# 為什麼 Recursive Descent Parsing 需要 FIRST set？

Recursive Descent Parsing 的「不回頭」決策原理

lookahead　FIRST　FOLLOW　LL(1)

# 核心想法：看一個 token 就要選路

Predictive recursive descent：不靠 backtracking 也能做正確選擇

「不回頭」的關鍵：一眼看 token 就要做決定

**看一個 lookahead token**

例如 currentToken = "(" 或 ";"

**查 FIRST set 決定分支**

token ∈ FIRST(α) → 選 A → α；否則試另一分支

**匹配 / 前進（或報錯）**

都不符合 → unexpected token → syntax error

直覺

- Recursive Descent的每個 function 對應一個 nonterminal。
- 遇到「A → α | β」時，必須立刻決定選 α 還是 β。
- 若只用 1 個 lookahead token 就能決定，這類 grammar 常稱為 LL(1)。
- 因此需要 FIRST（必要時再加上 FOLLOW）來支援「唯一決策」。

如果兩條路「可能以同一個 token 開頭」→ 只看一眼就選不出來 → 會需要改寫 grammar（left factoring 等）

# 例子：遇到兩條分支，怎麼選？

Grammar

```
function_definition_or_declarators
    : function_definition_without_ID
    | rest_of_declarators
```

問題

我們在寫
Function_definition_or_declarators()：
現在有兩條路可以走，要走哪一條？

路 1：function_definition_without_ID → 路 2：rest_of_declarators

**答案：看「目前的 token」屬於哪個 FIRST set。**

- 先算：FIRST(function_definition_without_ID) 與 FIRST(rest_of_declarators)
- current token ∈ FIRST(分支) → 選那條分支
- 兩個都不符合 → unexpected token → error

# FIRST set 是什麼？

「某段符號串」可能出現的第一個 terminal token

**定義**

FIRST(α) = 可能出現在「α 推導出來的字串」最前面的所有 terminal tokens

- 若 α 的最左邊是 terminal：FIRST(α) 就是那個 terminal。
- 若最左邊是 nonterminal：看它能推導出哪些開頭 token。
- 若 α 可能推導出 ε（空字串）：則 ε ∈ FIRST(α)。

**小例子**

**例 1：α = "(" Expr ")"**

`FIRST(α) = { "(" }`

**例 2：B → "b" | ε**

`FIRST(B) = { "b", ε }`

→ 因為 B 可能「什麼都不吃」就結束

**例 3：α = B C（且 B 可能是 ε）**

FIRST(BC) 會需要把 FIRST(C) 也考慮進來（下一頁）

# 用 FIRST set 做分支：if / else 就能寫 parser

分支的 FIRST

**對於：A → α | β**

先算：Aα = FIRST(α)、Aβ = FIRST(β)

Aα（路 1）
current token ∈ FIRST(α) → 選 α

Aβ（路 2）
current token ∈ FIRST(β) → 選 β

你的例子

```
A = { LP }  // "("
B = { LB, COMMA, SEMI_COLON }  // "["  ","  ";"
```

（示意）單一 lookahead 的 if/else 分支

```
1 function
parseFunction_definition_or_declarators() {
2   if (lookahead == LP) {
3     parse_function_definition_without_ID();
4   } else if (lookahead == LB || lookahead ==
COMMA || lookahead == SEMI_COLON) {
5     parse_rest_of_declarators();
6   } else {
7     error("unexpected token");
8   }
9 }
```

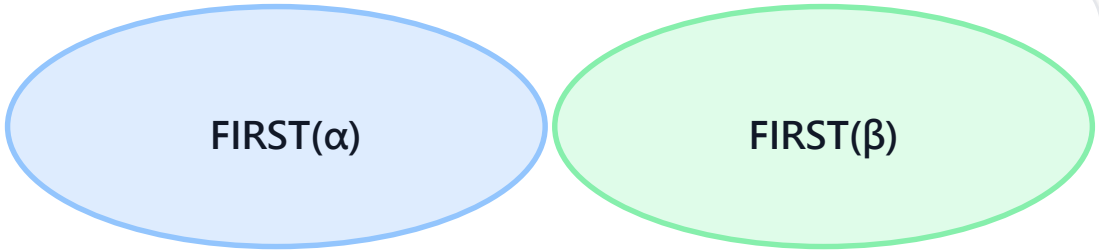
**重點：A 與 B 不能有交集（否則選路會歧義）**

# LL(1) 的直覺檢查：什麼時候「看一眼」就夠？

**必要條件（常用版本）**

- 同一個 nonterminal X 的不同 alternatives，其 FIRST 集合要互斥（disjoint）。
- 若某個 alternative 可能推出 ε（nullable）：還要確保 FIRST(X) 與 FOLLOW(X) 不衝突，避免「空走」造成選路不明。

**視覺化**




（理想狀況：不重疊）

**不滿足怎麼辦？**

- Left factoring：把共同前綴提到前面。
- 消除 left recursion（常見的 expression grammar）。
- 必要時提高 lookahead（LL(k)）或改用 LR/LALR 工具。

# 麻煩情況 1：B 可能是 ε（空字串）

因此 FIRST(BC) 不能只看 FIRST(B)

**情境**

A → B C | ...
（假設 B 可推出 ε）

**直覺**

- 如果 B 走 ε：那麼輸入的第一個 token 其實會交給 C 來決定。
- 所以 FIRST(BC) 必須包含 FIRST(C)。

**規則（常用寫法）**

$$FIRST(BC) = FIRST(B) - \{\varepsilon\} \cup FIRST(C) \quad (\text{當 } \varepsilon \in FIRST(B) \text{ 時})$$

你可以把它想成：

B C

若 B = ε → 直接由 C 接手

# 麻煩情況 2：B、C 都可能是 ε → 需要看 FOLLOW(A)

因為走 BC 可能「完全不吃 token」

**情境**

$$A \to B\ C \mid \ldots$$

$$B \Rightarrow^* \varepsilon \quad 且 \quad C \Rightarrow^* \varepsilon$$

此時走「BC」這條路，第一個真正看到的 token 可能是 A 後面的東西。

**FOLLOW 的直覺**

FOLLOW(A) = 在合法句子中，A 後面「可能立刻出現」的 terminal tokens

用白話說：
如果 A 本身可能「空掉」，就要看下一個可能是什麼，才能判斷是否可空走。

**實作上常用：Predict set**

對 production A → α，常用一個集合來判斷「何時可選這條 production」：

$$\text{Predict}(A \to \alpha) = \text{FIRST}(\alpha) \cup \text{FOLLOW}(A) \quad （當 \ \alpha \ 可推出 \ \varepsilon \ 時）$$

→ 也就是：α 先吃得到什麼就用 FIRST；若 α 會空掉，就用 FOLLOW 來決定可不可以空走。

# 小結：把概念落到實作流程

nullable → FIRST → FOLLOW → if/else（predictive recursive descent）

### 你要記住的 4 件事

- Recursive descent 若不回頭，遇到 choice（|）必須靠 lookahead 唯一決策。
- FIRST(α) 告訴你：走 α 時「第一個可能看到」的 token 是哪些。
- 若 α 可能推出 ε：選路時要把 FOLLOW(A) 也納入（Predict set）。
- LL(1) 的直覺檢查：alternatives 的 FIRST 不重疊；nullable 時 FIRST/FOLLOW 不衝突。

### 建議教學流程（可當作作業步驟）

**Step 1　找出 nullable（可推出 ε 的 nonterminal）**

**Step 2　計算 FIRST（含串接規則）**

**Step 3　計算 FOLLOW（用於 nullable 相關判斷）**

Step 4　為每個 nonterminal 產生 if/else：
if lookahead ∈ Predict(A→α) → 走 α