (*simb s₁* ... *s_k* . *variable*) *cuerpo*) Equivalente a
(define *simb*
(lambda (*s₁...s_k* . *var*)
cuerpo))

Abstracción de procedimientos

(apply *proc ex₁* [*ex₂ ...ex_k*]) (ex_k una lista) aplica *proc* con argumentos *ex₁ ...ex_{k-1}* y los elementos de *ex_k*.

(map *proc l₁* [*l₂ ...l_k*]) Aplica *proc* a cada elemento de *l₁*; si existe más de una lista todas han de tener la misma longitud y aplica *proc* tomando como argumentos un elemento de cada lista, devuelve la lista de los resultados.

(for-each *proc l₁* [*l₂ ...l_k*]) Aplica *proc* a cada elemento de *l₁*; si existe más de una lista todas han de tener la misma longitud y aplica *proc* tomando como argumentos un elemento de cada lista, devuelve un valor no específico

Procedimientos numéricos

(+ [*z₁ ...z_k*]) Suma de los argumentos; sin argumentos, 0.

(- *z₁* [*z₂ ...z_k*]) Resta de los argumentos, asociando por la izquierda; con un solo argumento, -*z₁*.

(* [*z₁ ...z_k*]) Producto de los argumentos; sin argumentos, 1.

(/ *z₁* [*z₂ ...z_k*]) División de los argumentos asociando por la izquierda; con un solo argumento, 1/*z*.

(sqrt *z*) Raíz cuadrada principal de *z* (si *z* es real, la raíz cuadrada positiva).

(abs *x*) Valor absoluto de *x*.

(sin *z*) Seno de *z*.

(cos *z*) Coseno de *z*.

(tan *z*) Tangente de *z*.

(asin *z*) Arcoseno de *z*.

(acos *z*) Arcocoseno de *z*.

(atan *z*) Arcotangente de *z*.

(max *x₁* [*x₂ ...x_k*]) Máximo entre los argumentos.

(min *x₁* [*x₂ ...x_k*]) Mínimo entre los argumentos.

(quotient *n₁ n₂*) (*n₂* distinto de cero), cociente de *n₁* entre *n₂*.

(remainder *n₁ n₂*) (*n₂* distinto de cero), resto de *n₁* entre *n₂*.

(expt *z₁ z₂*) La potencia *z₁z₂* (con 00 = 1).

(exp *z*) La potencia e^{*z*}.

(log z)	Logaritmo en base e de z.
(gcd [n₁ ... n_k])	Máximo común divisor entre los argumentos; sin argumentos, 0.
(lcm [n₁ ... n_k])	Mínimo común múltiplo entre los argumentos; sin argumentos, 1.
(floor x)	Mayor entero menor o igual que x.
(ceiling x)	Menor entero mayor o igual que x.
(truncate x)	Parte entera de x.
(round x)	Entero más cercano a x, en caso de equidistancia número entero par más cercano.
(exact->inexact z)	El número inexacto numéricamente más cercano a z.
(inexact->exact z)	El número exacto numéricamente más cercano a z.

Predicados numéricos

(complex? obj)	Si <i>obj</i> es un número complejo entonces #t; en otro caso #f.
(real? obj)	Si <i>obj</i> es un número real entonces #t; en otro caso #f.
(rational? obj)	Si <i>obj</i> es un número racional entonces #t; en otro caso #f.
(exact? z)	Si <i>z</i> es exacto, entonces #t; en otro caso #f.
(inexact? z)	Si <i>z</i> es inexacto, entonces #t; en otro caso #f.
(integer? obj)	Si <i>obj</i> es un número entero entonces #t; en otro caso #f.
(even? n)	Si <i>n</i> es par entonces #t; en otro caso #f.
(odd? n)	Si <i>n</i> es impar entonces #t; en otro caso #f.
(zero? z)	Si <i>z</i> es el cero entonces #t; en otro caso #f.
(positive? x)	Si <i>x</i> es mayor estricto que cero entonces #t; en otro caso #f.
(negative? x)	Si <i>x</i> es menor estricto que cero entonces #t; en otro caso #f.

Relaciones numéricas

(> x₁ [x₂ ... x_k])	Los argumentos están en orden decreciente.
(< x₁ [x₂ ... x_k])	Los argumentos están en orden creciente.
(>= x₁ [x₂ ... x_k])	Los argumentos están en orden no creciente.
(<= x₁ [x₂ ... x_k])	Los argumentos están en orden no decreciente.

Pares

(cons *obj₁* *obj₂*)

El par cuyo car es *obj₁* y cuyo cdr es *obj₂*

(car *par*)

Primer elemento de *par*.

(cdr *par*)

Segundo elemento de *par*.

Procedimientos sobre listas

(cons *obj* *lista*)

Lista que resulta al incluir *obj* como primer elemento de lista.

(list [*obj₁*
... *obj_k*])

La lista de los argumentos; sin argumentos la lista vacía.

(car *lista*)

Primer elemento de *lista*.

(cdr *lista*)

Lista que resulta al quitarle el primer elemento a lista.

(caar *lista*)

(cadr *lista*)

...

Composiciones de car y cdr.

(cdddar *lista*)

(cddddr *lista*)

(append [*lista₁*
... *lista_k*])

Lista que resulta al unir los argumentos; sin argumentos, la lista vacía.

(reverse *lista*)

Una lista con los mismos elementos que *lista*, pero dispuestos en orden inverso.

(length *lista*)

Longitud de lista.

(list-ref *lista* *k*)

Elemento de lista que ocupa la k-exima posición.

(list-tail *lista* *k*)

Sub-lista de *lista* obtenida eliminando los *k* primeros elementos.

(set-car! *lista* *obj*)

Almacena *obj* como el car de *lista* y devuelve un valor no específico

(set-cdr! *lista* *obj*)

Hace que el cdr de lista apunte a *obj* y devuelve un valor no específico.

Predicados de pertenencia

Teniendo en cuenta que una sublistas de una lista se obtiene por aplicaciones sucesivas de cdr

(memq *obj* *lista*)

Primera sublistas de *lista* cuyo primer elemento es igual que *obj*, comparando con eq?; en otro caso #f.

(memv *obj* *lista*)

Primera sublistas de *lista* cuyo primer elemento es igual que *obj*, comparando con eqv?; en otro caso #f.

(member *obj* *lista*)

Primera sublistas de lista cuyo primer elemento es igual que *obj* comparando con equal?; en otro caso #f.

(assq *obj* *lista-par*)

Primer elemento de lista-par (una lista de pares punteados) cuyo primer elemento es igual que *obj*, comparando con eq?; en otro caso #f.

(assv *obj* *lista-par*)

Primer elemento de lista-par (una lista de pares punteados) cuyo primer elemento es igual que *obj*, comparando con eqv?; en otro caso #f.

(assoc *obj* *lista-par*)

Primer elemento de lista-par (una lista de pares punteados) cuyo primer elemento es igual que *obj*, comparando con equal?; en otro caso #f.

Expresiones condicionales

```
(if test consecuencia
      [alternativa])
(cond
  (test1 [expresiones1])
  ...
  (testk [expresionesk])
  | (else
    [expresionesk+1])]

(case clave
  ((datos1) expresiones1)
  ...
  ((datosk) expresionesk)
  | (else expresionesk+1))]
```

Si *test* tiene como valor #f entonces alternativa (si no existe alternativa entonces un valor no específico), en otro caso consecuencia.

Evalúa *test*₁ ... *test*_k sucesivamente hasta encontrar el primer *test*_i que no tenga como valor #f, en cuyo caso evalúa en orden las *expresiones*_i devolviendo el valor de la última (si no existen devuelve el valor de dicho *test*_i). Si todo *test*_i tiene como valor #f y existe la cláusula else, evalúa en orden *expresiones*_{k+1} devolviendo el valor de la última; si no existen o no existe cláusula else, devuelve un valor no específico.

Evalúa *clave* y compara el resultado obtenido con cada uno de los *datos*_i, sucesivamente. Si encuentra alguno que es igual (comparando con eqv?) evalúa en orden *expresiones*_i devolviendo el valor de la última. Si el valor de clave es distinto a todos los *datos*_i y existe la cláusula else, evalúa en orden *expresiones*_{k+1} devolviendo el valor de la última; si no existe cláusula else, devuelve un valor no específico.

Operadores lógicos

(not <i>obj</i>)	Si <i>obj</i> tiene como valor #f entonces #t; en otro caso #f.
(or [<i>obj</i> ₁ ... <i>obj</i> _k])	Evaluá <i>obj</i> ₁ ... <i>obj</i> _k sucesivamente hasta el primero que no tenga como valor #f y devuelve su valor, en otro caso #f; sin argumentos, #f.
and [<i>obj</i> ₁ ... <i>obj</i> _k])	Evaluá <i>obj</i> ₁ ... <i>obj</i> _k sucesivamente hasta el primero que tenga como valor #f, en otro caso el valor de <i>obj</i> _k ; sin argumentos, #t.

Iteraciones

(do	Cada <i>simbolo</i> ₁ recibe el valor de <i>obj</i> ₁ . En cada iteración del bucle se evalúa <i>test</i> :
((<i>simbolo</i> ₁ <i>obj</i> ₁ [<i>paso</i> ₁])	sí es #f se evalúan sucesivamente y se actualizan los valores de cada <i>simbolo</i> _i según <i>paso</i> _i (cuando existen). Comienza una nueva iteración
...	si no, evalúan <i>ex</i> ₁ ... <i>ex</i> _r sucesivamente devolviendo el valor de la última. Termina el bucle.
(<i>simbolo</i> _k <i>obj</i> _k [<i>paso</i> _k]))	
(test [<i>ex</i> ₁ ... <i>ex</i> _r])	
[<i>expresion</i> ₁	
...	
<i>expresion</i> _J])	

Procedimientos sobre vectores

(make-vector <i>k</i> [<i>obj</i>])	Construye un vector con <i>k</i> elementos iguales a <i>obj</i> , si no existe <i>obj</i> el contenido del vector es indeterminado.
(vector [<i>obj</i> ₁ ... <i>obj</i> _k])	Construye un vector con <i>k</i> elementos, cada uno de los cuales es <i>obj</i> _i .
(vector-ref <i>vector</i> <i>k</i>)	Elemento de <i>vector</i> que ocupa la posición <i>k</i> -exima.
(vector-length <i>vector</i>)	Número de elementos de <i>vector</i> .
(vector->list <i>vector</i>)	Lista con los elementos de <i>vector</i> .
(list->vector <i>lista</i>)	<i>Vector</i> con los elementos de <i>lista</i> .
(vector-set! <i>vector</i> <i>k</i> <i>obj</i>)	Almacena <i>obj</i> en el <i>k</i> -eximo elemento de <i>vector</i> , devuelve un valor no específico.
(vector-fill! <i>vector</i> <i>obj</i>)	Cambia cada elemento de <i>vector</i> por <i>obj</i> . Devuelve un valor no específico.

Procedimientos sobre caracteres

(char->integer <i>caracter</i>)	Código ASCII de carácter.
(integer->char <i>n</i>)	Carácter con código ASCII <i>n</i> .
(char<? <i>cter</i> ₁ [<i>cter</i> ₂ ... <i>cter</i> _k])	Comparación de los valores numéricos de <i>cter</i> _i con <.

(char<=? cter₁ [cter₂ ...cter_k])	Comparación de los valores numéricos de cter _i con <=.
(char>? cter₁ [cter₂ ...cter_k])	Comparación de los valores numéricos de cter _i con >.
(char>=? cter₁ [cter₂ ...cter_k])	Comparación de los valores numéricos de cter _i con >=.
(char=? cter₁ [cter₂ ...cter_k])	Comparación de los valores numéricos de cter _i con =.
(char-ci<? cter₁ [cter₂ ...cter_k])	Comparación de los valores numéricos de cter _i con <, sin distinguir mayúsculas de minúsculas.
(char-ci<=? cter₁ [cter₂ ...cter_k])	Comparación de los valores numéricos de cter _i con <=, sin distinguir mayúsculas de minúsculas.
(char-ci>? cter₁ [cter₂ ...cter_k])	Comparación de los valores numéricos de cter _i con >, sin distinguir mayúsculas de minúsculas.
(char-ci>=? cter₁ [cter₂ ...cter_k])	Comparación de los valores numéricos de cter _i con >=, sin distinguir mayúsculas de minúsculas.
(char-ci=? cter₁ [cter₂ ...cter_k])	Comparación de los valores numéricos de cter _i con =, sin distinguir mayúsculas de minúsculas.
(char-upcase carácter)	Si carácter es minúscula devuelve el mismo en mayúscula; devuelve carácter en otro caso
(char-downcase carácter)	Si carácter es mayúscula devuelve el mismo en minúscula; devuelve carácter en otro caso
(char-upper-case? obj)	Si obj es de tipo carácter mayúscula , entonces #t; en otro caso #f.
(char-lower-case? obj)	Si obj es de tipo carácter minúscula , entonces #t; en otro caso #f.
(char-alphabetic? obj)	Si obj es de tipo carácter alfabético , entonces #t; en otro caso #f.
(char-numeric? obj)	Si obj es de tipo carácter numérico , entonces #t; en otro caso #f.
(char-whitespace? obj)	Si obj es de tipo carácter de espacio en blanco (espacio, tabulación, salto de línea, retorno de carro), entonces #t; en otro caso #f.

Procedimientos sobre cadenas

(string-length <i>cadena</i>)	Número de caracteres de cadena
(make-string <i>k</i> [<i>caracter</i>])	Construye una cadena con <i>k</i> caracteres iguales a carácter, si no existe carácter el contenido de la cadena es indeterminado.
(string [<i>cter1</i> ... <i>cterk</i>])	Construye una cadena de <i>k</i> caracteres, cada uno de los cuales es <i>cter_i</i> .
(string-ref <i>cadena</i> <i>k</i>)	Carácter de cadena que ocupa la posición <i>k</i> -exima.
(string-set! <i>cadena</i> <i>k</i> <i>cter</i>)	Almacena <i>cter</i> como elemento <i>k</i> -eximo de cadena y devuelve un valor no específico.
(substring <i>cadena</i> <i>k1</i> <i>k2</i>)	Trozo de cadena entre las posiciones <i>k1</i> y <i>k2</i> , la primera incluida.
(string-append [<i>cad1</i> ... <i>cadk</i>])	Concatenación de las cadenas <i>cad_i</i> , sin argumentos la cadena vacía.
(string->list <i>cadena</i>)	Lista con los caracteres de cadena.
(list->string <i>lista</i>)	Cadena con los caracteres de lista (todos sus elementos han de ser caracteres).
(string-copy <i>cadena</i>)	Una nueva cadena, copia exacta de cadena.
(string-fill! <i>cadena</i> <i>cter</i>)	Cambia cada elemento de cadena por el carácter <i>cter</i> . Devuelve un valor no específico.
(string<? <i>cad1</i> [<i>cad2</i> ... <i>cadk</i>])	Comparación de <i>cad_i</i> con la extensión lexicográfica de char<.
(string<=? <i>cad1</i> [<i>cad2</i> ... <i>cadk</i>])	Comparación de <i>cad_i</i> con la extensión lexicográfica de char<=.
(string>? <i>cad1</i> [<i>cad2</i> ... <i>cadk</i>])	Comparación de <i>cad_i</i> con la extensión lexicográfica de char>.
(string>=? <i>cad1</i> [<i>cad2</i> ... <i>cadk</i>])	Comparación de <i>cad_i</i> con la extensión lexicográfica de char>=.
(string=? <i>cad1</i> [<i>cad2</i> ... <i>cadk</i>])	Comparación de <i>cad_i</i> con la extensión lexicográfica de char=.
(string-ci<? <i>cad1</i> [<i>cad2</i> ... <i>cadk</i>])	Comparación de <i>cad_i</i> con la extensión lexicográfica de char-ci<.
(string-ci<=? <i>cad1</i> [<i>cad2</i> ... <i>cadk</i>])	Comparación de <i>cad_i</i> con la extensión lexicográfica de char-ci<=.
(string-ci>? <i>cad1</i> [<i>cad2</i> ... <i>cadk</i>])	Comparación de <i>cad_i</i> con la extensión lexicográfica de char-ci>.

(string-ci>=? cad₁ [cad₂ ...cad_k])

Comparación de cad_i con la extensión lexicográfica de char-ci>=.

(string-ci=? cad₁ [cad₂ ...cad_k])

Comparación de cad_i con la extensión lexicográfica de char-ci=.

Procedimientos de entrada y salida

(input-port? obj)
(output-port? obj)

Devuelve #t si obj es un puerto de entrada o salida, respectivamente; en caso contrario devuelve #f.

(current-input-port)
(current-output-port)

Devuelve el puerto de entrada o salida por defecto (inicialmente el teclado y el monitor, respectivamente).

(open-input-file fichero)

Devuelve un puerto capaz de proporcionar caracteres del fichero, que debe existir.

(open-output-file fichero)

Devuelve un puerto capaz de escribir caracteres en fichero, que es creado al evaluar la expresión y, por tanto, inicialmente no debe existir

**(call-with-input-file
fichero proc)** (**call-with-**
output-file *fichero proc*)

Evalúan el procedimiento *proc* con un argumento: el puerto obtenido al abrir *fichero* para entrada o salida, respectivamente

**(with-input-from-file
fichero proc)** (**with-output-**
to-file *fichero proc*)

Se crea un puerto de entrada o salida conectado a chero, convirtiéndose dicho puerto en el puerto por defecto. A continuación, se evalúa *proc* sin argumentos.

(eof-object? obj)

Devuelve #t si obj es un dato de tipo *final* de chero y #f en caso contrario.

(read [puerto])

Devuelve el siguiente objeto de Scheme que se puede obtener del puerto de entrada dado (o el puerto de entrada por defecto, si se omite). Al llegar al *final* del fichero conectado a puerto, devuelve un objeto de tipo *final de fichero*.

(read-char [puerto])

Devuelve el siguiente carácter que se puede obtener del puerto de entrada dado (o el puerto de entrada por defecto, si se omite). Al llegar al final del chero conectado a puerto, devuelve un objeto de tipo final de fichero.

(peek-char [puerto])

Devuelve el siguiente objeto de Scheme que se puede obtener del puerto de entrada dado (o el puerto de entrada por defecto, si se omite), sin actualizar dicho puerto para que apunte al siguiente carácter. Si no hay caracteres disponibles, devuelve un objeto de tipo final de fichero.

(newline [puerto])

Escribe un objeto de tipo final de línea en puerto (o el puerto de entrada por defecto, si Este se omite). Devuelve un valor no específico.

(write *obj* [*puerto*])

Escribe una representación escrita de *obj* en puerto (o el puerto de entrada por defecto, si este se omite). Las cadenas se escriben delimitadas por comillas y los caracteres con la notación #\|. Devuelve un valor no específico.

(display *obj* [*puerto*])

Escribe una representación de *obj* en puerto (o el puerto de entrada por defecto, si este se omite). Las cadenas se escriben sin estar delimitadas por comillas y los caracteres sin la notación #\|. Devuelve un valor no específico.

Nota: `Write` se usa para producir salida legible por el ordenador, mientras que `display` se usa para producir salida legible por una persona.

(write-char *cter* [*puerto*])

Escribe el carácter *cter* (sin usar la notación #\|) en *puerto* (o en el puerto de entrada por defecto, si este se omite). Devuelve un valor no específico.

Procedimientos de la librería gráfica

Para utilizar la librería gráfica que se describe a continuación, es necesario cargarla mediante la siguiente orden: (require-library graphics.ss graphics).

(open-graphics)

Inicializa las rutinas de la librería gráfica. Debe ser utilizada antes que ningún otro procedimiento de la misma.

(close-graphics)

Cierra todas las ventanas. Hasta que no vuelva a utilizarse `open-graphics` no funcionara ninguna rutina grafica

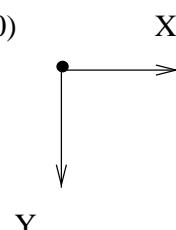
(open-viewport *cad* *n1* *n2*)

Crea una ventana nueva, con nombre *cad*, de *n1* pixels de anchura y *n2* pixels de altura. Devuelve un objeto de tipo ventana.

(close-viewport *ventana*)

Borra la ventana indicada de la pantalla, impidiendo su uso posterior.

Un objeto de tipo posición es la localización de un pixel en un objeto de tipo ventana. El pixel situado en la esquina superior izquierda es el de coordenadas (0,0) y la orientación del eje de coordenadas viene representada en el siguiente dibujo:



(make-posn *n1* *n2*)

Devuelve el objeto de tipo posición de coordenadas *n1* y *n2*.

(posn-x *posicion*)

Devuelve las coordenadas X e Y, respectivamente, de posición.

(posn-y *posicion*)

(posn? obj)	Devuelve #t si obj es un objeto de tipo posición; en caso contrario devuelve #f.
Un color puede representarse de dos formas distintas: como una cadena (con el nombre de este, p.e. red) o como un objeto de tipo rgb. Cualquiera de los procedimientos que toma color como argumento acepta cualquiera de las representaciones	
(make-rgb rojo verde azul)	Dados tres números reales de 0 (oscuro) a 1 (claro) devuelve un objeto de tipo rgb.
(rgb-red rgb)	Devuelve los valores de rojo, verde y azul, respectivamente, de rgb.
(rgb-green rgb)	
(rgb-blue rgb)	
(rgb? obj)	Devuelve #t si obj es un objeto de tipo rgb; en caso contrario devuelve #f.
((draw-viewport ventana) [color])	Dada una <i>ventana</i> devuelve un procedimiento que colorea (resp. borra) el contenido completo de la misma utilizando <i>color</i> o negro, si se omite éste.
((clear-viewport ventana))	
((draw-pixel ventana) posicion [color])	Dada una ventana devuelve un procedimiento que dibuja (resp. borra) un pixel en la misma, en la posición especiada utilizando <i>color</i> ; o negro, si se omite este.
((clear-pixel ventana) posicion)	
((draw-line ventana) posicion1 posicion2 [color])	Dada una ventana devuelve un procedimiento que dibuja (resp. borra) una linea en la misma, conectando las posiciones especificadas utilizando <i>color</i> ; o negro, si se omite este.
((clear-line ventana) posicion1 posicion2)	
((draw-rectangle ventana) posicion altura anchura [color])	Dada una ventana devuelve un procedimiento que dibuja (resp. borra) el borde de un rectángulo en la misma, de <i>altura</i> y <i>anchura</i> dadas, siendo el vértice superior izquierdo el que ocupa la posición especificada, utilizando <i>color</i> ; o negro, si se omite éste.
((clear-rectangle ventana) posicion altura anchura)	
((draw-solid-rectangle ventana) posicion altura anchura [color])	Como el anterior, pero colorea toda la figura utilizando <i>color</i> ; o negro, si se omite éste.
((clear-solid-rectangle ventana) posicion altura anchura)	

((draw-ellipse ventana) <i>posición altura anchura [color])</i>	Dada una ventana devuelve un procedimiento que dibuja (resp. borra) el borde de la elipse inscrita en el rectángulo de altura y anchura dadas, siendo el vértice superior izquierdo el que ocupa la posición especificada, utilizando color; o negro, si se omite este.
((draw-solid-ellipse ventana) <i>posición altura anchura [color])</i>	Como el anterior, pero colorea toda la gura utilizando color; o negro, si se omite este.
((clear-solid-ellipse ventana) <i>posición altura anchura)</i>	
((draw-polygon ventana) <i>l-</i> <i>posiciones posicion [color])</i>	Dada una ventana devuelve un procedimiento que dibuja (resp. borra) el borde de un polígono en la misma, siendo los vértices los elementos de <i>l-posiciones</i> y considerando <i>posición</i> como el desplazamiento de este. Utilizando color; o negro, si se omite Este.
((clear-polygon ventana) <i>l-</i> <i>posiciones posicion)</i>	
((draw-solid-polygon ventana) <i>l-posiciones posicion [color])</i>	Como el anterior, pero colorea toda la gura utilizando color; o negro, si se omite este.
((clear-solid-polygon ventana) <i>l-posiciones posicion)</i>	
((draw-string ventana) <i>posiciones posicion [color])</i>	Dada una <i>ventana</i> devuelve un procedimiento <i>posición cadena [color]</i> que dibuja (resp. borra) una cadena en la misma, siendo el vértice inferior izquierdo el que <i>posición cadena</i> ocupa la posición especificada, utilizando color; o negro, si se omite este.
((clear-string ventana)	
((get-string-size ventana)	Dada una <i>ventana</i> devuelve un procedimiento <i>cadena</i>) que devuelve el tamaño de cadena como una lista de dos números, la anchura y la altura.

Otras funciones de interés

(load nombre-fichero)	Evalúa el contenido de nombre-fichero.
(trace [proc₁...proc_k])	Rede ne proc ₁ ...proc _k . Estos nuevos procedimientos escriben los argumentos y salidas de cada llamada a los mismos. Devuelve la lista (proc ₁ ...proc _k).
(untrace [proc₁...proc_r])	Anula la anterior para cada proc _i . Devuelve la lista (proc ₁ ...proc _r).
(error cadena [obj₁...obj_k])	Interrumpe al intØrprete y devuelve un mensaje compuesto por cadena y una versi n en forma de cadena de obj ₁ ...obj _k .

(begin [ex ₁ ...ex _k])	Evalúa de manera sucesiva ex ₁ ...ex _k y devuelve el valor de ex _k ; sin argumentos devuelve un valor no específico.
(random k)	Devuelve un número entero aleatorio entre 0 y k-1.
(exit)	Cierra una sesión con el intérprete de Scheme.