LISP O EL DOMINIO DE LA IA

Autor: *R. Garrote*

```
(DE EQUAL (SEX1 SEX2)
(COND ((ATOM SEX1) (EQ SEX1 SEX2))
      ((ATOM SEX2) NIL)
      ((EQUAL (CAR SEX1) (CAR SEX2))
      (EQUAL (CDR SEX1) (CDR SEX2)))
      (T NIL)
      ) )
```

Cuando vi por primera vez un programa LISP quedé sorprendido. ¿Cómo era posible que eso, que no tenía asignaciones, ni bucles, ni saltos, pudiese hacer algo?

Para cargar tu intérprete de LISP teclea RUN "MINILISP". Este programa es el cargador; te muestra la insignia del programa y te pide el tamaño de la tabla de identificadores, que es el diccionario interno del intérprete. Debes dar un número entre 100 y 1.000. Pongamos, de momento, 300. (El programa está diseñado de modo que te deje la mayor cantidad posible de memoria libre —esto lo consigue con el comando CHAIN MERGE y su opción DELETE— de modo que el programa se carga en cuatro etapas. Entre estas etapas hace dos pausas pronunciadas para inicializar las estructuras internas). **Cuando el intérprete de MINILISP esté dispuesto escucharás un pitido y aparecerá en pantalla un mensaje indicándote la cantidad de memoria disponible.** Justo debajo aparece el «saludo» de tu intérprete:

```
>
```

Uno se puede entonces imaginar que dentro de la máquina se halla un genielcillo que está dispuesto a satisfacer todas tus órdenes siempre que se las des en su idioma, que es LISP. (El genielcillo de nuestro computador es un poco perezoso para cumplir los recados que le pedimos).

Vamos a pedirle al genio algo que tanto tú como él entenderéis:

```
> (+ 2 2) (¡y ENTER claro!)
```

En efecto, queremos que el genio nos diga cuántas son dos más dos. Cuando el genio comprende nuestra solicitud indica con puntos suspensivos que está pensando (evaluando, se dice). Cuando sabe la respuesta nos enseña su reloj para que veamos cuánto ha tardado en calcularla y nos dice que dos más dos son cuatro. ¡Magnífico!

El intérprete de LISP «lee» nuestro deseo, lo evalúa, imprime el resultado y nos indica a continuación que está dispuesto a cumplir un nuevo mandato.

¿Cómo podemos comunicarnos con el duende de LISP y de qué forma nos responderá? En principio se le pueden pedir cosas sencillas, pero escritas en una forma un tanto extraña (a los que hayáis programado algunas calculadoras tal vez os suene: se llama notación polaca). Por ejemplo. Para que calcule (2*3)+15—(6/2) se podría escribir:

```
> (+ (* 2 3)
> (SUB 15
> (DIV 6 2))
```

¡No parece LISP un lenguaje especialmente pensado para realizar operaciones aritméticas! Además, MINILISP opera sólo con números enteros. LISP es un lenguaje especializado en manipular listas de objetos (el nombre de LISP viene de LISt Processing, procesamiento de listas). Los elementos básicos que maneja LISP son los átomos.

Pero, ¿qué es un átomo? En principio, un átomo es cualquier sucesión de letras y números que empiece por una letra. Por ejemplo, son átomos

```
ATOMO átomo Casa A123 Lista LIstA NIL T
```

y no son átomos

```
123a (EstoNoEsUnAtomo)
```

Tu intérprete de LISP no distingue entre mayúsculas y minúsculas, de modo que ATOMO, Atomo y AtoMO son uno y el mismo objeto.

Sin embargo, también son átomos

```
ANDRES PEREZ ES UNA ? *ESTRELLAS*
```

Todos estos átomos se llaman átomos literales. Podemos dar entonces una definición más general de los átomos literales: un átomo literal es cualquier sucesión de símbolos con dos únicas restricciones:

i) Un átomo literal no puede empezar ni por un dígito ni por el símbolo '.

ii) Un átomo literal no puede contener ninguno de los siguientes símbolos: ( ) . % ' .

Las razones de que estos símbolos no se puedan utilizar las iremos explicando poco a poco. Ya hemos visto que LISP también maneja números. Los números también son átomos aunque, no son átomos literales. (A veces se llaman átomos numerales o numéricos).

En LISP, todas las instrucciones son llamadas a funciones (véase el artículo sobre Inteligencia Artificial en el número xxx). Una función básica (primitiva) en LISP es QUOTE. Esta función tiene como valor su argumento, es decir

```
> (QUOTE ANTONIO)
```

vale ANTONIO. QUOTE tiene como misión impedir que se evalúe su argumento. Por ejemplo, cuando tú escribes en Basic.

```
PRINT 2+2
```

el intérprete de Basic evalúa la expresión 2+2 y escribe su valor: 4. Si quieres que no se evalúe, debes escribir

```
PRINT "2+2"
```

De la misma forma

```
PRINT ANTONIO
```

escribiría el valor de la variable ANTONIO, mientras que

```
PRINT "ANTONIO"
```

escribiría ANTONIO. Para impedir que se interprete ANTONIO como una variable debe entrecomillarlo; ese efecto de **«entrecomillado»** es el que se consigue en LISP con la función QUOTE.

En LISP, los átomos literales no tienen valor excepto los átomos NIL y T cuyos valores son respectivamente NIL y T. Este par de valores se usa con frecuencia como valores booleanos: NIL equivale a falso y T a cierto (True, en inglés). En MINILISP, existen otros dos átomos literales con valor: PIZQ que vale (, cuyo valor es). Sirve para que puedas escribir ambos paréntesis. Los átomos numerales sí tienen valor y su valor es el número que representan. Si se evalúa un átomo que no tiene valor se produce un error. (Algunos átomos literales pueden actuar como variables y guardar temporalmente valores, pero de las variables hablaremos luego).

Como hay que utilizar mucho la función QUOTE se ha ideado una abreviatura para ella (igual que puede utilizar ? en vez de PRINT cuando programas en Basic). Esta abreviatura es '. Por tanto

```
'ANTONIO
```

también vale ANTONIO. En adelante, en lugar de QUOTE utilizaremos su abreviatura.

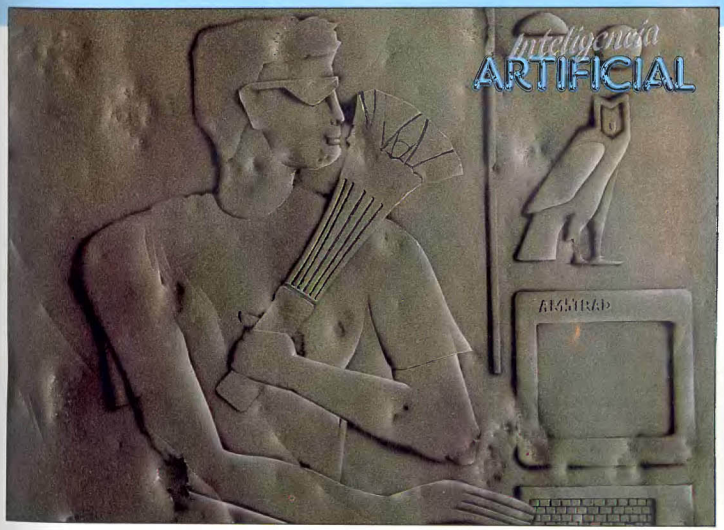
Otra función primitiva de LISP es CONS, que produce pares de objetos. Por ejemplo:

```
> (CONS 'A 'B)
```

vale (A.B). Nos encontramos entonces ante un objeto que no es un átomo. Se llama par, el par que tiene como primer elemento A y como segundo elemento B. ¿Por qué hemos utilizado QUOTE? Pues porque CONS también evalúa sus argumentos y queríamos obtener el par formado por A y B y no por sus valores. Un ejemplo más:

```
> (CONS 'A 1)
```

vale (A. 1). El átomo 1 tiene como valor 1 y por eso no hay que ponerle QUOTE delante.



Ahora hagamos la operación contraria: dado un par, obtener sus componentes. Hay dos funciones que hacen esto: CAR devuelve el primer elemento del par y CDR su segundo elemento:

```
> (CAR ' (A.B))
```

```
> (CDR ' (A.B))
```

vale B. Además de construir y «destruir» objetos podemos hacer preguntar al genio sobre estos objetos. Por ejemplo, el predicado EQ se utiliza para comprobar la igualdad de dos átomos literales mientras que para comprobar la igualdad de números debe usarse =

```
> (= 4 8)
```

vale NIL, es decir, es falso, mientras que

```
> (EQ ' A (CAR ' (A.B)))
```

vale T. Date cuenta que el segundo parámetro de EQ, (CAR ' (A.B)), no lleva QUOTE. En consecuencia, el genio evalúa esto para dar A, según vimos en un ejemplo anterior. El resultado de evaluar 'A también es A y por tanto se verifica la igualdad.

Vamos a construir ahora estructuras más complicadas.

```
> CONS 'ANA (CONS 'ES 'BONITA))
```

vale (ANA. (ES.BONITA)). CONS sólo admite dos parámetros, de modo que para obtener estructuras de más de dos elementos debe formarse pares de dos en dos. Para reflejar esta forma de construir estructuras es mejor escribir

```
> (CONS 'ANA
>       'ES
>       'BONITA)))
```

Si ahora le preguntamos al genio

```
+ (CONS 'A NIL)
```

nos responderá con (A). *¿Qué ha ocurrido?* ¿No debería ser la respuesta (A. NIL)? Antes dije que LISP es un lenguaje especializado en manipular listas y, sin embargo, hasta ahora sólo conocíamos átomos y pares. Tanto átomos, como pares y listas se conocen con el nombre genérico de expresiones simbólicas (S-expresiones o SEXos). Ocurre que el átomo NIL es un poco «raro». Pero antes de seguir daré la definición de lista:

i) () es una lista con 0 elementos (se llama lista vacía).

ii) el par que tiene como primer elemento una S-expresión S y como segundo elemento una lista L es, a su vez, una lista que tiene como primer elemento la S-expresión S y como resto de la lista la lista L.

Para los que no estéis acostumbrados a las definiciones recursivas —las que usan el objeto que se pretende definir en la propia definición— daré otra definición: una lista es o una lista que no tiene elementos, representada por (), o cualquier cosa que contenga átomos, pares u otras listas entre un paréntesis abierto "(" y un paréntesis cerrado ")". Por ejemplo:

```
(ESTO ES UNA LISTA DE 7 ELEMENTOS)
```

```
(ESTO ES (UNA (LISTA DE 3) ELEMENTOS))
(ESTO (TAMBIEN.ES) ((UNA)) LISTA)
```

El tercer ejemplo es una lista de tres elementos, el tercero de los cuales es a su vez una lista de tres elementos.

```
(UNA (LISTA DE 3) ELEMENTOS)
```

y cuyo segundo elemento es otra lista de tres elementos.

```
(LISTA DE 3)
```

Veamos algunos ejemplos de cosas que no son listas·

```
LE FALTAN LOS PARENTESIS
((LE FALTA) EL PARENTESIS DERECHO
```

Todos los objetos que manipula LISP o son átomos, o son pares, o son listas. Sin embargo, existe en LISP un objeto un tanto esquizofrénico: es un átomo que también es una lista o una lista que también es un átomo. Tiene dos representaciones: como átomo, NIL, y como lista () (por eso a veces se dice que es BISEXual). LISP no distingue entre una y otra representación: para LISP son una y la misma cosa. En adelante, usaré la representación que me parezca más conveniente, sin hacer referencia a la otra.

Por la definición de lista, como A es un átomo y NIL es una lista, entonces (CONS 'A NIL) también es una lista, que tiene como primer elemento A y como resto la lista vacía (NIL o ()). ¿Entiendes ahora por qué (CONS 'A NIL)

es (À)? Para construir una lista sólo tiene que poner como último parámetro del CONS «más interno» () o NIL. Por ejemplo:

```
> (CONS ' ESTO
>       (CONS 'ES
>             (CONS ' UNA
>                   (CONS
'LARGA
  +
(CONS ' LISTA
  +)      )      )      )
```

cuyo valor es (ESTO ES UNA LARGA LISTA). En este caso, el CONS más interno es (CONS ' LISTA ()).

Construir listas de esta forma es un duro trabajo, de modo que existe en LISP una función que puede tener cualquier número de argumentos y que produce una lista con los resultados de evaluar cada uno de sus argumentos; es la función LIST.

```
> (LIST ' ESTO ' ES ' UNA ' LARGA ' LISTA)
```

vale (ESTO ES UNA LARGA LISTA).

Ya habrás observado la cantidad de paréntesis que hay que escribir en LISP. Esto no suele ser un problema salvo cuando hay que cerrar todos los paréntesis que permanecían abiertos. Para no tener que andar contando con cuidado, en MICROLISP puedes escribir el símbolo ⁄ para indicarle al genio que quieres cerrar todos los paréntesis.

Hasta ahora no hemos necesitado conocer las propiedades de los objetos que le presentábamos a LISP pues ya sabemos que 1 es un numeral, PERRO es un átomo y (JUAN.MARI) es un par. Sin embargo, vamos a empezar a tratar con variables (sí, ¡por fin!) y entonces sólo conoceremos el nombre del objeto y no sus características. Por eso voy a explicar algunas funciones, llamadas reconocedores, que tienen como fin informarnos de las características de los objetos que manejemos. ATOM nos dice si un SEXo es un átomo o no

```
> (ATOM 'A)
vale T.
> (ATOM 1)
vale T.
> (ATOM (CAR (A.B ⁄
```

vale T, pues (CAR (A.B)) vale A, que es un átomo. Pero

```
> (ATOM ' (CAR (A.B ⁄
```

vale NIL, pues (CAR (A.B)) es una lista. Esta es una característica muy importante de los programas LISP: todo programa desde otros programas LISP y evaluarlos inmediatamente. Esta es una de las ventajas de LISP sobre otros lenguajes de programación y una de las razones de que se utilice en IA (Inteligencia Artificial). Más adelante veremos ejemplos de esto.

Otros reconocedores son LITATOM, NUMBERP y PAIRP. El primero nos dice si un objeto es un átomo literal, el segundo si es un número y el tercero si es un par.

```
> (LITATOM ' A)
vale T, pero
> (LITATOM 1)
vale NIL.
> (NUMBERP 1)
vale T y
> (NUMBERP 'A)
vale NIL.
> (PAIRP ' (A B C))
```

vale T, pues (A B C) es la representación en forma de lista de (A. (B. (C.NIL))), que es un par con primer elemento A y segundo elemento (B. (C.NIL)).

La función NULL sirve para reconocer la lista vacía.

```
> (NULL () )
vale T. Espero que sepas cuanto vale.
> (NULL NIL)
```

Si no estás muy seguro pregúntaselo a tu genio. Un ejemplo más.

```
> (NULL ' (A B))
```

vale NIL.

Hasta aquí hemos visto algunas de las funciones internas del sistema. Ahora veremos cómo podemos definir las nuestras. Supongamos que estamos escribiendo un programa para encontrar la pareja ideal de algunas personas y cuando la encontramos se casan. Para casar a la feliz PAReja podríamos definir:

```
+ (DE BODA (JOSE ANA) (CONS JOSE ANA))
```

¿Por qué ahora JOSE y ANA no llevan QUOTE? Pues porque ahora JOSE y ANA actúan como variables (se llaman parámetros de la función) y no como objetos. Podríamos haberles llamado X e Y, pero una boda entre X e Y puede sonar un tanto «robótica». En LISP, cualquier átomo literal sirve como nombre de función o como nombre de variable, pero no puedes definir funciones con los nombres de las palabras reservadas de LISP. (Si quieres saber cuáles son estas palabras reservadas dile a tu intérprete (OBLIST). Esta función te muestra en pantalla todos los identificadores que MINILISP conoce hasta el momento).

La forma de definir funciones en LISP es semejante a como se hace en Basic: se utiliza una palabra reservada para indicar que lo que sigue es una definición, DE, y luego se da el nombre de la función a definir, BODA, seguido de los parámetros de la función, JOSE y ANA, y de las instrucciones para calcular el valor de la función, (CONS JOSE ANA).

Ahora podemos utilizar la función BODA que hemos definido de la misma forma que una función interna del sistema. Al usarla, deberemos dar valores a los parámetros. Por ejemplo.

```
+ (BODA ' SEGISMUNDO ' ROSALINDA)
```

y el feliz enlace sería (SEGISMUNDO. ROSALINDA).

La posibilidad de definir funciones nos permite modificar los nombres de las funciones del sistema LISP. Por ejemplo, las funciones CAR y CDR pueden tener nombres adecuados cuando se trata de obtener las componentes primera y segunda de un par, pero

```
> (CAR ' (A B C))
```

que vale A, y

```
> (CDR ' (A B C))
```

que vale (B C), no son nombres adecuados cuando se pretende obtener el primer elemento de una lista y el resto de la lista (la lista formada al quitar el primer elemento). Por eso:

```
> (DE PRIMERO (LIS) (CAR LIS))
> (DE RESTO (LIS) (CDR LIS))
```

serán las funciones que use cuando quiera hacer referencia al primer elemento de una lista y a su resto. Por ejemplo:

```
> (PRIMERO ' (A B C))
vale A, y
> (RESTO ' (A B C))
```

Vale (B C). Se trata de conseguir que los nombres de las funciones y de las variables te digan lo que hacen o son. Ahora tu puedes cambiar los nombres de algunas funciones LISP si los que tienen no te gustan. Por ejemplo:

```
> (DE SUMA (X Y) (+ X Y))
> (DE RESTA (X Y) (SUB X Y))
> (DE MULTIPLICA (X Y) (* X Y))
> (DE DIVIDE (X Y) (/ X Y))
```

De esta manera, (2*3) + 15 – (6/2) ahora se puede escribir como

```
> (SUMA (MULTIPLICA 2 3)
>       (RESTA 15
>              (DIVIDE 6 2
  +)      )      )
```

Antes de empezar a escribir programas más grandes debemos conocer algunas otras funciones. La primera es COND, que es algo semejante a la instrucción IF condicion THEN instruccion ELSE instruccion. La forma general de la función COND es:

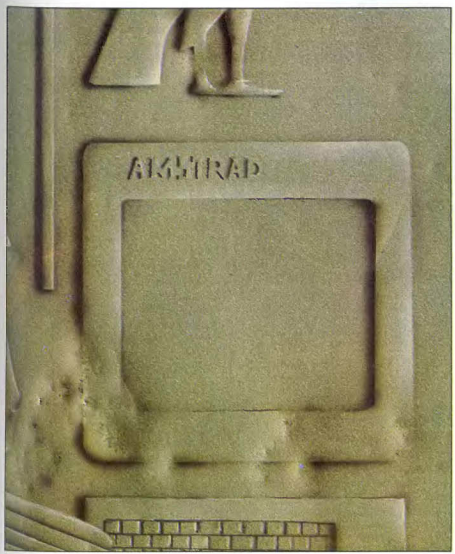
```
(COND (condición1 instrucción1)
      (condición2 instrucción2)
      ...
      (condiciónk instrucciónk) )
```

donde condición1, instrucción1, ..., condiciónk, instrucciónk deben ser sustituidos por llamadas a funciones. En Basic se escribiría así:

```
IF condición1 THEN instrucción1
ELSE IF condición2 THEN instrucción2
     ...
ELSE IF condiciónk THEN instrucciónk
```

Si todas las condiciones valen NIL entonces el genio ya sabe que han fallado todas las condiciones y el valor de la función COND es NIL. Si el valor de alguna de las condiciones no es NIL, entonces el valor de COND es el de la correspondiente instrucción. Veamos un ejemplo que nos aclara todo esto:

```
> (DE SEGUNDO (LIS)
>       (COND ((NULL LIS) ' (LA LISTA ES VACIA))
>             ((NULL (RESTO LIS))
(LA LISTA SOLO TIENE UN ELEMENTO))
>             (T (PRIMERO (RESTO
LIS)))
  +) )
```



Pidámosle al genio que nos diga cuál es el segundo elemento de algunas listas.

```
> (SEGUNDO () )
```

nos responde (LA LISTA ES VACIA), ya que ahora LIS vale () luego (NULL LIS) vale T, que es distinto de NIL, y por tanto el valor de COND es el de evaluar ' (LA LISTA ES VACIA).

```
> (SEGUNDO ' (A))
```

es (LA LISTA SOLO TIENE UN ELEMENTO). Ahora LIS vale (A), luego (NULL LIS) vale NIL y se pasa a evaluar la segunda condición. Como (RESTO LIS) es () se verifica (NULL (RESTO LIS)), que vale T y, por tanto, el valor de COND es el de evaluar ' (LA LISTA SOLO TIENE UN ELEMENTO).

```
> (SEGUNDO ' (A B))
```

vale *B*, ya que LIS vale (A B), (NULL LIS) es NIL y (RESTO LIS) es (B), luego (NULL (RESTO LIS)) es NIL. Como T vale T, el valor de COND es el de la tercera instrucción, o sea, el resultado de evaluar (PRIMERO (RESTO LIS)), que es lo mismo que (PRIMERO ' (B)), es decir, B.

El truco de poner T como última condición sirve para que siempre se verifique alguna condición. Se suele poner casi siempre, pero no es necesario.

De igual forma que en Basic, en LISP puedes utilizar las funciones booleanas AND, OR y NOT. AND y OR admiten cualquier número de parámetros, mientras que NOT sólo tiene uno. AND vale T si ninguno de sus argumentos vale NIL y vale NIL si alguno de sus argumentos vale NIL.

```
+ (AND (3 + = 1) (1 < + 23))
```

vale T.

```
+ (AND (NULL ()) (CAR ' (NILL)))
```

vale NIL, pues (CAR ' (NILL)) es NIL.

La función OR toma el valor del primer argumento cuyo valor no sea NIL y vale NIL si todos sus argumentos lo valen.

```
+ (OR (NULL ' (A)) (CAR ' (B C)))
```

vale B, pues (NULL ' (A)) es NIL, pero (CAR ' (B C) es B, que es distinto de NIL.

La función NOT vale T, si el valor de su argumento es NIL, y vale NIL cuando el valor de su argumento no sea NIL.

Si has estado jugando con el genio de MINILISP habrás visto que no es necesario que le digas que escriba un valor para que lo haga. Sin embargo, todos los mensajes que aparecen en la pantalla pueden ser molestos en ciertas ocasiones (cuando quieras que tu programa tenga una «salida» agradable, por ejemplo). En ese caso, si escribes (PREVAL NIL) suprimirás todos los mensajes que el genio te muestra en pantalla (excepto su saludo de «listo»). Cuando quieras que los mensajes aparezcan de nuevo escribe (PREVAL T), por ejemplo. Si has quitado los mensajes, el genio no escribirá nada si tú no se lo dices. Para eso están las funciones de entrada y salida de datos por pantalla: PRINT, PRINT1, TERPRI y READ.

La función PRINT escribe el valor de su único parámetro y no salta de línea. Como toda función LISP, debe tener un valor que es el de su argumento. Si escribes:

```
+ (PREVAL NIL)
+ (PRINT ' ESCRIBO)
```

aparecerá en pantalla.

```
ESCRIBO +
```

pues no ha saltado de línea. La función PRINT1 es igual que PRINT, pero con salto de línea. Luego:

```
+ (PRINT1 ' ESCRIBO)
ESCRIBO
+
```

La función TERPRI no tiene parámetros, vale NIL y su efecto es producir un salto de línea.

Para leer una S-expresión se utiliza la función READ, que tampoco tiene parámetros y cuyo valor es la S-expresión que ha leído.

```
+ (PRINT1 (READ))
+
```

y se queda esperando que le des una S-expresión. Dale cualquiera, por ejemplo:

```
+ (ESCRIBE)
(ESCRIBE)
+
```

Otra función de entrada/salida es CLS. La llamada:

```
+ (CLS)
```

borra la pantalla.

Antes hable de los programas que podían ser generados y evaluados desde otro programa LISP. Construir programas ya sabemos

```
+ (LIST ' SUMA 2 2)
```

vale (SUMA 2 2), pero no 4. Para evaluar esta lista se utiliza una nueva función, llamada EVAL.

```
> (EVAL (LIST ' SUMA 2 2)
```

si vale 4.

Explicaré ahora el ejemplo que abre este artículo.

```
+ (DE EQUAL (SEX1 SEX2)
+     (COND ((ATOM SEX1) (EQ
SEX1 SEX2))
+           ((EQUAL (CAR SEX1)
(CAR SEX2))
+            (EQUAL (CDR SEX1)
(CDR SEX2)))
+           (T NIL)
+ ) )
```

EQUAL es una función con dos parámetros, dos S-expresiones cualesquiera. Su valor es T si las dos S-expresiones SEX1 y SEX2 son igua-

les y NIL en otro caso. Si el primer SEXo es un átomo entonces el valor de EQUAL es el de (EQ SEX1 SEX2). Recordemos que EQ era cierto, T, si sus dos argumentos eran dos átomos literales iguales y NIL en otro caso. Por tanto, si SEX2 no es un átomo literal o, siéndolo, es distinto de SEX1 no es un átomo, pero si lo es SEX2 entonces el valor de EQUAL es NIL, como debe ser. Si ninguno de los SEXos es un átomo, entonces ambos deben ser pares o listas. Ninguno puede ser la lista *vacía* (pues (), o NIL, también es un átomo) luego podemos tomar sus CAR y sus CDR. Para que dos listas sean iguales elemento a elemento debe ocurrir que tengan el primer elemento igual y los restos de sus listas también sean iguales. Si sus primeros elementos no son iguales entonces las listas ya no serán iguales.

El esquema de la función EQUAL representa una forma usual de programar en LISP:

Para calcular el valor de una función aplicada a una lista hacer: Mientras la lista no sea vacía hacer:

Aplicar alguna operación sobre el primer elemento de la lista; aplicar la función al resto de la lista.

La función EQUAL es una función del sistema y que no es necesario que la programes. Sin embargo, si quieres ver como funciona puedes definirla con otro nombre, por ejemplo, IGUALES:

```
+ (DE IGUALES (SEX1 SEX2)...
Ahora escribe
+ (TRACE ' (IGUALES))
y llama a IGUALES con algunos argumentos algo complicados.
+ (IGUALES ' (A (B (C.D) E) () F)
+          ' (A (B (C.D) E) ()))
```

Irán apareciendo en pantalla los sucesivos argumentos de la función IGUALES y sus respectivos valores. Cuando ya no quieras ver la **«traza»** (TRACE) de la función, puedes escribir:

```
+ (UNTRACE ' (IGUALES))
```

La función IGUALES, tal como está definida, no funciona para números, es decir:

```
+ (IGUALES 5 5)
```

vale NIL.

Edición de programas

Existen en MINILISP algunas facilidades para una mejor escritura de programas:

i) ' SEX es una abreviatura de (QUOTE SEX).

El símbolo ' no puede formar parte de un identificador pues PEPE'S es interpretado por MINILISP como:

```
PEPE (QUOTE S)
```

ii) % hace que MICROLISP ignore el resto de la línea. Sirve para escribir comentarios.

iii) / cierra todos los paréntesis que quedan por cerrar. Se produce un error si ya estaban todos cerrados.

La edición de programas LISP es una tarea costosa, de manera que MINILISP no lleva incorporado editor. Por ello te recomendamos que cuando sepas que vas a escribir lo hagas con un editor de textos y luego le pases el fichero donde está tu programa a MINILISP [luego explicaremos como puede hacerse esto]. Desgraciadamente los programas no suelen funcionar a la primera, así es que tendrás que corregirlos y suele ser más cómodo hacer esto ayudado por MINILISP. Para ello te recomendamos que sigas los siguientes pasos:

1. Borra la definición incorrecta con la llamada:

(REMPROP ' < nombre + 'EXPR)

donde < nombre + es el nombre de la función errónea. Esta llamada devuelve la lambda-expresión que define a la función.

2. Escribe

(PUT ' < nombre + 'EXPR'

y a continuación copiar con las teclas del cursor la lambda-expresión que obtuvimos antes, corrigiendo la parte que esté mal. [Si la definición es muy larga y no cabe en 255 caracteres tendrás que pulsar alguna vez la teda [ENTER], pero ten cuidado de no partir algún identificador ya que, entonces, MINILISP lo consideraría como dos átomas].

3. Te puedes evitar escribir los paréntesis finales si escribes el **símbolo** /.

4. Pulsa [ENTER].

Funciones de comunicación externa

MINILISP puede mantener abiertos un fichero de entrada, uno de salida y una impresora. Para abrir y cerrar estas vías de comunicación se utilizan las funciones OPEN y CLOSE. Las dos funciones son de tipo e.s.

OPEN tiene dos argumentos. El primero debe tener como valor un átomo, que junto con la extensión .LIS se considera como nombre del fichero. El segundo argumento es el modo de operación: INPUT, OUTPUT, DEFS, PROPS o PRINTER.

Si el modo es INPUT el fichero se abre como fichero de entrada [desde disco o cinta]. En este modo, cada vez que MINILISP intente leer lo hará del fichero establecido. Cuando se alcanza el final de fichero, automáticamente MINILISP vuelve a establecer como fichero de entrada el teclado de la computadora [este hecho lo advierte con un pitido]. Para detectar el final del fichero se puede utilizar la función EOF, que no tiene argumentos. Su valor es T, si se ha alcanzado el final del fichero, y NIL en otro caso.

Si el modo es OUTPUT, se abre el fichero como fichero de salida y todo lo que MINILISP envíe a pantalla lo mandará también al fichero.

Con el modo DEFS se abre un fichero de salida en el que se guardan las definiciones de las funciones de usuario que MINILISP conozca hasta ese momento. Cada definición se guarda escrita en la forma (DE f (x1...xk) c) por lo que se genera un fichero que puede ser **«comprendido»** por MINILISP.

Si el fichero se abre en modo PROPS, es considerado como fichero de salida y en el que se escriben las propiedades de los identificadores que MINILISP conozca hasta el momento. Estas propiedades se guardan en la forma (PUT 'id 'prop 'val) con lo que se genera un fichero que puede ser leído por MINILISP. De esta forma se puede preservar el contexto actual y reanudar posteriormente la sesión desde el punto en el que dejó. Para ello bastaría hacer (OPEN ' < nombre + 'INPUT) donde < nombre + es el nombre del fichero [sin la extensión, ya que se asume .LIS].

El modo PRINTER sirve para mandar la salida por pantalla también a la impresora. Además, si se abre algún fichero en los modos DEFS o PROPS, aquello que se escriba también en estos ficheros se manda a la impresora.

La función CLOSE cierra el fichero que este abierto con el modo especificado por su parámetro. El valor del argumento debe ser INPUT, OUTPUT o PRINTER. Tiene como valor el de su argumento.

La función CAT, que no tiene parámetros, sirve para obtener un catálogo del disco o la cinta. Si se está utilizando cinta debe pulsarse la tecla [ESC] para terminar el catálogo. Si se usa disco esto no es necesario.

NOTA PARA LA CARGA DE LA CINTA

— El LISP de la cara A de la cinta es para usuarios del 664 y 6128. No es 72 utilizable directamente sino que le obligará a efectuar una copia en Disco. Siga las instrucciones que se le vayan dando.

— La cara 2 es para usuarios 464.

— No obstante al principio de cada cara puede usted ganar un ordenador. carguelas sea cual sea su modelo.

— Minilisp tarda en inicializarse una vez cargado, espere a que salga el promp.

— En ambas caras se encuentra información sobre el gran concurso AMSTRAD SEMANAL-OFITES INFORMATICA.

MANUAL DE MINILISP

Ofrecemos aquí un manual de referencia que sirva sobre todo, cuando ya domines el lenguaje, para resolver todas aquellas pequeñas dudas que siempre surgen.

R. G. Bernal

MINILISP: Es el encargado de cargar el resto del programa. También inicializa el tamaño de la tabla de símbolos.

2. LISP. BAS: Es el intérprete de LISP [en Basic]. Permanece constantemente en memoria [no así el anterior, que se borra al cargarse éste].

3. LISP. DAT: Este es un fichero ASCII que contiene los datos necesarios para rellenar el contexto inicial.

4. ERRLISP. BAS: Contiene la parte de gestión de errores y mensajes del sistema. Se carga tras borrar de la memoria la rutina de inicialización de las estructuras.

2.º Tipos de datos primitivos

Todas las consecuencias admisibles por MINILISP son expresiones simbólicas [S-expresiones]. Entre éstas podemos distinguir varios tipos y sus reconocedores:

S-expresiones:
- Atomos: ATOM
  - Literales: LITATOM
  - Números: NUMBERP
- Pares: PAIRP

Sintaxis de las S-expresiones

Para definir la sintaxis de un lenguaje de programación se suelen utilizar algunos formalismos como las reglas de Backus-Naur o los diagramas sintácticos. Aquí emplearemos el primero de ellos.

Las reglas en forma de Backus-Naur [reglas FBN] constan de tres partes: un lado izquierdo, que es lo que se quiere definir: el símbolo ::=, y un lado derecho, que es la definición. Los objetos que se quiere definir se suelen encerrar entre los símbolos < y >, por ejemplo < Número2 >. La yuxtaposición de objetos **[escribir uno al lado de otro]** indica que a continuación de un objeto del primer tipo se escribe uno del segundo. Por ejemplo:

< Número2 > ::= < Dígito > < Dígito >

indica que un objeto de tipo < Número2 > se representa por un objeto de tipo < Dígito > seguido por otro del mismo tipo. La barra vertical ! actúa como disjuntor [0 lógico]. Por ejemplo:

< Dígito > ::= 0 ! 1 ! 2 ! 3 ! 4 ! 5 ! 6 ! 7 ! 8 ! 9

quiere decir que un objeto de tipo < Dígito > es o bien 0, o bien 1, o bien..., o bien 9. Entonces, ejemplos de objetos de tipo < Número2 > serían 55, 13, 17, 80, 01, 00 y 99.

Otros símbolos que se usan en las reglas FBN son [,] y {, }. Los nombres de objetos se encierran entre corchetes para indicar que es opcional escribirlos o no. Por ejemplo:

< Número2—con—signo > ::= [< Signo >] < Número2 >

quiere decir que un objeto de tipo < Número2—con—signo > es un objeto de tipo < Número2 > que puede ir precedido o no de un signo. Si definimos:

< Signo > ::= + ! —

entonces ejemplos de objetos de tipo < Número2—con—signo + son —34, +13, —00, +00, 00, 99 y 17.

Las llaves { y } indican que el objeto al que encierran se puede repetir tantas veces como se quiera. Por ejemplo:

< Número—sin—signo + > ::= < Dígito + { < Dígito + }

quiere decir que un objeto de tipo < Número2—sin—signo + es una secuencia de objetos de tipo < Dígito + de cualquier longitud [debe tener al menos un dígito]. Por ejemplo, 12345567890, 1111111111111111111111111111111111, etc. Aunque conceptualmente el tamaño del objeto no esté limitado, físicamente sí que lo está (número de bytes de un computador, número de folios que puedes escribir, etc.).

Si has entendido todo esto no tendrás problemas en comprender las reglas FBN que definen la sintaxis de las S-expresiones y que damos a continuación. Sin embargo, si no las entiendes no te preocupes, pues más adelante se dan ejemplos.

< S-expresión + > ::= < Atomo + ! < Par + ! < Lista + >
< Atomo + > ::= < Literal + ! < Numeral + >
< Literal + > ::= < Marca + >
< Atomo + >
< Numeral + > ::= [< Signo + >] < Dígito + { < Dígito + } >
< Par + > ::= (< S-expresión + . < S-expresión + >)
< Lista + > ::= () ! (< Lista—de—objetos + >
< Lista—de—objetos + > ::= < S-expresión + { < Lista—de—objetos + } >
< Marca + > ::= < Letra + > !
< Símbolo + >
< Dígito + > ::= 0 ! 1 ! 2 ! 3 ! 4 ! 5 ! 6 ! 7 ! 8 ! 9
< Signo + > ::= + ! —
< Letra + > ::= A ! B ! C ! D ! E ! F ! G ! H ! I ! J ! K ! L ! M ! N ! O ! P ! Q ! R ! S ! T ! U ! V ! W ! X ! Y ! Z ! a ! b ! c ! d ! e ! f ! g ! h ! i ! j ! k ! l ! m ! n ! o ! p ! q ! r ! s ! t ! u ! v ! x ! y ! z !
< Símbolo + > ::= ! ! " ! # ! $! & ! ' ! + ! , ! / ! : ! ; ! < ! = ! + ! ? ! @ ! [!] ! : ! — ! { ! | ! } ! #

Obsérvese que como símbolos no se admiten:
- — Operador unario: —
- — Contracción de QUOTE: '
- — Terminador de paréntesis:)
- — Comentario: %
- — Punto del par: .

Veamos algunos ejemplos:

Literales

suma + suma—+ ES—suma$ +suma *no—Suma dolar$

Numerales

00000000 1523 —153 -00 0000001 32767 —32768

[En MINILISP los ceros a la izquierda no sirven para nada. Hay que notar también que números mayores que 32767 o menores que —32768 darán un error 6 de desbordamiento numérico.]

Atomos

Todos los ejemplos anteriores lo son también de átomos, pues los átomos o son literales o son numerales.

Pares

(suma. +) (UNO.1) (PAR. (uno.1)) ((par. de). (dos.pares)) (JOSE. (quiere.(a.(ANA.NIL))))

Listas

(JOSE quiere a ANA) ((LISTA.DE) (DOS.0) (TRES.PARES))

(LISTA (DE (LISTAS (DE (MAS) LISTAS)))

S-expresiones

Todos los ejemplos lo son también de S-expresiones, por la primera regla.

3.º Constantes predefinidas

En el contexto inicial de MINILISP se han introducido un reducido conjunto de constantes necesarias para el uso del intérprete. Podemos dividirlas en dos tipos, según que tengan o no atributo VALUE:

1. Constantes con valor:
NIL: valor NIL
T: valor T
PIZQ: valor (
PDER: valor)

2. Constantes sin valor:
VALUE, LAMBDA, FSUBR, SUBR, EXPR, FEXPR y MACRO.

4.º Lista de objetos y listas de propiedades

MINILISP mantiene una lista de los identificadores conocidos hasta el momento de una tabla hash. En cualquier momento es posible ver el contenido de esta tabla mediante la llamada:

(OBLIST)

Sin embargo, por razones de economía de memoria y de velocidad de proceso, la lista de objetos no se mantiene en una lista LISP por lo que OBLIST tiene valor NIL y como efecto colateral la impresión de la tabla de identificadores.

MINILISP inserta un identificador en la tabla de símbolos sólo y cuando éste no se encuentra ya en la tabla. Por tanto, el predicado de igualdad de átomos, EQ, se implementa mediante la comprobación de igualdad de punteros. Por este motivo, EQ no es válido para verificar ni la igualdad de pares ni la de números. Para pares debe utilizarse la función EQUAL. La igualdad entre números se comprueba con el predicado =.

Funciones que modifican la lista de objetos:

(REMOVE id) id debe evaluar a un átomo. Tiene como efecto suprimir el identificador id de la lista de objetos. Su valor es NIL.

(REMOVE 'PRIMERO) eliminaría el átomo PRIMERO de la lista de objetos y hace inaccesibles todas sus propiedades.

(COMPRESS lis) lis debe tener como valor una lista de átomos, debiendo ser el primero de estos un átomo literal. Comprime la lista para formar un identificador literal. El átomo resultante se coloca en la lista de objetos y se devuelve como valor de la función.

(COMPRESS '(VAR 1 4)) devuelve como valor VAR14 a la vez que inserta este átomo en la lista de objetos (si no estaba ya).

(EXPLODE id) id debe evaluar a un átomo. Tiene el efecto contrario a COMPRESS: genera la lista de los caracteres que componen id, que se da como valor de la función.

Los caracteres que no son dígitos se colocan en la lista de objetos.

(EXPLODE 'PALABRA) vale (P A L A B R A) y coloca los átomos P, A, L, B y R en la lista de objetos (si aún no estaban).

(EXPLODE 'VAR14) vale (V A R 1 4) e introduce los átomos V, A y R en la lista de objetos (si no estaban).

(EXPLODE 'A-Z) vale (A I-Z) pues el símbolo - no está permitido como átomo.

Listas de propiedades:

MINILISP mantiene una lista de propiedades (atributos) para cada identificador presente en la lista de objetos.

Existen tres funciones en MINILISP encargadas de crear, consultar y modificar listas de propiedades. Todas ellas evalúan sus argumentos:

(PUT id prop val) Coloca el resultado de evaluar val como valor de la propiedad prop del identificador id. Si ya existía la propiedad, da error y no modifica el valor de prop.

(PUT 'PRIMERO 'EXP' (LAMBDA (LIS) (CAR LIS))) coloca la lambda-expresión (LAMBDA (LIS) (CAR LIS) como valor del atributo EXPR en la lista de propiedades del identificador PRIMERO. El valor que devuelve es PRIMERO.

(GET prop id) Devuelve el valor de la propiedad prop en la lista de propiedades de id [o NIL si no aparece en la lista de propiedades de id].

(GET 'EXPR 'PRIMERO) vale (LAMBDA (LIS) (CAR LIS) si se hizo la instrucción anterior y NIL si no existe la propiedad EXPR de PRIMERO.

(REMPROP id prop) Elimina prop y su valor val de la lista de propiedades de id.

Devuelve val [o id y un mensaje si prop no existía como propiedad id].

'REMPROP 'PRIMERO 'EXPR) elimina la propiedad EXPR de la lista de propiedades de PRIMERO, y tiene como valor la S-expresión (LAMBDA (LIS) (CAR LIS)). Si después de esta llamada se hace (GET 'EXPR 'PRIMERO) se devuelve el valor NIL pues ya no existe la propiedad 'EXPR.

5.º Definición de funciones y macros

Funciones:

De los cuatro tipos posibles de funciones en LISP [según que se evalúen o no los argumentos y que el número de éstos sea fijo o variable] sólo dos son directamente definibles en MINILISP: eval-spread [evalúa y número de argumentos fijo] y noevalnospread [no evalúa y número de argumentos variable]. Ninguno de estos «definidores» evalúa sus argumentos:

(DE f (x1... xn) c) Define la función f como de tipo e.s. con variables formales x1, ..., xn y cuerpo c. Tiene como valor f y su efecto es poner la lambda-expresión (LAMBDA (x1 ... xn) c) como valor del atributo EXPR de f.

(DF f (1) c) Define la función f como de tipo n.n. con cuerpo c y lista de argumentos 1. Su efecto es poner la lambda-expresión (LAMBDA (1) c) como valor de la propiedad FEXPR de f. Tiene como valor f.

Macros:

Es este un tipo especial de definición. En MINILISP se puede utilizar mediante la función DM con la siguientes sintaxis:

(DM f (1) c) No evalúa sus argumentos. Tiene valor f y como efecto pone la lambda-expresión (LAMBDA (1) c) como valor del atributo MACRO de f.

Las llamadas a una macro son de la forma:

(f e1... en) (n cualquiera)

y se evalúan en dos fases:

i) Se evalúa c en un contexto ampliado con la ligadura de 1 a la forma (f e1... en).

ii) Se evalúa la forma obtenida en i) siendo el resultado el valor de la llamada.

Un ejemplo de macro es la función LET:

```
(DM LET (LA) (CONS (LIST 'LAMDA
        VARIABLES (PARES LA))
              (EXPCUALIF LA))
    (EXPRESIONES (PARES LA))))
```

siendo

```
(DE PARES (U) (RESTO (RESTO U)))
(DE VARIABLES (LISPAR)
(COND ((NULL LISPAR( ())
        (T (CONS (VAR (PRIME-
             RO LISPAR))
                 (VARIABLES
                  (RESTO
                   LISPAR))))
))
(DE EXPRESIONES (LISPAR)
(COND ((NULL LISPAR) ())
        (T (CONS (EXP (PRIMERO
        LISPAR))
                 (EXPRESIONES
                  (RESTO
                   LISPAR))))
))
(DE EXPCUALIF (U) (SEGUNDO U))
(DE VAR (PAR) (PRIMERO PAR))
(DE EXP (PAR) (SEGUNDO PAR))
(DE PRIMERO (LIS) (CAR (LIS)
(DE SEGUNDO (LIS) (PRIMERO
(RESTO LIS)))
(DE RESTO (LIS) (CDR LIS))
```

Supongamos que queremos evaluar la expresión:

```
(LET (LIST U V U)
     (U (CAR '(A B C)))
     (V (CDR '(D E F))))
```

En el primer paso se da como valor del parámetro LA de la función LET el valor

```
(LET (LIST U V U) (U (CAR '(A B C)))
(V CDR '(D E F))))
```

y se evalúa de forma usual el cuerpo de la función LET. El resultado de esta evaluación es:

```
(LAMBDA (U V) (LIST U V U) (CAR
'(A B C)) (CDR '(D E F)))
```

En el segundo paso se evalúa esta expresión dando como valor

```
(A (E F) A)
```

que pasa a ser el valor de la función LET.

6.° Construcciones iterativas

Aunque la programación imperativa no es necesaria en un lenguaje funcional como LISP su uso, no obstante, está justificado en un pequeño computador como en el que se ejecuta MINILISP, tanto por el ahorro de memoria [que no es excesiva y no se puede «derrochar» como de tiempo [no prolongar mucho programas ya que de por sí lentos], incluso hay problemas técnicos [como el del tamaño de la pila que soporta la recursión].

En la base de estas construcciones se hallan las funciones de asignación:

(SET e1 e 2) De tipo e.s., la evaluación de e1 debe ser una variable formal o declarada. Liga al valor de e1 el de e2 y devuelve como valor el de e2.

(SETQ id e) Es de tipo n.n. Tiene el mismo efecto y valor que (SET 'id e).

Los **«constructores»** de instrucciones iterativas son dos, ambos de tipo n.n.:

PROGN e1... ek)

Permite la llamada consecutiva a varias funciones. Tiene como valor el de ek [e1, ..., e(k-1) se evalúan por su «efecto»].

(PROG (x1...xn) e1...ek)

x1, ..., xn son las variables locales de la forma PROG (sólo se pueden utilizar dentro del cuerpo de la instrucción PROG). Las formas e1, ..., ek constituyen el cuerpo de la forma PROG.

La lista de variables puede ser (), pero debe existir. Las variables de una forma PROG se inicializan todas a NIL.

Las formas que aparecen en el cuerpo de una forma PROG pueden ser: condicionales, formas aplicativas, formas RETURN, formas GO, etiquetas y asignaciones.

Las formas condicionales no necesitan tener un lado izquierdo cierto [en una forma condicional fuera de un PROG, si todos los lados izquierdos se evalúan a NIL se genera un mensaje de aviso]. En este caso toma el valor NIL y se cede el control a la siguiente instrucción.

(RETURN e) Es de tipo e.s. Termina la evaluación del PROG, devolviendo como valor el de e.

(GO id) Es de tipo n.n. Cede el control a la instrucción siguiente a la etiqueta id [para MINILISP, una etiqueta es cualquier átomo literal]. Si id no es una etiqueta o no existe dentro de la forma PROG se genera un error.

Los argumentos de PROG son evaluados uno tras otro, a partir del segundo [el primero es la lista de variables], con las siguientes excepciones:

a) Un argumento atómico no se evalúa: se considera como una etiqueta.

b) La evaluación de la función GO hace que LISP continúe la evaluación de PROG tras la etiqueta que es el argumento de GO.

c) Si la función RETURN se evalúa, entonces acaba la evaluación de PROG, y el valor de la función PROG es el valor del argumento de RETURN.

d) Si el último argumento de PROG ha sido evaluado y no era ni una forma GO ni una forma RETURN entonces el valor de PROG es NIL.

e) Si un argumento de PROG es un COND y ninguno de sus lados izquierdos vale T entonces la evaluación continúa con el siguiente argumento de PROG.

7.° Aritmética y lógica

Aritmética

Tanto los operadores aritméticos como los relacionales entre números son de tipo e.s. [Los números se evalúan a sí mismos] y tienen dos argumentos. Son los siguientes:

(+ e1 e2) e1 + e2
(SUB e1 e2) e1 — e2
(* e1 e2) e1 * e2
(DIV e1 e2) e1/e2 [división entera]
(ê1 e2) $e1^{e2}$
(MAX e1 e2) el máximo entre e1 y e2

Los operadores relacionales son:

= + < + = <=

Lógica

Las funciones lógicas existentes en MINILISP son:

(NOT e). Es de tipo e.s. Coincide con la función NULL: vale T, si [EQ e NIL), y NIL, en otro caso.

(AND e1... ek). Es de tipo n.n. Para calcular su valor se evalúan las ei en orden, desde la izquierda hacia la derecha. Si alguna da NIL se devuelve NIL y no se evalúan las demás; si no, se devuelve T.

(OR e1... ek). Es de tipo n.n. Para calcular su valor se evalúan en orden las ei. Devuelve el valor del primer ei que no sea NIL, o NIL si todos los ei valen NIL.

8.° Funciones MAP

Las mejoras introducidas en esta nueva versión de MINILISP permiten que se utilicen argumentos funcionales, de ahí que sea posible la definición de funciones map. Por razones de espacio el intérprete de MINILISP no lleva incorporada ninguna.

Un ejemplo de función map es el siguiente:

```
(DE MAP (LIS FUN)
 (COND ((NULL LIS) NIL)
(T (PROGN (FUN LIS) (MAP (RESTO
```

LIS) FUN))
))

FUN es el argumento funcional, es decir, el valor de FUN es o bien el nombre de una función o una lambda-expresión. Entonces (FUN LIS) quiere decir que el valor de FUN se aplica al valor de LIS (se espera que LIS sea una lista). Otros ejemplos de funciones MAP pueden verse en el fichero de ejemplo SURTIDO.LIS.

9.º Funciones de edición

Existen en MINILISP una serie de facilidades para la mejor escritura de programas:

i) 'SEX es una abreviatura de (QUOTE SEX).

El símbolo ' no puede formar parte de un identificador pues PEPE'S es interpretado por MINILISP como:

PEPE (QUOTE S)

ii) % hace que MICROLISP ignore el resto de la línea. Sirve para escribir comentarios.

iii)) cierra todos los paréntesis que quedan por cerrar. Se produce un error si ya estaban todos cerrados.

La edición de programas LISP es una tarea costosa, de manera que MINILISP no lleva incorporado editor. Por ello te recomendamos que cuando sepas que vas a escribir lo hagas con un editor de textos y luego le pases el fichero donde está tu programa a MINILISP. Desgraciadamente los programas no suelen funcionar a la primera, así es que tendrás que corregirlos y suele ser más cómodo corregirlos ayudado por MINILISP. Para ello te recomendamos que sigas los siguientes pasos:

1. Borra la definición incorrecta con la llamada:

(REMPROP ' < nombre + 'EXPR)

donde < nombre + es el nombre de la función errónea. Esta llamada devuelve la lambda-expresión que define a la función.

2. Escribe (PUT ' < nombre + 'EXPR' y a continuación copiar con las teclas del cursor la lambda-expresión que obtuvimos antes, corrigiendo la parte que esté mal. (Si la definición es muy larga y no cabe en 255 caracteres tendrás que pulsar alguna vez la tecla [ENTER], pero ten cuidado de no partir algún identificador ya que, entonces, MINILISP lo consideraría como dos átomos).

3. Te puedes evitar escribir los paréntesis finales si escribes el símbolo /

4. Pulsa ENTER.

10.º Entrada y salida

(READ). No tiene argumentos. Su valor es la primera S-expresión que lee del dispositivo actual de entrada.

(PRINT e). Es de tipo e.s. Escribe, con salto de línea, el valor de e, que también es el valor de la función.

(PRINT1 e). Igual que PRINT, pero sin salto de línea.

(TERPRI). No tiene argumentos. Tiene como valor NIL y como efecto produce un salto de línea en los dispositivos de salida establecidos.

(SPACE e). Es de tipo e.s. El valor de e debe ser un número que es el número de caracteres blancos que deja. (No tiene salto de línea). El valor de la función es el mismo que el de e.

Funciones de comunicación externa

MINILISP puede mantener abiertos un fichero de entrada, uno de salida y una impresora. Para abrir y cerrar estas vías de comunicación se utilizan las funciones OPEN y CLOSE. Las dos funciones son de tipo e.s.

OPEN tiene dos argumentos. El primero debe tener como valor un átomo, que junto con la extensión .LIS se considera como nombre del fichero. El segundo argumento es el modo de operación: INPUT, OUTPUT, DEFS, PROPS o PRINTER.

Si el modo es INPUT el fichero se abre como fichero de entrada [desde disco o cinta]. En este modo, cada vez que MINILISP intente leer lo hará del fichero establecido. Cuando se alcanza el final de fichero, automáticamente MINILISP vuelve a establecer como fichero de entrada el teclado de la computadora [este hecho lo advierte con un pitido]. Para detectar el final del fichero se puede utilizar la función EOF, que no tiene argumentos. Su valor es T, si se ha alcanzado el final del fichero, y NIL en otro caso.

Si el modo es OUTPUT, se abre el fichero como fichero de salida y todo lo que MINILISP envíe a pantalla lo mandará también al fichero.

Con el modo DEFS se abre un fichero de salida en el que se guardan las definiciones de las funciones de usuario que MINILISP conozca hasta ese momento. Cada definición se guarda MINILISP conozca hasta ese momento. Cada definición se guarda en la forma (DE f (x1... xk) c) por lo que se genera un fichero que puede ser «comprendido» por MINILISP.

Si el fichero se abre en modo PROPS, es considerado como fichero de salida y en él se escriben las propiedades de los identificadores que MINILISP conozca hasta el momento. Estas propiedades se guardan en la forma (PUT 'id 'prop 'val) con lo que se genera un fichero que puede ser leído por MINILISP. De esta forma se puede preservar el contexto actual y reanudar posteriormente la sesión desde el punto en el que se dejó. Para ello bastará hacer (OPEN ' < nombre + 'INPUT) donde < nombre + es el nombre del fichero [sin la extensión, ya que se asume .LIS].

El modo PRINTER sirve para mandar la salida por pantalla también a la impresora. Además, si se abre algún fichero en los modos DEFS o PROPS, aquello que se escriba también en estos ficheros se manda a la impresora.

La función CLOSE cierra el fichero que esté abierto con el modo especificado por su parámetro. El valor del argumento debe ser INPUT, OUTPUT o PRINTER. Tiene como valor el de su argumento.

La función CAT, que no tiene parámetros, sirve para obtener un catálogo del disco o la cinta. Si se está utilizando cinta debe pulsarse la tecla [ESC] para terminar el catálogo. Si se usa disco esto no es necesario.

11.º Errores

MINILISP informa de dos tipos de errores: los producidos por él y los generados por Basic. De estos últimos sólo indica el número de error [estos números los puedes encontrar en el manual del ordenador]. Los errores producidos por Basic suelen ser errores numéricos [6, 11] o de espacio para cadenas lleno [14] si has utilizado identificadores excesivamente largos [se ha calculado que cada identificador ocupará en media 8 caracteres], pero eventualmente es posible que aparezcan errores de otro tipo si se ha producido algún fallo anterior.

Los errores generados por fallos en los programas LISP están documentados con mensajes de error y avisos.