A Literate Programming Environment for Human and Machine Agents

ADAM T. BURKE, Queensland University of Technology, Australia

This paper introduces an environment for constructing literate programs in concert with language-aware machine agents. This environment includes a grammar for executable program essays, a parser that treats names as first-class objects, an internal name-graph which relates prose, names and executable artifacts, and a binding mechanism for existing languages and testing toolsets. This supports co-location of code with its most relevant natural language and structured data context, making better use of Large Language Model (LLM) context windows. It also provides LLM coding agents with a toolset more analogous to the symbol-aware search and usage information available in human programmer-facing Integrated Development Environments (IDEs). We describe a working implementation with bindings to three established programming languages, and several example programs.

Additional Key Words and Phrases: programming language, literate programming, Large Language Models, agent programming

1 Introduction

We are in a technological moment when the use of Large Language Models (LLMs) [30] has triggered major changes in programming practice. This ranges from the authoring of small scripts by programming novices, to analysis of codebases for security flaws, to the construction and extension of large software systems by experienced teams of software engineers.

As language-aware AI coding tools are changing programming, our programming languages and environments should change as well. This paper introduces a new programming language and environment informed by three observations. Firstly, and unsurprisingly, natural language text dominates the specification and design organization of programs developed with language-aware AI tooling. Secondly, machine checkable computational artifacts, which constrain the operation and execution of programs, retain or increase their value by acting as an automated feedback surface for coding agents. Thirdly, the influence of the context window is dominant in the current generation of language-aware AI.

In the transformer architecture, a very large language corpus is used to train a model, then interrogated at inference time by next token prediction. No change is made to the model during basic inference, for example, a session where a programmer works with a coding agent. The model itself does not learn. Any memory or continuity in such a session comes from the re-injection of symbols into the context window from chat history or other text artifacts.

Peter Naur has an influential argument that programming is theory building [6, 22]. Crucially, Naur argues that the theory being built lives primarily in the programmer's head, not in program artifacts such as source code. From this point of view, when using coding agents, there is no way for the model itself to learn this theory, so the responsibility for building and carrying the theory remains entirely with the programmer.

As Dominic Fox points out [11], Donald Knuth's literate programming [18] is a competing school of thought. It holds that a theory of a program can not only be conveyed in written artifacts, but that well-composed natural language is a highly effective vehicle for doing so. Knuth accordingly created tools and a style of writing code and essay-style prose together, an idea that has influenced much scholarship and language tooling since, without really centring software development on prose as Knuth envisioned. Recent research explores the sympathies between language-competent agents and literate programming [28, 31], even declaring it a literate programming renaissance [31].

"The hottest new programming language is English", according to Andrej Karpathy in 2023 [15], though this observation was as much fashion journalism as design intent. With this "literate turn" in programming history, we

Author's Contact Information: Adam T. Burke, at.burke@qut.edu.au, Queensland University of Technology, Brisbane, Queensland, Australia.

```
#Fibonacci

The Fibonacci sequence: each number is the sum of the two before
it, starting from 0 and 1.

    fib :: Int -> Int
    fib 0 = 0
    fib 1 = 1
    fib n = fib (n - 1) + fib (n - 2)

The defining recurrence, checked here as a property rather than
assumed for a handful of examples.

~property
    prop_recurrence :: Int -> Bool
    prop_recurrence n =
        let k = abs n `mod` 20 + 2
        in fib k == fib (k - 1) + fib (k - 2)

_
#Tests

##base cases
    fib 0 == 0
    fib 1 == 1

##known values
    fib 10 == 55
```

Fig. 1. A notlob program which calculates the Fibonacci sequence, using the Haskell binding.

should reach for that genre of writing which is used to explore and advance an idea: the essay. Like Knuth, we should do so without abandoning the productive mechanics of programming language design. Unlike Knuth, both humans and computers will be readers of the essays and the executable code.

This paper presents a literate programming environment for human and machine agents. This includes:

- a machine interpretable language, comprising both prose and executable elements;
- embedded code fragments, using existing programming languages, and their rich libraries;
- an internal data structure relating both concept names and executable symbols, called the name-graph;
- tools which expose name-graph relationships, providing IDE-like support for agents and other tools; and
- example programs.

The three observations motivate this design. The dominance of natural language text leads to a language with both prose and executable elements. The usefulness of machine checkable artifacts leads to keywords for property and unit tests. The importance of the context window re-motivates literate programming's co-location of prose, code, and validation in files, and navigation using the name-graph. The programming language and environment is named notlob, after a section in Monty Python's parrot sketch where a dissimulating shopkeeper makes a false but easily checkable claim about palindromes and the town of Bolton.

This paper focuses on design and example programs of the language and tooling environment; experimental evaluation is left to future work. The remainder of this paper is organised as follows. Section 2 introduces the notlob grammar, semantics and name-graph. Section 3 introduces a reference implementation of the language and environment. Section 4 shares example programs and projects. Section 5 reviews related work, and Section 6 concludes.

2 Design

The notlob language consists of structured prose with embedded executable blocks, demarcated by headings. The macro structure of a file is modelled on a technical report. The source is divided into a main body and supplementary text. The body is intended for those functions and data structures used at runtime, including the underlying concepts. The supplemental text, after a text literal evoking a horizontal break, is for tests, appendices, and references, including code and heading imports. Where some approaches for programming with LLMs use specifications placed upstream of a pipeline which produces code, notlob co-locates prose, code, and checks in a single source artifact.

```
⟨start⟩           ::= ⟨module⟩
⟨module⟩          ::= ⟨MOD_HEAD⟩ ⟨body⟩ [ ⟨post_text⟩ ]
⟨body⟩            ::= { ⟨body_item⟩ }
⟨body_item⟩       ::= ⟨subheading⟩
                    | ⟨code_block⟩
                    | ⟨claim⟩
                    | ⟨prose_block⟩
                    | ⟨bullet_block⟩
                    | ⟨BLANK⟩
⟨subheading⟩      ::= ⟨SUBHEAD⟩ { ⟨_sub_item⟩ }
⟨_sub_item⟩       ::= ⟨code_block⟩
                    | ⟨claim⟩
                    | ⟨prose_block⟩
                    | ⟨bullet_block⟩
                    | ⟨BLANK⟩
⟨code_block⟩      ::= ⟨INDENTED_LINE⟩ { ⟨_body_line⟩ }
⟨claim⟩           ::= ⟨SIGIL⟩ ⟨_body_line⟩+
⟨prose_block⟩     ::= ⟨prose_line⟩+
⟨prose_line⟩      ::= (⟨LINE_START_TEXT⟩ | ⟨PROSE_TEXT⟩ | ⟨REF⟩)+ ⟨BLANK⟩
⟨bullet_block⟩    ::= ⟨BULLET⟩+
⟨_body_line⟩      ::= ⟨INDENTED_LINE⟩
                    | ⟨BLANK⟩
⟨post_text⟩       ::= ⟨SEPARATOR⟩ { (⟨BLANK⟩ | ⟨post_section⟩) }
⟨post_section⟩    ::= ⟨tests_section⟩
                    | ⟨binding_section⟩
                    | ⟨references_section⟩
                    | ⟨appendix_section⟩
⟨tests_section⟩   ::= ⟨TESTS_HEAD⟩ { (⟨BLANK⟩ | ⟨test_item⟩ | ⟨prose_block⟩) }
⟨test_item⟩       ::= ⟨test_group⟩
                    | ⟨INDENTED_LINE⟩
⟨test_group⟩      ::= ⟨SUBHEAD⟩ { (⟨INDENTED_LINE⟩ | ⟨BLANK⟩ | ⟨prose_block⟩ | ⟨named_test⟩) }
⟨named_test⟩      ::= ⟨TEST_SIGIL⟩ ⟨_body_line⟩+
⟨binding_section⟩ ::= ⟨BINDING_HEAD⟩ { ((⟨INDENT⟩ ⟨bind_detail_decl⟩) | ⟨BLANK⟩) }
⟨bind_detail_decl⟩ ::= ⟨LANGUAGE_DECL⟩
                    | ⟨EXTERNAL_DECL⟩
                    | ⟨ON_BUILD_DECL⟩
                    | ⟨KEEP_SRC_DECL⟩
⟨references_section⟩ ::= ⟨REFERENCES_HEAD⟩ { (⟨INDENTED_LINE⟩ | ⟨BLANK⟩) }
⟨appendix_section⟩ ::= ⟨APPENDIX_HEAD⟩ { ⟨body_item⟩ }
```

Fig. 2. Notlob grammar - Productions.

A small example notlob program for calculating the Fibonacci sequence is in Figure 1. This main section demonstrates the heading, introductory text, and function declaration. A property declaration indicates an executable property test. After the break characters, unit tests give statements of Fibonacci facts, similar to an appendix of historical chronology or other data tables. This example uses a Haskell binding for the executable layer.

2.1 Language Syntax

In general the gross structure supports putting the explanation, execution and supporting checks together in a single file. The necessary context for human and machine agents can often be found immediately adjacent in the same file read.

```
⟨LINE_CHAR⟩          ::=  any character other than newline
⟨REST_OF_LINE⟩       ::=  { ⟨LINE_CHAR⟩ } ⟨NewLine⟩
⟨MOD_HEAD⟩           ::=  # ⟨NonHashLineChar⟩ ⟨REST_OF_LINE⟩
⟨SUBHEAD⟩            ::=  ## ⟨REST_OF_LINE⟩
⟨SIGIL⟩              ::=  ~example ⟨NewLine⟩
                          | ~run ⟨NewLine⟩
                          | ~run on-load ⟨NewLine⟩
                          | ~run on-invocation ⟨NewLine⟩
                          | ~property ⟨NewLine⟩
                          | ~property  ⟨REST_OF_LINE⟩
⟨TEST_SIGIL⟩         ::=  ~test  ⟨LINE_CHAR⟩ ⟨REST_OF_LINE⟩
⟨LANGUAGE_DECL⟩      ::=  ~language  ⟨LINE_CHAR⟩ ⟨REST_OF_LINE⟩
⟨EXTERNAL_DECL⟩      ::=  ~external  ⟨LINE_CHAR⟩ ⟨REST_OF_LINE⟩
⟨ON_BUILD_DECL⟩      ::=  ~on-build  ⟨LINE_CHAR⟩ ⟨REST_OF_LINE⟩
⟨KEEP_SRC_DECL⟩      ::=  ~keep-generated-src ⟨REST_OF_LINE⟩
⟨SEPARATOR⟩          ::=  --- ⟨NewLine⟩
⟨TESTS_HEAD⟩         ::=  #Tests ⟨NewLine⟩
⟨BINDING_HEAD⟩       ::=  #Binding ⟨NewLine⟩
⟨REFERENCES_HEAD⟩    ::=  #References ⟨NewLine⟩
⟨APPENDIX_HEAD⟩      ::=  #Appendix ⟨REST_OF_LINE⟩
⟨INDENT⟩             ::=  (⟨Space⟩ | ⟨Tab⟩)+
⟨INDENTED_LINE⟩      ::=  ⟨INDENT⟩ ⟨REST_OF_LINE⟩
⟨BLANK⟩              ::=  ⟨NewLine⟩
⟨BULLET⟩             ::=  * (⟨NewLine⟩ | ((⟨Space⟩ | ⟨Tab⟩) ⟨REST_OF_LINE⟩))
⟨REF⟩                ::=  (# | ##) ⟨UpperLetter⟩ { ⟨WordChar⟩ } { (⟨Space⟩ ⟨UpperLetter⟩ { ⟨WordChar⟩ }) }
⟨LINE_START_TEXT⟩    ::=  ⟨ProseInitial⟩ { ⟨ProseTail⟩ }
⟨PROSE_TEXT⟩         ::=  ⟨ProseTail⟩+
⟨NonHashLineChar⟩    ::=  ⟨LINE_CHAR⟩ − #
⟨UpperLetter⟩        ::=  an uppercase letter
⟨WordChar⟩           ::=  a Unicode letter, digit, or underscore
⟨Space⟩              ::=  a space character
⟨Tab⟩                ::=  the tab character
⟨NewLine⟩            ::=  the newline character
```

Fig. 3. Notlob grammar - Terminals.

An EBNF grammar for notlob is in Figures 2, for productions, and 3, for terminals. Title and section headers use a Markdown-style # prefix [12]. Inline references to headings, and so contexts, uses the same # prefix. Some headings have structural meaning and are reserved words: *#Binding*, *#Tests*, *#Appendix*, and *#References*. Executable blocks are indicated by indentation, with the internal syntax of those blocks determined by the language binding. Types of executable blocks are distinguished using `~`-prefixed keywords called sigils. Some sigils can also carry a label, as in `~test mcmxciv`. A group of sigils declare and configure language binding: `~language`, `~external`, `~on-build`, and `~keep-generated-src`. Another group of sigils are used for tests: `~property`, `~example`, and `~test`. `~run` indicates an execution entry point. The grammar restricts certain sigils to certain gross sections of the file, such as `~test` in the *#Tests* section.

The lexer uses an ambiguous regular grammar, disambiguated first by rule priority, then maximal munch for specific token classes. This allows prose fallback to be lowest priority. Rule priorities are listed in Table 1.

2.2 Semantics and Names

The detail of execution is delegated to other programming languages, tools, and libraries, defined by a structure called a binding. The semantics of the notlob language itself are then about firstly, defining hooks into that binding, and secondly, describing the relationships between names, prose and executable code.

2.2.1 Bindings. A binding defines a programming language, unit test library, property test library, and linter. For example, the Haskell binding uses stack-ghc, hunit, quickcheck, and hlint. This defines the context into which code blocks are assembled, with `~property` blocks becoming property tests, but `~example` and `~test` blocks becoming unit tests. All code blocks within the *#Tests* section are interpreted as unit tests, under the scope of either the `~test` label, or the most recent heading. Two separate sigils are provided for tests. Examples declared with `~example`, like Python doctests [25], are considered part of the exposition of concepts, and restricted to the main body. Unit tests are considered supplementary facts, and restricted to the supplemental material. A declared property is treated as a formal invariant, and expected in the main section near prose explaining the concepts and motivation for the constraint.

As an implementation convention, the file `binding.lob` declares the binding for a project, and does not need to be imported by other source files.

2.2.2 Traversing Program Semantics. Names are first class objects in notlob. Titles and headings are part of the syntax, and available in the parse tree. Name and concept maintenance are important infrastructural burdens in large software systems. We consider names load bearing structures for holding theory with source code artifacts across people, machines, and time. We are inspired in this by craft observations on the importance of names, such as Michael Feathers' observation that "Rename Class is the most powerful refactoring" [9, 10], as well as broad projects of making conceptual structure legible, such as the semantic web [13, 20].

As commentary is not discarded by the compiler, it is straightforward to use the parse tree to build a graph of names, prose, concepts and code that can be accessed by tooling. There are then two types of adjacency available cheaply to human and machine agents: file-adjacency, where elements are close together in the same chunk of serial text on the filesystem, and concept-adjacency, where related elements are connected on a path with a small number of edges. The name-graph can be explored by human and machine programmers at build time with accompanying tooling. It can also be used for consistency checks across the codebase, which includes the prose. The name-graph is an analogue to Knuth's generated index [18], and many documentation tools since, but with a non-linear data structure.

Priority	Terminals
20	*SEPARATOR*, *TESTS_HEAD*, *BINDING_HEAD*, *REFERENCES_HEAD*, *APPENDIX_HEAD*
10	*MOD_HEAD*, *SUBHEAD*, *SIGIL*, *TEST_SIGIL*
8	*INDENTED_LINE*, *BLANK*, *BULLET*
5	*REF*
1	*LINE_START_TEXT*, *PROSE_TEXT* (fallback)

Table 1. Notlob terminal priorities.

Table 2. Command Overview for the Notlob Toolchain.

Command	Description
`run`	Assemble and execute a `.lob` file
`test`	Run all claims in a `.lob` file, or the whole project
`build`	Assemble a `.lob` file (or the whole project) to source artifacts
`weave`	Render a `.lob` file (or the whole project) as Markdown
`graph`	Export the package name-graph (JSON or Turtle RDF)
`query`	Query the package name-graph
`check`	Run semantic checks on the project name-graph
`init`	Initialise a new notlob project in the current directory
`new`	Create a new `.lob` module
`docs`	Write the language reference to `notlob-docs/`
`mcp`	Start the MCP tool server (stdin/stdout)

3 Implementation

These design ideas have been implemented in an open source Python project[1]. A Lark grammar [29] is used as the basis of the parser.

Bindings have been implemented for Haskell, Python, and TypeScript. Sigils allow for working with external tooling and the executable layer, particularly `~external` for referencing external sources without binding support, `~on-build` as a hook for external invocation, and `~keep-generated-src` for retaining generated source files in the target executable language.

Notlob source files end with a `.lob` suffix. Files are laid out on the file system using a deterministic rule where each word in the file heading is a new subdirectory. So a file headed *#Roman Numerals* is expected to be found in `roman/numerals.lob`, and this is enforced by the compiler.

Command line tools have been provided to work with the notlob language, as summarised in Table 2. This includes standard build time tooling to compile, resolve dependencies, and run tests, as in tools such as `maven` or `stack`. The `weave` command is a literate programming-inspired mechanism for rendering the project as Markdown. In working with notlob, the author did not primarily work through separate documentation and code artifacts, as with Knuth's WEB [18], but rather with the `.lob` source files themselves. However documentation output still has its uses for other forms of publication and distribution.

The name-graph can be exported using `graph` or queried with `query`. This also includes export as RDF for possible use in semantic web tools [20]. Deterministic checks for semantic consistency are executed by the `check` command. Missing imports and unused references are treated as errors. Possible typos (similar names off by 1-2 characters), violated conventions, and other style checks are provided as information-only warnings. The name-graph is intended for use as structured data input to LLMs to suggest semantic inconsistencies across prose and code elements of the codebase. The `query` command can also be used for graph-based navigation by machine agents, similar to keyboard shortcuts to jump to type declarations or function callers in an IDE.

The `init` command creates a new basic notlob project with instructions for coding agents on how to use notlob. The same set of commands can also be accessed via an MCP server.

The notlob implementation and the example projects were built with the assistance of Claude Code, using the Sonnet and Opus models.

[1]Source code and packages available at https://github.com/adamburkegh/notlob.

```
#Roman Numerals

Convert integers to Roman numeral strings. The numeral table
maps each milestone value to its symbol. Conversion is greedy: find
the largest milestone that fits, append its symbol, subtract its value,
repeat.

    numerals :: [(Int, String)]
    numerals =
        [ (1000, "M"), (900, "CM"), (500, "D"),
          (400, "CD"), (100,  "C"), (90,  "XC"),
          (50,  "L"), (40,  "XL"), (10,   "X"),
          (9,   "IX"), (5,   "V"), (4,   "IV"),
          (1,    "I")
        ]

    toRoman :: Int -> String
    toRoman 0 = ""
    toRoman n = snd h ++ toRoman (n - fst h)
      where h = head $ filter ((<=n) . fst) numerals

~example

    toRoman 1    == "I"
    toRoman 6    == "VI"
    toRoman 1994 == "MCMXCIV"

##Positive and Non-Empty

The length of the result is always positive for positive inputs, and
toRoman never returns an empty string for a positive integer.

~property

    prop_positive :: Int -> Bool
    prop_positive n =
        let m = abs n `mod` 4000 + 1
        in not (null (toRoman m))

-

#Tests

##basic

    toRoman 0    == ""
    toRoman 1    == "I"
    toRoman 5    == "V"
    toRoman 10   == "X"
    toRoman 50   == "L"
    toRoman 100  == "C"
    toRoman 500  == "D"
    toRoman 1000 == "M"

##subtractive

    toRoman 4    == "IV"
    toRoman 9    == "IX"
    toRoman 40   == "XL"
    toRoman 90   == "XC"
    toRoman 400  == "CD"
    toRoman 900  == "CM"

##compound

    toRoman 2024 == "MMXXIV"
    toRoman 3999 == "MMMCMXCIX"

The 1994 case stacks all three subtractive forms at once (CM, XC,
IV), so it's worth naming on its own.

~test mcmxciv

    toRoman 1994 == "MCMXCIV"
```

Fig. 4. A notlob program for Roman numerals, using the Haskell binding.

4 Programs and Experiences

The following section describes programs developed in notlob.

4.1 Roman Numerals

Roman Numerals is a small program to parse and translate Arabic numerals to Roman. A listing of the main program is in Figure 4. This example illustrates the heading and subheading syntax, the use of `~example` and `~property` to support the description, and the macro-structure of main program and supporting facts[2]. An excerpt of the name-graph is shown in Figure 5 using the output of `notlob graph`. Nodes for modules, symbols and tests can be seen, as well as edges for definitions, uses and tests. Nodes reference their location in source files to allow precise navigation among related elements of code. A unique address is generated for each node from its location on the file system and within source files.

[2]This and other examples can be found in the notlob project itself under https://github.com/adamburkegh/notlob/tree/main/examples.

```
{
  "nodes": [
    {
      "address": "roman",
      "label": "Roman",
      "kind": "MODULE",
      "start_line": 1
    },
    {
      "address": "roman/numerals/app",
      "label": "Roman Numerals App",
      "kind": "MODULE",
      "start_line": 1
    },
    {
      "address": "roman/numerals/app#main",
      "label": "main",
      "kind": "SYMBOL",
      "start_line": 6
    },
    {
      "address": "roman/numerals/app#example#1",
      "label": "example#1",
      "kind": "EXAMPLE",
      "start_line": 9
    },
...
{
      "address": "roman/numerals#Positive Length Natural Numbers",
      "label": "Positive Length Natural Numbers",
      "kind": "SUBHEADING",
      "start_line": 29
    },
    {
      "address": "roman/numerals#Tests#basic",
      "label": "basic",
      "kind": "TEST",
      "start_line": 45
    },
...
  "edges": [
...
    {
      "source": "roman/numerals#Positive Length Natural Numbers#property#1",
      "target": "roman/numerals#Positive Length Natural Numbers#property#1#prop_positive",
      "kind": "DEFINES"
    },
    {
      "source": "roman/numerals/app",
      "target": "roman/numerals",
      "kind": "IMPORTS",
      "start_line": 16
    },
    {
      "source": "roman/numerals/app#main",
      "target": "roman/numerals#toRoman",
      "kind": "USES",
      "start_line": 7
    },
...
}
```


Fig. 5. Excerpt of the output of `notlob graph`, showing part of the name-graph for this the Roman Numerals program in Figure 4.

4.2 Petri Net Chomper

Petri Net Chomper is a web based game inspired by classic arcade games, but using Petri nets [1] as maps. It was developed as a fun puzzle game and a teaching tool for those new to Petri nets. The implementation uses the TypeScript binding.

Chomper was developed using a new agent session as a case study on a small to medium sized project, and source of design feedback. It has ten source files divided across modules for Petri net data structures and behaviour, game play, rendering, and I/O. The file `overview.lob` is a 73-line prose overview of the project that is also a notlob source file, referring to the concepts and modules, and part of the name-graph itself.[3]

Chomper was useful both as an extended TypeScript example, and a source of design pressure to introduce and improve integration with external build and execution points. It was also notable that some development tooling built into platform coding agents, such as Claude Code, use optimisations that undermine the context sharing advantages of notlob file adjacency. For example, Claude will delegate to tools that execute `grep` commands with output restricted to a small number of lines. Prompts to take advantage of the name-graph and contextual reads mitigated this somewhat; it is also hardly fundamental to the architecture of coding agents, but rather a sign of development tools which are a work in progress.

4.3 Pleiades

Pleiades is a project to independently reproduce the calculations for dating the *Midnight Poem* traditionally attributed to Sappho [7]. It is implemented using the Python binding and third party libraries for astronomical and geographical calculations. This project started with an entirely human-authored specification and design stub. The specification consisted of a prose overview of the motivation and goal. The design was sketched using two function signatures with empty implementations, and two unit tests written as notlob `~example` blocks. This code compiled with tests failing, following a Test Driven Development (TDD) style. The coding agent was then introduced and prompted to implement with reference material such as the source paper and notlob documentation. The session continued using interactive prompting where the agent implemented most code independently. Human editing for style and clarity helped keep the code and prose well-organised [4].

Code, prose and tests produced by the agent followed the design sketch out quite closely. The agent detected type signature inconsistency in the example code almost immediately, and asked for input to resolve. It suggested a new module for historical dates, as BCE dates are not supported by Python standard libraries, which it delivered in a tight literate style with test coverage. The agent also propagated unfortunate human spelling mistakes in the proper names Mytilene and Alcyone. Under-examined assumptions introduced in the original specification, such as the direction of Mt Olympus of Lesbos from Mytilene, were also propagated without being flagged. The implementation included agent-written code making idiomatic use of astronomical libraries originally unknown to the developer.

A round of project review by agents resulted in increasingly fine-grained, sometimes pedantic, criticism, reminiscent of academic paper reviews and probably related to the research content. It also yielded a change to horizon calculation that impacted the calculated dates and may improve the result in the paper.

4.4 Design Observations

This section describes partial projects and cross-cutting concerns.

[3] Code for the project can be found at https://github.com/adamburkegh/pn-chomper.
[4] Code and commentary https://github.com/adamburkegh/pleiades, including tag v0.1 before introduction of the coding agent.

An experimental notlob port of an established Digital Signals Processing (DSP) project written in Rust resulted in the coding agent spontaneously adding property tests which surfaced a subtle bug in the original library. The bug related to instability in the clamping calculation that had not been caught by other testing. This project also required developing an experimental Rust binding. Work was done by another developer who had not otherwise contributed to the notlob codebase[5].

As design guidance, we suggest that prose in notlob programs should mostly provide additional information the code does not, rather than be a redescription of the code itself. Prose can explain the motivation for a feature, design constraints, computational complexity, or related parts of the system that are not directly related through executable paths like function invocation. With the inclusion of heading references, such conceptual dependencies can also be made visible in the name-graph structure itself.

After the establishment of a running notlob implementation with a number of example programs, two Claude Code chat sessions were initiated to provide design criticism informed by the ideas of the project but independent of any particular codebase. The prompt used, in part, was:

> I would like to engage you in the new role of notlob critic. Notlob is a new experimental literate programming environment. (URL)
> As notlob critic, you would draw on software engineering and literary practice, a little high theory, and a deep knowledge of this new tool. You will write in the precise, well-crafted prose of a good longform book reviewer, perhaps working for the New Yorker or the London Review of Books.
> From time to time I will ask your opinion of notlob projects, or on the shape of notlob itself. Like good designers and artists, you should try and develop a point of view, and have opinions on the development on the tool, its usage, and comparable projects. As well as reviews of particular projects, you would eventually author a style guide. We can also maintain informal notes - you can suggest the best way to store those as artifacts.

In the second critic session, the prompt was varied, to add:

> As a diversity of opinions is useful to me, as an experiment, let's say you are going through a gonzo-influenced period, inspired by Hunter S Thompson, David Foster Wallace, and Patricia Lockwood.

The critic sessions were engaged to help hold the theory of notlob without being tied to the fine details of implementation. Both found issues and suggested improvements. In general, the gonzo critic session found more bugs, in both notlob language projects and notlob itself. This included errors of function and drift between prose and code. The critic sessions also suggested fixes in the target projects, such as the promotion of implicit principles mentioned in prose to declared checkable `~property` blocks. We note this as an anecdotal observation rather than the result of a controlled experiment.

Across multiple agent-assisted coding projects we observed a tendency for agents to neglect the declarative artifacts that scaffold their work. In notlob itself, the Lark grammar file was created, but many keywords were handled in a Python parsing layer, until specific re-engineering lifted them back into the grammar. On another occasion, USES nodes were added in a way that limited the scope only to external artifacts, a scope-reducing literal interpretation that undermined the usefulness of the feature. In Chomper, no property tests were added, even while formal properties of Petri nets were asserted in prose and essential to the functioning of the game. In another project outside of notlob, executable workarounds grew around LinkML schemas established as metadata descriptors. The recurrence across four

[5]See patches-dsp project developed by Dominic Fox https://github.com/poetix/notlob/tree/rust-bindings/examples/patches-dsp

different artifact types suggests the dynamic is general rather than tool-specific. In all of these cases human inspection and intervention was necessary to retain these artifacts as load-bearing points of consistency in the design, in a dynamic that will be eerily familiar to senior developers and architects on teams of human software developers everywhere.

5 Related Work

A number of language designs have been advanced for structuring interactions with LLMs. The Language Model Query Language (LMQL) [3] provides an SQL-like declarative language for structuring LLMs query prompts. DSPy [16] defines executable data pipelines for LLMs using function declarations that take and emit natural language. SGLang [32] structures programs that call into LLMs for their work, including LLM-aware language primitives like `gen`, `select` or `image`. APPL [8] is also intended to structure prompts within programs, using a special Python function decorator. Functions so decorated have their comment used as a prompt. Each of these tools improves prompting with program syntax, but without the focus on code and documentation as durable artifacts used by human programmers and LLM coding agents in notlob.

Literate programming was conceived by Knuth [18, 19] and the broader intellectual project has had a great influence in research and practice, with documentation tools a part of many language toolkits, as well as an influence on coding notebooks such as Jupyter [17]. Comprehensive review of this impact is beyond this paper, though we will note that well-structured prose explanations are not a routine feature of program source code observed in the wild. Notlob sits closer to the syntactically spare style of Ramsey's noweb [27] than Knuth's original WEB. Like noweb, notlob is language-independent across bindings, and the literate layer is orthogonal to the code language. More recently, LLMs have triggered what one paper calls a "renaissance of literate programming" [31]. The connection to literate programming can be a process link, that is, disciplined use of LLMs to summarise and regenerate code at different units of structure [28]. It is also linked in the context of new tools. Goldfish Scheme [31] is a dialect of Scheme used to structure precise natural language prompts with documentation, with a target of the development of large programs. The authors introduce the term "Interoperable Literate Programming", to summarise three principles of literate programming: flexible code organisation, tangling (automated processing for a compilable source), and bidirectional workflow. They place Jupyter as a literate programming tool, but not ILP, due to the difficulty of integrating or exporting notebooks into a broader project structure. Using this taxonomy, notlob is an ILP project. Source files can be organised in familiar modular ways and automatically made into reusable executable artifacts. Experiments with Goldfish Scheme show smaller language models using literate programming can be more effective software developers than larger language models using conventional programming languages alone.

As well as literate programming, the notlob design is influenced by tools for Behaviour Driven Development (BDD) [4, 24], and Domain Specific Languages [21]. Behaviour Driven Development tools supply natural language-like use cases as code artifacts which are also executable functional tests. BDD existed before LLMs, so a natural language translation layer of some kind was always needed, reducing the use cases to a kind of domain pidgin, and in our experience exerting flattening pressure on the underlying test API. In notlob there is no translation layer, as using LLMs, well-crafted natural language and precise asserted facts and other executable quality mechanisms can reinforce each other in machine checkable common artifacts. Notlob also combines the importance of names, from BDD, with the web of name relationships made use of by ontologies and the semantic web [20].

Behaviour Driven Development tests, and notlob programs, are related to Domain Specific Languages (DSLs) [21], and the interleaved prose and executable fragments in notlob programs make it a kind of Embedded DSL [14]. Notlob itself is not a DSL toolkit, in that it is not primarily a mechanism for introducing custom syntax for effective executable

expression of domain problems. It is not so much a Domain Specific Language as a Domain Entangling Language, evolving a design approach from literate programming and Behaviour Driven Development.

Our approach contrasts with the emerging tools and software methodologies of Specification Driven Development (SDD) [26]. SDD focuses on structured natural language specifications as primary artifacts maintained upstream of code generated by LLMs. Criticism of SDD [5] points out the impact of LLM non-determinism on generated code quality, as well as struggles with articulating intent at the most useful levels of abstraction. SDD reinvents waterfall [2] in LLM-accelerated form, but we doubt this will resolve waterfall's problems. SDD focuses on specification as the start of a pipeline. notlob co-locates specifications, code and tests at the same workbench, ready to be worked on together in frequent iterations by humans and machine agents. The specification is not upstream of code, but beside it. We follow the view of James Noble, that while natural language in programming presents engineering opportunities, "switching from a formally defined programming language to the implicit prompt language of an AI model, with zero specification, zero syntax, zero semantics, zero consistency across models and between releases, has zero appeal" [23].

6 Conclusion and Future Work

This paper presents a programming language and toolset to solve some problems of natural language-literate coding agents by using the techniques of literate programming. The design provides conceptual co-location within files, and in a parse-tree based name-graph, with the intent that coding agents can make more effective use of the context window and necessary external memory artifacts. The language and toolset was implemented in Python, along with a number of example programs of small to medium size. Future work may explore bindings with multiple execution languages for "full-stack" concept descriptions, experimental evaluation with zero- or one-shot prompts, notlob-aware coding agents, and the construction of larger systems.

Acknowledgments

Thanks to Phil Cook and Dominic Fox for their thoughts on this research.

References

[1] F. Bause and P.S. Kritzinger. 2002. *Stochastic Petri Nets: An Introduction to the Theory.* Vieweg+Teubner Verlag.
[2] Herbert D. Benington. 1987. Production of Large Computer Programs. *IEEE Annals of the History of Computing* 9, 4 (1987), 350–361.
[3] Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. 2023. Prompting is programming: A query language for large language models. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1946–1969.
[4] Sumit Bisht. 2013. *Robot framework test automation.* Packt Publishing Ltd.
[5] Birgitta Böckeler. 2025. Understanding Spec-Driven Development: Kiro, spec-kit, and Tessl. martinfowler.com. https://martinfowler.com/articles/exploring-gen-ai/sdd-3-tools.html
[6] A. Cockburn. 2006. *Agile Software Development: The Cooperative Game.* Pearson Education.
[7] Manfred Cuntz, Levent Gurdemir, and Martin George. 2016. Seasonal Dating of Sappho's 'Midnight Poem' Revisited. *Journal of Astronomical History and Heritage* 19, 1 (2016), 18–24.
[8] Honghua Dong, Qidong Su, Yubo Gao, Zhaoyu Li, Yangjun Ruan, Gennady Pekhimenko, Chris J Maddison, and Xujie Si. 2025. Appl: A prompt programming language for harmonious integration of programs and large language model prompts. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers).* 1243–1266.
[9] M. Feathers. 2004. *Working Effectively with Legacy Code.* Pearson Education.
[10] Michael Feathers. 2006. System Metaphor. WikiWikiWeb (C2 Wiki). https://wiki.c2.com/?SystemMetaphor Accessed: July 20, 2026.
[11] Dominic Fox. 2026. Holding A Theory. https://codepoetics.substack.com/p/holding-a-theory
[12] John Gruber. 2004. Markdown. https://daringfireball.net/projects/markdown/. Accessed: 2026-07-20.
[13] Pascal Hitzler. 2021. A review of the semantic web field. *Commun. ACM* 64, 2 (2021), 76–83.
[14] Paul Hudak. 1996. Building domain-specific embedded languages. *Comput. Surveys* 28, 4es (1996), 196.

[15] Andrej Karpathy. 2023. *The hottest new programming language is English.* X (formerly Twitter). https://twitter.com/karpathy/status/1617979122625712128

[16] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Saiful Haq, Ashutosh Sharma, Thomas Joshi, Hanna Moazam, Heather Miller, et al. 2024. DSPy: compiling declarative language model calls into state-of-the-art pipelines. In *International Conference on Learning Representations*, Vol. 2024. 54928–54958.

[17] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, and Carol Willing. 2016. Jupyter Notebooks—a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, Fernando Loizides and Birgit Schmidt (Eds.). IOS Press, 87–90. doi:10.3233/978-1-61499-649-1-87

[18] Donald Ervin Knuth. 1984. Literate programming. *The computer journal* 27, 2 (1984), 97–111.

[19] Donald E Knuth. 1992. Literate Programming. Number 27 in CSLI Lecture Notes. *Center for the Study of Language and Information* (1992), 349–358.

[20] Ora Lassila, James Hendler, and Tim Berners-Lee. 2001. The semantic web. *Scientific American* 284, 5 (2001), 34–43.

[21] Marjan Mernik, Jan Heering, and Anthony M Sloane. 2005. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)* 37, 4 (2005), 316–344.

[22] Peter Naur. 1985. Programming as theory building. *Microprocessing and microprogramming* 15, 5 (1985), 253–261.

[23] James Noble. 2024. Automatic Programming vs. Artificial Intelligence. In *Proceedings of the 1st ACM International Conference on AI-Powered Software* (Porto de Galinhas, Brazil) *(AIware 2024)*. Association for Computing Machinery, New York, NY, USA, 144–146. doi:10.1145/3664646.3664775

[24] Dan North. 2006. Introducing BDD. dannorth.net. https://dannorth.net/introducing-bdd/

[25] Ashwin Pajankar. 2017. *Python Unit Test Automation.* Springer.

[26] Deepak Babu Piskala. 2026. Spec-driven development: From code to contract in the age of AI coding assistants. *arXiv preprint arXiv:2602.00180* (2026).

[27] Norman Ramsey. 1994. Literate programming simplified. *IEEE software* 11, 5 (1994), 97–105.

[28] Kensen Shi, Deniz Altınbüken, Saswat Anand, Mihai Christodorescu, Katja Grünwedel, Alexa Koenings, Sai Naidu, Anurag Pathak, Marc Rasi, Fredde Ribeiro, et al. 2025. Natural language outlines for code: Literate programming in the llm era. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*. 150–161.

[29] Erez Shinan. 2026. *Lark: A parsing toolkit for Python.* https://github.com/lark-parser/lark

[30] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[31] Wuyang Zhang, Yansong Li, Zeyu Dong, Yu Wu, Yingyao Zhou, Duolei Wang, Songsirou Xing, Chichun Zhou, and Da Shen. 2024. Renaissance of literate programming in the era of llms: Enhancing llm-based code generation in large-scale projects. *arXiv preprint arXiv:2502.17441* (2024).

[32] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody H Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. 2024. SGLang: Efficient execution of structured language model programs. *Advances in neural information processing systems* 37 (2024), 62557–62583.

Received 2026