Backus、McCarthy 與 Hickey：函數、符號與簡單性的不同道路

Backus, McCarthy, and Hickey: Different Roads Through Functions, Symbols, and Simplicity

v1.0 / 2026-07-30 / Neo.K / 公開版／第四部跨設計師正式比較研究

https://thisoneisneok.com/html/papers/pldst-025.html

英文名稱： Backus, McCarthy, and Hickey: Different Roads Through Functions, Symbols, and Simplicity

系列： Programming Language Designer Style Taxonomy（PLDST）

文件編號： PLDST-025

版本： v1.0

日期： 2026-07-30

作者： Neo.K

文件狀態： 公開版／第四部跨設計師正式比較研究

SECTION

摘要

John Backus、John McCarthy 與 Rich Hickey 都曾把函數、組合、符號表示與簡單性置於程式語言設計的中心。然而，若只把三人共同歸入「函數式／Lisp／極簡設計」傳統，會同時誤解三者。

Backus 的後期工作不是一般意義上的 Lambda-calculus 語言推廣，而是試圖建立 Function-level programming：程式由完整函數及固定的 Combining forms 組成，不依賴命名參數、逐值變數、賦值與顯式遞迴。他希望程式語言同時成為程式代數，使程式可像方程式一樣在同一語言內被轉換、證明與求解。[R1][R2][R3]

McCarthy 的核心問題則不是先消除所有命名，而是讓機器能操作符號表達式、邏輯句子與可變大小結構。LISP 透過 S-expression、List、Recursive function、Conditional expression、Lambda、apply 與 eval，建立一個小型但具有極高生成力的符號核心。其簡單性是「少數表示與規則能生成整個活系統」，而不是「系統只有少量行為」。[R4][R5][R6]

Hickey 繼承 Lisp 的 Code-as-data、Macro、REPL 與函數組合，但將「簡單」重新定義為不交纏。Clojure 的重點不是宣稱真實世界沒有狀態，而是區分 Value、Identity、State、Reference 與 Time，讓不可變值、函數變換與受控參照各自承擔不同責任；再透過 Persistent data、宿主 JVM、Protocol、Transducer 與克制演化，把簡單性變成一項長期系統設計紀律。[R7][R8][R9][R10]

本文提出三種簡單性原型：

> 【Backus：Algebraic Transformational Simplicity;
> McCarthy：Generative Symbolic Simplicity;
> Hickey：Decomplective Structural Simplicity】

三者對「函數」的理解分別是：

> 【Backus：函數是組合與程式代數的基本單位;
> McCarthy：函數是遞迴符號計算與可執行語義的核心;
> Hickey：函數是不可變值之間的可組合轉換】

本文以十四個共同軸比較三人：

- 原始問題；
- 函數的地位；
- 符號與資料；
- 命名；
- 變數與賦值；
- 遞迴；
- 組合；
- 語言核心；
- 語言延展；
- 狀態與時間；
- 形式推理；

- 宿主與實作；
- 複雜度配置；
- 歷史限制與自我修正。

核心結論是：

> 三者都追求簡單，但 Backus 希望程式可以被代數化，McCarthy 希望計算可以被符號化，Hickey 希望系統可以被去交纏化。

因此「簡單語言」至少包含三種不同工程政治：

- 限制組合機制，使程式可推導；
- 統一表示，使系統可生成；
- 分離責任，使系統可理解。

關鍵詞： John Backus、John McCarthy、Rich Hickey、FP、LISP、Clojure、函數級程式設計、符號計算、簡單性、組合、不可變值、程式代數、PLDST

SECTION

一、不能以功能數量直接衡量簡單

一門語言的核心功能很少，不表示：

- 執行模型簡單；
- 程式容易理解；
- 生態不會分裂；
- 巨集與元程式設計沒有代價；
- 狀態與時間已被處理；
- 工具容易建立。

因此：

> CoreSizedownarrow
> not⇒
> SystemComplexitydownarrow

SECTION

二、簡單、容易、熟悉與小型必須分開

本文區分：

> Simple
> ≠
> Easy
> ≠
> Familiar
> ≠
> Small
> ≠
> Short

其中：

- Simple：概念是否少交纏、可分解或可由少數規則推導；
- Easy：是否接近既有能力與工具；
- Familiar：是否符合既有經驗；
- Small：核心功能或語法數量是否少；
- Short：表面字元或程式行數是否少。

SECTION

三、簡單性向量

定義：

> cal-S(L)=
> (
> S_(core),
> S_(composition),
> S_(representation),
> S_(state),
> S_(reasoning),
> S_(tool),
> S_(evolution)
>)

三位設計者優化的分量不同。

SECTION

四、簡單性不是總量消失

更合理的模型是：

> C_(total)
> =
> C_(surface)
> +
> C_(runtime)
> +
> C_(compiler)
> +
> C_(library)
> +
> C_(state)
> +
> C_(tool)
> +
> C_(governance)

設計者通常只能重分配，而不能消滅所有複雜度。

SECTION

五、三種簡單性問題

Backus 問：

> 能否讓程式由少量可推導的函數組合構成，而不再逐字操縱狀態？

McCarthy 問：

> 能否用少數符號表示與遞迴規則，建立可自我解釋並持續生長的計算系統？

Hickey 問：

> 能否把值、身分、狀態、時間、資料處理與執行情境拆開，使每項機制只承擔清楚責任？

SECTION

六、本文所稱「道路」

「道路」不是線性演進：

Backus → McCarthy → Hickey
事實上 McCarthy 的 LISP 早於 Backus 1977 年的 FP 論述，Hickey 則在數十年後重組 Lisp、函數式與產業平台。

本文比較的是三種概念路線，而不是單一路線的前後版本。

SECTION

七、都拒絕讓機器偶發細節主導語言

Backus 批判 Word-at-a-time、Storage cell、Assignment 與 Jump 對語言的支配。[R1]

McCarthy 建立與 IBM 704 相對獨立的 S-expression 與 Recursive function 表示，使符號問題不必直接映射固定記憶體陣列。[R4]

Hickey 讓值保持不可變，將變化放入明確 Reference semantics，而不是把記憶體位置內容直接等同於事物狀態。[R8]

SECTION

八、都把函數提高為一級構造

三者都拒絕把函數只視為：

- 一段可跳轉 Subroutine；
- 一個帶副作用的命令容器；
- 對機器操作的薄包裝。

  但三者提高函數的方式不同。

SECTION

九、都追求組合

> NewProgram
> =
> Combine(ExistingPrograms)

然而：

- Backus 要固定 Combining forms 及其代數律；
- McCarthy 依靠 Lambda、Recursive definition、List 與 eval；
- Hickey 依靠 First-class function、Persistent values、Sequence abstraction、Transducer 與 Protocol。

SECTION

十、都把執行系統視為語言的一部分

三者的成果都不能只靠語法：

- FP 需要實作模型、程式代數與狀態轉換系統；
- LISP 需要 Interpreter、GC、Reader、Printer、Compiler 與 Online environment；
- Clojure 需要 JVM、Persistent collection、Compiler、Reference type、REPL 與 Library。

SECTION

十一、都不是純粹功能刪除

三者都不是把語言縮小到不能工作。

Backus 希望少量 Combining forms 產生高階程式。

McCarthy 希望小型核心能生成新的語言能力。

Hickey 希望功能不交纏，並以宿主平台與資料導向擴張。

SECTION

十二、都面對現實邊界

- Backus 的 FP 難以乾淨處理 I/O 與周邊系統；
- LISP 的小核心生成方言、動態錯誤與治理問題；
- Clojure 的簡單語義依賴複雜 JVM、Persistent structure、Compiler 與工具。

所以三者都證明：

> ConceptualSimplicity
> ≠
> CompleteSystemSimplicity

SECTION

十三、問題不是函數不足，而是程式仍被狀態語法統治

Backus 將傳統語言看成馮紐曼機器的高階複製：

- Variable 對應 Storage cell；
- Assignment 對應 Fetch／Store；
- Control statement 對應 Jump／Test；
- 程式必須逐字搬運資料。[R1]

  其批判不是「效能太低」，而是：

> 機器架構同時成為人類思想的瓶頸。

SECTION

十四、Word-at-a-time 思考

馮紐曼瓶頸不只是 CPU 與 Memory 之間的資料通道，也是一種概念瓶頸：

> Problem
> →
> SequenceOfWordOperations

人必須把大尺度轉換拆成大量地址、載入、儲存及更新。

SECTION

十五、Function-level 與 Value-level

Backus 的 FP 特別區分：

- Value-level：以變數表示值，建立 Lambda、替換與遞迴；
- Function-level：直接把完整函數當作組合對象。

  在 FP 中：

  程式就是函數
  新程式由既有函數與固定 Combining forms 建立

不必反覆命名每個輸入值

SECTION

十六、函數是程式單位

Backus 的函數不是只接受值並回傳值。

它同時是：

- 程式；
- 代數中的變量；
- 可組合構造；
- 可被等式替換的對象；
- 可由定理描述的行為單位。

SECTION

十七、Combining forms

Combining forms 可由既有函數建立新函數，例如：

- Composition；
- Construction；
- Condition；
- Insert；
- Apply-to-all；
- Constant。

其目標為：

> f⊠,f⊠,…,f⊠
> xrightarrowϕ
> g

其中 φ 本身具有可理解的代數律。

SECTION

十八、固定組合機制的政治

Backus 並不主張每個使用者任意發明無數組合形式。

他警告：

> UnrestrictedCombiningFreedom
> ⇒
> Chaos

因此 FP 偏好：

- 少量固定 Combining forms；
- 長期熟悉其性質；
- 以代數律支持推理；
- 透過組合而非核心功能堆疊擴張。

SECTION

十九、程式代數

Backus 希望：

> ProgramEquation
> ↔
> AlgebraicEquation

程式語言同時是：

- 撰寫程式的語言；
- 轉換程式的語言；

- 證明程式的語言；
- 解函數方程的語言。

  而不是另用一套外部邏輯只「談論」程式。[R1]

SECTION

二十、同一語言內推理

理想狀態為：

> P
> =
> Q

既表示兩個程式計算同一函數，也允許用代數律把 P 逐步轉換成 Q。

因此：

> Programming
> +
> Transformation
> +
> Proof

不再被完全分開。

SECTION

二十一、無命名並非反符號

Backus 反對的是高密度命名與替換機制對程式結構的支配，而不是所有符號。

FP 仍需要：

- 函數名稱；

- Combining form；
- 定義；
- Algebraic variable。

但它試圖避免：

> 為每一次值流動建立局部名稱，並讓名稱管理成為主要程式工作。

SECTION

二十二、無顯式遞迴

許多 FP 程式可透過高階 Combining forms 隱含遞迴行為，而不直接寫自我呼叫。

這將責任移至：

- α 類 Apply-to-all；
- insert 類 Fold；
- 組合形式；
- 系統執行模型。

SECTION

二十三、資料作為整體

Backus 希望程式處理：

- Vector；
- Tree；
- Structured data；
- 完整轉換；

而不是逐字操作。

因此：

> UnitOfThought
> uparrow

SECTION

二十四、狀態沒有完全消失

Backus 並非否認歷史敏感系統需要狀態。

他提出 Applicative state transition 系統，希望：

- 主要計算保持 Applicative；
- 狀態轉換只在大尺度發生；
- 語義不與每一個微小操作緊密耦合。

所以：

> StateFrequencydownarrow
>
> StateSemanticCouplingdownarrow

而不是：

> State=0

SECTION

二十五、Backus 的簡單性

Backus 的簡單性為：

> S_B
> =
> FixedCombiningForms
> +
> AlgebraicLaws
> +
> NoValueVariables
> +
> LargeConceptualUnits
> -
> WordAtATimeControl

SECTION

二十六、實作與現實限制

Backus 後來承認：

- Combining forms 較容易形成；
- 完整 I/O、互動與周邊系統難以乾淨納入；
- FP 作為完整系統最終未成功；
- 仍需要新的方式處理歷史與外部世界。[R2]

這使 Backus 的簡單性具有強烈反身性：

> 他不只批判舊語言，也批判自己方案未完成之處。

SECTION

二十七、Backus 原型

本文將 Backus 判定為：

> 【Algebraic Transformational Simplifier】

其核心命題：

> 只有當程式可以由少量函數組合建立，並能在同一語言內推導與轉換，抽象才真正脫離逐字操作。

SECTION

二十八、問題先於簡潔語法

LISP 的原始需求是讓 Advice Taker 操作：

- 宣告句；
- 命令句；
- 邏輯關係；
- 形式知識；
- 推論過程。[R4]

因此 McCarthy 不是先問：

> 怎樣設計最少語法？

而是問：

> 什麼表示能讓機器處理任意大小與深度的符號結構？

SECTION

二十九、S-expression

S-expression 以兩項基本規則建立：

- Atom 是 S-expression；
- 若 e⊠,e⊠ 是 S-expression，則 Pair 也是 S-expression。

這產生：

- List；
- Tree；
- Nested expression；
- Program representation；
- Symbolic data。

SECTION

三十、符號是可操作對象

傳統數值語言主要計算：

> Number→ Number

LISP 擴張為：

> SymbolicExpression
> →
> SymbolicExpression

程式可：

- 分解公式；
- 重新排列；
- 代換；
- 推理；
- 生成新程式；
- 解釋程式。

SECTION

三十一、函數是遞迴規則

McCarthy 的函數觀包含：

- Partial function；
- Predicate；
- Conditional expression；
- Lambda abstraction；
- Recursive definition；
- Universal function。

  函數描述符號結構如何被分解與重組。

SECTION

三十二、條件式成為表達式

cond 使分支進入函數定義：

> Branch
> ∈
> Expression

計算不必完全分成：

Expression world
Statement world
這一點與 Backus 對傳統語言分裂的批判形成呼應。

SECTION

三十三、遞迴對齊資料結構

若資料是遞迴樹：

> Data
> =
> Atom
> ∪
> Pair(Data,Data)

則程式也可遞迴：

> Program(Data)
> =
> Case(Atom)
> ∪
> Combine(Program(left),Program(right))

問題、資料與控制結構因此互相對齊。

SECTION

三十四、Lambda 的角色

McCarthy 使用 Lambda 表示：

- 參數；
- 函數；
- 綁定；
- 高階傳遞。

  這與 Backus 後期試圖減少 Lambda 式變量替換形成直接張力。

SECTION

三十五、apply 與 eval

1960 年論文中的 Universal S-function：

- 理論上類似 Universal Turing machine；
- 實際上可作 Interpreter；
- 使語言形式與執行系統接近。

> FormalDefinition
> →
> ExecutableSemantics

SECTION

三十六、程式即資料

當程式與資料都使用 S-expression：

> Representation(program)
> =
> Representation(data)

因此程式可以：

- 讀取程式；
- 建構程式；
- 改寫程式；
- 解釋程式；
- 建立 DSL；
- 建立 Macro。

SECTION

三十七、極小核心是生成器

McCarthy 的簡單性不是：

所有 Lisp 程式都很短

而是：

少量原語可以生成大量語言能力

可表示為：

> GenerativePower
> =
> (ReachableLanguageSpace)/(CoreMechanisms)

SECTION

三十八、Reader 與 Printer

統一表示需要：

- Reader 將外部文字轉成 S-expression；
- Printer 將結構重新輸出；
- Symbol table 維持 Atom；
- List structure 保存資料；
- GC 自動回收。

  因此括號背後是一整個運行制度。

SECTION

三十九、GC 的人本與理論價值

若使用者必須手工回收 List node：

> SymbolicAbstraction
> →
> MemoryBookkeeping

GC 把儲存生命週期交給 Runtime，使符號操作保持高階。

SECTION

四十、互動環境

LISP 很快包含：

- 定義函數；
- 直接求值；
- Trace；
- Error diagnostic；
- Interpreter；
- Compiler；
- Reader／Printer。

小核心因此變成 Live system。

SECTION

四十一、可延展性

McCarthy 型簡單性允許使用者建立：

- 新控制形式；
- 新資料表示；
- Macro；
- Interpreter；
- Symbolic tool；
- Domain language。

簡單性因此帶來權力，而不是只帶來限制。

SECTION

四十二、生成性代價

> SmallCore
> +
> Metaprogramming
> +
> DynamicRepresentation
> ⇒
> LargeEmergentLanguage

代價包括：

- 方言；
- 巨集慣例；
- 動態錯誤；
- 工具難度；
- 安全邊界；
- 社群分裂；
- 語言治理。

SECTION

四十三、偶發穩定

McCarthy 原本區分 M-expression 與 S-expression，後者偏向內部表示。

但因 M-expression 未完成普及：

> IntermediateRepresentation
> →
> SurfaceSyntax

Lisp 最具代表性的表面，部分來自歷史偶發，而非完整預定終局。

SECTION

四十四、符號統一的邊界

統一表示不表示：

- 所有資料都最適合 List；
- 所有領域都應以 Macro 建模；
- 所有語義都應由 eval 開放；
- 所有系統狀態都天然簡單。

Lisp 後代必須持續補充：

- Vector；
- Record；
- Module；
- Type；
- Package；
- Safety；
- Namespace；
- Concurrency。

SECTION

四十五、McCarthy 的簡單性

> S_M
> =
> UniformRepresentation
> +
> RecursiveCore
> +
> ExecutableSemantics
> +
> ProgramDataUnity
> -
> PrimitiveDiversity

SECTION

四十六、McCarthy 原型

本文將 McCarthy 判定為：

> 【Generative Symbolic Simplifier】

其核心命題：

> 一個核心若能統一資料、程式與語義，便不必預先內建所有未來能力；使用者可以讓語言從內部繼續生長。

SECTION

四十七、繼承 Lisp，但不複製 Lisp

Clojure 繼承：

- S-expression；
- Code-as-data；
- Macro；

- REPL；
- First-class function；
- Dynamic development。

  但 Hickey 重新選擇：

- Vector、Map、Set 與 List 都是一級資料；
- Immutable persistent collection；
- JVM interoperability；
- 多種 Reference semantics；
- Namespace；
- Protocol；
- 宿主編譯。

SECTION

四十八、Simple 不等於 Easy

〈Simple Made Easy〉區分：

> Simple
> =
> NotIntertwined

> Easy
> =
> NearToExistingAbility

熟悉的物件可變模型可能容易，卻不簡單。

不熟悉的 Persistent value 可能不容易，卻更容易被獨立理解。[R7]

SECTION

四十九、Complect

Complect 指：

> 把原本可獨立變化、推理或替換的概念編織成同一件事。

例如：

- Identity 與 State；
- Data 與 Behavior；
- Order 與 Result；
- Location 與 Value；
- Algorithm 與 Source／Sink；
- Policy 與 Mechanism。

SECTION

五十、Value

Value 是：

- 不可變；
- 可比較；
- 可分享；
- 可傳遞；
- 表示某項資訊；
- 不因位置改寫而改變。

> Value□
> ≠
> Value_(t+1)

新值不是舊值被修改，而是另一個值。

SECTION

五十一、Identity

Identity 是跨時間指稱同一事物的名稱：

- 帳戶；
- 訂單；
- 玩家；
- 程序；
- 資料庫。

  Identity 不等於任何單一 State。

SECTION

五十二、State

> State(identity,t)=Value□

某 Identity 在時間 t 關聯一個不可變值。

變化表示：

> Value□
> →
> Value□
> →
> Value□

由 Identity 串接，而不是舊值本體被改造成新值。

SECTION

五十三、Reference

Reference 管理：

- 目前 Value；
- 更新規則；
- 同步；
- 協調；
- 可觀察性；
- 時間順序。

  Clojure 使用 Ref、Atom、Agent、Var 等不同構造，不讓單一「可變變數」概念承擔所有狀態語義。

SECTION

五十四、函數是值轉換

Hickey 型函數主要是：

> f:Value→ Value

函數可在沒有 Identity 更新的情況下先計算新值。

狀態變化再由明確參照機制提交。

SECTION

五十五、計算與提交分離

> newValue=f(oldValue)

與：

> Reference← newValue

是兩項不同責任。

這讓：

- 測試；
- 重試；
- 並行；
- Snapshot；
- 推理；
- Audit；

更容易。

SECTION

五十六、Persistent data

不可變若每次完整複製，成本過高。

Persistent data structure 以結構共享使：

> UpdateCost
> ≪
> FullCopyCost

Runtime 承擔結構複雜度，使用者獲得穩定值語義。

SECTION

五十七、資料高於物件封裝

Clojure 傾向使用開放資料：

- Map；
- Vector；
- Set；
- Keyword；
- Sequence；
- Metadata。

  行為多由 Namespace 中函數處理，而不是每種資料都封裝於可變物件。

SECTION

五十八、函數與資料保持可分離

傳統物件可能將：

> Data
> +
> Identity
> +
> Methods
> +
> Mutation
> +
> Lifecycle

綁成同一單位。

Hickey 偏好：

> Data
> +
> Functions
> +
> ExplicitReferences

SECTION

五十九、Sequence abstraction

Clojure 的 Sequence 讓多種 Collection 可由共同函數操作。

這保留 McCarthy 式統一操作精神，但不要求所有資料都實際是 List。

SECTION

六十、Transducer

Transducer 將資料轉換 Algorithm 從：

- Input source；
- Output sink；
- Collection type；
- Execution strategy；

  中拆開。

  可抽象為：

> Transformation
> ≠
> CollectionTraversalContext

這是 Hickey 去交纏方法的典型後續。

SECTION

六十一、宿主平台槓桿

Clojure 不建立自己的完整 OS、VM 與 Library 世界。

它使用：

- JVM；
- Java Library；
- Java Thread；
- Java Class；
- Existing deployment。

  因此：

> NewSemanticModel
> +
> ExistingIndustrialRuntime

SECTION

六十二、簡單不等於獨立實作

依賴 JVM 會降低 Clojure 自建能力數量，卻引入：

- Java type；
- Reflection；
- Classloader；
- Host exception；

- Startup；
- Interop edge case；
- Performance model。

  所以：

> CoreSimplicity
> +
> HostComplexity

同時存在。

SECTION

六十三、克制核心演化

Hickey 與 Clojure Core 長期傾向：

- 少加功能；
- 尋找一般解法；
- 依靠 Library；
- 使用 Data／Function；
- 保持向後相容；
- 等待需求成熟。

  這使簡單性成為治理准入條件。

SECTION

六十四、Hickey 的簡單性

> S_H
> =
> SeparationOfConcerns

SECTION

六十五、Hickey 原型

本文將 Hickey 判定為：

> 【Decomplective Structural Simplifier】

其核心命題：

> 系統難以理解，往往不是因為功能太多，而是因為本可分離的概念被綁成同一個機制；簡單性首先是解除交纏。

SECTION

六十六、Backus：程式本身

Backus 的函數主要是：

- 完整程式；
- 組合對象；
- 代數變量；
- 大尺度轉換。

他希望避免把注意力放在每個參數名與值替換上。

SECTION

六十七、McCarthy：遞迴符號規則

McCarthy 的函數主要是：

- 對 Symbolic expression 的操作；
- Conditional recursive definition；

- Lambda abstraction；
- apply／eval 的可執行語義。

SECTION

六十八、Hickey：值之間的轉換

Hickey 的函數主要是：

- First-class value；
- 對不可變資料的轉換；
- 與 Identity 更新分離；
- 可從 Collection context 解耦；
- 可由 REPL 互動組合。

SECTION

六十九、三種函數尺度

> 【Backus：ProgramScaleFunction;
> McCarthy：SymbolicRecursiveFunction;
> Hickey：ValueTransformationFunction】

SECTION

七十、函數並非同義詞

把三者都稱為 Functional programming 會掩蓋：

- Backus 對 Lambda naming／substitution 的批判；
- McCarthy 對 Lambda、Recursion 與 Universal function 的依賴；

- Hickey 對 Impure host、Explicit references 與 Pragmatism 的接受。

SECTION

七十一、Backus：降低值名稱支配

Backus 希望程式主要呈現：

- 函數結構；
- 資料轉換；
- 組合關係；
- 代數等式。

符號服務於函數級關係，而不是逐步地址操作。

SECTION

七十二、McCarthy：符號是計算材料

McCarthy 把 Atom、List 與 S-expression 建立為：

- 資料；
- 公式；
- 程式；
- 語言描述；
- 推理內容。

符號在其系統中具有本體中心地位。

SECTION

七十三、Hickey：從符號中心轉向值中心

Clojure 保留 Symbol 與 S-expression，但更強調：

- Map；
- Vector；
- Set；
- Keyword；
- Value；
- EDN；
- 開放資料。

  程式即資料仍重要，但「資訊是值」比「一切都是 List」更中心。

SECTION

七十四、表示統一程度

> RepresentationUnity:
> McCarthy > Hickey > Backus

但這不是優劣排名。

Backus 更關心程式組合，McCarthy 更關心統一符號表示，Hickey 更關心不同資料角色保持清楚。

SECTION

七十五、Backus：反命名密度

Backus 認為大量變數與 Naming convention：

- 阻礙組合；
- 增加 Procedure declaration；
- 讓程式侷限於特定參數；

- 強化 Word-at-a-time 思考。

SECTION

七十六、McCarthy：命名是遞迴與符號操作資源

LISP 使用：

- Symbol；
- Parameter；
- Function name；
- Lambda variable；
- Label；
- Association list。

命名不是應消除的負擔，而是符號計算核心。

SECTION

七十七、Hickey：區分名稱與值

Clojure 的 Symbol／Var／Value 分開：

- Symbol 是程式表示；
- Var 是 Namespace 中名稱到值的可重定義參照；
- Value 本身不可變；
- Local binding 不代表可變儲存格。

這是對 Lisp 命名模型的再結構化。

SECTION

七十八、命名成本公式

> C_(name)
> =
> Creation
> +
> Scope
> +
> Resolution
> +
> Substitution
> +
> MutationAmbiguity

Backus 主要降低 Creation 與 Substitution。

McCarthy 利用 Name 生成語言能力。

Hickey 主要降低 MutationAmbiguity，並以 Namespace 管理 Resolution。

SECTION

七十九、Backus：固定 Combining forms

> g=ϕ(f□,…,f□)

組合形式數量受限，並具有代數律。

SECTION

八十、McCarthy：Lambda、List 與 Macro 生長

組合來自：

- Function application；
- Recursive definition；
- List construction；

- Higher-order operation；
- eval；
- Macro tradition。

  其可生成空間更開放。

SECTION

八十一、Hickey：資料、函數、Protocol 與 Transducer

組合來自：

- Pure／mostly pure function；
- Persistent values；
- Sequence；
- Higher-order function；
- Protocol；
- Multimethod；
- Transducer；
- Macro。

  但核心功能加入受到較強治理克制。

SECTION

八十二、組合自由與推理性

> CompositionFreedomuparrow
> ⇒
> PotentialVariationuparrow

Backus 以固定形式換取推理。

McCarthy 以開放生成換取延展。

Hickey 以結構分離與保守核心換取長期可理解性。

SECTION

八十三、Backus：把遞迴封裝進組合形式

使用者不必反覆寫：

function calls itself
而以高階結構表示：

- Map；
- Fold；
- Insert；
- Tree／Sequence transform。

SECTION

八十四、McCarthy：遞迴是語言本體

Recursive function 是 LISP 形式核心之一。

資料與程式都由遞迴規則構成。

SECTION

八十五、Hickey：遞迴是可用機制，但不必支配日常

Clojure 支援：

- recur；
- Loop；

- Sequence operation；
- Reduce；
- Lazy sequence；
- Transducer。

日常程式通常由 Library 高階函數承擔遞迴。

SECTION

八十六、遞迴的三種責任分配

Backus：遞迴下沉到 Combining forms
McCarthy：遞迴顯性成為形式定義
Hickey：遞迴多下沉到資料操作與 Library

SECTION

八十七、Backus：降低狀態耦合

Backus 不滿意每個微小計算都修改全域機器狀態。

其目標：

> Computation
> ≫
> StateTransitionFrequency

SECTION

八十八、McCarthy：形式核心 Applicative，系統很快加入命令能力

1960 年 LISP 核心以函數與 S-expression 描述，但實際系統已包含：

- Assignment；
- go to；

- Machine-language function；
- Association list；
- I/O；
- Compiler。

  因此：

> PureFormalCore
> ≠
> EntireOperationalSystem

SECTION

八十九、Hickey：狀態重新建模

Hickey 不試圖把狀態藏起來，而是要求：

> Identity
> ≠
> State
> ≠
> Value
> ≠
> Reference
> ≠
> Time

SECTION

九十、Backus 與 Hickey 的重要差異

Backus 的主要策略：

> 讓狀態轉換更少、更大尺度、更鬆耦合。

Hickey 的主要策略：

> 讓狀態轉換具有明確 Identity、Reference、時間與協調語義。

SECTION

九十一、McCarthy 的中間位置

McCarthy 提供函數式形式核心，但其真正革命是符號與語言自我表示，而不是完整狀態理論。

後代 Lisp 因此可採：

- Functional；
- Imperative；
- Object；
- Actor；
- STM；
- Persistent value。

SECTION

九十二、Backus：代數轉換

推理理想：

> P
> =
> P□
> =
> P□
> =
> …
> =

每一步由程式代數律支持。

SECTION

九十三、McCarthy：遞迴形式與 Universal function

推理理想：

- 形式定義；
- Conditional equation；
- Recursive function；
- apply／eval；
- 程式與資料表示。

  它建立語言的數學與執行連接。

SECTION

九十四、Hickey：局部可推理性

Hickey 追求：

- 值不變；
- 函數輸入輸出穩定；
- 狀態邊界明確；
- 資料開放；
- Algorithm 與執行情境分離。

  不是建立完整程式代數，而是降低推理所需同時追蹤的維度。

SECTION

九十五、三種推理對象

> 【Backus：程式等式;
> McCarthy：符號函數與語言語義;
> Hickey：值、時間與責任邊界】

SECTION

九十六、Backus：核心固定，程式由組合擴張

> StableCombinators
> →
> NewPrograms

他反對不斷把特殊功能加入 Framework。

SECTION

九十七、McCarthy：核心小，語言可由自身擴張

> SmallCore
> +
> ProgramAsData
> +
> Eval
> →
> NewLanguageForms

SECTION

九十八、Hickey：核心克制，擴張依靠資料、函數、宿主與 Library

> ConservativeCore
> +
> JVM

SECTION

九十九、延展性的不同治理

- Backus：以固定 Combining forms 約束；
- McCarthy：以語言內生成能力開放；
- Hickey：以創始品味、Core team 與宿主槓桿克制。

SECTION

一百、Backus：尋找非馮紐曼整體系統

Backus 希望：

- 語言；
- 程式代數；
- 狀態轉換；
- 甚至機器架構；

共同脫離馮紐曼模型。

這是一種高風險整體替代。

SECTION

一百零一、McCarthy：形式獨立，實作進入 IBM 704

S-expression 與 S-function 被描述為與特定機器獨立。

但實際成功依賴：

- IBM 704；
- List storage；

- GC；
- Stack；
- Reader；
- Compiler；
- Interpreter。

SECTION

一百零二、Hickey：語義重建，宿主不重建

Clojure 主動接受 JVM：

- 不重新建立全部 Runtime；
- 不重新建立企業 Library；
- 直接進入既有部署；
- 在宿主上重建 Value／State／Function model。

SECTION

一百零三、替代幅度

> SystemReplacement:
> Backus > McCarthy > Hickey

其中：

- Backus 想替代整體範式；
- McCarthy 建立新符號系統但仍落地既有機器；
- Hickey 明確選擇宿主共生。

SECTION

一百零四、Backus 的配置

降低：

- 值變數；
- 賦值；
- 逐字控制；
- 顯式遞迴；
- 外部證明語言依賴。

增加：

- Combining system；
- Function algebra；
- Specialized implementation；
- I/O integration difficulty；
- 新計算模型成本。

SECTION

一百零五、McCarthy 的配置

降低：

- 固定資料格式；
- 語法原語數量；
- 程式／資料表示差異；
- 手工記憶體回收；

- 新語言機制准入門檻。

  增加：

- Runtime；
- GC；
- Dynamic reasoning；
- Macro／eval 安全；
- 方言；
- 工具；
- 社群治理。

SECTION

一百零六、Hickey 的配置

降低：

- Mutable object reasoning；
- Defensive copying；
- State ambiguity；
- Algorithm／context coupling；
- 核心功能膨脹。

  增加：

- Persistent structure；
- JVM interop；
- Reference model learning；

- Dynamic tooling；
- Macro discipline；
- Core governance dependence。

SECTION

一百零七、三種複雜度搬運

> 【Backus：操作複雜度→組合代數與系統實作;
> McCarthy：表示複雜度→Runtime 與語言內生成;
> Hickey：狀態複雜度→值結構與明確參照】

SECTION

一百零八、Backus 對 Lambda 傳統的批評

Backus 認為 Lambda-calculus 型系統雖強大，卻仍帶有：

- 命名；
- Substitute；
- Variable；
- 自由發明 Combining form；

  等複雜性。

  這直接區別於 McCarthy。

SECTION

一百零九、McCarthy 的成功反駁了部分疑慮，也證實部分疑慮

Lisp 的歷史顯示：

- Lambda、Symbol 與 eval 可形成長壽語族；
- 統一表示具有極高延展性；
- 小核心能支撐大量應用。

  同時也顯示：

- 方言分裂；
- 巨集文化差異；
- 動態工具成本；
- 大型語言表面；
- 相容與治理；

  確實會累積。

SECTION

一百一十、Hickey 對 Lisp 的選擇性繼承

Hickey 保留：

- Code-as-data；
- Macro；
- REPL；
- Function；
- S-expression。

  但拒絕或重構：

- 普遍可變資料；
- Identity／State 混合；

- List 唯一中心；
- 自建封閉平台；
- 無限制功能增長；
- 物件式資料—行為—狀態綁定。

SECTION

一百一十一、Hickey 不是 Backus 的直接完成者

Clojure 並沒有實作 Backus FP 的固定 Function-level algebra。

它仍使用：

- Local binding；
- Lambda／fn；
- Symbol；
- Variable name；
- Recursion；
- State references；
- Host side effects。

所以 Hickey 與 Backus 的共同點是反交纏與重視值轉換，不是語言機制相同。

SECTION

一百一十二、McCarthy 不是只有歷史中介

McCarthy 提出的核心統一具有獨立方向：

> Program
> =
> Data
> =

這一方向既不同於 Backus 的 Program algebra，也不同於 Hickey 的 Value／Identity separation。

SECTION

一百一十三、Backus 不等於現代函數式語言總代表

Backus FP：

- 不以 Lambda calculus 為中心；
- 偏 Function-level；
- 使用固定 Combining forms；
- 具有自己的代數與狀態轉換構想。

  不能把 Haskell、ML、Scheme、Clojure 全部直接回寫為其方案。

SECTION

一百一十四、McCarthy 不等於 Lisp 所有成果

LISP 是多人系統：

- Steve Russell；
- Daniel Edwards；
- Timothy Hart；
- Michael Levin；
- Paul Abrahams；
- Phyllis Fox；
- Klim Maling；
- 多個後續實作與共同體。

  eval 成為 Interpreter 的實作洞見尤其不能只歸於 McCarthy。

SECTION

一百一十五、Hickey 的簡單不表示 Clojure 容易

Clojure 仍要求理解：

- Lisp syntax；
- Persistent collection；
- JVM；
- Lazy sequence；
- Macro；
- Reference type；
- Namespace；
- Host interop；
- Toolchain。

  簡單性是結構目標，不是入門成本保證。

SECTION

一百一十六、小核心不等於小生態

> SmallLanguageKernel
> +
> Time
> +
> Users
> +
> Libraries
> =
> LargeOperationalSystem

Lisp 與 Clojure 都證明核心與實際工作環境必須分開評估。

SECTION

一百一十七、不可變不等於沒有時間

Hickey 不是消除時間，而是讓時間顯性：

> Identity
> xrightarrowt
> Value□

Backus 也承認歷史敏感系統需要某種狀態轉換。

SECTION

一百一十八、形式優雅不能自動處理外部世界

- I/O；
- Failure；
- Resource；
- Concurrency；
- Distribution；
- Deployment；
- Interop；
- Security；

  都需要額外模型。

  三人的簡單性方案都必須在此接受壓力測試。

SECTION

一百一十九、比較矩陣

軸　Backus　McCarthy　Hickey

原始問題　擺脫逐字狀態控制　操作符號與形式知識　拆開值、身分、狀態與時間

函數地位　完整程式與代數單位　遞迴符號函數與語義　不可變值之間的轉換

符號地位　降低值命名支配　計算的基本材料　保留 Lisp Symbol，但轉向值與資料

命名態度　減少命名與替換　命名、Lambda、Symbol 為核心　區分 Symbol、Var、Binding、Value

變數／賦值　強烈批判　形式核心弱化，實際系統納入　Local binding 與明確 Reference 分離

遞迴　多封裝進 Combining forms　核心形式　多由 Library／Sequence／Reduce 承擔

組合　固定 Combining forms　Lambda、List、eval、Macro　Function、Data、Protocol、Transducer

核心　小且固定　小且生成性高　小而克制，宿主槓桿大

延展　組合既有函數　語言內生成新形式　Library、Macro、Protocol、JVM

狀態　降低頻率與語義耦合　核心 Applicative，系統混合　Value／Identity／Reference 明確分離

形式推理　程式代數　遞迴函數與可執行語義　局部可推理與責任分離

宿主策略　傾向整體替代　形式獨立、落地 IBM 704　JVM 共生

主要優勢　可轉換、可推導、大尺度思考　統一表示、自我延展、活系統　並發可推理、資訊穩定、實用生態

主要風險　I/O／完整系統不足　方言、動態、安全、工具　學習、宿主複雜度、集中治理

SECTION

一百二十、三種目標函數

Backus：

> J_B
> =
> α A_(algebra)
> +

McCarthy：

> J_M
> =
> α U_(representation)
> +
> β G_(generativity)
> +
> γ E_(executable-semantics)
> -
> λ P_(primitive-diversity)

Hickey：

> J_H
> =
> α D_(decomplecting)
> +
> β V_(immutability)
> +
> γ R_(explicit-state)
> +
> δ H_(host-leverage)
> -
> λ I_(intertwining)

這些是本文分析模型，不是三人提出的量化公式。

SECTION

一百二十一、先回答你要的是哪種簡單

新語言應明確選擇：

代數可推導？
核心少且生成力高？
責任不交纏？

入門熟悉？
表面短？
工具容易？
生態一致？
不能用「簡單」一詞同時代替全部目標。

SECTION

一百二十二、建立簡單性負擔表

每個設計決策應記錄：

刪除了什麼概念？
複雜度被搬到哪一層？
使用者還要理解什麼？
Runtime 新增什麼責任？
Tooling 能否恢復隱藏資訊？
是否增加語言內生成權力？
是否增加狀態或時間歧義？
是否形成新的治理瓶頸？

SECTION

一百二十三、建立函數尺度標記

語言文件應說明函數主要被用作：

- Value-level calculation；
- Program-level composition；
- Effectful procedure；
- Symbolic rule；
- Transformation pipeline；
- Stateful transition。

  否則「支援函數式」資訊不足。

SECTION

一百二十四、建立符號權力邊界

若程式與資料統一，必須設計：

- Read／eval 分離；
- Macro phase；
- Namespace；
- Security；
- Tooling；
- Source mapping；
- Generated code diagnostics。

McCarthy 的生成性必須配合現代邊界。

SECTION

一百二十五、建立狀態頻率與狀態語義雙指標

Backus 提醒我們降低：

> Frequency(StateTransition)

Hickey 提醒我們明確：

> Semantics(StateTransition)

兩者缺一不可。

SECTION

一百二十六、固定組合與開放延展需分層

可以設計：

核心層：少量固定、可推導組合
語言層：受限 Macro／Protocol
程式庫層：開放組合
應用層：領域 DSL
這是三人路線可互補之處。

SECTION

一百二十七、宿主不是免費午餐

使用既有 Runtime 可節省：

- GC；
- Thread；
- Library；
- Deployment；
- Tooling。

但必須公開宿主洩漏：

- Type；
- Exception；
- Performance；
- Reflection；
- Lifecycle；
- Interop。

SECTION

一百二十八、不要把簡單性道德化

複雜需求有時真的需要：

- 多種狀態模型；
- Effect；
- Resource；
- Transaction；
- Distribution；
- Compatibility；
- Optimization。

簡單性不是把所有複雜需求稱為錯誤，而是讓複雜度具有可辨識位置。

SECTION

一百二十九、三人共同反對的是什麼

三者共同反對：

> 讓使用者在沒有必要時，替機器、表示或歷史偶發支付重複認知成本。

SECTION

一百三十、三人的分歧

Backus 認為根本問題是：

> Programming
> ≈
> WordAtATimeStateManipulation

McCarthy 認為根本機會是：

> Programs
> can manipulate
> Programs
> as symbols

Hickey 認為根本問題是：

> Concepts
> are complected

SECTION

一百三十一、三種簡單性憲法

Backus：

> 語言應以少量固定函數組合建立程式，並具有可用於程式轉換與證明的代數。

McCarthy：

> 語言應以統一符號表示與遞迴語義形成小型生成核心，使程式能操作並延展自身語言。

Hickey：

> 語言應把值、身分、狀態、時間與處理情境分離，讓必要變化具有明確邊界。

SECTION

一百三十二、三種複雜度政治

> 【Backus：限制組合語彙，以換取代數推理;
> McCarthy：統一表示語彙，以換取生成能力;
> Hickey：分離責任語彙，以換取結構理解】

SECTION

一百三十三、三者可否混合

可以，但不是簡單相加。

例如一門新語言可同時採：

- Backus 式固定核心 Combinator；
- McCarthy 式程式資料表示；
- Hickey 式不可變值與明確 State model。

但會產生衝突：

- Macro 是否破壞代數律？
- Host effect 是否破壞純轉換？
- 開放符號生成是否破壞核心克制？
- 多種 Reference 是否增加核心表面？
- 程式代數如何處理 I/O 與時間？

SECTION

一百三十四、最終 PLDST 判定

> 【John Backus
> ：
> Algebraic Transformational Simplifier;
> John McCarthy
> ：
> Generative Symbolic Simplifier;
> Rich Hickey
> ：
> Decomplective Structural Simplifier】

SECTION

一百三十五、本文最後命題

> 簡單性不是「沒有東西」，而是決定哪些東西應該被固定、哪些東西可以生成，以及哪些東西絕不能被交纏。

因此：

> 【Simplicity
> =
> Constraint
> +
> Generativity
> +
> Separation
> -
> UntrackedCoupling】

Backus、McCarthy 與 Hickey 分別把其中一項推向極致：

- Backus 深化 Constraint 與 Algebra；
- McCarthy 深化 Generativity 與 Representation；
- Hickey 深化 Separation 與 Explicit change。

這三條道路共同構成 PLDST 對「簡單性」最重要的比較基礎。

比較組：John Backus／John McCarthy／Rich Hickey
主要語言／系統：FP／LISP／Clojure
共同命題：以函數與高階表示降低命令式偶發負擔
Backus：
核心問題＝逐字狀態與命名支配
核心策略＝Function-level、Combining forms、Program algebra
簡單性＝固定組合與可推導轉換
複雜度去向＝代數基礎、實作、I/O 與狀態整合
主要限制＝完整系統與外部世界

McCarthy：
核心問題＝符號知識與可變結構的計算
核心策略＝S-expression、Recursion、Lambda、eval
簡單性＝小型生成核心與統一表示
複雜度去向＝Runtime、GC、Macro、方言與治理
主要限制＝動態、安全、工具、分支
Hickey：
核心問題＝值、身分、狀態、時間與情境交纏
核心策略＝Immutable value、Reference semantics、Function、Host JVM
簡單性＝解除交纏與責任分離
複雜度去向＝Persistent structure、JVM、工具、學習與治理
主要限制＝宿主洩漏、動態工具、集中演化
核心比較：
Backus＝代數轉換型簡單性
McCarthy＝生成符號型簡單性
Hickey＝去交纏結構型簡單性
歸因信心：高

人物	問題	決策	簡單性類型	複雜度去向	PLDST 標記
Backus	逐字控制支配程式	Function-level programming	代數／尺度	FP system	B-F
Backus	程式難以同語言轉換	Program algebra	可推導	Algebra foundation	B-A
Backus	自由組合可能混亂	固定 Combining forms	約束型	核心選擇	B-C
Backus	歷史系統仍需狀態	Applicative state transition	鬆耦合	State model	B-S
McCarthy	符號結構長度不固定	List／S-expression	表示統一	GC／Runtime	M-S
McCarthy	符號函數需遞迴	Conditional／Lambda／Label	生成核心	Dynamic execution	M-R
McCarthy	語言需可執行定義	apply／eval	語義統一	Interpreter／Safety	M-E
McCarthy	程式需操作程式	Program as data	延展型	Macro／Dialect	M-P
Hickey	值與狀態被混同	Immutable values	去交纏	Persistent structure	H-V
Hickey	Identity 與更新規則混同	Ref／Atom／Agent／Var	責任分離	Runtime semantics	H-I
Hickey	Algorithm 綁定 Collection context	Transducer	情境分離	Library abstraction	H-T
Hickey	新語言重建成本過高	JVM symbiosis	宿主槓桿	Interop complexity	H-J

[R1] John Backus, “Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs,” Communications of the ACM 21(8), 1978.

— 馮紐曼瓶頸、Function-level programming、Combining forms、Program algebra、Applicative state transition。

[R2] Grady Booch, “Oral History of John Backus,” Computer History Museum, 2006.

— Backus 對 FP 起源、I/O 困難、完整系統限制及後期自我評價的直接敘述。

[R3] Computer History Museum Software Preservation Group, “History of John Backus’s Functional Programming Project.”

— FP、FL、FFP、文件與實作歷史保存。

[R4] John McCarthy, “Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I,” Communications of the ACM 3(4), 1960.

— S-expression、S-function、Conditional、Lambda、apply、Interpreter、List structure 與 GC。

[R5] John McCarthy, “History of Lisp,” HOPL I, 1978／1979.

— LISP 問題形成、M-expression／S-expression、eval 實作與歷史歸因。

[R6] John McCarthy et al., LISP 1.5 Programmer’s Manual, MIT Press, 1962.

— Interpreter、Compiler、Reader／Printer、GC、實作團隊與早期系統。

[R7] Rich Hickey, “Simple Made Easy,” Strange Loop, 2011.

— Simple／Easy、Complect、Composition 與結構簡單性。

[R8] Rich Hickey, “Values and Change: Clojure’s Approach to Identity and State,” Clojure official site.

— Value、Identity、State、Reference、Time、STM 與 Agent。

[R9] Rich Hickey, “A History of Clojure,” Proceedings of the ACM on Programming Languages, HOPL IV, 2020.

— Clojure 形成、JVM、設計目標、Persistent data、Concurrency 與取捨。

[R10] Clojure official Rationale、Functional Programming、Development 與 Reference 文件。

— Dynamic compiled language、Persistent collection、Sequence、Host interop、克制演化與現行語義。

[R11] PLDST-011, John Backus：從 FORTRAN 到函數級程式設計的自我反省.

— 本系列前置個案研究。

[R12] PLDST-012, John McCarthy：極小核心、符號計算與語言可延展性.

— 本系列前置個案研究。

[R13] PLDST-021, Rich Hickey：價值、身分與簡單性的分離.

— 本系列前置個案研究。

資料查核日期： 2026-07-30。

[B-F] Backus: function-level programming
[B-C] Backus: fixed combining forms
[B-A] Backus: algebra of programs
[B-V] Backus: anti word-at-a-time
[B-S] Backus: loose state coupling
[B-R] Backus: reflexive revision
[M-S] McCarthy: symbolic representation
[M-R] McCarthy: recursive core
[M-L] McCarthy: lambda and naming
[M-E] McCarthy: executable semantics
[M-P] McCarthy: program-data unity
[M-G] McCarthy: generative extensibility
[H-S] Hickey: simple-not-easy
[H-D] Hickey: decomplecting
[H-V] Hickey: immutable values
[H-I] Hickey: identity-state separation
[H-T] Hickey: transducer/context separation
[H-J] Hickey: JVM host leverage
[H-C] Hickey: conservative core evolution
[C-F] comparative function model
[C-S] comparative simplicity model
[C-R] representation politics
[C-T] time-state allocation
[C-X] extension governance
[C-B] complexity burden

SECTION

E.1 Backus FP 不是一般 Lambda 語言

重新核對圖靈獎演講：

- FP 程式被定義為沒有變量的函數；

- 新函數主要由固定 Combining forms 與簡單定義建立；
- Backus 明確把 FP 與 Lambda-calculus 型 Applicative system 比較；
- 他認為無限制的替換與組合自由會造成複雜性。

因此本文使用：

Function-level programming
而沒有把 Backus 簡化成「現代純函數語言倡議者」。

SECTION

E.2 Backus 沒有否認所有狀態

圖靈獎演講提出 Applicative state transition 系統，承認歷史敏感計算需要狀態轉換。

其要求是：

狀態轉換不應與每一個微小計算緊密耦合
而不是所有系統永遠無狀態。

SECTION

E.3 Backus 對 FP 的後期修正

口述歷史明確指出：

- Combining forms 的核心構想相對容易；
- I/O 與周邊能力難以乾淨納入；
- FP 作為完整系統最終沒有成功；
- 他仍認為資料轉換型問題適合函數式方法。

本文因此沒有把 FP 描述成已完成的通用替代。

SECTION

E.4 McCarthy 核心與實際 LISP 系統分開

1960 論文的形式核心以 S-function 與遞迴定義為主，但早期 LISP 系統已包含：

- 編譯；
- Machine-language function；
- Assignment；
- go to；
- I/O；
- Trace。

本文因此分開：

> FormalCore
> ≠
> OperationalSystem

SECTION

E.5 eval 歸因

McCarthy 提出 Universal function 的形式；Steve Russell 看出並實作其 Interpreter 意義；其他團隊成員建立 GC、Compiler、Reader／Printer 及系統。

本文不把整套 LISP 實作歸於 McCarthy 個人。

SECTION

E.6 S-expression 的歷史偶發

McCarthy 原計畫以 M-expression 作外部語法、S-expression 作內部表示。

M-expression 沒有完成普及，使 S-expression 成為主要表面。

本文將其標記為：

事故性穩定

而非完全預定設計終局。

SECTION

E.7 Hickey 的 Simple 定義

〈Simple Made Easy〉的 Simple 主要指：

- 單一責任；
- 未被交織；
- 可獨立理解與組合。

它不等同：

- 少字元；
- 初學容易；
- 熟悉；
- 無功能；
- 無狀態。

SECTION

E.8 Clojure 沒有宣稱完全純函數

官方資料明確稱 Clojure predominantly functional／impure，並提供 Ref、Atom、Agent、Var 及 Java interop。

本文使用：

函數式優先＋明確變化邊界

而不是「完全純函數語言」。

SECTION

E.9 Value 與 Identity

官方 Identity／State 文件明確區分：

- Identity 可以在不同時間具有不同 State；
- State 本身是不可變 Value；
- 新值由舊值透過函數產生；
- Reference 管理變化與協調。

本文的形式化為分析重述，不是官方原始數學公式。

SECTION

E.10 三種 Simplifier 是本文原型

下列名稱不是三位設計者自稱：

Algebraic Transformational Simplifier
Generative Symbolic Simplifier
Decomplective Structural Simplifier
其用途是把「簡單性」分解成可比較的設計風格。

PLDST-025 比較三種簡單性：

代數轉換
符號生成
結構去交纏
PLDST-026 將從設計內容轉向設計權力：

> 當語言由個人創始者、仁慈獨裁者、核心團隊、標準委員會或 RFC 社群治理時，誰有權定義「這仍是同一門語言」？

下一篇預定為：

PLDST-026：個人設計者、仁慈獨裁者與 RFC 制度——語言治理風格比較。