第8章　編譯器

編譯器是高階語言的一種實作方式，在本章當中，我們將承襲上一章的高階語言理論，繼續說明編譯器的設計原理。

總體來說，編譯器是將高階語言轉換為組合語言的工具，如果我們將編譯的步驟詳細分解後，大致可以分為詞彙掃描 (8.2 節)、語法剖析(8.3 節)、語意分析(8.4 節)、中間碼產生(8.5 節)、最佳化(8.7 節)、組合語言產生 (8.6 節) 等六大階段，在本章中，我們將採用範例導向的方式，說明這些階段的功能。

我們首先在 8.1 節介紹編譯器的整體架構，然後，從 8.2 節的掃描器開始，說明詞彙分析的實作方式，接著，在 8.3 節中說明剖析器的設計方法，由於剖析器的方法既多且複雜，我們將專注在最常見的遞迴下降式剖析器上，以實例說明其建構方式，接著，在 8.4 節當中，利用語意分析對語法樹加上型態標記，然後在 8.5 節中介紹中間碼，接著在 8.6 節中討論如何將中間碼轉換為組合語言、最後在 8.7 節中討論最佳化的主題。

8.1 簡介

編譯器是用來將高階語言轉換成組合語言 (或者是機器碼) 的工具程式。有了編譯器或直譯器，程式設計師才能用高階語言撰寫程式。因此，編譯器是程式設計師的重要工具，也是系統程式課程的重點之一。

圖 8.1 顯示了一個編譯器的基本功能，在該圖中，像 sum=sum+i 這樣的高階語言指令，輸入到編譯器之後，會被轉換成一連串的組合語言指令，然後，這些組合語言指令再度被組譯器轉換成機器碼，成為執行檔以便在目標機器上執行。

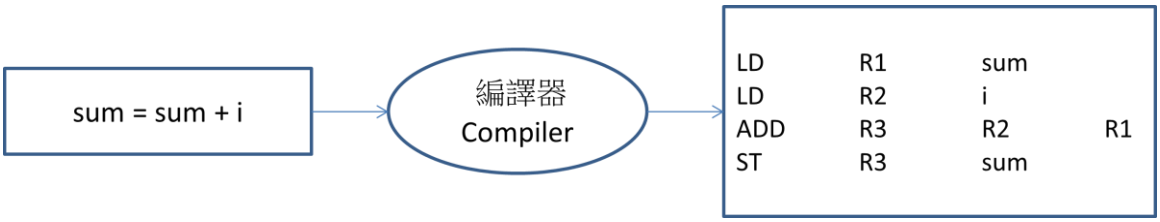

圖 8.1 編譯器的輸入與輸出

編譯器的理論核心是前一章所述的語法理論，為了說明整個編譯器的流程，我們

設計了一個非常簡單的高階語言，由於該語言非常類似 C 語言，因此被稱為 C0，代表 C 語言第 0 版的意思，C0 語言的語法與 C 語言非常相似，但是為了容易理解，C0 語言只包含 for 迴圈與基本的運算式而已，並不包含像 if、函數呼叫、甚至是結構 struct 等功能。

讓我們來看看一個完整 C0 語言範例，如範例 8.1 所示，該範例的用途乃是計算 1 + 2 + … + 10 的結果，其語法與 C 語言幾相當類似，但是卻不需要宣告變數的型態，因此，你在該程式當中看不到像 int sum; 這樣的型態宣告指令，因為，目前在 C0 語言當中只有一種型態，那就是整數。(當然，這樣的語言並沒有太大用途，其目的只是用來說明編譯器的設計原理而已)。

範例 8.1 一個 C0 語言的程式範例

C0 語言程式 (位於 sum.c0 範例檔中)

```
sum = 0;
for (i=1; i<=10; i++)
{
   sum = sum + i;
}
return sum;
```

為了製作 C0 語言的編譯器，我們寫出了 C0 語言的 EBNF 語法規則，如圖 8.2 所示，C0 語言總共包含 11 條 EBNF 規則，可以用來撰寫一些小型的程式。其中的第 10, 11 條的 id 與 number 是詞彙的組成規則，而第 1-9 條則是剖析時使用的語法規則。

	EBNF 語法規則	
1	PROG	= BaseList
2a	BaseList	= (BASE)*
3	BASE	= FOR | STMT ';'
4	FOR	= 'for' '(' STMT ';' COND ';' STMT ')' BLOCK
5	STMT	= 'return' id | id '=' EXP | id ('++'|'--')
6	BLOCK	= '{' BaseList '}'
7a	EXP	= ITEM ([+-*/] ITEM)?
8	COND	= EXP ('=='|'!='|'<='|'>='|'<'|'>') EXP
9	ITEM	= id | number
10	id	= [A-Za-z_][A-Za-z0-9_]*
11	number	= [0-9]+

圖 8.2 C0 語言的 EBNF 語法規則

在上述的規則中，星號 * 代表重複零次以上，加號代表重複一次以上，因此，第 7 條的 BaseList = (BASE)* 代表 BaseList 可以由許多個 BASE 所組成(包含零次)，而第 11 條的 number = [0-9]+ 代表 number 由 0,1,2,3,4,5,6,7,8,9 等數字重複一次以上所組成的。

如果規則中的一個 (…) 區塊或 […] 區塊後面跟著問號 ?，那麼就代表該區塊可以出現零次或一次，例如在第 7 條的 EXP = ITEM ([+-*/] ITEM)? 當中的問號，代表 ([+-*/] ITEM) 這個區塊可以出現零次或一次，也就是 EXP = ITEM 或 EXP = ITEM [+-*/] ITEM 都是符合語法的語句。

假如規則中的 (…) 或 […] 區塊後沒有跟著任何符號，那麼就代表該區塊只能出現一次，舉例而言，在第 8 條的 COND = EXP ('=='|'!='|'<='|'>='|'<'|'>') EXP 當中，代表 ('=='|'!='|'<='|'>='|'<'|'>') 這個區塊只能出現一次，而其中的直線符號 | 代表或者的意思，該語句的意義是 '=='、'!='、'<='、'>='、'<'、'>' 這些符號其中的一個會出現一次。

第 4 條的 FOR 規則是整個語法中最複雜的一條，其中包含三個重要的部分，也就是 STMT、COND 與 BLOCK 等三者，STMT 用來描述 i=0; i++ 等敘述，而 COND 則描述條件判斷部分，像是 i<=10 等，而最後的 BLOCK 則是 for 迴圈的主體部分，BLOCK 乃是由一對大括號 {..} 夾住的 BaseList 區段所組成，BLOCK 透過 BaseList 會回到 BASE，於是又遞迴的定義了下一層的程式區段。

因此， 圖 8.2 的規則實際上已經定義了多層的 for 結構語法，所以可以透過下列的生成方式可以產生多層次的 FOR 語句。

PROG→BASE→FOR→BLOCK→BaseList→BASE→FOR→BLOCK→BaseList→BASE→STMT …

編譯器的六大階段

編譯的步驟可以細分為六大階段，分別是詞彙掃描、語法剖析、語意分析、中間碼產生，最佳化、組合語言產生等六大階段，圖 8.3 顯示了這六大階段的輸入與輸出，這個圖非常的重要，請讀者務必仔細觀察其輸入與輸出，以便理解每一個階段的功能。

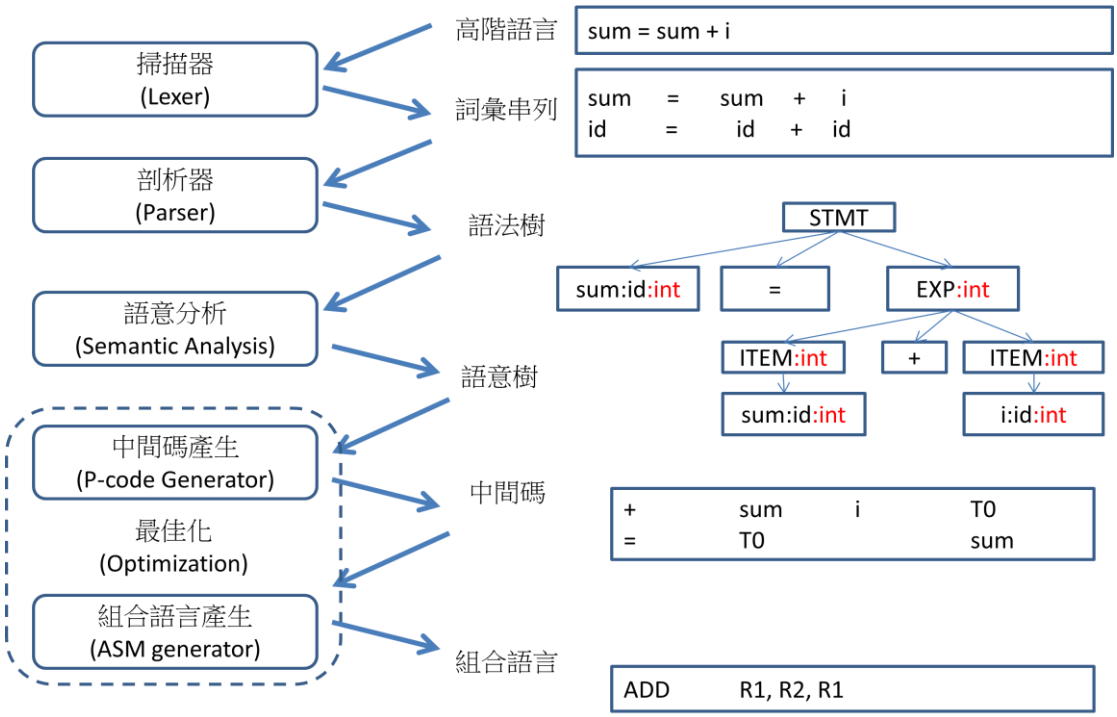

圖 8.3 編譯器的六大階段

在第一階段當中，高階語言的程式碼，像是圖 8.3 中的 sum = sum + i 這個語句，會被輸入到掃描器 (Lexer 或 Scanner) 當中，掃描器會將整個程式分成一個一個的基本詞彙 (token)，並為每個詞彙標註型態，於是會輸出 (sum:id) (=) (sum:id) (+) (i:id) 這樣的詞彙串列。

接著，這個詞彙串列成為剖析器 (Parser) 的輸入，剖析器利用語法規則進行比對，以逐步建立語法樹，直到整個程式成為一棵完整的大樹為止，於是 sum = sum + i 這個語句將會形成圖 8.3 中的語法樹 (Abstract Syntax Tree)。

在語意分析 (Semantic Analysis) 階段，編譯器會為這個語法樹加註節點型態，並檢查這些型態是否相容，然後輸出語意樹，在圖 8.3 的語法樹中，sum、i、ITEM、EXP 等節點，就是在語意分析階段被加註了 int 型態，於是形成了具有語意標記的語法樹 (Annotated Abstract Syntax Tree)，或稱為語意樹 (Semantic Tree)。

接著，在中間碼產生階段，語意樹被轉換成一種平坦的結構，這種結構很像組合語言，但卻又不是特定機器的組合語言，這種結構被稱為中間碼 p-code (Pseudo Code)。

中間碼是一種『與機器結構無關的組合語言』，像是 Java 的 bytecode，就是一種廣為人知的中間碼。在中間碼的指令中，通常沒有暫存器的概念，因此所有運算可以直接對變數進行，而不需要考慮暫存器配置的問題。

接著，我們就可以將中間碼轉換為組合語言，在轉換時必須考慮暫存器的配置問題，以及程式的效率問題，因此，編譯器通常會進行某些最佳化的動作之後，才將中間碼轉換為組合語言輸出。

舉例而言，圖 8.3 的 + sum i T0 與 = T0 sum; 等兩行中間碼，被轉換為組合語言時竟然只剩下一行 ADD R1, R2, R1 的指令，這是由於前面的程式已經將 sum, i 等變數載入到暫存器 R1, R2 當中，因此最佳化後的程式才能直接對 R1, R2 進行加法動作，否則的話，這兩行中間碼應該會轉換成將近十行的組合語言才對。

現在，我們已經說明完編譯器的六大階段了，接著，讓我們以 C0 語言為範例，更詳細的說明每一個階段的功能。

8.2 詞彙掃描

詞彙掃描步驟的功能，是將程式切分成一個一個的詞彙，以便作為剖析器的輸入。

高階語言的程式可以視為是一個字串，其基本單位是字元。然而，剖析器通常不接受以字元為單位的輸入，而是期待能以詞彙為單位，而這個將字元轉換成詞彙的動作，正是掃描器的功能。

掃描階段的輸出乃是一個詞彙串列，而且會在每個詞彙上標註特殊的詞類標記，圖 8.4 顯示了這個詞彙串列以及詞類標記，其中，包含了識別字[1](id) 以及一些基本運算，像是等於 (=), 加號 (+), 乘號 (*) 等等。像是 sum 與 i 等變數就被標註上 id 這個詞類標記，以便後續的剖析階段處理，而對於基本運算而言，在本章的範例當中，將直接以該運算符號作為標記。

詞彙 (Token)	sum	=	sum	+	i
詞類標記 (Type)	id	=	id	+	id

圖 8.4 掃描階段的輸出 – 具類型標記的詞彙串列

在高階語言當中常見的詞類標記有識別字(id), 常數 (number), 字串 (string)等等，在圖 8.11 當中我們已經看到識別字 id 的範例，為了更清楚的說明這些標記的意義，讓我們再舉一個範例說明。

[1] 所謂的識別字就是像變數名稱、函數名稱、標記名稱等，利用文字型的名稱代表某個程式中的物體，即是本文中所說的識別字。

若 C 語言當中的 printf("%d", 30) 指令被掃描器掃入後，會輸出如圖 8.12 的詞彙串列，其中的 printf 是識別字 (id)，"%d" 則是字串 (string)，而 30 則被標上數字標記 (number)。

詞彙 (Token)	printf	(	"%d"	,	30	)
詞類標記 (Type)	id	(	string	,	number	)

圖 8.5 掃描階段的輸出 – 包含常數與字串的範例

簡單來說，掃描器是用來將程式轉換成一串詞彙序列的程式，但是在實作上，通常會將掃描器撰寫成像 nextToken() 這樣的單一個函數，以供剖析器呼叫使用，每當剖析器希望取得下一個詞彙 (token) 時，就可以呼叫該函數。

我們可以使用逐字比對的方式，利用判斷與迴圈等方式製作掃描器，舉例而言，我們可以將圖 8.2 當中的與 number = [0-9]+ 等詞彙規則，轉換成圖 8.6 掃描程式中的第 3~8 行，並將 id = [A-Za-z_][A-Za-z0-9_]* 這個規則，轉換成第 9~13 行。圖 8.6 掃描程式的其他部分，則用來取得像 <=, >=, ++, --, +, -, *, / 等運算詞彙 (第 19~24 行)，或者判斷某詞彙是否為關鍵詞 (Keyword) (第 14~18 行)，然後將其他的字元，像是 (,), {, } 等，都視為單一字元的詞彙，直接傳回 (第 25~27 行)。

	C0 語言的掃描器演算法	說明
1	function nextToken(file, c)[2]	file 為原始程式碼檔案, c 為下一個字元
2	token = new string()	建立 token 字串
3	if (c in [0-9])	如果是數字 (number)
4	while (c in [0-9])	不斷取得數字
5	token.append(c)	放入 token 字串中
6	c = file.nextchar()	再取得下一個字元
7	end while	
8	tag = "number"	設定詞類標記(tag)為數字
9	else if (c in [a-zA-Z_])	如果是英文字母 (id)
10	while (c in [a-zA-Z0-9_])	不斷取得英文、數字或底線
11	token.append(c)	放入 token 字串中
12	c = file.nextchar()	再取得下一個字元
13	end while	

[2] 函數 nextToken(file, c) 當中的 c 參數，代表上一次所取得的字元，這是因為掃描器往往會在掃過頭之後才會知道不應該再讀取了。舉例而言，當我們從 32+x 這個字串想要掃描一個整數時，一定會掃到 + 號後，才知道原來整數已經結束了，因此需要用 c 參數以儲存上次多掃到的那個字元。

14	if token is keyword	如果是關鍵字(C0 的關鍵字只有 for)
15	tag = token	設定詞類標記為該關鍵字
16	else	否則
17	tag = id	設定詞類標記為 id
18	end if	
19	else if (c in [+-*<=>!])	如果是運算符號
20	while (c in [+-*<=>!])	不斷取得運算符號
21	token.append(c)	放入 token 字串中
22	c = file.nextchar()	再取得下一個字元
23	end while	
24	tag = token	設定詞類標記為該詞彙
25	else	否則就是單一字元，像是 { 或 }
26	token = c;	設定 token 為該字元
27	tag = token;	設定 tag 為該字元
28	end if	
29	return (token, tag)	傳回取得的詞彙
30	end	

圖 8.6 C0 語言的掃描器演算法

接著，我們可以利用不斷呼叫 nextToken() 函數的方法，不斷的取得詞彙，直到檔案結束為止，這樣，就能將程式分解成一連串的詞彙。

C0 語言的掃描器演算法	說明
Algorithm tokenize(file)	file 為原始程式碼檔案
c = file.nextchar()	取得第一個字元
while not file.isEnd()	在檔案尚未結束時
token = nextToken(file, c)	取得下一個詞彙
print(token)	輸出該詞彙
end while	
End Algorithm	

圖 8.7 將 C0 語言的程式分解成詞彙的演算法

掃描器只是編譯器的一個小元件而已，真正重要的元件是剖析器，剖析器會利用掃描器所取得的詞彙，將整個程式轉換為一棵語法樹，在下一節當中，我們將說明如何利用本節的掃描器，製作出剖析器的方法。

8.3 語法剖析

剖析器的設計方法有很多種，大致可分為由上而下的方法 (像是遞迴下降法、LL 法)，與由下而上的方法 (像是運算子優先矩陣法、LR 法) 等，在本節中，我們將使用遞迴下降法作為主要的剖析法，其餘方法請參考編譯器的專門書籍。

遞迴下降法

遞迴下降法是一種由上而下的剖析法，該方法的實作比 LL、LR 等剖析法更為簡單而直接，這是我們採用遞迴下降法進行說明的原因。

剖析器的撰寫者，只要能夠將 EBNF 語法轉換為遞迴下降函數，就能製作出遞迴下降剖析器，舉例而言，在圖 8.2 的 C0 語言與法中，規則 7a 的 EXP = ITEM ([+-*/] ITEM)? 可以被翻譯成如圖 8.8 的演算法。

C0 語言 EXP 規則的剖析函數	說明

function parseExp()
  pushNode("EXP")
  parseItem();
  if isNext("+|-|*|/")
  next("+|-|*|/")
  parseItem()
  end if
  popNode("EXP")
end
	剖析 EXP 語法
	  建立 EXP 節點，推入堆疊中
	  剖析 ITEM 語法
	  如果下一個是加減乘除符號
	  取得該符號
	  剖析下一個 ITEM 語法
	
	  取出並傳回 EXP 這棵語法樹

圖 8.8 將規則 EXP= TERM ([+-] TERM)* 翻譯成遞迴下降剖析程式

從圖 8.8 的演算法當中，我們可以很清楚的看到，要將 EBNF 語法翻譯為遞迴下降剖析程式並不困難，其過程相當的機械性，只要在函數的開始以 pushNode(…) 建立語法樹的節點，然後在結尾以 popNode(…) 移除該節點，並且在比對的過程當中，利用 parseXXXX() 等函數，繼續比對下層節點，並利用 next() 函數比對詞彙即可。

舉例而言，在圖 8.8 當中，我們就根據 EXP = ITEM ([+-*/] ITEM)? 這條規則，先利用 pushNode("EXP") 建立 EXP 節點，然後根據語法規則 ITEM ([+-*/] ITEM)?，將 ITEM 轉換為 parseItem()，並用 next("+|-|*|/") 比對加減乘除這些詞彙，最後將 EXP 節點彈出並傳回，即完成的該規則的遞迴下降之程式實作。

必須注意的是，由於規則 ([+-*/] ITEM)? 當中有問號，代表該區塊可能出現零次或一次，因此必須用 if isNext("+|-|*|/") 這個函數，事先判斷到底後面有沒有跟著加減乘除的符號，如果有就繼續取得 ([+-*/] ITEM) 區塊，否則就不應該繼續比對 ([+-*/] ITEM) 區塊了。

在實務上，甚至有人發展出可以自動將 EBNF 或 BNF 語法轉換為編譯器的程式，此種程式稱為編譯器的編譯器 (Compiler Compiler)，或者稱為剖析器產生程式 (Parser Generator)，像是 YACC 就是一個著名的剖析器產生程式，其全名是 Yat Another Compiler Compiler。Bison 則是開放原始碼組織 GNU 模仿 YACC 所撰寫的一個剖析器產生程式，這些程式可以搭配掃描器產生程式，像是 Lex 或 Flex，形成一組完整的編譯器設計工具，以降低設計編譯器的困難度。

利用這種方法，我們就可以撰寫完整的 C0 語言剖析器，其演算法如圖 8.9 所示。

C0 語言的掃描器演算法	說明
Algorithm C0Parser	剖析器演算法
Stack stack	共用變數，包含 stack : 堆疊
File file	file : 輸入的程式檔
Token token	token: 目前的詞彙
char c	c : 掃描器的目前字元
// functions for parser	函數區開始
function parse(fileName)	剖析器的主要函數 - parse()
stack = new stack()	宣告堆疊
file = new File(fileName)	取得輸入檔，建立物件
c = file.nextchar()	取得第一個字元
getNextToken()	取得第一個 (詞彙, 標記)
parseProg()	開始剖析該輸入程式檔
end	
function getNextToken()	取得下一個 (詞彙, 標記)
(token,tag) = nextToken(file,c)	取得下一個 (詞彙, 標記)
end	
function isNext(tags)	判斷下一個詞彙標記是否為 tags 之一
if (tokenNext.tag in tags)	
return true;	
else	
return false;	

end	
function next(tags)	將下一個詞彙建立為新節點，放入父節點中
if isNext(tags)	
child = new Node(token);	
parent = stack.peek()	
parent.addChild(child)	
end if	
end	
function pushNode(tag)	建立具有 tag 標記的新節點，推入堆疊中
node = new Node(tag)	
stack.push(node)	
end	
function popNode(tag)	取出節點
node = stack.pop()	取出節點
if (node.tag == tag)	看看是否具有 tag 標記
parentNode = stack.peek()	取得上一層的樹
parentNode.addChild(node)	將該節點設定為上一層的子節點
else	如果不具有 tag 標記
error("Parse error")	則是語法錯誤，進行錯誤處理
end if	
end	
function parseProg()	剖析規則 1:
pushNode("PROG")	PROG = BaseList
parseBaseList()	
popNode("PROG")[3]	
end	
function parseBaseList()	剖析規則 2a:
pushNode("BaseList")	BaseList = (BASE)*
while not file.isEnd()	
parseNext("BASE")	
popNode("BaseList")	
end	
function parseBase()	剖析規則 3: BASE = FOR | STMT ';'
pushNode("BASE")	
if isNext("for")	處理 FOR
parseFor()	

[3] 當 popNode("PROG") 執行之前，堆疊中尚有一個 PROG 節點，因為 parseBaseList() 函數只會將 BaseList 標記取出就跳回了，因此必須再取出 PROG 節點之後，堆疊才會清空。

else
  parseStmt()
  next(";");
end if
popNode("BASE")
end
	處理 STMT ';'

function parseFor() pushNode("FOR") next("for") next("(") parseStmt() next(";") parseCond() next(";") parseStmt() next(")") parseBlock() popNode("FOR") end	剖析規則 4: FOR = 'for' '(' STMT ';' COND ';' STMT ')' BLOCK

function parseStmt()
pushNode("STMT")
if (isNext(p, "return"))
next(p, "return");
next(p, "id");
else
next(id)
if isNext("=")
next("=")
parseExp()
else
next("++|--")
end if
end if
popNode("STMT")
end
	剖析規則 5:
	STMT = 'return' id | id '=' EXP | id ('++'|'--')
	處理 'return' id
	處理 id '=' EXP
	處理 id ('++'|'--')

function parseBlock() pushNode("BLOCK") next("{")	剖析規則 6: BLOCK = '{' BaseList '}'

parseBaseList() next("}") popNode("BLOCK") end	

function parseExp()
pushNode("EXP")
parseItem();
if isNext("+|-|*|/")
next("+|-|*|/")
parseItem()
end if
popNode("EXP")
end
	剖析規則 7a: EXP = ITEM ([+-*/] ITEM)?
	建立 EXP 節點，推入堆疊中
	剖析 ITEM 語法
	如果下一個是加減乘除符號
	取得該符號
	剖析下一個 ITEM 語法
	
	取出並傳回 EXP 這棵語法樹

function parseCond() pushNode("COND") parseExp() next("==|!=|<=|>=|<|>") parseExp() popNode("COND") end	剖析規則 8: COND = EXP ('=='|'!='|'<='|'>='|'<'|'>') EXP
function parseItem() pushNode("ITEM") next("id|number"); popNode("ITEM") end	剖析規則 9: ITEM = id | number
end Algorithm	

圖 8.9 C0 語言的剖析器演算法

在圖 8.9 的演算法中，堆疊 stack 是用來建立語法樹的關鍵物件，當遞迴下降演算法剖析程式時，會將剖析的節點依照階層順序推入堆疊當中，每次比對一個規則時，就會建立一個新的節點，例如，在 parseProg() 當中，一開始就會利用 pushNode("PROG") 建立一個新的根節點，推入堆疊當中。

然後在呼叫 parseBaseList() 時，又建立了一個新節點，推入堆疊中，此時，堆疊中將有 PROG, BaseList 等兩個節點。等到 parseBaseList() 函數要離開前，再利用 popNode("BaseList") 取回已經建立好的 BaseList 節點，此時，該節點已經是一棵完整的樹，並且附加到 PROG 節點之下，再回到 parseProg() 函數時，就形成一棵更大的樹，於是，整個輸入程式就在遞迴的過程當中被建立為語法樹。

這樣說明或許尚不清楚，讓我們實際用一個範例進行分析，假如我們將範例 8.1 的 C0 語言程式輸入到圖 8.9 的演算法中，一開始剖析函數 parse() 會做好基本設定，然後就會呼叫 parseProg()，parseProg() 會呼叫 parseBaseList()，parseBaseList() 會呼叫 parseBase()，而 parseBase() 會呼叫 parseStmt()，由於 STMT 的規則為 'return' id | id '=' EXP | id ('++'|'--')，此時遞迴下降程式會發現第一個詞彙為 id 型態，因此在 parseStmt() 中會呼叫 next("id"); next("="); parseExp(); 等函數，其中 next("id") 指令會取得 sum 變數，然後 parseExp() 會繼續進行遞迴下降呼叫，企圖建立 EXP 的語法樹。

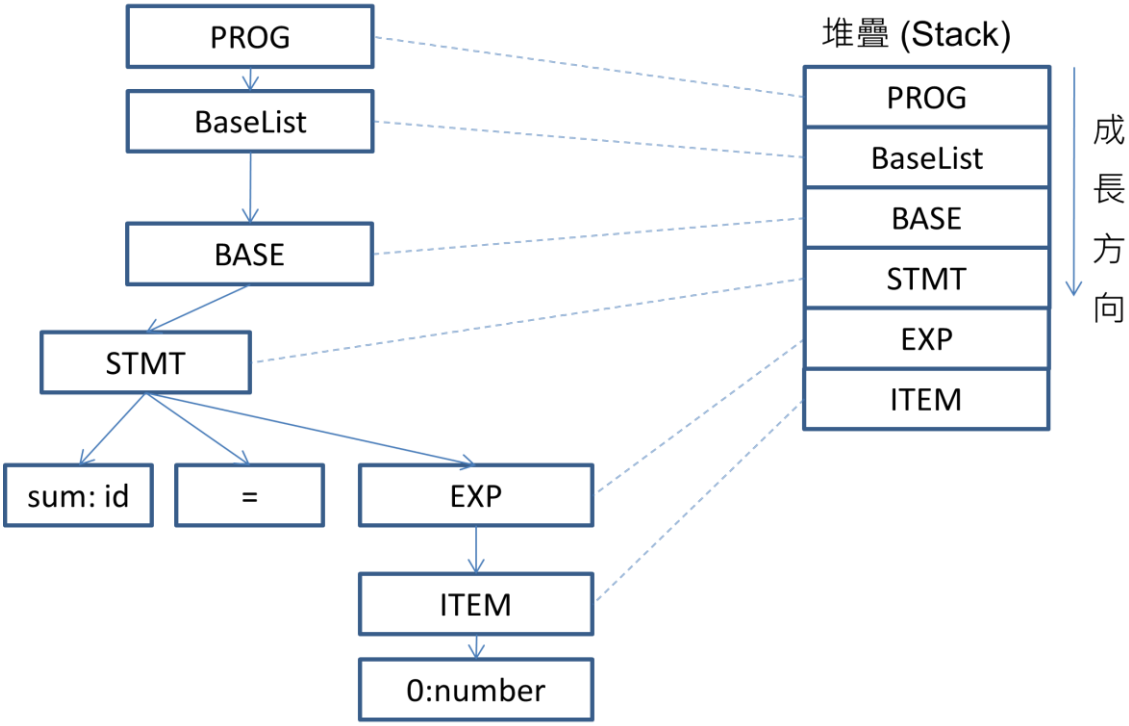

圖 8.10 <範例 8.1> 剖析到 sum=0 時的語法樹與堆疊結構

接著 parseExp() 又會呼叫 parseItem()，於是 parseItem() 用 next("id|number") 指令比對到 0 的詞彙標記為 number，於是一層一層的傳回，建構出整個 sum=0 這個指令的語法樹。在這個過程當中，堆疊的深度最多會達到 6 層，如圖 8.10 所示。

在剖析的過程當中，堆疊 stack 的高度原則上就是剖析樹的深度，堆疊隨著剖析函數 parseXXX() 的呼叫而成長，並隨著函數的離開而縮短，於是，堆疊與樹的成長形成一種緊密的關係，整個遞迴下降法與堆疊的成長過程，形成一種類似深度優先搜尋 (Depth First Search) 的建構順序。

雖然本書當中只討論了遞迴下降剖析法，但這種方法並不是唯一的剖析法。大致上來說，剖析的方法可以分為兩類，第一類稱為由上而下的剖析法，第二類稱為由下而上的剖析法。

由上而下的剖析法乃是從整個程式的最高節點開始，企圖比對最上層的節點，因此，一開始就會從完整程式這個最高節點開始比對，逐步向下遞迴比對，直到最底層的樹葉節點，像是遞迴下降剖析法，還有 LL(1)、… LL(K) 等方法，都是屬於由上而下的方式。

由下而上的剖析法則恰好相反，從最底層的規則開始比對，然後，不斷透過比對與組合的方式，向上建構出更高層的節點，直到整個程式成為單一棵大樹為止，由下而上的剖析方法有很多類，像是運算子優先順序法 (Operator Precedence Parser)、Shift-Reduce Parser、 ，LR(1)、…、LR(K) 等，都是由下而上的方式。

在本書中，我們並不介紹 LL 與 LR 等較為複雜的方法，有興趣的讀者請進一步閱讀編譯器的專門書籍。

8.4 語意分析

當語法樹建立完成後，緊接著通常會進行語意分析的動作。語意分析必須確定每個節點的型態，並且檢查這些型態是否可以相容，然後才輸出具有標記的語法樹，也就是語意樹。圖 8.11 顯示了語意分析的輸入與輸出，其輸入為圖 8.11 (a) 當中的語法樹，而輸出則是圖 8.11 (b) 當中的語意樹。

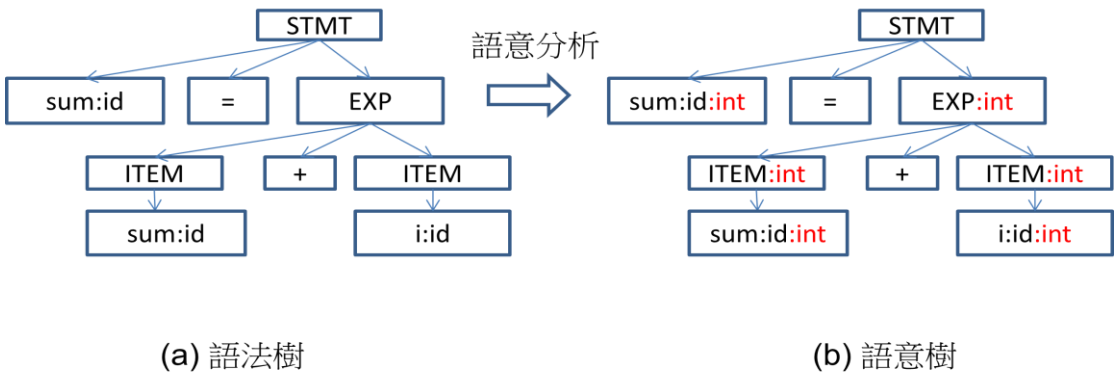

圖 8.11 語意分析：將語法樹標註上節點型態

在將語法樹轉成語意樹的過程中，必須檢查型態的相容問題，舉例而言，我們通

常不能將一個整數與一個字串變數進行相乘的動作，因為這兩種型態是無法相乘的。所以，當您撰寫了範例 8.2 這樣的程式並進行編譯時，語意分析器將會發現到 a 與 b 是無法相乘的，因而輸出錯誤訊息。

範例 8.2 語法正確但語意錯誤的程式範例

C 語言程式	語意分析的結果

int a=5, c;
char b[10];
c = a * b;
	a, c 是整數
	b 是字元陣列
	c (型態錯誤) = a (整數) * b (字元陣列)

8.5 中間碼產生

一但語意樹建立完成，我們就可以利用程式將樹中的每個節點，展開為中間碼，其方法是利用遞迴的方式，從根節點開始，遞迴的展開每個子節點，直到所有節點的中間碼都產生完畢為止。

讓我們看看範例 8.3 的程式與其中間碼，該中間碼採用的是後置式語法[4]，因此，最後一個參數通常是目標運算元，而第一個參數則固定是運算符號。

在範例 8.3 的中間碼內，FOR0 是迴圈的起始標記，_FOR0 是迴圈的結束標記；而 T0 是臨時變數，用來儲存計算時的中間結果；JEQ 是條件式跳躍指令，JMP 是跳躍指令，這與 CPU0 的組合語言很像，只是採用了後置式的語法。

範例 8.3 將 C0 語言編譯成中間碼的範例

	C0 語言程式	中間碼	說明
1	sum = 0;	= 0 sum	設定 sum 為 0
2	for (i=1;i<=10;i++)	= 1 i	設定 i 為 1
3	{	FOR0:	for 迴圈的起始點
4	sum = sum + i;	CMP i 10	i >=< 10 ?
5	}	J > _FOR0	if (i>10) goto FOR1
6	return sum;	+ sum i T0	T0=sum + i
7		= T0 sum	sum = T0
8		+ i 1 i	i = i + 1

[4] 在組合語言當中，到底應採用前置式或後置式語法，並沒有特別的理由，像是 GNU 的組合語言就採用後置式，而微軟的組合語言則採用前置式。其實，只要能夠前後一致，不要讓程式設計師無所適從就可以了。在中間碼的表示上，由於大多數人習慣採用後置式，因此我們也採用後置式的寫法，以便與此習慣一致。

9 10 11		J FOR0 _FOR0: RET sum	 傳回 sum

在範例 8.3 的中間碼內，我們直接用+, - , *, / … 等符號代表運算名稱，而不需要像 CPU0 使用 ADD, SUB, MUL, DIV 等英文詞彙。而且，所有的運算可以直接在變數當中進行，並不需要仰賴暫存器，這讓中間碼產生器不需要考慮暫存器的分配問題，可以簡化程式碼產生器的複雜度。

那麼，要怎樣才能產生中間碼呢？其過程與直譯器的方法類似，都是利用遞迴的方式，從代表整個程式的根節點開始，檢視每一個語法樹上的節點，然後以遞迴的方式產生對應的目標碼。

舉例而言，假如我們想處理 STMT = id '=' EXP 這個規則，那麼，首先要先判斷一個節點是否為 STMT 節點，若是，則以遞迴的方式 expVar=generate(exp) 產生程式碼，並傳回結果變數 expVar，然後將結果變數 expVar 存入 id 所代表的變數中，其演算法片段如圖 8.12 所示。

演算法	說明

Algorithm generate(node)
…
else if (node.tag=STMT)
id = node.childs[0].token;
exp = node.childs[2];
expVar = generate(exp);
pcode("", "=", expVar, "", id);
return expVar;
…
End Algorithm
	
	處理 STMT 陳述
	取得 id
	取得 EXP 節點
	產生 EXP 的程式，並傳回變數 (例如 T0)
	產生指定敘述 (例如 =T0 sum)
	傳回臨時變數 (例如 T0)

圖 8.12 產生 id=EXP 中間碼的演算法

圖 8.13 顯示了 C0 語言的中間碼產生演算法，該演算法單獨處理了 FOR、STMT、COND、EXP 等語法節點，也處理了 id, number 等詞彙規則的節點。

中間碼產生的演算法	說明
Algorithm generate(node) if (node.tag=FOR) stmt1 = node.childs[2]	產生中間碼的演算法 處理 FOR 迴圈 語法：for (STMT;

cond = node.childs[4]	COND;
stmt2 = node.childs[6]	STMT)
block = node.childs[8]	BLOCK
generate(stmt1);	產生 STMT 的程式
tempForCount = forCount++;	取得下一個 for 標記代號
forBeginLabel = "+FOR" + tempForCount;	for 迴圈的起始標記(+FOR0)
forEndLabel = "-FOR"+tempForCount;	for 迴圈的結束標記(-FOR0)
pcode(forBeginLabel+":", "", "", "", "");	輸出迴圈起頭標記 (FOR0:)
condOp = generate(cond);	產生比較指令 (CMP i 10)
negateOp(condOp, negOp);	將運算反向 (i<=10 變 >)
pcode("", "J", negOp, "", forEndLabel);	輸出跳離指令 (J > _FOR0)
generate(block, nullVar);	產生 BLOCK 的程式
generate(stmt2, nullVar);	產生 STMT 的程式
pcode("", "J", "", "", forBeginLabel);	跳回到迴圈起頭 (J +FOR0)
pcode(forEndLabel, "", "", "", "");	輸出迴圈結束標記 (-FOR0:)
return NULL;	for 迴圈無傳回值
else if (node.tag=STMT)	處理 STMT 陳述
id = node.childs[0].token;	id = EXP | id [++|--]
if (node.childs[1].tag = "=")	如果 id 之後為等號，id=EXP
exp = node.childs[2];	取得 EXP 節點
expVar = generate(exp);	產生 EXP 的程式
pcode("", "=", expVar, "", id);	產生指定敘述 (= T0 sum)
return expVar;	傳回臨時變數 (T0)
else	否則，id [++|--]
op1 = node.childs[1].token;	取得運算碼 (++ 或 --)
pcode("", op1[0], id.value, "1", id.value)	輸出運算指令 (+ i 1 i)
return id;	傳回運算變數 (i)
else if (node.tag=COND)	處理布林判斷式 COND =
expVar1 = generate(node->childs[0])	EXP
op = node->child[1];	[==|!=|<=|>=|<|>]
expVar2 = generate(node->childs[2]);	EXP
pcode("", "CMP", expVar1, expVar2, "");	輸出比較指令 (CMP i 10)
return op.value	傳回比較運算 (例如 <=)
else if node.tag in [EXP]	處理算式 EXP=ITEM([+-*/]ITEM)?
item1 = node.childs[0];	取得 ITEM
var1 = generate(item1);	產生第一個運算元的程式
for (ti=1; ti<node.childs.Count; ti+=2)	針對後續的([+-*/]ITEM)?
op = node.childs[ti].token;	取得 [+-*/]

```
        item2 = node.childs[ti + 1];          取得 ITEM
        var2 = generate(item2);               產生 ITEM 的中間碼
        tempVar = nextTempVar();              取得新的臨時變數
        pcode("", op, var1, var2, tempVar);   輸出運算指令(+ sum i T0)
        var1 = tempVar;                       設定新臨時變數為傳回值
      return var1;                          傳回結果 (T0)
   else if (node.tag in [number|id])        遇到變數或常數
      return node.token;                    傳回其 token 名稱
   else if (node.childs != null)            針對其他狀況，若有子代
      foreach (child in node.childs)        則對每個子代
         generate(child);                      遞迴產生程式
      return null;                          不傳回值
   else return null;                        否則，不傳回值
End Algorithm

Algorithm pcode                             演算法 pcode()
Input label, op, params                     輸入：標記、運算、參數
   If (label is not empty)                  如果有標記
      output label+":"                      就輸出標記到中間檔
   output op, param[0], param[1], param[2]  輸出中間碼
End Algorithm
```

圖 8.13 中間碼產生的演算法

細心的讀者可能會發現我們沒有處理 PROG、BaseList、BASE、BLOCK、ITEM 等節點，原因是這些節點的處理非常簡單，只要遞迴產生子節點的中間碼即可，因此，我們統一由 foreach (child in node.childs) generate(child); 這個敘述處理掉了，這樣就可以簡化圖 8.13 的演算法，讓整個程式看起來較短一些。

根據圖 8.13 的演算法，讀者可以對照範例 8.1 的程式，追蹤其中間碼產生的過程，以便理解該演算法的意義，並學習中間碼產生器的撰寫方式。

8.6 組合語言產生

一旦中間碼產生完畢，程式就可以輕易的將中間碼轉換成組合語言，但是，若要考慮最佳化與暫存器配置等問題，那麼，轉換的過程就變得困難許多。為了簡單起見，在本節中，我們首先看看沒有最佳化的組合語言產生方法，然後在下一節當中才討論有關最佳化的主題。

要將中間碼轉換為組合語言，只要根據中間碼的運算，將中間碼翻譯為組合語言就可以了，範例 8.4 顯示了這個翻譯過程，其中 (a) 欄為中間碼，而 (b) 欄則是轉換後的組合語言。

範例 8.4 將中間碼轉換為組合語言的範例

	(a) 中間碼	(b) 組合語言 (無最佳化)
1	= 0 sum	LDI R1 0
2		ST R1 sum
3	= 0 i	LDI R1 0
4		ST R1 i
5		
6		
7	FOR0:	FOR0:
8		LD R1 i
9		LDI R2 10
10	CMP i 10	CMP R1 R2
11	J > _FOR0	JGT _FOR0
12		LD R1 sum
13		LD R2 i
14	+ sum i T0	ADD R3 R1 R2
15		ST R3 T0
16		LD R1 T0
17	= T0 sum	ST R1 sum
18		LD R1 i
19		LDI R2 1
20	+ i 1 i	ADD R3 R2 R1
21		ST R3 i
22	J FOR0	JMP FOR0
23	_FOR0:	_FOR0:
24		LD R1 sum
25	RET sum	RET
26		sum:RESW 1
27		i: RESW 1
28		T0: RESW 1

在範例 8.4 當中，中間碼 = 0 sum 被翻譯成 { LDI R1 0; ST R1 sum} 等兩個指令，

而中間碼 + sum i T0 這個指令，則被翻譯成 { LD R1 sum; LD R2 i; ADD R3 R1 R2; ST R3 T0 } 等四個指令，這是因為中間碼的加法運算指令 (+) 可以直接存取變數，但在 CPU0 的組合語言當中，ADD 指令卻只能以暫存器作為運算參數，因此，我們必須先將 sum, i 載入到暫存器 R1, R2 之後，才執行 ADD 指令，最後，還必須將位於暫存器 R3 中的結果存回變數 T0 當中。

圖 8.14 顯示了一個可將中間碼 p-code 轉換為 CPU0 組合語言的演算法，在該演算法當中，完全沒有使用最佳化的功能，因此產生的組合語言相當的冗長，假如我們將範例 8.4 (a) 的每一行中間碼都傳給圖 8.14 的演算法進行轉換，則轉換後的結果就會是範例 8.4 (b) 的組合語言。

中間碼轉組合語言的演算法	說明
Algorithm pcodeToAsm	將 pcode 轉換為組合語言
Input label, op, p1, p2, p3	輸入：label 標記、op 運算、p1:參數 1…
if (label is not empty)	
output(label)	如果有標記 (例如 FOR0)
	輸出標記
if (op is "=")	
rewrite(LD R1, p1)	如果是指定運算 = (例如 = T0 sum)
rewrite(ST R1, p3)	(例如：輸出 LD R1, T0)
	(例如：輸出 ST R1, sum)
else if (op in [+-*/])	
rewrite(LD R1, p1)	如果是加減乘除 (例如：+ sum i T0)
rewrite(LD R2, p2)	(例如：輸出 LD R1, sum)
rewrite(ASM(op), R3, R1, R2)	(例如：輸出 LD R2, i)
rewrite(ST R3, p3)	(例如：輸出 ADD R3, R1, R2)
	(例如：輸出 LD R1, sum)
else if (op is CMP)	
rewrite(LD R1, p1)	如果是 CMP 比較 (例如：CMP i 10)
rewrite(LD R2, p2)	(例如：輸出 LD R1, i)
rewrite(CMP R1, R2)	(例如：輸出 LDI R2, 10)
	(例如：輸出 CMP R1, R2)
else if (op is J)	
jop = AsmJumpOp(op, p1);	如果是跳躍 (例如：J > _FOR0)
rewrite(jop, p3)	(例如：將 J > 改為 JGT)
	(例如：輸出 JGT _FOR0)
else if (op is RET)	
rewrite(LD R1, p3)	如果是 RET (例如：RET sum)
rewrite(RET)	(例如：輸出 LD R1, sum)
	(例如：輸出 RET)
End Algorithm	
Algorithm rewrite(op, p1, p2, p3)	
if (op is LD) and isNumber(p2)	(例如：LD R2, 10 改為 LDI R2, 10)

op = LDI output(op, p1, p2, p3) End Algorithm	如果 op 是 LD 且 p2 是整數 將 op 改為 LDI 輸出該指令

圖 8.14 將中間碼轉換為 CPU0 組合語言的演算法

8.7 最佳化

對於商業用的編譯器而言，最佳化是非常重要的功能，否則編譯出來的執行檔將會又大又慢，因此編譯器必須盡可能的將輸出的組合語言最佳化，以提高程式的效率。

範例 8.5 顯示了兩個 CPU0 組合語言的對照版本，其中 (b) 欄是沒有最佳化的組合語言，而 (c) 欄則是最佳化後的結果。這兩個組合語言都是由 (a) 欄的中間碼轉換而來的，您可以看出有最佳化的版本程式較短，而且效率會明顯變好。

範例 8.5 最佳化的範例

	(a) 中間碼	(b) 組合語言 (無最佳化)	(c) 組合語言 (有最佳化)
1	= 0 sum	LDI R1 0	LDI R1, 0
2		ST R1 sum	ST R1, sum
3	= 0 i	LDI R1 0	LDI R2, 0
4		ST R1 i	ST R2, i
5			LDI R3, 1
6			LDI R4, 10
7	FOR0:	FOR0:	
8		LD R1 i	
9		LDI R2 10	
10	CMP i 10	CMP R1 R2	CMP R2, R4
11	J > _FOR0	JGT _FOR0	JGT _FOR0
12		LD R1 sum	
13		LD R2 i	
14	+ sum i T0	ADD R3 R1 R2	ADD R1, R1, R2
15		ST R3 T0	
16		LD R1 T0	
17	= T0 sum	ST R1 sum	
18		LD R1 i	
19		LDI R2 1	

20	+ i 1 i	ADD R3 R2 R1	ADD R2, R2, R3
21		ST R3 i	
22	J FOR0	JMP FOR0	JMP _FOR0
23	_FOR0:	_FOR0:	
24		LD R1 sum	ST R1, sum
25	RET sum	RET	RET
26		sum:RESW 1	i: RESW 1
27		i: RESW 1	sum:RESW 1
28		T0: RESW 1	

仔細閱讀範例 8.5 (b) 的『組合語言(無最佳化)』一欄，讀者會看到許多『載入』(LD) 與『儲存』(ST) 的動作。這些動作會不斷從記憶體載入資料到暫存器內，然後在運算後又立刻將暫存器的結果存回記憶體。像是『LD R1 sum; LD R2 i; ADD R3 R2 R1; ST R3 T0』 這四個指令，其實只是為了實作出中間碼的 『+ sum i T0』一個指令而已，但是卻耗費了額外的三個指令進行載入與儲存的動作。

其實，只要有足夠的暫存器，經過適當的安排，就能省去過程中的載入儲存指令。舉例而言，像是『+ sum i T0; = T0 sum』等兩中間個指令，在無最佳化的版本中，其對應的程式碼為『LD R1 sum; LD R2 i; ADD R3 R1 R2; ST R3 T0; LD R1 T0; ST R1 sum』等六個指令，但是在最佳化後的版本當中，竟然被濃縮為『ADD R1, R1, R2』這個單一的指令。

最佳化版本可以這麼精簡的原因，是由於變數 sum 與 i 早已被載入到 R1, R2 當中，而且在整個段落當中 R1, R2 都一直保持著與 sum, i 兩者的對應關係，所以就不需要重新載入或儲存 sum 與 i。整個最佳化的版本都利用這樣的方式，大量的減少了組合語言指令的數量，這樣的作法除了能增快執行速度之外，還能降低程式所佔的空間。

從範例 8.5 當中，我們可以看出最佳化功能的用途，只要經過適當的暫存器安排，就能讓輸出的組合語言程式變得更簡單。有效的分配暫存器，並加以充分利用，是最佳化程式的重要功能之一。

另外，如果能將某些暫存器初值的設定動作，從迴圈中提出到迴圈之外，也可以提升整體的程式效率，像是範例 8.5 (c) 的『組合語言 (有最佳化)』一欄中的『LDI R3, 1; LDI R4, 10』等兩個指令，可以放在迴圈中設定，也可以放在迴圈之外。但是，放在迴圈之外的方法較有效率，因為只會被執行一次，如果放在迴圈之中，那麼，每次當迴圈重複執行時，就會再度執行一次已經算過的結果，因而浪費了 CPU 時間。如果能將所有『不變量』提出到迴圈之外，就能增快程式的執行速度，

這也是最佳化功能的任務之一。

至此，我們已經完整的說明了編譯器的理論與演算法，有關進一步的編譯器方法，像是最佳化與 LL、LR 剖析法等技術，請讀者參考專門的編譯器書籍。

8.8 實務案例

在本節中，我們將說明 gcc 編譯器的設計原理，並且示範如何將 C 語言程式轉換成中間碼[5]與組合語言，以及如何進行最佳化等動作，透過這樣的實作，讓讀者實際感受編譯器的設計原理與使用方法，以便讓用實務印證上述的編譯器理論。

8.8.1 gcc 編譯器

編譯器 gcc 是 GNU 工具的核心程式，除了可以編譯 C 語言之外，也提供將 C 語言轉換為組合語言的功能，甚至可以直接組譯組合語言，這些功能對學習系統程式有很大的幫助，因為 gcc 讓我們可以將編譯、組譯、連結等動作合併或分開進行，讓我們得以觀察許多中間過程，以便理解系統程式的編譯、組譯、連結等觀念。

中間碼

在 2004 年之前，gcc 原本只採用了一層稱為 RTL 的中間碼，但是在 gcc 4.0 版之後，則將中間碼增加到三層，這三層的中間碼分別是 Generic、Gimple 與 RTL 中間碼，其中的 Generic 代表剖析樹，Gimple 是高階中間碼，而 RTL 則是低階中間碼，整個 gcc 的編譯過程大致上如圖 8.15 所示。

[5] RTL Representation, http://gcc.gnu.org/onlinedocs/gccint/RTL.html

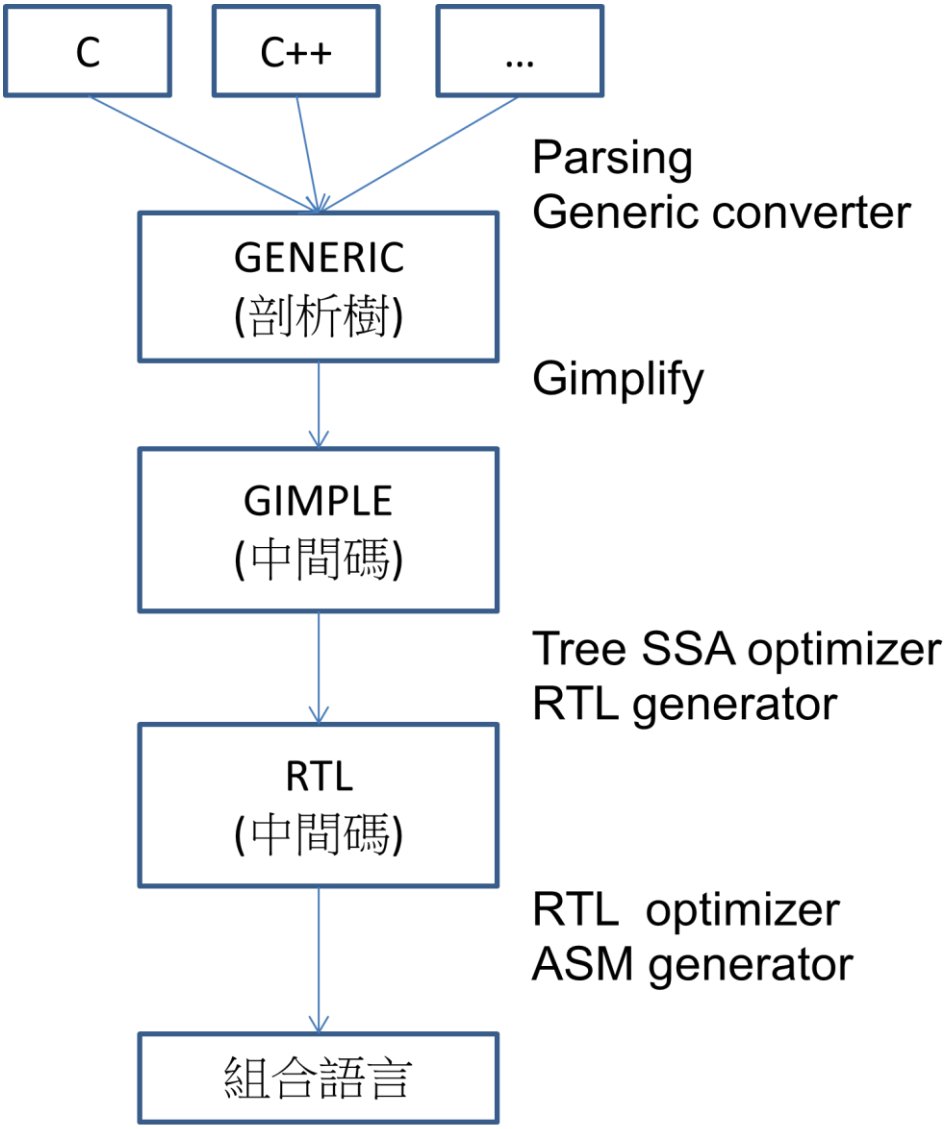

圖 8.15 GNU 編譯器的流程

在圖 8.15 當中，我們看到 GNU 用了『Generic、Gimple、RTL』等結構，這些結構通常儲存在編譯器當中而沒有被輸出，但是在概念上都可以表示為某種中間碼，其中的 RTL 結構較為複雜，所以我們先將焦點集中在 Generic 與 Gimple 上。

範例 8.6 顯示了 GNU 的 Generic 與 Gimple 結構的表示法，讀者可以看到 Generic 結構其實是一種剖析樹的輸出格式，而 Gimple 則是某類似 p-code 的中間碼格式。

範例 8.6 gcc 編譯器的中間碼格式

(a) C 語言	(b) Generic 中間碼	(c) Gimple 中間碼

if (foo(a+b, c)) {
  c = b++ / a;
	if (foo (a + b,c))
	  c = b++ / a
		t1 = a + b
		t2 = foo (t1, c)

} return c;	endif return c	if (t2 != 0) <L1,L2> L1: t3 = b b = b + 1 c = t3 / a goto L3 L2: L3: return c

當剖析動作完成之後，GNU 的剖析器會將剖析後的結果表達為 Generic 格式，Generic 是一種和語言無關的剖析樹結構，舉例而言，GNU 工具當中其實包含了 C/C++/Obj C/Java 等編譯器，這些編譯器在剖析動作完成後，都會將程式轉換成 Generic 結構，然後再交由後續的程式處理，因此 Generic 可以說是一種標準的語法樹結構。

當 GNU 的程式產生器接收到 Generic 語法樹之後，就會將語法樹轉換為 Gimple 中間碼，這個過程被 GNU 稱為 Gimplify。在 Gimplify 的過程當中，主要是將 for, while 等區塊結構，轉換為 if 與 goto 所組成的指令序列，Gimple 中間碼其實與本章中所使用的 p-code 中間碼相當類似，幾乎就是同一個東西。

Gcc 的最佳化動作有兩段，第一段是在將 Gimple 轉為 RTL 前，先用 Tree SSA optimizer 進行最佳化，第二段是在 RTL 轉回組合語言前，使用 RTL optimizer 進行最佳化的動作。

Tree SSA optimizer 當中的 SSA 是靜態單一賦值 (Static Single Assignment) 的意思，其意義是將一個變數分為很多的版本，每個版本只能被指定一次，舉例而言，假如將 x 變數分為 x.1, x.2, ….x.n，那麼每個變數 x.i 就可以只被用 x.i=y 指定一次。

當一個 Gimple 中間碼被轉換成 SSA 形式，也就是為每個變數加上版本時，只要能利用最佳化程式，降低變數的版本數量，就能達到減少載入指令的功能，達到最佳化的效果。

接著，RTL generator 會將轉換成 SSA 形式的 Gimple 中間碼，進一步轉換為 RTL 中間碼，然後再利用 RTL optimizer 進行最佳化動作。RTL 是一種形式較為複雜的中間碼，其中的每個節點都被加上了型態的描述，舉例而言，當範例 8.7 (a) 的 Gimple 指令 b = a-1 被轉換成 RTL 時，會轉換成範例 8.7 (b) 中的形式，這

個形式看起來複雜了許多，但實際上並沒有那麼難懂，讓我們稍做說明，您就可以輕易的看懂 RTL 中間碼了。

範例 8.7 中間碼 Gimple 與 RTL 之對照範例

(a) Gimple	(b) RTL	(c) 簡化後的 RTL
b = a – 1	(set (reg/v:SI 59 [b]) (plus:SI (reg/v:SI 60 [a] (const_int -1 [0xffffffff]))))	b (59,reg/v:SI) = a (60, reg/v:SI) + -1 (const_int)

RTL 當中的 set 代表指定陳述 "="，plus 代表加法運算，reg/v 代表該變數可以存在暫存器 (register) 或記憶體 (variable) 中，而 SI (Single Integer) 則代表長度為 4 bytes 的整數。因此範例 8.7 (b) 的 RTL 指令，可以轉換為範例 8.7 (c) 的 Gimple 寫法如下。

b (59,reg/v:SI) = a (60, reg/v:SI) + -1 (const_int)

這種寫法應該簡單多了，也就是將 b, a, 與 -1 等符號，加上代號與類型限制，舉例而言，b (59,reg/v:SI) 這個句子，代表 b 是編號 59 號的變數，可以被儲存在暫存器或記憶體當中，而且其形態為 4bytes 的整數。

我們可以利用 –dr 參數，要求 gcc 編譯器輸出 rtl 中間碼，以下是筆者的操作過程，您可以看到其中的 sum.c.01.rtl 檔案被產生出來，該檔案就是 RTL 中間碼。

指令與操作過程	說明
C:\ch08>gcc -c -dr sum.c -o sum.o C:\ch08>dir … 2010/03/12 上午 09:00 105 sum.c 2010/04/09 上午 09:29 3,784 sum.c.01.rtl 2010/04/09 上午 09:29 372 sum.o 3 個檔案 4,261 位元組 3 個目錄 9,196,593,152 位元組可用	編譯並用 –dr 參數要求輸出中間碼 RTL 中間碼檔案

由於 RTL 的中間碼很冗長，在此我們只列出其中的一小段片段，我們並不嘗試解讀這個 RTL 檔案，請有興趣的讀者使用 gcc 指令產生 RTL 檔後自行研究其內容。

範例 8.8 C 語言與其 RTL 片段

(a) C 語言程式	(b) 對應的 RTL 檔案

int sum(int n) {
  int s=0;
  int i;
  for (i=1; i<=n;i++) {
  s = s + i;
  }
  return s;
}
	(note 2 0 3 NOTE_INSN_DELETED)
	…
	(insn 8 6 11 (set (mem/f:SI (plus:SI (reg/f:SI 54 virtual-stack-vars)
	(const_int -4 [0xfffffffc])) [0 s+0 S4 A32])
	(const_int 0 [0x0])) -1 (nil)
	(nil))
	…
	(jump_insn 16 15 17 (set (pc)
	(if_then_else (gt (reg:CCGC 17 flags)
	(const_int 0 [0x0]))
	(label_ref 30)
	(pc))) -1 (nil)
	(nil))
	…

在 RTL 中間碼內，包含了語意分析的型態標記，因此在 RTL 處理時早已進行過語意分析階段了，這讓 RTL optimizer 可以進行語意相關的最佳化功能，而且其最佳化動作與處理器的種類無關。

Gcc 的最佳化功能

gcc 提供了四個層級的最佳化功能，包含完全不最佳化，以及 -O1, -O2, -O3 等不同等級的最佳化功能，讓我們利用 –S 參數，將最佳化的結果以組合語言輸出，真槍實彈的觀察 gcc 的最佳化結果。

範例 8.9 的 (a) 欄顯示了一個具有函數 f() 的 C 語言程式，然而，函數 f 雖然做了一些計算，但實際上傳回值固定為 14，我們試圖利用這個函數測試 gcc 的最佳化能力，看看 gcc 的最佳化能做到何種程度。

我們分別用 -O0 的無最佳化與–O3 的最高等級最佳化進行編譯，其結果如範例 8.9 的 (b), (c) 欄所示，讀者可以看到在 (c) 欄的 optimize_O3.s 中，f() 函數除了前後的堆疊框架處理之外，幾乎只剩下了 movl $14, %eax 這個指令，該指令直接將 f() 的傳回值 14 塞入到 %eax 暫存器後傳回，這顯示了 gcc 的 -O3 編譯方式具有很好的最佳化能力。

範例 8.9 gcc 不同層級的最佳化實例

編譯指令(無最佳化)： gcc -S optimize.c -o optimize_O0.s -O0
編譯指令(O3 級最佳化)： gcc -S optimize.c -o optimize_O3.s -O3

(a) optimize.c

```
int f() {
   int a=3, b=4, c, d;
   c=a+b;
   d=a+b;
   return c+d;
}
```

(b) optimize_O0.s (無最佳化)

```
        .file   "optimize.c"
        .text
.globl _f
        .def    _f;    .scl    2;     .type
        32;    .endef
_f:
        pushl   %ebp
        movl    %esp, %ebp
        subl $16, %esp
        movl    $3, -4(%ebp)
        movl    $4, -8(%ebp)
        movl    -8(%ebp), %eax
        addl -4(%ebp), %eax
        movl    %eax, -12(%ebp)
        movl    -8(%ebp), %eax
        addl -4(%ebp), %eax
        movl    %eax, -16(%ebp)
        movl    -16(%ebp), %eax
        addl -12(%ebp), %eax
        leave
        ret
```

(c) optimize_O3.s (有最佳化)

```
        .file   "optimize.c"
        .text
        .p2align 4,,15
.globl _f
        .def    _f;    .scl    2;     .type
        32;    .endef
_f:
        pushl   %ebp
        movl    $14, %eax
        movl    %esp, %ebp
        popl %ebp
        ret
```

在編譯器最佳化的議題上，有許多相關的研究與技術，若要更深入的理解這些技術，請進一步參考編譯器的相關書籍，本書將不作進一步的介紹。

習題

8.1 請為 C0 語言加上 if 語句的 EBNF 語法，加入到圖 8.2 中。

8.2 接續上題，請在圖 8.9 當中加入剖析 if 語句的演算法。

8.3 接續上題，請在圖 8.13 當中加入將 if 語句轉為中間碼的演算法。

8.4 請為範例 8.5 (b) 的無最佳化組合語言，提出一個簡單的最佳化機制，並寫出您的最佳化方法實施後，所產生的組合語言程式碼。

8.5 請使用 gcc 的 -O0 與 -O3 參數，分別已無最佳化與高級最佳化的方式，編譯任意一個 C 語言程式為組合語言，並觀察其編譯後的組合語言，指出最佳

化後哪些指令被省略了。