Reconstructing Program Genealogies from Abstract Program Descriptions: The ELIZA Case Study

Jeff Shrager
Bennu Climate, Inc.
Palo Alto, CA 94301
jshrager@gmail.com

December 8, 2025

Abstract

Software phylogeny—the project of understanding how earlier programs influence later ones—is methodologically elusive because software lacks the material continuity that anchors biological or textual genealogies. Programs can be independently reconstructed from descriptions, translated across languages, simplified for different contexts, or critically reimagined, all while claiming membership in the same tradition. This paper develops a phylogenetic method for analyzing software relationships based on abstract architectural features rather than source code similarity, treating software artifacts as participants in textual traditions rather than as biological lineages.

Using ELIZA—Joseph Weizenbaum's 1966 conversational program—as a model system, I assembled a corpus of 27 implementations spanning six decades (1965-2025) and eight programming languages. Each implementation was characterized using 13 binary features encoding architectural properties such as pattern-matching mechanisms, memory structures, and script separation, extracted through LLM-assisted analysis and manual curation. The resulting character matrix was analyzed using PAUP* phylogenetic software to produce parsimony-based trees representing relationships among implementations.

The analysis included three validation specimens: a pair of byte-for-byte identical clones (to verify correct clustering of identical programs), a non-ELIZA pattern-matching program (to test whether the method distinguishes ELIZA-specific features from general pattern-matching architecture), and a critical art project that invokes ELIZA by name while rejecting its architecture entirely (to examine how critical engagement relates to the tradition). The clones clustered correctly with zero branch length. The non-ELIZA control nested within the ELIZA clade, revealing that the character set captures general string-processing architecture rather than

conversation-specific features. The critical intervention appeared at maximal phylogenetic distance, accurately representing engagement through rejection rather than implementation.

The resulting phylogeny reveals two major patterns: a basal polytomy of implementations preserving the complete ELIZA architectural suite across different languages and eras, and a derived clade showing simplifications driven by material constraints (microcomputer BASIC versions), pedagogical purposes (teaching implementations), and different social contexts (hobbyist communities, academic settings, web frameworks). These patterns support interpreting the phylogeny not as a literal genealogy but as a map of a textual tradition—showing how implementations relate through preservation, adaptation, convergence, and critique. The method demonstrates that phylogenetic analysis can reveal the structure of software traditions even when material descent is absent, obscure, or deliberately rejected.

1 Introduction

This project set out to ask a simple, methodological question: how can we reconstruct a genealogy of programs? Not the history as described in manuals or recollections, but a genealogy grounded in the artifacts themselves—the code, the structure, the linguistic patterns, and the operational behaviors that survive across generations of rewrites.

Software phylogeny – the project of trying to understand how earlier programs influence later, similar ones – is not a new concept, but it turns out to be elusive. Whereas organism have real (if often obscure) evolutionary pathways via inheritance or horizontal gene transfer, software artifacts evolve through many different pathways, including revision, partial direct copying, translation into different languages, and even de novo conceptual reconstruction. It may even be difficult to define what it means for a program to be the same as or similar enough to another program to be considered a clone or knock-off. Analysis of version control histories is (mostly) clear, but much too narrow to encompass many interesting cases of program inter-influence.

ELIZA, the early conversational program has generated an unusually large family of rewrites, ports, parodies, and reinterpretations. There are dozens and probably hundreds of ELIZA near-descendants scattered across dozens of programming languages, employing a variety of language-processing algorithms. The family of ELIZAs comprises a sprawling, poorly documented software "clade" that might be understood through systematic taxonomic analysis ELIZA provides an ideal proving ground for this. The family of ELIZA-like programs is large, diverse, inconsistent, and full of partial or fragmentary examples. Some versions survive in LISP or SLIP listings from the 1960s; others from 1980s hobbyist BASIC magazines; others as parodies, pedagogical rewrites, or one-off re-creations on long-dead university servers.

2 Method

2.1 Corpus Assembly and Normalization

Over almost two decades I have been assembling a corpus of historical and contemporary ELIZA implementations (`http://elizagen.org`) drawn from public archives, academic collections, personal code repositories, and recovered sources. Each program was treated as a *specimen* in a software–genealogical sense. To enable comparison across languages and coding styles, all specimens underwent a light normalization pass in which indentation and comment formats were standardized, superfluous whitespace was removed, and file metadata was stripped. These steps preserved the structural and logical features of each program while eliminating irrelevant surface variation.

2.2 Initial Feature Extraction and YAML Encoding

Each specimen was abstracted into a structured, analyzable form through a feature–extraction process driven by a large language model (LLM). For each program, the model was prompted to identify salient structural and behavioral properties, including control-flow patterns, pattern-matching mechanisms, rule formats, data structures, and characteristic idioms associated with ELIZA-style systems.

The resulting feature lists were stored in a structured `YAML` representation that served as an intermediate "phenotypic" description of each specimen. At this stage, we also performed *manual feature augmentation*, adding expert-recognized traits that the model sometimes omitted or generalized too broadly. These additions included historically known algorithmic markers, distinctive syntactic quirks, and subtle divergences from canonical ELIZA logic. The YAML files resulting from this phase constituted the initial character matrices for downstream inference.

2.3 Round-Robin Vocabulary Expansion

Because initial model outputs vary in the features they surface, we employed an iterative, round-robin expansion procedure to harmonize and enrich the shared feature vocabulary across the corpus. In each iteration, a specimen's current feature list was presented to the LLM together with the aggregate vocabulary drawn from all other specimens. The model was asked to identify potentially relevant but missing features, propose distinctions, and ensure that each specimen was evaluated in light of an increasingly comprehensive, cross-specimen vocabulary.

Following each automated expansion cycle, we again conducted *manual curation*. This included removing spurious or over-abstracted features and adding additional domain-specific distinctions that the model sometimes conflated. Iteration continued until the vocabulary stabilized and no significant new traits were produced.

2.4 Final Feature Matrix Construction

After the round-robin process converged, the final feature list for each specimen was assembled by merging all automatically inferred features with all manually augmented ones. Features were encoded either as binary or categorical characters, depending on their form, yielding a specimen-by-character matrix analogous to a morphological character matrix in biological systematics.

Pairwise dissimilarities between specimens were computed by treating each character as an independent trait and scoring mismatches in character state. The resulting distance or character-state matrix served as the input for phylogenetic analysis.

2.5 Phylogenetic Inference Using PAUP*

Phylogenetic trees were inferred using PAUP* under a parsimony criterion. The character matrix was imported into PAUP*, and heuristic tree searches were performed using random addition sequences, tree-bisection–reconnection (TBR) branch swapping, and multiple replicates to mitigate the risk of entrapment in local optima. All most-parsimonious trees were retained.

Strict and majority-rule consensus trees were then computed from the set of optimal trees. These consensus trees represent hypothesized genealogical relationships among ELIZA implementations, based solely on shared and divergent software features encoded in the final character matrix.

2.6 Interpretation

The resulting phylogenies should be interpreted as hypotheses of software relatedness rather than literal historical lineages. The pipeline—from normalized source programs, through LLM-derived and manually curated feature sets, and finally to PAUP*-based phylogenetic reconstruction—was designed to parallel biological systematics while accommodating the distinctive evolutionary dynamics of software artifacts.

What this project demonstrates is that software artifacts can be analyzed genealogically—not narratively or historically, but computationally. If one can extract a consistent set of characters across variants, one can apply the same tools used in evolutionary biology to visualize relationships among programs. ELIZA, because of its long history of transformation, provides an unusually rich test case, but the workflow generalizes. Any software ecosystem with many descendants or reinterpretations—adventure games, early operating systems, AI toolkits, interpreters—could be analyzed in this way.

The broader idea is that software, like biological organisms, carries traces of its ancestry. Those traces persist even when the syntax is rewritten, the structure reorganized, or the implementation style completely changed. And by approaching software genealogically—through careful artifact recovery, character extraction, and computational inference—we can uncover those traces in a principled way. The ELIZA Genealogy Project is still in progress, but the

Table 1: Program List

ID	**Program Name**
E01	Cosell66Lisp
E02	Trembly17Cobol
E03	SamsonXXLisp
E04	KMPXXLisp
E05	JonLXXLisp
E06	Babcock15BASIC
E07	Hay21CPP
E08	Juul23BASIC
E09	DavenportXXBASIC
E10	Srijak08Python
E11	Strout05Python
E12	Shrager73BASIC
E13	Norvig91Lisp
E14	Cherry82Lisp
E15	Bell79BASIC
E16	Dunlop97JS
E17	Platt79BASIC
E18	JonLXXLisp_clone
E19	Weizenbaum65MADSLIP
E20	Shrager73BASIC_clone
E21	Ciston15LadyMouthPython
E22	Shrager15FilerCPP_foil
E23	Jul99Java
E24	HaydenXXJava
E25	Haxpor15PHP
E26	Duguet70SNOBOL
E27	Fricks70B5500Algol

Table 2: Feature Definitions

rule_based_pattern_matching Indicates whether the program uses an explicit set of rules, each with patterns and associated responses, to determine its behavior, as opposed to purely procedural or statistical logic.
keyword_priority_ranking Indicates whether the program assigns numeric priorities to keywords or rules and uses these priorities to select which rule or phrase to apply.
synonym_translation_table Indicates whether the program maintains a mapping that normalizes different surface forms (e.g., synonyms, inflections) into canonical tokens before rule application.
backtracking_pattern_matcher Indicates whether the pattern matching engine can consume variable-length segments of the input and backtrack over them to satisfy complex patterns.
wildcard_and_class_tokens Indicates whether the pattern language supports wildcards (e.g., numeric segment placeholders) and token classes or property-based matches within patterns.
parse_segment_storage Indicates whether the matcher stores matched input segments in an auxiliary structure for later reference during response generation.
template_based_reconstruction Indicates whether responses are generated by filling templates that reference previously matched segments by position or index.
conversational_memory_rules Indicates whether the program stores selected past user inputs or derived phrases and later retrieves them via dedicated memory rules to generate responses.
fallback_last_resort_rule Indicates whether there is an explicit default or "last resort" rule that is applied when no higher-priority or more specific rule matches.
punctuation_sensitive_tokenization Indicates whether the input is tokenized using an explicit set of separator and breaker characters so that punctuation affects how tokens and phrases are formed.
ordered_memory Indicates whether the recall is ordered (True) or random (False) when memory is called upon to produce a response.
list&symbol_processing_v_string_processing A value of True for this feature indicates that the program primarily uses an list and symbol processing paradigm. A value of False indicates that the program uses a primarily string based representation.
separable_script A value of True for this feature indicates that the script is essentially separate from the language processing code, possibly (although not necessarily) in a separate file. A value of False indicates that the script is a built-in part of the program. (Just because the script appears in the given code doesn't necessarily mean that it is inseparable. If there is code that reads the script in at the beginning, even from its own file, the value here could be True.)

methodology is already clear: collect, normalize, extract, encode, and visualize. The goal is not to produce the definitive history of ELIZA, but to develop a method for computational taxonomy—an approach that brings rigor to the study of how software ideas evolve across time, languages, and cultures. ELIZA simply turns out to be the ideal proving ground for that experiment.

3 Results

Table 3: Program Features Comparison

Program	Rule-Based Pattern	Keyword Priority	Synonym Translation	Backtracking Pattern	Wildcard & Class	Parse Segment Storage	Template-Based Recon.	Conversational Memory	Fallback Last Resort	Punctuation Sensitive	Ordered Memory	List & Symbol vs String	Separable Script
E01_Cosell66Lisp	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
E02_Trembly17Cobol	✓		✓			✓	✓		✓	✓	✓		
E03_SamsonXXLisp	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
E04_KMPXXLisp	✓		✓			✓	✓	✓	✓	✓	✓	✓	
E05_JonLXXLisp	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
E06_Babcock15BASIC	✓	✓				✓	✓		✓	✓	✓		
E07_Hay21CPP	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
E08_Juul23BASIC	✓	✓	✓			✓	✓		✓		✓		
E09_DavenportXXBASIC	✓	✓	✓		✓	✓	✓		✓		✓		
E10_Srijak08Python	✓		✓	✓	✓	✓	✓	✓	✓		✓	✓	
E11_Strout05Python	✓		✓		✓	✓	✓		✓		✓		
E12_Shrager73BASIC	✓	✓	✓				✓		✓		✓		✓
E13_Norvig91Lisp	✓			✓	✓	✓	✓			✓	✓	✓	
E14_Cherry82Lisp	✓	✓				✓	✓		✓	?	✓	✓	
E15_Bell79BASIC	✓		✓			✓	✓		✓		✓		
E16_Dunlop97JS	✓					✓	✓		✓	✓	✓		
E17_Platt79BASIC	✓					✓	✓		✓				
E18_JonLXXLisp_clone	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
E19_Weizenbaum65MADSLIP	✓	✓	?	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
E20_Shrager73BASIC_clone	✓	✓	✓				✓		✓		✓		✓
E21_Ciston15LadyMouthPython													
E22_Shrager15FilerCPP_foil	✓			✓	✓	✓	✓						
E23_Jul99Java	✓		✓				✓		✓		✓		
E24_HaydenXXJava	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
E25_Haxpor15PHP	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓			✓
E26_Duguet70SNOBOL	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
E27_Fricks70B5500Algol	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

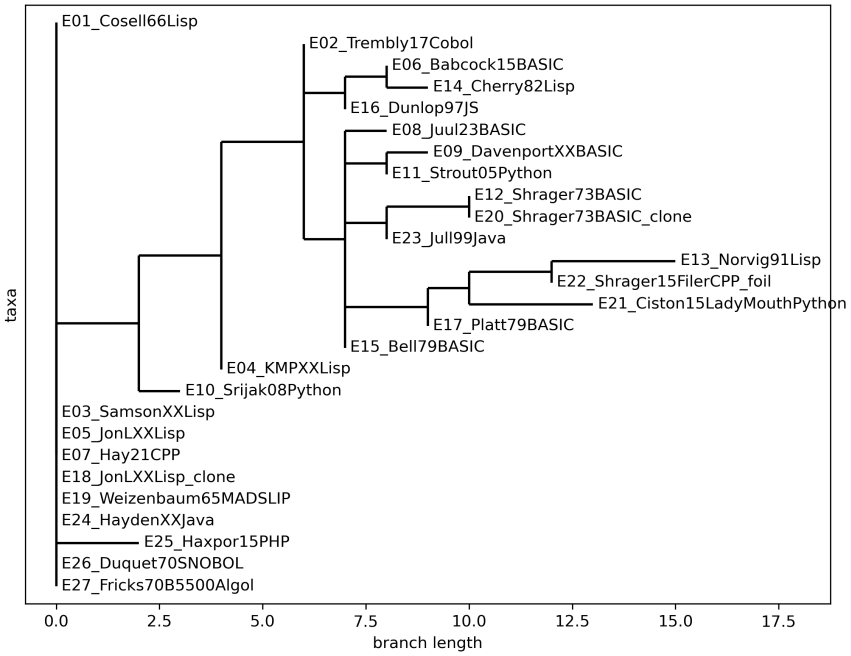

Figure 1: Python Bio:Phylo:draw rendition of this PAUP tree: (1:0, ((((2:0, ((6:0, 14:1):1, 16:0):1, (8:1, (9:1, 11:0):1, ((12:0, 20:0):2, 23:0):1, (((13:3, 22:0):2, 21:3):1, 17:0):2, 15:0):1):2, 4:0):2, 10:1):2, 3:0, 5:0, 7:0, 18:0, 19:0, 24:0, 25:2, 26:0, 27:0):0);. (Spaces added to permit line wrap; Numbering is as the E-numbers in the source codes.)

3.1 Phylogenetic Structure and Validation

The phylogenetic analysis (Figure 1) reveals a tree structure that reflects both the historical transmission of ELIZA implementations and the different modes of engagement with the ELIZA tradition discussed above. The tree exhibits two major structural features: a large basal polytomy containing implementations that share the complete ELIZA architectural suite, and a derived clade showing various degrees of architectural modification and simplification. Additionally, two validation specimens—one pair of clones and one non-ELIZA control—provide critical tests of the methodology's reliability and interpretive limits.

3.2 Validation Tests: Clones and Control

Two pairs of test specimens were included to validate the phylogenetic methodology. The first pair consists of exact copies: E12_Shrager73BASIC and E20_Shrager73BASIC_clone are byte-for-byte identical implementations differing only in filename. As ex-

pected, these cluster together as perfect sisters with zero branch length between them, confirming that the LLM-based feature extraction and phylogenetic reconstruction correctly identifies truly identical implementations. Their position within a larger clade of modified BASIC implementations reflects their shared architectural simplifications relative to the original ELIZA design.

The second pair tests whether the method can distinguish ELIZA implementations from non-ELIZA pattern-matching programs. E22_Shrager15FilerCPP_foil is a file system utility that performs pattern matching on filenames and directory structures but was never positioned as conversational software and makes no claim to be part of the ELIZA tradition. Despite sharing some architectural features with ELIZA implementations—rule-based pattern matching, wildcard tokens, backtracking capabilities—it nests within the derived ELIZA clade rather than placing as an outgroup. This placement reveals an important limitation: the thirteen binary characters used in this analysis capture general pattern-matching program architecture rather than ELIZA-specific conversational logic. A file utility and a chatbot can share sufficient architectural features to cluster together when only structural characters are considered.

This finding has two implications. First, it suggests that future refinements of the character set should include features more specific to conversational coherence and therapeutic dialogue patterns—features that would distinguish ELIZA-style chatbots from other pattern-matching systems. Second, and more fundamentally, it demonstrates that structural similarity alone does not constitute participation in the ELIZA tradition. The foil shares bibliographical codes (pattern-matching in C++, string processing paradigms) without sharing the social and intentional codes (naming, positioning, dialogue purpose) that define membership in the tradition.

3.3 The Basal Polytomy: Architectural Completeness

The most striking feature of the tree is the large basal polytomy containing eight implementations, all coded as having the complete ELIZA architectural suite (all characters scored as present): E01_Cosell66Lisp, E03_SamsonXXLisp, E05_JonLXXLisp, E07_Hay21CPP, E18_JonLXXLisp_clone, E24_HaydenXXJava, E26_Duguet70SNOBOL, and E27_Fricks70B5500Algol. Additionally, E19_Weizenbaum65MADSLIP—the original 1965 implementation—resides in or immediately adjacent to this polytomy, differing only by a single unknown character state (synonym translation table, which could not be definitively determined from available documentation).

This polytomy should not be interpreted as phylogenetic uncertainty or insufficient data. Rather, it accurately reflects the fact that these implementations are architecturally indistinguishable given the thirteen characters analyzed. They all preserve the complete ELIZA design: rule-based pattern matching with keyword priority ranking, synonym translation, backtracking pattern matchers, wildcard and class tokens, parse segment storage, template-based reconstruction, conversational memory with ordered recall, fallback rules, punctuation-sensitive tokenization, list-and-symbol processing paradigms, and

separable script architecture.

What makes this result particularly meaningful is that these implementations span dramatically different programming languages (Lisp, C++, Java, SNOBOL, Algol) and were created across nearly six decades (1965-2021). The fact that they cluster together demonstrates that programmers working in different languages and eras successfully preserved the complete architectural specification of the original ELIZA. This preservation occurred despite the absence of formal version control, despite translation across radically different programming paradigms, and despite the informal, distributed nature of ELIZA's transmission through academic networks, hobbyist communities, and historical preservation efforts.

The presence of both Cosell's 1966 reconstruction and the original MADSLIP version in this group is particularly significant. As discussed in the introduction, Cosell recreated ELIZA from Weizenbaum's verbal description without access to the original source code. The fact that his reconstruction is architecturally identical to the original—and clusters with other faithful implementations—validates both the robustness of Weizenbaum's conceptual design and the effectiveness of informal knowledge transmission in early computing communities. Similarly, Anthony Hay's 2021 C++ version (E07) clustering with the 1960s implementations confirms his claim that his reconstruction, based on careful study of historical documents and the surviving 1966 code fragments, successfully recovered the original architecture.

3.4 The Derived Clade: Architectural Modification and Simplification

Branching off from the basal polytomy is a well-supported monophyletic group showing substantially longer branch lengths, indicating significant architectural divergence from the complete ELIZA specification. This derived clade contains implementations from the 1970s onward, predominantly written in BASIC, Python, and JavaScript, along with several Lisp versions that simplified or modified the original architecture.

Within this clade, implementations show various patterns of feature loss or modification. Many BASIC implementations (E06_Babcock15BASIC, E08_Juul23BASIC, E09_DavenportXXBASIC, E15_Bell79BASIC, E17_Platt79BASIC) lack backtracking pattern matchers and conversational memory rules, likely reflecting the constraints of early microcomputer BASIC interpreters with limited memory and processing power. These versions often circulated through hobbyist magazines as type-in programs, where simplification was both a practical necessity and a pedagogical choice—making ELIZA understandable and implementable by amateur programmers working on home computers.

The Python implementations (E10_Srijak08Python, E11_Strout05Python) show different simplification patterns, sometimes preserving backtracking and memory features while abandoning keyword priority ranking or separable script architecture. These choices likely reflect modern programming paradigms where

different architectural patterns (object-oriented design, functional programming, web frameworks) shaped how ELIZA's concepts were reimplemented.

Peter Norvig's 1991 Lisp version (E13_Norvig91Lisp) occupies an interesting position in this clade. Despite being written in Lisp—the language of the original implementations—it lacks several features present in the basal polytomy versions: keyword priority ranking, synonym translation, fallback rules, and separable script architecture. This appears to be a deliberate pedagogical choice rather than a limitation: Norvig's implementation, published in "Paradigms of Artificial Intelligence Programming," was designed to illustrate core AI pattern-matching concepts in minimal, readable code rather than to faithfully reproduce the complete ELIZA architecture.

The clustering of implementations within this derived clade often reflects shared social contexts and material constraints rather than direct descent. BASIC implementations cluster together not necessarily because they copied from one another, but because they faced similar constraints (limited memory, string-handling limitations, lack of sophisticated data structures) and circulated through similar communities (hobbyist magazines, educational settings, early personal computing). This demonstrates how bibliographical codes and sociological contexts shape architectural features in parallel ways, producing convergent similarity without requiring direct transmission.

3.5 Critical Engagement: LadyMouth and the Boundaries of Tradition

The most extreme outlier in the analysis is E21_Ciston15LadyMouthPython, which scores zero on every architectural character. LadyMouth contains no rule-based pattern matching, no keyword priority ranking, no template-based reconstruction, no conversational memory—none of the structural features that define ELIZA implementations. Yet it explicitly positions itself as engaging with ELIZA through its title, its invocation of therapeutic dialogue, and its critical examination of gender, language, and computational authority in conversational AI.

LadyMouth represents what we might call *critical participation* in the ELIZA tradition—engagement through interrogation and rejection rather than preservation or adaptation. Where canonical ELIZA implementations accept the premise that therapeutic dialogue can be simulated through pattern matching and canned responses, LadyMouth questions that premise by constructing a system that generates responses through fundamentally different mechanisms (in this case, through poetic recombination and feminist linguistic intervention). Where traditional ELIZAs maintain the illusion of understanding through clever rule-based responses, LadyMouth deliberately breaks that illusion to expose the assumptions underlying computational approaches to conversation.

In the framework of textual traditions discussed above, LadyMouth's all-zero character state is not noise to be filtered out but meaningful data about modes of engagement. Its maximal phylogenetic distance from all other implementations accurately represents its relationship to the tradition: it participates

through critique rather than continuation. The naming paradox is fully visible here—LadyMouth claims the ELIZA name precisely *because* it rejects ELIZA's architecture, using that rejection to make a point about what ELIZA represents and what alternatives might exist.

This raises a broader methodological point about what phylogenetic analysis can reveal when applied to cultural artifacts like software. In biological phylogenetics, extreme divergence typically indicates either deep evolutionary time or radical environmental pressure driving morphological change. In software phylogenetics, extreme divergence can also indicate *intentional critique*—a deliberate choice to engage with a tradition by showing what it excludes or misses. The phylogenetic tree thus maps not only transmission and transformation but also critical distance and oppositional positioning.

LadyMouth's inclusion in this analysis demonstrates that the ELIZA tradition encompasses more than faithful implementations and practical adaptations. It also includes works that use ELIZA as a reference point for asking what conversational AI could or should be—programs that engage with ELIZA by proposing alternatives rather than variations. These critical interventions are part of the textual tradition in the sense that they respond to, reference, and reinterpret the original work, even when they share no structural features with it. The phylogenetic method, by assigning them maximal distance while still including them in the analysis, correctly represents their relationship to the tradition: connected through discourse and positioning, distant through architectural choice.

4 Prior Art

The study of software genealogy draws on several overlapping research traditions, each of which offers methods or conceptual frameworks that help illuminate the problem of reconstructing ancestry among dispersed, informally transmitted code artifacts—precisely the situation in which many historical ELIZA variants reside. Early work on clone detection recognized that duplicated or near-duplicated fragments could reveal latent relationships among programs. Baxter et al.'s AST-based clone detection system Baxter et al. (1998) showed that structural similarities in code, even when lexical tokens differ, can expose deep shared lineage. Kamiya et al.'s CCFinder Kamiya et al. (2002) extended this insight by demonstrating that multi-linguistic, token-based similarity analysis can robustly detect large-scale duplication in the wild. Roy and Cordy's survey Roy and Cordy (2007) codified the idea that code similarity is not merely a question of plagiarism or redundancy but can support inference about evolutionary relationships. For the present project, these works collectively underscore that structural and token-level patterns in ELIZA variants, even if superficially noisy or idiosyncratic, carry usable genealogical signals. This supports both the initial clustering of variants and the rationale behind using LLM-based vocabulary normalization to regularize surface diversity into comparable forms.

Studies of software evolution have long recognized that programs change

over time through copying, modification, and gradual accumulation of differences. Godfrey and Tu's analysis of open source software evolution Godfrey and Tu (2000) demonstrates that software systems exhibit branching patterns and divergence similar to biological lineages, though the mechanisms differ fundamentally. While this work focused on version control histories within single projects, it established that evolutionary patterns in software can be traced and analyzed systematically. The present work extends these observations by applying classical phylogenetic reconstruction methods to variants that lack formal version control metadata, using structural and behavioral features as evolutionary characters. These efforts demonstrate the feasibility of producing evolutionary trees of code artifacts when direct historical documentation is incomplete, providing a methodological foundation for treating the ELIZA corpus as an evolving population of software "organisms."

These efforts demonstrate the feasibility of producing evolutionary trees of code artifacts when direct historical documentation is incomplete. They provide a methodological and conceptual foundation for treating the ELIZA corpus as an evolving population of software "organisms," subject to informal copying, mutation, recombination, and drift across decades and platforms. While these studies mostly concern contemporary, large-scale ecosystems, the analogy holds strongly for ELIZA, whose variants proliferated through hobbyist networks, academic archives, bulletin boards, and early internet repositories in a manner closely resembling biological or cultural diffusion.

The third major tradition—malware lineage reconstruction—might seem distant at first glance, but it is methodologically the closest analog to the ELIZA problem. Malware families evolve through rapid, decentralized, and poorly documented copying, which makes reconstructing their ancestry using code similarity and structural deltas an essential forensic task. Walenstein et al.'s design-space analysis Walenstein et al. (2007) surveys the similarity metrics used to detect relatedness among polymorphic variants, while Bailey et al. Bailey et al. (2013) present one of the first practical systems for reconstructing malware family trees directly from code bodies. These papers demonstrate that coherent genealogies can be inferred even when artifacts are heavily mutated, distributed without provenance, and subject to opportunistic reuse—precisely the conditions under which ELIZA variants propagated. For the present project, malware lineage work provides both an empirical precedent and a methodological toolkit: the logic of identifying minimal mutations, shared scaffolding, inherited routines, and branching divergence is directly applicable to the ELIZA corpus, and validates the notion that lineage can be inferred even when artifacts are partial or degraded.

A fourth line of work emerges from mining software repositories, where researchers use version-control systems to reconstruct histories of files, functions, and subsystems. German and Hassan German and Hassan (2009) and Bird et al. Bird et al. (2006) demonstrate that even messy, inconsistent repository data contains enough signal to reveal complex evolutionary patterns. Although the ELIZA ecosystem lacks such structured metadata—its variants often circulated before version control was ubiquitous—the methodological stance is instructive:

software artifacts carry historical traces even when explicit documentation is incomplete or unreliable. Techniques such as semantic differencing, temporal clustering, and ancestry inference, originally developed for large VCS-backed projects, help frame the ELIZA effort as a special case of a broader problem in recovering the hidden structure of software evolution.

Finally, historical and cultural analyses of software, represented by work such as Marino's Critical Code Studies Marino (2020), emphasize that source code is also a cultural and historical object whose genealogy reflects social networks, educational lineages, and the circulation of ideas. This perspective is crucial for ELIZA, which was not merely copied but taught, rewritten, and reinterpreted across multiple communities—early AI laboratories, computer clubs, commercial microcomputers, hobbyist magazines, and eventually the open internet. Understanding ELIZA's genealogy therefore requires not only technical similarity metrics but also contextual reasoning about author intent, pedagogical transmission, and cultural reuse patterns. This is where the present project's hybrid approach—combining structural analysis, contextual metadata, code archaeology, and LLM-assisted normalization—extends beyond prior work.

Taken together, these bodies of literature demonstrate both the feasibility and the novelty of the current project. Clone detection research shows that even noisy lexical and structural patterns carry genealogical information. Software phylogenetics establishes that code artifacts can be treated as evolving populations. Malware lineage reconstruction proves that coherent trees can be inferred from informal, uncurated ecosystems. Mining-software-repository work demonstrates that lineage persists in artifacts even when metadata is sparse. And cultural code studies remind us that genealogies reflect not just code mechanics but the social pathways through which programs spread. The ELIZA genealogy project synthesizes these strands by building a reconstruction of a historically significant software family whose variants lack formal version control, whose mutation history spans half a century, and whose textual diversity requires normalization beyond conventional tools. In doing so, it occupies a unique position in the landscape of software evolution research, extending known methods to a domain where code, culture, and history are deeply entangled.

5 ELIZA as a Textual Tradition

5.1 The Limits of Biological and Philological Models

Although this work is inspired by biological phylogenetics and textual stemmatics, the artifact we are reconstructing is neither a phylogeny nor a classical stemma codicum. Those frameworks assume a set of conditions that simply do not hold for software lineages. Biological phylogenies presuppose predominantly bifurcating descent, relatively clocklike mutation, and the absence of extensive horizontal transfer. Textual stemmatics assumes manuscripts are copied with occasional errors, that variants propagate largely through single-parent descent, and that contamination—borrowing from multiple sources—is excep-

tional enough to be treated as an anomaly. In contrast, the evolution of software such as ELIZA is fundamentally polyparental and reticulated. Programs are translated, ported, rewritten, partially copied, and recombined across languages and platforms; distinctive fragments are borrowed and reinserted; and entire structures are periodically obliterated and rebuilt. These properties violate the assumptions required for a tree-shaped genealogy in either biology or philology.

The problem runs deeper than mere methodological mismatch. In biological taxonomy, identity is anchored in material continuity: an orange is identifiable as such not merely by its properties but by its causal history—it grew on an orange tree, which itself descended through an unbroken chain of reproduction from ancestral citrus lineages. This is what Kripke Kripke (1980) termed "rigid designation" in natural kinds: physical continuity of descent provides the anchor for reference. But software fundamentally lacks this anchor. A program can be recreated from scratch by someone who has never seen the original source code, guided only by a high-level description or conceptual understanding. Two independently written implementations of the same algorithm may be functionally identical yet share no material history whatsoever. Indeed, this is precisely how the first ELIZA variant—Bernie Cosell's 1966 Lisp implementation—came about: Weizenbaum described the algorithm in conversation, and Cosell reconstructed it independently without access to the original MAD-SLIP source Cosell (2023).

Moreover, programs are routinely copied, modified, compiled, decompiled, translated across languages, and reconstituted from fragments—each transformation potentially severing or obscuring the "chain of custody" that would establish lineage in the biological sense. For ELIZA specifically, the identity problem has an additional wrinkle: what counts as an ELIZA? Is ELIZA defined by Weizenbaum's specific pattern-matching algorithm, his use of decomposition rules, reassembly rules, keyword ranking, and memory stack? If so, then many programs commonly called "ELIZA" are not ELIZAs at all. ALICE, for example, uses a much more elaborate rule system (with AIML) and much larger scripts, yet Wallace explicitly positions ALICE as "ELIZA-like" in its reliance on pattern matching and canned responses Wallace (2009). Similarly, countless pre-LLM chatbots used keyword matching and template-based responses without implementing Weizenbaum's specific architecture; are they ELIZAs by virtue of shared conceptual approach, or distinct lineages that happened to converge on similar solutions?

Making matters worse is widespread confusion about what "ELIZA" even refers to. For Weizenbaum, ELIZA was designed as a language-processing framework capable of running different scripts for different conversational domains, with DOCTOR being merely one example script—albeit the most famous one. However, DOCTOR was so iconic that many implementations, including the present author's own 1973 BASIC version, conflated the two, hardcoding the psychiatric script directly into the program and thereby erasing the separation between platform and content that was central to Weizenbaum's original design. In such cases, is the program still an "ELIZA" if it has abandoned the

platform/script distinction?

One might fall back on self-identification: a program is an ELIZA if its creator says it is, typically by naming it ELIZA or DOCTOR. But this seems unsatisfying and appears to make phylogenetic analysis irrelevant, reducing "ELIZA-ness" to a matter of social convention rather than objective structure. However, as we argue below, this apparent failure of rigid designation is actually the key to understanding what ELIZA *is*—not as a species or a text, but as a tradition.

5.2 What the Phylogeny Represents

For these reasons, the phylogenetic trees presented in this study should not be interpreted as literal genealogies in the biological sense, nor as stemmata in the philological sense. They are better understood as maps of a textual tradition—visualizations of how implementations relate to one another through multiple, overlapping dimensions: algorithmic similarity, bibliographical codes, social contexts of production, and explicit claims of affiliation.

Where implementations cluster together with minimal branch lengths, this may indicate genuine transmission (direct copying or porting), convergent reimplementation under similar constraints, or shared social contexts that impose similar bibliographical codes. Where implementations diverge, this may reflect translation across programming paradigms, adaptation to different institutional settings, deliberate modification for pedagogical or artistic purposes, or critical engagement that retains the name while rejecting the architecture.

The character matrix underlying our analysis captures features at multiple levels. Some characters (rule-based pattern matching, keyword priority ranking, template-based reconstruction) encode core algorithmic properties that might be transmitted through direct ancestry or converge through similar design choices. Others (list-and-symbol processing vs. string processing, separable script architecture) capture bibliographical codes shaped by the affordances of particular programming languages and development environments. Still others (conversational memory rules, ordered vs. random memory recall) represent design decisions that vary even among implementations claiming high fidelity to Weizenbaum's original vision.

The resulting trees therefore reveal not a single, unambiguous history but a complex pattern of relationships within a textual tradition. They show how ELIZA exists not as a fixed entity but as a network of implementations, each bearing traces of what came before while introducing new variations shaped by material constraints, social contexts, and authorial intentions. This is not a failure of the phylogenetic method but rather an accurate reflection of how software ideas actually propagate—through teaching and learning, through informal knowledge transfer, through explicit citation and implicit influence, through faithful reproduction and critical intervention.

In presenting these phylogenies, we do not claim to have reconstructed *the* history of ELIZA but rather to have developed a method for computationally mapping the structure of a software tradition. The trees are hypotheses about patterns of relationship, grounded in observable features of the artifacts them-

selves, subject to revision as new specimens are discovered and new characters are defined. What they offer is a principled, replicable approach to understanding how software lineages form, persist, and transform across time, languages, and communities—an approach that respects both the material reality of code and the social reality of naming, claiming, and belonging.

5.3 Software as Textual Tradition

However, we can reframe the identity problem by drawing on approaches from textual criticism and the sociology of texts, particularly the work of Jerome McGann and D. F. McKenzie. Rather than treating ELIZA implementations as more or less accurate copies of an "original," we can view each as a distinct textual version within a complex transmission history McGann (1991); McKenzie (1999). The original MAD-SLIP version is one version; the various LISP implementations constitute another set of versions; contemporary JavaScript or Python versions represent yet other lines of descent. Like texts in a manuscript tradition, each implementation bears traces of its genealogy while also introducing variations, whether through deliberate modification or through the constraints and possibilities of its particular social and material milieu.

McGann's notion of "bibliographical codes" McGann (1991) extends naturally to software. In literary scholarship, bibliographical codes encompass the physical and material aspects of texts—typefaces, formatting, page layout, binding—that carry meaning beyond the linguistic content. For software, these codes might include the programming language itself, but also comments and documentation, the formatting and structure of the code, the names given to variables and functions, and the overall architecture of the program. Two implementations of ELIZA might embody a similar algorithm yet differ significantly in their bibliographical codes, and these differences matter profoundly for how programmers understand, modify, and maintain the software. A 1960s LISP version with terse variable names and minimal comments reflects the material constraints of its time—scarce memory, expensive computing resources, programmers working in isolation. A modern Python version with extensive docstrings, type hints, and modular architecture reflects contemporary norms of collaborative development, version control, and software engineering pedagogy. These are not merely superficial differences but constitutive features of each version's identity.

McKenzie's "sociology of texts" McKenzie (1999) directs attention to the institutional and social contexts in which software is produced and circulated. MIT's Project MAC time-sharing system was crucial for shaping what could be implemented in the original ELIZA, how it would be documented, and what purposes it would serve. Later versions emerged from radically different contexts: university computer science departments teaching AI concepts, hobby computing magazines distributing type-in programs, commercial software development, and eventually open-source communities sharing code through GitHub and Stack Overflow. Just as literary texts undergo transformation as they are copied, edited, and republished, software mutates as it is ported, adapted, and

reimplemented across these different social contexts. Authenticity, in this framework, cannot be located in an "original"—indeed, few people besides Weizenbaum ever saw the original MAD-SLIP source—but rather in the integrity of each version within its own context. Each implementation exists as part of a tradition, bearing relations to what came before and enabling what comes after.

Under this view, the identity of ELIZA resides not in any single canonical implementation but in the entire network of implementations, documentations, discussions, and uses that constitute its textual condition. This shifts the question from "what is the true ELIZA?" to "how do implementations position themselves within the ELIZA tradition?" The phylogenetic analysis then becomes a method for mapping the structure of this tradition—revealing not biological descent but patterns of transmission, transformation, and engagement across different material and social contexts.

5.4 The Naming Paradox: Intentional Adoption vs. Rigid Designation

This textual framework resolves what initially appears as a fatal flaw in phylogenetic analysis of software: the problem of naming and self-identification. In Kripke's account of rigid designation, names like "water" refer to H_2O regardless of what anyone believes about water, because physical continuity anchors the reference. But software inverts this logic. A program can be named "ELIZA" precisely *because* there is no material continuity requirement. The act of naming becomes a claim of membership in the textual tradition, even when the bibliographical codes have been radically transformed or explicitly rejected.

Consider two programs in our corpus that illustrate this paradox. The first is a file-processing utility (E22_Shrager15FilerCPP) included as a control specimen—a program that performs pattern matching on filenames but makes no claim whatsoever to be part of the ELIZA tradition. It was never called ELIZA, never positioned as conversational software, and emerged from an entirely different lineage of systems programming. Yet when coded for architectural features, it shares several characteristics with ELIZA implementations: rule-based pattern matching, wildcard tokens, backtracking capabilities. These shared features arise not from descent but from convergent solutions to similar computational problems.

The second is LadyMouth (E21_Ciston15LadyMouthPython), a critical art project by Sarah Ciston that explicitly engages with ELIZA by name and concept, positioning itself as a feminist computational intervention that interrogates ELIZA's assumptions about language, gender, and therapeutic authority Ciston (2019). When coded for the same architectural features used across our corpus, LadyMouth scores zero on every character—no keyword ranking, no template-based reconstruction, no conversational memory, no fallback rules. Architecturally, it shares nothing with canonical ELIZA implementations. Yet it undeniably participates in the ELIZA tradition through critical engagement and explicit reference.

The naming paradox is this: self-identification through naming is actually the *opposite* of rigid designation. It represents intentional adoption into a tradition rather than material continuity. LadyMouth says "I am ELIZA" not because of descent but because of critical engagement with the ELIZA concept. The file utility shares architectural features but makes no such claim. In the textual tradition framework, both positions are meaningful data rather than noise. When someone names their program ELIZA, they perform an act of acknowledgment (recognizing the tradition exists), positioning (claiming a relationship to it), and interpretation (offering their understanding of what ELIZA means).

This means that phylogenetic analysis of software does not become irrelevant when identity is socially constructed—rather, it reveals the *structure* of that social construction. The tree does not map biological descent but instead maps different modes of participation in a textual tradition. Characters in our matrix capture both algorithmic features (which may reflect genuine transmission or convergent evolution) and bibliographical codes (which reflect material and social contexts of production). The resulting phylogeny shows how implementations cluster according to their modes of engagement: faithful preservation and recreation, adaptation through different contexts, convergent reimplementation, or critical intervention through rejection.

In this light, our phylogenetic analysis is not undermined by the lack of rigid designation but is instead precisely suited to studying traditions constituted through naming, reference, and intentional positioning. The ELIZA corpus functions as a textual community—a set of artifacts united not by material descent but by shared reference to a founding concept and ongoing dialogue about what that concept means.

Code Availability

All of the python programs, and ELIZA sources used in these experiments are available here: https://github.com/jeffshrager/elizagen.org/tree/master/genealogy

Acknowledgments

Thanks to the members of *Team ELIZA* (David Berry, Sarah Ciston, Anthony Hay, Rupert Lane, Mark Marino, Peter Millican, Art Schwarz, and Peggy Weil) for years of useful discussion about ELIZA, and especially Anthony Hay and Art Schwarz, for detailed technical advice on this particular project. Rupert Lane obtaind the codes for the SNOBOL and ALGOL ELIZAs. Some of this work, including some of the programming and some of the writing of this paper, were aided by OpenAI and Anthropic LLMs.

References

Bailey, M., Dumitras, T., and et al. (2013). A system for malware lineage. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 73–89.

Baxter, I. D., Yahin, A., Moura, L., Sant'Anna, M., and Bier, L. (1998). Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance*, pages 368–377. IEEE.

Bird, C., Zimmermann, T., and Nagappan, N. (2006). Mining windows for software engineering tools. In *IEEE Software*.

Ciston, S. (2019). Ladymouth: A feminist computational intervention. Digital Art Project. Available at: https://sarahciston.com/ladymouth.

Cosell, B. (1966–2023). Personal communication regarding 1966 eliza reconstruction. Multiple conversations and email exchanges regarding the development of the first Lisp ELIZA implementation.

German, D. M. and Hassan, A. E. (2009). Studying software evolution using cvs, subversion, and git. *Empirical Software Engineering*, 14(2):127–168.

Godfrey, M. W. and Tu, Q. (2000). Evolution in open source software: A case study. In *Proceedings of the International Conference on Software Maintenance*, pages 131–142. IEEE.

Kamiya, T., Kusumoto, S., and Inoue, K. (2002). Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. In *IEEE Transactions on Software Engineering*, volume 28, pages 654–670.

Kripke, S. A. (1980). *Naming and Necessity*. Harvard University Press, Cambridge, MA.

Marino, M. C. (2020). *Critical Code Studies*. MIT Press.

McGann, J. J. (1991). *The Textual Condition*. Princeton University Press, Princeton, NJ.

McKenzie, D. F. (1999). *Bibliography and the Sociology of Texts*. Cambridge University Press, Cambridge, UK.

Roy, C. K. and Cordy, J. (2007). A survey on software clone detection research. *Queen's School of Computing Technical Report*.

Walenstein, A., Mathur, A., Chouchane, R., and Lakhotia, A. (2007). The design space of code similarity algorithms. In *Proceedings of the 2nd ACM Workshop on Recurring Malcode*, pages 1–10.

Wallace, R. S. (2009). The anatomy of a.l.i.c.e. In Epstein, R., Roberts, G., and Beber, G., editors, *Parsing the Turing Test: Philosophical and Methodological Issues in the Quest for the Thinking Computer*, pages 181–210. Springer.