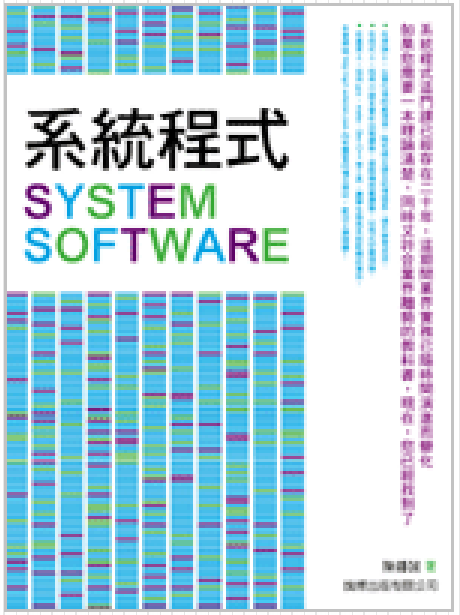
第 12 章、系統軟體實作

作者：陳鍾誠

旗標出版社



第 12 章、系統軟體實作

- 12.1　簡介
- 12.2　組譯器實作
- 12.3　虛擬機實作
- 12.4　剖析器實作
- 12.5　編譯器實作
- 12.6　整合測試

12.1 簡介

- C 語言實作
  - 動態陣列 + 雜湊表格 (12.1 節)
  - 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)
- 目的
  - 展示系統軟體的實作方法與技巧
  - 讓讀者熟悉系統軟體的原理
  - 培養開發系統軟體的能力
  - 以實作印證理論, 以理論支持實作

實作程式列表

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。

動態陣列

- 實作目的
  - 標準 C 語言函式庫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像符號表、指令表、詞彙串列、剖析樹等資料。

動態陣列的資料結構與函數

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
...
```

動態陣列新增 – ArrayAdd()

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

```
void ArrayAdd(Array *array, void *item) {
  if (array->count == array->size) {
    int itemSize = sizeof(void*);
    int newSize = array->count*2;
    void **newItems = ObjNew(void*, newSize);
    memcpy(newItems,
      array->item, array->size*itemSize);
    free(array->item);
    array->item = newItems;
    array->size = newSize;
  }
  array->item[array->count++] = item;
}
```

說明

加入 item 到 array 當中
如果大小不夠了

擴大 size 為兩倍
取得兩倍大小的空間
將舊資料複製到新陣列中

釋放原本的陣列
將指標指向新陣列
設定新陣列的大小

將 item 加入陣列中

動態陣列的使用

▶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);
}
```

ArrayTest() 函數的執行結果

```
John
Mary
George
Bob
ArrayPop()=Bob
ArrayLast()=George
ArrayFind(John)=0
ArrayFind(Mary)=1
ArrayFind(George)=2
ArrayFind(Bob)=-1
John
Mary
George
```

動態陣列的測試結果

▶範例 12.4 ArrayTest() 函數的執行結果

ArrayTest() 函數的執行結果	說明
John	利用 ArrayEach(…) 印出的所有元素
Mary	
George	
Bob	
ArrayPop()=Bob	取出最後的 Bob
ArrayLast()=George	印出最後一個元素為 George
ArrayFind(John)=0	利用 ArrayFind(…) 找尋所有元素
ArrayFind(Mary)=1	
ArrayFind(George)=2	
ArrayFind(Bob)=-1	由於 Bob 已經取出了，所以找不到
John	再度印出的所有元素
Mary	
George	(Bob 已去除)

動態陣列的列舉 - ArrayEach()

範例 12.5 ArrayEach() 函數的原始碼

檔案 ch12/Array.c 中的 ArrayEach() 函數

```
void ArrayEach(Array *array, FuncPtr1 f) {
  int i;
  for (i=0; i<array->count; i++)
    f(array->item[i]);
}
```

說明

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

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

```
...
// 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);
...
```

說明

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

雜湊表

- 實作目的
  - 標準 C 語言函式庫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像符號表、指令表、詞彙串列、剖析樹等資料。

雜湊表的資料結構與函數

▶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);`	Entry 的建立函數
`int EntryCompare(Entry *e1, Entry *e2);`	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

雜湊函數的實作

- 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
- 這個方法雖然不是很好的雜湊函數，但是簡單卻足夠了。

範例 12.8 雜湊函數實作方式

檔案 ch12/HashTable.c 中的雜湊函數 hash()

```
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;
}
```

說明

```
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：取得對應鍵值的元素
- HashTableGet(table, key)

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
}
```

雜湊表：放入鍵值與元素
- HashTablePut(table, key, data)

```
// 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
 }
}
```

12.2 組譯器實作

- 撰寫一個簡單的 CPU0 組譯器 – as0
- 印證組譯器的理論
- 學習組譯器的實作方式
- 使用方式
  - as0 <asmFile> <objFile>

組譯範例：as0 ArraySum.asm0 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	000000030000000700000004
0044	aptr: WORD a	00000038
0048	sum: WORD 0	00000000

組譯器的資料結構

▶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

組譯器的函數

```
void assemble(char *asmFile, char *objFile);

Assembler* AsmNew();
void AsmFree(Assembler *a);

void AsmPass1(Assembler *a, char *text);
void AsmPass2(Assembler *a);
void AsmSaveObjFile(Assembler *a,
     char *objFile);
void AsmTranslateCode(Assembler *a,
     AsmCode *code);

AsmCode* AsmCodeNew(char *line);
void AsmCodeFree(AsmCode *code);
int AsmCodePrintln(AsmCode *code);
```

組譯器的主程式

組譯器的建構函數
組譯器的解構函數

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

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(asmFile, objFile)

▶範例 12.12 CPU0 組譯器的主要函數 assemble()

檔案 ch12/Assembler.c 中的 assemble() 函數

```
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);
}
```

說明

```
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
```

組譯器的第一階段 (計算符號位址)

檔案 ch12/Assembler.c 中的 AsmPass1() 函數

```
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");
   for (i=0; i<lines->count; i++) {
     strReplace(lines->item[i], SPACE, ' ');
     AsmCode *code =
       AsmCodeNew(lines->item[i]);
     code->address = address;
     Op *op = HashTableGet(opTable,
       code->op);
     if (op != NULL) {
       code->opCode = op->code;
       code->type = op->type;
     }
     if (strlen(code->label)>0)
       HashTablePut(a->symTable,
            code->label, code);
     ArrayAdd(a->codes, code);
     AsmCodePrintln(code);
     code->size = AsmCodeSize(code);
     address += code->size;
   }
   ArrayFree(lines, strFree);
}
```

說明

第一階段的組譯

將組合語言分割成一行一行

對於每一行

建立指令物件

設定該行的位址
取得運算碼與型態

設定指令物件的
運算碼與型態

如果有標記符號
加入符號表中

建構指令物件陣列
印出觀察
計算指令大小
下一個指令位址

釋放記憶體

計算指令大小 – AsmCodeSize()

```
int AsmCodeSize(AsmCode *code) {
   switch (code->opCode) {
     case OP_RESW :
       return 4 * atoi(code->args);
     case OP_RESB :
       return atoi(code->args);
     case OP_WORD :
       return 4 * (strCountChar(
            code->args, ", ")+1);
     case OP_BYTE :
       return strCountChar(
            code->args, ", ")+1;
     case OP_NULL :
       return 0;
     default :
       return 4;
   }
}
```

計算指令的大小
根據運算碼 op
如果是 RESW
　大小為 4*保留量
如果是 RESB
　大小為 1*保留量
如果是 WORD
　大小為 4*參數個數

如果是 BYTE
　大小為 1*參數個數

如果只是標記
　大小為 0
其他情形 (指令)
　大小為 4

組譯器的第2階段

▶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

```
void AsmPass2(Assembler *a) {
   printf("=============PASS2==============\n");
   int i;
   for (i=0; i<a->codes->count; i++) {
      AsmCode *code = a->codes->item[i];
      AsmTranslateCode(a, code);
      AsmCodePrintln(code);
   }
}
```

說明

組譯器的第二階段

對每一個指令

進行編碼動作

指令轉機器碼 (J 型指令)

```
void AsmTranslateCode(Assembler *a,
   AsmCode *code) {
…
   char cxCode[9]="00000000", objCode[100]="";
…
   int pc = code->address + 4;
   switch (code->type) {
      case 'J' :
        if (!strEqual(args, "")) {
           labelCode=
              HashTableGet(a->symTable, args);
           cx = labelCode->address - pc;
           sprintf(cxCode, "%8x", cx);
        }
        sprintf(objCode, "%2x%s", code->opCode,
           &cxCode[2]);
        break;
   …
```

指令的編碼函數

…

宣告變數

…

提取後 PC 為位址+4

根據指令型態

處理 J 型指令

取得符號位址

計算 cx 欄位

編出目的碼(16 進位)

指令轉機器碼 (A 型指令)

```
case 'A' :
  sscanf(args, "%s %s %s", p1, p2, p3);
  sscanf(p1, "R%d", &ra);
  sscanf(p2, "R%d", &rb);
  if (sscanf(p3, "R%d", &rc)<=0)
     sscanf(p3, "%d", &cx);
  sprintf(cxCode, "%8x", cx);
  sprintf(objCode, "%2x%x%x%x%s",
     code->opCode, ra, rb, rc, &cxCode[5]);
  break;
```

處理 A 型指令
- 取得參數
- 取得 ra 暫存器代號
- 取得 rb 暫存器代號
- 取得 rc 暫存器代號
- 或者是 cx 參數

- 編出目的碼(16 進位)

資料轉二進位碼

```
      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

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
- 這是因為筆者所使用的處理器並不是 CPU0, 而是 Intel 的 IA32 CPU 為核心的電腦。
- 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。

虛擬機的執行 vm0 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	SUB R3, 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	SUB R3, 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	SUB R3, 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

執行後的傾印

```
===CPU0 dump registers===
IR =0x2c000000=738197504
R[00]=0x00000000=0
R[01]=0x0000000e=14
R[02]=0x00000044=68
R[03]=0x00000000=0
R[04]=0x00000004=4
R[05]=0x00000004=4
R[06]=0x00000000=0
R[07]=0x00000000=0
R[08]=0x0000000e=14
R[09]=0x00000001=1
R[10]=0x00000000=0
R[11]=0x00000000=0
R[12]=0x40000000=1073741824
R[13]=0x00000000=0
R[14]=0xffffffff=-1
R[15]=0x00000038=56
```

```
sum=R1=3+7+4=14
R2=陣列 a 的結尾

R4 = 4

R8=R1=sum
R9=1

SW (R12) 的 Z=1

RET 位址為-1,
結束 PC 最後為 0x38
```

虛擬機的資料結構與函數

▶範例 12.16 CPU0 虛擬機器的資料結構與主要函數

檔案 ch12/Cpu0.h	說明

typedef struct {
  BYTE *m;
  int mSize;
  int R[16], IR;
} Cpu0;
	CPU0 虛擬機器 Vm0 的主要結構
	代表記憶體的陣列
	記憶體的大小
	R0, R1, …, R15, IR 暫存器

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

虛擬機的最上層函數 – runObjFile(objFile)

▶範例 12.17 CPU0 虛擬機器 VM0 的主要程式

檔案 ch12/Cpu0.c

```
#include "Cpu0.h"

void runObjFile(char *objFile) {
  printf("===VM0:run %s on CPU0===\n", objFile);
  Cpu0 *cpu0 = Cpu0New(objFile);
  Cpu0Run(cpu0, 0);
  Cpu0Dump(cpu0);
  Cpu0Free(cpu0);
}

Cpu0* Cpu0New(char *objFile) {
  Cpu0 *cpu0=ObjNew(Cpu0, 1);
  cpu0->m = newFileBytes(objFile, &cpu0->mSize);
  return cpu0;
}

void Cpu0Free(Cpu0 *cpu0) {
  freeMemory(cpu0->m);
  ObjFree(cpu0);
}
```

說明

```
虛擬機器主函數

建立 CPU0 物件
開始執行
傾印暫存器
釋放記憶體

建立 CPU0 物件
 分配記憶體
 載入 objFile

釋放 CPU0 物件
```

虛擬機使用的位元操作函數

```
#define bits(i, from, to) \
  ((UINT32) i << (31-to) >> (31-to+from))
#define ROR(i, k) \
  (((UINT32)i>>k)|(bits(i, 32-k, 31)<<(32-k)))
#define ROL(i, k) \
  (((UINT32)i<<k)|(bits(i, 0, k-1)<<(32-k)))
#define SHR(i, k) ((UINT32)i>>k)
#define SHL(i, k) ((UINT32)i<<k)
#define bytesToInt32(p) \
  (INT32)(p[0]<<24|p[1]<<16|p[2]<<8|p[3])
#define bytesToInt16(p) (INT16)(p[0]<<8|p[1])
#define int32ToBytes(i, bp) \
  { bp[0]=i>>24; bp[1]=i>>16; \
   bp[2]=i>>8; bp[3]=i;}
#define StoreInt32(i, m, addr) \
  { BYTE *p=&m[addr]; int32ToBytes(i, p); }
#define LoadInt32(i, m, addr) \
  { BYTE *p=&m[addr]; i=bytesToInt32(p); }
#define StoreByte(b, m, addr) \
  { m[addr] = (BYTE) b; }
#define LoadByte(b, m, addr) { b = m[addr]; }
```

```
取得 from 到 to
  之間的位元
向右旋轉 k 位元

向左旋轉 k 位元

向右移位 k 位元
向左移位 k 位元
4 byte 轉 int

2 byte 轉 INT16
int 轉為 4 byte


i=m[addr…addr+3]

m[addr..addr+3]=i

m[addr]=b

b=m[addr]
```

定義暫存器別名

```
#define PC R[15]
#define LR R[14]
#define SP R[13]
#define SW R[12]
```

```
PC is R[15]
LR is R[14]
SP is R[13]
SW is R[12]
```

虛擬機的主要函數 – Cpu0Run()

```
void Cpu0Run(Cpu0 *cpu0, int start) {
  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;
  LR = -1;
  BOOL stop = FALSE;
  while (!stop) {
   R[0] = 0;
   LoadInt32(IR, m, PC);
   cpu0->IR = IR;
   PC += 4;
   op = bits(IR, 24, 31);
   ra = bits(IR, 20, 23);
   rb = bits(IR, 16, 19);
   rc = bits(IR, 12, 15);
   c5 = bits(IR,  0, 4);
```

虛擬機器的主要執行函數

設定起始位址準備開始執行

如果尚未結束
R[0] 永遠為 0
指令擷取
IR←m[PC..PC+3]
擷取完將 PC 加 4 指向下一個指令
  取得 op 欄位
  取得 Ra 欄位
  取得 Rb 欄位,
  取得 Rc 欄位
取得 5, 12, 16, 24

Cpu0Run() - 指令擷取階段

```
void Cpu0Run(Cpu0 *cpu0, int start) {
  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;
  LR = -1;
  BOOL stop = FALSE;
  while (!stop) {
   R[0] = 0;
   LoadInt32(IR, m, PC);
   cpu0->IR = IR;
   PC += 4;
```

虛擬機器的主要執行函數

設定起始位址準備開始執行

如果尚未結束
R[0] 永遠為 0
指令擷取
IR←m[PC..PC+3]
擷取完將 PC 加 4 指向下一個指令

CpuORun() – 解碼階段

```
op = bits(IR, 24, 31);
ra = bits(IR, 20, 23);
rb = bits(IR, 16, 19);
rc = bits(IR, 12, 15);
c5 = bits(IR,  0, 4);
c12= bits(IR, 0, 11);
c16= bits(IR, 0, 15);
c24= bits(IR, 0, 23);
N  = bits(SW, 31, 31);

Z  = bits(SW, 30, 30);

if (bits(IR, 11, 11)!=0) c12 |= 0xFFFFF000;
if (bits(IR, 15, 15)!=0) c16 |= 0xFFFF0000;
if (bits(IR, 23, 23)!=0) c24 |= 0xFF000000;
caddr = R[rb]+c16;
raddr = R[rb]+R[rc];
```

```
  取得 op 欄位
  取得 Ra 欄位
  取得 Rb 欄位,
  取得 Rc 欄位
取得 5, 12, 16, 24
  位元的常數欄位


  取得狀態暫存器中
    的 N 旗標
  取得狀態暫存器中
    的 Z 旗標
  若為負數,
    則調整為 2 補數格式

  取得位址 [Rb+cx]
  取得位址 [Rb+Rc]
```

CpuORun() – 執行階段 (載入指令)

```
switch (op) {
  case OP_LD : LoadInt32(R[ra], m, caddr);
   break;
  case OP_ST : StoreInt32(R[ra], m, caddr);
   break;
  case OP_LDB : LoadByte(R[ra], m, caddr);
   break;
  case OP_STB : StoreByte(R[ra], m, caddr);
   break;
  case OP_LDR: LoadInt32(R[ra], m, raddr);
   break;
  case OP_STR: StoreInt32(R[ra], m, raddr);
   break;
  case OP_LBR: LoadByte(R[ra], m, raddr);
   break;
  case OP_SBR: StoreByte(R[ra], m, raddr);
   break;
  case OP_LDI: R[ra] = c16; break;
```

根據 op 執行動作

處理 LD 指令

處理 ST 指令

處理 LDB 指令

處理 STB 指令

處理 LDR 指令

處理 STR 指令

處理 LBR 指令

處理 SBR 指令

處理 LDI 指令

CpuORun() – 執行階段 (運算指令)

```
case OP_CMP: {
 if (R[ra] > R[rb]) {
   SW &= 0x3FFFFFFF;
 } else if (R[ra] < R[rb]) {
   SW |= 0x80000000;
   SW &= 0xBFFFFFFF;
 } else {
   SW &= 0x7FFFFFFF;
   SW |= 0x40000000;
 }
 ra = 12;
 break;
} case OP_MOV: R[ra] = R[rb]; break;
case OP_ADD: R[ra] = R[rb] + R[rc]; break;
case OP_SUB: R[ra] = R[rb] - R[rc]; break;
case OP_MUL: R[ra] = R[rb] * R[rc]; break;
case OP_DIV: R[ra] = R[rb] / R[rc]; break;
case OP_AND: R[ra] = R[rb] & R[rc]; break;
case OP_OR:  R[ra] = R[rb] | R[rc]; break;
case OP_XOR: R[ra] = R[rb] ^ R[rc]; break;
case OP_ROL: R[ra] = ROL(R[rb], c5); break;
case OP_ROR: R[ra] = ROR(R[rb], c5); break;
case OP_SHL: R[ra] = SHL(R[rb], c5); break;
case OP_SHR: R[ra] = SHR(R[rb], c5); break;
```

```
處理 CMP 指令
> : SW(N=0, Z=0)
 設定 N=0, Z=0
< : SW(N=1, Z=0, ....)
 設定 N=1;
 設定 Z=0;
= : SW(N=0, Z=1)
 設定 N=0;
 設定 Z=1;

在指令執行完後
  輸出 R12
處理 MOV 指令
處理 ADD 指令
處理 SUB 指令
處理 MUL 指令
處理 DIV 指令
處理 AND 指令
處理 OR 指令
處理 XOR 指令
處理 ROL 指令
  處理 ROR 指令
  處理 SHL 指令
  處理 SHR 指令
```

CpuORun() – 執行階段 (跳躍指令)

```
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;           處理 JLT 指令
 break;
case OP_JGT: if (N==0&&Z==0) PC += c24;           處理 JGT 指令
 break;
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;
```

CpuORun() – 執行階段 (結尾)

```
      case OP_PUSH:SP-=4;
       StoreInt32(R[ra], m, SP); break;
      case OP_POP: LoadInt32(R[ra], m, SP);
       SP+=4; break;
      case OP_PUSHB:SP--;
       StoreByte(R[ra], m, SP); break;
      case OP_POPB:LoadByte(R[ra], m, SP);
       SP++; break;
      default:
      printf( "Error:invalid op (%02x)  ", op);
    }
    sprintf(buffer,  "PC=%08x IR=%08x SW=%08x
      R[%02d]=0x%08x=%d\n" ,
      PC, IR, SW, ra, R[ra], R[ra]);
    strToUpper(buffer);
    printf(buffer);
   }
}
```

處理 PUSH 指令

處理 POP 指令

處理 PUSHB 指令

處理 POPB 指令

印出 PC, IR, SW,
R[ra], R[rb]
暫存器的值，以利觀察

虛擬機：傾印暫存器 - Cpu0Dump()

```
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

12.4 剖析器實作

- C0 語言的剖析器 (Parser)
  - 是編譯器與直譯器中的關鍵程式
  - 作為 c0c 編譯器的語法剖析程式
  - 會呼叫詞彙掃描器 Scanner 取的詞彙

掃描器

- 詞彙掃描器 Scanner 是一個較為簡單的物件
- 用來取得下一個詞彙 (token)
- 提供剖析器呼叫使用

掃描器的使用方法

```
Array* tokenize(char *text) {
   Array *tokens = ArrayNew(10);
   Scanner *scanner = ScannerNew(text);
   char *token = NULL;
   while ((token = ScannerScan(scanner))
      != NULL) {
      ArrayAdd(tokens, newStr(token));
      printf("token=%s\n", token);
   }
   ScannerFree(scanner);
   return tokens;
}
```

將程式轉換成一個一個的詞彙

不斷取出下一個詞彙，
直到程式字串結束為止

掃描器的資料結構

▶範例 12.18 C0 語言掃描器的資料結構

檔案 Scanner.h

```
typedef struct {
  char *text;
  int textLen;
  int textIdx;
  char token[MAX_LEN];
} Scanner;
```

說明

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

判斷詞彙的型態 – tokenToType(token)

```
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
```

掃描器的主要函數

檔案 Scanner.c

```
char *ScannerScan(Scanner *scanner) {
  while (strMember(ch(), SPACE))
    next();
  if (scanner->textIdx >=
    scanner->textLen)
    return NULL;
  char c = ch();
  int begin = scanner->textIdx;
  if (c == '\"') { // string = ".."
    next(); // skip begin quote "
    while (ch() != '\"') next();
    next(); // skip end quote "
  } else if (strMember(c, OP)) {
    while (strMember(ch(), OP)) next();
  } else if (strMember(c, DIGIT)) {
    while (strMember(ch(), DIGIT))
     next();
  } else if (strMember(c, ALPHA)) {
    while (strMember(ch(), ALPHA)
      || strMember(ch(), DIGIT))
     next();
  } else
    next();
  strSubstr(scanner->token,
    scanner->text, begin,
    scanner->textIdx-begin);
  return scanner->token;
}
```

說明

```
掃描下一個詞彙
忽略空白

檢查是否超過範圍


取得下一個字元
記住詞彙開始點
如果是 ", 代表字串開頭,
   一直讀到下一個 " 符號
   為止。

如果是 OP(+-*/<=>!等符號)
   一直讀到不是 OP 為止
如果是數字
   一直讀到不是數字為止

如果是英文字母
   一直讀到不是英文字母
   (或數字)為止 (ex: x1y2z)

否則, 傳回單一字元

設定 token 為 (begin…textIdx)
   之間的子字串

傳回 token 詞彙
```

剖析器

- Parser.h
  - 剖析器的資料結構與函數宣告
- Parser.c
  - 剖析器的程式實作

剖析器的資料結構與函數

▶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;
	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

Parser *parse(char *text);	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

剖析器的最上層函數 parse()

▶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) {	剖析器的主要函數
Parser *p=ParserNew();	建立剖析器
ParserParse(p, text);	開始剖析
return p;	傳回剖析器
}	
…	
void ParserParse(Parser *p, char *text) {	剖析物件的主函數
printf("======= tokenize =======\n");	首先呼叫掃描器的主函數
p->tokens = tokenize(text);	tokenize() 將程式轉換為
printTokens(p->tokens);	詞彙串列
p->tokenIdx = 0;	
printf("======= parsing ========\n");	開始剖析
}	
printf("parse fail:stack.count=%d",	那就是剖析成功；
p->stack->count);	否則提示錯誤訊息
error();	
}	

遞迴下降剖析法

```
// PROG = BaseList
Tree *parseProg(Parser *p) {
  push(p, "PROG");
  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");
}
```

```
剖析 PROG=BaseList 規則

建立 PROG 的樹根
 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 迴圈語法

```
// 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 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
```

剖析器中的 next() 函數

▶範例 12.22 C0 語言剖析器的程式片段

檔案 Parser.c

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

說明

```
...
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)
    印出錯誤訊息


  前進到下一個節點
```

剖析器中的 push(), pop() 函數

```
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;
}
...
```

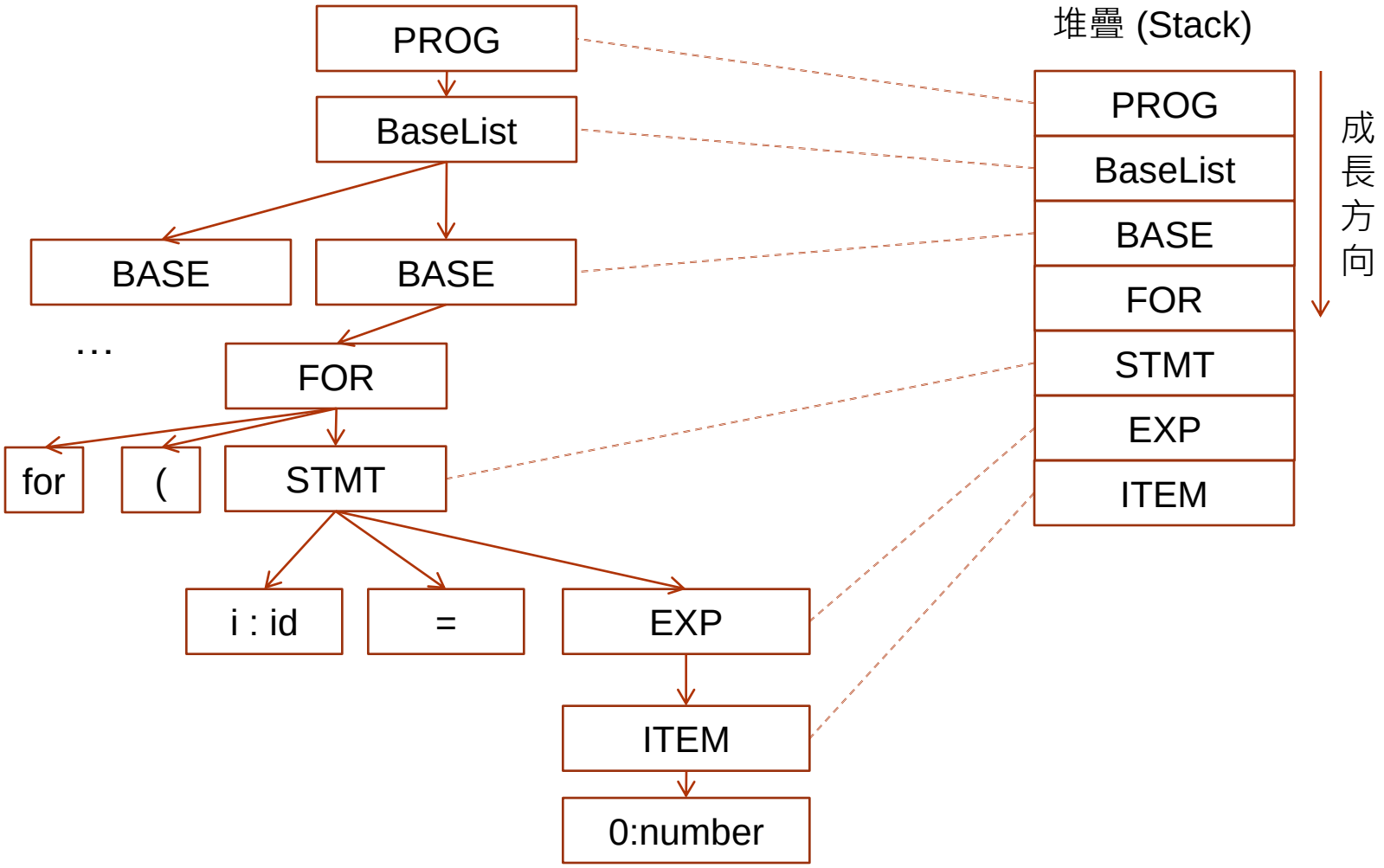
```
建立 pType 型態的子樹,
推入堆疊中

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　遞迴下降剖析器的執行過程



12.5 編譯器實作

- c0c 編譯器
  - 輸入：C0 語法的程式
  - 輸出：CPU0 的組合語言

- 執行方法
  - c0c <c0File> <asmFile>

編譯的範例

(a) ch12/test.c0

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

(b) 虛擬碼 pcode

```
  =       0     sum
  =       0     i
FOR0:
  CMP     i     0
  J       >     _FOR0
  +       sumi  T0
  =       T0    sum
  +i      1     i
  J             FOR0
_FOR0:
  RET           sum
```

(c) 組合語言 ch12/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
  LD   R1 i
  LDI  R2 1
  ADD  R3 R1    R2
  ST   R3 i
  JMP  FOR0
_FOR0:
LDR1   sum
  RET
sum:   RESW 1
i:     RESW 1
T0:    RESW 1
```

編譯器的最上層函數

- 範例 12.24 的 compile(cFile, asmFile)
  - 剖析器：parse(cText)
  - 程式產生器： generate(parser->tree, asmFile)

▶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);
}
```

說明

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

程式碼產生器的最上層函數

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

程式碼產生器的資料結構與函數

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

程式碼產生器 GenCode() – 開頭

▶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) {
  strcpy(nullVar, "");
  strcpy(rzVar, "");
  if (node == NULL) return NULL;
	遞迴產生節點
	node 的程式碼
	
	遞迴終止條件

GenCode() – 處理 FOR

```
if (strEqual(node->type, "FOR")) {                    處理 FOR 節點
  // FOR ::= for (STMT;COND;STMT) BLOCK
  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;
```

GenCode() – 處理 STMT

```
} else if (strEqual(node->type, "STMT")) {
    // STMT:= return | id '=' EXP |
    //                      id ('++' | '-- ')
    Tree *c1 = node->childs->item[0];
    if (strEqual(c1->type, "return")) {
     Tree *id = node->childs->item[1];
     GenPcode(g, "", "RET", "", "", id->value);
    } else {
     Tree *id = node->childs->item[0];
     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);
     }
    }
```

```
處理 STMT 節點


   取得子節點
處理 return 指令

中間碼： 例如 RET sum

   取得子節點

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
```

GenCode() – 處理 COND

```
} 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);
```

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 >

GenCode() – 處理 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);
  }
  strcpy(rzVar, var1);
```

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

傳回臨時變數，例如 T0

GenCode() – 結尾

```
  } 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;
}
```

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

輸出中間碼 P-Code

▶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

P-Code 轉為組合語言 (前半段)

```
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", "");
```

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

P-Code 轉為組合語言 (後半段)

```
  } else if (strEqual(op, "J")) { // J op label
    char asmOp[100];
    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, "", "");
  } else if (strEqual(op, "RET")) {
    GenAsmCode(g, "", "LD", "R1", pTo, "");
    GenAsmCode(g, "", "RET", "", "", "");
  }
}
```

處理 J (J > _FOR)
根據 op 設定
跳躍指令

例如 JGT _FOR0
例如 RET sum
轉成 LD R1, sum
RET

輸出組合語言

```
void GenAsmCode(Generator *g, char* label,
   char* op, char* p1, char* p2, char* pTo) {
   char *realOp = op;
   if (strEqual(op, "LD" ))
     if (isdigit(p2[0]))
       realOp = "LDI" ;
   fprintf(g->asmFile,
      "%-8s %-4s %-4s %-4s %-4s\n" ,
     label, realOp, p1, p2, pTo);
}
```

輸出組合語言指令

如果指令是 LD，而且
p2 為常數
改用 LDI 指令
輸出組合語言指令

12.6 整合測試

- 單一主程式, 以條件編譯的方式
  - 編譯出 test, c0c, as0, vm0 等四個執行檔

- 方法是利用 C 語言的巨集編譯指令
  - #if … #elif…#endif

整個系統的主程式 (前半段)

▶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",`	預設組合語言為
`  *asmFile=asmFile0;`	test.asm0
`  char objFile0[]="test.obj0",`	預設目的檔為
`  *objFile=objFile0;`	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

整個系統的主程式 (後半段)

```
   OpTableTest();
   compile(cFile, asmFile);
   assemble(asmFile, objFile);
   runObjFile(objFile);
   checkMemory();
#elif TARGET==C0C
   if (argc == 3) {
      cFile=argv[1]; asmFile=argv[2];
   } else
      argError("c0c <c0File> <asmFile>");
   compile(cFile, asmFile);
#elif TARGET==AS0
   if (argc == 3) {
      asmFile=argv[1]; objFile=argv[2];
   } else
      argError("as0 <asmFile> <objFile>");
   assemble(asmFile, objFile);
#elif TARGET==VM0
   if (argc == 2)
      objFile=argv[1];
   else
      argError("vm0 <objFile>");
   runObjFile(objFile);
#endif
   system("pause");
   return 0;
}
```

```
   測試指令表物件
   測試編譯器
   測試組譯器
   測試虛擬機器
   檢查記憶體使用狀況
如果編譯目標為 C0C
   如果有 3 個參數
      設定參數
   否則
      提示程式執行方法
   開始編譯
如果編譯目標為 AS0
   如果有 3 個參數
      設定參數
   否則
      提示程式執行方法
   開始組譯
如果編譯目標為 VM0
   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)
```

專案建置檔 makefile (第1部分)

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

專案建置檔 makefile (第2部分)

```
  $(CC) main.c $(LINKOBJ) -DTARGET=TEST \
    -o test $(LIBS)

c0c: $(OBJ)                                          c0c 編譯器
  $(CC) main.c $(LINKOBJ) -DTARGET=C0C \
    -o c0c $(LIBS)

as0: $(OBJ)                                          as0 組譯器
  $(CC) main.c $(LINKOBJ) -DTARGET=AS0 \
    -o as0 $(LIBS)

vm0: $(OBJ)                                          vm0 虛擬機
  $(CC) main.c $(LINKOBJ) -DTARGET=VM0 \
    -o vm0 $(LIBS)

clean:                                               清除上一次 make 所
  ${RM} $(OBJ) $(BIN)                                產生的檔案

Parser.o: Parser.c                                   剖析器
  $(CC) -c Parser.c -o Parser.o $(CFLAGS)

Tree.o: Tree.c                                       語法樹
  $(CC) -c Tree.c -o Tree.o $(CFLAGS)

Lib.o: Lib.c                                         基礎函式庫
  $(CC) -c Lib.c -o Lib.o $(CFLAGS)
```

專案建置檔 makefile (第3部分)

```
Scanner.o: Scanner.c
  $(CC) -c Scanner.c -o Scanner.o $(CFLAGS)

Array.o: Array.c
  $(CC) -c Array.c -o Array.o $(CFLAGS)

Compiler.o: Compiler.c
  $(CC) -c Compiler.c -o Compiler.o $(CFLAGS)

HashTable.o: HashTable.c
  $(CC) -c HashTable.c -o HashTable.o $(CFLAGS)

Generator.o: Generator.c
  $(CC) -c Generator.c -o Generator.o $(CFLAGS)

Assembler.o: Assembler.c
  $(CC) -c Assembler.c -o Assembler.o $(CFLAGS)

Cpu0.o: Cpu0.c
  $(CC) -c Cpu0.c -o Cpu0.o $(CFLAGS)

OpTable.o: OpTable.c
  $(CC) -c OpTable.c -o OpTable.o $(CFLAGS)
```

掃描器

動態陣列

編譯器

雜湊表

程式產生器

組譯器

虛擬機

指令表

建置執行過程 (1)

▶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	切換到 ch12/
C:\ch12>make	開始建置專案
gcc.exe -D__DEBUG__ -c Parser.c -o Parser.o -g3	專案編譯過程
…	…

執行：編譯器 (剖析)

```
C:\ch12>c0c test.c0 test.asm0
...
...
...
...
======= parsing ========
+PROG
 +BaseList
  +BASE
   +STMT
      idx=0, token=sum, type=id
      idx=1, token==, type==
     +EXP
      idx=2, token=0, type=number
     -EXP
   -STMT
...
```

編譯 test.c0
輸出 test.asm0
組合語言

印出剖析樹
...

執行：編譯器 (中間碼產生)

```
=====PCODE=====
                =     0          sum
                =     0          i
FOR0:
                CMP   i     10
                J     >           _FOR0
                +     sum   i     T0
                =     T0          sum
                +     i     1     i
                J                 FOR0
_FOR0:
                RET               sum
```

印出虛擬碼

…

執行：編譯器 (產生組合語言)

```
=====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
                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
```

印出組合語言程式

…

執行：組譯器 (第1階段：計算符號位址)

```
C:\ch12>as0 test.asm0 test.obj0

Assembler:asmFile=test.asm0 objFile=test.obj0
===============Assemble=============
0000          LDI   R1    0          L   8 (NULL)
0004          ST    R1    SUM        L   1 (NULL)
```

組譯 test.asm0,
輸出 test.obj0
開始組譯
第一階段組譯
(計算位址)

```
...
===============SYMBOL TABLE=========
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)
```

印出符號表

執行：組譯器 (第2階段：指令轉機器碼)

```
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
```

執行：組譯器 (輸出：目的碼)

```
==========Save to ObjFile:test.obj0==========
08100000011F004C08100000011F00480…
```

將目的碼存入
test.obj0

執行：虛擬機

```
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
```

結語

- 系統軟體實作
  - 本章以 C 語言實作了
    - 動態陣列
    - 雜湊表
    - 編譯器
      - c0c test.c0 test.asm0
    - 組譯器
      - as0 test.asm0 test.obj0
    - 虛擬機
      - vm0 test.obj0

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 請利用 c0c 編譯器, 將 fib.c0 編譯為組合語言 fib.asm0。
- 12.3 請利用 as0 組譯器, 將 fib.asm0 組譯為目的檔 fib.obj0。
- 12.4 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 ')' BLOCK (' elseif' BLOCK)* (else BLOCK)?, 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。