

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 éstos 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 analogía 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 del mismo.

NOTACIÓN

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

(procedimiento *expresion*₁ ... *expresion*_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</i> , <i>expresion</i> , <i>ex</i>	de cualquier tipo
<i>expresiones</i>	sucesión de expresiones

Utilizaremos los corchetes para denotar *expresiones* opcionales y los puntos suspensivos para denotar varias ocurrencias.

(procedimiento *obj*₁ ... *obj*_k) Indica que **procedimiento** es de aridad variable

(procedimiento *obj*₁ [*obj*₂]) 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; e.o.c. #f.
(procedure? <i>obj</i>)	Si <i>obj</i> es de tipo “procedimiento” entonces #t; e.o.c. #f.
(number? <i>obj</i>)	Si <i>obj</i> es de tipo “número” entonces #t; e.o.c. #f.
(pair? <i>obj</i>)	Si <i>obj</i> es de tipo “par punteado” entonces #t; e.o.c. #f.
(null? <i>obj</i>)	Si <i>obj</i> es la “lista vacía” entonces #t; e.o.c. #f.
(boolean? <i>obj</i>)	Si <i>obj</i> es uno de los valores de verdad (o booleanos), #t o #f, entonces #t; e.o.c. #f.
(vector? <i>obj</i>)	Si <i>obj</i> es de tipo “vector” entonces #t; e.o.c. #f.
(char? <i>obj</i>)	Si <i>obj</i> es de tipo “carácter” entonces #t; e.o.c. #f.
(string? <i>obj</i>)	Si <i>obj</i> es de tipo “cadena” entonces #t; e.o.c. #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; e.o.c. #f.
---------------------	--

PREDICADOS DE IGUALDAD

(= z_1 [z_2 ... z_k])	Igualdad numérica entre los argumentos.
(eq? <i>obj</i> ₁ <i>obj</i> ₂)	Igualdad simbólica.
(eqv? <i>obj</i> ₁ <i>obj</i> ₂)	Igualdad numérico-simbólica.
(equal? <i>obj</i> ₁ <i>obj</i> ₂)	Igualdad de valores.

VARIABLES Y LITERALES

<i>expresion</i>	Valor de <i>expresion</i> .
(define <i>simbolo 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 si mismos por lo que no necesitan `quote`.

(let ([(<i>simbolo</i> ₁ <i>obj</i> ₁) : (<i>simbolo</i> _{<i>k</i>} <i>obj</i> _{<i>k</i>})] <i>ex</i> ₁ [<i>ex</i> ₂ ... <i>ex</i> _{<i>r</i>}])	(Todos los <i>simbolo</i> _{<i>i</i>} deben ser distintos). Evalúa cada <i>obj</i> _{<i>j</i>} , en un entorno local, asigna a cada <i>simbolo</i> _{<i>s</i>} el valor de <i>obj</i> _{<i>s</i>} y a continuación evalúa de forma consecutiva <i>ex</i> ₁ ... <i>ex</i> _{<i>r</i>} , devolviendo el valor de la última.
--	--

<code>(let*</code> <code>([</code> <i>simbolo</i> ₁ <i>obj</i> ₁) : <code>(</code> <i>simbolo</i> _{<i>k</i>} <i>obj</i> _{<i>k</i>}) <code>]</code>) <code>ex</code> ₁ [<code>ex</code> ₂ ... <code>ex</code> _{<i>r</i>}])	En un entorno local, asigna, de manera secuencial, a cada <i>simbolo</i> _{<i>s</i>} el valor de <i>obj</i> _{<i>s</i>} y a continuación evalúa de forma consecutiva <i>ex</i> ₁ ... <i>ex</i> _{<i>r</i>} , devolviendo el valor de la última.
<code>(letrec</code> <i>variables</i> <i>cuerpo</i>)	Similar a <code>let</code> , pero permite hacer definiciones de procedimientos recursivos.
<code>(set!</code> <i>simbolo</i> <i>expresion</i>)	Asigna el valor de <i>expresion</i> a <i>simbolo</i> , que ya debe tener asignado algún valor. Devuelve un valor no específico.

EXPRESIONES LAMBDA

<code>(lambda</code> <i>argumentos</i> <i>cuerpo</i>)	Devuelve un procedimiento.
<i>argumentos</i> puede ser: <code>(</code> <i>simbolo</i> ₁ ... <i>simbolo</i> _{<i>k</i>})	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.
<code>(</code> <i>simb</i> ₁ ... <i>simb</i> _{<i>k</i>} . <i>variable</i>)	Los primeros <i>k</i> argumentos se guardan en las variables <i>simb</i> ₁ a <i>simb</i> _{<i>k</i>} . El resto de argumentos se guardan en una lista en <i>variable</i> . El procedimiento es, por tanto, de aridad al menos <i>k</i> .
<i>cuerpo</i> : Sucesión de expresiones que describen la función.	

<code>(define</code> (<i>simb</i> <i>simb</i> ₁ ... <i>simb</i> _{<i>k</i>}) <i>cuerpo</i>)	Equivalente a <code>(define</code> <i>simb</i> <code>(lambda</code> (<i>simb</i> ₁ ... <i>simb</i> _{<i>k</i>}) <i>cuerpo</i>) <code>)</code>
<code>(define</code> (<i>simbolo</i> . <i>variable</i>) <i>cuerpo</i>)	Equivalente a <code>(define</code> <i>simbolo</i> <code>(lambda</code> <i>variable</i> <i>cuerpo</i>) <code>)</code>
<code>(define</code> (<i>simb</i> <i>s</i> ₁ ... <i>s</i> _{<i>k</i>} . <i>var</i>) <i>cuerpo</i>)	Equivalente a <code>(define</code> <i>simb</i> <code>(lambda</code> (<i>s</i> ₁ ... <i>s</i> _{<i>k</i>} . <i>var</i>) <i>cuerpo</i>) <code>)</code>

ABSTRACCIÓN DE PROCEDIMIENTOS

<code>(apply</code> <i>proc</i> <i>ex</i> ₁ [<i>ex</i> ₂ ... <i>ex</i> _{<i>k</i>}])	(<i>ex</i> _{<i>k</i>} una lista), aplica <i>proc</i> con argumentos <i>ex</i> ₁ ... <i>ex</i> _{<i>k-1</i>} y los elementos de <i>ex</i> _{<i>k</i>} .
--	---

<code>(map proc l₁ [l₂ ... l_k])</code>	Aplica <i>proc</i> a cada elemento de l_1 ; si existe más de una lista, todas han de tener la misma longitud y aplica <i>proc</i> tomando como argumentos un elemento de cada lista, devuelve la lista de los resultados.
<code>(for-each proc l₁ [l₂ ... l_k])</code>	Aplica <i>proc</i> a cada elemento de l_1 ; si existe más de una lista, todas han de tener la misma longitud y aplica <i>proc</i> tomando como argumentos un elemento de cada lista, devuelve un valor no específico.

PROCEDIMIENTOS NUMÉRICOS

<code>(+ [z₁ ... z_k])</code>	Suma de los argumentos; sin argumentos, 0.
<code>(- z₁ [z₂ ... z_k])</code>	Resta de los argumentos, asociando por la izquierda; con un solo argumento, $-z_1$.
<code>(* [z₁ ... z_k])</code>	Producto de los argumentos; sin argumentos, 1.
<code>(/ z₁ [z₂ ... z_k])</code>	División de los argumentos asociando por la izquierda; con un solo argumento, $1/z$.
<code>(sqrt z)</code>	Raíz cuadrada principal de z (si z es real, la raíz cuadrada positiva).
<code>(abs x)</code>	Valor absoluto de x .
<code>(sin z)</code>	Seno de z .
<code>(cos z)</code>	Coseno de z .
<code>(tan z)</code>	Tangente de z .
<code>(asin z)</code>	Arco seno de z .
<code>(acos z)</code>	Arco coseno de z .
<code>(atan z)</code>	Arco tangente de z .
<code>(max x₁ [x₂ ... x_k])</code>	Máximo entre los argumentos.
<code>(min x₁ [x₂ ... x_k])</code>	Mínimo entre los argumentos.
<code>(quotient n₁ n₂)</code>	(n_2 distinto de cero), cociente de n_1 entre n_2 .
<code>(remainder n₁ n₂)</code>	(n_2 distinto de cero), resto de n_1 entre n_2 .
<code>(expt z₁ z₂)</code>	La potencia $z_1^{z_2}$ (con $0^0 = 1$).
<code>(exp z)</code>	La potencia e^z .
<code>(log z)</code>	Logaritmo en base e de z .
<code>(gcd [n₁ ... n_k])</code>	Máximo común divisor entre los argumentos; sin argumentos, 0.

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

PREDICADOS NUMÉRICOS

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

RELACIONES NUMÉRICAS

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

PARES

<code>(cons obj₁ obj₂)</code>	El par cuyo <i>car</i> es <i>obj₁</i> y cuyo <i>cdr</i> es <i>obj₂</i> .
<code>(car par)</code>	Primer elemento de <i>par</i> .
<code>(cdr par)</code>	Segundo elemento de <i>par</i> .

PROCEDIMIENTOS SOBRE LISTAS

<code>(cons obj lista)</code>	Lista que resulta al incluir <i>obj</i> como primer elemento de <i>lista</i> .
<code>(list [obj₁ ... obj_k])</code>	La lista de los argumentos; sin argumentos la lista vacía.
<code>(car lista)</code>	Primer elemento de <i>lista</i> .
<code>(cdr lista)</code>	Lista que resulta al quitarle el primer elemento a <i>lista</i> .
<code>(caar lista)</code> <code>(cadr lista)</code> : <code>(cdddar lista)</code> <code>(cddddr lista)</code>	Composiciones de <i>car</i> y <i>cdr</i> .
<code>(append [lista₁ ... lista_k])</code>	Lista que resulta al unir los argumentos; sin argumentos, la lista vacía.
<code>(reverse lista)</code>	Una lista con los mismos elementos que <i>lista</i> , pero dispuestos en orden inverso.
<code>(length lista)</code>	Longitud de <i>lista</i> .
<code>(list-ref lista k)</code>	Elemento de <i>lista</i> que ocupa la <i>k</i> -ésima posición.
<code>(list-tail lista k)</code>	Sublista de <i>lista</i> obtenida eliminando los <i>k</i> primeros elementos.
<code>(set-car! lista obj)</code>	Almacena <i>obj</i> como el <i>car</i> de <i>lista</i> y devuelve un valor no específico.
<code>(set-cdr! lista obj)</code>	Hace que el <i>cdr</i> de <i>lista</i> apunte a <i>obj</i> y devuelve un valor no específico.

PREDICADOS DE PERTENENCIA

Teniendo en cuenta que una *sublista* de una *lista* se obtiene por aplicaciones sucesivas de *cdr*.

<code>(memq obj lista)</code>	Primera <i>sublista</i> de <i>lista</i> cuyo primer elemento es igual que <i>obj</i> , comparando con <i>eq?</i> ; e.o.c. #f.
-------------------------------	---

<code>(memv obj lista)</code>	Primera <i>sublista</i> de <i>lista</i> cuyo primer elemento es igual que <i>obj</i> , comparando con <code>eqv?</code> ; e.o.c. <code>#f</code> .
<code>(member obj lista)</code>	Primera <i>sublista</i> de <i>lista</i> cuyo primer elemento es igual que <i>obj</i> comparando con <code>equal?</code> ; e.o.c. <code>#f</code> .
<code>(assq obj lista-par)</code>	Primer elemento de <i>lista-par</i> (una lista de pares punteados) cuyo primer elemento es igual que <i>obj</i> , comparando con <code>eq?</code> ; e.o.c. <code>#f</code> .
<code>(assv obj lista-par)</code>	Primer elemento de <i>lista-par</i> (una lista de pares punteados) cuyo primer elemento es igual que <i>obj</i> , comparando con <code>eqv?</code> ; e.o.c. <code>#f</code> .
<code>(assoc obj lista-par)</code>	Primer elemento de <i>lista-par</i> (una lista de pares punteados) cuyo primer elemento es igual que <i>obj</i> , comparando con <code>equal?</code> ; e.o.c. <code>#f</code> .

EXPRESIONES CONDICIONALES

<code>(if test consecuencia [alternativa])</code>	Si <i>test</i> tiene como valor <code>#f</code> entonces <i>alternativa</i> (si no existe <i>alternativa</i> entonces un valor no específico), e.o.c. <i>consecuencia</i> .
<code>(cond (test₁ [expresiones₁]) : (test_k [expresiones_k]) [(else [expresiones_{k+1}])])</code>	Evalúa <i>test</i> ₁ ... <i>test</i> _k <u>sucesivamente</u> hasta encontrar el primer <i>test</i> _i que no tenga como valor <code>#f</code> , en cuyo caso evalúa en orden las <i>expresiones</i> _i devolviendo el valor de la última (si no existen devuelve el valor de dicho <i>test</i> _i). Si todo <i>test</i> _i tiene como valor <code>#f</code> y existe la cláusula <code>else</code> , evalúa en orden <i>expresiones</i> _{k+1} devolviendo el valor de la última; si no existen o no existe cláusula <code>else</code> , devuelve un valor no específico.
<code>(case clave ((datos₁) expresiones₁) : ((datos_k) expresiones_k) [(else expresiones_{k+1})])</code>	Evalúa <i>clave</i> y compara el resultado obtenido con cada uno de los <i>datos</i> _i , <u>sucesivamente</u> . Si encuentra alguno que es igual (comparando con <code>eqv?</code>) evalúa en orden <i>expresiones</i> _i devolviendo el valor de la última. Si el valor de <i>clave</i> es distinto a todos los <i>datos</i> _i y existe la cláusula <code>else</code> , evalúa en orden <i>expresiones</i> _{k+1} devolviendo el valor de la última; si no existe cláusula <code>else</code> , devuelve un valor no específico.

OPERADORES LÓGICOS

<code>(not obj)</code>	Si <i>obj</i> tiene como valor <code>#f</code> entonces <code>#t</code> ; e.o.c. <code>#f</code> .
<code>(or [obj₁ ... obj_k])</code>	Evalúa <i>obj</i> ₁ ... <i>obj</i> _k sucesivamente hasta el primero que no tenga como valor <code>#f</code> y devuelve su valor, e.o.c. <code>#f</code> ; sin argumentos, <code>#f</code> .

(and [*obj*₁ ... *obj*_{*k*}]) Evalúa *obj*₁ ... *obj*_{*k*} sucesivamente hasta el primero que tenga como valor #f, e.o.c. el valor de *obj*_{*k*}; sin argumentos, #t.

ITERACIONES

(do
 ((*simbolo*₁ *obj*₁ [*paso*₁])
 :
 (*simbolo*_{*k*} *obj*_{*k*} [*paso*_{*k*}]))
 (*test* [*ex*₁ ... *ex*_{*r*}])
 [*expresion*₁
 :
 *expresion*_{*s*}])

Cada *simbolo*_{*i*} recibe el valor de *obj*_{*i*}. En cada iteración del bucle se evalúa *test*:

- si es #f se evalúan *expresion*₁ ... *expresion*_{*s*} sucesivamente y se actualizan los valores de cada *simbolo*_{*i*} según *paso*_{*i*} (cuando existen). Comienza una nueva iteración.
- si no, se evalúan *ex*₁ ... *ex*_{*r*} sucesivamente devolviendo el valor de la última. Termina el bucle.

PROCEDIMIENTOS SOBRE VECTORES

(make-vector *k* [*obj*]) Construye un vector con *k* elementos iguales a *obj*, si no existe *obj* el contenido del vector es indeterminado.

(vector [*obj*₁ ... *obj*_{*k*}]) Construye un vector con *k* elementos, cada uno de los cuales es *obj*_{*i*}.

(vector-ref *vector* *k*) Elemento de *vector* que ocupa la posición *k*-ésima.

(vector-length *vector*) Número de elementos de *vector*.

(vector->list *vector*) Lista con los elementos de *vector*.

(list->vector *lista*) Vector con los elementos de *lista*.

(vector-set! *vector* *k* *obj*) Almacena *obj* en el *k*-ésimo elemento de *vector*, devuelve un valor no específico.

(vector-fill! *vector* *obj*) Cambia cada elemento de *vector* por *obj*. Devuelve un valor no específico.

PROCEDIMIENTOS SOBRE CARACTERES

(char->integer *caracter*) Código ASCII de *caracter*.

(integer->char *n*) Carácter con código ASCII *n*.

(char<? *cter*₁ [*cter*₂ ... *cter*_{*k*}]) Comparación de los valores numéricos de *cter*_{*i*} con <.

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

PROCEDIMIENTOS SOBRE CADENAS

`(string-length cadena)` Número de caracteres de *cadena*.

<code>(make-string k [character])</code>	Construye una cadena con k caracteres iguales a <i>character</i> , si no existe <i>character</i> el contenido de la cadena es indeterminado.
<code>(string [cter₁ ... cter_k])</code>	Construye una cadena de k caracteres, cada uno de los cuales es <i>cter_i</i> .
<code>(string-ref cadena k)</code>	Carácter de <i>cadena</i> que ocupa la posición k -ésima.
<code>(string-set! cadena k cter)</code>	Almacena <i>cter</i> como elemento k -ésimo de <i>cadena</i> y devuelve un valor no específico.
<code>(substring cadena k₁ k₂)</code>	Trozo de <i>cadena</i> entre las posiciones k_1 y k_2 , la primera incluida.
<code>(string-append [cad₁ ... cad_k])</code>	Concatenación de las cadenas <i>cad_i</i> , sin argumentos la cadena vacía.
<code>(string->list cadena)</code>	Lista con los caracteres de <i>cadena</i> .
<code>(list->string lista)</code>	Cadena con los caracteres de <i>lista</i> (todos sus elementos han de ser caracteres).
<code>(string-copy cadena)</code>	Una nueva cadena, copia exacta de <i>cadena</i> .
<code>(string-fill! cadena cter)</code>	Cambia cada elemento de <i>cadena</i> por el carácter <i>cter</i> . Devuelve un valor no específico.
<code>(string<? cad₁ [cad₂ ... cad_k])</code>	Comparación de <i>cad_i</i> con la extensión lexicográfica de <code>char<</code> .
<code>(string<=? cad₁ [cad₂ ... cad_k])</code>	Comparación de <i>cad_i</i> con la extensión lexicográfica de <code>char<=</code> .
<code>(string>? cad₁ [cad₂ ... cad_k])</code>	Comparación de <i>cad_i</i> con la extensión lexicográfica de <code>char></code> .
<code>(string>=? cad₁ [cad₂ ... cad_k])</code>	Comparación de <i>cad_i</i> con la extensión lexicográfica de <code>char>=</code> .
<code>(string=? cad₁ [cad₂ ... cad_k])</code>	Comparación de <i>cad_i</i> con la extensión lexicográfica de <code>char=</code> .
<code>(string-ci<? cad₁ [cad₂ ... cad_k])</code>	Comparación de <i>cad_i</i> con la extensión lexicográfica de <code>char-ci<</code> .
<code>(string-ci<=? cad₁ [cad₂ ... cad_k])</code>	Comparación de <i>cad_i</i> con la extensión lexicográfica de <code>char-ci<=</code> .
<code>(string-ci>? cad₁ [cad₂ ... cad_k])</code>	Comparación de <i>cad_i</i> con la extensión lexicográfica de <code>char-ci></code> .
<code>(string-ci>=? cad₁ [cad₂ ... cad_k])</code>	Comparación de <i>cad_i</i> con la extensión lexicográfica de <code>char-ci>=</code> .
<code>(string-ci=? cad₁ [cad₂ ... cad_k])</code>	Comparación de <i>cad_i</i> con la extensión lexicográfica de <code>char-ci=</code> .

PROCEDIMIENTOS DE ENTRADA Y SALIDA

(input-port? <i>obj</i>) (output-port? <i>obj</i>)	Devuelve #t si <i>obj</i> 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 <i>fichero</i>)	Devuelve un puerto capaz de proporcionar caracteres del <i>fichero</i> , que debe existir.
(open-output-file <i>fichero</i>)	Devuelve un puerto capaz de escribir caracteres en <i>fichero</i> , que es creado al evaluar la expresión y, por tanto, inicialmente no debe existir.
(call-with-input-file <i>fichero proc</i>) (call-with-output-file <i>fichero proc</i>)	Evalúan el procedimiento <i>proc</i> con un argumento: el puerto obtenido al abrir <i>fichero</i> para entrada o salida, respectivamente.
(with-input-from-file <i>fichero proc</i>) (with-output-to-file <i>fichero proc</i>)	Se crea un puerto de entrada o salida conectado a <i>fichero</i> , convirtiéndose dicho puerto en el puerto por defecto. A continuación, se evalúa <i>proc</i> sin argumentos.
(eof-object? <i>obj</i>)	Devuelve #t si <i>obj</i> es un dato de tipo <i>final de fichero</i> y #f en caso contrario.
(read [<i>puerto</i>])	Devuelve el siguiente objeto de Scheme que se puede obtener del <i>puerto</i> de entrada dado (o el puerto de entrada por defecto, si se omite). Al llegar al final del fichero conectado a <i>puerto</i> , devuelve un objeto de tipo <i>final de fichero</i> .
(read-char [<i>puerto</i>])	Devuelve el siguiente carácter que se puede obtener del <i>puerto</i> de entrada dado (o el puerto de entrada por defecto, si se omite). Al llegar al final del fichero conectado a <i>puerto</i> , devuelve un objeto de tipo <i>final de fichero</i> .
(peek-char [<i>puerto</i>])	Devuelve el siguiente objeto de Scheme que se puede obtener del <i>puerto</i> de entrada dado (o el puerto de entrada por defecto, si se omite), sin actualizar dicho <i>puerto</i> para que apunte al siguiente carácter. Si no hay caracteres disponibles, devuelve un objeto de tipo <i>final de fichero</i> .
(newline [<i>puerto</i>])	Escribe un objeto de tipo <i>final de línea</i> en <i>puerto</i> (o el puerto de entrada por defecto, si éste se omite). Devuelve un valor no específico.

<code>(write obj [puerto])</code>	Escribe una representación escrita de <i>obj</i> en <i>puerto</i> (o el puerto de entrada por defecto, si éste se omite). Las cadenas se escriben delimitadas por comillas y los caracteres con la notación <code>#\</code> . Devuelve un valor no específico.
<code>(display obj [puerto])</code>	Escribe una representación de <i>obj</i> en <i>puerto</i> (o el puerto de entrada por defecto, si éste se omite). Las cadenas se escriben sin estar delimitadas por comillas y los caracteres sin la notación <code>#\</code> . 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.

<code>(write-char cter [puerto])</code>	Escribe el carácter <i>cter</i> (sin usar la notación <code>#\</code>) en <i>puerto</i> (o en el puerto de entrada por defecto, si éste se omite). Devuelve un valor no específico.
---	--

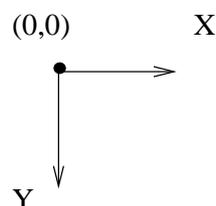
PROCEDIMIENTOS DE LA LIBRERÍA GRÁFICA

Para utilizar la librería gráfica que se describe a continuación, es necesario cargarla mediante la siguiente orden:

```
(require-library "graphics.ss" "graphics").
```

<code>(open-graphics)</code>	Inicializa las rutinas de la librería gráfica. Debe ser utilizada antes que ningún otro procedimiento de la misma.
<code>(close-graphics)</code>	Cierra todas las ventanas. Hasta que no vuelva a utilizarse <code>open-graphics</code> no funcionará ninguna rutina gráfica
<code>(open-viewport cad n1 n2)</code>	Crea una ventana nueva, con nombre <i>cad</i> , de <i>n1</i> pixels de anchura y <i>n2</i> pixels de altura. Devuelve un objeto de tipo <i>ventana</i> .
<code>(close-viewport ventana)</code>	Borra la <i>ventana</i> indicada de la pantalla, impidiendo su uso posterior.

Un objeto de tipo *posicion* 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:



<code>(make-posn <i>n1</i> <i>n2</i>)</code>	Devuelve el objeto de tipo <i>posicion</i> de coordenadas <i>n1</i> y <i>n2</i> .
<code>(posn-x <i>posicion</i>)</code> <code>(posn-y <i>posicion</i>)</code>	Devuelve las coordenadas X e Y, respectivamente, de <i>posicion</i> .
<code>(posn? <i>obj</i>)</code>	Devuelve <code>#t</code> si <i>obj</i> es un objeto de tipo <i>posicion</i> ; en caso contrario devuelve <code>#f</code> .

Un color puede representarse de dos formas distintas: como una cadena (con el nombre del mismo, p.e. “red”) o como un objeto de tipo *rgb*. Cualquiera de los procedimientos que toma *color* como argumento acepta cualquiera de las representaciones.

<code>(make-rgb <i>rojo</i> <i>verde</i> <i>azul</i>)</code>	Dados tres números reales de 0 (oscuro) a 1 (claro) devuelve un objeto de tipo <i>rgb</i> .
<code>(rgb-red <i>rgb</i>)</code> <code>(rgb-green <i>rgb</i>)</code> <code>(rgb-blue <i>rgb</i>)</code>	Devuelve los valores de rojo, verde y azul, respectivamente, de <i>rgb</i> .
<code>(rgb? <i>obj</i>)</code>	Devuelve <code>#t</code> si <i>obj</i> es un objeto de tipo <i>rgb</i> ; en caso contrario devuelve <code>#f</code> .
<code>((draw-viewport <i>ventana</i>) [<i>color</i>])</code> <code>((clear-viewport <i>ventana</i>))</code>	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.
<code>((draw-pixel <i>ventana</i>) <i>posicion</i> [<i>color</i>])</code> <code>((clear-pixel <i>ventana</i>) <i>posicion</i>)</code>	Dada una <i>ventana</i> devuelve un procedimiento que dibuja (resp. borra) un pixel en la misma, en la <i>posicion</i> especificada utilizando <i>color</i> ; o negro, si se omite éste.
<code>((draw-line <i>ventana</i>) <i>posicion1</i> <i>posicion2</i> [<i>color</i>])</code> <code>((clear-line <i>ventana</i>) <i>posicion1</i> <i>posicion2</i>)</code>	Dada una <i>ventana</i> devuelve un procedimiento que dibuja (resp. borra) una línea en la misma, conectando las posiciones especificadas utilizando <i>color</i> ; o negro, si se omite éste.
<code>((draw-rectangle <i>ventana</i>) <i>posicion</i> <i>altura</i> <i>anchura</i> [<i>color</i>])</code> <code>((clear-rectangle <i>ventana</i>) <i>posicion</i> <i>altura</i> <i>anchura</i>)</code>	Dada una <i>ventana</i> 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 <i>posicion</i> especificada, utilizando <i>color</i> ; o negro, si se omite éste.
<code>((draw-solid-rectangle <i>ventana</i>) <i>posicion</i> <i>altura</i> <i>anchura</i> [<i>color</i>])</code> <code>((clear-solid-rectangle <i>ventana</i>) <i>posicion</i> <i>altura</i> <i>anchura</i>)</code>	Como el anterior, pero colorea toda la figura utilizando <i>color</i> ; o negro, si se omite éste.

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

OTRAS FUNCIONES DE INTERÉS

<pre>(load "nombre-fichero")</pre>	<p>Evalúa el contenido de <i>nombre-fichero</i>.</p>
<pre>(trace [proc₁ ... proc_k])</pre>	<p>Redefine <i>proc₁ ... proc_k</i>. Estos nuevos procedimientos escriben los argumentos y salidas de cada llamada a los mismos. Devuelve la lista (<i>proc₁ ... proc_k</i>).</p>
<pre>(untrace [proc₁ ... proc_r])</pre>	<p>Anula la anterior para cada <i>proc_i</i>. Devuelve la lista (<i>proc₁ ... proc_r</i>).</p>
<pre>(error cadena [obj₁ ... obj_k])</pre>	<p>Interrumpe al intérprete y devuelve un mensaje compuesto por <i>cadena</i> y una versión en forma de cadena de <i>obj₁ ... obj_k</i>.</p>
<pre>(begin [ex₁ ... ex_k])</pre>	<p>Evalúa de manera sucesiva <i>ex₁ ... ex_k</i> y devuelve el valor de <i>ex_k</i>; sin argumentos devuelve un valor no específico.</p>

<code>(random k)</code>	Devuelve un número entero aleatorio entre 0 y $k-1$.
<code>(exit)</code>	Cierra una sesión con el intérprete de Scheme.