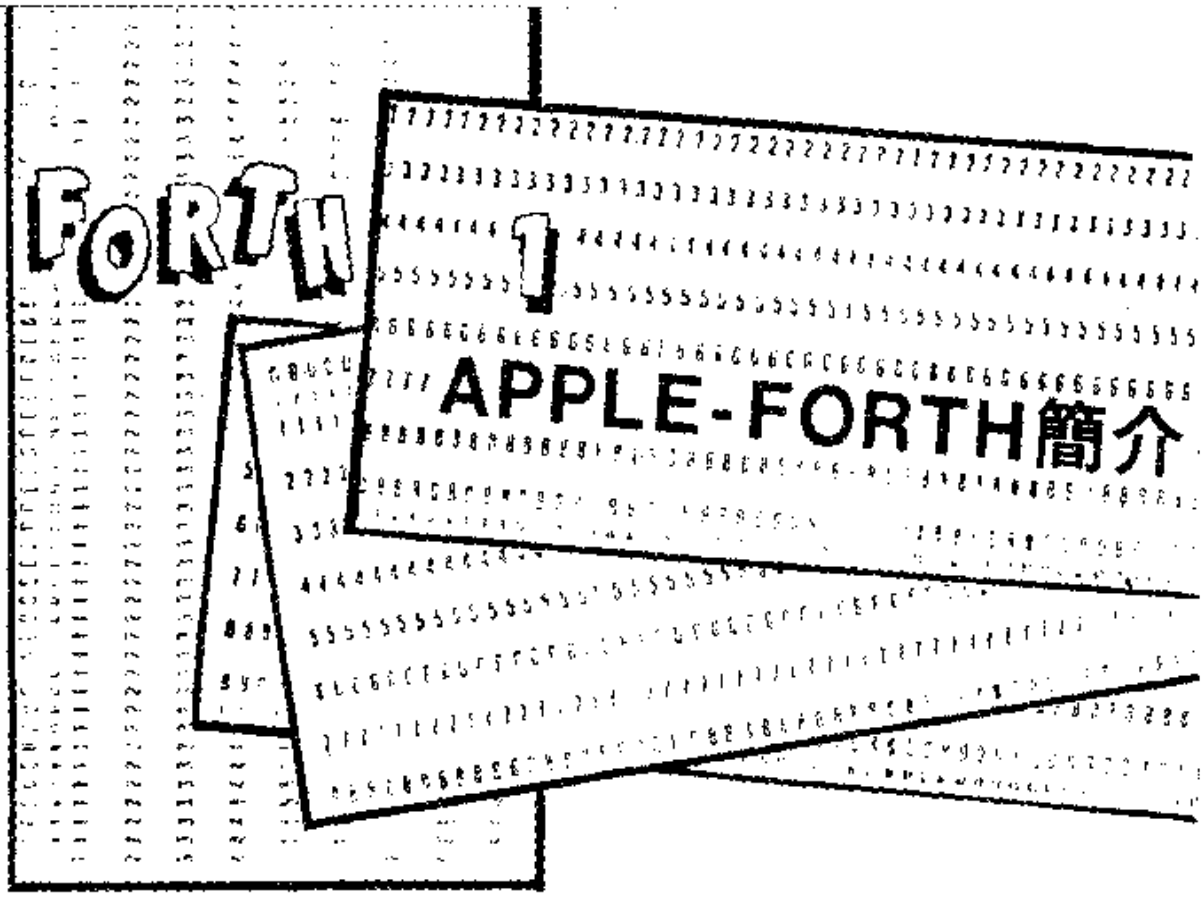
目　　錄



1-1 FORTH的起源

FORTH是美國的一位程式師查爾斯・摩爾（Charles Moore）發明的一種電腦語言。摩爾是一個所謂自由工作的程式師，爲不同的公司機構寫作各種電腦程式，因而他使用過各種型式的電腦，也接觸過各種電腦語言。他極不滿意電腦公司供給電腦界使用的各種語言及發展軟體的工具，因爲這些軟體的工具都不能讓使用者有效地指揮電腦工作以解決他們所面對的程式問題。摩爾在他自己寫作程式時，是持守着藝術家對他作品美化的理念，對程式的品質要求十分嚴格。他深自相信，雖然一個程式問題可以有許多不同的解，但是最好的解，必定是最簡單的解。因爲簡單（Simplicity）是效率、速度，及對問題的基本了解等程式品質的總括。摩爾在五十年代及六十年代所積聚的程式經驗在一九六九年結晶成爲我們現在所知的FORTH語言。FORTH乃是FOURTH（第四）的縮寫，因爲摩爾認爲這套語言是第四代的電腦語言，超越了當時方才出現的第三代電腦語言。

摩爾在一九六九年受聘於美國佛吉尼亞州的國立無線電天文台。他的任務是設計全套的中型電腦的程式系統，來控制巨大的無線電天文望遠鏡的自動運轉，由收集、記錄並加以處理這架望遠鏡所收錄的天文資料。在很短的時間內，摩爾成功地將FORTH語言裝置到天文台中各式的電腦上做各種不同的工作。FORTH語言也隨着無線電天文望遠鏡而傳佈到美國、歐洲、南美洲及澳洲各地，成爲天文界廣用的電腦語言。國際天文學會也在一九七六年決定採用FORTH作爲天文台自動化的標準語言。

摩爾在一九七三年離開美國國立無線電天文台，他邀集幾位同僚創設了FORTH，Inc.公司，專門銷售FORTH的軟體系統並提供各種特殊的軟體設計與服務。他們陸續地發展並推出爲各種中型電腦用的mini-FORTH，爲微電腦用的micro-FORTH。在一九七九年他們推出一個多用途，多人共用（multi-programming）的系統，稱爲Poly-FORTH。這是目前在FORTH語言市場上功能最高的FORTH系統。FORTH, Inc.公司雖然樂意銷售他們的FORTH軟體系統，但是爲了保護他們的權益，却不希望使用者完全瞭解FORTH語言。因此他們並不熱心於推廣這種語言。又因爲他們的產品價格相當貴，例如poly-FORTH是在美金五千元左右，所以傳佈的幅度有限。與同時發明的PASCAL相比，步調就落後了許多。

在歐洲由於天文台界的引進，許多熱心於FORTH語言的程式師組織了一個歐洲FORTH使用者協會（European FORTH User's Group）簡稱FUG。他們經常集會研究FORTH語言，並且交換FORTH程式以及實用的經驗。爲了便利使用者間交換程式及傳佈FORTH資料。他們在一九七七年制定了一套FORTH的語言標準，稱爲FORTH-77標準。這是一套基本的FORTH指令集，用其中的指令寫作的程式，可以很方便地在不同的電腦上操作。

一九七五年以後，微電腦逐漸普及，各型微電腦的使用者都在各地區成立微電腦俱樂部，在舊金山南郊的矽谷中最早成立的組織叫做Homebrew Com-Wter Club，經常在史坦福大學集會，一些使用FORTH的程式師在這個俱樂部裏經常碰面而決定自行成立FORTH的討論會。他們在一九七八年正式成立了FORTH協會（FORTH Interest Group），這是一個非營利的組織，它的目的乃是宣揚及教導這種語言，並且是針對微電腦使用者的需要而以定期的討論會及各種出版物來進行推廣的工作。

FORTH協會有兩項重要的成就。其一是他們在成立之際就組織了一個FORTH工作團（FORTH Implememtation Team），根據William Ragsdale在Apple電腦上發展出來的一套FORTH程式作爲模型，建成了六套FORTH的程式，分別用在6502、8080、9900、6800、PDP-11及PACE等六個不同的微電腦上，這就是所謂的FIG-FORTH模式。這六個FORTH系統的原始程式都在一九七九年初發表，FORTH協會將這些程式的版權公開。每份原程式僅售美金十元，因爲這低廉的價格及系統的品質，使得FORTH在美國能迅速的傳佈開來。同時FORTH協會並與Byte雜誌合作，在一九八〇年八月份的Byte中刊出了FORTH專集，使得玩微電腦的人，差不多都知道了FORTH的存在。

FORTH協會的另一項成就是FORTH語言的標準化。它在一九七八年及一九七九年兩次邀集了出版FORTH系統的公司及歐美有經驗的FORTH程式師在加州洛山磯附近的Catalira島上集會討論FORTH語言標準化的問題，這兩次會議的結果是FORTH-78及FORTH-79兩套標準。FORTH-78的語言標準規定及考慮不甚周詳，在FORTH-79標準中加以改進而成爲一套基本而定義詳密的指令集。因爲主要的FORTH公司及使用者都同意以此套標準作爲他們以後發展的基礎，使

得各種FORTH語言分歧的現象得以改善。這對FORTH語言的推展，程式的交換及資料的出版都有很大的助益。

我是在一九八一年九月回到台灣，來前收集了許多FORTH的系統程式及資料，以期提供國人這種發展軟體的犀利工具。在這半年多的時間裏，也激發了不少朋友對這種語言的興趣。大家合力在不少微電腦系統上發展出來許多FORTH的系統，我所知道的目前能使用的系統有CP/M、Apple、PDP-11、ZDS、MDS、AIM-65、68000、PA-800等。北部好些大專及機構都開始在利用FORTH做一些具體的工作。每個月第四個星期六的下午在台大電機館也固定地舉行FORTH討論會，交換研究的心得。最近Apple逐漸普及，我深感若能有一個燒錄在ROM中的FORTH系統，不必依賴磁碟機貯存程式的話，極有助於大家學習使用FORTH語言。這一套Apple-FORTH系統，具有Editor及Assembler功能而且符合FORTH-79標準。使用Apple的朋友們，可以利用它來熟悉FORTH並能做相當大的專業工作。

1-2 Apple-FORTH

Apple-FORTH 是一片ROM Card，可以插在Apple電腦主機板左後任意一個插孔中，這片ROM板上的四片2716 EPROM記憶IC，錄有一個FORTH語言系統。開機後這個FORTH系統即可開始操作。因爲整個系統已燒錄在ROM中的，所以可以很可靠地工作，如果有問題時，按RESET鍵或重行開機操作也是很方便的事。藉着這一片ROM板，可以將Apple電腦變成一個可靠而方便的FORTH電腦，供給有興趣研究FORTH的朋友一個很方便的工具。

這個Apple-FORTH的程式，是根據FIG-FORTH的6502程式建成的。它具有譯碼（Interpreter）、編譯（Compiler）、編輯（Editor）及組合（Assembler）的功能，可以用FORTH的高階語

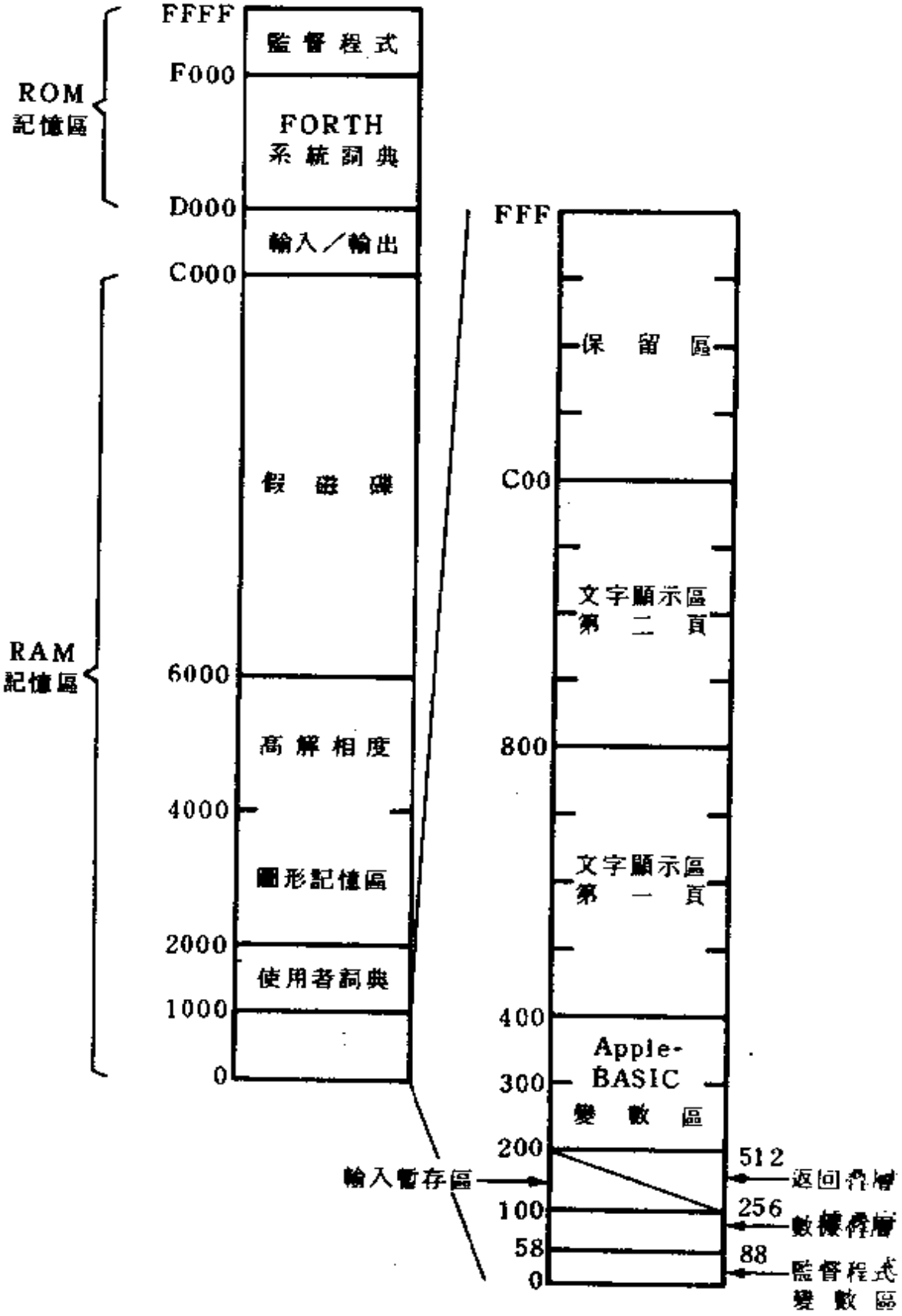


圖1-1 Apple-FORTH記憶區分圖

言來遵循結構化程式法（Structured Programming）寫作程式。它需要一個有48K RAM的Apple電腦作爲主機，並且使用一般用的錄音機作爲貯存程式的媒介。因爲Apple的磁碟機必須由國外進口，價格昂貴也不普及，所以我們這個Apple-FORTH不包括使用磁碟的指令，而設法儘量用磁帶的形式來存取大量的資料或程式。其目的乃是提供給使用者一個合理價格的FORTH系統，以便FORTH語言的推廣。

圖1-1是記憶區分圖中顯示這個FORTH系統中各部份ROM及RAM記憶的位置及其功能。Apple-FORTH實際上取代了BASIC在記憶中的地位。圖1-2是FORTH詞典中指令按其功能區分的示意圖。

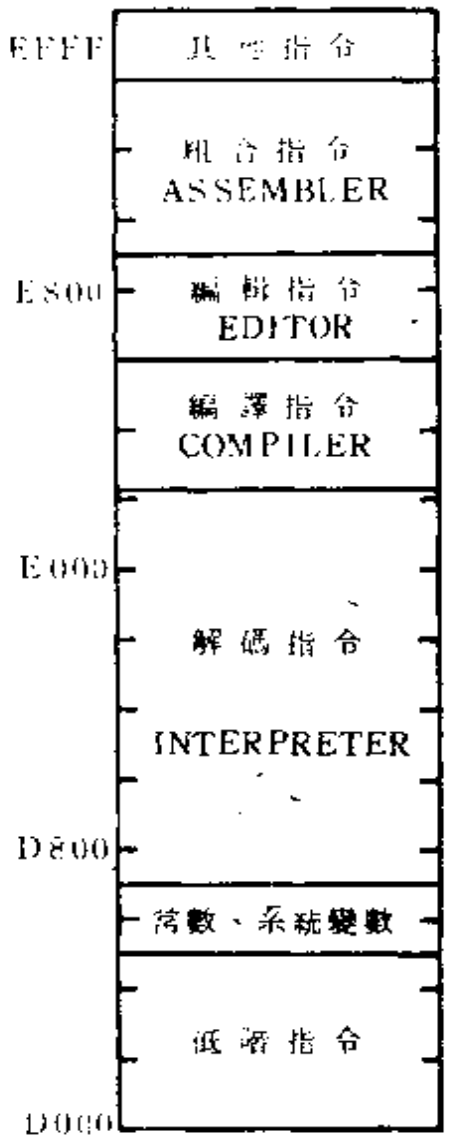


圖1-2 Apple-FORTH系統詞典及其功能區分

開機程序

將Apple主機電源切斷後，把Apple-FORTH ROM Card插入主機板後部任意一個插孔中。打開CRT螢幕機，再打開Apple主機電源，立時可以聽到“嗶”的一聲，同時在螢幕上會顯示下列字樣：

```
APPLE ][
79-FORTH 2.1
OK
```

表示已經跳入FORTH系統，等待使用者按入FORTH指令了。

連續按入Return鍵，螢幕上就顯示一連串的OK，表示FORTH系統正常的操作。按入VLIST指令後再按Return鍵即可。VLIST指令是最常用以確定FORTH系統是在正常情況下的指令。

按入MON指令再按Return，則跳Monitor系統；

```
OK
OK
MON
*
```

此時出現星號可以執行任何Apple Monitor的指令。FORTH與Monitor共存在此Apple中，可以隨意呼叫交換使用。在Monitor中我們最需要的功能是將資料存入磁帶及將資料自磁帶讀入虛磁碟記憶區。若磁帶上錄有FORTH程式，而我們希望將其讀入時，可在跳入Monitor，再鍵入以下Monitor指令，將磁帶上的FORTH程式讀回Apple系統中。如：

```
*6000.BFFFR
```

先調整適當音量並按下磁帶機PLAY鍵後再在Apple主機上按 Return 鍵，即可開始讀入資料。至星號重新顯現時，資料即正確讀入，否則有 ERR 字樣出現，表示讀取資料錯誤。

將虛磁碟記憶區的FORTH程式或資料存入磁帶中的指令是

```
＊6000·BFFF  W
```

有關磁帶機操作的詳細步驟，請參考 Apple Ⅱ Reference Manual 第 46 頁及其後的說明。

由Moniter跳回FORTH時，有好幾個跳入點，地址是開機之後建立FORTH系統的跳人點，如果FORTH系統已經在使用中（即CLOD START），許多新指令已建入詞典，則必須用下述指令進行 Warm Start的動作，以免將建好的詞典清洗掉：

```
＊D004  G
OK
```

如此回到FORTH系統而詞典的內容還是保留，可繼續原來在FORTH中的工作。跳入D000時：

```
＊D000  G
79-FORTH  2.1
OK
```

Cold Start 的動作，會將系統以外的指令清除，剩下的詞典內容與開機後的相同。同時也是清除詞典的一個方法。在FORTH系統中操作時，指令COLD也做這樣相同的動作。

假如 Return 及 VLIST 等指令動作無誤時，FORTH系統是可以正常地操作。如果系統有任何問題，VLIST 通常不能操作。在這種情

形之下，使用者應該檢查Apple本機是否能正常操作。關掉主機電源，將 Apple-FORTH ROM Card拔出來，再開Apple看BASIC是否能正常操作。一般拼裝Apple最常見的問題是RAM晶片故障，在 Apple BASIC中可用

] PRINT FRE(0)

指令列印RAM記憶最高值，若此值不在－16000附近，則必定有RAM記憶失效，將之換新可能即可正常操作。

其他問題則會可能是ROM Card插入不適當，接觸不良，此外問題多半會影響到BASIC的操作，可參照在BASIC中偵錯的方法去進行分析改正。

若主機板後方沒有ROM Card的插孔座時，使用者可以將主機板上CPU前面的四片BASIC的2716 IC拔出來，將ROM Card上的ROM拔出依序插入主機上2716的插座，FORTH的ROM #0，1，2，3分別插入D0、D8、E0、E8等位置，F8位置的ROM必需保留，這樣Apple即變成一架眞正的FORTH電腦了。這是我希望見到的事。因爲BASIC語言只是很好的學習電腦的工具，眞正高品質的程式，不宜用BASIC來發展。

學習FORTH語言，請參考STARTING一書，並且按其中例題實習，可按步就班地熟悉各個FORTH指令，並以之爲基礎發展應用程式。

錯誤信號 (Error Messages)

FORTH系統在執行時若遇到錯誤的地方，在錯誤發生的地方即終止運算而返回操作系統，並會印出一列文字：

MSG # 4

此數之值即指示錯誤發生的原因。通常發生的錯誤原因及其號數如下表：

MSG #	原　因
0	無此指令
1	Data Stack
2	詞典空間填滿
4	指令名稱重覆
17	指令不能用在定義以外
18	指令不能用在定義之內
19	結構錯誤
20	定義未能完成
21	指令在系統中不能刪除
22	指令只能在編譯磁碟區時用
24	詞彙錯誤

FORTH標準的信號文字輸出法是將信號文字儲存在磁碟記憶的第4、5兩塊（BLOCK）中，然後用 MESSAGE 的指令將某一行文字印出來。這些文字可以是錯誤信號，或是使用者自己選定的印出文字。使用這種方法輸出文字時，除了必須將這些文字資料存入磁碟記憶區之外，並要將變數 WARNING 設定爲 1。之後，所有的錯誤信息即不再以數目字表示而印出適當的文字。錯誤信息的內容及排列方式，請參閱 FIG-FORTH Installation Manual 第 20 頁。其使用法請查閱 MESSAGE，ERROR，? ERROR 及 WARNING 等指令之詳細定義及其使用法。

因爲此套 Apple-FORTH 不使用磁碟，所以使用者必須利用Editor 在第 4、5 塊中加入適當文字以後，再以下列指令

```
1 WARNING !
```

便可將錯誤訊號的文字出現，說明錯誤原因。若不再須要文字說明時，可按入

```
0 WARNING !
```

即可恢復錯誤訊號數目的輸出。

1-3 詞典 (Dictionary)

FORTH的詞典（ Dictionary ）正與其他普通的詞典一樣，含有一系列的指令（Words）及其定義。在FORTH中的指令是一串英文字母，彼此以空格（Space）分開。字母是全套的 ASCII 字母，除了退格（ Backspace ）、換行（ Carriage Return ）、冒號（ Carat ）及空格（Space）等字母之外，所有的 ASCII 字母都可以用來組成指令。許多算術符號及標點符號只要用空格分開都可以成爲FORTH的指令。例如下列的字串都是FORTH的指令：

```
FORTH , BEGIN + ? ID.
```

ASCII 字母中共有 128個字母，包括大小寫的英文字母、數目字、標點符號及許多特殊的符號，所以下列的字串也都可以用來做爲指令

```
begin （BEGIN） !BEGIN 3.1415
```

FORTH的詞典差不多佔了全部FORTH系統所使用的記憶區。詞典是一個單向串連的表，其中的每一個指令長度不盡相同，而且具備了這個指令的完整定義或執行時所需的全部資料。詞典可以增長，而向高記憶區擴張。詞典中也可以再分支而成好幾個詞彙（ Vocabulary ），

容納相關的指令集。

新的指令是一些特殊的定義指令（Defining Words）加進詞典裏去。最常用的定義指令是：（Colon或冒號指令）。系統在執行冒號（：）指令的時候就在詞典的頂頭上建造一個新的指令，其中所含的是一系列詞典中已有指令的地址。新指令是以分號（；）指令爲其終點。

例如在詞典中已建有OPEN SHUTTER CLOSE EXPOSE TIME等指令，分別做控制照相機特定的動作，如欲將所有動作連在一起執行則可建造一指令如下：

```
: PHOTOGRAPH SHUTTER OPEN TIME
  EXPOSE SHUTTER CLOSE ;
```

1-4 疊層（Stacks）

FORTH系統使用兩個疊層來暫存操作時使用的數據及地址等資料，所以一個疊層稱爲數據疊層（Data Stack），另一個稱爲返回疊層（Return Stack）。數據疊層是最常用的疊層，所以如果我們不特別申明的時候，疊層就是專指數據疊層而言的。

疊層是電腦中一個資料儲存的區域及存取資料的機構，或稱爲一個後入先出（Last-in First-out）的記憶區。要說明疊層的用法，我們可以在終端機上按入下列字串：

```
2 4 6 8 （CR） OK
```

在按下換行（CR）鍵之後，系統立即回應 OK 字樣，表示所有在CR以前的指令都已執行無誤。四個數值現在都存放在數據疊層上了。最先按入的數值（2）是在疊層的最底下，而最後進去的8是在疊層最頂端。現在如果鍵入：

. . . . (CR)

FORTH電腦立即會回應：

8 6 4 2 OK

.（Dot）是一個FORTH的指令，將疊層頂端的數值取出而在終端機上顯示出來。第一個．指令將8印出來，以此類推印出四個數目字。最先按入的2是最後印出來的數值。

圖1-3是FORTH電腦系統中記憶功能的示意圖，其中用了兩個疊

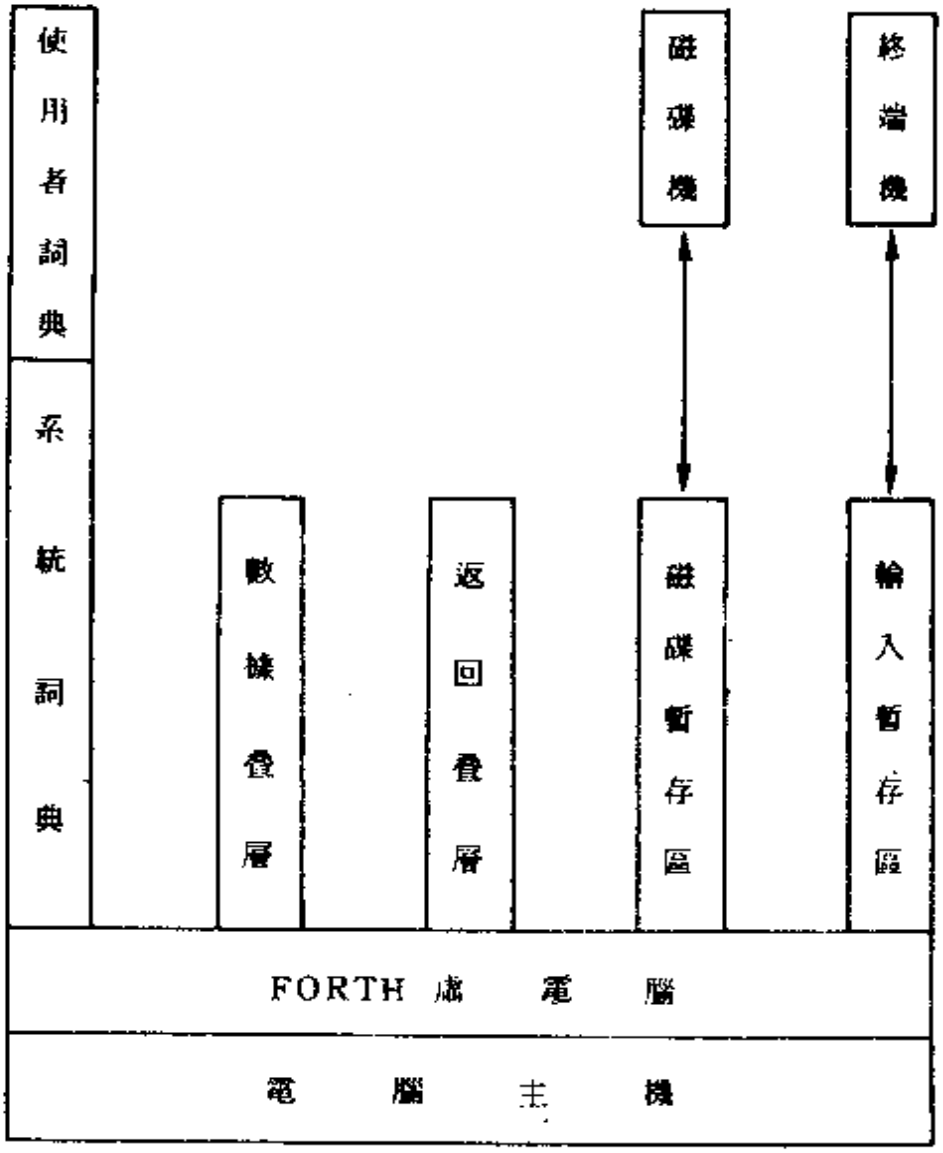
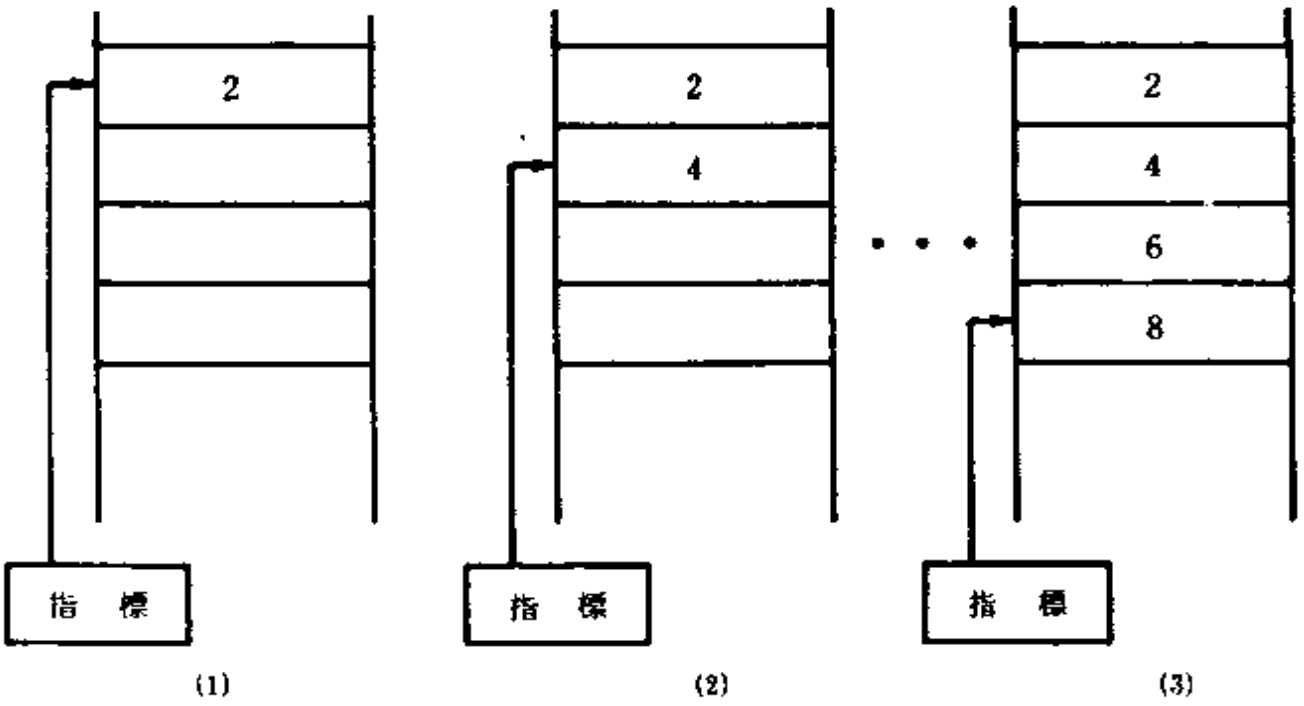
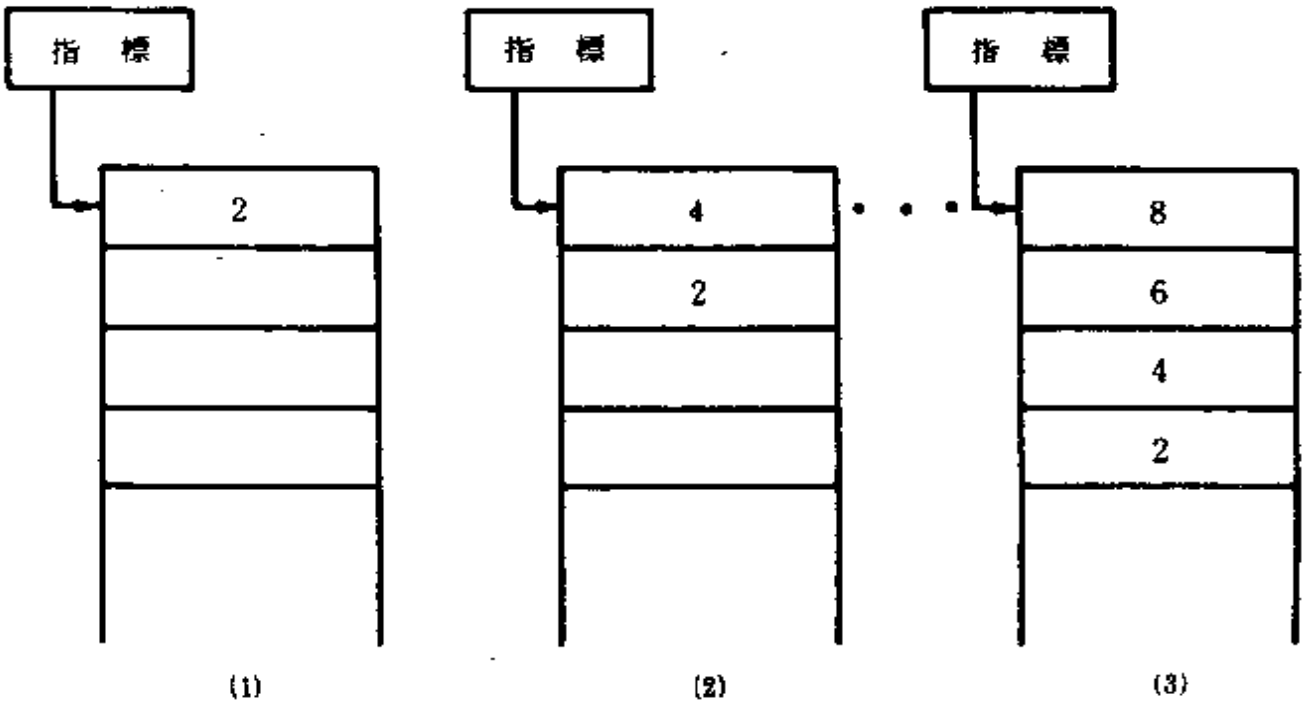


圖1-3 FORTH電腦系統

層。圖1-4是顯示疊層的兩種不同的表示方法。圖1-4(a)是由記憶區的觀點來畫的疊層圖。先按入的數值放在高記憶區，後來按入的數目加在其下，所以疊層是往下延伸的。



(a)記憶區圖示法



(b)示意圖示法

圖1-4 後入先出的疊層，在疊層上依次加入2、4、6及8四個數值的過程

圖 1-4(b)則是一個疊層的示意圖，其疊層是由下往上延伸的。好像餐館中碟子的擺法，後擺的碟子先被取用。我們在討論疊層時是用這一種說法。因此，在疊層上新加一個數值時我們說將一個數值堆上疊層。而由疊層上取用一個數值時我們說取下數值。

1-5 數據疊層 (Data Stack)

FORTH的指令所須用的數據資料大都是由數據疊層上索取的。數據資料通常是用以下三種方法置放到疊層上去：

(1) 由終端機上鍵入的數字都放在疊層上。

(2) 在原程式中的數字也都會被放上疊層。

(3) 許多指令都會將它們操作後的結果數值留在疊層上。

> 因FORTH指令必須將它們自己所用的數據由疊層上取下使用。所有的指令都只能將它們所特別產生的結果資料留在疊層上。

許多FORTH的指令都要用到在疊層上的數據資料，須用數值的數目則視指令而定。所以使用者在下達一個指令給電腦執行之前，必須十分清楚他在數據疊層上留着有適當的數據，且其排列也必須有一定的順序才行。FORTH的數值可以有許多不同的型式，其用法完全決定在使用數值的指令，圖1-5顯示FORTH數值的幾種主要的型式。

在疊層上的數據，可以用表1-1中的疊層操作指令來搬動改變其順序。順序改變的目的，乃是將須要使用的數值搬到疊層頂上，以便以下的指令使用，或是複錄一些數值免得被以下的指令使用而被移去。

表1-2中的指令，是將疊層頂上的數值列印在終端機上用的。這些指令可以很方便地將疊層上的數值內容顯示出來，以備使用人檢視。這些輸出的指令我們還會在第二章討論。

等我們學完了再下一節的〔後算符運算〕(Postfix Notation)

型　　式	代　號	範　　圍
旗　號 (Flag)	f	零或非零
字　碼 (Character)	c	0 至 127
字　元 (Byte)	b	0 至 255
數　元 (Number) 或單整數	n	-32768 至 32767
正整數 (Unsigned Number)	un	0 至 65535
雙整數 (Double Number)	d	-2,147,483,648, 至 2,147,483,647
正雙整數 (Unsigned Double Number)	ud	0 至 4,294,967,295
地　址 (Address)	a	0 至 65535

圖 1-5　FORTH的數值型式

之後，希望學者能將這幾類的疊層操作指令多加練習，充分消化之後便可以應用自如了。

表列說明：

疊層變化　（指令執行前疊層數值　　執行後疊層數值）

疊層頂上之值在最右邊。

n	16 位元單整數	b	8位元之字元
d	32 位元雙整數	f	布爾旗號
u	16 位元單正整數	c	ASCII 字碼
ud	32 位元雙正整數	addr	地址

表 1-1 數據疊層指令

指　　令	疊 層 動 作 說 明	中文指令
DUP (n — nn)	複錄疊層頂上的數值。	重
DROP (n —)	棄去疊層頂上的數值。	棄
SWAP (n1 n2 — n2 n1)	交換疊層頂的兩個數值。	換
OVER (n1 n2 — n1 n2 n1)	複錄疊層頂上第二個數值。	疊
ROT (n1 n2 n3 — n2 n3 n1)	將疊層頂上第三個數值轉到最頂上。	轉
?DUP (n — n(n))	若疊層頂值不爲零則複錄一次。	?重
PICK (n1 — n2)	將疊層頂第 n1 個數值複錄一次。	選
ROLL (n —)	將疊層頂的 n 個數值滾轉，使第 n 值到頂上。	滾
DEPTH (— n)	將疊層上現有數值的數目放在疊層上。	深
2SWAP (d1 d2 — d2 d1)	交換疊層頂上兩個雙整數。	雙換
2DUP (d — dd)	重複疊層頂上的雙整數。	雙重
2DROP (d —)	棄去疊層頂上的雙整數。	雙棄
2OVER (d1 d2 — d1 d2 d1)	重複疊層頂上第二個雙整數。	雙疊

表1-2 數值輸出指令

指 令	疊 層 動 作 說 明	中文指令
D.R	(dn— 將雙整數d列印在n欄中，右邊對齊。	右雙點
D.	(d— 將雙整數d列印出來，右邊加一空格。	雙點
U.	(un—) 將正整數un印出來，右邊加一空格。	正點
.R	(n1 n2—) 將整數n1印在n2欄中，右邊對齊。	右點
.	(n—) 將單整數n印出來，右邊加一空格。	點
?	(a—) 將地址a中的單整數印出來，右邊加一空格。	問

1-6 返回疊層 (Return Stack)

FORTH系統利用返回疊層來儲存下一個將被執行的指令的地址，這個返回疊層和一般電腦系統中的疊層相似，因爲一般電腦的疊層是在主程式呼叫副程式(Subrautihe)時儲存主程式下一指令的地址。返回疊層是專門用來控制指令間呼叫及回返的動作之機構。但使用者也可以用返回疊層來做一些額外的工作：

(1) 暫存循環用的指標及上限。

(2) 暫存數據疊層上不易處理的數值。

(3) 其它系統操作時的數據資料。

返回疊層與系統的操作有極密切的關係，使用時必須十分小心。誤用數據疊層，會得到不正確的數值答案，誤用返回疊層，是使FORTH系統

當掉（ down ）最方便的辦法。

返回疊層詳細的使用法及其相關指令將在第五章中討論。

1-7 後算符算法 (Postfix Notation)

FORTH語法十分簡單，其主要原因是指令之間參數或數據資料的傳送是經由數據疊層，避免使用大量的常數及變數做爲傳資料的參數。即使在十分複雜的應用程式中，數據疊層可以供給在不同層次間的指令交換資料。爲了要充分發揮疊層的功用，FORTH使用後算符算法（ Postfix Notation ）來表示運算的步驟，而不用大家比較熟悉的代數法或是內算符算法（ Infix Notation ）。後算符算法的特色就是運算的指令必須放在運算參數之後。

FORTH中的算術指令都是以在數據疊層頂上的數值做爲操作的參數。例如下面的指令

```
3  1  -
```

是先將3堆上疊層，再將1堆到3之上。減法指令“－”則將3減去1而將結果2留在疊層上。此時3與1兩值已被“－”指令運算過而回疊層上除去。

有些算術指令是沒有交換法（ Non - Comutative ）的，例如減法指令“－”及除法指令“／”。它們的用法是將指令符號塞在兩個參數中間，即是疊層最頂上的數值是減數或除數而其下的第二值是被減數或被除數。下面是三個簡單的例子：

FORTH後算符算法	內算符算法
3 1 -	3－1
24 3 4 */	24＊3/4

24 3 4 * / 24/(3*4)

請注意在兩種算法中，參數排列的方式是相同的。在後算符算法中，我們不需要用括弧來改變計算的順序，計算的順序完全決定在算術指令排列的先後次序而絕對不會像內算符法中指令操作的順序可能會混淆不清。

在上面第二個例子中的指令 */ 是FORTH系統中一個獨立的指令。它將疊層上第三值乘以第二值之後再用頂上第一值去除，乘除的次序即如內算符算法所列。在第三個例子中我們須要特別注意指令的順序。此處*及 / 是兩個獨立的指令，依其先後次序來執行。所以在執行*指令時，它是將此刻在疊層頂上的3與4相乘，得到的乘積放置於疊層上。執行 / 指令時，即取 24 被 12 除。其結果亦放置於疊層上。

由這些簡單的例子來看，後算符算法好像是很麻煩。但在解比較龐大的問題時，這種方法可以使工作簡化。內算符算法是用來「敍述」一個算術的問題，但是後算符算法是定法問題的解法。例如在FORTRAN語言中，使用者可以用代數式來敍述他所希望得到的結果，而FORTRAN的編譯程式（Compiler）必須要設法將這個代數式翻譯成爲電腦解決這個問題的詳細步驟。許多FORTRAN的使用者花很多時間在摸索這個編譯程式究竟是如何處理他寫的式子，而編譯程式也在摸索使用者所寫的式子的眞正意圖。兩邊猜測摸索的結果，與使用者的原意可能會產生很大的出入。

使用FORTH時，使用者直接控制每一步的運算。後算符法不但避免了代數法中不明確獨一的表示方式，也提供了結構程式法中需要的彈性，以簡化程式單元間資料交換的工作。所以每個指令只要有正確的數據參數在疊層上供它使用就可以得到正確的答案，不管這些數據參數是在操作時那個層次上的指令所留下來的結果。這樣的指令能由任意的其他指令獲取它所需要的數據參數，它就不受語意的限制（Context-free

）而可以廣用在不同的程式及單元中，它的功用就更廣泛而普遍。FORTH的指令大多數都是沒有語意限制的。產生數值的指令只是將其結果留在疊層上，讓後讀的指令自行取用。

下面的例子是在一個實驗室中電腦必須控制許多儀器的溫度。這個例子只是用來說明怎樣利用疊層來傳送資料訊號。其中有一段程式單元可能是如此定義的：

```
: HEAT CELSIUS ?SHUTDOWN FAHRENHEIT
       TABULATE ;
```

其中各別指令的意義及功能是：

CELSIUS	由疊層上的裝置號碼，測定裝置的溫度（如 3 HEAT）。它量好攝氏溫度之後即將溫度與裝置號碼一起留在疊層上。
?SHUTDOWN	假如此裝置的溫度超過了一個極限溫度，即將此裝置停電。它會再留下一個當時正確的溫度值在疊層上。
FAHRENHEIT	將疊層上的攝氏溫度轉換成爲華氏溫度值
TABULATE	將華氏溫度值印在印表機上。

HEAT指令的內容十分簡單，我們也不必另行說明或在程式中寫明疊層上的值如何變化。輸入及輸出的資料都不必另外用變數的方法來儲存。當然FORTH也提供使用者定義新的變數或常數的工具，這些常數變數的問題將留到第四章時再詳細討論說明。

最自然的電腦語言必須能夠精確地表示出使用者的設計構想。在使用結構化程式法（Structured Programming）的原理來製作程式時，FORTH讓使用者很方便，很直接，而且很簡潔地告訴電腦他要電腦

做的工作。

1-8 FORTH名詞集解

FORTH語言及系統與一般電腦不同，因此它也有些特殊的名詞，其用法與一般電腦之用法有異。這兒我們列出一些名詞及其意義或用法，讀者可以不時瀏覽，對瞭解本書內容將有很大助益。

地　址 （Address）	電腦選擇記憶資料時輸出的選擇訊號，一般是 16 位元，可以選擇 65536 個字元的資料。
應用程式 （Application）	一組由使用者定出的指令，用以解決使用者的問題。相當於一般語言中寫作的程式。
塊 （Blocks）	FORTH儲存大量資料的單位。每塊通常有1024字元的長度，在磁碟上是按順序排列的。
位　元 （Bit）	電腦記憶的最小單位，是二進位制中的一位數字，相當於一個正反記憶器。
字　元 （Byte）	八個位元合成的記憶單位，是微電腦的基本記憶單位。
數　元 （Cells）	16 位元合成的記憶單位，是FORTH中常用表示數值的基本單位。一個數元可記憶一個單整數。
字　母	美國標準通訊用字母（ASCII）。共有128

（Character）	個不同的字母或符號，以7個位元表示。在一個字元中較低的7位元可用來表示一個字母。
編 譯 （Compilation）	由原程式文字建造新指令加入FORTH系統的過程。它是與執行（Execution）相對的。在編譯時，系統不立即執行程式中的指令而是將之編入詞典。
詞 典 （Dictionary）	FORTH的詞典是系統中全部指令的總集，包括系統原有的指令及使用者加入的應用指令。FORTH的詞典是存在電腦主記憶中，且經過編譯的指令集是可以立即執行的。
磁 碟 （Disk）	泛指FORTH系統使用的大量資料儲存裝置，可以是軟磁碟、硬磁碟或是磁帶。
執 行 （Execution）	FORTH系統實際上進行某個指令所規定的工作。執行低階指令是執行其中的機器碼。執行高階指令是按照其中所列的地址依序執行每個地址中的指令。
輸入字串 （Input String）	供給FORTH的文字解碼程式（Text Interpreter）處理的文字串。通常是指在終端機輸入暫存區中的文字，但也可以是在磁碟記憶區中的文字資料。
解 碼 （Terpretation）	通常指文字解碼程式的工作過程。即是將輸入字串中各個指令分辨出來，立即執行或轉

	換爲一個數值。亦可指各種不同指令實際執行的過程，這一些指令都是經由其個別的內解碼程式（ Inner Interpreters ）來執行的。
載入譯碼（ LOAD ）	載入是指將磁碟上所存的程式文字編譯加入詞典或執行磁碟上所存指令的過程。
輸出字串（ Output String ）	即將要輸送至終端機或印表機去的文字資料，或是文字資料用 TYPE 指令列印者，或是一個內存數值轉換成輸出文字過程中建造的字串。
疊　層（ Stack ）	一個後入先出的數字資料暫存記憶區。FORTH系統均有兩個疊層：(1)數據疊層用以暫存數值資料；(2)返回疊層用以存放回返高階指令時的地址資料。
系統變數（ User Variables ）	系統在操作時須用的一組變數資料。在系統開始時這些變數不須有合適的起始值。在多人共用的FORTH系統中，每一個使用者均有一套此種變數資料，故稱爲User Variable 。
詞　彙（ Vocabulary ）	詞典中獨立連結成集的一組指令。在一般的FORTH系統中都有 FORTH ， EDITOR 及ASSEMBLER三個詞彙。
指　令	在詞典中一個可以獨立執行的個體。每個指

令都有其特殊的名稱，及所有執行它所需要的一切資料。

練習題一

將下列式子改寫爲後算符算法的順序。例如

(a * b) + c → c b a * +

(a + b) * c →

$\frac{3a-b}{4}+c \rightarrow$

$0.5 * a * \frac{b}{100} \rightarrow$

$a^2 + ab + c \rightarrow$

2. 將下列後算符法表示的算式改爲內算符式：

n 1+ n/ →

5152 6 100 */ 3 − →

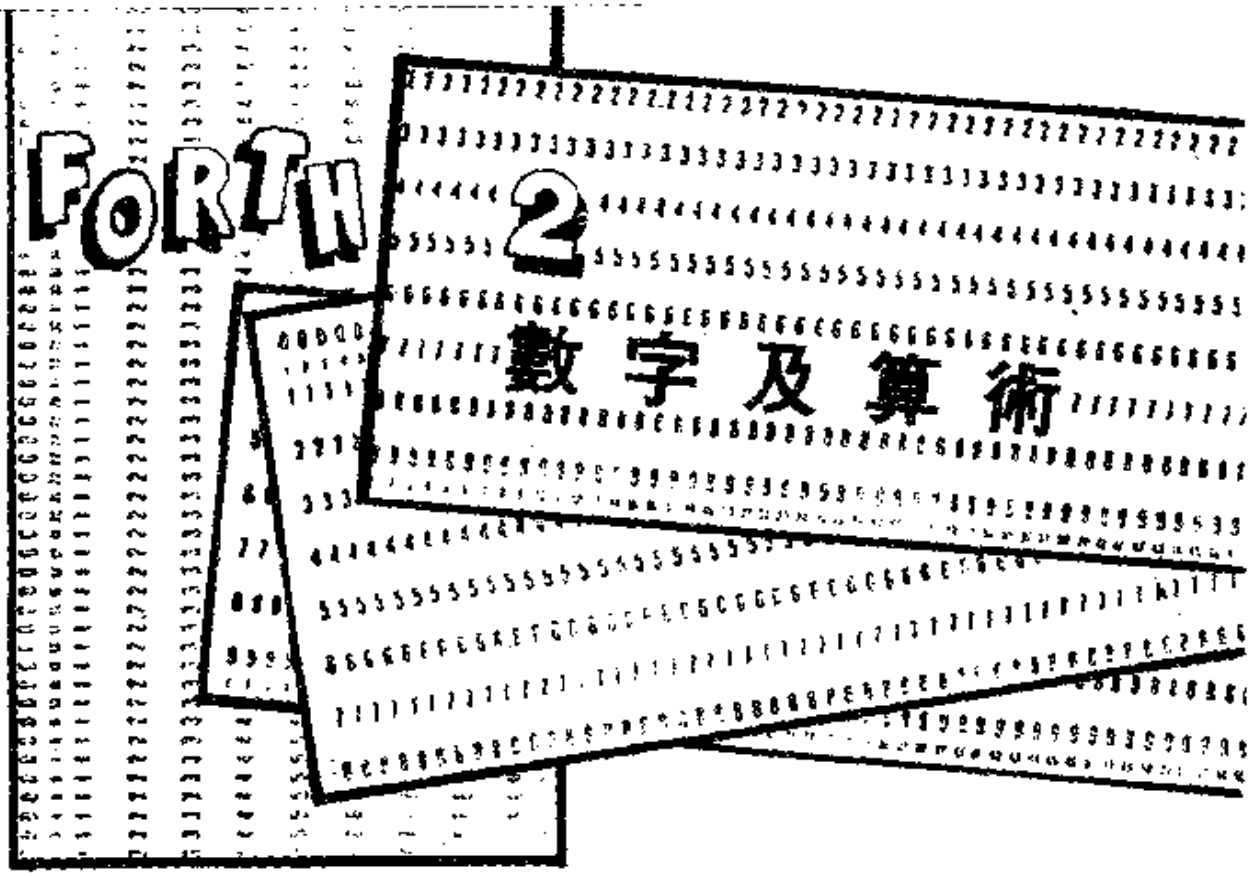
7x * 5 + x * →

a b − b a + / →

a b 10 * / →

3. DUP DUP與2DUP 間有什麼不同？

4. 試寫一指令可以將疊層上面三個數值複錄一遍。如 n1 n2 n3 → n1 n2 n3 n1 n2 n3 ）。盡可能簡化此指令，並將其轉錄到磁帶上以便長期儲存。



我們已經簡單地介紹了詞典、疊層的指令及後算符算法。現在我們要討論 FORTH如何使用數字進行各種算術演算。在學習本章時，最好能使用 FORTH電腦的終端機時練習並定義所需要的應用指令。我們暫時不用磁碟來儲存程式資料，這是下一章中討論 EDITOR 時要做的事。本章中所有的練習，都可以直接由終端機輸入。

2-1 數字的輸入及輸出

FORTH的算術主要是處理整數的數字，大多的數字都以 16 位元的數元來表示。雖然有些FORTH系統有額外的指令能夠做浮點數字的算術，但爲了要充分利用微電腦做整數算術的速度，一般FORTH系統都只供給整數的算術指令。FORTH算術指令集的設計是在提供使用者一套算術工具，使能很方便地用整數算術來解決各種問題，但不犧牲計算的速度或精密度。

當FORTH系統接受了一個整數數字（或由終端機輸入或由磁碟中輸入），它立即將此數字轉換爲二進位形式並將它堆上數據疊層。輸入的數字可以是 16 位元的單整數或是 32 位元的雙整數。解碼程式若見到一個數字串中有一個小數點時，即將此數當做雙整數處理。若數字串中沒有小數點，則將它當做單整數處理。

2-1-1 單整數

FORTH能辨別單整數（或正或負）及單正整數。負數以二進制補碼（Two's Complement）來表示。單整數的範圍是由−32768到32767，單正整數的範圍是0到65535。不熟悉二進位制的讀者可以參考圖1-3中的一些說明。

在上一章中我們已經介紹過了印出數字的指令“．”號。它將疊層最頂上的數值移去而把它轉換成爲數字串，由終端機上列印出來。列印的方式是當它做一個單整數，即是若數值大於 32767 時即當做一個負數印出來。

請用你的終端機練習以下的習題。現在開始熟悉你的FORTH系統對你以後學習寫程式有很大的幫助。

請你按入	電腦回應
0 .	0 OK
1 .	1 OK
−400 .	−400 OK
32767 .	32767 OK
32768 .	−32768 OK
32769 .	−32767 OK

最後兩行顯示當數值超過 32767 時FORTH當它做負整數處理的

結果。若要將負數當做正數列印時，應該按入以下的指令：

```
32768  0  D.  32768 OK
```

我們將 32768 上堆上一個零，即將此數變成一個雙整數，再 D. 列印這個雙整數時，即得 32768 之值。

一個數值究竟是用作單整數還是單正整數是完全決定於用這個數值的指令，也就是說使用者必須要選用合適的指令來處理這些數值。數值的使用，使用者必須負完全的責任。

2-1-2　雙整數

輸入的數字串中如果沒有小數點，FORTH就當做單整數。若其中有一個小數點時，FORTH就當它做雙整數來處理了。例如

```
65.536
```

即是一個 32 位元的雙整數。FORTH只認得小數點及小數點所在的位置，小數點並不影響數值的轉換，因此，以下所有的數值轉換之後都成爲相同的雙整數：

```
12345.    D.  12345  OK
123.45    D.  12345  OK
1.2345    D.  12345  OK
1.23.45   D.  12345  OK
```

第一個小數點後面的字數是記在一個系統變數 DPL 中的。所以使用者如果要確定小數點後的字數用以計算時，就可以由 DPL 中找出這個字數來做下一步的計算工作，這是一個最原始且十分好用的浮點指標。

一個雙整數也可以當做兩個單整數來處理。因爲雙整數是用兩個連

續的單整數來代表的。如果我們按入：

```
65.536  .  .  1  0  OK
```

按入的兩個“．”號指令是將雙整數 65536 分做兩部份印出。先印出 32 位元中的上半部 16 位元，因爲高半數是在疊層頂上，第二個由“．”號指令印出的是下半部。再練習一下：

```
7．D．7 OK
7．．．0 7 OK
```

小數點就將一個數字（不管它多小）變爲成雙整數。雙整數有 32 位元，它的範圍中從－2,147,483,648到2,147,483,647。涵蓋相當大的一個數值範圍。

2-2 算術及比例指令

2-2-1 算術指令

表2-1列出了一些FORTH的算術指令，分爲單整數指令、雙整數指令及混合指令三類。使用者需要用的算術指令依應用目的的不同而有很大的差異。FORTH也不可能提供一套「完整的」算術及邏輯指令集。FORTH系統的設計者只供應一套最常用的算術指令集，但希望使用者按自己的需要另外加上特殊的指令，以最方便的方法解決他的問題。

許多FORTH的供應商在推銷各式各樣的算術指令集及程式，包括浮點算術、分數、三角及高級函數，傅利葉變換及微積分等等，供使用者選擇。但在FIG-FORTH及 79 FORTH標準指令集中只有整數的算術指令。

2-2-2　比例指令

在FORTH的算術指令中有一個很特別的指令是 ＊/ ，是專門用來做比例的計算的。這個指令使得我們不再需要用浮點算術。使用這個比例指令的好處是它比浮點算術要快好幾倍。另外一個好處是單精密度的浮點數要佔用兩個數元，而其精密度只有六位數。如果用相同的兩個數元來做一個雙整數的話，精密度可以超過九位數。

比例指令 ＊/ 是將兩個單整數相乘而以第三個單整數去除其積。中間的積是一個雙整數，因此商數具有 16 位元的精密度。假定我們要將長度由厘米（ mm ）換算成爲mil（千分之一時）的單位，我們可以定義一個新指令MM，它的功用是：

```
: MM  10000  254 */  ;
```

如果要算20mm是多少mil時，只要按入

```
20  MM
```

即可得到換算結果。算百分比時我們可用以下的指令：

```
: %  100  */  ;
675  15  %
```

即可算出675的 15 % 之值。類似的定義可以用來自動修正實驗所得的大批數據資料。

許多問題中，輸入的大量資料可以用相同的比例來處理，使其結果可以用簡單的整數來代表。例如在商業簿記中銀錢的往來帳目都可以用分來做單位。在輸入以元爲單位的帳目時，可以在元數後多加兩個零。實驗室中可取用 1/10 秒，千分之一伏特等等做爲計數的單位。這樣所

表2-1 算術及邏輯指令

指　令	疊層動作說明	中文指令
+	(n1 n2－n3) n1加n2，和留在疊層上。	加
－	(n1 n2－n3) n1減n2，差留在疊層上。	減
1 +	(n－n＋1)	加一
1 －	(n－n－1)	減一
2 +	(n－n＋2)	加二
2 －	(n－n 2)	減二
*	(n1 n2－n3)	乘
/	(n1 n2－n3) n1被n2除，商留在疊層上。	除
/MOD	(n1 n2－n3 n4) n1被n2除，餘數n3與商n4留在疊層上。	餘
*/	(n1 n2 n3－n4) n1乘n2再被n3除，中間積是一個雙整數，商數留在疊層上。	乘除
*/MOD	(n1 n2 n3－n4 n5) 與*/相同，餘數n4及商n5留在疊層上。	乘除餘

表2-1　算術及邏輯指令（續）

指　　令	疊　層　動　作　說　明	中文指令
U*	(un1 un2 — ud) 將兩正數相乘，其雙整數積 ud 留在疊層上。	雙乘
U/MOD	(ud un1 — un2 un3) 將雙整數 ud 被 un1 除，留下餘及商在疊層上。	雙除餘
MAX	(n1 n2 — n3) 將n1及n2之大者留在疊層上。	極大
MIN	(n1 n2 — n3) 將n1及n2之小者留在疊層上。	極小
ABS	(n1 — n2) 將 n1 之絕對值留在疊層上。	絕對
NEGATE	(n1 — n2) 將疊層頂值變號。	負
AND	(n1 n2 — n3) 留下n1與n2每位位元的邏輯和值。	與
OR	(n1 n2 — n3) 留下n1與n2每位位元的邏輯或值。	或
XOR	(n1 n2 — n3) 留下n1與n2之邏輯反或值。	僅或
D+	(d1 d2 — d3) 將二雙整數d1加d2，和d3留在疊層上。	雙加
DNEGATE	(d1 — d2) 將疊層頂之雙整數變號。	雙負
DABS	(d1 — d2) 留下疊層頂之雙值的絕對值。	雙絕對

有的資料都可以用整數法來表示及運算，只要在輸出時注明資料數據的小數點位置（在 DPL 中）即可。

這種「整塊浮動」（Block Floating）的技術是很方便的。它用一個整數來記住小數點的位置，整塊的資料都用這一樣的比例來處理而得到正確的結果。但用這種方法，所有的數據資料就都可以用一個16位元的數元來代表。且許多問題都可以用這種方法來解決。

2-3 數字的轉換

開機以後FORTH系統是用十進位法來將輸入的數字換算成電腦需用的二進位數值。在輸出時則須將在疊層上的數值轉換成數字的文字串才能顯示在終端機或是印表機上，我們按着需要可以很方便地改換數據轉換資料爲一個字串以備程式的輸出之用。

```
: OCTAL 8 BASE ! ;
: HEX 16 BASE ! ;
: BINARY 2 BASE ! ;
: DECIMAL 10 BASE ! ;
```

要換用其他的數字轉換法時只要下達OCTAL（轉爲8進位），HEX（16進位）或 BINARY（二進位）指令即可。一旦改變了轉換的基數（BASE），所有的輸出輸入的數字就都是用新的基數來轉換了，直到我們再設定一個新的基數爲止。所以我們要養成一個習慣在其他種進位法中的工作做完時要立即按入 DECIMAL 指令回到十進位制。

BASE是一個系統變數，儲存着現在做輸入輸出轉換的基數。若使用者要選擇以上各種基數以外的轉換法，只要將他所要的基數值存入BASE：

```
n BASE !
```

要輸入的數值能成功地轉換爲機器內的二進位數，數字串中的各個數字必須小於BASE中的基數值，或者是小數點．號。否則系統會發生錯誤而致停止執行輸入的指令串。在使用十六進位（HEX）時，最好將每個用A至F開頭的數值前面加一個數字0，如

```
0A 0B 或 0FF
```

這樣可以很方便區分這些數值與系統原有的一些指令如B、C、D或F而不再做轉換的動作。

2-4 實值 (Literals)

在一個指令的定義中的數字是以實值的方式編入詞典的。實值就是一個數值轉換成爲二進位值編入詞典的方式，在此指令執行時，實值就會將這個二進位的數值堆上疊層以供後續的指令來使用。在編入實值時，詞典中編入兩個數元，第一個數元是指令LIT的執行地址（Code Field Address或cfa），第二個數元中即是存轉換出來的二進位值。

當我們編譯一個高階指令時，轉換數字所用的基數是BASE在編譯過程中所存的數值。例如我們如果執行下列的指令：

```
DECIMAL
: EXAMPLE HEX 20 . ;
```

HEX 後面的數目 20 是以十進位的方式轉換的不是十六進位。在定義中的 HEX 指令並不馬上生效而是被編入詞典去的，當時的基數不是十進位的 10。只有在執行EXAMPLE的時候，BASE中的數值變成16，之後所有的數值轉換都採取 16 進位的方式，所以十進位的 20 印出來

卻是十六進位的 14 了。

```
EXAMPLE 14 OK
```

在 Apple FORTH 系統中，定義裏若遇到雙整數，就會用雙整數的賦值方式編入詞典。雙整數賦值的編譯產生四個數元：第一個數元是指令LIT；第二個是雙整數的下半；第三個又是LIT；第四個數元是雙整數的上半。所以在執行這個新的定義的時候，雙整數的下半先堆上疊層，然後它的上半再堆上去，正好是雙整數在疊層上要求的形式。

練習題二

1. 十進位的 1234 在八進制中應是多少？在二進位制時應是多少？
2. 我們按入下列指令時其結果爲何？

   BASE ?

   BASE .

   爲何有不同的結果？
3. 2 * 65535 + 3 若用雙整數來表示時，它的上半個十六位元的單整數值是多少？它的下半個數元又是多少？
4. 執行下列指令後，疊層上留下了那些數值？

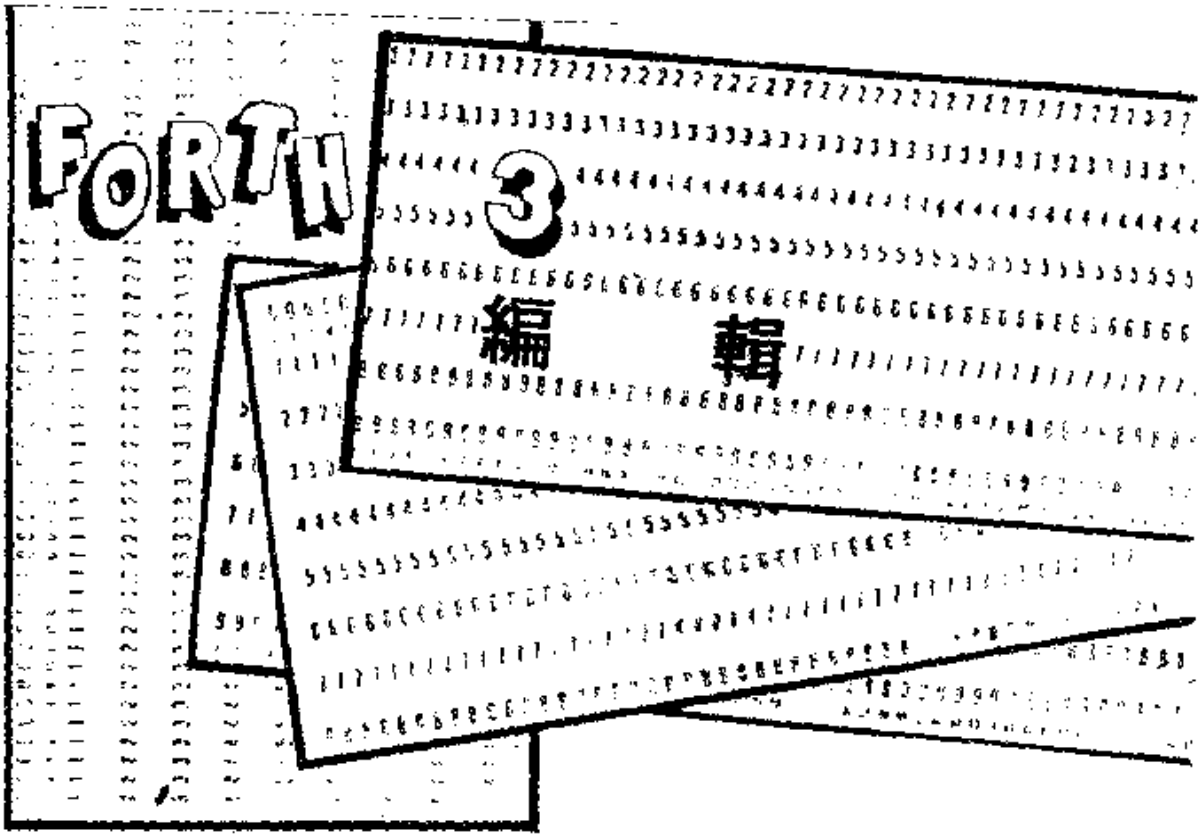
   15 12 NEGATE

   15 12 -

   15 12 D-
5. 試計算下列百分數：

   $50,040的6%

   $50,040的6½%
6. 將第一章中各個問題的解寫成新的指令或定義。這些定義在我們做第三章的習題時是練習使用編輯指令的好例題。



本章討論我們如何將原程式的文字存在磁碟記憶區中，如何將程式載入（ Load ）詞典，如何測試這些新建的指令，以及如何將這些程式文字列印在終端機上。我們先討論這些編輯的指令是爲了方便讀者現在就可以開始在磁碟記憶區中撰寫練習的程式。

3-1 文字的編輯工作

我們雖然可以隨時在終端機上按入一些文字，建造新的指令，這些加入的新指令是以內碼的方式編入了詞典，原來按入的原程式文字就都被清洗乾淨了。因此我們無法再將原程式文字叫回來檢查或修改。要留下原程式的文字以備以後修正使用的話，程式文字必須寫在磁碟記憶區中，以便修改測試或應用。若要將程式文字或資料永遠儲存起來，這些磁碟記憶區的文字或資料可以用Apple監督系統的W指令存到磁帶上去。以後這些資料還可以由磁帶上讀回到磁碟記憶區去，以備應用。

表 3-1 編輯指令

指　　令	疊 層 動 作 說 明	中文指令
T	(—) 印出第 n 行，並將指標移至行首。	印
P ×××	(—) 將字串×××置於指標所在之行（現用行）。	置
U ×××	(—) 將字串×××插於現用行之下，以後各行皆退一行。	插
X	(—) 刪去現用行，以後各行皆進一行。刪去文字存於 PAD 暫存區中。	刪行
F ×××	(—) 由指標向後找尋字串×××，並將指標置於尋得字串之後。無此字串時，指標回置第 0 行之首。	尋
D ×××	(—) 由指標向後刪去字串×××。	刪
I ×××	(—) 將×××塞入指標現在位置。	塞
TILL ×××	(—) 將指標至×××間之文字全部刪除。	刪至
COPY	(n1 n2 —) 將第n1 塊資料抄至n2 塊中。	抄錄
CLEAR	(n —) 將第 n 塊清除爲零。	清除
TOP	(—) 將指標移至第 0 行之首。	頂
L	(—) 將現用塊文字重新列印。	列
LIST	(n —) 將第 n 塊列印出來，並將其設定爲現用塊。	列印
TRIAD	(n —) 將第 n 塊附近三塊文字全部列印。第一塊之序數需被 3 整除。	參列印
INDEX	(n1 n2 —) 將第n1 與n2 塊之第 0 行文字列印出來。	目錄
CONTROL I	(n —)) 將指標移動 n 個字元。Control I 即是 Tab 鍵。	

Apple-FORTH 系統中用 24K的RAM 做一個假磁碟。這個假磁碟是分做 48 個塊，每塊有512字元的容量。各塊用 0 至 47 的數字依順序標示。每塊中的文字在Apple的螢幕上顯示時是分爲 16 行，每行 32 個字，行數是由0到 15 。

當要做編輯的工作時，我們應按入EDITOR指令，這樣就選擇了編輯詞彙作爲我們的語意詞彙（Context Vocabulary）所有的編輯指令都可以叫出來使用。在使用 UST 指令時，編輯詞彙也是自動被選做語意詞彙。編輯指令彙集在表3-1中。

當我們選擇第 n 個塊做編輯的工作時，應該按入：

```
n LIST
```

的指令，n 是0 到 47 間任何的塊數。UST 將 n 存在系統變數SCR中，將第 n 塊設爲現用塊並將此塊中文字列印在螢幕上。一旦第 n 塊設定爲現用塊之後，一個更簡單的指令：

```
L
```

就可以將此塊中之文字顯示出來。

3-1-1 字串暫存區 (String Buffer) 和編輯指標 (Cursor)

編輯指令的工作主要是將字串輸入到現用塊中，或修改塊中的文字。大多數的編輯指令都是處理字串（String）用的。在Apple-FORTH的編輯指令中，經常將字串存在一個特別的字串暫存區（String Buffer）中。這個暫存區的開始地址是用 PAD 這個指令叫出來的。

在做編輯工作時，我們要輸入、插入、刪除、尋找的字串，都是暫時存放在 PAD 開始的暫存區內，這對編輯的工作有很多的便利。因爲

PAD中存的文字可以重複使用而可以節省許多次按鍵。

編輯指標（Cursor）是用來指示目前編輯工作進行到的字元的位置，它的值在0到511之間，存在一個系統變數R#之中。編輯指標隨時記錄了我們編輯工作進行到的行數和字數。許多編輯指令都是用這個指標來進行下一階段的編輯工作。

3-1-2 行編輯指令

若要改變整行的文字資料（32個字元），我們有下列幾個主要指令：TPU Tab與X。選定某一行做爲編輯的對象（或現用行），是要按入以下的指令：

```
n  T
```

n是現用行的行數（由0到15）。T將此行首尾對齊之後再將此現用行再列出來一次，這次列印時，編輯指標的位置即用一個帽號（∧）插本行文字中頂頭的位置。這個指標的地址也存到R#中去。

在用LIST選定一個塊開始編輯時，必須先用nT的指令來選定要編輯的那一行。在進行編輯中只有在需要重新將指標設定到另一行的開始位置時才須要再用T指令。若要將指標設定到塊的開始或是第0行的開頭，我們有另外一個指令：

```
TOP
```

TOP相當於0T，只是TOP並不將第0行印出來。

以下三個指令是改變整行文字的指令：

P 將讀後的文字串放到PRD暫存區，然後再將此文字串放在現在指標所在之行中。此行內原有的文字即消失。

U 將後讀的文字串暫存在 PAD 中，然後將它塞在現用行之下一行。現用行原來下面各行都向下移動一行，最後第 15 行即被刪除。

X 將現用行由塊中刪除，其下每一行都向上移動一行。最後的第 15 行就用空格（SP）裝滿。刪除的一行文字存在 PAD 暫存區內。

大部份的編輯工作，都可以用這幾個行編輯指令來完成。所以在繼續研讀以下字串編輯指令之前，學者應該先將這幾個行編輯指令練習純熟。

3-1-3 字串編輯 (String Editor)

在修改一行文字中的一小段時，不必動用行編輯指令輸入整行的文字。而用以下的幾個字串編輯指令。

F D TILL I

可以更有效地尋找加入或刪除一段文字或字串。這些字串指令和編輯指標的位置有密切的關係，所以我們應學習如果準確地控制指標以標定現用行數及現用指標的位置。一般的做法是先用 T 指令選定現用行並將指標移到此行的開始位置，然後再用 F 指令將指標移到我們所預期的位置。因爲這些字串編輯指令使用起來不是很容易，在開始做練習時，最好是找一塊不重要的磁碟區來實驗。

F 由指標現在的位置開始往下尋找後續的字串。如果找到了這個字串，印出字串所在的整行文字，並將指標移到此字串之後。如果在此塊中找不到這個字串，則印出一個錯誤的信號，並將指標復置在本塊

文字之首。

D　在指標後之文字中尋找後續的字串並將其刪除，指標置於刪除字串之後，如果本塊文字中設有此字串，則印出一個錯誤訊號，並將指標復置在本塊文字之首。

TILL　刪除在本行中的文字，由現用指標處直到後續的字串並包括此字串。刪除後空缺由行後面補足空格。

I　將後續的字串插在現用指標之後，並將指標移到此字串之後。

這些字串指令配合了上節的行編輯指令就足夠做各種編輯的工作了，但也需要多加練習才能熟練。

有一點我們需要特別注意的是，當我們用P、U、F、D、TILL及I等指令時，其後的文字串會存到PAD中。如果一個字串要反復使用多次時，在字串指令（P、U等）之後馬上按Return鍵，則系統會去用PAD內的字串來做編輯的工作。這是所謂的無字字串（Null String）的處理工作。例如我們要將第七行文字搬到第三行以下（第四行），則可用下例指令順序：

7 T　將指標放在第7行頭上。

X　刪除第7行文字，並存於PAD中。

3 T　將指標移到第3行頭。

U　將PAD中文字塞在第3行之下。

在表3-1中我們已將這些編輯指令表列出來供讀者參考。另外在表3-2中列有一些用編輯來寫FORTH程式需要注意的事項。這是許多有經驗的FORTH程式師歸納出來的程式寫作法，這樣寫的程式將會很容

表3-2 編輯規則

1	第零行必需是一個有括弧的注解，以便用INDEX列印目錄。
2	每三塊組成一頁，起始塊數須能被3整除，以便用TRIAD列印。
3	每塊中只包括功能有關且在相同層次上的指令。 不相關的指令不可寫在同一塊中。
4	每塊中文字不可太擁擠。預留數行以便擴充。
5	每行中不可有一個以上的定義，除非是一些有關的常數或變數。
6	每個定義的名稱後應留三個空格，與指令分開。
7	將定義中的指令分割成句，每句以兩個空格分開。
8	若定義佔兩行文字以上，第二行以後行首要縮進三格以上。
9	在CODE定義中，每個機器指令句要用三個空格分開。

易讀，也可以作爲程式的檔案記錄（ Documentation ），爲以後修改增刪的工作減少許多不必要的困擾。所以我們建議讀者在學習寫FORTH程式時就切實遵循這些規則，養成良好的寫作習慣。

3-1-4 其他有關的指令

左括弧“（ ”是一個重要的指令，它在程式中插入一段文字註解，說明程式本身不易表達而對使用者很重要的事項。“（ ”後的文字一直到右括弧爲止，在編譯時候都被略去，所以使用者可以隨意加入註解文字，例如

```
SYSTEM  DEFINITIONS
```

是放在系統程式塊中的第0行，標明此塊內程式的功能。適時適地地使用註解，是做程式檔案的不二法門。

開機之後 Apple FORTH 的磁碟記憶區中充滿着一些雜亂的資料

。要將某一塊的資料清除，以便重新撰寫程式，可以用CLEAR的指令

```
10 CLEAR
```

會將第 10 塊中所有的字元都清除爲NUL（ASCII 0）。如此清除後，文字必須由第0行開始依序一行一行輸入。因爲FORTH系統在作編譯工作時，碰到ASCII NUL時立即停止編譯。所以若隔了一空行插入的文字，在空行後即不被編譯。

如果一塊資料不是用CLEAR來清除的時候，最後的一行必定要包括一個指令EXIT 來停止編譯。EXIT的效用和ASCII NUL相同。

當我們在試驗一塊中的指令時，往往需要將此塊資料另存一處以作參考，塊間資料重錄的指令是COPY：

```
3 12 COPY
```

將第3塊中的文字全部抄錄到第 12 塊中。

要將在記憶塊或假磁碟中的資料存到錄音帶上去的時候，應先按入MON 指令跳到Apple的監督程式中，在星號出現之後按入錄音的指令：

```
＊6000.BFFFW
```

這是將假磁碟中全部24K字元的資料都錄在磁帶上的方法。如果實際使用的磁碟沒有這麼多，可以酌減記錄的上限。

將存在磁帶上的資料讀入假磁碟中也是先要用 MON 指令跳入監督程式，然後按入讀取資料的指令

```
＊6000.BFFFR
```

錄音機操作的要點可以參考Apple的使用手册。

3-2 FORTH程式的測試

FORTH是一種交談式的語言，電腦通常都是立即執行你交給它的指令並將結果交還給你。因此在發展程式時使用者大部份的時間是花在終端機前做輸入及測試的工作，而不要費很多時間在辦公桌上寫程式。通常使用者是寫一些有關程式解法的流程與摘要，或許有幾行實際的程式。假如問題較大時，就可能需要將問題解剖成爲較小的單元，分別地設法解決。然後，就要坐到終端機前面，開始建造一些新的指令，這些指令必須經過很詳密的測試，說明它們完全能達到設計的標準。然後，這些基本的指令可以用來建造更高一層功能更廣的指令。這樣一層一層地堆砌起來，往往到最後的一個指令便成爲整個問題的解了。

所以在FORTH中，我們並不寫作像一般語言中所謂的程式，而是創造一堆的新指令，這堆指令的適宜結合便解決了整個的程式問題。

當你定義好了一個新指令之後，應該立即執行這個指令並按入"."這個指令，看看在疊層上的數值是不是你所預期的結果。一個指令若在疊層上意外地留下一些數值時，會使得整個系統操作變得不穩定，可能在完全不相干的地方產生無法預知的結果。FORTH中這種程式的錯誤佔了絕大多數。

假若你的一個指令所得的結果不對，而在仔細核對程式原文也看不出什麼破綻時，最穩當的除錯（Debugging）法是把這個定義中個別的指令按着順序一個一個執行，並隨時檢查每一個指令的結果，直到找出那一個產生問題的指令來爲止。在測試過程中，你當然要把每一個指令需要的參數先放在疊層上來模仿待測定義中疊層的變化。另外也要注意DO-LOOP，BEGIN-UNTIL及IF-ELSE-THEN等結構也須要模做，因爲這些結構是無法直接執行的。

千萬不要將許多未經測試的指令堆砌起來疊了許多疊之後再來測試

。若有一處以上的錯誤同時發生時，就很不容易確定錯誤在什麼地方了。測試一定要由最低層的指令開始，只有在將低層的指令完全測試，證實無誤之後才去用它們來建造較高層的新指令。FORTH的單元化結構使得程式的測試十分容易，只要你能遵循這個由低而高的測試方法。

當你在測試了一大堆的指令之後，詞典裏一定會堆積了許多有用無用的指令。如果你決定要重新來過而將詞典中你所加進去的指令都清除掉，有一個很好的指令做這件事：

```
COLD
```

它將你的詞典清除乾淨，將FORTH系統回到開機時的狀態。如果你測試的某個指令是一個無限循環或執行的時間過長而你想要終止這個指令的執行，你可以同時按Reset鍵，放開Reset鍵後即返回FORTH系統。這與COLD不同之處是它不清除你的詞典，所以你可以繼續原先的工作，這個重置（Reset）的過程稱爲WARM START。

我們也可以有選擇性地清除詞典的一部份。如果我們要除去在FORTH詞彙中的一些指令，例如是TEST以後所有的指令，就應該這樣做：

```
FORTH DEFINITIONS      ;將FORTH設爲詞意及現用
                       ; 詞彙。
FORTH TEST             ;將詞典清除到TEST。
```

在測試指令的時候，我們必須由最低層的指令先測試起，然後慢慢進到較高層的指令。但如果我們在磁碟上發展整套的系統，却是可以由高層向低層來寫。這即所謂"上而下設計"法（Top-down Design）。如果要先定義一大堆的低層指令我們猜想以後可能會有用處的，然後慢慢將它們組合起來，這種方法一定在時間及精力上多所浪費。上而下

的設計可以避免這些浪費而得到一個最經濟最有效的系統。

例如我們想到要設計一個電腦控制的照相機，當我們給這個電腦以下的指令時：

```
5 PHOTOS
```

它就自動地拍攝五張相片。PHOTOS 的定義中必定有一個DO-LOOP的循環結構：

```
: PHOTOS 0 DO PHOTOGRAPH LOOP ;
```

然後我們再來考慮如何定義 PHOTOGRAPH 來拍攝一張相片。再往下，我們就要定義一些單獨的指令來做打開快門，延遲一段時間，再將快門關閉等簡單的工作。

眞正發展程式的過程應該就是如此，不管程式是用來解什麼問題。在此我們要特別强調兩點：

(1) 發展及寫作程式是應當由上而下的，雖然在編譯及測試時應該反向進行。

(2) 獨立的工作都應該定義成爲獨立的指令，FORTH的單元化構造允許並鼓勵使用者把問題分解而簡化。每個定義也可以簡化，以利測試。

這些特性不但幫助我們更快更容易地發展程式，而且更使許多低層的指令易於修改，組合成功能稍稍不同的指令，而更容易適應不同的使用環境。

3-3 FORTH指令的編譯

如果程式是寫在假磁碟的記憶塊中的話，在測試之前一定要將這些記憶塊裏的指令編譯到詞典上面去。若要編譯第六塊中的指令時，應按

6 LOAD

的指令，第六塊中的指令卽被依序執行。若有一些新指令的定義，便會被編譯而加入詞典。

很少有一些應用程式可以在一個記憶塊中寫完。在Apple-FORTH中的塊又小（512字元）更是困難。一個完整的系統是分散在許多不同的塊中。要按許多的LOAD指令將這些塊編譯起來是很不方便的事，但我們可以選用一塊把所有的LOAD指令都放在這一塊中。當這一個"編譯塊"被編譯的時候就把所有需要的塊都編譯起來了。

當一塊程式中要編譯另外一塊時，有三個數字被堆上了返回疊層：現用的塊數、現在編譯到的指令位置、以及返回的地址（因爲LOAD是一個高階指令）。爲了保護返回疊層不致超載，最好不要用連鎖編譯的方法，每個塊自行呼叫下一塊。不用連鎖編譯的另外一個好處是程式的管理也比較方便，因爲"編譯塊"是一個很好的程式索引。例如每個LOAD指令前都加一點註解：

```
(DATA I/O) 20 LOAD
```

便可很容易的知道第 20 塊中的工作功能。

如果一個系統中有幾個編譯塊用來做不同的工作時，我們可以把編譯塊的序數定爲常數：

```
20 CONSTANT I/O
```

以後我們需要做輸入及輸出的工作時，就不必記得編譯塊的序數是 20，而可以更明白地命令電腦：

```
I/O  LOAD
```

將編譯塊定一個適當的名稱．以便取用是很有趣的技術，可以讓我們在一個系統中換用不同的應用程式系統，一般稱之爲多重程式（Overlays）。

普通在多重程式中由一個系統換到另一個系統時，先將上一系統所添加的指令清除掉。我們可以用COLD的指令，來將使用者加入的全部指令都清除乾淨。有時我們需要保留一部份的指令就要用 FORGET 的指令了。用 FORGET 之前，詞典中一定要有特殊的指令做爲標誌，以備 FORGET 來清除其後續的指令，這個特殊的指令實際上不做任何工作，而是當做詞典中的一個標籤：

```
: OVERLAY ;
```

然後我們在每一個多重程式系統的編譯塊的最前面加入以下的指令：

```
(LABELS) FORGET OVERLAY  : OVERLAY ;
```

這樣的話所有的多重程式都會在詞典的相同位置開始各自的編譯工作。

小心地使用COLD及如上所述地使用 FORGET 指令還有一個好處，它們能夠把程式規則簡化而方便最後使用這些程式的用戶。我們在不同的多重程式中可以用相同名字的指令做類似的工作。這對使用的人來說是方便的，雖然我們寫程式的人知道這些指令是不相同的，而且是在不同的多重程式中。用上面舉的兩個例子，I/O 與 LABELS 是兩個多重程式，它們可能都有一個指令名叫 ENTER 由鍵盤接受不同形式的資料。它們接受資料的型式不同，但對最後的使用者來說，他只要記得 ENTER 一個指令，操作起來也容易得多。

3-4 列印原程式

將原程式列印在終端機上是在編輯過程中佔很重要的，可讓使用者詳細檢視程式原文，以便修改、測試。列印原程式的基本指令是 LIST，將某一塊的資料以 16 行每行 32 字的方式列印在終端螢幕上。標準的FORTH系統列印的方式是每行 64 個字，每行的程式可以寫得較長，但是Apple的螢幕每行最多 40 字，所以 Apple-FORTH 採用了較小的塊及較短的行。較小的塊及較短的行迫使用者寫短小的指令，對FORTH的單元化的特性也有加強的作用。

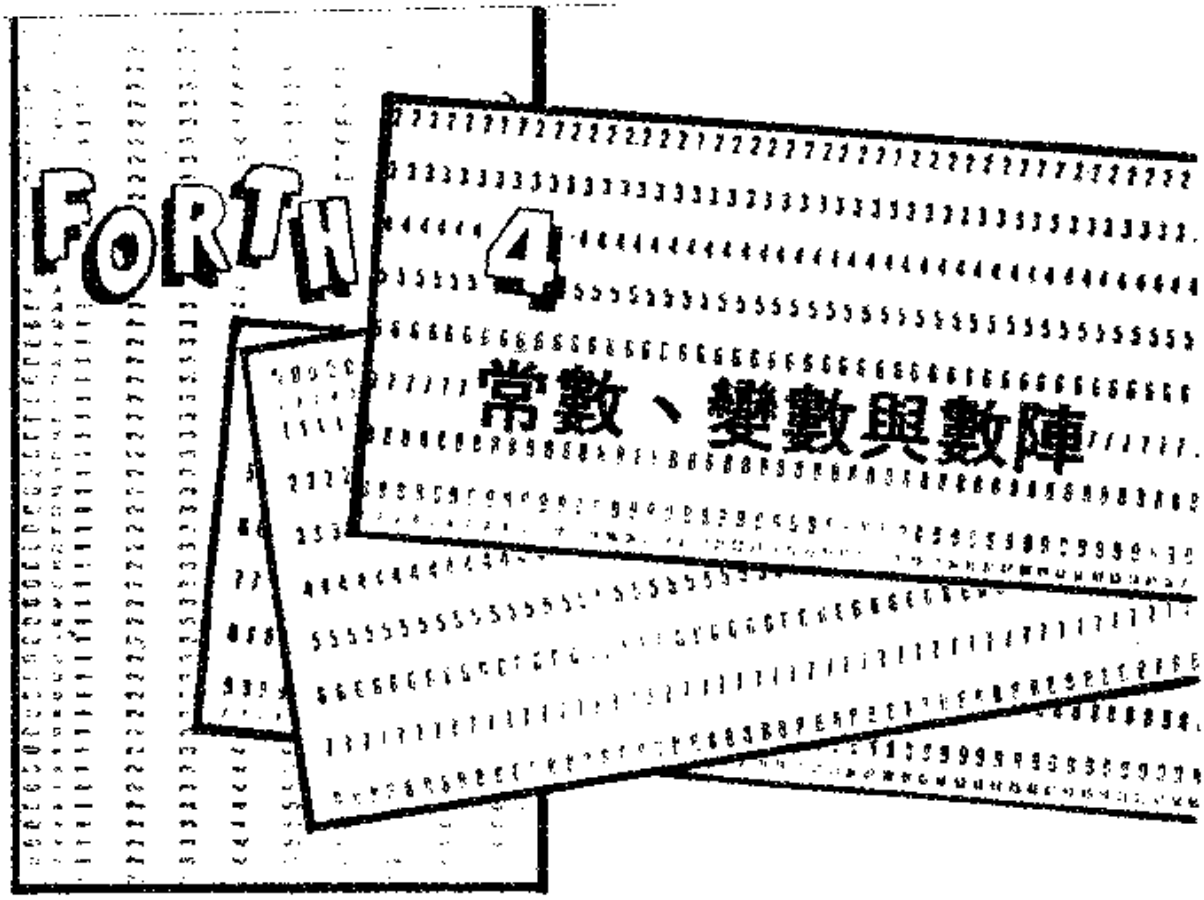
LIST 指令除了將一個塊中的原程式文字列印出來，還將EDITOR定義詞意詞彙，使用者立刻可使用各個編輯指令來修改程式文字。

主要的幾個列印指令的功能略述如下：

n LIST	將第 n 塊中的原程式文字列印在終端機上。
n TRIAD	連續列印三塊原程式，其中包括第 n 塊，而第一塊的序數須能被 3 整除。
n m INDEX	將第 n 至 m 塊間每一塊的第 0 行文字列印在終端機上。因此，每塊程式的第 0 行通常應該用一個註解來說明本塊內容。
VUST	將語意詞彙及FORTH詞彙中所有指令的名稱按尋找的順序印出。按任何一鍵均可停止印出的動作而返回FORTH系統。

練習題三

1. 將以前的練習題解法都儲存到磁碟記憶中，練習使用各個編輯指令。
2. 將這些練習的原程式及其索引列印出來。



FORTH使用數據疊層來暫時存放數據資料，以便在指令之間轉遞與運算。但如果數據資料要儲存較久的時候，數據疊層就不方便了。比較長久的資料，要用常數CONSTANT，變數VARIABLE等方式來儲存。常數及變數在FORTH中使用得不多，因爲它們佔用了詞典中的位置。表4-1中列出有關數據存取的指令。

表4-1　記憶指令

指　令	疊層動作說明	中文指令
@	(a — n) 將地址 a 中的數值 n 放在疊層上。	取
!	(na —) 將數值 n 存在地址 a 中。	存

C @	(a — b) 將地址 a 中的字元 b 放在疊層上。	取字
C !	(ba —) 將字元 b 存入地址 a 。	存字
2 @	(a — d) 將地址 a 中的雙整數 d 放在疊層上。	取雙
2 !	(da —) 將雙整數 d 存在地址 a 的記憶中。	存雙
+ !	(na —) 將 a 處所存的單整數加 n 。	加存

4-1 常 數

假如一個數值是常常要用的，而且又與一個特殊的功能有密切關係，這個數值可以定義爲一個有名稱的指令。如果名稱選擇得合宜，這個數值指令將比數值本身容易正確的輸入或使用。這種指令稱爲常數，常數指令是用定義指令CONSTANT來建進詞典中。例如將噸換算成磅的換算因數，可以定義爲--常數稱之爲 TONS：

```
2000  CONSTANT  TONS
```

即在詞典中加入一個常數指令TONS；且將其值設定爲2000 。此後，我們使用TONS的時候，TONS即將2000這個數值堆上疊層，以下的指令句：

```
TONS  3  *
```

即可算出三噸重的總磅數。一旦一個常數定義好了之後，它的值即不再

跟着現用的基數而改變。在經常改變基數的程式裏，使用常數即可避免基數轉換的問題。

4-2 變　數

一個數據資料若在程式中經常會被改變者，稱爲變數。在FORTH系統中VARIABLE指令即可在詞典中建造一個變數新指令，這個指令在執行的時候將一個變數在記憶中的地址放在疊層上。例如我們要做一台機器自動計算進入一家百貨公司的客人數目，在FORTH即可定義：

```
VARIABLE  PATRONS
```

即在詞典中加入一個新指令 PATRONS（客人的意思）。這個指令執行後即將客人數目的儲存地址放在疊層上。在開始計數之前，我們必須將此數值歸零：

```
0  PATRONS  !
```

指令！（存）是將疊層上的值0存到 PATRONS 的地址中去。用這種方法可以將任何十六位元的數值存入 PATRONS 中。

有一個和"！"相關的指令是 +!（加存）：

```
101  PATRONS  +!
```

即將PATRONS中現存的數值加上101再存回去。原來PATRONS中的值若是零的話，現在就是101了。

指令@（取）是將疊層頂上的地址取出，找出這個地址中所存的數值而將之放回疊層上。因此要從地址PATRONS中將客人的總數取出來，放在疊層上的指令是：

PATRONS @

指令？則將一個變數中所存的數值印出來：

PATRONS ? 101 OK

如果要將一個變數抄錄到另外一個變數中，例如將 OLDPATRONS 的值轉錄到 PATRONS 中去的指令句是：

OLDPATRONS @ PATRONS !

請試看自己定義一些常數及變數並試看用@，？，！，及＋！等指令來改變及檢查它們。確定每一個指令的用法及其特性。

在一個用 RAM 記憶的 FORTH 系統中，常數和變數的差別並不大，主要是看使用的方便。常數是用來表示很少改變的數值，使用時可以直接叫到疊層上。變數的值會常常改變，所以叫到疊層上的不是變數的值而是它的地址，其中的資料隨要另外用@及！等指令來存或取。如果 FORTH 系統最終目的是要放在 ROM 記憶中時，常數和變數的差別可就大了。因爲常數可以建在詞典中一同燒錄在PROM中，其數值就不可再改變了。在PROM中的變數燒錄進去的是一個在 RAM 中的地址，所以在使用時，RAM 中的數值可以改變，但要注意其初值必須在開機的時候設定才行。

在 Apple-FORTH 系統中的常數如 FPRST、LIMIT、C/L等都是燒在 ROM 中所以不能改動。許多變數如 BASE、CONTEXT、CURRENT 等都指着在系統變數區（User Area）中的一些地址，它們的值可以隨時設定、改變。使用者用 CONSTANT及 VARIABLE 建造的新常數及變數都是存在 RAM 中的，所以它們的值都可以改變。

改變在 RAM 中的常數值要用到一個特別的 FORTH 指令 " ' " "

，"接受了它後面的字串而到詞典中去查有沒有這個字串代表的指令，如果找到了這個指令"，"即會將此指令的參數欄的地址堆上疊層。如果這個指令是變數或常數，數值資料就是存在這個參數欄中的。如果我們要將方才定義的TONS中的數值改變以代表長噸（2240磅）的換算因素時：

```
2240 ' TONS !
```

我們先將2240放在疊層上，'TONS將TONS的參數欄地址堆上疊層，而指令！將2240存了進去。以後我們用TONS指令的時候，它即將2240放在疊層上供作計算之用。

一個很有趣的現象是在FORTH程式中常數及變數的使用頻率相當小。因爲大部份的數值都是經由疊層來傳送的，所以不必另外定義一些常數及變數來儲存這些數據資料。用疊層來傳送資料節省記憶的空間並加快程式的運算。所以除非是系統中的基本參數要用常數及變數方式來定義，其他的資料儘可能用疊層來處理。

初學FORTH的人常常會定義一大堆的常數和變數，這是由其他常用的電腦語言中遺留下來的習慣。要充分利用FORTH的特性，我們必須熟悉疊層的使用法而儘可能避免用常數和變數。在FORTH中常數及變數都是所謂的大範圍變數（Global Vaviables），可以在任何指令中使用，這些變數容易混用及誤用，這樣產生的錯誤很不容易偵測及改正。用常數及變數來傳送資料是不符合結構化程式法（Structured Programming）原則的，因此使用時須小心。

4-3　双常數與双變數

許多FORTH系統中有2CONSTANT指令可以定義雙整數（32位元）的常數，及2VARIABLE指令可以定義雙整數的變數。在

Apple-FORTH 系統中雖然沒有這兩個指令，我們可以很方便地加進去：

```
: 2CONSTANT CONSTANT , DOES> 2@ ;
: 2VARIABLE VARIABLE 2 ALLOT ;
```

它們的意義我們將在第六章中詳細討論。

若有了以上的兩個定義指令，我們要定義一個雙整數的常數或雙常數時就可以用下面的指令句：

```
d 2CONSTANT COMMENT
```

d是一個雙整數。我們在用 COMMENT 的時候，它就將雙整數 d 堆到疊層的頂上。我們也可以用兩個單整數組合成一個雙整數來定義雙常數：

```
n1 n2 2CONSTANT COMMENT
```

以後我們按入COMMENT 後，n1及n2就被堆上了疊層，n2 是雙整數的上半，所以在疊層最頂上。

另外舉兩個例子：

```
100.000 2CONSTANT BIG
```

使用BIG的時候，就產生一個雙整數：

```
BIG D. 100000 OK
0 5000 2CONSTANT RANGE
RANGE . . 5000 0 OK
```

雙常數也可以用 ' 指令來改變其數值，這和常數類似：

```
200.000 ' BIG 2
BIG D. 200000 OK
```

2！指令將一個雙整數存到一個記憶地址中去。

雙變數的用法和變數一樣，會將一個地址留在疊層上，只是雙變數的參數欄可以容納兩個數元或 32 位元的資料。例如我們可以定義 SHEEP 爲一個雙變數：

```
2VARIABLE SHEEP
450.000 SHEEP 2!
```

使用雙整數的2@（雙取）指令即可將 SHEEP 中的雙整數取出堆上疊層：

```
SHEEP 2@ D. 450000 OK
```

2@及2！處理一對單整數組成的雙整數，上半數是在疊層頂上，在記憶中上半數是在低記憶位址中。這樣雙整數的上下次序在疊層上或是在詞典中是一致的。

如果我們定義一個雙變數 LIMITS

```
2VARIABLE LIMITS
```

而用來儲存兩個單整數：

```
50 1000 LIMITS 2!
```

其中 1000 是雙整數的上半而500是其下半。我們現在可以用下面的指令句：

```
LIMITS 2@
```

取得兩個單整數，這時 1000仍然是在 50 之上，和將它們編譯時的情形完全相同。再如以下指令句：

```
LIMITS  2+  @
```

將 LIMITS 的參數欄地址加2後留在疊層上，則我們取得 LIMITS 的下半部的單整數50 。因爲在Apple-FORTH中記憶內容是以字元 byte 爲單位的，一個單整數是兩個字元。

4-4 數陣 (Arrays)

既然VARIABLE定義的變數在執行的時候是將它的參數欄地址留在疊層上，我們可以將參數欄更加擴大而用來容納許多數元或字元，成爲數陣(Arrays)的構造。假定我們要建造一個數陣其中有 20 個字元的數據：

```
VARIABLE  VALUES  18  ALLOT
VALUES  20  ERASE
```

18 ALLOT在參數欄已預留的2字元外增加18字元的容量。VALUES 20 ERASE則將VALUES參數欄中20字元統統清除爲零。

要由這個數陣中索取資料可以用以下的指令句：

```
VALUES  @          取得第一個數元的值
VALUES  2+  @      加2取得第二個數元的值
...                ...以此類推
```

數陣也可以用 CREATE 指令來建造。CREATE 造成的數陣和 VARIABLE 造成的相同，使用時也是將參數欄的地址送回來。其不同處是VARIABLE 僅預留兩個字元準備儲存一個數元，CREATE 則不

預留任何空間。用 CREATE 來建造數陣時，可以用指令來儲存資料：

```
CREATE TENS 1 , 10 , 100 , 1000 ,
```

這就建好了一個表，開始的地址是用TENS取得的，其中所含的數值是 10 的四個冪數。它的用法是

```
TENS ? 1 OK
TENS 2+ ? 10 OK
```

TENS給了這個數陣的開始地址，其中的數值可以依序取得。

TENS眞正的目的是在計算 10 的冪次值：

```
: **10 2* TENS + @ U* ;
```

有了這個指令之後我們可以計算：

```
25 4 **10 D. 250000 OK
```

即是計算 25×10^4 之值，最後以雙整數形式列印出來。指令中 2* 的句子是因爲地址是以字元爲單位而資料是以數元的方式存在TENS數陣中的。

用VARIABLE 及用 CREATE 建造的數陣在RAM系統中沒有什麼差別，但是在ROM的FORTH系統中就大不相同了。因爲CREATE建造的數陣是貯存在詞典中，是不可以改變的，但VARIABLE建造的數陣還是會放在RAM中，操作的時候其中數值可以改變。第六章中我們還會再討論這種情形。

練習題四

1. 由A與B定義的 x 有何區別？

A. 100 0 2CONSTANT x

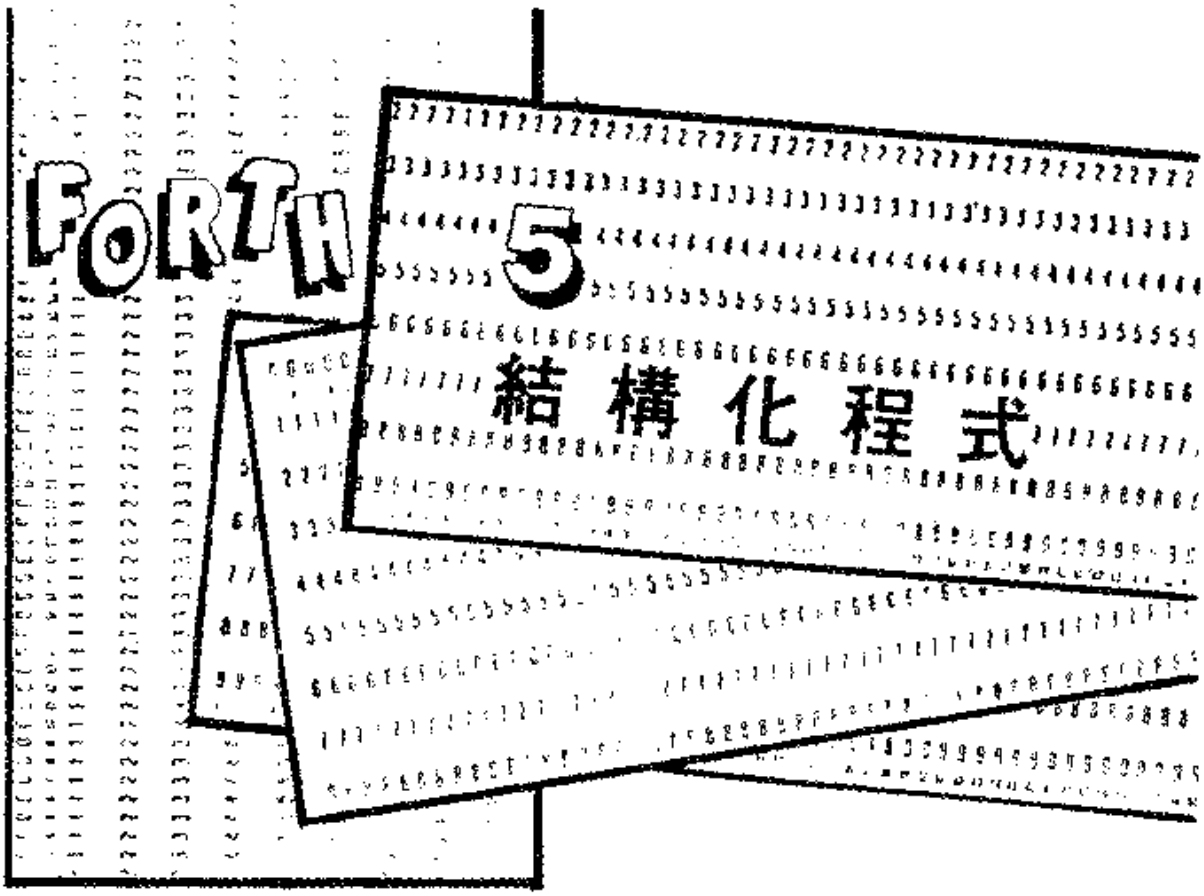
B. 100 CONSTANT x 0 ,

寫出一指令用以計算二次方程式之值：

$ax^2 + bx + c$

x 在堆疊上。如果 a = 7 , b = 20, c = 5 , 試求最大的 x 值使此式之結果仍在 16 位元的範圍中？

3. 改寫上題的指令，使其方程式值爲 32 位元的雙整數。



結構化的程式是指在程式中的邏輯流程（ Logical flow ）必須依照下面三種方式之一來進行：

(1) 連續步驟：即是一步接一步地進行操作。一般高階指令即是如此定義的。如圖5-1所示。

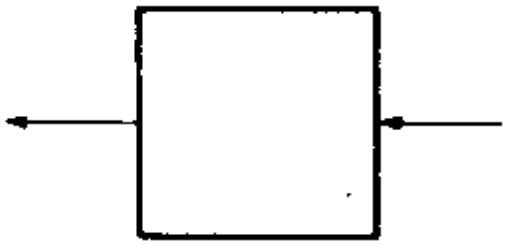
(2) 條件分支（ Conditional Branch ）：假如某一條件是眞即進行A；否則進行B；A或B完畢之後繼續進行C。

(3) 循環：重覆A項步驟直到某項條件爲眞，然後繼續進行B。

FORTH允許我們用以上三種方法來建造結構化的程式。

5-1 條件分支 (Conditionals)

一個條件分支的結構使得電腦可以"做決定"。在FORTH裏，在條件分支的結構部份，電腦測驗疊層頂上之數值，並以此決定是否要改變執行的順序。條件分支的形式是：



結構單元

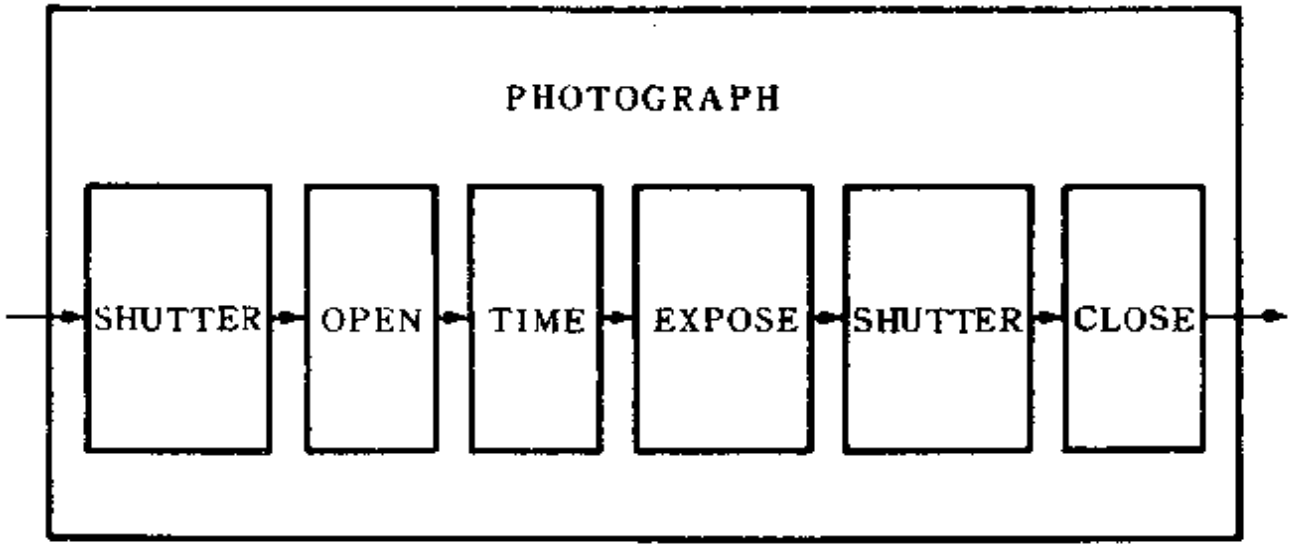


圖 5-1 簡單的高階指令結構

```
: DEFINITION Condition IF this ELSE that
    THEN Continue ;
```

其詳細的說明如下：

: DEFINITION	定義一個新指令。
Condition	將一個邏輯旗號（眞或假）放在疊層上。
IF	將旗號取下並試驗之。
this	執行這一段指令若旗號爲眞（非零值）。
ELSE	

```
    that            執行這一段指令若旗號爲假（零值）
THEN
Continue ;          繼續執行以下的指令。
```

IF，ELSE及THEN等指令必須用在高階指令的定義中。IF到THEN之間所有的指令構成一個結構（Structure）。IF會測試在疊層頂上的值，若此值不是零，則繼續執行IF與ELSE之間的所有指令，到ELSE之後則跳到THEN再執行THEN以後的指令。若疊層頂上的值是零，則跳過IF與ELSE間的指令而執行ELSE與THEN間的指令，再繼續THEN以後的指令。

IF 指令包括了一個0＝的測試指令，所以會將疊層最頂上的數值用掉一個。假若這個數值在IF…THEN間還要用，就必需在IF以前將其複錄保存起來。下面的例子是避免用零作爲除數的方法：

```
: /CHECK  DEMON @ DUP
     IF  NOMERATOR @ SWAP / RATIO !
     ELSE DROP
     THEN ;
```

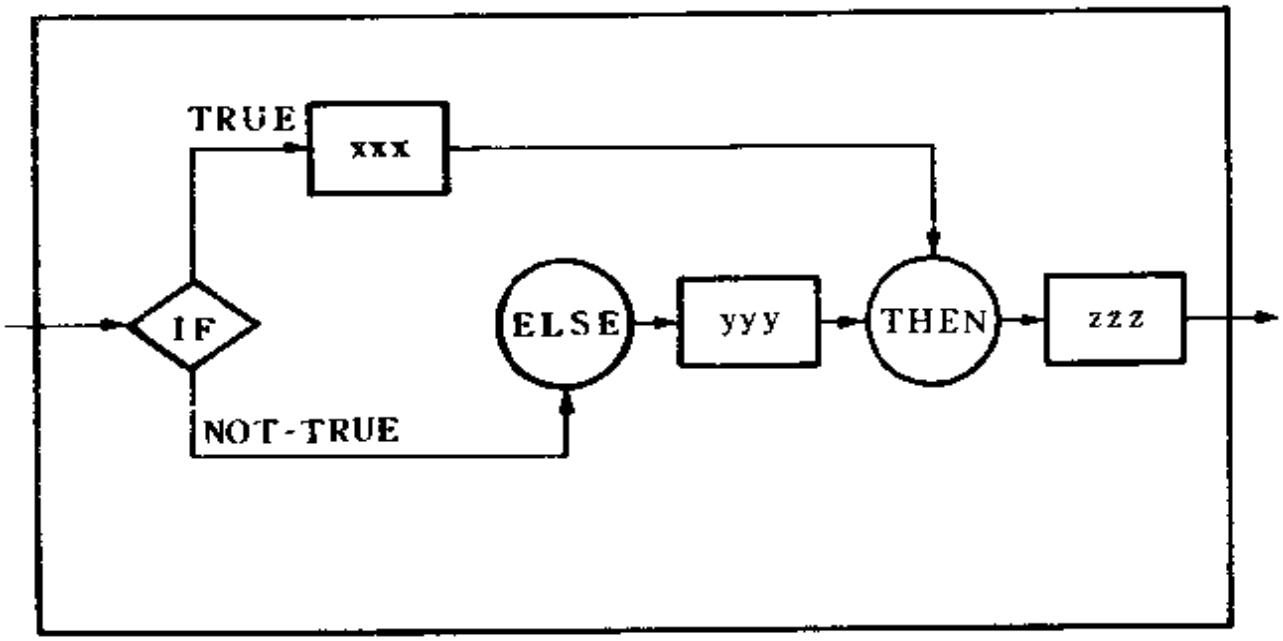
說明如下：

```
: /CHECK
   DEMON @              取出分母之值
                        （存在DEMON中）。
   DUP                  將分母複錄一次。
   IF
     NUMERATOR @        分母不爲零，將分子取出
     SWAP /             分子被分母除。
```

```
    RATIO !          除商存在RATIO中。
ELSE
    DROP             分母爲零時，將分母棄去，
THEN ;               不做除法。
```

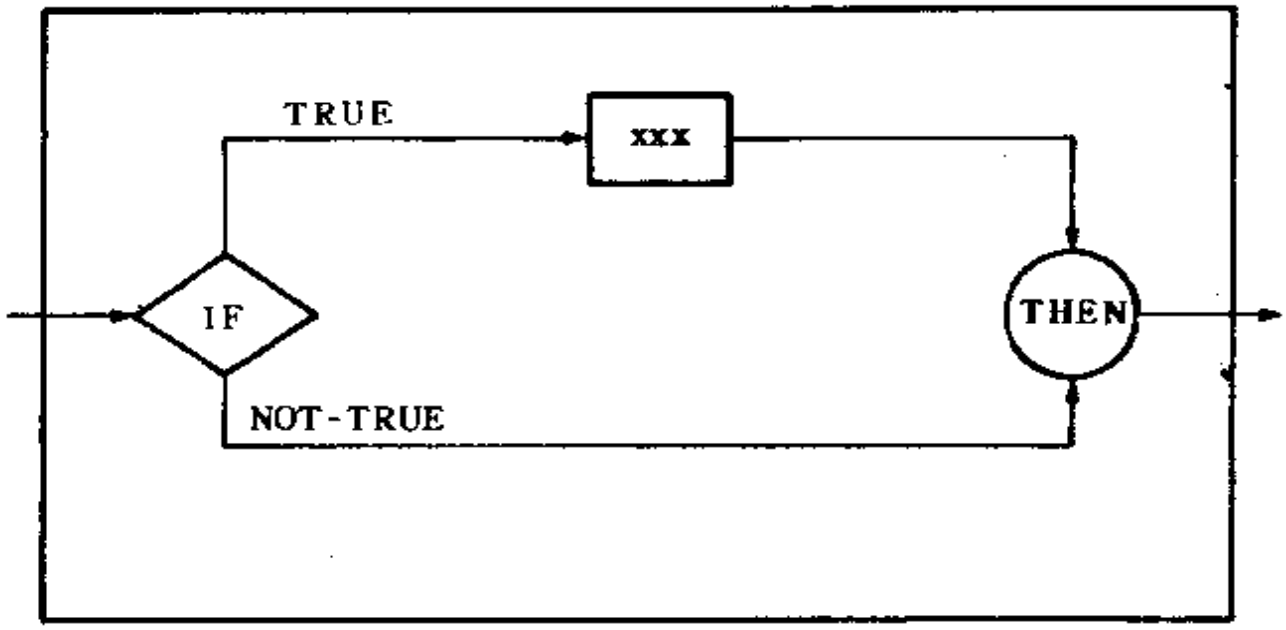
圖5-2是分支結構的兩個示意圖。在IF…ELSE…THEN的結構中，ELSE…一句可以沒有，這樣若IF測試的結果是假的時候，IF…THEN間的指令都不執行而跳到THEN以後再繼續。在上個例子中，如果我們用?DUP指令即可將ELSE句略去。因爲?DUP先試驗疊層上的值，只在其不爲零時才複錄，所以不須要在ELSE以後將重複的零值棄去：

```
: /CHECK DEMON @ ?DUP IF NUMERATOR @
  SWAP / THEN ;
```



(a) IF xxx ELSE yyy THEN zzz

圖5-2 分支結構



(b) IF xxx THEN

5-2　比較指令

FORTH中有三種比較指令：

(1) 測試疊層最頂上的一個數元的，如0＝，0＞及0＜。

(2) 測試疊層最上面兩個數元的，如＝，＞，＜等等。

(3) 測試疊層上 32 位元雙整數的指令，如D＜。

以上這些比較指令都會將其所用的數值由疊層上取走，再還置一個旗號回到疊層上。如果測試的結果爲眞，則留一個 1 在疊層上，否則留一個 0 。1即代表眞（非零），而零代表假。NOT 指令是將旗號反置，即是零變爲 1，而非零變爲零。例如要試驗不小於的條件，可用

```
0<  NOT
```

幾次試驗的結果可以用 AND，OR 及 XOR 等邏輯指令來合併處理。也可以把旗號當做數值而用算術指令＋，－，＊，/ 等來處理。

重要的比較指令都列在表5-1內，這些指令常常用在IF與UNTIL之前，給IF或UNTIL所需要的旗號以選擇下一步應該執行的指令順序。有時在用IF指令前也不需要特別做比較的工作，例如：

```
… 0= NOT IF …
```

的句中0= NOT是多餘的指令，不應該寫在定義裏。

若測試的結果是眞，比較指令一定是留一個1在疊層上做爲眞的旗

表5-1 比較指令

指令	疊層動作說明	中文指令
<	(n1 n2 — f) 若n1 < n2，f爲1否則爲零。	小於
=	(n1 n2 — f) 若n1 = n2，f爲1。	等於
>	(n1 n2 — f) 若n1 > n2，f爲1。	大於
0<	(n — f) 若n < 0，f爲1。	小於零
0=	(n — f) 若n = 0，f爲1。	等於零
0>	(n — f) 若n > 0，f爲1。	大於零
D<	(d1 d2 — f) 若d1 < d2，f爲1。	雙小
U<	(un1 un2 —) 若正整數un1 < un2，f爲1。	正小
NOT	(f1 — f2) 將疊層上的旗號值反轉。	反

號。這個旗號可以當做數值來做計算工作，例如加到一個數據資料上，用來選擇數陣中的某一個數值，做爲邏輯的罩子。方才說 0= NOT 在邏輯上可能是多餘的，但也有時這個句子可以用來將眞值（非零）變爲 1，以便計算處理。

減法指令"－"也可以用來做比較指令，因爲兩個相同的數值相減必定是零。如果兩數不相等，減的結果不爲零即是眞。但其眞值並不一定是 1。只有用 = NOT 才會給我們等於 1 的眞值。

5-3 循環（Loops）

循環有三種：

(1) 有限循環：一串指令重複執行一定的次數。

(2) 不定循環：一串指令重複執行，直到一定的條件爲眞而止。

(3) 永久循環：一串指令永遠重複執行。

FORTH 有一些重要的指令，可以在指令的定義中建造上述各種循環結構，以處理程式中需要重複執行的指令串。這些結構與 IF…ELSE…THEN 相似，只能定義在新指令之中而不能由鍵盤輸入執行。以下我們分別討論這三種循環的用法。

5-3-1 有限循環（Finite Loops）

有限循環依其循環指數增加的方式而有兩種：

(1) limit index DO words LOOP

執行在 DO 及 LOOP 中間的指令串，執行時循環指數由 index 增加到 limit，每循環一次指數加 1，到指數等於 limit 時終止循環。循環的指數及上限（limit）是暫存在返回疊層上的。I 指令將指數抄錄到數據疊層上

(2) limit index DO words incr +LOOP

重複指令串words，重複指標由index增加到超過limit，每循環一次指標增加 incr 。到指標等於或大於limit爲止。當 incr 有負值時，循環是反方向進行，即是limit必須小於指標(index)。

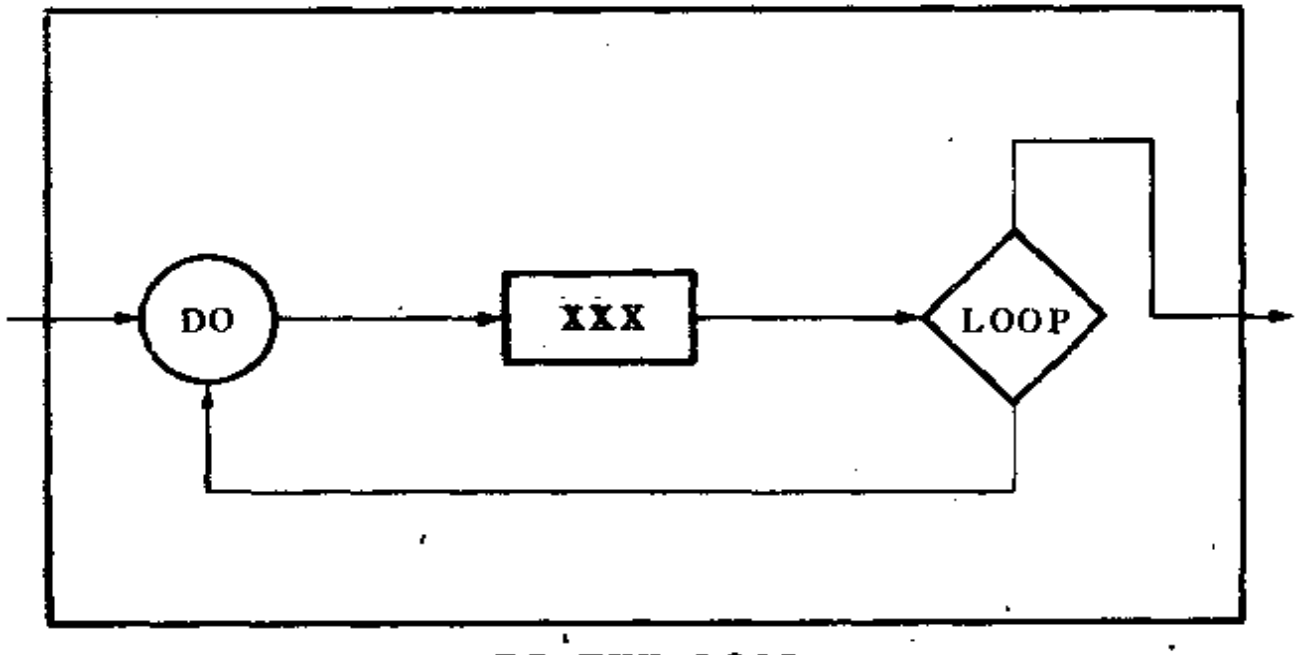
DO，LOOP，+LOOP等指令，必須用在定義內，不能在定義以外執行。在定義外執行，一般都會使系統失去控制而必須重置(Reset)或重新開機。

舉一個很簡單的例子。我們寫一個指令 COUNT-UP，使其印出由1到9的九個數目字：

```
: COUNT-UP  10  0  DO  I  .  LOOP  CR  ;
```

在DO前面的兩個數目是控制循環指數用的。0是指標的初值，10是指標的上限。在用LOOP時，指標必須小於上限，循環才能繼續。在用+LOOP時，在增值爲正時，指標應小於上限，但若增值爲負時，指標必須小於或等於上限之值。

循環執行到LOOP的時候，指標的現值加上增值後再與上限比較，



DO XXX LOOP

圖5-3 有限循環的結構

如果已經超過上限或等於上限時，循環的動作即終止而執行LOOP以後的指令。若指標還是在上限之內，則跳回 DO 之處而重複執行 DO 及 LOOP間的所有指令。圖 5-3顯示出循環結構中的流程。

在上例中，COUNT-UP印出的最高數目是 9，因爲印 9 的時候，指標是 9，到LOOP處指標加 1 變成 10，已經達到了上限，這時，循環立即終止。假如COUNT-UP 另行定義如下：

```
: COUNT-UP  10  0  DO  I  .  3  +LOOP  ;
```

循環到指標爲 9 時也停止，因爲在+LOOP時指標加 3 變成了 12，也超過了上限。

假如指標增值是負的，如：

```
…  -10  0  DO  I  .  -1  +LOOP  …
```

印出的數目將是 0，－ 1，－ 2，…，－ 10 。最後印出的值是－ 10，因爲此時指標必須超出 － 10 的範圍才會終止循環。

當執行到 DO 指令的時候，DO 將疊層上的兩個數值搬到返回疊層上去。上限是在指標的下面，指標是在返回疊層的頂上。所以在 DO … LOOP中，用 I 的指令可以將返回疊層頂上的指標再抄錄到數據疊層上來。因爲 DO - LOOP 使用返回疊層，所以在 DO - LOOP 中我們不能擾亂返回疊層，不然會使系統失去控制。返回疊層上的數值只能由 DO，LOOP等指令來改變。

表5-2列出在定義中建造結構的指令。表5-3中列出的是控制返回疊層的指令。I，J 兩指令是必須用在 DO - LOOP 中間，以抄錄在返回疊層上循環指標的數值。>R，R>及R@則讓我們自由地使用返回疊層來暫存一些數據資料。但是在 DO - LOOP 之間使用>R 及R>需要特別小心，因爲循環的指標及下限也在返回疊層上，並且有一定的順

表5-2 結構指令

指　令	疊層動作說明	中文指令
IF XXX ELSE YYY THEN ZZZ	IF：(f —) 若f不爲零，執行XXX，否則執行YYY。然後執行ZZZ。ELSE YYY可以不用。	若 否 則
DO XXX LOOP	DO：(n1 n2 —) LOOP：(—) 設定一個循環結構，指標由n2變到n1—1。	起 合
DO XXX +LOOP	DO：(n1 n2 —) +LOOP：(n3 —) 與DO……LOOP相同。在+LOOP時用疊層上n3爲指標的增值。	起 加合
LEAVE	(—) 將循環上限設等於指標，在LOOP或+LOOP時終止循環。	離
BEGIN XXX UNTIL	UNTIL：(f —) 設定一個不定循環的結構。UNTIL時若疊層上旗號爲零時則繼續循環。	始……終
BEGIN XXX WHILE YYY REPEAT	WHILE：(f —) 設定一個不定循環，執行WHILE時若疊層上的旗號爲零時，就跳到REPEAT以後終止循環。否則執行YYY後再循環。	起……出……返
BEGIN XXX AGAIN	設定一個永久循環。	起……循環

字。用了>R後一定要在R>之後才能使用I，J等循環指令。

以下是兩個十分重要的規則：

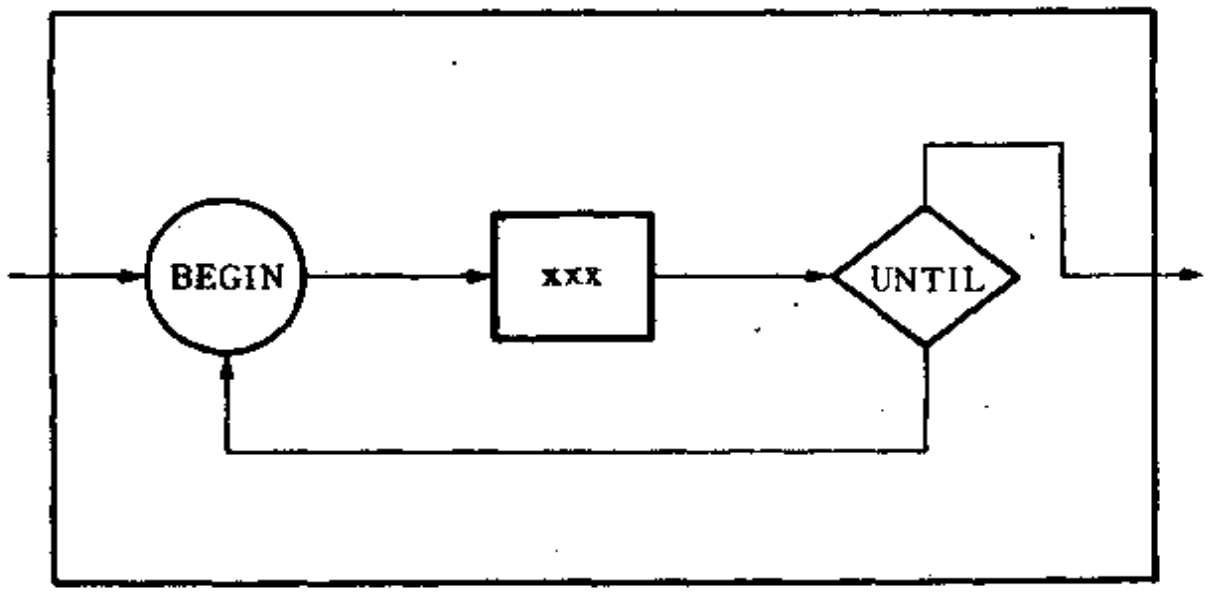
(1) DO後面必須有LOOP或+LOOP；DO-LOOP只能用在相同的一個定義中。

(2) 在DO…LOOP之間的指令串，不能改變疊層的高度，即是疊層

不應增長或減少。例外情形很少而以必須儘量避免。因爲叠層可能會飽和或不足。

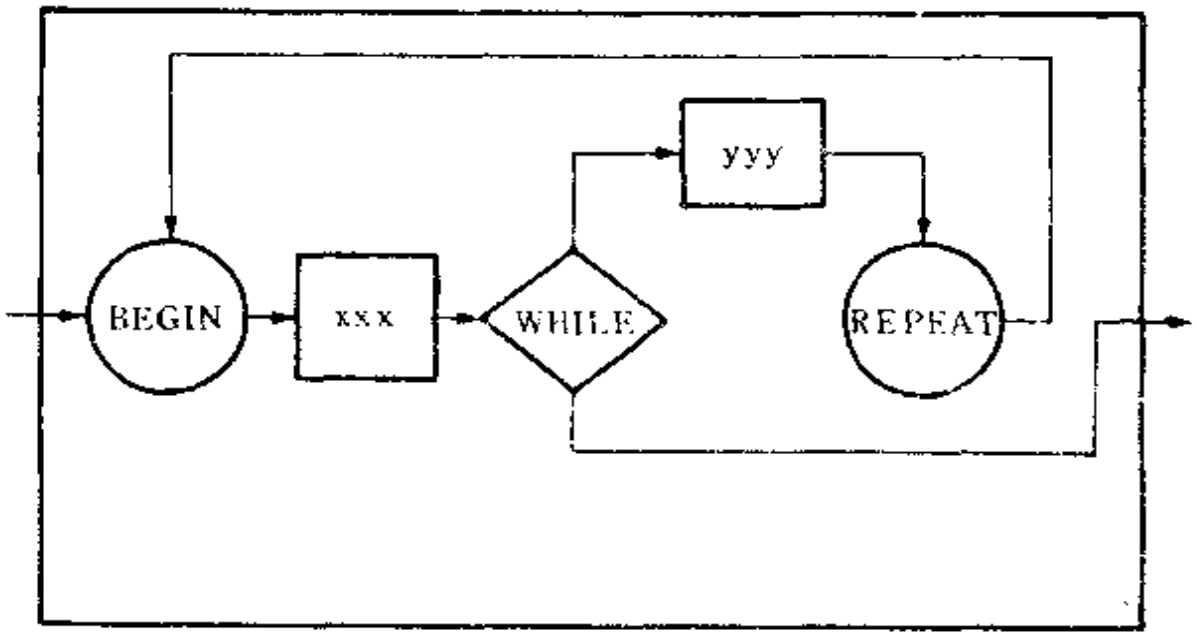
表5-3　返回叠層指令

指令	叠層動作說明	中文指令
>R	(n —) 將叠層頂值除去，暫存於返回叠層頂上。	去
R>	(— n) 將返回叠層頂值搬回數據叠層	來
R@	(— n) 將返回叠層的頂值抄錄到叠層頂上。	複
I	(— n) 與R@相同，用在DO-LOOP中將循環指標放在叠層上。	指標
J	(— n) 用在套裝的DO-LOOP中，將外層循環的指標抄錄到叠層上。	外指標



(a) BEGIN xxx UNTIL

圖5-4　不定循環結構



(b) BEGIN xxx WHILE yyy REPEAT

圖5-4 （續）

5-3-2 不定循環 (Indefinite Loops)

不定循環的結構中，循環的次數在編譯時無法決定而須在執行時由一些條件來決定。不定循環有兩種不同的形式（圖5-4）：

(1) BEGIN words Condition UNTIL

繼續不斷執行words代表的指令，當Condition條件留一個眞的旗號在疊層上時才終止循環

(2) BEGIN this Condition WHILE that REPEAT

至少執行this一次，若Condition條件爲眞，則繼續執行that，在REPEAT處跳回來執行this。若Condition條件爲假，則終止循環跳到REPEAT以後。

> 注意：在BEGIN之後一定要有一個UNTIL或REPEAT，組成不定循環的結構。

我們可以用DO…+LOOP來建造一個不定循環，就是用一個旗號

來做循環指標的增值。這樣若旗號是假（0）時，指標值不增加循環就可以一直繼續下去。BEGIN … UNTIL的結構却是更緊湊、簡單。因爲BEGIN並不在詞典中編譯任何數據，也不用任何參數，它只是在編譯時註明一個循環的開頭位置。UNTIL在疊層上取用一個旗號，如果這個旗號是零（假），則跳回到BEGIN處繼續循環。當這個旗號不是零的時候，循環結束，跳到UNTIL後的指令繼續執行。下面是一個簡單的例子：

```
: MONITOR BEGIN KEY OVER = UNTIL DROP ,
```

此例中KEY由鍵盤上讀入一個字母的代碼，只有在這個代碼等於執行 MONITOR 前疊層上的字碼時，循環才會終止。例如65 MONITOR 會等到你按A鍵時才跳出來終止循環的動作。

5-3-3　永久循環 (Infinite Loops)

永遠繼續下去的循環稱爲永久循環。用 BEGIN … UNTIL 可以造成一個永久循環的結構：

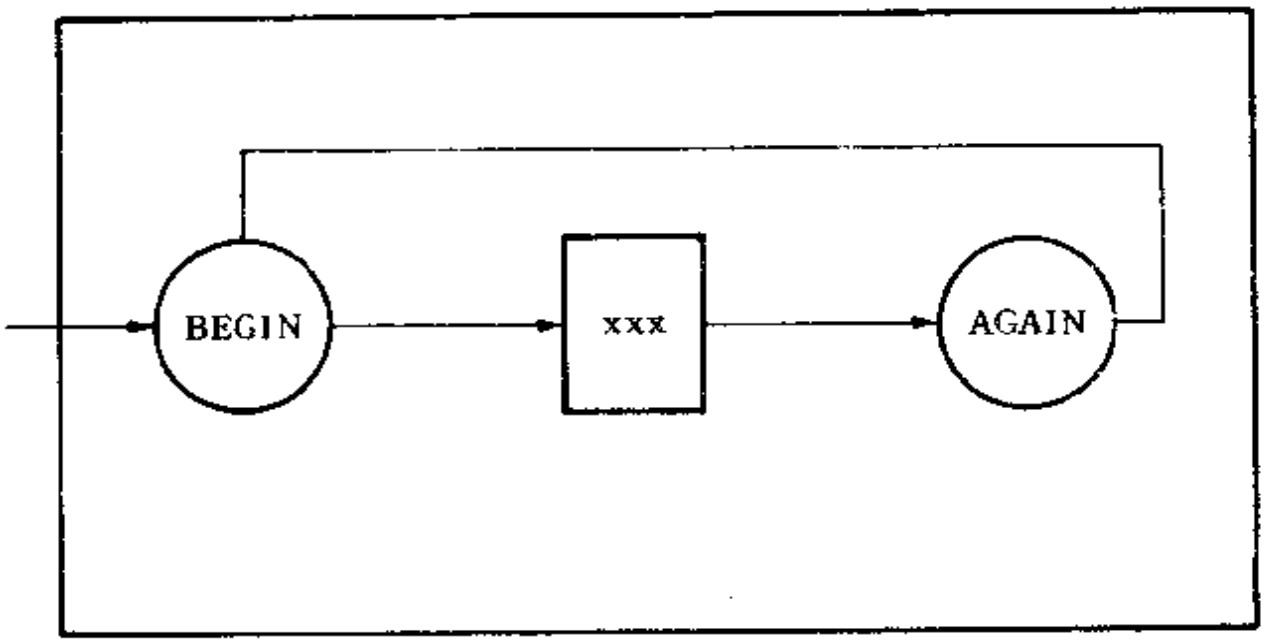
```
… BEGIN … 0 UNTIL …
```

因爲UNTIL看見的旗號永遠是假，所以循環就不會停止。在FORTH-79標準中，有專用的永久循環結構：

```
… BEGIN … AGAIN …
```

BEGIN與AGAIN間所有的指令都循環地執行（圖5-5）。

永久循環通常是用在一個完整的操作系統中作爲系統的主程式。它由一些輸入裝置讀入資料，根據這些資料做一些計算或處理的工作，將其結果輸出，然後再回去重複這同樣的程序。



BEGIN xxx AGAIN

圖 5-5 無限循環的結構

永久循環和不定循環一樣，在循環結構中疊層的高度在執行一遍之後不能改變。否則循環多次之後數據疊層必定會飽和或不足，而會使系統失敗。這是在定義循環時必須切實注意的。另外一點是循環結構只能包括在定義中而不能在定義之外執行。

5-4 循環結構舉例

結構指令及一些返回疊層指令彙集在表5-2、表5-3中，以供讀者參考。以下我們舉幾個例子來說明在FORTH中如何建造循環結構而藉以解決許多不同性質的應用問題。請仔細研究這些例子，以充分瞭解各個循環指令的用法。

1. 可變的上限

假如我們要定義一個循環結構用來處理許多不同個數的物件，最方便的方法便是將循環的上限當做一個參數，而在執行前才把需要的上限放在堆疊上。例如：

```
: PRINTS 0 DO . LOOP ;
```

其中我們只給了指標的初值。在使用 PRINTS 前，我們不但要在疊層上有一堆數值，我們還需要知道這些數值的總數。它的使用方法是：

```
OCTAL 10 100 1000 10000 DECIMAL 4
  PRINTS
```

電腦的答案是：4096 512 64 8 OK 。

2. 可變的上限及指標初值

我們可以在執行時才將上限及指標初值置於 DO - LOOP 的定義中。我們只要在定義中略去上限及初值即可。FORTH用的順序是上限先於初值，但平常大家習慣的用法是初值先於上限。因爲LOOP在指標等於上限時即終止循環，我們可以將上限值加 1，使得LOOP能包括上限之值。

在下面的例子中，假設T是 EDITOR 中印出磁碟區某一塊中某一行的指令，下面的定義即可印出好幾行的文字：

```
: LINES 1+ SWAP DO I EDITOR T LOOP ;
```

在使用時 1 16 LINES 即可印出第 1 至第 16 行及其間所有的文字。

3. 函數的積分

此例是將一函數在某一範圍中的值相加或積分。假設函數值是用 FX 指令計算。FX 要一個參數 x 作其變數。以下的指令即可計算其在一定範圍中的積分：

```
: SUM 1+ SWAP DO I FX + LOOP ;
```

SUM 的使用法是：0 100 200 SUM。在執行完畢後，積分值即留

在疊層頂上。積分的範圍是100到200，其前的0是積分的初值。

積分的計算可以用下面的方法更加簡化：

```
: INTEGRATE 0 ROT ROT SUM ;
```

這樣我們就不必在初值及上限之前再放一個零在疊層上了，這個零值對使用者是不大方便。INTEGRATE的用法是

```
100 200 INTEGRATE
```

它的作用和0 100 200 SUM完全相同。

4. 套裝的結構

下面這個例子是將兩個 DO - LOOP 結構套在一起變成一個二度空間的構造，以列印出整負的資料：

```
: DUMP SWAP DO CR
     I 10 + I DO I C@ . LOOP
     10 + LOOP ;
```

DUMP 由疊層上取得兩個地址後將其間所存的全部資料都在終端機上列印出來，每行印 10 個數目。Apple-FORTH系統中的DUMP指令是更漂亮的一個指令，列印得比較整齊。

各種不同的程式結構都可以套裝起來，一個結構可以套在另一個結構中，但兩個不同的結構却不能部份重疊。例如：

```
-LOOP ?DUP IF 0 DO PRINT LOOP
  THEN ;
```

-LOOP只有在疊層上的上限是正值時才做PRINT的工作。

5. 減值計數器（Declining Counter）

下面的例子相當複雜，除了有套裝的 DO-LOOP 和IF-ELSE-THEN結構外，也用+LOOP指令做一個減值計數器的工作。我們給它一個地址做爲參數，由此地址加100開始往回將地址中的數值一個個地取出來與LIMIT 比較。若遇到一個數值比 LIMIT 大，這個數值的地址與其負值就放在疊層，然後在+LOOP時指標與其負值相加成爲零，超出了上限的範圍而停止循環，且將找到的地址留在疊層上。假若所有 50 個數值都未超過 LIMIT，疊層上留下的即是零：

```
: CHECK 0 SWAP DUP 100 + DO
    I @ LIMIT >
    IF DROP I 1 NEGATE
    ELSE -2 THEN
    +LOOP ;
```

在這個例子中有好些有用的技術，值得讀者仔細地研究。

下面圖5-6中圖示一些正確的及不正確的套裝結構。

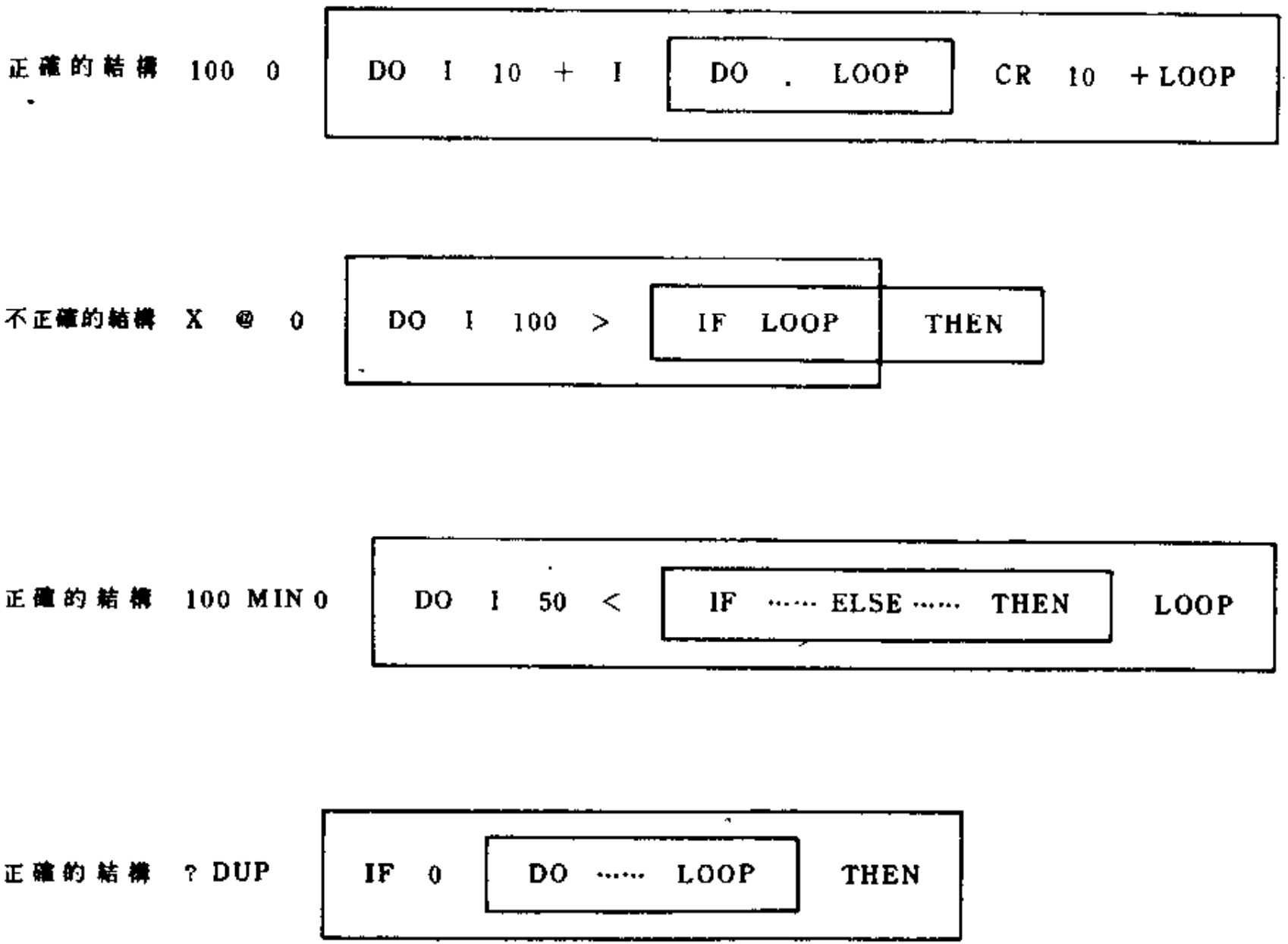


圖5-6 正確與不正確的套裝結構

練習題五

1　0＝與 NOT 有何不同？

2　定義一個指令，當疊層最頂上的兩個數值都是零時，它會在終端機上印出一個 4 ，否則印出 5 字。

3　定義一個求階乘（factorials）的指令。

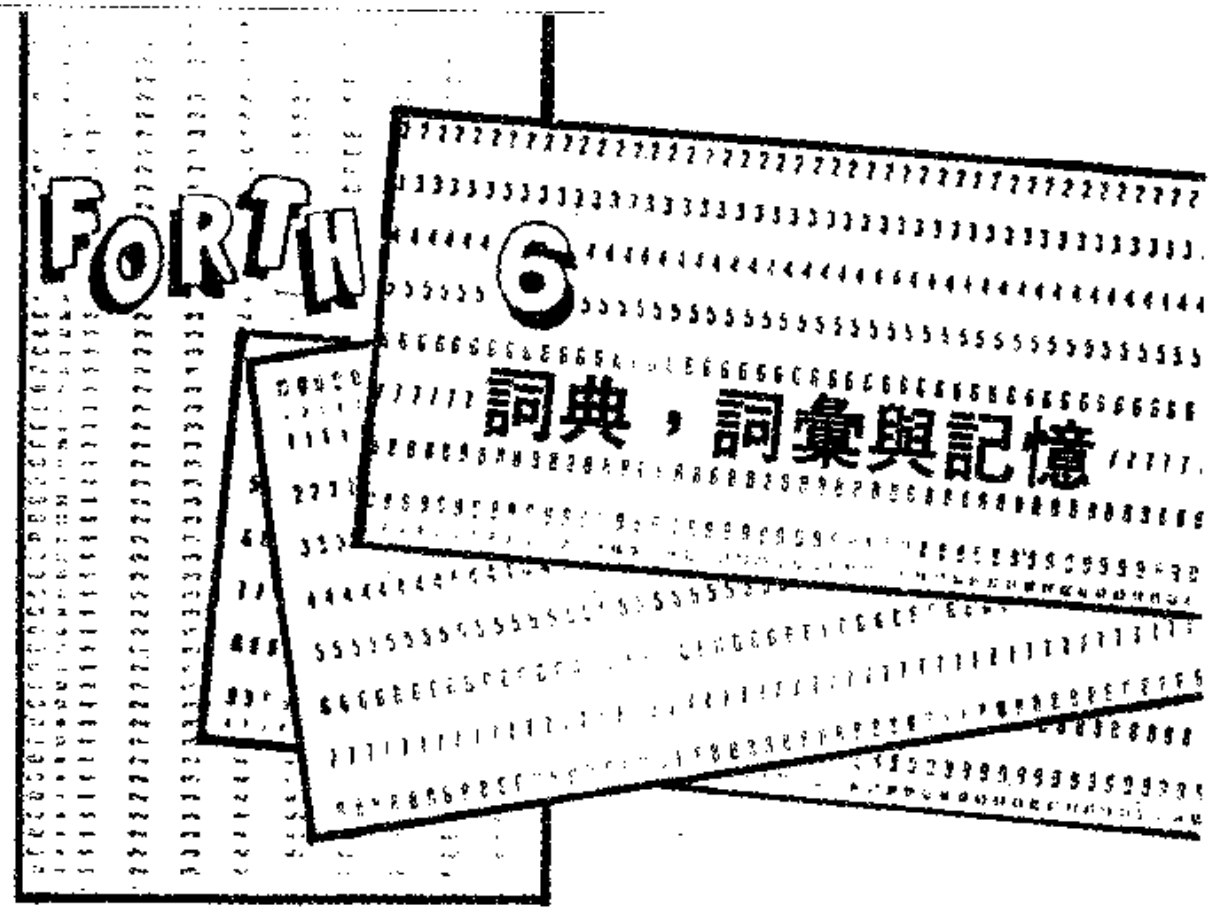
4　定義一個求階乘的指令，積用 32 位元的雙整數來表示。

5　第四題的指令不能用負數，寫一個有保護功能的指令。

6　定義一個指令可以印出 n 個由 0 開始的整數之和，$n \leq 10$ 。

7　在 1624 年印第安人以 24 元美金將紐約市賣給白人。假如這筆錢是以年息 6 %存在銀行中的話，現在該有多少錢了？用單利來計算。

8　第七題中的問題，但用複利來計算。



雖然詞典、詞彙等觀念相當抽象不易說明及了解，但這些問題是FORTH系統中十分重要的支柱。讀者若要有效地利用FORTH來寫程式，這些觀念和相關的指令也都必須相當熟悉才行。

6-1 記憶區分配

請複習圖1-1中Apple-FORTH 記憶區的名稱、地址及其功用。

Apple-FORTH是架在Apple Ⅱ 電腦上的一個操作系統，它對記憶體的區分及利用，一部份是決定於其中央處理單位（Contral Processing Unit 或CPU）6502的特性。6502中，第0項與第1頁記憶有特殊的用途。第0頁前面 88 個字元是給Apple的監督程式，貯存監督程式所需用的變數資料，由88至255字元是保留做爲數據疊層。數據疊層是由地址255處向下延伸。

第1頁的記憶有兩個用處，一是做爲返回疊層，它的基本地址是

512，向下伸。另一個用途是做爲輸入字串的暫存區（Terminal Input Buffer），它的基本地址是256，向上延伸，因爲返回疊層和輸入暫存區的長度是互補的，所以共用第1頁記憶不會有衝突。

第2頁記憶是Apple的BASIC系統用來儲存輸入字串的，第3頁也是給BASIC儲存它的系統資料的。這兩頁在Apple-FORTH中都不使用。使用者可以利用它們來儲存他自己的數據。要注意的是使用時不要與監督程式衝突。

第4至7頁是Apple的文字顯示區及低解相度圖形區，裡面的文字資料隨時展示在CRT螢幕上。第8至11頁是第2頁的文字和低解度圖形區，可以與4至7頁交替使用。在Apple-FORTH中，文字與圖像的顯示也是經由這個區域。它的用法是照Apple Ⅱ的規則。

第12至15頁是空的。

第16至31頁有4K字元，是使用者詞典區。使用者定義的新指令，編譯之後就存在這一區中，成爲系統詞典的一部份。4K字元對FORTH來說是很大的記憶，可以容納很大的應用程式。如果程式長度超過4K字元，還可以向上延伸進入32頁以上的高解相度圖形記憶區。

32至63頁是第一個高解相度圖形區，64至95頁是第二個高解相度圖形區。這兩個區如果不用來顯示圖形，可以用作詞典的擴充區，或者用來做第二個假磁碟，可以使磁碟記憶區增加16K字元的容量。

96頁至191頁有24K字元的容量是Apple-FORTH主要的磁碟記憶區，或稱爲假磁碟（Psvdo Disk）。它是分爲48個塊，每塊是512字元長，用以貯存原程式文字或數據資料。假磁碟的資料可以和磁帶相互傳遞。

第192至207頁是Apple的輸入輸出界面區。6502的輸入輸出是用記憶指令來控制的，這一段4K字元的地址，保留起來給輸出／入裝置使用，使得Apple能與許多不同的界面裝置連結起來，應付不同的問

題及要求。

第208頁到239頁即是 Apple - FORTH 系統的詞典，含有所有的系統指令，它取代了BASIC 的地位，開機之後就進入FORTH系統開始操作。第240到247頁目前是空的，以備日後用機器碼擴充FORTH系統時使用。最後248到255頁則是Apple的監督程式。這個監督程式必須保留在 Apple - FORTH 中，因爲許多 I / O 的工作還是透過監督程式中的一些副程式來做的。

Apple - FORTH至少要16K RAM才能動作，加到48K 字元的RAM後，Apple - FORTH 就可以有一個 24K 的假磁碟（並可以擴充到 32K 字元）。這個假磁碟可以相當方便地與磁帶錄音機交換資料，使得這個系統可以用來發展大型程式系統。

6-2　磁碟記憶

標準的FORTH系統中，因爲要利用磁碟來儲存大量的程式及資料，採取了虛記憶（Virtual Memary）的方式來使用磁碟。虛記憶是以磁碟記憶模擬電腦主記憶的一種方法，使用者可以用主記憶的讀取及儲存指令來處理磁碟上的資料。

FORTH將磁碟區分的塊，在磁碟上每塊都有一個連續的序數，系統用這序數來輸入或輸出整塊的資料。在輸入時，整塊的資料由磁碟上讀入主記憶的一個磁碟暫存區（Disk Buffer），使用者就可以自暫存區中提取所須的資料或改變其內容，當系統需要這個暫存區存取其他的資料時，改動過的塊就會被輸出存到它的磁碟上原來的位置。如此，使用者可以自磁碟上任何位置取得他要的資料而不必擔心這些資料的讀入及寫出的詳細工作方式。

表6-1中列出了使用磁碟記憶的指令，要取得第n塊中的資料所用的基本指令是

n BLOCK

BLOCK將第n塊資料讀入一個磁碟暫存區之後，便將此暫存區開始的地址留在疊層上。舉一個例子來看，如果我們在第 13 塊的第0行中有有這樣的一段文字：

(PRACTICE BLOCK)

表 6-1 磁碟指令

指　　令	疊 層 動 作 說 明	中文指令
BLOCK	(n — a) 將第 n 塊資料輸入到一個磁碟暫存區中，並將此區開始地址留在疊層上。	塊
BUFFER	(n — a) 在磁碟暫存區中尋找一個暫存區以容納第 n 塊的新資料。將暫存區地址留在疊層上。	暫存區
UPDATE	(—) 將最近使用的暫存區標明其中資料已被改動。	新
SAVE-BUFFERS	(—) 將所有改動過的暫存區資料存回磁碟。	存塊
EMPYT BUFFERS	(—) 將所有暫存區記憶都清除爲零，以免其被存回磁碟。	清塊
LIST	(n —) 將第 n 塊文字輸入至一暫存區並列印出來。	列表
LOAD	(n —) 將第 n 塊文字輸入並將其編譯或執行。	編塊
SCR	(— a) 存放現用塊數的系統變數。	現用塊

如果我們按入下列的指令

```
13 BLOCK C@ . 40 OK
```

13 BLOCK將第13塊的資料讀入暫存區，並將此暫存區的第一字元的地址留在疊層上。C@將此字元取出放在疊層上，．即將其值取出。因爲在括號的ASCII代碼是40，即是印出的數字。

因爲BLOCK將暫存區的開始地址留在疊層上，我們可用這個地址爲基底而去尋找塊中任何一個字元的值。例如下面的指令

```
13 BLOCK 4 + C@ EMIT A OK
```

將暫存區的第5個字元找出並列印出來，即是A的字母。

通常BLOCK所用的參數是一個 16 位元的單整數，其中最高次的第16位元稱爲UPDATE 位元。所以實際上BLOCK可使用的塊數是由0到 32767 ，如果每一塊存 1024 字元的話，BLOCK可使用 32 Mbytes 的磁碟。

UPDATE位元是一個旗號，當EDITOR的指令將一塊中的資料改動的時候，這指令也就將這個位元設爲 1 ，即是告訴操作系統此塊中資料和存磁碟上的不相同。下次當系統需要使用此塊的暫存區時，在讀入另一塊資料之前，會先將此塊資料抄錄到磁碟原處。這樣可以保證磁碟上的資料在適當的時候被更新，同樣也將磁碟的讀寫動作減至最少。如果 UPDATE 位元是 0 ，表示此塊中資料未經改動而與存在磁碟上者完全相同，在這種情形下，此塊資料不必抄錄到磁碟上去。所以當系統要用此塊的暫存區時，就可以直接將另一塊資料讀入此暫存區，不必做輸出至磁碟的工作。

若要强迫所有改動(UPDATE)的塊都寫回磁碟上去，可用SAVE-BUFFERS或FLUSH的指令。在關機之前一定要先下三個指令，否

則最後的改動的幾塊就很可能沒有機會被抄錄回到磁碟上去。許多FORTH程式師都在做一些編輯工作後立即按SAVE-BUFFERS的指令，避免誤動或遺漏的不幸情形。

EMPTY-BUFFERS是將所有的磁碟區都清除爲零或是空格。這是當使用者發現一些暫存區中的資料被意外的改動了，而他不希望這些無用的資料被寫進磁碟去。暫存區清除之後，使用者可以再用 LIST, LOAD, BLOCK 等指令重新將磁碟上原存的資料讀到暫存區中去。

磁碟塊是用來儲存原程式的文字資料或數據資料的。文字資料是用ASCII 代碼，一碼佔一字元，一塊即有1024字（或512字）。數據資料可用字元或數元的方式儲存，視實際應用之目的而定。資料的存取都是以BLOCK指令的樞紐。舉一個例子是由一個A/D轉換器錄下100點的數據並將其存入磁碟。假設A/D是一個指令，將轉換器讀入的資料以一個數元（16位）的形式堆上疊層。我們可以定義一個記錄數據的指令：

```
: DATA 2 * 20 BLOCK + ;
```

2＊是將數元的序數變成字元的序數，20 BLOCK＋是將字元的序數加到第20塊在磁碟暫存區的基本地址而得到這個數元應該存入的地址。

其次我們再定義

```
: INPUT 100 0 DO I A/D
    I DATA ! UPDATE LOOP ;
```

這裡面 I A/D 用循環指標I做爲序數讀入一個數據，I DATA !則將此數據存入100個資料應在的地址。UPDATE則是保證此塊資料最後會被錄上磁碟去。按入INPUT指令即會將100點資料存入第20塊中，以循環指標當做數值資料的序數。這是將資料儲存到磁碟上去的一

種方法。

SCR是一個系統變數，儲存現用塊的塊數EDITOR編輯時即用SCR來標示現在正在編輯的塊。如果利用SCR的話，上述的DATA指令可改寫爲：

```
: DATA 2 * SCR @ BLOCK + ;
```

這樣可以不限定只用第 20 塊來存資料，而可用任一塊來存。只要將SCR中存入要用的塊數即可。

仿照 SCR 的用法，我們可以將許多塊定出不同的名稱，以便使用。例如：

```
: RAW 20 SCR ! ;
: SCALED 22 SCR ! ;
```

有了這兩個定義，以後我們可以用下列的指令句：

```
I RAW DATA @ SCALE
I SCALED DATA ! UPDATE
```

將RAW塊（第20塊）中的資料取出，經過一番計算處理之後，將處理過的資料存到SCALED塊（第22塊）中去。

爲了不使塊數超出磁碟的容量，我們在DATA中可以加入一些安全檢查的步驟，限制塊數在一定的範圍中：

```
: DATA 47 MIN 0 MAX
  2 * SCR @ BLOCK + ;
```

如此即强迫可用的塊數在0與47之間。

假如有些數值資料的數量超過一塊所能容納的，我們可以用 1024

/MOD 兩個指令將資料的序數分爲塊數增值及字元序數兩部份分別處理：

```
: DATA 2047 MIN 0 MAX 2 *
  1024 /MOD SCR @ + BLOCK + ;
```

此外還可以有許多的變化，讓我們可以自由地使用磁碟來存取各式的資料。我們希望這些討論能啓發你的思考及創造力，利用FORTH現有的這些工具去解決你的問題。

6-3 Apple-FORTH的假磁碟(Pseudo disk)

以上所討論的磁碟指令是一般FORTH系統用來指揮磁碟動作的指令。在Apple-FORTH 中沒有眞正的磁碟，只用了記憶的一部份(24K字元）來做一個假磁碟，而且每一塊的長度限制爲512字元。所以許多磁碟指令在此都不適用而須要稍加改變。其中最重要的BLOCK指令是這樣定義的：

```
: BLOCK 48 MOD 512 * FIRST + ;
```

48 MOD將塊數限在0至47之間，512 *即得到此塊開始的字元序數。FIRST 是假磁碟開始的地址，加上字元序數就得到了這一塊第一個字元的地址。這樣我們實際上是把假磁碟當做 48 個磁碟暫存區來使用。

在Apple-FORTH中，UPDATE，SAVE-BUFFERS，EMPTY-BUFFERS等指令都不做任何工作，它們之保留在系統中是爲了符合FORTH-79標準的要求。

6-4　指令的結構

FORTH指令都有相同的結構，不論是高階指令、低階指令、常數或是變數。在圖6-1中顯示一個高階指令的構造做爲例子。每個指令都分爲四欄：名稱欄、連結欄、解碼欄及參數欄。以下我們詳細地說明各欄的內容及其功能。

指令最前面的一欄是名稱欄（Name Field），這欄的長度是隨着指令的名稱長度而定的。其第一個字元中存的是此指令名稱的字母總數。例如：PHOTO GRAPH指令有10字元，第一字元所存的數目就是10。第二至第十一字元就存了這個名稱的所有字母。在標準的FORTH系統中，指令的名稱最多可以有31個字母。可用的ASCII字碼有124個，只有換行（CR），零碼（NUL），退格（BS）及空格（SP）四個字碼不能用來做名稱的字母。所以使用者有很大的自由來選擇一個指令的名稱。

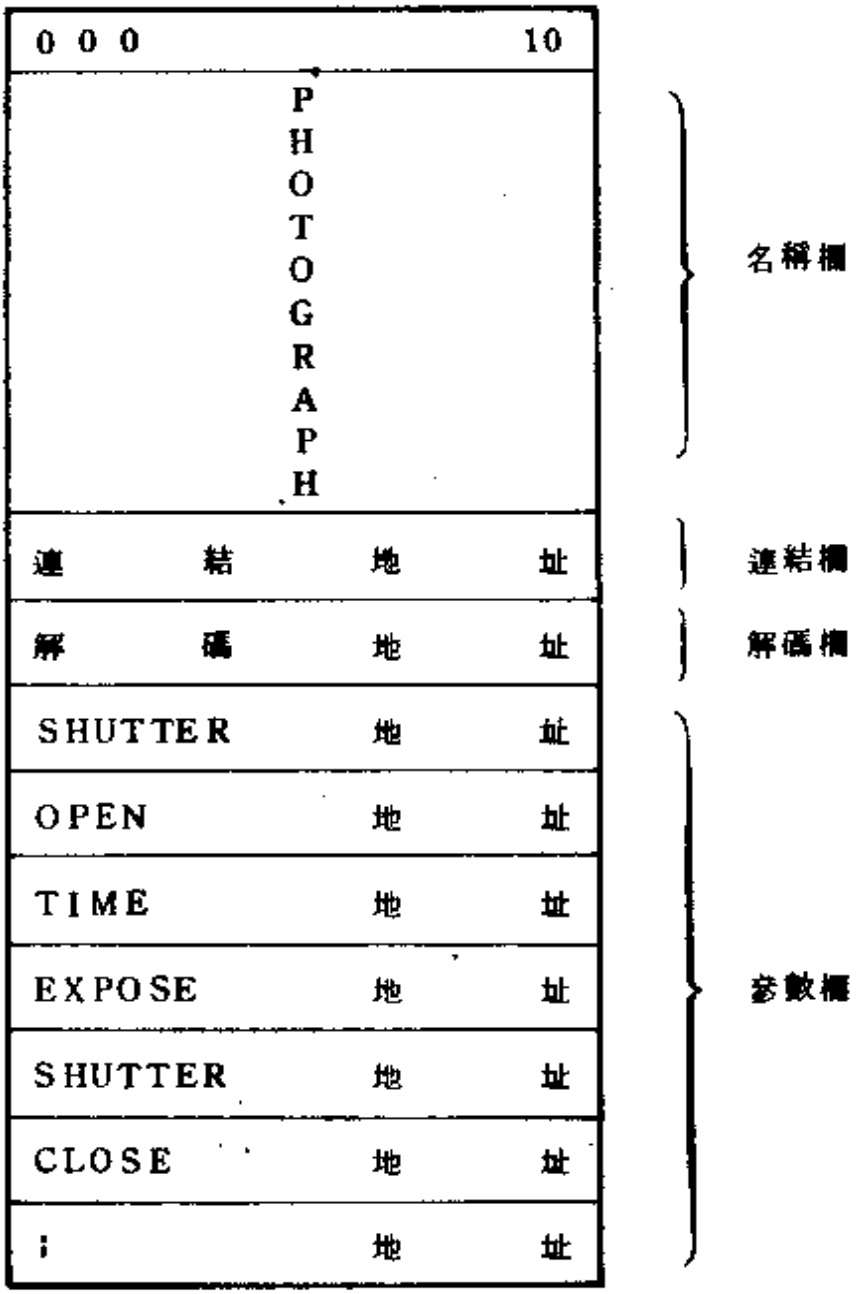
名稱欄的第一字元及最後一字元的第7位元（MSB）都定爲1，以指示名稱欄的範圍；第一字元的第6位元稱爲優先位元（Precedence Bit），這個位元用來控制FORTH的編譯工作。凡是編譯命令（Compiler Directives）在編譯工作進行時不被編譯加入詞典而必須立即執行，以完成特殊的編譯工作者，其優先位元都是定做1。大部份的指令優先位元都是零，編譯時其地址就被編入詞典成爲高階指令的一部份。

名稱欄第一字元的第5位元是塗污位元（Smudge Bit），它是在指令沒有建成功的時候保護編碼程式不去編譯這個未成熟的指令。當一個高階指令已經建造成功之後，這個塗污位元就被清除爲零，才能被編譯或被解碼而執行。

名稱欄下面就是連結欄，它存了一個地址，即是詞典中前一個指令的名稱欄地址。名稱欄和連結欄將詞典中所有的指令全部連成一串。在

這個詞典裡去找一個特別的指令就是沿着這一串連的構造，將輸入的名稱，與各個指令的名稱欄比較。若名稱不同，則由連結欄中跳至前一個指令的名稱欄做下一次的指令名稱比較。如此可以在詞典中找出任何指令。

連結欄以下是解碼欄(Code field)，其中存着一個地址，這地址




```
: PHOTOGRAPH SHUTTER OPEN
        TIME EXPOSE SHUTTER CLOSE ;
```

圖6-1 高階指令的構造

所指的記憶中存着一段機器碼。這個指令被找出來將執行時，就是先執行這一段機器碼。不同種類的指令的解碼欄，指着不同的機器碼，這些機器碼程式，即是指令的解碼程式（Interpreter）或稱爲內解碼（Inner Interpreter）以別於操作系統中的文字解碼（Text Interpreter）。

指令的最後一欄是參數欄（Parameter field），參數欄的長度不一定，視指令的種類而定。當執行內解碼程式時，內解碼即利用存在參數欄中的資料，來完成這個指令所規定的工作。常數及變數等指令即在參數欄中貯存常數或變數之值。高階指令的參數欄中貯存的是一系列的地址；這些地址是詞典中其他指令的解碼欄地址。高階指令的解碼程式執行時，就會依序找出這些地址並去執行這些地址所代表的指令。所以高階指令的解碼通常稱爲地址解碼（Address Interpreter）。

在低階指令中，參數欄內就是一系列的機器碼，解碼欄中的地址，就是參數欄的地址。所以執行低階指令時，就會直接執行參數欄中的機器碼程式，它就是低階指令本身的解碼程式（Code Interpreter）。

6-4-1　指令種類的溫習

我們已討論過的指令，主要是由下列幾個定義指令所建造成的：

: CONSTANT VARIABLE CREATE CODE

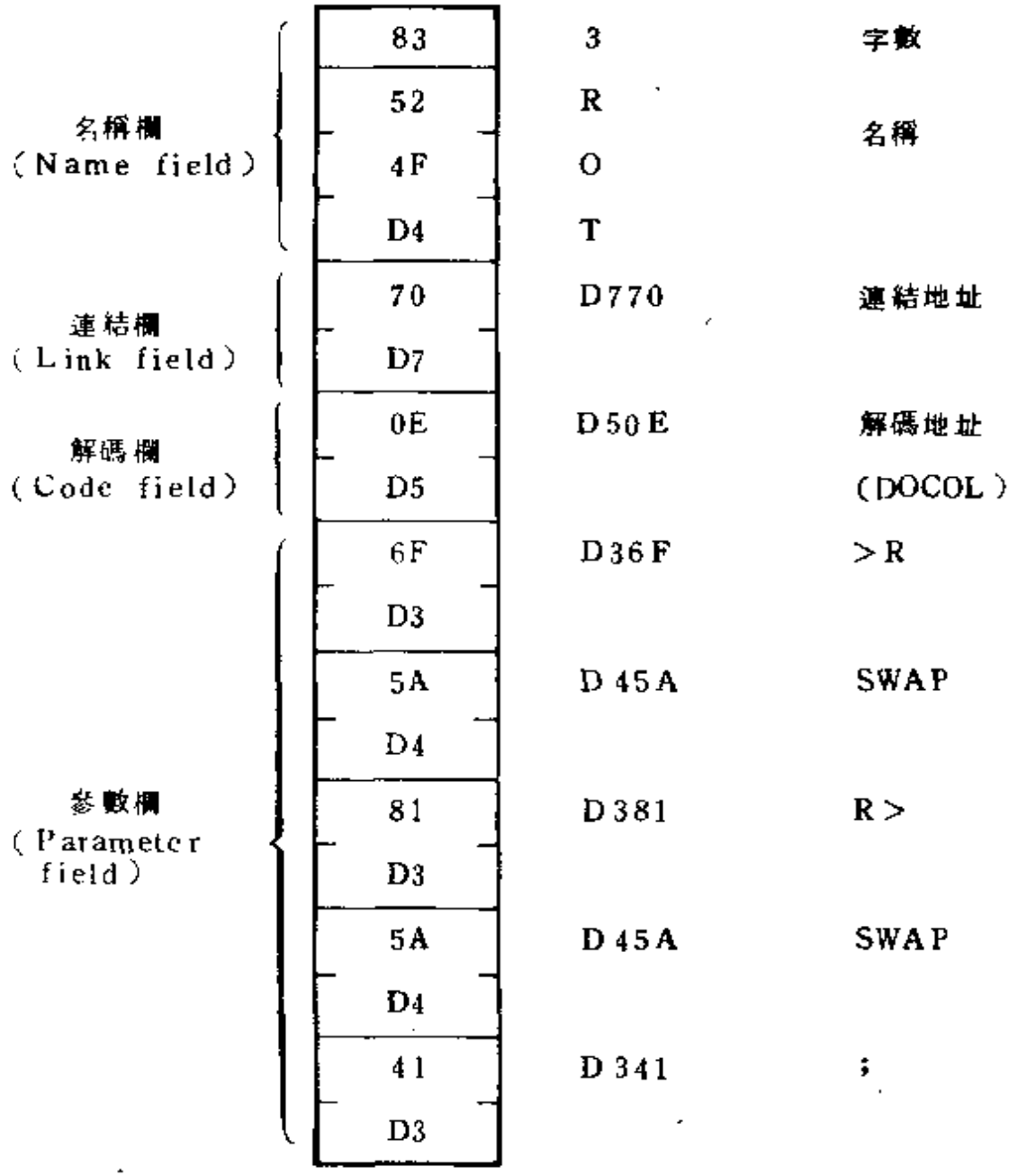
每種不同的指令都有不同的解碼程式。在 Apple-FORTH 中，這些指令及其解碼程式是這樣的：

(1) 用：來定義的高階指令，它的解碼程式是地址解碼（Address Interpreter）。在組合語言原程式中代表的名稱是DOCOL。DOCOL執行一個高階指令中所有的指令，直到見到；爲止。

(2) 用CONSTANT定義的常數，其解碼欄的地址指向DOCON解碼程

式。DOCON執行的結果就是將參數欄中的常數值抄到數據疊層上。

(3) 用VARIABLE定義的變數，其解碼程式在DOVAR處。DOVAR將參數欄的地址抄錄到數據疊層上，以便使用者取用或貯存其中的資料。



: ROT >R SWAP R> SWAP ;

圖6-2 高階指令實例：ROT

(4) 用 CREATE 定義的指令，在執行時也是將參數欄的地址抄到數據疊層上。它與VARIABLE 不同之處乃是在編譯時 CREATE 並不預留 2 字元給參數欄。

(5) 用CODE 定義的低階指令，解碼欄中存的地址就是參數欄的地址。參數欄中的機器碼就是這個低階指令的解碼程式。

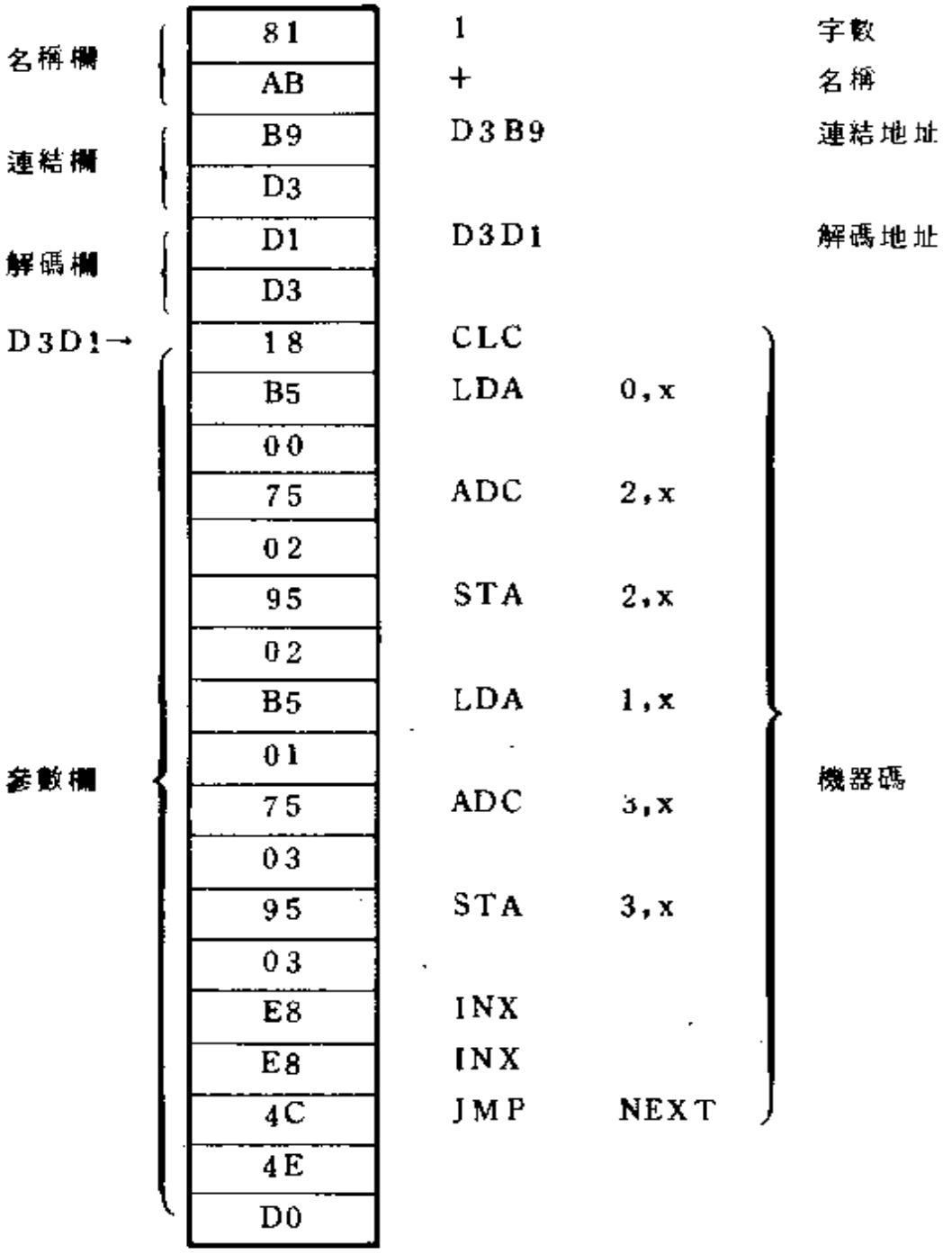
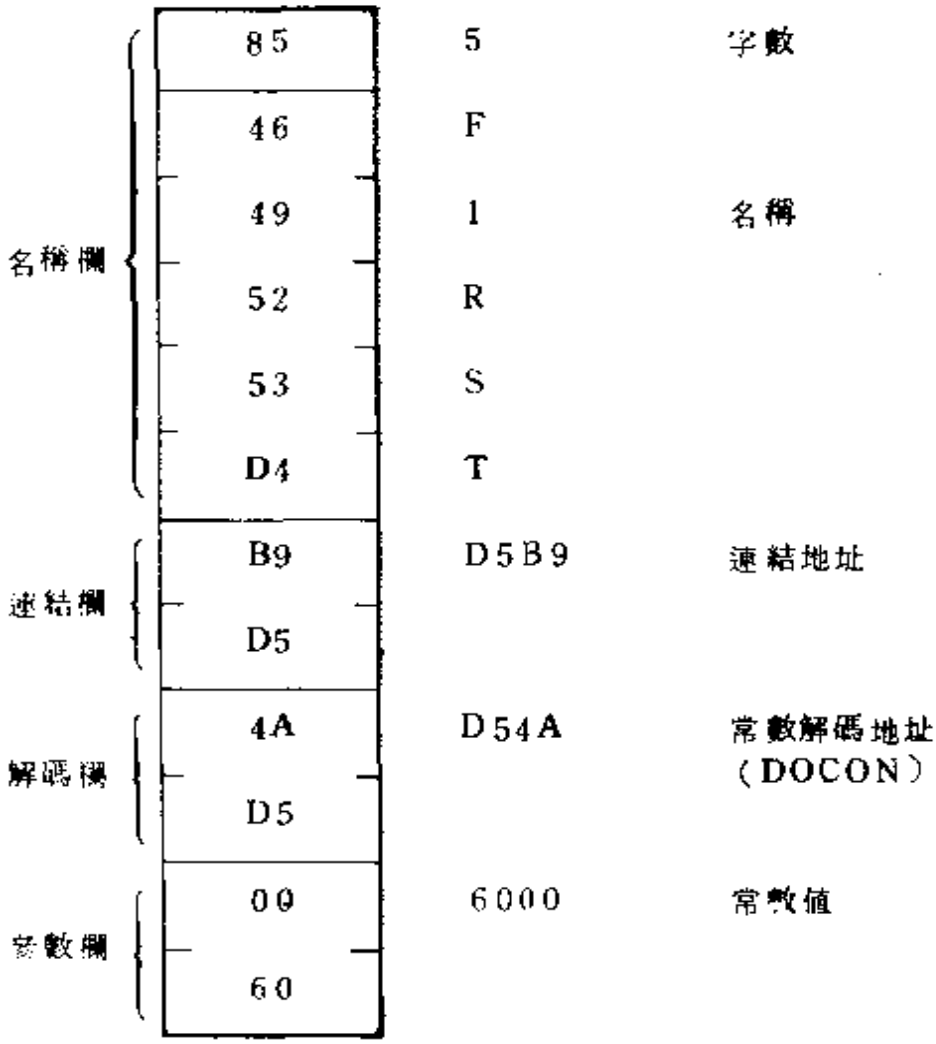


圖 6-3　低階指令實例：+.

圖 6-2 至圖 6-5 中舉例說明這幾種不同指令的實際構造。

當你熟悉了 FORTH 的語言構造並對它的編譯及解碼等工作的方式有更深一層的瞭解之後，你應該就能用＜CREATE……DOES＞或＜BUILDS……DOES＞等指令來自己定義你自己所需要的特殊定義指令。你自己設計的定義指令能針對你的問題，在詞典中建立最有效的指令或結構，來解決你的應用問題。




```
6000  CONSTANT  FIRST
```

圖 6-4　常數指令實例：FIRST

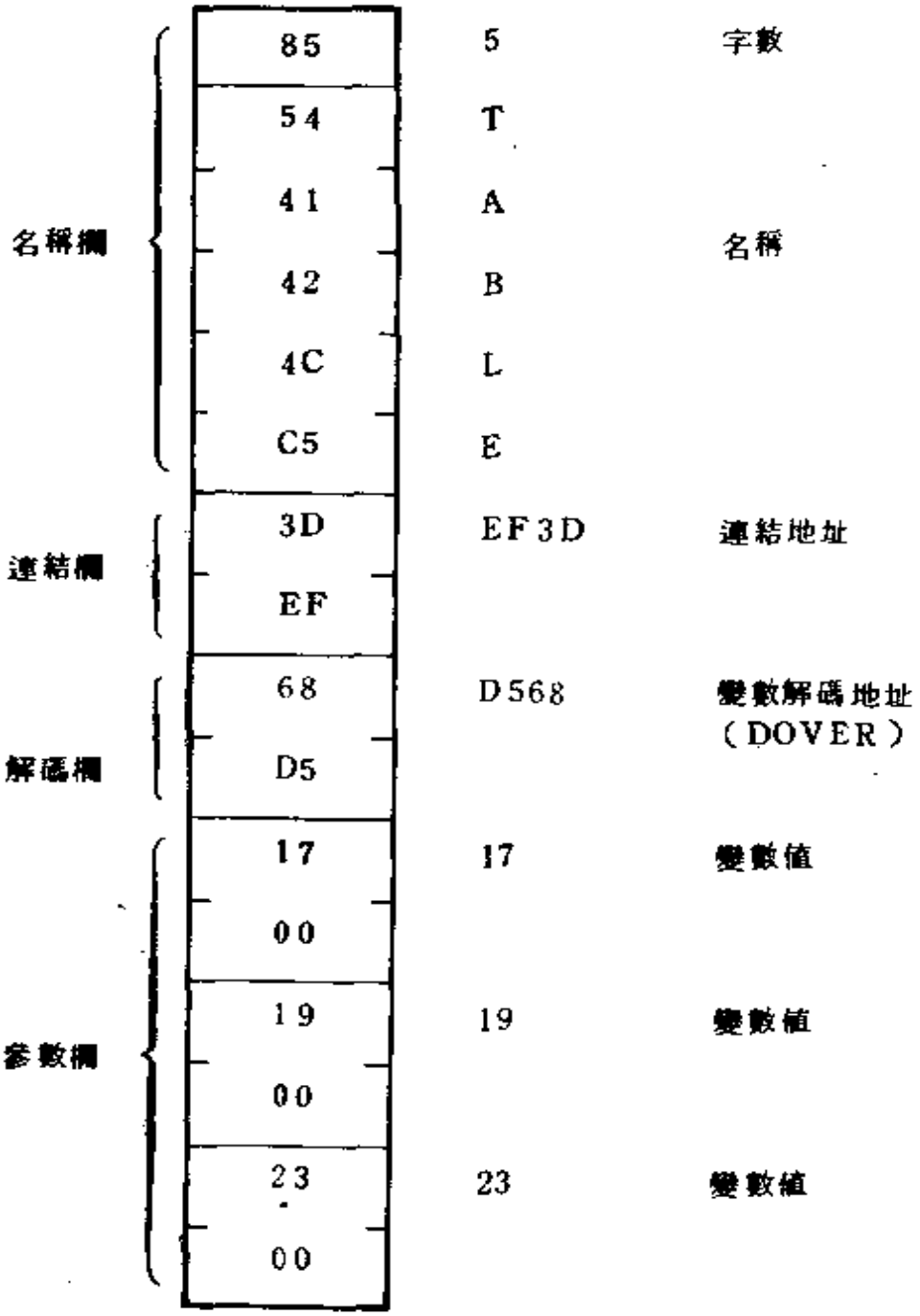


圖6-5　變數（數陣）指令實例：TABLE

6-4-2　詞典指令

表6-2例舉一些指令用來控制或改變詞典的內容。這些只是編譯程式中用到的工具，但使用者也可以酌量利用。

表 6-2 詞典指令

指　　令	疊 層 動 作 說 明	中文指令
DP	(— a) 存放詞典頂端地址的系統變數。	詞典指標
HERE	(— a) 詞典頂上第一個可用字元的地址。	此
ALLOT	(n —) 在詞典上留下 n 字元的空間。	留
,	(n —) 將疊層上數值 n 編在詞典最上面。	編
C,	(n —) 將疊層上的字元 b 編在詞典最上面。	編字
LITERAL	(n —) 將疊層上的數元 n 按眞值方式編入詞典。	編數
COMPILE	XXX (—) 將後續的指令 XXX 地址編入詞典。	編譯
(COMPILE)	XXX (—) 將後續的立即指令 XXX 的地址編入詞典。	立即編譯

DP 是一個系統變數，做爲詞典指標之用。DP 中所存的地址是詞典頂端第一個可用字元的地址。DP 所指的字元及其以上的記憶區，一般稱之爲名稱暫存區（Word Buffer），用來存放解碼時輸入的指令名稱。在建造一個新指令的時候，新指令的名稱欄就可由此處開始。

HERE 的定義很簡單：

```
: HERE DP @ ;
```

只是把DP所存的詞典指標堆上疊層而已。HERE就把名稱暫存區的地

址放在疊層上，我們可藉此索取這個暫存區中的資料，或是存入新的資料。

ALLOT 指令將詞典指標搬動到另外一個地址。它的定義是：

```
: ALLOT DP +! ;
```

將要搬動的字元數加到 DP 中指標原值上。這樣我們可以在詞典中留一段空檔，以備儲存相當量的資料。例如：

```
VARIABLE DATA 98 ALLOT
```

先保留了兩個字元給DATA 做參數欄，98 ALLOT即是將DATA的參數欄增加了 98 個字元，所以DATA 變成一個有 100 個字元容量的數陣了。

另一個重要的編譯指令是“ , ”（逗點Comma）。它的定義是

```
: , HERE ! 2 ALLOT ;
```

將疊層上的數元存在詞典頂上。再將 DP 向上移兩格。這就將一個數元編入了詞典。“ , ”事實上就是FORTH中最基本，也是最原始的編譯程式。用這一個指令，我們可以將任何資料編入詞典。

標準的FORTH系統中都有一個字元編譯指令“C,”將字元資料編入詞典：

```
: C, HERE C! 1 ALLOT ;
```

因爲在 8 位元的微處理機上，許多資料或機器碼都是以 8 位元爲單位的。用 C, 來編譯這類資料是比較適宜的。

6-4-3 詞彙指令

FORTH的詞典是一大堆串連起來的指令。文字解碼的工作就是在這一條串連的指令鏈中找出所要用的指令。事實上，在詞典中的指令，並不是串連成爲一條鏈，而是並存着有好幾條鏈。每一條鏈都串着一些相關的指令，這種指令的鏈型構造，稱做詞彙（ Vocabulary ）。表6-3 中我們列出了一些與詞彙相關的指令。

在 Apple-FORTH 及一般通用的FORTH 系統中，經常有下列三個詞彙：

表6-3 詞彙指令

指　　令	說　　明
FORTH	指定FORTH詞彙爲語意詞彙。
EDITOR	指定編輯詞彙爲語意詞彙。
ASSEMBLER	指定組合詞彙爲語意詞彙。
CONTEXT	儲存語意詞彙指標的系統變數。
CURRENT	儲存現用詞彙指標的系統變數。
DEFINITIONS	將語意詞彙設定爲現用詞彙。
FORGET XXX	將指令XXX及其後的指令全部清除。
COLD	將使用者加入的所有指令都清除到開機或重置時的狀態。
VOCABULARY XXX	定義一個新詞彙並將其命名爲XXX。
' XXX	在詞典中尋找XXX指令，並將其參數欄地址留在疊層上。若無此指令，則印出一錯誤信息。

(1) FORTH　　包括所有的基本算術、疊層，I/O，及記憶的指令。

(2) ASSEMBLER　　包括建造低階指令所需要的所有工具，如機器碼代碼（Mnemonics），及組成結構的指令等。

(3) EDITOR　　輸入並修改原程式文字並將其存放在磁碟上所有的指令。

當我們需要用到某一個詞彙中的指令時，只要先按入詞彙的名稱如FORTH，EDITOR，ASSEMBLER 等，即可選用某一詞彙作爲詞意詞彙（Context Vocabulary）而由其中優先找尋指令。開機後，FORTH即被自動設定爲詞意詞彙。

當我們執行一個詞彙指令如EDITOR時，EDITOR的地址即被存進了系統參數CONTEXT中。EDITOR的參數欄中存有EDITOR 詞彙中最後定義之指令的名稱欄地址，文字解碼即是由這個指令開始尋找所指定的指令。例如I在FORTH中是將循環指數抄到疊層上，而在EDITOR 中是要插入一段文字串。如果我們要用在 EDITOR 中的I指令，則要用以下的指令句：

```
…… EDITOR  I  FORTH ……
```

如在EDITOR中而要用FORTH的I時，則要

```
…… FORTH  I  EDITOR ……
```

詞彙指令FORTH，EDITOR及ASSEMBLER改變了CONTEXT 的內容而規定了詞意詞彙的起點。

在標準的FORTH系統中，EDITOR及ASSEMBLER 等詞彙最後還是連到FORTH詞彙上。所以如果一個指令在 EDITOR 中找不到

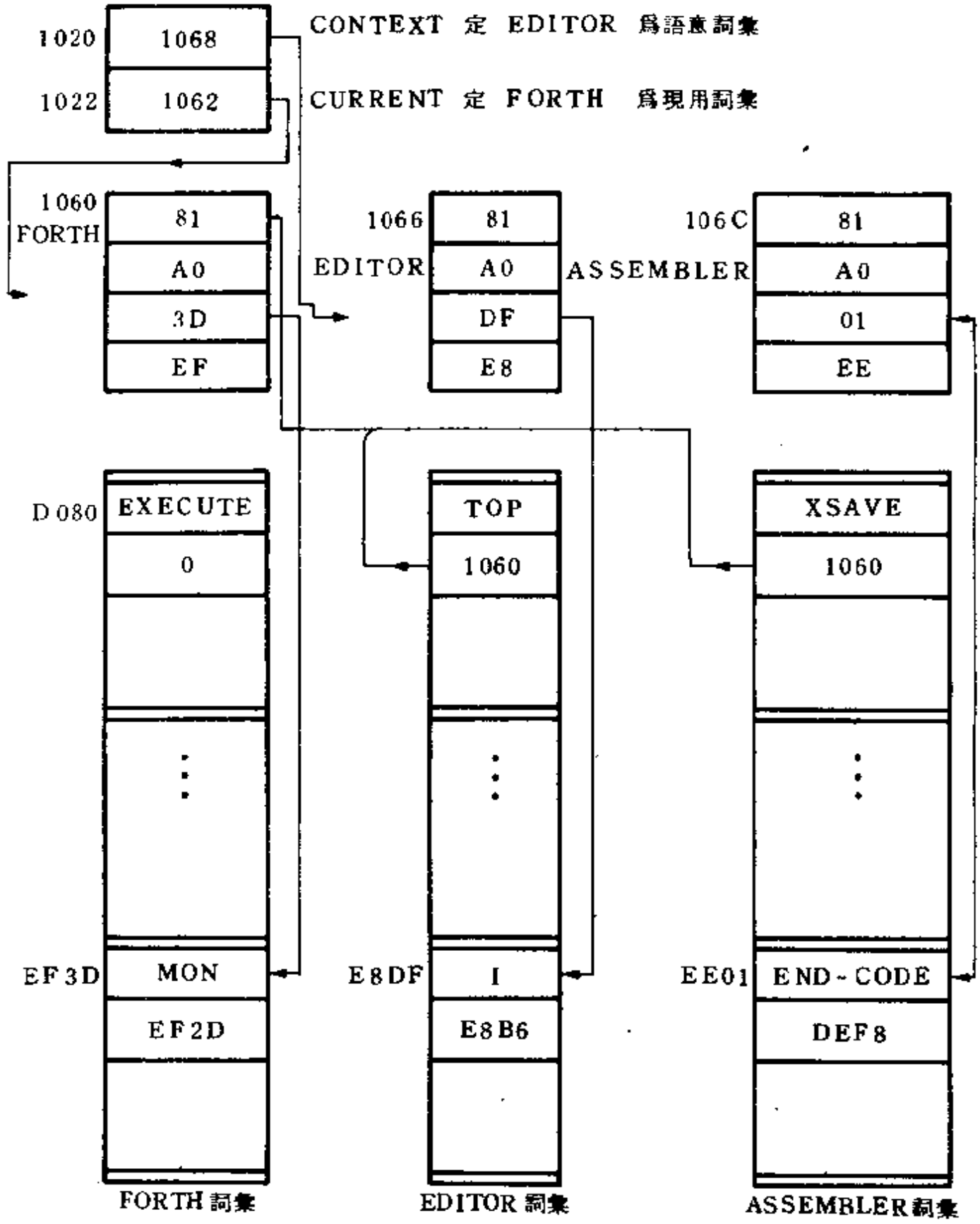

1020
1068
CONTEXT 定 EDITOR 爲語意詞彙
1022
1062
CURRENT 定 FORTH 爲現用詞彙
1060
FORTH
81
A0
3D
EF
1066
EDITOR
81
A0
DF
E8
106C
ASSEMBLER
81
A0
01
EE
D080
EXECUTE
0
TOP
1060
XSAVE
1060
EF3D
MON
EF2D
E8DF
I
E8B6
EE01
END-CODE
DEF8
FORTH 詞彙
EDITOR 詞彙
ASSEMBLER 詞彙

圖 6-6 詞彙的連結

，文字解碼還會去FORTH詞彙中再尋找一遍。FORTH中的指令不與詞意詞彙中同名指令衝突者，都可以自由使用。詞彙中的觀念使得不同的指令可以用相同的名稱共存於詞典中，而以詞意詞彙指令來選擇執行其中之一。CONTEXT即是詞意詞彙的指標。這些連結的情形如圖6-6所示。

在發展程式時，我們定義的新指令通常是連在FORTH詞彙的頂上，這樣可以，這樣可以很方便地執行，因爲FORTH通常是詞意詞彙。我們自行定義的新指令也可以連到別的詞彙裡去，例如我們要加一個新指令進入 EDITOR 詞彙時，可先按入以下指令：

```
EDITOR DEFINITIONS
```

EDITOR將編譯詞彙設爲詞意詞彙，即是將EDITOR參數欄的地址存在CONTEXT中。DEFINITIONS將CONTEXT中的地址抄錄到另外一個系統變數CURRENT中去。CURRENT即指定EDITOR 成爲現用詞彙（Current Vocabulary），此後所有新增加的指令都連結到EDITOR詞彙裡去。要再設定FORTH爲現用詞彙時，所要用下面的指令：

```
FORTH DEFINITIONS
```

使用者也可以設立自己專用的詞彙，例如一個APPLICATIONS詞彙：

```
VOCABULARY APPLICATIONS IMMEDIATE
```

VOCABULARY 是一個定義指令，在詞典中建立一個新的詞彙指令，命名爲APPLICATIONS。因爲詞彙指令常常要用爲編譯指揮指令（Compiler Directive），在編譯的過程中設定詞意詞彙，所以詞彙之定義指令都必須用 IMMEDIATE指定其執行優先（Precedence）加

高。定義了APPLICATIONS，要將指令加入這個詞彙以前要用下列指令句，設定APPLICATIONS爲現用詞彙：

```
APPLICATIONS DEFINITIONS
```

而後即可將新指令加到這個新詞彙裡去。

要尋找在詞典中某一個指令所在的地址，可以用“ ’ ”（Tick）這個指令。“ ’ ”指令若找到所要的指令，就會將這個指令參數欄的地址放在疊層上。例如：

```
’ BLOCK
```

即將BLOCK指令的參數欄地址堆上疊層。如果詞典中沒有這個指令，系統會在終端機上顯示“ ？ ”號或一個錯誤訊號。’指令一個有趣的用途是改變一個常數的值，例如

```
10 CONSTANT TEN
TEN . 10 OK
20 ’ TEN !
TEN . 20 OK
```

20 ’ TEN !即是將20存進TEN常數指令的參數欄，此後執行TEN時，疊層上即堆上一個20的數值。

6-4-4 清除詞典

在發展程式時，我們往詞典上加入許多指令，有些指令會重複地改動。每新定義一次，詞典中就多加了一個指令。爲了要節省詞典的空間，有時我們必須把詞典清理一下，除去不必要的指令或重新開始一段發展程式的工作。

將使用者加入的指令全部清除而使詞典恢復到開機時候的狀態，即是所謂重置（Reset）的指令是 COLD 。執行 COLD 指令後，整個FORTH系統所有重要的系統變數都恢復到開機時應有的值，兩個疊層也都清除，並設定FORTH爲現用及語意詞彙，輸入裝置也定爲終端機的鍵盤，印出一段開機文字後等待使用者由鍵盤上按入指令。

如果只要清除部份指令而不要清除全部指令，應該用 FORGET 指令。FORGET 利用 ' 指令找出我們要清除的指令，然後將詞典的指標設在此指令的名稱欄之處，同時將CONTEXT，CURRENT及各詞彙指令中有關連結地址的資料更動，使得查字典的動作由被清除的指令的前一個指令做爲起點。如此，我們就將要清除的指令，連同以後定義的所有指令都清除乾淨了。例如在多重程式的設計上：

```
FORGET OVERLAY : OVERLAY ;
```

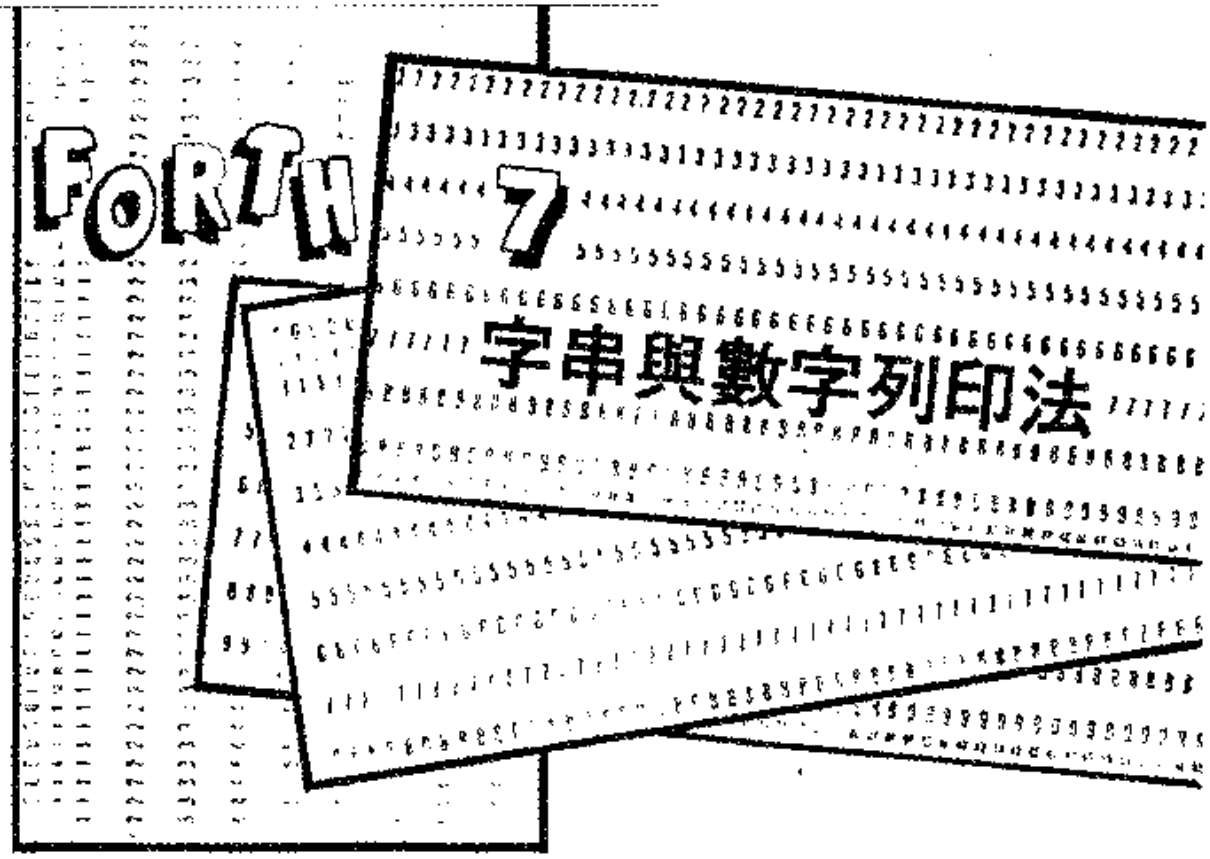
上述指令句即是將 OVERLAY 及其後的指令都清除，然後再定義一個掛名的 OVERLAY ，就可以繼續建造一個多重程序的結構了。

練習題六

1. 設計一套指令，可以由終端機接受一對數值並將其儲存在一個磁碟塊中。輸入的指令形式如下：

   300 400 ENTER

   數值現在可以存放的地址必須弄清楚。在ENTER以後請將這個地址印出來以供使用者參考，並決定它是否可以塞入更多的數據資料。
2. 設計一個指令 PRINT，可將上述的資料全部列印出來。列印的形式是每行8個數元。
3. 建立一個數陣表，列出整數中前面 n 個質數，$n \leq 10$。然後再寫一個指令印出這些質數陣的總和。



字串是一組字母符號，以ASCII 字母的方式存在電腦記憶中。字串是電腦輸入及輸出與人類交換信息的唯一方式，使用者以終端機的鍵盤輸入指令及資料給電腦，這些指令及資料是以字串方式輸入的，電腦必須將字串翻譯成爲它能執行的指令碼。當電腦要將結果告訴使用者時，也必須先將資料變成字串的型式，送到終端機的螢幕上或印在印表機上，方能爲使用者接受利用。

特別在輸出數值資料時，使用者常需要嚴格地控制數字字母列印的形式及位置。在FORTH系統中，使用者可以用字串組合的方法來規定數字的轉換及列印方式。因爲系統所使用的列印指令都能供給使用者應用，所以FORTH有很完整的數字輸出工具，適應不同程式的列印需求。

7-1 字串操作指令

表7-1中是一些基本的記憶區字串指令，可以設定字串或搬動字串

表 7-1 記憶區指令

指　令	疊 層 動 作 說 明	中文指令
CMOVE	(a1 a2 n —) 將 n 個字由地址 a1 搬到地址 a2 去。	搬字
FILL	(a n b —) 將 a 地址開始的 n 個字元定爲 b 值。	填字
ERASE	(a n —) 將 a 地址開始的 n 個字元清爲零。	清字
BLANKS	(a n —) 將 a 地址開始的 n 個字元清爲空格(32)。	空字
DUMP	(a n —) 將 a 地址開始的 n 個字元列印出來。	顯字

資料。將字串搬動的指令是CMOVE，它需要三個參數：原字串地址、搬動目的地址及字數。

addr1 addr2 n CMOVE　將 n 個字的字串 addr 1 地址搬到 addr 2 地址。如果 n 是零或負值，則不做任何搬動工作。

設定字串的基本指令是FILL，可以將一個字串設定爲一個指定的ASCII字母。ERASE 及BLANKS兩個指令則是將字串清除爲ASCII NUL 或空格(SP)：

addr n C FILL　將 n 個 ASCII 字母 C 填入由地址 addr 處開始的字串中。

addr n ERASE　將自 addr 開始的 n 個字元清爲零。

addr　n　BLANKS　　將自 addr 開始的 n 個字元清除爲空格（ASCII　32）。

將某一段記憶中的資料同時用 ASCII 及數字方式顯示在終端機上的指令是 DUMP 。這是檢查記憶中存放的資料最有效的工具：

addr　n　DUMP　　將由 addr 開始處之後的 n 個字元以字母及數字的方式表列在終端機上。

* Apple-FORTH顯示的方式是每行 8 個字元。

7-2　字串的暫存區

我們雖然可以將 ASCII 字碼存在疊堆上使用，若要處理許多字母的字串就很不方便了。在FORTH系統中，經常使用一個字串暫存區，它的地址是用 PAD 來取得的：

```
: PAD HERE 68 + ;
```

所以 PAD 暫存區是在詞典上字串暫存區（Word Buffer）之上的一個記憶區。PAD 暫存區的容量很大，通常有數千個字元，可以存很大量的文字資料。只是 PAD 是隨着詞典的增長而移動的。所以在 PAD 中的資料必須在定義新指令前使用，否則就不能確定原存的資料可以無誤地讀取回來。

EDITOR 使用 PAD暫存區來儲存輸入的字串，以進行文字編譯的工作。在數字輸出時，也是用 PAD 來儲存輸出的文字串。

7-3　字串的輸出指令

輸出一個 ASCII 字母到終端機螢幕或印表機的指令是 EMIT 。

EMIT取出疊層最上值，而將其最低的7位元當做一個ASCII字母輸出到終端機上。

```
65 EMIT A OK
```

65是ASCII A字母的代碼，EMIT將其印出。

輸出一整個字串的指令是TYPE。TYPE要兩個參數，一是字串在記憶中的地址，一是字串的長度或字數。如下面的指令句：

```
PAD 16 TYPE
```

將在PAD暫存區中16個字母印出來。

在輸出字串時，字尾巴上往往帶着許多空格。這些空格沒有必要輸出，特別是在用慢速印字機將字母列印的時候，印這些空格很是浪費時間－TRAILING指令是將這些空格除去的指令，它檢查一個字串的尾部，若有空格的時候即將要列印字串的字母數減少而避免了印空格的動作。假如在PAD中有HELLO這五個字母且有至少11個空格時，如果用PAD 16 TYPE的指令句時：

```
PAD 16 TYPE HELLO          OK
```

在HELLO後面將印出 11 個空格。但加上－TRAILING的指令，就變成：

```
PAD 16 -TRAILING TYPE HELLO OK
```

只印出5個字母。

如果在指令中需要列印一些字串以引起操作者的注意的話，有兩個方法可以使用。一是字串以．"×××"的方式包括在指令的定義中；另一是將字串存在一個磁碟暫存區中而用MESSAGE 指令將其印出。

. "text"　　"是一個FORTH指令，它會將其後隨的字串text（到下一個"字母爲止）列印出來。它對疊層沒有影響。

n MESSAGE　　將在第4個磁碟塊中第n行的文字列印出來。n以第4塊的第0行爲準，亦可爲負數。如果WARNING變數爲0時，MESSAGE僅印出MSG # n字樣而不印文字。

表7-2中也一併列出了字串指令的用法及功能。

表7-2　字串指令

指　　令	疊　層　動　作　說　明	中文指令
."XXX"	（－） 印出XXX字串。最後的"號是分界字母。	
TYPE	（a n－） 印出在地址a開始的n個字元。	印
－TRAILING	（a n1－a n2） 將在地址a的n1字元的字串後續的空格除去，將n1減少爲n2以便TYPE列印。	減空
MESSAGE	（n－） 印出在第4塊中第n行的文字。n可爲負數亦可大於15，此時即印4塊附近之文字。WARNING中存有0則只印出行數n，WARNING中存有1時則印出磁碟塊中文字。	信
PAD	（－a） 將字串暫存區的開始地址a放在疊層上，字串暫存區隨詞典之頂端漂移。輸出輸入之字串暫存此區，以備使用。	
COUNT	（a－a＋1　n） 取出地址a中字串的長度n放在疊層上，並將地址a加1。所得的結果可供TYPE印出此字串。	算

表 7-3 輸入輸出指令

指令	疊層動作說明	中文指令
KEY	(- c) 讀入按下鍵並將其代碼放在疊層上。	鍵
EMIT	(c -) 將疊層上的字母代碼送給終端機顯示。	放
EXPECT	(a n -) 由鍵盤輸入 n 個字母並將其存在地址 a 開始的記憶中。	望
QUERY	(-) 由鍵盤讀入一行(最多80個)字母，存入輸入暫存區。	讀
WORD	(c - a) 由輸入暫存區讀入一個字串並將其搬到詞典頂端。並將詞典頂端地址放在疊層上。C 是讀取字串的分界字母。	詞
TEXT	(c -) 由輸入暫存區讀入一個字串(以字母 c 做分界字母)，並將這字串存到PAD暫存區中。	文
CR	(-) 將列印指標移至下一行之首。	換行
SPACE	(-) 印出一個空格	空
SPACES	(n -) 印出 n 個空格。	空格
? TERMINAL	(- f) 若鍵盤上有鍵被按下，則將 1 放在疊層上。若沒有鍵被按，則放回零。	?鍵

7-4 字串輸入指令

輸入字母或字串及輸出它們的FORTH指令彙集在表7-2、表7-3中。最基本的字串輸入指令是KEY。KEY在執行的時候等待使用者按鍵盤上任何一個鍵。當一個鍵按下後，KEY即當此鍵的ASCII代碼堆上疊層。使用者可以由疊層上取得這個代碼而按其需要來處理(表7-4

表7-4 字母及其代碼

Char.	ASCII	Char.	ASCII	Char.	ASCII	Char.	ASCII
NUL	00	SP	20	@	40	`	60
SOH	01	!	21	A	41	a	61
STX	02	"	22	B	42	b	62
ETX	03	#	23	C	43	c	63
EOT	04	$	24	D	44	d	64
ENQ	05	%	25	E	45	e	65
ACK	06	&	26	F	46	f	66
BEL	07	'	27	G	47	g	67
BS	08	(	28	H	48	h	68
HT	09	)	29	I	49	i	69
LF	0A	*	2A	J	4A	j	6A
VT	0B	+	2B	K	4B	k	6B
FF	0C	,	2C	L	4C	l	6C
CR	0D	-	2D	M	4D	m	6D
SM	0E	.	2E	N	4E	n	6E
SI	0F	/	2F	O	4F	o	6F
DLE	10	0	30	P	50	p	70
DC1	11	1	31	Q	51	q	71
DC2	12	2	32	R	52	r	72
DC3	13	3	33	S	53	s	73
DC4	14	4	34	T	54	t	74
NAK	15	5	35	U	55	u	75
SYN	16	6	36	V	56	v	76
ETB	17	7	37	W	57	w	77
CAN	18	8	38	X	58	x	78
EM	19	9	39	Y	59	y	79
SUB	1A	:	3A	Z	5A	z	7A
ESC	1B	;	3B	[	5B	{	7B
FS	1C	<	3C	\	5C	|	7C
GS	1D	=	3D	]	5D	}	7D
RS	1E	>	3E	^	5E	~	7E
US	1F	?	3F	_	5F	DEL	7F

(RB)

)輸入；另一是把輸入的文字放在字串處理指令的後面一同輸入。前面的基本指令是EXPECT，後面的基本指令是WORD。

addr n EXPECT　等候使用者在鍵盤上按n次鍵，輸入n個字母，並將此字串存在addr的地址。

這樣可以將整個輸入的字串放在記憶中任意位置。但一般的FORTH系統都有一特定的區域專門儲存鍵盤輸入的文字供給文字解碼使用，這個區域的開始地址是存在系統變數的TIB中的。用此區域輸入字串的指令是QUERY：

```
: QUERY TIB @ 80 EXPECT 0 >IN ! ;
```

QUERY接受80個字母或是到CR前所有的字母，放到輸入暫存區中，並將文字解碼的字數指標 > IN 設定爲零，以便開始解碼的工作。

QUERY的用法可以下面的例子來解釋：

```
VARIABLE NAME 40 ALLOT
: NAMED QUERY TIB @ NAME 40 CMOVE ;
```

按入NAMED及Return之後可以開始按入40個字母鍵，將一個名字輸入到NAME的參數欄裡去。如果再定義

```
: WHO NAME 40 TYPE ;
```

按入WHO則可將方才字在NAME中的名字串印出來。

使用QUERY的問題是它一定要輸入40個字，如果沒有輸足40個字而以Return結束輸入字串時，字串以後到40個字的區域還會有以前在TIB中遺留的字母會一併被搬到NAME區中去。要更精確地控制字串的輸入就須要用到文字解碼的基本指令WORD及其相關的指令

TEXT了。

C WORD　將輸入暫字區中的一個字串（以C字母作爲分界字母）找出並搬到文字暫存區（Word Buffer）中去。Word Buffer 中預先都清除爲空格，WORD 並將文字暫存區的開始地址（HERE）放在疊層上。

上述的輸入暫存區（Input Buffer）一般是指鍵盤輸入暫存區（TIB），但也可以是一個磁碟暫存區。系統參數 BLK 中存的是磁碟塊的塊數，如果BLK是零的話，就用 TIB 作爲輸入暫存區。另一個系統變數 > IN 是指示現在處理過的字串尾的字母序數。所謂的下一個字串即是由 > IN 所指的字母往下去尋找的。

因爲WORD指令是FORTH系統中，文字解碼的重要指令，而文字暫存區（Word Buffer）也有其專有的用途，不宜做字串儲存的地方。所以在處理文字串時，通常用TEXT 指令將字串再搬到PAD中去使用：

C TEXT　用WORD將下一個以C字母分界的字串搬到PAD暫存區。在搬字串之前PAD起 64 個字都預先清除爲空格。

一般輸入整行文字的指令是 1 TEXT，因爲ASCII 1 是一個控制字碼，文字處理中不會用到。在輸入時以 Return 鍵作爲字串的終點。如果我們能選用合適的分界字母，通常可空格（ASCII 32）或逗點（ASCII 44），就能很自然地連續輸入好幾個字串。

繼續上面的例子，用TEXT來輸入字串：

```
: NAMED 1 TEXT PAD NAME 40 CMOVE ;
```

即將NAMED後面連下去的字串存到NAME 裡去，它們的用法是：

```
NAMED TRAVIS MC GEE
```

再按入

```
WHO TRAVIS MC GEE OK
```

即印出輸入的人名。

更複雜一些的輸入工作也可以很方便地解決。如果在上例中我們要分別輸入一個人的姓和名，可以用下列方法：

```
VARIABLE LAST 16 ALLOT
VARIABLE FIRST 12 ALLOT
: PUT PAD ROT CMOVE ;
: NAMED 44 TEXT LAST 16 PUT
        1 TEXT FIRST 12 PUT ;
: WHO FIRST 12 -TRAILING TYPE SPACE
      16 -TRAILING TYPE ;
```

此後我們可以按入下列語句輸入人名：

```
NAMED MC GEE , TRAUIS A ,
```

逗點即可用做姓與名的分界。用 WHO 印出時是照美國人的習慣先印名再印姓。

爲了實際操作者的方便，電腦在請他輸入字串的時候應當先列印一些簡單但是很容易了解的指示，操作者就可以在接受最少的訓練後來使用電腦系統做輸入的工作。上面的例子可以再加以變化：

```
: BACK 0 DO 8 EMIT LOOP ;
: GREET ."PLEASE TYPE YOUR NAME:"
  ."LAST,FIRST" 11 BACK
  QUERY NAMED ;
```

8 EMIT 是命令終端機將其字母指標往前挪動一格。BACK是要它挪動好幾格的指令。GREET中即用QUERY來輸入一行字，再交給NAMED把姓和名分解開來分別儲入LAST及FIRST的參數欄中去。

使用的時候，在執行GREET的指令時，終端機上會印出下列指示文字：

```
PLEASE TYPE YOUR NAME: LAST, FIRST
```

終端機的字母指標會挪到L字母之下，等待操作者鍵入他的名字，操作者按入他的名字：

```
PLEASE TYPE YOUR NAME: DOE, JOHN
```

後DOE即存入LAST，而JOHN存入FIRST中。

7-5 字串轉變爲數值

數值輸入最直接的方法是把數值鍵入，文字解碼在確定數字字串不是FORTH指令之後會自動將其轉變爲一個數值並將之堆上參層供後續指令使用。但也常有特殊的情形，數值是用文字串的方式輸入，而使用者必須自己將其轉換成爲數值。例如因磁帶機或是特殊儀器上讀入數字資料。

將數字字串轉換成爲數值的指令是NUMBER，它常與WORD連用將分割出來的字串轉換成爲雙整數：

addr NUMBER	把在 addr 地址開始的一個字串轉換成爲一個雙整數，並將這個雙整數留在疊層上，假若字串中有一個不是數字的字母時，系統會停止執行以後的指令而跳回文字解碼程式。

7-6 數字的列印指令

最基本的數字列印指令是D.R，其他的列印指令都是由D.R衍生出來的。

d n D.R	將雙整數 d 在 n 列欄中印出，右邊對齊第 n 列。
d D.	將雙整數 d 印出，右邊加一空格。
n1 n2 .R	將 n1 在 n2 欄中印出，右邊對齊第 n2 欄。
n .	將 n 印出，右邊留一空格。
addr ?	將 addr 中的數值印出，右邊留一空格。

這些指令可以供我們將整數數字很方便地印出，而且可以排列成爲整齊的表格。只是印出的數字只能呈現整數的形式，無法插入其他特殊的符號如小數點或逗點等。

7-7 特殊的數字列印法

在列印報表時，數字字串往往要經過程特殊的處理，插入不同的專用符號，例如電話號碼中的“－”號，日期中的“／”號及時分秒間的“：”等。FORTH系統中供給我們一套完整的數字列印的基本指令集。利用這一套指令，我們可以隨心所欲地組合數字字串，以印出合乎我們需要的數字資料。（請看表 7-5 數字轉換指令）

表7-5　數字轉換指令

指　　令	疊　層　動　作　說　明	中文指令
NUMBER	(a — d) 將地址 a 處的字串轉換成爲一雙整數。	數
CONVERT	(d1 a1 — d2 a2) 將d1處的字串轉換爲數值加在 d1 上變成d2 。a2是第一個不能轉換的字母。	換算
BASE	(— a) 存有數字轉換的基數的系統變數 。	基
HEX	(—) 將16存入BASE，開始用十六進位制轉換數字 。	十六進位
DECIMAL	(—) 將10存入BASE，開始用十進位制轉換數字 。	十進位
<#	<# 開始將數值轉換爲數字串的過程 。	轉換
#	(ud1 — ud2) 將ud1雙整個的下一位數字算出，商數ud2留在疊層上，數字加入輸出數字字串 。	位
#S	(ud — 0) 將 ud 全部換算至餘數爲零，算出數字加入輸出子串。	位數
HOLD	(c —) 將字母 c 加入輸出字串 。	含
SIGN	(n —) 若 n < 0 ，則在輸出字串上加一負號字母。	負號
#>	#> 棄去雙整數 d ，將輸出的數字字串地址 a 及字數 n 留在疊層上。	轉完

FORTH將內存數值轉換成數字字列是依照下列兩個規則進行的：

(1) 數字是由右開始轉換向左進行。

(2) 轉換的數值一定要是一個雙整數。如果要轉換單整數則可在疊層上多加一個零。

記住這兩個原則我們就可以討論數字轉換和列印的指令了。這些指令也收集在表7-5中。下面我們列舉了一些數值的處理工作，使它們成爲能夠列印的形式：

	列印的數值	在<# 前應該
(1)	16位元正整數	加0變成32位元雙正整數
(2)	15位元單整數	DUP ABS 0，將正負號存在疊層第三位以備SIGN用。
(3)	32位元雙正整數	不須任何處理。
(4)	31位元雙整數	SWAP OVER DABS，保存正負號。

注意在第(1)、(3)兩種情形疊層上只有一個雙整數或兩個數元，但在(2)、(4)兩種情形却有三個數元，因爲第三個數元要給SIGN來插入適當的負號。

下面就是一個將16位元正整數印在一對括弧中的方法：

```
HEX
: '(' 28 HOLD ;
: ')' 29 HOLD ; DECIMAL
: NO. DUP ABS 0 <# ')' #S '(' #> TYPE ;
```

'('及')'是在輸出的數字字串中插入左右括弧的指令。在NO.轉換及列印的指令中，DUP ABS 0多存一個數元以備SIGN輸出負號。<#開始轉換的程序，')'插入右括弧，#S將全部數值換算爲數字串，'('再插入左括弧。#>終止了轉換工作並將數字串交給TYPE

列印出來。

用SIGN指令可以把負號塞在數字串中，如果原數是負的話。例如

```
<# SIGN #S #>     可以將負號塞在字串右側
<# #S SIGN #>     則將負號塞在字串的左側
```

在一般應用中，負號應該在數字的左側。但在商業賬務中，有時應該把負號列在數字的右側。

在將數值轉換爲數字串時，BASE 中存的變數是用來做轉換的基數的。假如我們要把時間（以秒的單位）轉換而以 hh：mm：ss 的形式列印出來的話，可以用下列的指令：

```
HEX
: ':' 3A HOLD ; DECIMAL
: :00 # 6 BASE ! # ':' DECIMAL ;
: SEC <# :00 :00 # # #> TYPE SPACE ;
```

其中':'在數字串中插入一個冒號。:00 先以十進位法轉換一位數字，然後將基數換成6再轉換下一位數字，插入冒號後再將基數換成 10，回到十進位制，這樣我們是以 60 進位的方法轉換了兩位數字。最後 SEC 指令使用：00 兩次，轉換求出了秒和分的數目並將餘下的數字做爲時數。用這種方法很方便地把分、秒之值算出並列印出來。

以上的例子中，我們都用TYPE將轉換的結果印出來。假如我們經常要將轉換的結果送到終端機以外的輸出裝置中去，我們應該把組合數字串部份與列印的指令分開定義：

```
: (.) DOP ABS 0 <# #S SIGN #> ;
: . (.) TYPE SPACE
```

與這類似的是U.：

```
: (U.) <# #S #> ;
: U. (U.) TYPE SPACE ;
```

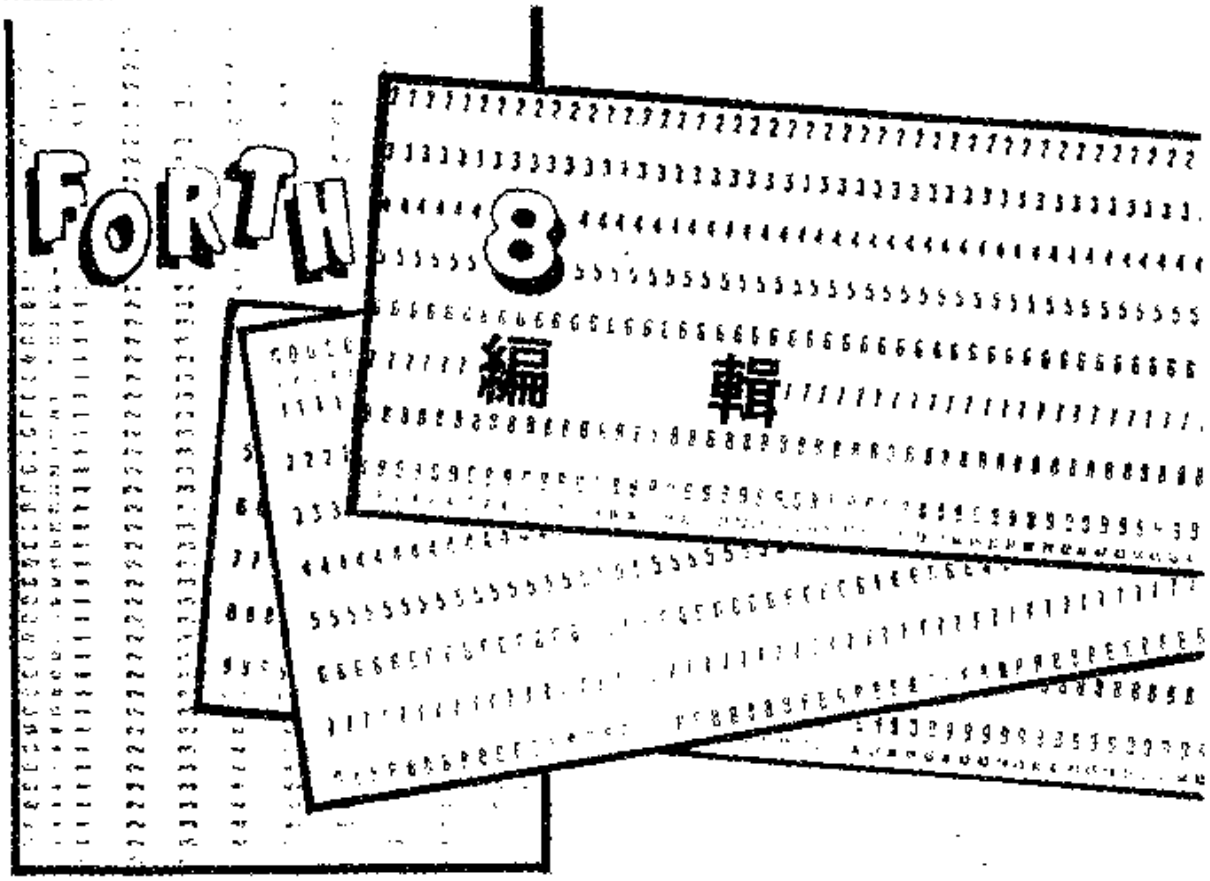
如果WRITE是將一些數字送去另外一個 ASCII 接收器，我們只要定義：

```
: .W ( .) WRITE ;
```

即可。SCR @ .W是一個很好的例子將SEC中存數送去 WRITE 的指令。

練習題七

1. 定義一個指令在終端機上印一段文字：HELP!
2. 定義一個BELL 指令會叫終端機的鈴響一次。
3. 定義一個指令將疊層頂上的一個 ASCII 字母用普通符號印出來。
4. 定義一個 32 位元雙整數的輸出指令：
   - 每個數字間有一個小數點。
   - 列印在一個垂直欄中。
   - 仿照社會安全號碼（××-××××-××××）列印。
5. 自行定義一個 INDEX指令，可以列印出連續一系列塊的塊數及其第0行的內容。如果一塊中第一個字元是0的話，這塊的第0行就不印。



8-1 程式的編輯

在解碼程式下我們可以隨時由鍵盤上按入程式建造新的指令。可是在建造好了指令，原來的程式文字就不見了，我們無法再將原程式文字叫回來做修改增刪的工作。編輯的指令允許我們將原程式文字永久儲存在磁碟中，隨時可以找出來執行，編譯成爲指令，或加以修改。

程式文字是以塊爲單位存在磁碟上的，在標準的FORTH系統中，每塊有1024個字母，分爲16行，每行64個字。在Apple-FORTH中因爲螢幕只能容納40個字一行，所以每塊縮成爲512個字母，32字一行共16行。塊在磁碟上是按順序排列的。各塊都有一個號數，我們按着塊號去檢索其中的資料。在Apple-FORTH中使用的虛磁碟，共有24K字元，分爲48塊，序號由0到47。

當開始做編輯工作時，我們必須先呼叫EDITOR。EDITOR是一個詞彙指令，執行完畢後，即將現用詞彙（Context Vocabulary）設定是編輯詞彙，這樣我們才能使用系統中有關編輯的指令。

選定一塊文字進行編輯工作的指令是LIST，按入

```
10 LIST
```

即選定第10塊來進行編輯，並將第10塊的文字列印在螢幕上。10也被存入系統變數 SCR 中，即是被設定爲現用塊，所有的編輯指令，都只改動這一塊中的文字資料。LIST也再選擇 EDITOR 做爲現用詞彙，在這使用上比較方便，不須要屢次按入EDITOR 指令。

選定了現用塊以後，要顯示其中的文字，只要按入：

```
L
```

指令即可。L 由 SCR 中取出塊的序號並用LIST將其印出。

8-2 行編輯指令

四個行編輯指令T，P，U及X是用來處理整行（32 字元）的文字資料，以下我們詳細解釋這四個指令的用法及其結果。

選擇第n 行使之成爲現用行的指令是：

```
n T
```

n 應由0至 15 ，指示現在編輯的行數，並將此行印出。同時將整行文字存在 PAD 開始的暫存區中。編輯指標（R＃中的值）也置於n 行文字第一字母之前。T經常是用來將指標搬動以便進行以下的編輯工作。

將一行文字寫入某一行的指令是P. 按入

```
P ××××
```

（××××代表一個字串，最多有32個有效字母）並按 Return 鍵後，字串××××即被塞在編輯指標目前所在的那一行中，蓋過此行中原

有的文字，××××並被存於 PAD 開始的暫存區中。如果P 後面緊跟着 Return ，則其間沒有任何字母時，P 卽將 PAD 中現存的文字搬到現在工作的那一行中。PAD 中的文字不變。如果P 後面有兩個空格才按Return時，PAD 與現用行中的文字都全部清除爲空格。

因爲P 是一個獨立的指令，所以它與字串××××間必須有一個空格作爲分界。第二個及以上的空格，則成爲字串××××的一部份而會被塞在文字塊中。總之，P 有三種不同的用法：

(1)　P　××××　　將××××置於現用行中。

(2)　P　　將PAD中的文字搬到現用行中。

(3)　P　␣␣　　將PAD 及現用行清除爲空格（␣表空格）。

將一行文字塞於現用行之下，而將後續各行往下各推一行的指令是

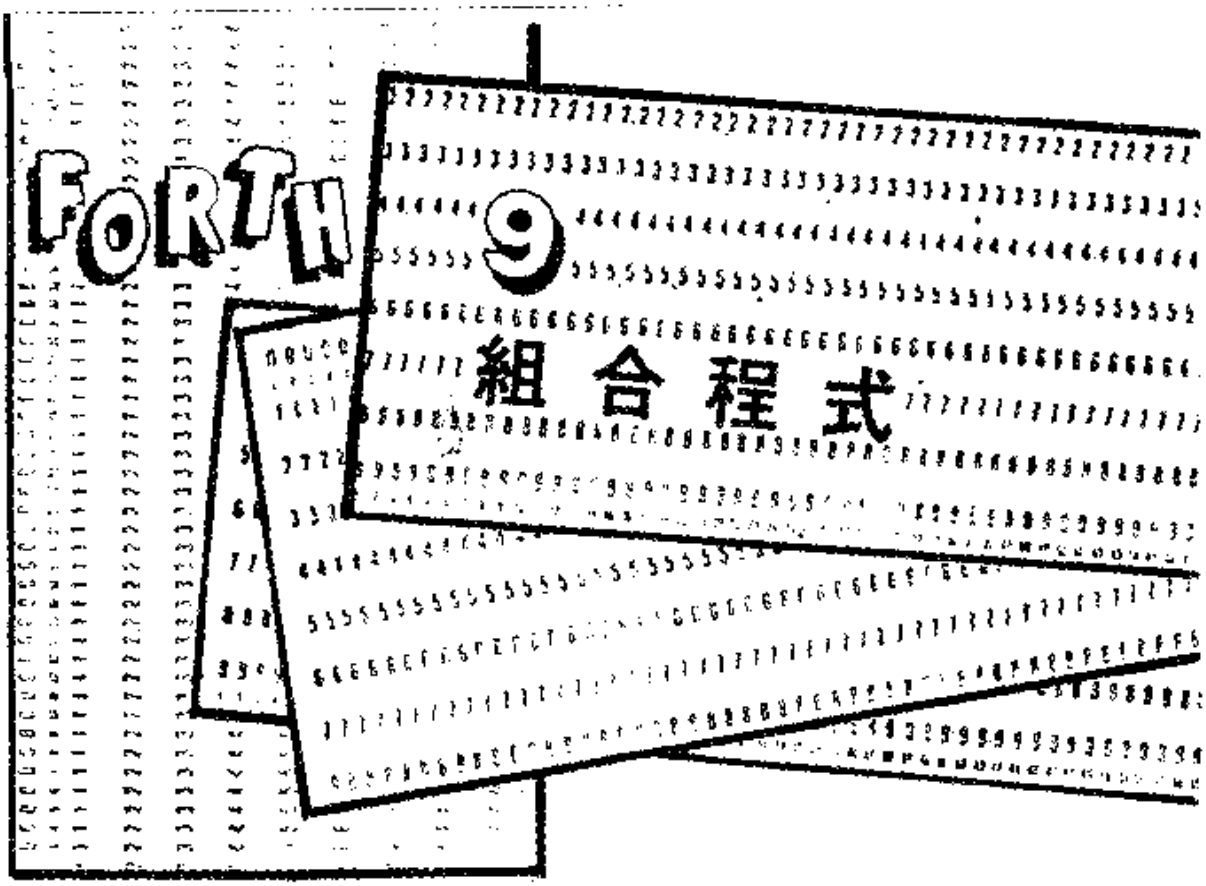
U　××××

第十五行中的文字卽被除去。U的用法和P 的用法相同。

(1)　U　××××　　將××××塞在現用行之下，以致各行均向下移動一行，第 15 行消失。

(2)　U　　將PAD 中的文字塞在現用行之後。

(3)　U␣␣　　將PAD 中的文字抄錄到現用行之下。



在 Apple-FORTH 中的組合程式是按照 William Ragsdale 在 Dr. Dobb's Journal 59 期（1981）中發表的程式製成的。這個程式在FORTH中只佔了一千多個字元，但是却具備了巨指令如 IF，UNTIL，地址計算，基數轉換，沒有指標（Label）的環路結構等特性。這個組合程式給使用者建造 6502 的低階指令，以備解決特殊的硬體界面問題或增加執行的速度。

建造低階指令的方法是用 CODE 在詞典中先造好一個指令的名稱欄、連結欄及解碼欄，然後再在參數欄中塞入適當的機器碼。例如 MON 這個指令讓系統跳到 Apple 的監督程式中去：

```
HEX
CODE MON XSAVE STX, FF69 JSR,
        XSAVE LDX, NEXT JMP, END-CODE
```

MON 執行時，先將數據疊層指標（在×暫存器中）存到XSAVE中去，再跳到監督程式的起點去。在回來之後恢復疊層指標，以NEXT JMP，結束此指令。

所有的FORTH低階指令中最後執行的機器碼都必須是NEXT JMP，或是其他可以回去執行下一個高階指令的地址如POP，POP-TWO，PUT，PUSH 等，使機器碼結束了現在的低階指令後繼續執行以後的指令，維持整個FORTH系統的正常操作。

9-1 6502的機器碼

在組合程式的詞彙中，主要的成員是6502的機器碼組合指令。簡單的6502機器碼指令在低階指令中變成一個字元的機器碼：

```
BRK,  CLC,  CLD,  CLI,  CLV,  DEX.
DEY,  INX,  INY,  NOP,  PHA,  PHP,
PLA,  PLP,  RTI,  RTS,  SEC,  SED,
SEI,  TAX,  TAY,  TSX,  TXS,  TXA,
TYA,
```

此外許多較複雜的指令會照着使用時的情況集合多個字元的機器碼：

```
ADC,  AND,  CMP,  EOR,  LDA,  ORA,
SBC,  STA,  ASL,  DEC,  INC,  LSR,
ROL,  ROR,  STX,  CPX,  CPY,  LDX,
LDY,  STY,  JSR,  JMP,  BIT,
```

如何決定用不同的選地址方法，是利用在疊層上預留的一些數值，如果疊層上沒有指令選地址的方式，就假設用的是第零頁上的絕對地址。表9-1中顯示各種不同選地址的方法及一些實例。

表9-1　機器碼選址方法

符號	選址法	參數	FORTH例子	一般組合指令
,A	累積器A	無	,A ROL,	ROL A
#	立即	字元	1 # LDY,	LDY #1
,X	X指標	零頁或絕對	DATA ,X STA,	STA DATA,X
,Y	Y指標	零頁或絕對	DATA ,Y CMP,	CMP DATA,Y
X)	間接X指標	零頁	6 X) ADC,	ADC (6,X)
)Y	Y指標間接	零頁	POINT)Y STA,	STA (POINT),Y
)	間接	絕對	VECTOR) JMP,	JMP (VECTOR)
無	記憶	零頁或絕對	NEXT JMP,	JMP NEXT

請注意選地址的指令也是FORTH指令，所以與其前後的指令必須以空格分開。指令所須用的地址及選地址資料都必須預先放在疊層上，所以組合指令也是使用後算符法來製作的。這樣可以最有效地使用組合的程式。

DATA及VECTOR都代表了實際的地址，在6 X) ADC，的例子中，就直接使用6爲參數，沒有將其定義爲一個有名稱的常數。

9-2　數據疊層的用法

數據疊層是放在第零頁由$ OFE開始向下延伸的。可以用零頁n，X的方法來取得其上的數據，X暫存器是專用爲疊層指標的，所以將X增加2即相當於由疊層上取出一個數元，將X減2則留一個數元的位置給新的數值。

因爲疊層頂值和其下的數值是最常用的兩個數值，所以有 BOT 及 SEC 兩個指令專用來選取這兩個數值：

```
BOT  LDA,  即是 LDA (0,X)
SEC  ADC,  即是 ADC (2,X)
```

BOT將0留在疊層上，並將選址法設定爲 ,X 法，SEC將2留在疊層上，也設定 ,X 選址法。

下面的例子是求疊層頂上四個字元的"或"值：

```
BOT      LDA,          LDA (0,X)
BOT 1+   ORA,          ORA (1,X)
SEC      ORA,  相當於  ORA (2,X)
SEC 1+   ORA,          ORA (3,X)
```

若要知道疊層上第14個字元的值，則用：

```
BOT 13 + LDA,
```

9-3 返回疊層

FORTH使用6502的疊層做完它的返回疊層，它是在第一頁記憶中，由$01FE向下延伸的。6502的S暫存器指向返回疊層上，空着可以使用的字元地址。

返回疊層上的值可以用PLA及PLP，兩指令取得。取出任意數值的正確方法是：

(1) 將X暫存在XSAVE中。

(2) 執行TSX，將S的內容搬入X暫存器。

(3) 用RP)來取得返回疊層上最低的一個數值，用,X的方式選取較高

的字元值。

(4) 將X暫存器還原。

下例是檢查返回疊層上第二個數值是否爲零的指令：

```
CODE IS-IT  ( -f )
    XSAVE STX,          ( 保存X )
    RP) 2+ LDA,
    RP) 3+ ORA,         ( 求或值 )
  0= IF, INY, THEN,     ( 若爲零則Y=1 )
   TYA, PHA,            ( 將Y值轉到疊層上 )
   XSAVE LDX,           ( X還原 )
   PUSH JMP,
   END-CODE
```

9-4 FORTH的暫存器

FORTH操作時使用的一些暫存器在低階指令中可以呼叫使用。其中有：

IP　解碼指標，指向NEXT下一次將執行的指令的地址。

W　指令指標，指向現在正執行時的指令的解碼欄。W－1中有$6C即是間接JMP的指令。因此W－1 JMP，即用以執行在W中的指令。

UP　系統變數區地址。

N　9位元的一個資料暫存區，在第零頁中。

9-5 6502的暫存器

當FORTH電腦執行NEXT JMP，而終止一個低階指令時，它並

完成了以下的工作：

(1) Y暫存器清除爲零，可以自由使用。

(2) X暫存器指示數據疊層最下一值的低位元，即是數據疊層指標。

(3) S暫存器指向返回疊層最下空值之處，爲返回疊層的指標。

(4) 累積器A可以自由使用。

(5) CPU是在二進位狀態，必須保持於此狀態中才能結束。

9-6 備用記憶區

FORTH還分出在第零頁中的一些記憶供使用者暫時存放必要的資料。

XSAVE是專供存放X暫存器之值用的：

```
CODE DEMO
    XSAVE STX, USER'S JSR,
    XSAVE LDX, NEXT JMP, END-CODE
```

USER'S 副程式會改變X值，所以在跳進去之前要將X中之值保存在XSAVE中，在結束之前還原之。

N指向在零頁中9個位元的暫存區，供給使用者一些零頁記憶做爲間接跳越指令的地址存放處。讀者可以參考CMOVE中N的使用方法。因爲許多FORTH指令在執行時都使用N區域，所以不可以用N區來傳遞資料，只能在一個指令中使用，例如：

```
CODE DEMO
    6 # LDA, N 1 - STA,        (計數器)
    BEGIN, CO81 BIT,           (輸出一次)
        N 1 - DEC,             (計數)
```

```
0< UNTIL,                              (循環)
NEXT JMP, END-CODE
```

SETUP是將一些疊層數值存入N暫存區以便使用的機器碼程式。N區可以存1到4個數元，數元的數目須先放在A累積器中：

```
3 # LDA, SETUP JSR,
```

將疊層頂上的3個數值移出而存在N區中。

9-7 流程的控制

FORTH揚棄了傳統的組合程式中的地址代名(Labels)。取而代之的是FORTH指令的名稱及在指令內部的結構。因爲FORTH的指令一旦加入詞典後，即可以其名稱呼叫出來執行或用來堆砌高階的指令；而指令內部的結構(Structures)可以利用結構化程式法的原理來建造分支或循環的結構。在低階指令內的結構建造的方法與在高階指令相似，Apple-FORTH的組合程式使用了下列指令：

```
BEGIN, UNTIL, IF, ELSE, THEN,
```

這些指令後面都跟了“，”符號，一則表示它們執行時會將一些機器碼塞入詞典，另一方面是要區別高階指令中的類似指令。

低階結構指令和高階結構指令有一個重要的不同處：高階指令用的測試旗號是疊層頂上的數值，而低階結構指令測試的是6502 CPU中的旗號位元，因此使用者在分支指令前必須指明他要測試的是那一個旗號。例如：

```
PORT LDA, 0= IF, …… THEN,
PORT LDA, 0= NOT IF, …… THEN,
```

在6502中可測試的旗號及其指令如下表：

指　　令	測試位元	測試條件
CS	進位旗號	C＝1
0<	負值旗號	N＝1
0=	零值旗號	Z＝1
CS NOT	進位旗號	C＝0
0< NOT	正值旗號	N＝0
0= NOT	零值旗號	Z＝0

用BEGIN，及UNTIL，可以建造一個有限循環：

```
6 # LDA, N STA,
BEGIN, PORT DEC,
     N DEC,
0= UNTIL,
```

用 IF， ELSE，及THEN，可以建造分支結構：

```
PORT LDA,
0= IF, N DEC,          （PORT爲零時執行之）
   ELSE, N INC,        （PORT不爲零時，執行之）
   THEN,    ……
```

結構可以套裝但不能重疊，下例即是一個錯誤的套裝：

```
BEGIN, PORT LDA,
     0= IF, BOT INC,
```

```
0= UNTIL,
   THEN,
```

UNTIL，必須結束外圈（由BEGIN 開始），所以不能放在THEN之前。

9-8 其他的結束指令

低階指令必須以NEXT JMP，結束其機器碼流程。如果指令結束時疊層上數值有增減變化，也可以跳至以下地址作爲結束：

POP	棄去疊層頂值
POPTWO	棄去疊層頂上兩個數值
PUSH	增加疊層的一個頂值
PUT	將疊層頂值改變

下面的例子是將疊層頂上地址所指之內容反轉：

```
CODE INVERT (adr —)
    BOT X) LDA, FF # EOR, BOT X) STA,
    POP JMP, END-CODE
```

下例是在疊層上添一個1的數值：

```
CODE ONE (— 1)
    DEX, DEX, 1 # LDA,
    BOT STA, BOT 1+ STY,
    NEXT JMP, END-CODE
```

用PUSH時就可以簡化如下：

```
CODE ONE (— 1)
    1 # LDA, PHA,
    TYA, PUSH JMP,
    END-CODE
```

FORTH的Assembler 因此也承繼着許多FORTH的特點，例如用後定位算符法來處理指令的參數。每個指令所用的Register 及選址用的參數都是放在指令代碼之前，這樣使用者可以更精確更直接地控制組合的過程。

一般程式問題都儘可能不用低階的機器碼來寫作指令而用高階的方法來發展，因爲高階指令比較容易寫作及測試。這樣可以很快地把程式的結構建立起來而且能很容易地測試完成。高階程式唯一的缺點就是在執行的時候做許多 Indirect Jump 及類似Subroutine Call 及 Return 等動作，而使得速度減慢。在FORTH系統中的對策就是在高階的系統發展完成之後再去測定其執行的速度。如果速度不夠快的話，使用者就必須做一些詳密的分析，檢查出執行過程中，循環數目最多的一些指令。這些指令可以挑出來，用機器碼來改寫，以增進其執行的速度。這種Optimizing 的方法可以很顯著地改進整個系統執行的速度。在這種修改的過程中，整個系統的結構不須要改變，因爲個別的指令或用低階或用高階，對系統的結構完全沒有影響。

用機器碼寫作低階程式，須要使用者對CPU 及硬體都有相當深刻的瞭解才行，FORTH也只能提供一些比傳統的組合語言較有利的工具而已。眞正問題的解決還是在使用者本身。在使用 Apple-FORTH 而設法用機器語之前，請讀者先研讀有關 6502 CPU 的結構，Apple Reference Manual 等硬體方面的參考資料，在FORTH部份則須前面提到Ragsdale 的 6502 Assemblero fig-FORTH Installation Manual中有極精彩、豐富的例子，值得讀者詳加研究的。

附錄一　表　圖

附錄二　圖　目

·全書完·