The Metaprogrammer's Journey

*Twenty Years of Code Generation,
Seven Languages, One Answer*

John Grillo

Draft — February 2026

QUOTE:I put the grammar in Lisp. Then it was just a dolist.

Preface

I spent 20 years chasing metaprogramming. Always wanting something better, always searching for the next level of code generation. Nineteen of those years were in Ruby. The last year was everything else.

This book is the record of that search. Each chapter is a language, a system I built, a wall I hit, and what the wall taught me. The code is real. The line counts are real. The frustration was real.

The answer, when I finally found it, had been sitting there since 1958. It took me 20 years to understand why.

The s-expression grammar used in the YAML project is derived from the YamlReference Haskell package by Oren Ben-Kiki, co-creator of the YAML specification. His reference implementation translated the YAML 1.2 spec into precise, machine-verified BNF productions — without which the final chapter of this book would not exist.

AI assistance (Claude, Anthropic) was used throughout the later projects — building language walkers in Prag, the conversion from Haskell BNF to s-expressions, and mechanical code generation. The architecture, the journey between languages, the recognition of each wall, and the final insight that s-expressions were the right representation are original work.

A note on AI and code generation: AI could build walkers. It could write walker 1 perfectly, walker 5 competently, and by walker 15 it would start drifting — inconsistent naming, forgotten conventions, patterns that didn't match the earlier ones. By walker 25, I was spending as much time fixing drift as I would writing by hand. AI made Level 2 more comfortable. It didn't remove the wall.

Worse: when the API changed — a new field in the AST, a renamed method — every walker broke. All 30 of them. Each one had to be regenerated from its source, each in a separate conversation, each with its own context, each with its own opportunity to drift. One API change meant 30 regenerations and 30 reviews.

The projector eliminated this entirely. One grammar change, one run, 18 parsers updated simultaneously. They cannot drift because they are not independent artifacts. They are projections of the same source. That is the difference between code generation and projection — and the AI experience is what made it visible.

Chapter 1

Ruby — Nineteen Years of Templates

The Beginning

Ruby was the first language where I felt the pull of metaprogramming. ERB templates, string interpolation, method_missing, define_method, open classes. Everything was an object. Everything was mutable. Everything felt possible.

The first code generators were simple. A template with holes. Fill in the struct name, fill in the field names, emit the boilerplate. It worked. I stopped writing the same code by hand. That felt like power.

The Plateau

But templates are strings. The generator doesn't know what it's generating. It doesn't know that the thing it's emitting is a class with fields and methods. It knows it's concatenating text with variables interpolated.

When the output needed to change — a different naming convention, a new field type, a conditional include — the template got another if statement. Another branch. Another special case. The templates became programs, but programs written in strings with no structure, no type checking, no way to ask what they contained.

This was Level 1. Programs that emit text. The text happens to be code. But to the generator it's just characters.

Nineteen Years

I stayed in Ruby for nineteen years. Not because I didn't see the limitations. Because Ruby kept offering new tricks. Metaprogramming. DSLs. Blocks and procs. Method objects. The eigenclass. Each one felt like it might be the thing that broke through to the next level.

None of them did. They were all runtime tricks. Powerful, expressive, beautiful runtime tricks. But the code was still opaque. You could call a method. You couldn't read a method as data. You could define a class dynamically. You couldn't walk a class definition as a tree.

Nineteen years. Hundreds of generators. Thousands of templates. Real systems that shipped and ran and made money. And every one of them was Level 1.

The Lesson

Ruby taught me what metaprogramming feels like — and what it isn't. It feels like power. It feels like abstraction. It feels like you're writing programs that write programs.

But you're not. You're writing programs that write strings. The difference is everything.

Somewhere around year nineteen, I started looking for something else. I didn't know what. I just knew that templates weren't it. That string concatenation wasn't it. That runtime tricks weren't it.

I needed the code to be data. I didn't have those words yet. But that was what I was looking for.

Chapter 2

C++ — The Richest Type System That Can't Read Itself

The Idea

C++ has the richest type system in mainstream use. Real types. Real constraints. Concepts, templates, constexpr, static_assert. If you want a schema language with teeth, C++ structs already are one.

The idea was simple: make C++ structs the source of truth. Not a dumbed-down YAML schema. Not a JSON config. The actual structs, with their actual types, as the single definition from which everything else is generated. CSV serializers. SQL builders. JSON emitters. All of it derived from the struct metadata. One source of truth, universal output.

It was the right goal. C++ was the wrong host.

The Architecture

The system had three layers. First, the structs — plain C++ structs, the source of truth. Second, the metatuples — a template metaprogramming framework that attached metadata to struct fields at compile time. Field names, types, attributes like SqlColumn, CsvColumn, PrimaryKey. Third, the builders — one per output format, each walking the metatuples with std::apply and fold expressions, dispatching on types with if constexpr.

The Crowbar: libclang

C++ can't read itself. A C++ program can't look at a struct definition and list its fields. There's no read. There's no runtime reflection. The type information is erased at compile time.

So I brought in libclang — a full C compiler frontend — to parse C++ headers into an AST that a program could walk.

692 lines. A full C++ program that links against libclang, walks the AST, extracts field names and types and access specifiers and annotations, handles anonymous nested structs, resolves qualified names, and emits the metatuple declarations.

692 lines to do what Lisp's read does in one function call.

The Wall

The system generated output. It didn't generate generators.

I couldn't take the CSV builder and emit a standalone Python script that does the same thing. The builders were compiled C++. The templates were resolved. The functions were opaque. The structure was gone.

Every new output format meant writing a new builder in C++, recompiling, and shipping the whole toolchain. The struct metadata was the single source of truth — but it was trapped inside C++. Nothing outside the C++ ecosystem could use it.

```
field.h 373 lines — Field<> template, attributes, validation
meta.h 1,298 lines — serialization framework, Builder API
meta_lang.h 813 lines — language-level metaprogramming
metafront2.cpp 691 lines — libclang AST parser
meta_csv.h 460 lines — CSV builder
meta_db.h 209 lines — SQL generation
Total 4,304 lines of C++ to reach Level 2
```

The Lesson

C++ has the best type system for describing data. It has one of the worst for reading those descriptions programmatically. The struct knows everything about itself at compile time. Nothing can ask it at runtime. And nothing outside C++ can ask it at all without a 692-line program that links against a compiler frontend.

I wasn't looking for Lisp yet. But I was looking for what Lisp already had.

Chapter 3

D — Level 3 for the First Time

The Idea

D has compile-time reflection built into the language. FieldNameTuple gives you the field names of any struct. __traits(getMember, T, fieldName) gives you the field type. static foreach iterates at compile time. No libclang. No external parser. The language can read its own structs natively.

So I built a universal code generator. Define your structs in D. Annotate them with UDAs. Press a button. Get working C++, Java, Python, Rust, TypeScript, Go, Haskell, Zig, Ruby, SQL, CSV, XML, YAML — thirteen languages from one definition.

What Worked

It actually generated code. Real code. You defined a struct in D and the system emitted a C++ class with std::string, constructors, getters, const references. A Python dataclass. A Rust struct with pub fields. A TypeScript interface. Thirteen outputs.

This was Level 3. Define once, generate everywhere. D's compile-time reflection made the pipeline trivial.

The Wall

Each language module was hand-written. The C++ generator was 942 lines. The Ruby generator was 212 lines. The Haskell generator was 168 lines. Adding a fourteenth language meant writing another module.

Level 3 for output. Level 2 for the generators themselves. You could generate code in any language. You couldn't generate generators.

```
types.d 751 lines — language registry, type system
cpp.d 942 lines — C++ code generator
generator.d 419 lines — universal engine
Total 2,892 lines of D (shown) + 13 language modules
```

The Lesson

D proved that compile-time reflection eliminates the crowbar. But it also revealed the next wall. Reading the spec is necessary but not sufficient. You also need to read the generators. You need the code that generates code to itself be data — walkable, translatable, emittable.

D can read structs. It can't read functions. The output was universal. The generators were trapped.

Chapter 4

Rust — Three Attempts at the Same Wall

Three Attempts

I tried three times. First: pr-rust.rs, 115 lines, using syn to parse Rust source and emit s-expressions. The bridge from Rust to Lisp. But it needed syn — a full Rust parser library — to read what the language wrote. That's the libclang story again.

Second: about 170 lines, emitting C++ directly with a type registry. One language in, one language out. The D story with fewer languages.

Third: reflect.rs, 550 lines, a homegrown reflection system using Rust macros. One macro call gave you full serialization and deserialization. But it was 550 lines to get what D gave you with FieldNameTuple. And it was still Level 2.

The Tell

The s-expression bridge was a tell. I was converting Rust to s-expressions so something else could read it. I was building a pipeline that ended in parenthesized lists because that was the format that could be universally consumed.

I wasn't looking at Lisp yet. But I was building half of it by hand.

```
pr-rust.rs 115 lines — syn-based, emits s-expressions
gen-cpp.rs 170 lines — Rust-to-C++ generator
reflect.rs 550 lines — macro-based reflection
Total 835 lines of Rust (plus ~40,000 lines of syn)
```

Chapter 5

Go — The Shortest Bridge

The Bridge

Go has a built-in AST parser. Not a third-party library. It ships with the language: go/ast, go/parser, go/token. Part of the standard library.

165 lines. Parse a Go source file. Walk the AST. Find every struct. Convert each field's type to a portable representation. Emit JSON. The shortest bridge of any language.

A second attempt — 430 lines — built a three-stage pipeline: Go AST to C++ AST to C++ code. The intermediate AST was data. You could walk it. But the walkers were Go functions. To add a new output language, you wrote a new visitor in Go and recompiled.

The Lesson

Even when the parser is free — built into the standard library — reading source code is still a one-way trip. You parse into an AST, you extract data, you emit something. But the generator doesn't generate generators.

Every language kept building the same pipeline: parse, extract, emit. And every language kept stopping at emit.

```
pr-go.go 165 lines — Go struct parser, emits JSON
gen-cpp.go 430 lines — Go-to-C++ with intermediate AST
Total 595 lines of Go
```

Chapter 6

Python — Everything Except the Thing You Need

The Seduction

Python has everything. ast.parse gives you the syntax tree. typing.get_type_hints gives you annotations. inspect gives you runtime reflection. dataclasses.fields gives you field metadata. Metaclasses let you intercept class creation. Decorators let you wrap anything. eval and exec let you run strings as code.

More ways to read itself than almost any language. And it reads beautifully. A Python class with type hints is cleaner than a C++ struct with metatuples. The parser is built in. No libclang, no syn, no external toolchain.

The Parser

248 lines. Parse a Python file with ast.parse. Walk the class definitions. Extract field names from annotated assignments. Convert type hints — List[str], Optional[int], Dict[str, float] — into reified types. Emit JSON.

The type mapping was familiar by now. Python's 'int' becomes 'i64'. 'str' becomes 'String'. 'List[str]' becomes 'Vec[String]'. The same registry, written for the sixth time.

```
def annotation_to_type_string(annotation: ast.expr) -> str:
if isinstance(annotation, ast.Name):
return annotation.id
if isinstance(annotation, ast.Subscript):
base = annotation.value
...
```

ast.Subscript. ast.Name. isinstance checks. You're pattern-matching on syntax nodes to recover what the programmer already knew when they wrote the type.

The Trap

Python is the most seductive trap of all. It has ast.parse and inspect and metaclasses and eval and exec and decorators and dataclasses. Every one of those feels like it should be enough to get to Level 3. All the pieces are there.

But ast.parse gives you syntax nodes, not reified types. ast.Subscript tells you the source said List[str]. It doesn't give you a collection-of-strings. You write the type registry again. Same as C++, same as Rust, same as Go.

And the generators are still Python functions. f"def {name}(self):" — string concatenation with nice syntax. You can call them. You can't walk them. You can't project them into Rust. They're f-strings.

Level 1 with better formatting.

The Lesson

Every other language hit the wall and you could feel it. C++ needed 692 lines of libclang. Rust needed syn. The friction told you something was wrong.

Python doesn't have friction. It has everything you ask for. Parse? Built in. Reflect? Built in. Eval? Built in. Every question has an answer. Except the one that matters: can the generator generate generators?

No. The code is still opaque. The functions are still compiled. The generators are still strings. Same wall. Nicest furniture.

```
pr-python.py 248 lines — ast.parse-based parser, emits JSON
```

Chapter 7

C# — The Compiler as a Library

Roslyn

C# has Roslyn — the C# compiler exposed as a library. Not a separate parser. The actual compiler, available as a NuGet package. If libclang was a crowbar, Roslyn was the whole toolbox.

About 200 lines. Same bridge pattern. Parse, extract, emit JSON. It handled generics, nullable types, properties and fields. Roslyn is polished. LINQ made the tree walking concise.

Eight Languages, Eight Bridges

By this point the wall was predictable. The program read C# structs and emitted JSON. It couldn't read itself. Roslyn was the most sophisticated parser yet — the actual compiler — and it still produced a one-way trip from source to data.

The reading problem was solved seven different ways. The generating problem wasn't solved once.

Every parser — libclang, syn, go/parser, ast.parse, Roslyn — gave you syntax, not types. Every program needed a hand-written type registry to bridge from syntax to semantics.

I had tried every mainstream approach. I needed a language where there was no bridge. Where the code was already data.

```
pr-csharp.cs 200 lines — Roslyn-based parser, emits JSON
```

Chapter 8

Prag — The Universal Bridge

The Diversion

Somewhere along the way, I stopped looking for Level 3 and started building the best Level 2 I could.

Prag was a universal schema translator. Any input format to any output language, through a shared AST. 17 input parsers: Avro, Cap'n Proto, C#, FlatBuffers, F#, Go, GraphQL, C/C++ headers, Haskell, JSON Schema, Access databases, OCaml, OpenAPI, Protobuf, Thrift, TypeScript. 30+ output walkers. 500+ possible transformations.

The Reified AST

The shared AST was the insight every previous chapter was converging toward. Not syntax nodes — reified types. ReifiedTypeId::Bool. ReifiedTypeId::String. When the Cap'n Proto parser saw 'Text' it produced ReifiedTypeId::String. When the Go parser saw 'string' it produced the same. Every parser converged to meaning.

The type mapping was done once, in each parser, and the walkers worked with meaning. No hand-written type registry in every walker.

The Wall

Every parser was a hand-written C++ class with its own lexer, its own token types, its own recursive descent. Every walker was a hand-written C++ header. 30+ walker files.

Adding walker 31 meant writing another C++ header. Fixing a pattern across all walkers meant editing 30+ files. The scale didn't solve the problem. It multiplied the hand-written code.

Prag was the most complete Level 2 system possible. 17 parsers. 30+ walkers. 500+ transformations. A shared reified AST. And every parser and every walker was compiled C++ that couldn't read itself, couldn't be emitted in another language, couldn't generate generators.

I had 30+ walkers that proved the pattern worked. And I couldn't generate walker 31 from the pattern of the first 30.

The Lesson

The bridge was universal. The bridge was still compiled code. I needed a language where the bridge was data.

Chapter 9

Common Lisp — A Dolist

The Path

Nineteen years of Ruby. Then one year: C++, D, Rust, Go, Python, C#, Prag. Each one tried and measured and found wanting. Each one teaching me the next thing to look for.

And then I saw it. The whole thing. All at once.

The Representation

The YAML 1.2 specification defines 211 grammar productions. They already existed as a Haskell reference implementation — Oren Ben-Kiki's YamlReference package on Hackage. Precise, machine-verified, complete. But trapped in Haskell's operator overloading. Beautiful and unreachable.

I converted the 211 productions to s-expressions. Every rule became a list. Every alternation, every sequence, every parameterized dispatch, every binding form — data. Walkable data.

```
(Rule 28 b-break
(Alt (Seq (Ref b-carriage-return) (Ref b-line-feed))
(Ref b-carriage-return)
(Ref b-line-feed)))
```

Same rule as the spec. Same rule as the Haskell. But now it's data a program can walk.

The Projector

The projector reads the s-expression grammar. For each rule, it walks the tree and emits the parser in whatever target language the target spec defines. Alternation becomes alternation. Sequence becomes sequence. Reference becomes function call. The projector doesn't interpret. It translates.

In 1971, Yoshihiko Futamura described how partially evaluating an interpreter with respect to a program produces a compiled version of that program. The grammar is the program. The PEG combinator framework is the interpreter. The projector specializes one against the other to produce standalone parsers.

Write a target spec — 300 to 500 lines that tells the projector how the language expresses functions, closures, sequences, and alternatives. The projector does the rest. In seconds.

The Results

One grammar. Eighteen languages. Two weeks.

Go, Rust, Python, C++, Java, Kotlin, C#, F#, Haskell, Swift, Objective-C, Zig, OCaml, Erlang, Lua, Bash, PowerShell, x86-64 assembly.

308 out of 308 tests passing. Every parser. The YAML Test Suite.

The projector does exactly what the spec says. No workarounds, no interpretation. So when the spec has a bug, the parser has a bug. That's how I found a bug in Rule 78 of the YAML 1.2 specification — present since 2009. Every hand-written parser silently works around it without noticing. A projector can't. Spec bugs have nowhere to hide.

Two Weeks

Twenty years to see the path. Two weeks to walk it.

That's how you know the representation was right. When it's right, the work collapses. Eighteen languages and 308 tests wasn't grinding. It was falling downhill.

And the twenty years weren't wasted. They were the cost of being able to see the path. Without Ruby I wouldn't know why metaprogramming isn't enough. Without C++ I wouldn't know why rich types aren't enough. Without D I wouldn't know why compile-time reflection isn't enough. Without Rust and Go and Python and C# I wouldn't know why better parsers aren't enough. Without Prag I wouldn't know why scale isn't enough.

Each failure was a wall that narrowed the search space until there was only one direction left.

The Answer

Lisp is the discovery that code and data are the same structure. Everything else — the REPL, the macros, eval, the whole 67-year ecosystem — is a consequence of that one discovery.

If you use Lisp as a runtime but you don't use s-expressions as your representation for the problem you're trying to solve, you're using the consequence without understanding the cause.

I used s-expressions to represent. The projector fell out. The parsers fell out. The tests fell out.

It was just a dolist.

The repository is at github.com/johnagrillo62/yaml-project

The grammar is there. The projector is there. The tests pass.

Do what you want with it.