The Next Programming Language

Why It Won't Be Written by Humans

Ajay Malik

A free sample

This excerpt is the Preface and Chapter 1 of *The Next Programming Language – Why It Won't Be Written by Humans* by Ajay Malik.

The complete book is 27 chapters, four appendices and eight reproducible laboratory exercises, and is available in paperback and for Kindle. ISBN 979-8-9877684-1-9.

Preface

This book is about a programming language that does not exist yet, and about why it has to.

When a model writes a program for you, it writes it in Python. Or TypeScript, or Java, or Go. There was never a clean-sheet decision that these were the right targets for a machine author. They are heavily represented in the training distribution, and unsurprisingly they are what tends to come out.

This book is about what should come out instead.

The answer turns on one property, and it is not the one most people expect. The usual assumption is that a language for machine authors should be easier to *generate* — terser, more regular, closer to what a model already produces. That is the wrong half of the problem. Generation is the part that already works. What does not work is what happens after the code exists and turns out to be wrong: the faulty assumption was never written down anywhere, the representation gives you nothing to point at, and so the cheapest available fix is to generate the whole thing again and hope.

The property that matters is the ability to correct a program **deterministically after it has been produced** — to name the thing that is wrong, bound how much of the program may change in response, and confirm that the change was enough. Nearly every specific decision in the second half of this book falls out of taking that one requirement seriously.

The argument takes some setting up, because the obvious objection — *models are getting better at Python, so what's the problem?* — is answering a question I am not asking. The question is not whether a model can write good Python. It plainly can. The question is what the intermediate language should look like when the thing doing the producing is a model rather than a compiler — and whether the answer is a language we designed for ourselves in 1991.

The first half of this book is the argument. The second half is the language.

Who this is for

Anyone who writes software, reviews it, buys it, or is responsible for it working. You should be comfortable reading a program. You do not need to know how a compiler works — Part III explains what you need, and no more than that.

If you have never seen assembly, you will by page 30. If you write compilers for a living, Part III will be familiar and Part IV will not.

What you need

A machine with Python 3 and Clang on it. Every program in this book runs, and the labs are small enough to read in one sitting. Nothing in here is pseudocode.

On method

Part I is history, and it is not decoration. Programming has been through this twice before, and the arguments that were made against each transition are on the record — made by intelligent people, for defensible reasons, and mostly correct at the time. Knowing how those arguments went is the fastest way to see the shape of this one.

Part II demonstrates. Each cost is shown on a worked example small enough to check by hand and reproduce from the labs — an existence proof that the cost is real and has the shape I claim, not a survey establishing how large it is across the population of programs. Where a number is owed and has not been taken, the chapter says so rather than estimating one.

Part V is a working language, and the labs compile and run. Nothing in this book is pseudocode — though Appendix A.12 is explicit about the fragment of computation the language covers today, which is smaller than a reader might assume from the argument around it.

And the book ends by listing what would prove it wrong — eight observations, each with the experiment that would produce it, including the two measurements the argument openly owes and has not taken. A design book that cannot say what would refute it is not making a claim, it is expressing a preference.

If you read one chapter

Chapter 21, on diagnostics. It specifies a structured record — a stable code, the fragment at fault, the obligation that was not met, whether it is false or merely unproven, the finite set of legal repairs, and how much of the program may be touched in response.

It is the part of this book with no dependency on the rest of it. Anyone building a coding agent can adopt that record next week, against Python or Rust or anything else, without writing a line of the language in Part V. It is also the cheapest item on the falsification list in Chapter 27 — emit both forms from the same compiler and compare the repairs. If the argument in this book is wrong everywhere else, that chapter still stands on its own.

A note on the language

The language in Part V is real. It parses, it verifies, it compiles, and it runs — within the fragment of computation Appendix A.12 sets out, which is smaller than the argument around it might suggest. It is small deliberately, because a language you can read in an afternoon is a language you can argue with. Everything about it is designed to be disagreed with in specifics and, I hope, agreed with in shape.

---

1. The Author Changed

Programming has lived through three broad language eras in roughly eighty years: machine code, assembly, and high-level languages. Twice, authorship moved upward. Both times for the same reason: the author changed.

We wrote machine code by hand until the programs got too big to hold in our heads. So we invented assembly — `mov` instead of `b8` — because people remember words, not numbers. Then we invented high-level languages: names, types, modules, readability. Those features do real engineering work — types enable analysis, modules enable separate compilation — but the criteria that selected them were overwhelmingly about what a person could hold in their head.

And look what happened underneath. Almost nobody hand-writes machine code any more. Most assembly is no longer written by hand. They didn't die — they stopped being written and started being emitted.

Today AI writes Python. But Python didn't win because it's the best language. It won because it was the best language *for a human author*. The author has just changed. So the language will change too.

That's the fourth one. This book is about what it should be.

The same function, four times

Here is a function that returns the number 42, written four ways.

```
b8 2a 00 00 00 c3
```

```
mov eax, 42
ret
```

```
int answer(void) { return 42; }
```

```
Write a function that returns 42.
```

Only the first one *is* bytes. The rest are descriptions of bytes, and they get less specific as you go down.

Compile that C for x86-64 with optimization on and the frame pointer omitted and you get exactly `b8 2a 00 00 00 c3` back. Leave the frame pointer in — the default on macOS — and you get five more bytes of prologue and epilogue around it. Compile it on the Apple Silicon laptop this was written on and you get eight entirely different bytes:

```
40 05 80 52 c0 03 5f d6
```

Same C. Same meaning. Different machine, different bytes, and nothing in the source said which.

That is the whole progression in one example. As you move down the list you stop writing the bytes and start describing them — and you hand somebody else the decision about what they actually turn out to be. The English at the bottom hands over the most of all.

So isn't English the fourth language?

That is the obvious objection and it arrives on this page, so let me take it seriously rather than leave it until Part IV.

If the pattern is *humans keep moving up*, then the fourth level is the last line in that list. English. We are already writing it. Python is already the thing being emitted. Book over.

I think that is half right, and the half it gets wrong is the interesting half.

Look at what actually happens when you type that English sentence. Something reads it, decides what it means, chooses a strategy, and produces a program in a language you did not name. That is not a new level of programming language. It occupies the position a **compiler** occupies: it takes a description in one language and emits a program in another.

I am going to call it a **compiler-like producer** rather than a compiler, and the reason matters more than the pedantry suggests.

A compiler has two things this does not. It has *defined source semantics* — there is a specification saying what a C program means, independent of any implementation. And it has a *preservation obligation*: the output must mean what the input meant, and because the input's meaning is specified, that obligation can be stated, argued about, and in some cases proved. A miscompile is a bug with a definition.

English has neither. There is no specification of what *sort these records by date* means. Ascending or descending, stable or not, what happens to null dates, whether the sort must be deterministic across equal keys — none of that is in the sentence, and no amount of care in reading it will find what was never there. So the producer is not *preserving* a meaning. It is **choosing** one, from a space the request left open, and then committing to the choice silently by writing code that does one particular thing.

That distinction is not a weakness in the analogy. It is the most useful thing on this page, because it says exactly what the missing level has to do.

A compiler's intermediate representation exists to hold a program while the compiler decides how to lower it. What is needed here is that, plus a job no compiler IR has ever had: somewhere for the choices to be **written down**. If ambiguous intent is being turned into a specific program, then the point at which ambiguity becomes commitment is the point where a representation should be recording what was committed to — so it can be checked, disagreed with, and repaired without regenerating everything around it. Every previous compiler got its source semantics for free, from a standards document. This one has to manufacture them, one program at a time, and currently throws away the manufacturing record.

That is the language this book is about. Hold on to it; Chapter 15 builds directly on it, and it is why *intent* is the first thing AIR makes explicit.

And here is the thing about serious compilers. Nearly every optimising compiler has one or more intermediate representations — Part III is an inventory of them. A single-pass compiler can lower fairly directly, but the moment a compiler wants to *decide* anything it needs somewhere to hold the program while it thinks. The intermediate representation is where a compiler does its actual work, and choosing it is among the most consequential decisions in its design.

So: what does this one emit into?

Python. Or TypeScript, or Java. Not because anyone evaluated the options — because that is what was in the corpus.

To be precise about the claim: nobody knows what a model's internal representation looks like, and Python is not it. Python is the **target** — the representation the whole system commits its output to, the artifact everything downstream consumes. And we have built a new kind of code generator whose de facto target representation was not chosen through compiler design at all. It emerged from the training distribution. The thing it emerged from is a set of languages designed around the working memory of a primate.

That is the question this book is about, and notice that the previous three transitions never had to ask it. Every compiler before this one was designed by people who chose its IR deliberately, for reasons they could state. This is the first time the compiler arrived before its intermediate language did.

So the ladder in this chapter is real, and English is genuinely the level humans are moving to. But it is not the end of the story, because underneath that move a new machine appeared, and that machine needs a language of its own. Humans will not write it. That is the language this book is about.

Where everything sits

There are several moving pieces in the last few pages, and they are the ones the title depends on, so here they are in one place.

	AUTHORED BY A PERSON	EMITTED BY A MACHINE
era 1	machine code	—
era 2	assembly	machine code
era 3	high-level source	assembly, and below
now	English	high-level source, and below
		↑
		the only layer here that
		nobody designed

Read it as three language eras and two transitions between them. Each transition moved a person up one row, and handed the row they left to a machine.

What is happening now has the same shape. People are moving to English. The row they are leaving — high-level source — is passing into machine hands, and it is doing so *unchanged*, still carrying every design decision that was made for the reader who just walked away.

English is not the fourth language. English is where humans went. The fourth language is the one in the right-hand column of the bottom row: what a machine should emit, now that a machine is doing the emitting. Every previous entry in that column was designed on purpose, by people who could say why. This one was not designed at all.

Each level was built for its author

Machine code is where we started, not a transition. People wrote it directly — setting switches, punching cards, memorising opcodes — possible only because programs were small enough to

hold in one head. It is the baseline the two transitions moved away from, which matters for the arithmetic later: three levels, two moves, and the objection has a record against the moves.

Assembly arrived because human memory is bad at numbers and good at words. `b8` became `mov`. That is the entire innovation, and the machine did not care: it was perfectly happy with `b8`. The assembler bought the processor nothing and the programmer everything.

High-level languages arrived for the same reason, scaled up. A person can hold a hundred lines of assembly in their head, not a hundred thousand. So we invented names, so we would not track addresses. Functions, so we could forget the inside of things. Types, so the compiler would catch what we could not. Modules, so a team could divide the work. Indentation, because human eyes find structure in shape.

Read that list again. Every item on it was *put on the agenda* by a limit of human cognition.

That is narrower than it may sound, and the narrowness is the point. It is not that these features serve only people. Types catch errors a human would miss, and they also let a compiler generate better code and check an interface across a compilation boundary. Modules organise a team, and they also make separate compilation possible. Those benefits are real, they would be worth having if nobody ever read the program, and Chapter 14 keeps both for exactly that reason.

So the claim is about *origin and optimisation target*, not exclusive benefit. Each of these entered the language because a person could not hold something in their head, and each was then tuned against how well it relieved that pressure. Where a feature helped the machine too, that was a second benefit, not the thing that put it there. A processor does not need a variable to have a meaningful name, does not benefit from consistent indentation, and has no opinion about whether a function is short enough to grasp at a glance.

None of it is free, though almost none of the cost is at run time; Chapter 4 totals it up. The distinction above does real work later. When Part IV asks what to keep, the answer is not "whatever a machine strictly requires" — that discards types and modules and produces something unusable. It is: keep what survives the change of author on its own merits, and re-derive the rest.

What happened to the levels below

When a new level appears, the level below it does not die. It changes hands — and that is the part people skip.

There is more machine code executing today than at any point in history, and almost none of it was written by hand. Assembly is still the right answer for a boot sequence or a codec kernel, and almost nobody writes it. Both stopped being **written** and started being **emitted**. Chapter 5 follows what that actually looked like.

If you want one sentence for the whole book, it is this:

> Assembly did not replace machine code. High-level languages did not replace assembly. Each new level changed *who was responsible for writing the level below it.*

What this argument does not claim

The most common objection to this book attacks a claim it does not make, and the difference between the two is easy to state.

Nothing in the argument requires that humans stop being involved. It requires only that **the primary producer of a representation changes** — that the default first draft of the artifact stops coming from a person.

Those are very different claims, and only the second one is supported by anything. The two transitions above are the evidence, and in both of them the people stayed. What ended was not human involvement. It was the level below as the thing a person sat down and *wrote first*, at volume, as a matter of course.

So the objection — *humans will still review, debug, specify and own AI-generated systems* — is not a counterexample. It is a description of what both previous transitions looked like from the inside. Engineers reviewed compiler output constantly, and distrusted it for years, and the reviewing never made them the authors again.

That also tells you how to check this book rather than argue with it. The question is not whether humans still touch high-level source; they will, for a long time, and some of them will be very good at it. The question is whether the *first draft, at volume*, keeps coming from people. If in ten years the default origin of new high-level source is still a person typing it, the thesis is wrong, and wrong in a way anyone can observe for themselves.

One more consequence, and it is the one that matters for the rest of the book: if humans remain readers and reviewers but stop being authors, then a language optimised for *writing by humans* is being maintained for a job nobody is doing any more, while the job people actually still do — reading, checking, repairing — is a different job with different requirements. That gap is what Part II measures.

The objection that keeps returning

Each transition was resisted, and after the first one it was resisted with the same sentence:

> The generated code isn't as good as hand-written.

Chapters 3 and 4 take that seriously, because it deserves it — it was made by people who were right. What matters here is only the shape it had both times. The objection was true when it was made. It stayed true for years. It was never quite refuted: there is still hand-written assembly in your phone's codec, and there are still workloads where a garbage collector is unacceptable. What happened instead is that the economics moved underneath the argument, and the argument stopped mattering.

The sentence is being said again now, about code written by models, and it may well be true today. It was true the last two times.

What it has never once been is a reason the transition did not happen.

Why a model writes Python

Nobody evaluated the options and concluded that Python was the right target for a machine author. Those languages are what the training data contained, and Chapter 6 is about that mechanism. What belongs here is the consequence. A new author has been handed the previous author's tools and told to get on with it, and it works well enough that most people have stopped asking why the tools look the way they do. But go back to the list: readable names because humans

forget, indentation because humans see shape, comments because humans need reminding, small functions because human working memory is limited.

A model's failure modes are not ours. It benefits from structure too; what it does not need is *our* structure. And given the choice between carefully editing three lines and regenerating the whole function, it regenerates, because for a model that is the cheaper operation. For us it never was.

Every reason we built the third level began with us. None of those requirements should be assumed to transfer unchanged.

This is not a prediction about AI getting better

The argument in this book does not depend on models improving. That is worth saying plainly, because most arguments in this space do.

It is an argument about **selection pressure**. Python's success had many causes — libraries, timing, teachability. But its *design criteria* were overwhelmingly human-centric, and so were those of every language it beat. For thirty years the only kind of author there was, was a person.

Change the author and you have not changed the winner. You have changed the race — and nobody has yet run it.

I have watched this happen with infrastructure rather than languages. Token Ring was deterministic under load in a way early Ethernet was not, and for a decade that was the criterion that mattered — which is why serious networks bought it. It lost when the thing being selected for changed from predictable latency to cost per port. The engineers arguing for determinism were not wrong about determinism. They were answering a question the market had stopped asking.

That is the position we are in with programming languages. Anyone evaluating a machine-authored language by how pleasant it is to read is applying the old fitness function to the new race, and will be exactly as correct, and exactly as overtaken.

What a compiler's intermediate language is for

Here is a bug.

A model writes a function that takes an altitude, compares it against a safety threshold, and adjusts a rate of descent. It assumes the altitude is in metres. It is in feet.

Now fix it.

In Python there is nothing to fix. The assumption was never written down. It is not in a declaration, because there is no declaration; it is not in the type, because the type is `float`. It exists only in the arithmetic — in every comparison, every threshold, every multiplication that touched the value, and in the head of whoever wrote it, who was a model, and is not available for questioning.

There is no line to change, because the mistake was never in a line.

A person hits this too, of course. But a person hits it once, forms a theory, greps for the other sites, and reasons about which ones matter. That is a skill, and it is not one a generator has. Told that its program is wrong, a model's cheapest and most reliable move is to produce the function again. Sometimes the whole file. The cost of the fix is proportional to the size of the *program*, not

the size of the mistake — and every regeneration is a fresh chance to break something that was already correct.

You can argue that models increasingly edit rather than resample, and they do. But watch what an edit requires: the fault has to be locatable, and the fix has to be expressible in one place.

There are two ways to get that, and keeping them apart matters more than anything else in this chapter.

Inferred repair locality is what a good producer reconstructs — reading the code, forming a theory about where the assumption leaked, and deciding which sites matter. Models do this now, and they will do it better. Nothing here claims otherwise.

Intrinsic repair locality is when the representation states it, so the answer is looked up rather than inferred, and the completeness of a fix can be checked mechanically rather than believed.

The claim of this book is not that a model cannot infer its way to a correct repair. It is that inference and lookup scale differently. Inferred locality degrades as programs grow, as the sites spread across files, and as the assumption gets further from the code that violates it — because the information was never written down and there is no way to confirm the answer was complete. Intrinsic locality does not degrade, because there was nothing to reconstruct.

That is an empirical claim about scaling, not an impossibility claim about models, and Chapter 27 says exactly what would refute it.

Repair locality is the name for the property, and it is the first thing I would ask of a language whose author is a machine. It is also a property no human-centric language ever had a reason to optimise for, because human authors do the localising themselves, for free, and always have.

There is a general principle underneath it, and if you keep one sentence from this book it should probably be this one rather than the one about authorship.

There are two ways to make a machine producer more effective. You can make the code easier to **generate** — fewer tokens, a simpler grammar, a shape closer to the training distribution. Or you can make the result easier to **correct and verify** once it exists. Nearly all current effort, including nearly all of the argument people expect this book to be making, goes into the first.

> The property that matters in a machine-authored representation is not ease of generation. It is ease of deterministic correction and verification after generation.

Generation is the half that is already working. Models write plausible code now — at volume, cheaply — and they get better at it every few months without anyone changing the language. What has not improved, because no language change has been made to improve it, is what happens *after* the code exists and turns out to be wrong.

That asymmetry is the whole opportunity. It is also the half a representation actually controls: a better model writes better first drafts, but no model improvement can make a fault addressable that the representation never gave an address to. The unit bug earlier in this chapter is not hard because the model is weak. It is hard because there is nothing to point at.

So the fitness function below is not a list of things that would be nice. It is mostly one idea — make the consequences of being wrong proportional and addressable — taken apart into the properties a language would need to deliver it.

That is one criterion. There are six. Call them the machine-author fitness function, because that is what they are — the criteria that replace the ones we have been optimising against since Fortran.

1. Repair locality. Can a mistake be corrected without regenerating what was already right? Above.

2. Diagnostics. When the toolchain rejects a program, what does it say?

For a human author an error message is a courtesy — one input among many, alongside the debugger, experience, and a colleague.

A machine author has many inputs too. An agentic system reads test failures, runtime output, stack traces, repository search, static-analysis results and whatever its tools return. The diagnostic is not its only signal and I do not want to pretend otherwise.

What makes it different is its *kind*. It is the only channel that is authoritative — it decides whether the program is accepted at all. It is deterministic: the same program produces the same verdict, every time, with no sampling in between. And it is the only one that knows which rule was violated, rather than reporting a symptom downstream of the violation. A test failure says the answer was wrong. A trace says where it died. Neither can tell you what would have satisfied the rule, and Chapter 8 measures how often they say nothing at all.

That combination is what makes it worth designing. Diagnostics are the highest-value deterministic feedback available to a machine author, so in a language built for one they belong in the language definition rather than in whatever the compiler team got round to. Chapter 21 specifies them. Compare:

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

with what the same rejection could say:

```
E204  operand type mismatch
      at:        %7, argument 2
      expected:  exact_i64
      found:     text
      legal:     exact_i64 | approx_f64 | duration
      fix scope: this operand only
```

The first message describes a symptom. The second names a location, states the constraint that was violated, enumerates what would satisfy it, and — the part nobody currently does — tells the generator *how much of the program it is allowed to touch*. That last line is the difference between a patch and a regeneration.

3. Validity. How much can be established mechanically, before running it? Python does better here than its reputation suggests — it parses cheaply, and annotations plus external tooling check a great deal. The difference is what is *mandatory*. In Python the useful properties are optional or discovered at run time; units, effects, shapes and precision policies are carried by convention if at all. A representation built for a verifier can require them and reject a program that omits them. Chapter 8 measures what that difference is worth.

4. Correctness surface. How many low-level details must the author get right for the program to be correct at all? Assembly's is enormous — registers, calling conventions, stack alignment. Python's is small. This is the one criterion where Python already scores well, and Chapter 9 is about not throwing that away.

5. Semantic density. How much of the original intent survives? Write a Python loop that sums even squares and the fact that it *was* a sum over a filtered range is gone the moment the loop exists. Nothing records that the iteration order was irrelevant, or that the result was meant to be exact. Ask for a change later and that has to be reconstructed by guessing.

6. Tool leverage. Once the program exists, what can deterministic machinery do with it — check, prove, optimise, retarget, sandbox — without asking the model anything at all?

Three of these six barely mattered before. Repair locality, diagnostics and validity are properties nobody optimised a language for, because the author could be trusted to iterate, to read, and to recover. That assumption is the one that just broke.

Three objections I owe you

There are three obvious problems with everything above, and I would rather name them than have you carry them silently through four chapters.

If nobody reads the code, who reviews it? Code review, incident response, security audit, regulatory sign-off, liability when it fails — all of it currently rests on a person being able to read the program. This is the first objection every practitioner raises and it is a good one. The answer this book gives is that machine-checkable contracts, declared effects and enforced capabilities are a *stronger* audit surface than a human reading Python, not a weaker one, because they can be checked exhaustively and cannot be skimmed. Whether that convinces you is Chapter 17 onward.

Nothing beats numpy. Python's real moat is not its syntax, it is thirty years of libraries. A language with no ecosystem is unusable however well it scores on six criteria. Worse, there is a circularity: my explanation for why models emit Python is that Python is what they were trained on — which is also the strongest argument that anything new never gets adopted. I do not think this is fatal, and the reason is that the fourth language does not have to replace the ecosystem. It has to *call* it. Chapter 26 is about exactly that, and it is the difference between proposing a language and proposing an island.

Where are the numbers? I have asserted that the current arrangement is expensive. Part II measures what can be measured on one machine: how much of a generated function is layout rather than computation, how many sites one wrong assumption touches, and how many injected faults survive every checker a working programmer would have switched on. Two further questions need a model and a fixed task set; those chapters say so and leave the space marked. If the numbers come back small, the argument here is correspondingly weaker, and I will have said so in print.

What this book does

Part I is the history, because authorship has moved up twice already and the pattern is the argument. Part II is the diagnosis, with numbers. Part III is an inventory of the middle — SSA, LLVM IR, MLIR — and why none of it is the answer, which is the most useful thing in the book. Part IV derives the language by asking what to keep and what to remove. Part V is the language: grammar, types, verifier, compiler, all of it running.

The fourth language is coming whether or not this particular attempt at it is the right one. It is worth being specific about what it has to do.