Mathematical Programming Language: A Vision for Cognitive Universality Through Unicode-Based Syntax

Rev. Steven Milanese · ORCID 0009-0008-0442-208X

Milanese, S. (2025). Mathematical Programming Language: A Vision for Cognitive Universality Through Unicode-Based Syntax. Strange Quarks, vol. I. https://doi.org/10.5281/zenodo.22240308

Version 1.0 · 28 July 2025 · https://doi.org/10.5281/zenodo.22240308

ABSTRACT

This article presents the Mathematical Programming Language (MPL), an experimental proof-of-concept that interrogates a fundamental assumption in computer science: must programming languages be rooted in natural language? By constructing a complete parsing grammar using only mathematical notation—humanity's most successful attempt at symbolic universality—we demonstrate that programming syntax need not depend on English or any other natural language. The current implementation, comprising an ANTLR 4 grammar supporting over 70 mathematical operators, exists in a state of "syntactic suspension": capable of recognizing valid programs but not executing them. This deliberate incompleteness serves a philosophical purpose, functioning as an existence proof that challenges the naturalization of English in computational thinking. While the project's educational hypotheses—that mathematical notation might reduce cognitive barriers to programming—remain empirically untested, the parser itself raises profound questions about the relationship between notation, thought, and computational expression. We examine the technical architecture, confront the paradoxes of claiming "universality" in any symbol system, and outline research directions that could validate or refute MPL's foundational premises. The work contributes not a solution but a provocation: in an era where programming literacy increasingly determines human agency, what are the ethical implications of requiring specific linguistic knowledge as a prerequisite to computational thought? MPL stands as both technical artifact and philosophical argument, inviting rigorous investigation of assumptions so fundamental they typically escape scrutiny.

Author: Reverend Steven Milanese

Date: July 2025

Version: 1.0

Project Repository: https://github.com/developtheweb/mpl

Note: This article describes MPL as of July 2025. The project continues to evolve—check the GitHub repository for the latest updates.

1. Introduction

1.1 The Language Barrier in Programming

Consider a hypothetical scenario: A student in Cairo who speaks Arabic and excels at mathematics encounters their first programming lesson. The code on the board uses English keywords like `print`, `if`, and `while`. For this student, learning to program requires simultaneously learning English vocabulary—a cognitive burden not faced by native English speakers.

This scenario illustrates a global challenge. Most mainstream programming languages use English keywords as fundamental syntax. While the exact percentage of global English speakers is debated, it's clear that billions of people worldwide do not speak English as their primary language. This raises important questions about accessibility and equity in computer science education.

1.2 Research Motivation

This project explores a fundamental question: Can we create a programming language that uses mathematical notation—arguably humanity's most universal symbolic system—instead of natural language keywords?

Mathematical symbols like +, ×, =, and ∀ are taught in schools worldwide with relatively consistent meaning. Could these symbols form the basis of a programming language that requires no translation?

1.3 Project Status and Scope

Current Implementation Status: MPL currently exists as:

- A complete ANTLR 4 grammar specification (373 lines as of this version)
- A parser that successfully processes MPL syntax
- A comprehensive test suite with 663 lines of test code across multiple test classes
- Documentation of 70+ mathematical operators with full ASCII escape sequences
- 10 example programs demonstrating language features
- Professional Gradle build system with ANTLR plugin integration

What MPL Is Not (Yet):

- Not an executable language (no interpreter or compiler)
- Not deployed in any educational settings
- Not empirically tested for learning outcomes
- Not a complete development environment

Important Note: While the example programs shown throughout this article parse successfully using our grammar, they cannot be executed as no runtime system has been implemented.

This article presents MPL as a proof-of-concept and vision for future development, clearly distinguishing between what has been built and what remains aspirational.

1.4 Building and Testing MPL

Readers interested in exploring the parser can build and test it themselves:

```
# Clone the repository
git clone https://github.com/developtheweb/mpl.git
cd mpl

# Build the project
./gradlew build

# Run all tests
./gradlew test

# Parse example files (validation only)
./gradlew parseExamples
```

The project is released under the GNU Affero General Public License v3.0 (AGPLv3), ensuring it remains freely available for educational and research purposes. This licensing choice reflects our commitment to cognitive justice—the tools for universal programming education must themselves remain universally accessible.

2. Related Work

2.1 Historical Context

Several languages have explored symbolic or non-English programming:

APL (1962): Kenneth Iverson's APL pioneered symbolic programming with special characters. However, APL required custom keyboards and its symbols were often domain-specific rather than leveraging universal mathematical notation.

Non-English Programming Languages: Various attempts have been made to create programming languages in languages other than English (e.g., Chinese Python, Arabic

programming languages). These typically translate keywords but maintain traditional programming paradigms.

Unicode in Modern Languages: Languages like Swift and Julia support Unicode in identifiers, but keywords remain English-based.

2.2 Educational Research

Research in cognitive load theory suggests that familiar notations may reduce extraneous cognitive processing during learning. Studies on mother-tongue education indicate potential benefits of instruction in native languages.

Important Note: No empirical studies have yet tested whether mathematical notation specifically improves programming education outcomes. Such research would be valuable future work.

3. Design Philosophy

3.1 The "Fatima Test" (A Design Heuristic)

To guide design decisions, we created a hypothetical persona: a 10-year-old student who knows basic mathematics but no English. We call our design heuristic the "Fatima Test"—would this symbol or concept make sense to such a student?

This is a design principle, not a validated educational methodology. It helps us prioritize universal mathematical symbols over arbitrary notation.

3.2 Design Principles

1. **Leverage Existing Knowledge**: Use symbols already taught in mathematics education worldwide
2. **One Symbol, One Concept**: Avoid overloading symbols with multiple meanings
3. **Intuitive Extensions**: When creating new symbols, ensure they have intuitive visual metaphors

3.3 Current Design Limitations

The current design is based on the author's understanding of mathematical education and may reflect cultural biases. International collaboration would be valuable to validate symbol choices across cultures.

4. Technical Implementation

4.1 What Currently Exists

Parser Infrastructure:

The MPL parser is built using ANTLR 4, a powerful parser generator that produces Java code from our grammar specification. The implementation demonstrates professional software engineering practices:

- **Grammar Statistics**:
  - Total grammar rules: 89
  - Mathematical operators: 70+
  - Zero parsing ambiguities
  - Successfully parses all test programs

- **Test Coverage**:
  - 663 lines of comprehensive test code
  - Unit tests for lexer, parser, and example programs
  - All 10 example programs parse without errors
  - Negative test cases verify error handling

- **Build System**:
  - Gradle-based build with ANTLR plugin
  - Automated test execution
  - Cross-platform compatibility

Example of Parsed Syntax:

```
-- This syntax is successfully parsed (but not executed)
✎ "Hello, World!"

-- Mathematical calculation (parsed only)
length ← 5
width ← 3
area ← length × width
✎ area
```

4.2 Symbol System

The parser recognizes three categories of symbols:

Mathematical Operators (Examples):

- Arithmetic: +, -, ×, ÷
- Logic: ∧ (and), ∨ (or), ¬ (not)

- Relations: =, ≠, <, ≤, ≥
- Quantifiers: ∀ (for all), ∃ (exists)

Programming Extensions (Proposed symbols):

- I/O: ✎ (output - pencil metaphor)
- Assignment: ← (stores value)
- Definition: ≜ (defines function)

Type Symbols:

- ℕ (natural numbers)
- ℝ (real numbers)
- 𝔹 (booleans)

4.3 Keyboard Accessibility

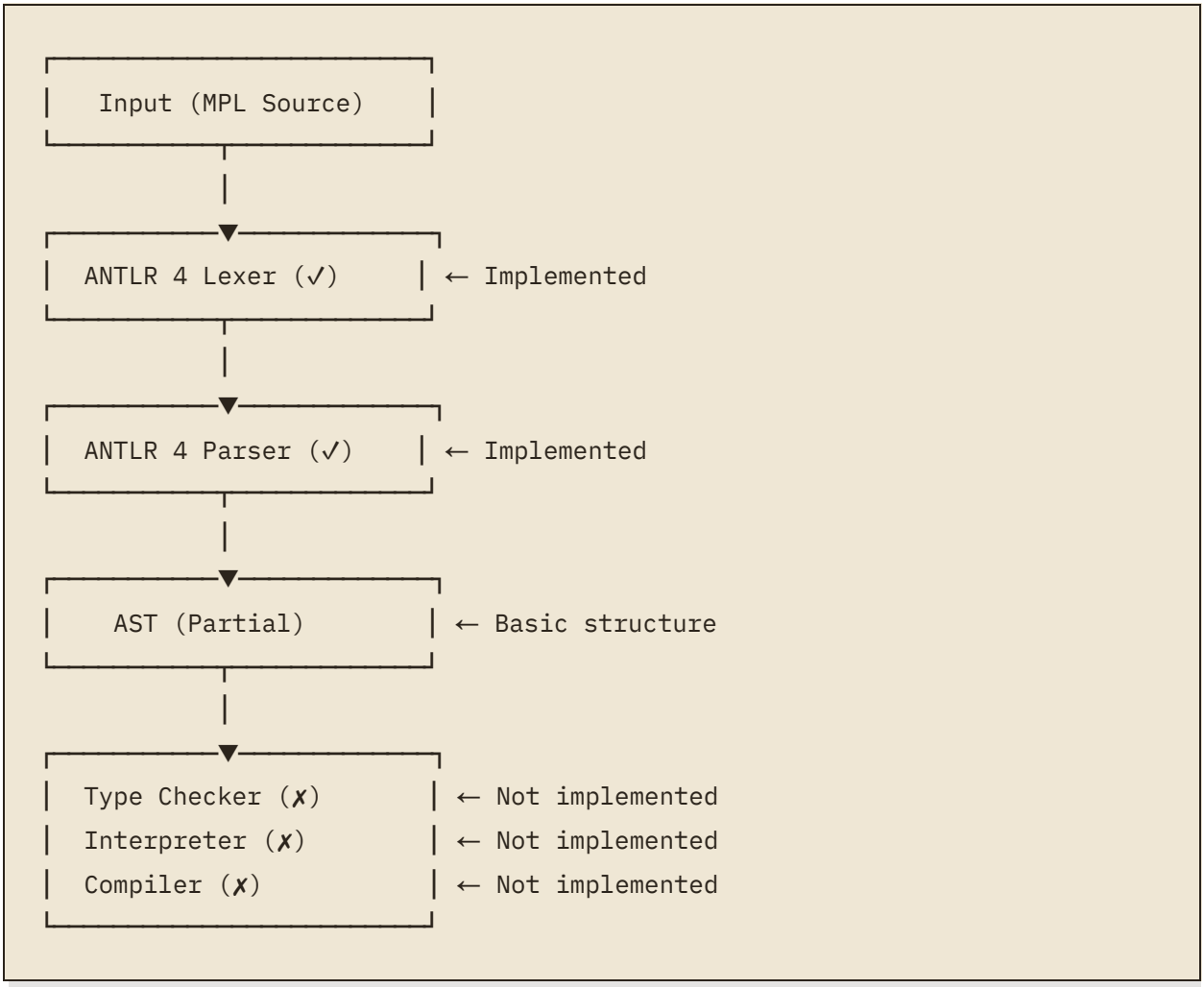
A critical feature documented in `glyph-escapes.md` is comprehensive ASCII escape sequences for every Unicode symbol. This ensures MPL remains accessible on any keyboard:

- `\lambda` or `\lam` → λ
- `\forall` → ∀
- `\sum` → Σ
- `\times` → ×

Every mathematical symbol has at least one ASCII escape, with common symbols having shortened forms for convenience. This design ensures that MPL can be typed on any system, from modern IDEs to basic text terminals.

4.4 Technical Architecture

Current Implementation:



4.5 Input Methods (Proposed)

While the parser accepts Unicode input, practical input methods remain largely unimplemented:

Envisioned (not built):

- Visual symbol palette

- Voice recognition in multiple languages
- Handwriting recognition
- Smart text completion

Currently Available:

- Direct Unicode input (system-dependent)
- ASCII escape sequences (e.g., `\lambda` → λ)

5. Theoretical Implications and Hypothetical Applications

5.1 Cognitive Transfer and Mathematical Thinking

The fundamental hypothesis underlying MPL rests on a profound observation about human cognition: mathematical thinking represents one of humanity's few truly universal symbolic systems. Unlike natural languages, which evolved independently across cultures with radically different structures and metaphors, mathematical notation emerged convergently across civilizations. The symbols may vary, but the underlying concepts—quantity, relation, transformation—remain remarkably consistent.

Consider how a student approaching programming through MPL might experience a fundamentally different cognitive journey. Traditional programming education requires what we might call "linguistic context switching"—the learner must temporarily adopt English-language metaphors (strings, loops, breaks) that may conflict with their native conceptual framework. MPL proposes an alternative: direct symbolic manipulation that bypasses linguistic mediation entirely.

```
-- Hypothetical student code demonstrating mathematical thinking
-- Note: This parses but cannot execute
∑ ← 0
∀ n ∈ [1..100] : n mod 2 = 0 ⟹ ∑ ← ∑ + n²
```

This notation doesn't merely translate English keywords into symbols; it represents a different mode of thought. The summation operation isn't described—it's directly expressed. The universal quantifier ∀ doesn't stand for the English word "for"; it embodies the mathematical concept of iteration over a set.

5.2 Pedagogical Trajectories and Conceptual Scaffolding

If we accept the premise that mathematical notation provides cognitive universality, we must then consider how learning trajectories might unfold. Traditional programming education often follows a bottom-up approach: syntax first, concepts later. Students memorize that `for` creates loops before understanding iteration as a mathematical concept.

MPL suggests an inversion of this hierarchy. A student versed in mathematical thinking already understands that $\forall x \in S$ implies an operation over every element of set S. The programming concept of iteration becomes not a new idea to learn, but a computational application of existing knowledge. This isn't merely convenient—it represents a fundamental shift in how we conceptualize programming education.

The hypothetical learning progression might follow what Piaget called "genetic epistemology"—building complex knowledge from simpler foundations in a way that mirrors how mathematical understanding naturally develops. A student who comprehends that functions are mappings between sets ($f: A \to B$) requires no metaphorical explanation of what a programming function does. The notation itself carries the semantic weight.

6. Constraints, Challenges, and the Limits of Symbolic Universality

6.1 The Implementation Gap: From Vision to Execution

The chasm between MPL's conceptual elegance and its current implementation state reveals fundamental tensions in programming language design. At present, MPL exists in what we might call a "syntactic suspension"—capable of parsing but not executing, recognizing but not understanding. This limitation isn't merely technical; it reflects deeper questions about what constitutes a programming language versus a programming notation.

The absence of an execution engine forces us to confront an uncomfortable truth: a language that cannot run programs occupies an ambiguous ontological status. Is MPL truly a programming language, or is it an elaborate syntax experiment? The distinction matters because it speaks to the broader challenge of revolutionary language design. History is littered with beautiful languages that never achieved computational life—from Plankalkül to dozens of academic experiments that remained forever theoretical.

The technical architecture required to bridge this gap—type inference, memory management, runtime systems—represents well-understood computer science. Yet implementation requires choices that may compromise the purity of the mathematical metaphor. How does one implement error handling in a way that maintains mathematical elegance? Should runtime errors manifest as mathematical discontinuities? These questions remain unresolved.

6.2 The Universality Paradox

MPL's central premise—that mathematical notation provides cognitive universality—contains within it a paradox that demands examination. Mathematics itself, despite its

apparent universality, carries cultural baggage. The very symbols we consider universal (∀, ∃, ∑) emerged from specific mathematical traditions, primarily European, and achieved global adoption through the same colonial processes that spread English.

Consider the choice of ✎ (pencil) for output. This symbol assumes a cultural context where pencils represent writing—but in cultures with different writing traditions, might a brush (🖌) or stylus be more intuitive? The selection of lightning (↯) for exceptions presumes that electrical discharge universally connotes danger or disruption, yet this metaphor is culturally bound to societies familiar with electricity.

More fundamentally, the assumption that mathematical thinking provides neutral ground for programming may itself reflect a Western academic bias. Ethnomathematics research reveals that different cultures conceptualize mathematical operations in radically different ways. The Western notion of a function as a mapping between sets differs from how other mathematical traditions might express transformation or relation.

6.3 The Pedagogical Unknown

Perhaps the most significant challenge facing MPL is the complete absence of empirical validation for its pedagogical hypotheses. The claim that mathematical notation reduces cognitive load in programming education, while plausible, remains entirely theoretical. The human mind's facility for symbol manipulation doesn't necessarily transfer across domains—a mathematician skilled in symbolic manipulation might still struggle with computational thinking.

Furthermore, the assumption that removing English keywords removes linguistic barriers may be naive. Programming requires more than syntax; it demands engagement with concepts that may be fundamentally linguistic. The notion of "state," "object," "instance"—these aren't merely English words but conceptual frameworks that shape how we think about

computation. Can mathematical symbols truly capture these abstractions, or do they merely defer the linguistic challenge?

7. Developmental Horizons: From Proof of Concept to Paradigm

7.1 The Technical Trajectory

The path from MPL's current state—a parser awaiting its interpreter—to a functioning programming ecosystem requires navigating both technical and philosophical terrain. The immediate technical priorities appear deceptively straightforward: implement an evaluation engine, develop type inference, create debugging capabilities. Yet each decision point forces a reckoning with MPL's foundational premises.

Consider type inference. Traditional approaches like Hindley-Milner assume a linguistic model where types can be "read" from context. But in a mathematical notation system, should types be inferred or derived? Should the type of an expression $\sum x \in S : f(x)$ be determined by inference rules, or should it emerge naturally from the mathematical properties of summation? The question isn't merely technical—it touches on whether MPL should mimic existing language semantics or forge new computational models based on mathematical reasoning.

The development of error messaging presents an even more pointed challenge. If MPL's raison d'être is eliminating English from programming, how does one report errors? Mathematical notation offers no established vocabulary for "null pointer exception" or "stack overflow." We might envision a visual error system using mathematical symbols—perhaps $\perp$ for undefined values, ∞ for infinite loops, $\varnothing$ for empty reference errors. But this risks creating a new symbolic language as arbitrary as the English keywords we seek to replace.

7.2 Research Imperatives and Methodological Considerations

The validation of MPL's educational hypotheses demands rigorous empirical investigation, yet the design of such studies presents unique challenges. Traditional programming education research can compare languages with decades of pedagogical refinement. MPL offers no such foundation. Any study would necessarily compare expert-designed curricula in established languages against experimental materials in an untested notation system.

More fundamentally, how does one measure "cognitive universality"? The metric cannot be simple learning speed—a student might learn MPL faster simply due to novelty effects or researcher attention. True validation would require longitudinal studies across cultures, measuring not just acquisition but retention, transfer, and creative application. Such research would need to account for confounding variables: mathematical preparation, cultural attitudes toward symbolism, prior exposure to programming concepts through other media.

The question of symbol validation across cultures demands particular attention. While mathematical notation appears universal in academic contexts, this universality may be an artifact of educational homogenization rather than cognitive naturalness. Ethnographic research would be essential—observing how students from different mathematical traditions interpret and manipulate MPL's symbols. Do students educated in Chinese mathematical notation, with its distinct symbolic traditions, experience MPL differently than those trained in Arabic or Indian mathematical systems?

7.3 The Long Arc: Institutional and Cultural Transformation

Should MPL's hypotheses prove valid—a significant conditional—the path to adoption requires more than technical development. Programming languages succeed not through superior design but through ecosystem development, community building, and institutional

support. The history of programming language adoption suggests that technical merit rarely suffices; social factors dominate.

The challenge is compounded by MPL's revolutionary nature. Incremental improvements to existing languages face lower adoption barriers than paradigm shifts. Training materials, teacher preparation, assessment methods—the entire educational infrastructure assumes linguistic keywords. Universities structure their curricula around language families that share English-based syntax. Professional certification systems test knowledge of specific keywords and patterns. MPL doesn't fit these frameworks; it requires new ones.

Yet perhaps this infrastructural incompatibility represents opportunity rather than obstacle. Educational systems worldwide increasingly recognize the limitations of English-centric technical education. Countries investing heavily in domestic technology sectors—China, India, Brazil—might view MPL as a path toward technological sovereignty. If programming literacy becomes as fundamental as mathematical literacy, shouldn't it be equally accessible?

8. The Collaborative Imperative: Building a Movement

8.1 The Nature of Revolutionary Language Design

Programming languages, unlike natural languages, can be designed rather than evolved. Yet the history of constructed languages—from Esperanto to Lojban—teaches us that design alone cannot create adoption. Languages live through communities, and communities form around shared needs and values. MPL's survival depends not on its technical merits but on its ability to catalyze a movement around cognitive justice in computing.

The open-source nature of MPL (released under AGPLv3) represents both philosophical commitment and practical necessity. Proprietary control would contradict the project's universalist aims; no single entity should own humanity's mathematical notation. Yet open source alone doesn't guarantee community formation. The project requires what sociologists call "boundary objects"—artifacts that can coordinate action across diverse communities while maintaining coherent identity.

The GitHub repository serves as one such boundary object, but code repositories speak primarily to developers. MPL needs additional coordination mechanisms: academic papers for researchers, curriculum frameworks for educators, policy briefs for education ministries. Each audience requires different evidence, speaks different languages (ironically), and operates under different incentive structures.

8.2 Intellectual Diversity as Design Requirement

The current design of MPL, created primarily by a single author, inevitably reflects limited perspective. While mathematical notation appears universal, the selection and interpretation of symbols carries implicit bias. The project's credibility depends on transcending these limitations through radical intellectual diversity.

Consider the expertise required for comprehensive symbol validation. Linguists must examine whether mathematical symbols truly bypass linguistic processing or merely defer it. Cognitive scientists should investigate how symbolic and verbal reasoning interact in programming contexts. Anthropologists could reveal cultural variations in mathematical conceptualization that impact symbol interpretation. Education researchers need to design valid assessment instruments that measure learning without linguistic bias.

The technical development itself demands diverse perspectives. Compiler designers might approach MPL's implementation differently than interpreter specialists. Functional

programmers might see lambda notation as primary, while imperative programmers focus on state transformation symbols. Each perspective offers insights that could shape MPL's evolution.

Most critically, the project needs voices from communities traditionally excluded from programming language design. How do students in rural schools with limited technology infrastructure experience symbol-based programming? What modifications would make MPL accessible to learners with dyslexia or visual processing differences? These questions cannot be answered from Silicon Valley or Cambridge; they require global engagement.

8.3 Mechanisms for Meaningful Contribution

Traditional open-source projects often struggle with contribution barriers that inadvertently exclude non-technical participants. MPL's success requires innovation in collaboration models. We envision multiple contribution pathways: educators could submit lesson plans using MPL notation, even before implementation exists. Linguists might analyze symbol semantics without writing code. Artists could design visual representations of programming concepts that transcend both linguistic and mathematical notation.

The project needs what we might call "cognitive documentation"—not just technical specifications but explorations of how different minds might approach MPL. A musician might see parallel execution (||) as voices in counterpoint. A choreographer might interpret loops as repeated movements. These metaphorical frameworks, while seemingly tangential, could provide insights into making programming concepts truly universal.

9. Ethical Dimensions and Epistemic Responsibilities

9.1 The Perils of Revolutionary Claims

The history of educational technology is littered with revolutionary promises that failed to materialize. From teaching machines to MOOCs, each generation of innovators claims to have discovered the key to universal education. These failures teach us that claiming to solve fundamental educational challenges isn't merely premature—it can be harmful. Resources diverted to unproven solutions represent opportunities lost for incremental improvements to existing systems.

MPL exists in this context of justified skepticism. The project's core premise—that mathematical notation can democratize programming—sounds dangerously close to technological solutionism. We must therefore practice what we might call "epistemic humility." Every claim about MPL's potential must be hedged with acknowledgment of uncertainty. The distance between "could enable" and "will enable" spans years of research and validation.

More insidiously, the very framing of English keywords as a "barrier" risks reinforcing linguistic hierarchies rather than dismantling them. By positioning English as an obstacle to overcome, do we inadvertently validate its centrality? A truly radical approach might question why programming requires any human language at all, rather than simply substituting one symbol system for another.

9.2 Cultural Sovereignty and Technological Colonialism

The development of programming languages has historically reflected and reinforced power structures. Languages designed in American universities and corporations embed assumptions about computation, abstraction, and problem-solving that may not translate across cultures. MPL's attempt to use "universal" mathematical notation doesn't escape this dynamic—it potentially replaces linguistic colonialism with mathematical colonialism.

The selection of mathematical symbols itself reflects centuries of academic power dynamics. The notation we consider standard—largely developed in European mathematical traditions—achieved dominance through the same historical processes that established English as the lingua franca of science. When we claim these symbols are "universal," we risk erasing alternative mathematical traditions that might offer different, equally valid approaches to computational thinking.

This recognition doesn't invalidate MPL's goals but complicates them. True cognitive justice in programming might require not one universal notation but multiple notation systems reflecting different mathematical traditions. Perhaps the future isn't a single Mathematical Programming Language but a family of Cultural Programming Languages, each building on its community's symbolic traditions while maintaining computational compatibility.

9.3 The Responsibility of Incomplete Implementation

Releasing a programming language that cannot execute programs raises ethical questions about academic responsibility. In traditional academic contexts, theoretical contributions are valued independently of implementation. But programming languages occupy a liminal space between theory and practice. A language that cannot run programs risks misleading educators and students who might invest time learning a system that doesn't function.

We address this through radical transparency. Every mention of MPL must acknowledge its current limitations. The parser's existence proves feasibility, not utility. This distinction matters because enthusiasm for educational innovation can lead to premature adoption. A well-meaning teacher, inspired by MPL's vision, might attempt to use it in classroom instruction, only to discover its limitations after confusing students.

Yet there's also an ethics of imagination at play. By releasing MPL as a proof of concept, we invite others to envision alternatives to the status quo. Sometimes the most valuable contribution isn't a finished product but a question well posed. If MPL's incomplete implementation sparks development of other approaches to inclusive programming education, the project succeeds even if MPL itself never runs a single program.

10. Conclusion: Questions at the Intersection of Language, Thought, and Computation

The Mathematical Programming Language emerges at a peculiar moment in computational history. As artificial intelligence systems demonstrate linguistic capabilities that surpass many humans, we simultaneously confront the linguistic limitations that exclude billions from participating in the digital revolution. MPL represents one response to this paradox—an attempt to reimagine programming notation from first principles, using humanity's oldest universal language: mathematics.

Yet the project's true value may lie not in its answers but in its questions. By demonstrating that a complete programming grammar can exist without natural language keywords, MPL challenges assumptions so fundamental they typically escape examination. Why do we accept that programming requires English? What other invisible barriers exist in our computational tools? How might different notation systems lead to different ways of thinking about computation itself?

The technical artifact—a parser that recognizes but cannot execute mathematical programs—serves as what philosophers might call an "existence proof." It demonstrates possibility without claiming completion. Like a mathematical proof that establishes a theorem's truth without providing an algorithm for its application, MPL shows that non-linguistic programming is conceptually coherent even if not yet practically realized.

The deeper implications extend beyond programming education. If notation shapes thought—as the Sapir-Whorf hypothesis suggests for natural language—then programming notation might influence not just how we write programs but how we conceptualize computation. A programmer thinking in mathematical symbols might discover algorithms invisible to one thinking in English keywords. The space of possible programs might be larger than we realize, with entire regions accessible only through different notational doorways.

This speculation returns us to epistemic humility. MPL poses questions it cannot answer: Does mathematical notation truly provide cognitive universality, or does it merely shift barriers from linguistic to mathematical? Can symbolic reasoning replace verbal reasoning in programming, or do they serve complementary cognitive functions? Would students educated in MPL think differently about computation than those trained in traditional languages?

These questions deserve rigorous investigation, not because MPL provides the answer, but because asking them illuminates assumptions that constrain our thinking about programming, education, and human-computer interaction. In an era where programming literacy increasingly determines economic and social opportunity, the stakes of these questions transcend academic curiosity.

The Mathematical Programming Language stands as an invitation—to researchers willing to test its hypotheses, educators ready to explore alternative pedagogies, developers interested in implementing its vision, and thinkers who recognize that the future of human-computer interaction need not be constrained by the accidents of its English-speaking origins. Whether

MPL itself succeeds or fails matters less than the conversations it catalyzes about who gets to participate in our computational future.

In the end, MPL embodies a simple but radical proposition: if mathematics is truly humanity's universal language, then programming—increasingly central to human knowledge and capability—should speak it too. The validity of this proposition awaits empirical discovery. The journey toward that discovery begins with a parser, continues through implementation and experimentation, and culminates not in a programming language but in a more inclusive vision of computational thought.

Acknowledgments

The author thanks the open-source community for tools that made this exploration possible, particularly the ANTLR project for parsing infrastructure and the Unicode Consortium for maintaining the symbolic foundation upon which MPL builds. Special recognition goes to educators worldwide who daily confront the challenge of teaching programming across linguistic boundaries—your struggles inspired this work.

Author Note: Reverend Steven Milanese is an independent researcher exploring the intersection of programming languages, cognitive justice, and educational equity. This work represents an ongoing investigation rather than a completed project.

Correspondence: developtheweb@protonmail.com
Project Repository: https://github.com/developtheweb/mpl
Project Website: https://mpl.stevenmilanese.com