第7章　高階語言

在本章中，我們將介紹高階語言的語法、語意與執行平台，以便讓讀者建立高階語言的基本概念。然後，我們將在下一章中，介紹編譯器這個主題，這是高階語言最重要的工具，可以將高階語言轉換為組合語言。

在本書中，我們已經透過組合語言，說明了機器指令與電腦架構等概念，但是，讀者應該可以感覺到這樣的程式撰寫方式相當原始。如果要能更快速的寫出程式，提升程式人員的生產力，必須提高語言的層次，這也就是高階語言的目的。

除此之外，由於高階語言的語法和機器通常不相關，因此，可攜性比組合語言好很多，這使得程式設計師所寫的高階語言可以在各種不同的電腦上執行。這兩個好處使得程式設計師會傾向於使用高階語言，而盡量避免使用組合語言。

在本書前半部的 2-6 章當中，已經說明了組譯器、連結器、載入器、巨集處理器等主題，這些系統軟體都是撰寫組合語言時所需要用到的工具，因此前半部的焦點專注在組合語言上。

在本書後半部的 7-11 章當中，將會說明高階語言的系統軟體 – 直譯器與編譯器，以及高階語言的執行平台 – 虛擬機器、作業系統與嵌入式系統等主題，因此高階語言將會是本書後半部的焦點。

在本章的後續小節中，我們將按照語法理論 (7.2 節)、語意理論 (7.3 節)、執行環境 (7.4 節) 等三個主題，說明高階語言的理論部分，然後在下一章中，則按照編譯器的階段，詞彙掃描、語法剖析與程式碼產生，說明如何將程式編譯為組合語言。

7.1 簡介

高階語言的設計一直是程式設計人員關注的焦點，從 1960 年代以來，人們不斷發明新的程式語言，然而，這些語言就好像流行服飾一般，不斷的推陳出新，這使得目前已知的程式語言達數百種之多，沒有任何人能熟悉所有的高階程式語言。

為了說明常見的高階語言之間的關係，我們在圖 7.1 當中列出了高階語言的發展

歷史年表，以便讓讀者對現今的高階語言能有一個整體性的概念。

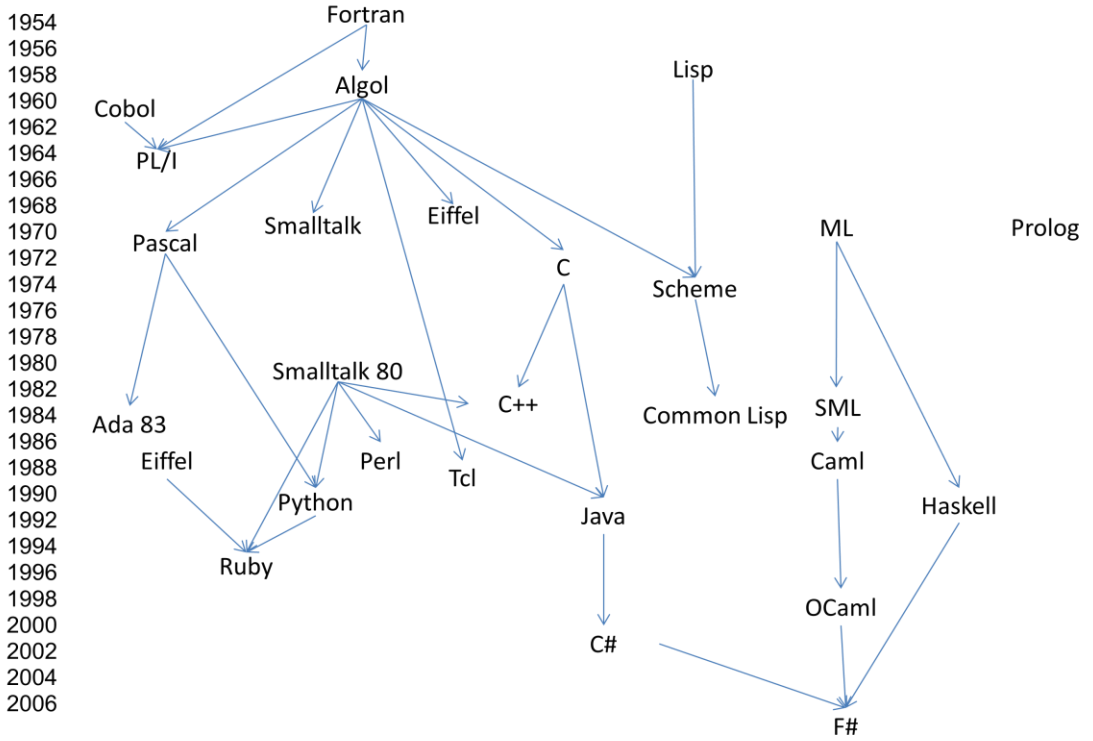


圖 7.1 高階語言的歷史年表

程式語言的多樣性對程式設計師而言往往是一大困擾，每一種語言通常在特定領域擁有一群支持該語言的程式設計團體，同時也伴隨著一些程式開發工具。這也造成了程式開發工具的多樣性。多年來，一直有人試圖創造最佳的通用程式語言，但最後都只造成了程式語言更為多樣。或許，這樣的狀況會一直持續下去，程式設計師仍然必須隨時準備學習新語言。

在資訊系的相關課程當中，與高階語言相關的課程，包含程式語言 (Programming Language)、正規語言 (Formal Language)、以及編譯器 (Compiler) 等等。這些課程的核心是語法理論，我們可以利用生成規則 (例如：BNF, EBNF 等) 描述程式的語法。一但能正確的描述某個程式語言，就能撰寫該語言的剖析程式，將這些語法轉換成語法樹 (或稱剖析樹)。

一但語法樹建構完成，就可以進行『解譯』或『編譯』的動作。如果我們撰寫程式以解讀該語法樹，並根據節點類型執行對應的動作，這樣的程式就被稱為『直譯器』。但是，如果我們撰寫程式將語法樹轉換為組合語言 (或目的碼)，那麼，這樣的程式就被稱為編譯器。

7.2 語法理論

高階語言所使用的語法，大致上分為兩個層次，在詞彙的語法上會使用 Regular Expression (簡稱 RE)，而語句的層次則使用 Context-Free Grammar (簡稱 CFG) 表示，在 RE 與 CFG 等兩個層次都可以使用『生成規則』描述其語法。

『生成規則』是近代語言學之父的喬姆斯基 (Chomsky) 的所提出的一種語法規則，是生成語法 (Generative Grammar) 理論的基礎，生成語法理論在近代語言學當中具有非常重要的地位，可以說是語言學當中最重要的理論之一。

Chomsky 是個語言學家，提出的生成語法的目的是為了描述人類所說的語言，像是英文、中文等，這種人類所說的語言被稱為『自然語言』(Natural Language)，以便與程式語言 (Programming Langauge) 區分開來。

雖然生成語法是為了描述自然語言而提出的，但是也可以用來描述程式語言的語法，在程式語言的領域，這些生成規則通常被寫成 BNF (Backus–Naur Form) 規則。

BNF 是由 John Backus 與 Peter Naur 所提出的一種規則寫法，這種寫法很適合用來描述程式語言的語法，BNF 的發明人幾乎與 Chomsky 同時發明了生成語法，只是一個在屬於電腦領域，一個屬於語言學領域而已。

我們可以利用少許的 BNF 規則，就能描述變化無窮的語句結構，這種化繁為簡能力，是語法理論的精隨所在。BNF 規則可以適用『程式語言』領域，完完全全的描述整個語言的語法，甚至可以適用在『自然語言』上，但是通常無法完全掌握像英文與中文這樣的語言結構。

為了學習 BNF 語法規則，首先讓我們來看看一個極為簡易的語法，如圖 7.2 所示，其中的 a,b,c,d 等稱為終端符號，而 S, A, B 等稱為非終端符號。

(a) BNF 語法	(b) 生成的語言
S = A B A = 'a' | 'b' B = 'c' | 'd'	L = {ac, ad, bc, bd}

圖 7.2 簡單的生成語法範例

在生成語法規則當中，等號左邊的符號可以被代換成右邊的符號，上述規則當中

的 S = A B 代表 S 符號可以被代換成 A 與 B 的連接，而 A = 'a' | 'b' 則代表 A 符號可以被代換成 a 或 b 字元。於是，若我們選定 S 為起始符號，則由 S 所可能導出的字串就有 L = {ac, ad, bc, bd } 等四種可能性，於是我們稱 L 為這組語法所代表的語言 (Language)。

對於圖 7.2 所描述的範例，可能較難讓讀者聯想到真正的語言上，為了讓讀者能理解 BNF 語法的意義，我們將圖 7.2 當中的符號名稱與內容改變，但是規則的形式不變，我們可以得到如圖 7.3 的語法

(a) BNF 語法	(b) 生成的語言
S = N ' ' V A = 'John' | 'Mary' B = 'eats' | 'talks'	L = {John eats, John talks, Mary eats, Mary talks}

圖 7.3 一個簡單的英語語法範例

在圖 7.3(a) 當中，S 代表句子 (Sentence)，N 代表名詞 (Noun)，V 代表動詞 (Verb)。於是，從 S 符號可以導出 John eats, John talks, Mary eats, Mary talks 等四個英文語句，這也就是當初 Chomsky 發明生成語法的主要目的，描述英文並說明語言的組成方法。

在程式語言當中，表達數學運算式是很重要的能力，以下，就讓我們來看看數學運算式的 BNF 規則，以便進一步說明程式和語法規則之間的關係。

數學運算式乃是數字與加減乘除符號的組合，例如，在 3+5*8-4/6 這個運算式中，就包含了許多數字與符號，其中的符號可能是加減乘除等四種，圖 7.4 顯示了一組很簡單的數學運算式之語法規則。

(a) BNF 語法	(b) 語言的實際範例
E = N | E [+-*/] E N = [0-9]+	3 3 + 5 3 + 5 * 8 - 4 / 6

圖 7.4 簡單的數學運算式語法

在圖 7.4 當中，我們利用符號 N 代表整數，N = [0-9]+ 這一個規則可以用來表示所有的整數字串。其中，括號所框住的部分是候選字詞，因此 [0-9] 代表字元 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 等字元，而加號 + 代表的是這些字元會出現一次以上 (包含一次)。

在圖 7.4 當中，我們利用了符號 E 代表數學運算式，E = N | E [+-*/] E 這樣的語法規則代表了一個數學式可以由單一的整數構成，也可以由兩個運算式透過 [+-*/] 符號連接而成，其中的直線符號 | 代表『或者』的意思。

雖然圖 7.4 (a) 的語法雖然可以產生許多數學運算式，但是，若要作為程式語言的語法，用來製作編譯器，那就會產生相當大的問題。因為圖 7.4 (a) 的語法具有歧義性 (Ambiguous)，也就是同一個運算式可能會被剖析為不同的語法樹。

舉例而言，對於 3-1-2 這個運算式而言，根據圖 7.4 的語法，可以生成圖 7.5 中的兩棵樹狀結構，其中圖 7.5 (a) 是 (3-1)-2 的語法樹，而圖 7.5 (b) 則代表 3-(1-2) 的語法樹。

這兩科語法樹的運算順序不同，導致兩棵樹的運算結果也不相同，在圖 7.5 (a) 當中，(3-1) - 2 的運算結果為 0，但是在圖 7.5 (b) 當中，3-(1-2) 的運算結果卻是 4。

這樣的特性稱為語法的歧義性，也就是該語法會導出不同的語法樹，而且這些語法樹將代表不同意義。

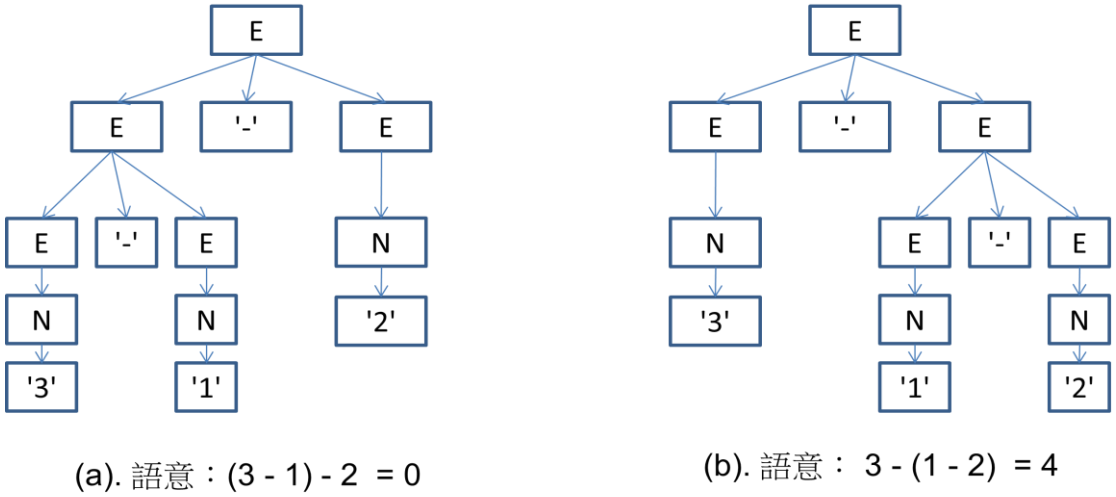


圖 7.5 具有歧義的語法範例

程式語言的語法是不能有歧義性的，否則，程式編譯後有時 3-1-2 會計算出 0，有時卻會算出 4，這樣將導致程式設計師無法確定程式的執行結果，而陷入混亂崩潰的狀況。

圖 7.5 (a) 的語法，必須被修正成無歧義的語法，才能在程式當中使用。圖 7.6 就顯示了該語法的一個修正版本，這個版本雖然較為複雜，但是並沒有歧義性的問題。

(a) BNF 語法	(b) 語言的實際範例
E = T | E [+-] T T = F | T [*/] F F = N | ' (' E ') ' N = [0-9]+	3 3 + 5 3 + 5 * 8 - 4 / 6 (3 + 5) * 8 - 6

圖 7.6 無歧義的數學運算式語法

圖 7.6 (a) 中包含了 E (Expression), T(Term) , F(Factor), N(Number) 等四條規則，透過巧妙的規則設計方式，讓乘除運算的優先順序比加減運算高，並且加入了 '(' E ') ' 這個語法，讓我們可以用括號強制某些運算優先執行。

根據圖 7.6 (a) 的語法規則，我們可以導出 1+2*3 的唯一語法樹如圖 7.7 所示。由於該語法樹是唯一符合規則的語法樹，因此不會導致歧義性的問題。

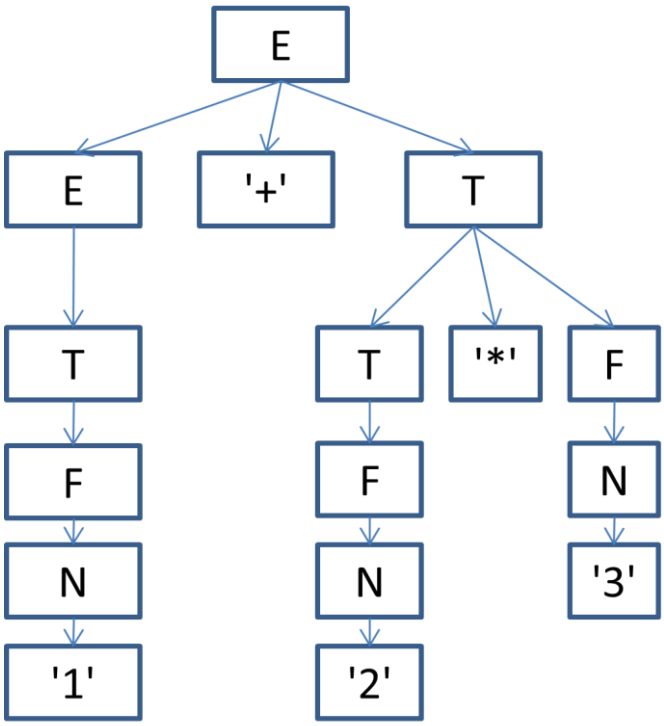


圖 7.7 數學運算式 (1+2*3) 的語法樹

在圖 7.6 (a) **Error! Reference source not found.**的 BNF 語法當中，有一個難以處理的『左遞迴』問題。像是 E = E [+-] T 與 T = T [*/] F 這樣的語法，都包含了『左遞迴』結構。當等號左邊的非終端符號 (E, T)，在同一條規則當中也出現在等號右邊的第一個位置時，就導致了左遞迴的語法。

在編譯器的設計上，左遞迴是相當難以處理的，還好，Pascal 語言的發明人

Nicklaus Wirth 發明了一種 BNF 的延伸語法，稱為 EBNF (Extended Backus–Naur Form) 語法可以處理左遞迴問題。EBNF 可以用來消除大部分的左遞迴，其方法是加入『迴圈語法』用以表示出現數次的意思。

在本文中，我們用 (…)* 符號，代表 … 部分重複比對，其中的星號 * 代表出現零次以上，現就像先前圖 7.6 中的加號 […]+ 代表出現一次以上一樣 [1]。另外，如果在規則中出現 (…)? 這樣的語法，代表 … 的部分會出現最多一次 (也就是 0 次或者 1 次)。

我們可以利用 ()* 符號將圖 7.8 (a) 中具有左遞迴的語法，修改為圖 7.8 (b) 當 **Error! Reference source not found.**中沒有左遞迴的語法，這種用重複符號方式取代左遞迴的語法，就是 EBNF 語法。

(a) BNF 語法	(b) EBNF 語法
E = T | E [+-] T T = F | T [*/] F F = N | ' (' E ') ' N = [0-9]+	E = T ([+-] T)* T = F ([*/] F)* F = N | ' (' E ') ' N = [0-9]+

圖 7.8 將 **Error! Reference source not found.**數學運算式的 BNF 改寫為 EBNF 語法

現在，我們已經具備了足夠的語法理論基礎了，但是語法理論只能讓我們將程式轉換成語法樹，卻沒有告訴我們應該如何解讀這棵語法樹，因此我們需要語意理論，以便解讀語法樹，讓程式真正能夠執行。

7.3 語意理論

語意理論所探討的是語法所代表的意義，也就是某個語法應該如何被執行，或者如何轉換成組合語言的問題。在本節中，我們將專注在結構化程式語言 (像是 C 語言) 的語意問題上，而不去探討其他種類的語言，像是 Prolog 等語言的語意上。

結構化的語意

目前產業界的主流語言是結構化語言，像是 C 語言就是典型的結構化語言，但是自從物件導向技術盛行以來，結構化語言都被加上了物件導向的語法，像是

[1] 在 Nichlaus Wirth 原始的語法中，使用 {…} 代表重複比對零次或以上，但我們採用 ()* 的符號，以替代此種方式，目的是為了與現今常用的 Regular Expression 之語法一致。

C++、C#、 Java、Obj C 等語言，都是將 C 語言加上物件導向功能後的結果。

結構化語言的主要結構有六種，包含『指定、運算、循序、分支、迴圈、函數』等語法，表格 7.1 顯示了結構化程式的構造方式，讓我們來看看這些結構所隱含的語意。

表格 7.1 結構化程式的構造方式

結構類型	語法	範例
指定結構	ASSIGN = ID '=' EXP	x = 3*y+5
運算結構	EXP = T ([+-] <T>)* T = F ([*/] <F>)*	3*y+5
循序結構	BASE_LIST = (BASE)*	t=a; a=b; b=t;
分支結構	IF = 'if' ' (' COND ') ' BASE ('elseif' '(' COND ')' BASE)* ('else' BASE)	if (a>b) c=a; else c=b;
迴圈結構	WHILE = 'while' '(' COND ')' BASE[2]	while (i<=10) { sum = sum+i; i++;}
函數結構	FDEF = ID '(' ARGS ')' BLOCK FCALL = ID '(' PARAMS ')' '; '	定義：max(a,b) {if (a>b) return a; else return b;} 呼叫：c = max(3,5);

指定結構的語法是 ID '=' EXP，其意義乃是將 EXP 的運算結果傳送給變數 ID，於是，ID 變數將會設定為 EXP 的結果。舉例而言，指定敘述 x = 3*y+5 會將 3*y+5 的結果傳送給 x，假如 y 的值為 4，則執行完後 x 的值將變成 17。

運算結構的語法是從數學中借用過來的，基本上 **Error! Reference source not found.**的四個規則就是運算結構的語法。其語意正是數學中的加減乘除之語意，舉例而言，規則 EXP = T ([+-] T)* 定義了加減的語法，而其語意則是將 T+T 或 T-T 的結果傳回。

循序結構的語法是 BASE_LIST = (BASE)*，這代表 BASE 可以連續出現很多次，其意義是循序的執行每個 BASE，例如像 t=a; a=b; b=t; 這樣的語句，其意義是當 t=a 執行完後，接著執行 a=b，然後再執行 b=t。

分支結構可以用 IF = 'if' ' (' COND ') ' BASE ('elseif' '(' COND ')' BASE)* ('else' BASE)

[22] 節點 <BASE> 的語法為 <BASE> ::= id | int | "{" <BASE_LIST> "}"，這種遞迴形式的定義在 BNF 語法當中很常見。在這種遞迴定義方法中，<BASE> 語法可以包含{ sum = sum+i; i++;} 這樣的結構。

這樣的語法表示，其意義是當條件 COND 成立時，就執行對應的 BASE ，如果都不成立，則執行最後的 else 語句。例如像 if (a>b) c=a; else c=b; 這樣的語句，其意義是當 a>b 條件成立時，就執行 c=a，否則，就執行 else 中的敘述 c=b。

迴圈結構通常有 for 迴圈與 while 迴圈，在此我們以 while 迴圈為例，while 迴圈的語法為 WHILE = 'while' '(' COND ')' BASE，其意義是當是當 COND 條件成立時，就繼續執行 BASE 節點，直到 COND 條件不成立才離開迴圈。例如，while (i<=10) { sum = sum+i; i++;} 這樣的語句，在 i<=10 時，會執行 { sum = sum +i; i++; } 區塊，直到 i 大於 10 為止。

函數的語法分為定義與呼叫等兩部分，函數可用 FDEF = ID '(' ARGS ')' BLOCK 的方式定義，其中的 ID 代表函數名稱，ARGS 是參數串列，而 BLOCK 則是函數的內容區塊。舉例而言，像是 max(a,b) { if (a>b) return a; else return b;} 這樣一個函數定義，其中的函數名稱 ID=max，參數串列 ARGS = a,b，而內容區塊 BLOCK = { if (a>b) return a; else return b;}。

函數呼叫的語法為 FCALL = ID '(' PARAMS ')' '; '，其中的 ID 部分為函數名稱，PARAMS 部分則為參數串列。舉例而言，在函數呼叫 max(3,5) 當中，函數名稱 ID=max，參數串列 PARAMS = 3,5。

函數的定義與呼叫兩者，形成了一組『呼叫者/被呼叫者』的語意，舉例而言，max(3,5) 這個語句，與 max(a,b) {…} 這個函數，形成了一組對應關係。語句 max(3,5) 代表將 3 傳給 a, 5 傳給 b，呼叫時會將 PARAMS 當中的引數，傳遞給 ARGS 當中的參數，形成一對一的關係。

上述的『指定、運算、循序、分支、迴圈、函數』等六種語意結構，是結構化程式的基本語意，現今的大部分的程式語言都具備這些結構，而這也六種語意也正好對應到 C 語言中最重要的六種基本語法。

7.4 執行環境

即使有了語法及語意，我們仍然需要將程式語言放入真實的電腦當中，才能夠真正執行。目前常見的執行的環境，大致上可以分為直譯環境 (本節) 和編譯環境 (第 8 章) 兩類，其中的編譯環境又可進一步細分為三種，一種是在虛擬機器上執行 (第 9 章)，一種是在有作業系統的環境中執行 (第 10 章)，另一種是在沒有作業系統的嵌入式環境中執行 (第 11 章)，這三種環境恰好是本書第 9, 10, 11 章的內容。

以下，我們將大略的介紹這些執行環境，以便作為後續章節的基礎，首先，讓我們來看看第一種的直譯式環境，也就是透過直譯器執行的方式。

透過直譯器執行

直譯器是一種可以直接執行高階語言的系統程式，通常在直譯器當中會包含一個剖析器，將高階語言先轉換成語法樹之後，才開始執行這棵語法樹。

當語法剖析的階段完成後，就可以透過直譯器解譯該語法樹，實作 7.3 節中的語意理論。這種方法是利用直譯器建構出一個模仿語意理論的環境，然後利用直譯器的動作，模擬出對應的操作語意。

在上一節的語意理論中，我們曾經看過結構化程式語言的六種語法及語意，現在，我們就針對這六種結構，說明其直譯動作的實作方式，表格 7.2 顯示了這六種結構所對應到的直譯器動作。

表格 7.2 結構化程式的直譯過程

結構類型	語法	直譯器動作
指定結構	ASSIGN = ID '=' EXP	計算 EXP 取得結果後，將結果放入符號表的 ID 變數中
運算結構	EXP = T ([+-] <T>)*	將 T_1 [+-] T_2 …[+-] T_n 的結果，放入 EXP 節點中。
循序結構	BASE_LIST = (BASE)*	循序執行 BASE_LIST 的子節點，$BASE_1$ $BASE_2$ … $BASE_n$
分支結構	IF = 'if' ' (' COND ') ' BASE ('elseif' '(' COND ')' BASE)* ('else' BASE)	檢查條件 COND 節點的值，如果為真，則執行對應的 BASE，若均為假，則執行 else 語句中的 BASE
迴圈結構	WHILE = 'while' '(' COND ')' BASE	當 COND 節點的值為真時，執行 BASE 節點，直到 COND 節點的值為假時，才跳到下一個語句中。
函數結構	FDEF = ID '(' ARGS ')' BLOCK FCALL = ID '(' PARAMS ')' '; '	當呼叫函數 FCALL 時，將 ARGS 參數取代為 PARAMS，然後執行 BLOCK 區塊

語法樹的解譯過程是以遞迴方式進行的，因此，我們可以使用遞迴的方式撰寫直譯器，圖 7.9 顯示了直譯器的演算法，該演算法的參數為一剖析樹的節點，從代

表整個程式的根節點開始，不斷以遞迴下降的方式解譯子節點，直到程式執行完成為止。

直譯器的演算法	說明

Algorithm run(node)
  switch (node.tag) {
  …
  case ASSIGN
  id = node.childs[0]
  exp = node.childs[2]
  SymbolTable[id] = run(exp)
  case EXP
  term1 = node.childs[0]
  run(term1)
  node.value = term1.value
  for (i=1; i<node.childCount; i+=2)
  op = node.childs[i].tag
  term2 = node.childs[i+1]
  run(term2)
  if (op="+")
  node.value += term2.value
  else if (op="-")
  node.value - term2.value
  end if
  end for
  case BASE_LIST
  for (i=0; i<node.childCount; i++)
  run(node.childs[i])
  end for
  case IF
  for (i=0; i<node.chiidCount; i++)
  if (i==0 && node.childs[i].token = "if")
  or (i>0 && node.childs[i].token = "elseif")
  cond = node.childs[i+2]
  run(cond)
  if (cond.value = true)
  base = node.childs[i+4]
  run(base)
  break
	解譯 node 節點 (以遞迴方式)
	判斷節點類型
	
	ASSIGN = ID '=' EXP
	  取出變數
	  取出算式
	  將算式的結果指定給變數
	EXP = T ([+-] T)*
	  取得第一個項目
	  解譯第一個項目
	  設定父節點的值 (運算結果)
	
	  取得下一個運算符號
	  取得下一個運算元
	  解譯下一個運算元
	  如果是加號
	  運算結果 += 運算元
	  如果是減號
	  運算結果 -= 運算元
	
	BASE_LIST= (BASE)*
	  循序的執行每個子節點
	
	IF = 'if' '(' COND ')' BASE 'elseif' …
	  查看每個子節點
	  如果是第一個 if 關鍵字
	  或者是 elseif 關鍵字
	  取得條件節點 [3]
	  計算條件節點
	  如果條件為真
	  取得 BASE 節點
	  執行 BASE 節點

[3] 由於 IF 規則為 'if' '(' COND ')' 或 'elseif ' '('COND ')'，第 i 個如果是 'if'，那麼第 i+2 個將會是 COND，所以此處用 node.childs[i+2] 取得條件節點 COND。

```
            end if
            i += 4
          else if (node.childs[i].token = "else")
            base = node.childs[i+1]
            run(base)
          end if
        end for
      case WHILE
        cond = node.childs[2]
        base = node.childs[4]
        while (run(cond)==true)
          run(base)
        end
      case FCALL
        id = node.childs[0]
        params = node.childs[2]
        fdef = functionTable[id]
        call(fdef, params)
    end switch
End Algorithm

Algorithm call(fdef, params)
  body = fdef.body.replace(fdef.args, params)

  bodyNode = parse(body)
  run(bodyNode)
End Algorithm
```

```
      跳過 'if' '('EXP ')' BASE
    如果是 else 關鍵字
      取得 BASE 節點
      執行 BASE 節點


WHILE = 'while' '(' COND ')' BASE
  取得 COND 節點
  取得 BASE 節點
  當條件 COND 為真時
    執行 BASE 節點

FCALL = ID '(' PARAMS ')' '; '
  取得函數名稱
  取得參數
  取得函數內容
  呼叫該函數


FDEF = ID '(' ARGS ')' BLOCK
  將程式內容取出，並將參數
    ARGS 取代為 PARAMS
  剖析 body 程式
  執行 body 程式
```

圖 7.9 直譯器的演算法

在圖 7.9 的演算法當中，我們對每一條規則進行語法解讀以及語意模擬的程序。舉例而言，當直譯器遇到 ASSIGN 節點時，就會計算 EXP 節點的結果，然後放到符號表中的 ID 變數內，於是 ID 變數的值就會被改變，這種作法可以模擬指定語句 ASSIGN 的語意。

讀者應仔細閱讀圖 7.9 的直譯器演算法，以便理解整個直譯的過程，但要能理解這個演算法，至少必須具備程式設計中的遞迴概念，請讀者自行參考程式設計與演算法的相關書籍，以便理解遞迴的執行過程。

由於直譯器在執行時，程式與變數都存放在記憶體內，而且可以很容易的被直譯器存取，因此直譯器可以在執行時期動態的改變程式與變數的值。舉例而言，我們可以在執行到發生錯誤的中途，透過使用者介面修改變數的值，然後繼續執行程式，達到動態除錯的功能，這讓程式設計師可以一邊執行一邊修改程式。

另外，我們也可以將某個看來像是程式的字串參數，直接利用剖析器展開後掛在某個節點之下執行，這樣就能把參數展開成程式執行，讓程式更為動態，這種技術是編譯器很難達成的功能。

但是，直譯器的缺點是執行速度緩慢，因此，在強調速度的應用上，通常會採用編譯式的方法，因為編譯式的執行速度通常比直譯式的快上幾十倍。

在本章中，我們已經說明了高階語言的語法、語意以及執行環境等理論，但是仍然有些部分尚未完全說明完畢的，我們將在第 8 章當中繼續說明編譯器的設計原理，並且在第 9-11 章當中，說明程式的三大執行環境，也就是虛擬機器 (第 9 章)、作業系統 (第 10 章) 以及嵌入式的環境 (第 11 章) 等主題。

7.5 實務案例

7.5.1 C 語言

1970 年 Dennis Ritchie 和 Ken Thompson 所設計出來的 C 語言，可以說是歷久彌新的語言，很少語言可以和 C 語言一樣，能夠經歷 40 年而仍然被廣泛使用的。 1973 年，Unix 作業系統的核心正式用 C 語言改寫，從此奠定了 C 語言在系統程式上的地位。近代的作業系統，像是 Linux、FreeBSD、Mac OS X 等作業系統，都深受 Unix 的影響，這讓 C 語言成為系統程式中的尚方寶劍，Dennis Ritchie 和 Ken Thompson 也因 C/UNIX 而獲頒資訊科學界的諾貝爾獎 - 圖靈獎 (Turing Award)。

以下我們將利用 C 語言作為範例，分別解說語法、語意與執行平台的設計方式，讓讀者能夠實際感受高階語言的設計原理。

C 語言的語法及語意

C 語言的語法基本上是遵循結構化程式語法的，包含『指定結構』、『運算結構』、『循序結構』、『分支結構』、『迴圈結構』、『函數結構』等。這些結構貫穿了整個語法和語意層面，形成 C 語言的主要語言結構。

基本單元

C 語言的基本單元由圖 7.10 的基本算式 primary-exp 與後置算式 postfix_exp 所構成，像是 x, 35, "hello! ", x[3], f(x), f(), rec.x, rec->x, x++, x-- 等，這兩個算式是所有結構的基礎，因此被我們稱為基本單元。

C 語言的 EBNF 語法 (基本單元)	說明

postfix_exp =
  primary_exp
  | postfix_exp '[' exp ']'
  | postfix_exp '(' arg_exp_list ')'
  | postfix_exp '( )'
  | postfix_exp '.' id
  | postfix_exp '->' id
  | postfix_exp '++'
  | postfix_exp '--'
;
primary_exp =
  id
  | const
  | string
  | '(' exp ')'
;
	後置算式 =
	  基本算式
	  陣列索引 x[3]
	  函數呼叫 f(x)
	  函數呼叫 f()
	  結構欄位 rec.x
	  結構欄位 rec->x (指標版)
	  x++
	  x--
	
	基本算式
	  變數
	  常數
	  字串
	  (運算式)

圖 7.10 C 語言基本單元的語法

指定結構

C 語言指定結構的語法如圖 7.11 的 assign_exp 所示，像是 a=3*x 就是 C 語言中的一個運算式，但必須注意的是，根據 (var-ref assign-op) cond-exp 這個語法， a = b = 3*x 也是一個合法的運算式。

C 語言的 EBNF 語法 (指定結構)	說明
assign_exp = (var_ref assign_op)* cond_exp	指定運算

圖 7.11 C 語言指定結構的語法

assign_op 並非只有等號，還可以加上某些前置運算符號，像是 +=, -=, *=, /=, %=, <<=, >>=, &=, ^=, |= 特殊型的指定運算，這二元運算 (+, -, *, /, %, <<, >>, &, ^, |) 中的變數將會被用來與 assign_exp 中的結果進行運算，然後再將結果存回該變數中，舉例而言，像是 a += b 就代表了 a=a+b 的語意，這種表示法可以視為一種縮寫。

運算結構

C 語言的運算結構可分為邏輯運算、位元運算、關係運算、數學運算、單元運算等等，其語法如圖 7.12 所示。

C 語言的 EBNF 語法 (運算結構)	說明
cond_exp = logic_or_exp ('?' exp : logic_or_exp)*	條件運算
logic_or_exp = logic_and_exp (logic_or_op logic_and_exp)*	邏輯運算
logic_and_exp = bit_or_exp (logic_and_op bit_or_exp)*	
bit_or_exp = bit_xor_exp (bit_or_op bit_xor_exp)*	位元運算
bit_xor_exp = bit_and_exp (bit_xor_op bit_and_exp)*	
bit_and_exp = equal_exp (bit_and_op equal_exp)*	
equal_exp = relational_exp (equal_op relational_exp)*	
relational_exp = shift_exp (relational_op shift_exp)*	關係運算
shift_exp = add_exp (shift_op add_exp)*	
add_exp = mult_exp (add_op mult_exp)*	數學運算
mult_exp = cast_exp (mult_op cast_exp)*	
cast_exp = ((type_name))* unary_exp	轉型運算
unary_exp = unary_op cast_exp	單元運算
| (prefix_op)* postfix_exp |'sizeof' '(' type ')'	
postfix_exp = primary_exp postfix_phrase	後置運算

圖 7.12 C 語言運算結構的語法

數學運算結構從加減運算 additive_exp 開始，衍生出乘除運算的 mult_exp，舉例而言，在 a * 3 + b[5] 這個語句中，比對的情況會如 additive_exp (a*3+b) : additive_exp (a*3) + mult_exp (b[5]) 算式所示，然後再經由 additive_exp = mult_exp 這個式子，透過 mult_exp = mult_exp * cast_exp，再進一步分化，最後會透過後置運算 postfix_exp 銜接上基本單元，因而導出像 a, 3, b[5] 這樣的基本元素。

循序結構

C 語言中的指定敘述，透過分號 ";" 串聯起來，形成循序結構，像是 i=1; x=f(3); t=a; a=b; b=t; 這樣連續的指定敘述，形成更大的單元，這些單元會一個接著一個執行，以下的 seq_exp 就是 C 語言循序結構的主要語法。

C 語言的 EBNF 語法 (循序結構)	說明
exp = seq_exp seq_exp = assign_exp (seq_op assign_exp)*	循序結構 a = b; b = t;

圖 7.13 C 語言循序結構的語法

分支結構

C 語言包含 if 與 switch 等兩種分支指令，if 指令較為簡單，像是 if (i>0) x=i; 這樣的指令就是一個簡單的範例，if 指令還可以跟著 else，形成像 if(a>b) x=a; else x=b; 這樣的結構。而 switch 指令則用在多重分支結構上，像是 switch (c) { case 'a': x+=a; case 'b': x+=b; default: x+=c; } 這樣的範例，就是一個多重分支的例子。

C 語言的 EBNF 語法 (分支結構)	說明
sel_stat = 'if' '(' exp ')' stat ('else' 'if' '(' exp ')' stat)* ('else' stat)? | switch (exp) stat	分支結構 if (ab) x=a; else x=b; switch (c) {…}

圖 7.14 C 語言分支結構的語法

迴圈結構

C 語言的迴圈結構包含 while, do while 與 for 迴圈等三種，所有迴圈結構都是透過某種判斷式 exp 決定是否要跳離迴圈，而其執行的內容則都是某種陳述式 stat。

C 語言的 EBNF 語法 (迴圈結構)	說明

iter_stat
= 'while' '(' exp ')' stat
  | 'do' stat 'while' '(' exp ')' ';'
  | 'for' '(' exp ';' exp ';' exp ')' stat
…
	迴圈結構
	while 迴圈
	do while 迴圈
	for 迴圈

圖 7.15 C 語言迴圈結構的語法

函數結構

由於 C 語言是一種強型態 (Strong Typed) 的語言，所有的變數都必須宣告形態，而且又包含指標、陣列、函數指標等較為複雜的形態，因此其函數結構的語法相對複雜，以下是 C 語言函數相關的 EBNF 語法。

C 語言的 EBNF 語法 (函數結構)	說明
function_def =	函數本體
decl_specs declarator	static int f(n)
decl_list	int n;
compound_stat	{ … }
…	
declarator =	函數宣告
pointer d_declarator	static int f(n)
…	
d_declarator =	函數宣告 (無指標)
id	x
| '(' declarator ')'	(int (f*)(int))
| d_declarator '[' const_exp ']'	x[10]
| d_declarator '[' ']'	x[]
| d_declarator '(' param_types ')'	f(int x)
| d_declarator '(' id_list ')'	f(x, y)
| d_declarator '(' ')'	f()
…	…

圖 7.16 C 語言函數結構的語法

圖 7.16 中的 function_def 代表函數的定義，像是 static int f(n) int n; { return n * n; } 就是一個函數。其中的 static 是 decl 的部分，int 是 specs 的部分，而 f(n) 是 declarator，int n; 則是 decl_list，compound_stat 則比對到 { return n*n; } 區塊。

declarator 代表函數的宣告部分，由於函數中的參又有可能是一個函數指標，因此 d_declarator 又可能會導回 (declarator)，形成某種遞迴結構。

以上的 BNF 語法僅是 C 語言語法的一部分，並非全部的 EBNF 語法。在 C 語言當中還有關於資料結構的語法，像是 struct, union, enum 等，在此我們將不詳述，有興趣者請參考網路上的 C 語言語法之資訊 [4] [5]。

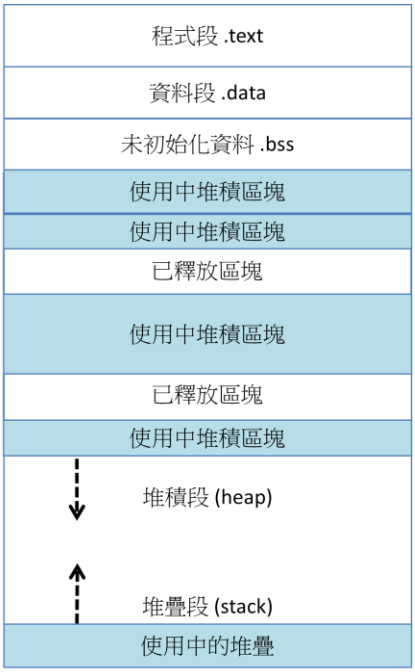
[4] C Syntax in BNF - www.cs.man.ac.uk，筆者存取時間為 3/22/2010，網址

C 語言的執行環境

C 語言通常採用編譯的方式，先將程式編譯為機器碼 (目的檔或執行檔)，然後才在目標平台上執行 C 語言。C 語言編譯後的機器碼通常是與平台相關的，是可以直接被 CPU 執行的二進位碼，因此速度非常的快，這也是 C 語言的優點之一。

C語言在執行時，通常會編譯為目的檔或執行檔的形式，這個些檔案包含程式段、資料段、BSS 段等區域，但在執行時還會多出堆疊 (Stack) 與堆積 (Heap) 等兩個區段。

(a) 程式開始時的記憶體分配情況



(b) 程式執行中的記憶體分配情況

圖 7.17 C 語言的執行時的記憶體配置圖

在程式執行的過程中，經常會需要取得某些記憶空間，以便儲存電腦運算過程中所產生的資料，程式中的資料通常被放在兩個記憶體區塊中，一個稱為堆疊區 (Stack)，一個稱為堆積區 (Heap)。

堆疊段與堆積段的成長方向是相反的，假如堆積由上往下成長，堆疊段的成長方

---

http://shell.sh.cvut.cz/~wilx/repos/c2pas/_darcs/current/docs/c_syntax.bnf。

[5] ISO/IEC 9899:1999, C99 Specification，筆者存取時間為 3/22/2010，網址 http://www.open-std.org/JTC1/SC22/wg14/www/docs/n1124.pdf

向就會是由下往上。堆疊與堆積兩段共用同一塊記憶體空間，但是起始點與成長方向完全相反。

編譯器會將副程式的參數、區域變數與返回點等資訊會被推入堆疊中，並且會從堆疊中分配空間給區域變數使用。堆疊的記憶體的配置並不困難，當需要記憶體時，一定是從堆疊的最上層開始分配，編譯器只要根據變數的型態與數量，決定配置空間的大小即可。

堆積區的記憶體使用方法就較為複雜了，在 C 語言當中，malloc() 函數是主要的記憶體請求指令，這種指令通常被稱為動態記憶體 (Dynamic Allocation) 配置請求，因為 malloc() 函數會在執行的過程當中，動態的取得足夠的記憶體空間，以便分配給程式使用。

記憶體配置函數 malloc() 會從堆積段當中分配一塊記憶體後傳回其指標，於是呼叫端的程式就可以利用這個指標進行資料存取。但是，由於 malloc() 會導致堆積區的成長，而函數呼叫則會導致堆疊段的成長，如果兩個區域成長過頭而導致重疊的情況，就會相互覆蓋而導致資料破壞的情況。這對程式設計人員而言是一種很難處理的錯誤，最好能設計其錯誤處理機制以防止此種情況。

因此、C 語言的程式設計師必須在使用完 malloc() 所分配的記憶體後，盡快的利用 free() 函數以歸還記憶體給堆積區，這樣才能避免堆疊溢出 (或堆積溢出) 的情況，讓程式能在堆積尚未溢出之前完成。但是如果所有堆積空間不足，而且沒有任何的『未分配記憶體區塊』可以滿足記憶體分配的請求時，程式仍然會被迫停止，或者進入不可預知的錯誤狀況。

使用框架存取參數與區域變數

在第 3 章中，我們曾經介紹過兩種組合語言呼叫副程式 (函數) 的方法，但是這兩種方法並不適合被編譯器採用，原因是編譯器的函數參數可能很多，不一定能完全以暫存器來容納。

另外，當 C 語言的函數想要存取參數或區域變數時，通常不能透過變數名稱存取這些變數，否則就不能支援遞迴呼叫了。因為在遞迴呼叫的過程中，參數名稱與區域變數的名稱雖然相同，但是不同層次的遞迴所『看見的』變數內容是不同的。

換句話說，當我們將 C 語言程式轉換為組合語言時，不能將參數與區域變數轉換為組合語言中的標記，而必須轉換為堆疊區域的存取指令。

一個函數的參數與區域變數所形成的堆疊區塊，通常稱之為框架 (Frame)，為了要存取這個框架，我們可以設定一個框架暫存器 (Frame Pointer, FP)，然後使用相對定址的方式存取這些變數。

在 CPU0 當中，我們可以利用 R1~R11 當中的任何一個暫存器，作為框架暫存器，在本書中，我們會習慣以 R11 作為框架暫存器，因此我們也用 FP 稱呼 R11。

為了說明框架的用法，我們將使用範例 7.1 的 C 語言程式進行說明，該範例中有兩層的函數呼叫，主程式 main() 會利用 f1(x) 指令呼叫函數 f1，然後在 f1() 中又利用 f2(&t)指令呼叫了 f2，其中 f1(x) 傳遞的是數值參數，而 f2(&t) 傳遞的則是位址。

範例 7.1 具有兩層函數呼叫的 C 語言程式

```
1   int main() {
2       int x = 1;
3       int y;
4       y = f1(x);
5       return 1;
6   }
7   int f1(int t) {
8       int b = f2(&t);
9       return b+b;
10  }
11  int f2(int *p) {
12      int r= *p+5;
13      return r;
14  }
```

在進行函數呼叫時，母函數必須先將參數推入堆疊當中，然後在進入函數後，再將母函數的『框架暫存器』堆入堆疊保存，接著分配區域變數的空間，然後才能進行函數的真正功能。圖 7.18 顯示了上述範例程式的堆疊變化情況，以及 FP、SP 等指標的位置，其中的 FP 是框架暫存器。

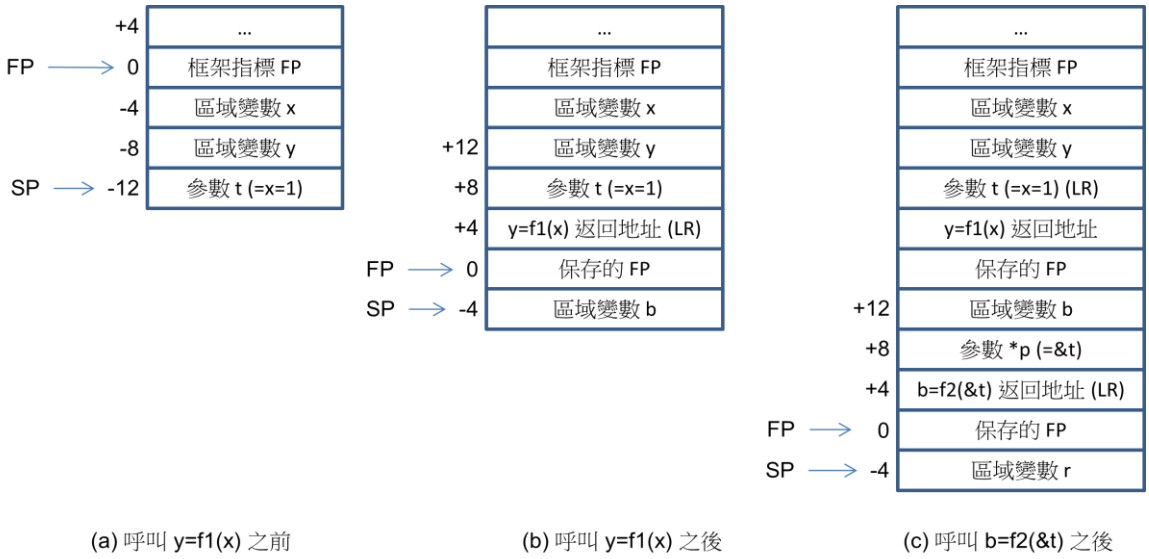


圖 7.18 函數呼叫時的堆疊與框架變化情形

根據圖 7.18 的說明，我們可以將範例 7.1 編譯為 CPU0 的組合語言，其內容如範例 7.2 所示。

範例 7.2 <範例 7.1> 程式對應的組合語言

	組合語言	說明	C 語言 (真實版)
1	_main:	void main()	void main() {
2	//****前置段********		
3	PUSH LR	將 LR 推入堆疊	
4	PUSH FP	將 FP 推入堆疊	
5	MOV FP, SP	設定新的 FP	
6	SUB SP, SP, 8	分配參數空間	
7	//****主體段********		
8	CALL _init	呼叫 _init 進行初始化	
9	MOV R3, 1	R3=1	int x = 1;
10	ST R3, [FP-4]	x = [FP-4] // = R3 = 1	int y;
11	PUSH R3	將 x 推入堆疊	
12	CALL _f1	呼叫函數 f1();	y = f1(x);
13	ADD SP, SP, 4	恢復原先堆疊指標	
14	MOV R3, R1	R3=R1 //=回傳值 f1(x)	
15	ST R3, [FP-8]	y=[FP-8] = R3	
16	//****結束段********		
17	MOV SP, FP	恢復 SP	
18	POP FP	恢復 FP	
19	RET	PC=LR, 回到呼叫點	}

20	f1:		int f1(int t) {
21	//****前置段********		
22	PUSH LR	將 LR 推入堆疊	
23	PUSH FP	將 FP 推入堆疊	
24	MOV FP, SP	設定新的 FP	
25	SUB SP, SP, 4	分配區域變數空間 b	
26	//****主體段********		
27	ADD R3, FP, 8	R3 = FP+8 = &t	
28	PUSH R3	PUSH R3 // (&t)	
29	CALL f2	呼叫函數 f2()	int b = f2(&t);
30	ADD SP, SP, 4	恢復原先堆疊指標	
31	ST R1, [FP-4]	b = R1	
32	LD R3, [FP-4]	R3= [FP-4] // = b	
33	LD R2, [FP-4]	R2 = [FP-4] // = b	
34	ADD R3, R3, R2	R3 = R3+R2 = b + b	return b+b;
35	MOV R1, R3	傳回值 R1 = R3	
36	//****結束段********		
37	MOV SP, FP	恢復堆疊	
38	POP FP	恢復 FP	
39	POP LR	恢復 LR	
40	RET	返回	}
41	f2:		int f2(int *p) {
42	//****前置段********		
43	PUSH LR	將 LR 推入堆疊	
44	PUSH FP	將 FP 推入堆疊	
45	MOV FP, SP	設定新的 FP	
46	SUB SP, SP, 4	分配區域變數空間 r	int r=*p+5;
47	//****主體段********		
48	LD R3, [FP+8]	R3=[FP+8] // =*p 的位址	
49	LD R2, [R3]	R2 = [R3] = *p	
50	ADD R3, R2, 5	R3 = R2+5 = *P+5	
51	ST R3, [FP-4]	r = [FP-4] = R3	
52	MOV R1, R3	傳回值 R1 = R3	return r;
53	//****結束段********	******結束段********	
54	POP FP	恢復 FP	
55	POP LR	PC=LR, 回到呼叫點	}
56	RET	}	

由於範例 7.2 的組合語言相當複雜，在此，我們有必要搭配圖 7.18 進一步說明，以下請讀者同時參考兩者以方便理解。

使用框架暫存器 FP 之目的，是要對『區域變數』與『參數』進行定址工作。如此，就不需要依靠暫存器傳遞參數，而是直接以 FP 作為定址的基準，利用相對於 FP 的位移定址，存取這些『區域變數』與『參數』。

採用此種作法，在函數呼叫之前，組合語言程式會先將參數推入到堆疊當中。舉例而言，在範例 7.2 中的第 11 行與 28 行的 PUSH 指令，都是在進行參數推入的工作，其中，第 11 行執行完後的情況如圖 7.18a 所示。

然後，在進入函數後，會先執行一些『前置段』程式，像是範例 7.2 中的所有『前置段』程式，都執行了如下的程式碼。

前置段程式	說明
PUSH LR	將 LR 推入堆疊
PUSH FP	將 FP 推入堆疊
MOV FP, SP	設定新的 FP
SUB SP, SP, <N>	分配大小為 <N> 的參數空間

上述程式會先保存連結暫存器 LR 的值，以避免該函數再度呼叫子函數時，LR 的值會被覆蓋。接著再保存舊的框架暫存器 FP，以便函數返回前可以恢復 FP。接著，將框架暫存器更新為堆疊的頂端 (MOV FP, SP)。最後，再分配好區域變數的空間之後，前置段的工作就完成了。

接著，就可以進入函數的主體段，執行函數真正需要做的動作。當程式需要存取『區域變數』或『參數』時，就可以採用相對於 FP 之定址方式，也就是以 [FP + 位移] 的形式，進行變數的存取。舉例而言，範例 7.2 的第 31-33 行，即是以 [FP-4] 的方式存取區域變數 b，讀者可以參照圖 7.18 (b)，就能很容易得知變數 b 相對於 FP 的位移為 -4。

同樣的，在 f2 函數中存取區域變數 r 時，其相對於 FP 的位移也是 -4 (請參照圖 7.18c)，因此，在 51 行當中，使用 ST R3, [FP-4] 將 R3 中的 *p+5 存回 r 當中，完成 r=*p+5 的動作。

在圖 7.18 當中，區域變數的位移為負值，而參數的位移則為正值，這是因為參數是在 FP 推入前就已經被推入堆疊的，像是 f2 中的參數 *p，其位移是 +8。因此，在 48 行中，就利用 LD R3, [FP+8] 指令將 *p 所對應的參數 &t 載入到暫

存器 R3 當中。

同樣的，在 f1 當中的參數 t，其位移也是 +8，因此，在 27-28 行當中，就使用 ADD R3, FP, 8，PUSH R3 等兩個指令，將參數 t 的位址 &t 推入到堆疊中，以便在 f2 函數中能取得該參數。

另外，在範例 7.2 當中，我們固定使用暫存器 R1 儲存函數的傳回值。由於函數的傳回值只有一個，因此通常不會有暫存器不足的問題。這是在編譯器設計時很常見的一種作法，這樣可以避免掉傳回值的推入與取出動作，增加程式的效率。

請讀者仔細追蹤範例 7.2 與圖 7.18，就能理解框架的運作原理，以及編譯器如何利用框架暫存器存取參數與區域變數的方法。

至此，我們已經說明了 C 語言的語法、語意以及執行環境等主題，透過 C 語言作為範例，我們可以進一步的認識真實的程式語言，是如何被設計與實作出來的，在下一章當中，我們會進一步介紹編譯器這個主題，以便更深入的理解高階語言如何被轉換為組合語言。

習題

7.1 請說明何謂 BNF 語法？何謂 EBNF 語法？並比較兩者的異同。
7.2 請將 BNF 語法 A = B | A '.' B 轉換為 EBNF 語法。
7.3 請寫出 C 語言當中 for 迴圈的 BNF 語法。
7.4 請說明何謂直譯器？
7.5 請說明何謂編譯器？
7.6 請比較直譯器與編譯器兩者的異同。
7.7 請說明何謂語法理論？
7.8 請說明何謂語意理論？
7.9 請說明何謂框架 ？
7.10 請舉例說明 C 語言如何利用框架暫存器存取參數與區域變數？