SPELL

AI-Native Programming Language

AUTHOR

Elio Maurizi Santino

research@santino.world

2025-12-05

SANTINO RESEARCH

ABSTRACT

We introduce a programming paradigm optimized for the cognitive architecture of Large Language Models, rather than human authors.

Traditional languages rely on sequential reasoning and implicit context: **abstractions that conflict with the pattern-completion nature of LLMs**.

SPELL is the first AI-Native language. By aligning syntax with the generation mechanism of the model, we demonstrate that the optimal abstraction for AI is fundamentally different from the one for humans.

1. Hardware

A processor executes instructions. The execution model appears sequential: **fetch, decode, execute**. But modern processors do not operate this way internally.

Out-of-order execution reorders instructions based on data dependencies. If instruction B does not depend on instruction A, they execute in parallel. The processor builds a dependency graph and executes nodes when their inputs are ready.

Multiple cores extend this. Pipelines overlap instruction phases. Branch prediction speculates execution paths. The hardware is fundamentally parallel and data-driven.

At the hardware level, computation is a graph of data dependencies. Sequence is an illusion maintained for the programmer.

2. Abstraction

Programming languages abstract over hardware. The abstraction is designed for the author: **the entity writing the code**.

Human authors think sequentially. We track state mentally. We simulate execution step by step. Languages designed for humans reflect this: statements execute in order, variables mutate over time, control flows through branches.

```
total = 0
for item in items:
    if item.value > 20:
        total += item.value
```

This abstraction hides the parallel, data-driven reality of the hardware. It presents a sequential narrative. The human simulates the narrative to understand the program.

The abstraction is useful because humans reason narratively. It is not a property of computation itself.

3. New Author

LLMs generate code through pattern completion. Given a sequence of tokens, the model predicts the next token based on statistical patterns in training data.

The mechanism is not sequential reasoning. There is no state tracking. There is no mental simulation of execution. There is pattern completion over tokens.

When an LLM generates imperative code, it produces tokens that match patterns from training. Whether those tokens form a correct program depends on whether correctness was consistently represented in those patterns.

The sequential abstraction was designed for human cognition. LLMs do not share that cognition. They have a different mechanism. The abstraction does not fit.

4. Different Abstraction

If the author changes, the optimal abstraction changes. SPELL is an abstraction designed for pattern completion.

4.1 Explicit Dependencies

Computation is a graph of nodes. Each node declares its inputs explicitly. Dependencies are structure, not sequence.

```
{
  "filtered": {
    "op": "Filter",
    "list": { "ref": "items", "type": "Array" },
    "apply_op": { "literal": "Gt", "type": "String" },
    "params": { "literal": {"b": 20}, "type": "Any" },
    "returns": "Array"
  },
  "total": {
    "op": "Reduce",
    "list": { "ref": "filtered", "type": "Array" },
    "apply_op": { "literal": "Add", "type": "String" },
    "initial": { "literal": 0, "type": "Number" },
    "returns": "Number"
  }
}
```

Each node has a name (`filtered` , `total`) and declares:

- **op**: The operation to perform
- **inputs**: References to other nodes (`ref`) or literal values (`literal`), each with explicit type
- **returns**: The output type of this node

For higher-order operations like `Filter` and `Reduce` , `apply_op` specifies which operation to apply, and `params` provides additional arguments.

The LLM does not simulate execution. It generates structure. The structure defines the computation.

4.2 Explicit Types

Every value has an explicit type annotation. Types are not inferred: **they are stated**.

```
"a": { "ref": "x", "type": "Number" }
"b": { "literal": 5, "type": "Number" }
```

The type is adjacent to the value. The LLM does not need to track type information across the program. Local tokens contain complete information.

4.3 Structured Format

JSON. Why JSON?

- **Standard format**: Extensively represented in LLM training data. The model has seen millions of valid JSON structures.
- **Native output**: Like Markdown, JSON is a format LLMs produce reliably without special prompting.
- **Hierarchical structure**: JSON objects naturally model nested computations. Structure mirrors logic.
- **Rigid syntax**: Well-formedness is determined by token structure alone. Brackets must match. Commas must appear. The syntax is unambiguous.

5. The Model

SPELL fits both ends of the computation stack.

Hardware: The dependency graph model is closer to how processors actually execute. No sequential illusion to maintain. No state mutation to track. The abstraction exposes the computational reality rather than hiding it.

Author: Pattern completion operates on structure. Explicit dependencies, explicit types, rigid syntax. The model generates tokens that form valid programs through pattern matching alone.

The sequential abstraction served one type of author. A different author benefits from a different abstraction.

6. Operations

The following operations are implemented in SPELL v0.1 (pre-alpha). This minimal set is sufficient for data-complete computation: any transformation over data structures can be expressed through composition of these primitives.

Operation	Inputs	Output
Const	value	Value
Add, Sub, Mul, Div	a, b	Number
Eq, Gt, Lt	a, b	Boolean
Map, Filter	list, apply_op, params	Array
Reduce	list, apply_op, initial	Value
Switch	cond, true, false	Value
Len	list	Number

With `Map` , `Filter` , `Reduce` , and `Switch` , the language is expressively complete for data transformations.

7. Observations

Tested across LLM systems without documentation or examples provided.

Models correctly interpret program semantics. They trace execution through the dependency graph. They generate valid programs from natural language specifications.

Models propose optimizations consistent with program semantics. They suggest language extensions that follow existing design patterns.

The code is self-explanatory. The model reasons about it effectively.

8. Applications

LLM-authored programs. Code generated by LLMs in a format LLMs understand natively. Generation and comprehension use the same mechanism.

Language co-design. LLMs propose language extensions. The language evolves through the same pattern completion that generates programs. Author and medium develop together.

Tool orchestration. LLMs coordinate tools, APIs, and workflows through explicit dependency graphs. The orchestration logic is inspectable and modifiable by both humans and machines.

Program optimization. LLMs optimize SPELL programs by reasoning about the dependency structure. Optimization operates on the same representation as generation.

9. Conclusion

Programming languages are abstractions. Abstractions are designed for authors. When the author was human, the abstraction was sequential, implicit, narrative.

The author is changing. LLMs generate code through pattern completion. They do not track state. They do not simulate execution. They complete patterns.

SPELL is an abstraction for this mechanism: explicit dependencies, explicit types, structured format. The abstraction fits the author.