COMSM1302 電腦架構概論

Week 8 作業：A Hack Assembler 完整詳解

用 C 寫一個 Hack 組譯器：符號表 · 詞法分析 · 語法分析 · 機器碼生成

完整實作指南（含所有填空函式的參考實作與測試流程）

2026 年 7 月

本作業的任務：

1. 思考如何用 C 實作符號表（symbol table）；
2. 用 C 寫一個 Hack 詞法分析器（lexer），用提供的測試腳本驗證；
3. 把 lexer 擴充成完整的組譯器（assembler），再驗證。

需要的軟體：文字比對工具（Windows 的 `fc`、Linux/Mac 的 `diff`、或圖形化的 Meld）＋一個 C 編譯器。課程提供 `symboltable.h/.c`、`token.h/.c` 與 `main.c` 骨架——我們的工作是把骨架裡的空缺填滿。

目錄

1 全局視角：兩趟式組譯器 **3**

2 符號表（Task 1） **3**

3 權杖與 union（token.h） **5**

4 詞法分析：三個填空函式 **6**

4.1 lex_line：空白、註解與 ROM 位址 6

4.2 lex_token：關鍵字與識別字 7

4.3 lex_label 與 lexer 測試 9

5 語法分析與機器碼生成 **10**

5.1 骨架導讀與 A-指令偵測 10

5.2 parse_a_instruction：三種運算元 10

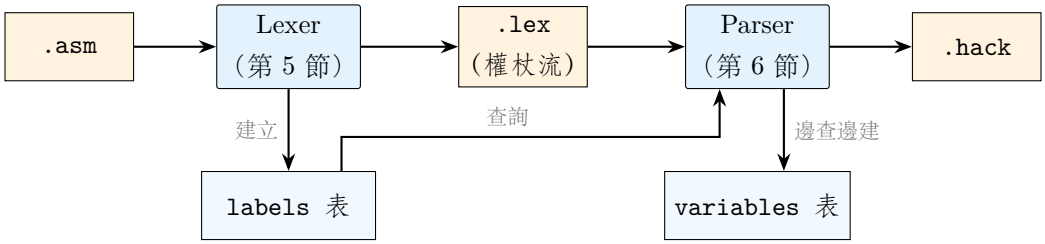
5.3 parse_c_instruction：切出 dest ／ comp ／ jump 11

5.4 parse_c_dest 與 parse_c_jump 12
5.5 parse_c_comp：查表的藝術 13
5.6 組裝與整體測試 15

A 附錄：速查表 **16**
A.1 常見錯誤型錄 16
A.2 C-指令編碼總表 16
A.3 Hack 詞法規格 17
A.4 預定義符號 → 位址 17
A.5 參考資料 17

1　全局視角：兩趟式組譯器

動工前先看清楚整條生產線。本作業的組譯器分**兩趟（two-pass）**：



- **第一趟（lexer）**：把原始碼切成**權杖（tokens）**寫進中繼檔，同時把所有**標籤**（LOOP）記進 labels 符號表——標籤可以**先用後宣告**（向前引用），所以必須先掃完全檔才能解析；
- **第二趟（parser）**：逐條指令讀回權杖、查 labels 表、邊遇到新變數邊填 variables 表，輸出 16 位元機器碼字串。

為什麼標籤和變數要分開兩張表？因為它們的位址空間不同：標籤住在 **ROM**（指令位址），變數住在 **RAM**（從 16 號開始配置）——同名時語意完全不同，混在一張表會出錯。

2　符號表（Task 1）

題目

花幾分鐘思考：`symboltable.h` 宣告的介面（圖 1）你會怎麼實作？

```
struct TableEntry {          // 一筆：名字 + 整數位址
    char *name;
    int address;
}; typedef struct TableEntry TableEntry;

struct SymbolTable {         // 表：entry 指標的動態陣列
    TableEntry **table_array;
    int table_length;        // 已存筆數
    int table_space;         // 內部用：table_array 已配置容量
}; typedef struct SymbolTable SymbolTable;

SymbolTable *malloc_table();                    // 建立空表
void free_table(SymbolTable *table);            // 釋放整張表
void add_to_table(SymbolTable *table,
                  const char *name, int address);
// 找到回傳索引 i（table->table_array[i]），找不到回傳 -1
int get_table_entry(const SymbolTable *table,
                    const char *search_name);
```

解答

設計判讀：這是「**指標的動態陣列 + 線性搜尋**」——C 語言裡最直白的關聯容器。`table_space` 追蹤已配置容量，讓 `add_to_table` 可以**倍增擴容**（攤銷 $O(1)$ 插入）。參考實作：

```
SymbolTable *malloc_table() {
    SymbolTable *t = malloc(sizeof(SymbolTable));
    t->table_length = 0;
    t->table_space  = 8;   // 初始容量隨意，8 或 16 都行
    t->table_array  = malloc(t->table_space * sizeof(TableEntry *));
    return t;
}

void add_to_table(SymbolTable *t, const char *name, int address) {
    if (t->table_length == t->table_space) {   // 滿了就倍增
        t->table_space *= 2;
        t->table_array = realloc(t->table_array,
                                 t->table_space * sizeof(TableEntry *));
    }
    TableEntry *e = malloc(sizeof(TableEntry));
    e->name = malloc(strlen(name) + 1);   // 必須「複製」名字!
    strcpy(e->name, name);
    e->address = address;
    t->table_array[t->table_length++] = e;
}

int get_table_entry(const SymbolTable *t, const char *search_name) {
    for (int i = 0; i < t->table_length; i++)
        if (strcmp(t->table_array[i]->name, search_name) == 0)
            return i;
    return -1;    // 找不到
}

void free_table(SymbolTable *t) {
    for (int i = 0; i < t->table_length; i++) {
        free(t->table_array[i]->name);    // 先釋放名字
        free(t->table_array[i]);          // 再釋放 entry
    }
    free(t->table_array);                 // 再釋放指標陣列
    free(t);                              // 最後釋放表本身
}
```

三個值得咀嚼的細節：

1. **為什麼要複製 `name`？** 呼叫端傳進來的字串常常是行緩衝區裡的暫時內容，下一行就被覆寫——只存指標會留下懸空引用。這是 C 字串處理的經典陷阱。

2. **比較用 strcmp，不能用 ==**：== 比的是指標位址，不是字串內容。

3. **釋放順序由內而外**：name → entry → array → table，順序反了就是 use-after-free。

為什麼課程直接提供這份程式碼？ 因為它「是很好的 C 練習，但不是架構課該花時間的地方」——多數語言有內建型別（Java 的 HashMap、Python 的 dict）。注意提供的版本查詢是 $O(n)$ 線性掃描；真正的雜湊表（hash table）平均 $O(1)$。對本作業的規模（連 25000 行的 pong.asm 也只有幾百個符號）線性掃描完全夠用。

3　權杖與 union（token.h）

題目

讀懂 `token.h/.c`：每個 Token 有一個型態（`SYMBOL`、`KEYWORD`、`INTEGER_LITERAL`、`IDENTIFIER`、`NEWLINE`）與一個值，值存放在 `union` 裡（圖 2）：

```
union TokenData {
    Keyword key_val;   // 關鍵字（enum）
    int     int_val;   // 整數字面值
    char    char_val;  // 符號（單一字元）
    char   *str_val;   // 識別字（字串）
}; typedef union TokenData TokenData;
```

union 是什麼？為什麼這裡適合用它？

解答

union ＝ 同一塊記憶體的多種讀法。 struct 的欄位**並排**存放（大小 ≈ 各欄位總和）；union 的欄位**重疊**在同一位置（大小 ＝ 最大欄位）。寫入 `key_val` 會覆蓋 `int_val` 的內容——同一時刻只有一個欄位「有效」。

圖 2 的使用範例正好展示這種「同一位元組、不同眼鏡」的效果：

```
data.int_val = 42;   printf("%d", data.int_val);   // 印 42
data.char_val = '@'; printf("%c", data.char_val);  // 印 @
data.char_val = '@'; printf("%d", data.int_val);   // 印 64!
```

最後一行：寫進去的是字元 `'@'`（一個位元組，ASCII 值 64），用 `int` 的眼鏡去讀同一塊記憶體，讀到的就是 64——**union 不做任何轉換，只是原封不動的位元圖案。**

哪個欄位有效由誰記錄？ union 本身不記——這正是 Token struct 需要 `TokenType` enum 的原因：

TokenType	有效欄位	例
SYMBOL	char_val	@ = ; + - & | !
KEYWORD	key_val	A D M、JGT…JMP、R0–R15、SP LCL ARG THIS THAT SCREEN KBD
INTEGER_LITERAL	int_val	431
IDENTIFIER	str_val	BANANA、loop
NEWLINE	（無）	行尾

「type 標籤 + union」這個組合叫 **tagged union**（帶標籤聯合），是 C 模擬「多型值」的標準手法；Rust 的 enum、Haskell 的代數資料型別是它的安全版。

`token.c` 另外提供：建立新權杖、釋放舊權杖、把權杖寫進檔案（半人類可讀格式）、從檔案讀回權杖——lexer 與 parser 之間的中繼檔就靠這組 `write_token` ／ read 函式溝通。

4　詞法分析：三個填空函式

骨架的分工（由外而內）：`lex_file`（已給，逐行讀檔）→ `lex_line`（半給，掃一行）→ `lex_token` ／ `lex_label`（處理單一權杖／標籤）。

4.1　lex_line：空白、註解與 ROM 位址

題目

`lex_line(line, rom_address, output, labels)` 掃描字串 `line`：遇到權杖呼叫 `lex_token`、遇到標籤呼叫 `lex_label`（骨架已給）。請補上：(1) 空白處理；(2) 註解處理；(3) 更新 ROM 位址。提醒：只含標籤／註解／空白的行**不**對應機器碼；其他行恰對應**一行**機器碼。

解答

核心邏輯（骨架變數名可能略有出入，邏輯不變）：

```
void lex_line(const char *line, int *rom_address,
              FILE *output, SymbolTable *labels) {
    int i = 0;
    bool emitted = false;          // 這一行有沒有吐出真正的權杖?
    while (line[i] != '\0' && line[i] != '\n') {
        // --- 填空 1: 空白直接跳過 ---
        if (line[i] == ' ' || line[i] == '\t' || line[i] == '\r') {
            i++;
            continue;
        }
        // --- 填空 2: // 之後整行都是註解, 直接收工 ---
        if (line[i] == '/' && line[i+1] == '/')
            break;
        // --- 骨架已給: 標籤與一般權杖 ---
        if (line[i] == '(') {
```

```
            i += lex_label(line + i, *rom_address, labels);
        } else {
            Token *t = ...;                  // 依骨架建立權杖
            i += lex_token(line + i, t);
            write_token(output, t);          // 寫進 .lex 中繼檔
            emitted = true;
        }
    }
    if (emitted) {
        write_token(output, newline_token);  // 指令以換行權杖收尾
        (*rom_address)++;                    // --- 填空 3 ---
    }
}
```

為什麼 ROM 位址只在 `emitted` 時遞增？ 這是整個 lexer 最重要的不變量：`*rom_address` 必須永遠等於「**下一條真指令**會落在 ROM 的哪一格」。標籤行、註解行、空白行在機器碼裡**不存在**，如果它們也遞增位址，之後每個標籤都會偏移，所有跳躍全部跳錯——這是本作業最經典的 bug（症狀：max.asm 開始出錯，而無標籤的 add.asm 正常）。
同一個理由解釋了為什麼 `lex_label` 拿到的是 **當前的** `rom_address`：標籤指向「它**之後**第一條真指令」的位址，而掃到標籤的當下，`rom_address` 恰好就是這個值。

4.2　lex_token：關鍵字與識別字

題目

`lex_token(line, dest)` 要辨認 `line` 開頭的第一個權杖、寫進 `dest`、回傳權杖長度（例如 `@` 回傳 1）。換行、符號、整數字面值已給，請補上**關鍵字**與**識別字**。測試案例：`@431`、`@THAT`（THAT 是關鍵字）、`@R13`（R13 是關鍵字）、`@BANANA`、`@APPLE`（**不可**把 A 切成關鍵字）、`@AD`（**不可**切成 A、D 兩個關鍵字）、`0`、`0;JMP`、`D=0;JLT`、`D=0`、`AD=D-M;JNE`。

解答

關鍵原則：最長吞法（maximal munch）。遇到字母**不能**逐字元判斷「A 是不是關鍵字」——必須先把**整段連續的「單字」**讀完，再拿完整的單字去關鍵字表查：查到 → 關鍵字權杖；查不到 → 識別字權杖。這一刀就同時解決 `@APPLE`（整字 APPLE 不在表裡 → 識別字）與 `@AD`（整字 AD 不在表裡 → 識別字，不會被切成 A、D）。

```
// 講義給的關鍵字集合（順序須與 token.h 的 Keyword enum 一致）
static const char *KEYWORDS[] = {
    "A", "D", "M",
    "JGT", "JEQ", "JLT", "JGE", "JNE", "JLE", "JMP",
    "SP", "LCL", "ARG", "THIS", "THAT", "SCREEN", "KBD",
    "R0","R1","R2","R3","R4","R5","R6","R7",
    "R8","R9","R10","R11","R12","R13","R14","R15",
};
```

```c
#define NUM_KEYWORDS (sizeof KEYWORDS / sizeof KEYWORDS[0])

// 識別字的合法字元：字母數字加 _ . $ :（pong.asm 會用到！）
static bool is_word_char(char c) {
    return isalnum((unsigned char)c) ||
           c == '_' || c == '.' || c == '$' || c == ':';
}

int lex_token(const char *line, Token *dest) {
    // ...（已給）換行、符號 @=;+-&|!、整數字面值 ...

    // --- 填空：關鍵字與識別字 ---
    if (isalpha((unsigned char)line[0]) || line[0] == '_') {
        int len = 0;                          // 1. 最長吞法：先讀整個單字
        while (is_word_char(line[len]))
            len++;
        char word[MAX_TOKEN_LENGTH];
        strncpy(word, line, len);
        word[len] = '\0';
        for (int k = 0; k < NUM_KEYWORDS; k++)   // 2. 查關鍵字表
            if (strcmp(word, KEYWORDS[k]) == 0) {
                *dest = /* 依骨架建立 KEYWORD 權杖，key_val = k */;
                return len;
            }
        *dest = /* 依骨架建立 IDENTIFIER 權杖（複製 word！） */;
        return len;                           // 3. 都不是 -> 識別字
    }
    ...
}
```

逐一驗證題目給的測試案例（權杖流；Nl = 換行權杖）：

輸入	預期權杖流
`@431`	Sym(@)、Int(431)、Nl
`@THAT`	Sym(@)、**Kw**(THAT)、Nl
`@R13`	Sym(@)、**Kw**(R13)、Nl （R13 整字在表裡）
`@BANANA`	Sym(@)、Id(BANANA)、Nl
`@APPLE`	Sym(@)、Id(APPLE)、Nl （✗ Kw(A)+Id(PPLE)）
`@AD`	Sym(@)、Id(AD)、Nl （✗ Kw(A)+Kw(D)）
`0`	Int(0)、Nl
`0;JMP`	Int(0)、Sym(;)、Kw(JMP)、Nl
`D=0;JLT`	Kw(D)、Sym(=)、Int(0)、Sym(;)、Kw(JLT)、Nl
`D=0`	Kw(D)、Sym(=)、Int(0)、Nl
`AD=D-M;JNE`	Id(AD)、Sym(=)、Kw(D)、Sym(-)、Kw(M)、Sym(;)、Kw(JNE)、Nl

注意最後一列的驚喜：**dest 欄位的 AD 是識別字**，不是關鍵字（表裡只有單一的 A、D、M）！lexer 不在乎——它只負責切；讓 AD 產生正確 d 位元是 **parser** 的工作（見 6.4 節，這是個容易踩到的銜接點）。

4.3　lex_label 與 lexer 測試

題目

補完 `lex_label`：把標籤加進符號表（配上正確的 ROM 位址）並回傳。然後用課程提供的四個 Nand2Tetris 測試腳本驗證 lexer 輸出。

解答

lex_label 參考實作（進入時 `line[0]` 必為 `(`）：

```
int lex_label(const char *line, int rom_address,
              SymbolTable *labels) {
    int len = 1;                          // 跳過 '('
    while (line[len] != ')')
        len++;                            // 找到 ')'
    char name[MAX_TOKEN_LENGTH];
    strncpy(name, line + 1, len - 1);  // 取括號中間的名字
    name[len - 1] = '\0';
    add_to_table(labels, name, rom_address);
        // rom_address = 下一條真指令的位址（4.1 節的不變量）
    return len + 1;                       // 連同 ')' 一共吃掉的字元數
}
```

標籤**不產生權杖**——它只在 lexer 的符號表留下痕跡，權杖流裡完全看不到它（這正是講義說「標籤行不佔 ROM 位址」在程式碼層面的體現）。

測試流程（循序漸進，一次只引入一種新難度）：

順序	腳本	測到什麼
1	`add.asm`	最小可行：無標籤、無變數
2	`max.asm`	標籤（`OUTPUT_FIRST` 等）＋ R0–R2 關鍵字
3	`rect.asm`	變數（`counter`、`address`）＋ `SCREEN`
4	`pong.asm`	25000+ 行的壓力測試（高階語言自動生成）

把你的 `.lex` 輸出與課程提供的正確輸出 `diff`（Mac/Linux）、`fc`（Windows）或 Meld 比對——**第一行差異**就是 bug 的準確座標。卡住時用 `max-L.asm`、`rect-L.asm`、`pong-L.asm`（把標籤與識別字都換成數字的版本）先隔離「標籤／識別字處理」以外的問題。

5　語法分析與機器碼生成

5.1　骨架導讀與 A-指令偵測

題目

讀懂 `parse_file`、`get_next_instruction`、`parse_instruction`，然後補上 `parse_instruction` 裡**偵測 A-指令**的邏輯。

解答

骨架的流水線：`parse_file` 反覆用 `get_next_instruction` 從 `.lex` 檔讀回權杖，靠「**指令必以換行權杖結尾**」這個事實切出一條條指令（這就是 lexer 需要 NEWLINE 權杖的原因——它是指令的邊界記號）。每條指令連同 labels 表、variables 表、權杖陣列與長度、輸出檔交給 `parse_instruction`，後者判斷 A 或 C 指令、分派給對應函式，把它們寫進 buffer 的機器碼字串輸出。

A-指令的判準：第一個權杖是符號 `@`——

```
bool is_a_instruction =
    instruction[0].type == SYMBOL &&
    instruction[0].data.char_val == '@';
```

兩個條件缺一不可：只查 `type == SYMBOL` 不夠（`0;JMP` 的第二個權杖 `;` 也是符號），必須同時確認字元值是 `@`；而且要先確認 type 才能讀 `char_val` ——union 的欄位只有在 type 正確時才有意義（第 3 節）。

5.2　parse_a_instruction：三種運算元

題目

補完 `parse_a_instruction`：把 A-指令的運算元載入 `value_to_load`，函式尾端的 `int_to_bin_string` 會把它轉成 16 位元的 0/1 字串放進 `dest`。

解答

`@` 後面的權杖（`instruction[1]`）有三種可能，逐型態分派：

```
Token operand = instruction[1];
int value_to_load;

switch (operand.type) {
case INTEGER_LITERAL:                    // @431
    value_to_load = operand.data.int_val;
    break;

case KEYWORD:                            // @THAT、@R13、@SCREEN
    value_to_load = keyword_address(operand.data.key_val);
    break;
```

```
case IDENTIFIER: {                          // @LOOP 或 @counter
    int i = get_table_entry(labels, operand.data.str_val);
    if (i != -1) {                          // (1) 先查標籤表
        value_to_load = labels->table_array[i]->address;
        break;
    }
    i = get_table_entry(variables, operand.data.str_val);
    if (i != -1) {                          // (2) 再查已知變數
        value_to_load = variables->table_array[i]->address;
        break;
    }
    value_to_load = 16 + variables->table_length;
    add_to_table(variables, operand.data.str_val,
                 value_to_load);            // (3) 新變數: 16 起依序配
    break;
}
}
int_to_bin_string(value_to_load, dest, 16);   // 骨架已給
```

關鍵字 → 位址是一張固定的小表（對應 nand2tetris 的預定義符號）：

關鍵字	位址	關鍵字	位址
SP	0	R0–R15	0–15
LCL	1	SCREEN	16384
ARG	2	KBD	24576
THIS	3		
THAT	4		

查詢順序是規格的一部分：標籤表**必須**先查。若程式裡有 (END) 標籤，@END 應該解析成 ROM 位址；只有「既不是預定義關鍵字、也不是任何標籤」的名字才是變數，從 RAM 16 號開始**依首次出現順序**配置。順序查反的症狀：rect.asm 還對，pong.asm（標籤與變數混用）大片錯。（新變數位址用 16 + table_length 之所以成立，是因為 variables 表裡**只有**變數——這正是 labels 和 variables 必須分成兩張表的另一個理由。）

5.3　parse_c_instruction：切出 dest ／ comp ／ jump

題目

parse_c_instruction（已給）把工作拆給 parse_c_dest、parse_c_comp、parse_c_jump，再把三段二進位字串接成一條機器碼。理解它如何切分 dest=comp;jump。

解答

C-指令的文法是 `[dest=]comp[;jump]`——兩個可省略的部分由**分隔符號**定界：掃一遍權杖陣列找 `=` 和 `;` 的位置，

- 有 `=`：它**之前**的權杖是 dest；沒有 `=`：dest 為空（$d = 000$）；
- 有 `;`：它**之後**的權杖是 jump；沒有 `;`：jump 為空（$j = 000$）；
- 夾在中間的就是 comp（**必定存在**）。

對照四個例子（Nl 前的權杖）：

指令	dest	comp	jump
`0;JMP`	—	`0`	`JMP`
`D=0`	`D`	`0`	—
`D=0;JLT`	`D`	`0`	`JLT`
`AD=D-M;JNE`	`AD`	`D`、`-`、`M`	`JNE`

輸出格式：`111` + $ac_1..c_6$（7 位元，comp）+ $d_1d_2d_3$（dest）+ $j_1j_2j_3$（jump）。題目特別約定（配合測試資料）：comp 不含 A ／ M 時 a 位**必須**為 0，且第 14、13 位元恆為 1——所以 `D;JMP` 必須輸出 `1110001100000111` 而非 `1001001100000111`（兩者硬體行為相同，但 diff 比對是逐字元的）。

5.4　parse_c_dest 與 parse_c_jump

題目

補完 `parse_c_dest` 與 `parse_c_jump`。

解答

jump 最簡單：分號後恰好一個關鍵字，查表即得（j_1：< 0 跳、j_2：$= 0$ 跳、j_3：> 0 跳）：

jump	j_1	j_2	j_3	jump	j_1	j_2	j_3
（無）	0	0	0	JLT	1	0	0
JGT	0	0	1	JNE	1	0	1
JEQ	0	1	0	JLE	1	1	0
JGE	0	1	1	JMP	1	1	1

```
// 無 jump 部分時輸出 "000"
void parse_c_jump(Token *jump_token, char *dest) {
    if (jump_token == NULL) { strcpy(dest, "000"); return; }
    switch (jump_token->data.key_val) {
    case KW_JGT: strcpy(dest, "001"); break;
    case KW_JEQ: strcpy(dest, "010"); break;
    case KW_JGE: strcpy(dest, "011"); break;
```

```
    case KW_JLT: strcpy(dest, "100"); break;
    case KW_JNE: strcpy(dest, "101"); break;
    case KW_JLE: strcpy(dest, "110"); break;
    case KW_JMP: strcpy(dest, "111"); break;
    }
}
```

dest 有個陷阱（5.2 節埋的伏筆）：dest 欄位可能是**關鍵字**權杖（`D=`、`M=`、`A=`）也可能是**識別字**權杖（`AD=`、`MD=`、`AMD=`——多字母組合不在關鍵字表裡！）。穩健的做法是不管哪種，都化成「**含哪些字母**」：d_1 = 含 A、d_2 = 含 D、d_3 = 含 M：

```
void parse_c_dest(Token *dest_token, char *dest) {
    bool a = false, d = false, m = false;
    if (dest_token != NULL) {
        if (dest_token->type == KEYWORD) {        // A= / D= / M=
            a = (dest_token->data.key_val == KW_A);
            d = (dest_token->data.key_val == KW_D);
            m = (dest_token->data.key_val == KW_M);
        } else {                                  // AD= / MD= / AMD=
            const char *s = dest_token->data.str_val;
            a = (strchr(s, 'A') != NULL);
            d = (strchr(s, 'D') != NULL);
            m = (strchr(s, 'M') != NULL);
        }
    }
    dest[0] = a ? '1' : '0';    // d1
    dest[1] = d ? '1' : '0';    // d2
    dest[2] = m ? '1' : '0';    // d3
    dest[3] = '\0';
}
```

好處是順帶支援 `DA=`、`DM=` 等任意字母順序（合法組譯器本就該接受）。

5.5　parse_c_comp：查表的藝術

題目

補完 `parse_c_comp` 剩下的部分（寫的時候把講義的指令集攤在旁邊！）。

解答

comp 欄位是 1–3 個權杖的小運算式。最省力也最不易錯的做法：**把權杖串回文字，拿整個字串查一張 28 列的表**，直接得到 $ac_1..c_6$ 七個位元：

```
void parse_c_comp(Token *comp_tokens, int n, char *dest) {
    char text[8] = "";                    // 1. 權杖串回文字
```

```
    for (int i = 0; i < n; i++) {
        switch (comp_tokens[i].type) {
        case KEYWORD:
            strcat(text, keyword_string(comp_tokens[i].data.key_val));
            break;                          // "A" / "D" / "M"
        case SYMBOL: {
            char s[2] = { comp_tokens[i].data.char_val, '\0' };
            strcat(text, s);                // "+" "-" "!" "&" "|"
            break;
        }
        case INTEGER_LITERAL:               // comp 裡只會有 0 或 1
            strcat(text, comp_tokens[i].data.int_val ? "1" : "0");
            break;
        }
    }
    static const struct { const char *comp; const char *bits; }
    COMP_TABLE[] = {                        // 2. 查表（a + c1..c6）
        {"0",  "0101010"}, {"1",  "0111111"}, {"-1", "0111010"},
        {"D",  "0001100"}, {"A",  "0110000"}, {"M",  "1110000"},
        {"!D", "0001101"}, {"!A", "0110001"}, {"!M", "1110001"},
        {"-D", "0001111"}, {"-A", "0110011"}, {"-M", "1110011"},
        {"D+1","0011111"}, {"A+1","0110111"}, {"M+1","1110111"},
        {"D-1","0001110"}, {"A-1","0110010"}, {"M-1","1110010"},
        {"D+A","0000010"}, {"D+M","1000010"},
        {"D-A","0010011"}, {"D-M","1010011"},
        {"A-D","0000111"}, {"M-D","1000111"},
        {"D&A","0000000"}, {"D&M","1000000"},
        {"D|A","0010101"}, {"D|M","1010101"},
    };
    for (size_t i = 0; i < sizeof COMP_TABLE / sizeof *COMP_TABLE; i++)
        if (strcmp(text, COMP_TABLE[i].comp) == 0) {
            strcpy(dest, COMP_TABLE[i].bits);
            return;
        }
}
```

表的內容直接來自講義的指令集：$c_1..c_6$ 就是 ALU 的 zx, nx, zy, ny, f, no，a 位選擇 ALU 的 y 輸入是 A（$a=0$）還是 M（$a=1$）——所以「M 系列」的位元圖案與「A 系列」**完全相同、只差 a 位**，這是驗表時的好用檢查。表也自動滿足題目的約定：不含 A ／ M 的 comp （0、1、D 等）a 位都是 0。

為什麼不「聰明地」分別解析運算子和運算元？因為 comp 只有 28 種合法組合，窮舉表**就是**最清晰的規格書——真實組譯器（如 nand2tetris 官方實作）也是這樣寫的。想省打字，可以利用上面的觀察只存 A 系列，遇 M 時把 a 改成 1、文字裡 M 換成 A 再查。

5.6　組裝與整體測試

題目

把 `main.c` 裡呼叫 parser 的程式碼解除註解，用四個測試腳本的 `.hack` 正確輸出驗證組譯器。

解答

完整測試矩陣（lexer 通過不代表 parser 通過，每階段重測全部四支）：

腳本	新增考點	典型翻車處
`add.asm`	純 A/C 指令	comp 表抄錯、`111` 前綴忘了
`max.asm`	標籤跳躍	標籤位址偏移（4.1 節不變量被破壞）
`rect.asm`	變數配置	變數沒從 16 開始、重複配置
`pong.asm`	規模 + 全特性	查表順序（標籤 vs 變數）、效能

驗證指令（Mac/Linux）：

```
./assembler add.asm          # 產生 add.hack
diff add.hack add-expected.hack && echo PASS
```

`diff` 安靜無輸出就是全對；有輸出時，**第一個**差異行號 n 就是第 n 條指令錯——拿它對照 `.asm`（記得跳過註解行）可直接定位。若整段成批錯位，幾乎必是 ROM 位址不變量的問題；若只有零星幾行錯，通常是 comp ／ dest ／ jump 表的個別筆誤。

進階討論

你剛寫完的其實是一個「編譯器前端」的縮影。 lexer（正規語言、最長吞法）+ parser（線性文法、查表生成碼）+ 兩張符號表（作用域雛形）——這條流水線放大一百倍就是 GCC 的前端。幾個往外延伸的方向：（1）**錯誤處理**：本作業刻意跳過；真組譯器要報告「第 17 行：未知的 comp `D+D`」而不是默默輸出垃圾——這需要在每個查表失敗處插入診斷路徑。（2）**效能**：線性符號表讓 pong.asm 的第二趟是 $O(nm)$（n 指令數、m 符號數）；換成雜湊表降到 $O(n)$。實務上更常見的優化反而是**避免中繼檔**——權杖直接留在記憶體裡傳給 parser。（3）**單趟可行嗎？**變數可以單趟解決（首次出現就配置），但**標籤不行**：`@END` 可能出現在 `(END)` 之前（向前引用）。單趟組譯器得用「回填（backpatching）」——先留洞、記下待補清單、看到標籤宣告再回頭補——複雜度遠高於乾脆掃兩趟。這就是幾乎所有組譯器都是兩趟式的原因。

A 附錄：速查表

A.1 常見錯誤型錄

症狀	病因
`@APPLE` 被切成 `A` + `PPLE`	沒用最長吞法——先讀完整個單字再查關鍵字表
`add.asm` 對、`max.asm` 起錯，且錯的行成批偏移	標籤／註解／空白行也遞增了 ROM 位址
`AD=...` 的 d 位元是 000	parser 只處理了 KEYWORD 型的 dest，漏了 IDENTIFIER 型（`AD` 不是關鍵字!）
`D;JMP` 輸出 `1001...` 被 diff 判錯	第 14、13 位元沒有依約定設成 1（前綴恆為 `111`）
`@LOOP` 解析成 RAM 位址	查表順序錯——必須**先查標籤表**再查／配變數
變數位址從 0 或 17 開始	首個變數必須是 16；用 `16 + variables->table_length` 配置
同一變數兩次出現拿到不同位址	配置前忘了先查 variables 表
符號表存的名字變成亂碼	`add_to_table` 沒複製字串，存了指向行緩衝區的懸空指標
pong.asm 輸出對但跑很久	正常（線性符號表 $O(nm)$）；想加速可換雜湊表

A.2 C-指令編碼總表

格式： `111` $a\,c_1c_2c_3c_4c_5c_6\,d_1d_2d_3\,j_1j_2j_3$

a=0	a=1	c_1	c_2	c_3	c_4	c_5	c_6
`0`		1	0	1	0	1	0
`1`		1	1	1	1	1	1
`-1`		1	1	1	0	1	0
`D`		0	0	1	1	0	0
`A`	`M`	1	1	0	0	0	0
`!D`		0	0	1	1	0	1
`!A`	`!M`	1	1	0	0	0	1
`-D`		0	0	1	1	1	1
`-A`	`-M`	1	1	0	0	1	1
`D+1`		0	1	1	1	1	1
`A+1`	`M+1`	1	1	0	1	1	1
`D-1`		0	0	1	1	1	0
`A-1`	`M-1`	1	1	0	0	1	0
`D+A`	`D+M`	0	0	0	0	1	0
`D-A`	`D-M`	0	1	0	0	1	1
`A-D`	`M-D`	0	0	0	1	1	1
`D&A`	`D&M`	0	0	0	0	0	0
`D|A`	`D|M`	0	1	0	1	0	1

dest	d_1	d_2	d_3
—	0	0	0
`M=`	0	0	1
`D=`	0	1	0
`DM=`	0	1	1
`A=`	1	0	0
`AM=`	1	0	1
`AD=`	1	1	0
`ADM=`	1	1	1

jump	j_1	j_2	j_3
—	0	0	0
`JGT`	0	0	1
`JEQ`	0	1	0
`JGE`	0	1	1
`JLT`	1	0	0
`JNE`	1	0	1
`JLE`	1	1	0
`JMP`	1	1	1

A.3 Hack 詞法規格

權杖類別	內容
關鍵字	`A D M; JGT JEQ JLT JGE JNE JLE JMP; SP LCL ARG THIS THAT SCREEN KBD; R0–R15`
符號	`@ + - & | = ; !`
整數字面值	0…32767 的十進位整數
識別字	不含空白、非關鍵字、字母開頭（可含 `_ . $ :`）
換行	指令邊界記號

A.4 預定義符號 → 位址

符號	位址	符號	位址	符號	位址
`SP`	0	`THAT`	4	`SCREEN`	16384
`LCL`	1	`R0–R15`	0–15	`KBD`	24576
`ARG`	2	新變數	16 起依序	標籤	ROM 位址
`THIS`	3				

A.5 參考資料

- COMSM1302 第八週講義與影片（編譯器階段、詞法分析、符號表、剖析），University of Bristol.
- Nisan & Schocken, *The Elements of Computing Systems*（nand2tetris），Ch. 6（Assembler）——測試腳本 add/max/rect/pong 的出處與兩趟式組譯器規格.
- nand2tetris Project 6（nand2tetris.org/project06）——`Prog.asm` 與 `ProgL.asm`（去符號版）的官方說明.
- C11 標準 §6.7.2.1（union 的儲存重疊語意）；K&R *The C Programming Language* Ch. 6.