極簡核心與功能擴張：語言應保持多小，又能成長到多大？

Minimal Cores and Feature Expansion: How Small Should a Language Remain, and How Far Should It Grow?

v1.0 / 2026-07-30 / Neo.K / 公開版／完成第二輪查核

https://thisoneisneok.com/html/papers/pldst-006.html

英文名稱： Minimal Cores and Feature Expansion: How Small Should a Language Remain, and How Far Should It Grow?

系列： Programming Language Designer Style Taxonomy（PLDST）

文件編號： PLDST-006

版本： v1.0

日期： 2026-07-30

作者： Neo.K

文件狀態： 公開版／完成第二輪查核

SECTION

摘要

「保持語言簡單」幾乎是所有程式語言設計者都願意公開支持的價值，但不同設計者所稱的簡單，可能指向完全不同的系統結構。

Scheme 以極少量、可自由組合的規則支撐多種程式設計範式；Oberon 透過刪除 Modula-2 中被認為非基本或交互成本過高的功能，同時加入少量高覆蓋能力，追求概念經濟；Lua 將表、函數、閉包與元表等少數機制組成可嵌入、可擴展的語言；Go 刻意限制語言特徵，將閱讀、建置、工具與大型組織協作視為核心設計成本，但在泛型等功能上仍接受經長期驗證後的擴張；Clojure 以 Lisp 資料表示、不可變資料、抽象與宿主互操作組成「簡單而非只是容易」的系統，同時維持高度保守的核心功能准入；C++ 則以一般機制、硬體映射、零額外成本與長期相容性支持廣泛領域，形成一種「少數深層原則支撐大型表面」的設計，而不是小型語

因此，語言極簡不能只用關鍵字數量、語法產生式數量、標準頁數、編譯器程式碼行數或初學範例長度來判斷。

本文提出 核心—擴張配置模型（Core–Expansion Allocation Model, CEAM），將語言表示為：

> cal-L
> =
> (
> K,
> X,
> B,
> I,
> T,
> E,
> G
>)

其中：

- K：Kernel，核心語義與基本構造；
- X：Extension mechanisms，擴展機制；
- B：Built-in features，內建功能；
- I：Interoperability，宿主與外部互操作；
- T：Tooling，工具所承擔的語言能力；
- E：Ecosystem，程式庫、框架與慣例；
- G：Governance，功能准入、相容性與淘汰制度。

本文區分六種「小」：語法小、語義小、規格小、實作小、特徵交互圖小，以及有效語言小；並提出六種典型核心—擴張風格：刪減式極簡、生成式極簡、嵌入擴展式極簡、正交組合式極簡、組織工程式極簡與一般機制式擴張。

核心命題為：

> 【小核心
> not⇒
> 小型有效語言】

以及：

> 【新增功能
> not⇒
> 總體複雜度必然增加】

若一個新機制能取代大量不一致的程式庫慣例、手寫模板、動態逃生口與工具補丁，它可能增加核心表面，卻降低整個語言系統的有效複雜度。相反地，一個看似不加入功能的語言，也可能把複雜度轉移到程式庫碎片、程式碼生成、反射、宿主語言與組織慣例中。

因此，真正的問題不是「語言應該有多少功能」，而是：

> 哪些能力必須在核心中保持一致，哪些應由可組合機制生成，哪些應留給程式庫、宿主或工具，以及語言用什麼制度阻止局部便利累積成全域複雜度？

關鍵詞： 程式語言設計、極簡主義、核心語言、功能擴張、正交性、可擴展語言、Scheme、Oberon、Lua、Go、Clojure、C++

SECTION

一、語法小

語法小通常表示關鍵字少、產生式少、表面形式統一、特殊標點少與語法糖有限。它能降低 Parser 與基本記憶負擔，卻可能把複雜度轉入巨集展開、名稱解析、隱式語義、動態查找或程式庫慣例。

因此：

> Size_(syntax)downarrow
> not⇒
> Complexity_(meaning)downarrow

SECTION

二、語義小

語義小表示語言依靠少量基本概念解釋大量行為，例如 Lambda、Lexical scope、Message passing、Table、Pattern、Trait、Module 或 Value／Identity 分離。

語義小通常比語法小更重要，因為使用者能以少量規則推導大量結果。但單一基本元件若過度強大，也可能難以分析、最佳化與工具化。

SECTION

三、規格小

規格小表示完整語言定義能以較小文件精確表達。它可能來自機制少、規則正交、不定義大量平台行為、把能力留給程式庫，或只有單一主要實作。

規格小不等於使用者世界小。未寫入核心規格的行為，可能被分散到 FFI、ABI、套件、Build system、社群慣例與實作特有功能。

SECTION

四、實作小

實作小表示編譯器或直譯器體積小、Runtime 小、移植容易、可信計算基較小，且少數人可以完整理解。

Lua 與 Oberon 都將小型、可理解實作視為重要設計結果，但理由不同：Lua 重視嵌入、可攜與小型 Runtime；Oberon 則將小型語言、編譯器與作業系統視為同一完整系統工程的一部分。[R2][R3]

SECTION

五、交互作用小

令功能集合為：

> $F=\{f_1,f_2,\ldots,f_n\}$

語言具有功能交互圖：

$$G_F=(F,E_F)$$

其中邊表示語義、語法、型別、Runtime、工具或相容性交互。其負擔可啟發式表示為：

$$C_F = \Sigma \boxtimes c(f\boxtimes) + \lambda\Sigma_{((i,j)\in E_F)}c(f\boxtimes,f\boxtimes) + \mu C_{(\geq 3)}$$

$C_{(\geq 3)}$ 表示三個以上功能形成的高階交互。此公式不是嚴格定律，而是提醒：新增功能的成本還包括它與既有機制的規格、測試、診斷及演化關係。

SECTION

六、有效語言小

使用者真正使用的語言不是只有核心規格：

$$L_{(effective)} = K + Libraries + Macros + Tools + Interop + Frameworks +$$

小核心可能形成大型巨集方言、Framework 專屬語言、宿主 API 驅動語言、大量編譯器插件與組織內部規則。因此，有效語言是否小，必須在真實開發活動中評估。

SECTION

七、什麼應屬於核心

一項能力較適合進入核心，通常因為它：

- 影響程式基本意義；
- 必須跨程式庫一致；
- 需要編譯器或 Runtime 特權；
- 無法由現有機制可靠表達；
- 若分散實作會造成安全或互操作問題；
- 具有高覆蓋率；
- 能取代大量不一致機制；
- 需要全語言工具理解。

SECTION

八、什麼應留在程式庫

較適合程式庫的能力通常可由普通語言機制表達、只服務特定領域、仍在快速演化、不需要特殊語法、可由不同方案競爭，並可獨立版本化。

但程式庫方案若無法提供效能、診斷、語法整合、安全保證或跨套件一致性，就可能產生升格核心的壓力。

SECTION

九、什麼應交給工具

工具適合承擔格式、Lint、遷移、重構、程式碼生成、非核心靜態分析與組織政策。工具能避免核心膨脹，但若某項工具成為所有使用者不可缺少的語義前提，它實際上已成為有效語言的一部分。

SECTION

十、什麼應交給宿主互操作

嵌入式或宿主語言可以利用外部平台提供 I/O、GUI、作業系統、程式庫、網路與原生效能。Lua 與 Clojure 都以互操作降低重新建立完整世界的需要，但代價是宿主洩漏、平台綁定、錯誤模型不一致、資料轉換與多套慣例。

SECTION

十一、刪減式極簡

代表思路是：

> 先問哪些既有功能應被刪除，再問必須新增什麼。

Wirth 在 Oberon 回顧中明確說明，其策略是先決定從 Modula-2 省略哪些功能，再決定哪些新增能力真正必要；目標是提高能力的同時降低複雜度。[R2]

典型特徵：核心封閉、功能准入嚴格、重視完整實作、偏好少量基本機制，並願意犧牲相容性修正舊設計。

優勢是概念一致、教學清楚、編譯器與系統可理解、特徵交互少。風險是特定領域便利不足、使用者轉向非標準擴展、生態吸引力有限，以及設計者可能低估不同使用情境。

SECTION

十二、生成式極簡

代表思路是：

> 以極少規則構成可生成多種表達形式的語言。

R7RS 對 Scheme 的自我描述強調：少量形成表達式的規則，在自由組合下足以形成支援多種主要程式設計範式的實用語言。[R1]

典型機制包括 Lambda、First-class procedures、Lexical scope、資料表示、Hygienic macros 與 Proper tail calls。

優勢是小型語義核心、高組合性與 DSL 能力；風險是巨集建立局部方言、多實作與標準分層帶來互操作負擔，以及使用者需要理解高階抽象。

Scheme 標準程序將 R7RS 分為 Small Language 與 Large Language 工作方向，也表明小核心與完整現代程式庫需求之間需要制度化分工。[R7]

SECTION

十三、嵌入擴展式極簡

代表思路是：

> 語言本身保持小型，以宿主 API 與可擴展語義適應不同領域。

Lua 從一開始即以簡單、小型、可攜、快速與易嵌入為目標；其表結構可以表示模組、物件、紀錄、陣列、集合等多種構造，元表則提供可擴展語義。[R3]

優勢是易嵌入、易移植，以少量機制適應多領域。風險是宿主與腳本責任邊界複雜、大型程式可能缺少強制結構、不同嵌入環境形成不同有效語言，以及動態擴展提高工具分析難度。

SECTION

十四、正交組合式極簡

代表思路是：

> 每一項功能保持相對簡單，並允許它們自由組合。

Go 官方對泛型設計的說明，將降低複雜度的方法表述為讓單項功能簡單、獨立、正交，並透過自由組合增加收益。[R4]

優勢是預測性高、工具較易統一、團隊使用的語言子集較一致。風險是缺少常見抽象時產生重複、使用者建立程式碼生成或反射替代，以及過度保守把成本推入程式庫。

SECTION

十五、組織工程式極簡

代表思路是：

> 語言小不是終極美學，而是降低大型組織協作、閱讀、建置與維護成本的手段。

Go 的原始設計問題包括慢建置、不受控依賴、不同程式設計者使用不同語言子集、程式難讀、更新與工具困難。[R4]

因此，Go 的極簡不只保護編譯器或初學者，也保護 Code review、團隊溝通、建置系統、長期維護與工具鏈。Go 官方對實驗與簡化的回顧也承認，可以增加發行物內部程式碼與複雜度，換取編寫 Go 程式的整體簡化。[R8]

SECTION

十六、一般機制式擴張

代表思路是：

> 與其保持小表面，不如提供能覆蓋未知需求的一般機制，但要求它們符合若干深層原則。

C++ 的設計追求直接硬體映射、零額外成本抽象、一般性、型別豐富、多範式與長期相容。Stroustrup 直接承認 C++ 是大型且複雜的語言，同時主張良好 C++ 主要依靠少數基本技術；舊語言為相容而保留的附帶功能，會干擾理解。[R6]

這是一種深層原則相對集中、表面與歷史功能廣泛的風格。它能服務多種領域並保留硬體效率，代價則是大型表面、特徵交互、不同團隊子集與相容性累積。

SECTION

十七、簡單不等於熟悉

Clojure 官方將自身描述為由一組有用功能構成的簡單、連貫且有力的工具，並以不可變資料、函數、值與 Identity／State 分離等原則降低交纏。[R5]

這裡的簡單更接近概念不交纏、資料與行為分離、使用少量抽象、依靠宿主 JVM，而非第一次使用時的熟悉感。

因此：

> Simple
> ≠
> Familiar
> ≠
> EasyToStart

SECTION

十八、保守核心與宿主擴張

Clojure 治理回顧明確形容 Rich Hickey 對加入語言功能極為保守，核心團隊會以此篩選提案。[R5]

Clojure 的擴張更多依靠函數與資料、Macro、Protocol、JVM interop、Library 與 Spec。這降低核心變更頻率，卻可能把複雜度放入 Macro 生態、Java 互操作、依賴工具、動態資料契約與程式庫慣例。

SECTION

十九、新領域

語言進入新領域後，原核心可能缺乏並行、非同步、GPU、分散式、模組、套件、安全與外部互操作。完全拒絕擴張，可能使使用者建立非標準替代。

SECTION

二十、重複解法

當所有使用者重複建立同一模式時，可能意味核心缺少表達能力、程式庫抽象不足、工具無法理解或安全保證需要語言特權。

例如泛型增加語言複雜度，卻也可能取代重複函式、動態轉型、程式碼生成、手寫容器與多套不一致 API。Go 在接受泛型時，官方文件明確承認它會增加語言複雜度，同時試圖用簡單、正交且可組合的設計控制成本。[R4]

SECTION

二十一、工具與診斷

某項能力若只存在於程式庫，編譯器可能無法給出精確錯誤、最佳化、驗證安全、支援重構或建立跨模組分析。將它提升為語言功能可能增加核心，卻改善整體工具體驗。

SECTION

二十二、安全與互操作

當不同程式庫各自定義資源生命週期、非同步、錯誤、序列化與外部函式時，系統可能缺乏一致邊界。核心功能有時是為了讓整個生態共享同一安全或互操作模型。

SECTION

二十三、歷史相容性

語言功能也可能不是因為仍受偏好，而只是不能移除。因此，功能數量增長要區分 active expansion、compatibility retention、temporary experiment、deprecated fossil、library migration 與 implementation extension。

SECTION

二十四、功能效用

對候選功能 f，定義：

> U(f)
> =
> Coverage
> +
> Safety
> +
> Composability
> +
> Toolability
> +
> Interoperability
> +
> Elimination

其中 Elimination 表示它能否刪除重複、逃生口與不一致替代方案。

SECTION

二十五、功能成本

> K(f)
> =
> Learn
> +
> Specify
> +
> Implement
> +
> Interact
> +
> Diagnose
> +
> Migrate
> +
> Govern
> +
> Maintain

不能只計算實作這項功能需要多少程式碼。

SECTION

二十六、准入條件

候選功能應至少滿足：

> U(f)-K(f)>Threshold

且現有核心組合不能以更低總成本提供同等能力。

此外還要回答：能否由程式庫或工具完成、是否需要 Runtime 特權、是否破壞相容性、是否形成高階交互、是否可被教學與診斷、是否有真實實作與使用證據，以及若失敗能否撤回。

SECTION

二十七、候選功能成熟階段

Problem
→ Library experiment
→ Tool support
→ Prototype
→ Language experiment
→ Evidence
→ Stable core
並非每項功能都必須走完整路徑，但成熟階段能減少過早永久化。

SECTION

二十八、貧困式極簡

為了功能數量小而拒絕必要能力，導致重複程式、樣板、不安全逃生口、程式碼生成與不可攜擴展。這是小而不足，不是概念經濟。

SECTION

二十九、隱藏式極簡

語言表面很小，但大量行為依靠隱式轉換、Runtime 魔法、反射、搜尋順序、宿主狀態與工具生成。使用者看到的語法少，但預測程式需要更多隱藏規則。

SECTION

三十、巨集碎片化

強大巨集能讓核心小，也可能產生局部語法、非標準控制流、工具失效、團隊知識孤島與跨專案閱讀困難。生成式極簡需要衛生、Expansion 可見性、工具 API、慣例與語義邊界。

SECTION

三十一、程式庫無政府狀態

若所有問題都丟給程式庫，可能形成多套錯誤模型、非同步模型、資料表示、依賴衝突、安全品質與版本分裂。

SECTION

三十二、宿主洩漏

嵌入式極簡若過度依賴宿主，使用者仍需同時理解宿主記憶體、型別、執行緒、錯誤、FFI 與部署平台。此時語言本身小，但使用系統不小。

SECTION

三十三、便利功能堆疊

每個功能單獨合理，合在一起卻重複、衝突、形成多種做法並增加教學與工具負擔。

SECTION

三十四、特例取代一般機制

大量領域專用特徵可能讓語言快速回應需求，但降低正交性、可組合性、規格清晰與長期演化能力。

SECTION

三十五、相容性累積

功能一旦進入穩定語言，就可能永久增加 Parser、編譯器、Runtime、文件、測試、教學、安全分析與新功能交互負擔。

SECTION

三十六、委員會聯合體

不同群體各自取得所需功能，最後形成所有人都有一點、沒有人理解全部、缺少整體刪減，以及功能間缺乏共同哲學。

SECTION

三十七、表達力超過工具能力

語言可表達的內容若大幅超過型別分析、IDE、Refactoring、Debugger 與 Build system 的理解能力，使用者會為理論表達力支付日常工程成本。

SECTION

三十八、核心表面向量

> K_s
> =
> (
> Keywords,
> Forms,
> Operators,
> Declarations,
> ControlForms
>)

它只適合表面比較。

SECTION

三十九、核心語義向量

> K_m
> =
> (
> Binding,
> Evaluation,
> Types,

評估每一類有多少獨立規則。

SECTION

四十、交互密度

> D_F
> =
> (2|E_F|)/(|F|(|F|-1))

此式只適合相對比較，實際交互邊需明確定義。更重要的是標記高風險交互，例如 Generic × overload、Macro × name resolution、Exception × destructor、Async × cancellation、Reflection × type safety、Inheritance × overload 及 Module × initialization。

SECTION

四十一、有效語言半徑

> R_(eff)(task)
> =
> Core
> ∪
> Libraries
> ∪
> Tools
> ∪
> Conventions
> ∪
> Interop

不同任務具有不同有效語言半徑。Lua 嵌入遊戲與用 Lua 建立大型獨立系統，所需半徑不同；C++ 裸機與大型企業 Framework 也不同。

SECTION

四十二、可完整理解性

應分開詢問：使用者是否理解基本語義、實作者是否理解完整編譯器、維護者是否理解特徵交互、組織是否能限制使用子集，以及工具是否能建立完整模型。

SECTION

四十三、Scheme

核心策略：極少規則自由組合
主要擴展：程序、巨集、程式庫
保護對象：語義經濟與可生成性
主要風險：方言、巨集與標準分層

SECTION

四十四、Oberon

核心策略：先刪除，再加入必要高覆蓋機制
主要擴展：模組、型別延伸、完整系統
保護對象：概念、教學、實作完整性
主要風險：領域便利與生態規模

SECTION

四十五、Lua

核心策略：小型嵌入語言與可擴展語義
主要擴展：Table、Metatable、C API、程式庫
保護對象：可攜、嵌入、實作體積
主要風險：宿主洩漏與大型程式結構

SECTION

四十六、Go

核心策略：少量正交功能服務組織工程
主要擴展：標準程式庫、工具、受控語言提案
保護對象：閱讀、建置、團隊一致性
主要風險：缺少抽象時的重複與外部替代

SECTION

四十七、Clojure

核心策略：資料、函數、不可變性與宿主互操作
主要擴展：Macro、Protocol、JVM、生態
保護對象：概念不交纏與動態開發
主要風險：學習門檻、巨集與宿主雙重世界

SECTION

四十八、C++

核心策略：一般機制、零額外成本、相容性
主要擴展：語言與程式庫共同演化
保護對象：廣泛領域、硬體效率、既有生態
主要風險：大型表面與高階交互

SECTION

四十九、極簡的真正分類

PLDST 不使用單一極簡分數，而輸出：

語法極簡
語義極簡
實作極簡
交互極簡
組織極簡
生態極簡
治理極簡

一位設計者可能高語義極簡、低表面極簡；或高組織極簡、低生態極簡。

SECTION

五十、功能擴張偏好

記錄設計者偏好內建、一般機制、程式庫、巨集、工具、宿主、標準委員會，或實驗後穩定。

SECTION

五十一、刪除能力

設計風格不只看加入什麼，也看是否願意刪除、何時刪除、是否建立遷移、是否因相容性停止刪除，以及是否只在新語言重啟時刪除。

SECTION

五十二、核心邊界哲學

PLDST 個案需回答：

> 哪些能力設計者認為只有語言核心能合法、一致地提供？

此問題比功能多寡更能揭示風格。

SECTION

五十三、輸入

designer
language
version_or_period
candidate_feature
core_spec
library_ecosystem
tooling
governance_documents

SECTION

五十四、分析管線

重新網路搜尋
→ 核心構造抽取
→ 擴展通道抽取
→ 功能交互圖
→ 有效語言半徑
→ 功能准入與拒絕紀錄
→ 相容性化石檢查
→ 核心／程式庫／工具／宿主配置
→ 反例搜尋
→ 第二輪校對
→ 風格報告

SECTION

五十五、功能分析 JSON

```
{
 "feature": "generics",
 "candidate_locations": [
  "language core",
  "library",
  "code generation",
  "tooling"
 ],
 "utility": {
  "coverage": "high",
  "eliminates_duplication": "high",
  "toolability": "high"
 },
 "cost": {
  "specification": "high",
  "implementation": "high",
  "interaction": "medium-to-high",
  "learning": "medium"
 },
 "style_interpretation": "controlled expansion after evidence"
}
```

SECTION

五十六、SKILL 禁止事項

不得以關鍵字數直接判定簡單、以標準頁數判定複雜、忽略程式庫與宿主、將小 Runtime 等同小語義、將新增功能一律寫成退化、將拒絕功能一律寫成美德、將程式庫替代視為零成本、忽略相容性化石或不分析功能交互。

SECTION

五十七、保持核心稀缺，而不是任意貧乏

核心功能應具有高覆蓋、高一致性與高工具價值。

SECTION

五十八、先找生成機制，再加專用功能

若少量一般機制能自然、可診斷地表達需求，優先使用一般機制。但「可以勉強表達」不等於「適合交給使用者反覆表達」。

SECTION

五十九、程式庫方案必須接受有效語言審計

不能因功能不在語法中，就宣稱語言保持簡單。

SECTION

六十、擴展機制必須有邊界

強大擴展應提供名稱衛生、權限限制、展開可見性、工具支援、相容規則與版本管理。

SECTION

六十一、新增功能必須說明它刪除了什麼

優質提案不只列出能做什麼，也列出可取代的樣板、不安全慣例、重複程式庫、反射、程式碼生成與特例。

SECTION

六十二、每項穩定功能都是長期負債契約

功能准入必須計算未來的相容、教學、多實作、工具、安全與新功能交互。

SECTION

六十三、允許語言分層成長

可以建立 small core、standard libraries、optional profiles、experimental extensions、domain packages 與 tooling conventions，但層級與保證必須清楚。

SECTION

六十四、核心邊界具有社會性

什麼被稱為核心，常取決於發行方式、標準文件、套件管理、預設安裝、公司支持與教育，不是純技術邊界。

SECTION

六十五、功能數量難以客觀比較

一個功能可以拆成多個，或多個功能可被描述成一個一般機制，因此功能計數高度依賴分類方式。

SECTION

六十六、正交性不是完全可測

設計者常稱功能正交，但實際仍可能在型別、解析、Runtime 與工具中產生交互，需要用具體決策與測試評估。

SECTION

六十七、小型語言不一定適合大型生態

語言可以是教學工具、嵌入式腳本、研究核心、完整平台或國際標準。不同使命需要不同核心邊界。

「語言應該保持多小」沒有一個脫離用途的答案。

Scheme 展示生成式小核心；Oberon 展示刪減與完整系統一致性；Lua 展示嵌入與宿主擴展；Go 展示正交性與組織工程；Clojure 展示概念簡單、宿主互操作與保守准入；C++ 展示少數深層原則如何支撐大型、相容且多領域的語言。

本文提出：

> cal-L
> =
> (
> K,
> X,

並主張語言大小至少要分成：

> 【語法
> +
> 語義
> +
> 規格
> +
> 實作
> +
> 交互
> +
> 有效語言】

因此：

> 【SmallCore
> not⇒
> SmallSystem】

而：

> 【FeatureAddition
> not⇒
> NetComplexityIncrease】

PLDST 不再將設計者簡單分為「喜歡加功能」與「喜歡刪功能」。更成熟的風格判定應回答：

- 他保護哪一種核心稀缺性？
- 他用什麼機制讓語言成長？
- 他願意把能力留給程式庫、工具或宿主到什麼程度？
- 他如何評估功能交互？
- 他是否願意刪除既有功能？

- 他如何處理相容性化石？
- 他認為何種能力只有核心才能一致提供？
- 新功能需要什麼證據才能永久進入語言？

  真正成熟的極簡，不是以功能貧乏證明純粹，而是以最少、最清楚、最可組合的核心，承擔那些只有核心能承擔的責任；並讓其餘能力在具有邊界、證據、工具與治理的條件下成長。

  最終原則為：

> 【核心必須稀缺
> ∧
> 語言必須能成長
> ∧
> 成長不能摧毀核心的可理解性】

語言：
時期：
核心語法：
核心語義：
Runtime 核心：
標準程式庫：
擴展機制：
巨集：
宿主互操作：
工具承擔能力：
有效語言半徑：
主要功能交互：
功能准入制度：
刪除制度：
相容性化石：
主要保護對象：
主要擴張風險：
證據：
信心：
候選功能：
原始問題：
現有替代：
核心特權需求：
覆蓋率：
可組合性：
安全收益：

工具收益：
刪除的複雜度：
新增語義：
特徵交互：
實作成本：
教學成本：
遷移成本：
治理成本：
實驗證據：
撤回能力：
建議位置：

[R1] Scheme Language Steering Committee and editors, Revised^7 Report on the Algorithmic Language Scheme; Scheme Standards official website.

— 少量表達規則、自由組合、Small Language 與標準程序。

[R2] Niklaus Wirth, “Modula-2 and Oberon,” revised 2006; “A Plea for Lean Software”; Project Oberon.

— 設計簡潔、概念經濟、先刪後增、提高能力並降低複雜度。

[R3] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes, “The Evolution of Lua,” HOPL III, 2007; Lua official history and documentation.

— 小型、可攜、快速、易嵌入、Table 與可擴展語義。

[R4] Rob Pike, “Go at Google: Language Design in the Service of Software Engineering”; Go official talks and blog posts “Less is exponentially more,” “Why Generics?,” and “Experiment, Simplify, Ship.”

— 功能交互、正交性、組織工程與受控擴張。

[R5] Clojure official “Rationale,” “Features,” “Values and Change,” and “Clojure Governance and How It Got That Way.”

— 簡單而連貫的機制、不可變資料、宿主互操作與保守核心准入。

[R6] Bjarne Stroustrup, “Foundations of C++,” 2012; “The Design of C++0x”; “Abstraction and the C++ Machine Model.”

— 大型語言中的基本技術、一般機制、硬體映射、零額外成本與相容性附帶功能。

[R7] Scheme Standards, R7RS process documents.

— Working Group 1 Small Language 與語言標準治理。

[R8] Go Team, “Experiment, Simplify, Ship,” 2019.

— 增加發行物內部複雜度以降低整體程式設計經驗負擔，以及重新定義與移除功能的簡化方式。

[R9] Guy L. Steele Jr., “Growing a Language,” OOPSLA 1998; Higher-Order and Symbolic Computation 12(3), 1999.

— 現代語言應規劃成長，而非假設中央設計者一次預知全部需求。

[M-S] Syntactic minimalism
[M-M] Semantic minimalism
[M-I] Implementation minimalism
[M-X] Interaction minimalism
[M-O] Organizational minimalism
[M-E] Effective-language minimalism
[G-E] Eliminative growth
[G-G] Generative growth
[G-H] Host-based growth
[G-L] Library growth
[G-T] Tool-based growth
[G-C] Core feature growth

SECTION

E.1 Scheme 的「小核心」

已重新核對 R7RS 正文與 Scheme Standards 程序頁：

- R7RS 確實主張不應以功能疊加設計語言，而應移除使額外功能看似必要的弱點與限制；
- 報告確實以少量、可自由組合的表達式規則說明 Scheme 的設計；
- R7RS 標準工作確實設有 Working Group 1 — Small Language 與 Working Group 2 — Large Language。

  本文因此保留「生成式極簡」與 Small／Large 制度分層的描述，但不把所有 Scheme 實作視為同一規模，也不宣稱 Small 與 Large 已形成單一完整、同時發布的規格結果。

SECTION

E.2 Oberon 的先刪後增

已重新核對 Wirth〈Modula-2 and Oberon〉：

- Wirth 明確把設計簡潔稱為最重要的指導原則；
- Oberon 的策略確實是先判斷從 Modula-2 省略什麼，再決定必要新增；
- 其目標明確包含提高 Modula-2 的能力並同時降低複雜度。

  本文所稱「刪減式極簡」是對多項直接設計決策的分析標籤，不是 Wirth 本人使用的正式流派名稱。

SECTION

E.3 Lua 的小型與擴展

已重新核對 Lua HOPL 論文與官方歷史：

- Lua 自創始起即以 simple、small、portable、fast、easily embedded 為設計目標；
- Table 確實是唯一核心資料結構種類，並支撐多種資料與物件表示；
- Lua 早期三人設計小組以一致同意作為加入新功能的重要門檻，官方歷史也直接表達「之後加入功能通常比移除容易」。

  本文沒有把 Lua 的小型性誤寫成其所有嵌入環境或大型應用都同樣簡單。

SECTION

E.4 Go 的泛型與受控擴張

已重新核對 Go 官方〈Why Generics?〉、〈An Introduction to Generics〉與〈Experiment, Simplify, Ship〉：

- Go 官方在泛型設計階段明確承認泛型會增加語言複雜度；
- 官方提出減少新概念、保持單項功能簡單與正交，並把較多複雜度交給泛型程式庫作者而非使用者；
- Go 1.18 正式加入泛型，官方稱其為開源發布以來最大語言變更；
- Go 官方也明確說明，增加語言或發行物內部複雜度，可能在整體上簡化程式設計經驗。

  因此本文沒有把 Go 描述為永遠拒絕新功能，而是描述其對功能永久化具有高門檻。

SECTION

E.5 Clojure 的保守准入

已重新核對 Clojure 官方 Rationale 與 2012 年治理回顧：

- 官方 Rationale 確實以少量主要資料結構、不可變資料、宿主互操作等方式說明其設計；
- 2012 年治理回顧明確說明 Rich Hickey 對加入語言功能極為保守，並以 named arguments／destructuring 案例說明等待一般解法的理由。

該治理文章是特定歷史時期的官方回顧；本文只用來分析核心功能准入風格，不把其中所有組織描述直接當成 2026 年完整治理現況。

SECTION

E.6 C++ 的大型表面與基本原則

已重新核對 Stroustrup〈Foundations of C++〉：

- 文章明確稱 C++ 為大型且複雜的語言；
- 同時主張良好 C++ 主要依賴少數基本構造、技術與模型；
- 文章明確指出，老語言因相容性保留的 incidental features 可能干擾理解；
- 直接硬體映射與零額外成本抽象確實是其公開設計理想。

本文因此沒有將 C++ 歸為極簡語言，而是歸為「一般機制式擴張」：深層原則集中，但表面、歷史與交互成本很大。

SECTION

E.7 數學模型的地位

本文的功能交互圖、交互密度、效用與成本公式均為分析工具，不是已經得到經驗定律驗證的精確量表。後續 PLDST SKILL 若輸出數值，必須附上：

- 功能集合如何定義；

- 交互邊如何判定；
- 研究任務與使用者；
- 證據與信心；
- 未納入的生態與工具成本。