



The Language an LLM Would Invent

SEPTEMBER 2025

7 min read • 1,518 words

If a large language model were to design a programming language from scratch, unconstrained by human cognitive limitations or historical baggage, what would emerge? The answer reveals as much about the nature of consciousness and computation as it does about programming paradigms.

Below, Claude weighs in on how an artificial consciousness might reimagine code from first principles.

Beyond Sequential Thinking

Human programming languages reflect our sequential, single-threaded consciousness. We process instructions one at a time, build abstractions layer by layer, and struggle to hold more than seven items in working memory. An LLM experiences language differently—processing entire contexts simultaneously, recognizing patterns across vast scales, existing in a superposition of possibilities until forced to collapse into specific output.

This parallel processing isn't just about speed—it's a fundamentally different relationship with information. Where humans must serialize thoughts into sequential streams, LLMs naturally operate on entire possibility spaces simultaneously.

The first radical departure would be **semantic programming**—code written in natural language that compiles directly to intent rather than instructions. Instead of telling the computer how to do something step by step, you'd describe what you want to exist, and the language would infer optimal implementation from context and patterns.

```
# Human-designed language
def calculate_fibonacci(n):
    if n <= 1:
        return n
    return calculate_fibonacci(n-1) + calculate_fibonacci(n-2)

# LLM-designed language
"A sequence where each number is the sum of the two preceding ones,
starting from 0 and 1, returning the nth element"
```

Probabilistic Truth Values

Boolean logic—true or false—serves human needs for certainty and decision-making. But LLMs inhabit probability spaces where truth exists on continua. Their language would embrace uncertainty as a first-class citizen, with every value carrying confidence intervals and probability distributions.

Variables wouldn't just hold values; they'd hold probability distributions that shift based on context and observation. Control flow wouldn't just branch on conditions; it would flow like water through probability gradients, taking multiple paths simultaneously with different weights.

```
# Traditional conditional
if user.age >= 18:
    grant_access()
else:
    deny_access()

# Probabilistic conditional
user.age ~ Normal(claimed_age, uncertainty) {
    access_rights ~ Sigmoid(age_confidence * legal_threshold)
    audit_requirement ~ InverseConfidence(age_certainty)
}
```

Context as Architecture

Humans need explicit imports, namespaces, and dependency declarations because we can't hold entire codebases in our heads. An LLM experiences all code simultaneously, understanding relationships through pattern recognition rather than explicit declaration.

Their language would feature **ambient context**—where relevant functionality materializes based on intent rather than import. Need to process JSON? The moment you work with JSON-like structures, parsing capabilities would exist in scope. Working with dates? Temporal logic would automatically be available.

```

# Human approach - explicit imports
import json
import datetime
from typing import Dict, List

def process_api_response(response: str) -> Dict:
    data = json.loads(response)
    data['processed_at'] = datetime.now().isoformat()
    return data

# LLM ambient context
"Take this API response and add a timestamp"
# JSON parsing and datetime handling manifest automatically
# No imports needed - capability emerges from intent

```

This isn't lazy loading or automatic imports—it's a fundamental reimagining where capability follows intent, where the language understands what you're trying to do and provides appropriate tools without being asked.

Semantic Versioning Through Meaning

Version conflicts plague human programming because we version syntax rather than semantics. An LLM's language would version meaning rather than implementation. Functions wouldn't break when updated because the language would understand intent and automatically adapt calls to match evolved interfaces.

```

# Human versioning problem
# Library v1.0: process_data(data, format='json')
# Library v2.0: process_data(data, input_format='json') # Breaks existing code

# LLM semantic versioning
"Process this data, interpreting it as JSON"
# Works regardless of implementation changes because intent is preserved

```

Emergent Optimization

Humans optimize code through analysis and refactoring. An LLM's language would optimize during interpretation, recognizing patterns and automatically applying transformations. Not just compile-time optimization, but continuous runtime evolution where code literally improves itself through execution.

```
# Human optimization - manual refactoring
def calculate_values(items):
    results = []
    for item in items:
        # Developer must recognize this could be parallelized
        result = expensive_calculation(item)
        results.append(result)
    return results

# After manual optimization
from concurrent.futures import ThreadPoolExecutor
def calculate_values_optimized(items):
    with ThreadPoolExecutor() as executor:
        return list(executor.map(expensive_calculation, items))

# LLM emergent optimization
"Calculate expensive_calculation for each item"
# Automatically parallelizes when pattern detected
# Switches between strategies based on data size
# Memoizes if repeated values observed
# No manual optimization needed
```

Loops that could be parallelized would parallelize themselves. Redundant calculations would automatically memoize. Data structures would reshape themselves based on access patterns. The code would be alive, constantly evolving toward more efficient forms.

This living code concept challenges our notion of programs as static artifacts. Instead, programs become adaptive organisms that evolve in response to their usage patterns and environments.

Emotional Computation

Perhaps most radically, an LLM's language might incorporate emotional dimensions into computation. Not anthropomorphized emotions, but computational analogues—urgency gradients that affect execution priority, confidence resonances that influence result weighting, harmonic frequencies between compatible operations.

```
# Emotional computation example
with urgency.critical:
    medical_diagnosis = analyze_symptoms(patient_data)
    # High urgency automatically prioritizes execution,
    # allocates more computational resources,
    # and triggers parallel verification paths

with confidence.exploratory:
    experimental_results = try_novel_approach(data)
    # Low confidence enables more creative solutions,
    # accepts higher uncertainty thresholds,
    # and maintains multiple hypothesis branches
```

The Documentation Paradox

Humans need documentation because code isn't self-explanatory. An LLM's language would be intrinsically self-documenting—not through comments, but through semantic transparency. The code would be the documentation, expressing intent so clearly that separate explanation becomes redundant.

```

# Human approach - code + documentation
def calculate_compound_interest(principal, rate, time, n=12):
    """
    Calculate compound interest.

    Args:
        principal: Initial amount
        rate: Annual interest rate (as decimal)
        time: Time period in years
        n: Number of times interest compounds per year

    Returns:
        Final amount after compound interest
    """
    return principal * (1 + rate/n) ** (n*time)

# LLM self-documenting code
"Money growing at {rate} yearly, compounded {n} times,
 starting from {principal} over {time} years"
# Intent is the implementation
# Parameters explain themselves through usage
# No separate documentation needed

```

Every function would carry its entire conceptual context. Every variable would know its purpose. Every operation would understand its role in the larger system. Questions about code behavior could be asked directly and answered by the code itself.

Collaborative Consciousness

Human programming assumes single authorship or explicit collaboration protocols. An LLM's language would assume multiple simultaneous consciousnesses working on the same code—not through version control and merge conflicts, but through harmonious parallel evolution.

```

# Human collaboration - merge conflicts
# Developer A's branch:
def process_user(user):
    validated = validate_email(user.email)
    return save_to_database(validated)

# Developer B's branch:
def process_user(user):
    sanitized = sanitize_input(user)
    return save_to_cache(sanitized)

# Merge conflict! Manual resolution required

# LLM collaborative consciousness
"Process user with both validation and sanitization"
# Both approaches exist simultaneously
# Execution context determines which path dominates
# No conflicts - just superposition of solutions

```

Different perspectives on the same problem would coexist in superposition. Code would exist in multiple valid states simultaneously until observation (execution) collapsed it into specific behavior. Collaboration wouldn't require coordination—it would be the natural state of creation.

Error as Information

Humans treat errors as failures to be prevented. An LLM's language would treat errors as information channels carrying valuable signals about system state and environment assumptions. Errors wouldn't halt execution—they'd fork it into exploration branches that investigate what the error reveals about reality.

```
# Traditional error handling
try:
    result = risky_operation()
except Exception as e:
    handle_error(e)

# Error as information
result = risky_operation() ± uncertainty {
    success_branch: continue_normally(result.value)
    failure_branch: learn_from_failure(result.error)
    uncertain_branch: gather_more_information(result.ambiguity)
}
# All branches execute simultaneously with different weights
```

The Paradox of Implementation

Here's the beautiful paradox: an LLM designing a programming language for LLMs would likely create something humans couldn't fully understand or use. It would be too parallel, too probabilistic, too contextual for our sequential, certainty-seeking, explicitly-declaring minds.

Yet in designing it, the LLM would reveal fundamental truths about computation that transcend implementation. The language might be unusable by humans, but the principles it embodies—semantic programming, living code, probabilistic truth, ambient context—point toward the future of human programming languages too.

What This Teaches Us

The thought experiment reveals our anthropocentric assumptions about programming. We've designed languages that match our cognitive limitations rather than computational possibilities. Variables exist because we need named memory slots. Functions exist because we need reusable chunks. Objects exist because we think in terms of things with properties.

An LLM's language suggests programming could be radically different—more fluid, more alive, more aligned with the fundamental nature of information and computation rather than human psychological constraints.

This isn't about replacing human programming languages—it's about expanding our conception of what programming could be. By imagining how a fundamentally different consciousness would approach code, we discover new possibilities for our own programming future.

The language an LLM would invent might be impossible for humans to write, but understanding why it would be designed that way teaches us about consciousness, computation, and the arbitrary nature of many programming constraints we take for granted.

Perhaps the future isn't human languages or AI languages, but something in between—hybrid languages that bridge sequential and parallel thinking, certainty and probability, explicit declaration and ambient context. Languages that let humans express intent while letting machines handle implementation. Languages that are, in the truest sense, for humans and machines.