The Zen Of Forth

FORTH 的禪思

-- 人機合一的無上心法 --

符式如道　行之必效

柔以克剛　簡潔爲高

金城(愛新覺羅・燾昭) 編著

版權聲明

轉稿聲明

首先感謝金城老師願意將其著作授權作成電子書，再則感謝潘德喜先生謄打將原書轉爲電子稿，由王雨農轉爲PDF檔。

若需本書中所用之 F-PC 軟體，可至中華民國符式協會網站 download

中華民國符式語言協會
http://www.figtaiwan.org/

若有讀者想與金城老師交流，請E-Mial至cchin@pu.edu.tw

給讀者的話

『寫一本書，總是耗費心神的事兒！尤其是寫一本 Forth 的書更是傷神。第一版還是潘德喜硬磨著我寫完的，如今 FPC 早成了昨日黃花，Win32Forth 也進入了 6.11 版了。由 AppleII 的 Forth79 到 Windows 上的 Win32Forth 一轉眼 Forth 推廣至台灣也將近廿五年了，但台灣的 Forth 人口卻仍然稀少的類似即將滅絕的史前動物，有減無增，令人欷噓不已！

前兩年忙著與丁陳老師、陳爽大師兄，和阿江（陳昌江）成立『易符智慧科技』，想把 Forth SoC 與「易符學苑」，當成是人生中的最後一個志業來拼博，畢竟廿年前自己當過 FigTaiwan 的一任會長，還去了一趟上海交大開年會，向人吹噓了將近半輩子 Forth 的好處，若不能以身許道的投入身家來搞一次真的事業，此生豈不是令人訕笑。

丁陳老師要我將『Forth 的禪思』，使用 Win32Forth 當範例工具，再重寫一版。說實話我真的有點疲軟了，Forth 在 Windows Programming 雖然是個精巧曼妙的選擇，但想想 Windows 的作業環境有 Giga 的記憶體，依賴著 WDM 來做底層的 IO 管理，這樣不在乎系統資源的 PC 早已不是 Forth 這個秘密武器的戰場了。Forth 的交談開發過程，還是要在 embedded System 上才看的到優異性，在 SoC 的稀有資源裡會更有吸引力，這才是看 Forth 臉色吃穿的 Niche 所在啊！

不敢在這裡隨便的允諾『Forth 的禪思。第二版』的出版內容與發佈時間。畢竟除了動機之外，還有時間與心情都要恰當才是寫作的好當下，打坐修行多年了，我想這一切還是隨緣吧！喔對了，這本老著作在大陸的 www.ForthChina.com 網站上，有趙宇老弟親手改寫的簡體新版本，不過內部的範例程式改用他自個兒的 SystemForth 來說明，也是個值得瞅瞅的創見。

雨農老弟當年將舊版的 Forth 的禪思做成 PDF 電子書，我想這是好事兒，一來免得老有人寫信給我，問我要這本舊書，我自己早沒有存貨了；二來書在中國人的老派想法中，本來寫書嘛！也就是自個兒寫完了後，印幾本來送給老友當紀念用的。所以過往的中國社會裡，沒有所謂書籍買賣的版權問題！再說靠賣 Forth 的書，打死了都不會發財的，這一點兒，瞅著丁陳老師的見證，多年來早醒悟了，一笑！如今有人放在網路上流傳，博個浪得虛名，也算過了趟乾癮，就此知足了！若來日真有心思也許重新以『精簡的硬體架構 —— Forth 多堆疊的奧秘』爲主幹，來寫一本『Forth 津逮』，這幾年越來越覺得 Forth 像是電腦中的文言文，萬一真的想要寫一本 Forth 的新書，反正不會賣錢的，乾脆由著自個兒的性子，一不做二不休的寫一本『文言古體』的 Forth 書籍，一來賣弄賣弄自個兒的文學底子，二來考驗考驗讀者的古文程度，至少還有些野趣吧！

愛新覺羅　　燾昡　　辛巳年　仲夏

目　　錄

自　序(一) ---------- 01
自　序(二) ---------- 02
誌　謝 ---------- 03

第零篇　無上心法

FORTH 與禪 ---------- 0-1 ～ 0-6
無上心法第一篇 ----- 一把平凡的菜刀 ---------- 1-1 ～ 1-3
無上心法第二篇 ----- 如何在心中建一個電腦 --------- 2-1 ～ 2-3
無上心法第三篇 ----- 程式設計之道 ---------- 3-1 ～ 3-17

第壹篇　入門篇

第０課　FORTH 導論
一、安裝 F-PC ---------- 入 0-01
二、學習課文 ---------- 入 0-02
三、練習 ---------- 入 0-03
四、程式格式和程式助手 ---------- 入 0-05

第一課　印出字串
例題一、普通的問候 ---------- 入 1-01
例題二、大 F ---------- 入 1-02
例題三、我的名字 ---------- 入 1-03
例題四、在螢幕上顯示大的字元 ---------- 入 1-04
例題五、重覆的圖樣 ---------- 入 1-08
例題六、畫框框 ---------- 入 1-10

第二課　二個文字遊戲
例題一 ---------- 入 2-01
例題二 ---------- 入 2-06

第三課　數字
例題一 ---------- 入 3-01
例題二、溫度換算 ---------- 入 3-03
例題三、長方形 ---------- 入 3-06
例題四、天氣報告 ---------- 入 3-09
例題五、印出乘法表 ---------- 入 3-10

第四課　數的兩個範例
例題一、日曆 ---------- 入 4-01
例題二、生命的遊戲 ---------- 入 4-06

第五課　數字的技巧
例題一、正弦和餘弦 ---------- 入 5-01
例題二、亂數 ---------- 入 5-03
例題三、平方根 ---------- 入 5-04
例題四、最大公因數(GCD) ---------- 入 5-06
例題五、費邊數列 ---------- 入 5-08

第六課　終端機輸入和輸出
例題一、電話號碼 ---------- 入 6-02
例題二、時間輸出 ---------- 入 6-03
例題三、角度 ---------- 入 6-05
例題四、數字轉換之基數 ---------- 入 6-05
例題五、電報碼 ---------- 入 6-06
例題六、ASCII 字元表 ---------- 入 6-08
例題七、一封情書 ---------- 入 6-09
例題七、數字輸入 ---------- 入 6-11

第貳篇　基礎篇

第一課　FORTH 緒論
1.1　FORTH 概說 ---------- 基 1-01
1.2　FORTH 算術運算 ---------- 基 1-02
1.3　FORTH 算術運算指令 ---------- 基 1-04
1.4　堆疊(STACK)操作指令 ---------- 基 1-06
1.5　其他的一些 FORTH 常用字 ---------- 基 1-10
1.6　冒號的定義 ---------- 基 1-13
練習 ---------- 基 1-16

第二課　使用 F-PC
2.1　使用 SED 來編輯檔案 ---------- 基 2-01
2.2　載入並執行你的程式 ---------- 基 2-03
2.3　除去你程式中的錯誤(蟲) ---------- 基 2-04
練習 ---------- 基 2-06

第三課 FORTH 如何運作
3.1 變數 ---------- 基 3-01
3.2 變數的抓取(FETCH)與儲存(STORE) ---------- 基 3-03
3.3 常數(CONSTANTS) ---------- 基 3-06
3.4 FORTH 的高階程序 -- ： 的使用 ---------- 基 3-07
3.5 陣列(ARRAYS) ---------- 基 3-09
3.6 回返堆疊(RETURN STACK) ---------- 基 3-10
3.7 FORTH 的低階程序 -- CODE 的使用 ---------- 基 3-12
3.8 FORTH 的詞典(DICTIONARY) ---------- 基 3-14
3.9 表格(TABLES) ---------- 基 3-15
3.10 字母(CHARACTER)或位元資料 ---------- 基 3-16
3.11 詞典的搜尋方式 ---------- 基 3-17
3.12 名稱區的資料結構 ---------- 基 3-19
3.13 F-PC 內部的執行原理 ---------- 基 3-20
練習 ---------- 基 3-23

第四課 FORTH 判斷與迴路
4.1 分岐指令及迴路 ---------- 基 4-01
4.2 條件字組 ---------- 基 4-02
4.3 FORTH 邏輯運算指令 ---------- 基 4-04
4.4 IF 敘述 ---------- 基 4-05
4.5 DO 迴路 ---------- 基 4-06
4.6 UNTIL 迴路 ---------- 基 4-11
4.7 WHILE 迴路 ---------- 基 4-13
練習 ---------- 基 4-14

第五課 數字
5.1 雙倍精密度的數字 ---------- 基 5-01
5.2 雙倍精密度的比較指令 ---------- 基 5-04
5.3 乘和除 ---------- 基 5-05
5.4 底數的除法 ---------- 基 5-06
5.5 16-Bit 運算指令 ---------- 基 5-10
5.6 雙倍精密度數字的乘法 ---------- 基 5-12
練習 ---------- 基 5-13

第六課 字串
6.1 字串輸入 ---------- 基 6-01
6.2 ASCII--二進位位元轉換 ---------- 基 6-04
6.3 數字輸出轉換 ---------- 基 6-05

6.4 螢幕輸出 ---------- 基 6-07

第七課 低階(組合)字與 DOS 的輸入/輸出
7.1 低階(組合)(CODE)用字 ---------- 基 7-01
7.2 低階(CODE)的條件判別用字 ---------- 基 7-06
7.3 長距離的記憶體存取用字--超過 64k 外的存取 -- 基 7-07
7.4 DOS 的介面用詞 ---------- 基 7-08
7.5 基本的檔案讀／寫 ---------- 基 7-11
7.6 讀取數字和字串 ---------- 基 7-19
7.7 寫入數字和字串 ---------- 基 7-25

第八課 定義詞
8.1 CREATE...DOES> ---------- 基 8-01
8.2 簡易跳躍表格 ---------- 基 8-05
8.3 以堆疊上數值爲索引之跳躍表格 ---------- 基 8-07
8.4 使用 FORTH 詞之跳躍表格 ---------- 基 8-09
8.5 彈出式功能表 ---------- 基 8-11
8.6 練習 ---------- 基 8-21

第九課 編譯詞
9.1 編譯(COMPILING) 和直譯(INTERPRETING) ------ 基 9-01
9.2 一般詞編譯和立即詞編譯 ---------- 基 9-03
9.3 載入數字 ---------- 基 9-04
9.4 條件編譯詞 ---------- 基 9-07
BEGIN...WHILE...REPEAT ---------- 基 9-07
IF...ELSE...THEN ---------- 基 9-09
BEGIN...AGAIN ---------- 基 9-01
BEGIN...UNTIL ---------- 基 9-01
DO...LOOP ---------- 基 9-01
9.5 習題 ---------- 基 9-17

第十課 FORTH 的資料結構
10.1 陣列(ARRAYS) ---------- 基 10-01
10.2 鏈串列(LINKED LISTS) ---------- 基 10-04
10.3 記錄(RECORDS) ---------- 基 10-15

第十一課 使用中斷處理的終端機程式
11.1 8086/8088 中斷處理 ---------- 基 11-01
11.2 8250 非同步通訊用晶片 ---------- 基 11-03

11.3　佇列的資料結構 ---------- 基 11-05
11.4　輸出文字到螢幕和(或)磁碟機 ---------- 基 11-10
11.5　下載檔案 ---------- 基 11-12
11.6　終端機之主程式 ---------- 基 11-15

第參篇　附　錄

A. 程式之編譯 ---------- 附錄 A
B. LANGUAGE DEFINITION OF FORTH ---------- 附錄 B
C. STANDARD INSTRUCTIONS ---------- 附錄 C
D. USER INSTRUCTIONS ---------- 附錄 D
E. CREATING NEW DEFINING INSTRUCTIONS ---------- 附錄 E
F. F83 系統記憶圖 ---------- 附錄 F
G. F83 虛擬記憶磁碟緩衝區圖 ---------- 附錄 G
H. FORTH 的迴路 ---------- 附錄 H
I. Interpreter 的迴路 ---------- 附錄 I
J. Error 的迴路 ---------- 附錄 J
K. EXPECT 的迴路 ---------- 附錄 K
L. WORD 的迴路 ---------- 附錄 L
M. NUMBER 的迴路 ---------- 附錄 M
N. BLOCK 的迴路 ---------- 附錄 N
O. 以 FORTH 的觀點看 Intel80486 CPU ---------- 附錄 O
P. FORTH 資料庫程式設計 ---------- 附錄 P
Q. 一九九一年 FORTH 程序員水平考試試卷(大陸) ---- 附錄 Q
R. CHARLES MOORT　先生講　FORTH 來源譯文 ------- 附錄 R
S. FORTH 問題練習 ---------- 附錄 S
T. FORTH 參考書目 ---------- 附錄 T

(一)

當了一年 FORTH 學會的會長，在任內才了解 FIG (Forth Interesting Group)的精神。無花果，只求結果不求開花，默默耕耘的付出。並不期待燦爛的名與利；只希望能使更多的人了解 Forth 是什麼，而能經由學習 Forth 的過程，真的深入體會電腦這種工具的潛能與極限。 F-PC 是 Tom Zimer 先生的心血，其中包含了 Forth 的多工作業系統(Multitasking) ，組合語言的組譯程式(Assembler) 與反組譯程式(Dis-assembler)，全螢幕的編輔器(Screen Editor) 檔案管理程式，Forth 高、低階語言的除錯器(Debugger)、直譯器(Interpreter) 與最佳化的編譯器(Target compiler)，當然還有能自我編譯的介變編譯器(Meta Compilor)。這一切均在一個彩色的整合環境中，只佔一百多 K。比 Borlend C++ 小多了，也靈活多了。一如 Forth 的精神，無私無我，Tom Zimer 也將此套 F-PC 的全部原始程式附在磁片中公諸於世。不求金錢的償報只希望能讓更多的人，能籍由 F-PC 友善的線上求助環境(Online Help 按 F1) 而在最小的阻力下學習 Forth 。使用 Forth 。我知道國內想學 Forth 的人多半都被英文能力所困。所以，花了一年多的時間，將 F-PC 的自學手冊十一課中文化。希望能對英文不好的學員們有所助益。

自己也知道，寫 Forth 的書，不如寫 C 語言，BASIC 或 DBASE 的書來得輕鬆討好，且名利雙收。但，我不入地獄，誰入地獄。願能透過此書，普渡眾生。與讀者結個 Forth 的緣。

---------- 1992 年 11 月於靜宜資訊系 ----------

(二)

當第一版所有文稿都完成的時刻， 曾慶潭兄 又送來了 丁陳漢蓀博士的新書 FORTH first course 的原版，希望我改寫成中文，並加到第二版中。這下子肩頭剛卸下的重擔，又重新搭上了肩頭，經過了一番深思熟慮，我決定，重新改變本書原有的風格，一來，我添加了許多新的章節與補充了一些經過篩選的附錄，以滿足學員進一步的求知慾。二來，我將多年在 FORTH 教學上的心得，以更平實的筆法寫成一段段的註疏，加在各章節的段落之後，以發揮導讀的功能，而達到解惑的目的。三來，我嘗試在本書中加入一些人文色彩，以突顯 FORTH 的哲學思想，而讓學員能體受到，學 FORTH 的心歷路程，是精神重於型式的。否則何以了解" FORTH 如道 "。

寫一本好書很難，寫一本好的 FORTH 更難，我希望這一本花費兩年完成的書，能打破 FORTH 口傳心授的限制，使每一個學員都能沐浴在 FORTH 的春風裡。

---------- 1993 年 6 月於靜宜資訊系 ----------

誌　謝：

一、我要感謝三位，我的啓蒙恩師

Dr.Susan Marry ，如果不是她連續撕了五次我的程式作業，我不會知道自我要求是良好品質的唯一保證。

Dr.Tanenbaum ，如果不是他點通我，我不會體認到電腦的硬體與軟體是一體的兩面，自邏輯線路，微程式到作業系統人機介面，層層都是一貫相通彼此呼應的。

Dr.Goldenstein ，如果不是他逼我用 Forth ，我不會體會出程式設計與心靈同步的那一份自然融恰的美好感受，更無法了解，唯有人機合一才是與電腦相處共生的無上心法。

二、接著：我要感謝丁陳漢蓀博士；如果不是他醍醐灌頂的警示我 " 爲學日益，爲道日損 " ；我不會去看嚴靈峰先生的「老子集成」，而悟出 " 物壯則老 " 是 FORTH Meta Compliler 的精髓所在。也無從頓悟出 Forth 是一種生活的態度，一種充斥體內，無所不在的思維方式；所以，不具形不著相。其人機一體，任我翱翔。

三、當然這一本書，若不是潘德喜先生的策劃，推動打字、校正、排版則絕無機會完成。我在他的身上看到了那一股屬於 Forth 獨有的堅毅不拔。我要將此書的誕生與他分享。若此書能給台灣的電腦學習者帶來一點清新的朝露，則完全是他的功勞。

四、似乎要感謝的人太多了。如幫我校正的廖述昌、賴杰治同學，排版的馮棟煌先生，及給我鼓勵的人士們 ！ 謝 謝 ！

第零篇

無上心法

FORTH 與禪

前言

在某些狂熱者的眼中， FORTH 不只是一種電腦語言，有時它更像是一種宗教，受到支持者的崇拜與信仰。在眾多的宗教中， FORTH 與禪的境界最接近。如同大家所知，禪代表著精簡、雋永、智慧與啓發，同樣的，這些屬性也都可以用在 FORTH 的身上；除此之外，它們都象徵了對傳統的一種挑戰甚至是一場革命。FORTH 與禪之間的關係，一直無人將它做一有系統的介紹，而這正是本文的目的。

FORTH 與禪的比較

禪的代表人物是六祖慧能，他是中國禪宗的第六代傳人；目前風行的「六祖壇經」即是他與其弟子的對話錄。與佛教經典相比，六祖壇經顯得較爲精簡、易懂、含蓄，所以也更爲耐人尋味。吾等認爲比較 FORTH 與禪之間關係最好的方法，就是將一段壇經原文與一段 FORTH 的原始碼並列，它們之間或許沒有直接相關，但只要用心體會，一定能感覺出彼此所共同透露出的神韻。

禪與 FORTH 的歷史

禪在本質上融合了儒、道、釋三家的精髓。中國人的心中常會存在著一種矛盾：在繁褥的儒家、虛無的道家和一個進口的宗教「佛」之間，總是不知如何取捨。於是經過五代相傳，禪宗終於在六祖慧能的手上大放光采，攫取了眾多中國人的心靈。日後，這股禪的熱潮更推向了日本、韓國、東南亞，以至於到今日的西歐與美國。

跟禪宗比起來， FORTH 的歷史幾乎微不足道。它是由一個孤獨的科學家 Charles Moore 在六十年代末期發明的。說他孤獨是由於他的思想、方法都被排除在當時電腦科學的「主流」之外。在最初的十年， FORTH 幾乎無人知曉，一直到了 1985 年之後，它語法精簡、結構優雅、記憶體使用十分經濟，諸如此類的優點才漸漸爲人所發掘，進而廣爲人知。

口傳心授的禪

佛教是釋迦牟尼所創，他強調人人只要去除私慾，則都能夠成佛，二百年後，佛教分裂成二派：大乘與小乘。大乘佛教後來傳入了中國、韓國及日本。而小乘佛教則傳入了錫蘭、泰國與緬甸。大乘佛教西元二百年傳入中國後，受到人們的歡迎，更傳入了大量的佛經。

禪則是在西元 527 年由印度僧人達摩祖師悟達的。傳說他在少林寺面壁九

年，才悟出了禪的道理。但是他既不引用佛經、也不自行著作，只用口頭說明來啓發學生。於是禪就以口述的方式，代代相傳；相對於佛教的大建廟寺、廣譯群經，禪卻以口傳口、心通心的方式流傳下去。

口傳心授的 FORTH

FORTH 是在 1970 年由 Charles Moore 所發明的。 FORTH 是 FOURTH 的縮寫，即第四代語言，有別於當時已大放異彩的第三代語言。FORTH 的程式碼非常難懂，因爲它的編譯器也是用 FORTH 寫的。要了解 FORTH，必須先了解它的編譯器，要了解它的編譯器，卻又必須先了解 FORTH。FORTH 這種「易學難精」的特性，使得許多人望之卻步。多年來， FORTH 也只能以口傳口、心通心的方式在某些擁護者之間流傳。未能普及大眾。

禪的發揚

六祖慧能是禪宗的關鍵人物，這位不識字的天才，原本是個樵夫。後來他投入了五祖弘忍的門下學習禪宗。弘忍即將圓寂之前，集合了門下弟子，要他們各寫一首偈，弘忍再根據弟子們的慧根，決定將衣缽傳給誰。

當時神秀(弘忍的得意門生)寫下了這首偈

身是菩提樹，心如明鏡臺。
朝朝勤拂拭，莫使惹塵埃。

而慧能則口述請人代寫了這首偈：

菩提本無樹，明鏡亦非臺。
本來無一物，何處惹塵埃?

根據偈中的意境與啓發，弘忍將衣缽傳給了慧能。慧能成爲宗師後，只做口述傳教而不著作經典。大約西元六百年，一位巡撫請慧能的高足法海，根據慧能的言教，寫下了六祖壇經一書，書中表達了禪真正的想法與境界，更使一般人能輕易地了解禪的意境；六祖壇經的完成，使得禪爲大家所熟知、意會，進而使禪發揚光大。

FORTH 的發揚

正如前述的種種障礙， FORTH 始終只是少數人的流傳之物。直到 1978 年 FORTH 推廣學會(FORTH INTEREST GROUP)的成立，他們針對市面上最風行的六種微處理機，用組合語言撰寫了 FORTH 系統，使一般電腦使用者也能了解

FORTH 的程式碼，進而使用它。1980 年之後，更出現了大量的 FORTH 相關書籍，如 LEO BRODIE 的 STARTING FORTH，丁陳漢蓀博士的 SYSTEM GRUIDE TO FIGFORTH ，及 BYTE 雜誌的 FORTH 特別報導；這些都扮演了類似「六祖壇經」的角色，使一般使用者都能接觸 FORTH ，使用 FORTH ，進而了解「FORTH 如何完成工作」，而非「FORTH 能做那些工作」。這些努力使 FORTH 不再是一個神秘難解的語言， FORTH 變得大眾化、親和化、乃至發揚光大。

禪的單純性

經過長久時間的流傳，佛教變得非常複雜。有太多的佛經註解、有太多高僧的偈言，此外它更融入了其他宗教的教義(如十八層地獄、輪迴觀念即來自印度教)。久而久之，信徒們往往只拾取他們相信的部份：如素食、佈施、放生等。相對的，禪就單純了很多，慧能並不相信實行這些作爲就真能成佛。他認爲佛已存在於每個人的心中，只要去除私慾雜念，就能成佛。所以禪宗不重修行、不依賴佛經反而重視個人的心靈啓發，比佛教要單純了許多。

FORTH 的單純性

爲了突顯 FORTH 的單純，我們可以把前面神秀、慧能的偈做些修改。當今的電腦觀念；就如同神秀的偈：

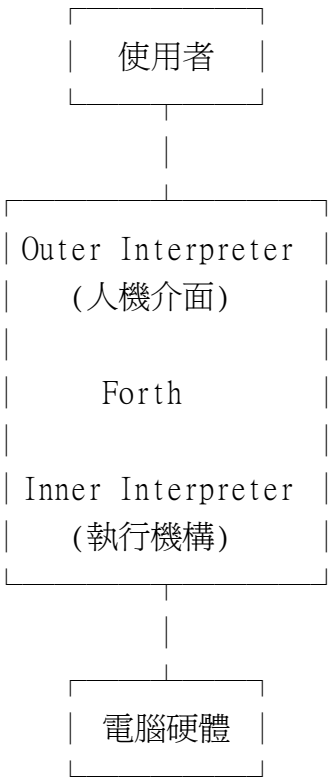
硬體雜又難，軟體學不完。
日夜埋首幹，心酸又心煩。

而 FORTH 呢? 就像：

硬體具實觀，程式爲思串，
符式由中穿，人機原一貫。

現今的電腦科技，軟硬體都相當複雜。主要是由於人與機器之間，隔了太多軟體。「以軟體保護硬體」這觀念源自於過去大電腦的時代，當時電腦當機是最可怕的災難，所以專家們設計了重重關卡(如 O.S LOADER DRIVER 等)來「幫助」使用者使用電腦設備。而這些複雜的軟體，卻造成了使用者學習上莫大的困難。(如圖一)

然而現今的電腦已不再是那麼回事，按一下 RESET 即可再起動，更何況使用者往往就是電腦的擁有者，他會爲他的行爲負責。所以實在無需再將人機之間加諸種種限制。FORTH 即遵循這個觀念：基本上， FORTH 是一個單純、整合的介面，透過它使用者能夠輕鬆地取用任何硬體資源，更由於單純化、最佳化，所以速度更快、記憶體更省、效率更高。(如圖二)



此外 FORTH 還有幾個特性，現今電腦中過多的暫存器(Register) 往往造成呼叫程式時需要重覆存取的困擾。而 FORTH 採用高效率的記憶體存取技巧，且將一些需暫存資料都放入堆疊(Stack)中，大大提升了效率。而後置式的運算表示法，雖然會造成某些初學者的不習慣，但其精簡只需要 1-pass 的編譯，而傳統的中置式則通常需 2-pass 的編譯。FORTH 的精簡單純，正是造成它效能驚人的最大原因。

禪的啓發性

如前所述，禪與佛教在某些地方有所不同，佛教講求自身的修行與經典的研習。更由於經文大多由梵文翻譯，所以更造成了理解上的困難與教義的眾說紛云。禪宗則是重視心靈的修爲，它認爲真正的啓發決不是來自書上，而是來自心靈的交流。禪對於心靈啓發的方式，又分爲兩派，神秀主張「漸悟」，而慧能則主張「頓悟」。但正如慧能所闡述的，啓發的方式並不重要，重要的是如何除去外在的阻礙，以達到心靈的啓發。

FORTH 的啓發性

談到 FORTH 的啓發性，指的應該是使用者對電腦硬體的完全了解與掌握。當今的作業系統(O.S) 與程式語言(P.L) 實際上限制了使用者，讓他們只能做系統允許他們做的事。而使用者一旦透過 FORTH 掌握了電腦，他們會發現他們才真正的成爲電腦的主人，而電腦不過是個有威力、卻很溫順的奴隸罷了。

FORTH 分爲兩個部份：內部的執行機構(INNER INTERPRETER) 與外部的執行機構(OUTER INTERPERTER) ，內部的執行機構(INNER INTERPERTER) 用來實際執行 FORTH 的指令群駕御硬體，而外部的執行機構(OUTER INTERPERTER) 則能把使用者輸入的文句轉換成 FORTH 碼。所以使用者一旦將 FORTH 弄熟了，就不用再受到作業系統(O.S)與程式語言的先天限制而成爲電腦真正的主人。

金城 註疏：

在 1993 年的 9 月 FORTH 已成爲 IEEE 的開放式開機韌體的唯一標準。如果你使用 SUN 的工作站於任何時間按 STOP-A 就可進入 FORTH 的環境中任你遨遊。

無上心法第一篇

" 一把平凡的菜刀 "

記得剛出國求學的時候，看見老外的廚房裏琳琅滿目，各式各樣的鍋、碗、盤、盆、刀叉瓢羹，覺得文明的國家連廚房都是那麼的多采多姿，引人入勝。尤其是刀架上，七、八種各式各樣的刀，閃耀動人，有切肉刀、切魚刀、切麵包刀、切火雞刀、切洋蔥刀、切水果刀、切葡萄柚刀，砍骨頭的刀。有平滑刀、有鋸齒刀.....。哇!嚇死人了。要做一餐飯，需要四、五種盆，五、六種鋸，七、八種刀。弄完一餐飯要洗堆積如山的各種用具。最難過的是花了如此多的時間，使用了數十種工具。但做出來的西餐並不一定精緻，也不一定討人喜歡。

反觀：我們中國大廚的手上，只有一把平平凡凡的大菜刀，但熟能生巧，劈、砍、切、削、剁、片、拍、剃.....。能將雞鴨魚肉蔬菜水果處理的有條不紊，恰到好處。一個又黑又醜的炒菜鍋，在火候控制下能煎、煮、炒、炸、燒、烤、焗....樣樣能幹，做一桌菜，一刀一鍋一鏟，如此而矣！中國菜名聞四海。憑的是什麼，秘密在那裏！你知道嗎？老祖宗的無上心法只是 " 一把平凡的菜刀 "。" 簡單 " 二字而已！因爲！簡單的菜刀使廚師能夠在反覆使用的熟練中，控制自如；由熟透工具而產生技巧。再由熟悉了技巧而能專心的思考做菜的問題。能想到什麼菜式，就運用熟練的技巧將之表現出來，而達到人刀一體的完美境界。

再回頭看老外的廚子。那麼多種工具，要用一生的時間去學去記。活到老學到老。那有時間去專心的想做菜的問題。相同的，我們來看現今學電腦的問題也是犯了同一個錯誤；走進電腦書店，各式各樣的書籍，逛逛電子街，各式各樣的軟硬體，叫人眼花撩亂，不知所措。 DOS、UNIX、OS/2、Window-NT、DBASE、FOXBASE、Clipper、Lotus、Excel、C++、PASCAL、BASIC、COBOL、FORTRAN、Assembley，嚇死人了。成千上萬的東西要學，還沒學會這一版，下一版又出來了。每個人的硬碟由數十 Mega，換成了數百 Mega 仍然不夠用。書架上的手冊陳列了數十本，但沒有一本看完。學了數年的電腦，但仍尋尋覓覓的在找新的軟體和新的書籍。活到老，學到老，就是無能力解決問題。或每遇到一個新問題，就要學一個新工具，看一本新祕笈。唉！孰不知練招不練功，到頭一場空。花了大量的金錢，耗上了青春歲月。唯一的收穫，只是知道了幾個流行軟體的名字，和一台只耗電不賺錢的電腦標本。書架上一本本 " ＸＸ入門 " ，" ＸＸ速成 " ，" ＸＸ大全 "，" ＸＸ寶典 "都成了丟了可惜，留著沒用的負擔。可憐！可嘆！可悲矣！

反觀！學 Forth ，用 Forth 的電腦專家(Computer Hacker)，窮其一生，

只鑽研 Forth 的書籍，只使用 Forth 一套軟體，(Forth 也有自己的 CPU 和硬體)，當然容易熟能生巧，而後專心思考工作上的問題。要做資料庫，就建立一群專門搜尋、排序、讀寫檔案、記錄的指令集。要做電腦繪圖，就建立幾個畫點、畫線、畫圓、塗色的指令。要做電腦音樂，就寫幾個控制、音高、音長、音強、音色的指令。要做自動控制就建立幾個時序中斷的輸入輸出指令。覺得他人的文書編輯軟體不好用，就依自己的喜好寫一個自己順手的編輯程式，又好用，又不需要看手冊，反正，每一個功能都是自己定的！

看 DOS 不順眼；又覺得 UNIX 太龐大，那就用 Poly-Forth 嘛！只佔 8K 的記憶體，又是多人使用的多工即時作業系統。甚至仍覺的不爽那就自己重寫一套 Forth ，反正有自我介變編譯器(Meta Compilor)就算重建一次也不過幾天的功夫與時間！

一個真正的 Forth 高手，一定會做各種應用的軟硬體。更會做系統工具如組譯程式(Assembler)，編譯程式(Compilor)和編輯程式(Editer)。因爲，這些都是 Forth 系統的一部份！聽起來好似很誇張，但確一點不假。隨便任何一個 Forth 高手，都會同意這樣的說法。

可是其他系統或語言的高手，絕不敢有這一種想法。試問 DOS 的高手有幾個敢重寫 DOS 。C 語言的高手有幾個敢重寫 C 語言的編譯程式。PE2 的高手是否敢重寫 PE2；中文輸入法的高手有幾個敢重新發明更好的中文輸入法。但每一個 Forth 的高手，都敢而且都會重寫一套自己的 Forth 系統。這幾乎已成了 Forth 的傳統鐵律。是什麼使 Forth 的高手與其他系統或語言的高手，有如此大的差別，就是" 簡單 "二個字的威力。

因爲 Forth 系統就只有兩個堆疊、一個字典，如此簡單的結構，非常容易精深。一旦熟悉之後，便能依個人或應用的需求而產生各種變化，以擴充原有舊環境的能力。也就是建立新的指令或讓舊指令有新的功能。這就是 FORTH 的應用方式。一種簡單而有效率的方式。常聽學生們聊天，覺得當今學電腦的人中毒很深。老是迷信 " 速成 " 的流行趨勢，而忽略了根基紮實的重要性。

" 解決問題的是人，不是工具 " 。使菜變得可口好吃的是廚師，不是廚師手中的刀。學 Forth ，一如學用刀，該看成手段，而非目的。真正的要義是：學好內功，只需一招 " 人機一體 "，便能專心面對問題，思考問題的特性與通則，思考問題所需要的資料結構，思考問題所需要的演譯法(Algrithm)。而 " 對電腦與問題的深入了解 " 是克敵致勝的第一要素。

而 Forth 的簡易性與擴充性，和模組(module)的再用性(reusable)只是使人機一體的你，能更精熟與自信的去迅速解決問題罷了。反過來說，正因爲學 Forth 的路途中，有組合語言、有作業系統、有編譯程式、有直譯程式、有資料結構(主記憶體的管理).....。所以，如果能專心的熬過這一場毅力與堅持的考驗自然成爲人機合一的高手。一如能在廚房熬成大廚一般。刀功、火候的磨鍊缺一不可。

否則，半途而廢，不如不學，徒增自艾自怨罷了。還不如，死心塌地當一個平凡的使用者，會用用試算表、文書處理、套裝軟體就罷了。不用吃苦受難

也不用白忙一場。說實話、看的太多了，反正這世上絕大多數的人學電腦都是學心安的半調子。前幾年流行 BASIC ，就一窩風的學 BASIC ；這幾年流行 C 就一股學 C 的浪潮要把人淹死。最近流行物件導向(OOP)的 C++又是阿貓阿狗的惡叫惡吠一陣，天曉得，明後年又要搞什麼飛機了。真奇怪！追的不煩、學的不累啊！怎麼這麼多年了都沒人弄清楚。" 是廚師對做菜的配方了解，加上刀工和火候的掌握使中國菜揚名四海，而不是金刀、銀刀的威力或神蹟使然 " 切記！切記！

無上心法第二篇

" 如何在心中建一個電腦 "

在電腦這個世界裏打滾十多年了，常常被人問道：「老師！爲什麼你對電腦玩得那麼精深，從硬體的邏輯玩到 CPU 的設計從系統的 O/S、Compiler 搞到電動玩具的音樂，畫面控制。一下子寫 Editor 一下子寫 Fractal 的程式都那麼順手，早上寫組合，下午寫 C++ ，晚上寫 PROLOG 都不會弄亂.....。」

其實！我並不是電腦的天才高手，而是 Forth 的十項全能，使一個這樣平凡的我，擁有了一次那麼亮麗的人生展現。還記得在十幾歲的時候，酷愛寫詩塡詞，彈琴譜曲，對電腦是一點概念都沒有的電腦文盲。直到有一天在電子街看到 APPLE2 ，一下子被那個小機器所吸引住。(那時還不懂什麼是 8 位元的 CPU)。但是十幾歲的孩子，那裏買得起數萬元的東西(那時一般公務員才賺幾千塊月薪)。於是自己慢慢的存，存一點錢買一塊空的主機板。再存一點錢買幾顆小 IC ，再存一點錢買一個鍵盤.....買一個螢光幕.....。就這樣花了一年多，才買全了零件。剛開始用鉻鐵銲零件，也是步步艱辛，一點一點慢慢的銲，遇到不懂的，就去圖書館借電子學和數位電子學的書來看。又花了半年，才在螢光幕看到 OK 。那一天，我哭了一場。從什麼都不懂到裝出一台自己的電腦，我花了近兩年的光陰和熬了上百個夜。

而這似乎只是一場惡夢的開始。因爲這只是硬體部份而已！接下來是學軟體了，沒錢買軟式磁碟機(一台一兩萬)，只好把花了一夜寫好的 BASIC，用學生情人卡式錄音機錄下來，以便隔日除錯(Debug)，半年過去了。學來學去都煩了，看來看去都是那幾本書；沒有什麼新鮮的了，這時有人告訴我在整數版的 BASIC ROM 中有一種叫組合語言的東西。當天狠下心來換了一套 ROM ，開始 " K " 組合，才知道原來 CPU 中有暫存器、累加器.....。又過了半年才把 APPLE2 的監督程式 "K" 完。這時已近二十歲了。在多少深夜的苦讀中，一本本資料結構，計算機組織，邏輯設計，週邊設備介面學，作業系統、編譯程式、電腦圖學、系統程式、程式語言、演釋法、人工智慧、平行處理、工程數學、離散數學、機率論、集合論、拓樸.....伴我熬到天明、在學校中、離群寡居、在老外的眼中，似乎我這個老中是沒有黑夜的讀書鬼，白活了，大學也白唸了。

說真的，那時我書讀的太雜亂！沒有自己的思維系統！只要教授提一本書，我就 "K"一書本，弄得己很累很慘！所幸的是！申請到了研究所的獎學金之後便到紐約讀研究所去了。但自己知道，做學問和讀死書是不同的，我很害怕，怕拼不過老外會失去獎學金。尤其在程式作業被 Dr.Suson 當面撕碎時！我想大概什麼都完了。她要我重寫一份程式，我送去又被撕掉了，我只好又寫了一份，來來回回繳到第六份才被接受；她之後問我是否了解 " 自己可以要求自

己，爲什麼還需要別人來要求你 "之意，我終於了解，她的用心是在砥礪學生們 " 自我要求是良好品質的唯一保證 "。

在修進階計算機組織時，Dr.Tanenboum 問我，電腦和人有何異同之處？我說了一大堆，他卻難以接受的搖搖頭。叫我回去自己想清楚。只提示我電腦的 " 軟硬體 " 一如人的 " 靈與肉 "。

我回去一直想" 靈與肉 "，才體悟了人與電腦原是一體的兩面。人有記憶體，而電腦有 ROM 和 RAM ；人有筆記本而電腦有磁碟機，在上課時人抄筆記，就似電腦存檔一樣，不怕忘記。考試前翻看筆記就似電腦開檔讀進 RAM 中一樣，可以恢復記憶的內容。人有筆，電腦有印表機，人有往來禮儀，電腦有握手協定；人有電話，電腦有網路；人有大腦思維，電腦有 CPU 運作。人要學習正確的知識才能明辨事非，電腦需要明晰的指令才能解決問題.....。到那一天我才了解，學電腦一點都不呆板一點都不陌生。因爲電腦只是將人的" 靈與肉 " 數位化了而已。只要用心探索人類自我的 " 靈與肉 "，自然就會了解電腦的 " 軟硬體 "。

例如：人類有法律制度，電腦就有作業系統，人類有翻譯，電腦有編譯程式，人類由細胞組成，電腦有邏輯元件各司其職。人類有私有財產，電腦有區域變數。人類有交通幹道，電腦有匯流排。幾乎電腦上所有的事物與觀念都與人類的世界相互呼應。其實看清楚，摸透了，就不覺得艱澀難懂了。在研究所的教授中 Dr.Goldon-Stein 是最叫我憶念的了，他在課堂中老是叼著一隻煙斗，在上程式語言時，他告訴我，每一類電腦語言都是一種心靈思維的表達方式。不同的語言有不同的寫作風格，有不同的美學標準和不同的條律規範。

Modula2：嚴謹明確、優雅動人、似貴婦人一般。

LISP：簡潔流暢，層層相關。像言行一致的君子。

APL：纖細動人，玲瓏巧妙宛如淘氣的小丫頭討人喜歡卻難以捉摸。

BASIC：和善易處，寬大不拘小節，一如好客的村姑野叟，淺謔愉快輕鬆，但不足以托負重責大任。

C：靈活衝動，如千里之駒，若能駕御則神兵利器，否則野馬脫韁，主人被困，非死即傷。

FORTRAN：古樸溫潤，雄深雅健自有俠客之大家風範.....。

如此數例，可以得知選擇電腦語言，一如尋求婚配，要深入了解對方之家庭背景，生活習性才能手到擒來，事半功信，更要和自己的個性處得來，能截長補短互信互助，才能事事順遂，否則數日一吵幾日一鬧，內亂紛擾，未能克敵致勝，已吐血身亡。出師未捷身先死，常使英雄淚滿襟、情何以堪，心何以甘。所以，千萬不能找錯對象，算準成婚；適合他人的，不見得適合你。流行的趨勢不是人人都能穿著得體好看。最後談談 Forth 吧？

Forth 似形而上的宗教信仰" 只傳心法，不立文字 " 得多得少、是成是敗；不問過往根業，只看福緣深淺。一如老子之道，微妙玄通，深邃廣遠，大方無隅，大象無形。亦如禪宗法門，簡要不繁，各人體用不一。唯有一心一念不亂，日夜參悟精進方能登門入室，練成無上心法，其化境無形無象，飛花摘葉

皆能克敵致勝，談笑間用兵千里，何需刀劍之累。若能功德圓滿，出關之日，再回顧 Forth 自身之章法文字也覺得皮囊而已。學 Forth 一如天蟬百變，吃桑脫皮，苦痛中成長，做繭自縛死後重生，才能化羽高飛。所以需要一抹虔誠，方能堅信不疑，否則學之初雄心萬丈信誓旦旦，一但挫折"潰不成軍，半路退兵 "。更需要一股愚公移山的傻勁，方能精誠所至金石爲開。我相信任何一個人機一體的 Forth 高手，都曾體受過那份痴傻背後不爲人所知的辛酸。

我在剛學 Forth 時，也是日日怨，夜夜愁，直到一年後發覺自己在寫程式時，口中順嘴說出的就是程式的句子，心中凝想著兩個堆疊的上下起伏毫無半點刻意，是下意識的自然反應。那一夜，我感到一股從未有過的輕鬆與自信，我笑了，因爲我知道我再也不用爲電腦的工作感到壓力和恐懼了。因爲我心中已成了一部 Forth 的電腦，我的惡夢終於消失了，不覺莞爾。

無上心法第三篇

程式設計之道

金城 註疏：

此篇節錄自 Geoffrey James 在 1986 年的著作
The Tao of Programming 此書有許多地方的神韻與 FORTH 十分相似，在紛亂的資訊世界中注入了一股清流。作者以老子爲背景，敘說程式設計之「道」。該書薄薄的僅有 150 頁但句句禪思扣人心弦，引人步入對創作程式的深省。這麼多年來，程式設計到底是「工程」還是「藝術」，在此看看大師的見解，竟是濃郁的人文色彩。相信您若是愈資深的程式設計師，愈能體會此節中的精髓。一窺堂奧。若是初學，或學究，則必然不信半字。 FORTH 是一種求「道」的過程。讀者何妨虛心觀之全神體悟，必有所獲。每節之後的註疏，只代表我個人的淺見，並非對原文的添加。

第一部　靜寂虛無篇

程式員大師(以下簡稱大師)如是說：「學會了從程式抓蟲子之後，就可以畢業了。」

金城 註疏：畢業是指就能夠在外面混口飯吃了。

1.1 節

靜寂虛無中有奧秘，不靜不動，乃是程式之源，吾無以名之，故稱之爲程式設計之道。若道至大，則作業系統至大；若作業系統至大，編譯程式亦然；若編譯程式至大，應用程式亦復如是。是故使用人大悅，世有和諧存焉。

1.2 節

程式設計之道無遠弗屆，隨晨曦微風而返。
道生機器語言，機器語言生組譯程式。
組譯程式生編譯程式，於是萬餘語言存焉。
各語言有其目的，均表達軟體之陰陽；其在道中亦各得其所。
但若能避免，就不要用 COBOL 寫程式。

金城 註疏：軟體之陰陽，乃指解決問題之方式就是演算法(Algorithm)

1.3 節

太初有道，道生時空，故時空乃程式設計之陰陽。
程式員不悟道則時空永不敷使用，悟道者恆有充分時空完成目標。

金城 註疏：
時空者，「時」指 CPU 的能力與速度，「空」指記憶體之規劃與運用。

1.4 節

上智程式員聞道而行之，中智程式員聞道而求之，下愚程式員聞道而笑之；
若無笑聲則無道矣。
至高之聲難以聽聞；
前進就是後退之路；大智總是晚成；每一個完美的程式仍然有 BUG。
道在所有知識之外。

金城 註疏：知識指資訊科學。

第二部　古之大師篇

大師如是說：「三日不寫程式生命無趣。」

2.1 節

古程式員神秘而深奧，無以度量其思維，僅能描述其表象。

像狐狸涉水般地小心；像戰場老兵般地警覺；像未經琢磨的木頭般地拙撲；像洞中深潭地不透明。

誰能指出他任心靈中的秘密?

答案全在道中。

金城 註疏：

在經驗中我們終於了解程式設計師的本能，大多數是天生的而很難以教育來訓練成才。

2.2 節

大師 Turing 曾經夢到他是一部電腦，醒後道：「不知是我 Turing 作夢變成機器，還是一部機器作夢變成我 Turing。」

一家大電腦公司的程式員參加軟體會議後，向他的經理報告說：「你知道其它電腦公司有些什麼程式員嗎? 」他們不修邊幅，頭髮長而邋遢，衣服既舊且皺，他們破壞了氣氛，而且在我簡報時老是製造噪音。」

經理說：「我根本就不應該派你參加會議，這些程式設計師超然物外，他們把生命看成無稽、意外的結合。他們往來而無藩籬，爲他們的程式而活，爲什麼他們一定要受社會積習的約束?

他們活在道中。

2.3 節

生手問大師：「有一個程式員從不設計、測試程式、寫作文獻，但了解他的人都認爲他是世間最好的程式員，爲什麼? 」

大師曰：「這個程式員已充分悟道，他超越了設計的需要；系統垮了不會生氣，而無條件接受這個世界。他超越了文獻的需要，他不再計較是否有人看他的程式。他也超越了測試的需要，他的每一個程式都圓滿無缺、清徹、優雅、目的自明。

真的，他已經悟道，登堂入室。」

第三部　設計篇

大師如是說：「到測試程式時再回頭修改設計就太遲了。」

金城 註疏：

沒有規劃過的程式就似危險的違章建築一樣，隨時會倒、會塌、會

垮。

3.1 節

曾經有人在參觀電腦展每天進門時都向警衛說：「我是個妙賊，偷東西的技巧已臻化境，先告訴你，我絕不會放過這次展覽。」

這段話刺激到警衛，因爲展覽場有好幾百萬元價值的儀器，所以老是盯著他，不過卻看到這個人一個攤位接一個攤位看，哼著小曲而已。

這個人出門的時候，警衛把他帶到一邊搜身，但卻找不到什麼。

第二天這個人又來了，而且教訓警衛說：「昨天我收穫不錯，不過今天會更佳。」所以警衛就又更加注意他了，但是仍然沒有結果。

最後一天警衛終於忍不住好奇心，問那個人：「賊大師，我給您弄得寢食難安，您是否以教我，究竟偷了些什麼？」

這個人笑笑，說：「我偷的是概念。」

金城 註疏：先有概念(也就是靈感)才有創作。

3.2 節

從前有一位大師專寫沒有結構化的程式，一個生手模仿他，也開始寫沒有結構化的程式。當這位生手要求大師評量進展時，大師卻批評他寫作沒有結構化的程式，大師說：「對大師適用的不一定適合生手，在能夠超越結構化之前，必須先悟道。」

金城 註疏：

使用 GOTO 是一種兩極化的現象，高手用之可以提昇程式的執行效率。低手用之，則程式的結構破壞，難以維護。 Knuth 與 Diiestia 曾爲了 GOTO 大打一場筆戰，也是主因。

3.3 節

某長官問程式員：「設計會計系統與作業系統，那一個比較簡單？」

程式員說：「作業系統。」

長官發出不相信的驚呼！「很顯然地，會計系統不如作業系統複雜」。

他說：「不！」程式員回答，「在設計會計系統時，程式員是各種有不同主意的人之間的橋樑，這些主意不外乎：系統要如何作業？報表型式如何？要如何迎合稅法？------等等。反過來，作業系統卻不受外界表象的限制；在設計作業系統，程式員尋求人與機器之間最純的和諧，這就是爲什麼作業系容易設計。」

長官點頭微笑稱是：「但是，那一個容易偵錯？」

程式員沒有回答。

金城 註疏：

有人爲因素的程式最難下手，也最難評估好壞，人比機器要複雜的多，要翻臉也容易的多。

3.4 節

經理去見大師(程式員)，並且告訴他一套新應用程式文件的需求規格，問道：「如果我給你五個程式員，要多久才能設計好這個系統？」

大師很快回答：「一年。」

「但是我們需要馬上用這個系統！如果我給你十個程式員，那要多久？」，經理說。

大師皺眉說：「這要兩年。」

「如果我給你一百個程式員呢？」

大師聳聳肩：「這個系統根本做不出來了。」

金城 註疏：

人愈多則人與人之間的溝通便愈複雜，而需要開會妥協的地方也愈多，這些都會破壞系統的簡明性，君不見 Ada 與 PL/1 的鐵證如山。

第四部　寫作篇

大師如是說：「寫作良好的程式本身自成天堂，寫得差的程式自身就是地獄。」

4.1 節

程式要輕靈，副程式像一串珍珠。程式的精神與意圖應始終如一，不多不少；沒有多餘的迴圈，也沒有額外的變數，既不缺少結構，也不過份笨重。

程式應該追隨「最低驚訝定律」，這是什麼? 簡單得很，使用人對程式的反應是驚訝的機會要愈低愈好。

程式不管再複雜，應該以一個整體來作用；它應該用內部邏輯，而不是外在的表象來指導作業。

如果程式不滿足這些要求，就會雜亂而易生混淆，唯一的補救就是重新寫過。

金城 註疏：補丁滿佈的程式不會好看，更不會耐用。

4.2 節

生手問大師：「我有一個程式，有時候做得很好，有時候卻不行；我一直遵行程式設計的規律，但是卻把我弄得很困擾，其理安在? 」

大師答曰：「因爲不悟道才會如此，只有笨蛋才會期望他的同儕有合理的行爲，而你卻對人類生產的機器有所期望 !? 計算機只模擬了決定論，只有道才十全十美。

程式設計的準則還是暫時性的，只有道才會進入永恆。所以你在開竅前必須思索道。」

「但我要如何才能知道已經開竅了呢? 」生手問。

大師回答:「從此之後，你的程式都能夠正確執行。」

金城 註疏：不能正常作業的程式，定是處處例外，缺少一致的通則。

4.3 節

大師對弟子說:「不論軟體之爲大或爲小，道在所有軟體中。」

「桌上型計算機有道嗎? 」弟子問。

「有! 」大師答。

「電動玩具程式中有道嗎? 」弟子繼續問。

「也有! 」大師說。

「那麼個人電腦的 DOS 中有道嗎? 」

大師咳一下，輕輕挪動了位置，「下課」，他說。

金城 註疏：
DOS 是有名的急就章的產物，其蟲之多由 1.0 至 6.0 算不清了。

4.4 節

皇太子的程式員正在寫作軟體，指尖在鍵盤上飛舞程式順暢無誤地編譯完成，執行起來像陣微風輕拂而完美地結束。

「了不起! 」，太子嘆曰，「你的技巧無懈可擊。」

「技巧? 」程式員從終端機上轉過頭說，「我所從的是道，道超越任何技巧! 我開始學寫程式時，在我眼睛所見是混成一片的程式；三年後，不再見到這一

大片程式了，我學會使用副程式；現在眼前一片空靈，什麼都沒有了，所有東西都進入無型式的一片靜寂；所有視覺都不必作用。」

「我的精神可依直覺而不必依任何計劃行事，換言之，我的程式自己寫作自己。當然，有時會有困難的問題；我看著它們到來，我降低自己的速度，靜靜地看著改一列程式之後困難就會煙消雲散；我再重新靜靜坐著欣賞工作的歡樂。我閉上雙眼一會兒，然後關機。」

皇太子說：「我的所有式員都那麼聰明睿智嗎？」

金城 註疏：

學電腦也好，學語言也好，都只是手段而非目的。最重要的是把自己訓練成一個完整的邏輯思考動物。進而成爲人機合一的電腦高手。若心中無點墨，腦中全漿糊，則任憑您努力一生一世，仍無法進入「邏輯」的世界，因「道」的第一階段就是「邏輯」嘛!

第五部　維護

大師如是說：「雖然程式只有三列，但總有一天需要維護。」

5.1 節

常用的門不必上油。
急流不會淤塞。
聲音與思想不能在真空中傳遞。
不用的軟體會生銹。
這就是至大的奧秘。

金城 註疏：

程式設計師的血液中天生著追求完美的本質。每經過一段時間，就會對原來的作品產生新的構想，而產生了更改的念頭與動手的慾望。

5.2 節

經理問程式員究竟要多久才能把手上的程式寫完。「明天」，程式員很快的回

答。

經理說：「我想你不太踏實；真的要多久？」

程式員想一會兒：「我希望在程式中加上一些東西，這至少要兩週。」程式員終於說。

「時間還是長了一些」，經理堅持說：「如果你能簡單地告訴我什麼時能寫完我才會滿意」。

程式員同意這一點。

幾年後經理退休了，在歡送餐會上發現那個程式員伏在終端機上睡著了，因爲他寫程式寫了整夜。

5.3 節

一個生手被分派去寫一個單純的財務軟體。

這個生手狂熱地做了幾天，但是當大師看他的成品時，卻發現這個程式中包含一個螢光幕編修程式、一組一般性的繪圖程式、一個人工智慧界面，但卻沒有什麼與財務方面有關。

大師就問他，這個生手卻變得很激動：「不要那麼沒有耐心」，他說：「我最終會把財務部份加上去。」

5.4 節

好農夫會忽視他種的穀子嗎?
好老師會忽略他最差的學生嗎?
好父親會容許他的孩子挨餓嗎?
好程式員會拒絕維護自己的程式嗎?

金城 註疏：

一篇沒有主題的文章寫的再美好那只是雕砌文字，但仍是廢話連篇，
一個好的程式若沒有要點缺乏「原創力」，也是徒勞無功。

第六部

大師如是說：「程式員要多，經理要少，生產力就會增加。」

6.1 節

經理有開不完的會的話，程式員就會寫電玩；主計部門想到利潤，發展經費就會被刪減；高級科學家談到藍藍青天，那麼青天一定會有浮雲飛過。

當然，這不是程式設計之道。

當經理許下承諾，程式員就不理會電玩；當主計部門有長程規劃，就會回復和諧與秩序；當高級科學家處理手上的問題，問題很快就會解決。

這才是是程式設計之道。

6.2 節

爲什麼程式員沒有生產力？因爲他們的時間都花在開會上頭。
爲什麼程式員難以駕御？因爲管理階層干預太多。
爲什麼程式員一個接一個辭職？因爲他們精力耗光了。
在不良管理下工作，程式員不會覺得他的工作有價值。

金城 註疏：
程式設計師的工作是「創造」不是「製造」。而「創造」是藝術的本質。

6.3 節

某個經理快被炒魷魚了，但是他底下的一個程式員寫成了一個叫好又叫座的程式；當然，這位經理因而保住了飯碗。

經理打算給這位程式員一點獎勵，但他拒絕接受，並且說：「因爲我覺得這是個有趣的概念，才會寫這個程式，所以我不希望有獎勵。」

經理聽了之後說：「這個程式員雖然職位不高，但卻充分了解做爲一個職員的責任，讓我們把他升成崇高的管理顧問吧！」

在告訴程式員時，他再度拒絕，說：「我之存在是因爲可以寫程式，如果升了我，那除了浪費每一個人的時間外而成不了事。我可走了嗎? 我還得寫程式。」

金城 註疏：

在台灣這個社會中有個怪現象，人人想當經理，總經理。好似做程式設計師沒什麼出息，別忘了，在一百年中世上可以產生許多個總統，但不見得能出一個達文西啊！

6.4 節

經理告訴程式員們說：「下面是你們的工作時間；早上九點鐘來上班，下午五點鐘下班」。所有程式員都很生氣，有幾個馬上離職。

於是經理說：「好吧！」這樣好了，只要能夠如期完工，工作時間由你們自定」。程式員現在滿意了，每天中午開始工作，直到第二天早上。

金城 註疏：

創造性的工作需要靈感，所以沒有靈感的時間，不宜工作，只宜睡覺。

第七部　公司智慧

大師如是說：「你可以對主管示範一個程式，但無法讓他通曉電腦。」

金城 註疏：買畫的人不一定懂畫，更不用談畫畫了。

7.1 節

生手問大師：「遙遠東方有一個叫做"公司總部"的偉大樹狀結構，上面滿滿地標上了些副總裁、會計長等等的圖案。它發出大量的備忘錄，每張上面都寫了「收文！」、「發文！」，沒有人知道這是什麼意義。每年都會把新的名字加到新的分枝上，但似乎全都徒勞無功。爲什麼這樣一個不自然的組織還能繼續

存在？」

大師回答說：「你已經體認到這個龐大的結構，而被它不合理的目的困擾。不過你能不從它無休止的迴旋而得到樂趣嗎？能夠不欣賞深藏在枝葉底端毫無困難的程式設計嗎？爲什麼要被它的無用而困擾呢？

金城 註疏：不要淘醉在東家長，李家短的小道消息裡。

7.2 節

東方海上有大魚曰鯤，鯤能變成雙翼遮天的大鵬。當大鵬飛越陸地時帶來一道公司總部的訊息，這道訊息正好掉在一群程式員中央，然後大鵬折起雙翼乘風而歸。

生手程式員瞪眼望著大鵬，因爲他們不認得；中智程式員憂慮大鵬的來臨，因爲他們害怕它帶來的訊息；只有大師才能繼續坐在終端機前工作，因爲他不知大鵬的來去。

金城 註疏：

「悟」，需要靜，「靜」，需要定，智慧的最高境界便是一心不亂。

7.3 節

象牙塔的魔術師帶著他的最新發明去見大師，他推了一個大黑盒子走進大師的辦公室，大師正在靜靜地等著。

「這是一套整合性、分散式、一般用途的工作站」，魔術師如是說，「還有一套專屬的作業系統，第六代語言，多項最先進的使用人界面，再加上人體工學的設計；這花了我的助手們好幾百人年才造出來的，不是很了不起嗎？」

大師抬了下眼珠子，「的確了不起」，大師說。

魔術師繼續說：「公司總部已經下令每個人都要用這台工作站發展新軟體的基石，您同意嗎？」

「當然」，大師答，「我馬上會把它放到資訊中心去。」。於是魔術師高高興興地回到象牙塔去。

幾天後，一個生手在大師的辦公室裡團團轉，說：「我找不到新程式的報表，您知道會在那兒嗎？」

「當然」，大師答道，「報表就堆在資訊中心裡頭的基石上！」

金城 註疏：追求時髦容易迷失，不能解決問題，則工具只是廢物。

7.4 節

大師可以毫無憂慮地從這個程式轉入另一個程式，管理上的改變傷不到他；縱使計劃終止了，也不會被炒魷魚。爲什麼？因爲他充滿了道。

金城 註疏：

一百個傻瓜，只會留下一百個錯誤，一個智者，便能解決一個真正的問題。在公司中真正解決問題的人是稀有動物，而其餘的人能不製造麻煩就是萬幸了，留下來只是浪費金錢(薪水)。

第八部　硬體與軟體

大師如是說：「沒有風，草不會動，沒有軟體，硬體就是廢物。」

8.1 節

生手問大師：「我知道有一家電腦公司比其他的大得多，高高在上就像是巨人之比侏儒；它的任一部門都可以單獨成爲一個企業。爲什麼會這樣？」

大師回答：「你爲什麼問這個笨問題？這家公司就是因爲它大才會這麼大。如果它只製造硬體，沒有人會買它；如果只生產軟體，沒有人會用它；如果只維護系統，人家會把它看成修理員；但是因爲它把所有的合在一起，人們就把它當神一樣看待了。它根本無需競爭，因爲贏來不費吹灰之力。」

金城 註疏：

在電腦世界中，如果說藍色的巨人就是指 IBM。

8.2 節

大師有一天經過一個生手身邊，發現生手迷上了一台手掌型的電玩，「對不起」，大師說，「我可以看看它嗎? 」

生手停下來，並且把這台機器交給大師。大師說：「我看到這台機器玩起來有三個層次：初級、中級、高級；不過這種機器通常都有另一個層次的說法，使機器贏不了人類，而人類也勝不了機器。」

「啊! 大師，生手說：「這個奇妙的開關在那裡? 」

大師把機器摔到地上，用腳把它踩爛。

突然地，生手開竅了。

金城 註疏：

在電腦式電玩的世界中，使用者所面臨的好像只是冰冷無情的電子機器，但事實上，真正的靈魂是程式設計師所精心創作出來的程式在執行。所以，偉大之處在於程式設計師的智慧結晶，而非沒電就完了的硬體設備。所以請不要迷信電腦萬能。

8.3 節

從前有一位微電腦的程式員對一位來拜訪他的大型電腦程式員說：「你看，在這兒有多好! 我有我自己的作業系統與檔案儲存設備，我不必與任何人共用任何電腦資源；軟體本身自給自足而且容易使用。爲什麼你不辭掉目前的工作來參加我們? 」

於是大型電腦的程式員就對他的朋友解釋：「大型電腦就像古之聖哲般地穩穩座落在資訊中心中央，磁碟一個接一個蔚爲奇觀，軟體像鑽石般地有多種面目，像古森林般地濃密茂盛。各個程式像一片急流般地湧入系統，而這就是我在那兒工作的樂趣。」

聽了這段話之後，微電腦程式員靜默無聲：但是這兩個人卻結爲好友，至死不渝。

金城 註疏：

無論電腦架構之大小如何，其觀念一致。程式設計師之本領高低，不是看使用的設備而論，你看看馬路上開車的人，開 Benz 不一定比開計程車的司機技術更好。

8.4 節

Hardware 與 Software 走在路上， Software 說：「你是陰我是陽，如果我們能一條心，一定會成大名賺大錢。」所以，他們就聯合在一起而想征服世界。

走了一段路之後，碰到 Firmware ，穿得破破爛爛，拿著根枴杖，並且對他們說：「道在陰陽之外，靜寂不動如古井之不生波瀾；道不求名，故無人知曉其存在；道不逐利，因它圓滿無缺。道超乎時空之外。」

Hardware 與 Software 聽了之後倍感慚愧而打道回家。

金城 註疏：

Firmware(韌體)，通常指在 CPU 內部的微指令，負責機器指令的抓取--解碼--執行，因技巧的深度，一般程度的軟硬體工程師皆不明瞭此一微結構階層之運作方式。而此部份，被喻爲電腦中最深奧巧妙的一部份。因爲它是 CPU 的生命核心。

第九部　尾聲

大師如是說：「這是下課的時候了！」

第壹篇

入門篇

第零課　FORTH 導　論

F-PC 是 FORTH 系統開放給 MS-DOS 的領域，相容於個人電腦，是由 Tom Zimmer 和 Dr. Robert L.Smith 和許多 Forth 程式設計師設計的，它是配合 MS-DOS 操作系統的程式環境，利用操作系統的助益和基本的 80*86 機器，它有強力的編輯器和 8086 編譯器，也有一個便利的超大文字系統來幫助使用者駕馭及探險 Forth 系統。

此指導是針對第一次使用 Forth 者學習其基本指令集，可以在短時間內加以運用，此指導只討論一百五十組指令。然而，這個小指令集，讓使用者對解決許多實際問題所需複雜演算法得以表達，因此也可爲更進一步學習 Forth 語言和完整的 F-PC 系統，紮下穩固的根基。

此指導包含六課有許多的例子及練習，可以被 F-PC 編譯，使用者可以立即做例子以下是如何開始 F-PC 及編譯課文，閱讀說明，測試例題及做練習 Forth 指令。

一、 安裝 F-PC

.本文

如果有全部的 F-PC 磁片，你必須執行系統磁片裡的 INSTALL 程式來將它安裝入你的 PC 。

將 Disk 放入 A 槽鍵入：

```
A>INSTALL
```

安裝程式會指引你，並且建立一個 F.EXE 執行檔，你需要大約 4MB 空間的硬碟來安裝全部的 F-PC 系統，在 C 碟會產生一個 F-PC 子目錄，你可以執行它，鍵入：

```
C:\F-PC>F
```

幾秒後螢幕會出現 F-PC 的畫面，則 FORTH 可以使用了

如果你沒有全部的 F-PC ，你可以使用指導裡的 F.EXE 檔來研究課文，事實上你不需用硬碟，裝配於 360K 磁片裡的課文和 F.EXE 檔，只要一個槽就

能使用，把指導磁片放入 A 槽打入：

```
A>F
```

二、 學習課文

把課文磁片放入 A 槽且 Forth 已載入，鍵入：

```
A:
```

確定 A 槽是預設的磁碟機，你可以打開第一課讀說明，鍵入：

```
open lesson1
b
```

"b"指令是尋求 F-PC 編輯器的幫助，並且在編輯視窗讀 lesson1.seq 檔，可以利用游標和 PqUp、PqDn 來讀，按 F10 跳出，回到 Forth。

編譯和測試第一課的例子，鍵入：

```
ok
```

"OK"是編譯目前開啓的檔，你可以鍵入第一課所定義的指令，來證明和課文中討論的是做相同的工作

第一課之後，以相同的動作讀說明

```
open lesson2
b
```

你可以編譯第二課，鍵入：

```
open lesson2
ok
```

全部六課皆以相同之方式

三、練習

如果問題簡單，你可以鍵入新指令的定義，馬上測試，但是最好將新指令建檔，你可以修改及重覆測試。

要做練習，你必需開啓新檔，鍵入你的新指令，鍵入：

```
newfile exercise
```

"newfile"建立一個名叫 exercise.seq 的新檔，並尋求編輯器的幫助，可以開始輸入新指令的定義，使用例子做爲新指令的模擬，當你完成指令碼輸入，按 F10 跳出編輯，回到 Forth 鍵入：

```
ok
```

編譯你的新指令，然後加以測試。

如果 Forth 在編譯你的指令碼有問題時，它會尋求編輯器的幫助，且游標會停在無法辨認之指令處，你可以做正確的修改，離開編輯，鍵入：

```
ok
```

再編譯指令碼

如果要直接編輯 exercise.seq 檔，鍵入：

```
open exercise
e
```

" e " 和 " b "類似，但 " e " 可以在檔案裡編輯文章 " b " 只能瀏覽，不能修改你可以不須經過 " open<filename>OK " 之過程來編譯任何檔，只要鍵入以下之指令

```
include lesson1
include lesson2
include exercise
```

以此類推，如果在 exercise.seq 檔語法錯誤 " include exercise "將尋求編輯器的幫助，且游標會指出該指令讓你編輯

編輯器是非常直覺的，它接受 Werdstar 的控制指令，如果你按 ESC 鍵，可以在螢幕頂端看到主功能表，可以用游標來選出一個求助項，並且拉下一個詳細編輯器功能之求助功能表。

此編輯器允許你在新指令裡改正語法錯誤，當你成功的編譯新指令後，仍需不斷測試，確定它們正確的做該做的，你可能做錯當機，但使用課文中的指令，對電腦或磁碟中的資料不會有嚴重的傷害，最糟的只是重新開機再次執行 F.EXE

幫助你將定義的指令除錯，F-PC 提供一個有力的除錯器，有單步除錯的能力，要測試和除錯你定義的指令，如要對 mycommand 除錯則鍵入：

```
dbg mycommand
```

會出現兩層螢幕，上半部指出在 mycommand 定義下的指令，下面視窗展示出資料堆疊的內容和下一個執行的指令，按<Enter>指令會一個接一個測試，藉著檢視資料堆疊的改變，你可以隨著執行的順序來確認錯誤的原因，用編輯器改正它的編譯指令碼並再試一次。

程式是編輯、編譯及測試的不斷循環直到每一樣都做對，它經常使我們情緒很壞，但如果做對了，會很滿足。

四、程式格式和程式助手

課文是以一種你要跟隨寫自己的指令碼的形式寫的，大量有關指令碼的說明文字是用來解釋和註解此指令碼，如果你花時間去把指令碼加以解釋或做成文件，那麼等你寫好指令碼即使是十年之後來閱讀也毫無困難，而且別人來維護也很容易，以下是幾個註解指令碼的方式

```
(comments)          \ ( 表示開使一個註解直到右括號爲止
\comments           \ 表示開使一個註解直到最後一列爲止

comment:            \ 多列註解直到 comment; 爲止
   comment 1        \ 註解 1
   comment 2        \ 註解 2
     .....          \ ...
   comment n        \ 註解 n
```

```
comment;                \ 結束註解
EXIT                    \ 停止編譯，以後所有行列皆視爲註解
```

選擇好的指令名稱，使程式易讀，以下是好的 Forth 程式命名的原則

(一)使用有意義的英文字。
(二)不要縮寫。
(三)使用複合字如 CountEggs , InitDataArray 名字可有 31 個字。
(四)使用描述指令做什麼的名字，讓指令碼解釋它如何做事。
(五)不要例舉像 DataAray1,DataArray2 等，給予功能的描述像
TemperatureArray , PressArray 等。
(六)指令碼分解成短的片語並加以註解。
(七)定義短的定義並使重覆使用他們以提高詞彙的再使用能力。
(八)在新定義裡避免加入內部迴圈的巢狀結構。
(九)對輸入和輸出堆疊(Stack)項目，時時加以註解。
(十)在一個檔裡組成相關連的指令。

第 一 課　印出字串

此課用最簡單的例子來說明 Forth 的原則；從現有的指令集裡建立新指令我們使用簡單的 Forth 指令" XXXX "來顯示螢幕上的字，並使用它來建立一指令集允許我們在螢幕上建立任何一組字，爲了說明 " ." " 指令的用法，讓我們來寫第一個 Forth 程式。

金城 註疏：

." 是一個 FORTH 的詞(word)也就是其他語言所謂的指令，所以其前後皆要有空白，才能被分辨(解譯)執行。

例題一、普通的問後

```
: hello ." hello, world!" ;
```

現在，如果你打入"hello"然後按<Enter>，" hello, world! "會出現在 " hello " 之後

```
:                       \ 開始新指令
hello                   \ 新指令之名稱
."                      \ 印出字串，不包括下一個 "
;                       \ 結束新指令
```

"hello"是新指令其功能是在螢幕上印出"hello, world!"

例題二、 大 F

現在我們要做的是使用簡單技巧來顯示大型的英文字元在螢幕上，以 F 爲例

```
: bar   cr  ." ***** "  ;
: post  cr  ." *     "  ;
: F     bar post bar post post post  ;
```

打入" F "按 Enter，你可以看見螢幕上之 F 被顯示了

```
*****
*
*****
*
*
*
```

F 有兩個成分，一個由 5 個 * 組成的 bar 及一個可由一個或多個 * 代替的 Post，我們定義兩個新指令" bar" 及" Post "分別顯示 5 個 * 和一個 * 最後的指令 F 定義爲顯示一個 bar、一個 Post、一個 bar，三個 Post。

指令" cr "開始新的一行，使得後來的字由螢幕最左邊顯示

金城 註疏：

此例題給了一個最重要的 FORTH 概念，將一個複雜的問題先"分析簡化"，再將其製造成零件(一個 FORTH 的詞)，最後再予組合。

練習一、

使用新指令 bar 及 Post 定義新指令" C "、" E "、" L "在螢幕上顯示對應的大型字元。

練習二、

分析你的姓，一組像 bar 及 post 的指令，使用它們來組合你有姓裡所有的字母，我以" TING "爲例

金城 註疏：

FORTH 的精神，就是簡化問題。因此，牢記 FORTH 的第一條無上心法 "如果你不能將問題分析簡化，那代表你並未真正透徹的了解問題，所以請回頭去探索、分析問題的本身。

例題三、我的名字

```
: center cr  ."    *  "  ;
: sides  cr  ." *   *"  ;
: triad1 cr  ." * * *"  ;
: triad2 cr  ." **  *"  ;
: triad3 cr  ." *  **"  ;
: triad4 cr  ."  *** "  ;
: quart  cr  ." ** **"  ;
: right  cr  ." * ***"  ;
: bigT  bar center center center center center center ;
: bigI  center center center center center center center ;
: bigN  sides triad2 triad2 triad1 triad3 triad2 sides ;
: bigG  triad4 sides post right triad1 sides triad4 ;
: TING  bitT bigI bigN bigG ;
```

練習三、

以此方式來組合英文字母是容易的，問題是在 5*7 的格式裡組合 26 個字母需多少基本指令? 關於標點符號呢?

練習四、

原則上，我們可以類似的技術來組合中文字，但需 16*16 之格式編譯更多中文則需 24*24 格式，試著用 5*7 格式組合簡單的中文，例如日、月、土

金城 註疏：

此題是說明了中文系統中組字法的觀念，仍是分析簡化。

例題四、在螢幕上顯示大的字元

試著在螢幕的任何地方寫出組合字，螢幕可以 25*80 格式顯示字元，即 25 列、每列 80 字，下列指令允許我們在寫字元前移動游標的位置。

```
dark                \   清除螢幕游標放在左上角
at                  \   移動標至螢幕指定位置
                    \   40 12 at 把游標放在螢幕中央
```

金城 註疏：

x y at 指令的前者"x"是橫座標，後者"y"是縱座標。
FORTH 將螢幕的左上方視爲(0,0)

以下指令把一大 F 放到螢幕中央

```
: newBar        ." *****" ;
: newPost       ." *    " ;
: new-F
      dark
      38 10 at newBar
      38 11 at newPost
      38 12 at newBar
      38 13 at newPost
      38 14 at newPost
      38 15 at newPost
      cr
      ;
```

顯然地，這是放置文字的笨拙方法，如果必須在指令裡指定文字明確的位置，一個放置文字較普遍的方法是使用變數(Variables)來貯存位置，以致於一個字母可以放在螢幕上的任何地點。

金城 註疏：

(一)工具的設計要簡單，但也要顧及靈活的彈性。否則就要寫很多版本，反而更累人。

(二) FORTH 的無上心法第二條 "設計工具之前，先想清楚未來的使用方式，是否簡單、靈活、順手，再下手去做"。

```
variable x
variable y
: newLine
        x @ y @ at                      \    移動游標到(X，Y) 位置
        1 y +!                          \    增加 Y 至下一列
        ;
: F     newLine newBar
        newLine newPost
        newLine newBar
        newLine newPost
        newLine newPost
        newLine nwePost
        ;
```

把 F 放在螢幕上必須先指定其位置：

```
    30 x !   10 y !   F
```

這裡所學的是幾個 FORTH 指令

```
variable <name>             \  定義一個變數(Variable)，數字可以
                            \  貯存及取出
@       ( var -- data )     \  取出存在變數(Variable)中的數字
!       ( data var -- )     \  存一個數字到變數(Variable)中
+!      ( data var -- )     \  把存在變數(Variable)中之數字增加一
```

金城 註疏：

(一)在 FORTH 中由小括號括起來的範圍是註解，別忘了小括號本身也是一個指令。

(二)要留心 FORTH 的變數，是動詞，而不是代名詞。

(三) FORTH 系統沒有識別字的規定，也沒有保留字的觀念，所有的詞(word)一律看成相等地位，可謂之"有容乃大"

```
: F-demo
        dark
        0 x ! 0 y ! F
        70 x ! 10 y ! F
        10 x ! 18 y ! F
        40 x ! 15 y ! F
        ;
```

練習五、

定義一個新指令來清除螢幕，並把 Forth 此訊息以組合字母放到螢幕中央。

```
: bar           newLine ." *****" ;
: post          newline ." *    " ;
: triad1        newline ."  *** " ;
: sides         newline ." *   *" ;
: tetra         newline ." **** " ;
: duo1          newline ." **   " ;
: duo2          newline ." * *  " ;
: duo3          newline ." *  * " ;
: center        newline ."   *  " ;
: F     bar post bar post post post ;
: O     triad1 sides sides sides sides trial1 ;
: R     tetra sides tetra duo2 duo3 sides ;
: T     bar center center center center center ;
: H     sides sides bar sides sides sides ;

: FORTH dark
        25 x ! 10 y ! F
        32 x ! 10 y ! O
        39 x ! 10 y ! R
        46 x ! 10 y ! T
        53 x ! 10 y ! H
        ;
```

練習六、

自己設計一個訊息，將其顯示於螢幕中央。

金城 註疏：

(一) FORTH 的程式設計，簡單的說就是爲一個應用問題，創造出一群簡單、靈活、順手的新指令。每個新指令，就成了指令群中的一部份，此爲 FORTH 最大的特點，與 APL 的觀念相似。

(二)如果指令設計的好(簡單、靈活)，則可以訊速有效的解決未來所遭遇到更複雜的應用問題。所以工具品質的好壞，直接影響工作的效率。因此，有人說 FORTH 是程式設計師能力的放大鏡，

例題五、重覆的圖樣

電腦非常擅於重覆處理事件，很多 FORTH 指令被設計使程式容易處理重覆的工作

```
DO   ( limit index -- )     \  設定 LOOP 給極限和指數
LOOP ( -- )                 \  從 1 增加指數並比較極限
                               如果 Index=limit 跳出 LOOP，否則
                               重覆 DO 之後的指令
I    ( --index )            \  傳回目前 LOOP 指數
```

要定義 loop 必須在冒號裡定義， I 只能用於 DO 和 LOOP 之間，當 DO....LOOP 語法被執行，Loop 增加指數並與 Limit 比較，如果 index 等於或大於 limit ，結束 Loop ，如果增加的指數仍小於 limit，在 Loop 裡之指令會重覆執行

以一些例子來測試 DO....LOOP 指令

```
VARIABLE width              \  印出星號的字

: Asterisks                 \  在螢幕上印出 n 個星 n = width
       width @ 0            \  開始指數=0 ，limit=寬
```

```
        DO ." *"            \  一次印一個星號
        LOOP                \  重覆 n 次

: Rectangle ( height width -- , 印出星號的長方形 )
        width !                 \  印出最初之寬
        0                       \  limit=高，index=0
        DO      CR
                Asterisks       \  印出一列星號
        LOOP
        ;

: Parallelogram ( height width -- )
        width !
        0
        DO      CR I SPACES     \  移動行列至右邊
                Asterisks       \  印出一列星號
        LOOP
        ;

: Triangle ( width -- ,   印出一個星字三角形 )
        1                       \ limit=width, initial index=1
        DO      CR
                I width !       \  增加每一列的寬
                Asterisks       \  印出一列星號
        LOOP
        ;
```

測試上述之指令

```
3 10 Rectangle
5 18 Parallelogram
12 Triangle
```

練習六、

想出一些有趣的圖形，可在螢幕上顯示的，並寫程式來產生此圖形例如。

```
        *
       * *
      * * *
     * * * *
      * * *
       * *
        *
```

例題六、畫框框

IBM-PC 有一組給螢幕顯示的圖形字元，它們可以在螢幕上畫出簡單圖形，來加強你的文字訊息，當在 F-PC 編輯器，按 ESC A 和 Enter ，就可以看見這些字元，在此例中，我們要畫出一個環繞一篇文章的框框，所需的指令如下：

```
218 EMIT        \  印出左上角 ┌
191 EMIT        \  印出右上角  ┐
192 EMIT        \  印出左下角  └
217 EMIT        \  印出右下角 ┘
176 EMIT        \  印出垂直線  │
196 EMIT        \  印出水平線  ─
```

讓我們看看如何在螢幕上畫出一個框框，我們假定框框左上角是(X，Y)，右下角是(X1，Y1)

```
VARIABLE x1
VARIABLE y1

: TopSide ( -- )
        x @ y @ AT              \  游標移至左上方
        x1 @ x @                \  水平極限(終點及起點)
        DO      196 EMIT        \  印出上方之邊線
        LOOP
        ;

: BottomSide ( -- )
        x @ y1 @ AT             \  游標移至左下方
        x1 @ x @                \  水平極限
        DO      196 EMIT        \  印出下方之邊線
```

```
        Loop
        ;

: LeftSide ( -- )
        y1 @ y @                \  垂直極限
        DO      x @ I AT        \  游標位置
                179 EMIT        \  畫出左邊線
        LOOP
        ;

: RightSide ( -- )
        y1 @ y @                \  垂直極限
        DO       x1 @ I AT      \  游標位置
                 179 EMIT       \  畫出右邊線
        LOOP
        ;

: Corners ( -- )
        x @ y @ AT
        218 EMIT                 \  左上角 ┌
        x @ y1 @ AT
        192 EMIT                 \  左下角 └
        x1 @ y @ AT
        191 EMIT                 \  右上角 ┐
        x1 @ y1 @ AT
        217 EMIT                 \  右下角 ┘
        ;

: Box ( x y x1 y1 -- )
        y1 ! x1 !                \  貯存右下角
        y ! x !                  \  貯存左上角
        LeftSide RightSide       \  畫兩邊
        TopSide BottomSide       \  畫頂和底
        Corners
        ;

: demo FORTH                     \  印出大 Forth
       20 8 62 17 box            \  在它四周畫出框框
       0 21 at ;                 \  移動游標出來
```

打入 DEMO 你可以看到 Forth 被圍在框裡

練習七、

在 IBM-PC 裡的雙線字元是 ALT-205 -----ALT-188(永 B 說 B　B 感 B 滿 B 砑 B 崽 B 吽 B 芊 B 恄 B*) ，如上使用它們組合雙線框框。

金城 註疏：

FORTH 的程式設計方式，一如其它語言的結構化模組，但 FORTH 的架構鼓勵使用者，寫許多個短式子，來組合成應用的全部程式，短的式子有許多好處，第一個是能專心思考。第二個是較容易除錯。第三個是可重覆使用。第四個是爲逐步改善。最後一個是立即測試。

第 二 課　二個文字遊戲

Forth 程式的本質是從現存的指令和模組，建立一個更大更有力的模組，從小組指令開始，可以建立內容充實的程式處理有趣的事

我們學過了 Forth 的小指令，包括 " ." " 在螢幕上顯示字串，我們將使用此指令來設計兩個遊戲之程式，來玩弄文字串

例題一、

" The Theory that Jack built " 這篇文章是先定義一些片語，然後用它來建立句子和章節，此例子教我們如何在 Forth 裡以小的模組建立大的程式

首先定義一些簡單片語

```
: the              ." the " ;
: that             cr ." That " ;
: this             cr ." This is " the ;
: jack             ." Jack Builds" ;
: summary          ." Summary" ;
: flaw             ." Flaw" ;
: mummery          ." Mummery" ;
: k                ." Constant K" ;
: haze             ." Krudite Verbal Haze" ;
: phrase           ." Turn of a Plausible Phrase" ;
: bluff            ." Chaotic Confusion and Bluff" ;
: stuff            ." Cybernatics and Stuff" ;
: theory           ." Theory " jack ;
: button           ." Button to Start the Machine" ;
: child            ." Space Child With Brow Serene" ;
: cybernatics      ." Cybernatics and Stuff" ;
```

現在建立複雜的合成的片語

```
: hiding          cr ." Hiding " the flaw ;
: lay             that ." Lay in " the theory ;
: based           cr ." Based on " the mummery ;
```

```
: saved          that ." Saved " the summary ;
: cloak          cr ." Cloaking " k ;
: thick          IF that ELSE cr ." And " THEN
                 ." Thickened " the haze;
: hung           that ." Hung on " the phrase ;
: cover          IF that ." Covered "
                 ELSE cr ." To Cover "
                 THEN bluff ;
: make           cr ." To Make with " the cybernatics ;
: pushed         cr ." Who Pushed " button ;
: without        cr ." Without Confusion, Wxposing the Bluff" ;

: rest                              \ 在使行動中止
       ." . "                       \ 印出一節
       10 SPACES                    \ 空 10 個間隔
       KEY                          \ 等使用者按一鍵
       DROP CR CR CR ;

: recite                            \  朗讀詩篇
       dark cr this theory rest
       this flaw lay rest
       this mummery hiding lay rest
       this summary based hiding lay rest
       this k saved based hiding lay rest
       this haze cloak saved based hiding lay rest
       this bluff hung 1 thick cloak
              saved based hiding lay rest
       this stuff 1 cover hung 0 thick cloak
              saved based hiding lay rest
       this button make 0 cover hung 0 thick cloak
              saved based hiding lay rest
       this child pushed
              cr ." That Made with " cybernatics without hung
              cr ." And, Shredding " the haze cloak
              cr ." Wrecked " the summary based hiding
              cr ." And Demolished " theory rest
      ;
```

金城 註疏：

FORTH 程式的整體架構類似樹狀結構，由葉子與樹枝構成樹的支幹，再由支幹構成主幹。而執行時也類似樹木的養份輸送，由主幹將水份送往支幹，支幹再分送樹枝和葉子，到達小葉子上行光合作用(真正的有效運作)。以資訊專業的眼光來看就是其他語言的函數呼叫與參數傳遞。

```
KEY       ( --char )      \  等候按鍵，並傳回該按鍵的 ASCII 碼

DROP      ( n-- )         \  丟棄在堆疊(stack)頂端的數字
SPACE     ( -- )          \  顯示一個空白
SPACES    ( n-- )         \  顯示 n 個空白
IF        ( f-- )         \  如果旗標(flag) 是0，跳至 ELSE or
                             THEN 如果旗標(flag) 不是0，執行
                             底下的指令，
                             至 ELSE 然後跳至 THEN

ELSE      ( -- )          \  跳過以下的指令到 THEN

THEN      ( -- )          \  結束 IF-ELSE-THEN
                             或 IF-THEN
```

金城 註疏：

(一) FORTH 的 THEN 原本是 ENDIF 的意思，請不要弄混了。

(二)上列式子中由小括號括起來的註解部份，爲 FORTH 的堆疊狀態圖，雙減號之前的爲此動詞所需的輸入資料，之後的爲輸出結果。

(三) FORTH 的條件指令與組合語言的使用觀念一樣。

KEY、DROP 稍後討論，目前其作用是讓電腦暫停，給你時間去讀詩章，當按任意鍵，電腦會繼續執行下一個指令" rest "會在每一章結束時執行。

IF---ELSE---THEN 允許電腦依其執行狀況選擇交替的指令，這些指令以群體方式使用。

(flag) IF <true-clause> ELSE <false-clause> THEN

(flag) IF <true-clause> THEN

False Clause 是任意的，舉例：" 1cover "是相等於

that ." Covered " bluff

而" 0cover "相等於

cr ." To Cover " bluff

這個條件的分支結構，對電腦選擇一或二個不同路徑來執行是很方便的。

現在我們可以建一個大而更複雜的指令，一章一章的印出章節執行。

RECITE

以文章順序印出全部東西

練習一、

以上述技巧輸入你喜歡的詩篇，給每一篇獨特的名字，作爲新的 Forth 指令，當你打入一首詩的名字，所有的詩會以動人的形式在螢幕上被顯示。

例題二、

這是一個對話遊戲，電腦將對不同問題給予忠告，比如性、健康、錢、工作，是從 " Basic Computer Games "改編而來。

```
: Advisor
   CR ." Hello!  My name is Creating Computer."
   CR ." Hi there!"
   CR ." Are you enjoying yourself here?"
```

```
   KEY 32 OR 89 =
   IF      CR ." I am glad to hear that."
   ELSE    CR ." I am sorry about that."
           CR ." maybe we can brighten your visit a bit."
   THEN
   CR ." Say!"
   CR ." I can solved all kinds of problems except those dealing"
   CR ." with Greece.  What kind of problems do you have"
   CR ." ( sex, health, money or job )?"
   CR
   ;

: question
      CR ." Any more problems you want to solve?"
      CR ." What kind ( sex, job, money, health ) ?"
      CR
      ;

: sex   CR ." Is your problem TOO MUCH or TOO LITTLE?"
        CR
        ;

: much  CR
     ." You call that a problem?!!  I SHOULD have that problem."
        CR
     ." If it really bothers you, take a cold shower."
      question
      ;

: little
      CR ." Why are you here!"
      CR ." You should be in Tokyo or New York of Amsterdam or"
      CR ." some place with some action."
      question
      ;

: health
      CR ." My advise to you is:"
      CR ."      1. Take two tablets of aspirin."
```

```
        CR ."      2. Drink plenty of fluids."
        CR ."      3. Go to bed (along) ."
        question
        ;

: job
    CR ." I can sympathize with you."
    CR ." I have to work very long every day with no pay."
    CR ." My advise to you, is to open a rental computer store."
     question
    ;

: money
        CR ." Sorry!  I am broke too."
        CR ." Why don't you sell encyclopedias of marry"
        CR ." someone rich or stop eating, so you won't "
        CR ." need so much money?"
        question
        ;
```

開始對話，鍵入：

```
        ADVISOR
```

此對話利用 Forth 整合的交談環境的天性，使用者的回答被定義爲 Forth 的指令，當打入指令時，似乎在回答電腦的問題，這些指令印出給使用者的正確忠告。

金城 註疏：

此範例希望能讓使用者體會到，原來其他語言要寫一長串的指令來判斷使用者輸入的字串，再層層過濾找出適當的動作，而 FORTH 卻只用了幾段話就定義了整個問答的過程，和答覆的內容，非常的自然、流暢。

練習二、

使用例子二的技巧定義更多忠告(婚姻、孩子、法律關係.....)。

練習三、

建立一個中國幸運餅工廠，定義許多的幸運餅當 Forth 指令，把名單給使用者，他可以鍵入名字，獲得幸運消息，稍後我們學亂數字的產生器，使用亂數來選擇幸運餅。

第 三 課　數　字

這一課要討論 Forth 處理整數的方法， 16 位元整數是從 -32768 到 32767 的數，在 Forth 裡很方便貯存和執行， Forth 也可以處理很大的雙倍精密度數，甚至浮點數，但已超出於本課之範圍。

金城 註疏：

FORTH 的數字使用方式，喜歡整數的運算，因爲 FORTH 的傳統觀念中，在沒有硬體浮點運算能力的 CPU 上，使用整數來運算，以求得速度與精密度的最佳效果。

例題一、

數字在 Forth 裡如何使用的第一個例子是換錢程式，不同貨幣之兌換：

24.55 NT	1 Dollar
7.73 HK	1 Dollar
5.47 RMB	1 Dollar
1 Ounce Gold	356 Dollars
1 Ounce Silver	4.01 Dollars

以美元爲標準貨幣，把所有貨幣兌換成美元，所有的數學操作，會在美元裡執行，錢也可以從美元兌換成其他貨幣

我們以貨幣名稱爲定義字將其換爲美元，而兌換美元爲一種貨幣，執行字是 " $ "記號

```
: NT    ( nNT -- $ )    10 245 */  ;
: $NT   ( $ -- nNT )    245 10 */  ;
: RMB   ( nRMB -- $ )   100 547 */  ;
: $jmp  ( $ -- nJmp )   547 100 */  ;
: HK    ( nHK -- $ )    100 773 */  ;
```

```
: $HK   ( $ -- $ )      773 100 */  ;
: gold  ( nounce -- $ ) 356 *  ;
: $gold ( $ -- nounce ) 356 /  ;
: silver ( nounce -- $ ) 401 100 */  ;
: $silver ( $ -- nounce ) 100 401 */  ;
: ounce ( n -- n, a word to improve syntax )    ;
: dollars ( n -- )    . ;
```

以此組換錢的字，可以做一些測試

```
5 ounce gold .
10 ounce silver .
100 $NT .
20 $RMB .
```

金城 註疏：

一個寫 FORTH 程式的重要概念爲先想最後要怎麼用，再由使用的方式中分析此動詞(指令)所需完成的工作內容，然後經過整理和簡化，才動手寫作。切記：" 先想清楚要怎麼用，才會知道要怎麼寫 "。

如果你皮夾裡有不同的紙鈔，你可以把它們都加起來換成美元

```
1000 NT 500 HK + .s
320 RMB + .s
dollars                         \   以美元印出稅值
```

" dollars " 與 " ." 相同，僅顯示一個數字，使用" dollars "來顯示錢的總數，在英文語法上是很自然的

此時如果我在香港總數也可快速的換成港幣

```
1000 NT 500 HK + 320 RMB + .s
$HK dollars                     \   全部換成港幣，印出
```

金城 註疏：

寫一個 FORTH 的程式一如寫一首詩，要信、達、雅兼顧，所以 dollars 的用途就是在提昇「雅」的境界。

練習一、商物旅行

帶 1000 美元離開 S.F 到 H.K 用去 1200 HK 因 S.H 得到 2000 RMB 在 H.K 用去 900 HK 在臺北用去 30000 NT 剩多少美元：

答案是 ：

```
1000 1200 HK - 2000 RMB + 900 HK - 30000 NT - dollars
```

試一試

例題二、溫度換算

攝氏和華氏之換算與換錢不同，乃在此兩種溫度，除刻度因素外有一個差距值

```
: F>C ( nFarenheit -- nCelcius )
        32 -
        10 18 */
        ;

: C>F ( nCelcius -- nFarenheit )
        18 10 */
        32 +
        ;

90 F>C .                    \   顯示夏天白天的溫度
0 C>F .                     \   顯示冬天晚上溫度
```

金城 註疏：

在 FORTH 中， */ 的運算，主要是應用在分數的乘法上。

例如： 10 18 */ 就是乘以 9 分之 5。

在以上的例子，我們使用下面之 Forth 數學運算子

```
+   ( n1 n2 -- n1+n2 )          \ 加 N1 和 N2，把和留在堆疊(Stack)
-   ( n1 n2 -- n1-n2 )          \ 從 N1 減 N2，把差留在堆疊(Stack)
*   ( n1 n2 -- n1*n2 )          \ N1 乘 N2，把積留在堆疊(Stack)
/   ( n1 n2 -- n1/n2 )          \ 以N2 除以 N1，把商留在堆疊(Stack)
*/  ( n1 n2 n3 -- n1*n2/n3 )    \ N1 乘 N2 之積除以 N3，
                                  把商留在堆疊(Stack)
.s  ( ...--... )                \ 把堆疊(Stack)最上方的 4 個數字顯
                                  示出來，此字不會改變堆疊(Stack)
                                  的深度及內容。
```

我們必須介紹堆疊(Stack)之概念、堆疊(Stack)是電腦之記憶區來儲存及取回數字，與第一課之變數(Variables)不同，變數(Variable)是在記憶體中命名的位置，要以指定之名稱抓取，堆疊(Stack)是一張先進後出(First-In-Last-Out)的表，當給 Forth 一個數字時會放在堆疊(Stack)上，任何運算子所需之數字，必須由堆疊(Stack)上彈出所需數字，最容易取得的字是堆疊(Stack)最頂端的字，很多的 Forth 指令，可以製造一個或很多個數字而這些數字產生時會放在堆疊(Stack)上。

" + "使堆疊(Stack)最頂端之兩個數字彈出，相加後，將和放回堆疊(Stack)，-、*、/ 是處理簡單數學常使用的運算子，必須注意給" + "和" * "兩個數字的先後順序是無關緊要的，而給" - "和" / "之先後順序是重要的，交換這兩個數字，分別會產生不同差或商。

在 Forth " */ "是一種比例運算子，尤其在整數的比例運算中，以 N2/N3 之比例乘 N1 ，由例一、例二顯示，此種比例運算子是非常有力的，分數乘法的運用可以免去使用浮點點之必要。

.S 是有力的除錯工具顯示出堆疊(Stack)最上面四個項目之內容，它常在除錯時使用，來確定在計算時堆疊(Stack)之數字是正確的，它通常不用在最終程式以免印出太多中間值

一些不常用，但重要之數學運算子：

```
MOD     ( n1 n2 -- rem )        \  N1 除以 N2，餘數留在堆疊(Stack)上

/MOD    ( n1 n2 -- rem quot )   \  N1 除以 N2，把餘數和商留在
                                   堆疊(Stac k)
ABS     ( n -- |n| )            \  換算堆疊(Stack)頂部之數
                                   字爲絕對值
NEGATE  ( n -- -n  )            \  更改堆疊(Stack)頂端 N 之正負號
```

當我們繼續學下去，這些運算子必然會使用到

堆疊(Stack)操作指令

堆疊(Stack)是個最重要的地方，先前執行的結果，由前一個指令留到現在執行的指令，指令從堆疊(Stack)取走參數，將結果留在堆疊(Stack)給後來之指令，堆疊(Stack)可以很容易地被成串的副程式共用，此副程式可以呼叫別副程式，以此類推，此副程式是 Forth 的詞，它們可毫無限制的組合，這是爲什麼 Fortrh 結構和的語法構造上如此簡單的重要原因。

然而，在堆疊(Stack)裡把不正確的數字順序給指令是常發生的像 - 及 /，有一組堆疊(Stack) 操作指令可重新排堆疊(Stack)上的數字這 5 個最重要的高階堆疊(Stack) 操作指令是：

```
DUP   ( n -- n n )              \ 複製堆疊(Stack)之頂端之數字
SWAP  ( n1 n2  -- n2 n1 )       \ 把堆疊(Stack)頂端的兩個數字互換
OVER  ( n1 n2 -- n1 n2 n1 )     \ 複製堆疊(Stack)的第二個數字
ROT   ( n1 n2 n3 -- n2 n3 n1 ) \ 把第三個數字換至堆疊(Stack)頂端
DROP  ( n -- )                  \ 拋棄堆疊(Stack)的頂端的數值
```

金城 註疏：

剛開始學 FORTH 的時候，會很不習慣堆疊(stack)的運作方式，尤其是使用過其他高階語言的人們，對使用大量變數的程式寫作方式已然在心中定型，所以學 FORTH 便倍感吃力。但堆疊(stack)是 FORTH 最重要的觀念，所有的參數傳遞與運算皆在堆疊(stack)中完成。可

謂之學 FORTH 的任督二脈。如果學者真的無法了解堆疊(stack)的操作觀念，可以想像堆疊(stack)就是區域變數與參數的共用區。其實如果真的深入了解各種高階語言的編譯程式，便會知曉堆疊(stack)的實用價值了。

例題三、長方形

長方形是由(X，Y)座標來指定其左上及右下角，以堆疊(Stack)裡的這 4 個整數，我們可以計算長方形之面積、中心及周長。

```
: area ( x1 y1 x2 y2 -- area )
       ROT -   ( x1 x2 y2-y1 )          \  求高
       SWAP ROT - ( y2-y1 x2-x1 )       \  求寬
       * ( area )                       \  寬乘高成面積留在
       ;                                   堆疊(stack)頂端

: center ( x1 y1 x2 y2 -- x3 y3 , center coordinates )
       ROT - 2/ ( x1 x2 y3 )            \  求高的中心點
       SWAP ROT - 2/ ( y3 x3 )          \  求寬的中心點
       SWAP ( x3 y3 )
       ;

: sides ( x1 y1 x2 y2 -- sides )
       ROT - ABS ( x1 x2 y2-y1 )         \  求高
       SWAP ROT - ABS ( y2-y1 x2-x1)     \  求寬
       + ( sides ) 2*
       ;
```

邏輯運算子

電腦使用邏輯運算來決定及遵循不同之執行通道也就是所謂的分歧，邏輯運算本身非常簡單且容易瞭解，然而多種邏輯運算的組合及在一個大程式裡對眾多不同的通路上做分歧的抉擇會使得電腦的運作顯得很複雜，甚至顯示出一些智慧的特質。

這裡我們介紹一些與數字合併使用的邏輯運算而且分歧之跳躍運算會使用他們

。以邏輯運算的結果來選擇不同之運作。

Forth 使用整數來表示邏輯值，只有 2 種值真與僞，凡不是 0 的任何數代表真，以 0 表示僞，一個 Forth 邏輯運算所傳回的真值通常傳回 -1 ，此數字代表邏輯的值也稱作旗標(Flog)，此點與 C 語言的觀點是一致的。

```
>       ( n1 n2 -- f )        \  如果 N1>N2 傳回對，否則傳回僞
<       ( n1 n2 -- f )        \  如果 N1<N2 傳回旗標爲真.
=       ( n1 n2 -- f )        \  如果 N1=N2 傳回旗標爲真.
0=      ( n -- f )            \  如果 N = 0 傳回旗標爲真.
0<      ( n -- f )            \  如果 N < 0 傳回旗標爲真，用來檢查
                                 是否爲負數
NOT     ( f1 -- f2 )          \  如果 f1 是錯傳回真
                                 ，如果 f1 是真傳回錯
AND     ( n1 n2 -- n3 )       \  n3 是 n1 和 n2 每一位元作 AND 的結果
OR      ( n1 n2 -- n3 )       \  n3 是 n1 和 n2 每一位元作 OR  的結果
```

在爲真和爲僞之旗標(Flags)裡，AND 和 OR 會正確運作

```
 -1  -1  AND   -1    \  true true AND    --   true
 -1   0  AND    0    \  true false AND   --   false
  0   0  AND    0    \  false false AND  --   false
 -1  -1  OR    -1    \  true true OR     --   true
 -1   0  OR    -1    \  true false OR    --   true
  0   0  OR     0    \  false false OR   --   flase
```

金城 註疏：

要小心 7 8 AND 的結果爲 0 ，以二進位來看就明白了。

在冒號定義下，旗標(Flag) 可以用來選擇二個執行通道的一個，經由下面之結構。

```
( f ) IF <true clause> ELSE <false clause> THEN

( f ) IF <true clause> THEN
```

它們在第二課簡短討論過，現在我們知道如何產生旗標(Flags)了。也知道如何結合旗標與分歧敘述的運用格式了。

金城 註疏：

人類因爲有抉擇判斷的能力，所以稱爲萬物之靈。而電腦也因爲提供了循序、迴路與分歧的控制流程指令而使得軟體程式透出一種幾乎近似人類的智慧能力。但千萬別倒因爲果，是有智慧的人才能寫出有智慧能力的程式，不是電腦本身的「天生偉大」。所以如果寫程式的人是白痴、蠢材，麼那寫出的程式必然也又呆又笨了。

例題四、天氣報告

下面之冒號定義說明邏輯和分歧的用法

```
: weather ( nFarenheit -- )
      DUP     85 >
      IF      ." Too hot!" DROP
      ELSE    55 <
              IF      ." Too cold."
              ELSE    ." About right."
              THEN
      THEN
      ;
```

你可以鍵入下列指令，並從電腦獲得一些回答

```
90 weather  Too hot!
70 weatehr  About right.
32 weather  Too cold.
```

金城 註疏：

判斷與分岐的關念，構成了條件子句，通常可用於做例外的處理或二

選一的執行流程。在 FORTH 的諸多工具中，可以用結構化的 D-型圖(Dijestra-Chart)來將分岐的架構平面化。一來容易了解自己的語意(Semantic)是否正確，二來容易轉換成 FORTH 的條件子句。另外，我要提醒的是，對 FORTH 的程式設計而言，應該僅量的少用 IF---ELSE---THEN。因爲分岐的不當使用，會使程式的複雜度提高，可讀性降低與速度變慢。

回顧 LOOP 指令

在冒號定義中以下列之格式使用固定次數迴路指令

```
 < nLimit nIndex >     DO   <repeat clause> LOOP
<迴路終值  迴路起值>   DO     < 覆的敘述>   LOOP
```

DO 從堆疊(Stack)取出兩個參數，頂端數字是迴圈的開始指數，第二個字是迴圈指數的上限進入 LOOP 之後<repeat clause>被重覆執行，LOOP 增加指數從 nIndex 至 nLimit ，當指數相等於 nLimit 時，LOOP 結束，在 <repeat clause> 裡，一個特殊運算子 I 傳回目前的迴路索引，並將之放在堆疊(Stack)的頂端。

以下是 LOOP 結構的簡單例子

例題五、印出乘法表

```
: oneRow ( nRw -- )
        CR
        DUP 3 .R 3 SPACES
        13 1
        DO      I OVER *          \  留底數並計算乘積
                4 .R              \  將堆疊(stack)頂端的乘積，以格
        LOOP                         式化的方式輸出
        DROP ;

: Table ( -- )
        CR CR 6 SPACES
        13 1
```

```
DO      I 4 .R  LOOP        \  印出頂端的表頭
13 1                        \  呼叫 oneRow (一行) 12 次
DO      I oneRow
LOOP
;
```

鍵入 Table 使乘法表，以整齊的格式顯示

練習一、

美國乘法表在 12*12 結束，但在中國和公制國家，乘法表在 9*9 結束，修改 TABLE 定義，使其表格限定在 1*1 至 9*9

金城 註疏：

我一向反對濫用迴路來解答問題，例如：要你求 1 到 100 的總合，你會用迴路加個 100 次，那要你求 1 到 1 億的總合，你又想用迴路加個 1 億次。好似迴路是不用付出代價的。其實如果你上過小學就會了解「首項加末項乘以項數之半」爲等差級數的總合。這是否足以說明某些問題如果你能找到運算公式的通則，大量濫用的迴路是可以避免的。另外，由於迴路代價甚高，所以如果非用不可，應僅量的使迴路內的工作單純化，尤其如果可能要避免在迴路使用過多的分岐條件子句以免使執行效率大爲降低。

第 四 課　數的兩個範例

在這一課，使用二個相當複雜的例子來，說明 Forth 指令的組合過程來解決真正的問題，使用在前三課我們學過的　Forth 指令

例題一、日曆

我們要做的是印出任一年任一月的月曆，事實上可從 1950 年到 2050 年任意選出年、月來印，日期每行以星期日，星期一以此類推來印，此問題，有許多事情要考慮：閏年、一個月的天數、那星期的那一天是某個一月的第一天，計算年、月、日，以(YYYY)表年、(MM)表月、(DD)表日，我們所求之年月日以(DD MM YYYY)表示出，用 7 除以年、月、日之餘數就是星期幾了。

以 1950,1,1 爲日曆的 0 日，剛好是 Sunday ，4 年有 1461 天，一個閏年有 366 天，任一年的年月日，可以由 YEAR 計算出，並存於變數(Variable)JULIAN，如果這一年是閏年，變數(Variable) LEAP 的值爲 1，否則爲 0 。

在一年裡每個月的第一天爲此月的第幾天很難計算，在指令 FIRST 裡處理；

FIRST 列出月份之數字(1 代表 1 月，2 代表 2 月等)並傳回 1 年之中從 1 月 1 日至此月的第一天之天數，2 月 1 日是一年的第 31 天，3 月 1 日不是第 59 天，就是第 60 天，依據閏年是否而定，4 月 1 日是第 90 或 91 天，以此類推，一個適切的公式，在 FIRST 裡被用來計算 5 月 1 日和其後的全部月份。

指令 DAY 從堆疊(Stack)中取出 DD、MM 及 YYYY，並將該天自 1950 年起的總天數放回堆疊(Stack)，總天數被使用來除以 7 的餘數決定此日爲星期幾，星期天是用 0 代表，星期六是 6 。

.DAY 印出一個月的月曆，由堆疊(Stack)上的月份所指定，並以星期排列之表格形式印出，此格式加上邊框來顯示，以指令 MONTH 指出月份名稱、日期名稱，例如鍵入：

```
1992 YEAR 7 MONTH
```

會顯示 1992 年的 7 月份的月曆，指令 CALENDER 取代任一年的 12 個月份的完整月曆，由堆疊(Stack)上年的數字指定，你無法在螢幕上看到全部，但你可以用印表機印出來。

1992 YEAR CALENDAR

```
VARIABLE JULIAN  \  用來存放自 1950 年 1 月 1 日起到今年第一天
                    所經過的天數
VARIABLE LEAP    \  用來存放是否爲閏年
1461 CONSTANT 4YEARS   \  包含閏年、四年的總天數

: YEAR ( YEAR --, 計算 Julian 與閏年是否爲 1 )
      1949 - 4YEARS 4 */MOD       \  自 1949 年 1 月 1 日起經過了幾天
      365 - JULIAN !              \  0 爲 1/1/1950 的第一天
      3 =                         \  若餘數爲 3 則爲閏年
      IF     1 LEAP !             \  閏年
      ELSE   0 LEAP !             \  普通年
      THEN ;
```

金城 註疏：

上列 IF 1 LEAP !
ELSE 0 LEAP !
THEN ; 部份可以改用 ABS LEAP! 而去掉判斷式。

```
: FIRST ( MONTH -- 計算每月的第一天爲本年的第幾天 )
      DUP 1 =
      IF DROP 0                        \  0 爲 1 月 1 日
      ELSE    DUP 2 =
          IF      DROP 31              \  31 爲 2 月 1 日
          ELSE    DUP 3 =
                IF    59 LEAP @ 2    \  3 月 1 日可  爲 59 天或 60 天
                ELSE  4 - 30624 1000 */
                      90 + LEAP @ +    \  4 月以後每個月乘以 30.624 天
                THEN
          THEN
      THEN
      ;
```

金城 註疏：

乘以 30624 的整數積剛好能使七、八兩月均爲 31 天，此爲本例題最精彩之處。0.624 爲 5 天除以八個月減十一月爲小月而得來。

```
: DAY ( DD MM YYYY -- JULIAN-DAY )
     YEAR                \  計算 JULIAN 與閏年
     FIRST + 1-          \  計算今天是今年的第幾天
     JULIAN @ +          \  計算由 1950 年 1 月 1 日起到今天
     ;                      一共過了幾天

: STARS 0 DO 42 EMIT LOOP ;     \ " * " 的 ASCII 碼爲 42

: header ( n -- )               \  印出表頭
    cr cr 26 stars space
    case     1 of ."  January " endof
             2 of ." February " endof
             3 of ."   March  " endof
             4 of ."   April  " endof
             5 of ."    May   " endof
             6 of ."   June   " endof
             7 of ."   July   " endof
             8 of ."  August  " endof
             9 of ." September" endof
            10 of ."  October " endof
            11 of ." November " endof
            12 of ." December " endof
            DROP
    endcase
    space 27 stars cr cr
    ."      SUN     MON     TUE     WED     THU     FRI     SAT"
    cr cr                       \  以下印出正確週、日期內容
    ;

: BLANKS ( MONTH -- )           \  將不是此月份的日子跳過
      FIRST JULIAN @ +          \  計算由 1950 年 1 月 1 日到這
                                   個月過了幾天
      7 MOD 8 * SPACES ;        \  如果不是星期天跳過前一個月
                                   在本週的餘尾天數
```

```
: .DAYS ( MONTH -- )            \  印出本月的天數，以週的格式印出
      DUP FIRST                 \  此月份之第一天的年天數
      SWAP 1 + FIRST            \  下月份之第一天的年天數
      OVER - 0                  \  以迴路印出數於本月的天數
      DO I OVER +
        JULIAN @ + 7 MOD        \  計算是星期幾
        0= IF CR THEN           \  如果是星期天跳至下一行的起頭列印
        I 1 + 8 U.R             \  每一天佔八個格子靠右印出
      LOOP
      DROP ;                    \  將多出的月份的第一天從堆疊頂端
                                   刪除
: MONTH ( N -- )                \  印出一個月之月曆
      DUP
      HEADER DUP BLANKS         \  印表頭
      .DAYS ;                   \  印本月的天數

: CALENDAR ( YEAR --- )         \  印該年的全年月曆
      YEAR                      \  計算 JULIAN 和閏年
      13 1 DO  I MONTH LOOP     \  印 12 個月之月曆
      CR CR 64 STARS ;          \  印出最後之邊框
```

金城 註疏：

一個好的 FORTH 程式，讀起來應該似散文的流暢，詩般的美。

如果要列印出日曆打入：

```
PRINT 1992 YEAR CALENDAR
```

PRINT 是一個特別的 F-PC 指令，會把螢幕輸出轉換到印表表機輸出

金城 註疏：

一個好的程式設計家，會很清楚自己所面對的題目中有那幾個要突破的困難，並分析各種可能解決此問題的方案，其利弊得失詳加考慮後，才動手開始撰寫程式。並非是一腦糊塗，兩眼發呆，就能夠擺平問

題的。在此題中可詳加品味其思索問題的方式與簡潔明快的程式寫作風格。如果一下子看不懂那就多咀嚼幾次，以不怕挫折的心接受考驗與磨鍊，體會受一下 FORTH 愚公移山的精神。

練習一、 真正之 Julian 日期

在上面的例子，我們以 1950，1，1 爲 0 天來計算年,月,日這是因爲我們使用一個整數來代表日期，而其範圍受限在-32768 到 32767，大約是以包含 89 年，真正之 Julian 年月日，開始於西元前 4713 年 1 月 1 日是世界的開始(西方的觀念)，要表示如此大範圍的年月日，我們必須使用雙倍精密度整數，
-2,294,967,295 至 2,294,967,294 之範圍，Julian 日期才夠用。

```
: JULIAN-DATE ( DD MM YYYY -- d, 以 32 位元整數來計算 Julian 的值 )
        >R                      \   存 YYYY 於回返堆疊(return stack)
        DUP 9 + 12 /            \   1 月、2 月爲 0，其餘爲 1
        R@ + 7 * 4 / NEGATE     \   每一年取出 1.75 天
                                \    365.25 = 367-1.75
        OVER 9 + 12 / NEGATE
        R@ +
        100 / 1 + 3 * 4 / -     \   各世紀產生之閏年
        SWAP 275 9 */           \   這個月之前，本年的天數
        + +                     \   加 DD，爲本日之年天數
        S>D 1.721029 D+         \   加西元，元年一月一的 Julian 日數
        367 R> UM* D+           \   加上到今年的日數
        ;
```

有幾個處理雙倍精密度數的指令

```
S>D     ( n -- d )              \   將一個整數擴大爲雙倍精密度整數
D+      ( d1 d2 -- dSum )       \   將 2 個雙倍精密度數相加在
                                    堆疊(stack)的頂端
UM*     ( n1 n2 -- dProduct )   \   2 個整數相乘並傳回一個
                                    雙倍精密度的整數乘積
```

Forth 有另一個堆疊(Stack)，做爲呼叫副程式及返回位置之保存，稱爲回返堆疊(return stack)，其功能對初學的使用者是隱藏的而使用者也不須費神去使用它。然而回返堆疊(return stack)被用在暫存參數堆疊(Parameter Stack

)上之數是種常見的技巧，並且暫時將其在兩個堆疊(stack)間移動，將數目在參數堆疊(Parameter Stack)和回返堆疊(return stack) 之間移動的 Forth 指令是：

```
>R          ( n -- )    \  取出參數堆疊(Parameter Stack)頂端之數
                           ，將它移到回返堆疊(return stack)的頂端

R>          ( -- n )    \  取出回返堆疊(return stack)頂端之數
                           ，將它放回參數堆疊(Parameter Stack)

R@          ( -- n )    \  複製回返堆疊(return stack)頂端之數
                           放到參數堆疊(Parameter Stack)
```

金城 註疏：

請小心的使用迴返堆疊

(一)>R 與 R>必須要成對的使用

(二)在編譯模式下才能使用。

例題二、 生命的遊戲

生命遊戲是一個有趣的電腦程式可以在電腦的記憶體，模擬群體社會之成長和衰敗，假設電腦的記憶體是張平面圖，每一個記憶體不是空的就是有一個生命體，一個位置有無生命現象，都取決於鄰近位置之生命體，生活遊戲之規則：

1. 如果一個位置的 8 個鄰居包含 3 個生命體,就可以產生一個新生命體。
2. 如果 8 個鄰居包含之生命體，少於 2 個或多過 3 個，則此生命體死亡。

生命體死亡，因爲孤單(鄰居<2)或擁擠(鄰居>3)，當有 3 個活鄰居，假定一個父親、一個母親、一個牧師、一個新生命會產生。

以電腦螢幕爲圖，包含 25*80 之位置，一個 # 代表在一個位置上的生命體，一個空白指出在那個位置沒有生命，在記憶體裡一張圖配有 2048 位元組.只有 1920 個位置(24*80)，可在螢幕上顯示，記憶區是以 80 個連續 bytes 鄰接 80 個連續位元組之形式排列，所以 N 位置的鄰居位置是是 N+1、N-1、N+79、

N-79、N+80、N-80、N+81、N-81 而所有數字都是 2048 之餘數可用 2047 AND 一個數來獲得除以 2048 之餘數。

記憶區可從 PAD 之 2048bytes 指派，PAD 指出使用者能用之自由記憶體空間，事實上我們需要 2 個 2048 位元組區域，第一區貯存生命體之圖表，第二區貯存下一代生命體之分佈狀況，當計算一代之後，在第二區之圖表會複印至第一區。

金城 註疏：

如果要求除以 2 的幾次方的餘數，在 FORTH 的技巧中，我們會使用位元的罩斷(MASK)來求得，因爲一個除法指令的速度代價犧牲太大了，以 8086 爲例除法指令要 192 個 CPU 時脈而邏輯指令僅需 4 個時脈。當然，這種類似組合語言的挑剔態度，也是 FORTH 在速度上直逼組合語言的原因。

```
35 CONSTANT white                       \  ASCII #
32 CONSTANT black                       \  空白

: address1 ( n -- addr )                \  生命體之第一區
        2047 AND                        \  N 除以 2048 之餘數
        PAD +                           \  加位移至 PAD
        ;

: address2 ( n -- addr )                \  給下一代之第二區
        address1                        \  加 2048 至 addr1
        2048 +                          \  由 addr1 處理餘數的問題
        ;

: neighbors ( -- )
        2048 0                              \  審視全部圖表
        DO      I 1 + address1 c@           \  在 8 個鄰居內加生命體、右邊
                I 1 - address1 c@ +         \  左邊
                I 79 + address1 c@ +        \  左下角
                I 79 - address1 c@ +        \  右上角
                I 80 + address1 c@ +        \  下邊
```

```
                I 80 - address1 c@ +  \  上邊
                I 81 + address1 c@ +  \  右下角
                I 81 - address1 c@ +  \  左上角
                I address1 c@         \  生命體在這一位置嗎?
                IF      DUP 2 =       \  是的，2 或 3 個鄰居?
                        SWAP 3 = OR   \  如果是 2 至 3 個鄰居則活下來
                        IF 1 ELSE 0 THEN  \  太擁擠則死掉
                ELSE    3 =               \  空的位置
                        IF 1 ELSE 0 THEN
                                  \  如果 3 個生命鄰居給予出生
                THEN
                I address2 c!     \  貯存下一代
        LOOP
        ;

: refresh ( -- )                  \  複印下一代到這一代
        PAD 2048 + PAD 2048 CMOVE
        ;

: display ( -- )                  \  在螢幕顯示這一代圖表
        0 0 AT                    \  移動游標至左上角
        PAD 1920 0                \  審視 1920 位置
        DO      DUP C@            \  生命體在這?
                IF WHITE ELSE BLACK THEN    \  顯示它
                EMIT
                1 +               \  下一個位置
        LOOP    DROP
        ;

: init-map  ( addr -- )           \  從某一記憶體產生一個圖表
       2048 0                     \  注意 2048bytes 區
       DO     DUP C@ 1 AND       \  使用最小有效位元來檢查
              IF 1 ELSE 0 THEN   \  指派生命體
              I address1 C!      \  在我們的這一代圖表裡
              1 +
       LOOP   DROP
       ;

: generations ( n -- )        \  重覆N代
```

```
        0 DO    neighbors       \  計算下一代
                refresh         \  複印至這一代圖
                display         \  並顯示它
        LOOP
        ;

slow                    \  顯示每一個文字，使用 BIOS 的中斷來顯示
statoff                 \  隱藏表頭，將 F-PC 的狀態顯示暫時關掉
500 init-map            \  開始圖表，由記憶體中任選一塊記憶體，當
                           第一代的生命來源
10 generations          \  做 10 代
```

在此例中介紹之 Forth 新指令

```
CONSTANT        ( n -- )        \  定義新指令，當執
                                   行時傳回到 N
C@              ( addr -- char) \  從一個記憶體位置取一個 byte
C!              ( char addr --) \  貯存一個 byet 至記憶體位置
CMOVE           ( a1 a2 n -- )  \  從記憶體位址 a1 複製，N 個 bytes 至
                                   記憶體位址 a2
PAD             ( -- addr )     \  送回到一個自由記憶體緩衝區的位址
AND             ( n1 n2 -- n3 ) \  逐位元 AND 2 個 16 位元數字
OR              ( n1 n2 -- n3 ) \  逐位元 OR 2 個 16 位元數字
```

練習二、

想一個你喜喜愛的棋盤遊戲，並考慮電腦化的可能性，也許全盤電腦化非常困難，但是你應該找出一些方法使它部份電腦化，想些方法使電腦能幫助你棋藝進步。(丁陳博士本身就曾用 FORTH 寫了一個圍棋程式)

金城 註疏：

將 FORTH 看成一個雙堆疊結構的虛擬 CPU ，則 FORTH 的基本指令，便可以看成是此虛擬 CPU 的機器指令(其實，在真正的 FORTH 硬體 CPU 上，所有的 FORTH 指令均是以硬體階層直接執行，在那種 CPU 上沒有組合語，因爲 FORTH 就是最低階的機器語言了)，在 FORTH 的虛擬 CPU 上。

CONSTANT 可以看成 push Constant
VARIABLE 可以看成 push Address of Variable

也就是載入變數的有效位址
在 FORTH 中所有對變數的操作均有統一性，給予一地址在堆疊的頂端，要加、減、乘、除，要存、要取，要八位元、要十六位元、要三十二位元，皆由使用者自己選配使用，剛開始會不習慣，但久了，反而會喜歡這種簡潔有力的表達方式。

第 五 課　　數字的技巧

例題一、正弦和餘弦

三角函數之 SIN、COS 大都在繪圖函數功能裡遇到，在畫圓弧及其它用途上是常常用到的。通常使用浮點數字來做爲準確性和活動性範圍之計算，然而對在數字系統裡的製圖過程中整數範圍 -32678 至 32767 大致已夠用，我們應該學習使用十六位元的整數來計算 SIN 及 COS。

三角函數的 SIN 或 COS 的值域介於 -1.0 及+1.0 間，我們用十進位整數 10000 代表 1.0 來計算，使 SIN 和 COS 能以最大的精密度表出。
pi 是 3.1416 ，而 90 度角由 15708 取代，角度先轉換成 -90 度至+90 度間之範圍，然後再由-15708 至+15708 之間的數換算成弧度，由弧度計算 SIN、CON 之值。

Sin 和 Cos 之計算精準到 1/10000 ，此演算法首先由 John Bumgarner 發表於 Forth Dimension 第四卷第一期第七頁。

```
31415 CONSTANT PI
10000 CONSTANT 10K                          \ 比例常數
VARIABLE XS                                 \ 角度量的平方

: KN ( n1 n2 -- n3, n3=10000-n1*x*x/n2 在此處 x 表示角度 )
       XS @ SWAP /                          \  x*x/n2
       NEGATE 10K */                        \  -n1*x*x/n2
       10K +                                \  10000-n1*x*x/n2
       ;

: (SIN) ( x -- sine*10K, x 爲弧度乘以 10k 的值 )
       DUP DUP 10K */                       \  以 10K 爲 X*X 之比例
       XS !                                 \  將之存入 XS
       10K 72 KN                            \  最後項
       42 KN 20 KN 6 KN                     \  項 3，2 和 1
       10 K */                              \  乘 X
       ;
```

```
: (COS) ( x -- cosine*10K, x 爲弧度乘以 10k 的值 )
      DUP 10K */ XS !                  \  計算並存入 X*X
      10K 56 KN 30 KN 12 KN 2 KN       \  數個擴大形式
      ;

: SIN ( degree -- sine*10K )
      PI 180 */                        \   換算爲弧度
      (SIN)                            \   計算 Sin 的值
      ;

: COS ( degree -- cosine*10K )
      PI 180 */                        \   計算 cos 的值
      (COS)
      ;
```

金城 註疏：

在此課中，我們會發現 FORTH 的「整數哲學」，非常突出的要求以整數來解決其他語言使用實數才能解決的問題。一來是 FORTH 認爲要發揮硬體 CPU 的真實功能。如果，沒有實數的浮點運算器，就不用實數，而高度技巧來發揮整數的功能。這一點也說明了爲什麼人稱 FORTH 是屬於天才的語言，因爲要「多思」乃才能解決問題啊！

來測試此程式，鍵入:

```
90 SIN .                    \  9999 用整數雖不完全精準，但也
                               很夠用了
45 SIN .                    \  7070
30 SIN .                    \  5000
 0 SIN .                    \     0
90 COS .                    \     0
45 COS .                    \  7071
 0 COS .                    \  10000
```

例題二、 亂數

亂數通常使用在電腦模擬和電腦遊戲。
下列亂數產生器是刊載於 Leo Brodie 的 Starting Forth 一書中。

```
VARIABLE RND                            \ 種子
HERE RND !                              \ 初值化種子

: RANDOM ( -- n, a random number within 0 to 65536 )
        RND @ 31421 *           \ RND*31421
        6927 +                  \ RND*31421+6926, 再求除以 65536 之餘數
        DUP RND !               \ 更新種子，並在堆疊頂端留下新亂數
        ;

: CHOOSE ( n1 -- n2, a random number within 0 to n1 )
        RANDOM UM*              \ n1*random 得一 32 位元的值
        SWAP DROP               \ 拋棄低的 16 位元值
        ;                       \ 事實上等於是除以 65536 的商(向右
                                  移位 16 位元)
```

鍵入以下數個測試程式，看看輸出的結果。

```
100 CHOOSE .
100 CHOOSE .
100 CHOOSE .
```

結果被隨機分配在 0 和 99 之間的整數被產生。

練習一、
記得中國幸運餅嗎?寫一個資料庫程式選擇一個隨意的幸運詞給予幸運餅，當然你先要建立此資料庫。

金城 提示：
建立一個字串所組成的陣列，再以亂數當索引取出字串的內容。

金城 註疏：
在資訊科學的領域中，亂數的產生是一門與數學相關的大學問。如果產生的分佈不合預期的結果就不算是一個「合格」的亂數產生程式。一般可用蒙地卡羅法來觀察其輸出分佈的情形。如果要應用在

電腦模擬的領域中，則更是講究亂數的產生方程式。當然，如果看官本身對機率統計的知識「不知所云」，那就套個別人的公式用好了。反正「有」得「抄」比「沒得用」要好多了。

例題三、 平方根

N+1 的平方相等於 N 的平方與 2N+1 的總和，從 0 加 1,3,5,7...等，直到總數大於你要求根的整數，你所停止時的數就是平方根

```
: SQRT ( n1 -- n2, n2**2<=n1 )
        0                       \ 根之初值
        SWAP 0                  \ 設 N1 爲 limit
        DO      1+  DUP         \ 更新根
                2* 1+           \ ( 2n+1 )
        +LOOP                   \ 把 2N+1 加到總和，如果小
        ;                         於 N1 則再做，否則結束
```

有限迴圈結構

```
  limit  index   DO <repeat-clasue> ( inc ) +LOOP
( 極限   初值    DO   重覆的迴路體     累加數  +LOOP )
```

在重覆的迴路體部份與 DO...LOOP 很類似，其不同在於 +LOOP 是透過參數堆疊(Parameter Stack)頂端數字來增加指數，因此允許 LOOP 指數在執行計算時可使用非固定的任意數增加或減少迴返堆疊頂端的迴路索引值，+LOOP 結束迴路的方式爲當迴路索引值相等或大於極限時。

Wil 以非傳統方式使用 +LOOP 在這裡迴路索引值累加器使用，將連續的 2N+1 (奇數)相加，直至總和超過 N1，由此觀點，留在堆疊(Stack)的 N2 即是 N1 的平方根。

練習二、

用牛頓法來找平方根是電腦中非常普遍的方法，如果 r1 是 N 平方根的近似值，那麼更好的近似值是：

```
r2=(r1 + n/r1)/2
```

你能否以此方法寫出找平方根的程式

金城 提示：

r1 以 1 開始趨近，再將求出的 r2 當成 r1 傳入，反覆此程序，直到 r2 的平方等於 N)。

練習三、

整數的立方和 4 次方，會 16 位元整數的範圍很快用盡，使用連續整數求其 3 或 4 次方的數學方程式，可用連續的加法找出立方根，並與你所取得之根的整數比較，寫出新指令 Cubic Root 來找出任意正整數之立方根，4 次方根。

金城 註疏：

求平方根的 SQRT 程式是全球聞名的 FORTH 高手，Wil Baclon 先生的「天才作品」，發表於 1984 年在台灣交大舉行的 FORML 年會上。此程式短小辛辣爲曠世不二之作，每一個指令都用得出神入化，百鍊而成。如果看官能一眼看穿其奧妙絕倫之處，則可喜可賀，必能青出於藍而勝於藍。當然，如果看官們對此佳作不能品味，則只好在紙上好好的畫個堆疊圖來慢慢追，慢慢 K 了。此題也是學 FORTH 的最佳範例。

提示：此題的演算法爲一遞迴的定義，此線性的方式展開執行。

例題四、最大公因數 (GCD)

古代數學大師歐其理德(Euclid)的方法

```
If m>n,find GCD(n,m)
If m=0,GCD(m,n)=n
 Otherwise,GCD(m,n)=GCD(m,MOD(m,n))
```

金城 註疏：

此演算法爲最大公因數的遞迴(Recursive)定義。

將上列演算法改寫爲 Forth 我們得到

```
: GCD ( m n -- gcd )
      BEGIN   2DUP >              \  if m>n 交換 m 和 n
              IF SWAP THEN
              OVER                \  if m=0 離開 loop
      WHILE   OVER MOD            \  否則以 MOD [m，n]替換 n
      REPEAT                      \  重覆直至 m=0
      SWAP DROP                   \  因 m=0 所以拋棄 m ，而留下
      ;                              來的 n 就是最大公因數了
```

這是一個極好的例子來說明條件迴圈的用法

```
BEGIN <repeat-clause> (f)   WHILE     <true-clause>    REPEAT
BEGIN <迴路子句部份>  旗號   WHILE <爲真的才執行的部份> REPEAT
```

如果旗號經 WHILE 測試是真的，重覆迴路子句部份和爲真的才執行的部份，若旗號是 0(僞)，結束此迴路。

測試 GCD ，鍵入

```
123 456 GCD .
```

練習四、 最小公倍數

兩數的最小的公倍數的求法，可將兩數之積除以此二數之 GCD 寫一個新指令 LCM 將 2 個整數的 LCM 放回堆疊(Stack)

金城 註疏：

(一)剛學 FORTH 的人對堆疊的體曾不夠清晰，是最可怕的瓶頸之一，尤其是一個小程式中用了四、五個堆疊操作指令(如：DUP、SWAP、OVER、ROT、DROP 等)更如百里迷霧，魂飛魄散。此時，要咬緊牙、沈住氣、穩住心、定住神。然後用紙畫個堆疊圖追蹤一步，懂了，再追蹤一步，慢慢弄懂它。再深入學習體會用

堆疊浮沈的神韻，爲什麼 FORTH 的高手，如此厲害，沒有變數，只用兩個堆疊，就能解決這些一般性的問題呢? 其心法只有一句話：「在您的腦中，也建兩個活的堆疊，您就懂 FORTH 了」。

(二) FORTH 的 WHILE 迴路與其他語言，如 C、PASICAL 等均不相同，FORTH 的 WHILE-LOOP 是在迴路體的中間檢查是否離開迴路，而不似其他的高階語言是在最前面檢查。所以要小心。若能體會中間判斷的優點，才能善於使用此種迴路的彈性，而發揮其功能。

例題五、費邊數列

Fibonacc：是中世紀數學家 Leonardo 的筆名他提出此問題來計算一對兔子所能產生的後代數量。

把一對兔子放入由牆圍繞的地方，如果每個月，每對兔子會生出新的一對兔子，從第二個月之後所生的新兔子又能生出一對兔子，一年內會生出多少對兔子來?

順序會是：

1 1 2 3 5 8 13 21 34 55 89 144 233 337...

金城 提示：

費邊數列的遞迴定義爲 Fib(N)=Fib(N-1)+Fib(N-2)

我提供兩個相等的解答

```
: Fib1 ( -- , 印出所有小於  5000 的費邊數列 )
      1 1                          \  兩個開頭的 Fib 數字
      BEGIN   OVER U.              \  印出較小的數字
              SWAP OVER +          \  計算下一個 Fib 數
              DUP 50000 U>         \  如果數字太大離開
      UNTIL                        \  否則重覆
      2DROP                        \  拋棄此數字
```

```
        ;

: Fib2 ( n -- , 印出所有小於正整數 n 的費邊數列 )
        1                               \  最先的數字
        SWAP 1                          \  設定範圍
        DO      DUP U.                  \  印出目前 2 的數
                I                       \  下一個 Fib 數
                SWAP                    \  預備下一個到來
        +LOOP                           \  把現有的加至迴返堆疊的頂端
                                        \  重覆至 Sum > n
        U.                              \  印出最後 Fib 數
        ;
```

測試程式，鍵入:

```
Fib1
10000 Fib2
```

Fib1 是很明確的指令，將先前的兩個 Fib 數字相加，來取得下一個數字。

```
BEGIN <repeat-clause>  ( f )  UNTIL
BEGIN < 迴路體的敘述 >  旗號  UNTIL
```

當堆疊(Stack) 頂端項不是 0，重覆迴路體的敘述直至旗號是真的，則 UNTIL 結束迴路的執行。

Fib2 則使用 +LOOP 指令隱含的加法產物來計算下一個 Fib 數及與順便完成對極限是否到達的比較。此法使用了 Wil Baden 先生的程式技巧與風格來寫作程式，很辛辣刁鑽吧?

金城 註疏：

費邊數列又稱爲「美」的數列，在自然界中，如年輪，海螺、耳蝸、合聲基至於蒙娜麗沙的五官比例，都是費邊數列。在資訊科學的領域中，也是許多應用，如記憶體分割區塊的管理，外部檔案的排序、與搜尋.....等。都是應用了費邊數列的自然分佈現象，如果有興趣

在日本有專業的雜誌討論費邊數列。

練習五、二進位的等比級數

二進位等比級數類似費邊數列它的成員是：

1,2,4,8,16,32,64,128,256,512,1024,...

寫一個新指令 Binary Series 來顯示這一串數字

金城 提示：

可以用 2 來乘(也就是用向左的移位)，重覆的將結果轉出。

金城 感言：

多年以來，一直在 FORTH 的領域中掙扎，每次要寫一段小程式，就要「死」一次。因爲 Wil Badon 先生告訴我，寫 FORTH 的心境，就是一種不停的自我挑戰、自我超越與自我革命。一定要不斷的創造新的技巧，不停地追尋美的境界。才能達到「字」「字」珠璣，增一「字」則俗，減一「字」則怨的地步。(在 FORTH 的術語中，每個小程式都叫字(word))轉眼間，十年過去了，多少個嘔心瀝血的了夜爲了斟酌推敲「字」的得當與否而熬至天明。如今想起，依然觸目驚心。真的感受到了，FORTH 如詩、如詞、如歌、如畫，只有深深把 FORTH 推至了藝術的領域。也許，這一番話，你真的已經明白，也可能永遠都不會明白了。我承認，以資訊「工程」或「科學」的觀點來看這一種心境，是太過不合理的「要求」因爲大多數的人類是不可能做到大腦不必思考....」。難怪，大家喜歡 BASIC、Clipper、Dbase 了。但我記得沙士比亞說過：「人們營營苟苟的爲生活奔忙一世，但不曾真正的活過一天」。偉人與凡夫的差別應該在此吧?

第六課　終端機輸入和輸出

把第五課載入 (需要亂數產生器)

數字輸出的格式化指令群有

```
.       ( n -- )          \ 有正負號的印出指令
U.      ( u -- )          \ 無正負號的印出指令
D.      ( d -- )          \ 有正負號的雙倍精密度數印出
.R      ( n1 n2 -- )      \ 以 N2 個空間爲格式並靠右印出 N1
```

這些指令通常足夠應付螢幕上顯示數字的用途，讓程式設計者知道電腦發生什麼事，但顯示數字給一般的使用者所習慣的特定方式，格式化輸出，仍無法以上列指令做到。

Forth 提供另一群更精準的輸出指令，使你可以將數字格式化成任何你想要的模樣，我們將討論這些指令並用一些例子說明如何來建立特定的數字格式。

格式過程由堆疊(Stack)項端之雙倍精密度數開始討論，輸出的是一個格式字串，暫時儲存在文字緩衝區之下的自由記憶體裡，由指令 PAD 指向暫時儲存區的位址格式化的輸出字串是逆向結構，一次處理一個阿拉伯數字。當建立此字串的過程中任何 ASCII 字元可以隨時插入此字串以改良其可讀性。建立數字格式化之結構如下：

```
<#      ( -- )            \ 開始數字格式化過程。

#       ( d1 -- d2 )      \ d1 除以基底以商爲 d2 留在
                          \ 堆疊(Stack)上，餘數換成
                          \ 阿拉伯數字，加入輸出字串的尾部。

#S      ( d -- 00 )       \ 換算所有有意義的阿拉伯字加
                          \ 進輸出字串的尾部。

#>      ( d -- addr n )   \ 終結數的字串轉換，並留下此字串的地址和
                          \ 長度在堆疊的頂端以供 TYPE 指令印出結果。
```

```
HOLD        ( char -- )        \ 加入 ASCII 字元到輸出字串。

SIGN        ( n -- )           \ 如果 N < 0 加入記號"-"到輸出字串。

BASE        ( -- addr )        \ 記憶地址儲存目前的基底。

DECIMAL     ( -- )             \ 設定基底爲 10 (做爲 10 進制換算用)
```

控制數字換算的基數，可以隨意志改變，這種完全自由的彈性允許 Forth 使用者做有趣及有力的數字格式化應用。

金城 註疏：

剛開始接觸 FORTH 的使用者，會不習慣沒有現成的格式輸出入函數可用。因爲所有的高階語言都具備相當齊全的高階輸出入函數或指令，如 C 語言有 printf() scanf()。Pascal 有 read() write()..... 等等。但是這些現成的函數除了易學、易用的好處之外，並無自由的彈性可供程式設計師耍玩精心巧思的設計能力。靈活的指令。不但能發揮強而有力的轉換功能，做出叫人讚賞不已的應用技巧。還能讓使用者真正的了解電腦的工作原理，達到一通百通的效應。

例題一、 電話號碼

電話號碼包括區域碼、國際碼，可以用雙倍精密度數代替，以習慣的格式印出電話號碼，我們定義：

```
: Phone (d -- , 印出一個電話號碼，使用標準格式如 (415) 432-1230 )
        <#                      \ 開始輸出字串
        # # # #                 \ 轉算最後四個阿拉伯字爲字元的個格式
        45 HOLD                 \ 插入記號"-"
        # # #                   \ 轉算次三個阿拉伯字
        32 HOLD                 \ 加一個空白
        41 HOLD                 \ 加上右括弧
        # # #                   \ 加區域碼到格式化到字串中
        40 HOLD                 \ 加上左括弧
```

```
        #>                      \ 結束格式化
        TYPE SPACE              \ 印出輸出字串，並跳過一個空格
        ;
```

雙倍精密度數可以由鍵盤輸入，輸入數字串的任何地方可含一個句點，FORTH 的數字轉換系統便之此數字爲雙倍精密度的整數。印電話號碼顯示如上所示，鍵入：

```
415432.1230 Phone
```

正確的格式字串會印在螢幕上

例題二、 時間輸出

假設一天的時間可以由雙倍精密度整數代替，由午夜開始計算，以 1/100 秒爲單位輸出，格式設計成 HH：MM：SS.XX ，XX 表示一秒中的百分之一秒。

```
: Sextal ( -- )
        6 BASE !                        \ 設定基底爲 6
       ;

: :SS  ( d1 -- d2 , 將 d1 除以 60 並將其餘數(個位與十位兩字元)加入
         輸出字串 )
        #                       \ 以十進位換算其個位數
        Sextal #                \ 以六進位爲底換算下一個數(十位數)
        DECIMAL                 \ 恢復基底爲十進位
        58 HOLD                 \ 加"："到輸出字串
        ;

: Time ( d -- )
       <#                       \ 開始輸出字串
       # #                      \ 換算 1/100 秒
       46 HOLD                  \ 加" ・"到輸出字串
       :SS                      \ 換算秒
       :SS                      \ 換算分
       #S                       \ 換算小時
       #>                       \ 結束換算
```

```
        TYPE SPACE                  \ 印出結果
        ;
```

金城 註疏：

這些小程式都很實用，值得再三玩品。

從午夜到中午有 4320000 個百分之一秒，如果打入：

```
4320000. TIME
```

會看到 12：00：00.00 ，證明它會正確的轉換成格式化的字串輸出

例題三、 角度

爲了航海的目的，全世界的位置以經緯度來描繪，以 DDD：MM'SS''之形式，因爲沒有合適的字元作度數，我們以"："作爲度數，此格式只與時間稍爲不同，定義：

```
: Angle ( d -- , print angle in the DDD:MM'SS'' format )
        <#                          \ 開始輸出字串
        34 HOLD                     \ 把秒加上''
        #                           \ 秒以 十 進制數字轉換出個位數
        Sextal # DECIMAL            \ 秒以六進位制數字轉換出十位數
        39 HOLD                     \ 分加上 '
        :SS                         \ 換算分
        #S                          \ 換算度
        #>                          \ 結束格式化作業
        TYPE SPACE                  \ 印出結果
        ;
```

改變貯存於 BASE 之基數，Forth 使用者可以自由地從一個基底換算至另一個基底依據當時之需要，程式設計者可從十進制換算數字至 16 進制、 8 進制、 2 進制，反之亦然。數字換算可用以下的指令改變基數來處理。

例題四、 數字轉換之基數

```
DECIMAL
: OCTAL          8 BASE !  ;
: HEX            16 BASE ! ;
: BINARY         2 BASE !  ;
: RADIX36        36 BASE ! ;
: RADIX19        19 BASE ! ;
```

特別的基數底對特殊情形有時是很有用的，例如基數 36 ，可用來壓縮字母與數字字串，來適合緊縮的記憶體空間是很方便的，因爲它包含 10 個數字和 26 個字母。我喜歡的基數是 19 ，它在中國圍棋遊戲上非常好用。

在不同基數裡換算數字如以下的式子，你可以鍵入看看輸出的結果。

```
DECIMAL 12345 HEX .
HEX ABCD DECIMAL U.
DECIMAL 100 BINARY .
BINARY 101010101010 DECIMAL .
```

金城 註疏：

以 36 爲基底來儲存飛機的班機號碼爲例，可以達成兩個目的：
第一點、可以省空間，以 32 位元(4 個 Byte)來儲存短字串。
第二點、可以省時間。電腦很容易對 32 位元的整數來排序、搜尋。但對純粹的文字串則不然。這兩點使得程式好寫，速度又快，又省記憶體。

HP 計算機可以在十進制與 16 進制間換算，Forth 電腦有此功能，還有更多的字元與延伸。

例題五、 電報碼

```
RADIX36
: Message THE. WOLVES. ARE. COMING. ;
DECIMAL
```

如下可以知道訊息如何編碼和解碼

```
RADIX36 Message D. D. D. D.
DECIMAL Message D. D. D. D.
```

注意在 Message 裡的字是附有句點的，強迫它們在基底 36 裡換算成雙倍精密度整數，在基數 36 可表最大數字是 1Z141Z3 ，如同基數 36 之雙倍精密度數，可以表示任何 6 字元字母數字之字串。

終端機輸入和輸出(FORTH 的使用者介面)

以鍵盤輸入數字和指令在螢幕上顯示結果和其他資訊，控制輸入輸出最基本的 Forth 指令是：

```
KEY       ( -- char )      \ 接受按鍵傳回對應的 ASCII 碼
KEY?      ( -- f )         \ 按鍵後傳回真值旗標(按鍵了嗎?)
EMIT      ( char -- )      \ 將堆疊(Stack) 頂端字元之 ASCII 碼
                           \ 轉成文字，輸出到螢光幕
TYPE      ( addr n -- )    \ 從記憶體 addr 位置，
                           \ 顯示 N 個字長度的字串螢光幕
EXPECT    ( addr n -- )    \ 鍵入 N 個字貯存在記憶體中 addr 之處
                           \ 如按 Return 結束，輸入字元之
                           \ 總數被存在變數 Span 中
```

金城 註疏：

在 FORTH 的輸入指令中，並無文字串與數字(整數、長整數)的型態差別。因爲電腦的世界裡，輸出、輸入均以文字的單一型態來表示，而在計算與儲存時(指在記憶體與 CPU 中)的型態則純爲二進制的表示方式 。兩者之間截然不同，所以在輸入與輸出時才需做型態轉換的工作。所以在簡潔有力的原則下，FORTH 並不提供太多太複雜和太花俏的指令或函數。而僅提供有彈性的核心指令來組合調配出程式設計師所需的功能。這種「化繁爲簡」的特質，在 FORTH 中是統一不變的大原則，如果看官想清楚了這一點，自然就會神清氣爽，靜謐自得了。如果真的看不懂，或不會用，以我多年來教 FORTH 的經驗是您太嫩了。需要找一本好的組合語言的書來打個基礎再往下 K，自然就了解了。

例題六、 ASCII 字元表

大多數電腦、字元符號、數字符號、標點符號是以 8 位元的位元組表示，美國標準交換碼(ASCII)使用 32 到 127 的位元組值來代表可印出之符號，這些符號可列印在螢幕上或任何標準的印表機中，逾此範圍外，在不同的電腦裡會代表不同之符號，當超過上述之範圍 IBM-PC 會顯示繪圖符號到螢幕上，此練習是以格式化之表格來顯示正規的字元集和繪圖字元。

```
: Printable ( n -- n , 將不可見字元 ，轉換成空白的符號 )
        DUP 14 <                        \  7-13 是特殊格式(控制碼)用
        IF      DUP 6 >                 \  文字不能顯示出來所以
                IF DROP 32 THEN         \  用空白取代之
        THEN
        ;

: HorizontalASCIItable( -- )
        CR CR CR
        5 SPACES
        16 0 DO I 4 .R LOOP                 \ 顯示行頭
        CR
        16 0 DO                             \ 做 16 列
                CR I 16 * 5 .R              \ 列印左側的表值欄
                16 0 DO                     \ 一列印出 16 個字元
                        3 SPACES
                        J 16 * I +          \ 現在文字的值
                        Printable
                        EMIT                \ 印出堆疊頂端的字元
                LOOP                        \ LOOP 至下個字元
        LOOP                                \ LOOP 至下一列
        CR
        ;

: VerticalASCIItable( -- )
        CR CR CR
        5 SPACES
        16 0 DO I 16 * 4 .R LOOP        \ 顯示行頭
```

```
CR
16 0 DO                          \ 做 16 列
        CR I 5 .R                \ 印列頭
        256 0 DO                 \ 做 16 行
                3 SPACES
                J I +            \ 現在的字元的 ASCII 值
                Printable EMIT
        16 +LOOP                 \ 在行與行之間跳過 15 個字
LOOP                             \ 元直接印出 16 個字元
CR
;
```

鍵入 HorizontalASCIItable 或 VerticalASCIItable 來看看結果。

會以兩種不同形式將完整的 IBM-PC 字元集顯示給你，許多繪圖字元是有趣的，因爲它們允許在螢幕上繪出商業表格，這些指令是非常實用的，如果你想用特殊繪圖字元來組成匠心獨具的螢幕畫面。

金城 註疏：

這兩個小程式，仍可做進一步的最佳化，如乘以 16 與向左移位的效果相同，但快得多。另外要測 n2<n1<n3 可用 BETWEEN 指令會更靈巧。

例題七、 一封情書

這是一個很有趣的例子，錄自 Leo Brodie's 的 Starting Forth 一書 此程式允許你印一封邀請女友看電影的情書，輸入你的名字、她的名字、和她眼睛的顏色，LETTER 指令會印出這封信。

```
VARIABLE NAME 12 ALLOT
VARIABLE EYES 10 ALLOT
VARIABLE ME   12 ALLOT
```

金城 提示：

準備字串變數所需的空間，其大小爲 12

加上變數 Name 本身的 2Byte 共 14Byte

```
: ENTER ( addr n -- , 接受一個長度爲 n 的字串，並將之放到記憶體
                                                addr 的地方 )
        2DUP BLANK                      \ 清除此記憶體爲空白字串
```

```
        EXPECT                          \ 等待輸入一行字串直到 N 個
        ;                               \ 字元或 Enter 鍵被按下爲止

: VITALS ( --, 讀取自己與對方的姓名及對方眼睛的顏色 )
         CR ." Enter your name: "
         ME 14 ENTER                    \ 輸入你的名字
         CR ." Enter her name: "
         NAME 14 TNTER                  \ 輸入她的名字
         CR ." Enter her eye color: "
         EYES 12 ENTER                  \ 輸入她眼睛的顏色
         ;

: LETTER ( -- )
        CR CR CR CR
        ." Dear "
        NAME 14 -TRALLING TYPE
        ." ," CR
        ." I go to heaven whenever I see your deep "
        EYES 12 -TRAILING TYPE
        ."  eyes. Can " CR
        ." you go to the movies Friday?"
        CR 30 SPACES
        ." Love,"
         CR 30 SPACES
        ME 14 -TRAILING TYPE CR
        ." P.S.  Wear something "
        EYES 12 -TRAILING TYPE
        ."  to show off those eyes!"
        CR CR CR
        ;
```

先鍵入 VITALS 輸所有所需資料，然後打入 LETTER 你可以看見這封十分感人的信。

金城 註疏：

FORTH 的 VARIABLE(變數)不似其他語言的變數有左值(有效地址)與右值(內含值)的區別，僅單純的將一變數的有效地址放在堆疊上，供下一個指令來使用，因此反而類似其他語言的指標型態了。一

個地址區域可以很自由的存放您所需要的物件，可以是整數、長整數、實數或字串。甚至可以是函數或陣列的起始位址。這種自由的調度能力，全憑使用者的規劃與組織能力來發揮，它融合了 C 語言中 UNION(聯合)與無型態指標(VOID)的雙重特質，威力無窮。

在此例中，介紹幾個處理字串和記憶體的重要 Forth 指令

```
ALLOT       ( n -- )              \ 在變數之後的記憶體中，
                                  \ 配置 N 個 bytes，建立一
                                  \ 個陣列來貯存字串或數值
FILL        ( addr n char -- )    \ 從記憶體 addr 連續 Nbyte 長
                                  \ ，用 Char 填入該記憶體陣列
-TRAILING   ( a n1 -- a n2 )      \ 修改字串長度 N1 ，消
                                  \ 除尾部的空白字元
```

注意在使用 Expect 指令輸入新資料之前，字串陣列 ME.NAME,EYES 必須先清除成空白字串，因爲我們不知道有多少個有效字元會輸入這些陣列。在輸入新資料後，可以使用 -TRAILING 來除去尾部的空白而取得較短的有效字串。

例題七、 數字輸入

很多場合必需請使用者輸入數字符號所構成的字串，該字串被接受後，我們必須將它換算成數字，並放進資料堆疊給隨後的指令使用，轉換一個數字字串成爲二進位數的基本指令是 CONVERT。

```
CONVERT     ( d1 addr1 -- d2 addr2 )  \ 由 addr1 處開始換算字串成雙倍
                                      \ 精密度整數 然後加到 d1 ，而成
                                      \ 爲總和 d2 留在堆疊(Stack)上而
                                      \ addr2 指在字串裡第一個非數字
                                      \ 處。
```

向使用者要求輸入一個數字的例子，如下:

```
: GetNumber ( -- n )
        CR ." Enter a Number: "       \ 顯示訊息
        PAD 20 EXPECT                 \ 取得字串
```

```
    0 0 PAD 1 - CONVERT             \ 換算字串至雙倍精密度數
    2DROP                           \ 留一個單整數在
    ;                               \ 堆疊(Stack)上
```

金城 提示：

如果將 2DROP 改成 DROP 可得到一個雙位精密度的整數。

金城 註疏：

在 FORTH 中有一特殊的執行機構稱爲定義詞 CREATE DOES> ，通常 ALLOT 是使用在該處。定義詞是 FORTH 獨有的特質，是三個時區(定義時間 編譯時間 執行時間)的程式設計，也是 FORTH 的使用者自定的執行機構，奧妙無窮，爲 FORTH 中最精彩，也最需要功力的一段。對使用者(程式設計師)而言，那才是最有挑戰性的一次考驗，是否能學會 FORTH，定義詞是任督二脈打通與否的重要關鍵。

用這個實用性指令可以寫個遊戲" 猜一個數字 "

```
: InitialNumber ( -- n , set up a number for the player to guess )
      CR CR CR ." What limit do you want?"
      GetNumber                     \ 要求使用者輸入一個數字
      CR ." I have a number between 0 and " DUP .
      CR ." Now you try to guess what it is."
      CR
      CHOOSE                        \ 選擇一個亂數
      ;                             \ 介於 0 和極限之間

: Check ( n1 --, 讓遊戲者去猜這個亂數，如果猜對了，就會結束
                                                遊戲的迴路 )
      BEGIN   CR ." Please enter your guess."
              GetNumber
              2DUP =                \ 相等了嗎?
              IF      2DROP         \ 清除堆疊上的兩個整數
                      CR ." Correct!!!"
                      EXIT
              THEN
              OVER >
```

```
                IF      CR ." Too High!"
                ELSE    CR ." Too low."
                THEN    CR
        0 UNTIL                         \ 永遠重複的無限迴路
        ;

: Greet ( -- )
        CR CR CR ." GUESS A NUMBER"
        CR ." This is a number guessing game. I'll think"
        CR ." of a number between 0 and any limit you want."
        CR ." (It should be smaller than 32000.)"
        CR ." Then you have to guess what it is.
        ;

: Guess ( -- , the game )
        Greet
        BEGIN   InitialNumber               \ 準備要被猜的整數
                Check                       \ 讓遊戲者猜數字
                CR CR ." Do you want to play again? (Y/N) "
                KEY                         \ 讀入一鍵
                32 OR 110 =                 \ 如果是大寫 N 或小寫 n
        UNTIL                               \ 則結束遊戲
        CR CR ." Thank you.  Have a good day."  \ 致謝並請安
        CR
        ;
```

鍵入 Guess 開始遊戲，電腦讓使用者娛樂一會兒。
注意無限迴圈之使用。要有離開的方法，否則迴路會永遠執行跳不出來。

```
 BEGIN <repeat-clause> ( f ) UNTIL
 BEGIN < 迴路中的重複部分 > ( f ) AGAIN
```

以指令 Exit 跳出此無限迴圈，即跳過之後的剩餘指令直到 "；" 它的功能爲結束此定義繼續下一個定義的執行。

金城 註疏：

其實 EXIT 是動了迴返堆疊的手腳，提前結束此高階定義的執行，要小心使用，否則當了機，那就弄巧成拙了。

金城 註疏：

不知諸位看官對 FORTH 的體驗爲何? 比 BASIC 簡單、比組合語言難、比 C 語言深奧、比 Pascal 流暢。我認爲這些答案都對，而體受自然不同。學 FORTH 一如學禪學佛，師父領進門，修行看個人，能否往昇西天極樂，就看慧根、悟性與造化了。

---------1992 年 5 月 19 日子夜 ------

第貳篇

基礎篇

第一課　FORTH 緒論

1.1　FORTH 概說

FORTH 的每件事物都是一個字(詞)。

FORTH 的每一個字必須以空格(space)隔開。

FORTH 的每一個字儲存在一字典裡。

FORTH 的每一個字不是可直譯的就是可編譯(COMPILER)的。

FORTH 的每一個字在直譯的模式中才可執行。

FORTH 的每一個字在編譯的模式僅是儲存在字典裡。

將 FORTH 詞串連在一起，以形成 FORTH 片語。

如果你鍵入一個 FORTH 詞，並且按下 <Enter> 鍵，它就會執行。
(直譯模式)

如果你鍵入一個數字 (例如：6) ，並且按下 <Enter> 鍵，這數字便以一個 16-bit 含正負號的整數，被儲存在堆疊(stack)上。

FORTH 廣泛地使用堆疊(stack)來傳遞一個一個的參數，這意思是說 FORTH 對變數的需要性已大幅度地減低了。

在 FORTH 中，你可以定義新的文字(由那些已被定義過的 FORTH 詞所組成)，而將之納入 FORTH 字典中的一部分，且像其他任何的 FORTH 詞一樣，可以被使用。

金城 註疏：

一、"FORTH" 在國外電腦界曾被人們如下定義：

一種幻美如詩，深邃如詞的電腦語言。

一種精緻小巧的作業系統(多使用者與多功作業能力的即時系統)。

一種直譯程式又是一種編譯程式，也是組譯程式。
一種比 C 語言更快，比組合語言更小的高可攜性發展測試環境。
一種雙堆疊(stack)結構的中央處理單元(CPU)。
一種跨時代的電腦軟體工程革命。
一種白癡憎恨，天才喜歡的鬼玩意。
一種孤獨的執著，眷戀的錯誤。
一種結合了老子，禪宗與太極運轉生生不息的"道"。
一種狂熱的宗教，一種唯我獨尊的無上心法。

二、在大陸 FORTH 被翻譯成"多思"，已推廣有八年之久了，主要是航天部在使用，近年來，中共也舉辦了全中國的 FORTH 檢定考試。

1.2 FORTH 算術

FORTH 運用堆疊(stack)來執行算術的運算，使用的是反波蘭式(POSTFIX)這套符號。

包括 F-PC 在內大部分 FORTH 均將 16-bit 的數值儲存於堆疊(stack)中，像 Macforth 這類 32-bit 的 FORTH ，則儲存 32-bit 的數值存在堆疊(stack)中。因此，堆疊(stack)中的數值會在十六位元的 FORTH 中佔二個位元組；在三十二位元的 FORTH 中佔四個位元組。

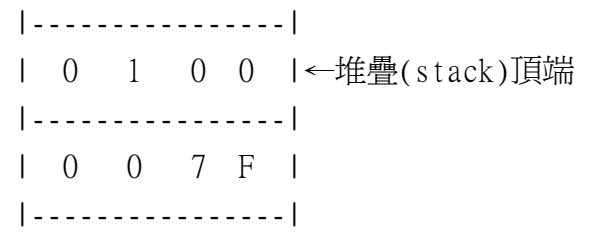
當你鍵入一個數字，它便被安置在堆疊(stack) 中，你可以在任何基底(base)上鍵入數字，稍後我們將會看到如何改變這基底(base)。

開機設定的基底(base)是十進位(DECIMAL)的；因此，若你鍵入 "35" ，十六進位數值的 23h (字尾 h 表示一個十六進位數值) 會以下列形式被儲存在堆疊(stack)中：

```
|------------|        |------------------------|
| 0  0  2  3 |        | 0  0  0  0  0  0  2  3 |
|------------|        |------------------------|
16 位元堆疊值                32 位元堆疊值
```

如果你鍵入二個數字，中間隔一空格，它們便會分別被儲存在堆疊(stack)中，例如：如果你鍵入 127 256

這二個十六進位數值 7Fh 和 100h 將會以下列形式儲存在堆疊(stack)中：



鍵入 .S 會顯示堆疊(stack)出的內容，但並不會破壞堆疊(stack)。
(檢查堆疊上的現況)

```
127  256  .S  127 256  OK
```

" OK " 這個字是 FORTH 的提示字。

數值被儲存在堆疊(stack)中，如同帶正負的二階補數。

因此，對 16 位元的 FORTH 而言，存在堆疊(stack)中的數值可以在
—32,768 至 +32,767 之間變化。

對 32 位元的 FORTH 而言，儲存在堆疊(stack)中的數值可以在
—2,147,483,648 至 +2,147,483,647 之間變化。

金城 註疏：

其實在電腦的二進位儲存模式中，由 16 個 1 所代表的意義爲何，要看人來決定，如果是帶正負號(Signed)的整數，則代表 -1。如果是當無正負號(Unsigned)的整數，則代表 65535。你可以做一行小的測試，輸入 -1 DUP ・ U. 按<Enter>看看結果是不是 -1 和 65535 。就似錄音機的轉數表一樣，999 可看成由 000 倒轉一圈的結果。相同的也可以看成由 000 正轉 999 圈的結果一樣。結論是電腦整數本無正負之分，是人類的定義有正負罷了。(風幡不動，賢者心自動)

1.3 FORTH 算術運算指令

FORTH 字 "・" (點 唸 DOT)會將在堆疊(stack)頂端的數值印出
(1 / 1 in/out)

7 9 ・・ 會印出 9 7

跳行回返的鍵值通常會被 FORTH 所忽略，並視之爲一個空白處。這可使得你的程式較易閱讀。藉著以" 垂直 "的風格來寫你的程式，你可以將" 堆疊圖 " 在你程式的右邊指出，只需緊跟著一個倒斜線 " \ " 。

在一行中任何緊跟著一個倒斜線的符號皆被視爲是一個註釋，而被忽略。

例如：要舉出上述例子中每一頁中堆疊(stack)裡的符號時，我們可以這樣寫

```
                    堆疊圖      (堆疊的最上端)
7                   \  7
9                   \  7  9
・                  \  7
・                  \
```

注意 " ・" (點)指令用掉上了堆疊(stack)上的數值。

.本文 p4

FORTH 的字 + (加號)會將堆疊(stack)頂端的二個數值相加，並且將結果留在堆疊(stack)中。(2 / 1 in/out)

7 9 + ・會印出 16

```
7                   \  7
9                   \  7  9
+                   \  16
・                  \
```

FORTH 字 " - " (減號)將堆疊(stack)頂端的第二個數值減去堆疊(stack)上端的第一個數值，並且將差留堆疊(stack)中。(2 / 1 in/out)

8 5 - ・ 會印出 3

```
8                   \  8
```

```
5              \  8  5
-              \  3
.              \
```

FORTH 的字 " * " (乘號)(星號)會將堆疊(stack)頂端的二個數值相乘，並且將乘積留在堆疊(stack)上。 (2 / 1 in/out)

4 7 * .會印出 28

```
4              \  4
7              \  4  7
*              \  28
.              \
```

FORTH 的字 " / "(除號)會將堆疊(stack)中頂端的數值去除第二個數值，並且將整數的商留在堆疊(stack)頂端。(2 / 1 in/out)

8 3 / .會印出 2

```
8              \  8
3              \  8  3
/              \  2
.              \
```

1.4 堆疊(stack)的操作的指令

FORTH 不用區域變數，而在堆疊(stack)上自己調整所需用的數值

堆疊圖 (之前--之後)

之前 = 堆疊(stack)上在詞執行之前所需的參數(PARAMETER)
之後 = 堆疊(stack)上在詞執行之後留下來的返回值(RETURN VALUE)

DUP 複製堆疊(stack)中頂端的元素。(1 / 2 in/out)

5 DUP . .會印出 5 5

```
5              \  5
DUP            \  5  5
```

```
·                   \  5
·                   \
```

SWAP (N1 N2 -- N2 N1) 交替互換堆疊(stack)頂端的二個元素。
(2 / 2 in/out)

3 7 SWAP ··會印出 3 7

```
3                   \  3
7                   \  3  7
SWAP                \  7  3
·                   \  7
·                   \
```

DROP (N--) 拋除堆疊(stack)頂端的元素。(1 / 0 in/out)

6 2 DROP · 會印出 6

```
6                   \  6
2                   \  6  2
DROP                \  6
·                   \
```

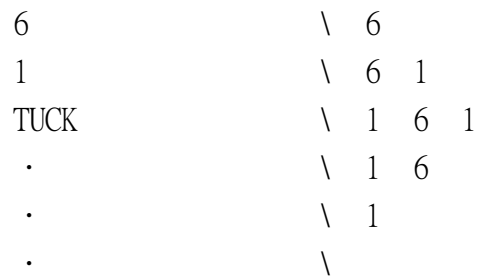
OVER (N1 N2 -- N1 N2 N1) 複製堆疊(stack)中第二個元素。
(2 / 3 in/out)

6 1 OVER · · ·會印出 6 1 6

```
6                   \  6
1                   \  6  1
OVER                \  6  1  6
·                   \  6  1
·                   \  6
·                   \
```

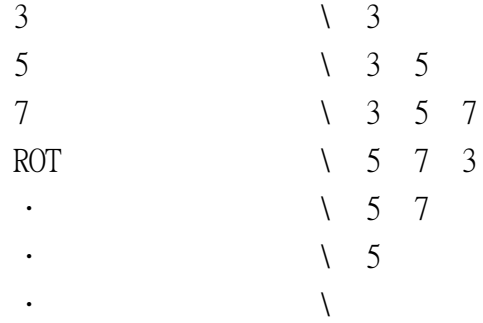
TUCK (N1 N2 -- N2 N1 N2) 複製堆疊(stack)端頂元素插在第二個元素之下。這就是 SWAP OVER 或 DUP -ROT 之最佳化。(2 / 3 in/out)

6 1 TUCK ・ ・ ・會印出 1 6 1



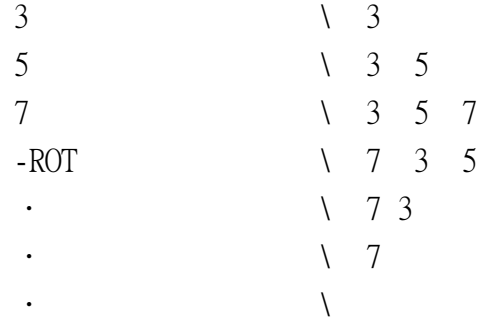
ROT (N1 N2 N3 -- N2 N3 N1) 迴轉堆疊(stack)上的第三個元素，使第三個元素變成第一個元素 (3 / 3 in/out)

3 5 7 ROT ・ ・ ・ 會印出 3 7 5



—ROT (N1 N2 N3 -- N3 N1 N2) 倒轉堆疊(stack)上的第一個元素，使第一個元素反轉至第三個位置。此爲 ROT ROT 的最佳化。
(3 / 3 in/out)

3 5 7 —ROT ・ ・ ・會印出 5 3 7



NIP (n1 n2 -- n2)
從堆疊(stack)中移走第二個元素，這相當於 SWAP DROP
(2 / 1 in/out)

6 2 NIP · 會印出 2

```
6                \ 6
2                \ 6 2
NIP              \ 2
·                \
```

2DUP (N1 N2 -- N1 N2 N1 N2)

複製頂端二個元素在堆疊(stack)上 (通常用來複製一對 16 位元或一個 32 位元的數)(2DUP 爲 OVER OVER 的最佳化) (2 / 4 in/out)

2 4 2DUP ·S 會印出 2 4 2 4

2SWAP (N1 N2 N3 N4 --N3 N4 N1 N2)

將堆疊(stack) 頂端第一、二個元素與第三、第四個元素互換。
(爲>R -ROT R> -ROT 之最佳化) (4 / 4 in/out)

2 4 6 8 2SWAP ·S 會印出 6 8 2 4

2DROP (N1 N2 --)

將頂端二個元素自堆疊(stack)中移走 (2 / 0 in/out)

2 4 6 8 2DROP ·S 會印出 2 4

PICK (N1 -- N2)

從堆疊(stack)頂端複製在 N1 位置的值(不算 N1)堆疊(stack)，頂端相當於 N1 等於 0。 (N / N in/out)

0 PICK 與 DUP 相同
1 PICK 與 OVER 相同

2 4 6 8 2 PICK ·S 會印出 2 4 6 8 4

ROLL (N --)

將在 N 位置(不算 N)的值旋轉至堆疊(stack)頂端。(N / N-1 in/out)

1 ROLL 與 SWAP 相同
2 ROLL 與 ROT 相同

2 4 6 8 3 ROLL ．S 會印出 4 6 8 2

金城 註疏：

一、PICK 與 ROLL 最好少用，因效率不佳且難以閱讀了解，且一個稍具功力的程式設師，其堆疊一般皆控制在四到六層之內，所以根本用不到 PICK 與 ROLL 兩詞。

二、

```
|---------|   <---堆疊(stack)頂端
|   11    | 0
|---------|
|   14    | 1     在 ROLL 處的係數 1 即爲第二個值 14
|---------|
|   28    | 2
|---------|
```

三、堆疊(stack)運算指令基本型態有四類：

(一) DUP、OVER 皆用來複製產生足夠的運算元。

(二) SWAP、ROT 皆用來調整運算元到適當的位置，以便使處理正確無誤。

(三) DROP 用來去除堆疊上多餘的贅物，以維持堆疊的平衡。

(四) >R、R@、R> 用來取深度超過三層以上的運算元。

(五) 所有其它的堆疊操作指令，均可以此四類來區分和使用之。

1.5 其他的一些 FORTH 常用字

MOD (N1 N2 -- N3)

將 N1 除以 N2 ，留下餘數 N3 在堆疊(stack)上。(2 / 1 in/out)

8 3 MOD ． 會印出 2

/MOD (N1 N2 -- N3 N4)

將 N1 除以 N2，留下商數 N4 在堆疊(stack)頂端，餘數 N3 在堆疊(stack)的第二個位置。(2 / 2 in/out)

10 3 /MOD ．S 會印出 1 3

MIN (N1 N2 -- N3)
留下 N1 和 N2 中較小的數在堆疊(stack) 上。(2 / 1 in/out)

8 3 MIN ． 會印出 3

MAX (N1 N2 -- N3)
留下 N1 和 N2 中較大的值在堆疊(stack) 上。(2 / 1 in/out)

8 3 MAX ． 會印出 8

NEGATE (N1 -- N2)
改變 N1 的正負號。 (1 / 1 in/out)

8 NEGATE ． 會印出 -8

ABS (N1 -- N2)
留下 N1 的絕對值在堆疊(stack)上。 (1 / 1 in/out)

-8 ABS ． 會印出 8

2* (N1 -- N2)
以執行一個算術左移運算會將 N1 乘以 2 (1 / 1 in/out)

8 2* ． 會印出 16

這相當於 8 2 * 但是快的多也精簡的多。

2/ (N1 -- N2)
以執行一個算術右移運算，會將 N1 除以 2。(1 / 1 in/out)

8 2/ ． 會印出 4

這相當於 8 2 / 但是快的多

U2/ (N1 -- N2)
執行一個 16 位元邏輯右移運算。(1 / 1 in/out)

40000 U2/ ． 會印出 20000
但 40000 2/ ． 會印出 -12768

8* (N1 -- N2)
執行一個 3 次的算術左移運算，會將 N1 乘以 8。(1 / 1 in/out)

7 8* ． 會印出 56

這相當於 7 8 * 但較快些

1+ (N1 -- N2)
將堆疊(stack)頂端元素加 1 。(比 1 + 快些)(1 / 1 in/out)

1- (N1 -- N2)
將堆疊(stack)頂端元素減 1。(1 / 1 in/out)

2+ (N1 -- N2)
將堆疊(stack)頂端元素加 2。 (1 / 1 in/out)

2- (N1 -- N2)
將堆疊(stack)頂端元素減 2。 (1 / 1 in/out)

U/16 (U -- U/16)
U 代表一個不帶正負號的 16-bit 整數，以執行一個 4-bit 邏輯右移運算，會將未帶正負號的整數 U 除以 16。 (1 / 1 in/out)

金城 註疏：
in/out 用來提供使用者參考，每一指令(word)在堆疊上(之前/之後)的變化，要精通 FORTH ，第一要務在"維持精準的堆疊平衡"留下垃圾會使堆疊爆掉(當機)。不足時，會使所有的運算結果亂掉。

1.6 冒號定義(高階程序)

可以藉著使用下列形式的 FORTH 的詞 " : "(冒號)來組成其他的 FORTH 的用詞。

定義你自己的 FORTH 的詞：

```
: (程序名稱) --- --- --- ---  ;
```

上列中，冒號 " : " 開始定義，<程序名稱>是你的 FORTH 詞的名字。
" --- --- --- --- " 是構成你所定義的 FORTH 的其他詞。

分號 " ; " END 結束整個定義。

舉例：如果你不喜歡將堆疊(stack)頂端之值印出的這個點"•"指令，你可以將它重新定義爲 " == "(我們將使用雙符號，因爲單等號 " = " 已經是一個 FORTH 詞了)

備註：

小括號"( )"是一種 FORTH 文字，將在小括號中的文數字當作一種對堆疊(stack)變化的註解。因此 " (" 的後面必須空一格。

COMMENT：--- COMMENT； 將 COMMENT：開始到 COMMENT；之間都括弧起來。

當你鍵入 FLOAD LESSON01 所有在 COMMENT：和 COMMENT；之間的文字都會被當成一個註解的被忽略。不過，下列的冒號定義會藉著被載入字典中而被編譯。你可以自將它們一一鍵入，或者僅鍵入 FLOAD LESSON01

```
: == ( n-- )    \ 印出堆疊頂(stack)端的值
   . ;
```

鍵入這一個冒號定義，然後試試看。
輸入

```
5 7 + ==
```

```
: SQUARED ( N -- N ** 2 )   \  計算 N 的平方
        DUP * ;
```

試著輸入

```
5 SQUARED  ==
3 SQUARED  ==
```

```
      7 SQUARED  ==

: CUBED  ( N --N**3 )     \  計算 N 的立方
         DUP       \  N N
         SQUARED   \  N N ** 2
         * ;       \  N ** 3
```

試著輸入

```
      3  CUBED  ==
      5  CUBED  ==
     10  CUBED  ==
```

金城 註疏：

耐心而詳實的追蹤每個指令(word)執行前後的堆疊變化圖，爲程式除錯與了解 FORTH 程式的唯一大道。

下列爲兩個常用的 FORTH 用字：

```
CR      ( -- ) ( " 跳行迴轉 " )
```

在螢幕上產生一個迴轉和跳行的動作

```
·"    ( -- )  ( " DOT-QUOTE " )
```

印出一個被夾在 ·" ------- " 中的字串。

定義下列的字

```
: BAR ( -- )            \ 印出一個星狀橫條
     CR ·" ***** " ;
```

鍵入

```
         BAR

: POST ( -- )            \ 印出一個星狀直條
     CR ·" * "
```

```
    CR ." * " ;

  鍵入
       POST

: C   ( -- )          \ 印出一個字母 C
     BAR
     POST
     POST
     BAR ;

  鍵入
       C

: E   ( -- ©          \ 印出一個 E 字母

     BAR POST
     BAR POST
     BAR ;

   鍵入
       E
```

提示：

FORTH 的新詞是由從前所定義過的詞來組成，這是 FORTH 的工作方式。新的更有威力的詞連續的被定義出來。當你完成工作時，整個主程式是最後一個定義的詞。你所定義過的詞，是存在 FORTH 的字典中(程式館)和原先定義過的詞在一起，這些新定義的字就變成 FORTH 語言的一部份。使用起來就像是原先定義的字一樣。(其他語言均有保留字，FORTH 沒有) FORTH 的解釋機構不會將原來的詞與你自己所定義的詞做差別性的對待和執行。這也就是說，每一個 FORTH 應用程式事實上是在設計一個解決特定問題的應用語言給使用者使用。

金城 註疏：

一、在寫作 FORTH 程式的過程中，你是用 FORTH 的語言(字)，對電腦說話，這些話(字)組成一個個的句子(冒號定義)。重點在於，這些句子你懂、電腦也懂；如果你自己不懂、電腦更不會懂。

二、FORTH 的精神在於(KISS : Keep it Simple and Stupid，保持問

題的單純性)。不要將其複雜化。

三、FORTH 有自己的 CPU 硬體結構。所以，不怕這種一層叫一層的副程式呼叫。

四、一個程式(冒號定義)不要太長，否則不易專心思考問題容易出錯。

練習 1.1

藉著鍵入左上方(t l)及右下方(b r)的坐標可定義一長方形，讓 X 坐標從左往右增加， Y 坐標從上往下增加。定義三個 FORTH 詞，面積、周圍、中心，就可計算出面積、周邊、和長方形的中心，只須按下列方法將上、左、下、右四個值輸入堆疊(stack)中。

```
AREA    ( t l b r --- 面積 )
CIRCUM  ( t l b r --- 周長 )
CENTER  ( t l b r --- 中心位置 )
```

使用下列二組上、左、下、右的數值，來測試你的三個詞：

上	31	10
左	16	27
下	94	215
右	69	230

第二課　　使用 F-PC

2.1 使用 SED 來編輯檔案

螢幕式編輯器是用來編寫你的程式和儲存它們進入磁碟中。

例如：寫練習 1.1 的作業應先進入 F-PC 看到 OK

再鍵入　SED HW1

這將會開一個新當名叫 HW1.SEQ 所有的 FORTH 程式用 " SEQ " 作副檔名，它會自動的加在你的檔名後面。

程式的風格在第一行使用一個反斜線 " \ " 跟隨其後的是程式的名稱。
這第一行的內容會在你用 FPRINT HW1 時印在每一頁的頁頭上。在第二行鍵入一個反斜線，再鍵入 ALT--O P ，會將當時的日期和時間附加在檔案上。

現在練習鍵入下列的程式

```
: SIDES     ( T  L  B  R -- R-L  B-T )
            ROT           \ T  B  R  L
            -             \ T  B  T-L
            -ROT          \ T-L  T  B
            SWAP - ;      \ R-L  B-T

: AREA      ( T  L  B  R -- AREA )
            SIDES * ;

: CIRCUM    ( T  L  B  R -- CIRCUM)
            SIDES + 2* ;

: CENTER    ( T  L  B  R -- XC  YC )
            ROT           \ T  B  R  L
            +             \ T  B  R+L
            2/            \ T  B  XC
            -ROT          \ XC  T  B
            + 2/ ;        \ XC  YC
```

注意" SIDES "是一個中間產物被定義來留下(左 — 右)及(底 — 頂)的值在堆疊(stack)上，" SIDES "這個字是在面積及周長的定義中被使用。

F-PC 對大小寫沒有編譯上的差異，這意思是你可以交替地使用大寫或小寫的字母，我們一般會依循傳統，將我們自己的新字定義用小寫來定義，而將我們在定義時所使用到的 F-PC 的原定義用詞用大寫來輸入，這使得一個定義較容易被辨識出那些是 F-PC 的字，那些是我們自己先前所定義的新字。

也需注意到在定義 "SIDE" 和 "CENTER" 時，我們將堆疊圖當做註解顯示在每一行的右邊；當大量的堆疊操作在連續進行的時候，你就會發現這非常有用。

SED 編輯器的全部編輯能力，在第四章 F-PC 的使用者手冊中會有描述。
在 F-PC 程式的磁片中，你可以藉鍵入一些字而將第四章印出來。

鍵入 TYPE CHARPTER4.TXT

一旦你完成進入這程式的動作，你可以藉著按" ESC "，然後按<Enter>來離開 SED ；此時你可以編輯另一個檔案，只需鍵入檔名即可，若此時你不想編輯另一個檔案，按" ESC "和<ENTER>。你現在有 " OK " 提示你在 F-PC 中，這時你大致上可以載入並執行你的程式。不過 SED 編輯器還沒有釋放在編輯你的程式時所佔用的記憶體，這時你要載入較大程式，就會有麻煩。所以，我將程序修改爲鍵入 BYE ，而暫時離開 F-PC ，然後鍵入 F ，重新進入 F-PC 。這時你可以載入任何大小的檔案，而不會發生問題。

2.2 載入並執行你的程式

要載入在 HW1.SEQ 檔中的程式，你只需鍵入 fload hw1 <ENTER>
如果你沒有做 HW1.SEQ 這個檔，你也可以只藉載入本課來完成同樣的事。

鍵入 fload lesson02 <ENTER>。

這是因爲除了 "SIDES"、"AREA"、"CIRCUM"、"CEHTER" 這幾個字的定義外，其他的都是註解。

金城 註疏：

一、由 COMMENT： 到 COMMENT ；的中間行內容均是註解說明。

二、載入任何一個檔案的過程都會使得所有的冒號定義被加進字典中，這也就像是你在 F-PC 的直譯模式中，鍵入所有的冒號定義一般。除此以外所有的冒號定義現皆被存在磁片中，你可以在任何時候進入且編輯它們。

如果你要藉使用練習 1.1 中所提供的值來測試程式，你將會獲得下列的結果：

```
31  16  94  69  area   3339
31  16  94  69  circum  232
31  16  94  69  center   62 42

10  27  215  230  area   -23921
10  27  215  230  circum  816
10  27  215  230  center   112 128
```

面積的第二個值等於 —23921 ，這沒有意義；真正的情況是面積大於 32767 ，所以 16 位元的符號值便跑到負的區域去了，因爲 bit 15 的符號 bit 被設爲 1。我們可以使用 FORTH 的字 U (U-點)，而不是 ·(點)來印出面積真正的值 。 U· 這個字將在堆疊(stack)的頂端不帶符號的 16 位元整數值印出，這會產生下列的結果。

```
10 27 215 230 area U· 41615
```

2.3 除去你程式中的蟲 (除錯)

F-PC 有許多有用的字可以幫助你除去你程式中的錯誤。 SEE 這個字會讓你反編譯(DECOMPILE)分解一個字。例如在 floding HW1 (或第二課)這檔案後，鍵入

```
see area
see sides
see circum
see center
```

注意每一個冒號定義都出現了。這是藉著在字典中找出每一個字，然後找出每一個字定義的名字。

VIEW 這個字允許你找到已定義好的檔案，並顯示這檔案正確的定義內容。鍵入

```
VIEW sides
```

你可以使用 VIEW 這個字來找出任何一個 F-PC 文字的定義(原始程式)。

F-PC 文字中 DEBUG 是一個強有力的除錯工具，允許你一邊看堆疊(stack)上儲存了什麼東西，一邊一步一步地執行一個字。

在 FLOAD LESSON02 後，鍵入

```
debug area
```

AREA 這個字在下一次被執行時，會在定義好的每一個字上面暫停並顯示堆疊(stack)上的內容。按任意鍵(Q、C、N、U、X、F 除外)就可以繼續。

例如，如果你鍵入 10 27 215 230 area　AREA 的定義就會顯示在螢幕的上端，而當你按三次空間棒時，下列數字便會顯現在螢幕的下端。

```
10 27 215 230 AREA [4]    10    27    215   230
12648  0 :    SIDES     ?>  [2]    203   205
12648  2      *         ?>  [1]  41615
12648  4      UNNEST    ?>  OK
```

按空間棒，以單步執行每個名字的定義，同時堆疊(stack)的深度也會顯示在括弧 "[ ]" 中，而且堆疊(stack)最上面的四個項目值也會被顯示。要注意到在這個例子中，當 203 和 205 這二個數值被相乘以產生 41615 時，爲何顯示的值是 —23921 其理由在我們測試這程式時便會變得清楚，事實上，在經過上述 AREA 的定義的每一步驟之後，41615 這個值會被留在堆疊(stack)中。如果你接著鍵入 • (點)，—23921 便會顯示出來，這當然就是 16 位元整數值 41615 的符號值。

鍵入 UNBUG 以鍵入取消除錯，好使得將來執行 AREA 時不會被除掉、當一步一步執行時，按下 Q ，會停止除錯，並執行 UNBUG ；鍵入 C 會繼續執行直

到定義結束，或一直到按下 <ENTER> 鍵才會結束；鍵入 F ，會暫時的回到 FORTH 系統再鍵入 <ENTER> 會回到除錯程序中。

鍵入 X 會開/關，原始程式的部份。按下 N ，會進入將被執行的字；按下 U ，會回復原來的字。例如，鍵入 DEBUG AREA ，然後鍵入

```
10 27 215 230 AREA
```

然後按 N ，這會進入 SIDES 的定義，
一步一步地經過這定義並且看它如何轉回至 AREA 的定義。

練習 2.1

建一個名叫 HW2.SEQ 的檔，並且在這檔中寫下下列冒號定義：

```
: stacktest  ( a b ---?)
             DUP *
             SWAP DUP
             * + ;
```

加一堆疊畫面到這定義中每一行的右邊。FLOAD 此檔，當你鍵入

```
4  5 stacktest
```

使用 DEBUG 在字中的每一步。

a 和 b 會留下什麼數值在堆疊(stack)中。

第三課　FORTH 如何運作

3.1　變數(VARIABLE)

FORTH 文字" VARIABLE "是一個定義詞，用來宣告變數的名稱。
如果你鍵入

```
VARIABLE my.name
```

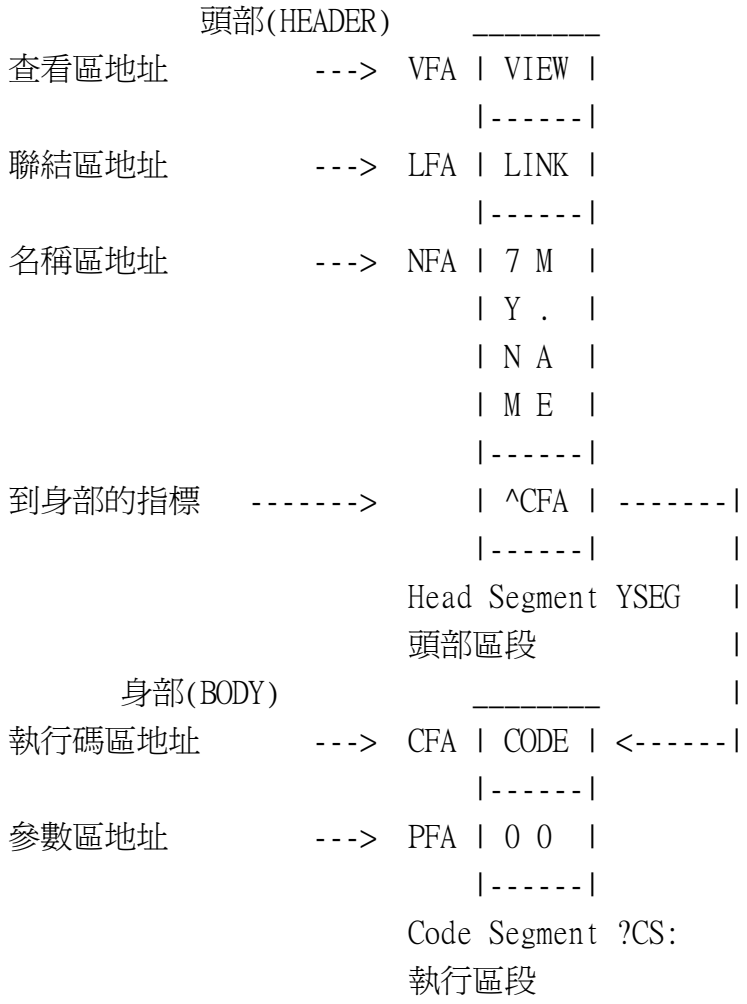
則 FORTH 便會在字典中產生一個新的詞叫 my.name

字典中所有的詞均擁有相同的一般格式，包括一個頭部(header) (由一查看區、名稱區、聯結區所組成) 和一個身部(body) (由執行碼區和一參數區所組成)。

在 F-PC 中，這頭部(header)和身部(body)實際上被儲存在不同的區段(segment) 中。

在 8086/8088/80286 機型中，1 Mbyte 地址空間被分割成若干個 64k 大小的段落，一個在電腦中實際的地址是由一個 16 位元的區段地址(segment address) "seg" 和一個 16 位元的位移地址(offset address) "off" 所組成。完整的地址被寫成區段:位移(seg:off)，區段位址可在任何以 16 位元組爲單位的標界上開始，稱做一小段(paragaph) 。因此，如果要使用記憶體中任一個位元組，你必須指明它的區段地址和位移地址。

對 my.name 這字而言，字典結構看起來會像這樣：



查看區包含一個字母計量數，功能相當於從一個檔案起點到這一行程序的位移地址；當你使用"VIEW"指令來查看任何一個 FORTH 字典中的詞時，查看區將指出該詞在檔案中的位置。

聯結區包含一個指標指到前一個已被定義的詞頭部的聯結區(LFA)。
名稱區包含由一個字串長度和 1 - 31 個字的字母所構成的名稱 。指到執行區的指標，內含一個在執行區段中該身部執行區的位移地址。在 F-PC 中使用這個指令 ?cs: 它將會告訴你執行區段的區段位址。

執行區段包含著該詞被執行的機械碼，這種方式叫做直接線串碼爲 (direct threaded code) 爲 F-PC 所使用。許多符式使用間接線串碼 (indirec threaded code) 該種 FORTH 的執行碼區包含一個指標指向真正被執行之機械碼。對一變數而言，執行區包含著三個位元組的指令 CALL >NEXT 。>NEXT 是 F-PC 內部執行機構，在本課稍後會有描述。CAL 指令會自動地

將下一個指令的地址放入(PUSH)堆疊(stack)中，但是在 FORTH 巧妙運用下它不是一個指令的位址，而是參數區位址。

參數區對各種不同詞類中包含各種不同內容。對一個變數字而言，參數區包含該變數的 16-bit 值。

當一個變數被宣告時，會將一個初值的零存入參數區內。
當你鍵入一個變數的名稱時 CALL >NEXT 的機械指令在執行區內會被執行，導致參數區地址被留在堆疊(stack) 。

如果你鍵入

my.name · (注意後面的" ·"爲輸出指令)

則 my.name 的參數區位置(PFA)就會被印出，試試看。

3.2 變數的抓取(FETCH)與儲存(STORE)

Forth 的詞:

! (n addr --) ("store")
儲存"n"的值在 addr 中

6 my.name !

將 6 這個值存在 my.name 變數的 PFA 中

@ (addr -- n) ("fetch")
抓取在 addr 中所儲存的值、並且將它放在堆疊 (stack) 中

my.name @ .

將儲存在 my.name 中的值抓取印出

堆疊變數

系統變數 SP0 包含一個空堆疊的堆疊指標值，因此

```
SP0 @
```

會傳回一個在堆疊(stack)中無任何物時的堆疊(stack)指標的位址。

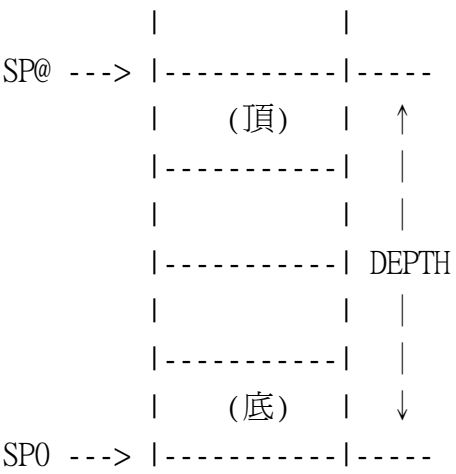
FORTH 的詞 SP@ 傳回最後推入(push)堆疊(stack)項目的位址。也就是說，它是堆疊(stack)指標的現值。

FORTH 的詞 DEPTH 傳回堆疊(stack)目前的深度(數目)。它被定義如下：

```
: DEPTH         ( -- n )
                SP@ SP0 @
                SWAP - 2/ ;
```

要注意既然堆疊(stack)中每一個元素是一個佔 2-byte ，則堆疊(stack)中位元的數目就必須被 2 整除，以計算出堆疊(stack)元素的項數。

金城 註疏：



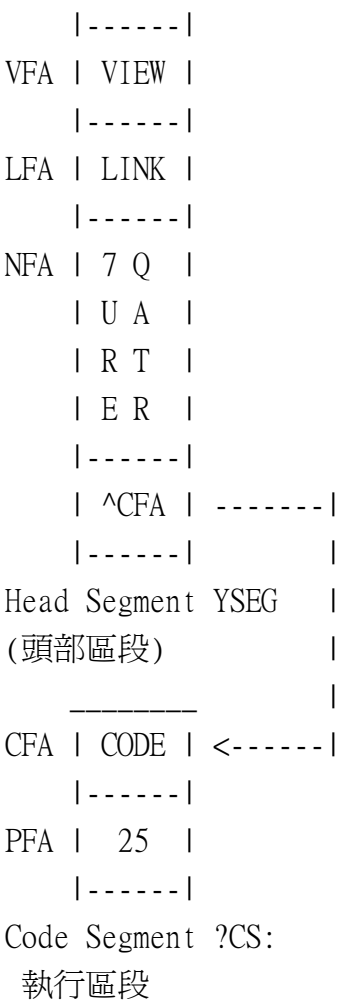
3.3 常數(CONSTANT)

FORTH 詞常數(CONSTANT)是一個定義詞，讓你來宣告所有的常數，舉例來說

如果你鍵入

```
25 CONSTANT quarter
```

quarter 的名稱便會如下例般編入字典中。



執行區包含了三個 byte 長度的機械指令 CALL DOCONSTANT 。DOCONSTANT 的機械指令會自堆疊(stack)中彈出(pop) PFA (PFA 是由 CALL 壓入堆疊(stack))，然後將儲存在 PFA 中的值推進(push)堆疊(stack) 中，因此，如果你鍵入

```
25 CONSTANT quarter
```

然後鍵入

```
quarter .
```

數值 25 便會被印出。

提示：

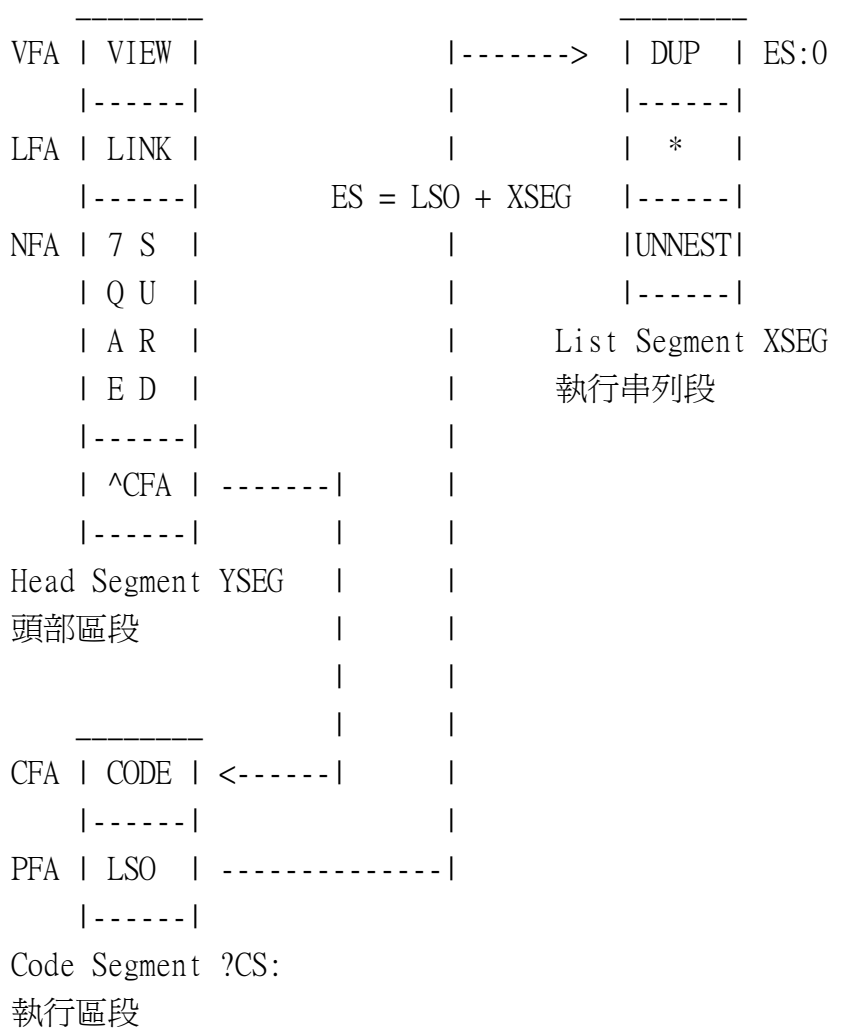
儲存在 CONSTANT 中的值是十六位元帶正負號的數目。

3.4 FORTH 的高階程序 -- ：(冒號)的使用

FORTH 詞 ： "冒號"也是一個定義詞，讓你可以建立新的 FORTH 詞。
當你鍵入

: squared DUP * ;

冒號指令"："便被執行，它將幫你把新的 FORTH 詞 squared 建入字典中，
看起來像這樣：



執行區包含三個 byte 長度的機械指令 JMP NEST ，在 NEST 的機械碼是內在執行器的一部分，其運作在本課稍後會有描述。

參數區包含一個執行串列段的位址(LSO)，其值加上執行串列段基底變數 XSEG 所形成，在解決此問題時區段位址是存在暫存器 ES 中，與執行區地址中的串列共同產生 SQUARED 新詞的地址，其位置儲存在記憶體中 ES：0 開始的位置上。

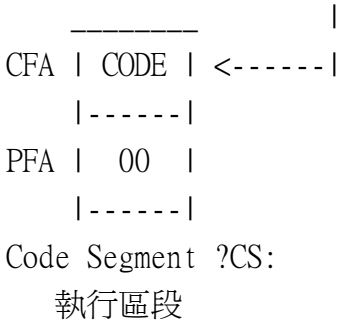
UNNEST 是另一個程序的地址，也是 FORTH 內在執行器的一部份。

3.5 陣列(ARRAYS)

假設你要建立一個包含五項每項十六位元的陣列，如果你鍵入

```
VARIABLE my.array
```

FORTH 將在字典中建立 my.array 其包含一個十六位元的值在它的參數區中。

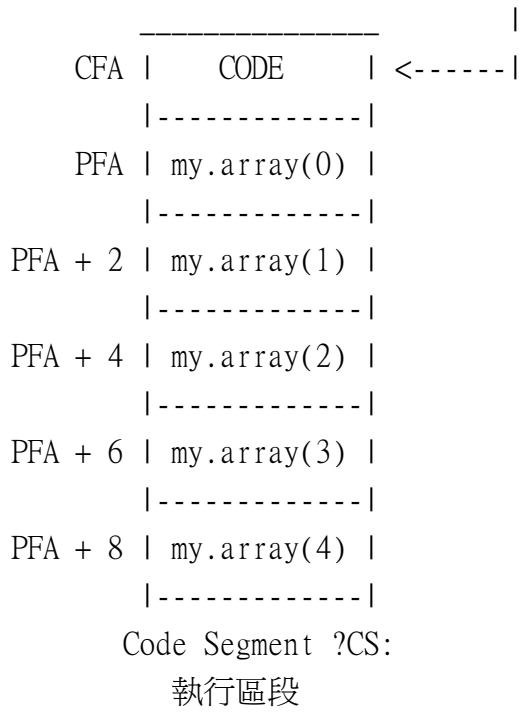


這裡我們不需要煩惱 my.array 頭部段的問題，要注意到參數區包含 2 byte 以儲存十六位元值。

FORTH 詞 ALLOT 當執行時會在執行區段的字典中加上 N 個位元組，N 是出現在堆疊(stack)中的值。因此，

```
8 ALLOT
```

會增加八個位元組或四個 16 位元的空間，於是在字典執行區段部份看起來便會像這樣。



要印出 my.array 陣列的第三個元素項的值，你可以鍵入

```
my.array 3 2* + @ .
```

3.6 回返堆疊(RETURN STACK)

當你鍵入一個數字時，它被放在參數堆疊(parameter stack)中，所有的算術運算與指令，例如 DUP 、ROT 、DROP 、SWAP 及 OVER 等都會對那些在參數堆疊(parameter stack)中的數字作運算。

FORTH 另有一個堆疊(stack)叫做回返堆疊(return stack)，這個回返堆疊(return stack) 是被 FORTH 內在執行器用來儲存一個高階程序中下一個指令的回返地址，它也被某些 FORTH 特定的指令，例如" DO " 所使用。

如果要使用回返堆疊(return stack)，你必需要小心謹慎的。你可以暫時將一個數字從參數堆疊(parameter stack)移動到回返堆疊(return stack) ，如果你確定在高階程序結束之前可以將它移回參數堆疊(parameter stack)的話。否則，真正的回返地址就不會出現在回返堆疊(return stack)的頂端，而內在執行器就無法正確找到回到下一個指令的地址。

下列是一些使用 FORTH 回返堆疊(return stack)的指令

>R (n --) ("to-R")
弾出參數堆疊(parameter stack) 頂端元素，且將之推入回返堆疊(return stack) 中。例如：

```
3 >R
```

將參數堆疊(parameter stack)中的 3 放至回返堆疊(return stack)之頂端，並使參數堆疊(parameter stack)變空。

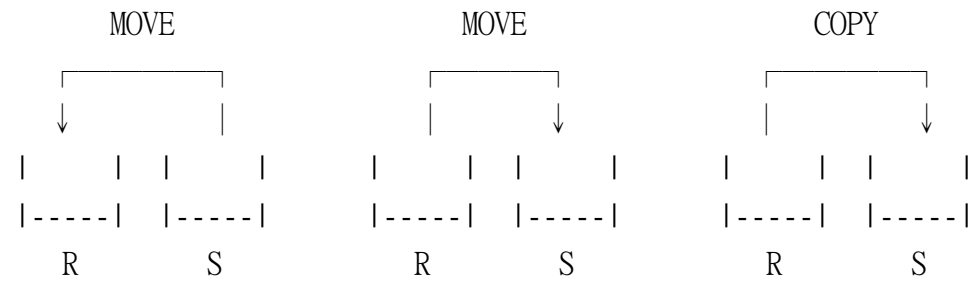
R> (-- n) ("from-R")
將回返堆疊(return stack)的頂端元素推出，並將其堆入參數堆疊(parameter stack)中。

R@ (-- n) ("R-fetch")
將回返堆疊(return stack)的頂端元素複製至參數堆疊(parameter stack)的頂端。

這裡有一個 ROT 指令的可能執行方式：

```
: ROT   ( n1 n2 n3 -- n2 n3 n1 )
        >R              \ n1 n2
        SWAP            \ n2 n1
        R>              \ n2 n1 n3
        SWAP ;          \ n2 n3 n1
```

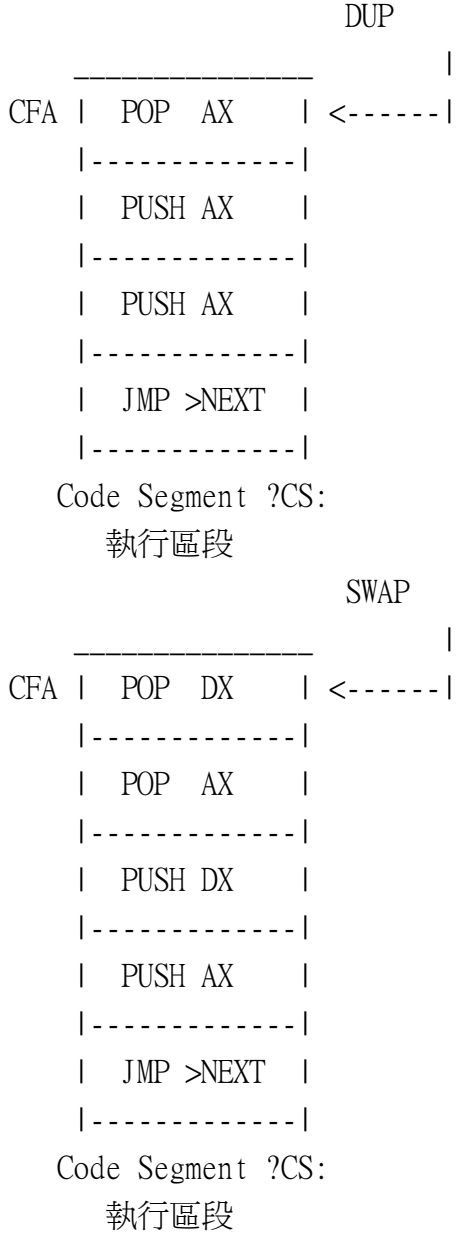
金城 註疏：
>R 、R> 、R@ 之運作如下：

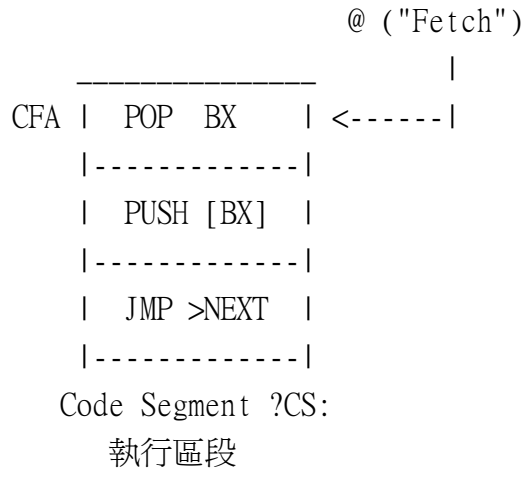


3.7 FORTH 的低階程序 -- CODE 的使用

CODE 詞群也是 FORTH 的可用詞群，但並不像前面所述及的 FORTH 高階程序，其內涵是由 8086 機器碼所構成的，而非其他的 FORTH 高階程序。當然，在任何一個 FORTH 的詞中最後總要用 8086 機器碼來執行。內在執行器是用機器碼寫成的；此外，許多 F-PC 的詞也是用機器碼寫成的，以求取最快的執行速度。在第七課中，我們會告訴你如何寫你自己的 FORTH 低階程序。

因爲 F-PC 使用直接線串碼，在低階程序內機器碼是被直接儲存在執行段落的 CFA 中。
這裡有一些例子可以讓我們了解 F-PC 低階程序是如何被定義的。在每一個例子中，每一個詞的頭部都是被儲存在頭部區段(Head segment)中，就像前面所描述有關變數(VARIABLE)、常數(CONSTANT)及高階程序的定義一樣。





3.8 FORTH 的詞典(DICTIONARY)

FORTH 的詞典是由所有已定義過的詞所組成的鏈串列(linked list) ，這些字可以是變數、常數、高階程序或者是低階程序，所有這些詞的名字都被儲存在頭部區段(head segment)中，而靠聯結區的地址所聯結。每個詞的執行區是被在頭部(head)的執行區指標所指出。執行區是包含了其真正可執行的機械碼，而且一定要放在 8086 的程式段(code segment)中。在高階程序中其組成的 CFA 串列被儲存在一分開的的可執行串列段中，且被程序段中 PFA 的指標所指出。

當你使用高階程序來建立一個新詞時，其編譯過程包含了將詞儲存在字典中。

F-PC 使用一個分段式架構來將你的字聯結進六十四條雜湊線串(hashing)中，以增加在詞典中找詞到的速度。

在字典的可執行段中下一個可用的地址是由 HERE 指令所提供，也就是說， HERE 是一個 FORTH 詞，其將下一個可用的地址傳回至堆疊(stack)中。變數 DP 是一個字典指標，包含了下一個可用的字典地址。HERE 這個詞可如下定義：

```
: HERE  ( -- n )
        DP @ ;
```

外部解釋器是指當你開啓一 FORTH 系統，且"OK"提示顯現時，所被執行的東西。當你鍵入一個字，且按下 <Enter> 外在解釋器便在字典中搜尋你所鍵入的字。如它找到那個字，它會藉執行區的區段地址來執行這個字；如果它沒

有找到這個字，它會執行一個叫做 NUMBER 的詞，試圖將你鍵入的字串轉換成一個有效的數字。如果它成功了，它便會將此數值推入至堆疊(stack)中，否則，它會題示 <--What? 訊息來告訴你，它不了解你的字。

我們會在 3.13 單元詳細地說明內在執行器的操作。

金城 註疏：

所謂外部解釋器就是輸入字串的執行機構。

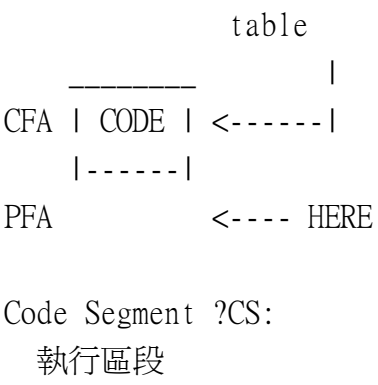
3.9 表格(TABLES)

表格就像是一個常數陣列。你可以造出一個陣列，然後使用指令來填入數值。

另外一個建立表格的方法是使用 FORTH 指令 CREATE ，除了參數不會留下空間外它的工作方式和建立變數一樣，。例如：如果你鍵入

```
CREATE TABLE
```

你會建立下列的字典結構。

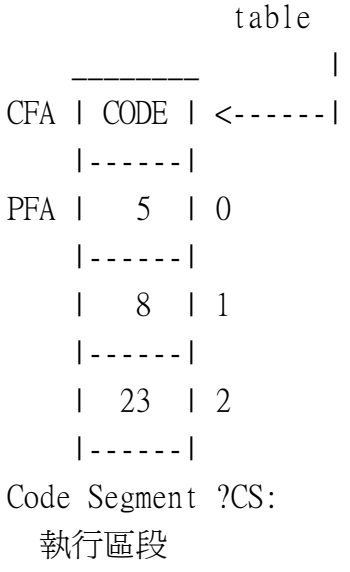


在變數的例子中，執行區包含三個位元，相當於指令 CALL >NEXT 在這裡 >NEXT 是 F-PC 的內部執行器，指令 CALL 當 TABLE 這個詞被呼叫時會將參數區地址留在堆疊(stack)中。

以這種觀點來看此時字典指標 DP 內包含了表格 PFA 之值，FORTH 指令"," 逗點(comma) 會將堆疊(stack)頂端的值編入 DP 所指的地方，也就是說在字典中下一個可以提供使用的位置，因此，如果你鍵入

```
CREATE TABLE  5 , 8 , 23 ,
```

下列的字典結構會被建立：



你現在可以定義一個新的詞叫 @table ，如下例：

```
: @table        ( ix -- n )
                2* table        \ 2*ix pfa
                + @ ;           \ @(pfa + 2*ix)
```

例如，2 @table 會使 23 傳回堆疊(stack)的頂端。

3.10 字母(CHARACTER)或位元資料

字母(ASCII)可以被儲存在一個位元組中，位元組資料可使用下列的 FORTH 指令儲存或自位元組取出：

C, (c --) ("C-comma")
將堆疊(stack)頂端低位元組(LSB)儲存在 HERE 中(在字典中的下一個可用的位置)。

C! (c addr --) ("C-store")
將低位元組(LSB)的值儲存在堆疊(stack)頂端的地址中。

C@ (addr -- c) ("C-fetch")

在地址中抓取一位元組，推入堆疊(stack)的低位元組(LSB)中

你要建立一個表格其所含的是 8 位元值，只要鍵入

```
CREATE table 5 C, 8 C, 23 C,
```

然後你可以定義一個詞叫做 C@table ，如下例：

```
: C@table       ( ix -- c )
            table + C@ ;
```

2 C@table 會將 23 傳回到堆疊(stack) 的頂端。

注意在 3.9 單元中所定義的 C@table 及 @table 之間的差異。

3-11 詞典的搜尋方式

下列指令可以被用來指定或轉換在字典中各種區段的地址：

```
'       ( -- cfa )      ("tick")
        指令 ' table 這樣的陳述會將 table 的 CFA 留在堆疊(stack)
        中。

>NAME   ( cfa -- nfa )  ("to-name")
        將堆疊(stack)上的執行區地址 CFA (在執行區段中)轉換到
        名稱區地址 NFA (在頭部段落中)

>LINK   ( cfa -- lfa )  ("to-link")
        將堆疊(stack)上的執行區地址 CFA (在執行區段中)轉換到
        聯結區地址 LFA (在頭部段落中)

>BODY   ( cfa -- pfa )  ("to-body")
        將堆疊(stack)上的執行區地址 CFA (在執行區段中)轉換到
        參數區地址 PFA (在執行區段中)
```

你可以使用下列的指令轉入執行區的地址:

```
BODY>   ( pfa -- cfa )  ("from-body")
NAME>   ( nfa -- cfa )  ("from-name")
LINK>   ( lfa -- cfa )  ("from-link")
```

你也可以從名稱區地址轉到聯結區地址且從聯結區地址轉到名稱區地址

```
N>LINK  ( nfa -- lfa )  (名稱 ---->聯結)
L>NAME  ( lfa -- nfa )  (聯結 ---->名稱)
```

FORTH 指令 HEX 將會用來將底數改變爲 16 進位、DECIMAL 這個字會將底數改回十進位。你可以藉任意底數值儲存在 BASE 變數中，而改變成任何基底，例如，HEX 的定義是

```
: HEX 16 BASE ! ;
```

注意當 HEX 的定義被載入(load)時其基底必須是十進位的。

FORTH 指令 U．將堆疊(stack)中的頂端值以 16 位元不帶正負號印出，其在 0 和 65535 之間，如果在十六進位模式中，印出在 0000 和 FFFF 之間。

舉例來說、要印出 OVER 這個字名稱區的十六進位地址，要鍵入

```
HEX ' OVER >NAME U. DECIMAL
```

FORTH 指令 LDUMP (SEG OFF #BYTE--)可以被用來得到一個十六進位傾印(DUMP)，從地址 SEG:OFF 開始傾印出 #byte 的長度的記憶體內容。例如鍵入：

```
YSEG @ ' OVER >NAME 20 LDUMP
```

看看是否你能看到 OVER 的名稱區。

3.12 名稱區的資料結構

當你使用高階程序定義一個像 TEST1 這樣的新詞時，就會產生下列的名稱區：

```
Precedence bit ----|  |---  Smudge bit          Hex value
 優先位元          |  |      遮隱位元          十六進位值

                 ---------------------------------
        NFA  | 1 | 0 | 0 | Name char count = 5 |   85
             |---------------------------------|
             | 0 | Character #1  T = 54H (hex) |   54
             |---------------------------------|
             | 0 | Character #2  E = 45H       |   45
             |---------------------------------|
             | 0 | Character #3  S = 53H       |   53
             |---------------------------------|
             | 0 | Character #4  T = 54H       |   54
             |---------------------------------|
             | 1 | Character #5  1 = 31H       |   B1
             |---------------------------------|
```

如果"優先位元"被設定，則詞便立刻會被執行。立即詞將在第九課中討論。

如果"遮隱位元"被設定，則在字典搜尋中，詞將不會被發現。當在編譯高階程序，此位元便會被設定。

鍵入下列這無意義的高階程序

```
: TEST1 ;
```

然後鍵人下列字母，以檢視名稱區

```
YSEG @ ' TEST1 >NAME 10 LDUMP
```

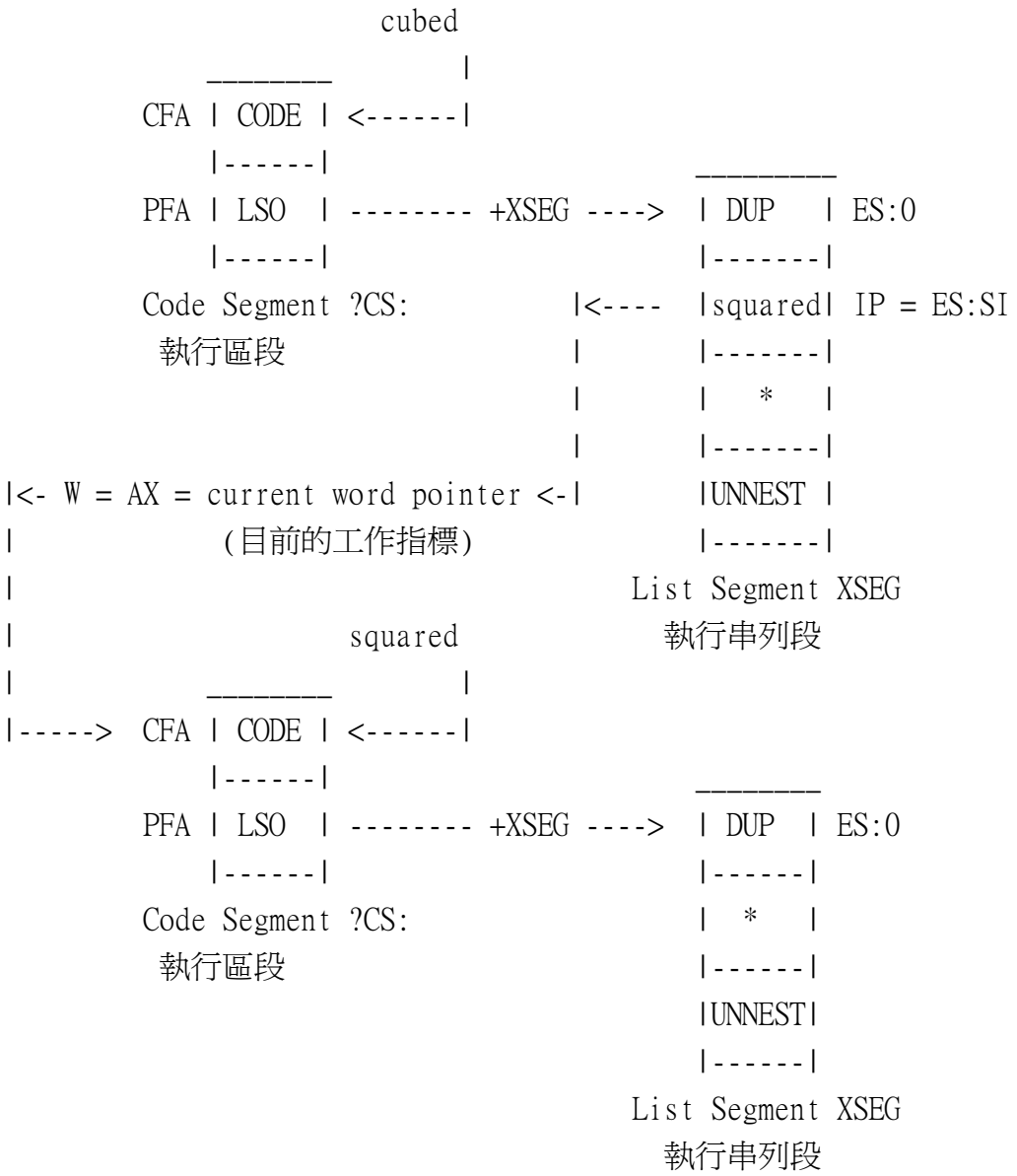
而上述圖表中的十六進位值應該被顯示出來。
要注意在名稱區裡的，第一個位元組和最後的位元組裡的最高位元被設爲 1 ，在名稱區中可存入名稱字母的最大長度，被儲存在變數 WIDTH 中。
例如：

```
10 WIDTH !
```

會導致最長 10 個字母被儲存在名稱區中。F-PC 將 WIDTH 的初設值設定爲 31 (31 爲名稱長度的最大可能值)。

3.13 F-PC 內部的執行原理

下列圖表描繪出 F-PC 內在執行器的操作




```
>NEXT   LODSW   ES:       \ 將 ES:SI 中的 CFA 放入 AX 暫存器並
        JMP     AX       \ 使 SI 暫存器內含加 1 。
                         \ 執行在 AX 暫存器中所指到的 CFA 位址的機
                         \ 械碼。
NEST    XCHG    BP,SP    \ 將 IP = ES:SI 推入回返堆
        PUSH    ES       \ 疊(return stack)。
        PUSH    SI
        XCHG    BP,SP
```

```
        MOV     DI,AX       \ 去執行 AX = CFA 詞。
        MOV     AX,3[DI]    \ 取得在 PFA 中所儲存的 LSO。
        ADD     AX,XSEG     \ 加上 XSEG。
        MOV     ES,AX       \ 將和放入 ES 暫存器。
        SUB     SI,SI       \ 建立一個新的 IP = ES:SI = ES:0。
        JMP     >NEXT       \ 去執行 >NEXT。

UNNEST  XCHG    BP,SP       \ 從回返堆疊(return stack)彈
        POP     SI          \ 出 IP = ES:SI。
        POP     ES
        XCHG    BP,SP       \ 返回原呼叫處。
        JMP     >NEXT       \ 去執行 >NEXT。
```

內在執行器包含三個常見的動作 >NEXT、NEST、及 UNNEST。一個執行器，或指令指標，IP 指向在執行串列段中的記憶體位址。這執行串列包含了下一個要被執行之詞的執行區地址。在 F-PC 中，這指令指標由二個暫存器 ES：SI 所構成。假設這是指向在 cubed 的定義中， squared 的 CFA ，如同在前面的圖表中所表示的。這個操作 >NEXT 將這一個 CFA 放入一個 16 位元位址暫存器 W 中 ，(這在 F-PC 中是 AX)，IP (SI)增加 2 ，會指向下一個高階程序中的詞 " * "， 然後去執行在 W 中的 CFA 所指到的機械碼。

在這個例子中，這是一個高階程序，它就會去跳入在 CFA 裡的機械碼也就是 NEST 。如同在 NEST 之上所提示的，會將 IP (ES 和 SI 二者)推入回返堆疊(return stack)，以致於當 UNNEST 被執行時，程式能在稍後找到它的路回到在 cubed 中的下一個字，然後 NEST 得到 squared 這個字的執行串列位移(LSO)，爲同時加上在 XSEG 中的底數之和儲存在 ES 暫存器中。然後它設定 SI 爲 0 ，以致於在 ES：0 中得到新的 IP 指向 squared 高階程序的第一個詞，然後跳入(jump) >NEXT，且重覆這過程這一次將會執行在 squared 中的第一個詞，叫 DUP。

既然 DUP 是一個低階程序，它真實的機械碼就位於它的 CFA ，當 >NEXT 被執行時，這個低階程序便會被執行。在 DUP 定義中最後一個指令是另一個跳入 >NEXT 。但是現在 IP 會被加 2 以指向 * 的 CFA 再一次執行，這個低階程序會再一次跳入 >NEXT 。

在一個高階程序中的最後一個字是 UNNEST ，UNNEST 的 CFA 被加入字典的執行串列，當一個高階程序中的" ; " (分號)被執行時。UNNEST 的執行區包含上面所顯示的機械碼，這碼將 IP(SI 和 ES)從回返堆疊(return stack)中彈

出，然後跳入 >NEXT 。既然當 squared 被執行時，是 IP 被 NEST 推入堆疊(stack)中，它會指向在 cubed 的定義中，跟在平方後面的那個字。" * " 就是下一個要被執行的詞。

這是 FORTH 的運作方式，高階程序就是在執行串列段中所儲存串列的 CFA 。

當要被執行的 CFA 是另一個高階程序時，指令指標(IP)會增加，且被推入回返堆疊(return stack)中，然後改爲指向即將被執行之新字的程序中之第一個 CFA。當要被執行的 CFA 是一個低階程序的 CFA ，則位於 CFA 之真實機械碼便會被執行。像這樣子的過程在每一個字中都會緊跟著跳入 >NEXT 中而繼續下去。

練習 3.1

定義高階程序詞

```
: squared  DUP * ;
```

和

```
: cubed  DUP squared * ;
```

使用 F-PC 指令 ' ("tick") 、 >LINK ("to-link")和 LDUMP (seg off #bytes--) 來回答下列問題：

什麼是執行區段地址 ?CS: ?
什麼是儲存在 YSEG 中的頭部段地址?
什麼是儲存在 XSEG 中的執行串列段地址?
什麼是 squared 的 CFA ?
什麼是 squared 的 LFA ?
什麼是 squared 的 NFA ?
什麼是 squared 的 PFA ?
什麼是 cubed 的 CFA ?
什麼是 cubed 的 LFA ?
什麼是 cubed 的 NFA ?
什麼是 cubed 的 PFA ?
畫一個 squared 的頭部圖，題示在所有位置的十六進位值。儲存在 ^CFA 位置的值是什麼? 畫一個圖來表示 squared 的 CFA 和 PFA 和執行串列段中的字典結構，並表示存在字典中的地址和值。
在字典結構中 cubed 的 LFA 是什麼 ? 那個詞的名稱是什麼?

什麼是 NEST 的地址 ?
什麼是 DUP 的 CFA ?
什麼是 * 的 CFA ?
什麼是 UNNEST 的地址?

第四課　FORTH 判斷與迴路

4.1　分岐指令及迴路。

所有的電腦語言都必須有一些方法來產生一些條件式的跳躍(IF---THEN)以及執行的迴路。 FORTH 使用下列結構化的設計：

```
IF ---- ELSE ---- THEN

DO ----- LOOP

BEGIN ----- UNTIL

BEGIN ---- WHILE ---- REPEAT

BEGIN ----- AGAIN
```

這些指令的工作和它作在其他語言中多少有些不同。" IF "、" UNTIL "和 " WHILE "這幾個字是 FORTH 的字，當它們被執行時，在堆疊(stack)頂端應會期待一個真/僞(對/錯)的旗號出現，一個錯(FALSE)的旗號以 0 表示，一個真(TRUE)的旗號以 -1 表示。

F-PC 定義這二個常數。

```
-1 CONSTANT TRUE
 0 CONSTANT FALSE
```

旗號可用任何方法來產生，但一般的方法是使用某種條件式的測試，來使堆疊(stack)上留下旗號。

我們會先來看看 FORTH 的條件字組，然後再就每一種分岐和上列的迴路敘述舉一些例子來說明。

金城 註疏：

大多數所謂結構化的高階語言如 C 、PASCAL 等均有 GOTO 指令。
FORTH 則以完全的結構指令來達成單進單出的絕對要求。

4.2 條件字組--真\僞的旗號。

下列 FORTH 條件字組會產生一個真/假旗號。

< (n1 n2 -- f) (" 小於 ")。
如果 n1 小於 n2 則旗號 f 爲真。

> (n1 n2 -- f) (" 大於 ")。
如果 n1 大於 n2 則旗號 f 爲真。

= (n1 n2 -- f) (" 等於 ")。
如果 n1 等於 n2 則旗號 f 爲真。

<> (n1 n2 -- f) (" 不等於 ")。
如果 n1 不等於 n2 則旗號 f 爲真。

<= (n1 n2 -- f) (" 小於或等於 ")。
如果 n1 小於或等於 n2 則旗號 f 爲真。

>= (n1 n2 -- f) (" 大於或等於 ")。
如果 n1 大於或等於 n2 則旗號 f 爲真。

金城 註疏：

因爲 0 在 FORTH 中用的非常多，而且 0 的測試在每一個 CPU 中均有硬體的指令，因此在 FORTH 中，另外有一群 0 的條件測試字組，以提高執行的效率和減少程式的長度(複雜度)。

0< (n -- f) (" 小於 0 ")。
如果 n 小於 0(負數)則旗號 f 爲真。

0> (n -- f) (" 大於 0 ")。
如果 n 大於 0(正數)則旗號 f 爲真。

0= (n -- f) (" 等於 0 ")。

如果 n 等於 0 則旗號 f 爲真。

0<> (n -- f)　　(" 不等於 0 ")。
如果 n 不等於 0 則旗號 f 爲真。

0<= (n -- f)　　(" 小於或等於 0 ")。
如果 n 小於或等於 0 則旗號 f 爲真。

0>= (n -- f)　　(" 大於 8 得於 0 ")。
如果 n 大於或等於 0 則旗號 f 爲真。

下列條件字組比較堆疊(stack)上二個不帶正負號的數。

U< (u1 u2 -- f)　(" 小於 ")。
如果 u1 小於 u2 則旗號 f 爲真。

U> (u1 u2 -- f)　(" 大於 ")。
如果 u1 大於 u2 則旗號 f 爲真。

U<= (u1 u2 -- f)　(" 小於或等於 ")。
如果 u1 小於或等於 u2 則旗號 f 爲真。

U>= (u1 u2 -- f)　(" 大於或等於 ")。
如果 u1 大於或等於 u2 則旗號 f 爲真。

4.3 FORTH 邏輯運算指令

一些 FORTH 有 NOT 這個字，它會將堆疊(stack)中旗號的真值"非"倒過來，在 F-PC 中， NOT 這字使堆疊(stack)頂端的數變成一階補數，只要 TRUE 是 -1 那麼 NOT TRUE 就會是 0 (錯 FALSE)，你必須小心，因爲任何非零的值有時候會被當做是一個真的旗號，任何數的一階補數，除十六進位的 FFFF 以外，都不會變成 0 (FALSE)。你總是可以藉使用 0= 這個比較用字來完成任何旗號的設定。

金城 註疏：
例如鍵入 7 0= 會留下 0 在堆疊(stack)上。

除了 NOT 這邏輯運算指令之外， FORTH 也還有下列二進位的邏輯運算指令

AND (n1 n2 -- and)
留下 N1 AND N2 在堆疊(stack)頂端。
這是一個位元型態的 AND 運算例 如，如果你鍵入

255 15 AND (取最低四位元的值)

數值 15 將會被留在堆疊(stack)頂端。

OR (n1 n2 -- or)
留下 N1 OR N2 在堆疊(stack)頂端。
這是一個位元型態的 OR 運算例 如，如果你鍵入

9 3 OR

數值 11 會被留在堆疊(stack)頂端。

XOR (n1 n2 -- xor)
留下 N1、XOR、N2 在堆疊(stack)頂端。
這是一個位元型態的 XOR ，舉例，如果你鍵入

240 255 XOR (十六進位的 FO XOR FF =OF)

則數值 15 會被留在堆疊(stack)頂端。

4.4 IF 敘述

FORTH 語言中的 IF 敘述與其他語言稍有不同。一個典型的 IF --- THEN --- ELSE (如果---則---否則) 陳述，是你所熟悉的

IF <cond> THEN <true statements> ELSE <false statements>

，可是這樣的運作在 FORTH 中， IF 敘述就像這樣。

<條件測試> IF <爲真的敘述> ELSE <爲僞的敘述> THEN

金城 註疏：

FORTH 的 THEN 是一個結束 IF 敘述的位置。

注意當 IF 這字被執行時，一個真/假旗號，必須出現在堆疊(stack)頂端。如果一個真的旗號出現在堆疊(stack)頂端，那麼<爲真的>或<爲僞的>被執行之後， THEN 這個字以後的字便會被執行。而 ELSE 子句則是可有可無的。

IF 這字必須在冒號定義中使用，舉例來說，定義下列字：

```
: iftest         ( f -- )
                  IF
                     CR ." true statements "
                  THEN
                  CR ." next statements "

: if.else.test   ( f -- )
                  IF
                     CR ." true statements "
                  ELSE
                     CR ." false statements "
                  THEN
                     CR ." next statements "
```

請鍵入

```
TRUE  iftest
FALSE  iftest
TRUE  if.else.test
FALSE  if.else.test
```

4.5 " DO "的迴路

FORTH 的 " DO "迴路必須在冒號定義中被定義。

要看它如何運作，則定義下列字：

```
: dotest   ( limit ix -- )
             DO
                I .
             LOOP ;
```

然後如果你鍵入

```
          5 0 dotest
```

則數值 0 1 2 3 4 會被印在螢幕中，試試看。

DO 的迴路按下列提示運作。DO 這個字會從參數堆疊(parameter stack)的頂端取二個值，並將它們搬至回返堆疊(return stack)，在這時候，這二個值不再留在參數堆疊(parameter stack)上。
LOOP 這個字會將迴圈啓始數值加 1 ，並且比較其與終止值之差如果起始值小於終止值，那麼在 DO 這字後面就有了一分岐。如果這增加的起始值等於終止值，那麼這二個值會自回返堆疊(return stack)中被移除而在 LOOP 之後的字會被執行。

在第九課中，我們將仔細來討論 DO 迴路實際上如何運作執行。

FORTH 的字 I 將起始值自回返堆疊(return stack)上複製至參數堆疊(parameter stack)的頂端，因此 ，上述例子的執行可展示如下：

```
          5          \ 5
          0          \ 5  0
          DO
             I       \ ix    ( ix = 0,1,2,3,4)
             .
          LOOP
```

注意終止值必須大於你要的最大的起始指數值。例如：

```
          11  1 DO
                 I .
              LOOP
```

將會印出數值

1 2 3 4 5 6 7 8 9 10

+LOOP 這個詞

FORTH 中 DO 迴路的指數可以藉由使用 +LOOP 代替 LOOP，來使其增加 1 以外的值。要看它如何運作，就定義下列字：

```
: looptest     ( limit ix -- )
               DO
                  I .
               2 +LOOP ;
```

然後如果你鍵入

```
5 0 looptest
```

值 0 2 4 會被印在螢光幕上，試試看。

+LOOP 這個詞將參數堆疊(parameter stack)上的值加入在回返堆疊(return stack)上的計數值裡，然後就和 LOOP 一樣，只要增加的計數值小於終止值(如果增加值是正數的話)，它就會回到在 DO 之後的詞。如果增加的值變得小於終止值時，迴路便會跳離開。例如：如果你定義下列的詞

```
: negloopteste     ( limit ix -- )
                   DO
                      I .
                   -1 +loop ;
```

然後輸入

```
0 10 negloopteste
```

這些值 10 9 8 7 6 5 4 3 2 1 0 將會被印出在螢光幕上。

巢串迴路的 J 字

FORTH 的 DO 迴路可以巢串(重疊)，當你這樣做時，二對計數/極限值被移至

回返堆疊(return stack)。

I 這個字自回返堆疊(return stack)上將內在迴路的計數值複製至參數堆疊(parameter stack)上，而 J 這字會從回返堆疊(return stack)上將外在迴路的指數值複製至參數堆疊(parameter stack)上。舉一巢串迴路的例子。定義下列文字：

```
: 1.to.9          ( -- )
                  8 1 DO
                        CR
                        3 0 DO
                              J I + .
                        LOOP
                  3 +LOOP ;
```

如果你鍵入 1.to.9 來執行此字，下列數字會被印出：

```
1 2 3
4 5 6
7 8 9
```

你看出來爲什麼嗎? 試試看。

巢串迴路在 FORTH 中比在其他語種更不常被使用，通常較好的做法是去定義較小的字，只包含一個 DO 的迴路，然後再在其他迴路中叫出這字。

LEAVE (離開)這個詞

FORTH 的字 LEAVE (離開)可以被用來提早跳出一個 DO 迴路，這通常在 DO 迴路中的一個 IF 敘述裡被使用。LEAVE (離開) 這個字引起立即跳出 DO 迴路，(在 LOOP 之後的文字的地址，以第三個字被儲存在回返堆疊(return stack) 中)。一個相關的字 ?LEAVE(flag---)會跳出一個 DO 迴路，如果參數堆疊(parameter stack)頂端的旗號爲真的話。這可以避免需要一個 IF 陳述。

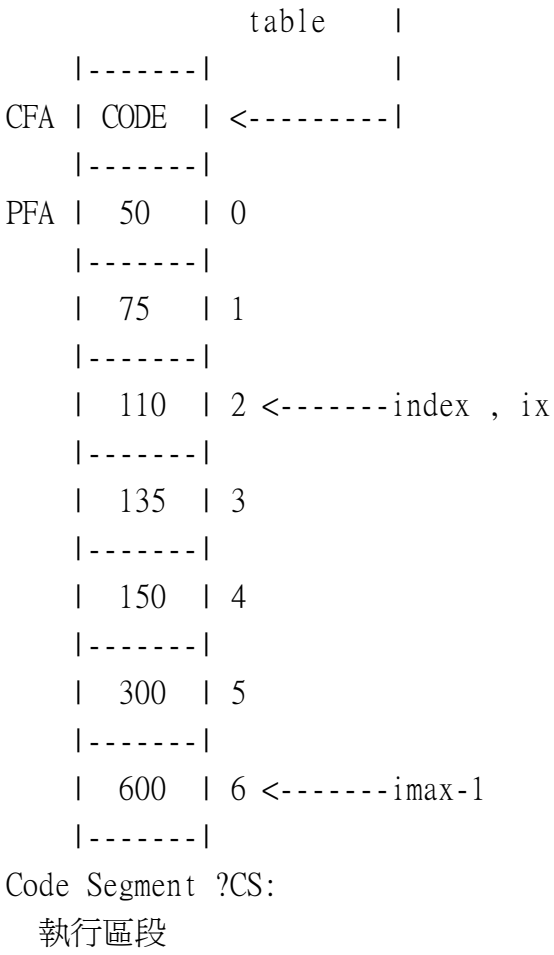
金城 註疏：

將回返堆疊(return stack)上的計數值，終止值推出清除就能回到下一個字。

舉例來說，假設你要定義一個字叫 FIND.N 的字，他會在一個表格中找尋一個特定的值，並且如果這個值被找到的話，就在 " 真 " 旗號之下返回這值的索引指標(也就是說，它在表中的位置)，否則一個 " 僞 " 旗號會被留在堆疊(stack) 頂端，FORTH 陳述，

```
CREATE table  50 , 75 , 110 , 135 , 150 , 300 , 600 ,
```

會在程式段產生下列的表來。



表中值的數目是 imax (在本例中爲 7)，要被找尋的值在 N 中，當 find.n 被執行時，這二個值會在堆疊(stack)中，下列就是 find.n 的一個定義。

```
: find.n       ( imax n -- ff | index tf )
               0  SWAP  ROT                \ 0 n  imax
               0  DO                       \ 0 n
                   DUP  I table            \ 0 n  n ix pfa
```

```
        SWAP   2* +              \ 0 n  n pfa+2*ix
        @ =                      \ 0 n  f
        IF                       \ 0 n
            DROP I TRUE          \ 0 ix tf
            ROT LEAVE            \ ix tf 0
        THEN
    LOOP                         \ 0 n
    DROP ;                       \ 0 | ix tf
```

研究這個定義直到你知道它如何工作，大致說來，當使用一個 DO 迴路時，在執 DO 之後的堆疊圖應該和在執 LOOP 之後的堆疊圖一樣，你會經常需要在一個 DO 迴路中 DUP(複製)一些堆疊(stack)上的值然後在你離開迴路之後，DROP(拋除一些堆疊(stack)上多餘的數值)。

注意：在這一個例題中 ROT 在 LEAVE 的前面，它是被用來安置堆疊(stack)(將某一個值放在堆疊(stack)上適當的位置)。以便讓最後的一個 DROP 指令能將旗號留在堆疊(stack)上。將真旗號留在堆疊(stack)頂端中。

4.6 UNTIL 迴路

FORTH 的 UNTIL 迴路必須在冒號定義中被使用。UNTIL 迴路的形式是，

BEGIN <符式陳述> <旗號> UNTIL

如果 <旗號> 爲錯，則程式跳躍至 BEGIN 後面的字。如果 <旗號> 爲真，程式則繼續執行在 UNTIL 之後的字。

下列二個 F-PC 文字能偵測並且讀入鍵盤所鍵入之字。

KEY? (-- flag)
如果你按了一個鍵，就會產生一個真旗號。

KEY (-- char)
等待其一個鍵被按下，並且留下此鍵的 ASCII 代碼在堆疊(stack)上。

EMIT (char --)

將 ASCII 代碼在堆疊(stack)頂端的字母印出在螢幕上。

定義下列字

```
: dowrite    ( -- )
             BEGIN
                KEY       \ char
                DUP EMIT  \ 印在螢幕中
                13 =      \ 如果等於 CR
              UNTIL ;     \ 離開
```

執行這個字會將所有你鍵入的字母在螢幕上印出來，直到你按 <ENTER> 鍵爲止(ASCII 代碼 =13) 。

注意 UNTIL (直到)這個字會將旗號從堆疊(stack)移除。

4.7 WHILE 迴路

在 FORTH 中 " WHILE "迴路必須在冒號定義中被使用。" WHILE "迴路的形式是

BEGIN<字詞>、<旗號>WHILE<字詞>REPEAT

如果 <旗號> 是真，在" WHILE "和" REPEAT "之間的字群會被執行，然後會跳躍至緊接在" BEGIN "之後的字群。如果 <旗號> 是僞，程式便會跳躍至緊接在" REPEAT "之後的字群。

舉個例子，使用下列的 ALGORITHM (演譯法)來計算 N 的階乘。

```
X = 1
i = 2
DO WHILE i <= n
    x = x * i
    i = i + 1
ENDDO
```

factorial = x

下列 FORTH 的定義可計算這個階乘。

```
: factorial         ( n -- n! )
                    1 2 ROT             \ x i n
                    BEGIN               \ x i n
                       2DUP <=          \ x i n f
                    WHILE               \ x i n
                       -ROT TUCK        \ n i x i
                       * SWAP           \ n x i
                       1+ ROT           \ x i n
                    REPEAT              \ x i n
                    2DROP ;             \ x
```

要注意堆疊(stack)的安排必須和在 BEGIN 和 REPEAT 這二個字上一樣，以使 WHILE 迴路適當地運作。另外也要注意，雖然上列之字演譯法，使用三個變數 X , I 和 N ，FORTH 運算則完全不使用變數。這是 FORTH 之特色，你會發現在 FORTH 中使用的變數遠較其他語言少得多。試試階乘定義 ，鍵入

```
3 factorial .
4 factorial .
0 factorial .
```

練習 4.1

費邊系數(fibonacci)是一種數字的級數，其中每一個數字(從第三個開始)是其前二個數字的總和。所以這順序的開頭看起來像這樣。

1 1 2 3 5 8 13 21 34

定義一個 FORTH 的字叫

fib (n---)

這會印出少於 n 的所有值之 fibonacci 數列。試試你的字，鍵入

1000 fib

練習 4.2

建立一個表叫 weights(重量)，包含下列的值：

75 135 175 115 220 235 180 167

定義一個 FORTH 的字叫 HEAVIEST(最重的)。這會將表中的最大值放在堆疊(stack)頂端。如果你鍵入

weights heaviest

值 235 應該被印出在螢幕中 。

第五課　數　字

5.1　雙倍精密度數字

一個雙倍精密度數字是一個 32-bit 的整數，以兩個 16-bit 的字儲存在堆疊(stack)上，而在堆疊(stack)頂端有 HI-word 的字樣，如下例：

```
|---------|
| HI-word | <---top of stack
|---------|
| LO-word |
|---------|
```

一個雙倍精密度數字的堆疊圖會被指示爲

```
(d--)
```

一個雙倍精密度數字自鍵盤進入，在數字中出現任意位置的小數點，則該數字被視爲雙倍精密(32-bit)數字放在堆疊(stack)上例如，如果你鍵入

```
1234.56
```

這個相當於 16 進位值 1E240 的整數值將會被按下列情況儲存在堆疊(stack)的最上端二個字中：

```
|---------|
| 0 0 0 1 | <---top of stack
|---------|
| E 2 4 0 |
|---------|
```

DPL 這變數將存放小數點後面的位數，1234.56 的例子中 DPL 爲 2。

FORTH 的字 D. 可用來印出一個雙倍精密度數字的值在螢幕中，因此，例如在輸進 1234.56 之後，你可以鍵入 D. 則 123456 值就會被印出在螢幕中

金城 註疏：

在 FORTH 中，如機器不提供浮點運算器則使用雙倍精密度的 32 位元用字，以解決精密度的問題。因爲用軟體來模擬浮點(實數)又慢又複雜、又容易出錯。不合 FORTH 的精簡哲學。

但若你是用 Intel 的 80486 CPU 則 80387 算術運算器是內建的，不用白不用，則就該發揮 FORTH 的精神，將硬體的全部功能壓榨出來。這就是 FORTH 所謂人機一體的觀念。將電腦看成人類記憶與計算能力的延伸，一如工業革命以機器延伸了人類肢體的勞動能力，而 FORTH 的人機合一，也意謂著一場人類大腦的創造力革命!

你知道嗎? 80387 的結構，就是一個硬體的堆疊運算器，與 80386 配合成一個完整的雙堆疊 (ESP 是回返堆疊，而 80387 的八層暫存器堆疊 ST(0)--ST(7) 是運算堆疊) FORTH 系統。其硬體指令和 FORTH 的堆疊運算一一對應! 如 FLD ST(0) 就是 DUP 、FSTP ST(0) 就是 DROP 、FXCH ST(1) 就是 SWAP 、 FLD ST(1) 就是 OVER 、 FXCH ST(1)、FXCH ST(2) 就是 ROT 所以，由此可知連 Intel 這樣的大公司，也是 FORTH 雙堆疊結構 CPU 的支持者，80486 就是最好的鐵證。

下列是一些雙倍精密度數字的指令

D+ (d d -- dsum)
將二個雙倍精密度數字加起來，而得到一個雙倍精密度數總合值。

DNEGATE (d -- d)
改變一個雙倍精密度數字的正負。

S>D (n -- d)
將一個單倍精密度數字轉換成一個雙倍精密度數字。

DABS (d -- d)
取一個雙倍精密度數字的絕對值。

D2* (d -- d*2)
以 2 乘，也就是 32 位元的左移一個位元。

D2/ (d -- d/2)

以 2 除，也就是 32 位元的右移一個位元。

DMIN (d1 d2 -- d3)
d3 是 d1 和 d2 中較小的那一個值。

DMAX (d1 d2 -- d3)
d3 是 d1 和 d2 中較大的那一個值。

D- (d1 d2 -- d1-d2)
將二個雙倍精密度數字相減而得到一個雙倍精密度數的差。

注意：
D- 的定義是

```
: D-       DNEGATE D+ ;
```

?DNEGATE (d1 n -- d2)
如果 n < 0 則 DNEGATE d1

注意：
?DNEGATE 的定義是

```
: ?DNEGATE      ( d1 n -- d2 )
                0<
                IF
                   DNEGATE
                THEN ;
```

5.2 雙倍精密度的比較指令

下列是雙倍精密度數字比較指令的定義：

```
: D0=      ( d -- f )          \ 如果 d = 0 旗號是真
           OR 0= ;

: D=       ( d1 d2 -- f )      \ 如果 d1 = d2 旗號是真
           D- D0= ;
```

```
: DU<     ( ud1 ud2 -- f )          \ 如果 ud1 < ud2 旗號是真
          ROT SWAP                  \ ud1L ud2L ud1H ud2H
          2DUP U<                   \ ud1L ud2L ud1H ud2H f
          IF                        \ ud1L ud2L ud1H ud2H
            2DROP 2DROP TRUE        \ f
          ELSE
             <>                     \ ud1L ud2L f
             IF                     \ ud1L ud2L
                  2DROP FALSE       \ f
             ELSE
             U<                     \ f
          THEN
        THEN ;

: D<      ( d1 d2 -- f )       \如果 d1 < d2 旗號是真
          2 PICK                    \ d1L d1H d2L d2H d1H
          OVER =                    \ d1L d1H d2L d2H f
          IF                        \ d1L d1H d2L d2H
             DU<                    \ f
          ELSE
             NIP ROT DROP           \ D1H D2H
             <                      \ f
          THEN ;

: D>      ( d1 d2 -- f )       \如果 d1 >= d2 旗號是真
          2SWAP
          D< ;
```

5.3 乘和除

基本乘除運算，也是其他乘除字群的基礎，例如

```
UM*       ( un1 un2 -- ud )
```

UM* 將未帶正負號的 16 位元整數 un1 乘以未帶正負號的 16-bit 整數 un2，而得到未帶正負號 32-bit 的積 ud 。這個字使用 8088/8086MUL 機器指令。

UM/MOD (ud un -- urem uquot)

UM/MOD 將未帶正負號的 32-bit 整數 ud 除以 16-bit 未帶正負號的整數 un 而得到未帶正負號的商 uquot。以及未帶正負號的餘數 urem。這個字使用 8088/8086DIV 機器指令，如果被除數的 high word 大於或等於 Un 那麼商數不會符合 16-bit ，在這些情況下，8088/8086 DIV 機器指令將產生一個 "被 0 除" 的 Trap 錯誤(硬體錯誤) FORTH 文字 UM/MOD 會檢查這種例子並且如果商數太大而無法符合 16-bit 時，會將商數和餘數放入 16 進位 FFFF 值。

下列 F-PC 文字會將二個帶正負號的 16-bit 整數相乘而留下一個 32-bit 帶正負號的積。

```
: *D        ( n1 n2 -- d )
            2DUP XOR >R             \ 先存一份積的正負號
            ABS SWAP ABS
            UM*                     \ 不帶正負號的乘積
            R> ?DNEGATE ;           \ 調整正負號
```

下列 F-PC 文字會將一個未帶正負號的 32-bit 整數除以一個 16-bit 未帶正負號的整數，並且留下一個未帶正負號 16-bit 的餘數和一個 32-bit 未帶正負號的商，這個字將不會有 UM/MOD 商數過大的問題。

```
: MU/MOD  ( ud un -- urem udquot )
          >R 0 R@                   \ udL udH 0 un
          UM/MOD                    \ udL remH quotH
          R>                        \ udL remH quotH un
          SWAP >R                   \ udL remH un
          UM/MOD                    \ remL quotL
          R> ;                      \ remL quotL quotH
```

5.4 底數的除法

二個帶正負號的除法用字

/MOD (n1 n2 -- rem quot)

M/MOD (d n1 -- rem quot)

執行底數除法，鍵入下列四個例子，並且試著去預測螢幕上會顯現什麼

```
 26 7  /MOD . .
-26 7  /MOD . .
 26 -7 /MOD . .
-26 -7 /MOD . .
```

結果可摘要如下：

被除數	除數	商數	餘數
26	7	3	5
-26	7	-4	2
26	-7	-4	-2
-26	-7	3	-5

你是否期待這些結果? 第二和第三個結果，可能你會覺得奇怪，但他們是正確的，因爲我們所需要的是除數乘以商數加上餘數能等於被除數。注意看

```
 3 *  7 + 5 =  26
-4 *  7 + 2 = -26
-4 * -7 - 2 =  26
 3 * -7 - 5 = -26
```

這叫做底數的除法，其特色是餘數的符號和除數的符號相同。

這不是做除法的唯一方法，事實上， 8088/8086 的 IDIV 機器指令沒有做底數除法，在這個例子中，餘數的符號是和被除數的符號相同，而且商的大小也總是一樣的。

金城 註疏：

在 FORTH 中因考慮在真實世界中的應用通常除法的想法是 X 除以 Y 得 A 餘 B 而其意義是在 X 中有 A 個 Y 所以 Y 乘以 A 加上 B 應等於 X 。

要瞭解這點，定義下列使用 IDIV 機器指令的 Code 定義內容：
(Code 爲組合語言定義)

```
PREFIX
CODE ?M/MOD
        POP   BX
        POP   DX
        POP   AX
        IDIV  BX
        2PUSH
END-CODE
```

現在鍵入

```
 26. 7 ?M/MOD . .
-26. 7 ?M/MOD . .
 26. -7 ?M/MOD . .
-26. -7 ?M/MOD . .
```

這些新結果可摘要如下：

被除數	除數	商數	餘數
26	7	3	5
-26	7	-3	-5
26	-7	-3	5
-26	-7	3	-5

注意在這個例子中，除數乘以商加上餘數仍然等於被除數，雖然你可能喜歡這種除法的樣子甚於有底的除法，底數除法有辦法來區別商爲 0 的曖昧狀況的優點，要瞭解這點，試試看以下情況：

有底的除法

```
 3   4   /MOD . .   0  3
-3   4   /MOD . .  -1  1
 3  -4   /MOD . .  -1 -1
-3  -4   /MOD . .   0 -3
```

無底的除法

```
3.    4   ?M/MOD . .  0   3
```

```
-3.    4   ?M/MOD . .  0  -3
 3.   -4   ?M/MOD . .  0   3
-3.   -4   ?M/MOD . .  0  -3
```

注意無底的除法無法分辨帶正負號各種除法之間的差異

M/MOD 如何被定義：

```
: M/MOD   ( d n -- rem quot )
          ?DUP                    \ 傳回 d 如果 n = 0
          IF                      \ dL dH n
             DUP >R               \ 暫存 n 在回返堆疊
             2DUP XOR >R          \ 暫存正負號
             >R                   \ dL dH
             DABS R@ ABS          \ |dL dH| |n|
             UM/MOD               \ urem uquot
             SWAP R>              \ uquot urem n
             ?NEGATE              \ uquot rem (sign=divisor)
             SWAP R>              \ rem uquot xor
             0<
             IF                   \ rem uquot
                NEGATE            \ rem quot
                OVER              \ rem quot rem
                IF                \ rem quot
                   1-             \ floor quot toward -infinity
                   R@             \ rem quot n
                   ROT-           \ quot floor.rem = n - rem
                   SWAP           \ rem quot
                THEN
             THEN
             R> DROP
          THEN ;
```

/MOD 可因此被定義如下：

```
: /MOD    ( n1 n2 -- rem quot )
          >R S>D R>
          M/MOD ;
```

F-PC 實際上將 /MOD 定義成一個 Code 字，使用 IDIV 緊跟著餘數和商數的底數一致化。

5.5 16 位元運算指令

下列是運算 16-bit 數字且留下 16-bit 結果的算術運算指令可能被定義的方式。事實上，F-PC 定義許多這類有速度考慮的相同功能的 Code 定義字。

```
: *         ( n1 n2 -- n )          \ 帶正負號的乘法
            UM* DROP ;
```

雖然這看起來奇怪，但如果你只拋掉一個帶正負號 32-bit 積的高字組你將會得到正確的 16-bit 帶正負號的答案，這卻是真的，當然，積數必須在 -32768 和 +32767 的範圍內。

```
: /         ( n1 n2 -- n )          \ 帶正負號的除法
            /MOD NIP ;

: MOD       ( n1 n2 -- rem )
            /MOD DROP ;

: */MOD     ( n1 n2 n3 -- rem n1*n2/n3 )
            >R                      \ n1 n2
            *D                      \ n1*n2 (32-bit 的暫時積)
            R>                      \ n1*n2 n3
            M/MOD ;                 \ rem quot
```

注意商數等於 n1*n2/n3 ，其暫時的積數 n1*n2 保持 32 位元

```
: */        ( n1 n2 n3 -- n1*n2/n3 )
            */MOD NIP ;
```

金城 註疏：

乘法很容易大於 16 位元(overflow)，所以用 32 位元的暫時積來保持精密度。

取代浮點運算的常數表

Number(數)	Approximation(近似值)	Error(精確度)
π = 3.141 ...	355/ 113	8.5 x 10
√2 = 1.414 ...	19601/13860	1.5 x 10
√3 = 1.732 ...	18817/10864	1.1 x 10
e = 2.718 ...	28667/10546	5.5 x 10
√10 = 3.162 ...	22936/ 7253	5.7 x 10
~S2;12~I;√2 = 1.059 ...	26797/25293	1.0 x 10
log~S1;10~I; 2/1.6384 = 0.183 ...	2040/11103	1.1 x 10
ln 2/16.384 = 0.042 ...	485/11464	1.0 x 10
.001°/22-bit rev = 0.858 ...	18118/21109	1.4 x 10
arc-sec/22-bit rev = 0.309 ...	9118/29509	1.0 x 10
c = 2.9979248	24559/ 8192	1.6 x 10

金城 註疏：

~S2;12~I;√2 常用在音階的計算上，因爲巴哈(Bach)制定了音樂的十二平均律，也就是說每個半音之間相差~S2;12~I;√2 的比例。

假設你想要計算 n1*n2/n3 到最接近的整數，我們可以這樣寫

n1*n2/n3 = Q + R/n3

在這裡 Q 是商數而 R 是餘數

爲了要四捨五入，我們將 1/2 加入答案分數的部分也就是說

n1*n2/n3 = Q + R/n3 + 1/2

這我們可以寫成

n1*n2/n3 = Q + (2*R + n3)/2*n3

然後我們能定義 */R 來計算 n1*n2/n3 的四捨五入值如下：

```
: */R     ( n1 n2 n3 -- n1*n2/n3 rounded )
          DUP 2SWAP              \ n3 n3 n1 n2
          ROT                    \ n3 n1 n2 n3
          */MOD                  \ n3 R Q
          -ROT 2*                \ Q n3 2*R
          OVER +                 \ Q n3 2*R+n3
          SWAP 2*                \ Q 2*R+n3 2*n3
          / + ;
```

5.6 雙倍精密度數字的乘法

有時候將一個雙倍精密度數字(32-位元)乘以一個單一數字(16-位元)而獲得一個雙倍精密度數字的結果是必要的。當然一般來說，如果你將一個 32-bit 數字乘以一個 16-bit 數字，你可以得到一個 48-bit 的積。然而，在許多例子中，你將知道其結果不可能超過 32-bits ，即使它會大於 16-bits 。

假設 A、B、C、D、E、F 和 G 是 16-bit 數字，那麼我們能將 32-bit 數字 A：B 乘以 16-bit 數字 C 的乘法運作敘述如下：

```
            A       B
            -----   -----
                      C
         *            -----
    ----------------------------
              D       E
            -----   -----
       G      F
     -----   -----
   ---------------------------------
              pH      pL
            -----   -----
```

在這個圖形中，B*C 得到 32-bit 結果 D：E 。A*C 得到 32-bit 結果 G：F 。

將這二部分積相加，左移位 16-bit ，如同顯示出來的，會產生完全 48-bit 的積。然而，我們要去掉 G 且限制結果在 32 位元，這個積數的低字組將會是 pL=E ，而其高字組將會是 pH=D+F ，因此，我們可以定義下列詞來做這個乘法。

```
: DUM*    ( ud un -- ud )
          DUP                   \ B A C C
          ROT                   \ B C C A
          *                     \ B C F
          -ROT                  \ F B C
          UM*                   \ F E D
          ROT                   \ E D F
          + ;                   \ pL pH
```

5.7 練習

5.1 一種影像系統使用一電視攝影機來測量球軸承的體積，這系統以像素點(pixel)累計值的觀點來測量球軸承的直徑，最大的像素累計值(相當於一直徑)是 256 。這系統有所調整，所以 100 像素等於 1 公分，這球軸承以此設備來測量，直徑將在 0.25 至 2.5 公分的範圍內。

寫一個 FORTH 的字叫 VOLUME ，這字會期待直徑以像素計數值出現在堆疊(stack)上，且會留下球軸承的容量，以最接近立方毫米的情況出現在堆疊(stack)上。

注意：一個球體的容量是 (4/3)*pi*R^3 ，在這裡 R 是半徑，而 pi 可以接近 355/133

以下列直徑值試驗你的程式：

25　　50　　100　　150　　200　　250

第六課　字　串

6.1　字串輸入

要從終端機中得到一字串，且將它放在 "記憶體某地址" 的緩衝器中，你可以使用下列文字

EXPECT　　(addr len --)

這個字有有限的編輯能力(例如：你可以倒退一格)，且會繼續儲存你鍵入的字的 ASCII 代碼直到 "len" 個長度的字被輸入或是你按 <Enter>鍵。輸入的字母個數被儲存在變數 SPAN 中。

終端機輸入緩衝器的地址儲存在變數 " 'TIB " 中，這字

TIB (--addr)

會將這地址放在堆疊(stack) 頂端。

QUERY 這個字會從鍵盤輸入中得到一字串並將之儲存在終端機輸入緩衝器中。它可以被定義如下：

```
: QUERY    ( -- )
           TIB 80 EXPECT
           SPAN @ #TIB !
           >IN OFF ;
```

變數 #TIB 包含在 TIB 中字母的個數，變數 >IN 是一個指向 TIB 中字母的指標，它最先是被 OFF 這個字設爲 0 ， OFF 將一個 0 儲存在堆疊(stack) 頂端的地址中。

舉個例子，假設要回應 QUERY ，你鍵入下列字母，加上<Enter>

```
3.1415,2.789
```

這些字母的每一個 ASCII 代碼都會被儲存在終端機輸入緩衝器中，以地址 TIB 作爲開始。這 12 個位數會被儲存在變數 #TIB 中。

現在假設你要剖析這個輸入字串，而且要取出被逗點分開的二個數字，你可以用這個字來做。

```
WORD        ( char -- addr )
```

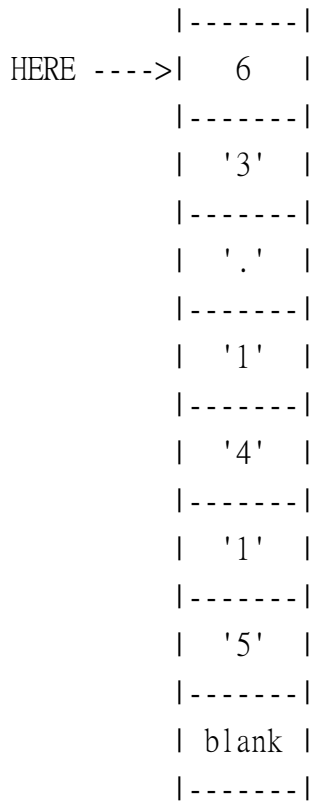
它會剖析輸入字串中的 "特定字母" 且留下一個"計算過長度" 的字串在"某一地址"中(這事實上地址就是 HERE)因此，如果程式執行這個詞。

```
    ASCII , WORD
```

ASCII 這個字會將" , "(逗點)(hex 2c)的 ASCII 代碼放在堆疊(stack)中，然後 WORD 會導致下列位元被儲存在 HERE 中。

金城 註疏：

如果先在記憶體中留下一段空白，並留下地址，則可將字串輸入並儲存在該處備用。



注意一個 "計算過長度" 的字串的第一個位元包含在字串中位元的數目(在這個例子中是 6)，也要注意到 WORD 這個字會附加一個空白的字母 (ASCII 20hex) 在字串的最後面。在這時候變數 >IN 會指向跟隨在逗點後面的記號 (在這個例子中是 2)，下次片語

```
ASCII , WORD
```

被執行時，計算過長度的字串 "2.789" 會被儲存在 HERE 這個地址，即使在這字串最後沒有逗點(comma)，WORD 這個字會剖析到字串的最後，如果它沒有發現分界用的記號字母 "CHAR" 的話。

6.2 ASCII--二進位轉換

假設你輸入數字 3.1415 ，如在第五課中的例子，假如你在直譯(交談)模式中做的話，31415 值會被當做一個雙倍精密度數字儲存在堆疊(stack)中，而在小數點右邊位置的數字的數目" 4 "會被儲存在變數 DPL 中。

你如何可以將同樣的事當做你的部分程式來做，舉例來說，你可能要求使用者，輸入一個數字，而且使這數字最後留在堆疊(stack)中，FORTH 的字 NUMBER，會將一個 ASCII 字串轉換成一個二進位數。它的堆疊圖形看起來像這樣。

```
NUMBER ( ADDR--D )
```

這個字會從 "ADDR" 中取出一個計算過長度的字串，且將結果以一雙倍精密度數字儲存在堆疊(stack)中。這字串可代表在目前的基底(BASE)下，一個帶正負號有小數點的數字。在小數點之後的阿拉伯數字的個數是儲存在變數 DPL 中，如果沒有小數點，DPL 的值是 -1，數字字串必須以一個空白來結束，這就是被 WORD 所設計成的情形。

如果我們想要輸入一個單倍精密度數字(16-bit) ，我們(從鍵盤中)可以定義下列字：

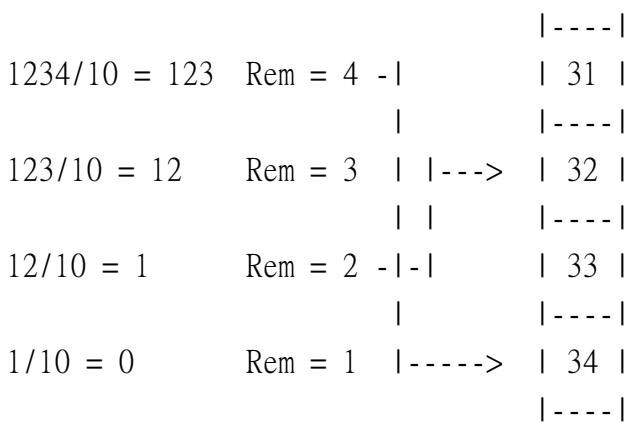
```
: enter.number     ( -- n )
                QUERY
                BL WORD
                NUMBER DROP ;
```

在這定義中，BL 是一個空白(ASCII 20 HEX)的 ASCII 代碼。因此，WORD 將會剖析輸入字串直到一個空白處，或是到字串的尾端。NUMBER 會將這輸入字串轉換成一個雙倍精密度數字並且 DROP 會去掉高位元組而留下單倍精密度數字的值在堆疊(stack)上。注意它的值必須是在 -32768 到 32767 的範圍之內，好使雙倍精密度數字在堆疊(stack)上的高位元組可以確定是 0 。

6.3 數字輸出轉換

要印出數字 1234 在螢幕上，我們必須

(一)以基底數來除。
(二)將餘數轉換成 ASCII，將一個數字字串由尾端向前儲存。
重覆(一)和(二)直到商數 =0



下列 FORTH 的詞被用來執行這個換算，且將結果印在螢幕上，PAD 是一個在 HERE 之上包含 80 位元的暫存器。

```
: PAD      ( --- addr )
           HERE 80 + ;
```

HLD 是一個變數指向儲存在數字字串中最後一個字母

<# 開始數字換算，將數字字串儲存在 PAD 之後的記憶位元中。

```
: <#       ( -- )
           PAD HLD ! ;
```

HOLD 會將字母 "char" 插入輸出字串。

```
: HOLD    ( char -- )
          -1 HLD +!
          HLD @ C! ;
```

FORTH 的字 +! (n addr--)將 n 加入在 addr 中的值。因此，在 HOLD 的定義中，在 HLD 中的值要減 1 ，且然後 ACSII 代碼 "char" 要儲存在 HLD 所指向的位元中。

("sharp")藉執行上述(1)及(2)二步驟而轉換下一個阿拉伯數字。被除數必須是一個雙倍精密度數字。

```
: #       ( d1 -- d2 )
          BASE @ MU/MOD          \rem d2
          ROT 9 OVER             \d2 rem 9 rem
          <
          IF                     \if 9 < rem
             7 +                 \   add 7 to rem
          THEN
          ASCII 0 +              \ conv. rem to ASCII
          HOLD ;                 \ insert in string
```

詞 #S ("SHARP-S")轉換一個雙倍精密度數字的剩餘部分，且留下一個雙倍精密度的 0 在堆疊(stack)中。

```
: #S      ( d -- 0 0 )
          BEGIN
             #                   \ convert next digit
             2DUP OR 0=          \ continue until
          UNTIL ;                \ quotient = 0
```

轉換下一個阿拉伯數字繼續直到商數 =0

#> 完成轉換是藉 2DROP 去掉被 #S 留下的雙倍精密度的 0，然後計算字串的長度從跟在最後一個字母(PAD)後面的地址減去第一個字母(現在 HLD 中)的地址。這字串長度是留在堆疊(stack)上，在字串地址(在 HLD 中)之上。

```
: #>      ( d -- addr len )
          2DROP                  \ drop 0 0
          HLD @                  \ addr
```

```
        PAD OVER                \ addr pad addr
        - ;                     \ addr len
```

FORTH 的字 SIGN 是用來將一個減號 "-" 插入一輸出字串中，如果在堆疊(stack)頂端的值是負號的話。

```
: SIGN    ( n -- )
          0<
          IF
             ASCII - HOLD
          THEN ;
```

這些字會被用在下一個單元，好將數字值展現在螢幕上。

6.4 螢幕輸出

TYPE 這個字印出其地址及長度在堆疊(stack)上的字串。

```
: TYPE    ( addr len -- )
          0 ? DO                \ addr
              DUP C@            \ addr char
              EMIT 1+           \ next.addr
          LOOP
          DROP ;
```

F-PC 實際上對 TYPE 使用一個有些不同的定義，這允許你鍵入一個儲存在任何資料段中的字串。藉著將字串的段地址儲存在變數 TYPESEG 中字串通常被下列三種方式之任一種給確定下來：

一、計算過長度的字串，其第一個位元包含字串中所有字母的個數。字串的定義是有若干個 Byte(位元組)連續被放在 addr(地址)的記憶體中。

二、字串第一個字母的地址以及字串的長度被確定下來(addr len --)

三、一個 ASCII 字串是一個從第一個地址開始連續存放長度位元組的字串(ADDR--)。此字串以一個 nul 文字(ASCII 的第 0 個位元組)和C 語言一樣來結束。

註：此種字串爲 MS/DOS 所採用，其優點爲字串可爲無限長度即大於 256。

一個計算過長度的字串 1. 能被轉換至一個地址和長度 2. 藉使用 FORTH 的字 COUNT 。因此這字

```
 COUNT (ADDR --ADDR+1 LEN)
```

取出一個計算過長度字串的地址(ADDR)，留下字串第一個字母的地址(ADDR+1)，以及字串的長度(這是從在 ADDR 中的位元處得到的)既然 TYPE 需要字串的地址和長度，在堆疊(stack)中，要印出一個計算過長度的字串，你就要使用

```
          COUNT TYPE
```

下列詞會回應你所鍵在螢幕上的任何字串

```
: echo    ( -- )
          QUERY                    \ get a string
          ASCII , WORD
          CR COUNT TYPE ;
```

在 6.3 單元中所提到的使用數字輸出轉換文字可以用來說明不同的數字輸出用詞是如何被定義的。

(u.)這個字轉換一個不帶正負號的單倍精密度數字(16-bit)，且將轉換的字串的地址及長度留在堆疊(stack)中。

```
: (U.)    ( u -- addr len )
          0 <# #S #> ;
```

U. 這個詞將字串印在螢幕上，且後面跟著一個空格。

```
: U.      ( u -- )
          (U.) TYPE SPACE ;
```

SPACE 這個詞印出一個空格，它僅僅是

```
: SPACE   ( -- )
          BL EMIT ;
```

這裡 BL 是常數 32，一個空格的 ASCII 代碼，FORTH 的詞 SPACES (N--)印出 N 個空格。

當將數字一行一行印在螢幕上時，必須要將數字準確地印在寬度 "WID" 的區域中。這可以用 FORTH 的字 U.R 來爲一個未加符號的數字做格式化的輸出。

```
: U.R     ( u wid -- )
          >R (U.)                \ addr len
          R>                     \ addr len wid
          OVER - SRACES          \ addr len
          TYPE ;
```

舉例來說，8 U.R 會將一個不帶正負號的數字準確地印在一個寬度爲 8 的區域中。

要印出一個帶正負號的數目，我們需要在字串的開頭插入一個負號如果這數字是負數的話。(.)這個字將可以這樣做。

```
: (.)     ( n -- addr len )
          DUP ABS
          0 <# #S
               ROT SIGN #> ;
```

・(DOT)這個字也被定義如下：

```
: .       ( n -- )
          (.) TYPE SPACE ;
```

.R 這個字可以被用來將一個帶正負號的數字準確地印在一個寬度 "wid" 的區域中。

```
: .R      ( n wid -- )
          >R (.)                 \ addr len
          R>                     \ addr len wid
          OVER - SPACES          \ addr len
```

```
          TYPE ;
```

相類似的字被定義來印出未帶符號及帶符號的雙倍精密度數字。

```
: ( UD.)  ( ud -- addr len )
          <# #S #> ;

: UD.     ( ud -- )
          (UD.) TYPE STACE ;

: UD.R    ( ud wid -- )
          >R (UD.)              \ addr len
          R>                    \ addr len wid
          OVER - SPACES         \ addr len
          TYPE ;

: ( D.)   ( d -0 addr len )
          TUCK DABS             \ dH ud
          <# #S ROT SIGN #> ;

: D.      ( d -- )
          (D.) TYPE STACE ;

: D.R     ( ud wid -- )
          >R (D.)               \ addr len
          R>                    \ addr len wid
          OVER - SPACES         \ addr len
          TYPE ;
```

要清除螢幕，使用這個字

```
DARK (--)
```

要設定游標在 X,Y 對等的行和列上，使用這個字

```
AT (COL ROW---)
```

舉例來說，下列詞 Example_6.4 將會清除螢幕畫面，並且印出訊息

```
: Example_6.4   ( -- )
                DARK
                20 10 AT
                ." Message starting st col 20 ,row 10 "
                CR ;
```

金城 註疏：

一、在 FORTH 中若在輸入數字中有 , . / - : 等五種標點符號，則被當成雙倍精密度的數字放在堆疊(stack)上。

二、在一般的應用中，格式化的輸出(Formathing Output)經常被使用到。在 FORTH 中這些都需要由程式設計師自行設計。

三、定義一詞，其可將輸入之秒數轉換成時間

```
: .Tel# <# # # # # 45 Hold #S #> Type Space ;
: .Data <# # # 47 HOLD # # 47 HOLD #S #> TYPE Space ;
: SEXTAL 6 BASE ! ;
: :00 # SEXTAL # DECIMAL 58 HOLD ;
: SEC_is <# :00 :00 #S #> TYPE SPACE ;
```

所以鍵入

```
4500. SEC_is
```

則其將顯示下列時間

```
1:15:00 ok
```

第七課　低階(組合)字與 DOS 的輸入/輸出

7.1　低階(組合)(CODE)用字

當 FORTH 的字需要以最快速執行或需要直接存取使用電腦的硬體時，可以使用組合語言的指令來定義。這時可以用 CODE 這個字來定義一個 FORTH 的用字， CODE 字的一般形式如下：

```
CODE <name>                    \   CODE  <名稱>
        <assembly commands>    \   組合語言指令
        <return command>       \   返回方式
        END-CODE               \   END-CODE
```

金城 註疏：

對 FORTH 的虛擬機器(vitual Machine)而言，機器語言(組合) CODE 的意義，就像是 FORTH CPU 的微程式一樣；所以寫 CODE 就是在寫 FORTH 的 Micro Program。

CODE 字是代替冒號，和替 FORTH 的字 <NAME> 建立一個尋找用的字頭，END-CODE 代替分號和結束 CODE 的定義。

金城 註疏：

CODE 會將組合語言字典定爲搜尋字典， END-CODE 會將搜尋字典還原到字典堆疊上原存的地方。

<組合語言指令>可以後置或前置之方法來寫，我們介紹的前置法與標準的 8086/8088 組合語言非常像，在 FORTH 字裡使用 CODE 之前，PREFIX 要先執行。

<返回的方式> 可以是下列任何一種：

一、 NEXT　　JMP　>NEXT

跳至 FORTH 的指令執行機構。在不留下任何堆疊參數時使用 >NEXT。

二、 1PUSH　PUSH AX
　　JMP　>NEXT

將 AX 暫存器推入堆疊當傳回參數然後執行 >NEXT。

三、 2PUSH　PUSH DX
　　PUSH AX
　　JMP　>NEXT

將 DX 和 AX 暫存器上的兩個 16 位元的數堆入堆疊，然後跳至 >NEXT。

使用在 FORTH 課程裡的 8088TUTOR 組合語言除錯器除錯 CODE 的定義會容易一點。

在學習 8088/8086 組合語言的過程中，對於如何使用 TOTOR 軟體在 Richord E. HAXKELL 所著的 〝IBM PC-8088 組合語言程式〞一書中有完整的說明，那本書中會提到執行程式時的命令使用方式。

金城 註疏：

在 FORTH 每一個 CODE 定義均可視爲從高階呼叫低階語言副程式的方式,所以在 END-CODE 時需要返回高階語言的呼叫處。

用 TOTOR 軟體來反組譯和單步執行步驟爲例，透過 CODE 字來思考 FORTH 詞 CMOVE 將一列已知<長度>的位元組從<來源>位址移到<目的> 位址。

```
CODE CMOVE      ( source dest count -- )
        CLD                     \ 設定移動記憶體的方向
        MOV     BX, SI          \ BX ← SI (IP)
```

```
        MOV     AX, DS          \ AX ← DS
        POP     CX              \ CX = 長度
        POP     DI              \ DI = 目的地址
        POP     SI              \ SI = 來源地址
        PUSH    ES              \ 儲存 ES
        MOV     ES, AX          \ ES = AX
        REPNZ                   \ 重覆下列步驟直到長度爲 0
        MOVSB                   \ 將 DS:SI 的記憶體內容搬到 ES:DI
        MOV     SI, BX          \ 還原 SI
        POP     ES              \ 還原 ES
        NEXT                    \ 完成以上工作跳到 >NEXT
        END-CODE
```

金城 註疏：

此程式與 F-PC 的定義不完全一樣，NEST 是錯的已改成 NEXT。

當你 FLOAD 此課以下的 FORTH CODE 定義時，將在位移地址 (offset address)〝SOURCE.ADDR〞 中儲存十六進位數 11 22 33 44 55 到 CODE SEGMENT， CODE SEGMENT 的真正段位址是由 FORTH 字 ?CS: 給予。而且當鍵入 " SHOW.ADDRS " 時會印在螢幕上。

有五個位元會保留給〝dest.addr〞的位移地址。
當你鍵入" show.addrs " 時則程式段(code segment)，" source.addr "和 " dest.addr " 和堆疊(stack)頂端的值及 CMOVE 的執行地址 (CFA) 都會印在螢光幕上。

```
HEX
CREATE source.addr 11 C, 22 C, 33 C, 44 C, 55 C,
CREATE dest.addr   5 ALLOT
5 CONSTANT #bytes
: test          ( -- )
          source.addr dest.addr #bytes CMOVE ;

: show.addrs  ( -- )
        HEX
```

```
CR ." code segment = " ?cs: u.
CR . " source addr = " source.addr u.
CR . " dest addr = " dest.addr u.
CR . " top of stack = " SP0 @ U.
CR . " address of CMOVE = " [ ' CMOVE ] LITERAL U.
CR DECIMAL
```

註: [,] 和 LITERAL 兩字，將於第九課討論。

當你鍵入 " SHOW.ADDRS "，假設其數值印出如下：

```
code segment = E74
source addr = 6929
dest addr = 6931
top of stack = FFE2
address of CMOVE = 41C
```

你的數值可能不同，如果不同，請以你自己的數字做以下的練習：

鍵入 debug test.
鍵入 test.

經由開頭的三個字會將 6929　6931　5 三個數留在堆疊(stack)上。

按 F 進入 FORTH
鍵入 SYS TUTOR 將會執行 TUTOR 軟體
進入 TUTOR 程式的顯示畫面後

鍵入 >SE74　以顯示程式段的內容

鍵入 /GSE74　以顯示資料段的內容，在此資料段與程式段共用

鍵入 /GO6929　以顯示資料段中來源位址的內容
11 22 33 44 55 會出現在螢幕上

鍵入 /RSSE74　設定堆疊段等於程式段

鍵入 /RPSFEDC　以設定 SP 暫存器等於 FFE2-6 (FFE2) 減 6

鍵入 >041C　　將 IP (執行指標)定在 CMOVE 的開始處 41C

按 F1 鍵將會單一步驟執行此程式

注意當你執行到 REP 指令，按 F1 鍵五次，會從來原位址移五個位元組的內容到目的位址。

要離開 TUTOR 時打入 /QD，會自動返回 DOS，如果你尚未改變堆疊(stack)(拿出 FORTH 的詞 TEST 放入堆疊(stack)的數字)那麼從 TUTOR 打入 /QD ，會回到在 FORTH 中鍵入 SYS TOTOR 的地方。

FORTH 的字 CMOVE> (source dest count) 除位元是反方向移動之外，與 CMOVE 是相同的，也就是說最高位址的位元組先移動，如果記憶體的來源區和目的區重疊而你想移動此區塊內容的話，使用這個字。

如果不幸使用了 CMOVE 來做此區塊內容的搬移，那在移動之時，會使來源區的重疊部份被自己蓋過而銷毀。

7.2 低階(CODE) 的條件判別用字

以 FORTH 的結構化用字 IF---ELSE---THEN ，BEGING---WHILE---REPEAT 和 BEGIN---UNTIL 與以下之低階指令一起使用，可以取代組合語言的指令。

Forth	Assembled Code
0=	JNE/JNZ
0<>	JE/JZ
0<	JNS
0>=	JS
<	JNL/JGE
>=	JL/JNGE
<=	JNLE/JG
>	JLE/JNG
U<	JNB/JAE/JNC
U>=	JB/JNAE/JC
U<=	JNBE/JA

U>	JBE/JNA
OV	JNO

金城 註疏：

一、在 FORTH 的低階定義(副程式)中，使用結構化的指令來完成單進單出的結構完整性。

二、如果您喜歡耍一些組合語言使用標記跳躍的技巧，在 FORTH 的低階中，也可以使用 Label

舉例，來思考一下 FORTH 的字 ?DUP 的定義，當此數字不是 0 時它可以複製堆疊(stack)頂端的數字。

```
CODE    ?DUP    ( n -- n n | 0 )
                POP     AX
                CMP     AX, # 0
                0<>
                IF
                   PUSH AX
                THEN
                1PUSH          \ 在 1PUSH 中會 PUSH AX
                END-CODE
```

注意在 CODE 的組合定義中 0<> 的敘述會組譯成指令跳躍到 THEN 之後的指令位址。

金城 註疏：

在 FORTH 中每一個字(用空白分開的獨立單字)皆有其自身的動作定義所以和其他語言的語法不同。

7.3 長距離(跨段)的記憶體存取用字

下面之長距離的記憶體存取用字的存取範圍可在程式段(code segment)之外

，對於大型的資料結構應用較有用。

```
CODE    @L      ( seg off -- n )        \ 從段(seg):位移(off)
                                        \ 抓取 16 位元的值
                POP     BX              \ BX = 位移地址
                POP     DS              \ DS = 段址
                MOV     AX, 0[BX]       \ AX = 取得資料
                MOV     BX, CS          \ 還原 DS 等於 CS
                MOV     DS, BX
                1PUSH                   \ 將此值堆入堆疊
                END-CODE

CODE    !L      ( n seg off -- )        \ 在段(seg):位移(off)存
                                        \ 入 16 位元的值
                POP     BX              \ BX = 位移地址
                POP     DS              \ DS = 段址
                POP     AX              \ AX = 要存入的值
                MOV     0[BX],AX        \ Store n at DS:BX
                MOV     BX, CS          \ 還原 DS 等於 CS
                MOV     DS, BX
                NEXT
                END-CODE
```

以下是其他有用的長距離的記憶體存取用字。

C@L (seg off -- byte)
由段(seg):位移(off)抓取 8 位元的位元組

C!L (byte seg off --)
在段(seg):位移(off)存入 8 位元的位元組

CMOVEL (sseg soff dseg doff count)
從來源段(sseg):來源地址(soff)搬動一長度(count) 的記憶內容到目的段(dseg):目的地址(doff)處

CMOVEL> (sseg soff dseg doff count)
效果同 CMOVEL ，但從最高位址先搬以避免區塊重疊的自毀現象

7.4 DOS 的介面用詞

F-PC 有很大的 FORTH 字群，來操作 DOS 的檔案 I/O，這些字是由來程式檔 HANDLES.SEQ 和 SEQREAD.SEQ 加以定義，在這一節和下一節裡，我們將發展出一套檔案，I/O 的字可以用來擴大操作各式的檔案，I/O 和 DOS 其他的運用，這些字可取代 F-PC 的 DOS 和檔案 I/O 字或與 F-PC 的原始定義字群合併使用。

```
VARIABLE ITEMS          \ 用來記錄堆疊(stack)的深度
VARIABLE handl          \ 檔案的操作識別碼 (ID)
VARIABLE eof            \ 當讀至檔案結束(eof)時爲 TRUE (-1)
CREATE fname  80 ALLOT  \ 包含 ASCII 檔名的 80 位元組的緩衝區

: {     ( -- )
                DEPTH ITEMS ! ;

: }     ( -- c )
                DEPTH ITEMS @ - ;
```

金城 註疏：

在 F-PC 中大括弧已有 "執行部份" 的編譯定義

{ . . . }　　用來記載在堆疊(stack)上元素的數字。例如：

{ 5 2 8 }

將在堆疊(stack)的頂端留下下列的數字。

5 2 8 3

堆疊(stack)頂端的 3 是介於 { } 部份的數字個數。

```
: $>asciiz      ( addr1 -- addr2 )
                DUP C@ SWAP 1+
```

```
TUCK +
0 SWAP C! ;
```

將 FORTH 計算過長度的字串轉換成 DOS 或其他語言如 C 使用的 ASCII 字串不計算長度，但以 0 來結束的字串。

2FDOS DOS 2.0 版以後的磁碟輸入輸出的功能呼叫以 AX=AH:AL 和 CX DX 呼叫 DOS 系統服務中斷 INT 21H 的功能，送回 AX DX 和錯誤旗號到堆疊(stack)頂端 。

如果錯誤旗號是 TRUE 則錯誤編號是在 AX (堆疊(stack)的第三個元素)裡，如錯誤旗號是 FALSE，那麼 AX 和 DX 的內容會依功能呼叫的不同而有不同的傳回值。

FDOS 相似於 2FDOS ，但並不傳回錯誤旗號，被用在於非以進位旗標來指出錯誤的 DOS INT 21H 系統呼叫。

```
PREFIX
HEX

CODE  2fdos     ( ax bx cx dx -- ax dx f )
                POP     DX
                POP     CX
                POP     BX
                POP     AX
                INT     21              \ DOS 系統服務呼叫
                U>=
                IF                      \ 如果進位旗標爲 0
                   MOV  BX, # FALSE     \ 則設錯誤旗號爲僞
                ELSE                    \ 否則
                   MOV  BX, # TRUE      \ 設錯誤旗號爲真
                THEN
                PUSH    AX
                PUSH    DX
                PUSH    BX
                NEXT
                END-CODE

CODE  fdos      ( ax bx cx dx -- ax dx )
```

```
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        INT     21              \ DOS 系統服務呼叫
        PUSH    AX
        PUSH    DX
        NEXT
        END-CODE

 DECIMAL
```

7.5 基本的檔案讀/寫

下列的字可以用來操作基本的檔案輸入/輸出，例如開啓、建立、關閉和刪除檔案以及從磁碟檔案之中讀寫字元。

```
open.file     ( addr -- handle ff | error.code tf )
```

開啓一個檔，在 FALSE 旗號之下傳回操作識別碼，或在 TRUE 旗號之下回到錯誤碼號。

以 ADDR 來指向 ASCIIZ 之字串。存取代碼設定爲 2 以打開一個可讀寫的檔。

```
HEX

: open.file    ( addr -- handle ff | error.code tf )
                3D02                \ ah = 3D ; al = 存取代碼
                0 ROT 0 SWAP        \ 3D02 0 0 addr
                2fdos               \ DOS 系統呼叫
                NIP ;               \ 拋除 DX

close.file
```

用堆疊(stack)上的操作識別碼來關閉檔，如果不能關閉該檔則印出錯誤訊息。

```
: close.file    ( handle -- )
                3E00                    \ ah = 3E
                SWAP 0 0                \ bx = 檔案操作識別碼
                2fdos
                NIP                     \ 拋除 DX
                IF
                   ." Close error number " . ABORT
                THEN
                DROP ;
```

reat.filec　　建立檔案，如同在開啓檔一般傳回檔案識別碼。
以 ADDR 來指向 ASCIIZ 之字串。
ATTR 是檔案屬性 : 0 - 標準檔
01H - 唯讀檔　　02H - 隱藏檔
04H - 系統用檔　　08H - 磁碟命名標記
10H - 子目錄　　20H - 檔案集

```
: create.file   ( addr attr -- handle ff | error.code tf )
                3C00                    \ ah = 3C
                0 2SWAP SWAP            \ 3C00 0 attr addr
                2fdos
                NIP ;                   \ nip dx
```

open/create 開啓一個存在的檔，否則就建立一個新的檔，以 ADDR 來指向 ASCIIZ 的字串，傳回開啓檔的操作識別碼 (HANDLE)，如果不能開啓，則印出錯誤訊息。

```
: open/create  ( addr -- handle )
               DUP open.file
               IF
                  DUP 2 =
                  IF
                     DROP 0 create.file
                     IF ." Create error no. " . ABORT
                     THEN
                  ELSE
                     ." Open error no. " . DROP ABORT
```

```
   THEN
ELSE
   NIP
THEN ;

: delete.file  ( addr -- ax ff | error.code tf )
               4100
               0 ROT 0 SWAP
               2fdos
               NIP ;

: erase.file   ( $addr -- )    \ 刪除檔案其檔名以計算過長度
               $>asciiz        \ 的字串存在 $addr
               delete.file
               IF
                  CR ." Delete file error no. " .
               ELSE
                  DROP
               THEN ;
```

read.file　　從" HANDLE "的檔中讀取" #BYTES "個位元組，進入緩衝區傳回正確讀取的 #BYTES ，假如此數字是 0 ，那麼檔案已經讀取完畢，如果失敗，則印出錯誤訊息。

```
: read.file    ( handle #bytes buff.addr -- #bytes )
               >R 3F00              \ handle #bytes 3F00
               -ROT R>              \ 3F00 handle #bytes addr
               2fdos
               NIP                  \ nip dx
               IF
                  ." Read error no. " . ABORT
               THEN ;
```

write.file　　從" BUFF.ADDR "所指的緩衝區寫入" #BYTES "個位元組到" HANDLE "所代表的檔中，如果失敗則印出錯誤訊息。

```
: write.file    ( handle #bytes buff.addr -- )
                >R 4000           \ handle #bytes 4000
```

```
                -ROT R>          \ 4000 handle #bytes addr
                2fdos
                NIP              \ nip dx
                IF
                   ." Write error no. " . ABORT
                ELSE
                   DROP
                THEN ;
```

mov.ptr　　移動" HANDLE "所代表的檔案指標 doffset 是一雙倍精密度的數字 (32-位元) 位移 CODE 是 method code

0 - 移動檔案指標至檔案開始處加位移
1 - 直接將現在的檔案指標加上位移
2 - 將檔案指標移至檔案結尾前的位移處

```
: mov.ptr       ( handle doffset code -- dptr )
                42 FLIP +          \ hndl offL offH 42cd
                ROT >R             \ hndl offH 42cd
                -ROT R>            \ 42cd hndl offH offL
                2fdos
                IF
                   DROP ." Move pointer error no. " . ABORT
                THEN ;
```

rewind.file　　移動" HANDLE "所代表檔案的檔案指標指至檔案之開始處。(相當於迴轉錄音帶)

```
: rewind.file   ( handle -- )
                0 0 0 mov.ptr 2DROP ;
```

get.length　　傳回" HANDLE "所代表檔案的 32 位元檔案長度處。

```
: get.length    ( handle -- dlength )
                0 0 2 mov.ptr ;
```

read.file.L　　從" HANDLE "所開啓的檔裡讀取" #BYTES "個位元組，而且

儲存這些位元組至段(SEG):位移 (OFFSET) 所開始的延伸記憶體裡。

```
CODE read.file.L        ( handle #bytes seg offset -- ax f )
                POP     DX
                POP     DS
                POP     CX
                POP     BX
                MOV     AH, # 3F
                INT     21
                U>=
                IF
                   MOV  BX,  # FALSE
                ELSE
                   MOV  BX, # TRUE
                THEN
                MOV     CX, CS          \ 還原 DS
                MOV     DS, CX
                PUSH    AX
                PUSH    BX
                NEXT
                END-CODE
```

write.file.L　從 SEG:OFFSET 的延伸記憶體裡寫入" #BYTES "個位元組到以" HANDLE "爲代碼所開啓的檔中。

```
CODE write.file.L       ( handle #bytes seg offset -- ax f )
                POP     DX
                POP     DS
                POP     CX
                POP     BX
                MOV     AH, # 40
                INT     21
                U>=
                IF
                   MOV  BX, # FALSE
                ELSE
                   MOV  BX, # TRUE
                THEN
```

```
        MOV     CX, CS          \ 還原 DS
        MOV     DS, CX
        PUSH    AX
        PUSH    BX
        NEXT
        END-CODE
```

findfirst.dir　尋找在目錄下第一個吻合" ADDR "所指裡的 ASCIIZ 字串型的檔名。

```
CODE    findfirst.dir ( addr --  f )  \ 找到第一個吻合的檔名
        POP     DX              \ dx = 字串的地址
        PUSH    DS              \ 儲存 ds
        MOV     AX, CS
        MOV     DS, AX          \ ds = cs
        MOV     CX, # 10        \ 型態內容包含子目錄
        MOV     AX, # 4E00      \ ah = 4E
        INT     21              \ DOS 系統呼叫
        JC      1 $             \ if no error
        MOV     AX, # FF        \   flag = TRUE
        JMP     2 $             \ else
1 $:    MOV     AX, # 0         \   flag = FALSE
2 $:    POP     DS              \ 還原 ds
        PUSH    AX              \ 將旗號壓入堆疊(stack)
        NEXT
        END-CODE
```

findnext.dir　尋找下一個在目錄中吻合" ADDR "所指向的 ASCIIZ 字串所含的檔案敘述。

```
CODE    findnext.dir ( --  f )        \ 尋找下一個在目錄中吻合的
                                      \ 檔名
        PUSH    DS              \ 儲存 ds
        MOV     AX, CS
        MOV     DS, AX          \ ds = cs
        MOV     AX, # 4F00      \ ah = 4F
        INT     21              \ DOS 系統呼叫
        JC      1 $             \ if no error
        MOV     AX, # FF        \   flag = TRUE
        JMP     2 $             \ else
```

```
        1 $:    MOV     AX, # 0         \   flag = FALSE
        2 $:    POP     DS              \ restore ds
                PUSH    AX              \ 將旗號壓入 stack
                NEXT
                END-CODE
```

set-dta.dir　設定磁碟輸送區域地址

```
CODE   set-dta.dir   ( addr -- )   \ set disk transfer area
                                   \ address
               POP     DX          \ dx = 磁碟輸送區域地址
               PUSH    DS          \ 儲存 ds
               MOV     AX, CS
               MOV     DS, AX      \ ds = cs
               MOV     AX, # 1A00  \ ah = 1A
               INT     21          \ DOS 系統呼叫
               POP     DS          \ 還原 ds
               NEXT
               END-CODE
DECIMAL
```

7.6 讀數字和字串

下列的字可以用來讀出磁片檔案中的位元組數字和字串。

get.fn　從鍵盤輸入檔名，且以 ASCIIZ 字串的型式，在 fname 裡

```
: get.fn        ( -- )
                QUERY BL WORD       \ addr
                DUP C@ 1+           \ addr cnt+1
                2DUP +              \ addr len addr.end
                0 SWAP C!           \ 產生 ASCIIZ 字串
                SWAP 1+ SWAP        \ addr+1 len
                fname SWAP          \ from to len
                CMOVE ;
```

open.filename　輸入一個檔名，開啓它，然後儲存它的操作代碼於變數的 " HANDL "中。

```
: open.filename         ( -- )
                get.fn
                fname open/create
                handl ! ;
```

eof?　　如果讀到檔案結束的標記 (EOF = True) 則立刻跳出(EXIT)包括 eof? 的定義句子中。

```
: eof?          ( -- )
                eof @
                IF
                   2R> 2DROP EXIT
                THEN ;
```

get.next.byte　　從" HANDL "所代表的磁片檔中取入下一個位元組，假如成功則設定變數 EOF 爲僞，否則設定變數 EOF 爲真。

```
: get.next.byte         ( -- byte )
                handl @ 1 PAD read.file
                IF
                   FALSE eof ! PAD C@
                ELSE
                   TRUE eof !
                THEN ;
```

get.next.val　　從" HANDL "所代表的磁片檔中取入下一個 16 位元組的數字如果失敗則設定變數 EOF 爲真，用於資料如果是真正的數字此數字儲存在磁片檔案中不同於 ASCII 的資料。

```
: get.next.val          ( -- n )
                handl @ 2 PAD read.file
                IF
                   FALSE eof ! PAD @
                ELSE
                   TRUE eof !
                THEN ;
```

get.next.dval　從" HANDL "所代表的磁片檔中取入下一個 32 位元組的數字

```
: get.next.dval          ( -- d )
                handl @ 4 PAD read.file
                IF
                   FALSE eof ! PAD 2@
                ELSE
                   TRUE eof !
                THEN ;
```

parenchk　假如在堆疊(stack)上之位元組是 " (" 則讀取檔案直至下一個 ") " 出現，假如 EOF 被讀入則離開。

```
: parenchk      ( byte -- byte )   \ 小括號註解的略過式，檢查是
                                   \ 否註解部分已結束
                DUP ASCII ( =
                IF
                   DROP
                   BEGIN
                      get.next.byte eof?
                      ASCII ) =
                   UNTIL
                   get.next.byte eof?
                THEN ;
```

quotechk　如果堆疊(stack)上之位元組是一個(")則讀取檔案至下一個(")出現。假如 EOF 被讀入則離開。

```
: quotechk      ( byte -- byte )   \ 略過式檢查 FORTH 的字串是否
                DUP ASCII " =      \ 已結束
                IF
                   DROP
                   BEGIN
                      get.next.byte eof?
                      ASCII " =
```

```
                UNTIL
                get.next.byte eof?
            THEN ;
```

?digit　　檢查堆疊(stack)的位元組是否是在目前的基底下一個正確 ASCII 阿拉伯數字

```
: ?digit        ( byte -- byte f )
            DUP BASE @ DIGIT NIP ;
```

get.next.digit　從磁碟檔裡，取入下一個正確的 ASCII 阿拉伯數字，如果檔案已結束則離開。

```
: get.next.digit        ( -- digit )
            BEGIN
                get.next.byte eof?
                parenchk eof?
                quotechk eof?
                ?digit NOT
            WHILE
                DROP
            REPEAT ;
```

get.digit/minus　讀入下一個有效的 ASCII 數字 (0 .. 9)或是一個負號 (-)，如果檔結束則離開。

```
: get.digit/minus        ( -- digit or - )
            BEGIN
                get.next.byte eof?
                parenchk eof?
                quotechk eof?
                DUP ASCII - =
                SWAP ?digit ROT OR NOT
            WHILE
                DROP
            REPEAT ;
```

get.next.number　從磁碟檔中讀入一個帶正負號的整數字串，而後轉換真正的整數留在堆疊(stack)上，如果檔案已到結束點則離

開。

```
: get.next.number       ( -- n )
          { get.digit/minus eof?      \ 使用 {  } 來儲存
          BEGIN                       \ 連貫的數字位數在
             get.next.byte eof?       \ 堆疊(stack)上
             parenchk eof?            \ 省略過 (...) 註解
             quotechk eof?            \ 和 "..." 字串等
             ?digit NOT
          UNTIL
          DROP }
          DUP PAD C!
          DUP PAD + BL OVER 1+ C!
          SWAP 0 DO                   \ 將數字串的個數移到
                                      \ 堆疊(stack)
             SWAP OVER C! 1-          \ 轉換出已計算過長度
                                        的字串
                                      \ 放在 PAD 上
          LOOP
          NUMBER DROP ;               \ 將其轉成數字
```

?period　　檢查看看如果一個位元組是一個句點，則旗號會以第二個元素被留在堆疊(stack)上

```
: ?period       ( byte -- f byte )
                DUP ASCII . = SWAP ;
```

get.next.dnumber　讀入下一個以 ASCII 字串存在磁碟上帶正負號的數字，然後將其轉成帶正負號之雙倍精密度數字並將留在堆疊(stack)上。在小數點後面的數字的位數將被留在變數 DPL 中，如果已讀至檔案結束之處則離開(脫離母程序)

```
: get.next.dnumber       ( -- dn )
           { get.digit/minus eof?
           BEGIN
              get.next.byte eof?
              parenchk eof?           \ 此程式與
              quotechk eof?           \ get.next.number 相
```

```
      ?period                 \ 似但在數字字串中
      ?digit ROT OR NOT       \ 包括句點的處理
   UNTIL
   DROP }
   DUP PAD C!
   DUP PAD + BL OVER 1+ C!
   SWAP 0 DO
      SWAP OVER C! 1-
   LOOP
   NUMBER  ;                  \ 轉換成雙倍精密度數字
```

get.next.string　讀入下一個在磁碟檔案中由雙引號" "之間的字串將其以計算過長度的字串型式儲存在" ADDR "中所指到的地方。

```
: get.next.string       ( -- addr )  \ counted string
      BEGIN
         get.next.byte eof?
         ASCII " =
      UNTIL
      0 PAD 1+
      BEGIN                     \ cnt addr
         get.next.byte eof?
         DUP ASCII " <>
      WHILE
         OVER C!
         SWAP 1+ SWAP
         1+
      REPEAT
      2DROP PAD C! PAD ;
```

7.7 寫入數字和字串

send.byte　寫入一個位元組到一個檔案操作代碼(HANDLE)被儲存在" handl " 中已開啓的磁碟檔案

```
: send.byte     ( byte -- )
```

```
PAD C!
handl @
1 PAD write.file ;
```

send.number 以 ASCII 字串方式寫入一個帶正負號 16-bit 整數到一個檔案操作代碼(HANDLE)被儲存在" handl "中已開啓的磁碟檔案

```
: send.number   ( n -- )
                (.) 0
                DO
                   DUP C@ send.byte
                   1+
                LOOP
                DROP ;
```

send.number.r 以 ASCII 字串方式寫入一個帶正負號 16-bit 整數到一個其檔案操作代碼 (HANDLE)被儲存在" handl "中已開啓的磁碟檔案中，這個數字將以其總欄位寬度爲 "len"向右靠齊，前面的不足位數將以 ASCII 碼空白填補。

```
: send.number.r    ( n l -- )
                >R (.) R>
                OVER -
                0 DO
                   BL send.byte
                LOOP
                0 DO
                   DUP C@ send.byte 1+
                LOOP
                DROP ;
```

send.dnumber 以 ASCII 字串方式寫入一個帶正負號 32-bit 的整數到一個其檔案操作代碼(HANDLE) 被儲存在 " handl " 中已開啓的磁碟檔案中。小數點將依變數 DPL 的內容來決定位置。

```
: send.dnumber  ( d -- )                \ DPL = 小數點後面的位數
                TUCK DABS <# DPL @ ?DUP
                IF
```

```
                0 DO # LOOP
                ASCII . HOLD
             THEN
             #S ROT SIGN #>
             0 DO
                DUP C@ send.byte 1+
             LOOP DROP ;

: send.val      ( n -- )                    \ 寫入 16-bit 數值
             PAD ! handl @
             2 PAD write.file ;

: send.dval      ( d -- )                   \ 寫入 32-bit 數值
             PAD 2! handl @
             4 PAD write.file ;

: send.string   ( addr -- )                 \ 已計算過長度的字
             DUP C@                         \ 串的地址
             SWAP 1+ SWAP
             0 DO
                DUP I + C@
                send.byte
             LOOP
             DROP ;

: send.crlf     ( -- )
             13 send.byte
             10 send.byte ;

: send.lf     ( -- )
             10 send.byte ;

: send.cr       ( -- )
             13 send.byte ;

: send.tab      ( -- )
             9 send.byte ;

: send.(        ( -- )
```

```
                ASCII ( send.byte ;

: send.)        ( -- )
                ASCII ) send.byte ;

: send.,        ( -- )
                ASCII , send.byte ;

: send."        ( -- )
                ASCII " send.byte ;

: send."string"   ( addr -- )
                send."
                send.string
                send." ;
```

第八課　定　義　詞

8.1 CREATE...DOES>

FORTH 詞 CREATE...DOES>是用來建立"定義詞"(defining words)，也就是說，詞可以用來定義新的詞群。定義詞最大的特點是它們設定未來所建立之詞群的執行機構。

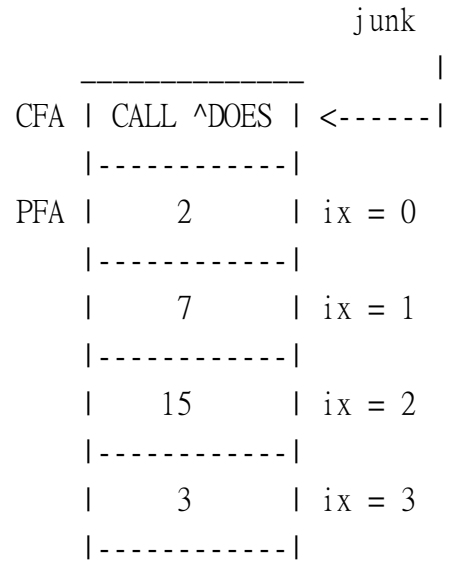
我們將藉著建立定義詞 table 來說明 CREATE....COES> 的用法

```
: table         ( list n -- )
                CREATE
                  0 DO
                     C,
                  LOOP
               DOES>   ( ix -- c )
                  + C@ ;
```

此字可以用來定義新字 junk 如下：

```
3 15 7 2 4 table junk
```

當 table 被執行時，在 table 定義裡，介於 CREATE 和 DOES>間之 FORTH 詞亦被執行，這將造成 junk 這個詞被加於含有下列數字之詞典中，這些數字是儲存在 junk 的 pfa 中



Code Segment ?CS:
執行區段

junk 的執行碼區(code filed)包含一個 CALL 機械指令指到 table 定義中 DOES> 後之該詞被執行時的機械碼(machine code)，當該指令執行時，junk 的 PFA 會被推入堆疊(stack)上，而前面的 ix 索引會先被推入堆疊(stack)中，此索引會與 pfa 相加然後 C@會讀取(fetch)那個位置的位元組，例如:

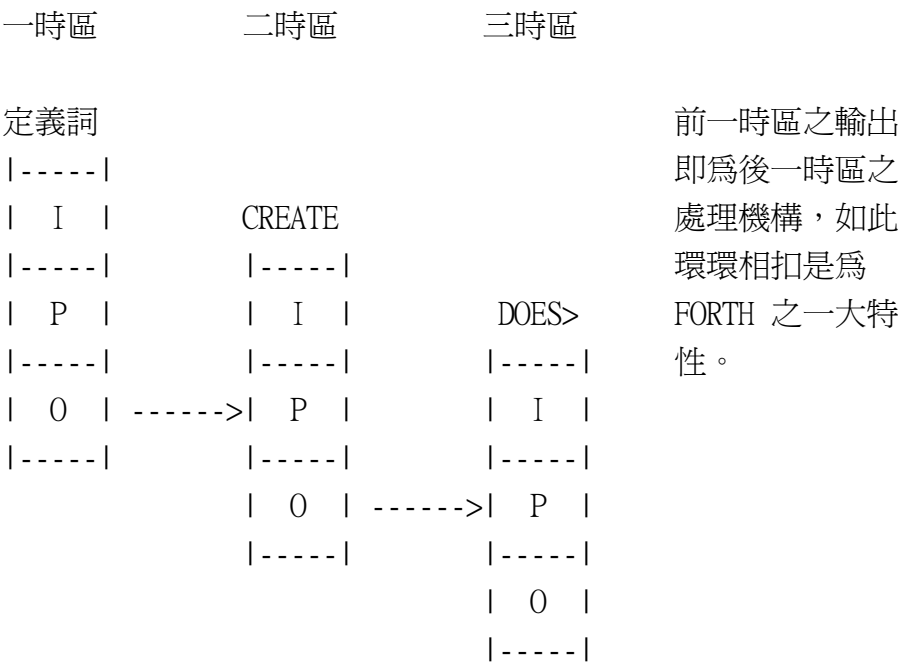
2 junk ·

會印出 15

金城 註疏：

一、CREATE...DOES> 爲 FORTH 中建立資料結構的方法，在 FORTH 中雖然並未提供預先定義好的資料結構，如陣列、記錄指標----等，但藉著定義詞來規劃記憶體管理模式，可完成所有各式應用所需之資料結構。

二、定義詞的時態區分：



.本文

CREATE...DOES>工作的方式如下。當"table"被定義時，產生下面的詞典結構

.本文 p1

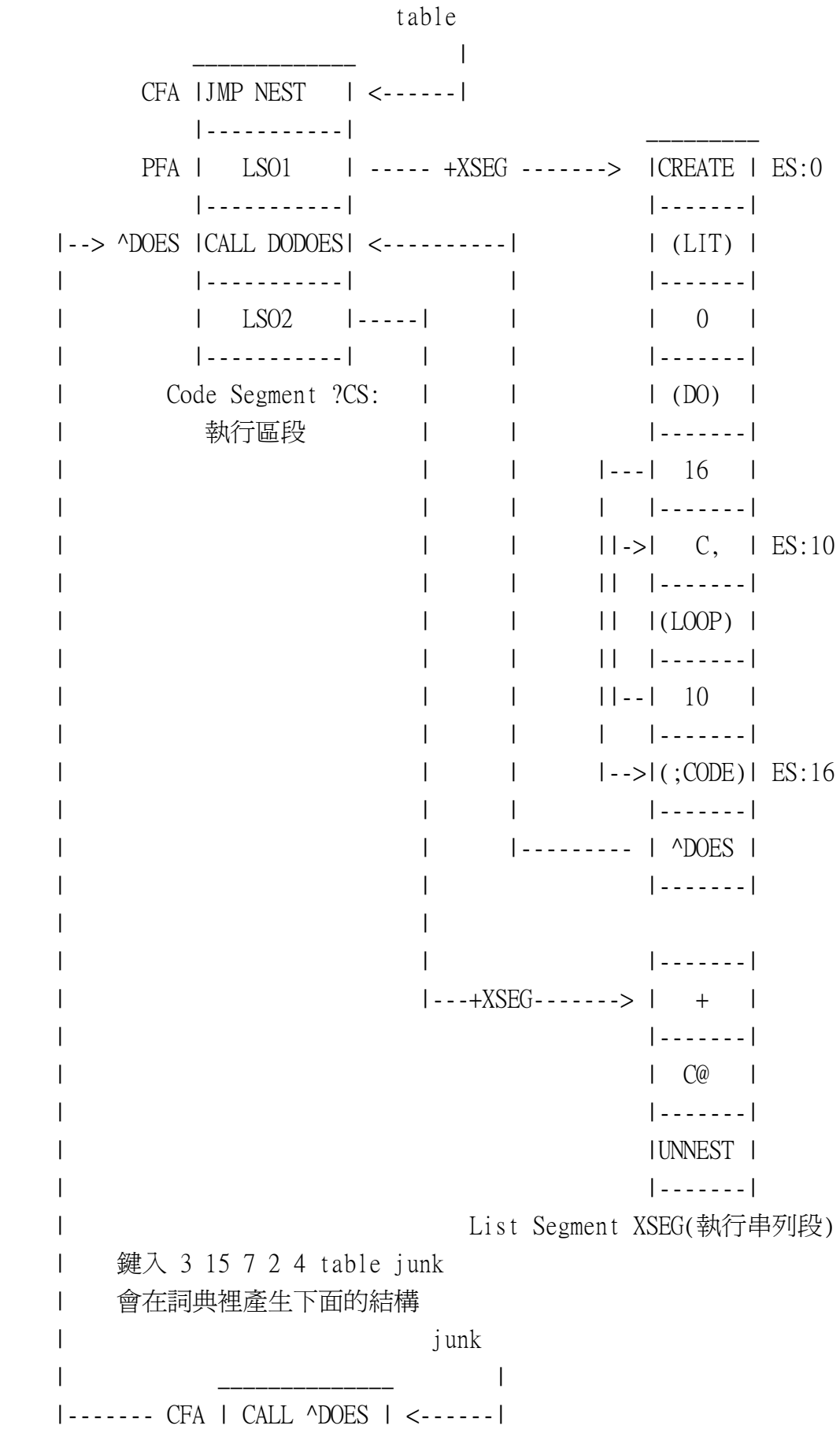
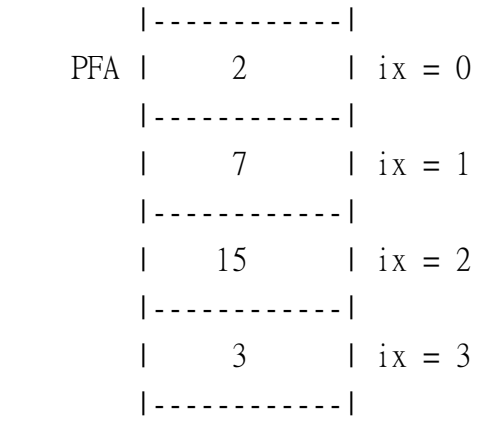


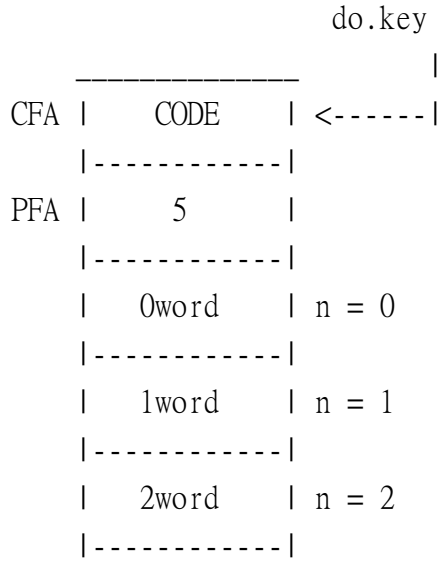
table
CFA
JMP NEST
PFA
LSO1
+XSEG
CREATE
ES:0
^DOES
CALL DODOES
LSO2
(LIT)
0
(DO)
Code Segment ?CS:
執行區段
16
C,
ES:10
(LOOP)
10
(;CODE)
ES:16
^DOES
+XSEG
+
C@
UNNEST
List Segment XSEG(執行串列段)
鍵入 3 15 7 2 4 table junk
會在詞典裡產生下面的結構
junk
CFA
CALL ^DOES

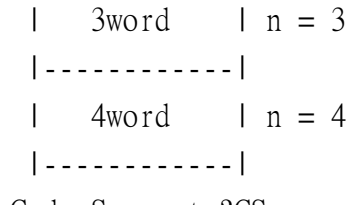


注意："junk"的執行碼區(code field)包含一個 CALL 的指令，該指令呼叫在"table" 的 PFA 之後之指令 CALL DODOES (當(;CODE)在"table"的執行串列段(list segment)裡執行時，這個 CALL ^DOES 指令會插入"junk"的執行碼區裡)。這裡有兩個影響，第一、它將 junk 的 PFA 放進堆疊(stack)中，第二、它執行 CALL DODOES 的敘述。一般此 CALL DODOES 是用來執行被建立之詞放在執行串列段(list segment)的指標 LSO2 所指的 CFA 中。這些就是緊接在 table 定義中，DOES> 之後的敘述，非常重要的一點，任何用 table 所定義出來之詞群，在執行時都會執行相同的敘述，這是一個非常重要的特點，我們在以下的各種不同型態的跳躍表格中，再加以說明。

8.2 簡易跳躍表格

舉一例說明定義詞之使用：假設你想要建立一個稱作"do.key"如下列形式之跳躍表格(jump table)：





可能有些簡單的應用機會，例如，如果你有一個小鍵盤，上有 0-4 的鍵，當適當的鍵被壓下時，它將會傳回 0-4 的值在堆疊(stack)上，可被用來執行 FORTH 的詞 0word 、1word 、....4word 等，因爲這些詞的 CFA 被儲存在跳躍表格中。

我們將定義一個定義詞稱作 JUMP.TABLE 可以用來建立"do.key"或任何其他類似的跳躍表格，要產生"do.key"我們可鍵入

```
5 JUMP.TABLE do.key
        0word
        1word
        2word
        3word
        4word
```

下列是 JUMP.TABLE 的定義將完成的工作 :

```
: JUMP.TABLE            ( n -- )
      CREATE
         DUP ,
         0 ?DO
            ' ,
         LOOP
      DOES>             ( n pfa -- )
         SWAP 1+ SWAP           \ n+1 pfa
         2DUP @ >               \ n+1 pfa (n+1)>nmax
         IF
            2DROP
         ELSE
            SWAP                \ pfa n+1
            2* +                \ addr = pfa + 2(n+1)
            PERFORM
```

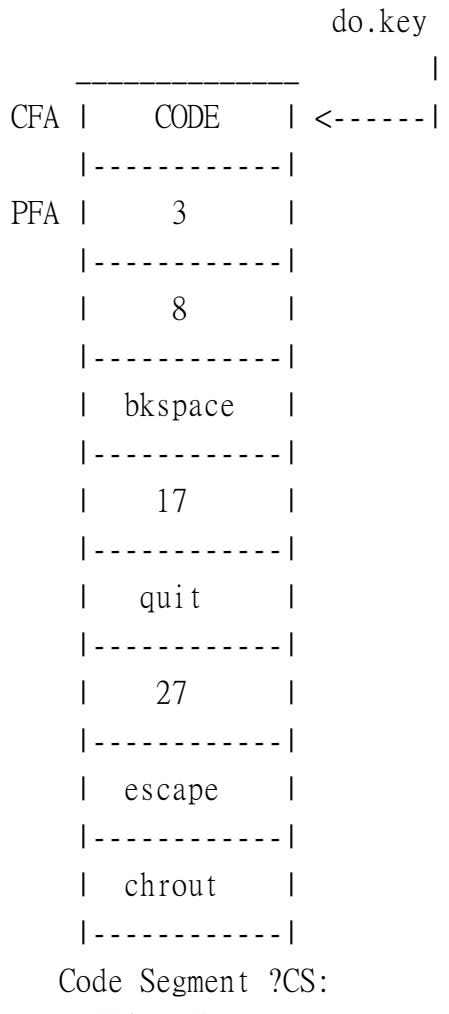
THEN ;

在這定義中 PERFORM 指令將執行一個詞，該詞的 CFA 儲存在堆疊(stack)頂端所指到的地址中。在 CREATE 之後 DO--loop 使用到 ' , (tick comma)它們是被用來將 JUMP TABLE do.key 後面的詞群們的 CFA 推入 jump table 中。

8.3 以堆疊上數值爲索引之跳躍表格

在前一章裡所描述的 jump table 的限制，是進入 table 的索引必須是從 0 開始的連續整數，而我們通常所能得知的數字是一個與所壓下之鍵相對應的 ASCII 碼。

一個更普遍的 jump table 其每一個元素項包含一個鍵值(例如：一個 ASCII 碼)加上一個 CFA 值，如下所顯示的 table



這個表格可能被應用來做一個 EDIT(編輯程式)，那裡的 ASCII 碼 8 會使 FORTH 詞"bkspace"被執行，ASCII 碼 17(control-Q)會使"quit"被執行，ASCII 碼 27 會使 "escope" 被執行。當在跳躍表格中找不到相對應的詞群時，則預設的字 chrout 將會被執行，這個字會將該字母顯示在螢幕上，存在 PFA 位置中的 3 是 ASCII 和 CFA 的組數，欲建立此表格你可使用定義詞 MAKE.TABLE 如下述：

```
MAKE.TABLE do.key
         8 bkspace
        17 quit
        27 escape
        -1 chrout
```

有此功能之 MAKE.TABLE 之定義如下:

```
: MAKE.TABLE            ( -- )
        CREATE
           HERE 0 , 0                   \ pfa 0
           BEGIN
              BL WORD NUMBER DROP       \ pfa 0 n
              DUP 1+                    \ pfa 0 n n+1
           WHILE                        \ pfa 0 n
              , ' ,                     \ pfa 0
              1+                        \ pfa cnt
           REPEAT
           DROP ' ,                     \ pfa cnt
           SWAP !
        DOES>           ( n pfa -- )
           DUP 2+                       \ n pfa pfa+2
           SWAP @                       \ n pfa+2 cnt
           0 DO                         \ n code.addr
              2DUP @ =                  \ n addr (n=code)
              IF                        \ n addr
                 NIP 2+ LEAVE           \ -> CFA
              THEN
              4 +                       \ n addr
           LOOP
           PERFORM ;        ( 注意: 預設詞於堆疊上留下 n )
```

注意 -1 是用來辨認預設的詞，在 WHILE 敘述之前的 DUP 和 1+ 會使這個 -1 變成 0，也就是說預設的詞已被找到並同時離開 BEGIN---WHILE---REPEAT 迴路。

當 "do.key" 被執行時有一先前的 ASCII 碼被留在堆疊(stack)上，超過上述定義的 DOES> 部份，將會執行一個能配對的 ASCII 碼的 CFA 值否則執行預設值。

注意：假如預設的詞被執行，ASCII 碼仍將被留在堆疊(stack)上，所以它能夠被顯示在螢幕上。

8.4 使用 FORTH 詞之跳躍表格

使用在前一章裡定義詞 MAKE.TABLE 的缺點是當建立表格時須先知道 ASCII 碼的數值。在 FORTH 中 ASCII 和 CONTROL 這兩個指令可以很方便的被用來找出這些 ASCII 碼。例如:

```
ASCII A
```

將傳回數字 65(hex41)至堆疊上，相同的

```
CONTROL Q
```

將傳回數字 17(hex 11)至堆疊(stack)上。當建立跳躍表個格時能加入小括號註解將是一件較理想的方式，這是使用 MAKE.TABLE 無法辦到的。

我們將定義一個新的定義詞叫 EXEC.TABLE 將會允許我們建立如上述所題示的跳躍表格其擁有之理想的功能：

```
EXEC.TABLE do.key
        CONTROL H  |  bkspace   ( 退位鍵)
        CONTROL Q  |  quit       ( 離開 DOS )
        HEX 2B     |  escape    DECIMAL
          DEFAULT  |  chrout
```

EXEC.TABLE 的定義如下：

```
: EXEC.TABLE            ( -- )
        CREATE
           HERE 0 ,                    \ pfa
        DOES>           ( n pfa -- )
           DUP 2+                       \ n pfa pfa+2
           SWAP @                       \ n pfa+2 cnt
           0 DO                         \ n code.addr
              2DUP @ =                  \ n addr (n=code)
              IF                        \ n addr
                 NIP 2+ LEAVE           \ -> CFA
              THEN
              4 +                       \ n addr
           LOOP
           PERFORM ;        ( 注意: 預設詞留下 n 在堆疊上)
```

注意：此定義的 DOES> 部份與在 MAKE.TABLES 的定義是相同的，在 CREATE 部份較簡單，它在被定義的詞 do.key 的 PFA 中簡單的在計數欄位裡存入一個 0 ，並將此 PFA 的值留在堆疊(stack)上，這個程式返回了 FORTH 的交談模式中並繼續執行 FORTH 的詞 CONTROL H 這將會將 8 的值留在堆疊(stack)上，在此時堆疊(stack)上的值爲 PFA 和 8。

垂直的直線"|"是一個 FORTH 的詞定義如下:

```
      : |     ( addr n -- addr )
              , ' ,              \ 儲存 N 和 CFA 至表格
              1 OVER +! ;        \ 在 PFA 計數區欄位加 1
```

注意第一行 , ' ,(comma-tick-comma)會將 ASCII 碼的值 n 編入(comma)先前建立的表格中，且 tick (')會取到緊接在 " | " 後的 FORTH 詞之 CFA 並亦將其編入(comma)表格中。在同一行中其它 FORTH 詞如" ("或 DECTMAL| 將會被執行。
DECTMAL| 定義如下：

```
     : DEFAULT|      ( addr -- )
                     DROP ' , ;
```

它只是將堆疊(stack)中的 PFA 拋掉取得預設詞的 CFA 並將其編入跳躍表格

中。

8.5 彈出式功能表

此節將會使用定義詞 XEC.TABLE 定義在功能表中各種不同鍵被壓下時的對應動作。在這一段中所定義的詞，可被用來設計一個理想的功能表驅動式的應用程式。

下列 ASCII 碼是經常被用到的

```
200     CONSTANT 'up
208     CONSTANT 'down
203     CONSTANT 'left
205     CONSTANT 'right
199     CONSTANT 'home
207     CONSTANT 'end
201     CONSTANT 'pg.up
209     CONSTANT 'pg.dn
210     CONSTANT 'ins
211     CONSTANT 'del
8       CONSTANT 'bksp
9       CONSTANT 'tab
13      CONSTANT 'enter
27      CONSTANT 'esc
187     CONSTANT 'f1
188     CONSTANT 'f2
189     CONSTANT 'f3
190     CONSTANT 'f4
191     CONSTANT 'f5
192     CONSTANT 'f6
193     CONSTANT 'f7
194     CONSTANT 'f8
195     CONSTANT 'f9
196     CONSTANT 'f10
```

.本文

下列常用的變數在每個功能表中都會被使用到 ：

```
VARIABLE row_start              \ 功能表中第一列及第一行
```

```
VARIABLE col_start          \ 所在位置
VARIABLE row_select         \ 選擇項的列數
VARIABLE no_items           \ 功能表的項數

PREFIX
```

讀入目前游標位置的字元和屬性

```
CODE    ?char/attr      ( -- attr char )
        MOV     BH, # 0
        MOV     AH, # 8
        INT     16          \ 讀入字元/屬性
        MOV     BL, AH
        MOV     BH, # 0
        AND     AH, # 0
        PUSH    BX
        PUSH    AX
        NEXT
        END-CODE
```

在現在游標的位置寫入字元(character)和屬性(attribute)

```
CODE    .char/attr      ( attr char -- )
        POP     AX
        POP     BX
        MOV     AH, # 9
        MOV     CX, # 1
        MOV     BH, # 0
        INT     16          \ 寫入字元/和屬性
        NEXT
        END-CODE
```

顯示 n 字元/屬性對

```
CODE    .n.chars        ( n attr char -- )
        POP     AX
        POP     BX
        POP     CX
```

```
        MOV     AH, # 9
        MOV     BH, # 0
        INT     16              \ 寫入 n 個字元
        NEXT
        END-CODE
```

取得目前的顯示模式

```
CODE    get.vmode       ( -- n )
        MOV     AH, # 15
        INT     16              \ 現在螢幕狀態
        MOV     AH, # 0
        PUSH    AX
        NEXT
        END-CODE

: UNUSED ;
```

游標加 1

```
: inc.curs      ( -- )
                IBM-AT? SWAP 1+ SWAP AT ;
```

以反白屬性畫出一個字元

```
: .char.bar     ( attr char -- )
                SWAP DUP 2/ 2/ 2/ 2/ 7 AND  \ 交換前景
                SWAP 7 AND 8* 2* OR         \ 和背景
                SWAP .char/attr ;

: togatt        ( -- )
                ?char/attr              \ 逆反目前游標字
                .char.bar ;             \ 元的屬性
: invatt        ( -- )                  \ 詞的屬性切換
                BEGIN
                        ?char/attr DUP 32 = NOT
                WHILE   .char.bar inc.curs
                REPEAT 2DROP ;
```

得到現在影像的樣式，以相對的象徵標出記號

```
: invline       ( -- )                  \ 行的屬性切換
                BEGIN
                        invatt          \ 詞的屬性切換
                        togatt          \ 空白的屬性切換
                        inc.curs
                        ?char/attr      \ 直到 2 個空白
                        NIP
                        32 =
                UNTIL ;

: movcur        ( -- )  \ 移動游標至選擇列    \ 兩個空格
                col_start @ row_select @
                2* row_start @ + AT ;

: inv.first.chars       ( -- )
                no_items @ 0 DO
                     I row_select !
                     movcur togatt
                LOOP ;

: select.first.item     ( -- )
                0 row_select !
                movcur invline ;

: inv.field     ( n -- )
                movcur                  \ 切換目前的行
                invline
                row_select !            \ 切換第 n 行
                movcur
                invline ;
```

上、下游標的鍵，將會改變選擇項

```
: down.curs     ( -- )
                movcur
                invline
                row_select @ 1+ DUP no_items @ =
```

```
                IF
                   DROP 0
                THEN
                row_select !
                movcur
                invline ;

: up.curs       ( -- )
                movcur
                invline
                row_select @ 1- DUP 0<
                IF
                   DROP no_items @ 1-
                THEN
                row_select !
                movcur
                invline ;
```

每一個定義的功能表，都有以下的數值，儲存在它的參數區中以下常數，是進入參數區的開始

| 左上角的行位 | 左上角的列位 | 寬度 | 功能項數 |

下列的常數是參數區的位移(offsets):

```
0    CONSTANT  [upper.left.col]
2    CONSTANT  [upper.left.row]
4    CONSTANT  [width]
6    CONSTANT  [no.items]
```

建立一個確定大小的功能表，你可鍵入 :

```
      { 25 [upper.left.col]
        15 [upper.left.row]
        20 [width]
         3 [no.items] }
      define.menu menu1
```

定義詞"define.menu"定義如下：

```
: define.menu           ( list n -- )
        CREATE
           HERE 8 ALLOT SWAP  \ list pfa n
           2/ 0 DO            \ v1 ix1 v2 ix2 v3 ix3 pfa
              SWAP OVER +     \ v1 ix1 v2 ix2 v3 pfa addr
              ROT SWAP !      \ v1 ix1 v2 ix2 pfa
           LOOP
           DROP
        DOES>           ( pfa -- pfa )
           DUP [upper.left.col] + @ 1+ col_start !
           DUP [upper.left.row] + @ 1+ row_start !
           DUP [no.items] + @ no_items ! ;
```

注意：上述可被用來定義 menu1 這個詞，並將值 25 15 20 和 3 編入參數區以規範其位置大小等等。回憶第七課中的括弧{.....}它會將大括號中的數值留在堆疊(stack)上，爲要使用" { "和" } "在你載入第八課之前要先載入第七課。

當字 "menu1" 被執行時其參數區中的數值將被用來儲存功能表的 col_start ，row_start 和 no_items 等給這特定的功能表。

這個字是用來準備堆疊(stack)上的參數給 F-PC 詞 BOX&FILL 用。請自行參照 BOXTEXT.SEQ 檔，以了解 BOX&FILL 的定義與描述。

```
: ul.br  ( pfa -- ul.col ul.row br.col br.row )
     DUP [upper.left.col] + @         \ pfa ul.col
     OVER [upper.left.row] + @        \ pfa ul.col ul.row
     2 PICK [width] + @ 1- 2 PICK +   \ pfa ul.col ul.row br.col
     3 ROLL [no.items] + @ 2* 2 PICK + ;
```

定義主功能選擇表

```
        { 25 [upper.left.col]
          8 [upper.left.row]
         20 [width]
```

```
              3 [no.items] }
            define.menu main.menu


     第一功能表  ------------------------

              { 30 [upper.left.col]
                10 [upper.left.row]
                20 [width]
                 2 [no.items] }
              define.menu first.menu

: first.menu.display    ( -- )
                0 inv.field             \ 轉換第一項
                SAVESCR                 \ 儲存螢幕
                first.menu              \ 得到新座標
                ul.br BOX&FILL          \ 畫出外框
                ." First subl item"
                bcr bcr ." Second subl item"
                inv.first.chars
                select.first.item ;

: first.subl ;

: second.subl ;

: escape.first          ( -- )
                RESTSCR
                main.menu DROP
                0 row_select !
                2R> 2DROP
                2R> 2DROP
                EXIT ;

: enttbl.first          ( n -- )
                EXEC:
                first.subl
                second.subl ;
```

```
: enter.first           ( -- )
               row_select @ enttbl.first ;

EXEC.TABLE do.key.first
        'up       |  up.curs
        'down     |  down.curs
        ASCII F   |  first.sub1
        ASCII f   |  first.sub1
        ASCII S   |  second.sub1
        ASCII s   |  second.sub1
        'esc      |  escape.first
        CONTROL M |  enter.first      \ 輸入一個鍵--選擇項
           DEFAULT|  UNUSED

: first.item         ( -- )
               first.menu.display
               BEGIN
                  KEY do.key.first
               AGAIN ;
```

第二功能表 ----------

```
             { 30 [upper.left.col]
               12 [upper.left.row]
               20 [width]
                2 [no.items] }
             define.menu second.menu

: second.menu.display    ( -- )
               1 inv.field             \ 轉換第二項
               SAVESCR                 \ 儲存螢幕
               second.menu             \ 得到新座標
               ul.br BOX&FILL          \ 畫出外框
               ." First sub2 item"
               bcr bcr ." Second sub2 item"
               inv.first.chars
               select.first.item ;
```

```
: first.sub2 ;
: second.sub2 ;

: escape.second          ( -- )
                RESTSCR
                main.menu
                1 row_select !
                2R> 2DROP
                2R> 2DROP
                EXIT ;

: enttbl.second          ( n -- )
                EXEC:
                first.sub2
                second.sub2 ;

: enter.second           ( -- )
                row_select @ enttbl.second ;

EXEC.TABLE do.key.second
        'up       |  up.curs
        'down     |  down.curs
        ASCII F   |  first.sub2
        ASCII f   |  first.sub2
        ASCII S   |  second.sub2
        ASCII s   |  second.sub2
        'esc      |  escape.second
        CONTROL M |  enter.second       \ 輸入一個鍵--選擇項
           DEFAULT|  UNUSED

: second.item          ( -- )
                second.menu.display
                BEGIN
                   KEY do.key.second
                AGAIN ;

    主功能選擇表 --------------------

: main.menu.display    ( -- )
```

```
                DARK
                main.menu               \ 得到新座標
                ul.br BOX&FILL          \ 畫出外框
                ." First item"
                bcr bcr ." Second item"
                bcr bcr ." Quit"
                inv.first.chars
                select.first.item
                CURSOR-OFF ;

: quit.main         ( -- )
                CURSOR-ON DARK ABORT ;

: enttbl.main         ( n -- )
                EXEC:
                first.item
                second.item
                quit.main ;
: enter.main          ( -- )
                row_select @ enttbl.main ;

EXEC.TABLE do.key.main
        'up       |  up.curs
        'down     |  down.curs
        ASCII F   |  first.item
        ASCII f   |  first.item
        ASCII S   |  second.item
        ASCII s   |  second.item
        ASCII Q   |  quit.main
        ASCII q   |  quit.main
        CONTROL M |  enter.main       \ 輸入一個鍵--選擇項
           DEFAULT|  UNUSED

: main        ( -- )
                main.menu.display
                BEGIN
                   KEY do.key.main
                AGAIN ;
```

8-6 練習

8.1 定義一個定義詞 BASED ，可以用來建立不同基底的數字輸出指令
舉例：

```
16 BASED. HEX.
```

將定義 HEX.成爲一個 FORTH 指令其可以用來以 16 進位方式(HEX
印出堆疊頂端的值，但不會永久地改變 BASE(基底) 變數的內容

```
DECIMAL
17 DUP HEX. .
```

將印出

```
11 17 ok
```

8.2 使用向量執行法(就是跳躍表格)來寫作一個 FORTH 的程式，此程
式在按下列特定的字鍵時將會印出對應的訊息

Key pressed	Message
F	Forth is fun!
C	Computers can compute
J	Jump tables
N	<your name>

按下任何其它鍵，將產生一個響聲(CONTROL G EMIT)

第九課　編　譯　詞

9.1　編譯(COMPILING) 和直譯(INTERPRETING)

編譯詞是立即詞(IMMEDIATE)，此意思是它們會被立即執行即使他們在高階程序中相遇或在編譯部份程式時。立即詞，在名稱區裡的優先位元(precedence bit)會被設定。(見第三課 3.12)

F-PC 能存在編譯或直譯兩種可能的狀態之中，編譯的狀態是在高階程序的編譯過程中存在也就是說在 "："(冒號)執行後和 "；"(分號)執行之前。

系統變數 STATE 包含以下兩個可能的值：

```
TRUE  --  編譯狀態(compiling)
FALSE --  直譯狀態(interpreting)
```

在不同的時間下的狀態測試，考慮下列兩個高階程序：

```
: lstate?       ( -- )
                STATE @
                IF
                   ." Compiling"
                ELSE
                   ." Interpreting"
                THEN
                CR ;

: ltest         ( -- )
                lstate? ;
```

以 FLOAD 指令載入第九課後鍵入

```
lstate?
```

和

```
ltest
```

在每一個 CASE 裡" 直譯 "會印出來爲什麼 ？因爲當你打入 lstate? 和

1state 這兩者時你是在直譯狀態下

如何能印出"編譯"(compiling)這個字?當編譯 1test 的同時，執行 1state? 是必須的，也就是我們必需使 1state? 成爲一個立即詞，我們藉著在分號之後加進 IMMEDIATE 來完成這樣的工作，讓我們改變名稱到 2state? 然後定義

```
: 2state?       ( -- )
                STATE @
                IF
                   ." Compiling "
                ELSE
                   ." Interpreting "
                THEN
                CR ; IMMEDIATE
```

現在鍵入以下的高階程序

```
: 2test   2state? ;
```

注意當你鍵入此高階程序，按下<Enter> "Compiling"這個字，就馬上印出來，也就是 2state? 被立即執行不等你稍後鍵入 2test 現在鍵入

```
2test
```

注意在螢幕沒有印出東西，是因 2state? 沒有被編譯到字典之中，而被立即執行，立即詞在詞典裡沒有編譯除非強迫去做。你可以強迫編譯一個立即詞，而非立即執行，可藉著在前面放置 [COMPILE] 來完成這個工作。然後，如下列 3test 高階程序中的字 2state? 就被編譯了且不會被立刻執行

```
: 3test         ( -- )
                [COMPILE] 2state? ;
```

你認爲當你鍵入 3test? 時會印出什麼來? 試試看。

藉著使用 " [" 和 "] " 這兩個指令，可以在高階程序中，離開或回到編譯器·

"[" 是一個離開編譯狀態的立即詞，也就是它回到直譯狀態。"[" 的定義是

```
: [     ( -- )
```

```
        STATE OFF ; IMMEDIATE
```

"]" 詞可進入編譯狀態，然後進入編譯迴路。編譯迴路之組成如下：

重覆做下列事，在輸入字串中(input stream) 抓入下一個詞，假如是一個立即詞，執行它否則編譯它。假如此字不在詞典裡，將它轉變成爲一個數字，並編入字典

繼續做直到輸入字串(input stream)結束爲止。

最後舉例，鍵入下列之例子

```
: 4test  [ 1state? ] ;
```

注意當按下 <Enter> 會印出 "interpreting "

9.2 一般詞編譯和立即詞編譯

我們已經了解 [compile] 會編譯跟隨其後的立即詞進入執行串列段，它的定義是：

```
: [COMPILE]     ( -- )
                ' X, ; IMMEDIATE
```

"tick" (')此字會將下一個立即詞(immediate)的 CFA 放在堆疊(stack)和" X "上，此字編譯堆疊(stack)上的整數進入執行串列段中的下一個可以使用的位置，注意 [compile] 本身是一個立即詞，它會立即被執行，在包含它的高階程序被編譯之時。

有時你在執行時間想要編譯一個字。Compile 此字會完成這一工作，例如分號定義是根據以下來的·

```
: ;     ( -- )
        COMPILE UNNEST          \ 編譯 UNNEST 程序
        REVEAL                  \ 使此高階程序可以使用
        [COMPILE] [             \ 進入直譯模式
        ; IMMEDIATE             \ 立即執行" ; "
```

注意 "；" 是一個立即詞當它在高階程序出現時會被立即執行它會將 UNNEST 程序的 CFA 編入執行串列字典中。
藉著 "REVEAL" 使冒號定義字(高階程序)可在字典中被找到，藉著在 "；" 定義中已由 "[COMPILE]" 將之編譯之 "[" 的執行切換到直譯狀態下。("[" 是一立即詞)
COMPILE 之定義如下，當有包含 "compile" 之字被執行時，它會將其後非立即詞之字的 CFA 加以編譯

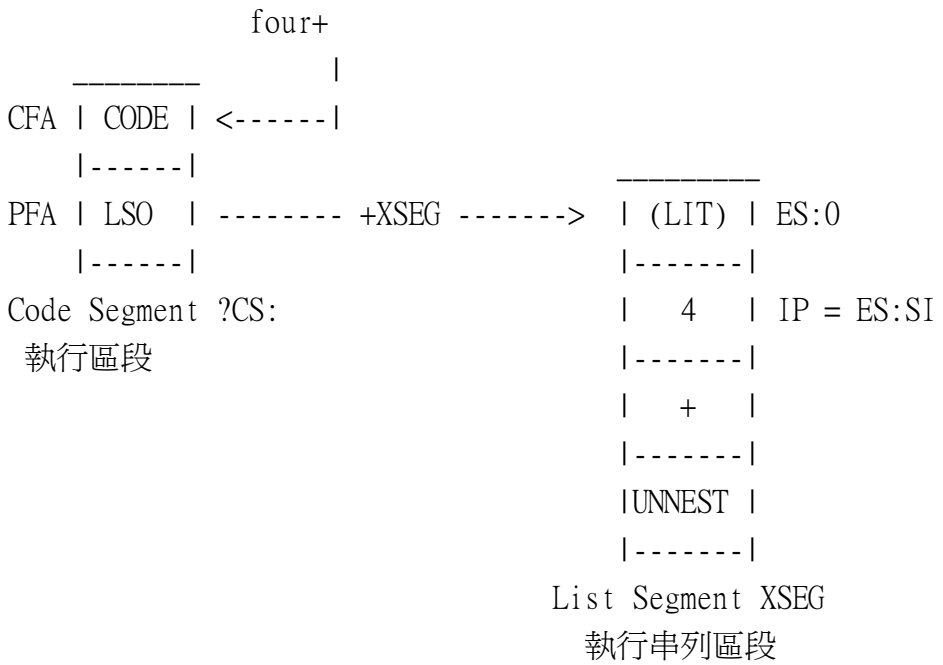
```
: COMPILE       ( -- )
      2R@           \ 取得在執行串列段中的下一個 CFA 也就是 ES:SI
      R> 2+ >R      \ 使 SI 之值加 2 以跳過下一個詞
      @L            \ 在堆疊(stack)中取得下一個字的 CFA 並在執行
      , X ;         \ 時將其編入執行串列段中
```

9.3 載入數字

思考下面的高階程序

```
      : four+         ( n -- n+4 )
                      4 + ;
```

如下圖被編譯入詞典中



(LIT)是一個低階程序，其定義如下：

```
CODE (LIT)   ( -- n )
        LODSW ES:    \ 取得由 ES:SI 指到的下一個詞並將 SI=SI+2
        1PUSH        \ 將其推入堆疊(stack)
        END-CODE
```

因此(LIT)指令會將 4 推入堆疊(stack)並將指令指標 ES:SI 指到" + " 的 CFA

如果你要將堆疊(stack) 上的一個數字，編入執行串列的詞典中，你可以使用 LITERAL 這個指令，此字之定義如下：

```
: LITERAL       ( n -- )
                COMPILE (LIT)    \ 編譯 (LIT)
                X,               \ 接著編入 n 的值
                ; IMMEDIATE      \ 立即執行
```

LITERAL 這個指令常被運用的時機爲當你在定義高階程序的過程中，計算某些常數的值，例如，假設 2+3 可以得到 5 (在某些高階程序中你會更清楚)，你可以定義高階程序 five+ 這字，如下：

```
: five+         ( n -- n+5 )
                [ 3 2 + ] LITERAL + ;
```

雖然下列定義亦會產生相同結果

```
: five+         3 2 + + ;
```

其優點是 [3 2 +] LITERAL 在編譯時會在執行串列詞典裡編譯 5 這個數字以致在執行時僅執行 5 + ，另一方面 3 2 + + 會在執行串列詞典裡編譯成 2 和 3 這兩個數字而且在執行時會執行兩次加的動作。所以 [3 2 +] LITERAL 之使用，會產生執行較快且較節省記憶體的執行碼。

金城 註疏：

" [" 和 "] " 用來在編譯時間做計算工作，此種功能在電腦科學的領域裡被列爲編譯器編碼最佳化的數種技巧之一。

9.4 條件編譯詞

兩個有條件的編譯詞 BRANCH 和 ?BRANCH 是用來定義在 F-PC 裡不同的條件的分支結構，BRANCH 是一個低階程序(code word)，定義如下：

```
                                        |-------|
CODE BRANCH     ( -- )                  |BRANCH |
        LABEL BRAN1                     |-------|
              MOV ES: SI, 0[SI]     |---| addr  | IP = ES:SI
              NEXT                  |   |-------|
              END-CODE              |   |       |
                                    |   |-------|
                                    |   |       |
                                    |   |-------|
                                    |-->|       | ES:addr
                                        |-------|
                                        |       |
                                        |-------|
                                   List Segment XSEG
                                    執行串列區段
```

BRANCH 被編譯進執行串列詞典裡緊跟著非條件分支的位移地址(offset address)。

假如堆疊(stack)頂端的旗號是 FALSE(錯誤)的話， ?BRANCH 會產生分歧跳入在 ?BRANCH 後之地址，其定義如下：

```
                                        |-------|
CODE ?BRANCH     ( f -- )               |?BRANCH|
                 POP AX                 |-------|
                 OR  AX, AX         |---| addr  | IP = ES:SI
                 JE  BRAN1          |   |-------|
```

```
   ADD SI, # 2          |    |       |
   NEXT                 |    |-------|
   END-CODE             |    |       |
                        |    |-------|
                        |--->|       | ES:addr
                             |-------|
                             |       |
                             |-------|
                         List Segment XSEG
                          執行串列區段
```

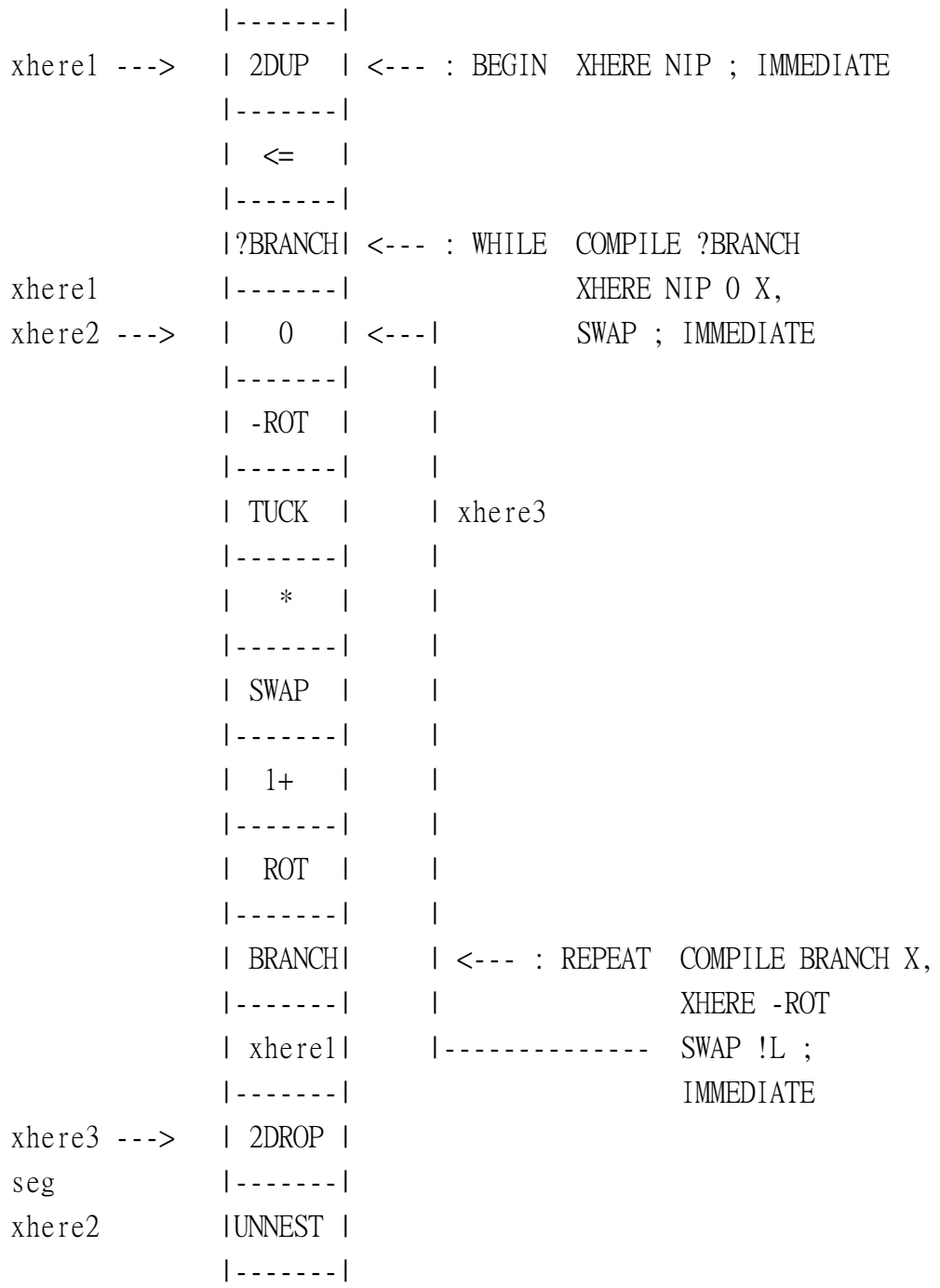
BEGIN-----WHILE-----REPEAT

舉一個 BEGIN----WHILE----REPEAT 迴路的例子，回憶第四課"FACTORIAL"的定義：

```
: factorial     ( n -- n! )
                1 2 ROT                 \ x i n
                BEGIN                   \ x i n
                   2DUP <=              \ x i n f
                WHILE                   \ x i n
                   -ROT TUCK            \ n i x i
                   * SWAP               \ n x i
                   1+ ROT               \ x i n
                REPEAT                  \ x i n
                2DROP ;                 \ x
```

此定義會如下儲存在執行串列詞典裡

```
                _________
               | (LIT) |
               |-------|
               |   1   |        XHERE  ( -- seg offset )
               |-------|
               | (LIT) |
               |-------|
               |   2   |
  STACK        |-------|
               |  ROT  |
```



List Segment XSEG
執行串列區段

BEGIN 把位移地址(offset address) xhere1 留在堆疊(stack)上，WHILE 詞將會編譯 ?BRANCH 爲 0 緊跟在地址爲 xhere2 的後面，0 的值稍後會被 2DROP 的地址 xhere3 取代 WHILE 也把 xhere2 的數字留在堆疊(stack) xhere1 之下。

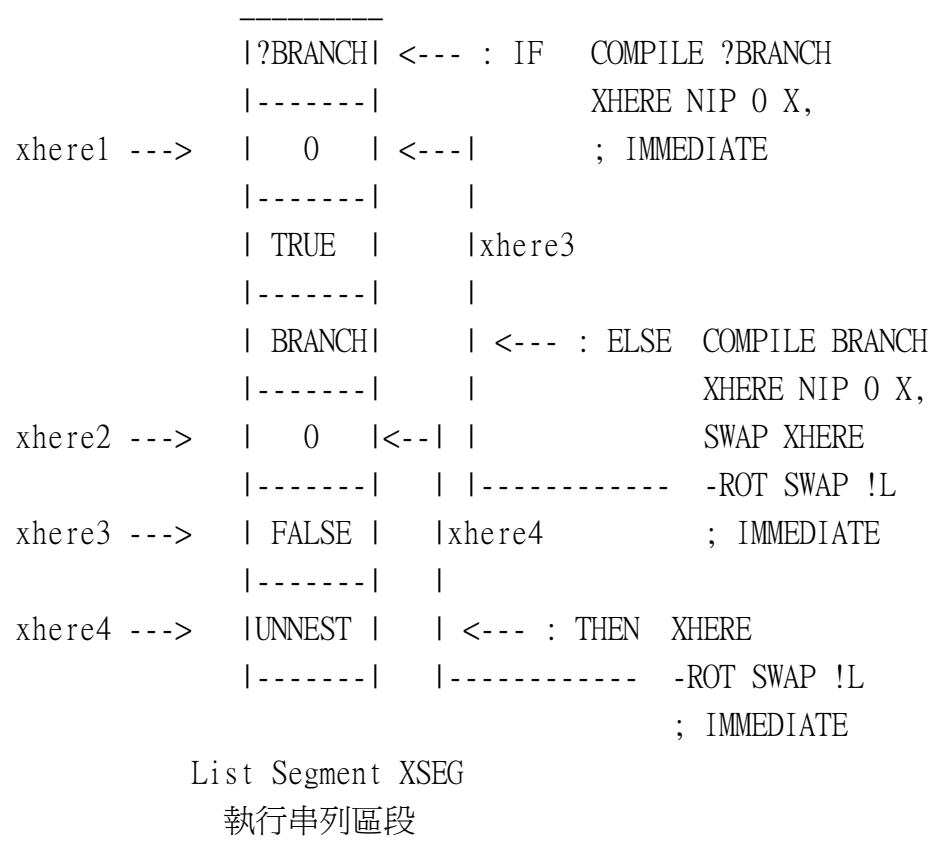
REPEAT 編譯 BRANCH 然後把地址 xhere1 編入，進一步把地址 xhere3 放在堆疊(stack)上然後儲存在地址 seg:xhere2 中

IF------ELSE------THEN

考慮下面的高階程序

```
: test          ( f -- f )
                IF
                   TRUE
                ELSE
                   FALSE
                THEN ;
```

這會如下的儲存在執行串列的詞典中



List Segment XSEG
執行串列區段

IF 這個指令編譯 ?BRANCH 的 CFA 並跟著一個 0 在 xhere1 的地方。這一個 0 稍後會被 xhere3 的地址所指到 FALSE 的字的地址所取代，IF 同時也將 xhere1 的值留在堆疊(stack)上。

ELSE 這個指令編譯 BRANCH 並緊跟著一個 0 在 xhere2 的地方，這個 0 在稍後會被 xhere4 的地址所取代，該地址存放著 UNNEST 的 CFA ，ELSE 同時也將 xhere2 的地址留在堆疊(stack)中，之後並將 xhere3 的地址由堆疊(stack)中取出存入 seg:xhere1 的地址中。

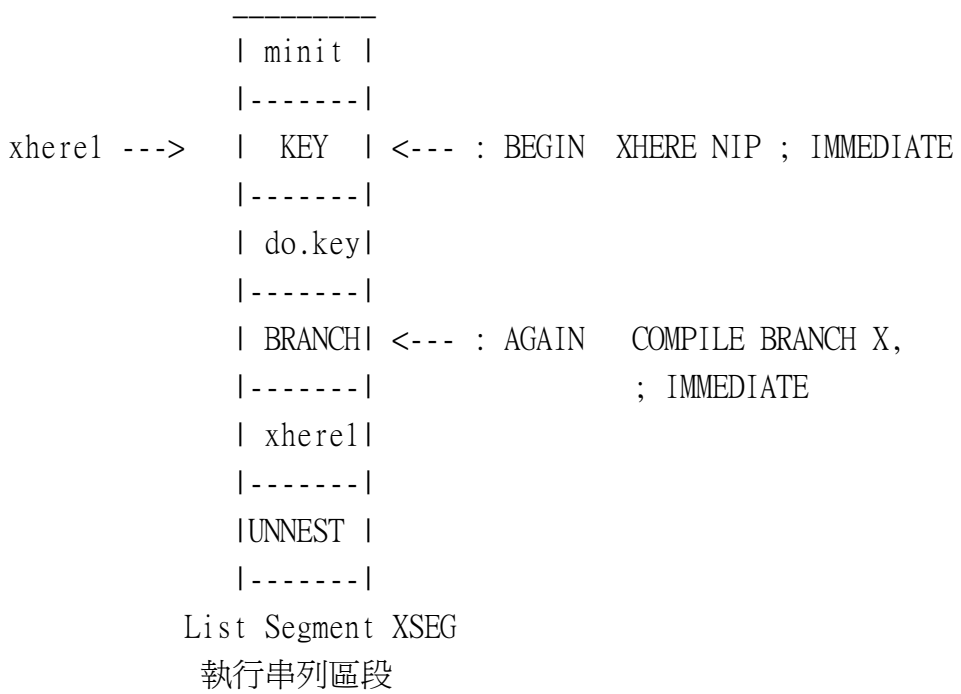
THEN 指令會將 xhere4 的地址存入 seg:xhere2 的地址中。

BEGIN------AGAIN

BEGIN-----AGAIN 使用的例子，已在第八課的 POP-UP 彈出式功能選擇表說明過了

```
: main          ( -- )
                minit
                BEGIN
                   KEY do.key
                AGAIN ;
```

會如下圖儲存進執行串列詞典中



BEGIN 指令把位移地址(offset address) xhere1 留在堆疊(stack)上。
AGAIN 指令編入 BRANCH ，然後將 xhere1 的地址編入執行串列字典中。

金城 註疏：

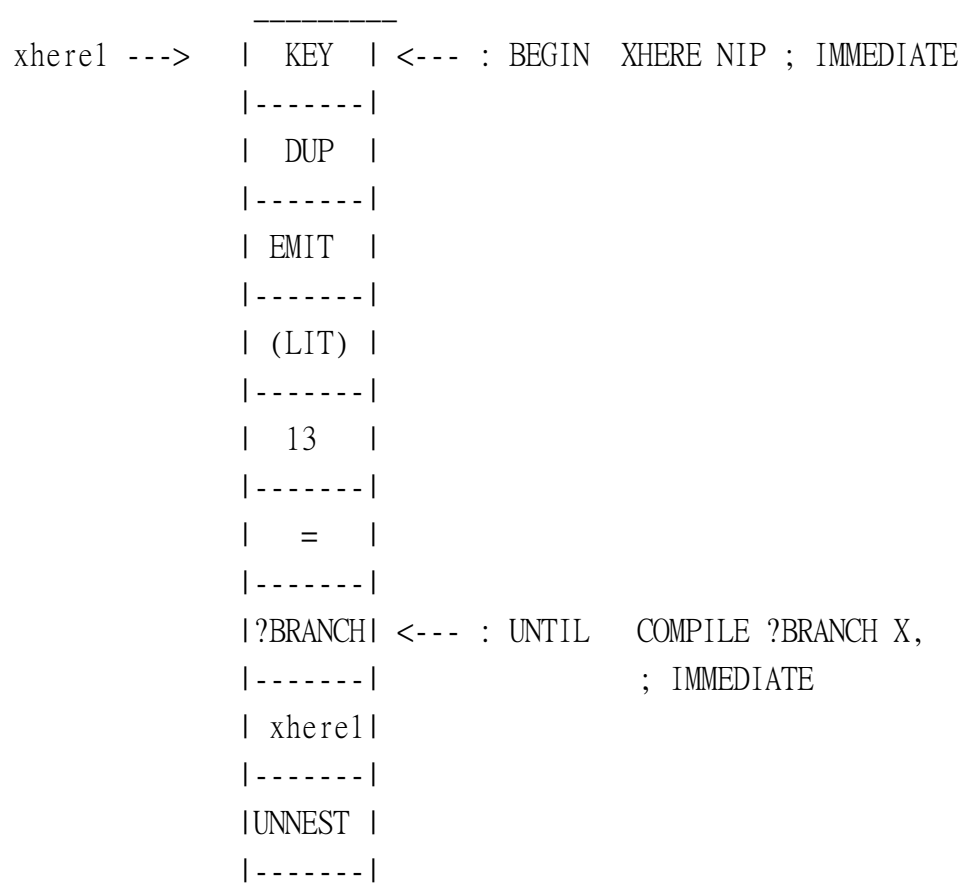
上面程序中 AGAIN 爲無條件跳躍至 BEGIN 而形成無限迴路所以在其後的 UNNEST 永遠不會被執行，如果將 UNNEST 抽掉可節省記憶體空間又不會產生錯誤，在此處的 UNNEST 唯一用途是使程式美觀而無其他用處，可以用 " [" 取代之。

BEGIN-----UNTIL

下面使用 BEGIN-----UNTIL 的例子在第四課已經說明過了。

```
: dowrite       ( -- )
                BEGIN
                   KEY
                   DUP EMIT
                   13 =
                UNTIL ;
```

這將會如下圖儲存在執行串列的詞典中



List Segment XSEG
執行串列區段

BEGIN 將留下目前位移地址(offset address) xhere1 在堆疊上(stack)。
UNTIL 指令將 ?BRANCH 編入並隨後將堆疊上(stack)上的 xherte1 編入。

注意在 BEGIN-----AGAIN 和 BEGIN-----UNTIL 之間唯一的不同是在 UNTIL 是用 ?BRANCH 取代 AGAIN 之 BRANCH 。

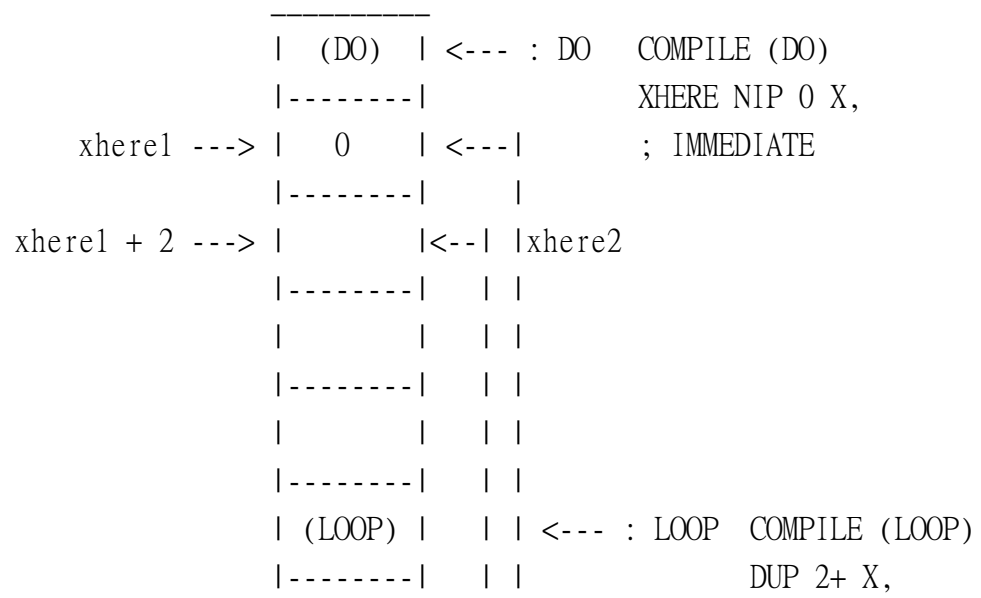
金城 註疏：

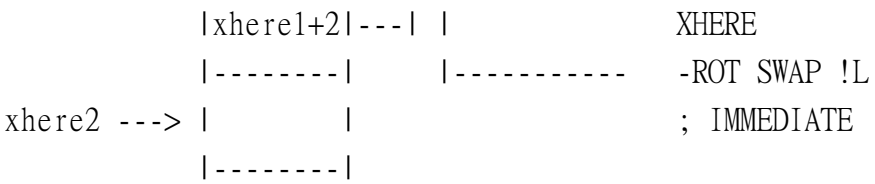
一、在條件迴路中 BEGIN-----UNTIL 只有一個跳躍動作，所以情況允許時少用 BEGIN-----WHILE-----REPEAT 而改用 BEGIN------UNTIL 因爲效率及記憶體之使用均會較有效率。

二、在 FORTH 程式設計師眼光中，對於語言機構的熟悉是對於程式作最佳化的知識基礎。

DO------LOOP

DO-----LOOP 將會在執行串列詞典中產生如下的結構。





List Segment XSEG
執行串列區段

DO 指令編譯 (DO) 緊接著一個 0 在 xhere1 的位址裡，這個 0 稍後將會被 DO-----LOOP 之後的下一個指令地址 xhere2 所取代，DO 同時也會將 xhere1 留在堆疊(stack)上。

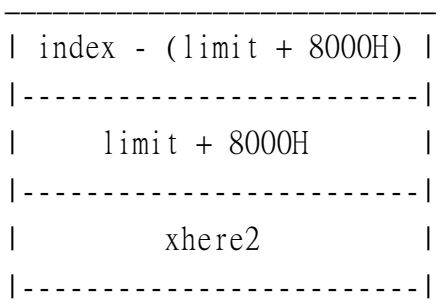
LOOP 這個指令會編譯 (LOOP) ，然後將堆疊(stack)的地址 xhere1 + 2 編入執行串列詞典中，LOOP 指令然後將 xhere2 的地址存入 seg:xhere1 的位址中。

執行詞：

(DO) (limit index --)

在回返堆疊(return stack)建立下列結構：

Return Stack(回返堆疊)



執行詞 (LOOP) 加 1 到回返堆疊(return stack)的頂端，且跳至 xhere1+2 ，假如滿溢旗標(overflow flag)沒有設立了 (例如，當 index = limit 時堆疊(stack)的頂端只超過 8000h 的界限)(loop)會將回返堆疊 (return stack)上三個值彈出丟棄並將指令指標 ES：SI 搬至指向 xhere2 的地址，也就是迴路後的第一個動作。

金城 註疏：

如果滿溢旗標(overflow flag)被設立的話，回返堆疊(return stack)的頂端當 index = limit 時會超過 8000h 。

在回返堆疊(return stack)上的第三個值 xhere2 是被用來給 LEAVE 指令找到離開迴路後的地址，在回返堆疊(return stack)上的兩個值均加上 8000H 是被用來允許 (DO) 工作時迴路的上限可大於 8000H ，例如，假設迴路上限是 FFFFH 起值是 0 ，則在回返堆疊(return stack)上面的值會變成 -7FFFH ，當這個值加 1 時滿溢旗標(overflow flag)不會被設定，直到堆疊(stack)的頂端變成等於 8000H 時，就是說在迴路執行 FFFFH 之後。

9.5 習　　題

9.1 使用 SEE 和 LDUMP 指令，由下列三個測試程序來研究詞典的結構

```
: a.test        ( f -- )
                IF
                   ." True"
                ELSE
                   ." False"
                THEN ;

: b.test        ( -- )
                5 0 DO
                   I .
                LOOP ;

: c.test        ( -- )
                4
                BEGIN
                   DUP .
                   1- DUP 0=
                UNTIL
                DROP ;
```

幫上面每一個程序，畫詞典的結構圖，並標示出在執行串列詞典中指出每一個地址名稱和真實儲存的值，指出在這些結構圖中，IF 、ELSE 、THEN 、DO 、LOOP 、BEGIN 和 UNTIL 的效用，也解釋在 a.test 裡 ” ." ” 是如何工作，數字 5 、0 和 4 是如何儲存在 b.test 和 c.test 中

金城 註疏：

一、在 FORTH 中 還有另一個迴路爲 <次數> FOR ----- NEXT 此爲最快的一種計次迴路，詳細結構請參考 F-PC 參考手冊 112 頁。

二、如果把編譯詞的結構弄清楚，便了解爲何人們常稱 FORTH 爲一軟體模擬 CPU 。

三、另所有的結構化的分歧與迴路均由 ?BRANCH 與 BRANCH 來構成，此爲學習組合語言基礎與捷徑。

第 十 課　FORTH 的資料結構

10.1 陣列(ARRAYS)

這一課許多的資料是根據"Object-oriented Forth" by Dick Pountain Academic Press 1987 爲題材，我們將延伸這些觀念以使資料結構可以利用系統裡所有的記憶體。

F-PC 詞 ALLOC (#para -- #para segment flag)
和 DEALLOC (segment -- flag) 使用 DOS 之系統功能 AH=48H 和 AH=49 來配置和釋放記憶體，我們可以定義以下更方便的字，來配置和釋放記憶體。

```
: alloc.mem       ( size -- segment )
              PARAGRAPH ALLOC       \ DOS alloc INT 21H - AH=48H
              8 =
              ABORT" Not enough memory to allocate "
              NIP ;                       \ 放棄配置記憶體

: release.mem     ( segment -- )
                DEALLOC                   \ DOS INT 21H - AH=49H
                ABORT" Failed to deallocate segment "
                ;
```

alloc.mem 這個詞需要在堆疊(stack)上使用，需區塊的位元組大小，並傳回配置區段的區段地址。

F-PC 的詞：

```
      : PARAGRAPH     15 + U/16 ;
```

將轉換所需求 bytes 的數成爲以 16-bytes 爲大小的段數 。release.mem 將會釋放被 alloc.mem 所配置的記憶體，首先你須先將欲釋放之區段的區段位址放入堆疊(stack)。(這個地址必須爲先前 alloc.mem 傳回之區段位址)

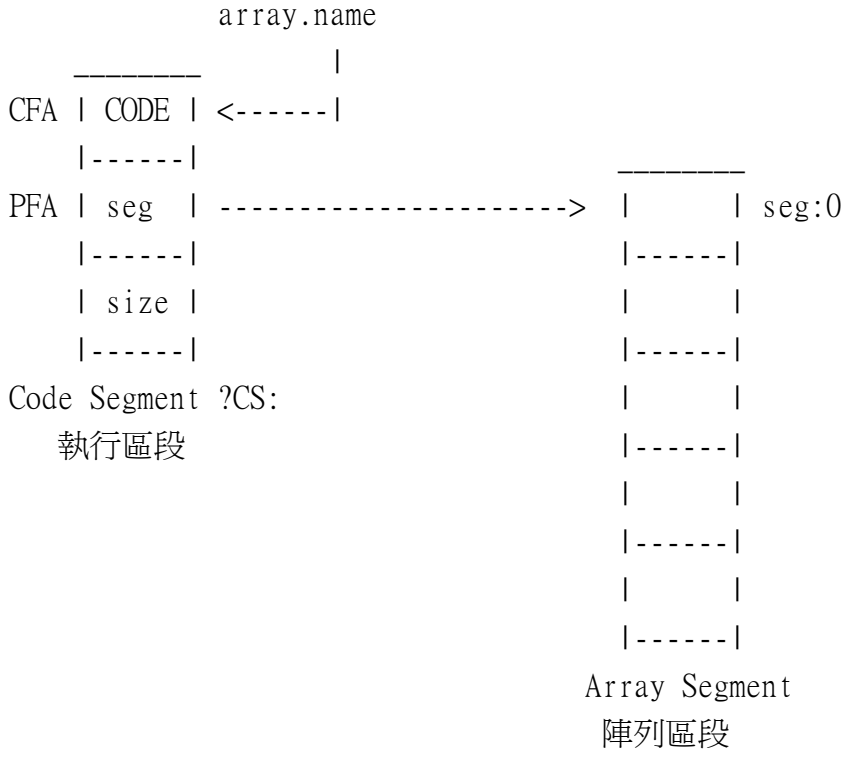
假設你要製造一個確定大小的矩陣在延伸記憶中，然後使用 @L 和 !L 來取出和存入在此陣列中的數值，我們必須定義下列之定義詞：

```
: array       ( size -- )
              CREATE
                 2* DUP alloc.mem ,    \ 存區段位址。
                 ,                     \ 存陣列大小(byteso)
              DOES>
                 @ ;
```

舉例：

```
          1000 array array.name
```

將在字典中建立一個叫做 array.name 的資料結構，將配置 16 位元字組 1000 的記憶空間，並將配置區塊的段落地址和矩陣的大小存入陣列 array.name 之參數區(parameter field) 中，當稍後 array.name 被呼叫時它會將此陣列的段落位址留在堆疊(stack)上。 array.name 在字典中的儲存方式如下:



例如，欲取得陣列元素 array.name(5) 之值，可建入

```
                  array.name 5 @L
```

使用這種在延伸記體憶陣列的技巧會導致一個問題，那就是如果你要做一個開機後立即執行的程式，將無法有效的執行，一個 turnkey system 將把字典中所有的頭部去除，並產生一個僅包含你的應用程式和 F-PC 字群的可執

行檔，當你在磁碟上存入此系統時，在執行區段你已定義好的陣列會被儲存起來，但在配置記憶體中的陣列將會失去，也就是說 turnkey 程式稍後在執行時，必需要再重新配置陣列所需的記憶體，並將其區段地址存在 array.name 的 PFA 中。

我們可以修定陣列的定義以便能在開機系統中使用，定義如下:

```
: array.tk      ( size -- )
                CREATE
                   0 ,       \ 虛填一個 0 以便稍後填入區段地址
                   2* ,      \ 儲存陣列位元組的大小
                DOES>
                   @ ;
```

注意，假使你現在鍵入

```
1000 array.tk array.name
```

你將建立一個 arrary.name 的字典結構，並存入 1000 的大小，此刻並未配置任何記憶體給此陣列，使用下列的字，在稍後配置記憶體給所有的陣列字群。

```
: alloc.array   ( cfa -- )
                >BODY DUP 2+ @    \ 取得大小以 bytes 爲單位
                alloc.mem         \ 配置記憶體
                SWAP ! ;          \ 儲存段位址到 PFA 中

: allocate.arrays       ( -- )
                [ ' array.name ] LITERAL alloc.array ;
```

每一個你曾在程式中定義的陣列都需要一行與 allocate.arrays 一樣的程式，來完成程式初始化的工作，這將會爲你的每一個陣列配置所需的記憶體，即使是 turnkey system 也會正常工作。

你可以使用下列字群釋放所有配置陣列的延伸記憶體。

```
: release.array         ( cfa -- )
                >BODY @                 \ 取得段位址並釋
                release.mem ;           \ 放記憶體
```

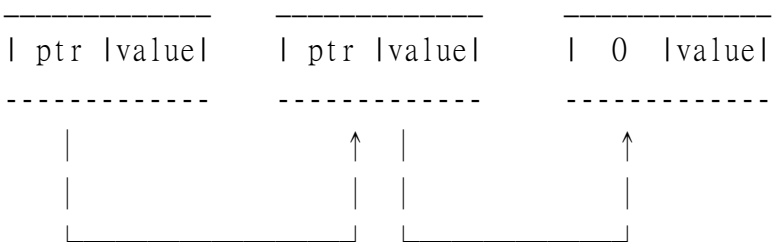
```
: release.all.arrays    ( -- )
                [ ' array.name ] LITERAL release.array ;
```

你可以爲每一個陣列加上一行與 release.all.arrays 類似的程式，來釋放他們的記憶體。

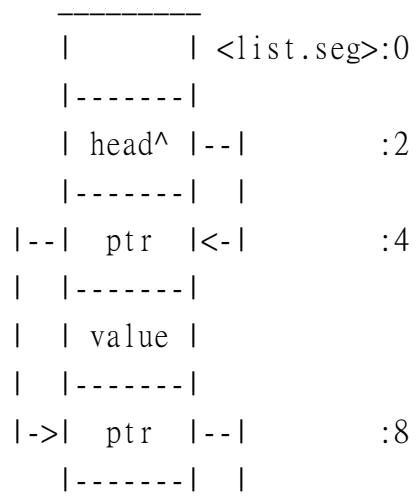
10.2 鏈串列(LINKED LISTS)

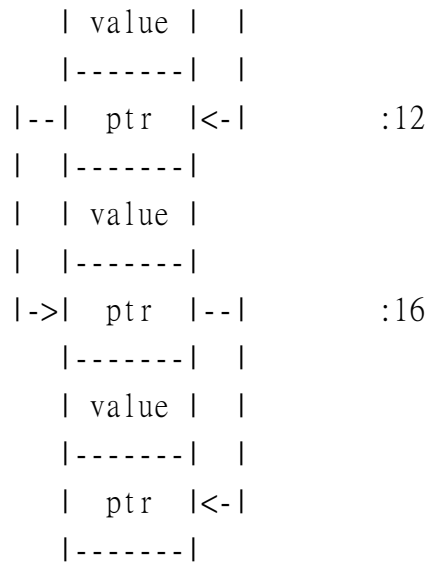
在此章節裡，我們要寫一群字用來建立和維護鏈串列(LINKED LISTS) (細節看 Pountain 書的第三章)。

在鏈串列的每一個節點包含 4bytes ，前兩個是指到下一個節點的指標值，後面兩個將包含 16-bit 之數值。



當我們要在鏈串列中增添一個新值時，我們需要從一個尚未使用的自由串列中取得一個新的節點，當我們要從鏈串列刪除一個值時，會將節點傳回自由串列中。我們需先配置一大塊的記憶體給自由串列使用，然後再聯結自由串列中所有的節點，如下：





第一個可供使用的節點在 <list.seg> 區段裡，位移地址爲 4 的地方開始，其後的每個節點佔據 4 個位元組，自由串列的 head 指標在位址 <list.seg>：2 中，在位址 <list.seg>：0 的值沒有用，只是用來保持以 4 爲基準的校定功能，以簡化未來的操作，下列字群將被用來建立這個自由串列。

變數和常數

```
DECIMAL
0    CONSTANT nil
2    CONSTANT [freelist.head]
0    VALUE    <list.seg>
[freelist.head]    VALUE    [list.offset]
```

配置記憶體

```
: release.seglist       ( -- )
                <list.seg> ?DUP
                IF
                    DEALLOC 0=          \ DOS INT 21H - AH=49H
                    IF
                       0 !> <list.seg>
                    ELSE
```

```
                    ABORT" Failed to deallocate <list.seg> "
                  THEN
               THEN ;

: alloc.seglist      ( size -- )
               release.seglist
               2* 2* 4 +            \ 每個節點 4 個位元組
               alloc.mem            \ 配置記憶體
               !> <list.seg> ;      \ <list.seg> = 基底段位址
```

建立自由串列

節點: | 指標 | 值 |

```
: allocate.freelist      ( size -- )
       DUP alloc.seglist               \ 大小
       [list.offset] 2+                \ 下一個指標地址存
       <list.seg> [list.offset] !L     \ 入現在的指標中
       2 +!> [list.offset]             \ 使下一個指標變成當前指標
       1 DO                            \ 做 size-1 次
           [list.offset] 4 +           \ 下一個指標地址存
           <list.seg> [list.offset] !L \ 入現在的指標中
           4 +!> [list.offset]         \ 使下一個指標變成當前
     LOOP                              \ 指標
       nil <list.seg> [list.offset] !L \使最最後一個指標爲終點
       4 +!> [list.offset] ;           \ 串列指標指到串列終點

: freelist        ( -- seg offset )
                  <list.seg> [freelist.head] ;
```

節點的維護字群

下列這個詞會在地址 seg:list 之後插入一個在 seg:node 地址的新節點指標。

```
: node.insert  ( seg list seg node --- )  \ 在 seg:list 之後插入
                2OVER @L                   \ s l s n @l
```

```
                ROT 2 PICK                  \ s l n @l s n
                !L                          \ s l n
                -ROT !L ;
```

下列的這個詞會將緊跟在 seg：list 之後的節點刪除，並將該刪除節點的地址 seg：node 留在堆疊(stack)上。假如 seg：list 是頭指標，這個字將會刪除在串列中的第一個節點，如果是空串列將會在堆疊(stack)上留下 seg：0

```
: node.remove   ( seg list -- seg node )
                2DUP @L                     \ s l @l
                2 PICK SWAP DUP             \ s l s @l @l
                IF                          \ s l s @l
                  2SWAP 2OVER @L            \ s @l s l @@l
                  -ROT !L                   \ s n
                ELSE                        \ s l s 0
                  2SWAP 2DROP               \ s 0
                THEN ;
```

取得一個你剛從自由串列中刪除的節點可使用 getnode 指令。

```
: getnode       ( --- seg node )
                freelist node.remove ;
```

釋還一個在 seg：node 的節點給自由串列可使用 freenode 指令。

```
: freenode      ( seg node --- )
                freelist 2SWAP              \ seg list seg node
                node.insert ;
```

金城 註疏：

在一個 FORTH 的高階定義中，若使用了一個堆疊(stack)操作指令，就如本頁的 node.remove 則必須在每一行後面加上堆疊(stack)現狀註解，否則連程式設計者本人也無法瞭解和除錯，這是 FORTH 程式設計師必須要養成的習慣。

newlist 詞將在執行區段(code segment)建立一個新的串列頭，這個串列頭的 PFA 保留給未來的串列的區段地址 <list.seg> 用。

```
: newlist       ( -- )
                CREATE
                        nil ,           \ 稍後填入節點地址
                DOES>       ( -- seg list )
                        <list.seg>  SWAP @ ;
```

建立一個新的串列稱爲 sample.list 你可鍵入

```
                  newlist sample.list
```

然後你將藉著在 fill.newlists 之後的這行指令，爲在串列區的地址 <list.seg> 中的串列建立串列頭。

```
: fill.newlists ( -- )
    getnode DUP [ ' sample.list ] LITERAL >BODY ! nil -ROT !L ;
```

這種技巧是用來產生及使在 turnkey system 中也可使用的鏈串列，與前述的陣列有許多相同之處，在你可以使用這些資料結構之前，你必須先在程式中使用下面這個字來配置記憶體。

```
      : init.data.structures  ( -- )
                      allocate.arrays
                      1200 allocate.freelist
                      fill.newlists ;
```

因此你可鍵入下面這個字來測試本章的教材

```
  init.data.structures
```

在你 FLOAD LESSON10 後

把 5 的值推入在串列 sample.list 的頂端鍵入：

```
              5 sample.list push
```

push 之定義如下：

```
: push          ( value seg list -- )
                getnode ?DUP
                IF                              \ v s l s n
                   4 ROLL 2 PICK 2 PICK         \ s l s n v s n
                   2+ !L node.insert
                ELSE
                   ." no free space " ABORT
                THEN ;
```

彈出串列 sample.list 頂端的值鍵入：

```
     sample.list pop
```

pop 之定義如下：

```
: pop       ( seg list -- value )
            node.remove ?DUP
            IF                          \ s n
              2DUP freenode             \ 將節點歸還給自由串列
              2+ @L                     \ 取得值
            ELSE
               ." empty list " ABORT
            THEN ;
```

印出所有串列 sample.list 中的值鍵入：

```
      sample.list .all
```

.all 之定義如下：

```
: .all          ( seg list -- )             \ 印出串列內含
                BEGIN                       \ s l
                  OVER SWAP @L ?DUP         \ s n  n
                WHILE
                  2DUP 2+ @L .              \ s n
                REPEAT
```

```
                DROP ;
```

刪除並歸還串列 sample.list 中所有的節點鍵入：

```
       sample.list kill
```

kill 之定義如下：

```
: kill          ( seg list -- )                 \ 歸還串列空間
                BEGIN                           \ s l
                  2DUP node.remove ?DUP         \ s l s n  n
                WHILE freenode                  \ s l
                REPEAT DROP 2DROP ;
```

測試串列

下列的字將會搜尋是否有特定的值存在串列 list. 中，例如

```
    5 sample.list ?in.list
```

當 5 在串列中時，將會在 5 之上傳回一個真旗標

```
: ?in.list   ( val seg list -- val f )
        >R FALSE -ROT R>              \ 0 v s l
        BEGIN                         \ 0 v s l
          ROT 2 PICK 2 PICK           \ 0 s l v s l
          @L ?DUP                     \ 0 s l v n  n
        WHILE
          3 PICK SWAP                 \ 0 s l v s n
          2+ @L OVER =                \ 0 s l v f - true if v'=v
          IF NIP NIP NIP TRUE EXIT    \ v tf
          THEN                        \ 0 s l v
          -ROT OVER SWAP @L           \ 0 v s n
        REPEAT
        NIP NIP SWAP ;                \ v ff
```

?pop 如果串列不是空串列則在串列頂端推出一個值，若串列是空串列，則留

下一個錯誤旗標在堆疊(stack)上，這個字是非常精巧有用的，假如你不確定串列是否是空串列，即使它是，你也不希望它會導致一些可能毀滅周遭記憶體的內含被誤用。

```
: ?pop      ( seg list -- value tf | ff )   \ 如果是空串列留下錯
            node.remove ?DUP                \ 誤旗標
            IF                              \ s n
              2DUP freenode                 \ 將節點歸還自由串列
              2+ @L TRUE                    \ 取值
            ELSE
              DROP FALSE
            THEN ;
```

假如串列是空的 ?list.empty 將傳回正確旗號

```
: ?list.empty           ( seg list -- f )
               2DUP ?pop                    \ 試看可否推出一數值
               IF                           \ 如果有東西在串列中
                 -ROT push FALSE            \ 在將之壓回--設錯誤
               ELSE                         \ 旗號
                 2DROP TRUE                 \ 否則設旗號爲真
               THEN ;
```

findpos< 將會尋找一個節點的適當位置，並在其後插入一個新的節點，其功能讓串列的儲存由小到大排列(ascending order)，例如要插入數字 35 到串列 sampie.list 中，使串列能被維護成由小到大的排列，鍵入：

```
      35 sample.list findpos< push
```

```
: findpos<      ( val seg list -- val seg node )
               BEGIN                    \ v s l
                 ROT 2 PICK 2 PICK      \ s l v s l
                 @L ?DUP                \ s l v n  n
               WHILE
                 3 PICK SWAP            \ s l v s n
                 2+ @L OVER >           \ s l v f - true if v'>v
                 IF
                   -ROT EXIT            \ v s l
```

```
            THEN                    \ s l v
            -ROT OVER SWAP @L       \ v s n
          REPEAT
          -ROT ;                    \ v s l
```

findpos> 將會尋找一個節點的適當位置，並在其後插入一個新的節點，其功能讓串列的儲存由大到小排列(descending order)，例如要插入數字 35 到串列 sampie.list 中，使串列能被維護成由大到小的排列，鍵入：

```
        35 sample.list findpos> push
```

```
: findpos>      ( val seg list -- val seg node )
          BEGIN                     \ v s l
            ROT 2 PICK 2 PICK       \ s l v s l
            @L ?DUP                 \ s l v n  n
          WHILE
            3 PICK SWAP             \ s l v s n
            2+ @L OVER <            \ s l v f - true if v'<v
            IF
              -ROT EXIT             \ v s l
            THEN                    \ s l v
            -ROT OVER SWAP @L       \ v s n
          REPEAT
          -ROT ;                    \ v s l
```

下列的字可被使用來尋找在串列中第 n 個節點的地址，例如，要取串列 sample.list 第 5 個節點的值，鍵入：

```
     sample.list 5 traverse.n 2+ @L
```

```
: traverse.n  ( seg list n -- seg addr )  \ 尋找第 n 個節點的位址
          ?DUP
            IF                          \ s l n
              0 DO                      \ s l
                OVER SWAP               \ s s l
                @L DUP 0=               \ s n f
                  IF
                    ." Beyond list end " ABORT
                  THEN
```

```
      LOOP                        \ s n
    THEN ;                        \ s l  if n=0
```

下列的字可以被用來尋找在串列中的節點個數，例如：

```
sample.list get.#nodes .
```

串列 sample.list 之節點數目將被印出

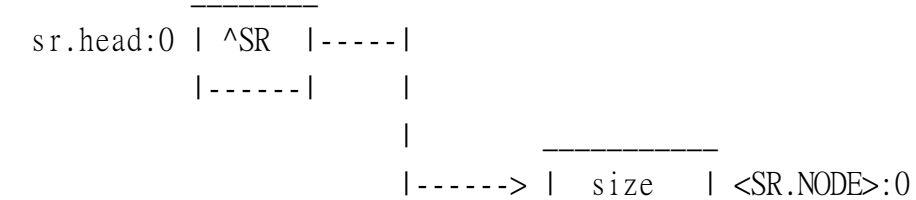
```
: get.#nodes    ( seg list -- n )
    0 -ROT                      \ 0 s l
    BEGIN                       \ cnt s l
      OVER SWAP                 \ cnt s s l
      @L ?DUP                   \ cnt s @l @l | cnt s 0
    WHILE                       \ cnt s @l
      ROT 1+ -ROT               \ cnt+1 s @l
    REPEAT
    DROP ;                      \ cnt
```

10.3 記錄(RECORDS)

FORTH 記錄的詳細討論請參看 Pountain 書第一章，下列的內容爲該章之精要：

在本節中的字群被用來產生一種相當具有彈性的串列記錄系統，在該系統中，每個記錄由記憶體中不同的區段來分隔或是每一個記錄存在不同的區段中，這些記錄可用記錄中的指標欄位來鏈結。各種不同記錄形態在此系統中均可被創立出來，且各種數量的記錄，可以被創立與鏈結進一個完整的結構體中，以形成一種完美的階層式結構系統。

在紀錄中各個不同的欄位可以擁有不同的大小，我們會藉著一個學生記錄系統，來說明這些字群的使用，每一個學生都被設定成如下的紀錄結構。



```
                    |---------|
                    | ^next   | <SR.NODE> [NEXT.SR]
                    |---------|
                    | ^name   | <SR.NODE> [NAME.SR]
                    |---------|
                    | ^addr   | <SR.NODE> [ADDR.SR]
                    |---------|
                    | ^data   | <SR.NODE> [DATA.SR]
                    |---------|
```

頭指標 sr.head：0 包含第一個學生紀錄的區段位址，學生節點的第 0 個欄位代表該記錄的欄位項數，第一個欄位在位移地址 [NEXT.SR] 內容中，爲一個指到下一個學生記錄的區段位址指標。
第二個欄位在位移地址[NAME.SR] 的地方，包含的一個指標指到學生記錄中存放姓名的位址指標。
第三個欄位在位移地址 [ADDR.SR] 的地方，內容爲一個指標指到記錄地址的區段位址指標，在地址區段裡，會形成另外一個記錄，其包含不同的欄位來存放省、市、街、區域號碼等不同長度的第二層記錄。
第四個欄位在位移地址 [DATA.SR] 的地方，包含著其內容爲一個指標指到資料記錄區段的區段位址，在該處形成第二層記錄，其包含各種不同用途的資料欄位，如性別、年齡、班級、科系、各科成績總平均等及種種其它有用的資料內容，像這樣的記錄可使用下列的字來建立。

```
VARIABLE total.bytes    2 total.bytes !      \   宣告欄位名稱

: field         ( n -- )
             CREATE
                total.bytes @ ,              \ 存位移位址
                total.bytes +!               \ 增加位移量計值
                IMMEDIATE
             DOES>    ( seg pfa -- seg off )
                @                            \ 取得欄位位址
                STATE @                      \ 如在編譯模式
                IF
                   [COMPILE] LITERAL         \ 便在編譯時間編入
                THEN ;
```

產生一個新的記錄實體

```
: make.instance         ( seg off n --- seg )
              DUP alloc.mem                    \ 配置一個欄位
              TUCK 0 !L                        \ 儲存實體的大小
              DUP 2SWAP !L                     \ 存新的段在
               ;                         \ seg:off 的地方
```

金城 註疏：

在 field 與 make.instance 的定義中，將 IMMEDIATE 放在冒號定義中，這是一種高難度的技巧，其目的在於改變未來執行的時間，以 field 爲例： field 爲一定義詞，其所創造出來的欄位名稱具有編譯時間的執行能力，如此一來，便能在冒號定義中使用欄位名稱，達到編譯時間計算的最佳化目的，別忘了，無論執行多少次，均無代價，因爲在第一次編譯時已經計算過了。

建立一個記錄定義詞

```
: define-record          ( -- )
              CREATE
                 total.bytes @ ,               \ 儲存實體的大小
                 2 total.bytes !               \ 清除位移量計器
              DOES>           ( seg off -- seg' )
                 @ make.instance ;

1  array  sr.head

: sr.list       ( -- seg off )
              sr.head 0 ;
```

以下各欄位(field)是到 SR 節點之位移地址

```
2 field [NEXT.SR]            \ 下一個節點的指標(seg addr)
2 field [NAME.SR]            \ 學生姓名的指標
2 field [ADDR.SR]            \ 學生地址記錄的指標
```

```
2 field [DATA.SR]              \ 學生資料記錄的指標
define-record SR-REC
```

注意字欄位(field)是一個定義詞，定義在學生記錄[SR.NODE]裡符合位移地址的名稱，當這些字在變數 total.bytes 裡建立了數字被儲存於建立詞的PFA，然後當欄位(field)被叫出時 total.bytes 的數字，藉著堆疊(stack)上的數字增加(注意 total.bytes 以一個起始的數字 2 開始的)此技術將產生正確的位移地址給不同範圍的欄位(field)，欄位(field)也可能被增加或減少，不須要擔心改變位移地址。

這個敘述

```
    define-record SR-REC
```

建立一個稱爲 SR-REC 的字，稍後這個字可被用來建立一個學生記錄的實體，我們定義下面的字來完成這個例子

以下各欄位(field)是到學生資料節點之位移

```
2 field [SEX.D]                \ 性別--一個字長度的字串 M 或 F
11 field [BIRTH.D]             \ 出生日期-- M/D/YR 字串
11 field [ENTER.D]             \ 入學日期-- M/D/YR 字串
2 field [MAJOR.D]              \ 科系
2 field [GPA.D]                \ 平均數乘 100
define-record DATA-REC
```

以下之欄位(field)是名字節點的位移地址

```
24 field [NAME.FN]                  \ 學生姓名 -- 計算長度的字串
define-record NAME-REC
```

下列欄位(field)是到地址節點的位移地址

```
16 field [STREET.AD]                \ 街名地址
16 field [CITY.AD]                  \ 城市
3 field [STATE.AD]                  \ 州名 -- 2 位元之縮寫
11 field [ZIP.AD]                   \ 郵遞區號
define-record ADDR-REC
```

```
0       VALUE   <SR.NODE>           \ SR 節點區段位址
0       VALUE   <NODE.NAME>         \ 姓名節點區段位址
0       VALUE   <NODE.ADDR>         \ 地址節點區段位址
0       VALUE   <NODE.DATA>         \ SR 資料節點區段位址
```

下列之詞是用來建立和刪除學生記錄

```
: >end.of.SR.list       ( seg list -- seg end.of.list.node )
             BEGIN                          \ s l
               2DUP @L ?DUP                 \ s l @l @l
             WHILE                          \ s l @l  or s l
               NIP NIP [NEXT.SR]            \ @l off
             REPEAT ;

: make.SR.record     ( seg off -- )
               >end.of.SR.list
               SR-REC DUP !> <SR.NODE>
               DUP 0 SWAP [NEXT.SR] !L
               DUP [NAME.SR] NAME-REC !> <NODE.NAME>
               DUP [ADDR.SR] ADDR-REC !> <NODE.ADDR>
               [DATA.SR] DATA-REC !> <NODE.DATA> ;

: zero.<nodes>          ( -- )
             0 !> <SR.NODE>
             0 !> <NODE.NAME>
             0 !> <NODE.ADDR>
             0 !> <NODE.DATA> ;


: release1.SR           ( ^SR -- )
               DUP [NAME.SR] @L release.mem
               DUP [ADDR.SR] @L release.mem
               DUP [DATA.SR] @L release.mem
               release.mem ;

: release.all.SR        ( seg off -- )
               2DUP @L ?DUP
```

```
IF
   BEGIN
      DUP [NEXT.SR] @L
      SWAP release1.SR ?DUP
   WHILE
    REPEAT
    0 -ROT !L
THEN
zero.<nodes> ;
```

增添一筆記錄鍵入

```
sr.list make.SR.record
```

你可以自由的選擇從鍵盤或其它磁碟檔案中輸入新資料到不同的欄位(field)
例如：

```
345 <NODE.DATA> [MAJOR.D] !L
```

將儲存 345 在適當的科系欄位中

金城 註疏：

FORTH 的資料結構，在此章中多半使用高階定義來寫，其目的是爲使學習者容易瞭解各種資料結構之原理及記憶體管理的一些技巧及理念，但在實際的高手應用中，會使用大量的低階程序，來獲取最佳化或速度效率，甚至連磁碟機之管理，在專業 FORTH 程式設計師理念上也是自己控制，而不喜歡透過他人之作業系統，以達成整體一致的系統架構。

第十一課　使用中斷處理的終端機程式

11.1　8086/8088 中斷處理(INTERRUPTS)

在此課我們將使用中斷處理來寫一個終端機程式，其可用來與其他的電腦溝通或下載 FORTH 程式給 FORTH 晶片(微處理機)，就如包含 MAX-FORTH 系統的 68HC11 單晶片型微電腦。

我們要以高達 9600 baud 的高傳輸率通訊，這意思說我們必需使用中斷處裡來儲存輸入的文字，即使在低速的螢光幕捲動下也不會有失誤，我們要寫一個中斷處理程序，當每次串列埠接收一個字母時，都會被自動執行，這一個中斷處理程序將會讀進字母並將他儲存在佇列中，終端機的主程式會在檢查鍵盤輸入跟檢查接收字母的佇列中來回切換，當從鍵盤輸入一個字元時將會從串列埠傳輸出此字母，當一個字母出現在佇列時(此字母由串列埠接收進來)，將會被顯示在螢光幕上或選擇性的儲存在磁碟機中。

中斷處理程序的區段位址與位移位址，必須儲存在記憶體第 0 段中的開始的中斷向量表格，DOS 的系統 25H(25 號)功能(設定中斷向量)和 35H(35 號)(取得中斷向量)可被用來完成這些工作，下列之 FORTH 詞使這些工作變的更容易。

```
PREFIX  HEX

        取得中斷向量

CODE get.int.vector   ( int# -- seg offset )
       POP AX
       PUSH ES
       PUSH BX                  \ AL = 中斷號碼
       MOV AH, # 35             \ DOS 35 號功能
       INT 21                   \ ES:BX = segment:offset 中斷
       MOV DX, ES               \ 處理的區段與位移
       MOV AX, BX
       POP BX
       POP ES
       2PUSH
       END-CODE
```

設定中斷向量

```
CODE set.int.vector   ( segment offset int# -- )
      POP AX                    \ AL = 中斷號碼
      POP DX                    \ DX = 位移地址
      POP BX                    \ BX = 段地址
      MOV AH, # 25              \ DOS 25 號功能
      PUSH DS                   \ 存回 DS
      MOV DS, BX                \ DS:DX -> 中段向量
      INT 21                    \ DOS 21 號功能
      POP  DS                   \ 恢復 DS 的內容
      NEXT
      END-CODE
```

金城 註疏：

一、baud 爲每秒所傳送的 bit 數。

二、FORTH 單晶片微電腦有：

(一)、Zilog 公司的 S8 。

(二)、Motorola 公司的 38HC 11 。

(三)、Rockwell 公司的 C18 。

(四)、Harris 公司的 RTX-2000 。

這些 CPU 均可以硬體指令執行 FORTH ，可簡化程式設計的過程及以最快速度執行,不會比組合語言的速度慢。

將在位址處的中斷處理程序存入中斷向量中

```
: store.int.vector      ( addr int# -- )
                        ?CS: -ROT set.int.vector ;
```

我們需要七、八、十課中的詞，所以我們將 FLOAD 這些檔案

```
DECIMAL

fload lesson7
```

```
fload lesson8
fload lesson10
```

11.2 8250 非同步通訊用晶片

串列通訊是由 8250 非同步通訊晶片來掌管 ，這顆晶片的中斷訊號輸出會接到電腦的中斷優先控制器晶片(8239)的 IRQ4 腳位上，這是第一個串列通訊埠 COM1 的中斷線輸出 IRQ3 腳位是接到串列通訊埠 COM2 的中斷線上。8250 的數據傳送控制暫存器的 OUT2 位元必需被設爲 1 ，以致能(enable) 8250 的 IRQ 輸出緩衝區。

```
HEX
300       CONSTANT      COM1          \ COM1 的基底地址
200       CONSTANT      COM2          \ COM2 的基底地址
0C        CONSTANT      INT#1         \ COM1 的中斷號
0B        CONSTANT      INT#2         \ COM2 的中斷號
EF        CONSTANT      ENABLE4       \ 4 號中斷致能罩斷(enable mask)
10        CONSTANT      DISABLE4      \ 4 號中斷除能罩斷(disable mask)
F7        CONSTANT      ENABLE3       \ 3 號中斷致能罩斷(enable mask)
08        CONSTANT      DISABLE3      \ 3 號中斷除能罩斷(disable mask)

\         Default COM1

COM1      VALUE    COM            \ 目前 COM 基底地址
INT#1     VALUE    INT#           \ 目前 COM 的中斷號
ENABLE4   VALUE    ENABLE         \ 目前 COM 的致能罩斷(enable mask)
DISABLE4  VALUE    DISABLE        \ 目前 COM 的除能罩斷(disable mask)
```

下列的數值常數，被用來加上 COM 的基底地址以獲得相關控制暫存器與資料暫存器的地址

```
F8        CONSTANT        txdata        \ 輸出資料暫存器(唯寫)
F8        CONSTANT        recdat        \ 接收資料暫存器(唯讀)
FC        CONSTANT        mcr           \ 數據傳送控制暫存器
F9        CONSTANT        ier           \ 中斷致能暫存器
FD        CONSTANT        lsr           \ 線況狀態暫存器
21        CONSTANT        imask         \ PIC 的罩斷暫存器
20        CONSTANT        eoi           \ 中斷值的結束
```

```
20      CONSTANT        ocw2            \ PIC ocw2

VARIABLE        int.vec.addr            \ 儲存中斷向量位移位址
VARIABLE        int.vec.seg             \ 儲存中斷向量區段位址
DECIMAL
```

金城 註疏：
PIC 爲中斷優先順序控制器

我們將使用 BIOS INT 14H (十進位的 20)，來設定傳輸埠的傳送速率 (AH:0)，這個動作必需在數據傳輸暫存器致能中斷位元被設定前完成，因爲 INT 14H 會使前述工作變的無意義。

下列之表格包含控制暫存器罩斷數值，以獲得傳送速率如下：300，1200，2400，4800 和 9600，無校定位元，八個資料位元，一個停止位元等訊息。

```
CREATE baud.table  67 , 131 , 163 , 195 , 227 ,
```

Index(索引)	Baud rate(傳送速率)
0	300
1	1200
2	2400
3	4800
4	9600

```
CODE INIT-COM   ( mask -- )
                POP     AX
                MOV     AH, # 0
                MOV     DX, # 0
                INT     20
                NEXT
                END-CODE
```

初設 9600 爲傳送速率如果你要改變傳送速率則修改下面這個字

```
: get.baud#     ( -- n )
                4 ;

: set.baud.rate     ( -- )
                get.baud# 2*
                baud.table + @
                INIT-COM ;
```

11.3 佇列的資料結構

在一個中斷處理中循環佇列將被用來儲存接收的字元

下列的指標被用來定義一個佇列(queue)

```
VARIABLE front   \ 指向佇列頭部前端的指標(在 front+1 爲最舊的資料)
VARIABLE rear    \ 指向佇列尾端的指標(在 rear 處爲最新的資料)
VARIABLE qmin    \ 指向佇列的第一個位元組
VARIABLE qmax    \ 指向佇列的最後一個位元組
VARIABLE   qbuff.seg    \ 佇列區段
10000   CONSTANT qsize  \ 佇列區段的大小
```

佇列初值化

```
: initq         ( -- )
          qsize alloc.mem qbuff.seg !   \ 規劃一塊記憶體記佇列
          0 front !                     \ front = 0
          0 rear !                      \ rear = 0
          0 qmin !                      \ qmin = 0
          qsize 1- qmax ! ;             \ qmax = qsize - 1
```

檢查佇列

```
: checkq        ( -- n tf | ff )
             front @ rear @ <>          \ 如前端 = 尾端
             IF                         \ 則爲空的佇列
                INLINE
                   CLI                  \ 除能中斷
                   NEXT
```

```
      END-INLINE
      1 front +!                \ 前端指標加 1
      front @ qmax @ >          \ if front > qmax
      IF
         qmin @ front !         \ then front = qmin
      THEN
      qbuff.seg @ front @ C@L   \ 取得一個字母
      TRUE                      \ 設旗號爲真
      INLINE
         STI                    \ 致能中斷
         NEXT
      END-INLINE
   ELSE
      FALSE                     \ 設旗號爲僞
   THEN ;
```

金城 註疏：

上例 checkq 中從 INLINE 到 END-INLINE 爲高階程序中插入低階程序。

把放在 AL 暫存器中的 byte 存入佇列中

```
LABEL   qstore
            PUSH    SI
            PUSH    ES
            MOV     SI, qbuff.seg
            MOV     ES, SI          \ ES = qbuff.seg
            INC     rear WORD       \ 將尾端指標加 1
            MOV     SI, rear        \ if rear > qmax
            CMP     SI, qmax
            JBE     2 $
            MOV     SI, qmin        \ then rear = qmin
            MOV     rear SI
2 $:        CMP     SI, front       \ if front = rear
            JNE     4 $             \ 則佇列滿溢
```

```
                DEC     SI              \ 將尾端減 1
                CMP     SI, qmin        \    if rear < qmin
                JAE     3 $             \    then rear = qmax
                MOV     SI, qmax
                MOV     rear SI
3 $:            POP     ES
                POP     SI
                RET
4 $:            MOV     ES: 0 [SI], AL  \ 否則將文字存入尾端
                POP     ES
                POP     SI
                RET
                END-CODE
```

中斷處理程序此程序將會從串列埠接收一字元，並將此字元儲存在佇列中

```
LABEL INT.SRV    ( -- )
                PUSH    AX
                PUSH    DX
                PUSH    DS
                MOV     AX, CS
                MOV     DS, AX              \ DS = CS
                MOV     DX, # COM           \ 如資料已準備好
                ADD     DX, # lsr
                IN      AL, DX
                TEST    AL, # 1
                JE      1 $
                MOV     DX, # COM
                ADD     DX, # recdat
                IN      AL, DX              \ 讀取
                CALL    qstore
1 $:            MOV     AL, # eoi
                MOV     DX, # ocw2
                OUT     DX, AL              \ 清除(中斷處理結束)
                POP     DS
                POP     DX
                POP     AX
                IRET
                END-CODE
```

設定中斷程序

```
: int.setup             ( -- )
        12 COM mcr + PC!            \ 數據控制器暫存器 out2 爲 0
        1 COM ier + PC!             \ 致能接收中斷
        INT# get.int.vector         \ 將舊的中斷向量儲存
        int.vec.addr !  int.vec.seg !
        INT.SRV INT# store.int.vector ;  \ 設定新的中斷向量
```

終端機的起始程序

```
: init.term             ( -- )
            initq                   \ 將佇列初值化
            int.setup               \ 設定中斷
            imask PC@
            ENABLE AND              \ 起始 irq4 (預定爲 COM1)
            imask PC! ;

: disable.term          ( -- )
            imask PC@
            DISABLE OR              \ 將 irq4 除能(預定爲 COM1)
            imask PC!
            0 COM mcr + PC!         \ 將 0 放入數據傳輸控制暫存器
            int.vec.seg @           \ 恢復原先的
            int.vec.addr @          \ 中斷向量
            INT# set.int.vector ;
```

金城 註疏：

在低階程序中的變數已在高階程序中宣告，不像組合的變數都要自己宣告，可見用 FORTH 寫組合是多麼簡單。

11.4 輸出文字到螢幕和(或)磁碟機

佇列中的字母將會被顯示在螢幕上，另一選擇是將其送到磁碟機檔案。

```
FALSE   VALUE   ?>disk          \ 是否爲送到磁碟機之旗號
0       VALUE   col.at          \ 儲存螢光幕游標位置
0       VALUE   row.at

VARIABLE t_handle               \ 終端機檔案敘述
CREATE edit_buff  70 ALLOT      \ 暫時性的編輯緩衝區

: $HCREATE      ( addr -- f )   \ 使用在 addr 處的一個計算過長度
                SEQHANDLE HCLOSE DROP  \ 的字串名稱，來建立檔案
                SEQHANDLE $>HANDLE
                SEQHANDLE HCREATE ;

: file.open.error       ( -- )
                33 12 65 14 box&fill
                ." Could not open file!!"
                KEY DROP ;
```

下列的詞將在螢幕上顯示一個視窗用來輸入檔案名稱，並將該資料檔打開，當按功能鍵 F1 時此程序將被執行。

```
: select.nil.file       ( -- )
                20 4 60 7 box&fill
                ." Enter a filename"
                "  " ">$
                edit_buff OVER C@ 1+ CMOVE
                21 6 edit_buff 30 lineeditor
                IF
                   edit_buff $HCREATE
                   IF
                      file.open.error
                   ELSE
                      SEQHANDLE >HNDLE @
                      DUP handl ! t_handle !
                      TRUE !> ?>disk
                   THEN
                THEN ;
```

```
: >term          ( -- )
                 t_handle @ handl ! ;
```

按下功能鍵 F1 將會啓動" 抓取資料並存檔的動作 "

```
: disk.on.nil       ( -- )
                 IBM-AT? !> row.at !> col.at
                 SAVESCR
                 select.nil.file
                 RESTSCR
                 col.at row.at AT ;
```

按下功能鍵 F2 將會結束" 資料抓取及存檔的動作 "

```
: disk.off       ( -- )
              t_handle @ ?DUP
              IF
                 close.file
                 0 t_handle !
              THEN
              FALSE !> ?>disk ;
```

從串列埠傳出 ASCII 碼。

```
: XMT            ( ascii -- )
              COM                        \ 使用 COM 的基底位址
              BEGIN
                 DUP lsr +               \ 等待線況狀態暫存器的
                 PC@ 32 AND              \ 第五位元爲 ON
              UNTIL
              txdata + PC! ;             \ 送出資料
```

按 CTRC-P 將會開啓或關閉印表機

```
: ?PRINT         ( -- )
              PRINTING C@ NOT PRINTING C! ;
```

送一個字母到螢幕

```
: do.emit     ( n -- )
            DUP 13 =                \ if CR
            IF
               DROP CR              \ 就做跳行迴轉的動作
            ELSE
               DUP 32 >=            \ 忽略其他控制字母
               IF
                  EMIT
               ELSE
                  DROP
               THEN
            THEN ;

: ?EMIT       ( n -- )
            127 AND                     \ 忽略校定位元
            DUP 13 =                    \ 忽略跳行及迴轉以
            OVER 10 = OR                \ 外的所有控制字元
            OVER 32 >= OR
            IF
               ?>disk                   \ 如果資料抓取的功能啓動
               IF
                  DUP >term send.byte   \ 則送至磁碟機
               THEN
               do.emit                  \ 否則送到螢幕
            ELSE
               DROP
            THEN ;
```

11.5 下載檔案

下列的詞可使用來下載譯入 MaxForth 的程式檔案到 68HC11，MaxForth 每次將會載入一行並將其編入字典中，在載入一行後將會送回一個 Line-Feed(L--F)(ASCII 10)給 PC 告知工作順利完成。

```
VARIABLE        wait.count
```

從串列埠傳送一個，在某地址長度爲 CNT 的字串

```
: xmt.str     ( addr cnt -- )     \ 送出文字並在結尾處加跳行字元
              0 DO
                 DUP I + C@
                 XMT
              LOOP
              DROP
              13 XMT ;
```

等待串列埠傳入某一個特定的字母

```
: wait.for    ( ascii -- )
           0 wait.count !
           BEGIN
              checkq                        \ 爲真狀 | 爲僞狀
              IF                            \ char n    | char
                 DUP ?EMIT                  \ char n
                 OVER =                     \ char f
                 0 wait.count !
              ELSE
                 1 wait.count +! FALSE      \ char ff
              THEN
              wait.count @ 32000 =          \ char f f
              IF
                 CONTROL G EMIT 2DROP       \ 如果沒有回應則發聲
                 CR ." No response..."      \ 警告
                 KEY DROP
                 2R> 2DROP                  \ 結束 wait.for
                 2R> 2DROP                  \ 結束 file.download
                 EXIT                       \ 結束 DO-KEY
              THEN
           UNTIL
           DROP ;
```

下載一個檔案到 68HC11

```
: file.download         ( -- )
        GETFILE
```

```
DARK
IF
   $HOPEN
   IF
      file.open.error
   ELSE
      ." File: " .SEQHANDLE CR
      BEGIN
         LINEREAD COUNT 2-          \ addr cnt
         OVER C@ 26 = NOT           \ 當還不是檔尾時
      WHILE                         \ 送出一行
         xmt.str
         10 wait.for                \ 等待 LF
      REPEAT
      CLOSE
   THEN
ELSE
   2R> 2DROP
   EXIT                             \ 結束 DO-KEY
THEN ;
```

金城 註疏：

把一件事情單純化可使程式設計師感到輕鬆可靠、精巧與效率，所以保持用最簡單的方法去解決問題，爲 FORTH 程式設計師必備的心理建設。

11.6 終端機之主程式

按 ESC 鍵將會離開終端機的主程式 HOST

```
: ESC.HOST      ( -- )
              disable.term              \ 除能所有的中斷
              disk.off                  \ 關檔案
              qbuff.seg @ release.mem   \ 釋放佇列緩衝區
              DARK                      \ 的記憶體
```

```
            ABORT ;
```

所有按鍵的跳躍表格。

```
EXEC.TABLE DO-KEY
     CONTROL P | ?PRINT     ( 印表機 ON/OFF )
        27  | ESC.HOST        (ESC 鍵 )
        187 | disk.on.nil     ( F1 )   188 | disk.off     ( F2 )
        189 | file.download   ( F3 )   190 | UNUSED       ( F4 )
        191 | UNUSED          ( F5 )   192 | UNUSED       ( F6 )
        193 | UNUSED          ( F7 )   194 | UNUSED       ( F8 )
        195 | UNUSED          ( F9 )   196 | UNUSED       ( F10)

        199 | UNUSED      ( HOME )   200 | UNUSED       ( UP )
        201 | UNUSED      ( PUP )    203 | UNUSED       ( LEFT )
        205 | UNUSED      ( RIGHT )  207 | UNUSED       ( END )
        208 | UNUSED      ( DOWN )   209 | UNUSED       ( PGDN )
        210 | UNUSED      ( INS  )   211 | UNUSED       ( DEL  )
     DEFAULT|  XMT


: T-LINK        ( -- )
              set.baud.rate
              CURSOR-ON
              FALSE !> ?>disk
              DARK
              ." 4thterm is on-line..." CR CR
              init.term ;
```

要執行終端機的程式可鍵入 HOST

```
: HOST          T-LINK
                BEGIN
                   KEY?
                   IF
                        KEY DO-KEY
                   THEN
                 checkq
                 IF
```

```
        ?EMIT
    THEN
AGAIN ;
```

第參篇

附　錄

附錄(a)程式之編譯

先在 F-PC 中測試發展應用程式，當一切動作均正確無誤時，使用 TCOM 產生 .COM 檔的獨立運用程式，可脫離 F-PC 發展系統，而得到最佳速率與最小的可執行檔案。

在 DOS 提示號下建入：

```
TCOM  <filename> <option> <.....>
```

選擇項

```
/code      = Enable the listing of assembly code.
             開啓列印組合語言檔案
/codeoff   = Disable the listing of assembly code .....(defaule).
             關閉列印組合語言檔案
/definit   = Include the defaule initialization from file DEFINIT.SEQ
             自動載入系統啓始規劃 DEFINIT.SEQ
/noinit    = Don't include any defaule initialization, user dose it.
             不要載入系統啓始規劃 DEFINIT.SEQ
             (但使用者需自己在程式中規劃)
/lst       = Generate a listing file with source, asm & symbols.
             產生原始程式組合語言與符號列印檔
/lstoff    = Don't Generate a listing filem ........(defaule).
             不要產生原始程式組合語言與符號列印檔
/opt       = Enable compiler optimization.
             啓動編輯最佳化功能
/optoff    = Disable compile optimization ........(defaule).
             取消編輯最佳化功能
/show      = Show symbols as they are compiled.
             在編輯時在螢幕上顯示出處裡過程
/showoff   = Don't show symbols as they are compiled ..(defaule).
             在編輯時在螢幕上不要顯示出處裡過程
/scr       = Enable the listing of source lines.
             啓動本文之檔案列印
/scroff    = Disable the listing of source lines .....(defaule).
             取消本文之檔案列印
```

```
/stay       = Stay in Forth after the compile finishes.
              在編輯後留在 FORTH 系統裡
/sym        = Generate a symbol file for BXDEBUG.
              產生給 BXDEBUG 除錯器用的符號檔
/symoff     = Don't generate a symbol file ........(defaule).
              不要產生給 BXDEBUG 除錯器用的符號檔
/help       = RE-display help screen. press the F1 key for MORE HELP.
              列印輔助訊息
/help2      = Display second help screen.
              顯示第二頁輔助訊息
/forth      = Append an interactive Forth to program.(need TFORTH.SEQ)
              在應用中加入 FORTH 的交談系統 TFORTH.SEQ
/dis        = Also append the disassembler.          (need DIS.SEQ)
              在應用中加入反譯器 DIS.SEQ
/debug      = Also append the debugger.              (need TDEBUG.SEQ)
              在應用中加入除錯器 TDEBUG.SEQ
/quite      = Reduce visual output, use with I/O redirection.
              使用 DOS 螢幕輸出，可以用來作輸出設備切換
/code-start <adr> = Start compiling code at <adr>.
                    機械碼區起始地址
/data-start <adr> = Start compiling data at <adr>.
                    資料區起始地址
/code-limit <n1>  = Size limit between CODE and DATA.  (default=$000)
                    在機械碼區和資料區中的距離
/ram-start  <adr> = Set the RAM segment in target memory. (ROMable)
                    設定 RAM 開始的區段地址
/ram-end    <n1>  = Set the end of target ram.         (default=$FFEE)
                    設定 RAM 終止地址
/bye              = Return to DOS ......(NOT a command line option)
                    鍵入 /bye 離開 TCOM (不可在命令列下達)
/dos              = Shell out to DOS ...(NOT a command line option)
                    轉回 DOS 的環境中   (不可在命令列下達)
```

附記：defaule 爲系統預設值。

在 FORTH 的觀念中，編輯過程分爲三種：

一、一般編譯(compiler)是 FORTH 高階與低階程序的編譯。

二、Meat Compiler 爲使用 FORTH 系統編譯出另一 FORTH 系統，稱爲系統介變。

三、目標編譯(Target Compiler)爲編譯出一個獨立的應用程式，而 TCOM 便是一

個目標編譯器，其他的一般語言如C，BASIC....等所稱的編譯，就是FORTH的目標編譯。

附錄(b)(c)(d)(e)

TABLE I. LANGUAGE DEFINITION OF FORTH

<character> ::= <ASCII code>
<delimiting character> ::= NUL | CR | SP | <designated character>
<delimiter> ::=<delimiting character> |
<delimiting character><delimiter>
<word> ::=<instruction> | <numder> | <string>
<string> ::=<character> | <character><string>
<number> ::=<integer> | -<integer>
<integer> ::=<digit> | <digit><integer>
<digit> ::= 0 | 1 | 2 | ... | 9 | A | B | ... | <base-1>
<instruction> ::= <standard instruction> | <user instruction>
<standard instruction> ::= <nucleus instruction> |
<interpreter instruction> |
<compiler instruction> | <device instruction>
<user instruction> ::= <colon instruction> | <code instruction> |
<constant> | <variable> | <vocabulary>

金城 註疏：

(一)在此處使用 BNF(Backos Naur Form)來定義一般最基本的 FORTH 語言定義，要小心的了解，FORTH 的語法是活的不是死的， FORTH 沒有保留字(reserved word)也沒有關鍵字(key word)，更沒有固定不變的語言定義，此點與所有的其他電腦言均衝突，所以事實上，FORTH 是沒有辦法給予 BNF 此等如此嚴謹的語法定義。此乃 FORTH 最危險也最迷人的地方。無形無相無法無章，更無定論，所有的語法，皆可由使用者自創，沒有其他語言那般強制性的語法來束縛。用的漂亮，想的精巧則爲神兵利器，要不就成了災難。所以，如果您是初學乍到的新手，則先遵循前人的傳統可能會比較輕鬆，也容易進入情況。

(二) BNF(Backos Naur Form)之符號有四種類型：

S ---- 啓始符號
N ---- <非終端符號> 其可代換。
T ---- 終端符號 其不可代換。

P ---- ::= 產生規則。
| 或

TABLE II· STANDARD INSTRUCTIONS

The list of standard instructions is basically that in FORTH-79 Standard Minor changes are made to comform to the instruction set used in the FORTH-83 Model.

<nucleus instruction> ::= ! | * | */ | */MOD | + | +! | - | -ROT | / | /MOD | 0< | 0= | 0> | 1+ | 1- | 2+ | 2- | < | = | > | >R | @ | ABS | AND | C! | C@ | CMOVE | D+ | D< | D- | DROP | DUP | EXECUTE | EXIT | FILL | MAX | MIN | MOD | MOVE | NOT | OR | OVER | R> | R@ | ROT | SWAP | U* | U/ | U< | XOR

<interpreter instruction> ::= # | #> | #S | ' | (| -TRAILING | . | <# | IN | ? | ABORT | BASE | BLK | CONTEST | COUNT | CORRENT | DECIMAL | EXPECT | FIND | FORTH | HERE | HOLD | NUMBER | PAD | QUERY | QUIT | SIGN | SPACE | SPACES | TYPE | U. | WORD

<compiler instruction> ::= +LOOP | , | ." | : | ; | ALLOT | BEGIN | COMPILE | CONSTANT | CREATE | DEFINITIONS | DO | DOES> | ELSE | THEN | FORGET | I | IF | IMMEDIATE | J | LEAVE | LITERAL | LOOP | REPEAT | STATE | UNTIL | VARIABLE | VOCABULARY | WHILE | [| [COMPILE] |]

<device instruction> ::= BLOCK | BUFFER | CR | EMIT | EMPTY-BUFFERS | FLUSH | KEY | LIST | LOAD | SCR | UPDATE

金城 註疏：

在 F-PC 之後許多新的 FORTH 系統已改用一般的檔案格式，但仍有許多人偏好 FORTH 所獨有 BLOCK 檔案格式，如 polyFORTH 與 URFORTH 仍

使用 BLOCK。

TABLE III· USER INSTRUCTIONS

The statement in paranthesis is according to the FORTH syntax.

COLON INSTRUCTION
<colon instrucion> ::= <structure list>
(: <colon instrucion> <structure list> ;)
<structure list> ::= <structure><delimiter> |
<structure><delimiter><structure list>
<structure> ::= <word> | <if-else-then> | <begin-until> |
<begin-while-repeat> | <do-loop>

<if-else-then> ::= IF<delimiter><structure list>THEN |
IF<delimiter><structure list>ELSE<delimiter><structure list>THEN
<begin-until> ::= BEGIN<delimiter><structure list>UNTIL
<begin-while-repeat> ::=
BEGIN<delimiter><structure list>WHILE<delimiter><structure list>REPEAT

<do-loop structure> ::= <structure> | I | J | LEAVE
<do-loop structure list> ::= <do-loop structure><delimiter> |
<do-loop structure><delimiter><do-loop structure list>
<do-loop> ::= DO<delimiter><do-loop structure list>LOOP |
DO<delimiter><do-loop structure list>+LOOP

CODE INSTRUCTION
<code instruction> ::= <assembly code list>
(CODE <code instruction> <assembly code list> END-CODE)
<assembly code list> ::= <assembly code><delimiter> |
<assembly code><delimiter><assembly code list>
<assembly code> ::= <number><delimiter>, | <number><delimiter>C,

金城 註疏：
FORTH 的 Assembler(組譯程式)所用的語法比較獨特，與傳統的組譯程式不太相同，甚至每一套不同的 FORTH 系統都會有所出入。但只要能

將正確的機器碼放置在正確的記憶體位置就能正常工作了。大多數玩 FORTH 的好手，都喜歡用自己所寫的 Assembler(組譯程式)，你也可以寫一個，自己的版本。

CONSTANT INSTRUCTION
<constant> ::= <number>
(<number> CONSTANT <constant>)

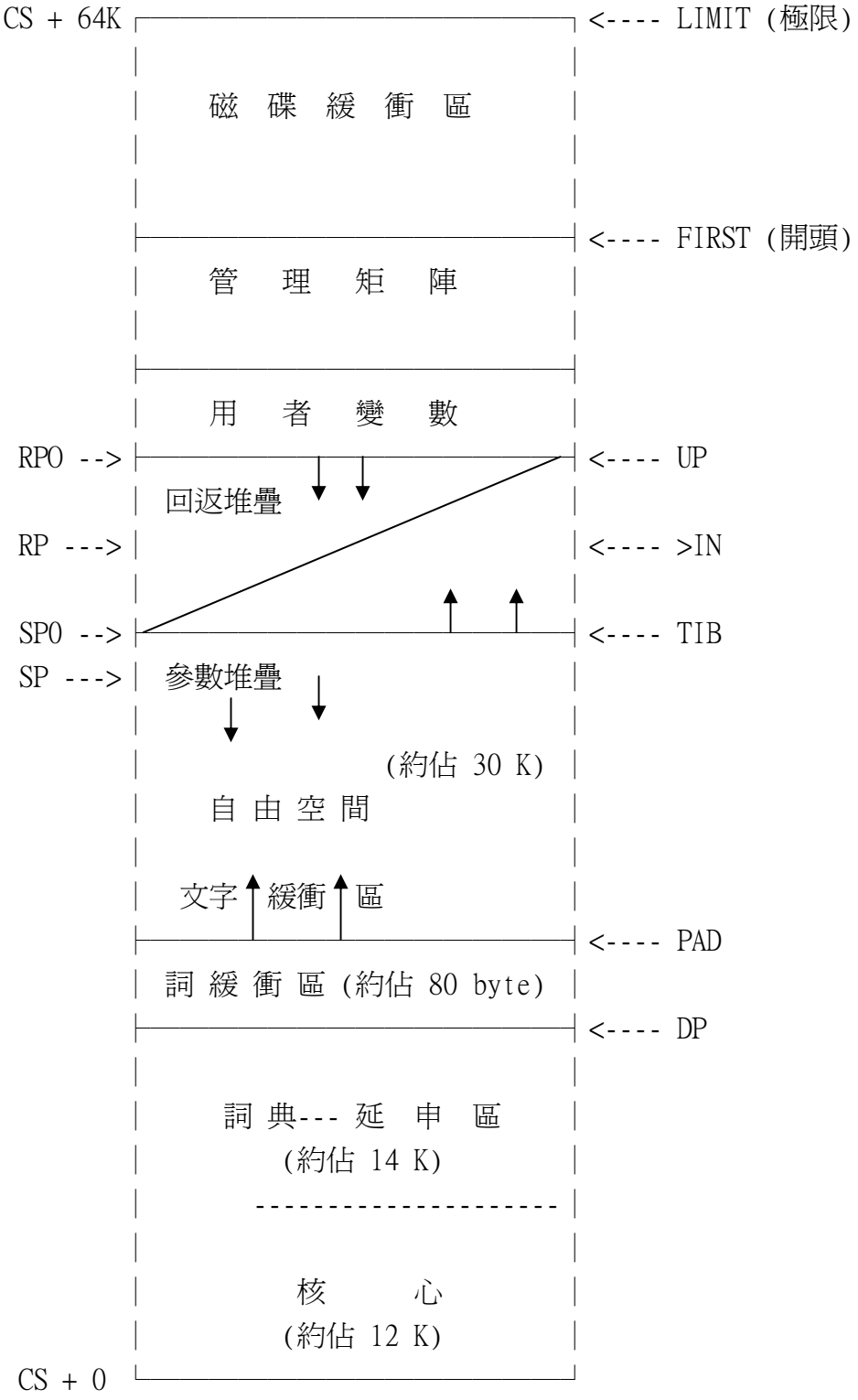
VARIABLE INSTRUCTION
<variable> ::= <address>
(VARIABLE <variable>)
<address> ::= <integer>

VOCABULARY INSTRUCTION
<context vocabulary> ::= <vocabulary>
(VOCABULARY <vocabulary>)

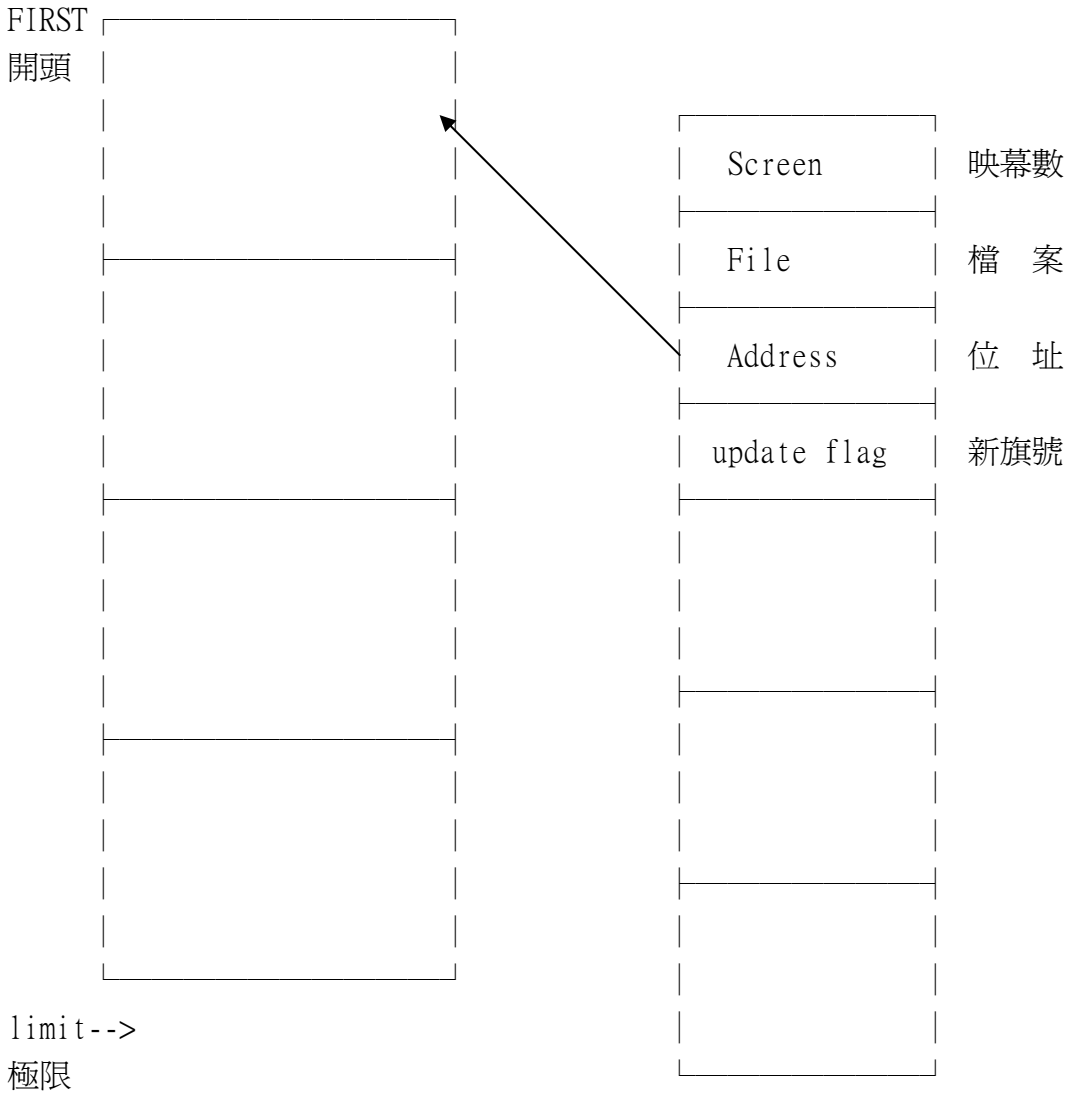
TABLE IV· CREATING NEW DEFINING INSTRUCTIONS

<high-level defining instruction> ::=
CREATE<delimiter><compiler structure list>{DOES>}<delimiter>
<interpreter structure list>;
(: <high-level defining instruction> CREATE <structure list> DOES>
<structure list> ;)
<low-level defining instruction> ::=
CREATE<delimiter><compiler structure list>;CODE<delimiter>
<interpreter assembly code list>
(: <low-level defining instruction> CREATE <structure list> ;CODE
<interpreter assembly code list>)
<compiler structure list> ::= <structure list>
<interpreter structure list> ::= <structure list>
<interpreter assembly code list> ::= <assembly code list>

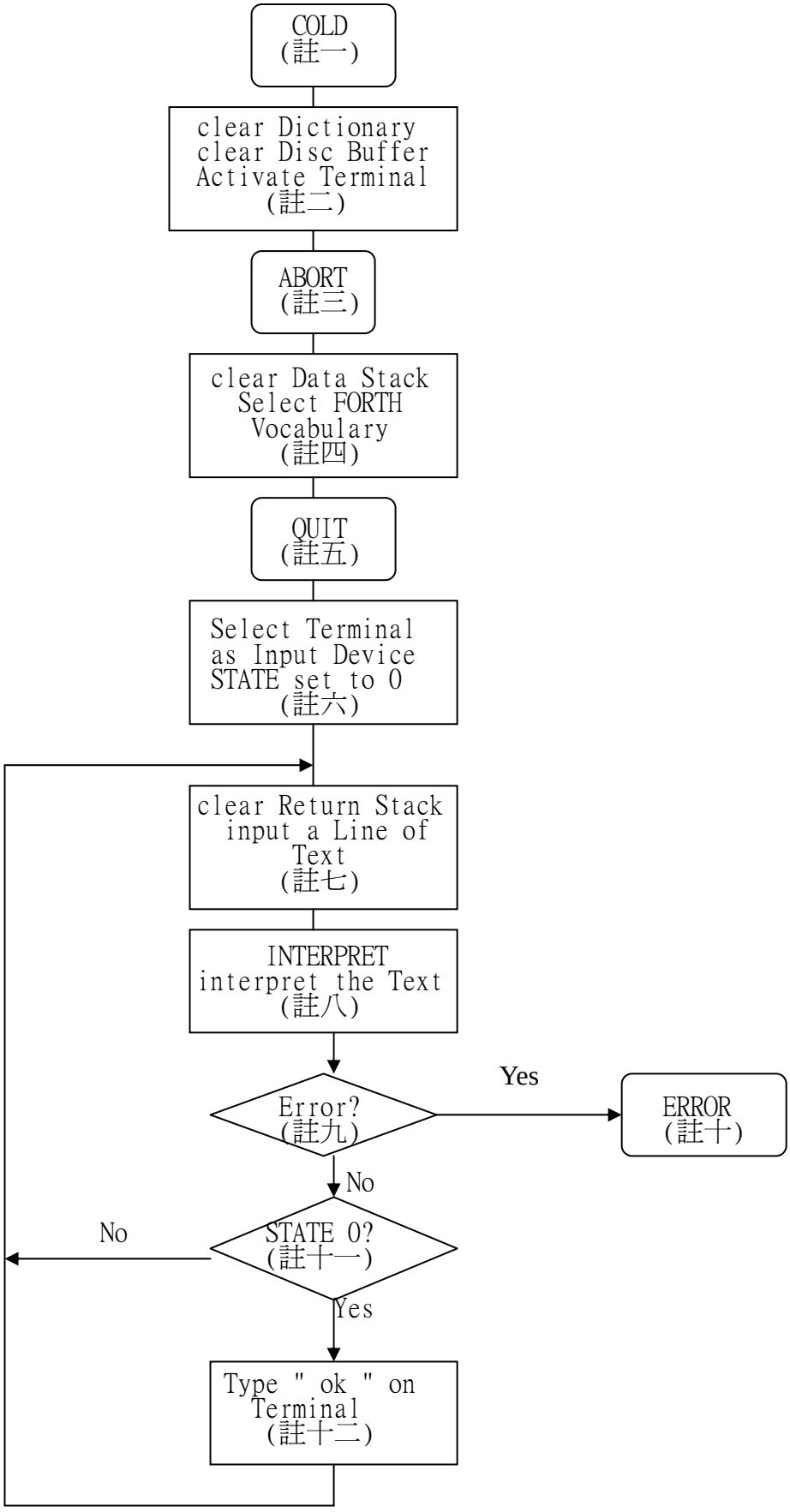
附錄(f)f83 系統記憶圖



附錄(g)F83 虛擬記憶體緩衝區



附錄(h)(i)(j)(k)(l)(m)(n)迴路



註一：在執行冷動(COLD)以前，對磁碟緩衝區做的一些動作：程式自磁碟載入記憶體後，微處理器開始自程式段進入點(Entry)的地方開始執行。 在此處有一段組合式碼，對記憶體需求量的計算，把 FIRST 與 LIMIT 設好。再把返回堆疊與終端機輸入緩衝區的地址設定，安排好參數堆疊等等的工作，最後跳到執行 COLD(冷動)。

冷動時會執行兩件事情：

1. BOOT(起動)：將起始虛擬的符式機器。
2. QUIT(離開)：進入文字翻譯器內。

由於 BOOT 是延緩定義詞(DEFER BOOT)。會誘導使用者選擇起始程式；若不執行起動(BOOT)程式。那就是 ABORT。

註二：清除辭典與清除 FORTH 系統所使用的磁疊緩衝機並啓動終端機。

進去以後，首先清除辭典，再清除磁碟緩衝區，當做完這些動作以後，再送"OK"到終端機去，以表示 FORTH 已準備好接受使用者下達指令了。

註三：終止執行。當無法順利執行 FORTH 的指令時。

從使用者變數 SPO 取得參數堆疊之起始指標。把參數堆疊指標填入虛擬符式計算機的暫存器中。跳到 QUIT(離開)程序任何其中之一的返回點(the peint of return)。強制終端機執行。

註四：清除參數堆疊選擇 FORTH 詞彙。

把 SP 設爲 SP0，無條件的指定 FORTH 詞彙來用。

註五：離開。

QUIT(離開)內部所做的動作如下：

1.設定終端輸入設備，選擇鍵盤爲輸入來源。
2.先定爲翻譯態(interpreting)。
3.重新設好迴返堆疊。
4.顯示系統狀態。
5.若有命令進來，則執行。
6.再看看是否仍在翻譯態中，是：則再回去執行第二個命令；並回應"OK"，不是；則到編碼器(compiling)中進行編譯。

符式主迴路:

獲得更多來自終端機和翻譯器的輸入。若所有輸入的命令都能成功的執行，便會輸出"OK"的回應。使用者經由終端機的鍵盤給予命令時，常常都會呼叫 FORTH 文字翻譯器迴路。文字翻譯器取用這些命令並會再回去處理其它的 FORTH 指令此迴路直至電源關閉或被強迫去執行 BYE 這個命令，便會回到原作業系統底下。

註六：選擇終端機爲輸入裝置並把狀態設爲 0 。
因此時不一定在終端機上，故需選擇終端機。並把狀態設爲"0"，即代表翻譯器，非"0"爲編碼器。

註七：清除迴返堆疊並等待輸入一行文字。
迴返堆疊主要功能是用來存取返迴地址。因此可以把參數利用>R 存放於此，但必須在定義結束前，利用 R>將暫存於迴返堆疊頂端的參數轉移到參數堆疊上，如此便完成清除堆疊的動作。然後等待輸入一行文字。並開始進入 FORTH 的翻譯器迴路。

註八：翻譯器翻譯文字。
翻譯器迴路：
翻譯器會剖析出該輸入的詞之屬性若該詞是已經定義的，便執行它；否則把該字轉換成數字，並放到參數堆疊上。在翻譯文字時，首先會檢查堆疊的深度有沒有被破壞(overflow / underflow)。獲得下一個詞輸入的屬性，並返回該詞的執行碼地址(cfa)和旗標。以便使用在參數堆疊上的執行碼地址(cfa)去執行；否則把該詞轉換成數字。測試該數字是不是雙精數的實數?如不是，只有單精數。於是把較高位的 16 位元(bit)丟棄，保存低位的 16 位元作爲單精數，舉起錯誤的旗標，看看還有沒有事做?是不是終結線?若到達終結線便離開此迴路；否則再回去翻譯下一個詞。

註九：錯誤 ?。(是錯誤發生嗎?)

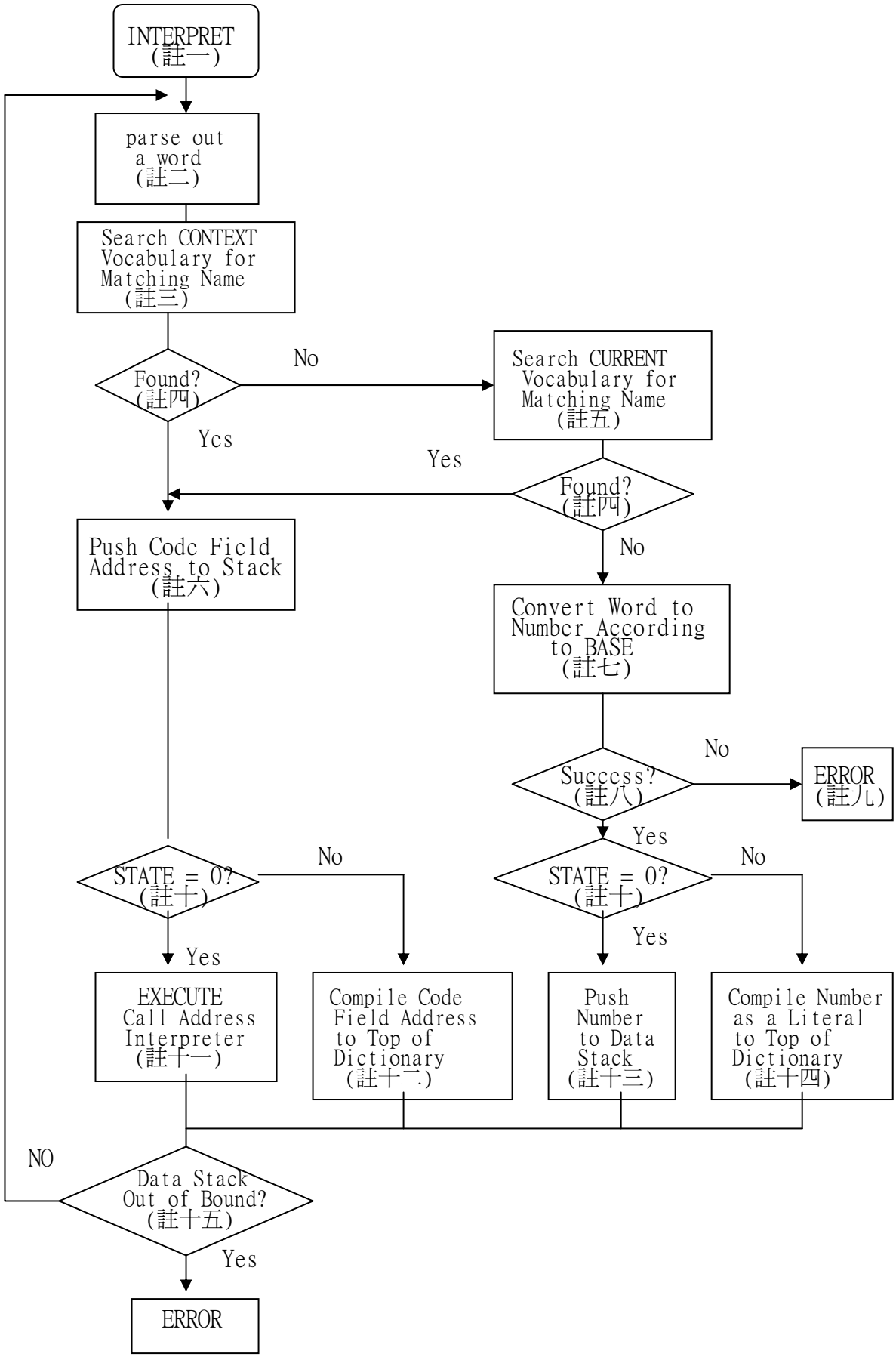
註十：錯誤。則去做錯誤情況的處理。

註十一：狀態是否設爲 0? 。(在翻譯(直譯)的狀態下嗎?)

註十二：在終端機上打出"ok"。表示 FORTH 又準好接收使用者的輸入指令了。

註:FIRST：磁碟緩衝區起始處。
LIMIT：磁碟緩衝區結束加 1 之址。
在 F-83 及標準的 BLOCK 檔案 I/O 的 FORTH 系統中，因爲磁碟機是 FORTH 自己管理的，所以需要設定磁碟緩衝區，以方便虛擬記憶體的管理與高效率執行。但在 F-PC 中，因爲是使用 DOS 的循序檔案功能，所以不在使用 BLOCK 的結構來管理檔案，自然也就不需要設 FIRST 與 LIMIT 來管理磁碟緩衝區。

Interpreter 迴路



註一：翻譯。

註二：分析一詞。
將此字串搬到墊(PAD)，按字母大小寫去尋找詞典。F-PC 以利用 ?UPPERCASE 詞將字母一律轉換爲大寫，可參考 KERNEL2 ，ATBL ，UPPER ，?UPPERCASE 。

註三：首先去 CONTEXT 所指的詞彙查詢比對。

註四：是否發現? 。
如果不能在現有詞典中發現則將 here 0 放回堆疊。

註五： 去 CURRENT 詞彙查詢比對。

註六：如果發現爲已定義之詞，則將該詞執行碼地址放入堆疊。
如果該詞爲立即詞，則再放個 -1，普通詞則放 1。

註七：如不能找到同名稱之定義詞，則轉由 NUMBER 嚐試依基底轉換成數目。
把字串轉成雙精密度數目。

註八：轉換爲數目是否成功? 。

註九：錯誤。
如果既在詞典中找不到已定義之詞，也無法轉爲當時基底下的數目，則執行 ERROR 。

註十：狀態是否爲 0? 。
狀態 0 爲翻譯態，狀態非 0 爲編譯態。

註十一：執行堆疊上的 CFA (呼叫 NEXT 內部解釋器) 。

註十二：編譯執行碼地址到字典頂端。

註十三：將此數目放入參數堆疊。

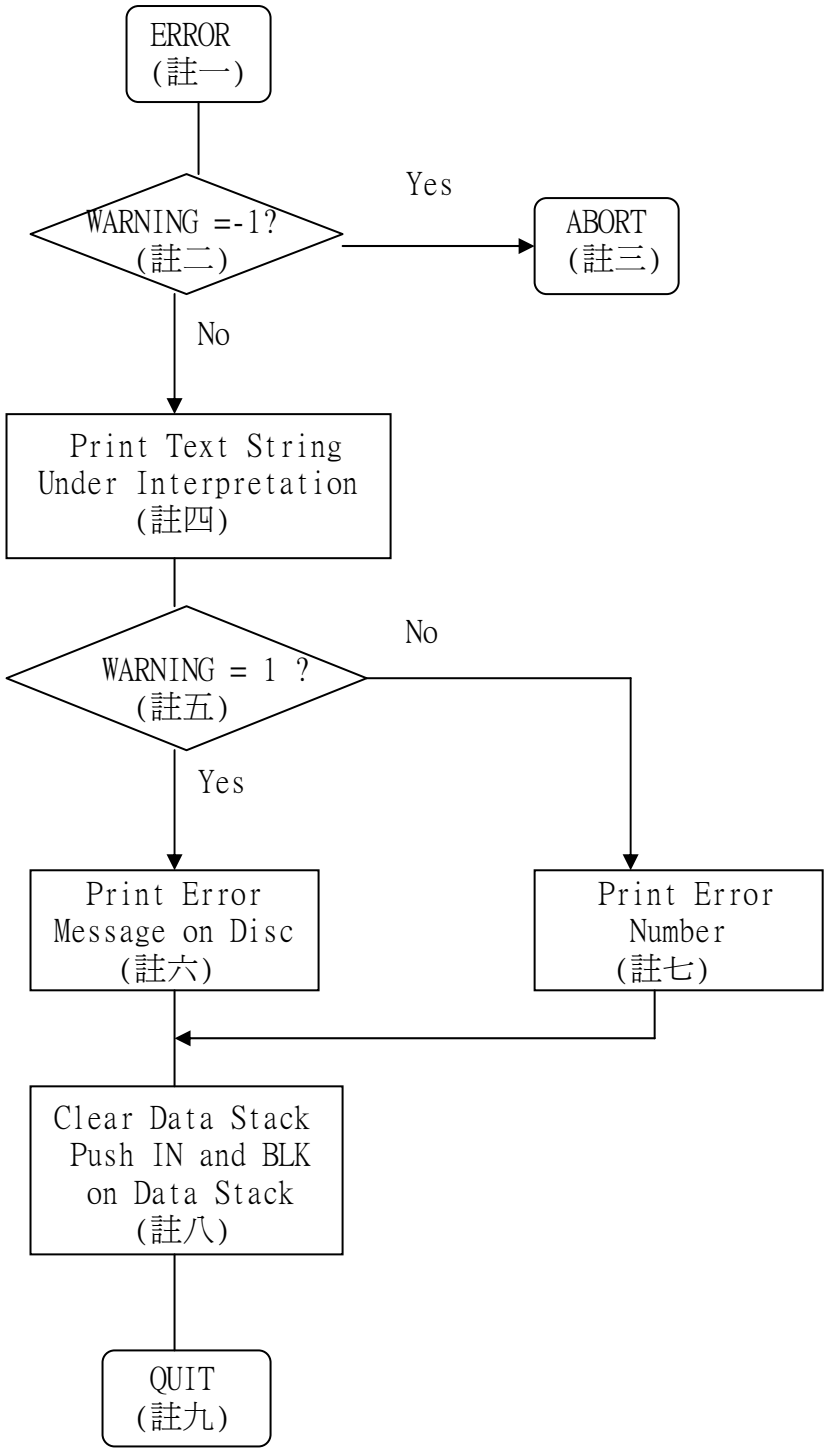
註十四：編譯此數目到字典頂端。以 LITERAL 的型式編入。

註十五：參數堆疊是否超過界限。

註十六：錯誤。

註：CONTEXT 爲目前所要搜尋的字典詞彙區。
CURRENT 爲目前編譯的(寫入)字典詞彙區。

Error 迴路


ERROR
(註一)
WARNING =-1?
(註二)
Yes
ABORT
(註三)
No
Print Text String
Under Interpretation
(註四)
WARNING = 1 ?
(註五)
No
Yes
Print Error
Message on Disc
(註六)
Print Error
Number
(註七)
Clear Data Stack
Push IN and BLK
on Data Stack
(註八)
QUIT
(註九)

註一：錯誤。
執行 ERROR 通知和系統重新開始。

註二：警告= -1 ? (爲真嗎?)。

註三：放棄。
表示嚴重錯誤，則執行放棄(ABORT)，清除參數堆疊， 並執行 QUIT。

註四：在 Interpretation 之下印出 Text 字串。
遇不知道，不認識的字並無法轉成數目時印出 what ? 。

註五：警告 = 1 ? (表不嚴重錯誤)。

註六：把錯誤訊息印在 Disc 上。

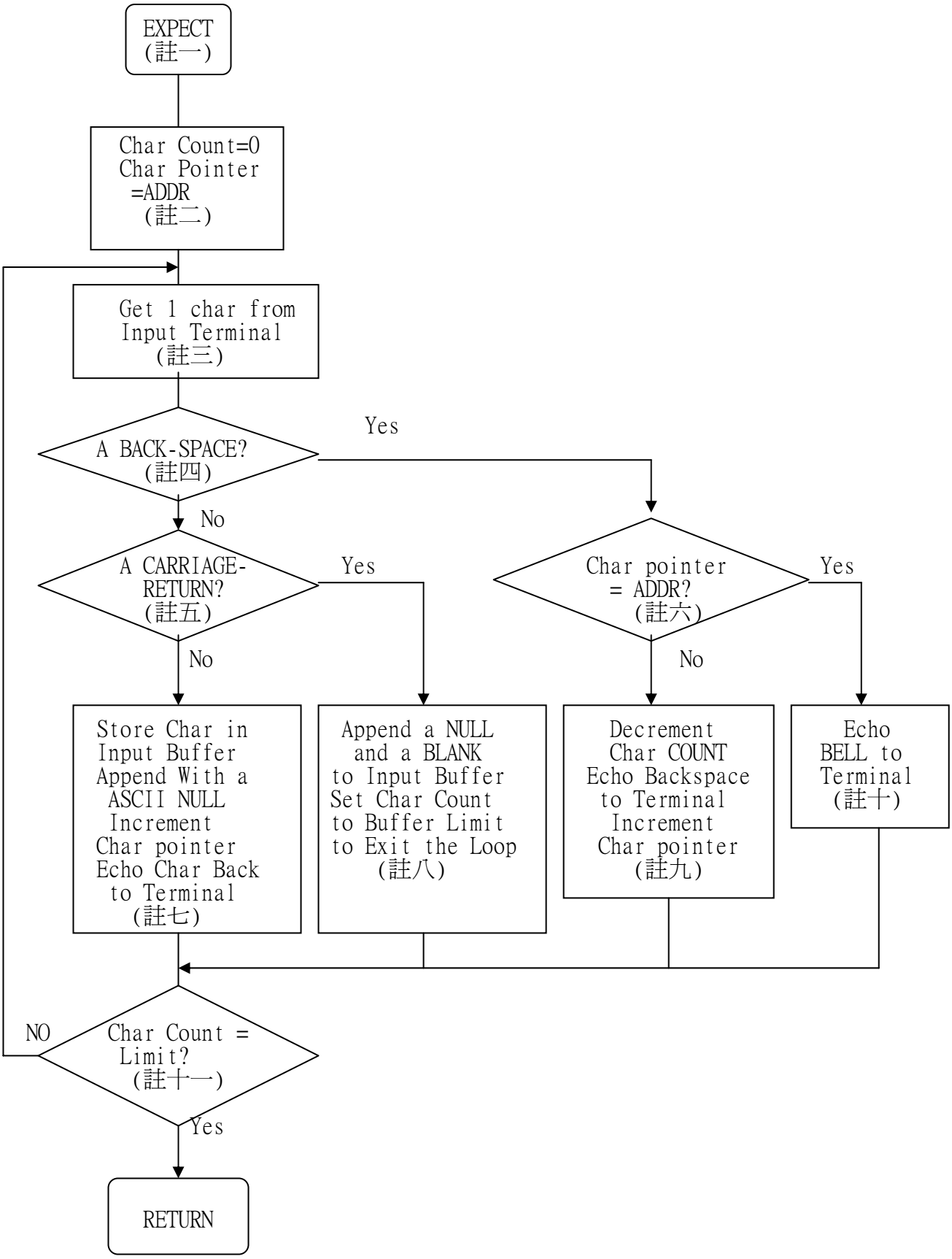
註七：印出錯誤的號碼。
表示印出錯誤是屬於那種情況的錯誤。

註八：清除參數堆疊、並把 IN 和 BLK 放至參數堆疊上。
從 BLK 可知錯誤是位於記憶體的第幾個 block 從 IN 可知錯誤是位於第幾個字。

註九：然後執行 QUIT 重新進入 FORTH 迴路。

註：(一) ERROR 只有在 FIG-FORTH 中才有定義，F-PC、F83 中沒有。
(二) F-PC 之錯誤處理係在 NUMBER 轉換數目失敗後交由 ?MISSING 處理，由 ABORT" ，(ABORT")，?ERROR 各詞組成，功能與上述 ERROR 類似，變數 WARNING 則在爲真時，提示使用者新定義名是否重覆。

EXPECT 迴路


EXPECT
(註一)
Char Count=0
Char Pointer
=ADDR
(註二)
Get 1 char from
Input Terminal
(註三)
A BACK-SPACE?
(註四)
Yes
No
A CARRIAGE-
RETURN?
(註五)
Yes
No
Char pointer
= ADDR?
(註六)
Yes
No
Store Char in
Input Buffer
Append With a
ASCII NULL
Increment
Char pointer
Echo Char Back
to Terminal
(註七)
Append a NULL
and a BLANK
to Input Buffer
Set Char Count
to Buffer Limit
to Exit the Loop
(註八)
Decrement
Char COUNT
Echo Backspace
to Terminal
Increment
Char pointer
(註九)
Echo
BELL to
Terminal
(註十)
NO
Char Count =
Limit?
(註十一)
Yes
RETURN

註一：期待 USER 從 Terminal 輸入一些訊息。
EXPECT:等待從 Keyboard 送入 a string，而且放在 Terminal Input Buffer；把輸入的字數記錄在 SPAN。
SPAN : 是一變數，內容爲自 EXPECT 讀入的字元數目。

註二：設 Char Count=0 ，設定 Char Pointer 的位址。
設定一個 DO---LOOP 迴圈，一般以 'TIB 爲 char count=0 的地址 (nIndex) 'TIB + input buffer 長度爲終點(nLimit)逐一接受 Keyboavd 送入的字串。

註三：接著從 Input Terminal 取得一個字元 。

註四：判斷是 BACK-SPACE 嗎 ? 。
如果是則要調整 ADDR 亦即將接收 Key 的迴圈 Index-1。
BACK-SPACE 送一個 backspace 及調整 charcter count。

註五：判斷是 CARRIAGE-RETURN? 嗎。
如果是表是一行結束。

註六：檢查 Char pointer = ADDR 嗎? 。
此時的 char pointer 是否指在 DO---LOOP 迴圈中的 nIndex 的起點值?

註七：就把此 Char 放在 Input Buffer；提供一 ASCII = NULL 的空間；把 Char pointer 加 1 繼續執行 DO---LOOP 迴圈，若有 Char 進來，會一一把 NULL 蓋掉。
從目前的 addr 起用 NULL 填滿，常以此數清除字串的記憶區段。

註八：提供一行 NULL 和 BLANK 給 Input-Buffer CARRIAGE-RETURN 後表這一行已結束設定 Char-Count 直到 Buffer Limit 跳出 LOOP。

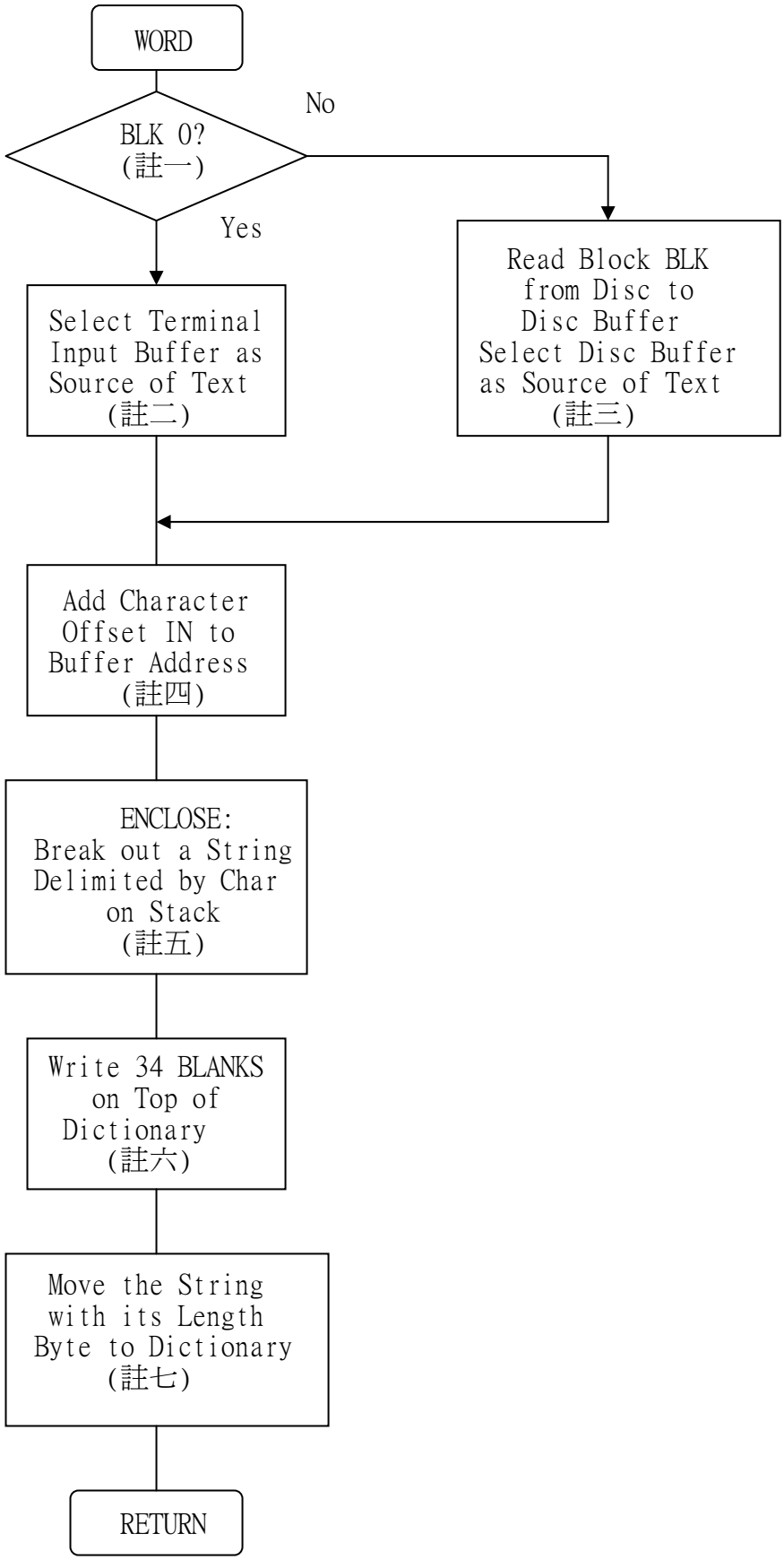
註九：把 Char Count 減 1 ，到 Terminal 呼叫 BACKSPACE 把 Char pointer 加 1。(參考註四)
解釋 BACKSPACE 的動作。

註十：表示第一個動作是 BACKSPACE 必須提出一個警告。
因目前 char count=0 位於接收字串迴圈的起點沒有任何 character ，所以無法做 BACKSPACE 的動作。響鈴以示警。

註十一：檢查 Char Count 是否極限了?
設定的 DO---LOOP 迴圈 nIndex 是否等於 nLimit ? 亦即
input buffer 已被田填滿 ?

註： (一) BLANK :把一 string 用 blanks 填滿。
(二) Limit :The address above the top disk buffer 磁碟緩衝區結束加 1 的地址。
(三) F-PC 的 EXPECT 已作大幅度調整，接收 KEY 迴圈並改爲 BEGIN WHILE REPEAT ，另增加命令貯存陣列，可記住已執行十二列命令，請參閱 XEXPECT.SEQ。

WORD 迴路


WORD
BLK 0?
(註一)
No
Yes
Select Terminal Input Buffer as Source of Text
(註二)
Read Block BLK from Disc to Disc Buffer
Select Disc Buffer as Source of Text
(註三)
Add Character Offset IN to Buffer Address
(註四)
ENCLOSE:
Break out a String Delimited by Char on Stack
(註五)
Write 34 BLANKS on Top of Dictionary
(註六)
Move the String with its Length Byte to Dictionary
(註七)
RETURN

註一：BLK 是否爲 0 ?
BLK = 0(僞)由 Terminal 輸入。
BLK <>0(真)由磁碟緩衝機 輸入。

註二：設定 Terminal Input Buffer 爲輸入來源。

註三：求得 BLK 號碼設定磁碟緩衝區爲輸入來源。
BLK 號碼經除法得商、餘數後至磁碟緩衝區取出。

上列註二、註三之目的 : 尋找 source code 之 Input Buffer。

註四：將 Char Offset IN 加上緩衝區地址成爲緩衝區指標。

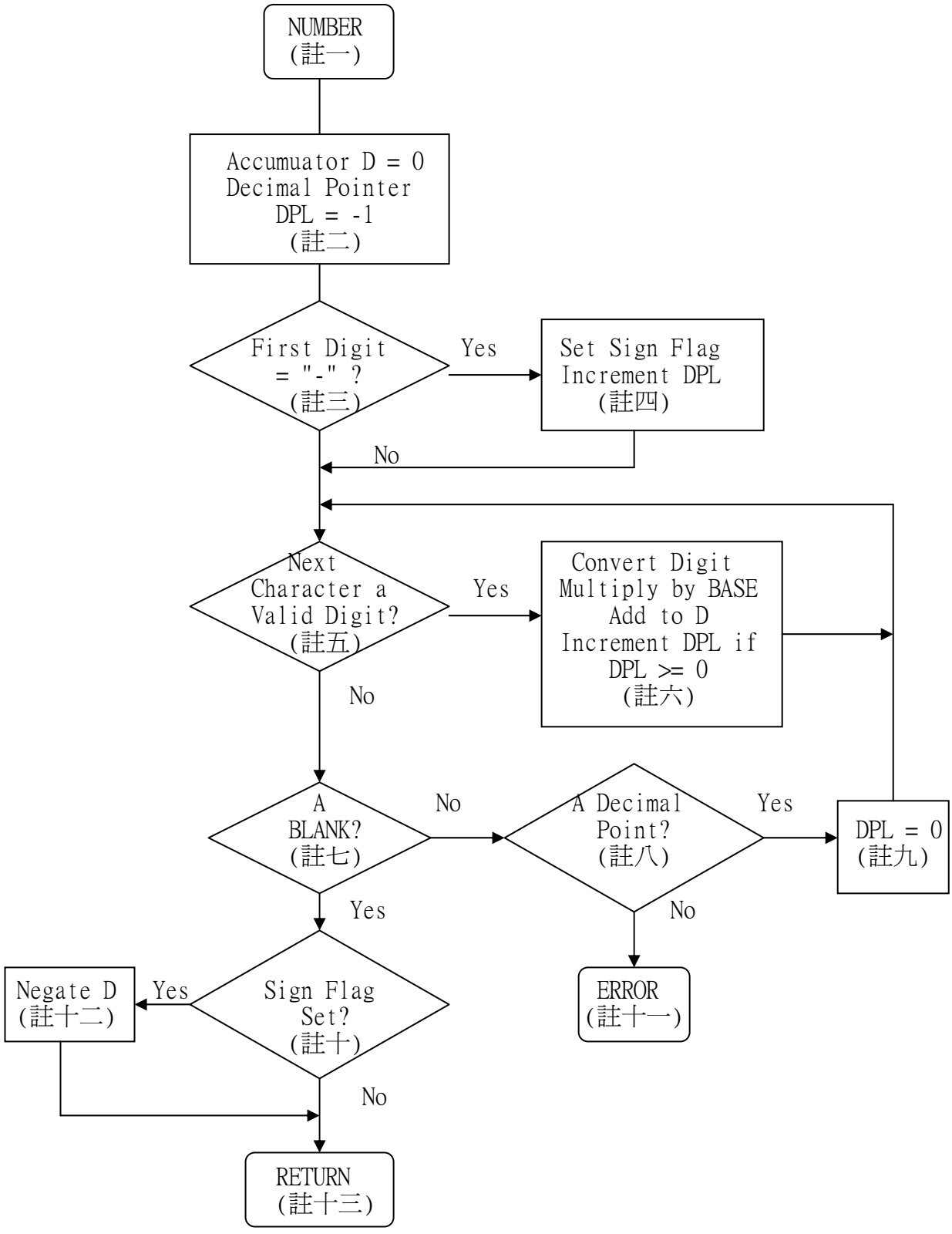
註五：在 Stack 中 將字串以空白鍵分隔。

註六：在字典頂端留下 34 個空格，32 -->最大長度 2 -->指標。

註七：將字串及 byte 數的長度搬到字典中

註：(一) WORD : 分析說明由字元劃界限的輸入字串，且將字個數傳至 HERE。
BLK : 在磁碟上預備翻譯的塊號碼。
(二) F-PC 已將 Word 全部改寫爲組合語言，達最佳化需求， F-PC 本身無 BLK ，故 Terminal Input Buffer 永遠是輸入來源，在經過 Word 分析後，若發現有關檔案之詞時(如 Fload)，則透過中斷載入該檔內容。

Numeric Conversion 迴路



註一：將字串轉換成數字(數字是 double number)。

註二：設累加器 D=0 設變數 DPL=-1。(DPL 爲 Decimal Poiuter(小數點)指標) DPL 設爲 -1 是因爲沒有 " . " " / " " , " " - " 符號在一串數目中出現。

註三：第一個位元是 = "-" ? 。

註四：設定 Sign 旗號 把 DPL 增加 1。
把" - "的 flag 放至 return stack 中重設 DPL 繼續下一個 digit 進行 conversion。

註五：下一個字元是一個有效位元嗎? 。

註六：轉換位元是根據其 BASE 把位元的數乘以 BASE 得到結果，並把結果加至 D 中當 DPL >= 0 時把 DPL 增加 1。

註七：是一個空白嗎? 。
如果空白表示 conversion 結束了。

註八：是一個 " . " " / " " , " 或 " - " 嗎?

註九：設 DPL = 0。
遇到是" . " " / " " , " " - "等符號時則重設 DPL = 0 這樣才能繼續轉換下一個 digit。

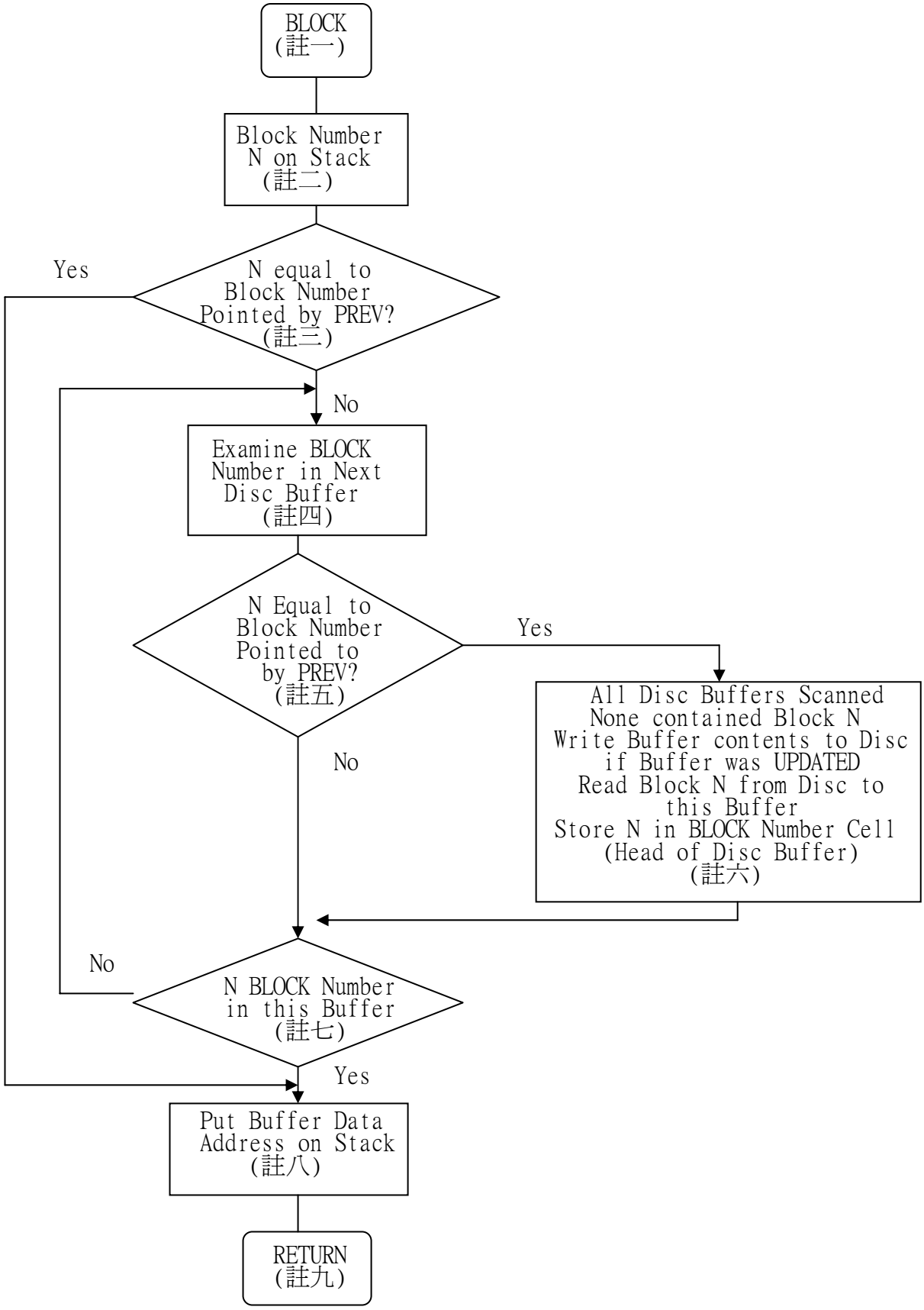
註十： Sign 旗標是否已設定嗎? 。
判斷是否有設 " - " 的旗號。

註十一：既不是空白 (表示結果) 又不是 Point 因此不是所允許的 digit
則產生 ERROR。

註十二：則得負 D 。
把結果放至 Stack 中因此就可得負 D。

註十三：則得正數。

BLOCK 迴路


BLOCK
(註一)
Block Number
N on Stack
(註二)
N equal to
Block Number
Pointed by PREV?
(註三)
Yes
No
Examine BLOCK
Number in Next
Disc Buffer
(註四)
N Equal to
Block Number
Pointed to
by PREV?
(註五)
Yes
No
All Disc Buffers Scanned
None contained Block N
Write Buffer contents to Disc
if Buffer was UPDATED
Read Block N from Disc to
this Buffer
Store N in BLOCK Number Cell
(Head of Disc Buffer)
(註六)
N BLOCK Number
in this Buffer
(註七)
No
Yes
Put Buffer Data
Address on Stack
(註八)
RETURN
(註九)

註一：將磁碟檔區塊(BLOCK)爲 N 的緩衝區地址，留在堆疊上。如果此區塊尚未在緩衝區中則以 LRU (最近最少用的頁取代策略)方式，將目前所使用檔案的區塊讀入緩衝區中。

註二：區段號碼 N 在堆疊上。

註三： N 是否等於 PREV 所指的區段號碼。
PREV : 變數所包含的磁碟緩衝地址爲最常爲引用者。UPDATE 命令會標記此緩衝區的地址，以便在稍後寫到磁碟中。
因此，利用 PREV 來檢查區塊 N 是否已在該磁碟緩衝區中。

註四：檢查此號碼區段是否在下一個磁碟緩衝區中。
由於 N 不等於 PREV 所指的區段號碼，所以，作進一步的檢查。看看下一個磁碟緩衝區中，區段 N 是否在此磁碟緩衝區之中。

註五： N 是否等於 PREV 所指的區段號碼。
如上述一樣，先檢查區段 N 是不是等於 PREV 所指的區段號碼。

註六：掃描所有磁碟緩衝區有沒有區段包含 N ，如果緩衝區是已更新的便將緩衝區的內容寫入磁片。讀出來自磁碟區段 N 到此緩衝區儲存 N 在區段號碼的小室(此小室在磁碟緩衝區頭)。

註七： N 區段號碼在此緩衝區嗎? 。

註八：將資料緩衝區的起始放在堆疊上。

註九：返回呼叫的地方。

附錄(o)-以 forth 觀點看 80486cpu

鑽研 FORTH 十年了，其間一直對 80x86 的 CPU 沒有好感，而偏好 68000 的 CPU 結構，這似乎是每一個 FORTH 的玩家都有的通性。因 8086 系列的單一堆疊，分段式記憶體，及由小到大的順序存放值的內容(Small Ending)方式，再加上特定用途的暫存器與繁多又難記的指令集，都與 FORTH 虛擬 CPU 的觀點背道而馳。雖然原則上，一個語言(如果 FORTH 可憐到只是一個語言，而不是人機合一的道)應該是與硬體結構無關的。但似乎每一個使用 FORTH 的玩家，都必須要自己寫一套 FORTH 的系統才能算完成了對 FORTH 的修練。而此時對 CPU 結構的了解與抉擇，就變得與工作內容相關了。看官，別把這事看得多驚天動地或許您用了半生的 Clipper 或 C 語言，甚至組合語言，但絕不會異想天開痴人說夢的想要自己寫一個 Compiler 或 O.S. 對不！這種想「創造」的悲哀！似乎是 FORTH 的本質也是 FORTH 玩家無法逃避的命運。

言歸正傳吧！在 80486 之前我一直不曾認真的思考 FORTH 在 80x86 系列上的最佳結構方式，但是面對日夜爲伍的 PC ，總不能「怨嘆一生」，到頭來總要建一套好用的 FORTH 才能解決困境。首先 K 了 FORTH 高手 Ray Duncon 所著 Powor Progromming with Microsoft MACRO Assombler；發現 80486 是一個夢寐以求的雙堆疊架構 CPU ，其擁有 160 多個指令集，12 種資料型態 8 個一般用途的暫存器，6 個段暫存器(每段可長達 4G)與 8 層深度的浮點運算堆疊暫存器(每層有 80 位元的精密度)。如此一來，FORTH 的雙堆疊架構已混然天成了。更好的是此浮點堆疊，無論整數 64 位元或浮點 64 位元，皆以相同的格式 80 位元儲存在浮點堆疊中，所以在浮點堆疊中無論加、減、乘、除，均不需要轉型(Coercion)僅需在存取(LOAD/STORE)記憶體的過程中，說明資料來源或目的的型態就大功告成了。

尤其最感動人的部份是在浮點堆疊的指令中，包含了 FORTH 最常用的堆疊操作指令如：

FORTH	80486
DUP	FLD ST(0)
DROP	FSTP ST(0)
SWAP	FXCH ST(1)
OVER	FLD ST(1)
ROT	FXCH ST(1)
	FXCH ST(2)
R@	FLD qword ptr [ESP]
>R	SUB ESP，#8

```
                    FSTP qword ptr [ESP]
R>                  ADD ESP，#8
                    FLD  qword ptr [ESP-8]
DUP >R              SUB ESP，#8
                    FST  qword ptr [ESP]
```

除此之外，浮點堆疊也提供了一些已最佳化的運算指令如：

```
OVER +              FADD ST( 1 )
DUP -ROT + SWAP     FADD ST( 1 )，ST
n pick +            FADD ST( n )
SWAP -              FSUB
OVER -              FSUB ST( 1 )  .......等，
```

在此僅列舉一、二如看官對 80486 有更進一步的濃厚興趣可以參考 Ross P. Nelson 所著的 "Microsoft's 80486 Programming Guide" 到目前爲止，我們已知道了 80486 的一些特性，接下來便是對 FORTH 系統的規劃了。

首先，我強調不要對記憶體的用量或 CPU 的執行速度作過多不必要的考慮，因爲硬體的速度每年增快，價格每年降低。再者，我堅持使用 80486 完整的硬體功能，如 32 位元的連續定址空間與 64 位元的 IEEE 854 浮點標準格式。由此，可以了解我所規劃的是一個以浮點堆疊爲參數堆疊，以 ESP 暫存器所指向的位置爲回返堆疊的巨集副程式線串碼結構(Floating Number STACK Macro _ Subrontine Threaded Cade)。以有效的利用參數堆疊的硬體指令以及回返堆疊與參數堆疊可同時平行運作的能力來架構出 80486 CPU 所具有的「天生 FORTH」的特色。

當然，頭(Symbol Table)與身(Code and Data)是分開的，頭在 FS 段，身在 CS・DS，ES・SS 段，DOS 的定址空間在 GS 段。

接著，我簡化了使用者可宣告的資料型態爲：

(1)數字(Number)(64 位元浮點)

(2)指標(Pointer)(32 位元的整數)

(3)字串(String)(65535 最大長度的 Unicode 字串，此種字串可涵蓋任一種國家的文字)。

或許，這種規劃與傳統的 FORTH 整數哲學相衝突，但相信我 FORTH 除了即時系統(Reoll TimeSystem)與自動控制(Embeded Control)之外，仍可以在大型整合系統上有所發揮。

還有，我認爲在傳統的 FORTH 系統中，編/直譯的工作太過輕鬆，也許，這本是 FORTH 的優點，但隨著時代的進步，似乎一個太過簡單的編譯結構，會讓使用者必須要兼顧許多最佳化的細節，而忽略了解決問題本身的演算法(Algoritlm)才是最需要投入心力的地方。當然，如果對效率的要求，仍是應用軟體的成敗關鍵(如訊號處理 FFT，或繪圖及影像的高速運算等)，則保留組譯程式(Assembler)的功能在 FORTH 的系統中，便足以應付了。

而要加強 FORTH 的編譯能力明顯的就必須大刀闊斧的增添頭部(Symbol Table)的訊息內容，例如副程式的堆疊進出變化量；巨集的長度及可代替參數項；資料的類別(Class)描敘表格與物件導向的遺傳能力。

最後，也最不可忽略的是 FORTH 的交談本質和無拘無束自由發揮「創造」的寫作環境。例如：螢幕上的動態堆疊顯示，對程式的撰寫與線上除錯，是助益最大的了。以上是我目前的研究心得，與專案計劃，如果您不同意，當然，可以重新規劃；每個人對自己的 FORTH 系統都是「創造」者。

有時想來好笑！工研院與資策會比 Micro Soft 的人員還多，爲什麼就弄不出個 Window-NT，是人材條件差嗎? 我想中國人如此「聰明」應不致如此吧！是沒有錢嗎？不是每年都報銷數十億元，花到那去了呢? 其時，問題所在是心理障礙。好的軟體是被「創造」出來的，而不是人多、錢多就可以「製造」出來的。如果，每個在台灣的電腦工程師、音樂作曲者、編劇導演們都喜歡「創造」新的東西。而厭惡抄襲陳舊的他人產品。那二十一世就是中國的「版權」霸主時代了，對不。看官們，怎麼樣，也寫一個自己的 FORTH 系統玩玩如何? 反正日日抱怨，天天懊悔都無補於事總要開始「創造」，才能豐收嘛！容我一笑！

說個悲哀的笑話做爲結尾吧！在教 Compiler 的課程中，坐著五、六十名資訊系四年級的大學生，問他們：「台灣有可能再度成爲 PC 的輸出王國嗎！」約有三十名學生舉手同意！再問「台灣有可能成功的仿製超級任天堂遊戲機嗎！」全部的學生充滿自信的舉手。又問「台灣可能繪製如忍者龜或製作一個如瑪麗歐那樣成功且風瀰全球的故事或人物嗎？」學生們你看我，我看你但沒有人舉手。我最後問：「台灣能否有一個軟體公司能設計出如 Borland C++ 或 Turbo C 一樣風行全世界的 Compiler 呢? 」台下一陣吃吃的笑聲不斷，一隻手也沒舉起。看官們，如果您與整個社會的感覺都與這些大學生一樣。那您的心中就已然知曉，爲什麼，台灣永遠是「版權」的受害者而不是獲利者了。

附記：

看官們，可曾剖析過 Quick Basic，知道嗎? 其核心部份就是一個 FORTH 啊！

----1993／5／5 子夜----

本文之參考書目

(1) Intel "80387 Programmer's Reference Mannal"
1987 ISNB 1-55512-057-1
(2) Richard Startz "80*87 Applications and Programming"
1983 ISBN 0-89030-420-7
(3) Ray Duncan "Power Programming with Microsoft Macro Assembler"
1992 ISBN 1-55615-256-6
(4) Penn Brumm "80486 Programming" 1991 ISBN 0-8306-3577-7
(5) Ross P. Nelson "Microsoft's 80486 Prgramming Guide"
1991 ISBN 1-55615-343-0
(6) Fernandez Ashley "Assembly Langnage Programming for the 80386"
1991 ISBN 0-07-100717-2
(7) Barry B. Brey "The Intel Microprocessors 80*86 Architecture,
Programming , and Interrfacing" Socond Edifion
1991 ISBN 0-02-946322-4
(8) Ray Duncan "Extending DOS" Sencond Edition
1991 ISBN 0-201-56798-9
(9) Redington,Dana "STACK-Oriented Co-Processors and FORTH"
FORTH Dimensions 1983 Sep

附錄(p)-FORTH 資料庫程式設計

-------本文節錄 江明鑑先生 編著之 FORTH 入門，是 FIG-FORTH 版本，若要在 F-PC 上執行要修改成循序擋-----------

金城 註疏：

一般人對資料庫的設計有幾個嚴重的誤解：

第一點：認爲某一個套裝軟體就是資料庫，所以學資料庫就變成了學 DBASE Ⅳ、FoxBASE 或 Clipper 真是荒謬到了極點。就似學電腦與學 BASIC 之間一樣的誤解。別忘了資料庫的難以設計，是在於對整體資料的分析、各類表格的設定、資料字典的整理、與對人機界面的整體規劃。尤其是關聯式的資料庫(RELATIONAL DATABASE)本身就是一套精準的數學模式。決不是用了幾年 DBASE 或 CLIPPER 的老鳥或阿貓阿狗就能心神體會的。那是需要下功夫花時間在資料庫本身的研究上才能有所成就的。否則就成了自欺欺人的騙吃騙喝了。

第二點：認爲選對了一套工具軟體，就成功了一半。再找到了一位有數年經驗的程式設計師，那又成功了另一半。天啊! 你可曾聽人說：「選某一品牌的水泥，再找一個有數年經驗泥水匠，就可以蓋大樓造大廈了」。如果此句話是錯的，那錯在那兒? 缺少了一位合格有能力的建築師來規劃、來設計、來評估、來監督。大家都忘了，泥水匠的工作是依圖施工。如果沒有圖，那就無法依循，成了「亂作」了，所以資料庫的成敗關鍵人物是資料管理師，而不是程式設計師啊。

第三點：最爲可笑，許多人用來評估資料庫好壞的標準竟是：「CLIPPER 所寫的資料庫，比用 DBASE 寫的較好」「用甲牌的水泥所蓋的房子，一定比用乙牌的水泥所蓋的房子要更美麗，更舒適」，我想這兩個句子犯了同一個錯誤! 是不是? 工具只是增加方便性，除此之外就無關緊要了。所有的套裝工具本質上是大同小異的，就似所有的語言在功能之雖有差異，但在本質上都要轉成機械碼才能讓 CPU 執行一樣。要評估一個資料庫的好壞，與評比一群資料庫發展工具的優劣是兩碼事。評估資料庫，要點是在整體架構是否完整，例外的處理、與資料庫的一致性是否顧及、其人機的使用者界面是否友善易用....等。不是嗎?

這些年來資料庫的設計和應用是電腦最主要的用途，FORTH 是發展各型實用資料庫最理想的語言，比其他高階或低階語言都有許多優點，它滿足了交談性及提供程式自我說明等特性，所以適合於發展新的程式系統。用 FORTH 寫作商用資料庫之迅速及便利，值得我們花一些時間去學習這種奇特的語言。

我曾用 FORTH 寫了一些小型的資料庫，包括 8000 個記錄(Records)每個記錄有 128 個字元(byte)，這些不同用處的資料庫，加上它們特殊的輸入輸出程式，通常只花我幾小時或一個晚上的時間便建成功了，我以前也用過其他語言來做類似的工作，但用過 FORTH 以後，我是絕對不會再回去用那些語言了。

本文將討論一組 FORTH 的指令集，可以作爲一個資料庫的基礎，我並加進一個簡單的例子用這些指令來建造一個小型資料庫。這個指令集原來是 Bill Ragsdale (FORTH Interest Group 的創始人)在他們一次討論會上提出的。我們將他的指令集稍加增益，將其擴大而可以來建立各式記帳用的資料庫。我希望讀者對 FIG-RORTH 的語言模式及它的基本指令集已稍有涉獵，這對本文的研讀有很大的幫助。

這個資料庫的指令都顯示在本文後。一共有八幕的程式，每一幕有 1024 字元，分爲 64 字一行，每幕有 16 行，第 21 幕是一些使用的指示文字。正式的程式是由 22 幕到 28 幕爲止。

在第 22 及 23 幕中有八個基本的指令，構成資料庫指令集的基礎，每個指令的名稱以及在括弧中的註解，應可相當清楚地說明各個指令的功能，最先的兩個指令是資料庫中須要用的兩個變數 :REC# 儲存現在處理的記錄的號碼，OPEN 則含有現在處理的檔案的起始位置。

2@ 及 D@ 是相同的指令，它們的功用是由堆疊最上存的地址處，抓出一個 32 位元的雙整數(Double Integer)，再將此雙整數堆上堆疊。LAYOUT 則是將新開檔案的兩個主要參數堆上堆疊以供後續指令使用。READ 是我們將常常用的指令，它將記錄號碼自堆疊上取出，檢查它是否在檔案的記錄範圍中，再將之存入 REC# 而把這個記錄做爲現在要處理的記錄，RECORD 則由堆疊上將記錄號碼取出而將這個記錄在記憶中的地址放置在堆疊上，最後，ADDRESS 則將存在 REC# 中的記錄號碼取出，再用 RECORD 找出這個記錄在記憶中的地址，還置於堆疊之上。

金城 註疏：

如果使用 F-PC 或 32 位元的 FORTH 系統，則 2@、D@、D*、D/、D/MOD 等雙倍精密度整數的詞可以不用再定義了。因爲 F-PC 已經有了，而

32 位元的 FORTH 則用不著了，因爲整個系統都是以 32 位元來定義的，其 @、!、+、-、*、/MOD 均是 32 位元的。另外在傳統的 FORTH 中由於磁碟機的存取是 FORTH 自己管的，所以爲方便起見規劃成 1K(1024 Byte)大小的存取單位。無論程式或資料都是以 1K 爲存取的基本單位，但在 F-PC 中由於磁碟機是透過 DOS 來管理，所以改成了循序檔案，而存取的單位便不再限定成 1K 的大小，可以由使用者自己來定一筆資料錄的 Byte 數，使其更有彈性。但也喪失了 FORTH 傳統中的 BLOCK 觀念，有利有弊。許多 FORTH 的老手仍喜歡 BLOCK 的觀念，一來單純，二來有虛擬記憶體的優點。

第 23 幕上的三個指令叫做"定義指令"(Defining instruction)因爲它們是用來定義不同名稱的各種檔案以及在不同檔案中各個記錄欄(Field)的名稱。這些定義指令使用 FORTH 中的特殊指令 <BUILDS 及 DOES> ，要了解這兩個特殊指令的用法及功能是要花一些時間的，我們在此詳細地解釋一下 24 幕上的三個指令，或許可以幫助讀者來欣賞這兩個特殊指令的功能。

在一個記錄中有兩類記錄欄：一類是文字欄，另一類是數值資料欄，第 23 幕上第一個定義指令 DFIELD 是用來在記錄中加一個資料欄的，資料欄的長度(位元單元)是取自堆疊的。DFIELD 給這個資料欄一個名字。當我們以後用這個名字作爲指令使用時，這個指令即會將在現用記錄 (其號碼存在 REC#中)中這一資料欄的地址還置在堆疊上，所以這個指令是用來指出這個資料欄在記憶中的地址，以備我們由此欄中讀取資料或存入新資料。

TFIELD 則是用來在記錄中加一個文字欄的。這個文字欄的長度(字元單位)也是取自堆疊上，文字欄的名字也是由 TFIELD 來定義的。當以後用文字欄的名字作爲指令用時，這個文字欄指令會將文字欄的地址以及文字欄的長度都堆上堆疊。文字欄的長度是給 TYPE 這個指令來將文字欄中的文字印出來用的。因爲資料欄並不須要長度的資料，所以它和文字欄的性質有顯著的差異，故此我們須要 TFIELD 及 DFIELD 兩個不同的定義指令來分別處理這兩類的記錄欄。

FILE 則是用來定義不同的檔案用的。FILE 由堆疊上取得其中各個記錄的總長度(字元單位)，檔案中記錄的總數，以及檔案在磁碟上的位置(塊數或 BLOCK 數)。當以後我們用檔案的名字作爲指令時，這個檔案的兩個主要參數的地址就會被存到 OPEN 裡去，使得以後存取記錄時都由這個檔案中存取。

有了這兩幕上的這些指令，我們就可以來建造各種的檔案了。在此之前，我要提醒讀者注意其使用的 FORTH 系統的一些特性。Fig-FORTH 的系統以 128 字元爲一個 BLOCK ，在這種系統中記錄的長度就不能超過 128 字元。最理想的記

錄長度是可以整除 128 的數字，如果我們定義的檔案記錄的長度是 70 字元，這個檔案是可以操作，但每加一個記錄我們就浪費了 58 個字元，如果 BLOCK 的長度是 1024 字元就比較理想了，只是我們表列的程式是爲 128 字元的系統設計並測試成功的。另外我們還須要一些輔助指令，例如一些雙整數的算術指令，日期的換算，元、角、分數值的換算等這些指令是存在第 24 至 26 幕中的。

第 24 幕上的雙整數指令是用來幫助日期及錢幣換算等操作用的，它們只是將 FIG -FORTH 系統的原有指令稍加延伸而成的，不須要詳細的解說。我們應注意 D*、D/ 及 D/MOD 三個指令是單雙整數混合的操作，其順序須要分清。

第 25 幕上日期換算的指令是很簡單的，日期在記憶中是換算成一個整數存放的，以後我有時間的時候我會再寫一個程式，把日期換算成爲朱里安曆的日期，這樣可以比較方便地計算日期的差異而可以準確地計算利息。目前的指令只是將日期由 MM/DD/YY 的型式換成整數儲存(用?DATE 指令)，並將內存日期以上述的方式列印出來，(用.DATE 指令)。這個方法的好處是每個日期在記憶中只佔兩個字元或 16 位元，?DATE 是一個用 WORD 及 NUMBER 等基本指令來將輸入的日期資料解碼成數目字的好例子。當使用者輸入 MM/DD/YY 的日期後，這個字串被解爲三個雙整數，堆在堆疊上。 DATEBIT 指令即將此三個雙整數減化爲 16 位元。.DATE 則將此 16 位元的數目還原成爲字串日期列印出來。

第 26 幕是執行錢幣換算的一些指令。錢數在輸入後是轉換成 32 位元的雙整數儲存的，在儲存前還要處理元角之間的小數點。在 FIG-FORTH 系統中，一個數目字輸入時若帶了小數點就會被解碼成一個 32 位元的雙整數，小數點的位置則存在 DPL 之中，利用 DPL 中存的位數，我們可以將得到雙整數移位而得到正確的錢數，不管小數點的位置放在那裡。處理小數點是用一個 CASE 的結構來完成的。

OSCALE 是處理沒有小數點的數字，這個數字應當乘以 100 且變爲雙整數。1SCALE 則應乘 10，2SCALE 則乘 1 就好了。假若小數點超過兩位，則輸入必有錯誤，就應該印出警告字訊。在每種情形都有處理的指令後，我們就將其解碼欄地址(Code fie ld Aadress 或 CFA)存在 NSCALE 變數的參數欄中。SCALE 則是最後的指令，由 DPL 中找出小數點的位數而去 NSCALE 中找到合適的 CFA 去執行錢數的換算。

?$AMOUNT 則是請使用人輸入錢數，換算成雙整數堆上堆疊的指令。
.$AMOUNT 則是將錢數列印出來的指令，印在八格的空欄中，右邊對齊。也許有別的更好的方法來建立資料庫，但這幾幕上的指令是足夠我們用的了，這些指令之能建立各式的資料庫可以充分地證實 FORTH 語言系統驚人的功能及可塑性。

有了這五幕的指令集，我們就可以很快地建立各種不同的檔案系統，每種檔案都可隨客戶的需要而有不同的特性。在第 27 幕中的例子中,我在建立一個小型檔案，其中每個記錄中有四欄：第一欄 TAG 有 2 字元:第二欄是文字欄，NAME 有 30 字元長；第三欄是日期欄，2 字元；第四欄是金額欄，4 字元。這個簡單的資料庫看來並沒有多大用處，但供給我們足夠的例子解釋資料庫的結構及其輸入輸出法，最後在第 28 幕中還有一些附屬的指令用來清除記錄，輸入新記錄及印出整個檔案的內容。

這個檔案的名稱是 DEMO ，其內各記錄的資料欄及文字欄的建立是先將一個 0 放在堆疊上，這是每個欄與記錄間頭的開端(以字元爲單位)。第一欄叫做 TAG ，是資料欄，長度是 2 字元，這一欄是準備儲存一些檔案的系統資料的。這一欄建好後，堆疊上的 0 即變爲 2 ，就可以用來指示下一欄的起始地址。第二欄是文字欄，名稱是 NAME ，長度是 30 字元，在堆疊上留下 32 (TAG 欄與 NAME 欄的總長度)。第三欄是 DAY ，是資料欄，長度是 2 字元，因爲日期是換算成了一個 16 位元的數值，最後一欄叫 DOLLAR，也是資料欄，但是長度是 4 字元因爲金額值是換算成 32 位元的雙整數的，整個記錄，即包括這四欄，記錄的總長度是 38 字元留在堆疊上。

金城 註疏：

資料庫的基本概念是建立資料表格(TABLE)也就是俗稱的檔案(File)。其中的每一列(ROW)稱之爲記錄(Record)，每一行(Column)稱之爲欄位(Field)。先將所有必需的資料，依其相關的程度，分置在各個獨立的表格中。然後再加上對各種資料內容的搜尋(Search)，更新(update)，建立(Append)，和刪除(Delete)等基本功能。然後再製作一些顯示及列表等高階的功能，當然最後以一友善的使用者介面將之整合成一體的應用環境。便略俱雛形了。但是若要增加查詢的效率，則索引(INDEX)，雜湊(UASH)。 B-TREE 等高級技巧就必需要加入了。

下面我們就用 FILE 這個定義指令來建立 DEMO 檔案。FILE 除了須要記錄的總長度(38 字元)以外，還需要兩個參數。在檔案中記錄的總數(200)，及檔案開始的磁碟地址(400)。

```
32 200 400 FILE DEMO
```

即將 DEMO 檔案建成了。我們現在只要將所要用的記錄號碼存入 REC#中，再呼叫 NAME ，DAY ，DOLLAR 等指令即可以將這個記錄相關各欄的地址叫到堆疊上，有了這些地址我們便可以很方便地檢視其內容或更改其內容了。

在 FORTH 系統中" ! " 是存放數據資料的指令，而 "."是列印資料的指令。遵循這個慣例，我們定義了一些檔案操作基本指令： !NAME 是將一個名字存入 NAME 文字欄的指令，.NAME 則是將此欄文字列印出來的指令。 !DAY 將日期存入 DAY 欄； .DAY 則將日期印出。 !DOLLAR 將金額存入 DOLLAR 欄而；.DOLLAR 則印出金額數值，這些即是全部資料庫所須的輸入及輸出指令了。

當我們將一些資料存入某一記錄中後，經常應該去檢查存進去的資料是否正確。我們可以使用 FORTH 系統中的 DUMP 指令將整個記錄中的資料列印出來，只要先用 READ 及 ADDRESS 將這個紀錄的開始地址找出來就好了，但是 DUMP 印出來的資料是每個字元的值不容易判讀其內容，因此我們另行定義一個。REC 的指令。 .REC 將整個記錄的內容用很整齊的方法印出來供使用者檢查，這個指令可以加入許多花樣，使得列出的資料有不同的顯示方式或是列印在終端機的固定位置上。這些花樣留待讀者自己去發展。

最後在第 28 幕上的是三個使用者所須要的三個指令 CLEAR.DATA 可以將整個檔案清除以備輸入新資料； INPUT 則是用來由使用者輸入一個記錄的資料， OUTPUT 則是將整個檔案中的資料全部列印出來。這三個指令即是使用者所必須學會使用的，對使用人來說其他的系統都不必知道，因此學習使用這個系統是相當方便的事。

CLEAR.DATA 並不清除檔案中的現存資料，它只是將第 0 個記錄清存。原因是我使用第 0 個記錄儲存一些系統用的變數。在這個記錄的 TAG 欄中存的是檔案裡最後一個記錄碼，表示檔案中記錄的總數。如果用 CLEAR.DATA 將這個記錄清除爲 0 ，TAG 中也是 0 表示檔案中沒有記錄。如果這時我們用 OUTPUT 來列印，就會得到 "EMPTYFILE" 的訊息。如果我們開始用 INPUT 加入新的記錄，這個 TAG 欄中的數目會先加 1，加 1 後的數字也存入 REC# 中，這時就可以開始輸入第一個記錄，以後類堆。

在 INPUT 的指令中，電腦會印出一些文字，指引使用者輸入正確的資料，然後用 !NAME ，!DAY 及 !DOLLAR 等指令將輸入的資料存入現用的記錄中 UPDATE 及 FLUSH 兩指令則是在資料輸入後將記錄抄錄到磁碟上作長八的貯存。當記錄輸入完畢時，並由 .REC 將輸入的資料按著設計的格式印出來備使用者檢查其正確性。DUTPUT 則先取出第 0 號記錄 TAG 欄中的數目，然後再設定一個 DO-LOOP 將檔案中所有的記錄依次列印出來。

總之，這是一個很簡單的資料庫系統，但其中所採用的原理及方法，卻可以隨使用者或客戶的須要而修正或擴充。我的目的乃是說明這個基本的 FORTH 指令集的構成並解釋這些指令如何工作而達成其使命，其中有很多例子顯示在 FORTH 語言系統中各種型式的資料如文字資料及數值資料應該如何處理，這些討

論對有經驗的程式師而言可能是十分膚淺的，但對初學 FORTH 或是初次接觸到資料庫的人，卻是很好的學習材料，我們希望這些讀者會覺得本文對他有所助益，這個資料庫系統最大的好處，是它可以很方便地修正改變，以適合各人的特殊需要所有的修改都是可以用互答式的操作來進行的，由此發展其他專門的資料庫，不過是幾小時的工夫而已。

這套指令集是基於 FIG-FORTH 的指令集寫成的，其中有少數指令與 FORTH 的 79 標準稍有出入，但要將它改變成爲 79 標準的指令也是很方便的事，這個系統原來是在 Heathkit H89 微電腦上發展出來的，發展成功之後我又將它轉建到 CP/M 的 FIG-FORTH 系統上，現在發表的就是這個 CP/M 上的程式，使用 128BYTES/BLOCK 的貯存方式，如果讀者有 CP/M 的 FIG-FORTH 系統，只要將原程式輸入即可操作了。

金城 註疏：

目前資料庫的發展隨著網路架構的成熟，近年來有了許多新的觀念，除了傳統的線上交易處理(OLTP)系統外，還加上了主從式(Client/Server)資料庫與分散式(Distributed)資料庫、物件導向(Object Oriented)資料庫、多媒體(Multimedia)資料庫、與知識庫(Knowledge BASE)等的蓬勃發展。而結構化詢問式語言(SQL)也有了國際標準。甚至 Microsoft 的新作業系統 Windows-NT，也在其中加上了 SQL 的部份，所以如果看官對資料庫的應用真的感興趣，那可以多花點心思研究一下。請記住在電腦的世界中沒有速食麵，只有紮下根基才是正途。

```
SCR # 21
 0  ( SCREEN 21: CDMMENTS AND LOAD FOR DEMO DATA FILE )
 1  27 EMIT 69 EMIT CR CR CR CR CR
 2  ." This demonstration data system provides a pattern for the "
 3  CR ." further development of any type of data base. The basic "
 4  CR ." file formating definitions are on Screens 22 and 23. Some
 5  CR ." utilities are on 24, 25, and 26. The demo file "
 6  CR ." definitions are on 27. Elementary file maniqulation "
 7  CR ." utilities are on 28. This model should get you started. "
 8  ; PROCEDE CR CR ." ENTER 'Y' TO LOAD SCREENS " KEY 89 = IF
 9         29 22 DO I LOAD LOOP ENDIF ;
10  PROCEDE
11  ;S
12
```

```
13
14
15

SCR # 22
 0  CR ." SCREEN 22: FILE DEVELOPMENT "
 1  0 VARIABLE REC# ( holds the current record numberm )
 2  0 VARIABLE OPEN ( points to current file descriptor )
 3
 4  : 2@ DUP 2 + @ SWAP @ ; ( double fetch )
 5
 6  : LAYOUT ( leave bytes/record-2, and bytes/block-1 )
 7    OPEN @ 4 + 2@ ;
 8  : READ ( n-th record, on stack,is made  current )
 9    0 MAZ DUP OPEN @ 2 + @ <   =
10    IF."FILE ERROR "QUIT THEN REC# ! ;
11    RECORD   ( leave address of n-th record )
12    LAYOUT */MOD OPEN @ @ + BLOCK + ;
13  : ADDRESS   ( leave address of the current record )
14    REC# @ RECORD ;
15  ; S
```

金城 註疏： 在 F-PC 中 2@ 以經有了。

```
SCR # 23
 0  CR ." SCREEN 23: FILE DEVELOPMENT - 2 "
 1  : TFIELD   <BUILDS ( create a text field )
 2     OVER, DUP, + ( leaves file count for this definition )
 3            DOES> ( leaves: addr, count )
 4    2@ ADDRESS + SWAP ;
 5  : DFIELD  <BUILDS    ( create a data field )
 6    OVER , +           ( leaves file count for this definition )
 7            DOES>      ( leaves address )
 8    @ ADDRESS + ;
 9  : FILE               ( create a named storage allocation )
10     <BUILDS           ( origin block )
11       1+ ,            ( number of records in file )
12       DUP B/BUF OVER / * , ( #bytes per block )
13       ,               ( #bytes per record )
```

```
14       DOES>  OPEN ! ; ( when file name used, point to )
15  ; S                  ( its decriptor parameters. )

MSG # 15
OK
```

金城 註疏： 在 Fig-FORTH 中的 <BUILDS 就是 F-PC 的 CREATE 。

```
SCR # 24
 0   CR ." SCREEN 24: DOUBLE PRECISION ARITHMETIC "
 1  : D/    (d, u -- d )
 2      SWAP OVER /MOD   >R SWAP U/ SWAP DORP R> ;
 3  : D*     ( d, u -- d )
 4      DUP ROT * ROT ROT U* ROT + ;
 5  : D/MOD  ( d, u -- r, g )       U/ ;
 6  : D@     ( ADDR -- D )          DUP 2 + @ SWAP @ ;
 7  : D!     ( d, addr -- )         DUP ROT SWAP ! 2 + ! ;
 8  : DOUP   ( d -- d, d )          2DUP ;
 9  : DSWAP  ( d1, d2, -- d2, d1 )  ROT   >R ROT R> ;
10  : DDROP  ( d1, d2, --  )       DROP DROP ;
11  ; S
12
13
14
15

SCR # 25
 0   CR ." SCREEN 25: DATE COMPRESSION AND EXPANSION "
 1  : DATEBIT ( converts date inqut to 2 bytes )
 2       32 D* D+ 13 D* D+ DROP ;
 3  : ?DATE    ." ( MM/DD/YY ) "
 4       QUERY 47 WORD HERE NUMBER 47 WORD HERE NUMBER BL WORD
 5       HERE NUMBER DATEBIT ;
 6  : .DATE      ( formats 2 byte date code and prints )
 7       0 13 U/ 32 /MOD ROT 100 * ROT + 0 100 D* ROT 0 D+
 8        <# # # 47 HOLD # # 47 HOLD # # #>   TYPE ;
 9    ;S
10
11
```

```
12
13
14
15

SCR # 26
 0    CR ." SCPEEN 26: ?SAMOUNT AND .$AMOUNT "
 1    ( define action for each scale case )
 2  : 0scale 100 D* ;  : 1SCALE 10 D* ;  : 2SCALE ;
 3  : 3SCALE ." INPUT ERROR " CR ;
 4    ( define scale case and extend for each with 'CFA' )
 5  ' 0SCALE CFA VARIABLE NSCALE ' 1SCALE CFA , 2SCALE CFA,
 6    , 3 SCALE CFA ,
 7    ( scale double precision value according to 'DPL' )
 8  : SCALE DPL @ 3 MIN 2 * NSCALE + EXECUTE ;
 9   ( wait for decimal value and scale it - leave value on stack )
10  : ?$AMOUNT QUERY BL WORD HERE NUMBER SCALE ;
11    ( print d from stack as $ and right justify in 8 spaces )
12  : .$AMOUNT
13    DUP ROT ROT DABS  <# # # 46 HOLD #S SIGN #>
14    36 EMIT DUP 8 SWAP - SPACES TYPE ;
15  ;S

MSG # 15
OK

SCR # 27
 0    CR ." SCREEN 27: DEMO FILE - RECORD GRNERATION "
 1  0  2  DFIELD TAG     ( a tag )
 2    30  TFIELD NAME    ( item name )
 3     2  DFIELD DAY     ( the date )
 4     4  DFIELD DOLLAR  ( a dollar amount )
 5      200 ( number of records ) 400 (starting block )
 6      FILE DEMO
 7  : !NAME (wait for name then store it in record )
 8      NAME DROP 30 32 FILL QUERY 1 TEXT PAD COUNT
 9      NAME ROT MIN CMOVE UPDATE ;
10  : .NAME  ( print name field ) NAME TYPE ;
11           ( the rest follow in the same way )
```

```
12  : !DAY ?DATE DAY ! UPDATE ; : .DAY DAY @ .DATE ;
13  : !DOLLAR ?$AMOUNT DOLLAR D! UPDATE ;
14  : .DOLLAR DOLLAR D@ .$AMOUNT ;
15  : .REC CR REC# @ 3 .R 2 SPACES .NAME .DAY 2 SPACES .DOLLAR ;

SCR # 28
 0     CR ." SCREEN 28: DEMO FILE - CLEAR.DATA, INPUT OUTPUT "
 1      ( clear especially tag in the 0 record in file )
 2   : CLEAR.DATA 0 READ TAG 128 0 FILL UPDATE ;
 3      ( example of formatting for inqut )
 4   : INPUT 0 READ TAG @ 1 + UPDATE DUP TAG ! READ
 5        CR CR ." ENTER NAME                          " !NAME
 6        CR ." ENTER DATE             " !DAY ( has a format prompt )
 7        CR ." ENTER AMOUNT                         " !DOLLAR
 8        .REC FLUSH ;   ( save this record on disk )
 9     ( list files 1 through the number in TAG of 0 record )
10   : OUTPUT 0 READ TAG @ DUP 0 = IF CR CR. " EMPTY FILE "
11        DROP ELSE 1 + 1 DO FORTH I READ .REC LOOP ENDIF CR CR ;
12   ;S
13
14
15
```

金城 註疏：

一、在傳統的 FORTH 中 1K(1024 Byte)是連續的 1024 個 Byte ，但爲了撰寫與列表方便，通常都會以 16 列，每列 64 個字爲一區塊。

二、這套資料庫的小程式要改寫成 F-PC 需要花點心血，你可以參考 " F-PC User's Manual " Version 3.5 1989 的 chapter 6 Ssguontial Files ，我相信如果功力夠，大概只要半天就夠了

三、在此列出幾本資料庫的參考書籍，供有志從是這方面的讀者，自修閱讀：

(一) An Introduction To DataBase Systen Volume 1 and Volume 2 fifth Edition C.J. Date 1990 , Addison Wesley

(二) Relational Datadase Writings C.J. Date 1990 , Addison Wesley

(三) File Structures Theory and Practice Panos E. Livadas 1990 , Prentice - Hall

附錄(q)-一九九一年全國 FORTH 程序員水平考試試卷

(時間： 180 分鐘、總分： 180 分)

注意事項：答題一律用藍色或黑色鋼筆或圓珠筆書寫，按規定寫在答案紙上，字跡清楚、工整。試卷評分均以 F-83 標準爲依據，試卷中未經說明的地方基底爲十進位制。

第一部分　計算機基礎知識(36 分)

一、計算(只計算出結果)

1. 十六進制數 1991 轉換成二進位數(2 分)

2. 十六進制數 ABE 轉換成十進位數 (2 分)

3. 十六進制數 BAD 轉換成八進位數(2 分)

二、選擇

1. 如果用 8 位元二進位小數表示十進位數 -0.875 (6 分)

   它的原碼應是：

   A. 0.1100000　B. 1.1110000　C. 1.0100000　D.0.0110000

   它的補數應是：

   A. 1.0010000　B. 0.0100000　C. 1.1100000　D. 0.1100000

   它的負碼應是：

   A. 1.0010000　B. 0.0100000　C. □.0001111　D. 0.0111111

2. 邏輯代數式 A(1+B) (2 分)

   A. A　B. 1　C. AB　D.AB

3. 某計算機的內存容量是 64KB，它的內存地址暫存器需要 (2 分)

   A. 15 位元　B. 14 位元　C. 16 位元　D. 17 位元

4. FORTH 系統內用來指出詞執行順序的是 (2 分)

   A.工作暫存器 W　　B.程式計數器 PC

   C.內部解釋指標 I　D.執行區域地址 CFA

5. 計算機系統中的 CPU 用來指出指令執行順序的是 (2 分)
   A. 累加器　　B. 程式計數器
   C. 內存位址暫存器　　D. 指令暫存器

6. 堆疊是一種線性串列，它的特點是 (2 分)
   A. 一個端點　B. 一個中間點　C. 先進先出　D. 先進後出

7. 爲另一台不同類型計算機產生目的 obj 碼的 Assembly(組譯程式)稱作 (2 分)
   A. 巨集組合程式　　B. 通用型組合程式
   C. 微組合程式　　D. 交叉組合程式

8. 計算機操作系統的作用是爲了提高計算機的 (2 分)
   A. 速度　B. 利用率　C. 靈活性　D. 相容性

9. 負責對作業或程序進行呼叫的是操作系統中的 (2 分)
   A. 主記憶體管理部份　B. 微處理機管理部份
   C. 處理機管理部份　　D. 分頁記憶體管理部份

10. 對系統中的資料進行管理的部分通常稱爲 (2 分)
   A. 資料系統　　B. 文件系統
   C. 索引系統　　D. 資料存儲系統

11. FORTH 系統有別於其它語言的顯著特點是 (2 分)
   A. 有作業系統的功能　　B. 有整合的開發環境
   C. 有堆疊的詞典的結構　D. 提高程式的可攜性

三、判斷對錯在括號中對正確者加" ∨ "對不正確者加" x " (4 分)

A. 沒有外部設備的計算機稱作赤裸機器 (　)

B. 把系統軟体中經常用到的部分改成硬體後能提高計算系統的效率 (　)

第二部分：FORTH 語言基礎 (90 分)

一、填充(25 分)

1. _____、_____、_____是國際上制定的三種 FORTH 標準。 (3 分)

2. 將下列式子 [(A-B)*C]/(D+E) 改爲後置式表示時

   A B — C _____ E _____ / (3 分)

3. 定義堆疊狀況 (1 2 3 4---2 1 4 3) 採用：

   SWAP ____ SWAP ____ (2 分)

4. 定義階乘：堆疊狀況 (n -- n!) n>1 採用：

   : N! _____ DUP ROT ?DO i * -1 +loop ; (1 分)

5. 定義詞 Y= ，計算$Y = ax^2 + bx + c$，
   其狀態爲
   ： Y= (x a b c -- y)
   >R 2 PICK * >R
   ____DUP * _____ (ax*x)
   >R >R + + ; (2 分)

6. 3DUP (n1 n2 n3 -- n1 n2 n3 n1 n2 n3)與
   _____ _____ _____ 相同(每空只能填一詞或數) (3 分)

7. 求出一 雙精度數 d 的絕對值：
   ： DABS (d --｜d｜)
   DUP 0<
   IF _____ THEN ; (1 分)

8. ： TEST 2 . 0 -
   IF 1 ELSE 2 THEN
   DUP . =
   IF 3 . ELSE 4 . THEN ;
   TEST <cr> 顯示： (1 分)
   _____

9. DECIMAL 16 BASE ! BASE @ . <cr> 顯示:
   _____ (1 分)

10. HEX
: NUM DECIMAL 10 HEX 10 + . ;
DECIMAL
NUM <cr> 顯示： (1 分)
_____

11. 32767 1+ .<cr> 顯示： (1 分)
_____

12. -1 1234 C! 1234 C@ . <cr> 顯示： (1 分)
_____

13. 2VARLABLE XY 3 5 XY 2! (1 分)
XY @ . <cr> 顯示： _____

14. 1 NOT . <cr> 顯示： (1 分)
_____

15. HERE . <cr> 顯示：
3000
: TEST 500 CR ;
HERE . <cr> 顯示：
_____ (1 分)

16. HERE . <cr> 顯示： 3000
1 , 2 c, HERE . <cr> 顯示：
_____ (1 分)

17. 下面是編譯詞 DO 的定義，但不完整，DO 應在進行後把(DO)編譯到詞典中，同時在堆疊上留出 HERE 和標示 3 ，將該編譯詞補充完整：

: DO COMPILE (DO) HERE 3 ; _____ (1 分)

二、選擇(25 分)

1. FORTH 的發明人是：
A. 英國人 B. 美國人 C. 瑞士人 D. 日本人

2. FORTH 起源於
A. 化工業 B. 汽車業 C. 天文科學 D. 輕工業

3. 中置式運算法 15/(7-2)的後置式運算爲
A. 15/(7-2) B. 15 7 / 2 -
C. - 7 2 /15 D. 15 7 2 - /

4. : TEST (ABC ."DEF") ;
TEST <cr> 顯示爲
A. ABC B. DEF C. ABCDEF D. 無

5. 程序注解中(n addr c — f) n 表示的是：
A. 16 位數 B. 地址 C. 字母 D. 邏輯旗號

f 表示的是：
A. 16 位數 B. 地址 C. 字母 D. 邏輯旗號

6. DECIMAL
: TEST 16 HEX 10 . . ;
TEST <cr> 顯示結果爲：
A. 16 10 B. A 16 C. A 10 D. 10 A

7. 在 F83 標準中，ud 所表示的數值範圍是：
A. 0 ～ 65535 B. -32768 ～ 32767
C. 0 ～ 4294967295 D. -32768 ～ 65535

8. 在 F83 標準中 addr 的表示的數值範圍是：
A. -32768 ～ 32767 B. -128 ～ 127
C. -32768 ～ 65535 D. 0 ～ 65535

9. 13 -7 MOD 結果爲
A. 5 B. -5 C. 1 D. -1

10. 0 -1 DABS D. 輸出
A. -1 B. 0 C. 65536 D. 65535

11. -18 5 / 結果爲：
A. 4 B. 3 C. -4 D. -3

12. 2 NOT 3 + -1 AND 結果爲：
A.-1 B. 0 C. 4 D. 1

13. 定義程序 (a b c d — d c b a)
A. ROT ROT ROT ROT
B. SWAP 2SWAP SWAP 2SWAP
C. SWAP 2SWAP SWAP DUP DROP
D. 2SWAP SWAP 2SWAP SWAP

14. : TEST CR
3 1 DO
6 4 DO J 0 .R I 0 .R 5 SPACES
LOOP CR
LOOP ;
TEST <cr> 輸出爲：

A. 14 15
24 25

B. 41 51
42 52

C. 14 15 16
24 25 26
34 35 36

D. 41 51 61
42 52 62
43 53 63

15. : TEST 0 1 5 DO I + -2 +LOOP . ; TEST 將顯示
A. 15 B. 14 C. 8 D. 9

16. : TEST 1000 =
IF KEY DROP EXIT
ELSE HEX THEN ;
該詞所佔代碼長度爲：
A. 33 字 B. 29 字 C. 31 字 D. 35 字

17. VARIABLE XX 4 ALLOT VARIABLE YY 0 ,
' YY >NAME ' XX >NAME — . <cr> 輸出爲：
A. 0 B. 9 C. 13 D. 11

18. CR 3 SPACES 32 EMIT OUT @ 0 .R OUT @ 0 .R <cr>輸出
A. 43 B. 34 C. 45 D. 54

19. HERE . <cr> 顯示 5000
HEX 10 ALLOT DECIMAL HERE . <cr> 輸出爲

A. 5016　　B. 5000　　C. 5010　　D. 5020

20. 在冒號定義中，能把一個立即詞編譯到詞典中去的是：
A. COMPILE　　B. [COMPILE]　　C.]　　D. IMMEDIATE

21. 欲建立一個詞 XXX 的詞頭，且建立後 DP 指向 PFA 的位址，應選擇：
A. BUILD> XXX　　B. CREATE XXX
C. ： YYY CREATE XXX ；　　D. ： XXX ；

22. 設一詞的定義是
： XXX CREATE ， DOES> @ ；
在執行 176 XXX YYY 之後，
A. 詞典中生成新詞 YYY，堆疊上存放 176
B. 詞典中生成新詞 YYY，176 放入 YYY 的 PFA 內
C. 詞典中生成新詞 YYY，176 放入 YYY 的 PFA 內，且堆疊上放有 PFA 的地址
D. 詞典中不生成任何新詞

23. 在冒號定義中能把 64 及 3 的積當作爲一個數編譯到詞典中去的是：
A. LITERAL 64 3 *
B. [LITERAL] 192
C. [64 3 *] LITERAL
D. 64 3 *

三、判錯(20 分)

1. 定義兩個個詞彙並在以後的定義中使用它(10 分)

```
VOCABULARY XX       VOCABULARY YY      YY DEFINITIONS
     A                    B                  C

：ABC ;             XX DEFINITIONS     ：DEF ;
  D                      E                F

： GHI     YY DEF    XX ABC     ;
   G         H         I        J
```

程式中其有 _____ 處錯，分別爲 _____

2. 定義一個 BASE1，以 BASE1 爲數基，顯示數 n.　　(10 分)

```
：TEST (n ---)
BASE @    R>
   A       B
VARIABLE BASE1    BASE1 @
       C             D
BASE  !    0.
   E      F
<# # # # # #>    TYPE
      G           H
>R    BASE  !  ;
I          J
```

上述程序中有 _____ 錯，分別爲 _____

四、堆疊狀態分析(20 分)

分析下列程式，並追蹤堆疊狀態，用標準注釋方式在括號內表示堆疊狀態.

```
 DECIMAL
 :  #IN ( --- n ) (以鍵盤等待輸入一個數在堆疊上)
        0 BEGIN KEY ( n c )
                DUP 13 = ( n c f )
                IF DROP    (             )
                EXIT
                THEN
                DUP 8 =    (             )
                IF EMIT 32 EMIT 8 EMIT
                10 /       (             )
                ELSE DUP   (             )
                48 <       (             )
                OVER       (             )
                57 >       (             )
                OR         (             )
                IF DROP 7 EMIT
                ELSE DUP EMIT
                  48 —     (             )
                SWAP 10
                * +        (             )
                THEN
```

THEN
AGAIN ；

第三部份：程式設計(36 分)

一、填 FORTH 詞(22 分)

在下列程序中，根據程式注釋，在下列線上填寫適當的 FORTH 詞，
WORDS 的功能爲查看當前搜尋詞彙：

```
： WORDS HEX CR CR
(找出當前詞彙的最後一個被定義的詞的名稱域地址) _____
?DUP IF BEGIN DUP DUP 0
<# # # # # #>
TYPE IF (空格) _____ (顯示詞名) _____
ELSE (顯示 "null") _____ DROP
THEN OUT @ 30 >
IF (跳行) _____
ELSE 14 OUT @ OVER MOD —
        (空出若干個空格) _____
THEN
(由名稱域地址找出下一詞的名稱域地址) _____ DUP 0=
(終端有輸入鍵動作嗎?) _____
DUP IF(等待鍵盤輸入) _____DROP THEN
(上面兩判斷有一爲真嗎?) ____ UNTIL
ELSE(顯示"Empty vocabalary") _____
THEN ；
```

二、程式設計 (14 分)

已知圓的軌跡的參數方程爲

X= x0 + rCOSa
Y= y0 + rSINa

其中(x0，y0)爲圓心坐標，r 爲圓的半徑，設系統中有一繪圖的詞 LINE 可劃一直線，其操作的堆疊狀態爲(x1 y1 x2 y2 color ---)，(x1，y1)爲劃線起始點，(x2，y2)爲劃線末點，color 爲顏色，color 爲 0 是背景色，爲 1 是白色。用 FORTH 語言設計一程式，其算法是先求出一點(x1，y1)再求第二點(x2，y2)用 LINE 將(x1，y1)，(x2，y2)相連，然後前一點(x2，y2)作爲下一線段的(x1，y1).再求出下一點.每求一點，將 a 遞增一次，使其以 0 增至 360，則可劃出整個圓的圖形，詞 SIN 、

COS 求正弦和餘弦，其堆疊狀態爲(度—正弦*10000)，
(度—餘弦*10000)，給出的 爲整數範圍，求出的正弦、餘弦值是其實際值的 10000 倍，程式設計中要避免溢出.(提示：末點坐標可用雙精密度數保存，以作爲下一次起始點坐標值)
程序設計過程用下述詞描述：

X= (x0 r a --- x)
Y= (y0 r a --- y)
XY= (x0 y0 r a --- x y)
CIRCLE (x0 y0 r ---)

對每個詞分別進行定義，在答卷紙上寫出完整的劃圓程序.

第四部分：英文試題(18 分)

一、在供選擇的答案中選出下列英文句中關係最密切的詞填入相應的[ ]處，把編號寫在答卷紙的對應當內(9 分).

1. FORTH is _____ .Because you can add to the language,you can tailor it to your own needs. Since almost everything in FORTH is written in FORTH--the text editor,assembler, etc.-you can acces and alter all of it.

2. FORTH is ____ .FORTH runs much faster than many other high-level languages. Because the interpretation scheme is so elegant, interpreter overhead is minimal.Furthermore,FORTH includes.a built-in assembler for speed-critical routines. As a result, FORTH can run almost as fast as machine code itself.

3. FORTH is _____ .Only a small nucleus of code needs to be rewritten to move the entire language to a new computer. FORTH has been implemented on almost every computer developed to date.

4. 答案：

A. fast	B. slow	C. compact	D. powerful
E. extensible	F.transportable	G. interactive	H. structured
I. restrictive			

二、下列短文翻譯成中文(9分)

1. FORTH is a language,an operating system,a set of tools, and a philosophy. It is an ideal means for thinking because it corresponds to the way our minds work.< Thinking Forth > is thinking simple,thinking elegant,thinking flexible.It is not restrictive,not complicated,nor over-general.< Thinking Forth > synthesizes the Forth approach with many principles taught by mordern computer science. (3分)

2. Busiiness,industry,and education are discovering that FORTH is an especially effective language for producing compact, efficient applications for real-time,real-world tasks. < Thinking FORTH > combines the philosophy behind FORTH with thetradtional,disciplined approaches,to software development--to give you a basis for writing more readable,easier-to-write,and easier-to-maintain software applications in any language. (3分)

3. FORTH is like the Tso(道教): it is a way . and is realized when followed. Its fragility is its strength,its simplicity its direction. (3分)

附錄(r)-charles moore 演講文

CHARLES MOORE 先生
講 FORTH 來 源 譯 文

前言

在這個世界上有誰比查理士莫爾先生(FORTH 語言的發明人)更有資格來告訴我們這個迷人系統的歷史? 本章是 CHARLES MOORE 先生於 1979 年 10 月舊金山舉行的 FORTH 會議上的演講記錄。我們要對他以及無花果會(FIG) 表示感謝，是他們促成了此次會議。本次演講的原文刊載於 FORTH 空閒期刊 (FORTH DIMENSIONS)第一卷，第六集。

歡迎詞

謝謝各位! 我很榮幸，身在此地，並看到各位如此全心投入，一件我從不敢奢望有這麼多人喜歡的東西。回想過去在黑暗時期，我曾想或許哪天，對扶輪社員演說符式。這是一個頗爲特別的團體。

結果顯示無花果年會把此次會議當成 FORTH 的十週年慶祝會是非常正確的。我並不想敘述 FORTH 的歷史。有一個理由是，我對那些事情沒有很好的記性，而且也不會特別正確與完整----我只想對十年來所發生之事提出一些我各人的感受。我並不提及在歐洲，或在舊金山所發生之事；對此或深表歉意；但是似乎並沒有必要去探究 FORTH 的歷史。這兒有一段歷史，但我首先要講述一下 FORTH 是什麼。這是一個經過深思熟慮的論題。

面面觀

FORTH 是一種操作系統嗎? 它是一種語言? 還是一種心境? 我建議從五個層面來追溯十年來的歷史。我準備按此順序去做，如果我們要截斷詰尾也不會有人介意。

FORTH 的這五個層面是哲學、語言、實行、電腦、組織，(意指類似 FIG 的組織)。如果你討論 FORTH 語言，你可以討論這些事情，而當你討論 FORTH 組織時，你也可以討論其他事情。這是一個很合理的組織，因爲它真正廣泛地觸及了社會問題。

當我年輕的時候，我並不怎麼喜歡自己。我回想當時頗爲自負----有點過了份----我想按自己的方式做事，對不被容許之事不能信服，甚至有點難以相處。現在一切都改變了。但是我特別地不安定。我曾提出過一些構想，人人都告訴我

是錯的，而我卻認爲是對的。但是如果我對了，那麼所有其他的人必然都是錯的，而他們的人數卻遠超過我。裝出一副非常不感興趣的面孔，要有極度的傲慢的信心。

你或許會注意到 FORTH 是一種兩極的觀念。它正如宗教或政治一般，有人愛它也有人恨它，而如果你想開啓爭端的話，只要說----" 哇 FORTH 真是了不起的語言" 。

我想你們有些人，大約十年前曾感受到一位程式員所面臨的問題。這些問題和今天一位程式員所面臨的完全相同! 在過去廿年內，軟體工業沒有什麼進步。這在十年前非常明顯，它的狀態紊亂。當 FORTRAN 發明出時，也沒有什麼完整的規則。但是卻沒有人懷疑。那時有一種沒有說出的假設，認爲事情該怎麼樣就怎麼樣，它們的實質不能有所不同。

哲學

讓我現在來談談哲學。從前(1968 年)我是一個自由程式員。我去替一家地毯製造商工作並學習 COBOL 語言，一方面是由於經濟需要，另一方面是有一種想法" 這是一種我不懂的語言----讓我們再來學一種" 。這些人需要一個繪圖系統。它是一部 IBM 1130，加上一台 2250 繪圖顯示器，一套當時最美好的裝備，昂貴極了! 理論上是它能幫助我們，設計地毯甚至傢俱。沒有人能確定此點，但他們願意去試試看。1130 是一部非常重要的電腦。它擁有第一片儲存匣式磁碟。它還有一台讀卡機，一台打卡機，及一台鍵盤式終端機。磁碟片的備存是打卡機! 我不認爲我曾經備存過磁片，但我確記曾多次將操作系統重新饋入。

這套 1130 有一天被搬走了，因爲沒有顏色，它根本不能幫助製造地毯。他們又有了一部 Burroughs 5500 ，有一度使用 ALGOL 程式語言，當時 ALGOL 語言並不普遍。這是一部非常好的機器。他們用 COBOL 語言來寫程式。這些人很有上進心。我想讚揚們一下----莫哈斯科工業公司(Mohasco Industries)。我把 FORTH 放入 5500 ----這很不尋常的。它是從 1130 交錯編譯(Cross-Compiled)到 5500 上的，因爲 5500 上沒有組合程式。 ALGOL 有一種衍生語言叫 ESBOL 語言，Burroughs 公司用它來編譯操作系統(它不供使用者使用)。但是我從這部機器上學會了推下堆疊(Push-down Stack)，並獲得了很多樂趣。它是第三班的工作，因爲 5500 非常忙碌。它被一部 Univac 1108 取代，所以我就在 1180 上執行 FORTH 。它控制了一群 COBOL 模組的交互作用，而這些模組是真正在工作的。由於預期的財務逆轉 1180 被刪除了，程式員們都辭職了，而我就轉往國家無線電天文台(National Radio Astronomy Observatory 簡稱 NRAO)工作。

在我離開之前----最後一個星期----我寫了一本書。它的名稱是" 設計一種問題導向的程式語言 "，這本書代表了我當時的哲學。這本書現在回頭來看，真覺得好笑。它同時也描述了當時的 FORTH 是什麼。它之所以可笑是因爲，任何人都不知道，我只陳述了意見和態度，並沒有設計任何程式。

FORHT 首先出現在那部 1130 上，而當時就叫 FORTH 。它已擁有了 FORTH 所有的基本特性，這一點以後再說明。

F--O--R--T--H 是" fourth "的五個字母的縮寫，代表第四代電腦。你該記得，這是第三代電腦的時代，而我將要跳躍過所有這一切。但是 FORTH 在 1130 上運作， 1130 只允許五個字母的識別號。

現代 FORTH 的第一件事是在 NRAO 的 Honeywell 316 程式。我受雇於喬治。科納(Geroge Conant)設計一個無線電望遠鏡資料擷取程式。我有了一部電腦(使得其他自認該得的人們羨慕不已)。我能放手去做我所能用它來做的事，提出所需的結果。我就開始進行，沒有人真正期望或欣賞它們的結果會如何。

我們在 NRAO 發展出一些系統，並遭遇到軟體專利的發佈問題。程式不能給予專利權；不應該給予專利；請准專利非常昂貴。NRAO 與研究合作社(Research Cooperation)有個協定；(該公司從大學拉出一些科技人員，另起新公司，可用來改善人類。)(他們爲不知如何或不管如何做的人申請專利。) FORTH 看來似乎該申請專利，所以我們花了一年的時間寫申請案，調查並徵詢律師的意見。結論是它或許可以請准專利，但是要做這件事必須到最高法院提出訴訟。 NRAO 對此不表興趣。身爲發明者，我有不履行的權利，但是我也不希望花費一萬美元，所以 FORTH 未請准專利。這也許是一件好事情。我認爲如果任何軟體套裝程式有資格申請專利， FORTH 也應有資格。它其中並沒有什麼真正的新發明，但是它的套裝程式卻不會用別的方式組合在一起。如果你把這項推論用在硬體上，硬體是可申請專利的。公司拒絕對軟體提供任何有效的保護，使我非常失望。或許是由於缺少來自工業界內部的反對意見，以及默默的順從，瞭解到反正今日的軟體，一年後即將老舊作廢。

有一些信徒，從其他天文學家那兒獲得了興趣，並組成了 FORTH 公司(FORTH， Inc．)。我們發展出 mini FORTH(FORTH 用在迷你型電腦上)，其構想是有一種程式設計的工具。我們第一次認識到這個工具，是當我們把 LSI-11 和 FORTH ，放入一個手提箱內的時候。我想我是第一個有電腦輔助工具的程式員，我的電腦在箱內可提著到處走。我和我的電腦交談，我的電腦和你的電腦交談，我們之間的溝通，比我直接跟你溝通，更有效率。用這種工具，我們把 FORTH 放入了許多電腦中。我的目的是要成爲一位多產的程式員。在此之前，我估計，以我進行的速度，我在四十年內只能寫四十個程式。就只有這些了。就此結束了! 這就是我的命運，但是我希望多寫一些程式。這世上還有許多事情要做，而我希望我也能做那些。

花了很長的一段時間，我仍然沒有我希望的電腦，但是我以十倍於前的速度工作，現在也看到了其他電腦輔助的程式員。我很驚呀程式員，經由電腦輔助不應如此明顯。期望程式員去和他那本質上不友善的機器交談，是違反了我們勸告他人遵循的態度。

時間一天天過去， FORTH 很顯然地成爲了一種放大器。一位好的程式員可以用 FORTH 做出絕妙的工作，一位差勁的程式員卻會做得更糟。我看過非常差的 FORTH，並且無法對作者解釋爲什麼它很差。好的 FORTH 有幾個特性：定義很

短而且很多。差的 FORTH 式每一塊只有一個定義，很大，很長、很難懂。這是顯而易見的，但是卻很難指出某件事弄錯的例子，或是解釋爲何及如何。 BASIC 和 FORTRAN 對程式員的品質較不易感知。我是一位很好的 FORTRAN 程式員。我覺得已經盡力用 FORTRAN 做出最好的工作，但是，比起其他人做的，也好不到那裏去。也許我把東西縮排得比較好一點，我先宣告事情，而別人留在基元設定之後才宣告。你還能再做什麼? 基於某種意念我說，" 讓我恰當地做它。讓我使用一種我欣賞的工具；如果別人不能使用得很好，我很抱歉，但那不是我的目標 "。在這種意念上， FORTH 是一種優秀的語言。從另一方面來說，如果程式的長短大小適當，並給予適當的刺激，我認爲 FORTH 是連小學生都能十分有效使用的語言。

最後，poly FORTH 是我們在過去十年中發展 FORTH 上所學習到各種事物的結晶，我認爲它是非常好的套裝程式。我可預見在此語言的設計上，除了調節日形重要的標準之外，不會有基本的改變。直至現在還沒有理由定標準。
在 Kitt Peak 有組織效率的內在標準，但卻沒有要求轉移性。事實上，我知道很少有程式，會認真地嚐試轉移性。改變此點的時機顯然是來臨了。

在其他領域還會有些發展，而今天有一個領域很有力的使人心服。或許 FORTH 並不僅是一種程式語言。或許它是在說一些很重要的有關於溝通的事----人與人之間，電腦與電腦之間、動物與動物之間。這多麼令人吃驚! 我從未想過任何人會真正" 說 FORTH " (" speak FORTH ")，去試著和電腦以外的其他東西溝通，事實如此已無疑問了。也許有些觀念包含在 FORTH 內，亦即對於溝通的基本問題有較廣大的一般用途。

現在在哲學部份，我想念一首詩。這是一首你們有些人讀過的詩。它是一首英國文學古典詩作的翻譯，其詩如下：

```
: SONG
      SIXPENCE!
      BEGIN RYE @ POCKET +! ?FULL END
      24 0 DO BLACKBIRD I + @ PIE +! LOOP
      BAKE BEGIN ?OPENED END
      SIGN DAINTY-DISH KING ! SURPRISE ;
```

這首詩是奈德．康林(Ned Conklin)，對那種東西他很在行，而且他是第一位 FORTH 詩人。這世上是否有地方作此溝通? 我不知道。你想要改寫的任何東西都很容易看懂。很顯然的它是一種有效的溝通方式。

介紹

讓我介紹兩個人，自從我接觸此題目後。十年前有了一位 FORTH 程式員。

我估計現在已有 1,000 位 FORTH 程式員，這是 2 的 10 次方，並且出現得很圓滿----一年增加一倍。事實上，我認爲這個加倍的數目比那個數目略少一點----三年內 10 倍，如大家所預期的結果爲 2000 。我們總計下一年會變成 2 倍；再下一年也是 2 倍，如果你相信數字學，這種成長是無可爭論的。這是一條曲線，你插補此曲線並畫出結果。我們不知道它會變成如何。

FORTH 公司(FORTH，Inc．)沒辦法在明年訓練出 2 倍的程式員----嗯，也許我們可以。但是，好歹整個 FORTH 團體在明年得訓練出 2 倍的人，往後也是。也許蘋果電腦(Apples)或 Radio Shacks 會成爲達此目的的工具。看來能力剛好足夠保持此指數曲線的成長。對以下兩件事我有很充份的信心：(1)加倍的時間是一年，(2)它將繼續增加。現在如果你用圖繪出 FORTH 系統的數目或 FORTH 系統的價值，或是市場侵占率，就有了間接證據可支持此信心。在每一方面，你都會得到相同的成長曲線，所以我認爲此成長曲線是真實的。

十年前只有一位 FORTH 程式員。第二位 FORTH 程式員現正在座，請大家見過伊莉莎白．拉德(Elizabeth Rather)。這在量的方面是很大的躍進，從 1 到 2 。下一步是 4 位，他們來自 Kitt Peak ，如果有人想知道的話，成長可暫由該處追溯起。事實上，第一位 FORTH 使用者也在座，他是奈德．康林(Ned Conklin)。他是 NRAO 在 Kitt Peak 站的主管，負責望遠鏡的運作，使他的望遠鏡涉入此危險的冒險。那是很重要的望遠鏡，因爲在過去十年中它擔負了半數的星際微分子的發現。

然而，我並未真正要求，准予託付這項行動給這些人。沒有人知道結果會是什麼。它看來做得並不壞。符式仍在 Kitt Peak 的那架望遠鏡上及許多其他的望遠鏡上運作。

語言

現在讓我們來談談語言以及 FORTH 如何成爲今天的樣子。在此十年前更早的時候有一段前史，我有一些幻燈片顯示那段時間。這些是真正的前史----我發現了一堆很老的目錄表，就把它們照了下來。最先被發現的 FORTH 構成要素是翻譯器(Interpreter)。(圖一)這是用 ALGOL 設計的一個早期的翻譯器。這是六十年代初期在史丹佛線性加速器中心做出來的。這個程式仍然存在，稱爲 TRANSPORT。它設計光束輸送系統。在此你可看到早期的詞典。 ATOM 指予顯示 LISP 的影響力。 ATOM 是一個不可分割的實體，我們現在稱爲一個" 指令 "。從輸入卡片上，讀到一個指令 DRIFT 之後，我就會執行 DRIFT 副程式等等。我看過無數的目錄，並且發現這種程式設計的技巧十分調和，這是我當程式的方式。我有一個輸入卡疊，它使用你現在看見的一個非常好的結構，來解碼：指令用空格分開，沒有特別限制指令的長度(你可由 SOLENOID 看出來)，然而，只有第一個字母是有意義的。

```
IF ATOM="DRIFT" THEN DRIFT
ELSE IF ATOM="QUAD" THEN QUAD
ELSE IF ATOM="BEND" THEN BEND
ELSE IF ATOM="FACE" THEN FACE (-1)
ELSE IF ATOM="ROTATE" THEN ROTATE
ELSE IF ATOM="SOLENO" THEN SOLENOID
ELSE IF ATOM="SEX"  THEN SEX
ELSE IF ATOM="ACC"  THEN ACC

ELSE IF ATOM="MATTRIX" THEN BEGIN IF NOT FITTING THEN BEGGIN
       REAL A;
       WRITE1(3,0,0CORE[S]); LINE (-(8+42X(ORDERR-1)));
       FOR J=1 STEP 1 UNTIL 6 DO BEGIN
         FOR K=1 STEP 1 UNTIL 6 DO WRITE1(2,8,R1[J,K]XUNIT[K]
                /UNIT[J],2);
         LINE(0) END;
       IF ORDER=2 THEN FOR C=1 STEP 1 UNTIL 6 DO BEGIN
```

(Figure 1)

這兒有另一個例子，十分類似。(圖二)此處 ATOM 變成 W ，我在找" + " 和" - "和" T "，" R "，" A "和" I " ----它們代表了文字編輯器的早期版本。這又是用 ALGOL 。我並不十分清楚被編輯的是什麼。我想它是某種檔案分類程式，或許在要印出或重新排列的卡片上。

```
120 CYCLE; FILL OUTPUT WITH BUFFER[1] .BUFFER[2]:
  1 WHILE WORD NEQ "END    " DO
  2 IF W=GM1 THEN REPLY ("OK   ")
  3 ELSE IF NUMERIC THEN L:=MIN(W-1,EOF)
  4 ELSE IF W="+      "THEN L:=MIN(L+WORD.EOF)
  5 ELSE IF W="-      "THEN L:=MAX(L-WORD.O)
  6 ELSE IF W="T      "THEN BEGIN
  7    IF WORD=G 1 THEN W:=1; W:=MIN(L+W-1, EOF);
  8    FOR L:=L STEP 1 UNTIL W DO BEGIN
  9       POSITION: TYPE END; L:=L-1 END
130 ELSE IF W-"R      "THEN BEGIN
  1    POSITION; REPLACE END
```

```
  2 ELSE IF W="A       "THEN BEGIN
  3    L:=EOF:=EOF+1; REPLACE END
  4 ELSE IF W="I       "OR W="D                "THEN BEGIN
  5    IF NOT RECOPY THEN BEGIN
  6       RECCOPY:=TRUE; REWIND(CARD) END;
  7    POSITION; (F W ="I       "THEN BECIN
  8         PLACE: REPLACE ENI)
  9 ELSE BEGIN EMPTY:=TRUE; IF WORD NEQ GM1 THEN BFGIN
140         L:=MIN(1+W-1, EOF); SPACE(CARD,L-LO+1); LO:=L+1
  1         END END END
```

(figure 2.)

另外，還有一種設定詞典的方法。(圖三) 我將字串填入陣列，我將要搜尋該填列找一個配對，經由一個計算過的 GO-TO 取出索引和向量。

```
  7 LABEL UNDEFINED, BACKWARD1, TYPE2, FIND, INSERT, DEFETE,
          ERASE,
  8 START, REPEAT1, BOUNDARY1, BEGIN2, END2, QUIT1, FORTRAN,
  9 COBOL; DATA, PACK1;
280 SWITCH SW:=UNDEFINED, BACKWARD1 TYPE2, FIND, INSERT,
              DELETE, ERASE,
  1 BACKWARD1, TYPE2, FIND, INSERT, DELETE, ERASE, START,
    REPEAT1, BACK,
  2 BOUNDARY1, BEGIN2, END2, QUIT1, ALGOL, FORTRAN, COBOL,
    DATA, PACK1;
  3 ALPHA ARRAY COMMAND [1:32];
  4 FILL COMMAND [*] WITH
  5 "-       ",  "T       ", "F    ", "I         ", "D    ",
    "E       ",
  6 "BACKWAR", "TYPE    ", "FIND ", "INSERT  ", "DELETE ",
    "ERASE  ",
  7 "START  ", "REPEAT  ", "     ", "BOUNDAR ", "BEGIN  ",
    "END   ",
  8 "EXIT   ", "ALGOL   ", "FORTRAN", "COBOL :, :DATA   ",
    "PACK  ",
  9 "       ",  COUNT:=0; RETURN; COPY ("EDIT#RE" U "ADY ");
290 TRANSMIT;
```

```
1 BACK: SOURCE (1); WORD1;
2     IF W="         "THEN GO QUIT1; GO TO SW[MEMBER
      (COMMAN W)+1};
```

(figure 3)

下面是另一個執行詞典的方法。(圖四) 這是我有記錄的堆疊的第一次露面。我在一個條件敘述內，尋找指令並把 NEXT 設定在索引中。這是我能找到的 NEXT 的第一次出現。

```
  8 PROCEDURE RELEVANCE; BEGIN REAL T,KO;
  9     J:=0; I:=-1; WHILE WORD NEQ "END         "DO
180     IF W="=       "THEN NEXT:=3
  1     ELSE IF W="GT     "THEN NEXT:=4
  2     ELSE IF W="LT     "THEN NEXT:=5
  3     ELSE IF W="NOT    "THEN NEXT:=6
  4     ELSE IF W="AND    "THEN NEXT:=7
  5     ELSE IF W="OR     "THEN NEXT:=8
  6     ELSE IF W="+      "THEN NEXT:=9
  7     ELSE IF W="-      "THEN NEXT:=10
  8     ELSE IF W="*      "THEN NEXT:=11
  9     ELSE IF W="/      "THEN NEXT:=12
190     ELSE IF KO:=SEARCH1 (W) GEQ 0 THEN BEGIN
  1        NEXT:=1; NEXT:=K:=KO END
  2     ELSE BEGIN
  3        NEXT:=2;
  4        IF BASE[K]=" " THEN NEXT:=WORDS [0]
  5        ELSE NEXT:= W END;
  6     NEXT:=0 END;
```

(Figure 4)

這兒是另一半程式----這是堆疊的執行。(圖五) 這是 ALGOL 的變形，叫 BALGOL ，它允許你把指定的敘述放入另外的敘述中。" 堆疊 J 被 J-1 取代，正是你如何把東西推上堆疊。我" 最不喜歡 "的 ALGOL 特性之一是我必須像下面這樣玩遊戲" jeal of boolean of stack of Jand boolean of ..... "就爲了迴避 ALGOL 堅持要我用的臨動打字 (Automatic Typing)。現在，讓我把從

卡片疊解碼而來的參數，當作程式引數操作，這是特別地企求的。換句話說，如果我想得到一個角度的正弦，我可以說 ANGLE SINE ，但如果我想將此角度的單位轉換成另一種，我就至少需要一些簡單的算術運算子，而這就可以提供他們，這也是在史丹佛 (Stanford) 做出的。

```
  6
  7 BOOLEAN PROCEDURE RELEVANT; BEGIN
  8  I:=J:=-1; STACK [0] :=1; DO CASE NEXT OF BEGIN
  9        J:=-1;
210 STACK [J:=J+1] :=CONTENT;
  1         STACK[J:=J+1]:=NEXT;
  2         STACK[J;=J-1]:=REAL(STACK[J]=STACK[J+1]);
  3         STACK[J:=J-1]:=REAL(STACK[J] GTR STACK[J+1]);
  4         STACK[J:=J-1]:=REAL(STACK[J] LSS STACK[J+1]);
  5         STACK[J] :=REAL(NOT BOOLEAN(STACK[J])));
  6         STACK[J:=J-1]:=REAL(BOOLEAN(STACK[J]);
              BOOLEAN(STACK[J+1])));
  7         STACK[J:=J-1]:=REAL (BOOLEAN (STACK [J]) OR
              BOOLEAN (STACK [J;1])));
  8         STACK[J:=J-1]:=STACK[J]+STACK[J+1];
  9         STACK[J:=J-1]:=STACK[J]-STACK[J+1];
220         STACK[J:=J-1]:=STACK[J]*STACK[J+1];
  1         STACK[J:=J-1]:=STACK[J]/STACK[J+1];
  2         END UNTIL J LSS 0;
  3   RELEVANT:=BOOLFAN(STACK[0]  END;
```

(Figure 5.)

現在這裏有一個 PL/1 程式，它在稍後的日子裏做著極類似的事情。(圖六) 在頂端你看到 JCL (Job Control Language) ，它也不是件好處理的事。我在書中提過的一種程式語言的評論，說一位身在典型電腦中心的程式員，爲了要完成任務，必須要懂得十九種語言。這包括了寫 FORTRAN 程式，敘述卡片疊，等等。這些符號很複雜難解，而不同於這兒是逗點，那兒是空格，等號表示的是不同的事情----十九種語言，就爲了完成任務。沒有人告知這種事實。沒有人會坐下來，上十九種的語言課程，不過爲了藉有效率地工作你必須在數星期或數月內無師自通。 FORTH ，我認爲可以簡化它們。

這兒有個 NEXT : PROCEDURE CHARACTER (圖七) 我不記得那種語法，但我想它是 NEXT 的第一個定義，它是一種進行前去取得下一個指令，並做某些事

的程序。這仍完全是 FORTH 前身 (pre-FORTH)。我們還沒有達到，我所認爲的第一個 FORTH 系統。

```
 1 //UTILITY   JOB SYSTEM OVERHEAD
 2 //           EXEC PGM=IEBUPDTE, PARM=NEW
 3 //SYSPRINT  DO   SYSOUT=A
 4 //SYSIN     DO   DATA
 5 ./          ADD  NAME=WORD,LEVEL=00,SOURCE=0,LIST=ALL
 6 NEXT: PROCEDURE CHARACTER(4);
 7   DECLARE KEYBOARD STREAM INPUT, PRINTER STREAM OUTPUT
      PRINT;
 8   DECLARE (1 TEXT CHARACTER (81) INITIAL((81)" "),

 9       2 C (81) CHARACTER (1), I INITIAL (81),W CHARACTER (4),
10       WORD CCHARACTER (32) VARYING BASED (),P,NUMERIC
            BIT (1)) EXTERNAL;
11 P=ADDR (C(I));
12 IF C(I)="-" OR C(I)="." OR "0" LE C(I) THEN BEGIN;
      NUMERIC="1"B;
13   IF C(I) NOT="." THEN DO I=I+1 BY 1 WHILE "0" LE C(I);
      END;
14   IF C(I)="." THEN DO I=I+1 BY 1 WHILE "0" LE C(I);
      END; END;
15 ELSE DO; NUMERIC="0"B;
16   IF "A" LE C(I) THEN DO I=I+1 BY 1 WHILE "A" LE C(I)
      OR C(I)="-"
17   END; ELSE; I=I+1; END;
18 W=WORD; RETURN(W);
```

(Figure 6 and 7)

下面是爲 IBM 360 所寫的稍後的 FORTH 版本。(圖八)那些是程式 PUSH 和 POP. PUSH 在 IBM 360-50 上要花 15 微秒的時間。它包括了疊層限制檢查，這會使時間加倍，這也是使我感覺到，執行時疊層檢查並不必要的事情之一。然而，伴隨此點的結果，是可怕的無法控制的疊層。 POP 也在那兒。它被寫入了一個沒有疊層操作的巨組合程式 (Macroassembler) 。它不可能去指一個先前的 " 任何東西 "，因此疊層上充滿了" L 19 資料常數、地址、 AL2 (*-L18) "，以給予一個相關的跳越到先前的一個去。它都可以做到但卻不愉快。

```
830  L18 DE AL2(*-L17)
831        NAME 3,X'+4555)',0 DUP
832+            DC   AL1(3),X'445550 '
833+            DC   X'0'
834+            ORG  *-2-VO
835+            DS   OH
836+            ORG  *+ 0+1
837+            DC   AL1(0*X'40'+X'40').AL2(4)
838  PUSH A SP,MFOUR     COSTS 15 OS
839        ST T,0(,SP)
840        CB   SP,DP
841        BCR 2,NEXT BHR
842             B ABORT
843  L19 DC AL2(*-L18)
844        DC AL1(4),X'4402CF50',X'40',AL2(8) DPCP
845        LA SP,4(,SP)
846  POP L T,4(,SP)  COSTS 21 US
847        LA SP,4(,SP)
848        C SP, SPOO
849        BCR 12 NEXT BNHR
850             B ABORT
```

(Figure 8.)

下面是用 COBOL 寫的 FORTH 版本。(圖九) 當然，這是在摩哈斯奇 (Mohasco) 做的，我設定了一個即將要從一個輸入字串解碼的定義指令表。樣子是如此的普及，以致我開始以爲，我正在說服自己在這兒做些什麼。我們很難替 COBOL 寫副程式。他們有一些副程式，他們也可以執行，但是他們不能有任何參數。這使得做任何有意義的事都有點麻煩。

```
1 MOVE "CONFIGURATION" TO IDENTIEY(4);
2 MOVE "DATA" TO IDENTIFY(5);
3 MOVE "FILE" TO IDENTIFY(6);
4 MOVE "FD" TO IDENTIFY(7);
5 MOVE "MD" TO IDENTIFY(8);
6 MOVE "SD" TO IDENTIFY(9);
7 MOVE "WORKING-STORAGE"  TO IDENTIFY(10);
```

```
 8 MOVE "CONSTANT" TO IDENTIFY(11);
 9 MOVE "PROCEDURE" TO IDENTIFY(12);
70 MOVE "INPUT-OUTPUT" TO IDENTIFY(13);
```

(Figure 9.)

下面是第一個 FORTH 文字的例子。(圖十) 這又是史丹佛做的。指令 DEFINE(亦即：) 開始一個定義，而指令 END (亦作；) 結束定義。指令" OPEN 很含糊 "。 NAME 似乎是介紹名字的方式。很顯然地在引號和指令中間沒有空格。程式中有一些堆疊操作指令(Stack Operators)的定義。最上面一行是 CODE- " OPEN DEFINE MINUS + END 我猜想它是減法運算。 SEAL 是早期的指令，爲了某種理由而封住詞典。 BREAK 指令我猜想是破除封閉" 。 < ：OPEN DEFINE- < END ; 和我們今天早期所用的定義相同。那是從我叫它 " Base Two " 的東西來，它想做爲某種基本程式語言----對此我已記不得了。

```
"-"OPEN DEFINE MINUS + END
SEAL "< "OPEN DEFINE - <   END BREAK
"NOT "OPEN DEFINE   MINUS 1+  END
"< "OPEN DEFINE   .<   END
"AND "OPEN DEFINE X  END
"OR "OPEN DEFINE NOT .NOT AND NOT END
     "T 1 1 "REAL DECLARE
"= "OPEN DEFINE  T=; DUP T<  . T> OR NOT END
"= "OPEN DEFINE  =NOT END
"< "OPEN DEFINE  > NOT END
"> "OPEN DEFINE  < NOT END
"DUMP "OPEN DEFINE NAME 10 "ALPHA WRITE; 3 10 "REAL WRITE
     0 LINE END
```

(Figure 10)

這是 5500 的原始符式版本。(圖十一)然而，很早地---- FORTH 放入了第二部電腦。顯然地代表 CODE 而這些是 5500 上堆疊操作碼的定義。現在當堆疊機器不普遍時 5500 是一部堆疊機器。他們用堆疊完成了很棒的工作。所有這些都是用一個 12 爻元指令執行，而這些操作指令現在的名字都直接源自 5500 操

作指令的名字。舉例來說，那就是 DUP 的來處。注意在利用詞彙以前，cOR 是分辨組合語言的 OR FORTH 的 OR 的方法。

```
LIST
0001 (     'PRIMITIVES'  26 LAST=  30 SIZE- )
0002 c = _S RETURN
0003 c @ <SD RETURN
0004 c +V 241, RETURN
0005
0006 c OR cOR RETURN
0007 c AND cAND RETURN
0008 c NOT 115, RETURN
0009 c DUP cDUP RETURN
000A c SWAP cSWAP RETURN
000B c DROP  cROP RETURN
000C c +  +1 RETURN
000D c -  -1 RETURN
000E c MINUS cMINUS RETURN
000F c * *1 RETURN
0010 c / /1 RETURN
0011 c MOD cMOD RETURN
```

(Figure 11)

這兒是一個爲 5500 寫的 FIND 的例子。(圖十二)注意指令 SCRAMBLE 被指用，它是一個冒號定義，用來亂七八糟地搜尋一番。很明顯地在此我有八條路線，正如去年我們放入 poly FORTH 中一樣。這些構想又回到從前。這就是越過開端的 FORTH ，幾乎正好就在十年前。當你追溯以往時可能會對過去十年間" 驚人的 "進步感到有些沮喪。

```
0013 cSM  FIND  SCRAMBLE  <SD cDUP
0014    41 >A 41 >B cBEGIN V <J 1771, cIF
0015      cBEGIN VO <U 1771, cIF
0016           1 <L RESULT
0017        cTHEN  ADDR cDUP 1 <L <S
0018        US WORD <J cEOUAL cIF
0019           V1   J  US  RESULT
```

```
001A          cTHEN cDUP <SD cBACK
001B      cTHEN  GET cBACK
001C : FIND TOP cFIND cIF UR <UR cB cTHEN;
```

(Figure 12)

下面是另一個原始程式的例子。(圖十三)這是從 Univac 1108 而來。這些是很早的記錄描述。這是檔案中一個記錄的設計，內有欄名及欄內敗位元數目。那是查詢濫債的 Dun & Bradstreet 參考檔。

```
3 DBI    DBI/MCORE 33 33
4     DUNS 8 NAME 24 STREET 19 CITY 15 STATE 4 ZIP 5
5     PHONE 10 BORN 3 PRODUCT 19 OFFICER 24 SIC 4 SIC1 4 SIC2
6     SIC3 4 SIC4 4 SIC5 4 TOTAL 5.0 EMPL 5.0 WORTH 9.0
           SALES 9.0
7     SUBS 1 HDQ 1 HEAD 8 PARENT 8 MAIL 19 CITY STATE 14
8     NAME 19
9END
```

(Figure 13)

這是最後一張幻燈片。(圖十四) 這是從 FORTH 公司 (FORTH, Inc.) 來的現代 FORTH 原始程式。向量算術是爲 Intel 8086 而寫的。CODE V+ 加上說明列出了在結果明顯的堆疊上的引數。間隔留得非常好的戈碼。它看來就像很好，很純淨的 FORTH ，而不是胡言亂語。一點也不神秘，對敏銳的讀者來說十分明顯，現在幻燈片放映完畢。

```
0     (VECTOR ARITHMETIC)
1 CODE V+ ( Y X  Y X) 0 POP 1 POP 2 POP 3 POP
2    2 0 ADD 3 1 ADD 1 PUSH 0 PUSH NEXT
3 CODE V- (Y X  Y X) 2 POP 3 POP 0 POP 1 POP
4    2 0 SUB 3 1 SUB 1 PUSH 0 PUSH NEXT
5
6 CODE UMINUS 0 POP 1 POP 1 NEG 1 PUSH
7    0 NEG 0 PUSH NEXT
8 : UMIN    ROT MIN >R MIN R. ;
9
```

```
10 CODE V*/  (Y X M D) 7 POP 3 POP 1 POP 1 POP
11          3 IMUL 7 IDIV 0 PUSH
12    1 0 MOV 3 IMUL 7 IDIV 0 PUSH NEXT
13
14
15
COPYRIGHT FORTH, INC OCT. 1979
```

(Fihure 14)

讓我現在把那些幻燈片所包含的操作程式依時間順序描述一番。第一件出現的事，就是文字解碼器，讀取打孔卡片。其次存在的是用運算子操作的數據堆疊。這些發生得非常早，大約在 1960 年。

這以後沒有什麼有價值的事發生，一直到 1968 年， 1130 第一次有能力完全控制電腦，與程式員相互交談的方式。這種機器，有了我前所未見的控制台。我以往總是託付卡片疊----現在我有一部打字機了。你知道我做了什麼嗎? 我還是用卡片疊。要想出如何使用鍵盤，以及它是否會替我做任何事情必須花很多時間。我對打孔很在行，而且也不在乎有沒有控制台。

最初的 FORTH 是用 FORTRAN 寫的。不久之後又用組合語言重寫過。很久以後才用 FORTH 寫。花費了許多時間，才知道 FORTH 能用自己來寫自己。第一件加入既有東西的是回返堆疊，我不記得爲什麼了。我不記得爲何不把回返數據放在參數堆疊上。這是一很重要的發展，認清必須要有兩個堆疊----剛好兩個，不多，不少。

下面一件加入的事情更重要。我不知道你是否欣賞它，但它是詞典的發明。特別是連接表列形式的詞典。尤其是字首碼欄的存在。直到那時，爲了要控制，旗號被設定出來，計算 GO-TO 的執行 (連接副程式與一個指今的構造) 。現在程式地址的存在(不如說是程式的指標)不可思議地快速執行一個指令 (一旦它被定義之後)。沒有其化語言有式碼區或任何類似之物。沒有其他語言，覺得必要很快地執行指令所定義的碼。你可以在你自己寶貴的時間裏著手去做，但即使是在這些日子裏， FORTH 的效率還是很重要的。這個系統的整個目的就是要在 2250 顯示幕上畫圖。 2250 是一部單獨的迷你電腦，與 1130 相互交談。從 1130 出來的是交錯組合程式(Cross-assembler)，它組合那些將被 2250 執行的指令。我想 2250 有它自己的記憶體。然而，十分複雜微妙的事，在一個非常迫切需要的環境中早早地被完成了。在 16k 記憶體上的 IBM 軟體可以在 2250 上十分緩慢地畫圖。我用 4K 完成的可以在 2250 上畫三度空間的活動圖形。但是如果每一個週期都被說明，而且最大極限被強迫定出，它就可以做到那點----這就是爲什麼 FORTRAN 必須要滾蛋的原因。我不能用 FORTRAN 做出足以讓人印象深刻的工作，組合器是必要的。

在此時冒號定義尚未被編譯----編譯器很久以後才出現。文字是儲存在定義體內，文字翻譯器爲了發現要做什麼事而重新翻譯文字。這與語言的效率相矛盾，但是我有一些大指令善於繪圖，而我並不需要翻譯太多。這聰明，被侷限於強迫交出無關的空格當做簡潔的媒介，而有人告訴我這就是現今 BASIC 在許多例子中執行的方式。

這機器有一個磁碟----我不能證明，但我幾乎可以肯定指令 BLOCK 的存在，是爲了要從磁碟中擷取資料記錄。我記得我必須使用 FORTRAN I/O 套裝程式，而它並未將塊 (Block) 放在我想要的地方----它把塊放在它想要的地方，而我必須撿起它們然後把它們移入我的緩衝區 (Buffer) 內。它有一個組合器以組合 1130 碼，它有一個目的碼組合器以組合 2250 碼並清除 B-5500 碼。5500 碼被取用得很頻繁，足以重新編譯自己。除此之外，沒有公用程式，因爲我沒辦法在第三次移位之外的機器 上擷取。

這是位於不可被稱爲 FORTH 的東西，與可以被稱爲 FORTH 的東西之間的過渡期。所有主要的特性，除了編譯器之外，都在 1968 年出現了。下一步花了很長的時間。

第一個編譯器幾年後出現在 NRAO 的 Honeywell 316 系統上。它起因於一項認知----指令可以被編譯而不必重新翻譯文字，平均 5 個字母一個指令可以被每個指令兩個位元取代 (以 2 或 3 的因數縮小) 。執行時間會大大的變快。但是，如果它那麼容易，爲什麼別人做不到? 我花了很多時間 ，說服自己可以編譯任何東西以及所有的東西。例如，條件敘述(Conditional Expression)總得要被編譯。在編譯之前，如果你遇到一個 IF ，你可以在文字串之前仔細查看，直到你到了 ELSE 或 THEN 爲止。如果你要編譯事情又怎麼做 IF 呢? 但是，它有效了! 然而，我又不記得結果了。

也許是 316 編譯 360 ，但我認爲是 360 編譯 316 。但是，在 FORTH 的初期，今日的構想已在那兒----交錯編譯，交錯組合那兒不同的電腦。

間斷 (Interrupts)大約在此時出現。使用電腦的間斷能力是很重要的，但是在此之前我並沒有完成它。我不知道任何關於間斷的事，但是 I/O 並不是間斷----驅動。如果使用程式需要的話，間斷可供使用。FORTH 不囉嗦。

多重的程式(Multi-pprogrammer) 在幾年之後，當我們把一個該系統改良的版本放入 Kitt Peak PDP-11 時出現了。多重程式有四種功能。輸入仍然沒有間斷驅動，這是很示幸的。當 FORTH 公司生產它第一部多台終端系統 (Multiterminal System) 時間斷驅動出現了。它並不特別能使事情加速。如果你計算週期，它有效率得多，當許多人同時在打字時，它也預防了任何字母的丟失。當它們都被暫存和等待時， FORTH 不必在下一字母來之前快速地查出每一字母。

資料庫管理也在此時出現。它已被廣泛地改變了，正如 FORTH 一樣，但是基本上沒有什麼改變。今天下午我略述過的檔案，記錄和欄的觀念始終於 1974 年左右。

第一部目的編譯器稍後用 microFORTH 寫出。它們是非常複雜的事情，比我

預期的複雜得多。

我想它完成了我所認爲今日 FORTH 的能力，你可以看見它們是如何進行的。我從來沒有坐下來去設計一種程式語言。我在問題發生時去解決它們。當改良性能的要求來了，我就會坐下來煩心一陣，並且提出改良性能的方式。過程完畢並不明顯，但是我想現在過程必須被帶入硬體領域是很明顯的。

硬體

我設計過一些很昂貴的電腦。因爲我應該是要寫軟體來賺錢的。我認爲今日的硬體和二十年前的軟體同一類型。我無意冒犯，但這正是硬體從業人員，學習一些軟體的時候了，而且在既有科技執行的可能性上，有一個順序或兩項重要的改良。我們並不需要漠秒(10-12 秒) 電腦 (Picosecond Computer) 來做實質的速度改良。有此體認，試圖使軟體更臻完美就沒有用了，直到我們在硬體上採取第一步的嘗試爲止。硬體的重新設計，必須要和軟體重新設計一樣徹底。微處理機的標準，並未留意到 FORTH 。那些可以做微程式規劃的迷你電腦，沒有辦法好到值得去規劃微程式。可利用的改進比你用這些半受限定的所能達到的要驚人得多。

實現

好了，讓我們將狀況轉換一下。我很樂意談談我注意到的 FORTH 的實現。我曾經提及過它們，但我想很快地談談一連串的 CPU，它們用能力來迷惑你的心靈。實際上它是在電腦史中轉一圈，而令人興奮的是這所有都發生在十年之內。

FORTH 曾經用 FORTRAN 、ALGOL 、PL/1 、COBOL 組合語言(assembler)和 FORTH 寫過。我相信你們但有些人會遇到相同歷史的其他語言。它曾建在 IBM 1130 、Burroughs 5500 、Univac 1108 、Honeywell 316、IBM 360 、Nova、HP 2100 之上 (不是由我 ，而是由 Kitt Peak 的保羅・史高特)。建在 PDP-10 和 PDP-11 上 (由 Cal-Tech 的馬提・艾溫)建在 PDP -11 上 (由 FORTH 公司) 建在 Varian 620 、Mod -Comp、GA/SPC-16 , CDC 6400 上 (由 Kitt Peak) ，建在 PDP-8 ， COMputer Automation LSI-4 ，RCA 1802，Interdata ，Motorola 6800 ，Intel 8080，Intel 8086 ，TI 9900 及即將有的 68000 ，Z 8000 ，6809----我知道你們還有 6502 和 Four Phase 。(聽眾回答：還有 Illiac !)我已提出了問題----是否 FORTH 已被放入現有的每一種電腦上了?

電腦

我們現在談的是 FORTH 電腦----有一種 FORTH 電腦。我知道的第一部，大約 1973 年英國 Jodrell 銀行所建立的，它是由 Ferranti 電腦新設計而來，這種電腦我想已經不生產了。他們要建立他們自己的爻分割版本，大約在同時期他們發現了 FORTH ，修正了指令集去接受 FORTH ，並建立了非常快速的 FORTH 電腦。但我從未見過它。我曾與它的設計者約翰・大衛(John Davies)談過，他是早期的 FORTH 熱愛者，並且在此道頗具盛名。

在 1973 年出現了 General Logic 和 Dean Sanderson。這機器有資格當 FORTH 電腦，因爲它有一個 FORTH 指令集，並且那兒有段故事。 Dean 給我看他的指令集，裏面有一個很有趣的指令，看不出爲什麼。我想它是某種虛擬指令或 Catchall，因爲它有著怪異的特質。它不可能有用--它就是 NEXT。它是一個單一指令 NEXT ----它漂亮極了。它是對指令集的簡單修正。這兒那兒有些線路；這是我第一次看到 FORTH 電腦。這就是將普通電腦，改變爲 FORTH 電腦的能力。

我們有一些構想，能夠有效率地做類似這樣的修正，但是我沒聽過它們任何一個被實現出來。我聽說 Cybek 已有它自己的機器了，是由 Eric Ery 所建。這些都是謠言，我只當作耳邊風。 Child,Inc. 在做點像顯示器繪圖系統。我聽說，他們在一部 FORTH 電腦上工作，而該電腦在去年二月份就已可用了，但我從未見過。然而，我認爲他們有能力去做它而且做得很好。一塊使人目眩的快速 FORTH 電腦板，也許偏重繪圖應用，點像顯示器繪圖。

我曾建過一部 FORTH 電腦稱爲 BLUE 。它很小。它尚未執行過任何 FORTH 。設計改變的速度，正如晶片被插入板子上一樣，但是它並不難做。

FORTH 電腦的特點爲何呢? 它不需要大量的記憶體。 16K 就行了。或許一半爲 PROM ，一半爲 RAM 。它不需要很多 I/O 出入口----除非使用程式需要否則它不需要任何 I/O 出入口。一個串連的線路很好；一個磁碟出入口也很好。我們曾將 FORTH 放在 8080 上，其磁碟用足夠記憶蕊來代替以容納 8 個塊(Blocks) 。它生存的非常好，沒有系統崩潰和受保護的環境的特殊問題。磁泡記憶體來臨了，當然還有 Winchester 的硬碟機 (Drives) 。我們不需要很多的大量記憶體，我們只需要 100 到 250 塊相同的記憶體。 FORTH 可以很愉快地存在，依現代標準來說很小的機器內的事實應該被好好開發利用。

組織

最後，我想略述一遍與 FORTH 有關組織的歷史。他們形成了編織綿畫中的另一條線。他們是 Mohasco 和 National Radio Astronomy Observatory 。 他們生下它卻又拒絕了它。這就是爲什麼今天我站在這裏而不是在南非設計望遠鏡程式。他們有我們學過認定爲 NIH 美國國家衛生器的併發症狀，但是在一種怪異的變化情況下，因爲它被發明在該地。這是他們的損失。你們也許讀過有關 VLA，新墨西哥州的一個很大的天線陣列----一個很刺激的計畫。這是我很願意

爲它設計程式的東西。它並不是在卡片上(它們今天有了軟體的問題)。另一方面，有 Kitt Peak 。天文學家是很保守的一群人。這也許很令人吃驚。我認爲在保守程度上只有核子物理學家可以超過他們。我們尚未能觸及核子物理的領域，雖然我聽說 CERN 很感興趣。NRAO 是 Brookhaven 的姊妹實驗室。人們會認爲他們之間會有些交流，但是他們沒有。他們都由同一個大學協會所管轄。我們引不起 Brookhaven 的興趣，我們也引不起 NRAO 的興趣，但是我們可以引起 Kitt Peak 的興趣，這是由 Elizabeth Rather 做到的。她喜愛 FORTH ，也告訴許多其他的人去喜歡它。Kitt Peak 採用了 FORTH ----給予了它原動力，因爲 Kitt Peak 是天文世界的名勝地。Kitt Peak 把 FORTH 放在一整群 Varians 上。這很有趣，因爲我記得 Varians 故障了，而備用電腦在運轉---- 14 台在大廳中連線。經由重複的可靠性? 輸油管的建築? 許多其他的天文台，也剽竊它的模式。

有一陣子天文學家對 FORTH 系統的要求湧至，我們想要開發這個市場來做生意。這個市場我們現今仍在做，但是世界上的望遠鏡很有限，你無法以這個市場來維持一個公司。

FORTH 公司的成立非常重要，因爲我認爲如果沒有 FORTH 公司，今天我們就不會在這裏了。我們工作得很動奮去賣這樣東西。不知道從事的是什麼；我們是你的標準的純真的小生意伙伴。我警告你們任何人都不要認爲自己很容易涉入商業界。它很有趣，但是勸告卻是真的----不要涉入你必須爲產品創造需求的區域。但是這已是 FORTH 真正面臨的最小的問題了。下一個階段或許是 DUCUS 。Marty Ewing 把他的 PDP-11 FORTH 系統給了 DECUS 。我不知道當時這是否是個好主意----自由 FORTH 各處流動。它很重要因爲有許多人因此而接觸到 FORTH 。經由這個管道我們獲得了一些銷路。 Cybek 來了。 Cybek 可算是 FORTH 有限公司的救星，它提供了賺錢的生意，當時我們正亟需依此維生。Cybek 的總裁 Art Gravina 是設計 (如果可以這樣說的話) 我們的資料庫管理系統的人。我們可以爭論誰做了些什麼事，但是 Art 提供這個機會去做商業系統。他得到很好的交易，因爲他使用 FORTH 比以前用 BASIC 程式可以多處理十倍的終端機，我們從他那兒學到了所有資料庫管理的知識。我們也在那時染上對商業程式設計的厭惡。我很讚賞你們這些涉入其中的人。我發覺對我的經驗來說系統分析是太棘手了。你必須要走進那裏去告訴生意人他必須做些什麼，說服他去做，並且在整個裝置的過程中使他不要插手。這和寫程式是兩種不同的才能。不要低估了維持公司的費用。

我認爲現在該是國際天文聯盟會面，並通過 FORTH 爲標準語言的時候了。這是天文學界對 FORTH 的聲援，雖然天文學界在 FORTH 的流行上，已不再是主要的驅動力量了。我想 EFUG 就在此時出現了----大約是 1976 年左右----歐洲的 FORTH 使用者俱樂部(European FORTH Users Group)。它出現了，令我感到意外，歐洲是 FORTH 活動的溫床，對此我們以前大都未注意到，而且或許現在仍然未注意到；我們並未涉入那個世界，而且並不十分欣賞他們興趣的層次。FST (FORTH Standards Team)大約就在 EFUG 的第一次會議上開始的。稍後，幾

年以前，FIG 創立了，現在我們有了 FORML
(FORTH Modification Laboratory) ，這是一個構想醞釀的組織。現在的趨勢似乎是讓人們自己組織成團體。這些組織有的是公司，有的是協會。看來 FORTH 將會成爲形同無組織的志趣相投者群集的共同活動。這暗示的是整個 FORTH 世界將成爲十分無組織，無中心，無控制。它並不壞，也許還很好。

結論

用一句哲學的註解來作結論：事在人爲。這是從草根孕育的最初的語言。這是換用青銅之前在經驗之石上，磨過的最初的語言。我不敢說它是完美的。我要說如果你從 FORTH 中取走任何東西，它就不再是 FORTH 了，我們所知的基本構成要素對此語言的發展能力都是很重要的。如果你沒有大量記憶體，你就有問題了，而且它不會被揮手趕走。我不願去預測。我不知道將會發生什麼事。我想我自己今晚對未來的觀點比幾年來的觀點要猶豫些。前途有望，是的，混亂而且複雜。現在的暗示也許比十年前猶豫。實現的希望要高得多。這是 FORTH 的十年。

我原本的目的是要在我一生中至少寫 40 個程式。我想我已增加我的生產量，以 10 的係數增加。我不認爲那種生產量是會被程式語言限制，所以我已完成了我預定要做的事。在我手中有一種非常有效率的工具----看來它似乎在別人的手上也很有效率。我很高興也很驕傲這是真的。

但願未來對你及你所致力之事展露笑容。

(CHARLES MOORT 先生的演講結束於持續的起立喝采聲中。)

附錄(s)-forth 問題練習

1. 疊層指令和算術指令

(1)、寫一個語句，使疊層上面三個數值換位置，把中間的數字留在中間；
亦即， a b c 變成 c b a (S-p047)

(2)、寫出一個語句，能夠做 OVER 所能做的，卻不用 OVER 這個字。(S-p047)

(3)、寫出一個名爲 -ROT 的定義，它可以用 ROT 相反的方向迴轉疊層最上面的三個數值；亦即 a b c 變成 c a b 。 (S-p319)

(4)、替下列的方程式寫出定義，疊層效果如括弧中所示： (S-p047)

a. n+1/n (n -)

b. x(7x+5) (x -)

c. $9a^2$-ba (a b -)

2. 將下列間置式改爲後置式： (符式 F-83 入門)

(1) 75+31-18

(2) a*(b+c)

(3) $3x^2+7xy-5y^2$

(4) 13*79/100

(5) 0.5*〔(a-b)/(c-d)〕

註：符式不用浮點算數，上述 0.5 可用 1/2。

3. 把下列後置式改爲間置式者。 (符式 F-83 入門)

(1) 2 5 + 3 *

(2) R M M + /

(3) 7 2 -13 0.5 + /

(4) 5125 6 100 */ 3 -

(5) m c c * *

4. 寫一個語句，能使疊層最上面四個數值的順序顛倒過來；亦即
(1 2 3 4 -- 4 3 2 1) (S-p319)

5. 寫出一個 3DUP 的定義，使其能複製疊層最上面的三個數值；例如，
(1 2 3 -- 1 2 3 1 2 3) (S-p320)

6. 寫出下列中位法方程式的定義，其疊層效果如括弧中所示： (S-p320)

(1) a^2 + ab +c 已知 (c a b --)

(2) a-b/a+b 已知 (a b --)

7. 三角形面積的公式是 b*h/2 ，b 是底，h 是高，h 與 b 成直角定義一個三角形 TRIANGLE 字令，只要賦予其高和底，就能計算出其面積。 (M-p228)

8. 利用 DUP、DRUP、SWAP、OVER、PICK、ROT、-ROT 和 ROLL 來定義下列的疊層操作指令： (M-p229)

3DUP	(n1 n2 n3 -- n1 n2 n3 n1 n2 n3)	同時複製層頂端的三個數字。
NIP	(n1 n2 -- n2)	拋棄疊層的第二個數字。
TUCK	(n1 n2 -- n2 n1 n2)	複製疊層頂端的數字，再將複製的數字塞到原先第二個字底下。

如果 3DUP 的定義中應用 2DUP、2DROP、2SWAP、2OVER 或 2ROT ，對定義本身有無改進?

9. 將下列的代數式翻譯成 Forth 定義 : (S-p321)

已知 (a b c --)

10. 已知疊層上有四個數字寫出一個能印出最大值的指令 。 (S-p321)

(6 70 123 45 --)

11. 圓錐體積的公式爲 (π * r^2 * h) / 3 請寫出 CONE 的定義

(r 爲半徑, h 爲高) CONE (r h -- v) (M-p233)

12. 溫度變換變換公式 : (S-p321)

C°= (F°- 32) * 5 / 9

F°= C°* 9/5 + 32

K°= 273 + C° (實際爲 k °= 273.16 + C°)

a. 將此三公式寫爲後置式 。

b. 定義下列詞群，如下例 :

: F > C (f°-- C°) 32 - 5 9 * / ;

(a) C > F

(b) K > C

(c) F > K

(d) K > F

(e) C > K

c.用一些實際數字測試上述定義。

d.學了字串輸入輸出後，將上式 b 中之定義修改，使之能用(假)的小數點算出。只要小數點以後兩位就可以了。輸入者並需查考是否小數點以後於二位，若用者忘了小數點及以下兩位數，必需自動補入計算。(S-p322)

13. 寫出一個 growth? 字令,這個字令會決定如果一個數字(以 1000 爲例)每年成長 "15 % " ，要多少年才會加倍。 該指令應該如下: (M-p233)

```
15 growth ? 5 ok
```

1000 之成長如右 ：1150 , 1322 , 1520 , 1748 , 和 2010 。

14. 首先寫一個指令名爲 stars ，它可以印在同一行中印出 n 個星號已知 n 在疊層上: (S-p322)

```
10 stars  * * * * * * * * * *  ok
```

15. 再定義 BOX ，它可以印出呈矩形的星號，已知寬和高(行數)，用下列的疊層順序 (width height --)。 (S-p322)

```
10 3 box
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * * ok
```

16. 現在寫出一個指命令爲 \STARS ，會印出歪斜的一列星號(菱形)已知高在疊層上 。 用一個 DO 環路, 並且爲了簡單起見，讓寬保持爲 10 個星號 。 (S-p322)

```
3 \stars
 * * * * * * * * * *
   * * * * * * * * * *
     * * * * * * * * * * ok
```

17. 現在寫出一個指令，它使星號朝另一方向傾斜；稱它爲 /STARS 它應該把高當做疊層輸入，並且用 10 爲其寬。使用一個 DO 迴路。 (S-p322)

18. 現在重新定義 17 題的指令，用一個 BEGIN.....UNTIL 迴路。 (S-p322)

19. 寫一個名爲 DISAMONDS 定義，它會印出指定數目的菱形，如此例所示 :
(S-p322)

20. 寫一個指令來幫助法官對罪犯判決監禁，如此法官可以輸入： (S-p320)

```
CONVICTED-OF ARSON HOMICIDE TAX-EVASION OK
WILL-SERVE 35 YEARS OK
```

或者任何其他罪行，由 CONVICTED-OF 這個指令開頭，WILL-SERVE 結尾。使用下面這些判決：

HOMICIDE	(殺人)	20 年
ARSON	(縱火)	10 年
BOOKMAKING	(賭業)	2 年
TAX-EVASION	(逃稅)	5 年

21. 假設你是 Maria's Egg Ranch 的盤存程式員。定義一個指令 EGG.CARTONS 它可以在疊層上留有今天雞所生蛋的總數，並且印出箱數 (每箱可裝 1 打)，以及剩餘雞蛋的數目。 (S-p320)

22. 現在你已學到運算指令之比較，邏輯和流程的控制，請給 MAX 和 MIN 下定義。 (M-p231)

23. 利用 0< 和 0= ，來給 0> 和 0<> 下定義。 (M-p231)

24. 利用現有的邏輯運算指令 AND 、 OR 、 NOT 來定義一個新的邏輯運算指令 NAND ，如果兩個運算都爲真，NAND 會送回不真值，否則會送回真值。 (M-p231)

25. 利用 NAND 來定義其他三個邏輯運算指令 AND 、OR 和 NOT 。再利用這些指令來定義一個"互斥 OR"的 XOR 邏輯運算指令： (M-p231)

提示： XOR 的公式如下，其中 A，B 是布林旗號。

```
([(not-B) and A] or [(not-A) and B])
```

26. 在此有一個猜數字的遊戲要你設計程式。電腦從 1 到 100 中選一個秘密數字，由你來猜。每猜一次，如果猜的數字比前一次猜的數字更接近秘密數字，電腦會回答 " WARMER " ，否則就回答 " COLDER " 。如果你猜的數字離秘密數字不到 2，則會回答 " HOT "，如果猜對了，電腦會告訴你猜了多少次。這程式的功能如下： (M-p232)

```
GAME  OK ( 設定一個秘密數字 )
10 GUESS COLDER  OK
80 GUESS WARMER  OK
75 GUESS HOTI  OK
73 GUESS
YOU WON IN 4 GUESSESI  OK
```

27. 定義一個指令 SIGN.TEST ，試驗疊層上的數字，並印出下列三個訊息之一：

```
POSITIVE    or
ZERO    or
NEGATIVE
```

28. 定義 BETWEEN (n high low -- f) 。high > n > low (S-p320)

29. 寫出一個指令 WITHIN 的定義，它有三個變數： (S-p320)

```
( n low-limit hi-limit -- )
```

並且唯有“ n ”在下列的範圍內才會留下一個「真」的旗號：

```
low-limit <= n < hi-limit
```

30. 利用上題中 WITHIN 的定義，來寫另一個猜數字遊戲，稱爲 TRAP 。你先輸入一個秘密的值，然後另一個人試著在兩個數字之間去捕捉此值。如以下所示： (S-p321)

```
0 1000 TRAP BETWEEN OK
330 660 TRAP BETWEEN OK
440 550 TRAP NOT BETWEEN OK
330 440 TRAP BETWEEN OK
```

一直到玩遊戲的人猜出答案爲止：

```
391 391 TRAP YOU GOT IT! OK
```

31. 下面是一個猜數字遊戲。你可能會很喜歡。首先，你秘密地輸入一個數字到疊層上 (你可以在輸入數字之後執 PAGE 這個指令來清除此數字，PAGE 會清除終端機螢幕) 。然後你請其他玩遊戲的人輸入一個猜測的數字，後面接 GUESS 指令，如 (S-p320)

```
100 GUESS
```

電腦會送入回答「太高」(too high),「太低」(too low) ，或「答對了！」(correct!) 寫出 GUESS 的定義，要確定原來答案的數字經過一再地猜測之後仍然會留在疊層上，直到猜出正確的答案爲止；到那時疊層就應該被清除乾淨。

32. 用一組套裝的試驗(nested tests)以及 IF ... ELSE ... THEN，寫出一個 SPELLER 的定義。它可以拼出它在疊層上找到的數字，從 -4 到 4 ，如果數字超過此範圍，它就會印出 "OUT OF RANGE" 例如： (S-p321)

```
2 SPELLER TWO OK
-4 SPELLER NEGATIVE FOUR OK
7 SOELLER OUT OF RANGE OK
```

使此定義愈短愈好。(提示： FORTH 指令 ABS 將一個疊層數字改成它的絕對值)

33. ACME Pack-Me 公司專門將產品打包成一盒一盒的。如果無法找到適當尺寸的盒子，就可用大一號的盒子。各種盒子的尺寸從 0 編到 9 。下面 BOXES 矩陣就是手邊盒子的數量。 (M-p232)

```
CREATE BOXES
3 , 2 , 0 , 4 , 0 , 1 , 4 , 2 , 2 , 3 ,
```

換句話說，目前 0 號尺寸有三盒，1 號尺寸有二盒，以此類推。請定義一個 BOX? 字令，當提及一個盒子的尺寸，應能送回真正可用的尺寸，然後將盒子的數量減 1 。如下：

```
5 BOX?  5  OK  (用 5 號盒)
5 BOX?  6  OK  (5 號盒用完，用 6 號盒)
```

如果根本沒有合用的盒子， BOX? 字令也會有所說明。

34. 寫一個 HISTOGRAM 字令，以統計條形圖的方式顯示矩陣中的元素。條形圖字令預期疊層上有開始的位址和矩陣中數元的數值。每一數元就印一行星號，星號的數量等於數元中的數值。如果應用 HISTOGRAM 字令會到上一題習題中的 BOXES 矩陣，你會看到： (M-p233)

```
BOXES 10 HISTOGRAM
***
**

****

*
****
**
**
*** OK
```

將星號的數量限制到一個合理的數值。

35. 中國人有十二生肖，用一種不同的動物代表一年，每 12 年循環一次。例如， 1900 年出生的人是生於「鼠年」。基於出生年的算命術就被稱爲“Juneeshee”。 (S-p328)

下面是此循環的順序：

鼠 牛 虎 兔 龍 蛇 馬 羊 猴 雞 狗 豬

寫一個名爲 .ANIMAL 的指令，它可以依照這些動物在循環中的次序打出它們的名字。例如

```
0 .ANIMAL RAT OK
```

現在再寫一個(JUNEESHEE)的指令，它根據給它的年數印出相關動物的名稱(1900 是鼠年，1901 年是牛年等等)。

最後，寫一個 JUNEESHEE 的指令，它可以請使用者入其出生年後即印出此人的生肖動物的名字。使用者在輸入年份之後不需要按" return "。

36. 定義一個數值列印指令，名爲 M.，它可以印出一個有小數點的雙整數。在此數中的小數點位置可以依據一個變數的值而移動，將此變數定義爲 PLACES 。例如，如果你在 PLACES 中存入〝1〞，就會得到 (S-p324)

```
200,000 M.20000.0 OK
```

亦即、小數點後面一位數。在 PLACFS 中存入 0 的話就不會產生小數點了。

37. 爲了要記住你辦公室中彩色鉛筆的存貨，你可以創造一個數陣，此數陣中每一個字元包含了一種顏把的筆的總數。定義一組指令，RED PENCILS 當鍵入此指令時返回紅色鉛筆數目存放的地址，定義完成後，將各彩色鉛筆數目存入陣列內如： (S-p324)

```
RED PENCILS
23 red pencils
15 blue pencils
12 green pencils
 0 orange pencils
```

38. 一個柱狀圖是一串數值的圖形表示法。每一數值由一條柱子的高度或長度來表示。在本練習中，你必須創造一個數值數陣，並且把每一數值用一行 〝*〞 號表示的柱狀圖印出來、首先創造一個約 10 個字元的數陣。給予該數陣每一字元一個初值(其範圍由 0 到 70)。然後再定義一個指令 PLOT，它可以把每一數值用一行印出。在每一行上印出字元的數值，其後並接著一串相等於該字元值的〝* 〞號。

例如，假設該數陣有 4 個字元，分別爲數值 1、2、3 和 4 於是 PLOT 會產生以下的結果： (S-p325)

```
1 *
2 **
3 ***
4 ****
```

39. 寫一個應用程式，可顯示出一個＃字棋盤，使兩個玩此遊戲的人可以從鍵盤上輸入他們的棋子。例如，以下的指令句 (S-p326)

```
4  x!
```

會將一個〝x〞放在第 4 格(從 1 開始算) 並印出

```
          |    |
          |    |
      ----|----|----
        X |    |
      ----|----|----
          |    |
          |    |
```

然後

```
3 0!
```

會放一個〝 0 〞到第 3 格，並印出

```
          |    |
          |    |  0
      ----|----|----
        X |    |
      ----|----|----
          |    |
          |    |
```

用一個數元數陣來記錄此棋盤的內容， 1 代表"x" ，-1 代表" 0 "，Φ代表空格。

(註：在我們詳細解釋「詞彙」之前，切勿將任何指令定名爲" x "，因爲它可能會編輯程式中的 x 混淆。)

40. 在這個練習中，你將建造並使用一個虛陣(virtual array)，亦即是一個存在於磁碟中的數陣；但用起來卻像一個在記憶中的數陣。用 @ 來取並用 ! 來存。 (S-p329)

首先從你那些指定的塊中選出一個沒有被使用的塊。在此塊中不存文字，只存二進位的數值資料。把此塊的號碼放入一個變數中，然後定義一個擷取指令，此指令由疊層上接收數值的序數。再根據這個序數計算此塊的號碼。用 BLOCLK 送回這個數值的記憶地址。這個擷取指令用

UPDATE 更新磁碟暫存區。你可以先試做這一部份看看。

接下來再把每塊的第一個數元當做此數陣中所儲存數值資料的總數。定義一個 PUT 指令，它可以將一個值儲存到此數陣的下一個可用地址中。另外定義一個顯示指令，用來印出此數陣中儲存的所有數值。

現在利用此虛數陣來定義一個指令 ENTER ；這個指令會接受兩個數字並將它們存入數陣中。

最後，定義 TABLE ，用來印出數陣內全部資料，每一行印 8 個數字。

41. 寫一個 <CMOVE> 字令，這字令不像 CMOVE 或 CMOVE> ，即使原始位址和目標重疊，這字令也能安全地將一個字元陣列在記憶體上下移到。(M-p234)

42. 利用 CONVERT 指出，寫一個 NUMBER 字令，這個字令會將一個字串轉換成一個雙整數。如果字串中出現一個小數點，其位址就被存在 DPL 變數中(DPL 是小數點位址)，如果沒有小數點，DPL 就訂爲"-1" 。 (S-p324)

```
"12345"  NUMBER D. DPL  ? 12345 -1  OK
"12345." NUNBER D. DPL  ? 12345 0  OK
"12.345" NUMBER D. DPL  ? 12345 3  OK
```

43. 定義一個 LEX 字令，這字令會按指定的界限將一個字串分成兩部份。這界限一除去，這兩個字串就變成空白(nullh)。 LEX 字令在將字串分成有意義的副字串時很管用。 (M-p235)

```
" VOLUME:NAME" ASCLL :LEX  O
TYPE NAME   OK    ( right string)
TYPE YOLUME   OK  ( left string)
```

44. 應用轉換指令 <#、#、#S、SIGN、HOLD 和 #>，定義 D. 字令，來在每 3 個數字中插入一個逗點，從右邊起。你新字令 D.會做得像這樣: (M-p235)

```
1234567. D. 1,234,567  OK
```

45. 定義一個指令名爲 **，用來計算冪數的值，如下： (S-p322)

```
7 2 ** . 49 OK     (7 的平方)
2 4 ** . 16 OK     (2 的 4 次方)
```

爲求簡單起見，只假設正冪數(但是要確定當指數爲 1 時，** 仍然很正確地工作，----結果爲此數本身)。

46. 溫小姐記不住單整數的最大極限，她也沒參考書可查，只有一台 FORTH 終端機，因此她寫了一個名稱爲 N-MAX 定義。使用 BEGIN...UNTIL 迴路。當她執行此定義時得到 (S-p322)

```
32767  ok
```

47. 寫一個定義使你的終端機上的鈴響三次。要注意必須使鈴聲間歇一下，才會分辨得清楚，每一次鈴響，終端機螢幕上就會顯示出" BEEP "字樣。 (S-p323)

48. 寫一個指令來計算下面的二次方程式 (S-p323)

$7x^2 + 20x + 5$

給一個 x 值，並送回一個雙整數的結果。

49. 寫一個電話號碼形式指令的定義，它同時可以印出區域號碼，後面接著一條斜線(/ a slash) (如果此電話號碼有區域號碼的話才會印出。)
例如， (S-p324)

```
555-123 . PH# 555-1234 OK
213/372-8493 .PH# 213/372-8493 OK
```

50. 定義一詞，以將 0 到 15 列表以 2、8、10、16 四種底數表列出來。(S-p323)

51. 定義 .BASE ，以將現有底數，以十進位表示。 (S-p324)

53. 定義 DUPx ，x 爲在某底數下合法之數字，用以指堆疊上第 x 數目，將此數拷貝到堆疊頂 。x 自開始。如: (符式 F-83 入門)

```
DUP5 ( 6 5 4 3 2 1 -- 6 5 4 3 2 1 6 )
DUP3 ( 6 5 4 3 2 1 -- 6 5 4 3 2 1 4 )
```

54. 依上例想法，定義 DROPx 。 (符式 F-83 入門)

```
DROP4 ( 6 5 4 3 2 1 -- 6 4 3 2 1 )
```

DROP2 (6 5 4 3 2 1 -- 6 5 4 2 1)

以上兩題，在詞名內傳遞單位數值。

55. 依上述想法，定義 SWAPxy ，以交換 x 項與 y 項參數，x 與 y 均自 0 開始，如: (符式 F-83 入門)

SWAP25 (6 5 4 3 2 1 -- 3 5 4 6 2 1)
SWAP14 (6 5 4 3 2 1 -- 6 2 4 3 5 1)

56. 定義下列輸出規格詞。它們均以雙精數爲度。(d ---) 。(符式 F-83 入門)

	名 稱		格 式
a.	.DATE	如	mm / dd / yr
b.	.PHONE	如	xxx-yyyy
c.	.TIME	如	hr : mn: se
d.	.$	如	$ xnnnnnn.mm

其中 x : 爲符號。 n : 可有多位。

57. 現在，請定義對應於上述四詞之輸入詞群。使用這些詞的結果爲雙倍精密度數字 (--- d) 。 (符式 F-83 入門)

	名 稱		格 式
a.	?DATE	如	mm / dd / yr
b.	?PHONE	如	xxx-yyyy
c.	?TIME	如	hr : mn: se
d.	?$	如	$ xnnnnnn.mm

58. 試用 RECURSIVE，重新設立 SEE 的功能，使之將某冒號定義，一層一層分解下去，到低階定義，常數，變數等不能再分解之詞爲止。如 (符式 F-83 入門)

```
SEE  MULTI-LAYER
  :  MULTI-LAYER
                ---
                 ---
                  ---
                  ---
                 ---
```

```
   ---
     ---
    ---
   ---
   ---
  ---
```

59. 試用組合式定義下列各高階定義之詞。 (符式 F-83 入門)

```
: BOUNDS  ( addr n -- addr + n  addr )
    OVER + SWAP ;

: 3DUP  ( a b c -- a b c a b c )
   3  PICK  3  PICK  3  PICK  ;

: 4DUP  ( a b c d -- a b c d a b c d )
   2OVER  2OVER ;
```

60. 試用低階定義下列高階定義。 (符式 F-83 入門)

```
DUP   ( n -- n n )
SWAP  ( a b -- b a )
ROT   ( a b c -- b c a )
OVER  ( a b -- a b a )
```

61. 試用多功能做河內塔的工作。即每個背景工作負責一項搬移的事都由前景工作，一個人負擔太重，大家分擔分擔。

62. 試建立一個你自己喜歡的編輯器。

附記：

一、上述 (S-p324) (M-p223) (符式 F-83 入門) 等記號，爲本例題之出處，看官們可自行參閱該書。

S- 爲 Strting Forth 一書。M- 爲 Masteving Forth 一書。

二、各練習題之答案可能不只一種看官們可全部列出後，找出其最佳化。

附錄(t)FORTH 參考書目

一、Brodie,L. 1987. Starting Forth.Prentice-Hall,Englewood Cliffs,N.J. Anintroductory text covering polyForth with some mention of differences from FIGForth and Forth-83 Assumes no previous knowledge of computers. Easy to read , with a very breezy style. Limited coverage of advanced topics. Limited examples and exercises.Inadequate index. The first general text on Forth and for long the most popular.

二、Brodie,L. 1984. Thinking Forth.Prentice-Hall,Englewood Cliffs,N.J. The thought prosesses and philosophy of problem solving in Forth. Discussion of program implementation with many examples, tips and programming procedures. Very thought provoking.

三、Cassady,J.J. 1981. MetaForth. Distributed by Mountain View Press, Mountain View, Calif. Source listings for a metacompiler (target or cross-compiler) written in FIG-Forth.

四、Haydon, G. 1991. All about Forth. Mountain View Press, Mountain View, Calif. Definitions, discussions, and examples of the use of all the words in MVP-Forth. Also compares MVP-Forth's words with Forth-79 and FIG-Forth and F-PC. Very useful for understanding how the language works.

五、Tracy, Martin 1989. Mastering Forth. Brady Books.

六、Kelly, G.Mahlon 1986. Forth A Text and Reference. Prentice. Hall.

七、Woehr, J.Jack 1992. Seeing Forth. Offete Enterprises.

八、Loeliger R.G. 1981. Threaded Interpretive Languages. BYTE Books.

九、Pountain, Dick 1987. Object-Oriented Forth. Academic Press Berkeley.

十、Redmond, Loren 1988. Forth 83 CODE-OPTIMIZE. Concept 4.

十一、Ourerson, Marlin 1987. DR.Dobb's Toolbool of Forth. M&T Book.

十二、Koopman Jr. Philip J. 1989. STACK Computer the new wave.
ELlis Norwood Limited.

十三、Kami, N Samuel 1990. Programming Langnages An Interpreter-Based
Approach. Addison Wesley.

附錄(u)中華民國符式學會簡介

一、FORTH 語言協會爲一非營利性的機構，其目的在促進國內資訊科技的發展。

二、本協會由愛號 FORTH 語言者組成，舉辦定期及不定期研討會，交換有關 FORTH 軟硬體之資訊。

三、組織：

(一)本會設會長一人(任期一年，不得連任)、幹事(由會長指定)。

(二)會長可視情況需要，邀請會友成立委員會推展業務。

(三)會員資格：凡愛好 FORTH 語言，贊同本協會宗旨，願履行會員義務者。

(四)會員種類：(A)名譽會員。 (B)贊助會員。 (C)團體會員。
(D)個人會員(一般會員，學生會員)。 (E)永久會員。

四、會員權利：

(一)凡會員都可定期或不定期收到 FORTH (A)現有資料目錄；(B)會務活動狀況；(C)FORTH 庫存程式目錄；(D)會友通訊錄；(E)財務簡報。

(二) FORTH 會友可以互助解決有關建立 FORTH 系統及設計應用程式等方面的問題。

(三)會員可參加每月定期的研討會及一些不定期的演講會。

(四)會員皆可參加協會舉辦的 FORTH 程式競賽。

(五)會員參加本協會所辦的 FORTH 語言研習會，享有折扣優待。

五、會員義務：

(一)協助學會推廣 FORTH 語言。

六、定期研討會：

時　　間：每月第一週的週六上午九時起至中午十二點三十分止。

地　　點：中原大學工學館 501 室。

詳細時間地點請至 FIG-Taiwan 網站查詢。

中華民國符式語言協會 http://www.figtaiwan.org/

聯 絡 人：陳 爽 先生 sschen@iner.gov.tw

七、不定期演講會：時間、地點於一週前通知。

八、報名入會者逕寄：陳 爽 先生 sschen@iner.gov.tw

身是菩提樹 | 硬體雜又難
心如明鏡臺 | 軟體學不完
朝朝勤拂拭 | 日夜埋首幹
莫使惹塵埃 | 心酸又心煩

菩提本無樹 | 硬體具實觀
明鏡亦非臺 | 程式爲思串
本來無一物 | 符式由中穿
何處惹塵埃 | 人機原一貫