第12章 系統軟體實作

在本書中，我們已經說明了許多系統軟體的設計原理，包含組譯器、連結器、載入器、巨集處理器、編譯器、虛擬機器、作業系統等，相信讀者應該已經理解了這些系統軟體的設計原理了，接著，讓我們開始進入實作的領域。

在本章中，我們將利用 C 語言實作一些較簡單且重要的系統軟體，這包含組譯器 (12.2 節)、虛擬機器 (12.3 節)、剖析器 (12.4 節)、編譯器 (12.5 節) 等。筆者希望能透過這些範例性的實作，向讀者展示系統軟體的實作方法與技巧，以便讓讀者能更熟悉系統軟體的原理，進一步培養開發系統軟體的能力，並且能以實作印證理論，以理論支持實作，達到相輔相成的目的。

12.1 簡介

在本章中，我們將透過實作的方式，直接以 C 語言程式展現系統軟體的設計方法。當然，我們的實作都經過簡化，像是編譯器所採用的 C0 語言就不是一個完整的高階語言，只是為了本書說明方便而設計的一種最小型語言。

即使如此，為了實做這些系統軟體，我們必須撰寫不少程式，雖然筆者已盡可能的簡化其設計方式，但每個軟體仍需約 100~300 行的程式碼。表格 12.1 顯示了本章中筆者所實作的所有程式，包含基礎函式庫、動態陣列、雜湊表、指令表、組譯器、虛擬機器、掃描器、剖析器、程式產生器等物件。

表格 12.1 本章中所實作程式列表，位於光碟的/ch12/ 資料夾中。

物件	檔案	說明
基礎函式庫	Lib.c, Lib.h	包含字串、記憶體與檔案處理函數
動態陣列	Array.c, Array.h	可動態成長的陣列結構
雜湊表	HashTable.c, HashTable.h	利用二維 Array 物件建構的雜湊表
指令表	OpTable.c, OpTable.h	CPU0 的指令表
組譯器	Assembler.c Assembler.h	AS0 組譯器，可組譯 CPU0 的組合語言
虛擬機器	Cpu0.c, Cpu0.h	CPU0 的虛擬機器，可執行 AS0 組譯出來的目的檔
掃描器	Scanner.c, Scanner.h	C0 語言的掃描器，可將程式切割成詞彙 (token)
語法樹	Tree.c, Tree.h	代表語法樹的物件，可以有很多子樹

剖析器	Parser.c, Parser.h	C0 語言的剖析器，可將程式轉換成語法樹
程式產生器	Generator.c, Generator.h	C0 語言的程式產生器，可將語法樹轉換成 pcode 與組合語言
編譯器	Compiler.c, Compiler.h	C0 語言的編譯器，可將程式編譯為組合語言
主程式	main.c	所有系統軟體的主程式，包含測試程式、編譯器主程式、組譯器主程式與虛擬機器主程式，可使用條件編譯的方式產生上述四種不同的執行檔。

雖然 C 語言並不支援物件導向的語法，但是在本實作中，我們仍然採用了物件導向式的撰寫方式，將每個物件的資料都封裝成一個結構型態 typedef struct {…} <ClassName>，然後撰寫對應的成員函數，我們會以 <ClassName> <MethodName> (…) 作為函數名稱，以方便記憶與使用。

由於標準 C 語言函式庫 (Ansi C) 當中缺乏一些基本資料結構的相關函數，因此我們必須先設計出這些資料結構，像是動態陣列 (Array)、雜湊表 (Hash Table) 等，以便在後續的組譯器、虛擬機、剖析器與編譯器中，可以很容易的利用這些結構存放符號表、指令表、詞彙串列、剖析樹等資料。

以下我們將以範例說明動態陣列與雜湊表的使用方式，讓讀者先行熟悉這兩個結構，以便作為理解本章實作的基礎。

動態陣列

範例 12.1 是一個用物件導向觀念撰寫 C 語言程式的實例，我們宣告了 Array 這個動態陣列結構，然後宣告了 ArrayNew() 這個建構函數與 ArrayFree() 這個解構函數。接著，開始宣告 Array 類別的成員函數，包含 ArrayAdd(), ArrayPush(), ArrayPop(), ArrayPeek(), ArrayLast(), ArrayEach() 等。透過這樣的方式，我們可以在 C 語言當中模仿物件導向的精神，建構出物件導向式的 C 語言函式庫。

範例 12.1 動態陣列的程式表頭

檔案：ch12/Array.h

```
…
typedef struct {
  int size;      // 陣列目前的上限
  int count;     // 陣列目前的元素個數
```

```
    void **item; // 每個陣列元素的指標
} Array;            // 動態陣列的資料結構
…
Array* ArrayNew(int size);// 建立新陣列
void ArrayFree(Array *array); // 釋放該陣列
void ArrayAdd(Array *array, void *item); // 新增一個元素
void ArrayPush(Array *array,void *item); // (模擬堆疊) 推入一個元素
void* ArrayPop(Array *array);   //(模擬堆疊) 彈出一個元素
void* ArrayPeek(Array *array); //(模擬堆疊) 取得最上面的元素
void* ArrayLast(Array *array); // 取得最後一個元素
void ArrayEach(Array *array, FuncPtr1 f); //對每個元素都執行 f 函數
…
```

在範例 12.2 的 Array.h 檔案中，定義了 Array 這個資料結構，該結構會利用 void** item 這個 void 指標陣列存放任意型態的資料。Array 中的 size 變數是陣列的大小，而 count 則是該陣列目前包含的元素個數。

為何我們要自行實作 Array 類別呢？這是由於 ANSI C 只提供固定大小的靜態陣列，在使用靜態陣列儲存資料之前，我們必須先知道陣列的大小上限是多少。但這是很令人困擾的。舉例而言，在組譯器的符號表中，我們無法事先預知會有多少個符號，因此，無法知道符號表的大小上限，硬性的規定一個上限往往會太大，消耗太多的記憶體，而且也無法保證符號表的大小永遠不會超過該上限。

因此，我們在程式 Array.c 中實作了動態陣列，其中主要的函數為 ArrayAdd() 函數，當該函數發現該陣列空間不足時，會將陣列擴大兩倍，然後將舊陣列的元素搬入後，才加入新元素，如此，就能製作出一個不限大小的動態陣列。

範例 12.2 動態陣列的程式主體 - ArrayAdd() 函數

檔案 ch12/Array.c 中的 ArrayAdd() 函數	說明
void ArrayAdd(Array *array, void *item) {	加入 item 到 array 當中
if (array->count == array->size) {	如果大小不夠了
int itemSize = sizeof(void*);	
int newSize = array->count*2;	擴大 size 為兩倍
void **newItems = ObjNew(void*, newSize);	取得兩倍大小的空間
memcpy(newItems,	將舊資料複製到新陣列中
array->item, array->size*itemSize);	
free(array->item);	釋放原本的陣列
array->item = newItems;	將指標指向新陣列

array->size = newSize;
}
array->item[array->count++] = item;
}
	設定新陣列的大小
	
	將 item 加入陣列中

有了動態陣列 Array 之後，我們就可以很方便的將任意資料儲存在 Array 中，而不必事先預測陣列的大小上限。範例 12.4 是我們用來對 Array 類別進行單元測試的 ArrayTest() 函數，該函數會將 names 中的人名全部加入 array 中，然後利用 ArrayPop() 彈出最後的 Bob，再利用 ArrayFind() 搜尋出每個人名在 array 中的位置，其執行結果如範例 12.4 所示。

範例 12.3 動態陣列的測試程式- ArrayTest() 函數

檔案 ch12/Array.c 中的 ArrayTest()函數

```
void ArrayTest() {
   char *names[] = { "John", "Mary", "George", "Bob" };
   Array *array = ArrayNew(1);
   int i;
   for (i=0; i<4; i++)
      ArrayAdd(array, names[i]);
   ArrayEach(array, StrPrintln);
   printf("ArrayPop()=%s\n", ArrayPop(array));
   printf("ArrayLast()=%s\n", ArrayLast(array));
   for (i=0; i<4; i++) {
      int arrayIdx = ArrayFind(array, names[i], strcmp);
      printf("ArrayFind(%s)=%d\n", names[i], arrayIdx);
   }
   ArrayEach(array, StrPrintln);
   ArrayFree(array);
}
```

範例 12.4 ArrayTest() 函數的執行結果

ArrayTest() 函數的執行結果	說明
John Mary George Bob ArrayPop()=Bob ArrayLast()=George	利用 ArrayEach(…) 印出的所有元素 取出最後的 Bob 印出最後一個元素為 George

ArrayFind(John)=0 ArrayFind(Mary)=1 ArrayFind(George)=2 ArrayFind(Bob)=-1 John Mary George	利用 ArrayFind(…) 找尋所有元素 由於 Bob 已經取出了，所以找不到 再度印出的所有元素 (Bob 已去除)

仔細觀察範例 12.3 的程式，讀者可能會發現我們用了一個較難懂的函數 ArrayEach()，例如 ArrayEach(array, StrPrintln) 可以將該陣列的元素當作字串型態印出。這種設計方式是從 Ruby 語言所學習過來的，由於 Ruby 具有一個 C 語言所沒有的 yield 指令，因此我們利用函數指標的方式，搭配反向呼叫 (Callback Function) 機制，讓該函數指標循序處理陣列中的每一個元素。ArrayEach() 的程式原始碼很簡單，如範例 12.5 所示。

範例 12.5 ArrayEach() 函數的原始碼

檔案 ch12/Array.c 中的 ArrayEach() 函數	說明

void ArrayEach(Array *array, FuncPtr1 f) {
  int i;
  for (i=0; i<array->count; i++)
  f(array->item[i]);
}
	傳入函數指標 f
	
	對 array 中的每個元素
	  都傳給 f() 函數處理一次

在 ArrayEach() 中，參數 f 的型態是 FuncPtr1，FuncPtr1 代表具有一個參數的函數指標，其定義方式在 Lib.h 檔案中，由 typedef int (*FuncPtr1) (const void *) 這個型別指令所定義，其原始程式片段如範例 12.6 所示。

範例 12.6 FuncPtr1 函數指標的定義方式

檔案 ch12/Lib.h 中的函數指標定義區域	說明

…
// 函數指標 (用於 ArrayEach(), HashTableEach()中)
typedef int (*FuncPtr1) (const void *);
typedef int (*FuncPtr2) (const void *, const void *);
int StrPrint(const void *data);
int StrPrintln(const void *data);
…
	
	函數指標：單一參數
	函數指標：兩個參數
	印出字串的函數
	印出字串並換行的函數

函數指標是 C 語言當中的進階功能，但卻是相當重要的一部分，有效的利用函數

指標可以讓演算法更為通用，像是標準函式庫中的快速排序法 qsort() 就利用傳入具有兩個參數的指標 compare()，以便對任意型態進行排序。qsort() 的函數原型如下：

```
void qsort(void *buf, size_t n,size_t size,int (*cmp)(const void*, const void *));
```

在範例 12.6 的實作中，也宣告了 FuncPtr2 型態代表具有兩個參數的函數指標，然後，宣告了 StrPrint 與 StrPrintln 兩個函數，可用來傳入 ArrayEach() 函數中，以便印出這些結構的每一個元素。

雜湊表

有了動態陣列，我們就可以很容易的實作出雜湊表 (Hash Table)，雜湊表中的基本元素是索引鍵與資料 (key, data) 所形成的配對 (Entry)，如範例 12.7 的 Entry 結構所示，該結構在概念上是一個物件導向的類別。Entry 類別有兩個基本的成員函數，EntryNew() 與 EntryCompare()，EntryNew() 可以建立一個新的配對，而 EntryCompare() 則可以比較兩個配對的索引鍵 (key) 是否相同。

範例 12.7 雜湊表的資料結構與函數定義

檔案 ch12/HashTable.h 中的雜湊表定義部分	說明

typedef struct {
  char *key;
  void *data;
} Entry;
	雜湊表中的一個配對記錄
	  索引鍵 (key)
	  內容值 (data)
	稱為 Entry。

Entry* EntryNew(char *key, void *data); int EntryCompare(Entry *e1, Entry *e2);	Entry 的建立函數 Entry 的比較函數
int hash(char *key, int range);	雜湊函數：字串 key 的雜湊值
#define HashTable Array	雜湊表是由二維動態陣列 所組成的
HashTable* HashTableNew(int size);	雜湊表建構函數
void HashTableFree(HashTable *table);	雜湊表解構函數
void* HashTablePut(HashTable *table, char *key, void *data);	新增 (key, data)
void *HashTableGet(HashTable *table, char *key);	取得 data
void HashTableEach(HashTable *table, FuncPtr1 f);	對每個元素執行 f
Array* HashTableToArray(HashTable *table);	將雜湊表轉為陣列

在我們的實作中，雜湊表本身就是一個動態陣列 (Array)，因此，我們直接利用 #define HashTable Array 的方式，將雜湊表 HashTable 定義為 Array。但不同的是，雜湊表中的每一個元素又是一個動態陣列，因此，雜湊表是由動態陣列所形成的二維陣列，但與一般二維陣列有所不同的是，第二維的大小是不固定的。

雜湊表 HashTable 也是一個物件導向概念上的類別，包含建構函數 HashTableNew() 與 HashTableFree() 解構函數，並且擁有 HashTablePut(), HashTableGet(), HashTableEach(), HashTableToArray() 等函數，範例 12.7 的最後顯示了這些成員函數的原型定義。

雜湊表是依靠雜湊函數 (Hash Function) 所計算出來的值，以決定要將某個元素放在哪一列中，雜湊函數的分布越平均越好，這樣才能增快搜尋時間，我們所使用的雜湊函數很簡單，就是將字串 key 中的每個位元組相加後，對雜湊表的大小取餘數後的結果，範例 12.8 顯示了該雜湊函數的原始碼。

範例 12.8 雜湊函數實作方式

檔案 ch12/HashTable.c 中的雜湊函數 hash()	說明

int hash(char *key, int range) {
  int i, hashCode=0;
  for (i=0; i<strlen(key); i++) {
  BYTE value = (BYTE) key[i];
  hashCode += value;
  hashCode %= range;
  }
  return hashCode;
}
	雜湊函數
	  設定 hashCode 初始值為 0
	對 key 中的每個位元
	  轉換為無號數
	  將其與 hashCode 相加
	  對 range 取餘數
	
	傳回雜湊值 hashCode

雜湊表的兩個主要函數是 HashTablePut() 與 HashTableGet()，我們可以利用 HashTablePut() 將 (key, data) 配對放入雜湊表當中，然後利用 HashTableGet() 函數查詢某元素是否存在，範例 12.9 顯示了這兩個成員函數的原始程式碼。

範例 12.9 雜湊表的主要成員函數 HashTableGet() 與 HashTablePut()

```
// 尋找雜湊表中 key 所對應的元素並傳回
void *HashTableGet(HashTable *table, char *key) {
  int slot = hash(key, table->size);                  // 取得雜湊值 (列號)
  Array *hitArray = (Array*) table->item[slot]; // 取得該列
```

```
  // 找出該列中是否有包含 key 的索引值，若有則傳回
  int keyIdx = ArrayFind(hitArray, EntryNew(key, ""), EntryCompare);
  if (keyIdx >= 0) { // 若有，則傳回該配對的資料元素
    Entry *e = hitArray->item[keyIdx];
    return e->data;
  }
  return NULL; // 否則傳回 NULL
}

// 將 (key, data) 配對放入雜湊表中
void HashTablePut(HashTable *table, char *key, void *data) {
  Entry *e;
  int slot = hash(key, table->size);                    // 取得雜湊值 (列號)
  Array *hitArray = (Array*) table->item[slot]; // 取得該列
  int keyIdx = ArrayFind(hitArray, EntryNew(key, ""), EntryCompare);
  if (keyIdx >= 0) { // 若有，則以新資料取代該配對資料
    e = hitArray->item[keyIdx];
    e->data = data;
  } else {
    e= EntryNew(key, data);              // 建立配對
    ArrayAdd(table->item[slot], e); // 放入對應的列中
  }
}
```

HashTableGet(table, key) 的實作方法，是利用雜湊函數 hash(key, table->size) 計算出該元素會被放在哪一列中，然後，在該列中搜尋是否有關鍵值為 key 的元素。

HashTablePut(table, key, data) 的作法類似 HashTableGet()，但是當 key 已經存在時，會用新的 data 元素取代舊的 data 元素，所以 HashTable 中總是保存最新的一筆具有關鍵值 key 的資料 data，後來者會覆蓋掉原先的資料。

12.2組譯器實作

在本節當中，我們將實際撰寫一個簡單的 CPU0 組譯器，以便印證 2, 3, 4 章當中的組譯器理論。

要撰寫 CPU0 的組譯器，必須參考附錄 A 中的指令表以及指令格式。因為組譯器必須負責將指令轉換為機器碼，所以必須知道運算碼、暫存器、常數欄位等各佔據哪些位元，如此才能順利的撰寫程式。

我們所撰寫的組譯器之資料結構宣告在 Assembler.h 檔中，而程式主要位於 Assembler.c 中，我們用 make 專案的方式可以建構出 as0.exe 這個代表組譯器的執行檔，其執行方法為 as0 <asmFile> <objFile>。

舉例而言，我們可以利用 as0 ArraySum.asm0 ArraySum.obj0 這個指令，將組合語言檔案 ArraySum.asm0 組譯為目的檔 ArraySum.obj0，其組譯報表如範例 12.10 所示，我們刻意在組譯報表中將目的碼與組合語言指令對齊，以方便讀者閱讀。

實際上 *.obj0 的目的檔是一個二進位的檔案，這些檔案都在本書的光碟中可以找到，但若用一般的文字編輯器檢視目的檔會看到亂碼，必須用 16 進位的文字編輯器才能看到其 16 進位編碼，像是 Notepad++中就有個外掛模組能以 16 進位的方式顯示目的檔 ArraySum.obj0。

範例 12.10 利用 as0 組譯器組譯 ArraySum.asm0 檔案為目的檔 ArraySum.obj0

組譯指令：as0 ArraySum.asm0 ArraySum.obj0		
位址	組合語言檔案 ch12/ArraySum.asm0	目的檔 ch12/ArraySum.obj0
0000	LDI R1, 0	08100000
0004	LD R2, aptr	002F003C
0008	LDI R3, 3	08300003
000C	LDI R4, 4	08400004
0010	LDI R9, 1	08900001
0014	FOR:	
0014	LD R5, [R2] ; R5=*aptr	00520000
0018	ADD R1,R1,R5 ; R1+=*aptr	13115000
001C	ADD R2, R2, R4 ; R2+=4	13224000
0020	SUB R3, R3, R9 ; R3--;	14339000
0024	CMP R3, R0 ; if (R3!=0)	10309000
0028	JNE FOR ; goto FOR	21FFFFE8
002C	ST R1, sum ; sum=R1	011F0018
0030	LD R8, sum ; R8=sum	008F0014
0034	RET	2C000000
0038	a: WORD 3, 7, 4	00000003000000700000004
0044	aptr: WORD a	00000038
0048	sum: WORD 0	00000000

範例 12.10 也顯示了指令的位址，這只是為了讓讀者閱讀方便用的，組譯器的第一階段 PASS1，必須能計算出每個指令的位址，接著再經過 PASS2 之後，才產生出目的碼。

為了簡單起見，as0 組合語言所有的標記後必須加上 ":" 符號，像是範例 12.10 中的 FOR:、a: aptr:、sum: 等，這讓 as0 組譯器可以很容易的解析並取出各個欄位。

在宣告檔 Assembler.h 中，我們宣告了組譯器物件 (Assembler) 與指令物件 (AsmCode)，這兩個物件是組譯器的核心，另外我們也使用了陣列物件 (Array) 與雜湊表物件 (HashTable)，以便儲存符號表 (symTable), 指令表 (opTable) 與指令陣列 (codes)。

範例 12.11 CPU0 組譯器的資料結構與函數

檔案：Assembler.h	說明

typedef struct {
  Array *codes;
  HashTable *symTable;
  HashTable *opTable;
} Assembler;
	組譯器物件
	  指令物件串列
	  符號表
	  指令表

typedef struct {
  int address, opCode, size;
  char *label, *op, *args, type;
  char *objCode;
} AsmCode;
	指令物件
	  包含位址、運算碼、
	  空間大小、op、標記、
	  參數、型態、目的碼
	  等欄位

void assemble(char *asmFile, char *objFile);	組譯器的主程式
Assembler* AsmNew(); void AsmFree(Assembler *a);	組譯器的建構函數 組譯器的解構函數

void AsmPass1(Assembler *a, char *text);
void AsmPass2(Assembler *a);
void AsmSaveObjFile(Assembler *a, char *objFile);
void AsmTranslateCode(Assembler *a, AsmCode *code);
	組譯器的第一階段
	組譯器的第二階段
	儲存目的檔
	將指令轉為目的碼

AsmCode* AsmCodeNew(char *line);
void AsmCodeFree(AsmCode *code);
int AsmCodePrintln(AsmCode *code);
	指令物件的建構函數
	指令物件的解構函數
	指令物件的列印函數

組譯器的主要函數 assemble() 可將輸入的組合語言檔 asmFile 轉換成目的檔 objFile，該函數首先會利用 newFileStr(fileName) 函數讀取整個組合語言檔，成為單一個 C 語言的字串，然後再建立組譯器物件 (Assembler *a)，接著進行第一階段 AsmPass1() 與第二階段 AsmPass2() 的處理，最後將目的碼輸出到 objFile 中，範例 12.12 顯示了 assemble() 函數的程式碼。

範例 12.12 CPU0 組譯器的主要函數 assemble()

檔案 ch12/Assembler.c 中的 assemble() 函數	說明

void assemble(char *asmFile, char *objFile) {
  printf("Assembler:asmFile=%s objFile=%s\n",asmFile,objFile);
  printf("==========Assemble==========\n");
  char *text = newFileStr(asmFile);
  Assembler *a = AsmNew();
  AsmPass1(a, text);
  printf("==========SYMBOL TABLE=========\n");
  HashTableEach(a->symTable, (FuncPtr1) AsmCodePrintln);
  AsmPass2(a);
 	
  AsmSaveObjFile(a, objFile);
  AsmFree(a);
  freeMemory(text);
}
	組譯器函數
	輸入組合語言
	輸出目的檔
	讀取檔案到
	  text 字串中
	第一階段
	  計算位址
	印出符號表
	第二階段
	  建構目的碼
	輸出目的檔
	釋放 a 的記憶體
	釋放 text

組譯器第一階段的工作主要是計算每個指令的位址，並且建立符號表，範例 12.13 顯示了組譯器第一階段的主要程式 AsmPass1()，該程式主要呼叫 AsmCodeSize() 函數以計算指令所占空間的大小。

範例 12.13 CPU0 組譯器的第一階段 AsmPass1() – 編位址

檔案 ch12/Assembler.c 中的 AsmPass1() 函數	說明

void AsmPass1(Assembler *a, char *text) {
  int i, address = 0, number;
  Array* lines = split(text, "\r\n", REMOVE_SPLITER);
  ArrayEach(lines, strPrintln);
  printf("=====PASS1=====\n");
	第一階段的組譯
	
	將組合語言分割成一行一行

```
   for (i=0; i<lines->count; i++) {                    對於每一行
         strReplace(lines->item[i], SPACE, ' ');
         AsmCode *code = AsmCodeNew(lines->item[i]);    建立指令物件
         code->address = address;                       設定該行的位址
         Op *op = HashTableGet(opTable, code->op);      取得運算碼與型態
         if (op != NULL) {
            code->opCode = op->code;                    設定指令物件的
            code->type = op->type;                         運算碼與型態
         }
         if (strlen(code->label)>0)                     如果有標記符號
            HashTablePut(a->symTable, code->label, code);   加入符號表中
         ArrayAdd(a->codes, code);                      建構指令物件陣列
         AsmCodePrintln(code);                             印出觀察
         code->size = AsmCodeSize(code);                計算指令大小
         address += code->size;                         下一個指令位址
   }
   ArrayFree(lines, strFree);                           釋放記憶體
}
…
int AsmCodeSize(AsmCode *code) {                        計算指令的大小
   switch (code->opCode) {                              根據運算碼 op
      case OP_RESW :                                     如果是 RESW
         return 4 * atoi(code->args);                       大小為 4*保留量
      case OP_RESB :                                    如果是 RESB
         return atoi(code->args);                           大小為 1*保留量
      case OP_WORD :                                    如果是 WORD
         return 4 * (strCountChar(code->args, ",")+1);      大小為 4*參數個數
      case OP_BYTE :                                    如果是 BYTE
         return strCountChar(code->args, ",")+1;            大小為 1*參數個數
      case OP_NULL :                                    如果只是標記
         return 0;                                          大小為 0
      default :                                         其他情形 (指令)
         return 4;                                          大小為 4
   }
}
```

組譯器第二階段的工作，是利用指令表 opTable 與第一階段所計算出來的位址，進行指令的編碼。範例 12.14 顯示了 Assembler.c 檔中的第二階段主要函數

AsmPass2()，以及其中用來將指令轉為目的碼的 AsmTranslateCode() 函數，AsmTranslateCode() 會根據指令的類型 (A 型, J 型, L 型)，以決定如何編定指令碼。

範例 12.14 CPU0 組譯器的第二階段 AsmPass2() – 指令轉機器碼

檔案 ch12/Assembler.c 中的 AsmPass2() 函數	說明
void AsmPass2(Assembler *a) {	組譯器的第二階段
printf("=============PASS2==============\n");	
int i;	
for (i=0; i<a->codes->count; i++) {	對每一個指令
AsmCode *code = a->codes->item[i];	
AsmTranslateCode(a, code);	進行編碼動作
AsmCodePrintln(code);	
}	
}	
void AsmTranslateCode(Assembler *a, AsmCode *code) {	指令的編碼函數
…	…
char cxCode[9]="00000000", objCode[100]="";	宣告變數
…	…
int pc = code->address + 4;	提取後 PC 為位址+4
switch (code->type) {	根據指令型態
case 'J' :	處理 J 型指令
if (!strEqual(args, "")) {	
labelCode= HashTableGet(a->symTable,args);	取得符號位址
cx = labelCode->address - pc;	計算 cx 欄位
sprintf(cxCode, "%8x", cx);	
}	
sprintf(objCode, "%2x%s", code->opCode, &cxCode[2]);	編出目的碼(16 進位)
break;	
…	
case 'A' :	處理 A 型指令
sscanf(args, "%s %s %s", p1, p2, p3);	取得參數
sscanf(p1, "R%d", &ra);	取得 ra 暫存器代號
sscanf(p2, "R%d", &rb);	取得 rb 暫存器代號
if (sscanf(p3, "R%d", &rc)<=0)	取得 rc 暫存器代號
sscanf(p3, "%d", &cx);	或者是 cx 參數
sprintf(cxCode, "%8x", cx);	
sprintf(objCode, "%2x%x%x%x%s",	編出目的碼(16 進位)

code->opCode,ra,rb,rc,&cxCode[5]);
break;
case 'D' : {
switch (code->opCode) {
case OP_RESW:
case OP_RESB:
memset(objCode, '0', code->size*2);
objCode[code->size*2] = '\0';
break;
case OP_WORD:
format = format4;
case OP_BYTE:
…
…
code->objCode = newStr(objCode);
}
	如果是資料宣告
	註：我們將資料宣告 RESW, RESB, WORD, BYTE 也視為一種指令，其形態為 D
	RESW,RESB : 目的碼為 0000….
	WORD , BYTE : 其目的碼為每個資料轉為 16 進位的結果
	…
	設定目的碼到指令物件 AsmCode 中

最後，我們會利用 AsmSaveObjFile(a, objFile) 函數，將產生出來的 16 進位格式目的碼，轉換成二進位格式後寫入目的檔 objFile 中，於是就完成了組譯器 as0 的整個設計。

12.3 虛擬機實作

當我們用 as0 組譯出了目的碼之後，這個目的碼仍然無法被執行，因為我們所使用的處理器並不是 CPU0，為了讓讀者能執行 CPU0 的目的碼，筆者只好繼續撰寫一個虛擬機器 vm0，以便讓讀者執行 as0 所組譯出的目的碼，本節將描述 vm0 的使用方式與設計原理。

要撰寫 CPU0 的虛擬機，同樣必須參考附錄 A 中的指令表與指令格式。因為虛擬機必須負責解讀指令中的各個欄位，包含指令碼 (op)、暫存器、常數欄位等。虛擬機必須從代表指令暫存器的變數中，取出這些欄位，然後才能根據指令碼，執行對應的動作。

我們所撰寫的 CPU0 虛擬機器 vm0 的資料結構宣告在 Cpu0.h 檔中，而程式主要位於 Cpu0.c 中。我們用 make 專案的方式可以建構出 vm0.exe 這個代表組譯器的執行檔，其執行方法為 vm0 <objFile>。

舉例而言，當我們用 vm0 ArraySum.obj0 執行上一節中所組譯出的目的檔時，可以看到如範例 12.15 的結果，左欄是執行時印出的印出 PC, IR 與目標暫存器等資訊，右欄則是被執行的指令，讀者應可看到 R[1] 暫存器的值從 0, 3, 10, 到 14 的過程，也就是將陣列 a: WORD 3, 7, 4 一路加總，最後 R[1] 當中存有 3+7+4 = 14 的結果，正是計算出來的總和值。

範例 12.15 利用 vm0 執行目的檔 ArraySum.obj0

虛擬機器執行指令：vm0 ArraySum.obj0	
執行過程	說明 (對應指令)
===VM0:run ArraySum.obj0 on CPU0===	
PC=00000004 IR=08100000 SW=00000000 R[01]=0X00000000=0	LDI R1, 0
PC=00000008 IR=002F003C SW=00000000 R[02]=0X00000038=56	LD R2, aptr
PC=0000000C IR=08300003 SW=00000000 R[03]=0X00000003=3	LDI R3, 3
PC=00000010 IR=08400004 SW=00000000 R[04]=0X00000004=4	LDI R4, 4
PC=00000014 IR=08900001 SW=00000000 R[09]=0X00000001=1	LDI R9, 1
PC=00000018 IR=00520000 SW=00000000 R[05]=0X00000003=3	FOR: LD R5, [R2]
PC=0000001C IR=13115000 SW=00000000 R[01]=0X00000003=3	ADD R1,R1,R5
PC=00000020 IR=13224000 SW=00000000 R[02]=0X0000003C=60	ADD R2, R2, R4
PC=00000024 IR=14339000 SW=00000000 R[03]=0X00000002=2	SUBR3, R3, R9
PC=00000028 IR=10309000 SW=00000000 R[12]=0X00000000=0	CMP R3, R0
PC=00000014 IR=21FFFFE8 SW=00000000 R[15]=0X00000014=20	JNE FOR
PC=00000018 IR=00520000 SW=00000000 R[05]=0X00000007=7	FOR: LD R5, [R2]
PC=0000001C IR=13115000 SW=00000000 R[01]=0X0000000A=10	ADD R1,R1,R5
PC=00000020 IR=13224000 SW=00000000 R[02]=0X00000040=64	ADD R2, R2, R4
PC=00000024 IR=14339000 SW=00000000 R[03]=0X00000001=1	SUBR3, R3, R9
PC=00000028 IR=10309000 SW=00000000 R[12]=0X00000000=0	CMP R3, R0
PC=00000014 IR=21FFFFE8 SW=00000000 R[15]=0X00000014=20	JNE FOR
PC=00000018 IR=00520000 SW=00000000 R[05]=0X00000004=4	FOR: LD R5, [R2]
PC=0000001C IR=13115000 SW=00000000 R[01]=0X0000000E=14	ADD R1,R1,R5
PC=00000020 IR=13224000 SW=00000000 R[02]=0X00000044=68	ADD R2, R2, R4
PC=00000024 IR=14339000 SW=00000000 R[03]=0X00000000=0	SUBR3, R3, R9
PC=00000028 IR=10309000 SW=40000000 R[12]=0X40000000=1073741824	CMP R3, R0
PC=0000002C IR=21FFFFE8 SW=40000000 R[15]=0X0000002C=44	JNE FOR
PC=00000030 IR=011F0018 SW=40000000 R[01]=0X0000000E=14	ST R1, sum
PC=00000034 IR=008F0014 SW=40000000 R[08]=0X0000000E=14	LD R8, sum
PC=00000038 IR=2C000000 SW=40000000 R[00]=0X00000000=0	RET
===CPU0 dump registers===	

IR =0x2c000000=738197504	
R[00]=0x00000000=0	
R[01]=0x0000000e=14	sum=R1=3+7+4=14
R[02]=0x00000044=68	R2=陣列 a 的結尾
R[03]=0x00000000=0	
R[04]=0x00000004=4	R4 = 4
R[05]=0x00000004=4	
R[06]=0x00000000=0	
R[07]=0x00000000=0	
R[08]=0x0000000e=14	R8=R1=sum
R[09]=0x00000001=1	R9=1
R[10]=0x00000000=0	
R[11]=0x00000000=0	
R[12]=0x40000000=1073741824	SW (R12) 的 Z=1
R[13]=0x00000000=0	
R[14]=0xffffffff=-1	RET 位址為-1，結束
R[15]=0x00000038=56	PC 最後為 0x38

虛擬機器 Vm0 的主要物件稱為 Cpu0，定義於 Cpu0.h 檔案中，其中包含指令表 opTable，指令暫存器 IR，代表暫存器 R0, R1, …, R15 的陣列 R[16]，還有代表記憶體的 BYTE *m 陣列，範例 12.16 顯示了 Cpu0.h 的宣告內容。

範例 12.16 CPU0 虛擬機器的資料結構與主要函數

檔案 ch12/Cpu0.h	
typedef struct {	CPU0 虛擬機器 Vm0 的主要結構
BYTE *m;	代表記憶體的陣列
int mSize;	記憶體的大小
int R[16], IR;	R0, R1, …, R15, IR 暫存器
} Cpu0;	
void runObjFile(char *objFile);	虛擬機器 Vm0 的主要函數，可執行目的檔 objFile
Cpu0* Cpu0New(char *objFile);	建立 CPU0 物件並載入目的檔
void Cpu0Free(Cpu0 *cpu0);	釋放 CPU0 物件
void Cpu0Run(Cpu0 *cpu0, int startPC);	從 startPC 位址開始執行程式
void Cpu0Dump(Cpu0 *cpu0);	印出所有暫存器

由於 as0 組譯器所輸出的目的檔格式是極為簡單的映像檔，因此我們可以直接

將目的檔載入 m 陣列中即可開始執行，不需要進行任何的重定位動作。範例 12.17 顯示了 CPU0 虛擬機器 VM0 的主要程式檔 Cpu0.c 的全部程式，讀者可以看到這種模擬是非常直接且簡單的，如果不考慮速度與安全性的問題，那麼虛擬機器的撰寫是非常容易的。

範例 12.17 CPU0 虛擬機器 VM0 的主要程式

檔案 ch12/Cpu0.c	說明
#include "Cpu0.h"	
void runObjFile(char *objFile) {	虛擬機器主函數
printf("===VM0:run %s on CPU0===\n", objFile);	
Cpu0 *cpu0 = Cpu0New(objFile);	建立 CPU0 物件
Cpu0Run(cpu0, 0);	開始執行
Cpu0Dump(cpu0);	傾印暫存器
Cpu0Free(cpu0);	釋放記憶體
}	
Cpu0* Cpu0New(char *objFile) {	建立 CPU0 物件
Cpu0 *cpu0=ObjNew(Cpu0, 1);	分配記憶體
cpu0->m = newFileBytes(objFile, &cpu0->mSize);	載入 objFile
return cpu0;	
}	
void Cpu0Free(Cpu0 *cpu0) {	釋放 CPU0 物件
freeMemory(cpu0->m);	
ObjFree(cpu0);	
}	
#define bits(i, from, to) \	取得 from 到 to
((UINT32) i << (31-to) >> (31-to+from))	之間的位元
#define ROR(i, k) \	向右旋轉 k 位元
(((UINT32)i>>k)|(bits(i,32-k, 31)<<(32-k)))	
#define ROL(i, k) \	向左旋轉 k 位元
(((UINT32)i<<k)|(bits(i,0,k-1)<<(32-k)))	
#define SHR(i, k) ((UINT32)i>>k)	向右移位 k 位元
#define SHL(i, k) ((UINT32)i<<k)	向左移位 k 位元
#define bytesToInt32(p) \	4 byte 轉 int
(INT32)(p[0]<<24|p[1]<<16|p[2]<<8|p[3])	

#define bytesToInt16(p) (INT16)(p[0]<<8|p[1])	2 byte 轉 INT16
#define int32ToBytes(i, bp) \	int 轉為 4 byte
{ bp[0]=i>>24; bp[1]=i>>16; bp[2]=i>>8; bp[3]=i;}	
#define StoreInt32(i, m, addr) \	i=m[addr…addr+3]
{ BYTE *p=&m[addr]; int32ToBytes(i, p); }	
#define LoadInt32(i, m, addr) \	m[addr..addr+3]=i
{ BYTE *p=&m[addr]; i=bytesToInt32(p); }	
#define StoreByte(b, m, addr) \	m[addr]=b
{ m[addr] = (BYTE) b; }	
#define LoadByte(b, m, addr) { b = m[addr]; }	b=m[addr]
#define PC R[15]	PC is R[15]
#define LR R[14]	LR is R[14]
#define SP R[13]	SP is R[13]
#define SW R[12]	SW is R[12]
void Cpu0Run(Cpu0 *cpu0, int start) {	虛擬機器的主要執行函數
char buffer[200];	
unsigned int IR, op, ra, rb, rc, cc;	
int c5, c12, c16, c24, caddr, raddr;	
unsigned int N, Z;	
BYTE *m=cpu0->m;	
int *R=cpu0->R;	
PC = start;	設定起始位址準備開始執行
LR = -1;	
BOOL stop = FALSE;	
while (!stop) {	如果尚未結束
R[0] = 0;	R[0] 永遠為 0
LoadInt32(IR, m, PC);	指令擷取
cpu0->IR = IR;	IR←m[PC..PC+3]
PC += 4;	擷取完將 PC 加 4，到下一個指令
op = bits(IR, 24, 31);	取得 op 欄位
ra = bits(IR, 20, 23);	取得 Ra 欄位
rb = bits(IR, 16, 19);	取得 Rb 欄位,
rc = bits(IR, 12, 15);	取得 Rc 欄位
c5 = bits(IR, 0, 4);	取得 5,12,16,24
c12= bits(IR, 0, 11);	位元的常數欄位
c16= bits(IR, 0, 15);	

```
c24= bits(IR, 0, 23);
N    = bits(SW, 31, 31);                    取得狀態暫存器中的 N 旗標
Z    = bits(SW, 30, 30);                    取得狀態暫存器中的 Z 旗標
if (bits(IR, 11, 11)!=0) c12 |= 0xFFFFF000;  若為負數，則調整為 2 補數格式
if (bits(IR, 15, 15)!=0) c16 |= 0xFFFF0000;
if (bits(IR, 23, 23)!=0) c24 |= 0xFF000000;
caddr = R[rb]+c16;                          取得位址[Rb+cx]
raddr = R[rb]+R[rc];                        取得位址[Rb+Rc]
switch (op) {                               根據 op 執行動作
   case OP_LD : LoadInt32(R[ra], m, caddr); break;     處理 LD 指令
   case OP_ST : StoreInt32(R[ra], m, caddr); break;    處理 ST 指令
   case OP_LB : LoadByte(R[ra], m, caddr); break;      處理 LB 指令
   case OP_SB : StoreByte(R[ra], m, caddr); break;     處理 SB 指令
   case OP_LDR: LoadInt32(R[ra], m, raddr); break;     處理 LDR 指令
   case OP_STR: StoreInt32(R[ra], m, raddr); break;    處理 STR 指令
   case OP_LBR: LoadByte(R[ra], m, raddr); break;      處理 LBR 指令
   case OP_SBR: StoreByte(R[ra], m, raddr); break;     處理 SBR 指令
   case OP_LDI: R[ra] = c16; break;                    處理 LDI 指令
   case OP_CMP: {                                      處理 CMP 指令
      if (R[ra] > R[rb]) {                              > : SW(N=0, Z=0)
         SW &= 0x3FFFFFFF;                               設定 N=0, Z=0
      } else if (R[ra] < R[rb]) {                       < : SW(N=1, Z=0, ....)
         SW |= 0x80000000;                               設定 N=1;
         SW &= 0xBFFFFFFF;                               設定 Z=0;
      } else {                                          = : SW(N=0, Z=1)
         SW &= 0x7FFFFFFF;                               設定 N=0;
         SW |= 0x40000000;                               設定 Z=1;
      }
      ra = 12;                                          在指令執行完後輸出 R12
      break;
   }
   case OP_MOV: R[ra] = R[rb]; break;                  處理 MOV 指令
   case OP_ADD: R[ra] = R[rb] + R[rc]; break;          處理 ADD 指令
   case OP_SUB: R[ra] = R[rb] - R[rc]; break;          處理 SUB 指令
   case OP_MUL: R[ra] = R[rb] * R[rc]; break;          處理 MUL 指令
   case OP_DIV: R[ra] = R[rb] / R[rc]; break;          處理 DIV 指令
   case OP_AND: R[ra] = R[rb] & R[rc]; break;          處理 AND 指令
   case OP_OR:    R[ra] = R[rb] | R[rc]; break;        處理 OR 指令
```

case OP_XOR: R[ra] = R[rb] ^ R[rc]; break;	處理 XOR 指令
case OP_ROL: R[ra] = ROL(R[rb],c5); break;	處理 ROL 指令
case OP_ROR: R[ra] = ROR(R[rb],c5); break;	處理 ROR 指令
case OP_SHL: R[ra] = SHL(R[rb],c5); break;	處理 SHL 指令
case OP_SHR: R[ra] = SHR(R[rb],c5); break;	處理 SHR 指令
case OP_JEQ: if (Z==1) PC += c24; break;	處理 JEQ 指令
case OP_JNE: if (Z==0) PC += c24; break;	處理 JNE 指令
case OP_JLT: if (N==1&&Z==0) PC += c24; break;	處理 JLT 指令
case OP_JGT: if (N==0&&Z==0) PC += c24; break;	處理 JGT 指令
case OP_JLE:	處理 JLE 指令
if ((N==1&&Z==0)||(N==0&&Z==1))	
PC+=c24;	
break;	
case OP_JGE:	處理 JGE 指令
if ((N==0&&Z==0)||(N==0&&Z==1))	
PC+=c24;	
break;	
case OP_JMP: PC+=c24; break;	處理 JMP 指令
case OP_SWI: LR = PC; PC=c24; break;	處理 SWI 指令
case OP_JSUB:LR = PC; PC+=c24; break;	處理 JSUB 指令
case OP_RET: if (LR<0) stop=TRUE;	處理 RET 指令
else PC=LR;	
break;	
case OP_PUSH:SP-=4;	處理 PUSH 指令
StoreInt32(R[ra], m, SP); break;	
case OP_POP: LoadInt32(R[ra], m, SP);	處理 POP 指令
SP+=4; break;	
case OP_PUSHB:SP--;	處理 PUSHB 指令
StoreByte(R[ra], m, SP); break;	
case OP_POPB:LoadByte(R[ra], m, SP);	處理 POPB 指令
SP++; break;	
default: printf("Error:invalid op (%02x) ", op);	
}	
sprintf(buffer, "PC=%08x IR=%08x SW=%08x	印出 PC, IR, SW, R[ra], R[rb]
R[%02d]=0x%08x=%d\n",	暫存器的值，以利觀察
PC, IR, SW, ra, R[ra],R[ra]);	
strToUpper(buffer);	
printf(buffer);	

}
}
 	
void Cpu0Dump(Cpu0 *cpu0) {
printf("\n===CPU0 dump registers===\n");
printf("IR =0x%08x=%d\n", cpu0->IR, cpu0->IR);
int i;
for (i=0; i<16; i++)
printf("R[%02d]=0x%08x=%d\n",
i,cpu0->R[i],cpu0->R[i]);
}
	印出所有暫存器的值，以利觀察
	
	印出 IR
	
	印出 R0~R15

舉例而言，當我們希望模擬 ADD Ra, Rb, Rc 的結果時，只要用 R[ra] = R[rb] + R[rc] 一行程式就可以做完了，而像是 LD Ra, [Rb+Cx]，也只要用 caddr = R[rb]+c16; LoadInt32(R[ra], m, caddr) 這樣的動作就能模擬完畢，於是我們在 Cpu0.c 檔案中模擬出 CPU0 的所有個指令，程式並不算很長。

必須注意的是，我們必須讓虛擬機器 VM0 可以在某個時候跳出程式，否則將會產生當機的情形。因為真實的 CPU 是不斷執行，除非關掉電源，否則永遠都不會停的。因此我們選擇在程式 Cpu0Run() 這個函數開始執行時，用 LR = -1 這個指令將連結暫存器設定為一個非法的負值 (照理講 LR 是 PC 的備份，不應該有負值的)，然後當 RET 指令執行時，若發現返回時 LR 是一個負值，則直接跳出程式，這樣當虛擬機器 VM0 試圖模擬 RET 指令時，如果發現沒有上一層了，就會直接跳出程式，這相當符合我們對 RET 指令的期待。

我們採用很簡單的方式實作了 CPU0 的虛擬機器 VM0，以便解決無法執行 CPU0 目的檔的問題。另外，我們還希望透過這樣的實作方式，說明虛擬機的設計原理。

12.4 剖析器實作

剖析器是編譯器與直譯器中的關鍵程式，在本節與下一節當中，我們將採用遞迴下降 (Recursive Descendent) 的方法，實作一個 C0 語言的編譯器 c0c.exe，以印證編譯器的理論。

要使用遞迴下降的方法撰寫 C0 語言的剖析器，必須使用附錄 B 中的 EBNF 語法 (圖 B.2)。請讀者先行閱讀該語法之後，再行閱讀本節的內容。如果能將圖 B.2 印下來加以對照，那將有助於理解本節的程式。

剖析器 Parser 是編譯器 C0C 的一個主要物件，我們在 Parser 中使用了詞彙掃描器 Scanner 進行掃描的動作，以便將程式分解成詞彙串列 (tokens)，首先讓我們來看看掃描器的實作方式。

詞彙掃描器

詞彙掃描器 Scanner 是一個較為簡單的物件，其資料結構定義在 Scanner.h 檔案中，如範例 12.18 所示。其中包含程式字串 (text)、程式長度 (textLen)、目前掃描位置 (textIdx) 與詞彙 token 等。

範例 12.18 C0 語言掃描器的資料結構

檔案 Scanner.h	說明

typedef struct {
  char *text;
  int textLen;
  int textIdx;
  char token[MAX_LEN];
} Scanner;
	掃描器的物件結構
	  輸入的程式 (text)
	  目前掃描位置
	  程式的總長度
	  目前掃描到的詞彙

掃描器的主要函數是 ScannerScan()，該函數會取得下一個詞彙後傳回，方法是檢查第一個字元以判斷詞彙類型，然後再進行取得的動作。舉例而言，如果看到第一個字元是數字，那麼 ScannerScan() 將會取得連續的數字，形成一個 NUMBER。

範例 12.19　C0 語言掃描器的主要函數

檔案 Scanner.c	說明
char *ScannerScan(Scanner *scanner) {	掃描下一個詞彙
while (strMember(ch(), SPACE))	忽略空白
next();	
if (scanner->textIdx >= scanner->textLen)	檢查是否超過範圍
return NULL;	
char c = ch();	取得下一個字元
int begin = scanner->textIdx;	記住詞彙開始點
if (c == '\"') { // string = ".."	如果是 "，代表字串開頭，
next(); // skip begin quote "	一直讀到下一個 " 符號
while (ch() != '\"') next();	為止。
next(); // skip end quote "	
} else if (strMember(c, OP)) {	如果是 OP(+-*/<=>!等符號)
while (strMember(ch(), OP)) next();	一直讀到不是 OP 為止
} else if (strMember(c, DIGIT)) {	如果是數字
while (strMember(ch(), DIGIT)) next();	一直讀到不是數字為止
} else if (strMember(c, ALPHA)) {	如果是英文字母
while (strMember(ch(), ALPHA)	一直讀到不是英文字母
|| strMember(ch(), DIGIT))	(或數字)為止 (ex: x1y2z)
next();	
} else	否則，傳回單一字元
next();	
strSubstr(scanner->token, scanner->text,	設定 token 為(begin…textIdx)
begin, scanner->textIdx-begin);	之間的子字串
return scanner->token;	傳回 token 詞彙
}	
Array* tokenize(char *text) {	將程式轉換成一個一個的詞彙
Array *tokens = ArrayNew(10);	
Scanner *scanner = ScannerNew(text);	
char *token = NULL;	
while ((token = ScannerScan(scanner))!= NULL) {	不斷取出下一個詞彙，
ArrayAdd(tokens, newStr(token));	直到程式字串結束為止
printf("token=%s\n", token);	
}	
ScannerFree(scanner);	

return tokens; }	

char *tokenToType(char *token) {
  if (strPartOf(token, KEYWORDS))
  return token;
  else if (token[0] == '\"')
  return STRING;
  else if (strMember(token[0], DIGIT))
  return NUMBER;
  else if (strMember(token[0], ALPHA))
  return ID;
  else
  return token;
}
	判斷並取得 token 的型態
	  如果是關鍵字 if, for, …
	  型態即為該關鍵字
	  如果以符號 " 開頭，則
	  型態為 STRING
	  如果是數字開頭，則
	  型態為 NUMBER
	  如果是英文字母開頭，則
	  型態為 ID
	  否則 (像是 +,-,*,/,>,<,….)
	  型態即為該 token

範例 12.19 中的 tokenize() 函數會不斷的呼叫 ScannerScan() 函數，以將程式切割成一個一個的詞彙，並傳回整個詞彙陣列。另外，tokenToType(token) 這個函數可以用來判斷詞彙的型態，像是 STRING, NUMBER, ID,… 等等。

剖析器

剖析器的資料結構宣告在 Parser.h 檔中，而其程式主要位於 Parser.c 中，剖析器結構 Parser 中包含有詞彙串列 (tokens)、剖析樹 (tree)、剖析過程用的堆疊 (stack)、以及目前掃描到的詞彙位置 (tokenIdx) 等成員，範例 12.20 顯示了 Parser.h 檔中該結構的宣告部分。

範例 12.20 C0 語言剖析器的資料結構

檔案 Parser.h	說明

typedef struct {
  Array *tokens;
  Tree *tree;
  Array* stack;
  int tokenIdx;
} Parser;
 	
Parser *parse(char *text);
	剖析器的物件結構
	  詞彙串列
	  剖析樹 (樹根)
	  剖析過程用的堆疊
	  詞彙指標
	
	剖析器的主程式

Parser *ParserNew();
void ParserParse(Parser *p, char *text);
void ParserFree(Parser *parser);
	剖析器的建構函數
	剖析器的剖析函數
	釋放記憶體

剖析器的主要函數 parse() 會呼叫 ParserParse() 函數將輸入的 C0 語言程式 (text) 轉換成剖析樹 (p->tree)。ParseParse() 函數會利用遞迴下降的方式，呼叫 parseProg() 函數從 PROG 規則開始試圖建構出整棵剖析樹，如果建構完畢後堆疊恰好清空，那麼代表程式建構成功，否則代表程式不完整，導致剖析沒有完成。範例 12.21 顯示了遞迴下降剖析程式的代表性片段，包含 parse(), ParseParse(), parseProg(), parseBaseList(), parseBase(), parseFor() 等函數的程式內容。這些函數被用來剖析對應的語法規則，以建構出該規則的語法樹。

範例 12.21 C0 語言剖析器的程式片段

檔案 Parser.c 中的遞迴下降剖析程式	說明

…
Parser *parse(char *text) {
  Parser *p=ParserNew();
  ParserParse(p, text);
  return p;
}
…
void ParserParse(Parser *p, char *text) {
  printf("======= tokenize =======\n");
  p->tokens = tokenize(text);
  printTokens(p->tokens);
  p->tokenIdx = 0;
  printf("======= parsing ========\n");
  p->tree = parseProg(p);
  if (p->stack->count != 0) {
  printf("parse fail:stack.count=%d",
  p->stack->count);
  error();
  }
}
 	
// PROG = BaseList
Tree *parseProg(Parser *p) {
  push(p, "PROG");
	…
	剖析器的主要函數
	建立剖析器
	開始剖析
	傳回剖析器
	
	剖析物件的主函數
	  首先呼叫掃描器的主函數
	  tokenize() 將程式轉換為
	  詞彙串列
	
	開始剖析
	  PROG= BaseList
	如果剖析完成後堆疊是空的，
	  那就是剖析成功；
	否則提示錯誤訊息
	
	剖析 PROG=BaseList 規則
	
	建立 PROG 的樹根

```
    parseBaseList(p);
    return pop(p, "PROG");
}
// BaseList= (BASE)*
void parseBaseList(Parser *p) {
    push(p, "BaseList");
    while (!isEnd(p) && !isNext(p, "}"))
        parseBase(p);
    pop(p, "BaseList");
}

// BASE = FOR | STMT;
void parseBase(Parser *p) {
    push(p, "BASE");
    if (isNext(p, "for"))
        parseFor(p);
    else {
        parseStmt(p);
        next(p, ";");
    }
    pop(p, "BASE");
}

// FOR = for (STMT; COND; STMT) BLOCK
void parseFor(Parser *p) {
    push(p, "FOR");
    next(p, "for");
    next(p, "(");
    parseStmt(p);
    next(p, ";");
    parseCond(p);
    next(p, ";");
    parseStmt(p);
    next(p, ")");
    parseBlock(p);
    pop(p, "FOR");
}
```

```
 剖析 BaseList，
取出 PROG 的剖析樹

剖析 BaseList=(BASE)* 規則

建立 BaseList 的樹根
 剖析 BASE，直到程式結束
  或碰到 } 為止
取出 BaseList 的剖析樹


剖析 BASE=FOR|STMT 規則

建立 BASE 的樹根
如果下一個詞彙是 for
 根據 FOR 規則進行剖析
否則
 根據 STMT 規則進行剖析
 取得分號 ;

取出 BASE 的剖析樹


剖析 FOR = for (STMT;COND;STMT) BLOCK

 建立 FOR 的樹根
 取得 for
 取得 (
 剖析 STMT
 取得 ;
 剖析 COND
 取得 ;
 剖析 STMT
 取得 )
 剖析 BLOCK
取出 FOR 的剖析樹
```

根據 C0 語言的 EBNF 語法規則，PROG = BaseList; BaseList = (BASE)*; BASE = FOR | STMT; FOR=for (STMT ; COND; STMT) BLOCK….，我們將其中的非終端節點 X 轉換成 parseX() 這樣的遞迴下降函數，並將終端節點轉換成 next(pTypes)，如此就完成了將語法轉換為程式的動作。

舉例而言，FOR=for (STMT ; COND ; STMT) BLOCK 規則被轉換成 parseFor() 函數中的 next("for"); next("("); parseStmt(); next(";"); parseCond(); next(";"); parseStmt(); next(")"); parseBlock(); 等敘述，然後函數的前後加上 push("FOR") 與 pop("FOR")，即可將規則轉換為程式。

當有重複語法，像是 BaseList = (BASE)* 這種語法時，就必須使用 while 迴圈剖析 BASE 數次，直到後續沒有 BASE 為止。對於選擇語法，像是 BASE = FOR | STMT，則必須用 if 敘述先判斷到底是哪一種情況，再根據判斷決定呼叫 parseFor() 或 parseStmt() 函數。

讀者可能會感到疑惑，剖析樹的建構動作到底在哪裡執行呢？答案在 next(), push(), pop() 函數中，究竟這些函數做了甚麼動作呢？

函數 push(p, pType) 會根據 pType 建立一個樹狀節點，然後堆入堆疊中，函數 pop(p, pType) 則會取出該節點，然後掛到其上一層的節點之下，以形成剖析樹。類似的，next(p, pTypes) 函數也會建構出一個樹狀節點並掛到上一層節點中，以便形成剖析樹。

範例 12.22 C0 語言剖析器的程式片段

檔案 Parser.c	說明

…
char *next(Parser *p, char *pTypes) {
  char *token = nextToken(p);
  if (isNext(p, pTypes)) {
  char *type = tokenToType(token);
  Tree *child = TreeNew(type, token);
  Tree *parentTree = ArrayPeek(p->stack);
  TreeAddChild(parentTree, child);
  printf("%s idx=%d, token=%s, type=%s\n",
  level(p), p->tokenIdx, token,type);
  p->tokenIdx++;
  return token;
  } else {
	…
	檢查下一個詞彙的型態
	  取得下一個詞彙
	  如果是 pTypes 型態之一
	  取得型態
	  建立詞彙節點(token,type)
	  取得父節點，
	  加入父節點成為子樹
	  印出詞彙以便觀察
	
	  前進到下一個節點
	  傳回該詞彙
	  否則(下一個節點型態錯誤)

```
    printf("next():%s is not type(%s)\n",
            token, pTypes);
    error();
    p->tokenIdx++;
    return NULL;
  }
}

Tree* push(Parser *p, char* pType) {
    printf("%s+%s\n", level(p), pType);
    Tree* tree = TreeNew(pType, "");
    ArrayPush(p->stack, tree);
    return tree;
}

Tree* pop(Parser *p, char* pType) {
  Tree *tree = ArrayPop(p->stack);
  printf("%s-%s\n", level(p), tree->type);
  if (strcmp(tree->type, pType)!=0) {
    printf("pop(%s):should be %s\n", tree->type,
              pType);
    error();
  }
  if (p->stack->count > 0) {
    Tree *parentTree = ArrayPeek(p->stack);
    TreeAddChild(parentTree, tree);
  }
  return tree;
}
…
```

```
    印出錯誤訊息


    前進到下一個節點


建立 pType 型態的子樹，推
入堆疊中


取出 pType 型態的子樹
  取得堆疊最上層的子樹
  印出以便觀察
  如果型態不符合
    印出錯誤訊息


  如果堆疊不是空的
    取出上一層剖析樹
    將建構完成的剖析樹加入
      上一層節點中，成為子樹


…
```

圖 12.1 顯示了遞迴下降剖析器的執行過程，當 BASE 節點的 parseBase() 函數被執行時，會先用 push(“Base”) 建立 BASE 節點，然後呼叫 parseFor() 函數剖析 FOR 語句。parseFor() 函數中又會利用 push("FOR") 建立 FOR 節點，然後再繼續呼叫 next("for"); next("("); parseStmt(); … 等函數建立 FOR 的語法樹，並且不斷的將建立完成的 for, (, STMT 等節點掛到 FOR 節點之下。

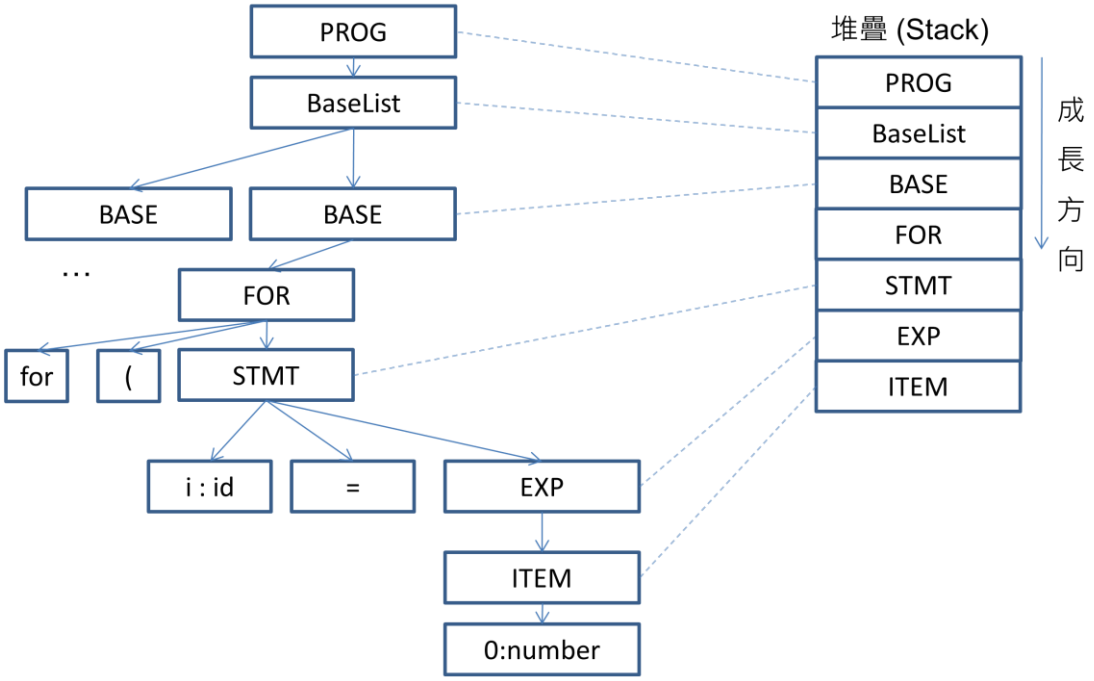

圖 12.1 遞迴下降剖析器的執行過程

當 parseFor() 函數執行完成後，FOR 節點的語法樹也就建構完成了，於是最後利用 pop("FOR") 函數將 FOR 的語法樹掛到上一層的 BASE 節點中。如此不斷的透過『遞迴、下降、掛載、上升』等動作，就能建構出整棵語法樹了。

12.5 編譯器實作

編譯器 C0C 的輸入為一個符合 C0 語法的程式檔，而輸出則是一個 CPU0 的組合語言檔案，在本章的實作中，我們利用 make 專案的方式建構出 c0c.exe 這個執行檔，其執行方法為 c0c <c0File> <asmFile>。

舉例而言，當我們用 c0c test.c0 test.asm0 編譯範例 12.23(a) 中的 test.c0 程式時，c0c 會先將 test.c0 轉換成 (b) 中的虛擬碼 (pcode) 之後，然後才產生 (c) 中的 test.asm0 組合語言檔，我們也可以進一步利用 as0 組譯器將 test.asm0 組譯為 test.obj0，然後再利用 vm0 去執行該目的檔，於是本章的實作就形成了一組簡單的小型開發工具，包含編譯器 (c0c)、組譯器 (as0) 與虛擬機器 (vm0) 等程式。

範例 12.23 透過 c0c 編譯器將 test.c0 轉回組合語言 test.asm0

(a) ch12/test.c0	(b) 虛擬碼 pcode	(c) 組合語言 ch12/test.asm0

sum = 0;
for (i=0; i<=10; i++)
{
  sum = sum + i;
}
return sum;
	= 0 sum
	= 0 i
	FOR0:
	CMP i 10
	J > _FOR0
	+ sum i T0
	= T0 sum
	+ i 1 i
	J FOR0
	_FOR0:
	RET sum
		LDI R1 0
		ST R1 sum
		LDI R1 0
		ST R1 i
		FOR0:
		LD R1 i
		LDI R2 10
		CMP R1 R2
		JGT _FOR0
		LD R1 sum
		LD R2 i
		ADD R3 R1 R2
		ST R3 T0
		LD R1 T0
		ST R1 sum
		LD R1 i
		LDI R2 1
		ADD R3 R1 R2
		ST R3 i

		JMP FOR0 _FOR0: LD R1 sum RET sum: RESW 1 i: RESW 1 T0: RESW 1

編譯器 C0C 的主程式為 compiler.c 中的 compile() 函數，該函數會呼叫剖析器 (Parser) 將 C0 程式轉換為語法樹，然後再呼叫程式碼產生器 (Generator) 的主程式 generate() 將語法樹轉換為目的碼。

範例 12.24 C0 語言編譯器的主要函數

檔案 compiler.c 中的 compile() 函數	

void compile(char *cFile, char *asmFile) {
printf("compile file:%s\n", cFile, asmFile);
char *cText = newFileStr(cFile);
Parser *parser = parse(cText);
generate(parser->tree, asmFile);
ParserFree(parser);
freeMemory(cText);
}
	編譯器主程式
	
	讀取檔案到 cText 字串中。
	剖析程式 (cText) 轉為語法樹
	程式碼產生
	釋放記憶體

程式碼產生器的資料結構定義於 Generator.h 檔案的 Generator 結構中，包含符號表 (symTable)、語法樹 (tree)、組合語言檔 (asmFile)、for 迴圈的數量記錄 (forCount)、以及臨時變數的數量 (varCount) 等。

範例 12.25 C0 語言程式碼產生器的資料結構與主要函數

檔案 Generator.h	說明
typedef struct { HashTable *symTable;	程式碼產生器物件 符號表

Tree *tree; FILE *asmFile; int forCount, varCount; } Generator;	剖析樹 輸出的 CPU0 組合語言檔 For 迴圈與臨時變數的數量
void generate(Tree *tree, char *asmFile);	程式碼產生器的主函數

Generator *GenNew();
void GenFree(…);
Tree* GenCode(...);
void GenData(...);
void GenPcode(...);
void GenPcodeToAsm(...);
void GenAsmCode(...);
void GenTempVar(...);
void negateOp(…);
	Generator 的建構函數
	Generator 的解構函數
	產生組合語言程式碼
	產生資料宣告
	輸出虛擬碼 pcode
	將虛擬碼轉為組合語言
	輸出組合語言指令
	取得下一個臨時變數名稱
	取比較運算的互補運算,ex: < 變 >=

Generator 的主程式是 generate() 函數，該函數會將語法樹轉換為組合語言檔案。程式轉換的動作主要由 GenCode(g, tree) 函數完成，最後再利用 GenData(g) 將符號表中的所有符號轉換為資料宣告的 RESW 敘述，如此就完成了程式產生器的工作，範例 12.26 顯示了 generate() 函數的原始碼。

範例 12.26 C0 程式碼產生器的主程式 - generate()

檔案 Generator.c 中的 generate() 函數	說明

void generate(Tree *tree, char *asmFile) {
char nullVar[100]="";
Generator *g = GenNew();
g->asmFile = fopen(asmFile, "w");
printf("=====PCODE=====\n");
GenCode(g, tree, nullVar);
GenData(g);
fclose(g->asmFile);
GenFree(g);
char *asmText = newFileStr(asmFile);
printf("=====AsmFile:%s======\n", asmFile);
printf("%s\n", asmText);
freeMemory(asmText);
}
	將剖析樹 tree 轉為組合語言檔 asmFile
	
	開啟組合語言檔以便輸出
	
	產生程式碼
	產生資料宣告
	關閉組合語言檔
	釋放記憶體
	讀入組合語言檔並印出
	
	釋放記憶體

範例 12.27 顯示了 GenCode() 函數的原始碼，GenCode() 函數是利用遞迴的方式訪問每一個節點，然後根據節點類型取出對應的子節點，利用這些子節點傳回的內容進行轉換的動作。舉例而言，當 GenCode() 函數訪問到 COND 節點時，我們會根據 COND = EXP ('=='|'!='|'<='|'>='|'<'|'>') EXP 這個語法，判斷應該有 exp, (…), exp 等三個子節點，然後利用 GenPcode(g, "", "CMP", expVar1, expVar2, nullVar) 這行程式將語法樹轉換成 pcode 虛擬碼，接著在 GenPcode() 中又呼叫 GenPcodeToAsm() 函數，以產生對應的 CPU0 組合語言。

在 C0 語言的語法中，FOR 語法是程式產生器中較難以處理的一段，因為程式產生器必須將該語法轉換成條件式跳躍的組合語言指令，這個對應較不直覺，是高階語言與組合語言的主要差異所在。我們必須在 FOR 迴圈開始與結束時輸出起始標記與結束標記，然後在 COND 條件比較時根據比較結果進行條件式的跳躍，這是為何其程式碼較長的原因。

範例 12.27 C0 語言程式碼產生器的 GenCode() 函數

檔案 Generator.c 中的 GenCode() 遞迴轉換函數	說明
Tree* GenCode(Generator *g, Tree *node, char *rzVar) {	遞迴產生節點 node 的程式碼
strcpy(nullVar, "");	
strcpy(rzVar, "");	
if (node == NULL) return NULL;	遞迴終止條件。
if (strEqual(node->type, "FOR")) {	
// FOR ::= for (STMT;COND;STMT) BLOCK	處理 FOR 節點
char forBeginLabel[100], forEndLabel[100],	
condOp[100];	
Tree *stmt1 = node->childs->item[2],	取得子節點
*cond = node->childs->item[4],	
*stmt2 = node->childs->item[6],	
*block = node->childs->item[8];	
GenCode(g, stmt1, nullVar);	遞迴產生<STMT>
int tempForCount = g->forCount++;	設定 FOR 迴圈的
sprintf(forBeginLabel, "FOR%d", tempForCount);	進入標記
sprintf(forEndLabel, "_FOR%d", tempForCount);	離開標記
GenPcode(g, forBeginLabel, "", "", "", "");	中間碼：例如 FOR0:
GenCode(g, cond, condOp);	遞迴產生 COND
char negOp[100];	
negateOp(condOp, negOp);	互補運算 negOp
GenPcode(g, "", "J", negOp, "", forEndLabel);	中間碼：例如 J > _FOR0
GenCode(g, block, nullVar);	遞迴產生 BLOCK
GenCode(g, stmt2, nullVar);	遞迴產生 STMT
GenPcode(g, "", "J", "", "", forBeginLabel);	中間碼：例如 J FOR0
GenPcode(g, forEndLabel, "", "", "", "");	中間碼：例如 _FOR0
return NULL;	
} else if (strEqual(node->type, "STMT")) {	處理 STMT 節點
// STMT:= return | id '=' EXP | id ('++' | '-- ')	
Tree *c1 = node->childs->item[0];	取得子節點
if (strEqual(c1->type, "return")) {	處理 return 指令
Tree *id = node->childs->item[1];	
GenPcode(g, "", "RET", "", "", id->value);	中間碼： 例如 RET sum
} else {	
Tree *id = node->childs->item[0];	取得子節點
Tree *op = node->childs->item[1];	

```
    if (strEqual(op->type, "=")) {
      Tree *exp = node->childs->item[2];
      char expVar[100];
      GenCode(g, exp, expVar);
      GenPcode(g, "", "=", expVar, "", id->value);
      HashTablePut(g->symTable, id->value, id->value);
      strcpy(rzVar, expVar);
    } else {
      char addsub[100];
      if (strEqual(op->value, "++"))
        strcpy(addsub, "+");
      else
        strcpy(addsub, "-");
      GenPcode(g, "", addsub, id->value, "1", id->value);
      strcpy(rzVar, id->value);
    }
  }
} else if (strEqual(node->type, "COND")) {
  // COND = EXP ('=='|'!='|'<='|'>='|'<'|'>') EXP
  Tree* op = node->childs->item[1];
  char expVar1[100], expVar2[100];
  GenCode(g, node->childs->item[0], expVar1);
  GenCode(g, node->childs->item[2], expVar2);
  GenPcode(g, "", "CMP", expVar1, expVar2, nullVar);
  strcpy(rzVar, op->value);
} else if (strPartOf(node->type, "|EXP|")) {
  } else if (strPartOf(node->type, "|EXP|")) {
    // 處理運算式 EXP = ITEM ([+-*/] ITEM)*
    Tree *item1 = node->childs->item[0];
    char var1[100], var2[100], tempVar[100];
    GenCode(g, item1, var1);
    if (node->childs->count > 1) {
      Tree* op = node->childs->item[1];
      Tree* item2 = node->childs->item[2];
      GenCode(g, item2, var2);
      GenTempVar(g, tempVar);
      GenPcode(g, "", op->value, var1, var2, tempVar);
      strcpy(var1, tempVar);
```

```
處理 id = EXP
 取得子節點

 遞迴產生 EXP
 中間碼：例如 = 0 sum
 將 id 加入到符號表中
 傳回 EXP 的變數，例如 T0
處理 id++ 或 id--

 如果是 id++
  設定運算為 + 法
 否則
  設定運算為 - 法
 中間碼：例如 ADD i, 1, i
 傳回 id，例如 i


處理 COND 節點

取得子節點

 遞迴產生 EXP
 遞迴產生 EXP
 中間碼：例如 CMP i,10
 傳回比較運算，例如 >
處理 EXP

取得子節點 ITEM

遞迴產生 ITEM

連續取得 (op ITEM}

遞迴產生 TERM
取得臨時變數，例如 T0
中間碼：例如 + sum i T0
傳回臨時變數，例如 T0
```

}
strcpy(rzVar, var1);
} else if (strPartOf(node->type, "|number|id|")) {
strcpy(rzVar, node->value);
} else if (node->childs != NULL) {
int i;
for (i=0; i<node->childs->count; i++)
GenCode(g, node->childs->item[i], nullVar);
}
return NULL;
}
	傳回臨時變數，例如 T0
	
	處理 id|number
	遇到變數或常數，直接傳回 value
	直接傳回 id 或 number
	其他情況
	
	遞迴處理所有子節點

在將 pcode 轉換為組合語言的過程中，由於 pcode 虛擬碼中並不需要考慮暫存器的配置方式，但在 CPU0 的組合語言中只有 R0-R15 等暫存器可以使用，因此我們必須在 GenPcodeToAsm() 函數中處理暫存器的配置問題。GenPcodeToAsm() 的產生方法很簡單，永遠都是按照暫存器 R1,R2, R3, … 的方式配置，每次都會將 pcode 中的運算元重新載入暫存器中，然後才進行運算，最後再將目標暫存器存回記憶體中。

舉例而言，在執行 GenPcodeToAsm(g, "", "CMP", "i", "10", "") 這個呼叫時，會將一個 CMP i, 10 的指令，拆解成 LD R1, i; LDI R2, 10; CMP R1, R2 等三個組合語言指令，因為在 CPU0 當中，不能直接用 CMP 指令比較記憶體變數 i 與常數 10，因此必須轉換成三個指令，這也正是 GenPcodeToAsm() 函數中處理 CMP 指令的 GenCode() 與 GenPCode() 等三行程式碼所作的事情。

範例 12.28 C0 語言程式碼產生器的 pcode 與組合語言之產生片段

檔案 Generaor.c 中的 GenPcode(), GenPcodeToAsm(),GenAsm()函數	說明

void GenPcode(Generator *g, char* label,
  char* op, char* p1, char* p2, char* pTo) {
  char labelTemp[100];
  if (strlen(label)>0)
  sprintf(labelTemp, "%s:", label);
  else
  strcpy(labelTemp, "");
  printf("%-8s %-4s %-4s %-4s %-4s\n",
  labelTemp, op, p1, p2, pTo);
  GenPcodeToAsm(g, labelTemp, op, p1, p2, pTo);
}
	輸出 pcode 後再轉為組合語言
	
	印出 pcode
	
	將 pcode 轉為組合語言

void GenPcodeToAsm(Generator *g, char* label,
  char* op, char* p1, char* p2, char* pTo) {
  if (strlen(label)>0)
  GenAsmCode(g, label, "", "", "", "");
  if (strEqual(op, "=")) { // pTo = p1
  GenAsmCode(g, "", "LD", "R1", p1, "");
  GenAsmCode(g, "", "ST", "R1", pTo, "");
  } else if (strPartOf(op, "|+|-|*|/|")) {
  char asmOp[100];
  if (strEqual(op, "+")) strcpy(asmOp, "ADD");
  else if (strEqual(op, "-")) strcpy(asmOp, "SUB");
  else if (strEqual(op, "*")) strcpy(asmOp, "MUL");
  else if (strEqual(op, "/")) strcpy(asmOp, "DIV");
  GenAsmCode(g, "", "LD", "R1", p1, "");
  GenAsmCode(g, "", "LD", "R2", p2, "");
  GenAsmCode(g, "", asmOp,"R3", "R2", "R1");
  GenAsmCode(g, "", "ST", "R3", pTo, "");
  } else if (strEqual(op, "CMP")) { // CMP p1, p2
  GenAsmCode(g, "", "LD", "R1", p1, "");
  GenAsmCode(g, "", "LD", "R2", p2, "");
  GenAsmCode(g, "", "CMP", "R1", "R2", "");
  } else if (strEqual(op, "J")) { // J op label
	將 pcode 轉為組合語言的函數
	
	如果有標記
	  輸出標記
	處理等號 (= 0 sum)
	例如 LDI R, 0
	  ST R1, sum
	處理運算 (+ sum i sum)
	
	根據 op 設定運算指令
	
	例如 LD R1, sum
	  LD R2, i
	  ADD R3, R1, R2
	  ST R3, sum
	處理 CMP (cmp i 10)
	例如 LD R1, i
	  LDI R2, 10
	  CMP R1, R2
	處理 J (J > _FOR)

```
        char asmOp[100];                                        根據 op 設定跳躍指令
        if (strEqual(p1, "=")) strcpy(asmOp, "JEQ");
        else if (strEqual(p1, "!=")) strcpy(asmOp, "JNE");
        else if (strEqual(p1, "<")) strcpy(asmOp, "JLT");
        else if (strEqual(p1, ">")) strcpy(asmOp, "JGT");
        else if (strEqual(p1, "<=")) strcpy(asmOp, "JLE");
        else if (strEqual(p1, ">=")) strcpy(asmOp, "JGE");
        else strcpy(asmOp, "JMP");
        GenAsmCode(g, "", asmOp, pTo, "", "");                  例如 JGT _FOR0
    } else if (strEqual(op, "RET")) {                           例如 RET sum
        GenAsmCode(g, "", "LD", "R1", pTo, "");                 轉成 LD R1, sum
        GenAsmCode(g, "", "RET", "", "", "");                           RET
    }
}

void GenAsmCode(Generator *g, char* label,                      輸出組合語言指令
    char* op, char* p1, char* p2, char* pTo) {
    char *realOp = op;
    if (strEqual(op, "LD"))                                     如果指令是 LD，而且
        if (isdigit(p2[0]))                                       p2 為常數
            realOp = "LDI";                                         改用 LDI 指令
    fprintf(g->asmFile, "%-8s %-4s %-4s %-4s %-4s\n",           輸出組合語言指令
        label, realOp, p1, p2, pTo);
}
…
```

完成了程式碼產生器之後，編譯器的實作也就完成了，接著，讓我們來看看本章中所有的程式是如何進行建置與測試的。

12.6整合測試

我們利用一個主程式，以條件編譯的方式，分別編譯出 test, c0c, as0, vm0 等四個執行檔，其方法是利用 C 語言的巨集編譯指令 #if … #elif…#endif，如範例 12.29 所示，這是 C 語言中一個很常見的技巧。

範例 12.29 本章所有程式的主程式

檔案 ch12/main.c	說明

#include "Assembler.h"	引用組譯器檔頭
#include "Compiler.h"	引用編譯器檔頭
#define TEST 1	編譯目標 1: test
#define C0C 2	編譯目標 2: c0c
#define AS0 3	編譯目標 3: as0
#define VM0 4	編譯目標 4: vm0
void argError(char *msg) {	處理參數錯誤的情況
printf("%s\n", msg);	
exit(1);	
}	
int main(int argc, char *argv[]) {	主程式開始
char cFile0[]="test.c0", *cFile=cFile0;	預設程式檔為 test.c0
char asmFile0[]="test.asm0", *asmFile=asmFile0;	預設組合語言為 test.asm0
char objFile0[]="test.obj0", *objFile=objFile0;	預設目的檔為 test.obj0
#if TARGET==TEST	如果編譯目標為 TEST
ArrayTest();	測試陣列物件
HashTableTest();	測試雜湊表物件
OpTableTest();	測試指令表物件
compile(cFile, asmFile);	測試編譯器
assemble(asmFile, objFile);	測試組譯器
runObjFile(objFile);	測試虛擬機器
checkMemory();	檢查記憶體使用狀況
#elif TARGET==C0C	如果編譯目標為 C0C
if (argc == 3) {	如果有 3 個參數
cFile=argv[1]; asmFile=argv[2];	設定參數
} else	否則
argError("c0c <c0File> <asmFile>");	提示程式執行方法
compile(cFile, asmFile);	開始編譯
#elif TARGET==AS0	如果編譯目標為 AS0
if (argc == 3) {	如果有 3 個參數
asmFile=argv[1]; objFile=argv[2];	設定參數
} else	否則
argError("as0 <asmFile> <objFile>");	提示程式執行方法
assemble(asmFile, objFile);	開始組譯
#elif TARGET==VM0	如果編譯目標為 VM0

if (argc == 2)
  objFile=argv[1];
else
  argError("vm0 <objFile>");
runObjFile(objFile);
#endif
system("pause");
return 0;
}
	如果有 2 個參數
	  設定參數
	否則
	  提示程式執行方法
	開始執行 (虛擬機)
	
	暫停 (給 Dev C++ 使用的)

然後，我們就可以撰寫一個專案建置的 makefile 檔案，以方便建置工作的進行，makefile 檔案的內容如範例 12.30 所示。

範例 12.30 本章所有程式的專案建置檔 makefile

檔案 ch12/makefile	說明

CC = gcc.exe -D__DEBUG__
OBJ = Parser.o Tree.o Lib.o Scanner.o Array.o Compiler.o
HashTable.o Generator.o Assembler.o Cpu0.o OpTable.o
LINKOBJ = $(OBJ)
LIBS =
INCS =
BIN = test.exe c0c.exe as0.exe vm0.exe
CFLAGS = $(INCS) -g3
RM = rm -f
 	
.PHONY: all clean
 	
all: $(OBJ) test c0c as0 vm0
 	
test: $(OBJ)
  $(CC) main.c $(LINKOBJ) -DTARGET=TEST -o test $(LIBS)
 	
c0c: $(OBJ)
  $(CC) main.c $(LINKOBJ) -DTARGET=C0C -o c0c $(LIBS)
 	
as0: $(OBJ)
  $(CC) main.c $(LINKOBJ) -DTARGET=AS0 -o as0 $(LIBS)
	編譯器為 gcc
	目的檔列表
	
	連結用的目的檔
	無額外函式庫
	無額外連結目錄
	執行檔列表
	編譯用的旗標
	移除指令
	
	預設的 make 動作
	
	全部編譯
	
	測試程式 test.exe
	
	c0c 編譯器
	
	as0 組譯器

vm0: $(OBJ) $(CC) main.c $(LINKOBJ) -DTARGET=VM0 -o vm0 $(LIBS)	vm0 虛擬機
clean: ${RM} $(OBJ) $(BIN)	清除上一次 make 所產生的檔案
Parser.o: Parser.c $(CC) -c Parser.c -o Parser.o $(CFLAGS)	剖析器
Tree.o: Tree.c $(CC) -c Tree.c -o Tree.o $(CFLAGS)	語法樹
Lib.o: Lib.c $(CC) -c Lib.c -o Lib.o $(CFLAGS)	基礎函式庫
Scanner.o: Scanner.c $(CC) -c Scanner.c -o Scanner.o $(CFLAGS)	掃描器
Array.o: Array.c $(CC) -c Array.c -o Array.o $(CFLAGS)	動態陣列
Compiler.o: Compiler.c $(CC) -c Compiler.c -o Compiler.o $(CFLAGS)	編譯器
HashTable.o: HashTable.c $(CC) -c HashTable.c -o HashTable.o $(CFLAGS)	雜湊表
Generator.o: Generator.c $(CC) -c Generator.c -o Generator.o $(CFLAGS)	程式產生器
Assembler.o: Assembler.c $(CC) -c Assembler.c -o Assembler.o $(CFLAGS)	組譯器
Cpu0.o: Cpu0.c $(CC) -c Cpu0.c -o Cpu0.o $(CFLAGS)	虛擬機
OpTable.o: OpTable.c $(CC) -c OpTable.c -o OpTable.o $(CFLAGS)	指令表

您可以利用 GNU 工具中的 make 指令，建置 ch12 資料夾中的 Makefile 檔案，只要在將 make 的環境設定好後，在 ch12 的資料夾中打 make 指令即可。另外，如果您習慣使用 Dev C++ 開發環境，我們已經如圖 12.2 所示，將 Dev C++ 功能表選項 Project/Project Options/Makefile 當中的 Use custom Makefile 設定為 ch12/Makefile 了。因此當您在 Dev C++ 當中建置專案時，就會呼叫我們自己寫的 Makefile，於是在建置時會按照範例 12.30 的指示，產生測試程式 (test.exe), 編譯器 (c0c.exe), 組譯器 (as0.exe),與虛擬機 (vm0.exe) 等四個執行檔，因此您也可以用 Dev C++ 直接打開 ch12 中的 Ch12.dev 以建置整個專案。

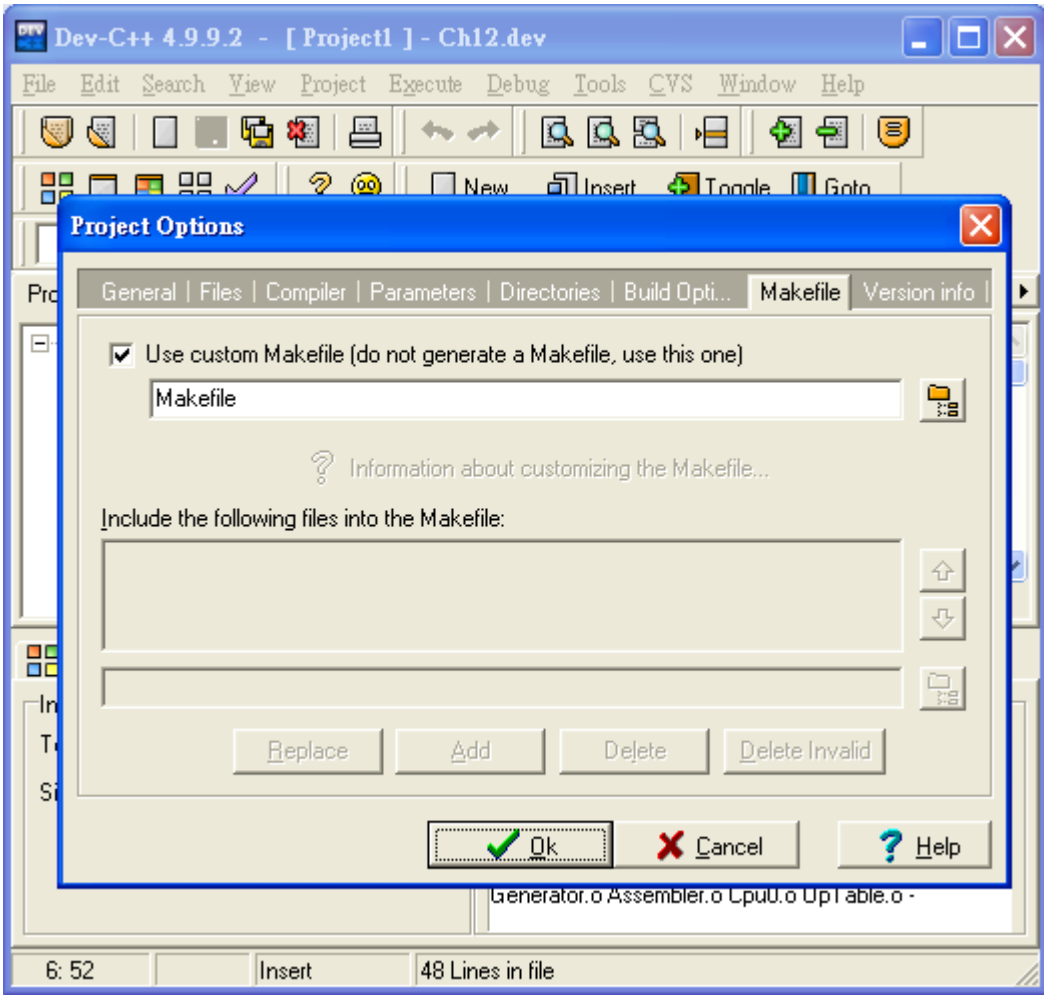

圖 12.2 在 Dev C++ 當中設定使用自己寫的 Makefile

另外，為了讓 Dev C++ 能在按下 Execute/Run 功能時執行我們的測試程式 test.exe，我們設定了 Prject/Project Options/Build Options 當中的 override output filename 為 test.exe，如圖 12.3 所示。當您按下 Run 選項時，就會執行 test.exe 測試程式，將 test.c0 程式編譯為 test.asm0 組合語言，然後再將 test.asm0 程式組譯為目的檔 test.obj0，最後利用虛擬機執行 test.obj0 而顯示出執行結果。

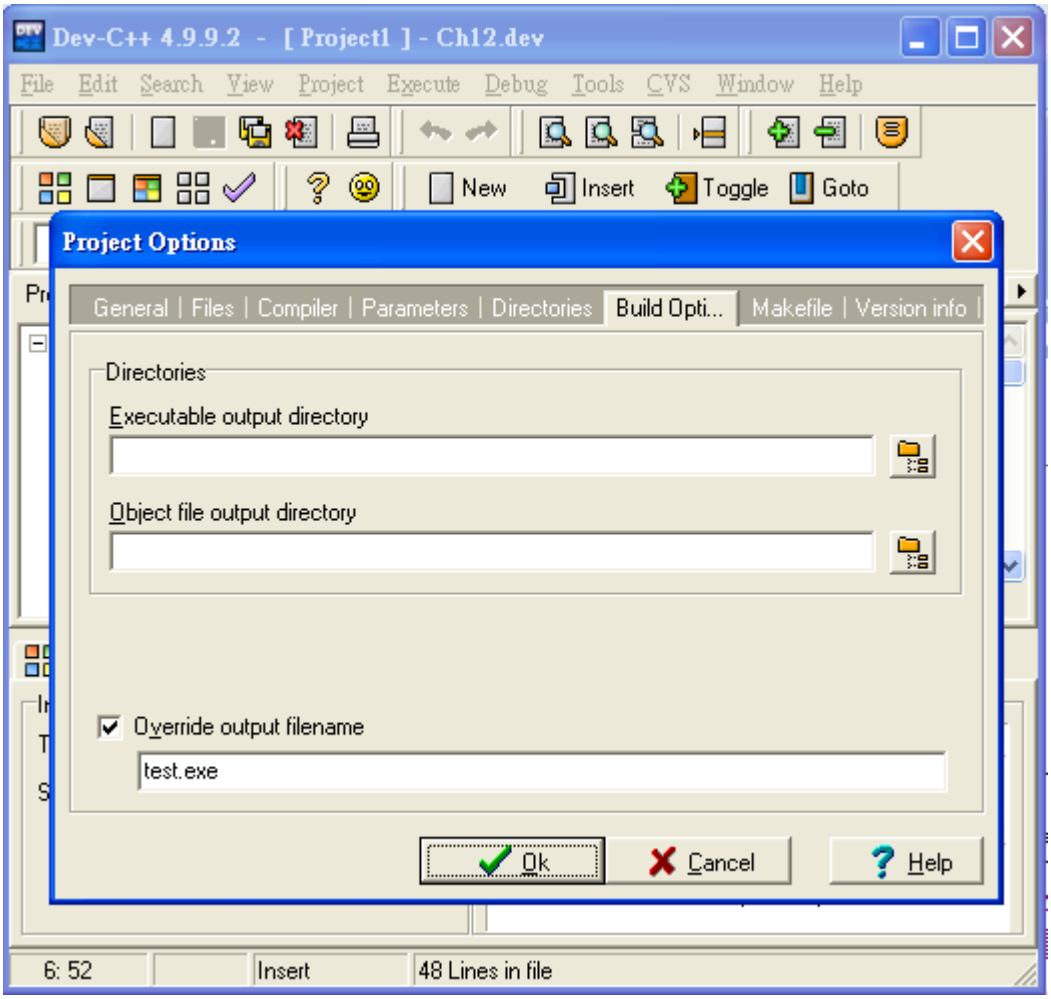

圖 12.3 在 Dev C++ 中設定執行的輸出檔案為 test.exe

範例 12.31 顯示了我們用 make 指令建置整個專案後，並利用 c0c, as0, vm0 等程式逐步執行編譯、組譯與執行的過程片段。

範例 12.31 本章所有程式的建置與執行過程

本章範例程式的建置執行過程	說明
C:\ch12 C:\ch12>make gcc.exe -D__DEBUG__ -c Parser.c -o Parser.o -g3 … C:\ch12>c0c test.c0 test.asm0 … … ======= parsing ======== +PROG	切換到 ch12/ 開始建置專案 專案編譯過程 … 將 test.c0 檔案編譯為 test.asm0 組合語言 印出剖析樹 …

```
 +BaseList
  +BASE
   +STMT
     idx=0, token=sum, type=id
     idx=1, token==, type==
    +EXP
      idx=2, token=0, type=number
    -EXP
   -STMT
…
=====PCODE=====
         =    0          sum
         =    0          i
FOR0:
         CMP  i    10
         J    >          _FOR0
         +    sum  i     T0
         =    T0         sum
         +    i    1     i
         J               FOR0
_FOR0:
         RET             sum
=====AsmFile:test.asm0=====
         LDI  R1   0
         ST   R1   sum
         LDI  R1   0
         ST   R1   i
FOR0:
         LD   R1   i
         LDI  R2   10
         CMP  R1   R2
         JGT  _FOR0
         LD   R1   sum
         LD   R2   i
         ADD  R3   R1   R2
         ST   R3   T0
         LD   R1   T0
         ST   R1   sum
```

印出虛擬碼

…

印出組合語言程式

…

```
         LD   R1   i
         LDI  R2   1
         ADD  R3   R1   R2
         ST   R3   i
         JMP  FOR0
_FOR0:
         LD   R1   sum
         RET
sum:     RESW 1
i:       RESW 1
T0:      RESW 1

C:\ch12>as0 test.asm0 test.obj0                    組譯 test.asm0，輸出 test.obj0
Assembler:asmFile=test.asm0 objFile=test.obj0      開始組譯
==============Assemble=============               第一階段組譯 (計算位址)
0000          LDI  R1   0          L  8 (NULL)
0004          ST   R1   SUM        L  1 (NULL)
…
==============SYMBOL TABLE=========               印出符號表
005C T0:      RESW 1               D F0 (NULL)
0010 FOR0:                           FF (NULL)
0054 SUM:     RESW 1               D F0 (NULL)
0058 I:       RESW 1               D F0 (NULL)
004C _FOR0:                          FF (NULL)
============PASS2==============                  第二階段組譯 (產生程式碼)
0000          LDI  R1   0          L  8 08100000
0004          ST   R1   SUM        L  1 011F004C
0008          LDI  R1   0          L  8 08100000
000C          ST   R1   I          L  1 011F0048
0010 FOR0:                           FF
0010          LD   R1   I          L  0 001F0044
0014          LDI  R2   10         L  8 0820000A
0018          CMP  R1   R2         A 10 10120000
001C          JGT  _FOR0           J 23 2300002C
0020          LD   R1   SUM        L  0 001F0030
0024          LD   R2   I          L  0 002F0030
0028          ADD  R3   R1   R2    A 13 13312000
002C          ST   R3   T0         L  1 013F002C
```

0030 LD R1 T0 L 0 001F0028	
0034 ST R1 SUM L 1 011F001C	
0038 LD R1 I L 0 001F001C	
003C LDI R2 1 L 8 08200001	
0040 ADD R3 R1 R2 A 13 13312000	
0044 ST R3 I L 1 013F0010	
0048 JMP FOR0 J 26 26FFFFC4	
004C _FOR0: FF	
004C LD R1 SUM L 0 001F0004	
0050 RET J 2C 2C000000	
0054 SUM: RESW 1 D F0 00000000	
0058 I: RESW 1 D F0 00000000	
005C T0: RESW 1 D F0 00000000	
==========Save to ObjFile:test.obj0==========	將目的碼存入 test.obj0
08100000011F004C08100000011F00480…	
C:\ch12>vm0 test.obj0	以 vm0 虛擬機器執行該目的檔
===VM0:run test.obj0 on CPU0===	…
PC=00000004 IR=08100000 R[01]=0X00000000=0	執行中…
PC=00000008 IR=011F004C R[01]=0X00000000=0	…
…	
PC=00000044 IR=13321000 R[03]=0X00000001=1	
PC=00000048 IR=013F0010 R[03]=0X00000001=1	JMP FOR 跳回 0x10
PC=00000010 IR=26FFFFC4 R[15]=0X00000010=16	…
PC=00000014 IR=001F0044 R[01]=0X00000001=1	
…	
PC=0000001C IR=10120000 R[01]=0X0000000B=11	JGT _FOR 跳到 0x4C
PC=0000004C IR=2300002C R[00]=0X00000000=0	LD R1, sum 將 sum (55)
PC=00000050 IR=001F0004 R[01]=0X00000037=55	放到暫存器 R1，
PC=00000054 IR=2C000000 R[00]=0X00000000=0	所以 R1=55
===CPU0 dump registers===	印出所有暫存器
IR =0x2c000000=738197504	
R[00]=0x00000000=0	
R[01]=0x00000037=55	sum = R1 = 55
R[02]=0x0000000a=10	
R[03]=0x0000000b=11	
R[04]=0x00000000=0	

R[05]=0x00000000=0 R[06]=0x00000000=0 R[07]=0x00000000=0 R[08]=0x00000000=0 R[09]=0x00000000=0 R[10]=0x00000000=0 R[11]=0x00000000=0 R[12]=0x00000000=0 R[13]=0x00000000=0 R[14]=0xffffffff=-1 R[15]=0x00000054=84	

讀者可以看到指令 vm0 test.obj0 的執行結果，由於 return sum 指令所產生的組合語言程式碼 LD R1, sum; RET 這兩行指令，使得暫存器 R1 的值變為 55，而這正是 test.c0 中所計算出的 1+2+…+10 的結果，因此可以驗證該程式的編譯、組譯與執行過程是正確的。

我們在本章中已經實作了簡單的系統程式，包含編譯器 c0c、組譯器 as0 與虛擬機 vm0。有興趣的讀者可以進一步參考光碟中 ch12 資料夾下的程式，並且實際編譯執行看看，應該能更深入的理解這些系統軟體的原理與實作方法。

在本書中，我們已經盡可能的採取較寬廣的視野，企圖兼顧理論與實務兩個面向，以 CPU0 為主要的理論基礎，並以 GNU 為主要的實作工具，讓讀者從這兩方面同時並進，以便理解系統軟體的設計原理，並且能加以實作。然而，畢竟系統程式是一個涵蓋面甚廣的領域，本書無法深入的對每個主題進行詳細的討論，有興趣的讀者可以進一步學習『計算機組織』、『嵌入式系統』、『編譯器』與『作業系統』等課程，這些課程中會更深入的介紹進一步的相關主題。

習題

12.1 請撰寫一個 C0 語言的程式 fib.c0，可以利用 for 迴圈的方式算出費氏序列中的 f(10) 的值，費氏序列的規則為 f(n) = f(n-1)+f(n-2)，而且 f(0) = 1, f(1)=1。

12.2 接續前一題，請利用 c0c 編譯器，將 fib.c0 編譯為組合語言 fib.asm0。

12.3 接續前一題，請利用 as0 組譯器，將 fib.asm0 組譯為目的檔 fib.obj0。

12.4 接續前一題，請利用 vm0 虛擬機，執行 fib.obj0，並檢查看看 f(10) 的結果是否正確。

12.5 請為 C0 語言加上 if 條件的規則為 IF = 'if' '(' COND ')' BASE ('else' 'if' '('

COND ')' BASE)* ('else' BASE)?，然後修改本章的剖析器程式，加入可以處理該規則的程式。

12.6 繼續前一題，請修改本章的程式碼產生器程式，以產生上述的 if 規則之程式。