GRAN BIBLIOTECA

AMSTRAD

LOGO

UNA TORTUGA EN LA ESCUELA

EDICIONES INGELEK

GRAN BIBLIOTECA AMSTRAD

11

LOGO

Director editor:
Antonio M.ª Ferrer Abelló

Director de producción:
Vicente Robles

Director de la obra:
Fernando López Martínez

Redactor técnico:
Victoriano Gómez Delgado

Colaboradores:
H-L Servicios Informáticos
Pilar Manzanera Amaro

Diseño:
Bravo/Lofish

Maquetación:
Carlos González Amezúa

Dibujos:
José Ochoa

Fotografía:
Grupo Gálata

ISBN del tomo: 84-7708-042-9
ISBN de la obra: 84-7708-004-6
Fotocomposición: Andueza, S. A.
Imprime: Eurosur
Depósito Legal: M-85231987
Precio en Canarias, Ceuta y Melilla: 435 ptas.
Junio 1987

LOGO

INTRODUCCIÓN

o podíamos dejar fuera de esta colección de libros destinada a los AMSTRAD un tema como el LOGO, que paulatinamente cobra mayor importancia, por varios motivos. En primer lugar, se trata de un lenguaje que está demostrando su eficacia a nivel educativo, como introducción para los más pequeños, incluso muy pequeños, dentro del mundo de la programación. Por otra parte, el conocimiento del LOGO es más interesante para los usuarios de AMSTRAD, que para los de cualquier otro ordenador, dado que una versión de este lenguaje, acompaña de forma gratuita a todas las versiones de disco de las series CPC y PCW.

Este libro pretende ser una guía completa de explicación de los comandos de este lenguaje, con el fin de que el lector vislumbre las limitaciones y posibilidades que éste le ofrece. Se trata pues, de una considerable ampliación del manual que acompaña al ordenador, el cual, en bastantes ocasiones, apenas si explica nada, mal endémico de este tipo de guías de usuario.

Con respecto al contenido de este decimoprimer volumen de la GRAN BIBLIOTECA AMSTRAD, refleja todas aquellas primitivas

de importancia, incluyendo ejemplares aclaratorios sobre su empleo, con programas y gráficos para contribuir a su correcta asimilación. Hecho de especial interés es la confección en el apéndice de una lista completa de primitivas de las diferentes versiones de DR. LOGO (Digital Research LOGO), especificando los modelos de AMSTRAD en los que se encuentran disponibles.

Un último consejo: aunque aquí ofrecemos una serie de ejemplos aclaratorios sobre la función de las diferentes primitivas, el mejor método para aprender LOGO es realizar pruebas y cambios en los programas listados; así pues, no nos cohíbamos y efectuemos todas las alteraciones que consideremos oportunas; sin duda, saldremos ganando.

CARGAMOS EL PROGRAMA

Independientemente del modelo de ordenador AMSTRAD que tengamos es imprescindible, previamente a trabajar con LOGO, cargar el CP/M. Así pues, en la serie CPC se introduce el disco con el sistema operativo en la unidad de disco A, la principal, con la cara del mismo donde se encuentra el CP/M hacia arriba y se escribe:

|CPM

A continuación, debemos pulsar la tecla ENTER o RETURN. Si trabajamos con la versión 2.2 (en los CPC 464, 472 y 664), la pantalla se tornará de color azul claro, mostrándonos poco tiempo después el mensaje de presentación. En el CPC 6128, que soporta la versión PLUS o 3.0, la pantalla permanecerá en azul oscuro.

Para cargar el sistema operativo en los ordenadores PCW, se introduce el disco con la cara en que se encuentra en el lado izquierdo. Si la carga no se inicia automáticamente, se pulsará la barra espaciadora. En la pantalla aparecerán una serie de franjas, indicándonos que se está cargando. En este caso, también la versión de CP/M soportada es la 3.0 (Plus).

Cuando hayamos cargado el sistema operativo, en cualquiera de los ordenadores, aparecerá el *prompt* (mensaje que nos invita a introducir un comando), con la unidad de disco en uso, que al empezar a trabajar siempre será la «A», o principal:

A>

En los PCW, para adaptar el teclado a la versión LOGO debemos teclear ahora:

language 0

A continuación, cambiaremos el disquete e introduciremos el de LOGO. Es importante que este disco NO esté protegido contra escritura, por tanto, la pestañita de protección debe hallarse visible. En la carga del lenguaje, no es posible que suceda ningún accidente que estropee nuestra copia; no obstante, siempre debemos evitar trabajar con los disquetes originales. El proceso inicial debería ser extraer copias de seguridad, con las que operar normalmente. A tal fin, podemos consultar el volumen segundo de esta misma colección, SISTEMA OPERATIVO CP/M.

Si trabajamos con CP/M 2.2, para cargar el LOGO debemos escribir:

A>SUBMIT LOGO2

Si es con el CP/M 3.0 (o PLUS) en el CPC 6128, escribiremos, en cambio:

A>SUBMIT LOGO3

Finalmente, con el CP/M 3.0 de los PCW:

A>SUBMIT LOGO

Siempre después de haber tecleado el correspondiente mensaje, es imprescindible pulsar la tecla RETURN o ENTER, según el caso.

A continuación tendremos un nuevo mensaje de presentación, siguiendo el proceso con un limpiado de pantalla, apareciendo en la parte superior el prompt del LOGO:

?

Este es el momento en el que podemos trabajar con él y empezar a escribir los primeros comandos.

Si, a pesar de las advertencias, utilizamos los discos originales, en los PCW debemos realizar algunos cambios en el proceso de lectura del LOGO. Tras cargar en el mismo disco del CP/M la adaptación del teclado, escribiremos:

A>SUBMIT

```
Welcome to

Amstrad LOGO V2.0
Copyright (c) 1983, Digital Research
Pacific Grove, California

Dr. Logo is a trademark of
Digital Research

Product No. 6002-1232

Please Wait
```

Instantes más tarde, aparecerá en la pantalla:

CP/M 3 SUBMIT Version 3.0
Enter File to SUBMIT:

Así, se solicita que escribamos el nombre de un fichero. Debemos cambiar ahora de discos e introducir el de LOGO (el lado donde se encuentra este lenguaje debe estar a la izquierda, como siempre) desprotegido de escritura, escribiendo a continuación:

LOGO

Este proceso es complicado, porque en el disco original del LOGO no cabe el fichero SUBMIT. Por tanto, al sacar las copias se debería intentar unir ambos ficheros, según se explica en la guía del ordenador. En este caso, el proceso de carga es como el ya indicado anteriormente.

```
CP/M Plus  Amstrad Consumer Electronics plc

v 1.2, 61K TPA, 1 disco, 112K disco M:

A)submit
CP/M 3 SUBMIT Version 3.0
Enter File to SUBMIT: LOGO
```

LOS GRÁFICOS

asi todos nosotros hemos oído hablar algo sobre el LOGO, principalmente de su tortuga, lo más conocido del lenguaje. Ahora tendremos ocasión de conocerla personalmente, y poder comprobar sus reacciones ante nuestras órdenes.

LA TORTUGA

La tortuga es un método sencillo y rápido de dibujar, en actual proceso de incorporación a otros lenguajes. Conociendo unos pocos comandos, podemos mover este simpático animalito y realizar gráficos complejos, lo cual resultaría bastante más difícil siguiendo otro sistema.

No obstante, hemos cargado nuestro LOGO, y no estamos viendo a la tortuga. Lo único que tenemos en la pantalla es el signo de fin de interrogación (el *prompt* del lenguaje). Tecleemos lo siguiente:

?fd 50

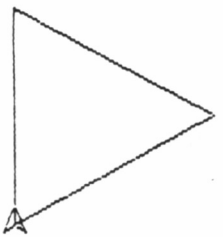
No nos olvidemos que después de escribirlo debemos pulsar la tecla RETURN o ENTER. La pulsación de alguna de estas teclas es imprescindible para que el ordenador ejecute las órdenes; es nuestra forma de decirle que ya hemos terminado de escribir y ahora debe ponerse a trabajar.

Si todo lo hemos hecho bien, las consecuencias se habrán hecho patentes en la pantalla. En un primer momento, apareció un triangulito, inmediatamente desapareció, y a continuación, vimos cómo se dibujaba una recta y al final de ella volvía a aparecer el mismo triangulito de antes.

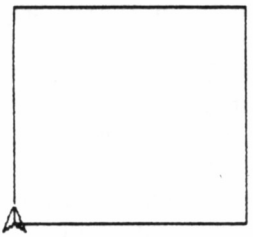
Ya hemos realizado nuestro primer gráfico y, al mismo tiempo, conocemos a la tortuga, que es ese triangulito, o más propiamente hablando, una punta de flecha. Desde luego, hace falta una buena dosis de imaginación para sacarle el parecido con una tortuga, pero al menos cumple su misión perfectamente.

El mensaje que hemos tecleado **(fd 50)** es el responsable del dibujo de la recta. **fd** (del inglés *ForwarD*, adelante) es una primitiva (así se llaman en LOGO los comandos) que indica a la tortuga que avance su posición los pasos que a continuación se expresan; en nuestro ejemplo 50.

Podríamos esperar que la recta hubiera salido de la esquina inferior izquierda de la pantalla, dado que éste es el punto normal de origen de gráficos en BASIC. Sin embargo, en LOGO este punto de origen es el centro de la pantalla, gracias a lo cual nos ahorramos desplazar la tortuga al centro cada vez que efectuemos un dibujo, para evitar que nuestro galápago se salga por los bordes de la pantalla.



```
?fd 200 rt 120
?fd 200 rt 120
?fd 200 rt 120
?repeat 3 [fd 200 rt 120]
?
```



```
?repeat 4 [fd 200 rt 90]
?
```

LA TORTUGA QUE GIRA

Ahora vamos a realizar un gráfico algo más complicado. Para ello, indicaremos a la tortuga que regrese a su punto de origen y limpie al mismo tiempo la pantalla. Lo que en un correcto inglés se diría *Clear Screen,* en LOGO se dice:

?cs

La tortuga estará, aproximadamente, en el centro de la pantalla, mirando hacia la parte de arriba, y la recta dibujada anteriormente habrá desaparecido.

Escribamos ahora, sin olvidar pulsar RETURN o ENTER después de cada línea:

?fd 100 rt 120
?fd 100 rt 120
?fd 100 rt 120

Según tecleábamos cada línea, la tortuga se iba moviendo y podíamos ver cómo se formaba un triángulo. En la primera línea, le dijimos que avanzara 100 pasos (ya conocemos esa primitiva) y después diera un giro de 120 grados hacia la derecha (de *RighT,* derecha). La segunda línea es igual que la anterior: dibuja una recta y gira a la derecha. Y, la tercera, que cierra la figura, cumple la misma función, teniendo, al final del proceso, un triángulo en la pantalla y la tortuga en su posición de inicio, mirando hacia arriba.

Si hubiéramos suprimido la última primitiva de la tercera línea (**rt 120**), el dibujo hubiera sido el mismo, pero la tortuga no estaría en su

posición inicial, sino en la misma dirección que el tercer lado y al final de éste.

Para realizar un triángulo punto a punto, el método que siempre se ha utilizado, supone efectuar cálculos de dónde debe partir el dibujo y en qué punto ha de terminar, repitiéndolos para cada lado. Quizás en una figura tan simple como un triángulo, no se puedan apreciar las ventajas del uso de la tortuga, pero, sin duda, las advertiremos según compliquemos los dibujos.

Este tipo de gráficos se confeccionan como si dirigiéramos un coche a distancia, en este caso una tortuga, indicándole todos los pasos que debe seguir: avanzar, girar a derecha, retroceder... En la pantalla, nosotros somos los conductores de esa tortuga motorizada a la que dictamos los movimientos a realizar.

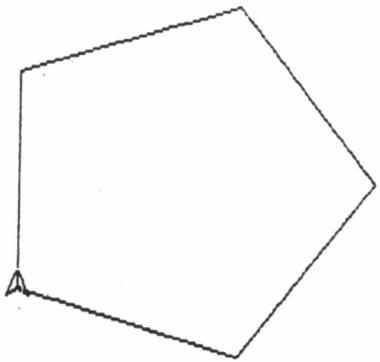
MOVIMIENTOS QUE SE REPITEN

Volviendo a nuestro ejemplo anterior, vemos de qué forma tan sencilla podemos dibujar un triángulo, pero se puede simplificar aún más con sólo:

?repeat 3 [fd 100 rt 129]

Antes de teclearlo y comprobar su efecto, debemos borrar el dibujo anterior y la tortuga debe estar en el centro. Ya sabemos como conseguirlo:

?cs



?repeat 5 [fd 200 rt 72]

Ahora podemos ejecutar la anterior línea de comando. En la pantalla volvemos a tener el mismo triángulo. La primera primitiva, **repeat,** cuyo significado en inglés es repetir, tiene la función de ejecutar lo que se encuentra a continuación, el número de veces expresado, tres en este caso. Las instrucciones que se van a repetir deben figurar entre corchetes [].

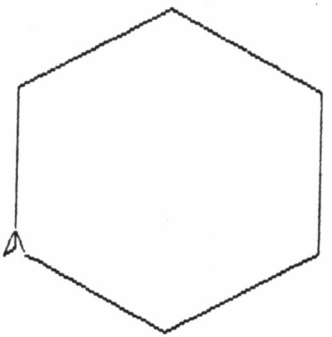
Con nuestros conocimientos actuales, es fácil escribir las primitivas para realizar cualquier figura regular. Lo único que debemos variar es el número de veces que se van a repetir las instrucciones que dibujarán los lados, y el ángulo con que se trazan. Así, para realizar las primeras figuras regulares, las primitivas que debemos teclear son:

CUADRADO: **?repeat 4 [fd 100 rt 90]**
PENTAGONO: **?repeat 5 [fd 100 rt 72]**
HEXAGONO: **?repeat 6 [fd 75 rt 60]**

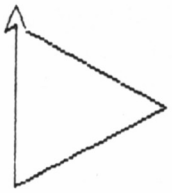
Nuestra tortuga es bastante hábil, y conoce otros comandos que nos permitirán disponer de ella con mayor flexibilidad de movimiento. Borremos la pantalla con **cs** (siempre es aconsejable antes de ejecutar cualquier programa o comando limpiar la pantalla con esta primitiva, de no ser así es posible que nuestros gráficos tomen formas extrañas) y escribamos lo siguiente:

?repeat 3 [bk 150 lt 120]

Vemos como en la parte inferior se dibuja un nuevo triángulo y, si nos fijamos bien, que este segundo se ha trazado moviéndose la tortuga hacia atrás. Este es el efecto de la nueva primitiva **bk** (del inglés *Backwards,* atrás). Por otra parte, la misión de **lt** ya la podemos imaginar: girar hacia la izquierda (en inglés *LefT).*



```
?repeat 6 [fd 150 rt 60]
?
```



```
?repeat 3 [bk 150 lt 120]
?
```

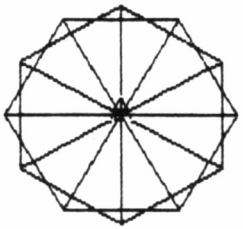
Antes de seguir, es necesario que hagamos una pequeña aclaración sintáctica: todas las primitivas que utilicemos deben teclearse en minúsculas, porque Dr. LOGO, la versión de este lenguaje para AMSTRAD, no admite que se escriba en mayúsculas.

Una vez aclarada esta cuestión, vamos a complicar los dibujos para comprobar la efectividad de los gráficos de tortuga. Escribamos la siguiente línea:

?repeat 12 [repeat 3 [fd 75 rt 120] rt 30]

Sorprendente figura ¿verdad? Si examinamos lo que hemos escrito, veremos que se repite 12 veces lo que viene a continuación, que es el proceso por nosotros conocido de dibujar un triángulo, para después de cada uno, producirse un giro de 30 grados. Al final del proceso, tendremos en la pantalla 12 triángulos con un vértice común.

Los usuarios de CPC, habrán podido comprobar también otro curioso fenómeno: al teclear el espacio tras el último **rt,** se habrá escrito un signo de admiración en la pantalla y efectuado un salto de línea en la zona de introducción de comandos. Esto no debe importarnos,



```
?repeat 12 [repeat 3 [fd 100 rt 120] rt 30]
?
```

con ello simplemente se nos indica que se ha sobrepasado la longitud de una línea de comando, pero esto no afecta en absoluto al correcto funcionamiento de la primitiva.

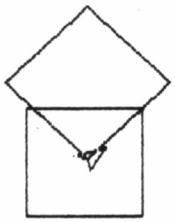
LOS PROCEDIMIENTOS

Aunque hasta ahora todas las series de primitivas que hemos ensayado han sido ejecutadas en modo directo, LOGO dispone de la posibilidad de realizar programas, lo cual nos permite llevar a cabo una serie de acciones de forma continuada, e incluso almacenarlas en diskette, lo cual nos evita teclear el programa cada vez que queramos utilizarlo. Escribamos lo siguiente:

to cuadrados

Al contrario de lo ocurrido en ocasiones anteriores, no hemos apreciado ningún cambio en la pantalla, ni la tortuga se ha desplazado. Sólo, si nos fijamos bien, vemos que el prompt ya no es el que teníamos hasta ahora (?), sino un signo mayor que (>). Esto se debe a que hemos indicado a LOGO que vamos a escribir un programa, mediante la primitiva **to** (en inglés, para). La palabra que viene a continuación es el nombre del programa: cuadrados. Ahora ya podemos codificarlo, análogamente a como lo hacíamos en modo directo:

```
>repeat 4 [fd 100 rt 90]
>pu rt 45 fd 75 pd
>repeat 4 [fd 100 lt 90]
>end
cuadrados defined
?
```



```
?cuadrados
?
```

He aquí nuestro primer programa. Al igual que cuando escribimos **to** el prompt cambia de forma, se produce un efecto similar al escribir **end,** esta vez retornando al signo de interrogación. Asimismo, antes de producirse este cambio, el sistema nos anuncia que nuestro programa ha sido admitido mediante el mensaje *cuadrados defined* (cuadrados definido).

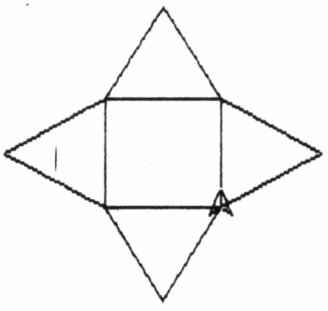
Para ejecutar un programa en LOGO, no existe ningún comando especial, como puede ser el RUN del BASIC, sino que éste empieza a funcionar llamándole por su nombre, como si se tratara de una primitiva más:

?cuadrados

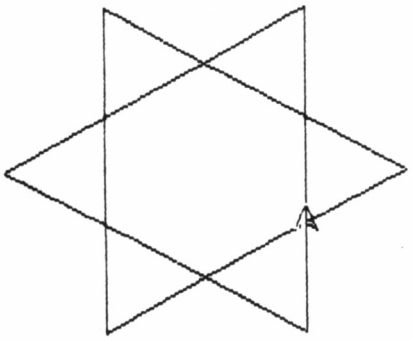
En ese momento, veremos cómo se dibuja primeramente un cuadrado, y la tortuga dando un salto, moviéndose «sin dejar huella», comienza a dibujar el siguiente. En realidad, más que un salto, es como si el animalito cogiera su lápiz y lo levantara para no dejar trazo; tal es el efecto conseguido con la primitiva **pu** (abreviatura en inglés de *Pen Up,* lápiz arriba). Para que la tortuga vuelva a dibujar normalmente, deberá bajar el lápiz. Esta es la misión de **pd** (de *Pen Down,* lápiz abajo).

PRIMITIVAS Y PROGRAMAS

Cuando hicimos nuestro primer programa, ya indicamos que para ejecutarlo había que escribir su nombre, lo cual suponía que habíamos



```
?repeat 4 [triangulo fd 100 lt 90]
?
```



```
?trihexa
?
```

creado una nueva primitiva. Este mecanismo lo veremos más claro en el siguiente ejemplo:

```
?triangulo
>repeat 3 [fd 100 rt 120]
>end
```

Ya sabemos bien lo que hace la reunión de estas primitivas. La única diferencia estriba en que ahora no se introducen directamente, sino como un programa. Para ejecutarlo, escribiremos:

?triángulo

Veamos ahora el efecto de la siguiente instrucción:

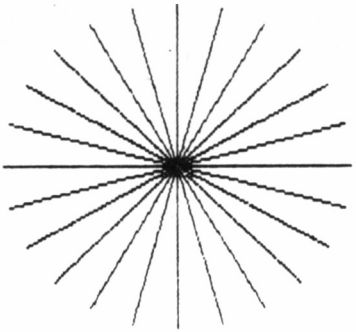
?repeat 4 [triangulo fd 100 lt 90]

Se va a ejecutar cuatro veces lo que se encuentra entre corchetes, entre lo cual se cuenta triángulo que, evidentemente, no es ninguna primitiva, sino el programa que escribimos antes. Por tanto, cuando LOGO se encuentra esta palabra, la busca en su memoria para comprobar si para él tiene algún significado. Cuando la encuentra, ejecuta su función y posteriormente continúa con el proceso original.

La serie anterior de comandos es equivalente, como nos podemos figurar, a:

?repeat 4 [repeat 3 [fd 100 rt 120] fd 100 lt 90]

Triángulo es, pues, una nueva primitiva para LOGO, que podemos utilizar igual que cualquiera otra de aquéllas. De modo similar, se pueden efectuar llamadas a un programa desde otro, en este caso, aquél recibe el nombre de PROCEDIMIENTO:



```
?circulo
```

```
?to trihexa
>repeat 6 [triangulo fd 100 lt 60]
>end
```

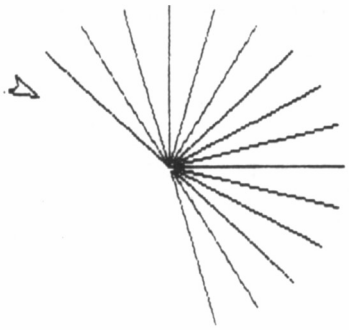
Cuando ejecutamos el programa, escribiendo trihexa, y llega a triángulo, LOGO busca el programa con ese nombre, ahora llamado procedimiento o programa secundario, y lo ejecuta. Una vez que ha sido cumplimentado, continúa el proceso de trihexa, terminando el programa.

Esta llamada a un procedimiento o programa puede realizarse desde sí mismo. Es decir, un programa se ejecuta y a lo largo de su proceso se encuentra con un nombre de procedimiento que es el suyo propio; por tanto, vuelve a comenzar su ejecución. Este mecanismo recibe el nombre de RECURSIVIDAD. Veamos el siguiente ejemplo recursivo:

```
?to circulo
>bk 150
>fd 150 rt 15
>circulo
>end
```

Al ponerlo en marcha, primeramente se van dibujando una serie de rectas que acaban formando un círculo. Cuando ya lo tenemos completo, en la pantalla aparentemente no ocurre nada nuevo; sin embargo, el programa continúa ejecutándose, porque no aparece el *prompt* (?). Lo podremos comprobar apretando la tecla para interrumpir, la ejecución (CONTROL + G en los CPC y ALT + G en los PCW). Inmediatamente aparecerá en la pantalla el mensaje:

?Stopped! in...



```
?circulo1
```

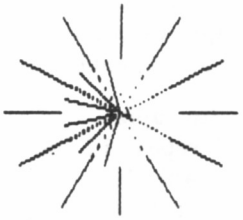
Figurando en vez de los puntos suspensivos, el lugar en el cual se ha detenido el proceso. Este es un ejemplo de la recursividad: cuando el programa llega a círculo, recomienza su ejecución.

LÁPIZ ARRIBA, LÁPIZ ABAJO

Hagamos un nuevo programa, variando ligeramente el anterior:

```
?to circulo1
>px bk 150
>pu fd 150 rt 15
>circulo1
>end
```

Si lo ejecutamos, en un primer momento vemos que no ha habido ningún cambio, porque el círculo se forma normalmente. Sin em-



```
ahora tengo el lapiz px
ahora tengo la goma de borrar
?
```

bargo, una vez que ya ha sido dibujado, veremos como éste va desapareciendo. Cuando se haya esfumado por completo, volverá a dibujarse de nuevo, y así sucesivamente. La diferencia entre ambos programas se encuentra en dos primitivas: px, pu.

La primera de ellas dota a la tortuga de un lápiz especial. Cuando vaya a moverse por un sitio donde no haya nada escrito pintará, pero si en el lugar por donde se mueve ya hay algo dibujado, entonces lo borrará. Por tanto, se dibuja primeramente el círculo y como la pantalla está limpia «deja huella». Cuando comienza la segunda vuelta y el resto de las pares, puesto que ya está presente el trazo anterior, lo borra.

La segunda primitiva, **pu,** ya es conocida: indica a la tortuga que en su movimiento levante el lápiz y no dibuje nada. Por tanto, en el programa el lápiz va cambiando de **px** a **pu.** Probemos a suprimir esta última primitiva. Al comenzar la ejecución del programa, la tortuga toma el lápiz **px** y traza la primera recta, porque la pantalla está vacía, con **bk 100,** pero al regresar al punto de origen, con **fd 100,** borra la recta, con lo cual el círculo nunca se dibuja. Así pues, como no hay mal que por bien no venga, nos hemos hecho con un programa de simulación de un segundero.

Antes de continuar, es necesario advertir que al detener el programa anterior, o el propio círculo1, el lápiz habrá quedado arriba o en modo inverso. Por tanto, si no queremos obtener resultados inesperados en ejemplos posteriores, conviene que lo restauremos a su modo original, ejecutando **pd** en comando directo.

Hemos visto tres lápices distintos que puede utilizar la tortuga. Todavía nos queda por estudiar un cuarto: **pe.** Este, más que lápiz es una goma de borrar *(Pen Eraser,* lápiz borrador), porque por donde pasa él, se borra todo lo escrito. La diferencia con **px** está en que, este último escribe donde no haya nada y borra donde haya algo escrito. Estos dos modos y **pu** son anulados, o vuelven al estado normal, con **pd,** que es como se encuentra la tortuga hasta que se le indica otra cosa. Juguemos con estos cuatro lápices.

```
?to lapiz
>repeat 12 [bk 100 fd 100 rt 30]
>wait 100
>pr [ahora tengo el lapiz px]
>wait 100
>repeat 12 [px bk 50 pu fd 50 rt 15]
>wait 100
>pr [ahora tengo la goma de borrar]
```

```
>wait 100
>pe
>repeat 12 [bk 50 fd 50 rt 15]
>end
```

La primera primitiva desconocida en el programa es **wait.** Produce una parada en la ejecución del programa durante el tiempo indicado a continuación en unidades de 2 centésimas (0'02 segundos). Es muy importante saber que esta primitiva no se haya implementada en el LOGO para PCW. Por otra parte, **pr** no presenta ningún problema y rápidamente habremos adivinado su función: escribir el texto que se encuentra entre corchetes (lo que se llama lista), de *PRint,* cuyo significado en inglés es escribir.

Advirtamos finalmente que los puntos que se quedan sin borrar en el círculo interior son debidos a los errores de precisión gráfica en la pantalla.

VARIABLES

n los ejemplos vistos hasta el momento no es posible realizar transformaciones de una manera sencilla. Para ello, en un programa de LOGO podemos utilizar las variables, cuyo concepto es idéntico al de otros lenguajes.

```
?to variable
>make "lado 100
>repeat 3 [fd :lado rt 120]
>end
```

En la segunda línea, distinguimos una nueva primitiva, **make** (en inglés, hacer), cuya misión es que la variable especificada a continuación tome el valor expresado. En nuestro ejemplo, la variable es lado y almacenará 100. Podemos utilizar el símil de una caja que tenga ese nombre. Con la instrucción **make** introducimos en la caja un papel con el valor 100 escrito en él. Cuando necesitamos utilizarlo, como ocurre al dibujar el triángulo en la tercera línea, nos asomamos al borde de la caja y leemos lo que está escrito en el papel.

Como en el caso del BASIC, una variable no sólo puede almacenar números, sino también letras:

```
? make "nombre "Pedro
```

Así en la variable, o en la caja llamada nombre hemos guardado Pedro. Lo podremos comprobar visualizando el contenido de la variable:

```
?type :nombre
Pedro
```

Por contra al BASIC, LOGO puede utilizar una misma variable para contener indistintamente un valor numérico o de cadena, práctica que se denomina almacenaje alfanumérico.

```
?make "lado "cien
?type :lado
cien
```

Puede darse el caso que tengamos alguna duda sobre si una palabra es nombre de variable o no: puede ser nombre de procedimiento, contenido de la variable... La siguiente primitiva nos lo aclarará:

```
?namep
```

Si la palabra que se indica a continuación es el nombre de una variable, entonces contesta con TRUE (verdad), de no ser así, la respuesta es FALSE (falso).

```
?namep "lado        ?namep "cien
TRUE                FALSE
```

En el primer caso, como la palabra especificada es el nombre de una variable, la contestación es verdad, TRUE, y, puesto que cien no lo es, el ordenador contesta con falso FALSE.

Antes de profundizar en el tema de las variables, conviene hacer algunas aclaraciones a lo explicado hasta el momento. En primer lugar, habremos notado que los nombres de las variables nunca aparecen solos, pues se confundirían con primitivas, sino precedidos por el signo dos puntos (:) o comillas ("), según lo precise su sintaxis.

Así, cuando se reclama el valor de una variable, como sucede al escribirla o utilizarla en un programa, se precede de dos puntos (:) y en caso de hacer referencia a ella misma y no a su contenido, como a la hora de asignarle un valor o averiguar si se trata de una variable **(namep),** le anteceden las comillas.

THING, EN INGLÉS: COSA

Además de con **type,** podemos ver el contenido de una variable gracias a **thing,** especificándose a continuación el nombre de una variable. No obstante, sería absurdo que existieran dos primitivas con idéntica misión. Efectivamente, **thing** averigua el contenido de una

variable, pero de forma diferida. Lo entenderemos mejor con un ejemplo.

Supongamos que tenemos una variable llamada dólar, cuyo contenido alfanumérico es cotización, y este último es a su vez el nombre de otra variable cuyo valor numérico es 130. Esta condición se puede dar si especificamos:

```
?make "dolar "cotizacion
?make "cotizacion 130
```

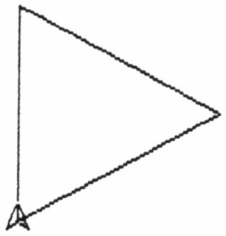
Así, el resultado de **thing :dolar** será averiguar no el valor de esta variable, sino el de una que se llame como el contenido de dólar. En nuestro caso, el resultado será el contenido de cotización, es decir, 130.

```
?thing :dolar
130
```

Ahondando en el funcionamiento de esta primitiva, hemos de saber que no es necesario que el valor final, en nuestro ejemplo el de cotización, sea numérico, sino que puede ser una serie de caracteres. Así, podemos cambiar la voluble cotización del dólar por un lacónico mensaje "alguna":

```
?make "cotización "alguna
?thing :dolar
alguna
```

En lo que sí debemos poner cuidado es en el contenido de la primera variable, el argumento de **thing,** dado que su valor no debe ser



```
?variable 200
?
```

numérico (puesto que, como en BASIC, no existen nombres de variables que sean números). Lógicamente, también debe existir como tal variable, tanto ella misma como el valor que contiene. Si infrinjimos la primera norma, aparecerá el siguiente mensaje:

```
?make "numero 100
?thing "numero
thing doesn't like 100 as input
```

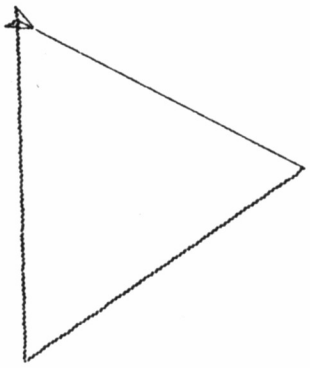
que traducido viene a querer decir que a **thing** no le gusta 100 como argumento. Siendo 100 el valor que hemos dado a la variable en el ejemplo.

De forma similar, si nuestro error consiste en utilizar para cualquiera de las dos variables, es decir, para la denominación de la primera o para el contenido de ésta, nombres inexistentes, recibiremos la cariñosa felicitación por parte de LOGO con un "nombre de la variable *has no value*", o lo que es lo mismo, el nombre de la variable inexistente no tiene valor.

Veamos un ejemplo. Para empezar, cambiaremos el contenido de la variable dólar a una inexistente, en nuestro caso, la llamaremos precisamente así: inexistente.

```
?make "dolar "inexistente
?thing :dolar
inexistente has no value
```

O bien, utilizaremos para el propio argumento de **thing** una varia-



```
?variable 300
?
```

ble inexistente, que para seguir la tradición, volverá a llamarse inexistente.

```
?thing :inexistente
inexistente has no value
```

PROGRAMAS CON PARÁMETROS

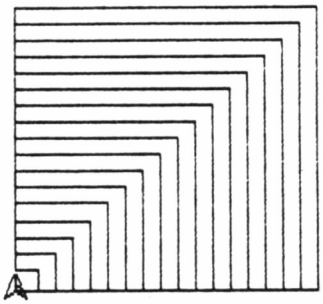
Si nos fijamos en el programa que llamamos variable, no es tan fácil realizar en él las transformaciones que antes indicábamos, ya que sigue siendo imprescindible entrar en edición para ello. Por tanto, el programa variable no es tan variable como parece, valga la aparente redundancia que no es tal. Transformémoslo:

```
?to variable :lado
>repeat 3 [fd :lado rt 120]
>end
```

A continuación del nombre del programa, hemos puesto el nombre de la variable que va a ser utilizada como parámetro. Para indicar que debe comenzar la ejecuciónse escribe:

```
?variable 150
```

El número expresado se introduce directamente en lado, funcionando el programa con ese valor (en este caso 150). Ahora es más fácil realizar variaciones en el programa; sólo debemos cambiar la



```
?cuadrados 5
```

cantidad tras escribir el nombre. Así, para obtener un triángulo menor:

?variable 75

A este tipo de programas, o procedimientos, se les llama procedimientos con parámetros; en el ejemplo, el parámetro es lado. Se pueden tener todos los parámetros que se deseen, con la única precaución de señalar todos sus valores al ejecutar el programa. En el caso de que alguno se nos olvide, LOGO nos contestará con el mensaje de error *not enough inputs to...*, que en traducción literal nos indica que "no hay suficientes entradas" (se refiere a entradas de datos) en el procedimiento indicado seguidamente.

Ampliando el uso de los parámetros, hemos de destacar que es posible emplear con ellos la técnica de recursividad anteriormente estudiada:

```
?to cuadrados :lado
>repeat 4 [fd :lado rt 90]
>cuadrados :lado + 15
>end
```

Lado es el valor inicial que tomarán los lados de los cuadrados que se dibujan. En la tercera línea, se produce la llamada al mismo procedimiento, y, como vemos, es necesario especificar el parámetro, al igual que cuando comenzamos la ejecución del programa. La única diferencia consiste en que éste es aumentado en 15 unidades. Por lo tanto, cada vez que recomienza el programa, lado ha crecido esa cantidad; así, los cuadrados formados serán cada vez más grandes.

EN CASO DE ERROR...

i nos hemos equivocado al teclear un programa o deseamos hacer alguna variante en él, es preciso editarlo. Escribamos:

?ed "cuadrados

En la pantalla veremos cómo, en la zona superior, habitualmente dedicada a los gráficos, se lista el programa solicitado, y en la parte inferior aparece el mensaje «Edith», para recordarnos que estamos en modo de edición. Ahora es posible realizar todos los cambios que consideremos oportunos.

Si estamos trabajando con más de un procedimiento, puede interesanos editar varios de ellos a la vez. A tal fin, utilizaremos la misma primitiva **(ed),** con la variante indicar los distintos nombres entre corchetes. Por ejemplo:

?ed [cuadrados variable]

Por otra parte, en caso de solicitar la edición de un programa no existente, el sistema lo generará por nosotros, aunque lógicamente sin ningún contenido; simplemente la cabecera (nombre) y **end.**

Llegados a este punto, es necesario recurrir a la guía de LOGO de

nuestro ordenador, para saber qué teclas podemos emplear en la corrección de nuestros programas, dado que éstas cambian con cada modelo de AMSTRAD.

Completando las facilidades de edición antes apuntadas, si disponemos de varios procedimientos o programas y los queremos editar todos, no es necesario ir uno por uno, basta con escribir:

?edall

De esta manera *(EDit ALL,* editar todos) entraremos en el modo de edición y tendremos en la pantalla todos los procedimientos creados disponibles para realizar las transformaciones. Y no sólo esto, sino como información adicional, el contenido de las variables.

Con la práctica nos daremos cuenta que es más eficaz escribir los programas en modo de edición directamente, porque así podremos corregir algún error que cometamos al escribirlo. Para ello basta que escribamos **ed** simplemente o seguido del nombre del programa.

LISTAR PARA VER

Si únicamente nos interesa ver el listado del programa y no necesitamos hacer variación alguna, no es necesario entrar en edición, podemos conseguirlo con:

?po "cuadrados

Análogamente, podemos visualizar varios procedimientos a la vez, para lo cual debemos teclear los nombres de éstos entre corchetes:

?po [cuadrados variable]

De modo similar, también podemos listar todos los procedimientos que hayamos definido con las variables utilizadas en ellos, tecleando:

?poall

Si lo que nos interesa son solamente los títulos de todos los procedimientos definidos, escribiremos:

?pots

También podemos ver el nombre de las variables con las que trabajamos y el valor que tienen en ese momento con:

?pons

Todas estas primitivas son útiles cuando estamos programando, ya que constituyen un sistema fácil de comprobar si hemos cometido algún error, porque podremos ver rápidamente el valor de las variables, sus nombres y observar si alguno de ellos es erróneo.

```
to cuadrados :lado
repeat 4 [fd :lado rt 90]
cuadrados :lado + 15
end
```

La última primitiva relacionada con la edición es:

?pops

Gracias a ella, se visualizan todos los procedimientos definidos que tengamos en la memoria. Observemos que es muy similar a **poall,** encontrándose la diferencia en que **pops** no presenta los nombres de las variables que se están manejando.

LIMPIANDO LA MEMORIA

Como era de esperar, LOGO también nos permite limpiar la memoria de aquella información que ya no nos sea de utilidad. Podemos borrar procedimientos y variables, algo aconsejable para dejar el máximo espacio posible al trabajo que estemos realizando. Es importante saber que todo lo que se borre con las primitivas especificadas a continuación ya no es recuperable en manera alguna; por tanto, antes de ejecutar la correspondiente instrucción hay que pensárselo dos veces.

Para borrar un procedimiento concreto se emplea **er,** de *ERase,*

```
?poall
to lapiz
repeat 12 [bk 100 fd 100 rt 30]
pr [ahora tengo el lapiz px]
repeat 12 [px bk 50 pu fd 50 rt 15]
pr [ahora tengo la goma de borrar]
pe
repeat 12 [bk 50 fd 50 rt 15]
end
to variable :lado
bk 50
repeat 3 [fd :lado rt 120]
end
to cuadrados :lado
repeat 4 [fd :lado rt 90]
cuadrados :lado + 15
end
?pots
to lapiz
to variable :lado
to cuadrados :lado
?pons
?
```

borrar. A continuación, y al igual que ocurría con **po,** es necesario especificar el nombre precedido de comillas:

?er "cuadrados

Como ya supondremos, es posible borrar varios procedimientos a la vez. El sistema no es diferente a lo visto hasta ahora.

?er [cuadrados variable]

Si queremos borrar todos los procedimientos, la primitiva que debemos utilizar es **erall,** aunque dado su efecto, debemos poner un especial cuidado en su uso.

Por otra parte, si sabemos que algunas de las variables que hemos estado utilizando hasta ahora ya no nos sirven, es aconsejable borrarlas, puesto que así disponemos de más espacio en la memoria. La primitiva al efecto es **ern.** Debemos especificar a continuación los nombres de las que queremos borrar. Así por ejemplo, podemos eliminar una sola variable...

?ern "lado

O varias a la vez...

?ern [lado longitud]

EN LA PANTALLA

En el programa anterior, cuando las figuras han alcanzado un tamaño considerable, llegan al borde de la pantalla y lo sobrepasan, perdiéndose parte del dibujo. A este modo se le denomina ventana (en inglés, *window),* por la similitud que supone cuando miramos desde el interior de una habitación a través de una ventana: lo que rebasa sus límites ya no lo vemos más.

Sin embargo, podemos trabajar con la pantalla en otros dos modos diferentes. Para ello, disponemos de las primitivas **wrap** y **fence,** que, como todas, pueden ser incorporadas en un programa o directamente, además lógicamente, de **window,** que restaura la pantalla al modo que ya conocemos. Estudiemos los modos restantes, comenzando por **wrap:**

Ejecutemos esta primitiva

```
?wrap
```

y a continuación el siguiente programa:

```
?to pajaritas :lado
>fd :lado rt 120
>fd 2 * : lado lt 120
>fd :lado lt 120
>fd 2 * :lado
>fd 15 rt 120
>pajaritas 15 + :lado
>end
```

Dejemos que el programa se ejecute durante un rato y no lo perdamos de vista cuando la tortuga se acerque al borde de la pantalla. En un primer momento, puede parecer que no hay ninguna diferencia con el modo implícito al comenzar a trabajar con LOGO **(window),** pero veremos como por un lado de la pantalla aparecen rectas, que justamente coinciden con los fragmentos que no han cabido en el lado contrario. Así pues, **wrap** convierte la pantalla en una especie de globo, coincidiendo los bordes superior con el inferior y el izquierdo con el derecho. Por lo tanto, cuando un gráfico se sale por un lado aparece por el contrario.

Por el contrario, **fence,** impone para el borde de la pantalla unos límites rígidos que no se pueden sobrepasar. Lo podremos comprobar con el programa anterior, pero tecleando previamente:

?fence

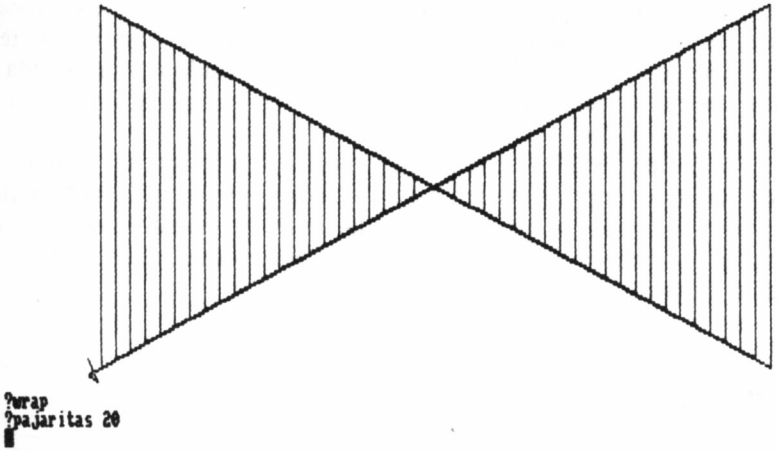
Cuando el gráfico supere los límites de la pantalla, LOGO emitirá el mensaje de error *Turtle out of bounds,* cuyo significado está bien claro (para cualquiera que hable inglés): tortuga fuera de límites. En ese momento, la ejecución del programa se detiene.

Probablemente, si somos observadores, habremos reparado en que durante la ejecución de los programas anteriores, cualquiera que *sea el modo* **(fence, wrap** o **window),** la zona inferior de la pantalla se encuentra ajena al trazado de dibujos, constituyéndose en una especie de pantalla de comunicación con el usuario, a través de la cual se toman datos y hacen ver los mensajes de error. Este sistema de partición de la pantalla se consigue con la primitiva **ss,** abreviatura de **Split Screen,** que en inglés significa literalmente pantalla dividida.

En todo caso, es más que probable que en muchas ocasiones queramos utilizar la pantalla completa para la representación de un dibujo. A tal fin, disponemos de la primitiva **fs,** abreviatura de *Full Screen* (pantalla completa). Podremos constatar su efecto en el programa pajaritas, ejecutando esta primitiva antes del procedimiento.

Al poner en práctica este modo de pantalla, habremos apreciado que lo que se escribe en el teclado a partir de **fs,** no se ve representado en la pantalla. He ahí el principal inconveniente de este modo, que en principio puede ser solapado de una forma muy sencilla: encadenando las primitivas previas a la ejecución del programa.

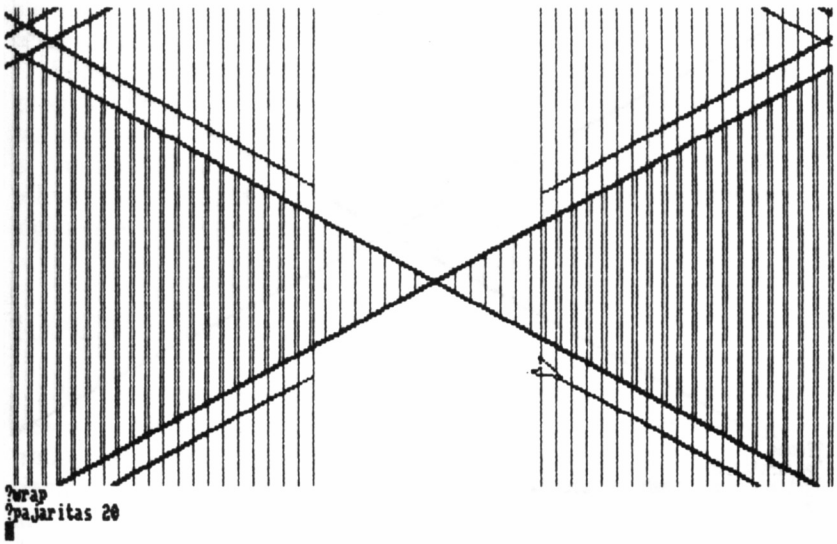
Como ya hemos visto, es posible encadenar varias primitivas en una misma línea, sólo con separarlas por un espacio. Así pues, si por ejemplo queremos borrar la pantalla y ejecutar el programa pajaritas



en modo *full screen,* bastará con introducir el siguiente comando directo.

?fs cs pajaritas 100

Por otra parte, una vez finalizada la ejecución de un procedimiento, existen varias formas de salir del modo de pantalla completa. El primero es la primitiva **ss,** aunque deberá ser tecleada sin verla o añadirla al final de la línea de ejecución en comando directo.



Otro sistema equivalente, aunque con un efecto bastante más drástico, es el propiciado por la primitiva **ts** (Text Screen, pantalla de texto), dado que borra la pantalla de gráficos, pasando a utilizar toda la superficie de visualización para texto, como sucede al entrar en el modo de edición.

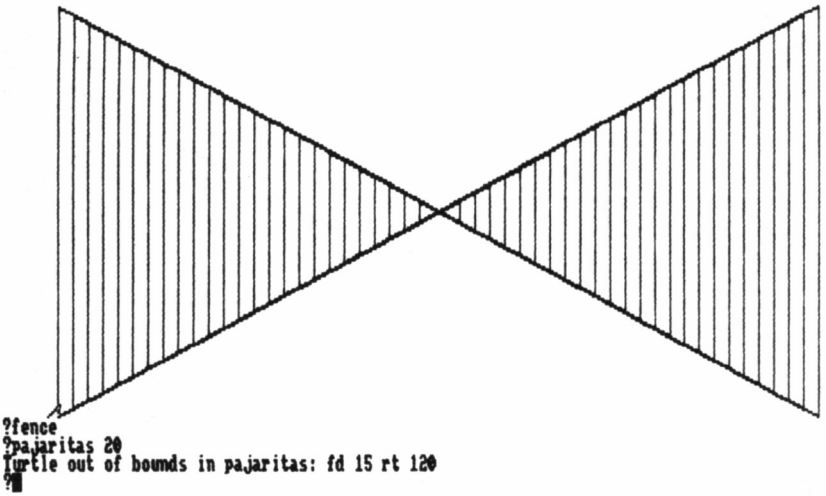
Por último, cualquier mensaje de error o interrupción durante la ejecución de un procedimiento en *full screen,* producirá el automático retorno al modo de pantalla partida, caso que siempre se dará en nuestro programa de ejemplo pajaritas, puesto que sólo se detiene si se le fuerza la interrupción.

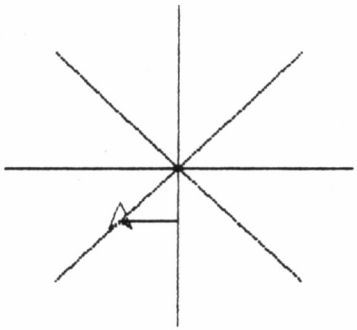
La Tortuga Se Esconde

La tortuga, pese a ser un bichito muy simpático, puede llegar en determinados momentos a estropear la buena presencia de algunos dibujos. Para evitarlo, LOGO pone a nuestra disposición un «manto mágico» que puede hacer totalmente invisible al galápago, aunque sin disminuir con ello sus facultades artísticas. Para esconder la tortuga debemos utilizar:

?ht

abreviatura de *Hide Turtle,* literalmente «esconde tortuga». La acción de esta primitiva es permanente, hasta que no se contrarreste con el





```
?nueva_casa
escribe desplazamiento horizontal 50
escribe desplazamiento vertical 50
regreso al centro
escribe desplazamiento horizontal
```

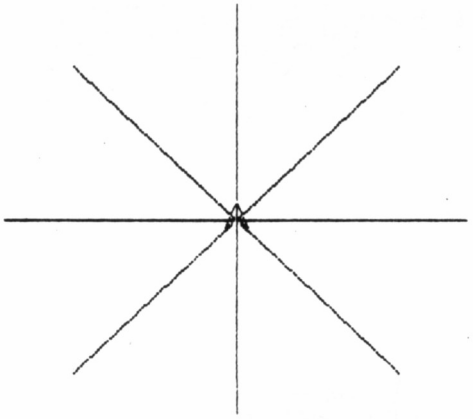
correspondiente antídoto: la primitiva **st,** del inglés *Show Turtle* (muestra tortuga).

Esconder la tortuga no sólo tiene un efecto meramente estético, sino otro quizá más práctico: aumenta considerablemente la velocidad de la misma. A este hecho le podríamos otorgar una romántica explicación, amparada en la coquetería del animalillo, que cuando es visible tiene que guardar su buena imagen ante nosotros y si corre mucho se despeina.

Pero, evidentemente, la explicación real es otra. Cuando la tortuga se hace visible en la pantalla, el ordenador ha de dedicarse a realizar los gráficos del procedimiento y también borrar y volver a dibujar la punta de flecha en su nueva posición. En caso contrario, LOGO sólo emplea su tiempo en dibujar lo indicado en los procedimientos, y lógicamente, su velocidad es mucho mayor.

La Casa De La Tortuga

Posiblemente, si hemos intentado realizar nuestros propios «pinitos» con LOGO, uno de los problemas con que nos habremos encontrado, habrá sido el inconveniente que supone que la tortuga parta siempre en sus movimientos desde el centro de la pantalla (o casi desde el centro). Esto tiene también fácil solución, ya que es posible desplazarla por la pantalla, dándole otro punto de origen. Ahora ve-



```
escribe donde quieres ir: 175 175
te desplazarias los siguientes grados: 45
escribe donde quieres ir: -175 175
te desplazarias los siguientes grados: 315
escribe donde quieres ir:
```

remos todas las primitivas relacionadas con la «casa» de la tortuga y sus posibles mudanzas.

```
?to nueva-casa
>wrap
>type [escribe desplazamiento horizontal\ ]
>setx rq
>type [escribe desplazamiento vertical\ ]
>sety rq
>repeat 8 [fd 150 bk 150 rt 45]
>pr [regreso al centro]
>wait 100
>home
>nueva-casa
>end
```

Sigamos con detenimiento la ejecución de este programa. Para empezar, aparece en la pantalla la solicitud de escribir un valor para el desplazamiento horizontal de la tortuga, gracias a la acción de la primitiva **type,** estudiada anteriormente. Inmediatamente, el programa continúa en la siguiente línea, deteniéndose para tomar un dato del teclado en virtud de la presencia de **rq.**

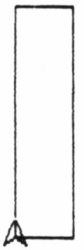
Una vez introducido el dato, vemos cómo la tortuga se desplaza la

cantidad especificada horizontalmente. Análogamente, se opera con el bloque concerniente al movimiento vertical, para finalmente, trazar un dibujo simple y retornar al punto de origen.

Llegados a este punto, conviene recordar que la primitiva **wait,** cuyo único fin es detener momentáneamente la ejecución de un programa, no se encuentra implementada en el LOGO del PCW, motivo por el cual deberá ser suprimida.

De todo el programa, las únicas primitivas desconocidas por el momento son, además de **rq, setx** y **sety,** que se han ocupado de llevar a cabo los desplazamientos de la tortuga, aunque de un modo diferente al que conocemos, **home,** que como su propio nombre indica (**home,** en inglés casa), tiene por fin hacer volver a la tortuga a su casa, es decir, su punto de origen.

El desplazamiento mediante **setx** y **sety** tiene un carácter diferente a los sistemas ya conocidos, porque instituye el nuevo punto de origen de dibujo de la tortuga. Los valores que hay que indicar a continuación son el desplazamiento horizontal, para **setx,** o el vertical, con **sety,** exactamente igual que en un eje de coordenadas cartesianas, que tiene como punto 0,0 el *home.* Esto implica, que valores positivos para **setx** y **sety** propiciarán un desplazamiento de la tortuga hacia la derecha y arriba, respectivamente; y datos negativos, izquierda y abajo.



```
?setscrunch 2
?repeat 4 [fd 50 rt 90]
?
```

Aunque en un programa no es imprescindible utilizar en primer lugar **setx** y a continuación **sety,** sino que se puede alterar el orden e incluso suprimir uno de ellos, lo que si es cierto es que si se van a utilizar ambas primitivas para situar a la tortuga en un nuevo punto de origen, es más cómodo emplear **setpos.**

Esta es equivalente a las dos anteriores, pero reunidas en una. Sitúa el nuevo punto de origen de dibujo, como ya sabemos, pero dando los dos valores al mismo tiempo. Así, si nos interesa que la tortuga se ubique en la parte inferior izquierda de la pantalla, ambos números deben ser negativos. Por ejemplo: **setpos [−100 −50].** Donde el primer dato es el desplazamiento horizontal y el segundo el vertical.

Como ya hemos dicho, en el anterior procedimiento (nueva-casa) existe también otra primitiva nueva: **home,** cuya función es regresar la tortuga al punto central, es decir, su auténtica casa. Es como si en las demás situaciones sólo estuviera «de paso».

Igualmente, se regresa al punto central con una primitiva que ya conocemos, y que además tiene el efecto de borrar la pantalla: **cs.** No obstante, si queremos borrar la pantalla y que la tortuga no se mueva, ya sea en un nuevo punto de origen o al terminar cualquier gráfico, podemos utilizar **clean** (limpiar), cuyo efecto es menos drástico.

```
?estrujar
numero de lados: 5
escribe relacion: 1.5
numero de lados:
```

```
?estrujar
numero de lados: 3
escribe relacion: 2
numero de lados: 
```

GRADOS

Una primitiva a la que posiblemente no veamos utilidad en un primer momento es **towards** (en inglés, hacia). Su función es decirnos el ángulo de giro que tomaría la tortuga si pretendiéramos desplazarla al punto que le especifiquemos. Pero sólo «si pretendiéramos», ya que el movimiento no se produce de forma efectiva.

Tengamos en cuenta que el resultado se expresa respecto a la posición y orientación actual de la tortuga, y no sobre el origen. Veamos un ejemplo:

```
?to ejes
>cs
>repeat 8 [fd 200 bk 200 rt 45]
>hacia
>end

?to hacia
>type [escribe donde quieres ir:\ ]
>make "lugar rl
>dot :lugar
>type [giraria los siguientes grados:\ ]
>pr towards :lugar
>hacia
>end
```

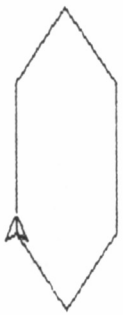
Este programa está constituido por dos procedimientos: ejes y hacia. La misión del primero es dibujar unos ejes de coordenadas que nos servirán para entender mejor la función de **towards,** que se utiliza en el siguiente procedimiento. La primitiva **rl** es una variante de **rq,** que ya vimos anteriormente, y sirve para introducir datos por el teclado. Es imprescindible introducir dos números separados por un espacio, pertenecientes a las coordenadas horizontal y vertical, respectivamente. Una vez otorgados los valores, el contenido es asignado a la variable lugar (ya sabemos: **make "lugar rl).**

A continuación, se dibuja el punto a donde queremos que «mire» la tortuga. Para ello, disponemos de la primitiva **dot,** que sirve de complemento a **setx, sety** y **setpos,** lo cual nos permite elegir entre dos formas de realizar gráficos: utilizar la tortuga, indicándole cuanto debe desplazarse y girar; y acceder directamente a las coordenadas de la pantalla, teniendo en cuenta que el origen se encuentra en el centro de la pantalla y no en una esquina, como ocurre en el BASIC.

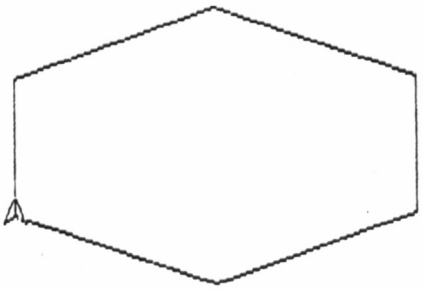
Por último, con un mensaje nos avisa del número de grados que giraría la tortuga.

En caso de querer utilizar **dot** y **towards** con parámetros numéricos, sin variables, es necesario escribirlos entre corchetes:

```
?dot [50 50]
?towards [100 50]
```



```
?estrujar
numero de lados: 6
escribe relacion: 1.3
numero de lados:
```



```
?estrujar
numero de lados: 6
escribe relacion: .3
numero de lados:
```

En relación con **towards,** se presenta una nueva primitiva, **seth.** Esta última efectúa el giro en grados que se indique a continuación de modo absoluto, es decir, sin tener en cuenta la orientación actual de la tortuga y considerando ángulo cero la posición origen de la misma (apuntando en vertical hacia arriba).

RELACION DE ESCALAS HORIZONTAL Y VERTICAL

En todos los dibujos realizados hasta ahora, hemos utilizado líneas rectas en las que se guardaba la misma proporción en los ejes horizontal y vertical. No obstante, LOGO pone a nuestra disposición una herramienta que nos permite alterar dicha relación entre ejes. Así, el dibujo de un cuadrado, puede convertirse en un rectángulo, aún empleando las mismas primitivas y valores para su trazado.

Esta relación de escalas se fija mediante **setscrunch,** cuyo valor inicial es 1. En este estado de cosas, por cada punto que le digamos a la tortuga que avance, realmente se moverá un punto, tanto en horizontal como en vertical. Si, por ejemplo, el valor que asignamos es 2 **(setscrunch 2),** por cada punto de avance que se le imponga a la tortuga, ésta realmente se desplazará dos en vertical, mientras que el movimiento en el eje horizontal no se verá afectado por la primitiva.

Nos podemos imaginar lo que supone dibujar un cuadrado con este tipo de escala (o estrujamiento, que es el significado literal de *scrunch):* mientras que los lados horizontales tendrán su longitud normal, los verticales medirán el doble.

Podremos realizar muchas pruebas con el siguiente procedimiento.

```
?to estrujar
>type [número de lados:\ ]
>make "lados rq
>type [escribe relación:\ ]
>make "relación rq
>setscrunch :relación
>make "ángulo 360 / :lados
>repeat :lados [fd 100 rt :ángulo]
>estrujar
>end
```

En las líneas segunda y tercera indicamos el número de lados que va a tener nuestra figura, en la cuarta y quinta debemos teclear la relación entre las escalas verticales y horizontales, que será tenido en cuenta en la sexta línea. En la séptima se obtienen los ángulos de la figura para que sea regular, realizándose este dibujo en la octava línea. Es aconsejable, por su utilidad, tener en cuenta este procedimiento para dibujar cualquier figura regular.

Con este programa podemos probar los diferentes índices de «estrujamiento» y apreciar los efectos que causan sobre las figuras que se dibujan. Si el índice es menor que 1 y mayor que 0 (LOGO no admitirá valores menores que éste), veremos cómo las figuras se aplanan, avanzando más rápidamente las líneas en sentido horizontal que en vertical.

Un detalle final. Como hemos podido observar en los últimos procedimientos, cada vez que queríamos escribir una frase, la terminábamos con una barra invertida «\». Esta es la forma de escribir el espacio al final de cada frase, ya que en LOGO no se puede hacer directamente, pues el editor lo elimina.

LA TORTUGA TAMBIÉN TRAZA CURVAS

En todo lo que llevamos de libro, hemos estado viendo diferentes gráficos, para conocer las primitivas de LOGO y comprender mejor su funcionamiento. Nos habremos percatado que todos los dibujos han sido trazados con líneas rectas; incluso uno de los primeros procedimientos, que denominamos «círculo», y realizaba una circunferencia, se construía por infinidad de rectas que partían del centro. ¿Dónde están las curvas en este lenguaje?

La respuesta es que DR LOGO carece de la posibilidad directa de dibujar curvas. Un antiguo proverbio reza: «el hambre aguza el ingenio»; si no se pueden realizar curvas directamente, tendremos que construirnos a tal fin nuestros propios procedimientos, que como ya sabemos, pueden ser utilizados posteriormente como una primitiva más.

Estudiemos el problema en teoría. Partimos de la base de conseguir una circunferencia mediante rectas. En la explicación de **setscrunch,** vimos un método para dibujar cualquier figura geométrica regular (líneas 7 y 8). Pues bien, si con él vamos dibujando polígonos

con mayor número de lados, y éstos a su vez, más pequeños, llegaremos por aproximación a la circunferencia perseguida. Por tanto, podemos construir un polígono regular de 360 lados, con giros de la tortuga de 1 grado. En realidad, esto no es una circunferencia, pero nos conduce al efecto óptico deseado.

Mayor precisión obtendríamos con figuras con 720 lados y giros de 0,5 grados, aunque el tiempo invertido en su trazado sería el doble, no justificándose este retraso con el aumento de calidad en el resultado final.

Llevemos nuestra teoría a la práctica:

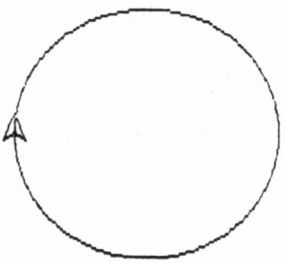
```
?to circunferencia
>type [longitud de los lados:\ ]
>make "longitud rq
>repeat 350 [fd :longitud rt 1]
>end
```

Le pregunta «longitud de los lados» puede resultar chocante en una circunferencia, pero no olvidemos que en el fondo no es sino un polígono regular de 360 lados. Dependiendo del valor que demos a esa longitud, obtendremos diferentes tamaños del gráfico.

Ahora podemos hacer todas las variantes que queramos para sacarle más partido a las curvas. Por ejemplo, dibujar arcos de circunferencia, dando el tamaño (la longitud del lado) y el ángulo.

```
?to arco :longitud :angulo
>repeat :ángulo [fd :longitud rt 1]
>end
```

El proceso es muy simple y no se diferencia mucho de dibujar una circunferencia, que no es más en realidad que un arco de 360 grados.



```
?circunferencia
longitud de los lados: 2
?
```

```
?arco 3 180
?
```

Si nos fijamos bien, la única variante introducida en el proceso es las veces que se repite el dibujo: ya no son 360, sino el valor que determinemos nosotros por medio de «ángulo». Por lo demás, no existe ningún cambio, salvo que es un procedimiento con parámetros y a continuación del nombre se deben introducir los valores de éstos, lo cual no presenta diferencia significativa en cuanto al funcionamiento del programa.

MÁS DIFÍCIL TODAVÍA: ELIPSES

Empecemos practicando:

```
?to elipse
>rt 45
>repeat 2 [repeat 90 [fd 1 rt 1] rt 90]
>end
```

```
?elipse
?
```

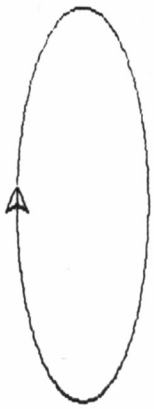
El efecto no está del todo logrado, ya que no se obtiene una auténtica elipse, sino más bien un balón de rugby. Esto es debido a que en realidad se trazan dos arcos de 90 grados unidos, lo que explica esos picos que en la elipse auténtica no existen. Mejores resultados podemos conseguir, si dibujamos una circunferencia con diferente escala horizontal-vertical (**setscrunch**).

Anteriormente aprendimos a evolucionar por la pantalla sin necesidad de utilizar la tortuga, utilizando **setx, sety, setpos** y **dot.** Igualmente, podemos dibujar curvas sin tortuga, aunque el efecto estético es menor, incrementándose asimismo la complejidad del procedimiento al hacer uso de funciones trigonométricas. Por tanto, no se trata de algo muy útil, aunque nos servirá de introducción a la trigonometría.

El siguiente programa traza arcos y, por tanto, también circunferencias; basta con indicar el ángulo deseado. No obstante, en esta ocasión dibuja el ángulo complementario. Así, un arco de 90 grados corresponde al valor 270, la circunferencia completa al cero, etc.

```
?to curva-puntos :angulo
>if :ángulo>360 [stop]
>dot list 90*sin :angulo 90*cos :angulo
>curva-puntos :angulo+5
>end
```

En este programa tenemos tres primitivas nuevas. La primera de



```
?setscrunch 1.5
?circunferencia
longitud de los lados: 1
?
```

```
?curva_puntos 0
?
```

ellas, resultará conocida a cualquiera que haya trabajado en BASIC, Pascal, etc...

if

Cuando en la ejecución de un programa se llega a esta primitiva, se produce una comprobación de lo que viene a continuación (en este caso, si la variable ángulo va a ser mayor que 360). De confirmarse la condición, el programa seguirá su ejecución con las primitivas que se encuentren a continuación de la línea, las cuales deberán estar encerradas entre corchetes. Si la condición no se cumple, se ejecutará la siguiente línea del programa.

Otra de las primitivas recién llegadas es **stop,** cuya misión suponemos no es muy difícil de adivinar. Efectivamente; detiene el programa.

La penúltima línea, contiene las primitivas correspondientes a las funciones trigonométricas: **sin** y **cos,** que, como era de esperar, se limitan a calcular el seno y coseno, respectivamente, del ángulo especificado a continuación. Además, hacemos uso de **list;** diremos por ahora que indica a **dot** que a continuación vienen dos parámetros. Es en algunos casos equivalente a los corchetes y por tanto, en tales circunstancias, sustituible.

```
?curva_puntos 90
?
```

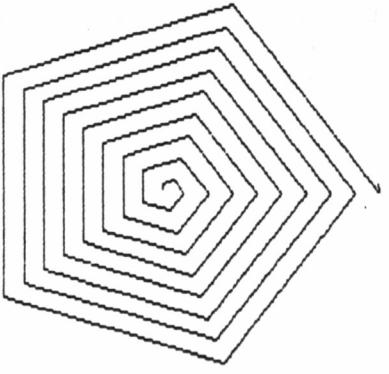
```
?seno -360
```

Aprovechando las funciones trigonométricas, podemos construir programas que realicen dibujos originales, como puede ser el siguiente:

```
?to seno :angulo
>dot list :angulo 50*sin :angulo
>coseno :angulo
>end

?to coseno :angulo
>dot list :angulo 50*cos :angulo
>seno :angulo+5
>end
```

Estos dos procedimientos se llaman el uno al otro; podríamos hablar de una recursividad recíproca, porque el primero llama al segundo y viceversa, constituyendo un proceso sin fin, o más concretamente, hasta que se rebasa el *stack* de LOGO. Es aconsejable probar el procedimiento con el valor de **«ángulo» −360.**



```
?espiral_recta 5 5
```

ESPIRALES

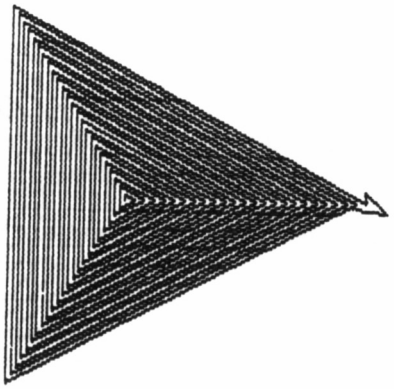
Volviendo con nuestra amiga la tortuga, podemos construir diferentes tipos de curvas y combinarlos para conseguir nuevos efectos; por ejemplo, definir espirales. El mecanismo de dibujo de una espiral, podemos verlo en funcionamiento en el siguiente procedimiento, que aunque sin generar espirales curvas, nos sirve para extraer el proceso general:

```
?to espiral-recta :lados :longitud
>make "angulo 360/:lados
>fd :longitud rt :angulo
>espiral-recta :lados :longitud+5
>end
```

La tercera línea dibuja el lado y da el ángulo de giro para trazar el siguiente, recomenzando el programa con un valor de longitud aumentada en 5 unidades; si cambiamos este valor, obtendremos diferentes tipos de espirales, más abiertas o cerradas.

Similar proceso tenemos que seguir con las espirales curvas: ir aumentando el valor del lado que se está dibujando. El programa es el siguiente:

```
?to espiral-curva :long
>repeat 30 [fd :long rt 1]
>espiral-curva :long+0.002
>end
```



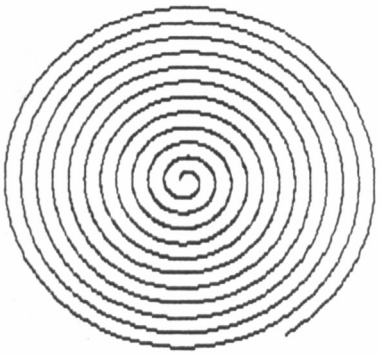
```
?espiral_recta 3 5
```

Para ejecutarlo, es aconsejable dar un valor inicial de «long» bastante bajo, como puede ser 0.001 (equivalente a .001), 0.1 en el PCW; para este mismo ordenador, debemos cambiar el valor 0.002 de la tercera línea por 0.02. El arco de 30 grados que se traza en la segunda línea es el equivalente al dibujo del lado en las espirales rectas, tras el cual aumenta la longitud, en una escala muy pequeña, para evitar que rápidamente se salga de la pantalla y sólo podamos ver una pequeña fracción del dibujo.

Complicamos más los gráficos si dibujamos algo similar a las espirales, pero utilizando no lados, sino figuras:

```
?to espiral-figura :long :lado
?make "angulo 360/:lado
>repeat :lado [fd :long rt :angulo]
>rt 15
>espiral-figura :long+10 :lado
>end
```

Una vez que comienza la ejecución del procedimiento, dando la longitud inicial de la figura y el número de lados que va a tener, se dibuja ésta (tercera línea de programa), se le da un giro a la tortuga y el programa vuelve a comenzar, pero con el valor de la longitud de los lados aumentada, dibujándose nuevamente otra figura más grande, y girada con respecto a la anterior, consiguiéndose así el efecto de una complicada y original espiral.



```
?espiral_curva 0.1
```

LOS COLORES DE LA TORTUGA

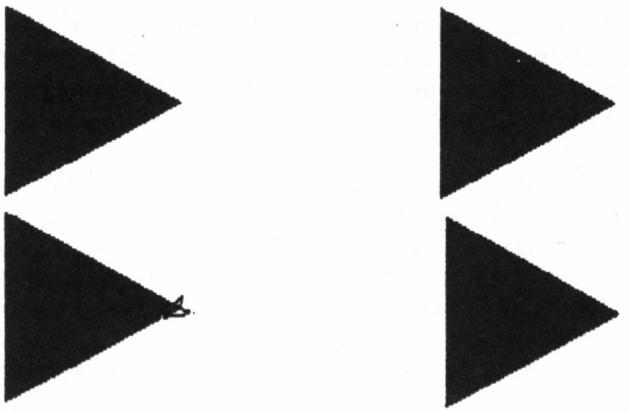
Pasamos a regalarle unos lápices de color a nuestra tortuga, pero no sin antes hacer una advertencia previa. Los que dispongan de un ordenador CPC con monitor de fósforo verde no verán, evidentemente, dichos colores (salvo derroche de imaginación), sino diferentes escalas de verde; no obstante, las primitivas, en cualquier caso, son totalmente aplicables. Por otra parte, para los usuarios del PCW, se indicarán las posibilidades de color de su versión, bastante mermadas con respecto a los otros modelos.

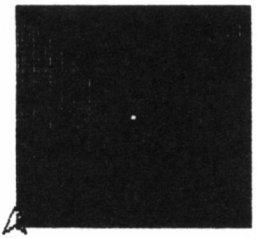
El Dr. LOGO de los CPC tiene posibilidad de manejar cuatro colores al mismo tiempo sobre la pantalla. La forma de trabajar es como si la tortuga llevara a sus espaldas cuatro tinteros, que en un principio están rellenos de los siguientes colores: azul, blanco, azul claro y rojo, aunque su contenido puede ser alterado, como más adelante veremos.

Para empezar, el siguiente procedimiento muestra los cuatro colores que trae la tortuga al comenzar a trabajar. Cuando ejecutemos el programa, aparentemente no sucederá nada, situación que se prolon-

gará durante un rato (sobre todo si tenemos la tortuga escondida, **ht**). Esto es debido a que estará dibujando una figura con el mismo color del fondo y por tanto ésta no será visible.

```
?to colores
>fs cs
>;izquierda arriba
>pu setpos [-190 95] pd
>setpc 0
>make "l 5
>repeat 170 [fd :l rt 120 make "l :l+1]
>;derecha arriba
>pu setpos [190 95] pd
>setpc 1
>make "l 5
>repeat 170 [fd :l rt 120 make "l :l+1]
>;derecha abajo
>pu setpos [190 -95] pd
>setpc 2
>make "l 5
>repeat 170 [fd :l rt 120 make "l :l+1]
>;izquierda abajo
>pu setpos [-190 -95] pd
>setpc 3
>make "l 5
>repeat 170 [fd :l rt 120 make "l :l+1]
>end
```





```
?mezclando
escribe combinacion: 0 2 0
escribe combinacion:
```

En el programa nos encontramos como novedades, en primer lugar, el signo punto y coma (;) seguido de una frase. Al ejecutarlo, nos habremos percatado que dichas palabras no aparecen en ningún momento durante la ejecución del programa. Efectivamente, cuando LOGO se encuentra con este símbolo, ignora lo que viene a continuación. La finalidad, simplemente de cara al programador, es introducir un comentario para entender el programa y dividirlo en bloques, como un REM en BASIC.

Otra novedad es **setpc,** responsable de que la tortuga cambie de color. Como dijimos, existen cuatro colores que se numeran de 0 a 3, señalando a continuación de la primitiva cual de ellos se va a utilizar. Este es el único comando de color que utilizan los PCW en LOGO, admitiendo sólo los valores 0 (mismo color que el fondo) y 1.

```
?abanico
escribe combinacion: 1 1 0
escribe combinacion: 2 2 0
escribe combinacion: 0 0 2
para empezar de nuevo aprieta "s" x
```

El color del fondo de la pantalla, lo que se conoce como «papel», también puede ser alterado por un método similar: **setbg.**

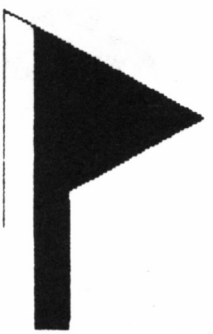
MÁS COLORES

Como ya anticipamos al comienzo de este capítulo, aunque nuestra tortuga adopte cuatro colores iniciales, es posible crear otros nuevos. Para ello, debemos comprender previamente el sistema de pigmentación seguido por el simpático animalillo.

El color de cada tintero, está formado por una mezcla de otros tres colores básicos: rojo, verde y azul, respectivamente, en una proporción determinada:

tintero 0	azul	combinación: 0-0-1
tintero 1	blanco	combinación: 2-2-2
tintero 2	azul claro	combinación: 0-2-2
tintero 3	rojo	combinación: 2-0-0

Así, la combinación de colores del tintero 0 (azul), posee una parte de azul y ninguna de rojo y verde. El 1 (blanco) está constituido por dos partes de cada color, rojo, verde y azul. El 2 (azul claro) utiliza en su mezcla dos partes de azul y otras dos de verde. Finalmente, el 3 está formado exclusivamente por rojo, sin incluir nada de verde y de azul.



```
?abierto
```

A lo que nosotros hemos llamado tintero, LOGO le denomina «paleta». Así, para mostrarnos la anterior combinación ejecutaremos:

?pal 0
?pal 1
?pal 2
?pal 3

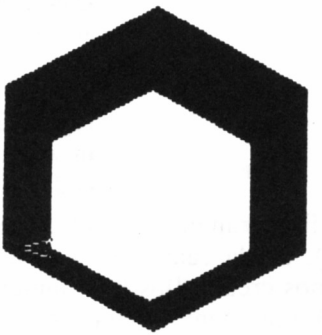
Obteniendo como respuesta, seguidamente a cada una de las anteriores primitivas:

[0 0 1]
[2 2 2]
[0 2 2]
[2 0 0]

Como podemos comprobar, los resultados coinciden con la combinación que expusimos anteriormente. Basándonos en este sistema de paletas de colores, podemos alterar los colores que la tortuga adopta por defecto. Así, por ejemplo:

?setpal 2 [0 2 0]

quiere decir que en la paleta 2, donde antes teníamos el azul claro, ahora tenemos un nuevo color, formado por verde en dos partes sin nada de rojo ni de azul. Evidentemente, hemos obtenido el verde, nuevo color que la tortuga no soportaba originalmente. Por otra parte, no es posible añadir más de dos partes de un color en una mezcla con **setpal.**



?rellenar
?

Gracias a este sistema, podemos variar la combinación de colores de las cuatro paletas, alcanzando teóricamente un total de 27 colores diferentes, aunque en la práctica algunas combinaciones se parecen tanto que apenas se distinguen. Estas 27 combinaciones son las siguientes (hemos marcado con un asterisco las adoptadas por defecto):

0-0-0	0-0-1*	0-0-2
0-1-0	0-1-1	0-1-2
0-2-0	0-2-1	0-2-2*
1-0-0	1-0-1	1-0-2
1-1-0	1-1-1	1-1-2
1-2-0	1-2-1	1-2-2
2-0-0*	2-0-1	2-0-2
2-1-0	2-1-1	2-1-2
2-2-0	2-2-1	2-2-2*

PRUEBAS DE COLOR

Podemos hacer pruebas de los diferentes colores, utilizando el siguiente procedimiento:

```
?to mezclando
>setpc 2
>type [escribe combinacion:\ ]
>setpal 2 rl
>cs
>make " l 5
>repeat 200 [fd :l rt 90 make " l :l+1]
>mezclando
>end
```

Primeramente, se selecciona la segunda paleta (línea 2 del programa). En la cuarta línea se hace efectiva la nueva combinación que hemos creado (los tres números deben estar separados por un espacio); recordemos que **rl** es un método de introducción de datos con el teclado. En la línea 7, se dibuja la figura con la segunda paleta y la combinación de colores introducida por nosotros.

Por otro lado, los colores del texto también pueden ser cambiados, aunque alterando la mezcla de la paleta 1, dado que éste siempre se escribe con el color de dicha paleta. Por tanto, a no ser que la tortuga dibuje con la paleta 1, el color utilizado para escribir texto es independiente del utilizado para gráficos. Podremos comprobarlo

con el anterior programa, cambiando en la segunda línea (**setpc 2**) por **setpc 1** y la cuarta (**setpal 2 rl**) por **setpal 1 rl.**

LOGO pone a nuestra disposición más herramientas. En caso que precisemos averiguar qué color tiene un punto concreto de la pantalla gráfica, tendremos que indicar las coordenadas del mismo, entre corchetes y separadas por un espacio, a la primitiva **dotc,** que responderá con el número de tintero utilizado.

Hagamos más pruebas de color. Con el siguiente programa podemos comparar diferentes combinaciones entre sí, hasta un máximo de tres.

```
?to abanico
>setbg 0
>cs
>type [escribe combinacion:\ ]
>setpal 1 rl
>setpc 1
>pu setpos [-200 50] pd
>lt 90
>repeat 180 [fd 70 bk 70 rt 1]
>type [escribe combinacion:\ ]
>setpal 2 rl
>setpc 2
>pu setpos [0 50] pd
>repeat 180 [fd 70 bk 70 rt 1]
>type [escribe combinacion\ ]
>setpal 3 rl
>setpc 3
>pu setpos [200 50] pd
>repeat 180 [fd 70 bk 70 rt 1]
>type [para empezar de nuevo pulsa «s»\ ]
>if rq="s [abanico]
>end
```

Para abandonar la ejecución del programa, basta con teclear algo diferente a «s», cuando nos lo pregunte en las tres últimas líneas. Por contra, mientras contestemos con «s», la condición de la penúltima línea es cierta y por tanto el programa se vuelve a ejecutar, como si fuera recursivo.

Otra primitiva de interés para el tema que nos ocupa es **fill,** que permite rellenar de color una determinada área, aunque sólo está implementada en el LOGO de PCW (¡alguna ventaja debía tener!), por tanto, no se acompaña de ningún parámetro. El sistema de rellenado

consiste en que la tortuga va cambiando el color de todos los puntos contiguos, hasta que se encuentra con otro de su mismo color. Así pues, si queremos rellenar una figura geométrica, ésta no puede estar en un lado de la pantalla. Análogamente, podemos suponer qué es lo que pasará si rellenamos una figura abierta:

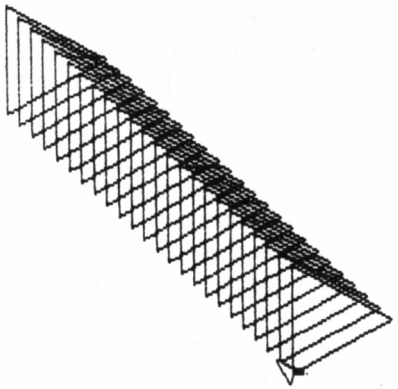
```
?to abierto
>repeat 2 [fd 200 rt 120]
>fd 150 bk 150
>pu rt 30 fd 3 pd
>fill
>end
```

Como se rellenan los puntos contiguos hasta que se encuentran los del mismo color, éstos no aparecen y por tanto termina toda la pantalla «manchada».

Lo habitual, pues, es colorear alguna figura, o parte de ella, cerrada. Sirva como botón de muestra:

```
?to rellenar
>cs
>repeat 6 [fd 150 rt 60]
>pu rt 90 fd 40 rt 90 pd
>repeat 6 [lt 60 fd 100]
>pu rt 90 fd 6 pd
>fill
>end
```

La línea que se inicia con **pu rt 90...** es la que mueve la tortuga a una zona inter-líneas, para que sea posible el **fill.**



EXAMINAMOS LA TORTUGA Y EL PAPEL

En los gráficos ensayados hasta el momento, hemos visto diferentes estados de la tortuga, como puede ser si estaba o no escondida, su color, tipo de lápiz (**pd, px, pu, pe**), e igualmente de la pantalla: si toda era para gráficos, su color, modo (**wrap, window** o **fence**)... Es posible visualizar todos estos datos y algunos más, por si necesitamos saber la situación actual de la tortuga y la pantalla.

En el primer caso, utilizaremos **tf,** lo cual generará un mensaje del tipo...

[30 59 90 PX 1 TRUE]

Su contenido podrá cambiar, pero siempre serán seis parámetros, que por orden nos indicarán:

— COORDENADA X: Situación de la tortuga según el eje horizontal. Se modifica con **setx** o con **setpos.**

— COORDENADA Y: Situación de la tortuga según el eje vertical. Se modifica con **sety** o con **setpos.**

— DIRECCION: Es el ángulo en que se encuentra girada la tortuga. Se modifica con las primitivas de giro: **rt, lt, seth.**

— TIPO DE LAPIZ: Indica si la tortuga está utilizando **pd, pu, px** o **pe.**

— PALETA DE COLOR: Señala qué paleta está utilizando la tortuga. Se modifica con **setpc.**

— VISIBILIDAD DE TORTUGA: Cuando aparezca TRUE, indica que la tortuga es visible. En caso contrario, el mensaje será FALSE. Las primitivas que alteran su estado son **st** y **ht.**

Si la información que precisamos atañe a la pantalla, deberemos ejecutar **sf,** cuyo resultado será del tipo...

[0 SS 7 WINDOW 1]

Cada parámetro nos indicará por orden, en este caso:

— PALETA DE COLOR: El número de paleta que se está utilizando para el color del fondo. Se altera con **setbg** seguido de **cs.**

— TIPO DE PANTALLA: Indica si estamos trabajando con la pantalla mixta, que se define con **ss,** sólo para textos, con **ts,** o para gráficos exclusivamente **(fs).**

— LINEAS DE TEXTO: Es el número de líneas definidas para texto en la pantalla mixta (**ss**). Se altera con **setsplit** seguido del valor deseado, como por ejemplo, **setsplit 13.**

— LIMITES DE PANTALLA: Nos señala si la pantalla se encuentra en alguno de los tres modos siguientes: **fence, wrap** o **window.**

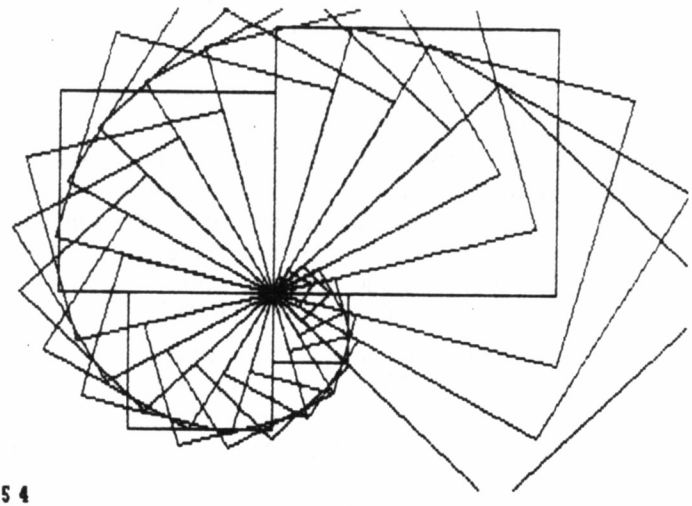
— RELACION DE ESCALAS: Es el nivel de «estrujamiento» o

la relación entre las coordenadas verticales y horizontales, afectado por **setscrunch.**

Podemos poner a prueba ambas primitivas con el siguiente procedimiento. Si cortamos su ejecución en cualquier momento y a continuación tecleamos las primitivas anteriores, veremos el estado del fondo y de la tortuga.

```
?to color :paleta
>fs wrap
>if :paleta>3 [make "paleta 0]
>setpc :paleta
>setbg 3-:paleta cs
>repeat 3 [fd 100 rt 120]
>color :paleta+1
>end
```

Una variante interesante de este programa, se obtiene eliminando la quinta línea e insertando entre la sexta y la séptima: **pu lt 45 bk 15 rt 45 pd.**



```
?espiral_figura 5 4
```

OTRA DEFINICIÓN DE PROCEDIMIENTOS

hora que hemos cogido práctica, creemos que conviene profundizar sobre algunos aspectos del funcionamiento del LOGO, como puede ser una nueva forma de definir procedimientos.

Hasta ahora hemos visto como se hacía a partir de **to,** pero también es posible gracias a **define.**

Aunque no lo aparente, esta palabra es inglesa, y su significado no es difícil de deducir: definir. Tras escribirla y SIN pulsar RETURN o ENTER, se teclea el nuevo procedimiento, comenzando por el nombre y seguido de una apertura de corchete ([), de la siguiente manera:

define "rectangulo [[] [fd 300 rt 90 fd 150 rt 90 fd 300 rt 90 fd 150 rt 90]]

Cuando el procedimiento haya terminado de escribirse, es el momento de pulsar RETURN (o ENTER). Desde luego, un programa escrito de esta manera presenta la principal característica de su dificultad en interpretación. Esta pega puede ser obviada de diferentes maneras: o bien separando por espacios las diferentes primitivas...

define "rectangulo [[] [fd 300 rt 90 fd 150 rt 90 fd 30 rt 90 fd 150 rt 90]]

O bien separando con corchetes las primitivas o grupos de ellas:
define "rectangulo [[] [fd 300 rt 90] [fd 150 rt 90] [fd 300 rt 90] [fd 150 rt 90]]

Esta segunda alternativa tiene más consecuencias que las meramente estéticas o aclaratorias, ya que para LOGO cada conjunto de comandos limitados por corchetes forma una línea de programa. Posiblemente sea algo ilógico hablar de líneas, si todo el procedimiento se escribe en una única, como un todo continuo, sin embargo, en el momento de editar o de listar, el procedimiento toma la forma clásica que ya conocemos. Así, la primera presentación del procedimiento «rectangulo» al editarlo (con **ed "rectangulo**) o listarlo (con **po "rectangulo**) sería:

```
to rectangulo
fd 100 rt 90 fd 50 rt 90 fd 100 rt 90 fd 50 rt 90
end
```

Por el contrario siguiendo el segundo sistema:

```
to rectangulo
fd 100 rt 90
fd 50 rt 90
fd 100 rt 90
fd 50 rt 90
end
```

LA SINTAXIS DE DEFINE

Si observamos la definición con **define,** apreciaremos a continuación tres corchetes juntos, de la forma «[[]». El primero de ellos es el que indica el comienzo de la definición de procedimientos, los otros dos sirven para encerrar los parámetros que se vayan a utilizar en el programa. Así, como variante de «rectángulo», con parámetros se definiría:

define " rectangulo [[lado-mayor lado-menor] [fd :lado-mayor rt 90] [fd :lado-menor rt 90] [fd :lado-mayor rt 90] [fd :lado-menor rt 90]]

Como podemos comprobar, no es necesario escribir antes de dichos parámetros los dos puntos (:), porque con este método ya tienen su lugar asignado: los primeros corchetes después del nombre.

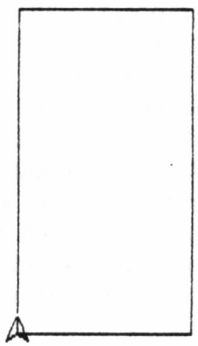
Complementando la acción de **define,** un procedimiento escrito por el sistema «clásico», puede ser visualizado en una línea, gracias a

text, que deberá ir seguido del nombre del programa a mostrar. Veamos un ejemplo:

```
?to cuadrado
>repeat 2 [fd 50 rt 90]
>repeat 2 [fd 50 rt 90]
>end
cuadrado defined
?text "cuadrado
[[] [repeat 2 [fd 50 rt 90]] [repeat 2 [fd 50 rt 90]]]
```

¿Cuándo es conveniente definir un procedimiento por uno u otro método? En el fondo, como hemos demostrado, da exactamente igual; lo único que puede inclinar la balanza es la claridad de lo escrito, como ya dijimos, y la posibilidad de corregir un programa que se esté escribiendo.

Si escribimos un programa con **to-end** y tras entrar una línea nos percatamos de un error cometido en otra anterior, éste ya no es posible corregirlo hasta que no se termine de escribir, pasando a edición y solucionando el problema. Esto no ocurre con **define,** porque en cualquier momento es posible volver atrás para corregir una equivocación. Sin embargo, si antes de empezar a escribir cualquier procedimiento, entramos directamente en edición, podremos teclearlo y cualquier error cometido en alguna línea anterior podrá ser subsanado, con la ventaja de la claridad en la escritura.



```
?define "rectangulo [[] [fd 300 rt 90 fd 150 rt 90 fd 300 rt 90 fd 150 rt 90]]
?rectangulo
?
```

Así pues, es aconsejable utilizar **define** en aquellos procedimientos cortos o en los que vayamos improvisando directamente sobre nuestro ordenador, sin tener el programa escrito previamente en un papel. El método **to-end** se utilizaría en los casos restantes, aunque siempre es bastante conveniente escribirlo directamente sobre el editor (entrando en edición con ed "...), para poder solucionar cualquier eventualidad en algún punto del programa.

La Memoria De Nuestra Tortuga

Vamos ahora a examinar la capacidad de trabajo de nuestro galápago, echando un vistazo a su interior, exactamente a su cabeza. En realidad, será al AMSTRAD al que examinaremos, dando unas nociones de cómo maneja Dr. LOGO la memoria a su disposición.

Es una costumbre en LOGO medir la memoria disponible, no en Kbytes como era de esperar, sino en «nodos» *(nodes,* en inglés). Siempre es posible averiguar cuanta memoria queda libre. No tenemos más que escribir:

?nodes

Inmediatamente se visualizará un número, que será la memoria disponible. Naturalmente, esta cifra variará según si tenemos algún procedimiento cargado, alguna variable definida o el tiempo que haya estado ejecutándose un programa, dado que cuanto más tiempo esté trabajando un procedimiento, más memoria consumirá, incluso aquellos que no tengan cálculos matemáticos. Este hecho se acentúa notablemente en los programas recursivos.

Se puede pensar, por tanto, y muy acertadamente, que un programa recursivo sin fin en incansable ejecución, acabaría llenando toda la memoria disponible, produciéndose un inevitable bloqueo. Efectivamente, así sería si no llega a ser porque Dr. LOGO dispone de un «sistema automático» de liberación de memoria. Es decir, en un momento determinado, se autoexamina y todos aquellos nodos que están ocupados de forma inútil son borrados y puestos a disposición del programa.

Este sistema, que no sólo se da en este lenguaje, recibe la denominación genérica de *garbage collection* (recogida de basuras). Nosotros mismos podemos apreciar su actuación, si al ejecutar un procedimiento con recursividad le dejamos el suficiente tiempo. Lo notaremos porque el programa que se ejecuta normalmente, de repente se detiene, sin ningún motivo aparente y poco después continúa su proceso,

como si nada hubiera ocurrido. En esta detención se ha producido la liberación de los nodos ocupados.

MANEJANDO LA MEMORIA

Juguemos un poco con la memoria, aprovechando el siguiente programa:

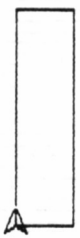
```
?to memoria :longitud
>;comprobacion de memoria libre
>ht wrap
>fd :longitud rt 90
>type nodes type [...]
>memoria :longitud+5
>end
```

Una vez cargado, preguntamos por la memoria libre...

```
?nodes
3665
```

Esta ha disminuido algo respecto a la inicial y no es de extrañar, porque nuestro programa ocupa un lugar en la memoria.

Al ejecutarlo, veremos cómo al mismo tiempo que se forma una «espiral-cuadrada», aparece en la porción inferior de la pantalla la



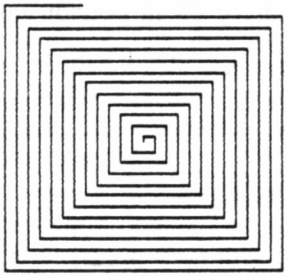
```
?text "rectangulo
[[la_mayor la_menor] [fd :la_mayor rt 90 fd :la_menor rt 90 fd :la_mayor rt 90 fd :la_men!
or rt 90]]
?rectangulo 200 50
?
```

memoria que va quedando disponible, que inevitablemente, va disminuyendo. Tengamos paciencia y dejemos que el programa siga su curso. Cuando aproximadamente queden 300 nodos, se producirá la «recogida de basura», con el efecto de elevar considerablemente la cantidad de memoria libre.

Por otra parte, además de LOGO, nosotros mismos podemos forzar la acción del «basurero», con la primitiva **recycle.** Constataremos, igualmente, que se detiene el proceso. Si ahora ejecutamos **nodes,** certificaremos que tal liberación se ha producido.

AHORRANDO MEMORIA

Para ahorrar memoria, en procedimientos largos o complicados (como pueden ser excesivos cálculos), además de utilizar **recycle,** deberemos evitar los comentarios, dado que también ocupan memoria, aunque LOGO no los ejecute. En caso de que los hayamos escrito ya, **noformat** los elimina. Así, si examinamos la cantidad de memoria disponible inmediatamente antes y después de la ejecución de esta primitiva, apreciaremos su efecto.



```
?memoria 5
376...365...354...343...332...321...310...299...288...277...266...255...244...233...222...
.211...200...3181...3170...3159...3148...3137...3126...3115...3104...3093...3082...3071...
.3060...3049...3038...3027...3016...3005...2994...2983...2972...2961...2950...2939...2928
...2917...2906...2895...2884...2873...2862...2851...2840...
```

PALABRAS Y LISTAS

Hemos abandonado ya a la tortuga en sus evoluciones gráficas por la pantalla y ahora nos disponemos a estudiar otro de los puntos fuertes del LOGO: el tratamiento de palabras o texto. Aunque es una vertiente totalmente diferente a los gráficos de tortuga, su comprensión no es en principio difícil, aunque se pueden llegar a efectuar operaciones auténticamente complejas cuando se domina. En este aspecto, LOGO se basa, principalmente, en el LISP, lenguaje del cual procede, perteneciente al campo llamado de «inteligencia artificial».

¿QUE ES UNA PALABRA Y UNA LISTA?

La estructura más sencilla de las dos es la palabra. Este concepto coincide totalmente con el que nosotros manejamos en la vida normal; así, una palabra es:

"mesa
"teclado
"castillo
"12345
"abc1-2f
"rio-revuelto

La única característica que las distingue, es que han de ir precedidas de comillas y no deben contener ningún espacio de separación. Así por ejemplo: "río revuelto no sería una palabra LOGO.

Por otra parte, una lista es un conjunto de palabras o de otras listas, que, a su vez, pueden constituirse por más palabras y más listas. A diferencia de las anteriores, no se preceden por comillas, sino que están acotadas con corchetes ([]). Ya las hemos visto en programas anteriores, cuando teníamos que escribir algún mensaje. He aquí varios ejemplos de listas:

```
[escribe numero lados]
[1 2 3 uno dos tres]
[azul rojo [bermellon burdeos] verde]
```

Las palabras dentro de ellas deben estar separadas por espacio y no llevan las comillas precediéndolas. Si nos fijamos en esta última lista, vemos que está formada por tres palabras («azul rojo verde») y una nueva lista ([bermellon burdeos]), que a su vez contiene dos palabras. Para que en un primer momento nos sea más fácil, las listas podemos entenderlas como equivalentes a las oraciones o frases, que igualmente están formadas por palabras.

PALABRAS Y LISTAS QUE SE UNEN

También es posible crear palabras o listas a partir de otras, para lo cual es preciso recurrir a la primitiva **word** (en ingles, palabra). Su efecto es tomar dos palabras para formar una tercera, unión de las anteriores. Así:

```
?word "para "aguas
paraaguas
```

Para un trabajo muy similar al anterior, aunque de aplicación en las listas, LOGO dispone de dos primitivas:

list une dos elementos (ya sea palabra o lista) formando una tercera, resultante de dicha unión. Si los elementos a fusionar son listas, los corchetes originales se conservan.

Así por ejemplo, para unir palabras:

```
?list "rojo "negro
[rojo negro]
```

Si se aplica sobre listas...

```
?list [rojo negro] [amarillo azul]
[[rojo negro] [amarillo azul]]
```

Como vemos, se ha formado una nueva lista integrada por otras, ya que se han conservado los corchetes originales. Análogamente, el resultado obtenido aplicando la primitiva sobre una palabra y una lista...

?list "colores [amarillo azul]
[colores [amarillo azul]]

Otra nueva primitiva, **se,** lleva a cabo una tarea muy similar a la de **list,** aunque eliminando los corchetes originales:

?se [rojo negro] [amarillo azul]
[rojo negro amarillo azul]

?se "colores [amarillo azul]
[colores amarillo azul]

¿LISTAS O PALABRAS?

Para los casos de duda en que no sepamos si estamos trabajando con una lista o con una palabra, Dr. LOGO nos ofrece una serie de primitivas que «preguntan» las características del conjunto a investigar. Estas son **worp** y **listp.**

La primera de ellas *(WOED Property,* propiedad de palabra) averigua si el conjunto en cuestión es una palabra. En caso afirmativo, responde TRUE, y de no ser así, FALSE.

hemos acabado en: 18 25
?

```
?wordp "esto-es-una-palabra
TRUE
?wordp [esto no es una palabra]
FALSE
```

La otra primitiva *(LIST Property,* propiedad de lista) actúa de forma similar, aunque refiriéndose a una lista. Así pues, el resultado de la aplicación de ambas primitivas sobre un mismo objeto de estudio siempre será complementario.

```
?list [esto es una lista]
TRUE
?listp "esto-no-es-una-lista
FALSE
```

Evidentemente, en los ejemplos expuestos hasta el momento, no existe ninguna duda sobre si tratamos con listas o palabras; el problema se puede presentar en caso de trabajar con variables, cuyo contenido real desconocemos a priori.

```
make "flores [rosa clavel]
?listp :flores
TRUE
?wordp :flores
FALSE
make "numeros "1-2-3-4
?listp :numeros
FALSE
?wordp :numeros
TRUE
```

Al utilizar con las variables las primitivas que acabamos de estudiar, hemos de tener en cuenta que debemos preguntar por el contenido de ellas y, por tanto, se han de preceder por dos puntos (:); si sustituyéramos éstos por comillas ("), **wordp** y **listp** preguntarían sobre la propia palabra y no por su contenido.

ESCRIBIR EN LA PANTALLA

De anteriores programas ya conocemos algunas formas de escribir mensajes en la pantalla. Repasemos todas ellas, estudiando sus diferencias. Las tres primitivas relacionadas con esta función son **pr, type** y **show.**

La primera de ellas, tras escribir el mensaje, efectúa lo que se conoce como un «retorno de carro», es decir, un salto a la línea siguiente, de tal forma que los próximos mensajes a escribir no aparecerán en la misma línea:

?pr "lineas pr "distintas

El resultado obtenido será:

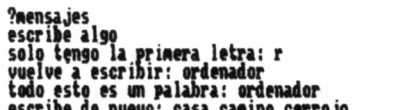
lineas
distintas

Con **type** no se produce el retorno de carro, por lo que el siguiente mensaje a imprimir se posicionará en la misma línea:

?type "misma- pr "linea
misma-linea

Como podemos comprobar, ambas palabras se han visualizado en la misma línea y sin guardar ningún espacio de separación entre ellas. Si ahora quisiéramos escribir una tercera palabra, lo haría en la siguiente, porque la última primitiva utilizada ha sido **pr.**

La última de ellas es **show,** que al igual que **pr** realiza el salto a la siguiente línea. La diferencia entre ambas se encuentra a la hora de imprimir listas. Mientras que con **pr** se ignoran los corchetes, con **show** son visualizados:



?show [con corchetes diferente] pr [linea]
[con corchetes diferente]
linea

Sin embargo, los mensajes que queremos imprimir no tienen que escribirse todos necesariamente, sobre una misma columna, sino que en LOGO podemos controlar el lugar de impresión para conseguir así unas presentaciones en pantalla bastante aceptables.

Para ello, necesitaremos indicar las coordenadas de fila y columna a una primitiva, para que sea en ese punto donde se visualice el mensaje. Dicha primitiva es **setcursor.** A continuación de ella se debe especificar la lista de coordenadas, ya sea en forma de variable o de dato numérico:

```
setcursor list :x :y
setcursor [4 7]
```

En el primer caso, vemos un ejemplo de trabajo con variables. El primer parámetro (x) hace referencia a las coordenadas horizontales, columnas, y el segundo (y) a las verticales, filas, siguiendo el mismo sistema que en **dot.** Así, en el segundo ejemplo, el mensaje que pudiera venir a continuación se empezaría a escribir en la columna 4 fila 7. A este respecto hay que aclarar que la pantalla de LOGO en los CPC dispone de 40 columnas por 25 filas, en los PCW es de 90 columnas y 30 filas.

Una primitiva complementaria a la anterior es **cursor.** Su diferencia estriba en que esta última dice dónde se encuentra el cursor en el momento de su ejecución, pero sin desplazarlo.

En el siguiente programa, se emplean ambas primitivas:

```
?to cursores
>ts ct
>make "x 0 make "y 0
>make "orden 1
>repeat 22 [setcursor list :x :y pr "X
>make "x :x+1 make y :y+1]
>setcursor [0 25]
>type [hemos acabado en:\ ]
>pr cursor
>end
```

Al ejecutarlo, veremos como una «X» se va imprimiendo en diagonal por la pantalla, debido a los diferentes valores de los parámetros de **setcursor.** Antes de terminar el programa, nos avisa de en qué posición se ha detenido el cursor, gracias al uso de la primitiva **cursor.**

En lo referente a la función de las dos primeras primitivas, **ts** indica que toda la pantalla se destina a texto (ya lo vimos al hablar de **ss** y **fs**), y ct limpia toda la pantalla de texto (**cs** borra la pantalla gráfica).

Ya sabemos que existe un modo de la pantalla, **ss**, en el cual podemos escribir texto y gráficos, pero nunca mezclados. El número de líneas para escribir texto en este modo es, normalmente, de cinco, pero nosotros lo podemos modificar con **setsplit.**

```
?to lineas-texto
>ss setbg 2 cs
>make "lineas 1
>repeat 24 [make "lineas :lineas+1 setsplit :lineas
>type :lineas pr [lineas de texto]]
>end
```

Cada vez que se realiza de nuevo el conjunto de instrucciones del **repeat,** el número de líneas de texto aumenta. Lo podemos ver fácilmente, porque el color diferente de la pantalla gráfica va disminuyendo según el texto va aumentando.

INTRODUCIMOS INFORMACION

También anteriormente hemos visto, aunque muy por encima, algunas primitivas para la introducción de información en un programa: **rc, rq** y **rl.** Estudiémoslas ahora en profundidad.

Las tres son muy parecidas en su sintaxis. La primera de ellas es una abreviatura (del inglés, ¡cómo no!) de *Read Character,* lee carácter. Cuando aparece en un programa, la ejecución de éste se detiene, esperando que se pulse una tecla y automáticamente continúa el proceso.

rq proviene de *Read Quote,* leer literal. Igualmente, el programa se detiene al llegar a esta instrucción, para que escribamos un dato que será considerado en su totalidad como una palabra. Cuando terminemos de escribir, se debe pulsar RETURN o ENTER para que el programa continúe.

La última de ellas, **rl,** cuyo nombre proviene de *Read List,* leer lista, efectúa una detención como la de **rq,** con la diferencia de que el dato entrado es considerado como una lista y no una palabra. Tampoco debemos olvidar en este caso, pulsar las teclas RETURN o ENTER al terminar.

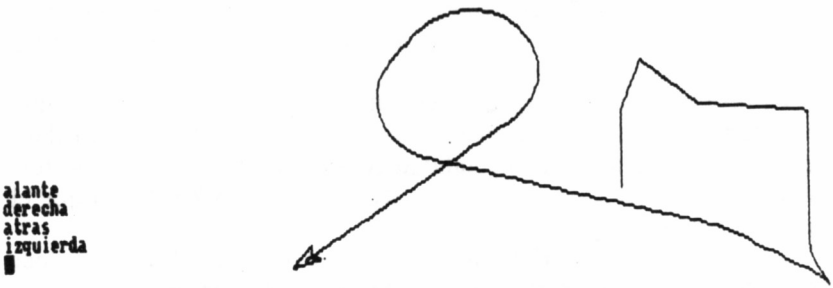
El siguiente programa nos servirá para poner en práctica las últimas seis primitivas que acabamos de aprender.

```
?to mensajes
>pr [escribe algo]
>make "letra rc
>type [solo tengo la primera letra:\ ]
>pr :letra
>type [vuelve a escribir:\ ]
>make "palabra rq
>type [todo esto es una palabra:\ ]
>show :palabra
>type [escribe de nuevo:\ ]
>make "lista rl
>type [esto es una lista:\ ]
>show :lista
>end
```

En la línea 9 de este programa se visualiza el contenido de una variable con **show.** Si lo que contiene «palabra» fuera una lista, ya sabemos que se escribirían los corchetes, cosa por otra parte imposible, dado que la primitiva que nos ha servido para introducir los datos ha sido **rq.** Por el contrario, en la línea 11 la primitiva de toma de datos empleada es **rl,** lo que supone considerar como lista lo que escribamos.

Es importante fijarse en el detalle que cada vez que se utilicen las primitivas para introducir datos (**rc, rq** y **rl),** se deben asignar directamente a una variable (como hicimos en el programa anterior) o utilizarlas directamente en la primitiva que precise el parámetro, como:

```
?fd rq
```



En este caso, el valor entrado se utiliza en la primitiva para desplazar la tortuga. Es por tanto equivalente a:

```
?make "movimiento rq
?fd :movimiento
```

UN BUEN EJEMPLO

Para sentar los conocimientos adquiridos, comprobaremos en un ejemplo la eficacia de **rc,** consiguiendo un programa «telesketch» para dibujar en la pantalla.

```
?to teles
>make "tecla rc
>if :tecla="e [fd 1 pr "alante]
>if :tecla="x [bk 1 pr "atras]
>if :tecla="d [rt 1 pr "derecha]
>if :tecla="s [lt 1 pr "izquierda]
>teles
>end
```

Como hemos podido comprobar, el sistema es bastante lento. Se puede solucionar algo cambiando los parámetros de las primitivas de movimiento (**fd, bk, rt** y **lti**) por valores más elevados.

Pero aún disponemos de otra primitiva relacionada con el teclado, aunque trabaja de una forma diferente a las descritas anteriormente: **keyp.** Siempre debe ir acompañada de un **if** y examina el teclado por si se ha pulsado alguna tecla; según el resultado de esta investigación, continuará la ejecución del programa en un punto u otro.

```
?to teclado
>if keyp [pr [por fin gracias] teclado]
>pr [por favor pulsame]
>teclado
>end
```

Al ejecutar el programa, en un primer momento se está visualizando:

```
por favor
pulsame
por favor
pulsame
por favor
pulsame
```

En el momento en que se pulse una tecla, el mensaje cambiará definitivamente:

por fin gracias
por fin gracias
por fin gracias
por fin gracias

En teoría, tenía que aparecer el mensaje anterior una vez, y posteriormente continuar con el otro; sin embargo, esto no ocurre porque LOGO en su memoria ha almacenado la pulsación de una tecla.

Esta primitiva no almacena el carácter pulsado, simplemente determina la pulsación; luego, si es necesario almacenarlo, debe ir acompañada de **rc,** como por ejemplo **if keyp [rc...].**

LITERALES: VARIABLES Y PROCEDIMIENTOS

Aunque a estas alturas del libro no creemos que haya ningún problema, vamos a estudiar, comparativamente, las tres formas que existen en LOGO de manejar una palabra, en cuanto a su sintaxis.

Cuando no lleva nada delante:

lados

nos estamos refiriendo al nombre de un procedimiento. Cada vez que escribamos una palabra de esta manera, un procedimiento con el mismo nombre comenzará a ejecutarse.

Si va precedido por dos puntos (:):

:lados

nos referimos al contenido de una variable. Si anteriormente esa variable tomó el valor 5, por ejemplo, cuando ejecutemos:

?pr :lados

veremos dicho valor en la pantalla:

5

La última opción que nos queda es que la palabra esté precedida por comillas. En este caso, estamos trabajando con el nombre de la palabra en sí, lo que se llama literal. Así, cuando introducimos un valor en una variable:

make "lados 5

nos referimos que en la palabra con ese nombre se va a almacenar dicho valor. Igualmente, para visualizar por pantalla, será el nombre y no su contenido:

?pr "lados
lados

TRATAMIENTO DE LAS LISTAS Y PALABRAS

i en el capítulo anterior aprendimos el significado de listas y palabras, ha llegado el momento de acometer el tema de su tratamiento.

Tanto de una palabra como de una lista, podemos extraer sus elementos constituyentes y trabajar con ellos e, igualmente, introducir nuevos datos en las mismas. La única característica especial que presentan es que tales elementos se pueden obtener siempre que estén en primer o último lugar y de modo similar, sólo se pueden introducir en estas mismas posiciones:

```
?make "nombre "Federico
?make "letras [a b c d e f g]
```

Si a nosotros nos interesa el primero...

```
?first :nombre
F
?first :letras
a
```

Si queremos ver el último...

```
?last :nombre
o
?last :letras
g
```

```
?deletrear
escribe palabra: otorrinolaringologo
o-t-o-r-r-i-n-o-l-a-r-i-n-g-o-l-o-g-o-
?deletrear
escribe palabra: rinoceronte
r-i-n-o-c-e-r-o-n-t-e-
?deletrear
escribe palabra: ordenador
o-r-d-e-n-a-d-o-r-
?deletrear
escribe palabra: 1234567890
1-2-3-4-5-6-7-8-9-0-
?deletrear
escribe palabra: a1b2c3d4e5f6g7h8i9j0
a-1-b-2-c-3-d-4-e-5-f-6-g-7-h-8-i-9-j-0-
?
```

Es bien sencillo, ¿verdad? Podemos imaginar que estamos en la cola del autobús, en la cual hay una persona en primer lugar (*first*, en inglés) y otra la última *(last)*. Basta indicar con la primitiva correspondiente cuál de las dos nos interesa.

ENTRE EL PRIMERO Y EL ULTIMO

El problema está en cómo acceder a los elementos que se encuentran entre el primero y el último. Es como si en nuestra cola del autobús, las personas que se encontraran en ese tramo central no pu-

```
?numeros
escribe numero: 1
[1 5]
escribe numero: 7
[1 5 7]
escribe numero: 2
[1 5 7 2]
escribe numero: 6
[1 5 7 2 6]
escribe numero: -2
[-2 1 5 7 2 6]
escribe numero: 8
[-2 1 5 7 2 6 8]
escribe numero: 4
[-2 1 5 7 2 6 8 4]
escribe numero: 8
[-2 1 5 7 2 6 8 4 8]
escribe numero: 9
[-2 1 5 7 2 6 8 4 8 9]
escribe numero: 0
[-2 1 5 7 2 6 8 4 8 9 0]
escribe numero: -7
[-2 1 5 7 2 6 8 4 8 9 0 -7]
escribe numero: 45
[-2 1 5 7 2 6 8 4 8 9 0 -7 45]
escribe numero: 2
[-2 1 5 7 2 6 8 4 8 9 0 -7 45 2]
escribe numero: 32
[-2 1 5 7 2 6 8 4 8 9 0 -7 45 2 32]
escribe numero:
```

dieran salirse, a no ser que previamente ocuparan la primera o última plaza de la fila. Para ello, disponemos de dos primitivas: **bf** y **bl.** La primera elimina el elemento inicial, dejando disponibles el resto; **bl** opera de modo similar, pero con el último elemento.

Como hemos visto, para trabajar con las listas no es necesario utilizar ninguna primitiva de visualización, **pr, show** o **type,** ya que LOGO se encarga de mostrar el resultado; sólo será necesario en los procedimientos o programas:

?bf :nombre
ederico

?bf :letras
b c d e f g

?bl :nombre
Federic

?bl :letras
a b c d e f

Con estas simples operaciones, de nuestra imaginaria cola del autobús, hemos eliminado al primero o último, quedándonos con el resto. Ahora ya es más fácil acceder a los elementos centrales.

?first bf :nombre
e

?first bf :letras
b

El mecanismo es, en resumen, suprimir, por ejemplo, la primera persona, para así convertir la segunda en la primera y poder tener acceso a ella. Lógicamente, podemos hacer lo propio comenzando por la parte trasera de la fila.

```
?last bl :nombre
c
?last bl :letras
f
```

En este caso, eliminamos el último elemento, para convertir el penúltimo en último y poderlo extraer.

RIZANDO EL RIZO

Supongamos ahora que, rizando el rizo, queremos acceder al tercer elemento, ya sea por delante o por detrás:

```
?first bf bf :nombre
d
?first bf bf :letras
c
?last bl bl :nombre
i
?last bl bl :letras
e
```

Siguiendo el mismo sistema, si quisiéramos extraer el cuarto, deberíamos ejecutar tres **bf** o **bl;** para el quinto, serían cuatro, y así sucesivamente. En resumen, para acceder a un elemento cualquiera, por el comienzo o el final, deberemos ejecutar tantos **bf** o **bl** como número de orden tenga ese elemento menos uno.

Como puesta en práctica de nuestros conocimientos de este capítulo, he aquí un programa para deletrear una palabra que escribamos.

```
?to deletrear
>type [escribe palabra:\ ]
>make "palabra rq
>make "numero count :palabra
>repeat :numero [type first :palabra type "—
>make "palabra bf :palabra]
>end
```

Lo primero que hace este procedimiento es solicitar una palabra. A cõntinuación, examina cuántas letras tiene, lo cual se consigue mediante la primitiva **count,** que inmediatamente veremos. Este valor es asignado a «número», que nos servirá para controlar el proceso de deletreo que se realiza en la siguiente línea.

count puede utilizarse igualmente con palabras o listas, dado que en realidad no cuenta letras sino elementos, ya sea de una palabra o lista. Veamos unos ejemplos:

```
?count "carnicero
9

?count [los tres cerditos]
3
```

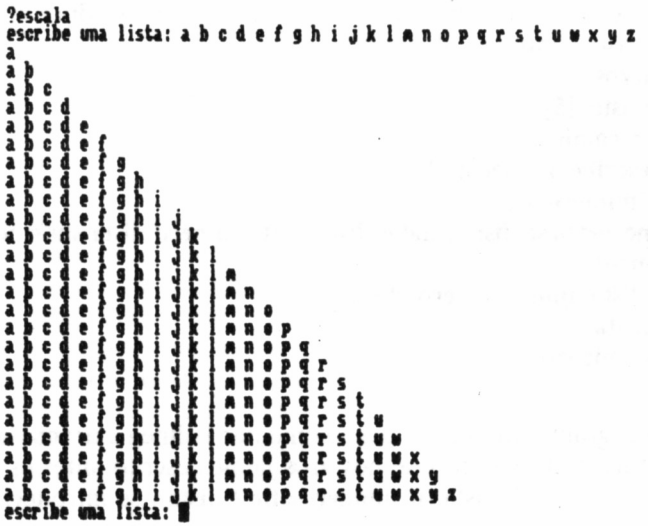
ANTES QUITAMOS Y AHORA PONEMOS

Como ya apuntamos al comienzo de este capítulo, no sólo se pueden visualizar los elementos de una lista o palabra, sino que también es posible añadírselos, aunque sólo por el comienzo o por el final.

```
?fput "el :nombre
elFederico

?fput "z :letras
z a b c d e f g
```

Según apreciamos en los ejemplos anteriores, **fput** se utiliza para introducir elementos por el principio de una lista o palabra. Pero como ya hemos dicho, también se pueden situar al final. En LOGO, la primitiva que realiza esta función es **lput.**



?lput "s :nombre
Federicos

?lput "h :letras
a b c d e f g h

No obstante, estas ampliaciones de la lista o de la palabra no son permanentes, ya que si inmediatamente realizado uno de los ejemplos anteriores, imprimimos su resultado por la pantalla, tendremos los conjuntos originales y no los ampliados:

?pr :nombre
Federico

?pr :letras
a b c d e f g

La única manera de conseguir que estos cambios permanezcan es asignarlos a variables de la siguiente forma:

?make "nombre fput "el :nombre
?pr :nombre
elFederico

?make "letras lput "h :letras
?pr :letras
a b c d e f g h

Ahora ya podemos hacer lo que queramos con los conjuntos, que los elementos añadidos no se perderán. Hagamos pruebas con el siguiente programa, que una vez escrito un número, examina si el primero de la lista es menor que éste. De ser así, lo introduce al principio, en caso contrario al final:

```
?to numeros
>make "lista [5]
>label "recomienzo
>type [escribe numero:\ ]
>make "numero rq
>if :numero<first :lista [make "lista fput :numero :lista show :lista go
"recomienzo]
>make "lista lput :numero :lista
>show :lista
>go "recomienzo
>end
```

Del programa, merece la pena comentar la línea extremadamente larga del **if;** si el número que nosotros hemos escrito antes es menor que el primero de la lista **(if :numero<first :lista),** se introducirá éste

```
adivina un color: rojo

ENHORABUENA

Numero de posicion: 1

FALLASTE

PRUEBA DE NUEVO
```

al principio (**make "lista fput :numero :lista**). En caso que la condición no se cumpla, el número irá al final.

La variable «lista» toma un valor inicial al comenzar el programa (5). Esto es necesario porque si no tuviera ninguno, la primera vez que se ejecutara la línea de la pregunta, buscaría el primer elemento de esa lista y, al no existir, el programa se interrumpiría con un mensaje de error.

Otro aspecto también a destacar son las primitivas **go** y **label.** Ambas deben ir siempre juntas en el programa; nunca estarán solas. **label** identifica una línea (su significado es etiqueta) con un nombre (en el ejemplo **label "recomienzo**). Cuando el programa llega a la instrucción **go,** la ejecución del mismo da un salto y continúa por la línea del **label** que tenga el mismo nombre (**go "recomienzo,** donde *go* significa ir), en este caso, casi al comienzo del procedimiento. Es el equivalente al GOTO del BASIC, e, igualmente, su uso no es recomendable, porque los programas se hacen bastante difíciles de seguir a la hora de corregirlos.

MÁS PRIMITIVAS

Hemos visto hasta aquí las principales primitivas para manejar palabras y listas, introducir y extraer elementos de ellas. A partir de ahora, estudiaremos otras que complementarán a aquellas, dando mayor potencia a nuestro LOGO.

Lo primero que vamos a hacer es transformar el programa «deletrear»:

```
?to del
>make "letra 1
>type [escribe palabra:\ ]
>make "pal rq
>if emptyp :pal [type [te dije que escribieras algo:\ ] make "pal rq]
>make "numero count :pal
>repeat :numero [type uc item :letra :pal
>make "letra :letra+1]
>end
```

Aunque el programa, en un primer momento, asuste por ser más largo y la cantidad en aumento de primitivas desconocidas, comprobaremos que ha mejorado considerablemente. En primer lugar (línea 3) nos pide que escribamos una palabra. En la quinta línea, se comprueba si en realidad hemos escrito algo; si nos hemos vuelto un poco vagos y simplemente hemos apretado la tecla ENTER, el programa insistirá mostrando otro mensaje que casi nos fuerza a escribir la palabra.

Esta evaluación sobre si hemos escrito algo o no, corre en la quinta línea a cargo de la primitiva **emptyp** (del ingles *EMPTY Property,* propiedad de vacío). Examina si el parámetro que le acompaña, en este caso **:pal,** contiene simplemente el vacío. En tal caso, el resultado es TRUE y se ejecuta lo que viene a continuación; de no ser así, el valor que toma es FALSE y la ejecución prosigue en la línea inferior.

La siguiente primitiva desconocida es **uc** *(Upper Case),* que se encuentra directamente relacionada con **lc** *(Lower Case).*

El efecto que produce la primera de ellas, lo podemos ver inmediatamente si tomamos la precaución de escribir la palabra en letra minúscula. Transforma la palabra o un elemento de la lista (no actúa sobre listas enteras) en su correspondiente en mayúsculas. Evidentemente, **lc** opera de forma inversa: la pasa de mayúsculas a minúsculas.

```
?uc "CaSeRiO
CASERIO
```

?lc "cAlEnDaRiO
calendario

La última primitiva nueva en el programa es **item,** artífice del deletreo en cuestión. Escoge del conjunto de entrada el número de elemento que se especifique.

?item 5 "relojero
j

Ya sea palabra o lista.

?item 3 [ordenador impresora monitor teclado]
monitor

En el programa va aumentando el valor de la variable **:letra,** que determina qué letra de la palabra, contenida en **:pal,** se va a imprimir.

En estrecha relación de significado con **item** se encuentra **piece.** Mientras que la anterior se refiere a un único elemento, ésta lo hace a un intervalo de elementos especificado por dos parámetros:

?piece 2 4 "calendario
ale

?piece 2 3 [ordenador impresora monitor teclado]
impresora monitor

En el primer ejemplo, se visualiza del segundo al cuarto elemento, ambos inclusive, de la palabra que se encuentra a continuación. En el siguiente, nos referimos al segundo y tercer elemento de la lista.

Es fácil deducir que, en realidad, **item** es un caso particular de **piece,** de tal forma que pueden ser equivalentes, en determinadas situaciones:

```
?item 3 "caniche
n
?piece 3 3 "caniche
n
```

Con el siguiente programa, podremos ver claramente la función de esta última primitiva:

```
?to escala
>make "letra 1
>type [escribe una lista:\ ]
>make "lista rl
>make "numero count :lista
>repeat :numero [pr piece 1 :letra :lista make "letra :letra + 1]
>escala
>end
```

SEGUIMOS EXTRAYENDO ELEMENTOS

Siguiendo con el grupo de primitivas que extraen elementos, nos encontramos con dos nuevas, **memberp** y **where,** cuya presencia conjunta en los programas no es necesaria, aunque sí están en relación.

La primera averigua si el elemento indicado se encuentra dentro del conjunto que nosotros determinemos. En caso afirmativo, da como respuesta TRUE; si no, el resultado es FALSE.

```
?memberp "r "calendario
TRUE
?memberp "b [a b c d e f]
TRUE
?memberp "r "castillo
FALSE
?memberp "b [a b c d e f]
FALSE
```

La segunda primitiva, **where,** proporciona el número de orden del elemento hallado con **memberp:**

```
?memberp "r "calendario
TRUE
?pr where
8
?memberp "b [a b c d e f]
TRUE
?pr where
2
```

Como podemos comprobar, **where** tiene que ir acompañada de **memberp,** pero no viceversa. En el siguiente programa, se utilizan ambas primitivas, simulando un juego sencillo, en el cual hemos cuidado la presentación en pantalla como sugerencia de presentación de textos.

```
?to colores
>wait 100 ct
>make "lista [rojo naranja amarillo verde azul violeta]
>setcursor [1 12] type [adivina un color:\ ]
>make "color rq ct
>if memberp :color :lista [setcursor [14 7] pr "ENHORABUENA set-
cursor [10 17] type [Numero de posicion:\ ] pr where
colores]
>setcursor [16 10] pr "FALLASTE
>setcursor [12 17] pr [PRUEBA DE NUEVO]
>colores
>end
```

Si el color introducido coincide con alguno de la lista, el valor que toma **memberp** es TRUE, ejecutándose lo que se encuentra a continuación en la línea. Con **where,** nos indica de una manera sencilla en qué posición se encuentra. Otra forma de manejar información de una lista, y bastante más original es **shuffle.**

Esta primitiva cuyo significado es barajar, ya sabemos en qué idioma, varía la posición de los elementos de una lista de una forma aleatoria. Lo mejor es verlo con un ejemplo:

```
?make "dado [1 2 3 4 5 6]
?shuffle :dado
[3 4 1 6 2 5]
```

Sin embargo, la lista «dado» no ha cambiado:

```
?show :dado
[1 2 3 4 5 6]
```

Como ya sabemos, la única manera de alterar su contenido es realizando una asignación (con **make**) a la misma lista, o a otra distinta.

Cualquiera de las dos opciones anteriores es válida: en el primer caso, el valor original de la lista se cambia definitivamente, mientras que en el segundo éste se conserva:

```
?make "dado shuffle :dado
?show :dado
[1 5 3 2 6 4]
?make "nuevo-dado shuffle :dado
?show :nuevo-dado
[4 3 5 1 2 6]
```

En el siguiente programa, se pone en práctica esta nueva forma de manejar la información de una lista.

```
?to dados
>make "numeros [1 2 3 4 5 6]
>make "tirada shuffle :numeros
>type [tiro el dado y...\ ]
>pr first :tirada
>dados
>end
```

OPERACIONES MATEMÁTICAS

as operaciones aritméticas, como sabemos, es necesario realizarlas con números, o con variables cuyo contenido sea igualmente numérico. En Dr. LOGO existe una primitiva que examina si un elemento concreto es un número o no. En caso de que no lo sea, podremos actuar del modo conveniente, y si lo es, operaremos con él. Esta primitiva es **numberp.**

Ya conocemos varias de su misma familia (**memberp, emptyp...**), y dada nuestra experiencia habremos adivinado que su nombre proviene de *NUMBER Property,* propiedad de número. Si el parámetro que le acompaña tiene un valor numérico, su resultado es TRUE; en caso contrario, FALSE. En el siguiente procedimiento podemos ver cómo trabaja:

```
?to numero
>type [escribe algo:\ ]
>if numberp rq pr [es un numero] numero] pr [no es un numero]
>numero
>end
```

Las operaciones más simples, obviamente, son las cuatro principales: suma, resta, multiplicación y división. Las podemos hacer de dos formas; una de ellas ya conocida:

```
?32+17      ?45-13      ?13*3       ?14/2
49          32          39          7
```

No debemos olvidar que el asterisco (*) es el símbolo utilizado para la multiplicación, para evitar confusiones con «x». De forma similar, la división se obtiene con la barra inclinada (/) y no con dos puntos (:).

El otro método para calcular con estas cuatro operaciones es:

```
?+ 32 17    ?- 45 13    ?* 13 3     ?/ 14 2
49          32          39          7
```

En realidad, da lo mismo utilizar un sistema que otro. Este último método en otras versiones de LOGO se complementa con primitivas que indican las operaciones a realizar, como pueden ser ADD (suma), SUBSTRACT (resta), MULTIPLY (multiplicación) y DIVIDE (división). Sin embargo, esta posibilidad no existe en Dr. LOGO, debiendo recurrir en todo caso a los mencionados signos.

Por otra parte, es posible obtener directamente el cociente de la división de dos números:

```
?quotient 30 5
6
```

Análogamente, el resto de la división:

```
?remainder 30 5
0
```

Asimismo, disponemos de dos primitivas para averiguar el valor exacto de un número, sin decimales: **round** e **int.**

Mientras que la primera redondea el número que escribamos, la segunda calcula su parte entera. La diferencia entre ambos procesos se encuentra en que, mientras en un redondeo, el número decimal se transforma en el entero más próximo (así 3.2 se convierte en 3 y 3.7 en 4), **int** halla el número decimal en el entero inmediatamente inferior (3.2 y 3.7 se convierten en 3).

```
?round 2.1    ?round 2.9
2             3
?int 2.1      ?int 2.9
2             2
```

```
?numero
escribe algo: r
no es un numero
escribe algo: a
no es un numero
escribe algo: 1
es un numero
escribe algo: 456
es un numero
escribe algo: 45a
no es un numero
escribe algo: a34
no es un numero
escribe algo: abc
no es un numero
escribe algo: 1234
es un numero
escribe algo:
no es un numero
escribe algo: 0
es un numero
escribe algo: -9
es un numero
escribe algo: █
```

```
?nuevo_dados
tiro el dado y... 1
tiro el dado y... 1
tiro el dado y... 4
tiro el dado y... 4
tiro el dado y... 1
tiro el dado y... 3
tiro el dado y... 2
tiro el dado y... 1
tiro el dado y... 1
tiro el dado y... 6
tiro el dado y... 2
tiro el dado y... 2
tiro el dado y... 6
tiro el dado y... 2
tiro el dado y... 3
tiro el dado y... 2
tiro el dado y... 4
tiro el dado y... 5
tiro el dado y... 5
tiro el dado y... 6
tiro el dado y... 2
tiro el dado y... 5
tiro el dado y... 2
tiro el dado y... 6
tiro el dado y... 5
tiro el dado y... 2
tiro el dado y... 4
tiro el dado y... 5
tiro el dado y... 6
tiro el dado y... 4
tiro el dado y... 6
tiro el dado y... 6
tiro el dado y... 3
tiro el dado y... 3
tiro el dado y... 6
tiro el dado y... 3
tiro el dado y... 4
tiro el dado y... 4
```

CUESTIÓN DE SUERTE

Como en BASIC, LOGO puede proporcionarnos números aleatorios a través de **random.** Estos serán enteros y menores que el parámetro que indiquemos **random.** Por ejemplo:

?random 12

Indica que va a dar un número aleatorio desde cero hasta once (recordemos que no llega al parámetro especificado), todos ellos enteros.

Anteriormente, simulamos el lanzamiento de un dado con las listas, utilizando **shuffle.** Ahora vamos a ver el mismo caso con **random.** Para ello, debemos limitar el valor de esta primitiva, en primer lugar por abajo, ya que no es posible que salga el valor cero, porque no lo tienen los dados. Lo mejor en este caso es sumar 1 al número resultante; así, si resulta cero, al añadirle este valor se convierte en uno: **1 + random.**

También debemos limitar el valor superior. Como el 7 no lo tienen los dados, el parámetro de la primitia debería ser este número (ya que los resultados no lo alcanzarán nunca), pero como anteriormente ya indicamos que al número que saliera se le sumaba uno (para evitar el cero), cada vez que obtengamos el seis, el resultado será siete. Por tanto, el parámetro de **random** debe ser 6; todos los números serán menores que este valor, pero cuando obtengamos un cinco al añadirle uno, obtendremos el 6 de los dados:

?1+random 6

El programa quedaría:

```
?to nuevo-dados
>make "tirada 1+random 6
>type [tiro el dado y...\ ]
>pr :tirada
>nuevo-dados
>end
```

Como complemento a **random** tenemos **rerandom.** Su misión es que la aleatoriedad de **random** se encuentra limitada, ya que cada vez que se utilice esta primitiva, la sucesión de números al azar es la misma. Veamos un ejemplo:

```
?repeat 5 [type random 6 type "-]
3-1-2-4-3
?rerandom
?repeat 5 [type random 6 type "-]
3-1-5-0-2
```

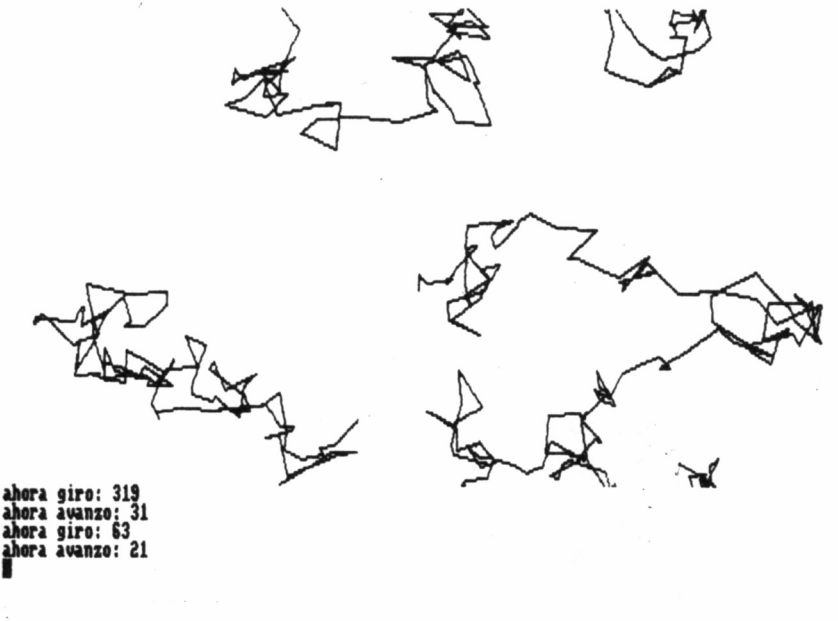
```
?rerandom
?repeat 5 [type random 6 type "-]
3-1-5-0-2
```

¿Qué explicación tiene este fenómeno? Nuestro ordenador no está capacitado para extraer números al azar, como si fuéramos nosotros, que los sacamos de «la nada». Cuando se le pide un valor aleatorio, lo que hace es poner en marcha una función matemática que da una serie de números, sin aparente relación, pero que siguen una secuencia específica. El valor que toma para operar la función es el que tiene el reloj interno del ordenador, que se conecta al enchufarlo.

Si implementamos esta primitiva en la segunda línea del programa anterior «nuevo-dados», veremos que siempre sale el mismo valor del dado.

Por otra parte, podemos combinar aleatoriedad con gráficos y dejar al ordenador que se convierta en artista durante unos momentos. En el siguiente programa, LOGO realizará gráficos aleatorios, determinado el mismo cuanto girar y avanzar:

```
?to dibujo-yo
>ss ht wrap
>make "avance random 40
```



```
>type [ahora avanzo:\ ]
>pr :avance
>fd :avance
>make "giro random 360
>type [ahora giro:\ ]
>pr :giro
>rt :giro
>dibujo-yo
>end
```

FUNCIONES TRIGONOMÉTRICAS

En el campo matemático, Dr. LOGO también tiene un espacio reservado a la trigonometría. De hecho ya hemos trabajado con ella al efectuar algunos gráficos de tortuga. Las funciones trigonométricas que nos ofrece son: **sin, cos** y **arctan.**

Respectivamente, se refieren al seno, coseno y arcotangente. A partir de éstas, se pueden extraer otras razones trigonométricas de ángulos:

— TANGENTE = SENO/COSENO
— SECANTE = 1/COSENO

— COTANGENTE = COSENO/SENO
— COSECANTE = 1/SENO

Algunos ejemplos de utilización son:

?sin 0	**?cos 0**	**?arctan 0**
0	1	0
?sin 90	**?cos 90**	**?arctan 1**
1	0	45

Ya sabemos que en LOGO podemos dibujar con la tortuga (no nos habremos olvidado de este útil animalito) o directamente, con puntos, utilizando ejes cartesianos. Aprovechando precisamente este último sistema y sobre todo, que el origen de referencia se encuentra en el centro de la pantalla, podemos representar funciones matemáticas. La dificultad en LOGO se encuentra que no trabaja con el concepto de función, como se hace en el BASIC. Por tanto, cada vez que queramos representar otra diferente, habrá que alterar el programa. Este consta de dos procedimientos:

```
?to funciones
>ss
>make "x -55
>dibujando :x
>end
```

En el primer procedimiento, simplemente se determina la disposición de la pantalla (**ss,** mixta de gráfico y texto) y el valor inicial de la coordenada «x» (en los PCW se aconseja que sea −65), que variará en cada caso, dependiendo de la función con la que trabajemos, para ocupar el máximo de pantalla. Una forma más elegante de introducir este valor, es mediante **rq.**

El segundo procedimiento es:

```
?to dibujando :x
>make "y int 1/15*:x*:x
>type [x=] type :x
>type [\ Y=] pr :y
>dot list :x :y
>dibujando :x+1
>end
```

Este es el procedimiento que se encarga de los cálculos y de dibujar los puntos. La función está en la segunda línea del programa; lo que traducido al lenguaje de las matemáticas sería: Y = 1/15 * X ↑ 2. Se trata de una función de segundo grado («*» es el signo de multiplicar en informática y « ↑ » es el de la potenciación, operación de la cual no disponemos en LOGO).

Otras funciones que dan en pantalla unos dibujos interesantes son:

Y = X * SEN X

Que en LOGO supondría:

>make "y :x*sin :x

Siendo aconsejable que «x» comience por el valor −215.

>make "x −215

Otra función es:

Y = X/2 * SEN 2 * X

Su «traducción»:

>make "y :x/2*sin 2*:x

El valor inicial de «x» podría ser −295:

>make "x −295

OPERACIONES LÓGICAS

Este tipo de operaciones, a diferencia de las anteriores, no dan ningún resultado numérico, sólo lógico: TRUE, FALSE (verdadero o falso). Ya hemos trabajado con tres de estos operadores: > < y =.

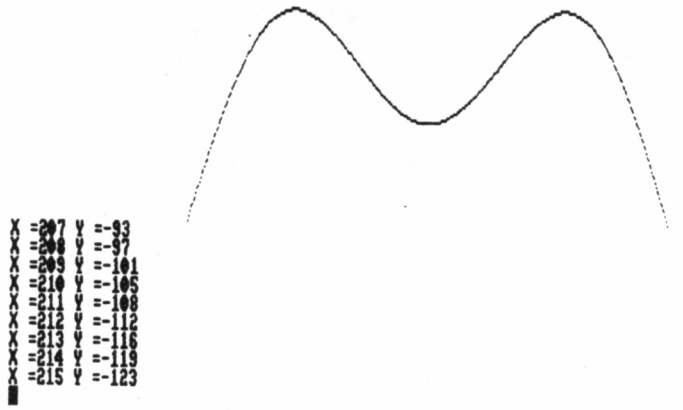
Su significado respectivo es «mayor», «menor» e «igual». Son utilizados para las comparaciones, y como hemos dicho, responden con el valor TRUE si se cumple la condición, y FALSE en el caso contrario. Si se utilizan en un programa, se ejecutan junto con la primitiva **if.**

```
?make "mayor 9
?make "menor 5

?:mayor<:menor  ?:mayor>:menor  ?:mayor=:menor
FALSE           TRUE            FALSE
```

Efecto idéntico a «=» se obtiene con la primitiva **equalp.** Si los dos números que se deben poner a continuación son iguales (proviene de *EQUAL Property,* propiedad de igualdad) su respuesta es TRUE; si no, es FALSE.

```
?equalp :mayor :menor
FALSE
```



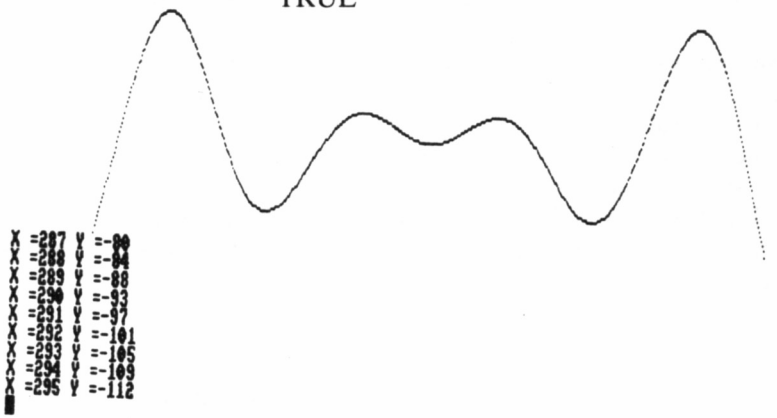
En LOGO, no tenemos la posibilidad de realizar las comparaciones de «menor o igual» (<=), «mayor o igual» (>=) o «distinto» (<>) directamente. Para ello, necesitamos un operador lógico: **not.**

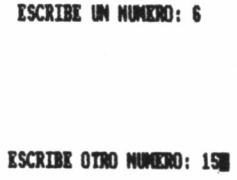
Supone la negación de la expresión que le acompaña. Así, para averiguar si dos números son distintos, deberíamos preguntar si «no son iguales»:

```
?not :mayor=:menor
TRUE
```

Si preguntamos sobre si un número es «mayor o igual» que otro, es lo mismo que averiguar si ese número «no es menor»:

```
?not :mayor<:menor
TRUE
```





Análogamente, decir que un número es «menor o igual» que otro, equivale a que «no sea mayor»:

```
?not :mayor>:menor
FALSE
```

La forma de evaluar este tipo de expresiones es algo complicada, ya que hablamos de negaciones o de contrarios. El método más sencillo es mirar el resultado de la expresión sin el **not** y a continuación, cambiar el resultado con la negación.

Igualmente, es posible combinar **not** con **equalp,** con idéntica función que =, así como todas las primitivas vistas hasta ahora sobre «propiedad»: **not equalp, not wordp, not emptyp** y **not listp...**

OPERADORES LÓGICOS

El resto de operadores lógicos, a diferencia de **not,** se aplican sobre dos parámetros. Estos son: **and** y **or.** Con el primero de ellos, el resultado es verdadero cuando lo sea el de TODAS las expresiones que se estén evaluando. En caso de que en una de ellas sea FALSE lo será el resultado final del **and.** Con **or,** el resultado es verdadero cuando lo sea AL MENOS una de las expresiones y por tanto FALSE, si ambas lo son.

Las llamadas «tablas de verdad», parecidas a las de multiplicar y sumar aritméticamente, resumen los posibles resultados de los dos operadores:

TABLA AND	TABLA OR
0 and 0 = 0	0 or 0 = 0
0 and 1 = 0	0 or 1 = 1
1 and 0 = 0	1 or 0 = 1
1 and 1 = 1	1 or 1 = 1

OPERACIONES CON AND Y OR

En LOGO se trabaja con los operadores de la siguiente manera:

```
?and (:mayor<:menor) (:mayor>:menor)
FALSE

?or (:mayor<:menor) (:mayor>:menor)
TRUE
```

Si manejamos más de dos expresiones:

```
?and (3=5) (2=2) (3=2)
FALSE
FALSE

?or (3=3) (2=1) (1=2)
TRUE
FALSE
```

Aparecen dos respuestas porque son realizadas dos comparaciones: la primera condición con la segunda y ésta con la tercera.

De igual forma, podemos combinar **and** con **or:**

```
?and (3=3) or (2=1) (1=1)
TRUE
```

```
LO SIENTO


INTENTALO DE NUEVO
```

En este ejemplo se compara la primera expresión con la segunda por medio de **and,** siendo el resultado FALSE. A continuación, se compara la segunda con la tercera con **or,** lo cual da como resultado TRUE. La combinación de ambos resultados se hace con **or,** por ser el último operador empleado y obtenemos finalmente TRUE.

Asimismo, podemos combinar estos operadores con **not:**

```
?and not (:mayor<:menor) (:mayor>:menor)
TRUE

?or (:mayor<:menor) not (:mayor>:menor)
FALSE
```

El siguiente programa utiliza algunos de los operadores en un juego muy simple, en el cual volvemos a cuidar la presentación de la pantalla:

```
?to par-impar
>wait 100 ct
>setcursor [9 8]
>type [ESCRIBE UN NUMERO:\ ]
>make "par rq
>setcursor [8 16]
>type [ESCRIBE OTRO NUMERO:\ ]
>make "impar rq
>ct
>if and (:par/2=int :par/2) not (:impar/2=int :impar/2) [setcursor [14
12] pr [HAS ACERTADO] par-impar]
>setcursor [15 8] pr [LO SIENTO]
>setcursor [11 16] pr [INTENTALO DE NUEVO]
>par-impar
>end
```

Con este programa obtendremos un resultado favorable si el primer número que escribamos es par y el segundo impar, circunstancia que se comprueba en la línea del **if.** Todos los números pares son divisibles por dos, por tanto, si la mitad de un número es igual a la parte entera de su mitad, será par; si no, impar, porque la división por dos no es exacta y el resultado no coincide con el de la parte entera del cociente.

LOS PERIFÉRICOS

En este capítulo trataremos de todos los periféricos con los que se puede relacionar Dr. LOGO y las primitivas necesarias para sacar el máximo partido a los mismos.

FICHEROS EN DISCO

Desde LOGO, es posible guardar información en un dispositivo externo, para que si desenchufamos el ordenador, ésta no se pierda definitivamente y podamos utilizarla otro día sin necesidad de repetir todo el proceso. El único dispositivo externo que utiliza Dr. LOGO para almacenar información es el disco, pudiendo guardar dos tipos de ficheros: procedimientos y pantallas.

Uno de los fallos principales de casi todas las versiones de este lenguaje es la imposibilidad de almacenar ficheros directamente, lo cual impide la realización de procedimientos de bases de datos (agendas, por ejemplo); una lástima, dada la potencia de búsqueda del manejo de listas (aunque sea de escasa velocidad).

Primeramente, trabajaremos con procedimientos. Para guardarlos, disponemos de la primitiva **save,** que deberá ir seguida del nombre del programa. Por ejemplo, **save "division**

No obstante, no se graba únicamente el procedimiento que tenga ese nombre, sino también todos aquellos que estén en ese momento en memoria. Así pues, el nombre del fichero que se graba no tiene porqué coincidir con el del procedimiento.

Para recuperar el fichero, emplearemos habitualmente **load,** seguido lógicamente del nombre de referencia, aunque existe otro método más: **edf "division.**

La diferencia con **load** estriba en que **edf,** tras cargar el procedimiento, entra en edición *(EDit File,* edita fichero), por si nos interesa realizar algunas transformaciones.

Por otra parte, si queremos ver el catálogo de procedimientos LOGO que tenemos grabados en el disco, deberemos ejecutar:

?dir
[DIVISION]

En caso que no tengamos ningún fichero LOGO aparecerá una lista vacía ([]), puesto que esta primitiva ignora todos aquellos ficheros no generados por LOGO, incluso los de pantalla.

El borrado de ficheros se consigue con **erasefile,** seguido del nombre a eliminar del directorio. Por ejemplo: **erasefile "division**

PANTALLAS

Para trabajar con pantallas utilizaremos prácticamente las mismas primitivas empleadas para los programas, añadiendo el sufijo «pic». Como siempre, hay que escribir el nombre de la pantalla que queremos grabar o cargar.

Para grabar:

?savepic "triangulos

Para cargar:

?loadpic "triangulos

Por otra parte, este mismo sufijo se aplica para visualizar el directorio de pantallas, que no mostrará los programas.

?dirpic
[TRIANGULOS]

En relación con estas primitivas hay que añadir que tanto **dir** como **dirpic** aceptan el símbolo comodín «?», el cual puede sustituir a

una letra. Si, por ejemplo, queremos ver cuántos ficheros tenemos que empiecen por algún prefijo concreto, basta con que ejecutemos, por ejemplo:

?dir "tri?????

Así, obtendremos sólo los ficheros que empiecen por esas tres letras, resultando indiferente el resto de su nombre.

De igual modo, se emplea **erasepic** para el borrado de pantalla...

?erasepic "triangulos

En lo que se refiere a los nombres, Dr. LOGO acepta cualquiera sin límite teórico en el número de caracteres, pero en el momento de grabar los recorta a un máximo de ocho, por exigencias del sistema operativo.

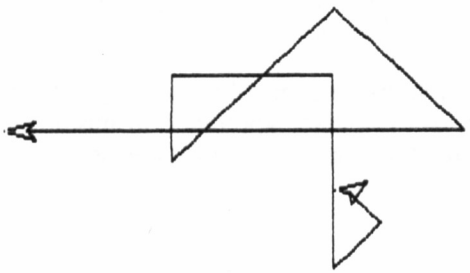
Finalmente, podemos cambiar el nombre a los ficheros grabados, gracias a **changef.** Para ello, debemos especificar dos parámetros: el nuevo nombre que va a tener el fichero y el antiguo, siguiendo ese mismo orden.

?changef "suma "division

El programa que habíamos llamado «division», se denomina ahora «suma».

DOS DRIVES

En el supuesto de disponer de dos unidades de disco, podemos elegir con cuál de ellas trabajar, mediante la primitiva **setd.** Ya sabemos que los AMSTRAD sólo admiten dos unidades de disco, y por tanto, los parámetros de la primitiva anterior serán «a:» o «b:».



Así, para trabajar con la unidad «a»:

?setd a:

Si queremos cambiar a la «b»:

?setd b:

Podremos comprobar con qué unidad estamos trabajando con:

?defaultd

Esta primitiva nos da como resultado cuál de las dos unidades se está utilizando. Evidentemente, sólo hay dos respuestas posibles: A: o B:

No obstante, siempre tenemos la posibilidad de trabajar con un diskete diferente al que nos indica **defaultd.** Si estamos trabajando con la unidd A y queremos grabar una pantalla en la B, no necesitamos cambiar de unidad con **setd,** sino que basta ejecutar, por ejemplo:

?savepic "b:castillo

Así, la pantalla «castillo» es almacenada en la unidad de disco B. De forma similar se procede con el resto de las primitivas que manejan los discos. Por ejemplo, con

?dir "b:mat?????

obtenemos el nombre de los programas del disco B cuyo nombre empieza por «mat».

LA IMPRESORA

El manejo de la impresora desde este lenguaje es fácil, porque sólo tiene dos primitivas: **copyon** y **copyoff.** La primera de ellas activa la salida por impresora, de tal forma que todos los caracteres que aparezcan en la pantalla saldrán igualmente por la impresora. La segunda desactiva este modo.

Así pues, no es posible hacer copias de dibujos por la impresora. Sólo los PCW tienen esta facilidad, apretando al mismo tiempo las teclas EXTRA e IMPR, destinadas a este uso, que se pueden utilizar en cualquier situación, no solo en LOGO.

JOYSTICK

En los CPC existe la posibilidad de conectar directamente un joys-

tick al port destinado a tal uso. Esto no sucede en los PCW, que se tienen que conformar únicamente con el teclado.

Las primitivas para controlar el joystick son **buttonp** y **paddle.** La primera, controla si ha sido pulsado el botón de disparo, y la segunda determina la posición del joystick. Ambas tienen un parámetro, por si queremos utilizar dos joysticks:0 para el primero y 1 para el segundo. **paddle** da 9 valores para que nosotros identifiquemos cuál es la posición actual del mando. Uno de ellos es 255 en el caso de la posición central. Los ocho restantes son:

0 ARRIBA
1 ARRIBA Y DERECHA
2 DERECHA
3 ABAJO Y DERECHA
4 ABAJO
5 ABAJO E IZQUIERDA
6 IZQUIERDA
7 ARRIBA E IZQUIERDA

Si el joystick que estamos utilizando es de cuatro posiciones, son ignorados los valores 1, 3, 5 y 7.

```
?to joy
>if paddle 0=0 [seth 0 type "0\ joy]
>if paddle 0=1 [seth 45 type "1\ joy]
>if paddle 0=2 [seth 90 type "2\ joy]
>if paddle 0=3 [seth 135 type "3\ joy]
>if paddle 0=4 [seth 180 type "4\ joy]
>if paddle 0=5 [seth 225 type "5\ joy]
>if paddle 0=6 [seth 270 type "6\ joy]
>if paddle 0=7 [seth 315 type "7\ joy]
>if buttonp 0 [fd 20 type "A\ ]
>joy
>end
```

En este programa se van realizando preguntas sucesivas sobre la posición del primer joystick y cuando se cumpla la condición se ejecuta el giro correspondiente. Por último, se interroga sobre si el botón de disparo ha sido pulsado (también del primer joystick); en ese caso la tortuga avanza 20 posiciones.

EL SONIDO

Los AMSTRAD CPC son ordenadores con gran capacidad para la generación de sonidos. Dr. LOGO también accede a esta posibilidad. Esta circunstancia no es habitual entre las diferentes versiones de LOGO, por lo que las primitivas de este campo no están extendidas como la gran mayoría de las vistas hasta ahora. Esto llega al extremo que estas primitivas son las mismas que maneja el ordenador en BASIC y su uso es idéntico, por lo que huelga extenderse en exceso sobre el particular. Aquí veremos simplemente unas ligeras nociones; para ampliarlas es necesario dirigirse al manual de BASIC del ordenador. El Dr. LOGO de los PCW no tiene ninguna posibilidad sonora.

La primitiva para la generación de sonido es **sound** y tiene siete parámetros que iremos describiendo por orden:

— Canal de salida: Podemos elegir tres vías, o canales diferentes (A, B o C), para que el sonido salga. Es como si tuviéramos tres altavoces totalmente independientes, pero con la particularidad de que si queremos un mismo sonido, puede salir por dos de ellos o los tres.

Los valores que puede tomar son:

Canal A: 1	Canal B: 2
Canal C: 4	Sincronización con A: 8
Sincronización con B: 16	Sincronización con C: 32
Retención del sonido: 64	Borrado del canal: 128

Si por ejemplo queremos enviar un sonido por el canal B, el valor del parámetro será 2. Si queremos que el sonido salga al mismo tiempo por el B y el C, deberá ser 6 (2 de enviar el sonido por el B más 4 de hacerlo por el C). Los dos últimos valores, 64 y 128, son para retener el sonido o no. Por otra parte, por los canales de sincronización podemos enviar diferentes sonidos a un mismo canal.

— Altura tonal: Es el que determina el tono del sonido (más grave o más agudo). Como referencia diremos que la nota DO media tiene el valor 239. Cuanto mayor sea dicho valor, más agudo será el sonido.

— Duración del sonido: Se mide en centésimas de segundo (0.01 s); por tanto, para un segundo deberemos escribir 100. Si no se especifica duración toma el valor 20 (2 décimas de segundo, 0.2 s).

— Volumen: Además de regularse con el potenciómetro del ordenador, puede controlarse desde un programa. Los valores van desde 0 hasta 15 y, en caso de no especificarlo, toma como implícito 12.

— Número de envolvente de volumen: Mientras se está emitiendo un sonido, es posible hacer variar el volumen. Para conseguir este efecto, necesitamos una nueva primitiva: **env.**

— Número de envolvente de tono: El tono también puede variarse según se emite un sonido. Igual que ocurría con el volumen, hay que recurrir a otra primitiva: **ent.**

— Ruido: Se puede entender como la distorsión emitida junto al sonido puro. El valor mínimo es 1 y el máximo 31.

Con **sound** sólo son imprescindibles los dos primeros parámetros: número de canal y tono, siendo el resto opcionales.

Control De Las Envolventes

Las envolventes de volumen y tono, **env** y **ent,** respectivamente, tienen un funcionamiento similar. Para establecer la variación, debemos tener en cuenta que se van a realizar en una serie de escalas, habiendo entre cada una de ellas una diferencia de tono o volumen

concreta, con una duración del sonido determinada en cada etapa.

— Número de envolvente: Es el número con el que hemos identificado a las envolventes. Debe coincidir con el de **sound.** Así, cuando se vaya a ejecutar esta última primitiva, se buscará la envolvente con el mismo número.

— Número de etapas: Se debe establecer en cuántos escalones se va a producir la variación. Los valores entre los que debe estar comprendido son 0 y 127.

— Diferencia entre etapas: Se especificará la diferencia que se produce al saltar de una etapa a otra, en volumen o tono. El margen de valores es de 0 a 127. En el caso de **env,** cuando sea mayor que 15 volverá automáticamente a 0.

— Duración de cada etapa: Establece cuanto va a durar el sonido en cada etapa, que irá desde 0 a 255.

Una vez vista toda la teoría, pasemos a los casos prácticos. Los parámetros de las tres primitivas deben ser especificados en una lista (por tanto, entre corchetes):

```
?sound [1 239 150 10 0 1]
```

En este caso, nos indica que vamos a utilizar el canal A, la nota 239 (el DO normal) con una duración de 1.5 segundos, un volumen de 10, no se emplea envolvente de volumen, pero si la de tono, con el número 1 y el ruido no se utiliza (su parámetro no se especifica).

```
?env [1 20 3 10]
```

Ahora utilizamos la envolvente de volumen 1, cuya variación se desarrollará en 20 etapas, entre las cuales el volumen variará en 3 unidades y la duración del sonido en cada etapa será una décima de segundo.

```
?ent [1 20 15 10]
```

El caso de **ent** es prácticamente el mismo que el de **env;** la única diferencia se encuentra en el tercer parámetro, en el cual se indica que en cada salto, el tono variará en 15 unidades.

Con los siguientes programas podremos probar la efectividad de la combinación de estas tres primitivas:

```
?to volumen
>env [1 100 20 10]
>sound [1 142 1000 10 1]
>end

?to tono
>ent [1 120 5 2]
>sound [1 100 240 15 0 1 5]
```

```
>tono
>end
```

Como dijimos al principio del capítulo, sólo vamos a dar unas breves orientaciones sobre cómo se trabaja el sonido en el Dr. LOGO; la mejor forma de conocer esta posibilidad a fondo es hacer pruebas. Por tanto, aconsejamos que en los programas anteriores, se vayan alterando todos los parámetros para observar sus efectos, al igual que del próximo, que nos muestra el uso de los canales, y en el cual es necesario aguzar el oído:

```
?to canal
>end [1 100 2 2]
>sound [1 100 400 10] ;canal A
>sound [2 200 300 9 0 0 5] ;canal B
>sound [4 400 200 15 0 1] ;canal C
>end
```

Si probamos repetidamente el programa, nos daremos cuenta que están saliendo diferentes sonidos a la vez. Uno de ellos variará de agudo a grave, el del canal C que utiliza la envolvente de tono. Este será el primero en dejar de sonar (dura 2 segundos). Otro de ellos emitirá ruido, canal B, y se parará el siguiente y el último en enmudecer será bastante agudo. Hemos emitido a la vez tres sonidos diferentes, gracias a los canales.

Queda una última primitiva que nos sirve para liberar («callar») el sonido retenido por el valor 64 en el primer parámetro de **sound;** ésta es **release.** Se acompaña de un parámetro que podrá tomar los siguientes valores:

1 libera canal A
2 libera canal B
3 libera canales A y B
4 libera canal C
5 libera canales A y C
6 libera canales B y C
7 libera canales A B y C

EL RESTO DE LAS PRIMITIVAS

En este último capítulo, estudiaremos todas aquellas primitivas que se han quedado fuera en el resto del libro, principalmente porque no tenían cabida en el tema de ninguno de sus capítulos. No obstante, para su mejor comprensión, presentaremos juntas aquellas que guarden alguna relación.

ASCII. CHAR

Cada uno de los caracteres que maneja el ordenador (letras y demás símbolos) tienen asignado un código. Podemos saber cuál es el código de cada carácter y viceversa. **ascii** proporciona el código del carácter que escribamos o el primero de la palabra de entrada.

```
?ascii "Arco
65
?ascii "arco
97
```

char efectúa la operación inversa: nos da el carácter correspondiente al código que escribamos.

```
?char 65
A
```

Uno de los caracteres más curiosos es el pitido de aviso del ordenador; lo escucharemos tecleando **char 7.** En el siguiente programa, podemos ver los caracteres del ordenador «visibles» (otros caracteres «invisibles», con código ASCII, código estándar en ordenadores, son, por ejemplo, el espacio el RETURN, borrado de caracteres...):

```
?to caracteres
>make "num 33
>repeat 223 [type (list char :num "\ )
>make "num :num+1]
>end
```

LOCAL

Cuando en un procedimiento trabajamos con una variable, el valor de ésta se transmite a todos los procedimientos que estén en memoria, de tal forma que si se emplea en otro diferente, seguirá con el valor que ya tenía asignado. Podemos, aún así, trabajar con una variable en un procedimiento cuyo valor sólo será utilizado en éste, gracias a la primitiva **local.** Con el siguiente programa veremos más claramente su uso.

```
?to general
>make "variable 10
>particular
>type [resultado general=]
>pr :variable
>end

?to particular
>local "variable
>make "variable 5
```

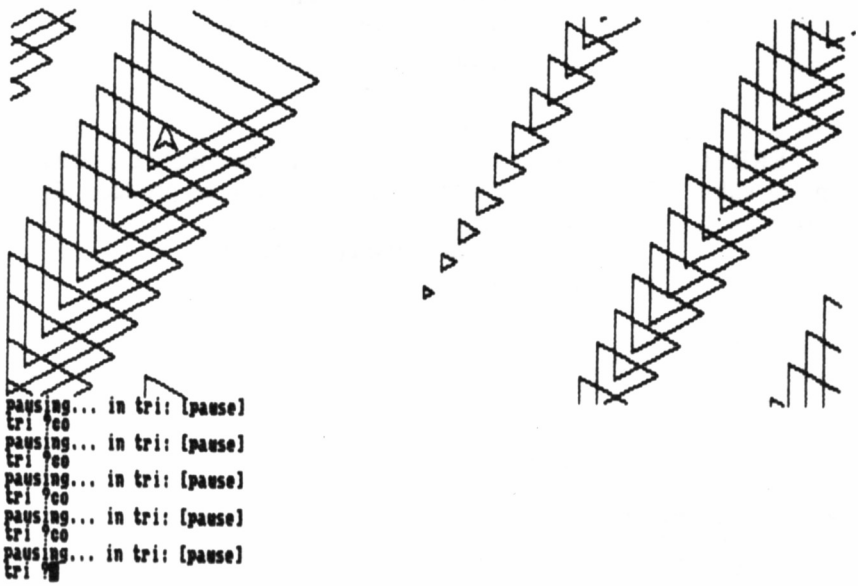
```
?general
resultado particular =5
resultado general =10
?
```

```
>type [resultado particular=]
>pr :variable
>end
```

En «general», «variable» vale 10. En «particular», «variable» la hacemos específica de ese procedimiento concreto y toma el valor 10, que se visualiza en primer lugar. Si no fuera una variable local, ese mismo valor se imprimiría en el procedimiento «general», sin embargo, podemos comprobar que no es así.

RUN

A lo largo del libro, hemos ejecutado comandos en «modo directo», es decir, sin escribir en programas. Si nos hemos equivocado al



hacerlo, debemos volver a teclear todo de nuevo. Con esta primitiva ahorramos trabajo. Con un ejemplo lo veremos más claro:

```
?make "cuadrado [repeat 4 [fd 50 rt 90]]
?run :cuadrado
```

Primeramente, se realiza una asignación, como si fuera una variable, y posteriormente se la ejecuta con **run.** La potencia de este comando es tal que podemos editarlo **(ed)** o listarlo:

```
?po "cuadrado
cuadrado is [repeat 4 [fd 50 rt 90]]
```

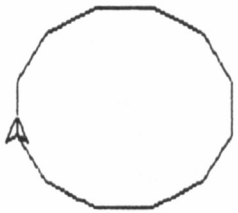
PAUSE. CO. ERRACT

Ya sabemos que el efecto de **stop** es detener la ejecución de un procedimiento, y, la única forma de reanudarla es volviendo a empezar. Sin embargo, ¡gracias a **pause** (o también apretando CONTROL + Z) el procedimiento es detenido, con posibilidad de continuar por el mismo punto, ejecutando la primitiva **co.**

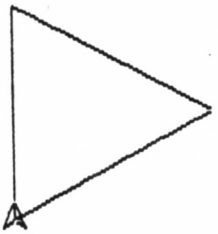
```
?to tri :lado
>wrap
>repeat 3 [fd :lado rt 120]
>pu rt 30 fd 30 lt 30 pd
>pause
>tri :lado+5
>end
```

ERRACT también provoca una pausa, pero solamente cuando tiene el valor TRUE (normalmente es FALSE) y si se produce algún error en la ejecución del programa:

```
?make "ERRACT "TRUE
?to mas-errores
```



```
?make "ERRACT "TRUE
?mas_errores
pausing... in mas_errores: :contenido
mas_errores ?
```



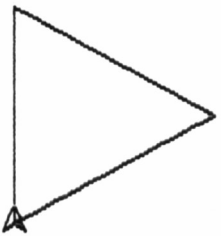
```
ya hemos terminado
?a
```

```
>repeat 12 [fd 50 rt 30]
>pr :contenido
>end
```

Al intentar imprimir la variable «contenido», como no tiene ningún valor asignado, se produce un error, pero al ser ERRACT TRUE, se genera una pausa, indicando el punto de interrupción.

CATCH. THROW

Estas dos primitivas simulan un trabajo con subrutinas, ya que **catch** tiene una etiqueta y cuando aparece en el programa una instrucción **throw** con esa misma etiqueta, el programa continúa por la siguiente línea a donde se encuentra **catch.**



```
?inicio
[36 [lado has no value] dibujo [pr :lado] pr :lado]
?
```

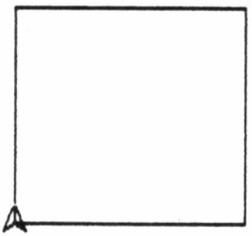
```
?to primero
>ts cs px
>catch "proc [segundo 10]
>pr [ya hemos terminado]
>end
?to segundo :lado
>repeat 2 [repeat 3 [fd :lado rt 120] wait 50]
>if keyp [throw "proc]
>segundo :lado+5
>end
```

En el **catch,** además de tener la etiqueta («proc») entre corchetes, como si fuera una lista, se encuentra el nombre del procedimiento por el que va a continuar la ejecución del programa. Cuando en «segundo» se llegue a **throw** con el mismo nombre el programa continuará por la siguiente línea del **catch** (**pr** en el ejemplo).

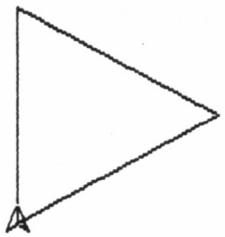
ERROR

Esta primitiva se utiliza en combinación con **catch.** Cuando se produce algún error de ejecución en el programa, éste es mostrado en forma de lista, con su número.

```
?to inicio
>catch "error [dibujo]
```



```
?trace comienzo
[1] Evaluating comienzo
Numero de lados: 4
[1] making "lados 4
[1] making "angulo 90
[2] Evaluating dibujando
[3] Evaluating texto
?[END 4LADOS
[4] Evaluating comienzo
Numero de lados:
```



```
?watch comienzo
[1] In comienzo, type [Numero de lados: ]
Numero de lados:
[1] In comienzo, make "lados rq
3
[1] In comienzo, make "angulo 360 / :lados
[1] In comienzo, dibujando
[2] In dibujando, repeat :lados [fd 200 rt :angulo]
[2] In dibujando, texto
[3] In texto, type "TIENE  type :lados pr "LADOS
```

```
>show error
>end
?to dibujo
>repeat 3 [fd 50 rt 120]
>pr :lado
>pr [fin del dibujo]
>end
```

Cuando el primer procedimiento llega a la instrucción **catch,** se ejecuta «dibujo». Al producirse la situación de error (intentar imprimir el contenido de una variable, «lado», sin nada), se regresa a «inicio», mostrándose qué error se ha cometido.

```
?glist ".PRN
[dirpic numberp arctan quotient setsplit copyoff nowatch shuffle notrace define savepic p
addle setcursor memberp px recycle st ts noformat rt ss pu rq pr or lt po op rl ht loadpi
c tf fs sf se wc rerandom er ct go es rc run pe pd pps co not lc if bl bk changef erasepi
c int sin fd ed defaultd bf dot pots stop lput ern text cos setq setx type pops setscrume
h show pons fput dir ) = pal list word ( wrap keep remainder wait last throw seth and ppr
op erasefile setd item edf save / - listp wordp plist + error * dotc) count home tones (f
irst gprop round fill .deposit glist load emptyp setpc char make pause setpos cursor .con
tents where thing nodes poall copyon window watch setbg .examine namep erall trace setpen
 buttonp local ascii setpal equalp piece .out remprop .in towards catch clean edall repea
t random fence label]
?
```

TRACE. NOTRACE

La primera primitiva activa el seguimiento de ejecución de LOGO, indicándose qué procedimiento se está ejecutando. Dr. LOGO se encuentra por defecto en **notrace.**

```
?to comienzo
>type [Numero de lados:\ ]
>make "lados rq
>make "angulo 360/:lados
>dibujando
>end

?to dibujando
>repeat :lados [fd 50 rt :angulo]
>texto
>end

?to texto
>type "TIENE\ type :lados pr "LADOS
>comienzo
>end
```

Antes de ejecutar el programa, escribamos **trace** y veremos cómo nos va informando del procedimiento que está en ejecución.

WATCH. NOWATCH

Estas primitivas son similares a las anteriores, pero el seguimiento es más cercano, al indicarnos qué primitiva se va a ejecutar, deteniéndose al mismo tiempo el programa. La ejecución continuará pulsando RETURN. Probemos estas primitivas (por supuesto, **nowatch** desconecta este modo de seguimiento) con el programa anterior, tecleando **watch** antes de la ejecución.

GLIST. PLIST. GPROP. PPROP. PPS. REMPROP. .APV. .DEF. .PRM

.APV se refiere al valor que tiene una variable, .DEF a la definición de un procedimiento y .PRM a una primitiva. Sin embargo, estos comandos no pueden ser utilizados por sí solos y requieren el resto de las primitivas del grupo.

glist proporciona una lista de los objetos que cumplen la condición que se especifique a continuación. Si tenemos en memoria los tres programas anteriores y ejecutamos:

?glist ".APV
[lados angulo]

Hemos obtenido una lista de las variables que tiene algún valor (a lo que se refiere .APV). Si queremos ver lcs nombres de los procedimientos que tenemos en memoria ejecutaremos:

?glist ".DEF
[dibujo inicio texto]

Podríamos suponer que si hacemos lo mismo con .PRM obtendríamos las primitivas empleadas, sin embargo, veremos las que se encuentran en memoria, es decir, todas las primitivas que se pueden utilizar en Dr. LOGO:

?glist ".PRM
[dirpic numberp arctan quotient...

plist escribe la propiedad (primitiva, variable, programa) a la que pertenece el parámetro que indiquemos:

?plist "inicio
[.DEF [[] [type [Numero de lados:]]...]]]

Nos señala que «inicio» es un procedimiento (.DEF) y a continuación escribe todo el procedimiento. Ya sabremos lo que significan los siguientes casos:

?plist "angulo
[.APV 90]

(el «90» se refiere al valor actual de la variable «angulo»)

?plist "list
[.PRM 2965]

(el «2965» es la posición de memoria donde se encuentra almacenada la primitiva)

gprop proporciona el valor del objeto cuya propiedad se especifica. Este valor del objeto puede ser el de una variable o el procedimiento:

?gprop "angulo ".APV
?gprop "list ".PRM
2965

?gprop "inicio ".DEF
[[] [type [Numero de lados:]] [make "lados rq]...

pprop consigue que una palabra tenga la propiedad que nosotros le indiquemos. Por ejemplo:

?pprop "numero ".APV "456

número es una variable (.APV) cuyo valor es «456»:

?:numero
456

Aunque de una forma complicada, podemos crear así un procedimiento:

?pprop "saludo ".DEF [[] [pr "hola]]
?saludo
hola

Este comando nos permite también crear propiedades nuevas que no estén incorporadas en Dr. LOGO, como .APV, .PRM, .DEF:

?pprop "femenino "nombres "Rosa

Hemos incluido el nombre «Rosa» en «femenino», con la propiedad «nombres». Lo comprobaremos con:

?plist "femenino
[nombres Rosa]

pps nos permite ver todas esas propiedades nuevas que hemos creado:

?pps
femeninos nombres is Rosa

Con **remprop** eliminamos la propiedad que hemos definido al objeto que concretemos:

?remprop "femenino "nombres

Hemos eliminado, en este ejemplo, la propiedad «nombres» de «femenino». Podremos comprobar que ha desaparecido con dos de las primitivas que conocemos:

?pps
?
?plist "femenino
[]

.CONTENTS

Escribe todo el contenido del espacio destinado al área de trabajo. No sólo se refiere a las primitivas, sino también a los nombres de procedimientos y variables que hayamos creado. Si se borran éstos, se sigue manteniendo en dicho espacio. Esta primitiva no lleva parámetros:

?.contents
[.DEF dirpic A: an empty...]

Guarda cierto parecido con **glist** ".**PRM,** con la diferencia que éste sólo proporciona las primitivas.

.EXAMINE. .DEPOSIT

La memoria de un ordenador se puede comparar con una serie de celdillas ordenadas, donde se va guardando la información. Con **.examine** vemos el contenido de la celdilla (más correctamente llamando posición) de memoria que indiquemos. **.deposit,** por su parte, «depositamos» un número en una determinada posición de memoria.

?.examine 0
1

Hemos examinado el contenido de la posición 0, que es un 1. Ahora vamos a cambiarlo:

?.deposit 0 15

Ya sabemos cómo comprobarlo:

?.examine 0
15

.IN. .OUT

Estas primitivas son muy similares a las anteriores, pero no se refiere a posiciones de memoria, sino a «puertas» de comunicación.

.in lee el valor de la puerta indicada y **.out** envía un valor a dicha puerta:

```
?.contents
[primero .DEF :contenido dibujo dirpic inicio an empty word numberp arctan quotient setsp
lit copyoff has no valor Numero segundo nowatch LADOS shuffle lados:  notrace define save
pic FALSE TOPLEVEL paddle setcursor nonberp px recycle st ts noformat rt ss pu comienzo r
q "ERRACT pr "ERROR or a lt po op rl ht loadpic tf :lado ya fs sf se uc rerandom er et go
cs "proc "texto re run pe pd pps co not le if bl bk :lados changef erasepic "error int si
n fd pd default de bf dot pots stop lput grn text cos sety setx terminado type pops sets
crunch TYPE show ERRACT pons ERROR :angulo fput dir .PWD ) = pal fin list word ( ; wrap k
eep remainder "TIEMPO  READPT wait del "lados last throw seth proc texto 4 and 3 pprop era
sefile setd dibujando iten edf save / - listp wordp plist + error # dote) count bone tone
s (first gprop round "angulo fill .deposit glist mas_errores load lado emptyp setpc "dibu
jo char make pause setpos cursor .PRM .contents hemos where thing nodes pcall copyon wind
ow watch "segundo setbg .examine .APV .PWI I don't know how to "LADOS TIEMPO  .SPC lados .
REM nanep erall trace setpen .EOL buttonp local ascii setpal equalp piece .out angulo pen
prop .in towards catch clean "a edall repeat random fence I can't find catch for label]
?
```

```
?.in 0
0
?.out 0 15 .in 0
15
```

En este último caso, hemos enviado por la puerta 0 el valor 15 e inmediatamente lo hemos leído con **.in.**

OP

op finaliza la ejecución del procedimiento en que se encuentra, y el parámetro adjunto puede ser utilizado en el programa que le llamó:

```
?to proced :numero
>repeat 5 [type (list :numero "-) make "numero :numero*2]
>op [fin de numeracion]
>pro :numero
>end
```

Al ejecutar el programa, veremos cómo una vez visualizado el mensaje se detiene a pesar de la recursividad con **pro :numero.**

```
1 -2 -4 -8 -16 -[fin de numeracion]
```

REDEFP

La utilización de **REDEFP** debe ser prudente, dado que los cambios que realiza pueden ser drásticos. Cuando tiene el valor TRUE es posible redefinir las primitivas, «olvidándose» de la función primaria que tenían.

```
?to first
first is a primitive
```

Hemos intentado escribir un procedimiento llamado first, pero LOGO lo impide porque es nombre de primitiva. Si ahora hacemos:

```
?make "REDEFP "TRUE
```

Ya podemos escribir nuestro procedimiento:

```
?to first
>pr [first ya no es primitiva]
>end
```

Al escribir ahora su nombre, se ejecutará el programa. Para volver a la situación original debemos salir de LOGO:

```
?first
first ya no es primitiva
```

TOPLEVEL

Se utiliza con **throw** y sale de todos los procedimientos que estén en proceso. La diferencia con **stop** es que ésta regresa al procedimiento anterior y TOPLEVEL no lo hace, saliendo directamente:

```
?to primero
>segundo
>pr [primer procedimiento]
>end

?to segundo
>pr [segundo procedimiento]
>if rc="f [stop]
>primero
>end
```

Al comenzar la ejecución, salta al segundo procedimiento y si pulsamos «f» el programa se detiene:

```
?primero
segundo procedimiento
segundo procedimiento
segundo procedimiento
```

(pulsamos «f»)

```
primer procedimiento
primer procedimiento
primer procedimiento
?
```

Cambiemos la línea de **if** por:

```
if rc="f [throw "TOPLEVEL]
```

Ejecutemos ahora el programa:

```
?primero
segundo procedimiento
segundo procedimiento
segundo procedimiento
```

(pulsamos «f»)

```
?
```

En esta ocasión, no se ha vuelto al procedimiento original, debido a **TOPLEVEL.**

IT

Esta primitiva nos permite escribir texto en la zona de gráficos, en la misma posición en la que se encuentra la tortuga. Es una lástima que sólo esté disponible en el Dr. LOGO de CM/M 2.2, porque se pueden conseguir efectos interesantes, además de ser el único sistema de mezclar gráficos y texto. Como ejemplo simple:

?repeat 6 [fd 100 tt "# rt 60]

BYE

Dejamos para el final esta primitiva cuya función es casi evidente. No requiere parámetros y, como su nombre indica en inglés, se despide de nosotros: es la manera de abandonar LOGO y regresar al CP/M.

?bye

LAS PRIMITIVAS EN LOS AMSTRAD

resentamos ahora una lista completa de las primitivas de Dr. LOGO. Con un asterisco se señalan qué modelos de AMSTRAD las incorporan en su versión de dicho lenguaje, teniendo en cuenta que la columna CPC-464 se refiere al CP/M 2.2, válida también en los CPC 472/464 y la titulada CPC 6128 afecta al CP/M 3.0 ó PLUS, implementada en los PCW 8256/8512.

Primitivas	**CPC-464**	**CPC-6128**	**PCW-8256**
+	*	*	*
–	*	*	*
*	*	*	*
/	*	*	*
<	*	*	*
>	*	*	*
=	*	*	*
.APV	*	*	*
.contents	*	*	*

.DEF	*	*	*
.deposit	*	*	*
.EMT		*	*
.ENL		*	*
.examine	*	*	*
.in		*	*
.out		*	*
.PRM	*	*	*
.REM		*	*
.SPC		*	*
and	*	*	*
arctan		*	*
ascii	*	*	*
bf	*	*	*
bk	*	*	*
bl	*	*	*
buttonp	*	*	*
bye	*	*	*
catch	*	*	*
changef		*	*
char	*	*	*
clean	*	*	*
co	*	*	*
copyoff		*	*
copyon		*	*
cos	*	*	*
count	*	*	*
cs	*	*	*
ct	*	*	*
cursor		*	*
defaultd		*	*
define		*	*
dir	*	*	*
dirpic		*	*
dot	*	*	*
dotc		*	*
ed	*	*	*
edall		*	*
edf		*	*
emptyp	*	*	*
end	*	*	*
ent	*	*	
env	*	*	
equalp		*	*
er	*	*	*

erall		*	*
erasefile		*	*
erasepic		*	*
ern	*	*	*
ERRACT	*	*	*
error	*	*	*
FALSE	*	*	*
fd	*	*	*
fence	*	*	*
fill			*
first	*	*	*
fput	*	*	*
fs	*	*	*
glist	*	*	*
go	*	*	*
gprop	*	*	*
home		*	*
ht	*	*	*
if	*	*	*
int	*	*	*
item	*	*	*
keyp	*	*	*
label	*	*	*
last		*	*
lc		*	*
list	*	*	*
listp		*	*
load	*	*	*
loadpic		*	*
local	*	*	*
lput		*	*
lt	*	*	*
make	*	*	*
memberp		*	*
namep		*	*
nodes	*	*	*
noformat		*	*
not	*	*	*
notrace		*	*
nowatch		*	*
numberp	*	*	*
op	*	*	*
or	*	*	*
paddle	*	*	
pal	*	*	

pause	*	*	*
pd	*	*	*
pe	*	*	*
piece		*	*
plist	*	*	*
po	*	*	*
poall		*	*
pons		*	*
pops		*	*
pots	*	*	*
pprop	*	*	*
pps		*	*
pr	*	*	*
pu	*	*	*
px	*	*	*
quotient		*	*
random	*	*	*
rc	*	*	*
recycle	*	*	
REDEFP	*	*	*
release	*	*	*
remainder		*	*
remprop	*	*	*
repeat	*	*	*
rerandom		*	*
rl	*	*	*
round		*	*
rq	*	*	*
rt	*	*	*
run	*	*	*
save	*	*	*
savepic		*	*
se	*	*	*
setbg		*	
setcursor		*	*
setd		*	*
seth	*	*	*
setpal		*	
setpc	*	*	*
setpen	*	*	*
setpos	*	*	*
setscrunch		*	*
setsplit	*	*	*
setx		*	*
sety	*	*	*

sf	*	*	*
show	*	*	*
shuffle		*	*
sin	*	*	*
sound	*	*	
ss	*	*	*
st	*	*	*
stop	*	*	*
text		*	*
tf	*	*	*
thing		*	*
throw	*	*	*
to	*	*	*
TOPLEVEL	*	*	*
towards		*	*
trace		*	*
TRUE	*	*	*
ts	*	*	*
tt	*		
type	*	*	*
uc		*	*
wait	*	*	
watch		*	*
where		*	*
window	*	*	*
word	*	*	*
wordp	*	*	*
wrap	*	*	*

Este libro pretende ser una guía completa de explicación de los comandos de LOGO, con el fin de que el lector vislumbre las limitaciones y posibilidades que éste le ofrece. Se trata pues, de una considerable ampliación del manual, incluyendo incluso algunos comandos que en éste se omiten, acompañado de gran cantidad de programas y ejemplos que, intercalados en el texto, hacen la lectura más amena, a la par que didáctica.

GRAN BIBLIOTECA AMSTRAD

450 ptas.
(incluido IVA)
Precio en Canarias, Ceuta y Melilla: 435 ptas.