

系統程式-課程筆記

- 指令

<code>gcc sum.c -o sum</code>	用 gcc 編譯器去編譯 sum.c 程式，輸出一個執行檔 sum
<code>./sum</code>	執行 sum

- 把 codeblocks 編譯器加進 vs code：

1. 找到資料夾.....codeblocks\MinGW\bin 並複製路徑
2. 去控制台，選擇系統及安全性->系統->進階系統設定
3. 環境變數，選擇系統變數的 path->新增並貼上路徑->重開 vs code

- hash 雜湊：把一個字串用固定的方式轉成一種數字

ex:

`unsigned int h = 37`

`h = h*147 + *p;` *//*p 是 ASCII 碼*

`hash(h) = 37` `hash(h) = 5543` *//5543=37*147+104(h 的 ASCII)*

`hash(h) = 814922 = 5543*147+101` *//e 是 101 的 ASCII*

04-map

(main.c)循序搜尋

`mapNew(&jMap, 17);`

呼叫 `mapNew`：建立一個大小為 17 的表格，變數名為 `jmap`

`jMap.table = jList;`：把 `jList` 塞進去

`jMap.top = 8;`：有 8 個元素

`mapLookup(&jMap, "JLE");`：在 `jMap` 裡尋找“JLE”結構

(map.c)

`map->size = size;`：在 `main.c` 裡設為 17，東西不能塞超過 17 個

`mapFind`：在 `map` 陣列裡找

生成語法

S	=	N	Y
句子	=	名詞	動詞

N = cat | dog |:or 的意思

V = run | eat

N->dog, V->run 產生 dog run

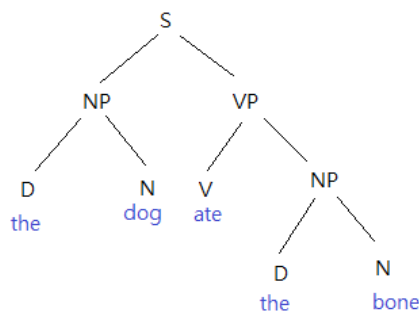
• BNF 語法

S = NP VP 句子 = 名詞子句 + 動詞子句

NP = D N 名詞子句 = 定詞 + 名詞

VP = V NP 動詞子句 = 動詞 + 名詞子句

語法樹：



E = F ([+ -]) *

E:運算式 F:Factor () *:可出現 0 次以上

F = Number | ' (' E ') '

//E 只能寫 0~9 ex: ' 3 + 5 '

運算式編譯器 (exp0)

ex:

E = F ([+ -]) * //假設輸入 ' 3 + 5 '

F = Number | ' (' E ') '

parse: 3 + 5

結果：

t0 = 3 //t0 是一個暫存器

t1 = 5 //t1 存 5

t2 = t0 + t1 //t2 = t0 + t1 = 3 + 5，最後產生的東西放在 t2

部分程式碼註解：

// 取得目前字元，同時進到下一格

```
char ch() {  
    char c = tokens[tokenIdx];  
    return c;  
}
```

// 取得目前字元，同時進到下一格

```
char next() {  
    char c = ch();  
    tokenIdx++;  
    return c;  
}
```

// ex: isNext("+ -") 用來判斷下一個字元是不是 + 或 -

```
int isNext(char *set) {  
    char c = ch();  
    return (c!='\0' && strchr(set, c)!=NULL);  
}
```

// 產生下一個臨時變數的代號， ex: 3 代表 t3。

```
int nextTemp() {  
    static int tempIdx = 0;  
    return tempIdx++;  
}
```

// F = Number | '(' E ')'

```
int F() {  
    int f;  
    char c = ch();  
    if (isdigit(c)) {  
        next(); // skip c  
        f = nextTemp();  
        printf("t%d=%c\n", f, c);  
    } else if (c=='(') { // '(' E ')'  
        next();  
        f = E();  
        assert(ch()==')');  
        next();  
    } else {  
        error("F = (E) | Number fail!");  
    }  
    return f;  
}
```

// E = F ([+-] F)*

```
int E() {
    int i1 = F();
    while (isNext("+ -")) {
        char op=next();
        int i2 = F();
        int i = nextTemp();
        printf("t%d=t%d%ct%d\n", i, i1, op, i2);
        i1 = i;
    }
    return i1;
}
```

輸入指令：gcc exp0.c -o exp0：編譯出一個執行檔 exp0.exe

```
co108a > sp > code > c > 02-compiler > 00-exp0 > C EE.c
1  #include <stdio.h>
2  void F();
3
4  // E = F
5  void E() {
6      printf("E started\n");
7      // E();
8      F();
9      printf("E finished\n");
10 }
11
12 // F = 'F'
13 void F() {
14     printf(" F started\n");
15     printf(" F\n");
16     printf(" F finished\n");
17 }
18
19 int main() {
20     E();
21 }
22
```

E 印出 E started

E 呼叫了 F，此 F 就是下面的 F，但 F 比後面定義，但在前面就呼叫了，所以前面要加一個 void F ()

主程式呼叫一個 E

結果：

E started
F started
F finished
E finished

```

void parse(char *str) {
    tokens = str;
    E();
}

int main(int argc, char * argv[]) {
    printf("argv[0]=%s argv[1]=%s\n", argv[0], argv[1]);
    printf("=== EBNF Grammar =====\n");
    printf("E=F ([+-] F)*\n");
    printf("F=Number | '(' E ')' \n");
    printf("==== parse:%s =====\n", argv[1]);
    parse(argv[1]);
}

```

parse(argv[1]); //把 argv[1]傳進 parse 函數

arg: 參數 argc: 參數個數 argv: 參數變數, 陣列

argv[1]: 第一個參數

./lexer 第 0 個參數 sum.c 第一個參數

- 若輸入./exp0 'x + 5 - y' 就會編譯中間碼

#t0 = x , @x (變數), D = M / @5 (數字), D = A , #t1 = 5

exp1

```

printf("E=T ([+-] T)*\n");
printf("T=F ([*/] F)*\n");

```

越下層的運算，優先序越高

執行./exp1 '3+5*8'

結果：

```

PS C:\Users\user\Desktop\110710519\co108a\sp\code\c\02-compiler\01-exp1> ./exp1 '3+5*8'
=== EBNF Grammar =====
E=T ([+-] T)*
T=F ([*/] F)*
F=Number | Id | '(' E ')'
==== parse:3+5*8 =====
t0=3
t1=5
t2=8
t3=t1*t2
t4=t0+t3

```

E = 3 + 5 * 8

呼叫E去比對	T + T	展開去比對
整個'式子'	F * F	
	5 8	

Compiler.c 語法

PROG = STMTS 一個程式就是一堆陳述

BLOCK = { STMTS } 一個區塊就是有{ }中間加一堆陳述

STMTS = STMT* STMT 是一堆 0 次以上的 STMT

STMT = WHILE / BLOCK / ASSIGN

一個陳述有可能是 while 迴圈/BLOCK 區塊/ASSIGN 指定

WHILE = while (E) STMT

一個 while 迴圈是用 while 開頭，再接(運算式)，最後陳述

ASSIGN = id = ' ' E 一個指定是一個變數名稱，他等於一個運算式

E = F (OP E)*

F = (E) | Number | ID

05-compiler-run

執行指令：

minge32-make : 編譯原始碼的建置工具

./compiler test/sum.c -ir -run

只打 -ir 代表會輸出中間碼但不執行
若加上 -run 代表除了輸出中間碼，還會執行

irvm.c 裡的 trace = printf 印出指令 = emit

*xxxx.c 檔裡面放程式碼 xxxx.h 檔裡面放標頭，放 define 的東西

```
// while (E) STMT
void WHILE() {
    int whileBegin = nextLabel();
    int whileEnd = nextLabel();
    irEmitLabel(whileBegin);
    // emit("(L%d)\n", whileBegin);
    skip("while");
    skip("(");
    int e = E();
    irEmitIfNotGoto(e, whileEnd);
    // emit("goif T%d L%d\n", whileEnd, e);
    skip(")");
    STMT();
    irEmitGoto(whileBegin);
    // emit("goto L%d\n", whileBegin);
    irEmitLabel(whileEnd);
    // emit("(L%d)\n", whileEnd);
}
```

irEmitLabel :

本來是直接 emit，變成直接呼叫 irEmitLabel。目的是讓它格式統一，可以存在陣列裡。

```
#define emit printf
```

ir.h 裡面定義的東西

```
extern void irEmitArg(int t1) ;
extern void irEmitCall(char *fname, int t1);
extern void irEmitAssignTs(int t, char *s);
extern void irEmitAssignSt(char *s, int t);
extern void irEmitOp2(int t, int p1, char *op, int p2);
extern void irEmitLabel(int label);
extern void irEmitGoto(int label);
extern void irEmitIfGoto(int t, int label);
extern void irEmitIfNotGoto(int t, int label);
extern void irDump();
```

```
typedef struct {
    IrType type;
    int t, t1, t2, label;
    char *s, *op;
} IR;
```

t,t1,t2:臨時參數 label:標記代號
op:做什麼動作。Ex:加法 *s:目標參數

整個結構會儲存成一個陣列，放在 ir[] 裡面

```
void irEmitAssignTs(int t, char *s) {
    irNew((IR) {.type=IrAssignTs, .op="t=s", .t=t, .s=s});
}
```

IrAssignTs : t = sum 之類的 t:取得代號 s:取得名字

```
void irDump() {
    printf("====irDump()=====\n");
    for (int i=0; i<irTop; i++) {
        printf("%02d: ", i);
        irPrint(&ir[i]);
    }
}
```

irDump:把中間碼全部印出來

irvm.c:用來執行中間碼的程式

```
if (eq(op, "s=t")) {    //如果是 s=t 這樣的指令
    int *vp = varLookup(p->s);
    *vp = t[p->t];    //先把 t 取出來，再塞到新的變數
    trace("%s = t%d (%d)\n", p->s, p->t, *vp);    //印出來
}
```

檔案間的關係圖：

```
main.c => lexer.c
        compiler.c
        ir.c      => irvm.c
```

main.c 會呼叫 compiler.c / lexer.c / ir.c

ir.c 裡面會呼叫 irvm.c 其他的都是呼叫對應的.h 檔

asmVm

執行指令：

```
mingw32-make
./asm ../test/Add
```

測試範例：Add.asm

讓 $R0 = 2 + 3$

// Computes $R0 = 2 + 3$

@2

D=A

@3

D=D+A

@0

M=D //讓第 0 個記憶體塞入 $2 + 3$

執行結果：

00: @2	00000000000000010 0002
01: D=A	1110110000010000 ec10
02: @3	00000000000000011 0003
03: D=D+A	1110000010010000 e090
04: @0	0000000000000000 0000
05: M=D	1110001100001000 e308

把 hack 的指令，每一個都轉成機器碼。2 進位 (1110...000)；16 進位(ec10)
把組合語言編成機器碼的方式，就稱為組譯器

第一階段目的：編出每一個符號的位置

```
void pass1(string inFile) { //輸入是一個檔案名稱
    printf("===== PASS1 =====\n");
    char line[100]="";
    FILE *fp = fopen(inFile, "r"); //開檔案
    int address = 0;
    while (fgets(line, sizeof(line), fp)) { //開完之後就一行一行讀
        char *code = parse(line); //如果那行是註解(如果不是就編碼)
        if (strlen(code)==0) continue; //就在這裡做 continue
        printf("%02d:%s\n", address, code);
        if (code[0] == '(') {
            char label[100];
            sscanf(code, "(%[^)])", label);
            symAdd(&symMap, label, address); // 記住符號位址，給 pass2 編碼時使
用
        } else {
            address ++;
        }
    }
    fclose(fp);
}
```

```
===== PASS1 =====
00:@10
01:D=A
02:@0
03:M=D
04:@i
05:M=1
06:@sum
07:M=0
08:(LOOP)
    p.key=LOOP *p.value=8 top=24
08:@i
09:D=M
10:@R0
11:D=D-M
12:@STOP
13:D;JGT
14:@i
```

碰到標記(LOOP)就記住位置
p.key=LOOP 是 8
接下來每個指令都一直加 1

第二階段：處理文字指令轉成二進位(真正編碼的動作)

```
void pass2(string inFile, string hackFile, string binFile) {
    printf("===== PASS2 =====\n");
    char line[100], binary[17];
    FILE *fp = fopen(inFile, "r"); // 開啟組合語言檔
    FILE *hfp = fopen(hackFile, "w"); // 開啟輸出的 .hack 二進位字串檔案
    FILE *bfp = fopen(binFile, "wb"); // 開啟輸出的 .bin 二進位檔
    int address = 0;
    while (fgets(line, sizeof(line), fp)) { // 一行一行讀
        char *code = parse(line); // 取得該行的程式碼部分
        if (strlen(code)==0) continue;
        if (line[0] == '(') { // 這行是符號 ex: (LOOP)
            printf("%s\n", line); // 印出該符號
        } else {
            code2binary(code, binary); // 將指令編碼為二進位字串 string
            uint16_t b = c6btoi(binary); // 將二進位字串 string 轉成 int16
            printf("%02X: %-20s %s %04x\n", address, code, binary, b);
            fprintf(hfp, "%s\n", binary); // 輸出 .hack 的二進位字串檔
            fwrite(&b, sizeof(b), 1, bfp); // 輸出 .bin 的二進位檔
            address ++;
        }
    }
    fclose(fp);
    fclose(hfp);
    fclose(bfp);
}
```

因為 PASS2 有輸出，所以會開一個 16 進位的輸出檔，然後再開一個二進位的輸出檔
再一行一行讀，如果是標記就印出來，因為第一階段已經記住標記的位置

vm.c

執行指令：

```
mingw32-make
./vm ../test/Add.bin
```

```
int imTop = 0;
int16_t im[32768], m[65536];
```

im：指令記憶體，在 hack cpu 這台電腦裡面有兩個記憶體

指令放在指令記憶體，指令記憶體最大道 32768

資料放在資料記憶體，資料記憶體基本上也是 32768 就夠了，但為了容易擴充，設為 65536

```
// run: ./vm <file.bin>
int main(int argc, char *argv[]) {
    char *binFileName = argv[1];
    FILE *binFile = fopen(binFileName, "rb");
    imTop = fread(im, sizeof(uint16_t), 32768, binFile);
    fclose(binFile);
    run(im, m);
}
```

先把指定的檔案打開讀進來，binFileName 輸入檔就是 bin 檔

argv[1]:取得第一個參數

im:打開之後讀進來，讀到指令記憶體裡面

fclose(binFile); :讀完就關閉檔案

run(im, m); : 把指令記憶體的程式從 0 開始跑

***run 做的事情：模擬機器的執行過程**

```
int run(uint16_t *im, int16_t *m) {  
    int16_t D = 0, A = 0, PC = 0;  
    uint16_t I = 0;  
    uint16_t a, c, d, j;
```

在 hackcpu 裡有一個程式計數器 pc、主要儲存位置的 A 暫存器、儲存資料的 D 暫存器。

PC = 0 :從第 0 個指令開始執行

***指令執行到超過程式大小的時候會跳出一個無窮迴圈 while(1)**

```
while (1) {  
    int16_t aluOut = 0, AM = 0;  
    if (PC >= imTop) {  
        debug("exit program !\n");  
        break;  
    }  
}
```

只有虛擬機才知道超過程式的範圍！

```
I = im[PC]; //指令記憶體目前的指令把它取出來放到 I 裡面  
debug("PC=%04X I=%04X", PC, I); //印出現在的 PC 跟 I 是多少(影片中的 debug  
寫的是 printf)  
PC ++; //取完之後 PC +1  
if ((I & 0x8000) == 0) { // A 指令 檢查 A 指令的第一碼是不是 0  
A = I; } //如果指令的第一碼是 0，就直接把 A 設成 0  
else { // 如果指令的第一碼是 1，就是 C 指令  
    a = (I & 0x1000) >> 12;  
    c = (I & 0x0FC0) >> 6;  
    d = (I & 0x0038) >> 3;  
    j = (I & 0x0007) >> 0;
```

a = (I & 0x1000) >> 12 :

取出 A 欄位，往右移 12 個，就會把它移到最右邊從 1 開始的地方，以此類推

vm 的指令欄位提取

格式：111 a c1..c6 d1..d3 j1..j3

	a	c	d	j
	111a	1234	5612	3123
0x1000	0001	0000	0000	000a
0x0FC0	0000	1111	1100	0000
0x0038	0000	0000	0011	1000
0x0007	0000	0000	0000	0111

變成：12 == 0000 0000 0000 000a

6 == 0000 0000 00 c1 .. c6

3 == 0000 0000 0000 0 d1 .. d3

0 == 0000 0000 0000 0 j1 .. j3

```
AM = (a == 0) ? A : m[A];
switch (c) { // 處理 c1..6, 計算 aluOut
    case 0x2A: aluOut = 0;          break; // "0",    "101010"
              .
              .
              .
    case 0x15: aluOut = D | AM; break; // "D|AM", "010101"
    default: assert(0);
}
```

```
if (d & BIT(2)) A = aluOut;
if (d & BIT(1)) D = aluOut;
if (d & BIT(0)) m[A] = aluOut;
```

如果 d 的第 2 位元是 1，就要寫入 A

如果 d 的第 1 位元是 1，就要寫入 D

如果 d 的第 0 位元是 1，就要寫入 M

透過這樣可以正確的寫入到暫存器或記憶體

*處理跳躍指令，看 alu 的輸出

j 欄位是 0，不管怎樣都不跳

其他就要看 alu 的 out 是大於 0 還是小於 0 決定

```
switch (j) {
    case 0x0: break;          //
```

```

        case 0x1: if (aluOut > 0) PC = A; break; // JGT
                .
                .
                .
case 0x6: if (aluOut <= 0) PC = A; break; // JLE
        case 0x7: PC = A; break;                // JMP
    }

```

如果 j 欄位是 0x7=111，不管怎樣都要跳(把 PC 設成 A)

```
debug(" A=%04X D=%04X m[A]=%04X", A, D, m[A]);
```

把 A,D,M 印出來觀察

03asmVm/gcc/01-add(配合 main.c)

執行指令：

```
gcc main.c add.c -o add
```

```
./add
```

```
gcc -S add.c -o add.s
```

結果：add(5, 8)=13

```
gcc -fverbose-asm -S add.c -o
```

```
add.s :
```

-fverbose-asm 代表要在產生的組合語言裡面產生詳細的格式

-S 是要產生組合語言的意思
所以它會把 add.c 轉換成組合語言 add.s

add.s

```

.text
.globl _add
.def _add; .scl 2; .type 32; .endef
_add: //函數名稱
    pushl    %ebp      #
    movl     %esp, %ebp #,
    subl $16, %esp     #,
    movl     8(%ebp), %eax # a, tmp89  //後面是註解，意思是 8(%ebp)=a，
    取 ebp + 8 的記憶體內容，所以 ebp + 8 = a
    movl     %eax, -4(%ebp) # tmp89, t  // -4(%ebp)=t，取 ebp - 4 的記憶體
    內容，ebp - 4 = t
    movl     12(%ebp), %eax # b, tmp90  // %eax 暫存器，ebp + 12 = b
    movl     %eax, -8(%ebp) # tmp90, x  // ebp - 8 = x
    movl     8(%ebp), %edx # a, tmp91
    movl     12(%ebp), %eax # b, tmp92
    addl %edx, %eax      # tmp91, D.1490
    leave

```

此三行在做堆疊的動作

```
ret  
.ident    "GCC: (tdm-1) 5.1.0"
```

add.c：把 a 和 b 相加回傳

```
int add(int a, int b) {  
    int t = a, x=b; //臨時變數  
    return a+b;  
}
```