

# INTRODUCCIÓN A FORTH

F. J. Gil Chica & Jorge Acereda Maciá  
gil@disc.ua.es

Dpto. de Física, Ingeniería de Sistemas y Teoría de la Señal  
Escuela Politécnica Superior  
Universidad de Alicante  
ESPAÑA

\*\*\*

ISBN 978-84-690-3594-8

<http://www.disc.ua.es/~gil/libros.html>

versión enero 2007

*Juan Manuel Gil Delgado,  
in memoriam*

# Índice general

<b>1. Introducción a Forth</b>	<b>5</b>
1.1. Una filosofía distinta . . . . .	5
1.2. Un entorno distinto . . . . .	6
1.3. La máquina virtual . . . . .	8
1.3.1. La pila de parámetros . . . . .	8
1.3.2. Crear nuevas palabras . . . . .	11
<b>2. Pila y aritmética</b>	<b>14</b>
2.1. Vocabulario para la pila . . . . .	14
2.2. Aritmética básica . . . . .	15
2.2.1. Enteros simples . . . . .	15
2.2.2. Enteros dobles . . . . .	17
2.2.3. Operadores mixtos . . . . .	18
2.2.4. Números con signo y sin signo . . . . .	19
2.3. Salida numérica con formato . . . . .	19
2.4. La filosofía del punto fijo . . . . .	23
2.5. Números racionales . . . . .	24
<b>3. Programación estructurada</b>	<b>26</b>
3.1. Operadores relacionales . . . . .	26
3.2. Condicionales . . . . .	26
3.3. Bucles . . . . .	27
3.4. Más bucles . . . . .	30
3.5. Fin abrupto de un bucle . . . . .	31
3.6. El estilo Forth . . . . .	31
<b>4. Constantes y variables. El diccionario</b>	<b>36</b>
4.1. Constantes y variables . . . . .	36
4.2. Inciso: cambio de base . . . . .	40
4.3. Estructura del diccionario . . . . .	41
4.4. La pareja <code>create ...does&gt;</code> . . . . .	44

4.5.	Aplicaciones . . . . .	47
4.6.	Ejecución vectorizada . . . . .	50
4.7.	Distinción entre ' y ['] . . . . .	54
<b>5.</b>	<b>Cadenas de caracteres</b>	<b>59</b>
5.1.	Formato libre . . . . .	59
5.2.	Las palabras <code>accept</code> , <code>type</code> y <code>-trailing</code> . . . . .	59
5.3.	Las palabras <code>blank</code> y <code>fill</code> . . . . .	62
5.4.	Las palabras <code>move</code> y <code>compare</code> . . . . .	62
5.5.	La palabra <code>compare</code> . . . . .	64
5.6.	Algunas funciones útiles . . . . .	65
<b>6.</b>	<b>Control del compilador</b>	<b>68</b>
6.1.	Nueva visita al diccionario . . . . .	68
6.2.	Inmediato pero... . . . .	70
6.3.	Parar y reiniciar el compilador . . . . .	72
<b>7.</b>	<b>Entrada y salida sobre disco</b>	<b>74</b>
7.1.	Un disco es un conjunto de bloques . . . . .	74
7.2.	Cómo usa Forth los bloques . . . . .	75
7.3.	Palabras de interfaz . . . . .	76
7.4.	Forth como aplicación . . . . .	78
<b>8.</b>	<b>Estructuras y memoria dinámica</b>	<b>82</b>
8.1.	Estructuras en Forth . . . . .	82
8.2.	Memoria dinámica . . . . .	84
<b>9.</b>	<b>Algunas funciones matemáticas</b>	<b>94</b>
9.1.	Distintas opciones . . . . .	94
9.2.	Sólo enteros . . . . .	95
9.2.1.	Factoriales y combinaciones . . . . .	95
9.2.2.	Raíz cuadrada . . . . .	96
9.2.3.	Seno, coseno y tangente . . . . .	98
9.2.4.	Exponencial . . . . .	99
9.3.	Números reales . . . . .	100
<b>10.</b>	<b>Lézar: en busca del espíritu</b>	<b>103</b>
10.1.	La cuestión . . . . .	103
10.2.	El nombre . . . . .	103
10.3.	Características generales de Lézar . . . . .	104
10.4.	El diccionario . . . . .	105
10.5.	El código y su ejecución . . . . .	106

10.6. Palabras intrínsecas . . . . .	108
10.7. Sistema de archivos . . . . .	110
10.8. Extensiones del núcleo . . . . .	112
10.9. Compilación de algunas palabras . . . . .	123
<b>11. Miscelánea de Forth</b>	<b>128</b>
11.1. Historia de Forth . . . . .	128
11.2. PostScript y JOY . . . . .	132
11.2.1. PostScript . . . . .	132
11.2.2. JOY . . . . .	133
11.3. El panorama en 2006 . . . . .	136
11.4. Referencias comentadas . . . . .	136
11.5. Palabras finales . . . . .	138
<b>12. Ejercicios</b>	<b>139</b>
12.1. Introducción . . . . .	139
12.2. Ejercicios resueltos . . . . .	140

# Capítulo 1

## Introducción a Forth

### 1.1. Una filosofía distinta

Este es un libro sobre el lenguaje de programación Forth. Un lenguaje desconocido por la mayoría de los estudiantes y profesionales a pesar de que su historia arranca a finales de los años 60. Un lenguaje de nicho, limitado en la práctica a programación de microcontroladores, pero a la vez el precursor de la programación estructurada, de la programación orientada a objetos, de las implementaciones de máquina virtual. Un lenguaje que encarna una filosofía radicalmente distinta, que se aparta de los caminos trillados, de las tendencias actuales que conducen a muchos lenguajes a ser meros bastidores en los que apoyar librerías pesadas y complejas.

Pero no nos engañemos, Forth tuvo su oportunidad, su momento dulce alrededor de 1980, y no prosperó. Esa es la realidad. Entonces, ¿por qué Forth? La respuesta es bien sencilla: porque es divertido, gratificante; porque enseña cosas que los demás no enseñan. Quien menosprecie sus lecciones sólo porque el azar ha llevado a Forth a ejecutarse en sistemas empotrados en lugar de en computadores personales comete un error y se priva a sí mismo de muchas horas gratificantes, y de lecciones sobre programación que pueden aplicarse a cualquier otro lenguaje.

No valdría la pena acercarse a Forth para terminar traduciendo C a Forth, o php a Forth, o ensamblador a Forth. Primero, porque ya existen C, php y ensamblador, y funcionan. Segundo, porque el objeto de Forth, al menos desde el punto de vista del libro que tiene en sus manos, no es escribir código Forth, sino *pensar en Forth*. Es legítima la pregunta de qué es pensar en Forth. Valga el siguiente ejemplo. Imaginemos un fragmento de código, por

ejemplo en C, que ha de tomar un argumento numérico y efectuar, según su valor, una operación. Pero este argumento ha de estar comprendido entre, digamos, los valores 0 y 5; de forma que, si el argumento es menor que cero, se sustituye por 0, y si es mayor que 5, se sustituye por 5. Escribiríamos en C:

```
if (n<0){
    n=0;
} else
if (n>5){
    n=5;
}
```

Ciertamente, hay formas mejores de hacerlo, aún en C, pero esta es, probablemente, la que elegirá la mayoría de los programadores. Esta es la versión Forth (de momento, es irrelevante si no sabe la forma de interpretarla):

```
0 max 5 min
```

En su sorprendente concisión, esta línea de código exhibe la esencia del problema. El programador Forth busca la esencia del problema, piensa, y después escribe. El buen código Forth apela a un sentido estético particular que se deleita con las cosas sencillas. Espero poder transmitir al lector parte de ese gusto estético.

## 1.2. Un entorno distinto

El programador corriente cuenta con una serie de herramientas para su trabajo: un editor, un compilador, un depurador, ensamblador y desensamblador y algunas herramientas adicionales. Cada una de estas herramientas es a su vez un programa que se ejecuta sobre un sistema operativo. En la filosofía UNIX, estas herramientas son pequeñas y especializadas, y pueden comunicarse entre sí compartiendo corrientes de texto. En la filosofía Windows, todas se encuentran integradas en un entorno de desarrollo que puede ser muy, muy grande.

El panorama para el programador Forth es totalmente distinto. En su forma más sencilla, un sistema Forth es sólo un programa que se ejecuta sobre un sistema operativo. En su forma más genuina, un sistema Forth *es el sistema operativo*, e incluye desde los controladores de dispositivo básicos hasta

una forma de memoria virtual, compilador, ensamblador y desensamblador, editor y depurador. Pero este entorno, completamente funcional, es a la vez extraordinariamente simple, de forma que un sistema operativo Forth puede ocupar sólo unas pocas decenas de kilobytes.

En estos tiempos es difícil de creer que pueda construirse un entorno de desarrollo en, digamos, treinta kilobytes. Pero si el lector se sorprende de ello puede tomar esa sorpresa como la prueba de que los derroteros de la programación actual son cuando menos discutibles. ¿Quién puede entender a nivel de *bit* el funcionamiento de un compilador tradicional? Un compilador Forth está escrito en Forth, y ocupa unas diez líneas de código <sup>1</sup>.

Otro de los aspectos que pueden sorprender al recién llegado a Forth es la forma transparente en que se integran un intérprete y un compilador. En la corriente tradicional, los lenguajes se implementan como intérpretes o como compiladores; en este último caso, sabemos que es preciso editar y guardar un programa, compilarlo, ejecutarlo y volver al editor para depurar los errores. Forth es un entorno interactivo que ejecuta ininterrumpidamente un bucle. En ese bucle, el programador introduce órdenes y el intérprete las ejecuta. Pero esas órdenes pueden ser de compilación, y de esta forma el programador puede codificar una función en una línea de código y el sistema la compilará de forma instantánea. Ahora, esa función es una extensión del lenguaje, y puede usarse como si fuese parte del mismo.

Forth es extensible. Está diseñado para eso, y esa es la razón por la que sobrevive. Por ejemplo, Forth carece de un tipo de cadenas de caracteres. En su lugar, el programador puede elegir el tipo de cadena que desee, según la aplicación, e implementar el nuevo tipo en un par de líneas de código. Forth carece de estructuras como C, pero si la aplicación lo requiere el lenguaje puede extenderse sin dificultad para incluirlas. Forth incluye una serie de estructuras de control, como bucles y condicionales. Pero, al contrario que los lenguajes tradicionales, estas estructuras no son parte del lenguaje, sino que *están escritas en Forth*, y por esa razón el programador puede extenderlo con otras estructuras *ad hoc*.

Este último punto es muy interesante. Forth está, en su mayor parte escrito en Forth. Cuando veamos la forma en que Forth puede extenderse, comprobaremos como el lenguaje es a su vez una extensión de un núcleo pequeño

---

<sup>1</sup>Véase *Starting Forth*, de Leo Brodie

que generalmente se escribe en el lenguaje ensamblador del procesador sobre el que se ejecutará el sistema. Lo reducido de este núcleo llevó en un momento a pensar en la posibilidad de diseñar un procesador cuyas instrucciones nativas fuesen justamente las primitivas sobre las que Forth se construye. De esta forma podría disponerse de un procesador que ejecutase de forma nativa un lenguaje de alto nivel. El primer microprocesador con estas características fue lanzado en 1985 por Charles Moore, el creador de Forth.

## 1.3. La máquina virtual

Aprender este lenguaje es a su vez aprender la arquitectura y funcionamiento de un sistema. Este sistema se basa en un modelo de máquina virtual extremadamente sencillo, que consta de tres elementos principales: dos pilas y un diccionario. La primera pila sirve esencialmente para efectuar operaciones y evaluar expresiones. Nos referiremos a ella como la pila de parámetros. La segunda, para guardar direcciones de retorno. El diccionario guarda las extensiones del lenguaje. Pero *lenguaje* en este contexto tiene un significado muy concreto: una colección de *palabras*. Forth es un conjunto de palabras, y cada una de ellas tiene un efecto definido sobre la máquina virtual. Puesto que cada palabra, por sí misma, opera sobre la máquina virtual, el lenguaje realmente carece de sintaxis. Así que la explicación sobre cuestiones sintácticas, que ocupa buena parte de los manuales al uso, aquí es brevísima: no hay.

### 1.3.1. La pila de parámetros

Iniciemos una sesión Forth. Una vez puesto en marcha el sistema, ya sea como sistema autónomo o como programa ejecutándose sobre un sistema operativo, el programador recibe la invitación de un inductor para que introduzca una línea, y después pulse `intro`. Este proceso se repite indefinidamente, de forma que podría describirse en pseudocódigo como:

```
while(1)
{
    aceptar cadena desde teclado
    interpretar cadena
}
```

A su vez, el intérprete de cadenas se describe como sigue:

```

while ( no fin de cadena )
{
    extraer siguiente palabra

    Si la palabra representa un numero,
    llevarlo a la pila de parametros

    Si la palabra representa un operador, ejecutarlo
}

```

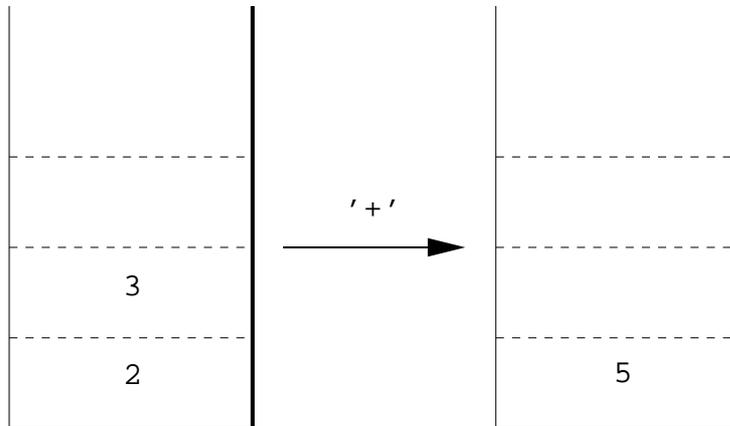
Esta descripción es aún pobre, pero sirve a nuestros propósitos. Ahora traslademos esta mecánica a una corta sesión. Mediante el carácter \$ indicamos la pulsación de `intro` por parte del usuario, indicando el fin de su entrada.

```
2 3 + . $ 5 ok
```

Cuando el intérprete se hace cargo de la línea, en primer lugar identifica a '2' como un número, y lo deja en la pila de parámetros (simplemente pila, en lo sucesivo). A continuación identifica a '3' como un número, y por tanto lo deja también en la pila. Después, encuentra la palabra '+', que identifica como operador. Este operador toma sus argumentos de la pila, y deja en la misma el resultado de su operación, que en este caso es el número 5. Finalmente, el intérprete encuentra la palabra '.', que se emplea para imprimir un número. Esta palabra toma su argumento de la pila, el número 5, y hace su trabajo: imprimirlo en pantalla. Al terminar, el intérprete emite un lacónico `ok` indicando que todo fue bien, y espera la entrada de la siguiente cadena.

Esta sesión nos enseña además que las expresiones en Forth se escriben en notación postfija. No hemos introducido '2+3', sino '2 3 +'. Los operandos primero, los operadores después. Las expresiones postfijas son más compactas, porque eliminan la necesidad de paréntesis, y son mucho más fáciles de evaluar por un programa. A los usuarios de las calculadoras HP les resultará familiar esta notación. Nótese también como las palabras en Forth deben quedar delimitadas por espacios en blanco: escribimos '2 3 +', no '23+' ni '2 3+'.

Pero, volviendo a la arquitectura del sistema más que a su representación para el usuario, notemos como el uso de una pila para la evaluación de expresiones tiene otras ventajas. Los operadores, y en general las palabras (funciones en la terminología de los lenguajes convencionales), toman sus



**Figura 1.1** Representación simbólica de la pila, y efecto sobre ella de la palabra '+' cuando contiene dos números.

argumentos de la pila, y dejan sus resultados también en la pila. Tenemos por tanto un sencillo protocolo de paso de parámetros, y la posibilidad de que una palabra deje como resultado un número arbitrario de parámetros. Por consiguiente, una palabra recién creada por el programador no necesita ser integrada en un programa para comprobar su funcionamiento y eventualmente ser depurada. Una palabra recién creada en Forth tiene acceso a la pila, toma de ella sus argumentos y deja en ella el o los resultados, de forma autónoma. Esto a su vez elimina la necesidad de declarar variables locales.

Existen tres primitivas para la manipulación de la pila. Estas tres palabras básicas son 'dup', 'drop' y 'swap'. La primera, duplica el último elemento de la pila. La segunda, elimina el último elemento de la pila. La tercera, intercambia los dos últimos elementos de la pila. Es útil introducir aquí la palabra '.s', que imprime los argumentos que contenga la pila, sin alterarlos. La usaremos para comprobar el funcionamiento de las primitivas introducidas:

```
1 2 .s $ <2> 1 2 ok
```

La salida de '.s' indica en primer lugar el número de elementos contenidos en la pila, y después los imprime desde el primero que fue introducido hasta el último.

```
1 2 .s $ <2> 1 2 ok
swap .s $ <2> 2 1 ok
dup .s $ <3> 2 1 1 ok
```

```
drop .s $ <2> 2 1 ok
drop .s $ <1> 2 ok
```

Dejaremos para más adelante la discusión sobre la segunda pila. Únicamente presentaremos dos nuevas palabras: '>r' y 'r>'. La primera, permite pasar un valor desde la pila de parámetros a la pila de retorno. La segunda, pasar un valor desde la pila de retorno a la pila de parámetros.

### 1.3.2. Crear nuevas palabras

Extender Forth es muy sencillo. Lo ilustraremos con un ejemplo trivial: una palabra que llamaremos 'cuadrado'. Esta palabra toma un número de la pila y deja como resultado el cuadrado de ese número.

```
: cuadrado dup * ; $
ok
```

Ahora, podemos usarla:

```
12 cuadrado . $ 144 ok
```

Lo que ha ocurrido ha sido que la palabra ':' ha activado el compilador, que ha creado una nueva entrada en el diccionario, de nombre 'cuadrado'. Esta palabra tiene un código asociado, en este caso 'dup \*'. Finalmente, la palabra ';' conmuta a modo de interpretación. Más adelante daremos detalles sobre la estructura del diccionario. De momento, vemos dos características: cómo se amplía con nuevas palabras y cómo cada nueva palabra se apoya en palabras anteriores. Cuando el intérprete encuentra una palabra que no representa un número, busca en el diccionario, que habitualmente se implementa como una lista enlazada. De momento, es irrelevante el mecanismo por el cual se ejecuta el código propio de cada palabra. El punto clave aquí es que ese código contendrá aparte de valores literales referencias a otras palabras. Los detalles dependen de la implementación.

Cerraremos la sección y el capítulo actuales con un ejemplo real: un programa que controla una lavadora industrial<sup>2</sup>. Aunque aparecen palabras que aún no hemos presentado, servirá para ilustrar algunas características generales.

---

<sup>2</sup>Adaptación del que aparece en "Forth programmer's handbook", de Edward K. Conklin & Elizabeth D. Rather, equipo técnico de **Forth Inc.**; disponible en Internet.

```

1 HEX
2 7000 CONSTANT MOTOR
3 7006 CONSTANT DETERGENTE
4 700A CONSTANT INTERRUPTOR
5 7002 CONSTANT VALVULA
6 7008 CONSTANT RELOJ
7 7010 CONSTANT LLENO
8 7004 CONSTANT GRIFO
9 DECIMAL
10 : ON ( PUERTO) -1 SWAP OUTPUT ;
11 : OFF ( PUERTO) 0 SWAP OUTPUT ;
12 : SEGUNDOS ( N) 1000 * MS ;
13 : MINUTOS ( N) 60 * SEGUNDOS ;
14 : AGREGAR ( PORT) DUP ON 10 SEGUNDOS OFF ;
15 : ESPERA ( PORT) BEGIN DUP INPUT UNTIL DROP ;
16 : VACIAR VALVULA ON 3 MINUTOS VALVULA OFF ;
17 : AGITAR MOTOR ON 10 MINUTOS MOTOR OFF ;
18 : CENTRIFUGAR INTERRUPTOR ON MOTOR ON
19 10 MINUTOS MOTOR OFF INTERRUPTOR OFF ;
20 : RELLENAR GRIFO ON LLENO ESPERA GRIFO OFF ;
21 : LAVAR RELLENAR DETERGENTE AGREGAR AGITAR VACIAR ;
22 : ENJUAGAR RELLENAR AGITAR VACIAR ;
23 : LAVADORA LAVAR CENTRIFUGAR ENJUAGAR CENTRIFUGAR ;

```

El programa comienza definiendo una serie de constantes que serán necesarias después, y a continuación procede creando nuevas palabras a partir de palabras preexistentes. Cada nueva palabra es una pequeña frase, y la última de ellas es la aplicación, que expresa de forma natural la operación de la lavadora, como es evidente leyendo la línea 22 del programa.

Podemos seguir el camino inverso para averiguar la forma en que el programa funciona. Por ejemplo, ¿qué hace la palabra 'ENJUAGAR'? La respuesta, en la línea 22: RELLENAR AGITAR VACIAR. ¿Cómo funciona la palabra 'VACIAR'? La línea dieciseis nos dice que primero se abre una válvula, se espera durante tres minutos y se cierra la válvula. ¿Cómo se abre una válvula? Este extremo, es ya interacción con el hardware. La constante 'VALVULA' tiene asignado el valor de un puerto, y la palabra 'ON' escribe ese valor en el puerto.

Este sencillo ejemplo en definitiva nos enseña que programar en Forth es crear un vocabulario específico que se ajuste al problema que hemos de re-

solver. En el paso final, una frase expresa con naturalidad qué es lo que deseamos hacer. Como puede comprobarse, Forth es un lenguaje extremadamente legible. Cuando se le acusa de lo contrario, se toma como ejemplo código que resulta de la traducción de C a Forth, pero ya dijimos al inicio de este capítulo que C traducido a Forth no es Forth. Los programas escritos en este lenguaje deben ser *pensados* en él desde el principio. Entonces, el resultado es conciso, elegante y legible. Como beneficio adicional, extremadamente compacto. El programa que hemos presentado queda compilado y listo para ejecutarse en unos 500 bytes.

Obsérvese además que las definiciones de las palabras tienen una extensión de una línea. En este sentido, éste es un programa típico. No se encuentran en Forth palabras de varias páginas, como sucede con las funciones que se suelen escribir en C, por poner un ejemplo. Otra característica que destaca a partir del código presentado es la facilidad extrema de depuración. Por una parte, ¿cuantos errores pueden esconderse en una sola línea de código? por otra, no es preciso tener una aplicación completa para probarla. Puede escribirse la palabra 'ON' y comprobar sobre el hardware que la válvula se abre. Una vez depurada esta palabra, puede pasarse a la siguiente.

# Capítulo 2

## Pila y aritmética

### 2.1. Vocabulario para la pila

En el capítulo anterior presentamos los operadores básicos de pila: 'drop', 'dup' y 'swap', junto con '>r' y 'r>'. Ahora usaremos estas palabras para definir otras que nos permitan usar la pila para evaluar cualquier expresión.

'rot' efectúa una rotación sobre los tres últimos elementos de la pila, y en función de las palabras básicas se puede expresar como

```
: rot ( a b c -- b c a ) >r swap r> swap ;
```

La cadena '( a b c -- b c a )' es un comentario de pila, que se inicia con la palabra '(' (como tal palabra, es esencial delimitarla mediante espacios en blanco); indica que, antes de la aplicación de 'rot' a la pila, ésta contenía los valores 'a b c', donde 'c' es el último elemento que se introdujo en la pila, y que después de la aplicación del operador la pila contiene 'b c a'.

'-rot' efectúa también una rotación, pero en sentido contrario:

```
: -rot ( a b c -- c a b ) rot rot ;
```

pero esta definición no es única, ya que también puede definirse como

```
: -rot ( a b c -- c a b ) swap >r swap r> ;
```

que contiene sólo cuatro instrucciones, mientras que la anterior contiene ocho. Por tanto, ésta es preferible.

La palabra 'over' copia sobre el elemento superior de la pila el segundo elemento:

```
: over ( a b -- a b a) >r dup r> swap ;
```

'nip' borra el segundo elemento en la pila:

```
: nip ( a b -- b) swap drop ;
```

y 'tuck' hace una copia del primer elemento bajo el segundo:

```
: tuck ( a b -- b a b) swap over ;
```

Finalmente, '2dup' hace una copia de los dos últimos elementos de la pila y '2swap' intercambia las dos primeras parejas de la pila. Por conveniencia, reunimos el vocabulario presentado, tal y como se escribe en Forth:

```
: rot ( a b c -- b c a) >r swap r> swap ;
: -rot ( a b c -- c a b) swap >r swap r> :
: over ( a b -- a b a) >r dup r> swap ;
: nip ( a b -- b) swap drop ;
: tuck ( a b -- b a b) swap over ;
: 2dup ( a b -- a b a b) over over ;
: 2swap ( a b c d -- c d a b) rot >r rot r> ;
```

## 2.2. Aritmética básica

### 2.2.1. Enteros simples

Conviene aquí introducir el concepto de *celda*. Una celda es un conjunto de bytes sucesivos en memoria cuyo tamaño coincide con el de los enteros simples en la representación del procesador. Por ejemplo, en un procesador de 32 bits, una celda es un conjunto de 32 bits sucesivos. Para Forth, la memoria es una sucesión de celdas, y por tanto el diccionario y las pilas son conjuntos de celdas.

Así, un entero simple ocupa una celda, al igual que un puntero. Un entero doble ocupa dos celdas, y así sucesivamente. Una vez establecido este punto, iniciemos la discusión sobre los operadores aritméticos.

El primer aspecto que es preciso recordar es que los operadores aritméticos en Forth no están sobrecargados. Existen operadores para trabajar con enteros simples, operadores para trabajar con enteros dobles y operadores que trabajan con operandos de tipos distintos. Desde el punto de vista de la simplicidad, esto no es una ventaja, pero, como todo lenguaje, Forth lleva implícitos una serie de compromisos en su diseño, y puesto que fue pensado para sacar el máximo provecho del hardware no es de extrañar que se adoptase un diseño que, forzoso es reconocerlo, no favorece ni a la simplicidad ni a la elegancia.

Los operadores básicos para trabajar con enteros simples son los de suma, resta, multiplicación, división y resto. Su operación es trivial:

```
2 3 + . $ 5 ok
8 1 - . $ 7 ok
3 3 * . $ 9 ok
6 3 / . $ 2 ok
5 3 / . $ 1 ok
5 3 mod $ 2 ok
```

y la penúltima línea deja claro que del operador de división entera sólo podemos esperar un resultado entero. Otros operadores frecuentes son el de negación, máximo y mínimo, valor absoluto, multiplicación y división por 2 y desplazamientos binarios a izquierda y derecha:

```
5 negate . $ -5 ok
10 2 max . $ 10 ok
2 10 min . $ 2 ok
-1 abs . $ 1 ok
4 2* . $ 8 ok
10 2/ . $ 5 ok
11 2/ . $ 5 ok
2 3 lshift . $ 16 ok
16 3 rshift . $ 2 ok
```

Existen también combinaciones útiles como `'/mod'`, que deja en la pila el resto y cociente de sus operandos:

```
9 3 /mod .s $ <2> 0 3 ok
```

¿Cómo implementaríamos este operador? Por ejemplo, mediante

```
: /mod 2dup / >r mod r> ;
```

Otro operador útil es `'*/'`, que toma tres argumentos de la pila y deja en la misma el resultado de multiplicar el tercero por el segundo y dividir el resultado por el primero:

```
10 2 6 */ . $ 3 ok
```

Parece inmediato definir esta palabra como

```
: */ ( a b c -- (a*b/c)) >r * r> / ;
```

pero esta definición es incorrecta, porque si `'a'` y `'b'` son enteros simples y ocupan una celda, su producto es un entero doble, que ocupa dos celdas, y por tanto dos posiciones en la pila, en lugar de una, de manera que el operador de división fallaría. Esto demuestra cómo programar en Forth exige conocer la representación interna que hace de los números y cómo son acomodados en la máquina virtual.

Incluimos también en esta sección los operadores lógicos binarios `'and'` y `'or'`, que operan en la forma esperada:

```
1 2 and . $ 0 ok
```

```
1 2 or . $ 3 ok
```

### 2.2.2. Enteros dobles

Como queda dicho, un entero doble, en un procesador de 32 bits, ocupa 64 bits, y por tanto ocupa dos celdas consecutivas en la pila. Los operadores que trabajan con números de longitud doble se identifican mediante la letra `'d'`. Pero ¿cómo introducir números dobles en el sistema? Mediante uno o varios `'.'` intercalados el intérprete reconoce los números de longitud doble:

```
200.000 d. $ 200000 ok
```

```
2.000.000 d. $ 2000000 ok
```

Las versiones para argumentos dobles de algunas funciones anteriormente presentadas en sus versiones de longitud simple se ilustran a continuación:

```
2.000 1.000.345 d+ d. $ 1002345 ok
1.000.345 2.000 d- d. $ 998345 ok
30.000 100.000 dmax d. $ 100000 ok
20.000 100.000 dmin d. $ 20000 ok
10.000 dnegate d. $ -10000 ok
```

### 2.2.3. Operadores mixtos

Los operadores mixtos toman como argumentos números de tipos distintos, simples y dobles; o bien toman números del mismo tipo pero dejan en la pila un resultado de tipo distinto. El más elemental es 'm+', que suma un entero doble y uno simple dejando en la pila el resultado, de longitud doble:

```
200.000 7 m+ d. $ 200007 ok
```

No existe 'm-', pero tampoco cuesta nada definirlo:

```
: m- negate m+ ;
```

'm\*' multiplica dos enteros simples, dejando en la pila un entero doble:

```
23 789 m* d. $ 18147 ok
```

Algo más exótico es el operador 'm\*/': toma de la pila un entero doble y lo multiplica por un entero simple, dando como resultado un entero triple que después se divide por un entero simple para dar, finalmente, un entero doble que queda en la pila. Esta descripción ya es algo más larga de lo que sería conveniente, y por eso se hace necesario introducir una notación descriptiva que indique no sólo el número de argumentos en la pila, sino también su tipo. La convención que adoptaremos a partir de este punto es la siguiente: un número simple se representa por 'n'; un número doble por 'd'. Más adelante, introduciremos otros convenios para indicar caracteres, punteros, etc. Así, el contenido de la pila que espera 'm\*/' lo representamos por 'd n1 n2', donde 'n2' ocupa la celda de la cima de la pila, 'n1' la que hay debajo y 'd' las dos inferiores a ésta.

```
200.000 12335 76 m*/ d. $ 32460526 ok
```

El último operador mixto que presentamos en esta sección es 'fm/mod'. Toma un entero doble y lo divide entre un entero simple, dejando en la pila un resto simple y un cociente, también simple. El resultado no está definido si el divisor es cero, o si el resultado de la división es de tamaño doble.

#### 2.2.4. Números con signo y sin signo

Una celda de 32 bits puede representar bien un número binario sin signo en el rango  $[0, 2^{32})$  bien un número con signo en el rango  $[-2^{31}, 2^{31})$ . Existen operadores específicos para tratar con números sin signo. Citaremos aquí 'um\*' y 'um/mod'. El primero, multiplica dos números sin signo, dejando en la pila un entero doble sin signo. El segundo, divide un entero doble sin signo entre un entero sin signo, dejando en la pila un resto y un cociente, sin signo. En los comentarios de pila, usaremos 'u' para indicar números sin signo.

### 2.3. Salida numérica con formato

En páginas anteriores hemos presentado la más básica de las palabras relacionadas con la salida: '.'. Otra palabra útil es 'cr', que introduce en la salida un salto de línea y un retorno de carro. Véase por ejemplo

```
2 3 + . $ 5 ok
2 3 + . cr $ 5
ok
```

Relacionada con las anteriores está 'emit', que permite imprimir un carácter cuyo código fue depositado previamente en la pila:

```
65 emit $ A ok
```

De hecho,

```
: cr 10 emit ;
: bl 32 emit ; \ espacio en blanco
```

La palabra '.r' permite imprimir un número ajustándolo a un campo de anchura especificada; así, es útil para producir columnas numéricas:

```
23 4 .r    23 ok
23 6 .r    23 ok
23 9 .r    23 ok
-7 9 .r    -7 ok
```

Operador	Comentario de pila
+	( n1 n2 -- n-suma)
-	( n1 n2 -- n-resta)
*	( n1 n2 -- n-producto)
/	( n1 n2 -- n-cociente)
*/	( n1 n2 n3 -- n1*n2/n3)
mod	( n1 n2 -- n-resto)
/mod	( n1 n2 -- n-resto n-cociente)
max	( n1 n2 -- n-max)
min	( n1 n2 -- n-min)
abs	( n -- n-abs)
negate	( n -- -n)
2*	( n -- 2*n)
2/	( n -- 2/n)
lshift	( n u -- n<<u)
rshift	( n u -- n>>u)
d+	( d1 d2 -- d-suma)
d-	( d1 d2 -- d-resta)
dmax	( d1 d2 -- d-max)
dmin	( d1 d2 -- d-min)
dabs	( d --  d )
m+	( d n -- d-suma)
m*	( d n -- d-producto)
m*/	( d n1 n2 -- d-d*n1/n2)
fm/mod	( d n -- n-resto n-cociente)
um*	( u1 u2 -- ud)
fm/mod	( ud u1 -- u-resto u-cociente)
dup	( a -- a a)
drop	( a b -- a)
swap	( a b -- b a)
rot	( a b c -- b c a)
-rot	( a b c -- c a b)
nip	( a b -- b)
tuck	( a b -- b a b)
2dup	( a b -- a b a b)
2swap	( a b c d -- c d a b)

**Tabla 2.1** Operadores para enteros simples y dobles, operadores mixtos y números sin signo. Operadores de pila.

Pero fuera de estas necesidades básicas, el programador necesita imprimir fechas, como 2005/25/07; números de teléfono, como 951-33-45-60; códigos numéricos con separadores, como 12-233455-09 y por supuesto números enteros y reales, con signo y sin signo, de longitud simple o doble.

Aquí puede apreciarse la especial idiosincrasia de Forth: en lugar de proveer un minilenguaje para describir el formato de salida, como del que disponen las funciones `print()` de C o (`format` ) de Lisp, Forth usa un mecanismo totalmente distinto, pasmosamente sencillo, basado esencialmente en tres palabras: '<#', '#' y '#>'. La primera, inicia el proceso de conversión de un entero doble en la pila a una cadena de caracteres; la segunda, produce un dígito de esta cadena cada vez, y la tercera termina el proceso de conversión, dejando en la pila la dirección de la cadena creada y un contador con su longitud. Entonces, la palabra '`type`', que toma justamente estos argumentos, puede usarse para efectuar la impresión en pantalla. La cadena con la representación del número se genera, como decimos, dígito a dígito, de derecha a izquierda. Existe otra palabra, '`#s`' que genera todos los dígitos que falten en un momento dado para terminar la conversión.

```
1.000.000 <# #s #> type $ 1000000 ok
23 0 <# #s #> type $ 23 ok
23. <# #s #> type $ 23 ok
```

Compárense las líneas segunda y tercera. '<#' inicia la conversión de un entero doble, pero el número 23, así escrito, es interpretado como un entero simple, que ocupa una sola celda en la pila. Por ese motivo, es necesario extenderlo a la celda siguiente, con valor 0.

La utilidad de este procedimiento reside en que, en cualquier momento, es posible insertar en la cadena de salida el carácter que se desee. De esto se encarga la palabra '`hold`', que toma como argumento el código ASCII de un carácter. ¿Quién deja en la pila ese código? La palabra '`char`'. Véase por ejemplo:

```
1.000.000 <# # # # char , hold # # #
char , hold #s #> type $ 1,000,000 ok
```

Es necesario reparar por un momento en la frase '`char ,`', que parece violar la sintaxis postfija propia de Forth. La explicación es la siguiente: cada vez que el programador introduce una cadena para ponerla a disposición

del intérprete, esta cadena se almacena en un área conocida a la que todo programa tiene acceso; allí la cadena es dividida en palabras, tomando una cada vez, y actualizándose un puntero que marca el inicio de la palabra siguiente. Este puntero también está disponible para cualquier programa. De esta forma, 'char' puede obtener el carácter que se encuentra a continuación y depositar su código ASCII en la pila. Un segundo ejemplo para dar formato a una fecha:

```
20050626. <# # # char / hold # # char / hold #s
#> type $ 2005/06/26 ok
```

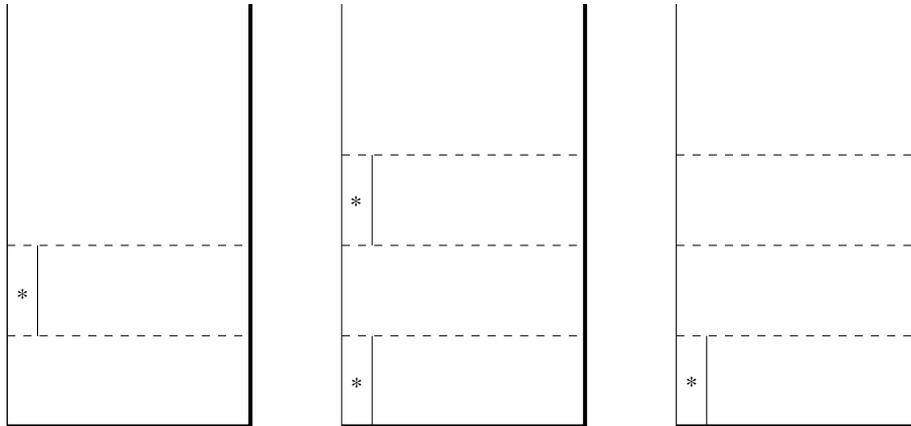
La discusión anterior es válida únicamente para enteros sin signo. Habrá advertido el lector cuando, en un ejemplo anterior, introducíamos un entero simple que era preciso extender a la siguiente celda de la pila colocando en ella un 0. Así que, bien cuando tenemos enteros simples sin signo, bien cuando tenemos enteros dobles sin signo, sabemos el procedimiento para efectuar una salida con formato. ¿Qué ocurre cuando los enteros tienen signo? Consideremos en primer lugar los enteros dobles. La palabra 'd.' es la encargada de imprimir un entero doble con signo. Pero esta palabra se escribe en Forth como sigue:

```
: d. tuck dabs <# #s rot sign #> type ;
```

En la Figura 2.1 se representan tres momentos de la pila. En el primero, contiene un entero doble con signo, que ocupa dos celdas. El bit más significativo indica el signo, y lo representamos mediante el asterisco. En primer lugar, se hace una copia de la celda más significativa mediante 'tuck'. A continuación 'dabs' toma el valor absoluto del número original, con lo que queda en la pila un entero doble sin signo apropiado para la conversión y un número negativo. Realizamos la conversión mediante la frase '<# #s' pero antes de terminar llevamos el entero negativo al tope de la pila mediante 'rot'; allí, la palabra 'sign' se encarga de comprobar el signo del número e insertar en caso de que sea negativo un carácter '-' en la cadena de salida. Finalmente, '#>' deja en la pila la dirección de la cadena resultante y su longitud.

Con esta discusión, es fácil escribir un procedimiento para imprimir un entero simple con signo:

```
: e.s dup abs 0 <# #s rot sign #> type ;
```



**Figura 2.1** Tres momentos de la pila en el proceso de conversión de un entero doble con signo.

## 2.4. La filosofía del punto fijo

Muchas de las implementaciones de Forth se ejecutan sobre procesadores que no soportan la aritmética con números reales. Para muchos programas, esto es irrelevante: un juego de ajedrez, un editor de textos, un controlador para una tarjeta de red ... ¿Y un programa de contabilidad? Según: si contamos en unidades monetarias, necesitaremos decimales para representar los céntimos. Pero si contamos con céntimos, todas las operaciones se pueden realizar con números enteros. A lo sumo, en el momento de imprimir las cantidades, deberíamos insertar un '.' que separe los dos últimos dígitos, pero esto ya sabemos como realizarlo. Como ejemplo, escribiremos una palabra, a la que llamaremos './.2', que imprime el resultado real, con dos decimales, de dividir dos enteros, que consideraremos simples y sin signo, para evitar distracciones.

```
: .2 0 <# # # [char] . hold #s #> type ;
: /.2 swap 100 * swap / .2 ;
125 67 /.2 $ 1.86 ok
1001 34 /.2 $ 29.44 ok
```

Obsérvese que la primera operación de './.2' consiste en multiplicar el dividendo por 100. De esta forma, los dos últimos dígitos del resultado de la división, que es un número entero, se corresponde con los dos decimales de la división real. La *conversión* a real no es tal, porque en la pila el número almacenado es un entero. Únicamente en el momento de la impresión este

entero se representa como real. En el ejemplo anterior, la operación 1001/34 en realidad se sustituye por la operación 100100/34, y la pila contiene después de realizarla el resultado entero 2944, que sabemos que es cien veces mayor que el real, debido a la multiplicación por 100 del dividendo; por eso, en el momento de la impresión se introduce un punto después de generar los dos primeros dígitos.

## 2.5. Números racionales

Los números racionales son otra alternativa para representar los números reales. Aunque existen números reales irracionales, como  $\pi$  o  $\sqrt{2}$ , ciertamente nunca trabajamos con ellos, sino con aproximaciones racionales. Por ejemplo, podemos tomar  $\pi = 3,1415$ , que no es más que  $31415/10000$ , que a su vez es  $6283/200$ <sup>1</sup>.

Respecto a la representación hardware de los números reales, y en particular a la representación binaria que adopta el IEEE e implementan todos los procesadores que soportan reales hoy en día, la representación mediante racionales tiene al menos dos ventajas: primero, que la operación de división es trivial, y en particular el cálculo de la inversa, ya que se reduce a intercambiar numerador y denominador; segundo, que no hay pérdida de cifras significativas. Dicho de otro modo,  $3 * \frac{1}{3} = 1$ , en lugar del resultado  $3 * \frac{1}{3} = 0,99999999$  que ofrece cualquier calculadora.

Interesa, siempre que se trabaje con racionales, tomar la fracción canónica de cada uno de ellos, que resulta de la división tanto del numerador como del denominador entre el máximo común divisor a ambos. Por ejemplo, en lugar de  $85/136$  preferimos  $5/8$ .

Forth carece de soporte para racionales, pero es natural pensar que un racional no son más que dos enteros depositados en la pila, primero el numerador y después el denominador. Así, es fácil escribir unas pocas palabras que extiendan Forth para operar con racionales:

```
\ palabras para operar con racionales
\ (n1 d1 n2 d2 -- n d)
: mcd ?dup if tuck mod recurse then ;
: reduce 2dup mcd tuck / >r / r> ;
: q+ rot 2dup * >r rot * -rot * + r> reduce ;
: q- swap negate swap q+ ;
: q* rot * >r * r> reduce ;
```

---

<sup>1</sup>Existen aproximaciones muy buenas de algunos irracionales significativos mediante fracciones. Por ejemplo,  $\pi$  se aproxima con seis decimales mediante la fracción  $355/113$ .

```
: q/ >r * swap r> * swap reduce ;
```

Todas estas palabras esperan en la pila cuatro números, numerador y denominador del primer racional, numerador y denominador del segundo. Las operaciones básicas son la suma, resta, multiplicación y división: 'q+', 'q-', 'q\*' y 'q/'. Estas efectúan sus correspondientes operaciones, y después reducen el resultado a la fracción canónica mediante la palabra 'reduce'. El algoritmo en que se basa esta es sencillo: si  $a$  y  $b$  son dos enteros y  $r$  es el resto de la división entera entre ambos, se satisface que  $mcd(a, b) = mcd(b, r)$ . Operando de forma iterativa alcanzaremos un punto en que el segundo argumento sea 0. Entonces, el primer argumento es el máximo común divisor. 'mcd' busca este máximo común divisor y 'reduce' simplemente divide por él numerador y denominador.

Quizás necesite el lector volver a este punto una vez que sean presentados los condicionales, que aparecen tanto en 'mcd' como en 'reduce'.

# Capítulo 3

## Programación estructurada

### 3.1. Operadores relacionales

La ejecución condicional es un elemento básico en cualquier lenguaje de programación. Una bifurcación en el flujo lineal de ejecución de un programa se produce atendiendo al resultado de una operación previa, que generalmente será la comparación entre dos números. Como es de esperar, los dos números que precisan compararse son depositados en la pila, y los operadores relacionales realizarán la comparación que corresponda y dejarán en la pila el resultado de esa comparación. Para Forth, *falso* se representa mediante un 0 (todos los bits a cero) y *verdadero* mediante un  $-1$  (todos los bits a uno).

Los operadores relacionales básicos son '=', '<' y '>'. Su funcionamiento se ilustra en las líneas siguientes:

```
1 2 = . $ 0 ok
1 2 > . $ 0 ok
1 2 < . $ -1 ok
```

### 3.2. Condicionales

Una vez que pueden efectuarse comparaciones, es posible usar el resultado de éstas para decidir si ejecutar o no ciertas porciones de código. La estructura que permite hacer esto es 'if...then'. 'if' toma un número de la pila y según sea *verdadero* o *falso*, en el sentido definido anteriormente, se ejecutará o no el código que se encuentra entre el 'if' y el 'then'. Si 'if' encuentra un valor *falso* en la pila, la ejecución proseguirá a continuación de 'then', y esto impone una limitación fundamental, y es que la construcción 'if...then'

sólo puede encontrarse contenida en la definición de una palabra, es decir, compilada. `'if ... then'` no puede usarse en modo interactivo, ya que es preciso conocer donde se encuentra `'then'` para saltar a continuación. Si Forth admitiese esta construcción en modo interactivo, tras un `'if'` que encuentra un valor falso en la pila, ¿donde saltar si el código siguiente puede que ni siquiera esté introducido aún? El otro aspecto a tener en cuenta, es que `'if'` consume su argumento, es decir, el valor encontrado en la pila y usado para decidir qué ejecutar, se elimina una vez tomada la decisión.

Es estrictamente innecesaria, pero útil, la estructura `'if ... else...then'`. Si la condición es cierta, se ejecuta la porción de código entre `'if'` y `'else'` y a continuación la ejecución continúa más allá de `'then'`; si la condición es falsa, se ejecuta la porción de código entre `'else'` y `'then'`.

Hemos presentado arriba los operadores relacionales básicos. Existen otros de uso muy frecuente, como son `'>='`, `'<='`, `'0='`, `'0>'` y `'0<'`. Naturalmente, estos operadores pueden escribirse en Forth como nuevas palabras. Podríamos, irreflexivamente, escribir por ejemplo:

```
: 0= 0 = if -1 else 0 then ;
```

`'0='` coloca en la pila un 0, y comprueba si son iguales el elemento anterior y el 0 recién apilado. Si son iguales se deja el resultado `-1` en la pila, y se deja 0 si son distintos. Lo que hay que notar es que el primer `'='` ya deja el resultado que necesitamos en la pila, por lo que el resto de la definición es innecesario. Es suficiente con

```
: 0= 0 = ;  
: 0< 0 < ;  
: 0> 0 > ;
```

La primera versión de `'0='` es la que escribiría la mayoría de los programadores habituados a C o Java, y eso demuestra, una vez más, como Forth y C no son directamente comparables, porque el código Forth es el resultado de una forma de pensamiento distinta.

### 3.3. Bucles

La estructura repetitiva básica en Forth es `'do...loop'`. `'do'` espera en la pila dos números, el límite superior del contador implícito en el bucle y el

valor de partida. Por defecto, el valor del contador se incrementa en una unidad por cada iteración, y el bucle termina cuando el contador alcanza el mismo valor que el límite que fue introducido en la pila. Al igual que 'if', 'do' sólo puede usarse, por la misma razón, dentro de una definición. Por otra parte, hemos dicho que un bucle tiene asociado implícitamente un contador. Este contador no necesita ser declarado: es creado por el sistema en tiempo de ejecución y puede accederse a él mediante la palabra 'I', que toma su valor y lo coloca en la pila.

```
: contar cr 6 0 do I . cr loop ; $
contar $
0
1
2
3
4
5
ok
```

Existe la palabra '?do' que no ejecuta el bucle en el caso de que el valor inicial coincida con el límite. Por ejemplo, pruebe el lector

```
: b1 cr 10 10 do I . cr loop ;
: b2 cr 10 10 ?do I . cr loop ;
```

Naturalmente, tanto los condicionales como los bucles pueden anidarse. En el caso de los primeros, no parece haber inconveniente; para los segundos, podemos preguntarnos como acceder al contador de cada uno de los bucles anidados. La respuesta de Forth no es quizás elegante, pero sí práctica. Al igual que existe la palabra 'I', existe la palabra 'J'. La primera accede al contador del bucle interno, la segunda al contador del bucle externo, como muestra el ejemplo <sup>1</sup>:

```
: anidar cr
  3 0 do
  3 0 do
  I 4 .r J 4 .r cr
```

---

<sup>1</sup>Para definiciones que ocupan más de una línea, es suficiente con teclear '\$' (tecla Intro) al final de cada línea. La palabra ':' pone al sistema en estado de compilación, y las sucesivas líneas son compiladas hasta que se encuentre la palabra ';', que finaliza la compilación y devuelve al sistema a modo de interpretación.

```

loop loop ;

anidar $
0 0
1 0
2 0
0 1
1 1
2 1
0 2
1 2
2 2
ok

```

No están previstos niveles más profundos de anidamiento, lo cual puede juzgarse como una carencia, pero en cualquier caso su importancia práctica es limitada, y siempre es posible declarar variables y usarlas como contadores para anidar bucles hasta la profundidad que se desee, como por otra parte es obligatorio hacer en casi cualquier otro lenguaje. Ahora bien, se considera más elegante prescindir de 'J' y factorizar el bucle interno:

```

: 4.r 4 .r ;
: interno 3 0 do I 4.r dup 4.r cr loop drop ;
: anidar 3 0 do I interno loop ;

```

Aquí, como en otros lugares, el programador tendrá que buscar un equilibrio: entre escribir código estilísticamente superior, según la forma de programar que promueve Forth, o escribir código estilísticamente menos satisfactorio pero más acorde con la nomenclatura natural en el área a la que pertenezca el problema. Por ejemplo, a un físico o ingeniero que programa operaciones con matrices le resultará muy difícil prescindir de 'I' y 'J' para indexar los elementos de una matriz.

Por su parte 'loop' tiene la variante '+loop', que permite incrementar el contador en la cantidad que se quiera:

```

: de-a-dos cr 10 1 do I . cr 2 +loop ; $
de-a-dos $
1
3
5

```

7  
9  
ok

### 3.4. Más bucles

La forma `'begin...again'` es la del bucle incondicional, que se repite una y otra vez. No es una estructura muy útil, a menos que dispongamos de un procedimiento para abandonar el bucle cuando se cumpla una condición determinada. Más adelante nos ocuparemos de este asunto.

La forma `'begin...until'` difiere de `'do...loop'` en que el número de iteraciones está indefinido. En lugar de mantener un contador que se incrementa en cada ciclo y un límite con el que se compara, el bucle se ejecuta una y otra vez, comprobando en cada una de ellas si el valor que `'until'` encuentra en la pila es *verdadero* o *falso*. El siguiente fragmento de código define una palabra llamada `'bucle'`, que acepta pulsaciones de tecla y termina con `'a'`.

```
: bucle begin key dup . 97 = until ;
```

Es claro en este ejemplo que la palabra `'key'` espera una pulsación de teclado, y deja en la pila el código de la tecla pulsada. En nuestro caso, el bucle termina cuando se pulsa la tecla `intro`, cuyo código es el 13.

Finalmente, existe la variante `'begin...while...repeat'`. Esta forma ejecuta una serie de operaciones entre `'begin'` y `'while'`. Como resultado de estas operaciones queda un valor *verdadero* o *falso* en la pila. Si el valor es *verdadero*, se ejecuta el código entre `'while'` y `'repeat'`. En caso contrario, se abandona el bucle saltando a continuación de `'repeat'`. Mientras que los bucles `'do...loop'` y `'begin...until'` se ejecutan al menos una vez, `'begin...while...repeat'` puede no ejecutarse ninguna. Para ilustrar esta estructura, hemos escrito la palabra `dig`. Esta palabra ejecuta un bucle que identifica entradas desde teclado que sean dígitos y termina cuando se pulsa una tecla que no lo es.

```
: in ( a b c -- ?a<=b<c ) over > -rot <= and ;  
: dig? 48 58 rot swap in ;  
: dig begin  
  key dig?  
  while
```

```
. " digito! " cr
repeat ;
```

### 3.5. Fin abrupto de un bucle

Como apuntamos en la sección anterior al presentar la estructura `'begin... again'`, su utilidad depende de la posibilidad de abandonar el bucle cuando se cumpla una determinada condición, excepto en aquellos casos en que sea esencial la ejecución ininterrumpida de dicho bucle. Por ejemplo, un sistema operativo Forth, desde el más alto nivel, es un bucle llamado `'quit'`. Este bucle lee cadenas desde teclado o disco y las interpreta o compila. La ruptura de `'quit'` equivale a la detención del sistema, que quedaría en un estado indefinido.

Existen algunas palabras que permiten abandonar un bucle de forma inmediata. `'leave'` abandona el bucle actual, y se usa en combinación con una prueba. La estructura típica es

```
do ... (?) if leave then ... loop
```

Nótese que una cosa es abandonar el bucle actual y otra la rutina que se está ejecutando. `'leave'` sólo sale del bucle, pero no de la rutina. Para salir de ambos es precisa la secuencia `'unloop exit'`. Pueden usarse varios `'unloop'` para salir de varios bucles anidados:

```
do ... do (?) if unloop unloop exit then ... loop ... loop
```

### 3.6. El estilo Forth

Aunque este capítulo ha sido titulado *Programación estructurada*, es preciso poner de manifiesto que existe una diferencia estilística esencial respecto a la corriente tradicional que comparten casi todos los lenguajes actuales, incluyendo a Pascal y sus descendientes, Java, C/C++, etc.

Para ver por qué esto es así, consideremos un programa sencillo que controla un cajero automático <sup>2</sup>. En el espíritu tradicional, el programador anida una serie de condicionales, descartando los casos que impiden realizar una

---

<sup>2</sup>La discusión que sigue está basada en el capítulo *minimizing control structures* de *Thinking Forth*, edición abierta del año 2004 del clásico libro de Leo Brodie

operación. Así, en primer lugar se comprueba que la tarjeta es válida. Si no lo es, se emite un mensaje de error y se devuelve la tarjeta. Si lo es, se comprueba que el usuario de la tarjeta es su propietario, pidiendo un número secreto. Si el número que introduce el usuario es incorrecto (pueden darse varias oportunidades) el cajero retiene la tarjeta y da por terminada la operación. Si lo es, se pide al usuario que introduzca un código de operación (los códigos pueden estar ligados a determinadas teclas del terminal)... Esta descripción se codificará de forma parecida a ésta:

```
if (tarjeta-valida) {
    if (propietario-valido) {
        solicitar-codigo
        if (codigo==EXTRAER){
            solicitar-cantidad
            if (cantidad < saldo){
                emitir cantidad
                actualizar saldo
                mensaje
            }
            else
            {
                mensaje
            }
        }
        else
        {
            ...otras operaciones
        }
    }
    else
    {
        retener tarjeta
        mensaje
    }
} else
{
    mensaje
}
```

¿Qué tiene de malo el fragmento de código anterior? Al fin y al cabo, ¿No es así como se hace? Hay algunas cosas inconvenientes. Primero, que

la legibilidad del código depende de su correcta indentación. Segundo, que esta legibilidad disminuye a medida que crece el número de condiciones que es preciso considerar. Tercero, que la acción que se ejecuta cuando una condición no es válida está codificada lejos de esa condición (puede estar de hecho *mu*y lejos). Una aproximación más cercana al estilo Forth consiste en ocultar los condicionales en el interior de procedimientos con nombre:

```
procedure comprobar-tarjeta
  if (tarjeta-valida){
    comprobar-propietario
  }
  else
  {
    mensaje
  }
end
procedure comprobar-propietario
  if (propietario-valido){
    solicitar-codigo
  }
  else
  {
    retener tarjeta
    mensaje
  }
end
procedure solicitar-codigo
  solicitar-codigo
  if (codigo==EXTRAER){
    extraer-dinero
  }
  else
  {
    ... otras operaciones
  }
end
...
```

De esta forma, el programa queda más legible:

```
program
```

```
    comprobar-tarjeta
end
```

Queda sin embargo un inconveniente: que el diseño de cada procedimiento depende del punto en que sea llamado dentro del programa, lo que hará difícil modificarlo en el futuro. Esto es debido a que cada procedimiento se encarga de realizar la llamada al siguiente. La aproximación de Forth a este problema es parecida, pero elimina este inconveniente. La palabra de alto nivel que resume la aplicación tendrá el siguiente aspecto:

```
: operar
    comprobar-tarjeta
    comprobar-propietario
    solicitar-codigo
    ...
;
```

y a su vez

```
: comprobar-tarjeta
    tarjeta-valida not if expulsar-tarjeta quit then ;
: comprobar-propietario
    propietario-valido not if mensaje quit then ;
...

```

Forth permite cualquiera de las tres aproximaciones, pero ésta última no sólo captura el espíritu del lenguaje, sino que además es mejor: más legible y más fácilmente modificable, ya que cada palabra opera de forma independiente de su contexto.

*Thinking Forth* es la fuente donde mejor se exponen los principios estilísticos de Forth. El lector interesado encontrará allí discusiones útiles sobre la programación estructurada desde el punto de vista de Forth. Bástenos aquí de momento poner de manifiesto dos puntos. Primero, que es posible escribir palabras cuya estructura consista en una serie de condicionales anidados. De esta forma, una palabra puede hacerse funcionalmente independiente de su contexto, evitando hacer explícitos los condicionales; pero por otra parte, el diccionario en sí mismo puede verse como una estructura condicional profundamente anidada que a su vez evita evaluar condiciones de forma explícita.

Piénsese por ejemplo en la coexistencia en el diccionario de las palabras '+' y 'd+': una palabra para una situación. Segundo, que existen muchas oportunidades para simplificar las expresiones condicionales, aportando además simplicidad al código. Para ilustrar este segundo punto, consideremos una sencilla frase donde la evaluación de una expresión deja en la pila un valor verdadero o falso; si el valor es verdadero, se deposita el valor 'n' en la pila, si falso, el valor 0. Llamando '(?)' a esta evaluación previa, escribiríamos

```
(?) if n else 0 then
```

Lo que queremos poner de manifiesto es que en Forth los booleanos se representan mediante números enteros, lo que permite sustituir la frase anterior por otra más sencilla:

```
(?) n and
```

que expresa exactamente lo que deseamos porque el entero que se asocia al booleano *verdadero* es el -1 (todos los bits a 1). Análogamente, consideremos la siguiente frase, que añade el valor 45 a 'n' según los valores relativos de 'a' y 'b':

```
n a b < if 45 + then
```

Si 'a<b', se suma 45 a 'n'. La idea que puede simplificar esta operación es que si 'a>b' no se suma nada, lo que equivale a sumar 0. Entonces la frase anterior es equivalente a esta otra:

```
n a b < 45 and +
```

Estos dos ejemplos comparten un mismo principio: no decidir aquello que se pueda calcular. En el capítulo siguiente presentaremos la ejecución vectorizada, y tendremos ocasión de volver sobre este punto, mostrando como en realidad las estructuras de decisión anidadas son innecesarias.

## Capítulo 4

# Constantes y variables. El diccionario

### 4.1. Constantes y variables

Hemos presentado ya los operadores de pila. Estos operadores son suficientes para calcular cualquier expresión aritmética. Véanse los ejemplos siguientes, donde 'a', 'b' y 'c' son números depositados previamente en la pila:

```
( a b -- (a+b)(a-b))    2dup + >r - r> *
( a b -- (a+b)^2(a-b))  2dup + dup * >r - r> *
( a b -- (a-1)(b+1))    1+ swap 1- *
( a b -- (a+5)/(b-6))   6 - swap 5 + swap /
( a b -- (a+9)(a-b^2))  2dup dup * - nip swap 9 + *
( a b c -- b^2-4*a*c))  swap dup * >r 4 * * r> swap -
```

La columna de la izquierda es un comentario que indica el contenido previo de la pila y el resultado que queda en ella. La columna de la derecha, la secuencia de operaciones que conducen al resultado deseado. Es un principio de la programación en Forth que la pila no contenga más de tres o cuatro elementos sobre los que operar. De lo contrario, se llega inmediatamente a código difícil de escribir y sobre todo difícil de leer. Así que, cuando no existe más remedio, conviene usar variables con nombre, al igual que hacen el resto de lenguajes de programación.

La forma más sencilla de una variable es lo que llamamos una constante, es decir, una variable de sólo lectura. Declarar la constante 'max' con el valor 20 es tan sencillo como escribir

```
20 constant KA
```

El valor almacenado en la constante 'KA' puede recuperarse simplemente escribiendo el nombre de la misma. Entonces, su valor se deposita en la pila y queda disponible para operar con él:

```
KA . $ 20 ok
12 constant KB $ ok
KA KB - . $ 8 ok
```

Al contrario que las constantes, las variables pueden leerse y escribirse. Cuando se escribe una variable, es preciso pasar el nuevo valor, que se encontrará en la pila, al lugar que tenga reservada la variable en la memoria del computador. Cuando se lee una variable, es preciso tomar el valor que tiene asignado y depositarlo en la pila. Para estas dos operaciones existen dos palabras: '!' y '@'. Como es de esperar, la palabra 'variable' es la encargada de crear una variable:

```
variable X $ ok
3 X ! $ ok
X @ . $ 3 ok
```

Las variables ayudan a escribir código legible, pero disminuyen el rendimiento, pues los valores han de tomarse de memoria y llevarlos a la pila y viceversa. Un código bien comentado puede eliminar muchas veces la necesidad de declarar variables, pero es preciso usar la pila con mesura, evitando que acumule muchos valores y evitando usarla como un vector de valores de acceso aleatorio <sup>1</sup>. El sentido común nos dirá cuando conviene y cuando no declarar variables. Ilustraremos esta discusión con dos versiones de la función que calcula iterativamente el factorial de un número. La primera usa exclusivamente el valor cuyo factorial quiere calcularse y que la función 'fct' ha de encontrar en la pila. La segunda almacena ese valor en una variable y después lo usa para calcular el factorial. En ambos casos, el resultado queda en la pila.

```
: fct ( n--n!) \ primera version
  1 over 0 do
    over * swap 1- swap
  loop ;
```

---

<sup>1</sup>Puede escribirse (y queda como ejercicio para el lector) una palabra que permita acceder a una posición cualquiera de la pila, dejando el valor que allí se encuentre en el tope de la misma. De hecho, muchos sistemas Forth tienen una palabra llamada 'pick' que hace exactamente eso.

```

variable n
: fct_2 ( n--n!) \ segunda version
  n ! n @ 1 1 -rot
  do
    n @ dup 1- n ! *
  loop ;

```

La primera versión consta de 10 palabras, y se ejecuta más velozmente que la segunda, que consta de 16 palabras. Por contra, esta segunda es más legible. En realidad, no era necesario declarar la variable 'n', puesto que el bucle ya declara de forma implícita una variable a cuyo valor se accede mediante la palabra 'I', de forma que podríamos haber escrito:

```

: fct_3 ( n--n!) \ tercera version
  1 swap 1+ 2 do I * loop ;

```

y en este caso está claro que la concisión y claridad compensan la falta de velocidad que pueda existir respecto a la primera versión.

No es mal momento este para presentar la técnica de la recursión, herramienta de que dispone Forth y que permite expresar de forma sencilla y natural, aunque no siempre eficientemente, muchos problemas de programación. Forth cuenta con la palabra 'recurse', que permite hacer una llamada a una palabra desde su propia definición. Ya que hemos presentado tres versiones para el cálculo del factorial, escribamos una cuarta que use recursión:

```

: fct_4 ( n--n!) \ cuarta version
  dup 2 > if dup 1- recurse * then ;

```

Esta versión no sólo es más compacta, sino que es mejor, ya que previene el caso en que el argumento para el factorial sea el número cero. Una versión descuidada de la función factorial, usando recursión, sería más compacta aún:

```

: fct_5 dup 1- recurse * ;

```

pero fallaría en el mismo punto en que fallan muchas implementaciones recursivas escritas por principiantes, a saber, ¿cuando terminar la recursión? Como segundo ejemplo, consideremos la función de Ackerman. Esta función toma dos argumentos enteros, y ya para valores bajos genera una enorme

cantidad de llamadas, por lo que se ha usado para evaluar las prestaciones de un lenguaje en lo tocante a la gestión de llamadas recursivas. La función de Ackerman se define de la siguiente forma:

$$ack(m, n) = \begin{cases} n + 1 & \text{si } m=0 \\ ack(m - 1, 1) & \text{si } n=0 \\ ack(m - 1, ack(m, n - 1)) & \text{en otro caso} \end{cases}$$

De la sola definición de la función, parece evidente que precisaremos más de una línea para codificarla con suficiente claridad. Conviene entonces usar un editor de textos, proporcionar al código un formato adecuado y después cargar dicho código desde el entorno Forth. Componemos entonces un documento ascii que llamamos `ack.f` cuyo contenido es el siguiente:

```

\ version: 1 4-11-2005
\ autor:   javier gil
\ ( m n --ack(m, n))
\ ack(m, n)= ack(m-1, 1)           si n=0
\           n+1                    si m=0
\           ack(m-1, ack(m, n-1)) en otro caso
\

dup 0=           \ comprobar n=0
if
  drop 1- 1 recurse \ ack(m-1, 1)
else
  swap dup 0=     \ comprobar m=0
  if
    drop 1+       \ eliminar m, dejar n+1
  else
    dup 1- swap rot 1- \ ( n m -- m-1 m n-1)
    recurse recurse
  then
then

```

Este documento puede cargarse mediante la palabra 'included':

```

s" ack.f" included $
3 8 ack . $ 2045 ok

```

En este punto, el lector puede colocar en un documento de texto las palabras aparecidas al final del capítulo 2, por ejemplo en `q.f`, y cargarlas cuando se desee extender Forth para operar con números racionales. Por ejemplo, para sumar  $7/11$  y  $38/119$  ( $7/11+38/119=1251/1309$ ):

```
s" q.f" included
7 11 38 119 q+ $ ok
```

y si queremos un mínimo formato para la salida

```
cr swap . cr ." ----" cr . cr $
1251
----
1309
ok
```

## 4.2. Inciso: cambio de base

Presentaremos una de las variables del sistema a las que tenemos acceso. Se trata de `'base'`, que es donde Forth almacena la base numérica para convertir cadenas (por ejemplo, introducidas desde teclado) en números y viceversa. Puede usarse como cualquier otra variable:

```
2 base ! $ ok
1001 0110 + . $ 1111 ok
10000 base ! $ ok
0a30 0cbf + . $ 15ff ok
0a base ! $ ok
```

En la primera línea, cambiamos a base 2. En la segunda, efectuamos una operación en esta base. En la tercera, cambiamos a base 16; pero cuidado, las entradas se efectúan en base 2, y en esa base el número 16 se representa como 10000. Sumamos dos cantidades en base 16 y volvemos a base 10. De nuevo, hemos de recordar la base en que nos encontramos, y que el número 10, en base 16, se escribe como 0a.

Esto sugiere la conveniencia de poder cambiar de base sin necesidad de teclearla en la base actual. Hay dos palabras que ayudan, `'hex'` para cambiar a hexadecimal y `'decimal'` para cambiar a base 10.

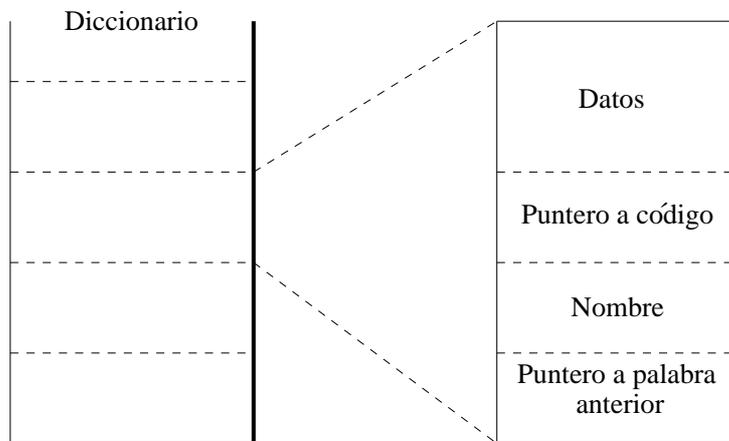


Figura 4.1 Estructura del diccionario.

A veces querremos saber en qu base estamos, y escribiremos

base @ .

Sin embargo, esto siempre produce  $10^2$  . He aqu un pequeño truco:

base @ 1- .

### 4.3. Estructura del diccionario

El diccionario es la estructura de datos fundamental de un sistema Forth. Allí es donde se encuentran las palabras, allí donde el intérprete las busca y allí donde las palabras recién creadas se guardan. Existen muchas formas distintas de implementar un diccionario, pero nosotros vamos a describir aquella que con la máxima simplicidad ofrece la funcionalidad que de él se espera.

Imaginémoslo como una lista enlazada de bloques donde cada bloque contiene la información pertinente a una palabra. Cada bloque contiene un puntero, no al elemento siguiente, sino al anterior. El puntero del primer bloque es nulo (**null**). De esta forma, cuando se precisa encontrar una palabra, se busca desde el último elemento que fue introducido en el diccionario hacia atrás. El sistema mantiene un puntero al último bloque, para saber en

---

<sup>2</sup>En efecto, en base 2, 2 es '10'. En base 10, 10 es '10'. En base 16, 16 es '10' y así sucesivamente

qué punto comenzar la búsqueda, y también para saber adonde tiene que apuntar el enlace del próximo bloque que sea introducido en el diccionario.

Además del enlace, un bloque ha de contener el nombre de la palabra. A su vez, el nombre es una cadena de caracteres de longitud variable, por lo que reservaremos el primer byte para guardar la longitud. Ahora bien, la longitud máxima razonable del nombre de una palabra puede estar en torno a los 30 caracteres, por lo que no precisamos usar todos los bits del byte de longitud. Digamos que en el byte de longitud dedicamos cuatro o cinco bits para guardar la longitud del nombre. Los tres o cuatro restantes pueden usarse para tareas de control de las que hablaremos más adelante.

Llegamos así a un punto sumamente interesante. Creo que no se ha puesto explícitamente de relieve que Forth, ya en 1969, fue el precursor de la programación orientada a objetos. En una forma primitiva, es cierto, pero funcional y potente. Y esto es así porque el bloque que contiene el nombre de una palabra y un enlace al bloque anterior contiene otros dos campos: un puntero al código que ha de ejecutarse cuando el intérprete encuentre la palabra y los datos contenidos en la misma. Para ver esto, consideremos tres objetos distintos, la forma en que se crean y la forma en que se comportan: una constante, una variable y una definición de una palabra compilada mediante `.`.

Una constante se crea en la forma

```
20 constant XX $ ok
```

En el momento de su creación, se introduce una nueva entrada en el diccionario con el nombre `XX` y en el campo de datos se almacena el valor 20. Después, cuando el intérprete encuentra en una línea la palabra `XX` ejecutará el código al que la palabra `constant` hizo apuntar el puntero a código de la palabra `XX` y que dice simplemente que *se tome el valor del campo de datos y se coloque en la pila*, como cuando escribimos:

```
XX . $ 20 ok
```

El comportamiento de las variables es distinto. Podemos declarar una variable y asignarle un valor:

```
variable ZZ $ ok  
12 ZZ !
```

Pero cuando después el intérprete encuentre 'ZZ' en una línea, no será el valor de la variable el que se deposite en la pila, sino su dirección. Es precisa la palabra '@' para recuperar el valor y colocarlo en la pila:

```
ZZ @ . $ 12 ok
```

De nuevo, esto es así porque la palabra 'variable', al crear la entrada en el diccionario de nombre 'ZZ' hizo apuntar el puntero a código de 'ZZ' a las instrucciones que indican que *ha de colocarse en la pila la dirección del campo de datos de 'ZZ'*.

Finalmente, consideremos una palabra 'YY' que fue definida mediante ':'. Esta palabra fue definida a partir de otras. Son otras palabras las que se encuentran entre ':' y ';'. El código asociado a la palabra 'YY' cuando fue creada consta de las direcciones de las palabras que forman su definición. Esto explica en parte por qué no nos preocupó demasiado que la estructura del diccionario no fuese especialmente sofisticada. Como una palabra se define en función de otras, compilar una palabra implica escribir sucesivamente en su definición las direcciones de las palabras que la constituyen. Así, cuando se encuentre la palabra 'YY', el código asociado dice *ejecútense sucesivamente las palabras que se encuentran en las direcciones almacenadas en la dirección indicada por el puntero a código que se encuentra en la definición*. Como esas direcciones ya apuntan directamente cada una a su palabra, no es preciso recorrer el diccionario buscando la definición de cada palabra contenida en aquella que se está ejecutando.

Así que podemos ver el diccionario como una secuencia de bloques, con un puntero que indica el lugar a partir de donde puede ampliarse con nuevas palabras. Pero la creación de una nueva palabra se realiza a partir de operaciones más elementales. De hecho, el bloque que contiene la definición de una palabra es a su vez un conjunto de celdas. Reservar espacio en el diccionario no es más que mover hacia adelante el puntero a la primera celda libre. La palabra 'allot' permite reservar celdas del diccionario, y la palabra ',,' compila un valor en el diccionario, es decir, lo toma de la pila y lo guarda en el diccionario, en la primera posición libre, avanzando una celda el puntero. Para terminar presentamos dos palabras adicionales: 'c,' y 'c@' que son las contrapartidas de ',,' y '@' para compilar y acceder a bytes individuales.

## 4.4. La pareja `create ...does`>

Se ha llamado a esta pareja *La perla de Forth*, y en esta sección veremos por qué. En los lenguajes convencionales, las constantes, variables, vectores, matrices y cadenas se declaran y se usan de una forma determinada y el lenguaje, por diseño, establece cómo van a almacenarse esos objetos y cuál será su comportamiento en tiempo de ejecución. Por ejemplo, en C las cadenas son una secuencia de bytes con un carácter de terminación al final, mientras que en Pascal el primer byte se reserva indicando la longitud total de la cadena. Pero en Forth, la forma en que se crean los objetos que se encuentran en el diccionario, y el comportamiento que tienen en tiempo de ejecución son programables. Esto abre interesantísimas posibilidades. Por ejemplo, imagínese que se está desarrollando un paquete de cálculo estadístico que hará uso frecuente de vectores de dimensión arbitraria. Sería interesante que el objeto vector contuviese un entero indicando sus dimensiones. Ese entero podría usarse para evitar lecturas y escrituras fuera del intervalo de direcciones ocupado por el vector, y para asegurarse de que las operaciones entre vectores, por ejemplo el producto escalar, se realizan entre vectores de igual dimensión. Deseos similares pueden formularse cuando se escribe código para manipular matrices. Forth permite extender el lenguaje y crear una nueva palabra, por ejemplo `'vector'`, para crear vectores con el formato que se ajuste a las necesidades del programa y con el comportamiento más adecuado según su naturaleza. Cuando estuviese implementada la palabra `'vector'` podría usarse depositando en la pila las dimensiones del vector que quiere crearse e indicando el nombre, como en el código hipotético

```
10 vector w
```

que crea una entrada en el diccionario con espacio para diez enteros y de nombre `w`. De hecho `'vector'` puede implementarse en una de línea de código, pero lo que deseo hacer notar ahora es que tanto `'vector'` como `'constant'` como `'variable'` no son en absoluto palabras especiales: se codifican, compilan y ejecutan como cualquier otra palabra, y como cualquier otra palabra tienen su entrada en el diccionario, con el mismo formato que el resto de palabras.

Pero empecemos por el principio. Las palabras `'constant'` y `'variable'` crean nuevas entradas en el diccionario. Tienen eso en común. Como Forth estimula la factorización del código y como Forth está escrito esencialmente en Forth, no es de extrañar que tanto una como la otra estén escritas a partir

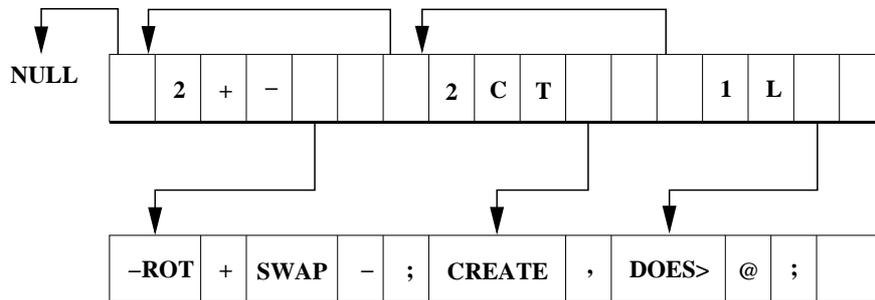


Figura 4.2 Estructura del diccionario.

de palabras de más bajo nivel ¡ y por tanto de funcionamiento aún más simple !.

Para fijar ideas, examinemos un fragmento de diccionario que contiene las palabras '+-' y una implementación de 'constant' que llamaremos 'ct'. Después de compilar ambas, la estructura del diccionario es la que se muestra en la Figura 4.2.

```
: +- ( a b c -- a+b-c) -rot + swap - ;
: ct create , does> @ ;
```

En realidad, la que se presenta es una de las muchas posibles implementaciones. Las entradas del diccionario, junto con los enlaces necesarios, ocupan un espacio de direcciones, mientras que el código asociado a cada palabra ocupa otro espacio. Suponemos que el sistema mantiene un puntero a la última palabra que fue introducida en el diccionario y otro a la primera posición libre. En un primer momento, el diccionario se encuentra vacío. Al crear la palabra '+-' el puntero a la palabra anterior ha de ser nulo, indicando así que '+-' es la primera entrada. A continuación se guarda la longitud de la palabra, en nuestro caso 2, el nombre de la palabra, '+-', el puntero al código asociado a esa palabra y finalmente los datos, que son un campo indeterminado. Entonces, se crea la palabra 'ct'. Como el sistema mantiene un puntero a la última palabra creada, '+-', es posible asignar el puntero de la nueva palabra a la palabra anterior. Después se almacena el byte de longitud, de nuevo 2, y un puntero al código asociado a 'ct' en su espacio de direcciones.

La palabra ':', que inicia la compilación de una palabra, activa también una bandera indicando el estado. En principio, mientras se compila una palabra

lo único que es preciso hacer es traducir cada palabra entre ':' y ';'. Más adelante precisaremos qué significa esta traducción.

Supongamos que ahora se desea crear una constante de nombre L con valor 11, para lo cual se escribe

```
11 ct L $ ok
```

¿Qué ocurre entonces? La pulsación de `intro`, que representamos mediante `$`, indica el final de la entrada del usuario. La cadena que ha sido introducida pasa a un área del sistema llamada TIB (de *Terminal Input Buffer*), y allí el intérprete aísla cada una de las palabras de que consta, y comienza el proceso de interpretación. La primera palabra se identifica como un número, y por tanto se coloca en la pila. La segunda palabra no es un número, y por tanto es preciso localizarla en el diccionario. Una vez localizada, se ejecuta el código que tiene asociado. Cuando se comienza a ejecutar `'ct'`, primero se encuentra `'create'`, cuyo funcionamiento es sencillo: toma del TIB la siguiente palabra, en este caso L, y crea una nueva entrada de diccionario. Asignará el puntero a la palabra anterior, el byte de longitud, almacenará la cadena "L" y reservará un par de celdas: una para el puntero al código de la palabra y otra para los datos. A continuación se ejecutará `','`. Esta palabra toma un valor de la pila, el 11 depositado anteriormente, lo inserta en el diccionario y actualiza el puntero a la siguiente celda libre. Es el turno de `'does>'`. La función de esta palabra es igualmente simple: toma el puntero que indica el código que se está ejecutando en ese momento, y que apunta a ella misma, y le da ese valor al puntero a código de la palabra que se está creando.

El sistema *sabe* que está en proceso de creación de una entrada en el diccionario, para lo cual mantiene una bandera. Entonces, termina la ejecución de `'ct'` en la palabra `'does>'`. Tiempo después el usuario escribe una expresión que contiene la palabra `'L'`.

```
L 2 + . $ 13 ok
```

De nuevo, la cadena introducida "L 2 + ." pasa el TIB. La primera palabra que encuentra el intérprete no es un número, la busca en el diccionario y comienza la ejecución del código asociado: `"does> @ ;"`. Ahora, la bandera que indica la creación de una palabra está desactivada y `'does>'` se interpreta como *colocar en la pila la dirección del campo de datos de la palabra que se está ejecutando*. A continuación `'@'` indica *tómese el valor contenido*

en la dirección que indica el número que hay en la pila y sustitúyase éste por aquel. La ejecución termina con ';' y como resultado el valor 11 queda depositado en la pila. El intérprete continúa con la cadena que se dejó en el TIB ejecutando '2 + .' ... Y esta es la historia de lo que sucede.

Si se crease otra constante, o un número cualquiera de ellas, los punteros a código de cada una apuntarían al mismo sitio: a la palabra 'does>' que se encuentra en el código de 'ct'.

Para terminar, hemos de advertir que la Figura 4.2 representa la estructura del diccionario en la implementación que acabamos de explicar, pero no la implementación misma. Es evidente que los caracteres ocupan un byte mientras que los punteros ocupan dos o cuatro. Por otra parte, en el espacio de código hemos colocado mnemónicos. Dependiendo de la implementación, ese espacio de código puede contener código nativo del procesador, o punteros a ese código, o *tokens* que indiquen la operación, o punteros al propio diccionario, como ocurrirá con las palabras implementadas en Forth, como '-rot'. En cualquier caso, el sistema necesitará almacenar las direcciones de retorno. Si la palabra X contiene varias palabras y una de ellas es la Y que a su vez contiene varias palabras, una de las cuales es la Z, al terminar de ejecutar Z el intérprete debe saber que ha de seguir con aquella que sigue a Z en Y; y cuando termine de ejecutar Y habrá de saber que la ejecución prosigue con la palabra que va después de Y en el código de X. Las direcciones de retorno se guardan, por supuesto, en la pila de retorno. Por eso, las palabras '>r' y 'r>' han de usarse con cuidado: toda operación '>r' dentro de una palabra ha de contar con una 'r>' antes de salir. Por la misma razón, '>r' y 'r>' han de estar anidadas rigurosamente con los bucles, que guardan también en la pila de retorno las direcciones de vuelta.

## 4.5. Aplicaciones

Como aplicación de las ideas expuestas en el párrafo anterior, escribiremos un pequeño vocabulario para operar sobre vectores. Deseamos una palabra que nos permita crear una variable de tipo vector, de dimensión arbitraria. El proceso de declaración de la variable concluirá con la creación del vector en el diccionario. La primera celda indicará sus dimensiones, y a continuación quedará el espacio reservado para el vector propiamente dicho. En cuanto al comportamiento en tiempo de ejecución, han de quedar en la pila la dirección del primer elemento y el número de elementos. Desde el punto de vista del programador, deseamos que pueda escribirse algo como:

```
10 vector t
```

para declarar un vector de nombre 't' y diez elementos. Véase la implementación de la palabra 'vector':

```
: vector create dup , cells allot
  does> dup 1 cells + swap @ ;
```

En primer lugar, el valor '10' queda en la pila. A continuación, el intérprete encuentra la palabra 'vector' y pasa a ejecutarla. Para ello, se ejecutarán por orden las palabras que lo componen. En primer lugar, 'create' tomará del 'TIB' la palabra siguiente, en este caso 't', y creará una entrada en el diccionario con ese nombre. A continuación hará copia del valor que se encuentra en la pila, el número '10', mediante 'dup'. La copia recién hecha será incorporada al diccionario, compilada, mediante ', '. Con la copia que queda, se reservará igual número de celdas mediante 'cells allot'. Finalmente, la palabra 'does>' establece el puntero a código de la palabra 't'. Este será el código que se ejecute cuando posteriormente 't' se encuentre en una frase.

Cuando esto sucede, 'does>' coloca en la pila la dirección de la primera celda del campo de datos. Esta primera celda contiene el número de elementos del vector. La dirección del primer elemento del vector será la de la celda siguiente. Esta dirección se consigue mediante '1 cells +'. 'swap' deja en la pila la dirección del primer elemento del vector y la dirección de la celda que contiene el número de elementos del vector. Este número se recupera mediante '@'.

Queremos ahora algunas funciones que nos permitan manipular el vector recién creado. En particular, necesitamos dos funciones básicas para escribir y leer un elemento determinado del vector. Como característica adicional, deseamos control sobre los límites del vector, de forma que no pueda accederse más allá del último elemento. Esta es la palabra que permite leer un elemento del vector:

```
: v@ ( n dir N --)
  1- rot min cells + @ ;
```

Primero, una explicación sobre el comentario de pila. Llamaremos 'n' al elemento al que deseamos acceder. 'dir' y 'N' son respectivamente la dirección del primer elemento y el número de elementos del vector. Estos dos últimos

valores han sido dejados en la pila por el código en tiempo de ejecución asociado a la variable de tipo vector que hemos creado previamente. De esta forma, podemos escribir frases como

```
4 t v@
```

para acceder al cuarto elemento del vector 't'. La frase '1- rot min' deja en la pila la dirección del primer elemento y el desplazamiento del elemento al que se quiere acceder. 'rot min' se asegura de que el índice no supere al desplazamiento máximo, que es 'N-1'. Finalmente, 'cells + @' incrementa la dirección base del vector y recupera el elemento contenido en la celda correspondiente, depositándolo en la pila.

En cuanto a la función de escritura:

```
: v! ( valor n dir N --)
  1- rot min cells + ! ;
```

espera en la pila el valor que se desea escribir, el elemento del vector en el que se desea guardar ese valor, la dirección del elemento base y el número de elementos del vector. Estos dos últimos parámetros, de nuevo, han sido dejados en la pila por el código en tiempo de ejecución de la variable de tipo vector. En definitiva, ahora podemos escribir frases como

```
10 2 t v!
```

para guardar el valor 10 en el tercer elemento (desplazamiento 2 respecto a la dirección base) del vector 't'.

Obsérvese que las palabras 'v!' y 'v@' son muy parecidas, lo que sugiere que pueden mejorarse las definiciones anteriores buscando una factorización. En efecto:

```
: v ( n dir N -- dir' ) 1- rot min cells + ;
: v! v ! ;
: v@ v @ ;
```

La comprensión de las funciones que siguen es fácil. En primer lugar, '0v!' pone el valor '0' en todos los elementos del vector:

```
: 0v! ( dir N --)
  0 do dup I cells + 0 swap ! loop drop ;
```

Una implementación más elegante de '0v!' es

```
: 0v! cells erase ;
```

de la cual se deduce sin dificultad cual pueda ser la operación de otra palabra que presentamos en este momento: 'erase'.

La palabra 'v.' imprime una columna con los elementos del vector:

```
: v. ( dir N --)
  cr 0 do dup I cells + @ . cr loop drop ;
```

Finalmente, pueden ser útiles palabras para extraer los elementos máximo o mínimo del vector y para obtener la suma de sus elementos:

```
: v-max ( dir N -- valor maximo)
  >r dup @ r> 1 do
  over I cells + @ max loop
  swap drop ;
: v-min ( dir N -- valor minimo)
  >r dup @ r> 1 do
  over I cells + @ min loop
  swap drop ;
: v-sum ( dir N -- sumatoria)
  0 tuck do over I cells + @ + loop swap drop ;
```

## 4.6. Ejecución vectorizada

Repárese que en las palabras 'v-max' y 'v-min' son idénticas, salvo porque la primera usa 'max' y la segunda 'min'. Sería deseable entonces que una palabra pudiese recibir como parámetro la dirección de otra, y que pudiese ejecutarse el código asociado a una palabra cuya dirección se encuentra en la pila. Ambas cosas pueden hacerse. En primer lugar, existe la palabra '' que toma la dirección del código asociado a una palabra y la coloca en la pila. En segundo lugar, la palabra 'execute' es capaz de iniciar la ejecución del código indicado por una dirección que se encuentre en la pila. Estúdiese el siguiente código:

```
: cubo dup dup * * ; $ ok
4 cubo . $ 64 ok
4 ' cubo execute $ 64 ok
```

Las líneas segunda y tercera son equivalentes. `''` toma del TIB la siguiente palabra, `'cubo'`, la localiza en el diccionario y coloca en la pila el valor de su puntero a código. Después, `'execute'` pasa el control a la dirección indicada en la pila. De hecho, el intérprete Forth, que está escrito en Forth, usa las palabras `''` y `'execute'`. Para poner en práctica estas ideas, ampliaremos el vocabulario para trabajar con vectores con dos palabras nuevas. La primera, `'v-map'` permite aplicar una función cualquiera a los elementos de un vector. La segunda, `'v-tab'` permite elaborar tablas, asignando a los elementos del vector un valor que es una función del índice de cada elemento.

```
: v-map ( dir N xt --)
  swap 0 do
    over I cells + @ over execute
    rot dup I cells + rot swap ! swap loop
  2drop ;
: v-tab ( dir N xt --)
  swap 0 do
    2dup I swap execute swap I cells + ! loop
  2drop
```

Es costumbre en los comentarios de pila usar la notación `'xt'` para indicar direcciones no de datos sino de código ejecutable (`'xt'` proviene de *execution token*) Para ilustrar su funcionamiento, creamos un vector de seis elementos, que toman como valores iniciales los que proporciona la función polinómica del índice  $f(n) = 4n - 2$ .

```
: pol 4 * 2 - ; $ ok
6 vector p $ ok
p ' pol v-tab $ ok
p v. $
-2
2
6
10
14
18
ok
```

A continuación, aplicamos la función `cubo` a los elementos del vector resultante:

```

: cubo dup dup * * ; $ ok
p ' cubo v-map $ ok
p v.
-8
8
216
1000
2744
5832
ok

```

Aquellos que no desconozcan Lisp, reconocerán en 'v-map' una implementación de la función (`mapcar`). ¿No es sorprendente que cuatro líneas de código permitan salvar, en lo tocante a este punto, la distancia que media entre un lenguaje de bajo nivel como Forth y otro de altísimo nivel, como Lisp? ¿O es que después de todo Forth no es un lenguaje de bajo nivel?

Consideremos ahora una estructura de programación muy frecuente que surge cuando en función del valor de una variable es preciso tomar una decisión. La forma más elemental de resolver esta cuestión consiste en implementar una secuencia de condicionales:

```

if (<condicion 1>)
  <accion 1>
else
if (<condicion 2>)
  <accion 2>
else
...

```

Es algo más elegante usar una estructura similar al 'switch()' de C:

```

switch (<variable>){
  case <valor 1>: <accion 1> ;
  case <valor 2>: <accion 2> ;
  case <valor 3>: <accion 3> ;
  ...
}

```

Pero en cualquiera de los dos casos, la solución es insatisfactoria, y se resuelve de forma más compacta, con un código más legible y con una ejecución más veloz implementando una tabla de decisión. Una tabla de decisión es un vector que contiene direcciones de funciones. El valor de la variable que en las dos soluciones anteriores ha de usarse para averiguar qué acción tomar, podemos usarlo simplemente para indexar el vector de direcciones, y pasar la ejecución a la dirección que indique la entrada correspondiente. Consideremos por ejemplo el caso es que ha de tomarse una decisión distinta según que el valor de una variable sea 0, 1 o 2. En lugar de escribir tres bloques `'if...else'` consecutivos, o un `'switch()'` con tres `'case'`, creamos un vector de tres entradas y en cada una de ellas escribimos la dirección de la rutina que se ejecutará cuando el valor de la variable sea 0, 1 o 2. Llamemos `'n'` a la variable que determina la decisión. En lenguaje Forth: el valor de `'n'` quedará en la pila, lo usaremos para incrementar la dirección de la primera celda de la tabla, usaremos la palabra `'@'` para acceder a ese valor y a continuación la palabra `'execute'` para efectuar la operación que corresponda. La ventaja de este método es más evidente cuanto mayor es el tamaño de la tabla. Si es preciso decidir para un rango de veinte valores, las estructuras `'if...else'` o `'switch()'` tendrán por término medio que efectuar diez comparaciones. Con una tabla de decisión, no importa su tamaño, el salto a la función pertinente se produce siempre en el mismo tiempo, con el mismo número de instrucciones.

Pero, ¿qué sucede si los valores de `'n'` no son consecutivos? Pueden darse dos situaciones: que el rango de valores que puede tomar esta variable sea estrecho o que no lo sea. En el primer caso, conviene implementar de todas formas una tabla de decisión, colocando en las posiciones correspondientes a los valores del índice que no pueden darse punteros a una función vacía. Por ejemplo, si `'n'` puede tomar valores 0, 1 y 5 construiremos una tabla con seis celdas, colocando las direcciones adecuadas en las celdas primera, segunda y sexta y la dirección de una palabra vacía en el resto de las celdas de la tabla. En el segundo, será preciso construir una tabla con parejas de valores: el valor de la variable sobre el que hay que decidir y la dirección de la palabra que ha de ejecutarse. En ese caso, será preciso recorrer la tabla hasta encontrar la pareja `'n,<direccion>'`. Si la tabla consta de  $M$  entradas, será preciso por término medio recorrer  $M/2$  entradas, salvo que tengamos la precaución de rellenar la tabla ordenadamente, según valores crecientes o decrecientes de `'n'`. En ese caso, puede hacerse una búsqueda binaria, más eficiente que el recorrido lineal de la tabla. Imaginemos que `'n'` puede tomar valores en el intervalo  $[1, 64]$ , pero que sólo pueden producirse los valores 1, 17 y 64. En

ese caso, una tabla simple contendría 61 direcciones de una función vacía. No sería práctico. Se puede sin embargo comprimir muy eficientemente una tabla casi vacía, pero esa discusión ya nos llevaría demasiado lejos, habiendo quedada la idea expresada con claridad. En cada situación, el sentido común nos aconsejará sobre la mejor solución, que ya sabemos que no es un largo `'switch()'`.

## 4.7. Distinción entre ' y [']

La palabra `''` apila la dirección del código asociado a la palabra siguiente de la línea de entrada. Así, es posible escribir

```
: 2* 2 * ; $ ok
: a ' execute ; $ ok
30 a 2* . 60 ok
```

La dirección de `'2*'` no se compila en `'a'`, sino que se toma en tiempo de ejecución. Si lo que deseamos es que la dirección de determinada palabra quede compilada en el cuerpo de otra que está siendo definida, es preciso usar `'[']`:

```
: a '[' 2* execute ; $ ok
3 a . $ 6 ok
```

Terminamos este capítulo reuniendo el vocabulario que hemos creado para trabajar con vectores. Contiene algunas palabras que no están en el texto:

```
\ -----
\ VECTORES. versio'n de 20 de marzo de 2008
\ -----

\ -----
\ crea un vector; en tiempo de creacio'n, deja
\ como primer elemento la dimensio'n; en tiempo
\ de ejecucio'n, deja en la pila la direccio'n
\ del primer elemento y el nu'mero de elementos
\ -----
: vector create dup , cells allot
  does> dup @ swap 1 cells + swap ;
```

```

\ -----
\ trae a la pila el elemento n
\ -----
: v@ ( n d N -- )
  1- rot min cells + @ ;

\ -----
\ escribe un valor 'v' en la posicin 'n'
\ -----
: v! ( v n d N -- )
  1- rot min cells + ! ;

\ -----
\ rellena el vector con el valor '0'
\ -----
: 0v! ( d N -- )
  0 do dup I cells + 0 swap ! loop ;

\ -----
\ imprime los elementos del vector
\ -----
: v. ( d N -- )
  cr 0 do dup I cells + @ . cr loop
  drop ;

\ -----
\ rellena un vector con valores aleatorios
\ -----
: random 899 * 32768 mod ;
: v-random ( semilla d N -- )
  0 do swap random tuck over I cells + ! loop
  2drop ;

\ -----
\ obtiene el valor ma'ximo del vector
\ -----
: v-max ( d N -- max )
  over @ swap 0 do over I cells + @ max loop
  nip ;

```

```

\ -----
\ obtiene el valor mi'nimo del vector
\ -----
: v-min ( d N -- min )
  over @ swap 0 do over I cells + @ min loop
  nip ;

\ -----
\ obtiene la suma de los elementos del vector
\ -----
: v-sum ( d N -- sum )
  0 swap 0 do over I cells + @ + loop
  nip ;

\ -----
\ obtiene la suma de cada elemento al cuadrado
\ -----
: v-sum2 ( d N -- sum(v[i]^2) )
  0 -rot 0 do dup I cells + @
  dup * >r swap r> + swap loop drop ;

\ -----
\ valor medio
\ -----
: v-avrg ( d N -- media )
  tuck v-sum swap / ;

\ -----
\ obtiene el lugar en que el valor 'x' se
\ encuentra, o -1 si no se encuentra
\ -----
: v-pos ( x d N -- indice )
  0 do 2dup I cells + @ =
  if 2drop I unloop exit then
  loop 2drop -1 ;

\ -----
\ posicio'n del valor ma'ximo del vector
\ -----
: v-pos-max ( d N -- indice valor max. )
  2dup v-max -rot v-pos ;

```

```

\ -----
\ posicio'n del valor mi'nimo del vector
\ -----
: v-pos-min ( d N -- indice valor min. )
  2dup v-min -rot v-pos ;

\ -----
\ aplica una funcin a cada elemento del vector
\ -----
: v-map ( xt d N -- )
  0 do 2dup I cells + @ swap execute
  over I cells + ! loop
  2drop ;

\ -----
\ asigna a cada elemento una funcio'n de su i'ndice
\ -----
: v-tab ( xt d N -- )
  0 do over I swap execute
  over I cells + ! loop
  2drop ;

\ -----
\ producto escalar de dos vectores
\ -----
: v-dot ( d N d' N -- x.y)
  drop swap >r 0 -rot r>
  0 do
    2dup I cells + @
    swap I cells + @ *
    >r rot r> + -rot loop 2drop ;

\ -----
\ ordena un vector
\ -----
: @@ ( d d' --v v' )
  swap @ swap @ ;
: exch ( d d' -- )
  2dup @@ >r swap !
  r> swap ! ;

```

```

: in-sort ( d N -- )
  1 do
    dup I cells + 2dup @@ >
    if over exch else drop then
  loop drop ;
: v-sort ( d N -- )
  dup 1- 0 do 2dup in-sort 1- swap 1 cells + swap
  loop 2drop ;

\ -----
\ acumulador: sustituye cada elemento por la
\ sumatoria de los elementos anteriores, incluido
\ e'l mismo
\ -----
: v-acc ( d N --)
  1 do dup I cells +
    over I cells + 1 cells -
    @@ + over I cells + ! loop drop ;

\ -----
\ invierte un vector
\ -----
: v-reverse ( d N --)
  1- over swap cells +
  begin 2dup < while
    2dup exch
    1 cells - swap 1 cells + swap
  repeat 2drop ;

```

# Capítulo 5

## Cadenas de caracteres

### 5.1. Formato libre

Forth carece de un formato para cadenas de caracteres. Éste queda a la elección del programador, pero es común trabajar con cadenas que incluyen la longitud en el primer byte o en su caso en la primera celda. La estructura es pues la misma que hemos creado para los vectores en el capítulo anterior. Aquí presentaremos una docena de palabras para trabajar con cadenas, pero puesto que no hay un formato predefinido tampoco habrá palabras que sea preciso escribir a bajo nivel. Todas las que presentaremos se pueden escribir en Forth, y será un buen ejercicio estudiar la forma en que se implementan algunas de ellas.

### 5.2. Las palabras `accept`, `type` y `-trailing`

La primera de las palabras que consideramos es `accept`, que sirve para leer cadenas desde teclado. Ya tenemos la palabra `key` para leer pulsaciones de tecla individuales, de manera que es natural pensar en un bucle aceptando teclas y guardándolas en direcciones crecientes a partir de una dada. De hecho, `accept` espera en la pila una dirección a partir de la cual guardar la cadena y el número máximo de caracteres que se introducirán.

La edición de una cadena puede ser una tarea compleja, dependiendo de las facilidades que se ofrezcan: introducir caracteres, borrarlos, moverse al principio o al final de la cadena, sobrescribir o insertar... Por este motivo, vamos a implementar únicamente el motivo central de esta palabra: tomar caracteres del teclado y colocarlos en direcciones consecutivas de memoria a

partir de una dada. Al terminar, `accept` deja en la pila el número de caracteres introducidos. La lectura de caracteres puede interrumpirse en cualquier momento con la tecla `intro`. Esta es una muy sencilla implementación de `accept`<sup>1</sup>:

```
\ version sencilla de accept

1 : accept ( dir N -- n )
2   0 -rot           \ pone un contador a 0 en la base
3   0 ?do           \ inicia bucle
4     key dup 13 =   \ lee tecla
5     if            \ comprueba si es INTRO
6       drop leave  \ si lo es, salir
7     else
8       over c!     \ almacenar valor en direccion
9       1+ swap    \ incrementar direccion
10      1+ swap    \ incrementar contador
11    then
12  loop
13  drop ;         \ dejar en la pila solo contador
```

Para facilitar los comentarios al programa hemos introducido números de línea (que no pertenecen al programa, por supuesto). El comentario de pila de la primera línea indica que `accept` espera en la pila una dirección y un número, que es el máximo de caracteres que se leerán. Al terminar, deja en la pila el número de caracteres leídos. La línea 2 introduce en la pila bajo `dir` un contador, con un valor inicial de 0. En caso de que `N` sea distinto de cero, comienza en la línea 3 un bucle. `key` leerá un carácter y colocará su código en la pila. Mediante `dup` se obtiene copia de su valor y se compara con el valor 13, que es el código de la tecla `intro`. Si coinciden, `drop` elimina el código de la tecla y a continuación se abandona el bucle. Si el código es distinto del código de salida, se hace una copia de la dirección sobre el código de la tecla para tener en la pila en el orden adecuado los parámetros que necesita `'c!'` para guardar el carácter. Finalizada esta operación, quedan en la pila la dirección original y el contador: ambos han de ser incrementados en una unidad, que es lo que hacen las líneas 9 y 10. Finalmente, en la línea 13 se elimina de la pila la dirección del último carácter almacenado y queda por tanto sólo el valor del contador.

---

<sup>1</sup>En el capítulo 10 se encuentra una versión mejor

Obsérvese que en ningún momento se envía copia del carácter leído a pantalla, por lo que la entrada se realiza *a ciegas*. Bastaría después de la línea 7 insertar la frase 'dup emit' para tener copia en pantalla.

Una versión mucho mejor:

```
: accept ( dir N -- n )
  over >r 0 ?do
    key dup 13 = if drop leave then
    over c! char+
  loop r> - ;
```

Pasemos a la descripción de la palabra 'type'. Esta palabra espera en la pila una dirección y un número de caracteres, y presenta a partir de la dirección base tantos caracteres como se indique. Por ejemplo:

```
pad 80 accept $ Hola mundo $ ok
pad 3 type Hol ok
```

La implementación de esta palabra en Forth es sencilla:

```
: type ( dir n --)
  0 ?do dup c@ emit 1+ loop ;
```

El hecho de que el primer byte de una cadena de caracteres contenga la longitud de la misma facilita algunas operaciones comunes, pues evita recorrer cada vez la cadena buscando el final, como sucede con aquellos formatos que, como C, usan un byte especial para indicarlo. La palabra '-trailing' permite eliminar los caracteres en blanco al final de una cadena, pero, en lugar de eliminarlos físicamente, redimensionando la cadena, se limita a modificar el byte de longitud. Para implementar esta palabra sólo es preciso acceder al último carácter de la cadena y retroceder hasta encontrar un carácter distinto del espacio en blanco. Finalmente, el número de caracteres retrocedidos se resta del byte de longitud. '-trailing' espera en la pila una dirección base y el byte de longitud, y deja la pila preparada con la dirección base y el nuevo byte de longitud de manera que 'type' pueda ser llamada a continuación.

```
pad 80 accept $ Forth es ... $ ok
pad swap -trailing type $ Forth es ... ok
```

La primera línea lee de teclado hasta 80 caracteres y los coloca en `pad`, dejando en la pila la cuenta de los caracteres introducidos. Si ahora llevamos la dirección de `pad` a la pila e intercambiamos la dirección con el contador dejado por `'accept'`, `'trailing'` ya puede efectuar su trabajo, dejando la dirección y el nuevo contador en lugar del original. Finalmente, `'type'` puede, usando el nuevo contador, imprimir la cadena omitiendo los espacios en blanco del final.

Eh aquí la implementación de `'-trailing'`:

```
: -trailing ( dir N -- dir M)
  begin
    2dup + 1- c@ bl = over and
  while
    1-
  repeat ;
```

### 5.3. Las palabras `blank` y `fill`

La palabra `'blank'` espera en la pila una dirección base y un número, y almacena tantos caracteres de espacio en blanco como indique ese número a partir de la dirección dada. `'blank'` es un caso particular de `'fill'`, que funciona igual salvo que ha de especificarse qué carácter va a usarse. La implementación no difiere mucho de `'0v!'` que como se recordará del capítulo anterior rellena un vector con ceros.

```
: fill ( dir N char --)
  swap 0 ?do
    2dup swap I + c! loop ;
```

Por ejemplo:

```
pad 10 char A fill $ ok
pad 10 type $ AAAAAAAAAA ok
```

### 5.4. Las palabras `move` y `compare`

La palabra `'move'` espera en la pila dos direcciones y un contador, y mueve un bloque de tantos bytes como indique el contador desde la primera a la segunda dirección. Esta operación parece trivial, pero sólo lo es si la diferencia

entre las direcciones origen y destino es mayor o igual al tamaño del bloque que se desea mover. Si no es así, se producirá solape entre las posiciones del bloque antes y después de moverlo. Es fácil comprender que si la dirección destino es mayor que la dirección origen será preciso copiar byte a byte pero comenzando por el del final del bloque, mientras que si la dirección destino es menor que la dirección origen será preciso copiar empezando por el primer byte del bloque. Forth cuenta con las variantes 'cmove' y 'cmove>', pero no las discutiremos aquí por dos razones: primero, porque en general sólo interesa mover un bloque evitando solapes; segundo, porque generalmente copiaremos unas cadenas en otras. Como cada cadena tendrá reservado su espacio, si el tamaño del bloque es inferior o igual a ese espacio nunca podrán ocurrir solapes, independientemente de que la dirección origen sea menor o mayor que la dirección destino.

Para ilustrar el uso de esta palabra, copiaremos una cadena depositada en el PAD a la posición de una cadena previamente reservada. El PAD es un área de memoria que usa Forth para tareas como conversión de números a cadenas y otras que necesiten un *buffer* para uso temporal. Si estamos trabajando con un tamaño de celda de 4 bytes, reservaremos con la primera línea espacio para cuarenta caracteres, leeremos una cadena desde teclado y la depositaremos en la variable recién creada:

```
variable cadena 9 cells allot $ ok
pad 40 accept $ Era un hombre extraordinario $ ok
pad cadena 20 move $ ok
cadena 20 type $ Era un hombre extrao ok
```

Por supuesto, 'accept' puede tomar a *cadena* como dirección de destino, pero lo que queremos es ilustrar el uso de 'move'. ¿Cómo la implementamos? Si la dirección destino es mayor que la dirección origen, copiaremos desde el último byte hacia atrás. Implementaremos esta operación mediante una palabra a la que llamaremos 'move+'. Si por el contrario la dirección destino es menor que la dirección origen, copiaremos desde el primer byte para evitar solapes. Llamaremos a esta operación 'move-'. 'move' ha de averiguar cuál es el caso, y actuar en consecuencia. Por supuesto, es preciso descartar el caso trivial en que coincidan las direcciones origen y destino.

```
: move ( d d' n --)
  >r 2dup < if r> move+ else r> move- then ;
```

En cuanto a 'move+':

```
: move+ ( d d' n -- )
  0 do
    2dup I +
    swap I +
    c@ swap c!
  loop 2drop ;
```

Quede la palabra 'move-' como ejercicio para el lector. En el siguiente fragmento de sesión comprobamos el funcionamiento de 'move+':

```
variable cadena 9 cells allot $ ok
cadena 40 accept $ Es hora de auxiliar a nuestra Nacion $ ok
cadena cadena 4 + 6 move+ $ ok
cadena 40 type $ Es hEs hor auxiliar a nuestra Nacion ok
```

Si los bloques origen y destino no se solapan:

```
variable c1 9 cells allot $ ok
variable c2 9 cells allot $ ok
c1 40 accept $ Es hora de auxiliar a nuestra Nacion $ ok
c1 c2 40 move+ $ ok
c2 40 type $ Es hora de auxiliar a nuestra Nacion ok
```

## 5.5. La palabra compare

`compare` realiza la comparación alfabética de dos cadenas de caracteres, dejando en la pila un resultado que depende de si la primera cadena es mayor (alfabéticamente), menor o igual que la segunda. Espera en la pila la dirección y la longitud de la primera cadena y la dirección y longitud de la segunda. Llegados a este punto, es conveniente detenerse momentáneamente. Hemos presentado varias funciones de cadena que esperan en la pila una dirección y un contador con la longitud de la cadena. Pero también hemos dicho que Forth carece de un tipo predefinido de cadena. Es conveniente entonces crear una palabra que llamaremos 'cadena', similar a la palabra 'vector' que pueda usarse en la forma

```
20 cadena S
```

para crear una cadena de nombre 'S' en este caso con una longitud de veinte caracteres.

Interesa también que la palabra creada con 'cadena', en tiempo de ejecución, deje en la pila la dirección del primer elemento y el contador con la longitud de la cadena. Es sencillo:

```
: cadena create dup , chars allot
  does> dup @ swap 1 cells + swap ;
```

En esta implementación, no reservamos un único byte, sino una celda entera para almacenar la longitud de la cadena. Por otro lado, 'chars' es necesario aquí para asegurarnos de que la palabra está correctamente definida en sistemas donde los caracteres puedan ocupar más de un byte, como en UTF-16. Ahora podemos trabajar cómodamente, por ejemplo, para comprobar el funcionamiento de la palabra 'compare':

```
20 cadena S $ ok
20 cadena T $ ok
S accept drop $ sanchez romero $ ok
T accept drop $ martinez garcia $ ok
S T compare . $ 1 ok
T S compare . $ -1 ok
S S compare . $ 0 ok
```

## 5.6. Algunas funciones útiles

Terminamos este capítulo escribiendo un vocabulario sencillo pero útil para trabajar con cadenas. Usaremos el formato creado con 'cadena', y que reserva una celda para la longitud de la cadena y a continuación espacio para tantos caracteres como se indiquen en esa celda.

Un par de transformaciones sencillas consisten en pasar una cadena de minúsculas a mayúsculas o viceversa. Para implementar estas palabras, usaremos otra, que llamamos 'within'. 'within' toma tres números de la pila y nos dice si el tercero está o no contenido en el intervalo cerrado indicado por los dos primeros.

```
: within ( a b c -- test)
  tuck >= -rot <= and ;
```

Puede emplearse otra implementación, más eficiente pero menos evidente:

```
: within over - >r - r> u< ;
```

En la tabla ASCII, las letras minúsculas ocupan posiciones entre la 97 y la 132, mientras que las mayúsculas se encuentran en el intervalo 65,90. La palabra 'toupper' toma una cadena y convierte a mayúsculas todas las letras minúsculas. La palabra 'tolower' toma una cadena y convierte a minúsculas todas las letras mayúsculas. El modo de proceder es sencillo: se generan direcciones a partir de la dirección base, se comprueba que el carácter correspondiente se encuentre en el rango de las mayúsculas o minúsculas y en ese caso se realiza una conversión. Si es de mayúsculas a minúsculas, sumando 32 y restando en caso contrario:

```
: bounds ( d N -- d1 d2) over + swap ;
: between 1+ within ;
: upcase? [char] A [char] Z between ;
: lowercase? [char] a [char] z between ;
: toupper dup lowercase? -32 and + ;
: tolower dup upcase? 32 and + ;
: $toupper (dir N -- )
  bounds do
    I c@ toupper I c!
  loop ;
: $tolower ( dir N -- )
  bounds do
    I c@ tolower I c!
  loop ;
```

Antes de continuar, presentaremos otra palabra que usaremos más adelante. Se trata de '/string'. Esta palabra toma la dirección de una cadena, su longitud y un número entero de caracteres que quieren eliminarse y deja en la pila los parámetros correspondientes a la nueva cadena. Si  $d_1$  es la dirección de la cadena original,  $N_1$  su longitud y  $n$  el número de caracteres que quieren eliminarse, entonces quedan en la pila  $d_2 = d_1 + n$  y  $N_2 = N_1 - n$ :

```
: /string ( d1 N1 n -- d2 N2 )
  tuck - -rot + swap ;
```

Si deseamos asegurarnos de que no se van a eliminar más caracteres de los que contiene la cadena también podemos escribir:

```
: /string over min >r swap r@ + swap r> - ;
```

La palabra 'x-character' se ocupa de eliminar un carácter de una cadena. Espera en la pila la dirección del primer carácter de la cadena, el número de caracteres y la posición del carácter que será eliminado. Esta palabra es la base, aunque no muy eficiente, para eliminar un rango de caracteres dentro de una cadena. El procedimiento es sencillo: el caracter que será eliminado es sustituido por el siguiente, éste por el siguiente, y así sucesivamente. El último caracter es sustituido por un espacio en blanco. 'i-character' inserta un caracter en una cadena. Se esperan en la pila la dirección de la cadena, su longitud, la posición donde el nuevo carácter será insertado y el carácter a insertar. Esta palabra es la base para insertar subcadenas dentro de cadenas. El funcionamiento es también sencillo: el último carácter de la cadena es sustituido por el penúltimo, éste por el anterior y así sucesivamente hasta hacer hueco al caracter que se vá a insertar.

```
: x-character ( d N m -- )
  /string
  2dup + 1- >r
  over 1+ -rot 1- cmove
  bl r> c! ;

: i-character ( d N m c -- )
  >r /string
  over >r
  over 1+ swap 1- cmove>
  r> r> swap c! ;
```

# Capítulo 6

## Control del compilador

### 6.1. Nueva visita al diccionario

Teníamos una descripción de la estructura del diccionario, y la forma en que se añaden, compilan, nuevas palabras. Esencialmente, cuando se crea una nueva palabra se crea un enlace con la palabra anterior, se reserva espacio para el nombre, espacio para un enlace con el código asociado a la palabra y otro espacio para los datos. A su vez, el código asociado consiste esencialmente en las direcciones de las palabras que aparecen en la definición de la nueva entrada que se está compilando.

Ahora vamos a pulir esta descripción. La corrección más importante que hemos de hacer al esquema anterior es: no siempre una palabra contenida en la definición de otra que se está compilando deja huella en el diccionario. Hay palabras que el compilador no compila, sino que ejecuta directamente. En primer lugar, ilustraremos este concepto, y después explicaremos su razón de ser.

Antes de nada, recordemos que en el byte de longitud del nombre de una palabra disponemos de algunos bits libres. Uno de estos bits es el que indica si una palabra, al encontrarse en la definición de otra, ha de compilarse o ejecutarse inmediatamente. A las palabras que tienen activo este bit se les llama 'immediate'. Por ejemplo

```
: saludos ." Hola" ; immediate $ ok
saludos $ Hola ok
```

Hemos definido la palabra 'saludos' y podemos usarla normalmente. Ahora bien, si apareciese en la definición de otra palabra, no se incorporaría a ella: se ejecutaría de forma *inmediata* sin dejar huella.

```
: rosaura saludos ." Rosaura" ; $ hola ok
rosaura Rosaura ok
```

En la definición de 'rosaura' el compilador encontró la palabra 'saludos', y antes que nada comprobó el bit que indica si la palabra ha de ser compilada en la nueva definición o ejecutada inmediatamente. Como ese bit fue activado justo después de que se definiera 'saludos' mediante 'immediate', el compilador ejecuta directamente 'saludos', sin incorporarla a la definición de 'rosaura'. Cuando ejecutamos 'rosaura' comprobamos que efectivamente no hay ni rastro de 'saludos'.

¿Cuál es la utilidad de esto? Hemos repetido algunas veces que Forth está escrito en Forth, y hemos dado algunos ejemplos. Hemos dicho en algún momento también que incluso las que en otros lenguajes son palabras reservadas para construir bucles y condicionales en Forth son palabras definidas como cualesquiera otras. Veamos un ejemplo tan simple que probablemente arranque una sonrisa al lector <sup>1</sup>.

```
: begin here ; immediate
```

Imaginemos que estamos compilando una palabra de nombre 'test':

```
: test <#1> <#2>...<#n> begin <#n+1> <#n+2>... until ;
```

El compilador crea una entrada en el diccionario con el nombre 'test' y comienza a compilar las palabras <#1>, <#2> ... Entonces encuentra 'begin'. Pero 'begin' está marcada como *immediate*, luego, en lugar de compilarla, la ejecuta. ¿Qué hace? Recordemos la definición de 'here':

```
: here cp @ ;
```

---

<sup>1</sup>Advertimos que no siempre es ésta la implementación

Es decir, se apila el valor de `cp`, que es el puntero a la celda que hubiese ocupado al compilarse `'begin'` si no hubiese sido *immediate*. Así que `'begin'` no ha dejado huella en `'test'`, sino unicamente el valor de `cp` en la pila. Después de esto, se compila `<#n+1>` en la dirección en que hubiese sido compilada `begin`. Pero esa dirección es la dirección de retorno que necesita `'until'`. Tarde o temprano aparecerá un `'until'`. El código asociado a `'until'` deberá decidir si salir o no del bucle; si es éste el caso, necesitará conocer la dirección de vuelta. Esa dirección es la dirección donde se compiló `<#n+1>`, y se encuentra en la pila porque fue dejada allí por `'begin'`. ¡Atención! no debe confundirse la dirección donde se compiló `<#n+1>` con la dirección donde reside el código de `<#n+1>`. La primera contiene un puntero a la segunda.

## 6.2. Inmediato pero...

Existen palabras *immediate* que no pueden ejecutarse enteras inmediatamente. Dicho de otra forma, dentro de palabras *immediate* pueden existir palabras que necesariamente hayan de compilarse. Para ver la necesidad de esto, consideremos una variante de `'test'` que usa `'do'` en lugar de `'begin'`.

```
: test <#1> <#2>...<#n> do <#n+1> <#n+2>... loop ;
```

Por lo que respecta a la dirección de retorno de `'loop'`, `'do'` se comporta igual que `'begin'`. Ahora bien, `'do'` ha de tomar de la pila un valor inicial y un valor final para el contador, y estos valores no pueden conocerse de antemano. En particular, no pueden conocerse cuando `'test'` está siendo compilada. Se conocerán cuando se ejecute `'test'`. Por tanto, hay una parte de `'do'` que se ejecuta de forma inmediata, pero hay otra parte que ha de quedar en `'test'`, para que, cuando llegue el momento, pueda tomar de la pila los contadores y pasarlos a la pila de retorno. Estas palabras incluidas en otras de tipo *immediate* pero cuya ejecución no puede ser inmediata, sino que ha de posponerse al momento en que se ejecute la palabra que está siendo compilada se distinguen mediante una nueva palabra: `'postpone'`.

Véase la implementación de `'do'`:

```
: do postpone 2>r here ; immediate
```

Para ahondar algo más en el comportamiento de `'postpone'` consideremos la siguiente secuencia:

```
: A ." aaa" ; $
: B postpone A ." bbb" ; immediate $
: C B ." ccc" ; $ bbb
C $ aaaccc
```

¿Qué ocurre? En primer lugar, hemos creado la palabra 'A', que en tiempo de ejecución se limitará a imprimir la pequeña cadena 'aaa'. A continuación creamos la palabra 'B' y la declaramos como 'immediate'. Esta palabra contiene a la palabra 'A' y a la pequeña cadena 'bbb'. Cuando compilo la palabra 'C', durante el proceso de compilación se encuentra 'B'. Ahora bien, 'B' se ha declarado como 'immediate' y por tanto no será compilada en 'C', sino ejecutada directamente. De esta forma, 'B' no dejará huella en 'C'. En realidad, sí que deja una huella, ya que 'B' contiene a 'A' y ésta viene precedida de 'postpone'. Lo que sucede es que, mientras se está compilando 'C', se encuentra 'B', declarada como 'immediate', se interrumpe el proceso de compilación y se pasa a ejecutar 'B'. Durante la ejecución de 'B' se encuentra un 'postpone', y eso indica que es preciso tomar la palabra siguiente y compilarla. De esta forma, aunque 'B' está declarada como 'immediate', una parte de ella, 'A' escapa a la regla y es compilada. De esta forma 'A' queda incorporada a 'C'. Si al lector le parece algo farragoso este párrafo (a nosotros también), vuelva sobre las cuatro líneas de código de ejemplo y trate de comprender qué está ocurriendo.

Otra forma de considerar 'postpone' es desde el punto de vista de la operación del sistema Forth, no desde el punto de vista del programador que usa la palabra. Así, consideremos las dos situaciones siguientes que pueden darse al encontrar una palabra 'postpone': cuando el sistema está compilando y cuando está ejecutando una palabra declarada como 'immediate'.

#### 1. Durante la compilación

- a) si la palabra siguiente a 'postpone' es 'immediate', no se compila la primera, sino la segunda
- b) si la palabra siguiente a 'postpone' no es 'immediate', se compilan ambas

#### 2. Durante la ejecución de una palabra 'immediate'

- a) si se encuentra una palabra 'postpone', necesariamente procedemos del segundo caso del punto anterior. En ese caso, se compila la palabra que sigue a 'postpone'

Como se ve, hay cierta 'inteligencia' incorporada en esta palabra.

### 6.3. Parar y reiniciar el compilador

Veamos una definición sencilla:

```
: mas-2 2 + ;
```

Hemos visto que en el código asociado a una palabra hay, esencialmente, las direcciones de otras palabras: las que forman la definición de la primera. Pero en esta definición también puede haber números. En ese caso ha de compilarse el número y el código para tratar con números, que dirá sencillamente *cójase el número y apílese*. Imaginemos ahora que en lugar de la definición sencilla anterior tenemos otra donde la cantidad a sumar es el resultado de una operación aritmética:

```
: mas-x 2 3 + 7 * + ;
```

Cada vez que se llame a 'mas-x' habrá de evaluarse la expresión que contiene, lo cual resulta poco eficiente. La primera ocurrencia que tenemos es sustituir la expresión  $2\ 3 + 7 *$  por el valor 35. Pero quizás este valor no nos diga nada en poco tiempo y el código fuente resulte más claro si se mantiene la expresión que da origen al valor 35. Estas dos exigencias, eficiencia y claridad, pueden conjugarse de forma sencilla. Dentro de una definición, el compilador puede detenerse temporalmente, efectuar una operación, activarse de nuevo y compilar el valor resultante. La palabra '[' detiene el compilador. La palabra ']' vuelve a activarlo. De esta forma, podríamos escribir:

```
: mas-x [ 2 3 + 7 * ] + ;
```

Sólo queda compilado el resultado '35 +', pero el código fuente retiene la expresión que produce ese 35. Pero ¡cuidado!, la definición anterior no va a funcionar. ¿Por qué? Porque el compilador compila la línea depositada en el TIB. Coge de ahí palabra por palabra y las analiza, identificando en su caso a los números. Pero si suspendemos el compilador, el resultado de la operación entre corchetes va a ser depositado en la pila. Una vez que se reactive el compilador encontrará la palabra '+'. ¿Cómo va a saber que había en la pila un valor numérico para compilar? Es preciso decírselo explícitamente, y de eso se ocupa la palabra 'literal'. Su función es simple: le dice al compilador que compile un valor literal que se encuentra en la pila. La versión correcta de 'mas-x' es entonces

```
: mas-x [ 2 3 + 7 * ] literal + ;
```

¿Qué lección se sigue de esta discusión? Los compiladores tradicionales son programas grandes y complejos, porque han de saber reconocer todas las combinaciones posibles de operandos y operadores, y todas las estructuras sintácticas que puedan construirse, además de todas las formas posibles en que una sentencia o una estructura pueden escribirse incorrectamente, para emitir en ese caso un mensaje de error. En lugar de esa aproximación, Forth adopta otra mucho más sencilla y flexible. Cada palabra es una unidad *per se* que realiza una tarea simple que modifica de algún modo una o ambas pilas. Lo que en otros lenguajes es sintaxis, en Forth es un protocolo entre palabras que intercambian información a través de la pila. Pero es el programador quien da sentido a este protocolo.

# Capítulo 7

## Entrada y salida sobre disco

### 7.1. Un disco es un conjunto de bloques

Todos los sistemas informáticos han de contar con al menos una pequeña cantidad de memoria permanente. Puede ser una cantidad de memoria ROM o puede ser memoria *flash*, pero casi siempre, por precio, capacidad y prestaciones, será una unidad de disco. Una unidad de disco consta de un elevado número de sectores, cada uno de los cuales permite almacenar 512 bytes. Normalmente se agrupa un cierto número de sectores en lo que se llama un bloque. Un bloque suele contener un número de sectores que es un múltiplo de dos. Valores habituales para el tamaño de un bloque son 1K, 2K, 4K, 8K, etc. Es preciso un método para organizar este conjunto de bloques. Es preciso saber qué bloques se encuentran ocupados y qué bloques libres, y es preciso saber qué bloques pertenecen a un archivo. En definitiva, es precisa una base de datos de bloques. A esta base de datos de bloques es a lo que se llama *sistema de archivos*. El acceso a esta base de datos está definido mediante una interfaz que consta de alrededor de una veintena de funciones. Es el sistema operativo quien ofrece estas funciones a los programas.

Cuando arranca el sistema operativo, una de las primeras cosas que ha de hacer es hacerse cargo de la base de datos de bloques, es decir, iniciar el sistema de archivos. Para eso necesita información, y esta información se encuentra en el mismo disco, ocupando una parte de los bloques.

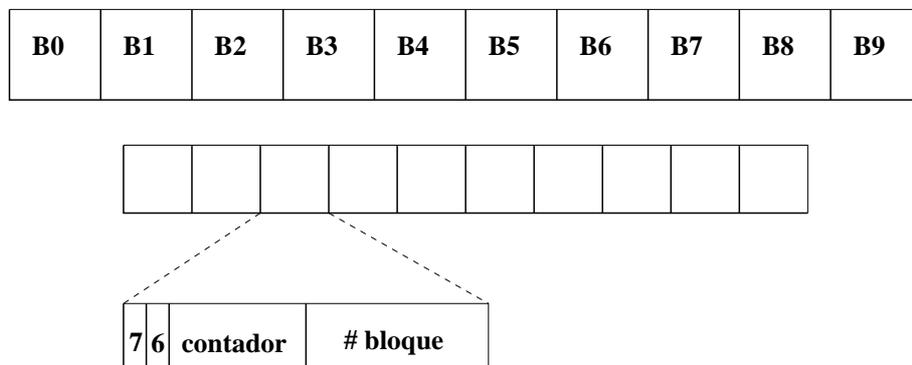
Por otra parte, hemos dicho que Forth puede funcionar como un sistema operativo independiente o como una aplicación que se ejecuta sobre otro sistema. Por eso, Forth puede interactuar con el disco de tres formas distintas:

1. Forth se hace cargo totalmente del disco, que es considerado como una reserva de bloques de 1K, sobre la que no se impone ninguna estructura.
2. Forth, ejecutándose sobre otro sistema, usa un archivo determinado como si fuese un conjunto de bloques. Las funciones para acceder a archivos ofrecidas por el sistema operativo permiten leer los bloques, que no son más que trozos de 1K del archivo.
3. Forth hace un uso completo del sistema de archivos del sistema operativo sobre el que se ejecuta para leer, escribir, crear, borrar archivos, etc.

## 7.2. Cómo usa Forth los bloques

En esta sección nos centraremos en el primer método. Las implementaciones modernas de Forth usan los métodos segundo y tercero, pero aquí queremos retener la simplicidad y el *tacto* del Forth tradicional. Para fijar ideas, supondremos una unidad de disco de 256 bloques. Las operaciones de lectura y escritura en disco son varios órdenes de magnitud más lentas que las operaciones de lectura y escritura en memoria, por lo que es una buena idea tener en ésta una pequeña parte del disco. La experiencia demuestra que el acceso a disco se hace con mucha mayor probabilidad sobre una pequeña fracción de los bloques. Si conseguimos identificar cuales son estos bloques más probables y tenerlos en memoria aumentaremos considerablemente el rendimiento del sistema.

En primer lugar, Forth reserva una cantidad de memoria para tener allí unos pocos bloques. Supongamos que es espacio para acomodar a una decena de bloques, nombrados desde B0 hasta B9. En total, 10K de RAM. Para gestionar este espacio, usamos a su vez una decena de palabras (2 bytes). Así, cada bloque Bx tiene asignada una palabra con algunas informaciones útiles. En particular, un bit (el 7 en la Figura) indicará si el bloque correspondiente en RAM ha sido cargado con alguno de los bloques de disco, o si está vacío. Otro bit indicará si un bloque cargado desde disco a RAM ha sido modificado o no por el programa. Los bits 0-5 pueden usarse como un contador que indica el número de veces que se ha accedido al bloque. En cada operación de lectura o escritura, este contador se incrementa en una unidad, hasta que alcanza el máximo. Finalmente, el segundo byte contiene el número de bloque que ha sido cargado. Por ejemplo, si en el bloque cuarto de memoria acaba de cargarse el bloque 107 del disco, la cuarta palabra de administración contendrán los valores 128,107. Si sobre la copia en memoria del bloque de disco 107 se



**Figura 7.1** *Caché* de disco.

efectúan cuatro operaciones y a continuación el bloque se marca como modificado, los bytes administrativos contendrán 196,107 (el 196 corresponde al patrón de bits 11000100: el bloque de memoria contiene un bloque de disco, éste ha sido modificado y el número de accesos ha sido de cuatro hasta el momento) <sup>1</sup>.

Con esta introducción es fácil entender cual es la forma en que Forth usa la *caché* de disco. Cuando se requiere leer o escribir sobre un bloque de disco, se busca uno vacío de los diez que se encuentran en RAM. Si se encuentra, se deja allí copia del bloque de disco y se trabaja sobre la misma. Si no se encuentra, es preciso desalojar a un bloque anterior. En ese caso, se consultan los contadores y se localiza aquel que tiene el contador más bajo, es decir, el que se ha usado menos. Si había sido modificado, se reescribe en el disco y a continuación se sobrescribe su copia en RAM con el nuevo bloque. Si no había sido modificado, simplemente se sobrescribe.

### 7.3. Palabras de interfaz

Forth ofrece un reducido número de palabras para gestionar los bloques. La más sencilla es `'list'`, que produce en pantalla un listado del contenido del bloque. Espera en la pila el número de bloque. Como los bloques tienen 1024 bytes, se imprimen como un conjunto de 16 líneas de 64 caracteres por línea. Por ejemplo, si el pequeño vocabulario que en su momento desarrollamos para trabajar con racionales se hubiese guardado en el bloque 18, podríamos consultarlo mediante `18 list`:

---

<sup>1</sup>Esta descripción tiene como objeto fijar ideas, no ilustrar ninguna implementación particular que el autor conozca.

```

18 list $
0 \ palabras para operar con racionales
1 \ (n1 d1 n2 d2 -- n d)
2 : mcd ?dup if tuck mod recurse then ;
3 : reduce 2dup mcd tuck / >r / r> ;
4 : q+ rot 2dup * >r rot * -rot * + r> reduce ;
5 : q- swap negate swap q+ ;
6 : q* rot * >r * r> reduce ;
7 : q/ >r * swap r> * swap reduce ;
8
9
10
11
12
13
14
15
ok

```

La palabra 'block' toma de la pila un número de bloque, busca en la *caché* un hueco, siguiendo la mecánica explicada en la sección anterior, y devuelve en la pila la dirección de comienzo del bloque. En caso de que el bloque ya estuviese en la *caché*, simplemente se devuelve la dirección de comienzo. Con esta dirección, un programa puede editar el bloque, leer su contenido o modificarlo de cualquier forma que se desee.

La palabra 'load' toma una por una las líneas de un bloque y las traspasa al TIB, donde son procesadas de la forma acostumbrada. Si hay definiciones de palabras, son compiladas, si son operaciones en modo de intérprete, ejecutadas. Para el sistema Forth es irrelevante la procedencia de una línea que acaba de ser depositada en el TIB.

Cuando un bloque ha sido modificado, no se marca como tal automáticamente, sino que es el usuario del sistema quien ha de hacerlo mediante la palabra 'update', que no toma argumentos de la pila, sino que actúa sobre el bloque sobre el que se esté trabajando <sup>2</sup>. Ahora bien, los bloques marcados como modificados no son reescritos a disco salvo que hayan de ser desalojados en favor de otros bloques, pero si esta circunstancia no se diese, al terminar la

---

<sup>2</sup>Esta es al menos la implementación que se presenta en *Starting Forth*. Nosotros creemos que esta palabra debería de tomar de la pila un número de bloque.

sesión sus contenidos se perderían. Por este motivo, existe la palabra `'flush'`, que fuerza la reescritura en disco de todos los bloques modificados, al tiempo que marca los bloques en memoria como libres. Si se desea únicamente salvar la *caché* a disco pero mantener disponibles los bloques, en lugar de `'flush'` ha de usarse `'save-buffers'`.

La circunstancia opuesta es que los bloques contenidos en la *caché* no se necesiten y puedan ser descartados. Para eso existe la palabra `'empty-buffers'`, que marca como libres todos los bloques de la *caché*.

Muchos sistemas Forth incluyen un editor de bloques, y en la red pueden encontrarse algunos. Son editores sencillos ya que la edición de una cantidad de texto tan modesta como 1K elimina muchas de las necesidades a las que tienen que hacer frente editores convencionales. Cuando una aplicación requiere varios bloques, puede dedicarse el primero para documentar mínimamente el programa y para incluir una serie de `'load's` que carguen el resto de bloques.

## 7.4. Forth como aplicación

Consideremos cómo usa Forth los bloques cuando se ejecuta como una aplicación sobre un sistema operativo. Nada mejor para ilustrar este punto que describir la implementación que realiza *Gforth*, una de las más populares distribuciones de Forth. Cuando se invoca alguna de las palabras de la interfaz con el sistema de bloques que hemos presentado en la sección anterior, *Gforth* crea un archivo de nombre `blocks.gfb`. El tamaño que se asigna a ese archivo depende del número de bloque mayor que se use durante la sesión. Por ejemplo, después de

```
3 block $ ok
```

suponiendo que aún no existiese `blocks.gfb` se crea con un tamaño de 4K (bloques 0,1,2,3). Al terminar la sesión, el archivo permanece en el disco. En una sesión posterior, la palabra `'use'` permite establecer el nombre que se indique como el archivo que simula el disco Forth. Por ejemplo

```
use blocks.gfb $ ok
```

Ahora, si se requiere una operación con un bloque cuyo número es mayor que el mayor que puede contener `blocks.gfb`, el archivo se expande automáticamente hasta poder acomodar el número de bloque requerido. Mediante este sistema se establece un enlace con aplicaciones Forth diseñadas para el uso de bloques.

Finalmente, resta por discutir la interfaz de Forth con el sistema de archivos nativo del Sistema Operativo sobre el que se ejecute como aplicación. Para este menester, Forth cuenta con un par de docenas de palabras. Para ilustrarlas, comenzaremos con un programa simple que abre un archivo, lee línea a línea e imprime cada línea en pantalla:

```
\ -----
\ lectura de un archivo de texto
\ -----

create mibuffer 256 allot
variable fileid

: lee-y-escribe

s" mani2.tex" r/o open-file      \ fileid ior
0= if fileid ! else drop abort" error e/s" then
begin
  mibuffer 256 fileid @ read-line \ n flag ior
  0<> if abort" error e/s" then   \ n flag
while
  mibuffer swap type cr          \
repeat
fileid @ close-file ;
```

En primer lugar, creamos un *buffer* de nombre `mibuffer`, con un tamaño de 256 *bytes*. El objeto de este espacio es el de albergar cada una de las líneas, a medida que se vayan leyendo. También creamos una variable de nombre `fileid`. En esta variable se guardará un entero devuelto por `open-file` mediante el que podremos en ocasiones sucesivas referirnos al archivo que abrimos. `open-file` espera en la pila la dirección de una cadena con el nombre del archivo, la longitud de dicha cadena y el método que se usará para abrirlo. En cuanto al tercer parámetro, hemos elegido `r/o`, de *read only*. Otras posibilidades son `w/o`, de *write only* y `r/w`, que permite tanto leer

como escribir. Como resultado de `open-file`, quedan en la pila un entero que identificará en lo sucesivo al archivo y otro al que nos referiremos por `ior` (*input output result*). Si no se produce ninguna excepción, como por ejemplo que el archivo no exista o que sus permisos sean incompatibles con el método elegido, `ior` vale 0.

Una vez abierto el archivo, procedemos a leer línea a línea mediante la palabra `read-line`. Esta palabra espera en la pila la dirección y el tamaño máximo de un *buffer* y el entero que identifica al archivo, y deja como resultado el número de caracteres leídos de la línea, una bandera que será `true` mientras no se haya alcanzado el final del archivo y un `ior` que será 0 mientras no se produzca una excepción. El número de caracteres leídos se combina con la dirección del *buffer* y la línea se imprime mediante `type`. Una vez abandonado el bucle, `close-file`, que espera el identificador del archivo en la pila, lo cierra.

`create-file` crea un archivo. Si ya existe, lo reemplaza por otro del mismo nombre, pero vacío. Espera en la pila la dirección de la cadena que contiene el nombre del archivo, la longitud de dicha cadena y un método de acceso. Como resultado, quedan en la pila un identificador de archivo y un `ior`, que si todo fue bien valdrá 0. Si se produjo alguna excepción, `ior` tendrá un valor distinto de cero, y el identificador del archivo tendrá un valor indefinido.

`delete-file` espera en la pila la dirección y la longitud de la cadena que contiene el nombre del archivo, y deja, después de borrarlo, un `ior` que será 0 si todo fue bien.

`read-file` permite leer carácter a carácter de un archivo. Espera en la pila la dirección de un *buffer*, el número máximo de caracteres a leer y un identificador de fichero, y deja el número de caracteres leídos y un `ior`. Si no se produjo excepción, `ior` vale 0. Si se alcanzó el final del fichero antes de leer el número de caracteres indicados, el número de caracteres leídos obviamente no coincidirá con el número de caracteres pedidos.

Cuando se abre un archivo para lectura o escritura y se realizan algunas de estas operaciones, el sistema mantiene un entero largo con la posición del siguiente carácter que será leído o la posición donde el siguiente carácter será escrito. A este entero largo se accede mediante la palabra `file-position`. `file-size` por su parte devuelve un entero doble con el tamaño del archivo. `write-file` es la contraparte para escritura de `read-file`.

Espera en la pila una dirección, un número de caracteres que se encuentran a partir de esa posición y que queremos escribir al archivo y el identificador del mismo. Como resultado, se actualiza si es preciso el tamaño del archivo y también el puntero que indica la posición dentro del mismo. Similar a `write-file` es `write-line`, que espera la dirección y longitud de la cadena que desea escribirse.

# Capítulo 8

## Estructuras y memoria dinámica

### 8.1. Estructuras en Forth

Para un programador C, una de las carencias más evidentes de Forth son las estructuras. Por este motivo, mostraremos en esta sección de qué forma tan sencilla es posible extender el lenguaje para usarlas. Nuestro tratamiento será limitado, ya que vamos a descartar las estructuras anidadas, pero creemos que será ilustrativo y útil.

Ante todo, ¿cómo usarlas? Declaremos un nuevo tipo de dato, de la clase Libro:

```
[struct
  20 field autor
  40 field titulo
  10 field ISBN
  4 field precio
  4 field paginas
struct] Libro
```

Una vez creado el nuevo tipo, es posible declarar una variable de ese tipo con create:

```
create milibro Libro allot
```

El acceso a alguno de los campos de `milibro` es en la forma que se espera:

```
34 milibro precio !
milibro precio @
```

¿Y cómo se extiende el compilador Forth para manejar estructuras? Sorpréndase <sup>1</sup>:

```
0 constant [struct
: field create over , + does> @ + ;
: struct] constant ;
```

En realidad, de las tres líneas sobra una, pero la mantendremos para que el código fuente después sea tan legible como hemos mostrado. La función esencial de un miembro en una estructura es proporcionar un desplazamiento a partir de una dirección base, para indicar el comienzo de dicho miembro. La estructura comienza con desplazamiento cero, de manera que la primera palabra, `[struct` es simplemente una constante, que en tiempo de ejecución, como corresponde a las palabras creadas con `constant`, coloca su valor en la pila. De manera que la llamada a `[struct` deja un 0 en la pila.

A partir de aquí, será preciso ir añadiendo campos. En tiempo de compilación, cada campo ha de almacenar su desplazamiento respecto a la dirección base, y dejar en la pila el desplazamiento para el campo siguiente. Volviendo al ejemplo anterior donde hemos definido un tipo de dato `Libro`, la primera línea deja en la pila un valor 0, que es el desplazamiento del primer campo. A continuación, se deja en la pila el valor 20, tamaño del primer campo, y se invoca a la palabra `field`. Esta palabra compila una nueva entrada en el diccionario de nombre `autor`, y en su campo de datos deja mediante `'over ,'` el valor 0, que es su desplazamiento dentro de la estructura. Quedan en la pila un 0 y un 20, que se suman para dar el desplazamiento del segundo campo. Después, en tiempo de ejecución, la palabra `autor` dejará en la pila el valor 0, y lo sumará a la dirección base, proporcionando un puntero al campo del mismo nombre dentro de la estructura `Libro`. En la siguiente línea, `field` crea una nueva entrada de nombre `titulo`. En tiempo de compilación, el valor 20 dejado en la pila por la llamada anterior se guarda en el campo de datos de `titulo`, y se suman `20 40 +`, dejando en la pila el desplazamiento para el tercer campo. En tiempo de ejecución, `titulo` dejará el valor 20 en

---

<sup>1</sup>Hay muchas formas de implementar las estructuras. Aquí seguimos el tratamiento que hace Stephen Pelc en *Programming Forth*. Nos parece simple y adecuado.

la pila y lo sumará a una dirección base, proporcionando acceso al campo `titulo`. Y así sucesivamente.

Cuando se han declarado todos los campos, el valor que queda en la pila es el tamaño total de la estructura. Finalmente, la palabra `struct` crea una entrada en el diccionario con el nombre del nuevo tipo creado. Al ser el nombre para el nuevo tipo una constante, se le asignará el valor que hubiese en la pila, el tamaño de la estructura, y dejará ese tamaño en la pila en tiempo de ejecución, proporcionando un argumento para que `allot` reserve el espacio necesario.

Ahora se ve que la línea

```
0 constant [struct
```

es en realidad innecesaria. Sería suficiente simplemente dejar un 0 en la pila. Por otro lado, `field` puede tomarse como base para declarar palabras que hagan referencia a tipos específicos. Por ejemplo, para declarar un determinado campo como entero podemos hacer

```
: int cell field ;
```

y ya podemos escribir

```
[struct  
    int cilindrada  
    int potencia  
    int par  
    int rpm  
struct] Motor
```

## 8.2. Memoria dinámica

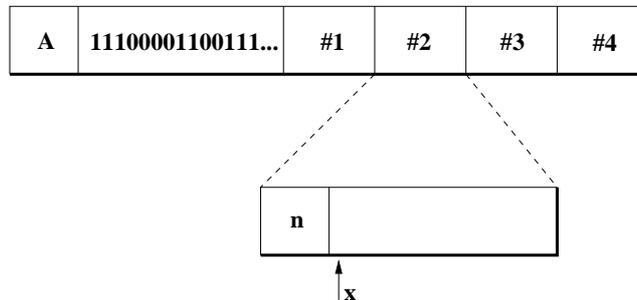
Muchas aplicaciones necesitan hacer uso de memoria dinámica, porque es la forma natural de implementar muchas estructuras de datos, cuyo tamaño no puede estar prefijado. Por ejemplo, un editor de textos desconoce tanto la longitud de cada línea como el número de estas, y para hacer un uso eficiente de la memoria es preciso crear una lista enlazada de punteros con tantos elementos como líneas contenga el texto que se desea editar. A su vez, cada puntero apuntará a un área de longitud variable, que será reservada

en función de la longitud de cada línea. Pero las líneas pueden crecer o menguar, y su número puede aumentar o disminuir, de forma que se precisa un mecanismo para solicitar y liberar porciones de memoria de tamaños distintos.

Existen muchas formas de organizar una porción de memoria y repartir fragmentos entre quienes los soliciten, y aquí son pertinentes muchas de las consideraciones que se hacen al discutir los sistemas de archivos ya que conceptualmente el problema es el mismo. Sin embargo, los accesos a memoria son mucho más frecuentes que los accesos a disco, y se efectúan a mucha mayor velocidad. Hay al menos otras dos diferencias fundamentales; primero, que la cantidad de memoria que se asigna a una variable consta de bloques contiguos y segundo que esa cantidad no suele variar. Esto simplifica las cosas. En nuestra implementación, la memoria se asigna en múltiplos de un tamaño predeterminado que llamaremos  $b$ , de forma que la solicitud de una cantidad de memoria de  $n$  bytes realmente conduce a la asignación de ' $n / b + 1$ ' bloques, salvo cuando  $n$  sea múltiplo exacto de  $b$ , en cuyo caso no es preciso el incremento final. Para saber si un bloque puede asignarse, es preciso comprobar si está o no libre. La mejor forma de almacenar esta información es mediante un mapa de bits. El bit  $z$  a 1 indica que el bloque  $z$  se encuentra asignado.

Cuando un programa solicita memoria para una variable, el gestor de memoria calcula el número de bloques que han de asignarse para satisfacer esa petición, recorre el mapa de bits buscando tantos bits contiguos a cero como bloques ha de asignar y devuelve o bien un valor que indique que la asignación no ha sido posible, o bien la dirección de comienzo de la porción de memoria asignada. En el mapa de bits, la secuencia de bits a cero se cambia por una secuencia de bits a uno.

La liberación de memoria es ligeramente distinta, porque se espera simplemente que el programa libere la memoria de la variable para la que antes hizo la solicitud, pero sin indicar el tamaño que fue solicitado entonces. Es evidente que este tamaño ha de ser almacenado en algún sitio, y el lugar más lógico es el comienzo del bloque asignado. Por tanto, cuando un programa solicita memoria, el gestor ha de sumarle a esa cantidad una celda adicional (cuatro bytes en una implementación de 32 bits), reservar la memoria, poner a uno los bits correspondientes en el mapa de bits, escribir en la primera celda de la porción asignada el número de bloques que le pertenecen y devolver al programa la dirección de la segunda celda.



**Figura 8.1** Organización del montículo.

La implementación que presentamos aquí toma una porción del diccionario como montículo <sup>2</sup>, y en él aloja los bloques solicitados más una cabecera que contiene el número de bloques y el mapa de bits. La Figura 8.1 muestra un esquema de la organización del montículo. La sección 'A' contiene el número de bloques de datos. A continuación se aloja el mapa de bits. Finalmente, los bloques de datos. La primera celda de una región que ha sido reservada contiene el número  $n$  de bloques reservados, y 'malloc' devuelve la dirección siguiente, marcada con 'x'.

Con este esquema general, es fácil seguir los comentarios que acompañan al código fuente que cierra este capítulo. Este código incluye una pequeña utilidad llamada 'mem.' que imprime en pantalla el número de bloques de que consta un montículo y el mapa de bits, indicando mediante una 'X' los bloques ocupados y mediante un '-' los bloques libres. Comenzamos definiendo los tamaños en bits para un byte y en bytes para un bloque y a continuación creamos una pequeña utilidad para consultar y modificar bits individuales dentro del mapa de bits. Dentro de este conjunto de palabras, es central 'mcreate', que crea una entrada en el diccionario, reserva espacio para la cabecera y los bytes solicitados y rellena la primera. Después, 'malloc' recorrerá el mapa de bits hasta encontrar una secuencia lo suficientemente larga de bloques como para acomodar una solicitud, devolviendo la segunda celda del bloque reservado (en la primera escribe el número de bloques que se reservan para que puedan en un momento posterior ser liberados).

En la siguiente pequeña sesión ilustramos el uso de 'malloc', 'free' y 'mem.':

variable a

<sup>2</sup>El término corriente en la literatura inglesa es *heap*.

```

variable b
variable c
variable d
400 mcreate monticulo $ ok
monticulo mem. $
25
-----
ok
50 monticulo malloc a ! $ ok
60 monticulo malloc b ! $ ok
30 monticulo malloc c ! $ ok
100 monticulo malloc d ! $ ok
monticulo mem.
25
XXXXXXXXXXXXXXXXXXXXX-----
ok
c @ monticulo free monticulo mem. $ ok
25
XXXXXXXXX---XXXXXX-----
ok
a @ monticulo free monticulo mem. $ ok
25
----XXXX---XXXXXX-----
ok
1000 monticulo malloc . $ 0 ok

```

Las variables 'a', 'b', 'c' y 'd' serán usadas como punteros, para almacenar las direcciones devueltas por 'malloc'. A continuación, se crea un montículo para albergar 400 bytes, que son contenidos en 25 bloques. Una primera llamada a 'mem.' muestra el mapa de bits vacío. A continuación se realizan algunas llamadas a 'malloc' para asignar memoria a los punteros, en las cantidades que se indican. Como resultado, 17 bloques son asignados. Finalmente, se libera la memoria asignada a los punteros 'a' y 'c' y se visualiza el nuevo estado del mapa de bits. Una petición que no puede satisfacerse devuelve un valor 0 a la pila.

*Gforth* compila el vocabulario para asignación dinámica de memoria en 1568 bytes. Una cantidad ciertamente modesta que apenas se incrementa compilando las palabras de la primera sección para tratar con estructuras. Con ese coste tan pequeño es posible crear complejas estructuras de datos

que se adapten de forma natural a cada problema en particular: listas, colas, árboles... Esta es una buena prueba de la flexibilidad de Forth.

```
\ gestor de memoria dinamica
\ version: 1
\ autor: javier gil

\ -----
\ El tamaño del byte se fija en 8 bits, y el
\ tamaño del bloque en 16 bytes
\ -----
8 constant tbyte
16 constant tbloque

\ -----
\ Dado un numero de unidades n y un tamaño
\ de bloque m, determina el numero de bloques
\ minimo para contener completamente a las n
\ unidades
\ -----
: nbq ( n m--b)
  /mod swap if 1+ then ;

\ -----
\ Obtiene el numero de bloques necesarios para
\ contener a n bytes de memoria. Tiene en cuenta
\ si el tamaño solicitado es multiplo exacto
\ del tamaño del bloque
\ -----
: nbloques ( n--m)
  tbloque nbq ;

\ -----
\ Dado un numero de bloques, calcula el numero
\ de bytes necesarios para alojar el mapa de
\ bits
\ -----
: tmapa ( n--m)
  tbyte nbq ;
```

```

\ -----
\ Crea un espacio sobre el que se realizara la
\ asignacion y liberacion dinamica de memoria.
\ Este espacio consta de tres secciones: A) el
\ numero de bloques que se van a gestionar;
\ B) un mapa de bits y C) los bloques de datos.
\ mcreate reserva la memoria y rellena la
\ seccion A) con el valor que sea y la seccion
\ B) con ceros. La sintaxis es 'n mcreate X'
\ donde 'n' es el numero de bytes que se
\ quieren gestionar. En tiempo de ejecucion,
\ X devuelve la direccion de la seccion A)
\
\   +-----+-----+-----+
\   |  A    | Mapa de bits | Datos      |
\   +-----+-----+-----+
\
\
\ -----
: mcreate ( n-- )
  create nbloques
    dup ,                \ seccion A
    dup tmapa 0 do 0 c, loop \ seccion B
    tbloque * allot      \ seccion C
  does> ;

\ -----
\ Devuelve verdadero o falso segun que el
\ bit b del byte n se encuentre a 1 o 0
\ -----
: bit? ( n b --flag )
  1 swap lshift dup rot and = ;

\ -----
\ Activa el bit b del byte n
\ -----
: bit+ ( n b--n' )
  1 swap lshift or ;

\ -----
\ Desactiva el bit b del byte n

```

```

\ -----
: bit- ( n b--n')
  tuck bit+ swap 1 swap lshift xor ;

\ -----
\ Dada la direccion dir de un mapa de bits y
\ el numero b de un bit dentro de ese mapa,
\ coloca en la pila el byte n que contiene el
\ bit b, y el numero b' del bit dentro de ese
\ byte
\ -----
: byte> ( dir b --n b')
  dup 8 mod >r 8 / + @ r> ;

\ -----
\ Cuando es preciso modificar un bit dentro de
\ un mapa de bits, es preciso preservar la
\ direccion del byte que contiene el bit, al
\ objeto de reescribirlo. dirbyte> toma de la
\ pila la direccion del mapa de bits dir y el
\ bit b que es preciso modificar, y deja en la
\ pila la direccion dir' del byte que contiene
\ al bit que hay que modificar, el byte mismo
\ y el bit b' dentro del byte
\ -----
: dirbyte> ( dir b -- dir' n b')
  dup 8 mod >r 8 / + dup @ r> ;

\ -----
\ mapbit? indica si el bit b del mapa de bits
\ que comienza en la direccion dir esta activo
\ o inactivo
\ -----
: mapbit? ( dir b --flag)
  byte> bit? ;

\ -----
\ Esta funcion accede al bit b del mapa de bits
\ que comienza en la direccion dir y lo pone a
\ 1
\ -----

```

```

: mapbit+ ( dir b --)
  dirbyte> bit+ swap ! ;

\ -----
\ Esta funcion accede al bit b del mapa de bits
\ que comienza en la direccion dir y lo pone a
\ cero
\ -----
: mapbit- ( dir b --)
  dirbyte> bit- swap ! ;

\ -----
\ malloc toma como argumentos la direccion del
\ area de memoria compartida y el numero de bytes
\ que se solicitan, y devuelve 0 si no fue posible
\ realizar la asignacion, o la direccion donde
\ comienza la memoria asignada. Cuando malloc
\ hace una reserva, reserva en realidad una celda
\ adicional al principio donde almacena el numero
\ de bloques reservados, y devuelve la direccion
\ de la celda siguiente. De esta forma free
\ podra efectuar la liberacion mas tarde. Llamare
\ A a la direccion de la primera celda,
\ B a la direccion del mapa de bits, C
\ al inicio de los bloques y X al bloque
\ cuya direccion ha de devolverse. N es el maximo
\ numero de bloques, tal y como esta escrito en
\ la direccion A; c es un contador; b indica
\ bytes y n bloques
\ -----
: malloc ( b A -- 0 | X )
  swap cell+ nbloques swap          \ n A
  dup @                              \ n A N
  0 swap                             \ n A c=0 N
  0 do                                \ n A c=0
    over cell+                       \ n A c=0 B
    I mapbit? if                     \ n A c=0
      drop 0                          \ n A c=0
    else
      1+ rot 2dup = if                \ A c+1 n
        drop                          \ A n

```

```

                2dup                \ A n A n
                over @ tmapa        \ A n A n tmapa
                rot cell+ +        \ A n n C
                swap I swap        \ A n C I n
                - 1+ tbloque * + -rot \ X A n
                rot 2dup ! cell+ -rot \ X+cell A n
                swap cell+ swap    \ X+cell B n
                I swap - 1+ I 1+ swap do \ X+cell B
                    dup I mapbit+
                loop
                drop                \ X
                unloop exit
            else
                -rot
            then
                then
            loop
            3 ndrop 0 ;

```

```

\ -----
\ free toma dos direcciones: la de la region
\ que se quiere liberar y la del monticulo que
\ se esta gestionando; solo tiene que poner a
\ cero los bits que correspondan en el mapa
\ de bits, borrando n de ellos a partir del
\ numero m, es decir, desde n hasta n+m-1.
\ Aqui enseguida se hace sobre X cell -, para
\ apuntar a la cabecera que contiene el numero
\ de bloques asignados al puntero a X
\ -----

```

```

: free ( X A --)
    swap cell - over                \ A X A
    dup @ tmapa                    \ A X A tmapa
    + cell+                        \ A X C
    over @                          \ A X C n
    -rot - tbloque /               \ A n m
    cr .s cr
    tuck + swap do                 \ A
        dup cell+ I mapbit-
    loop drop ;

```

```

\ -----
\ Imprime el numero de bloques gestionados en
\ A y el mapa de bits, marcando con una X los
\ bloques ocupados, y con un - los bloques libres
\ -----
: mem. ( A --)
  dup @ swap cell+ swap          \ B n
  cr dup . cr
  0 do
    dup I mapbit? if
      [char] X emit
    else
      [char] - emit
    then
  loop drop cr ;

```

# Capítulo 9

## Algunas funciones matemáticas

### 9.1. Distintas opciones

En este capítulo presentaremos algunas funciones matemáticas comunes, como potencias, raíces, funciones trigonométricas, exponenciales y logarítmicas. Veremos que es posible en efecto calcular funciones de este tipo aún usando únicamente aritmética entera. Pero con un rango limitado para los números enteros es preciso estudiar la mejor forma de implementar este vocabulario. En principio, disponemos de tres opciones: 1) encontrar algoritmos inteligentes e ingeniosos; 2) usar algoritmos ni inteligentes ni ingeniosos y como consecuencia probablemente poco eficientes, pero que funcionen y 3) recurrir a tablas con valores precalculados que permitan realizar interpolación.

Muchos sistemas Forth funcionan sobre procesadores que sólo realizan operaciones con enteros. En estos casos, las operaciones aritméticas que se esperan son sencillas, pero pueden ser suficientes para muchas aplicaciones. Es el caso de la mayoría de los sistemas incrustados. Incluso en sistemas de sobremesa, aplicaciones populares pueden estar basadas exclusivamente en aritmética entera: procesadores de texto, navegadores, multimedia, etc.

No obstante, no han de cerrarse los ojos al hecho de que los procesadores modernos están diseñados específicamente para realizar operaciones con números reales, de que estas operaciones se realizan a una velocidad comparable, cuando no mayor, a las operaciones con enteros y a que existe un formato para representar números reales casi universalmente aceptado. En estas condiciones y teniendo Forth como una de sus principales características su proximidad a la máquina, no parece aconsejable ignorar las posibilida-

des de otras implementaciones Forth (*Gforth*, por ejemplo) para operar con números reales.

## 9.2. Sólo enteros

### 9.2.1. Factoriales y combinaciones

La función factorial ya fue presentada anteriormente, cuando introdujimos la recursión. A modo de referencia, reproducimos aquí dos implementaciones, una recursiva y otra iterativa:

```
: fct-i ( n--n!)
  1 2dup
  swap 1+ swap do
    over * swap 1- swap
  loop ;
: fct-r ( n--n!)
  dup 0= if drop 1 else dup 1- recurse * then ;
```

la función factorial aparece frecuentemente en matemática discreta, en teoría de probabilidad y combinatoria. A modo de ejemplo, mostramos una implementación para los coeficientes binomiales. Si disponemos de un conjunto de  $n$  elementos, el número de formas distintas de tomar  $m$  de ellos, sin importar el orden, es

$$C_n^m = \frac{n!}{m!(n-m)!} \quad (9.1)$$

La única precaución consiste en no efectuar en primer lugar la multiplicación del denominador, para evitar un resultado que pudiese exceder el rango de los enteros en la implementación de Forth que se esté usando:

```
: c(n,m) ( n m--n!/(m!(n-m)!)
  2dup -
  fct swap fct
  rot fct
  swap / swap / ;
```

## 9.2.2. Raíz cuadrada

El algoritmo obvio para encontrar la raíz cuadrada de un número  $n$  consiste en tomar los enteros sucesivos, elevarlos al cuadrado y comprobar que el resultado es menor que el número cuya raíz deseamos calcular. El primer entero para el cual esta condición no se cumple es una aproximación por exceso a la raíz buscada. Por ejemplo, la raíz aproximada de 10 es 4, la de 26 es 6 y así sucesivamente. Obviamente, deseamos un resultado mejor, pero obtengamos esta primera aproximación y veamos después cómo mejorarla. La palabra 'rc0' toma un entero de la pila, y deja ese mismo entero y una aproximación a su raíz:

```
: rc0 ( n--n i)
  dup dup 1 do
    I dup *
    over >= if
      drop I
    unloop exit
  then
loop ;
```

Ocupémonos ahora de mejorar este valor. Para ello, consideremos el desarrollo en serie de primer orden que relaciona el valor  $f(x)$  de una función en un punto con el valor en otro punto cercano  $x_0$ ;

$$f(x) \sim f(x_0) + f'(x_0)(x - x_0) + \dots \quad (9.2)$$

En particular, para la función raíz cuadrada:

$$\sqrt{x} \sim \sqrt{x_0} + \frac{1}{2\sqrt{x_0}}(x - x_0) + \dots \quad (9.3)$$

Si llamamos  $n$  e  $i_0$  a los parámetros que han quedado en la pila después de la ejecución de 'rc0', es claro que  $\sqrt{x_0} = i_0$ , que  $x_0 = i_0^2$  y que  $x = n$ . Por tanto, una aproximación a la raíz buscada es

$$i_1 = \frac{1}{2} \left( i_0 + \frac{n}{i_0} \right) \quad (9.4)$$

Ahora bien, hemos obtenido una mejor aproximación  $i_1$  a partir de una aproximación anterior  $i_0$ , pero el procedimiento por el cual lo hemos conseguido es independiente del procedimiento por el que se obtuvo en primer lugar  $i_0$ . Por consiguiente, tenemos un algoritmo iterativo para obtener sucesivas aproximaciones para la raíz buscada:

$$i_k = \frac{1}{2} \left( i_{k-1} + \frac{n}{i_{k-1}} \right) \quad (9.5)$$

Serán útiles ahora las palabras que desarrollamos hace algunos capítulos para operar con números racionales. La palabra 'rc' toma las representaciones racionales de  $n$  e  $i$  que 'rc0' dejó en la pila y a partir de ahí calcula una mejor aproximación. Puesto que el procedimiento puede iterarse, dejaremos en la pila también el valor de  $n$  original.

```
: rc ( n 1 i 1-- n 1 j 1 )
  2over 2over
  q/ q+ 1 2 q* ;
```

Por ejemplo:

```
98 rc0 .s $ <2> 98 10 ok
1 tuck rc .s $ <4> 98 1 99 10 ok
rc .s $ <4> 98 1 19601 1980 ok
rc .s $ <4> 98 1 768398401 77619960 ok
```

En la primera línea, obtenemos la primera aproximación a  $\sqrt{98}$ , que es 10. En la segunda línea, dejamos las representaciones racionales de 98 y 10 (98/1 y 10/1), y obtenemos una mejor aproximación a la raíz buscada (99/10). Sucesivas llamadas a 'rc' proporcionan aún mejores aproximaciones: 19601/1980 y 768398401/77619960. Para la mayoría de las aplicaciones prácticas, la tercera aproximación es más que suficiente. Además, aparece el problema inevitable de trabajar con representaciones racionales: el crecimiento de numerador y denominador, que rápidamente pueden exceder el rango que la implementación conceda a los números enteros. Es evidente entonces que el uso de aritmética entera para el cálculo de raíces cuadradas ha de hacerse examinando cuidadosamente las condiciones en que el algoritmo será aplicado. No obstante, sea aritmética de 32 o aritmética de 64 bits, la mera existencia de un rango limitado es una molestia que sólo puede subsanarse implementando, a un coste computacional muy alto, aritmética de cadenas. Esa es la solución cuando la precisión es la consideración fundamental.

### 9.2.3. Seno, coseno y tangente

Puesto que la función  $\tan(x)$  se obtiene a partir de las funciones  $\sin(x)$  y  $\cos(x)$ , nos centraremos en las dos primeras. Es conocido el desarrollo en serie de potencias en un entorno de 0:

$$\sin(x) \sim x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots \quad (9.6)$$

El problema es que la aproximación anterior es tanto peor cuanto más nos alejamos del origen. Cabría entonces buscar un punto intermedio de un intervalo suficiente para calcular la función para cualquier argumento. Por ejemplo, conocido el valor de la función en el intervalo  $[0, \pi/4]$ , la fórmula del ángulo doble nos lo proporciona en el intervalo  $[0, \pi/2]$  y de ahí lo podemos obtener para cualquier otro valor del argumento. Pero si elegimos un punto del intervalo  $[0, \pi/4]$  para efectuar el desarrollo en serie, obtendremos finalmente un polinomio de grado tres, pero con coeficientes distintos. Entonces, se puede plantear el problema como el de la obtención de los coeficientes del polinomio

$$p(x) = ax + bx^3 \quad (9.7)$$

tales que sea mínima la integral

$$J = \int_0^{\pi/4} (\sin(x) - ax - bx^3)^2 dx \quad (9.8)$$

Las dos ecuaciones

$$\frac{\partial J}{\partial a} = 0 \quad (9.9)$$

$$\frac{\partial J}{\partial b} = 0 \quad (9.10)$$

proporcionan las dos incógnitas  $a$  y  $b$ , de manera que obtenemos

$$p(x) = 0,999258x - 0,161032x^3 \quad (9.11)$$

En el intervalo  $[0, \pi/4]$ ,  $p(x)$  puede sustituir a la función  $\sin(x)$  con un error siempre inferior al 0.1 %, y ésta es una aproximación muy buena, siempre que no se pierda de vista que no es más que una aproximación.

El mismo razonamiento es válido para la función coseno, que puede aproximarse mediante el polinomio

$$q(x) = 0,998676 - 0,478361x^2 \quad (9.12)$$

En este caso, el error relativo se mantiene en torno al 0.1% en la mayor parte del intervalo, pero crece hasta el 0.5% cerca del extremo superior. Dependiendo de la aplicación, podría ser necesario el cálculo de un término adicional para  $q(x)$ , del tipo  $cx^4$ . Ahora bien, los coeficientes que aparecen en estas aproximaciones polinómicas son números reales, que será preciso aproximar por racionales. Entonces aparece una disyuntiva: si los coeficientes se aproximan muy bien tomando números grandes para el numerador y el denominador, el cálculo de las potencias de  $x$  puede fácilmente desbordar el rango de los enteros. Si la aproximación para los coeficientes se realiza con fracciones cuyos numeradores y denominadores sean pequeños, se resiente la precisión de los cálculos. Como siempre, el programador deberá hacer una evaluación cuidadosa de las condiciones en que aplicar estos procedimientos. Por ejemplo, la sustitución de  $p(x)$  por el polinomio

$$p(x) = x - \frac{4}{25}x^3 \quad (9.13)$$

puede ser suficiente para un programa de trazado de rayos, pero yo no la usaría en un programa de agrimensura. Para terminar la discusión, existe la posibilidad de cargar una tabla de valores de la función que se desee, y efectuar interpolación lineal, cuadrática, cúbica...

#### 9.2.4. Exponencial

El problema de la función exponencial es que crece más rápido que cualquier función polinómica; como además no es periódica, no es posible centrarse en ningún intervalo en particular. Otro efecto de este rápido crecimiento es que se alcanza pronto el límite para los enteros. Si en una implementación de 32 bits el máximo entero que puede representarse es el número  $2^{32}$ , esto significa que el máximo entero  $n$  del que puede calcularse su exponencial es aquel para el que

$$e^n = 2^{32} \quad (9.14)$$

de donde se sigue que

$$n = 32 \ln 2 \sim 22 \tag{9.15}$$

Podemos dividir el intervalo  $[0, 22]$  en cierto número de subintervalos, y buscar una aproximación polinómica para cada intervalo. Entonces, el argumento de la exponencial (entero) puede usarse para indexar una tabla de punteros a la aproximación que corresponda según el intervalo al que el argumento pertenezca. Pero aún así tendremos dificultades. Por ejemplo

$$e^{21} - e^{20} \sim 833 \times 10^6 \tag{9.16}$$

lo que resulta un rango muy grande para realizar un ajuste sencillo. Estas mismas consideraciones pueden hacerse para la función logarítmica. En ambos casos, el problema no puede formularse y resolverse con un mínimo de generalidad y será preciso en cada problema en particular hacer un análisis y obtener la aproximación que conjugue suficiente precisión, simplicidad y adecuación al rango de números enteros que ofrezca el sistema.

### 9.3. Números reales

Las implementaciones de Forth que ofrecen operaciones con reales suelen ofrecer el conjunto de palabras definidas en la norma ANSI. En general, estas palabras se distinguen por el prefijo 'f', e incluyen funciones matemáticas y operadores lógicos. El intérprete identifica a los números reales por el formato  $[+|-]d*[e|E][+|-]d*$ . Por ejemplo: 628e-2, -31415E-4, 21e4 etc. Existe también una palabra 'f.' para imprimir reales.

Hay alrededor de setenta palabras para tratar con reales, y la mayoría de ellas son la contrapartida de las palabras para enteros. La razón es que aunque es corriente que tanto enteros como reales de precisión simple tengan 32 bits, pudiera no ser este el caso. Por eso existen dos pilas separadas: una para enteros y otra para reales. En los párrafos siguientes hacemos un repaso sumario del vocabulario para reales.

'fconstant', 'fvariable' y 'fliteral' tienen la misma función que sus contrapartidas para enteros.

'f.', 'fe.' y 'fs.' son tres versiones distintas para imprimir reales. Veámoslas en acción:

```
23e-8 $ ok
fdup f. cr $ 0.00000023
ok
fdup fe. cr $ 230.000000000000E-9
ok
fdup fs. cr $ 2.30000000000000E-9
```

'set-precision' establece el número de dígitos significativos usados por cualquiera de las tres funciones anteriores. 'precision' devuelve este número.

'fconstant', 'fvariable' y 'fliteral' hacen la misma función que su homólogas para enteros.

'f!' y 'f@' sirven para guardar en y recuperar de una dirección dada un número real.

'd>f' convierte un entero doble en número real, y 'f>d' convierte un real en entero doble, depositando el resultado en ambos casos en la pila correspondiente.

'fdepth', 'fdrop', 'fswap', 'fdup', 'frot' y 'fover' permiten manipular la pila de reales.

Las funciones matemáticas básicas son 'f\*', 'f\*\*' para elevar un número a una potencia, 'f+', 'f-', 'f/', 'fabs', 'floor' redondea al entero inferior, 'fmax', 'fmin', 'fnegate', 'fround' redondea al entero más próximo, 'fsqrt'.

Algunos condicionales son 'f<', 'f0<' y 'f0='.

Para calcular funciones trascendentes tenemos 'facos', 'facosh', 'falog', 'fasinh', 'fatan', 'fatan2', 'fatanh', 'fcos', 'fexp', 'fln', 'flog', 'fsincos', 'fsinh', 'ftan', 'ftanh' ...

'float+' y 'floats' son los equivalentes de 'cell+' y 'cells'.

Esta no es una lista exhaustiva, pero sí representativa. Consulte la documentación de su sistema para descubrir funciones adicionales. En caso de duda con alguna de las funciones anteriores, lo mejor es experimentar.

# Capítulo 10

## Lézar: en busca del espíritu

### 10.1. La cuestión

Charles Moore, el descubridor de Forth, ha sentido durante toda su vida que el camino correcto era el de la simplicidad. Forth es simple. Por eso Forth es entendible. Cuando creció y se sintió la necesidad de normalizarlo, se apartó de ese espíritu, se hizo más complejo. Como muchos, el autor de este libro aprendió sobre un sistema ANSI Forth, concretamente sobre *Gforth*.

La versión *Gforth 0.5.0* para DOS/Windows, por ejemplo, contiene más de doscientos archivos, y el ejecutable está alrededor de los 280k. Aunque es un sistema bueno, estable y completo, uno siente que eso no es Forth, que de alguna forma traiciona el espíritu. De ahí surge la pregunta de qué es exactamente lo que hace a Forth ser Forth. Hay unas pocas cosas: su carácter interactivo, su extensibilidad, el modelo de máquina virtual basado en dos pilas y unas pocas palabras que le confieren cierta idiosincrasia, como `create` y `does>`, `immediate` y `postpone` y alguna más.

Por eso consideré interesante escribir una interpretación personal de Forth, no tanto para disponer de un sistema como para tratar de recuperar lo más fielmente posible ese espíritu que hace tan atractivo al lenguaje. De ahí surgió *Lézar*.

### 10.2. El nombre

El puerto de Lézar se encuentra en el parque natural de la sierra de Castril, en la provincia de Granada. Remontando el río Castril alcanzamos su nacimiento, y allí comienza una ascensión que culmina en un collado desde el

```

'I trust a straigh stick to support my weight along
its length (though not as a lever). I don't trust
charmingly gnarled or high-tech telescoping sticks.
I don't want a gripo as I'm always sliding my hand
along the shaft to adjust height. Two sticks are
too many.'

```

Charles Moore

```

#####  #####  #####  #####  #    #
#        #    #  #    #    #    #    #
####    #    #  #    #    #    #    #
#        #    #  #####  #    #####
#        #    #  #    #    #    #    #
#        #####  #    #    #    #    #

```

L E Z A R - 2006

\$ ■

**Figura 10.1** Aspecto de *Lézar* en ejecución.

que se accede al barranco del puerto de Lézar. Es un lugar solitario, austero y duro. Por el fondo discurre un riachuelo de aguas claras, entre la hierba verde que en primavera alcanza a la rodilla. El suelo está cubierto de huesos blancos de animales que acabaron muriendo allí. Cerrado por las moles del Tornajuelos, el Buitre y el Caballo, desde el cielo vigilan permanentemente los buitres. Es un lugar especial. No hay muchos árboles y dominan las rocas desnudas, que con frecuencia adoptan formas caprichosas. En invierno el viento hiela. En verano el Sol abrasa. Es naturaleza sin disfraces y el montañero que llega allí lo hace para participar de ese espíritu. El mismo que puede percibirse en Forth. De ahí el nombre que he elegido para el sistema.

### 10.3. Características generales de Lézar

El sistema consta únicamente de dos archivos, un ejecutable `a.out` y un archivo de texto `lezar.f` que contiene una colección de palabras que extienden ese núcleo. El núcleo es muy reducido, apenas 23k. Contiene las palabras primitivas, el intérprete de líneas, el compilador, algunas herramientas de depuración y la interfaz con el sistema de archivos del sistema operativo.

Está escrito en C. El archivo de texto contiene como se ha dicho las extensiones del núcleo, con funciones para el manejo de la pila, aritméticas, lógicas, entrada y salida, cadenas y una extensión para aritmética con números racionales.

*Lézar* no es ANSI, no ofrece aritmética de doble precisión, ni operadores mixtos, ni aritmética de punto flotante. No son esenciales. Hay dos palabras poco usadas que no se encuentran en *Lézar*: **reveal** y **recurse**. En ambos casos, porque no son necesarias. Una palabras es accesible desde el mismo momento en que está siendo creada su entrada en el diccionario. De esta forma, en la definición de una palabras puede usarse esa misma palabra.

El bucle principal tiene este aspecto:

```
while (!salir){
    leer_cadena();
    dividir_cadena();
    interpretar_cadena()
}
```

La lectura de las cadenas puede hacerse desde teclado o desde archivo. Existe una bandera que lo indica, y que es consultada por `leer_cadena()`. En cuanto a `dividir_cadena()`, procede de una vez. Es decir, no se extrae una palabras cada vez y se pasa a la siguiente función, sino que se identifican todas las palabras de la cadena y a continuación se intrepentan una tras otra. Para esto, existe un vector de punteros a cadena. La dimensión de este vector limita el máximo número de palabras que puede contener una línea. 28 es una cantidad razonable. Si la línea contiene menor número de palabras, a los punteros sobrantes se les asigna el valor NULL. Así, hay un puntero al principio de cada palabra, y el primer espacio en blanco tras cada palabras se sustituye por un carácter 0. Finalmente, la cadena es interpretada. La función `interpretar_cadena()` consulta una bandera para saber si se encuentra en modo de intérprete o en modo de compilador. La palabra `':` crea una nueva entrada en el diccionario y pasa a modo de compilación. La palabra `;` termina la compilación y pasa a modo intérprete.

## 10.4. El diccionario

La estructura del diccionario es muy sencilla. Al arrancar, el núcleo pide al sistema operativo anfitrión espacio para construir el diccionario. En reali-

dad, el diccionario está compuesto por dos espacios de memoria disjuntos y distintos. Uno contiene sólo las entradas del diccionario y los datos del programa. Otro, el código asociado a cada entrada del diccionario. El primero es de lectura/escritura, el segundo es inaccesible tanto para lectura como para escritura. Desde el punto de vista de los programas, existe un espacio de memoria, desde 0 hasta el tamaño máximo que el núcleo solicitó al sistema anfitrión. Para el núcleo, estas direcciones son desplazamientos a partir de la primera dirección del espacio concedido por el anfitrión.

La estructura del diccionario es muy sencilla. Contiene en primer lugar un puntero a la última entrada. A continuación, un byte para indicar si la palabra es `immediate`. Como para este menester sólo se precisa un bit, los siete restantes quedan de momento sin usar. En un futuro, podrían usarse para añadir alguna funcionalidad. Por ejemplo, para ocultar o hacer visible a voluntad alguna palabra o sección del diccionario. Tras el byte `immediate` viene el nombre de la palabra. Tradicionalmente, se usa un byte para indicar la longitud y a continuación se coloca el nombre. Nosotros usamos una cadena terminada en 0. Después del nombre está el puntero al código asociado a la palabra. `create` no asigna más espacio que para los campos que hemos descrito. Palabras como `variable`, `constant` o `vector` se ocuparán de reservar el espacio pertinente mediante `allot`.

Se mantienen dos punteros: uno a la primera posición libre en la sección del diccionario donde se registran las entradas y otro a la primera posición libre en la sección que contiene el código.

## 10.5. El código y su ejecución

*Lézar* compila bytecode. Cada palabra perteneciente al núcleo tiene un bytecode asociado. Hay valores reservados para indicar que a continuación viene un entero, una cadena o la dirección de una función a la que se llama. Puesto que el núcleo contiene alrededor de sesenta palabras, sería muy ineficiente identificar los bytecodes mediante un largo `if . . . else`. Por ese motivo, cada bytecode indexa un vector de punteros a función. De esta forma cada bytecode se ejecuta en tiempo constante. Ejecutar una función entonces es tan sencillo como:

```

int ejecutar(int n)
{
    int w=0;
    EXIT=0;
    PXT=C+n;
    marca=tope_r;

    while(!EXIT && !w){
        w>(*F[*PXT])();
    }
    return(0);
}

```

PXT es un puntero a carácter. Cuando se desea ejecutar una palabra cuyo código comienza en la posición *n* de la sección de código (para el programa), primero se le asigna el valor  $PXT=C+n$ , donde *C* es la dirección (para el núcleo) donde comienza la sección de código del diccionario. Una vez que PXT apunta al primer byte del código asociado a la función, se guarda el valor en ese momento del tope de la pila de retorno. Si una función no contiene llamadas a otras funciones, sino sólo llamadas a funciones intrínsecas al núcleo, no habrá ningún problema. Si una función contiene llamadas a otras funciones, que pueden contener llamadas a otras y así sucesivamente, es preciso saber, cuando se encuentra el final del código de una función si se ha acabado de ejecutar la función que fue llamada en primer lugar o si simplemente se sale del nivel más profundo de anidamiento. En este segundo caso, se recupera de la pila de retorno la dirección de vuelta dentro de la función que llamó a la que ahora termina. Cuando se declara una palabra, se finaliza su compilación mediante `;`. Este `;` tiene asociado un byte code que se compila. Este bytecode indexa en el vector de punteros a función un puntero asignado a la función siguiente:

```

int ejct_punto_coma()
    if (marca==tope_r){
        EXIT=1;
    } else
    {
        PXT=(char *)(C+ret[tope_r-1]);
        --tope_r;
    }
    return(0);
}

```

donde `ret` es un vector de enteros donde se implementa la pila de retorno.

Así, si la primera condición es cierta, se ha acabado de ejecutar la función que se llamó originariamente. Si no es cierta, significa que sólo se ha salido de un nivel de anidamiento, y se toma de la pila la posición por donde ha de continuar la ejecución. La segunda mitad de la historia es qué hacer cuando se encuentra el bytecode que indica que a continuación viene la dirección de la función a la que ha de saltarse desde la actual:

```
int ejct_llamar_funcion()
{
    int *q;
    /* guardar direccion de vuelta */
    ret[tope_r]=(int)PXT-(int)C+sizeof(int)+1;
    ++tope_r;
    /* saltar a la nueva funcion */
    q=(int *) (PXT+1);
    PXT=C+(*q);
    return(0);
}
```

## 10.6. Palabras intrínsecas

Este es el conjunto de palabras implementadas en el núcleo:

```
+ - * / mod and or = < > dup
drop swap >r r> : , create
does> immediate postpone [ ]
tick execute if else then
do loop +loop begin while
repeat I break exit key char
emit @ c@ ! c! c, date time
open-file close-file parse
read-line write-line channel
restart rebuild ." allot M N
. .s depth dd dc mark back
```

Como se ha dicho, *Lézar* no es ANSI, así que algunas de las palabras requieren explicación.

`dd` y `dc` son herramientas de depuración, que muestran por pantalla respectivamente una porción de memoria de la sección de lectura/escritura o de la sección de código. Toman como argumento la dirección a partir de la cual hacer el volcado y el número de bytes de que éste constará.

`restart` limpia el diccionario, dejándolo completamente limpio, vacío. Por el contrario, `rebuild` limpia el diccionario pero a continuación lo reconstruye con las extensiones del archivo `leazar.f`.

`M` deja en la pila la primera posición libre de la sección de código del diccionario, mientras que `N` hace lo propio con la sección de lectura/escritura donde se registran las entradas y donde se escriben y leen los datos de los programas. La dirección de comienzo del código de una función se obtiene mediante `tick`, más visible que `'`.

A diferencia de otras implementación, existe una única palabra para indicar una cadena literal, y esta palabra es `."`. En modo intérprete, la cadena se coloca en el área de lectura/escritura, a partir de la primera posición libre indicada por `N`. Allí puede procesarse, imprimirse o desde allí copiarse a algún lugar seguro. Por el contrario, en modo de compilación la cadena se compila en la sección de código, dentro de la palabra en cuya definición aparezca. Finalmente, en tiempo de ejecución, cuando se encuentra el bytecode que indica que a continuación se encuentra una cadena literal, la función encargada de ejecutar el bytecode toma la cadena y la coloca en el espacio del diccionario destinado a lectura/escritura, a partir de la dirección indicada por `N`.

`break` es la palabra que permite abandonar un bucle `do ... loop` o `do ... +loop` y `exit` termina la función en ejecución, deshaciendo el anidamiento en toda su profundidad.

`parse` permite a un programa extraer palabras del TIB. En realidad, éste no existe como tal. Una línea es cargada en un buffer interno al núcleo, dividida en palabras y luego interpretada. No existe un puntero al TIB al que puedan acceder los programas, ni el TIB es direccionable por estos. Ahora bien, un programa puede solicitar al núcleo palabras individuales con `parse`, que espera en la pila el número de la palabra que desea extraer. Como resultado de la operación, pueden suceder varias cosas. a) que se solicite una palabra por encima del valor 27 (dijimos que 28 palabras en total se consideraba razonable como límite de la implementación); b) que se llame a `parse` pero que la pila se encuentre vacía; c) que el valor en la pila esté dentro del

rango que se espera pero que la línea contenga menos palabras que ese valor y d) que la línea contenga tantas o más palabras que el número indicado. En a) y b) se produce un error y se vacían las pilas; en el caso c) se deja un 0 en la pila; en el caso d) se deja un -1 en la pila, y la palabra se copia a partir de la primera posición libre en el diccionario, donde el programa puede acceder a ella, copiarla a un destino permanente, etc.

`mark` permite guardar en un momento dado el estado del diccionario. En realidad, no se guarda una imagen del diccionario, sino los valores de los punteros que lo gestionan. `back` recupera estos valores. Cuando se desarrollan programas en Forth, se suelen escribir varias versiones de la misma palabra, mientras se depuran errores. Puede entonces que en un momento dado deseemos descartar estas pruebas, que están ocupando espacio, pero no deshacernos de otras palabras que escribimos después de arrancar el sistema y que por tanto aún no están en `lezar.f`. Cuando preveamos que esto puede suceder, una llamada a `mark` guarda los valores de los punteros, y una llamada posterior a `back` elimina del diccionario sólo aquellas palabras que se escribieron después de llamar a `mark`.

Sólo queda hablar de `channel`, pero a esta palabra dedicamos la siguiente sección.

## 10.7. Sistema de archivos

La interfaz con el sistema de archivos es extremadamente simple. No existen archivos binarios. No existe la posibilidad de moverse a voluntad dentro de un archivo abierto. Sólo existen archivos de texto, que pueden abrirse para lectura, escritura o ampliación.

El núcleo mantiene ocho canales que pueden corresponder a otros tantos archivos abiertos para cualquiera de los modos indicados. La palabra `open-file` espera en la pila la dirección de una cadena conteniendo el nombre del archivo que se desee abrir y un entero que indique el modo de apertura: 1 para sólo lectura, 2 para sólo escritura y 3 para ampliación. Este número puede depositarse sin más, o usar las palabras `fr`, `fw` o `fa`, cuya definición es trivial. `open-file` puede encontrar dos tipos de errores: a) que en la pila no se encuentre el número de parámetros requerido o que el modo de apertura esté fuera del rango [1..3] y b) que el archivo, por alguna razón, no pueda abrirse. En el primer caso, se produce un error y por defecto las pilas se

vacían. En el segundo, se deja en la pila un código de error. Tanto si la operación fue exitosa como si se produjo un error del tipo b), quedan en la pila el número de canal y un valor numérico: 0 si la operación se completó con éxito y 1 en caso contrario. Ese número de canal normalmente se asignará a una variable o constante cuyo valor depositaremos en la pila cuando queramos referirnos al archivo abierto.

Cuando se llama a `open-file` el núcleo busca el primero de los ocho canales que se encuentre libre, y si la operación de apertura se realizó con éxito se le asigna un valor numérico al canal, indicando el modo de apertura. La palabra `channel` espera un número de canal en la pila y devuelve su valor, es decir, el modo en el que el canal fue abierto. Por ejemplo

```
." datos.txt" fr open-file .s $ <2> 0 0 ok
0 channel . $ 1 ok
```

La primera línea asigna el canal 0 al archivo `datos.txt`. La segunda comprueba el estado del canal 0. El valor 1 indica que el canal fue abierto para lectura. Normalmente, comprobaremos el código de error de `open-file` y asignaremos a una variable el número del canal:

```
variable fileid
." datos.txt" fr open-file
0= if fileid ! else drop ." error" type then
```

Una vez usado, el archivo se cierra con `close-file`:

```
fileid @ close-file
```

En tanto en cuanto el archivo esté abierto, y suponiendo que el modo de apertura permita escritura, la palabra `write-line` permite escribir líneas. Se espera en la pila la dirección de la cadena y el número del canal donde se escribirá. Se devuelven la dirección de la cadena y un código de error: -1 si la operación se completó con éxito y 0 si no. Por su parte, `read-line` espera en la pila la dirección del buffer donde será colocada la cadena, el número máximo de caracteres que se leerán y el número de canal. Devuelve la dirección del buffer y un código de error: -1 si la lectura fue exitosa y 0 si no.

## 10.8. Extensiones del núcleo

La extensibilidad es característica fundamental de Forth, y de hecho más de la mitad de *Lézar* es una extensión del núcleo. Por supuesto, pueden añadirse más palabras. Cuando el número de éstas sea muy grande, pueden darse conflictos de nombres. Sin embargo, no consideramos característica esencial el manejo de vocabularios, aunque no sería difícil de implementar una gestión de espacios de nombres. En las páginas siguientes mostramos `lezar.f`, y a continuación una recopilación de las extensiones:

```
cell cells rot -rot over nip
tuck 2dup 3dup 2drop ndrop
2swap cls plas 0= ?dup 0< 0>
>= <= != 1+ 1- 2* negate abs
^ fact << >> max min /mod
variable constant +! n->dd
[char] ESC[ at-xy cr cursor^
cursor_ cursor> <cursor
cursor^^ cursor__ cursor>>
<<cursor page <# # #s #>
hold sign type .r fill erase
copy 2c@ compare accept <-X
string strlen upcase? strup
lowcase? strlow strscan used
.time .date channels fr fw
fa q+ q- q* q/ mcd reduce q.
```

Antes de eso, una observación importante: una vez escrito el núcleo y escritas las extensiones, observamos que algunas de las palabras intrínsecas podrían re-implementarse como extensiones, eliminándolas del núcleo. De hecho, una observación detallada nos llevaría a la conclusión de que ciertas palabras del núcleo que en principio no pueden escribirse como extensiones, podrían serlo si dispusiésemos de algunas, dos o tres, palabras de nivel aún más bajo, más simples. Estaríamos entonces aún más cerca del ideal.

```
\ =====
\ tamaño de la celda
\ =====
: cell 4 ;
: cells 4 * ;
```

```

\ =====
\ manipulacion de la pila
\ =====
: rot >r swap r> swap ;
: -rot swap >r swap r> ;
: over >r dup r> swap ;
: nip swap drop ;
: tuck swap over ;
: 2dup over over ;
: 3dup >r 2dup r> dup >r -rot r> ;
: 2drop drop drop ;
: ndrop 0 do drop loop ;
: 2swap >r -rot r> -rot ;
: cls depth dup 0 = if drop else 0 do drop loop then ;

\ -----
\ plas toma una direccion base y un desplazamiento,
\ y deja en la pila la direccion base, el
\ desplazamiento incrementado en uno y la
\ direccion completa ( b d -- b d+1 b+d)
\ -----
: plas 2dup + swap 1 + swap ;

\ =====
\ complementos logicos
\ =====
: 0= 0 = ;
: ?dup dup 0= if else dup then ;
: 0< 0 < ;
: 0> 0 > ;
: >= 2dup > -rot = or ;
: <= 2dup < -rot = or ;
: != = if 0 else -1 then ;

\ =====
\ complementos de aritmetica
\ =====
: 1+ 1 + ;
: 1- 1 - ;
: 2* 2 * ;

```

```

: negate -1 * ;
: abs dup 0< if negate then ;
: ^ dup 0= if drop drop 1 else over swap 1- ^ * then ;
: fact dup 0= if 1+ else dup 1- fact * then ;
: << 0 do 2 * loop ;
: >> 0 do 2 / loop ;
: max 2dup > if drop else nip then ;
: min 2dup < if drop else nip then ;
: /mod 2dup mod -rot / ;

```

```

\ =====
\ variables y constantes
\ =====
: variable create 1 cells allot does> ;
: constant create , does> @ ;
: +! dup @ rot + swap ! ;

```

```

\ =====
\ complementos de entrada/salida
\ =====

```

```

\ -----
\ toma un numero <= 99 y deja en la pila los
\ dos caracteres que lo componen, para imprimirlo:
\ esta funcion es auxiliar para las secuencias
\ ansi
\ -----
: n->dd dup 10 mod 48 + swap 10 / 48 + ;
: [char] char postpone literal ; immediate
: ESC[ 27 emit [char] [ emit ;

```

```

\ -----
\ espera fila y columna en la pila ESC[f;cH
\ -----
: at-xy swap ESC[ n->dd emit emit
    [char] ; emit
    n->dd emit emit [char] H emit ;
: cr 10 emit ;

```

```

\ -----
\ mueve cursor una posicion arriba, abajo,

```

```

\ derecha e izquierda
\ -----
: cursor^ ESC[ [char] 0 emit [char] 1 emit
    [char] A emit ;
: cursor_ ESC[ [char] 0 emit [char] 1 emit
    [char] B emit ;
: cursor> ESC[ [char] 0 emit [char] 1 emit
    [char] C emit ;
: <cursor ESC[ [char] 0 emit [char] 1 emit
    [char] D emit ;

\ -----
\ mueve el cursor un numero de posiciones
\ -----
: cursor^^ 0 do cursor^ loop ;
: cursor__ 0 do cursor_ loop ;
: cursor>> 0 do cursor> loop ;
: <<cursor 0 do <cursor loop ;

\ -----
\ limpia pantalla ESC[2J
\ -----
: page ESC[ [char] 2 emit [char] J emit
    0 0 at-xy ;

\ -----
\ Palabras para salida numerica con formato.
\ La cadena se forma en un buffer que abarca
\ desde N hasta N+21. El ultimo byte es el \0,
\ y el primero un indice sobre el ultimo caracter
\ generado. <# limpia el buffer y coloca el
\ \0 y el valor del indice a 21; # genera un
\ nmero que almacena en el buffer actualizando
\ el indice; hold inserta el caracter que se
\ especifique; #s realiza la conversion pendiente
\ en un momento dado
\ -----

\ -----
\ inicia la conversion: usa un buffer de 1+20+1
\ caracteres que comienza en N; en el primer

```

```

\ byte va el indice, en los veinte siguientes
\ la cadena; el ultimo es \0. El indice apunta
\ al primer caracter que se imprimira
\ -----
: <# N 21 1 do dup I + 32 swap c! loop
  dup 21 + 0 swap c!
  21 swap c! ;

\ -----
\ obtiene un digito y deja el resto en la
\ pila: la primera linea genera el caracter,
\ la segunda lo escribe en el buffer, la
\ tercera actualiza el puntero y la ultima
\ actualiza el numero que queda en la pila
\ -----
: # dup 10 mod 48 +
  N c@ 1- N c!
  N c@ N + c!
  10 / ;

\ -----
\ Realiza el proceso de conversion hasta que
\ en la pila quede un cero
\ -----
: #s begin dup while # repeat ;

\ -----
\ inserta un caracter en la cadena con formato;
\ la primera linea apunta a la posicion correcta,
\ la segunda escribe el caracter; la tercera
\ actualiza el indice; presupone que hay algo
\ en la pila
\ -----
: hold N dup c@ 1- swap c!
  N dup c@ + c! ;

\ -----
\ inserta un signo -
\ -----
: sign [char] - hold ;

```

```

\ -----
\ termina el proceso de conversion, eliminando
\ el 0 de la pila y dejando en ella la direccion
\ del primer caracter para que type la use
\ -----
: #> drop N dup c@ + ;

\ -----
\ imprime una cadena de la que se conoce la
\ direccion. Al contrario que en Forth ansi,
\ las cadenas estan terminadas en \0. Por supuesto,
\ puede reescribirse la funcion para hacerla
\ conforme ANSI
\ -----
: type begin dup c@ while dup c@ emit 1+ repeat drop ;

\ -----
\ .r, para impresion por columnas. Simplemente
\ se hace la conversion y luego se ajusta la
\ direccion. El programador es responsable de no
\ cortar la cadena. La unica precaucion tiene
\ que ver con el signo; con la primera linea de
\ en la pila el numero de espacios, el numero a
\ imprimir y su valor absoluto; con la segunda
\ realizo la impresion; con la tercera añado si
\ es preciso el signo; con la cuarta coloco el
\ puntero e imprimo
\ -----
: .r swap dup abs
  0= if
    drop
    <#
    [char] 0 N 20 + c!
  else
    dup abs <# #s #> drop
    0< if sign then
  then
  21 swap - N + type ;

\ -----
\ fill espera en la pila una direccion, un

```

```

\ contador y un caracter de relleno
\ -----
: fill -rot 0 do 2dup I + c! loop 2drop ;

\ -----
\ erase se basa en fill, y rellena un rango de
\ direcciones con el caracter nulo; espera en
\ la pila una direccion y una cuenta
\ -----
: erase 0 fill ;

\ -----
\ copy copia una region de memoria en otra.
\ Supone que no hay solape (si lo hubiese,
\ una forma segura de hacer la copia es usando
\ un buffer intermedio). Espera una direccion
\ origen, una direccion destino y un contador
\ -----
: copy 0 do 2dup swap c@ swap c!
      1+ swap 1+ swap loop
      2drop ;

\ -----
\ compare compara cadenas; espera en la pila una
\ direccion origen, una direccion destino y un
\ contador. Devuelve en la pila un -1 si hasta
\ el valor del contador incluido las dos cadenas
\ son iguales, o el numero del caracter donde
\ las dos cadenas difieren. Ya que hay que acceder
\ a caracter origen y caracter destino, es util
\ la funcion 2c@, que sustituye dos direcciones
\ en la pila por sus contenidos. En cuanto a
\ compare, la idea es poner en la pila, bajo las
\ dos direcciones, una bandera a -1, y cambiarla
\ a I si se encuentra una diferencia
\ -----
: 2c@ swap c@ swap c@ ;
: compare >r -1 -rot r>
  0 do 2dup 2c@
    != if    rot drop I -rot break
    else 1+ swap 1+ swap then loop

```

```

drop drop ;

\ -----
\ accept espera en la pila una direccion base
\ y un limite a partir de esa base, y toma de
\ teclado una cadena que deja en el buffer. Como
\ la cadena sera terminada en \0, para un tamao
\ l el desplazamiento maximo a partir de la
\ direccion base sera l-2; llamaremos b a la
\ direccin base, l al limite, i al indice que
\ incrementa la direccion base y k al caracter
\ leido; el indice indica la proxima posicion
\ libre
\ -----
: <-X <cursor 32 emit <cursor ;
: accept          \ b l
  swap 0          \ l b i
  begin
    key dup 10 = if      \ l b i k (k 10 =)
      drop + 0 swap c!  \ l
      drop 0            \ 0
    else
      dup              \ l b i k k
    then
  while            \ l b i k
    dup 127 = if      \ l b i k (k 127 =)
      drop dup 0= if  \ l b i (i 0 =)
      else
        1- <-X        \ l b i-1 , borra atras
      then
    else              \ l b i k
      >r rot           \ b i l ; k
      2dup < if       \ b i l (i<l)?
        -rot          \ l b i
        plas r> dup emit \ l b i+1 b+i k
        swap c!       \ l b i+1
      else
        -rot
        r> drop       \ l b i
      then
    then
  then

```

```

    repeat ;
\ =====
\ cadenas
\ =====

\ -----
\ crea una cadena; la longitud incluye el \0
\ -----
: string create cell - allot does> ;

\ -----
\ longitud, sigo suponiendo cadenas \0; strlen
\ espera en la pila la direccion base, y deja
\ la longitud de la cadena, incluyendo el \0
\ -----
: strlen 0 begin plus c@ 0= until nip ;

\ -----
\ strup espera una direccion base en la pila;
\ transforma la cadena a letras mayusculas
\ -----
: upcase?   dup 65 >= swap 90 <= and ;
: lowercase? dup 97 >= swap 122 <= and ;
: strup     dup strlen
            1- 0 do
                dup I + dup
                c@ dup lowercase? if
                    32 - swap c!
                else
                    drop drop
                then loop ;
: strlow    dup strlen
            1- 0 do
                dup I + dup
                c@ dup upcase? if
                    32 + swap c!
                else
                    drop drop
                then loop ;

\ -----

```

```

\ strscan recorre una cadena a la busqueda de
\ un caracter; deja -1 en la pila si no lo
\ encuentra, o su posicion si lo encuentra
\ -----
: strscan          \ b k
  -1 -rot          \ -1 b k
  over strlen     \ -1 b k long
  0 do            \ -1 b k
    over I + c@   \ -1 b k k'
    over = if     \ -1 b k ?
      rot drop
      I -rot     \ I b k
      break
    then
  loop 2drop ;

\ =====
\ complementos de entorno
\ =====

\ -----
\ espacio usado
\ -----
: used N 100 * 10000 / 3 .r [char] % emit ." dictionary" type cr
  M 100 * 10000 / 3 .r [char] % emit ." code" type ;

\ -----
\ fecha y hora en formato simpatico
\ -----
: .time time swap rot
  . [char] : emit . [char] : emit . ;
: .date date swap rot
  . [char] / emit . [char] / emit . ;

\ =====
\ archivos
\ =====
: fr 1 ;

```

```

: fw 2 ;
: fa 3 ;
: channels 8 0 do I channel . loop ;

\ =====
\ operaciones con numeros racionales
\ =====

\ q+,q-,q*,q/ toman dos racionales de la pila
\ y devuelven el racional resultante. Un
\ racional se representa mediante dos enteros
\ que se depositan en la pila: primero el
\ numerador y despues el denominador. El
\ resultado consta de numerador y denominador

\ -----
\ calcula el maximo comun divisor de dos numeros;
\ se basa en que  $\text{mcd}(a,b)=\text{mcd}(b,r)$  donde r es el
\ resto de a/b. Cuando r=0, b es el mcd buscado
\ -----
: mcd ?dup if tuck mod mcd then ;

\ -----
\ reduce una fraccion a la forma canonica,
\ dividiendo numerador y denominador por el
\ maximo comun divisor de ambos
\ -----
: reduce 2dup mcd tuck / >r / r> ;

\ -----
\ suma de racionales
\ -----
: q+ rot 2dup * >r rot * -rot * + r> reduce ;

\ -----
\ resta de racionales
\ -----
: q- negate q+ ;

\ -----
\ multiplicacion de racionales

```

```

\ -----
: q* rot * >r * r> reduce ;

\ -----
\ division de racionales
\ -----
: q/ >r * swap r> * swap reduce ;

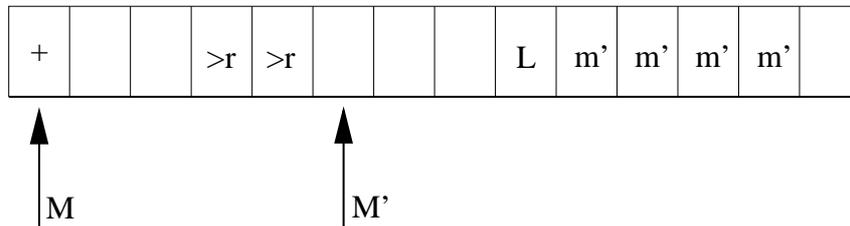
\ -----
\ imprime primero numerador y luego denominador
\ -----
: q. swap . . ;

page cr
." ''I trust a straigh stick to support my weight along " type cr
." its length (though not as a lever). I don't trust " type cr
." charmingly gnarled or high-tech telescoping sticks." type cr
." I don't want a gripo as I'm always sliding my hand " type cr
." along the shaft to adjust height. Two sticks are " type cr
." too many.'' " type cr
." Charles Moore " type cr
cr
." ##### # # # # # # # " type cr
." # # # # # # # " type cr
." #### # # # # # # # " type cr
." # # # ##### # #####" type cr
." # # # # # # # # " type cr
." # ##### # # # # # # " type cr
cr
." L E Z A R - 2006 " type cr

```

## 10.9. Compilación de algunas palabras

Algunas palabras se compilan de forma trivial. Considérese la Figura 10.2. En un momento dado, el puntero al siguiente byte libre tiene el valor  $M$ . Si en modo de compilación entonces se encuentra la palabra  $+$  sólo es preciso escribir su bytecode e incrementar el puntero.



**Figura 10.2** Compilación de do.

Otras palabras no pueden compilarse de forma trivial. Por ejemplo, cuando encontramos la palabra `do` será preciso en primer lugar traspasar desde la pila de parámetros a la pila de retorno los valores base y límite para el bucle. Por tanto, la compilación de `do` no consiste en escribir su bytecode, sino en codificar por dos veces `>r` para que, en tiempo de ejecución, se realice el traslado. Una vez compilada por dos veces `>r` el puntero al siguiente byte libre toma el valor `M'`. Este valor será el de la dirección de retorno de `loop`, así que debe ser preservado. ¿Cómo? Colocándolo en la pila de retorno. Algún tiempo después, se encontrará para compilar un `loop`. En la Figura 10.2 su bytecode se representa mediante `L`. Al alcanzar el `loop`, será preciso saltar a `M'`, valor que quedó en la pila de retorno al compilar `do`. Entonces, en tiempo de compilación, se toma de la pila de retorno el valor de `M'` y se compila a continuación de `L`. En una implementación de 32 bits ese valor ocupará cuatro bytes, que es lo que se intenta representar en la figura. Así cuando en tiempo de ejecución se encuentre `loop` solo habrá que comprobar el valor del índice y el límite y efectuar un salto: bien volviendo al inicio del bucle, bien saltando sobre el valor compilado de `M'`:

```

int compilar_do()
{
    /* 12 es el bytecode de 'do' */
    *(C+M)=12; ++M;
    *(C+M)=12; ++M;
    ret[tope_r]=M;
    ++tope_r;
    return(0);
}
int compilar_loop()
{
    int *q;

```

```

*(C+M)=36;
++M;
q=(int *) (C+M);
*q=ret[tope_r-1];
--tope_r;
M+=sizeof(int);
return(0);
}

```

Obsérvese que no queda huella de `do` en el bytecode, es decir, no hay un comportamiento de `do` en tiempo de ejecución: no es necesario. El comportamiento de `loop` es éste:

```

int ejct_loop()
{
    int *q;
    if (tope_r<2) return(1);
    if (ret[tope_r-2]<ret[tope_r-1]-1){
        ++ret[tope_r-2];
        q=(int *) (PXT+1);
        PXT=C+(*q);
    } else
    {
        tope_r-=2;
        PXT+=sizeof(int)+1;
    }
    return(0);
}

```

Como segundo ejemplo, consideremos la compilación de la estructura `if ... else ... then`. En relación con la Figura 10.3, veamos que, al compilar `if` hemos de reservar un entero que indicará el salto tras el `else`. Así que, de momento, compilamos el bytecode para `if` y guardamos en la pila de retorno el valor de `M`, indicando la posición donde después habrá que escribir.

Cuando después encontremos la palabra `else`, habrá que reservar espacio para un entero que indique el salto a `then`. Así que tomamos el valor de `M'` y lo escribimos en la dirección `M`, que tomamos de la pila de retorno. A continuación guardamos en la pila de retorno el valor de `M'`. Un tiempo después encontraremos `then`. Cuando esto suceda, el valor del puntero en el

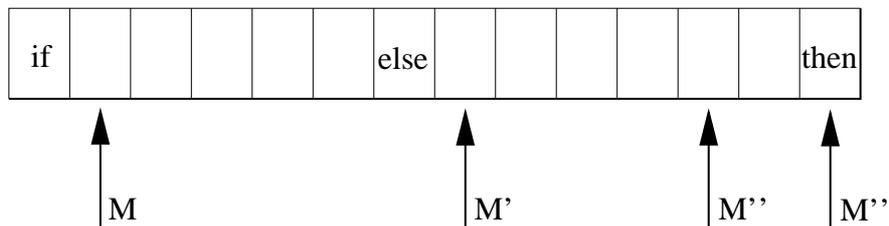


Figura 10.3 Compilación de `if ... else ... then`.

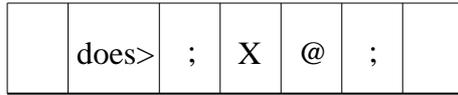
espacio de código será  $M'''$ . Entonces, en lugar de compilar un bytecode para `then`, simplemente tomamos el valor de  $M'''$  y lo escribimos en el espacio que reservamos tras el bytecode de `else`, cuya posición  $M'$  habíamos colocado en la pila de retorno. De esta forma, cuando en tiempo de ejecución encontremos un `if`, saltaremos o bien  $1+\text{sizeof}(\text{int})$  o bien a la posición indicada por el entero que hay a continuación del propio `if`. En este último caso, tarde o temprano encontraremos un `else`, justo a continuación del cual se encuentra un entero que indica el salto a `then`.

Para terminar, consideremos la compilación de la palabra `does>`. La compilación de esta palabra no es trivial. Para fijar ideas, consideremos

```
: constant create , does> @ ;
```

Cuando compilamos `constant`, evidentemente hemos de compilarla hasta el final. Pero cuando la ejecutamos no, porque el código entre `does>` y `;` es código perteneciente a la constante que se cree mediante `constant`. En *Lézar*, `does>` se compila mediante tres bytecodes, tal y como muestra la Figura 10.4.

`X` es una palabra interna al núcleo. Como se ve, compilamos `does>`, seguido de un fin de palabra, seguida de `X`. Ahora, en tiempo de ejecución de `constant` se ejecutará únicamente hasta el primer `;`. En concreto el comportamiento en tiempo de ejecución de `does>` es el siguiente: copiar desde `X` hasta el segundo `;`, incluidos, en el código de la palabra que acaba de crearse, y en cuya creación el puntero a código de su entrada ha sido ajustado a la primera posición libre en el espacio de código, indicada por el puntero `M`. De esta forma, cada vez que se cree una constante, a la nueva entrada en el diccionario se le asignará el código `X @ ;`. Esto indica ya qué es lo que hace `X`. Cuando el bytecode para `X` sea apuntado por el puntero que recorre el código, y que hemos llamado `PXT`, lo que se hace es recorrer el diccionario,



**Figura 10.4** Compilación de does>.

buscando la palabra cuyo puntero a código coincide con PXT en ese momento, y dejando en la pila la dirección de la primera celda de datos de esa palabra. Recomiendo al lector que tome en este punto papel y lápiz y reproduzca, dibujando una porción de diccionario, la explicación que acabamos de dar.

# Capítulo 11

## Miscelánea de Forth

### 11.1. Historia de Forth

Existen dos artículos históricos interesantes: *The FORTH Program for Spectral Line Observing*, de Charles H. Moore y Elizabeth Rather, y *The Forth approach to operating systems*, de los mismos autores. Sin embargo, haremos aquí un resumen de la documentación que sobre la historia de Forth se ofrece a través de Forth Inc., la empresa fundada por Moore para ofrecer soluciones basadas en Forth <sup>1</sup>.

Hay en toda esta historia un hecho llamativo: que el lenguaje no fue apoyado por instituciones académicas ni empresas, sino que su diseño, implementación y expansión fue el resultado del esfuerzo de un individuo.

Charles H. Moore comenzó su carrera como programador a finales de los años 50, en el Observatorio Astrofísico Smithsonian. Su trabajo consistía en programar algoritmos para cálculo de efemérides, seguimiento de satélites, cálculo de elementos orbitales, etc. Se programaba en Fortran, codificando sobre tarjetas perforadas, y el trabajo resultaba penoso, más a medida que crecía el número de estas tarjetas. Por eso escribió un intérprete que se ocupaba de la lectura de las mismas. En aquel intérprete se encontraban ya algunos de los conceptos que después formarían parte de Forth. En 1961 se graduó en Física por el MIT y pasó a Stanford, involucrándose en algunos proyectos de programación. De aquella época datan algunas mejoras en el intérprete

---

<sup>1</sup>Se ha contado muchas veces la anécdota, y la repetiremos una vez más aunque sea a pié de página: el nombre de Forth es en realidad 'Fourth' afectado de una limitación a cinco caracteres en los nombres de fichero del sistema donde fue desarrollado por primera vez

que había desarrollado en el Smithsonian. Aparece el concepto de pila de parámetros y se añaden operadores lógicos y aritméticos y la capacidad para definir procedimientos.

En 1965 se trasladó a Nueva York, donde trabajó como programador independiente con varios lenguajes, como Fortran, Algol, PL/I y diversos ensambladores. Continuó sin embargo mejorando su intérprete. Por ejemplo, a finales de los 60 aparecieron los terminales y con ellos la necesidad de añadir funciones para entrada y salida de caracteres.

En 1968 se trasladó a una pequeña ciudad para trabajar en una empresa llamada Mohasco Industries, como programador de gráficos sobre una máquina IBM 1130. Esta computadora disponía de una CPU de 16 bits, 8K de RAM, disco, teclado, impresora, perforadora de tarjetas y compilador Fortran. De aquella época provienen algunas primitivas utilidades para manejar código fuente y un editor. Escribió por diversión un videojuego para aquel sistema y pasó su programa de ajedrez de Algol a Forth. Esta fue la primera vez que el sistema de Moore se llamó así, sugiriendo que se trataba de software para una cuarta generación de ordenadores. Pronto se encontró más cómodo con este sistema de desarrollo que con el sistema Fortran original de la máquina, y así introdujo algunas mejoras y conceptos más, como los bucles y los bloques de código de 1K. Al implementar el diccionario en la forma en que lo conocemos, se introdujo el concepto de *indirect threaded code*: cada entrada en el diccionario contiene un puntero a su código asociado, y este a su vez consta, esencialmente, de punteros a otras palabras de alto nivel o a rutinas en código máquina. También de esta época es la introducción de una pila de retorno.

En 1970 Moore fue puesto al frente de un ambicioso proyecto sobre un UNIVAC 1108. Moore tradujo de nuevo su sistema a la nueva máquina y añadió capacidades multitarea y mejores mecanismos para manejar bloques. El proyecto sin embargo fue cancelado antes de terminarse. Como resultado de aquella frustración, Moore escribió un libro sobre Forth que nunca llegó a publicarse. En él consideraba los inconvenientes de que los sistemas se construyan como una jerarquía de lenguajes que abarca desde el ensamblador hasta el lenguaje propio de las aplicaciones, y del esfuerzo que su mantenimiento, modificación y comprensión supone, tanto para los programadores del sistema como para los usuarios finales. Su solución: Forth. Una simple capa entre la máquina y el usuario, con una interfaz usuario-Forth y otra Forth-máquina. Recomendaba insistentemente escribir software simple, no

especular con necesidades futuras (resolver problemas concretos, no problemas generales que no existen) reescribir las rutinas clave una y otra vez, buscando la forma de hacerlas mejores, no reutilizar código sin reexaminarlo sólo porque funciona. Durante los 70, Moore escribió sistemas Forth para 18 CPU's distintas, escribiendo en cada caso un ensamblador, controladores de dispositivo e incluso las rutinas aritméticas básicas de multiplicación y división allí donde era preciso.

La primera implementación completa y autónoma de Forth data de 1971, para el radiotelescopio de Kitt Peak. El sistema se basaba en dos máquinas: un PDP-11 de 16K y un H316 de 32K unidos por una conexión serie. Era por tanto un sistema multiprogramado y multiprocesador, encargado de apuntar el radiotelescopio, recopilar datos y ofrecer un entorno interactivo a los investigadores, que habían de realizar tratamiento de imágenes y procesado de datos. En aquella época, los miniordenadores no contaban con sistemas de desarrollo, pues sus recursos eran muy limitados. En su lugar, el desarrollo se hacía sobre un *mainframe* con un compilador cruzado. Sin embargo, Forth, escrito en Forth, se ejecutaba de forma autónoma y servía al mismo tiempo como sistema multiprogramado y multiusuario que ejecutaba programas y que permitía desarrollarlos.

El sistema se trasladó en 1973 a un nuevo PDP-11 con unidad de disco como un sistema multiusuario que ofrecía cuatro terminales. Resultó tan avanzado que rápidamente se difundió entre la comunidad de astrofísicos. Se instaló en el Observatorio Steward, en Cerro Tololo, en el MIT, en el Imperial College de Londres y en la Universidad de Utrech. En 1976 fue adoptado por la Unión Astronómica Internacional.

Muchas versiones multiusuario fueron implementadas en años sucesivos, sobre máquinas y procesadores Honeywell, IBM, Varian, HP, PDP-11, Interdata, Raytheon, etc. abarcando aplicaciones científicas, de gestión, de control y de procesado de imágenes. Moore era capaz de portar Forth en un par de semanas, ya que Forth estaba escrito en Forth, salvo un pequeño núcleo de unas sesenta primitivas hechas en ensamblador.

En 1976 Forth Inc., la compañía creada por Charles Moore y Elizabeth Rather para explotar comercialmente el concepto Forth, produjo una versión para microprocesadores de ocho bits. Esta versión, llamada microForth, fue llevada al Intel 8080, Zilog z80 y Motorola 6800. El salto al ecosistema de los microcomputadores produjo una comunidad de entusiastas y aficionados,

pero el control sobre Forth estuvo totalmente en manos de Moore hasta 1978. A partir de entonces, al tiempo que aparecían competidores de Forth Inc. con sus propias versiones, Moore se fue interesando cada vez más por las implementaciones hardware de Forth, hasta el punto de abandonar los proyectos software en 1982.

La comunidad Forth entretanto hizo sus propios progresos: se creó el Forth Interest Group y algunos personajes relevantes hicieron aportaciones significativas, como Bill Ragsdale y Robert Selzer, que produjeron un sistema Forth para el Motorola 6502, que fue el germen de lo que sería más tarde FIG Forth Model: una especificación abierta para implementar y portar sistemas Forth. La idea funcionó, y hacia 1983 existían sistemas Forth para al menos dieciseis arquitecturas.

Poco después llegó el desembarco de IBM con su PC. Uno de los primeros productos que ofreció fue el procesador de textos EasyWriter, escrito en Forth. Compañías como Laboratory Microsystems produjeron sistemas avanzados Forth para PC, como una versión de 32 bits en Febrero de 1983 y versiones corresidentes de Forth con OS/2 (1988) y Windows (1992). Por su parte, Forth Inc. ofrecía polyFORTH sobre PC: un sistema multiusuario capaz de dar servicio a dieciseis usuarios sin degradación visible en el rendimiento. Y cuando apareció el procesador 80386, polyFORTH sobre máquinas NCR basadas en este procesador era capaz de dar servicio a 150 usuarios. A mediados de los 80, más de medio centenar de fabricantes ofrecían sistemas Forth, y el modelo FIG Forth fue poco a poco reemplazado por otra versión de dominio público: F83.

La queja justificada de compañías como Forth Inc. es que la facilidad para crear sistemas Forth, si bien contribuyó a su expansión, generó asimismo la impresión de que todos los sistemas Forth eran iguales, y por decirlo de alguna manera, poco más que juguetes. Efectivamente, es muy fácil programar un juguetito Forth para uso personal, pero hay una enorme diferencia entre implementaciones.

Por otra parte, Forth siguió su camino fuera del mundo PC, en el ambiente en que mejor se defendía: los sistemas dedicados. Sería demasiado larga y tediosa la enumeración de todos los sistemas de estas características programados en Forth, así que remitimos al lector interesado a la página de Forth Inc.

Finalmente, la tercera gran rama de Forth es la que comienza en 1973 cuando John Davies, del observatorio de Jodrell Bank, modificó un computador Ferranti para ejecutar Forth más eficientemente. Desde entonces, se han sucedido las implementaciones hardware destinadas a la ejecución eficiente de Forth. En 1976, la compañía californiana Standard Logic consiguió, mediante una pequeña modificación en su placa base, dotar a ésta del mecanismo del intérprete Forth que permite pasar de una palabra a la siguiente. En 1985 Moore consiguió los primeros procesadores Forth, diseño comprado y mejorado por Harris Semiconductors Inc. y que fue el núcleo de los procesadores RTX.

En fin, hemos dado una visión si no exhaustiva si creemos que ilustrativa de lo que ha sido la evolución de Forth, con especial énfasis en los años fundacionales. Un último hito importante en esta historia fue la especificación ANSI FORTH de 1994. Preferimos dejar aquí la historia, en parte porque creemos que si bien la norma de 1994 fue importante como referencia a la que adherirse, se aparta ya de la filosofía original de Forth: simplicidad.

## **11.2. PostScript y JOY**

### **11.2.1. PostScript**

PostScript es el más obvio descendiente de Forth. Fue desarrollado por Adobe Systems Inc. alrededor de 1985 como lenguaje de descripción de páginas independiente del dispositivo y comparte buena parte de la filosofía de Forth y un aire de familia por su estructura basada en una pila y un diccionario. Este diseño es consecuencia del deseo de mantener durante el mayor tiempo posible los programas en formato fuente. De esta manera, pueden pasar de un sistema a otro sin modificación, transmitirse sin problemas a través de una red o enviarse a dispositivos tan distintos como pueden ser una impresora láser o un terminal gráfico. Para que esto sea posible, el intérprete ha de residir en el periférico destinatario del programa PostScript, y para que en el presumiblemente reducido espacio de memoria de un periférico pueda ejecutarse un intérprete lo mejor es una máquina virtual sobre la que cada palabra tiene una acción definida: de ahí el uso de una pila y por ende de la notación postfija.

A pesar de todo esto, PostScript tiene un nivel superior a Forth, aunque a cambio pierde buena parte de esa maravillosa modularidad que nos

sorprende en Forth. Este más alto nivel se manifiesta de varias maneras. Primero, los operadores aritméticos están ligeramente sobrecargados, de modo que puedan hacerse cargo tanto de reales como de enteros, de forma transparente. Segundo, el lenguaje cuenta con varias estructuras de datos de alto nivel predefinidas. Tercero, el lenguaje incluye varios cientos de funciones. Cuarto, la máquina virtual es sólo parcialmente accesible. En este alto nivel se filtran ciertas esencias de Lisp, como por ejemplo algunas sutilezas en la comparación de objetos.

No es este el lugar para describir PostScript. Existen algunos buenos libros sobre el tema. *Adobe PostScript tutorial and cookbook* es de Adobe System y está disponible en la red. Existe un segundo título que puede considerarse la continuación del primero: *PostScript language program design*, y una de las mejores referencias es *Thinking in PostScript*, de Glenn Reid.

En definitiva, puede considerarse a PostScript como una encarnación de Forth fuertemente orientada a los gráficos que mantiene su vigencia cuando ha cumplido ya un par de décadas. A ello ha contribuido su adopción por el sistema UNIX, y la popularización de versiones gratuitas de este sistema.

### 11.2.2. JOY

JOY es un interesante lenguaje diseñado por Manfred von Thun, del departamento de filosofía de la Universidad de La Trobe, en Melbourne, Australia. Es un lenguaje funcional heredero en muchos aspectos de Forth. En lugar de entrar en discusiones formales sobre lenguajes imperativos y funcionales, aquí va el código para sumar los primeros diez enteros en un lenguaje típicamente imperativo, como C:

```
total=0;
for(i=0;i<10;++i) total+=i;
```

y aquí el código para efectuar la misma operación con un lenguaje típicamente funcional, como Haskell

```
sum [1..10]
```

En palabras de su creador, JOY es un lenguaje puramente funcional de alto nivel que elimina la abstracción del cálculo lambda y la aplicación de funciones y reemplaza ambos por la composición de funciones. En la página

principal del lenguaje <sup>2</sup> se encuentran resúmenes e introducciones, prácticas y teóricas, incluyendo una justificación de los fundamentos matemáticos usados en el lenguaje. Esta fundamentación matemática se remonta a las primeras décadas del siglo XX y recoge la opinión de Backus de que los conceptos generadores de los lenguajes habían de ser elegidos de forma que tuviesen una base matemática firme.

No pretendemos aquí una exposición siquiera somera del lenguaje, más bien, un recorrido a vista de pájaro para trasladar al lector el aspecto y el *tacto* de JOY.

En un primer nivel, JOY usa notación postfija y una pila:

```
2 3 +
4.53 8.16 *
```

Pero en la pila, además de valores literales de varios tipos, pueden colocarse estructuras compuestas, como listas. Una lista se identifica por unos corchetes, y puede contener elementos de varios tipos, incluidas otras listas:

```
[ 3.14 42 [ 0 1 2] -9 ]
```

Pero las listas pueden ser estructuras pasivas, es decir, simples datos, o programas:

```
[ dup * ]
```

Hay muchas formas de combinar ambos tipos de listas, por ejemplo en

```
[ 1 2 3 4 ] [ dup * ] map
```

la palabra `map` aplica la lista que se encuentra arriba de la pila sobre la que se encuentra justo debajo, produciendo como resultado la lista

```
[ 1 4 9 16 ]
```

En la definición de funciones no hay parámetros formales:

```
cuadrado == dup *
```

---

<sup>2</sup><http://www.latrobe.edu.au/philosophy/phimvt/joy>

Pueden agruparse varias definiciones en un bloque, delimitado por `DEFINE` y `::`:

```
DEFINE
  cuadrado == dup * ;
  cubo     == dup dup * * .
```

Las definiciones de funciones pueden incluir referencias recursivas, pero existen operadores que permiten definir funciones recursivas de forma no explícita. Por ejemplo, `primrec`, de *primitive recursion*, puede usarse de este modo para calcular el factorial de 5:

```
5 [1] [*] primrec
```

Al ejecutarse la línea anterior, primero los dos programas identificados como listas son retirados de la pila, quedando sólo el número 5. Si el número que queda en la pila es distinto de cero, se duplica y se decrementa. Si es cero, se ejecuta el primero de los programas retirados por `primrec`, es decir `[1]`, que se limita a apilar el valor 1. Cuando termina la recursión, se aplica tantas veces como se necesario el segundo programa que `primrec` retiró de la pila. Como resultado final, un 120 queda en la pila.

JOY tiene tipos de datos simples y tipos compuestos. Entre los simples están los enteros, reales, caracteres y booleanos; entre los compuestos las listas, las cadenas y los conjuntos. Las listas se identifican mediante los corchetes, los conjuntos mediante llaves y las cadenas mediante comillas dobles:

```
[ 1 2 3 ]
{ 34 45 0 }
"A"
```

En cuanto a los datos simples, los enteros y reales se escriben como es habitual (para los reales se admite notación exponencial, p. ej 314e-2) y los caracteres se identifican con una comilla simple, como en 'A'. Existen los habituales operadores aritméticos y lógicos.

El operador `cons` permite añadir elementos de un agregado. Por ejemplo `[ 1 2 3 ] 4 cons` produce la lista `[ 1 2 3 4 ]`. `first` y `rest` permite extraer elementos de la lista.

La entrada y salida se realiza con `get` y `put`. Por ejemplo, el siguiente programa es un diálogo trivial:

```
"Como te llamas?" put "Hola " get concat put
```

y produce, si la respuesta fue Rosa la salida

```
"Hola Rosa"
```

En resumen, JOY aparece como un híbrido entre Forth y Lisp que usa una aproximación original para dar respuesta a una necesidad teórica de fundamentar los lenguajes de programación.

### 11.3. El panorama en 2006

Una exploración de la red revela muy poca actividad en torno a Forth. No encontramos muchos sitios, y la mayoría de ellos no contienen material original, sino enlaces a media docena de fuentes, que a su vez llevan meses, o años, sin actualizarse. ¡Es casi seguro que Microsoft no va a lanzar un MS-Forth para antes del verano!

Por otro lado, gran parte del material gira en torno al mismo Forth, no alrededor del software que pueda escribirse en este lenguaje. La razón es sencilla: Forth es el entendimiento, la iluminación, y eso es muy de agradecer en estos tiempos complicados. Forth permanecerá porque apela a un sentido estético particular, a un modo de entender los problemas. Al mismo tiempo, es muy improbable que reverdezca. No es para todo el mundo.

Para mí, es una suerte de caligrafía. Una de esas artes raras en cuyo cultivo algunos iniciados encuentran una satisfacción profunda.

### 11.4. Referencias comentadas

*Starting Forth*, de Leo Brodie es la referencia fundamental para aquel que por primera vez se acerca al mundo de Forth. La edición en papel hace mucho que se encuentra agotada, pero afortunadamente puede encontrarse en la red. Por ejemplo, en alguna de las URL que se indican:

- [http://www.amresearch.com/starting\\_forth](http://www.amresearch.com/starting_forth)
- <http://home.vianetworks.nl/users/mhx/sf.html>

*Starting Forth* es una referencia clara y muy bien escrita e ilustrada, con ejemplos y ejercicios.

*Thinking Forth* también es una obra de Leo Brodie, más centrada en la ingeniería del software. Aunque el punto de vista es el de un programador en Forth, contiene enseñanzas intemporales aplicables a cualquier lenguaje de programación. En una palabra, *Thinking Forth* es un clásico. Puede encontrarse en

- <http://thinking-forth.sourceforge.net/>

De la página de Forth Inc., la empresa fundada por Charles Moore, puede obtenerse una buena cantidad de información. Para empezar, Forth Inc. ofrece un compilador Forth, SwiftX, para varias plataformas. Hay una introducción al lenguaje y pueden adquirirse dos buenos libros: *Forth Programmer's Handbook* y *Forth Application Techniques*. Aquí se encuentra también una historia de Forth (nosotros de hecho hemos tomado un resumen de estas páginas). Un tercer libro interesante es *Programming Forth*, de Stephen Pelc. Puede descargarse de la página de Microprocessor Engineering Limited, cuya URL es <http://www.mpeltd.demon.co.uk>.

Para comenzar a experimentar con Forth necesitamos desde luego un sistema Forth. Hay muchos, pero es recomendable *Gforth*, la implementación GNU. Puede encontrarse en

- <http://www.jwtdt.com/paysan/gforth.html>
- <http://www.gnu.org/software/gforth/>
- <http://www.complang.tuwien.ac.at/forth/gforth/Docs-html/>

La tercera referencia da acceso a la documentación en línea de *Gforth*. La página del *Forth Interest Group* (FIG), contiene muchos enlaces interesantes: implementaciones, documentación, artículos, etc. Se accede a través de <http://www.forth.org>.

En la página de *Taygeta Scientific Inc.* puede encontrarse mucha información sobre Forth, incluyendo un enlace para descargar un buen libro: *Real Time Forth*, de Tim Hendtlass, unas introducciones rápidas a Forth, las de Glen Haydon, Julian Noble, Dave Pochin y Phil Burk y un enlace al proyecto *Forth Scientific Library*, liderado por Skip Carter. También el libro *Stack Computers: the new wave*, de Philip Koopman.

Un artículo muy interesante es el de Michael Misamore *Introduction to Thoughtful Programming and the Forth Philosophy*. Este artículo contiene a su vez referencias a otros textos interesantes, en particular, a los artículos de Jeff Fox *Low Fat Computing* y *Forth - the LEGO of programming languages* entre otros.

Ya citamos dos artículos básicos: *The Forth approach to operating systems* y *The FORTH program for spectral line observing* de Charles Moore y Elizabeth Rather. Pueden conseguirse a través de ACM Portal. En este sitio se encuentran muchos otros de interés, en particular la serie que escribe desde hace años Paul Frenger y cuya lectura es un auténtico placer.

## 11.5. Palabras finales

No existe una obra perfecta, y esta se encuentra lejos de mis aspiraciones originales. Sin embargo, creo que es digna, pero sobre todo necesaria, pues, hasta donde conozco, no existía ningún libro de Forth en español. Hubiese sido desde luego más útil hace veinte años, cuando Forth vivía su mejor momento. Pero ¿qué importan las modas? Las lecciones de Forth son intemporales, como lo son la admiración que muchas veces produce, el placer que procura el entendimiento, la constatación de que hay otros caminos.

: buena suerte ;