John McCarthy：極小核心、符號計算與語言可延展性

John McCarthy: Minimal Cores, Symbolic Computation, and Language Extensibility

v1.0 / 2026-07-30 / Neo.K / 公開版／第三部設計師個案正式研究

https://thisoneisneok.com/html/papers/pldst-012.html

英文名稱： John McCarthy: Minimal Cores, Symbolic Computation, and Language Extensibility

系列： Programming Language Designer Style Taxonomy（PLDST）

文件編號： PLDST-012

版本： v1.0

日期： 2026-07-30

作者： Neo.K

文件狀態： 公開版／第三部設計師個案正式研究

SECTION

摘要

John McCarthy 的程式語言設計風格通常被濃縮成幾個標籤：LISP、List、Recursion、eval、程式即資料。然而，這些特徵若脫離其原始問題，就容易被誤解成純粹的語法極簡或無限制元程式設計。

LISP 最初並不是為了展示一種漂亮的括號語法，而是為人工智慧研究建立一個能操作符號表達式、邏輯句子、可變大小結構與遞迴定義的計算系統。McCarthy 在 1960 年論文中明確說明，該系統源自 Advice Taker 所需要的形式化宣告與命令表達，並經多次簡化，最後建立在與特定 IBM 704 獨立的 S-expression 與 Recursive function 表示上。[R1]

LISP 的設計核心包含：

- 以 List structure 表示可變大小的符號資料；
- 使用少量 Selector 與 Constructor；
- 以 Conditional expression 和 Recursion 定義計算；
- 以 Lambda expression 表示函數；
- 讓程式與資料共享表示；
- 用 eval 同時充當語言形式定義與 Interpreter；
- 以 Garbage collection 解決自動回收；
- 以最少 Declaration 支援互動式執行。[R1][R2]

然而，把這些成果全部歸於 McCarthy 一人同樣失真。Steve Russell 看到 McCarthy 作為數學練習寫出的 eval 可以直接實作為 Interpreter，並將其手工編成 IBM 704 機器碼；Daniel J. Edwards、Timothy Hart、Michael Levin、Paul Abrahams、Phyllis Fox、Klim Maling 等人對 Interpreter、Garbage collector、Compiler、Reader／Printer、Manual 與實際系統作出關鍵貢獻。[R3][R4]

本文使用 PLDST 模型，把 McCarthy 的語言設計分成四個相位：

- 符號問題形成期：AI、Advice Taker、邏輯表達與可變結構；
- 遞迴形式核心期：S-expression、S-function、Lambda、Conditional 與 apply／eval；
- 可執行元語言期：程式作為資料、Interpreter、互動環境與系統自我延展；
- 多分支語族期：LISP 1.5 之後由不同實作、機構和社群形成多條 Lisp 路線。

本文的核心判斷是：

> McCarthy 的極簡不是「功能越少越好」，而是尋找一組能同時表示資料、程式與語言定義的生成性核心，使語言能由其使用者和實作者繼續擴張。

其深層風格可表示為：

> 【統一表示
> +
> 遞迴定義
> +
> 可執行語義

但這種風格也有代價：

- 語法與資料的統一可能形成難以限制的元編程權力；
- 動態表示與晚期檢查把部分安全責任交給使用者及工具；
- 小核心可以生成龐大的方言與生態差異；
- 原本計畫使用的 M-expression 沒有完成，S-expression 從中間表示變成主要表面語法，形成一次重要的事故性穩定；
- LISP 後來的成功不能被視為 McCarthy 個人持續治理的單一路線。

關鍵詞： John McCarthy、LISP、符號計算、S-expression、eval、遞迴、垃圾回收、程式即資料、語言延展、PLDST

SECTION

一、本文研究對象

本文主要研究：

- 1956–1958 年的 LISP 問題形成；
- 1958–1962 年的設計與早期實作；
- 1960 年〈Recursive Functions of Symbolic Expressions〉；
- LISP 1／1.5；
- McCarthy 1978／1979 年的歷史回顧；
- 1980 年對 LISP 存續特徵的總結；
- 後期口述歷史中的自我敘述。

本文不把 Scheme、Common Lisp、Interlisp、Maclisp、Emacs Lisp、Clojure 等後代語言的全部設計歸於 McCarthy。

SECTION

二、創始設計與實作必須分開

LISP 1.5 Programmer’ s Manual 的原始歸因指出：

- 整體設計：John McCarthy；
- Interpreter：Stephen R. Russell、Daniel J. Edwards；
- Reader／Printer：McCarthy、Klim Maling、Edwards、Paul W. Abrahams；
- Garbage collector 與 Arithmetic：Daniel J. Edwards；
- Compiler／Assembler：Timothy P. Hart、Michael I. Levin；
- 早期 Compiler：Robert Brayton；
- LISP I Manual：Phyllis A. Fox；
- 其他成員亦提供程式和建議。[R4]

  因此：

  問題設定與核心形式：McCarthy 高
  eval 的形式定義：McCarthy 高
  eval 成為實際 Interpreter：Russell 關鍵
  運行系統與 Compiler：團隊共同
  Lisp 語族後期演化：多共同體

SECTION

三、McCarthy 自己也承認歷史歸因不完整

他在〈History of Lisp〉開頭直接指出：

- 該稿對許多實作者和思想貢獻者提及不足；
- 自己過去曾在歸因上犯錯；
- 後來發現其他文件研究者對部分歷史有更準確理解。[R2][R5]

  這使 McCarthy 個案非常適合 PLDST 的多主體歸因模型。

SECTION

四、Advice Taker 與符號推理

LISP 的起點是建立能操作：

- 宣告句；
- 命令句；
- 邏輯關係；
- 形式化知識；
- 推論過程；

  的系統。

  這要求資料不是固定數值陣列，而是具有：

- 可變大小；
- 嵌套結構；
- 任意深度；
- 符號名稱；
- 可被分解與重組；

  的表達。

SECTION

五、為什麼是 List

McCarthy 在口述歷史中說明，List 能讓程式的結構與資料結構相互對應。

例如一個和式可以表示為：

```
PLUS
├── term1
├── term2
└── term3
```

程式不需要先把整個表達式解析成固定長度陣列，而可以：

- 取出 Operator；

- 遍歷 Operand；
- 對子式遞迴；
- 重組新表達式。

SECTION

六、Recursion 不是裝飾功能

形式微分的定義天然具有遞迴結構：

> D(a+b)
> =
> D(a)+D(b)

如果表達式是一個和，微分程式就對各子表達式再次呼叫自己。

McCarthy 回顧，他在 1958 年發現 FORTRAN 不允許這種遞迴，因此判斷需要新語言。[R3]

SECTION

七、問題—資料—控制三者對齊

McCarthy 的重要設計方式是讓：

> Structure_(problem)
> ≈
> Structure_(data)
> ≈
> Structure_(program)

這是 Lisp 可讀性與元編程能力的共同來源。

SECTION

八、S-expression

S-expression 提供：

- Atom；
- Pair／List；
- Nested structure；
- 統一外部表示；
- 可由同一 Reader 解析；
- 可由程式自身操作。

  其核心能力不是括號本身，而是規則化樹狀表示。

SECTION

九、S-function 與 M-expression

McCarthy 原本區分：

- S-expression：資料與內部表示；
- M-expression：較接近一般數學符號的函數書寫法。

1960 年論文承認，S-expression 表示可寫且有一定可讀性；若為提高表面可讀性而改變符號，也會降低結構規則性。[R1]

原計畫是由 M-expression 翻譯為 S-expression，但這套外部語法沒有完成及普及。

結果是：

> IntermediateRepresentation
> →
> SurfaceLanguage

這是一個典型的事故性穩定。

SECTION

十、少量基本函數

早期核心包括：

- atom；
- eq；
- car；
- cdr；
- cons；
- cond；
- quote；
- lambda；
- label；
- apply；
- eval。

這些機制本身不多，卻能組成：

- Tree traversal；
- Substitution；
- Differentiation；
- Interpreter；
- Program transformation；
- Symbolic reasoning。

SECTION

十一、Conditional expression

條件不是只作為控制陳述，而被放入表達式和函數定義。

因此：

> Branch
> ∈
> Expression

而不需把計算分裂成表達式世界與陳述世界。

SECTION

十二、Lambda 與 Recursive definition

Lambda expression 提供匿名函數與參數綁定；label 等機制使函數能以名稱遞迴引用自己。

這讓語言核心同時具有：

- 函數抽象；
- 高階操作；
- 遞迴；
- 形式計算。

SECTION

十三、原始意圖

McCarthy 說明，他寫出 Universal function eval，主要是數學練習：

- 展示該形式具有通用性；
- 讓 Universal function 比 Universal Turing machine 更透明；

- 向數學家說明遞迴函數形式的價值。[R3]

SECTION

十四、Russell 的實作洞見

Steve Russell 看出：

若把 eval 寫成機器程式
它就是 LISP Interpreter
McCarthy 起初有所懷疑，Russell 仍完成實作。[R3]

因此：

> FormalDefinition
> xrightarrowRussell
> ExecutableInterpreter

這不是單純把 McCarthy 的完整計畫照表實作，而是實作者對形式定義作出的關鍵再解讀。

SECTION

十五、形式語義與實作合一

eval 同時是：

- 語言定義；
- 解釋器模型；
- 元循環能力的起點；
- 程式分析工具；
- 教學模型；
- 自我描述機制。

  McCarthy 後來把「eval 同時作為形式定義與 Interpreter」列為 LISP 長期存續的重要特徵。[R6]

SECTION

十六、這不是完整的自我實作神話

早期 Interpreter 仍依賴：

- IBM 704 機器碼；
- 手工實作；
- Reader；
- Printer；
- Garbage collector；
- Arithmetic；
- Runtime convention；
- Compiler。

所以：

語言可用自己的表示描述語義
≠
整個系統不需要外部基礎

SECTION

十七、統一表示的能力

當程式也以 List／S-expression 表示，程式可以：

- 讀取其他程式；
- 建立新程式；
- 轉換程式；
- 解釋程式；
- 儲存程式；

- 在互動環境中修改定義。

SECTION

十八、元編程與語言延展

統一表示使使用者可以建立：

- Macro；
- DSL；
- Compiler；
- Theorem prover；
- Symbolic algebra；
- Editor；
- Debugger；
- Language extension。

McCarthy 1980 年總結認為，程式作為 LISP 資料，降低了 System programmer 與 Application programmer 的分隔，許多使用者的「改進」最後成為語言改進。[R6]

SECTION

十九、不是所有語法樹都等同 Lisp

現代語言也有 AST，但 Lisp 的特殊之處在於：

- 來源與資料表示非常接近；
- Reader 與 Printer 形成標準轉換；
- 語言原生操作可處理表示；
- eval 直接賦予表示執行含義；

- 互動環境使修改立即可見。

SECTION

二十、表達自由的代價

同一能力可能造成：

- Macro 方言；
- 非局部展開；
- 工具分析困難；
- 隱藏控制流；
- Runtime Code generation；
- 安全邊界擴大；
- 不同專案形成不同有效語言。

因此：

> Extensibilityuparrow
> ⇒
> GovernanceNeeduparrow

SECTION

二十一、Erasure problem

List structure 會動態建立大量節點。

若要求每個 AI 程式設計者手工追蹤：

- 哪些 List 不再使用；
- 哪些節點共享；

- 哪些資料可回收；

  符號計算的抽象就會被記憶體管理摧毀。

SECTION

二十二、自動回收

McCarthy 將 Garbage collection 列為 LISP 的長期核心特徵；早期實際 Garbage collector 則由 Daniel J. Edwards 等團隊成員完成。[R4][R6]

複雜度配置為：

> C_(manual lifetime)downarrow
>
> C_(runtime)uparrow

SECTION

二十三、與符號計算模型一致

GC 不是與語言無關的附加便利，而是使：

- 動態 List；
- 結構共享；
- 遞迴生成；
- 互動式實驗；
- 程式轉換；

  可以實際運作的 Runtime 責任。

SECTION

二十四、最少前置宣告

McCarthy 把「最少 Declaration，讓 LISP statement 可在 Online environment 中直接執行」列為 LISP 存續特徵。[R6]

這使語言適合：

- 試驗；
- 即時修改；
- 探索式 AI；
- REPL；
- 增量建立系統；
- 長時間運行環境。

SECTION

二十五、互動性與動態性

互動環境需要：

- Late binding；
- Dynamic loading；
- Runtime inspection；
- 可重新定義；
- 立即回饋。

它提高研究生產力，也增加：

- 可重現性；
- 靜態驗證；
- 部署一致性；
- 版本狀態；

的治理問題。

SECTION

二十六、極小核心不是封閉核心

McCarthy 的核心是生成性的：

> SmallKernel
> +
> ProgramDataUnity
> +
> Eval
> +
> GC
> +
> InteractiveEnvironment
> →
> OpenEndedSystem

SECTION

二十七、擴展能力放在哪裡

LISP 把大量能力放在：

- Library；
- User-defined functions；
- Program transformation；
- Macro；
- Runtime；
- Dialect；

- Interactive environment。

  因此語言核心可以小，整個 Lisp 系統卻可以極大。

SECTION

二十八、可擴展性與標準化張力

當不同機構各自改進 Lisp，便產生：

- 不同資料表示；
- 不同 Scope；
- 不同 Compiler；
- 不同 Object system；
- 不同 Macro；
- 不同 Module；
- 不同編輯器與 OS。

  這既是 Lisp 生命力，也形成方言分裂。

SECTION

二十九、1962 後的多分支

McCarthy 自己將 LISP 歷史分為：

- 1956–1958：主要思想形成；
- 1958–1962：語言實作及 AI 應用；
- 1962 後：發展成多條分支。[R2]

  這表示：

> Style_(McCarthy)
> ≠
> Style_(all Lisp descendants)

SECTION

三十、後創始者制度

不同 Lisp 路線後來由：

- MIT；
- Stanford；
- BBN；
- Xerox；
- Symbolics；
- Scheme 社群；
- Common Lisp 標準化；
- Emacs；
- 其他研究與商業機構；

  共同發展。

  McCarthy 很早便不再是所有 Lisp 的直接治理者。

SECTION

三十一、Lisp 存續的判斷

McCarthy 1980 年把多項特徵視為某種程式語言空間中的近似局部最優：

- Symbolic expression；

- List representation；
- 少量 Constructor／Selector；
- Composition；
- Recursion；
- Lambda；
- Program as data；
- eval；
- GC；
- Online environment。[R6]

這是後期回顧，不應被當成 1958 年已完整預見的固定設計藍圖。

SECTION

三十二、問題形成期

目標：讓機器操作形式知識與邏輯句子
核心需要：可變結構、符號、遞迴

SECTION

三十三、形式核心期

目標：以少量遞迴函數描述符號計算
核心：S-expression、Lambda、Conditional、Recursion

SECTION

三十四、可執行語義期

目標：讓語言定義直接成為 Interpreter
轉折：Russell 實作 eval

SECTION

三十五、系統延展期

目標：讓使用者以程式操作程式、增量改進系統
機制：Program as data、eval、GC、Online environment

SECTION

三十六、多分支語族期

結果：可延展性產生大量 Dialect 與制度
權力：由創始者轉向多個共同體

SECTION

三十七、問題 framing

McCarthy 從問題結構出發：

> 若計算對象是形式句子與符號結構，語言本身就應能自然表示、分解、重組和遞迴處理這些對象。

SECTION

三十八、價值優先序

```
V_(McCarthy)
≈
(
FormalGenerality,
SymbolicRepresentation,
Recursion,
Uniformity,
Extensibility,
Interactivity,
MathematicalSemantics
)
```

SECTION

三十九、核心—擴張配置

核心：少量表達與列表操作
擴張：函數、程式生成、Macro、Interpreter、Dialect
這是 生成式極簡，不是功能貧乏。

SECTION

四十、複雜度配置

> C_(syntax core)downarrow

> C_(runtime)
> +
> C_(dynamic reasoning)
> +
> C_(extension governance)
> uparrow

SECTION

四十一、責任配置

語言與 Runtime 承擔：

- Memory recovery；
- Representation；
- Evaluation；
- Interactive execution。

使用者承擔：

- 動態資料契約；
- Program transformation correctness；
- Extension discipline；
- Macro／eval 安全；
- Dialect interoperability。

SECTION

四十二、顯式—推導偏好

McCarthy 不是以靜態型別推導為主，而是：

- 讓結構本身明示；
- 讓 Interpreter 根據資料表示求值；
- 以動態 Symbolic representation 保留彈性；
- 以最少 Declaration 支援即時使用。

SECTION

四十三、安全—自由配置

LISP 的自由來自：

- 動態資料；
- Program as data；
- eval；
- Runtime modification。

安全主要依賴：

- Runtime checks；

- Programmer discipline；
- Library convention；
- 後來 Dialect 的工具。

這不是以現代 Memory-unsafe 意義為中心，而是語義、效果與元編程權力的治理問題。

SECTION

四十四、LISP 不只是 McCarthy 的作品

若沒有 Russell、Edwards、Hart、Levin、Fox、Abrahams、Maling 等人的工作，形式核心不會自動變成可使用系統。

SECTION

四十五、S-expression 的表面成功具有偶然性

McCarthy 原本預期使用 M-expression 作外部語法；S-expression 長期成為 Lisp 表面，部分來自：

- Interpreter 已能直接使用；
- Reader／Printer；
- 實作便利；
- 使用者採用；
- 原翻譯器未完成。

因此括號語法不能完全被解釋成創始者預先規劃的最終美學。

SECTION

四十六、eval 不是所有延展性的必要條件

Macro、Compiler API、Module、Template 與 AST transformation 也可提供延展性。

eval 的優勢與風險取決於：

- Scope；
- Environment；
- Security；
- Compilation；
- Tooling；
- Deployment。

SECTION

四十七、小核心可能形成巨大有效語言

Lisp 的每個專案都可能：

- 建立 Macro；
- 修改 Reader；
- 建立 DSL；
- 使用不同慣例。

這證明：

> SmallKernel
> not⇒
> SmallEffectiveLanguage

SECTION

四十八、動態符號模型不適合所有問題

數值密集、硬即時、靜態安全、嵌入式記憶體限制等領域，可能需要不同配置。

SECTION

四十九、歷史回顧有記憶偏差

McCarthy 自己承認其 1978 歷史回憶並非所有細節最可靠版本，因此本文以：

- 1960 原始論文；
- LISP 1.5 Manual；
- 保存程式；
- 口述歷史；
- 後來文件研究；

交叉使用。

時期 問題 決策 複雜度去向 風格

1956–1958 AI 需操作形式句子 List／symbolic structure Runtime representation 問題對齊

1958 FORTRAN 缺遞迴 新語言＋Recursive functions 語言語義 遞迴核心

1959–1960 需要通用形式定義 apply／eval Interpreter 可執行語義

1960 動態 List 回收 GC Runtime 自動管理

1960s 使用者需改進語言 Program as data 使用者／系統共同 可延展

1962 後 多機構需求 多 Dialect 社群與標準 分支演化

SECTION

五十、主要原型

McCarthy 同時屬於：

- 符號問題對齊設計者；
- 生成式極簡架構師；

- 可執行語義設計者；
- 元語言與延展性設計者；
- 形式計算理論建築師。

SECTION

五十一、不適合的簡單標籤

不應只稱：

括號語言發明者
動態語言設計者
函數式程式設計者
eval 發明者
AI 語言之父
較精確的描述是：

> 把符號資料、程式表示與語言定義壓縮到同一個遞迴結構中，從而讓語言能操作並延展自身的設計者。

SECTION

五十二、最重要的設計連續性

從 Advice Taker 到 LISP：

> KnowledgeStructure
> →
> SymbolicData
> →
> RecursiveProgram
> →
> ExecutableRepresentation

SECTION

五十三、最重要的制度不連續性

LISP 的核心由 McCarthy 強烈塑造，但 Lisp 語族的長期演化迅速進入：

- 團隊實作；
- 機構分支；
- 方言競爭；
- 標準化；
- 生態治理。

SECTION

五十四、最重要的事故性穩定

S-expression 從：

統一內部表示

變成：

長期表面語法與文化身份

這不是單純錯誤，而是實作、形式規則與使用者採用共同選出的結果。

John McCarthy 的程式語言設計不是從「想做一門極簡語言」開始，而是從一個需要：

- 表示知識；
- 處理邏輯；
- 操作不定長結構；
- 定義遞迴計算；
- 在互動環境中快速實驗；

的問題開始。

他最深刻的設計不是任何單一關鍵字，而是建立以下閉環：

> 【資料可表示程式
> →
> 程式可操作資料
> →
> 語言定義本身可執行
> →
> 使用者可繼續擴展語言】

本文對 McCarthy 的 PLDST 判定為：

> 【Symbolic Problem Modeler
> →
> Recursive Core Architect
> →
> Executable Metalanguage Designer】

其核心優勢是：

- 少量機制具有極高生成能力；
- 形式語義與實作可以直接連接；
- Program as data 使語言具備自我延展能力；
- GC 與 Online environment 讓符號實驗成為實際工作流。

其核心風險是：

- 小核心生成大規模方言差異；
- eval 與 Macro 擴大安全和工具負擔；
- 動態模型把部分錯誤延至 Runtime；
- 統一表示不等於所有領域都具有最佳表達；
- 創始思想容易掩蓋關鍵實作者與後期共同體。

最終原則為：

> 【以最少核心統一資料、程式與語義
> ∧
> 讓系統可由使用者繼續生長
> ∧
> 承認生長必須帶來新的治理與安全邊界】

人物：John McCarthy
主要語言／系統：LISP、Advice Taker 相關形式
核心時期：1956–1962
主要問題：符號、邏輯與可變結構的計算
主要策略：List、Recursion、Lambda、Conditional、eval
核心擴張：Program as data、GC、Interactive environment
複雜度去向：Runtime、動態推理、語言治理
責任去向：Interpreter 與 GC 承擔執行細節
主要保護對象：AI 研究者、符號程式設計者
主要限制：方言分裂、安全、工具與動態錯誤
歸因信心：高

[R1] John McCarthy, “Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I,” Communications of the ACM 3(4), 1960, pp. 184–195.

— LISP 原始形式、S-expression、S-function、Lambda、Conditional、apply／eval 與 IBM 704 實作。

[R2] John McCarthy, “History of Lisp,” HOPL I draft, 1978／1979.

— LISP 前史、思想形成、實作、分支、M-expression 與歸因不確定性。

[R3] Nils J. Nilsson, “Oral History of John McCarthy,” Computer History Museum, 2007.

— FORTRAN 遞迴限制、List 選擇、eval 原意及 Steve Russell 的 Interpreter 洞見。

[R4] John McCarthy et al., LISP 1.5 Programmer’s Manual, MIT Press, 1962; Computer History Museum LISP 1.5 preservation materials.

— 整體設計、Interpreter、GC、Compiler、Reader／Printer、Manual 與團隊歸因。

[R5] John McCarthy historical pages, Stanford Formal Reasoning Group.

— McCarthy 對自身記憶限制、Herbert Stoyan 文件研究及歷史資料的說明。

[R6] John McCarthy, “LISP—Notes on Its Past and Future,” ACM LISP and Functional Programming Conference, 1980.

— LISP 長期存續特徵、Program as data、eval、GC 及 Online environment。

[R7] Computer History Museum Software Preservation Group, “LISP History Collection.”

— 原始程式、手冊、方言、實作、人物與多分支歷史。

[R8] John McCarthy, “A Basis for a Mathematical Theory of Computation,” 1961–1963.

— 遞迴函數、語言形式與程式變換的數學理論背景。

[T-P] Symbolic problem formation
[T-R] Recursive formal core
[T-E] Executable semantics
[T-X] Extensible system
[T-D] Dialect／community divergence
[S-S] Symbolic representation
[S-M] Minimal generative kernel
[S-R] Recursive architecture
[S-E] Eval／executable semantics
[S-P] Program-data unity
[S-I] Interactive extensibility

SECTION

D.1 LISP 的形成時期

第二輪重新核對 McCarthy 的〈History of Lisp〉：

- 1956 年夏至 1958 年夏是多數關鍵思想的形成期；
- 1958 年秋至 1962 年是實作與 AI 應用期；
- 1962 年後發展轉為多條路線；
- 這是 McCarthy 的歷史分期，不表示所有特徵在 1958 年已以最終形式存在。

  本文因此把「問題形成」「形式核心」「可執行系統」「多分支語族」分開，而沒有把後期 Lisp 特徵全部回寫到第一天。

SECTION

D.2 eval 與 Interpreter 歸因

第二輪重新核對 1960 原始論文、McCarthy 口述歷史與 LISP 1.5 Manual：

- McCarthy 寫出 Universal S-function apply／eval 的形式；
- 他主要把它視為展示通用性與數學透明度的練習；
- Steve Russell 看出將 eval 手工編成機器語言即可形成 Interpreter；
- McCarthy 表示自己一開始對此有所懷疑；
- LISP 1.5 Manual 把 Interpreter 程式歸於 Stephen B. Russell 與 Daniel J. Edwards。

  因此本文不使用「McCarthy 單人寫出第一個 Lisp Interpreter」的說法。

SECTION

D.3 LISP 1.5 團隊歸因

第二輪直接核對 Programmer’s Manual Preface：

整體設計：John McCarthy
Manual：Michael I. Levin
Interpreter：Stephen B. Russell、Daniel J. Edwards
Read／Print：McCarthy、Klim Maling、Edwards、Paul W. Abrahams
Garbage collector／Arithmetic：Daniel J. Edwards
Compiler／Assembler：Timothy P. Hart、Michael I. Levin
較早 Compiler：Robert Brayton
LISP I Manual：Phyllis A. Fox
本文保留 Manual 的當時正式歸因，也承認後來文件研究可能對更細部貢獻作出修正。

SECTION

D.4 Garbage collection

第二輪核對 McCarthy 1980 年回顧與 LISP 1.5 保存資料：

- McCarthy 把 Garbage collection 列為 LISP 長期可行特徵之一；
- 實際 LISP 1.5 Garbage collector 與 Arithmetic feature 由 Daniel J. Edwards 完成；
- 因此應分開：

- 將自動回收納入語言／系統方向的設計功勞；
- 具體回收器的實作功勞。

本文沒有把 GC 的所有發明與後續演算法全部歸於 McCarthy。

SECTION

D.5 M-expression 與 S-expression

第二輪重新核對 1960 論文與〈The Implementation of LISP〉：

- McCarthy 明確使用 M-expression 表示 Meta-level function notation；
- S-expression 用於符號資料與函數表示；
- 原計畫包含精確定義及翻譯 M-expression；
- 該專案既未正式完成，也未在某一時間點被明確宣布放棄；
- eval Interpreter、Reader／Printer 和使用實踐使 S-expression 直接作為程式表示變得方便；
- 因此「S-expression 原本只是一個短暫中間表示」是過度簡化；較準確的說法是：它原本主要承擔資料及規則化程式表示，而 M-expression 被預期作為較友善外部表示，但後者未完成，前者逐步成為實際表面。

本文已用「事故性穩定」描述結果，而不是聲稱 McCarthy 完全反對 S-expression 語法。

SECTION

D.6 「程式即資料」的邊界

第二輪核對 McCarthy 1980 年的存續特徵清單：

- LISP program 可表示為 LISP data；
- 這使 Application programmer 能修改與改進系統；
- eval 可同時作為語言形式定義與 Interpreter；
- 這種結合是 Lisp 延展性的基礎之一。

本文沒有把後來成熟 Macro system、Common Lisp Reader macro、Scheme hygiene 等全部歸於 McCarthy 的原始設計。

SECTION

D.7 McCarthy 的歷史自我修正

McCarthy 的 Stanford 歷史頁明確說：

- 1978 年的 LISP History 依其記憶撰寫；
- Herbert Stoyan 對原始文件的研究在若干地方更準確；
- McCarthy 的稿件也承認對實作者提及不足及歸因不確定。

因此本文將：

同期論文與手冊
>
保存程式與正式 Preface
>
後期口述與回憶
作為史實校準順序，而不是以創始者回憶壓過全部文件。

SECTION

D.8 「極小核心」的分析邊界

「生成式極簡架構師」是 PLDST 推論，而不是 McCarthy 本人正式使用的流派名稱。

它依據：

- 少量 Selector／Constructor；
- Conditional；
- Recursion；
- Lambda；
- Program-data unity；

- eval；
- GC；
- Online environment；

  共同產生高擴展能力。

  它不表示 LISP 1.5 完整系統、後代 Lisp 或所有 Lisp 程式都很小或容易理解。