COMSM1302 電腦架構概論

Week 11 作業：Compiling Jack 完整詳解

從 Jack 到 Hack VM：XML 解析樹 · 遞迴下降解析 · 符號表 · 程式碼產生

完整實作指南（parse_*** 與 compile_*** 全家的參考做法）

2026 年 7 月

本作業的任務：

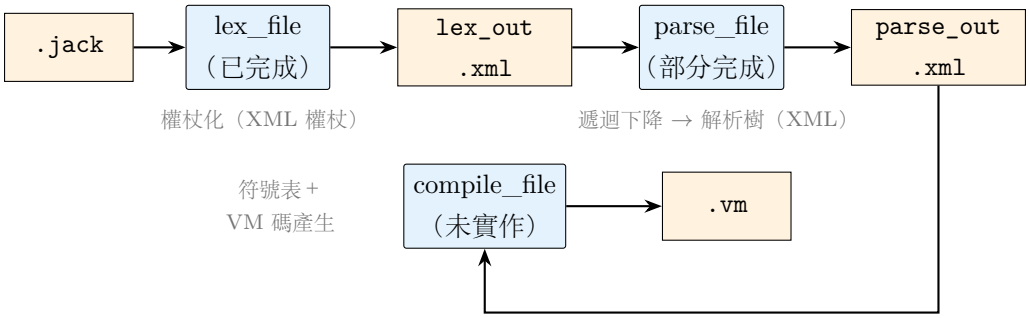
1. 把提供的骨架擴充成完整的 Jack **解析器（parser）**；
2. 再把解析器擴充成 Jack → Hack VM 的**編譯器**；
3. （選做！）與你的 Hack VM 翻譯器、Hack 組譯器串起來，完成 Jack → Hack 機器碼的**完整編譯管線**。

需要的軟體：文字比對工具（`diff` ／ `fc` ／ Meld）；測試用 nand2tetris 的 VM 模擬器。這是整個單元的壓軸——三個週次的工具在此合體。

目錄

1　全局視角：編譯管線

骨架的 `main` 把工作切成三段（第一個命令列參數是輸入檔、第二個是輸出檔）：



Part 1（第 3–4 節）補完中間的解析器；**Part 2**（第 5 節）實作最後的碼產生。兩趟之間的解析樹以 XML 檔存放——人類可讀（除錯友善）、格式標準、可以邊讀邊處理而不必整棵樹載入記憶體。

2　基礎設施：tag.c 與 tag.h

題目

讀懂 `tag.c` ／ `tag.h`（由第 8–10 週的 `token.c/.h` 改造而來）：權杖如何以 XML 標籤對存放？非終端符號如何表示？三個解析輔助函式各做什麼？

解答

(1) 權杖 → XML 標籤對。每個權杖存成 `<` 型態 `>` 內容 `</`型態 `>`，例如關鍵字 `else` 存成 `<keyword> else </keyword>`。術語：`<keyword>` 是**開標籤**、`</keyword>` 是**閉標籤**（以 `/` 開頭）、夾在中間的是**內容**。四個特殊字元不能直接放進內容，必須**跳脫**：

字元	跳脫序列	字元	跳脫序列
`<`	`&lt;`	`&`	`&amp;`
`>`	`&gt;`	`"`	`&quot;`

（題目原文把 `<` ／ `>` 的跳脫寫反了順序——XML 標準是 `<` → `&lt;`、`>` → `&gt;`，容易自行對照確認。為什麼需要跳脫？因為 `<` 在 XML 裡是標籤的開始記號，出現在內容裡會讓讀取器誤判結構——例如 Jack 運算子 `<` 就必須存成 `<symbol> &lt; </symbol>`。）

(2) Tag struct——權杖與非終端符號的統一容器：

- `type`（enum `JackTagType`）+ `data`（union `TagData`）——與第 8 週的 Token 同構；
- **非終端符號**有專屬的 type：`NON_TERMINAL`，此時 data 存 enum `NonTerminal`（class、whileStatement⋯）；
- `close_tag` 欄位：非終端的**閉**標籤為 true、開標籤為 false（權杖的標籤對用不到此欄位）——解析樹的巢狀結構就靠開閉標籤對表達，所有子節點都寫在開閉標籤之間；

- `malloc_tag` ／ `free_tag`（記憶體安全的建立與釋放）、`read_tag` ／ `write_tag`（讀寫 XML，非終端也適用，還會自動處理縮排）。

(3) 三個解析輔助函式（配合影片 11-2 的 `current` ／ `lookahead` 雙指標）：

函式	行為
`advance_tag`	從輸入讀下一個權杖：`current` ← `lookahead`、`lookahead` ← 新權杖（**丟棄**原 current，不寫輸出）
`copy_tag`	把 `current` 原樣寫到輸出，再呼叫 `advance_tag`（解析樹保留該權杖）
`write_non_terminal`	把指定非終端的開或閉標籤寫到輸出；**不動** `current` ／ `lookahead`（它不消耗權杖——非終端標籤是解析器「加上去」的結構）

整個解析器就用這三個動詞寫成：**copy**（這個權杖屬於解析樹）、**advance**（這個權杖只是文法噪音，AST 不需要）、**write_non_terminal**（在正確位置蓋結構章）。

3　Part 1：解析 Jack

3.1　解析器契約與骨架導讀

題目

骨架已給 `parse_file`、`parse_class`、`parse_class_var_dec`。為文法中其餘非終端符號各寫一個 `parse_***` 函式（⟨*type*⟩ 除外），簽名一律是 `(current, lookahead, input, output)`。

解答

每個 `parse_abc` 都遵守同一份契約（影片 11-2）：

- 進入時 `current` 是 ⟨*abc*⟩ 的 **第一個**權杖；
- 產生 ⟨*abc*⟩ 的全部 XML（開閉標籤 + 子節點）；
- 返回時 `current` 是 ⟨*abc*⟩ **結束後的第一個**權杖。

`parse_file` 的流程：建立 `current` ／ `lookahead` → 用一次 `advance_tag` 跳過檔案開頭為符合 XML 標準而存在的特殊 `<tokens>` 標籤 → 呼叫 `parse_class`（一個 `.jack` 檔就是一個 ⟨*class*⟩）→ 返回時 `current` 應指向檔尾的 `</tokens>`，收尾關檔。

已給的 `parse_class` 是所有函式的模板（文法：⟨*class*⟩ ::= 'class', identifier, '{', {⟨*classVarDec*⟩}, {⟨*subroutineDec*⟩}, '}'）：

```
// 骨架的形狀（示意）
write_non_terminal(CLASS, /*close=*/false, output);
copy_tag(...);   // 'class'
copy_tag(...);   // 類別名（識別字）
copy_tag(...);   // '{'
```

```
while (current 是 'static' 或 'field')
    parse_class_var_dec(...);
while (current 是 'constructor'/'function'/'method')
    parse_subroutine_dec(...);
copy_tag(...);   // '}'
write_non_terminal(CLASS, /*close=*/true, output);
```

「重複 {} 」翻成 while 迴圈、「擇一 | 」翻成 if/switch、「巢狀非終端」翻成遞迴呼叫——文法**就是**程式碼的形狀。

3.2 結構層：subroutineDec 到 varDec

解答

依文法逐一寫（縮排即巢狀關係；「複製」= `copy_tag`）：

parse_subroutine_dec （::= (‘constructor’ | ‘function’ | ‘method’), (‘void’ | ⟨*type*⟩), identifier, ‘(’, ⟨*parameterList*⟩, ‘)’, ⟨*subroutineBody*⟩）：複製 kind 關鍵字 → 複製返回型別（⟨*type*⟩ 不寫標籤，直接複製那個權杖——見下方「抑制規則」）→ 複製函式名 → 複製 `(` → `parse_parameter_list` → 複製 `)` → `parse_subroutine_body`。

parse_parameter_list （::= [⟨*type*⟩, identifier, {‘,’, ⟨*type*⟩, identifier}]）：若 `current` 是 `)`，參數列為空、直接返回；否則複製「型別、名字」，之後只要 `current` 是 `,` 就複製「逗號、型別、名字」。

parse_subroutine_body （::= ‘{’, {⟨*varDec*⟩}, ⟨*statements*⟩, ‘}’）：複製 `{` → 只要 `current` 是 `var` 就 `parse_var_dec` → `parse_statements` → 複製 `}`。

parse_var_dec （::= ‘var’, ⟨*type*⟩, identifier, {‘,’, identifier}, ‘;’）：與已給的 `parse_class_var_dec` 幾乎相同——複製 `var`、型別、名字，`,` 迴圈補名字，最後複製 `;`。

抑制規則（配合測試資料，必須遵守）：

- ⟨*type*⟩ 與 ⟨*parameterList*⟩：**不寫**開閉標籤（⟨*type*⟩ 甚至不需要專屬函式——它永遠恰好一個權杖，複製即可；⟨*parameterList*⟩ 保留函式但別呼叫 `write_non_terminal`）；
- ⟨*subroutineDec*⟩：也**不寫**標籤，但**強烈建議保留 `parse_subroutine_dec` 函式**——碼產生時「進入／離開一個副程式」的邊界就靠它（題目註腳：原版 nand2tetris 假設學生用比 C 舒服的語言、從零寫解析器，標籤位置很難擺對所以乾脆建議別寫；我們有穩健的框架，不受此限）。

3.3　敘述層：五種 statement

解答

parse_statements：分派器——只要 current 是 let ／ if ／ while ／ do ／ return 之一，就呼叫對應函式；遇到別的（實務上是 }）寫閉標籤返回。

函式	動作序列
parse_let_statement	複製 let、變數名；若 current 是 [：複製 [、parse_expression、複製]；複製 =、parse_expression、複製 ；
parse_if_statement	複製 if （、parse_expression、複製 ） {、parse_statements、複製 }；若 current 是 else：複製 else {、parse_statements、複製 }
parse_while_statement	複製 while （、parse_expression、複製 ） {、parse_statements、複製 }
parse_do_statement	複製 do、parse_subroutine_call、複製 ；
parse_return_statement	複製 return；若 current 不是 ；就 parse_expression；複製 ；

每個函式頭尾都夾 write_non_terminal 的開閉標籤（<letStatement>⋯</letStatement> 等）。注意 ⟨*ifStatement*⟩ ／ ⟨*whileStatement*⟩ 透過 ⟨*statements*⟩ **遞迴**——巢狀迴圈免費獲得。

3.4　運算式層：term 與唯一一次 lookahead

解答

parse_expression（::= ⟨*term*⟩, {*op*, ⟨*term*⟩}）：parse_term，然後只要 current 是九個二元運算子（+ - * / & | < > =）之一：複製運算子、再 parse_term。

parse_term——全解析器唯一需要 lookahead 的地方。按 current 分七種情況：

current	動作
整數／字串字面值	複製它，結束
true/false/null/this	複製它，結束
(	複製 （、parse_expression、複製 ）
- 或 ~（一元）	複製運算子、**遞迴 parse_term**
識別字，lookahead 是 (或 .	parse_subroutine_call
識別字，lookahead 是 [	複製名字、[、parse_expression、]
識別字（其他）	複製名字（純變數），結束

為什麼需要看 lookahead？遇到識別字開頭的 ⟨*term*⟩，光看 current 分不出它是變數、陣列存取還是 ⟨*subroutineCall*⟩ 的函式名——要看**下一個**權杖是不是 (／. ／ [。這就是「Jack 是 LL(2)」的全部內容；其他所有函式只看 current 就夠（LL(1)）。

parse_subroutine_call（::= identifier, ['.', identifier], '(', ⟨*expressionList*⟩, ')'）：複製名字；若 current 是 . 再複製 . 與名字；複製 （、parse_expression_list、複製 ）。

parse_expression_list（::= [⟨*expression*⟩, {',', ⟨*expression*⟩}]）：若 current 是) 直接返回

（空列表）；否則 `parse_expression`，之後 `,` 迴圈。

3.5　測試解析器

題目

用 nand2tetris 原版測試資料驗證：`ExpressionLessSquare`、`Square`、`ArrayTest`。每個 `foo.jack` 附兩個對照檔：`foo.xml`（解析器的期望輸出）與 `fooT.xml`（詞法器的期望輸出）。

解答

測試順序（難度遞增）：

測試	考什麼
ExpressionLessSquare	「無運算式」的胡話版 Square——每個運算式都被換成單一識別字、無陣列下標。先隔離**結構層與敘述層**的解析，不碰最難的 ⟨*term*⟩ ／ ⟨*expression*⟩ 遞迴
Square	真版（Nisan & Schocken 第 9 章詳述；本質上是第 5 週 Rogue 練習的 Jack 版）——完整運算式、方法、建構子，仍無陣列下標
ArrayTest	影片 11-1 的範例程式——運算式 + 陣列下標都有

對每個 `foo.jack` 跑你的解析器，把輸出與 `foo.xml` 用 `diff` ／ `fc` ／ Meld 比對。`fooT.xml` 平常用不到——它是詞法器輸出的對照檔，懷疑 bug 在詞法層時先比它排除（詞法器骨架已完整實作，理論上直接過）。

diff 的讀法： XML 每個標籤一行、含縮排，第一個差異行直接指出**哪個非終端的哪個位置**出錯。常見前三名：忘了抑制 ⟨*type*⟩ ／ ⟨*parameterList*⟩ ／ ⟨*subroutineDec*⟩ 標籤；`parse_term` 的識別字情況漏看 `lookahead`；⟨*returnStatement*⟩ 的可選運算式判斷寫反。

4　Part 2：語意分析與程式碼產生

4.1　升級版符號表與 CompileData

題目

讀懂 `symboltable.c/.h` 相對第 8 週的變更，以及 `CompileData` struct 的用途。

解答

TableEntry 的變化（從組譯器的「名字 → 位址」升級成編譯器的完整變數描述）：

欄位	型別	用途
`name`	字串	查詢鍵（不變）
`type`	字串	`int` ／ `char` ／ `boolean` ／類別名——方法呼叫時要知道變數是哪個類別
`kind`	enum `VariableKind`	local ／ argument ／ field ／ static ——決定 VM 記憶體段
`offset`	int	段內編號（舊 `address` 改名）

關鍵行為變更：`offset` 只對同 `kind` 遞增。例如表裡已有 3 筆 VK_VAR 與 10 筆 VK_ARGUMENT，再加一筆 VK_VAR 會拿到 offset = 3（第 4 個 VAR），而不是 14（第 14 筆）。原因：VM 的 `local` 與 `argument` 是**各自獨立**的記憶體段，編號各從 0 起——符號表直接維護這個不變量，碼產生時查到就能用。

`is_primitive`：型別是 `int` ／ `char` ／ `boolean` 回傳 true。用途在編譯 ⟨*subroutineCall*⟩ 時分辨 `foo.bar()` 的 `foo`：是**物件變數**（非 primitive）→ 方法呼叫，要把 `foo` 推成第一個引數；不在表裡 → 類別名，普通 `call`。

`CompileData`：把（類別符號表、副程式符號表、類別名、標籤計數器、輸出檔……）打包成一個 struct，省得每個 `compile_***` 都拖七八個參數——純工程便利，`compile_file` 已初始化好。

符號表的生命週期（Jack 特意設計成 **不需要**獨立的語意分析趟）：

- ⟨*class*⟩ 開標籤：建**類別表**，每個 ⟨*classVarDec*⟩ 加 field ／ static 項目（兩種 kind 分開編號）；閉標籤時釋放；
- ⟨*subroutineDec*⟩ 開始：建**副程式表**；若是**方法**先加一筆 `this`（type = 類別名、kind = argument、offset 0）；再為 ⟨*parameterList*⟩ 每個參數加 argument、為每個 ⟨*varDec*⟩ 加 local；本體結束時釋放；
- 查變數時**副程式表優先**（區域遮蔽）。

4.2　運算式與 term 的 VM 碼

解答

compile_expression：後序輸出——`compile_term`（第一個 term 的值上堆疊），然後每遇一個運算子與 term：先 `compile_term`、再輸出運算子的 VM 碼。由左至右、無優先順序（Jack 的設計——括號才有結合力，這正是 `()` 只作為 ⟨*term*⟩ 出現的原因）。

op	VM 碼	op	VM 碼
`+`	`add`	`&`	`and`
`-`	`sub`	`|`	`or`
`*`	`call Math.multiply 2`	`<`	`lt`
`/`	`call Math.divide 2`	`>`	`gt`
	（VM 沒有乘除指令！）	`=`	`eq`

compile_term——目標：算出值、留在堆疊頂：

term	VM 碼
整數 n	`push constant n`
`true`	`push constant 1`、`neg`（$-1 =$ 全 1）
`false` ／ `null`	`push constant 0`
`this`	`push pointer 0`
變數（查表）	`push local/argument/static/this` i
`a[e]`（陣列）	push a → 編譯 e → `add` → `pop pointer 1` → `push that 0`
`(e)`	遞迴 `compile_expression`
一元 `-`／`~`	先編譯 term，再 `neg` ／ `not`
⟨*subroutineCall*⟩	見 4.4 節
字串字面值	見 4.5 節

變數的段對映即符號表 kind：var → `local`、參數 → `argument`、static → `static`、field → `this`（不變式：`this 0` 永遠是當前物件的位址，所以第 i 個欄位就是 `this` i）。陣列則走 `that` 段：把基底＋下標的和放進 `pointer 1`，`that 0` 就是那一格——this 給欄位、that 給 `[]`，兩者互不打架。

4.3　敘述的 VM 碼

解答

compile_let（普通變數）：編譯右邊的 ⟨*expression*⟩，`pop` 到查表得到的段與編號。

compile_let（陣列目標）`let a[e1] = e2` ——本作業最精緻的一段舞步：

```
push a          // 基底
//（編譯 e1 的 VM 碼）
add             // 堆疊頂 = a+e1 的位址
//（編譯 e2 的 VM 碼）——過程中可能用到 pointer 1!
pop temp 0      // 先把 e2 的值收進 temp 0
pop pointer 1   // 現在才把 a+e1 裝進 pointer 1
push temp 0
pop that 0      // RAM[a+e1] = e2
```

為什麼不能「先 `pop pointer 1` 再編譯 e2」？因為 e2 自己可能含陣列存取（如 `let a[i] = b[j]`），會**覆寫 pointer 1**。先算位址、再算值、值暫存 temp 0、最後才設定指標——順序就是為了讓兩邊的陣列互不踩腳。ComplexArrays 測試（`let a[b[a[3]]] = a[a[5]] * b[...]`）專門抓這個。

compile_while（標籤由計數器生成，如 `while_start_4` ／ `while_end_4`）：

```
label while_start_4
//（條件運算式的 VM 碼）
not
if-goto while_end_4
//（本體 statements 的 VM 碼）
```

```
goto while_start_4
label while_end_4
```

compile_if：

```
// (條件運算式的 VM 碼)
not
if-goto if_else_7
// (then 部分)
goto if_end_7
label if_else_7
// (else 部分，若無 else 則兩標籤重合)
label if_end_7
```

compile_do：編譯 ⟨*subroutineCall*⟩ 之後 **務必** `pop temp 0`——呼叫必留一個返回值在堆疊上，沒人要就得丟掉，否則堆疊「漏水」，迴圈裡的 do 會讓堆疊無限長高。

compile_return：有運算式就編譯它；**void 函式推假值** `push constant 0`（VM 的 `return` 一律搬一個返回值，呼叫端的 `pop temp 0` 正好接走）。最後輸出 `return`。

4.4　副程式：呼叫與定義的三種情況

解答

編譯 ⟨*subroutineCall*⟩——三種形態：

形態	產生的碼
`var.method(args)`（. 左邊查表**是**變數）	`push` 該變數（物件位址當第一個引數）→ 編譯 args → `call` 型別`.method (n+1)`
`Class.sub(args)`（. 左邊**不在**表裡）	編譯 args → `call Class.sub n`（函式或建構子，無隱藏引數）
`method(args)`（無 .）	對**自己**呼叫方法：`push pointer 0` → 編譯 args → `call` 本類別`.method (n+1)`

注意第一種的 `call` 用的是變數的**型別**（查符號表的 type 欄位）拼出 `Type.method`——這就是 type 欄位存在的理由；`is_primitive` 幫你排除 `int` 等不可能有方法的情況。

編譯 ⟨*subroutineDec*⟩——輸出 `function` 類別名`.` 副程式名 `k`（k = local 數，從副程式表數 VK_VAR），然後按種類加開場白：

種類	開場白
函式	（無）
方法	`push argument 0`、`pop pointer 0` ——把呼叫者傳來的當前物件裝進 this 基底
建構子	`push constant` f（f ＝ 類別表的 field 數）、`call Memory.alloc 1`、`pop pointer 0`——在堆積配置新物件、設 this 基底；結尾必 `return this`（`push pointer 0` 再 `return`，Jack 文法已強制）

開場白之後，**本體內不要再動 `pointer 0`**——維持「`this 0` ＝ 當前物件」的不變式，欄位存取才永遠正確。

4.5　字串字面值

解答

官方做法——三步：

```
push constant L         // L = 字串長度
call String.new 1       // 堆疊頂 = 新字串的位址
// 對每個字元 c:
push constant 72        // 'H' 的 ASCII 碼
call String.appendChar 2   // 返回同一字串位址，可連鎖
//（重複 L 次）
```

題目的提示：Hack 字元集在一般字元上**與 ASCII 一致**——所以 C 端直接把 `char` 當 `int` 用（`sprintf("push constant %d", (int) c)`），**不需要** 50 多筆的查表。

（第十一週影片也誠實揭露：這個做法讓每次執行到字面值都在堆積配置新字串、永不釋放——Jack 的 Hello world 自帶記憶體洩漏。測試腳本都照官方語意設計，本作業照官方做法即可。）

5　測試你的編譯器

題目

編譯六個測試程式，在 VM 模擬器中**載入整個資料夾**執行（資料夾內含標準函式庫），驗證行為。

解答

測試矩陣（照序漸進，每個只引入少量新特性）：

程式	行為	新考點
Seven	螢幕左上印 7（$(3 \times 2) + 1$）	最小可行：do、運算式、`Math.multiply`
ConvertToBin	把 RAM[8000] 轉二進位輸出到 RAM[8001..8016]（低位在前，記憶體檢視器裡顯示是反序）——用望遠鏡圖示快速跳到位址	if/while、let、參數修改、多變數宣告
Square	方向鍵移動方塊；z 放大、x 縮小（限移動中或位於左上角）；q「結束」（畫面不變但不再回應輸入）	完整 OOP：方法、建構子、欄位
Average	讀入一串整數印平均值（影片 11-1 的程式微調版）	陣列 + 字串字面值
Pong	單人 Pong：得分、球拍隨得分縮短、game over 畫面	static 變數、一元 `-`、`|`、多類別合作
ComplexArrays	五組刁鑽陣列運算，印出期望值與實際值（應相同）	`let a[e1] = e2` 的 temp 0 舞步（4.3 節）

題目給的除錯錦囊（全部實測有用）：

1. VM 模擬器設 **No animation** 快跑時，動畫速度滑桿變成**時脈速度**——Square ／ Pong 這種互動程式一定要拉滿；
2. 用 `currentFunction` 變數設**斷點**——「每次進入某函式就停」，直接跳過一定正確的函式庫呼叫；
3. 程式崩潰時，視窗**左下角**有呼叫堆疊（按呼叫順序列出所有函式）——先看死在哪個函式再回頭讀碼；
4. 測試程式編不對時，**自己寫最小失敗範例**：把出錯的構造抽成十行的 Jack 程式，在 VM 模擬器先讓它動起來。Average 很慢的話，把 `Keyboard.readInt` 的字串字面值改短，重跑到出錯點的時間就短了。

（選做）任務 3——完整管線：把三個作業串起來：本週的編譯器（`.jack` → `.vm`）→ 第 9–10 週的 VM 翻譯器（`.vm` → `.asm`）→ 第 8 週的組譯器（`.asm` → `.hack`），最後裝進第 7 週你在 Logisim 蓋的 CPU——**從高階語言原始碼到你自己蓋的硬體，整條路每一吋都是你寫的。**實務注意：資料夾裡每個 `.jack` 都要編譯、連同 OS 函式庫的 `.vm` 一起交給 VM 翻譯器（它會自動加開機碼），輸出的 `.asm` 可能上萬行——組譯器的符號表若還是線性搜尋，這裡會第一次感覺到慢。

進階討論

回頭看：你用 11 週造了一整條「圖靈之塔」。NAND 閘（週 1）→ ALU 與時序邏輯（週 2–4）→ CPU（週 7）→ 組譯器（週 8）→ VM（週 9–10）→ 高階語言編譯器（週 11）。幾個對照現實的註腳：（1）真實編譯器（GCC/Clang）在解析與碼產生之間還有 **十幾層 IR 與上百個優化趟**

——你寫的是「無優化的單趟碼產生」，正是 1970 年代 Pascal P-code 編譯器的架構；（2）Jack 的「方法呼叫 = 物件當第一個引數」就是 Python 的 `self`、C++ 的隱含 `this` 指標——所有 OOP 語言在 ABI 層都這樣做；（3）`pop temp 0` 丟棄返回值的慣例，對應 C 中「運算式敘述丟棄值」——連 `printf(...);` 都默默丟掉一個 int。語言越高階，下面的堆疊機器越像——因為你現在知道，它們最後都要過同一種窄門。

A 附錄：速查表

A.1 Jack 文法（EBNF，本課程方言）

結構層：

⟨*class*⟩ ::= ‘`class`’, identifier, ‘`{`’, {⟨*classVarDec*⟩}, {⟨*subroutineDec*⟩}, ‘`}`’

⟨*classVarDec*⟩ ::= (‘`static`’ | ‘`field`’), ⟨*type*⟩, identifier, {‘`,`’, identifier}, ‘`;`’

⟨*type*⟩ ::= ‘`int`’ | ‘`char`’ | ‘`boolean`’ | identifier

⟨*subroutineDec*⟩ ::= (‘`constructor`’ | ‘`function`’ | ‘`method`’), (‘`void`’ | ⟨*type*⟩), identifier, ‘`(`’, ⟨*parameterList*⟩, ‘`)`’, ⟨*subroutineBody*⟩

⟨*parameterList*⟩ ::= [⟨*type*⟩, identifier, {‘`,`’, ⟨*type*⟩, identifier}]

⟨*subroutineBody*⟩ ::= ‘`{`’, {⟨*varDec*⟩}, ⟨*statements*⟩, ‘`}`’

⟨*varDec*⟩ ::= ‘`var`’, ⟨*type*⟩, identifier, {‘`,`’, identifier}, ‘`;`’

敘述層：

⟨*statements*⟩ ::= {⟨*letStatement*⟩ | ⟨*ifStatement*⟩ | ⟨*whileStatement*⟩ | ⟨*returnStatement*⟩ | ⟨*doStatement*⟩}

⟨*letStatement*⟩ ::= ‘`let`’, identifier, [‘`[`’, ⟨*expression*⟩, ‘`]`’], ‘`=`’, ⟨*expression*⟩, ‘`;`’

⟨*ifStatement*⟩ ::= ‘`if`’, ‘`(`’, ⟨*expression*⟩, ‘`)`’, ‘`{`’, ⟨*statements*⟩, ‘`}`’, [‘`else`’, ‘`{`’, ⟨*statements*⟩, ‘`}`’]

⟨*whileStatement*⟩ ::= ‘`while`’, ‘`(`’, ⟨*expression*⟩, ‘`)`’, ‘`{`’, ⟨*statements*⟩, ‘`}`’

⟨*doStatement*⟩ ::= ‘`do`’, ⟨*subroutineCall*⟩, ‘`;`’

⟨*returnStatement*⟩ ::= ‘`return`’, [⟨*expression*⟩], ‘`;`’

運算式層：

⟨*expression*⟩ ::= ⟨*term*⟩, {(‘`+`’ | ‘`-`’ | ‘`*`’ | ‘`/`’ | ‘`&`’ | ‘`|`’ | ‘`<`’ | ‘`>`’ | ‘`=`’), ⟨*term*⟩}

⟨*term*⟩ ::= integer literal | string literal | ‘`true`’ | ‘`false`’ | ‘`null`’ | ‘`this`’ | identifier, [‘`[`’, ⟨*expression*⟩, ‘`]`’] | ‘`(`’, ⟨*expression*⟩, ‘`)`’ | ((‘`-`’ | ‘`~`’), ⟨*term*⟩) | ⟨*subroutineCall*⟩

⟨*subroutineCall*⟩ ::= identifier, [‘`.`’, identifier], ‘`(`’, ⟨*expressionList*⟩, ‘`)`’

⟨*expressionList*⟩ ::= [⟨*expression*⟩, {‘`,`’, ⟨*expression*⟩}]

A.2 VM 碼模式速查

構造	VM 碼模式
變數讀	`push local/argument/static/this` i（查表）
`let v = e`	編譯 e；`pop` 段 i
`let a[e1] = e2`	push a；編譯 e_1；`add`；編譯 e_2；`pop temp 0`；`pop pointer 1`；`push temp 0`；`pop that 0`
`a[e]`（讀）	push a；編譯 e；`add`；`pop pointer 1`；`push that 0`
while	`label S`；條件；`not`；`if-goto E`；本體；`goto S`；`label E`
if/else	條件；`not`；`if-goto ELSE`；then；`goto END`；`label ELSE`；else；`label END`
do call	編譯呼叫；`pop temp 0`
return（void）	`push constant 0`；`return`
方法開場	`push argument 0`；`pop pointer 0`
建構子開場	`push constant` f；`call Memory.alloc 1`；`pop pointer 0`
`var.m(args)`	push var；args；`call Type.m (n+1)`
`m(args)`（本類別方法）	`push pointer 0`；args；`call` 本類別`.m (n+1)`
字串 `"Hi"`	`push constant 2`；`call String.new 1`；`push constant 72`；`call String.appendChar 2`；……

A.3 常見錯誤型錄

症狀	病因
解析輸出多了 `<type>` ／ `<parameterList>` ／ `<subroutineDec>` 標籤	忘了抑制規則（測試資料不含這三種標籤）
`foo.bar()` 被解析成變數 `foo`	`parse_term` 的識別字情況沒看 `lookahead`
Seven 印不出 7	`*` 沒翻成 `call Math.multiply 2`，或 do 後忘了 `pop temp 0`
方法內欄位讀到垃圾	方法開場忘了 `push argument 0; pop pointer 0`，或方法的符號表忘了先加 `this`（之後所有 argument 編號都差 1）
建構子返回的物件位址是 0	忘了 `Memory.alloc` ／ `pop pointer 0`，或 alloc 的大小算成 local 數而非 field 數
ComplexArrays 前三題對、後兩題錯	`let a[e1]=e2` 沒走 temp 0 舞步，e2 裡的陣列存取覆寫了 pointer 1
迴圈裡的 do 越跑越慢終致崩潰	返回值沒 `pop temp 0`，堆疊無限長高
void 函式的呼叫者拿到垃圾	void return 忘了 `push constant 0`
同名變數行為錯亂	查表順序錯——副程式表必須優先於類別表（區域遮蔽）
if 無 else 時跳錯	else ／ end 標籤生成邏輯沒處理「無 else」情況

A.4 變數映射與符號表備忘

Jack 宣告	kind	VM 段	符號表
`var`	VK_VAR	`local`	副程式表
參數	VK_ARGUMENT	`argument`	副程式表（方法：`this` 佔 offset 0）
`static`	VK_STATIC	`static`	類別表
`field`	VK_FIELD	`this`	類別表

A.5 參考資料

- COMSM1302 第十一週講義與影片（Jack 語言、解析與碼產生、類別編譯），University of Bristol.
- Nisan & Schocken, *The Elements of Computing Systems*（nand2tetris），Ch. 9–11（Jack、Syntax Analysis、Code Generation）——測試程式 Square ／ ArrayTest ／ Seven 等的出處.
- nand2tetris Project 10 與 11（nand2tetris.org/project10、/project11）.
- W3C, *Extensible Markup Language (XML) 1.0* ——標籤對與字元跳脫的正式定義.