COMSM1302 電腦架構概論
第八週完整教材：編譯器概念

詞法分析 · 符號表 · 語法分析與 BNF · Hack 組譯器

依據 John Lapinskas（University of Bristol）課程講義編寫並延伸

2026 年 7 月

本教材的使用方式：本講義整合了第八週四份投影片（8-1 Compiler concepts: Lexing、8-2 Compiler concepts: Symbol tables、8-3 Compiler concepts: Parsing、8-4 A parser for Hack assembly）的全部內容，並補充了遞迴下降解析（recursive descent）等延伸知識。每章結尾附有「本章重點」整理；第 5 章為綜合練習題，附完整詳解。上週我們蓋完了 Hack 的**硬體**；本週正式跨入**軟體**：組譯器（assembler）是怎麼把組語文字變成機器碼的？這也是本週作業——**用 C 寫一個自己的 Hack 組譯器**——的理論基礎。

目錄

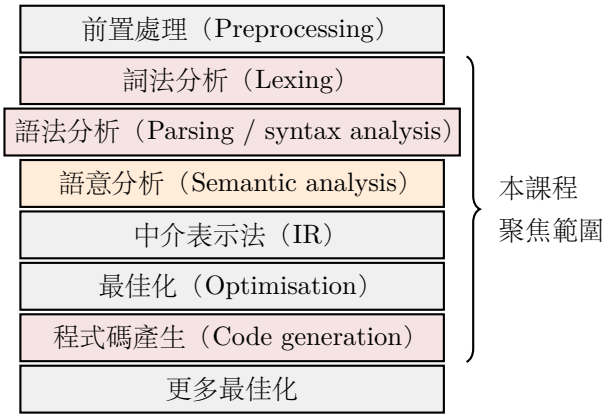
1 編譯器的全貌與詞法分析（Lexing）

1.1 下一個目標

我們已經懂了 Hack 組語與 Hack ISA，也會**用**組譯器把組語變成機器碼。但**組譯器本身是怎麼運作的**？本週的目標：**用 C 打造一個自己的 Hack 組譯器**！在抽象階梯上，我們正在處理「組合語言 → ISA」這一步的翻譯軟體——而它的原理與「高階語言 → 組語」的編譯器一脈相承。

1.2 編譯器的分階段全貌

現代編譯器通常以**離散的階段（phases）**工作：



- **前置處理**：語言特定的步驟，例如把程式檔連結起來、執行編譯器巨集。在 C 中包括讀 makefile、處理 `#include` 與 `#define`。
- **最佳化**：自動把低效程式碼變成高效程式碼的過程。

兩者都超出本課程範圍。

1.2.1 中介表示法（IR）

多數編譯器把語意分析萃取出的資訊放進程式碼的 **中介表示法（intermediate representation, IR）**，然後**對 IR（而非原始碼）做最佳化**。IR 是給電腦（而非人類）用的，相當接近組語；「最佳化 IR」是最佳化裡**困難的那一半**。許多語言可以編譯到**同一個** IR——例如 LLVM 被用於 C#、Java bytecode、Ruby 與 Rust 的編譯器。下週起我們會看到 Hack 的 IR；但一行組語本來就對應一行機器碼，所以**組譯器不需要 IR**。

1.2.2　語法 vs 語意

語法（syntax）

語言或程式碼的**邏輯結構**。以英文句子“The trophy wouldn’t fit in the case because it was too big.”為例，語法分析告訴你“trophy”和“case”是名詞、句子拆成兩個子句——關於文法結構的一切。以 `strcpy(x,"Test");` 為例，語法分析告訴你：這是對名為 `strcpy` 的函式的呼叫，第一個引數是變數 `x`、第二個是字串 `"Test"`。語法分析可以很複雜，但是**被透徹理解的問題**，有許多標準演算法。語法分析的過程稱為**解析（parsing）**。

語意（semantics）

語言或程式碼的**意義**。同一個英文句子，語意分析告訴你“it”指的是獎盃、不是櫃子——兩種解讀在文法上都合法，但只有一種說得通。對 `strcpy(x,"Test");`，語意分析會發現 `strcpy` 未定義（忘了 `#include <string.h>`）、`x` 是 `char *`，甚至找出「系統時間恰為午夜時指標指向未配置記憶體」的 bug。**語意分析非常困難**！型別檢查等部分已被充分理解，但自動抓 bug（「驗證」）仍是研究領域。

1.2.3　組譯器的簡化

對組譯器而言，語法與語意幾乎相同，常見做法是把語意分析**併入**詞法分析與解析。沒有複雜語意，也就不必把程式碼產生跟解析分開——逐行處理即可。

我們的 Hack 組譯器

只有兩個步驟：**詞法分析（lexing）**與**解析（parsing）**。就 Hack 組語而言這其實還是殺雞用牛刀——但我們會完整地做，以建立通用的原理。

1.3　The Joy of Lex：權杖

權杖（token）

又稱 lexeme 或詞法元素：程式碼中「不可分割」的片段——再拆下去就會失去意義。英文裡最接近的類比是**單字與標點**：

```
Never | gonna | give | you | up | , | never | gonna | let | you | down | .
```

詞法分析（lexing）

把一串程式碼**文字**轉換成一串**權杖**的過程。通常只需要移除空白並比對字串。

可用的權杖列表寫在程式語言的**規格書**裡，連同把權杖組成語句的**文法**（見第 3 章）。**什麼算一個權杖是語言設計者的選擇**，不是普世標準——例如 Java 把註解與空白當權杖，C 則不是。

1.4　Hack 組語的權杖

Nisan 與 Schocken 沒有給 Hack 組語文法，但我們可以自建。我們取以下權杖：

類別	內容
關鍵字（keywords）	`A`、`D`、`M`；`JGT`、`JEQ`、`JLT`、`JGE`、`JNE`、`JLE`、`JMP`；`SCREEN`、`KBD`、`SP`、`LCL`、`ARG`、`THIS`、`THAT`；`R0` 到 `R15`
符號（symbols）	`@`、`+`、`-`、`&`、`|`、`=`、`;`、`!`
整數字面值	任何 0…32767 範圍內的十進位整數
識別字（identifiers）	任何不含空白、不是關鍵字、以字母開頭的字串
換行（newlines）	

幾個關鍵觀察：

- `A` 與 `ARG` 是**不同的關鍵字**——區分它們要小心（不能看到 `A` 就停）。一般而言，**關鍵字**是語言本身賦予特殊意義、通常不能重新定義的字串。C 的關鍵字包括 `const`、`int`、`for`、`return`，但**不**包括 `printf`。
- C 把符號稱為 punctuators 與 operators，且有多字元符號如 `>=`、`==`、`%:` 與 `%:%:`（後兩者由於歷史因素是 `#` 與 `##` 的同義詞）。
- C 把字面值稱為 constants，還有字串、浮點、enum 與字元常數。
- **識別字**代表**在程式碼（而非語言）中定義**的實體，例如特定變數、函式或 struct 型別。在 C 中 `printf` 是 stdio 函式庫定義的識別字。注意我們**不區分變數與標籤**——都先當識別字。
- C 完全忽略換行，但我們**需要**換行權杖——否則無法區分「`A` 之後接 `D=M`」與 `AD=M`。

注意少了什麼！空白與註解直接**丟掉**，不產生權杖。**標籤也會被丟掉**而不變成權杖（見第 2 章）——所以符號表裡**不**包括 `(` 和 `)`。

1.5　詞法分析的好處與範例輸出

為什麼要費工夫做詞法分析？因為之後可以把權杖存成**乾淨的 struct**，從此不必再跟 C 糟糕的字串處理搏鬥（至少撐到程式碼產生階段…）。

詞法器輸出範例

考慮測試案例中的這行 Hack 程式碼：

```
D;JGT      // if D>0 goto output_first
```

你的詞法器會把它轉成**四個** Token struct：

	type	value
Token 1	Keyword	`D`
Token 2	Symbol	`;`
Token 3	Keyword	`JGT`
Token 4	Newline	None

其中 `D` 與 `JGT` 以 enum 儲存、`;` 以 char 儲存；接著呼叫 `write_token` 把權杖寫入暫存檔。你的解析器再用 `read_token` 讀取 Token struct ——處理的是權杖檔，而不是原始碼。註解 `// if`

`D>0 ...` 被整個丟棄，**不產生任何權杖**。你的詞法器也會處理**標籤**（通常屬於語意分析）——見下一章！

本章重點

- 編譯器分階段：前置處理 → 詞法 → 語法 → 語意 → IR → 最佳化 → 產碼。組譯器只需**詞法＋解析**兩步。
- 語法 = 結構（被透徹理解）；語意 = 意義（很難，驗證是研究領域）。
- 權杖 = 不可分割的程式碼片段；詞法分析 = 文字 → 權杖串。
- Hack 權杖五類：關鍵字／符號／整數字面值／識別字／換行；空白、註解、標籤宣告不產生權杖。
- 換行必須是權杖：否則 `A+D=M` 與 `AD=M` 無法區分。

2 符號表（Symbol Tables）

2.1 問題：標籤與變數

回想 Hack 組語中 `@` 後面可以接**數字**、**標籤**或 **變數**。組譯器必須：

1. 為每個**變數**配置一個 RAM 位址（從 16 開始）；
2. 用位址**取代**變數；
3. 為每個**標籤**指定 ROM 位址——等於其宣告處對應的**機器碼行號**；
4. 用位址**取代**標籤；
5. **然後才**把 `@` 語句轉成 A-指令。

以投影片的範例程式追蹤整個過程（左：原始碼，右：完成後）：

行	原始碼	⟹	行	翻譯後
0	`@input1`	⟹	0	`@16`
1	`D=M`	⟹	1	`D=M`
2	`@input2`	⟹	2	`@17`
3	`D=D-M`	⟹	3	`D=D-M`
4	`@output_first`	⟹	4	`@10`
5	`D;JGT`	⟹	5	`D;JGT`
6	`@input2`	⟹	6	`@17`
7	`D=M`	⟹	7	`D=M`
8	`@output_d`	⟹	8	`@12`
9	`0;JMP`	⟹	9	`0;JMP`
10	`(output_first)`	⟹	10	`@16`
11	`@input1`	⟹	11	`D=M`
12	`D=M`	⟹	12	`@18`
13	`(output_d)`	⟹	13	`M=D`
14	`@output_val`	⟹	14	`@14`
15	`M=D`	⟹	15	`0;JMP`
16	`(infinite_loop)`	⟹		
17	`@infinite_loop`	⟹		
18	`0;JMP`	⟹		

關鍵陷阱：標籤宣告不佔機器碼行！ `(output_first)` 這種行會被**整行移除**，所以它後面的指令行號要**往前縮**。例如 `(output_first)` 原本在第 10 行，移除後「下一條指令」`@input1` 才是機器碼第 10 行——所以 `output_first` 的 ROM 位址是 **10**。同理 `output_d` → 12、`infinite_loop` → 14 （三個標籤宣告依次被移除，行號累計前縮）。

我們用**符號表**完成這件事。

2.2　識別字與符號表

識別字（identifier）

意義**在程式碼本身（而非語言）中定義**的權杖的統稱。在 Hack 中，識別字就是**標籤與變數**——看到 `@output_first` 時我們知道 `@` 是什麼意思，但 `output_first` 只能**找它的定義**才能翻譯。在 C 中，函式名稱也是識別字。

符號表（symbol table）

把識別字的**名字**對應到其**意義**的資料結構。在 Hack 中我們用**兩張**符號表：標籤表（標籤名 → ROM 位址）與變數表（變數名 → RAM 位址）。在 C 中，符號表還會存變數的**型別**、函式的**引數**等——歷史上「高效填符號表」的需求正是**函式標頭（headers）存在的原因**。

上面範例對應的兩張表：

標籤表

名稱	ROM 位址
`output_first`	10
`output_d`	12
`infinite_loop`	14

變數表

名稱	RAM 位址
`input1`	16
`input2`	17
`output_val`	18

2.3　符號表如何運作?

符號表必須支援三種操作：

1. **加入**新的名稱與位址；
2. **檢查**某名稱是否在表中；
3. 若在表中，**取出**對應的位址。

「正確」的實作方式是**雜湊表（hash table）**（幾週後的 Programming in C 會教）。但用你已學過的**動態大小容器**（動態陣列）也能做，只是較沒效率——每次查詢線性掃描整個陣列即可。這是好的 C 練習、但不是好的架構練習，所以作業已幫你寫好——見 `symboltable.c` 與 `symboltable.h`。

2.4　如何填符號表?

在組語中，填符號表簡單到可以**整合進詞法分析與解析**（在 C 之類的語言中則發生在較後面的語意分析階段）。兩張表都從**空表**開始。

填表流程

詞法分析階段（處理標籤）：識別字是變數還是標籤，看有沒有**標籤宣告**（`label`）即可。於是在 lexing 時，我們**移除**標籤宣告、把它們與正確的 ROM 位址加進**標籤表**。（回想：標籤宣告**不產生任何權杖**！）

解析階段（處理變數）：對遇到的每個識別字，查兩張表：

- 在**標籤表**裡 → 太好了，代入 ROM 位址；
- 在**變數表**裡 → 太好了，代入 RAM 位址；
- **兩張表都沒有** → 必是某變數的**首次出現**：把它加進變數表，配置**第一個未使用的 RAM 位址**（從 16 起）。

為什麼標籤要在**詞法階段**先處理？因為變數與標籤在使用處**長得一模一樣**（都是 `@` 名字），唯一能區分它們的是**宣告**；而 `@` 名字可能出現在（名字）宣告**之前**（向前跳轉）。先掃一遍把所有標籤收齊，解析時才不會把標籤誤當新變數。

2.5　進階符號表：作用域（Scopes）

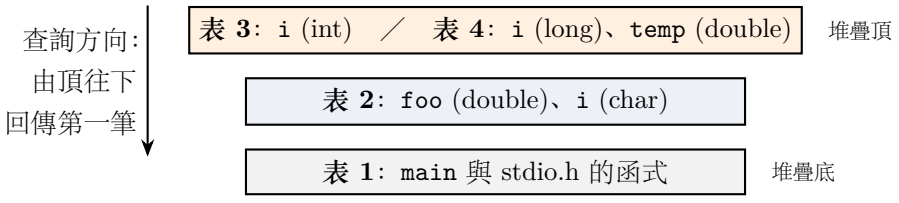
高階語言的編譯器需要追蹤**作用域**：**每個作用域建一張符號表**。語意分析建好表後，轉換成 IR 時可以把它們存在一個 **堆疊（stack）** 裡。考慮這段 C 程式：

```c
#include <stdio.h>

int main() {
    double foo = 7;
    char i = 'a';
    for (int i = 0; i <= 5; i++) {
        printf("%f, %d", foo, i);
        foo /= 2;
    }

    foo = 50;
    for (long i = 0; i <= 10; i++) {
        double temp = foo + 500;
        printf("%f, %d", temp, i);
        foo *= 2;
    }

    printf("%c", i);    // Prints 'a'
    printf("%d", temp); // Compile error!
    return 0;
}
```



編譯器的堆疊操作：從表 1 開始，第 3 行 **push** 表 2、第 6 行 push 表 3、第 9 行 **pop**（for 迴圈結束）、第 12 行 push 表 4、第 16 行 pop、第 21 行 pop。查詢變數時**從堆疊頂往下找，回傳第**

一筆結果——這就是「內層變數遮蔽（shadow）外層同名變數」的機制：在第一個 for 內查 `i` 會先找到表 3 的 `int i`，而不是表 2 的 `char i`。第 18 行印出 `'a'`（表 3、表 4 已被 pop，查到表 2 的 `char i`）；第 19 行編譯錯誤（`temp` 只存在於已被 pop 的表 4）。

本章重點

- 組譯器的任務順序：配置變數 RAM 位址（16 起）→ 標籤取 ROM 位址（宣告處的機器碼行號）→ 全部替換 → 才轉 A-指令。
- 標籤宣告行被移除、不佔機器碼行號——後續行號要前縮。
- 符號表 = 名稱 → 意義的映射；支援加入／檢查／取出；正解是雜湊表，動態陣列也可。
- 標籤在**詞法**階段填表（因為要靠宣告區分、且可能向前跳轉）；變數在**解析**階段「查無此人則新增」。
- 高階語言：一個作用域一張表、用堆疊管理，由頂往下查——內層遮蔽外層。

3　語法分析與文法（Parsing and Grammars）

3.1　描述語言：需要更好的工具

Nisan 與 Schocken 對 Hack 語法的描述散落在說明文字裡——對**人**讀還行，對**寫解析器**來說不夠精確方便。我們需要一個高度精密的數學工具：**Madlibs**（填詞遊戲）！

上下文無關文法（context-free grammar）

一種**快速且嚴謹**地指定「語言中哪些字串具有合法語法」的方式。背後有深厚的數學理論（好在不用學）。程式設計師用 **Backus-Naur Form（BNF）**表達文法，而且通常**看懂 BNF 就夠用了**。BNF 基本上就是**會遞迴的 Madlibs**。

3.2　BNF 入門

一個簡單例子：

$$\langle noun \rangle \ ::= \ \texttt{`lecturer'} \mid \texttt{`student'} \mid \texttt{`pizza'}$$
$$\langle presentVerb \rangle \ ::= \ \texttt{`eats'} \mid \texttt{`devours'} \mid \texttt{`consumes'}$$

每個 | 讀作「或」、每個 ::= 讀作「定義為」。例如一個 ⟨*noun*⟩ 定義為 `'lecturer'`、`'student'`、`'pizza'` 三個字串之一。定義可以**疊起來**：

$$\langle sentence \rangle \ ::= \ \texttt{`The'} \ \langle noun \rangle \ \langle presentVerb \rangle \ \texttt{`the'} \ \langle noun \rangle$$

合法的 ⟨*sentence*⟩ 包括「The lecturer consumes the pizza」、「The student eats the pizza」、「The lecturer devours the student」。

終端與非終端符號

文法中**我們自己定義**的東西必須包在 ⟨⟩ 裡，稱為**非終端符號（non-terminal symbols）**。其他的（如 `'lecturer'`）是**終端符號（terminal symbols）**——也就是**權杖**。

3.3　遞迴：BNF 力量的來源

BNF 的最後一個特性是力量的來源：**允許遞迴**。假設權杖是 `'0'` 到 `'9'`，想定義一個非終端符號恰好匹配所有非負整數（允許前導零）：

$$\langle digit \rangle \ ::= \ \texttt{`0'} \mid \texttt{`1'} \mid \texttt{`2'} \mid \texttt{`3'} \mid \texttt{`4'} \mid \texttt{`5'} \mid \texttt{`6'} \mid \texttt{`7'} \mid \texttt{`8'} \mid \texttt{`9'}$$
$$\langle number \rangle \ ::= \ \langle digit \rangle \mid \langle digit \rangle \ \langle number \rangle$$

例如 `016` 是 ⟨*number*⟩，因為可以這樣展開：

$$\langle number \rangle \to \langle digit \rangle \ \langle number \rangle \to \langle digit \rangle \ \langle digit \rangle \ \langle number \rangle \to \langle digit \rangle \ \langle digit \rangle \ \langle digit \rangle \to \texttt{`0'} \ \texttt{`1'} \ \texttt{`6'}.$$

就這樣！這就是全部的 BNF。用起來與推理起來可能很難，但語法本身很簡單。

3.4　練習：更好的整數

如何重新定義 ⟨*number*⟩，**允許負數但禁止前導零**？（權杖有 '0' 到 '9' 和 '-'。）任何重定義的健全性檢查：

- '-''1''0' 應該是 ⟨*number*⟩；✓
- '0' 應該是 ⟨*number*⟩；✓
- '0''1' **不**應該是 ⟨*number*⟩；✗
- '-''0' **不**應該是 ⟨*number*⟩。✗

做法很多——BNF 的文法表達**沒有唯一「正確」寫法**。其中一種：

$$\begin{aligned}
\langle posDigit \rangle &::= \text{'1'} \mid \text{'2'} \mid \text{'3'} \mid \text{'4'} \mid \text{'5'} \mid \text{'6'} \mid \text{'7'} \mid \text{'8'} \mid \text{'9'} \\
\langle posNumber \rangle &::= \langle posDigit \rangle \mid \langle posNumber \rangle\ \langle posDigit \rangle \mid \langle posNumber \rangle\ \text{'0'} \\
\langle number \rangle &::= \langle posNumber \rangle \mid \text{'0'} \mid \text{'-'}\ \langle posNumber \rangle
\end{aligned}$$

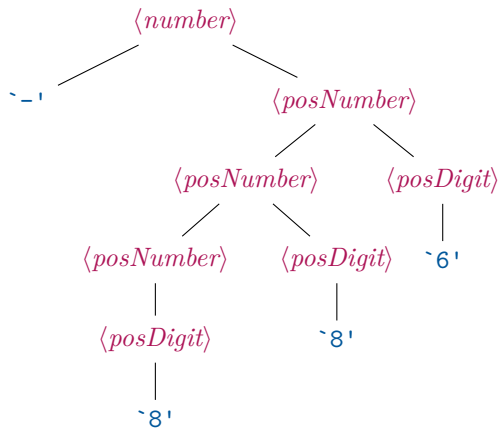
思路：⟨*posNumber*⟩ 的**第一個字元**只能經由 ⟨*posDigit*⟩ 產生（非零），之後可以接任何數字（包括 0）；'0' 只允許單獨出現，且負號後面只能接 ⟨*posNumber*⟩。

3.5　解析樹（Parse Trees / CST）

解析的目標

把權杖列表轉換成**解析樹（parse tree）**，又稱**具體語法樹（concrete syntax tree, CST）**，呈現它的 BNF 結構。**每個非終端符號是一個節點**，其子節點是它的 BNF 展開（由左至右）——所以**葉子恰好是權杖**。

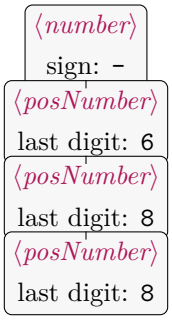
以"−886"為例（用上面的文法）：



展開路徑：⟨*number*⟩ → '-' ⟨*posNumber*⟩；⟨*posNumber*⟩ → ⟨*posNumber*⟩ ⟨*posDigit*⟩（吃掉 6）→ ⟨*posNumber*⟩ ⟨*posDigit*⟩（吃掉第二個 8）→ ⟨*posDigit*⟩（吃掉第一個 8）。

3.6　抽象語法樹（AST）

為了效率與方便，我們可以把 CST 加工成 **抽象語法樹（abstract syntax tree, AST）**：**相同的資訊、更方便的形式**。例如決定不需要 ⟨*posDigit*⟩ 節點，把數字直接標在節點上：



3.7　歧義（Ambiguity）

考慮這個簡單算術運算式的文法：

$$\begin{aligned}\langle expression\rangle &::= \langle number\rangle \mid \langle expression\rangle\ \langle operator\rangle\ \langle expression\rangle \\ &\quad \mid \text{`('}\ \langle expression\rangle\ \langle operator\rangle\ \langle expression\rangle\ \text{`)'} \\ \langle operator\rangle &::= \text{`+'} \mid \text{`-'} \mid \text{`*'} \mid \text{`/'} \mid \text{`\^{}'}\end{aligned}$$

那麼對 $(3+4)*(5-1)/3$，解析器可以輸出**好幾棵不同的合法 CST**（先把 $*$ 當根、還是先把 $/$ 當根?）。只要**語意**不歧義（此處無論哪棵樹算出來都一樣），這種歧義是可以接受的。但在真實語言中（如 $3-4-5$ 的左右結合、或懸掛 else），文法設計者通常會改寫文法來消除歧義。

3.8　實務上怎麼做? 用解析器產生器

解析是困難而微妙、但**被透徹理解**的問題——這代表**我們不該自己重新解決它**！應該用**解析器產生器（parser generator）**：把 BNF 文法餵給它，它輸出你選定語言的解析器程式碼。（例如 C 的 `yacc`。）

3.9　擴充 BNF（EBNF）

解析器產生器與語言規格書常為易用性給 BNF 加語法，但**沒有統一標準**。鬆散依據 ISO 14977，我們加入：

記號	意義	例子
( )	分組	(`0`|`1`) (`0`|`1`) 匹配 00、01、10、11
[ ]	可有可無	[`0`] `1` 匹配 01 或 1
{ }	重複任意次	{`0`|`1`} 匹配任何 0/1 串（含空字串）
$A-B$	匹配 A 但不匹配 B	⟨*number*⟩ − ⟨*digit*⟩：不是單一數字的數

（$A - B$ 只在 B 只能展開成**有限多**種權杖序列時是合法 EBNF，例如 $\langle number \rangle - \langle posNumber \rangle$ 就不合法。）

EBNF **沒有**讓 BNF 能表達任何原本表達不了的文法（每個 EBNF 構造都能機械地改寫回純 BNF——分組展開、選項用兩條規則、重複用遞迴），但**讀寫舒服得多**。例如「更好的整數」變成一行：

$$\langle number \rangle \ ::= \ (['-'] \ (\langle digit \rangle - '0') \ \{\langle digit \rangle\}) \mid '0'$$

有了 EBNF，連**整個 C 語言**的文法都能寫得可讀，何況 Hack!

本章重點

- BNF：::= 定義、| 或、⟨⟩ 非終端、引號內為終端（權杖）；**遞迴**是力量來源。
- 同一語言的 BNF 寫法不唯一；用健全性檢查驗證你的文法。
- CST：非終端 = 節點、BNF 展開 = 子節點、葉子 = 權杖。AST = 同資訊的方便形式。
- 歧義：多棵合法 CST；語意一致時可接受。
- 實務用解析器產生器（yacc 等）；EBNF 的 ()[]{} 與 − 只是語法糖。

4 Hack 組語的解析器

4.1 Hack 組語的 EBNF

把 Hack 形式化成 EBNF 的方式很多，以下是其中一種（, 表示串接、; 結束一條規則——這是 ISO 風格 EBNF 的慣例）：

Hack 組語文法

⟨*instruction*⟩ ::= (⟨*aInstruction*⟩ | ⟨*cInstruction*⟩), newline;

⟨*aInstruction*⟩ ::= "@", (integerLiteral | identifier | ⟨*memoryKeyword*⟩);

⟨*memoryKeyword*⟩ ::= "SCREEN" | "KBD" | "SP" | "LCL" | "ARG" | "THIS" | "THAT";

⟨*cInstruction*⟩ ::= [⟨*assignment*⟩], ⟨*computation*⟩, [⟨*jump*⟩];

⟨*assignment*⟩ ::= ("A" | "D" | "M" | ("A","D") | ("A","M") | ("D","M")
| ("A","D","M")), "=";

⟨*jump*⟩ ::= ";", ("JMP" | "JGT" | "JEQ" | "JLT" | "JGE" | "JNE" | "JLE");

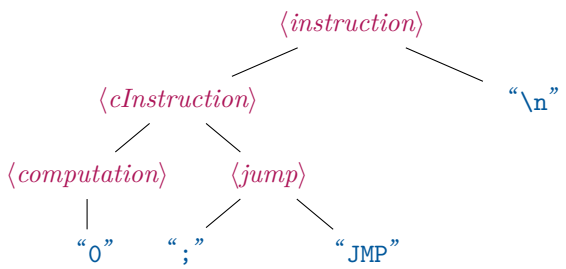
⟨*computation*⟩ ::= "0"
| (["-"], "1")
| (["-" | "!"], ("A" | "D" | "M"))
| (("A" | "D" | "M"), ("+" | "-"), "1")
| (("A" | "M"), ⟨*binaryOp*⟩, "D")
| ("D", ⟨*binaryOp*⟩, ("A" | "M"));

⟨*binaryOp*⟩ ::= "+" | "-" | "&" | "|";

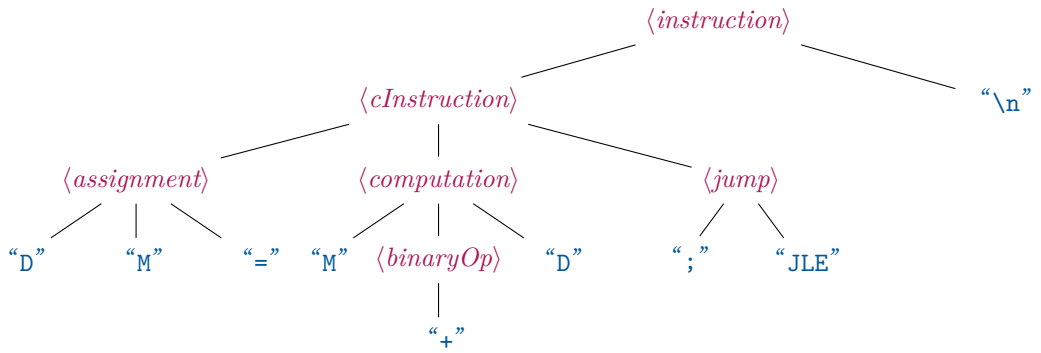
注意這個文法如何精準編碼第七週的 comp 表：⟨*computation*⟩ 的六個分支恰好對應常數（0、±1）、單運算元（含 -、!）、加減一、以及 D 與 A/M 的二元運算——**文法本身就排除了 A+M 這種不存在的運算。**

4.2 兩棵範例 CST

例一： `0;JMP\n`



例二： `DM=M+D;JLE\n`



4.3 LL 解析：由左往右、由上往下

LL 解析

本課程只考慮 **LL 解析**：由左至右掃過權杖，**只看接下來的幾個權杖**，從上往下建構 CST。（LL = **L**eft-to-right scan、**L**eftmost derivation。）

逐權杖追蹤例一 `0;JMP\n`：

1. 讀到 `0`：`0` 只可能是一整個 ⟨*computation*⟩，它必在 ⟨*cInstruction*⟩ 裡、又必在 ⟨*instruction*⟩ 裡——三個節點一次掛好。也得知這個 ⟨*cInstruction*⟩ **沒有** ⟨*assignment*⟩（但可能有 ⟨*jump*⟩）。
2. 讀到 `;`：只可能是 ⟨*jump*⟩ 的第一項，掛進已有的 ⟨*cInstruction*⟩。
3. 讀到 `JMP`：必是那個 ⟨*jump*⟩ 的延續。
4. 讀到 `\n`：必是 ⟨*instruction*⟩ 的結尾。完成！

若某一步**下一個權杖無處可掛**（例如解析 `01;JMP` 中的 `1` 時），就回報錯誤。

逐權杖追蹤例二 `DM=M+D;JLE\n`——需要往前看：

1. 讀到 `D`：**無法立刻判斷**它屬於 ⟨*assignment*⟩ 還是 ⟨*computation*⟩！解法：**往前看（lookahead）**第二個權杖——若是 `M` 或 `=`，必在 ⟨*assignment*⟩；若是 `+`、`-`、`&`、`|`、`;` 或 `\n`，必在 ⟨*computation*⟩。這裡下一個是 `M`，所以 `D` 開啟一個 ⟨*assignment*⟩。
2. 讀到 `M`：往前看見 `=`，確認仍在 ⟨*assignment*⟩。
3. 讀到 `=`：只能收尾這個 ⟨*assignment*⟩。
4. 讀到 `M`：往前看見 `+`，必在 ⟨*computation*⟩。
5. 讀到 `+`：可能是 `M+1` 型也可能是 `M+D` 型——再往前看一個權杖分辨；這裡是 ⟨*binaryOp*⟩，並預期 ⟨*computation*⟩ 的下一項是 `D`。
6. 讀到 `D`、`;`、`JLE`、`\n`：依次完成 ⟨*computation*⟩、⟨*jump*⟩ 與整個 ⟨*instruction*⟩。

LL(k)

能靠往前看 k 個權杖完成 LL 解析的文法稱為 **LL(k)**。Hack 是 **LL(2)**（因為 `DM=M+D` 這類情況）但**不是** LL(1)。

警告（原話）：這裡掃了很多很深的兔子洞到地毯下。不是所有文法都是 LL(k)，這也不是 LL(k) 的正式定義。但對一個 quick and dirty 的解析器實作來說夠用了！**延伸補充**：手寫 LL 解析器的標準做法叫 **遞迴下降（recursive descent）**：每條文法規則寫成一個函式，函式之間依文法結構互相呼叫；需要注意**左遞迴**的規則（如 ⟨*posNumber*⟩ ::= ⟨*posNumber*⟩ ⟨*posDigit*⟩）會讓樸素的遞迴下降無窮遞迴，須改寫成迴圈或右遞迴。GCC 與 Clang 的 C/C++ 解析器都是手寫的遞迴下降。

4.4 實際上怎麼解析 Hack?

CST 對**優雅的錯誤處理**和編譯 `i=(k+j++)/m` 這種複雜運算式極有用，但對 Hack 來說是殺雞用牛刀。我們**實際**需要知道的只有：

- 這條語句是 A-指令還是 C-指令？**（第一個權杖是不是 `@`？）**
- 若是 A-指令，該載入 A 的值是多少？**（`@` 後面是什麼？若是識別字，查符號表拿 RAM/ROM 位址。）**
- 若是 C-指令，dest、comp、jump 的值是多少？
  - `=` 有沒有出現？左邊有 `A`、`D`、`M` 中的哪些？
  - `;` 有沒有出現？右邊是哪個跳躍指令？
  - `=` 右邊、`;` 左邊是什麼？（約 30 種可能。）

這些問題全都能用**對權杖的簡單邏輯**回答。（不過如果有多的時間，把 CST 做出來仍是很好的練習！）

4.5 Hack 組譯器總結

完整的兩趟式（two-pass）組譯器

第一趟：詞法分析。對每一行：

- 移除所有註解與空白；
- 空行直接跳過；
- 若該行是**標籤宣告**，把它與**目前行對應的 ROM 位址**加入符號表（該行本身不輸出）；
- 否則把該行拆成權杖、輸出到暫存檔。

第二趟：解析。對每個 ⟨*instruction*⟩（以換行權杖分隔）：

- 若以 `@` 權杖開頭：

  - 用到**新變數** → 配置 RAM、加入符號表；
  - 用到**既有變數或標籤** → 從符號表取回 RAM/ROM 位址；
  - 產生並輸出對應的 **A-指令**。
- 否則：拆解成 assignment、computation、condition 三部分，分別映射到 dest、comp、jump 的位元值，產生並輸出對應的 **C-指令**。

注意這把**第七週（機器碼格式）**與本週 **（詞法＋符號表＋解析）**完全接起來了：第二趟輸出的每一行，就是查第七週那三張表得到的 16 個位元。

本章重點

- Hack 的 EBNF：⟨*instruction*⟩ = (⟨*aInstruction*⟩ | ⟨*cInstruction*⟩) + newline；⟨*cInstruction*⟩ = [⟨*assignment*⟩] ⟨*computation*⟩ [⟨*jump*⟩]。
- LL 解析：左到右、上到下、只看接下來幾個權杖；掛不進去就是語法錯誤。
- Hack 是 LL(2) 不是 LL(1)：看到 `D` 得再看一個權杖才知道是指定還是運算。
- 實務捷徑：不建 CST，用權杖邏輯直接回答「A 還是 C? dest ／ comp ／ jump 是什麼?」
- 兩趟式組譯器：第一趟收標籤、產權杖檔；第二趟配變數、查表、輸出機器碼。

5　綜合練習題（附詳解）

練習 1：詞法分析

把下列各行 Hack 程式碼轉成權杖序列（標明 type 與 value）：(a) `@R13`；(b) `AM=M-1  // pop`；(c) `@counter`。

解答

(a) Token 1: (Keyword, `R13`)——注意 `R13` 是 **關鍵字**不是識別字！前面還有 Token 0: (Symbol, `@`)，最後 (Newline)。完整：**Symbol `@`、Keyword `R13`、Newline**。

(b) **Symbol `A`?** 不對——`AM` 要拆成 **兩個關鍵字權杖**：Keyword `A`、Keyword `M`、Symbol `=`、Keyword `M`、Symbol `-`、整數字面值 `1`、Newline。註解 `// pop` 被丟棄，不產生權杖。

(c) Symbol `@`、**識別字** `counter`（不含空白、非關鍵字、字母開頭）、Newline。

練習 2：關鍵字 vs 識別字

判斷下列字串被詞法器歸為哪一類，並說明理由：(a) `ARG`；(b) `ARGS`；(c) `R16`；(d) `2fast`；(e) `loop`。

解答

(a) **關鍵字**——在關鍵字列表中。

(b) **識別字**——不在列表中（雖然以 `ARG` 開頭）。詞法器必須**完整比對**，不能貪心地匹配前綴就停——這正是「`A` 與 `ARG` 是不同關鍵字，要小心區分」的意思。

(c) **識別字**——關鍵字只有 `R0` 到 `R15`，`R16` 不在其中。

(d) **都不是——詞法錯誤**。它不是整數字面值（含字母）、也不是識別字（不以字母開頭）。

(e) **識別字**——Hack 組語沒有 `loop` 關鍵字（它多半是使用者定義的標籤或變數名）。

練習 3：為什麼需要換行權杖?

(a) C 完全忽略換行，為什麼我們的 Hack 詞法器不能? 舉具體例子。(b) 為什麼 `(` 和 `)` 不是我們的符號權杖?

解答

(a) 沒有換行權杖時，權杖序列 Keyword `A`、Keyword `D`、Symbol `=`、Keyword `M` 有兩種讀法：一行的 `AD=M`，或兩行的 `A` 與 `D=M` ——語意完全不同（前者是一條指令，後者第一行甚至不合法）。換行權杖讓指令邊界明確。C 不需要，因為 C 用 `;` 與大括號標記語句邊界。

(b) 因為括號只出現在**標籤宣告 `(label)`** 中，而標籤宣告在**詞法階段就被整行移除**（加入標籤表、不輸出權杖）——解析器永遠不會看到括號，自然不需要它們的權杖。

練習 4：填符號表

對下列程式跑完整個組譯流程，寫出標籤表、變數表與最終輸出（以 `@` 位址形式）：

```
@x
M=0
(loop)
@x
M=M+1
@y
D=M
@loop
D;JGT
```

解答

第一趟（詞法）：逐行給機器碼行號，遇到標籤宣告就記錄並移除：行 0 `@x`、行 1 `M=0`、`(loop)` → 標籤表 `loop` → **2**（下一條指令的行號）、行 2 `@x`、行 3 `M=M+1`、行 4 `@y`、行 5 `D=M`、行 6 `@loop`、行 7 `D;JGT`。

第二趟（解析）： `@x`：兩表皆無 → 新變數，RAM 16。`@x`（第二次）：變數表命中 → 16。`@y`：兩表皆無 → 新變數，RAM 17。`@loop`：標籤表命中 → ROM 2。

標籤	ROM
`loop`	2

變數	RAM
x	16
y	17

輸出： `@16`、`M=0`、`@16`、`M=M+1`、`@17`、`D=M`、`@2`、`D;JGT`（共 8 條機器碼指令）。

練習 5：標籤行號的陷阱

某程式有 20 行組語，其中第 5、10、15 行（從 0 數起）是標籤宣告 `(a)`、`(b)`、`(c)`。三個標籤的 ROM 位址各是多少？

解答

標籤宣告不佔機器碼行號，且**前面每個被移除的標籤**都會讓後續行號前縮一格。`(a)` 在原始第 5 行，前面沒有標籤被移除，其後第一條指令的機器碼行號是 $5 - 0 = \mathbf{5}$。`(b)` 在原始第 10 行，前面已移除 1 個標籤，ROM 位址 $= 10 - 1 = \mathbf{9}$。`(c)` 在原始第 15 行，前面已移除 2 個標籤，ROM 位址 $= 15 - 2 = \mathbf{13}$。（實作上不用做這種減法：第一趟維護一個「目前機器碼行號」計數器，只在輸出**指令**行時遞增、標籤行不遞增即可。）

練習 6：BNF 判定

用「更好的整數」文法（$\langle posDigit \rangle$ ／ $\langle posNumber \rangle$ ／ $\langle number \rangle$）判定下列字串是否為合法 $\langle number \rangle$，合法者寫出展開過程：(a) `-305`；(b) `007`；(c) `0`；(d) `--5`。

解答

(a) **合法**：⟨*number*⟩ → ‘-’ ⟨*posNumber*⟩ → ‘-’ ⟨*posNumber*⟩ ⟨*posDigit*⟩ → ‘-’ ⟨*posNumber*⟩ ‘0’ ⟨*posDigit*⟩⋯ 等等——小心！305 的展開：⟨*posNumber*⟩ → ⟨*posNumber*⟩ ⟨*posDigit*⟩（尾巴 ‘5’）→ (⟨*posNumber*⟩ ‘0’) ⟨*posDigit*⟩（中間 ‘0’）→ (⟨*posDigit*⟩ ‘0’) ⟨*posDigit*⟩ → ‘3’ ‘0’ ‘5’。✓

(b) **不合法**：⟨*number*⟩ 的三個分支中，單獨 ‘0’ 只匹配一個字元；⟨*posNumber*⟩ 的第一個字元必來自 ⟨*posDigit*⟩（1–9），不能是 0。前導零被擋掉。✗

(c) **合法**：⟨*number*⟩ → ‘0’（第二分支）。✓

(d) **不合法**：‘-’ 只能出現一次（第三分支 ‘-’ ⟨*posNumber*⟩），且 ⟨*posNumber*⟩ 展不出 ‘-’。✗

練習 7：自己寫 BNF

寫一個 BNF 文法，權杖為 ‘0’ 與 ‘1’，使 ⟨*binary*⟩ 恰好匹配**無前導零的二進位正整數**（1、10、101 合法；0、01 不合法）。再用 EBNF 重寫成一行。

解答

BNF（其中一種寫法）：

$$\langle bit \rangle \;::=\; \text{`0'} \mid \text{`1'}$$

$$\langle binary \rangle \;::=\; \text{`1'} \mid \langle binary \rangle\ \langle bit \rangle$$

第一個字元只能經由基底情形 ‘1’ 產生，之後每個 ⟨*bit*⟩ 都可以是 0 或 1。

EBNF 一行版：

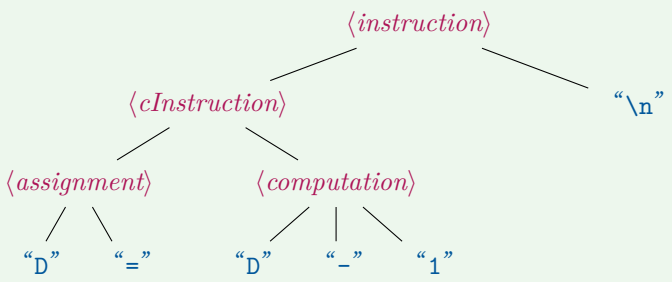
$$\langle binary \rangle \;::=\; \text{`1'},\ \{\text{`0'} \mid \text{`1'}\}$$

健全性檢查：1 ✓（重複零次）；10、101 ✓；0 ✗（必須以 ‘1’ 開頭）；01 ✗。

練習 8：畫 CST

用第 4 章的 Hack 文法，畫出 `D=D-1\n` 的 CST。

解答



要點：這條指令沒有 ⟨*jump*⟩（可選項缺席）；`D-1` 走的是 ⟨*computation*⟩ 的第四分支 (“A”|“D”|“M”), (“+”|“-”), “1”)，其中 - 是**直接的終端符號、不經過** ⟨*binaryOp*⟩（那是 `D-A` 型分支用的）。

練習 9：EBNF → BNF

把下列 EBNF 規則改寫成純 BNF（可新增非終端符號）：

$$\langle list \rangle \;::=\; \text{“(”},\ [\langle item \rangle,\ \{\text{“,”},\ \langle item \rangle\}],\ \text{“)”}$$

（一個括號包起來的、以逗號分隔的 item 列表，允許空列表。）

解答

$$\langle list \rangle \;::=\; \text{‘(’ ‘)’} \mid \text{‘(’ } \langle items \rangle \text{ ‘)’}$$
$$\langle items \rangle \;::=\; \langle item \rangle \mid \langle item \rangle \text{ ‘,’ } \langle items \rangle$$

技巧總結：**可選項** $[X] \to$ 寫兩條規則（有 X 與沒 X）；**重複** $\{X\} \to$ 新增遞迴非終端符號。這也示範了為什麼 EBNF 沒有增加表達能力——每個構造都能機械地展開回 BNF。

練習 10：LL(1) vs LL(2)

(a) 解釋為什麼解析器讀到 `DM=M+D` 的第一個 `D` 時，只看這一個權杖無法決定 CST 的形狀。(b) 往前看一個權杖後，哪些「下一個權杖」意味著 ⟨*assignment*⟩？哪些意味著 ⟨*computation*⟩？(c) `0;JMP` 需要 lookahead 嗎？為什麼？

解答

(a) `D` 既可以開啟 ⟨*assignment*⟩（如 `D=...`、`DM=...`），也可以自己就是 ⟨*computation*⟩（如 `D;JGT` 或單純 `D+1`）。兩種情況掛的節點不同，單看 `D` 無法分辨。

(b) 下一個權杖是 `M` 或 `=` → ⟨*assignment*⟩；是 `+`、`-`、`&`、`|`、`;` 或 `\n` → ⟨*computation*⟩。

(c) **不需要**。`0` 在文法中只能是一整個 ⟨*computation*⟩（⟨*assignment*⟩ 不可能以 `0` 開頭），所以第一個權杖就唯一決定了結構。需要兩個權杖的只有 `A` ／ `D` ／ `M` 開頭的情況——這正是 Hack 是 LL(2) 而非 LL(1) 的原因。

練習 11：作用域堆疊

用第 2 章的 C 程式回答：(a) 在第 7 行（第一個 for 內的 `printf`），查詢 `i` 得到什麼型別？查詢 `foo` 呢？(b) 為什麼第 18 行印出 `'a'`？(c) 為什麼第 19 行編譯錯誤？

解答

(a) 此時堆疊（頂到底）：表 3、表 2、表 1。查 `i`：表 3 命中 → **`int`**（第一個 for 自己宣告的計數器，遮蔽外層的 `char i`）。查 `foo`：表 3 沒有 → 往下到表 2 命中 → **`double`**。

(b) 第 18 行時表 3、表 4 都已被 pop，堆疊只剩表 2、表 1。查 `i` 命中表 2 的 `char i = 'a'`——迴圈裡對「它們自己的 `i`」做的所有遞增都發生在早已消失的內層變數上。

(c) `temp` 只存在於表 4（第二個 for 的作用域），該表在第 16 行已被 pop。查遍剩下的表都找不到 `temp` → 未宣告識別字，編譯錯誤。

練習 12：端到端組譯

把下列程式完整組譯成 16 位元機器碼（用第七週的編碼表）：

```
@5
D=A
(wait)
@wait
0;JMP
```

解答

第一趟：行 0 `@5`、行 1 `D=A`、標籤 `wait` → ROM **2**、行 2 `@wait`、行 3 `0;JMP`。

第二趟：

- `@5`：整數字面值，A-指令：opcode 0 + 5 的 15 位元二進位 → `0000000000000101`。
- `D=A`：comp `A`（a=0, c=110000）、dest `D=`（010）、無 jump（000）→ `111 0110000 010 000` = `1110110000010000`。
- `@wait`：標籤表命中 → `@2` → `0000000000000010`。
- `0;JMP`：comp `0`（0101010）、dest 000、jump `JMP`（111）→ `111 0101010 000 111` = `1110101010000111`。

四條機器碼指令，變數表保持空（本程式沒有變數）。這就是你的組譯器對這個輸入應產生的**精確輸出**。

A 附錄：速查表

A.1 Hack 權杖分類

類別	成員	C 的對應
關鍵字	`A D M; JGT JEQ JLT JGE JNE JLE JMP; SCREEN KBD SP LCL ARG THIS THAT; R0–R15`	keywords（`int`、`for`…）
符號	`@ + - & | = ; !`	punctuators ／ operators
整數字面值	十進位 0–32767	constants
識別字	無空白、非關鍵字、字母開頭	identifiers（`printf`…）
換行	`\n`	（C 忽略）
不產生權杖：空白、註解、標籤宣告 `(label)`（含括號本身）		

A.2 組譯器流程圖

A.3 EBNF 記號速查

記號	意義
::=	定義為
|	或
⟨*X*⟩	非終端符號
‘x’ 或“x”	終端符號（權杖）
( )	分組
[ ]	可選（0 或 1 次）
{ }	重複（0 次以上）
$A - B$	匹配 A 但排除 B（B 須有限）

A.4 名詞中英對照

英文	中文	英文	中文
lexing / lexer	詞法分析／詞法器	context-free grammar	上下文無關文法
token / lexeme	權杖／詞素	Backus-Naur Form (BNF)	BNF 範式
parsing / parser	解析／解析器	terminal symbol	終端符號
syntax / semantics	語法／語意	non-terminal symbol	非終端符號
keyword	關鍵字	parse tree / CST	解析樹／具體語法樹
identifier	識別字	abstract syntax tree	抽象語法樹
integer literal	整數字面值	ambiguity	歧義
symbol table	符號表	parser generator	解析器產生器
label / variable	標籤／變數	LL(k) parsing	LL(k) 解析
scope	作用域	lookahead	往前看
shadowing	遮蔽	recursive descent	遞迴下降
intermediate representation	中介表示法	two-pass assembler	兩趟式組譯器

A.5 參考資料

- John Lapinskas, *Compiler concepts: Lexing ／ Symbol tables ／ Parsing ／ A parser for Hack assembly*（8-1 至 8-4），COMSM1302, University of Bristol.
- Nisan & Schocken, *The Elements of Computing Systems*（nand2tetris），Ch. 6（The Assembler）.
- Aho, Lam, Sethi & Ullman, *Compilers: Principles, Techniques, and Tools*（「龍書」——詞法、文法與解析的標準教科書）.
- ISO/IEC 14977, *Extended BNF*（EBNF 標準）.
- C11 標準（ISO/IEC 9899:2011）§6.4 Lexical elements （C 的權杖定義）.
- Clang 原始碼 `clang/lib/Parse/`（工業級手寫遞迴下降解析器的實例）.