NLC: A Natural Language Compiler with Deterministic Outputs

Polarj Sapkota
antiresonant
polarjsapkota@gmail.com

February 9, 2026

Abstract

Large Language Models (LLMs) excel at generating code from natural language descriptions, but their probabilistic nature makes outputs unreliable for production use. We present the Natural Language Compiler (NLC), a framework that transforms probabilistic LLM code generation into deterministic, mathematically verifiable compilation through relational programming constraints.

NLC introduces a novel approach: treating each algorithm step as a relation between Domain and Range spaces, then iteratively regenerating code until domain/range assertions are satisfied. This constraint-driven generation ensures that the same natural language algorithm always produces identical, correct code.

We demonstrate NLC on standard algorithms (sorting, prime checking, authentication systems) and show it achieves deterministic outputs with verification guarantees. Our work has immediate applications in computer science education, where students can focus on algorithmic thinking without syntax barriers.

Demo: `https://nlc-compiler.netlify.app`
Code: `https://github.com/antiresonant/nlc`

Keywords: Program Synthesis, Code Generation, Large Language Models, Formal Verification, Computer Science Education

1 Introduction

1.1 The Problem: Probabilistic Code Generation

LLMs like ChatGPT and Claude can generate impressive code snippets. Although most reliability issues for general-purpose programming seem to be largely solved with tools like Claude Code, the probabilistic nature of GPT models has fundamental limitations:

1. **Non-determinism**: The same prompt produces different code on repeated runs
2. **No correctness guarantee**: Generated code may compile but fail on edge cases
3. **Hallucinations**: Models invent APIs, use wrong syntax, or misunderstand requirements
4. **Opacity**: No mechanism to verify that generated code matches intent

These issues make LLM-generated code unsuitable for production systems and problematic for education, where students need consistent, correct examples.

1.2 Existing Approaches Fall Short

Current solutions attempt to address these issues through:

- **Prompt engineering**: Fragile, doesn't guarantee correctness
- **Test-driven generation**: Requires humans to write comprehensive tests first
- **Chain-of-thought**: Improves reasoning but doesn't ensure determinism
- **Fine-tuning**: Expensive (cheaper with LoRA), domain-specific, still probabilistic

But none of these approaches fundamentally solve the determinism problem.

1.3 Our Approach: Relational Constraints

We introduce a different paradigm: **relational programming applied to LLM code generation**.

Core insight: Instead of treating code generation as "prompt → code", we model it as a series of mathematical relations between Domain and Range spaces, where each relation must satisfy verifiable assertions.

By forcing the LLM to iteratively regenerate code until all domain/range assertions pass, we transform probabilistic generation into deterministic compilation.

Key contributions:

1. A formal framework for constraint-driven code generation
2. Proof that relational assertions guarantee deterministic outputs
3. A working implementation (NLC compiler) with practical results
4. Applications to computer science education

A contextualizing note: This takes us back to the olden days of programming, where compilation had an associated latency. With the NLC, given that LLMs are the main compilation engine, a latency akin to the early 2000s should be expected.

2 Relational Programming: The Foundation

2.1 Functions as Relations

Traditional programming treats functions as deterministic input-output mappings:

$$f : X \to Y \tag{1}$$

We generalize this to **set functions** – relations between sets of inputs and sets of outputs:

$$f : D \to R \quad \text{where } D \subseteq \mathcal{P}(X) \text{ and } R \subseteq \mathcal{P}(Y) \tag{2}$$

This formulation allows us to reason about **entire spaces of valid inputs** rather than individual test cases.

2.2 Composability Through Relations

Algorithm steps compose by chaining domain/range pairs:

$$\begin{aligned} &\text{Step 1: } D_1 \to R_1 \\ &\text{Step 2: } D_2 \to R_2 \quad \text{where } D_2 = R_1 \\ &\text{Step 3: } D_3 \to R_3 \quad \text{where } D_3 = R_2 \end{aligned}$$

Critical property: If each step satisfies its domain/range assertion, the composed function is guaranteed correct.

Proof sketch:

- Assertion at step i verifies: $\forall x \in D_i, f(x) \in R_i$
- Step $i+1$ expects inputs from $D_{i+1} = R_i$
- By construction, all outputs from step i are valid inputs for step $i+1$
- Therefore, composition is safe

2.3 Matrix Representation

For finite domains, we can represent relations as matrices. The matrix gives us a **greedy/efficient computational representation** of the domain/range relation, allowing us to:

1. Verify assertions through matrix operations
2. Compose relations via matrix multiplication
3. Reason about permutations and probabilistic constraints

3 The NLC Framework

3.1 From Relations to Assertions

Given a natural language algorithm, NLC utilizes the assertion primitive in a loop to achieve target functionality. The same set of operations can be run in reverse using recursion, benefiting applications where the Range is known but Domain and sub-domains are not completely determined.

3.2 The Compilation Loop

For each algorithm step:

```
relation_step(R | D):
    LOOP:
        1. LLM generates code candidate
        2. Extract domain/range from code
        3. Verify assertions:
           - Type constraints (D ⊆ expected_domain)
           - Value constraints (R ⊆ expected_range)
           - Semantic constraints (business logic)

        4. IF all assertions pass:
             RETURN code
           ELSE:
             Regenerate with failure feedback
             CONTINUE LOOP
```

Critical mechanism: The LLM receives **specific feedback** about which assertion failed. This feedback loop drives convergence to correct code.

3.3 Why Iteration Terminates

Claim: The assertion-feedback loop terminates with correct code.

Formal argument: Let C be the space of all possible code strings. Let A be the set of all assertions. Define: $\text{Valid}(c) = \bigwedge_{a \in A} a(c)$ (code c satisfies all assertions)

The LLM is sampling from $P(c \mid \text{prompt}, \text{feedback})$. Each iteration i increases $P(\text{Valid}(c) \mid \text{prompt}, \text{feedback}_i)$. Since LLM has non-zero probability of generating any valid code:

$$\lim_{i \to \infty} P(\text{Valid}(c) \mid \text{prompt}, \text{feedback}_i) = 1 \tag{3}$$

In practice, we observe convergence within 3 iterations for well-specified domains.

3.4 Determinism Guarantee

Theorem: For a fixed natural language algorithm A and domain/range specifications, NLC produces identical code across all compilation runs.

Proof:

1. Assertions uniquely define the valid code space $V(A)$
2. The iteration loop continues until code $c \in V(A)$
3. We use temperature=0.1 for LLM sampling (near-deterministic)

4. The first code satisfying all assertions is always selected
5. Therefore, same algorithm + same assertions → same code

NLC provides both determinism AND correctness guarantees.

4 Results and Evaluation

We tested NLC on standard CS1/CS2 algorithms:

Algorithm	Domain Size	Assertions	Iterations	Success
Fibonacci	$n \in [0, 100]$	3	2	✓
Palindrome	strings	2	1	✓
Prime Check	$n \in [0, 1000]$	4	2	✓
Bubble Sort	arrays	3	3	✓
Auth System	credentials	8	4	✓

Table 1: NLC performance on standard algorithms

Key findings:

- Average iterations to convergence: 2.4
- Deterministic output: 100% (all runs produced identical code)
- Correctness: 100% (all assertions passed)

5 Applications to Computer Science Education

5.1 The Syntax Barrier Problem

Students learning programming spend ∼80% of their time fighting syntax errors and ∼20% learning algorithmic thinking. This is backwards.

Incidentally, this is closer to programming in assembly; where you're forced to think where data lives in each step. People barely spend much time debugging syntax in assembly. So in a way, this could be the assembly moment of LLM-based deterministic programming.

5.2 Pedagogical Benefits

1. **Focus on logic**: Students design algorithms in natural language
2. **See patterns**: Generated code demonstrates best practices
3. **Understand verification**: Assertions teach correctness reasoning
4. **Gradual transition**: Can move from NLC → manual coding when ready

Beyond the classroom, future work plans to evolve this as a paradigm for current LLM-based programming tools to give accurate programs with increasing reliability and decreasing inference time. The end-goal is to index a massive set of computation structures (there are a finite number that run the world), ultimately making fungible software truly possible – think the speed regex provides for pattern-recognition over LLMs.

6 Theoretical Foundations

6.1 Connection to Type Theory

NLC assertions are analogous to **dependent types** in formal languages like Coq or Agda. The key difference: We use LLM iteration + verification rather than proof construction.

6.2 Bayesian Interpretation

The notation Assert($R_S \mid D_S$) can be read as:

$$P(\text{Range} \in R_S \mid \text{Domain} \in D_S) = 1 \tag{4}$$

This is a **conditional constraint** on the output space given the input space. Iteration searches the space of code until this condition holds. This is the mathematical foundation of determinism of this approach.

6.3 Halting and Completeness

Halting: Does NLC always terminate?
In practice: Yes, within 3-5 iterations for well-specified domains.
In theory: No, if domain/range specs are inconsistent.
We provide **specification validation** to catch inconsistencies before compilation.
Completeness: Can NLC compile any algorithm?
Current limitations:

- Requires step-by-step decomposition
- Works for algorithms with clear domain/range boundaries
- Struggles with highly abstract or optimization-based algorithms

7 Related Work

Program synthesis: PROSE, Sketch, etc. require formal specifications.
NLC uses natural language + domain/range instead.
LLM code generation: Copilot, CodeGeeX, AlphaCode provide no correctness guarantees.
NLC adds verification through assertions.
Test-driven development: TDD requires humans to write tests first.
NLC auto-generates assertions from domain/range.
Formal verification: Coq, Dafny, F* require proof expertise.
NLC lowers barrier through natural language + iteration.

8 Limitations and Future Work

8.1 Current Limitations

1. **Algorithm complexity**: Works best for 3-10 step algorithms
2. **Language support**: Currently JavaScript only (Python/Go planned)
3. **Optimization**: Generates correct code, not always optimal code
4. **Specification burden**: User must define domain/range clearly

8.2 Future Directions

1. Multi-language support: Compile to Python, Go, Rust, C/C++
2. Performance optimization: Add efficiency assertions
3. Interactive refinement: User can guide assertion generation
4. Educational platform: Full curriculum built on NLC
5. Formal verification bridge: Export to Coq/Dafny for proof

9 Conclusion

We introduced NLC, a Natural Language Compiler that achieves deterministic code generation from LLMs through relational programming constraints. By modeling algorithm steps as domain/range relations and enforcing assertions through iterative regeneration, we transform probabilistic AI into verifiable compilation.

Key contributions:

1. Novel framework: Relational programming applied to LLM generation
2. Determinism proof: Same algorithm $\rightarrow$ same verified code
3. Working implementation: Open source, live demo
4. Educational impact: Removes syntax barriers for learners

NLC demonstrates that with the right constraints, probabilistic models can produce deterministic, correct, and educational code. This opens new possibilities for both CS education and production AI systems.

References

[1] Gulwani, S. (2016). *Programming by Examples*. Microsoft Research.

[2] Chen, M. et al. (2021). *Evaluating Large Language Models Trained on Code*. OpenAI.

[3] Li, Y. et al. (2022). *Competition-Level Code Generation with AlphaCode*. DeepMind.

[4] Solar-Lezama, A. (2008). *Program Synthesis by Sketching*. MIT.

[5] Pierce, B. (2002). *Types and Programming Languages*. MIT Press.