Software Polyglots

I had an experience recently that is worth writing down, because it pointed at something larger than the project that occasioned it, and because my thinking about it has moved several times since I started.

I'd been trying to port a compiler from Java to C#. The natural move, in 2024, was to hand the job to a language model, and that is what I did — Claude, mostly, with Devin.ai sitting on top of it to coordinate the work. My first instinct was charmingly naive, though I didn't know it at the time. I wrote a long prose description of what the compiler did, handed it to the model, and expected a fresh C# implementation to come back out. I assumed, without ever quite saying so to myself, that a statistics-driven engine could understand my intention from a blob of text.

This is worth pausing on, because the assumption is more common than people admit. We spend our days reading essays produced by these models and marveling at how well they pick up nuance, and it is easy to slide from "the model understood my email" to "the model will understand my specification." But an email has a receiver who brings half the meaning. A specification doesn't. A specification has to carry its meaning entirely in the words, because the thing on the other end is a compiler, and a compiler is humorless.

My first try skewed, predictably, toward the compiler I already had — because my prose, no matter how I tried to abstract it, was full of borrowings from the old implementation. So I switched approaches. I asked the model to lift the existing Java, more or less directly, into C#, with the plan of reshaping the architecture afterward. This went better. Not great, but useful. And it taught me the first real lesson of the project, which was that I had been using prose for a job prose can't do.

Which brings me to the deeper thing the project kept pointing at.

The same model that can translate Spanish to Japanese well enough to fool most humans can't translate Java to C# well enough to fool a compiler. Why not? The languages are closer to each other than Spanish and Japanese. They share curly braces, C heritage, a roughly common object model. You could read one and guess the other on a whiteboard. And yet the translation is harder.

When a model translates between human languages, it isn't really translating syntax. It is translating meaning. Every human language is a surface over roughly the same underlying world — the same objects, the same actions, the same emotional textures. Two thousand years of poets and merchants and children learning from their parents have made sure of

that. The model doesn't have to invent the shared layer. The shared layer is already there, baked into the training data at a depth no one had to engineer.

Programming languages do not have this shared layer. Or rather, they have one, but it's thin, and it's thinnest exactly where the languages most differ. Java has checked exceptions. C# has properties and events and operator overloading. Python has indentation as grammar. Each language encodes not just syntax but a whole set of decisions about how a programmer is supposed to think about a problem. The differences are not decorative. They are load-bearing. A clean translation has to pick apart what the original programmer meant — not what they wrote — and rebuild it using a different set of primitives. Human translators do this constantly, because the meaning they're translating was already abstract. Code translators have to invent the abstract layer on the fly, and they have nothing canonical to anchor it to.

So it shouldn't be surprising the model is worse at Java-to-C# than at Spanish-to-Japanese. The surprise would be if it weren't.

Once you see the problem this way, the interesting question is what to do about it. I've come to think there are four answers, and they're worth taking in order, because the order mirrors the order I discovered them in.

The first answer is to wait. Models will get better at inferring the abstract layer from code, and eventually the gap will narrow. Probably true, and worth some of the credit. But it isn't a plan. It's a posture. And it leaves you with nothing to do today.

The second answer is to build the shared layer ourselves. This is the old dream of a universal intermediate representation — a language underneath the languages. The industry has taken several runs at it. The JVM was one. The CLR was another. LLVM and its frontend Clang are a third, and in some ways the most honest attempt, because LLVM doesn't pretend to be a language. It pretends to be a substrate. Swift and WebAssembly and a dozen others rest on it. Haxe is another interesting case: a language whose transpilers target C++, C#, Java, Python, and JavaScript, and whose designers have thought carefully about how to extend cleanly into each host environment. None of these has become the universal layer. They all do some of the job. They all punt on the parts the language designers couldn't agree on. The universal layer is not impossible, but it's a generational project, not a weekend one.

The third answer is coexistence. For most of the history of programming, the idea that two languages could share one executable felt exotic. You wrote your program in one language and shelled out to others if you had to. The shelling-out was always ugly. What Swift and Objective-C have demonstrated, for years now, is that coexistence can be graceful. The

languages share a runtime. They share memory. They call each other the way two dialects of the same town might. Apple's stack has proven something the rest of the industry has been slow to internalize: the goal isn't one language to rule the others. The goal is languages that cohabit well enough that you don't have to choose.

The fourth answer is the one I arrived at last, and the one I wish I'd arrived at first. You don't translate from the inside. You translate from the outside in.

Here's what I mean. My original mistake was asking the model to understand a program by reading a description of it. The better question, it turns out, is: what does the program do? Not in the abstract, but concretely — what inputs produce what outputs, what edge cases matter, what invariants hold. These are the things tests describe. And tests, unlike prose, are unambiguous. They carry their meaning entirely in themselves. They don't depend on the goodwill of the reader.

If you start with a thorough test suite, written against the behavior you care about rather than against the structure of the original implementation, you have given the model something prose cannot give it: a set of guardrails, and a way to know it's done. Translation becomes a closed-loop activity rather than an open-ended one. The model writes code, the tests run, the tests pass or fail, the model iterates. The abstract layer that used to have to live in the prose — the shared meaning between the old language and the new — now lives in the tests, where it can be executed instead of merely hoped for.

This is really just Test-Driven Design, except applied across a language boundary. Nothing new in principle. What's new is that the thing on the other end of the tests isn't a programmer but a model, and a model responds very differently than a programmer does when the tests fail. A programmer gets frustrated. A model iterates. That is a genuinely useful asymmetry, and it is wasted if you don't give the model tests to iterate against.

The learning trajectory, for me, went like this. First I tried prose, because I had been lulled by the fluency of chat into thinking the model could read my mind. Then I tried lift-and-shift, which worked better because the old code was, in a sense, its own specification. Only after those first two failures did I notice the obvious thing: the cleanest specification I could give the model wasn't prose and wasn't old code. It was tests. Tests are the only form of specification that can't lie about itself, because they run.

I suspect this fourth approach is the one most professional teams will converge on, for reasons that have little to do with the theory and everything to do with evidence. When you hand a human manager the output of an AI-driven port, the first thing they want is not a compile success. It is a green test suite. Correctness, at the level of the organization, has always been a social fact rather than a technical one, and tests are the social currency. An

LLM working without tests is producing assertions. An LLM working with tests is producing evidence.

The four approaches aren't competitors. They stack. Wait, but don't only wait. Build shared layers where you can, because the industry needs them. Embrace coexistence, because it is here today and it works. And wrap the whole thing in tests, because tests are the one kind of specification the model cannot hallucinate past.

If I were starting the compiler port again tomorrow — and at some point I probably will — I'd do it very differently. I'd write a test harness first, in the target language, against the behavior of the source compiler. I'd treat the Java implementation not as a spec but as an oracle — something to compare against, not something to copy. And I'd let the model translate in small increments, checked at every step, rather than in one heroic pass.

This is not a clever idea. It is the idea programmers have used for thirty years to keep themselves honest. The only thing that's changed is who's being kept honest. The model needs the same discipline we do, for roughly the same reason: because intention, without a way to check it, is just a feeling. And a statistics engine reading a blob of prose, however charming its responses, is not a substitute for evidence that the program actually does what you meant.

I had to learn that the hard way. Which is, in fairness, the only way anyone learns anything worth knowing.