JUMBO REVISITED, PART ONE: JUM

MICHAEL ROBERTS
michael@vivtek.com

JANUARY 18, 2026

Douglas Hofstadter's program Jumbo was a model of the human cognitive mechanisms involved in solving anagram puzzles. It defined an approach that set the agenda for decades of research through a dozen doctoral dissertations, and my current focus is to recapitulate (some of) that work in a single, unified codebase. This report addresses the implementation decisions and lessons learned during my implementation of the first stage of Jumbo's process model.

All the code from this project can be found at https://codeberg.org/Vivtek/AI-FARG-Jumbo for your reading pleasure.

Contents

The story so far

The project reported on here is the first half of the functionality of Jumbo[1], Douglas Hofstadter's model of human cognition as applied to the process of solving anagram puzzles. I'll explain later what "first half" means and why I've broken the development into two parts, but first, let me note that this report is written to a more technical level than the last one. Apologies if you were looking forward to more in the same genre, but this is an *intermediate* report and I don't want to spend a lot of time on it. That said, this section is intended to describe the basic approach and the Jumble domain in very brief terms to get you oriented.

[1] Hofstadter, D. R. (1983) *The Architecture of Jumbo*

The architecture that Hofstadter described in his 1983 paper is called the *parallel terraced scan*, and I've written about it at more length in the first technical report in this series[2], which described the implementation of a pilot version of the architecture. The parallel terraced scan is the core of Hofstadter's research ap-

[2] Roberts, J. M. (2025) *A Minimum Viable Product for the Parallel Terraced Scan*

proach, the basic architecture used in about a dozen doctoral dissertations by Hofstadter's own students in the Fluid Analogies Research Group (*FARG*) and a handful of others elsewhere. It is a technique for building (or perceiving) complex structure in input data by progressing in very small steps executed in a random order, but with a bias toward pathways that seem more promising. When designed and balanced well, it cuts through the exponential complexity of searching every possible solution in a problem space. Based on my many weeks of experience with it, in fact, I can tell you that you don't have to design or balance it particularly well for this approach to feel like magic.

In the work covered in November's report, I implemented a parallel terraced scan engine and tested it in an extremely simple problem domain: the arrangement of a list of numbers into a sorted order. I called this domain *NaiveSort* because the rules used to build the list structure are simple and *incomplete*, in that numbers taken from different sublists for comparison are only merged into a single sort order in a few situations that are easy to code, and in other situations the program takes no action. Despite being hobbled in this way, NaiveSort still manages to sort lists of numbers robustly.

Hofstadter and his group call their application domains *microdomains* to distinguish them from the sort of larger-scale domains typical of cognitive research at the time. A "domain" in this sense in the 1980′s tended to mean a field of human endeavor, something like "physics" or "vision." Hofstadter's approach was to look at much smaller bits of thinking, such as a single type of puzzle. My own work is more software-focused—instead of being a cognitive scientist, I'm starting to think I'm more like a cognitive artisan—so I'm comfortable with just calling these "domains" instead of microdomains, saving me five letters every time I talk about them.

The first such (micro-)domain was in fact Jumbo, which Hofstadter worked out over the course of 1982 and described in his 1983 paper. Jumbo models the human cognitive mechanisms involved in solving Jumble puzzles in the newspaper, which is a far more restricted domain than *physics* but, Hofstadter insists (and I agree), actually reveals the underlying mental processes more clearly. I don't want to spend a lot of time describing Jumble in this report, since I plan a more complete presentation in Part Two of Jumbo Revisited, but if you've never seen a newspaper Jumble, it consists of several scrambled words which, when unscrambled and written into the answer boxes, drop their letters into slots to define a *second* puzzle. The unscrambled version of that second level is the punchline to what is generally a groaner of a joke in an accompanying cartoon. Some people live for this stuff.

When you do one of these puzzles, you might see something like *SIONE*, and the expectation is that you'll quickly perceive this as a scrambled version of a common word. In my experience, five-letter sequences jump out at me and six-letter sequences take a little chewing before they yield. When you see SIONE, you might think something like "EOSIN? NISOE? Oh, *NOISE*!" It's quick, well under a second. What's happening there? Hofstadter theorizes that there are *subcognitive* processes below the level of our conscious awareness that see groups of letters that look good together (like *OI* in NOISE), and then just kind of shuffle them around a little before recognizing a word. But notice that before we get to a word, we sometimes go through some intermediate steps that look like words, but aren't. In fact, even the sequences originally presented in the puzzle are usually arranged to look something like words, making it more difficult to see a different order for the letters.

The architecture Hofstadter proposed to model this type of process consists of a *Workspace* containing bits of data (which I call *units*). These units can either be things like letters (or numbers, for NaiveSort) or containing units that represent bonds between letters, any kind of structure that combines the low-level units into higher-level patterns.

This structure is built by the action of many bits code called *codelets* that are placed on the *Coderack*, basically a list of possible things to do next. Codelets can be either *scouts*, placed onto the rack at any point to look around the Workspace and find something to work on, or *follow-up*, which follow up on some task carried out earlier that needs more steps to reach a conclusion.

Most of the actual work involved in developing software to this architecture is writing codelets that define and process your target domain. The terraced scan engine itself is provided in a library. Of course, in addition to the codelets that the engine will call, you can include any other subroutines or libraries you need (Jumbo has a simple database of likely-looking letter groups called the *Chunkabet*, for instance). But the core of the development is the domain object.

My goal in this line of research—a goal I first formulated in 1996—has been to build that central terraced-scan library, then redevelop an assortment of domains featured in the FARG output over three decades of doctoral work in order to distill out the functionality and semantics they have in common. For obvious reasons, it felt right to start with Jumbo to do that, the first of them all.

So let's look at the implementation of the Jumbo domain in more detail, and why I only did half of it before feeling the need to step back and do more work on the terraced scan infrastructure.

Implementing Jumbo, or at least Jum

The implementation of terraced-scan domains turns out to be a complex endeavor. (I don't know why that surprised me, but somehow it did.) One of my goals in this recapitulation line of research is to implement several of them in order to get more intuition about how the parts of the architecture work together and how each overall system reflects whatever part of human semantics and cognition it models. In other words, I see these coding projects as a kind of learning how to learn; as I code more of them, I hope and expect it to be easier and easier to tackle a new one.

The Jumbo domain is probably about five times more complex than NaiveSort. In NaiveSort, we have two types of units and four codelets; in Jumbo we have four basic unit types and six auxiliary ones, and thirteen codelets implemented even at my cutoff point (I'll get to that in a minute, don't worry). In principle, this shouldn't matter, but in practice, as always, it does. The interactions are harder to understand and debug and that forced me to think of better ways to impose some order on the chaos. Ultimately I decided that I wanted to bring those techniques back into the core library and start over, and that's why we have a Part One.

The basic unit/codelet architecture of Jumbo is fairly close to Hofstadter's description in his 1983 article. We start with *letter* units, one for each letter in our problem statement. Letter-spark-scouts are released when there are unbound letters in the Workspace (more on the bound/unbound distinction later), and each scout chooses two (unbound) letters at random and decides whether to make a *spark* unit between them. If it does, it also posts a spark-checker codelet to check the letter pair for quality. If it doesn't, or it turns out there aren't two unbound letters to be selected, it *fizzles*. Fizzling means that the codelet just didn't find the right conditions by the time it ran, and it is deleted (well, it's put into the log) and nothing else is changed.

The spark-checker is more complex. When it is called (and remember that other spark-scouts might have been called in the meantime, or later codelets in the process) it can't be sure that the current Workspace situation is the same as when the spark was created, so the first thing it does is ask the spark for its "from" and "to" units, and it extracts a *framelet* from the Workspace that describes the mutual neighborhood of these two units. The framelet is just a little context record pertaining to some part of the current state of affairs, but at this stage all you need to know is that it can be seen as the answer to the question "Tell me about the situation of these two units". The extractor finds the units, checks whether they're already bonded together or separately, and returns all that in a little context record for the codelet to use to make decisions. I settled on this model relatively quickly in the development of NaiveSort, and it's still a good choice.

If the two letters turn out to be phonetically incompatible—which the spark-checker determines by asking the *Chunkabet*—the spark-checker *fails*. Unlike fizzling, failure means that something has to be done to the Workspace, and in this case we have to delete the spark unit and increment the *frustration* of both of the units we looked at. Frustration is part of global temperature; at some point it might also be used to focus codelet attention on problem spots. We'll see. So far, the only thing it does is raise the temperature, and so far all *that* does is provide an indication that things are bad enough to release breaker codelets.

The Chunkabet is a little database of workable letter clusters, along with some handy functions for comparing them and adding up sequence quality metrics.

If the two letters are compatible, though, then we might bond them. That decision depends on whether they're already involved in bonds and if the resulting bond would be superior to the bonds that would have to be broken up. If a single letter tries to bond to one of the letters in an existing bond, then a three-letter bond can result, if that three-letter sequence is phonetically valid. In other words, if an *m* tries to bond to an existing *oi*, then it will fail. Each bond group can only be a consonant cluster, a vowel cluster, or a syllable-final cluster like *se* or *ly*. This knowledge, by the way, is not reflected directly in the code; the Chunkabet simply doesn't include mixed clusters as valid.

Honestly, not having this kind of knowledge in the code is a semantically risky strategy and yet another indication that the parallel terraced scan itself is not a full cognitive model.

If we ultimately decide that the letters can bond, then the codelet *fires*, meaning it makes a change to the structure of the Workspace. In this case, it *promotes* the type of the spark to *bond* and sets the frustration of each letter to 0, lowering the temperature of the Workspace. By the way, even though by convention I'm telling you that fizzling makes no changes, failing makes limited changes, and firing is allowed to add structure, as far as the codelet handler is concerned you can do anything you want. The fizzle, fail, or fire messages are logged equivalently, and any of the three can be given *fff actions* to be taken against the Workspace. The distinction really applies to my own evolving sense of best practices for codelet design. As one of possibly two people on planet Earth writing codelets in the past few years, I think I get to set these industry standards as I see fit.

Let's step back just a little at this point. The ability to bond letters into bond groups based on Chunkabet quality judgments was as far as I got when I first started implementing Jumbo in 2024. That effort succumbed to technical debt and I went off doing other things for a while before getting back into the game at the end of 2024. And when I decided in August 2025 to develop the parallel terraced scan separately before returning to the Jumbo domain, I based NaiveSort on this stage of processing, eliminating the need for a Chunkabet by changing the "quality decision" to a simple greater-than check. So this part of Jumbo was essentially already done by the time I turned my attention back to Jumbo in November 2025.

The full Jumbo domain, though, not only builds higher levels of structure (gloms, syllables, and a wordoid), it also includes *shuffling* codelets that can move the letters between syllables in the attempt to find a better top-level word. Hofstadter groups these into *entropy-reducing* and *entropy-preserving* codelets; we reduce entropy by grouping things successively, but by shuffling things around we're neither grouping nor ungrouping. (The third category of *entropy-increasing* codelets are, of course, the breakers we release when the problem gets "stuck".)

It took me just a few days to port the NaiveSort style back to the Jumbo domain, and then I buckled down on higher-level structure. Once letters have coalesced into bond groups, they can be firmed up into *gloms*. A glom is the same group of letters, but it takes its component letters out of circulation for the bonding process. Concretely, a glom-spark-scout looks at bond groups and optionally creates a *glom-spark*. A letter-glom-spark also looks at single letters and judges whether they can be considered a single-letter "cluster" in their own right (for instance, the *h* and the *e* in *here* are single-letter gloms). The glom-spark is then judged by the follow-up glom-spark-checker and converted into a glom if everything looks good. To do that, it promotes the spark to *glom* type, deletes any bonds, and changes the letters to type *bound-letter* to take them out of circulation.

This is called chunking *and I intend to give it a little more formal infrastructure this month.*

"Wait," I hear you asking, "All you did was propose a link between two letters, and then check it. What might not look good?" Oh, my sweet summer child. Everything that follows describes bugs that took me hours or days to comprehend, but remember that codelets are posted and then execute in a *random order*. So by the time the glom-spark-checker follow-up runs, if it's looking at a bond group then the bond might have been deleted because another letter bonded one of its component letters. If the glom spark is looking at a single letter, that letter might now be bonded and no longer be a single letter. The bond group may already have been glommed by *another* spark, meaning the bond is deleted *and* the letters aren't unbound letters any more. If you fail to check for any of these conditions, very, very weird things will happen in your Workspace and you will undergo a mind-expanding experience. Lucky you!

At any rate, once some letters are glommed, we start looking at which gloms can fit together into a syllable. This turned out to be more complicated than I expected. (Not the first time I've said that, and won't be the last, I assure you.) Syllable construction is the first place where it becomes important to be able to get a syllable quality measure from the Chunkabet, and gloms don't all fit together into syllables in any old way—some letter combinations are only good at the start of a syllable, some only at the end, and some (typically vowel clusters) in the middle. So you have to check every combination for validity and return a combination that works. You can just see the code for how I ended up solving it, but that alone took me several days.

During syllable checking, we also have to regulate competition between syllables for the good gloms. Here's where the syllable quality comes into play. If a syllable ends up trying to claim one or more already-bound gloms, then we compare the quality of the newly proposed syllable against its competitor or competitors. If it's superior, it gets the gloms. In a more mature implementation of the domain, we would actually be looking at the global Workspace temperature to determine how strictly we insist on an improvement in quality, but so far I haven't needed to implement that to get reasonable results.

Once syllables are represented in the Workspace, we start releasing wordoid-scouts. All that does is look for a syllable and push it into the top-level word, chunking it in the process. Once all letters are bound into gloms, all gloms into syllables, and all syllables into the top-level wordoid, we're done.

In a full Jumbo implementation, this is where the shufflers would kick in (not necessarily *here*—they could also start working at the syllable level, for instance—but this is where I'd start implementing them). But it took me over a month to get to this point, and in that time I'd come up with a new organizing principle for Workspace changes that I call *moves*.

See, you can't test codelets in isolation—or rather, you need a known Workspace configuration to run a unit test of a given codelet. During the implementation of NaiveSort, this was relatively easy to do; I just defined a "move_and_step" that would post a given codelet explicitly and then single-step the terraced scan. Since there was only one codelet on the rack, and it had a known set of parameters, we can usually know its outcome and verify that it worked as expected. (Just a quick note from forty years of programming experience—any complex code that you're not running through regression tests is broken even if you don't know it yet. So getting codelets into a regression testing framework was *vital* for my sanity. Some people may have the mental discipline to change code without breaking something else; I am not one of those people.)

For the early stages of Jumbo, this approach still worked. By the time I got to gloms that technique was getting tricky—sometimes I needed to change the types of some letters so the glom codelets would only see the ones I wanted. But by the time I started putting gloms together into syllables, I realized that I needed to be able to build a known Workspace configuration from scratch. Which—this is key—is *really ugly* to do with codelets called explicitly. It occurred to me at some point that if I could just call the parts of the codelets that made changes to the Workspace independently, I could set things up any way I wanted.

A few weeks prior to this, I'd been musing that the state of the Workspace looked kind of like the board state in a game, and that changes to it could be seen as moves from one valid state to another. The trajectory taken by the terraced scan starts looking like search (which, of course, is exactly what it is). I realized that the "parts of codelets that made changes" were in fact those moves, so I converted most of the early-stage codelets to do all their work with explicitly defined moves, named those moves in the domain object, and made it possible for the terraced scan engine to execute those moves by name—all so I could call them from test setup scripts.

If you're now thinking, "Hey, that sounds like you've incurred some technical debt because I'll bet you didn't really test any of that, did you?" then you know me all too well. I started sweating at this point, but the moves organized the code so well that I had already decided there was *no way* I was going to write the shuffling codelets without them—or without the *other* new organizing principle I'd come up with. (December was a month full of insight.)

Sometime in November or December I had finally found Hofstadter's original 1983 article. Up to this point I'd been working from the version that had been rewritten as Chapter 2 in his 1995 book. (I should note that *sans* institutional affliation, my access to the literature is, well, let's call it stochastic.) The original article was substantially the same, but there was a remark in passing that doesn't seem to have made the 1995 cut, which was something to the effect that *until a spark succeeds, no change is made to the Workspace.* Huh. My kneejerk strategy back in 2024 had been to represent the spark as a unit; that's why I came up with the idea of promoting it to a bond if it succeeded. It had to be in the Workspace in March 2024 because that's the only data structure I had to store any explicit data about the process, and also because my original graphical output drew sparks in the Workspace representation, so they clearly had to be in the Workspace, right?

But in the meantime, I'd started relying on framelets for temporary storage of contextual information. If a spark was not stored as a unit but as a framelet, then only when the situation had passed the checker would any unit be made in the Workspace. It matched up with Hofstadter's original notion—his reason for mentioning it had very little bearing on what I was actually doing, but the more I thought about this, the more I liked it. When I originally thought of the notion of a framelet, I went tearing off in all mental directions about what they might become; they're meant to be the atomic component of semantic structure, after all. If I used framelets to keep track of each ongoing decision process, then I wouldn't necessarily have to delete any of it. In fact, I realized, the "framelet cloud" of perception of the current problem was really kind of the semantic description of its parts and how we'd got there. In fact, I further realized, the "framelet cloud" *was the episode of episodic memory.*

Episodic memory still seemed a far-off thing to be thinking about, but it was clear that I needed to go back to the terraced-scan engine and NaiveSort to do some proper testing before tackling the shuffling codelets. I decided to cut the wordoid construction process off and call it "Jum", and all I'd have to do to call it a milestone would be to turn my unit-tested codelets into a full test run configuration and shake any remaining bugs out, which I envisioned as a simple, quick step.

You're not supposed to define milestones as you reach them, but I don't have a project manager, so here we are.

Reader, it took three days of full-time agony to debug those codelets and get them working. I despaired of ever comprehending what was going on. (In the end, what was going on was my continued failure to check that units were still alive and that they hadn't changed type, just as I told you earlier. But it looked different at every level!) Finally, I wrote a done-checker to stop the process when the wordoid contained all the letters in the Workspace, and then, just as with NaiveSort, I saw that without a breaker codelet it would just get stuck early on. So I wrote a single breaker I called the *devourer* that found the highest-level unit it could, and disbanded it, freeing its content units. Then I ran the thing. A lot.

If I gave it a seed that permitted a wordoid that passed the Chunkabet filter, it always found a wordoid. Eventually. Sometimes it found a good one in 40 codelets. But sometimes it blundered around in syllable space finding the same bad combinations over and over and over again before it finally staggered to a conclusion. I usually got bored and killed any long runners, but once I let one run for over 30,000 codelets before it finally managed to end. My data structures were not designed to handle the thousands of dead units this created, so it got slower and slower as it went. I'd been considering putting this up on a webserver for public edification, but a process that takes 30 seconds to finish? That's going to get really old, and as it happens at least one time in five on *any input*, I can't do that.

Then the scales fell from my eyes—the framelet cloud is an episodic memory! The framelets that have just built the syllable *HEET* three hundred times in a row, if we first look through a hypothetical framelet index, could turn out to be the *same framelet* recognized as already having been tried. If we hit high temperatures repeatedly after forming *HEET*, then we can just *not do that again*. Obviously, that's not a binary choice—like anything in FARGland, it would be a gentle and growing pressure.

I had just reinvented the self-watching wheel, and the student was enlightened. No *wonder* Hofstadter was so focused on self-watching in 1983/1984. As it turned out, Copycat's focus mechanisms in the following years introduced enough randomness into the search that it didn't get stuck in snags as often as I was seeing in my implementation of Jumbo, but perhaps Hofstadter's implementation *had* blundered through the same short syllables five hundred times in a row before finally making better life choices. And on the hardware available in 1982, that didn't run in 30 seconds, either.

At any rate, I had reached my hastily-defined milestone, Jum went up to the Codeberg repository, and I sat down to write this report.

Lessons learned

I like a lessons-learned section, but given the arbitrary nature of this milestone it seems like a blurry snapshot, even aside from the fact that the last few paragraphs of the last section already sound like lessons learned. But let's try on a couple for size anyway.

I need better instrumentation for debugging. My usual method of placing printfs into the code helps, but it takes a long time for the meat to start to distill out a picture of what's going on, especially given that an error that only crops up after a particular sequence of codelets is a pain to reproduce (and you never do reproduce it entirely; instead, you have to learn to perceive a whole class of error behavior). It would be nice if I could inspect more detail of what happened in a given problematic run, and it would also be nice to have a test run facility that would run instances successively until a given run met certain criteria.

I need to work on efficiency. Longer runs might very well be important in many domains, but even Jum's looooong runs brought the Workspace table module to its knees. I suspect that's because my selection method for two gloms, for instance, scans the entire list of gloms to do that, which includes all the dead ones that the devourer has broken up. And that's a lot. A stopgap would be just to keep a short list of units that are still alive for selection calls, but ultimately I need to rebuild the entire engine in C or Rust, not in Perl, with proper heap management. It's said that all truly complex software ends up reimplementing half of Common Lisp, but badly—but I think I can do better than "badly."

Moves are a good move. Actually, *anything* that reduces the effort to write a codelet is a good move. I want to pull the move mechanism into the Workspace object entirely, logging the move trajectory separately from the Coderack log. Moves should also make it easier to provide event subscribers with information relevant to displays.

Self-watching is key. In my last report, I thought self-watching was key to the larger cognitive model because that's the level at which the model can build semantic models of what's going on. Now I realize that self-watching is vital even at the Workspace level.

Now what?

So where do we go from here? First and foremost, I go back to NaiveSort and the core terraced scan engine to implement and test what I've been thinking of as the two new codelet styles. The "move style" is based on my strict distinctions between fizzle, fail, and fire, and puts all Workspace changes into named moves. I do think it's important to allow codelet writers to design codelets in any way they see fit, so it won't be an error to work outside of the move system, it'll just be easier to stick with it—for me, anyway, and of course to date that's literally all that counts.

The "framelets-and-move style," then, makes no changes to the Workspace unless they're true perceived structure built by codelets working from framelet-based notes. I think I'd like to take this opportunity to think about the best way to index framelets during the run, perhaps even implement this form of self-watching to allow us to avoid earlier partial solutions that turned out unfortunately.

After that, I'll come back and finish off the shufflers to finish off the full Jumbo Revisited. If I've done this self-watching, I'll probably test it with Jum before moving on to shufflers, of course.

But after that? I'm glad you asked!

Numbo. After Jumbo comes Numbo, obviously, even though my wife has helpfully informed me that this would be leaving out the intermediate projects of Kumbo, Lumbo, and Mumbo. From an architectural standpoint, Numbo differs from Jumbo in a couple of ways, primarily due to the fact that it has a conceptual network. But it also maintains a small focus within the larger Workspace, and lately I've started to realize that the focus probably maps onto human working memory while the Workspace, which I had always thought was working memory, is probably best seen as episodic memory. This is unsettling, so I look forward to being able to settle it.

I also see Numbo as a warmup for Copycat, a far more involved domain. I want as much relevant experience as I can get before I tackle it.

Semantics and language. In November, I started reading Michael Dyer's early work, and after some thought I realized that his DYPAR parser is essentially a codelet system. Once I started down that conceptual path, I realized that my "research tier stack" was diverging from my earlier plan. In November, I wrote that I saw Tier 1 as the classical terraced scan, while Tier 2 would address self-watching, long-term memory with progressive recall, the use of framelets, and so on, because all of those felt more or less semantic in nature. But now I've just told you that self-watching and framelet use will be coming into Tier 1, so what's Tier 2? I think it's the way language allows us to work with semantics.

My first project on Tier 2, then, will probably be called NeoDYPAR and will attempt to use the terraced scan to process text on more or less the general plan of Dyer's work. I'm basically going straight to all the approaches from the 1980′s and trying to make them work on 2026 hardware. I feel like a time traveler, to be honest.

And that's the state of the FARG recap at the start of 2026. Let's see how far I get *this* year!

References

[1] D. R. Hofstadter, "The architecture of Jumbo," in *Proceedings of the International Machine Learning Workshop, June 22-24, 1983, Allerton House, Monticello, Illinois*, 1983.

[2] J. M. Roberts, "A Minimum Viable Product for the Parallel Terraced Scan," 2025.