Rigorous Nonsense

Contents

1	Math Isn't What You Think	3
6	On Hsu's Suggestivity and Idioms	6
7	What Does ⍋⍋ Even Mean	9
8	A Mundane Exotic Number System	12
9	Comments on Limits of Transformers on Compositionality	14
10	Implementing Brainfuck	16
17	Comments on Evaluating the World Model Implicit in a Generative Model	20
18	Advent of Code 2024, Day 13	22
19	Advent of Code 2024, Day 14	24
22	Readable Code is Unreadable	27
27	On 9P Semantics	33
31	Easy Random Trees	36

Math Isn't What You Think

Brandon Wilson
December 03, 2015

Math gets a bad rap.

To the non-specialist, when one says the word *math* most people seem to conjure up images of formulae and that one math teacher you had in high school who was particularly horrible.

This is really unfortunate, because at its heart, math is really about curiosity, honesty and collaboration.

Those sure don't seem like *mathy* terms, you say? Stick with me here.

2 Collaboration

In high school you might have memories of manipulating arbitrary symbols according to arbitrary rules, striving for *the* right answer. This is bullshit and isn't math. Seriously.

Practicing mathematicians do use formulae in daily work, but they are by far not the focus of attention. Rather, we strive to elucidate concepts and make them communicable to our peers. The symbols and formulae that fly around are more a byproduct of this process, honed and polished over time.

Actually, pick up a copy of Euclid's *The Elements* and try to read it. You'll get a pretty good idea of how math language has changed over the years. Here's an excerpt:

> If a first magnitude has to a second the same ratio as a third to a fourth, then any equimultiples whatever of the first and third also have the same ratio to any equimultiples whatever of the second and fourth respectively, taken in corresponding order.

Yes, it's pretty hard to make heads or tails of. In modern language, we'd say something like the following:

> Pick three fractions and call them a, b, and p. If $a = b$ then $pc \cdot a = pc \cdot b$.

While being more compact, I find it considerably more readable. Also, it might seem really obvious to you when written this way. If so, then great! That's the entire point of developing new notation. It's about getting the idea across in a clearer way.

This is the "collaboration" part of what I said above. Scary looking formulae and symbols, in a large part, are something we've honed over time to actually make things clearer and more communicable to our peers. Instead of a long, convoluted paragraph, why not just a quick one-liner?

3 Honesty

Making things clear like this also keeps one honest. For one it makes it considerably easier to find mistakes and oversights. Like in the example above, with the modern notation you might have thought

something along the lines of, "sure, just cancel the p on both sides". This is an excellent proof since it's simple and straight to the point.

Notably, this also applies in your own individual thinking. With a simple and clear way to express your idea, it's easier to think about it. And this is another main reason for the massive proliferation of technical jargon. For example, the word *multi-touch* would seem like gibberish to a person even a few decades ago. But now if you see a phone described as having a multi-touch screen, then you know that you can use it with all the fancy gestures like an iPhone. That word communicates all the ideas of pinch zoom, image rotation with a gesture, using multiple fingers to swipe and open a menu, *et cetera* with just a single word! None of those concepts are particularly difficult in and of themselves, it's just that it'd be a pain to have to explain every time instead of just using a single word.

So when the mathematician talks about "the pullback along the connection on a differentiable manifold" it *should* sound like gibberish. That is, until you spend a little time familiarizing yourself with the numerous, but not so difficult underlying concepts. If I were to unpack that statement into "everyday language" it would be a short book. If you had enough patience to read it, then you'd likely have no trouble understanding, but once you are comfortable with the notions, then we can talk about them easily with a simple word.

4 Curiosity

Which brings me to the last point. No collaboration or development of new concepts—no math—would get done without motivation. For practicing mathematicians the overwhelming motivation seems to simply be curiosity.

This is where primary education miserably fails with math. In schools it's all about teaching the prescribed rules to follow so you can get the prescribed answers to pass the prescribed standardized test. There do seem to be voices and teachers out there pushing to really engage students in STEM subjects, but math in particular seems to get short shift on institutional levels.

There's a thing called the Convention on the Organisation for Economic Co-operation and Development. It started in 1960 and currently has 34 countries as members. Semi-regularly the countries in this collaboration get together and perform a Programme for International Student Assessment. Basically, a "let's see how our students' academic performance compare" effort. The United States ranked 27$^{\text{th}}$ in 2012.

Now, there's a whole lot of room to quibble over these results, but one thing to take away is that it sure doesn't seem like a lot of interest in math and science is stirring in students in the states these days.

Which seems odd to me, 'cause as someone who studies math, it seems like pretty much everyone does math all the time without realizing it. If you've ever thought about how something works and, especially, if you've talked to someone about it, then you've done math. That's it.

The only real difference is that mathematicians in the field have put in the time to learn some specialized concepts and specialized symbols that turn out to be useful when you spend a lot of time thinking about how things work at really fundamental levels.

5 Takeaway

This is already a long and rambling post, written at one in the morning, but as a kickstart to my new blog, I wanted to argue that math isn't some esoteric practice for the math shamans, but instead

something inherently human that you and I can share in through these random and rambling musings here.

There's a lot more to be said, but for now I'll leave you with your own thoughts and your own mathings.

On Hsu's Suggestivity and Idioms

Brandon Wilson
July 24, 2023

Recently, Aaron Hsu posted a nice article [1] discussing a way in which APL's algebraic properties absolutely shine. The article ends up exhibiting two expressions—`⍸2<≠0,⍵` and `⍸2>≠⍵,0`—which respectively pick the first and last elements of groups of 1s in a boolean mask.

The punchline being how the simple algebraic change—< into >—corresponds to meaningful semantics. This is precisely what Iverson's "Notation as a Tool of Thought" is about, also echoing the "power of good notation" idea in 3Blue1Brown's Triangle of Power [2] video.

So, what about replacing < with an arbitrary boolean function? Hsu gives a nice table with the corresponding output, but the semantics of each are not spelled out. Paul Mansour makes a head start in his Boolean Techniques [3] article, but I think we can go all the way:

Y	0 1 1 1 0 0 0 0	0 1 1 1 0 0 0 1	1 1 1 1 0 0 0 0	1 1 1 1 0 0 0 1	Description
{2<≠0,⍵}¨Y	0 1 0 0 0 0 0 0	0 1 0 0 0 0 0 1	1 0 0 0 0 0 0 0	1 0 0 0 0 0 0 1	1-group head
{2<≠1,⍵}¨Y	0 1 0 0 0 0 0 0	0 1 0 0 0 0 0 1	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 1	1-group head, 1-boundary
{2<≠⍵,0}¨Y	1 0 0 0 0 0 0 0	1 0 0 0 0 0 1 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 1 0	0-group terminus
{2<≠⍵,1}¨Y	1 0 0 0 0 0 0 1	1 0 0 0 0 0 1 0	0 0 0 0 0 0 0 1	0 0 0 0 0 0 1 0	0-group terminus, 1-boundary
{2≤≠0,⍵}¨Y	1 1 1 1 0 1 1 1	1 1 1 1 0 1 1 1	1 1 1 1 0 1 1 1	1 1 1 1 0 1 1 1	not 0-group head
{2≤≠1,⍵}¨Y	0 1 1 1 0 1 1 1	0 1 1 1 0 1 1 1	1 1 1 1 0 1 1 1	1 1 1 1 0 1 1 1	not 0-group head, 1-boundary
{2≤≠⍵,0}¨Y	1 1 1 0 1 1 1 1	1 1 1 0 1 1 1 0	1 1 1 0 1 1 1 1	1 1 1 0 1 1 1 0	not 1-group terminus
{2≤≠⍵,1}¨Y	1 1 1 0 1 1 1 1	1 1 1 0 1 1 1 1	1 1 1 0 1 1 1 1	1 1 1 0 1 1 1 1	not 1-group terminus, 1-boundary
{2=≠0,⍵}¨Y	1 0 1 1 0 1 1 1	1 0 1 1 0 1 1 0	0 1 1 1 0 1 1 1	0 1 1 1 0 1 1 0	tail
{2=≠1,⍵}¨Y	0 0 1 1 0 1 1 1	0 0 1 1 0 1 1 0	1 1 1 1 0 1 1 1	1 1 1 1 0 1 1 0	tail, 1-boundary
{2=≠⍵,0}¨Y	0 1 1 0 1 1 1 1	0 1 1 0 1 1 0 0	1 1 1 0 1 1 1 1	1 1 1 0 1 1 0 0	not terminus
{2=≠⍵,1}¨Y	0 1 1 0 1 1 1 0	0 1 1 0 1 1 0 1	1 1 1 0 1 1 1 0	1 1 1 0 1 1 0 1	not terminus, 1-boundary
{2≠≠0,⍵}¨Y	0 1 0 0 1 0 0 0	0 1 0 0 1 0 0 1	1 0 0 0 1 0 0 0	1 0 0 0 1 0 0 1	head
{2≠≠1,⍵}¨Y	1 1 0 0 1 0 0 0	1 1 0 0 1 0 0 1	0 0 0 0 1 0 0 0	0 0 0 0 1 0 0 1	head, 1-boundary
{2≠≠⍵,0}¨Y	1 0 0 1 0 0 0 0	1 0 0 1 0 0 1 1	0 0 0 1 0 0 0 0	0 0 0 1 0 0 1 1	terminus
{2≠≠⍵,1}¨Y	1 0 0 1 0 0 0 1	1 0 0 1 0 0 1 0	0 0 0 1 0 0 0 1	0 0 0 1 0 0 1 0	terminus, 1-boundary
{2≥≠0,⍵}¨Y	1 0 1 1 1 1 1 1	1 0 1 1 1 1 1 0	0 1 1 1 1 1 1 1	0 1 1 1 1 1 1 0	not 1-group head
{2≥≠1,⍵}¨Y	1 0 1 1 1 1 1 1	1 0 1 1 1 1 1 0	1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 0	not 1-group head, 1-boundary
{2≥≠⍵,0}¨Y	0 1 1 1 1 1 1 1	0 1 1 1 1 1 0 1	1 1 1 1 1 1 1 1	1 1 1 1 1 1 0 1	not 0-group terminus
{2≥≠⍵,1}¨Y	0 1 1 1 1 1 1 0	0 1 1 1 1 1 0 1	1 1 1 1 1 1 1 0	1 1 1 1 1 1 0 1	not 0-group terminus, 1-boundary
{2>≠0,⍵}¨Y	0 0 0 0 1 0 0 0	0 0 0 0 1 0 0 0	0 0 0 0 1 0 0 0	0 0 0 0 1 0 0 0	0-group head
{2>≠1,⍵}¨Y	1 0 0 0 1 0 0 0	1 0 0 0 1 0 0 0	0 0 0 0 1 0 0 0	0 0 0 0 1 0 0 0	0-group head, 1-boundary
{2>≠⍵,0}¨Y	0 0 0 1 0 0 0 0	0 0 0 1 0 0 0 1	0 0 0 1 0 0 0 0	0 0 0 1 0 0 0 1	1-group terminus
{2>≠⍵,1}¨Y	0 0 0 1 0 0 0 0	0 0 0 1 0 0 0 0	0 0 0 1 0 0 0 0	0 0 0 1 0 0 0 0	1-group terminus, 1-boundary
{2∧≠0,⍵}¨Y	0 0 1 1 0 0 0 0	0 0 1 1 0 0 0 0	0 1 1 1 0 0 0 0	0 1 1 1 0 0 0 0	1-group tail
{2∧≠1,⍵}¨Y	0 0 1 1 0 0 0 0	0 0 1 1 0 0 0 0	1 1 1 1 0 0 0 0	1 1 1 1 0 0 0 0	1-group tail, 1-boundary
{2∧≠⍵,0}¨Y	0 1 1 0 0 0 0 0	0 1 1 0 0 0 0 0	1 1 1 0 0 0 0 0	1 1 1 0 0 0 0 0	1-group non-terminus
{2∧≠⍵,1}¨Y	0 1 1 0 0 0 0 0	0 1 1 0 0 0 0 1	1 1 1 0 0 0 0 0	1 1 1 0 0 0 0 1	1-group non-terminus, 1-boundary
{2⍲≠0,⍵}¨Y	1 1 0 0 1 1 1 1	1 1 0 0 1 1 1 1	1 0 0 0 1 1 1 1	1 0 0 0 1 1 1 1	not 1-group tail
{2⍲≠1,⍵}¨Y	1 1 0 0 1 1 1 1	1 1 0 0 1 1 1 1	0 0 0 0 1 1 1 1	0 0 0 0 1 1 1 1	not 1-group tail, 1-boundary
{2⍲≠⍵,0}¨Y	1 0 0 1 1 1 1 1	1 0 0 1 1 1 1 1	0 0 0 1 1 1 1 1	0 0 0 1 1 1 1 1	not 1-group non-terminus
{2⍲≠⍵,1}¨Y	1 0 0 1 1 1 1 1	1 0 0 1 1 1 1 0	0 0 0 1 1 1 1 1	0 0 0 1 1 1 1 0	not 1-group non-terminus, 1-boundary
{2∨≠0,⍵}¨Y	0 1 1 1 1 0 0 0	0 1 1 1 1 0 0 1	1 1 1 1 1 0 0 0	1 1 1 1 1 0 0 1	not 0-group tail
{2∨≠1,⍵}¨Y	1 1 1 1 1 0 0 0	1 1 1 1 1 0 0 1	1 1 1 1 1 0 0 0	1 1 1 1 1 0 0 1	not 0-group tail, 1-boundary
{2∨≠⍵,0}¨Y	1 1 1 1 0 0 0 0	1 1 1 1 0 0 1 1	1 1 1 1 0 0 0 0	1 1 1 1 0 0 1 1	not 0-group non-terminus
{2∨≠⍵,1}¨Y	1 1 1 1 0 0 0 1	1 1 1 1 0 0 1 1	1 1 1 1 0 0 0 1	1 1 1 1 0 0 1 1	not 0-group non-terminus, 1-boundary
{2⍱≠0,⍵}¨Y	1 0 0 0 0 1 1 1	1 0 0 0 0 1 1 0	0 0 0 0 0 1 1 1	0 0 0 0 0 1 1 0	0-group tail
{2⍱≠1,⍵}¨Y	0 0 0 0 0 1 1 1	0 0 0 0 0 1 1 0	0 0 0 0 0 1 1 1	0 0 0 0 0 1 1 0	0-group tail, 1-boundary
{2⍱≠⍵,0}¨Y	0 0 0 0 1 1 1 1	0 0 0 0 1 1 0 0	0 0 0 0 1 1 1 1	0 0 0 0 1 1 0 0	0-group non-terminus
{2⍱≠⍵,1}¨Y	0 0 0 0 1 1 1 0	0 0 0 0 1 1 0 0	0 0 0 0 1 1 1 0	0 0 0 0 1 1 0 0	0-group non-terminus, 1-boundary
{2⊢≠0,⍵}¨Y	0 1 1 1 0 0 0 0	0 1 1 1 0 0 0 1	1 1 1 1 0 0 0 0	1 1 1 1 0 0 0 1	id
{2⊢≠1,⍵}¨Y	0 1 1 1 0 0 0 0	0 1 1 1 0 0 0 1	1 1 1 1 0 0 0 0	1 1 1 1 0 0 0 1	id
{2⊢≠⍵,0}¨Y	1 1 1 0 0 0 0 0	1 1 1 0 0 0 1 0	1 1 1 0 0 0 0 0	1 1 1 0 0 0 1 0	left-shift, 0-insert
{2⊢≠⍵,1}¨Y	1 1 1 0 0 0 0 1	1 1 1 0 0 0 1 1	1 1 1 0 0 0 0 1	1 1 1 0 0 0 1 1	left-shift, 1-insert
{2⊣≠0,⍵}¨Y	0 0 1 1 1 0 0 0	0 0 1 1 1 0 0 0	0 1 1 1 1 0 0 0	0 1 1 1 1 0 0 0	right-shift, 0-insert
{2⊣≠1,⍵}¨Y	1 0 1 1 1 0 0 0	1 0 1 1 1 0 0 0	1 1 1 1 1 0 0 0	1 1 1 1 1 0 0 0	right-shift, 1-insert

```
{2⊣≠ω₇0}¨Y   0 1 1 1 0 0 0 0  0 1 1 1 0 0 0 1  1 1 1 1 0 0 0 0  1 1 1 1 0 0 0 1  id
{2⊣≠ω₇1}¨Y   0 1 1 1 0 0 0 0  0 1 1 1 0 0 0 1  1 1 1 1 0 0 0 0  1 1 1 1 0 0 0 1  id
```

It is interesting to note that the naming scheme can be thought of as a simple re-parameterization of the algebraic phrases. In particular, we have

$$\begin{aligned} 48 &= \#\text{functions} \\ &= B \cdot e \cdot d \\ &= 12 \cdot 2 \cdot 2 \end{aligned}$$

where

- B denotes the 12 count of binary functions you chose, which ignores the other four binary functions, 0⍨, 1⍨, ∼⍨⊢, and ∼⍨⊣,
- e denotes the 2 choices, 0 or 1, of edge conditions, and
- d denotes the 2 choices, pre- or post-pending, of direction.

Whereas the semantic descriptions partition the collection differently:

$$\begin{aligned} 48 &= \#\text{functions} \\ &= g \cdot p \cdot d \cdot b \cdot i + A + S + I \\ &= 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 + 8 + 4 + 4 \end{aligned}$$

where

- g denotes the choice of selecting on either 0- or 1-groups,
- p denotes the selecting the group head or tail,
- d denotes the choice of "direction", i.e. forwards selects head or tail and reverse selects terminus or non-terminus,
- b denotes the choice of 0 or 1 boundary condition,
- i denotes the choice of whether selection is inverted or not,
- A is the class of group-agnostic functions,
- S is the special class of shift functions, and
- I is the quadruple-counted identity function.

Just for fun, we can actually complete the table above, with the missing boolean functions:

```
Y              0 1 1 1 0 0 0 0  0 1 1 1 0 0 0 1  1 1 1 1 0 0 0 0  1 1 1 1 0 0 0 1
{2(0⍨)≠ω₇0}¨Y  0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0  constant 0
{2(0⍨)≠ω₇1}¨Y  0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0  constant 0
{2(0⍨)≠0₇ω}¨Y  0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0  constant 0
{2(0⍨)≠1₇ω}¨Y  0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0  constant 0
{2(1⍨)≠ω₇0}¨Y  1 1 1 1 1 1 1 1  1 1 1 1 1 1 1 1  1 1 1 1 1 1 1 1  1 1 1 1 1 1 1 1  constant 1
{2(1⍨)≠ω₇1}¨Y  1 1 1 1 1 1 1 1  1 1 1 1 1 1 1 1  1 1 1 1 1 1 1 1  1 1 1 1 1 1 1 1  constant 1
{2(1⍨)≠0₇ω}¨Y  1 1 1 1 1 1 1 1  1 1 1 1 1 1 1 1  1 1 1 1 1 1 1 1  1 1 1 1 1 1 1 1  constant 1
{2(1⍨)≠1₇ω}¨Y  1 1 1 1 1 1 1 1  1 1 1 1 1 1 1 1  1 1 1 1 1 1 1 1  1 1 1 1 1 1 1 1  constant 1
{2∼⍨⊢≠ω₇0}¨Y   0 0 0 1 1 1 1 1  0 0 0 1 1 1 0 1  0 0 0 1 1 1 1 1  0 0 0 1 1 1 0 1  invert left-shift, 1-insert
{2∼⍨⊢≠ω₇1}¨Y   0 0 0 1 1 1 1 0  0 0 0 1 1 1 0 0  0 0 0 1 1 1 1 0  0 0 0 1 1 1 0 0  invert left-shift, 0-insert
{2∼⍨⊢≠0₇ω}¨Y   1 0 0 0 1 1 1 1  1 0 0 0 1 1 1 0  0 0 0 0 1 1 1 1  0 0 0 0 1 1 1 0  invert
{2∼⍨⊢≠1₇ω}¨Y   1 0 0 0 1 1 1 1  1 0 0 0 1 1 1 0  0 0 0 0 1 1 1 1  0 0 0 0 1 1 1 0  invert
{2∼⍨⊣≠ω₇0}¨Y   1 0 0 0 1 1 1 1  1 0 0 0 1 1 1 0  0 0 0 0 1 1 1 1  0 0 0 0 1 1 1 0  invert
{2∼⍨⊣≠ω₇1}¨Y   1 0 0 0 1 1 1 1  1 0 0 0 1 1 1 0  0 0 0 0 1 1 1 1  0 0 0 0 1 1 1 0  invert
{2∼⍨⊣≠0₇ω}¨Y   1 1 0 0 0 1 1 1  1 1 0 0 0 1 1 1  1 0 0 0 0 1 1 1  1 0 0 0 0 1 1 1  invert right-shift, 1-insert
{2∼⍨⊣≠1₇ω}¨Y   0 1 0 0 0 1 1 1  0 1 0 0 0 1 1 1  0 0 0 0 0 1 1 1  0 0 0 0 0 1 1 1  invert right-shift, 0-insert
```

Which makes it manifest why they can be safely ignored.

Bibliography

[1] A. Hsu, “Suggestivity and Idioms in APL,” Fastidious Elegance. [Online]. Available: https://www.sacrideo.us/suggestivity-and-idioms-in-apl/

[2] G. Sanderson, *Triangle of Power*, (Jun. 25, 2016). [Online Video]. Available: https://www.youtube.com/watch?v=sULa9Lc4pck

[3] P. Mansour, “Boolean Techniques,” Tool of Thought. [Online]. Available: https://www.toolofthought.com/posts/boolean-techniques

What Does ⍋⍋ Even Mean

Brandon Wilson
August 04, 2023

A while back I was casually perusing the articles on Jôt Dọt T̆imes [1] and fell in love with almost everything that Paul Mansour writes up. In his Grade Down of Grade Up [2], he shares a quote that just feels like a challenge:

> So the conclusion is that if we are only dealing in permutations ⍒⍋ is worthless since it's simply a slower ⌽, and if we are not dealing in permutations even Adám doesn't know what ⍒⍋ does so it is probably equally worthless.

Mansour's answer is a really nice special application, but it turns out that there's actually a satisfying general answer! We will consider general arrays, but in the back of our mind let's keep a concrete example in mind, something like this:

```
      ⊢Y←⎕UCS 97+?10⍴26
lnpwbkmcxt
      ↑Y (⍋Y)
l n p w b k m c x t
4 7 5 0 6 1 2 9 3 8
```

Defining a couple terms up front will make our life much easier:

- An *item* of an array is a sub-array along its first axis. In particular, for a vector, the items correspond with the elements of the array, and for a matrix the items are the rows; and
- The *rank* of an item $y \in Y$ is the index of y when you sort Y ascendingly. Specifically, this is TAO [3] order. So if y has rank i, then that just means that y is the i^{th} smallest element of Y.

Applying these to the definition of Grade Up [4], we can say that the i^{th} element of ⍋Y simply indexes whichever item of Y has rank i. With a little squinting, we also see that this is equivalent to the reference definition:

> R is an integer vector being the permutation of ⍳1↑⍴Y that places the sub-arrays along the first axis in ascending order.

Let us add a little bit of notation. When the i^{th} item of Y has rank j, we write $< i, j >$. Notice then that for ⍋Y the condition $< j, i >$ holds—in other words, ⍋ just swaps i and j, meaning that ⍋⍋Y leaves $< i, j >$ unchanged! When Y is a permutation vector, this makes ⍋⍋ the identity, but in general we can think of it as the function `Rank`, since it produces a vector of the item ranks.

Better yet, since ⍒ is equal to ⌽⍋, we can say that $< N - j, i >$ holds for its result, where `N=¯1+≢Y` (in `⎕IO←0`), and if we think of i, j as living in $\mathbb{Z}_N$, this becomes even nicer: $< -j, i >$. Indeed,

```
      ↑Y (⍋⍋Y)
l n p w b k m c x t
3 5 6 8 0 2 4 1 9 7
```

Effectively, the negative sign in our $< \cdot, \cdot >$ symbol just means that the corresponding field starts counting from the back of the array, which we can think of as the function `ReverseRank`. Armed with this little bit of algebra, we can now fully characterize the fourfold combinations of ⍋ and ⍒:

Input		**Output**	**Interpretation**
$< i, j >$	⍋⍋	$< i, j >$	Ascending `Rank`
$< i, j >$	⍋⍒	$< i, -j >$	Ascending `ReverseRank`
$< i, j >$	⍒⍋	$< -i, j >$	Descending `Rank`
$< i, j >$	⍒⍒	$< -i, -j >$	Descending `ReverseRank`

Not bad! So the leading Grade chooses ascending or descending and the trailing Grade chooses `Rank` or `ReverseRank`. Applying to our example,

```
      ↑Y (⍋⍋Y) (⍋⍒Y) (⍒⍋Y) (⍒⍒Y)
l n p w b k m c x t
3 5 6 8 0 2 4 1 9 7
6 4 3 1 9 7 5 8 0 2
7 9 1 4 2 0 8 6 5 3
2 0 8 5 7 9 1 3 4 6
```

As a sanity check, note that `Rank + ReverseRank` should be constant. Indeed, for each item $y \in Y$, the number of items less than y plus the number of items greater than y should equal the total number of items in Y, minus 1 for y itself:

```
      (⍋∘⍋+⍋∘⍒)Y
9 9 9 9 9 9 9 9 9 9
      (⍒∘⍋+⍒∘⍒)Y
9 9 9 9 9 9 9 9 9 9
```

Actually, there is a bit of nuance when items of Y are equal. `Rank + ReverseRank` will not be constant because both ⍋ and ⍒ rank equal items the same way, no directionality involved. This property makes having both primitives useful, otherwise `(⍋≡⌽∘⍒)Y` would always hold. This property is exactly what Paul Mansour exploits in his article above with the `AverageRank` function.

Bibliography

[1] "The APL Sign of the Times." [Online]. Available: https://apl.news/

[2] P. Mansour, "Grade Down of Grade Up," Tool of Thought. [Online]. Available: https://www.toolofthought.com/posts/grade-up-of-grade-down

[3] "Total array ordering." [Online]. Available: https://aplwiki.com/wiki/Total_array_ordering

[4] “Grade Up (Monadic).” [Online]. Available: https://help.dyalog.com/latest/#Language/Primitive%20Functions/Grade%20Up%20Monadic.htm

A Mundane Exotic Number System

Brandon Wilson
August 29, 2023

Numeral systems contain a surprising amount of depth. Our familiarity and monolingualism with the base 10 positional system runs so deep that we tend to think of numbers as *being* their base 10 representation. Doing this actually has merit, given the simple algorithms for arithmetic it offers. However, other representations often provide interesting insights that seem completely opaque otherwise.

Take base 2, for example. The correspondence between sets of things and base 2 numbers can be mined for all sorts of interesting gems, given their correspondence with subsets of some set. Also of note is that the rows of Pascal's triangle sum to powers of two. If this is new to you, it's tons of fun to puzzle out why and how it's related to base 2.

A more exotic example is the factorial number system [1], wherein each position represents a multiple of a factorial—as opposed to the normal power of some base. Given that permutations are counted by the factorials, this numeral system provides a really direct enumeration for permutations.

Another system that pops up in surprisingly mundane places is the bijective numeration [2]. Excel, for example, uses letters to label its columns, like `A`, `AZ`, `GDJ`, *etc.* At first blush, this appears to be a straightforward base 26 system, and indeed there's a very simple APL expression to convert from column names to their index:

```
      (26⊥1+⎕A⍳⊢)¨'A' 'AZ' 'ZZ' 'GDJ'
1 52 702 4846
```

which *feels* like it's just doing base 26. However, the intuitive inverse expression fails:

```
      (⊂⍞⎕A⍨)∘(26⊥⍣¯1⊢)¨1 52 702 4846
 B  CA  BBA  HEK
```

There are actually two issues with the above. We're off by one in the index into `⎕A`,

```
      (⊂⍞⎕A⍨)∘(¯1+26⊥⍣¯1⊢)¨1
 A
```

however, when `⊥⍣¯1` decodes into digits with 0, we get `¯1` indices:

```
      (⊂⍞⎕A⍨)∘(¯1+26⊥⍣¯1⊢)¨26
INDEX ERROR
      (⊂⍞⎕A⍨)∘(¯1+26⊥⍣¯1⊢)¨26
                          ∧
```

In general, the standard representation in a base-k positional system has the same representation in the base-k bijection numeration as long as the former contains no zeroes. In the case when there are

zeroes, the bijective system needs to (recursively) borrow from the left, for essentially the same reason that borrowing happens in the pen-and-paper subtraction algorithm.

```
     26{⎕A⍞⍨⊂⍣¯1+⍺{
      n←⍵+(⍵×⍺)-0,⍨1↓⍵←0=⍵    ⍝ Borrow
      (∨\0≠n)/n               ⍝ Trim leading zeroes
     }⍣≡(⍺⊥⍣¯1⊢)⍵}¨1 52 702 4846
 A  AZ  ZZ  GDJ
```

While the above is a fully general solution, IMHO, its verbosity leaves a somewhat unsatisfying aftertaste. The borrow algorithm is ubiquitous, so there is likely some more parsimonious expression.

Paul Mansour shares a completely different approach in his article Excel Column Names [3], which leverages Excel's maximum column count to enumerate all names and simply index into them. Our algorithm above is probably $O(\log n)$, so if you were performing this conversion a whole lot, having a cached index likely performs vastly better.

Thanks for reading.

Bibliography

[1] "Factorial number system." [Online]. Available: https://en.wikipedia.org/wiki/Factorial_number_system

[2] "Bijective numeration." [Online]. Available: https://en.wikipedia.org/wiki/Bijective_numeration

[3] P. Mansour, "Excel Column Names," Tool of Thought. [Online]. Available: https://www.toolofthought.com/posts/excel-column-names

Comments on Limits of Transformers on Compositionality

Brandon Wilson
January 12, 2024

The first AI Safety Tokyo benkyoukai of 2024 covered Limits of Transformers on Compositionality [1], presented by Blaine Rogers [2]. Here I present my synthesis of how our group understood the paper within its larger context.

The topical paper frames itself as a counterpoint to the causal probing paper [3]. In particular its authors present an argument that LLMs lean more heavily toward the "memorize everything" side rather than the "true understanding" one. It stands as part of the larger stochastic parrot [4] debate.

Personally, I believe that interpretation to be overly broad. The team comes up with a specific operationalization of what it means for LLMs to generalize a problem out of distribution, boiling down to a particular incarnation of mathematical induction [5]. It's actually quite a brilliant operationalization, being relatively mathematically precise, easy to work with, and even intuitively fits into the intuition for "generalizing to a larger domain".

Specifically, the research team picks three standard toy problems that scale in size along a single, discrete parameter, *i.e.* multiplication of *n*-digit numbers, Einstein's puzzle [6] over *n* conditions, and a specific dynamic programming [7] problem that scales with complexity [8] *O(n)*.

The team then throws 90 kilo-USD at fine-tuning GPT 3 and 4 on each problem up to size *n*, specifically training with and without step-by-step solutions. Think what your math teacher wanted when they asked you to "show your work". The punchline drops in Figure 5 on page 6 which shows that the prediction accuracy remains high on problems up to size *n* and then precipitously drops for *n+1* and larger. This can be read as the models simply memorizing the example set domain.

My interpretation: This paper adds significant evidence that current models don't encode some generalized induction pattern, at least not one that's readily invoked with reasonable prompts. However, that's the limit of what it says. The limitations are threefold: 1) it doesn't address other forms of inductive reasoning, 2) it doesn't give insight into grokking [9], and 3) the GPT 3-based models are quite weak compared to frontier ones.

However, one really cool bonus analysis they share shows up in section 3.2.3. They present evidence about why LLMs might have trouble with deeply-chained reasoning, like large multiplication problems. Figure 7 shows that a significant portion of correct answers result from internally flawed reasoning with mistakes that just happen to "cancel each other out".

One can view consistently correct answers but with flawed reasoning as a kind of non-deductive learning separate from memorization. However, at the very least, they unlocked a very nice tool to give higher resolution beyond simply right vs. wrong answers.

Bibliography

[1] N. Dziri *et al.*, "Faith and Fate: Limits of Transformers on Compositionality," May 09, 2023. [Online]. Available: https://arxiv.org/abs/2305.18654

[2] B. Rogers, “Blaine Rogers.” [Online]. Available: https://www.linkedin.com/in/blainewilliamrogers/

[3] K. Li, A. H. K., D. Bau, F. Viégas, H. Pfister, and M. Wattenberg, “Emergent World Representations: Exploring a Sequence Model Trained on a Synthetic Task,” Oct. 24, 2022. [Online]. Available: https://arxiv.org/abs/2210.13382

[4] “Stochastic parrot.” [Online]. Available: https://en.wikipedia.org/wiki/Stochastic_parrot

[5] “Mathematical induction.” [Online]. Available: https://en.wikipedia.org/wiki/Mathematical_induction

[6] “Zebra Puzzle.” [Online]. Available: https://en.wikipedia.org/wiki/Zebra_Puzzle

[7] “Dynamic programming.” [Online]. Available: https://en.wikipedia.org/wiki/Dynamic_programming

[8] “Time complexity.” [Online]. Available: https://en.wikipedia.org/wiki/Time_complexity

[9] A. Power, Y. Burda, H. Edwards, I. Babuschkin, and V. Misra, “Grokking: Generalization Beyond Overfitting on Small Algorithmic Datasets,” Jan. 06, 2022. [Online]. Available: https://arxiv.org/abs/2201.02177

Implementing Brainfuck

Brandon Wilson
September 27, 2024

APL is a pretty great language for quickly iterating on program architecture. In particular, here is a quick comparison of a couple brainfuck implementations.

The brainfuck virtual machine is essentially a minimal Harvard architecture [1]: we have an instruction space, a data tape, and indices into each. If you are unfamiliar with the (tiny!) specification, take a look at brainfuck's official website [2].

Anyway, we start our implementation by initializing state:

```
run←{
 ⎕IO⍟←0 ⋄ n t b⍟←0 (1e6⍴0) 0 ⋄ prg←⍵              ⍝ Data index, tape, and buffer
 _←2{p[⍵]←α[α⍳⍵]}⌿⊢⍟⊂⍸¯1⌽+⍀1 ¯1 0['[]'⍳⍵⊣p⍟←⍳≠⍵]  ⍝ Parent vector for brackets
 _←⍵ αα⍣{⍵=¯1+≠prg}0                             ⍝ Run instructions from 0 to last
}
```

This operator runs single-instruction executors recursively over the full instruction input, which lets us plug and play different implementations of the execution unit.

11 Version 0

One obvious implementation mirrors the language specification directly:

```
v0←{
 α[⍵]='>':⍵+1⊣n+←1
 α[⍵]='<':⍵+1⊣n-←1
 α[⍵]='+':⍵+1⊣t[n]+←1
 α[⍵]='-':⍵+1⊣t[n]-←1
 α[⍵]='.':⍵+1⊣⎕←⎕UCS t[n]
 α[⍵]=',':⍵+1⊣t[n]←⍞
 (α[⍵]='[')∧t[n]≠0:⍵+1
 (α[⍵]='[')∧t[n]=0:⊃⌽⍳p=⍵
 (α[⍵]=']')∧t[n]=0:⍵+1
 (α[⍵]=']')∧t[n]≠0:p[⍵]
}
```

This essentially gives us a big switch statement that executes each instruction. Squinting our eyes a bit, this architecture is spiritually in the same family as an LR parser [3], since it fundamentally is limited to a small (1 token) parse window.

Since brainfuck has jumps with `[` and `]` instructions, our single-instruction executors need to also update the instruction pointer address. We're managing this state by taking the address in `⍵` and returning the address of the next instruction to execute.

The repetition of `⍵+1` is a tad ugly, and it really seems like we should be able to merge the `n+←1` and `n-←1` cases. Ditto for some of the other structural repetitions. For example, a strategy like the following lets us merge `<` and `>` cases:

```
2>i←a[⍵]∊'<>': ⍵+1⊣n+←¯1 1[i]
```

12 Version 1

Very similar code works for `+` and `-` as well; however, the similarity between `<>` and `+-` cases also begs for further fusion. And what about bracket instructions and `⍵+1` repetition?

Staring at the problem for a bit, we may notice that fusing instruction cases sometimes requires updating multiple state variables in a single case. This suggests an approach that updates all state in a single "case". Said another way, we can transpose how we think about the relationship between instructions and state. Instead of dispatching a specific state update per instruction, we can holistically update all state at once:

```
v1←{
 n+←¯1 1 0['<>'⍳⍺[⍵]]                              ⍝ Update data index
 t[n]+←¯1 1 0['-+'⍳⍺[⍵]]                           ⍝ Update data cell
 _←{⍞←⎕UCS⍵}⍣(⍺[⍵]='.')⊢t[n]                       ⍝ Handle output
 _←{t[n] b⍬←1 1⊂{⎕UCS⍞}⍣(b≡⍬)⊢b}⍣(⍺[⍵]=',')⊢⍬    ⍝ Handle input
 1+(3 2⍴⍵(⊃⍵⍳⍨p=⍵)(p[⍵])⍵ ⍵ ⍵)['[]'⍳⍺[⍵];t[n]=0]  ⍝ Update instruction index
}
```

Not only is the code shorter, but now it has us thinking in a data-first paradigm, compared to the more procedural flavor of `v0`. Indeed, each line of `v1` corresponds to a single state variable, modulo input/output handling.

13 A Desideratum

Speaking of input/output, notice the change in our input handler. Brainfuck's I/O instructions respectively read and write single characters at a time. However, `⍞` accepts input up until a newline, and brainfuck applications in the wild also typically process line-oriented input. By maintaining a buffer of one input line in `b` and only calling `⍞` when `b` is empty, we get much improved input UX.

Backporting this to `v0` changes the line

```
⍺[⍵]=',':⍵+1⊣t[n]←⍞
```

into

```
⍺[⍵]=',':⍵+1⊣t[n] b⍬←1 1⊂{⎕UCS⍞}⍣(b≡⍬)⊢b
```

14 Performance

How do these implementations stack up in terms of execution performance?

On the one hand, apart from the case-detection, `v0` executes only enough code to execute the current instruction, while `v1` is virtually constant time for every instruction. On the other hand, perhaps the guard statements in `v0` overwhelm any such advantage.

As a simple first-order comparison, we use the Hello World [4] example provided on Wikipedia:

```
    PRG ←'++++++++[>++++[>++>+++>+++>+<<<<-]>+>+>->>+[<]<-]>>.>---.+++++++.'
    PRG,←'.+++.>>.<-.<.+++.------.--------.>>+.>++.'
    cmpx 'v0 run PRG' 'v1 run PRG'
bf_v0 PRG → 1.2E¯1 |    0% □□□□□□□□□□□□□
bf_v1 PRG → 3.7E¯1 | +198% □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
```

15 Future Exploration

It's also tempting to write a brainfuck compiler. In fact, there are obvious opportunities for instruction fusion and parallel execution. Apart from its parsimony, what stands out to me with the `v1` architecture is that it doesn't bake in the "single instruction at a time" constraint, making data-parallelism more manifestly accessible.

Despite the above being ostensibly about a toy implementation, the procedural vs. data-first architectural difference touches on some deep topics. More serious projects that lean heavily on these ideas include my dayaml [5] parser, the Co-dfns [6] compiler, and data-parallel proof verification [7].

All this merits much more discussion, but that's whole 'nother story.

16 Acknowledgements

Many thanks to Adám Brudzewsky [8] for taking the time to read my code and suggest several spelling improvements in expressions. The depth of his knowledge, easy facility with APL, and sharp wit are always mutually impressive and a joy.

Bibliography

[1] "Harvard architecture." [Online]. Available: https://en.wikipedia.org/wiki/Harvard_architecture

[2] D. B. Christofani, "some brainfuck fluff." [Online]. Available: https://brainfuck.org/

[3] "LR parser." [Online]. Available: https://en.wikipedia.org/wiki/LR_parser

[4] "Brainfuck, Hello World!." [Online]. Available: https://en.wikipedia.org/wiki/Brainfuck#Hello_World!

[5] B. Wilson, *dayaml: YAML parser for Dyalog APL*. [Online]. Available: https://github.com/xelxebar/dayaml

[6] A. Hsu, *Co-dfns: High-performance, Reliable, and Parallel APL*. [Online]. Available: https://github.com/Co-dfns/Co-dfns/

[7] B. Wilson, "D14: Data Parallel Proof Verification in APL." Sep. 18, 2024. [Online]. Available: https://www.dyalog.com/user-meetings/dyalog24.htm#D14

[8] "Adám Brudzewsky." [Online]. Available: https://aplwiki.com/wiki/Ad%C3%A1m_Brudzewsky

Comments on Evaluating the World Model Implicit in a Generative Model

Brandon Wilson
December 11, 2024

Vafa *et al* [1] released a paper earlier this year covering a particularly lucid operationalization of the question, "Do LLMs embed a world model?" On the one hand, we have an LLM that literally generates interactive 3D worlds [2], dropped by DeepMind earlier this week. But on the other hand, LLMs are notorious for wildly unrealistic behavior on out-of-distribution inputs [3].

The central insight is that something called the Myhill-Nerode theorem [4] lets us *reconstruct* a world model simply from observing token sequences. Slightly more precisely, it provides sharp conditions such that, *simply by observing output behavior*, we know that some given computational system such as an LLM must be equivalent to a canonical description. In other words, in this special case, no matter how inscrutable the matrices may be, as long as the conditions are met, then the matrices must be isomorphic to a standard, human-style solution.

The details are absolutely fascinating, and I highly encourage you to read the original paper. Here I would like to poke at the edges of the argument and its particular implementation by Vafa *et al.*

Not only is the theorem application inspired, but since it applies to deterministic finite automata, we nicely restrict the domain of "world models" to a class of (computational) outputs that are particularly tractable, *cf* DFAs [5] and the Chomsky hierarchy [6].

And the golden goose egg is in section 2.6, where the authors define their evaluation metrics, *boundary precision* and *boundary recall*. It takes a bit of work to unwind the math, but they boil down to the fact that an LLM can be wrong about world states in precisely two ways: 1) it can be missing information about a state; or 2) it can encode counterfactual information about a state.

After unwinding definitions, it turns out that *boundary recall* provides a measure of the former and *boundary precision* the latter.

The magic of Myhill-Nerode, however, is that it lets us completely ignore the *content* of a world state, which is mostly unobservable in LLMs right now, and just focus on how one state can or cannot transition to another, *i.e.* their connectivity. Specifically, the connectivity is encoded in a DFA's transition graph or an LLM's set of all possible token sequences.

The authors then use these nicely set up tools to evaluate models of their own training. This is where the paper unfortunately loses some of its water-tightness. One obstacle is that we have no way of obtaining such a set of "all possible token sequences" for an LLM and must resort to stochastic sampling.

The paper finds that the reconstructed world models of their LLMs behave poorly under their *recall* and *precision* metrics. However, initial definitions didn't explore how expectation values of *recall* and *precision* behave under stochastic sampling. I don't have a clear idea what the expectation values should be. Coarsely, I believe that we should *a priori* expect to underestimate *recall* and overestimate *precision*, but the extent of this effect, especially relative to their particular problems and sampling methods, remains to be investigated.

A more parochial issue is that only one of the examined problems was measured relative to a breadth of multiple LLMs. In particular, the initial taxi route problem only investigated a single, small self-trained model.

Such a good set of scalpels deserves much broader application, IMHO!

Bibliography

[1] K. Vafa, J. C. Y., A. Rambachan, J. Kleinberg, and S. Mullainathan, "Evaluating the World Model Implicit in a Generative Model," Jun. 06, 2024. [Online]. Available: https://arxiv.org/abs/2406.03689

[2] J.-H. Parker *et al.*, "Genie 2: A large-scale foundation world model," Dec. 04, 2024. [Online]. Available: https://deepmind.google/blog/genie-2-a-large-scale-foundation-world-model/

[3] L. Huang *et al.*, "A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions," Sep. 09, 2023. [Online]. Available: https://arxiv.org/abs/2311.05232

[4] "Myhill–Nerode theorem." [Online]. Available: https://en.wikipedia.org/wiki/Myhill%E2%80%93Nerode_theorem

[5] "Deterministic finite automaton." [Online]. Available: https://en.wikipedia.org/wiki/Deterministic_finite_automaton

[6] "Chomsky hierarchy." [Online]. Available: https://en.wikipedia.org/wiki/Chomsky_hierarchy

Advent of Code 2024, Day 13

Brandon Wilson
December 14, 2024

This problem [1] is a standard basis transformation from linear algebra. To see this, consider a specialization to one dimension:

$$P = NA$$

where P is the prize x-coordinate, A is the offset vector of the button and N is the solution to the number of presses. Solving for N is trivial:

$$N = \frac{P}{A}$$

The magic of linear algebra is that the same algebra essentially holds when we move to higher dimensions as well; we just consider division to be multiplication by a matrix inverse.

Here is the full solution:

```
n←⊃⊢⍎⍠⍎⍠⎕VFI(⊃⎕NGET'aoc2024-13.txt'0)⊃⎕D⍷' '
ab←(320 2 2)⍴msk≠n⊣⍴z←(320 2)⍴n≠⍨~msk←(≢n)⍴1 1 1 1 0 0
+⌿(⊢×⊢=⌊)3 1+.×⍉⍪z⌹⍤1 2⊢(0 2 1)⍉ab
```

Below, I sketch my thought process:

Typical APL data ingestion starts with a data normalization pass. In this case, the input format is very regular, embedding integers in a pre-defined order, letting us leverage `⎕VFI` [2] to precisely extract the numeric data:

```
n←⊃⊢⍎⍠⍎⍠⎕VFI(⊃⎕NGET'aoc2024-13.txt'0)⊃⎕D⍷' '
```

This gives us an integer vector that is logically grouped into units of six: the first four integers define the A-B coordinates which provide the 2×2 basis transformation matrix, and the last two integers, which define coordinates for the prize.

```
      pz←(320 2)⍴n⌿⍨~msk←(≠n)⍴1 1 1 1 0 0
      ab←(320 2 2)⍴msk⌿n

      3↑ab
75 30
57 75

48 12
20 52

20 86
46 33

      3↑pz
9885 8130
1320 5260
 370  767
```

Notice the hard-coded numbers: Logically, we have 320 records, each containing a 2×2 matrix and a 2 -element vector. These serve as a nice visual signal that we are only concerned with one-off static data, instead of a more complex problem domain requiring particular kinds of generalization. Specifically, we're using an inverted table [3] format but storing columns as type-homogenous variables instead of type-homogenous rows in a single array.

After transposing the A-B matrices, our data is nicely normalized such that the solution simply becomes a direct matrix division [4]:

```
      +⌿(⊢×⊢=⌊)3 1+.×⍉⍉pz⌸⍤1 2⊢(0 2 1)⍉ab
```

How nice! This is a toy problem, but the utility of data normalization holds true in real systems as well. That is a writeup for another day, however.

Bibliography

[1] E. Wastl, "Advent of Code 2024, Day 13." [Online]. Available: https://adventofcode.com/2024/day/13

[2] "Verify & Fix Input." [Online]. Available: https://help.dyalog.com/19.0/#Language/System%20Functions/vfi.htm

[3] "Essays/Inverted Table." [Online]. Available: https://code.jsoftware.com/wiki/Essays/Inverted_Table

[4] "Matrix Divide." [Online]. Available: https://help.dyalog.com/19.0/#Language/Primitive%20Functions/Matrix%20Divide.htm

Advent of Code 2024, Day 14

Brandon Wilson
December 14, 2024

Day 14 [1] is another linear algebra problem with some classification spice thrown on top:

```
n←¯1↓⊃⊢⍞≠⍤≠(', '⍪LF)⎕VFI'¯'@{⍵='-'}(⊃⎕NGET'aoc.txt'0)∩⎕D⍪'-, '⍪LF←⎕UCS 10
p←msk≠m⊣v←(~msk←1000⍴1 0)≠m←1000 2⍴n
r←101 103|⍞1⊢p+100×v
×≠(50 51>⍞1⊢r)⊢∘≢⌸r⊣r≠⍨←∧/50 51≠⍞1⊢r
```

The input data is helpfully regular, *i.e.* all information about integers can be inferred from their index, so we take the standard approach of eliding everything but integers and separators, feeding the result to `⎕VFI` [2]:

```
      ⊢n←¯1↓⊃⊢⍞≠⍤≠(', '⍪LF)⎕VFI'¯'@{⍵='-'}(⊃⎕NGET'aoc.txt'0)∩⎕D⍪'-, '⍪LF←⎕UCS 10
67 43 80 86 47 24 ¯9 ¯96 6 38 10 81 73 92 36 ¯93 74 10 ¯35 19 57 59 ¯5 71 ⋯
```

We know the integers must encode pairs of position and velocity vectors, which is straightforward to extract:

```
      p←msk≠m⊣v←(~msk←1000⍴1 0)≠m←1000 2⍴n
      5↑p,'|',v
67 43 |  80  86
47 24 |  ¯9 ¯96
 6 38 |  10  81
73 92 |  36 ¯93
74 10 | ¯35  19
```

Now we get to the meat. Conceptually, we just calculate an affine offset on a torus (*i.e.* modulo the circumferences), which is nicely reflected directly in the code:

```
      5↑r←101 103|⍞1⊢p+100×v
88 94
56  3
97  1
37 62
 8 56
```

After filtering out the centerline positions, note that identifying quadrants corresponds to a simple bitmask that marks which side of each centerline a position lies.

```
      q←50 51>⍤1⊢r←r⌿⍨∧/50 51≠⍤1⊢r
      5↑q,'|',r
0 0 | 88 94
0 1 | 56  3
0 1 | 97  1
1 0 | 37 62
1 0 |  8 56
```

The solution is then just a matter of counting up each quadrant:

```
×⌿q⊢∘≢⌸r
```

20 Performance

One reasonable objection is the use of Key (⌸) [3] since it performs a sort, whereas tallying groups membership is sufficient. In fact, a direct tally is completely straightforward if we first convert the quadrant selector pairs into integer ids:

```
      +/(∪g)∘.=g←2 2⊥⍉q
```

Notice that this performs `≢∪q` reductions over `q`, giving it $O($`(≢q)×≢∪q`$)$ complexity. On the other hand, the Key solution performs a radix sort [4] over the keys `q`, resulting in the exact same complexity!

That said, `∘.=` over scalar arrays has all kinds of opportunities for SIMD optimizations, making it generally faster than Key, and indeed, comparing the two core expressions bears this out:

```
      )copy dfns cmpx
      g←2 2⊥⍉q
      cmpx 'q⊢∘≢⌸r' '+/(∪g)∘.=g'
  ×⌿q⊢∘≢⌸r     → 1.5E¯6 |   0% ⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕
  ×⌿+/(∪g)∘.=g → 7.9E¯7 | -48% ⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕
```

However, notice that we bias in favor of `∘.=` above by ignoring the overhead of calculating `g`, which turns out to overwhelm any advantage, making Key come out faster in the overall problem solution here:

```
      cmpx 'q⊢∘≢⌸r' '+/(∪g)∘.=g←2 2⊥⍉q'
  ⊢∘≢⌸q              → 1.9E¯6 |    0% ⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕
  +/(∪g)∘.=g←2 2⊥⍉q → 4.2E¯6 | +117% ⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕
```

21 Acknowledgements

Thank you to Adám [5] for pointing out the potential speedup with `∘.=`. Originally, I had a naïve outer product which slowed it down considerably:

```
      cmpx '×⌿q⊢∘≢⌸r' '×⌿(∪g)∘.{+⌿α=ω}⊂g'
  ×⌿q⊢∘≢⌸r          → 1.5E¯6 |    0% ⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕
  ×⌿(∪g)∘.{+⌿α=ω}⊂g → 2.0E¯6 | +38% ⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕
```

Even though `∘.{+⌿α=ω}` and `+/α∘.=ω` are performing the same work, apparently the former expression falls back to a generic implementation of a scalar loop, whereas the latter is a recognized pattern with special optimized code.

Bibliography

[1] E. Wastl, “Advent of Code 2024, Day 14.” [Online]. Available: https://adventofcode.com/2024/day/14

[2] “Verify & Fix Input.” [Online]. Available: https://help.dyalog.com/19.0/#Language/System%20Functions/vfi.htm

[3] “Key.” [Online]. Available: https://help.dyalog.com/19.0/#Language/Primitive%20Operators/Key.htm

[4] “Radix sort, Complexity and performance.” [Online]. Available: https://en.wikipedia.org/wiki/Radix_sort#Complexity_and_performance

[5] “Adám Brudzewsky.” [Online]. Available: https://aplwiki.com/wiki/Ad%C3%A1m_Brudzewsky

Readable Code is Unreadable

Brandon Wilson
June 06, 2025

J [1] has an infamous prototype implementation called the J Incunabulum [2]. If you click that link, you'll immediately understand why it has the reputation it has.

I'd like to share why I now think this "unreadable" code actually communicates better than so-called "readable" code.

Let me be clear; on my first encounter with the J Incunabulum, I found it inscrutable. However, after having heavily steeped in APL for a few years, I just gave it a quick re-read and had a completely different experience.

23 Seeing the Forest

The code catches flak for its single-letter names and terseness, but it directly references bog standard APL and parsing concepts. What's more important is that the code macro-structure is immediately apparent and clearly answers high-level questions up front.

23.1 What Are Our Basic Datatypes?

Right off the bat, Whitney tells us the kind of data we'll be thinking about:

```
typedef char C;typedef long I;
typedef struct a{I t,r,d[3],p[2];}*A;
```

The first line gives us our main C types, and we can easily guess the second defines an array structure. Indeed, at a bare minimum, APL arrays need to encode type, rank, and shape metadata, telling us what `t`, `r`, and `d` are. Thus `p` points to our element data. Note that `d` probably stands for "dimension" or perhaps "depth"; I'm unsure on this point.

We also notice that `d[3]` says we only support up to rank 3, already signaling the explicit limitations chosen for this proof of concept. It's not entirely clear why we have two `p` pointers, though. Let's keep reading.

23.2 What Are Our Basic Operations?

Next we find the important fundamental operations we'll be using to build our new array language:

```
#define P printf
#define R return
#define V1(f) A f(w)A w;
#define V2(f) A f(a,w)A a,w;
#define DO(n,x) {I i=0,_n=(n);for(;i<_n;++i){x;}}
I *ma(n){R(I*)malloc(n*4);}mv(d,s,n)I *d,*s;{DO(n,d[i]=s[i]);}
tr(r,d)I *d;{I z=1;DO(r,z=z*d[i]);R z;}
A ga(t,r,d)I *d;{A z=(A)ma(5+tr(r,d));z->t=t,z->r=r,mv(z->d,d,r);
 R z;}
```

Apparently, we'll be printing and returning a lot, as expected. APL functions have fixed arity of one or two, calling the left argument `⍺` (alpha) and the right `⍵` (omega). Considering that J functions are called "verbs", it becomes pretty clear that `V1` and `V2` are all the function prototypes needed for J primitives. Note the K&R-style definitions [3].

- `DO` defines our basic loop operation, so iterations will probably all naïvely be `O(1)`;
- `ma` says we'll be allocating 4-byte (*i.e.* 32-bit) chunks;
- `mv` essentially gives us a copy operation over chunks;
- `tr` or "times reduce" by inspection; and finally
- `ga` generates a new array.

All of these—except for `tr` perhaps—are clearly going to be useful. The 32-bit and data model [4] assumptions here are a bit dirty, but we can forgive that in a proof of concept.

23.3 How About Using the Basic Operations?

Now that we know our basic toolkit here's the implementation core!

```
V1(iota){I n=*w->p;A z=ga(0,1,&n);DO(n,z->p[i]=i);R z;}
V2(plus){I r=w->r,*d=w->d,n=tr(r,d);A z=ga(0,r,d);
 DO(n,z->p[i]=a->p[i]+w->p[i]);R z;}
V2(from){I r=w->r-1,*d=w->d+1,n=tr(r,d);
 A z=ga(w->t,r,d);mv(z->p,w->p+(n**a->p),n);R z;}
V1(box){A z=ga(1,0,0);*z->p=(I)w;R z;}
V2(cat){I an=tr(a->r,a->d),wn=tr(w->r,w->d),n=an+wn;
 A z=ga(w->t,1,&n);mv(z->p,a->p,an);mv(z->p+an,w->p,wn);R z;}
V2(find){}
V2(rsh){I r=a->r?*a->d:1,n=tr(r,a->p),wn=tr(w->r,w->d);
 A z=ga(w->t,r,a->p);mv(z->p,w->p,wn=n>wn?wn:n);
 if(n-=wn)mv(z->p+wn,z->p,n);R z;}
V1(sha){A z=ga(0,1,&w->r);mv(z->p,w->d,w->r);R z;}
V1(id){R w;}V1(size){A z=ga(0,0,0);*z->p=w->r?*w->d:1;R z;}
```

All our basic functions, both dyadic and monadic, sit together in a single block. In fact, each of these definitions is brutally simple and explicitly elides array language features in service of architectural clarity. For example,

```
V2(plus){I r=w->r,*d=w->d,n=tr(r,d);A z=ga(0,r,d);
 DO(n,z->p[i]=a->p[i]+w->p[i]);R z;}
```

Even if you don't know APL, one can guess what "plus" should do. We already know that `V2` defines a new arity-2 function, and what's the obvious way to add arrays? Just vector addition, *i.e.* element-by-element, right? Procedurally, we need to allocate an output array of the right size and then populate it with the result, one element at a time.

Now we see why we want the `tr` helper: it gives us the data element count of an arbitrary multi-dimensional array.

Also, as an APLer we immediately notice the elision of scalar extension [5]. That can be modeled with `rsh` on the left argument, so we're probably just defining some minimal subset of the primitive APL functions.

Other definitions all follow a similar pattern: 1) calculate the metadata for the result array, 2) allocate a result in the conventional name `z`, and 3) populate said array.

Notice, too, that `find` is not yet implemented. Curious. What else stands out?

```
V2(rsh){I r=a->r?*a->d:1,n=tr(r,a->p),wn=tr(w->r,w->d);
 A z=ga(w->t,r,a->p);mv(z->p,w->p,wn=n>wn?wn:n);
 if(n-=wn)mv(z->p+wn,z->p,n);R z;}
...
V1(id){R w;}V1(size){A z=ga(0,0,0);*z->p=w->r?*w->d:1;R z;}
```

These are the only branches we see, indicating intentional complications [6], and `rsh` (read "reshape") is obviously the longest implementation. From experience, we know that reshape recycles elements when extending, and indeed it's clear that's what the `if`-branch takes care of—cleverly using a circular copy on `z->p`.

Also notable is the complete lack of error handling. We are clearly trying to laser focus on exhibiting core ideas here.

23.4 What About Output?

We have just a single datatype, so display is brutally simple:

```
pi(i){P("%d ",i);}nl(){P("\n");}
pr(w)A w;{I r=w->r,*d=w->d,n=tr(r,d);DO(r,pi(d[i]));nl();
 if(w->t)DO(n,P("< ");pr(w->p[i]))else DO(n,pi(w->p[i]));nl();}
```

There are no pretty print facilities, but rank, element data, and recursive nesting are the bare minimum needed to fully understand array contents. This is all that is needed to get up and running.

23.5 What About Parsing?

First we setup a vtable [7] that contains our implementation functions:

```
C vt[]="+{~<#,";
A(*vd[])()={0,plus,from,find,0,rsh,cat},
 (*vm[])()={0,id,size,iota,box,sha,0};
```

Each J verb has three pieces of data: 1) its glyph, 2) its dyadic definition, and 3) its monadic definition. We use a standard APL technique of storing this data in a column-major table. The `vt` column names the verbs (via their glyphs), `vd` maps dyadic usage to its vtable function, and 3) similar for monadic usage with `vm`.

Nothing special, but clear and direct. In particular, this means that a verb's "ID" is its table offset.

```
I st[26]; qp(a){R  a>='a'&&a<='z';}qv(a){R a<'a';}
A ex(e)I *e;{I a=*e;
 if(qp(a)){if(e[1]=='=')R st[a-'a']=ex(e+2);a= st[ a-'a'];}
 R qv(a)?(*vm[a])(ex(e+1)):e[1]?(*vd[e[1]])(a,ex(e+2)):(A)a;}
```

Both `qp` and `qv` are obvious predicates (`q` for "query"?), and we can see that `ex` is calling functions in our vtable. This is obviously the executor. It even supports `=`, which stores values in `st`. The one-to-one map between alphabet characters and `st` indices is nice and minimal. We only allow single-letter names, probably enough for small language experiments.

If we're not an =-definition, though, then we execute. APL's function precedence [8] rule makes this dead simple: just right-recurse. The only question is whether we're a dyadic, monadic, or "niladic" application. The nested digraph here creates this obvious trident branch structure.

23.6 What About Input?

After a cool three dozen SLOC [9], we conclude our J interpreter:

```
noun(c){A z;if(c<'0'||c>'9')R 0;z=ga(0,0,0);*z->p=c-'0';R z;}
verb(c){I i=0;for(;vt[i];)if(vt[i++]==c)R i;R 0;}
I *wd(s)C *s;{I a,n=strlen(s),*e=ma(n+1);C c;
 DO(n,e[i]=(a=noun(c=s[i]))?a:(a=verb(c))?a:c);e[n]=0;R e;}

main(){C s[99];while(gets(s))pr(ex(wd(s)));}
```

At this point, the patterns are familiar. `noun` and `verb` do the obvious things, `wd` creates our token stream, notably only supporting single character tokens, and `main` reads exactly like R-E-P-L [10] (well, in reverse).

24 Points of Note

The overall code is organized in a top-down manner. Our audience here is APLers who know a thing or two about language implementation. It makes sense to lead with high level ideas, and the code exactly mirrors this structure. We literally read the code linearly top-to-bottom just like a short whitepaper. That's great communication, IMHO!

As mentioned offhand above a few times, the particular limitations chosen serve to clarify the presented message. It's not about implementation shortcuts. Consider that we imposed

- Single-character tokens only;
- No Spaces;
- No parentheses to control order of operations;

- No array literals;
- No scalar extension;
- No function definitions;
- Limited rank;
- *etc.*

What purpose did these concessions serve? Why not more?

Look at `rsh`. It would clearly be simpler to remove the final conditional that allows circular extension. Given all the other limitations, why not add another? Well, this would make the domain of inputs to `rsh` special and different from the other functions. That complicates the conceptual model.

J didn't exist when the Incunabulum was written. It was an idea. Iverson [11], Whitney [12], and Hui [13] were trying to discuss and discover what a new APL language could look and feel like. The Incunabulum is a single page of code that presents tools to concisely express the salient ideas in that design space—array model [14], usability of ASCII glyphs, leading axis theory [15], *etc.*

With that goal, poking at things like fancier array literals or parentheses support feels like bikeshedding. The rank cutoff is arbitrary but trivially changeable; none of the code depends on a specific cutoff.

Function definitions are interesting from a language design perspective. Worrying about scope and closure support introduces questions about syntax. We could directly store token vectors easily enough, but this complicates the recursion in `ex`. All-in-all, these are questions somewhat orthogonal to the others.

I'll stop there…

What is most interesting, IMHO, is that all the above comes across in short 20-odd minutes of reading this supposedly "inscrutable" code. What more could one ask for?

25 What about `find`?

Out of curiosity, I threw together a plausible Whitney implementation:

```
V2(find){I r=w->r,*d=w->d,n=tr(r,d),j=n;DO(n,if(a->p[0]==w->p[i])j=i);
 A z=ga(0,1,&w->r);DO(r,z->p[i]=(j/(r==1?1:tr(r-1-i,d+1+i)))%d[i]);R z;}
```

which is a bit subtle and certainly the most complicated logic in here. It works fine for simple integer arrays, but boxes compare as pointers. Recursive comparison of contents might make more sense. Apparently, `find` opens a whole design question about what array equality even means.

Maybe all these factors contributed to `find` remaining unimplemented, or maybe Whitney just ran out of time!

26 Epilogue: Getting Compiled

It's fun to play around with this little implementation. I definitely recommend compiling it yourself and giving it a little test drive.

Due to the K&R function prototypes, modern GCC will need the `-ansi` or `-std=c89` flag. Also, as noted above, the code assumes a 32-bit architecture in one place. We could cross compile, but the easiest workaround is a simple patch:

```
--- a/inucabulum.c
+++ b/inucabulum.c
@@ -5,7 +8,7 @@
 #define V1(f) A f(w)A w;
 #define V2(f) A f(a,w)A a,w;
 #define DO(n,x) {I i=0,_n=(n);for(;i<_n;++i){x;}}
-I *ma(n){R(I*)malloc(n*4);}mv(d,s,n)I *d,*s;{DO(n,d[i]=s[i]);}
+I *ma(n){R(I*)malloc(n*sizeof(I));}mv(d,s,n)I *d,*s;{DO(n,d[i]=s[i]);}
 tr(r,d)I *d;{I z=1;DO(r,z=z*d[i]);R z;}
 A ga(t,r,d)I *d;{A z=(A)ma(5+tr(r,d));z->t=t,z->r=r,mv(z->d,d,r);
  R z;}
```

It'll spit out some `builtin-declaration-mismatch` warnings, but those are immaterial here.

Bibliography

[1] "J." [Online]. Available: https://aplwiki.com/wiki/J

[2] "Incunabulum," in *An Implementation of J*, 1990. [Online]. Available: https://www.jsoftware.com/ioj/ioj.htm

[3] "Old-Style Function Definitions." 2023. [Online]. Available: https://www.gnu.org/software/c-intro-and-ref/manual/html_node/Old_002dStyle-Function-Definitions.html

[4] "C Language Data Type Models: LP64 and ILP32." [Online]. Available: https://docs.oracle.com/cd/E19620-01/805-3024/lp64-1/index.html

[5] "Scalar extension." [Online]. Available: https://aplwiki.com/wiki/Scalar_extension

[6] "Complication (horology)." [Online]. Available: https://en.wikipedia.org/wiki/Complication_(horology)

[7] "Virtual method table." [Online]. Available: https://en.wikipedia.org/wiki/Virtual_method_table

[8] "Precedence." [Online]. Available: https://aplwiki.com/wiki/Precedence

[9] "Source lines of code." [Online]. Available: https://en.wikipedia.org/wiki/Source_lines_of_code

[10] "Read-eval-print loop." [Online]. Available: https://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print_loop

[11] "Kenneth E. Iverson." [Online]. Available: https://aplwiki.com/wiki/Kenneth_E._Iverson

[12] "Arthur Whitney." [Online]. Available: https://aplwiki.com/wiki/Arthur_Whitney

[13] "Roger Hui." [Online]. Available: https://aplwiki.com/wiki/Roger_Hui

[14] "Array model." [Online]. Available: https://aplwiki.com/wiki/Array_model

[15] "Leading axis theory." [Online]. Available: https://aplwiki.com/wiki/Leading_axis_theory

On 9P Semantics

Brandon Wilson
June 22, 2025

Recently, I threw together a 9P file server [1] for Dyalog Ride [2], in the spirit of webfs [3]. It was my first foray into using 9P directly, and I found some of the semantics a bit confusing, so I am writing this up to document my newfound insights and throw them into the aether.

28 Conventions Make the Protocol

9P [4] simply presents a handful of request-reply message pairs. The protocol itself is extremely parsimonious and almost too flexible to be interesting on its own. However, in quintessential Plan 9 style, the power comes from using the protocol according to "globally understood conventions" (*cf* Plan 9 From Bell Labs [5] by Rob Pike). In other words, what breathes life into the protocol is the message dance that CLI tools like `ls`, `cp`, `cat`, *etc.* end up shaking out.

First, let's start a server with chatty9p [6] set to give debug output:

```
cpu% somefs -D
<-6- Tversion tag 65535 msize 32792 version '9P2000'
-6-> Rversion tag 65535 msize 32792 version '9P2000'
<-6- Tauth tag 5 afid 442 uname x aname
-6-> Rerror tag 5 ename somefs: authentication not required
<-6- Tattach tag 5 fid 442 afid -1 uname x aname
-6-> Rattach tag 5 qid (ffffffff00000000 0 d)
```

We see that initialization consists of a version ⟶ auth ⟶ attach triplet, just as documented [4]. In this case, somefs automounts to `/mnt/some`:

```
cpu% file /mnt/some
/mnt/ride: <-6- Twalk tag 5 fid 442 newfid 427 nwname 0
-6-> Rwalk tag 5 nwqid 0
<-6- Topen tag 5 fid 427 mode 0
fid mode is 0
-6-> Ropen tag 5 qid (ffffffff00000000 0 d) iounit 0
<-6- Tstat tag 5 fid 427
-6-> Rstat tag 5  stat '/' 'x' 'x' 'x' q (ffffffff00000000 0 d) m 020000000555 at ...
directory
<-6- Tclunk tag 7 fid 427
-6-> Rclunk tag 7
```

We see a walk [7], open [8], stat [9], and clunk [10]. Both `open` and `stat` are expected, as they're essentially the bare minimum libc calls we'd need to read the directory information, and we can guess that `clunk` corresponds to close(2) [11].

29 Why Walk?

The semantics of `walk` is what confused me, especially since implementations often provide a `clone` and `walk1` pair of callbacks to answer this one request. What are we cloning, and what is walking where?

Leading with the lede, the answer is that `walk(5)` provides the path resolution mechanism. Said another way, we need some way to map `fid`s to `qid`s, and this is it.

The defining feature of 9P, IMHO, is its handling of `fid`s and `qid`s, which are intended to be understood, respectively, as file descriptors and unique file identifiers. They're morally equivalent to file descriptors and inode ids [12] from Linux.

As a client connecting to a server for the first time, we have no idea what files it provides. Procedurally, we start by only knowing the fileserver root mount point, which we have to `read(2)` to see its content. The root could be a file, but in the case of a directory, reading returns a list of dirents. To `open(2)` any of these, we need to first assign a fid to its path—the role of `Twalk`—and resolve it to a qid—the role of `Rwalk`.

On the implementation side, `fid`s are `structs` that hold data associated with all the in-flight file structures—think connection information in the case of `webfs(4)`. `Clone` is our mechanism for producing a new `fid` from an old one, *a la* `clone(2)`ing a process. Since each `fid` and path component in a walk has fileserver-specific semantics, the `clone` and `walk1` pairs ensure that the new `fid` can acquire all the necessary content needed to figure out which `qid` is should map to.

Here's a typical access pattern on something like `webfs(4)` to initialize a new connection:

```
cpu% <clone {cd `{read}}
<-6- Twalk tag 5 fid 442 newfid 473 nwname 1 0:clone
-6-> Rwalk tag 5 nwqid 1 0:(ffffffff00000002 0 )
<-6- Topen tag 5 fid 473 mode 0
fid mode is 0
-6-> Ropen tag 5 qid (0000000000000004 0 ) iounit 0
<-6- Tread tag 5 fid 473 offset 0 count 1
-6-> Rread tag 5 count 1 '0'
<-6- Tread tag 5 fid 473 offset 1 count 1
-6-> Rread tag 5 count 1 '
'
<-6- Twalk tag 5 fid 442 newfid 488 nwname 1 0:0
-6-> Rwalk tag 5 nwqid 1 0:(0000000000000003 0 d)
<-6- Tclunk tag 5 fid 473
-6-> Rclunk tag 5
<-6- Twalk tag 8 fid 488 newfid 473 nwname 0
-6-> Rwalk tag 8 nwqid 0
<-6- Topen tag 8 fid 473 mode 0
fid mode is 0
-6-> Ropen tag 8 qid (0000000000000003 0 d) iounit 0
<-6- Tclunk tag 8 fid 473
-6-> Rclunk tag 8
```

We 1) walk to `clone`, 2) open and read it, and 3) walk to the connection directory that `clone` assigned. The final walk then simply renames the original starting fid with the one current directory, supposedly `cd` semantics.

30 Final Thoughts

The mischievous side of me wants to know how we might creatively abuse 9P. There is nothing preventing a client from issuing reads and writes before walks, and there is no obligation for the server to actually tell clients which dirents exist. We could easily imagine a port knocking-like scenario, where specific messages exchanges unlock hidden functionality.

What other creative fileservers can you think up to (ab)use 9P?

Bibliography

[1] R. Pike, D. Presotto, K. Thompson, H. Trickey, and P. Winterbottom, "The Use of Name Spaces in Plan 9," *Operating Systems Review*, vol. 27, no. 2, pp. 72–76, Apr. 1993, [Online]. Available: https://doc.cat-v.org/plan_9/4th_edition/papers/names

[2] "The Ride Protocol." [Online]. Available: https://github.com/Dyalog/ride/blob/master/docs/protocol.md

[3] "webfs - world wide web file system." [Online]. Available: https://man.9front.org/4/webfs

[4] "introduction to the Plan 9 File Protocol, 9P." [Online]. Available: https://man.9front.org/5/intro

[5] R. Pike *et al.*, "Plan 9 from Bell Labs," *Computing Systems*, vol. 8, no. 3, pp. 221–253, 1995, [Online]. Available: https://doc.cat-v.org/plan_9/4th_edition/papers/9

[6] "9P file service." [Online]. Available: https://man.9front.org/2/9p

[7] "walk - descend a directory hierarchy." [Online]. Available: https://man.9front.org/5/walk

[8] "open - prepare a fid for I/O on an existing or new file." [Online]. Available: https://man.9front.org/5/open

[9] "stat, wstat - inquire or change file attributes." [Online]. Available: https://man.9front.org/5/stat

[10] "clunk - forget about a fid." [Online]. Available: https://man.9front.org/5/clunk

[11] "open, create, close - open a file for reading or writing, create file." [Online]. Available: https://man.9front.org/2/open

[12] R. Gooch and P. Enberg, "Overview of the Linux Virtual File System," 2005. [Online]. Available: https://docs.kernel.org/filesystems/vfs.html

Easy Random Trees

Brandon Wilson
February 27, 2026

Can you think of a way to efficiently generate a random plane tree [1]?

Richard P. Stanley in his book Catalan Numbers [2] has a really nifty combinatorial proof of why Catalan numbers [3] have the formula

$$C_n = \frac{1}{n+1}\binom{2n}{n}$$

The standard proof uses generating functions [4] applied to an inductive definition of the Catalan numbers, which frankly does little to illuminate their connection with combinatorial objects, despite the fact that the appearance of a binomial coefficient [5] gives a hint that it's counting something, and the division suggests we might be looking at some kind of equivalence class.

Stanley's proof is so direct and beautiful, I cannot help but share. However, let us instantiate the proof as a little program that generates a random tree with n nodes:

```
D←{n←⍵-1                    ⍝ The ⍵-1st Catalan number counts number of ⍵-node trees
 x←x[?⍨≢x←1⍪(n⍴1)⍪n⍴¯1]     ⍝ Random step vector of length 1+2×⍵
 d←0⍪¯1↓+⍀x                 ⍝ Depth vector
 i←⊃⍸⌊(⌈⍀d)=d               ⍝ Rightmost lowest node
 x⌽⍨←i                      ⍝ Strict ballot sequence
 d←¯1+(x=1)≠+⍀x             ⍝ Corresponding tree node depths
}
```

At center stage here is the isomorphism between plane trees and strict ballot sequences [6], which is directly connected via the depth vector tree representation [7], which I'll assume is already familiar to readers here. The image to have in mind is a depth-first search [8] traversal pattern.

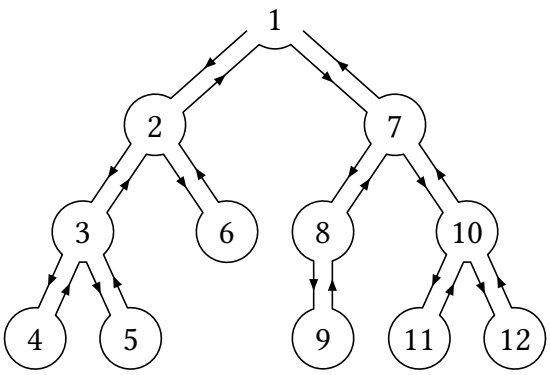


Figure 1: Depth-first search tree traversal

To construct a depth vector, we simply write down the current depth every time we first visit a node.

We can construct a related sequence by starting at the root and writing down a 1, then writing down a 1 every time we descend to a node, and finally, writing a ¯1 every time we ascend. Notice, in particular, that we both descend to and ascend from every node exactly once, except for the root. This guarantees

that the partial sums are at least 1 and the total sum is exactly 1. Such a vector of 1's and ¯1's is called a strict ballot sequence [6], and given one, it's straightforward enough to convert it into the depth vector of the corresponding tree:

```
      +⍀x←1 1 ¯1 1 1 1 ¯1 ¯1 ¯1 1 ¯1 1 1 ¯1  ⍝ Strict ballot sequences
1 2 1 2 3 4 3 2 1 2 1 2 3 2                  ⍝  have positive partial sums

      ⊢d←¯1+(x=1)≠+⍀x                        ⍝ Depth vectors ignore the ¯1's
0 1 1 2 3 1 1 2
```

Now let's think about random sequences of 1's and ¯1's. To have any hope of being a strict ballot sequence, the sum must be 1, so we know *a priori* that there must be one more 1 than ¯1's. We can parameterize this as sequences of length $2n+1$, with $n+1$ ones and n negative ones.

```
      ⊢x←x[?⍨≢x←1⍪(nρ1)⍪nρ¯1]  ⍝ Compare with D above
¯1 ¯1 ¯1 1 ¯1 ¯1 ¯1 1 ¯1 ¯1 1 1 1 1 ¯1 1 1 1 1 1 ¯1
```

Most likely, such a random vector will not be a strict ballot sequence, since partial sums can drop below 1. However, consider the rightmost lowest partial sum, N, at index i:

```
      i←⍳⌊/msk←s=N←⌊⌿s←+⍀x
      s⍪⍉⍪msk
¯1 ¯2 ¯3 ¯2 ¯3 ¯4 ¯5 ¯4 ¯5 ¯6 ¯5 ¯4 ¯3 ¯2 ¯3 ¯2 ¯1 0 1 2 1
 0  0  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0 0 0 0 0
```

Looking at the tail sequence starting at s_i, we know it must end in 1. That means s_i is $N+1$ below the tail. However, everything before s_i is at most N below the start. In other words, if we transplant the tail $x_i \ldots x_{2n}$ to the start and recompute partial sums, they will remain positive. Said another way, rotating the ballot sequence by i produces a strict ballot sequence. Or more concisely,

```
d←0⍪¯1↓+⍀x    ⍝ Depth vector
i←⊃⍋⍳(⌊⍀d)=d  ⍝ Rightmost lowest node
x⍋⍨←i         ⍝ Strict ballot sequence
```

Combining with our conversion to a depth vector finishes the implementation of `D`.

Better yet, the Catalan number formula falls right out of the above by noting that there are exactly $\binom{2n+1}{n}$ strings of 1's and ¯1's and that since n and $2n+1$ are coprime, no rotations produce equal sequences. This means that the sequence length, $2n+1$, must divide $\binom{2n+1}{n}$. Thus the number of strict ballot sequences with length $2n+1$ is

$$\frac{1}{2n+1}\binom{2n+1}{n}$$

It is a small matter of algebra to check that this equals the value of C_n above. For $n=0$, we have a strict ballot sequence of length 1, which corresponds to a tree of just a single root node as per the isomorphism outlined above. This means that C_n must count plane trees with $n+1$ nodes.

Stanley's book is delightful and worth perusing. The above just uses one isomorphism, but the book contains a veritable cornucopia and is replete with nicely illustrative diagrams to boot. Highly recommended!

Bibliography

[1] "Tree (graph theory), Ordered Tree." [Online]. Available: https://en.wikipedia.org/wiki/Tree_(graph_theory)#Ordered_tree

[2] R. P. Stanley, *Catalan Numbers*. Cambridge University Press, 2015.

[3] "Catalan number." [Online]. Available: https://en.wikipedia.org/wiki/Catalan_number

[4] "Generating function." [Online]. Available: https://en.wikipedia.org/wiki/Generating_function

[5] "Binomial coefficient." [Online]. Available: https://en.wikipedia.org/wiki/Binomial_coefficient

[6] "Bertrand's ballot theorem." [Online]. Available: https://en.wikipedia.org/wiki/Bertrand%27s_ballot_theorem

[7] Jacob, "Convert a parent vector to a depth vector." [Online]. Available: https://codegolf.stackexchange.com/questions/278877/convert-a-parent-vector-to-a-depth-vector

[8] "Depth-first search." [Online]. Available: https://en.wikipedia.org/wiki/Depth-first_search