En este programa hay muchas sutilezas. Primero, las funciones pueden tener varios argumentos, e incluso no tener ninguno (lo que no se puede omitir es los paréntesis).

Segundo, las sentencias DEF pueden estar en cualquier lugar del programa. Cuando el ordenador ha ejecutado la línea 10, se salta las líneas 20 y 30 para llegar a la 40. Sin embargo, tienen que estar en *alguna* parte del programa, no en una orden directa.

Tercero, tanto x como y son nombres de variables en el conjunto del programa y además son los nombres de los argumentos de la función FN p. Ésta olvida temporalmente las variables llamadas x e y, pero, puesto que no tiene ningún *argumento* con el nombre de a, sigue recordando la *variable* a. De este modo, cuando FN p(2,3) está siendo evaluada, a tiene el valor 10 porque es la variable, x tiene el valor 2 porque es el primer argumento, e y tiene el valor 3 porque es el segundo argumento. Así que el resultado es 10+2*3, que es igual a 16. Por otra parte, cuando FN q() está siendo evaluada, no hay argumentos, de modo que a, x e y siguen refiriéndose a las *variables* y, por lo tanto, tienen los valores 10, 0 y 0, respectivamente. La respuesta en este caso es 10+0*0, que es igual a 10.

Cambie ahora la línea 20 por:

```
20 DEF FN p(x,y)=FN q()
```

Esta vez FN p(2,3) tendrá el valor 10, ya que FN q volverá a las variables x e y y no utilizará los argumentos de FN p.

Algunas versiones de BASIC (aunque no el Spectrum BASIC) tienen las funciones LEFT$, RIGHT$, MID$ y TL$.

LEFT$(a$,n) da la subcadena de a$ consistente en los n primeros caracteres.

RIGHT$(a$,n) da la subcadena de a$ consistente en los últimos caracteres, a partir del n–ésimo.

MID$(a$,n1,n2) da la subcadena de a$ consistente en los n2 caracteres tomados a partir del n1–ésimo.

TL$(a$) da la subcadena de a$ consistente en todos sus caracteres menos el primero.

Usted puede definir funciones de usuario que produzcan los mismos resultados:

```
10 DEF FN t$(a$)=a$(2 TO ): REM TL$
20 DEF FN l$(a$,n)=a$( TO n): REM LEFT$
```

Compruebe que estas funciones operan con cadenas de longitud 0 o 1. Observe que nuestra FN l$ tiene dos argumentos; uno es un número y el otro una cadena. Una función puede tener hasta 26 argumentos numéricos (¿por qué 26?) y, al mismo tiempo, hasta 26 argumentos literales.

Ejercicio

1. Utilice la función FN c(x)=x*x para probar SQR. Observará que

   ```
   FN c(SQR x)
   ```

   es igual a x si se pone cualquier número positivo en lugar de x, y que

   ```
   SQR FN c(x)
   ```

   es igual a ABS x tanto si x es positivo como si es negativo. (¿Por qué aparece ABS aquí?)

Parte 10
Funciones matemáticas

Temas tratados:

↑
PI, EXP, LN, SIN, COS, TAN, ASN, ACS, ATN

En esta sección vamos a ocuparnos de las funciones matemáticas que puede manejar el +2. Es posible que usted nunca haya utilizado ninguna de estas funciones; si se le hace demasiado pesado continuar, no le importe saltarse esta sección. Examinaremos la operación ↑ (elevar a una potencia), las funciones EXP y LN y las funciones trigonométricas SIN, COS, TAN, y sus inversas, ASN, ACS y ATN.

↑ y EXP

Elevar un número a una potencia es multiplicar el número por sí mismo cierto número de veces. Esta operación se indica normalmente escribiendo el segundo número (el exponente) a la derecha del primero y un poco más arriba; obviamente, esto sería difícil en un ordenador, por lo que utilizamos el símbolo ↑. Por ejemplo, las potencias de 2 son:

2↑1=2
2↑2=2*2=4 (2 al cuadrado, escrito normalmente 2^2).
2↑3=2*2*2=8 (2 al cubo, escrito normalmente 2^3).
2↑4=2*2*2*2=16 (2 a la cuarta, escrito normalmente 2^4).
etcétera.

Así, al nivel más elemental, *a*↑*b* significa 'a multiplicado por sí mismo *b* veces'; pero, evidentemente, esto sólo tiene sentido si *b* es un número entero positivo. Para encontrar una definición que sirva para otros valores de *b*, consideremos la regla:

a↑(*b*+*c*)=*a*↑*b***a*↑*c*

(Observe que concedemos a ↑ una prioridad mayor que a * y a /, de forma que, cuando hay varias operaciones en una expresión, las potencias se evalúan antes que los productos y las divisiones.) Cualquiera puede aceptar sin demasiados reparos que esta regla es válida cuando *a* y *b* son números enteros positivos; pero si queremos que funcione también cuando no lo sean, nos vemos forzados a aceptar que

$a\uparrow 0=1$
$a\uparrow(-b)=1/a\uparrow b$
$a\uparrow(1/b)$=la b–ésima raíz de a, es decir, el número que hay que multiplicar por sí mismo b veces para obtener a

y que

$a\uparrow(b*c)=(a\uparrow b)\uparrow c$

Si usted no ha visto nunca antes nada de esto, no intente aprendérselo de memoria a la primera; basta con que recuerde que:

$a\uparrow(-1)=1/a$

y que

$a\uparrow(1/2)$=SQR a

Puede ser, cuando se familiarice con estas fórmulas, que todas las demás empiecen a cobrar sentido.

Experimente probando este programa:

```
10 INPUT a,b,c
20 PRINT a↑(b+c),a↑b*a↑c
30 GO TO 10
```

Por supuesto, si la regla que dimos antes es cierta, en cada pasada por la línea 20 los dos números que escriba el +2 serán iguales. (Observe que, debido al modo en que el ordenador calcula las potencias, el número que está a la izquierda de ↑, a en este caso, nunca debe ser negativo.)

Un ejemplo bastante típico de aplicación de esta operación es el interés compuesto. Imagine que invierte una parte de su dinero en una sociedad inmobiliaria que le produce un 15% de interés anual. Después de un año no tendrá solamente el 100% de lo invertido, sino también el 15% de interés que le ha pagado la inmobiliaria, que hace un total del 115% de lo que tenía en un principio. En otras palabras, tiene que multiplicar su dinero por 1.15, y esto es válido cualquiera que sea la cantidad inicial. Cuando transcurra otro año habrá ocurrido lo mismo, por lo que tendrá 1.15*1.15; o, dicho de otra forma, 1.15↑2, es decir, 1.3225 veces la cantidad inicial de dinero. En general, al cabo de y años, tendrá 1.15↑y veces la cantidad inicial.

Si prueba esta orden

```
FOR y=0 TO 100: PRINT y,10*1.15↑y: NEXT y
```

verá que, incluso empezando con 1000 pesetas, la suma se incrementará bastante deprisa y, lo que es más, el ritmo al que aumenta va siendo cada vez mayor. (Pero no se haga

ilusiones: para calcular el incremento real del valor de su dinero tendría que restar el portentaje de inflación de ese 15% que le da la inmobiliaria.)

Este tipo de comportamiento, en el que al cabo de cierto intervalo de tiempo una cantidad se multiplica por un número fijo, es lo que se llama *crecimiento exponencial.* El valor final se calcula elevando ese número fijo a la potencia dada por el número de unidades de tiempo.

Supongamos que definimos la siguiente función:

```
10 DEF FN a(x)=a↑x
```

La variable a habrá sido definida en una sentencia LET (su valor corresponde al tipo de interés, que cambia frecuentemente).

Hay un valor especial de a que hace la función FN a particularmente atractiva para el ojo experimentado de un matemático; este valor es el llamado *número e.* El +2 posee una función, llamada EXP, definida por

EXP x es igual a e↑x

Lamentablemente, el número e no es en sí mismo un número demasiado bonito; se trata de un decimal infinito no periódico. Puede ver sus primeros decimales escribiendo la orden:

```
PRINT EXP 1
```

ya que EXP 1 = e↑1 = e. Por supuesto, esto es sólo una aproximación. No es posible escribir e con exactitud.

LN

La función inversa de la exponencial es la *función logarítmica.* El *logaritmo* de base a de un número x es la potencia a la que hay que elevar a para obtener el número x; abreviadamente, $\log_a x$. Así, por definición, a↑$\log_a x = x$; además, $\log_a(a$↑$x) = x$.

Quizá sepa usted cómo utilizar logaritmos de base 10; son los logaritmos decimales. El +2 dispone de una función, LN, que calcula logaritmos de base e, o sea, logaritmos *neperianos* o *naturales.* Para calcular el logaritmo de un número en cualquier otra base se divide el logaritmo neperiano del número por el logaritmo neperiano de la base; es decir, $\log_a x$=LN x/LN a.

PI

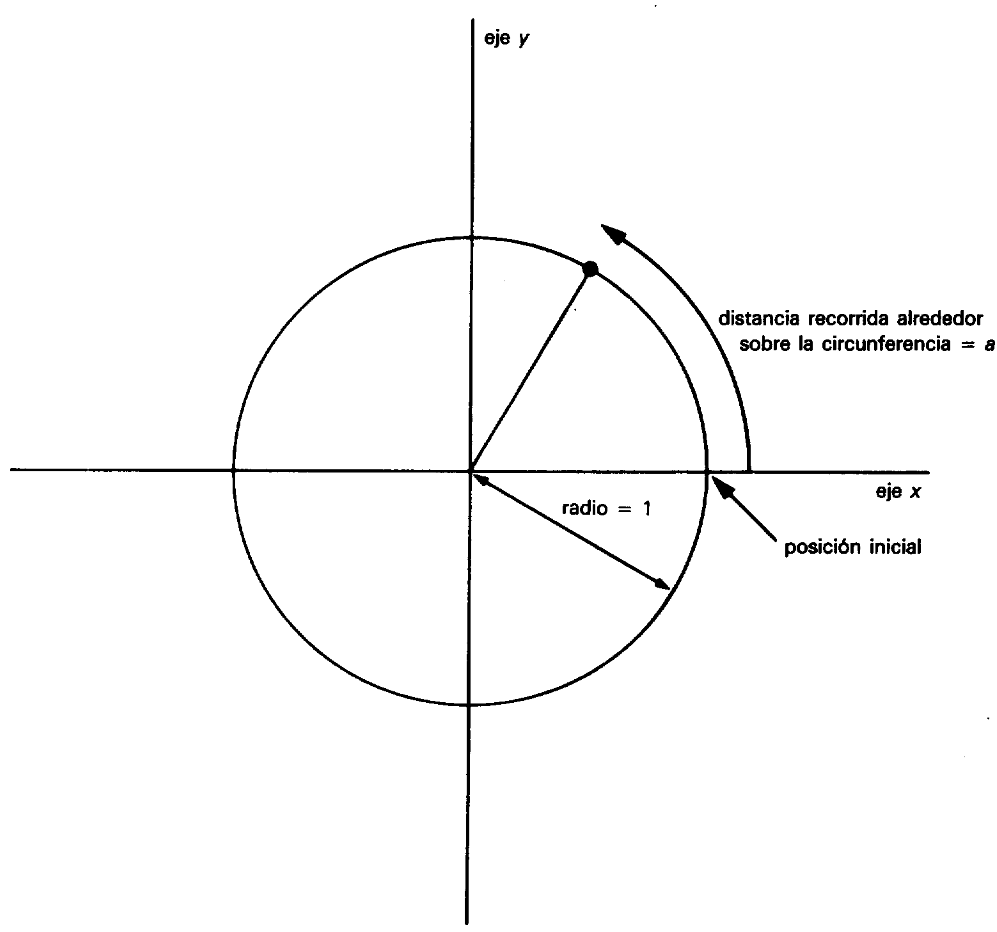
Como es bien sabido, la longitud de la circunferencia se obtiene multiplicado el diámetro por el número π. Este símbolo, π, es una 'p' griega que se lee 'pi'.

Al igual que e, π es un decimal infinito no periódico (empieza por 3.1415927). La palabra PI representa ese número en el +2. Pruebe lo siguiente:

```
PRINT PI
```

SIN, COS Y TAN: ASN, ACS Y ATN

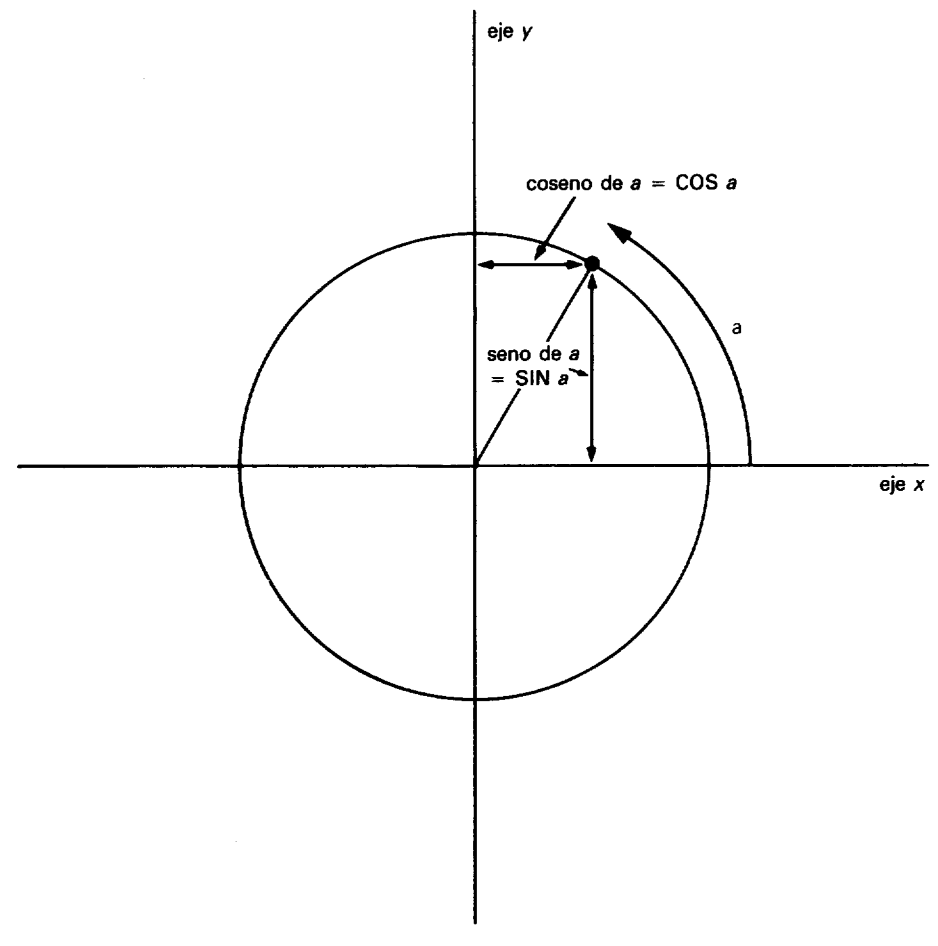
Las funciones *trigonométricas* dan una medida de lo que ocurre cuando un punto se mueve alrededor de una circunferencia. Tomemos una circunferencia de radio 1 y un punto que la recorre. El punto empieza en la posición de 'las tres' y se desplaza en el sentido contrario al de las agujas del reloj.



Hemos dibujados dos rectas, llamadas *ejes*, que pasan por el centro de la circunferencia. La que está en la posición de 'las tres' se llama eje x; la que pasa por 'las doce' es el eje y.

Para caracterizar la posición del punto basta con decir qué distancia ha recorrido sobre la circunferencia a partir de su posición inicial; vamos a llamar a a esta distancia. Sabemos que la longitud de la circunferencia es 2π (porque su radio es 1 y su diámetro, por consiguiente, es 2); así, cuando el punto ha recorrido un cuarto de la longitud de la circunferencia, $a=\pi/2$; cuando ha recorrido la mitad, $a=\pi$; y cuando ha recorrido todo el trecho, $a=2\pi$.

Dada la distancia recorrida sobre la circunferencia, a, puede interesar conocer la distancia a la que está el punto *a la derecha* del eje y y la distancia a la que está *por encima* del eje x. Estas distancias se llaman, respectivamente, *coseno* y *seno* de a, y son las calculadas por las funciones COS y SIN del +2.



Observe que si el punto está a la izquierda del eje y, el coseno es negativo; si el punto está por debajo del eje x, el seno es negativo.

Otra propiedad interesante es que, una vez que a ha crecido hasta 2π, el punto ha llegado a la posición inicial y el seno y el coseno empiezan a tomar otra vez los mismo valores. Es decir, SIN(a+2*PI) es igual a SIN a y COS(a+2*PI) es igual a COS a.

La *tangente* de a se define como el seno dividido por el coseno; la función correspondiente en el +2 se llama TAN.

Algunas veces necesitamos utilizar estas funciones al revés, o sea, averiguar qué valor de a ha producido cierto seno, coseno o tangente. Las funciones encargadas de esto se llaman *arco seno* (ASN en el +2), *arco coseno* (ACS) y *arco tangente* (ATN).

Observe el diagrama anterior y fíjese en el radio que une el punto con el centro. Está claro que la distancia que hemos llamado a (la distancia recorrida por el punto desde la posición inicial) es un modo de medir del ángulo formado por ese radio y el eje x. Cuando $a=\pi/2$, el ángulo tiene 90 grados; cuando $a=\pi$, el ángulo es de 180 grados, y así sucesivamente, hasta que $a=2\pi$ y el ángulo es de 360 grados. Podríamos olvidar completamente los grados y caracterizar el ángulo solamente con a; decimos entonces que estamos midiendo el ángulo en radianes. Así, $\pi/2$ radianes = 90 grados, etc.

Recuerde siempre que en el +2 las funciones SIN, COS, etc. trabajan siempre con radianes, no con grados. Para convertir grados en radianes se divide por 180 y se multiplica por π. Para convertir radianes en grados se divide por π y se multiplica por 180.

Parte 11
Números aleatorios

Temas tratados:

RANDOMIZE
RND

En esta sección vamos a estudiar las palabras clave RND y RANDOMIZE.

En algunos aspectos, RND es semejante a una función: realiza cálculos y genera un resultado. En cambio, es peculiar en el sentido de que no necesita argumento.

Cada vez que utilizamos RND, su resultado es un nuevo número aleatorio comprendido entre 0 y 1. (Algunas veces puede tomar el valor 0, pero nunca el 1.)

Pruebe lo siguiente:

```
10 PRINT RND
20 GO TO 10
```

¿Puede detectar alguna regla o algún patrón repetitivo en los números? No debería, ya que 'aleatorio' significa 'impredecible', 'sin normas'.

En realidad, RND no es perfectamente aleatoria, ya que los valores que genera están tomados de una secuencia fija que consta de 65536 números. Sin embargo, estos están tan «revueltos» que podemos considerarlos impredecibles a todos los efectos prácticos. Por eso decimos que RND es pseudoaleatoria.

RND da un número (pseudo)aleatorio comprendido entre 0 y 1, pero fácilmente se puede transformar ese intervalo en otro cualquiera. Por ejemplo, 5*RND da valores comprendidos entre 0 y 5; 1.3+0.7*RND los da en el margen de 1.3 a 2. Cuando necesitemos números enteros podremos utilizar INT (sin olvidar que INT siempre redondea hacia abajo), como en 1+INT (RND*6), expresión que utilizaremos en un programa que simulará el lanzamiento de dados. RND*6 genera valores que están en el margen de 0 a 6, pero, puesto que realmente nunca llega a 6, INT (RND*6) sólo puede ser 0, 1, 2, 3, 4 o 5.

He aquí el programa:

```
10 REM  Programa de lanzamiento de dados
20 CLS
30 FOR n=1 TO 2
40 PRINT 1+INT (RND*6);" ";
50 NEXT n
60 INPUT a$: GO TO 20
```

Pulse INTRO cada vez que quiera 'lanzar' los dados.

La sentencia RANDOMIZE sirve para hacer que RND se ponga en marcha a partir de una posición determinada en su secuencia de números, como demuestra este programa:

```
10 RANDOMIZE 1
20 FOR n=1 TO 5: PRINT RND,: NEXT n
30 PRINT: GO TO 10
```

Tras cada ejecución de RANDOMIZE 1, la secuencia de números aleatorios vuelve a arrancar a partir de 0.0022735596. Como parámetro de RANDOMIZE se puede poner cualquier número comprendido entre 1 y 65535 para poner en marcha la secuencia RND en diferentes posiciones.

Una aplicación de RANDOMIZE podría ser la siguiente: supongamos que usted tiene un programa que genera números aleatorios y que no funciona correctamente; puede usar RANDOMIZE para hacer que el programa genere siempre los mismos números y así poder estudiar más sistemáticamente su comportamiento.

RANDOMIZE a secas (o RANDOMIZE 0) tiene un efecto diferente: elige el punto de partida de forma bastante impredecible. Pruebe el siguiente programa:

```
10 RANDOMIZE
20 PRINT RND: GO TO 10
```

La secuencia que obtiene usted aquí no es muy aleatoria, pero esto tiene fácil explicación. RANDOMIZE toma como parámetro el *tiempo* transcurrido desde que se encendió el +2. Como en este programa no transcurre mucho tiempo desde una ejecución de RANDOMIZE a la siguiente, RND hace más o menos lo mismo todas las veces. Se obtendría una «aleatoriedad» mejor reemplazando GO TO 10 por GO TO 20.

El siguiente programa simula el lanzamiento de monedas y cuenta el número de caras y cruces:

```
10 LET caras=0: LET cruces=0
20 LET moneda=INT (RND*2)
30 IF moneda=0 THEN LET caras=caras+1
40 IF moneda=1 THEN LET cruces=cruces+1
50 PRINT caras;", ";cruces,
60 IF cruces<>0 THEN PRINT caras/cruces;
70 PRINT: GO TO 20
```

La proporción de caras y cruces debe llegar a ser aproximadamente 1 si se deja que el programa continúe el tiempo suficiente, ya que, a la larga, es de esperar más o menos igual número de caras que de cruces.

Ejercicio

1. Supongamos que usted elige un número comprendido entre 1 y 872 y escribe:

   RANDOMIZE *su número*

   Compruebe que el siguiente valor generado por RND es

   $(75*(\textit{su número}+1)-1)/65536$

   Pruébelo varias veces.

Parte 12
Matrices

Temas tratados:

Matrices (observe que el +2 maneja las matrices de forma algo diferente de la estándar)
DIM

Supongamos que tenemos una lista de números (por ejemplo, las notas de diez alumnos de una clase). La forma más obvia de almacenarlos en el +2 sería utilizar las variables *m1*, *m2*, *m3*,..., *m10*. Sin embargo, el programa necesario para asignar los valores a esas diez variables sería bastante largo y tedioso de escribir:

```
 10 LET m1=75
 20 LET m2=44
 30 LET m3=90
 40 LET m4=38
 50 LET m5=55
 60 LET m6=64
 70 LET m7=70
 80 LET m8=12
 90 LET m9=75
100 LET m10=60
```

Afortunadamente, existe un mecanismo, denominado *matriz*, que permite guardar todos esos valores con un solo nombre de variable. Una matriz es una variable especial que puede contener varios valores, llamados *elementos*; en esto se distingue de las variables ordinarias, que sólo pueden contener uno. Cada elemento de la matriz se identifica por un número, que es el *índice* (o *subíndice*). El índice se escribe entre paréntesis a continuación del nombre de la matriz. En el ejemplo anterior, el nombre global de los elementos de la matriz podría ser *m* (el nombre de las variables matriciales siempre tiene que consistir en una sola letra); por consiguiente, los diez elementos serían *m(1)*, *m(2)*, *m(3)*,..., *m(10)*.

Los elementos de una matriz reciben también el nombre de *variables indexadas*; las variables que habíamos conocido hasta ahora se llaman variables *sencillas* u *ordinarias*.

Antes de poder utilizar una matriz, es necesario haberle reservado espacio en la memoria del +2; esta operación es lo que se llama *dimensionar* la matriz y se realiza mediante la palabra clave DIM. Por ejemplo, la sentencia

```
DIM m(10)
```

reserva espacio en la memoria para una matriz llamada *m* cuyas dimensiones van a ser diez (es decir, equivaldrá a diez variables indexadas). La sentencia DIM «inicializa» los elementos de la matriz dándoles el valor 0. Además, si ya existía una matriz con ese mismo nombre, la borra. (Pero no afecta a la variable sencilla *m*, si es que existe; una variable matricial puede coexistir con una variable sencilla numérica del mismo nombre, ya que la matriz es siempre distinguible por su subíndice.)

Los subíndices de los elementos de la matriz pueden ser representados por cualquier expresión numérica que produzca un número válido como subíndice. Gracias a esto, las matrices pueden ser procesadas utilizando bucles FOR...NEXT. De ese modo, podemos olvidar el interminable programa que dimos al principio de esta sección y guardar los diez datos mediante el siguiente:

```
10 DIM m(10)
20 FOR n=1 TO 10
30 READ m(n)
40 NEXT n
50 DATA 75,44,90,38,55,64,70,12,75,60
```

(Observe que la sentencia DIM tiene que haber sido ejecutada *antes* de intentar acceder a la matriz.)

Si lo desea, puede usted examinar el contenido de la matriz escribiendo:

```
PRINT m(1)
PRINT m(2)
PRINT m(3)
etcétera
```

También podemos formar matrices con varias dimensiones. En una matriz bidimensional necesitaremos dos números para especificar cada uno de sus elementos, de la misma manera que necesitamos un número de fila y un número de columna para especificar la posición de un carácter en la pantalla. Si imaginamos que los números de línea y de columna (dos dimensiones) describen una página impresa, en la tercera dimensión tendríamos los números de página. Por supuesto, nosotros estamos hablando de matrices numéricas, por lo que los elementos de esa matriz tridimensional no serían caracteres de texto, sino números. Trate de imaginarse los elementos de una matriz tridimensional *v* especificados por *v(número de página,número de línea,número de columna).*

Por ejemplo, para formar una matriz bidimensional con dimensiones 3 y 6, se utiliza la siguiente sentencia DIM:

```
DIM c(3,6)
```

Esa matriz tendrá un total de 3*6=18 variables indexadas:

c(1,1), *c*(2,2),..., *c*(1,6)
c(2,1), *c*(2,2),..., *c*(2,6)
c(3,1), *c*(3,2),..., *c*(3,6)

La misma idea es válida para cualquier número de dimensiones.

Aunque es posible tener una variable sencilla y una matricial con el mismo nombre, no puede haber dos matrices que se llamen igual, aunque tengan diferente número de dimensiones.

Hasta ahora hemos descritos la *matrices numéricas*. También existen *matrices literales*, cuyos elementos son, naturalmente, cadenas literales. Las cadenas de una matriz se distinguen de las cadenas sencillas en que son de longitud fija y en que la asignación de valores se realiza por el procedimiento de Procrustes (¿lo recuerda?): si el valor es demasiado largo, se lo recorta; si es demasiado corto, se lo complementa con espacios por la derecha.

El nombre de toda matriz literal consiste en una sola letra seguida de $. A diferencia lo que ocurría con las matrices numéricas, una matriz literal y una variable literal sencilla no pueden tener el mismo nombre.

Supongamos, pues, que queremos una matriz a$ que pueda albergar cinco cadenas. Debemos decidir qué longitud van a tener las cadenas. Por ejemplo, una matriz para cinco cadenas de diez caracteres cada una se dimensiona con:

```
DIM a$(5,10)
```

Esta sentencia prepara una matriz de 5*10 caracteres, que podemos manejar individualmente o por filas completas:

a$(1)=*a*$(1,1)*a*$(1,2)...*a*$(1,10)
a$(2)=*a*$(2,1)*a*$(2,2)...*a*$(2,10)
a$(3)=*a*$(3,1)*a*$(3,2)...*a*$(3,10)
a$(4)=*a*$(4,1)*a*$(4,2)...*a*$(4,10)
a$(5)=*a*$(5,1)*a*$(5,2)...*a*$(5,10)

Si al usar la matriz especificamos el mismo número de subíndices (dos en este caso) que dimensiones hay en la sentencia DIM, obtenemos un solo carácter. Pero si omitimos el último, la matriz da la fila entera (una cadena de longitud fija). Así que, por ejemplo, *a*$(2,7) es el séptimo carácter de la cadena *a*$(2). Utilizando la notación de disección de cadenas (Parte 8 de este capítulo) también podríamos escribir *a*$(2)(7) en lugar de *a*$(2,7).

Ahora escriba:

```
LET a$(2)="1234567890"
```

y

```
PRINT a$(2), a$(2,7)
```

El resultado será:

```
1234567890    7
```

El último subíndice (el que se puede omitir), también puede tener la estructura de los parámetros de la disección de cadenas. Por ejemplo,

a$(2,4 TO 8) es igual a a$(2)(4 TO 8) que a su vez es igual a "45678"

Recuerde: en una matriz literal, todas las cadenas tienen la misma longitud.La sentencia DIM tiene un parámetro adicional (el último) que especifica esa longitud. Al escribir una variable indexada perteneciente a una matriz literal, podemos incluir un número adicional (o bien un *disector de cadenas*), que corresponderá al último parámetro de la sentencia DIM.

Si al dimensionar una matriz literal sólo especificamos un parámetro, la matriz se comportará como variable sencilla de longitud física. Escriba:

```
DIM a$(10)
```

y podrá comprobar que a$ funciona igual que una variable literal ordinaria, con la única diferencia de que su longitud siempre será 10 y los valores le serán asignados por el método de Procrustes.

Ejercicio

1. Utilice las sentencias READ y DATA para formar una matriz m$ en la cual m$(n) sea el nombre del *n*–ésimo mes. (*Sugerencia.* Suponiendo que usted escriba 'setiembre' y no 'septiembre', la sentencia DIM necesaria es DIM m$(12,9).) Compruebe todos los valores de m$(n) haciendo que los escriba un bucle FOR...NEXT.

Parte 13
Condiciones

Temas tratados:

AND, OR
NOT

Vimos en la Parte 3 de este capítulo que la sentencia IF toma la forma

IF *condición* THEN ...

Las condiciones eran entonces relaciones (construidas a base de =, <, >, <=, >= y <>) que comparaban dos números o dos cadenas. También podemos combinar varias relaciones utilizando los operadores lógicos: AND ('y'), OR ('o') y NOT ('no').

Una relación combinada mediante AND con otra relación da el valor 'verdadero' siempre que ambas sean verdaderas; por ejemplo, podríamos tener una línea del siguiente estilo:

```
IF a$="si" AND x>0 THEN PRINT x
```

en la que x sólo se escribe si a$ es igual a "si" y x mayor que cero.

Una relación combinada mediante OR con otra relación da el valor 'verdadero' siempre que al menos una de ellas sea verdadera. (El resultado también es verdadero si las dos relaciones son verdaderas.)

La relación NOT invierte el valor lógico: NOT *relación* es 'verdadero' siempre que la *relación* es falsa, y 'falso' cuando la *relación* es verdadera.

Las *expresiones lógicas* consisten en combinaciones de AND, OR y NOT, de la misma forma que las expresiones numéricas son combinaciones de +, −, *, /, etc. También se puede colocarlas entre paréntesis si es necesario. Las operaciones lógicas tienen orden de prioridad, lo mismo que +, −, *, / y ↑. OR es la menos prioritaria; en orden de prioridad ascendente le siguen AND y NOT.

NOT es en realidad una función, con un argumento y un resultado, pero su prioridad es muy inferior a la de las otras funciones. Por lo tanto, su argumento no necesita paréntesis, a menos que sea en sí una operación lógica (construida con AND, OR o ambas). NOT a=b significa lo mismo que NOT (a=b), y, por supuesto, lo mismo que a<>b.

<> es la negación de =, en el sentido de que <> es verdadero si y sólo si = es falso. En otras palabras

<>b   es lo mismo que   NOT a=b

y además

NOT a<>b   es lo mismo que   a=b

Si lo piensa, se dará cuenta de que >= y <= son la negación de < y >, respectivamente. Por consiguiente, para invertir el valor de una relación se puede escribir NOT delante de ella o sustituirla por su negación.

Por otra parte,

NOT (*primera expresión lógica* AND *segunda expresión lógica*)

es lo mismo que

NOT (*primera expresión lógica*) OR NOT (*segunda expresión lógica*)

Además

NOT (*primera expresión lógica* OR *segunda expresión lógica*)

es lo mismo que

NOT (*primera expresión lógica*) AND NOT (*segunda expresión lógica*)

Utilizando estas reglas, las negación de una expresión se puede reducir a una expresión en la que NOT esté aplicada directamente a relaciones. En último extremo, NOT no es imprescindible, aunque en la práctica facilita la programación al hacer más inteligibles las expresiones lógicas.

Lo que resta de esta sección es bastante complicado; si no le interesa mucho este tema, puede omitir su lectura y pasar directamente a la Parte 14.

Pruebe la siguiente orden:

```
PRINT 1=2, 1<>2
```

que aparentemente debería producir un error de sintaxis. En realidad, el ordenador no sabe qué es eso de 'valor lógico'; en lugar de los valores 'verdadero' y 'falso' utiliza números ordinarios, que están sujetos a unas cuantas reglas:

(i) Cuando el ordenador evalúa las relaciones construidas a base de =, <, >, <=, >= y <>, obtiene un valor numérico que es 1 cuando la relación es verdadera y 0 cuando la relación es falsa. Por eso la anterior orden PRINT escribió 0 para 1=2, que es una relación falsa, y 1 para 1<>2, que es verdadera.

(ii) En la sentencia

IF *condición* THEN ...

la *condición* puede ser en realidad una expresión numérica cualquiera. Si su valor es 0, el ordenador actúa como si se tratase de una expresión lógica con valor 'falso'; si el valor de *condición* es distinto de 0, el ordenador la considera 'verdadera'. Por tanto, la anterior sentencia IF es exactamente equivalente a

IF *condición*<>0 THEN ...

(iii) AND, OR y NOT son también operaciones aplicables a expresiones numéricas cualesquiera:

x AND y vale { *x* si *y* es verdadero (distinto de cero)
{ 0 (falso) si *y* es falso (cero)

x OR y vale { 1 (verdadero) si *y* es verdadero (distinto de cero)
{ *x* si *y* es falso (cero)

NOT x vale { 0 (falso) si *x* es verdadero (distinto de cero)
{ 1 (verdadero) si *x* es falso (cero)

(Observe que 'verdadero' significa 'distinto de cero' cuando estamos comprobando un valor dado, pero que significa '1' cuando estamos produciendo un valor nuevo.)

Ahora pruebe este programa:

```
10 INPUT a
20 INPUT b
30 PRINT (a AND a>=b)+(b AND a< b)
40 GO TO 10
```

En cada vuelta, este programa escribe el mayor de los dos números introducidos.

Si se para a pensarlo, se dará cuenta de que

x AND y

se podría leer así:

x si *y* (de lo contrario, el resultado es 0)

y de que

x OR y

significa lo mismo que

x a no ser que *y* (en cuyo caso el resultado es 1)

Una expresión en la que se utilice AND u OR de esta forma es lo que se llama *expresión condicional*. Un ejemplo con OR podría ser:

```
LET total=precio sin impuesto*(1.12 OR v$="Exento de IVA")
```

Observe que AND tiende a asociarse con la suma (porque su valor por defecto es 0) y OR con la multiplicación (porque su valor por defecto es 1).

También se puede formar expresiones condicionales cuyo valor sea una cadena literal, pero sólo utilizando AND:

x$ AND y vale	*x$* si y es distinto de cero
	"" (la cadena vacía) si y es cero

de modo que x$ AND y significa '*x$* si *y* (de lo contrario, su valor es la cadena vacía)'.

Pruebe este programa, que capta dos cadenas y las coloca en orden alfabético:

```
10 INPUT "Escriba dos cadenas" 'a$,b$
20 IF a$>b$ THEN LET c$=a$: LET a$=b$: LET b$=c$
30 PRINT a$;" ";("<" AND a$<b$)+("=" AND a$=b$);" ";b$
40 GO TO 10
```

Parte 14
El juego de caracteres

Temas tratados:

- CODE, CHR$
- POKE, PEEK
- USR
- BIN

Las letras, dígitos, espacios, signos de puntuación, etc. que pueden formar parte de las cadenas literales se llaman *caracteres* y componen el *juego de caracteres* del +2. En su mayor parte, estos caracteres son símbolos simples, pero algunos representan palabras completas, tales como PRINT, STOP, <>, etc.

Hay un total de 256 caracteres, cada uno de ellos identificado por un código comprendido entre 0 y 255 (en la Parte 27 de este capítulo damos la lista completa). Para efectuar la conversión entre códigos y caracteres disponemos de dos funciones: CODE y CHR$.

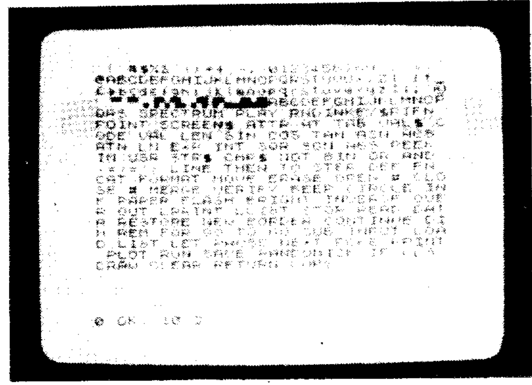
CODE se aplica a una cadena y da el código de su primer carácter (o, si se trata de la cadena vacía, da el número 0).

CHR$ se aplica a un número y da la cadena de un único carácter cuyo código es ese número.

El siguiente programa escribe el juego de caracteres del +2 (del 32.° en adelante):

```
10 FOR a=32 TO 255: PRINT CHR$ a;: NEXT a
```

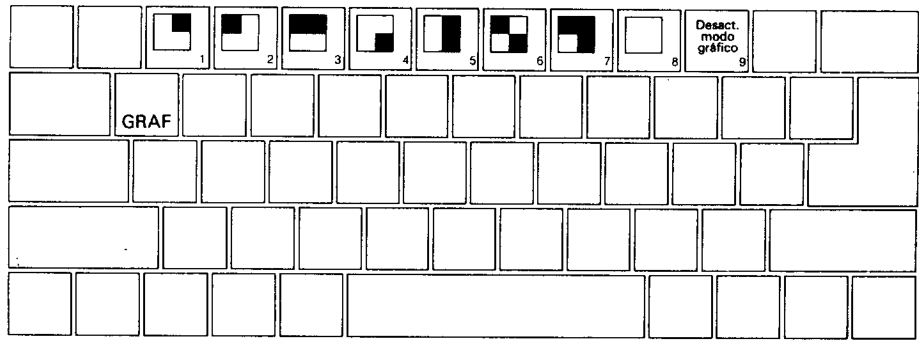
En la pantalla aparecerá lo siguiente...



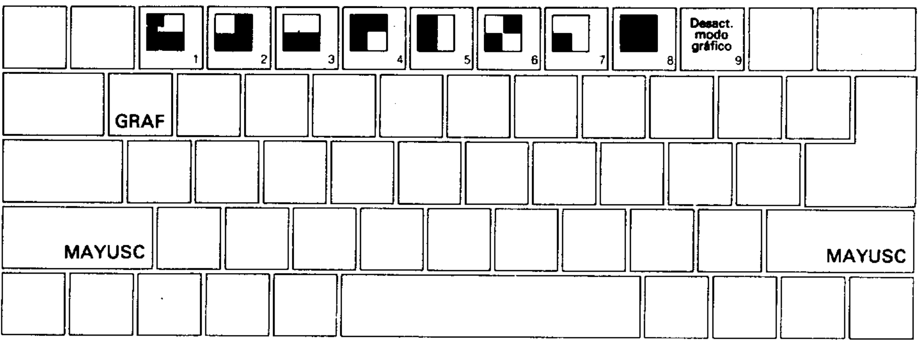
El juego de caracteres

Como puede apreciar, el juego de caracteres consta del espacio, 15 símbolos y signos de puntuación, los diez dígitos, otros siete símbolos, las mayúsculas, otros seis símbolos (entre los que se encuentra la 'Ñ'), las minúsculas y cinco símbolos más (entre los que se encuentra la 'ñ'). Todos ellos (excepto 'Pt', '©', '¡', '¿', 'ñ' y 'Ñ') están tomados de un juego de caracteres estándar, muy conocido y ampliamente utilizado: el juego ASCII (*American Standard Code for Information Interchange*, 'Código americano estándar para el intercambio de la información'). Los caracteres del juego ASCII están identificados por códigos, y esos códigos son los que emplea el +2.

Los demás caracteres no forman parte del juego ASCII, sino que son específicos de los ordenadores ZX Spectrum. Los primeros de ellos son un espacio y 15 combinaciones de cuadrados blancos y negros; éstos son los llamados *símbolos gráficos*, que pueden ser usados para dibujar. El usuario puede introducirlos por el teclado seleccionando el denominado *modo gráfico* (*modo G*). Al pulsar la tecla GRAF se activa el modo gráfico; los símbolos gráficos se obtienen entonces pulsando las teclas numéricas (del **1** al **8**).



Estando en modo gráfico, al pulsar las teclas numéricas mencionadas en combinación con MAYUSC se obtiene los mismos símbolos gráficos, pero 'invertidos' (es decir, convirtiendo lo negro en blanco, y vice versa).



En modo gráfico las teclas del cursor no funcionan como de costumbre, ya que el +2 las interpreta como teclas numéricas combinadas con MAYUSC y, por lo tanto, producen símbolos gráficos.

Pulsando la tecla del 9 (o pulsando por segunda vez GRAF) se sale del modo gráfico y se vuelve al normal. Pulsando la tecla del 0 se borra el carácter que está a la izquierda del cursor.

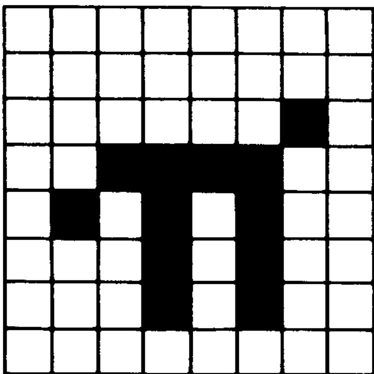
He aquí los 16 símbolos gráficos:

Símbolo	*Código*	*Símbolo*	*Código*
	128		143
	129		142
	130		141
	131		140
	132		139
	133		138
	134		137
	135		136

Los siguientes caracteres del juego del +2 son, aparentemente, una repetición de las mayúsculas de la 'A' a la 'S'. En realidad, se trata de caracteres cuya forma podemos rediseñar (sin embargo, mientras no los rediseñemos, tendrán por defecto la forma de esas letras mayúsculas); se les llama *gráficos definibles por el usuario*. Para introducirlos por el teclado primero se activa el modo gráfico y luego se pulsa una tecla alfabética (de la A a la S).

Para definir un nuevo carácter usted mismo, aplique la siguiente receta (que define un carácter con la forma de la letra griega 'π'):

(i) Decida qué forma quiere que tenga el carácter. Cada carácter ocupa una retícula de 8×8 puntos, cada uno de los cuales puede estar encendido o apagado (los cuadrados negros representan los puntos que están encendidos).



Cuando un punto está encendido, el +2 lo dibuja con el color de la tinta; cuando el punto está apagado, lo dibuja con el color del papel. (Los términos *tinta* y *papel* están explicados en la Parte 16 de este capítulo.)

Hemos dejado un borde libre alrededor del diseño del carácter porque todas las letras también lo tienen, excepto las minúsculas con descendente, en las que éste llega hasta el extremo inferior.

(ii) Decida a qué gráfico definible por el usuario quiere asignar la forma de la 'π'. Supongamos que elige el correspondiente a la 'P', de modo que cuando usted pulse P (en modo gráfico) obtenga π.

(iii) Almacene el nuevo diseño. Cada gráfico definido por el usuario se almacena en forma de una sucesión de ocho números, uno para cada fila. Usted puede especificar estos números en un programa escribiendo la palabra BIN seguida de 8 ceros o unos (0 para el papel, 1 para la tinta). De esta forma, los ocho números que describen nuestro carácter π son:

BIN 00000000 primera fila (superior)
BIN 00000000 segunda fila
BIN 00000010 tercera fila
BIN 00111100 cuarta fila
BIN 01010100 quinta fila
BIN 00010100 sexta fila

BIN 00010100 séptima fila
BIN 00000000 octava fila (inferior)

(Si conoce el sistema de numeración binario, podemos decirle que BIN se utiliza para escribir los números en ese sistema.) Observe este grupo de números binarios con los ojos entornados: quizá incluso pueda ver el carácter π.

Estos ocho números se almacenan en ocho posiciones (bytes) de memoria. Cada una de estas posiciones tiene una *dirección*. La dirección del primer byte (o grupo de ocho dígitos binarios) es USR "P" (porque escogimos la 'P' en el apartado (ii)). La dirección del segundo byte es USR "P"+1, y así hasta llegar al octavo byte, cuya dirección es USR "P"+7.

Aquí, USR es una función que convierte un argumento literal en la dirección del primer byte de memoria en que está almacenado el correspondiente gráfico definido por el usuario. El argumento literal debe ser un solo carácter, bien el propio gráfico definido por el usuario, bien la letra correspondiente (mayúscula o minúscula). USR tiene otro significado cuando su argumento es un número, pero de eso hablaremos más adelante.

Pues bien, aunque no haya conseguido seguirnos en el proceso que acabamos de explicar, pruebe el siguiente programa, que realiza la definición del carácter:

```
10 FOR n=0 to 7
20 READ fila: POKE USR "P"+n,fila
30 NEXT n
40 DATA BIN 00000000
50 DATA BIN 00000000
60 DATA BIN 00000010
70 DATA BIN 00111100
80 DATA BIN 01010100
90 DATA BIN 00010100
100 DATA BIN 00010100
110 DATA BIN 00000000
```

La sentencia POKE almacena directamente un número en una posición de memoria, eludiendo el mecanismo normalmente usado por BASIC. Lo inverso de POKE es PEEK, función que permite leer (sin modificarlo) el contenido de las posiciones de memoria. En la Parte 24 de este capítulo describiremos PEEK y POKE más detalladamente.

Continuando con el juego de caracteres, los siguientes son las *claves*.

Habrá notado que en el primer programa de esta sección no hemos escrito los 32 primeros caracteres (códigos 0 al 31); son los *caracteres de control*. No producen ningún carácter visible, sino que nos permiten controlar la imagen que se forma en la pantalla y algunas otras funciones del +2.

(Si usted intenta escribir caracteres de control, el +2 muestra un ? en la pantalla para indicar que no los entiende. Los caracteres de control están descritos más detalladamente en la Parte 27 de este capítulo.)

Tres de los códigos que afectan a la imagen de la pantalla son el 6, el 8 y el 13. Vamos a explicarlos.

De los tres, CHR$ 8 es posiblemente el único que tendrá interés para usted.

CHR$ 6 inserta espacios exactamente de la misma forma que lo hace la coma en las sentencias PRINT. Por ejemplo,

```
PRINT 1; CHR$ 6;2
```

produce el mismo efecto que

```
PRINT 1,2
```

Obviamente, ésta no es una forma muy clara de usarlo. Una manera más comprensible sería:

```
LET a$="1"+CHR$ 6+"2"
PRINT a$
```

CHR$ 8 es el *espacio hacia atrás* o *retroceso del cursor*: desplaza la posición de escritura un lugar hacia la izquierda. Pruebe la siguiente orden:

```
PRINT "1234"; CHR$ 8;"5"
```

cuyo efecto final es escribir

```
1235
```

CHR$ 13 es *línea nueva*: desplaza la posición de escritura al comienzo de la línea siguiente.

La pantalla reconoce también los códigos 16 al 23, que están explicados en las Partes 15 y 16 de este capítulo. (La lista completa de todos los códigos se encuentra en la Parte 27.)

Teniendo en cuenta todos los caracteres «visibles», podemos ampliar el concepto de 'orden alfanumérico' a cadenas que contengan cualquier carácter, no sólo letras. Para ello debemos considerar un alfabeto ampliado que, en lugar de constar de 26 letras, sea la lista de todos los 256 caracteres, en el mismo orden que sus códigos. Por ejemplo, las siguientes cadenas están, para el +2, en orden alfabético. (Observe que, curiosamente, las letras minúsculas van detrás de todas las mayúsculas, de forma que la 'a' es posterior a la 'Z'; además, los espacios son significativos.)

```
CHR$ 3+"ZOOLOGICO"
CHR$ 8+"AARRR"
" AAAARRR!"
"(Nota entre paréntesis)"
"100"
"129.95 inc. IVA"
"AAARRR"
"AaaRRR"
"Elton John"
"PRINT"
"Zoo"
"[interpolación]"
"aaarrr"
"aaasss"
"derecho"
"zoo"
"zoología"
```

La regla para ordenar dos cadenas es la siguiente. Primero se compara el primer carácter de una con el primer carácter de la otra. Si son diferentes, uno de los dos caracteres tendrá un código menor que el otro; la primera cadena en orden «alfabético» es la que empieza por el carácter cuyo código es menor. En cambio, si los dos caracteres son iguales, se pasa al segundo carácter de las dos cadenas y se realiza con ellos la comparación; y así sucesivamente hasta que los caracteres comparados sean diferentes. Si, en este proceso, una de las cadenas termina antes que la otra sin que se haya detectado diferencia entre los caracteres, la cadena más corta es la primera; de lo contrario, las dos cadenas son iguales.

Las relaciones =, <, >, <=, >= y <> son aplicables tanto a cadenas como a números: < significa ‘es anterior a’ y > ‘es posterior a’, de modo que, por ejemplo, las relaciones

```
"AA RRR"<""AAARRR"
"AAARRR">"AAA RRR"
```

son ambas verdaderas.

<= y >= funcionan de la misma forma que con números, y por lo tanto el valor de la relación

```
"Esta cadena"<="Esta cadena"
```

es ‘verdadero’, mientras que el de

```
"Esta cadena"<"Esta cadena"
```

es ‘falso’.

Para experimentar con todo esto, ejecute el programa siguiente, que capta dos cadenas y las ordena:

```
10 INPUT "Escriba dos cadenas:",a$,b$
20 IF a$>b$ THEN LET c$=a$: LET a$=b$: LET b$=c$
30 PRINT a$;" ";
40 IF a$<b$ THEN PRINT "<";: GO TO 60
50 PRINT "=";
60 PRINT " ";b$
70 GO TO 10
```

Observe que, tanto en este programa como en el del final de la Parte 13, hemos tenido que usar c$ (línea 20) para poder intercambiar los valores de a$ y b$. ¿Comprende por qué con

```
LET a$=b$: LET b$=a$
```

no habríamos obtenido el efecto deseado?

El siguiente programa define gráficos de forma que las siguientes teclas produzcan las piezas del ajedrez

B para el alfil
K para el rey
R para la torre
Q para la reina
P para el peón
N para el caballo

Piezas del ajedrez:

```
  5 LET b=BIN 01111100: LET c=BIN 00111000: LET d=BIN 00010000
 10 FOR n=1 TO 6: READ p$: REM 6 piezas
 20 FOR f=0 TO 7: REM leer 8 bytes
 30 READ a: POKE USR p$+f,a
 40 NEXT f
 50 NEXT n
100 REM alfil
110 DATA "b",0,d, BIN 00101000, BIN 01000100
120 DATA BIN 01101100,c,d,0
130 REM rey
140 DATA "k",0,d,c,d
150 DATA c, BIN 01000100,c,0
160 REM torre
170 DATA "r",0, BIN 01010100,b,c
180 DATA c,b,b,0
```

```
190 REM reina
200 DATA "q",0, BIN 01010100, BIN 00101000,d
210 DATA BIN 01101100,b,b,0
220 REM peón
230 DATA "p",0,0,d,c
240 DATA c,d,b,0
250 REM caballo
260 DATA "n",0,d,c, BIN 01111000
270 DATA BIN 00011000,c,b,0
```

Observe que en las sentencias DATA anteriores hemos puesto 0 en lugar de BIN 00000000.

Cuando haya ejecutado el programa, puede ver las piezas pulsando [GRAF] y cualquiera de las teclas: B, K, R, Q, P o N.

Ejercicios

1. Imagine el espacio para un símbolo dividido en cuatro cuartos. Si cada cuarto puede ser blanco o negro, hay $2^4=16$ posibilidades. Búsquelas todas en el juego de caracteres.

2. Ejecute este programa:

   ```
   10 INPUT a
   20 PRINT CHR$ a;
   30 GO TO 10
   ```

   Puede comprobar que CHR$ redondea el número a al entero más cercano; si a no está en el margen de 0 a 255, el programa se detiene y BASIC emite el mensaje de error B integer out of range ('Entero fuera de intervalo').

3. ¿Cuál de las dos cadenas siguientes es menor?

   ```
   "GATO"
   "gato"
   ```

Parte 15
Más sobre PRINT e INPUT

Temas tratados:

- CLS
- Elementos de PRINT
- PRINT 'nada'
- TAB, AT
- Separadores de PRINT: , ; '
- Elementos de INPUT
- LINE
- Elementos de PRINT que no empiezan con una letra
- Desplazamiento de la pantalla
- SCREEN$

Hemos utilizado la orden PRINT bastantes veces, así que usted ya tendrá una idea bastante buena de cómo funciona. Las expresiones cuyos valores escribimos con PRINT son los elementos de PRINT. Pueden estar separadas entre sí por comas, apóstrofos o signos de punto y coma. Todos estos signos de puntuación reciben el nombre de *separadores* de PRINT. Un elemento de PRINT puede ser también 'nada', y esto explica qué ocurre cuando en PRINT ponemos dos comas seguidas.

Hay otros dos tipos de elementos de PRINT que se emplean para comunicar al +2 *dónde* debe escribir, no *qué* debe escribir. Por ejemplo, la instrucción

```
10 PRINT AT 11,16;"*"
```

escribe un asterisco en el centro de la pantalla. Esto se debe a que

AT *fila,columna*

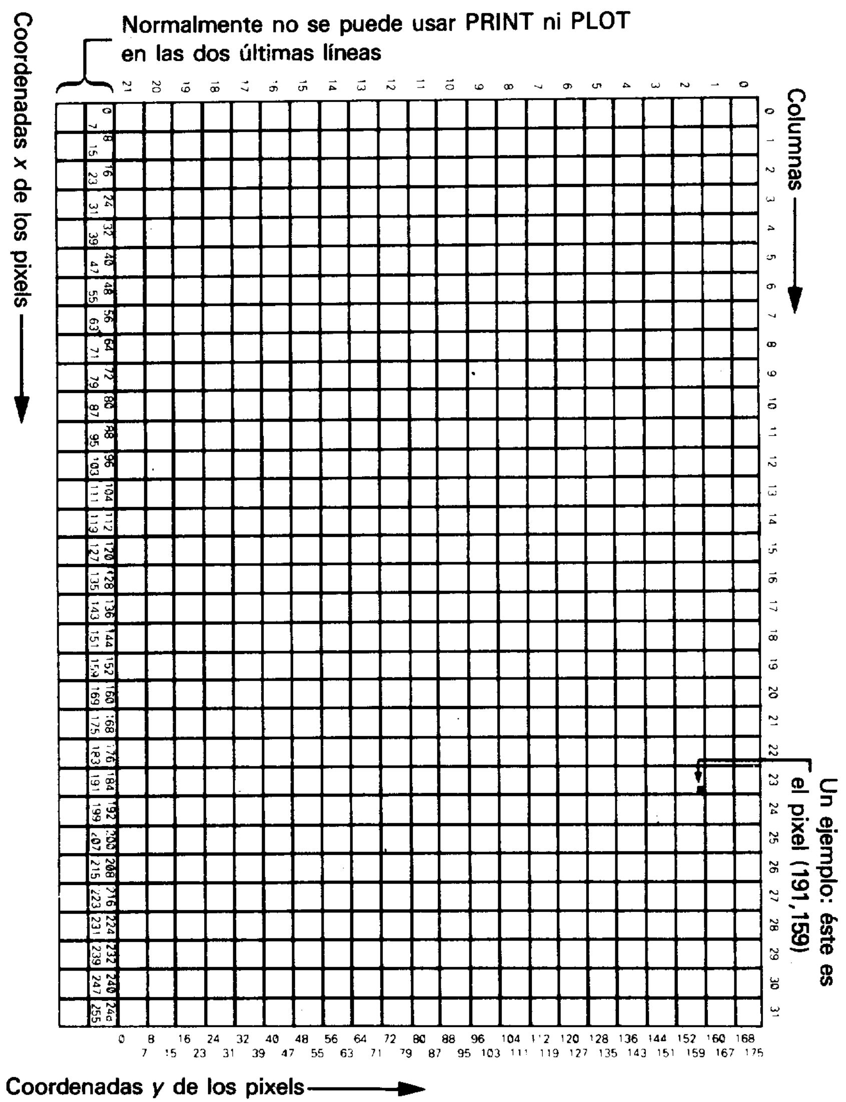
desplaza la *posición de escritura* (o sea, el lugar donde va a aparecer el siguiente elemento que se escriba) a la fila y columna especificadas. Las filas están numeradas de 0 (la más alta) a 21; las columnas están numeradas de 0 (extremo izquierdo) a 31.

SCREEN$ es la función inversa de PRINT AT, pues «lee» (dentro de ciertos límites) el carácter que se encuentra en la pantalla en la posición especificada. Sus parámetros son los números de fila y columna, con el mismo significado que en PRINT AT, pero puestos entre paréntesis. Por ejemplo, la instrucción

```
20 PRINT AT 0,0; SCREEN$ (11,16)
```

lee el asterisco que escribimos antes en el centro de la pantalla y luego lo reproduce en la posición 0,0 (el extremo superior izquierdo).

Los caracteres de las claves son leídos normalmente (como caracteres sencillos); los espacios son leídos como tales. En cambio, si se intenta leer caracteres definidos por el usuario, caracteres gráficos o líneas dibujadas por PLOT, DRAW o CIRCLE, la función SCREEN$ da la cadena vacía. Lo mismo ocurre si se ha usado OVER para crear un carácter compuesto. (Las palabras clave PLOT, DRAW, CIRCLE y OVER están descritas en las Partes 16 y 17 de este capítulo.)



La función

TAB *columna*

escribe los espacios necesarios para desplazar la posición de escritura hasta la columna especificada. Intenta no cambiar de línea pero, si ello la obliga a retroceder, prefiere saltar a la línea siguiente. Observe que el +2 toma el número de columna '*módulo 32*' (es decir, divide por 32 y toma el resto), de forma que TAB 33 significa lo mismo que TAB 1.

Por ejemplo,

```
PRINT TAB 30; 1; TAB 12; "Contenido"; AT 3,1; "Capitulo"; TAB 25; "Pagina"
```

sería una forma de escribir la cabecera de la primera página de un libro.

Pruebe ahora este programa

```
10 FOR n=0 TO 20
20 PRINT TAB 8*n;n;
30 NEXT n
```

Esto muestra lo que entendemos por 'tomar el número de columna módulo 32'.

Pruebe el programa después de cambiar el 8 de la línea 20 por un 6.

Observe los siguientes detalles:

(i) Las cláusulas TAB y los elementos de PRINT normalmente deberían terminar en punto y coma. No es que no se pueda poner comas (o nada, al final de la sentencia), pero, después de haber seleccionado tan cuidadosamente la posición de escritura, no nos interesa volver a desplazarla inmediatamente.

(ii) No se puede escribir en las dos últimas líneas de la pantalla (22 y 23) porque están reservadas para órdenes, captación de datos con INPUT, mensajes de error, informes, etc. Así pues, cuando en lo sucesivo hablemos de 'última línea' normalmente nos estaremos refiriendo a la línea 21.

(iii) Se puede utilizar AT para situar la posición de escritura en una posición en la que ya haya algo escrito (el nuevo elemento sencillamente reemplazará al antiguo).

Otra sentencia relacionada con PRINT es CLS, cuyo efecto es borrar toda la pantalla.

Cuando al escribir llegamos a la parte inferior de la pantalla, el texto empieza a desplazarse hacia arriba como si se tratase del folio de una máquina de escribir. Para comprobar cómo funciona este mecanismo, seleccione la opción Pantalla en el menú de edición (descrito en el capítulo 6) y luego escriba:

```
CLS: FOR n=1 TO 30: PRINT n: NEXT n
```

Cuando se llena la pantalla, el +2 se detiene y emite el mensaje scroll? ('¿desplazar?') en la parte inferior de la misma. Ahora puede usted observar a su gusto los 22 primeros números. Cuando haya terminado, pulse Y (para *yes*, 'sí') y el +2 mostrará la siguiente tanda de números. En realidad, da lo mismo pulsar Y que cualquier otra tecla, a excepción de N, la barra espaciadora y [BREAK]. Estas últimas hacen que el programa se detenga y que el ordenador emita el informe D BREAK – CONT repeats.

La sentencia INPUT es capaz de hacer mucho más de lo que le hemos exigido hasta ahora. Ya hemos visto sentencias INPUT similares a

```
INPUT "¿Cuantos años tienes?", edad
```

en la cual el +2 escribe la frase ¿Cuantos años tienes? en la parte inferior de la pantalla y queda a la espera de que el usuario introduzca un número. Sin embargo, una sentencia INPUT puede incluir elementos y separadores, exactamente igual que las sentencias PRINT. ¿Cuantos años tienes? y edad son elementos de INPUT.

Los elementos de INPUT son generalmente iguales a los de PRINT, aunque hay algunas diferencias importantes:

Primero, un elemento adicional obvio de INPUT es la *variable* a la que se asigna valor en la sentencia (en el ejemplo anterior, edad). La regla es que si un elemento INPUT comienza con una letra, BASIC considera que se trata de una variable cuyo valor se va a introducir.

Podría parecer que esto implica que no podremos escribir los valores de las variables como parte de un mensaje inductor. (*Mensaje inductor* es el que se escribe en una sentencia INPUT para indicar al usuario cuál es la naturaleza de los datos que debe introducir.) Sin embargo, esto se resuelve poniendo la variable entre paréntesis. Cualquier expresión que empiece con una letra debe ir entre paréntesis si queremos que INPUT la escriba como parte del mensaje inductor.

Cualquier clase de elemento de PRINT al que no afecten estas reglas es también un elemento de INPUT. Veamos un ejemplo:

```
LET mi edad = INT (RND*100): INPUT ("Tengo ";mi edad;" años.");
"¿Cuantos años tienes tú?", tu edad
```

Puesto que *mi edad* está entre paréntesis, INPUT escribe su valor. En cambio, *tu edad* no va entre paréntesis, y por eso INPUT entiende que se trata de la variable a la que debe asignar el valor captado.

Todo lo que escribe una sentencia INPUT va a la pantalla inferior, que actúa de forma bastante independiente de la pantalla superior. En particular, sus líneas se numeran a partir de la primera línea de la propia pantalla inferior, aun cuando ésta haya crecido hacia arriba en el monitor (lo cual ocurre cuando hay que escribir muchos datos en respuesta a INPUT). Cualquier cosa que haga la pantalla inferior durante la ejecución de INPUT, su tamaño siempre revertirá a las dos líneas habituales en cuanto se detenga el programa y volvamos al modo de edición.

Para ver cómo funciona AT en una sentencia INPUT, pruebe lo siguiente:

```
10 INPUT "Esta es la linea 1",a$; AT 0,0;"Esta es la linea 0",a$;AT 2,0;"Esta es la
   linea 2",a$; AT 1,0;"Esta sigue siendo la linea1",a$
```

Ejecute este programa (basta con que pulse INTRO cada vez que se detenga). Cuando el programa escribe Esta es la linea 2, la pantalla inferior se desplaza hacia arriba para dejarle sitio; pero también la numeración se desplaza hacia arriba, de forma que las líneas del texto conservan los mismos números.

Ahora pruebe esto:

```
10 FOR n=0 TO 19: PRINT AT n,0;: NEXT n
20 INPUT AT 0,0;a$; AT 1,0;a$; AT 2,0;a$; AT 3,0;a$; AT 4,0;a$; AT 5,0;a$;
```

Cuando la pantalla inferior empieza a subir, la pantalla superior permanece inalterada, hasta que la inferior amenaza con escribir en la misma línea que la última sentencia PRINT. Entonces la pantalla superior se desplaza hacia arriba para impedirlo.

Otro refinamiento de la sentencia INPUT que no hemos visto todavía es la opción LINE, que nos ofrece otra forma de captar por el teclado los valores de las variables literales. Si ponemos LINE antes del nombre de la variable literal que va a ser captada, como por ejemplo en

```
INPUT LINE a$
```

el +2 no mostrará las comillas como de costumbre (aunque para sus adentros actuará como si las hubiera escrito). Entonces, si respodemos escribiendo

```
gato
```

INPUT asignará el valor gato a a$. Puesto que las comillas no aparecen rodeando a la cadena, no podremos borrarlas. Recuerde que no se puede usar LINE cuando la variable es numérica.

Hay un interesante efecto secundario de INPUT. Mientras se está respondiendo a una sentencia INPUT, el antiguo sistema Spectrum de introducción de palabras clave por la pulsación de una sola tecla se comporta de una forma extraña, hasta que lo hacemos entrar en vereda al pulsar INTRO. Ejecute este programa:

```
10 INPUT numeros
20 PRINT numeros
30 GO TO 10
```

Introduzca unos cuantos números y verá cómo éstos son reproducidos fielmente en la pantalla. Ahora pulse la tecla EXTRA seguida de M. Aparece la palabra PI; si ahora pulsa INTRO, el +2 escribe 3.1415927 como por arte de magia. Sin embargo, si usted

escribe las letras PI sin pulsar antes [EXTRA] y M, el +2 detiene el programa y emite el informe

2 Variable not found, 10:1   ('Variable no encontrada')

Otra prueba: ejecute el programa; pulse [EXTRA] y H; aparece SQR; si ahora introduce 25 y pulsa [INTRO], el +2 responde escribiendo el número 5, que es la raíz cuadrada de 25.

No hay una explicación sencilla para este comportamiento. En todo caso, conviene saber que cosas de éstas pueden suceder si pulsamos ciertas combinaciones de teclas en respuesta a INPUT.

Los caracteres de control CHR$ 22 y CHR$ 23 producen unos efectos similares a los de AT y TAB. Siempre que se le pide al +2 que «escriba» uno de ellos, el carácter debe ir seguido por dos caracteres más, que son tratados por el ordenador como parámetros de AT o de TAB. Normalmente es más cómodo usar explícitamente AT y TAB que estos dos caracteres, aunque hay situaciones en que pueden ser útiles.

El carácter de control que corresponde a AT es CHR$ 22. El primer número que se especifica a continuación es el número de fila; el siguiente, el número de columna. Así,

```
PRINT CHR$ 22+CHR$ 1+CHR$ c;
```

equivale exactamente a

```
PRINT AT 1,c;
```

Esto es así a pesar de que CHR$ 1 o CHR$ c tendrían en otra situación significados diferentes (por ejemplo, cuando c=13); el CHR$ 22 que los precede les cambia el significado.

El carácter de control que corresponde a TAB es CHR$ 23; los dos números que le siguen se combinan para dar un número del margen de 0 a 65535, que es el que se toma como parámetro de TAB. La sentencia

```
PRINT CHR$ 23+CHR$ a+CHR$ b;
```

es equivalente a

```
PRINT TAB a+256*b;
```

Con POKE podemos hacer que el ordenador deje de preguntarnos si queremos desplazar la pantalla ('scroll?'). Para ello podemos escribir

```
POKE 23692,255
```

de vez en cuando. Después de hacer esto, la pantalla tendrá que desplazarse 255 veces antes de que el +2 nos haga la pregunta scroll?. Como ejemplo, pruebe

```
10 FOR n=0 TO 1000
20 PRINT n: POKE 23692,255
30 NEXT n
```

y observe cómo los números se escapan de la pantalla.

Ejercicio

1. Pruebe este programa con algún niño para ver si se sabe las tablas de multiplicar:

```
 10 LET m$=""
 20 LET a=INT (RND*10)+1: LET b=INT (RND*10)+1
 30 INPUT (m$) ' ' "¿Cuanto es ";(a);" por ";(b);"?";c
100 IF c=a*b THEN LET m$="Bien!": GO TO 20
110 LET m$="Mal. Inténtalo otra vez.": GO TO 30
```

   Si el niño es un poco despabilado, puede darse cuenta de que no necesita hacer el cálculo él mismo. Por ejemplo, si el +2 le pregunta que cuánto es 2 por 3, todo lo que tiene que hacer es escribir 2*3 literalmente.

Parte 16
Colores

Temas tratados

INK, PAPER, FLASH, BRIGHT, INVERSE, OVER, BORDER

Pruebe este programa

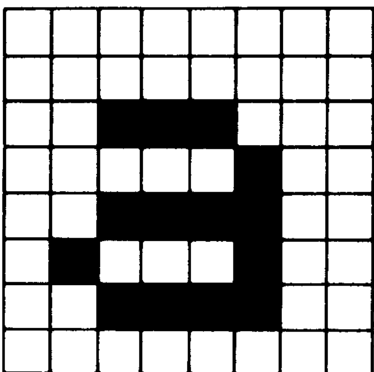
```
 10 FOR m=0 TO 1: BRIGHT m
 20 FOR n=1 TO 10
 30 FOR c=0 TO 7
 40 PAPER c: PRINT "    ";: REM  4 espacios de colores
 50 NEXT c: NEXT n: NEXT m
 60 FOR m=0 TO 1: BRIGHT m: PAPER 7
 70 FOR c=0 TO 3
 80 INK c: PRINT c;"  ";
 90 NEXT c: PAPER 0
100 FOR c=4 TO 7
110 INK c: PRINT c;"  ";
120 NEXT c: NEXT m
130 PAPER 7: INK 0: BRIGHT 0
```

Esto muestra los ocho colores (incluidos el blanco y el negro) y los dos niveles de brillo que puede producir el +2 en un televisor de color. (Si su aparato es de blanco y negro, lo que se aprecia es diversas intensidades de gris.) Una forma más rápida, y drástica, de obtener el mismo resultado consiste en reinicializar (botón **RESET**) el +2 mientras se tiene pulsada la tecla [BREAK]. Veamos la lista de los colores y sus códigos:

0 negro
1 azul
2 rojo
3 magenta
4 verde
5 cyan
6 amarillo
7 blanco

En los televisores de blanco y negro, estos números están por orden de luminosidad decreciente. Para usar los colores adecuadamente es necesario entender cómo se forma la imagen en la pantalla.

La imagen está dividida en 768 posiciones (24 filas por 32 columnas), llamadas *celdas*, en las que podemos escribir los caracteres.



Una celda de carácter típica

Cada celda de carácter consiste en una retícula de 8x8 puntos (como la de la figura anterior). Esto debería recordarle a usted los gráficos definidos por el usuario de la Parte 14, donde representábamos con un 0 los puntos blancos y con un 1 los negros.

El carácter tiene dos colores asociados con él: la *tinta*, o color del primer plano, que es el color con el que el +2 dibuja los puntos negros de nuestro cuadrado, y el *papel*, o color del fondo, que es el usado para los puntos blancos. Inicialmente todas las celdas tienen tinta negra y papel blanco, de forma que los caracteres aparecen en negro sobre blanco.

El carácter también tiene un nivel de brillo o luminosidad (normal o extra) asociado, y algo que indica si parpadea o no. El parpadeo consiste en un intercambio continuo de los colores de la tinta y el papel. Toda esta información puede ser codificada en números, de modo que un carácter tiene lo siguiente:

(i) Una retícula de ceros y unos que definen la forma del carácter (0 para el papel, 1 para la tinta).

(ii) Un código para el color de la tinta y otro para el del papel, cada uno codificado mediante un número del 0 al 7.

(iii) Un código para el brillo (0 para el normal, 1 para el extra).

(iv) Un código para el parpadeo (0 para constante, 1 para parpadeo).

Puesto que cada código de color de tinta y de papel se refiere a una celda de carácter completa, es imposible tener más de dos colores en un bloque dado de 64 puntos. Lo mismo ocurre con los códigos de brillo y parpadeo: se refieren a la célula de carácter completa, no a puntos individuales dentro de ella. Los códigos de color, brillo y parpadeo de un carácter dado son lo que denominamos *atributos* de ese carácter.