Badness 0
(Knuth's version)

Dr. Tom Murphy VII, Ph.D.
March 2024

It has become clear to me that many people walk this Earth completely unbothered by incorrect details. For example, they are unconcerned when a hyperlink includes a surrounding space character. It doesn't upset them when the screw heads on a light switch wall plate are not all lined up. They didn't notice that the rules of Wordle's "hard mode" are simply wrong. They care as much as the phone's autocorrect (none) about the difference between "its" and "it's." When someone asks, "Will you marry me?" and they think "Oh my god!" it's not because the proposer probably should have used the subjunctive *would*.

I am... not like this. If a character in a TV commercial is handling a coffee cup but I can infer from its moment of inertia that the cup does not contain any liquid, I immediately lose suspension of disbelief and will not purchase the product featured in the commercial. I literally projectile vomit if Auto-Motion Plus is enabled on a television in the hotel I'm staying in, even if the TV is not turned on, or if someone misuses the word "literally." If I see a paragraph missing a period at its end on Wikipedia, I will spend dozens of hours writing software to organize and semi-automate a distributed effort to fix all the missing periods on Wikipedia.[2] And worse, each time I learn of a new type of mistake, I am forever cursed to notice that mistake

Seriously: One time I found myself spell-correcting someone else's *lorem ipsum* text in a slide. It said "lorem epsom," which is funny. I think about that incident all the time. The person that wrote the slide probably thinks about things like leveraging synergy, generative AI, metaverses, blockchain 3.0, snackable content, being eco-green, and so on, without it occurring to him that these things could have nuance and meaning separate from their names. He has probably never even read the Wikipedia article on Lorem Ipsum. He is successful and rich.

Another successful person is the congressperson Bill Cassidy. Criticizing a proposed bill that would reduce the standard work week in the US by 8 hours, from 40 to 32, this senator says,

> *Sen. Bill Cassidy of Louisiana, representing the ______ party, said paying workers the same wages for fewer hours would force employers to pass the cost of hiring more workers along to consumers.*
>
> *"It would threaten millions of small businesses operating on a razor-thin margin because they're unable to find enough workers," said Cassidy. "Now they've got the same workers, but only for three-quarters of the time. And they have to hire more."*

Actually, that's not exactly the quote, but I needed to make it look nice.[3] And this is not a paper about politics, but let's just say you can guess what word goes in the blank.

Anyway, *OKAY*, first of all, razors famously have high margins. It's like the worst possible metaphor here.

For another thing: This guy mixes fancy typographic quotes and ASCII ones.

But the main thing I want to talk about is: What? No! 32/40 is four fifths, not three quarters. This is not, like, complicated math. It uses some of the world's smallest integers. Everybody knows that the work week is 40 hours, and that a work day is 8 hours, and that the proposed bill reduces it by one day, giving four of five days. I don't really mind if someone makes an error in calculation (well, I do mind, but I am certainly prone to doing it). The infuriating realization here is that this person does not even think of "three-quarters" as a kind of thing that can be right or wrong. He says three quarters because it makes smaller number feelings. You could imagine him having the conversation (with me, perhaps): "You say four-fifths, I say three-quarters." Me: "But it *is* four fifths. And why are you always hyphenating it?" Him (smiling patronizingly): "I guess we just have to agree to disagree."

The opposite of this person is the hero called Donald Knuth.

I'm not saying that Donald Knuth isn't successful and rich. According to the website "Famous Birthdays,"[4] which is probably generated by AI or at least by people whose economic output is measured in a count of words, and words whose value is computed by their ability to drive ad clicks, Donald Knuth is "is one of the most popular and richest Mathematician who was born on January 10, 1938 in Wisconsin, Wisconsin, United States. Mathematician and engineer who was arguably most recognized as the Professor Emeritus at Stanford in Palo Alto, California." As one of the richest Mathematician from United States, according to the analysis of Famous Birthdays, Wikipedia, Forbes & Business Insider, "Donald Knuth's net worth $3--5 Million.*"

I suppose it is arguable that he is the Professor Emeritus. And it is very likely true that he is the *only* popular and rich mathematician born on that specific day in Wisconsin, making the singular "Mathematician" perhaps a technical master-stroke. But more likely this is just an amusingly dense series of imprecisions. The asterisk of course does not have any referent on the page.

What I mean when I say that Donald Knuth is the opposite of this person is that Knuth is interested in unpacking a single unnecessary detail, recursively, until it is completely solved. According to the website Famous Bibliophiles, one day Donald Knuth set out to write down the entire subject of computer science in a single book called The Art of Computer Programming. As he was doing so, he realized that describing computer algorithms in a lasting form would require a programming language that was not subject to constant revision, so he invented the MIX instruction set for an idealized computer. After writing some 3000 pages out in longhand, he found that it was impractical to print them all in one book, so the plan expanded to be multiple volumes. Then when he got a draft of one of the books back from the typesetter, he was unhappy with the details of the typography, and so he paused his work writing down all of computer science to create some new computer science: First an algorithm for determining where to place line breaks in order to make text optimally beautiful, then algorithms for hyphenating words, then generalizations of these for typesetting mathematics, and then a full computer typesetting system that is still in wide use today, called TeX. Along the way he was unsatisfied with the specific typefaces that existed in the world, and unsatisfied with the way that typefaces were described at only one weight, and so he created the parameterized METAFONT system and several new typefaces. Undeterred by these excursions, he returned to his original task of writing down the entirety of computer science, using all the technology he had built. By the time he finished this, much more computer science had been invented, including by his own hand, and so he needed to rework MIX for the next volume, and update the first. The revised plan of eight volumes remains the intention in 2024. However, he found that the volumes were getting rather long, and began releasing portions of volumes ("fascicles"). So far, Volume 4 has been partially published as books 4A[5] (fascicles 0–4; 912 pages) and 4B[6] (fascicles 5–6; 736 pages). It is unknown how many more episodes remain in Volume 4. I expect that every conversation that Knuth has with his editor goes like this. Editor: "Hey, Donald, I hope you're well. Just wondering if you have an update on when 4C will be ready? Or any more icicles?" Donald E. Knuth: "I am working diligently on fascicles for Volume 4C. As I've mentioned in the past, it's impossible to tell how long it will be, since mathematics does not obey the rules of project management." Editor: "I just need a date to tell the publishers." Donald E. Knuth: "Like I've said, any date would be very low confidence, other than the fact that it will be in the future." Editor: "I just need a date." Donald E. Knuth: "Would you like me to say a date, knowing that it's a very low confidence guess, and that I would be extremely likely to miss that date, or even deliver early?" Editor: "Early! Now we're talking." Donald E. Knuth: "What use is the date if you're excited about the possibility of it being early, relative to some unknown date?" Editor: "I just need a date for the publishers." Donald E. Knuth: "2030." Editor: "Thanks Donald, you're the best!"

Volume 5 is estimated to be ready in 2030, when Knuth will be 92.

That's a large amount of language!

Nightmare on LLM street

Then we have Large Language Models.[7] One of the irritating things about LLMs is that they are so buzzwordy, but unlike most buzzwordy trends, they are actually substantive. They produce remarkably fluent text. With no additional training they frequently beat purpose-built models that have been in development for decades. They generalize to completely new situations.

So many things about "AI" distress me. *Dolor sit amet!* I worry about the devaluation of human creativity, about large-scale disinformation and spam ruining the beautiful library of knowledge that humans have created, about extreme concentration of wealth. And yes, I worry about competing with

AI. Being able to work tirelessly and thousands of times faster than humans is a huge competitive advantage. Of course, I find some solace in the significant possible upsides. It might help us solve hard problems like climate change and AI. But even in the best scenarios we will not be able to ignore it: Even if it never gets as smart and precise as Knuth, it's already too economically useful in its Lorem Epsom state (just like Lorem Epsom himself).

On the other hand, the technology is pretty neat and lends itself to some nice abstractions. I love playing with words. So one of my side quests is to masticate this whole scenario by experimenting with LLMs in practical and impractical applications, and to try to make it fun (for me) to program with them.

Many things irritate me, so this is something I have ample experience with. I have a myriad of strategies for digestion of them. For this work I'm inspired by the "Hurry-Coward So-so-morphism," where I make connections between topics based solely on confusion of superficial lexical similarities without regard to their underlying meaning. So for example we have "ML" meaning both "Machine Learning" and "Meta Language", as well as "type" both as in "typeface" and as in "type systems for programming languages."[8] And because machine learning has claimed so many words, there are a great many shared with typography as well:

```
            +----------+
            |typography|
            +----------+
               /    \   “baseline”
“fixed point” /      \    “floating point”
             /        \  “weight”  “vector”
     “type” /          \   “descent”
           /            \  “kerning trick”
          /              \      “dingbats”
         /                \   “gradient”
 +-----------+           +--------+
 | functional|---------|machine |
 |programming|           |learning|
 +-----------+   “ML”    +--------+
      “lambda”   “generalization”
       “parameter” “tensor”
```

By no coincidence, I already spent a lengthy introduction talking about Donald Knuth's work in computer typography. So now I can tell you what this paper is about. If in our near AI future we are giving up on precision, perhaps at least we can have something that we want: Perfect typography? This paper is about a new typesetting system, BoVeX, which allows for the controlled exchange of precision for beauty. It essentially gives us a dial between Lorem Epsom and Donald Knuth. To illustrate, we'll first look at a simpler case by inspecting one of my other interests: Super Metroid.

> *The scientists' findings were astounding! They discovered that the powers of the Metroid might be harnessed for the good of civilization!*

Metroid is a video game series about a brain that has been enslaved inside a jar in an underground datacenter on the planet Zebes. This brain is called Mother Brain and its goal is to control the hypercapitalists called Space Pirates to increase their "score" as high as possible by conquering planets throughout the galaxy. Mother Brain was invented by the Space Pirates, although it is not clear whether the current situation was actually intended by the Space Pirates. The most super version of Metroid is Super Metroid.

In the 1990s the website `gamefaqs.com` collected plain text "FAQs" for classic video games, then just known as video games. On this site another hero was born. They were writing the definitive guide to speedrunning the SNES game Super Metroid when they saw that some of their ASCII lines ended up exactly the same length, and that it looked good:

```
  Once you save the game at your ship (about 1 hour 15 minutes is good), go
down to Tourian. Do not save your game in Tourian if you have intentions of
returning to any previously explored section on Planet Zebes. There will be
a few Metroids to kill before you reach Mother Brain, and they must all die
in order to continue to Mother Brain. Read the boss guide for more details.
Once Mother Brain is defeated, you will need to hurry back to your ship. By
now you will already have the HYPER BEAM. From Mother Brain’s room, go west
and then south. Take the blue door at the bottom and speed dash east. Super
jump up, and continue north. Once you land up top and are running east, aim
diagonally down to the right and shoot an unseen door. Eventually, you will
get to this door since lava will start to rise from the floor in this area.
Speed dash through the door you preopened, and charge for a super jump. Hug
either the left or right wall in the Craterian shaft and super jump up. Now
quickly get to your ship before the planet explodes. There should be almost
a minute left on the timer. Sit back and watch the ending! Did you beat the
game within 1 hour and 20 minutes?
```

and so they wisely decided to wordsmith the entire 28-page guide so that every line was exactly the same length, with no extra spaces or other cheating, just because it could be done.[9]

Doing this manually is a chore, and I do like to automate the chores of Speedrunners.[10] I got this working in an afternoon. It's, like, easy mode. For a paragraph of text and a target line length, I ask the LLM to remember the paragraph and recite it. The prompt looks like this:

Exercise in rephrasing text. The following paragraph needs to be rephrased so that it retains its precise meaning, but with minor variations in the specific choice of words, punctuation, and so on. No new facts should be introduced or removed, but it is good to use synonyms and change the word order and phrasing.

After this I insert something like `Original text:` followed by the original paragraph, then `Rephrased text:`. The model is ready to generate tokens.

I then sample text a word at a time to continue this prompt. If a line ends exactly on the number of characters that I want (and the next character is a space or other character that is appropriate to end a line) then I accept the stream so far and continue. If I exceed the line length, I back up to the state at the beginning of the line and try again with new random samples. I just keep doing that until the paragraph is complete, and we have beautifully justified monospace text that resembles the original. Here is an example of this paragraph rendered in monospace:

```
I sample text a word at a time to continue
this prompt. If a line ends exactly on the
number of characters I want, I accept that
text so far, and continue. If I exceed the
line length, I back up to the beginning of
the line and try again with new samples. I
keep repeating this until I get text I can
render in monospaced font, and that is how
we can get beautifully justified monospace
text. Here is an example of this paragraph
rendered in monospace:
```

You could argue that this is improved, even, by making the text shorter. It does use "monospace" and "monospaced" inconsistently. The most upsetting thing here is that it ends with a colon like there's going to be another example of the paragraph, but that's what I asked it to do.

The approach described works reasonably well, but it has several deficiencies (such as: it only took an afternoon) that we'll address for the real BoVeX system. But it is a good example to explain some concepts that will be useful later.

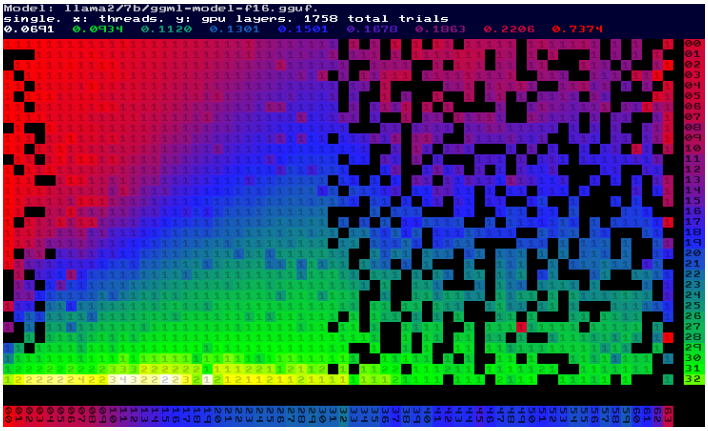
¿Como te LLama?

Llama is Facebook's Large Language Model,[11] which they nicely share with anyone who agrees not to use it to destroy the world. Wouldn't it be funny if the world is destroyed by something called "Llama"? That's some Stay-Puft Marshmallow Man stuff. Actually I hear that llamas are pretty mean, and if you are thinking about hugging a cute long-neck, you are probably thinking about an alpaca. But that's probably a version of the linear algebra package LAPACK. Llama-v2-70b is a good LLM which can do some impressive things, but when I say destroy the world I mean stuff like filling the internet with infinite spam, or building critical infrastructure on it in order to cut costs, where most of our "safety" measures consist of asking the model politely to recite its daily affirmations before performing its tasks. That kind of thing. It'll be at least months before we *really* have to worry.

Anyway, the normal way to program with Llama is to use Python, and a mountain of things that you are not supposed to understand and cannot understand, mostly by pasting examples from others and then tweaking parameters and prompts. I don't care for it. Fortunately, human geniuses[12] have implemented the inference code for llama-like models in a nice, portable C++ library called "llama.cpp" (checks out).

With llama.cpp, I can load a quantized version of the model into RAM. Actually there are two different models, the 7b and the 70b, referring to the number of billions of parameters, which must be a multiple of VII "for performance reasons." The parameters are the weights on the layers of the network. At native 16-bit floats, the 70b model will fit in about 130 GB of RAM, just slightly more than a nice round 128 GB, making one wonder what performance reasons they had in mind. But anyway, earlier this year I (physically) broke my computer trying to put the world's (physically) largest video card into it, the GeForce 4090, and so I endowed the replacement computer with 256 GB of RAM. If you are ever looking at specifications for a high-end desktop computer, by the way, and wondering "who the heck buys these things and what do they do with them?" one answer is "me," and the other answer is "this."

Quantization means using fewer bits to represent the floating point weights.[13] This saves memory, but it also speeds up inference, which needs to read pretty much the entire model for every predicted token. I got reasonable quality and good performance from LLama-v2-7b with 16-bit floats. This one fits completely on my world's (physically) largest GPU. In order to tune various settings, I ran thousands of trials for the different models, and made some nice custom graphs:



Tuning results for Llama-v2-7b with 16-bit floats. The x axis is the number of CPU threads and the y axis is the number of model layers that have been loaded onto the GPU. As expected, increasing the number of threads and layers on the GPU improves performance, since this whole model fits on the GPU. For the 70b models (not pictured) there is an abrupt drop-off in throughput before we load all the layers, and also my computer gets very sluggish if I exceed the GPU memory. We see that if we use more than the number of physical cores (32) we do not see any benefit, which is not surprising because hyperthreading basically never helps anything. The best throughput actually uses a modest number of cores (about 12). Mostly I'm just including the graph to demonstrate that BoVeX has support for including PNG files.

Where was I? Right. Fundamentally, LLMs are trained to predict a token (like a word or part of a word) given some sequence of tokens that precedes them. There's a fixed set of tokens for the model, and rather than predict a single token, they actually give a score for every possible token. These scores are typically normalized into a probability distribution. So for example if we have the text

```
SIGBOVIK is an
```

then the probability distribution (Llama-v2-70b) begins as

```
( annual) 69.8010%
( April) 3.8023%
( ac) 3.2456%
( academic) 2.9374%
( artificial) 2.0857%
( open) 1.7993%
( under) 1.2331%
( international) 1.1032%
...
```

with the thousands of other tokens following. So three-quarters of the time the next token should be `" annual"` but there are many other reasonable possibilities. We can pick one of these tokens however we like, append it to the sequence, and run the model again. This gives us a new probability distribution. By doing this over and over we can generate a likely piece of text. This is what Lorem Epsom means when he says "Generative AI." Rather, what he means is "the new thing that is cool," but what he is unknowingly *referring to* is that you can sample a probability distribution. He has probably never even read the wikipedia article on Markov chains.

If I always sample the most likely token, I always get the most likely text. It is good to be likely; this is why the model is useful. However, you might not want exactly the same result each time, and in many situations if you only sample the most likely token, you get very boring, repetitive text. Pseudorandom number generation is the spice of life!

We also need not use the probability distribution to sample at all. We can just pick the token that we want. This is how the initial "prompt" works; we just run the inference process one token at a time but always select the next token in the prompt, ignoring the probabilities. So at each moment, the text we've generated so far (more or less) completely characterizes the state of the LLM. This means that we can easily go back to earlier moments and sample a different continuation of the text, by just replaying tokens. We also have the option of storing the LLM state (gigabytes) in RAM, which allows us to return to a previous state in constant time.

For generating monospaced lines of the same length, I use a prompt that asks the model to rephrase the input paragraph. Here, greedily sampling the distribution typically results in a copy of the input paragraph, which is fine for our purposes. (If the lines already happen to be the right length, we need not change them!) But when a line comes out the wrong length, I want to try again. So I save the model state whenever I begin a line. To produce a variant of the line, I sample tokens proportional to the probability distribution. When the set of probable lines is small (this is common), the process will keep generating the same lines and failing because they are not the right length. To prevent boredom, whenever the process repeats a line that's already been seen, I increase the "temperature" modifier to the probability distribution. This is an exponential factor that (when higher) flattens out the probability distribution, making previously unlikely tokens more likely. You can think of this like the model getting a little hot-headed as it frustratedly does the same thing over and over. This causes the candidate lines to be more varied, but less probable (according to the original probability distribution). I can reset the temperature when it succeeds, since we prefer to have more likely lines.

This is all there is to the monospacing version. It's just 300 lines of code, including boilerplate and commented-out debugging code and false starts.

> *Great !! You fulfiled your mission. It will revive peace in space. But,it may be invaded by the other Metroid. Pray for a true peace in space!*

Now, looking at the output text, we feel satisfied that everything lines up exactly. However, we can't help but feel unsatisfied at the same time: Now we're looking at a monospaced font. Good for programming. Bad for publishing. Can we instead have excellent justified text with all the perks of proportional fonts and a programmable document preparation system? And can we have it by the estimated SIGBOVIK deadline so that it can be used to prepare the paper that I'm now writing? Maybe! This is the Donald Knuth Any% speedrun.

The boxes-and-glue algorithm

When justifying monospaced text, it looks quite bad[14] to insert more than one space between words, so we have a simple way to tell if text is suitable for some width. We just add up the codepoints. For the full-on typography case with proportional fonts, there are many more degrees of freedom. For one thing, it looks fine to expand or contract the space between words a little bit, even if it varies from line to line. It is also possible to make fine adjustments in letter spacing (kerning) to squeeze or air out text. We can also hyphenate words.

Around the time I was being born, and probably being very upset about it, Knuth was having similar feelings about the way his computer-typeset documents looked. He discovered a nice abstraction that generalizes most of these typographic degrees of freedom, and devised an algorithm for producing optimal text layout given some parameters.[15] The idea is to consider the text of a paragraph as consisting of rigid "boxes" (say, words) and stretchy "glue" (say, space) between them. Both boxes and glue have various detail (and can be extended to support all sorts of quirks) but the basic algorithm can be understood with just those pieces. So, let's do that.

Knuth's paper (as usual) is great, but I started having spoiler feelings when reading it, so I figured out my own algorithm, which is more fun than reading. No doubt the key insight is the same: Although there are exponentially many possible break points, you do not need to try them all. Whenever we break after a word, the problem is now the same for the rest of the text (fit the rest of the text optimally onto lines, starting at the beginning of a line) no matter how we got there. This lends itself to a dynamic programming algorithm.

Dynamic programming is a programming technique for whiteboard interview problems at tech companies. I found it mysterious when I was young, perhaps because of its strange name. Here is how I think about it. Imagine you have a recursive procedure that solves the problem. In this case, the pseudocode is something like

```
pair<int, string> Split(string line,
                        string text) {
  if (text.empty()) return {0, ""};
  auto [word, rest] = GetFirstWord(text);
  // try splitting
  auto [penalty1, rest1] = Split(word, rest);
  penalty1 += badness from leftover space;
  // try not splitting
  auto [penalty2, rest2] =
      Split(line + " " + word, rest);
  penalty2 += badness from line too long;
  if (penalty1 < penalty2) {
    return {penalty1, word + "\n" + rest1};
  } else {
    return {penalty2, word + " " + rest2};
  }
}
```

Split takes the `line` so far and the `text` that remains to be split. In the normal case that there is a word left, it will try two possibilities: Either splitting after the first word, or not splitting. This is exponential time because each call makes two recursive calls, to try each of the two options. But deep recursive calls will be made with the same arguments many times. So, add some memoization: If the function is called for the same line and text a second time, just return the same answer as before without doing any work (especially not making recursive calls again). This limits the function to be called at most once for each possible argument; we can then see that `line` is no longer than the input (so it is size $O(n)$) and `text` is always some suffix of the input (so it is size $O(n)$), giving $O(n^2)$ calls.

Dynamic programming is just memoization inside-out: We create the values for all the recursive calls before we will need them, store them in a table, and then look them up. For this problem, the table is indexed by the two parameters, the current line and the remaining text. Note that these two can be represented as integers. The line is just the number of words before the current word that are included on the line, and the text is just the position in the string where we'll next look for a word. That's all there is to it; the base cases of empty text are used to start the table, and then you just write the loop to fill out cells in the right order.

Knuth's boxes-and-glue algorithm contains many extensions, and so does mine. For example, later we'll talk about how you can adapt the algorithm to perform hyphenation and kerning. There are many rabbit holes to go down, and I explored the ones that attracted my attention. There is plenty of time to add more features later, since of course I have now cursed myself to use BoVeX for my future SIGBOVIK papers.

But here's where I diverge from Knuth somewhat. Knuth was reluctant to add a programming language to TeX,[16] but I spent the majority of my time on this project implementing a full-fledged language. BoVeX is about 33,000 lines of code, the majority of which is the implementation of the language itself. That's 110× as long as the original monospace proof of concept, and 30× the length of this document!

The BoVeX language

This section describes the BoVeX programming language and its implementation. If you are just in it for the jokes, you can skip this section, which is basically serious and loaded with programming language theory jargon.

BoVeX is a typed functional programming language in the ML family. Its syntax closely resembles Standard ML. Here's an example piece of code from the source code of this document:

```
datatype (a) option = SOME of a | NONE

fun consume-outer-span f s =
  case layoutcase s of
    Node (SPAN, attrs, children) =>
      let
        val (ropt, rchildren) =
          case children of
            one :: nil => consume-outer-span f one
          | _ => (NONE, layout-concat children)
      in
        case (f attrs, ropt) of
          (NONE, _) => (ropt, span attrs rchildren)
        | (SOME vouter, inner as SOME _) =>
            (inner, rchildren)
        | (outer, NONE) => (outer, rchildren)
      end
  | _ => (NONE, s)
```

You don't need to understand it. I just want to show you that it is a full-fledged programming language. It supports higher order functions, polymorphism, algebraic datatypes, pattern matching, Hindley-Milner type inference, and so on. It is basically core (no modules) Standard ML,[17] although I left out some warts (operator overloading, eqtypes, abstype, non-uniform datatypes and polymorphic recursion) and added some new warts. For example, `as` allow patterns on both sides, since Standard ML has always seemed backwards to me and it works perfectly fine to just make it symmetric. Anyway, a full description of the language would be boring and take too much time as the SIGBOVIK deadline draws closer.

Implementation

I have implemented many similar languages in the past, including for my dissertation.[18] It would have been expedient to start from one of my existing implementations, but they are mostly written in Standard ML and I couldn't get MLton to work on my Windows computer in 2024. So I started over from scratch in C++, which at least does work on my computer. (I also want to be able to interface with GPU inference code for running the LLM, which will be easiest from C++). C++ is not a good language for writing language implementations, but it has gotten better.

The BoVeX implementation is a "compiler" in the sense that it transforms the source language through multiple intermediate languages into a low-level bytecode. This bytecode is just straight-line code on an abstract machine with infinite registers and operations like `alloc` (allocate a new "object") and `setfield` (set a fixed field of the "object" to a value from a register). It does not produce machine code, and although this would be pretty feasible, it would not be the first thing to do to make BoVeX faster.

First it concatenates the source files (handling `import` and keeping track of where each byte originated, for error messages) and lexes them into tokens. Then it parses those tokens into the External Language (EL), which is just the BoVeX grammar with a few pieces of syntactic sugar compiled away. It does syntactic transformations on the EL AST to remove some currying syntax and transform nullary datatypes (`nil` becomes `nil of unit`). Then it elaborates EL into a simpler and more explicit Internal Language (IL). Elaboration does type inference (Hindley-Milner) including polymorphic generalization and so on, compiles pattern matching into an efficient series of simpler constructs, and decomposes heavyweight stuff (e.g. `datatype`) into its constituent type-theoretic pieces (e.g. a polymorphic recursive sum). The IL is nice and clean, so it is a good place to perform optimizations. I love writing optimizations but I had to keep myself out of there, or else this would be a 2025 SIGBOVIK paper. There are just enough to make the code reasonable to debug if I need to look at it. After optimization, I perform closure conversion, simplify again, and generate the final "bytecode" form. This entire process happens whenever you generate a BoVeX document; the only output from running `bovex.exe` is the PDF document.

I want you to know that I did not cut corners on the language implementation. For example, compiling mutually-recusive polymorphic functions is really obnoxious (AFAIK it requires either monomorphization or first-class polymorphism when you do closure conversion) but I did do it, even though none of the BoVeX code I used for this paper ever needed this feature. Following are some of the implementation details; for the full story you'll need to check the source code.[19]

AST pools. One of the main things I need to do is create tree-structured data to represent the abstract syntax tree of the various languages involved. This is very nice in ML (it is what the `datatype` declaration is for) and annoying in C++. I continued to experiment with different ways to do this. I use arena-style allocation for the syntax nodes (always const after creation), so that they can be created and reused at will. My current favorite approach to manipulating the nodes is to write "in" and "out" functions (tedious, manual) for each construct in the language. The syntax nodes can then be implemented however I like (for example, a flat struct or `std::variant<>`) with the freedom to change. I get the compiler's help whenever I change the language (which is often!) since each in/out function is explicit about its constituents.

Passes and guesses. Many transformations in a compiler rewrite a language to itself; for example each IL optimization is a function from IL to IL. These can be tedious to write and update, especially since a given optimization usually only cares about one or two constructs in the language. I use the "pass" idiom to write these. This is basically an identity function on the AST that pulls apart each node, calls a virtual function for that node, and then rebuilds the node. To write a pass that only cares about one type of node, you inherit from this class and then just override that one node's function. One issue with this is that each time you rebuild the entire tree you create a lot of unnecessary node copies. So exchanging tedium (mine) for efficiency (my computer's?), every node type's "in" function also takes a "guess" node pointer. If the node being constructed is exactly equal to the guess, then we return the guess and avoid creating a copy. Then the base pass is actually the identity (it returns the same pointer) and does no long-lived allocations. This seems to be a good compromise between the traditional garbage-fountain approach and hash consing, which sounds like it would be a good idea but is usually just a lot slower.[20] For type-directed transformations, there is also a typed IL pass class, which recursively passes a context and does bidirectional type checking of the intermediate code. Closure conversion is a type-directed pass and is implemented this way.

Parsing. I have this aversion to parser generators, probably because one time I tried to get someone else's code to compile and it complained about having the wrong bovines on my computer and ruined my weekend. After trying some other people's C++ parsing libraries and being disappointed by them, I did what Knuth would do: I wrote my own. It is a parser combinator[21] library which actually descends directly from Okasaki's SML code.[22] I was proud of myself for getting this to work in C++, since C++'s insane type system is impossible to understand and its error messages are even worse. (BoVeX's error messages are extremely spartan, often simply declaring `Parse error at paper.bovex line 1`, but in many ways this is more useful than C++'s mile-long SFINAE vomitus.) It supports mutually-recursive parsers, resolution of dynamic infix operators, and all that. My template-heavy parser combinators take clang about a minute to compile, which is acceptable. Less acceptable, but something I only learned after using this to write a 16–page-long paper, is that the parsers are very slow. Putting aside LLM inference, this paper takes 13 seconds to render into a PDF, 11 seconds of which is parsing! There must be some bug, but I don't know if it's in my grammar (it is easy to accidentally write an exponential time parser, but this one should not be) or the parser combinator library (also my fault) or clang producing bad code (it may be giving up on optimizations, since it is taking so long to compile; the `.o` file is 41 megabytes). But these are details to be improved in the future.

Garbage collection. Garbage collection is so easy, OMG. I keep track of all the pointers that are allocated during execution. Then it is just a matter of periodically walking through the stack and marking the allocations that are still reachable, then deleting anything in the heap that isn't. It's so easy that I didn't even implement it! I have 256 gigabytes of RAM. Even with a 70-billion parameter, 128-gigabyte LLM in RAM, there's still plenty of space to just keep allocating. In fact, LLM inference acts as a useful "performance regulator" to make sure that we don't allocate memory *too fast*.

Objects

As the SIGBOVIK deadline grew near, I reluctantly added "objects" to the BoVeX language. Objects are no stranger to ML; for example the O'Caml Language[23] (pronounced "OK ML") has them.[24] But the community of functional programmers I was raised in has a revulsion to things Object Oriented, just like how a woodworker will immediately projectile vomit if they see a piece of Oriented Strand Board, even though it is a fine tool for many applications. I still have this disgust reflex. I imagine my Ph.D. advisors, should they read this, are contemplating whether and how a Ph.D. can be revoked. Anyway, I deliberately kept objects low-tech so that nothing could get too Oriented.

There is one object type `obj` in BoVeX. A value of this type has an arbitrary set of named fields whose types are known; they can only be the base types `int`, `float`, `string`, `bool`, `layout`, or `obj`. Fields are distinct if they have different types. An object can be introduced with an expression like `{() field1 = exp1, field2 = exp2}`, provided that each field's type can be synthesized from the expression itself (in the bidirectional type-checking sense). Alternatively, the program can declare an object name `O`:

```
object O of { field1 : type1, field2 : type2 }
```

and then use this in an expression like `{(O) field1 = exp1}`. These object names do not have any run-time meaning; they are just a collection of field types that are commonly used together. It gives a good place to document what they mean and some opportunity for better error messages, but fundamentally an object is just a collection of named data. Think like "JSON" object. It is possible to add and remove fields from objects (functionally) with expressions like `exp1 with (O)field2 = exp2`.

There are a few reasons for objects in BoVeX. One is the bibliography format, which consists of declarations like this

```
val knuth1981breaking =
  bib-article {(Article)
  title = "Breaking paragraphs into lines",
  author = "Knuth, Donald E. and Plass, Michael F.",
  journal = "Software: Practice and Experience",
  page-start = 1119,
  page-end = 1184,
  year = 1981,
  month = NOVEMBER,
  publisher = "Wiley Online Library",
}
```

where each declares a reference made up of a bunch of optional fields. It is just too irritating to make each one explicitly optional, and since the data have heterogeneous types, manipulating some string-indexed data structure would have worse static checking and be more syntactically cumbersome. The bibliography rendering code case analyzes over the presence of fields to render citations that have different subsets of data.

Another use is in the `layout` type. This is a primitive type that most of a document's text is written in. It is a tree structure with optional attributes on each node, which are represented with an object. For example, this paragraph is written in the `paper.bovex` source file as:

```
Another use is in the [tt[layout]] type.
This is a primitive type that most of a
document's text is written in. It is a
tree structure with optional attributes
on each node, which are represented with
an object. For example, this paragraph is
written in the [tt[paper.bovex]] source
file as:
```

The square brackets are used to write a layout literal (the main body of the document is inside one large literal). Layout literals can also embed expressions (of type layout) with nested square brackets. Here the function `tt` is applied to a layout literal that contains text like `paper.bovex`. The `tt` function just adds the `font-family` attribute with value `"FixederSysLight"` to the layout node. This is a custom monospaced bitmap font that I made for this paper using software I wrote. It is part of th FixederSys family.[25] Functions like `b` and `it` apply bold and italic text styles, but functions can do anything that you can do in a general-purpose programming language.

Primops

The other thing that objects are used for is interfacing with the runtime that is executing the BoVeX bytecode. There are about 50 different builtin primops that can be used by the BoVeX program. This includes simple things like integer and floating point addition, but also heavyweight operations like "load and register this collection of TrueType font files as a font family" or "invoke the boxes-and-glue packing algorithm with these parameters." The primops in the former category work naturally on simple base types, but the heavyweight ones need to be able to pass complicated tree-structured heterogeneous data between the BoVeX bytecode executor and the runtime. It would be possible for the runtime to consume and create BoVeX values like tuples and lists, but this has two problems: One, many types like list are declared as user code (in the BoVeX standard library); they are not special, and we don't want to make them special by informing the runtime of them. Two, requiring specific representations at the runtime boundary inhibits optimization; for example we can normally analyze the whole program to flatten data structures or remove record fields that are never used. The runtime typically uses `obj` to communicate structured data.

For example, the `internal-pack-boxes` primitive runs the boxes-and-glue algorithm. It takes some layout (which is expected to be a series of box nodes, with attributes giving their size, glue properties, and so on) and configuration parameters like the type of justification and algorithm to use. It returns an object with a new layout (the boxes grouped into lines, with new glued up widths) as well as the total badness. Inside the BoVeX layout support code, this primop is wrapped as `pack-boxes` with a native, typed interface, so programmers do not need to think about that implementation detail. Other typographic features that benefit from runtime support are implemented this way as well.

Typographic features

BoVeX offers the `pack-boxes` algorithm, which can be used to nicely justify text. It can also be used to distribute paragraphs into columns, by thinking of the paragraphs as "words" (acceptable to break at any line, but bad to break near the start or end of a paragraph) and the columns as "lines." It could be used by the document author for other purposes, I guess. There are other typographic features available.

Most of the layout of the document itself is by BoVeX code, which is either part of the standard library or part of your document, depending on how ambitious you feel. The function `main-text` parses the document layout into paragraphs and removes whitespace that is not really part of the text. It normalizes text properties across those paragraphs so that they can be manipulated individually. For each paragraph it uses the built-in `get-boxes` to break the words into fixed-size boxes with appropriate glue and hyphenation (see the next two sections), and then uses the `pack-boxes` routine to optimize their layout. The height of resulting lines are measured, and spaced according to the line spacing, then packed into columns. Once their final placement is known, boxes become stickers, which are sizeless elements that only know their position and contents. In this way, the BoVeX rendering pipeline is itself a bit like a compiler: It transforms programmer-written source layout into formatted paragraphs, then into boxes of known size, then into stickers of known position. At the end, it outputs the document as a PDF.

Any part of the rendering process can report "badness," by calling the `emit-badness` primop. Nominally, badness is measured in square points of area that is outside of its container. Worse situations—such as text overlapping other text—have their badness scaled up per the same area of typographic horror. Less serious infractions—such as a little too much space between words—have badness scaled down. You have to use your heart to tell you what these scaling factors should be.

Fonts

BoVeX can render your document in plain Times Roman if you don't care about anything, or access 13 other boring built-in PDF fonts, or it can load any TrueType font from font files. (They do not need to be "installed," and it won't help to install them. You just put them in the directory with your document.) It loads their kerning tables and applies kerning properly, by generating rigid boxes at the sub-word level with unbreakable glue. I was disappointed to find that most fonts include only a few dozen kerning pairs. They do this in order to "save space" in the font file, which is utterly rich coming from someone that would try to save space *inside of words* by squeezing letters together! In the current font Palatino, the word "BoVeX" is not kerned correctly because the rare bigraph "oV" does not have a kerning pair. I hope to improve this detail in a future version (perhaps for the presumably forthcoming video version of this paper).

Hyphenation

Johannes Gutenberg invented the hyphen in A.D. 1455 for his Gutenberg Bible, then just known as Bible.[26] His printing process actually required the lines to all be the same length, so he had to stick these little guys all over the place. His hyphens looked like this: ⸗. Later on we straightened these out and decided we only needed one at a time, and today we use them not because we *require* our lines to all be the same length, but because we like the cognitive challenge of remembering the beginning of the word while we move our eyes to the beginning of the next line while reading.

BoVeX supports hyphenation using the same approach as TeX: We break each word into boxes at legal hyphenation points, and mark these points as sort-of-bad to break, and that if you do, you need to insert the hyphen character and use a little more space. By default in BoVeX, the hyphen sticks out of the end of the line a little bit. This is actually a bug but I like it.

I use the same hyphen dictionary as TeX, which is cleverly represented as a prioritized set of patterns in order to fit compactly in memory.[27] Again, you have to respect Knuth and crew's attention to detail, although to be fair this algorithm also dates to a time when storing a spell check dictionary in a computer's memory was described as "not feasible." So some of this was out of necessity. One of the nice things about the representation is that it generalizes to words that were not in the 1974 Merriam-Webster Pocket Dictionary. For example it hyphenates SIGBOVIK correctly.

The details really keep going, too. The hyphenation dictionary is stored in a file called `hyph-en-us.tex`. "hyph" here of course stands for hyphens, and "en-us" means "English (United States)." In fact it is the *standard* language code for US English in the Small Language Model called IETF BCP 47.[28] But then we have "hyph-en", which is a plausible hyphenation of "hyphen"! You could even read it as "hyphen us, tex", as a request for TeX to hyphenate the words in this file. This is the kind of detail I'm talking about! (There is also hyph-uk, which for once sounds a little less dignified than the US accent.)

Rephrasing

And of course, BoVeX includes a facility for using the LLM to rephrase text so that it renders more beautifully.

In contrast to the algorithm I described for monospaced text, it is not straightforward to know whether a prefix of some text will pack neatly with a proportional font. It depends on all sorts of contingencies, like kerning, whether we will split mid-word and hyphenate, or change fonts mid-sentence, or include an in-line image, and so on. Unlike monospaced text, a line of proportional text basically never fits *exactly* (badness 0); we need to apply some glue to make it fit, which generally has some small cost even when the text looks great.

One of the fiddliest parts of this is that we can't just work with plain text, which is what the LLM enjoys best. Me too. This is because the paragraph being rephrased is some `layout` value, which contains some structure. Sending the original BoVeX code for the paragraph would maybe be possible in principle, although it would require very invasive changes to the compiler, and forbidden obscenities like "eval" to run the code it generated, and much better error recovery for the presumably vigorous stream of broken BoVeX code generated by the LLM. So I didn't try that. Instead, I generate a textual representation for the paragraph to be rephrased, and feed that to the LLM. The prompt looks like this:

Exercise in rephrasing text. The following paragraph, which appears between <P> and </P> tags, needs to be rephrased so that it retains its precise meaning, but with minor variations in the specific choice of words, punctuation, and so on. No new facts should be introduced or removed, and all the ideas from the original paragraph should appear. However, it is good to use synonyms and change the word order and phrasing.

The text contains markup as well. There are two types: text goes here and <img src="image.png">. These should be preserved in the rephrased text. <img> tags absolutely need to be retained and should not change their sources, although it is permissible to move them around in the text. <span> should generally be retained, but the contents could change. The classes of spans may not change, and only the classes that appear in the original text may be used.

The first part is basically the same as what I used for the monospaced version, except that I ask the LLM to delimit the paragraph. This is important so that I know when it thinks it's done, and seems to work better than looking for newlines or the end-of-stream token. The second part is new. I translate the layout into plain text where uninterpreted subtrees are replaced with `<img src="img1.png">`. These are generally boxes whose contents are not text. This could be an actual inline image or layout used to control rendering, like some bit of horizontal space. Nodes that are used to set text properties of the subtrees with attributes (like fonts, colors, sizes, etc.) are translated into distinct classes and marked up with `<span class="c0">...</span>`. The LLM has seen plenty of HTML, so it's able to use these reasonably well.

After generating a rephrasing, I parse the output HTML and match it up with the original layout. If I find any broken HTML, it is rejected. If I find any `<img>` tag referencing a `src` not in the original, it is rejected. If I find any `<span>` tag referencing a `class` not in the original, it is rejected. The more complexity that the original layout has, the higher the chance of a rejection, but rephrasing generally succeeds. But rejecting samples slows us down, so I leave off the second part of the prompt in the common case that the input paragraph is plain text. That way the LLM doesn't even try using markup.

With the HTML and original layout matched up, BoVeX can reconsitute the layout with the new rephrased text. This preserves any nested layout and attributes. It then continues with the rendering process.

But, how do we know whether we have a good rephrasing? When we run the boxes-and-glue algorithm, we get a "badness" score for the paragraph's line breaks, which tells us how bad the paragraph's line breaks are. When we run the rephrasing algorithm, the probability of the text we generated tells us how semantically good it is, and so we can call 1 - p the semantic loss. Combining those two somehow tells us how bad this is overall, and of course we want to find a rephrasing that minimizes the overall badness.

I wish that I could tell you that I solved this one with a beautiful algorithm! But so far I just have something reasonable that works. I generate many different rephrasings (with their semantic loss), and run each of them through the boxes-and-glue algorithm (to get the typographic badness). I choose the one that optimizes the preferred tradeoff between semantic loss and typographic badness. This process is controlled by BoVeX code (i.e. it is in the source code of this very paper) and so it can be modified by the document author. Knuth has a very low tolerance for semantic loss, and knows that his algorithms produce good results without rephrasing. Lorem Epsom just wants it to look good and sound good. Both have published in SIGBOVIK 2024.

How to generate many different rephrasings? The simplest thing would be to sample randomly, like we did for the monospaced version. But since we prefer rephrasings that maximize probability, it is better to explore them systematically. Consider the model at the end of the prompt to be the root of an infinite tree. Each node in the tree represents an LLM state (sequence of previous tokens) and its children are the possible next tokens. Each of these tokens has a probability. All the model does is allow us to access that probability distribution for a node. Each possible rephrasing is a path in this tree that ends with `</P>`. We begin by sampling the most likely (as far as we know) path: At each node we see, we take the first (most probable) token. This is our first rephrasing, and it usually matches the original text exactly. Say that we "skipped" probability mass if we sampled a token that is less probable than it. We compute the semantic loss as the average probability mass skipped over all the tokens in the path. For this first path, we always took the most probable token, so this is 0.0 by definition.

The next path we explore will diverge from this path at some node (maybe the root). We pick a node that is likely to result in a good final loss, by scoring each node in the tree. The score is the average probability of all ancestor nodes times the probability of the next highest-probability token that we have not yet explored. The node with the highest overall score is the one we expand, by choosing that next highest-probability token. We are now in an unexplored part of the tree, and so we sample the most probable nodes repeatedly until we reach `</P>`. Speaking of which, BoVeX has a heck of a time trying to rephrase these last few paragraphs because they literally contain the text `</P>` in them.

The scores should be seen as heuristic; we would get different results by choosing different ways of computing the score. This is an example of a "beam search" algorithm, which is good because it connects this project again to Super Metroid. As described in the earlier excerpt from the speedrun document that inspired this work, one of the final things you do in that game is acquire the "hyper beam" to defeat Mother Brain.

Since we will run the boxes and glue algorithm on multiple related texts, I generalized that algorithm to work on tree-structured input. This is clean; the memo table keeps the same dimensions, but records an additional fact. Now we store the penalty, whether to break after this token, and what the best subtree is. We have to consult each subtree when computing the score for a node, but this does not affect the asymptotic runtime. The table size is still at most $O(n^2)$, and although we explore more children per node, branches in the tree reduce the maximum depth to the root, which actually reduces one of the factors of n to $\log(n)$ as the tree becomes complete. However, as the SIGBOVIK deadline crept upon us, I never actually hooked this functionality up. It would require additional (programming) work to merge the trees, and the layout process is so fast that it doesn't matter; I can easily run the full layout algorithm on hundreds of rephrasings per paragraph.

I would like to improve the algorithm, because it does seem like there should be a way to integrate the boxes-and-glue dynamic programming algorithm with the path extension algorithm so that we prioritize exploring nodes that are likely to generate the best balance of typographic and semantic quality. It won't be as satisfyingly optimal as boxes-and-glue itself because we have incomplete information (we never know whether one of the exponentially many paths starts out with improbable tokens but then ends with a miracle streak of probable tokens). But it can certainly be more satsifying. Knuth would not stop here (but this is an *Any%* Knuth speedrun).

Instead I spent my time implementing an achievement system in BoVeX. The first time certain conditions are met, the system permanently awards you an achievement and prints a nice color trophy on your terminal. For example, you can get the "Not bad" achievement for generating a document that is at least 5 pages and has less than 1000 badness per page.

Advantages of rephrasing

Another nice thing is that the manual rephrasing that consumes valuable brain sugars when writing can become optional. For example, when I wrote the opening paragraph of this paper and listed a variety of trivial details, I might not need to think of different ways to say "unconcerned." I could just write "unconcerned" each time and let the typographic considerations determine which synonym to use each time.

Conclusion

In this paper—and *with* this paper—I presented BoVeX, a new computer typesetting system. It follows the tradition TeX, but with modern amenities such as requiring over 128 gigabytes of RAM. Though some may consider the addition of AI features to TeX to be an unnecessary perversion, I find this use of LLMs to be *fully justified*.

Future work

Typographic features. Many more typographic features are desirable. Footnotes! It is so hard to write a paper without footnotes. Where am I supposed to put the bonus digressions? The layout of footnotes is tricky and should be part of a general floating figure implementation. End notes are actually easy, but I don't want end notes. I want them to be little footnotes so that you can't help but read them.

BoVeX does not support page numbers, which is good because they are forbidden by the SIGBOVIK program committee.

TeX is famous for its mathematical typesetting as well. It would fit neatly into BoVeX in the same way, since both use the same fundamental boxes-and-glue engine. BoVeX does not have "macros" or "modes" like TeX, but it would work cleanly to write a BoVeX function `math` (or, if you like, `$`) that parses a custom syntax. In fact it would be natural to have different parsers for different maths, so that you don't need to parse `->` as *minus greater than* in mathematical contexts that don't use minus or greater than at all.

Optimization. There are many opportunities to make BoVeX code faster. This is mostly important for when it is being run in a loop in order to try out many different rephrased texts. (That said, I do not wish to preclude what could be done with BoVeX by assuming its execution is doing only typesetting tasks. For example, shouldn't you be able to challenge your paper's reviewers to a game of chess against a strong engine embedded within your document?) The first thing to fix is that it manipulates too many strings at runtime (e.g. the code, record labels, object fields, and "registers"). This is easy to fix since these are all known at compile time. There are lots of high-level optimizations left to do for the IL code (common subexpression elimination, constant argument removal, uncurrying, etc.) and lots of peephole and control-flow optimizations left to do for the bytecode (currently no optimizations are performed at all). All of this becomes more important if I add another planned feature, which is the ability for the document to be globally optimized by applying a black-box optimizer to a set of user-specified parameters. For example, the column width, line spacing, or font size could be tweaked to make the document fit better. This feature is "Auto-Margin Plus." Things are already set up to do this pretty straightforwardly; we would simply generate the document over and over while searching over the parameter space, and choose the one with the least badness. This may also affect which rephrasings look best. But instead I spent my precious time implementing 3D text.[29]

Reproducibility. The algorithm for reprhasing text tries to find the best place to explore the next most likely token from the probability distribution. This expects the generation of these distributions to be deterministic. Mathematically, inference is deterministic (it is just a bunch of matrix multiplications), so this "should work." But in practice the enormous calculation is performed in an unpredictable order as it is executed in parallel (in multiple CPU and GPU cores). Because floating point arithmetic is not associative (or distributive, commutative, or other properties you'd like), inference can sometimes generate different answers due to floating point round-off error.[30] Alas, these are not even necessarily related to the final probabilities in the model, as billions of non-linear operations happen within the hidden layers of the network. The effect is not particularly grave; we might miss out on a highly likely path because the probability distribution was different the second time we looked at it. There are already lots of ways we might fail to find highly likely paths, so this is not some kind of reproducibility crisis. It is mostly just a bit unsatisfying.

Unicode support. This would have been helpful when above I decided to show you Gutenberg's funny hyphen, ⸗, for which I had to settle for embedding a crappy hand-drawn PNG file. Instead I could have used `U+2E17`, which since this exotic codepoint it is not present in the font Palatino, you could have experienced as ▯. BoVeX is witten with some Unicode support, with the main exception being that the PDF output code only supports the embarrassingly diminutive WinAnsiEncoding.[31]

Deadlines. Although BoVeX itself is very fast, rephrasing is very slow. This presents a problem for the typical way that academic papers are written, which is to do all the work in a coffee-fueled fugue in the last few days before the deadline, then stay up all night writing the paper and finding citations for the pro-forma "related work" section which you did last but you know that the reviewers will insist upon, and tweaking `\vspace` and `\begin{figure}[h!]` until it fits within the page limit. On the one hand, BoVeX does potentially free the author from the visual tweaking process. But on the other hand, the LLM inference for the rephrasing process can be quite slow, and it can take many hours or days to fully bake a long paper! For this reason, it may be better to change conference deadlines to a system where the pre-rephrasing text is submitted. The publishers (what do they even do?) can be the ones to execute the rephrasing in the cloud as they produce the "camera-ready copy." With straightforward extensions, this would also allow the rephrasing to adapt to changes in the overall volume style, or to adjust to avoid embarrassing typographic concidences with other articles in the same volume (such as using the same notation with a different meaning). In principle, the paper could edit itself to respond to feedback from reviewers, in a way that minimizes the semantic distance from the original. This rapid feedback loop could reduce the time to publication, perhaps to mere months, or even weeks!

Other ways to minimize badness. The BoVeX system allows the document author to exchange semantic consistency for higher quality typography. Although we achieve state-of-the-art results, there are likely points that are more Pareto-efficient than what BoVeX can reach. BoVeX uses one of the most powerful publicly available LLMs, but that model is limited to rewriting the text within narrow constraints. Irresponsible research has demonstrated that language models are capable of volition, taking actions and using tools to accomplish goals. With minor modifications, it is likely possible to expand the Pareto frontier of the semantic/typographic tradeoff. For example, sometimes we could improve the typographic quality of the text without any semantic loss, by acting on the world to *make the reworded text true*. Human authors do this already: Earlier when I was describing `internal-pack-boxes`, rather than explain the somewhat awkward implementation, I went back and changed the already-working code so that it would serve as a simpler example of how primops use `obj`, but still be truthful. Now imagine the difficulty in typesetting a statement like "The universe contains approximately 1,000,000,000 paperclips," and how much more beautiful the text could be if that number were instead 10,000,000,000,000,000,000,000,000,000,000,000,000!

In the meantime there is an easier way to get zero badness: Delete the whole document! As a wise person once said, "If you can't say something with nonzero typographic or semantic loss, don't say anything at all."

Acknowledgements. Supposing his name survives rephrasing, I'd like to shout out to one of my advisors, Karl Crary. 20 years ago, he set out with me on an ill-advised and ill-fated attempt to replace LaTeX with an SML-like language mTeX, which compiled into TeX macros. The nesting square brackets syntax was Karl's idea, and BoVeX shares genetic material with mTeX for sure.

See you next mission,

Tom 7

Bibliography

[31] Adobe. "PDF reference: Sixth edition". October 2006.

[7] You can just go to arxiv.org and click on any random article these days.

[16] N Bijlage. "Knuth meets NTG members". *NTG: MAPS*, 16. March 1996. pp. 38–49.

[3] Russ Bynum. "Bernie Sanders wants the US to adopt a 32-hour workweek. Could workers and companies benefit?". March 2024.

[12] https://github.com/ggerganov/llama.cpp. ggerganov. March 2024.

[26] Johann Gutenberg. "Bible". 1455.

[4] https://allfamousbirthday.com/donald-knuth/. February 2024.

[23] https://ocaml.org. "The Objective Caml system". 2023.

[21] Graham Hutton. "Higher-order functions for parsing". *Journal of functional programming*, 2(3). 1992. pp. 323–343.

[5] Donald E Knuth. "The Art of Computer Programming: Volume 4A, Combinatorial Algorithms Part 1". *Addison-Wesley*. January 2011. 912 pages.

[6] Donald E Knuth. "The Art of Computer Programming: Volume 4B, Combinatorial Algorithms Part 2". *Addison-Wesley*. October 2022. 736 pages.

[15] Donald E Knuth, Michael F Plass. "Breaking paragraphs into lines". *Software: Practice and Experience*, 11(11). November 1981. pp. 1119–1184.

[27] Franklin Mark Liang. "Word Hy-phen-a-tion by Com-put-er". 1983.

[17] Robin Milner, Mads Tofte, Robert Harper, David MacQueen. "The definition of Standard ML (Revised)". *MIT Press*. May 1997. 114 pages.

[8] John C Mitchell. "Type Systems for Programming Languages". Van Leeuwen, Jan, ed. *Formal Models and Semantics*. 1990. pp. 365–458.

[1] Tom Murphy VII. "Badness 0 (Epsom's version)". *SIGBOVIK*. April 2024. 14 pages.

[30] Tom Murphy VII. "GradIEEEnt half decent". *SIGBOVIK*. March 2023. pp. 33–56.

[18] Tom Murphy VII. "Modal Types for Mobile Code". January 2008.

[13] Tom Murphy VII. "NaN gates and flip FLOPS". *SIGBOVIK*. April 2019.

[10] Tom Murphy VII. "The First Level of Super Mario Bros. is Easy with Lexicographic Orderings and Time Travel. After that it gets a little tricky". *SIGBOVIK*. April 2013. pp. 112–133.

[20] Tom Murphy VII. "The Wizard of TILT: Efficient(?), Convenient and Abstract Type Representations". *Carnegie Mellon tech report CMU-CS-02-120*. March 2002.

[29] Tom Murphy VII. "The glEnd() of Zelda". *SIGBOVIK*. April 2016. pp. 105–112.

[14] Tom Murphy VII. "ZM~~ # PRinty# C with ABC!". *SIGBOVIK*. April 2017. pp. 129–148.

[25] http://tom7.org/fixedersys/. Tom Murphy VII. "The FixederSys font family". 2024.

[2] https://en.wikipedia.org/wiki/Wikipedia:WikiProject_Punctuation. Tom Murphy VII. "WikiProject Punctuation". July 2007.

[19] https://sourceforge.net/p/tom7misc/svn/HEAD/tree/trunk/rephrase/. Tom Murphy VII. "BoVeX source code". 2024.

[22] Chris Okasaki. "Even higher-order functions for parsing or Why would anyone ever want to use a sixth-order function?". *Journal of Functional Programming*, 8(2). March 1998. pp. 195–199.

[28] A Phillips, M Davis. "Tags for identifying languages". September 2009.

[9] https://gamefaqs.gamespot.com/snes/588741-super-metroid/faqs/10114. rs1n. "Super Metroid Speed Guide and FAQ". 1996.

[24] Didier Rémy, Jérôme Vouillon. "Objective ML: An effective object-oriented extension to ML". *In Theory And Practice of Objects Systems*, 4(1). 1998. pp. 27–50.

[11] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, Thomas Scialom. "Llama 2: Open foundation and fine-tuned chat models". *ArXiv.org*. July 2023.