Guía completa de programación en BASIC

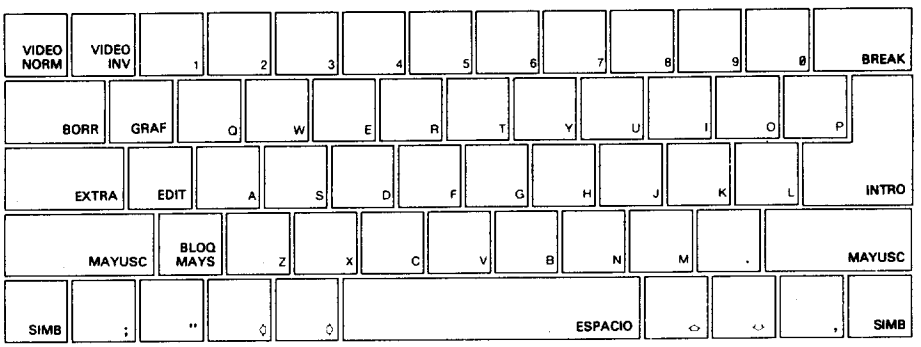
Parte 1
Introducción

Tanto si ha leído primero el capítulo 6 como si ha venido directamente a éste, debe saber que:

Las *órdenes* son obedecidas inmediatamente.
Las *instrucciones* comienzan con un número de línea y son almacenadas para su uso posterior.

Esta guía de BASIC comienza repitiendo algunas de las cosas tratadas en el capítulo 6, pero de forma más detallada. Al final de algunas secciones hemos incluido ejercicios: le recomendamos que no los ignore, pues muchos de ellos ilustran conceptos que sólo han sido mencionados de pasada en el texto. Écheles un vistazo y trabaje con los que más le interesen, o con los que traten de alguna cuestión que el texto no le haya dejado completamente clara. En cualquier caso, no deje de experimentar con el ordenador. Siempre que se pregunte qué haría el +2 si usted escribiese tal o cual cosa, la respuesta es sencilla: pruebe y lo verá. Recuerde que, escriba lo que escriba, no puede dañar al +2.

El teclado



Los caracteres utilizados en el +2 no son solamente símbolos simples (letras, dígitos, etc.) sino también símbolos compuestos (palabras clave, nombres de funciones, etc.). Todo debe ser escrito completamente; y en la mayor parte de los casos es indiferente hacerlo en mayúsculas o minúsculas. En el teclado hay tres clases de teclas: las de letras y números (llamadas alfanuméricas), las de símbolos (signos de puntuación) y las teclas de control (tales como MAYUSC, BORR, etc.).

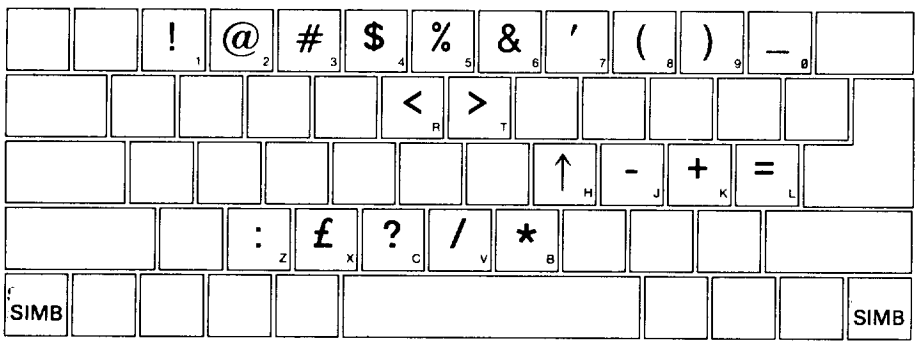
Las teclas alfanuméricas son las más frecuentememte utilizadas en BASIC. Cuando se pulsa una tecla alfabética, en la pantalla aparece una letra minúscula junto con un cuadrado parpadeante (cuyo color alterna entre azul y blanco), llamado *cursor*. Para obtener la mayúscula se debe mantener pulsada la tecla MAYUSC al tiempo que se escribe la letra.

Si desea escribir con mayúsculas continuadamente, pulse una vez la tecla BLOQ MAYS: todas las teclas alfabéticas que pulse en lo sucesivo producirán letras mayúsculas. Para volver a las minúsculas, pulse otra vez BLOQ MAYS.

Para escribir los símbolos que aparecen en las teclas alfanuméricas, es decir,

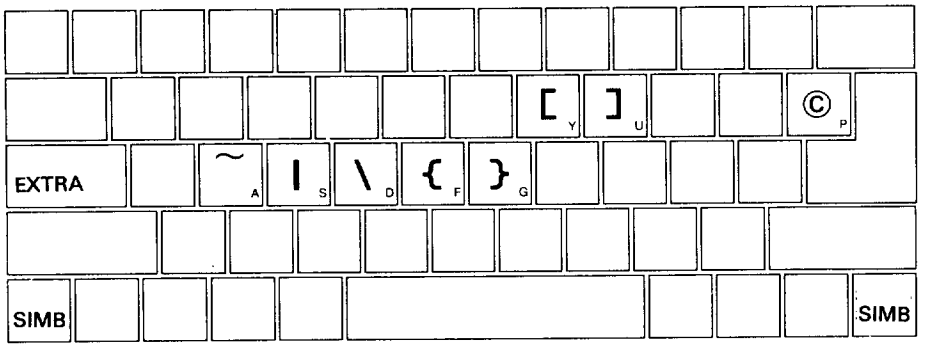
! @ # $ % & ' ( ) _ < > ↑ − + = : £ ? / *

pulse la tecla correspondiente en combinación con SIMB (véase el diagrama siguiente).



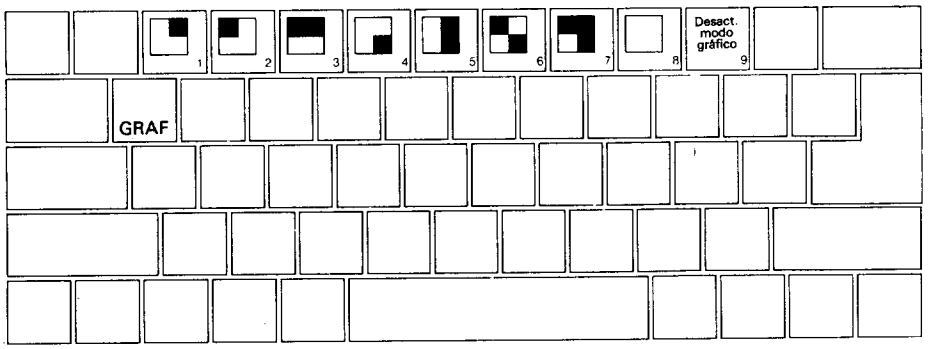
Símbolos disponibles con SIMB

Además se puede obtener los símbolos [ ] © ~ | \ { y } pulsando una vez la tecla EXTRA y pulsando luego una tecla alfabética en combinación con SIMB (véase el diagrama siguiente).



Símbolos disponibles con SIMB en modo **EXTRA**

Para activar el modo de gráficos, pulse una vez la tecla GRAF. Los gráficos de mosaico (véase el siguiente diagrama) se obtienen entonces pulsando las teclas numéricas (excepto 9 y 0). Combinado MAYUSC con las teclas numéricas citadas se obtiene los mismos gráficos de mosaico, pero con los colores invertidos. Pulsando las teclas alfabéticas (excepto **T**, **U**, **V**, **W**, **X**, **Y** y **Z**) se obtiene los gráficos definidos por el usuario.



Gráficos de mosaico disponibles en modo **GRAF**

Si tecla se mantiene pulsada una tecla durante más de 2 o 3 segundos, comenzará a repetirse. A medida que se va pulsando teclas, se forma una línea en la pantalla. Dicho sea de paso, por *línea* entendemos una línea de BASIC, la cual puede ocupar varias líneas físicas en la pantalla. La línea puede ser recorrida utilizando las teclas de movi-

miento del cursor: [←], [→], [↑], [↓]. Si la parte de la línea hacia la que se mueve el cursor se encuentra fuera de la pantalla, entonces el texto se moverá hacia arriba o hacia abajo para hacerla visible. Cualquier carácter que se escriba será insertado en la posición del cursor. Pulsando [BORR] se borra el carácter que está a la izquierda del cursor. En cuanto se pulsa [INTRO], o si se intenta sacar el cursor de la línea, el ordenador comprueba si la línea tiene sentido. Si lo tiene, genera un pitido agudo y obedece la línea inmediatamente, o bien la almacena como parte de un programa. Si la línea contiene un error, el ordenador emite un pitido grave y lleva el cursor a la zona en la que piense que se encuentra dicho error (además, el color del cursor cambia a rojo). Es imposible salir de una línea que contenga un error: el +2 siempre volverá a llevar el cursor a esa línea.

La pantalla

La pantalla consta de 24 líneas (de 32 caracteres cada una) y está dividida en dos partes. La más amplia (la superior) tiene a lo sumo 22 líneas y puede mostrar tanto el listado como los resultados del programa. Es la que generalmente se utiliza para editar líneas. Cuando la impresión en la *pantalla superior* ha llegado a su borde inferior, el contenido se desplaza una línea hacia arriba. No obstante, si este desplazamiento implica la pérdida de una línea que aún no se ha tenido la oportunidad de leer, el +2 se detiene y emite el mensaje:

```
scroll?
```

Si se pulsa cualquier tecla (excepto **N**, [BREAK] o la barra espaciadora), el desplazamiento en vertical continúa.

Si se pulsa la tecla **N**, [BREAK] o la barra espaciadora, el programa se detiene y emite el mensaje:

```
D  BREAK - CONT repeats
```

La parte más pequeña (inferior) de la pantalla se utiliza para editar programas cortos, introducir datos y órdenes directas (cuando no conviene usar la pantalla superior; por ejemplo, en programas de gráficos) y para exhibir informes.

Introducción de programas

Si el programa que está siendo introducido sobrepasa el tamaño de la pantalla, el +2 intentará presentar el área de mayor interés (generalmente, la última línea introducida junto con las cercanas a ella). No obstante, usted puede hacer que el ordenador muestre otra área del programa especificándola en la orden:

```
LIST xxx
```

donde *xxx* es un número de línea. Esta orden hace que el +2 exhiba la zona del programa especificada.

Cuando se ejecuta una orden o un programa, el resultado se muestra en la pantalla superior, donde permanece cuando el programa termina (hasta que se pulsa una tecla). Si el programa está siendo editado en la pantalla inferior, los resultados exhibidos en la pantalla superior permanecerán en ella hasta que se escriba sobre ellos, hasta que desaparezcan por efecto del desplazamiento en vertical o hasta que se ejecute una orden CLS. La pantalla inferior muestra un informe que consiste en un código (dígito o letra), cuyo significado explicaremos en la parte 28 de este capítulo. Ese informe permanecerá en la pantalla inferior hasta que se pulse una tecla.

Mientras el +2 está ejecutando un programa de BASIC, la tecla [BREAK] es comprobada de vez en cuando; concretamente, al final de cada sentencia, durante la utilización del magnetófono y de la impresora y mientras se interpreta la música. Si el +2 detecta que se ha pulsado [BREAK], detiene la ejecución del programa y se emite el informe:

D o bien L

y entonces el programa puede ser editado.

Parte 2
Conceptos sencillos de programación

Temas tratados:

Programas
Números de línea
Edición de programas utilizando las teclas del cursor
RUN, LIST
GO TO, CONTINUE, INPUT, NEW, REM, PRINT
Detención de un programa

Escriba las dos primeras líneas de un programa que, cuando esté completo, calculará y exhibirá en la pantalla la suma de dos números:

```
20 print a     (pulse INTRO)
10 let a=10    (pulse INTRO)
```

Observe que la pantalla muestra lo siguiente:

```
10 LET a=10
■
20 PRINT a
```

Como hemos visto antes, puesto que estas líneas comenzaban con números, el +2 no las obedeció inmediatamente, sino que las almacenó como líneas de programa. Este ejemplo muestra también que los números de línea rigen el *orden* en que las líneas de programa van a ser ejecutadas; tal y como puede ver en la pantalla, el +2 ordena todas las líneas cada vez que se introduce una nueva.

Observe además que, aunque habíamos escrito cada línea con letras minúsculas, el ordenador ha convertido a mayúsculas las palabras clave (PRINT y LET) en cuanto ha aceptado la línea. De ahora en adelante mostraremos en letras mayúsculas la información que se debe escribir; sin embargo, usted puede continuar escribiéndola en minúsculas.

Hasta ahora sólo hemos introducido un número de los dos que tenemos que sumar, así que escriba:

```
15 LET b=15     (pulse INTRO)
```

Ahora debemos transformar la línea 20 del siguiente modo:

```
20 PRINT a+b
```

Se podría escribir la nueva línea completa, pero es mucho más cómodo situar el cursor justamente detrás de la a y luego escribir:

```
+b      (no pulse todavía [INTRO])
```

La línea debería quedar así:

```
20 PRINT a+b
```

Ahora pulse [INTRO]; el cursor se desplaza a la línea de abajo, de forma tal que en la pantalla se ve lo siguiente:

```
10 LET a=10
15 LET b=15
20 PRINT a+b
■
```

Para ejecutar el programa dé la orden:

```
RUN     (pulse [INTRO])
```

La suma aparecerá en la pantalla.

Ejecute nuevamente el programa y después escriba:

```
PRINT a,b     (pulse [INTRO])
```

Observe que las variables continúan en la memoria, a pesar de que el programa ya ha terminado.

Si introducimos una línea por error, por ejemplo

```
12 LET b=8
```

para borrarla basta con escribir su número y pulsar [INTRO]:

```
12      (pulse [INTRO])
```

La línea 12 desaparece del listado y el cursor se sitúa en el lugar en donde antes estaba la línea.

Ahora escriba:

```
30      (pulse [INTRO])
```

El +2 busca la línea 30; como no hay ninguna con ese número, va a parar al final del programa. Por eso el cursor queda situado justamente detrás de la última línea. Si escribimos un número de línea que no existe, el +2 sitúa el cursor en el lugar en que la línea debería estar si realmente existiese. Ésta puede ser una forma útil de moverse en programas largos; pero tenga cuidado porque también puede ser muy peligroso. En efecto, si la línea ya existe antes de que escribamos el número y pulsemos [INTRO], ciertamente no existirá después.

Para listar un programa en la pantalla basta con escribir:

```
LIST      (pulse [INTRO])
```

A veces se prefiere listar solamente desde cierta línea en adelante (en particular cuando se trabaja con programas largos). Para ello se escribe el número de línea adecuado detrás de la orden LIST.
Escriba:

```
LIST 15     (pulse [INTRO])
```

y compruebe el resultado.

Observe cómo pudimos insertar la línea 15 entre las otras dos al escribir el programa anterior. Esto habría sido imposible si sus números hubiesen sido 1 y 2, en vez de 10 y 20. Por esta razón, siempre es conveniente dejar intervalos prudenciales entre los números de línea.

(Tenga en cuenta que todos los numeros de línea tiene que ser enteros y estar entre 1 y 9999.)

Si en algún momento se da usted cuenta de que no ha dejado suficiente espacio entre los números de línea, puede invocar el menú de edición y renumerar el programa. Para ello pulse la tecla [EDIT] y seleccione la opción **Renumerar** del menú. El programa queda renumerado a partir de la línea 10 y con intervalo de 10 unidades entre líneas sucesivas. Pruébelo y observe cómo cambian los números.

Ahora vamos a utilizar la orden NEW de BASIC. Esta orden borra el programa almacenado en el +2, junto con todas sus variables. Es la orden que se debe dar al ordenador cuando se quiere empezar partiendo de cero. Escriba:

```
NEW
```

y pulse [INTRO]. De ahora en adelante ya no escribiremos 'pulse [INTRO]' cada vez que usted deba pulsar esta tecla, pero no se olvide de hacerlo.

Con el menú de presentación en la pantalla, active BASIC seleccionando la opción **128 BASIC**.

A continuación transcriba cuidadosamente el siguiente programa, que convierte grados Fahrenheit en grados centígrados:

```
10 REM Conversión de temperatura
20 PRINT "Grados F","Grados C"
30 PRINT
40 INPUT "Introduzca grados F",f
50 PRINT f,(f-32)*5/9
60 GO TO 40
```

Observe que, si escribe toda la línea 10 en minúsculas, BASIC sólo convierte en mayúsculas la palabra 'REM', ya que ésta es la única palabra clave que hay en la línea. Además, a pesar de que nosotros hemos escrito **GO TO** en forma de dos palabras separadas por un espacio, usted puede escribirlas juntas (**GOTO**).

Ahora ejecute el programa. Verá que las cabeceras son escritas en la pantalla superior, como consecuencia de la línea 20. Pero ¿qué ha hecho la línea 10? Parece que el +2 la ha ignorado completamente; en efecto, eso es lo que ha hecho. **La palabra REM** de la línea 10 es en realidad una abreviatura de *remark* ('observación') y su único efecto es permitir que usted haga anotaciones al programa. Una sentencia **REM** consiste en la palabra **REM** seguida de cualquier cosa. El ordenador ignora todo lo que se escriba a la derecha de **REM** hasta el final de la línea.

El ordenador ya ha llegado a la orden INPUT de la línea 40 y está esperando que escriba usted un valor para la variable **f** (lo cual se confirma por el hecho de que el cursor está en la pantalla inferior).

Introduzca un número. El +2 muestra el resultado del cálculo y queda a la espera de otro número. Esto se debe a que la instrucción de la línea 60 es GO TO 40, que en castellano se puede leer 'ir a 40'; en otras palabras, 'en vez de salir del programa y detenerte, salta hacia atrás, a la línea 40, y continúa a partir de ella'.

Así pues, introduzca otra temperatura, luego otra,... . Quizá se pregunte usted si la máquina llegará a cansarse de hacer siempre lo mismo; pues no, no se cansa nunca. Para detener el programa debe hacer lo siguiente: en vez de introducir otro número, pulse la tecla **A** en combinación con [SIMB]. Esto hace que aparezca la palabra **STOP**; cuando usted pulse [INTRO], el +2 responderá con el informe:

```
H STOP in INPUT, 40:1
```

Este informe indica que el programa se ha detenido en una instrucción INPUT, que es la primera (**:1**) de la línea 40.

Si desea reanudar la ejecución del programa, escriba:

```
CONTINUE
```

y el +2 le pedirá otro número.

Cuando se le da la orden CONTINUE, el +2 recuerda el número de línea incluido en el último informe que ha emitido (siempre que no fuera **0 OK**) y salta a esa línea, que en nuestro caso es la 40 (la de la orden INPUT).

Detenga nuevamente el programa y cambie la línea 60 por:

```
60 GO TO 31
```

No habrá una diferencia perceptible en la ejecución del programa, porque si el número de línea de una orden GO TO se refiere a una línea inexistente, el salto se efectúa hasta la siguiente línea *posterior* al número especificado. Análogamente ocurre con la orden RUN (de hecho, RUN equivale a RUN 0).

Ahora introduzca números hasta que la pantalla superior empiece a llenarse. Cuando ya esté llena, el +2 desplazará hacia arriba todo el contenido de la pantalla superior para hacer sitio para nuevas líneas, y el encabezamiento se saldrá de la pantalla.

Si lo desea, ya puede detener el programa, tal y como lo hizo antes y activar el editor pulsando INTRO.

Observe la sentencia PRINT de la línea 50. En ella la coma (,) es muy importante.

Las comas sirven para hacer que la impresión comience en el margen izquierdo o en centro de la pantalla, según los casos. Así, en la línea 50 la coma hace que la temperatura centígrada se escriba a partir del centro de la línea.

Por otro lado, punto y coma (;) se utiliza para hacer que el siguiente número o cadena literal se imprima inmediatamente después del precedente.

Otro signo de puntuación que puede usted utilizar en las órdenes PRINT es el apóstrofo ('), el cual hace que lo que se escriba a continuación aparecezca al principio de la línea siguiente. Esto mismo ocurre por defecto (es decir, cuando no se especifica otra cosa) al final de cada orden PRINT . Si desea inhibir el salto a la línea siguiente, puede poner una coma o un punto y coma al final de la sentencia PRINT. Para ver cómo funciona, reemplace sucesivamente la línea 50 por cada una de éstas:

```
50 PRINT f,
50 PRINT f;
50 PRINT f
```

y ejecute el programa cada vez para observar la diferencia.

La línea que termina en coma lo escribe todo en dos columnas; la línea del punto y coma lo coloca todo junto; la línea que no lleva coma ni punto y coma escribe cada número en una nueva línea (esto mismo se conseguiría con PRINT f').

Recuerde siempre la diferencia entre la coma y el punto y coma en las órdenes PRINT y no los confunda con los dos puntos (:), los cuales se utilizan como separadores entre órdenes incluidas en una misma línea; por ejemplo:

```
PRINT f: GO TO 40
```

Ahora escriba estas líneas adicionales:

```
100 REM Este programa recuerda su nombre
110 INPUT n$
120 PRINT "Hola ";n$;"!"
130 GO TO 110
```

Éste programa es independiente del anterior, pero usted puede guardar ambos en el +2 al mismo tiempo. Para ejecutar el nuevo programa dé la orden:

```
RUN 100
```

Puesto que el programa espera que usted introduzca una *cadena literal* (o sea, un carácter o un grupo de caracteres) en vez de un número, escribirá el cursor entre comillas (" ") como recordatorio. Así pues, escriba un nombre y pulse [INTRO].

La próxima vez volverán a aparecer las comillas, pero usted no tiene que utilizarlas si no lo desea. Por ejemplo, pruebe lo siguiente: borre las comillas pulsando dos veces [→] y dos veces [BORR]; después escriba:

```
n$
```

Al no haber comillas, el +2 sabe que tiene que realizar algún cálculo (en este caso, averiguar el valor de la variable literal n$, que será el nombre introducido la vez anterior). De este modo, la sentencia INPUT actúa como LET n$=n$, así que el valor de n$ permanece inalterado.

Cuando quiera detener el programa, borre las comillas, pulse A en combinación con [SIMB] y finalmente pulse [INTRO].

Ahora examinemos la instrucción RUN 100, que provoca el salto a la línea 100 e inicia la ejecución del programa a partir de ella. Usted podría preguntarse: "¿cuál es la diferencia entre RUN 100 y GO TO 100?" Pues bien, RUN 100 empieza por borrar la pantalla y todas las variables, y luego ya realiza la misma función que GO TO 100. En cambio, GO TO 100 no borra nada. Hay ocasiones en que se necesita ejecutar un programa sin borrar las variables; en tales casos es necesario usar GO TO, porque RUN podría ser desastrosa. Por consiguiente, no conviene que se acostumbre a escribir RUN sistemáticamente para poner en marcha los programas.

Otra diferencia obvia consiste en que se puede escribir RUN sin número de línea (y entonces la ejecución comienza a partir de la primera línea del programa), mientras que GO TO siempre tiene que ir seguida de un número de línea.

Tanto este programa como el de conversión de la temperatura se detienen solamente cuando usted pulsa [SIMB] A en la línea de entrada. A veces se escribe por error un programa que no se detiene por sí mismo ni puede ser interrumpido por este procedimiento. Por ejemplo, escriba:

```
200 GO TO 200
RUN 200
```

Aunque la pantalla está en blanco, el programa sí está funcionando, ejecutando la línea 200 una y otra vez. Aparentemente va a continuar así para siempre, a menos que desenchufe el cable o pulse el botón **RESET**. Sin embargo, hay un remedio menos drástico: pulsar la tecla [BREAK]. El programa se detiene y el ordenador emite el informe:

```
L BREAK into program, 200:1
```

Cada vez que termina de ejecutar una sentencia, el ordenador comprueba si está pulsada la tecla [BREAK]; si lo está, interrumpe el programa. También se puede usar esta tecla para interrumpir las operaciones con el magnetófono, la impresora y algunos otros periféricos que pueden ser conectados al +2. En estos casos el informe es distinto:

```
D BREAK - CONT repeats
```

En esta situación (y también en muchas otras), la instrucción CONTINUE repite la sentencia en la que el programa fue inetrrumpido y continúa directamente con la siguiente.

Ejecute otra vez el programa del 'nombre' y, cuando le pida la entrada, escriba:

```
n$      (después de borrar las comillas)
```

Puesto que n$ es una variable que ho ha sido definida, el ordenador emite el siguiente mensaje de error:

```
2 Variable not found, 110:1      ('Variable no encontrada')
```

Defina la variable escribiendo la orden:

```
LET n$="cara de pez"
```

(que produce el informe 0 OK, 0:1) y luego escriba:

```
CONTINUE
```

Como puede comprobar, ahora ya se puede usar n$ como dato de entrada sin ningún problema.

En este caso, CONTINUE provoca el salto a la orden INPUT de la línea 110; ignora el informe producido por la sentencia LET porque era OK y salta a la orden aludida en el informe anterior, la 110. Esta característica puede ser muy útil, pues permite «reparar» un programa que se ha detenido a causa de un error y continuar (CONTINUE) a partir de donde se produjo la interrupción.

Como dijimos antes, el informe L BREAK into program es especial porque, tras él, CONTINUE no repite la orden en la que el programa se detuvo.

Ya hemos visto las sentencias PRINT, LET, INPUT, RUN, LIST, GO TO, CONTINUE, NEW y REM. Todas ellas pueden ser utilizadas como órdenes directas o en líneas de programa. (Esto es válido para casi todas las órdenes del Spectrum BASIC; sin embargo, no es frecuente incluir RUN, LIST, CONTINUE y NEW en las líneas de programa.)

Ejercicios

1. Ponga una sentencia LIST en un programa, de forma que el programa se liste a sí mismo al ejecutarlo.

2. Escriba un programa que pida los precios de los artículos y escriba el importe del IVA (12 %). Utilice sentencias PRINT mediante las que el ordenador explique qué va a hacer y pida los precios con extravagante amabilidad. Modifique luego el programa de forma que éste pregunte el porcentaje del impuesto (con lo cual el programa valdrá también para artículos exentos de IVA y para los que estén sujetos a porcentajes diferentes).

3. Elabore un programa que escriba la suma actualizada de los números que el usuario vaya introduciendo. (*Sugerencia*: cree una variable que se llame total, dándole inicialmente el valor 0; utilice otra variable que se llame número; en cada vuelta, asigne valor a la variable número en una instrucción INPUT y súmela a total; escriba los valores de ambas y pase a la siguiente vuelta.)

4. ¿Cuál sería el efecto de CONTINUE y NEW dentro de en un programa? ¿Se le ocurre alguna aplicación práctica?

Parte 3
Decisiones

Temas tratados:

CLS, IF, STOP
=, <, >, <=, >=, <>

Todos los programas que hemos visto hasta ahora eran bastante predecibles: obedecían sucesivamente las instrucciones y volvían al principio. Esto no es lo más útil que puede hacer el ordenador por nosotros; en la práctica, lo que esperamos de un ordenador es que sea capaz de tomar decisiones y obrar en consecuencia. En BASIC, la instrucción que lleva a cabo la toma de decisiones tiene la siguiente forma: 'IF (si) algo es cierto (o falso) THEN (entonces) haz tal cosa'.

Veamos un ejemplo. Dé la orden NEW para borrar el programa anterior de la memoria del +2, active 128 BASIC y transcriba y ejecute el siguiente programa (que, evidentemente, ha sido ideado para que jueguen dos personas):

```
10 REM  Adivinar un número
20 INPUT "Introduzca un número secreto",a: CLS
30 INPUT "Adivine el número",b
40 IF b=a THEN PRINT "Correcto!": STOP
50 IF b<a THEN PRINT "Ese es demasiado pequeño; vuelva a intentarlo"
60 IF b>a THEN PRINT "Ese es demasiado grande; vuelva a intentarlo"
70 GO TO 30
```

Observe que la orden CLS (en la línea 20) significa 'borrar la pantalla' (en inglés, clear screen). Nosotros la hemos incluido en este programa para impedir que la otra persona vea el número secreto una vez que éste ha sido introducido.

Como puede ver, la sentencia IF tiene la forma:

IF *condición* THEN *xxx*

donde *xxx* representa una orden (o una secuencia de órdenes separadas por signos de dos puntos). La *condición* es algo que el ordenador tiene que evaluar y que puede dar como resultado 'verdadero' o 'falso'. Si resulta 'verdadero', las sentencias del resto de la línea (posteriores a THEN) serán ejecutadas; de lo contrario, el ordenador las ignora y salta a la siguiente instrucción.

Las condiciones más sencillas consisten en la comparación de dos números o dos cadenas. Por ejemplo, se puede comparar dos números para averiguar si son iguales o si

uno es mayor que el otro, o comparar dos cadenas para ver cuál está antes en el orden alfabético. En las comparaciones se utiliza los símbolos =, <, >, <=, >= y <>, que son los denominados *operadores de relación*:

```
=   se lee 'es igual a'
<   se lee 'es menor que'
>   se lee 'es mayor que'
<=  se lee 'es igual o menor que'
>=  se lee 'es igual o mayor que'
<>  se lee 'es distinto de'
```

En el programa que acabamos de escribir, la línea 40 compara a con b. Si son iguales, el programa es interrumpido por la orden STOP. El mensaje que aparece en la pantalla inferior es:

```
9 STOP statement, 40:3
```

el cual indica que la tercera sentencia (es decir, la orden STOP) de la línea 40 hizo que el programa se detuviera.

La línea 50 averigua si b es menor que a; la línea 60, si b es mayor que a. Si una de estas condiciones es verdadera, el programa escribe el comentario apropiado y continúa en la línea 70, desde la cual salta a la 30, donde se reinicia el proceso.

Para terminar, observe que en algunas versiones de BASIC (no en el Spectrum BASIC) la sentencia IF puede tener la forma:

IF *condición* THEN *número de línea*

En Spectrum BASIC, esto equivale a:

IF *condición* THEN GO TO *número de línea*

Ejercicio

1. Pruebe el siguiente programa:

```
10 LET a=1
20 LET b=1
30 IF a>b THEN PRINT a;" es mayor"
40 IF a<b THEN PRINT b;" es mayor"
```

Antes de ejecutarlo, trate de adivinar qué va a aparecer en la pantalla.

Parte 4
Bucles

Temas tratados:

FOR, NEXT
TO, STEP

Supongamos que usted desea escribir un programa que capte cinco números por el teclado y los sume.

Un método podría ser el siguiente (no copie esto, a menos que quiera hacer prácticas de mecanografía):

```
 10 LET total=0
 20 INPUT a
 30 LET total=total+a
 40 INPUT a
 50 LET total=total+a
 60 INPUT a
 70 LET total=total+a
 80 INPUT a
 90 LET total=total+a
100 INPUT a
110 LET total=total+a
120 PRINT total
```

Este método es pésimo. El programa resulta más o menos manejable cuando se trata de sumar cinco números, pero imagínese que hubiera que sumar diez (o cien).

Lo que vamos a hacer es usar una variable para contar hasta 5 y luego detener el programa (éste sí debe copiarlo):

```
10 LET total=0
20 LET contador=1
30 INPUT a
40 REM contador indica el número de veces que se ha captado el valor de a hasta
el momento
50 LET total=total+a
60 LET contador=contador+1
70 IF contador<=5 THEN GO TO 30
80 PRINT total
```

Observe qué fácil sería ahora modificar la línea 70 para que el programa sumara diez números, o incluso cien.

Este método es tan útil que hay dos órdenes especiales para hacerlo más fácil: la orden FOR y la orden NEXT, las cuales siempre han de ser utilizadas conjuntamente. Introduciéndolas, el programa anterior se convierte en:

```
10 LET total=0
20 FOR c=1 TO 5
30 INPUT a
40 REM c es el número de veces que se ha captado el valor de a hasta ahora
50 LET total=total+a
60 NEXT c
80 PRINT total
```

(Para obtener este programa a partir del anterior, basta con editar las líneas 20, 40 y 60 y borrar la línea 70.)

Observe que hemos cambiado contador por c. Esto ha sido necesario porque el nombre de la variable de control de un bucle FOR...NEXT tiene que constar de una sola letra.

El mecanismo de este programa es el siguiente: c va tomando sucesivamente los valores 1 (*valor inicial*), 2, 3, 4 y 5 (*límite*), y para cada uno de ellos el ordenador ejecuta las líneas 30, 40 y 50. Después, cuando c ya ha recorrido sus cinco valores, se ejecuta la línea 80.

Ahora intente resolver el ejercicio 2 del final de esta Parte 4, que hace referencia al programa anterior.

Un refinamiento adicional de la estructura FOR...NEXT consiste en que la variable de control no tiene que crecer necesariamente de uno en uno, sino que usted puede cambiar ese 1 por cualquier otro número sin más que incluir la cláusula STEP (paso) en la orden FOR. La forma más general de una orden FOR es, pues,

FOR *variable de control* = *valor inicial* TO *límite* STEP *paso*

donde la *variable de control* es una sola letra, y el *valor inicial*, el *límite* y el *paso* son «expresiones» (es decir, cualquier cosa que el +2 pueda evaluar y cuyo valor sea un número: constantes numéricas, sumas de números, variables numéricas, etc.). Pues bien, si reemplazamos la línea 20 del programa por:

```
20 FOR c=1 TO 5 STEP 3/2
```

la variable de control se verá incrementada en 3/2 cada vez que se ejecute el bucle FOR. Observe que podríamos haber puesto STEP 1.5, o haber asignado el valor del paso a una variable (por ejemplo, p) y luego haber especificado STEP p.

Con esta modificación de arriba, la variable c tomará los valores 1, 2.5 y 4. Observe que no hay por qué limitarse a números enteros y que, por otra parte, no es necesario

que la variable de control llegue a coincidir exactamente con el valor del límite: el bucle se reitera mientras el valor de la variable de control sea igual o menor que el del límite.

Intente resolver el ejercicio 3 del final de esta Parte 4, que hace referencia al programa anterior.

Los valores del paso pueden ser negativos en vez de positivos. Pruebe el siguiente programa, que escribe los números del 1 al 10 en orden inverso. (Acuérdese siempre de dar la orden NEW antes de empezar a escribir un programa nuevo.)

```
10 FOR n=10 TO 1 STEP -1
20 PRINT n
30 NEXT n
```

Dijimos antes que el programa continúa realizando bucles mientras la variable de control sea igual o menor que el límite, pero eso sólo es válido cuando no se incluye la cláusula STEP o cuando el valor del *paso* es positivo. Si el paso es negativo, la regla es como sigue: el bucle se repite mientras la variable de control sea igual o mayor que el límite.

Intente resolver los ejercicios 4 y 5 del final de esta Parte 4, que hacen referencia al programa anterior.

Hay que tener cuidado cuando se utiliza dos bucles FOR...NEXT anidados (es decir, uno dentro de otro). Pruebe el siguiente programa, que escribe todos los valores posibles de las fichas de dominó:

```
10 FOR m=0 TO 6
20 FOR n=0 TO m
30 PRINT m;":";n;" ";
40 NEXT n
50 PRINT
60 NEXT m
```

bucle *n* (líneas 20 a 40)

bucle *m* (líneas 10 a 60)

Observe que el bucle *n* está completamente dentro del bucle *m*. Esto significa que el anidamiento es correcto.

Lo que siempre se debe evitar es tener dos bucles FOR...NEXT solapados, como ocurre en el siguiente programa:

```
 5 REM  Este programa es incorrecto
10 FOR m=0 TO 6
20 FOR n=0 TO m
30 PRINT m;":";n;" ";
40 NEXT m
50 PRINT
60 NEXT n
```

bucle *m* (líneas 20 a 40)

bucle *n* (líneas 10 a 60)

En resumen, si un programa contiene dos bucles FOR...NEXT, éstos deben estar, o bien uno dentro del otro, o bien completamente separados.

Otra cosa que hay que evitar es saltar al interior de un bucle FOR...NEXT desde fuera de él. La variable de control sólo se prepara correctamente cuando se ejecuta su sentencia FOR; si el ordenador no ejecuta una sentencia FOR, la sentencia NEXT lo dejará completamente confundido. La consecuencia más probable es el mensaje de error NEXT without FOR ('NEXT sin FOR'), o bien Variable not found ('Variable no encontrada').

Nada impide utilizar un bucle FOR...NEXT en una orden directa. Por ejemplo, pruebe la siguiente:

```
FOR m=0 TO 10: PRINT m: NEXT m
```

Hay una forma (un tanto artificial) de repetir automáticamente una orden directa. (Recuerde que GO TO necesita un número de línea, y eso es lo que una orden directa no puede tener). Por ejemplo,

```
FOR m=0 TO 1 STEP 0: INPUT a: PRINT a: NEXT m
```

El hecho de que el paso sea 0 hace que la orden se repita indefinidamente.

Pero este método no es muy recomendable: si se produce un error se pierde la orden y hay que volver a escribirla; además, CONTINUE no funcionará en este caso.

Ejercicios

1. Asegúrese de que entiende perfectamente que una variable de control no sólo tiene un nombre y un valor, como una variable ordinaria, sino además un límite, un paso y una conexión con la sentencia FOR correspondiente. Por otra parte, cuando se ejecuta la sentencia FOR, toda esta información se encuentra disponible y es suficiente para que la sentencia NEXT sepa cómo tiene que modificar el valor de la variable, si debe o no repetir el bucle y, de ser así, a qué línea tiene que saltar.

2. Ejecute el tercer programa de esta sección y luego dé la orden:

   ```
   PRINT c
   ```

   ¿Por qué la respuesta es 6 y no 5?

   (*Solución*: la orden NEXT de la línea 60 es ejecutada cinco veces, y en cada una de ellas se suma 1 a c. La última vez c se convierte en 6; por eso la orden NEXT decide no repetir el bucle, sino continuar, ya que c ha sobrepasado el límite.)

   ¿Qué ocurre si ponemos STEP 2 al final de la línea 20?

3. Modifique el tercer programa, de forma que, en lugar de sumar automáticamente los cinco números, pregunte al usuario cuántos números quiere sumar. Al ejecutar este programa, ¿qué ocurriría si usted respondiese con un 0 (lo que significaría que no quiere sumar ningún número)? ¿Cree que esto le causaría algún problema al +2, aun estando claro lo que usted quiere decir?

4. En la línea 10 del cuarto programa de esta sección, cambie 10 por 100 y ejecute el programa. En la pantalla aparecerán los números del 100 al 79 y luego, en la pantalla inferior, podrá ver el mensaje scroll? ('¿desplazar?'). El +2 le da así la oportunidad de observar los números antes de que sean desplazados hacia arriba. Si pulsa N, BREAK o la barra espaciadora, el programa se detendrá con el mensaje D BREAK - CONT repeats. Si pulsa cualquier otra tecla, el programa escribirá otras 22 líneas y volverá a preguntarle si desea otro desplazamiento vertical.

5. Borre la línea 30 del cuarto programa. Cuando ejecute la nueva versión, el programa escribirá el primer número y se detendrá con el mensaje 0 OK. Si a continuación escribe:

   ```
   NEXT n
   ```

   el programa ejecutará una vez el bucle y escribirá el número siguiente.

Parte 5
Subrutinas

Temas tratados:

GO SUB, RETURN

A veces nos encontramos con que diferentes partes del programa tienen que realizar funciones bastante similares, y esto fácilmente puede llevarnos a escribir las mismas instrucciones varias veces con diferente número de línea. Afortunadamente, en BASIC disponemos de un recurso que nos permite evitar este despilfarro: podemos escribir las instrucciones una sola vez, en lo que denominamos una *subrutina*, y luego invocar la subrutina siempre que sea necesario.

El control de las subrutinas se realiza con dos sentencias: GO SUB y RETURN. Para invocar una subrutina se da una orden del tipo:

```
GO SUB xxx
```

donde *xxx* es el número de la primera línea de la subrutina. El mecanismo es similar al de GO TO *xxx*, con la diferencia de que el ordenador recuerda dónde estaba la sentencia GO SUB y así sabe a dónde tiene que volver cuando termina de ejecutar la subrutina.

(Por si le interesa saberlo, el +2 «recuerda» en qué punto del programa estaba la sentencia GO SUB porque almacena la *dirección de retorno* en la *pila* de GO SUB).

Cuando el ordenador encuentra la orden

```
RETURN
```

sabe que ahí termina la subrutina; entonces recuerda dónde estaba la sentencia GO SUB (tomando de la pila la dirección de retorno) y continúa a partir de la sentencia siguiente.

Como ejemplo, echemos un vistazo de nuevo al programa de adivinar números. Reescríbalo de la siguiente forma:

```
 10 REM  Programa de adivinar números, reorganizado
 20 INPUT "Introduzca un número secreto",a: CLS
 30 INPUT "Adivine el número",b
 40 IF b=a THEN PRINT "Correcto!": STOP
 50 IF b<a THEN GO SUB 100
 60 IF b>a THEN GO SUB 100
 70 GO TO 30
100 PRINT "Vuelva a intentarlo"
110 RETURN
```

La sentencia GO TO 30 de la línea 70, y la sentencia STOP del programa siguiente, son muy importantes; si no fuera por ellas, los programas entrarían en las subrutinas y se produciría un error (7 RETURN without GO SUB, 'RETURN sin GO SUB') al llegar a la sentencia RETURN.

El siguiente programa utiliza una subrutina (líneas 100 a 150) que escribe la tabla de multiplicar correspondiente al *parámetro* n. La orden GO SUB 100 que invoca la subrutina puede estar en cualquier lugar del programa. Cuando el ordenador encuentra la orden RETURN en línea 150 de la subrutina, el control retorna al programa principal, que continúa funcionando a partir de la sentencia siguiente a aquélla en que se encontraba el GO SUB original. Al igual que ocurría con GO TO, se puede escribir GO SUB en una sola palabra: GOSUB.

```
 10 REM  Tabla de multiplicar del 2, el 5, el 10 y el 11
 20 LET n=2: GO SUB 100
 30 LET n=5: GO SUB 100
 40 LET n=10: GO SUB 100
 50 LET n=11: GO SUB 100
 60 STOP
 70 REM  Fin del programa principal, comienzo de la subrutina
100 PRINT "Tabla de multiplicar por ";n
110 FOR v=1 TO 9
120 PRINT v;" x ";n;" = ";v*n
130 NEXT v
140 PRINT
150 RETURN
```

Una subrutina puede invocar a otra, o incluso a sí misma. (Una rutina que se llama a sí misma es una *rutina recursiva*).

Parte 6
Datos en los programas

Temas tratados:

READ, DATA, RESTORE

En algunos de los programas anteriores hemos visto que la información (es decir, los datos) puede ser captada por el programa mediante la sentencia INPUT. En ocasiones esto resulta muy tedioso, especialmente cuando buena parte de la información que hay que introducir es la misma en todas las ejecuciones del programa. Se puede ahorrar un tiempo considerable utilizando las sentencias READ, DATA y RESTORE. Veamos un ejemplo:

```
10 READ a,b,c
20 PRINT a,b,c
30 DATA 1,2,3
```

Una sentencia READ consiste en la palabra clave READ seguida de una lista de nombres de variables separados entre sí por comas. Funciona de modo bastante parecido a INPUT, con la ventaja de que, en vez de hacerle a usted escribir los valores que hay que asignar a las variables, el ordenador los lee en la sentencia DATA.

Cada sentencia DATA es una lista de expresiones (numéricas o literales) separadas por comas. Las sentencias DATA pueden estar en cualquier lugar del programa, ya que el +2 las ignora completamente, excepto cuando está ejecutando una sentencia READ. Podemos imaginarnos las expresiones de todas las sentencias DATA del programa como si estuvieran colocadas todas juntas, formando una larga lista: la lista de datos. La primera vez que el +2 lee (con READ) un valor, consulta la primera expresión de la lista de datos; a la siguiente vez, lee la segunda; y así sucesivamente, según va encontrando instruciones READ, sigue su camino a lo largo de la lista de datos. (Si trata de leer más allá del fin de la lista, se produce un error.)

Observe que es una pérdida de tiempo poner instrucciones DATA en una orden directa, ya que READ no las encontrará. Las sentencias DATA tienen que estar en líneas de programa.

Vamos a ver cómo funciona todo esto en el programa que acaba de transcribir. La línea 10 le pide al ordenador que lea tres datos y los asigne a las variables a, b y c. La línea 20 le pide que escriba (PRINT) los valores de esas variables. La sentencia DATA de la línea 30 proporciona los valores de a, b y c para que los lea la línea 10.

La información contenida en una sentencia DATA puede ser leída por un bucle FOR...NEXT. Escriba lo siguiente:

```
10 FOR n=1 TO 6
20 DATA 2,4,6,8,10,12
30 READ d
40 PRINT d
50 NEXT n
```

Los dos programas anteriores han demostrado que las sentencias DATA pueden estar en cualquier lugar (antes o después de la instrucción READ).

Al ejecutar este último programa, la sentencia READ avanza por la lista de datos en cada pasada por el bucle FOR...NEXT.

Las sentencias READ pueden también asignar valores a variables literales. Por ejemplo,

```
10 READ d$
20 PRINT "La fecha es",d$
30 DATA "20 de diciembre de 1986".
```

No siempre hay que leer las sentencias DATA en orden, de la primera a la última; de hecho, podemos saltar de una sentencia DATA a otra mediante la orden RESTORE. La forma de esta orden es:

```
RESTORE xxx
```

donde *xxx* es el número de la línea en que se encuentra la sentencia DATA que debe ser leída. La orden RESTORE sin número de línea hace que el *puntero de datos* apunte hacia la primera sentencia DATA del programa.

Transcriba y pruebe el siguiente programa:

```
 10 DATA 1,2,3,4,5
 20 DATA 6,7,8,9
 30 GO SUB 110
 40 GO SUB 110
 50 GO SUB 110
 60 RESTORE 20
 70 GO SUB 110
 80 RESTORE
 90 GO SUB 110
100 STOP
110 READ a,b,c
120 PRINT a'b'c
130 PRINT
140 RETURN
```

La orden GO SUB 110 invoca una subrutina que lee los siguientes tres elementos de DATA y después los escribe (PRINT). Observe cómo la orden RESTORE decide qué elementos son leídos en cada caso.

Borre la línea 60 y ejecute este programa de nuevo para ver qué ocurre.

Parte 7
Expresiones

Temas tratados:

Operaciones: +, −, *, /
Expresiones, notación científica, nombres de variables

Ya ha visto algunas de las formas en las que el +2 realiza cálculos con números. Además de las cuatro operaciones aritméticas, +, −, * y / (recuerde que en BASIC * es el símbolo de la multiplicación y / el de la división), sabe encontrar el valor de una variable, dado su nombre.

La sentencia

```
LET impuesto=suma*15/100
```

es un ejemplo de cómo se puede combinar los cálculos. Una combinación tal como suma*15/100 es lo que se llama *expresión*. Por consiguiente, una expresión es una forma abreviada de decirle al +2 cómo debe realizar varios cálculos, uno a continuación de otro. En nuestro ejemplo, la expresión suma*15/100 significa: 'busca el valor de la variable llamada *sumas*', multiplícalo por 15 y divídelo por 100'.

En la parte 30 de este capítulo daremos la lista completa de las prioridades de las operaciones matemáticas (y lógicas).

En las expresiones que contienen *, /, + o −, la multiplicación y la división son prioritarias con respecto a la suma y la resta (es decir, el ordenador las realiza antes que la suma y la resta). La multiplicación y la división tienen la misma prioridad una que la otra, lo que significa que el +2 las realiza en el orden en que aparezcan en la expresión (de izquierda a derecha). Las siguientes operaciones que realiza el +2 son la suma y la resta; como también tienen el mismo nivel de prioridad, el ordenador las calcula de izquierda a derecha.

Así pues, en la expresión 8−12/4+2*2 la primera operación que se efectúa es la división 12/4, cuyo resultado es 3, por lo que la expresión equivale a 8−3+2*2.

La siguiente operación que se lleva a cabo es la multiplicación 2*2, que da 4, así que la expresión se convierte entonces en 8−3+4.

El siguiente paso es restar 8−3, que da 5; la expresión pasa a ser 5+4. Finalmente, se realiza la suma y el resultado es 9.

Para comprobarlo, dé la orden

```
PRINT 8−12/4+2*2
```

Se puede cambiar la prioridad de los cálculos dentro de una expresión mediante el uso adecuado de los paréntesis. Los cálculos que van entre paréntesis se realizan antes que todos los demás; por lo tanto, si en la expresión anterior queremos que se empiece por calcular la suma 4+2, basta con escribirla entre paréntesis. Compruébelo con la orden:

```
PRINT 8-12/(4+2)*2
```

El resultado es ahora 4 en lugar de 9.

Las expresiones son útiles por el hecho de que, siempre que el +2 esté esperando un número, nosotros podemos darle una expresión en su lugar y él hallará la respuesta.

También se puede «sumar» cadenas (o variables literales) en una expresión. Por ejemplo,

```
10 LET a$="arroz"
20 LET b$="leche"
30 PRINT a$;" con ";b$
```

Ha llegado el momento de que le digamos qué es lo que puede y lo que no puede usar como nombre de variable. Como ya sabe, el nombre de una variable literal tiene que ser una sola letra seguida de $, y el nombre de la variable de control de los bucles FOR...NEXT también tiene que consistir en una sola letra. Pero los nombres de las variables numéricas ordinarias son mucho más libres; pueden constar de dígitos o letras cualesquiera, siempre que el primer carácter sea una letra. También se puede introducir espacios en los nombres de las variables, para hacerlos más legibles, aunque el +2 los ignorará a todos los efectos, salvo en los listados. Además, es indiferente escribir los nombres en mayúsculas o en minúsculas. No obstante, hay algunas restricciones acerca de los nombres de variables: no pueden coincidir con las palabras clave y, en general, si un nombre de variable contiene una palabra clave con espacios a los lados, BASIC no lo aceptará.

He aquí unos cuantos ejemplos de nombres de variable válidos:

```
x
cualquier cosa
t42
este nombre no es conveniente porque es demasiado largo
tobeornottobe
espacios y mayusculas mezclados
EspaciosyMayusculasMezclados
```

(Observe que los dos últimos nombres son considerados iguales y se refieren a la misma variable.)

Los siguientes nombres de variable no son válidos:

pi	(PI es una palabra clave)
2001	(comienza con dígito)
Albany, New York	(contiene NEW entre espacios)
to be or not to be	(TO, OR y NOT son palabras clave de BASIC)
3osos	(comienza con un dígito)
M*A*S*H	(* no es letra ni dígito)
M-30	(- no es letra ni dígito)

Los valores numéricos pueden ser representados por un número y un exponente (en *notación científica*). Pruebe las siguientes órdenes:

```
PRINT 2.34e0
PRINT 2.34e1
PRINT 2.34e2
```

y así sucesivamente hasta

```
PRINT 2.34e15
```

PRINT sólo puede escribir los números con ocho cifras significativas. Pruebe la siguiente orden:

```
PRINT 4294967295, 4294967295-429e7
```

Esto demuestra que el ordenador guarda los dígitos de 429467295, aunque no es capaz de mostrarlos todos de una vez.

El +2 trabaja en *aritmética de punto flotante*, lo que significa que guarda por separado los dígitos del número (la *mantisa*) y la posición del punto decimal (el *exponente*). Este método no siempre da resultados exactos, ni siquiera con números enteros. Escriba:

```
PRINT 1e10+1-1e10,1e10-1e10+1
```

Los números se guardan con unos nueve dígitos y medio de precisión, de modo que 1e10 es demasiado grande como para ser almacenado con precisión absoluta. La inexactitud (en este caso cerca de 2) es mayor que 1, y por tanto los números 1e10 y 1e10+1 le parecen iguales al ordenador.

Un ejemplo aún más peculiar es:

```
PRINT 5e9+1-5e9
```

Aquí la inexactitud de 5e9 es sólo de alrededor de 1, y el 1 que debe ser sumado se redondea hasta 2. Los números 5e9+1 y 5e9+2 le parecen iguales al ordenador. El número completo más grande que se puede almacenar con completa exactitud es 4294967294.

La cadena "", que no tiene caracteres, se llama cadena vacía o cadena nula. Recuerde que los espacios son significativos y que una cadena vacía no es lo mismo que una que sólo contenga espacios.

Pruebe

```
PRINT "Leiste "El Pais" ayer?"
```

Al pulsar [INTRO] aparece el cursor parpadeante en rojo, que, como sabemos, indica la presencia de un error en algún lugar de la línea. Cuando el +2 encuentra las comillas al principio de "El Pais", da por supuesto que marcan el final de la cadena "Leiste", y entonces no sabe qué significa El Pais.

Hay un recurso especial para evitar esto: cuando necesite que las comillas formen parte de una cadena, escríbalas repetidas. Por ejemplo,

```
PRINT "Leiste ""El Pais"" ayer?"
```

Como puede ver en la pantalla, las comillas sólo aparecen una vez (pero usted tiene que escribirlas dos veces para que sean reconocidas).

Parte 8
Cadenas literales

Temas tratados:

Disección de cadenas, TO

Dada una cadena, una *subcadena* de ella consiste en varios de sus caracteres tomados secuencialmente. Así, "cadena" es una subcadena de "cadena mayor", pero "c mayor" y "caneda" no lo son.

Existe una notación específica para describir subcadenas que puede ser aplicada a expresiones literales cualesquiera. Su forma general es:

expresión literal (*principio* TO *fin*)

Así, por ejemplo

```
"abcdef"(2 TO 5)
```

es igual a bcde.

Si se omite el *principio*, BASIC toma por defecto el 1; si se omite el *fin*, se toma la longitud total de la cadena. De este modo:

```
"abcdef"( TO 5)    es igual a    abcde
"abcdef"(2 TO )    es igual a    bcdef
"abcdef"( TO )     es igual a    abcdef
```

Esta última expresión se puede escribir también en la forma "abcdef"().

Hay una forma ligeramente diferente en la que se omite el TO y sólo se escribe un número:

```
"abcdef"(3)    equivale a "abcdef"(3 TO 3)    que es igual a c
```

Aunque normalmente el *principio* y el *fin* deben hacer referencia a caracteres que realmente existan en la cadena, esta regla está supeditada a otra: si el *principio* es mayor que el *fin*, el resultado es la cadena vacía. Así,

```
"abcdef"(5 TO 7)
```

da el error 3 Subscript wrong ('Subíndice erróneo'), porque la cadena sólo tiene 6 caracteres y el valor 7 es, por consiguiente, demasiado grande; pero

```
"abcdef"(8 TO 7)
```

y

```
"abcdef"(1 TO 0)
```

son iguales a la cadena vacía y, por lo tanto, están permitidas.

El *principio* y el *fin* no deben ser negativos; si lo son se produce el error B integer out of range ('Entero fuera de margen'). El siguiente programa ilustra algunas de estas reglas:

```
10 LET a$="abcdef"
20 FOR n=1 TO 6
30 PRINT a$(n TO 6)
40 NEXT n
```

Escriba NEW cuando haya probado este programa y luego transcriba el siguiente:

```
10 LET a$="1234567890"
20 FOR n=1 TO 10
30 PRINT a$(n TO 10),a$((11-n) TO 10)
40 NEXT n
```

Si la cadena no es constante, sino variable, también podemos asignar valores a sus subcadenas. Por ejemplo, escriba:

```
LET a$="Me gusta mi Sinclair"
```

y después

```
LET a$(13 TO 20)="Amstrad*****"
```

Ahora dé la orden

```
PRINT a$
```

Observe que la subcadena a$(13 TO 20) sólo tiene ocho caracteres y que por eso en la segunda sentencia de asignación sólo se utiliza los ocho primeros caracteres de Amstrad*****. Los cuatro restantes, ****, son ignorados. Ésta es una característica de la asignación de valores a subcadenas: la subcadena debe tener exactamente la misma longitud antes que después de la asignación. Para ello, si la cadena asignada es demasiado larga, BASIC la recorta por la derecha; si es demasiado corta, la completa con espa-

cios por la derecha. Esto es algo parecido a lo que hacía Procrustes, un posadero que siempre adaptaba los huéspedes al tamaño de la cama: o los estiraba en el potro o les cortaba los pies.

Si ahora hace

```
LET a$()="Hola!"
```

y da la orden

```
PRINT a$;"."
```

comprobará que BASIC ha considerado a$ como subcadena de a$ y que ha completado su nuevo valor rellenando con espacios por la derecha.

En cambio,

```
LET a$="Hola!"
```

asigna un valor totalmente nuevo a a$, sin respetar la longitud anterior.

Las expresiones literales complicadas necesitarán estar entre paréntesis para poder extraer subcadenas de ellas. Por ejemplo,

"abc"+"def"(1 TO 2)	es igual a "abcde"
("abc"+"def")(1 TO 2)	es igual a "ab"

Ejercicio

1. Diseñe un programa que escriba el día de la semana recurriendo a la extracción de subcadenas. (*Sugerencia*. Forme la cadena con *LunMarMieJueVieSabDom*.)

Parte 9
Funciones

Temas tratados:

DEF
LEN, STR$, VAL, SGN, ABS, INT, SQR
FN

Para explicar en qué consisten las funciones, tomemos como ejemplo la máquina de hacer zumo. Usted introduce una pieza de fruta por un extremo, pone en marcha la máquina y por el otro lado sale el zumo. De una naranja sale zumo de naranja; de una pera, zumo de pera; y de una manzana, zumo de manzana.

Las funciones son como las máquinas de hacer zumo, aunque hay una diferencia: trabajan con números y cadenas en vez de con frutas. Usted suministra un valor (el *argumento*), lo *procesa* realizando con él algunos cálculos y, finalmente, obtiene otro valor: el *resultado*.

Entrada de fruta	→ Máquina de hacer zumo	→ Zumo
Argumento	→ Función	→ Resultado

Diferentes argumentos producen diferentes resultados; si el argumento es completamente inadecuado, la función no puede procesarlo y BASIC emite un mensaje de error.

Al igual que en la industria puede haber diferentes máquinas para fabricar diferentes productos (una para zumos, otra para manteles, una tercera para guantes de goma, etc.), en el ordenador necesitamos distintas funciones para realizar los diferentes cálculos. Cada función tiene un nombre que la distingue de todas las demás.

Para utilizar una función se escribe su nombre seguido del argumento; BASIC calcula la función cuando evalúa la expresión en la que aquélla inerviene.

Por ejemplo, hay una función llamada LEN que da como resultado la longitud de una cadena. El argumento de LEN es la cadena cuya longitud se desea hallar. Por ejemplo, si damos la orden

```
PRINT LEN "Spectrum +2"
```

el ordenador responde con el número 11, es decir, el número de caracteres (incluidos los espacios) de que consta la cadena Spectrum +2.

Si mezclamos funciones y operaciones en una sola expresión, aquéllas son evaluadas antes que éstas. Pero, por otra parte, podemos alterar este orden utilizando paréntesis.

Por ejemplo, las dos expresiones siguientes sólo se distinguen en los paréntesis y, precisamente por ello, los cálculos son realizados en un orden completamente distinto en cada caso (aunque los resultados finales son idénticos):

```
LEN "Pepe " + LEN "y Manolo"
 ↓
5+LEN "y Manolo"
 ↓
5+8
 ↓
13
```

```
LEN ("Pepe " + "y Manolo")
 ↓
LEN ("Pepe y Manolo")
 ↓
LEN "Pepe y Manolo"
 ↓
13
```

Veamos unas cuantas funciones más:

STR$ convierte números en cadenas. Su argumento es un número; su resultado es la cadena que aparecería en la pantalla si el número hubiera sido escrito por PRINT. Observe que el nombre de la función termina en un signo $ para indicar que el resultado es una cadena. Por ejemplo,

```
LET a$=STR$ 1e2
```

produce exactamente el mismo efecto que

```
LET a$="100"
```

Otro ejemplo:

```
PRINT LEN STR$ 100.0000
```

da como resultado 3, ya que STR$ 100.0000 es igual a la cadena 100, cuya longitud es 3 caracteres.

VAL es la función inversa de STR$: convierte cadenas en números. Por ejemplo,

```
VAL "3.5"
```

produce el número 3.5.

Decimos que VAL es la función inversa de STR$ porque si tomamos un número cualquiera, le aplicamos STR$ y después VAL, el resultado final es el número original.

En cambio, si tomamos una cadena, le aplicamos VAL y luego STR$, no siempre obtendremos la cadena original.

VAL es una función muy potente, ya que la cadena que le entregamos como argumento no necesariamemte ha de tener la forma de un número constante, sino que puede tener la de una expresión numérica. Así, por ejemplo...

```
VAL "2*3"
```

da como resultado el número 6; incluso

```
VAL ("2"+"*3")
```

produce el número 6. En este último caso han intervenido dos procesos. En primer lugar se ha producido la *concatenación* de "2" y "*3" para dar la cadena "2*3"; después VAL ha suprimido las comillas y ha calculado el valor del producto 2*3, que es 6.

Esto puede resultar bastante confuso cuando las expresiones son complicadas. Por ejemplo,

```
PRINT VAL "VAL""VAL""""2""""""""""
```

(Recuerde que dentro de una cadena las comillas deben ser escritas dos veces. Si bucea a más profundidad en las cadenas, resulta que las comillas han de ser cuadruplicadas, o incluso octuplicadas.)

Existe otra función, bastante similar a VAL, aunque probablemente menos útil, llamada VAL$. Su argumento sigue siendo una cadena, pero su resultado también lo es. Para ver cómo funciona, recuerde la forma en que VAL realiza los cálculos: primero evalúa su argumento hasta obtener una cadena sencilla; después suprime las comillas y todo lo que queda lo evalúa como número. En el caso de VAL$, el primer paso es el mismo, pero, una vez eliminadas las comillas en el segundo paso, todo lo que queda se evalúa como cadena. Por ejemplo,

```
VAL$ """"Ursula""""      es igual a   "Ursula"
```

(Observe cómo las comillas proliferan de nuevo.) Haga ahora

```
LET a$="99"
```

y haga que el ordenador escriba todo siguiente: VAL a$, VAL "a$", VAL """"a$"""", VAL$ a$, VAL$ "a$" y VAL$ """"a$"""". Algunas de ellas funcionarán y otras no; intente explicar todas las respuestas. (Mantenga la cabeza fría.)

SGN es la función 'signo'. Es la primera función que nos encontramos que no tiene nada que ver con las cadenas, ya que tanto su argumento como su resultado son números. El resultado es +1 si el argumento es positivo, 0 si el argumento es 0, y −1 si el argumento es negativo.

ABS es otra función cuyos argumento y resultado son números. Convierte el argumento en un número positivo (que es el resultado) olvidando el signo. Así, por ejemplo,

```
ABS -3.2
```

es igual a

```
ABS 3.2
```

que, sencillamente, es igual a 3.2.

INT significa 'parte entera', un número entero, posiblemente negativo. Esta función convierte un número fraccionario (con decimales) en un entero desechando los decimales. Así, por ejemplo,

```
INT 3.9
```

da 3.

Tenga cuidado al aplicar esta función a números negativos, ya que siempre los redondea hacia abajo. Por ejemplo,

```
INT -3.1
```

es igual a −4.

SQR calcula la raíz cuadrada del argumento; es decir, da un resultado que, multiplicado por sí mismo, es el argumento. Por ejemplo,

```
SQR 4
```

es igual a 2 porque 2*2=4.

```
SQR 0.25
```

es igual a 0.5 porque 0.5*0.5=0.25.

```
SQR 2
```

es (aproximadamente) igual a 1.4142136 porque 1.4142136*1.4142136=2.0000001.

Si usted multiplica cualquier número (incluso uno negativo) por sí mismo, el producto siempre es positivo. Esto significa que los números negativos no tienen raíz cuadrada; por consiguiete, si aplicamos SQR a un argumento negativo, obtenemos el mensaje de error A Invalid Argument ('Argumento no válido').

Aparte de la funciones que BASIC nos ofrece, podemos también definir otras. Sus nombres consistirán en las letras FN seguidas de una letra de nuestra elección (si el resul-

tado va a ser un número) o en FN seguidas de otra letra y de $ (si el resultado va ser una cadena). Estas funciones son mucho más estrictas en lo que se refiere a los paréntesis (el argumento siempre tiene que figurar entre paréntesis).

Una función se define poniendo una sentencia DEF en algún lugar del programa. Por ejemplo, la siguiente sentencia define la función FN c, cuyo resultado es el cuadrado del argumento:

```
10 DEF FN c(x)=x*x: REM el cuadrado de x
```

La letra c que sigue a DEF FN es el nombre que hemos elegido para la función. La x entre paréntesis es el nombre por el cual nos referiremos al argumento de la función. Para esto se puede utilizar una letra cualquiera (pero sólo una), o bien, si el argumento es una cadena, una sola letra seguida de $.
Tras el signo = viene la verdadera definición de la función. Ésta puede ser cualquier expresión, y puede hacer referencia al argumento utilizando el nombre que le hemos dado (en este caso, x), como si fuera cualquier otra variable.

Una vez ejecutada la sentencia DEF, la función se puede utilizar de la misma manera que las intrínsecas de BASIC: escribiendo su nombre, FN c, seguido por el argumento entre paréntesis. (Recuerde siempre que en las funciones definidas por el usuario el argumento tiene que estar entre paréntesis). Practique unas cuantas veces:

```
PRINT FN c(2)
PRINT FN c(3+4)
PRINT 1+INT FN c (LEN "pollo"/2+3)
```

Decíamos antes que INT siempre redondea hacia abajo. Para redondear al entero más cercano, sume 0.5 al número antes de aplicarle INT; si no quiere tener que hacer esto cada vez que necesite redondear un número, puede encomendarle la tarea a una función creada al efecto:

```
20 DEF FN r(x)=INT (x+0.5): REM da x redondeado al entero más cercano
```

Después puede comprobar que, por ejemplo,

```
FN r(2.9)    es igual a 3     FN r(2.4)    es igual a 2
FN r(−2.9)   es igual a −3    FN r(−2.4)es igual a −2
```

Compare estas respuestas con las que se obtiene empleando INT en lugar de FN r. Transcriba y ejecute este programa:

```
10 LET x=0: LET y=0: LET a=10
20 DEF FN p(x,y)=a+x*y
30 DEF FN q()=a+x*y
40 PRINT FN p(2,3),FN q()
```