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` $expresion_1 \ldots expresion_k$`)`

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:

z	"número complejo"	l	"lista"
x	"número real"	*cter*	"carácter"
n	"número entero"	*cad*	"cadena"
k	"número natural"	*proc*	"procedimiento"
obj, expresion, ex	de cualquier tipo		
expresiones	sucesión de expresiones		

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

`(procedimiento` $obj_1 \ldots obj_k$`)`	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?` *obj*`)`	Si *obj* es de tipo "símbolo" entonces `#t`; e.o.c. `#f`.
`(procedure?` *obj*`)`	Si *obj* es de tipo "procedimiento" entonces `#t`; e.o.c. `#f`.
`(number?` *obj*`)`	Si *obj* es de tipo "número" entonces `#t`; e.o.c. `#f`.
`(pair?` *obj*`)`	Si *obj* es de tipo "par punteado" entonces `#t`; e.o.c. `#f`.
`(null?` *obj*`)`	Si *obj* es la "lista vacía" entonces `#t`; e.o.c. `#f`.
`(boolean?` *obj*`)`	Si *obj* es uno de los valores de verdad (o booleanos), `#t` o `#f`, entonces `#t`; e.o.c. `#f`.
`(vector?` *obj*`)`	Si *obj* es de tipo "vector" entonces `#t`; e.o.c. `#f`.
`(char?` *obj*`)`	Si *obj* es de tipo "carácter" entonces `#t`; e.o.c. `#f`.
`(string?` *obj*`)`	Si *obj* 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?` *obj*`)`	Si *obj* es de tipo "lista" entonces `#t`; e.o.c. `#f`.

Predicados de igualdad

`(=` z_1 $[z_2 \ldots z_k]$`)`	Igualdad numérica entre los argumentos.
`(eq?` obj_1 obj_2`)`	Igualdad simbólica.
`(eqv?` obj_1 obj_2`)`	Igualdad numérico-simbólica.
`(equal?` obj_1 obj_2`)`	Igualdad de valores.

Variables y literales

expresion	Valor de *expresion*.
`(define` *simbolo obj*`)`	Le asigna a *simbolo* el valor de *obj*.
`(quote` *obj*`)` $\equiv$ *'obj*	*obj*.

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

`(let`
 `([(`$simbolo_1$ obj_1`)`
 $\vdots$
 `(`$simbolo_k$ obj_k`)])`
 ex_1 $[ex_2 \ldots ex_r]$`)`
	(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 $ex_1 \ldots ex_r$, devolviendo el valor de la última.

`(let*`
`([(`*simbolo*$_1$ *obj*$_1$`)`
⋮
`(`*simbolo*$_k$ *obj*$_k$`)])`
ex$_1$ [*ex*$_2$ ... *ex*$_r$]`)`
	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.

`(letrec` *variables cuerpo*`)`	Similar a `let`, pero permite hacer definiciones de procedimientos recursivos.
`(set!` *simbolo expresion*`)`	Asigna el valor de *expresion* a *simbolo*, que ya debe tener asignado algún valor. Devuelve un valor no específico.

Expresiones lambda

`(lambda` *argumentos cuerpo*`)`	Devuelve un procedimiento.

argumentos puede ser:

`(`*simbolo*$_1$ ... *simbolo*$_k$`)`	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.
variable	Símbolo que representa a una lista con los argumentos. Por tanto será de aridad variable.
`(`*simb*$_1$ ... *simb*$_k$ `.` *variable*`)`	Los primeros k argumentos se guardan en las variables *simb*$_1$ a *simb*$_k$. El resto de argumentos se guardan en una lista en *variable*. El procedimiento es, por tanto, de aridad al menos k.

cuerpo: Sucesión de expresiones que describen la función.

`(define (`*simb simb*$_1$ ... *simb*$_k$`)` *cuerpo*`)`	Equivalente a `(define` *simb* `(lambda (`*simb*$_1$ ... *simb*$_k$`)` *cuerpo*`))`
`(define (`*simbolo . variable*`)` *cuerpo*`)`	Equivalente a `(define` *simbolo* `(lambda` *variable* *cuerpo*`))`
`(define (`*simb* s_1 ... s_k `.` *var*`)` *cuerpo*`)`	Equivalente a `(define` *simb* `(lambda (`s_1 ... s_k `.` *var*`)` *cuerpo*`))`

Abstracción de procedimientos

`(apply` *proc ex*$_1$ [*ex*$_2$ ... *ex*$_k$]`)`	(*ex*$_k$ una lista), aplica *proc* con argumentos *ex*$_1$ ... *ex*$_{k-1}$ y los elementos de *ex*$_k$.

`(map` *proc* l_1 $[l_2 \ldots l_k]$`)`	Aplica *proc* a cada elemento de l_1; 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* l_1 $[l_2 \ldots l_k]$`)`	Aplica *proc* a cada elemento de l_1; 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

`(+` $[z_1 \ldots z_k]$`)`	Suma de los argumentos; sin argumentos, 0.
`(-` z_1 $[z_2 \ldots z_k]$`)`	Resta de los argumentos, asociando por la izquierda; con un solo argumento, $-z_1$.
`(*` $[z_1 \ldots z_k]$`)`	Producto de los argumentos; sin argumentos, 1.
`(/` z_1 $[z_2 \ldots z_k]$`)`	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` x_1 $[x_2 \ldots x_k]$`)`	Máximo entre los argumentos.
`(min` x_1 $[x_2 \ldots x_k]$`)`	Mínimo entre los argumentos.
`(quotient` n_1 n_2`)`	(n_2 distinto de cero), cociente de n_1 entre n_2.
`(remainder` n_1 n_2`)`	(n_2 distinto de cero), resto de n_1 entre n_2.
`(expt` z_1 z_2`)`	La potencia ${z_1}^{z_2}$ (con $0^0 = 1$).
`(exp` z`)`	La potencia $\mathbf{e}^z$.
`(log` z`)`	Logaritmo en base **e** de z.
`(gcd` $[n_1 \ldots n_k]$`)`	Máximo común divisor entre los argumentos; sin argumentos, 0.

`(lcm` $[n_1 \ldots n_k]$`)`	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 *obj* es un número complejo entonces `#t`; e.o.c. `#f`.
`(real?` *obj*`)`	Si *obj* es un número real entonces `#t`; e.o.c. `#f`.
`(rational?` *obj*`)`	Si *obj* es un número racional entonces `#t`; e.o.c. `#f`.
`(exact?` z`)`	Si z es exacto, entonces `#t`; e.o.c `#f`.
`(inexact?` z`)`	Si z es inexacto, entonces `#t`; e.o.c `#f`.
`(integer?` *obj*`)`	Si *obj* es un número entero entonces `#t`; e.o.c. `#f`.
`(even?` n`)`	Si n es par entonces `#t`; e.o.c. `#f`.
`(odd?` n`)`	Si n es impar entonces `#t`; e.o.c. `#f`.
`(zero?` z`)`	Si z es el cero entonces `#t`; e.o.c. `#f`.
`(positive?` x`)`	Si x es mayor estricto que cero entonces `#t`; e.o.c. `#f`.
`(negative?` x`)`	Si x es menor estricto que cero entonces `#t`; e.o.c. `#f`.

Relaciones numéricas

`(>` x_1 $[x_2 \ldots x_k]$`)`	Los argumentos están en orden decreciente.
`(<` x_1 $[x_2 \ldots x_k]$`)`	Los argumentos están en orden creciente.
`(>=` x_1 $[x_2 \ldots x_k]$`)`	Los argumentos están en orden no creciente.
`(<=` x_1 $[x_2 \ldots x_k]$`)`	Los argumentos están en orden no decreciente.

Pares

`(cons` *obj*$_1$ *obj*$_2$`)`	El par cuyo *car* es *obj*$_1$ y cuyo *cdr* es *obj*$_2$.
`(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` [*obj*$_1$ ... *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*`)`
⋮
`(cdddar` *lista*`)`
`(cddddr` *lista*`)`
	Composiciones de `car` y `cdr`.

`(append` [*lista*$_1$ ... *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-ésima posición.
`(list-tail` *lista k*`)`	Sublista 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 *sublista* de una *lista* se obtiene por aplicaciones sucesivas de `cdr`.

`(memq` *obj lista*`)`	Primera *sublista* de *lista* cuyo primer elemento es igual que *obj*, comparando con `eq?`; e.o.c. `#f`.

`(memv` *obj lista*`)`	Primera *sublista* de *lista* cuyo primer elemento es igual que *obj*, comparando con `eqv?`; e.o.c. `#f`.
`(member` *obj lista*`)`	Primera *sublista* de *lista* cuyo primer elemento es igual que *obj* comparando con `equal?`; e.o.c. `#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?`; e.o.c. `#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?`; e.o.c. `#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?`; e.o.c. `#f`.

Expresiones condicionales

`(if` *test* *consecuencia* [*alternativa*]`)`	Si *test* tiene como valor `#f` entonces *alternativa* (si no existe *alternativa* entonces un valor no específico), e.o.c. *consecuencia*.

`(cond`
`(`$test_1$ [$expresiones_1$]`)`
⋮
`(`$test_k$ [$expresiones_k$]`)`
[`(else` [$expresiones_{k+1}$]`)`]`)`
	Evalúa $test_1 \ldots test_k$ 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.

`(case` *clave*
`((`$datos_1$`)` $expresiones_1$`)`
⋮
`((`$datos_k$`)` $expresiones_k$`)`
[`(else` $expresiones_{k+1}$`)`]`)`
	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` *obj*`)`	Si *obj* tiene como valor `#f` entonces `#t`; e.o.c. `#f`.
`(or` [$obj_1 \ldots obj_k$]`)`	Evalúa $obj_1 \ldots obj_k$ sucesivamente hasta el primero que no tenga como valor `#f` y devuelve su valor, e.o.c. `#f`; sin argumentos, `#f`.

`(and` $[obj_1 \ldots obj_k]$`)` Evalúa $obj_1 \ldots obj_k$ 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 $[paso_k]$`))`
`(`$test$ $[ex_1 \ldots ex_r]$`)`
$[expresion_1$
⋮
$expresion_s]$`)`

Cada $simbolo_i$ recibe el valor de obj_i. En cada iteración del bucle se evalúa *test*:

- si es `#f` se evalúan $expresion_1 \ldots expresion_s$ sucesivamente y se actualizan los valores de cada $simbolo_i$ según $paso_i$ (cuando existen). Comienza una nueva iteración.
- si no, se evalúan $ex_1 \ldots ex_r$ 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` $[obj_1 \ldots obj_k]$`)` 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<?` $cter_1$ $[cter_2 \ldots cter_k]$`)` Comparación de los valores numéricos de $cter_i$ con $<$.

`(char<=?` $cter_1$ $[cter_2 \ldots cter_k]$`)`	Comparación de los valores numéricos de $cter_i$ con $<=$.
`(char>?` $cter_1$ $[cter_2 \ldots cter_k]$`)`	Comparación de los valores numéricos de $cter_i$ con $>$.
`(char>=?` $cter_1$ $[cter_2 \ldots cter_k]$`)`	Comparación de los valores numéricos de $cter_i$ con $>=$.
`(char=?` $cter_1$ $[cter_2 \ldots cter_k]$`)`	Comparación de los valores numéricos de $cter_i$ con $=$.
`(char-ci<?` $cter_1$ $[cter_2 \ldots cter_k]$`)`	Comparación de los valores numéricos de $cter_i$ con $<$, sin distinguir mayúsculas de minúsculas.
`(char-ci<=?` $cter_1$ $[cter_2 \ldots cter_k]$`)`	Comparación de los valores numéricos de $cter_i$ con $<=$, sin distinguir mayúsculas de minúsculas.
`(char-ci>?` $cter_1$ $[cter_2 \ldots cter_k]$`)`	Comparación de los valores numéricos de $cter_i$ con $>$, sin distinguir mayúsculas de minúsculas.
`(char-ci>=?` $cter_1$ $[cter_2 \ldots cter_k]$`)`	Comparación de los valores numéricos de $cter_i$ con $>=$, sin distinguir mayúsculas de minúsculas.
`(char-ci=?` $cter_1$ $[cter_2 \ldots cter_k]$`)`	Comparación de los valores numéricos de $cter_i$ con $=$, sin distinguir mayúsculas de minúsculas.
`(char-upcase` *caracter*`)`	Si *caracter* es minúscula devuelve el mismo en mayúscula; devuelve *caracter* e.o.c.
`(char-downcase` *caracter*`)`	Si *caracter* es mayúscula devuelve el mismo en minúscula; devuelve *caracter* e.o.c.
`(char-upper-case?` *obj*`)`	Si *obj* es de tipo "carácter mayúscula", entonces `#t`; e.o.c. `#f`.
`(char-lower-case?` *obj*`)`	Si *obj* es de tipo "carácter minúscula", entonces `#t`; e.o.c. `#f`.
`(char-alphabetic?` *obj*`)`	Si *obj* es de tipo "carácter alfabético", entonces `#t`; e.o.c. `#f`.
`(char-numeric?` *obj*`)`	Si *obj* es de tipo "carácter numérico", entonces `#t`; e.o.c. `#f`.
`(char-whitespace?` *obj*`)`	Si *obj* es de tipo "carácter de espacio en blanco" (espacio, tabulación, salto de línea, retorno de carro), entonces `#t`; e.o.c. `#f`.

Procedimientos sobre cadenas

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

`(make-string` k [*caracter*]`)`	Construye una cadena con k caracteres iguales a *caracter*, si no existe *caracter* el contenido de la cadena es indeterminado.
`(string` [$cter_1 \ldots cter_k$]`)`	Construye una cadena de k caracteres, cada uno de los cuales es $cter_i$.
`(string-ref` *cadena* k`)`	Carácter de *cadena* que ocupa la posición k-ésima.
`(string-set!` *cadena* k *cter*`)`	Almacena *cter* como elemento k-ésimo de *cadena* y devuelve un valor no específico.
`(substring` *cadena* k_1 k_2`)`	Trozo de *cadena* entre las posiciones k_1 y k_2, la primera incluida.
`(string-append` [$cad_1 \ldots cad_k$]`)`	Concatenación de las cadenas cad_i, sin argumentos la cadena vacía.
`(string->list` *cadena*`)`	Lista con los caracteres de *cadena*.
`(list->string` *lista*`)`	Cadena con los caracteres de *lista* (todos sus elementos han de ser caracteres).
`(string-copy` *cadena*`)`	Una nueva cadena, copia exacta de *cadena*.
`(string-fill!` *cadena cter*`)`	Cambia cada elemento de *cadena* por el carácter *cter*. Devuelve un valor no específico.
`(string<?` cad_1 [$cad_2 \ldots cad_k$]`)`	Comparación de cad_i con la extensión lexicográfica de `char<`.
`(string<=?` cad_1 [$cad_2 \ldots cad_k$]`)`	Comparación de cad_i con la extensión lexicográfica de `char<=`.
`(string>?` cad_1 [$cad_2 \ldots cad_k$]`)`	Comparación de cad_i con la extensión lexicográfica de `char>`.
`(string>=?` cad_1 [$cad_2 \ldots cad_k$]`)`	Comparación de cad_i con la extensión lexicográfica de `char>=`.
`(string=?` cad_1 [$cad_2 \ldots cad_k$]`)`	Comparación de cad_i con la extensión lexicográfica de `char=`.
`(string-ci<?` cad_1 [$cad_2 \ldots cad_k$]`)`	Comparación de cad_i con la extensión lexicográfica de `char-ci<`.
`(string-ci<=?` cad_1 [$cad_2 \ldots cad_k$]`)`	Comparación de cad_i con la extensión lexicográfica de `char-ci<=`.
`(string-ci>?` cad_1 [$cad_2 \ldots cad_k$]`)`	Comparación de cad_i con la extensión lexicográfica de `char-ci>`.
`(string-ci>=?` cad_1 [$cad_2 \ldots cad_k$]`)`	Comparación de cad_i con la extensión lexicográfica de `char-ci>=`.
`(string-ci=?` cad_1 [$cad_2 \ldots cad_k$]`)`	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 *fichero*, 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 fichero* 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 fichero 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 éste 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 éste 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 éste 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 éste se omite). Devuelve un valor no específico.

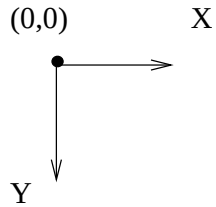
Procedimientos de la librería gráfica

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

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

`(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 funcionará ninguna rutina gráfica
`(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 *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:



`(make-posn` *n1 n2*`)` — Devuelve el objeto de tipo *posicion* de coordenadas *n1* y *n2*.

`(posn-x` *posicion*`)`
`(posn-y` *posicion*`)` — Devuelve las coordenadas X e Y, respectivamente, de *posicion*.

`(posn?` *obj*`)` — Devuelve `#t` si *obj* es un objeto de tipo *posicion*; en caso contrario devuelve `#f`.

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.

`(make-rgb` *rojo verde azul*`)` — Dados tres números reales de 0 (oscuro) a 1 (claro) devuelve un objeto de tipo *rgb*.

`(rgb-red` *rgb*`)`
`(rgb-green` *rgb*`)`
`(rgb-blue` *rgb*`)` — Devuelve los valores de rojo, verde y azul, respectivamente, de *rgb*.

`(rgb?` *obj*`)` — Devuelve `#t` si *obj* es un objeto de tipo *rgb*; en caso contrario devuelve `#f`.

`((draw-viewport` *ventana*`)` [*color*]`)`
`((clear-viewport` *ventana*`))` — Dada una *ventana* devuelve un procedimiento que colorea (resp. borra) el contenido completo de la misma utilizando *color* o negro, si se omite éste.

`((draw-pixel` *ventana*`)`
posicion [*color*]`)`
`((clear-pixel` *ventana*`)`
posicion`)` — Dada una *ventana* devuelve un procedimiento que dibuja (resp. borra) un pixel en la misma, en la *posicion* especificada utilizando *color*; o negro, si se omite éste.

`((draw-line` *ventana*`)`
posicion1 posicion2 [*color*]`)`
`((clear-line` *ventana*`)`
posicion1 posicion2`)` — Dada una *ventana* devuelve un procedimiento que dibuja (resp. borra) una línea en la misma, conectando las posiciones especificadas utilizando *color*; o negro, si se omite éste.

`((draw-rectangle` *ventana*`)`
posicion altura anchura [*color*]`)`
`((clear-rectangle` *ventana*`)`
posicion altura anchura`)` — Dada una ventana devuelve un procedimiento que dibuja (resp. borra) el borde de un rectángulo en la misma, de *altura* y *anchura* dadas, siendo el vértice superior izquierdo el que ocupa la *posicion* especificada, utilizando *color*; o negro, si se omite éste.

`((draw-solid-rectangle` *ventana*`)`
posicion altura anchura [*color*]`)`
`((clear-solid-rectangle` *ventana*`)`
posicion altura anchura`)` — Como el anterior, pero colorea toda la figura utilizando *color*; o negro, si se omite éste.

`((draw-ellipse` *ventana*`)` *posicion altura anchura* [*color*]`)` `((clear-ellipse` *ventana*`)` *posicion altura anchura*`)`	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 *posicion* especificada, utilizando *color*; o negro, si se omite éste.
`((draw-solid-ellipse` *ventana*`)` *posicion altura anchura* [*color*]`)` `((clear-solid-ellipse` *ventana*`)` *posicion altura anchura*`)`	Como el anterior, pero colorea toda la figura utilizando *color*; o negro, si se omite éste.
`((draw-polygon` *ventana*`)` *l-posiciones posicion* [*color*]`)` `((clear-polygon` *ventana*`)` *l-posiciones posicion*`)`	Dada una ventana devuelve un procedimiento que dibuja (resp. borra) el borde de un polígono en la misma, siendo los vértices los elementos de *l-posiciones* y considerando *posicion* como el desplazamiento del mismo. Utilizando *color*; o negro, si se omite éste.
`((draw-solid-polygon` *ventana*`)` *l-posiciones posicion* [*color*]`)` `((clear-solid-polygon` *ventana*`)` *l-posiciones posicion*`)`	Como el anterior, pero colorea toda la figura utilizando *color*; o negro, si se omite éste.
`((draw-string` *ventana*`)` *posicion cadena* [*color*]`)` `((clear-string` *ventana*`)` *posicion cadena*`)`	Dada una *ventana* devuelve un procedimiento que dibuja (resp. borra) una *cadena* en la misma, siendo el vértice inferior izquierdo el que ocupa la *posicion* especificada, utilizando *color*; o negro, si se omite éste.
`((get-string-size` *ventana*`)` *cadena*`)`	Dada una *ventana* devuelve un procedimiento 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` [$proc_1 \ldots proc_k$]`)`	Redefine $proc_1 \ldots proc_k$. Estos nuevos procedimientos escriben los argumentos y salidas de cada llamada a los mismos. Devuelve la lista $(proc_1 \ldots proc_k)$.
`(untrace` [$proc_1 \ldots proc_r$]`)`	Anula la anterior para cada $proc_i$. Devuelve la lista $(proc_1 \ldots procr)$.
`(error` *cadena* [$obj_1 \ldots obj_k$]`)`	Interrumpe al intérprete y devuelve un mensaje compuesto por *cadena* y una versión en forma de cadena de $obj_1 \ldots obj_k$.
`(begin` [$ex_1 \ldots ex_k$]`)`	Evalúa de manera sucesiva $ex_1 \ldots ex_k$ 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.