The Old New Thing

Here's something that has been bugging me for a while. Everyone talks about AI as if there were one kind. There's "old AI" — the symbolic, rule-based, Lisp-powered kind that tried to model the world with logic — and "new AI," meaning the large language models that learned to talk by reading the internet. And the story we've all decided to tell ourselves is that the new kind won, the old kind failed, so just move along.

But this story is wrong. Not just a little wrong. It's wrong in a way that's causing people to make big mistakes right now, mistakes visible in startups and established companies, various articles published on the Web, and elsewhere.

The old AI didn't fail because its ideas were bad. It failed because the hardware wasn't ready and the data wasn't there. And the new AI hasn't succeeded in doing what the old AI was trying to do — it succeeded at something different. Something genuinely impressive, but different. Confusing these two things is like saying the airplane replaced the submarine because they're both vehicles. They go in different directions.

I've spent *some* time in both worlds and notice something weird. The people building language models seem to almost never talk to the people who spent decades building symbolic reasoning systems (I never see articles about such conversation). And the symbolic reasoning people, for their part, seem mostly to have retreated into niche academic corners or places where Computer Science history is honored and can be preserved, mumbling about how the world has gone mad. Neither group may realize they each have exactly half of what the other needs.

Large language models are very good at generating plausible language, and it turns out that's enough to do a shocking number of useful things. But generating plausible language is not the same as reasoning about the structure of the world. An LLM can write you a poem about a bridge. It can't tell you whether the bridge will fall down. For that you need a model of the world — the kind of thing the old AI people were actually trying to build.

So what would happen if you combined them?

This is the niggling question for me, because almost nobody seems to be asking the most concrete version of it. Lots of people ask it in the abstract — "multimodal AI! neurosymbolic reasoning!" — while somehow missing the answer that's staring them in the face. Because there's a tool sitting right there that was designed from the ground up to do exactly this kind of integration. It's called Lisp, it's been around since 1958, and everyone in AI has been studiously ignoring it since about 2012.

I know, I know. Lisp. You've heard the jokes about the parentheses. There's a famous old quip that Lisp stands for Lots of Irritating Superfluous Parentheses. The people who say this are the same people who look at a violin and complain it has a weird shape. The shape is why it works.

Let me explain what I mean by vertical integration, because I think it's the key idea most people are missing.

In most programming setups, there are layers. You write your logic in Python or Java or whatever. If you need a domain-specific language — say, for describing robot motions or chemical reactions or financial instruments — you build a separate thing. Maybe it compiles to your host language, maybe it has its own parser, its own runtime, its own set of quirks. Every layer boundary is a place where information gets lost and complexity sneaks in.

Lisp doesn't work this way, and I don't think most people appreciate how different this makes it. In Lisp, the language and the metalanguage are the same thing. Code was data way back when, and before the idea that data is code. Programs are trees you can walk through and rearrange. Because of this, you can build a domain-specific language not as a separate system bolted on top, but as a natural extension of Lisp itself. You're not crossing a boundary. You're growing the language toward your problem.

This is what I mean by vertical integration. The DSL for describing robot motions, the symbolic world model, the planner, the inference engine — these don't have to be separate systems stitched together with duct tape and JSON. They can be expressions in the same language, and they compose with each other the way functions compose. I've written enough glue code between separate systems in my life to know that this difference matters enormously in practice.

Why does this matter for AI right now? Because the hard problem isn't making language models better at generating text. That's going to keep improving on its own. The hard problem is connecting language to the world. And "the world" here doesn't just mean the physical world, though that's a big part of it. It means any domain with structure — physics, biology, law, engineering, music, planning and logistics — where you can't just wing it with plausible-sounding output, where you need actual reasoning about actual things.

Consider a robot. A language model can understand your instruction — "pick up the red block and put it on the blue one." LLMs are great at that part. But actually executing that instruction requires a world model, a representation of where the blocks are, what the robot's arm can reach, what happens if you bump into the table. This is a symbolic

problem. You need to represent states, reason about transitions, and plan sequences of actions.

Now, you can build these two systems separately and wire them together with an API. Lots of people are doing exactly this. But the seam between them is where all the bugs live. Every time you cross from natural language to symbolic representation and back, you lose information and introduce ambiguity. The two systems disagree about the state of the world, and you write increasingly brittle glue code to reconcile them, and the whole thing slowly turns into the software equivalent of a Rube Goldberg machine.

Or you could use a language where there is no seam.

In Lisp, you can write your world model as a symbolic structure — because Lisp was literally invented for symbolic computation. You can write your planner on top of it — because Lisp has had planning and reasoning systems since before most of today's AI researchers were born. And you can write the interface to your language model as an ordinary function call that returns an ordinary Lisp expression, which you can then manipulate, transform, or feed right back into your symbolic pipeline. No serialization, no deserialization, no impedance mismatch. The LLM's output becomes a first-class citizen in your reasoning system.

This is compositionality, and I think it's Lisp's deepest advantage. Not the macros, though they’re pretty cool and offer powerful compiler/interpreter capabilities predating ANTLR by a few decades. Not the REPL, though Lisp users would be lost without it. Not even the homoiconicity, though that's what makes all of it possible. It's that everything composes. You can take any two pieces and stick them together and the result is also a valid piece. In a language where code is data and data is code, the distinction between "calling a language model" and "doing symbolic reasoning" dissolves into nothing. They're just different functions that return the same kind of thing.

I find it funny — funny in a slightly tragic way — that technologists in general and in the AI field keeps inventing new languages and frameworks to solve integration problems that Lisp solved in the 1970s. We see new languages, a new "AI agent framework", a new "prompt orchestration language", or a new DSL for writing robot policies emerging all the time – the pace of change is gobsmacking. Many reinvent a small, poorly understood subset of what you get for free in any decent Lisp. They trade away the vertical integration that's right there for the taking, and in return they get… what, exactly? A syntax without parentheses? If that's your biggest concern when building the future of machine intelligence, I'd gently suggest your priorities need rearranging.

I realize I'm being a little unfair. There are real, practical reasons people don't use Lisp, and the biggest one is that the ecosystem around Python is enormous. Libraries matter. Communities matter. When there are forty thousand machine learning packages on PyPI and twelve on Quicklisp, that's a genuine constraint. I get it, I really do—though I also observe that RESTful HTTP calls help to avoid being pinned to a language, and the ecosystem of RESTful calling frameworks can be easily made accessible via Swagger et al in at least one of your favorite languages. But there's a difference between choosing a language for practical reasons and choosing it because you think it's the best tool for the job. Most people picking Python for AI aren't making a technical judgment about language design. They're following the crowd, the way people in the nineties chose Java because their manager told them to.

And this matters more right now than it usually would, because we're at a genuinely unusual moment. We have, for the first time, systems that understand natural language well enough to be useful. And we have, from decades of work in symbolic AI, the tools and the theory to build formal models of specific domains. What we don't have — what almost nobody is seriously working on — is a good way to connect them. The opportunity is in the connection, and the connection is fundamentally a language design problem.

Think about what a domain-specific language actually is in this context. It's a formal language for talking about a specific part of the world — a language for describing chemical processes, or specifying legal contracts, or choreographing robot motions. Each one captures the structure of its domain precisely enough to reason about.

Now think about what an LLM does. It takes fuzzy natural language — the stuff humans actually say — and makes sense of it. It's a translator from the informal to the semi-formal.

If you squint even a little, the architecture writes itself. The LLM is the front end — it takes human language and compiles it into domain-specific expressions. The world model is the back end — it takes those expressions and reasons about them, simulates them, optimizes them, connects them to the physical world. And the language in which those domain-specific expressions live, the thing that makes the front end and the back-end work together cleanly, is a composable, extensible, symbolically grounded language. One where new DSLs can be spun up as easily as writing a new function.

It could be Lisp. Honestly it doesn't have to be, though I think you'd end up reinventing most of Lisp if you tried to build it from scratch. But whatever it is, it needs the properties Lisp has — homoiconicity, macros, first-class functions, and the ability to treat code as a manipulable data structure. And it's somewhat remarkable that the industry's response to

needing those properties has been to build seventeen different ad hoc systems, each missing half the features, rather than using the language that has all of them.

There's a deeper point here about LLMs and world models that I think most people get wrong. The conventional wisdom is that LLMs will eventually replace world models — that as language models get bigger and better, they'll absorb enough world knowledge that you won't need explicit symbolic representations anymore. Everything will just be weights in a neural network.

I don't buy it, for a reason that's almost mathematical. A world model is compositional. It's built out of parts that have independent meanings. You can take the piece that models gravity and combine it with the piece that models friction and get a model of a ball rolling down a ramp. You can inspect each piece separately. You can prove things about them. You can reuse them in new combinations their authors never imagined.

An LLM isn't compositional in this way. What it knows is smeared across billions of parameters in a way that's basically opaque. You can't extract the part that knows about gravity. You can't combine it with another model's understanding of friction in any principled way. You can't prove anything about what it will do. This isn't a flaw that better training will fix — it's a fundamental property of the architecture. Asking a neural network to be compositional is like asking water to be dry. That's just not how it works.

So LLMs and world models aren't competitors. They're complements. The LLM understands what you're saying. The world model understands what you're talking about. You need both, and you need a clean way to connect them.

Generative AI makes all of this even more interesting. When people say "generative AI" they usually mean generating text or images. But here's what I keep thinking about: what if you could generate executable world models? What if, instead of having an LLM write you an essay about bridge physics, it wrote you a program that simulated bridge physics — a program you could run, test, compose with other programs, and use to make real engineering decisions?

This is where Lisp's compositionality really shines. Because a generated Lisp program isn't just text that looks like code. It's an actual data structure you can walk, inspect, and transform before you ever run it. You can check it against constraints. You can merge it with other generated programs. You can build a library of generated components that snap together like Lego bricks, because the language enforces compositional structure at the deepest level. I've been saying for years that code generation is going to be huge, and I think the part most people haven't figured out yet is that the generated code needs to be in a

language where composition is natural, not something you have to fight the syntax to achieve.

This is the direction I think things should be going. Not just bigger language models. Not more layers of glue code. Not another framework for prompt chaining written in a language that treats code as text. The real opportunity is in making AI systems that can move fluently between natural language and formal representation — between the fuzzy and the precise, between what humans say and what the world actually does.

The irony is that the tool for this already exists. It's the same one the AI pioneers used sixty-seven years ago when they were trying to build thinking machines and the rest of the world thought they were crazy. It just needs to be picked up again, dusted off, and connected to the one genuinely new thing we have — language models that actually work.

The old AI and the new AI aren't enemies. They're two halves of a system that was never finished. And the language that was designed to build the first half happens to be exactly what you need to connect it to the second.

Sometimes the oldest tool in the shed is the sharpest.