可执行考古学：从其 IPL-V 源代码中重新唤醒逻辑理论家

Jeff Shrager，博士
Bennu Climate, Inc.
jshrager@gmail.com

arXiv:2603.13514v1 [cs.AI] 13 Mar 2026

摘要——**逻辑理论家（LT）由艾伦·纽厄尔、J. C. Shaw 和赫伯特·西蒙于 1955–1956 年创建，被广泛认为是第一个人工智能程序。尽管最初的概念模型于 1956 年提出，但随着底层信息处理语言（IPL）的发展，该系统经历了多次迭代。本文介绍了一种新的 IPL-V 解释器的构建过程。该解释器以 Common Lisp 编写，并基于直接从 Stefferud 的 1963年兰德技术报告中转录的代码，忠实地重新唤醒了逻辑理论家。Stefferud 的版本代表了将最初的启发式逻辑重新编码为标准化 IPL-V 的教学性尝试。重新唤醒的 LT 成功证明了《数学原理》第2章中 23 个尝试证明的定理中的 16 个；在其搜索限制范围内，这些结果在历史上与原系统的行为一致。据作者所知，这是半个多世纪以来首次成功执行原始逻辑理论家代码。**

索引词—**逻辑理论家、IPL-V、人工智能史、软件考古学、A. Newell、H. A. Simon、J. C. Shaw、列表处理、数字保存**

I. 引言

1956年夏天，一小群研究人员齐聚达特茅斯学院，参加一场关于约翰·麦卡锡所称“人工智能”的研讨会。其中有艾伦·纽厄尔和赫伯特·西蒙，他们带来了一样其他与会者都没有的东西：一个可运行的人工智能程序。其他参与者带来的是提案和想法，而纽厄尔和西蒙带来的是一个确实能运行的系统——一个能够发现数学定理证明的程序。[1]

逻辑理论家（LT） [1] 最终证明了怀特海和罗素《数学原理 [3], 》第2章前52个定理中的38个，有些情况下找到的证明甚至比人类作者给出的证明更加简洁优雅。对于定理 2.85，LT找到了一个比《数学原理》中发表的证明更短、更直接的证明。西蒙把这个新证明展示给伯特兰·罗素本人看，罗素“欣喜地作出了回应” [11,第 167]页。当时还是本科生、正在西蒙的研究生课程上课的爱德华·费根鲍姆，回忆起项目的重要意义首次被宣布的那一刻。正如他后来对Pamela McCorduck所说：“那是在圣诞假期之后——1956年1月——赫伯·西蒙走进教室，说道，‘圣诞节期间，艾伦·纽厄尔和我发明了一台会思考的机器。’我们都一脸茫然” [11, 第 138]页。

除了复现《数学原理》的证明之外，逻辑理论家还引入了启发式搜索策略——手段—目的分析和子目标分解——这些策略源自对人类问题求解过程的协议分析。这些思想后来成为人工智能和认知架构的基础，例如通用问题求解器（GPS）、Soar、ACT 和 EPAM [15]， [16]。因此，LT 处于人工智能与认知心理学融合的最初阶段。

LT 使用信息处理语言（IPL）实现。这是一种由同一团队——纽厄尔、肖和西蒙——专门为构建认知模型而创建的语言 [4]。IPL 引入了列表操作、动态内存分配、作为一等对象的符号、递归以及高阶函数。它明确地是 Lisp 的直接先驱 [5]——麦卡锡本人也承认 IPL 对 Lisp 的影响。然而，Lisp 已经蓬勃发展了近七十年，IPL 却自 20 世纪 60 年代中期以来便已消亡；20 世纪 60 年代，尽管在各种机器上曾存在许多 IPL-V 实现，但在现代环境中似乎没有可用的维护中实现。作者所知的唯一一次现代尝试，是 Richman 未发表的 IPL-V 解释器；该解释器使用 Microsoft BASIC 编写，Richman 曾在 20 世纪 90 年代初用它运行 EPAM IV [17]。[2]

本文介绍了如何用 Common Lisp 构建完整的 IPL-V 解释器，并依据直接从 Stefferud 于1963年撰写的兰德公司技术报告中转录的代码，忠实地重现逻辑理论家 [8]。此次重建中运行的 LT 代码有 99% 以上是该报告中的原始代码。重现后的 LT 证明了《数学原理》中的 23 个尝试证明的定理中的 16 个，这一结果与历史上已知的原始LT行为一致。

这项工作展示了一种可称为可执行考古学的方法：通过忠实重建并执行原始代码，研究具有重要历史意义的计算系统。早期人工智能并非仅仅是理论设想，而是能够运行的程序；仅凭已发表的描述，无法全面理解其详细行为。重现这些系统，为探究人工智能的起源提供了新的实证视角。

在此前的工作中，包括本文作者在内的团队恢复了魏岑鲍姆最初的 ELIZA，使其能够在重

[1]值得一提的是，阿瑟·塞缪尔的跳棋程序比逻辑理论家早几年就已问世，而且它同样采用了启发式方法。不过，塞缪尔的目标是开发一个跳棋程序，而纽厄尔、西蒙和肖的明确目标则是构建一个能够跨多个领域模拟人类认知的模型。塞缪尔的成果本身也十分出色——他还开发了可能是第一个机器学习程序，也是第一个能够通过自我对弈进行学习的系统。

[2]Richman 在写给作者的信中表示（个人通信，2025年9月26日，经许可引用）：“我用 Microsoft Basic 实现了 IPL-V，依据的是我在卡内基梅隆大学图书馆找到的 IPL-V 手册。”

建的 CTSS 上运行，该 CTSS 则运行于仿真的 IBM 7094 [9] 上。尽管逻辑理论家和 ELIZA 处理的问题截然不同——前者是形式化定理证明，后者是自然语言对话——但这两个程序都源自同一条符号编程脉络，该脉络始于 IPL，后来在 Lisp 及相关表处理系统中得以实现。两者都可以被视为一种作用于符号列表结构之上的基于规则的控制系统。不过，ELIZA 项目依托于一个仿真老式硬件和操作系统的栈，而逻辑理论家的复现则仿真了抽象的 IPL-V机器。[3]

II. 历史背景

A. 逻辑理论家的起源与版本

逻辑理论家的故事始于20世纪50年代初的兰德公司。赫伯特·西蒙和艾伦·纽厄尔与 J. C.（克利夫）肖合作，试图通过操纵符号来模拟复杂思维。然而，该程序的历史以信息处理语言不同版本之间一系列不断变化的规格说明为特征。

1956年兰德报告（P-868）中描述的系统是一个“纸面”版本，通常被称为 IPL-I 或“逻辑语言”（LL）。1956年夏举行达特茅斯研讨会时，该程序尚未在机器上运行；纽厄尔和西蒙指出，它“基本上已经完成规格定义”，手工模拟表明，它应能证明《数学原理》第2章中的六十多个定理。在程序于任何机器上运行之前，纽厄尔和西蒙曾广为人知地用手模拟了其中的部分功能，借助的是“人工计算员”——由家人和学生充当程序组件。

直到1956年末和 1957 [10] 年初在JOHNNIAC上实现IPL-II之后，机器可执行的证明才得以生成。广为引用的结果——LT证明了《数学原理》最初52个定理中的38个，其中包括发现了比怀特海和罗素给出的证明更简洁的定理2.85证明——源自这些1957年的运行。

本次复现所用的代码转录自Stefferud的1963年备忘录，代表了第三个不同阶段。Stefferud的版本是将原始启发式逻辑重新编码为IPL-V的一种教学性改编；IPL-V是首个广泛向公众开放使用的语言版本。因此，尽管我们的复现忠实执行了兰德公司团队发布的原始代码，它恢复的却是对1956年概念模型进行“现代化”后的1963年版本。

B. IPL：符号处理与列表处理的发明

为了实现 LT，纽厄尔、肖和西蒙需要一种适用于符号推理的编程语言。当时并不存在这样的语言，于是他们创造了“信息处理语言”。如前所述，IPL 经历了多个迭代版本。最早的概念版本（IPL-I 和 IPL-II）支持了 1956年的手工模拟以及随后在 JOHNNIAC 上的

[3]讽刺的是，在 IPL 的直接后继语言 Common Lisp 中。

表 I 逻辑理论家与 IPL 的演进

Year	版本	平台	历史里程碑
1956	IPL-I（LL）	手动	概念规格；仅经手工模拟。
1957	IPL-II	JOHNNIAC	已证明《数学原理》中的 52 个定理中的 38 个。
1958	IPL-IV	IBM 704	兰德公司内部使用；正在过渡到 IPL-V。
1963	IPL-V	各种	Stefferud 的教学用 LT 实现。
2025	IPL-V	Lisp/LLM	使用 Stefferud 的源代码进行复现。

运行。20世纪60年代初发布的 IPL-V [4], 才是后来成为正式公共标准的版本。

IPL-V运行在一种围绕单元组织的抽象机器上。每个单元都关联着一个符号。这里没有一个唯一的专用栈；每个单元都可能是一个符号栈，可供压入和弹出。符号是IPL-V的核心：它们是一等对象，每个单元、数据结构、例程和控制元素都可通过一个符号（其名称）寻址。一组控制单元（H0–H5）负责管理执行：H0作为参数和结果的“通信”单元，H1同时作为程序计数器和调用栈，H2是用于动态内存分配的空闲链表，H3则是周期计数器。十个工作单元（W0–W9）提供临时存储。IPL-V中的一切——数据、例程，甚至程序本身——都是单元列表，因此也是符号列表。调用例程时，要将其符号（名称）压入H1。IPL机器会解释该单元中指示的操作，然后继续执行所指示的下一个单元。巧妙之处在于，由于H1同时充当程序计数器和调用栈，除了基本的栈操作指令外，运行这台机器无需任何特殊机制。

该语言的内置操作称为J函数（因为其名称均以“J”开头：J0、J1、J2，依此类推）。这类函数约有150个，实现了从基本列表操作到“生成器”（迭代器）协议等各种操作。在最初的IPL-V中，许多J函数本身也是用IPL编写的，不过其中一些是在真实机器层面实现的。

如上所述，IPL是Lisp [5]的直接先驱。它引入了列表和符号操作、程序本身也是列表这一理念、动态内存分配、递归以及高阶函数。在花费数月实现IPL-V解释器之后，我可以说明，IPL基本上具备了Lisp所拥有的一切，但有两个关键例外：同构语法和自动垃圾回收。Lisp将代码与数据统一在一种简洁优雅的表示法中，而IPL代码则采用汇编语言风格来表达；这种风格虽然功能强大，却远比Lisp难以读写。此外，Lisp引入垃圾回收来自动完成内存回收，而IPL则要求显式管理空闲存储。

值得注意的是，原则上，IPL-V 程序可以在运行时修改自己的代码——这正是 Lisp 因之而备受称道的自修改能力。由于 IPL-V 过程本质上只是列表，而列表可以由 IPL-V 程序操纵，因此没有什么能阻止程序重写自己的过程。实际上，这会比在 Lisp 中麻烦得多，恰恰是因为它缺乏同像语法

，但这种能力确实存在。麦卡锡的天才之处不在于从头发明列表处理 *de novo*，而在于找到了恰当的语法形式和运行时形式，使这些理念——纽厄尔、西蒙和肖早已提出的理念——变得优雅、易懂且实用。

III. 源材料与转录

与 ELIZA 的复原工作不同，后者始于在麻省理工学院维森鲍姆的档案中戏剧性地发现了原始源代码打印件，[9], 逻辑理论家的源材料早已公开可得。两份文献不可或缺：

1) **Stefferud（1963年）**，逻辑理论家机器：一种模型启发式程序，兰德公司备忘录 RM-3731 [8]。这份报告完整列出了以 IPL-V 编写的 LT 程序，并详细描述了 LT 运作的各个方面，其中包括大量流程图。

2) **纽厄尔等人（1964年）**，信息处理语言-*V* 手册，第 2 版 [4]。这是 IPL-V 的权威参考资料，涵盖所有 J函数的语义、存储单元结构约定和生成器协议，并收录了大量学生练习与示例程序。

这两份文献的写作都极为出色。不过，偶尔仍有一些含糊之处——某些地方的说明默认读者熟悉 20 世纪 60 年代计算领域的惯例（例如 BCD字符编码），而这些如今已不再是常识。

LT 代码由作者和 Anthony Hay 将 Stefferud 报告中的内容转录到 Google电子表格中。随后，Rupert Lane 使用他的“gridlock”工具 [13], ——一种能够从 PDF文档中准确提取表格代码的软件——找出了几处转录错误。尽管有多人多次仔细检查代码，数量惊人的小型转录错误仍在多轮审阅中留存了下来。[4]

转录后的代码被转换为 S表达式格式（.liplv 文件），适合由基于Lisp的解释器加载。方便的是，这些文件支持 Lisp 注释字符，因此可以直接在代码中记录更正和注释。

IV. 使用 Lisp 编写的新 IPL-V 解释器

我实现了一个新的 IPL-V 解释器，用于执行逻辑理论家的程序清单。该解释器实现了 IPL-V 手册中描述的符号数据结构和执行模型，并支持程序所需的基本操作。尽管逻辑理论家最早的版本运行于 RAND JOHNNIAC 上，但在此处采用的 Stefferud 程序清单发表于 1963年 时，IPL-V 已经在多种机器上实现。因此，此次重建是在与机器无关的IPL-V解释器中执行该程序，而不是尝试复现某一特定的 JOHNNIAC 硬件环境。完整的解释器和转录文本逻辑理论家代码可在与本文相关联的代码库中获取。

出于若干原因，我选择模拟 IPL-V抽象机，而非模拟原始硬件环境（ELIZA 在 IBM 7094 和 CTSS 上就是采用后一种做法 [9]）。首先，如上所述，我使用的代码是 IPL-V，因此并非 JOHNNIAC 专用代码。此外，并不存在完善的 JOHNNIAC模拟器，而且无论如何，我们都没有 JOHNNIAC 的 IPL 代码。确实有一款 IBM7094模拟器，其中包含一个在 IBSYS批处理操作系统的变体下运行的 IPL-V 解释器 [7], ，该模拟器还附带了 IPL-V 演示脚本。然而，这个 IPL-V 解释器是一个二进制程序，没有源代码，也没有文档。我尝试在这个平台上运行 LT，但程序在执行初期就失败了，且没有提供任何有用的故障原因信息。由于没有源代码，这次故障实际上无法调试。

然而，更根本地说，我着手这个项目的核心目标之一，不只是让 LT 运行起来，而是要阐明IPL-V——深入理解这一语言、机器模型以及那个时代的基础设施。编写我自己的解释器对实现这一目标至关重要。IPL-V 被定义为一种抽象机规范，而非针对特定计算机的程序；在20世纪60年代中期被 Lisp 取代之前，它曾在许多不同的机器上实现。[5]

我的解释器（iplv.lisp）实现了1964年手册所述的完整 IPL-V抽象机：单元表、控制单元 H0–H5、工作单元 W0– W9，以及压入机制。列表以单元链的形式存储，完全遵循规范。值得注意的是，压入（栈）机制本身也在 IPL-V 系统中通过链表实现。压栈和弹栈操作看起来像是抽象机器的硬件原语，但在内部，它们也建立在同一种列表机制之上，而 IPL-V 的其他一切——包括数据结构、例程和 IPL 程序本身——也都以此为基础；和 Lisp 一样，IPL-V 从头到尾都是列表。

我的解释器核心是一个名为 ipl-eval 的函数，它通过执行程序计数器（H1）中所列出的程序来获取并执行指令。我的 IPL 求值器支持可重入——这一能力在 Lisp 中可以轻易实现——从而支持 IPL-V 所需的递归调用。J函数以 Lisp函数的形式实现。虽然该解释器实现了完整的 IPL-V机器，但目前只实现了 LT 所需的 J函数子集（以及少数测试程序），即手册中定义的约 150 个函数中的几十个。

LT 尝试证明 23 个定理的完整运行记录可在 [18] 查看。这一过程大约需要 574,000 个 IPL 机器周期，并生成超过 576,000 个符号[6]。在一台普通的现代笔记本电脑上，它只需几秒钟即可完成。而在20世纪50年代中期的兰德公司 JOHNNIAC 上，这一过程很可能需要数小时。

[4]考虑到本文经过多轮审阅后仍然留存了这么多错别字，我大概本不该对此感到惊讶！

[5]此外，用 Lisp 本身实现 IPL-V——Lisp 的直接祖先——还带有一种令人愉悦的循环意味。[6]是的，我也很惊讶。这真的只是巧合！

V. 逻辑理论家证明了什么

逻辑理论家使用《数学原理》第2章中的五条公理，尝试证明命题逻辑中的定理。原作者使用的符号见表 II。

表 II 逻辑理论家使用的逻辑符号

符号	含义
I	蕴涵（→）
V	析取（∨）
*	合取（∧）
-	否定（¬）
=	双条件命题（↔）
.=.	定义相等

LT 运用多种证明方法：

- **代换**——代换公理或先前已证明定理中的变量，使其与目标相匹配。
- **分离**（肯定前件式）——由 $A \rightarrow B$ 和 A，推出 B。
- **正向链接**——从已知定理出发，向前应用代换和分离。
- **反向链式推理**——从目标出发向后推理，寻找能够推出该目标的前提。
- **子层级替换**——利用已知的等价关系替换子表达式。

每次证明尝试都受可设定的努力限额约束，以 IPL 机器周期为单位。（以下所有示例中的限额均设为 20,000 个周期。）这是一个软上限：超过限额后，系统不会开始新的子问题探索，但不会中断正在进行的搜索。

表 III 展示了本文所述运行开始时提供给 LT 的公理和定义。重要的是，LT 证明某个定理后，该定理便会被视为公理，并可用于支持后续证明。因此，从某种意义上说，LT 可以被称为会学习！

表 III 公理与定义

NS

Name	公理/定义
1.01	((PIQ).=.(-PVQ))
2.33	((PVQVR).=.((PVQ)VR))
3.01	((P*Q).=.-(-(PV-Q))
4.01	((P=Q).=.((PIQ)*(QIP)))
1.2	((AVA)IA)
1.3	(BI(AVB))
1.4	((AVB)I(BVA))
1.5	((AV(BVC))I(BV(AVC)))
1.6	((BIC)I((AVB)I(AVC)))

表IV展示了我重新复现的LT对尝试证明的23个定理所取得的结果。其中，16个已证明，7个未能证明——鉴于努力限额为20,000个IPL机器周期，这些结果与原始LT已知的表现从历史上看是一致的。

下面展示一个具有代表性的证明轨迹。在此，LT 证明了定理 2.01，(PI-P)I-P：

表 IV 尝试证明的定理

Name	公式	结果
2.01	(PI-P)I-P	已证明
2.02	QI(PIQ)	已证明
2.04	(PI(QIR))I(QI(PIR))	已证明
2.05	(QIR)I((PIQ)I(PIR))	已证明
2.06	(PIQ)I((QIR)I(PIR))	已证明
2.07	PI(PVP)	已证明
2.08	PIP	已证明
2.10	-PVP	已证明
2.11	PV-P	已证明
2.12	PI--P	已证明
2.13	PV---P	已证明
2.14	--PIP	未证明
2.15	(-PIQ)I(-QIP)	未证明
2.20	PI(PVQ)	已证明
2.21	-PI(PIQ)	未证明
2.24	PI(-PVQ)	已证明
3.13	(-(P*Q))I(-PV-Q)	未证明
3.14	(-PV-Q)I(-(P*Q))	未证明
3.24	-(P*-P)	未证明
4.13	P=--P	未证明
4.20	P=P	已证明
4.24	P=(P*P)	已证明
4.25	P=(PVP)	已证明

```
2.01 (PI-P)I-P

证明成立。

      已知                *1.2    (AVA)IA
      代换                .0      (PI-P)IP
      子层替换            2.01    (PI-P)I-P
      证毕

运算量          限制 20000          实际 5579
子问题限制 50                       实际 1
代换限制 50                         实际 2
```

证明从公理 *1.2，(AVA)IA 开始，将 -P 代入 A，得到 (-PV-P)I-P；然后应用子层级替换（利用 P 蕴含 PVP 这一事实）来变换前件，从而得到所需的定理。附录 1 展示了一个更复杂的证明 (4.25)，而包含这两个证明的原始输出位于： [18]。

VI. 调试：单独运算，然后进行人工智能协作

A. 单独工作阶段

这个项目绝大部分的开发和调试工作都是在没有大语言模型辅助的情况下完成的。构建解释器、转录代码、理解 IPL-V 手册，以及处理第一阶段的错误——崩溃、尚未实现的 J函数、栈下溢、寄存器保存/恢复错误——都是我独自完成的，历时数月。这些错误令人沮丧，但也能够解决：硬崩溃几乎总能追溯到崩溃前某处的错误，而独自工作的程序员也能发现并修复它们。

在这段时间里，我逐渐确信 IPL-V 解释器基本上是能正常工作的——否则 LT 根本不可能执行完哪怕十几条指令！而且，

LT 转录是正确的。因此，几乎所有剩余错误都很可能出在 J函数上；这些函数是我根据纽厄尔等人手册中的描述用 Lisp 编写的。我的经验是，手册对 J函数的描述有时会在细微之处含糊不清。我以前就吃过这个亏，所以对最复杂的 J函数格外谨慎：生成器（J17–J19、J100）和列表操作函数（J74 及许多其他函数）。

B. 高墙——以及大语言模型如何突破了它

与可以逐步验证正确性的传统程序不同，LT 是一种启发式搜索程序，其行为由许多相互作用的过程共同涌现。解释器稳定到足以让 LT 顺利运行、不再崩溃之后，错误的性质便彻底变了。此时，输出变成了数千行执行跟踪——机器周期、寄存器状态、列表操作——而这些内容可能是正确的。LT 能证明一些平凡的定理（只需进行代换的定理），但对于本应轻而易举证明的简单定理，它却无法证明；它会陷入失控递归，无法识别何时已经找到证明，还会表现出其他种种异常行为，只是不至于崩溃。这个程序采用启发式方法；它本就应该进行探索和回溯。要分辨一次回溯是正确的，还是由细微错误导致的，就必须同时理解 IPL-V 机器语义和 LT 的高层证明策略，并追踪数万次解释执行的机器周期中状态的变化。

这一经历与早期开发者的叙述产生了一种奇异的共鸣。在 Pamela McCorduck 为其开创性的人工智能史著作进行的一次访谈中，Cliff Shaw 谈到了调试早期人工智能系统的特殊困难。他指出，最令人望而生畏的排错阶段出现在开发后期：那时系统已经不会崩溃，却会产生表面上看似合理的行为，让开发者陷入两难，不知这些结果究竟是启发式逻辑的有效推论，还是实际错误。这带来了巨大的诊断难题：由于系统内存已变成一张不断翻涌、难以穿透的互连列表网络，代码修订的影响很难追踪。与结构性故障或崩溃不同，后者的错误通常离故障点很近，而这些后期出现的异常却与任何明显的源头脱节，几乎不可能分辨它究竟是预期的涌现特性，还是隐藏的缺陷。 [6]

六十年后，面对同一类程序，我遇到了完全相同的挑战。此时，进展实际上已经停滞。要在脑中记住机器状态，同时逐行检查数千行追踪记录，并不断判断每一行反映的是正确行为还是错误行为，这种认知负担已超出我的承受能力。

这时，我转而求助于大语言模型——具体来说，是谷歌的 Gemini（通过 AntiGravity 界面）和 Anthropic 的 Claude （通过其基于终端的编程界面 Claude Code ）。它们没有参与项目的早期阶段；直到调试任务的性质从崩溃分析转变为难度高得多的问题——调试不会导致程序终止的错误行为——时，它们才加入进来。

C. 向大语言模型教授一种已消亡的语言

IPL-V 本质上是一门已经消亡的语言：手册有些地方的光学字符识别效果很差，即使能够辨认，内容有时也晦涩难懂。这门语言本身与任何现代编程语言都截然不同，也没有 Stack Overflow、GitHub 语料库或从中学习的大语言模型从业者社区。如今，仍有直接使用 IPL-V 编程经验的人寥寥无几。大语言模型了解 IPL-V 的主要信息来源是我自己的 Lisp 代码——而其中当然仍然包含错误。

这带来了一个引人入胜的认识论引导问题。大语言模型根据我的解释器推断 IPL-V 语义，有时推断正确，有时则不然。当解释器存在错误时，大语言模型偶尔会学到错误的语义，并在建议中传播这些错误。要理清这一切，就必须时刻警惕：哪些是已经确立的知识，哪些是根据可能存在错误的代码推断出来的。使用大语言模型编程总要在一定程度上教会它项目相关的知识，但在这个案例中，我还必须教会它这门语言本身——这是性质截然不同的挑战。

早期的一个难点是，大语言模型有时会试图调试 LT 程序，而不是我的解释器。我们必须坚持这样一种方法论假设：LT 代码是正确的（尽管我们并不完全确定这一点），并将调试工作集中在解释器上，尤其是 J函数。若不遵守这一原则，每个异常都会引出许多调查方向，导致进展无从谈起。

D. J74 错误：*Simon's J's* 力挽狂澜

大语言模型最擅长的是坚持不懈。它们愿意逐行阅读数千行跟踪记录和转储，在数百处插入断点，并持续进行这项任务所要求的繁琐而精确的状态跟踪。这种全面细致的关注与领域理解相结合，讽刺的是，恰恰是当前大语言模型的强项。

经过几天断断续续、累计数小时的大语言模型运行时间，Gemini 发现了第一个重大问题。正如预料的那样，问题出在一个 J 函数中。这带来了实质性进展：此时 LT 已能生成证明，但找到证明后却不会停止。又经过了好几个小时的大语言模型运行时间，Claude 找出了 J74 中的第二个问题——列表复制函数。但我们不知道该如何修复，因为 Claude 和我都认为，我的代码似乎符合手册中的描述。问题显而易见，但规范看起来却支持我的实现。根据 Claude 的探索，我对问题所在提出了一个假设。

项目在此迎来了最有趣的转折。我们获得了计算机历史博物馆提供的"Simon's J's" [12]——一套来自1962年的 IPL-V 原始穿孔卡片。这些卡片记录了 J函数在 IPL-V 开始标准化、以便更广泛使用之际的实际生产实现。到这时，我已经教会 Claude 足够多的 IPL-V，使它能够读懂1962年的原始代码

并将其与我用 Lisp 编写的 J74 实现进行比较，以验证我对问题的假设。结果：它们的差异恰好印证了我的假设，而我只需在 J74 实现中打上一行补丁，LT 就能完全正确地运行！

这是一个非凡的时刻：一个人工智能系统主要通过一种现代解释器学习了一门已经消亡的编程语言；它曾协助调试该解释器，如今则能够读取 1962 年的原始源代码，并据此识别和修复解释器核心中的一个隐蔽错误。这是一场跨越六十年的协作——西蒙最初的卡片组由计算机历史博物馆保存，并由其志愿者转录，为我们提供了可靠的事实依据；无论是手册的文字说明、我对它的解读，还是大语言模型根据我的代码作出的推断，都未能提供这一依据。

E. 合作者的角色

参与此项目的人类合作者和大语言模型各自发挥了互补作用。人类贡献者——尤其是 Anthony Hay、Arthur Schwarz、Leigh Klotz 和 David M. Berry——负责转录代码、检测错误，并通过每周进展更新持续给予智力支持。Rupert Lane 的 Gridlock 软件发现了经过多轮人工审阅仍未被发现的转录错误。这些贡献至关重要，无法由大语言模型取代。

大语言模型发挥了人类合作者无法替代的作用：耗费数小时，逐行阅读数千行执行跟踪、插入诊断工具，并在数万个周期中追踪机器状态，以定位细微的错误。Gemini 和 Claude 都为调试作出了实质性贡献。最终，我主要选择与 Claude 合作，并非因为两者的能力有显著差异，而是因为保持一个持续累积上下文的长会话，比在不同系统间切换更容易。我与两个大语言模型的交互记录散乱地保存在项目的 GitHub 仓库中。

VII. 反思：IPL、LISP 与人工智能的发展历程

我用 Lisp 编程已经有五十年了。构建 IPL-V 解释器让我认识到，早在20世纪50年代末，纽厄尔、西蒙和肖就已经发明了使 Lisp 如此强大的几乎所有概念要素：列表操作、符号处理、程序本身就是列表这一思想、动态内存分配、递归、高阶函数、生成器，甚至运行时自修改的可能性（坦率地说，所有机器级语言都具备这一能力——而大多数高级语言都失去了它，Lisp 除外）。IPL 所缺少的是优雅的、基于 lambda 演算的同像语法——这种优雅地将代码与数据统一在单一记法中的方式，使得 Lisp 程序可以被 Lisp 程序操作。IPL-V 与 Lisp 共存了大约五年，但 Lisp 如此简单且富有表现力，很快就取代了它的前身。（当然，即使早期的 Lisp 也拥有自动垃圾回收，这同样是由麦卡锡发明的（！）这使编程变得可以忍受，尽管代价是性能方面的可预测性有所降低。）

这一认识还具有个人层面的意义，也关乎项目本身的性质。20世纪80年代，我曾在卡内基梅隆大学师从纽厄尔和西蒙，当时我的研究重点是建立科学家与工程师认知的模型。那时，IPL 和 LT 早已成为历史；遗憾的是，我从未与西蒙或纽厄尔讨论过它们——我们在课堂上几乎没有学到这些开创性的工作。但他们灌输给我的学术价值观——严谨看待机制、尊重认知的复杂性，以及坚信思维可以通过计算方式理解——塑造了我此后整个职业生涯。如今，在职业生涯的尾声，我发现自己正在将纽厄尔、西蒙和肖视为科学家兼工程师来研究：反向工程分析他们的设计决策，重构他们的推理，并试图理解他们如何发明了人工智能和认知科学。我受训研究的主题——科学家与工程师的认知——与研究对象就此融为一体。

VIII. 贡献与相关工作 K

本文作出了三项贡献：1）一个用 Common Lisp 实现的可运行 IPL-V 解释器。2）成功执行了来自 Stefferud 的 IPL-V 源代码清单的逻辑理论家，这可能是五十多年来首次执行该程序。3）展示了可执行重建作为研究早期人工智能系统的一种方法。

近年来，历史软件系统的复原工作日益兴起。与本文最直接相关的是莱恩等人对 ELIZA 的复原项目。 [9], 在该项目中，包括本文作者在内的团队让魏岑鲍姆最初的 ELIZA 在重建的 CTSS 上恢复运行，而该 CTSS 则运行于一台经过模拟的 IBM 7094 上。该项目重建了完整的原始硬件和操作系统栈；而本文的工作则是在现代语言中模拟抽象机规范。

用现代语言重新实现逻辑理论家的算法是一项相对直接的工作——本质上相当于一项人工智能入门课程的项目，而且肯定已有许多人做过。例如，一篇1968年论文描述了用 LISP 1 实现 LT 的方法。5 [14]本文的工作则是根本不同的尝试：它不是用现代语言重写 LT 的逻辑，而是构建一个忠实的 IPL-V 抽象机模拟器，并运行直接转录自一份1963年技术报告的原始代码。这一区别意义重大：用 Lisp 重新实现，证明了 LT 的算法可以用 Lisp 表达（这并不令人意外）；而本文的工作则证明，纽厄尔、西蒙和肖团队编写的原始 IPL-V 代码确实能如描述所言运行，而且还能让我们在真实运行的机器与程序系统上开展实验。

IX. 后续步骤

为本项目开发的 IPL-V 解释器并非专为逻辑理论家（Logic Theorist）设计。它实现了通用 IPL-

V 抽象机，原则上可以运行任何 IPL-V 程序。我正在构建一个早期 IPL-V 人工智能程序语料库。最重要的目标是通用问题求解器（GPS） [15] 和 EPAM（初级感知器与记忆器） [16]。EPAM 的源代码可在斯坦福的费根鲍姆档案中找到，而寻找 GPS 完整源代码的档案研究仍在进行中。如今已有经过验证的 IPL-V 解释器，运行这些程序所需的运算量应该会比最初构建解释器本身少得多。

解释器、转录后的 LT 代码、本文所述的完整输出，以及部分大语言模型交互历史，均已在 https://github.com/jeffshrager/ IPL-V 开源。

致谢

以下人员（排名不分先后）以各种方式参与了本项目，从代码转录到每周鼓励不等：Anthony Hay、Arthur Schwarz、Leigh Klotz、David M. Berry、Rupert Lane、Amy Majczyk、Emily Davis、Ethan Ableman 和 Paul McJones。我尤其感谢 Hay、Schwarz、Klotz 和 Berry；在整个项目期间，他们一直耐心听取每周的进展汇报，并给予鼓励，这些鼓励总是很有帮助，有时还包含实质性的技术见解。Rupert Lane 的 Gridlock 软件发现了几处在多轮人工审查中仍未被发现的转录错误。

我还要感谢计算机历史博物馆保存并提供“Simon’s J’s”，即最初的 1962 年 IPL-V J-函数卡片组；它在调试过程中发挥了不可或缺的作用。

最后，我想感谢艾伦·纽厄尔和赫伯特·西蒙。我在20世纪80年代于卡内基梅隆大学师从他们。那段训练的主题——科学家与工程师的认知——在这个项目中已与其研究对象密不可分。

本文借助了各种大语言模型完成，但所有概念和最终编辑决定均由作者独自作出。

参考文献

[1] A. Newell 和 H. A. Simon，《逻辑理论机——一个复杂信息处理系统》，兰德公司，美国加利福尼亚州圣莫尼卡，报告 P-868，1956年。 [2] A. Newell、J. C. Shaw 和 H. A. Simon，“逻辑理论机的经验探索：启发式方法案例研究”，载于西部联合计算机会议论文集，1957年，第218–230页。 [3] A. N. Whitehead 和 B. Russell，《数学原理》。英国剑桥：剑桥大学出版社，1910–1913年。 [4] A. Newell 等，《信息处理语言-V手册》，第2版。美国新泽西州恩格尔伍德克利夫斯：Prentice-Hall，1964年。 [5] J. McCarthy，“符号表达式的递归函数及其机器计算，第一部分”，《美国计算机协会通讯》，第3卷，第4期，第184–195页，1960年4月。[6] 卡内基梅隆大学档案馆，Pamela McCorduck 收藏 [7] R. Storey，“B7094” [在线]。网址：https://github.com/Bertoid1311/ B7094（访问日期：2026年3月11日）。 [8] E. Stefferud，“逻辑理论机：一个模型启发式程序”，兰德公司，美国加利福尼亚州圣莫尼卡，报告 RM-3731-CC，1963年。 [9] R. Lane、A. Hay、A. Schwarz、D. M. Berry 和 J. Shrager，“ELIZA 重获新生：让聊天机器人之母重返世界上最早的分时系统之一”，《*IEEE*计算机史年刊》，第47卷，第1期，第8–23页，2025年。 [10] F. Gruenberger，“JOHNNIAC 的历史”，兰德公司，美国加利福尼亚州圣莫尼卡，报告 RM-5654-PR，1968年。 [11] P. McCorduck，《思考的机器》，第2版。美国马萨诸塞州纳蒂克：A. K. Peters，2004年。 [12] A. Bochannek，《Simon’s J’s》，计算机历史博物馆博客，2012 [ ] 年9月。在线。网址：https://computerhistory.org/blog/simons-js/（访问日期：2026年3月11日）。[13] R. Lane，《Gridlock：从 PDF 文件中准确提取代码》，2024年。 [在线]。网址：https://github.com/rupertl/gridlock [14] J. S. Millstein，《LISP 中的逻辑理论家》，《计算机杂志》，第11卷第1期，第39–44页，1968年2月。[15] A. Newell、J. C. Shaw 和 H. A. Simon，《关于通用问题求解程序的报告》，载于《国际信息处理会议论文集》，1959年，第256–264页。[16] E. A. Feigenbaum，《言语学习行为的模拟》，载于《西部联合计算机会议论文集》，1961年，第121–132页。[17] H. B. Richman、J. J. Staszewski 和 H. A. Simon，《使用 EPAM IV 模拟专家记忆》，《心理学评论》，第102卷，330，1995年2，第305页。[18] J. Shrager，《复活的逻辑理论家程序的输出》， [在线]。网址：https://github.com/jeffshrager/IPL-V/blob/master/major results/20260309 CompleteAndCorrect.drb（访问日期：2026年3月11日）。

附录

为展示重构后的逻辑理论家生成的一项复杂证明，以下推导对应《数学原理》中的定理 *4.25：

$$P = (P \vee P)$$

这说明了析取幂等律。在 LT 表示法中，等价算子使用蕴涵和合取定义：

$$(P = Q) \equiv ((P \to Q) \wedge (Q \to P))$$

逻辑理论家的证明轨迹如下：

1) **展开定义　　等价关系：** LT 首先将目标化简为两个蕴涵的合取。

4.25 P=(PVP) .0 (PI(PVP))*((PVP)IP) 4.01 ,0 替换

逻辑形式：

$P = (P \vee P) \Rightarrow (P \to (P \vee P)) \wedge ((P \vee P) \to P)$

2) **确定第一个蕴含　　关系：** 系统确定了一条前向链推理路径，以建立合取的前半部分　　.

PI(PVP)　　4.20　，0 正向链接

逻辑形式：

$P \to (P \vee P)$

3) **通过代换寻找证明：** LT 利用内存中的定理 *2.20作为代换的依据。这不是一个已知公理，而是它此前已证明并记住、随后在此处应用的定理；请参见下方注释。

已知 2.20 人工智能(AVB) 代换.0

PI(PVP)

逻辑形式：

一般定理 $A \to (A \vee B)$ 被检索出来

代入 $A := P$和 $B := P$，得到 $P \to (P \vee P)$

4）**完成等价关系　　关系：** LT 通过将已证明的蕴涵与原始目标相连接，完成推理链　　.

已知 4.20 A=A 前向链式推理 4.25 P=(PVP) 证毕

0

逻辑形式：

由于 $P \to (P \vee P)$ 和 $(P \vee P) \to P$均已确立，因此等价关系 $P = (P \vee P)$ 成立。

性能指标：

运算量限制 20000 实际 8727 子问题限制 50 实际 2 代换限制 50 实际

3

成功执行更复杂的推导（例如定理 4.25），充分展现了逻辑理论家具备一种原始形式的机器学习能力。系统将先前已证明的定理（例如 2.20）视为后续搜索路径中已确立的“已知”条件，从而有效地扩充自身的知识库，体现了代数推理累积发展的特性。