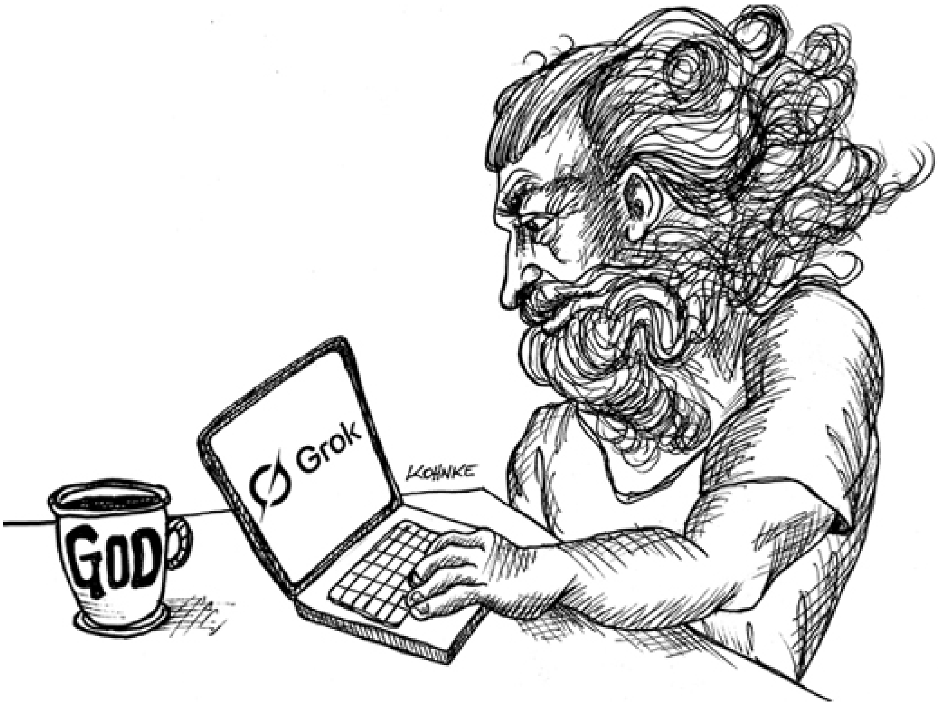
17

AIs, LLMs, and God Knows What



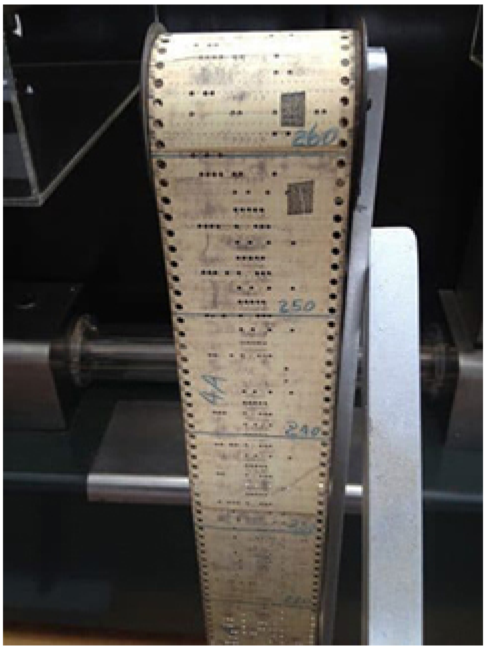
In the late 1940s, Grace Hopper wrote her first lines of code. Each line looked something like this[1]:

---

1. [We Programmers].

```
| 521|   53|   2|
```

Each digit represented a particular hole on a paper tape that looked like this:



Each row of that paper tape was an instruction for the Harvard Mark I to execute. The instructions were very simple—something like “Add Register 10 to Register 12.”

I’ll leave you to imagine just how hard programming was back then.

Within a few years, it became clear that a computer program could be used to generate codes like that from a more abstract textual representation like “add 10, 12.” Hopper called this *automatic programming*.

Before a decade had passed from Hopper’s first line of code, John Backus and his team had produced the first FORTRAN compiler, which used statements like “a = b + c.” Algol came a handful of years later, and within a decade, Dahl and Nygaard had created SIMULA 67, the first object-oriented language. C, Unix, Smalltalk, and C++ were born within a few years. And within the next few, Java and C# appeared.

At each one of those transitions, the productivity of programmers increased. The first leap in productivity was the greatest. In the ’50s, the productivity of programmers using a FORTRAN-like compiler was shown to be 45 times that of programmers using machine code.

Each productivity leap thereafter was smaller than the previous. C was probably twice that of FORTRAN. C++ was likely a 20% improvement on C. Java/C# was perhaps on the order of another 20%. Ruby/Python are likely 20% better than Java, and Clojure likely adds another 10%.

Each of these transitions in language was driven by an increase in abstraction. Each new language in that sequence was another step further removed from Grace Hopper's holes in that paper tape. And each of those steps was accompanied by the fear that programming was being ruined.

Yes, you read that correctly. At every stage, there were those who predicted the end of programming. Books were published that predicted the doom of the industry. They claimed that programmers would become as rare as dodo birds.

Every time we made programming a little easier, those of us who were invested in the current complexity were liable to fear that unwashed masses would take our jobs. And so we predicted the doom of the profession. And every time, those fears and predictions were the exact opposite of the actual outcomes. With every new advance, the number of projects and the demand for programmers grew.

Today, we face yet another such transition—the transition of AI. And there is, of course, rampant fear that programming is dead and that programmers will soon go extinct.

Baloney! This is just another increase in the level of abstraction of our languages—another step away from the holes in the paper tape. And as we come to grips with it, the number of projects and the demand for programmers will continue to grow.

Programming by Prompt

This is a very new technology, and we don't have a good handle on how to use it yet. This is reminiscent of our first forays into Java and the Web. Does anybody remember applets? It took several years before we worked out the rudiments of how to use Java/C# well. It took several more to refine those rudiments.

The same happened with Ruby and Python. At first we stumbled and bumbled around, and then we gradually figured out the best ways to proceed. And believe me, when FORTRAN first came out, there was plenty of fear, anguish, doubt, and bumbling and stumbling.

And so it will be with AIs. At this point, we have no idea what to do. We are just now barely able to bumble and stumble our way to write programs with prompts. We have no idea how to do it well.

So, for example, consider the following prompt that I gave to Grok[2] a few days ago:

2. Grok 3 beta (March 19, 2025).

```
write a clojure program that finds all sentences
in a given file that contain a given substring.
Choose one of those sentences at random and find
the next word in the substring, or a . if no word fol
Add the found word to the substring and remove the
first if the substring is longer than three words.
Repeat the entire process until the substring
ends in a . or has exceeded 50 words.
Accumulate all the words that were found and
return them appended to the initial substring.
```

Here's the code that Grok generated. It works—after a fashion.

```
(ns markov
  (:require [clojure.string :as str]))

(defn read-file [filename]
  "Reads the content of a file and splits it into ser
  (try
    (let [content (slurp filename)
          sentences (filter not-empty
                            (map str/trim
                                 (str/split content #
      sentences)
    (catch Exception e
      (throw (Exception. (str "Error reading file: "

(defn find-sentences-with-substring [sentences substr
  "Finds all sentences containing the substring."
  (filter #(str/includes? (str/lower-case %) (str/low
          sentences))
```

```
(defn get-next-word [sentence substring]
  "Finds the next word after the substring
  in the sentence, or '.' if none."
  (let [index (str/index-of (str/lower-case sentence)
                            (str/lower-case substring
    (if index
      (let [post-substring (subs sentence (+ index (c
            words (str/split (str/trim post-substring
        (if (seq words)
          (first words)
          "."))
      ".")))

(defn update-substring [substring next-word]
  "Updates the substring by adding the next word
   and trimming if longer than 3 words."
  (let [words (str/split substring #"\s+")
        new-words (conj (vec words) next-word)
        trimmed-words (if (> (count new-words) 3)
                        (subvec new-words 1)
                        new-words)]
    (str/join " " trimmed-words)))

(defn sentence-walk [filename initial-substring]
  "Main function to perform the sentence walking proc
  (let [sentences (read-file filename)]
    (loop [substring initial-substring
           accumulated-words []
           iterations 0]
      (if (or (= (last (str/split substring #"\s+"))
              (> iterations 50))
        (str/join " " (concat (str/split initial-subs
                      accumulated-words))
        (let [matching-sentences (find-sentences-with
                                  sentences substri
          (if (empty? matching-sentences)
            (str/join " " (concat (str/split initial-
                          accumulated-words))
            (let [random-sentence (rand-nth matching-
                  next-word (get-next-word random-ser
              (recur (update-substring substring next
                     (conj accumulated-words next-wor
                     (inc iterations))))))))
```

```
(defn -main [& args]
  (if (< (count args) 2)
    (println "Usage: program <filename> <initial-subs
    (let [filename (first args)
          initial-substring (second args)]
      (try
        (println (sentence-walk filename initial-subs
        (catch Exception e
          (println (.getMessage e)))))))
```

It is easy to be impressed by this. I certainly am. But it also terrifies me. I am not terrified by the power of the AI to generate code. I am terrified by the structure of that horrible prompt.

Look carefully at that prompt. What is the definition of `sentence`? What is the definition of `word`? What does the clause `the found word` mean? Are those dot characters punctuation or nouns? What does it mean to `accumulate words`?

Yes, Grok figured all this out well enough for this little experiment. But the program that resulted was by no means ready for production. It makes lots of dumb little mistakes, most of which have to do with the questions I raised in the previous paragraph.

Here, for example, is a sample run:

```
>clj -M -m markov
~/Documents/Books/Clean\ Code\ 2ded\ 20250311.txt "Th

The Single Responsibility Principle, and the Depender
relationships and wind up back at that first solutior
the project you were working on  it every day  busine
life
```

Notice the double spaces in the third and fourth lines. Why is it doing that?

I could fix those mistakes by adding clarification to the prompt. For example, I might add:

```
Make sure all accumulated words are separated by just
```

But there are a number of problems with this approach. First, I haven't figured out why those double spaces are there, so this change to the prompt is just a patch to fix the symptom. Second, am I sure that the clause `all accumulated words` will properly refer to the return value?

I could try it to find out; but that causes another problem. With each new change to the prompt, Grok *regenerates* the program from scratch as opposed to modifying the existing program. With all the ambiguities in the prompt, how can I be sure that the regenerated program doesn't break something that worked in the previous one? Will it use the same definitions of `word` and `sentence` that it did before? Will it interpret `the found word` the same as it did before? Given the statistical and fuzzy nature of the LLM algorithms, there's a good chance it won't!

Here is what Grok generated for that modified prompt. Notice that the two programs look like they were written by two different programmers.

```
(ns sentence-walker
  (:require [clojure.string :as str]
            [clojure.java.io :as io]))

(defn get-sentences [filename]
  "Read file and split into sentences"
  (-> (slurp filename)
      (str/replace #"[.!?]\s+" "$0|")
      (str/split #"\|")
      (->> (map str/trim)
           (filter not-empty))))

(defn find-matching-sentences [sentences substring]
  "Find all sentences containing the substring"
  (filter #(str/includes? (str/lower-case %)
                          (str/lower-case substring))
          sentences))

(defn get-next-word [sentence substring]
  "Find word following substring in sentence, or . if
  (let [idx (str/index-of (str/lower-case sentence)
                          (str/lower-case substring))
```

```
        substr-end (+ idx (count substring))
        remainder (subs sentence substr-end)]
    (if (empty? (str/trim remainder))
      "."
      (let [next-word (first (str/split (str/trim rem
        (if (or (nil? next-word) (empty? next-word))
          "."
          next-word)))))

(defn update-substring [substring next-word]
  "Add next word and trim if longer than 3 words"
  (let [words (str/split substring #"\s+")
        new-words (conj (vec words) next-word)]
    (if (> (count new-words) 3)
      (str/join " " (subvec new-words (- (count new-w
      (str/join " " new-words))))

(defn walk-sentences [filename initial-substring]
  "Main function to walk through sentences"
  (let [sentences (get-sentences filename)]
    (loop [current-substr initial-substring
           accumulated-words [initial-substring]
           iterations 0]
      (if (or (str/ends-with? current-substr ".")
              (> iterations 50))
        (str/join " " accumulated-words)
        (let [matching-sentences (find-matching-sente
                                       sentences current-
          (if (empty? matching-sentences)
            (str/join " " accumulated-words)
            (let [random-sentence (rand-nth matching-
                  next-word (get-next-word random-ser
                  new-substr (update-substring currer
              (recur new-substr
                     (conj accumulated-words next-wor
                     (inc iterations))))))))
;; Example usage:
;; (walk-sentences "sample.txt" "the")
```

This new program solves the double-space problem but creates a new problem at the end of the sentence. It also doesn’t deal with uppercase and lowercase very well.

```
clean code is hard Writing a system that handles one
```

Infancy

These are problems related to the infancy of the approach. In order to effectively program large systems by prompt, we are going to need AI tools that allow for incremental changes. We are also going to need ways to eliminate the ambiguities in our prompts.

In short, we are going to need a formalism—a prompt language that is unambiguous and precise and that cannot be misinterpreted by the statistical and fuzzy nature of the LLM algorithm. We'll need to come up with a way to precisely say what we mean.

We'll also need to come up with a way to mean what we say. Or, in other words, even if we have a perfectly precise language, it is easy to precisely say something that is wrong. To ensure that our precise statements are accurate descriptions of our intent, we should state our intent in at least two different ways (just as I did in this paragraph!).

When you say something in two different ways, you are overloading the concept you are trying to communicate. Such overloading is a way to ensure that any error in either of the two statements will be detected by the other. We error-prone humans absolutely require overloading.

Overloading is the strategy behind double-entry bookkeeping and the testing disciplines described in earlier chapters. It would be supremely irresponsible to trust a single set of precise statements written by a human if those statements aren't overloaded. What we are going to need is a way to create precise and overloaded prompts and a way to quickly verify their consistency.

A SWAG

So let me make a Scientific Wild-Axx Guess.

Imagine a prompt language that contains formal definitions and constraints overloaded by a set of BDD-style[3] `Given`-`When`-`Then` scenarios.

3. [RSpec].

Let's give that a try with a slightly simpler problem. Notice how much it looks like COBOL. That similarity might give us pause.

```
Language: Clojure
Name: Markov
Command line arguments:
  f a text file.
  s a text string.

Definitions:
  Word: A string of alphabetic characters.
  Sentence: One or more words separated by spaces anc
  a period, question mark, or exclamation mark.

Constraints:
  Ignore case in all comparisons.

Description:
  Find all sentences in f that contain s.
  Choose a sentence c at random with equal probabilit
  Return the word in c that immediately follows s;
    if none return a period.

Scenarios to be tested with speclj:
Given f contains the following lines:
  Oh, Hello there Bob, how are you today? I'm fine.
  Hello there Bill, I trust you are well.  How's the
  Ah yes, hello there John, great day!  Kowabunga!

When s is "not in any sentence"
Return "."

When s is "hello"
Then return "there"

When s is "are well"
Then return "."

When s is "hello there"
```

```
Then return "Bob", "Bill", or "John".
Assert a uniform distribution after 100 iterations.
```

Grok was quite happy to generate the following code.

```
(ns markov.core
  (:require [clojure.string :as str]
            [clojure.java.io :as io]))

(defn get-sentences [file]
  (->> (slurp file)
       (re-seq #"[A-Za-z\s]+[.!?]")
       (map str/trim)))

(defn words [sentence]
  (str/split (str/lower-case sentence) #"\s+"))

(defn find-following-word [sentences search-str]
  (let [search-str (str/lower-case search-str)
        matching-sentences (filter
                             #(str/includes? (str/low
                                               search-s
                             sentences)]
    (if (empty? matching-sentences)
      "."
      (let [chosen (rand-nth matching-sentences)
            word-seq (words chosen)
            search-words (str/split search-str #"\s+"
            idx (.indexOf word-seq (last search-words
        (if (or (< idx -1) (>= (+ idx 1) (count word-
          "."
          (nth word-seq (+ idx 1)))))))

(defn -main [f s]
  (let [sentences (get-sentences f)
        result (find-following-word sentences s)]
    (println result)))
```

And here are the tests Grok generated from the scenarios.

```
;; speclj tests
(ns markov.core-spec
```

```
  (:require [speclj.core :refer :all]
            [markov.core :refer :all]
            [clojure.java.io :as io]))

(def test-content
  "Oh, Hello there Bob, how are you today? I'm fine.
   Hello there Bill, I trust you are well.  How's the
   Ah yes, hello there John, great day!  Kowabunga!")

(describe "Markov"
  (before-all
    (spit "test.txt" test-content))

  (after-all
    (.delete (io/file "test.txt")))

  (it "returns period when string not found"
    (should= "." (find-following-word (get-sentences
                                       "not in any ser

  (it "returns 'there' when searching for 'hello'"
    (should= "there" (find-following-word (get-senter
                                           "hello")))

  (it "returns period when searching for 'are well'"
    (should= "." (find-following-word (get-sentences
                                       "are well")))

  (it "returns Bob, Bill, or John for 'hello there'
       with uniform distribution"
    (let [results (repeatedly 100
                    #(find-following-word
                      (get-sentences "test.txt") "hel
          freqs (frequencies results)
          expected ["bob" "bill" "john"]]
      (should= 3 (count freqs))
      (should (every? #(contains? freqs %) expected))
      (doseq [name expected]
        (let [count (get freqs name)]
          ;; Check if count is reasonably close to 33
          ;; with some tolerance
          (should (> count 20))
          (should (< count 50)))))))
```

Despite a valiant effort by Grok, these tests do not pass. That should drive home just how important it is to overload a prompt with testing scenarios!

```
Failures:

  1) Markov returns 'there' when searching for 'hello
     Expected: "there"
          got: "." (using =)

  2) Markov returns period when searching for 'are we
     Expected: "."
          got: "i" (using =)

  3) Markov returns Bob, Bill, or John for
     'hello there' with uniform distribution
     Expected: 3
          got: 1 (using =)
```

Debugging the code is relatively simple; but can you debug the prompt? And even if you can "fix" the prompt, are you sure it will generate code and tests that pass if you apply the prompt more than once?

The fact that Grok wrote the tests and yet could not use them to infer the desired behavior exposes the limitation of the LLM approach. LLMs are statistical in nature; they create their responses through probability and not inference. It is true that many inferences are trained into the statistical model; but the evaluation of the prompt does not infer in real time. Inference is the primary operation of reason. If LLMs do not actively infer, then LLMs do not reason!

If you debug the code that Grok produced, you will find that Grok apparently ignored the provided definition of "word" and did not properly interpret the case insensitivity constraint. The fact that Grok got confused by a formal prompt overloaded with test scenarios is not encouraging. It seems obvious, at least to me, that without overloaded formality, the prospect of using AIs to write our systems is doomed to spectacular failure. After all, programming is formal, and humans require overloading.

Conclusion

In the end, I expect that the LLMs and AIs will improve to the point that long, formal specifications will be within their grasp. I also expect that we programmers will design formal and overloaded specification languages that the AIs can consume to produce complex systems.

And who will learn and apply these formalisms? Who will provide the ability to reason that the AIs lack? Programmers, of course. We programmers will become the lawyers of AI, responsible (as we always have been) to reason through and draw up the precise and accurate formal requirements and contracts for the applications our clients need.