Estructura

En general, un programa BASIC está compuesto por una secuencia de líneas de programa, las cuales son ejecutadas en su orden, comenzando por la primera.

Así pues, el punto de entrada de programa es la primera línea existente.

En el BASIC original, cada línea de programa debía de ir numerada, con un número entero comenzando en 1.

Típicamente, la numeración comenzaba en 10, teniendo un salto de 10 entre dos líneas consecutivas, así por ejemplo, las 3 primeras línea un programa serían la 10, 20 y 30.

Versiones posteriores de BASIC permitieron eliminar el uso de tales números, y usar en su lugar las llamadas “etiquetas”.

iBASIC permite ambas formas de programa, con y sin números de línea.

Cada línea de programa puede tener una o más órdenes de programa, esto es, acciones que se deben hacer, aunque lo normal es que contenga una única orden.

Las órdenes, dentro de una misma línea, se separan por el signo “:”, y se ejecutan en el orden natural de aparición.

El BASIC convencional no es procedimental, sino lineal.

Todas las variables son globales.

Sin embargo, sí que existen subrutinas (orden GOSUB).

iBASIC es tanto procedimental como lineal, permite procedimientos (con ámbito local), subrutinas y ejecución lineal.

iBASIC también permite subprograma, esto es, ficheros de códigos separados que pueden ser llamados desde el programa principal. Típicamente estos subprogramas contendrán procedimientos, para ser formar librerías, pudiendo ser utilizadas desde varios programas.

Cuando un programa BASIC se ejecuta en iBASIC, éste comienza a hacerlo en la primera línea existente, como se ha comentado. Puede haber partes del código en donde estén definidos los procedimientos, a los cuales se podrán llamar.

Como resumen, un programa BASIC tendría el siguiente aspecto:

```
10 orden
20 orden
30 orden:orden:orden
40 orden
.
.
65535 orden
```

Rotura de líneas

Habrá ocasiones en las que una orden de programa sea demasiado larga y que pueda ser contenida en una sola línea, o bien, por razones estéticas o de compresión sea aconsejable dividirla en más de una línea.

Para ello se utiliza el carácter especial "\" que indica a iBASIC que la orden actual continúa en la siguiente línea de programa.

Sea el ejemplo:

```
IF a = 1 or b = 2 or c = 3 or d = 4 or a = b or b = c THEN PRINT "Valores de a,b,c,d: ", a, b, c, d
```

Es equivalente a:

```
IF a = 1 or \
   b = 2 or \
   c = 3 or \
   d = 4 or \
   a = b or \
   b = c THEN PRINT "Valores de a,b,c,d: ", a, b, c, d
```

En la línea en donde aparezca el carácter "\", a continuación de él no puede aparecer nada (excepto comentarios). En caso contrario se generaría un error:

```
IF a = 1 or \ b = 2 or        → ERROR
   c = 3 THEN PRINT a, b
```

Numeración de líneas. Etiquetas.

Como se ha comentado, un programa BASIC puede tener o no números de línea.

Los números de línea se utilizan como números de secuencia de las órdenes, y además como referencia para las órdenes de salto, como GOTO o GOSUB, las cuales cambian el flujo de ejecución.

El uso de tales números tiene más inconvenientes que ventajas, por lo que estos pueden ser eliminados, y usar "etiquetas" en aquellos puntos en donde sea necesario, que serán aquellos usados por las órdenes de salto.

En iBASIC, la definición de las etiquetas se hace por medio de la orden especial "LABEL", o bien, usando el carácter ":".

En ambos casos, la definición de una etiqueta debe realizar al comienzo de una línea:

```
orden
orden
LABEL etiqueta
orden
```

equivale a:

orden
orden
etiqueta:
orden

iBASIC dispone de una orden especial (DELETE LINE NUMBERS) para eliminar los números de línea de un programa existente, creando automáticamente las etiquetas que únicamente sean necesarias.

Edición de programas

Desde el punto de vista práctico, un programa BASIC es un fichero de texto que contiene las órdenes a ejecutar.

Por tanto, este fichero puede ser creado con cualquier editor, siempre y cuando su salida sea texto plano.

En el caso de Windows, se pueden utilizar los básicos editores Notepad o Wordpad.

Como alternativa, se puede utilizar el editor de línea que dispone iBASIC, cuyo funcionamiento es muy parecido a los intérpretes convenciones de BASIC, pudiendo crear líneas, modificarlas, borrarlas, y más específicamente, renumerar o eliminar los números de línea.

Cuando se entra en iBASIC se muestra la pantalla del intérprete, en modo interactivo (como los tradicionales).

En este modo, iBASIC está a la espera de recibir órdenes directas por teclado, o bien, recibir una nueva línea de programa.

En esta situación, se puede empezar a escribir un programa, directamente, como el siguiente:

```
10 CLS
20 PRINT “Hello World”

RUN
```

Las dos primeras línea (10 y 20) son línea de programa, que son guardadas en memoria, pero no ejecutadas.

La última línea, “RUN”, es una orden que será ejecutada de forma inmediata, y cuya finalidad es ejecutar el programa que está en memoria, esto es, las líneas 10 y 20.

Sin embargo, la recomendación es crear los programas con un editor externo y posteriormente cargarlos en iBASIC usando la orden LOAD.

De esta manera se ejecutarán los programas, y se podrán depurar o modificar, usando el resto de órdenes de edición:

Instrucción	Descripción
EDIT	Editar una línea para modificarla

LIST	Lista líneas del programa
DELETE	Borrar líneas del programa
LOAD	Cargar un fichero de programa
SAVE	Grabar programa actual a fichero
NEW	Borrar programa actual de la memoria
RUN	Ejecutar programa actual
AUTO	Activar autonumeración de líneas
RENUM	Renumerar líneas del programa
DELETE LINE NUMBERS	Eliminar números de líneas del programa
CHECK SYNTAX	Comprobar errores de sintaxis del programa

Para una descripción en detalle de estas órdenes, ver el manual de referencia.

Algunos ejemplos:

EDIT 10 : edita la orden 10

LIST: lista todo el programa

LIST 100- : lista desde la orden 100

DELETE 100 : borra la orden 100

DELETE 100-200 : borra todas las órdenes entre la 100 y 200.

Cada vez que se carga un programa, se crea o se modifica una línea, iBASIC realiza una primera comprobación de su sintaxis. Sin embargo, adicionalmente, puede emplearse la orden CHECK SYNTAX para realizar una comprobación más exhaustiva, lo cual reducirá los errores en tiempo de ejecución.

La orden AUTO se utiliza únicamente para crear programas dentro de iBASIC, y su uso evita tener que escribir el número de cada una de las líneas que se escriben.

Con RENUM se realiza una renumeración automática de las líneas del programa.

Carga y grabación de programas

Para la carga y grabación hay dos órdenes, LOAD y SAVE, respectivamente.

De forma general, un fichero de programa BASIC será un fichero de texto, sin embargo, iBASIC también soporta ficheros de programa binarios, con y sin protección.

En caso de usar un editor externo, siempre deberán de guardarse los programas en ficheros de texto plano.

Los programas guardados en texto, pueden estar codificados usando el juego de caracteres Windows ANSI o el juego de caracteres ASCII OEM.

Por defecto, para la carga y grabación de programa en texto, iBASIC utiliza ANSI. Sin embargo puede especificarse el uso de ASCII OEM (opción “,A” en las órdenes LOAD y SAVE).

Los ficheros binarios de programa son únicamente producidos por iBASIC, mediante la orden SAVE.

Con la orden LOAD se carga un programa en memoria (sin ejecutarlo), ya sea de texto o binario (protegido o no):

LOAD "programa1" : carga el programa contenido en el fichero "programa1.bas"

Observar que no es necesario indicar la extensión ".bas".

Una vez cargado el programa, se puede ejecutar (RUN), listar (LIST), modificar (EDIT), etc.

En el caso de cargar ficheros binarios protegidos, el programa solo podrá ser ejecutado, pero no visualizado ni modificado con ninguna orden, ese es el carácter de la protección.

Opcionalmente, se puede usar la orden RUN para cargar y ejecutar, simultáneamente, un programa:

RUN "programa1"

Con la orden SAVE se guarda el programa actual en memoria a un fichero de texto o binario:

SAVE "programa1" : guarda el programa en el fichero de texto "programa1.bas" (juego caracteres ANSI).

SAVE "programa1", A : igual al anterior pero usando juego de caracters ASCII OEM.

SAVE "programa1", B : guarda el programa en el fichero binario "programa1.bas" (no protegido)

SAVE "programa1", P : guarda el programa en el fichero binario protegido "programa1.bas".

La protección, como se ha indicado, consiste en que el programa no podrá ser visualizado ni editado, solo ejecutado.

Variables

En iBASIC las variables son locales al ámbito de ejecución actual, esto es, dependiendo del procedimiento en que se encuentren.

Las variables no necesitan declaración y se usan directamente, tomando todas ellas valor cero inicial.

En iBASIC se distinguen dos tipos de variables:

Numéricas: almacenan un número en punto flotante de 64 bits

De cadena: almacenan una cadena o buffer de caracteres de longitud entre 0 y 2^{32}-1 caracteres.

La distinción de tipo se realiza en el nombre de la variable.

Aquellas que tengan un carácter $ al final del nombre serán de cadena, en caso contrario, serán numéricas:

valor → variable numérica

nombre$ → variable de cadena

El identificador de una variable está compuesto de una serie de caracteres alfanuméricos o de caracteres "_", no pudiendo empezar por un número:

Valor, valor12, valor12modif, _valor → variables válidas

12valor → variable no válida

En BASIC no se hace distinción de mayúsculas/minúsculas en los nombres de variables.

La asignación de valor a una variable se realiza de forma directa:

variable = valor

iBASIC también soporta la orden histórica de asignación LET, pero su uso no es obligatorio:

LET *variable = valor*

Para las variables numéricas, "valor" es una expresión numérica, que puede tener cualquier cantidad de factores, números, variables o funciones.

De igual modo, para variables de cadena, "valor" es una expresión de cadena que puede tener cualquier cantidad de cadenas, literales, variables o funciones.

Como se ha mencionado, las variables no necesitan ser declaradas, y pueden usarse en cualquier momento.

Por ejemplo:

```
10 PRINT A
20 A=1
30 PRINT A

RUN
```

mostrará:

```
0
1
```

Algunos ejemplos de asignación:

```
valor = 100
valor2 = valor + 2
nombre$="Jose"
```

```
apellidos$=”del Rio”
nombre_completo$ = nombre$ + “ “ + apellidos$
```

El borrado del contenido de una variable puede hacerse manualmente a través de una asignación a valor cero, o cadena vacia, o bien, utilizar la orden CLEAR.

Valores numéricos:

Por defecto, los valores literales numéricos se indican en decimal (base 10), sin embargo también es posible utilizar números hexadecimales (base 16) y binarios (base 2).

Para especificar un número hexadecimal, se utilizará como prefijo “&H” ó “0x”.

Para especificar un número binario, se utilizará “&B” ó “0b”.

Sistema (Base)	**Prefijo**
Decimal (10)	-
Hexadecimal (16)	&H ó 0x
Binario (2)	&B ó 0b

El ejemplo siguiente imprime el número 42, usando los tres sistemas:

```
PRINT 42
PRINT &H2A
PRINT &B101010
```

De forma simétrica, las funciones HEX$ y BIN$ devuelven la cadena hexadecimal o binario con la representación del número que se indique.

Valores de cadena:

La especificación de una cadena literal se realiza entrecomillando su contenido (con comillas dobles).

Como excepción, si la cadena termina en fin de línea, la doble comilla final puede omitirse:

```
PRINT “La casa grande del campo”
```

Puede escribirse también como:

```
PRINT “La casa grande del campo  → No tiene las comillas del final
```

En caso de que una cadena debe contener la doble comilla, ésta puede especificarse indicando dos veces dicho signo:

```
PRINT “Las casa “”grande”” del campo” → imprime: La casa “grande” del campo
```

Cuando se necesiten incorporar signos no imprimibles en la cadena, deberá hacerse uso de la función CHR$:

```
PRINT “La casa grande”+CHR$(10)+”del campo” → impresión en dos líneas
```

Ámbitos de ejecución

iBASIC permite programas con o sin procedimientos.

Se define como “ámbito” a una región del programa que tiene su propio entorno de ejecución y sus propias variables (locales).

En un programa sin procedimientos, sólo existe un único ámbito (que es el propio programa en sí), pudiendo decirse que todas las variables son globales.

En un programa con procedimientos, habrá una zona del programa que es lineal, siendo el ámbito principal del programa, y otras zonas en donde están los procedimientos, que tienen cada uno de ellos su propio ámbito.

Las variables solo están visibles dentro de su propio ámbito, por lo que, un procedimiento no puede acceder a variables de otro, pero sí a las variables globales del ámbito principal.

Ejemplo:

```
10 ‘ Este es el ámbito principal
20 a = 1 : b = 2
30 call sumar(a, b, c)
40 print c
50 end
60 ‘ Procedimiento “sumar”. Tiene un propio ámbito
70 procedure sumar(in n1, in n2, out res)
80    print a, b, c  → imprime: 0  0  0
90    res = n1 + n2
100 endproc
```

En éste ejemplo, el ámbito principal está formado por las líneas 10 a 50, que es lo que empieza a ejecutarse. Aquí se definen las variables “a”, “b” y “c”.

Cuando se llama al procedimiento “sumar”, se crea un nuevo ámbito, por lo que la impresión de las variables “a”,”b” y “c” dan cero (son variables que no existen en su ámbito).

Acceso a variables globales:

Las variables globales son aquellas que se definen en el ámbito principal del programa.

Un procedimiento sólo podrá acceder a sus variables locales y a los globales.

El acceso a las variables globales se hace añadiendo el prefijo “__” (dos caracteres de subrayado) al nombre de la variable global en concreto.

El siguiente ejemplo define un procedimiento de dividir que accede a la variable global “error”, poniéndola a “true” en caso de que el divisor sea cero:

```
error = false
call dividir(100, 2, a):print a, error  → imprime 50   0
```

```
call dividir(100, 0, a):print a, error → imprime  0   1
end
procedure dividir(in a, in b, out res)
   __error = (b = 0)
  if not __error then res = a / b else res = 0
endproc
```

Matrices

Además de las variables escalares (que contienen un único valor), también se soportan matrices o arrays de varias dimensiones, tanto numéricas como de cadena.

Al contrario que las variables escalares, las matrices sí que deben ser declaradas, tanto en tipo como en tamaño (excepto en modo matricial), usándose para ello la orden DIM.

Algunos ejemplos:

```
DIM valores(100)
DIM nombre$(200)
DIM mapa(100,2)
```

El número (o números) indicados entre paréntesis son los índices máximos del array.

El índice para acceder a un elemento de un array por defecto comienza en cero, pero, usando la orden OPTION BASE, puede especificarse que sea en uno. El último valor de índice permitido es el indicado en DIM.

El acceso, tanto para lectura como escritura, a un elemento de un array, se realiza como una variable escalar, pero indicando entre paréntesis el índice:

```
valores(1) = 10
nombre$(100) = “Sofia”
mapa(5,1) = 20
```

Como se ha indicado, el uso de OPTION BASE permite modificar el valor del primer índice:

```
OPTION BASE 0
DIM valores(100)
‘ Indices válidos: desde 0 a 100

OPTION BASE 1
DIM valores(100)
‘ Indices válidos: desde 1 a 100
```

En el modo de funcionamiento normal de iBASIC (modo escalar), no se permite asignar matrices completas:

```
DIM matriz1(100), matriz2(100)
matriz1 = matriz2 → no permitido
```

Sin embargo, en el modo matricial esto sí que se permite, e incluso realizar operaciones a nivel de toda la matriz (o de submatrices de ella):

```
SET MODE MATRIX
DIM matriz1(100), matriz2(100)
matriz1 = 10 → todos los elementos a valor 10
matriz2 = matriz1
```

Expresiones numéricas

Una expresión numérica, como su nombre indica, es una expresión, o fórmula, que puede estar formada por números literales, variables y por funciones numéricas. También se permiten, dentro de la expresión numérica, expresiones lógicas y de comparación.

En cualquier orden que se necesite un argumento numérico puede usarse una expresión, pudiéndose usar paréntesis para agrupar términos.

Operaciones aritméticas:

Operador	Descripción
+	Suma
-	Resta
*	Multiplicación
/	División
^	Potencia
MOD	Resto de división entera
%	Igual a MOD

Operadores de comparación:

Operador	Descripción
=	Igual que
>	Mayor que
>=	Mayor o igual que
<	Menor que
<=	Menor o igual que
<>	Distinto que
IN […]	Incluido en

Operadores lógicos:

Operador	Descripción
AND	AND lógico a nivel de bit
OR	OR lógico a nivel de bit
XOR	XOR lógico a nivel de bit

NOT	Negación

Funciones numéricas:

Función	Descripción
RND	Número aleatorio
TIME	Instante de tiempo actual (ms.)
XPOS	Posición X del cursor
YPOS	Posición Y del cursor
XSIZE	Tamaño X de la pantalla
YSIZE	Tamaño Y de la pantalla
FORECOLOR	Color primer plano
BACKCOLOR	Color fondo
CODEPAGE	Código de página juego caracteres ASCII OEM
ERR	Código último error
ERL	Número línea último error
EOF	Indicador fin de fichero
FIELD	Acceso a campo de fichero binario
NREC	Número de registros de fichero
CREC	Número registro actual de fichero
SREC	Tamaño registro de fichero
LEN	Longitud de cadena
VAL	Valor numérico de cadena
ASC	Código ASCII primer carácter de cadena
INSTR	Posición subcadena en cadena
EMPTY	Indicador de cadena vacia
FSIZE	Tamaño de fichero
FEXISTS	Indicador existencia de fichero
DEXISTS	Indicador existencia de directorio
CVB	Convierte buffer a entero 8 bits
CVI	Convierte buffer a entero 16 bits
CVL	Convierte buffer a entero 32 bits
CVS	Convierte buffer a punto flotante 32 bits
CVD	Convierte buffer a punto flotante 64 bits
SIN	Seno
COS	Coseno
TAN	Tangente
ASIN	Arco seno
ACOS	Arco coseno
ATAN	Arco tangente
EXP	Potencia de número “e”
LOG	Logaritmo neperiano
SQRT	Raíz cuadrada
ROUND	Redondeo de número
SGN	Signo de número
INT	Parte entero
ABS	Valor absoluto
MAX	Máximo valor

MIN	Mínimo valor
AVG	Media aritmética
SUM	Suma de valores
ROL	Desplazamiento lógico a izquierda
ROR	Desplazamiento lógico a derecha
NEG	Complemento a 1 (inversión de bits)
SEARCH SEARCHE SEARCHI SEARCHN SEARCHNI	Búsqueda de valor en array
DBOUND	Limites de un array
DATALEN	Número de elementos DATA
CALL	Llamada a procedimiento
IF	Expresión condicional
LET	Asigna valor a variable numérica
EVALUATE	Ejecuta órdenes
FN…	Función de usuario (DEF FN)

Configuración unidad angular:

Operador	Descripción
SET ANGLE RAD	Ángulos en radianes
SET ANGLE DEG	Ángulos en grados sexagesimales
SET ANGLE GRAD	Ángulos en grados centesimales

Algunos ejemplos:

a = 10
b = 1 + a * 2 → b = 21
c = (1 + a) * 2 → c = 22
d = c MOD 3 → d = 1
e = a AND 3 → e = 2
f = a XOR 3 → f = 9
g = (a > 2) AND (b < 30) → g = 1
h = SUM(a, b, c-2) → h = 61
i = ROL(a, 2) → i = 40

Operador IN […]:

Este operador de comparación compruebe si el operando (número o cadena) se encuentra dentro del conjunto indicado entre corchetes.

La especificación del conjunto depende de la naturaleza del operando, si éste es número o cadena:

- **Números**: el conjunto puede estar forma por uno o más intervalos numéricos, separados por comas:

[intervalo, intervalo, …]

Un intervalo puede ser un número, una expresión real de intervalo, o un signo asterisco para indicar todo el rango de números:

número : intervalo compuesto sólo por el número indicado.
desde número .. hasta número : intervalo compuesto desde y hasta los números indicados
desde número .. +incremento : equivale a:

desde número .. desde número + incremento

desde número .. #cantidad : equivale a:

desde número .. desde número + cantidad – 1

desde número .. * : intervalo abierto desde el número indicado hasta el máximo posible.
***** : todo el rango numérico.

- **Cadenas**: el conjunto puede estar formado por una o más cadenas, o por intervalos de cadenas, separadas por comas:

  cadena : intervalo compuesto sólo por la cadena indicada.

  desde cadena .. hasta cadena : intervalo compuesto desde y hasta las cadenas indicadas.

El siguiente ejemplo muestra el uso con números:

```
INPUT “Introduzca un número: “; num
IF num IN [2, 5, 10..20, 100..*] THEN PRINT “Dentro de intervalo”
```

El ejemplo imprime “Dentro de intervalo” si el número introducido es un 2, un 5, está comprendido entre 10 a 20, o es mayor o igual a 100.

La condición equivalente sin el operador [] sería:

```
IF num = 2 OR num = 5 OR (num >= 10 AND num <= 20) OR num >= 100 THEN …
```

Un ejemplo de operador [] para cadenas es el siguiente:

```
INPUT “Introduzca un nombre: “; nombre$
IF lower$(nombre$) IN [“jose”, “antonio”, “luis”] THEN PRINT “Dentro de intervalo”

INPUT “Introduzca una letra entre de A a E: “; letra$
IF lower$(letra$) IN [“a”..”e”] THEN PRINT “Dentro de intervalo”
```

Modo matricial

En modo matricial las expresiones son de carácter matricial, siendo los operandos matrices.

Los operadores (suma, resta, …) y las funciones matemáticas se aplican término a término a las matrices implicados.

El siguiente ejemplo, suma dos matrices:

```
SET MODE MATRIX
DIM a(2, 2), b(2, 2), c
a = MXRND(2, 2)
b = MXRND(2, 2)
c = a +b
```

En modo matricial las expresiones son de carácter matricial, siendo los operandos matrices.

Ángulos:

En las funciones trigonométricas (SIN, COS, ASIN, …) la unidad de los ángulos que se utilizan puede ser radianes, grados sexagesimales o centesimales.

Para establecer la unidad a usar, se utilizan las órdenes SET ANGLE:

SET ANGLE RAD: radianes (0 .. 2π)
SET ANGLE DEG: grados sexagesimales (0 .. 360^{o})
SET ANGLE GRAD: grados centesimales (gradianes) (0 .. 400^{g})

La unidad por defecto es radián.

Expresiones de cadena

De forma análoga a las expresiones numéricas, en cualquier punto donde se precise una cadena, puede emplearse una expresión que dé como resultado tal cadena.

Una expresión puede tener cadenas literales, variables de cadena o funciones de cadena.

Operaciones:

Operador	Descripción
+	Concatenación
[…]	Subcadena

Funciones de cadena:

Función	Descripción
LEFT$	Subcadena por la izquierda
RIGHT$	Subcadena por la derecha
MID$	Subcadena en medio
STR$	Convierte número a cadena (decimal)

CHR$	Carácter ASCII
HEX$	Convierte número a cadena (hexadecimal)
BIN$	Convierte número a cadena (binario)
LOWER$	Convierte cadena a minúsculas
UPPER$	Convierte cadena a mayúsculas
SPACE$	Cadena de espacios
STRING$	Cadena de repetición de subcadenas
INKEY$	Cadena con la tecla pulsada
TIME$	Hora actual
DATE$	Fecha actual
TRIM$	Eliminar delimitadores
USING$	Formatear número
FIELD$	Campo de cadena de fichero binario
CURDIR$	Directorio actual
CLIPBOARD$	Contenido del portapapeles
CMDLINE$	Línea de comando
MKB$	Crear buffer de número entero de 8 bits
MKI$	Crear buffer de número entero de 16 bits
MKL$	Crear buffer de número entero de 32 bits
MKS$	Crear buffer de número punto flotante 32 bits
MKD$	Crear buffer de número punto flotante 64 bits
SCRCHR$	Obtener caracteres de una zona de la pantalla
ERR$	Cadena descripción del último error
ERP$	Nombre subprograma del último error
CALL$	Llamada a procedimiento
IF$	Expresión condicional
LET$	Asigna cadena a una variable
EVALUATE$	Ejecuta órdenes
ANSI$	Convertir cadena ASCII OEM a ANSI
ASCII$	Convertir cadena ANSI a ASCII OEM
FN …	Función de usuario

Operador []:

Este operador es una abreviación de las funciones LEFT$, RIGHT$ y MID$, permitiendo extraer subcadenas de una cadena.

Su sintaxis es idéntica a la especificación de intervalos del operador IN [], pero ahora trasladado a los caracteres internos de la cadena:

> ***cadena* [*índice*] :** extrae el carácter del índice indicado de la cadena
>
> ***cadena* [*índice inicial .. índice final*]** : extrae la subcadena formada por los caracteres entre los índices indicados.
>
> ***cadena* [*índice inicial .. **]** : extrae la subcadena formada desde el carácter “índice inicial” hasta el final de la cadena.
>
> ***cadena* [*índice inicial .. +número caracteres extra*] :** equivale a:

cadena [*índice inicial .. índice inicial +número caracteres extra*]

***cadena* [*índice inicial .. #número caracteres*] :** equivale a:

cadena [*índice inicial .. índice inicial +número caracteres - 1*]
MID$(*cadena*, *índice inicial*, *número caracteres*)

***cadena* [*] :** extra toda la cadena.

Algunos ejemplos con variables:

a$ = “el campo es verde”
b$ = a$[1] → b$ = “e”
c$ = a$[1 .. 2] → c$ = “el”
d$ = a$[4 .. 8] → d$ = “campo”
e$ = a$[4 .. +4] → e$ = “campo”
f$ = a$[4 .. #5] → f$ = “campo”
g$ = a$[13 .. *] → g$ = “verde”

El operador [] puede usarse también con cadenas literales y funciones:

PRINT “La casa es grande”[4 .. 7] → imprime “casa”
PRINT TIME$[1 .. 2] → imprime las dos cifras de la hora actual

Algunos ejemplos de expresiones de cadena:

a0$ = “el campo “
a$ = a0$ + “es verde” → a$ = “el campo es verde”
b$ = left$(a$, 2) → b$ = “el”
c$ = mid$(a$, 4, 5) → c$ = “campo”
d$ = trim$(“ campo “) → d$ = “campo”
e$ = space$(5) → e$ = cadena con 5 espacios
f$ = left$(a$[4..8], 2) → f$ = “ca”
g$ = right$(a$ + space$(10) + chr$(65), 2) → g$ = “ A”

También se puede usar [] como especificación del destino de una asignación de cadena, sustituyendo los caracteres especificados en [] por la cadena a asignar:

A$ = “La casa es grande”
A$[1..2] = “Una” → A$ = “Una casa es grande”
A$[13..+5] = “pequeña” → A$ = “Una casa es pequeña”
A$[10..12] = “” → A$ = “Una casa pequeña”

Condicionales

Los condicionales son instrucciones especiales que evalúan una condición y en función de su resultado ejecutan o no una serie de instrucciones.

iBASIC permite los condicionales convencionales de BASIC, de una sola línea, y condicionales de bloque, formados por varias líneas.

Adicionalmente también se disponen de dos funciones condicionales, IF() e IF$(), que permiten devolver un valor determinado en función del cumplimiento de una condición.

Condicionales de una línea

El formato general de la instrucción es el siguiente:

IF *expresión condición* THEN *ordenes*
IF *expresión condición* THEN *ordenes* ELSE *ordenes*

"expresión condición" es la condición a evaluar, que en caso de ser verdadera, se ejecutan las órdenes que están a continuación del THEN, pero no las del ELSE.

En caso de que la condición no se cumpla (sea falsa) el comportamiento es el inverso, se ejecutan las del ELSE pero no las del THEN.

La "expresión condición" es una expresión numérica normal, de forma que se interpreta como "verdadera" (la condición se cumple) si el resultado de la expresión es distinto de cero. En caso de ser cero, se considera condición falsa (no se cumple).

En las órdenes que van a continuación del THEN o ELSE, se pueden agrupar más de una por medio del signo ":".

En caso de condiciones complejas, es preferible el uso de los condicionales de bloque.

Un operador, para la condición, bastante útil es IN [], que permite determinar si un valor está dentro del intervalo, o de los valores, indicado dentro de los corchetes:

Especificación de valores: IN [*valor1, valor2, …*]
Especificación de intervalo: IN [*valor1..valor2*]

Esto es, los valores se indican tal cual, y los intervalos por medio de los dos puntos.

En un mismo IN [] pueden indicarse varios valores e intervalos, separándolos por comas.

Ejemplos:

```
a = 10
b = 1
nombre$="Jose"
IF a > 5 THEN PRINT "'a' es mayor que 5" ELSE PRINT "'a' es menor  o igual que 5"
IF a > 5 OR b = 0 THEN PRINT "esto se imprimirá"
IF nombre$="Jose" AND a > 5 THEN PRINT "esto también se imprimirá"
IF a IN (1..20, 22, 30..40) THEN PRINT "'a' en intervalo":a = 100
IF 0 THEN PRINT "esto nunca se imprimirá"
```

Condicionales de bloque

Estos condicionales permiten especificar las instrucciones del THEN o del ELSE en línea físicas separadas, lo cual es de gran utilidad en caso de tener varias órdenes, o de usar condicionales anidados.

El formato es:

```
IF expresión condición THEN
.
.
ELSE
.
.
ENDIF
```

El bloque ELSE es opcional, pudiendo quedar:

```
IF expresión condición THEN
.
.
ENDIF
```

En cualquier caso, siempre hay que terminar la condición de bloque con ENDIF.

La primera línea que comienza la condición IF debe de acabar en THEN, y no tener instrucciones detrás de él.

Para este tipo de condicionales en bloque se puede usar la instrucción BREAKIF cuyo efecto es saltar todo el resto del condicional y salir fuera de él. Esta orden será útil en condicionales complejos.

Ejemplos:

```
a = 10
IF a IN (1..20, 22, 30..40) THEN
   PRINT "'a' en intervalo"
   IF a < 30 THEN BREAKIF
   a = 100
   IF a > 50 THEN PRINT "esto se imprimirá"
ENDIF
```

Condicionales múltiples de selección (SELECT-CASE-ENDSEL)

Este tipo de condicionales contienen un conjunto de alternativas, de las cuales solo se ejecutará una en función del valor de un selector.

El formato general es el siguiente:

```
SELECT expresión selección
CASE lista expresiones caso 1
.
```

```
.
CASE lista expresiones caso 2
.
.
ELSE
.
.
ENDSEL
```

Cada CASE es una alternativa, y el selector está indicado en la propia instrucción SELECT.

En ejecución, cuando se encuentra un SELECT, se ejecutarán aquellas instrucciones del CASE cuya "*lista expresión caso*" contenga el valor de la expresión indicada en SELECT. Si ningún CASE coincide, se ejecutarán las instrucciones del ELSE (si éste existe).

SELECT permite selectores tanto numéricos como de cadena.

La "lista de expresiones" de un CASE puede tener una o más expresiones, e incluso intervalos de valores (para selectores numéricos).

El siguiente ejemplo muestra un SELECT numérico:

```
INPUT "Número opción: ";opcion
SELECT opcion
  CASE  1
    PRINT "Has elegido la opción 1"
  CASE 2, 3
    PRINT "Has elegido la opción 2 ó 3"
  CASE 4..10, 12
    PRINT "Has elegido la opción 4 al 10 ó la 12"
  ELSE
    PRINT "Has elegido otra opción"
ENDSEL
```

Las expresiones de los CASE no han de ser necesariamente literales:

```
SELECT opcion
  CASE  1
    PRINT "Has elegido la opción 1"
  CASE num .. num+2
    PRINT "Has elegido la opción "; num; "al"; num+2
  CASE VAL(a$)
    PRINT "Has elegido la opción "; VAL(a$)
  ELSE
    PRINT "Has elegido otra opción"
ENDSEL
```

El ejemplo siguiente muestra un SELECT de cadena:

```
INPUT “Introduce un dia de la semana: “;dia$
SELECT LOWER$(dia$)
  CASE  “lunes”
    PRINT “Has elegido el lunes”
  CASE “martes”, “miercoles”
    PRINT “Has elegido el martes o el miércoles”
  CASE “”
    PRINT “No has elegido ningún día”
  ELSE
    PRINT “Has elegido otro día de la seman”
ENDSEL
```

En caso de existir varios CASE cuyas expresiones se solapen, se ejecutará el coincidente que primero se encuentre:

```
INPUT “Número opción: “;opcion
SELECT opcion
  CASE  1
    PRINT “Has elegido la opción 1”
  CASE 2, 3
    PRINT “Has elegido la opción 2 ó 3”
  CASE 3
    PRINT “Esto nunca se ejecutará”
  CASE 4..10, 12
    PRINT “Has elegido la opción 4 al 10 ó la 12”
  ELSE
    PRINT “Has elegido otra opción”
ENDSEL
```

El CASE 3 nunca se ejecutará debido a que, delante de existe está CASE 2, 3 que lo contiene.

Condicionales múltiples en cadena (CHAIN-NODE-ENDCHAIN)

Este es un tipo especial de condicional estructurado en bloque, que permite la ejecución de instrucciones condicionadas a un conjunto de condiciones que han de ir cumpliéndose de forma encadenada.

La forma típica de construir este tipo de condiciones es usando sentencias IF anidadas:

```
IF condición 1 THEN
  instrucciones 1
  IF condición 2 THEN
    instrucciones 2
    IF condición 3 THEN
    .
    .
```

Mediante el uso de la estructura CHAIN-NODE-ENDCHAIN se sustituye la construcción anidada de IF por una forma lineal en dónde se indican las condiciones (nodos), quedando implícita su anidación.

También se soporta anidaciones ELSE-IF:

```
IF condición 1 THEN
   instrucciones
  ELSE
      IF condición 2 THEN
         instrucciones
      ELSE
            IF condición 3 THEN
            .
            .
```

El esqueleto de una estructura CHAIN-NODE-ENDCHAIN es el siguiente:

```
CHAIN
NODE expresión 1
      .
      .
      ELSE
      .
      .
NODE expresión 2
      .
      .
      ELSE
      .
      .
ELSENODE expresión
      .
      .
      ELSE
      .
      .
ELSECHAIN
      .
      .
ENDCHAIN
```

Los nodos tipo NODE son los equivalentes a IF-THEN y los de tipo ELSENODE a ELSE-IF-THEN.

La estructura CHAIN está compuesta por una serie de nodos (NODE y ELSENODE) que incluyen una condición y un conjunto de instrucciones asociadas.

Todos los nodos de un mismo tipo deben de estar juntos, debiendo de aparecer los ELSENODE después de los NODE.

Las instrucciones de un nodo de tipo “NODE” sólo se ejecutarán si todas las condiciones de los nodos NODE anteriores se cumplieron y además su propia condición se cumple.

Las instrucciones de un nodo de tipo “ELSENODE” sólo se ejecutarán si su condición se cumple y además:

- El anterior nodo es de tipo “NODE”, y fue el primero en no ejecutarse (su condición no se cumplió pero sí todas las de los nodos anteriores)
- El anterior nodo es de tipo “ELSENODE” y su condición no se cumplió.

Un nodo cualquiera puede ir seguido de un bloque ELSE, que se ejecutará en caso de que la condición del nodo no se cumpla y sea el primer nodo en no ejecutarse.

El bloque ELSECHAIN sólo se ejecutará si alguna de las condiciones de los nodos de tipo NODE no se cumplió, o bien, fallaron todos los nodos de tipo ELSENODE.

Retomando el esqueleto anterior de construcción IF anidada:

```
IF condición 1 THEN
   instrucciones 1
   IF condición 2 THEN
      instrucciones 2
      IF condición 3 THEN
      .
      .
```

Su equivalente en forma de CHAIN sería:

```
CHAIN
  NODE condición 1
    instrucciones 1
  NODE condición 2
    instrucciones 2
  NODE condición 3
    instrucciones 3
ENDCHAIN
```

Llevándolo a un sencillo ejemplo particular:

```
IF a = 1 THEN
   PRINT “a=1”
   IF b = 1 THEN
      PRINT “a=1 y b=1”
```

```
      IF c = 1 THEN
        PRINT “a=1, b=1 y c=1”
      ENDIF
    ENDIF
ENDIF
```

Es equivalente a:

```
CHAIN
  NODE a=1: PRINT “a=1”
  NODE b=1: PRINT “a=1 y b=1”
  NODE c=1: PRINT “a=1, b=1 y c=1”
ENDCHAIN
```

En este ejemplo se observa que CHAIN elimina la complejidad de los anidamientos, quedando una estructura limpia y fácil de leer.

Ampliando el ejemplo con partes ELSE:

```
IF a = 1 THEN
  PRINT “a=1”
  IF b = 1 THEN
    PRINT “a=1 y b=1”
    IF c = 1 THEN
      PRINT “a=1, b=1 y c=1”
    ELSE
      PRINT “a=1, b=1 y c<>1”
    ENDIF
  ELSE
    PRINT “a=1 y b<>1”
  ENDIF
ELSE
  PRINT “a<>1”
ENDIF
```

```
CHAIN
  ‘ Node 1
  NODE a=1
      PRINT “a=1”
  ELSE
      PRINT “a<>1”
  ‘ Node 2
  NODE b=1
      PRINT “a=1 y b=1”
  ELSE
      PRINT “a=1 y b<>1”
  ‘ Node 3
```

```
    NODE c=1
        PRINT “a=1, b=1 y c=1”
    ELSE
        PRINT “a=1, b=1 y c<>1”
ENDCHAIN
```

El uso de ELSECHAIN hace la función de “trigger” cuando la cadena de condiciones se rompe (alguna condición no se cumplió):

```
CHAIN
    NODE a=1: PRINT “a=1”
    NODE b=1: PRINT “a=1 y b=1”
    NODE c=1: PRINT “a=1, b=1 y c=1”
ELSECHAIN
    PRINT “a<>1 ó b<>1 ó c<>1”
ENDCHAIN
```

La traducción de anidaciones IF-ELSE-IF-THEN sería de la siguiente forma:

```
IF condición 1 THEN
    instrucciones 1
  ELSE
      IF condición 2 THEN
          instrucciones 2
      ELSE
          IF condición 3 THEN
          .
          .
```

Equivale a:

```
CHAIN
  NODE condición 1
    instrucciones 1
  ELSENODE condición 2
    instrucciones 2
  ELSENODE condición 3
    instrucciones 3
ENDCHAIN
```

Llevándolo a un ejemplo particular:

```
IF a = 1 THEN
    PRINT “a=1”
ELSE
      IF b = 1 THEN
          PRINT “a<>1 y b=1”
      ELSE
```

```
            IF c = 1 THEN
               PRINT “a<>1, b<>1 y c=1”
            ENDIF
         ENDIF
      ENDIF

      ‘ Equivalente
      CHAIN
         NODE a=1       : PRINT “a=1”
         ELSENODE b=1 : PRINT “a<>1 y b=1”
         ELSENODE c=1 : PRINT “a<>1, b<>1 y c=1”
      ENDCHAIN
```

El siguiente ejemplo ilustra un uso más real de CHAIN, cuya misión es abrir un fichero de texto y leer una línea determinada, que el usuario indica por su número:

```
‘ Lee una línea seleccionada de un fichero de texto
CHAIN
   NODE TRUE
      FileOpen = FALSE
      INPUT "Número de línea a leer: "; nlinea

   NODE nLinea > 0
   ELSE
      PRINT "Número de línea incorrecto"

   NODE FEXISTS("lista.txt")
      OPEN TEXT READ #1, "lista.txt"
      FileOpen = TRUE
   ELSE
      PRINT "Fichero no existe"

   NODE TRUE
      FOR i = 1 TO nlinea:INPUT #1, linea$:NEXT

   NODE NOT EOF(#1)
      PRINT "Contenido línea: "; linea$
   ELSE
      PRINT "Línea no existe"

ELSECHAIN
   IF FileOpen THEN CLOSE #1
ENDCHAIN
```

Como muestra el ejemplo, este situaciones es bastante frecuente, en donde hay que realizar un conjunto de comprobaciones anidadas para realizar la tarea. En este caso consiste en comprobar que el número de línea introducido sea correcto, el fichero exista, localizar y testear que la línea es correcta, para al final, imprimirla.

Funciones condicionales

Con las funciones IF() e IF$() se podrán obtener valores distintos en función del cumplimiento de una condición.

Esto es muy útil en aquellas ocasiones en que sea necesario alterar el valor de una variable en función de una condición, lo cual puede hacerse de forma sencilla con estas funciones.

El siguiente ejemplo muestra una situación en donde la variable "A" es incrementando en función del valor de "B":

```
A = 10
IF B > 10 THEN INC A, 2 ELSE INC A, 4
```

Esto mismo se puede hacer con la función IF() :

```
A = 10 + IF(B > 10, 2, 4)
```

Resultando un código de menor tamaño e igualmente entendible.

Del ejemplo anterior se deduce la sintaxis de la función IF():

IF(*expresión condición , expresión caso verdadero , expresión caso falso*)

Por su parte, la función IF$() es análoga a IF() pero para el caso de cadenas:

```
INPUT "Nombre: "; A$
INPUT "Apellidos: "; B$
IF B$ <> "" THEN A$ = A$ + " " + B$
```

Usando IF$ quedaría:

```
INPUT "Nombre: "; A$
INPUT "Apellidos: "; B$
A$ = A$ + IF$(B$ <> "", " " + B$, "")
```

Bucles

Los bucles son estructuras de programa que ejecutan repetidamente un conjunto de instrucciones en función del cumplimiento de una condición.

iBASIC dispone de 3 tipos de bucle:

```
FOR-NEXT
WHILE-WEND
REPEAT-UNTIL
```

También se pueden hacer bucles manuales por medio de la orden GOTO.

Bucle FOR-NEXT:

Su sintaxis general es:

FOR *variable índice=valor inicial* TO *valor final* STEP *incremento*
.
.
NEXT *variable índice*

Un bucle FOR-NEXT está dirigido por una variable que actúa de índice del bucle, y que inicialmente toma un valor inicial indicado.

El bucle se repetirá hasta que la variable (índice) alcance el valor final.

En cada iteración, la variable es incrementada automáticamente con el valor indicado en STEP.

La cláusula STEP es opcional, y en caso de omitirla se asume un incremento de 1.

El bucle acaba con la orden NEXT, que opcionalmente puede ir acompañada con la variable índice del bucle.

La variable índice del bucle es una variable normal, que puede ser usada y modificada dentro del bucle.

Para bucles anidados, se puede utilizar una única orden NEXT para finalizarlos a todos, indicando sus índices en orden inverso al de aparición.

Ejemplos:

```
‘ Imprime los números del 1 al 5
FOR i = 1 TO 5
    PRINT i
NEXT
‘ Imprime los números del 5 al 1
FOR i = 5 TO 1 STEP -1 : PRINT i : NEXT
‘ Imprime los números del 1 al 3
FOR i = 1 TO 5
    PRINT i
    IF i = 3 THEN i = 5
NEXT
‘ Imprime los números pares del 0 al 10
limit = 10
FOR i = 0 TO limit STEP 2 : PRINT I : NEXT
‘ Imprime los elementos de la matriz rectangular “a”
FOR i=1 TO DBOUND(a, 1)
FOR j=1 TO DBOUND(a,2)
PRINT “Elemento”; i; ”; ”j”; a(i,j)
NEXT j,i
```

Bucle WHILE-WEND:

WHILE *expresión condición*
.

```
.
WEND
```

Este bucle se ejecutará mientras la condición indicada en WHILE sea cierta (se cumpla).

Al estar la condición en el inicio del bucle, puede ocurrir que no se ejecuten sus instrucciones internas si dicha condición no se cumple la primera vez.

Ejemplos:

```
‘ Imprime los números del 1 al 5
i = 1
WHILE i <= 5
    PRINT i
    INC i
WEND
```

Bucle REPEAT-UNTIL:

```
REPEAT
.
.
UNTIL expresión condición
```

Este bucle se ejecutará hasta que la condición indicada en UNTIL sea cierta (se cumpla).

Al estar la condición la final del bucle, siempre se ejecutarán sus instrucciones, al menos, una vez.

Ejemplos:

```
‘ Imprime los números del 1 al 5
i = 1
REPEAT
    PRINT i
    INC i
UNTIL i > 5
```

Instrucciones BREAK y CONTINUE:

Estas dos instrucciones pueden ser utilizadas en cualquier tipo de bucle y permiten cambiar su comportamiento normal.

BREAK cancela la ejecución del bucle, finalizándolo, y continuando la ejecución en la siguiente instrucción presente después del mismo.

CONTINUE provoca que se inicie una nueva iteración del bucle, siempre y cuando la condición de éste se cumpla.

Ejemplos:

```
‘ Imprime los números del 1 al 5
FOR i = 1 TO 10
    PRINT i
    IF i = 5 THEN BREAK
NEXT
‘ Imprime los números del 1 al 10 excepto el 5
FOR i = 1 TO 10
    IF i = 5 THEN CONTINUE
    PRINT i
NEXT
```

Entrada y salida a pantalla

iBASIC está basado en texto y dispone de diversas instrucciones de entrada y salida a pantalla.

Las órdenes tradicionales más comunes son INPUT para la entrada, y PRINT para la salida.

Además de éstas dos, se disponen de otras para manejo del color y realizar desplazamientos, por ejemplo.

Instrucción	**Descripción**
INPUT	Esperar a entrada de texto del usuario
PRINT	Imprimir texto en pantalla
CLS	Borrar pantalla
LOCATE	Posicionar cursor
COLOR	Establecer color de primer plano/fondo
CURSOR	Cambiar tamaño del cursor
SCREEN FULL MODE	Cambiar a modo de pantalla completa
SCREEN WINDOW MODE	Cambiar a modo de ventana
SCROLL	Realiza desplazamiento de región de pantalla

Función	**Descripción**
INKEY$	Devuelve cadena con tecla pulsada
SCRCHR$	Devuelve cadena con los caracteres presentes en una zona de la pantalla
XSIZE	Anchura de la pantalla de iBASIC en caracteres (X máximo)
YSIZE	Altura de la pantalla de iBASIC en caracteres (Y máximo)
XSIZE(#0)	Anchura de la pantalla física (pixels)
YSIZE(#0)	Altura de la pantalla física (pixels)
FORECOLOR	Color actual de primer plano
BACKCOLOR	Color actual de fondo
XPOS	Posición X actual del cursor
YPOS	Posición Y actual del cursor

Instrucción INPUT

Con esta instrucción el programa se detiene a la espera de que el usuario introduzca un dato por teclado, que será almacenado en la variable que se indique en INPUT.

La sintaxis básica de INPUT es:

INPUT *variable*

La variable puede ser numérica o de cadena. En caso de ser numérica, sólo se aceptarán valores correctos por parte del usuario.

Al ejecutarse INPUT de esta forma, se muestra en pantalla un signo de interrogación (?) para indicar que se está a la espera de una entrada.

Cuando el usuario introduzca el dato y pulse ENTER, éste será almacenado en la variable y el programa proseguirá.

INPUT también admite otras sintaxis:

INPUT *cadena; variable*
INPUT *cadena, variable*

En ambos casos, se visualizará en pantalla la cadena indicada, que hace la función de texto descriptivo de la entrada.

La diferencia entre estas dos sintaxis es el uso del signo ";" y ",".

En caso de usar ";", se mostrará el interrogante después de la cadena. Por el contrario, si se emplea ",", tal signo no se visualizará.

Como opción, a estas dos sintaxis se le puede añadir la clausula INJECT:

INPUT INJECT *cadena; variable*
INPUT INJECT *cadena, variable*

Su comportamiento es igual al anterior, excepto en que el contenido que tenga la variable es "inyectado" a la entrada, de manera que se mostrará como texto introducido por defecto, el cual puede ser editado o borrado.

Ejemplos:

```
INPUT "Introduce un numero: ", num
INPUT "Cual es tu edad"; edad

borrar$="n"
INPUT INJECT "Borrar el fichero"; borrar$
```

Instrucción PRINT

PRINT es la instrucción para imprimir o visualizar información por pantalla, ya sean números o cadenas, realizándolo en la posición actual del cursor.

La sintaxis general es la siguiente:

PRINT *dato;dato;dato; ….*

Siendo "*dato*" un valor literal, expresión o variable, ya sean de número o de cadena.

Todos los datos presentes en una orden PRINT se imprimirán en la misma línea, uno a continuación de otro.

Por defecto, PRINT, después de imprimir todos los datos, saltará a la siguiente línea de pantalla, posicionando allí el cursor.

Sin embargo, si se añade un carácter ";", PRINT no saltará de línea, por lo que un PRINT siguiente imprimirá a continuación, en la misma línea.

Además de la utilización del carácter ";" se puede utilizar "," como separador entre datos a imprimir. La diferencia entre ambos es que ";" imprime los dos datos juntos, sin separación, al contrario que ",", que los imprime con una separación de un tabulador.

PRINT realiza la escritura en la posición actual del cursor, y lo avanza consecuentemente. En cualquier caso, el cursor puede reposicionarse manualmente usando la orden LOCATE.

Ejemplos:

num = 100
PRINT num → escribe 100
PRINT num, 200, "texto" → escribe 100 200 texto
PRINT num; 200; "texto" → escribe 100 200 texto
PRINT "texto";
PRINT "2" → escribe texto2

PRINT también admite una segunda sintaxis, usada para escribir números con un formato dado:

PRINT USING *cadena formato; expresión numérica …*

Esta forma permite imprimir números de una anchura fija, rellenados de ceros, o con separación de millares, por ejemplo.

Ver el manual de referencia para más detalles.

Ejemplos:

PRINT USING "###"; 12 → imprime 12
PRINT USING "000"; 12 → imprime 012
PRINT USING "###.0#"; 12 → imprime 12.0

Instrucción LOCATE

Con LOCATE se cambia la posición del cursor a unas coordenadas dadas.

LOCATE se usará conjuntamente con PRINT o INPUT.

La sintaxis de LOCATE es sencilla y únicamente consiste en indicar la nueva posición (fila, columna):

LOCATE fila, columna

Las coordenadas de "fila" y "columna" comienzan en valor 1.

La posición (1, 1) se refiere a la esquina superior izquierda de la pantalla.

Ejemplo:

LOCATE YSIZE / 2, 1 : PRINT "En medio"

Este ejemplo imprime el texto en la columna 1 de la fila central de la pantalla.

Instrucción CLS

CLS borra la pantalla y situa el cursor en la coordenada 1, 1.

Instrucción COLOR

Con COLOR se podrá cambiar el color actual de primer plano (color del cuerpo de las letras) y el del fondo, de forma separada o conjuntamente.

La sintaxis general es:

COLOR *color primer plano, color fondo*

Se puede especificar ambos colores, o solo uno.

El código de color es un número entre 0 a 15. Ver anexo 2 del manual de referencia para lista de colores.

Ejemplos:

COLOR 4 → color de primer plano a rojo
COLOR ,2 → color de fondo a verde
COLOR 12,0 → color de primer plano a rojo brillante y el fondo a negro

Instrucción CURSOR

CURSOR permite cambiar el tamaño del cursor, o bien, hacerlo invisible.

Su sintaxis es simple:

CURSOR *factor relleno*

"factor relleno" es un número entre 0 y 1 que indica el nivel de relleno del cuerpo del cursor.

El valor de 1 indica un cursor compacto, 100% relleno.

El valor de 0 hace que el cursor sea invisible.

Función INKEY$

INKEY$ es una función que devuelve una cadena representando a la tecla que se ha pulsado.

En caso de no haber ninguna tecla pulsada, devuelve cadena vacia.

A diferencia de INPUT, INKEY$ no se queda a la espera, sino que únicamente consulta la existencia de alguna tecla, y en caso de haberla, la extrae y devuelve como cadena.

La cadena devuelta puede tener 0, 1 ó 2 caracteres de longitud.

Devuelve 0, si no hay tecla pulsada.

Devuelve 1, si hay tecla pulsada y ésta corresponde a un carácter imprimible.

Devuelve 2, si hay tecla pulsada y corresponde a una tecla especial (de función, movimiento, etc.). Ver anexo 3 del manual de referencia para la tabla de códigos.

Ejemplos:

```
REM Esperar la pulsación de una tecla
WHILE NOT EMPTY(INKEY$):WEND
```

Instrucciones SCREEN …

Las órdenes SCREEN FULL MODE y SCREEN WINDOW MODE establecen el modo de pantalla de ejecución de iBASIC, esto es, a pantalla completa, o bien, en modo ventana.

Sus sintaxis son:

```
SCREEN FULL MODE
SCREEN WINDOW MODE número filas, número columnas
```

Para el caso de SCREEN WINDOW MODE, hay que indicar el tamaño (en caracteres) de la ventana.

Instrucción SCROLL

Esta instrucción mueve una zona de la pantalla a otra:

```
SCROLL fila1, columna1, fila2, columna2 TO filadest, columnadest
```

Los cuatros primeros argumentos definen la zona a mover, y los dos últimos las coordenadas de la nueva posición (esquina superior izquierda).

La región de la pantalla que se quede vacia, por el movimiento, se rellenará con espacios usando el color actual.

Ejemplo:

```
SCROLL  10, 1, 15, 20 TO 9, 1 → realiza un desplazamiento vertical hacia arriba
```

Subrutinas

Una subrutina es un fragmento de código (un conjunto de líneas) que puede ser invocado desde cualquier otra parte, y que, una vez acabado su trabajo, devuelve el control, ejecutándose la siguiente instrucción al punto que fue llamada.

Las instrucciones que se usan son GOSUB y RETURN.

GOSUB es la encargada de llamar a una subrutina dada, que se especifica con un número de línea, o por una etiqueta. Así pues, GOSUB será utilizado por el llamador de la subrutina.

RETURN se usa dentro de la subrutina, finalizándola y retornando al punto en donde fue invocada con GOSUB.

De lo anterior se deduce que, toda subrutina ha de ser llamada con GOSUB y debe finalizar con RETURN.

Una subrutina es un concepto distinto al de "procedimiento". La subrutina se ejecuta dentro del ámbito actual, y no permite parámetros, mientras que un procedimiento tiene su propio ámbito y admite parámetros.

Como extensión, un procedimiento puede contener subrutinas dentro de él.

Ejemplo:

```
10 GOSUB 100
20 PRINT "fin del programa"
30 END
.
.
100 PRINT "aquí empieza la subrutina"
110 RETURN
```

La versión del anterior ejemplo con etiquetas sería:

```
GOSUB subrutina1
PRINT "fin del programa"
END
.
.
subrutina1:
PRINT "aquí empieza la subrutina"
RETURN
```

En ambos casos, se imprimiría lo siguiente:

aquí empieza la subrutina
fin del programa

Pilas de usuario

iBASIC dispone de 32 pilas (stacks) de libre uso por los programas, que se numeran de 0 a 31, considerándose como pila principal la cero.

En estas pilas se pueden almacenar tanto número como cadenas, y se usarán, principalmente, para salvaguardar y recuperar variables.

Hay tres instrucciones y una función de pila:

Instrucción	**Descripción**
PUSH	Insertar dato en la cima de la pila
POP	Recuperar y borrar dato de la cima de la pila
CLEAR	Vaciar pila

Función	**Descripción**
EMPTY	Devuelve TRUE si la pila está vacia

Un ejemplo de utilización es la salvaguarda de variables que se vayan a utilizar en las subrutinas, de forma que al inicio de una se guarden las variables que ésta utiliza (PUSH), y antes del RETURN se recuperen (POP).

Insertar datos en pila (PUSH)

PUSH permite insertar datos literales, expresiones o contenidos de variables:

PUSH *dato*
PUSH *dato*, *dato, …*

PUSH #*número pila*, *dato*
PUSH #*número pila*, *dato*, *dato, …*

Extraer datos de pila (POP)

Por su parte, POP puede usarse sin argumentos, de forma que directamente borra el dato de la pila sin recuperar su valor, o bien, puede utilizarse indicando una o más variables, en las cuales se almacenarán los valores recuperados:

POP
POP *variable*
POP *variable*, *variable, …*

POP #*número pila*
POP #*número pila, variable*
POP #*número pila, variable*, *variable, …*

La especificación del número de pila (de 0 a 31), en PUSH y POP, es opcional, y en caso de no hacerlo se asume la realización de la operación sobre la pila cero (pila principal).

Vaciado de pila (CLEAR)

Cualquiera de las pilas se puede vaciar usando la orden CLEAR:

CLEAR #*número pila*

Para vaciar la pila principal hay que indicar como número el cero, CLEAR #0

Consulta estado de pila

Con la función EMPTY se podrá consultar si una pila está vacia o llena:

EMPTY(#*número pila*)

Está función devuelve TRUE si la pila está vacia.

Para consultar la pila principal hay que indicar como número el cero, EMPTY(#0)

Ejemplos

El siguiente ejemplo muestra un programa que llama a una subrutina, la cual salvaguarda el contenido de la variable “i”, que es utilizada dentro de la subrutina, y que, en el ejemplo, lo es del llamador (bucle FOR-NEXT):

```
randomize
for i = 1 to 10
   print "Número aleatorio";i;":";
   gosub print_random
next
end
print_random:
   push i
   i = 100 + round(rnd * 100)
   print i
   pop i
   return
```

Otro uso habitual de una pila es ser utilizada como mecanismo de paso de parámetros a subrutinas, de manera que el llamador deja en pila los valores de los parámetros, y la subrutina, al inicio, los recupera.

El ejemplo siguiente retoma el anterior, pero ahora la subrutina necesita dos parámetros por pila para indicar el intervalo de números aleatorios a generar. En este caso, se utiliza la pila principal (0) para el paso de parámetros, y la pila 1 para salvaguarda de variables:

```
randomize
for i = 1 to 10
```

```
    print "Número aleatorio";i;"(de";i*10;"a";i*10+9;"):";
    push i*10
    push i*10 + 9
    gosub print_random
  next
  end
  print_random:
    push #1, hasta, desde, i
    pop hasta
    pop desde
    i = desde + round(rnd * (hasta-desde))
    print i
    pop #1, i, desde, hasta
    return
```

A continuación un ejemplo que extrae todos los elementos de la pila 1, y los imprime en pantalla:

```
while not empty(#1)
  pop  #1, num
  print num
wend
```

Control de errores

Cuando un programa se ejecuta éste puede generar errores debidos a diversas causas, pero principalmente son por una incorrecta sintaxis en el programa en sí mismo, índices de matrices que se salen de rango, números de línea que no existen, etc.

En caso de que un error se produzca, el programa se detendrá mostrando dicho error y la línea en que se produjo.

Por ejemplo, el siguiente programa generará un error en la línea 20:

```
10 dim a(5)
20 a(6)=10
30 print a(6)
```

Al ejecutarse con RUN aparecerá el siguiente error:

Error 72 en línea 20: Especificación inválida de dimensiones

Como puede comprobarse, también la línea 30 es errónea, pero iBASIC solo se detiene en la primera línea que causa error.

Si la línea 20 se modificara a:

```
20 a(5)=10
```

Ya sería válida, pero una nueva ejecución daría el error en la línea 30:

Error 72 en línea 30: Especificación inválida de dimensiones

Este tipo de errores no son detectados por CHECK SYNTAX, debido a que no son errores sintácticos del lenguaje, sino que son errores dinámicos de ejecución.

Por ejemplo, si el programa fuera:

```
10 dim a(5)
20 a(5)=
30 print a(5)
```

El uso de CHECK SYNTAX detectaría el error de la línea 20:

Error 30 en línea 20: Orden incompleta

Sin embargo los errores en tiempo de ejecución se pueden controlar, de manera que el programa no sea detenido.

Este control de errores (en ejecución) se configura usando la orden ON ERROR, que admite diversas sintaxis en función de la acción que se desea realizar en caso de producirse un error:

ON ERROR CONTINUE: en caso de error, la ejecución continuará en la siguiente orden.
ON ERROR GOSUB *numero línea*: en caso de error, se realiza una llamada a la subrutina indicada. El retorno de la subrutina devolverá el control a la siguiente orden que causó el problema.
ON ERROR GOTO *numero línea*: en caso de error, se realiza un salto a la línea (o etiqueta) indicada.
ON ERROR STOP: en caso de error, se detiene la ejecución del programa.

ON ERROR puede aparecer cualquier número de veces en el programa, pero usualmente sólo habrá una al principio de éste.

Volviendo al primer ejemplo, se puede modificar el programa de la siguiente forma:

```
5 on error goto 100
10 dim a(5)
20 a(6)=10
30 print a(6)
.
.
100 print “Se ha producido un error en el programa”.
110 print “Pulse una tecla para acabar”
120 while empty(inkey$):wend
130 cls
140 end
```

Al ejecutarse el programa, de nuevo se producirá el error en la línea 20, pero ahora en lugar de finalizar directamente el programa (mostrando el error), se saltará a la línea 100, como se indica en ON ERROR GOTO, y se realizarán las instrucciones que allí se encuentren, en este caso, mostrar un aviso de error más "amigable" y salir del programa.

Habitualmente en las rutinas de tratamiento de errores de ON ERROR, se pueden realizar distintas acciones en función del error que haya habido.

Hay una serie de variables internas de iBASIC que contienen información sobre el último error producido:

Variable interna	Descripción
ERL	Número de línea del error
ERP$	Nombre del subprograma que generó el error
ERR	Código del error
ERR$	Cadena descriptiva del error

Estas variables pueden usarse en cualquier parte del programa, como si se variables se trataran. En caso de haber error, devuelven 0 o cadena vacia, según el caso.

En caso de programas sin números de línea, ERL devuelve el número de línea físico.

Volviendo al ejemplo:

```
5 on error goto 100
10 dim a(5)
20 a(6)=10
30 print a(6)
.
.
100 print "Se ha producido un error de tipo";ERR;"en el programa".
110 print "Pulse una tecla para acabar"
120 while empty(inkey$):wend
130 cls
140 end
```

Manejo de ficheros

iBASIC permite manejar ficheros de texto, binarios, multimedia y de red:

- Texto: acceso lectura/escritura en ficheros organizados en líneas de caracteres imprimibles.
- Binarios raw: acceso lectura/escritura aleatoria a ficheros binarios sin estructurar.
- Binarios estructurados: acceso lectura/escritura de ficheros binarios estructurados en registros de longitud fija.

- Multimedia: reproducción de ficheros de audio/video.
- Red: comunicación en modo cliente usando protocolos basados en UDP o TCP.

Con independencia de su tipo, todo fichero debe ser "abierto" para poder ser utilizando, usando la orden OPEN. Es con en orden donde se indica el tipo del fichero, y además, se especifica un número para identificarlo, el cuál será usando posteriormente para realizar operaciones sobre él.

Simultáneamente se pueden abrir varios ficheros, pero sus números de identificación deberán ser distintos.

Una vez finalizado el trabajo con un fichero, éste deberá ser "cerrado" con la orden CLOSE.

Las instrucciones de manejo son diferentes en función del tipo de fichero.

Los ficheros son globales al programa, siendo accesibles desde cualquier procedimiento. Así pues, un procedimiento puede abrir un fichero, y otro distinto realizar operaciones sobre él.

Ficheros de texto

Este tipo de ficheros son aquellos que están estructurados en líneas de texto, finalizadas con un carácter CR ó CR + LF.

Un fichero de este tipo puede ser abierto en modo lectura o escritura, pero no para leer y escribir a la vez.

Apertura

La instrucción OPEN para la apertura de ficheros de texto tiene las siguientes sintaxis:

OPEN TEXT APPEND #*número fichero*, *cadena nombre fichero*
OPEN TEXT CREATE #*número fichero*, *cadena nombre fichero*
OPEN TEXT READ #*número fichero*, *cadena nombre fichero*

La apertura puede hacerse de tres modos:

APPEND: se abre el fichero, en modo escritura, para añadir textos al final del mismo.

CREATE: se abre, en modo escritura, pero creándose el fichero. En caso de existir, éste es truncado, y toda su información se pierde.

READ: se abre el fichero en modo lectura, posicionándose el puntero de lectura al principio del mismo. El fichero debe existir.

El "número de fichero" es el identificador de éste para la realización posterior de operaciones sobre él.

Puede usarse la función "FEXISTS" para determinar si el fichero existe.

Operaciones

La siguiente tabla muestra la lista de instrucciones y funciones de operación para ficheros de texto:

Instrucción	Descripción
INPUT	Leer línea del fichero
PRINT	Escribir línea en fichero

Función	Descripción
EOF	Comprobar fin de fichero

En cualquiera de las anteriores hay que indicar el número de fichero sobre el cual se quiere realizar la operación.

La función EOF, que devuelve 1 si se alcanzó el fin de fichero, se usará en operaciones de lectura y determinar si hay más datos para leer.

EOF debe usarse después de una orden INPUT, ya que es un intento de lectura (mediante INPUT) quién realmente determina el fin de fichero.

INPUT y PRINT tienen la misma sintaxis que sus equivalentes para pantalla, excepto en que hay que indicar el número de fichero:

INPUT #*número fichero, …*
PRINT #*número fichero, …*

El programa siguiente realiza una lectura completa de un fichero y lo muestra en pantalla:

```
10 input "Fichero a leer: ",file$
20 open text read #1, file$
30 while not eof(#1)
40   input #1, line$
50   if not eof(#1) then print line$
60 wend
70 close #1
```

Ficheros binarios raw (no estructurados)

Este tipo de ficheros son aquellos a los que se puede acceder a cualquier bytes o región del mismo de forma directa: posicionar y leer/grabar.

Apertura

En la instrucción OPEN solamente hay que indicar el modo binario, nombre de fichero y su número:

OPEN BINARY #*número fichero*, *cadena nombre fichero*

De esta forma se abre el fichero en modo lectura/escritura, creándolo en caso de no existir.

Para abrirlo de forma "solo lectura", se incluye la clausula READ:

OPEN BINARY READ #*número fichero*, *cadena nombre fichero*

Operaciones

Las operaciones y funciones disponibles son las siguientes:

Instrucción	Descripción
READ	Leer bytes
WRITE	Escribir bytes
GOTO	Posicionar puntero del fichero

Función	Descripción
EOF	Comprobar si se está al final del fichero
CREC	Obtener posición actual dentro del fichero
NREC	Obtener número total de bytes del fichero (tamaño)

Asociado al fichero se mantiene un "puntero" que indica la posición actual de lectura/escritura dentro del fichero.

Este puntero comienza en 1 (principio de fichero) y puede llegar hasta NREC + 1.

Las operaciones READ y WRITE se realizan a partir de este puntero, y al ser realizadas, el puntero se incrementa adecuadamente:

OPEN #1 BINARY "datos.dat"
PRINT #1, REC(#1) → imprime 1
READ #1, A$, 5 → lee los primeros 5 bytes
READ #1, B$, 10 → lee los siguientes 10 bytes
WRITE #1, A$ → escribe los 5 bytes de A$ en las posiciones 11-12-13-14-15

El valor del puntero actual se puede obtener con la función CREC

Cuando el fichero se abre, éste se posiciona sobre el primer byte del mismo (posición 1).

El puntero se puede mover manualmente usando la orden GOTO:

GOTO #1, 100 → ir al byte número 100

Operación READ:

Se lee un número máximo de bytes desde la posición actual:

READ #*número fichero*, *variable cadena, longitud*

Los bytes leídos son guardados en la variable (de cadena) indicada, y se leerán un máximo de "longitud" bytes.

En situación normal, después de la lectura, la longitud de la cadena leída será igual a la "longitud" especificada en READ, sin embargo si se alcanza el fin de fichero dicha longitud será menor, o incluso cero.

En cualquier caso, la longitud leída se puede obtener usando la función LEN sobre la variable destino de lectura.

Operación WRITE:

Escribe el contenido de una variable de cadena en el fichero a partir de la posición actual:

WRITE #*número fichero*, *cadena*

Para añadir al final del fichero, primero hay que mover el puntero 1 byte más allá del final, y luego escribir:

```
GOTO #1, NREC(#1) + 1
WRITE #1, A$
```

Operación GOTO:

Posiciona el puntero de lectura/escritura:

GOTO #*número fichero, posición*

El rango de posiciones del puntero va desde 1 (principio de fichero) hasta NREC+1 (fin de fichero).

Ejemplo:

```
10 ' Copiar un fichero en bloques de 4Kb
20 input "Fichero origen : ", sfile$
30 input "Fichero destino: ", dfile$
40 ' Apertura
50 open binary #1, sfile$
60 open binary #2, dfile$
70 ' Bucle lectura/escritura
80 print "Copiando";nrec(#1);"bytes..."
90 while not eof(#1)
100 read #1, a$, 4096
110 write #2, a$
120 wend
130 close
140 print
150 print "Copia terminada"
160 end
```

Ficheros binarios estructurados

Este tipo de ficheros son aquellos que están organizados en registros de longitud fija, estando compuesto cada uno de ellos por un conjunto de campos de texto o numéricos, también de longitud fija.

El acceso a los registros es directo (también llamado aleatorio) a un número de registro, comenzando por el número uno.

Apertura

En la instrucción OPEN, además de indicarse el nombre de fichero y su número, se especifica la estructura de campos del registro:

OPEN BINARY #*número fichero*, *cadena nombre fichero*, FIELDS *especificación campos*

De esta forma se abre el fichero en modo lectura/escritura, creándolo en caso de no existir.

Para abrirlo de forma "solo lectura", se incluye la clausula READ:

OPEN BINARY READ #*número fichero*, …

La especificación de campos define la estructura de cada registro del fichero, y es una secuencia de uno o más campos, separados por comas, en donde se indica el nombre, tipo y longitud de cada uno ellos.

El formato de la especificación de campos es el siguiente:

nombre campo = *tipo campo* (*tamaño*), *nombre campo = …*

siendo:

nombre campo: variable o nombre del campo del registro, que es tratado como si de una variable del programa se tratase, pero a nivel local del fichero. Los campos de cadena, o de buffer, tendrán el carácter final "$" .

tipo campo: indica el tipo de datos y la interpretación que debe darse al campo:

INT = número entero con signo
UINT = número entero sin signo
FLOAT = número de punto flotante
STRING$ = cadena de caracteres o buffer de bytes

tamaño: longitud o tamaño del campo, en bytes. Dependiendo del tipo de campo, se permiten distintos tamaños:

INT = 1, 2 ó 4 bytes (enteros de 8, 16 ó 32 bits)
UINT = 1, 2 ó 4 bytes
FLOAT = 4 ó 8 bytes (números de 32 ó 64 bits)
STRING$ = de 1 a 2^{32} bytes

Los números enteros son almacenados según el modo actualmente configurado en el programa, pudiendo ser "Little endian" o "Big endian".

Por defecto se usa “Little endian”, pero puede ser alterado usando las órdenes:

SET ENDIAN LITTLE: cambiar a formato “Little endian”
SET ENDIAN BIG: cambiar a formato “Big endian”

Operaciones

La siguiente tabla muestra la lista de instrucciones y funciones de operación para ficheros binarios estructurados:

Instrucción	**Descripción**
GET	Leer registro
PUT	Escribir registro

Función	**Descripción**
FIELD FIELD$	Acceso a campo de registro
EOF	Comprobar fin de fichero
CREC	Obtener número de registro actual
NREC	Obtener número total de registro
SREC	Obtener tamaño de un registro (bytes)

En cualquiera de las anteriores hay que indicar el número de fichero sobre el cual se quiere realizar la operación.

Las instrucciones básicas son GET y PUT, y su sintaxis es idéntica:

GET #*numero fichero*
GET #*numero fichero, número registro*

PUT #*numero fichero*
PUT #*numero fichero, número registro*

Donde “número registro” es el número de registro (comenzando en 1) a leer o escribir. En caso de no indicar tal número, el registro es el siguiente a la de última operación realizada.

En estas operaciones, los valores de los campos del registro se almacenan en las “variables” especificadas en la clausula FIELDS de OPEN.

Estas variables son locales al fichero, pudiendo existir las mismas (con igual nombre) en distintos archivos.

Usando las funciones FIELD y FIELD$ se accede o se asigna el valor de un campo:

FIELD(#*número fichero, nombre campo numérico*)
FIELD$(#*número fichero, nombre campo cadena*)

Se usará FIELD para campos numéricos y FIELD$ para los alfanuméricos (cadena).

Opcionalmente, se disponen de un conjunto de funciones de conversión entre números a cadenas (representación en memoria):

Función	Descripción
CVB	Convertir cadena a número entero de 8 bits
CVI	Convertir cadena a número entero de 16 bits
CVL	Convertir cadena a número entero de 32 bits
CVS	Convertir cadena a número flotante de 32 bits
CVD	Convertir cadena a número flotante de 64 bits
MKB$	Convertir número entero de 8 bits a cadena
MKI$	Convertir número entero de 16 bits a cadena
MKL$	Convertir número entero de 32 bits a cadena
MKS$	Convertir número flotante de 32 bits a cadena
MKD$	Convertir número flotante de 64 bits a cadena

Las funciones MK* devuelven una cadena con la representación en memoria del número indicado, mientras que CV* hacen lo contrario. El formato de la representación en memoria será el actualmente configurado (little/big endian).

Sin embargo, éstas no se utilizarán en este tipo de ficheros (aunque puede hacerse), sino que están pensadas para uso en ficheros de red.

Operación GET

Típicamente el procedimiento para leer un registro será:

GET #*número fichero, número registro*
' Acceso a campos con: FIELD(#*número fichero, nombre campo i*)

Una vez hecha la lectura se tendrá disponible su conteniendo en los campos, accediendo a ellos por medio de FIELD.

Ejemplo:

OPEN BINARY #1, "datos.dat", FIELDS nombre$=STRING$(20), num=INT(4)
GET #1, 1 → leer el primer registro
PRINT FIELD$(#1, nombre$), FIELD(#1, num) → imprime contenido

Operación PUT

De forma análoga, el procedimiento para escribir un registro será:

FIELD(#*número fichero, nombre campo 1*) *= valor campo 1*
FIELD(#*número fichero, nombre campo 2*) *= valor campo 2*
.
.
FIELD(#*número fichero, nombre campo N*) *= valor campo N*
PUT #*número fichero, número registro*

Esto es, de forma previa a realizar el PUT hay que asignar los valores a los campos del registro usando FIELD:

```
OPEN BINARY #1, “datos.dat”, FIELDS nombre$=STRING$(20), num=INT(4)
FIELD$(#1, nombre$) = “Jose del Rio”
FIELD(#1, num) = 100
PUT #1,2 → graba registro 2
```

Una versión alternativa del ejemplo anterior, usando funciones CV* y MK* sería:

```
OPEN BINARY #1, “datos.dat”, FIELDS nombre$=STRING$(20), num$=STRING$(4)
FIELD$(#1, nombre$) = “Jose del Rio”
FIELD$(#1, num$) = MKL$(100)
PUT #1,2
‘ Leer registro 1
GET #1, 1
PRINT FIELD$(#1, nombre$), CVL(FIELD$(#1, num$))
```

Ficheros de red

Bajo este concepto se abstrae la comunicación que puede realizarse en un entorno de red usando protocolos UDP o TCP.

El cualquier caso, el programa BASIC actuará en modo cliente, pudiendo escribir o leer simultáneamente, siendo el procedimiento de una comunicación el siguiente:

- Abrir “fichero de red” (orden OPEN)
- Realizar operaciones de red (órdenes READ, WRITE, INPUT, PRINT)
- Cerrar “fichero de red” (orden CLOSE)

Usualmente las comunicaciones de red utilizan tramas o paquetes con una estructura interna que puede ser variable.

Para iBASIC, estas tramas pueden ser vistas como una cadena de caracteres, pudiéndose utilizar las funciones de tratamiento de éstas, y las funciones específicas CV* y MK* de conversión numérica.

De forma estándar, los números enteros son transmitidos en formato “big endian” (red), por lo que deberá utilizarse la orden SET ENDIAN BIG para ello.

<u>Comunicación UDP</u>

El procedimiento a seguir para realizar una comunicación UDP será:

- Abrir “fichero de red UDP”: orden OPEN
- Realizar operaciones: órdenes READ, WRITE
- Crear fichero de red: orden CLOSE

La orden OPEN tiene la siguiente sintaxis:

OPEN NETWORK UDP(*puerto) #número fichero*, *cadena host*

Esta orden prepara un medio para enviar y recibir tramas al host y puerto indicados.

Las operaciones y funciones disponibles son las siguientes:

Instrucción	**Descripción**
READ	Leer bytes
WRITE	Escribir bytes

Función	**Descripción**
EOF	Comprobar si hay bytes disponibles para leer
CREC	Obtener número de bytes leídos
NREC	Obtener número de bytes total recibidos
SREC	Obtener número de bytes pendientes por leer (recibidos pero no leídos)

Las instrucciones READ y WRITE están pensadas para leer un número de bytes, que típicamente será una trama, cuyo tamaño puede ser fijo y conocido o no serlo.

Es por ello que será necesario utilizar la función SREC, que devuelve el número de bytes recibidos pero que aún no se han leído con READ.

Así pues, normalmente un programa que use una comunicación UDP, enviará una trama al servidor (WRITE) y esperará una respuesta, dicha respuesta será una trama, que cuando se reciba, y ante el uso de SREC, se obtendrá su tamaño. De esta forma, bastará con sondear el valor de SREC para determinar si han llegado datos.

Otra forma, es utilizar EOF, en lugar de SREC. EOF devuelve 0 (FALSE) si hay datos a la espera de ser leídos (SREC > 0), y TRUE si no los hay.

READ admite dos sintaxis:

READ #*número fichero*, *variable cadena, longitud*
READ #*número fichero*, *variable cadena, longitud, timeout*

La única diferencia está en el parámetro opcional “timeout” que indica el tiempo máximo de espera (en milisegundos) para recibir el número de bytes especificado en longitud.

Los bytes recibidos serán almacenados en la variable de cadena indicada.

En caso de expirar el timeout, una llamada a la función EOF devolvería TRUE, y la variable de cadena estaría vacia.

Por su parte, la instrucción WRITE tiene la siguiente forma:

WRITE #*número fichero*, *cadena*

Que directamente envía el contenido de la cadena (típicamente una trama) a la conexión asociada con el número de fichero indicado.

El siguiente ejemplo ilustra una comunicación en donde el programa envía una solicitud (un byte con valor 1), espera la respuesta (una trama de 4 enteros de 32 bits) y los imprime en pantalla:

```
SET ENDIAN BIG
OPEN NETWORK UDP(1001) #1, “192.168.1.101”
‘ Envia solicitud
WRITE #1, MKI$(1)
‘ Leer respuesta: 4 enteros de 32 bits (timeout 1.5 segundos)
READ #1, trama$, 4 * 4, 1500
IF EMPTY(trama$) THEN PRINT “Timeout!”:CLOSE #1:END
num1 = CVL(trama$[1..4])
num2 = CVL(trama$[5..8])
num3 = CVL(trama$[9..12])
num4 = CVL(trama$[13..16])
PRINT “Datos recibidos:”;num1;num2;num3;num4
CLOSE #1
```

Comunicación TCP

Desde el punto de vista de un programa BASIC, una comunicación TCP es similar a una UDP, excepto en que permite un par de instrucciones más para dar soporte a protocolos de aplicación basados en texto.

Primeramente, la orden OPEN tiene la siguiente sintaxis:

OPEN NETWORK TCP(*puerto) #número fichero*, *cadena host*

Y las operaciones y funciones disponibles son las siguientes:

Instrucción	**Descripción**
READ	Leer bytes
WRITE	Escribir bytes
INPUT	Lee línea de texto
PRINT	Escribe línea de texto

Función	**Descripción**
EOF	Comprobar si hay bytes disponibles para leer
CREC	Obtener número de bytes leídos
NREC	Obtener número de bytes total recibidos
SREC	Obtener número de bytes pendientes por leer (recibidos pero no leídos)

Las instrucciones READ y WRITE tienen la misma sintaxis y comportamiento que las de UDP, al igual que las funciones, que tienen el mismo funcionamiento.

Las instrucciones INPUT y PRINT se usarán en los protocolos de aplicación basados en texto (como HTTP), y permiten leer o escribir líneas de texto:

INPUT #*número fichero, variable cadena*
INPUT #*número fichero, variable cadena, timeout*
PRINT #*número fichero, …*

El siguiente ejemplo realiza una conexión TCP a un servidor HTTP, se descarga la página "index.htm" y la escribe en pantalla:

```
open network tcp(80) #1, "192.168.1.100"
' Enviar commando
print #1,"GET /index.htm HTTP/1.1"
print #1, "Host: 192.168.1.100"
print #1
' Leer respuesta
if not eof(1) then
   while not eof(1)
      input #1, line$, 1500
      if not empty(line$) or not eof(1) then print line$
   wend
else
   print "Timeout"
endif
close #1
```

Ficheros multimedia

Este tipo de ficheros son una abstracción de iBASIC para la reproducción de ficheros de audio o video.

En el caso particular de ficheros de video, al reproducirlos se muestra una ventana para tal fin.

Existen instrucciones para la reproducción, pausa y detección del contenido multimedia.

El procedimiento de manejo de un fichero de este tipo es el siguiente:

- Abrir el fichero: orden OPEN
- Realizar operaciones: órdenes PLAY, STOP, PAUSE, GOTO
- Cerrar el fichero

Apertura

La apertura se realiza con la orden OPEN, que tiene la siguiente sintaxis:

OPEN MEDIA #*número fichero*, *cadena nombre fichero*

Operaciones

Las operaciones y funciones disponibles son las siguientes:

Instrucción	Descripción
PLAY	Reproducir el fichero
PAUSE	Pausar la reproducción
STOP	Parar la reproducción
GOTO	Ir a un instante de tiempo

Función	Descripción
EOF	Comprobar si se terminó la reproducción
CREC	Obtener instante de tiempo actual de reproducción (ms.)
NREC	Obtener longitud de la reproducción (ms.)
XSIZE	Devuelve la anchura del video en reproducción (pixels)
YSIZE	Devuelve la altura del video en reproducción (pixels)

La instrucción principal es PLAY con la cual se inicia o se reanuda la reproducción del fichero:

PLAY #*número fichero*
PLAY #*número fichero, opciones*

Por defecto, PLAY reproduce el contenido una sola vez, y en segundo plano.

Opcionalmente pueden indicarse una serie de opciones, separadas por comas:

Opción PLAY	Descripción
REPEAT	Repetir la reproducción
WAIT	Esperar hasta que acabe la reproducción
FULL SCREEN	Reproducción a pantalla completa (video)
WINDOW *xsize*, *ysize*	Reproducción en ventana (video), al tamaño indicado
LOCATE *x, y*	Posición de la ventana de reproducción (video)
TITLE *titulo*	Título de la ventana de reproducción (video)

Las órdenes PAUSE y STOP no tienen parámetros adiciones:

PAUSE #*número fichero*
STOP #*número fichero*

Al usarse PAUSE para pausar la reproducción, ésta puede reanudarse usando de nuevo PLAY.

La instrucción GOTO permite saltar la reproducción a un punto determinado del fichero, indicando el instante de tiempo en milisegundos:

GOTO #*número fichero, posición*

El siguiente ejemplo reproduce un fichero MP3 y visualiza en pantalla la marca de tiempo de reproducción:

```
OPEN MEDIA #1, “tema1.mp3”
PLAY #1
WHILE NOT EOF(#1)
  LOCATE 1,10:PRINT CREC(#1)
WEND
CLOSE #1
```

A continuación un segundo ejemplo que reproduce un video, en una ventana, y espera a su finalización:

```
OPEN MEDIA #1, “video1.avi”
PLAY #1, LOCATE 100, 200, TITLE “Mi video”, WAIT
CLOSE #1
```

Cálculo matricial

Además del cálculo escalar, iBASIC soporta de forma nativa el cálculo matricial.

Solo lenguajes o entornos más específicos soportan esta característica, como FORTRAN o Matlab.

Con iBASIC, el uso de vectores o matrices es igual de fácil que usar números escalares.

Modo de operación

iBASIC dispone de dos modos de funcionamiento, escalar (el tradicional) y matricial.

El modo se selecciona por código, y se puede cambiar de uno a otro cuantas veces se quiera:

SET MODE SCALAR: cambiar a modo escalar
SET MODE MATRIX: cambiar a modo matricial

Por defecto, iBASIC funciona en modo escalar.

Sólo es en el modo matricial en donde se podrán realizar las operaciones sobre matrices y usar la extensión de funciones matriciales.

La razón de disponer de estos son modos es la velocidad de ejecución del programa BASIC.

En el modo matricial, todos los números y expresiones son tratados como matrices, por lo que la velocidad de ejecución se ve sensiblemente mermada, disminuyendo sobre el 10% con respecto al modo escalar.

Es por ello, que se puede conmutar a modo matricial (SET MODE MATRIX) cuando se necesiten hacer cálculos matriciales para luego volver al escalar (SET MODE SCALAR).

Definición de matrices

La definición de vectores y matrices se hace de la forma habitual en BASIC, usando la orden DIM, y opcionalmente, con la orden OPTION BASE para definir el índice inicial:

DIM a(4) : define un vector de 5 elementos, a(0) hasta a(4)

DIM b(2,2): define una matrix 3x3, b(0,0) hasta b(2,2)

DIM c : sólo en modo matricial. Define una matriz dinámica, para ser usada para resultados de los cálculos.

Debido al funcionamiento de la orden DIM, en donde el índice comienza en cero, es aconsejable usar la orden OPTION BASE 1 para definir el índice inicial a uno:

OPTION BASE 1

DIM a(4): define un vector de 4 elementos, a(1) hasta a(4)

DIM b(2,2): matriz de 2x2, b(1,1) hasta b(2,2)

Operaciones con matrices

Se disponen de todas las operaciones y funciones matemáticas escalares pero reinterpretadas al ámbito matricial.

El siguiente programa muestra un ejemplo de las operaciones que se pueden realizar:

```
OPTION BASE 1
SET MODE MATRIX
DIM a(4), b(2,2), c
a = 1
b = MXIDENT(2)
b(1, *) = a(2 .. 3)
c = COS(MXRND(2, 2)) * b
IF c = 0 THEN PRINT "Matriz es cero"
PRINT c(1,*):PRINT c(2, *)
```

Paso a paso:

OPTION BASE 1: define el índice inicial de las matrices a 1

SET MODE MATRIX: conmutar a modo matricial

DIM a(4), b(2,2), c : define un vector, una matriz y una matriz dinámica

a = 1 : asignar valor 1 a todos los elementos de la matriz “a”.

b = MXIDENT(2) : asignar matriz identidad 2x2 a la matriz “b”.

b(1, *) = a(2 .. 3) : asignar los elementos "a(2)" hasta "a(3)" a la fila 1 de "b"

c = COS(MXRND(2,2)) * b : Calcula una matriz 2x2 con elementos aleatorios. Luego calcula sus cosenos, lo multiplica, término a término, por la matriz "b" y finalmente asigna el resultado a la matriz "c".

IF c = 0 THEN PRINT "Matriz es cero" : comprueba si todos los elementos de "c" valen cero.

PRINT c(1,*):PRINT c(2, *) : imprime las dos filas de la matriz "c".

Todas las operaciones y funciones matemáticas presentes en el modo escalar, en el modo matricial se pueden seguir usando pero aquí operarán sobre cada elemento de las matrices.

Funciones matriciales

Hay un conjunto de funciones específicas, que solo están disponibles en modo matricial, que operan sobre matrices, y cuyos nombres empiezan por MX:

Función matricial	Descripción
MXIDENT	Devuelve una matriz identidad del orden indicado
MXRND	Devuelve una matriz de elementos aleatorios
MXONES	Devuelve una matriz con todos sus elementos a 1.
MXPROD	Devuelve el producto vectorial de dos matrices
MXDET	Devuelve el determinante de una matriz
MXINV	Devuelve la matriz inversa
MXTRANS	Devuelve la matriz transpuesta
MXDIM	Devuelve el número de dimensiones de una matriz
MXCOF	Devuelve la matriz de cofactores
MXORDER	Devuelve el orden de una matriz
MXTRACE	Devuelve la traza de una matriz
MXROWS	Devuelve el número de filas de una matriz
MXCOLUMNS	Devuelve el número de columnas de una matriz
MXISUM	Devuelve la suma de todos los elementos de una matriz
MXIPROD	Devuelve el producto de todos los elementos de una matriz

Compatibilidad de matrices

Cuando se hacen operaciones con matrices, iBASIC comprueba que éstas sean compatibles, esto es, que la operación se pueda realizar.

Siguiendo el ejemplo anterior, no se puede hacer la igualdad b=a porque tienen distintas dimensiones, sin embargo sí que es posible la asignación de las submatrices b(1, *) = a(2..3) ya que ambas (submatrices) tienen la misma dimensión (1) y tamaño (2 elementos).

Un caso particular son las matrices que se definen como "dinámicas", como el caso de la matriz "c" del ejemplo anterior (DIM c).

Estas matrices, cuando se usan en una orden de asignación, se adaptan a las dimensiones de la matriz resultante del cálculo.

Submatrices

Como se ha visto en el ejemplo, iBASIC permite especificar submatrices, tanto a la izquierda como a la derecha de las expresiones, esto es, se pueden extraer o asignar zonas de una matriz.

La sintaxis para especificar submatrices es sencilla, como se indica en los siguientes ejemplos:

DIM c(4,4) : matriz 4x4 (con OPTION BASE 1)

c : acceso a toda la matriz

c(2,2) : elemento (2,2)

c(1, *) : submatriz fila 1 (todos los elementos de la fila 1)

c(*, 1) : submatriz columna 1

c(1, 2..3) : submatriz elementos 2 a 3 de la fila 1

c(2..3, 1) : submatriz elementos 2 a 3 de la columna 1

c(*, 2..3) : submatriz (4x2) con las columnas 2 a 3

c(*, 2..+1) : submatriz (4x2) con las columnas 2 a 3

c(*, 2..#2) : submatriz (4x2) con las columnas 2 a 3

c**(2..3, *)** : submatriz (2x4) con las filas 2 a 3

Las submatrices pueden usarse en cualquier expresión, tanto como operando y como destino de los cálculos:

c(*, 2..3) = 0 : pone a cero las columnas 2 y 3

Funciones de usuario (DEF FN)

Con la orden DEF FN se puede definir funciones de usuario que ejecutan una fórmula (numérica o de cadena) y devuelve su resultado.

Las funciones de usuario deben estar definidas en una única línea, y no disponen de variables locales (ámbito de ejecución), sino que ésta se ejecuta dentro del ámbito actual.

Las funciones de usuario se utilizarán dentro de cualquier expresión, como si de una función normal se tratara, no pudiendo contener ningún tipo de lógica, solamente la fórmula para obtener el valor o cadena resultante.

Definición:

La definición de una función se realiza por medio de la orden DEF FN:

DEF FN *nombre función* (*parámetros*) = *expresión función*

Siendo “expresión función” la fórmula que realiza la función, utilizando los nombres de los parámetros que se indiquen.

El nombre de la función determina si ésta es de tipo numérico o de cadena. Si termina con el signo “$” entonces es de cadena, sino es numérica.

Dentro de la expresión (fórmula) pueden utilizarse otras funciones y variables, correspondiendo éstas al ámbito actual de ejecución (siempre que no coincidan con los nombres de los parámetros).

También es posible con recursividad, usándose para ello la función de acción IF ó IF$ para establecer la condición de terminación.

El siguiente ejemplo define una función que calcula la media de dos números:

```
DEF FNmedia(a,b) = (a+b)/2
```

El siguiente, muestra la definición de una función de tipo cadena que devuelve la cadena pasada como parámetro con su primer carácter convertido a mayúscula:

```
DEF FNpmayus$(a$) = upper$(a$[1]) + a$(2 . .*)
```

Un ejemplo de función recursiva es el siguiente, que devuelve la cadena invertida de la entrada:

```
DEF FNinv$(a$)=if$(len(a$) > 1, right$(a$, 1) + FNinv$(left$(a$, len(a$)-1)), a$)
```

Utilización:

El uso de la función de usuario, dentro de una expresión, se realiza de la siguiente forma:

FN*nombre función*(argumentos)

Los siguientes dos ejemplos muestran el uso de las funciones definidas anteriormente:

PRINT FNmedia(10,20) → imprime 15
PRINT FNpmayus$(“la casa”) → imprime “La casa”
PRINT FNinv$(“casa”) → imprime “asac”

Procedimientos de usuario

iBASIC permite programas con o sin procedimientos.

A diferencia de una función de usuario de tipo DEF FN, un procedimiento puede verse como un pequeño programa, con su lógica y variables propias, que realiza una determinada operación.

En cualquier caso, el programa siempre empieza a ejecutarse en la primera línea existente, la cual debe ser una orden de programa.

Como se mencionó anteriormente, variables de un procedimiento no son accesibles desde otro.

Un procedimiento está definido por:

- Un nombre
- Una lista de parámetros de entrada y salida
- Un valor de retorno (opcional)

Su definición se realiza con las instrucciones PROCEDURE y ENDIF, estando entre ellas el cuerpo del procedimiento.

Instrucción	**Descripción**
PROCEDURE	Inicio definición de procedimiento
ENDIF	Finalizar procedimiento
CALL	Llamar a procedimiento

Función	**Descripción**
CALL	Llamar y devolver valor numérico de retorno de procedimiento
CALL$	Llamar y devolver valor de cadena de retorno de procedimiento

Estructura del programa

Un procedimiento puede estar en cualquier lugar del programa, pero la ejecución no puede "alcanzar" a ninguna instrucción "PROCEDURE".

Ejemplo:

```
a=1:b=1
call sumar(a,b,c)
print c
end
procedure sumar(in n1, in n2, out res)
   res = n1 + n2
endproc
```

Este es un ejemplo legal de programa con un procedimiento, que comienza a ejecutarse, llama al procedimiento, imprime el resultado y acaba.

Si el programa anterior fuese:

```
a=1:b=1
call sumar(a,b,c)
print c
REM **** Se suprime orden: end
procedure sumar(in n1, in n2, out res)
   res = n1 + n2
endproc
```

Aquí se ha comentado la orden “end”.

El programa empieza a ejecutarse de igual modo al anterior, llama al procedimiento, imprime el resultado y a continuación se encuentra con la instrucción “procedure”. Esto causa un error y el programa se detiene.

Así pues, la estructura de un programa con procedimiento debe ser:

```
‘ Cuerpo principal del programa
‘ Llamadas a procedimientos
CALL …
CALL …
‘ Fin del programa
END
‘
‘ Definición de procedimientos
‘
PROCEDURE ….
   .
   .
ENDIF
PROCEDURE ….
   .
   .
ENDIF
```

Definición de un procedimiento

Un procedimiento se define con las instrucciones PROCEDURE y ENDIF, las cuales marcan el inicio y final del mismo.

Con PROCEDURE se indica su nombre y la lista de argumentos de entrada y salida:

PROCEDURE *nombre procedimiento* (*argumento1, argumento2, …)*

Se pueden definir procedimientos sin argumentos:

PROCEDURE *nombre procedimiento*

De forma opcional también se puede especificar un valor de retorno:

PROCEDURE *nombre procedimiento* (*argumento1, argumento2, …)* RETURNS *variable*

Los argumentos pueden ser numéricos, de cadena, tanto escalares como matrices, ya sean de entrada o de salida.

En caso de indicador valor de retorno, su especificación (RETURNS) se hace usando un nombre de variable, cuyo valor final será el valor de retorno del procedimiento.

El tipo del valor de retorno es el mismo al indicado por el tipo de variable.

La especificación de un argumento tiene el siguiente formato:

dirección nombre_argumento
dirección nombre_argumento()

"Dirección" indica si el argumento es de entrada (IN), de salida (OUT) o ambos (INOUT).

"Nombre_argumento" es el nombre de la variable, local al procedimiento, que recibirá el valor de entrada, o bien, proporcionará el valor de salida.

Por defecto los argumentos son escalares. Para los matriciales se añade "()".

En resumen:

IN *variable*: declara un argumento de entrada escalar, cuyo valor estará en la variable indicada.

IN *variable()*: declara un argumento de entrada matricial (array)

OUT *variable*: declara un argumento de salida escalar. El valor de la variable será copiado a la variable indicada en CALL.

OUT *variable()*: declara un argumento de salida matricial (array).

INOUT *variable*: declara un argumento de entrada y salida escalar.

INOUT *variable()*: declara un argumento de entrada y salida matricial (array).

La diferencia entre OUT y INOUT está en que, un parámetro OUT, al comienzo del procedimiento, la variable asociada del procedimiento no tendrá valor, porque ésta es solo de salida, a diferencia de INOUT, que sí tendrá valor de entrada y además, en la finalización, será copiado a la variable asociada en CALL.

En modo matricial, y para argumentos matriciales, se puede llamar al procedimiento especificando una matriz completa, o bien, una submatriz.

Ejemplos:

```
procedure sumar(in a, in b, out c)
   c = a + b
endproc

procedure sumar2(in a, in b) returns c
   c = a + b
endproc

procedure invertir(in a$, out b$)
   for i = len(a$) to 1 step -1
      inc b$, a$[i]
   next
endproc
```

```
procedure vsumar(in vector(), out suma)
   for i = 1 to dbound(vector, 1)
      inc suma, vector(i)
   next
endproc
```

El primer ejemplo simplemente realiza la suma de dos números.

El segundo es igual al primero pero ahora la suma es devuelta como valor de retorno.

El tercer ejemplo, invierte los caracteres de una cadena.

El último ejemplo realiza la suma de los elementos de un vector.

Llamada a un procedimiento

La llamada a un procedimiento, definido en el programa, se realiza usando la orden CALL:

CALL *nombre procedimiento* (*parámetro1, parámetro2, …*)

La llamada a un procedimiento que no tenga argumentos se realiza directamente:

CALL *nombre procedimiento*

Si el procedimiento tiene valor de retorno y se desea almacenarlo, se llamará de la siguiente forma:

CALL *nombre procedimiento* (*parámetro1, parámetro2, …*) RETURNS *variable*

Almacenándose el valor de retorno en la variable indicada.

El número y tipo de los parámetros debe de coincidir con lo especificado en la definición del procedimiento:

Parámetros de entrada escalares: se permite cualquier expresión numérica o de cadena, según el tipo.

Parámetros de entrada matricial numéricos: depende del modo de ejecución:

Modo escalar: sólo se puede indicar una variable de tipo matriz.
Modo matricial: se permiten expresiones matriciales

Parámetros de entrada matricial de cadena: sólo se puede indicar una variable de tipo matriz.

Parámetros de salida escalar/matricial: solo se permite indicar una variable. En modo matricial, si el parámetro de salida es numérico matricial, se puede indicar una submatriz.

Siguiendo los ejemplos anteriores, llamadas legales son:

```
call sumar(10, 20, suma1)
call sumar2(10, 20) returns suma2
call sumar(suma1*20, 100, suma3)
call invertir("hola", a$)

dim vector(100)
call vsumar(vector, suma3)
```

Ejemplos ilegales serían:

```
call sumar(10, 20, a+b) → tercer parámetro debe ser una variable
call invertir("hola", c) → segundo parámetro debe ser una variable de cadena
call vsumar(10, suma3) → primer parámetro debe ser una matriz
```

Un ejemplo de uso en modo matricial seria:

```
set mode matrix
call vsumar(round(mxrnd(100)*10), suma4)
```

Aquí se crea un vector de 100 elementos, de valores aleatorios entre 0 y 10, y se realiza su suma interna.

De forma opcional puede indicarse la dirección de los argumentos en la orden CALL. Su inclusión no afecta en la ejecución, pero sí que debe de coincidir con lo especificado en la definición del procedimiento. Se usará para fines de documentación y legibilidad del código:

call vsumar(**in** round(mxrnd(100)*10), **out** suma4)

De esta manera, y sin necesitar ver la definición de procedimiento, se observa qué argumentos son de entrada o salida.

Si el tercer argumento, en lugar de indicar "out" se hubiera usado "in", produciría un error por no coincidir con la especificación del procedimiento.

<u>Llamada a un procedimiento dentro de expresiones</u>

Cualquier procedimiento puede ser invocado desde dentro de una expresión, numérica o de cadena.

Para ello se utilizan las funciones CALL() y CALL$():

CALL(*nombre procedimiento*(*argumentos*))
CALL$(*nombre procedimiento*(*argumentos*))

La sintaxis es sencilla, únicamente indicar dentro de CALL ó CALL$ el procedimiento a llamar, junto con sus argumentos, y devolverá el valor el retorno de dicho procedimiento.

La diferencia entre ambas es que CALL devuelve un número y CALL$ una cadena, por lo que, los procedimientos a llamar deberán ser del tipo adecuada.

Como excepción a esto último, se permite llamar a procedimientos de distinto tipo, realizándose la conversión adecuadamente.

Por ejemplo, el caso de usar CALL con un procedimiento que devuelve una cadena, ésta se intentará convertir a número, y de forma equivalente con CALL$. Si se llama a un procedimiento que devuelve un número, éste se convertirá a cadena que será que lo devuelva CALL$.

Por último, también es posible utilizar CALL() y CALL$() con procedimientos que no devuelvan valor ninguno, y en tal caso, CALL() devolverá 0 y CALL$() cadena vacia.

Siguiendo el ejemplo anterior del procedimiento "sumar2", se podría hacer:

PRINT 100 + CALL(sumar2(3, 2)) → imprime 105

A$ = CALL$(sumar2(3, 2)) → A$ = "5"

CALL() y CALL$() también permiten que el valor de retorno sea almacenado en una variable, además de ser devuelto por ellas mismas:

```
PRINT 100 + CALL(sumar2(3, 2) returns s2) → imprime 105
PRINT s2 → imprime 5
```

Procedimientos con argumentos de salida matriciales

En aquellos procedimientos que tengan argumentos de salida que sean matrices (arrays), estas matrices serán creadas dentro del procedimiento (con la orden DIM), y por tanto puede ser llamado el procedimiento sin necesidad de haberlas creados (las matrices) previamente.

Esto es aplicable tanto en modo escalar como matricial.

El siguiente ejemplo define un procedimiento con dos argumentos de tipo matriz, uno de entrada y otro de salida:

```
procedure vinvertir(in vector(), out ivector())
    num = dbound(vector, 1)
    dim ivector(num)
    for i = 1 to num
        ivector(num – i + 1) = vector(i)
    next
endproc
```

Este ejemplo devuelve en el vector "ivector" una copia del vector de entrada "vector" pero con sus elementos invertidos de posición (el primero al último).

Como se observa, dentro del procedimiento, hay una orden DIM para crear el vector de salida "ivector", que se crea al mismo tamaño al de entrada.

Las siguientes líneas muestran un ejemplo de llamada:

```
' Construye vector de entrada
dim a(10)
for i = 1 to 10 : a(i) = i : next
call vinvertir(a, b)  ' Vector "b" no se ha definido
for i = 1 to 10 : print b(i) : next
```

El resultado del programa es la impresión de los números 10 al 1.

Al procedimiento "vinvertir" se llama con el vector de entrada "a", el cual se ha creado y asignado valores, y con el vector de salida "b", que no se ha creado, sino que esto ocurre dentro del procedimiento.

En caso de caso de llamar a un procedimiento con un vector de salida existente, éste será destruido y construido según el funcionamiento interno del procedimiento.

Siguiendo con el ejemplo anterior, "vinvertir" podría llamarse de la siguiente forma:

```
dim a(10), b(100)
for i = 1 to 10 : a(i) = i : next
print dbound(b, 1)   → Imprime 100
call vinvertir(a, b)
print dbound(b, 1)   → Imprime 10
```

Aquí se está llamado a "vinvertir" con el vector de salida "b" que existe, y cuyo tamaño es de 100 elementos. Sin embargo, como éste se construido dentro del procedimiento, en el retorno de CALL, el vector "b" es reconstruido según la creación DIM que existe en "vinvertir", en este caso a 10 elementos.

En modo matricial, un procedimiento puede retornar una matriz (clausula RETURNS) y ser usado libremente:

```
procedure vinvertir2(in vector()) returns ivector
    num = dbound(vector, 1)
    dim ivector(num)
    for i = 1 to num
        ivector(num – i + 1) = vector(i)
    next
endproc
```

Y se usaría de forma equivalente a los ejemplos anteriores:

```
set mode matrix
dim a(10)
for i = 1 to 10 : a(i) = i : next
call vinvertir2(a)  returns b
print b
```

También es posible usar directamente la función CALL():

```
DIM b
b = call(vinvertir2(a))
```

Procedimientos externos

Un procedimiento externo es una función “estándar” que reside en una librería dinámica (DLL) y que ha sido publicada para poder ser invocada externamente.

Así pues, iBASIC permite llamar a todo tipo de funciones, pasándolas parámetros y recogiendo sus resultados.

Hay dos instrucciones:

Instrucción	Descripción
EXTERN PROCEDURE	Declarar procedimiento externo
CALL	Llamar a procedimiento externo

Declaración de un procedimiento externo

Para realizar la declaración de un procedimiento, primero hay que tener la especificación de su interfaz que, típicamente, estará escrita en lenguaje C.

Teniendo la interfaz, hay que “traducirla” a la sintaxis de iBASIC, utilizando la orden EXTERN PROCEDURE.

Hecho esto, la función/procedimiento es invocable de la forma regular, usando CALL.

EXTERN PROCEDURE tiene el siguiente formato general:

EXTERN PROCEDURE “*nombre fun*”,”*fichero dll*”,*nombre_proc*(*argumento1*, …) RETURNS *retorno*

siendo:

nombre fun: nombre de la función/procedimiento dentro de la DLL.

fichero dll: nombre del fichero de la librería DLL.

nombre_proc: nombre a dar al procedimiento, para ser usado en el programa.

argumentoX : especificación de un argumento. Tipo, longitud, y opcionalmente su nombre

retorno: especificación del tipo y longitud del valor de retorno.

La especificación del valor de retorno es opcional y puede omitirse:

EXTERN PROCEDURE “*nombre fun*”,”*fichero dll*”,*nombre_proc*(*argumento1*, …)

Para funciones sin argumentos, se usa directamente:

EXTERN PROCEDURE "*nombre fun*","*fichero dll*",*nombre_proc*
EXTERN PROCEDURE "*nombre fun*","*fichero dll*",*nombre_proc* RETURNS retorno

La especificación de un **argumento** sigue el formato:

dirección nombre = *tipo* (*longitud*)

siendo:

dirección: dirección del argumento:

IN = argumento de entrada
OUT = argumento de salida
INOUT = argumento de entrada y salida

nombre: nombre del argumento (opcional)

tipo: tipo del argumento:

INT = número entero con signo
UINT = número entero sin signo
FLOAT = número de punto flotante
STRING$ = cadena de caracteres o buffer de bytes

longitud: longitud del argumento en bytes. Los valores permitidos dependen del tipo indicado:

INT = 1, 2 ó 4 bytes (enteros de 8, 16 ó 32 bits)
UINT = 1, 2 ó 4 bytes
FLOAT = 4 ó 8 bytes (números de 32 ó 64 bits)
STRING$ = de 0 a 2^{32} bytes (opcional)

Para argumentos de tipo cadena (STRING$), la especificación de su longitud es opcional y tiene los siguientes efectos:

- Argumentos de entrada: el tamaño es ignorado
- Argumentos de salida: en caso de indicar tamaño, se crea una cadena de dicha longitud para almacenar los datos de salida. Si no se indica tamaño, se usará la longitud de la variable asociada en CALL para realizar la llamada.
- Argumentos de entrada y salida: igual al caso de "solo salida".

El nombre del argumento es opcional y puede omitirse, ya que solo tiene carácter informativo:

nombre = *tipo* (*longitud*)

La especificación del valor de retorno, es opcional, y tiene el siguiente formato:

tipo (longitud)
nombre = tipo (longitud)

El siguiente ejemplo muestra la definición de la función "GetComputerName" de Windows:

extern procedure "GetComputerNameA", "kernel32.dll", GetComputerName(out string$, inout int(4))

Traducción de la interfaz C

Hay ciertas reglas a seguir para realizar la declaración de de un procedimiento externo partiendo de su interfaz en C:

Dirección	Tipo dato	Valor / puntero	Formato C	Formato iBASIC
Entrada	Entero	Por valor	int num	IN num = INT(4)
Entrada	Flotante	Por valor	double num	IN num = FLOAT(8)
Entrada	Entero	Puntero	int *num	INOUT num = INT (4)
Entrada	Flotante	Puntero	double *num	INOUT num = FLOAT(8)
Entrada	Vector enteros	Puntero	int *vector	IN vector= STRING$
Entrada	Vector flotantes	Puntero	double *vector	IN vector= STRING$
Entrada	Cadena ASCIIZ	Puntero	char *str	IN str = STRING$
Entrada	Estructura	Puntero	struct *st	IN st = STRING$
Salida	Entero	Puntero	int *num	OUT num = INT (4)
Salida	Flotante	Puntero	double *num	OUT num = FLOAT(8)
Salida	Vector enteros	Puntero	int *vector	OUT vector= STRING$
Salida	Vector flotantes	Puntero	double *vector	OUT vector= STRING$
Salida	Cadena ASCIIZ	Puntero	char *str	OUT str = STRING$

Ejemplo:

Sea la función “ReadFile” con la interfaz:

```
BOOL WINAPI ReadFile(
  __in         HANDLE hFile,
  __out        LPVOID lpBuffer,
  __in         DWORD nNumberOfBytesToRead,
  __out_opt    LPDWORD lpNumberOfBytesRead,
  __inout_opt  LPOVERLAPPED lpOverlapped
);
```

Su traducción a iBASIC sería:

```
extern procedure "ReadFile", "kernel32.dll", ReadFile(
                 in    hFile       = uint(4),
                 out   Buffer      = string$,
                 in    BytesToRead = uint(4),
                 out   BytesRead   = uint(4),
                 inout Overlapped  = string$
                 ) returns RetCode = uint(4)
```

Llamada a un procedimiento externo

La llamada a un procedimiento externo se realiza prácticamente de la misma forma que a un procedimiento regular de usuario:

CALL *nombre procedimiento* (*parámetro1, parámetro2, …*) RETURNS *variable*

Si el procedimiento no devolviera nada (void) o bien, no se desea almacenar su valor:

CALL *nombre procedimiento* (*parámetro1, parámetro2, …*)

También es posible la llamada usando las funciones CALL() y CALL$().

De nuevo, los parámetros deben coincidir en número y en tipo con lo indicado en EXTERN PROCEDURE.

Opcionalmente también se puede indicar la dirección de cada parámetro, y en tal caso, deben de coincidir con lo especificado en PROCEDURE.

En aquellos argumentos que sean opcionales se puede indicar un valor de cero, en lugar de un nombre de variable o expresión. Esto es tanto válido para argumentos de entrada como de salida.

Para argumentos que han sido especificados como STRING$ y su dirección es IN o INOUT, ya sea porque son cadenas, buffers, estructura o vectores, debe crearse de forma previa una variable de tipo cadena con el tamaño adecuado para realizar la operación.

Siguiendo el ejemplo anterior, se realiza la llamada usando todos los argumentos, excepto el último:

```
buffer$ = SPACE$(512)
CALL ReadFile(file, buffer$, len(buffer$), bytesread, 0) RETURNS retcode
```

En el ejemplo se realiza una lectura del fichero de 512 bytes. En la variable “byteread” se guardará el número de bytes realmente leidos. El último parámetro se ha pasado un valor 0 para indicar que no se usa.

El ejemplo, con la indicación de dirección de argumentos, sería:

```
CALL ReadFile(IN file, OUT buffer$, IN len(buffer$), OUT bytesread, INOUT 0) RETURNS retcode
```

También se podría haber ignorado el parámetro de “número de bytes leídos”:

```
CALL ReadFile(file, buffer$, len(buffer$), 0, 0)
```

Hay que fijarse que se está usando la cadena “buffer$” como un buffer de caracteres, y que se ha creado con un tamaño de 512 bytes, usando la función SPACE$.

Para argumentos que sean estructuras o vectores, al tener que usar una cadena, son necesarios métodos manuales para su creación y extracción, usando las funciones MK*$ y CV*.

Subprogramas

Un subprograma es un fichero de programa, separado del programa principal, que puede ser cargado y ejecutado dinámicamente.

En una situación típica, se tendrá el programa principal, y posiblemente uno o más subprogramas (en ficheros distintos), que contendrán subrutinas o procedimientos, los cuales serán llamados por el programa principal, usando las órdenes habituales CALL o GOSUB.

También se permite que un subprograma haga uso de otros subprogramas a su vez.

Para poder usar un subprograma, éste debe cargarse en memoria y asignarle un identificador (un nombre).

Hay tres instrucciones de carga/descarga de subprogramas:

Instrucción	**Descripción**
LOAD SUBPROGRAM	Cargar subprograma en memoria
RUN SUBPROGRAM	Cargar y ejecutar subprograma
UNLOAD SUBPROGRAM	Descargar subprograma de memoria

Una vez cargado las instrucciones habituales de salto y llamada:

Instrucción	**Descripción**
GOTO	Saltar a línea
GOSUB	Llamar a subrutina
CALL	Llamar a procedimiento (usuario / externo)
RUN SUBPROGRAM	Ejecutar subprograma

La sintaxis de estas órdenes es igual a la normal, pero ahora, además, se especifica el nombre del subprograma.

<u>Relación subprogramas con programa principal</u>

El programa principal y todos los subprogramas comparten los siguientes elementos:

- Variables globales (ámbito actual de variables)
- Pilas de usuario
- Ficheros
- Formato representación números enteros (litte/big endian)
- Modo numérico (escalar/matricial) en llamadas a procedimientos

Según lo anterior, si el programa principal hace una llamada a subrutina (GOSUB) a un subprograma, las variables, ficheros y demás elementos anteriores son accesibles desde el subprograma.

Por otra parte, las secciones DATA son independientes entre subprogramas, de manera que cualquiera de ellos puede tener las suyas, y realizar las operaciones READ y RESTORE de forma aislada.

<u>Carga de un subprograma</u>

Las instrucciones LOAD SUBPROGRAM y RUN SUBPROGRAM son las encargadas de cargar un subprograma, siendo su sintaxis idéntica:

LOAD SUBPROGRAM *cadena nombre fichero*, *nombre subprograma*

RUN SUBPROGRAM *cadena nombre fichero*, *nombre subprograma*

La diferencia entre ambos, es que RUN SUBPROGRAM, además de cargarlo, lo ejecuta. Al hacer esto, el programa actual se quedará a la espera hasta que el subprograma acabe, continuando su ejecución a continuación.

Habitualmente RUN SUBPROGRAM se usará con subprogramas que declaren procedimientos externos (ficheros interfaz).

Se pueden cargar tantos subprogramas sean necesarios, siempre y cuando sus nombres sean distintos.

El ejemplo siguiente carga un subprograma que se encuentra en el fichero “calculos.bas”, dándole como nombre (identificador) el de “calc”:

LOAD SUBPROGRAM “calculos.bas”, calc

Ahora se carga y ejecuta un subprograma que hace de “interfaz” (declara procedimientos externos) de funciones API de Windows:

RUN SUBPROGRAM “win.bas”, win

RUN SUBPROGRAM admite una segunda sintaxis para ejecutar un subprograma que esté cargado en memoria:

RUN SUBPROGRAM *nombre subprograma*

Por ejemplo, para el caso del subprograma “win” se podría haber hecho lo siguiente:

LOAD SUBPROGRAM “win.bas”, win
RUN SUBPROGRAM win

<u>Descarga de un subprograma</u>

Opcionalmente, un subprograma cargado puede descargarse de memoria, por dos motivos:

- No va a ser usado más.
- Se va a cargar otro subprograma distinto con el mismo nombre

La instrucción que se utiliza para ello es UNLOAD SUBPROGRAM:

UNLOAD SUBPROGRAM *nombre subprograma*

Siguiendo el ejemplo anterior de “calculos.bas”, su descarga sería de la siguiente manera:

UNLOAD SUBPROGRAM calc

Saltos a subprogramas (GOTO)

De igual manera que se pueden hacer saltos incondicionales dentro de un programa, también se pueden realizar a un subprograma:

GOTO *número línea@subprograma*

Al hacer esto, la ejecución proseguirá en la línea (o etiqueta) del subprograma indicado, que cuando finalice, finalizará también el programa actual.

Hay que tener en cuenta que el subprograma tendrá acceso a las variables, pilas y ficheros actuales.

El siguiente ejemplo hace un salto a la línea con etiqueta “resultados” del subprograma “calc”:

```
LOAD SUBPROGRAM “calculos.bas”, calc
GOTO resultados@calc
PRINT “esto nunca se imprimirá”
```

En caso de que el subprograma tuviera números de línea, la forma de hacer un GOTO sería análoga:

```
GOTO 500@calc  ‘ Salta a línea 500 del subprograma calc
```

Llamada a subrutina de un subprograma (GOSUB)

De la misma forma se puede invocar a una subrutina que esté en un subprograma:

GOSUB *número línea@subprograma*

Al finalizar la subrutina (con RETURN) que ejecución proseguirá de la forma habitual.

Al igual que con GOTO, el subprograma tendrá acceso a las variables, pilas y ficheros actuales.

Siguiendo el ejemplo, ahora se llama a la subrutina de la línea con etiqueta “calcular_media”:

```
GOSUB calcular_media@calc
PRINT “esto se imprimirá después del GOSUB”
```

Llamada a procedimiento de un subprograma

Esta será la forma más habitual de uso de subprogramas, el llamar a procedimientos en él contenidos.

Para ello se utiliza la orden CALL, usando ahora la especificación de subprograma (con @):

CALL *nombre procedimiento@subprograma* (*parámetros*)

El procedimiento puede ser tanto uno definido por el usuario, como uno externo que esté declarado en el subprograma.

Al realizar CALL, se crea un nuevo ámbito de variables, y el subprograma hereda el modo numérico actual (escalar/matricial).

El ejemplo siguiente llama al procedimiento “obtener_aleatorio”:

```
CALL obtener_aleatorio@calc(100, numero)
PRINT numero
```

Subprogramas interfaz a librerías externas

Uno de los usos más interesantes de los subprogramas es poder usarlos como “interfaz” a librerías DLL de funciones externas, para poder ser usados desde cualquier otro programa de forma transparente.

Este tipo de subprogramas sólo contienen declaraciones EXTERN PROCEDURE que definen la interfaz de las funciones.

De esta manera, cualquier programa con solo cargar el subprograma ya tendrá acceso a esos procedimientos externos, sin necesidad de estar declarándolos él mismo.

El aspecto interno de un subprograma interfaz es el siguiente:

```
EXTERN PROCEDURE …
EXTERN PROCEDURE …
.
.
EXTERN PROCEDURE …
END
```

Un programa que quiera utilizar dichos procedimientos externos deberá cargar y ejecutar el subprograma, y luego, usarse normalmente con CALL:

```
RUN SUBPROGRAM “….”, ….

CALL …@... ( … )
```

Aquí es importante usar RUN SUBPROGRAM y no LOAD SUBPROGRAM, ya que el subprograma necesita ser ejecutado para que las declaraciones de los procedimientos externos sean cargads en memoria.

El típico ejemplo es el uso de un subprograma que haga de interfaz a funciones API de Windows:

```
extern procedure "GetComputerNameA”, "kernel32.dll", GetComputerName(out string$, inout int(4))
extern procedure "GetSystemDirectoryA","kernel32.dll", GetSystemDirectory(out string$, in int(4))
.
.
end
```

Desde un programa, la utilización de uno de ellos (por ejemplo GetComputerName) sería así:

```
RUN SUBPROGRAM “win.bas”, win  ‘ Esto solo hay que hacerlo una vez
computer$=space$(100)
computer_len = len(computer$)
CALL GetComputerName@win(computer$, computer_len)
PRINT “Nombre ordenador: “; computer$
```

Secciones DATA

Dentro de un programa BASIC pueden existir zonas de datos literales, llamados “DATA”, de libre configuración o uso.

Estos datos literales (números o cadenas), se indican usando instrucciones DATA, que como tales no realizan ninguna acción, sino solamente la de almacenar dichos datos.

Los DATA tienen carácter local al ámbito de ejecución, esto es, puede haber DATAs dentro del programa principal, de los procedimientos y de los subprogramas, siendo todos ellos independientes.

Típicamente, estos datos serán constantes o valores para inicializar variables o matrices del programa, que normalmente se realiza cuando el programa (procedimiento o subprograma) se inicia.

El acceso a los valores contenidos en los DATA se efectúa por medio de la instrucción READ, que lee el siguiente dato disponible en esos DATA y lo almacena en la variable que se especifique.

Instrucción	**Descripción**
DATA	Contener datos literales (números o cadenas)
READ	Leer siguiente dato de DATA
RESTORE	Posicionar puntero de lectura de DATA

Función	**Descripción**
DATALEN	Devuelve el número total de elementos en los DATA

Internamente, iBASIC tiene un puntero que “apunta” al siguiente dato pendiente de ser leído por READ, y que, inicialmente, se dirige al primer DATA disponible.

Cada vez que se hace un READ, se lee el dato apuntado por el puntero y éste es movido al siguiente elemento dentro del actual ámbito de ejecución.

Sin embargo, hay ocasiones en que hay que mover manualmente dicho puntero, para que “apunte” a una zona determinada. Esto se realiza con la orden RESTORE.

RESTORE permite dos sintaxis:

```
RESTORE
RESTORE número línea
```

Usando RESTORE directamente, hace que el puntero se sitúe sobre el primer DATA existente en el programa, procedimiento o subprograma.

Usando RETORE con un número de línea (o etiqueta) se consigue que el puntero se posicione sobre el primer DATA existente a partir de dicha línea (o etiqueta), siempre y cuando ésta se encuentre dentro del ámbito actual.

Los datos contenidos en DATA son solo visibles desde el ámbito actual de ejecución, esto es, DATAs definidos en un procedimiento no serán accesibles desde otro, o desde el programa principal.

La función DATALEN es muy interesante ya que devuelve el número de elementos (sean números o cadenas) disponibles en los DATA del ámbito actual.

El siguiente ejemplo muestra un programa sin procedimientos, en donde los DATA están definidos dentro del único ámbito de ejecución, que es el del programa principal:

```
10 READ n → lee 100
20 RESTORE 70
30 READ m → lee 300
40 END
50 DATA 100
60 DATA 200
70 DATA 300
```

Sin embargo, con hay procedimientos:

```
PRINT "Número elementos: "; DATALEN  → imprime 3
READ m
PRINT "Primer elemento: "; m  → imprime 100
DATA 100, 200, 300
END
PROCEDURE proc1
    PRINT "Número elementos: "; DATALEN  → imprime 2
    READ m
    PRINT "Primer elemento: "; m  → imprime 1
    DATA 1, 2
ENDPROC
```

Programación dinámica

Bajo éste término se entiende la capacidad de un programa de modificarse en tiempo de ejecución de tiempo, esto es, dinámicamente.

iBASIC dispone de una instrucción para ello, EVALUATE.

La sintaxis de EVALUATE es sencilla:

EVALUATE *cadena de órdenes*

Lo que hace EVALUATE es coger la cadena indicada y ejecutarla como si de una línea BASIC se tratará.

De esta forma se pueden ejecutar órdenes confeccionadas dinámicamente.

A nivel de ejecución de programa, la cadena indicada en EVALUATE, se considera como si de una línea insertada, en la posición actual, se tratara.

EVALUTE en expresiones

EVALUATE también está disponible en forma de función, tanto numérica como de cadena (EVALUATE$), para poder ser utilizada dentro de expresiones:

EVALUATE (*cadena órdenes , expresión numérica a devolver*)
EVALUATE$(*cadena órdenes , expresión de cadena a devolver*)

En ambos casos, primero se ejecutan las órdenes indicadas (como lo haría una instrucción EVALUATE convencional), y después se evalúa la expresión de retorno, cuyo valor o cadena es lo que devolverá la función.

De esta forma es posible confeccionar expresiones complejas en una sola línea, como muestra el ejemplo siguiente:

```
print evaluate$("read #1, a$", upper$(a$))
```

Esta instrucción lee una línea del fichero abierto #1, y la imprime en pantalla, convirtiéndola a mayúsculas.

Ejemplo

El siguiente programa almacena nombres y edades en variables creadas dinámicamente usando EVALUATE, en función del nombre de la persona indica. De esta forma, el acceso a la edad es inmediato. Las partes interesantes del programa son las subrutinas “introducir” y “consultar”:

```
while true
    print
    print "Elige opción:":print
    print "1: Introducir datos"
    print "2: Consultar datos"
    print
    print "9: Salir"
    repeat
        pause 100
        opcion$ = inkey$
    until opcion$ in ["1", "2", "9"]
    if opcion$="9" then break
    on val(opcion$) gosub introducir, consultar
wend
end

introducir:
    input "Nombre: ", nombre$
    if not empty(nombre$) then
```

```
        input "Edad  : ", edad
        var_nombre$ = nombre$ + "_nombre$"
        var_edad$   = nombre$ + "_edad"
        evaluate var_nombre$ + "= """ + nombre$ + """"
        evaluate var_edad$   + "= " + str$(edad)
    endif
    return

consultar:
    input "Nombre: ", nombre$
    if not empty(nombre$) then
        var_nombre$ = nombre$ + "_nombre$"
        var_edad$   = nombre$ + "_edad"
        evaluate "if empty(" + var_nombre$ + ") then print ""No
tengo datos"":return"
        evaluate "print ""Su edad es:"";" + var_edad$
    endif
    return
```

Generación de ejecutables

Además de poder usar iBASIC en modo intérprete para editar, depurar y ejecutar los programas, también es posible generar ejecutables autocontenidos de los programas realizados, sin necesidad de utilizar iBASIC para ser ejecutados.

A este proceso se le llama “compilación”, y para realizarlo iBASIC dispone de una instrucción que solo se puede emplear en modo interactivo, COMPILE.

Esta instrucción se usará cuando se quiera crear el ejecutable de un programa que esté cargado en memoria:

COMPILE n*ombre fichero ejecutable*

Cuando se utiliza, iBASIC realiza de forma previa una comprobación de sintaxis del programa (igual a lo que haría CHECK SYNTAX).

Una vez generado el ejecutable, éste puede ser usado libre y autónomamente.

Especificación de icono

Opcionalmente, al fichero ejecutable de salida se le puede cambiar el icono, para ello la orden COMPILE dispone de una opción:

COMPILE *fichero ejecutable*, ICON *fichero icono*

También es posible generar el fichero ejecutable sin ningún icono, para ello se debe indicar como “fichero de icono” una cadena vacia:

COMPILE *fichero ejecutable*, ICON “”

Compilación con subprogramas

En programas que usen subprogramas, se puede usar la forma de compilación anterior, pero de hacerlo así, los ficheros ".bas" de los subprogramas deberán ser distribuidos junto al ejecutable generado.

Para evitarlo, se puede usar una segunda sintaxis de COMPILE para que incluya los subprogramas que se indiquen:

COMPILE n*ombre fichero ejecutable,* SUBPROGRAM *fichero subprograma 1, …*

Los ficheros de los subprogramas deben de coincidir y ser sintácticamente correctos (iBASIC hace una comprobación para ello).

Además, dichos nombres de fichero deben de coincidir con los que se usen en los instrucciones LOAD/RUN SUBPROGRAM.

De forma opción, también se puede indicar el icono a usar por la aplicación, además de la especificación de subprogramas:

COMPILE *fichero ejecutable*, ICON *fichero icono*, SUBPROGRAM *ficheros subprogramas*